The Sparsity-Aware LazyGPU Architecture

Changxi Liu
National University of Singapore
Singapore
changxi_liu@u.nus.edu

Miao Yu
National University of Singapore
Singapore
miao.yu@u.nus.edu

Yifan Sun
William & Mary
USA
ysun25@wm.edu

Trevor E. Carlson
National University of Singapore
Singapore
tcarlson@comp.nus.edu.sg

ABSTRACT

General-Purpose Graphics Processing Units (GPUs) are essential accelerators in data-parallel applications, including machine learning, and physical simulations. Although GPUs utilize fast wavefront context switching to hide memory access latency, memory continues to be a significant bottleneck, limiting overall performance for many important workloads. Current GPU hardware enhancements focus on issuing memory requests in advance to help solve the memory bandwidth bottleneck and improve GPU performance. However, this approach can still be inefficient, leading to hardware contention and suboptimal resource utilization.

Instead of issuing memory requests in advance, we take an alternative view on improving GPU performance: lazily issuing memory requests to eliminate memory requests where either (a) the fetched values are zero or (b) they do not affect the result of the executing workload. Building on these insights, we propose LazyGPU, which integrates lazy execution cores with a *Zero Cache* to eliminate memory requests when all data required by a wavefront is zero. Moreover, LazyGPU utilizes instructions, including multiplication and multiply-add, to eliminate memory requests whose fetched values do not affect the outcomes of these instructions. For example, LazyGPU achieves a 2.18× speedup compared with the baseline at 60% weight sparsity for the inference of LLaMA 7B.

CCS CONCEPTS

• **Computer systems organization → Parallel architectures**.

KEYWORDS

GPU Architecture, Lazy Execution, Zero Cache

ACM Reference Format:
Changxi Liu, Miao Yu, Yifan Sun, and Trevor E. Carlson. 2025. The Sparsity-Aware LazyGPU Architecture. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3695053.3731009

ISCA '25, June 21–25, 2025, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/2025/06.
https://doi.org/10.1145/3695053.3731009

1 INTRODUCTION

General-Purpose Graphics Processing Units (GPUs) are used in a broad spectrum of applications, including machine learning [1, 14, 60], physical simulation [21, 39], autonomous driving [44, 83], and financial technology [18, 30]. However, as computing capabilities have increased significantly in recent years, memory bandwidth has not kept pace. Choked by the memory wall, it is challenging to fully utilize GPUs' computing capability [28]. Prior works have attempted to alleviate the memory bottleneck by improving the instruction execution logic, mainly using two sets of approaches, including *eager execution* [17] and *lazy execution* [8].

Eager execution. Eager execution fetches data from memory as early as possible so that the data-consuming instructions do not need to wait for the data, reducing stalls and improving Arithmetic Logic Unit (ALU) utilization. For example, prefetching machanisms [38, 42, 54] have been proposed to prevent long-latency memory accesses from blocking compute. The newest NVIDIA GPU chips with the Hopper architecture introduce Tensor Memory Accelerator (TMA) [17] and Asynchronous Transaction Barrier can issue memory requests from an independent group of threads or a hardware component so that the memory-fetching threads or hardware component can progress ahead of the computing threads. All these methodologies target issuing instructions as early as possible to increase hardware utilization.

However, eager execution does not always result in performance improvements, as it can introduce unnecessary memory requests or increase competition and congestion in the main memory system [11, 56]. Moreover, we find that eager execution misses opportunities to remove dead memory accesses, which have no effect on the program's outcome. Memory requests are usually eagerly issued to the memory system before eager execution cores realize that they are useless. Therefore, implementing mechanisms to cancel these issued memory requests is inefficient on eager execution cores.

Lazy execution. Lazy execution aims to address the limitations of eager execution by issuing memory requests only when needed. Prior works [8] have explored lazy execution in CPU architectures. However, the trade-off between the overhead and advantages of lazy execution on CPUs is critical as it introduces additional memory access delay. In contrast, our observation reveals that lazy execution is particularly suitable for GPUs. Unlike CPUs, GPU architectures are highly sensitive to memory bandwidth, and thread-level parallelism (TLP) on GPUs can mitigate the overhead of lazily issuing memory

requests, effectively hiding the long memory latency by leveraging a large number of threads. When one wavefront (also known as a warp on NVIDIA GPUs) encounters data to be fetched from memory, GPUs promptly schedule another wavefront to continue execution.

To investigate the effect of lazy execution, we first design the LazyCore, which executes load instructions by recording the transaction information and only issues the memory request when the data is needed by other instructions. The results show that LazyCore outperforms the baseline by up to 1.67× (1.08× on average) for benchmarks evaluated. These results demonstrate that LazyCore outperforms the baseline as it issues memory requests only when needed, thus reordering them and optimizing memory access sequencing.

To further exploit lazy execution, we propose LazyCore+①, where ① represents the optimization that eliminates memory requests, effectively reducing memory transactions when all data required by a wavefront is zero. Prior works [29, 59, 88] highlight that applications, especially deep learning workloads, load and store up to 90% of zero values [34]. While sparse tensor cores [16, 53, 82, 91] exploit zeros on GPUs, their reliance on specific sparsity patterns (e.g., 2:4 or 4:8) limits their sparsity rate and applicability across diverse workloads. Instead, LazyCore+① lazily issues memory requests, providing ample time to verify whether the data being fetched is zero. To accelerate this process, we introduce a specialized cache to store masks [26, 36], which indicates whether a value should be fetched. Unlike prior approaches that depend on rollback mechanisms [62], our lazy execution model allows verification to occur naturally between address calculation and data usage, avoiding unnecessary complexity. By eliminating redundant memory requests when a wavefront requires only zero values, LazyGPU reduces memory bandwidth consumption and congestion, ultimately improving efficiency.

Moreover, we propose LazyGPU (LazyCore+①②), where the optimization ② represents the elimination of dead memory requests whose values have no effect on the outcome of subsequent instructions. We define these types of instructions as ⊗ instructions as their result is unaffected by the value of one operand if its corresponding operand is zero, such as *multiply*, *multiply-add*, and *and* instructions. Load requests, whose destination register is subsequently processed by the ⊗ instructions are considered dead if the corresponding input register in the ⊗ instructions contains zero value. In this work, our proposed **LazyGPU** consists of the LazyCore with the additional optimization ① and ②.

We make the following contributions:

- We propose LazyCore, a new GPU architecture design to implement lazy execution of memory instructions. LazyCore addresses the challenge of bandwidth contention caused by a large number of memory requests by deferring issuing memory requests until they are needed. Our investigation illustrates that LazyCore outperforms the baseline by up to 1.67× (1.08× on average) across a wide range of GPU workloads. Exploring the use of lazy cores on GPUs is an underexplored design scheme. The purpose of lazy cores is to inspire future GPU architecture design.
- Next, we present LazyCore+①, which utilizes information only available in the core to eliminate memory requests whose fetched values required by wavefronts are all zero. Additionally, LazyCore+① can eliminate memory requests even when fetched data contains non-zero values, as long as those values are not needed.
- Furthermore, LazyGPU further eliminates memory requests whose value have no effect on subsequent instructions (optimization ②). Optimization ② achieves an additional memory request reduction by combining ⊗ instructions (e.g., multiplication) with zero values. Specifically, our results show that LazyGPU reduces memory requests by 31.1% for the inference and 31.4% for the training over the baseline of ResNet-18 with 50% weight sparsity.
- We evaluate LazyGPU on ResNet-18, demonstrating performance improvements of 1.20× for inference and 1.16× for training without pruning. When the weight sparsity is increased to 50%, our work achieves 1.31× for inference and 1.24× for training. Additionally, LazyGPU achieves a 2.18× speedup compared with the baseline at 60% weight sparsity for the inference of LLaMA 7B. Moreover, we also evaluate LazyGPU across a wide range of applications. The results show that LazyGPU outperforms the baseline hardware, achieving an average speedup of 1.08× (up to 1.67×) for benchmarks with default inputs and 1.28× on average (up to 3.66×) for benchmarks with 50% sparsity. The hardware overhead of LazyGPU amounts to only 0.009% of the total die area compared to the baseline GPU.

2 BACKGROUND

Eager execution architectures. The primary function of eager execution architectures is to promptly issue and commit instructions, aiming to improve hardware performance by increasing hardware utilization. Prior architecture directions encompass various key areas, including speculative execution [24, 50], branch prediction [65, 69], value prediction [63, 64, 67], and prefetching [38, 42]. For example, Lee et. al. [42] propose an adaptive prefetch throttling scheme to prefetch data for future access on GPUs. Each of these methodologies aims to make execution architecture more eager by issuing or committing instructions earlier. Moreover, the rollback mechanism associated with eager execution architectures, including speculative execution and branch prediction, introduces significant resource overheads and, in addition, can incur large speculation penalties. Alternatives are needed to improve performance in an energy-efficient way.

Lazy execution architectures. In contrast, lazy execution architectures execute instructions only when needed, aiming to optimize instructions and eliminate dead instructions. Lazy Superscalar is a representative lazy execution architecture work, which proposes fusing instructions to improve CPU performance and leveraging a large look-ahead distance provided by lazy execution [8]. Additionally, Alsop et. al. [3] introduce lazy release consistency to achieve more efficient coherence on GPUs. However, lazy execution architectures introduce additional delay as instructions are executed only when needed. Different from prior works, LazyGPU focuses on alleviating memory contention by reordering and eliminating unnecessary memory requests on GPUs.

GPU architecture. GPU architectures are well-suited for workloads with the same program but different data. Typically, GPUs execute hundreds to thousands of threads simultaneously to process the workload, organized in a hierarchy structure. For example,

NVIDIA A100 GPUs feature 108 streaming multiprocessors (SMs), with each SM containing 64 CUDA cores to process warps. AMD R9 Nano GPUs [4] contain 64 compute units (CUs), each equipped with 4 SIMD units. Each CU can support up to 40 active wavefronts (a 64-thread group), allowing for a theoretical maximum of 2560 concurrent wavefronts across the entire GPU. A wavefront is the basic execution unit on SIMD units (GPU cores) and the instruction executed for all threads within a wavefront is always the same. GPUs can hide long-latency memory requests by scheduling among wavefronts. GPUs maintain large register files and the context of wavefronts is stored in these registers, allowing them to be quickly scheduled into execution units. Most importantly, all modern GPUs follow an eager execution methodology. In the rest of this work, we will aim to demonstrate that current GPU architectures could be improved through the use of lazy processing techniques.

Zero-value related optimization. The zero values exist across various applications, particularly in neural network tasks. Specifically, pruning techniques that prune parameters in neural networks to zero values are widely studied [32, 72, 89]. Prior works [26, 27, 36, 57, 76, 84] explored the optimization on zero values in domains including algorithms, compilers, and hardware designs. Furthermore, existing works [26, 36] have explored utilizing zero caches to reduce memory access latency for zero values in CPU architectures. Zero caches track zero values compactly and are designed for fast responses. To minimize access latency, they are typically small. Each bit in the zero cache indicates if a data block consists of all zeros, increasing cache capacity and effective bandwidth when the zero-value rate is high [26, 36]. These methodologies typically issue memory requests for data and their zero masks concurrently to avoid long memory access latencies. However, directly applying these methodologies to memory bandwidth-sensitive hardware, like GPUs [86], might not significantly improve performance as memory requests for zero values, in these works, are still issued to the memory system.

Our goal, therefore, is to use zero values to eliminate unnecessary memory requests. However, current hardware processor architectures, including out-of-order execution [19] and pipelining [71], prevent hardware from utilizing these special values as zero-related instructions are usually issued before being optimized. Previous research [62] suggests using speculative execution to optimize these zero-related instructions in advance, but the cost of rollback operations caused by incorrect predictions is not negligible for GPU architectures. The lazy execution processor is more suited for optimizing zero-valued data as it provides a large look-ahead distance. To the best of our knowledge, there are no prior works that have been able to optimize the GPU with zero-related optimization and lazy execution.

Lazy execution cores provide long look-ahead distances to dynamically optimize instructions. This paper primarily discusses the integration of lazy execution cores with zero caches. Additionally, when combined with approximate computing methodologies like Doppleganger [52], lazy execution cores can dynamically coalesce memory requests for similar data at runtime. Furthermore, lazy execution cores can cooperate with other zero-skip methodologies, such as sparse tensor cores [82, 91] to optimize instructions online. There are also related accelerator works, including Forms [85], Gospa [22], S2ta [46], OLAccel [58].

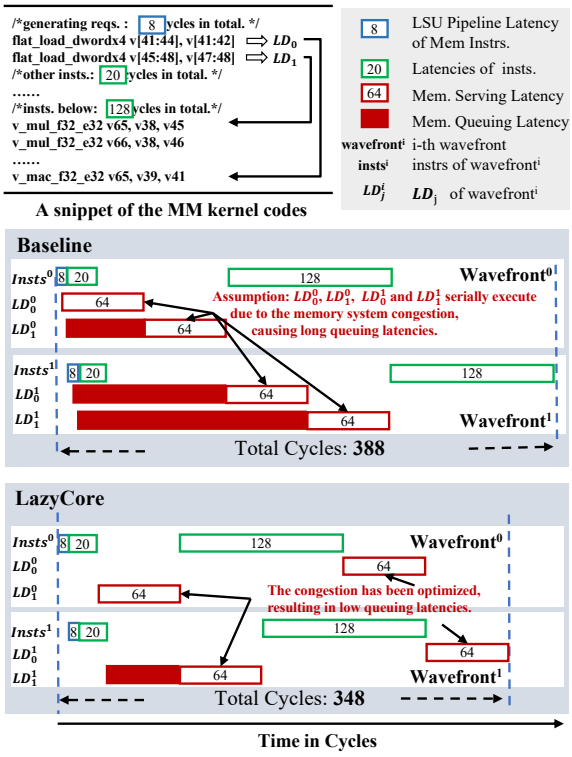


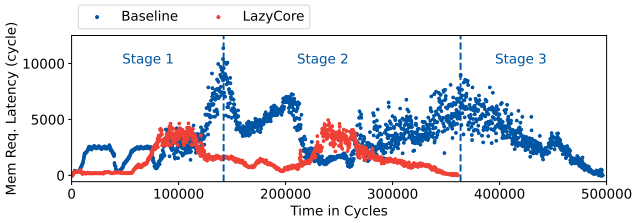
Figure 1: The execution timeline of a snippet from the MM [2] benchmark, using two wavefronts, illustrates the performance improvement of LazyCore over the baseline. To better illustrate the benefits of our methodology, we assume a memory request is served with a latency of 64 cycles. The baseline degrades overall GPU performance by eagerly issuing non-critical memory requests (LD_0^0), which block the critical memory requests LD_1^1 from Wavefront^1. In contrast, LazyCore issues memory requests when needed, and reduces memory system congestion.

3 OBSERVATIONS AND CHALLENGES

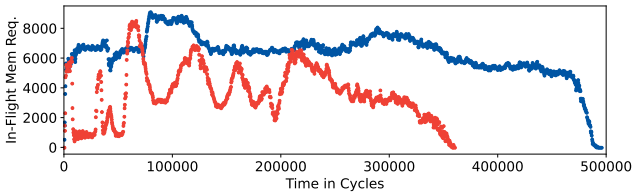
The LazyGPU architecture is rooted in insights into GPU execution and data patterns. In this section, we present our observations and challenges that motivate the benefits of the LazyGPU.

Observation 1: Issuing memory requests only when needed on GPUs has similar and better performance compared to eagerly issuing memory requests. To validate that lazily issuing memory requests does not significantly sacrifice GPU performance, we configure the LazyCore to postpone all memory requests and only issue them when the fetched value is needed. More details about the implementation of the LazyCore are discussed in Section 4.1.

Although GPUs offer high memory bandwidth, hardware resources like miss status holding registers (MSHRs) and read reorder buffers [6, 7] still limit memory system parallelism. When excessive concurrency blocks architectural resources, such as MSHRs, memory ports, or interconnect bandwidth, it prevents critical memory requests from entering the memory system. In such cases, high

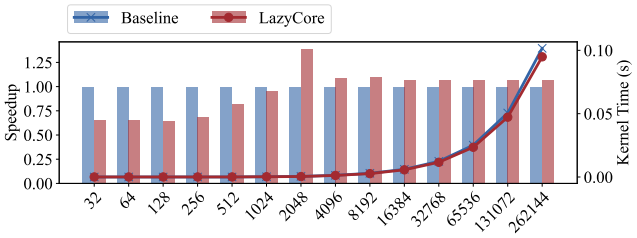


(a) Memory access latencies demonstrate that LazyCore is better. Although access latencies in the range of thousands of cycles may seem high, similar values have been reported in recent research [12, 81].

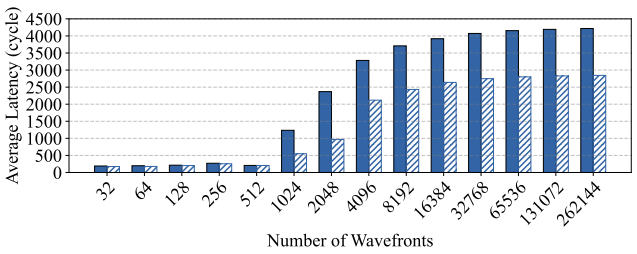


(b) The number of in-flight memory requests.

Figure 2: The memory access latencies and number of in-flight memory requests for the Matrix Multiplication (MM) benchmark [2] with 2048 wavefronts on both the baseline and LazyCore. (a) LazyCore can reduce memory access latencies compared to the baseline. (b) The memory request issue rate (increasing slope) is higher on LazyCore than on the baseline, as LazyCore generates the next batch of memory requests faster due to reduced memory access latencies. Additionally, the average ALU utilization of LazyCore is 39.4% higher than that of the baseline.



(a) The speedup (bars, left y-axis) and kernel execution time (lines, right y-axis) achieved by LazyCore compared to the baseline.



(b) The average latencies of all memory accesses achieved by the baseline and LazyCore. The latency for each memory access is defined as its retired time minus the issued time to the memory system. Lower is better for latency.

Figure 3: The result of the Matrix Multiplication (MM) benchmark [2] with the number of wavefronts ranging from 32 (tiny) to 262144 (large), where each wavefront processes the same workload. (a) LazyCore approaches the baseline as the number of wavefronts increases from 32 to 1024, and starts to outperform the baseline beyond 2048 wavefronts. LazyCore performs better than the baseline when the number of concurrency memory requests is excessive. (b) The memory access latency approaches around the average memory access latency when the number of concurrent memory requests is excessive. At this point, the latency of LazyCore remains lower than that of the baseline.

access latency becomes a performance bottleneck, as the GPU can no longer leverage parallelism to hide these latencies effectively. To mitigate congestion, prior works have explored throttling-based methods [15, 37, 68], at a cost of limiting bandwidth utilization by restricting concurrent memory accesses in certain cases.

Instead of reducing concurrency, LazyCore mitigates memory system congestion by deferring non-critical memory requests while ensuring that critical memory requests are issued. In general, memory access latency consist of three parts: queuing, serving, and networks-on-chip (NoC) latencies [61]. While the serving and NoC latencies remain relatively stable, queuing latency increases as memory system congestion grows. Long latency memory transactions, especially the memory transactions on the critical path, stall the instruction pipeline, preventing additional memory requests and reducing bandwidth utilization. By deferring non-critical memory requests, LazyCore reduces queuing latency, allowing memory instructions on the critical path to finish earlier and the instruction pipelines to issue more memory transactions. Consequently, the memory bandwidth utilization is increased.

Figure 1 provides an example illustrating the execution differences between LazyCore and the baseline. As seen in the snippet of the MM kernel assembly, memory instructions, issued almost at the same time, can have different levels of urgency. In this particular example, LD_1 is more urgent than LD_0. On the baseline GPU, memory requests are issued immediately. However, due to memory system congestion, some requests are served with a delay. The memory requests issued later (e.g., LD_1^1 from $wavefront^1$) suffer from a long latency and cause a delay in the execution of subsequent instructions that may not depend on fetched data. In contrast, the LazyCore avoids such congestion by allowing the subsequent instructions to execute and issuing memory requests only when needed. As a result, LazyCore achieves higher overall performance compared to the baseline.

To validate the simplified example introduced above, we conduct an experiment and monitor how key metrics change over time. Using MM as an example (see Figure 2), LazyCore significantly reduces memory access latencies, particularly extreme bursts of memory access latencies observed on the baseline (see Figure 2a). Consequently, the LazyCore can issue memory requests more efficiently, as shown by the steeper slope in Figure 2b. This leads to higher memory bandwidth utilization and an improvement in overall GPU performance.

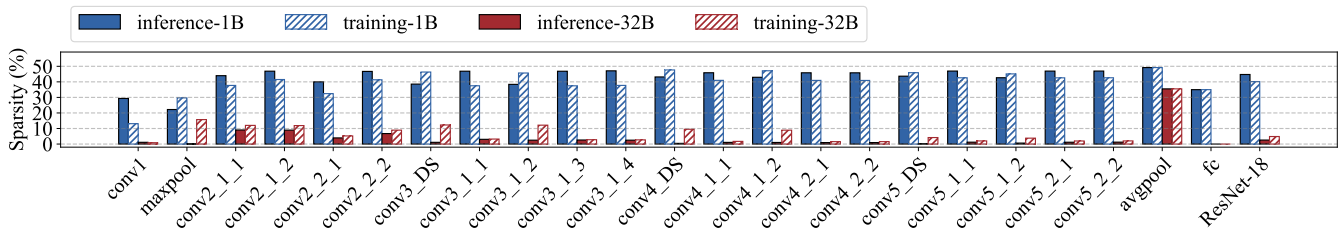


Figure 4: The sparsity of values accessed by memory transactions for the inference and training of ResNet-18 [35], where pruning [34] is applied to achieve 50% weight sparsity. The results demonstrate a potential for reducing the amount of bandwidth needed. The sparsity indicates the percentage of data blocks that are all zeros among all fetched data, measured at *1B* (byte-level granularity) and *32B* (transaction-level granularity). *DS* means downsampling [35].

The LazyCore outperforms the baseline when GPU workloads become concurrency sensitive, meaning that memory request latencies depend heavily on the number of concurrent memory requests. Here, we evaluate the matrix multiplication (MM) benchmark [2] with the number of wavefronts ranging from 32 to 262144, where each wavefront processes the same workload. The results demonstrate that the LazyCore approaches the performance of the baseline and outperforms it beyond 2048 wavefronts. This is because the LazyGPU reduces memory contention by delaying memory requests until they are needed. When the number of wavefronts in GPU workloads is large, GPU workloads become concurrency sensitive. LazyGPU enhances the memory bandwidth utilization, thereby improving overall performance.

We then investigate the reason behind the speedup being flattened beyond an input size of 2048. As shown in Figure 2a, memory contention on the baseline is highest in `Stage 1` when processing the first N wavefronts, as memory requests are issued almost simultaneously. On the simulated R9 Nano architecture, up to 2560 wavefronts can be executed concurrently. However, the tiled matrix multiplication (MM) algorithm incurs high register usage, reducing the maximum number of concurrent wavefronts (N) to 768. In `Stage 2`, contention decreases as requests become more interleaved. In `Stage 3`, it further declines as the number of concurrent wavefronts diminishes. Consequently, the speedup of LazyCore follows three phases: the highest in `Stage 1`, moderate in `Stage 2`, and the lowest at `Stage 3`. Thus, in Figure 3a, the speedup initially increases as the number of wavefront grows, since the overlap between `Stage 1` and `Stage 3` decreases. When the number of wavefronts reaches 2048, `Stage 1` does not have overlap with `Stage 3`, resulting in the maximum speedup 1.4×. Beyond this point, the length of `Stage 2` increases, causing the speedup to decline to 1.1× when the number of wavefronts is 8192, and finally aligns with `Stage 2` (1.07 × speedup).

Observation 2: Utilizing zero values in GPU applications, especially employing ⊗ instructions with zero values, is a potential way to alleviate memory contention. Zero values widely exist in GPU applications, such as DNA analysis software (BarraCUDA [41, 88]), singular value decomposition (SVD) [40], and neural network applications [27, 59, 66]. Among these applications, deep learning applications are the most popular applications for GPUs. Zero values in deep learning workloads are often introduced by layers like ReLU [43] or dropout [9]. Pruning methodologies [34, 70, 72, 89, 90] can significantly increase the presence of zero values by sparsifying the weights of neural networks.

We investigate the inference and training of ResNet-18 [35] with ImageNet dataset [23]. We apply pruning [34] to achieve 50% weight sparsity, and then analyze the zero-value rates for each layer. We find that the percentage of data blocks where all data is zero are 44.7% for inference and 40.2% for training when the data block granularity is 1 byte as evaluated for ResNet-18.

Moreover, we find that these zero values are usually processed by ⊗ instructions, where the result is unaffected by the value of one operand if the corresponding operand is zero. For example, the major floating-point ALU instructions for benchmarks, including MM and FIR, are multiply-add instructions (⊗ instructions). Leveraging this property in combination with memory and ⊗ instructions enables the elimination of memory requests for the operand whose corresponding operand is zero.

However, the zero values within the inputs and weights of neural networks can be randomly distributed [34], leading to a decrease in the percentage of data blocks where all data is zero as the granularity of the data block increases. As shown in Figure 4, when the granularity is 32 bytes, the percentages of data blocks where all data is zero are 2.7% for inference and 4.8% for training. These values are significantly lower compared to when the granularity of the data block is 1 byte, as shown in Figure 4. Unfortunately, on modern GPUs [5, 49, 77], the transaction sizes for data transfers from DRAM to L2 or L2 to L1 on GPU are typically 32 or 64 bytes, which limits the ability to fully exploit the zero values in memory transactions as the whole data block need to be fetched as long as one value inside is non-zero. Leveraging zero values to mitigate memory contention presents two significant challenges.

Challenge 1: The absence of information, available only within the core, prevents the memory hierarchy from optimizing transactions in which the portion of the data required by the wavefront is all zeros. To mitigate the impact of the random distribution of zero values, we observe that threads within a wavefront usually require only a portion of the data block to be fetched by one memory transaction. For example, the memory instruction with a stride (stride = 2) requires only half of the data fetched in a single transaction. Unfortunately, it is not feasible to eliminate such memory transactions where the required portion of the data is zero as memory systems lack this information about the specific data requirements of the wavefront.

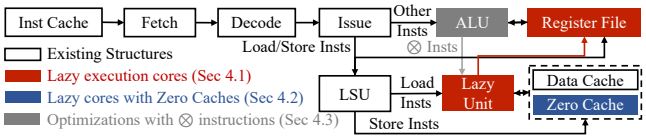


Figure 5: The LazyGPU architecture. The LazyGPU adds new structures (the *Lazy Unit* and the *Zero Caches*) and extends existing structures (ALU and Register Files) of the GPU cores.

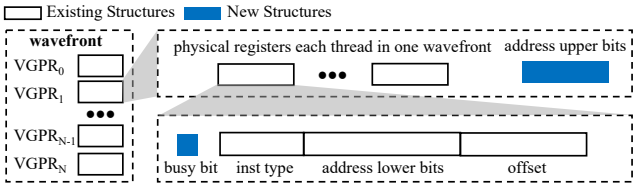


Figure 6: The data structure housed within the registers to accommodate memory requests pending to issue for lazy execution. The *busy bit* is set to indicate if the physical registers contain pending memory requests. The *inst type* is the type of load instructions. The combination of *address upper bits*, *address lower bits* and *offset* are the memory access address. These data can be stored with the destination registers to mitigate hardware overhead as these registers are available until the LazyCore issues memory requests.

Challenge 2: The lack of instruction information for processing fetched data impedes potential avenues to further reduce memory requests. Utilizing the combination of memory and ⊗ instructions poses a challenge, as GPU cores, when issuing memory requests, do not have future information about the instructions that will process the data being fetched. It is inefficient to issue these memory requests to the memory system and then send cancel requests once GPU cores realize that these requests are not needed. The need to cancel data requests would complicate the overall architecture. In this work, we have developed a solution to overcome these challenges and describe our solution in Section 4.

4 THE LAZYGPU ARCHITECTURE

In this section, we outline the workflow of the LazyGPU microarchitecture. We first introduce the mechanism of lazy execution cores, denoted as LazyCore. Subsequently, we introduce the LazyCore+①, which integrates a new structure, called *Zero Caches* (to be described below) with the LazyCore to eliminate memory requests when all data required by a wavefront is zero while ensuring cache coherency. Finally, we illustrate the LazyGPU architecture (LazyCore+①②), which eliminates dead memory instructions through the combination of optimizations with ⊗ instructions and LazyCore+①.

The workflow of the LazyGPU is depicted in Figure 5. The LazyGPU is built based on lazy execution cores, using the Load-Store Unit (*LSU*) to first send load memory requests into *Lazy Unit*. Next, the *Lazy Unit* stores memory requests information into target registers and marks them as busy (depicted in Figure 5 (red)). When the source registers of instructions are busy, the *Lazy Unit* issues the corresponding memory requests into the *Data Cache*. When the data is ready, the corresponding registers are set as non-busy, allowing the wavefront to be rescheduled into the compute units (CU) to resume executing the instruction. Details of the Lazy Unit and how it interacts with the rest of the GPU will be discussed in detail in Section 4.1.

Indicated in Figure 5 (blue), the LazyGPU integrates the lazy execution core with *Zero Caches*. The LazyCore+① first fetches the mask for memory requests from *Zero Caches*. Next, the LazyCore+① eliminates memory requests, where all data required by a wavefront is zero according to the mask, and then issues the remaining memory requests when needed. More details are discussed in Section 4.2.

Furthermore, the LazyGPU eliminates dead memory requests through the optimizations between ⊗ and memory instructions, depicted in Figure 5 (grey). Dead memory requests are those whose values have no effect on subsequent instructions. For example, memory requests, which are processed by ⊗ instructions with corresponding operands as zero, do not influence the output of this instruction. These memory requests are subsequently considered dead if their destination register is rewritten by subsequent instructions, signifying that these values are no longer used. We will elaborate on this in Section 4.3.

4.1 Lazy Execution Cores (LazyCore)

The LazyCore issues memory requests when needed. The LazyCore marks physical registers to be busy if they are destination registers of memory instructions, indicating that memory requests are pending due to the lazy mechanism. LazyGPU employs a busy bit for physical registers, enabling efficient dependency management with minimal hardware overhead. The *busy bit* illustrated in Figure 6 denotes the one-bit introduced by the LazyCore to mark the busy status. Memory instructions, including those in both AMD GCN3 [48], and NVIDIA's PTX [47], allow for multiple target registers for one instruction. On the LazyCore, each of these registers is marked as busy since subsequent instructions might independently utilize them. For example, the instruction `flat_load_dwordx4 v[33:36], v[30:31]` loads data into registers `v33` to `v36`, requiring that each register be marked as busy independently.

The LazyCore verifies if the source registers required by the executing instruction are ready, as their corresponding memory requests may still be pending due to lazy execution. When the source registers of instructions are non-busy, LazyCore processes these instructions the same as the baseline GPU cores except for memory instructions, which are issued when needed. When encountering busy source registers, the *Lazy Unit* dispatches the corresponding memory requests to the memory system. When the source registers become non-busy after completing these memory requests, the instruction is re-executed to avoid changing the pipeline.

Address translation. *Lazy Unit* operates on virtual addresses. On architectures with VIPT (Virtually Indexed, Physically Tagged) L1 caches, memory addresses are translated to physical addresses before being sent to the L1. In contrast, on architectures with VIVT (Virtually Indexed, Virtually Tagged) L1 caches, memory requests are sent to L1 without translation. The evaluated architecture uses

Table 1: The *inst type*, that is stored in the physical registers, indicates the instruction type or the offset of the first destination register for instructions with multiple target registers. The *ld.XB* means loading *X* bytes of data into current and its subsequent register and reg_Y indicates that the ID for the first destination register of multiple target register instructions is the current register ID minus Y.

Inst Type	ld.1B	ld.2B	ld.4B	ld.8B	ld.16B	reg_{-3}	reg_{-2}	reg_{-1}
Binary	100	101	110	111	000	011	010	001

VIPT L1 caches, and the TLB translation overhead is accounted for in the evaluation of LazyCore.

To mitigate hardware overhead, all memory request information from the LazyCore is stored within the destination registers of memory instructions, as shown in Figure 6. The destination registers are available as the LazyCore issues memory requests before using them. These memory requests usually encapsulate information, including the *tid*, *inst type*, *address*, and *offset*. The *tid* specifies the thread id within the wavefront requesting the data, while the *inst type* denotes the instruction type for memory requests. Furthermore, the *address* represents the starting memory address of the data block for the transaction, while the *offset* specifies the location of the required data within the block.

The *tid* is not required to be stored as memory requests are buffered within the registers corresponding to their respective threads. Moreover, we segregate the *address* into the *upper bits of address* and *lower bits of address*. The allocation of bits for the *lower bits* is contingent upon the available remaining space within the register, while the *upper bits of address* are those remaining bits, which are shared among all threads within the wavefront. In our experience, we find that it is rare for memory requests from one memory instruction of one wavefront to have different upper bits as they only require the highest remaining bits to be the same. Moreover, memory requests exhibiting different *upper bits of address* compared to their counterparts are promptly issued without lazy execution.

The *inst type* indicates the instruction type for the memory requests, as shown in Table 1. For GPU architectures, including AMD GCN3 and PTX, memory instructions allow for multiple target registers for one instruction. However, subsequent instructions might not initially utilize the first destination register for these memory instructions with multiple target registers. The reg_Y, as shown in Table 1, guides the LazyCore to locate where the memory request information is stored. Note that for GPU architectures allowing at most 4 target registers, 3 bits are sufficient to include the instruction information and the offset to the first destination register. For architectures that support loading 8 or more target registers, we can use alternative methodologies, such as extending the *inst type* bits or allowing multiple jumps to find the first destination registers.

In conclusion, GPUs have numerous cores, and even minor hardware additions can lead to substantial hardware overhead. LazyCore stores most memory transaction information into destination registers, effectively minimizing hardware overhead.

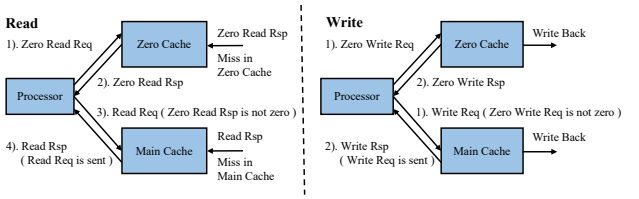


Figure 7: LazyGPU leverages *Zero Caches* to derive masks that indicate whether the data to be fetched is zero, subsequently eliminating read memory requests where the portion of data required by wavefronts is entirely zero. For the write instruction, LazyGPU writes zero masks into *Zero Caches* to maintain cache coherence.

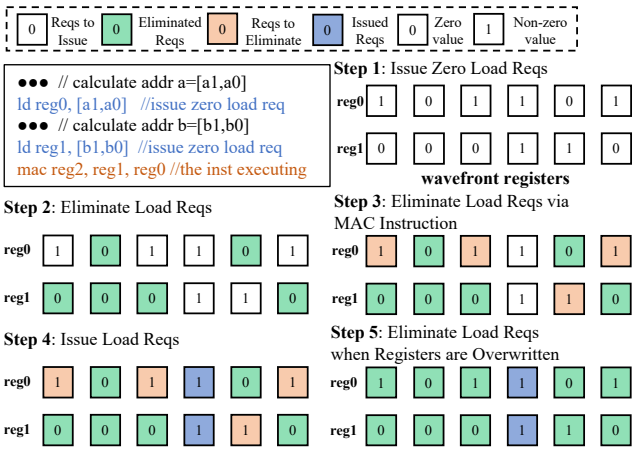


Figure 8: An example of how the LazyGPU eliminates memory requests via masks and multiply-add instructions.

4.2 Lazy Cores with *Zero Caches* (*LazyCore+①*)

Integrating lazy cores with *Zero Caches* (LazyCore+①) can further enhance GPU performance by leveraging information available only within the core to eliminate memory requests where the portion of the fetched data required by wavefronts is entirely zero. To address **Challenge 1**, where the memory hierarchy lacks memory transaction information within the cores, LazyCore+① first fetches masks from *Zero Caches* and then utilizes information from the core to identify the portions of data required by wavefronts. By combining the masks with the information from the core, LazyCore+① verifies if the required data is entirely zero. Removing these unnecessary memory requests reduces bandwidth contention and queuing latency, thereby improving GPU performance.

Figure 7 illustrates the workflow of LazyCore+① for handling read and write memory transactions. For read requests, LazyCore+① first retrieves the response (*Zero Read Rsp*) from the *Zero Caches* and then combines it with instruction information to determine if the required data is entirely zero. If this condition is met, LazyCore+① eliminates the corresponding memory request, sets the *busy bit* to false, and initializes the register to zero. Otherwise, the LazyCore+① issues the read request to the *main caches*. For write requests, the LazyCore+① only sends the *Zero Write Req* into the *Zero Caches* if

the data being written is zero. Otherwise, it sends both *Zero Write Req* and *Write Req* for the non-zero data to the *Zero Caches* and main caches, respectively. *Write Req* and *Zero Write Req* are the write requests for original data and their masks, respectively.

LazyCore+① can rapidly verify if the data being fetched is zero using *Zero Caches*. To ensure a fair comparison without adding cache structures for the LazyCore+① and LazyGPU, parts of the normal caches are repurposed as *Zero Caches*. Each single bit within the *Zero Caches* can represent $\mathcal{M}$ bits of data from the main memory (we set one bit to represent 32 bits throughout this work). Additionally, the *Zero Caches* have the same transaction size as the normal cache, allowing one transaction in the *Zero Caches* to transfer the zero masks for $\mathcal{M}$ transactions in the normal cache.

Furthermore, data is interleaved across multiple memory partitions to bolster memory-level parallelism. If the LazyCore+① requires load/store masks for multiple partitions, multiple zero memory requests are necessary, leading to significant memory contention. Therefore, the LazyCore+① configures each memory request within *Zero Caches* to correspond to data from the same partition. If the number of partitions is $\mathcal{P}$ and the interleaving size is $\mathcal{I}$, one memory transaction within *Zero Caches* represents $\mathcal{M}$ consecutive data blocks if $\mathcal{I} \geq \mathcal{M}$. If $\mathcal{I} < \mathcal{M}$, it represents $\frac{\mathcal{M}}{\mathcal{I} \cdot \mathcal{P}}$ consecutive sets of $\mathcal{I}$ data blocks.

Figure 8 (**Step 1**, **2**) illustrates an example of how LazyCore+① eliminates memory requests where the data required by a wavefront is entirely zero. Within the GPU, the instruction processes $\mathcal{W}$ data simultaneously, where $\mathcal{W}$ represents the number of threads within one wavefront. Typically, $\mathcal{W}$ is 32 or 64 for modern GPUs. For clarity, in this example, we set $\mathcal{W}$ to 6, and the transaction size is 4 bytes, represented as one box in Figure 8.

The LazyCore+① issues *Zero Read Reqs* during the execution of the *ld* instruction in **Step 1**. Upon receiving the responses, it identifies registers with zero values and sets these registers to zero while marking their *busy bit* as false. This initialization occurs before the execution of the *mac* instruction. Subsequently, in **Step 2**, LazyGPU eliminates all memory requests associated with registers that have zero values and *busy bits* marked as false, as illustrated in Figure 8 (Green Boxes). This process reduces unnecessary memory traffic and improves performance.

Cache coherence. Ensuring cache coherency is crucial for the LazyCore+①. LazyGPU updates only the zero cache when a memory request writes only zeros to reduce memory traffic. A coherence issue may arise if the normal caches return data faster than zero caches [26, 36]. We access the zero caches before the normal caches, ensuring that coherence operates correctly.

To maintain coherence between *Zero Caches* and other on-chip caches, we employ the same coherence mechanisms as those used by the corresponding normal caches. For example, in the evaluated architecture, L1 *Zero Caches* adopt the same write-around mechanism as the normal L1 cache. Since writes bypass the L1 and are forwarded directly to the L2, the L1 *Zero Caches* do not retain dirty data. L2 *Zero Caches*, consistent with the normal L2 design, is physically partitioned, with each bank responsible for a disjoint region of the memory address space. As a result, the L2 *Zero Caches* remain coherent without requiring additional hardware-level coherence mechanisms.

Table 2: The configuration parameters used for R9 Nano GPUs on the MGPUSim [73] to evaluate the LazyGPU. GPRs are the general-purpose registers.

Component	R9 Nano	LazyGPU
Shader Array (SA)	16 per GPU	16 per GPU
CU	1.0GHz, 4 per SA	1.0 GHz, 4 per SA
L1 Vector Cache	64KiB, 4-way per SA	56KiB, 4-way per SA
L1 Zero Cache		8KiB, 4-way 1 per SA
L2 Cache	256KiB, 16-way 8 per GPU	224KiB, 16-way 8 per GPU
L2 Zero Cache		32KiB, 16-way 8 per GPU
DRAM	4GiB	4GiB
GPRs	64KiB, 4 per CU	64KiB, 4 per CU

4.3 LazyCore+① with ⊗ Instructions (LazyGPU)

By incorporating ⊗ instructions, LazyGPU (LazyCore+①②) can further eliminate memory requests based on the LazyCore+①. As depicted in **Challenge 2**, conventional GPUs issue memory requests while executing memory instructions, making it impossible to know which instructions will process the fetched data at the time of issuing. LazyGPU addresses this limitation by issuing memory requests only when needed, allowing the LazyCore to possess the necessary information about the instructions that will process the data. When decoding instruction opcodes, LazyGPU identifies ⊗ instructions, and optimizes memory requests accordingly.

The LazyGPU eliminates memory requests associated with these ⊗ instructions via two steps. The LazyGPU first suspends issuing memory requests when their corresponding operands in ⊗ instructions evaluate to zero. This is because the outcomes of ⊗ instructions are unaffected by the values of these memory requests.

Then, LazyGPU permanently eliminates memory requests when the destination registers of these suspended requests are overwritten by other instructions, or when the execution of the current wavefront is completed. This indicates that the values of these memory requests are no longer useful. In contrast, if subsequent instructions still rely on the data from the suspended memory requests before their elimination, LazyGPU ensures these requests, whose *busy bit* is true, are issued to the memory system to satisfy the execution of those instructions.

Figure 8 (**Step 3**, **4**, **5**) illustrates an example of how LazyGPU eliminates memory requests whose corresponding operand contains zero value. As shown in **Step 3** of Figure 8, LazyGPU suspends memory requests (Orange Box) as they multiply with zero. Then, in **Step 4**, memory requests (*Blue Box*), associated with source registers containing non-zero values, are issued to the memory system. Subsequently, LazyGPU proceeds to execute the *mac* and subsequent instructions. Upon subsequent instructions overwriting *reg0* and *reg1*, the LazyGPU permanently eliminate these memory requests as depicted in **Step 5** (orange boxes change to green boxes).

5 EVALUATION

In this section, we present a comprehensive evaluation of LazyGPU.

5.1 Experimental Setup

Architecture simulated. We evaluate the LazyGPU using the AMD GCN3 architecture as its Instruction Set Architecture (ISA) is

Table 3: The benchmarks used to evaluate the LazyGPU.

Abbr.	Suite	Workload
AES	Hetero-Mark [74]	AES-256 Encryption.
FIR	Hetero-Mark [74]	FIR filter.
KMeans	Hetero-Mark [74]	Kmeans Clustering.
PR	Hetero-Mark [74]	Pagerank.
SC	AMD APP SDK [2]	Simple Convolution.
MM	AMD APP SDK [2]	Matrix Multiplication.
MT	AMD APP SDK [2]	Matrix Transpose.
NBody	AMD APP SDK [2]	Physics Simulation.
ReLU	DNNMark [25]	Rectified Linear Unit.
SPMV	SHOC [20]	Sparse Matrix-Vector Multiplication.
FFT	SHOC [20]	Fast Fourier Transform.
Stencil2d	SHOC [20]	Stencil Computation.
BFS	SHOC [20]	Breadth-First Search.
BICG	PolyBench [31]	BiCGStab Linear Solver [80].
ATAX	PolyBench [31]	Matrix Transpose, Vector Multiplication.
Backprop	Rodinia [13]	Back Propagation.
NW	Rodinia [13]	Needleman-Wunsch algorithm.
ResNet-18 [35]		Neural Network applications.
LLaMA 7B [79]		Large Language Model.

openly accessible. Gutierrez et. al. [33] have pointed out the substantial impact that ISAs on the conclusions from the simulation results. Specifically, we utilize R9 Nano, which is the default architecture of the MGPUSim [73] and belongs to the AMD GCN3 architecture, as the baseline to validate the LazyGPU. Recently, NaviSim [10] introduces RDNA, which is mainly for gaming and 3D rendering, into MGPUSim. We do not use RDNA as CDNA is still the default architecture for MGPUSim and current supercomputer centers, including El Capitan, and Frontier [78], still use AMD GPUs with CDNA architecture.

The detailed configuration for R9 Nano and the LazyGPU are shown in Table 2. The DRAM, L2, and L1 bandwidth of GPUs in this work are 256 GB/s, 512 GB/s, and 2 TB/s, respectively, with a theoretical performance of 8.192 TFLOPS. We use the default latencies of L1, L2, and DRAM in MGPUSim, which are 60, 112, and 146 cycles (round-trip from the CU), respectively. The GPU is configured to use core-side L1 caches and memory-side L2 caches. L1 and L2 caches are connected via a crossbar. The memory system consists of 16 L2 caches, each connected to a DRAM channel, with L2 caches organized in a 128B interleaving fashion. The DRAM operates in a first-come, first-served manner.

In the LazyGPU, the L1 Zero Cache is shared among all CUs within a shader array, enhancing their cache hit rate. Consequently, each GPU hosts 16 L1 Zero Caches, aligning with the number of shader arrays. The transaction size from DRAM to L2 and from L2 to L1 is typically 32 or 64 bytes on AMD and NVIDIA GPUs [5, 49, 77]. In all experiments, we set the default transaction size to 32 bytes.

Simulator utilized. We adapt MGPUSim [73] to support the LazyGPU. We choose MGPUSim as it is the simulator that supports the latest AMD GPU architecture and its comprehensive suite of tools for analyzing GPU workload behavior [75].

GPU workloads. We evaluate the LazyGPU on a wide range of applications as shown in Table 3. All GPU workloads are developed in OpenCL and compiled using the AMD ROCm compiler with the default settings. For benchmarks, including AES, FIR, SC, MM, and ReLU, we randomly initialize their inputs to zero based on the sparsity rate. For applications with a sparsity structure, such as SpMV, we set only their inputs without sparsity structure to zero according to the sparsity rate. This scenario is common when pruning weights for Graph Neural Networks (GNN) [32, 87]. For benchmarks whose inputs lack zeros, including NW and BFS, we always keep their default inputs. When the sparsity is 0, we keep the default inputs for all benchmarks.

The inputs and weights of ResNet-18 and large language models (LLMs) [55, 79] are obtained from PyTorch checkpoints [60]. The data type is *float* as full-precision training remains prevalent in most AI models to achieve high accuracy. For the ResNet-18, we use ImageNet [23] as the dataset. We apply the pruning methodology [34] to evaluate ResNet-18 with unstructured weight sparsity. Han et al. [34] point out that the weight sparsity can progressively increase from 0% to 90% during the training process, as redundant weights are pruned to improve model efficiency without sacrificing accuracy. The sparsity of weights is zero without pruning.

LLMs have emerged as one of the most important workloads on GPUs due to their remarkable performance across various natural language process tasks. To evaluate the efficiency of LazyGPU, we also test LLaMA 7B [79] with the WikiText [51] validation set. We use Wanda [72] to introduce unstructured sparsity by pruning weights while maintaining accuracy. Simulating end-to-end ML workloads on GPU simulators is extremely time-consuming. To address this, we employ the Photon sampling methodology [45] to simulate kernels when full execution is infeasible. Specifically, we apply kernel-level sampling to ResNet-18 and both kernel-level and wavefront-level sampling to LLaMA 7B.

Speedup. We compare the kernel execution time of the LazyGPU to that of the baseline GPU architecture, R9 Nano. The speedup is calculated using $\frac{T_{base}}{T}$, where T_{base} and T represent the kernel execution time of workloads executing on the baseline and our improved architecture, respectively.

5.2 Overall Performance

LazyCore. Figure 9 demonstrates that the LazyCore achieves a 1.05× and a 1.01× performance improvement for the inference and training of ResNet-18 compared with the baseline, establishing a foundation for further optimizations. Specifically, the layer `conv5_2_1` exhibits a 1.28× performance improvement for inference, while the `conv5_1_2` layer has a 1.28× performance improvement for the training, demonstrating that the LazyCore can achieve better performance compared with the baseline.

LazyCore+①. As depicted by Figure 9, the LazyCore+① is built upon the LazyCore and incorporates *Zero Caches* to eliminate memory requests whose values required by wavefronts are all zero. It achieves a 1.16× and a 1.07× speedup for the inference and training of the ResNet-18 over the baseline, respectively.

LazyGPU. The LazyGPU, which integrates the ⊗ optimization with LazyCore+①, yields a significant performance improvement over the baseline, achieving a 1.31× and a 1.24× improvement for the inference and training of ResNet-18, respectively, as shown in Figure 9. Notably, the layer `conv5_2_1` and `conv5_1_2` exhibit the highest speedups, with a 1.91× speedup for the inference and a 1.97× speedup for training among all layers.

Comparison with Eager Execution and Zero Caches No prior works have explored zero caches with eager execution in GPU

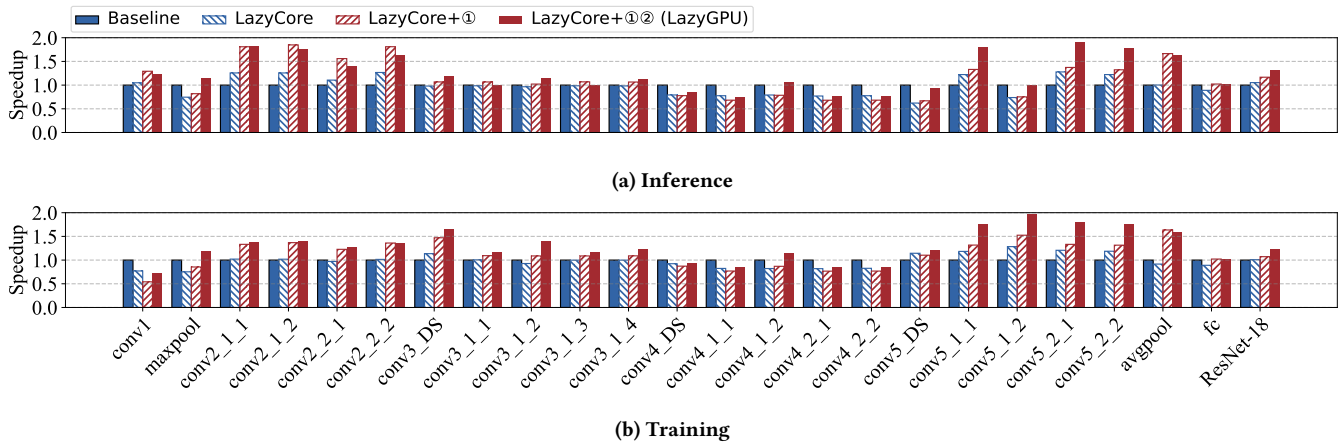


Figure 9: The speedup for ResNet-18 and its layers for the inference and training via the LazyCore, LazyCore+①, and LazyCore+①② (LazyGPU) over the baseline, where pruning [34] is applied to achieve 50% weight sparsity. *DS* means downsampling. LazyCore denotes issuing memory requests when needed. LazyCore+① represents the optimization of discarding memory requests whose values required by wavefronts are all zeros. LazyGPU signifies the optimization of eliminating memory requests whose value have no effect for subsequent instructions. The speedups of LazyGPU are 1.31× for inference and 1.24× for training. In comparison, we evaluate eager execution with zero caches [26, 36], achieving speedups of 1.26× and 1.02×, respectively.

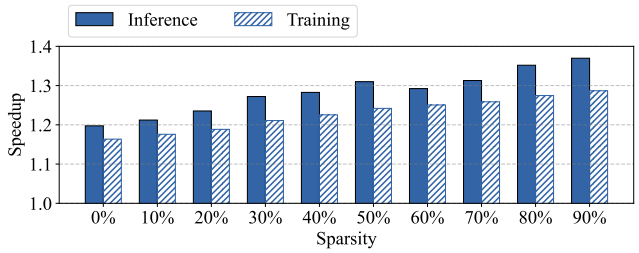
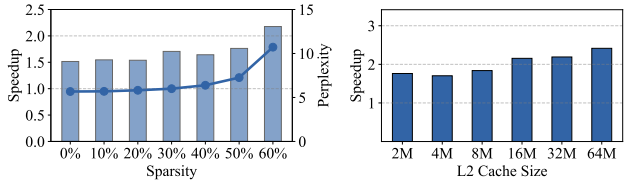


Figure 10: The LazyGPU outperforms the baseline for the inference and training of ResNet-18 across different weight sparsities [34]. The results show that the performance improvement increases as the weight sparsity increases. We set the beginning of the y-axis to 1 for clarity.



(a) The speedup and perplexity with increasing sparsity. **(b) The speedup with increasing L2 sizes.**

Figure 11: The LazyGPU outperforms the baseline on the same hardware configuration for the LLaMA 7B [79] inference, with unstructured pruning applied using Wanda [72]. The speedup (left y-axis with bars in (a)) increases as sparsity rises from 0% to 60%, while the perplexity (right y-axis with lines) maintains accuracy. (b) The speedup at 50% sparsity shows that LazyGPU achieves higher speedup with larger L2 cache sizes. The speedup is calculated compared to the baseline configuration with the same L2 cache size.

architectures. We evaluate eager execution with zero caches [36] on GPUs, showing that it improves ResNet-18 with 50% weight sparsity by 1.26× for the inference and 1.02× for the training. LazyGPU outperforms this approach [36] as the zero caches with eager execution still issue memory requests for zero values to the memory system.

ResNet-18 with different weight sparsities. Figure 10 presents the speedup achieved by LazyGPU compared to the baseline for ResNet-18 inference and training across different weight sparsities. We evaluate LazyGPU's performance under varying weight sparsity levels, as weight sparsity can increase from 0% to 90% during the training process by pruning redundant weights [34].

When weight sparsity is zero, LazyGPU still improves performance. This is because lazy execution contributes a performance improvement of 1.05× for inference and 1.01× for training. Additionally, zero values existing in the inputs, introduced by layers such as ReLU and dropout, further enhance performance. As a result, LazyGPU achieves a final speedup of 1.20× for inference and 1.16× for training when the weight sparsity is zero.

Moreover, we observe that the performance improvement increases from 1.20× to 1.37× as weight sparsity grows from 0% to 90% for inference, and from 1.16× to 1.29× for training. This indicates that LazyGPU performs better as more zero values are introduced into the neural networks.

Large language models (LLMs). We observe that LazyGPU can also improve the performance of LLMs, as shown in Figure 11. When the weight sparsity is zero, LazyGPU demonstrates a 1.52× speedup compared to the baseline. Unlike ResNet-18, the inference of LLaMA 7B does not have specific layers, such as ReLU and

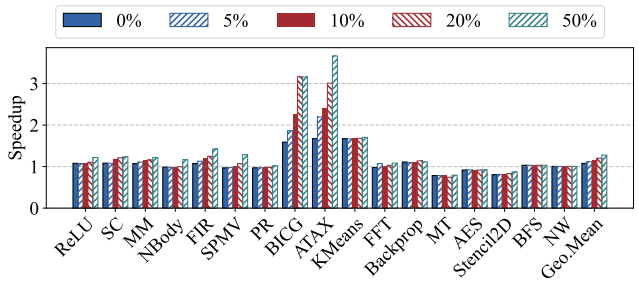


Figure 12: The speedup of LazyGPU compared to the baseline across each benchmark with original inputs (0% sparsity) and inputs with varying sparsity (5%, 10%, 20%, and 50%).

dropout, to introduce zero values. Therefore, performance speedup is solely attributed to the bandwidth improvement of lazy execution, as discussed in Section 3. Furthermore, as weight sparsity increases, the speedup achieved by LazyGPU also grows. The results presented in Figure 11a show that LazyGPU achieves a 2.18× speedup at 60% weight sparsity. This demonstrates that LazyGPU can continue improving the performance of LLaMA 7B as weight sparsity increases.

Additionally, we evaluate LazyGPU with varying L2 cache sizes as the inputs and weights for LLMs are large and highly sensitive to cache sizes. The results, as shown in Figure 11b, demonstrate that LazyGPU consistently delivers higher performance compared to the baseline, even with larger L2 sizes. This indicates that LazyGPU effectively optimizes memory accesses and reduces contention, leading to improved performance regardless of the cache size.

Results across diverse GPU workloads. As shown in Figure 12, LazyGPU achieves up to 1.67× speedup (1.08× on average) across various GPU workloads using the default inputs (sparsity = 0%). These results demonstrate that lazy execution is not limited to neural network applications. Instead, it benefits a broad spectrum of GPU workloads by enhancing memory bandwidth utilization, thereby improving overall performance.

Moreover, LazyGPU, as shown in Figure 12, consistently improves GPU performance as the sparsity of the inputs increases. When the sparsity reaches 50%, LazyGPU achieves up to 3.66× speedup (1.28× on average). Specifically, for workloads whose inputs lack zero, including `BFS` and `NW`, LazyGPU achieves similar performance. For streaming workloads such as `ReLU` and `FIR`, the performance improvement can be attributed to the compression of zero values into a single bit, which further reduces bandwidth consumption. However, LazyGPU is designed to optimize concurrency-sensitive workloads, resulting in lower performance improvements for latency-sensitive cases. Workloads such as `MT`, `AES` and `Stencil2D` [2] assume that all or most data can be buffered in shared memory. Consequently, they first fetch data in shared memory before executing ALU instructions, making them inherently latency-sensitive.

In summary, the LazyGPU improves the performance of GPU workloads by reordering memory requests and effectively eliminates memory requests associated with zero values or ⊗ instructions. It yields a significant speedup over GPU workloads, including

Table 4: Cache configurations. *Zero Caches* are denoted as *ZCache*. $\alpha L1 + \beta L2$ indicates that LazyGPU utilizes α L1 and β L2 caches as *Zero Caches*. $\frac{1}{8}$L1 + $\frac{1}{8}$L2 is the configuration utilized by the LazyGPU.

Configurations	L1 Vector Cache	L1 *ZCache*	L2 Cache	L2 ZCache
$\frac{1}{2}L1 + \frac{1}{2}L2$	32KiB per SA	32KiB per SA	128KiB	128KiB
$\frac{1}{2}L1 + \frac{1}{8}L2$	32KiB per SA	32KiB per SA	224KiB	32KiB
$\frac{1}{2}L1 + \frac{1}{32}L2$	32KiB per SA	32KiB per SA	248KiB	18KiB
$\frac{1}{8}L1 + \frac{1}{2}L2$	56KiB per SA	8KiB per SA	128KiB	128KiB
$\frac{1}{8}$L1 + $\frac{1}{8}$L2	**56KiB per SA**	**8KiB per SA**	**224KiB**	**32KiB**
$\frac{1}{8}L1 + \frac{1}{32}L2$	56KiB per SA	8KiB per SA	248KiB	18KiB
$\frac{1}{16}L1 + \frac{1}{2}L2$	60KiB per SA	4KiB per SA	128KiB	128KiB
$\frac{1}{16}L1 + \frac{1}{8}L2$	60KiB per SA	4KiB per SA	224KiB	32KiB
$\frac{1}{16}L1 + \frac{1}{32}L2$	60KiB per SA	4KiB per SA	248KiB	18KiB

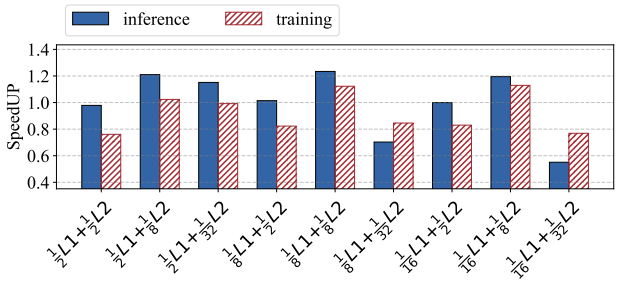


Figure 13: The speedup of LazyGPU outperforms that of the baseline for the inference and training of ResNet-18 without pruning across various cache configurations, as listed in Table 4. We select $\frac{1}{8}$L1 + $\frac{1}{8}$L2 as it performs the best performance among the evaluated configurations.

ResNet-18, LLaMA 7B. LazyGPU is not limited to neural network applications; it also demonstrates performance improvements in various benchmarks, highlighting its broad applicability and effectiveness in optimizing GPU performance.

5.3 Zero Caches

We opt to adopt the $\frac{1}{8}L1 + \frac{1}{8}L2$ configuration as the default configuration for the LazyGPU, as outlined in Table 2. This configuration offers an adequate balance by providing ample caches for buffering masks, while simultaneously ensuring that the remaining caches are sufficient to buffer the original data.

Selecting the right partitioning is important for LazyGPU to achieve good performance. As shown in Figure 13, the performance of the LazyGPU over the baseline is showcased across various cache configurations, with detailed specifications provided in Table 4. To ensure a fair comparison, the LazyGPU ensures that the aggregate normal cache and zero cache sizes match those of the baseline. We separate zero caches as they have different tag sizes compared to normal caches. For LazyGPU, the tag sizes are 21 for normal L1 caches and 19 for L1 zero-caches. This difference arises from their different cache sizes and each bit in the zero-cache representing 4B.

As illustrated in Figure 13, we note that small *Zero Caches* can impede the efficiency of *Zero Read/Write Req*, thereby resulting in diminished GPU performance. For example, the configuration

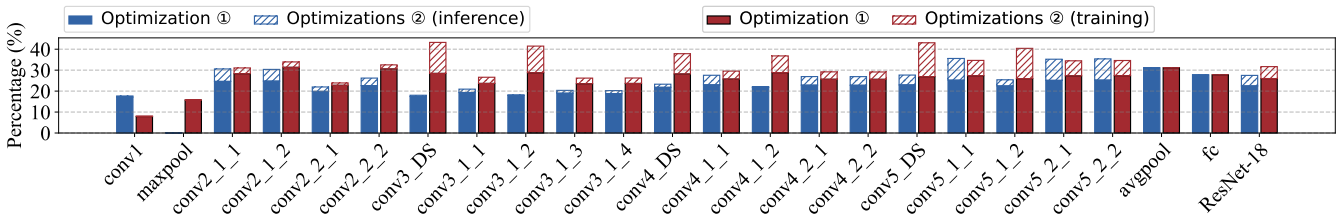
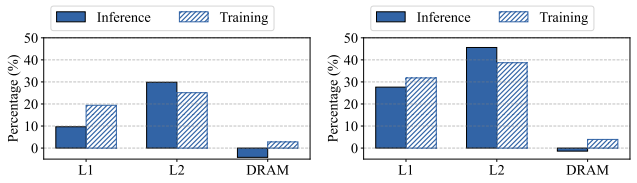
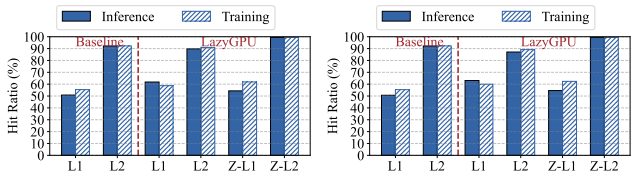


Figure 14: The memory requests that are eliminated by *Zero Caches* and ⊗ instructions for ResNet-18 and its layers for the inference and training, where pruning [34] is applied to achieve 50% weight sparsity. *DS* means downsampling.



(a) Weight without pruning. **(b) Weight sparsity is 50%.**

Figure 15: The percentage of memory requests mitigated by LazyGPU compared to the baseline for the inference and training of ResNet-18. It demonstrates that LazyGPU efficiently reduces the contention within the memory system.



(a) Weight without pruning. **(b) Weight sparsity is 50%.**

Figure 16: The cache hit rate of the baseline and the LazyGPU for the inference and training of ResNet-18. *Z-L1* and *Z-L2* mean L1 and L2 zero caches. The hit rate of Z-L2 on LazyGPU is about 99%, preventing zero mask fetches from becoming a bottleneck. Additionally, the L1 hit rate of LazyGPU is improved due to the *Z-L1*.

$\frac{1}{16}L1+\frac{1}{32}L2$ yields a reduction of performance to 55.1% and 76.9% of the baseline for the inference and training of ResNet-18. Conversely, excessively large *Zero Caches* lead to a significant reduction in the working-set size that the cache can contain, consequently slowing down GPU performance. For example, the configuration $\frac{1}{2}L1 + \frac{1}{2}L2$ results in just 97.9% and 76.1% baseline performance for the inference and training of ResNet-18.

Furthermore, the configurations with $\frac{1}{8}L2$ *Zero Caches* exhibit similar performance, highlighting the criticality of L2 cache and L2 *Zero Caches* for LazyGPU. When the average access latency in a cache miss of the L1 *Zero Caches* or L1 cache misses are high due to their small size, TLP mitigates memory access latencies. Nonetheless, the significance of size of the L2 normal caches and L2 *Zero Caches* persists due to substantial costs associated with accessing DRAM.

5.4 Reasons behind Performance Improvement

We investigate the reasons behind the performance improvement of the LazyCore+① and LazyGPU, focusing on the memory request elimination attributed to the optimization ① and ②. Figure 14 shows that the optimization ① eliminates memory requests whose fetched data required by a wavefront is all zero, contributing to 22.5% and 26.0% of the eliminated requests for the inference and training of ResNet-18, respectively. Inside these requests, transactions whose fetched data block is entirely zero only contribute to 2.7% and 4.8% of the eliminated requests for ResNet-18 inference and training, as depicted in Figure 4. The LazyCore+①, which is the combination of the LazyCore with *Zero Caches*, amplifies memory request elimination by removing requests whose fetched data block contains non-zero data.

The optimization ② also contributes to the memory request elimination. For example, it eliminates 8.6% and 5.4% of the memory requests for ResNet-18 inference and training, respectively. Additionally, for some layers of ResNet-18, including conv2_1_1 and conv5_1_1, the optimization ② outperforms the LazyCore+① on the contribution for the overall performance improvement.

We investigate the number of memory requests processed by each level of the memory hierarchy, as shown in Figure 15. The results reveal that LazyGPU effectively eliminates memory requests, reducing them by 9.7%, 29.9%, and -4.2% for inference and 19.4%, 25.1%, and 2.8% for training at the L1, L2, and DRAM levels, respectively, when the weight sparsity is zero. We observe that the memory requests at the DRAM level slightly increase. This is because the L2 normal cache size of LazyGPU is smaller than that of the baseline. Our findings show that LazyGPU primarily reduces memory requests at the L1 and L2 levels, which account for approximately 97.0% of the total memory requests.

When the weight sparsity is 50%, we observe significant reduction in memory requests. Specifically, LazyGPU reduces requests by 27.6%, 45.6%, and -1.4% for inference and 31.8%, 38.7%, and 3.9% for training at the L1, L2, and DRAM levels, respectively. This indicates that as LazyGPU eliminates more redundant memory requests, the overall pressure on each level of the memory system decreases.

Figure 16 demonstrates cache hit results of L1, L2 and corresponding zero caches. LazyGPU prevents zero mask fetching from becoming a bottleneck as the L2 zero cache hit rate reaches 99%. Additionally, one memory transaction from L2 zero caches to L1

zero caches provides masks for 1024 bytes. LazyGPU achieves better performance due to multiple factors, including reduced memory contention and improved L1 hit rate as shown in Figure 15 and Figure 16, respectively.

5.5 Hardware Overhead

The hardware overhead of the LazyGPU comes from two parts: (1). *busy bits* for GPRs to trace instruction dependencies and record if memory requests are completed; (2). *address upper bits* to store address upper bits shared by a group of registers with the same register name but belonging to different threads inside a wavefront.

Busy Bits. The LazyGPU uses *busy bits* to detect the instruction dependency and record the response of memory requests. As each SIMD unit has 16,384 physical registers, the LazyGPU requires 8KiB for each compute unit. Alternatively, reducing the number of registers to accommodate busy bits is another option to maintain the same area. However, performance may degrade if the reduced register count becomes insufficient for some workloads.

Addresses Upper Bits. Within R9Nano, there are four distinct types of load memory requests, loading 1, 2, 4, and 8, 16 bytes. The number of register offset types is 3 due to the maximum number of target registers. Therefore, a 3-bit field is designed for *inst type*, as shown in Table 1. Given the size of the transaction being 32 bytes, the required size for the *offset* field is determined to be 5 bits. The remaining *address lower bits* encompass 24 bits as the register of R9Nano has 32 bits. As the memory address of R9Nano is 64 bits, 35 bits are required to represent addresses' upper bits.

The *address upper bits* are shared by a group of registers with the same register name but belonging to different threads inside a wavefront to save hardware resources. For GPUs where the number of threads per wavefront is N and the physical register count is M, the LazyGPU requires $\frac{35M}{N}$ bits to record address upper bits. For example, each compute unit of the AMD R9 Rano has 4 SIMD units and each SIMD unit has 16,384 physical vector registers. Therefore, each compute unit requires an additional 4.375 KiB for *address upper bits*, given the number of threads per wavefront is 64.

In conclusion, the total hardware overhead introduced by the LazyGPU is an area increase of only 0.009% compared with the overall die size of R9 Nano [4].

6 CONCLUSION

In this paper, we propose a new GPU architecture, the LazyGPU. The LazyGPU employs lazy execution cores, deferring memory requests until needed to alleviate unnecessary memory contention. Additionally, LazyGPU utilizes *Zero Caches* and special instructions, such as multiplication and multiply-add, to eliminate memory requests with zero values or those that do not affect the outcome of the program. Our experimental results demonstrate that the LazyGPU outperforms the baseline architecture in both the inference and training of ResNet-18, achieving 1.31× and 1.24× speedup, respectively. LazyGPU can also improve the performance of LLaMA 7B by 1.52× and 2.18× when the weight sparsity is zero and 60%, respectively.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments. Changxi Liu is in part supported by a Ph.D. scholarship from the Ministry of Education, Singapore. We also thank Intel and VMware for supporting this research. This work was supported by the National Science Foundation (NSF) under Grant No. 2246035.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[2] Advanced Micro Devices, Inc. 2015. AMD APP SDK. https://github.com/ghostlander/AMD-APP-SDK/releases/download/v2.9.1/AMD-APP-SDK-v2.9.1-lnx64.tar.xz

[3] Johnathan Alsop, Marc S Orr, Bradford M Beckmann, and David A Wood. 2016. Lazy release consistency for GPUs. In *International Symposium on Microarchitecture (MICRO)*.

[4] AMD. 2015. AMD Radeon R9 Series Gaming Graphics Cards with High-Bandwidth Memory.

[5] AMD. 2019. RDNA Architecture. https://GPUopen.com/wp-content/uploads/2019/08/RDNA_Architecture_public.pdf.

[6] AMD. 2023. *AMD Radeon RX Graphics Cards.*

[7] AMD. 2024. *Versal Adaptive SoC Programmable Network on Chip and Integrated Memory Controller 1.1 LogiCORE IP Product Guide (PG313).* https://docs.amd.com/r/en-US/pg313-network-on-chip/Read-Reorder-Buffer

[8] Görkem Aşılıoğlu, Zhaoxiang Jin, Murat Köksal, Omkar Javeri, and Soner Önder. 2015. LaZy superscalar. In *International Symposium on Computer Architecture (ISCA)*.

[9] Pierre Baldi and Peter Sadowski. 2013. Understanding dropout. In *International Conference on Neural Information Processing Systems (NeurIPS)*.

[10] Yuhui Bao, Yifan Sun, Zlatan Feric, Michael Tian Shen, Micah Weston, José L Abellán, Trinayan Baruah, John Kim, Ajay Joshi, and David Kaeli. 2022. Navisim: A highly accurate GPU simulator for AMD RDNA GPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[11] Rahul Bera, Konstantinos Kanellopoulos, Shankar Balachandran, David Novo, Ataberk Olgun, Mohammad Sadrosadat, and Onur Mutlu. 2022. Hermes: Accelerating long-latency load requests via perceptron-based off-chip load prediction. In *International Symposium on Microarchitecture (MICRO)*.

[12] Niladrish Chatterjee, Mike O'Connor, Gabriel H Loh, Nuwan Jayasena, and Rajeev Balasubramonia. 2014. Managing DRAM latency divergence in irregular GPGPU applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[13] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *International Symposium on Workload Characterization (IISWC)*.

[14] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[15] Yanhao Chen, Ari B Hayes, Chi Zhang, Timothy Salmon, and Eddy Z Zhang. 2018. Locality-Aware Software Throttling for Sparse Matrix Operation on GPUs. In *USENIX Annual Technical Conference (USENIX ATC)*.

[16] Zhaodong Chen, Zheng Qu, Yuying Quan, Liu Liu, Yufei Ding, and Yuan Xie. 2023. Dynamic N:M fine-grained structured sparse attention mechanism. In *Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

[17] Jack Choquette. 2022. NVIDIA Hopper GPU: Scaling performance. In *Hot Chips 34 Symposium (HCS)*.

[18] Michael Creel and Mohammad Zubair. 2012. High performance implementation of an econometrics and financial application on GPUs. In *SC Companion: High Performance Computing, Networking Storage and Analysis*.

[19] Adrian Cristal, Daniel Ortega, Josep Llosa, and Mateo Valero. 2004. Out-of-order commit processors. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[20] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. 2010. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the General Purpose GPUs (GPGPU)*.

[21] Sambit Das, Phani Motamarri, Vishal Subramanian, David M Rogers, and Vikram Gavini. 2022. DFT-FE 1.0: A massively parallel hybrid CPU-GPU density functional theory code using finite-element discretization. *Computer Physics Communications* (2022).

[22] Chunhua Deng, Yang Sui, Siyu Liao, Xuehai Qian, and Bo Yuan. 2021. GOSPA: An energy-efficient high-performance globally optimized sparse convolutional neural network accelerator. In *International Symposium on Computer Architecture (ISCA)*.

[23] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *International Conference on Computer Vision and Pattern Recognition (CVPR)*.

[24] Gregory Diamos and Sudhakar Yalamanchili. 2010. Speculative execution on multi-GPU systems. In *International Symposium on Parallel & Distributed Processing (IPDPS)*.

[25] Shi Dong and David Kaeli. 2017. DNNMark: A Deep Neural Network Benchmark Suite for GPUs. In *Proceedings of the General Purpose GPUs (GPGPU)*.

[26] Julien Dusser, Thomas Piquet, and André Seznec. 2009. Zero-content augmented caches. In *International Conference on Supercomputing (ICS)*.

[27] Vinod Ganesan, Sanchari Sen, Pratyush Kumar, Neel Gala, Kamakoti Veezhinathan, and Anand Raghunathan. 2020. Sparsity-aware caches to accelerate deep neural networks. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[28] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W Mahoney, and Kurt Keutzer. 2024. AI and memory wall. *IEEE Micro* (2024).

[29] Zhangxiaowen Gong, Houxiang Ji, Christopher W Fletcher, Christopher J Hughes, Sara Baghsorkhi, and Josep Torrellas. 2020. SAVE: Sparsity-aware vector engine for accelerating DNN training and inference on cpus. In *International Symposium on Microarchitecture (MICRO)*.

[30] Scott Grauer-Gray, William Killian, Robert Searles, and John Cavazos. 2013. Accelerating financial applications on the GPU. In *Proceedings of the Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU)*.

[31] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *Innovative parallel computing (InPar)*.

[32] Deniz Gurevin, Mohsin Shan, Shaoyi Huang, MD Amit Hasan, Caiwen Ding, and Omer Khan. 2024. PruneGNN: Algorithm-Architecture Pruning Framework for Graph Neural Network Acceleration. In *International Symposium on High-Performance Computer Architecture (HPCA)*.

[33] Anthony Gutierrez, Bradford M. Beckmann, Alexandru Dutu, Joseph Gross, Michael LeBeane, John Kalamatianos, Onur Kayiran, Matthew Poremba, Brandon Potter, Sooraj Puthoor, Matthew D. Sinclair, Mark Wyse, Jieming Yin, Xianwei Zhang, Akshay Jain, and Timothy G. Rogers. 2018. Lost in abstraction: Pitfalls of analyzing GPUs at the intermediate language level. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[34] Song Han, Jeff Pool, John Tran, and William J Dally. 2015. Learning both weights and connections for efficient neural networks. In *International Conference on Neural Information Processing Systems (NeurIPS)*.

[35] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*.

[36] Mafijul Md Islam and Per Stenstrom. 2009. Zero-value caches: Cancelling loads that return zero. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[37] Hyunjun Kim, Sungin Hong, Hyeonsu Lee, Euiseong Seo, and Hwansoo Han. 2019. Compiler-assisted GPU thread throttling for reduced cache contention. In *Proceedings of International Conference on Parallel Processing (ICPP)*.

[38] Gunjae Koo, Hyeran Jeon, Zhenhong Liu, Nam Sung Kim, and Murali Annavaram. 2018. CTA-aware prefetching and scheduling for GPU. In *International Parallel and Distributed Processing Symposium (IPDPS)*.

[39] Boris Krasnopolsky and Alexey Medvedev. 2016. Acceleration of large scale OpenFOAM simulations on distributed systems with multicore CPUs and GPUs. In *Parallel Computing: On the Road to Exascale*.

[40] Sheetal Lahabar and PJ Narayanan. 2009. Singular value decomposition on GPU using CUDA. In *International Parallel & Distributed Processing Symposium (IPDPS)*.

[41] William B Langdon, Brian Yee Hong Lam, Justyna Petke, and Mark Harman. 2015. Improving CUDA DNA analysis software with genetic programming. In *Genetic and Evolutionary Computation Conference (GECCO)*.

[42] Jaekyu Lee, Nagesh B Lakshminarayana, Hyesoon Kim, and Richard Vuduc. 2010. Many-thread aware prefetching mechanisms for GPGPU applications. In *International Symposium on Microarchitecture (MICRO)*.

[43] Yuanzhi Li and Yang Yuan. 2017. Convergence analysis of two-layer neural networks with ReLU activation. In *International Conference on Neural Information Processing Systems (NeurIPS)*.

[44] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason Mars. 2018. The architectural implications of autonomous driving: Constraints and acceleration. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[45] Changxi Liu, Yifan Sun, and Trevor E Carlson. 2023. Photon: A fine-grained sampled simulation methodology for GPU workloads. In *International Symposium on Microarchitecture (MICRO)*.

[46] Zhi-Gang Liu, Paul N Whatmough, Yuhao Zhu, and Matthew Mattina. 2022. S2TA: Exploiting structured sparsity for energy-efficient mobile cnn acceleration. In *International Symposium on High-Performance Computer Architecture (HPCA)*.

[47] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A formal analysis of the NVIDIA PTX memory consistency model. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[48] Mike Mantor. 2012. AMD Radeon™ HD 7970 with graphics core next (GCN) architecture. In *Hot Chips 24 Symposium (HCS)*.

[49] Xinxin Mei and Xiaowen Chu. 2016. Dissecting GPU memory hierarchy through microbenchmarking. *Parallel and Distributed Systems (TPDS)* (2016).

[50] Jaikrishnan Menon, Marc De Kruijf, and Karthikeyan Sankaralingam. 2012. iGPU: exception support and speculative execution on GPUs. In *International Symposium on Computer Architecture (ISCA)*.

[51] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2022. Pointer Sentinel Mixture Models. In *International Conference on Learning Representations (ICLR)*.

[52] Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger. 2015. Doppelgänger: a cache for approximate computing. In *International Symposium on Microarchitecture (MICRO)*.

[53] Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. 2021. Accelerating sparse deep neural networks. *arXiv preprint arXiv:2104.08378* (2021).

[54] Saba Mostofi, Hajar Falahati, Negin Mahani, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2023. Snake: A Variable-length Chain-based Prefetching Mechanism for GPUs. In *International Symposium on Microarchitecture (MICRO)*.

[55] OpenAI. 2023. GPT-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[56] Samuel Pakalapati and Biswabandan Panda. 2020. Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching. In *International Symposium on Computer Architecture (ISCA)*.

[57] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *International Symposium on Computer Architecture (ISCA)*.

[58] Eunhyeok Park, Dongyoung Kim, and Sungjoo Yoo. 2018. Energy-efficient neural network accelerator based on outlier-aware low-precision computation. In *International Symposium on Computer Architecture (ISCA)*.

[59] Hyunsun Park, Dongyoung Kim, Junwhan Ahn, and Sungjoo Yoo. 2016. Zero and data reuse-aware fast convolution for deep neural networks on GPU. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*.

[60] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Conference on Neural Information Processing Systems (NeurIPS)*.

[61] Ashutosh Pattnaik, Xulong Tang, Onur Kayiran, Adwait Jog, Asit Mishra, Mahmut T Kandemir, Anand Sivasubramaniam, and Chita R Das. 2019. Opportunistic computing in GPU architectures. In *International Symposium on Computer Architecture (ISCA)*.

[62] Arthur Perais. 2021. A Case for Speculative Strength Reduction. *Computer Architecture Letters (CAL)* (2021).

[63] Arthur Perais and André Seznec. 2014. Practical data value speculation for future high-end processors. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[64] Arthur Perais and André Seznec. 2015. BeBoP: A cost effective predictor infrastructure for superscalar value prediction. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[65] Stephen Pruett and Yale Patt. 2021. Branch runahead: An alternative to branch prediction for impossible to predict branches. In *International Symposium on Microarchitecture (MICRO)*.

[66] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. Zero-infinity: Breaking the GPU memory wall for extreme scale deep learning. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[67] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. 2019. Efficient invisible speculative execution through selective delay and value prediction. In *International Symposium on Computer Architecture (ISCA)*.

[68] Ankit Sethia, D Anoushe Jamshidi, and Scott Mahlke. 2015. MASCAR: Speeding up GPU warps by reducing memory pitstops. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[69] André Seznec. 2011. A new case for the tage branch predictor. In *International Symposium on Microarchitecture (MICRO)*.

[70] Maying Shen, Pavlo Molchanov, Hongxu Yin, and Jose M Alvarez. 2022. When to prune? a policy towards early structural pruning. In *Conference on Computer Vision and Pattern Recognition (CVPR)*.

[71] E Sprangle and D Carmean. 2002. Increasing processor performance by implementing deeper pipelines. In *International Symposium on Computer Architecture (ISCA)*.

[72] Mingjie Sun, Zhuang Liu, Anna Bair, and J. Zico Kolter. 2024. A Simple and Effective Pruning Approach for Large Language Models. In *International Conference on Learning Representations (ICLR)*.

[73] Yifan Sun, Trinayan Baruah, Saiful A. Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, Harrison Barclay, Amir Kavyan Ziabari, Zhongliang Chen, Rafael Ubal, José L. Abellán, John Kim, Ajay Joshi, and David R. Kaeli. 2019. MGPUSim: enabling multi-GPU performance modeling and optimization. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[74] Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter McCardwell, Alejandro Villegas, and David Kaeli. 2016. Hetero-mark, a benchmark suite for CPU-GPU collaborative computing. In *IEEE International Symposium on Workload Characterization (IISWC)*.

[75] Yifan Sun, Yixuan Zhang, Ali Mosallaei, Michael D Shah, Cody Dunne, and David Kaeli. 2021. Daisen: a framework for visualizing detailed GPU execution. In *Computer Graphics Forum*.

[76] Mohammad Khavari Tavana, Yifan Sun, Nicolas Bohm Agostini, and David Kaeli. 2019. Exploiting adaptive data compression to improve performance and energy-efficiency of compute workloads in multi-GPU systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[77] G Thomas-Collignon and V Mehta. 2020. Optimizing cuda applications for nvidia a100 GPU. In *NVIDIA GPU Technology Conference*.

[78] TOP500. 2025. TOP500 Supercomputer Sites. https://www.top500.org/.

[79] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *CoRR* (2023). arXiv:2302.13971

[80] Henk A Van der Vorst. 1992. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on scientific and Statistical Computing* (1992).

[81] Lu Wang, Magnus Jahre, Almutaz Adileho, and Lieven Eeckhout. 2020. MDM: The GPU memory divergence model. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[82] Yang Wang, Chen Zhang, Zhiqiang Xie, Cong Guo, Yunxin Liu, and Jingwen Leng. 2021. Dual-side sparse tensor core. In *International Symposium on Computer Architecture (ISCA)*.

[83] Yecheng Xiang and Hyoseung Kim. 2019. Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference. In *Real-Time Systems Symposium (RTSS)*.

[84] Xin You, Changxi Liu, Hailong Yang, Pengbo Wang, Zhongzhi Luan, and Depei Qian. 2022. Vectorizing spmv by exploiting dynamic regular patterns. In *International Conference on Parallel Processing (ICPP)*.

[85] Geng Yuan, Payman Behnam, Zhengang Li, Ali Shafiee, Sheng Lin, Xiaolong Ma, Hang Liu, Xuehai Qian, Mahdi Nazm Bojnordi, Yanzhi Wang, and Caiwen Ding. 2021. Forms: Fine-grained polarized reram-based in-situ computation for mixed-signal DNN accelerator. In *International Symposium on Computer Architecture (ISCA)*.

[86] Xia Zhao, Magnus Jahre, and Lieven Eeckhout. 2020. Selective replication in memory-side GPU caches. In *International Symposium on Microarchitecture (MICRO)*.

[87] Hongkuan Zhou, Ajitesh Srivastava, Hanqing Zeng, Rajgopal Kannan, and Viktor Prasanna. 2021. Accelerating large scale real-time GNN inference using channel pruning. *Proceedings of the VLDB Endowment* (2021).

[88] Keren Zhou, Yueming Hao, John Mellor-Crummey, Xiaozhu Meng, and Xu Liu. 2022. ValueExpert: Exploring value patterns in GPU-Accelerated applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[89] Michael Zhu and Suyog Gupta. 2017. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878* (2017).

[90] Michael Zhu and Suyog Gupta. 2017. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878* (2017).

[91] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. 2019. Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern GPUs. In *International Symposium on Microarchitecture (MICRO)*.