The Dataflow Abstract Machine Simulator Framework

Nathan Zhang
Stanford University
Stanford, USA
stanfurd@stanford.edu

Rubens Lacouture
Stanford University
Stanford, USA
rubensl@stanford.edu

Gina Sohn
Stanford University
Stanford, USA
ginasohn@stanford.edu

Paul Mure
MIT
Cambridge, USA
paulmure@mit.edu

Qizheng Zhang
Stanford University
Stanford, USA
qizhengz@stanford.edu

Fredrik Kjolstad
Stanford University
Stanford, USA
kjolstad@stanford.edu

Kunle Olukotun
Stanford University
Stanford, USA
kunle@stanford.edu

***Abstract*—The growing interest in novel dataflow architectures and streaming execution paradigms has created the need for a simulator optimized for modeling dataflow systems.**

To fill this need, we present three new techniques that make it feasible to simulate complex systems consisting of thousands of components. First, we introduce an interface based on Communicating Sequential Processes which allows users to simultaneously describe functional and timing characteristics. Second, we introduce a scalable point-to-point synchronization scheme that avoids global synchronization. Finally, we demonstrate a technique to exploit slack in the *simulated* system, such as FIFOs, to increase simulation parallelism.

We implement these techniques in the Dataflow Abstract Machine (DAM), a parallel simulator framework for dataflow systems. We demonstrate the benefits of using DAM by highlighting three case studies using the framework. First, we use DAM directly as an exploration tool for streaming algorithms on dataflow hardware. We simulate two different implementations of the attention algorithm used in large language models, and use DAM to show that the second implementation only requires a constant amount of local memory. Second, we re-implement a simulator for a sparse tensor algebra accelerator, resulting in 57% less code and a simulation speedup of up to four orders of magnitude. Finally, we demonstrate a general technique for time-multiplexing real hardware to simulate multiple virtual copies of the hardware using DAM.

I. Introduction

Modern applications such as large language models (LLMs), data analytics, and sparse machine learning have ignited a flurry of research in both dataflow architectures, such as Reconfigurable Dataflow Accelerators (RDAs) and Coarse-Grained Reconfigurable Arrays (CGRAs) [9], [16], [26], [42], [43], [45], [47], and streaming abstractions such as the Sparse Abstract Machine [29]. To explore the functional behavior and performance characteristics of their proposed dataflow systems, many researchers develop bespoke simulators.

To simulate dataflow systems, a framework must (1) support communication among thousands of coupled units, (2) model fine-grained channel behaviors, and (3) simulate heterogenous user-defined models. Furthermore, the scale of these systems demands efficient parallelization. Providing these properties in sequential simulation is straightforward; however, no efficient method exists for parallel simulation.

For well-understood ISA-based systems, such as multi-core CPUs, simulators such as ZSim, SlackSim, Graphite, and gem5 provide trade-offs between accuracy and performance [12], [38], [40], [50]. However, this is not the case for general dataflow systems; programming models, hardware architectures, memory configurations, and even the functionality of individual blocks are constantly in flux.

The state of the art framework for parallel heterogeneous system simulation is the Structural Simulation Toolkit (SST) [49]. However, its programming interface does not model fine-grained channel behaviors, and its runtime scales poorly when simulating tightly-coupled systems. At the heart of this scaling challenge is the traditional event-driven event-queue execution scheme, which results in a runtime whose synchronization overhead increases with both increasing numbers of simulation workers as well as decreasing communication latency. This lack of scalability is particularly important as the number of cores in a single machine has increased dramatically over the years. In 2011, when the Structural Simulation Toolkit was published, the largest server contained four Intel E5-4650 CPUs with 32 cores / 64 hyperthreads total, while today a server can contain up to eight Intel Xeon 8490H CPUs totalling 480 cores / 960 hyperthreads.

To fill the need for high-performance, scalable, and flexible dataflow simulation, we present three contributions:

1) A Communicating Sequential Processes (CSP)-based programming interface for describing the functionality of components (Section III)
2) Asynchronous distributed time and two efficient local-only synchronization algorithms, enabling event-queue-free execution (Section IV)
3) Time-bridging channels which decouple execution of tightly coupled units while capturing fine-grained channel behaviors (Section V)

Using these ideas, we implement the Dataflow Abstract Machine (DAM) simulator framework with the aim of providing a tool for exploring novel ideas in dataflow systems.[1] DAM is an exact, deterministic system, producing the same results on

[1]DAM is available on Github at https://github.com/stanford-ppl/DAM-RS.

each execution. To demonstrate the broad applicability of this framework, we include three case studies on projects built on top of DAM.

First, we perform an algorithm-level exploration of streaming algorithms for computing attention – the key operation underpinning LLMs. Using an abstract dataflow hardware model [51], we characterized the interplay between the algorithms, their local memory usage, and end-to-end performance.

Second, we re-implement an existing functional simulator for a sparse tensor algebra accelerator, using 57% less code and achieving a geomean speedup of $594\times$. Additionally, we were able to evaluate designs whose complexity and scale were previously deemed infeasible (terminated after two days) in minutes. Leveraging the increased performance of our new simulator, we then calibrated the simulator against real hardware by evaluating thousands of possible timing behaviors using Opentuner [3].

Third, we demonstrate a general technique for time-multiplexing real hardware to simulate multiple pieces of hardware using DAM. This is particularly important for co-simulating systems in which large quantities of physical components are difficult to obtain. For example, an academic tape-out generally only yields a small handful of chips. As a proof of concept, we simulate execution of 1024 GPUs running PyTorch models on four physical GPUs [46], and show that the multiplexing system does not introduce significant error when taking real measurements.

II. Existing Simulation Environments

There are two primary aspects to a simulator framework: its user-facing interface and its underlying runtime. General purpose simulation frameworks such as the Structural Simulation Toolkit [49] and YACSIM [31] expose an event-driven interface, and use some form of ordered event queue [1]. The poor scalability of event queues is well known; many specialized simulators such as ZSim [50], SlackSim [12], and Graphite [40] circumvent this problem by using less precise methods, leveraging domain knowledge to trade accuracy for speed. However, due to the domain knowledge required, these are unsuitable for a general purpose simulator framework.

With an event-driven interface, components register event handlers with the framework. These handlers are then invoked by the framework as events arise. While simpler from the framework's perspective, this makes modelling objects with complex internal state difficult as handlers may be invoked at any time by the framework.

III. User-Facing Interface

Our dataflow simulator implementation framework exposes three constructs to users: *contexts*, *time*, and *channels*.

Work as early as the 1970s recognized that communicating sequential processes (CSP) [28] is a natural interface for programming dataflow systems [11]. We find that the converse is also true — the description of dataflow systems naturally maps to the CSP paradigm. Furthermore, this approach exposes abundant parallelism, as processes execute in a largely independent manner.

In order to support simulation, we augment the CSP model with local time (CSPT). Each context owns a *local* monotonic notion of simulated time — the context may step forward in time whenever it would like, and as far as it would like, but cannot step backwards. As illustrated in Listing 1, our model is nearly identical to CSP with the addition of local time annotations on lines 9, 18, and 19.

Contexts may *view* other contexts' times to obtain a lower bound on the viewed contexts' simulated progress, or to wait until a viewed context has reached a certain point in simulated time. This feature facilitates the implementation of complex logical units as a combination of several simpler contexts.

Finally, contexts communicate through *channels*, which connect a sender context and a receiver context. Channels are statically-connected, can optionally be bounded, and automatically handle the simulation of events such as backpressure and starvation. In the merge unit example, the operations on lines 7, 8, 12, 15, and 18 implicitly update the context's time as necessary. Channels do not inherently possess a notion of time and instead bridge between their sender and receiver time zones; this is necessary to decouple the sender and receiver contexts in order to enable parallel execution.

To highlight the qualitative differences between our proposed CSPT scheme and event-driven (ED) schemes, we present two pseudocode implementations of a merge unit in Listing 1 and Listing 2. Both of these modeled units operate with a two-cycle initiation interval and a six-cycle latency. The comparison highlights key advantages of CSPT over ED. First, ED is more verbose than CSPT due to needing dedicated code to align events on the two incoming channels (Listing 2 lines 18–20). Second, event handlers are not allowed to reject events, and the ED unit must therefore store events into a local queue and process them later (Listing 2 lines 3–15). Importantly, this means that the ED implementation cannot simulate backpressure as all channels are effectively unbounded.

IV. Asynchronous Distributed Time

The CSPT interface presents the user with a local notion of time when describing each component, which becomes a distributed notion of time when combining contexts and channels into a simulated system.

By using local time, the CSPT interface allows our runtime to avoid any notion of global time. As a result, DAM is entirely event-queue-free. Instead, our system relies on two lazy pairwise synchronization mechanisms that are triggered only as needed (Fig. 1). This allows contexts to execute far into the future relative to each other, a property that we call *asynchronous* distributed time. In our sparse tensor algebra accelerator case study in Section VIII, we have observed contexts that are thousands of cycles apart. This in turn provides the decoupling necessary to achieve high efficiency on modern multicore systems.

```
struct MergeCSP {
    fn run() {
        let latency = 6;
        let interval = 2;
        loop {
            // peek_next blocks until a value is available.
            let a = peek_next(input_chan_1);
            let b = peek_next(input_chan_2);
            let out_time = time.tick()+latency;
            // Tiebreak in favor of channel 1
            if (a <= b) {
                dequeue(input_chan_1);
                val = a;
            } else {
                dequeue(input_chan_2);
                val = b;
            }
            enqueue(output_chan, val, out_time);
            time.incr(interval);
        }
    }
};
```

Listing 1: Pseudocode for a stream merging unit with an initiation interval of two cycles and a latency of six cycles, implemented with a CSPT interface.

```
struct MergeEvent {
    // Internal data structures
    backlog_1: Queue;
    backlog_2: Queue;
    next_avail_time: Time;
    fn handle_1(evt) {
        backlog_1.push(evt);
        step();
    }
    fn handle_2(evt) {
        backlog_2.push(evt);
        step();
    }
    fn step() {
        if (backlog_1.is_empty() || backlog_2.is_empty()) {
            return;
        }
        let latency = 6;
        let interval = 2;
        let a = peek(backlog_1);
        let b = peek(backlog_2);
        let event_time = max(a.time, b.time, next_avail_time);
        next_avail_time = event_time + interval;
        // Tiebreak in favor of channel 1
        if (a <= b) {
            dequeue(backlog_1);
            val = a;
        } else {
            dequeue(backlog_2);
            val = b;
        }
        send_event(output_chan, val, event_time + latency);
    }
};
```

Listing 2: Pseudocode for an event-driven stream merging unit. While it can simulate initiation interval and latency, this implementation cannot simulate backpressure.

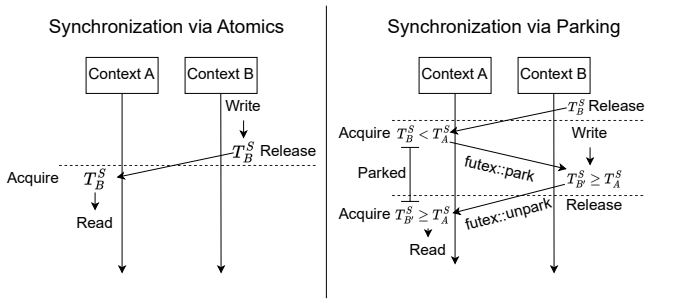

Fig. 1: The two local-only synchronization mechanisms used by DAM (Section IV-A, Section IV-B), illustrating two ways in which context A may synchronize with a context B. Real time flows downward, with lower events being temporally ordered after higher events. Dotted horizontal lines indicate synchronization points, where all prior writes from context B are guaranteed to become visible to context A.

In contrast, prior attempts at parallelizing execution, such as distributed event queues, either implicitly or explicitly constrain the variance between clocks, *synchronizing* across distributed time. For example, the Structural Simulation Toolkit utilizes global barriers with a frequency dictated by the lowest latency inter-thread communication channel, effectively bounding the parallel window. In a dataflow architecture, this would result in a global barrier every few cycles.

Several concepts such as time and channels are used in both the simulator and the system being simulated. To disambiguate, the term *real* concerns the simulator framework, while *simulated* refers to the system being simulated. Real times are represented as T^R and simulated times are represented by T^S.

A. Synchronization via Atomics (SVA)

It is frequently sufficient to optimistically synchronize between contexts; for example, when reading from a real channel, it is sufficient to know that the front contains a value with time $T_A^S = 3$, even if the receiver is at time $T_B^S = 100$. In general, when a simulated structure is a queue, observing the current front object, or the lack thereof, certifies that all previously enqueued objects have been dequeued. Furthermore, if the time of the viewed context is in the future relative to the time of the viewing context, then no further synchronization is necessary.

SVA leverages memory ordering semantics to perform this optimistic synchronization, using acquire/release ordering [25] with transitivity [5], [6]. On systems with strong memory consistency, such as x86, synchronization is handled automatically by the hardware without requiring extra instructions. Simply loading another context's time is sufficient to synchronize with it. On weaker models a memory barrier is required, such as a `dmb ish` on ARM [54].

Consider two contexts A, B from the perspective of A. Suppose that at time T_A^R, A viewed B and loaded (acquire) timestamp T_B^S, which was stored (release) at time T_B^R. Then,

by transitivity, all writes in B prior to T_B^R are visible to all reads in A after T_A^R. As a result, A has a complete history of writes by B up until T_B^S, synchronizing the viewing context A with the viewed context B.

B. Synchronization via Parking (SVP)

SVA is sometimes insufficient, as the viewing context may be in the future relative to the viewed context. In this case, the viewing context must wait for the viewed context. Instead of spin-waiting, SVP instead uses operating system-supported futex operations [21]. Due to the higher cost of SVP, the DAM runtime first attempts SVA before falling back to SVP.

Again, consider the same two contexts A, B, and suppose that at real time T_A^R and simulated time T_A^S, A viewed B and loaded $T_B^S < T_A^S$. While this provides the write history of B up until T_B^S, there may be events at $T_E^S \in (T_B^S, T_A^S)$ which could influence A. In order to handle this case, DAM uses a futex park/unpark pair to suspend the thread executing A *at least until* the thread executing B has reached the desired time. As futex park/unpark pairs execute with release/acquire ordering, SVP also synchronizes A with B.

V. Channels

Channels represent simulated communication links between units, and are directed connections between two contexts. Each channel may be unbounded, or have a user-defined capacity. Each datum is timestamped with its simulated time, which is then used to simulate backpressure and starvation.

Channels are *time-bridging* data structures that are coherent across real concurrent accesses at possibly different simulated times, and are separated into a sender and a receiver, each of which are owned by a different context.

An outline of the sender and receiver structures and their connections is provided in Fig. 2. Each sender-receiver pair communicates primarily through a pair of real channels – a data channel containing timestamped data, and a response channel that contains the timestamp of each read.

Each time the receiver dequeues a value, it responds with a simulated future time indicating when the sender should 'see' the change in channel size; if the sender should observe the read 10 cycles after when the read occurred, then the receiver should enqueue a value of `time + 10` to the response channel.

In Listing 3, we provide pseudocode for enqueuing to a channel. In order for the sender to determine if a channel is full *from the perspective of the sender*, it first checks its local perspective of the channel state. If this check reveals that the channel may be full, the sending context synchronizes with the receiving context using SVA, and then re-checks the response channel and updates its local perspective accordingly. If the channel is still perceived to be full, the sender then performs SVP to verify whether the channel is truly full.

A. Local Time Acceleration

Blocking operations provide opportunities to accelerate forward in time; if a context issues a blocking dequeue to a channel, and the next value received on the channel is timestamped for $T_{data}^S > T_{context}^S$, then the context can simply advance until T_{data}^S. Similarly, a sender can use the response channel to predict the time of the next available opening in a bounded channel and accelerate forward to the received time.

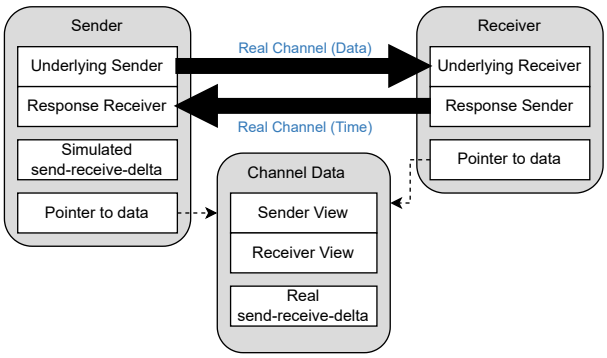

Fig. 2: An illustration of the real sender/receiver data structures and their connections. Simulated channels are backed by a pair of real channels – one which carries data from the sender to the receiver, and another which notifies the sender of the simulated time at which elements were removed from the channel.

As a result of local time acceleration and asynchronous distributed time, only the *real* enqueue and dequeue rates of contexts dictate the real performance of the simulation. This occurs due to two factors: first, by avoiding global synchronization, barriers, and null messages, contexts may execute arbitrarily far into the future until they are forced to halt by backpressure or starvation. Second, by leveraging local time acceleration, contexts are able to advance in time as needed; a sender with a simulated initiation interval of 200 simply causes its receiver to jump forward in increments of 200 cycles on each dequeue.

VI. Discussion

We would like to highlight two points of interest before analyzing the case studies. First, we discuss the impact of scheduling on real simulation performance. This is purely a performance optimization, and simulation results are unchanged. Second, we compare DAM against the current state-of-the-art and highlight the qualitative and quantitative advantages that DAM enjoys.

A. Scheduling

When executing a DAM simulation, users may choose among different operating system scheduling schemes. By default, most operating systems use a fairness-oriented scheduler which temporarily boosts the priority of newly woken threads. This behavior is generally undesirable, as it increases the number of thread switches incurred by the runtime. For example, consider a producer context P which feeds a consumer context C, where the *real* enqueue rate of P is higher than the *real* dequeue rate of C. In this case, after the channel fills, P must

```
fn Sender::enqueue(data) {
    loop {
        if send_receive_delta < capacity {
            underlying.enqueue(data);
            return;
        }
        // Perform SVA
        let recv_time = recv_view.tick_lower_bound();
        if try_recv_from_response() {
            // Attempt to check the response channel for a dequeue.
            // If there was a response, advance to that time since this is a blocking call
            time.advance(received_time);
            send_receive_delta -= 1;
            continue;
        }
        // Wait until the receiver has caught up (SVP)
        // And get the time it is now at since it can overshoot.
        let receiver_time =
            recv_view.wait_until(sender.tick_lower_bound());
        // Elided: the try_recv_from_response block
        // Jump forward in time, and try again.
        time.advance(receiver_time + response_latency);
    }
}
```

Listing 3: Pseudocode for enqueuing to channel which utilizes both SVA (Section IV-A) and SVP (Section IV-B). For brevity, code which updates the datum's timestamp to account for channel behaviors is omitted.

wait for C to progress before P can resume. Under a boosting fair scheduler, once C has progressed, P is immediately woken and preempts C due to its elevated priority. It then executes one step before becoming blocked again. As a result, when applied to oversaturated and imbalanced workloads, DAM should be used with a non-boosting FIFO policy, such as `SCHED_FIFO` on Linux-based systems. These schemes instead execute a context for as long as possible, and thereby reduces the number of thread switches, as well as allowing the slowest contexts to execute for the entire duration. We ran the multiheaded attention application from Section VIII with a parallelization factor of 32 with both `SCHED_FIFO` and the default Completely Fair Scheduler (CFS) [41], saturating a 88 core / 176 vCPU cloud instance, and recorded performance counters using `perf` (TABLE I). FIFO scheduling performs better than CFS in every metric, resulting in a $2.3\times$ speedup in this comparison.

There are also situations where FIFO underperforms; in undersaturated environments, CFS generally performs on-par or better due to lower scheduling overhead. However, these simulations do not typically execute for sufficient amounts of real time to warrant further optimization.

B. SST vs. DAM

The Structural Simulation Toolkit is the state-of-the-art simulation framework for creating architectural simulations, and is written in C++. In order to compare DAM's synchronization scheme and runtime against SST's event-queue event-driven model, we use a parameterized microbenchmark consisting of

Schedule	Metric			
	Context Switches	Page Faults	CPU Migrations	Total Time (s)
CFS	1146392270	1314807	11908306	89
FIFO	37777673	229320	3060225	38
Improvement	$30.3\times$	$5.7\times$	$3.9\times$	$2.3\times$

TABLE I: Performance comparisons between `SCHED_FIFO` and the default scheduler (CFS) on a 88 core / 176 vCPU cloud instance when running the Sparse MHA benchmark with a parallelization factor of 32. In oversaturated systems such as highly-parallelized MHA and the larger benchmarks in Section VI-B, FIFO scheduling is generally faster than CFS.

$\{2, 8, 32\}$ binary reduction trees of depth $\{8, 10\}$, and running 100000 reductions per tree. To vary the amount of work performed per-node, we compute the $\{16, 20\}$th Fibonacci number using the naive exponential method and add it to the result of each node. The same C++ implementation of naive Fibonacci is used for both systems, with DAM calling the function via foreign function interface. To compare how the two systems handle workload imbalance, we increase the computational workload for only the first reduction tree by either 0 (unchanged) or 4 ($16\times$ increase). This benchmark was executed on a 88 core / 176 hyperthread cloud server using Intel Xeon Platinum 8481C CPUs, and simulation results were equivalent modulo implementation differences.

As shown in Fig. 3, DAM outperforms SST in every configuration, with the largest advantage on problems with few, highly connected graphs. Additionally, notice that DAM's advantage is more pronounced with imbalanced workloads. This is due to asynchronous distributed time allowing the operating system scheduler to automatically load balance, while SST's event-driven event-queue model is unable to do so. The minimum speedup using the CFS scheduler was $1.93\times$, while `SCHED_FIFO` achieved at least $3.3\times$.

VII. Case Study: Streaming Attention

For our first case study, we use DAM to explore different streaming implementations of the attention algorithm on dataflow hardware. This case study implements the abstract hardware model and implementations described in parallel work using DAM [51]. The attention algorithm is a mechanism that captures the correlation between tokens within a sequence by exchanging information along the sequence and model dimension. It is used in transformers [55], the foundation for almost all large language models [14], [44], [53]. Given input tensors $Q, K, V \in \mathbb{R}^{N\times d}$ where N is the sequence length and d is the head dimension, the attention algorithm computes output tensor $O \in \mathbb{R}^{N\times d}$:

$$s_{ij} = \sum_k q_{ik} k_{kj}, \quad p_{ij} = \frac{e^{s_{ij}}}{\sum_j e^{s_{ij}}}, \quad o_{ij} = \sum_k p_{ik} v_{kj} \quad (1)$$

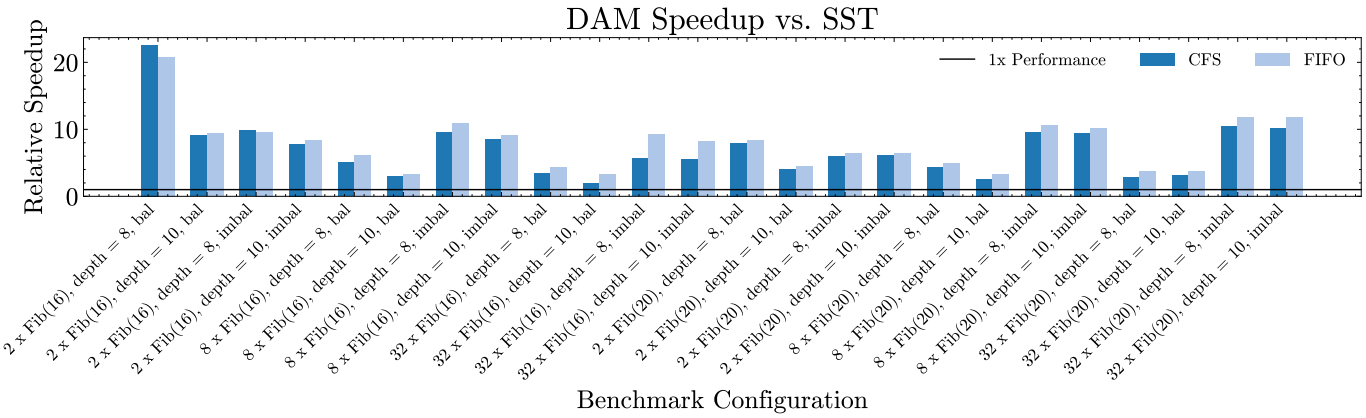

Fig. 3: Speedups over SST using the workload described in Section VI-B. Each configuration is denoted by the work-per-node, the number of reduction trees, workload imbalance, and the depth of each reduction tree. DAM was evaluated using both `SCHED_FIFO` as well as the default CFS scheduler. The horizontal line indicates parity with SST. CFS achieved a minimum speedup of $1.93\times$, while `SCHED_FIFO` achieved a minimum speedup of $3.3\times$.

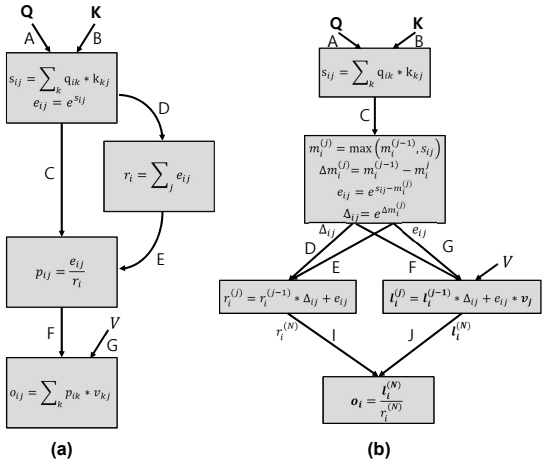

Fig. 4: Computation graph for standard attention (a) and the sequence-length-agnostic attention (b). Bold characters in the left graph are row vectors of each matrix where the subscript indicates the row index.

Attention with long sequence lengths enables models to consider long-range dependencies, and has been empirically shown to produce significantly better models. However, the algorithm has time and memory complexity quadratic in sequence length due to the $N \times N$ intermediate matrices S and P. Streaming execution is well-suited for such workload as it eliminates the need to store and read back the intermediate data by fusing operations [17], [19], [43]. In this study, DAM serves as a tool for algorithmic exploration on dataflow hardware.

A. Implementing Streaming Attention in DAM

To design a streaming implementation for a given application, the first step is to convert the application into a computation graph as shown in Fig. 4 (a). DAM provides a context-and-channel interface that maps naturally to these computation graphs by expressing each node as a context and connecting them with channels. In a manner similar to the example in Listing 1, each block is described functionally, and then timing information is injected by incrementing *local time* at various points. This enables iterating over different designs quickly by allowing the user to focus on the functionality and timing characteristics without necessarily specifying a low-level implementation. In dataflow hardware, the contexts can be mapped to compute units, and the channels can be mapped to either distributed memory or to network buffers depending on depth. Because there is a reduction operation between the first and third context, the depth of channel C in Fig. 4(a) must be at least $N + \alpha$ for the streaming implementation to execute at peak throughput. α is a constant value determined by the producer context's initiation interval and the consumer context's latency, which are constant values agnostic to the sequence length (N). In our implementation, the value of α was 22. The depth of other channels can be all set to a constant value based on the latency and initiation interval of adjacent contexts, which makes the given streaming implementation only require $O(N)$ local memory.

B. Sequence-length-agnostic Streaming Attention

Although the naive implementation can perform attention with $O(N)$ local memory, it is possible to eliminate the dependence on sequence length entirely using an alternate algorithm. To do so, the optimized attention implementation in [51] adapts memory-efficient attention algorithms [18], [48] for streaming execution as shown in the computation graph in Fig. 4(b)[2]. To implement this algorithm for a streaming execution model, an additional context that performs a running sum is required. To confirm that the sequence-length-agnostic implementation only requires a constant amount of local memory without any performance loss, we compare when the maximum channel depth is 22 with the case where the

[2]It uses softmax with scaling [18], [33], [39], [48] instead of the naive softmax as it is widely used in practice for numerical stability.

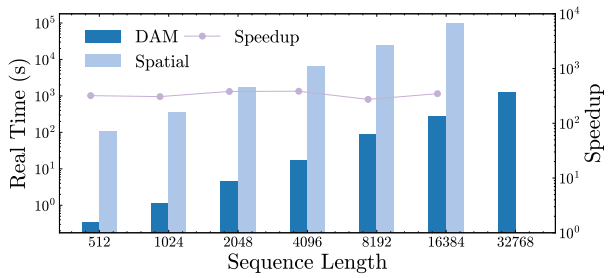

Fig. 5: The comparison between the real time to simulate the streaming implementation for Fig. 4(a) in Spatial and DAM. We omitted unfinished simulations for Spatial after running for three days.

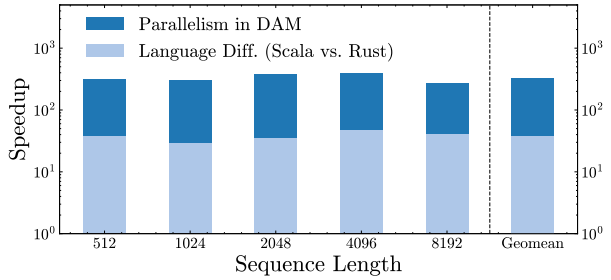

Fig. 6: The breakdown of the speedup in real time when simulating the streaming implementation of standard attention in Fig. 4(a).

channels have an infinite depth (this will show the peak throughput scenario). The required channel depth to reach peak throughput remains $O(1)$ because, in this streaming implementation, the channel depth only relies on the consumer context's latency and the producer context's initiation interval, which is a constant number agnostic to the sequence length. As shown in Table II, we confirm that in the sequence-length-agnostic implementation, all the units run at peak throughput only using $O(1)$ intermediate memory.

C. DAM vs. Spatial

We implement the same streaming algorithm described in VII-A using Spatial [36], a domain-specific language for high-level descriptions of hardware accelerators, and use the built-in hand-optimized simulator written in Scala. We compare the simulation time across different sequence lengths from 512 to $32K$ on an Intel Xeon Processor E7-8890 v3 at 2.5GHz. As shown in Fig. 5, DAM shows more than two orders of magnitude speedup and enables simulating very long sequence lengths such as $16K$ and $32K$ within a few minutes, which would have taken days in Spatial. The simulated cycles in the two simulators matched across different sequence lengths with a constant 8-cycle gap due to a minor difference in startup/shutdown behavior.

Spatial's simulator is written in Scala, while DAM is written in Rust. To measure how much of the speedup comes from the language difference, we emulate the single-threaded cycle-by-cycle simulation of Spatial by restricting DAM to run on a single core and changing the simulated context after each cycle. As shown in Fig. 6, the restricted DAM shows a geomean speedup of $38.25\times$ faster than Spatial across different sequence lengths. Although our implementation for standard attention only has four contexts, and the contexts communicate with each other frequently due to the short channels, we gain an additional geomean speedup of $8.65\times$ from parallel execution and lower thread-switching overhead due to asynchronous distributed time. The speedup from the parallelism in the framework itself will increase as the implementation is parallelized and uses more contexts.

Seq. Length	512	2048	8192	32768
Max Chan. Depth	22 | ∞	22 | ∞	22 | ∞	22 | ∞
Simulated cycles	524K	8M	134M	2B

TABLE II: Simulated cycle count of the streaming implementation for sequence-length-agnostic attention (Fig. 4 (b)) in DAM using different channel depths.

VIII. Case Study: Modeling a Sparse Tensor Algebra Hardware Accelerator

For our second case study, we use DAM to implement a hardware simulator for a sparse tensor algebra accelerator. Sparse tensor algebra is prevalent in various application domains [2], [4], [22], [24], [37]. Prior work, the Sparse Abstract Machine (SAM) [29], enabled the compilation of these applications on streaming dataflow hardware by presenting a streaming abstract machine model for representing sparse tensor algebra using a streaming tensor abstraction. The SAM compiler is an adapted version of the TACO compiler [35], lowering from Concrete Index Notation [34] into a graph of "primitive" blocks connected by streams. This generated dataflow graph can then be mapped onto a dataflow accelerator. SAM represents each tensor as one or more streams with data interleaved with control tokens, resulting in data-dependent behavior. Additionally, SAM also introduces a hardware accelerator design based on a Coarse Grain Reconfigurable Architecture (CGRA), aimed at accelerating sparse applications.

To validate the correctness of the compiler and to aid in the performance characterization of their CGRA hardware, SAM has a hand-written Python simulator to model various applications on the CGRA. However, the extant SAM simulator cannot handle large problem sizes; even simple applications such as matrix-matrix addition could take hours. Furthermore, since SAM is a single-threaded simulator, simulated parallelism in the program graph would *increase* the overall execution time despite taking fewer simulated cycles as this results in more nodes being simulated sequentially. In order to explore larger and more complex applications, we re-implemented the SAM simulator using the DAM framework. This allows us to both simulate the CGRA hardware as well as propose new capabilities by introducing new nodes. The faster simulation times of the new implementation in DAM enabled the use of

Application	MM Add	SpM SpM	SDD MM	MHA (w/ different par factors)						
				1	2	4	8	16	32	64
Contexts	11	15	31	60	92	156	284	540	1052	2076

TABLE III: Number of spawned contexts/threads for each simulated application in SAM including MHA with different parallel factors.

design space exploration techniques to automatically calibrate the simulator to the hardware using an RTL simulation of the components.

A. SAM Implementation in DAM

We mapped the primitives in SAM as contexts in DAM and we mapped streams to channels. We found that the CSPT interface used by DAM is substantially more concise and ergonomic than the cycle-by-cycle abstraction used in the SAM simulator. In Fig. 7, we show the implementation of a Repeat block in both the original SAM simulator and our re-implementation using DAM. Due to the original simulator's cycle-based abstraction, state must be stored in between each iteration – accounting for a majority of the code. Furthermore, the cycle-based abstraction compromises the structure of the code, as different concerns are interleaved throughout. In contrast, the CSPT interface eliminates the majority of the state manipulation code and instead resembles a purely functional description of each unit. As a result, the DAM implementation has 57% fewer lines of code than the implementation of the original SAM simulator.

1) Implementation of Sparse Attention in SAM: Using the SAM implementation in DAM, we were able to design a sparse variant of the multi-headed attention (MHA) algorithm in Transformer models. This was done using existing blocks in SAM now implemented in DAM along with new blocks for supporting memory movement primitives and non-linear operations. Additionally, while work on dense attention found that streaming implementations are vulnerable to deadlock due to undersized data channels when computing the softmax operation (Section VII), we additionally discovered that the metadata streams were similarly prone to causing deadlock and capacity-driven bottlenecks. Furthermore, we observed that random sparsity can cause stochastic deadlock; for example, while a random vector of length 64 with 10% nonzeroes expects to have < 7 nonzeroes, there is a 0.5% chance that it has more than 16 nonzeroes. If the channel was provisioned with depth 16, this would deadlock the system after only a few thousand iterations. This in turn motivates the need for runtime sparsity guarantees, such as a unit which drops excess nonzeroes. We leave exploration of such methods as future work.

2) Original SAM vs. SAM on DAM: To assess our SAM implementation on DAM, we conducted an evaluation running three distinct sparse tensor algebra kernels on synthetic benchmarks randomly generated each with uniformly random sparsity: matrix elementwise add (MMAdd) at 50% nonzeros, sparse matrix-sparse matrix multiplication (SpMSpM) at 10% nonzeros, and sampled dense-dense matrix multiplication (SDDMM) at 30% nonzeros. In addition, we benchmarked our implementation of the sparse MHA in the original SAM simulator and in the implementation of SAM on DAM. For our MHA benchmarks, we fixed the batch size to 8 and the number of heads to 8 and swept across sequence lengths from 64 to 512. All experiments were done on C3 Google cloud instances with Intel Xeon Platinum 8481C CPU at 2.7GHz. Detailed context usage for each of the applications are shown in Table III offering insights into the effectiveness of DAM in terms of context utilization. As shown in Fig. 8, we are able to achieve speed ups ranging from 31.2x to up to 4 orders of magnitude improvement in simulation times, and in some cases are able to simulate larger applications/datasets that the original SAM simulator could not complete within a few days. The speedup DAM achieves over SAM increases with problem size for all applications other than SDDMM.

3) Exploring parallelism in SAM Through DAM: In order to scale designs to handle real-world problems, dataflow systems should exploit more than just pipeline parallelism. To do so, we augmented the SAM model with coarse-grained parallelism. As part of our investigation into parallelizing the MHA algorithm, we performed a sweep of parallelization factors from 1 to 64 as shown in Fig. 9. We find that DAM scales with the amount of *simulated* parallelism until real hardware resources are exhausted, as shown in Fig. 9. We observe a decrease in scaling beyond a parallelization factor of 32, due to core saturation and contention, as core utilization consistently exceeds 95% and the number of contexts/threads in our application surpasses two thousand, as indicated in Table III.

4) Automated Performance Calibration: In the field of hardware-software co-design, ensuring accuracy and consistency between hardware simulators and RTL simulations is crucial for design validation. Often times, calibrating a simulator is a tedious process for hardware designers as it requires manual tuning. Discrepancies between high-level simulators and RTL simulation can be due to a variety of factors such as modeling differences, parameter mismatches, and implementation errors. When comparing the output of our original simulator to the output of the RTL simulation, we found discrepancies in the reported timings of the memory units across all applications on the order of hundreds of cycles. Leveraging the increased speed of the new simulator, we attempted to use autotuning methods to automatically calibrate the simulator's timing behavior to match that of the RTL simulation. This task is phrased as an optimization problem over timing parameters, with the objective of minimizing error with respect to a set of RTL simulation traces. The CSPT abstraction enables this process by providing *local time*; for example, in order to model pipeline bubbles after control tokens, we modified the context to call `time.incr_cycles(x)` when matching a control token, and expose x as a parameter to the autotuner.

This optimization strategy proved to be remarkably effective as shown in Fig. 10, calibrating the simulator to within

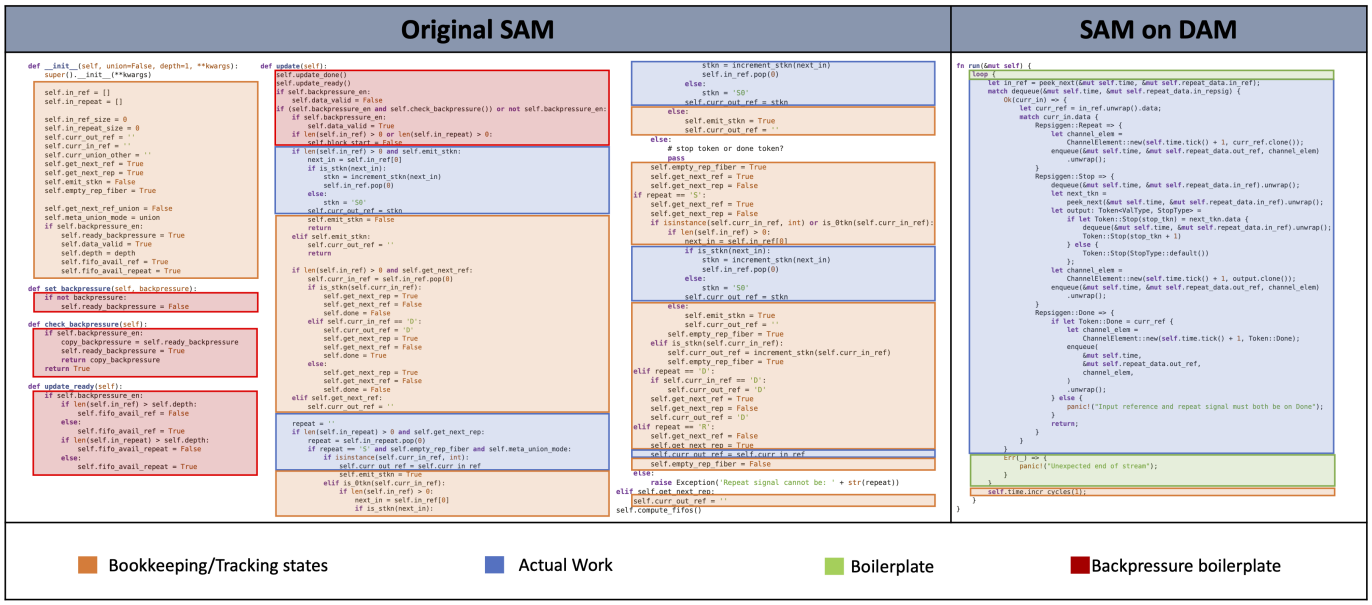

Fig. 7: Comparison of a repeat/broadcast primitive in SAM against its DAM implementation. The code samples are shown and shaded for emphasis, and qualitative comparison, rather than intended for direct legibility.

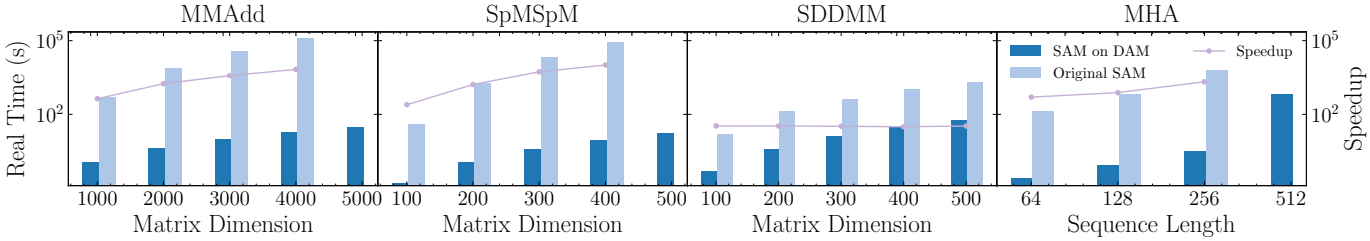

Fig. 8: Comparison of Simulation Times for MMAdd, SpMSpM, and SDDMM where dimensions indicate matrix sizes with row = column. We omitted unfinished simulations for original SAM after running for 2 days.

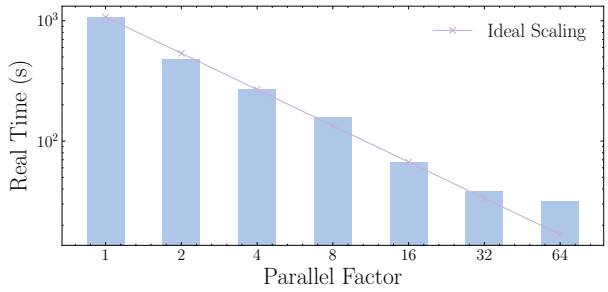

Fig. 9: MHA sweep across parallelization factors from 1 to 64. The machine's cores were saturated at par 32 with 95%+ utilization.

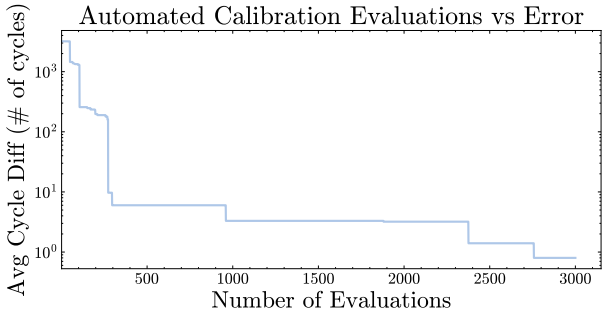

Fig. 10: Applying Opentuner for automated calibration to minimize the error in # of cycles against RTL simulation

an average of 0.8 cycles ($\sim$ 0.3%) of the RTL simulation numbers across the test streams, a significant improvement. To achieve this level of accuracy, we executed a total of 3000 optimization iterations using Opentuner [3], with the search process converging around $\sim$ 2700 iterations. The rapid simulation times enabled by our new simulator were instrumental in enabling this extensive optimization process, as performing such a large number of iterations would have been impractical with the previous simulator. The entire calibration process was completed within minutes, demonstrating the practical feasibility of our approach. We also attempted to use the Bayesian Compiler Optimization (BaCO) [27], but found that its sample efficiency was overshadowed by its significantly slower speed.

B. Python vs. Rust

The SAM simulator is written in Python 3, while DAM is written in Rust. We estimate how much of the performance improvement using DAM comes from the language difference by performing an ablation study on a sparse matrix-matrix multiplication (SpMSpM) benchmark. We establish a baseline by running the experiments restricting DAM to 1 core, using a channel depth of 1, while forcing each thread to yield after every cycle (yielding resulted in a $2\times$ slowdown on average compared to not yielding), and with the default CFS scheduler to emulate the behavior of single-threaded cycle-by-cycle Python execution.

Under these restrictions, DAM ran on average $24.8\times$ faster than the original SAM implementation across different problem sizes. This accounts for the performance benefit from the language difference between Rust and Python. We achieved a substantial performance improvement from parallelism with DAM beyond the language difference as demonstrated by an additional geomean speedup of $87\times$ as shown in Fig. 11. In addition, we analyzed the impact of channel depth on performance and observed that beyond a channel depth of 8, the performance improvement is generally minimal, with the exception of unbounded channels. Execution with unbounded channels is significantly faster than bounded channels, as they remove the need for simulating backpressure altogether.

IX. Case Study: Time-Multiplexed Simulation

An additional capability enabled by CSPT and asynchronous distributed time is the ability to time-multiplex real resources for co-simulation. This capability is critical to simulating large scale systems without access to the full system itself. For example, an academic tape-out may result in a dozen chips, of which only a few are physically set up in the laboratory.

Pseudocode for multiplexing a system is shown in Listing 4. The scheme effectively consists of a regular loop which attempts to lock and use a resource, combined with a performance estimation step. While the *real* lock is held, other contexts which attempt to use the same resource will block. The operating system can then schedule a different, unblocked context to be simulated in parallel. Once the lock is released, the operating system may then schedule the previously blocked context. Perhaps surprisingly, this system prefers an *unfair* lock – if the context re-acquires the lock immediately, then the on-system task will still be the same and therefore the task stash/load can be avoided. Task stashing is done on a *real* task basis, which dramatically reduces the amount of real memory and data movement needed if multiple virtual GPUs execute the same task. Asynchronous distributed time allows more iterations of each task to execute back-to-back, and further reduces the amount of real data movement necessary.

A. Latency-sensitive Inference Simulation

We constructed a simulation modelling latency-sensitive inference, which is common in real-world real-time systems. In this setting, inference is triggered when either: (1) A pre-determined batch size is filled up, or (2) a pre-determined amount of time has passed since the first input arrived. The simulated system seeks to maximize efficiency while simultaneously limiting the maximum inference latency of any input.

This scenario is particularly difficult to model in event-driven systems, as the batching behavior depends on simulated time. Given an input, it is generally impossible to immediately establish when the result should be produced, as it depends on the timing of possible future inputs.

Using CSPT, this system can be modelled by two connected contexts: a batching context and an inference context, connected by *real* channels that exchange simulated timing information and batch data. Asynchronous distributed time allows the batching context to execute arbitrarily far into the future relative to the inference context, and therefore the batching context can gather precise information about the size and timing of each batch *without* advancing the inference context. The inference context lags in the past, and proceeds using the timing information gathered by the batching context. Since downstream consumers only see the inference context, they are unaware of this time manipulation and receive data with the expected timestamps. Similarly upstream producers only see the batching context and thus receive accurate backpressure information.

B. Real Isolation Characterization

Finally, to characterize the error in *real* behavior when multiplexing GPUs, we collected the average and standard deviations in real-time-per-batch of a synthetic PyTorch model on various multiplexed virtual GPU / physical GPU configurations using the previously described setting (Fig. 12). In this experiment, the maximum batch size was 1024, and each batch was full when launched. Real time-per-batch was recorded for each batch, which upper-bounds the error that would be observed if using finer-grained metrics such as CUDA timers.

Average time-per-batch increased slightly with larger numbers of virtual GPUs, but remained within 10% of the original baseline results. Counterintuitively, the standard deviation in time-per-batch *decreased* as the number of virtual GPUs increased, likely due to more consistent loading on the physical GPUs. These results were gathered using a cloud machine with 4 NVIDIA T4 GPUs, with clock rate locked to $1590MHz$.

X. Related Work

a) Simulation schemes: Simulation methods generally fall under two primary families of simulators: optimistic and conservative [23]. Optimistic simulators, as popularized by Time Warp [30], attempt to predict the future, rolling back or undoing mistakes as necessary. DAM does not currently use optimistic approaches, though it is potential future work. Optimism places additional complexity on simulation writers for specifying how to speculate and roll back. Moreover, low-latency/high-frequency communication would cause rollbacks

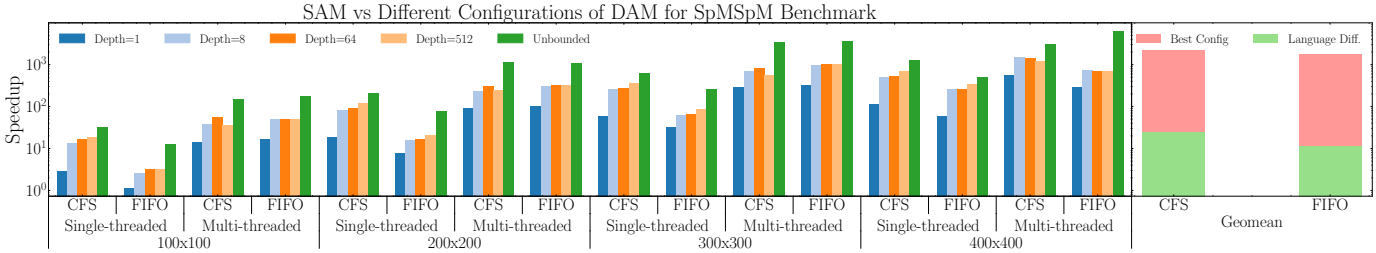

Fig. 11: A comparison between the performance of the original SAM simulator on SpMSpM benchmark against different configurations of DAM. This highlights the performance benefits accounted for by the language difference and the different factors contributing to the remaining speedups. The single-threaded CFS configuration with a channel depth of 1 emulates the single-threaded cycle-by-cycle execution of the original simulator in Python, and shows a $24.8\times$ geomean speedup. We attribute this speedup to the difference in performance between Rust and Python.

```
global_lock: Mutex<Resource>;

struct WrapperContext {
// Some workload which needs to be multiplexed for co-simulation
    let task = ...;
    fn run() {
        loop {
            // Elided: Fetch new inputs from channels
            // Elided: Exit loop if no new tasks
            global_lock.lock();
            if global_lock.task != task {
                // Put away the current job
                global_lock.task.stash();
                // Load our own task into the resource
                task.load();
            }
            task.run();
            global_lock.unlock();
    // use feedback from the task execution to modify local time
            time.incr_cycles(task.estimate_cycles());
            // Elided: Enqueue outputs to channels
        }
    }
}
```

Listing 4: Pseudocode multiplexing a physical resource using the DAM abstraction. While simple, we show in Fig. 12 that the system can achieve relatively stable behavior.

to propagate across significant portions of the system. Nevertheless, we are hopeful that optimism could be adopted on an opt-in basis, resulting in a hybrid simulation framework.

Bound-weave simulators such as ZSim [50] leverage the comparatively low communication intensity of multicore programs to execute thousands of cycles independently, and then re-executing collected traces to correct timings and observed data. This is generally unsuited for the low-latency and frequent communication in dataflow systems, which would lead to mispredictions and invalidation.

Conservative simulation algorithms instead only execute when guaranteed to be correct. These algorithms are generally variants of the original family of algorithms proposed by Chandy, Misra, and Bryant (CMB) [7], [10], [11]. DAM is a conservative algorithm, but does not adopt the event-driven/event-queue paradigm used by CMB-style algorithms.

b) Structural Simulation Toolkit (SST): SST is an open parallel simulation framework for architectural and microarchitectural exploration developed by the HPC community [49]. DAM differs from SST in many aspects: (1) DAM does not use any event queues, (2) DAM exposes a CSPT interface rather than an event driven interface, (3) DAM is able to simulate channel behaviors such as backpressure and deadlock, and (4) DAM does not require global fences or barriers.

c) Architectural Simulators: CPU simulators such as ZSim [50], parallel extensions of gem5 [15], [38], [57], Sniper, and Graphite are designed with a target class of systems in mind. By leveraging domain knowledge, these simulators are able to make performance-accuracy trade-offs. As a general-purpose framework, DAM does not possess the domain knowledge necessary to make such trade-offs, but this does not preclude users of DAM from doing so.

d) Ptolemy: Ptolemy is a project which seeks to model complex programs across multiple models of computation [8]. Ptolemy II is able to model dynamic dataflow programs, as well as other models of computation [56]. However, this work uses a global scheduler and synchronization, and does not model fine-grained channel behaviors.

e) FPGA-based simulation (RAMP [52], FAST [13], FireSim [32]): We see FPGA-based simulators as complementary to DAM. Components can be described and iterated using DAM, leveraging the fast debug and development pace of software. Once researchers are satisfied with projected results and functionality, time can be invested in RTL.

A. Adapting SST with DAM

Despite significant differences between the Structural Simulation Toolkit and DAM, it is worth exploring which of our proposals could be used to augment SST as to reuse years of engineering effort. Unfortunately, as DAM's efficiency stems from the codesign of its interface and runtime, these additions are unlikely to produce the full performance benefits in SST. While various interface improvements can be readily ported, the runtime changes are largely dependent on the interface changes.

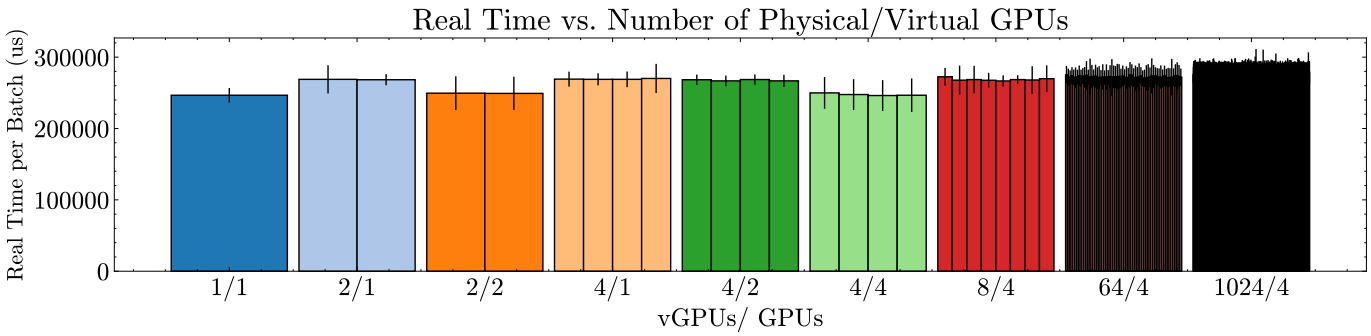

Fig. 12: The average and standard deviations for the real time taken per batch, across a variety of virtual GPU / physical GPU configurations. The average time-per-batch is used as an upper-bound on the error and variance that would be observed using this multiplexing scheme. The average real time-per-batch is stable across configurations, only increasing slightly as the number of virtual GPUs increase.

a) Bounded Channels: Support for bounded channels can be added to SST by replicating the structure illustrated in Fig. 2 using `SST::Links` instead of real channels. This would result in increased event-queue contention as it doubles the number of events, but would be fully backwards-compatible with existing code.

b) CSP w/ Time: With the addition of coroutines in C++20 [20], individual components can be expressed using the CSP w/ Time interface and integrate with the existing event-driven ecosystem. In this scheme, upon encountering a simulated operation which would block, the coroutine would yield control back to the worker. Combined with an appropriate "shim" to handle buffering, the improved ergonomics of CSP w/ Time could be integrated into SST as well.

c) Pairwise Synchronization and Balancing: SST could adapt DAM's SvA/SvP scheme to eliminate global synchronization barriers between workers. In this case, each worker thread would only need to query or await threads which write to connecting links, and straggling workers would only incur the comparatively low cost of SvA. With substantial additional engineering effort, a component-stealing approach could be adopted as well, improving SST's ability to handle imbalanced workloads.

XI. Conclusion

In this work, we tackle the challenge of efficient parallel simulation of dataflow systems. We first described a productive and straightforward interface for describing simulated dataflow systems based on communicating sequential processes augmented with time. We then presented two new mechanisms for performing local synchronization between two dataflow units. Using these mechanisms, we constructed time-bridging channels which expose additional parallelism by decoupling tightly connected units.

These techniques are implemented in the Dataflow Abstract Machine, an event-queue-free framework designed for efficient parallel dataflow system simulation. Using DAM, we are able to (1) perform algorithm-level exploration on abstract dataflow hardware, (2) extend the Sparse Abstract Machine to handle sparse machine learning, explore various tradeoffs in the system, and calibrate the simulator to accurately match the behavior of actual hardware, and (3) time-multiplex limited hardware resources for co-simulation of large-scale systems.

We hope that DAM will provide an efficient platform for conducting future research on dataflow systems and invite the exploration of previously intractable problems.

Acknowledgements

We would like to thank Rohan Yadav, Alexander J. Root, Shiv Sundram, Olivia Hsu, and our anonymous conference reviewers for their feedback and guidance on this manuscript. We would additionally like to thank Caroline Trippel for discussions on memory models, which provided the inspiration for leveraging atomics and futex operations. Finally, we would like to thank Arjun Deopajuri for his feedback on simulator-writer ergonomics as well as naming the project.

This work was supported in part by the NSF under grant numbers 1937301, 2028602, CCF-1563078, and 1563113. This research was also supported in part by the Stanford Data Analytics for What's Next (DAWN) Affiliate Program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies. Paul Mure was additionally supported by the MIT Fernando J. Corbató Fellowship. Gina Sohn was funded by the Stanford Graduate Fellowship.

Appendix

A. Abstract

This appendix describes the setup and instructions to reproduce the experimental results shown in each of the case studies. The provided virtual machine contains preconfigured setups for the DAM vs. SST comparison as well as all case studies. The artifact consists of a single x86-64 virtual machine image running Ubuntu 23.10.

B. Artifact check-list (meta-information)

- **Data set:** Synthetically generated sparse tensors are available in the `~/sam-benchmarks` directory of the virtual machine.
- **Run-time environment:** Any host capable of supporting an x86-64 VMDK image with Ubuntu 23.10.
- **Hardware:** CPU-based experiments used a Google Cloud C3-highcpu-176 (88 core Intel Xeon Platinum 8481C CPU @ 2.70 GHz), with the minimum configuration of 352 GiB of memory. GPU-based experiments used a Google Cloud n1-standard-16 instance with 4xNVIDIA T4 GPUs.
- **Metrics:** Wall-clock execution times.
- **Output:** Terminal Outputs, pickle files.
- **How much disk space required (approximately)?:** $128GiB$ will suffice for all experiments.
- **How much time is needed to prepare workflow (approximately)?:** VM images for both CPU-based and GPU-based experiments are available. Setting up the images on a cloud provider should take less than 30 minutes.
- **How much time is needed to complete experiments (approximately)?:**
  - SST vs. DAM (Fig. 3): 4 hours
  - Spatial vs. DAM (Fig. 5): 16.5 hours
  - DAM vs. Restricted DAM (Fig. 6): 24 minutes
  - Sequence Agnostic Attention (Table II): 4 hours
  - SAM vs. DAM (Fig. 8): 72 hours (We recommend running each original SAM Python simulator tests on separate machines, in parallel)
  - MHA sweep across parallelization factors (Fig. 9): 1 hour
  - Automated calibration (Fig. 10): 2 minutes
  - SAM vs Different Configurations of DAM for SpMSpM Benchmark (Fig. 11): 5 hours
  - GPU Experiments (Fig. 12): 6 hours
- **Publicly available?:** Yes, virtual machine image on Zenodo.
- **Archived (provide DOI)?:** 10.5281/zenodo.10895641

C. Description

1) How to access: The virtual machine image is available at 10.5281/zenodo.10895641.

2) Hardware dependencies:

- CPU scaling benchmarks require a large server, such as Google Cloud C3-highcpu-176 or Amazon EC2 c7i.metal-48xl.
- Smaller CPU benchmarks such as SAM and Spatial comparisons should be executed on machines with at least 8 physical cores and 64 GiB of memory.
- GPU-based simulation requires four CUDA 12.4 compatible GPUs, and a host machine with at least 8 cores.

3) Software dependencies: All required dependencies are packaged into the virtual machine images provided. Using a cloud service to host the virtual machines is recommended.

4) Data sets: We provide synthetically generated matrices/higher-order tensors in the artifact evaluation each with uniformly random sparsity: MMAdd at 50% nonzeros, SpMSpM at 10% nonzeros, SDDMM at 30%, and MHA at 40% nonzeros.

D. Installation

The provided virtual machine images can be used either locally or via cloud service. All experimental setups are under the `dam` user.

E. Evaluation and expected results

1) DAM vs. SST: The data for Fig. 3 can be reproduced using the sst-dam-benchmarks artifact. The resulting data will be placed into two pickle files: `sst_results.pkl` and `dam_results.pkl`. By default, the driver runs the sweep once due to the long runtime. Additional iterations can be performed by appending `-r <repeats>` to the driver script, consuming four additional hours per repeat.

```
$ python sst-dam-benchmarks/driver.py
```

2) Case Study: Streaming Attention:

a) Real time comparison (Fig. 5): This will run a sweep over different sequence lengths and print out the real time to run the simulation and the simulated cycles to the terminal. The `run_all.sh` and `DAM_sweep.sh` files have separate commands for each sequence length with a comment specifying the estimated runtime.

```
$ cd ~/spatial     # For Spatial
$ ./run_all.sh
$ cd ~/dam-experiments/stream-attn-dam
$ ./DAM_sweep.sh   # For DAM
```

The exact numbers for the real time to simulate Spatial and DAM can be slightly different from Fig. 5 because the hardware used for artifact evaluation differs from the hardware used when generating the figure. However, the ratio between the real time to simulate Spatial and DAM will still be consistent with Fig. 5 where DAM shows more than two orders of magnitude speedup over Spatial.

b) Simulated cycles comparison (Section VII-C): The simulated cycles reported from Section E2a can be used to verify that the simulated cycles in the two simulators match across different sequence lengths as mentioned in Section VII-C. Note that the constant 8-cycle gap is due to a minor difference in startup/shutdown behavior.

c) Sequence-length-agnostic attention (Table II): The following command will report the simulated cycles for the sequence-length-agnostic implementation when using channels with a bounded depth (maximum depth of 22) and channels with infinite depth. The `seq_agnostic_attn.sh` file has separate commands for each sequence length with a comment specifying the estimated runtime.

```
$ cd ~/dam-experiments/stream-attn-dam/
$ ./seq_agnostic_attn.sh
```

Checking whether the simulated cycles match for both cases will confirm that the sequence-length-agnostic implementation only requires a constant amount of local memory without any performance loss.

d) Breakdown of the speedup (Fig. 6): To measure the speedup from the parallelism in the framework itself, we restrict DAM to emulate the single-threaded cycle-by-cycle simulation of Spatial. We will call this the *restricted-DAM* simulation. The following command will sweep across different sequence lengths shown in Fig. 6 for DAM and restricted-DAM and print out the speedup breakdown in the terminal. The `speedup_breakdown.sh` file has separate commands

for each sequence length with a comment specifying the estimated runtime.

```
$ cd ~/dam-experiments/stream-attn-r-dam
$ ./speedup_breakdown.sh
```

The `speedup_breakdown.sh` script will compare the real time to simulate DAM and restricted-DAM to measure the speedup from the parallelism in the framework itself, which corresponds to the dark blue portion shown in Fig. 6. To measure the speedup factor from the language difference, we compare the real time to simulate restricted DAM with Spatial. This corresponds to the light blue portion in Fig. 6. Since the Spatial experiments to generate Fig. 6 will take 4 hours, we will use the real time reported from Section E2a. When running the Spatial simulation in Section E2a, it will have created a file called `Spatial_sweep.sh` under `~/dam-experiments/stream-attn-r-dam`. The `speedup_breakdown.sh` script will parse this file and report the speedup instead of running Spatial again.

The breakdown for speedup can have a constant factor difference compared to Fig. 5 because the hardware used for artifact evaluation differs from the hardware used when generating the figure. However, the speedup from the parallelism in the framework itself remains on average $11.2\times$ which is larger than the number reported in Fig. 5 ($8.65\times$).

3) Case Study: Modeling a Sparse Tensor Algebra Hardware Accelerator:

a) SAM vs. DAM (Fig. 8): To run the simulations shown in Fig. 8 comparing Sam on DAM simulator with the original SAM Python simulator, we provide a conda environment in the virtual machine image. The AHA environment is only used in this subsection for running the Python simulator, and is brittle; we do not recommend installing or updating packages. Replace `<Test_Name>` in the following command to one of `MMAdd`, `SpMSpM`, `SDDMM`, `MHA` to simulate each application showcased in the figure separately (note that unfinished simulations shown in the figure for MMAdd 5000, SpMSpM 500 and MHA 512 were aborted after running for 2 days).

```
$ conda activate aha
$ cd ~/sam-artifact/sam
$ pytest -s --durations=0 sam/sim/test/ \
sam-dam-benchmarks/test_<Test_Name>.py
```

To run the DAM simulations of the same applications, run

```
$ cd ~/dam-experiments/comal
$ ./compare_sam.sh
```

This script with sweep through each application and all the different dimensions for each and the simulated cycles and real time will be printed in the terminal. Since we use `time` to record the elapsed time, the time taken to run each experiment is the time shown next to `real` in the printed out text in the terminal. The simulated cycles is after `Elapsed Cycles:`. The speedup is found by dividing the real time values across all the applications in original SAM with the corresponding real time values in DAM.

b) MHA sweep across parallelization factors (Fig. 9): To reproduce the results for the parallel multi-head attention runs, we provide a script for sweeping across parallel factors as presented in the chart using the following steps:

```
$ cd ~/dam-experiments/comal
$ ./run_parallel_mha.sh
```

c) Automated calibration (Fig. 10): To reproduce the results for applying opentuner for automated calibration, use the following command to run opentuner:

```
$ cd ~/dam-experiments/comal
$ ./run_calibration.sh
```

This should run opentuner for many iterations with SAM on DAM to calibrate the simulator to the RTL simulation numbers of the hardware being simulated. This script should also regenerate the automated calibration evaluations vs. error plots (Fig. 10) for the run. Although the optimization run should converge to that cycle number, the plot may look slightly different from Fig. 10.

d) SAM vs. different configurations of DAM for SpMSpM benchmark (Fig. 11): For this ablation study, we sweep through various different configuration of DAM and compare with the original SAM simulator numbers for the SpMSpM benchmark. We also use this ablation study to establish how much of the performance improvement using DAM comes from the language difference. To do so, we establish a baseline running DAM with no parallelism as mentioned in subsection VIII-B. This mimics the behavior of single-threaded cycle-by-cycle Python execution of the original SAM simulator. Using restricted DAM, the single-threaded CFS configuration for each dataset was found to be on average 24.8x faster than the original SAM Python simulator. This is found by dividing the SpMSpM original SAM simulator numbers from part a in the instructions for this case study by the single-threaded CFS configuration runs in this ablation study. This study also shows that the performance improvement from parallelism with DAM beyond the language difference is roughly 87x. Note that the performance numbers may vary depending on the machine the experiments are run on.

4) Case Study: Time-Multiplexed Simulation: This case study uses the GPU setup described above. The following steps are for initial setup, and only need to be performed once.

```
# Start MongoDB in the background
$ tmux new -d "mongod --dbpath <path>"
# Lock GPU Clocks to 1590 MHz
$ sudo nvidia-smi --lock-gpu-clocks=1590
$ sudo nvidia-smi -pm 1
```

For each virtual/physical combination, executing the following script will print both the average time (in microseconds) taken per batch as well as its standard deviation. Fig. 12 was manually generated using these outputs.

```
$ bash dam-torch/run_experiment.sh \
    <vGPUs> <pGPUs>
```

REFERENCES

[1] A. Akram and L. Sawalha, "A survey of computer architecture simulation techniques and tools," *IEEE Access*, vol. 7, pp. 78 120–78 145, 2019.

[2] A. Anandkumar, R. Ge, D. Hsu, S. M. Kakade, and M. Telgarsky, "Tensor decompositions for learning latent variable models," *Journal of machine learning research*, vol. 15, pp. 2773–2832, 2014.

[3] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 303–316.

[4] B. W. Bader, M. W. Berry, and M. Browne, "Discussion tracking in enron email using parafac," in *Survey of Text Mining II: Clustering, Classification, and Retrieval*. Springer, 2008, pp. 147–163.

[5] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber, "Mathematizing c++ concurrency," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 55–66. [Online]. Available: https://doi.org/10.1145/1926385.1926394

[6] H.-J. Boehm and S. V. Adve, "Foundations of the c++ concurrency memory model," *SIGPLAN Not.*, vol. 43, no. 6, p. 68–78, jun 2008. [Online]. Available: https://doi.org/10.1145/1379022.1375591

[7] R. E. Bryant, "Simulation on a distributed system," in *Proc. of the 16th Design Automation Conference*, 1979, pp. 544–552.

[8] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," in *Readings in hardware/software co-design*, 2001, pp. 527–543.

[9] A. Carsello, K. Feng, T. Kong, K. Koul, Q. Liu, J. Melchert, G. Nyengele, M. Strange, K. Zhang, A. Nayak, J. Setter, J. Thomas, K. Sreedhar, P.-H. Chen, N. Bhagdikar, Z. Myers, B. D'Agostino, P. Joshi, S. Richardson, R. Bahr, C. Torng, M. Horowitz, and P. Raina, "Amber: A 367 gops, 538 gops/w 16nm soc with a coarse-grained reconfigurable array for flexible acceleration of dense linear algebra," in *2022 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*, 2022, pp. 70–71.

[10] K. M. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations," *Commun. ACM*, vol. 24, no. 4, p. 198–206, apr 1981. [Online]. Available: https://doi.org/10.1145/358598.358613

[11] K. Chandy and J. Misra, "Distributed simulation: A case study in design and verification of distributed programs," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 5, pp. 440–452, 1979.

[12] J. Chen, M. Annavaram, and M. Dubois, "Slacksim: A platform for parallel simulations of cmps on cmps," *SIGARCH Comput. Archit. News*, vol. 37, no. 2, p. 20–29, jul 2009. [Online]. Available: https://doi.org/10.1145/1577129.1577134

[13] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007, pp. 249–261.

[14] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, "Palm: Scaling language modeling with pathways," 2022.

[15] J. Cubero-Cascante, N. Zurstraßen, J. Nöller, R. Leupers, and J. Joseph, *parti-gem5: gem5's Timing Mode Parallelised*, 11 2023, pp. 177–192.

[16] V. Dadu, J. Weng, S. Liu, and T. Nowatzki, "Towards general purpose acceleration by exploiting common data-dependence forms," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 924–939. [Online]. Available: https://doi.org/10.1145/3352460.3358276

[17] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J.-H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi, "Merrimac: Supercomputing with streams," in *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, ser. SC '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 35. [Online]. Available: https://doi.org/10.1145/1048935.1050187

[18] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," in *Advances in Neural Information Processing Systems 35*, 2022.

[19] A. Das, W. J. Dally, and P. Mattson, "Compiling for stream processing," ser. PACT '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 33–42. [Online]. Available: https://doi.org/10.1145/1152154.1152164

[20] I. O. for Standardization (ISO) and I. E. C. (IEC), *Programming languages – C++*, ISO/IEC Std. ISO/IEC 14 882:2020(E), 2020.

[21] H. Franke, R. Russel, and M. Kirkwood, "Fuss, futexes and furwocks: Fast userlevel locking in linux," in *Proceedings of the Ottawa Linux Symposium*. Linux Symposium, June 2002, pp. 479–495, https://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf.

[22] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," *arXiv preprint arXiv:1803.03635*, 2018.

[23] R. M. Fujimoto, R. Bagrodia, R. E. Bryant, K. M. Chandy, D. Jefferson, J. Misra, D. Nicol, and B. Unger, "Parallel discrete event simulation: The making of a field," in *2017 Winter Simulation Conference (WSC)*. IEEE, 2017, pp. 262–291.

[24] T. Gale, E. Elsen, and S. Hooker, "The state of sparsity in deep neural networks," *arXiv preprint arXiv:1902.09574*, 2019.

[25] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ser. ISCA '90. New York, NY, USA: Association for Computing Machinery, 1990, p. 15–26. [Online]. Available: https://doi.org/10.1145/325164.325102

[26] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "Extensor: An accelerator for sparse tensor algebra," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 319–333.

[27] E. Hellsten, A. Souza, J. Lenfers, R. Lacouture, O. Hsu, A. Ejjeh, F. Kjolstad, M. Steuwer, K. Olukotun, and L. Nardi, "Baco: A fast and portable bayesian compiler optimization framework," 2023.

[28] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, p. 666–677, aug 1978. [Online]. Available: https://doi.org/10.1145/359576.359585

[29] O. Hsu, M. Strange, R. Sharma, J. Won, K. Olukotun, J. S. Emer, M. A. Horowitz, and F. Kjølstad, "The sparse abstract machine," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 710–726.

[30] D. Jefferson and H. Sowizral, *Fast concurrent simulation using the time warp mechanism: Part i, local control*. Rand Corporation, 1982.

[31] J. R. Jump, *YACSIM Reference Manual*, 1993.

[32] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanović, "FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 29–42. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00014

[33] N. Kitaev, L. Kaiser, and A. Levskaya, "Reformer: The efficient transformer," in *International Conference on Learning Representations*, 2020.

[34] F. Kjolstad, P. Ahrens, S. Kamil, and S. Amarasinghe, "Tensor algebra compilation with workspaces," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2019, pp. 180–192.

[35] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–29, 2017.

[36] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun, "Spatial: A language and compiler for application accelerators," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018.

[37] J. C. Kolecki, "An introduction to tensors for students of physics and engineering," Tech. Rep., 2002.

[38] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and Éder F. Zulian, "The gem5 simulator: Version 20.0+," 2020.

[39] M. Milakov and N. Gimelshein, "Online normalizer calculation for softmax," 2018.

[40] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–12.

[41] I. Molnar, "Modular scheduler core and completely fair scheduler (cfs)." [Online]. Available: https://lwn.net/Articles/230501/

[42] Q. M. Nguyen and D. Sanchez, "Fifer: Practical acceleration of irregular applications on reconfigurable architectures," in *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1064–1077.

[43] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 416–429.

[44] OpenAI, "Gpt-4 technical report," 2023.

[45] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer, "Triggered instructions: A control paradigm for spatially-programmed architectures," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, p. 142–153.

[46] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[47] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 389–402.

[48] M. N. Rabe and C. Staats, "Self-attention does not need o(n$\hat{2}$) memory," 2021.

[49] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis *et al.*, "The structural simulation toolkit," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 37–42, 2011.

[50] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 475–486. [Online]. Available: https://doi.org/10.1145/2485922.2485963

[51] G. Sohn, N. Zhang, and K. Olukotun, "Implementing and optimizing the scaled dot-product attention on streaming dataflow," *arXiv preprint arXiv:2404.16629*, 2024.

[52] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanovic´, "Ramp gold: An fpga-based architecture simulator for multiprocessors," in *Design Automation Conference*, 2010, pp. 463–468.

[53] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023.

[54] C. Trippel, "Concurrency and security verification in heterogeneous parallel systems," PhD Thesis, Princeton University, http://arks.princeton.edu/ark:/88435/dsp01gt54kq90n, 2019.

[55] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30*, 2017.

[56] G. Zhou and E. A. Lee, "Dynamic dataflow modeling in ptolemy," 2004. [Online]. Available: https://api.semanticscholar.org/CorpusID:7988955

[57] N. Zurstraßen, J. Cubero-Cascante, J. M. Joseph, L. Yichao, X. Xinghua, and R. Leupers, "par-gem5: Parallelizing gem5's atomic mode," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023, pp. 1–6.