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

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

Subhankar Pal
University of Michigan
Ann Arbor, USA
subh@umich.edu

Aporva Amarnath
University of Michigan
Ann Arbor, USA
aporvaa@umich.edu

Siying Feng
University of Michigan
Ann Arbor, USA
fengsy@umich.edu

Michael O'Boyle
University of Edinburgh
Edinburgh, UK
mob@inf.ed.ac.uk

Ronald Dreslinski
University of Michigan
Ann Arbor, USA
rdreslin@umich.edu

Christophe Dubach
McGill University
Montréal, Canada
christophe.dubach@mcgill.ca

ABSTRACT

Dynamic adaptation is a post-silicon optimization technique that adapts the hardware to workload phases. However, current adaptive approaches are oblivious to *implicit* phases that arise from operating on irregular data, such as sparse linear algebra operations. Implicit phases are short-lived and do not exhibit consistent behavior throughout execution. This calls for a high-accuracy, low overhead runtime mechanism for adaptation at a fine granularity. Moreover, adopting such techniques for reconfigurable manycore hardware, such as coarse-grained reconfigurable architectures (CGRAs), adds complexity due to synchronization and resource contention.

We propose a lightweight machine learning-based adaptive framework called SparseAdapt. It enables low-overhead control of configuration parameters to tailor the hardware to both implicit (data-driven) and explicit (code-driven) phase changes. SparseAdapt is implemented within the runtime of a recently-proposed CGRA called Transmuter, which has been shown to deliver high performance for irregular sparse operations. SparseAdapt can adapt configuration parameters such as resource sharing, cache capacities, prefetcher aggressiveness, and dynamic voltage-frequency scaling (DVFS). Moreover, it can operate under the constraints of either (i) high energy-efficiency (maximal GFLOPS/W), or (ii) high power-performance (maximal $GFLOPS^3/W$).

We evaluate SparseAdapt with sparse matrix-matrix and matrix-vector multiplication (SpMSpM and SpMSpV) routines across a suite of uniform random, power-law and real-world matrices, in addition to end-to-end evaluation on two graph algorithms. SparseAdapt achieves similar performance on SpMSpM as the largest static configuration, with 5.3× better energy-efficiency. Furthermore, on both performance and efficiency, SparseAdapt is at most within 13% of an Oracle that adapts the configuration of each phase with global knowledge of the entire program execution. Finally, SparseAdapt is able to outperform the state-of-the-art approach for runtime reconfiguration by up to 2.9× in terms of energy-efficiency.

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10.
https://doi.org/10.1145/3466752.3480134

CCS CONCEPTS

• **Computer systems organization** → **Reconfigurable computing**; • **Computing methodologies** → *Classification and regression trees*; • **Hardware** → Hardware accelerators.

KEYWORDS

reconfigurable accelerators, sparse linear algebra, energy-efficient computing, machine learning, predictive models

ACM Reference Format:
Subhankar Pal, Aporva Amarnath, Siying Feng, Michael O'Boyle, Ronald Dreslinski, and Christophe Dubach. 2021. SparseAdapt: Runtime Control for Sparse Linear Algebra on a Reconfigurable Accelerator. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3466752.3480134

1 INTRODUCTION

Sparse linear algebra operations are key components of a plethora of modern applications, from graph analytics to scientific computing and machine learning [4, 5, 9, 10, 23, 50, 55, 64, 65, 68–71]. Many graph analytics algorithms, such as breadth-first search, shortest path, *etc.* can be represented as sparse linear algebra operations, as encapsulated by the GraphBLAS framework [30]. Two important kernels belonging to BLAS levels 2 and 3, respectively, are sparse matrix – sparse matrix multiplication (SpMSpM) and sparse matrix – sparse vector multiplication (SpMSpV). These are highly inefficient on traditional computing platforms such as CPUs and GPUs, as they are heavily memory-bounded and thus bottlenecked on data movement rather than compute [40].

Recent work have led to a myriad of proposals on optimizing sparse computation through fixed-function accelerator designs [3, 40, 57, 58, 72]. While these demonstrate energy-efficiency improvements of the order of 100 of times over a GPU, there is an important trade-off in terms of loss of *flexibility*, *i.e.* such designs are only applicable to a few kernels. One class of solutions that propose to close this gap between flexibility and efficiency are coarse-grained reconfigurable architectures (CGRAs) [15, 25, 29, 39, 41, 49, 52, 62]. CGRAs incorporate word-granular operations to overcome the energy inefficiency of field-programmable gate arrays (FPGAs), while retaining programmability. They allow for hardware reconfiguration at the granularity of the processing element (PE) array, network fabric, or the memory subsystem. A key challenge, however, lies

with efficient mapping of sparse kernels onto CGRA hardware. Typically, the CGRA configuration is determined at compile-time by extracting a dataflow graph (DFG) from the application code, mapping it onto the PEs, and rewiring the on-chip memory and interconnect. There are two drawbacks of compile-time mapping.

- Real-world sparse datasets are seldom uniform, *e.g.* in graph analytics workloads, the input graphs are often clustered and follow power-law distributions [16, 33]. Thus, determining the best configuration requires intimate knowledge of the input, beyond its shape and number of nonzeros (NZEs), which is not feasible at compile-time without convoluted pre-processing.
- Compile-time optimizations fail if the dataset evolves over time, since no accurate estimations can be made for the distribution of non-zeros even with data pre-processing. This is common in the world of social networks, for instance, where connections between users form and break in real-time, which translates to dynamic changes in the underlying data structures [14, 21].

In order to tackle these challenges, we propose an adaptive runtime framework, SparseAdapt, that reconfigures a CGRA to adapt to evolving phases in sparse computation kernels. SparseAdapt establishes a feedback loop between the software (running on a host) and the CGRA hardware, that enables *fine-grained* introspection, *i.e.* the hardware exposes performance counters to the runtime software, which periodically collects this data to determine *when* and *how* to reconfigure the hardware, thus catering to transitions in both implicit (data-driven) and explicit (code-driven) phase changes. SparseAdapt is oblivious to the underlying program binary, and thus requires no programmer intervention.

We implement SparseAdapt as an extension to the runtime of a recently-proposed reconfigurable CGRA called Transmuter [41], however it is applicable to any CGRA system that exposes similar hooks from the hardware to the runtime. SparseAdapt is designed to be deployed on both cloud and edge scenarios, which have drastically different power-performance requirements. As such, SparseAdapt can operate in one of two optimization modes, an Energy-Efficient mode that optimizes for maximal giga floating-point operations executed per second per Watt (GFLOPS/W), and a Power-Performance mode that optimizes for maximal $GFLOPS^3/W$.

SparseAdapt is primarily evaluated on SpMSpM and SpMSpV in this work. However, it is applicable to arbitrary algorithms, since the technique is oblivious to the running software. We additionally demonstrate SparseAdapt on two popular graph algorithms, namely breadth-first search (BFS) and single-source shortest path (SSSP) [24, 63], which are representative irregular real-world workloads.

In summary, this work makes the following contributions:

- Identifies the existence of hardware reconfiguration opportunities associated with phase transitions *during* execution for sparse algebra routines, both due to (i) change in code, *i.e.* explicit phases, and (ii) evolving sparsity patterns, *i.e.* implicit phases.
- Proposes a framework called SparseAdapt that uses low-cost hardware performance monitoring to adapt to phase changes and reconfigure the underlying hardware configuration. SparseAdapt is integrated with the runtime of a recent reconfigurable accelerator, Transmuter [41].
- Proposes a predictive model based on an ensemble of decision trees, and a heuristic-based, reconfiguration cost-aware policy that allows for hardware reconfiguration at fine granularities ($\sim 500-5k$ floating-point instructions), while reducing the frequency of reconfiguration penalties.
- Demonstrates improvements in energy-efficiency and performance metrics on sparse linear algebra routines (SpMSpM and SpMSpV) and graph algorithms (BFS and SSSP). The evaluation is done across a suite of random, power-law and real-world matrices, and the proposed scheme is compared with various oracle adaptive control mechanisms.

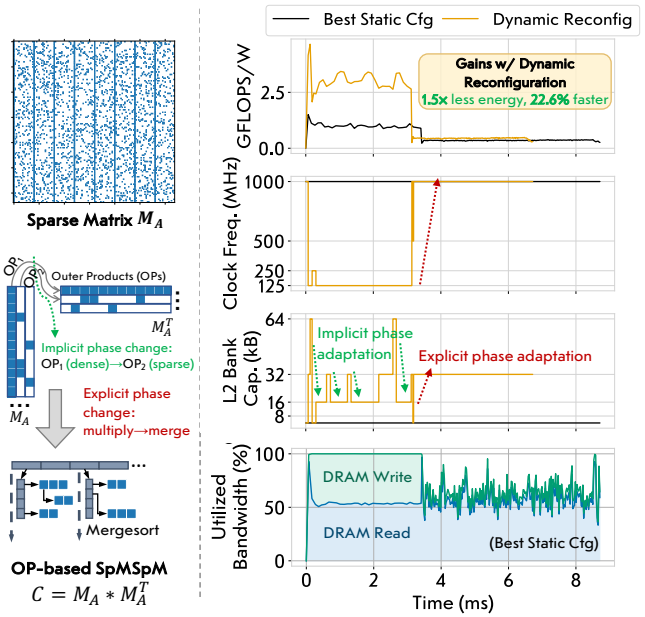


Figure 1: *Left.* Illustration of outer product (OP) SpMSpM showing implicit phase changes due to switch from dense to sparse outer products (column × row). *Right.* Execution timeline of OP-SpMSpM on a 128×128, 20% dense matrix and its transpose. Shown are the gains achieved using a dynamic scheme that adapts the hardware to these phase transitions.

In terms of evaluation, SparseAdapt achieves similar performance as the largest static configuration with 5.3× better energy-efficiency on SpMSpM. When compared to an Oracle scheme that exploits full knowledge of the entire program duration, SparseAdapt achieves within 13% performance and 5% efficiency in the Power-Performance optimization mode. Finally, in comparison to a prior adaptive control scheme by Dubach *et al.* [20], SparseAdapt achieves average gains of up to 2.9× and 2.8× in terms of energy-efficiency and performance, respectively.

2 MOTIVATION AND RELATED WORK

This section discusses the challenges and opportunities associated with dynamic reconfiguration, and prior work.

2.1 Existence of Implicit and Explicit Phases

Phases arise naturally during the execution of any non-trivial code. The obvious phase changes occur due to control flow from the program structure. We term these as *explicit* phase transitions. Explicit phase changes are relatively simple to detect, even at compile-time.

However, computation involving sparse datasets introduce another type of phase change owing to the irregularity in the data. Such phase changes may be correlated with instantaneous properties of the data, such as spatial locality. We define these phase transitions to be *implicit*. Real-world matrices have sparsity patterns that vary spatially across the matrix, thus introducing implicit phase changes during computation. Thus, a clever adaptation framework needs to incorporate runtime reconfigurability and cater to both implicit and explicit phase changes.

We illustrate the phenomenon of phase changes through the example of outer product (OP) based SpMSpM. We simulate the OP-SpMSpM algorithm (described in prior work [40, 42, 43]) on a tiled manycore architecture (details in Section 3) with two processing tiles. At a high level, OP-SpMSpM decomposes the computation into two explicit phases, namely *multiply* and *merge* (Figure 1 – left). We demonstrate this on a synthetic matrix generated with dense columns separating eight sparse strips to motivate our work. Similar structures of alternating dense and sparse row/column clusters exist in real-world datasets from graphs, optimization and economics problems, *etc.* [17]. We report the energy-efficiency (GFLOPS/W), instantaneous clock frequency, L2 cache bank size, and utilized memory bandwidth, with a dynamic reconfiguration scheme versus the best *static* hardware configuration in Figure 1. We first note the presence of the two explicit phases, corresponding to the multiply phase (ending at ∼ 3.5 ms) and merge phase of OP-SpMSpM. The dynamic control scheme observes the off-chip bandwidth utilization to be ∼100% during multiply (Figure 1 – bottom), and acts by applying DVFS to lower the clock speed and balance the compute-to-memory ratio of the system. This improves the multiply phase energy-efficiency over the baseline by ∼2×.

We further note the presence of implicit phases arising from the computation of dense columns with corresponding dense rows during the first phase (Figure 1 – left). These implicit phases are adapted to by adjusting the dynamic L2 cache capacity, based on observation of a combination of hardware counters (not shown in figure). Exploiting both types of phase changes leads to an overall speedup of 22% and energy savings of 1.5× over the static baseline.

In summary, this simple example illustrates the opportunities associated with dynamic reconfiguration, which we seek to exploit with our proposed framework, SparseAdapt.

2.2 Related Work

A few prior work have explored run-time adaptation for improving the efficiency of existing hardware architectures.

2.2.1 Adaptation for Traditional Hardware. Dubach *et al.* [19, 20] propose a maximum likelihood estimation (MLE) predictive model that adapts the sizes of microarchitectural structures in a single-threaded, out-of-order processor. CHARSTAR [51] uses a multilayer perceptron (MLP) model that accounts for the clock hierarchy and topology, and proposes a mechanism that jointly optimizes for both dynamic voltage-frequency scaling (DVFS) and clock-aware power gating. Both of these work are proposed only for single-core systems. Tarsa *et al.* [61] propose an adaptive CPU based on Intel Sky-Lake that uses random forests to dynamically adjust the issue width of a clustered core. These work explore hardware adaptation for traditional workloads that only have explicit phases. SparseAdapt, on the other hand, is designed for manycore hardware, and caters to workloads that have both explicit and implicit phases. Moreover, these prior techniques rely on SimPoint [45] for fast generation of offline profiling data. However, SimPoint fails when workloads have diverse implicit phases, arising from unstructured sparsity, throughout program execution. Our work instead considers end-to-end simulations, as no assumptions can be made about the nature of the input matrices.

Lukefahr *et al.* [36] introduce Composite Cores, that uses a linear regression based reactive online controller to switch execution between big and little μengines in a heterogeneous system. Flicker [46] is an adaptive multicore architecture designed to adapt to varying allocated power constraints. SOSA [18] is a resource manager that targets manycore systems and dynamic workloads using rule-based reinforcement learning (RL). The heavy weighted RL model requires additional acceleration, which the authors demonstrate on an FPGA. Sartor *et al.* [53] propose an MLP model coupled with a polymorphic VLIW processor that is trained at design-time and predicts at runtime. However, this framework predicts a static configuration for a kernel when the kernel is re-executed, whereas SparseAdapt adapts to the dynamic phase changes during run-time. Yukta [48] applies the structured singular value (SSV) control for EDP optimization on a multicore big.LITTLE system. Imes *et al.* [27, 28] propose a runtime system called POET that uses control theory and optimization techniques to minimize energy consumption while meeting soft real-time constraints, demonstrated on both big.LITTLE systems-on-chips (SoCs) and multicore server-class systems. However, these approaches do not explore configuration parameters associated with CGRAs, as well as workloads with varying implicit phases.

2.2.2 Adaptation for CGRAs. CGRAs have gained traction recently as they promise to deliver near-ASIC level performance with post-fabrication adaptation [35]. Earlier work have considered only compile-time CGRA reconfiguration for simplicity and ease-of-use [1, 13, 22, 49]. More recent work have alluded to the limitations with static adaptation, especially for irregular workloads like sparse linear algebra. Recent techniques have thus adopted dynamic scheduling and dataflow techniques to harness parallelism, however work on dynamic hardware reconfiguration that caters to both implicit and explicit phases in the workload has been comparatively lacking. Chen *et al.* [12] propose high-level synthesis techniques to map dynamic algorithms onto FPGAs. SparseAdapt, instead, generates configuration sequences at runtime to optimize for performance and energy on an existing CGRA. Dadu *et al.* [15] use network and PE-level decomposability to partition coarse resources for acceleration of different datatypes. In contrast, SparseAdapt operates by power-gating and using DVFS for performance and energy optimization, and tuning the uncore for performance. Nowatzki *et al.* [39] propose a CGRA where the control core and dispatcher are responsible for issuing work and enforcing dependencies (correctness). In SparseAdapt, the controller (host) instead performs heuristic-based resource management (performance/efficiency). Voitsechov *et al.* [66] propose a CGRA architecture that our framework can be adapted to, but is an orthogonal contribution.

2.2.3 Comparison with ProfileAdapt. The technique proposed in this paper is closely related to the work by Dubach *et al.* [20] (referred to as ProfileAdapt in this work) in terms of the offline training,

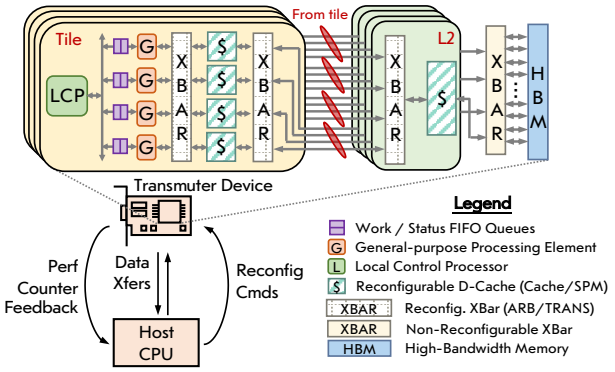


Figure 2: Simplified illustration of the Transmuter architecture and feedback loop between the host and Transmuter.

hardware telemetry and inference methodologies on the predictive model. However, there are several key differences. ProfileAdapt executes in the following order: detection of a new phase, execution of the new phase in a profiling configuration (where each reconfigurable parameter takes its maximum value) for a certain duration, hardware telemetry, inference on the predictive model, followed by reconfiguration and execution on the predicted configuration.

Since ProfileAdapt reconfigures at coarse phase boundaries, it cannot adapt to short-lived implicit phases due to the reconfiguration cost associated with switching to the profiling configuration. In contrast, our approach removes the need for a profiling configuration and instead directly feeds back the hardware configuration parameters as inputs to the predictive model. This enables us to reconfigure at a finer granularity with low-overhead, which is necessary to adapt to the implicit phases in sparse computation. Additionally, the evaluation in [20] relies on an external phase detection mechanism (*e.g.* SimPoint), whereas we assume that such a mechanism is not feasible for implicit phases. Our approach considers hardware reconfiguration at fixed epochs, guided by our predictive model and reconfiguration-cost aware heuristics.

3 HARDWARE DESIGN

We provide background on the Transmuter architecture [41], followed by the configuration parameters that we explore and implement in this work. We then discuss the performance counters that we implement, and the cost of reconfiguration.

3.1 Architectural Background

SparseAdapt is integrated with the runtime of a recently-proposed CGRA called Transmuter [41, 56, 67]. This section provides a brief overview of Transmuter, followed by a discussion of the configuration parameters that are considered. It concludes with some details on a set of proposed microarchitectural enhancements to implement the reconfigurability features.

Transmuter is a tiled CGRA design supporting runtime reconfiguration, as shown in Figure 2. It is composed of a sea of programmable, asynchronous general-purpose processing elements (GPEs), implemented as simple in-order cores. An $M{\times}N$ system consists of M processing tiles, where each tile has a group of N GPEs that are managed by a control plane processor called the local control processor (LCP). The GPEs within a tile are interfaced with an L1 cache-crossbar layer, and multiple tiles are connected via crossbars to an L2 cache layer. The L2 interfaces to multiple high-bandwidth memory (HBM) channels. Transmuter incorporates reconfigurability along the dimensions of on-chip memory type (cache, scratchpad memory – SPM, or FIFO+SPM), resource sharing (private or shared), and dataflow (demand-driven or spatial). These features are enabled using the flexible cache-crossbar hierarchy, composed of reconfigurable data-cache (R-DCache) banks and crossbars (R-XBars).

Table 1: Hardware configuration parameters that are tuned for the Transmuter design evaluated in this work.

Type	Parameter	Values Assumed	Count
Categorical	L1 R-DCache type†	Cache, SPM	2
	L1 sharing mode	Shared, private	2
	L2 sharing mode	Shared, private	2
Ordinal	L1 R-DCache bank capacity‡	4 kB → 64 kB : 2∗	5
	L2 R-DCache bank capacity	4 kB → 64 kB : 2∗	5
	System clock frequency	31.25 MHz → 1 GHz : 2∗	6
	Prefetcher aggressiveness	0 (off), 4, 8	3
Total Count			**3,600**

†Only this parameter is configured at compile-time. ‡ Not varied for SPM mode.

In this work, we enhance the Transmuter architecture and implement a feedback loop between a host CPU and the Transmuter device, as shown in Figure 2 (bottom). The host executes Python code and is responsible for offloading parallelizable kernels to Transmuter for accelerated execution. The first step associated with kernel dispatch consists of selecting a version of the code to execute on the target. This algorithmic selection is dependent on the properties of the input data, and the kernel itself. The next steps involve allocation of input and output buffers in the HBM, streaming data out, triggering Transmuter to start, waiting for it to finish *while executing runtime routines*, streaming data back in, and de-allocating memory. In this work, we assume shared physical memory between the host and Transmuter.

During execution, Transmuter sends performance counters to the host at continuous intervals (epochs). Using this information, the host makes a decision about when and how the architecture should be reconfigured. Actual reconfiguration is handled by dedicated blocks incorporated in the design.

3.2 Configuration Parameters

In this work, we consider seven hardware configuration parameters, namely, L1 R-DCache type, L1 R-DCache capacity, L2 R-DCache capacity, L1 sharing mode, L2 sharing mode, system clock frequency, and prefetcher aggressiveness. The values considered for these parameters are listed in Table 1. While much of prior work (Section 2.2) focuses on reconfiguration of the core microarchitecture, our work delves into reconfiguration of the interconnect, memory and dynamic voltage-frequency scaling (DVFS).

We next discuss the microarchitectural aspects and overhead of reconfiguring each selected hardware parameter.

3.2.1 Dynamic Voltage-Frequency Scaling (DVFS). We implement a simple clock divider composed of N flip-flops that enables generation of clocks with frequencies of $f/2, f/4, \ldots, f/2^N$, where f is the frequency of the system clock. For circuit-level simplicity, we

consider global DVFS, *i.e.* the same clock feeds into each microarchitectural block, as opposed to per-core or per-block DVFS. The overheads and switching times can be kept small through the use of on-chip regulators [31], or dual-voltage rail designs [37, 47] at the expense of a small area overhead.

In our 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:

$$\frac{f}{f_{target}} = \frac{\frac{(V_{DD}-V_t)^2}{V_{DD}}}{\frac{(V_{target}-V_t)^2}{V_{target}}}, V_{target} = \begin{cases} V_{target} & V_{target} \geq 1.3V_t \\ 1.3V_t & \text{otherwise} \end{cases}$$

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

DVFS allows for lowered dynamic power consumption, at the cost of some performance. However, the performance loss is negligible if the system is memory-bounded, which depends on the amount of compute resources, external memory bandwidth, and crucially, the application and its input data.

3.2.2 Cache Capacity. We re-implement each R-DCache bank as a set of sub-banks, *i.e.* each logical R-DCache bank is composed of a set of physical SRAM and tag arrays. There is a small combinatorial logic overhead to select between different set-index and tag bits, depending on the active cache capacity value. A larger cache allows for fewer misses, at the cost of increased latency and energy per access. Adaptable cache sizing has been explored in the past and shown to improve energy-efficiency in traditional architectures [60].

3.2.3 Sharing Mode. Transmuter is composed of swizzle-switch network based crossbars [54] that are augmented with crosspoint control units (XCUs). XCUs incorporate the ability to reconfigure the L1 and L2 memory between a shared and a private configuration, across the GPEs within a tile for the L1 and tiles for the L2.

The choice of sharing mode can have either a positive or negative effect on performance. On one hand, the shared R-XBar configuration incurs larger access latency, due to the overhead of arbitration between different requesters accessing the same resource(s), but allows for data sharing, which can lead to better hit-rates and improved reuse. On the other hand, the private mode offers a fixed, 1-cycle access latency, but privatizes the memory resource to the requester. Privatization can reduce cache pollution if the system encounters thrashing, but also causes duplication for shared data.

3.2.4 On-Chip Memory Type. We consider reconfiguration between two on-chip memory types, cache and scratchpad memory (SPM). SPM consumes lower power than an equivalent cache by power-gating the tag array and other unused logic. SPM is also faster when there is reuse across a set of arbitrary (but known) memory locations. It thereby takes advantage of a tailored replacement policy at the cost of additional instructions to orchestrate data. In contrast, a cache trades-off the flexibility of SPM by implicitly managing data, and thus generally outperforms for regular memory accesses or when there is low reuse.

Table 2: List of hardware performance counters in this work.

Hardware Block	Performance Counters
R-DCaches	Cache/SPM access throughput, *i.e.* number of accesses per unit time, cache occupancy, *i.e.* fraction of valid tags in the bank, overall miss rate, number of prefetches issued per cache access, current cache capacity.
R-XBars	Contention-to-Access Ratio, *i.e.* ratio of number of contentions across all output ports to the number of accesses through the crossbar.
LCP/GPE Cores	Floating-point instructions per cycle (including loads and stores), overall instructions per cycle (IPC), clock speed.
Memory Controller	Read and write memory bandwidth utilization, *i.e.* used bandwidth normalized to available bandwidth.

3.2.5 Prefetcher Aggressiveness. The L1 and L2 layers in Transmuter consist of a stride prefetcher based on a PC-based index table. We incorporate the ability to switch between different degrees of prefetching, *i.e.* the number of cachelines to prefetch ahead. The usefulness of this prefetcher is correlated with the amount of structure in the non-zeros within the input matrices. For highly unstructured data, turning off the prefetcher can lead to minimum performance degradation while saving power.

3.3 Performance Counters

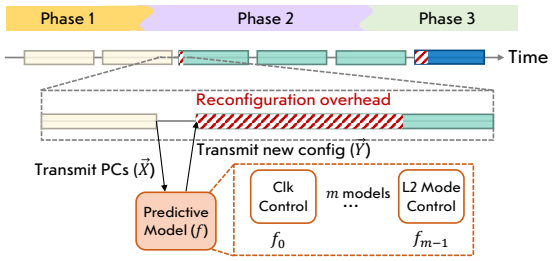
We implement low-overhead performance counters in the hardware that are reset after they are queried. These counters are listed in Table 2. The telemetry data is averaged both spatially (across all replicated hardware blocks) and temporally (normalized to the elapsed cycle count of the epoch) by the runtime. The runtime routine performs lightweight pre-processing upon receiving the telemetry data, such as normalization and feature set augmentation.

The performance counters are hand-picked based on architectural knowledge and correlation studies, such that we select those that have small cross-correlations and are oblivious to the code being executed. At the same time, they are designed with low hardware overhead considerations. The performance counters for GPE/LCP cores are available off-the-shelf. The remainder are constructed as simple saturating counters that poll on existing wires, with the exception of the crossbar contention-to-access-ratio for which dedicated wiring is used to detect contention for memory banks or a downstream crossbar channel. Overall, we estimate the storage overhead to be $<$ 1 kB for the evaluated 2×8 Transmuter system, and $\sim (42.5 \cdot M \cdot N)$ B for large M×N systems.

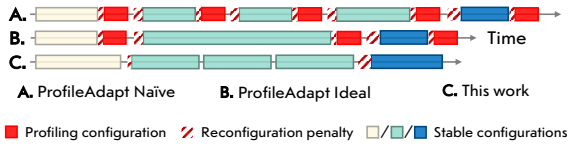
Query Mechanism. The performance counters in our implementation are memory-mapped and accessed by the host using a PCIe-like protocol. The counter data is queried and streamed out to DRAM, before being loaded by the host. These steps take place in the shadow of the workload executing on the device.

3.4 Cost of Telemetry and Reconfiguration

We estimate the decision-making and communication process to cost on the order of 50–100 host clock cycles. The greater runtime



(a) Overview of SparseAdapt illustrating how the predictive model is used to read performance counter (PC) feedback and predict the best configuration for the remainder of the program phase.



(b) Timing of SparseAdapt compared to ProfileAdapt considering A. fixed epochs, and B. epoch size equal to program phase size. SparseAdapt alleviates the overhead of switching to the profiling configuration. Energy benefits of avoiding the switch to the profiling configuration are not shown.

Figure 3: Overview of the functionality of SparseAdapt.

reconfiguration cost arises from the overhead to flush caches, as we pessimistically assume that the entire cache hierarchy is dirty prior to reconfiguration. The prediction problem is further complicated, because not all configuration changes incur the same cost, as not all configuration changes require cache flushing, or require local flushes only (*i.e.* from L1 to L2). As such, we introduce the following taxonomy of our configuration parameters.

- **Coarse-Grained.** Hardware parameters that require substantive change in the code running on the GPE and LCP cores, in addition to cache flushing. Both the memory type and dataflow configuration changes belong to this category.
- **Fine-Grained.** Parameters that require at most a cache flush, but do not impact the code executed on the cores.
- **Super Fine-Grained.** Parameters that incur a small, fixed reconfiguration cost and can be reconfigured without impacting the code or even necessitating a cache flush.

We assume in this work that the L1 memory type parameter is selected by the compiler and use SparseAdapt to predict for the remaining six parameters (Table 1). This is done to avoid the cost of checkpointing and rollback that would be incurred while switching between the cache and SPM modes for L1 memory. For fair comparisons in our evaluation, we compare against distinct static configurations that each perform best for the cache and SPM modes, respectively.

4 PREDICTIVE MODEL

We illustrate the functionality of the proposed SparseAdapt framework in Figure 3a. SparseAdapt partitions the program execution into multiple *epochs*. An epoch finishes when the number of floating-point operations (FP-ops) executed (inclusive of loads and stores), averaged across spatial hardware instances, exceeds a fixed value. We consider FP op-based epochs in this work, because SparseAdapt reacts to changes in the distribution of FP non-zeros in the inputs. Time-based epochs cause an imbalance in the number of reconfigurations across epochs that are executed with different clock speeds; op-based epochs count non-FP (*e.g.* bookkeeping) operations, which may differ for different NZEs; cycle-based epochs additionally add a dependency on the variable main memory access latency.

At the end of each epoch, a sequence of three operations is performed, namely, (i) hardware telemetry, (ii) inference using a predictive model, and (iii) reconfiguration of the hardware.

First, a set of performance counter registers in the Transmuter design are polled and their values are streamed through the off-chip interface to the host processor. Then, the predictive model is invoked on the host, and it is responsible for predicting the parameters of the hardware configuration that it deems to be the best for the next epoch of execution.

In contrast, the state-of-the-art ProfileAdapt, switches to a profiling configuration before the hardware telemetry is performed. It further assumes that the hardware has the ability to dynamically predict phases at runtime. This is a highly non-trivial task, particularly for real-world datasets in sparse computation, that produce implicit phase transitions during execution. We thus illustrate two versions of ProfileAdapt [20] in Figure 3b. **A.** represents the worst-case scenario that relies on reconfiguration at epoch-granularity when phases cannot be determined through other mechanisms. **B.** shows the execution timeline assuming phase changes could be known in advance, which is idealistic given the irregular nature of the sparse workload.

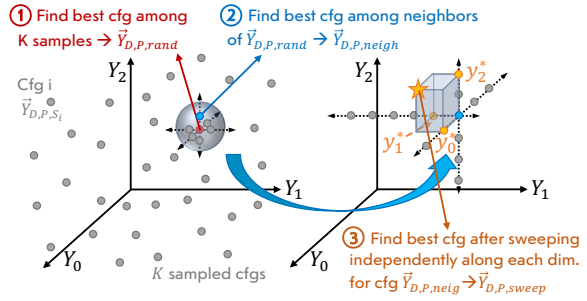
As we will see in the evaluation (Section 6.4), our approach is superior to both **A.** and **B.**, by virtue of eliminating the back-and-forth switch to the profiling configuration. While this may be of limited importance if the program phases are relatively stable and sufficiently long, it helps achieve significant savings for bursty program behavior consisting of unique implicit phases, such as with the SpMSpM example shown in the motivation (Figure 1). The rest of this section is dedicated to the predictive model.

4.1 Model Construction

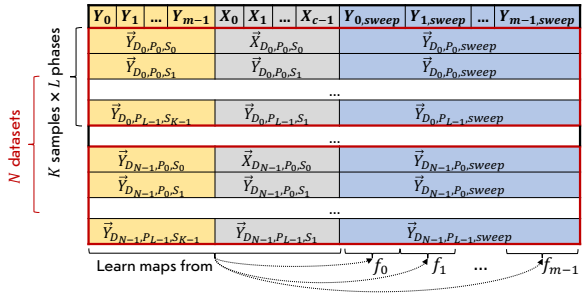
The predictive model (Figure 3a) can be formulated as a function $f : \vec{X} \rightarrow \vec{Y}$ where $\vec{X}$ is a tuple of selected performance counter values, and $\vec{Y}$ is the set of configuration parameter values. Similar to ProfileAdapt, we consider each configuration dimension $Y_i \; \forall \; i \in \vec{Y}$ to be conditionally independent given the values of performance counters $\vec{X}$, in order to simplify the model. The predictive model is thus an ensemble of independent functions $f_i, i \in M$ where M is the space of the configuration parameters (Table 1).

We describe the methodology we use to determine the "best" architectural configuration, where "best" refers to the one that achieves the highest GFLOPS/W (in Energy-Efficient mode) or GFLOPS^3/W (in Power-Performance mode). This is illustrated in Figure 4a, assuming three configuration parameters, $Y_0, Y_1 \cdots Y_{m-1}$ with $m = 3$, for ease of illustration. It involves the following steps:

(1) **Random Sampling.** We first sample K unique configurations from the space of *all* architectural configurations. Then, given a program phase P and input dataset D, we execute the code



(a) Simulations to find the "best" configuration are performed in three steps: 1) random sampling, 2) neighbor evaluation, and 3) dimension sweep.



(b) The training dataset uses both the current configuration tuple $\vec{Y}_{D,P,S}$ and the performance counters $\vec{X}_{D,P,S}$, given an input dataset D, program phase P, and sampled configuration S. The predictive model $\{f_0, f_1, \cdots, f_{m-1}\}$ learns the mapping of $\{\vec{Y}_{D,P,S}, \vec{X}_{D,P,S}\} \rightarrow \vec{Y}_{D,P,sweep}$.

Figure 4: Methodology of constructing the training dataset used for the predictive model in SparseAdapt.

on each of the K configurations. We record the "best" among these K configurations as $\vec{Y}_{D,P,rand}$.

(2) **Neighbor Evaluation.** We next evaluate the configurations in the m-dimensional hyper-sphere surrounding $\vec{Y}_{D,P,rand}$, and record the "best" one as $\vec{Y}_{D,P,neigh}$.

(3) **Dimension Sweep.** Finally, starting with $\vec{Y}_{D,P,neigh}$, we sweep each configuration dimension in isolation and record the "best" points as $\vec{y}_0^*$, $\vec{y}_1^*$ and $\vec{y}_2^*$ (orange dots in Figure 4a). Thus, given our conditional independence assumption, the point $\vec{Y}_{D,P,sweep} = \{\vec{y}_0^*, \vec{y}_1^*, \vec{y}_2^*\}$ (starred in figure) denotes the "best" configuration given D and P.

4.2 Dataset Construction and Training

Our approach differs from the ProfileAdapt approach [20] in how we construct the training dataset that is used for training our predictive model. *Our key insight is to use the values of configuration parameter of the last epoch as inputs to the predictive model.* This gives us access to vastly more amount of training data, since for each program phase P we can construct K examples containing performance counter inputs that should trigger reconfiguration to the "best" hardware configuration (Figure 4b). This approach also helps the model generalize and learn to predict from *any configuration* to the best configuration, rather than just from the profiling configuration to the best configuration.

4.3 Choice of Predictive Model

We pose the following requirements from an ideal model.

- **Accuracy.** The model should have high accuracy in predicting the best configuration parameters for the next epoch given the performance counter values of the current epoch.
- **Generalizability.** The model should generalize and predict for counter values that it has not been trained with.
- **Overhead.** The inference overhead should be sufficiently low such that the time cost of executing telemetry, inference, and reconfiguration $\ll$ the epoch size. This is necessary to enable fine-grained reconfiguration, in order to adapt to the short-lived implicit phases.

We experimented with four machine learning models, namely decision trees, random forests, linear regression, and logistic regression. We found similar inference accuracies between decision trees and random forests, whereas the linear and logistic regression models gave us poor accuracies. We thus selected decision trees (with pruning) as our predictive model, since they adhere to the aforementioned constraints the best.

4.4 Reconfiguration Cost-Aware Prediction

As discussed earlier, some architectural parameters incur a higher reconfiguration cost. While the super fine grained parameters incur relatively lower cost (Section 3.4), frequent reconfiguration for other parameters, such as cache capacity, can lead to performance degradation *even if the prediction is accurate.* In order to prevent the model from switching high-cost parameters too frequently, we add a degree of hysteresis through a set of heuristic-based schemes:

- **Conservative.** The predictor does not perform reconfiguration for a parameter that incurs more than a fixed cost.
- **Aggressive.** A scheme where the predictor always chooses to reconfigure based on its decision, regardless of the cost.
- **Hybrid.** For each configuration parameter, the predictor only reconfigures if the time cost of reconfiguring along that dimensions is within a certain percentage of the previous epoch's elapsed time. We consider this instead of an absolute threshold on the reconfiguration time, so as to penalize frequent bursts of reconfiguration within short periods (shorter epoch times), but allow for them when they occur occasionally (larger epoch times). The threshold is selected empirically based on our experiments.

5 EXPERIMENTAL SETUP

This section describes our training data collection and system modeling methodologies, followed by descriptions of the baseline schemes used for our evaluation.

5.1 Data Collection and Model Training

The ideal training dataset should produce a high variability in the program behavior in order to stimulate a wide range of performance counter values. For this purpose we select a set of sparse matrices with a broad range of input working set sizes, ranging from 1.5 kB

Table 3: Parameter sweeps to generate training data.

Kernel Name	Algorithm Variant	Matrix Dimension	Matrix Density	Memory Bandwidth (GB/s)
SpMSpM	Cache, SPM	$128 \rightarrow 1k : 2*$	$0.2 \rightarrow 13\% : 2*$	$0.01 \rightarrow 100 : 10*$
SpMSpV	Cache, SPM	$256 \rightarrow 8k : 2*$	$0.2 \rightarrow 13\% : 2*$	$0.01 \rightarrow 100 : 10*$

to 67 MB. We further vary the external memory bandwidth so as to stimulate the system with both memory-bound and compute-bound scenarios. Overall, we generate $\sim 360k$ training examples (total for two modes of operation) by sweeping the parameters in Table 3 and running them on ~ 285 discrete hardware configurations.

We consider uniform random datasets as our inputs, so that we can safely consider the entire program phase to have uniform behavior. Each training example is generated by running a program phase P until the program behavior stabilizes, terminating it, and sampling the performance counter values. For outer product based SpMSpM, there are two program phases, multiply and merge, while for SpMSpV the multiply and merge steps happen in tandem.

Predictive Model Training. We train a decision tree classifier for each of our configuration parameters. While a clear advantage of our choice of decision trees as the predictive model lies in its explainability, decision trees are prone to overfitting [8]. We thus train our decision trees using k-fold cross-validation [2] with $k = 3$, while sweeping the hyperparameters of `criterion`, `max_depth`, and `min_samples_leaf` using Python and Scikit-learn [44].

5.2 System Modeling

We used gem5 [6, 7] to model the Transmuter system based on the architectural specification in [41]. A power estimator is constructed using a combination of RTL synthesis reports for crossbars, Arm specification document for the cores, and CACTI [38] for the caches and SPM. The estimates are scaled to 14 nm across the evaluated systems. We made additional modifications to the simulator and power estimator to model the microarchitectural changes to Transmuter (discussed in Section 3.2).

For workload simulation, we do not use SimPoint in this work, as we evaluate for matrices with arbitrary sparsity structures (Section 2.2). This limits the maximum sizes of matrices we can simulate within a reasonable time, since we simulate each input-kernel combination with different hardware configurations. Due to these constraints, we consider a relatively small Transmuter system on which we exercise sparse linear algebra kernels with moderate dataset sizes which are much larger than the on-chip memory. Specifically, we consider a Transmuter system that has 8 GPEs (8 L1 cache banks) per tile, and 2 tiles (2 L2 cache banks). We also assume a reduced off-chip memory bandwidth of 1 GB/s in order to maintain similar compute-to-memory ratio as the full system in [41].

Reconfiguration Cost. Following the taxonomy in Section 3.4, we model variable penalty for reconfiguration of each parameter. Changes to the super fine grained parameters are assigned a fixed cost of 100 cycles. An increase in cache capacity also incurs this fixed cost, as our cache implementation (Section 3.2.2) allows for it.

For reconfiguration of the fine-grained parameters, we pessimistically assume that all the cache lines are dirty. We thus assign a penalty equal to the time to flush all the R-DCache banks in a layer to the next level of hierarchy, *i.e.* L1 to L2 ($100–961k$ cycles, up to 157 µJ energy), and L2 to main memory ($100–122k$ cycles, up to 22 µJ energy) with 1 GB/s off-chip memory bandwidth. In reality, these overheads are expected to be lower due to fewer dirty lines, since only the partial products and bookkeeping data structures incur read-modify-write operations whereas the rest can be written directly to main memory. The host selects the optimal clock speed for cache flushing based on the mode of operation, *i.e.* Energy-Efficient or Power-Performance, based upon a lookup table that is indexed by the operational mode, L1 capacity per bank, and L2 capacity per bank. A significant amount of energy savings is obtained by power-gating the cores, ICaches, work and status queues, and the synchronization SPM while caches are being flushed. Finally, the coarse-grained parameter decision is assumed to be made at compile-time, and so does not impact the runtime.

Table 4: Specifications of baseline Transmuter hardware.

Config Name	L1 Bank Size (kB)	L1 Mode	L2 Bank Size (kB)	L2 Mode	Clock (MHz)	Pref. Agg.
Baseline	4	Shared	4	Shared	1000	4
Best Avg (L1: cache)	4	Private	4	Shared	1000	0
Best Avg (L1: SPM)	4	Private	32	Private	500	8
Maximum	64	Shared	64	Shared	1000	8

Table 5: Properties of matrices used for evaluation: synthetic (top) and real-world [17, 34] (bottom).

ID	U1	U2	U3	P1	P2	P3
Type	Uniform	Uniform	Uniform	Power-Law	Power-Law	Power-Law
Dim.	8,192	8,192	8,192	8,192	8,192	8,192
NNZ	25,000	50,000	100,000	25,000	50,000	100,000

ID	Matrix Name Matrix Dimension, NNZ Application Domain	Plot	ID	Matrix Name Matrix Dimension, NNZ Application Domain	Plot
R01	California (9.7K, 16.2K) (Directed Graph)		R09	EX3 (1.8K, 52.7K) (Comp. Fluid Dyn.)	
R02	Si2 (0.8K, 17.8K) (Quant. Chemistry)		R10	Oregon-1 (11.5K,46.8K) (Undirected Graph)	
R03	bayer09 (3.1K, 11.8K) (Chemical Simulation)		R11	as-22july06 (23.0K, 96.9K) (Undirected Graph)	
R04	bcsstk08 (1.1K, 13.0K) (Structural Problem)		R12	crack (10.2K, 60.8K) (2D/3D Problem)	
R05	coater1 (1.3K, 19.5K) (Comp. Fluid Dyn.)		R13	kineticBatchReactor_3 (5.1K, 53.2K) (Optimal Control)	
R06	gemat12 (4.9K, 33.0K) (Power Network)		R14	nopoly (10.8K, 70.8K) (Undirected Graph)	
R07	p2p-Gnutella08 (6.3K, 20.8K) (Directed Graph)		R15	soc-sign-bitcoin-otc (5.9K, 35.6K) (Directed Graph)	
R08	spaceStation_11 (1.4K, 19.0K) (Optimal Control)		R16	wiki-Vote_11 (8.3K, 103.7K) (Directed Graph)	

5.3 Comparison Points

We consider several comparison points for evaluating the improvements achieved using SparseAdapt:

- **Baseline.** A non-reconfiguring Transmuter system that achieves the best average metric value across the broad set of applications (dense and sparse) evaluated in the prior work [41].

- **Best Avg.** A non-reconfiguring system that achieves the best average metric for the SpMSpM and SpMSpV kernels on the datasets considered in this work.
- **Max Cfg.** A non-reconfiguring system that has maximum values for each ordinal configuration parameter, and that uses shared caches in both the L1 and L2.
- **Ideal Static.** A non-reconfiguring system (from our sampled space) that achieves the best average metric for the given program and dataset.
- **Ideal Greedy.** A system that dynamically reconfigures during execution of the routine. It greedily selects the configuration that is optimal *for the next epoch*.
- **Oracle.** A system that dynamically reconfigures assuming full knowledge of the entire program execution. It selects the sequence of configuration changes that result in the maximum average metric value. We model this as a dynamic programming problem and solve it using a modified Dijkstra algorithm to derive the globally optimal sequence for the sampled configurations.

We note that Ideal Static, Ideal Greedy and Oracle are configurations that cannot be determined at runtime, and thus are hypothetical systems used for upper-bound studies in this work. All the static configurations are specified in Table 4.

5.4 Choice of Dataset and Parameters

We evaluate SparseAdapt on SpMSpM and SpMSpV kernels using a variety of input datasets. While SparseAdapt can work with other algorithms, such as inner product with compression [26], we limit our evaluation to OP-based sparse matrix multiplication as it has been shown to be superior for the density levels considered in this work (see Section 8.1 in [41]). Without loss of generality, we perform our evaluation on square matrices, stored in compressed sparse column (CSC) for Matrix A and compressed sparse row (CSR) for Matrix B (or as an array of index-value tuples for vector B).

Synthetic Dataset. We generate uniform-random matrices using the SciPy library and for power-law matrices, we use the R-MAT generator [11] with parameters $A = C = 0.1$ and $B = 0.4$. The properties of these matrices are listed in Table 5 (top). They are selected to show trends across increasing NNZ count with a fixed dimension (U1 to U3 and P1 to P3).

Real-World Dataset. We derive real-world matrices for our evaluation from two popular sparse matrix collections, namely SuiteSparse [17] and Stanford's SNAP [34]. The properties of each of these matrices is described in Table 5 (bottom).

We evaluate SpMSpM for matrices R01-R08, and SpMSpV using R09-R16. We select comparatively more modest matrix sizes for SpMSpM since it is computationally more expensive to simulate than SpMSpV. Sizes of this order provided us a balanced trade-off with the resources required to perform cycle-accurate simulations.

We use a small epoch size of 500 FP-ops for SpMSpV in order to capture implicit phases at runtime. This was determined based on a sweep of epoch size from 250-4k FP-ops across a set of representative matrices. For the same epoch size, SpMSpM generates significantly more performance counter data than SpMSpV. In order to consume tractable amount of data with the given resources, we use a larger epoch size of 5k FP-ops for SpMSpM. However, we note that SparseAdapt itself is not limited to any specific workload or epoch size, and the benefits reported in this section are representative of larger datasets. Finally, unless otherwise noted, we assign the conservative policy for SpMSpM and hybrid policy with 40% tolerance for SpMSpV, as these gave us the best results across a set of sweep studies on the real-world datasets.

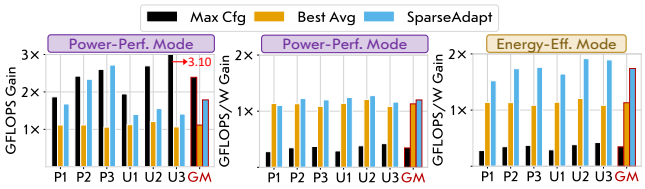


Figure 5: Gains over Baseline for SpMSpV on synthetic matrices for Power-Performance (left, middle) and Energy-Efficient (right) modes for L1 as cache.

6 EVALUATION

We evaluated our proposed framework first against the Baseline, Max Cfg and Best Avg configurations, followed by upper-bound studies. We then present insights into our predictive model. This is followed by comparison against ProfileAdapt [20]. We conclude with studies on the impact of SparseAdapt policies and the gains achieved across memory bandwidth sweeps.

6.1 Comparison with Standard Configurations

We discuss our analysis of improvements over the Baseline, Max Cfg and Best Avg systems here.

6.1.1 Analysis on Synthetic Dataset. We report evaluation of SpMSpV on our synthetic dataset against a uniform-random vector of density 50%. We omit our analysis of SpMSpM on the synthetic dataset, for brevity.

Power-Performance Mode. Figure 5 shows the performance in GFLOPS (left) and energy efficiency in GFLOPS/W (middle) of Baseline, Best Avg, Max Cfg and SparseAdapt. In this mode, SparseAdapt delivers average performance gains of 1.8× over Baseline. In this mode, the energy efficiency achieved is 3.5× better than Max Cfg, while the average performance remains within 34%. Although SparseAdapt achieves only 6% better efficiency than Best Avg, it delivers 1.6× better performance.

Energy-Efficient Mode. As seen in Figure 5 (right), SparseAdapt achieves an average energy-efficiency gain of 1.5–1.9× over the Baseline. The gains for both the power-law and uniform random matrices are observed to roughly saturate with increasing the density, as SparseAdapt follows the Oracle decisions more closely, particularly for the clock frequency and L2 mode. The Max Cfg configuration is 2.9× less energy efficient than Baseline despite being faster. In comparison, the Best Avg achieves 1.1× over the Baseline.

6.1.2 Analysis on Real-World Dataset. We present our analysis of SpMSpM on matrices R01-R08 in Table 5. Each matrix is multiplied with its transpose, *i.e.* $C = A \cdot A^T$, where A is the input matrix and C is the result matrix.

Power-Performance Mode. The gains for this mode are shown in Figure 6. The scope of performance improvements over Baseline is smaller than in the case of SpMSpV (for our synthetic dataset),

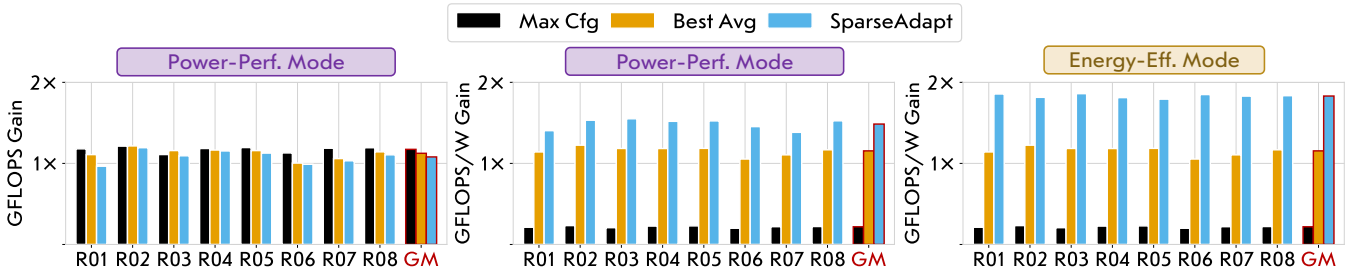
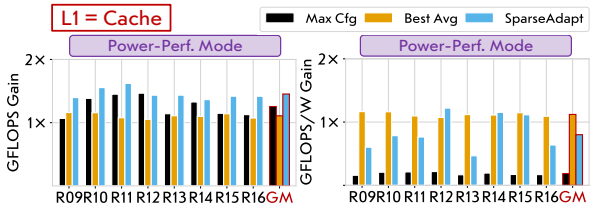
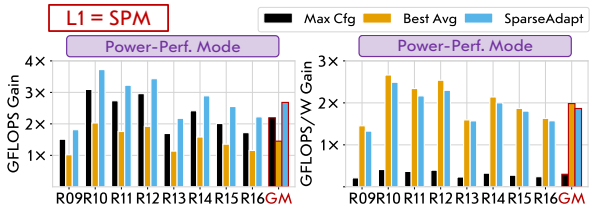


Figure 6: Improvements over Baseline for SpMSpM on real-world matrices in Power-Performance (left, middle) and Energy-Efficient (right) modes for L1 as cache.



(a) L1 memory banks configured as cache.



(b) L1 memory banks configured as scratchpad.

Figure 7: Improvements over Baseline for SpMSpV on real-world matrices in Power-Performance mode.

and SparseAdapt achieves similar performance as Best Avg (within 8% of Max Cfg). However, SparseAdapt delivers this performance at 1.3× less energy compared to Best Avg, and is 5.3× more efficient than Max Cfg.

Energy-Efficient Mode. Figure 6 (right) shows the efficiency gains with SparseAdapt, which are significantly higher (1.8×) than Baseline, and is better than Best Avg by (1.6×).

6.1.3 Analysis on Graph Algorithms. We implement two popular graph algorithms, namely BFS and SSSP. Our implementations map vertex programs to iterative SpMSpV operations, similar to GraphMat [59]. The end-to-end improvements in traversed edges per second (TEPS) per Watt are reported in Table 6, showing that SparseAdapt delivers gains of up to 1.5× over Baseline (1.3× over Best Avg). The largest gains are observed on graphs with highly power-law behavior, for instance, R10, R11 and R14. In contrast, the scope of improvement over Best Avg is small for R09, as it consists of local connections only and thus the non-zeros are distributed roughly uniformly along the diagonal (Table 5).

6.1.4 Analysis with Cache/Scratchpad L1 Memory. As noted in Section 3.4, we assume that the choice of L1 mode (cache or SPM) is made at compile-time. We thus have two configurations for the two L1 modes, listed in Table 4. The results for Power-Performance mode on SpMSpV for our real-world dataset (R09-R16) are shown in Figure 7. The performance gains are larger for L1 SPM (1.9× over Best Avg) compared to L1 cache (1.3× over Best Avg). This is 1.2× better than Max Cfg for cache and 1.2× better than Max Cfg for SPM, while being 4.3× and 6.2× more energy-efficient for the two L1 modes, respectively. The cache mode benefits are 1.5× better over Baseline, albeit with 20% greater energy due to the predictor selecting large L2 cache sizes.

Table 6: TEPS per Watt gains over Baseline for two graph algorithms in Energy-Efficient mode with L1 as cache.

		R09	R10	R11	R12	R13	R14	R15	R16	GM
BFS	Best Avg	1.26	1.14	1.11	1.07	1.17	1.13	1.17	1.24	**1.16**
	SparseAdapt	1.28	1.32	1.44	1.34	1.15	1.40	1.27	1.27	**1.31**
SSSP	Best Avg	1.21	1.13	1.10	1.03	1.13	1.05	1.15	1.18	**1.12**
	SparseAdapt	1.21	1.43	1.45	1.27	1.17	1.31	1.29	1.21	**1.29**

6.1.5 Insights from Configuration Choices. We observe that the model applies DVFS based on the bandwidth requirement of the explicit phase (Section 2.1), and selects faster clocks when there is greater data locality. The L1 size choice is correlated to the cache occupancy, however the number of reconfigurations are curbed based on our hybrid policy for fine-grained knobs (Section 4.4). In comparison, the prefetcher aggressiveness and L2 capacity are reconfigured more often, in response to implicit phases.

The L1 mode choice is highly data-dependent for SpMSpV; for SpMSpM, we observe the multiply phase to be amenable to shared L1 while merge performs better on private L1, which are consistent with the description in prior work [40]. When the L1 is configured as SPM, the model selects larger L2 sizes to accommodate auxiliary data structures (those not mapped to SPM) and spill variables. Finally, the model shows preference for larger L1 and L2 capacities for the Power-Performance mode, and for smaller sizes in the Energy-Efficient mode.

6.2 Comparison against Ideal and Oracle

We perform upper-bound studies to evaluate SparseAdapt and compare it with Ideal Static, Ideal Greedy and Oracle. Figure 8 shows the results for SpMSpM on real-world matrices R01-R08 (L1 as cache).

The gaps between each of these and SparseAdapt convey different insights; Ideal Static represents the gains achievable if we had an ideal *compile-time* predictor that can select the best average configuration. Ideal Greedy represents our SparseAdapt approach if the predictive model was ideal, and Oracle shows the

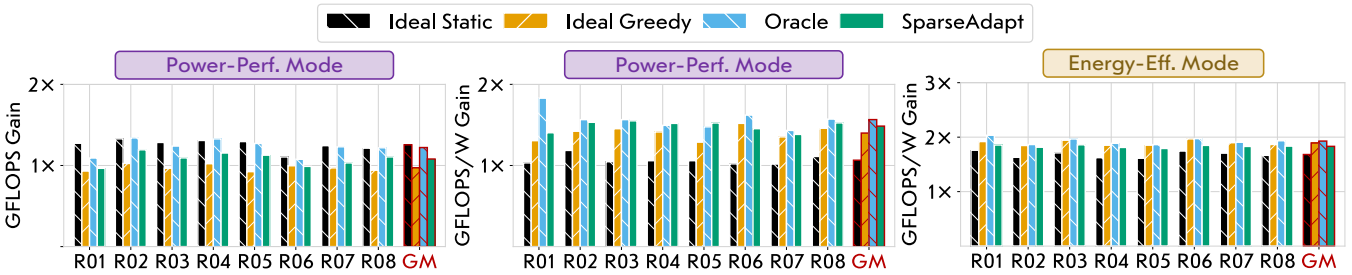


Figure 8: Comparison against Ideal Static, Ideal Greedy and Oracle for SpMSpM on real-world matrices in Power-Performance (left, middle) and Energy-Efficient (right) modes for L1 as cache. Gains are shown compared to Baseline.

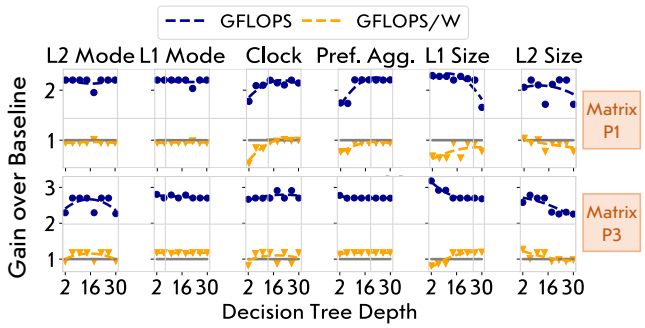


Figure 9: Effect of complexity of predictive model on the gains with SparseAdapt in Power-Performance mode for SpMSpV with L1 as cache.

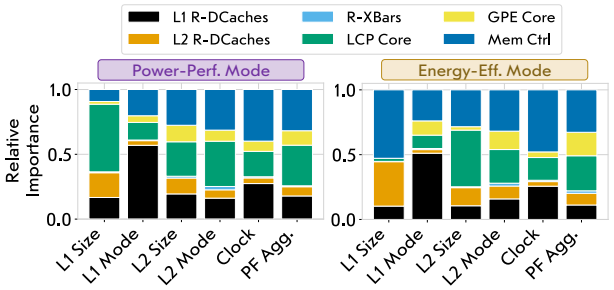


Figure 10: Relative importance of each class of performance counters for each trained model (x-axis) in Power-Performance (left) and Energy-Efficient (right) modes with L1 as cache.

gains achievable if the predictor had full knowledge of the future. While the performance gains are comparable between Oracle over Ideal Static (Power-Performance mode), we note the high scope of improvements with dynamic reconfiguration (1.3–1.8×) for GFLOPS/W. Compared to Oracle, SparseAdapt is within 13% performance for Power-Performance mode, and just 5% efficiency for both the modes. SparseAdapt achieves an energy efficiency within 3% of Ideal Greedy (Energy-Efficient mode), but interestingly the gains are 11% and 6% *better* for performance and efficiency, respectively, in Power-Performance mode. This is made possible by the policies introduced in Section 4.4, which generate inertia toward frequent reconfiguration changes along costly dimensions. When disabled, *i.e.* for the Aggressive scheme (not shown), the gains drop to 9% worse compared to Ideal Greedy for performance and energy efficiency, respectively.

Finally, for SpMSpV operation in the Power-Performance mode, SparseAdapt has 1% better (4% worse) GFLOPS and is within 56% (15%) in terms of GFLOPS/W compared to Oracle, for L1 as cache (SPM). We omit detailed analyses due to space constraints.

6.3 Analysis of Model and Features

We discuss insights from our analysis of hyperparameter exploration and feature selection for our predictive model.

6.3.1 Effect of Model Complexity. As decision trees tend to overfit, it is important to regularize them at training. We explore this phenomenon by training decision trees with depths ranging from $2 \rightarrow 26 : 4*$. We analyze the gains achieved while independently varying the depth of the tree corresponding to each of the configuration parameters one-at-a-time, while using the original trees for the remaining. Here, original trees refer to the trees trained using the methodology in Section 5. The improvements over the Baseline in Power-Performance mode, for the case of SpMSpV with two matrices, P1 and P3, with a 50% dense vector, are shown in Figure 9. Given that our Power-Performance mode gives greater importance to performance optimization, GFLOPS is more sensitive to model complexity compared to GFLOPS/W.

6.3.2 Feature Importance. Scikit-learn computes feature importance as the total reduction of criterion (*i.e.* function to measure the quality of a split in the tree) brought by that feature, also known as Gini importance. The feature importances of our performance counters in determining the decisions of the learned predictive model are shown in Figure 10. The counters are grouped into categories for ease of illustration. Overall, we observe that counters probing the L1 R-DCache block and the memory controller are the most important across the models for each configuration parameter. Interestingly, the LCP counters (IPC and FP-IPC) are given more importance than GPE counters in the trained models for most cases. This can be attributed to the fact that because LCPs are responsible for scheduling work and load-balancing, they have a "global" view of the activity within each tile, compared to any individual GPE core. Overall, this analysis is useful to reason about which counters are the most critical to have in order to balance prediction accuracy with hardware requirements.

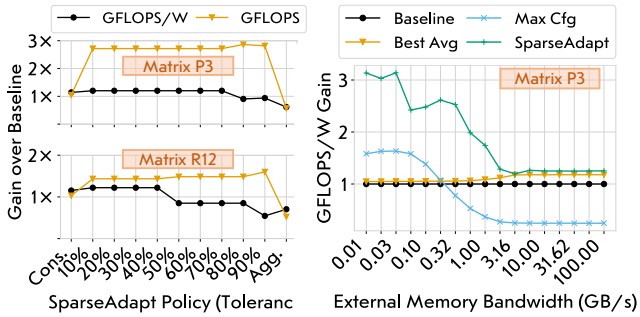


Figure 11: *Left.* Gains with different SparseAdapt policies (Power-Performance mode). *Right.* Impact of sweeping external memory bandwidth in Energy-Efficient mode. Both are for SpMSpV with L1 as cache.

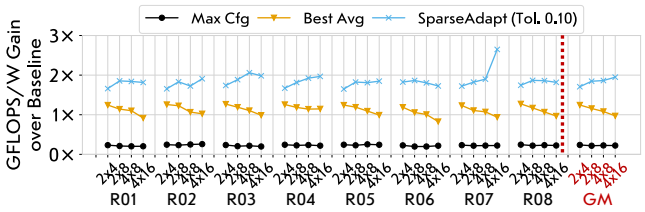


Figure 12: GFLOPS/W gains in Energy-Efficient mode for SpMSpM (R01-R08; L1 as cache) while varying the tile and GPEs per tile counts (fixed 1 GB/s bandwidth).

6.4 Comparison with ProfileAdapt Scheme

We compare SparseAdapt with a prior runtime reconfiguration scheme, ProfileAdapt [20]. ProfileAdapt triggers a reconfiguration into a "profiling" configuration at every epoch (naïve) or phase boundary (ideal) as illustrated in Figure 3, which can lead to large overheads especially for fine-grained phases. We implement ProfileAdapt by adding the cost to reconfigure to and from the profiling configuration into our Oracle sequence for each epoch (naïve), and for epochs following which there is a change in configuration (ideal). Furthermore, it would be unfair to compare ProfileAdapt and SparseAdapt at the same epoch size, since ProfileAdapt is designed to work with much larger epoch sizes. We, therefore, perform an epoch size sweep and select operational points for ProfileAdapt where the metric-of-interest is maximum, which is $6k$ FLOPS for the Power-Performance mode and $5k$ FLOPS for the Power-Performance mode. This is compared with SparseAdapt considering the chosen epoch sizes in Section 5.4.

We evaluate the SparseAdapt scheme against ProfileAdapt for SpMSpV in cache mode on the real-world dataset. Compared with the naïve ProfileAdapt scheme, which switches to the profiling configuration at each epoch, SparseAdapt achieves significant improvements of 2.8× and 2.0× in GFLOPS and GFLOPS/W, respectively for the Power-Performance mode, and 2.9× gain in GFLOPS/W for the Energy-Efficient mode. The ideal ProfileAdapt relies on a SimPoint based external phase detection mechanism which is unrealistic for workloads with implicit phases. Despite this idealistic assumption, SparseAdapt with its ability to efficiently reconfigure at implicit phase boundaries achieves GFLOPS/W gains of 1.7× and 1.1× in Power-Performance mode, and 2.4× in Energy-Efficient mode.

6.5 Effect of Parameter Sweeps

Cost-Aware Reconfiguration Policies. In Section 4.4, we introduced our conservative, aggressive, and hybrid prediction policies for reconfiguration-cost aware prediction. We evaluate the efficacy of each of these on SpMSpV execution (epoch size 500) on representative matrices from the synthetic and real-world datasets, P3 and R12. The results are shown in Figure 11 (left). The conservative and hybrid schemes with small tolerance for reconfiguration penalty overly restrict the system from reconfiguring across implicit phases during execution, resulting in limited gains. As the tolerance is increased, the reconfiguration cost starts dominating over the benefit achieved from switching to the new configuration. The ideal tolerance values are observed to be between 10-40% across most of our inputs, given this epoch size and kernel.

Memory Bandwidth. SparseAdapt can be deployed without retraining in scenarios where there is sharing of bandwidth across concurrent kernel executions, or if the device is interfaced to a different type of main memory. We study the impact of these scenarios by sweeping the external memory bandwidth. Figure 11 (right) shows the energy-efficiency gains with SparseAdapt (in Energy-Efficient mode) for SpMSpV. The trend of gains deviates from a smooth curve for smaller bandwidth values, because the reconfiguration cost increases, thus increasing the impact of discrete parameter choices. When the system is memory-bounded, as is generally true for sparse computation, SparseAdapt achieves large energy-efficiency gains of >3× over both Baseline and Best Avg. It selects smaller cache sizes and slower clocks to recover both static and dynamic energy, with negligible performance hit. Even at the other end of compute-boundedness, it is 1.1× better than Best Avg.

System Size. Figure 12 shows our gains over Baseline when scaling (*i*) the number of tiles and L2 banks (M), and (*ii*) the number of GPEs and L1 banks per tile (N), for an $M{\times}N$ system. We observe mean GFLOPS/W gains of 1.7-2.0× across the four systems, obtained using the predictive model trained for a 2×8 system (no re-training) and an epoch size of $5k$. As the system size grows, the benefits with DVFS dominate, particularly for the multiply phase of OP-SpMSpM. These gains demonstrate the scalability of our framework.

7 DISCUSSION

Adaptation to Different Hardware and Algorithms. This paper evaluates the SparseAdapt framework for sparse linear algebra operations. However, SparseAdapt is designed for *programmable* CGRAs (Transmuter in this work) and thus can be extended to additional kernels beyond SparseBLAS. This would simply require collection of additional training data using simulations for the new kernels, and re-training the predictive model. Since our performance counters are workload-agnostic, we expect no need for additional counters. Additionally, our approach can be adapted to any hardware with knobs for resource-sharing control, bank-level power gating, and so on.

While the proposed technique is evaluated with irregular workloads, it is applicable to regular applications as well. Our offline

analysis, however, shows minor differences (< 5%) between Ideal Static and Oracle for two regular kernels, GeMM and Conv [41]. We thus conclude that a dynamic control mechanism, while useful, may be an overkill if targeted exclusively for regular applications.

Bridging the Gap with Oracle. Ideal Greedy is a fundamental upper bound for the SparseAdapt approach in the Aggressive mode of operation. However, there is additional scope for improvement even between Oracle and Ideal Greedy in Section 6.2. An extension to SparseAdapt will explore using telemetry data from multiple past epochs to learn a history-based pattern of program execution, borrowing ideas from branch prediction and prefetching.

Memory Mode Reconfiguration. We assume that the choice of L1 mode (cache or SPM) is made at compile-time, since this determines the version of code executed. This leaves out some scope for optimization when different parts of the program show amenability to a cache or SPM. Dynamic cache-to-SPM reconfiguration can be enabled using existing hardware techniques, such as Stash [32], that map sections of the SPM to global memory.

8 CONCLUSION

This work proposed a dynamic control scheme called SparseAdapt that targets changes in phase due to evolving properties of the data in irregular workloads (implicit), in addition to those due to change in code (explicit). This requires fine-grained reconfiguration with low overhead, for which SparseAdapt uses runtime telemetry of hardware performance counters. SparseAdapt uses an ensemble of decision trees and a set of reconfiguration cost-aware heuristics to make decisions about configuration parameters of the hardware.

We evaluated SparseAdapt on key SparseBLAS kernels (SpMSpM and SpMSpV) in both Energy-Efficient and Power-Performance modes of operation, that predict for the best values of GFLOPS/W and GFLOPS^3/W, respectively. Our analysis on SpMSpM with real-world datasets show performance and energy-efficiency improvements of 1.1× and 1.5× over the baseline static configuration (similar performance as Max Cfg with 5.3× better efficiency) in Power-Performance mode, and 1.7× better efficiency in Energy-Efficient mode. Finally, the SparseAdapt technique achieves average gains of 1.7–2.8× in performance and 1.1–2.0× in energy-efficiency over a state-of-the-art control scheme for runtime adaptation.

ACKNOWLEDGMENTS

The material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7864. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government. We also acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants Program [grant RGPIN-2020-05889], and the Canada CIFAR AI Chairs Program.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact evaluation demonstrates the performance and energy efficiency benefits of using SparseAdapt across different sparse kernels and datasets on the Transmuter architecture, in addition to the scalability of SparseAdapt across different reconfiguration-aware policy and memory bandwidth values.

The artifact is packaged as a Docker image and contains: (i) gem5 model to simulate Transmuter, (ii) application source code for SpMSpM and SpMSpV on Transmuter, (iii) dataset used for training SparseAdapt's predictive models and pre-trained models, (iv) wrapper scripts to run the simulator using the various predictive schemes. Unfortunately, we could not make the artifact publicly available. For the artifact evaluation effort, we gave the evaluator(s) access to a private GitHub repository containing the infrastructure necessary to reproduce the results in the rest of this section.

A.2 Artifact Check-List (Meta-Information)

- **Algorithm:** SparseAdapt uses decision trees to predict the best set of configuration parameters given a set of performance counter data. Additionally, hysteresis schemes discussed in Section 4.4 are applied.
- **Program:** Outer product based SpMSpM and SpMSpV run on simulated hardware using gem5.
- **Compilation:** g++ 5.5.0, arm-linux-gnueabihf-g++ 5.4.0, Python 3.7.
- **Binary:** Binaries for SpMSpM and SpMSpV are generated using the arm-g++ compiler.
- **Model:** The proposed predictive model consists of decision tree models, one for each configuration parameter (Section 3.2).
- **Data set:** The training dataset is generated using the methodology in Section 4.2. Input datasets are obtained as mentioned in Section 5.4.
- **Run-time environment:** Ubuntu 16.04; Docker image provided.
- **Hardware:** While the artifact was developed on a 16-core Intel Xeon system, it can be evaluated on any commodity multiprocessor hardware.
- **Execution:** Bash and Python scripts are included for execution.
- **Metrics:** GFLOPS, GFLOPS/W gains over a non-reconfiguring system.
- **Output:** Terminal logs showing deviation of reproduced results with reported results; tarballs containing raw CSV results and PDF plots.
- **Experiments:** *Figures 5, 6 and 7*: comparison of SparseAdapt against Baseline, Max Cfg and Best Avg for SpMSpV with synthetic dataset (L1 as cache), SpMSpM with real-world dataset (L1 as cache), and SpMSpV with real-world dataset (L1 as cache and SPM). *Section 6.4*: comparison of SparseAdapt against ProfileAdapt for SpMSpV with L1 as cache. *Figure 11*: comparison with Baseline across (a) a sweep of SparseAdapt policies, and (b) different system memory bandwidth parameters.
- **How much disk space required (approximately)?:** Roughly 50-100 GB: 10 GB for the Docker image, and the remainder as temporary storage of gem5 build and simulation data depending on number of parallel runs.
- **How much time is needed to prepare workflow (approximately)?:** Less than an hour to install and familiarize oneself with Docker, followed by building the Docker image (20 mins with 16 cores).
- **How much time is needed to complete experiments (approximately)?:** Roughly 14 hours on a 16-core system with 64 GB of DRAM.
- **Publicly available?:** No

A.3 Description

A.3.1 How to Access. We have created a GitHub repository containing a Docker image of our artifact. We have created an account that the evaluators can use to clone this repository, and build and run the Docker image. See A.4 for step-by-step instructions.

A.3.2 Hardware Dependencies. Any commodity multicore CPU with a decent RAM capacity is sufficient to run our Docker image. We recommend a system with at least 4 cores and 16 GB of DRAM.

A.3.3 Software Dependencies. No additional software other than Docker is required as the rest of the dependencies are present within the provided Docker image. Docker can be downloaded from https://docs.docker.com/get-docker/.

A.3.4 Data Sets and Models. No external datasets are required. The datasets used to train our predictive model as well as the matrix datasets used for evaluation are available in the Docker image.

We have provided trained models corresponding to each of the microarchitectural parameters in the Docker image.

A.4 Installation

This section outlines the steps to download the code, build the Docker image and run the image. Note that you may need to run the Docker commands as sudo.

- **Setting up Docker.** Download Docker from https://docs.docker.com/get-docker/ and start the Docker daemon. Verify that Docker is working by running `docker run -t hello-world`.
- **Cloning the Artifact Repository.** We have created a GitHub account that the evaluator(s) can use to clone the repository containing our artifact. Run the clone command using the password (access token).
- **Building the Docker Image.** Once the repository is cloned, go into `sparseadapt/` and build the image: `docker build -f docker/Dockerfile -t sadapt:micro21 . .` It also builds the simulator and takes a while to finish (~ 20 mins with 16 cores).
- **Start the Docker Shell Prompt.** Now run the Docker image using the command `docker run --name micro21-ae -it -e SADAPT_NPROC=16 sadapt:micro21`. Set the `SADAPT_NPROC` parameter to the number of cores to be used for running our experiments. This step should start a shell prompt. It is recommended to start this inside a `tmux` or `screen` session.

A.5 Experiment Workflow

This section outlines the commands and scripts to be used to reproduce the key results of this paper. Note that all directory paths mentioned from here on are relative to `/sparseadapt/sim/`. Once inside the Docker container, run the parent script as `bash rep_run.sh` from `scripts/`. This script reproduces five key sets of results in the paper. Each set of experiments is reproduced by a corresponding child script, described below (approximate runtimes with 16 cores are also mentioned).

- **Figure 5.** Gains for SpMSpV on synthetic dataset with L1 cache: `rep_fig_5.sh` (~ 60 mins)
- **Figure 6.** Gains for SpMSpM on real-world dataset with L1 cache: `rep_fig_6.sh` (~ 120 mins)
- **Figure 7.** Gains for SpMSpV on real-world dataset with L1 cache and scratchpad: `rep_fig_7.sh` (~ 170 mins)
- **Section 6.4.** Comparison of SparseAdapt against ProfileAdapt: `rep_sec_6.4.sh` (~ 110 mins)

Raw simulation data for ProfileAdapt is pre-packaged in the Docker image, as it can take several days to simulate the datapoints required for this. However, simulation can be force-run by un-setting `$skip_sim` in the script.

- **Figure 11.** Effect of SparseAdapt policy and bandwidth sweeps for synthetic dataset with L1 cache: `rep_fig_11.sh` (~ 380 mins)

A.6 Evaluation and Expected Results

After the runs are completed, run `bash rep_check.sh` from `scripts/` to obtain the deviation from the original results obtained for the paper for each of the experiments (in `rep_data_orig/`). The results from the artifact evaluation are stored as tarballs in the path `rep_data/`. The tarballs can be copied back using `scp` or by running `docker cp micro21-ae:/sparseadapt/rep_data ./` from outside the container, for manual inspection of the data and figures.

The major source of variability between the data presented in this work and that from the experiments run by the evaluators arises due to differences in the compiled binaries. For instance, the binaries can have different data layout of the inputs in main memory (*i.e.* exact memory addresses that different data structures are mapped to). This influences the memory access patterns as well as the scheduling, and thus affects both the simulated runtime and energy. For Figures 5-7, a small deviation ($< \pm 5\%$) is expected, while a deviation of $\sim \pm 20\%$ is expected for the remaining experiments. There may be slightly larger deviations if the predictive model guides the configuration sequence in a different direction due to the aforementioned variability.

In this artifact evaluation, we have set up the experiments to reproduce the *key* results in this work. We have omitted the tasks to reproduce the remaining results (Figures 8-10, 12 and Table 6) as these simulation take of the order of weeks to run. However, we have included the original data used to make the graphs/tables within `rep_data_orig/`.

A.7 Methodology

We discuss in this section some details of the modeling and simulation methodology used for this work.

(1) **Simulating Baseline, Max Cfg, Best Avg and Ideal Static.** Each of these static (non-reconfiguring) systems are simulated by running gem5 with the configuration parameters set according to the static specifications (Section 5.3). The code that is run on the simulated GPE and LCP cores is present in `code/src/`. The system specifications are split between arguments to the simulation script (`scripts/run-gem5.py`) and defines in `include/`.
(2) **Training Dataset Construction.** We construct the training datasets for each L1 mode (cache and scratchpad) and optimization mode (Energy-Efficient and Power-Performance) using the approach in Section 4.2. The training dataset is stored in `dataset/<opt_mode>/<l1_mode>/dataset-exp.csv`. Each training example consists of the current configuration parameter values, the current performance counter values, and the "best" configuration parameters.
(3) **Pre-Processing and Predictive Model Training.** The scripts `scripts/parse_dataset.py` and `scripts/fit.py` are used to pre-process the training data and construct the predictive models, respectively. We use the `DecisionTreeClassifier` class and its methods from Scikit-learn for the latter. The original models used to produce the data in this work are stored in `best_models/`.

(4) **Performance Counter and Metric Collection.** To construct the ideal schemes (Section 6.2), the simulation data is collected by randomly sampling $S = 256$ configurations from the space of the explored configuration space (Table 1). Each configuration is used to simulate the workload in its entirety. Furthermore, we divide the workload into continuous intervals (epochs) during simulation. At the end of each simulated epoch, we evaluate the metrics-of-interest and collect the performance counter data, and repeat this for each unique configuration state explored.

(5) **SparseAdapt Construction.** The segments of simulation corresponding to each epoch are stitched together to represent a single dynamic evaluation. At the end of any given epoch, the configuration for the next epoch is selected by running the performance counter data through the predictive model, and applying a pre-determined hysteresis scheme (Section 4.4). This is handled by the simulation script (`scripts/run-gem5.py`). As the next step, the reconfiguration cost to switch between any two configurations is added and the metrics are re-evaluated. This is encapsulated by `scripts/build_oracle_pred.py`. The final step is the evaluation of the *overall* metrics-of-interest using `scripts/gather_oracle_pred.py`.

(6) **Ideal Greedy Construction.** The simulated epoch segments are stitched together as in the previous paragraph, however in the case of Ideal Greedy the next configuration is chosen as the one that has the best metric-of-interest *for the next epoch* (among the sampled points in (4)). The stitched profile is then modified to include the reconfiguration costs across epoch boundaries.

(7) **Oracle Construction.** We mimic a scheme that selects the sequence of configuration changes that optimizes the metric-of-interest for the entire program. We map and solve this as a dynamic programming problem with an implementation following Dijkstra's algorithm. We first construct a weighted directed acyclic graph. Each node is a configuration state S_e that can be chosen for a given epoch e. An edge between state S_i and S_{i+1} consists of the time and energy to execute the epoch with the system configured as S_{i+1} while considering the penalty of going from S_i to S_{i+1}. `scripts/build_oracle_pred.py` constructs this graph from the data in Step 4 and an external program `scripts/dijkstra.cpp` outputs the shortest "path" through the graph, *i.e.* the sequence of configuration choices leading to the (approximate) global optimum.

(8) **ProfileAdapt Construction.** We implement ProfileAdapt by emulating a transition to the *profiling configuration* before going to the next selected configuration (see Figure 3b). For pessimistic comparisons, we consider the ProfileAdapt scheme applied to our Ideal Greedy sequence for each epoch (Naïve), and for epochs following which there is a change in configuration (Ideal). We implement it by dividing the epoch in which there is a change to the profiling configuration into two sections. The first section is considered to be run in the profiling configuration, and second is run in that selected by Ideal Greedy. Since the execution of the workload in the profiling configuration also contributes to useful work done, we calculate the metrics-of-interest for the overall epoch by combining the metrics obtained from the two sections.

REFERENCES

[1] Omid Akbari, Mehdi Kamal, Ali Afzali-Kusha, Massoud Pedram, and Muhammad Shafique. 2018. PX-CGRA: Polymorphic approximate coarse-grained reconfigurable architecture. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 413–418.

[2] Martin Anthony and Sean B Holden. 1998. Cross-validation for binary classification by real-valued functions: theoretical analysis. In *Proceedings of the eleventh annual conference on Computational learning theory*. 218–229.

[3] Bahar Asgari, Ramyad Hadidi, Tushar Krishna, Hyesoon Kim, and Sudhakar Yalamanchili. 2020. Alrescha: A lightweight reconfigurable sparse-computation accelerator. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 249–260.

[4] A Azad, A Buluç, and J R Gilbert. 2015. Parallel Triangle Counting and Enumeration Using Matrix Algebra. *2015 IEEE Int'l Parallel and Distributed Processing Symposium Workshop* (2015), 804–811.

[5] N Bell, S Dalton, and L N Olson. 2011. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM J. Scientific Comput.* (2011).

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

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

[8] Max Bramer. 2007. Avoiding Overfitting of Decision Trees. *Principles of Data Mining* (2007), 119–134.

[9] S Brin and Page L. 1998. The anatomy of a large-scale hypertextual web search engine. *7th Int'l WWW Conference* (1998).

[10] A Buluç and John R Gilbert. [n.d.]. The Combinatorial BLAS: Design implementation and applications. *The Int'l Journal of High Performance Computing Applications* ([n. d.]).

[11] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.

[12] Tao Chen, Shreesha Srinath, Christopher Batten, and G Edward Suh. 2018. An Architectural Framework for Accelerating Dynamic Parallel Algorithms on Reconfigurable Hardware. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 55–67.

[13] S Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. 2017. CGRA-ME: A unified framework for CGRA modelling and exploration. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 184–189.

[14] Corinna Cortes, Daryl Pregibon, and Chris Volinsky. 2003. Computational methods for dynamic graphs. *Journal of Computational and Graphical Statistics* 12, 4 (2003), 950–970.

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

[16] Maximilien Danisch, Oana Balalau, and Mauro Sozio. 2018. Listing k-cliques in sparse real-world graphs. In *Proceedings of the 2018 World Wide Web Conference*. 589–598.

[17] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[18] Bryan Donyanavard, Tiago Mück, Amir M Rahmani, Nikil Dutt, Armin Sadighi, Florian Maurer, and Andreas Herkersdorf. 2019. Sosa: Self-optimizing learning with self-adaptive control for hierarchical system-on-chip management. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 685–698.

[19] Christophe Dubach, Timothy M Jones, and Edwin V Bonilla. 2013. Dynamic microarchitectural adaptation using machine learning. *ACM Transactions on Architecture and Code Optimization (TACO)* 10, 4 (2013), 1–28.

[20] Christophe Dubach, Timothy M Jones, Edwin V Bonilla, and Michael FP O'Boyle. 2010. A Predictive Model for Dynamic Microarchitectural Adaptivity Control. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 485–496.

[21] David Ediger, Rob McColl, Jason Riedy, and David A Bader. 2012. Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 1–5.

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

[23] John R Gilbert, S. Reinhardt, and V B Shah. [n.d.]. A Unified Framework for Numerical and Combinatorial Computing. *Computing in Science & Engineering* 10, 2 ([n. d.]), 20–25.

[24] John R Gilbert, Steve Reinhardt, and Viral B Shah. 2006. High-performance graph algorithms from parallel sparse matrices. In *International Workshop on Applied Parallel Computing*. Springer, 260–269.

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

[26] Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong-Hyeon Park, Austin Rovinski, Haojie Ye, Yuhan Chen, Ronald Dreslinski, and Trevor Mudge. 2020. Sparse-TPU: Adapting Systolic Arrays for Sparse Matrices. In *Proceedings of the 34th ACM International Conference on Supercomputing*. 1–12.

[27] Connor Imes, David HK Kim, Martina Maggio, and Henry Hoffmann. 2015. POET: a portable approach to minimizing energy under soft real-time constraints. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 75–86.

[28] Connor Imes, David HK Kim, Martina Maggio, and Henry Hoffmann. 2016. Portable multicore resource management for applications with performance constraints. In *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*. IEEE, 305–312.

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

[30] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. 2016. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–9.

[31] Wonyoung Kim, Meeta S Gupta, Gu-Yeon Wei, and David Brooks. 2008. System Level Analysis of Fast, Per-Core DVFS using On-Chip Switching Regulators. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. IEEE, 123–134.

[32] Rakesh Komuravelli, Matthew D Sinclair, Johnathan Alsop, Muhammad Huzaifa, Maria Kotsifakou, Prakalp Srivastava, Sarita V Adve, and Vikram S Adve. 2015. Stash: Have your scratchpad and cache it too. *ACM SIGARCH Computer Architecture News* 43, 3S (2015), 707–719.

[33] Matthieu Latapy. 2008. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical computer science* 407, 1-3 (2008), 458–473.

[34] Jure Leskovec and Rok Sosič. 2016. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1–20.

[35] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. 2019. A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. *ACM Computing Surveys (CSUR)* 52, 6 (2019), 1–39.

[36] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M Sleiman, Ronald Dreslinski, Thomas F Wenisch, and Scott Mahlke. 2012. Composite cores: Pushing heterogeneity into a core. In *2012 45th Annual IEEE/ACM international symposium on microarchitecture*. IEEE, 317–328.

[37] Timothy N Miller, Xiang Pan, Renji Thomas, Naser Sedaghati, and Radu Teodorescu. 2012. Booster: Reactive core acceleration for mitigating the effects of process variation and application imbalance in low-voltage chips. In *IEEE International Symposium on High-Performance Comp Architecture*. IEEE, 1–12.

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

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

[40] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 724–736.

[41] Subhankar Pal, Siying Feng, Dong-hyeon Park, Sung Kim, Aporva Amarnath, Chi-Sheng Yang, Xin He, Jonathan Beaumont, Kyle May, Yan Xiong, et al. 2020. Transmuter: Bridging the Efficiency Gap using Memory and Dataflow Reconfiguration. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 175–190.

[42] Subhankar Pal, Dong-hyeon Park, Siying Feng, Paul Gao, Jielun Tan, Austin Rovinski, Shaolin Xie, Chun Zhao, Aporva Amarnath, Timothy Wesley, et al. 2019. A 7.3 M Output Non-Zeros/J Sparse Matrix-Matrix Multiplication Accelerator using Memory Reconfiguration in 40 nm. In *2019 Symposium on VLSI Technology*. IEEE, C150–C151.

[43] Dong-Hyeon Park, Subhankar Pal, Siying Feng, Paul Gao, Jielun Tan, Austin Rovinski, Shaolin Xie, Chun Zhao, Aporva Amarnath, Timothy Wesley, et al. 2020. A 7.3 M Output Non-Zeros/J, 11.7 M Output Non-Zeros/GB Reconfigurable Sparse Matrix–Matrix Multiplication Accelerator. *IEEE Journal of Solid-State Circuits* 55, 4 (2020), 933–944.

[44] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[45] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. 2003. Using simpoint for accurate and efficient simulation. *ACM SIGMETRICS Performance Evaluation Review* 31, 1 (2003), 318–319.

[46] Paula Petrica, Adam M Izraelevitz, David H Albonesi, and Christine A Shoemaker. 2013. Flicker: A dynamically adaptive architecture for power limited multicore systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. 13–23.

[47] Nathaniel Pinckney, Matthew Fojtik, Bharan Giridhar, Dennis Sylvester, and David Blaauw. 2013. Shortstop: An On-Chip Fast Supply Boosting Technique. In *2013 Symposium on VLSI Circuits*. IEEE, C290–C291.

[48] Raghavendra Pradyumna Pothukuchi, Sweta Yamini Pothukuchi, Petros Voulgaris, and Josep Torrellas. 2018. Yukta: multilayer resource controllers to maximize efficiency. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 505–518.

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

[50] M O Rabin and V V Vazirani. 1989. Maximum matchings in general graphs through randomization. *Journal of Algorithms* 10, 4 (1989), 557–567. https://doi.org/10.1016/0196-6774(89)90005-9

[51] Gokul Subramanian Ravi and Mikko H Lipasti. 2017. Charstar: Clock hierarchy aware resource scaling in tiled architectures. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 147–160.

[52] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W Keckler, and Charles R Moore. 2003. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.* IEEE, 422–433.

[53] Anderson Luiz Sartor, Pedro Henrique Exenberger Becker, Stephan Wong, Radu Marculescu, and Antonio Carlos Schneider Beck. 2019. Machine Learning-Based Processor Adaptability Targeting Energy, Performance, and Reliability. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 158–163.

[54] Korey Sewell, Ronald G Dreslinski, Thomas Manville, Sudhir Satpathy, Nathaniel Pinckney, Geoffrey Blake, Michael Cieslak, Reetuparna Das, Thomas F Wenisch, Dennis Sylvester, et al. 2012. Swizzle-Switch Networks for Many-Core Systems. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 2, 2 (2012), 278–294.

[55] V B Shah. 2007. *An Interactive System for Combinatorial Scientific Computing with an Emphasis on Programmer Productivity*. Ph.D. Dissertation.

[56] Anuraag Soorishetty, Jian Zhou, Subhankar Pal, David Blaauw, H Kim, Trevor Mudge, Ronald Dreslinski, and Chaitali Chakrabarti. 2020. Accelerating linear algebra kernels on a massively parallel reconfigurable architecture. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 1558–1562.

[57] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 766–780.

[58] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 689–702.

[59] Narayanan Sundaram, Nadathur Rajagopalan Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. Graphmat: High performance graph analytics made productive. *arXiv preprint arXiv:1503.07241* (2015).

[60] Karthik T Sundararajan, Timothy M Jones, and Nigel P Topham. 2013. The smart cache: An energy-efficient cache architecture through dynamic adaptation. *International Journal of Parallel Programming* 41, 2 (2013), 305–330.

[61] Stephen J Tarsa, Rangeen Basu Roy Chowdhury, Julien Sebot, Gautham Chinya, Jayesh Gaur, Karthik Sankaranarayanan, Chit-Kwan Lin, Robert Chappell, Ronak Singhal, and Hong Wang. 2019. Post-Silicon CPU Adaptation Made Practical Using Machine Learning. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 14–26.

[62] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, et al. 2002. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE micro* 22, 2 (2002), 25–35.

[63] Mikkel Thorup. 1999. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM (JACM)* 46, 3 (1999), 362–394.

[64] S van Dongen. 2000. *Graph Clustering by Flow Simulation*. Ph.D. Dissertation.

[65] Shivaram Venkataraman, Erik Bodzsar, Indrajit Roy, Alvin AuYoung, and Robert S Schreiber. 2013. Presto: distributed machine learning and graph processing with sparse matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 197–210.

[66] Dani Voitsechov, Oron Port, and Yoav Etsion. 2018. Inter-thread communication in multithreaded, reconfigurable coarse-grain arrays. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 42–54.

[67] Yan Xiong, Jian Zhou, Subhankar Pal, D Blaauw, HS Kim, T Mudge, R Dreslinski, and Chaitali Chakrabarti. 2020. Accelerating deep neural network computation on a low power reconfigurable architecture. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1–5.

[68] I Yamazaki and X S Li. 2010. On techniques to improve robustness and scalability of a parallel hybrid linear solver. *Proceedings of the 9th Int'l meeting on high performance computing for computational science* (2010), 421–434.

[69] Zhuliang Yao, Shijie Cao, Wencong Xiao, Chen Zhang, and Lanshun Nie. 2019. Balanced sparsity for efficient dnn inference on gpu. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 5676–5683.

[70] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. 2017. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 548–560.

[71] R Yuster and U Zwick. 2004. Detecting short directed cycles using rectangular matrix multiplication and dynamic programming. *Proceedings of the 15th annual ACM-SIAM symposium on discrete algorithms* (2004).

[72] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. 2020. Sparch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 261–274.