UpDown: Efficient Manycore based on Many Threading and Scalable Memory Parallelism

Andronicus Rajasukumar
University of Chicago
Chicago, USA
andronicus@uchicago.edu

Ruiqi Xu
University of Chicago
Chicago, USA
ruiqix@uchicago.edu

Tianchi Zhang
University of Chicago
Chicago, USA
tonyztc@uchicago.edu

Yuqing (Ivy) Wang
University of Chicago
Chicago, USA
yqwang@uchicago.edu

Tianshuo Su*
Google LLC
Mountain View, USA
tssu@google.com

Marziyeh Nourian
University of Chicago
Chicago, USA
marziyeh.nourian@gmail.com

Jianru Ding
University of Chicago
Chicago, USA
jrding@uchicago.edu

Jiya Su
University of Chicago
Chicago, USA
jiya@uchicago.edu

Rajat Khandelwal
University of Chicago
Chicago, USA
rajatkha@uchicago.edu

Alexander Fell
University of Chicago
Chicago, USA
alefel@uchicago.edu

David F. Gleich
Purdue University
West Lafayette, USA
dgleich@purdue.edu

Yanjing Li*
Northeastern University
Oakland, USA
yanj.li@northeastern.edu

Henry Hoffmann
University of Chicago
Chicago, USA
hankhoffmann@cs.uchicago.edu

Andrew A. Chien
University of Chicago
Chicago, USA
Argonne National Laboratory
Chicago, USA
aachien@uchicago.edu

Abstract

Manycore architectures are a promising direction for single-chip performance. They typically use in-order cores and caches as building blocks and can produce good performance on regular applications. However, on irregular applications, they have low core utilization due to data-dependent control-flow and memory access.

We propose UpDown manycore that employs a novel core with Event-Driven Scheduling (EDS), Software-controlled Lightweight Threading (SLT), and Split-transaction Memory Access with software synchronization (SMA) to deliver both high core utilization and chip performance. On a variety of graph and sparse applications, UpDown outperforms a much larger commercial multicore chip (20-core, OoO) by up to 81x.

Compared to simple, in-order cores, the UpDown core's novel mechanisms provide a 2.4-5.9x performance advantage, specifically 1.9x (EDS), 1.4x (SLT), and 1.4x (SMA). For a manycore chip, these architectural benefits enable a 2048-core UpDown to outperform an 8192-core in-order chip of similar Si area by 3.1x overall. By efficiently scheduling computation from many parallel threads, UpDown achieves high core utilization. Further, UpDown mechanisms enable it to more effectively exploit the growing bandwidth available from HBMs and tolerate higher NoC latencies, supporting future scaling. Finally, we also show that UpDown achieves better performance than a state-of-the-art GPU normalized by area.

CCS Concepts

• **Computer systems organization → Multiple instruction, multiple data**; **Multicore architectures**.

Keywords

Multi-threading, Memory Level Parallelism, Irregular applications, Manycore architecture

*Work done when the author was at University of Chicago

ICS '26, Belfast, United Kingdom

ACM ISBN 979-8-4007-2522-7/26/07
https://doi.org/10.1145/3797905.3800534

ACM Reference Format:
Andronicus Rajasukumar, Ruiqi Xu, Tianchi Zhang, Yuqing (Ivy) Wang, Tianshuo Su, Marziyeh Nourian, Jianru Ding, Jiya Su, Rajat Khandelwal, Alexander Fell, David F. Gleich, Yanjing Li, Henry Hoffmann, and Andrew A. Chien. 2026. UpDown: Efficient Manycore based on Many Threading and

Scalable Memory Parallelism. In *2026 International Conference on Supercomputing (ICS '26), July 06–09, 2026, Belfast, United Kingdom.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3797905.3800534

1 Introduction

Irregular applications like graph processing and sparse computations are pervasive in domains like social networks [44], bioinformatics [7, 56], graph databases [68] and so on. These applications have extremely large datasets represented with pointer-based data structures (trees, graphs, hash-maps, sparse matrices, etc.). Computations in these applications are therefore characterized by unpredictable data-driven control and memory access patterns with low data reuse [85].

Traditional cores achieve efficient execution by running large grain sizes (long-running threads) that allow the machine to learn behavior (branch prediction) and aggregate state (registers, caches), making computation efficient. They expend transistors to support multiple issues, out-of-order and speculative execution that enable high performance on a single thread. They use complex and large multi-level memory hierarchies to create the illusion of large, low-latency memory. Today's multicore chips are aggregations of sophisticated large cores [3, 9, 10, 36]. However, using these large cores to scale further for high performance is difficult due to area and power limits.

To find another path for single-chip performance increase, researchers have proposed manycore, which includes many thousands of simple in-order cores [21, 24, 25, 42, 81]. These simple cores have potential, in aggregate, to achieve high performance because of their high performance / mm^2. Further, they can maintain software compatibility. However, these benefits come at reduced thread performance, increased thread parallelism requirement, and increased cache hierarchy complexity. While they have demonstrated good performance for regular applications, manycore chips face challenges on irregular applications. Because of their unpredictable compute structure and data use, such applications incur low datapath utilization, poor cache effectiveness and ineffective use of memory bandwidth [42, 52, 103]. The problems are exhibited in multicore, manycore, and GPU chips.

We adopt a different approach for irregular applications – filling a datapath with instructions from many threads, each running dozens of instructions (~10-100), then switching to another thread. We call this approach "many thread". It differs from traditional approaches based on avoidance (prediction, speculation, caching) [3, 55], and is conceptually a variant of multithreading, but at coarse-grained interleaving [48, 74].

We propose a novel "many threaded" core, called UpDown (UD) core, that forms the basis for an efficient manycore architecture. UpDown employs a novel set of three architectural mechanisms (event-driven scheduling, software-controlled lightweight threading, and split-transaction memory access with software synchronization) to enable efficient short thread executions and interleaving. Together, this produces high datapath utilization. Coarse-grained multithreading in UpDown simplifies both hardware and software design.

We describe the design of a single-chip UpDown manycore comprised of 2,048 cores, first comparing it to a 20-core CPU, and then to an 8,192-manycore using simple, in-order cores. The evaluation workload includes scientific and graph applications running on sparse, skewed, real-world matrices and graphs. We study the performance sensitivity of the UpDown manycore to memory bandwidth and NoC latency. Finally, we assess the cost of the architectural mechanisms (area and power). The full UpDown system [16] is a 16,384-node graph supercomputer designed under the IARPA AGILE program[1]. Specific contributions of the paper include:

- Design of a manycore building block (UpDown) incorporating Event-Driven Scheduling (EDS), Software-controlled Lightweight Threading (SLT), and Split-transaction Memory Access with software synchronization (SMA).
- Evaluation on graph kernels that shows UpDown-based manycore can significantly outperform a 20-core Out-of-Order multi-core CPU (geomean speedup 47x).
- Study shows UpDown's architectural mechanisms enable 2.4-5.9x performance improvement over in-order RISC building blocks, specifically 1.9x (EDS), 1.4x (SLT), and 1.4x (SMA).
- Scaling to a full chip, these architecture features overcome their cost, enabling a 2048-core UpDown to outperform an 8192-core in-order chip by 3.1x overall.
- UpDown mechanisms enable it to effectively exploit the growing bandwidth available from HBM and tolerate high NoC latencies, supporting future scaling.
- Compared to an H100 GPU, UpDown achieves 20x area normalized geomean speedup.

Section 2 covers background topics of multi-threading and memory level parallelism. Section 3 frames the problem and describes key strategies in UpDown approach. In Section 4 and 5, we present the UpDown architecture and highlight its programming. In Section 6, we evaluate the architecture on challenging graph applications. Related work is discussed in Section 7. Finally, we summarize our results in Section 8.

2 Background

2.1 Multicores, Multithreading, and Caches

Traditional multicore architectures federate independent cores with a shared memory, cache-coherent address space [4, 39]. Coordination amongst threads is achieved with *atomic* synchronization primitives such as test-and-set and compare-and-swap. All of these are supported in the multi-level cache hierarchy which exploits data reuse to avoid memory latency [66]. It is common for the complex out-of-order cores to also exploit multithreading (Simultaneous multithreading or Hyperthreading [27] or barrel threading [8, 74]). Recent manycore research seeks to scale to 1,000's of cores [21, 42].

2.2 Memory Bandwidth and Parallelism

The memory bandwidth (HBM) revolution [41] (10TB/s in one package) makes compact architectures with high-memory bandwidth possible [5, 60]. Systems can achieve 10TB/s bandwidth; literally 100x that of a high-end server from just a decade ago. DRAM architecture in HBM requires many concurrent request to extract this bandwidth (i.e., memory-level parallelism or MLP).

CPUs and their caches use out-of-order execution, multithreading, and cache prefetching [14, 31, 35, 48] to create MLP. However, CPU request MLP is limited by physical registers (reorder buffers) [67] that define out-of-order window size, and load-store queues

[75] that maintain the memory consistency. DRAM MLP is limited by miss status handling registers (MSHRs) in caches that store information to handle memory access responses [67]. In any actual design, these features limit outstanding memory requests, causing CPUs to stall upon depletion, thereby limiting the achievable MLP.

GPUs generate MLP from thread groups (warps) [51], which require register names and MSHRs proportional to MLP. In GPUs, SIMT execution exacerbates these costs, preventing register reuse until all threads advance (e.g., on divergence).

3 Problem and Approach

3.1 Problem

Traditional cores achieve high performance when they have a steady stream of useful instructions, as well as low memory latency and high data reuse. However, irregular applications break these assumptions, causing low datapath utilization and associated poor performance.

3.1.1 Poor Datapath Utilization. Datapaths in modern Out-of-Order(OoO) cores are designed for high single-thread performance, incorporating deep pipelining, branch prediction, speculative execution, etc. All these mechanisms seek to create a steady stream of instructions from a thread's instruction window to fill the datapath.

Irregular applications have many branches and unpredictable data-driven control flows, as well as complex pointer/indirection structures, and random data accesses. These attributes produce ineffective branch prediction, speculation, and cache performance. The result is poor datapath utilization as shown in Figure 1.

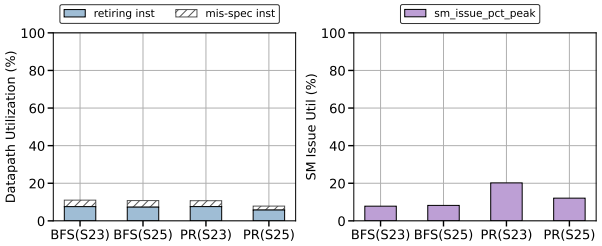

Figure 1: a) Datapath Utilization for PageRank and Breadth-first Search on real-world graphs. a) Core utilizations (blue) on a 20-core Skylake CPU [12, 37] are < 10%, b) SM Issue slot utilization (purple) on a NVIDIA-A40 GPU are < 20% [61, 90].

For example, two graph applications (BFS and PR) [12, 90] on real-world graphs have poor performance, both on CPUs (<10% datapath utilization, Figure 1a) and GPUs (<20% SM utilization, Figure 1b). The causes include high branch misprediction, thread divergence, and poor cache/memory performance due to irregular memory access patterns.

3.1.2 Low Cache and Prefetch Performance. Low irregular application data reuse causes high cache miss rates (>85%) that produce pipeline bubbles and stalls. Prefetchers – ineffective on the unpredictable access – create cache pollution and waste memory bandwidth. The need for per-miss MSHRs limits exploitation of HBM-scale memory bandwidth. The overall result is poor performance on irregular applications [52].

3.2 Approach

We pursue two ideas to increase performance on irregular applications. First, UpDown uses many parallel threads to fill the datapath. Having "many" is helpful because the threads are driven by unpredictable data dependencies (memory latencies) and unpredictable control dependencies (instruction sequence disruption). Second, novel UpDown mechanisms enable high memory parallelism, supporting more threads and computation and thus making more work available to fill the datapath.

3.2.1 Exploiting Many Threads. The UpDown core utilizes short sequences of instructions (10-100 typically) from one thread at a time. Efficient switching under explicit software control and having many threads available is key. In contrast, commercial server CPU cores run perhaps two threads concurrently to increase core utilization as illustrated in Figure 2.

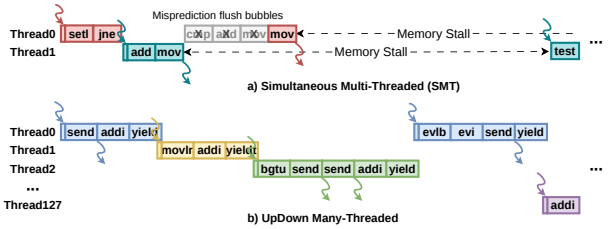

Figure 2: a) A traditional SMT core can suffer memory stalls and flushes that reduce datapath utilization; b) UpDown many-threaded core executes short sequences from many threads (different colors), using efficient switching and event-driven scheduling to fill the pipeline. Arrows represent messages from/to other core/memory.

In UpDown, explicit program thread management combined with event-driven scheduling converts unpredictable branches into many-thread scheduling. Fast hardware support for switching reduces cost to a single cycle, making many-thread execution efficient. UpDown has 128 hardware threads available to fill the datapath. This approach enables UpDown software to fill datapaths robustly, exploiting many threads.

3.2.2 Software Binding to Exploit Growing Hardware Memory Parallelism. Drawing on insights and memory access mechanisms we proposed in our earlier study[67], UpDown provides instructions for non-binding, split-transaction DRAM access. Each DRAM access specifies the memory address and data size to be read. When the response returns, it is handled in software, allowing the program to compute on the data directly or bind it to state resources (e.g., store it in a register). The program determines desired synchronization semantics. Critically, UpDown's novel DRAM operations do not bind outstanding DRAM access to any state element (e.g., register or MSHR), effectively creating unlimited memory parallelism.

In Figure 3, we contrast the conventional approach (left) with UpDown's non-binding approach (right). Traditionally, each memory request (① - ③) binds register names and MSHRs for each cache miss. Request ④ from core 1 is stalled because no MSHR is available in the last level cache. Request ⑤ from core 0 is stalled due to the MSHR shortage in private cache. Limited load/store stations and MSHR resources constrain memory parallelism. UpDown, on the other hand, does not bind DRAM requests to resources, so

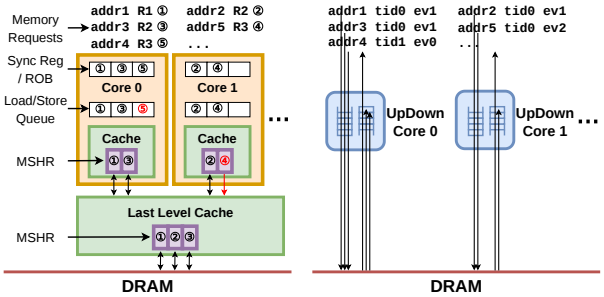

Figure 3: a) Each DRAM request (e.g., ③) is bound to an ROB, LSQ, and MSHR hardware resource. This limits memory parallelism. b) UpDown uses software names (address, thread ID, event label) to identify each DRAM request, and responses flow directly to software for handling. No hardware resources are bound, allowing unlimited outstanding requests.

unlimited memory requests can be in flight. UpDown uses software names (address, thread ID, event label) to trigger appropriate computation and synchronization for returning data. For example, to count responses, accumulate values, or specify the offset where the data should be stored. Software names enable an unlimited number of outstanding requests, enabling exploitation of growing hardware memory parallelism.

4 UpDown Architecture

UpDown is an event-driven, manycore parallel architecture on a single chip. It employs a large number of small, event-driven, MIMD cores. By comparison, each UpDown core is 200x smaller than a modern OoO core [96]. A group of 64 UpDown cores is called an accelerator, 8 accelerators form a cluster, and 4 clusters (2048 cores) with 4 HBM stacks [41] comprise an UpDown system, as in Figure 4. A 2-level interconnect architecture is used for communication between the components on the UpDown chip. The UpDown cores at the accelerator level are connected by a concentrated 2D-Mesh. The accelerators within a cluster communicate through a cluster switch and the cluster switches are connected by a chip-level switch. The HBM controllers are accessed through the cluster switches. The chip works alongside a Host CPU, accelerating workloads with high parallelism and low data reuse.

UpDown achieves high core utilization from many threads and exploits growing hardware memory parallelism with three key architectural mechanisms.

(1) ***Event-Driven Scheduling (EDS)***: UpDown threads are scheduled as events (short invocations), enabling fine-grained parallelism with hardware for scheduling and queue management.
(2) ***Software-controlled Lightweight Threading (SLT)***: UpDown threads have small state, and 128 hardware contexts. The threads are efficiently created, suspended, and terminated under software control.
(3) ***Split-transaction Memory Access with software synchronization (SMA)***[67]: Memory accesses are decoupled from thread execution (no tracking/bookkeeping) and synchronized using software names, enabling high and scalable memory level parallelism.

These mechanisms build on our prior research, including the Unified Automata Processor (UAP) [28], the Unstructured Data Processor (UDP) [29]. UpDown extends the fast symbol processing and multi-way dispatch capabilities in UAP and UDP to generic event-driven execution (EDS) and combines them with mechanisms for Scalable Memory Parallelism (SMA) proposed in [67] and Software-controlled Lightweight Threading (SLT). Each core in UpDown implements the UpDown-ISA instruction set. We describe the UpDown-ISA design (Section 4.1) and its innovative hardware implementation (Section 4.2).

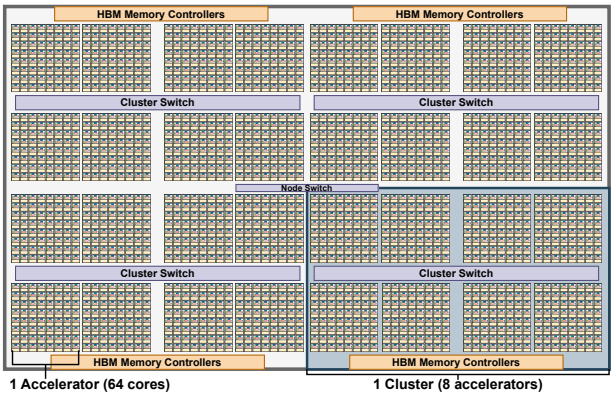

Figure 4: An UpDown Chip has 4 HBM stacks and 4 clusters, 8 accelerators per cluster, and 64 UpDown cores per accelerator for a total of 2,048 UpDown cores.

4.1 UpDown Instruction Set

We describe the key elements of UpDown's ISA, highlighting novel instructions. The full description is available [15].

4.1.1 UpDown Events. **Events** are first-class primitives and provide a unified interface to hardware events (e.g., DRAM responses) and software events (e.g., addValue, vertexUpdate, etc.) Each event triggers a short thread invocation and is comprised of an *event_word* (Figure 5) and a set of operands. The *event_word* has 4 fields: 1. *networkID* - location (core) the event will run; 2. *threadID* - thread to be executed; 3. *numOps* number of operands; and 4. *eventLabel* - offset for program. Events are synonymous with **Messages**.

Examples of how *event_words* provide unified handling of memory responses, application events, continuations, intercore coordination, and synchronization appear in Section 5.

4.1.2 UpDown State. **Operands** are the data payload in a message. They are directly integrated into thread's register namespace, eliminating the need for data movement instructions before use. Each core has 128 hardware thread contexts; 16 registers (**GPRs**) each. The thread register namespace (5 bits, 32 names) includes 16 GPRs, event operands, and several control registers as in Figure 6. Each core has 32KB of **Scratchpad Memory**. DRAM is 4 HBM stacks.

4.1.3 UpDown-ISA Highlights. The UpDown Instruction Set Architecture incorporates novel instructions (thread control, messaging, decoupled memory access, scratchpad synchronization, block move, etc.) that provide software access to special capabilities. Several of

the mechanisms, labeled *hw-opn*, are accessed implicitly. The novel UpDown instructions are summarized in Table 1.

The **thread control** instructions (`yield`, `yieldt`) provide zero-cycle thread creation and scheduling and single-cycle thread suspension and termination. Together, they enable efficient short thread execution under direct software control.

Messages can be created flexibly from many sources (blocks of scratchpad, registers, and sets of operands – the incoming message). This allows programs to transform, strip, and reformat messages remarkably efficiently. **Event words** are a critical machine abstraction, used to name both program entry points and threads[1]. The current event word is accessible in a special register, and customized instructions (`ev`, `evi`, `evlb`) provide convenient masked field updates, transforming it to trigger needed thread creation or activation. **Memory accesses to DRAM** expose split transactions to software – a message is sent to the memory controller, and the response (loaded values or write ack) comes in a return message (also see *hw-opn*). As with message sending, useful data sources for stores include scratchpad, registers, or incoming message.

Finally, there are traditional compute instructions (set of integer arithmetic, floating point operations, bitwise operations), control flow instructions and load/store instructions to the core scratchpad, which is only 32KB. So fast access is ensured. Since each core can access the scratchpad of other cores in its cluster (64 cores), `cswp` provides basic **synchronization**.

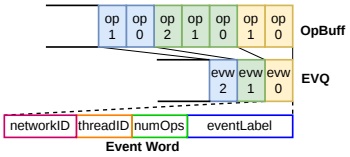

Figure 5: Events (<event_word,operands>) arrive at Event Queue (EVQ) and Operand Buffer (OpBuff) for hardware thread management. Any operands appear in the register namespace (see Figure 6).

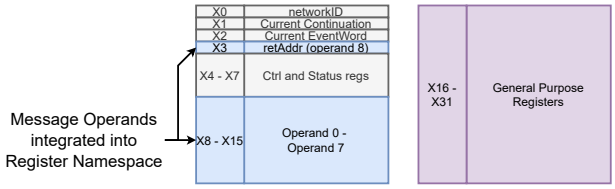

Figure 6: UpDown uses a portion of register namespace for message operands, supporting efficient short invocations. GPRs and a few special registers are also directly accessible.

4.2 Implementing UpDown Mechanisms

Each core executes a sequence of events, taken from its local Event Queue (EVQ) in FIFO order. Each event triggers a thread (existing / new) that executes on a single-issue in-order pipeline. Hardware thread contexts provide single-cycle thread switching. Figure 7 shows the UpDown micro-architecture highlighting the novel components in yellow.

[1]In the UpDown system, messages can be sent not only to a core or a memory, but also to a hardware thread!

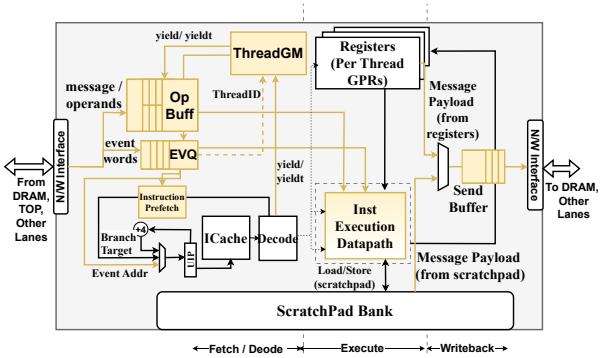

Figure 7: UpDown core has 128 thread contexts activated in single-cycle by events from the EVQ and OpBuff. ThreadGM activates and manages threads. Peeking into EVQ[head-1] enables prefetch for the ICache. Instructions are implemented in Inst Execution Datapath.

4.2.1 Event Driven Scheduling (EDS). Supported by fast switching enabled by hardware multithreading, EDS efficiently fills the datapath using many threads with irregular execution lengths and memory latencies. Event-driven scheduling is done by hardware when the current thread executes `yield`, freeing the core for the next event (see Figure 1). Efficient EDS is achieved with 1) FIFO event scheduling and 2) Single-cycle thread control.

a. FIFO scheduling: Implemented by the *Event Queue* (EVQ) and an *Operand Buffer* (OpBuff), see Figure 7. 64-bit Event words from incoming messages are pushed into the EVQ arrival order (2KB SRAM, up to 256 event_words). Message data payloads from incoming messages arrive in the OpBuff (18KB SRAM for 256 messages) (see Figure 5). The event-driven scheduling is simple, energy-efficient, enabling scaling to large queues and many cores. This is in contrast to complex hardware priority queues in [18, 65]. The OpBuff presents message operands to threads in their register namespace as in Figure 6. This eliminates copying message data into registers, saving instructions in UpDown's very short thread invocations.

b. Single-Cycle Thread activation: FIFO scheduling and the decoupled EVQ and OpBuff enable single-cycle thread activation. The event_word at the *head* of the EVQ contains the *threadID* used by the *ThreadGM* to either activate an existing thread context or allocate a new one if it is `0xff`. FIFO scheduling enables easy identification of the next thread (existing or current) as well as its instructions. *Instruction Prefetch* peeks into `EVQ[head-1]` to identify instructions for the next invocation to feed the decoder and *ICache*.

UpDown uses the multi-way dispatch mechanisms proposed in [28, 29] for efficient code layout and address computation (`base + event_label`). UpDown thread invocations are short, ~10-100 instructions, and control flow instructions have a short range, further reducing ICache misses.

4.2.2 Software-controlled Lightweight Threading (SLT). SLT allows UpDown cores to launch threads using `send`, `sendr`, `sendops` instructions. These instructions, which draw inspiration from [19, 59], format a message using data from the scratchpad, registers, or OpBuff, respectively, coupled with an event_word and a continuation_word (that are created using `ev`, `evi`, `evlb` instructions) and push them through the *SendBuffer* into the NoC. The event_word

Table 1: UpDown-ISA Instruction Classes

Instruction Class	Instruction	Function

Thread Control	*hw-opn*
	`yield`
	`yieldt`
		Thread schedule, Thread creation as specified by incoming message
		Suspend current Thread (self) and trigger hardware schedule
		Terminate current Thread (self) and trigger hardware schedule

Messaging	`send`
	`sendr`
	`sendops`
		send 1-8 message words, sourced from scratchpad memory
		send 1-4 message words, sourced from registers
		send 1-8 words, sourced from incoming message (operands)

DRAM Access (split-transaction)	`sendm`
	`sendmr`
	`sendmops`
	hw-opn
		load/store of 1-8 words DRAM: store data from scratchpad
		load/store 1-8 words: store data from registers
		load/store of 1-8 words; store data from incoming message (operands)
		load message return data or write ack - triggers user-customizable handler

Event-word update	`ev`
	`evi`
	`evlb`
		create/update fields in event/continuation_words from registers
		create/update fields in event/continuation_words from immediate values
		create/update event_label in event/continuation_words from immediate values

Synchronization	`cswp`	compare-and-swap across a set of 64 cores scratchpads
Traditional Instructions	(`add, sub, mul, and, or, cgt, ceq, clt`) - Integer, (`sl, sr, slor, sror, sland, srand`) - Bit Manipulation, (`beq, bne, blt, bgt`) - Control Flow, (`fmadd, fadd, fmul`) - Floating Point, (`vmadd, vmul, vadd`) - Vector (sub-word), (`movlr, movrl`) - Load/Store scratchpad	

includes a networkID that is used for routing. These messages will, in turn, trigger a thread creation/activation at the destination.

SLT provides direct software control over thread suspension and termination. Programs can identify variable latency operations (DRAM access, remote invocation, synchronization) or unpredictable branches, and release the datapath. `yield`, `yieldt` instructions suspend, terminate the current thread, respectively. These operations update EVQ and OpBuff pointers to move to the next event. The `yieldt` instruction also deallocates the thread context and *threadID*. Both instructions are processed directly by the *ThreadGM*. The thread contexts are implemented using banked SRAMs (16 1KB banks). Each bank supports 128 64-bit registers with GPR names to select banks. ThreadIDs (0-127) are used to address the SRAMs, achieving single-cycle thread switching.

4.2.3 Split-transaction Memory Access with software synchronization (SMA). UpDown architecture incorporates split-transactions for DRAM accesses as described in [67], where the authors evaluated benefits on limited workloads. We describe SMA briefly and in UpDown we integrate it with EDS, allowing software-defined and hardware events to be treated uniformly. We evaluate SMA on a broad set of workloads here. Combining SLT with SMA enables even higher memory parallelism.

In UpDown, SMA treats DRAM accesses as messages. The `sendm`, `sendmr`, and `sendmops` instructions have an analogous structure to the messaging operations (see Table 1). These DRAM access instructions are not only non-blocking, they allow control over the size of access (1-8 words). `sendm` instructions initiate loads or stores. For stores, data can be sourced from scratchpad (`sendm`), registers (`sendmr`), or operands (`sendmops`). The DRAM load/store messages are put into the *SendBuffer*, similar to other messages. The key is that the DRAM access messages are non-binding – they do not associate the returning load data with names (e.g., register, cache block) or physical resources. This enables effectively unlimited memory parallelism and flexible custom synchronization as outlined below.

a. Split-transaction DRAM access (non-binding): By treating DRAM accesses as messages, UpDown splits the memory request from its response. Because the request is not bound to any resources (e.g., load/store station, cache miss handling register - MSHR, etc.), memory parallelism can be scaled without hardware resource limitations [67]. Of course, the request must be identified - first it comes back to the requesting thread, and the invoked program event uses the memory address. We will see the new capabilities this creates, particularly for irregular applications, in Section 5.

Thus, a single UpDown thread can generate 100s of outstanding memory requests. Parallelism is further scaled up with multiple cores and accelerators. This allows UpDown's SMA to generate enough parallelism for even the most advanced HBMs.

b. Software-name based synchronization: Unlike conventional architectures that synchronize a load to its register target, in UpDown, DRAM responses trigger a user-defined event on a thread specified by the continuation_word ((*networkID, threadID, eventLabel*)). The DRAM return data appears in the OpBuff and is directly accessible using register names (X8-X15), or specified collectively and moved with special variants of `bcopyops`, `sendops`. Thus, the integration of the DRAM data into the computation is determined by the triggered program event. Such DRAM response *synchronization* is based on program-defined computation.

To illustrate, consider dot product in Figure 8. One vector is streamed from DRAM, the other is in the scratchpad. `stream_rows` issues concurrent reads of vector in 8-word chunks, specifying `dot_p` event for the responses. `sendm`'s non-blocking property allows the `stream_rows` loops to produce many concurrent accesses.

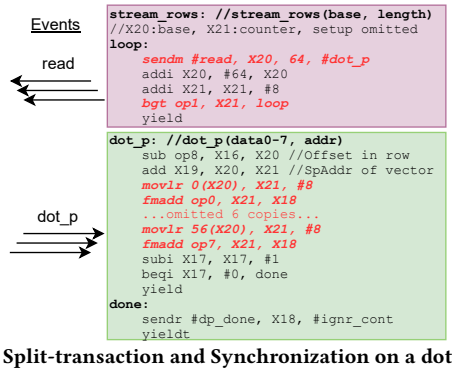

Figure 8: Split-transaction and Synchronization on a dot product.

Each DRAM response triggers a `dot_p` event execution that computes 8 elements of the dot product and accumulates in X18 (a commutative operation). The red `movlr` and `fmadd` instructions compute one element of the dot product chunk. A full dot product is multiple `dot_p` events, each synchronized by its memory response and uniquely identified by its memory address. Further, exclusive access to X18 is ensured by atomic thread execution. To

synchronize completion of the entire dot product, the counter (X17) tracks the number of chunks completed, sends a dp_done event with X18's value when all chunks are done, signaling the dot product is complete, and terminates the thread.

Generally, the program triggered by the DRAM access response is an arbitrary computation that can implement any complex and application-specific synchronization.

4.3 Event Driven Execution in a UpDown core

Putting it all together, Figure 9 depicts event-driven execution on an UpDown core. A 2-event program is shown loaded in the core's ICache. ① shows the structures that use the various fields of an event word. ② Incoming events (memory responses or software messages) are decoupled, with event_word enqueued in the EVQ and data payload in the Operand Buffer. ③ When an event_word is at the head of the EVQ, its event_label dispatches execution to the right instruction address (EDS). ④ The event_word and its operands (num_operands specified in event_word) are mapped into thread namespace, enabling them to be named directly in instructions (X1, X2, X7-X15). ⑤ The *threadID* is used to activate the corresponding register context (SLT). ⑥ Thread execution can further launch new events (send) or access DRAM (sendm) respectively (SMA). These are non-blocking (decoupled by the sendBuffer) and asynchronous memory references that do not stall the pipeline.

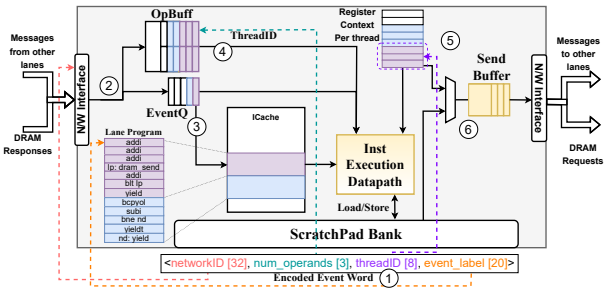

Figure 9: Logical Event-Driven Execution on UpDown cores.

5 UpDown Programming

A plethora of sparse and graph algorithms, including TC [78], BFS, PR [89, 95], and SpMM [97], have demonstrated outstanding performance on the UpDown system. These applications use a map-reduce framework, KVMSR [92] and a C-like system programming language - UDWeave [30]. UDWeave exposes the key event, messaging, and thread management mechanisms to user programs. Like C, it provides basic types and data structures and uses intrinsics to expose mechanisms. Each computation is a set of threads; each thread is a set of short, comprised of events that share the thread state. System-level libraries for flexible thread scheduling [92] and application-controlled data placement[91] enable easy program development. In this section, we illustrate basic UpDown programming via UDWeave code snippets from Sparse Matrix-Matrix Multiplication (SpMM) using Gustavson's algorithm [32].

5.1 Creating and Controlling Parallelism

The *UDWeave* code in Listing 1 declares an *spmm* thread type with two events. The first event (`receive_matrix()`) plays the role of spawning workers. In the for-loop, the `evw_new` instruction updates the event word `evw` to specify launching a new thread executing event `row_worker::init` on core `nwid`, and the `send_event` intrinsic is called to send the message to the target (maps to `send` and `ev`, `evlb` instructions in Table 1), which immediately schedules the event once it reaches the top of event queue.

Listing 1: Creating and Controlling Parallel Threads

```
thread spmm { // Thread variable declarations
  long row_id, counter, total_row;
  event receive_matrix() {
    for (counter = 0; counter < NUM_CORES; counter++) {
      long evw = evw_new(counter, row_worker::init);
      send_event(evw, row_id, worker_ret);
      row_id = row_id + 1;
  }}
  event worker_ret() {
    if (row_id < total_row) {
      send_event(row_worker::init, row_id, worker_ret);
      row_id = row_id + 1;
    } else {
      counter = counter - 1;
      if (counter == 0) { // Check for termination
        send_event(compuation_complete);
        yield_terminate;
}}}
```

After launching the child tasks, `spmm` thread yields, allowing the core to be used by other computations (`yield` at the end of the events are omitted for space). As specified in the send instruction, worker threads invoke `spmm::worker_ret` event on the `spmm` thread upon completion, which spawns new tasks if more rows are remaining or decrements `counter` and waits for all workers to return. Note that the work claiming scheme alleviates load imbalance in irregular applications. Once all workers have returned, the `spmm` thread replies to its caller and deallocates itself using `yield_terminate` (`yieldt` instruction).

5.2 Multi-level Memory Parallelism

Listing 2: Fine-Grained Computations and Multi-level Memory Parallelism in SpMM

```
thread row_worker {
  long counter;
  event init(long row_id) {
    // Allocate scratchpad for results omitted
    long* idx_addr = idx_base + row_id * sizeof(long);
    send_dram_read(idx_addr, 2, launch_nonzero_worker);
  }
  event launch_nonzero_worker(long start, long end) {
    counter = min(end - start, NONZERO_THREADS);
    for (long i = 0; i < counter; i = i + 1) {
      send_event(nonzero_worker::init, val_addr, col_addr, accum
          , worker_return);
      val_addr += sizeof(double);
      col_addr += sizeof(long);
  }}
  event worker_return() {// Check for termination
}}
thread nonzero_worker {
  double scale;
  double* base_addr;
  double* local acc_addr;
  int counter, num_completed;
  event process_nonzero(double* val_addr, long* col_addr, double
       * local acc_addr_) {
    // Fetch col, val addresses into thread variables
  }
  event stream_row() {
    float* addr = mat1_addr + mat1_width * col_idx;
    for (int i = 0; i < matrix_width; i = i + 8) {
      send_dram_read(addr, 8, update_accum);
  }}
  event update_accum(double v0 ... v7, double* addr) {
    double* local update_ptr = acc_addr + (addr - base_addr);
    update_ptr[0] += v0 * scale;
    ...
    update_ptr[7] += v7 * scale;
    // Termination check omitted
}}
```

In Listing 2, each `row_worker` thread computes a result row, scaling rows from the dense matrix based on nonzeros and adding up the results. To initialize, it computes its nonzeros' value and column index addresses. The `row_worker::launch_nonzero_worker` event creates the first level of parallelism across the nonzeros in a row by launching multiple new `nonzero_worker` threads running on the same core. Within each of these, the `nonzero_worker::stream_row` event generates the next level of parallelism through concurrent reads to the entire row enabled by split-transaction memory access, similar to dot product in Figure 8. These two levels of parallelism illustrate how programs can use multiple threads and events to increase program MLP.

5.3 Fine-Grained Computations

UpDown efficiently executes irregular applications with lightweight threads, scheduling short events. In the event shown in Listing 2, `nonzero_worker::update_accum`, the core receives 8 elements from a row of the dense matrix, scales them, and accumulates the results into values in the scratchpad. Processing 8 elements, the `update_accum` event executes only 29 instructions (with counter update, termination check, and yield), far fewer than could be efficiently scheduled on a conventional core. Moreover, EDS relaxes the ordering requirements, allowing any available data to be immediately used, maximizing UpDown's ability to hide latencies.

6 UpDown Evaluation

6.1 Simulation Models

We model three systems, an x86 system (BL), an in-order RISC manycore (IR), and UpDown. Each is configured as in Table 2. The IR model is based on Hammerblade [42] and performance validated on GEMM and 2DFFT, matching instruction counts, core utilization, and memory activity [2] from [42]. The UpDown model is a cycle-accurate instruction-level simulator for the UpDown accelerator (cores and scratchpad), integrating Ramulator [53] for HBM2 modeling. We use a channel-level latency and bandwidth model for HBM3e, since Ramulator does not support it. The model was calibrated with Ramulator/HBM2 to within 5% on the seven applications UpDown uses a 3-level NoC with a 2D mesh within accelerators, a switch connecting 8 accelerators in each cluster. The four switches are connected at the chip level. We integrate Booksim2[40] to model the accelerator-level NoC and use a latency-bandwidth model for the cluster-level and chip-level switches.

6.2 Methodology

We use a variety of workloads: 3 graph workloads - Breadth-First-Search(BFS), PageRank(PR), Triangle Counting(TC), 2 sparse computations - Sparse-Matrix-Vector product(SpMV) and Sparse-Matrix-Dense-Matrix product(SpMM), described in Section 5, and 2 dense computations - Dense-Matrix-Matrix product(GEMM) and 2D Fast-Fourier-Transform(2DFFT). The datasets are listed in Tables 4 and 5. For GEMM, we use 512×512×512 with a batch size 64 (M512) and for 2DFFT, an 8k×8k (M8K) and 16k×16k (M16K) input. BL performance for BFS, PR and TC is from [12] and for SpMV, SpMM[3], and GEMM from [38], and 2DFFT from mkl_fft [62]. The GPU graph applications' performance is cuGraph[17] from the RAPIDS ecosystem[82]. We use the PyTorch[11] for SpMV and SpMM. We define the metrics used for evaluation in Table 3.

[2]From the Hammerblade authors, we learned that HB performance was extrapolated from 128 PE simulations – 1/64th of a full chip design. Our model enables modeling the full Hammerblade chip (8192 cores + HBM). Note that in the HB paper, what is reported as memory utilization is actually memory activity.

[3]Similar to GNN aggregate kernels in DGL[88] with matrix width of 256

Table 2: System Configurations

System	Configuration
Baseline (BL)	20 x86 Intel (Skylake), OoO cores, @2GHz; Intel Xeon Gold 6138 L1d: 32KB, L1i: 32KB, L2: 1MB per core; L3: 27.5MB shared DDR4 with 128GB/s
In-order RISC (IR)	8192 cores (@1.35GHz + 4KB scratchpad) + 2048 cache tiles (32KB per tile), HBM2 x4 stacks @256GB/s each (16x channels / stack) NoC latency (cycles): Intra Accelerator - 8, Intra Cluster - 32, Chip - 64
UpDown (UD)	2048 cores (@2GHz + 32KB scratchpad) HBM3e x4 stacks @1200GB/s each (16x channels / stack) Accelerator NoC: 4x4 CMesh (Flit-width:256b, #VCs:4 #Buffs:4) Cluster Switch: 14x14x8 clos switch @3GHz (Flit-width:256b, #VCs:4 #Buffs:4) Chip Switch: 4x4x48 clos switch @3GHz (Flit-width:256b, #VCs:4 #Buffs:4) Modeled Switch Latency (cycles): Intra Cluster - 16, Chip - 32
H100 GPU	16,896 cores (@1.59GHz + 256KB (per SM) L1 + 50 MB L2 cache) HBM3 with 3.35TB/s

Table 3: Metrics

Metric	Definition
Runtime (Seconds)	Execution Time
Instruction Throughput	Total Instructions Executed / Runtime
Application Memory BW (GB/s)	DRAM Read+Write bytes / Runtime
Memory Utilization	Application BW / System BW
Memory Level Parallelism (DRAM requests)	Application Memory BW * Average Access Size / Memory Latency

Table 4: Real-World and Synthetic Graph Datasets

Dataset	#Vertices	#Edges	Max deg	Avg deg
hollywood (HW) [22]	1,139,905	112,751,422	11,469	100
com-Orkut(OR) [98]	3,072,441	117,185,082	33,313	76
Patents(PA) [47]	3,774,767	16,518,948	793	8
liveJournal (LJ) [22]	5,363,260	50,545,899	19,432	18
RMAT-s23 (S23) [83]	4,611,337	129,336,873	258,278	56
RMAT-s25 (S25) [83]	17,062,371	523,599,001	639,686	61

Table 5: Sparse Matrix Datasets

SpMV Dataset	#Rows	#Columns	#Nonzeros	#NNZ per Row
ML_Geer (ML) [22]	1,504,002	1,504,002	110,879,972	73.72
HV15R (HV) [22]	2,017,169	2,017,169	283,073,458	140.33
stokes (ST) [22]	11,449,533	11,449,533	349,321,980	30.51
uk-2005 (UK) [22]	39,459,925	39,459,925	936,364,282	23.73
SpMM Dataset	**#Rows**	**#Columns**	**#Nonzeros**	**#NNZ per Row**
amazon-2008 (AM) [22]	735,323	735,323	5,158,388	7.02
nv2 (NV) [22]	1,453,908	1,453,908	52,728,362	36.27
italy_osm (IT) [22]	6,686,493	6,686,493	14,027,956	2.10
delaunay_n23 (DE) [22]	8,388,608	8,388,608	50,331,568	6.00

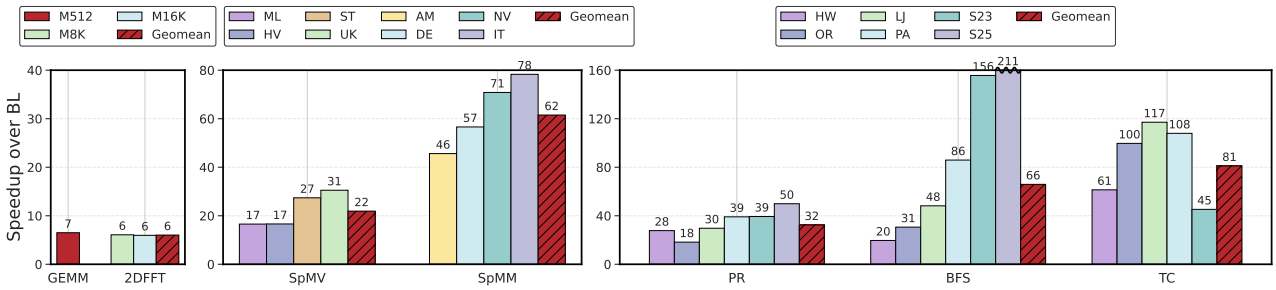

Figure 10: UpDown Speedup versus BL, varied workloads and datasets.

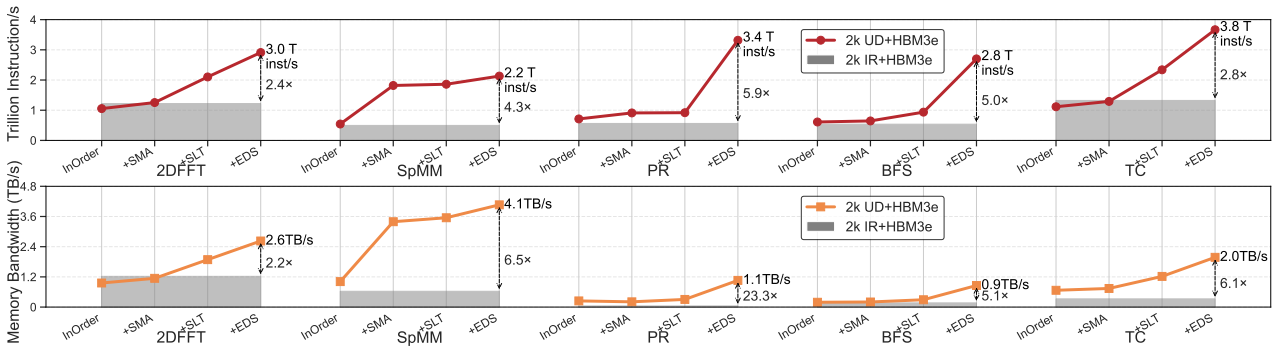

Figure 11: Understanding UpDown mechanism impact 2K core UD (red and orange) vs. IR (gray), achieved compute (instructions per second) and average memory bandwidth.

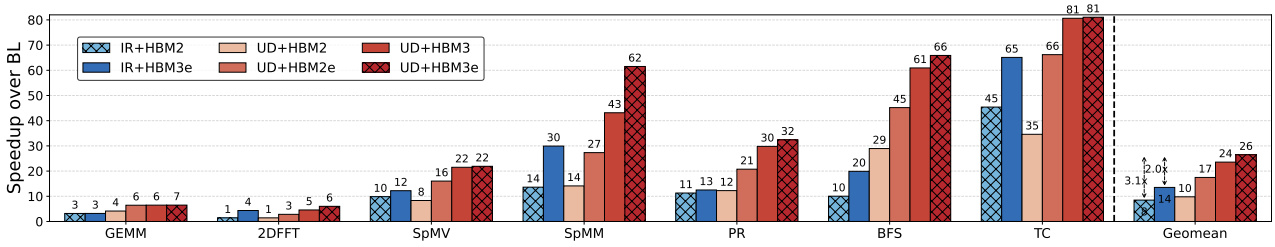

Figure 12: Comparing UpDown and IR Geomean of speedups versus BL, varied workloads and memory systems.

6.3 Performance Analysis

6.3.1 Comparison to Baseline (BL) multicore. Experiments show the benefits of manycore - UpDown achieves 7x, 6x, 22x, 62x, 32x, 66x, and 81x geomean speedup over the baseline multicore system on GEMM, 2DFFT, SpMV, SpMM, PR, BFS, and TC, respectively (see Figure 10). This outperformance is remarkable, considering that BL requires nearly twice the area of an UpDown system (see Section 6.6).

SpMV and SpMM perform irregular data-dependent memory accesses, which makes prefetching ineffective. UpDown can generate memory parallelism, achieving higher performance when data no longer fits into the cache. For GEMM and 2DFFT, despite BL's 8x compute advantage due to the AVX-512, UpDown's compute parallelism delivers 7x and 6x performance improvements.

The graph algorithms (PR, BFS, TC) all exploit UpDown's mechanisms to utilize fine-grained vertex and edge parallelism. This enables them to tolerate the extreme skew of these real-world graphs and effectively exploit their greater parallelism to achieve geomean speedups of 32x, 66x, and 81x.

6.3.2 Feature-wise Performance (normalized). To understand the source of the performance gains and the contribution from UpDown's mechanisms, we compare UpDown to a normalized IR system, matching the clock rate (2GHz), number of cores (2K), and memory system (HBM3e). We incrementally add SMA, SLT, and EDS mechanisms (see Figure 11), and report the instruction throughput for the 2K cores and application memory bandwidth on 4 memory stacks. Results are the geomean for each application over a set of graph inputs. The experiment results isolate the benefits of each mechanism, highlighting that each application needs different mechanisms. SpMM benefits most from SMA, increasing achieved memory bandwidth dramatically, and thereby achieving 4.3x performance advantage. PR and BFS benefit from EDS, achieving 5.9x and 5x better performance. TC and 2DFFT also benefit considerably from EDS, but also SLT, growing performance significantly. The in-order cores perform better on TC and 2DFFT, so the relative

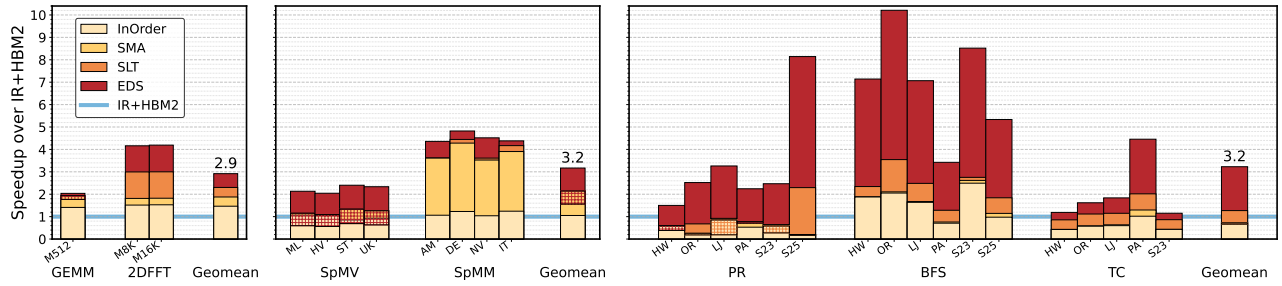

Figure 13: Comparing 2K core UpDown to full 8K core IR chip; contribution of UpDown mechanisms.

performance increases are only 2.8x and 2.4x. SpMV was omitted because it was difficult to create a fair comparison. Overall, the benefits range from 2.4x to 5.9x in performance, and 2.2x to 23.3x in memory utilization. These results show how UpDown mechanisms increase the productivity of the datapaths.

6.3.3 Overall Performance (Chip-level). Design of a manycore chip is a tradeoff between the complexity of the building blocks, and the number that can be included. As described in Section 6.6, each UpDown core is larger than the IR cores due to the hardware required for its novel mechanisms described in Section 4. So in Figure 12, we make an iso-area comparison 2K core UpDown and 8K core IR. The IR chip is designed for 2.7x higher instruction issue rate.

We compare the speedups of UpDown and IR with BL as the reference, and vary the memory systems (HBM2 and HBM3e). Reported performance is the geomean over varied input datasets. The experiment results show that IR matches or slightly outperforms UpDown with HBM2. However, with more memory bandwidth available, UpDown outperforms IR on all of the applications. This outperformance is significant, ranging from 1.2x to 3.3x. Geomean improvement is 2.0x relative to IR+HBM3e, and 3.1x relative to IR+HBM2. These results show that despite the greater compute potential of IR, its cores and caches cannot exploit increased memory bandwidth to scale up for irregular apps. In contrast, UpDown continues to increase performance as memory bandwidth grows.

Discussion on memory bandwidth: The in-order cores of IR are memory-bound on most irregular applications. Despite modification of the cores to support 63 non-blocking memory, performance does not scale beyond HBM2. In contrast, UpDown's cores use EDS, SLT and SMA to exploit increased memory bandwidth, generating many memory requests.

6.3.4 Feature-wise performance (Chip-level). We compare chip-level performance of UpDown and IR, iso-area, in Figure 13. This study drills down into how UpDown can outperform an equivalent-size IR chip, despite having many fewer cores. We add the novel UpDown mechanisms incrementally and show the performance uplift for each in a stacked bar chart.

Similar to Figure 11, the three mechanisms (SMA, SLT, and EDS) contribute different amounts for each application. Here, we see graphically how the benefits per-core, translate into speedups and outperformance of IR at the chip-level. The red bars show EDS is the biggest performance contributor, and improves all applications, but notably the graph computations and SpMV. The second is SMA, which significantly enhances SpMM performance. Finally SLT is a consistent contributor across most of the applications. For regular applications, geomean improvement is 2.9x. For the sparse matrix and graph applications, geomean improvement is 3.2x. Overall there is 1.9x (EDS), 1.4x (SLT), and 1.4x (SMA) performance advantage from the three mechanisms.

6.4 ManyCore Scaling: Compute and Memory Limits

An efficient manycore design should scale with increased compute resources (cores) and memory resources (bandwidth). In Figure 14, we explore UpDown scaling. The orange line shows scaling with infinite bandwidth and the red line shows the actual scaling for UpDown for various numbers of cores. Recall that UpDown delivered high performance for all of these benchmarks with 2K cores (Figure 10).

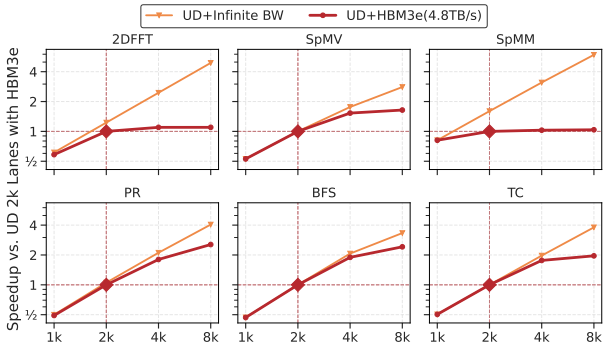

Figure 14: UpDown performance scaling shows that 2K-core is a good match for several applications, but 4K-core in a more advanced process might better exploit full HBM3e bandwidth.

Two matrix-based applications (SpMM and 2DFFT) see declining performance increases beyond 2K cores (note the log-log scale). For these applications, 2K UpDown cores is a good balance to the memory system. For SpMV, the performance benefit for scaling is better; similar to the graph applications (PR, BFS, and TC), performance increases nearly linearly to 4K cores and a tail-off beyond that. This suggests a UpDown with 4K cores (in denser Si process) might better exploit HBM3e memory bandwidth.

UpDown's Split-transaction Memory Access with software synchronization (SMA) mechanism, combined with Software-controlled Lightweight Threading (SLT), empowers applications to create high levels of memory level parallelism (MLP) critical to

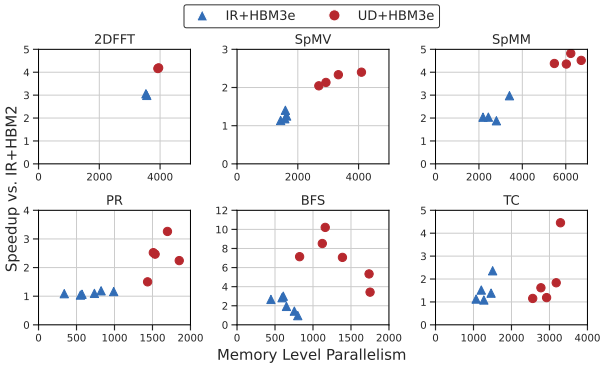

Figure 15: UpDown achieves higher MLP despite only 1/4 the cores, generally producing much higher performance.

effectively utilizing HBM. In Figure 15, we explore the relationship of MLP and performance. We ran each of our kernels on the full UpDown and IR systems, producing a performance and MLP scatter-plot. The results show that despite having many fewer cores, UpDown achieves much higher MLP than IR for all kernels – utilizing the HBM better. Further on UpDown, nearly all the kernels (2DFFT, SpMV, SpMM, PR, and BFS) show excellent performance scaling with large increases (3x) proportional to increased MLP (4.5x). Only TC, which becomes compute-bound for most graphs, fails to benefit as IR's caches maintain performance while reduce memory traffic.

6.5 Performance Impact of Network-on-Chip Latency

One key benefit of the UpDown approach is latency tolerance, that is crucial for scaling to high core counts. We vary NoC latencies (inter-accelerator and inter-cluster) in Figure 16, considering both lower and higher latencies. On the left, we show IR's performance with 0.5x to 4x our design latency. The IR cores are highly sensitive to NoC latency, with performance dropping as much as 60% (and a mean of 43% at 4x latency) compared to 0.5x. IR's single-threaded in-order cores cannot tolerate longer NoC latency to the cache, much less to DRAM.

In contrast, UpDown performance remains high even with 4x greater NoC latency. EDS and SMA enable memory parallelism and allow programs to loosen ordering requirements for processing memory responses. SLT can avoid stalling after issuing memory requests; another thread can be selected, providing additional instruction streams to fill the datapath. Because of the benefits of UpDown's three novel mechanisms in tolerating variable and high latencies, experiment results suggest that an UpDown with 4x NoC latency (e.g., 16x cores, additional memory bandwidth) could show excellent efficiency.

6.6 Area and Power Analysis

Finally, we evaluate area and power. UpDown core RTL has been synthesized and timing closed at 2GHz.The interconnect (NoC + switches), was generated using Constellation [105], and synthesized at 3GHz. Synopsys SAED 14nm [80] libraries were used for both.

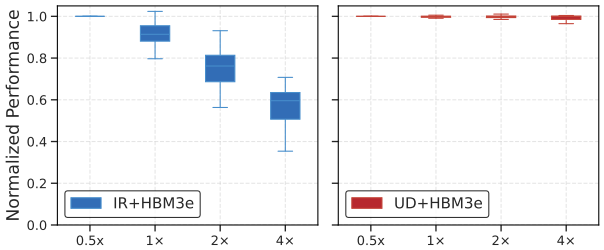

Figure 16: IR performance (left) declines with increased NoC latency due to limited latency tolerance. UpDown's performance (right) does not, suggesting better scalability to higher core counts.

SRAM area is estimated using CACTI 3D [49] in 28nm and scaled to 14nm using scaling laws in [77].

Table 6 shows the UpDown core area, including the NoC router and scratchpad (1 per core) area. The full UpDown chip (2048 cores) is 348 mm^2, which is similar in area to the 8192-core HammerBlade [42] and 2x smaller than the out-of-order multi-core baseline, which has an area of 694 mm^2 [96]. Compared to an in-order RISC, each UpDown core is roughly 4x larger, but achieves much higher performance. The larger area is a good investment as the UpDown chip is 3.1x more area efficient (perf/mm^2) than the IR chip. UpDown chip power is 59W based on the dynamic and leakage power reported during synthesis by Synopsys DC [80], below typical CPU package limits and the Baseline CPU's (Intel Xeon Gold 6138) 125W TDP[96].

Table 6: Chip Area Estimates [14nm]

Component	Area (mm^2)
1 UpDown Core	0.1041
1 UpDown Scratchpad (32KB)	0.0337
1 UpDown Accelerator (64x Cores + 64x Scratchpads + 2D-cmesh)	9.47
1 UpDown 14x14 Cluster Switch	10.71
1 UpDown Cluster (8x Accelerators + Cluster Switch)	86.47
1 4x4 Chip Switch	1.97
UpDown chip (4 Clusters + Chip Switch)	**347.85**
InOrder (IR) Chip (8k cores, NoC routers, local memory)	310.00
Baseline (BL) Chip (20 OoO cores, L1, L2, L3 caches) (Intel Xeon Gold 6138)	694.00

6.7 Comparing UpDown to GPUs

Next, we compare UpDown to an H100 GPU (TSMC 4N). We normalize area, scaling UpDown's area to 5nm makes it 103mm^2 (logic and SRAM scaling laws from [77] and [99]). UpDown's ISO-area performance is 20x greater, geomean (Figure 17). This reflects the effectiveness of manythreading (and MIMD) for datapath utilization on irregular applications.

7 Discussion and Related Work

The full UpDown system [16] is a 16,384 Updown node graph supercomputer designed under the IARPA AGILE program[1]. Nodes are connected by a high-bandwidth, low-diameter network [45], with latency of 0.5 microseconds and > 50 petabytes/ second bisection – 50x greater than today's largest supercomputers.

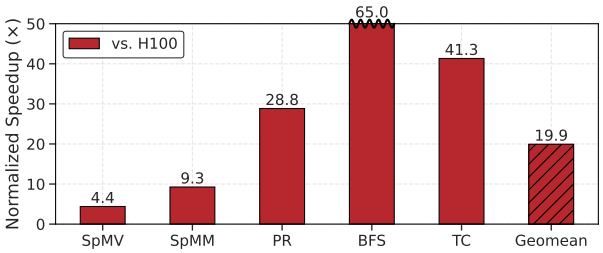

Figure 17: UpDown speedup over H100 (area normalized) on various irregular applications. The 20x improvement shows that UpDown utilizes its datapaths more effectively.

In this paper, we focus on the UpDown chip, which achieves high performance with 2,048 cores and high core utilization. The key is architectural mechanisms (event-driven scheduling, software-controlled lightweight threading, split-transaction memory access with software synchronization) that effectively exploit many unpredictable threads to fill datapaths. These approaches build on our prior research [28, 29, 67] and a variety of research in manycore, multithreading, and event-driven architectures.

7.1 Manycore Architectures

Manycore architectures refers to single-chip processors with 100's to 1000's of cores. Typical compute building blocks for manycores are in-order cores, chosen for their simplicity that produces small area and energy efficiency. To date, most manycore systems employ coherent data caches in a single address space [2, 21, 24, 25, 42, 63, 81], similar to commercial multicores [3, 55].

In-order cores have limited capability to tolerate cache misses or branch mispredictions, producing low datapath utilization on irregular applications. Research designs such as Intel's PIUMA add multithreading and special memory offload engines [2], and HammerBlade eschews multithreading but allows a single thread to have 63 outstanding memory requests [42]. In contrast, UpDown introduces SMA for unlimited outstanding memory accesses, and EDS and SLT for programmable composition to fill the datapath.

7.2 Multi-thread and Many-thread

UpDown's many thread model can be viewed as multithreading with coarse interleaving (10-100 instructions). This differs from most prior multithreaded processors that interleave individual instructions from multiple threads [3, 48, 55, 74, 84]. These designs have had challenges in scaling to large numbers of hardware threads with difficulty scaling clock rates and achieving high single-thread performance [74, 84]. An outlier, Cyclops64 [23] sought to use many threads, but had high context switch costs. UpDown's approach has several benefits: 1) Running one thread at a time simplifies the core, easing implementation (e.g., register file, execution unit scheduling, etc.). Simpler cores enable a large number of threads (128). 2) Running one thread at a time simplifies programming. Each thread has exclusive access to the scratchpad while running, easing coordination amongst a core's threads [33, 34]. 3) Explicit threading control by programs (e.g., SLT, EDS) manages long-latency operations at low switching cost. For example, a group of split-transaction memory references create MLP and overlap latency.

7.3 Event-driven Architectures

The value of event-driven execution for parallel computing is well-known [6, 26, 43, 46, 58, 59, 65, 69, 79, 93]. Early examples, J-machine and *T, both included hardware-supported message queueing and task scheduling followed by commercial systems with active message mechanisms [43, 86]. Cerebras schedules threads based on data arrival events, but relative to a programmed virtual topology, limiting it to systolic array-like programs [50]. The UpDown system blends EDS with mechanisms for short threads, program thread management, and lightweight messaging to achieve high datapath utilization. Together these enable UpDown to deliver high performance without data caches.

7.4 Memory Parallelism

Approaches for generating memory parallelism include streaming registers [54, 70, 71, 94, 101], speculative execution [57], decoupled access and execute [64, 76], slice cores [13] and multithreading [74, 84]). But in most systems, these will access a cache, so increasing memory parallelism depends on mechanisms that prefetch into cache or local memory with hardwired logic [87, 100] or programmable primitives [72, 73, 102, 104]. UpDown incorporates instructions for non-binding DRAM access (launching split-transaction), giving programs direct control of access parallelism and size. Programs can create the desired memory parallelism (multiple accesses) and structure (size). UpDown's single instruction message primitives build on the J-machine `send` [20]. But, using them for direct DRAM access is novel to UpDown.

8 Summary

We presented the architecture of an efficient manycore building block - UpDown. Its novel core achieves efficient computation by exploiting many threads through a novel synthesis of three mechanisms: 1) Event-Driven Scheduling (EDS), 2) Software-controlled Lightweight Threading (SLT), and 3) Split-transaction Memory Access with software synchronization (SMA). These mechanisms enable fine-grained computations and scalable memory-level parallelism, which are fundamental to achieving scalable performance on a diverse set of applications. Our evaluation examines these mechanisms in a 2,048-core UpDown chip, showing significant speedups over multicore and manycore architectures.

Acknowledgments

This research is based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), through the Advanced Graphical Intelligence Logical Computing Environment (AGILE) research program, under Army Research Office (ARO) contract number W911NF22C0082. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the ODNI, IARPA, or the U.S. Government.

This work is also supported in part by a Computation Innovation Fellows Award. Thanks also to the entire UChicago UpDown team, and also our collaborators at Purdue University and Tactical Computing Laboratories.

References

[1] 2022. AGILE: ADVANCED GRAPHIC INTELLIGENCE LOGICAL COMPUTING ENVIRONMENT Program. https://www.iarpa.gov/research-programs/agile.

[2] Sriram Aananthakrishnan, Shamsul Abedin, Vincent Cave, Fabio Checconi, Kristof Du Bois, Stijn Eyerman, Joshua B. Fryman, Wim Heirman, Jason Howard, Ibrahim Hur, Samkit Jain, Marek M. Landowski, Kevin Ma, Jarrod Nelson, Robert Pawlowski, Fabrizio Petrini Sebastian Szkoda, Sanjaya Tayal, Jesmin Jahan Tithi, and Yves Vandriessche. 2025. PIUMA: Programmable Integrated Unified Memory Architecture. arXiv:2010.06277 [cs.AR] https://arxiv.org/abs/2010.06277

[3] Advanced Micro Devices, Inc. 2025. 5th Generation AMD EPYC™ 9005 Series Processors. https://www.amd.com/en/products/processors/server/epyc/9005-series.html

[4] Advanced Micro Devices, Inc. 2025. AMD64 Technology AMD64 Architecture Programmer's Manual Volumes 1–5. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/40332.pdf.

[5] Advanced Micro Devices, Inc. 2025. *Introducing AMD CDNA™ 3 Architecture.* White Paper. Advanced Micro Devices, Inc. https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf

[6] Masab Ahmad, Halit Dogan, José A. Joao, and Omer Khan. 2020. In-Hardware Moving Compute to Data Model to Accelerate Thread Synchronization on Large Multicores. *IEEE Micro* 40, 1 (2020), 83–92. doi:10.1109/MM.2019.2955079

[7] Tero Aittokallio and Benno Schwikowski. 2006. Graph-based methods for analysing networks in cell biology. *Briefings in Bioinformatics* 7, 3 (09 2006), 243–255. doi:10.1093/bib/bbl022

[8] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. 1990. The Tera computer system. In *Proceedings of the 4th International Conference on Supercomputing*. 1–6.

[9] Amazon Web Services, Inc. 2025. AWS Graviton Processors. https://aws.amazon.com/ec2/graviton

[10] Ampere Computing LLC. 2025. Ampere Processing Platforms. https://amperecomputing.com/products/processors

[11] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM. doi:10.1145/3620665.3640366

[12] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP Benchmark Suite. http://arxiv.org/abs/1508.03619 arXiv:1508.03619 [cs].

[13] Trevor E. Carlson, Wim Heirman, Osman Allam, Stefanos Kaxiras, and Lieven Eeckhout. 2015. The load slice core microarchitecture. *SIGARCH Comput. Archit. News* 43, 3S (June 2015), 272–284. doi:10.1145/2872887.2750407

[14] Tien-Fu Chen and Jean-Loup Baer. 1995. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput.* 44, 5 (1995), 609–623. doi:10.1109/12.381947

[15] Andrew Chien, Andronicus Rajasukumar, Marziyeh Nourian, Yuqing Wang, Tianshuo Su, Chen Zou, and Yuanwei Fang. 2024. *UpDown Accelerator Instruction Set Architecture (ISA) v2.4.* Technical Report TR-2024-03. University of Chicago. https://newtraell.cs.uchicago.edu/research/publications/techreports/TR-2024-03

[16] Andrew A. Chien, Charles Colley, Jianru Ding, Alexander Fell, David F. Gleich, Henry Hoffmann, Moubarak Jeje, Rajat Khandelwal, Yanjing Li, Jose M Monsalve-Diaz, Marziyeh Nourian, Andronicus Rajasukumar, Jiya Su, Tianshuo Su, Yuqing (Ivy) Wang, Wenyi Wang, Ruiqi Xu, and Tianchi Zhang. under review. UpDown: A Supercomputer Co-designed for Scalable Graph Processing. *IEEE Trans. Parallel Distrib. Syst.* (under review).

[17] cuGraph Development Team. 2025. *cuGraph: GPU-accelerated Graph Analytics Library (RAPIDS).* https://github.com/rapidsai/cugraph

[18] 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)*. IEEE Computer Society, Los Alamitos, CA, USA, 595–608. doi:10.1109/ISCA52012.2021.00053

[19] W. J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills. 1987. Architecture of a Message-Driven Processor. In *Proceedings of the 14th Annual International Symposium on Computer Architecture* (Pittsburgh, Pennsylvania, USA) *(ISCA '87)*. Association for Computing Machinery, New York, NY, USA, 189–196. doi:10.1145/30350.30372

[20] William J. Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, Richard Lethin, Michael Noakes, Peter Nuth, Ellen Spertus, Deborah Wallach, D. Scott Wills, Andrew Chang, and John Keen. 1998. Retrospective: The J-Machine. In *25 Years of the International Symposia on Computer Architecture (Selected Papers)* (Barcelona, Spain) *(ISCA '98)*. Association for Computing Machinery, New York, NY, USA, 54–58. doi:10.1145/285930.285953

[21] Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawai, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, Aporva Amarnath, Bandhav Veluri, Paul Gao, Anuj Rao, Gai Liu, Rajesh K. Gupta, Zhiru Zhang, Ronald Dreslinski, Christopher Batten, and Michael Bedford Taylor. 2018. The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips. *IEEE Micro* 38, 2 (2018), 30–41. doi:10.1109/MM.2018.022071133

[22] Timothy A. Davis and Yifan Hu. 2011. The university of Florida sparse matrix collection. 25 pages. doi:10.1145/2049662.2049663

[23] J. del Cuvillo, W. Zhu, Z. Hu, and G.R. Gao. 2005. TiNy threads: a thread virtual machine for the Cyclops64 cellular architecture. In *19th IEEE International Parallel and Distributed Processing Symposium*. 8 pp.–. doi:10.1109/IPDPS.2005.434

[24] Saurabh Dighe, Sriram Vangal, Nitin Borkar, and Shekhar Borkar. 2009. LESSONS LEARNED FROM THE 80-CORE TERA-SCALE RESEARCH PROCESSOR. *Intel technology journal* 13, 4 (2009).

[25] David R. Ditzel and the Esperanto team. 2022. Accelerating ML Recommendation With Over 1,000 RISC-V/Tensor Processors on Esperanto's ET-SoC-1 Chip. *IEEE Micro* 42, 3 (2022), 31–38. doi:10.1109/MM.2022.3140674

[26] Timothy Dysart, Peter Kogge, Martin Deneroff, Eric Bovell, Preston Briggs, Jay Brockman, Kenneth Jacobsen, Yujen Juan, Shannon Kuntz, Richard Lethin, Janice McMahon, Chandra Pawar, Martin Perrigo, Sarah Rucker, John Ruttenberg, Max Ruttenberg, and Steve Stein. 2016. Highly Scalable Near Memory Processing with Migrating Threads on the Emu System Architecture. In *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*. 2–9. doi:10.1109/IA3.2016.007

[27] S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, R.L. Stamm, and D.M. Tullsen. 1997. Simultaneous multithreading: a platform for next-generation processors. *IEEE Micro* 17, 5 (1997), 12–19. doi:10.1109/40.621209

[28] Yuanwei Fang, Tung T. Hoang, Michela Becchi, and Andrew A. Chien. 2015. Fast Support for Unstructured Data Processing: The Unified Automata Processor. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 533–545. doi:10.1145/2830772.2830809

[29] Yuanwei Fang, Chen Zou, Aaron J. Elmore, and Andrew A. Chien. 2017. UDP: a programmable accelerator for extract-transform-load workloads and more. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 55–68. doi:10.1145/3123939.3123983

[30] Alexander Fell, Yuqing Wang, Tianshuo Su, Marziyeh Nourian, Wenyi Wang, Jose M. Monsalve-Diaz, Andronicus Samsundar Rajasukumar, Jiya Su, Ruiqi Xu, Rajat Khandelwal, Tianchi Zhang, David Gleich, Yanjing Li, Hank Hoffmann, and Andrew A. Chien. 2025. KVMSR+UDWeave: Extreme-Scaling with Fine-grained Parallelism on the UpDown Graph Supercomputer. In *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25)*. Association for Computing Machinery, New York, NY, USA, 1243–1262. doi:10.1145/3731599.3767499

[31] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. 1992. Hiding memory latency using dynamic scheduling in shared-memory multiprocessors. *ACM SIGARCH Computer Architecture News* 20, 2 (May 1992), 22–33. doi:10.1145/146628.139678

[32] Fred G Gustavson. 1978. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)* 4, 3 (1978), 250–269.

[33] Maurice Herlihy. 1991. Wait-free synchronization. 13, 1 (Jan. 1991), 124–149. doi:10.1145/114005.102808

[34] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann, Burlington, MA.

[35] Zhigang Hu, Margaret Martonosi, and Stefanos Kaxiras. 2003. TCP: Tag Correlating Prefetchers. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA '03)*. IEEE Computer Society, USA, 317.

[36] Intel Corporation. 2024. Intel® Xeon® 6960P Processor. Retrieved 2025-07-28 from https://www.intel.com/content/www/us/en/products/sku/240775/intel-xeon-6960p-processor-432m-cache-2-70-ghz/specifications.html

[37] Intel Corporation. 2025. Intel® VTune™ Profiler. Retrieved 2025-07-28 from https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

[38] Intel Corporation. 2025. Intel® Math Kernel Library (Intel® MKL). https://software.intel.com/en-us/intel-mkl.

[39] Intel Corporation. 2025. Intel® 64 and IA-32 Architectures Software Developer Manuals. https://www.intel.com/content/www/us/en/developer/articles/

technical/intel-sdm.html.

[40] Nan Jiang, Daniel U. Becker, George Michelogiannakis, James Balfour, Brian Towles, D. E. Shaw, John Kim, and William J. Dally. 2013. A detailed and flexible cycle-accurate Network-on-Chip simulator. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 86–96. doi:10.1109/ISPASS.2013.6557149

[41] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. 2017. HBM (High Bandwidth Memory) DRAM Technology and Architecture. In *2017 IEEE International Memory Workshop (IMW)*. 1–4. doi:10.1109/IMW.2017.7939084

[42] Dai Cheol Jung, Max Ruttenberg, Paul Gao, Scott Davidson, Daniel Petrisko, Kangli Li, Aditya K Kamath, Lin Cheng, Shaolin Xie, Peitian Pan, Zhongyuan Zhao, Zichao Yue, Bandhav Veluri, Sripathi Muralitharan, Adrian Sampson, Andrew Lumsdaine, Zhiru Zhang, Christopher Batten, Mark Oskin, Dustin Richmond, and Michael Bedford Taylor. 2024. Scalable, Programmable and Dense: The HammerBlade Open-Source RISC-V Manycore . In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 770–784. doi:10.1109/ISCA59077.2024.00061

[43] Sameer Kumar, Amith R. Mamidala, Daniel A. Faraj, Brian Smith, Michael Blocksome, Bob Cernohous, Douglas Miller, Jeff Parker, Joseph Ratterman, Philip Heidelberger, Dong Chen, and Burkhard Steinmacher-Burrow. 2012. PAMI: A Parallel Active Message Interface for the Blue Gene/Q Supercomputer. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 763–773. doi:10.1109/IPDPS.2012.73

[44] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th International Conference on World Wide Web* (Raleigh, North Carolina, USA) *(WWW '10)*. Association for Computing Machinery, New York, NY, USA, 591–600. doi:10.1145/1772690.1772751

[45] Kartik Lakhotia, Laura Monroe, Kelly Isham, Maciej Besta, Nils Blach, Torsten Hoefler, and Fabrizio Petrini. 2024. PolarStar: Expanding the Horizon of Diameter-3 Networks. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures* (Nantes, France) *(SPAA '24)*. Association for Computing Machinery, New York, NY, USA, 345–357. doi:10.1145/3626183.3659975

[46] Sanghoon Lee, Devesh Tiwari, Yan Solihin, and James Tuck. 2011. HAQu: Hardware-accelerated queueing for fine-grained threading on a chip multiprocessor. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. 99–110. doi:10.1109/HPCA.2011.5749720

[47] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining* (Chicago, Illinois, USA) *(KDD '05)*. Association for Computing Machinery, New York, NY, USA, 177–187. doi:10.1145/1081870.1081893

[48] H.M. Levy, Jack L. Lo, J.S. Emer, R.L. Stamm, S.J. Eggers, and D.M. Tullsen. 1996. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *23rd Annual International Symposium on Computer Architecture (ISCA'96)*. 191–191. doi:10.1145/232973.232993

[49] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. 2011. CACTI-P: architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *Proceedings of the International Conference on Computer-Aided Design* (San Jose, California) *(ICCAD '11)*. IEEE Press, 694–701.

[50] Sean Lie. 2023. Cerebras Architecture Deep Dive: First Look Inside the Hardware/Software Co-Design for Deep Learning. *IEEE Micro* 43, 3 (2023), 18–30. doi:10.1109/MM.2023.3256384

[51] David Luebke. 2008. CUDA: Scalable parallel programming for high-performance scientific computing. In *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*. 836–838. doi:10.1109/ISBI.2008.4541126

[52] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. 2007. Challenges in Parallel Graph Processing. *Parallel Processing Letters* 17 (03 2007), 5–20. doi:10.1142/S0129626407002843

[53] Haocong Luo, Yahya Can Tuğrul, F. Nisa Bostancı, Ataberk Olgun, A. Giray Yağlıkçı, , and Onur Mutlu. 2023. Ramulator 2.0: A Modern, Modular, and Extensible DRAM Simulator.

[54] Simone Manoni, Paul Scheffler, Alfio Di Mauro, Luca Zanatta, Andrea Acquaviva, Luca Benini, and Andrea Bartolini. 2024. NARS: Neuromorphic Acceleration through Register-Streaming Extensions on RISC-V Cores. In *Proceedings of the 21st ACM International Conference on Computing Frontiers: Workshops and Special Sessions* (Ischia, Italy) *(CF '24 Companion)*. Association for Computing Machinery, New York, NY, USA, 79–82. doi:10.1145/3637543.3652879

[55] John D. McCalpin. 2023. Bandwidth Limits in the Intel Xeon Max (Sapphire Rapids with HBM) Processors. In *High Performance Computing: ISC High Performance 2023 International Workshops, Hamburg, Germany, May 21–25, 2023, Revised Selected Papers* (Hamburg, Germany). Springer-Verlag, Berlin, Heidelberg, 403–413. doi:10.1007/978-3-031-40843-4_30

[56] Daniele Merico, David Gfeller, and Gary D Bader. 2009. How to visually interpret biological data using networks. *Nature biotechnology* 27, 10 (2009), 921–924.

[57] O. Mutlu, J. Stark, C. Wilkerson, and Y.N. Patt. 2003. Runahead execution: an alternative to very large instruction windows for out-of-order processors. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* IEEE Computer Society, Los Alamitos, CA, USA, 129–140. doi:10.1109/HPCA.2003.1183532

[58] R. S. Nikhil, G. M. Papadopoulos, and Arvind. 1992. T: A Multithreaded Massively Parallel Architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture* (Queensland, Australia) *(ISCA '92)*. Association for Computing Machinery, New York, NY, USA, 156–167. doi:10.1145/139669.139715

[59] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. 1993. The J-Machine Multicomputer: An Architectural Evaluation. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (San Diego, California, USA) *(ISCA '93)*. Association for Computing Machinery, New York, NY, USA, 224–235. doi:10.1145/165123.165158

[60] NVIDIA. 2023. *NVIDIA Blackwell Architecture.* Technical Brief. NVIDIA. https://resources.nvidia.com/en-us-blackwell-architecture

[61] NVIDIA Corporation. [n. d.]. NVIDIA Nsight Compute. Retrieved 2025-07-28 from https://developer.nvidia.com/nsight-compute

[62] Oleksandr Pavlyk, Denis Nagorny, Andres Guzman Ballen, Anton Malakhov, Hai Liu, Ehsan Totoni, Todd A. Anderson, and Sergey Maidanov. 2017. Accelerating Scientific Python with Intel Optimizations. In *Proceedings of the 16th Python in Science Conference*, Katy Huff, David Lippa, Dillon Niederhut, and M Pacer (Eds.). 106 – 112. doi:10.25080/shinma-7f4c6e7-00f

[63] Andreas Olofsson. 2016. Epiphany-V: A 1024 processor 64-bit RISC System-On-Chip. arXiv:1610.01832 [cs.AR] https://arxiv.org/abs/1610.01832

[64] Marcelo Orenes-Vera, Aninda Manocha, Jonathan Balkind, Fei Gao, Juan L. Aragón, David Wentzlaff, and Margaret Martonosi. 2022. Tiny but mighty: designing and realizing scalable latency tolerance for manycore SoCs. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ACM, New York New York, 817–830. doi:10.1145/3470496.3527400

[65] Marcelo Orenes-Vera, Esin Tureci, David Wentzlaff, and Margaret Martonosi. 2023. Dalorex: A Data-Local Program Execution and Architecture for Memory-bound Applications. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 718–730. doi:10.1109/HPCA56546.2023.10071089

[66] Steven A. Przybylski. 1990. *Cache and memory hierarchy design: a performance-directed approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[67] Andronicus Rajasukumar, Tianchi Zhang, Ruiqi Xu, and Andrew A. Chien. 2024. UpDown: A Novel Architecture for Unlimited Memory Parallelism. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '24)*. Association for Computing Machinery, New York, NY, USA, 61–77. doi:10.1145/3695794.3695801

[68] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. *Graph databases: new opportunities for connected data.* " O'Reilly Media, Inc.".

[69] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. 2010. Flexible architectural support for fine-grain scheduling. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, Pennsylvania, USA) *(ASPLOS XV)*. Association for Computing Machinery, New York, NY, USA, 311–322. doi:10.1145/1736020.1736055

[70] Paul Scheffler, Thomas Benz, Viviane Potocnik, Tim Fischer, Luca Colagrande, Nils Wistoff, Yichao Zhang, Luca Bertaccini, Gianmarco Ottavi, Manuel Eggimann, Matheus Cavalcante, Gianna Paulin, Frank K. Gürkaynak, Davide Rossi, and Luca Benini. 2025. Occamy: A 432-Core Dual-Chiplet Dual-HBM2E 768-DP-GFLOP/s RISC-V System for 8-to-64-bit Dense and Sparse Computing in 12-nm FinFET. *IEEE Journal of Solid-State Circuits* 60, 4 (2025), 1324–1338. doi:10.1109/JSSC.2025.3529249

[71] Paul Scheffler, Florian Zaruba, Fabian Schuiki, Torsten Hoefler, and Luca Benini. 2023. Sparse Stream Semantic Registers: A Lightweight ISA Extension Accelerating General Sparse Linear Algebra. *IEEE Transactions on Parallel and Distributed Systems* 34, 12 (2023), 3147–3161. doi:10.1109/TPDS.2023.3322029

[72] Brian C. Schwedock and Nathan Beckmann. 2024. Leviathan: A Unified System for General-Purpose Near-Data Computing. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, Los Alamitos, CA, USA, 1278–1294. doi:10.1109/MICRO61859.2024.00095

[73] Brian C. Schwedock, Piratach Yoovidhya, Jennifer Seibert, and Nathan Beckmann. 2022. täkō: a polymorphic cache hierarchy for general-purpose optimization of data movement. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ACM, New York New York, 42–58. doi:10.1145/3470496.3527379

[74] Burton J. Smith. 1982. Architecture And Applications Of The HEP Multiprocessor Computer System. In *Real-Time Signal Processing IV*, Tien F. Tao and Tien F. Tao (Eds.), Vol. 0298. International Society for Optics and Photonics, SPIE, 241–248. doi:10.1117/12.932535

[75] J.E. Smith and A.R. Pleszkun. 1988. Implementing precise interrupts in pipelined processors. *IEEE Trans. Comput.* 37, 5 (1988), 562–573. doi:10.1109/12.4607

[76] James E Smith. 1982. Decoupled Access/Execute Computer Architectures. In *Proceedings of the 9th Annual Symposium on Computer Architecture* (Austin,

Texas, USA) *(ISCA '82)*. IEEE Computer Society Press, Washington, DC, USA, 112–119.

[77] Aaron Stillmaker and Bevan Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180nm to 7nm. *Integration* 58 (2017), 74–81. doi:10.1016/j.vlsi.2017.02.002

[78] Jiya Su, Alexander Fell, Andronicus Rajasukumar, David F. Gleich, and Andrew A. Chien. 2025. Scaling Triangle Counting and K-Truss on the UpDown Architecture. In *2025 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. doi:10.1109/HPEC67600.2025.11196311

[79] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. 2009. Accelerating critical section execution with asymmetric multi-core architectures. *SIGARCH Comput. Archit. News* 37, 1 (March 2009), 253–264. doi:10.1145/2528521.1508274

[80] Synopsys, Inc. 2025. Synopsys University Program – Teaching Resources. https://www.synopsys.com/community/university-program/teaching-resources.html

[81] M.B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, Jae-Wook Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. 2002. The Raw microprocessor: a computational fabric for software circuits and general-purpose programs. *IEEE Micro* 22, 2 (2002), 25–35. doi:10.1109/MM.2002.997877

[82] RAPIDS Development Team. 2023. *RAPIDS: Libraries for End-to-End GPU Data Science*. https://rapids.ai

[83] The Graph 500 Steering Committee. [n. d.]. Graph 500 Benchmark. Retrieved 2026-03-09 from https://graph500.org/

[84] M. R. Thistle and B. J. Smith. 1988. A processor architecture for horizon. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing* (Orlando, Florida, USA) *(Supercomputing '88)*. IEEE Computer Society Press, Washington, DC, USA, 35–41.

[85] A. Tumeo and J. Feo. 2015. Irregular Applications: From Architectures to Algorithms [Guest editors; introduction]. *Computer* 48, 08 (aug 2015), 14–16. doi:10.1109/MC.2015.233

[86] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. 1992. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture* (Queensland, Australia) *(ISCA '92)*. Association for Computing Machinery, New York, NY, USA, 256–266. doi:10.1145/139669.140382

[87] Luming Wang, Xu Zhang, Songyue Wang, Zhuolun Jiang, Tianyue Lu, Mingyu Chen, Siwei Luo, and Keji Huang. 2024. Asynchronous Memory Access Unit: Exploiting Massive Parallelism for Far Memory Access. *ACM Trans. Archit. Code Optim.* 21, 3, Article 55 (Sept. 2024), 28 pages. doi:10.1145/3663479

[88] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2020. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. arXiv:1909.01315 [cs.LG] https://arxiv.org/abs/1909.01315

[89] Yuqing Wang, Charles Colley, Brian Wheatman, Jiya Su, David F. Gleich, and Andrew A. Chien. 2025. How Fast Can Graph Computations Go on Fine-grained Parallel Architectures. arXiv:2507.00949 [cs.DC] https://arxiv.org/abs/2507.00949

[90] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: a high-performance graph processing library on the GPU. *SIGPLAN Not.* 51, 8, Article 11 (feb 2016), 12 pages. doi:10.1145/3016078.2851145

[91] Yuqing Wang, Swann Perarnau, and Andrew A. Chien. 2024. UpDown: Combining Scalable Address Translation with Locality Control. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1014–1024. doi:10.1109/SCW63240.2024.00141

[92] Yuqing Wang, Andronicus Rajasukumar, Tianshuo Su, Marziyeh Nourian, Jose M. Monsalve Diaz, Ahsan Pervaiz, Jerry Ding, Charles Colley, Wenyi Wang, Yanjing Li, David F. Gleich, Hank Hoffmann, and Andrew A. Chien. 2025. Efficiently Exploiting Irregular Parallelism Using Keys at Scale. In *Languages and Compilers for Parallel Computing: 36th International Workshop, LCPC 2023, Lexington, KY, USA, October 11–13, 2023, Revised Selected Papers* (Lexington, KY, USA). Springer-Verlag, Berlin, Heidelberg, 78–95. doi:10.1007/978-3-032-02436-7_6

[93] Yipeng Wang, Ren Wang, Andrew Herdrich, James Tsai, and Yan Solihin. 2016. CAF: Core to core Communication Acceleration Framework. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 351–362. doi:10.1145/2967938.2967954

[94] 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 (ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 736–749. doi:10.1145/3307650.3322229

[95] Brian Wheatman and Andrew A. Chien. 2025. Scalable Graph Algorithms on Distributed UpDown Accelerators. In *2025 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–8. doi:10.1109/HPEC67600.2025.11196373

[96] x86-Guide. n.d.. Intel® Xeon® Gold 6138 — CPU specifications (x86-Guide). https://www.x86-guide.net/en/cpu/Intel-Xeon-Gold-6138-cpu-no6271.html

[97] Ruiqi Xu. 2025. Accelerating GNN Aggregation Using a Vertex-Centric Approach. https://people.cs.uchicago.edu/~aachien/lssg/research/10x10/Ruiqi_Xu_MS.pdf

[98] Jaewon Yang and Jure Leskovec. 2012. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics* (Beijing, China) *(MDS '12)*. Association for Computing Machinery, New York, NY, USA, Article 3, 8 pages. doi:10.1145/2350190.2350193

[99] Shimeng Yu and Tae-Hyeon Kim. 2024. Semiconductor Memory Technologies: State-of-the-Art and Future Trends. *Computer* 57, 4 (2024), 150–154. doi:10.1109/MC.2024.3363269

[100] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, Waikiki Hawaii, 178–190. doi:10.1145/2830772.2830807

[101] Florian Zaruba, Fabian Schuiki, Torsten Hoefler, and Luca Benini. 2021. Snitch: A Tiny Pseudo Dual-Issue Processor for Area and Energy Efficient Execution of Floating-Point Intensive Workloads. *IEEE Trans. Comput.* 70, 11 (Nov. 2021), 1845–1860. doi:10.1109/tc.2020.3027900

[102] Dan Zhang, Xiaoyu Ma, Michael Thomson, and Derek Chiou. 2018. Minnow: Lightweight Offload Engines for Worklist Management and Worklist-Directed Prefetching. *ACM SIGPLAN Notices* 53, 2 (March 2018), 593–607. doi:10.1145/3296957.3173197

[103] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. 2017. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*. 293–302. doi:10.1109/BigData.2017.8257937

[104] Yu Zhang, Xiaofei Liao, Hai Jin, Ligang He, Bingsheng He, Haikun Liu, and Lin Gu. 2021. DepGraph: A Dependency-Driven Accelerator for Efficient Iterative Graph Processing. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 371–384. doi:10.1109/HPCA51647.2021.00039 ISSN: 2378-203X.

[105] Jerry Zhao, Animesh Agrawal, Borivoje Nikolic, and Krste Asanović. 2022. Constellation: An Open-Source SoC-Capable NoC Generator. In *2022 15th IEEE/ACM International Workshop on Network on Chip Architectures (NoCArc)*. IEEE, 1–7.