THE UNIVERSITY OF CHICAGO

GENERATING UNLIMITED MEMORY LEVEL PARALLELISM WITH UPDOWN

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCE
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
TIANCHI ZHANG

CHICAGO, ILLINOIS
GRADUATION DATE

Dedication Text

Epigraph Text

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

ABSTRACT

With the rapid growth of machine learning hardware demand, High Bandwidth Memory (HBM) has emerged as a prominent choice for architectural and system design considerations. HBM3e achieves a remarkable memory bandwidth of 1.2 TB/s per stack and is projected to attain 2+ TB/s on the roadmap. This substantial bandwidth enhancement presents a compelling opportunity to accelerate memory bandwidth-critical applications. Nevertheless, current CPUs with a multilevel cache hierarchy lack the requisite memory level parallelism to fully saturate such high bandwidths.

We employ the Memory Parallelism Abstract Machine (MPAM) [83] to highlight the inherent namespace that limits the memory level parallelism in architectures. Through MPAM, we investigate the hardware constraints of four commercial platforms. We also evaluate the UpDown architecture using MPAM. The UpDown architecture mitigates these namespace limitations and achieves unlimited memory level parallelism through the split transaction of DRAM requests, explicit compute-name synchronization of memory responses, and efficient memory parallelism scaling.

As a result, a single UpDown lane can attain 3.5 times memory level parallelism over a single Out-of-Order core with a significantly smaller footprint. A 256-lane UpDown system achieves approximately 34 to 172 times speedup on STREAM, SPMV, and CNN compared to a single Out-of-Order core with prefetching, all while utilizing a comparable area. Furthermore, we demonstrate that UpDown can effectively scale out and saturate HBM3e and future HBM memory bandwidths at a low cost.

CHAPTER 1

INTRODUCTION

For decades, the design of CPU-based computer systems has focused on reducing memory bandwidth requirements due to the high cost and long latency associated with memory access. As CPU clock speeds increased, deep memory hierarchies emerged as a solution to optimize memory operations. These hierarchies effectively filtered the program's memory requests, reducing the Average Memory Access Time (AMAT) and limiting the number of requests needed to access the slower DRAM. This approach reduced the reliance on memory bandwidth and improved overall system efficiency.

One reason for the need to reduce memory access is that the DRAM memory (e.g., DDRs [36, 89], LPDDRs [56, 87], etc.) only provides low bandwidth, as it historically used inexpensive packaging and low-pin-count interfaces. This packaging design is the reason why a large number of memory packages are required to achieve a certain level of memory bandwidth. For example, most DDR systems have only reached a few hundred GB/s of memory bandwidth [31, 37, 94].

In recent years, the emergence of wide memory interfaces with thousands of IO's has significantly altered the landscape [53, 79]. These wide interfaces enable some DRAM chips to achieve bandwidths as high as 1.2 TB/s [63], effectively leveraging the band-level/rank-level parallelism inherent in DRAM. In modern commercial products, an 8-stack system can provide over 8 TB/s of bandwidth in a single package [97]. Therefore, stacked DRAM technologies have made memory bandwidth abundant. The green line depicted in Figure 1.1 represents the memory bandwidth of the current GPUs integrated with HBM, while the red line represents the memory bandwidth supported by the commercial CPU system. Figure 1.1 illustrates the substantial potential benefit that HBM can bring. Conversely, the CPU is unable to meet this demand due to its inefficiency in generating exceptional memory requests.

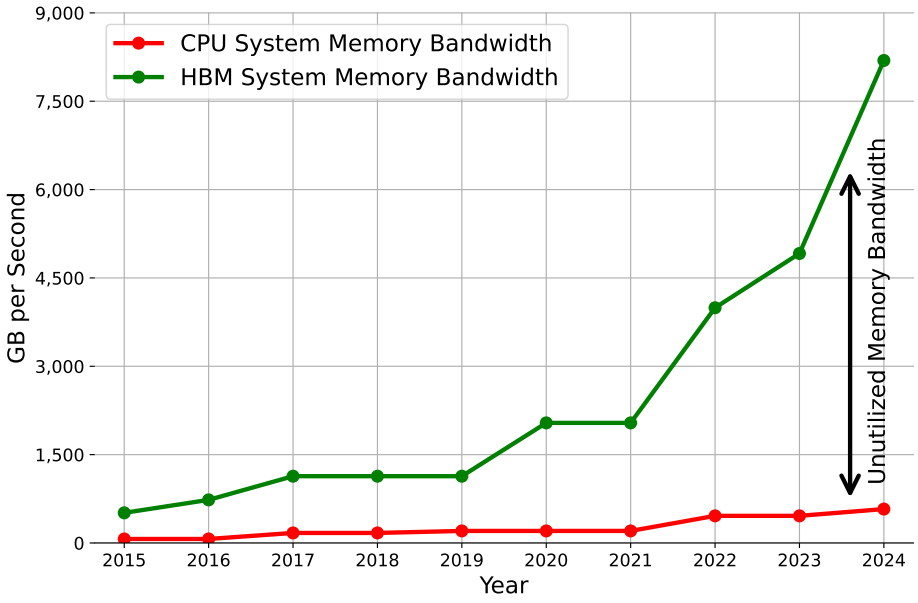


Figure 1.1: CPU cannot utilize HBM memory bandwidth. CPU memory systems [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] are selected from the most performant commercial server CPUs of each year. HBM systems are [73, 74, 75, 76, 77, 101].

Due to the long latency of DRAM, achieving high memory bandwidth requires compute elements to generate significant memory-level parallelism by issuing a large number of concurrent outstanding memory requests. This is a shift from the traditional goals of memory hierarchy design, which prioritize reducing latency and minimizing memory traffic [50, 80]. In light of this new landscape of abundant memory bandwidth, we explore and assess various methods to effectively generate memory parallelism.

To structure the evaluation, we adapt the Memory Parallelism Abstract Machine (MPAM) model from [83] to capture the fundamental limits and resources used by conventional architectures to generate memory level parallelism. We use MPAM to study four commercial systems. Subsequently, we apply MPAM to the UpDown accelerator, which can achieve unlimited memory level parallelism through split transaction DRAM memory requests, explicit compute-name synchronization of memory requests, and efficient memory parallelism scaling. Finally, we utilize diverse applications to compare the memory parallelism achieved (performance) and power/area (cost) for a range of approaches.

The contributions of the paper include the following:

1. Application of MPAM to four commercial CPUs, using it to characterize their design and performance accurately;

2. Application of MPAM to UpDown, explains that its split transaction DRAM memory request, explicit compute-name synchronization of memory request, and efficient memory parallelism scaling grant it unlimited ability to generate outstanding memory requests;

3. Evaluation shows UpDown efficiently generating memory level parallelism and achieving high performance. 1 UpDown lane achieves memory level parallelism up to 3.5x vs. OOO core with 207x smaller area; 256 UpDown lanes achieve memory level parallelism 26x-79x vs. one OOO core.

In Chapter 2, we describe the background for our work. We present the Memory Parallelism Abstract Machine (MPAM) model in Chapter 3, showing how various architectures map to the model. In Chapter 4, we present the design of the UpDown architecture. In Chapter 5, we describe the methodology used for evaluation and the result. In Chapter 6, we compare the cost of UpDown and other architectures to achieve certain memory level parallelism. Related work is discussed in Chapter 7. Finally, in Chapter 8, we summarize the results and discuss future research directions.

CHAPTER 2

BACKGROUND

2.1 Out-of-Order Execution

The out-of-order execution mechanism is a widely used feature in traditional CPU architecture. This mechanism enables the execution of instructions without being hindered by a memory read or write instruction. A large physical register file with register renaming removes false data dependency on the instructions [91]. Hundreds of lines of reorder buffer, multiple issues, and a couple of memory schedulers allow a wide instruction window to perform static or dynamic scheduling [48, 93, 103]. Large load-store queues are included to order memory accesses and ensure that a certain consistency model is maintained from the out-of-order sequence of execution [47, 88]. These mechanisms enable high instruction parallelism in compute-centric applications, but their size inherently limits the number of outstanding memory requests, as these requests are managed in multiples of these structures. Most of these structures are constructed using Content Addressable Memory (CAM), which incurs significant scaling costs.

2.2 Cache Mechanisms for Memory Parallelism

The introduction of the cache minimizes the number of accesses to the DRAM module. When the CPU requests data from an address accessed previously, the cache can respond with the data if the result of the previous access is still stored there, thereby avoiding the traffic to memory. However, caches are ineffective for applications with low data reuse, resulting in high miss rates and incurring additional costs. Miss Handling Architectures (MHA) in lock-up-free/nonblocking caches support multiple outstanding misses using Miss Status Holding Registers (MSHR) for bookkeeping, which usually uses CAM [44, 57]. To

avoid numerous accesses of the same cache line, MSHR creates a namespace limiting memory level parallelism, controlling the number of outstanding memory requests.

Hardware prefetching is a popular technology embedded in the cache hierarchy [30, 51, 105]. Several sophisticated prefetchers have been proposed and implemented to predict future memory accesses, fetch the value, and respond as cache hits if predicted correctly. However, this prefetching is sharing the namespace with other memory requests. It is also difficult to make prefetching effective in applications with irregular memory accesses [85, 115].

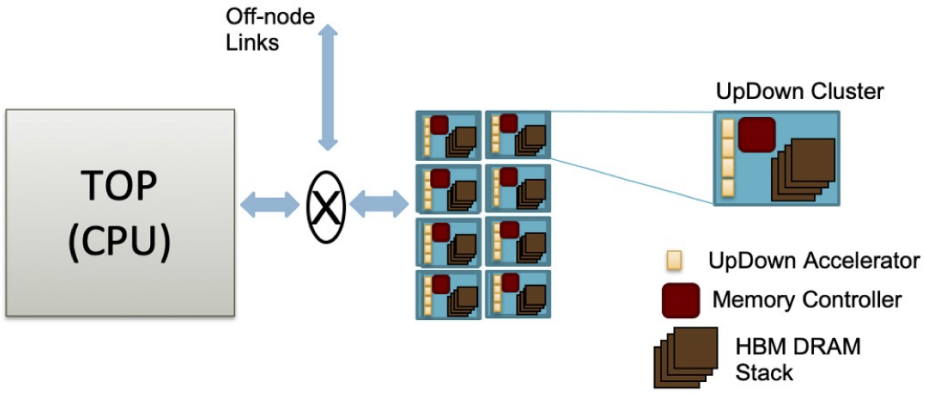
2.3 High Bandwidth Memories

Data-intensive workloads like machine learning and large-scale graph processing propose a large memory bandwidth requirement. They quickly adopt stacked DRAM technologies like High Bandwidth Memories (HBM). HBMs increase bandwidth many-fold by broadening the interface width from 64 bits to 1024 and beyond [63, 64, 89]. Table 2.1 lists current and future generations of HBMs. The current generation of HBM (HBM3e) can provide 1.2 TB/s memory bandwidth, which demands the system to generate ∼1800 concurrent outstanding memory requests to saturate the memory bandwidth. Current architectures cannot saturate such high bandwidth or require prohibitively expensive scaling to do so. One example is the Intel Xeon Max 9480 which, with 56 Sapphire Rapid Cores and four HBM2e stacks, can only achieve 43% peak utilization.

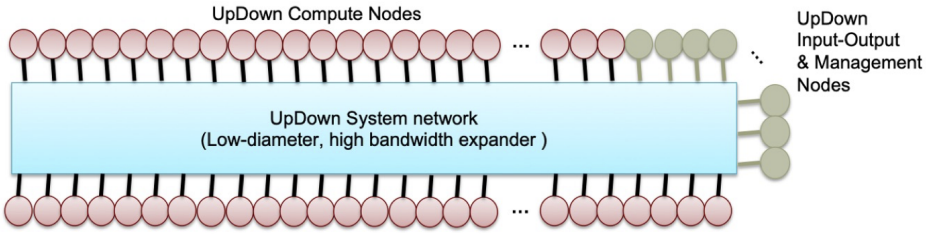
Table 2.1: High Bandwidth Memory technologies [63, 64]

DRAM Technology	**Interface per Stack**	**Max Data Rate Speed per Pin**	**Max Bandwidth per Stack**	**Max Capacity**
HBM2e	8×128 bit	3.6 GT/s	0.4 TB/s	16GB
HBM3	16×64 bit	6.4 GT/s	0.8 TB/s	24GB
HBM3e	16×64 bit	9.6 GT/s	1.2 TB/s	36GB
HBM4	32×64 bit	6.4 GT/s	1.6 TB/s	36-64GB

2.4 UpDown Graph Accelerator



(a) An UpDown node



(b) The UpDown System

Figure 2.1: The UpDown node and system

UpDown [35, 81, 82] is part of a larger system design project funded as part of IARPA's Advanced Graphic Intelligence Logical Computing Environment (AGILE) program [17]. The program aims to create breakthrough performance on irregular applications and graphs with extreme skew and low data reuse [82]. The UpDown architecture builds on previous research architectures, namely the Unified Automata Processor (UAP) and the Unstructured Data Processor (UDP) [42, 43]. UpDown extends the fast symbol processing and multi-way dispatch capabilities in UAP and UDP to generic event-driven execution.

UpDown adapts a Multiple Instruction, Multiple Data (MIMD) parallel computing architecture. Each lane can perform an independent stream of instructions on independent data. Single UpDown node provides huge computing power with a small die area (16 TFLOPS on FP32, about 120 mm^2 [82]). The full UpDown system has 16K nodes, providing 256

petaFLOPS computing power, which is comparable to a state-of-the-art multi-GPU system [21].

UpDown provides high memory bandwidth and high interconnection bandwidth. Each UpDown cluster (256 UpDown lanes) is connected to an HBM3e stack, providing 1.2 TB/s memory bandwidth. Each UpDown node consists of 8 UpDown clusters and 8 HBM3e DRAM stacks with 1.1 TB/s intercluster bandwidth, as illustrated in Figure 2.1(a). The DRAM is shared amongst all of the accelerators and the CPU on the node and can be globally addressed by all of the nodes in the system. The Updown system is 16K Updown nodes connected by a high-bandwidth, low-diameter PolarStar network with a latency of 0.5 microseconds and >50 petabytes/second bisection [58]. Each lane can send messages to any lane in the system, regardless of whether they are in the same physical tile or package. Meanwhile, with the support of global address space [108], each lane can access any part of the global memory space even if an address is located in a far stack (not in the local cluster).

CHAPTER 3

MEMORY PARALLELISM ABSTRACT MACHINE

The memory level parallelism achievable by contemporary architectures is fundamentally constrained by three internal namespaces utilized for bookkeeping and synchronization of outstanding memory requests. These namespace limitations are formally captured and defined by the Memory Parallelism Abstract Machine (MPAM), which accommodates any practical machine across its front end, back end, and shared components [83]. When a memory request originates from one front end based on assembly code, it will initially be limited by the synchronization namespace at the front end. Subsequently, it will be constrained by the outstanding request namespace as it traverses layers of bookkeeping at the back end. Finally, upon reaching shared resources within front ends and back ends, it will be constrained by the shared request namespace. These three namespaces are directly associated with architectural resources, including renamed registers, load/store buffers, MSHRs, shared queues, and so on. These resources impose limitations on the size of the namespaces or render scaling them up prohibitively costly. In this chapter, we present a comprehensive description of MPAM, validate its functionality, and demonstrate its application to derive architectural parameters on commercial platforms.

3.1 Namespace Limiting Memory Level Parallelism

3.1.1 Synchronization Namespace N_{sync}

The processor architecture front end uses names to allow memory responses to synchronize into thread execution. Traditional CPU architecture typically maps the architecture ISA register names to a larger physical register namespace via register renaming to retain only true dependencies between instructions. The physical register namespace will be further bond to a large physical register file, reorder buffers, reservation stations, etc., depending on

the specific implementations. When a load or store instruction is being executed, one of the names and one or more corresponding physical structures will be reserved for synchronization with the memory response message (returning data for load or commit confirmation for store). Thus, due to the limitation of physical resources, the Synchronization Namespace N_{sync} will limit the number of memory references the front end of a processor can send.

3.1.2 Outstanding Request Namespace N_{out}

The processor architecture back end uses a second namespace to bookkeep and track outstanding memory requests. Usually, it is bound to content addressable memories like load store queue and MSHRs in each level of cache, specifically, these outstanding memory requests can be categorized into two groups:

1. Synchronous Request Namespace N_{out_sync}: Synchronous memory requests denote memory requests that are generated by a load or store instructions. Their response will need to be send to the front end and synchronized to thread execution.

2. Asynchronous Request Namespace N_{out_async}: Asynchronous memory requests are not generated directly from an instruction. Usually, they are generated by mechanisms independent to instruction executions like hardware prefetching / direct memory access. Despite not synchronizing with thread execution, they are designed to fill up caches or scratchpads to increase hit rates and reduce memory latencies.

They will share the Outstanding Request Namespace N_{out}, so that

$$N_{out} = N_{out\_sync} + N_{out\_async}$$

Like the Synchronization Namespace N_{sync}, Outstanding Request Namespace N_{out} imposes a limitation of memory reference the back end of a processor can send.

3.1.3 *Shared Request Namespace N_{sh}*

Multi-core architectures usually have shared resources in the memory path, typically L3 cache, last-level cache, system-level caches, shared FIFO queues, and so on. They add a restriction on the sum of outstanding requests from all cores sharing them. For architectures that are not using these components, N_{sh} will be the entire address space.

3.2 MPAM Description

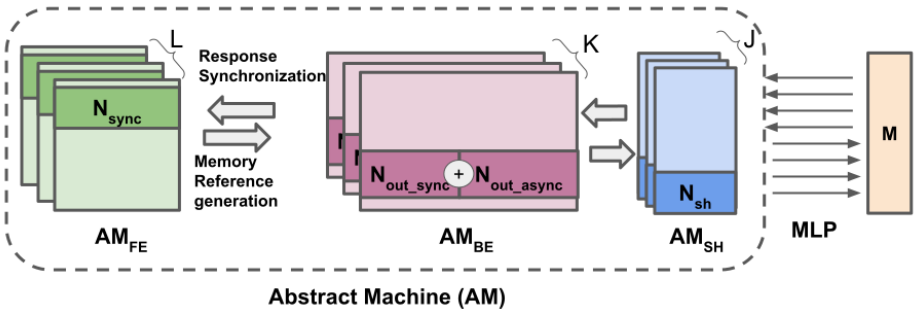


Figure 3.1: Memory Parallelism Abstract Machine captures the fundamental limits to memory parallelism in three separate resource bound namespaces - Synchronization Namespace (N_{sync}), Outstanding Request Namespace(N_{out}) and Shared Request Namespace(N_{sh})

The figure above shows the Memory Parallelism Abstract Machine that highlight three namespace limiting memory level parallelism described in the previous section. The Abstract Machine (AM) has L front ends (AM_{FE}), K back ends (AM_{BE}), and J shared resources (AM_{SH}). Thus there are L copies of N_{sync}, K copies of N_{out}, and J copies of N_{sh} correspondingly. Then we can describe the memory level parallelism (MLP) one machine can achieve as

$$MLP = \frac{1}{cl} \cdot min(min(L \cdot N_{sync} \cdot s_s, K \cdot N_{out\_sync} \cdot s_o) + K \cdot N_{out\_async} \cdot s_o, \\ J \cdot N_{sh} \cdot s_{sh}). \tag{3.1}$$

In the equation, s_s, s_o, s_{sh} are the request sizes (in words) of the three namespaces N_{sync},

N_{out} and N_{sh} respectively, and cl is the cache-line size (in words).

MPAM provides a model to reason about the memory level parallelism given the design of an architecture. It can also be used to demonstrate the bottleneck of the system, and reason about the size of the namespace and the corresponding resources in order to achieve certain MLP capability. Here are some ideas that can be immediately apparent from the model:

1. Different combination of L, K, and J leads to various configuration of single/multi-core architectures. When $L = K = J = 1$, we have a single core design. When $L = K = n_c$ (number of cores) leads to a multi-core architecture that each AM_{BE} been private to each AM_{FE}, for example, describing a private L1 and L2 cache system. When L is a multiple of K, we get an SMT/Hyperthreaded front end with (L/K) threads per core.

2. N_{sync} and N_{out_sync} are closely coordinated. Simply increasing the value of one of them will lead to MLP being bound by the other.

3. N_{out_async} can be increased independently from the AM_{FE}. It can saturate the AM_{SH} just by giving it enough resources.

4. The N_{sh} is an overarching limit on the total outstanding requests. Architectures using such a shared request namespace, such as a shared inclusive LLC / System Level Cache, will be limited by the MSHR size of the cache. Other architectures, like Intel's core microarch (post-Skylake) where the LLC is a non-inclusive victim cache, are not limited by this namespace.

In the following sections, we will first model three microarchitecture, single In-Order core, single Out-of-Order core, and multiple Out-of-Order core, with MPAM. Then we will validate MPAM with measurements of two commercial systems with architectural parameters published. Finally, we apply MPAM to measurements of two other systems to derive some of their architectural parameters.

3.3 Modeling Microarchitectures with MPAM

3.3.1 Single-core In-Order Load/Store Architectures

In this architecture without prefetch, $L = K = 1$. $N_{sync} = R$, the ISA register namespace. $N_{out\_sync} = 1$ since it can only sustain one load or store instruction, i.e. one outstanding request. $N_{out\_async} = 0$ as there is no prefetch. Additionally, $s_s = s_o = 1$. Based on Eqation3.1, we have

$$MLP = \frac{1}{cl} \cdot min(R, 1) = \frac{1}{cl} \tag{3.2}$$

3.3.2 Single-core Out-of-Order Load/Store Architectures

In this architecture, we still have $L = K = 1$. AM_{FE} is usually implemented using several sophisticated mechanisms like speculative execution, dynamic scheduling, etc. These mechanisms are supported using a large number of physical registers R_{phy} using register renaming and reorder buffers. And OOO cores use R_{phy} instead of R to synchronize outstanding requests. Additionally, $s_s = 1$. This gives

$$N_{sync} = R_{phy}$$

The AM_{BE} usually uses a multi-level of cache and implements bookkeeping with MSHRs at each level. MSHRs are tracking the outstanding requests at cache line level of granularity, gives $s_o = cl$. Here, we assume the load store queue is large enough so that $MSHR_{l1}$ will be the bottleneck. We also assume that the hardware prefetching is implemented in L2, with α

indicating the effectiveness of the prefetching. We get

$$N_{out\_sync} = MSHR_{l1}$$
$$N_{out\_async} = \alpha \cdot (MSHR_{l2} - MSHR_{l1})$$

Combining Eqation3.1 and the three equations above, we get

$$MLP = min(R_{phy}/cl, MSHR_{l1}) + \alpha \cdot (MSHR_{l2} - MSHR_{l1}) \tag{3.3}$$

3.3.3 Multi-core Out-of-Order Load/Store Architectures

In this architecture, we include a shared L3 with multiple cores (n_c) demonstrated in the previous section. $L = K = n_c$, $J = 1$. N_{sync}, We still have $s_s = 1$, $s_o = cl$, and

$$N_{sync} = R_{phy} \tag{3.4}$$
$$N_{out\_sync} = MSHR_{l1} \tag{3.5}$$
$$N_{out\_async} = \alpha \cdot (MSHR_{l2} - MSHR_{l1}) \tag{3.6}$$

Here, the MSHR for L3 is also at the granularity of cache line, so that $s_{sh} = cl$, and

$$N_{sh} = MSHR_{l3}$$

These four equation together with Eqation3.1 derives

$$MLP = min(n_c \cdot min(R_{phy}/cl, MSHR_{l1}) + n_c \cdot \alpha(MSHR_{l2} - MSHR_{l1}), MSHR_{l3}) \tag{3.7}$$

By setting $n_c = 1$, Equation3.7(with no L3) reduces back to Equation3.3.

3.4 Validating MPAM's ability to Model Commercial Systems

For this section, we summarized the commercial design we used and their published parameters in Table 3.1. These latencies were measured on commercial platforms using [90, 106].

Table 3.1: Systems for MPAM Validation [60, 31]

Systems	**# Cores**	**Cache**	**Memory**
Intel Platinum 8375C	32	L1d: 48KB per core, 1.4ns; L2: 1.25MB per core, 4.0ns; L3: 54MB on chip, 27.9ns	8 Channel DDR4-3200 103.3ns
AMD EPYC 9R14	96	L1d: 32KB per core, 0.7ns; L2: 1MB per core, 2.4ns; L3: 32MB per CCD, 10.9ns; 386MB on chip, 191.0ns	12 Channel DDR5-4400 129.4ns

3.4.1 Intel Platinum 8375C

We first validate the MPAM on Intel Platinum 8375C, based on the Icelake architecture. For this architecture, [20] suggested that the L3 cache in this architecture is a non-inclusive victim cache. Thus N_{sh} is the entire memory address space. According to [60], we have $MSHR_{l1} = 12$, $MSHR_{l2} = 32$, $R_{phy} = 224$ and $cl = 8$. We put these limits to Equation 3.4, 3.5, 3.6, and 3.7:

$$\begin{aligned} N_{sync} &= 224/8 n_c = 28 n_c \\ N_{out\_sync} &= 12 n_c \\ N_{out\_async} &= \alpha \cdot (32 n_c - 12 n_c) = 20 \alpha n_c \\ MLP &= (12 n_c + 20\alpha) n_c \quad \text{for} \quad 0 \le \alpha \le 1 \end{aligned}$$

We also plot STREAM [68] copy results for forward (ST-C-forward) and a modified version that is reversed and shuffled (ST-C-shuffle). As can be seen, $MLP\_alpha{=}0$ provides a lower bound for when the hardware prefetching is ineffective ($N_{out\_async} = 0$),

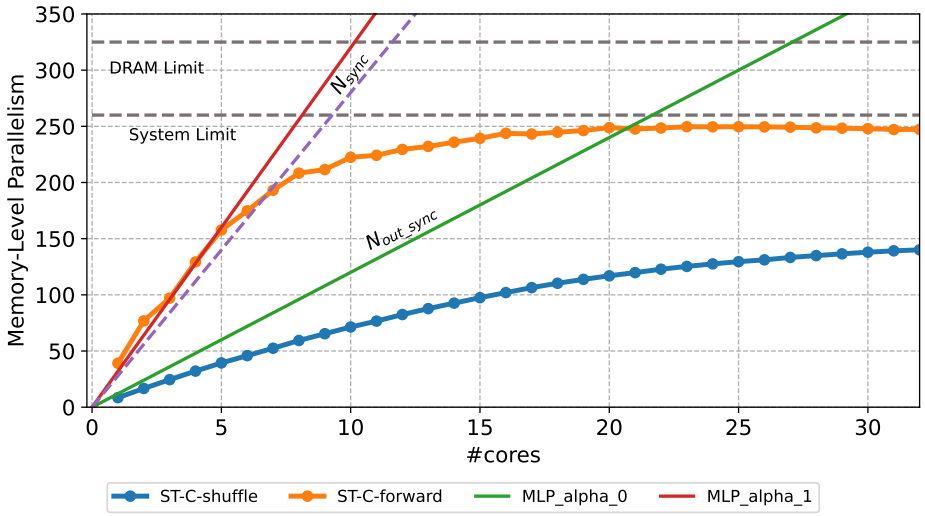


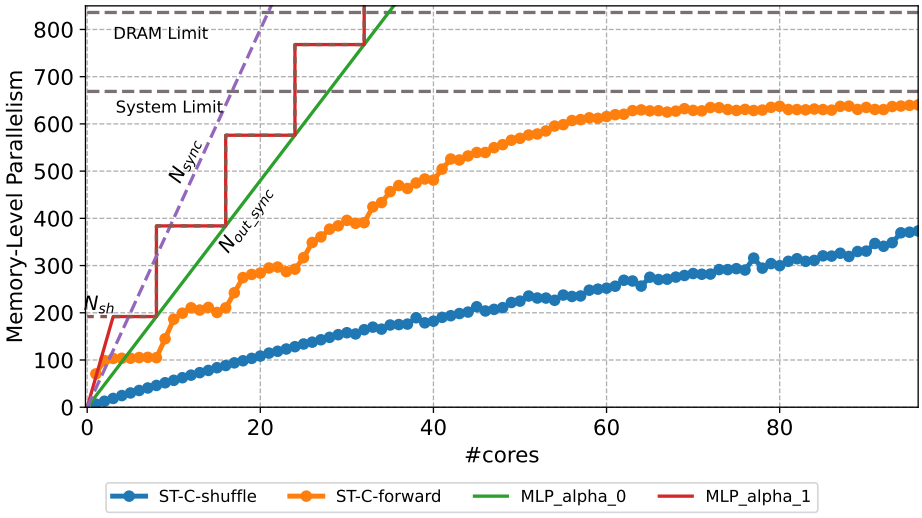
Figure 3.2: MPAM Corroboration with Intel Platinum 8375C Platform using published numbers

and *MLP_alpha=1* provides an upper bound for effective prefetching ($N_{out\_async} = 32n_c$). Finally, we plot the DRAM limit of the platform (8-channels, DDR4-3200).

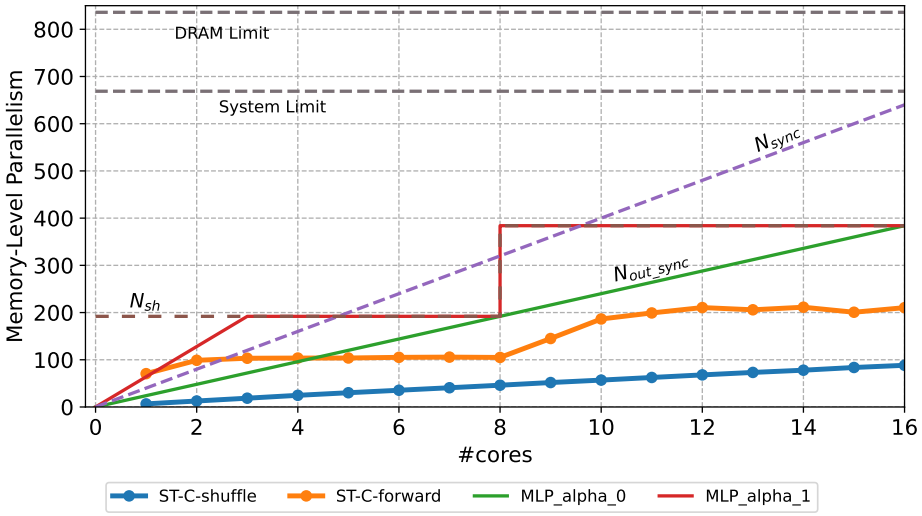
3.4.2 AMD EPYC 9R14

We then validate the MPAM on AMD EPYC 9R14, based on the Zen4 architecture. Using publicly available parameters, we have $MSHR_{l1} = 24$, $MSHR_{l2} = 64$, $MSHR_{l3} = 192$, $R_{phy} = 320$, and $cl = 8$ [84, 31]. This EPYC platform has an l3 per CCD (8 cores), which adds an $N_{sh} = 192\lceil n_c/8 \rceil$ namespace.

$$
\begin{aligned}
N_{sync} &= 320/8n_c = 40n_c \\
N_{out\_sync} &= 24n_c \\
N_{out\_async} &= \alpha \cdot (64n_c - 24n_c) = 40\alpha n_c \\
N_{sh} &= 192\lceil n_c/8 \rceil \\
MLP &= min((24 + 40\alpha)n_c, 192\lceil n_c/8 \rceil) \quad \text{for} \quad 0 \leq \alpha \leq 1
\end{aligned}
$$



(a) MPAM with all 96 cores on AMD EPYC 9R14



(b) Expanded graph for 16 cores to highlight effect of N_{sh} due to per-CCD L3

Figure 3.3: MPAM Corroboration with AMD EPYC 9R14 Platform

Similarly, we plot *MLP_ alpha=0*, *MLP_ alpha=1*, and DRAM limit of the platform (12-channels, DDR5-4400). We further zoom in to the first 16 cores in the second subfigure to highlight the effect of N_{sh}, which is shared by every 8 cores, and how MPAM predicts the effect of shared namespaces at various hierarchies. While we used 192 as $MSHR_{l3}$, the results show that the actual limit is probably lower ($\sim$ 100).

Thus, with these two examples on the Icelake and Zen4 architectures, MPAM effectively

predicts the memory parallelism limits for a given architecture. Designers can use this model to predict the effectiveness of their architectures, the memory parallelism scaling trends with increasing cores, and the cost of achieving a given level of memory parallelism required by applications.

3.5 Using MPAM to derive architectural parameters

In this section, we demonstrate MPAM's applicability in deriving architectural parameters in two commercial systems. Similar to the previous section, we run STREAM copy for forward (ST-C-forward) with an increasing number of hardware cores (without hyper-threading). We also run a modified STREAM version with a reverse shuffled scan of addresses (ST-C-shuffle) to reduce the effect of prefetching. We measure the memory level parallelism achieved in each system and use a least squares estimate of the first few (up to three) cores for projection. We predict the architectural parameters using MPAM and the slope of the line. The commercial architecture design and their published parameters are in Table 3.2 with latencies measured using [90, 106].

Table 3.2: Systems for MPAM Derivation [59, 46]

Systems	**# Cores**	**Cache**	**Memory**
Intel Platinum 8488C	48	L1d: 48KB per core, 1.3ns; L2: 2MB per core, 4.2ns; L3: 105MB on chip, 33.0ns	8 Channel DDR5-4800 127.9ns
AWS Graviton 3	64	L1d: 64KB per core, 1.6ns; L2: 1MB per core, 4.3ns; L3: 32 MB on chip, 30.6ns	8 Channel DDR5-4400 102.2ns

3.5.1 Intel Platinum 8488C

Intel Platinum 8488C is based on the Sapphire Rapids architecture. The forward and backward-and-shuffle scan of STREAM copy results are shown in the following figure, re-

spectively. The least squares fit of the first three cores for these measurements with y-axis interception set as 0 are shown in Equation 3.8 and 3.9. The STREAM benchmark flattens at $\sim 70\%$ of the maximum parallelism of the DRAM system (8 channels, DDR5-4800).

$$MLP_{shuffled} = 16n_c \tag{3.8}$$

$$MLP_{forward} = 44n_c \tag{3.9}$$

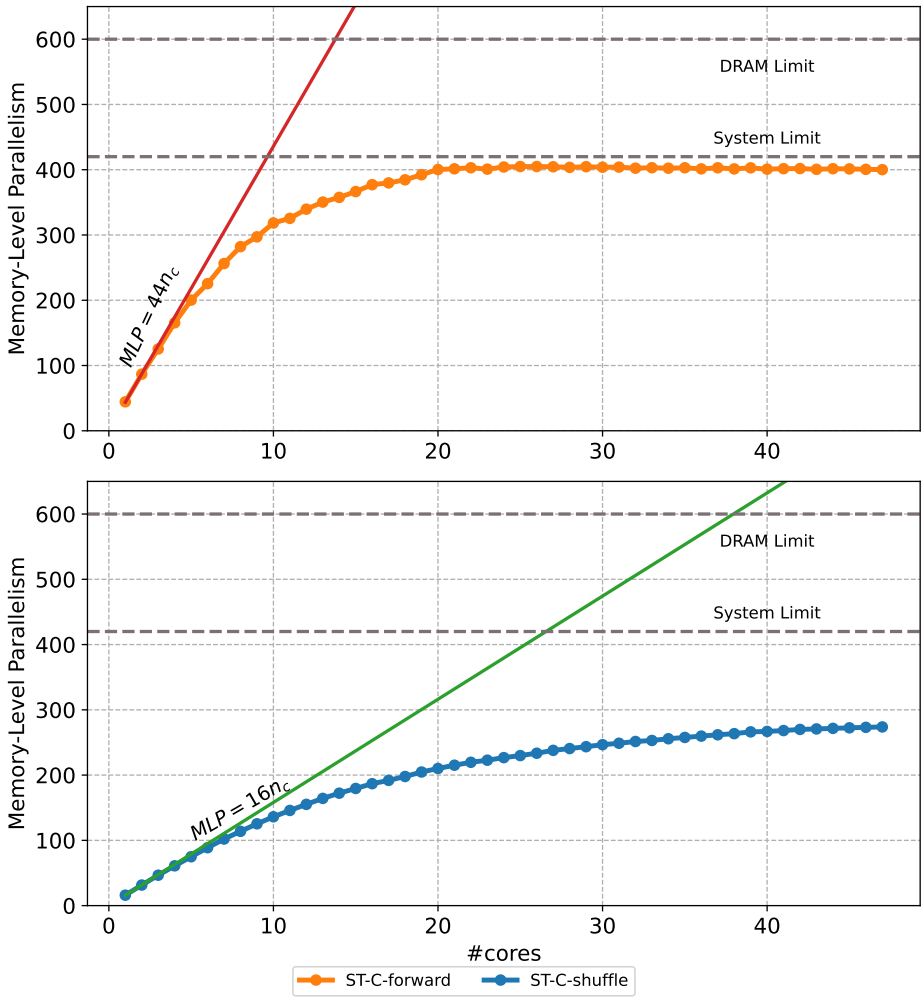


Figure 3.4: Memory Parallelism in Intel(R) Xeon(R) Platinum 8488C with - Forward Scan (top), Shuffle Scan (bottom)

Combining Equation 3.8, 3.7 and set $n_c = 1$, $\alpha = 0$ and remove l3 part, we have

$$MLP_{shuffled} = min(R_{phy}/8, MSHR_{l1}) = 16.$$

Considering that prior Intel microarchitecture already has $R_{phy} > 128$, we assume it also applies here. Thus,

$$MSHR_{l1} = 16.$$

Further putting in Equation 3.9 and not set α to 0 gives

$$MLP_{forward} = min(R_{phy}/8, MSHR_{l1}) + \alpha(MSHR_{l2} - MSHR_{l1}) = 16 + \alpha(MSHR_{l2} - 16) = 44$$

Comparing with the published number of this architecture [59], $MSHR_{l1} = 16$, $MSHR_{l2} = 48$, and $R_{phy} = 512$, we see that MPAM predict the size of MSHR with good accuracy. The off on the size of l2's MSHR suggests that the actual effectness of the hardware prefetched on STREAM copy is $\alpha = 44/48 = 92\%$.

3.5.2 AWS Graviton 3

We apply the same STREAM test on AWS Graviton 3, which is based on the ARM Neoverse V1 architecture. We also apply the least square fitting on the following figure. The STREAM benchmark flattens at $\sim 85\%$ of the maximum parallelism of the DRAM system (8 channels, DDR5-4400).

$$MLP_{shuffled} = 27n_c \tag{3.10}$$

$$MLP_{forward} = 70n_c \tag{3.11}$$

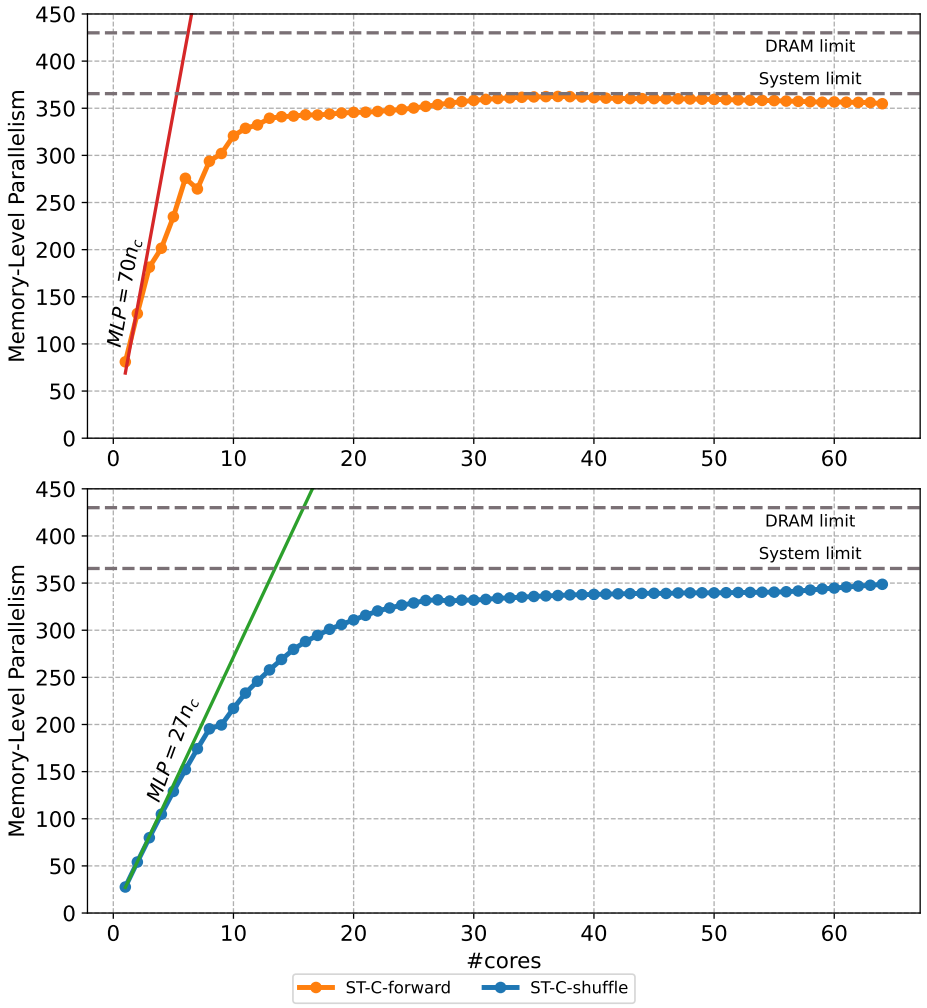


Figure 3.5: Memory Parallelism in AWS Graviton 3

Similarly, we can solve the *MSHR*s,

$$MSHR_{l1} = 27$$

$$MSHR_{l2} = 70$$

The exact values of these parameters are not publicly available, as they are customized by Amazon Web Services. However, MPAM can provide a model for how to estimate these parameters. It further guides the user to choose architecture for different applications that suit different memory level parallelism. Comparing with Table 3.3, we can deduce that Intel

Xeon Platinum 8848C is likely to have larger N_{sync}, and AWS Graviton 3 may have larger N_{out}.

Table 3.3: MPAM-derived architectural parameters for commercial systems

Platform	**MPAM Derived**		**Published**		
	$\boldsymbol{MSHR_{l1}}$	$\boldsymbol{MSHR_{l2}}$	$\boldsymbol{R_{phy}}$	$\boldsymbol{MSHR_{l1}}$	$\boldsymbol{MSHR_{l2}}$
Intel Xeon Platinum 8848C	16	44	512	16	48
AWS Graviton 3	27	70	256	-	-

CHAPTER 4

THE UPDOWN SYSTEM ARCHITECTURE

In this chapter, we first give an overview of the whole UpDown design [34], which is part of a larger system design project funded as part of IARPA's Advanced Graphic Intelligence Logical Computing Environment (AGILE) program [17]. Then we highlight three key mechanisms that allow UpDown to achieve high memory level parallelism: split transaction DRAM memory requests, explicit compute-name synchronization of memory responses, and efficient memory parallelism scaling. We further evaluate UpDown's memory parallelism using MPAM, showing that UpDown can achieve unlimited memory level parallelism. The only limit in implementation is the issue rate and the memory module UpDown attached.

4.1 UpDown Design

The UpDown architecture does not include a deep cache hierarchy typical in traditional CPUs. Memory requests from UpDown can be sent to the DRAM directly. Multiple UpDown accelerators can be integrated per HBM stack/DRAM system, providing hardware parallelism. A simple controller core is used to schedule and offload work on UpDown.

Each UpDown accelerator contains 64 UpDown lanes. Each UpDown lane is an event-driven programmable accelerator that generates unlimited memory parallelism under software control. Each lane also has a software-managed 64 KB scratchpad. Events are first-class primitives in the UpDown ISA. It combines traditional hardware events like memory return, write acks, and custom software events with a unified programmable event framework. The incoming events will be first stored in an event queue. Each event will create/activate a thread context on the UpDown lane. It will pull the register state from physical registers and the data payload from the operand buffer according to the event and the thread. The event will end and the lane will switch to the next event in the event queue when a `yield`

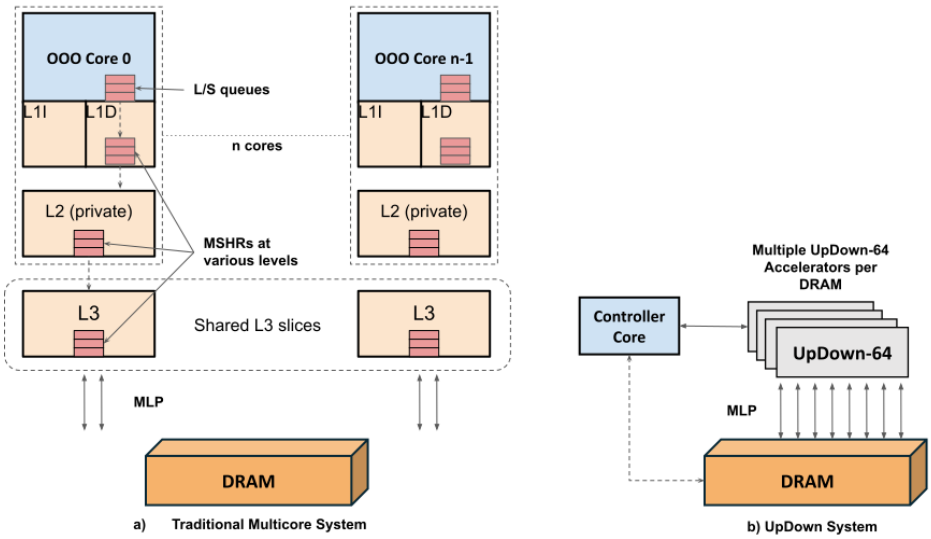


Figure 4.1: Varying memory access paths between a) Traditional Multicore CPU and b) UpDown System

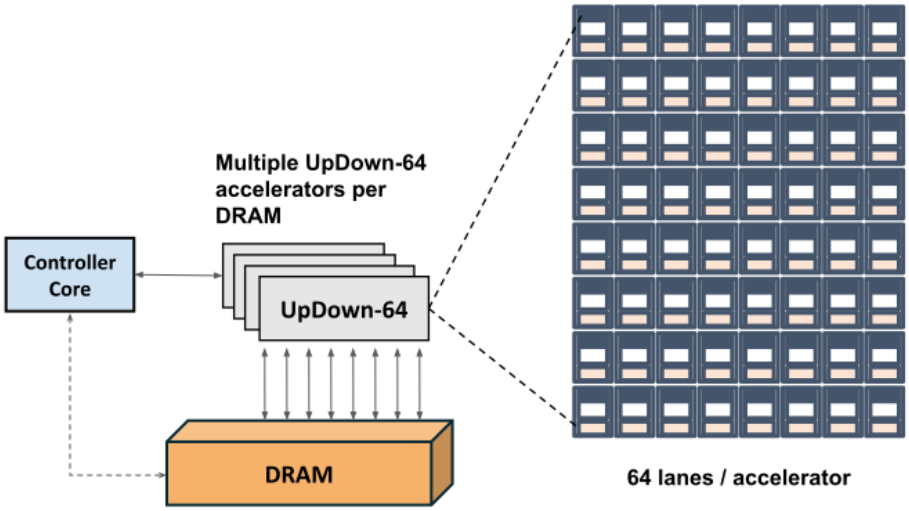
or `yieldt` instruction is executed. More details are described in Section 4.2.2. A detailed discussion of the UpDown instruction set architecture and event mechanisms can be found in the UpDown ISA [34]. Here we highlight some of the key mechanisms related to memory level parallelism in the next section.

4.2 Key Mechanisms for Memory Level Parallelism in UpDown

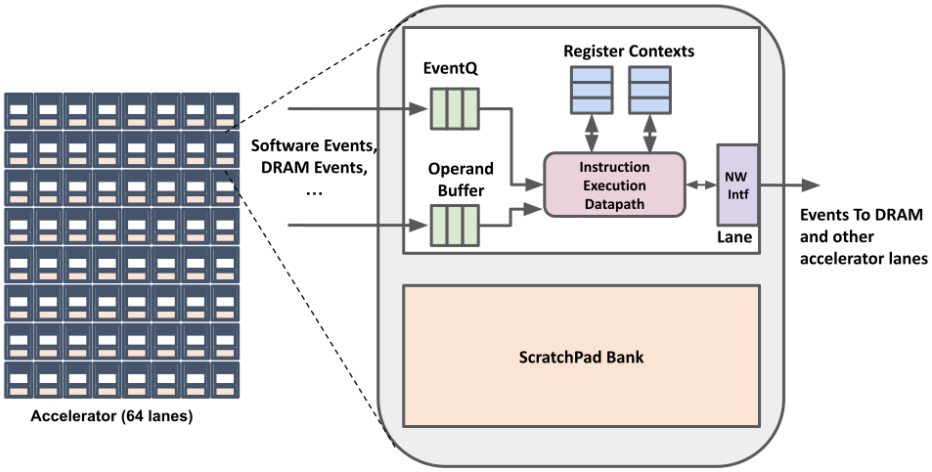
UpDown enables unlimited memory level parallelism through three key mechanisms: split transaction DRAM memory requests, explicit compute-name synchronization of memory responses, and efficient memory parallelism scaling. These mechanisms build on a number of previous research machine designs, like [28, 38, 42, 43, 72, 100], and are described in detail below.

4.2.1 Split Transaction DRAM Memory Requests

The UpDown ISA includes three different `send` instructions that allow each lane to form a message for split transaction DRAM access from the registers, scratchpad, or operand buffer.



(a) Multiple UpDown lanes are integrated into the memory path controlled by a simple controller core that offloads work on UpDown lanes



(b) UpDown Lane Architecture

Figure 4.2: The UpDown node and system

The `send` instructions are asynchronous, meaning lanes can continue executing instructions with the same thread after finishing the `send`.

Each message contains a continuation word, which enables the memory request's return to be customized. The continuation word contains the information for memory responses like destination lane, thread, and event, details described in the next section 4.2.3. Using continuation word creation instructions (`evs`), UpDown no longer needs to bookkeep the sent

Table 4.1: Instructions in UpDown

Instruction(s)	# Cycles	Description
`sendm`	2	Asynchronous Memory access instruction using buffer in scratchpad
`sendmops`	1	Asynchronous Memory access instruction using operand buffer
`sendmr`	1	Asynchronous Memory access instructions using register values
`ev, evi, evr`	1	Continuation creation instructions
`yield, yieldt`	1	Thread Management Instructions

message or wait for its return.

Each thread can launch multiple *send*s in one single event before it yields the lane (`yield`) or terminates itself (`yieldt`), details are described in the section 4.2.3.

With these instructions, all memory accesses in UpDown are software-controlled. Traditional cache-based systems rely on hardwired cache policies (replacement, coherence, and consistency) and prefetching, which is ineffective for applications that have low data reuse or complex data access patterns. UpDown can efficiently manage the local data between registers, scratchpad, and DRAM with software.

4.2.2 Explicit Compute-Name Synchronization of Memory Responses

Traditional CPU architecture requires expensive bookkeeping for outstanding memory requests so it can do synchronization when memory responds. This bookkeeping usually limits the outstanding request namespace (N_{out}) to a narrow local namespace (e.g. MSHR entry IDs). In contrast, the event-driven execution UpDown uses explicit compute-name synchronization. The compute name in UpDown contains the following information:

1. networkID: The destination UpDown lane of the memory response. The response event should be sent to that lane and executed there.

2. numOperands: The number of words that this event will occupy in the operand buffer.

3. threadID: The destination thread of the memory response. It corresponds to a thread that already exists in the networkID lane or it will be `0xFF`, creating a new thread on that lane.

4. eventLabel: The target instruction offset for the thread invocation.

As shown in the following figure, such a compute name allows powerful thread invocation and synchronization possibilities. In the figure, arrows with circled numbers represent how each part of the compute name (encoded event word) contributes to the synchronization.

① The memory response will be pushed to the event queue that is specified by the networkID.

② A data payload of size numOperands will be pushed to the operandBuffer. When the event pops from the event queue, the corresponding number of operands will also be fetched from the operand buffer.

③ The threadID makes sure that the correct register file is brought to the data path.

④ The eventLabel indicates the offset of the next instructions' address from the base instruction address. The UpDown lane will execute from that instruction till hitting a `yield` or `yieldt` with the event word from eventQueue, operands from the operandBuffer, and register file from physical registers according to the compute name.

Traditional architectures synchronize memory responses with physical registers in ROB, reservation stations, etc, that have a finite synchronization namespace limited by the physical resources. But in many fine-grained parallel programs, there are many unordered memory references, which favors UpDowns compute name synchronization for memory responses that allow an unlimited synchronization namespace. Even in the case when memory request ordering matters the correctness, it is inexpensive to separate memory requests to different events (to ensure memory request ordering in one lane) or send lane-to-lane synchronization messages (to ensure memory request ordering for multiple lanes).

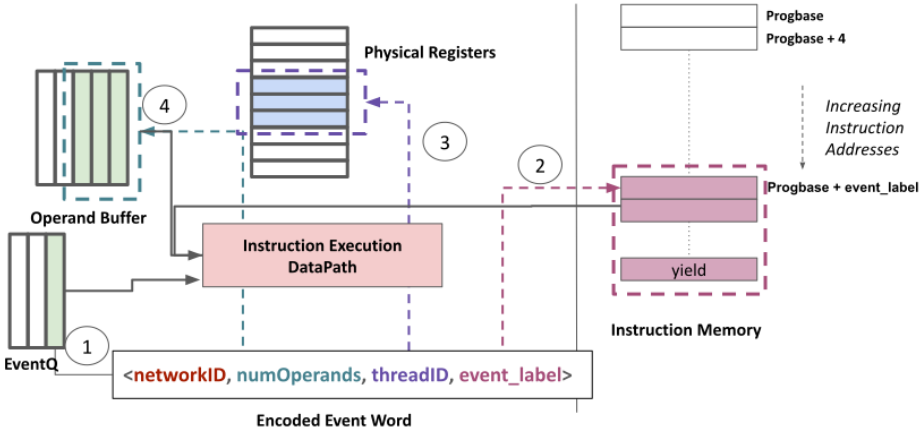


Figure 4.3: Synchronization using Compute-name (continuation-word) in UpDown.

4.2.3 Efficient Memory Parallelism Scaling

UpDown supports multiple light-weight thread contexts on each lane. Each lane can hold up to 128 threads with 16 general-purpose registers (*X16* to *X31*, 64-bit for registers) and 8 special registers (including network ID, continuation word, etc.) per thread. Each thread within one lane is identified by the threadID, which is also included in the encoded event word. As mentioned in the previous section, during execution, the correct register file will be pulled from the physical registers based on the threadID. A special case is when threadID equals `0xFF`, a new thread context will be created and it will be assigned with an unused threadID.

While the split transaction allows a single UpDown thread to launch unlimited memory requests and the explicit compute name synchronization allows a thread to do unlimited synchronization on memory responses, multi-thread contexts provide further potential to scale up memory level parallelism for applications that have strong ordering requirements on memory requests.

Further scale-out parallelism is provided by hardware parallelism. With $\sim 0.06mm^2$ per UpDown lane, an UpDown accelerator with 64 lanes is still a fraction of the area of a large OOO core; and a cluster with multiple accelerators per DRAM is still small in area compared

to a large OOO multi-core CPU. This scale-out parallelism, as shown in the following section 5.3, enables UpDown to achieve memory level parallelism sufficient to saturate an HBM3e stack's bandwidth of 1.2TB/s.

4.3 Memory Level Parallelism Analysis on UpDown using MPAM

Now, let us apply the MPAM model we described in the previous chapter and see why the UpDown architecture can achieve unlimited memory level parallelism.

4.3.1 Synchronization Namespace N_{sync}

Unlike traditional architectures that require allocation of physical resources to synchronize memory responses, UpDown uses compute names represented by event_words as described in 4.2.2. The compute name allows the memory response to synchronize to a specific instruction address (eventLabel), thread (threadID), and compute resource (networkID), with b_{elabel}, b_{tid}, and b_{nwid} bits to represent them respectively. There can be multiple outstanding memory requests (n_{rep}) that synchronize to the same thread on the same compute resource with the same instruction address. Thus the Synchronization Namespace can be represented as

$$N_{sync} = n_{rep} \cdot 2^{b_{elabel}+b_{tid}+b_{nwid}}. \tag{4.1}$$

4.3.2 Outstanding Request Namespace N_{out}

With the mechanism of split transactions for memory requests mentioned in section4.2.1, UpDown doesn't need to do bookkeeping like traditional architecture. Thus the synchronous request namespace for UpDown can be the entire address space multiplied by an arbitrary

number of requests (n_{rep}) needed on the same address

$$N_{out\_sync} = n_{rep} \cdot 2^{b_{mem}} \tag{4.2}$$

where b_{mem} is the number of bits used to represent the address space. In UpDown, all outstanding memory requests are generated by software, and there is no hardware prefetching. Thus

$$N_{out\_async} = 0. \tag{4.3}$$

4.3.3 Shared Request Namespace N_{sh}

UpDown does not include a shared namespace in the path between each UpDown lane to memory.

4.3.4 MPAM on UpDown

By applying Equation4.1, Equation4.2, and Equation4.3 to Equation3.1, we get

$$MLP = L \cdot n_{rep} \cdot \frac{s_{op}}{cl} \cdot min(2^{b_{elabel}+b_{tid}+b_{nwid}}, 2^{b_{mem}}) \tag{4.4}$$

where L is number of UpDown lanes, n_{rep} is the average number of repetitions of memory requests, s_{op} is the average requirement size of memory requests. The formula shows that the memory level parallelism is unlimited. In a real system, the number of outstanding memory requests will be limited by the size of the queue in the DRAM chip and also the issue rate of updown lanes.

CHAPTER 5

EVALUATION

5.1 Methodology

5.1.1 Workload and dataset

To assess the performance of UpDown and other architectures, we employ a series of memory-intensive workloads. In addition to STREAM [68], we incorporate Dot Product, Top-k, and Histogram, all of which exhibit substantial memory traffic and increasing merging costs. FFT serves as a workload representative of high memory access and heavy synchronization requirements. Furthermore, we include machine learning-related workloads such as SpMV and CNN.

Table 5.1: Workloads and Datasets

Workloads	**Dataset**
STREAM microbenchmark [68] STREAM-Copy (ST-C) STREAM-Scale (ST-S) STREAM-Add (ST-A) STREAM-Triad (ST-T)	Three arrays with 8,388,608 64-bit floating-point entries
Top-k (TK)	One array with 8,388,608 128-bit (64-bit unsigned integer, 64-bit floating point) key-value pairs; $k = 8$
Dot Product (DP)	Two arrays with 8,388,608 64-bit floating-point entries each
Image Processing (HIST)	One array with 16,777,216 64-bit unsigned integer entries
Sparse Matrix Vector Product (SpMV)	Square matrix of dimension 12,288 and 4% uniformly distributed nonzeros in CSR format (64-bit unsigned integer, 64-bit floating point)
Fast Fourier Transform (FFT)	One array with 262,144 128-bit (64-bit floating point, 64-bit floating point) complex numbers
Image Filtering (CNN)	One 3074×3074 2-D array with 9,449,476 64-bit floating points; One 3×3 convolution filter with nine 64-bit floating points

5.1.2 Simulator Configuration

We use gem5 [67] and DRAMSim3 [65] for cycle-level simulation. For DRAM, we use a configuration for HBM2e (1 stack, 8x channels) with 460 GB/s memory bandwidth and 100 ns latency in the comparison part and HBM3e (1 stack, 8x channels) with 1200 GB/s memory bandwidth in the scalability part.

5.1.3 Component Configuration

We use gem5 [67] integrated MinorCPU and O3CPU as our model for In-Order core and Out-of-Order (OOO) core setup. We tuned the OOO core's Load Store Queue and MSHR setting to match the STREAM performance of the Sapphire Rapids GoldenCove core we tested. For UpDown, we integrate a cycle-accurate model into the gem5 infrastructure. Each 64 Updown lanes form an accelerator that shares a 4MB scratchpad. All In-Order Cores, OOO Cores, and UpDown lanes are set to 2 GHz.

Table 5.2: Component and Specification

Component	**Specification**
In-Order Core	Simple InOrder core (gem5 MinorCPU) @2GHz
OOO Core	x86 OOO core (GoldenCove configuration using gem5 O3CPU) @2GHz 280 Physical Registers, 512 ROB entries 192 LoadBuff entries, 114 StoreBuff entries
Caches	L1: 32KB ICache, 64KB DCache per core (16 MSHR entries, 8 slots per entry), L2: 1MB unified per core (private) (48 MSHR entries, 8 slots per entry) L3: 64MB in total (shared) (512 MSHR entries, 8 slots per entry)
UpDown Accelerator	64 UpDown Lanes + 4MB scratchpad (64KB per lane) @2GHz

5.1.4 System Configuration

Using these components, we configure single In-Order and OOO systems, 32-core OOO systems, and UpDown systems with 1 and 256 lanes. The prefetcher is located in L2 and is enabled in setups with "+PF" [51]. For these UpDown systems, although a CPU is listed in the description, it is not used in any of the simulation.

Table 5.3: System Configuration and Description

System Configuration	**Description**
IN	In-Order Load/Store Core,No Cache,Word-Sized Memory Accesses
OOO	OOO Load/Store Core,No Cache, Word-Sized Memory Accesses
OOO-MSHR	OOO + L1/L2 Caches, Cache-Block sized Memory Accesses
OOO-MSHR+PF	OOO-MSHR + Prefetch enabled
OOO-MSHR+PF-32	32 OOO-MSHR + PF cores, private L1/L2 caches, shared L3 cache
UpDown-1	UpDown lane - CPU (Core+caches) + 1 UpDown Lane
UpDown-256	Scale-out UpDown - CPU (Core+caches) + 256 UpDown Lanes

5.1.5 Performance Matrix

The execution time for each experiment is measured according to the perflog of the gem5 simulation. We also calculate the memory traffic for each experiment based on the UpDown implementation. Despite the potential reduction in memory traffic caused by cache hits, we disregard this difference because the majority of the workloads we select have relatively low cache rates. We then compute the memory bandwidth and memory parallelism base on the time and traffic.

5.2 Performance

First, we present the comparison between a single UpDown lane relative to one In-Order core, showcasing the capability and efficiency of UpDown to achieve high memory level parallelism with a small area. Then we compare UpDown with OOO systems, showing how UpDown can

Table 5.4: Metric and Definition

Metric	**Definition**
Runtime (seconds)	Execution time
SpeedUp	Relative Performance
Memory Bandwidth (GB/s)	Memory Traffic (GB) / Runtime (s)
Memory Parallelism	No of outstanding memory requests (Memory Bandwidth (GB/s) / access_size (B) * latency (ns))

achieve extremely high memory parallelism. In both sets of experiments, we demonstrate that both performance speedup and memory bandwidth are achieved. We further verify that UpDown can be scaled up to HBM3e bandwidth.

Here, we estimate the die area of the In-Order core and Out-of-Order core with [54, 70]. For UpDown under 7 nm, UpDown-1 is $\sim$0.06 mm^2 [82], comparable in area to a single In-Order core, and UpDown-256 ($\sim$16 mm^2 [82]) is comparable to a single Out-of-Order core with its cache hierarchy ($\sim$13 mm^2 for Sapphire Rapids [66]).

5.2.1 Performance relative to In-Order core

We first present the overall runtime speedup over an In-Order Core and the corresponding memory bandwidths in the following figures.

As a result, UpDown-1 achieves 65x geomean speedup, and 52x times memory level parallelism over a single In-Order core with a comparable die area. On STREAM copy, one lane of UpDown can have more than 100x speedup and memory level parallelism.

5.2.2 Performance relative to Out-of-Order cores

Next, we compare the performance of UpDown-1 and UpDown-256 to OOO, OOO-MSHR, and OOO-MSHR+PF configurations.

UpDown-1 achieves a 2.1x geomean speedup over OOO and a 1.2x geomean speedup over OOO-MSHR+PF with a much smaller die size. With hardware parallelism, UpDown-

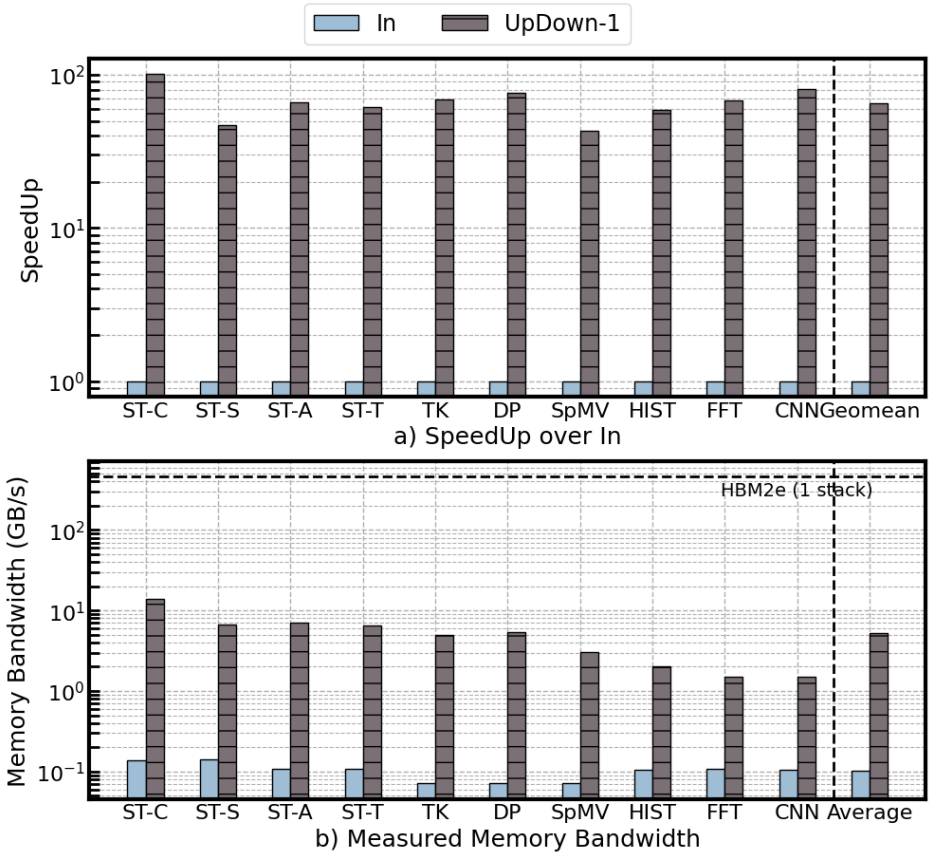


Figure 5.1: a) SpeedUp of UpDown-1 over In, b) Measured Memory BW of In and UpDown-1

256 achieves a 163x geomean speedup over OOO and a 91x over OOO-MSHR + PF with comparable die size. On applications like TK and DP, UpDown-256 nearly saturates the HBM2e bandwidth at $\sim$ 428 GB/s. On average, UpDown-256 achieves 325 GB/s across all applications.

It is worth noting that applications with low data reuse, like STREAM and TK, have low or even negative benefits from adding cache or enabling prefetch (OOO $\rightarrow$ OOO-MSHR $\rightarrow$ OOO-MSHR+PF). In contrast, UpDown's memory access mechanisms enable even a single lane UpDown-1 to achieve performance comparable to OOO and OOO-MSHR+PF across most applications, irrespective of the level of data reuse.

The measured memory bandwidth figure confirms the performance benefits being directly related to the higher bandwidths achieved in these applications. Even on applications like

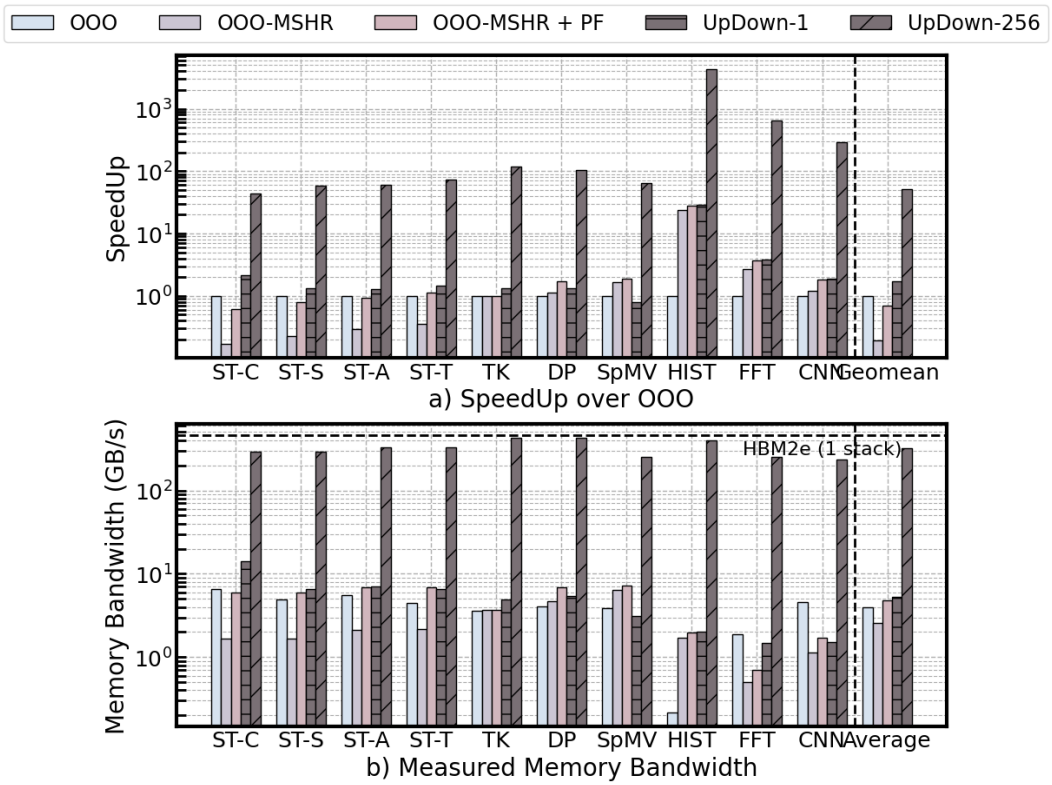


Figure 5.2: a) SpeedUp of UpDown-1 and UpDown-256 over OOO, OOO-MSHR, OOO-MSHR+PF, b) Measured Memory BW on OOO, OOO-MSHR, OOO-MSHR+PF, UpDown-1 and UpDown-256

FFT and CNN that have high data reuse and benefit from caching, UpDown-256 achieves higher performance than OOO systems. This suggests that it is feasible to achieve higher performance by utilizing high memory bandwidths more effectively, like UpDown, than traditionally adapting hardwired caching mechanisms to reduce latency and memory traffic.

5.2.3 Correlation between Performance and Memory Level Parallelism

We then compare the performance of one In-Order core, one and 32 Out-of-Order cores, and one and 256 UpDown lanes. We use the memory level parallelism and performance as two axises. We further partition the graph into four vertical regions based on the *MLP* we modeled for one In-Order core (MLP_{IN}), one ($MLP_{OOO\text{-}MSHR+PF}$) and 32 Out-of-Order cores ($MLP_{OOO\text{-}MSHR+PF\text{-}32}$) using MPAM.

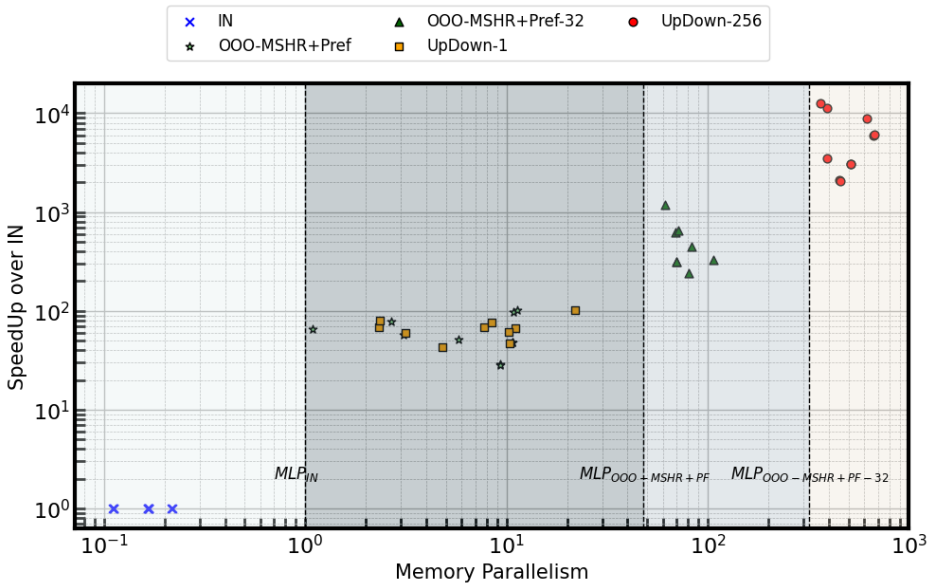


Figure 5.3: SpeedUp vs Memory Parallelism on In, OOO-MSHR+Pref, OOO-MSHR+Pref-32, UpDown-1 and UpDown-256

All results lie close to the diagonal of the graph, suggesting a strong correlation between the application performance and the memory level parallelism. Also, CPU results all stay in their corresponding band, suggesting MPAM is correctly identifying *MLP* limits.

The experiment results form four groups. From left to right, the first group shows that one In-Order core has a memory level parallelism of less than one. In the second band, we have results for one UpDown lane and one Out-of-Order core. We can see that one UpDown lane achieves up to 3.5x memory level parallelism compared with one Out-of-Order core, using about 207x smaller area. In the third band, we have data from 32 OOO cores. Compared with one OOO core, it is 31x faster in high data-reuse applications like CNN. But it has less than 20x speed up in other applications. The performance uplift here is limited by the memory parallelism limits of the shared namespace. In the fourth band, we have a 256 UpDown lanes system with the best performance. It achieves 26x$\sim$79x speedup compared with one Out-of-Order core using a comparable area. In all, our UpDown system achieves high memory level parallelism and high performance with low chip area.

5.3 Scalability

Noticing that UpDown-256 hits the HBM2e bandwidth in previous experiments, we next analyze memory level parallelism scalability with an increasing number of cores, considering HBM3e as a performance target. We evaluate the impact of UpDown's memory access mechanisms using a simple read-only microbenchmark that does a scan read of a contiguous block of memory in DRAM.

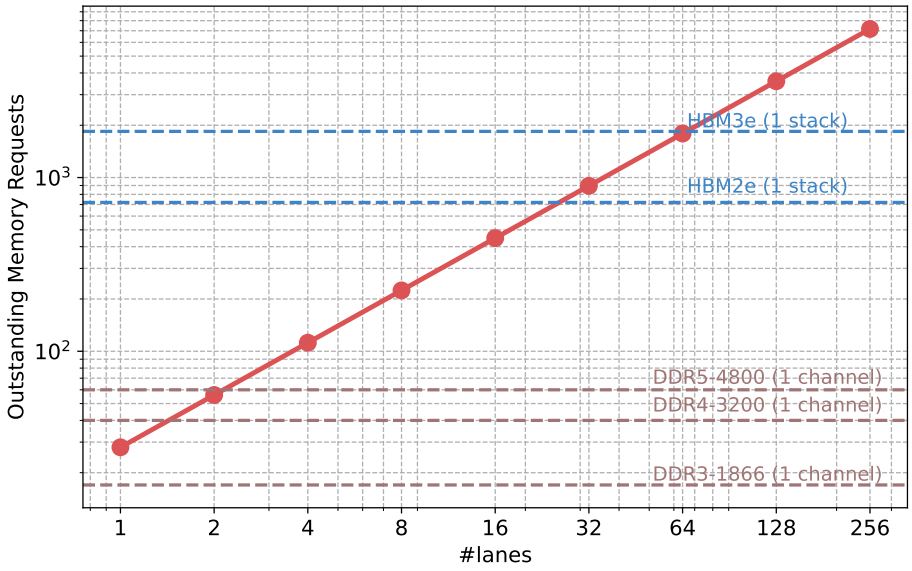


Figure 5.4: Memory Parallelism with scan read microbenchmark running on multiple UpDown lanes. Horizontal lines show memory parallelism required for various memory systems assuming 100ns latency

As can be seen, 1 lane can saturate a single DDR3-1866 channel (14.9 GB/s), 25 lanes can saturate an HBM2e stack (460 GB/s) and 64 lanes (1 UpDown accelerator) can saturate an HBM3e stack (1.2 TB/s with 1,673 requests). In contrast, a Sapphire Rapids system with 56 cores achieves about 240 GB/s or 590 GB/s with a DDR5-4800 or HBM2e memory system [69].

To further show UpDown's great scalability with the increasing number of lanes, we connect test both multi-core OOO-MSHR+PF and multi-lane UpDown with HBM2e and

HBM3e and benchmark STREAM copy.

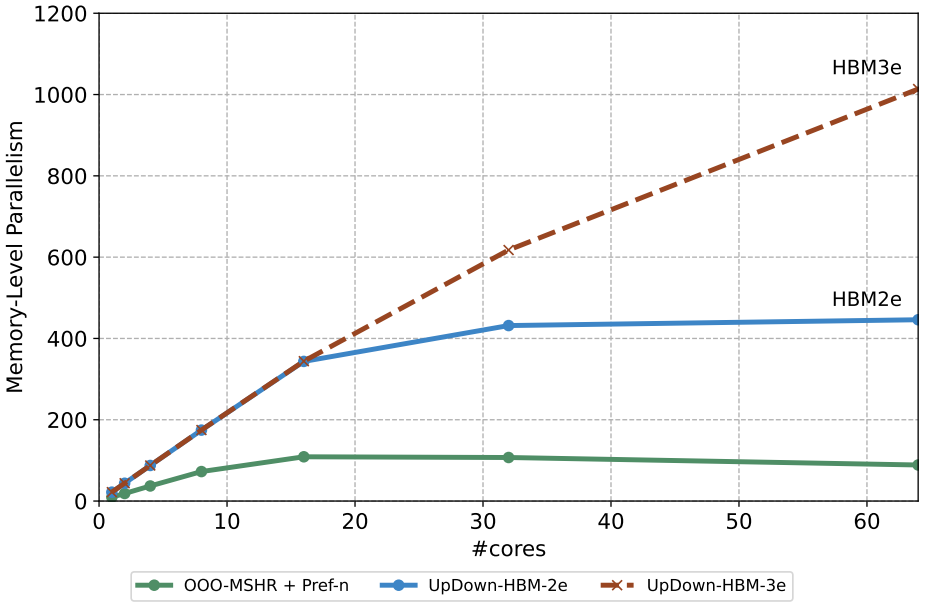


Figure 5.5: Memory Parallelism vs #cores on ST-C

For multi-core OOO-MSHR+PF, the memory level parallelism quickly stops scaling beyond 20 cores due to the namespace limits in the shared $MSHR_{l3}$. UpDown, on the other hand, can saturate the memory level parallelism a HBM2e stack can provide with 32 lanes. With HBM3e, 64 UpDown lanes can achieve >1000 outstanding requests on STREAM copy. In both cases, unlike the scan-read benchmark in Figure 13, STREAM is limited by the switching between reads and writes and the write bandwidth limits. It is less than the result of the read-only microbenchmark, as STREAM is limited by the switching between reads and writes and the write bandwidth limits.

CHAPTER 6

COST OF MEMORY LEVEL PARALLELISM

6.1 Cost of Area

We estimate the size of a single Out-of-Order core, including its cache hierarchy, based on the die shot of a Golden Cove server tile provided in [66]. As presented in Table 6.1, a significant portion of the chip's area is dedicated to the cache hierarchy. Further digging into the die shot, we notice that nearly half of the CPU core area is spent on Out-of-Order scheduling, retirement, and branch logic.

For UpDown, synthesis using Synopsys tools on 28nm libraries and projecting it to 7nm [29, 98, 99], gives ∼0.06 mm^2 per UpDown lane (UpDown-1) and UpDown-256 ∼16 mm^2 [82].

OOO-MSHR+PF [Intel 7]		**UpDown-1 [7 nm]**		**UpDown-256 [7 nm]**	
Component	**Area mm^2**	**Component**	**Area mm^2**	**Component**	**Area mm^2**
CPU Core	4.34	1 UpDown Lane	0.027	256 UpDown Lanes	6.90
Load & Store + L1D Control + L1D 48 KB	1.03	Scratchpad 64 KB	0.036	Scratchpad 16 MB	9.18
L2 2 MB	2.20				
L3 1.875 MB	1.52				
Whole tile	**13.04**	**Accelerator System**	**0.063**	**Accelerator System**	**16.08**

Table 6.1: Area Comparison between OOO-MSHR+PF (Golden Cove Server [66]), UpDown-1 [82], and UpDown-256 [82]

As illustrated in Figure 5.3, UpDown-1, despite its significantly reduced area by a factor of 207, achieves superior memory level parallelism compared to OOO-MSHR+PF. Furthermore, UpDown-256, with an area comparable to OOO-MSHR+PF, exhibits substantially higher memory level parallelism and performance against an Out-of-Order core. These demonstrats the remarkable area efficiency of the UpDown design.

6.2 Cost of Comparator

As demonstrated previously, for architectures that apply bookkeeping and tracking of in-flight memory requests, the memory level parallelism achievable is constrained by the size of load/store queues, MSHRs across the cache hierarchy, write buffers, and other components. In most cases, Content-Addressable Memories (CAMs) are used in these structures [49], necessitating read and write operations on the structure and rapid matching of incoming entries with all stored entries. When an outgoing memory request approaches a level of the memory hierarchy, an associative check is performed, typically involving the physical memory address, within the bookkeeping structure at the current level. If it is not matched in the current CAM and can be registered in the CAM, the memory request can be passed to the next level, and a similar associative check is repeated.

In this section, we continue to employ the concepts of OOO-MSHR+PF, OOO-MSHR+PF-32, and UpDown from Table 5.3 as configurations for comparison. Furthermore, we include a hypothetical configuration, OOO-largeMSHR+PF-32, an extension of OOO-MSHR+PF-32. This configuration has a larger CAM to match the memory level parallelism that HBM3e memory can provide. Details of these configurations are listed in Table 6.2. For each memory level parallelism, we decrease the size of the Load-Store Queue and MSHRs in each configuration to find the minimum resources needed for that configuration to support that certain memory level parallelism. It is not shown in the figure if the memory level parallelism is beyond one configuration's limit.

6.2.1 Increase of Comparator

A CAM with n entries requires n-way comparators and a $\log_2 n$ depth encoder [49]. By analyzing the size of CAM at each level, we can estimate the number and the depth of comparators in each system. We use *MLP* as the memory level parallelism each system approaches for each equation, and we do not consider prefetching.

For UpDown, since we avoid the bookkeeping of the outstanding memory requests by splitting transactions for memory requests and using explicit compute-name synchronization for memory response, achieving high memory level parallelism does not require the system to have a large number of comparators. But for all multicore systems we study, we have

$$\#\ Comparator = ((CAM_{LSQ} + CAM_{l_1} + CAM_{l_2}) * \#\ cores + CAM_{l_3}) * Address\ Tag\ Length,$$
$$Depth\ of\ Camparator = 2 * (\lceil \log_2 CAM_{LSQ} \rceil + \lceil \log_2 CAM_{l_1} \rceil + \lceil \log_2 CAM_{l_2} \rceil + \lceil \log_2 CAM_{l_3} \rceil + 4).$$

In the OOO-MSHR+PF system, the core will generate MLP outstanding memory requests, and each level needs to bookkeeping MLP requests.

$$CAM_{LSQ} = CAM_{l_1} = CAM_{l_2} = CAM_{l_3} = MLP,$$
$$\#\ Comparator = 208 * MLP,$$
$$Depth\ of\ Camparator = 8 * \lceil \log_2 MLP \rceil + 8$$

And the MLP it can achieve is limited by the size of L1 MSHR, i.e. $MLP \leq 16$.

In the OOO-MSHR+PF-32 system, in the best cases, each core will only need to generate $\frac{MLP}{32}$ outstanding memory requests.

$$CAM_{LSQ} = CAM_{l_1} = CAM_{l_2} = \frac{MLP}{32},$$
$$CAM_{l_3} = MLP,$$
$$\#\ Comparator = 208 * MLP,$$
$$Depth\ of\ Camparator = 8 * \lceil \log_2 MLP \rceil - 22$$

And the MLP it can achieve is limited by the size of L1 MSHR and L3 MSHR, i.e. $MLP \leq 512$.

To fulfill HBM3e memory bandwidth (1.2 TB/s), the OOO-largeMSHR+PF-32 system

needs to achieve $MLP = 1920$ under 100 ns memory request latency. To accomplish this, we increase the size of the Load-Store Queue and the MSHR in each memory hierarchy by a factor of four compared to the OOO-MSHR+PF-32 system. The equations for the number

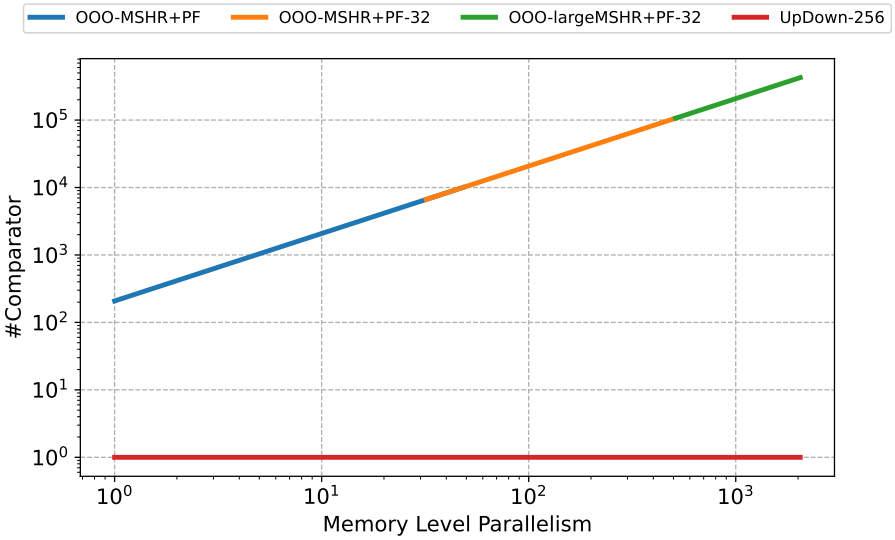
Configuration	OOO-MSHR+PF	OOO-MSHR+PF-32	OOO-largeMSHR+PF-32
# Core	1	32	32
LSQ per Core	114	114	456
L1 MSHR per Core	16	16	64
L2 MSHR per Core	48	48	192
L3 MSHR	512	512	2048

Table 6.2: OOO-largeMSHR+PF-32 Configuration

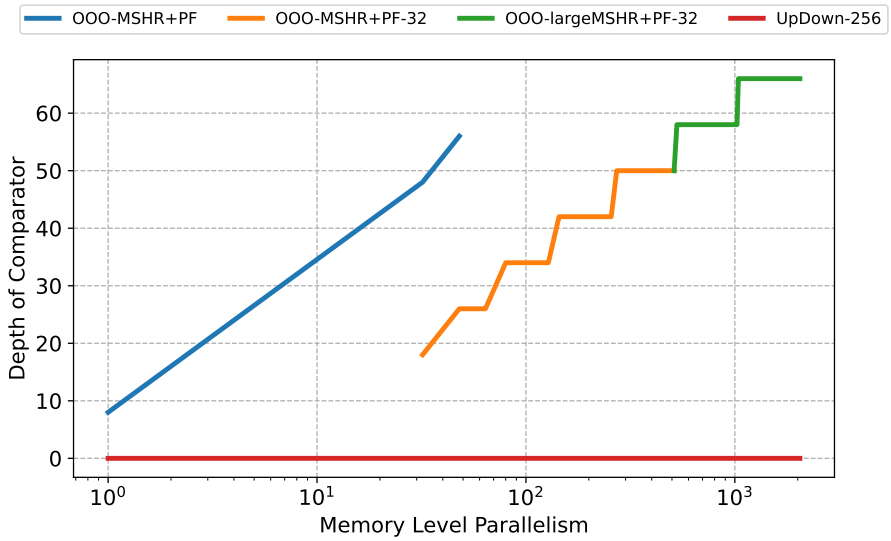
and the depth of comparators are the same as in the OOO-MSHR+PF-32 system.

Figure 6.1 illustrate the increasing number of comparators and the depth of the comparators as the systems' memory level parallelism increases. Since the number of comparators is correlated with the area it takes, we can see that the area cost grows linearly against the memory level parallelism. Meanwhile, the increasing depth of the comparators suggests additional latency for the memory request [49, 102, 117]. Changing from OOO-MSHR+PF-32 to OOO-largeMSHR+PF-32 makes the system capable of saturating the HBM3e's bandwidth but also requires at least 16 additional layers of comparators. These additional layers and extra distances for the signal to travel through the much larger CAM's fan-in and fan-out circuits introduce extra latency at each level in the cache hierarchy.

Current OOO cores cannot saturate a single HBM2e stack. 32 or more OOO-MSHR+PF cores with overhead >10,000 comparators in trees with depth >40 are required to match the requirement for HBM2e bandwidths. We also show a hypothetical OOO core with an extremely large MSHR (OOO_largeMSHR+PF-32) to estimate overhead to achieve HBM3e (1.2 TB/s) bandwidths. As can be seen, >400,000 comparators arranged in depths of > 60 would be required to achieve and sustain HBM3e bandwidths. In contrast, UpDown architecture does not track or bookkeep on the outstanding requests. It implements consistency in software; no dedicated hardware comparators are needed, and increasing memory parallelism



(a) Number of Comparators vs Memory Parallelism



(b) Depth of Comparators vs Memory Parallelism

Figure 6.1: The UpDown node and system

has no additional cost.

6.2.2 Increase of Comparisons Rate

Beyond the quantity of hardware, these comparators need to be run at a high rate; we estimate the giga-comparisons/second required to achieve a given memory bandwidth. Assume

that each core contributes to the memory level parallelism and that the MSHRs in each cache hierarchy level are filled evenly. When the system generates MLP, each memory request requires two activations of CAM in each level (request and respond). The average number of active entries in each level is $\frac{MLP}{\#\ cores}$.

$$
\begin{aligned}
\#\ Comparation\ per\ request_{LSQ} &= \#\ Comparation\ per\ request_{l1} \\
&= \#\ Comparation\ per\ request_{l2} \\
&= 2 * \frac{MLP}{\#\ cores} * Address\ Length \\
\#\ Comparation\ per\ request_{l3} &= 2 * MLP * Address\ Length \\
\#\ Comparation &= MLP * \sum_{level=LSQ,l1,l2,l3} \#\ Comparation\ per\ request_{level} \\
&= \left(104 + \frac{312}{\#cores}\right) * MLP^2
\end{aligned}
$$

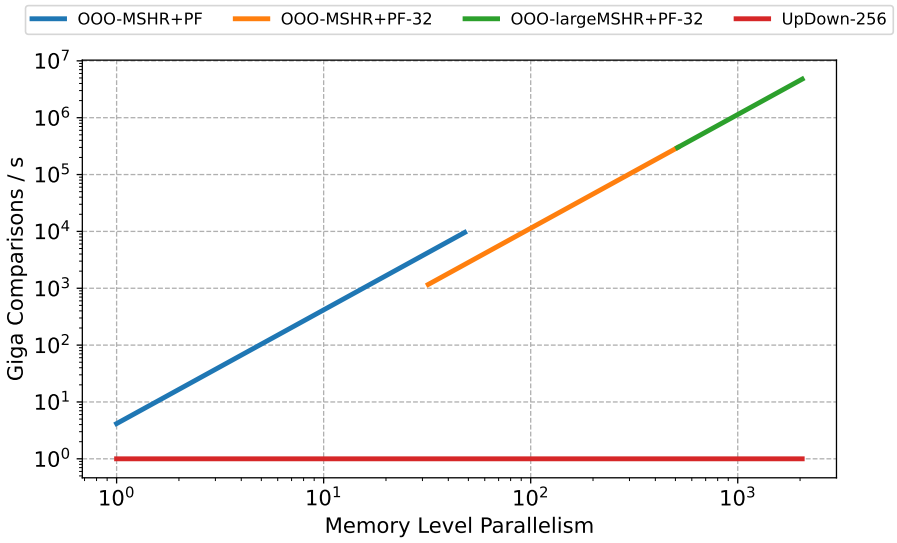


Figure 6.2: GigaComparisons/second vs Memory Parallelism

These giga-comparisons (and associated data movement) reflect the power consump-

tion required to sustain that memory parallelism. The results show that $\sim 6,000,000$ giga-comparisons per second (6 peta-comparisons/second) are required to reach the HBM2e bandwidths on a 32-core system. Again, no such operation rates are needed in UpDown.

CHAPTER 7

RELATED WORK

7.1 Hardware Prefetchers

Hardware prefetching is a mechanism that can generate additional outstanding memory requests using the asynchronous request namespace (N_{out_async}), as the program does not control the prefetch hardware. The effectiveness of prefetchers depends on the accuracy of the hardware prediction of access patterns. Although some custom prefetchers are proposed for specific irregular applications, such as [23, 71, 85, 115, 116], the lack of uniform or predictable data access patterns narrows the benefits that prefetchers can bring. In certain instances, hardware prefetching can even adversely affect system performance, as evidenced by the research presented in [61]. Hardware prefetching utilizes the same outstanding request namespace (N_{out}) as other synchronous memory requests. They compete with resources such as the MSHR at lower levels of the memory hierarchy, memory bandwidth, and cache storage. In some situations, hardware prefetching can have significant negative impacts when combined with software prefetching, as evidenced by [62]. Conversely, UpDown can almost fully utilize any given memory system by selectively fetching only useful data. Critically, UpDown enables applications to directly control prefetching with split transactions and scratchpad storage, exploiting application knowledge.

7.2 Multi-Threaded Architectures

Hardware multithreading has been proposed and adapted to different architectures for a couple of decades [22, 24, 25, 45]. The most recent commercial CPUs and GPUs employ hardware multithreading to hide memory latency and improve their instructions per cycle performance. For Intel CPUs with Hyper-Threading, the CPU exposes two execution contexts per physical core. Typically, two threads have logically partitioned but physically

shared components, such as the Resource Ownership Block (ROB), physical registers, and Lazy Set Query (LSQ), as outlined in [55]. This approach can improve the utilization of the synchronization namespace (N_{sync}) for applications that are not memory-intensive. However, the size of N_{sync} remains unchanged, which implies that it does not improve *MLP* for memory-intensive scenarios. For NVIDIA GPUs, they manage 32 threads as a warp. When accessing long-latency memory, the GPU can switch between warps. Additionally, the size of N_{sync} is increased by enabling shared local memory (at most 228KB per 128 cores [19]) to serve as a synchronization space. However, shared memory is expensive (sharing with L1 cache) [18]. Similarly, UpDown also supports lightweight threads. In contrast to SIMD architecture, which necessitates thread switching after each outstanding memory request is generated, UpDown's thread switching is software-controlled. Furthermore, software also controls the number of outstanding memory requests initiated by each thread. This enables high memory level parallelism per thread at a low cost. Additionally, UpDown achieves cheap but unlimited N_{sync} with explicit compute-name synchronization.

7.3 Decoupled Access Execute Architectures

Decoupled Access Execute Architectures separate access and execute programs and run them on different cores [92, 78, 107]. These architectures avoid competition from the execution of other instructions by N_{sync}. However, they require sophisticated compilers to achieve optimal performance on the decoupled version of the code. Recent proposals like [107] suggested using OOO cores to replace In-Order cores in [92]. This requires a complex design for mis-prediction, mis-speculation, and rollback. There are also approaches to replace some of the CAM structure in traditional architectures with program-managed data storage [107] or FIFO queues [78] to improve scalability. However, scaling up to memory level parallelism that can saturate HBM2e/3e will necessitate substantial effort in allocating and managing larger queues or more instances. Meanwhile, UpDown prevents memory access and program

execution from blocking each other by splitting transactions of memory requests. This enables applications to generate outstanding memory requests with few instructions and ensures software executes after sending memory requests. In this manner, UpDown decouples the access and execution of programs without separating them into distinct namespaces. Together with explicit compute-name synchronization, UpDown enhances programmability and avoids synchronization costs between access and execute cores.

CHAPTER 8

SUMMARY AND FUTURE WORK

8.1 Summary

We employ the Memory Parallelism Abstract Machine (MPAM) from [83] as a key matrix to characterize multiple local namespaces as limits to memory-level parallelism and identify the limits on key architectures. We applied MPAM to four commercial CPUs to characterize their design and performance. We further apply MPAM to UpDown [34], an architecture claimed to achieve unlimited memory level parallelism. UpDown avoids local namespace limitations in traditional architectures with split transaction DRAM memory requests, explicit compute-name synchronization of memory responses, and efficient memory parallelism scaling. As we test it with multiple kernels, a single UpDown lane achieves 3.5x memory level parallelism compared to a sophisticated OOO core. We also showed that UpDown can scale up and scale out cost-effectively to saturate an HBM3e stack's memory bandwidth.

8.2 Future Work

As demonstrated in this work, UpDown achieves high memory level parallelism under various testing kernels. However, our testing was limited to a single HBM stack and up to 256 UpDown lanes. It would be interesting to explore the performance of UpDown on a larger machine setup. With multiple memory stacks, load balancing for memory access may necessitate further investigation of algorithms and applications. Additionally, with multiple UpDown nodes, the inter-node latency needs to be carefully considered, as far memory accesses have more than ten times the latency compared with local accesses. This requires applications to consider allocation for better performance. Furthermore, it requests applications to generate more outstanding memory requests to hide the longer latency.

In the UpDown architecture, the compute portion (accelerators) and the memory portion (stacks of HBMs) are interconnected via switches. This enables the decoupling of computation and memory components. Different types of memory provide different characteristics [40]. For example, LPDDRs typically offer higher capacity per stack than HBMs. Although the majority of kernels applied in this thesis are memory-bound, some applications we implemented on UpDown are compute-bound or latency-bound, for example, matrix multiplication and exact match. It will be interesting to study different combinations of memory types and various numbers of UpDown lanes that will provide a substantial performance-cost-energy trade-off space to explore. Different types of applications, such as compute-demanding, memory bandwidth-demanding, or memory space-demanding, will have diverse preferences for various design combinations. Our next step will be to investigate this software-hardware co-design problem in greater detail.

REFERENCES

[1] Amd epyc 7601 specifications. https://www.techpowerup.com/cpu-specs/epyc-7601.c1920. Accessed: 2025-02-08.

[2] Amd epyc 7f72 specifications. https://www.techpowerup.com/cpu-specs/epyc-7f72.c2302. Accessed: 2025-02-08.

[3] Amd epyc 7h12 specifications. https://www.techpowerup.com/cpu-specs/epyc-7h12.c2244. Accessed: 2025-02-08.

[4] Amd epyc™ 7763. https://www.amd.com/en/products/processors/server/epyc/7003-series/amd-epyc-7763.html. Accessed: 2025-02-08.

[5] Amd epyc™ 9654. https://www.amd.com/en/products/processors/server/epyc/4th-generation-9004-and-8004-series/amd-epyc-9654.html. Accessed: 2025-02-08.

[6] Amd epyc™ 9754. https://www.amd.com/en/products/processors/server/epyc/4th-generation-9004-and-8004-series/amd-epyc-9754.html. Accessed: 2025-02-08.

[7] Amd epyc™ 9965. https://www.amd.com/en/products/processors/server/epyc/9005-series/amd-epyc-9965.html. Accessed: 2025-02-08.

[8] Intel xeon e5-2650 v4 specifications. https://www.techpowerup.com/cpu-specs/xeon-e5-2650-v4.c3791. Accessed: 2025-02-08.

[9] Intel xeon e5-2697 v2 specifications. https://www.techpowerup.com/cpu-specs/xeon-e5-2697-v2.c1671. Accessed: 2025-02-08.

[10] Intel xeon e5-2699 v3 specifications. https://www.techpowerup.com/cpu-specs/xeon-e5-2699-v3.c2904. Accessed: 2025-02-08.

[11] Intel xeon e5-4650 specifications. https://www.techpowerup.com/cpu-specs/xeon-e5-4650.c1463. Accessed: 2025-02-08.

[12] Intel xeon e7-8890 v3 specifications. https://www.techpowerup.com/cpu-specs/xeon-e7-8890-v3.c1819. Accessed: 2025-02-08.

[13] Intel® xeon® platinum 8280 processor (38.5m cache, 2.70 ghz) specifications. https://www.intel.com/content/www/us/en/products/sku/192478/intel-xeon-platinum-8280-processor-38-5m-cache-2-70-ghz/specifications.html. Accessed: 2025-02-08.

[14] Intel® xeon® processor e7-8870 (30m cache, 2.40 ghz, 6.40 gt/s intel® qpi) specifications. https://www.intel.com/content/www/us/en/products/sku/53580/intel-xeon-processor-e78870-30m-cache-2-40-ghz-6-40-gts-intel-qpi/specifications.html. Accessed: 2025-02-08.

[15] Intel® xeon® processor w5590 (8m cache, 3.33 ghz, 6.40 gt/s intel® qpi) specifications. https://www.intel.com/content/www/us/en/products/sku/41643/intel-xeon-processor-w5590-8m-cache-3-33-ghz-6-40-gts-intel-qpi/specifications.html. Accessed: 2025-02-08.

[16] Intel® xeon® processor x7560 (24m cache, 2.26 ghz, 6.40 gt/s intel® qpi) specifications. https://www.intel.com/content/www/us/en/products/sku/46499/intel-xeon-processor-x7560-24m-cache-2-26-ghz-6-40-gts-intel-qpi/specifications.html. Accessed: 2025-02-08.

[17] Agile: Advanced graphic intelligence logical computing environment program, January 2022. https://www.iarpa.gov/research-programs/agile.

[18] Nvidia h100 tensor core gpu architecture. *NVIDIA*, 2023.

[19] Blackwell tuning guide v12.8. *NVIDIA*, 2024.

[20] Intel® 64 and ia-32 architectures optimization reference manual volume 1. *Intel*, 2024.

[21] Nvidia blackwell - the engine of the new industrial revolution. *NVIDIA*, 2024.

[22] Sriram Aananthakrishnan, Nesreen K. Ahmed, Vincent Cave, Marcelo Cintra, Yigit Demir, Kristof Du Bois, Stijn Eyerman, Joshua B. Fryman, Ivan Ganev, Wim Heirman, Hans-Christian Hoppe, Jason Howard, Ibrahim Hur, MidhunChandra Kodiyath, Samkit Jain, Daniel S. Klowden, Marek M. Landowski, Laurent Montigny, Ankit More, Przemyslaw Ossowski, Robert Pawlowski, Nick Pepperling, Fabrizio Petrini, Mariusz Sikora, Balasubramanian Seshasayee, Shaden Smith, Sebastian Szkoda, Sanjaya Tayal, Jesmin Jahan Tithi, Yves Vandriessche, and Izajasz P. Wrosz. PIUMA: Programmable Integrated Unified Memory Architecture, October 2020. arXiv:2010.06277 [cs].

[23] Sam Ainsworth and Timothy M. Jones. Graph prefetching using data structure knowledge. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, New York, NY, USA, 2016. Association for Computing Machinery.

[24] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The tera computer system. In *Proceedings of the 4th International Conference on Supercomputing*, ICS '90, page 1–6, New York, NY, USA, 1990. Association for Computing Machinery.

[25] Wendell Anderson, Robert Rosenberg, and Marco Lanzagorta. Experiences using the cray multi-threaded architecture (mta-2). In *Proceedings of the 2003 DoD User Group Conference*, DOD_UGC '03, page 378, USA, 2003. IEEE Computer Society.

[26] Ravi Bhargava and Kai Troester. Amd next-generation "zen 4" core and 4th gen amd epyc server cpus. *IEEE Micro*, 44(3):8–17, 2024.

[27] Gavin Bonshor. The amd zen 5 microarchitecture: Powering ryzen ai 300 series for mobile and ryzen 9000 for desktop. https://www.anandtech.com/show/21469/amd-details-ryzen-ai-300-series-for-mobile-strix-point-with-rdna-35-igpu-xdna-2-npu, 2024. Accessed: 2025-02-08.

[28] Shekhar Borkar, Robert Cohn, George Cox, Thomas Gross, H. T. Kung, Monica Lam, Margie Levine, Brian Moore, Wire Moore, Craig Peterson, Jim Susman, Jim Sutton, John Urbanski, and Jon Webb. Supporting systolic and memory communication in iwarp. *SIGARCH Comput. Archit. News*, 18(2SI):70–81, may 1990.

[29] David Brooks. What's the future of technology scaling?, 2018.

[30] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609–623, 1995.

[31] George Cozma Chester Lam. Amd's zen 4, part 2: Memory subsystem and conclusion. *Chips and Cheese*, 2022.

[32] Daniel Chiang. Sk hynix to reveal 16-layer hbm3e to main the lead. *DigiTimesAsia*, 2024.

[33] Marco Chiappetta. Amd zen 2 architecture explored: What makes ryzen 3000 so powerful. https://hothardware.com/reviews/amd-zen-2-architecture-explained, 2019. Accessed: 2025-02-08.

[34] Andrew Chien, Andronicus Rajasukumar, Marziyeh Nourian, Yuqing Wang, Tianshuo Su, Chen Zou, and Yuanwei Fang. Updown accelerator instruction set architecture (isa) v2.4. Technical Report TR-2024-03, University of Chicago, July 2024.

[35] Andrew A. Chien, Jose M Monsalve-Diaz, Charles Colley, Jianru Ding, Alexander Fell, David F. Gleich, Henry Hoffmann, Moubarak Jeje, Rajat Khandelwal, Yanjing Li, Marziyeh Nourian, Andronicus Rajasukumar, Jiya Su, Tianshuo Su, Yuqing (Ivy) Wang, Wenyi Wang, Ruiqi Xu, and Tianchi Zhang. Updown: A supercomputer co-designed for scalable graph processing. 2024.

[36] IkJoon Choi, Seunghwan Hong, Kihyun Kim, Jeongsik Hwang, Seunghan Woo, Young-Sang Kim, Cheongryong Cho, Eun-Young Lee, Hun-Jae Lee, Min-Su Jung, Hee-Yun Jung, Ju-Seong Hwang, Junsub Yoon, Wonmook Lim, Hyeong-Jin Yoo, Won-Ki Lee, Jung-Kyun Oh, Dong-Su Lee, Jong-Eun Lee, Jun-Hyung Kim, Young-Kwan Kim, Su-Jin Park, Byung-Kyu Ho, Byongwook Na, Hye-In Choi, Chung-Ki Lee, Soo-Jung Lee, Hyunsung Shin, Young-Kyu Lee, Jang-Woo Ryu, Sangwoong Shin, Sungchul Park, Daihyun Lim, Seung-Jun Bae, Young-Soo Sohn, Tae-Young Oh, and SangJoon Hwang. 13.2 A 32gb 8.0gb/s/pin DDR5 SDRAM with a symmetric-mosaic architecture in a 5^{th}-generation 10nm DRAM process. In *IEEE International Solid-State Circuits Conference, ISSCC 2024, San Francisco, CA, USA, February 18-22, 2024*, pages 234–236. IEEE, 2024.

[37] Ian Cuttress. The Intel Skylake-X Review: Core i9 7900x, i7 7820x and i7 7800x. June 2017.

[38] W. J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills. Architecture of a message-driven processor. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, ISCA '87, page 189–196, New York, NY, USA, 1987. Association for Computing Machinery.

[39] Martin Dixon, Per Hammarlund, Stephan Jourdan, and Ronak Singhal. The next-generation intel® microarchitecture. *Intel Technology Journal*, 14(3), 2010. Accessed: 2025-02-08.

[40] Jonathon Evans. Nvidia grace. In *2022 IEEE Hot Chips 34 Symposium (HCS)*, pages 1–20, 2022.

[41] Mark Evers, Leslie Barnes, and Mike Clark. The amd next-generation "zen 3" core. *IEEE Micro*, 42(3):7–12, 2022.

[42] Yuanwei Fang, Tung T. Hoang, Michela Becchi, and Andrew A. Chien. Fast support for unstructured data processing: The unified automata processor. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 533–545, New York, NY, USA, 2015. Association for Computing Machinery.

[43] Yuanwei Fang, Chen Zou, Aaron J. Elmore, and Andrew A. Chien. 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, pages 55–68, New York, NY, USA, October 2017. Association for Computing Machinery.

[44] K. I. Farkas and N. P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. *SIGARCH Comput. Archit. News*, 22(2):211–222, apr 1994.

[45] John Feo, David Harper, Simon Kahan, and Petr Konecny. Eldorado. In *Proceedings of the 2nd Conference on Computing Frontiers*, CF '05, page 28–34, New York, NY, USA, 2005. Association for Computing Machinery.

[46] Chester Lam George Cozma. Graviton 3: First impressions. *Chips and Cheese*, May 2022.

[47] Kourosh Gharachorloo. MEMORY CONSISTENCY MODELS FOR SHARED-MEMORY MULTIPROCESSORS. 1995.

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

[49] Karl-Erwin Grosspietsch. Associative processors and memories: A survey. *IEEE Micro*, 12(3):12–19, 1992.

[50] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Amsterdam, 5 edition, 2012.

[51] Z. Hu, M. Martonosi, and S. Kaxiras. Tcp: tag correlating prefetchers. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pages 317–326, 2003.

[52] Zhigang Hu, Margaret Martonosi, and Stefanos Kaxiras. Tcp: Tag correlating prefetchers. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, page 317, USA, 2003. IEEE Computer Society.

[53] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. Hbm (high bandwidth memory) dram technology and architecture. In *2017 IEEE International Memory Workshop (IMW)*, pages 1–4, 2017.

[54] Jack Kang. Sifive u54-mc coreplex: Multicore, 64-bit application processor class risc-v cpu, 2017.

[55] David Kanter. Intel's haswell cpu microarchitecture.

[56] Dae-Hyun Kim, Byungkyu Song, Hyun-A. Ahn, Woongjoon Ko, Sung-Geun Do, Seokjin Cho, Kihan Kim, Seung-Hoon Oh, Hye-Yoon Joo, Geuntae Park, Jin-Hun Jang, Yong-Hun Kim, Donghun Lee, Jaehoon Jung, Yongmin Kwon, Youngjae Kim, Jaewoo Jung, Seongil O, Seoulmin Lee, Jaeseong Lim, Junho Son, Jisu Min, Haebin Do, Jaejun Yoon, Isak Hwang, Jinsol Park, Hong Shim, Seryeong Yoon, Dongyeong Choi, Jihoon Lee, Soohan Woo, Eunki Hong, Junha Choi, Jae-Sung Kim, Sangkeun Han, Jong-Min Bang, Bokgue Park, Jang-Hoo Kim, Seouk-Kyu Choi, Gong-Heum Han, Yoo-Chang Sung, Wonil Bae, Jeong-Don Lim, Seungjae Lee, Changsik Yoo, Sang Joon Hwang, and Jooyoung Lee. A 16gb 9.5gb/s/pin LPDDR5X SDRAM with low-power schemes exploiting dynamic voltage-frequency scaling and offset-calibrated readout sense amplifiers in a fourth generation 10nm DRAM process. In *IEEE International Solid-State Circuits Conference, ISSCC 2022, San Francisco, CA, USA, February 20-26, 2022*, pages 448–450. IEEE, 2022.

[57] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 195–201, Barcelona Spain, August 1998. ACM.

[58] Kartik Lakhotia, Laura Monroe, Kelly Isham, Maciej Besta, Nils Blach, Torsten Hoefler, and Fabrizio Petrini. Polarstar: Expanding the horizon of diameter-3 networks. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '24, page 345–357, New York, NY, USA, 2024. Association for Computing Machinery.

[59] Chester Lam. Popping the hood on golden cove. *Chips and Cheese*, 2021.

[60] Chester Lam. Sunny cove: Intel's lost generation. *Chips and Cheese*, 2022.

[61] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When prefetching works, when it doesn't, and why. *ACM Trans. Archit. Code Optim.*, 9(1), March 2012.

[62] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When prefetching works, when it doesn't, and why. *ACM Trans. Archit. Code Optim.*, 9(1), mar 2012.

[63] Jinhyung Lee, Kyungjun Cho, Chang Kwon Lee, Yeonho Lee, Jae-Hyung Park, Su-Hyun Oh, Yucheon Ju, Chunseok Jeong, Ho Sung Cho, Jaeseung Lee, Tae-Sik Yun, Jin Hee Cho, Sangmuk Oh, Junil Moon, Young-Jun Park, Hong-Seok Choi, In-Keun Kim, Seung Min Yang, Sun-Yeol Kim, Jaemin Jang, Jinwook Kim, Seong-Hee Lee, Younghyun Jeon, Juhyung Park, Tae-Kyun Kim, Dongyoon Ka, Sanghoon Oh, Jinse Kim, Junyeol Jeon, Seonhong Kim, Kyeong Tae Kim, Taeho Kim, Hyeonjin Yang, Dongju Yang, Minseop Lee, Heewoong Song, Dongwook Jang, Junghyun Shin, Hyunsik Kim, Chang-Ki Baek, Hajun Jeong, Jongchan Yoon, Seung-Kyun Lim, Kyo Yun Lee, Young Jun Koo, Myeong-Jae Park, Joohwan Cho, and Jonghwan Kim. 13.4 A 48gb 16-high 1280gb/s HBM3E DRAM with all-around power TSV and a 6-phase RDQS scheme for TSV area optimization. In *IEEE International Solid-State Circuits Conference, ISSCC 2024, San Francisco, CA, USA, February 18-22, 2024*, pages 238–240. IEEE, 2024.

[64] Levi Li. Micron advances hbm4 development, sets 2026 for mass production. *DigiTimesAsia*, 2024.

[65] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. Dramsim3: A cycle-accurate, thermal-capable dram simulator. *IEEE Computer Architecture Letters*, 19(2):106–109, 2020.

[66] Locuza. Die walkthrough: Alder lake-s/p and a touch of zen 3, 2023. Accessed: 2025-02-18.

[67] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhen-

grong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. The gem5 Simulator: Version 20.0+, September 2020. arXiv:2007.03152 [cs].

[68] John D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[69] John D. McCalpin. 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*, page 403–413, Berlin, Heidelberg, 2023. Springer-Verlag.

[70] Hassan Mujtaba. Intel sapphire rapids '4th gen xeon' cpu delidded by der8auer, unveils extreme core count die with 56 golden cove cores. *WCCFTech*, 2022.

[71] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14, 2018.

[72] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. The j-machine multicomputer: An architectural evaluation. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, page 224–235, New York, NY, USA, 1993. Association for Computing Machinery.

[73] NVIDIA Corporation. NVIDIA DGX B200 Datasheet. `https://resources.nvidia.com/en-us-dgx-systems/dgx-b200-datasheet`. Accessed: 2025-02-08.

[74] NVIDIA Corporation. NVIDIA H100 Tensor Core GPU Datasheet. `https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet`. Accessed: 2025-02-08.

[75] NVIDIA Corporation. NVIDIA Tesla P100 PCIe Datasheet. `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-p100/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf`, 2016. Accessed: 2025-02-08.

[76] NVIDIA Corporation. NVIDIA A100 Tensor Core GPU Datasheet. `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-nvidia-us-2188504-web.pdf`, 2020. Accessed: 2025-02-08.

[77] NVIDIA Corporation. NVIDIA V100 Tensor Core GPU Datasheet. `https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf`, 2020. Accessed: 2025-02-08.

[78] Marcelo Orenes-Vera, Aninda Manocha, Jonathan Balkind, Fei Gao, Juan L. Aragón, David Wentzlaff, and Margaret Martonosi. Tiny but mighty: designing and realizing

scalable latency tolerance for manycore SoCs. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 817–830, New York New York, 2022. ACM.

[79] J. Thomas Pawlowski. Hybrid memory cube (hmc). In *2011 IEEE Hot Chips 23 Symposium (HCS)*, pages 1–24, 2011.

[80] Steven A. Przybylski. Cache and memory hierarchy design. 1 1990.

[81] Andronicus Rajasukumar. Updown: An intelligent data movement architecture for large scale graph processing. 2023.

[82] Andronicus Rajasukumar, Jiya Su, Yuqing, Wang, Tianshuo Su, Marziyeh Nourian, Jose M Monsalve Diaz, Tianchi Zhang, Jianru Ding, Wenyi Wang, Ziyi Zhang, Moubarak Jeje, Henry Hoffmann, Yanjing Li, and Andrew A. Chien. Updown: Programmable fine-grained events for scalable performance on irregular applications, 2024.

[83] Andronicus Rajasukumar, Tianchi Zhang, Ruiqi Xu, and Andrew A. Chien. Updown: A novel architecture for unlimited memory parallelism. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '24, page 61–77, New York, NY, USA, 2024. Association for Computing Machinery.

[84] Cliff Robinson. Amd zen 3 at hot chips 33, August 2021.

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

[86] Andrey Semin. Intel® next generation nehalem microarchitecture. `https://janth.home.xs4all.nl/HPCourse/PDF/Nehalem_Uarch_TUDelft-Andrey_Semin.pdf`, 2007. Accessed: 2025-02-08.

[87] Yangho Seo, Jihee Choi, Sunki Cho, Hyunwook Han, Wonjong Kim, Gyeongha Ryu, Jungil Ahn, Younga Cho, Sungphil Choi, Seohee Lee, Wooju Lee, Chaehyuk Lee, Kiup Kim, Seongseop Lee, Sangbeom Park, Minjun Choi, Sungwoo Lee, Mino Kim, Taekyun Shin, Hyeongsoo Jeong, Hyunseung Kim, Houk Song, Yunsuk Hong, Seokju Yoon, Giwook Park, Hokeun You, Changkyu Choi, Hae-Kang Jung, Joohwan Cho, and Jonghwan Kim. 13.8 A 1a-nm 1.05v 10.5gb/s/pin 16gb LPDDR5 turbo DRAM with WCK correction strategy, a voltage-offset-calibrated receiver and parasitic capacitance reduction. In *IEEE International Solid-State Circuits Conference, ISSCC 2024, San Francisco, CA, USA, February 18-22, 2024*, pages 246–248. IEEE, 2024.

[88] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010.

[89] Seokbo Shim, Sungho Kim, Jooyoung Bae, Keunsik Ko, Eunryeong Lee, Kwidong Kim, Kyeongtae Kim, Sangho Lee, Jinhoon Hyun, Insung Koh, Joonhong Park, Minjeong Kim, Sunhye Shin, Dongha Lee, Yunyoung Lee, Sangah Hyun, Wonjohn Choi, Dain Im, Dongheon Lee, Jieun Jang, Sangho Lee, Junhyun Chun, Jonghoon Oh, Jinkook Kim, and Seok Hee Lee. A 16gb 1.2v 3.2gb/s/pin DDR4 SDRAM with improved power distribution and repair strategy. In *2018 IEEE International Solid-State Circuits Conference, ISSCC 2018, San Francisco, CA, USA, February 11-15, 2018*, pages 212–214. IEEE, 2018.

[90] Siarhei Siamashka. tinymembench. `https://github.com/ssvb/tinymembench`, 2017.

[91] D. Sima. The design space of register renaming techniques. *IEEE Micro*, 20(5):70–83, 2000.

[92] James E. Smith. Decoupled access/execute computer architectures. *ACM SIGARCH Computer Architecture News*, 10(3):112–119, April 1982.

[93] J.E. Smith and A.R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, 1988.

[94] Ryan Smith. Amd: Epyc genoa-x cpus with 1.1gb of l3 cache now available. *AnandTech*, 2023.

[95] Ryan Smith. Qualcomm snapdragon x elite performance preview: A first look at what's to come. October 2023.

[96] Ryan Smith. Apple announces m4 soc: Latest and greatest starts on 2024 ipad pro. May 2024.

[97] Ryan Smith. Nvidia blackwell architecture and b200/b100 accelerators announced: Going bigger with smaller data. *AnandTech*, 2024.

[98] Aaron Stillmaker and Bevan Baas. Scaling equations for the accurate prediction of cmos device performance from 180nm to 7nm. *Integration*, 58:74–81, 2017.

[99] Aaron Stillmaker and Bevan Baas. Corrigendum to "scaling equations for the accurate prediction of cmos device performance from 180nm to 7nm" [integr. vlsi j. 58 (2017) 74–81]. *Integr. VLSI J.*, 67(C):170, jul 2019.

[100] 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. The raw microprocessor: a computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[101] TechPowerUp. AMD Radeon R9 FURY X Specifications. `https://www.techpowerup.com/gpu-specs/radeon-r9-fury-x.c2677`. Accessed: 2025-02-08.

[102] G. Thirugnanam, N. Vijaykrishnan, and M.J. Irwin. A novel low power cam design. In *Proceedings 14th Annual IEEE International ASIC/SOC Conference (IEEE Cat. No.01TH8558)*, pages 198–202, 2001.

[103] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

[104] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1):25–33, 1967.

[105] Steve VanderWiel and David J Lilja. A survey of data prefetching techniques. In *Procs. of the 23rd International Symposium on Computer Architecture*. Citeseer, 1996.

[106] Krishnaswamy Viswanathan. Intel® memory latency checker v3.11, May 2024.

[107] Luming Wang, Xu Zhang, Songyue Wang, Zhuolun Jiang, Tianyue Lu, Mingyu Chen, Siwei Luo, and Keji Huang. Asynchronous Memory Access Unit: Exploiting Massive Parallelism for Far Memory Access, April 2024. arXiv:2404.11044 [cs].

[108] Yuqing Wang, Swann Perarnau, and Andrew A. Chien. Updown: Combining scalable address translation with locality control. In *Proceedings of the SC '24 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '24, page 1014–1024. IEEE Press, 2025.

[109] WikiChip. Broadwell microarchitecture. `https://en.wikichip.org/wiki/intel/microarchitectures/broadwell_(client)`. Accessed: 2025-02-08.

[110] WikiChip. Cascade lake microarchitecture. `https://en.wikichip.org/wiki/intel/microarchitectures/cascade_lake`. Accessed: 2025-02-08.

[111] WikiChip. Haswell microarchitecture. `https://en.wikichip.org/wiki/intel/microarchitectures/haswell_(client)`. Accessed: 2025-02-08.

[112] WikiChip. Ivy bridge microarchitecture. `https://en.wikichip.org/wiki/intel/microarchitectures/ivy_bridge_(client)`. Accessed: 2025-02-08.

[113] WikiChip. Sandy bridge (client) microarchitecture. `https://en.wikichip.org/wiki/intel/microarchitectures/sandy_bridge_(client)`. Accessed: 2025-02-08.

[114] WikiChip. Zen microarchitecture. `https://en.wikichip.org/wiki/amd/microarchitectures/zen`. Accessed: 2025-02-08.

[115] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. Imp: indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 178–190, New York, NY, USA, 2015. Association for Computing Machinery.

[116] Dan Zhang, Xiaoyu Ma, Michael Thomson, and Derek Chiou. Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 593–607, New York, NY, USA, 2018. Association for Computing Machinery.

[117] C.A. Zukowski and Shao-Yi Wang. Use of selective precharge for low-power content-addressable memories. In *1997 IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 3, pages 1788–1791 vol.3, 1997.