The Mozart Reuse Exposed Dataflow Processor for AI and Beyond

Industrial Product*

Karthikeyan Sankaralingam, Tony Nowatzki†, Vinay Gangadhar, Preyas Shah,
Michael Davies, William Galliher, Ziliang Guo, Jitu Khare, Deepak Vijay, Poly Palamuttam,
Maghawan Punde, Alex Tan, Vijay Thiruvengadam, Rongyi Wang, Shunmiao Xu

SimpleMachines Inc. †UCLA

ABSTRACT

In this paper we introduce the Mozart Processor, which implements a new processing paradigm called Reuse Exposed Dataflow (RED). RED is a counterpart to existing execution models of Von-Neumann, SIMT, Dataflow, and FPGA. Dataflow and data reuse are the fundamental architecture primitives in RED, implemented with mechanisms for inter-worker communication and synchronization. The paper defines the processor architecture, the details of the microarchitecture, chip implementation, software stack development, and performance results. The architecture's goal is to achieve near-CPU like flexibility while having ASIC-like efficiency for a large-class of data-intensive workloads. An additional goal was software maturity — have large coverage of applications immediately, avoiding the need for a long-drawn hand-tuning software development phase. The architecture was defined with this software-maturity/compiler friendliness in mind. In short, the goal was to do to GPUs, what GPUs did to CPUs — i.e. be a better solution for a large range of workloads, while preserving flexibility and programmability. The chip was implemented with HBM and PCIe interfaces and taken to production on a 16nm TSMC FFC process. For ML inference tasks with batch-size=4, Mozart is integer factors better than state-of-the-art GPUs even while being nearly 2 technology nodes behind. We conclude with a set of lessons learned, the unique challenges of a clean-slate architecture in a commercial setting, and pointers for uncovered research problems.

CCS CONCEPTS

• **Computer systems organization → Data flow architectures**;
• **Hardware → Hardware accelerators**.

KEYWORDS

dataflow, reuse, accelerator, multicasting, chips, machine learning

ACM Reference Format:
Karthikeyan Sankaralingam, Tony Nowatzki, Vinay Gangadhar, Preyas Shah, Michael Davies, William Galliher, Ziliang Guo, Jitu Khare, Deepak Kumar, Poly Palamuttam, Maghawan Punde, Alex Tan, Vijay Thiruvengadam, Rongyi Wang, Shunmiao Xu. 2022. The Mozart Reuse Exposed Dataflow Processor for AI and Beyond . In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3470496.3533040

*This paper is part of the Industry Track of ISCA 2022's program.

ISCA '22, June 18–22, 2022, New York, NY, USA

ACM ISBN 978-1-4503-8610-4/22/06…$15.00
https://doi.org/10.1145/3470496.3533040

1 INTRODUCTION

This paper introduces SimpleMachines' Mozart processor, which implements the Reuse Exposed Dataflow (RED) architecture. The processor architecture is a commercialization of two facets of academic research. On the architecture side, we implement the stream-dataflow execution [41] and hybrid Von-Neumann dataflow execution paradigms [42]. On the software side, our compiler framework utilizes synthesis based compilation [43]. The key metrics we focus on are: software maturity, compiler-friendliness and aspirational performance improvement goal of 10× performance per watt over competitive solutions.

Early in the engineering development cycle, we defined our problem-scope to first focus on small-batch AI inference, as this provided a customer/business differentiation and product-market fit[1]. The Mozart implementation was targeted at data-center installations, PCIe Half Height Half Length (75W HHHL) and Full Height Full Length (<300W) form-factors. The architecture itself was designed to scale to training, online training and general data-analytics (including other forms of ML, graph-processing, conventional HPC, etc.) in future implementations across different power envelopes. The business motivation was the low utilization of the state-of-the-art product NVIDIA T4 (and now A100) on small-batch inference: At batch-size=1, the A100 utilization from our measurements is: Resnet50 (2.6%), BERT (26%), SSD-Resnet34 (3.4%), RNN-T (0.75%).

Why a new architecture? As we looked at the needs of applications, their evolving behavior, and the low utilization/performance of GPUs, it became clear that exposing reuse present at other levels of algorithms could overcome the need for batch-level parallelism. Second, exposing abstractions close to ML algorithms — but not tied to deep-learning(DL) models — would allow architecture/application-fit even as the applications rapidly evolved. Data reuse is at the heart of ML/DL as data (weights/samples) is transmuted during training or inference.

To address the business/programmer expectation of software maturity, an architecture-level abstraction is necessary to allow software to access the hardware's benefits. By software maturity, we mean quick and easy integration with frameworks like TensorFlow, allowing a seamless user experience for AI developers/customers. The role of the microarchitecture then becomes hitting performance,

[1]Several Fortune-500 companies were focused on single-batch when engaged with us in customer discovery discussions.

power, area goals, and the role of the compiler becomes correct and performant code.

The existing architecture landscape of CPU, GPU, FPGA, and "general" dataflow failed to recognize reuse as a fundamental primitive. The first step in commercialization of the academic research was to define architecturally how to handle and exploit reuse, which led to defining the Reuse Exposed Dataflow architecture, which is a new class of ISAs. The key innovations are disclosed in these patents [50–52].

Contributions. This paper presents the architecture (section 2) — in our opinion, one of the first attempts at defining an architecture (as opposed to an implementation) for future AI workloads and beyond. It includes a summary of the formal ISA and points to our abstraction of a performance specification. Such a performance specification is going to be increasingly important for future chips as the age of "transparent" speedups fades. The paper also describes in depth the microarchitecture development needed to take such an architecture to commercial production (section 3). It is the most detailed and modern implementation of a dataflow machine disclosed. We touch upon physical design challenges (section 4). We also discuss software stack development needed to get an architecture to be software mature (section 5). We elaborate a bit here on the context of software support for a chip like ours. In a commercial setting, meeting developers at the TensorFlow/PyTorch level abstraction was considered a necessity and the hardware needed to be able to run diverse models easily with no hardware developer hand-holding. In a way, like how a CPU needs to be able to boot an OS, provide a GCC stack, and mother-board related firmware issues, a modern DL accelerator must run existing standard benchmarks like MLPerf at a minimum, and for customer viability allow them to easily run their workloads. This "turn-key" nature was a big factor in the architecture and implementation decisions — both hardware and software. Our quantitative evaluation presents key insights and results (section 6). We then present a reflection on what worked for us, what we learned and what the academic community could learn from our experience (section 7).

To set the context for the remainder of the paper, we distill our key lessons below:

- A behavior-oriented ISA design philosophy is critical for staying relevant in the face of rapid DNN algorithm evolution.
- The added programmer burden for accelerator architectures make accessible performance specs increasingly important.
- There are surprisingly many dimensions of dataflow machines left to innovate on, including basic microarchitecture, hybrid cache/scratchpads with prefetching, interconnection networks, scalable spatial scheduling algorithms, and even just datapath modules.
- Accurate early (pre-RTL) power estimates for novel microarchitectures could help reduce design-time.
- The software lift needed on non-novel pieces is quite high, thus the ease of software development is becoming a primary driver for future hardware.

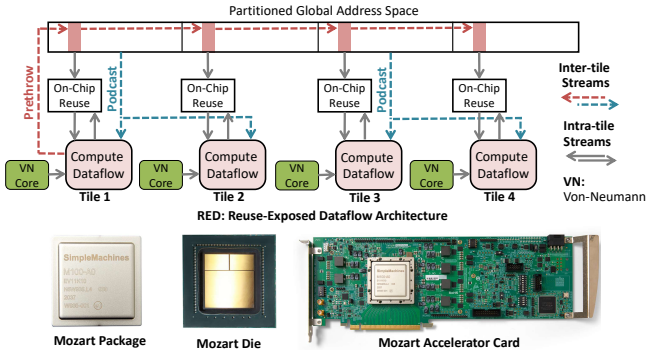

Figure 1: Mozart Hardware and its Reuse-Exposed Dataflow Architecture (RED)

2 ARCHITECTURE: REUSE EXPOSED DATAFLOW

Here we describe the RED execution model and ISA, and explain a performance spec to aid programmers.

Overview. Figure 1 overviews the key architecture abstractions for Reuse Exposed Dataflow. The architecture exposes multiple "tiles" (analogous to cores in a multicore architecture), where each tile has a dataflow accelerator and control core, local scratchpad for on-chip reuse, and fast access to a partition of the global address space. The tile-level execution model is stream-dataflow [41]: coarse-grain memory patterns are expressed as *streams*, which produce and consume values from computations expressed as a dataflow graph (DFG). Stream-dataflow abstractions are blended into a hybrid Von-Neumann/dataflow [42] execution model to allow an imperative ISA to simplify programming.

Other aspects of the execution model provide efficient multi-tile execution by giving the programmer control over locality and spatial reuse. First, RED uses a *partitioned global address space* that enables programmers to allocate tile-local data-structures for fast-access, while the global address space preserves programmability (essentially exposing the hardware's underlying NUMA nature). RED also introduces new communication abstractions: *prethrow* for multicast writes, and *podcast* for multicast reads.

2.1 Tile-level Execution Model

Abstractions. The RED ISA exposes three fundamental abstractions: streams, dataflow graphs, and ports. Memory streams express coarse-grain patterns of accesses. Mozart supports affine access patterns, parameterized by a starting address, access size, stride, and number of strides: `a[stride*j+i]` for $\mathrm{i}_0^{acc\_size}$,$\mathrm{j}_0^{num\_strides}$. Streams may access two address spaces: one global and one local for the scratchpad.

The *dataflow graph (DFG)* represents computation instructions and their dependences. *Ports* interface between streams and dataflow: each stream is connected to a port, and the port provides data to (or consumes from) one or more DFG instructions. DFGs will be "scheduled" by the compiler onto elements of the reconfigurable dataflow hardware. While the DFG is flexible, any hardware instance will impose limitations as described in subsection 2.3, including on instruction and port count.

DFGs execute in a series of *instances*. Each instance consumes a

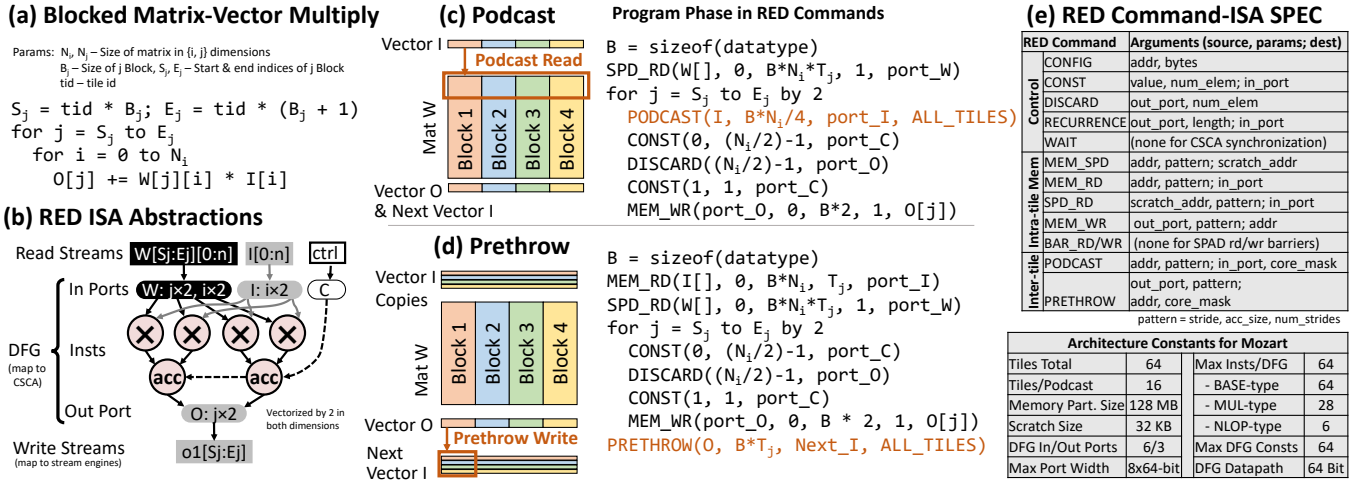

Figure 2: Example Program and RED ISA Specification

predefined amount of data from each input port (called the "port width"), executes instructions in the order defined by DFG dependences, and finally produces a predefined amount of data for each output port. All inter-instance dependences, generally arising from loop-carried dependences, must be explicit. They can either be handled with RECURRENCE streams, which forward data from output to input ports, or they can be handled by updating state that is maintained per-DFG instruction (e.g. accumulators).

In addition to memory and recurrence streams, RED includes additional abstractions to facilitate loop-control. This includes a stream which repeatedly sends a constant value (e.g. for loop invariants), and a DISCARD stream which discards a series of values (e.g. for discarding intermediate outputs of an accumulated value).

Tile-level Example. Figure 2(a) shows the pseudocode of a blocked matrix multiply of an input tensor I by a weight tensor W, and 2(b) shows the translation to RED ISA abstractions for a single tile: streams, ports, and the DFG for this program phase. Here, the weights are stored in scratchpad, and inputs/outputs are stored in global memory. Vectorization is performed by unrolling the loops corresponding to both tensor dimensions, so this DFG performs a 2x2 matrix multiply. A control signal C periodically resets the accumulator state held by the acc instruction.

Note that we will use this same example for the for multi-tile execution as described in the next section, where each block maps to one hardware tile. Hence, Figure 2(a) uses the variable "tid" (tile id) to index into each block to indicate how we parallelize this program across tiles for implementing single-batch parallelization.

2.2 Multi-Tile Execution model

Three abstractions enable the programmer to exploit locality and reuse across tiles: PGAS, podcast, and prethrow.

PGAS. RED adopts a *partitioned global address space (PGAS)*, with equal size memory partitions for each tile. Contiguous memory partitions are mapped to contiguous hardware tiles. This feature enables hardware implementations to use non-uniform cache/memory systems effectively (e.g. a mesh-topology interconnecting tiles), as programmers can reason about locality across tiles when partitioning work and data-structures. The purpose of this scheme is to enable better programmer control over locality, while preserving the simple programmability of a global address space.

2.3 ISA Specification

Podcast. A podcast is a multicast read stream that facilitates bandwidth-efficient access to the global address space, especially for arrays mapped across partitions. A podcast coordinates a set of tiles to take turns multicasting data read from their memory partition to the input ports of other tiles in the set. Podcasts work similarly to stream reads, but they also specify a tile mask to indicate participating tiles. All tiles in the podcast receive the same data at the specified input port, with data sequenced in increasing order of tile ID. Podcasts are limited to a subset of tiles called a pod (hence the name). This restriction enables podcast to be supported by a specialized multicast ring-network with sufficiently-low latency.

Figure 2(c) shows an example use case of podcast when parallelizing the code in Figure 2(a). Here, the input vector is partitioned across tiles. Instead of having each tile redundantly read the same data (causing slowdown due to contention), a podcast can be used to coordinate all tiles in the pod to multicast their data while preserving a consistent ordering. Note that the RED commands implement this strategy, and will be explained in subsection 2.3.

Prethrow. A prethrow is a multicast write stream, that enables efficient high-bandwidth writes to the global address space, also for arrays that are mapped across partitions[2]. A prethrow works similarly to a stream write, but it also includes a tile mask to specify which destination tiles will be updated.

Figure 2(d) is an alternative implementation of the matrix-vector multiply that shows how prethrow can be used instead. Here, the input vector is replicated across all tiles, so that conventional read streams can be used during the computation. At the end of the phase, each tile performs a prethrow to multicast its portion of the output vector to the input vector replicas of the next phase.

[2]Prethrows are not limited to be confined within Pods.

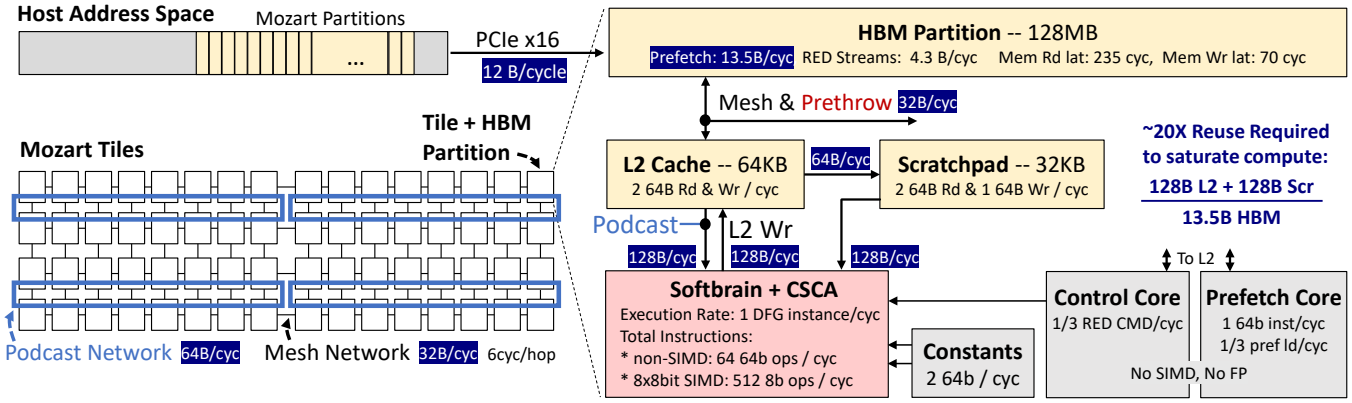

Figure 3: Mozart Performance Spec

Multicast Comparison. Both podcast and prethrow eliminate the expensive serialization required when communicating data from one tile to another. Even though both can be used for multicasting constant data or computation results, there are tradeoffs: Prethrow uniquely requires data replication in the global address space, so should be only be used for smaller data sizes. The implementation of podcast has a pipeline setup overhead, and is also less flexible because the source and destination tiles of the multicast must be the same. Having both enables better flexibility, and enables RED to exploit reuse without relying on batch-level parallelism

These multicast abstractions give RED an edge over existing CPU and GPU ISAs in helping to reduce network traffic and shared cache bandwidth demand. To explain, consider a convolutional neural network (CNN) which is parallelized across output feature maps. While each worker computes a portion of the output feature map, they all need all the input activations. On a GPU architecture, this means that all GPU cores would redundantly fetch the same data, putting additional pressure on the GPU network on chip (NoC) and shared cache bandwidth. Podcast solves this problem by multicasting the input activations to all consumers without requiring them to issue loads. This saves address generation overhead, cache access, and network traffic. In a CNN, regardless of the parallelization strategy, there will be at least one data structure which is read by more than one worker, and this reuse is often simultaneous; this highlights the importance of multicast-enabling optimizations.

Imperative ISA Integration. For programmability, RED commands are embedded into a conventional imperative ISA (and in Mozart encoded with RISC-V instructions). Figure 2(c) and 2(d) also show the podcast and prethrow strategies written in RED commands, with 2(e) summarizing the RED ISA.

A program can consist of multiple *kernels*, with each kernel comprising a set of RED accelerator commands and general purpose code running on the control core. The reconfigurable dataflow accelerator is configured to execute a DFG by loading configuration bits from memory using the `CONFIG` command. In our Mozart implementation, the reconfigurable dataflow hardware is called CSCA: circuit-switched compute array. The (low-level) programmer specifies the DFG in a simple SSA [13] format, and the compiler is responsible for scheduling the DFG onto elements of the CSCA and generating CSCA configuration bits.

Once configured, the accelerator is invoked purely by issuing stream commands to move data to/from its input ports — i.e. see the examples in Figure 2(c) and (d), which show the podcast and prethrow kernels in RED commands. Commands are defined to execute in parallel, unless they access the same port, or are serialized (e.g. scratchpad barrier `BAR_RD/WR` or `WAIT` command which blocks until accelerator execution completes).

We remark that the overall tile ISA is heterogeneous; this allows RED to be simple and focus on performance-critical abstractions, and leave complex control and memory accesses to a modest von-Neumann core. By allowing streams and DFGs to be configurable independently, DFGs can often be reused across different program phases, saving configuration time.

Architecture Constants. An implementation of RED exposes several hardware constants to programmers; Mozart's parameters are enumerated in Figure 2(e). These include a maximum memory partition and scratchpad size, as well as total tiles and tiles/podcast. The DFG is restricted in total instructions (including specific types), as well as input/output port counts and widths. Mozart's datapath size is 64-bits, and smaller ops use subword SIMD. All constants here are hard constraints (violating these constraints will generally cause compile-time errors).

Performance Specification. While RED exposes programmers to low-level ISA abstractions, it also simplifies performance reasoning by being sufficiently deterministic. Most importantly, it enables pipelined dataflow computation at the rate of one DFG instance per cycle[3]. Multiplying the DFG I/O by reuse ratio (streams abstractions make this easier to compute) gives the required bandwidth at each memory/network level — immediately revealing the bandwidth bottlenecks. To make this insight easier for programmers, we developed a pictorial performance spec that characterizes the capacity, fill-rates, and latency between architecture primitives. A simplified version is shown in Figure 3 for Mozart's implementation of the RED architecture. While this performance spec is not mathematically formal, it proved to be surprisingly effective, necessary *and sufficient*: our software team, with little knowledge of chip architecture or background in high-performance assembly coding, was able to write high performance code (See section 5).

[3]The latency of a DFG instance is generally in the 10s of cycles, but the hardware guarantees pipelined execution at the rate of one instance per cycle.

Host-to-RED Integration. Because RED accelerators are attached to a host, a host program needs to be able to invoke RED accelerator functionality. In our implementation, we enable this through an API resembling CUDA and OpenCL, including APIs for memory allocation/deallocation and kernel invocation/synchronization. Because PGAS and multicast requires some specific programmer considerations in memory allocation, we elaborate further on that aspect in particular.

First, the accelerator's global address space is mapped as a contiguous region of the host's virtual address space. As in CUDA, accelerator-specific malloc/free functions manage the accelerator address space, and there are specific APIs to copy data between host and device. Data structures allocated in the accelerator region may be used and modified by either the host or accelerator, but updates must be copied explicitly.

Both podcast and prethrow abstractions require that memory source (for podcast) or destination (for prethrow) addresses are to the same relative offset within each partition of the global address space. To simplify this for the programmer, we add a PGAS-aware malloc which allocates memory at the same relative offset in all designated partitions.

3 MICRO-ARCHITECTURE

In this section, we first describe the SoC design and then micro-architecture of the tile, which is the key building block.

3.1 SoC Organization

Mozart's tiled architecture, shown in Figure 4, is instantiated as a System-on-a-Chip (SoC) where tiles are arranged in a 4x16 organization using two-dimensional mesh. The 256-bit mesh network across all the tiles uses x-y dimensional routing and a packet-switching protocol with a single cycle hop-to-hop latency. For supporting podcast semantics, the tiles are further organized into the abstraction of *Pods*, consisting of 16 tiles organized in a 2x8 grid. A 512-bit broadcast bus running in anti-clockwise direction among the pod tiles helps achieve fast data sharing via the podcast mechanism. At the northern periphery of the chip are two HBM2 controllers, logically split into 16 DFI-level channel controllers [22], with 8 channel-controllers in each hemisphere, operating at maximum bandwidth of 2000Mbps. Each channel-controller further has 2 pseudo-channels, with each channel connecting to 2 tiles of a given pod for off-chip high-bandwidth memory access. A bridge module called Edge-memory-manager (EMM) is instantiated at the top of pods, for protocol conversion from HBM-AXI to the internal mesh network. A series of NetworkInterface-Controller (NIC) blocks form the configuration datapath of all the IP blocks (tiles, HBM/PCIe PHY and controllers during boot) and for low-speed inter-IP communication (SPI, I2C, SMBUS). The SoC also contains a Gen3 x16 PCIe endpoint controller and PHY. The host can query and configure Mozart through PCIe by accessing internal registers. The SoC runs on a single global clock, the PCIe and HBMs have their own clock and PLL, and the configuration registers run on their low-speed clock used only during boot/bringup. We included a debug subsystem that allowed memory and execution tracing of the control-core, which was invaluable during bringup.

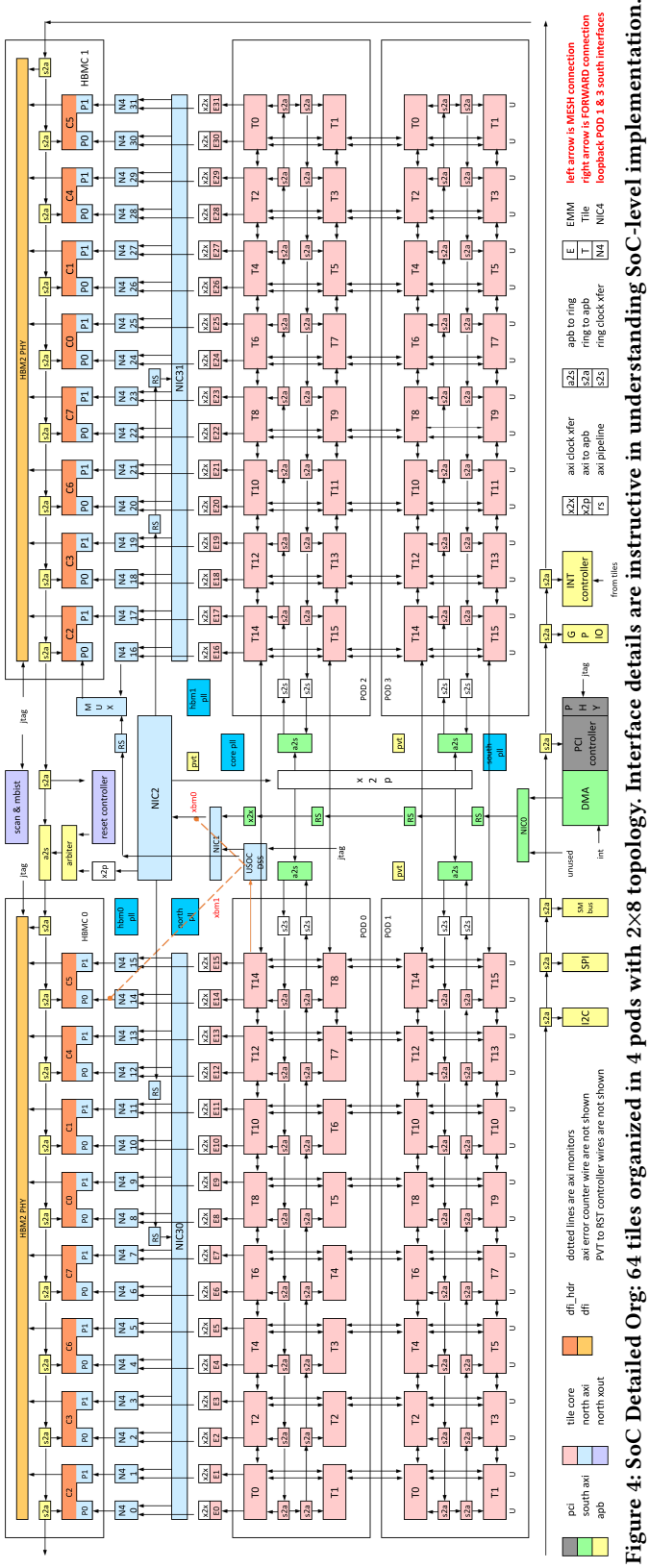

Figure 4: SoC Detailed Org: 64 tiles organized in 4 pods with 2×8 topology. Interface details are instructive in understanding SoC-level implementation.

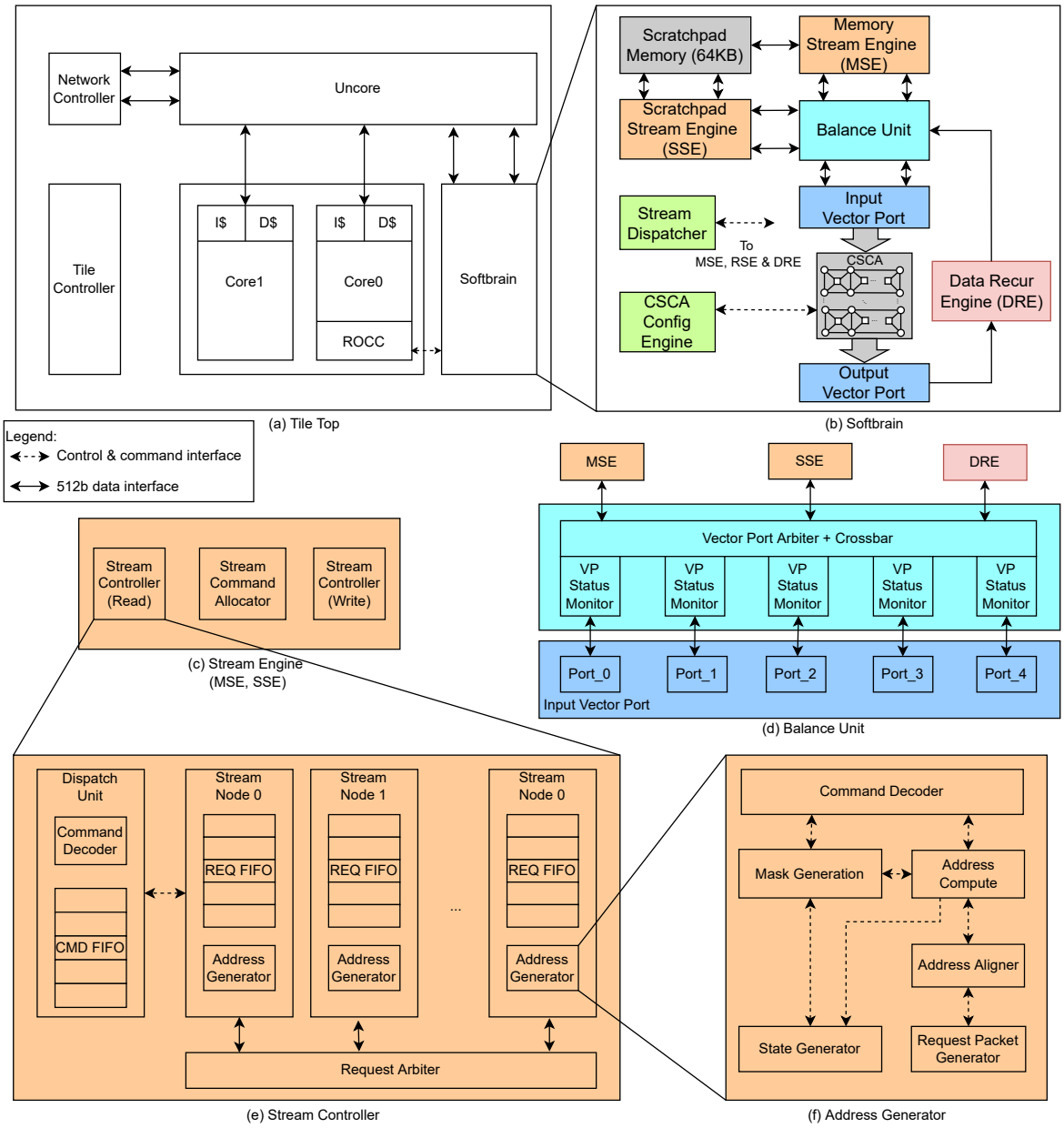

Figure 5: Mozart Tile Microarchitecture

3.2 Tile

Fig 5(a) shows the organization of an individual tile, which consists of 5 blocks: Tile Controller, Network Controller, Uncore, Softbrain, and Von-Neumann Engine. Here we detail the components of the tile, except the Softbrain compute accelerator. The tile is managed by the **tile controller**. This module orchestrates tile execution, completion, aborting, signalling the host via interrupts, clock gating, and tile configuration via the PCIe-accessible APB-based configuration ring. It also handles explicit cache flush mechanisms of uncore's L2-cache and the Von-Neumann Engine's L1-caches before or after kernel execution based on user-exposed configuration, as well as includes a global lookup table to manage the PGAS.

Network Controller. Each tile has a 256-bit mesh network interface in all 4 directions — north, south, east and west, used for inter-tile and peripheral (HBM2) communication; a forwarder mesh network interface is also present for the tiles in the bottom pods to forward their request and responses to/from the HBM2. The network controller is responsible for handling communication with the mesh. Its router block is a fairly conventional: 256-bit physical link, 8 virtual channels, credit-based flow-control, and dimension-ordered X-Y routing. The interesting/non-conventional pieces are hardware support structures for performing the podcast and prethrow operations. Incoming podcast streams, via the 512-bit broadcast bus, are forwarded to Softbrain ports for consumption and are back-pressured when ports are full. Similarly, the network controller also

monitors the prethrow responses and forwards the write request to the uncore and formulates the write request to remote tiles. Each tile can generate 128 64B worth of outstanding read and write requests to each of the HBM controller's AXI channels to saturate the memory bandwidth.

Uncore. The uncore is a collection of modules responsible for holding on-tile L2-cache storage, coordinating local and remote tile cache accesses, and synchronization. Regarding synchronization, we extend the original "A" extension of the RISC-V ISA to allow tile-local thread synchronization and global barriers. Mozart has a HRF consistency model [27], with programmers expected to use atomics to achieve cross-tile coherence and writing well-behaved code. The uncore also includes the control state machines needed for podcast and prethrow operations. In terms of performance, the primary requirements are i) to provide high-bandwidth access to Von-Neumann Engine's cores (1 cacheline per cycle) and Softbrain (2 cachelines per cycle to achieve one DFG instance per cycle execution rate in the common case), ii) fast load-to-use latency from the first request it receives, and iii) support many in-flight requests to exploit bandwidth — it supports up to 64 HBM2 memory requests in-flight.

The tile-wide L2-cache is a banked cache supporting a fully pipelined interface for local accesses from the cores and Softbrain. Each bank is a 16KB instantiated SRAM macro having one read and one write port. Uncore instantiates a crossbar called L1_L2_net for requests/responses incoming into the local L2 cache, and another crossbar called L2_mem_net for requests/responses to be sent to HBM2 via the network-controller. Both the crossbars implement a customized version of the tilelink protocol [12, 34] modified for RED ISA's semantics. Each cache bank further features read and write transaction handling registers (TSHRs), enabling up to 16/16 in-flight memory reads/writes in parallel.

Von-Neumann Engine. The Von-Neumann engine shown in Figure 5(a) is composed of 2 RISC-V in-order general purpose programmable cores. The *Control Core* is extended to run the RED command ISA, and the *Prefetch core* coordinates cache management, podcast and prethrow. The cores instantiated are based on an open-source implementation of Rocket core [3] and implements RV64IM ISA completely, with substantial modifications to support atomics (synchronization between 2 cores inside the tile and across tiles, faster L1-access pipeline, and L2-prefetch). Each core has a local 8KB instruction-cache and 16KB data-cache, and both the cores have high-bandwidth access to the uncore's L2-cache, and are able to read and write a 64B cacheline every cycle when there are L1-cache misses. Configuration registers in the core allows transparent mapping of the host address space to the PGAS address space based on tile location. The Rocket Custom Co-Processor (RoCC) interface was used to communicate between the cores and Softbrain.

3.3 Softbrain

The Softbrain module shown in Fig 5(b) implements the compute dataflow abstraction of the RED ISA, using pipelined computation and streaming access. It consists of a circuit-switched compute array (CSCA), which has similarities to a conventional coarse grain reconfigurable architecture (CGRA). To supply the CSCA at peak throughput, we have a stream-dispatcher (command dispatch unit), several stream engines to manage coarse grain memory streams, input and output vector ports (that implements the ISA's port abstraction), a programmable scratchpad, and a balance unit to arbitrate between these modules. Its goal is to achieve an execution rate of one DFG instance per cycle, which translates to a performance of 64 64-bit and 512 8-bit ops/cycle per tile.

CSCA. The CSCA is an 8x8 array of functional units interconnected by a circuit-switched, software re-configurable network. The configuration is produced by a scheduler, which we discuss in section 5. Each functional unit executes one of several operations (e.g. add, mul or non-linear operations). Operations can perform sub-word SIMD execution to operate on different datatypes (e.g. int8, float32). CSCA configuration takes 60 cycles if the configuration data is cache resident.

Stream Engines. Softbrain contains stream-engines (shown in Fig 5(c) for handling memory (by memory stream engine or MSE) and scratchpad (by scratchpad stream engine or SSE) data streams. The MSE and SSE stream engine consists of 2 main sub-blocks — stream controllers (read and write) and a command allocator. The command allocator receives coarse-grained stream commands from the stream dispatcher and allocates them to read and write stream controllers based on the data stream type, while respecting the dependency among them. Read and write stream controllers can both process the streams in parallel, continuously generating memory and scratchpad vector loads and stores whose access patterns are encoded in the stream commands received.

Recurrence and constant data streams are handled by the data recurrence engine (DRE). A recurrence stream causes one output port of the CSCA to be piped to an input port to support dependences between DFG instances. Constant streams transmit a user-specified constant value into input ports. The DRE is a stateless block responsible for implementing these stream behaviors. To support recurrence streams, it receives commands to transfer the partial data produced by CSCA from output ports back to input ports to be used in the next DFG instance.

The stream controller's internal design consists of a dispatch unit and multiple stream nodes (Fig 5(e)). Each stream node includes FIFOs to buffer multiple data stream requests and address generator units. Addresses for memory, scratchpad reads and writes are produced by high-performance, affine address generators (AGU) (Fig 5f). We include multiple units to support many requests to be generated per cycle. Each AGU, based on the data stream's access size and stride patterns, generates a bitmask at 8-byte word granularity which is used to compute cacheline aligned addresses to be sent to the tile's cache and local scratchpad. The state generator tracks the progress of the data stream and updates the parameters of the stream until the boundary conditions of the stream are met.

The basic stream engine design is instantiated twice to implement the MSE and SSE with each instance connecting to different ports on the input/output interfaces. The MSE is responsible for streaming data from the memory hierarchy to/from Softbrain through interfaces to the uncore and the vector ports. The MSE is capable of issuing and consuming two cacheline reads and two cacheline writes per cycle. The scratchpad stream engine (SSE) similarly handles data streams in and out of local scratchpad memory to CSCA with the same bandwidth requirement as the MSE.

Vector Ports. Input Vector Ports (IVPs) are a staging area for data

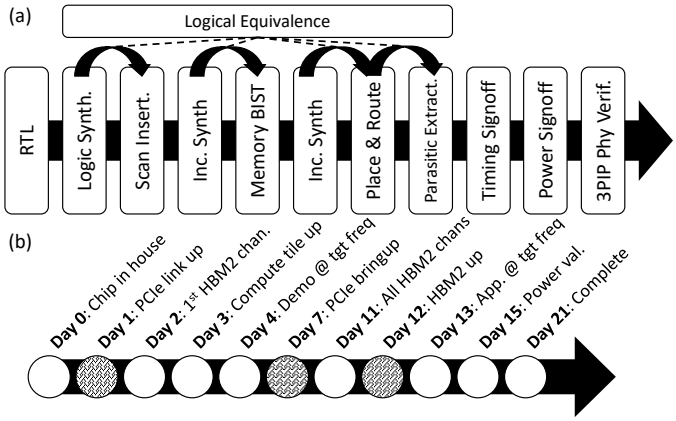

Figure 6: (a) Backend design flow and (b) chip bringup timeline. Shaded events required 3rd Party Vendor signoff.

to be injected into the CSCA to implement the RED execution model of graph execution. Multiple vector ports can link to switches on the CSCA, though only a subset are active at a given time, corresponding to the active configuration. IVPs can consume one cacheline worth of elements per cycle. Output Vector Ports (OVPs) serve a similar role, but as outputs. They consume words produced by computation in the CSCA and reassemble them into cacheline sized chunks. Stream engines consume these streams of data to be used to write to memory, or used in a recurrence stream.

Balance Unit. The balance unit (Fig 5d) is responsible for priority allocation of stream engines to allow streams to make forward progress to ensure optimal execution of Softbrain's compute resources and also to avoid CSCA stalls. It does this by tracking current state, fill-drain rates and the depth and width of all intermediate buffers in the dedicated status monitors for each vector port. A central arbiter and crossbar is responsible for switching streams via round-robin mechanism to hide the latency of vector ports fill-drain operation and also to allow multiple streams to be assigned to MSE, SSE and DRE.

4 CHIP IMPLEMENTATION

Figure 6 shows our development flow and bringup timeline. Due to the nature of the chip, the team comprised of two chip leads with 20+ years of implementation experience and numerous previous chip tapeouts, and also a fairly inexperienced team but most with intuitive knowledge on dataflow principles. The entire team was less than 20 engineers to allow an agile flow, and based on our mind-set of team-augmentation and to add members only if necessary. The chip bringup timeline is shown in Figure 6(b) — the massive time investment (manual detail-oriented work) in validation planning between tapeout and chips-in-house allowed a 21-day bringup, in spite of an unconventional HBM/interposer methodology and IP vendor mixing of controller/PHY. The main hiccup was in HBM PHY tuning, which required close collaboration with the vendor to hit the target frequency. Although generally "straight-forward," the amount of tricks/intuition necessary from experience for a successful tapeout was re-affirmed.

Most of what we did for implementation was conventional — except for the use of Chisel (https://www.chisel-lang.org/) for design entry, which we abandoned for the next chip[4].

On the physical-design, the amount of somewhat mechanical manual effort was large, but is considered common-place in industry. Much of the microarchitecture modeling-level area/power estimates were optimistic, but were useful as general guidelines and helped us quickly arrive at post first-RTL area/power estimates for floorplanning and package design (there were some quirks where the SRAM aspect-ratio had an outsize influence on area — less appreciated in academia but common knowledge in industry). Even though the design was logically distributed, we used a single global clock to minimize verification surprises at PD-level and CDC timing closure issues, which ultimately led to at most 25% hit in Fmax. In retrospect, we would stick to this even for a do-over, simply to minimize risk.

Our use of interposer was fairly straight-forward in-spite of some DFT/Wafer-level challenges from multi-vendor interaction. Due to constraints on what IP block was available in what technology node variant with test silicon, we were forced into combining PHY and controller from different vendors. At the time, only a prohibitively expensive vendor with turnkey IP and design service had a PCIe PHY+controller and HBM PHY+controller in a single technology node. This mixing led to a small dollar-cost in verification, and later in the project an unexpected nearly 4 month delay in the vendor completing the PHY (because of design bugs in level shifters inside the PHY, and a late-stage design bug in the PHY training linkup sequence).

On the verification side, a comprehensive random program generator at the RED ISA level, with industry-standard methodology of checkers, provided 100% coverage at RTL/microarchitecture level pre-tapeout. There was a concern whether the ISA abstracts too much, requiring other ad-hoc testing for high coverage, but we did not need it.

5 SOFTWARE STACK

Deep Learning Framework Integration. Mozart's toolchain and software stack were designed primarily to support modern deep learning (DL) applications. [35] describes these general components well. Additionally, because the RED architecture exposes AI primitives, the intellectual software lift did not become excessive in any one layer. Table 1 shows the capability of this toolchain and results from pushing through our flow the original FP32 trained models of applications from MLPerf. The Table shows the total number of operators in the dynamic graph for each DNN (Ops) and the number of unique operators (Uniq). Recall that here operator refers to a graph-level operator like convolution or matmul, which operate with tensors as inputs and outputs. Our software framework is comprehensive enough to cover a wide mix of operators across DNNs. The last 3 columns show performance, which we cover later. In general, we wrote a significant amount of infrastructure code,

[4]The choice of Chisel presented significant challenges as soon as initial physical design started, but we were too far along to revisit. The shortcomings (for which we developed workarounds) included: constructs which were highly unfriendly for synthesis for several design components, poor support for straight-forward high-fanout distribution creating clock tree synthesis challenges, failing IEEE standards for correctness in generated RTL leading to LEC violations, small design changes created major upheaval in RTL which caused churn in PD flow, naming conventions used broke industry-standard ECO flow, fundamental incompatibility with scan-insertion at RTL level, forcing netlist level scan insertion which is highly undesirable for productivity.

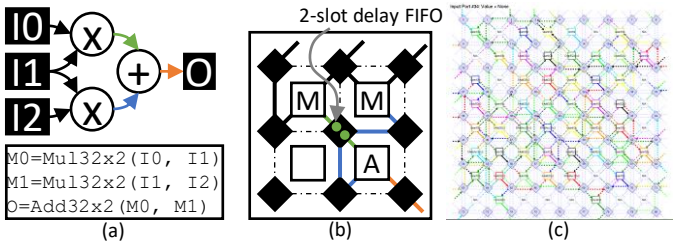

Figure 7: Mapping Dataflow Graphs (DFGs) to CSCA. (a) Pictorial representation and code representation of a DFG. (b) Mapped DFG to small CSCA. (c) Full mapping on Mozart's large CSCA. Colors represent different DFG edges.

which was mostly straightforward software engineering. We use a modified TensorFlow v1 for our frontend, adding parsing code for networks, device specific graph transformations, our own quantization/calibration flow, high-speed runtime, and low-level library toolchain. More than 90% of our software development time went into the above implementation, even though none of it was a new or hard problem.

To map operators to Mozart, we implemented an operator library with our low-level toolchain, which plugged into Tensorflow, to support execution of DL applications on the Mozart platform. This involved a manual approach to implementing TF-level operations on the RED ISA. Even NVIDIA, which has the most mature software stack in this space, uses a mix of hand-generation and auto-tuning [5, 44].

Kernel Compilation and Runtime. The kernel-level compilation approach is similar to prior work on stream-dataflow [41]. Kernels are written in C/C++, with intrinsics to specify RED commands, and DFGs to specify the computation for CSCA. For compiling the C/C++ program, we use RISCV GCC and extended its assembler with RED specific instructions. The "scheduler" compiles each programmer-specified DFG to the CSCA, and the configuration bits are embedded into the RISC-V binaries. Finally, on the host side, a low level management library communicates with a device driver to facilitate host-accelerator memory movement and kernel launch/synchronization.

DFG Scheduler. The scheduler is the key intellectual aspect on the software side. The scheduler's role is to map DFGs to the CSCA hardware: assigning instructions onto processing elements, routing dependences onto dedicated network paths, and matching the timing of operands for functional-units and ports (due to static scheduling). Figure 7 shows example mappings.

The scheduler uses a stochastic-search based approach that iteratively refines the schedule. The scheduler's performance was initially an issue due to the size and connectivity of the CSCA. Three innovations enabled fast scheduling. The first was allowing the scheduler to make certain kinds of mistakes during scheduling (i.e. overprovisioning a network or routing resource), where those mistakes are constrained and still permit analysis of "how good" the schedule is. This enables iterative refinement even when the intermediate schedule does not satisfy all hardware constraints. The second was using a consistent global objective function for all scheduling decisions rather than custom (often faster to compute) heuristics for each decision. This enabled better decision-making, e.g. mapping decisions take into account their impact on routing and timing. The third was related to delay-matching: matching the arrival time of data items for each hardware element. Delay-matching is critical for performance [40], and often there were overly-tight operand-timing paths (e.g. for controlling accumulator reset); therefore we developed a heuristic to insert "loops" to *lengthen* such paths. Integrating these enabled a scheduler that ran in seconds and did not sacrifice programmer productivity.

Table 1: Toolchain Results and Full application performance in terms of inf/sec/watt normalized to NVIDIA T4. Mozart power is 75W. Mozart++ 7nm is projected performance.

	FP32		Mozart Opt.		Performance		
Model	**Ops**	**Uniq**	**Ops**	**Uniq**	**A100**	**Mozart**	**Mozart++**
RN50	461	14	84	12	0.96x	3.3x	8.4x
SSD	3320	46	111	10	1.4x	1.9x	4.9x
BERT	757	24	322	17	1.63x	7.8x	20.1x
DLRM	-	-	-	-	0.5x	0.63x	3.52x
RNNT	-	-	-	-	1.5x	11.4x	29.5x

Overall Efficacy. The primitives in the architecture being closely aligned with AI behaviors enabled software programmers with little knowledge of hardware, ML, or linear algebra to produce high-performance code in 2-4 weeks.

Also, while not our primary goal, we remark that the Mozart compiler flow is fast. For example, starting with a Tensorflow frozen model as input on which calibration has been done for quantization, our compiler produces quantized models with object code in 3-29 seconds.

6 QUANTITATIVE CHARACTERIZATION

Performance. First, we discuss overall performance by comparing MLPerf application performance against the state-of-the-art, which is NVIDIA A100, at batch-size 4. Table 1 shows the results with the NVIDIA T4 as the baseline[5]. Since the A100 has a substantial technology advantage, we also report some simulation results of a Mozart++ 7nm chip that includes some microarchitecture optimizations as well (the final row in Table 4). In general, Mozart is integer factors better, even while being nearly 2 technology nodes behind. On DLRM, we are limited by PCIe bandwidth and software/algorithm optimization because of the nature of the workload. This DLRM issue of very large tables and a somewhat simple MLP for the computation, requiring memory layout and algorithm optimization, applies to other accelerators as well. On SSD-Resnet, the large layers at the start end up being conducive to efficiency on the GPU, providing ample embarrassing parallelism even at small batch-size. At very large batch-sizes, the gap between Mozart and a GPU are likely to be small. Our overall point is that such a latency-optimized design targeting small-batch efficiency could trigger new types of training algorithms, and is a better candidate for data-center and edge *inference*.

To get some insights on the RED architecture, microarchitecture, and application interplay, we examine two well understood operators that dominate CNNs and Transformers: convolution and

[5]We understand there may be some more recent results from NVIDIA on the A100 results. These are results as of late 2020.

Table 2: Chip Utilization for various ML operators.

Op-Type	Input-0/Input-1/Output Shape	% Util
AddRR	[56,56,256]/[56,56,256]/[56,56,256]	20%
AddRR	[14,14,1024]/[14,14,1024]/[14,14,1024]	13%
Conv2DBR	[x,14,14,256]/[1024,1,1,256]/[14,14,1024]	27%
Conv2DBR	[x,28,28,128]/[512,1,1,128]/[28,28,512]	32%
Conv2DBRR	[x,30,30,256]/[256,3,3,256]/[x,14,14,256]	9%
Conv2DBRR	[x,56,56,256]/[128,1,1,256]/[x,56,56,128]	44%
MatmulBR	[128,768]/[3072,768]/[128,3072]	34%
MatmulBRRT	[128,768]/[768,768]/[1,12,128,64]	13%
SoftMax	[1,12,128,128]/[]/[1,12,128,128]	9%

RR: RequantizeRelu; BR: BiasAddRequantize; BRR: BiasAddRequantizeRelu; BRRT: BiasAddRequantizeReshapeTranspose.

matmul. In a DL-application, they appear with different tensor shapes and are fused with different operators, imputing various needs on the hardware. Table 2 shows a representative sample that covers almost 60% of execution cycles, including easy and hard shapes, to showcase this diversity. By exploiting available reuse, we sustain around 45% utilization running at an extreme case of single-batch inference while considering all hardware latencies and contention issues. The primary micro-architecture-application behavior interactions that prevent 100% utilization are: i) fill and drain cycles for a tile to achieve steady-state, which involves core-bootstrapping, CSCA configuration etc.; ii) time taken for loading input tensors into local scratchpad is not fully hidden or overlapped with compute cycles, iii) special handling of boundary conditions for non-standard tensor shapes; iv) read streams' L2-Cache cold misses and servicing; v) write streams' low-cache utilization because of pollution/evictions, poor locality and other bottlenecks; vi) vector-port balancing overheads and stream switching control penalty. Utilization on a reuse-fits-on-chip mode (i.e. models that are L2 cache resident), is typically around 90%.

The takeaways from this performance analysis are: i) the RED architecture is able to successfully implement applications unknown at arch/design time (BERT's invention after architecture freeze), ii) we can extract high-performance using techniques like operator fusion, batching kernels and prefetching to maximize reuse, and iii) we can perform arbitrary specialized computation like Softmax, ERF and others using the CSCA array without any hardware functional unit specialization (while more fixed approaches like NVIDIA's tensor-cores cannot).

Power and Area. We now provide details on overall chip power and area. We note here that with RED being a new architecture and Mozart being its first implementation, power-aware microarchitecture design and RTL implementation was known to be a challenge — and we made a conscious decision that we were not going to be aggressive in power optimization, with the insight that the architecture provides high power efficiency to begin with. As a result, the breakdown of power should be considered as a starting point. Post-tapeout, with further design and implementation optimizations, power and area were reduced by more than a factor of 2 (see next subsection). Overall, the tiles consumed nearly 90% of the total power and area, and the CSCA was roughly 50% of chip power and 42% of chip area. Breakdown within a tile and within Softbrain is

Table 3: Power And Area Breakdown

Module	% Power	% Area
NetworkController	3.1%	5.2%
TileController	7.4%	0.3%
Uncore	17.1%	22.4%
Von-Neumann Engine	5.9%	10.6%
Softbrain	66.5%	61.5%
Softbrain breakdown		
CSCA	51.6%	42.5%
MSE	3.5%	3.3%
SSE	3.1%	7.4%
IVP	2.5%	5.2%
OVP	1.1%	1.8%
BALANCE-UNIT	2.3%	0.8%
RSE	0.5%	0.3%
STREAM-DISPATCHER	0.2%	0.2%

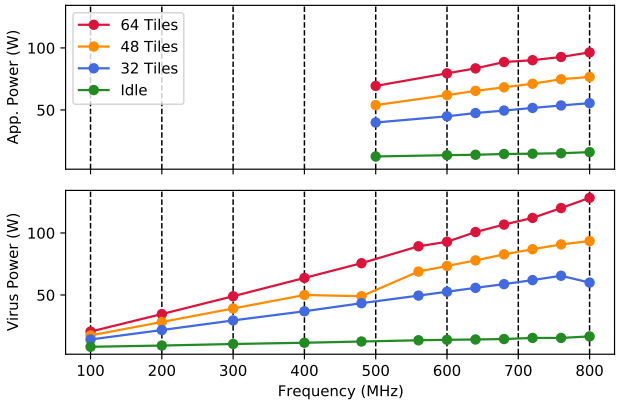

Figure 8: Power validation and peak power

shown in Table 3. The design ended up having a substantially larger ratio of flip-flops to gates than we expected, because of various physical design constraints that would have been best solved with a microarchitecture revision (but tapeout time constraints did not allow this) leading to some power inefficiency. Figure 8 shows the results of our power validation at the chip level, running a typical application and our synthetic power-virus used for TDP estimation and package design. Roughly, we see a linear increase with frequency (we did not implement DVFS as energy optimization was not a first order constraint for Mozart).

Sensitivity studies. We conclude with a summary of two sensitivity studies that highlight the RED architecture's potential and what implementations different from Mozart could achieve. Each row in Table 4 corresponds to a design feature change (removal or addition) with detailed post-tapeout area estimation (we expect the final Silicon area to be within 2% to 4% of this number). Through successive simplifications, and then adding optimizations, we can achieve a nearly 2X reduction in area and power savings, **with the same performance**. Additional research ideas could improve this even further, growing RED's advantage over conventional architectures. Second, a hybrid DLRM accelerator based on RED using LPDDR5 was designed based on how most of the tables are unused and a very small portion have hot traffic [2, 33]. We were able to

Table 4: Sensitivity of Mozart Microarchitecture

Version	Area (mm^2)	Normalized Area
Mozart POR	1.69	1
(-) Only Mul and Add opcodes	1.18	0.7
(-) No Non-linear Op	1.11	0.66
(-) Reduced CSCA connectivity	1.04	0.62
(+) 2x L2-Cache size	1.08	0.64
(+) 2x Scratchpad size	1.12	0.66
(-) Optimized uncore	1.01	0.6
(-) Reduced L1-cache size	0.97	0.57
(-) Optimized stream engines	0.94	0.56
(-) Total delay fifo depth = 8	0.9	0.53
(-) Total delay fifo depth = 0	0.87	0.51
(-) Replacing Mesh with a ring	0.82	0.49
(-) Software controlled L2-cache	0.75	0.44
(+) Add more opcodes	0.88	0.52
(+) Optimized CSCA network	0.89	0.53
(+) Full CSCA network	0.93	0.55
(+) Mesh Network for tile	0.96	0.57
(+) Delay fifo = 8	0.99	0.59
(+) Non-linear op back in	1	0.59

obtain a 2.6X speedup over an A100 and a nearly 8X improvement in throughput/watt (at the same latency) in a small M.2 form-factor chip, showing the RED architecture is able to scale along many dimensions, including large and sparse data-analytics problems.

We briefly comment on non-DL workloads we can support. This includes blockchain processing (particularly for memory-hardened protocols like Ethereum) — essentially, the stream abstractions and flexible DFGs allows the architecture to traverse the tree structure needed for Ethereum [23, 45], traditional genome sequence alignment like BWA-MEM [56], some forms of SSL-offload, where each tile does an independent network query, and gradient boosting trees for ML [7].

7 LESSONS, EXPERIENCES AND CHALLENGES AHEAD

7.1 Architecture Lessons

Behavior oriented principles are needed as primitives for new architectures. The Reuse Exposed Dataflow architecture provides high generality, efficiency and software friendliness. Overall, the key motivation of the architecture was to expose the reuse behavior and achieve as much low-latency execution as possible without **needing any** embarrassing parallelism in the form of large batch size, which we achieved. In practice, we were able to support many applications, unknown during architecture design time. For example, the architecture was frozen well before transformers [55] became mainstream, or the RNN-T paper's [25] publication. We had not had the internal resources to analyze LSTMs in depth. Since we had architecturally exposed primitives for AI behaviors; this allowed running the applications that arrived post-architecture finalization. Specifically, the operator-fusion and kernel-batching operations that eliminated the overheads of low-reuse element-wise operators were implemented purely with software (See section 5). The architecture was able to quite easily support a variety of DL and non-DL algorithms, including SSL, graph processing, and sequence alignment, and extreme DL workloads like DLRM. *A behavior-oriented ISA design philosophy is key to staying relevant in the face of rapid DNN evolution.*

Perf Specs must be communicated in a programmer-accessible way. Even very well understood architectures like SIMD/FPGA have failed in capturing performance in a way that is intuitive and well communicated to the programmer. AVX2, for example, relies on 72-pages of a performance manual [20, 29], FPGAs rely on informal guidelines that are hard-to-follow for software developers. Our concise pictorial form for RED ISA and Mozart worked, but it's just a start. We believe that achieving an intuitive performance specification demonstrates that we got the architecture "right" in balancing the role of programmer, compiler, and hardware which is as much art, science and luck.*The broader takeaway is that future architectures and chips must grapple with such performance specification — it's a place for academics to formalize with deeply thought-out principles and whether DSLs/compilers could obviate such a need.*

7.2 Microarchitecture Lessons

Microarchitecture surprisingly complex and new. The amount of micro-architecture invention, design, and specification we needed to do was surprising. With the long history of explicit dataflow machines in academia, we had felt the basic stream-dataflow design would be relatively quick to convert into an implementation. As we embarked on the detailed micro-architecture, we realized this space is not as well understood, and had to invent substantial components to get a working, high performance micro-architecture. Table 4 (section 6) shows 19 micro-architecture configurations we explored — we have by no stretch of imagination done justice to policies/parameters within those, or how these really interact. As another example, the compute pipeline setup latency to start execution of a graph was nearly 40 cycles; for loops that executed for as little as 200 cycles typically, this setup latency became an unexpected 1st order performance constraint. A fairly "simple" micro-architecture change in terms of out-of-order command execution with modifications to stream-dispatcher/IVP/OVP module's interface and efficient handshake mechanisms cut this down to 4 cycles, providing even more flexibility for programmer freedom. *The microarchitecture (various flavors) of dataflow machines is a rich unexplored design space. We say this with a bit of tongue in cheek: we think there are 100s of papers to write on address generators for streams, balance unit possibilities, stream engine design, vector port interface, much like the 100s of papers on branch prediction, cache policies, MSHR design, LSQ design etc. Especially with conventional cores being harder to transmogrify for future data-analytic workloads, and GPU speedups slowing down, architectures like RED that provide equally good software maturity, but much higher performance, could become essential to overcome the slowing of technology scaling while also retaining software productivity.*

Better cache/scratchpad to manage pollution would improve performance. Hybrid memory designs that support a blend of caching, prefetch, and scratchpad impute a complicated interplay of cache pollution and effectiveness with bandwidth, capacity, and algorithmic-reuse. Cache-management instructions in state-of-the-art high-end CPUs could be improved further (even with careful prefetching, Intel CPUs for example can use effectively half the cache when supporting well-behaved tensors for convolution [21]). We suffered some substantial performance degradation because we did not have hybrid scratchpad mode in the L2-cache. *In retrospect, we would have liked to mark certain addresses as evict-last in the L2 memory to manage cache pollution.*

7.3 Chip Implementation Lessons

Accurate power estimates at design time were hard since the microarchitecture was clean-slate. Power estimates at the concept stage were quite hard and proved to have almost 2X overall, and sometimes 5X error for some submodules. There are two reasons for this — first, some of the SRAM modeling mentioned above is not well handled by tools like CACTI [4]. Second, in a clean-slate architecture, it is hard to estimate the microarchitecture complexity, and we deferred power-optimization to the next chip. Consequently, our flip-flop to gate ratio was quite a bit different from a mature CPU's design, against which McPAT-like tools are calibrated [36]. This mismatch led to the high errors. However, we always felt the power numbers from our early modeling seemed quite aggressive whenever we compared our modeling power to an NVIDIA GPU's TDP. *Microarchitecture/CAD research explorations that better formalize the complexity impact on frequency/power/area would have built confidence for us to take on power-optimization also in the first implementation.*

Interconnect design unexpectedly complicated requiring a substantial amount of clean-state work. We had expected to be able to use an off-the-shelf commercial NoC. Our need of a heterogeneous NoC with 2D mesh virtual channels and a wider ring was not available, leaving us with the design, implementation, verification, and floorplanning and some PD tasks. Also, the low-frequency serial network we had to build to interconnect configuration register and low-speed peripherals was unexpected. Turn-key solutions did not play nice with other IP blocks and had odd IP inter-dependencies.

Off-the-shelf datapath modules still surprisingly unoptimized for power and area. We were surprised to find there was a decent amount of headroom in improving the PPA of basic things like adders, multipliers, FP16 blocks and compound functional units. DL-tuned arithmetic modules are an unexpected low-hanging fruit for architecture academic research as well, which we and others are pursuing [10, 17, 30, 54].

7.4 Software Stack Lessons

Software lift needed on non-novel software pieces is very large. The amount of code we needed to write to bringup the toolchain was surprisingly large (from informal discussions and looking at job positings, this is true in at-least 3 other chip-unicorn startups). Existing frameworks were bloated, and nearly impossible to disaggregate, missing a non-trivial number of features. For example, Halide [47] was nice, but there wasn't a way to cleanly roll it up into our flow. Similarly, TVM [8] looked nice conceptually, but TensorFlow provides everything we needed except TVM's automated mapper, and in the context of providing model parsing, quantization, operator fusion, and lowering into our operator library, with dynamically managed memory, it did not provide value for us. We needed to develop a surprising amount of domain knowledge to write the SW toolchain, including (in parallel with other companies) inventing the concepts of operator fusion, merging element-wise operators into other operators to optimize for memory bandwidth, and co-optimizations for performance and accuracy. On a positive note, the design of the RED architecture allowed us to contain these cross-cutting concerns; keeping them from affecting the architecture or implementation.

Spatial schedulers have several unsolved research problems. Our scheduler eventually needed heuristics to be programmer friendly, as ILP [40, 43] was not fast enough for the CSCA size, connectivity, and static-timing requirement. However, the heuristic-based scheduler took significant development time; before we completed it, the daunting uncertainty over the scheduler slowed codesign decisions, like what radix switch to use, or what CSCA topology and size. A fast, scalable, declarative scheduler would have a high value. Revisiting the capability of synthesis/optimization techniques to address mapping issues in emerging architectures could provide huge benefits in productive software toolchains. Recent examples like NVIDIA's DSL [5], CoSA which looks at pure static machines [28], and ML-based scheduling [37] address some aspects of this. Many unsolved problems remain open to architects, especially when considering static/dynamic hybrids like RED and GPUs.

8 RELATED WORK

Through the recent resurgence of hardware architecture for AI [48], most solutions are matrix-engine based [19] (Graphcore, Habana, Baidu Kunlun, Alibaba HanGuang, Huawei Ascend). Qualcomm's AI-100 seems to be based on VLIW DSPs. Google's TPU uses systolic arrays [31, 32]. Groq [1] uses an extremely static approach with sliced functional units (like an uber-FPGA), while Xilinx is taking FPGAs and adding matrix engines like structures to it with Versal. Sambanova implements a chip-scale CGRA with more restrictive thread and address space model than RED [61]. With the MI100, AMD has added matrix-engines to its GPUs, and in terms of raw performance and perf/watt it seems to rival the A100.

None of these seem to have addressed the software friendliness, basic usability, and hardware efficiency challenge yet, with the exception of NVIDIA (evolving CUDA over decades [11]) and Google's TPU (presumably benefiting from organic Tensorflow evolution). Mozart introduces a completely new architecture from the ground-up, rethinking the role of computation, communication, and storage for the new generations of data-intensive workloads, while being cognizant that supporting the deep learning software stack is a necessity.

In academia, while many purpose-built accelerators [6, 9, 24, 46]

have been proposed, including recent efforts targeting sparsity [18], they pay insufficient attention to software integration or rapid workload evolution, leading to quick obsolescence, alluded to by [26]. Mittal et al. [39] presents a survey of deep-learning on CPUs where the SW lift is ameliorated somewhat. Cambricon [38] is of interest, since it exposes linear algebra concepts as architecturally supported primitives — similar to how RED exposes new primitives at the ISA level, but lacks a performance contract between the implementation and programmer. In contrast, Mozart's RED instruction set has a concrete performance contract, which allows for verifiably high-performance code to be authored with relative ease. Our results show that a more general approach enabled by a clean-slate architecture is quite attractive, and we hope to steer more academic research into architecture, microarchitecture, and software ideas in this space.

Mozart focuses on expressive inter-core data reuse for stream-dataflow and defines the Reuse Exposed Dataflow ISA. Other works also build on stream-dataflow, including for supporting irregular memory [16], control [60], and parallelism [14, 15]. TaskStream [15] extends the ISA for task parallelism, enabling dynamic reordering of tasks to exploit opportunities for multicasting data shared between tasks. Prior work also adds stream abstractions to CPU ISAs [57–59]; the "stream confluence" optimization [59] enables recognizing simultaneous reuse across multiple cores and combines streams dynamically to reduce requests to shared cache and reduce traffic by multicasting. Overall, the realization of Mozart lends credence to the practicality of adopting these ideas in industry.

9 CONCLUSIONS

The RED architecture and Mozart implementation effectively targets low-latency with the ability for high performance without any batch parallelism (i.e. small batch size is enough) and provides a unique and disruptive design point to focus on. At small-batch, as shown in the microarchitecture section, we feel there is much room to innovate and push performance and efficiency even higher. Conversely, when abundant parallelism is available (large-batch training/inference), other uncore effects dominate, making savings from core micro-architecture insignificant, and the GPU is hard to beat. Indeed, DL has large data-level parallelism at the sample level [53]. However, this comes at an exorbitant demand on memory capacity and bandwidth — among DL/ML scientists, often multiple GPUs are used just for extra memory storage during training, resulting from doing large-batch training.

Architecture and hardware optimized for high utilization at *small-batch* can provide disruptively new ways to reduce the dollar-cost and power of DL (inference and training). For *large-batch*, TensorCore/MatrixEngines for DL are analogous to the mature OOO pipeline for CPUs — we feel it is not the place to compete for a startup, as their "demonstrable unfair advantage" [49] is negligible.

The overarching lesson for us is that a small design team with only 20 full-time team members was sufficient to build a high-performance *clean-slate architecture* on a leading edge technology node. The implementation of the chip demonstrates the effectiveness of the architecture in delivering on its goals of software-maturity and high-performance.

ACKNOWLEDGMENTS

The authors would like to thank the entire SimpleMachines team. We also thank the reviewers, Newsha Ardalani, Michael Pellauer, Joshua San-Miguel, Matt Sinclair, and David Wood for feedback on this paper.

REFERENCES

[1] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, Jennifer Hwang, Rebekah Leslie-Hurd, Michael Bye, E.R. Creswick, Matthew Boyd, Mahitha Venigalla, Evan Laforge, Jon Purdy, Purushotham Kamath, Dinesh Maheshwari, Michael Beidler, Geert Rosseel, Omar Ahmad, Gleb Gagarin, Richard Czekalski, Ashay Rane, Sahil Parmar, Jeff Werner, Jim Sproch, Adrian Macias, and Brian Kurtz. 2020. Think Fast: A Tensor Streaming Processor (TSP) for Accelerating Deep Learning Workloads. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 145–158. https://doi.org/10.1109/ISCA45697.2020.00023

[2] Ehsan K. Ardestani, Changkyu Kim, Seung Jae Lee, Luoshang Pan, Valmiki Rampersad, Jens Axboe, Banit Agrawal, Fuxun Yu, Ansha Yu, Trung Le, Hector Yuen, Shishir Juluri, Akshat Nanda, Manoj Wodekar, Dheevatsa Mudigere, Krishnakumar Nair, Maxim Naumov, Chris Peterson, Mikhail Smelyanskiy, and Vijay Rao. 2021. Supporting Massive DLRM Inference Through Software Defined Memory. arXiv:2110.11489 [cs.AR]

[3] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator.* Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[4] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Trans. Archit. Code Optim.* 14, 2, Article 14 (jun 2017), 25 pages. https://doi.org/10.1145/3085572

[5] Somashekaracharya G. Bhaskaracharya, Julien Demouth, and Vinod Grover. 2020. Automatic Kernel Generation for Volta Tensor Cores. arXiv:2006.12645 [cs.PL]

[6] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 269–284. https://doi.org/10.1145/2541940.2541967

[7] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An Automated {End-to-End} Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[9] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2019. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 2 (2019), 292–308. https://doi.org/10.1109/JETCAS.2019.2910232

[10] Sung-Gun Cho, Wei Tang, Chester Liu, and Zhengya Zhang. 2021. PETRA: A 22nm 6.97TFLOPS/W AIB-Enabled Configurable Matrix and Convolution Accelerator Integrated with an Intel Stratix 10 FPGA. In *2021 Symposium on VLSI Circuits*. 1–2. https://doi.org/10.23919/VLSICircuits52068.2021.9492517

[11] Don Clark. 2017. Why a 24-Year-Old Chipmaker Is One of Tech's Hot Prospects. *New York Times* (September 2017).

[12] Henry Cook, Wesley Terpstra, and Yunsup Lee. 2017. Diplomatic design patterns: A TileLink case study. In *1st Workshop on Computer Architecture Research with RISC-V*.

[13] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (oct 1991), 451–490. https://doi.org/10.1145/115372.115320

[14] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. 2021. PolyGraph: Exposing the Value of Flexibility for Graph Processing Accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 595–608. https://doi.org/10.1109/ISCA52012.2021.00053

[15] Vidushi Dadu and Tony Nowatzki. 2022. TaskStream: Accelerating Task-Parallel

Workloads by Recovering Program Structure. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3503222.3507706

[16] 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)*. Association for Computing Machinery, New York, NY, USA, 924–939. https://doi.org/10.1145/3352460.3358276

[17] Bita Darvish Rouhani, Daniel Lo, Ritchie Zhao, Ming Liu, Jeremy Fowers, Kalin Ovtcharov, Anna Vinogradsky, Sarah Massengill, Lita Yang, Ray Bittner, Alessandro Forin, Haishan Zhu, Taesik Na, Prerak Patel, Shuai Che, Lok Chand Koppaka, XIA SONG, Subhojit Som, Kaustav Das, Saurabh T, Steve Reinhardt, Sitaram Lanka, Eric Chung, and Doug Burger. 2020. Pushing the Limits of Narrow Precision Inferencing at Cloud Scale with Microsoft Floating Point. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 10271–10281. https://proceedings.neurips.cc/paper/2020/file/747e32ab0fea7fbd2ad9ec03daa3f840-Paper.pdf

[18] Shail Dave, Riyadh Baghdadi, Tony Nowatzki, Sasikanth Avancha, Aviral Shrivastava, and Baoxin Li. 2021. Hardware Acceleration of Sparse and Irregular Tensor Computations of ML Models: A Survey and Insights. *Proc. IEEE* 109, 10 (2021), 1706–1752. https://doi.org/10.1109/JPROC.2021.3098483

[19] Jens Domke, Emil Vatai, Aleksandr Drozd, Peng Chen, Yosuke Oyama, Lingqi Zhang, Shweta Salaria, Daichi Mukunoki, Artur Podobas, Mohamed Wahib, and Satoshi Matsuoka. 2021. Matrix Engines for High Performance Computing:A Paragon of Performance or Grasping at Straws?. In *IEEE International Parallel and Distributed Processing Symposium*.

[20] Junfeng Dong, John Morgan, and Li Tian. 2019. Accelerating Compute-Intensive Workloads with Intel AVX-512. https://devblogs.microsoft.com/cppblog/accelerating-compute-intensive-workloads-with-intel-avx-512/

[21] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. 2018. Anatomy of High-Performance Deep Learning Convolutions on SIMD Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas) *(SC '18)*. IEEE Press, Article 66, 12 pages. https://doi.org/10.1109/SC.2018.00069

[22] DFI Group. 2021. *DFI Specification*. http://www.ddr-phy.org/page/page/show?id=2351641%3APage%3A301

[23] R. Han, N. Foutris, and C. Kotselidis. 2019. Demystifying Crypto-Mining: Analysis and Optimizations of Memory-Hard PoW Algorithms. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE Computer Society, Los Alamitos, CA, USA, 22–33. https://doi.org/10.1109/ISPASS.2019.00011

[24] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. IEEE Press, 243–254. https://doi.org/10.1109/ISCA.2016.30

[25] Yanzhang He, Tara N. Sainath, Rohit Prabhavalkar, Ian McGraw, Raziel Alvarez, Ding Zhao, David Rybach, Anjuli Kannan, Yonghui Wu, Ruoming Pang, Qiao Liang, Deepti Bhatia, Yuan Shangguan, Bo Li, Golan Pundak, Khe Chai Sim, Tom Bagby, Shuo-yiin Chang, Kanishka Rao, and Alexander Gruenstein. 2019. Streaming End-to-end Speech Recognition for Mobile Devices. In *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 6381–6385. https://doi.org/10.1109/ICASSP.2019.8682336

[26] Sara Hooker. 2021. The Hardware Lottery. *Commun. ACM* 64, 12 (nov 2021), 58–65. https://doi.org/10.1145/3467017

[27] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2014. Heterogeneous-Race-Free Memory Models. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 427–440. https://doi.org/10.1145/2541940.2541981

[28] Qijing Huang, Minwoo Kang, Grace Dinh, Thomas Norell, Aravind Kalaiah, James Demmel, John Wawrzynek, and Yakun Sophia Shao. 2021. CoSA: Scheduling by Constrained Optimization for Spatial Accelerators. In *ISCA*.

[29] Intel. 2022. Intel 64 and IA-32 Architectures Optimization Reference Manual, Chapter 15. https://www.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html

[30] Jeff Johnson. 2018. Rethinking floating point for deep learning. arXiv:1811.01721 [cs.NA]

[31] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. 2021. Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 1–14. https://doi.org/10.1109/ISCA52012.2021.00010

[32] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. 2020. A Domain-Specific Supercomputer for Training Deep Neural Networks. *Commun. ACM* 63, 7 (jun 2020), 67–78. https://doi.org/10.1145/3360307

[33] Hongju Kal, Seokmin Lee, Gun Ko, and Won Woo Ro. 2021. SPACE: Locality-Aware Processing in Heterogeneous Memory for Personalized Recommendations. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 679–691. https://doi.org/10.1109/ISCA52012.2021.00059

[34] Yunsup Lee, Andrew Waterman, Henry Cook, Brian Zimmer, Ben Keller, Alberto Puggelli, Jaehwa Kwak, Ruzica Jevtic, Stevo Bailey, Milovan Blagojevic, et al. 2016. An agile approach to building RISC-V microprocessors. *ieee Micro* 36, 2 (2016), 8–20.

[35] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2021. The Deep Learning Compiler: A Comprehensive Survey. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (Mar 2021), 708–727. https://doi.org/10.1109/tpds.2020.3030548

[36] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, New York) *(MICRO 42)*. Association for Computing Machinery, New York, NY, USA, 469–480. https://doi.org/10.1145/1669112.1669172

[37] Zhaoying Li, Dan Wu, DM Dhananjaya Wijerathne, and Mitra Tulika. 2022. LISA: Graph Neural Network Based Portable Mapping on Spatial Accelerators. In *2022 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[38] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. 2016. Cambricon: An Instruction Set Architecture for Neural Networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 393–405. https://doi.org/10.1109/ISCA.2016.42

[39] Sparsh Mittal, Poonam Rajput, and Sreenivas Subramoney. 2021. A Survey of Deep Learning on CPUs: Opportunities and Co-Optimizations. *IEEE Transactions on Neural Networks and Learning Systems* (2021), 1–21. https://doi.org/10.1109/TNNLS.2021.3071762

[40] Tony Nowatzki, Newsha Ardalani, Jian Weng, and Karthikeyan Sankaralingam. 2018. Hybrid Op-timization/Heuristic Instruction Scheduling for Programmable Accelerator Codesign. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*.

[41] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-Dataflow Acceleration. In *Proceedings of the 44th International Symposium on Computer Architecture*.

[42] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. 2015. Exploring the Potential of Heterogeneous Von Neumann/Dataflow Execution Models. In *Proceedings of the 42nd International Symposium on Computer Architecture*.

[43] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. 2013. A General Constraint-centric Scheduling Framework for Spatial Architectures. In *Proceedings of 34th International Conference on Programming Language Design and Implementation. Distinguished Paper Award. SIGPLAN Research Highlights Nominee.*

[44] NVIDIA. 2021. *CUTLASS 2.8*. https://github.com/NVIDIA/cutlass

[45] Vijay Pradeep. 2017. Ethereum Memory Hardness Explained. https://www.vijaypradeep.com/blog/2017-04-28-ethereums-memory-hardness-explained

[46] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark Horowitz. 2015. Convolution Engine: Balancing Efficiency and Flexibility in Specialized Computing. *Commun. ACM* 58, 4 (mar 2015), 85–93. https://doi.org/10.1145/2735841

[47] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[48] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. 2020. Survey of Machine Learning Accelerators. *2020 IEEE High Performance Extreme Computing Conference (HPEC)* (Sep 2020). https://doi.org/10.1109/hpec43674.2020.9286149

[49] Eric Ries. 2011. *The Lean Startup*. Currency.

[50] Karthikeyan Sankaralingam, Vinay Gangadhar, Anthony Nowatzki, and Yunfeng Li. 2021. Method, computer program product, and apparatus for acceleration of simultaneous access to shared data. https://patents.google.com/patent/US10963384B2/en?oq=US10963384B2

[51] Karthikeyan Sankaralingam, Yunfeng Li, Vinay Gangadhar, and Anthony Nowatzki. 2020. Accelerating parallel processing of data in a recurrent neural network. https://patents.google.com/patent/US20200218965A1/en?oq=US2020218965A1

[52] Karthikeyan Sankaralingam, Anthony Nowatzki, Vinay Gangadhar, Preyas Shah,

and Newsha Ardalani. 2021. Systems and methods for stream-dataflow acceleration wherein a delay is implemented so as to equalize arrival times of data packets at a destination functional unit. https://patents.google.com/patent/US11048661B2/en?oq=US11048661B2

[53] Christopher Shallue, Jaehoon Lee, Joe Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George Dahl. 2018. Measuring the Effects of Data Parallelism on Neural Network Training. *Journal of Machine Learning Research* (11 2018).

[54] TESLA. 2021. *Tesla Dojo Technology — A Guide to Tesla's Configurable Floating Point Formats Arithmetic*. https://tesla-cdn.thron.com/delivery/public/document/tesla/bc895d60-8220-4323-a5ba-e21452d786c0/bvlatuR/WEB/tesla-dojo-technology

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[56] Thiruvengadam Vijayaraghavan, Amit Rajesh, and Karthikeyan Sankaralingam. 2018. MPU-BWM: Accelerating Sequence Alignment. *IEEE Computer Architecture Letters* 17, 2 (2018), 179–182. https://doi.org/10.1109/LCA.2018.2849064

[57] Zhengrong Wang and Tony Nowatzki. 2019. Stream-Based Memory Access Specialization for General Purpose Processors. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 736–749. https://doi.org/10.1145/3307650.3322229

[58] Zhengrong Wang, Jian Weng, Sihao Liu, and Tony Nowatzki. 2022. Near-Stream Computing: General and Transparent Near-Cache Acceleration. *HPCA. https://seanzw. github. io/pub/hpca2022-near-stream-computing. pdf* (2022).

[59] Zhengrong Wang, Jian Weng, Jason Lowe-Power, Jayesh Gaur, and Tony Nowatzki. 2021. Stream Floating: Enabling Proactive and Decentralized Cache Optimizations. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 640–653. https://doi.org/10.1109/HPCA51647.2021.00060

[60] Jian Weng, Sihao Liu, Zhengrong Wang, Vidushi Dadu, and Tony Nowatzki. 2020. A Hybrid Systolic-Dataflow Architecture for Inductive Matrix Algorithms. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 703–716. https://doi.org/10.1109/HPCA47549.2020.00063

[61] Bob Wheeler. 2021. SambaNova Takes On Nvidia's DGX. (Feb 2021).