


Article

Hardware Design of DRAM Memory Prefetching Engine for General-Purpose GPUs

Freddy Gabbay ^{1,*} , Benjamin Salomon ², Idan Golan ³ and Dolev Shema ³¹ Institute of Electrical Engineering and Applied Physics, The Hebrew University of Jerusalem, Jerusalem 9190401, Israel² Engineering Faculty, Ruppin Academic Center, Emek Hefer 4025000, Israel; bennysal@ruppin.ac.il³ Electrical and Computer Engineering Faculty, Technion–Israel Institute of Technology, Haifa 3200003, Israel; igolan@nvidia.com (I.G.); dshema@nvidia.com (D.S.)

* Correspondence: freddy.gabbay@mail.huji.ac.il

Abstract

General-purpose graphics computing on processing units (GPGPUs) face significant performance limitations due to memory access latencies, particularly when traditional memory hierarchies and thread-switching mechanisms prove insufficient for complex access patterns in data-intensive applications such as machine learning (ML) and scientific computing. This paper presents a novel hardware design for a memory prefetching subsystem targeted at DDR (Double Data Rate) memory in GPGPU architectures. The proposed prefetching subsystem features a modular architecture comprising multiple parallel prefetching engines, each handling distinct memory address ranges with dedicated data buffers and adaptive stride detection algorithms that dynamically identify recurring memory access patterns. The design incorporates robust system integration features, including context flushing, watchdog timers, and flexible configuration interfaces, for runtime optimization. Comprehensive experimental validation using real-world workloads examined critical design parameters, including block sizes, prefetch outstanding limits, and throttling rates, across diverse memory access patterns. Results demonstrate significant performance improvements with average memory access latency reductions of up to 82% compared to no-prefetch baselines, and speedups in the range of 1.240–1.794. The proposed prefetching subsystem successfully enhances memory hierarchy efficiency and provides practical design guidelines for deployment in production GPGPU systems, establishing clear parameter optimization strategies for different workload characteristics.

Keywords: GPGPU; hardware prefetching; memory system; DDR; DRAM memory; parallel computing; high-performance computing (HPC)



Academic Editors: Joshua M. Pearce, Gerard Ghibaudo and Francis Balestra

Received: 31 August 2025

Revised: 28 September 2025

Accepted: 30 September 2025

Published: 8 October 2025

Citation: Gabbay, F.; Salomon, B.; Golan, I.; Shema, D. Hardware Design of DRAM Memory Prefetching Engine for General-Purpose GPUs. *Technologies* **2025**, *13*, 455. <https://doi.org/10.3390/technologies13100455>

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

General-purpose computing on graphics processing units (GPGPUs) are specialized hardware devices designed to perform highly parallel computation systems [1–9]. Their target is to accelerate a wide range of computational tasks, including ML, high-performance computing, scientific simulations, data analytics, and more. Such workloads are typically offloaded from CPUs to GPGPUs, which can provide significant performance improvements. From an energy and power perspective, GPGPUs are highly efficient computing engines that exploit massive data parallelism by utilizing identical processing elements capable of executing numerous threads simultaneously in a Single Instruction Multiple

Threads (SIMT) manner. Consequently, GPGPUs are exceptionally suitable for throughput-oriented tasks, as their parallel architecture not only leverages inherent workload data-level parallelism but can also effectively mitigate high memory latency.

GPGPUs have gained significant popularity in recent years due to their ability to accelerate ML applications [8–10], such as vision recognition, natural language processing, and video analytics. ML algorithms often involve large datasets and complex computations, which can be time-consuming to execute on traditional CPUs. GPGPUs, however, are well suited for ML applications that exhibit massive data parallelism and require extensive matrix and vector processing during both training and inference. As GPGPUs execute a massive number of threads, their performance is highly dependent on the memory system to provide sufficient bandwidth. To support high-bandwidth connectivity with the memory, GPGPUs incorporate specific architectural features, such as a multi-level memory hierarchy with several levels of cache, wide off-chip multiple Double Data Rate (DDR) DRAM memory channels, memory compression mechanisms, and high-bandwidth memory (HBM). Although GPGPUs can hide memory latencies through thread switching, there are several scenarios where this approach is insufficient. To further reduce memory latency, GPGPUs often employ memory prefetching techniques.

Memory prefetching [11,12] is a technique used in computer systems to improve memory access times by predicting and loading data into the cache or local memory before it is requested by the processor. There are several approaches to memory prefetching, including hardware-based prefetching, software-based prefetching, and hybrid approaches that combine hardware and software prefetching. Including memory prefetching offers several potential benefits in GPGPUs.

1. **Improve memory hierarchy efficiency.** GPGPUs incorporate large memory hierarchies, enabling a wider range of applications to be executed efficiently. However, relying solely on the memory hierarchy system to handle complex memory access patterns in GPGPU workloads has been shown to be inefficient [2,13]. Moreover, due to the large number of concurrently executing threads, it is highly challenging for GPGPUs to exploit data locality and utilize cache structures as effectively as general-purpose processors. Consequently, data movement between main memory and caches often results in contention and degraded performance [14–16]. Memory prefetching can detect memory access patterns and proactively fetch the required data into the cache or local memory before it is needed, thereby improving the effectiveness of the memory hierarchy.
2. **Improve data locality.** The large number of threads running on a GPGPU can significantly impact data locality. Due to the limited capacity of the memory system, as the number of concurrent threads increases, the pressure on memory resources rises. This leads to an increased cache miss ratio and restricts the footprint size of each thread in the on-chip memory. In such scenarios, memory prefetching can enhance memory locality and reduce the average memory access time.
3. **Reduce asynchronous task switching overhead.** Heterogeneous systems consisting of a CPU and a GPGPU often execute workloads divided into two kernels: a control kernel running on the CPU and a computational kernel with data parallelism running on the GPGPU. These workloads are structured such that the control kernel executes on the CPU first. Once completed, the workload is passed to the GPGPU for execution of the computational kernel. Upon completion, the results are passed back to the CPU. This asynchronous back-and-forth task switching between the CPU and GPGPU can continue until the workload is complete. During each task switch, it is necessary to reload the thread context into the GPGPU's local memory, which increases task switching overhead and reduces overall throughput. Memory prefetching can mitigate

this overhead by preloading necessary data into the GPGPU's local memory, thereby improving overall utilization.

4. **Improve core utilization.** GPGPUs execute a massive number of threads using fine-grained scheduling. Since threads are switched every clock cycle, any long-latency memory access or absence of required data in the local memory may stall threads, reducing effective core utilization. Memory prefetching can minimize these memory access latencies, thereby enhancing GPGPU utilization and overall performance.

The efficient handling of memory access patterns in GPGPUs is crucial for achieving high performance in data-intensive workloads such as ML, scientific simulations, and high-performance computing. While traditional memory hierarchies and thread-switching mechanisms are effective in mitigating memory latency, they often fall short when memory resources are limited. This paper proposes a novel hardware prefetching subsystem tailored for GPGPU architectures to address these challenges.

We introduce a DRAM-based memory prefetching subsystem for GPGPUs, primarily targeted at DDR memory via a standard Advanced eXtensible Interface (AXI) bus [17], with similar applicability to HBM. The proposed prefetching subsystem features a modular architecture comprising multiple parallel prefetching engines, each configured to handle distinct non-overlapping memory address ranges. Each engine maintains a self-learned context that dynamically adapts to the memory access patterns of its assigned range, enabling efficient detection and prediction of stride-based and spatially local memory accesses. The prefetch subsystem integrates a primary scheduler that routes memory transactions to the appropriate prefetch engine or directly to the memory controller when outside the prefetch engine's address range. Additionally, each prefetch engine is equipped with dedicated data buffers, queues, and a configurable stride detection mechanism that identifies and predicts recurring memory access patterns. To ensure robustness and adaptability, the design incorporates features such as context flushing for handling changes in access patterns, watchdog timers to recover from system inactivity or deadlocks, and a flexible configuration interface for fine-tuning operational parameters. By leveraging these architectural elements, the prefetch engine proactively fetches data blocks before they are requested by the GPGPU, significantly reducing memory latency and improving overall system throughput. This microarchitectural design not only enhances memory hierarchy efficiency but also optimizes parallelism by minimizing thread stalls caused by long-latency memory accesses.

To validate the effectiveness of our proposed prefetch subsystem design, we conducted an extensive experimental analysis using real-world workloads across diverse GPGPU applications. Our comprehensive evaluation framework examines multiple critical design parameters including block sizes (ranging from 32 bytes to 256 bytes), prefetch outstanding limits, and throttling rates to characterize the prefetching engine's performance under various operational conditions. The experimental analysis encompasses memory access pattern analysis, demonstrating latency improvements, with particular focus on applications with predictable access patterns such as convolutional neural networks (CNNs) and more complex workloads with irregular memory access behaviors. Our results demonstrate that the proposed prefetching subsystem can achieve significant memory access latency reductions of up to 82% compared to no-prefetch baselines, with speedups in the range of $1.240\text{--}1.794\times$. The experimental validation reveals critical insights about the relationship between prefetch configuration parameters and performance, showing that larger block sizes (256 bytes) provide superior performance for spatially local access patterns, while smaller block sizes with minimal outstanding requests prove optimal for stride-based access patterns. Additionally, our analysis demonstrates the importance of proper throttling rate selection, revealing that aggressive prefetching beyond optimal thresholds can de-

grade performance below baseline levels, emphasizing the need for adaptive configuration mechanisms in practical GPGPU prefetching implementations.

The main contributions of this paper are as follows:

1. **Modular DRAM-based Prefetching Engine.** We propose a specialized hardware prefetch subsystem for GPGPU architectures featuring multiple parallel engines with dedicated address ranges, self-learned context mechanisms, and adaptive stride detection for efficient handling of diverse memory access patterns.
2. **Robust System Integration Features.** We design comprehensive system-level features, including context flushing for handling access pattern changes, watchdog timers for deadlock recovery, and flexible configuration interfaces for runtime parameter optimization.
3. **Comprehensive Experimental Validation.** We conduct an extensive performance analysis using real-world workloads, demonstrating memory access latency improvements of up to 82% and providing detailed characterization of optimal configuration parameters, including block sizes, outstanding limits, and throttling rates.
4. **Design Guidelines and Performance Insights.** We provide critical insights into the relationship between prefetch configuration parameters and performance across different workload types, establishing guidelines for optimal prefetcher deployment in practical GPGPU systems.

The remainder of this paper is organized as follows: Section 2 provides the background and discusses related work on memory prefetching techniques. Section 3 introduces the design and microarchitecture of our proposed DRAM-based prefetch subsystem, detailing its key components and operational mechanisms. Section 4 describes the simulation environment used to evaluate the performance of the prefetch subsystem, including the workload setup and metrics used for analysis. Section 5 presents the results of our experimental analysis, highlighting the improvements in memory hierarchy efficiency, data locality, and system throughput. Section 6 discusses the trade-offs of prior prefetching approaches for GPGPUs and presents a comparison with our proposed scheme. Finally, Section 7 concludes the paper, summarizing the contributions and outlining potential directions for future research.

2. Background and Related Work

This section provides an overview of GPGPU architecture, DDR and HBM memories, and prior works related to prefetching in GPGPUs.

2.1. GPGPU Architecture

A GPGPU typically comprises several key components, as illustrated in Figure 1, which depicts the architecture of Nvidia's Fermi GPU [1–7,9]:

- Multiple processing cores that are organized into one or more “streaming multiprocessor” (SM) units as illustrated in Figure 2. Each SM unit is responsible for executing a set of instructions in parallel such that all processing cores at any given time execute the same instruction of different data elements. SM also consists of a large register file, an instruction cache, a data cache/scratchpad memory, a dispatch unit, and a thread scheduler.
- L2 cache/shared memory is located between the SMs and the external physical memory. The shared memory is used to store shared data between all SMs.
- Multi-channel off-chip physical memory (DRAM) is typically slower than the other types of memory in the GPGPU but is much larger and can be used to store larger amounts of data. GPGPUs typically employ multiple memory controllers which provide multiple simultaneous access channels to the physical memory to satisfy the high bandwidth requirements.

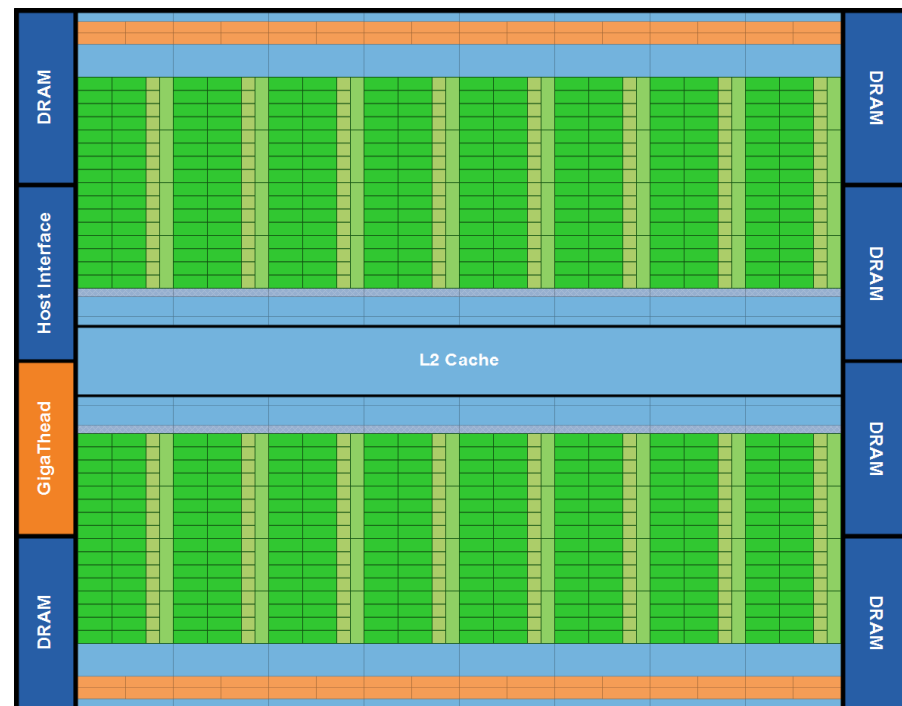


Figure 1. Fermi GPU architecture with 16 streaming multiprocessors (source: Nvidia [1]).

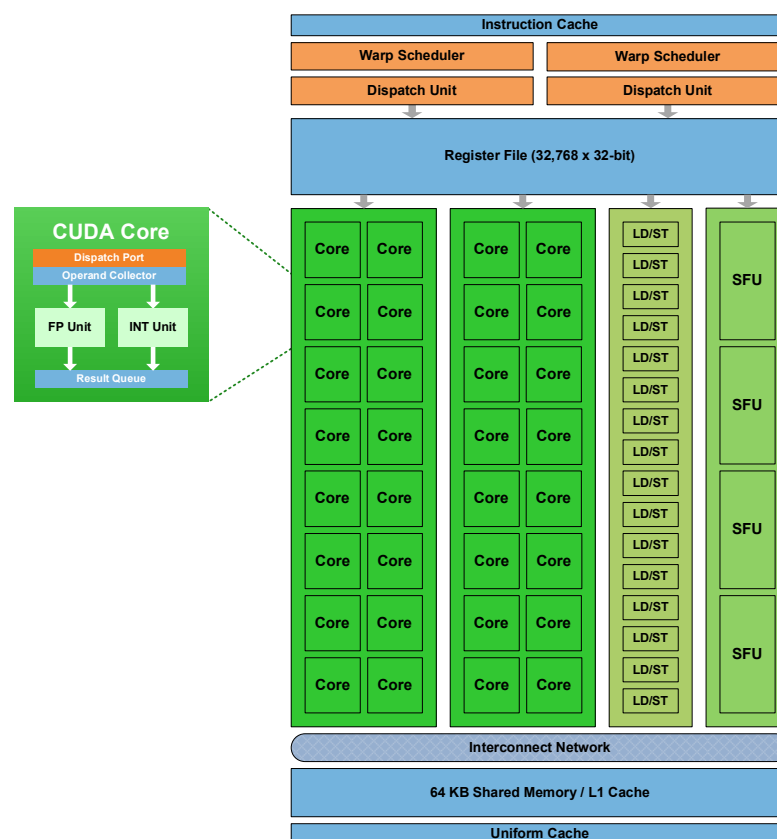


Figure 2. Fermi GPU streaming multiprocessor architecture (source: Nvidia [1]).

GPGPUs employ hierarchical thread scheduling mechanism [1]. A workload as-signed to the GPGPU, often referred to as a kernel, is composed of multiple blocks. Each block consists of multiple threads that are assigned to a single SM. The SM executes the threads within a block in a fine-grain scheduling manner. The SM scheduler divides the threads

within the block into groups of warps. In addition, the SM scheduler retrieves a single instruction from all the threads in a warp to be executed simultaneously. This process is repeated every clock cycle for each warp. Once the execution of an instruction is completed for all threads of the block, the thread scheduler proceeds to execute the next instruction in a similar fashion.

2.2. DDR Memory and HBM

DDR DRAM [18] has long been the primary memory technology in GPUs and CPUs, offering high density and relatively low cost. DDR achieves higher throughput than single-data-rate memories by transferring data on both the rising and falling edges of the clock signal. Its operation relies on multi-bank architectures and burst transfers, where rows are activated, data is read or written sequentially, and precharge operations restore the bank state. DDR modules are typically organized in channels and ranks, each with an independent command, address, and data interface, enabling parallelism at the system level. While DDR provides scalability and flexibility, its bandwidth is limited by off-chip signaling constraints and relatively narrow interfaces.

HBM [19], in contrast, is designed to overcome these bandwidth limitations by using 3D-stacked DRAM dies connected through through-silicon vias (TSVs). HBM integrates wide interfaces (typically 1024 bits per stack) directly adjacent to the GPU die via an interposer, drastically reducing signal distance and increasing bandwidth per watt. The principle of operation is similar to DDR in terms of bank activation and burst transfers, but HBM's massively parallel interface and physical integration enable much higher sustained throughput. Although this work primarily targets DDR-based memory systems, the same prefetching concepts can be extended to HBM, since the fundamental principles of bank-level access, burst transactions, and row-buffer locality remain applicable.

2.3. Prior Works

Extensive research has been conducted on prefetching methods. Generally, there are three major kinds of prefetching methods: software-based prefetching, hardware-based prefetching, and combined hardware and software prefetching.

Software-based prefetching methods typically rely on compiler algorithms to insert prefetching instructions [20–24]. A compiler algorithm for inserting prefetching instructions into codes that operate on dense matrices was introduced in [21]. Data references that are likely to be cache misses are identified by the compiler algorithm, and prefetches are issued only for them. An improved compiler algorithm, Resource-Aware Prefetching (RAP), was proposed in [22]. The RAP algorithm makes use of limited resources more efficiently. The number of miss information/status holding register entries is used to control the prefetch distance. Effective implementation of software prefetching can be challenging in real applications [23]. The reduced latency must be significant enough to compensate for the issue of prefetching instructions and the generation of addresses. An interesting use case of software prefetching in computational fluid dynamic modeling was demonstrated in [24]. The authors discuss the importance of auto-tuning for determining the optimal prefetch distances and the right prefetch destinations.

A hardware-based prefetching approach has been implemented in various types of processors and includes sequential prefetching, stride prefetching, and global history buffer (GHB) prefetching methods.

Sequential prefetching is based on spatial locality and assumes an access pattern in a specific direction. The spatial locality of the pattern is used to prefetch consecutive blocks in the same direction. One block look ahead (OBL), also known as next line prefetching, initiates a prefetch for cache line $b + 1$ when cache line b is accessed. The number of

prefetched cache lines can be increased from one to K , that is, prefetching cache lines $b + 1, \dots, b + K$ when cache line b is accessed. In stream prefetching, the prefetched data are stored in stream buffers and not in the cache, thus avoiding cache pollution. In refs. [25–27] the stream buffer is used to aid the cache, prefetching cache lines one after another in a certain pattern. The prefetched cache line will be presented to the cache in case of a missed cache.

Stride prefetcher aims to determine patterns in the delta between the addresses of the memory accesses. Once a pattern is detected, the prefetcher fetches one or more blocks according to the pattern. In [28], a separate cache called the Reference Prediction Table contains previous reference addresses and recent strides. The Reference Prediction Table is indexed by a look-ahead program counter (ahead of the real program counter), which is used to control the prefetches. In [29], a stride prediction table is used to determine the distances between strides made within the loop body of a numerical program.

The GHB prefetcher stores all the global addresses of the missed accesses in a FIFO buffer [30]. These addresses are placed at the bottom of a table and removed from the top. Linked lists within the buffer maintain GHB history information. GHB prefetcher can be used in conjunction with sequential prefetching or stride prefetching. Elements of software and hardware prefetching can be combined. For example, the number of prefetched cache lines K for each access may be calculated by the compiler and passed to the hardware prefetcher.

Prefetching mechanisms for GPGPU have been proposed in several studies. Most of them are variations of the described methods. However, straightforward implementations of CPU-like prefetching mechanisms may not improve performance [31]. A GPGPU compiler for memory optimization and parallelism management was proposed in [32]. Memory accesses are analyzed by the compiler and a temporary variable is used to prefetch data. GPGPU memory usage is improved, and workload is distributed in threads and thread blocks. Thread-level and data-level parallelism are assumed to be observed by application developers and the optimized compiler is used to optimize memory and parallelism performance. One of the challenges of the software-based prefetching approach is the limited capability of the compiler to predict prefetched memory addresses at compile time, especially for applications with irregular or dynamic memory access patterns. This limitation can result in suboptimal prefetching decisions, increasing memory utilization and may result in cache pollution.

A next-line prefetching technique, in conjunction with a warp scheduling technique for improving GPGPU performance and the efficiency of the prefetching, was proposed in [33]. The prefetching mechanism with the aware scheduling technique aims to improve L1 hit rates in GPGPUs and to improve DRAM row locality. However, this prefetching method may provide only a small improvement in performance [34].

Stride prefetching methods for GPGPUs may be categorized into intra-warp stride prefetching and inter-warp stride prefetching. Intra-warp stride prefetching is based on targeting future load instructions in the same warp. Intra-warp stride prefetching mechanisms for graph algorithms on GPGPUs were proposed in [35]. Target loads detected by the hardware data are prefetched into unallocated registers that are not being used by other active threads. Inter-warp prefetching mechanisms are based on the detection of stride patterns and base addresses in different warps. A many-thread-aware prefetching mechanism for GPGPU applications based on inter-stride prefetching was proposed in [31]. Cooperative thread array (CTA)-aware prefetcher and scheduler based on using a leading warp to compute base addresses of CTAs was proposed in [34,36]. Adaptive inter-stride prefetching mechanisms on GPGPUs for reducing power consumption were proposed

in [37]. Inter-warp prefetching combined with orchestrated scheduling mechanisms for GPGPUs were proposed in [38].

In [39], a prefetching mechanism for L1 caches was proposed based on using a dedicated pattern descriptor specification. Warp-aware selective prefetching based on the dynamic selection of slow progress warps was proposed in [40]. Adaptive prefetching and scheduling architecture (APRES) based on a dynamic L1 prefetch and data cache partitioning was proposed in [41]. In APRES, high locality and patterns of stride accesses across warps are used to increase cache hits. Latency-tolerant register file architecture (LTRF) based on hardware and software cooperative register prefetching was proposed in [42]. LTRF is based on a two-level hierarchical register structure where the estimated working set is being prefetched to a register cache.

Typical prefetching mechanisms may not be highly efficient when irregular memory accesses are required, e.g., in graph analytics problems. In [43], a data-structure-aware prefetching (DSAP) mechanism was proposed in order to accelerate the breadth-first search algorithm. Adaptive fine-grain prefetching management is used to increase the efficiency of the prefetching. Finally, ML-based prefetchers have gained a lot of interest recently. ML-driven prefetchers learn memory access patterns and prefetching can be modeled as a classification problem for sequence prediction. For attempts in this direction see, e.g., [44] and references therein.

3. GPGPU Prefetcher Design

In this section, we describe the microarchitecture of our proposed GPGPU data prefetcher and provide the details on the design implementation.

3.1. Microarchitectural Overview

The system-level overview of our proposed GPGPU data prefetcher is illustrated in Figure 3. The data prefetcher is located between the GPGPU memory bus and the DRAM controller. The GPGPU is connected to the data prefetcher through the standard Advanced eXtensible Interface (AXI) 4.0 [17] primary bus, where the GPGPU issues memory access transactions to the DRAM physical memory. The data prefetcher module is connected to a DRAM memory controller through a subsidiary AXI bus. On the primary AXI bus, the GPGPU is the bus master, while the prefetcher is the bus slave, and in the subsidiary AXI bus, the prefetcher acts as a master while the DRAM memory controller acts a slave. Table 1 lists the main AXI channel signals, including their bit-widths, directions, and functional roles in read and write transactions. The prefetcher maintains a set of control registers which are initialized by the GPGPU to configure the prefetcher operation and will be described later. The CPU, DRAM controller, and the DRAM physical layer (PHY) are illustrated in Figure 3 for the sake of clarity of the overall system and are out of the scope of this work.

A detailed microarchitectural scheme of the prefetcher is depicted in Figure 4. The prefetcher subsystem consists of a set of parallel prefetch engines (the number of engines is a design parameter that can be set at the design stage). Every prefetch engine maintains the following: 1. a self-learned context which is distinctly configured to handle a specific non-overlapping memory address range, and 2. data buffers and queues. A prefetch engine can handle only transactions that fall into the prefetch engines' address range, while transactions that fall out of all the prefetch engines' address ranges are routed directly to the subordinate AXI bus. The usage of multiple prefetch engines allows for handling multiple different working sets and data streams within a single or multiple threads. Memory accesses issued by the GPGPU are scheduled by the prefetch subsystem's primary scheduler and are assigned to a prefetch engine. The primary scheduler also forwards

all other accesses that do not fall into the memory regions of the prefetch engines to the subsidiary AXI bus. The primary scheduler also handles all the returning access responses from the memory system (e.g., read responses) or responses from the prefetch engines to the GPGPU. All DRAM memory accesses that are initiated by the prefetch engines are handled by the prefetch subsystem's subsidiary scheduler and are forwarded to the AXI subsidiary bus. The subsidiary scheduler also forwards all the returning responses from the DRAM memory controller to the corresponding prefetch engines or the primary scheduler (when the response is not related to any of the prefetch engines).

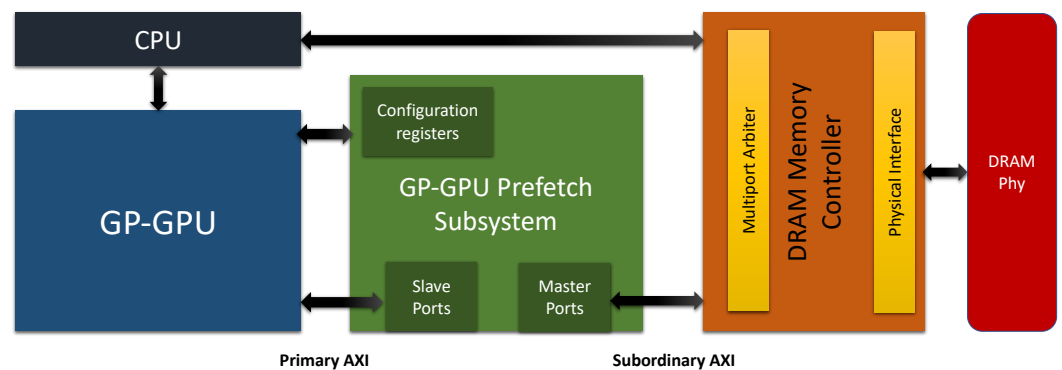


Figure 3. The GPGPU prefetcher system-level block diagram.

Table 1. AXI channel signal summary.

Data Field	Number of Bits	Direction (m/s)	Description
AXI Read Request Channel			
valid	1	Output/Input	Read request valid
ready	1	Input/Output	Read request ready
len	8	Output/Input	Burst length
addr	64	Output/Input	Read address
id	7	Output/Input	Read request transaction ID
AXI Read Data Channel			
valid	1	Input/Output	Read data valid
ready	1	Output/Input	Read data ready
last	1	Input/Output	Last read data indicator
data	256	Input/Output	Read data
id	7	Input/Output	Read response transaction ID
AXI Write Request Channel			
valid	1	Output/Input	Write request valid
ready	1	Input/Output	Write request ready
len	8	Output/Input	Burst length
addr	64	Output/Input	Write address
id	7	Output/Input	Write request transaction ID

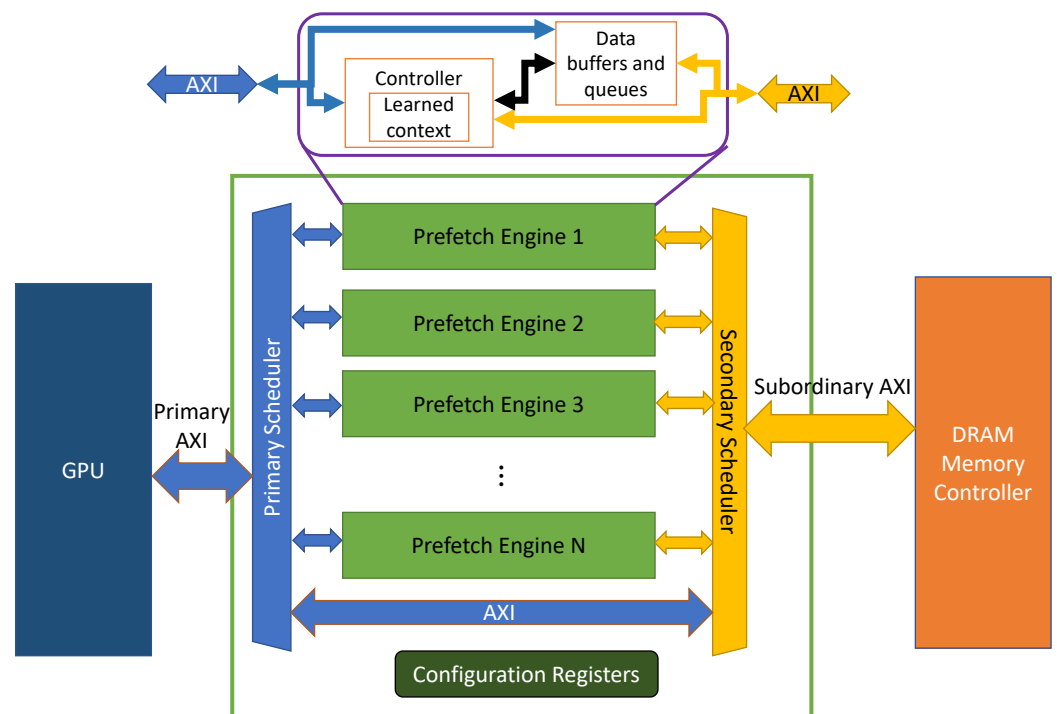


Figure 4. The GPGPU prefetch subsystem high-level block diagram.

The microarchitecture of the prefetch engine is depicted in Figure 5. The prefetch engine comprises two modules: a controller, which is responsible for managing the prefetch engine operation, and data buffers and queues that host the prefetched data blocks. The prefetch engine has the following features:

1. Detection and learning capability of relevant transactions. Based on the arbitration configuration policy (which will be described later) and the prefetcher engine context, each transaction is either claimed by its dedicated prefetch engine that is assumed to take the ownership for handling the transaction or forwarded to the DRAM controller.
2. Stride access detection and prediction. The prefetch engine can exploit both spatial and temporal locality of memory accesses on the primary AXI bus. Repeated accesses on the primary AXI bus (temporal locality) can leverage fast access to data stored in the prefetch engine data buffers. Additionally, memory accesses with a fixed address stride are learned by the prefetch engine resulting in prefetching data blocks with a corresponding stride.
3. Self-learned context, which maintains the state of the engine during the prefetch operation. The context consists of the following elements:
 - (a) AXI transaction ID (AXI ID): Specifying the transaction ID field on AXI bus for the prefetch transactions. The AXI ID is used to avoid ordering violations and guarantee consistency and coherency of all transactions within the same AXI ID.
 - (b) AXI address: Identifying the next address to be prefetched from the memory system.
 - (c) AXI burst length: Denoting the number of beats required for a burst operation. Typically, consecutive memory accesses are assumed to have the same burst length and size.
 - (d) Stride: Specifies the learned distance between two consecutive memory accesses issued on the primary AXI bus that are within the memory region of the prefetch engine.
4. Flush mechanism: When a new transaction mapped to the prefetch engine misses the learned context, i.e., the requested data are not available in the data buffer or when

the stride is changed, the prefetch engine will flush its learned context and restart the learning process.

5. Watchdog timer: Upon detection of a lack of activity either on the primary AXI bus or the subordinate AXI bus (due to a potential system failure or deadlocks), a designated watchdog mechanism will flush the learned context of the prefetcher and allow restarting to learn a new context.
6. Configurability: Allowing users to configure prefetch engine functionality, which will be further described.

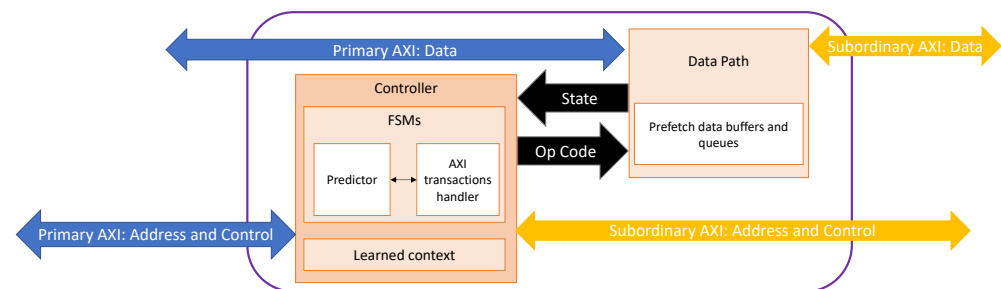


Figure 5. Prefetch engine microarchitecture.

The design implementation details of the GPGPU prefetch engines and subsystem are described in the next subsections.

3.2. Prefetch Engine Microarchitecture

The prefetch engine is connected to both the primary and subordinate AXI buses. Each prefetch engine AXI interface includes three channels: read address, read data, and write address. The AXI write data and write response channels bypass the prefetch engines and are forwarded from the GPGPU interface directly to the memory interface. Upon every memory transaction initiated by the GPGPU, the prefetch engine acts as follows:

1. Read or write memory accesses that are outside the address range of the prefetch engines are forwarded directly to the DRAM controller.
2. Read memory accesses that hit the address window of the prefetch engine are claimed by the prefetch engines, which take ownership generating a read response to the GPGPU.
3. Write memory accesses within the address range of the data stored in the prefetch buffer will result in flushing the learned context. The data prefetched by the engine will be disposed, outbound prefetch requests will be canceled, and the prefetch engine will initiate the learning process anew.

The prefetch engine controller and its data path, illustrated in Figure 5, operate concurrently. The controller is responsible for managing the AXI channels connected to the prefetch engines using two FSMs: The first FSM, denoted as a “Predictor” in Figure 5, manages the prefetch context, handles the learning process, and predicts the next prefetch transactions. The second FSM is the AXI transaction’s handler, which is responsible for managing inbound and outbound memory transactions in accordance with the AXI protocol. The data plane module consists of a prefetch queue that holds the pending prefetch requests and data blocks prefetched from the memory system. Both control and data planes communicate bidirectionally. The data plane exposes its internal state to the control plane, while the controller manages the data plane using a set of designated command through Opcodes (operation codes) that will be described in Section 3.4.

3.3. Controller Microarchitecture

The controller includes the predictor FSM and the AXI transaction’s handler. The predictor FSM consists of four states required to learn stride-based patterns, predicting the

next block address, and issuing the prefetch read access based on the learned pattern. The following are the predictor states:

- **IDLE.** An initial state. Upon the first read request, that falls into the prefetch engine address range, the predictor stores the AXI ID and AXI burst length, and transitions to the ARM state.
- **ARM.** Upon a read request, that falls into the prefetch engine address range, if the AXI ID and burst length match those learned in the IDLE state, then 1. the stride will be stored in the prefetch engine context as the distance from the current address to the address stored in the context, and 2. the predictor will transition into an ACTIVE state. In case of mismatch, the predictor will transition to a CLEANUP state.
- **ACTIVE.** On every cycle, the predictor will issue consecutive prefetch requests if the following conditions apply: 1. The number of outstanding prefetch requests have not reached the configured limit (specified in the configuration registers which will be described later). 2. The prefetch queue in the data plane has sufficient storage space for the next prefetch request. 3. The predicted prefetch address falls within the configured address range. In case of a new read request that mismatches the learned context, the controller will transition to the CLEANUP state.
- **CLEANUP.** This state can be reached by the conditions that have been described previously, or by a special signal indication from the arbiter. It prevents receiving new requests from the AXI initiator until the completion of all outstanding requests, to ensure a safe flush.

The AXI transaction handler (illustrated in Figure 5) executes memory transactions in accordance with the AXI bus protocol and the state of the data path (described in Section 3.4). When a transaction is initiated, the handler asserts the required AXI bus signals and sends the necessary indications to the Predictor and the data path through the designated Opcode. The handler prioritizes the handling of transactions in the following order:

- Valid read request from the AXI initiator. When a new transaction is initiated on the primary AXI bus, the handler acknowledges the initiator's request and assigns a new request to the data path. In the case of a read request falling into the prefetch window that has not been prefetched yet, it issues a new read request to the subordinate AXI bus.
- Data path responses. When new read responses have been prefetched to the data path, the handler dispatches the data from the data path to the AXI initiator.
- Subordinate AXI bus pending responses. When the subordinate AXI bus initiates a response, the handler acknowledges the subordinate bus response and places the data responses into the data path buffers.
- Ready address prediction. When a new address prediction is ready, the handler initiates a new prefetch transaction to the subordinate AXI bus.

In addition to initiating data transfer events, the prefetch engine also follows the AXI protocol sequence to communicate with the memory controller (DDR or HBM2). When a prefetch request is triggered, the prefetch subsystem asserts the `valid` signal on the AXI read request channel while providing the burst length (`len`), starting address (`addr`), and transaction identifier (`id`). The memory controller responds by asserting `ready`, completing the handshake and accepting the request. After the request is issued, the memory controller returns data over the AXI read data channel. For each beat of data, the memory controller asserts `valid` while driving the data and associated `id` fields. The prefetch subsystem responds with `ready`, ensuring proper flow control. The final beat of a burst is indicated by the `last` signal. The prefetch engine buffers the received data into its internal buffer until the demand access from the GPGPU core consumes it. Write requests follow a similar

handshake, where the prefetcher forwards a GPGPU request by issuing the *addr*, *len*, and *id* signals on the write request channel, followed by data transfers on the write data channel, each synchronized with *valid*/*ready* handshakes. The completion of the write burst is confirmed through the write response channel with the corresponding transaction *id*. Figure 6 illustrates the communication sequence for a typical read transaction.

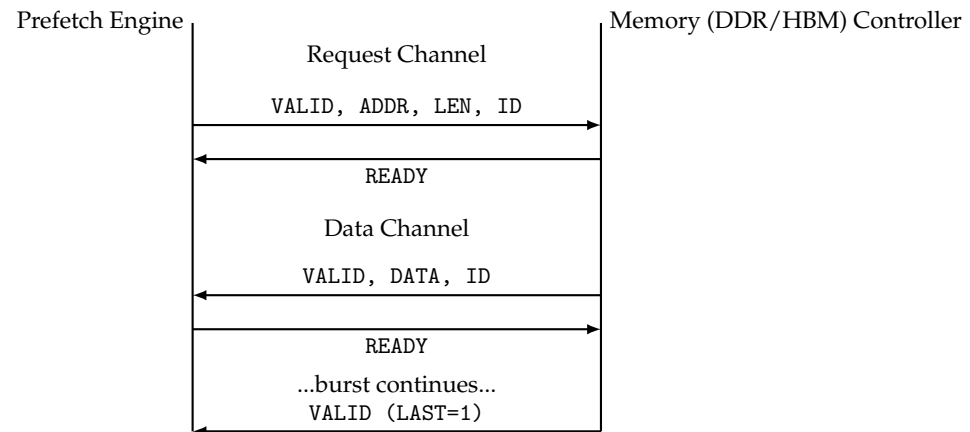


Figure 6. AXI read communication sequence between the prefetch engine and memory controller.

3.4. Data Path

The data path incorporates a prefetch data queue composed of data containers. Each data container serves as a storage unit for prefetched data within the queue, supporting a scalable number of data blocks and configurable block sizes, both defined by design parameters. Additionally, a data container is capable of handling burst transfers, which involve sequentially binding multiple AXI data beats. The queue of data containers is managed using three pointers:

1. The head-of-queue pointer, which identifies the block at the front of the queue;
2. The tail pointer, which indicates the next available block to be populated;
3. The burst offset pointer, which tracks the position of the next AXI beat within an ongoing burst transfer.

The data container is controlled by the controller using a set of Opcodes, which define specific operations for managing container blocks. These Opcodes enable the following functionalities:

1. **Reserve.** Allocates a container block in the queue for a new prefetch request.
2. **Read.** Retrieves data from a container block for transfer to the primary AXI bus initiator.
3. **Write.** Stores incoming data from the subordinate into a previously reserved block within the queue. The controller provides the burst length information from the prefetch engine context as part of this command.
4. **Flush.** Used by the controller to clear the container data when transitioning to CLEANUP state.

The data path communicates its status to the controller through the following signaling mechanisms:

- **Queue Almost Full.** Signals that the container queue has reached a predefined “almost full” threshold. This notification informs the controller about the queue’s occupancy level, enabling proactive management of incoming data.
- **Prefetch Request Count.** Provides the controller with the total number of prefetched requests currently stored in the container queue. This count allows the controller to enforce the configured limit on outstanding prefetch requests.
- **Outstanding Request.** Indicates the presence of at least one pending request in the queue. During the controller’s CLEANUP state, this signal ensures that the controller

refrains from initiating a flush operation until all outstanding requests have been fully processed.

3.5. Configurable Features

The prefetch engines and subsystem are equipped with dedicated configuration registers. The registers are exposed as memory-mapped configuration registers, allowing them to be programmed at runtime. They can be updated either by the external CPU, through the GPGPU's configuration space mapped into the CPU's address space, or by kernel code executed on the GPGPU if the configuration is embedded in the launch parameters. While the number and structure of these registers are fixed at the design stage, their contents can be reprogrammed dynamically, ensuring flexibility across different workloads. The configuration registers comprise the following:

- **Address Range Configuration (Bar and Limit).** Specifies the memory address range for each prefetch engine. Only read requests within the defined address range are processed by the prefetch engine, ensuring efficient handling of memory operations.
- **Prefetch Engine Performance Settings.**
 - Outstanding Prefetch Requests. Sets an upper limit on the number of outstanding prefetch requests. This limit prevents oversubscription of a prefetch engine and ensures fair resource allocation, avoiding starvation of other engines.
 - Bandwidth Throttling. Defines the maximum rate of prefetch requests. This configuration enables control over the bandwidth consumption of individual prefetch engines, optimizing system performance.
- **Almost Full Threshold.** Establishes the threshold value for prefetch buffer occupancy, signaling when the buffer is nearing its capacity.
- **Watchdog Timer.** Specifies the time limit for the watchdog timer, which will be elaborated upon in subsequent sections.

3.6. Prefetch Engine Flow of Operation

The complete flow of the prefetch engine is summarized in the flow diagram illustrated in Figure 7. The flow diagram shows that the prefetch engine's learning process is triggered by a read request that falls into the memory window. In this case, the prefetch engine moves from IDLE state to ARM state and learns the AXI ID, burst length, and start address of the memory access. In the ARM state, the prefetch engine waits for the next request that falls into the memory window. In the case of a write request or read request that does not match the AXI ID or burst length, the prefetch engine moves to CLEANUP state and flushes the learned context. Otherwise (when a read request matches AXI ID and burst length), the prefetch engine learns the address stride of the memory access. The stride is calculated by subtracting the new memory access address from the previous one. The prefetch engine then transitions from ARM to ACTIVE state. As long as the prefetch engine receives read requests that fall into the configured memory window and match the burst length and AXI ID, it remains in the ACTIVE state. In addition, in the ACTIVE state, the prefetch engine continuously performs prefetching of the learned context as long as it does not exceed the number of outstanding prefetch requests and bandwidth throttling limit described previously. In the case of write access or any access that does not match the AXI ID or burst length (assuming it falls into the prefetch engine memory windows), the prefetch engine transitions into the CLEANUP state and flushes the learned context. When the context is flushed, the prefetch engine moves back to the IDLE state and starts the learning process anew.

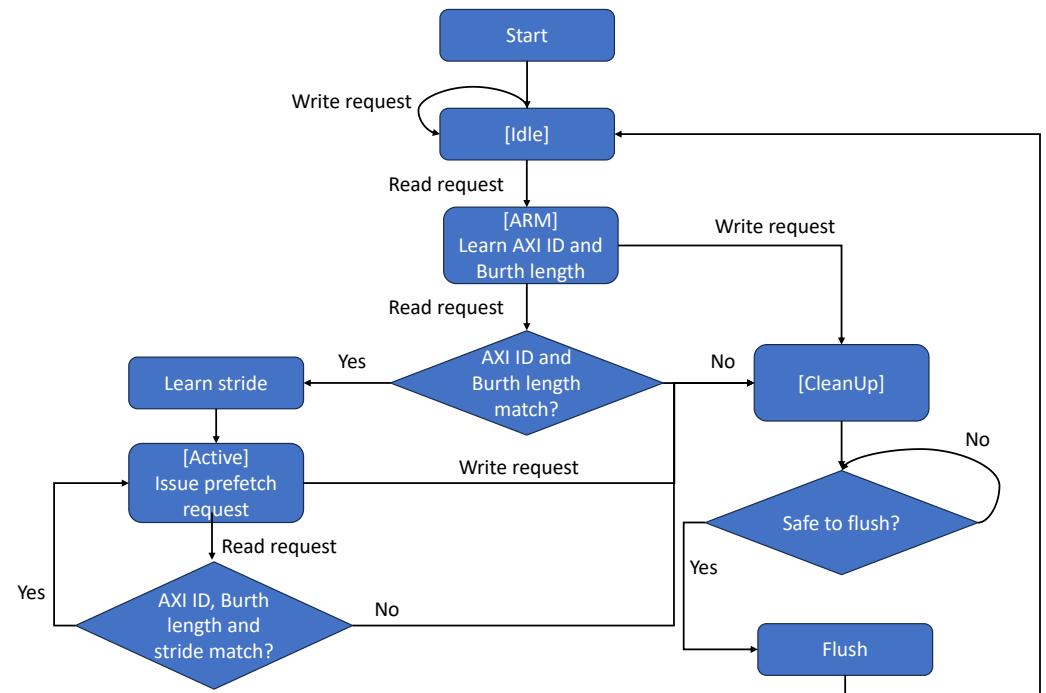


Figure 7. Prefetch engine flow diagram. The state of the predictor FSM is indicated by square brackets.

To illustrate the operational flow of the prefetch engine, consider the following example. Assume the prefetch engine is configured with an address range of 0x1000–0x2000 and its container includes 16 blocks. The following is the sequence of memory accesses performed by the GPGPU.

1. Read request. Address: 0x1000, burst length: 3, AXI ID: 10.
 - The prefetch engines store the address, AXI ID, and burst length in prefetch context.
 - The prefetch engine transitions into the 'ARM' state.
2. Read request. Address: 0x1004, burst length: 3, AXI ID: 10.
 - Since the access address falls into configured window and matches the learned burst length and AXI ID, the prefetch engine calculate and store the stride in the prefetch context which consists of the following learned context: stride: 0x4, burst length: 3, AXI ID: 10.
 - The prefetch engine transitions into the 'ACTIVE' state.
3. The prefetch engine issues a prefetch request to fetch the next predicted address: 0x1008, burst length: 3, AXI ID: 10. Once the data of the prefetch request is received from memory, the prefetch engine stores the data in the container.
4. The GPGPU issues a read request on address: 0x1008, burst length: 3, AXI ID: 10. Since the data has already been prefetched, the data is forwarded immediately to the GPGPU.
5. The GPGPU performs a read request of address: 0x1100, burst length: 3, AXI ID: 10.
 - The request has a mismatch with the learned stride in the prefetch engine.
 - The prefetch engine moves to CLEANUP state and forwards the request to the subsidiary AXI bus.
6. When there are no outstanding requests, the prefetch engine flushes the learned context and data container and transitions to IDLE state.

4. Simulation Environment

We conducted an experimental performance analysis of our prefetching subsystem within an integrated testbench simulation environment developed in SystemVerilog. Our

environment, illustrated in Figure 8, includes the prefetch subsystem, along with behavioral stubs representing the GPGPU, DDR memory, and CPU. The CPU stub serves as the central orchestrator, managing control over the entire system. The simulation environment executes a variety of GPGPU workloads, which will be detailed in subsequent sections.

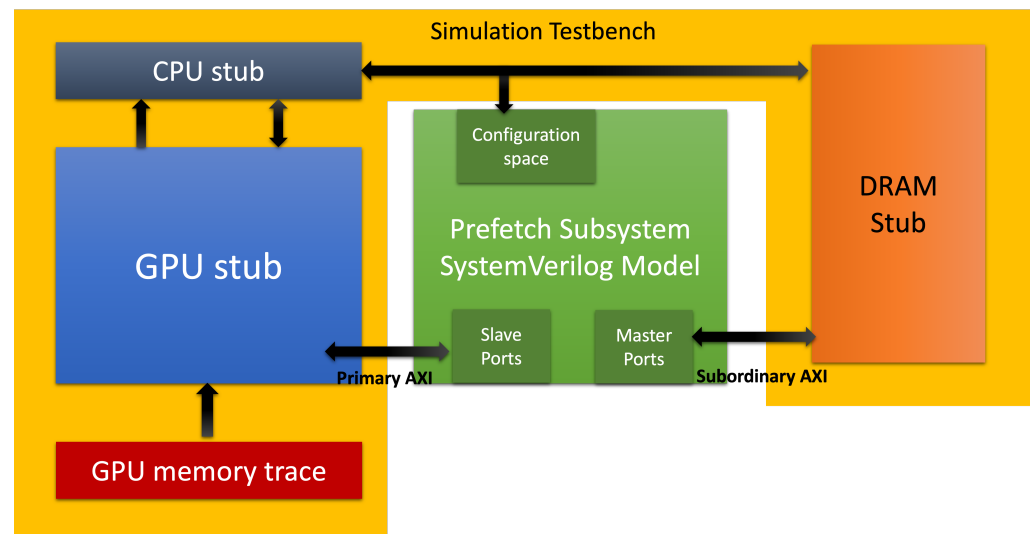


Figure 8. Prefetch engine simulation environment.

4.1. DRAM Memory Stub

The DRAM memory stub, illustrated in Figure 9, is assumed to model either DDR memory operation under the following assumptions.

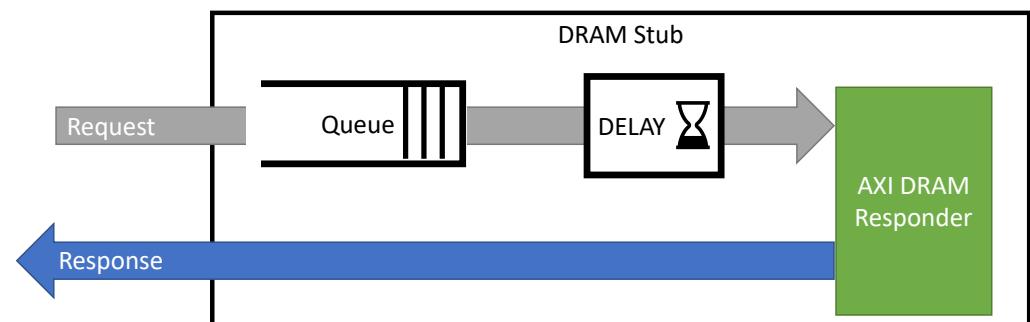


Figure 9. DRAM memory stub.

- Every prefetch engine is associated with a single DRAM memory window.
- The DRAM memory stub maintains a single queue for all memory requests serving in a “First-Come First-Serve” (FCFS) policy.
- The DRAM memory controller page policy assumes an open-page policy, i.e., last accessed memory page is kept open. This allows accesses that fall into the open page to be accessed faster (fast page mode).

All read requests are enqueued into a First-Come, First-Served (FCFS) queue. After being dequeued, the requests pass through a delay module that emulates DRAM memory latency. The latency is determined based on whether the memory access targets an open page or a closed page. An open-source AXI DRAM Responder module [45] is utilized to manage the responses to read requests. In contrast, all write requests are treated as posted transactions, with the DRAM stub immediately acknowledging them upon receipt.

4.2. GPGPU and CPU Stubs

The GPGPU stub processes memory access traces generated by the gpgpu-sim simulator [46]. The gpgpu-sim is a cycle-accurate GPU simulator that includes a functional model which simulates the parallel thread instruction execution on the GPGPU cores. The simulator can run CUDA or OpenCL computing workloads. The stub reads memory access records from the trace file and generates transactions on the AXI bus while adhering to the time intervals specified in the trace file. It comprises two key SystemVerilog modules:

- **Trace Module.** Responsible for reading the gpgpu-sim trace file, parsing the memory access records, and supplying the transactions.
- **AXI Initiator Module.** Converts the transactions provided by the trace module into AXI-compliant transactions and initiates them on the primary AXI bus.

The CPU stub is a SystemVerilog testbench module, which is responsible to configure the prefetch subsystem, and GPGPU and DRAM memory stubs.

4.3. Complete Simulation Flow

The simulation framework is designed to evaluate the performance of the prefetching subsystem within a SystemVerilog-based testbench environment. Figure 10 illustrates the overall simulation flow of our framework. A CUDA application is executed on gpgpu-sim, which generates memory access traces containing cycle-accurate address information. These traces are then processed by the Tracer module to drive AXI transactions into the prefetching subsystem. The subsystem communicates with the memory controller and DRAM stub, while the Testbench Monitors capture latency, efficiency, and performance metrics for analysis. This representation highlights the interaction between simulation components and the way the prefetching subsystem is evaluated. The flow, depicted in Figure 10, comprises several interconnected components.

- **Trace Generation ①.** Memory access traces are generated using gpgpu-sim executing CUDA-based applications. The traces contain memory access records, including cycle numbers and addresses.
- **Tracer Module ②.** The Tracer module reads the gpgpu-sim trace files and processes the memory access records. It ensures that transactions are generated on the AXI bus while adhering to the timing constraints specified in the trace file.
- **Prefetching Engine:** The prefetcher is the core component under evaluation. It interacts with two sets of AXI ports.
 - **Slave Ports ③.** Receive memory transactions from the Tracer module.
 - **Master Ports ④.** Issue prefetching requests to the DRAM memory controller and DRAM stub.

Additionally, the Configuration Space AXI Port is managed by the CPU stub, which orchestrates the simulation and configures the prefetcher.

- **Memory Controller and DRAM Stub ⑤:** The Memory Controller and DRAM stub emulate DRAM memory behavior, including latency modeling for open and closed page accesses. This stub responds to read and write requests generated by the prefetcher and Tracer modules.
- **Testbench Monitors ⑥.** Testbench Monitors collect statistics during the simulation, including memory latency, prefetching efficiency (defined as the fraction of memory accesses served by useful prefetches, as reflected in the reduced-latency bins of the histograms), and overall system performance. These metrics are compiled into a Statistics Report File for post-simulation analysis.

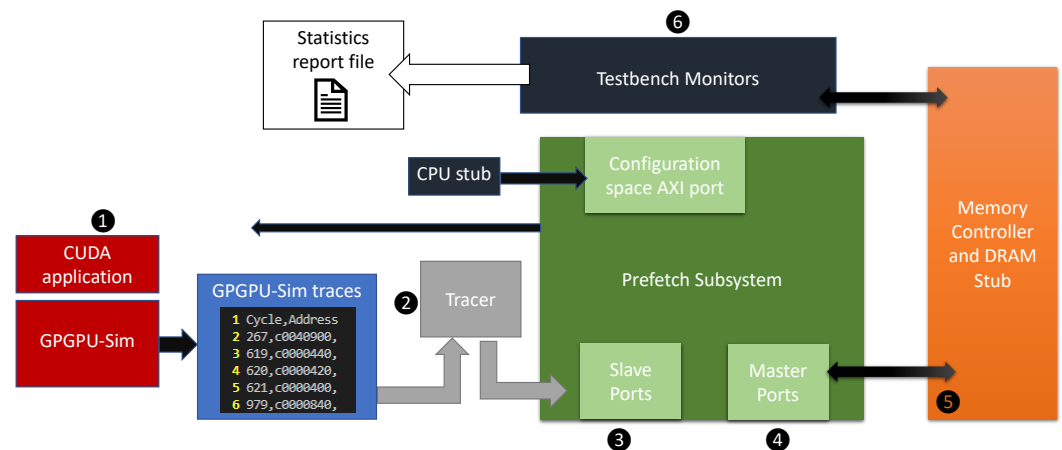


Figure 10. Simulation flow.

5. Experimental Analysis

Using the simulation environment introduced earlier, we conducted performance evaluations on two distinct benchmarks: a CNN which is part of the gpgpu-sim benchmark suite [46] and the Needleman–Wunsch (NW) [47] algorithm written in Cuda. These testbenches were selected to represent applications with varying memory access patterns, allowing for a comprehensive analysis of the prefetcher’s behavior under diverse workloads. Both testbenches share the following hardware and memory configurations:

- Operating Frequency: 667 MHz.
- DRAM Stub Configuration (aligned with DDR5 specifications):
 - Page Size: 2 KBytes.
 - Access Latency: 120 ns for page hits and 150 ns for page misses.

In addition, we have configured the prefetch engines’ memory regions based on the profiling of the benchmark behavior to identify and characterize the dominant memory access regions [48–50]. This ensures that the configuration reflects the actual access patterns of each benchmark rather than an arbitrary choice. In our simulations, we assume that memory regions are optimally configured, since arbitrary configurations could include unused regions and would not be representative of realistic system behavior.

5.1. Benchmark Characteristics

As part of our experimental analysis, we start by examining two benchmarks: the gpgpu-sim CNN model and the NW algorithms, which are described next.

5.1.1. CNN Model

The CNN model benchmark [46], part of the gpgpu-sim benchmark suite, consists of a 5-layer inference model trained for the classification of handwritten digits. The model reads activations and filter weights stored as tensors in memory and performs matrix multiplications at each layer. The results of these matrix multiplications, representing the activations for subsequent layers, are stored in memory for further processing. Figure 11 illustrates a sample of the CNN model’s memory access addresses over time during execution on gpgpu-sim. Each point corresponds to a memory request at a given cycle, showing clear evidence of spatial locality where consecutive addresses across multiple regions are accessed repeatedly. The highlighted region in red indicates one such memory range that is assigned to a dedicated prefetch engine. This mapping demonstrates how regions are identified and bound to prefetchers to exploit locality, enabling more efficient handling of access patterns and improving data retrieval performance.

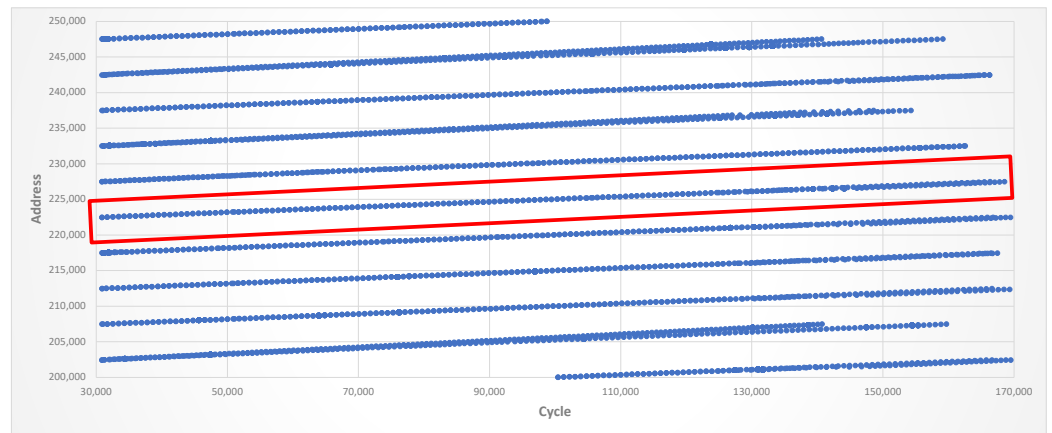


Figure 11. CNN model memory access patterns.

5.1.2. The Needleman–Wunch Algorithm

The Needleman–Wunsch algorithm [47] is a dynamic programming method used for global sequence alignment, commonly applied in bioinformatics to align DNA, RNA, or protein sequences. It constructs a scoring matrix to compute the optimal alignment between two sequences by maximizing similarity while accounting for mismatches and gaps. The algorithm initializes the first row and column of the matrix with gap penalties, representing alignments with leading gaps. Subsequently, each cell in the matrix is filled by calculating the maximum score among three possibilities: a match score derived from aligning corresponding elements of the sequences, a gap penalty for insertion, and a gap penalty for deletion. The following code snippet implements the matrix initialization and filling steps of the Needleman–Wunsch algorithm:

```

d ← Gap penalty score
for i = 0 to length(A)
    F(i,0) ← d * i
for j = 0 to length(B)
    F(0,j) ← d * j
for i = 1 to length(A)
    for j = 1 to length(B)
        Match ← F(i - 1, j - 1) + S(Ai, Bj)
        Delete ← F(i - 1, j) + d
        Insert ← F(i, j - 1) + d
        F(i, j) ← max(Match, Insert, Delete)

```

Here, ‘A’ and ‘B’ represent the two input sequences to be aligned, where ‘Ai’ and ‘Bj’ refer to the individual elements at position ‘i’ and ‘j’ in sequences ‘A’ and ‘B’, respectively. ‘S(Ai, Bj)’ is the substitution score, which quantifies the similarity or mismatch between the elements ‘Ai’ and ‘Bj’. Finally, ‘d’ is the gap penalty, a predefined negative score applied when a gap is introduced in either sequence during alignment. These components collectively define the scoring mechanism used to populate the matrix ‘F’ and determine the optimal alignment.

Figure 12 illustrates a segment of memory access addresses over time for the NW algorithm. The figure clearly shows that the algorithm exhibits both spatial and temporal locality: memory access addresses increment in fixed strides and then repeat across successive cycles. This repetitive pattern indicates predictable reuse of data, which can be effectively exploited by a prefetching strategy. Similar to the CNN test case, a prefetch engine is assigned to a narrow memory region, highlighted in red in the figure. This assignment demonstrates how profiling of access patterns enables targeted prefetching,

ensuring that recurring stride-based requests are serviced effectively by the prefetch engine and improving overall efficiency of memory accesses.

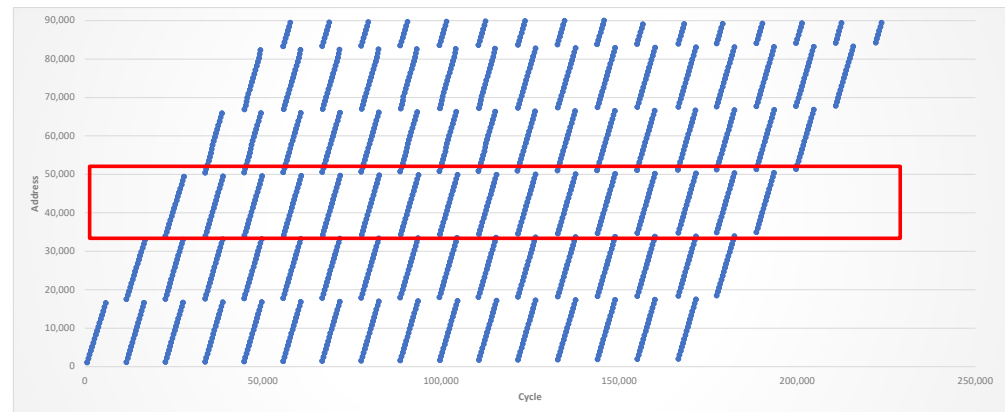


Figure 12. WN algorithm memory access patterns.

5.2. Performance Analysis

The performance analysis begins with illustrating the operation of the prefetch subsystem through Verilog simulations. The analysis is conducted using benchmarks from a CNN model, where we compare the latency of a single read request without prefetching to the scenario where the prefetch subsystem is utilized. The waveforms under consideration consist of five distinct groups of signals, described as follows:

1. **General Signals:** Includes the clock signal (clk) and the predictor state of the prefetcher (st_pr_cur).
2. **GPU to Prefetcher:** AXI signals representing read requests initiated by the GPU and directed to the prefetcher.
3. **Prefetcher to RAM:** AXI signals representing read requests initiated by the prefetcher and forwarded to the DRAM memory stub.
4. **RAM to Prefetcher:** AXI signals representing read data responses sent from the DRAM memory stub back to the prefetcher.
5. **Prefetcher to GPU:** AXI signals representing read data responses sent from the prefetcher to the GPU.

Figure 13 illustrates the waveform of signals for the scenario where a read request is issued by the GPGPU to the prefetch engine and subsequently forwarded by the prefetcher to the DRAM memory with no prefetching. In this case, as the requested data is not found in the prefetched buffer (the prefetch engine is in the ARM state), the overall latency is 114 clock cycles (171 ns).

Figure 14 depicts the waveform of signals when the prefetch engine is in the ACTIVE state and the requested data hits the data buffer within the prefetch engine. Here, the response is returned after a single clock cycle, demonstrating a latency reduction of over 200 times compared to the no-prefetching scenario.

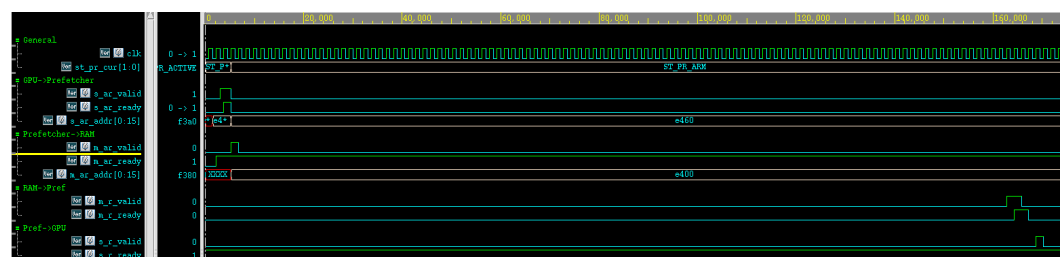


Figure 13. Prefetch engine signal waveform for a single read request with no prefetching.

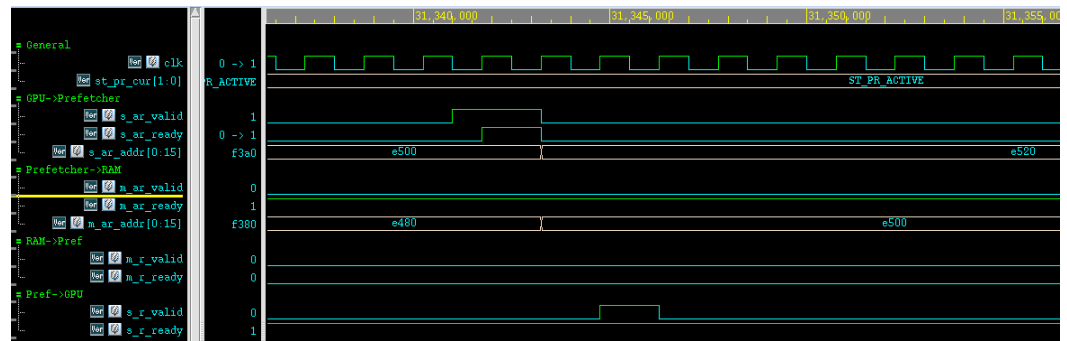


Figure 14. Prefetch engine signal waveform for a single read request with prefetching.

We use the WN algorithm benchmark to demonstrate the impact of block size on prefetch engine performance and its ability to exploit spatial locality. Figure 15 illustrates a waveform of the system signals when a data block of 128 bytes is used. As observed in the marked red box in this figure, the prefetch engine experiences frequent stride misses, resulting in short durations in the ACTIVE state and recurrent transitions to the CLEANUP state.

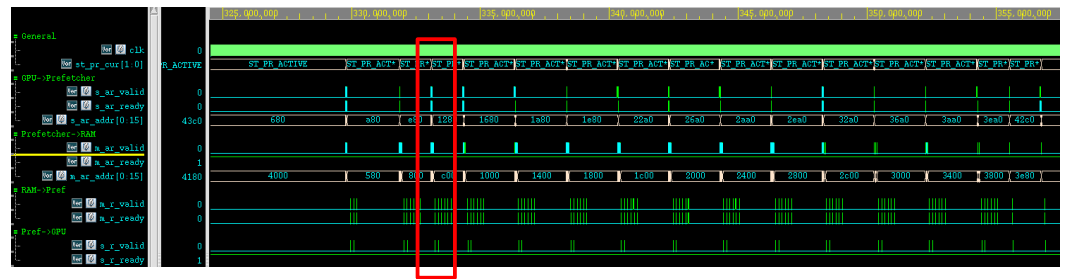


Figure 15. Prefetch engine signal waveform illustrating recurrent stride misses (illustrated in the red box).

Figure 16 provides a closer view of these interferences, where it can be seen that the learned stride of the prefetch engine is occasionally interrupted by an inconsistent negative stride ($-0x20$) instead of the fixed positive stride initially learned.

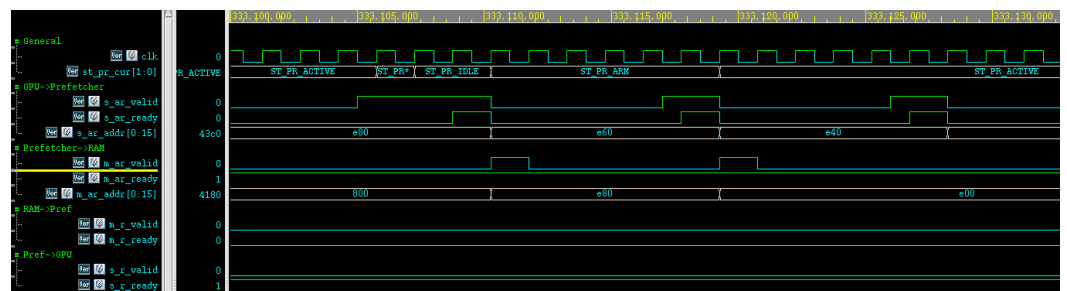


Figure 16. Prefetch engine signal waveform illustrating recurrent stride misses zoom in.

This observation is further supported by analyzing the memory access patterns of the WN algorithm, illustrated in Figure 12 and zoomed in Figure 17. The WN algorithm exhibits monotonically increasing clusters of memory accesses with a positive stride across cycles. Each cluster includes three memory accesses, which follow a pattern of decreasing stride, indicating localized access behavior followed by progressive jumps. These three accesses correspond to the code described in Section 5.1.2, where each iteration of the WN scoring matrix loop performs memory accesses with fixed negative strides for the Match, Delete, and Insert operations. Figure 17 highlights this pattern at a finer granularity. The circled region shows three clustered accesses per iteration, and the arrow illustrates their

progression across cycles. This zoomed-in view makes the relationship between algorithmic operations and memory behavior more explicit, confirming how the stride-based pattern is repeatedly generated by the NW scoring matrix update logic.

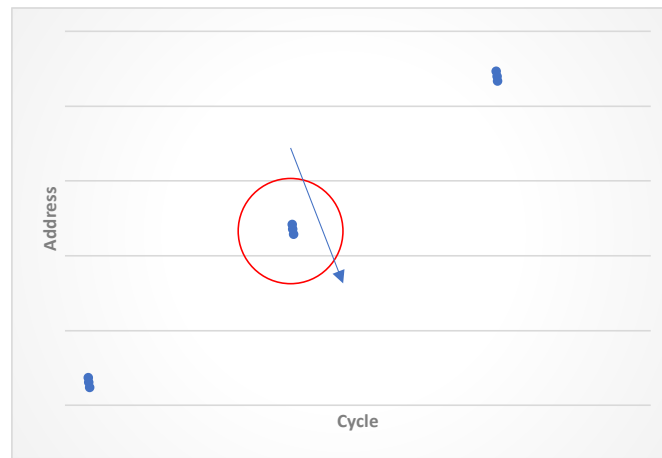


Figure 17. Zoomed-in image of WN algorithm memory access patterns.

Figure 18 presents a waveform of the system signals when the block size is increased to 256 bytes. In this configuration, the prefetch engine successfully captured memory accesses with strides between clusters (0x400 bytes) while also accommodating the negative strides within each cluster. This was achieved because the 256-byte block boundary encompassed the data of the accesses in the cluster within the prefetched block. As shown in Figure 18, this resulted in a stable ACTIVE state for the prefetch engine, with no disruptions to the learned stride.

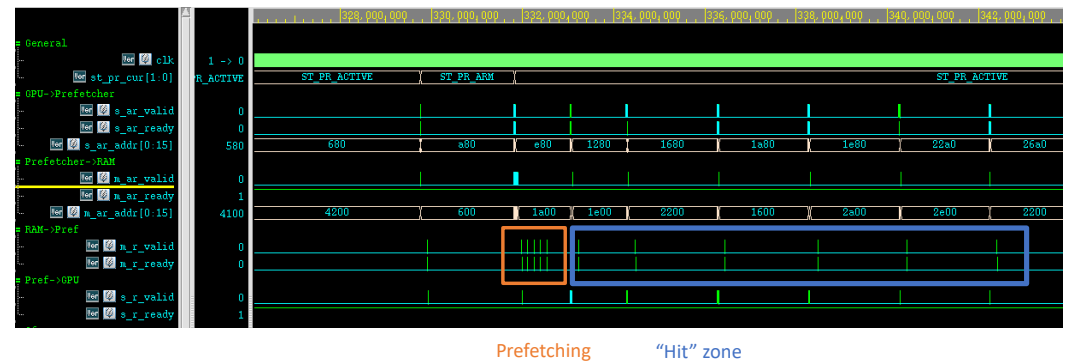


Figure 18. Prefetch engine signal waveform with data block of 256 bytes.

Figure 19 shows the CNN memory read access time histogram (block size = 64 bytes) as observed by the GPU, comparing cases with and without our hardware prefetch subsystem. Without prefetching, two distinct latency bins dominate: ~120 ns corresponding to DRAM page hits, and ~150 ns corresponding to DRAM page misses. When the prefetch engines are enabled with zero outstanding transactions, it effectively acts as a cache for GPGPU requests, reducing the fraction of high-latency accesses. While this configuration already improves average latency compared to the baseline, enabling prefetching with multiple outstanding transactions (in this case, four) further shifts the distribution toward the minimum-latency bin (10 ns). Notably, the number of minimum-latency accesses approximately doubles, demonstrating the effectiveness of our prefetch subsystem in hiding memory latency and improving overall throughput.

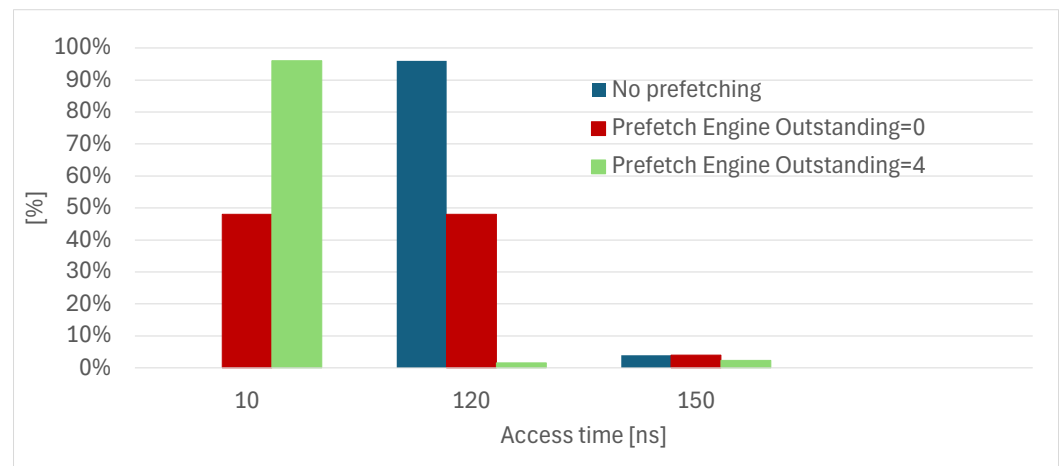


Figure 19. CNN memory read access time histogram (block size = 64 bytes).

Figure 20 presents a histogram analysis of NW memory read access times, demonstrating the significant performance benefits of the prefetching engines across different block sizes (assuming one outstanding prefetch request). The results show that prefetching dramatically improves memory access time compared to the no-prefetching baseline, which exhibits widely distributed access times ranging from 120 to 450 ns. Among the prefetching configurations, 256-byte blocks achieve the highest performance improvement, concentrating approximately 75% of memory accesses at the lowest latency bin (10 ns), while 128-byte blocks achieve about 43% of accesses at this optimal latency. In contrast, 64-byte blocks demonstrate a bimodal distribution with peaks at 120 ns (47%) and 150 ns (30%), indicating less effective spatial locality exploitation. The analysis clearly demonstrates that larger prefetch block sizes provide superior performance.

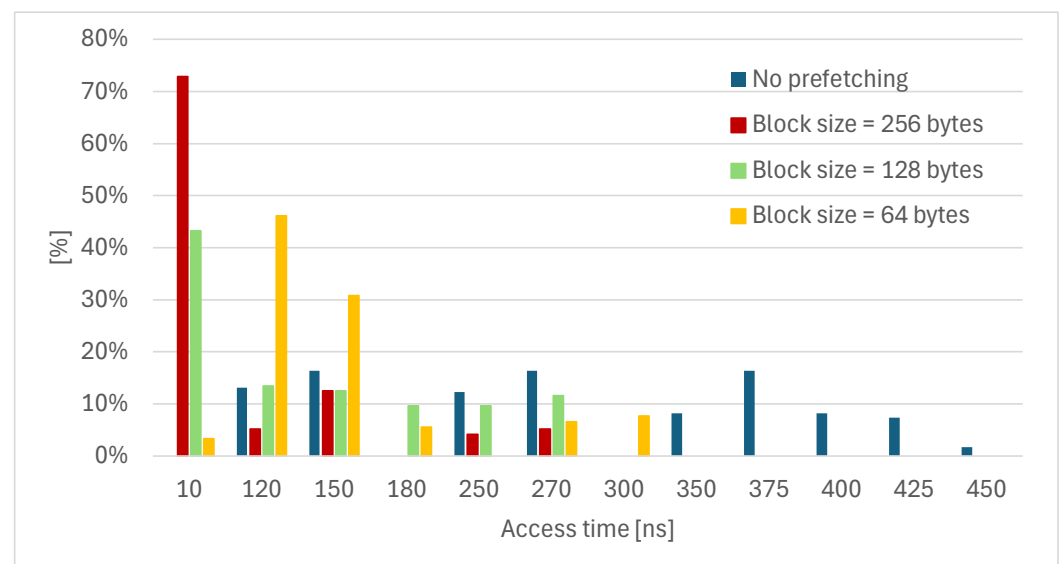


Figure 20. NW memory read access time histogram (number of outstanding prefetch requests = 1).

Figure 21 illustrates the relationship between average memory read latency and prefetch outstanding limits for the CNN benchmark, comparing different block sizes and a no-prefetch baseline. The results reveal several key insights about our prefetching engine effectiveness for the CNN benchmarks. The system with no-prefetch (green line) maintains consistently high latency at approximately 130 ns. In contrast, both prefetching configurations demonstrate dramatic latency reductions, achieving the lowest latency of approximately 8 ns.

Most significantly, the analysis shows that increasing the prefetch outstanding limit beyond one request provides negligible latency benefits. This finding suggests that for applications with highly predictable, constant-stride memory access patterns, aggressive prefetching strategies are unnecessary and potentially wasteful. Instead, optimal performance can be achieved using smaller block sizes (32 bytes) with minimal outstanding requests. Additionally, when the number of outstanding requests is set to zero, the prefetch engine operates as a cache. In this mode, a block size of 64 bytes yields a lower average latency compared to the 32-byte configuration.

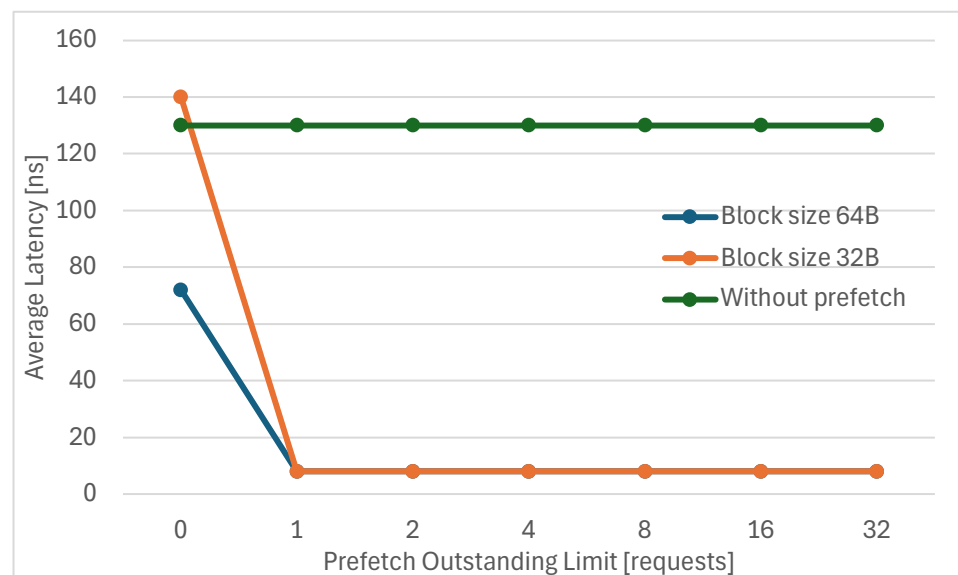


Figure 21. Average memory read access time for the CNN benchmark as a function of the outstanding prefetch request limit and block size.

Figure 22 illustrates the impact of prefetch engine configuration on memory read access latency for the NW benchmark, which exhibits a different behavior compared to the CNN. In this case, limiting the number of outstanding prefetch requests can degrade performance due to the CLEANUP phase of the prefetch engine: once in the CLEANUP state, the engine must wait for all outstanding prefetch responses to complete before returning to IDLE. The more outstanding requests allowed, the longer the engine remains stalled in this phase before resuming operation. As shown in the figure, the optimal configuration employs large data blocks (128 bytes or 256 bytes) with 1–8 outstanding requests, reducing the average memory access latency by up to 80%. A block size of 64 bytes also improves the average memory access latency; however, it is more effective to configure the prefetch engine as a cache only, since stride learning provides limited benefit for the NW benchmark.

Table 2 summarizes the best latency results achieved so far by the prefetch subsystem for each benchmark, along with the corresponding configuration parameters. As shown, the CNN benchmark reached a minimum average latency of 8 ns with small block sizes of 32 bytes, while the NW benchmark benefited from larger blocks of 256 bytes, achieving a latency of 65 ns.

Table 2. Best-performing configurations of the prefetch engines for the CNN and NW benchmarks, showing the lowest observed memory latency and the corresponding settings.

Benchmark	Prefetch Outstanding Limit [Requests]	Block Size [Bytes]	Latency [ns]
CNN	1	32	8
NW	1	256	65

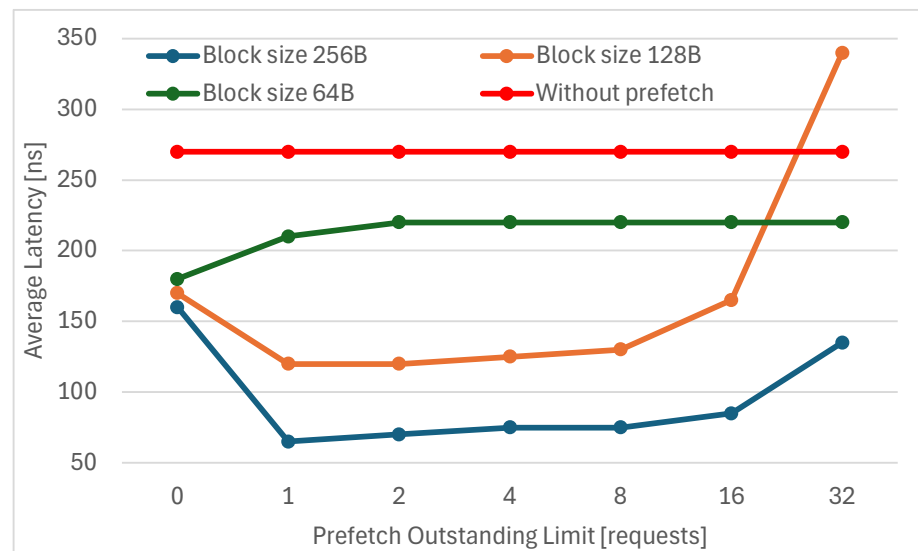


Figure 22. Average memory read access time for the NW benchmark as a function of the outstanding prefetch request limit and block size.

Figure 23 demonstrates the impact of prefetch throttling rates on memory access latency using a prefetch engine configured with 32 outstanding limit requests for the NW benchmark. The results are similar to those shown in Figure 22. Both prefetching configurations (256 bytes and 128 bytes block sizes) achieve substantial latency reductions compared to the no-prefetch baseline, which maintains a constant latency of approximately 250 ns across all throttling rates. The 256 byte block configuration delivers superior performance, achieving minimum latencies of 60–80 ns at low throttling rates (10^{-3} to 10^{-2} [1/cycle]), while the 128 byte block configuration achieves latencies of 120 ns in the same range. However, the results clearly illustrate that excessively high throttling rates severely degrade performance, with both configurations experiencing dramatic latency increases as the throttling rate approaches 10^{-1} [1/cycle] and beyond. Most notably, the 128 byte block configuration suffers major performance degradation at the highest throttling rate, reaching latencies of approximately 340 ns—worse than the no-prefetch case. The analysis confirms that selecting appropriate prefetching parameters within the optimal range of 10^{-3} to 10^{-2} [1/cycle] can reduce transaction latency by up to 80%.

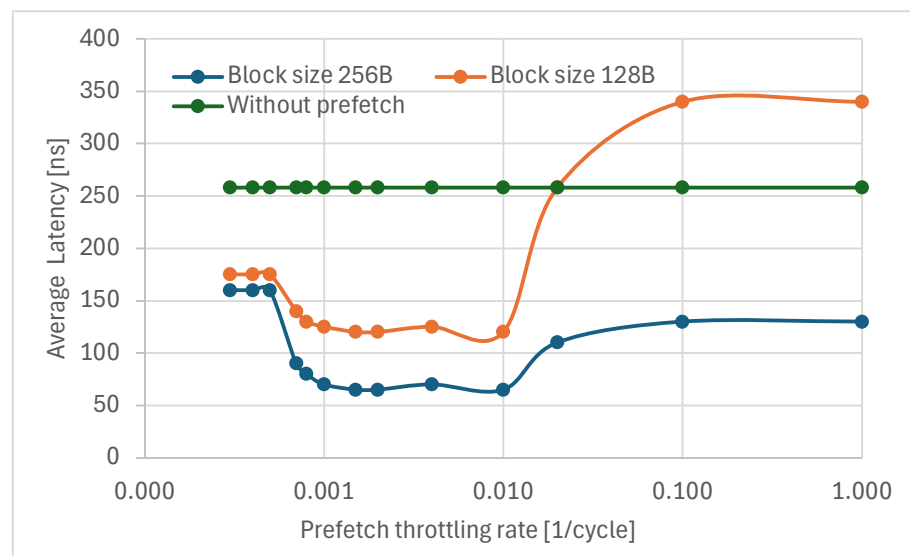


Figure 23. NW memory read access time histogram (number of outstanding prefetch requests = 1).

Table 3 reports the best-performing configuration identified for the NW benchmark. With a prefetch throttling rate of 0.01 cycles and a block size of 256 bytes, the prefetch engine reduced the average memory latency to 65 ns, significantly outperforming the baseline.

Table 3. Best performing prefetch throttling configuration of the prefetch engines for NW benchmark, showing the lowest observed memory latency and the corresponding settings.

Benchmark	Prefetch Throttling Rate [1/Cycles]	Block Size [Bytes]	Latency [ns]
NW	0.01	256	65

To evaluate the effectiveness of our prefetching subsystem, we ran simulations of the CNN and NW benchmarks using a 256-byte block size with one outstanding prefetch request to measure the resulting speedup. For the CNN, the prefetch subsystem reduced latency by up to 80% and achieved a speedup of $1.589\times$, while for NW, we observed latency reductions of up to 80% with a speedup of $1.794\times$. In addition to these two benchmarks, we evaluated five further applications from the gpgpu-sim benchmark suite [51] to demonstrate the generality of our approach. *MUMmerGPU*, a high-throughput DNA sequence alignment program [52], exhibits both regular and irregular access patterns; the prefetch subsystem achieved latency reductions of about 74% with speedups of up to $1.671\times$. *LPS*, a Jacobi solver for a 3D Laplace discretization [53], features regular strides, with latency reductions of up to 78% and speedups of $1.556\times$. *DG*, a mini Discontinuous Galerkin solver [54], combines block-based and irregular mesh accesses; here, the prefetch subsystem achieved moderate improvements with 64% latency reduction and speedups of $1.463\times$. *BFS*, a graph traversal benchmark [55] characterized by pointer chasing and highly irregular memory access, showed more limited benefits with 40% latency reduction and speedups of $1.240\times$. Finally, *WP*, a numerical weather prediction application [56], achieved substantial benefits with 82% latency reduction and speedups of $1.557\times$, thanks to its largely regular access patterns. Overall, these results demonstrate that the proposed prefetch subsystem consistently reduces memory latency across a wide range of workloads, with the greatest gains observed for benchmarks with regular memory access patterns. Table 4 summarizes the speedup, latency reduction, and minimum latency achieved for each of the examined benchmarks.

Table 4. Summary of prefetch engine performance results. The table reports the best-performing configurations of the proposed prefetch engines across a range of representative benchmarks, including machine learning (CNN), bioinformatics (NW, MUMmerGPU), scientific workloads (LPS, DG), graph analytics (BFS), and weather prediction (WP). For each case, the results highlight the maximum observed latency reduction, the corresponding minimum memory latency achieved, and the resulting application speedup. These results demonstrate that the proposed prefetching approach consistently reduces memory access latency across diverse application domains, with speedups ranging from $1.240\times$ (BFS) to $1.794\times$ (NW).

Benchmark	Type	Latency Reduction	Min. Latency [ns]	Speedup
CNN [46]	ML workload	Up to 80%	~8–10	$1.589\times$
NW [47]	Bioinformatics	Up to 80%	~60–80	$1.794\times$
MUMmerGPU [52]	Bioinformatics	Up to 74%	~50–100	$1.671\times$
LPS [53]	Scientific	Up to 78%	~8–15	$1.556\times$
DG [54]	Scientific	Up to 64%	~30–60	$1.463\times$
BFS [55]	Graph analytics	Up to 40%	~60–120	$1.240\times$
WP [56]	Weather forecast	Up to 82%	~10–20	$1.557\times$

6. Discussion

Table 5 highlights the trade-offs of prior prefetching approaches for GPGPUs, ranging from software compiler-based schemes to hardware-assisted warp- or thread-aware prefetchers. While most of these studies achieved relatively small performance improvements, they are often constrained by design-specific limitations. For example, compiler-based prefetching [32] can suffer from a lack of runtime visibility, leading to inaccurate decisions for irregular memory patterns. Inter-thread and CTA-aware prefetching [31,34,36] introduce dependencies between threads or may restrict the prefetching scope to cooperative arrays, which can reduce efficiency. Similarly, opportunistic or stream-based prefetchers [33,39] can be highly sensitive to application-specific patterns, limiting their general applicability. More advanced designs such as WASP [40], APRES [41], and LTRF [42] provide notable gains, but they require either scheduling modifications, cache partitioning, or cooperative hardware/software mechanisms that increase design complexity.

In contrast, our proposed prefetching subsystem combines *generality, adaptivity, and robustness* in a modular hardware framework. Its ability to dynamically reconfigure block sizes (32–256B) and outstanding request limits (1–64) enables runtime tuning across diverse workloads, which is not possible in static software-based prefetching. The inclusion of context flushing and watchdog timers ensures resilience against irregular or stalled access patterns, mitigating one of the key weaknesses of stride- or warp-based designs. Unlike thread-dependent methods, our prefetcher operates independently of scheduling or compiler hints, reducing reliance on programmer annotations or cooperative warp behavior. Furthermore, the prefetch subsystem employs *multiple memory windows*, allowing the prefetcher to operate concurrently on different memory regions and to adaptively exploit diverse spatial locality patterns. Most importantly, experimental results demonstrate average latency reductions of up to 80% and speeds of 1.585–1.794 \times for CNN and NW benchmarks, respectively.

Table 5. Comparison of GPU prefetching mechanisms. This table summarizes prior hardware- and software-based GPU prefetching techniques alongside the proposed approach in this study. For each method, the approach, main features, and limitations are described, together with reported performance improvements. The comparison highlights the evolution from compiler-based and opportunistic prefetching schemes to more advanced warp-aware and adaptive scheduling techniques. Unlike prior approaches, the proposed runtime-configurable prefetching engine (this study) supports a wide range of access patterns through dynamic block sizing, outstanding request control, context flushing, and watchdog timers. As shown, it achieves substantially higher latency reduction (up to 82%) and application speedup (1.794 \times), demonstrating clear advantages in both flexibility and effectiveness over existing solutions.

Method	Approach	Key Features	Limitations	Performance Improvement
Many-thread-aware prefetching [31]	Hardware and software	Inter-thread prefetching: prefetching is performed between threads	Limited in capturing runtime irregular patterns that are unknown at compile time; Inter-thread prefetching introduces dependencies between threads, reducing overall efficiency	15–16%
GPGPU compiler-based prefetching [32]	Software	Compiler prefetches via temporary variables, improving memory usage and workload distribution	Limited ability to predict dynamic/irregular memory access patterns; can result in cache pollution	Minor
OWL: Opportunistic prefetching [33]	Hardware	Opportunistic memory-side prefetching taking advantage of open DRAM rows	Limited to open DRAM rows	2%
CTA-aware prefetching [34,36]	Hardware	Cooperative thread arrays stride prefetching	Limited thread scope within the cooperative array	10%

Table 5. Cont.

Method	Approach	Key Features	Limitations	Performance Improvement
Spare register-aware prefetching [35]	Hardware	Data prefetching mechanism for load pairs where one load depends on the other	Limited to graph algorithms	10%
APOGEE [37]	Hardware	Uses adjacent threads to identify address patterns and dynamically adapt prefetching timeliness	Limited to adjacent threads; relies on latency hiding through SIMT	19%
Orchestrated prefetching and scheduling [38]	Hardware	Coordinates thread scheduling and prefetching decisions	Requires modification in warp scheduling	7–25%
Stream data prefetcher [39]	Hardware and software	Data prefetching based on application-specific data-pattern descriptions	Tightly coupled to an offline data-pattern specification per application	9.2×
Warp-aware selective prefetching (WASP) [40]	Hardware	Dynamically selects slow-progress warps for prefetching	Limited if warp behavior is unpredictable	16.8%
Adaptive prefetching and scheduling (APRES) [41]	Hardware	Group warps predicted to execute the same load instruction in the near future	Contentions between demand fetch and prefetch require cache partitioning; dependent on thread cooperative characteristics	27.8%
Latency-tolerant register file (LTRF) [42]	Hardware and software	Two-level hierarchical register structure; estimated working set prefetched to register cache	Relies on compile-time prefetch decisions, which may miss dynamic runtime behavior	31%
This study	Hardware	Runtime configurable (32–256B blocks, 1–64 outstanding requests); context flushing; watchdog timers; handles diverse access patterns	Hardware complexity; requires proper throttling configuration; power consumption considerations	Up to 82% latency reduction; and speedup up to 1.794×

7. Conclusions

This paper presents a comprehensive design and implementation of a hardware-based DRAM memory prefetching subsystem specifically tailored for GPGPU architectures. Our work addresses the critical challenge of memory access latencies in massively parallel GPU workloads, which often overwhelm traditional memory hierarchies and thread-switching mechanisms.

The proposed modular prefetching subsystem architecture demonstrates several key advantages over conventional approaches. The multi-engine design with dedicated address range handling enables efficient scaling and reduces contention, while the adaptive stride detection mechanism successfully identifies and predicts diverse memory access patterns across different workload types. The integration of robust system-level features, including context flushing, watchdog timers, and flexible configuration interfaces, ensures reliable operation and adaptability to varying computational demands.

Our comprehensive experimental validation reveals significant performance improvements across multiple metrics. The prefetching subsystem achieves memory access latency reductions of up to 82% and speed of 1.240–1.794× for the benchmarks examined. The analysis of different block sizes, outstanding limits, and throttling rates provides significant insights into prefetcher behavior, demonstrating that 256-byte block sizes are optimal for spatially local access patterns, while smaller blocks with minimal outstanding requests suit stride-based patterns. Importantly, our results highlight the necessity of proper throttling rate selection, as excessive prefetching can degrade performance below baseline levels.

The practical implications of this work extend beyond the specific design presented. Our findings establish clear guidelines for prefetcher deployment in production GPGPU systems and provide a foundation for future research in adaptive memory prefetching. The modular architecture enables straightforward integration into existing GPU memory

hierarchies, while the comprehensive parameter analysis offers actionable insights for system designers.

Future research directions include exploring ML-based pattern prediction algorithms, investigating adaptive throttling mechanisms that dynamically adjust to workload characteristics, and extending the design to handle emerging memory technologies such as HBM and processing-in-memory architectures. Additionally, the integration of our prefetching subsystem with GPU compiler optimizations and runtime systems presents opportunities for further performance enhancements in heterogeneous computing environments.

Author Contributions: Conceptualization, F.G., D.S. and I.G.; methodology, F.G.; software, D.S. and I.G.; validation, F.G., B.S., D.S. and I.G.; formal analysis, F.G., D.S. and I.G.; investigation, F.G., B.S., D.S. and I.G.; resources, F.G.; data curation, F.G., B.S., D.S. and I.G.; writing—original draft preparation, F.G. and B.S.; writing—review and editing, F.G. and B.S.; visualization, F.G., B.S., D.S. and I.G.; supervision, F.G.; project administration, F.G.; funding acquisition, F.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Design data is available in: <https://github.com/shemadolev/Verilog-Prefetcher>, accessed on 29 September 2025.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Fermi. 2009. Available online: https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf (accessed on 2 October 2025).
2. Kirk, D.B.; Hwu, W.W. *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed.; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2010.
3. Luebke, D.; Harris, M.; Govindaraju, N.; Lefohn, A.; Houston, M.; Owens, J.; Segal, M.; Papakipos, M.; Buck, I. GPGPU: General-purpose computation on graphics hardware. In Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, Tampa, FL, USA, 11–17 November 2006; p. 208-es.
4. Raju, K.; Chiplunkar, N.N. A survey on techniques for cooperative CPU-GPU computing. *Sustain. Comput. Inform. Syst.* **2018**, *19*, 72–85.
5. Hu, L.; Xilong, C.; Si-Qing, Z. A closer look at GPGPU. *ACM Comput. Surv. (CSUR)* **2016**, *48*, 1–20. [[CrossRef](#)]
6. Owens, J.D.; Luebke, D.; Govindaraju, N.; Harris, M.; Krüger, J.; Lefohn, A.E.; Purcell, T.J. A survey of general-purpose computation on graphics hardware. In *Computer Graphics Forum*; Blackwell Publishing Ltd.: Oxford, UK, 2007; Volume 26, pp. 80–113.
7. Dokken, T.; Hagen, T.R.; Hjelmervik, J.M. An introduction to general-purpose computing on programmable graphics hardware. In *Geometric Modelling, Numerical Simulation, and Optimization: Applied Mathematics at SINTEF*; Springer: Berlin/Heidelberg, Germany, 2007; pp. 123–161.
8. Kim, Y.; Choi, H.; Lee, J.; Kim, J.S.; Jei, H.; Roh, H. Efficient large-scale deep learning framework for heterogeneous multi-GPU cluster. In Proceedings of the 2019 IEEE 4th International Workshops on Foundations and Applications of Self Systems, Umea, Sweden, 16–20 June 2019; pp. 176–181.
9. Mittal, S.; Vaishay, S. A survey of techniques for optimizing deep learning on GPUs. *J. Syst. Archit.* **2019**, *99*, 101635. [[CrossRef](#)]
10. Ovtcharov, K.; Ruwase, O.; Kim, J.Y.; Fowers, J.; Strauss, K.; Chung, E.S. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Res. Whitepaper* **2015**, *2*, 1–4.
11. Mittal, S. A survey of recent prefetching techniques for processor caches. *ACM Comput. Surv. (CSUR)* **2016**, *49*, 1–35. [[CrossRef](#)]
12. Falahati, H.; Hessabi, S.; Abdi, M.; Baniasadi, A. Power-efficient prefetching on GPGPUs. *J. Supercomput.* **2015**, *71*, 2808–2829. [[CrossRef](#)]
13. Jia, W.; Shaw, K.A.; Martonosi, M. Characterizing and improving the use of demand-fetched caches in GPUs. In Proceedings of the 26th ACM International Conference on Supercomputing, Venice, Italy, 25–29 June 2012; pp. 15–24.
14. Jia, W.; Shaw, K.; Martonosi, M. MRPB: Memory request prioritization for massively parallel processors. In Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), Orlando, FL, USA, 15–19 February 2014; pp. 272–283.

15. Torres, Y.; Gonzalez-Escribano, A.; Llanos, D.R. Understanding the impact of CUDA tuning techniques for Fermi. In Proceedings of the International Conference on High Performance Computing and Simulation (HPCS), Istanbul, Turkey, 4–8 July 2011; pp. 631–639.
16. Xie, X.; Liang, Y.; Wang, Y.; Sun, G.; Wang, T. Coordinated static and dynamic cache bypassing for GPUs. In Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), Burlingame, CA, USA, 7–11 February 2015; pp. 76–88.
17. AXI Protocol Specification. Document Number: ARM IHI 0022, August 2025. Available online: <https://developer.arm.com/documentation/ih0022/latest> (accessed on 2 October 2025).
18. Yu, S. *Semiconductor Memory Devices and Circuits*; CRC Press: Boca Raton, FL, USA, 2022.
19. Lee, D.U. Tutorial: HBM DRAM and 3D Stacked Memory. In Proceedings of the 2022 IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, CA, USA, 20–24 February 2022; pp. 1–113.
20. Callahan, D.; Kennedy, K.; Porterfield, A. Software Prefetching. In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV), New York, NY, USA, 8–11 April 1991; pp. 40–52. [\[CrossRef\]](#)
21. Mowry, T.C.; Lam, M.S.; Gupta, A. Design and Evaluation of a Compiler Algorithm for Prefetching. *ACM SIGPLAN Not.* **1992**, *27*, 62–73. [\[CrossRef\]](#)
22. Caragea, G.C.; Taznnes, A.; Keceli, F.; Barua, R.; Vishkin, U. Resource-Aware Compiler Prefetching for Many-Cores. In Proceedings of the IEEE 9th International Symposium on Parallel and Distributed Computing, Istanbul, Turkey, 7–9 July 2010; pp. 133–140.
23. Ainsworth, S.; Jones, T.M. Software prefetching for indirect memory accesses. In Proceedings of the IEEE/ACM International Symposium of Code Generation and Optimization (CGO), Austin, TX, USA, 4–8 February 2017; pp. 305–317.
24. Hadade, I.; Jones, T.M.; Wang, F.; di Mare, L. Software Prefetching for Unstructured Mesh Applications. *ACM Trans. Parallel Comput.* **2020**, *7*, 1–23. [\[CrossRef\]](#)
25. Falsafi, B.; Wenisch, T.F. *A Primer on Hardware Prefetching*; Springer Nature: Berlin/Heidelberg, Germany, 2022.
26. Jouppi, N.P. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In Proceedings of the 17th Annual International Symposium on Computer Architecture, Seattle, WA, USA, 28–31 May 1990; pp. 364–373.
27. Palacharla, S.; Kessler, R.E. Evaluating stream buffers as a secondary cache replacement. In Proceedings of the 21st International Symposium on Computer Architecture (ISCA), Chicago, IL, USA, 18–21 April 1994; pp. 24–33.
28. Chen, T.F.; Baer, J.L. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput.* **1995**, *44*, 609–623. [\[CrossRef\]](#)
29. Fu, J.W.C.; Patel, J.H.; Janssens, B.L. Stride directed prefetching in scalar processors. *ACM Sigmicro Newsl.* **1992**, *23*, 102–110. [\[CrossRef\]](#)
30. Nesbit, K.J.; Smith, J.E. Data Cache Prefetching Using a Global History Buffer. In Proceedings of the IEEE 10th International Symposium on High Performance Computer Architecture (HPCA'04), Madrid, Spain, 14–18 February 2004.
31. Lee, J.; Lakshminarayana, N.B.; Kim, H.; Vuduc, R. Many-Thread Aware Prefetching Mechanisms for GPGPU Applications. In Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Atlanta, GA, USA, 4–8 December 2010; pp. 213–224.
32. Yang, Y.; Xiang, P.; Kong, J.; Zhou, H. A GPGPU compiler for memory optimization and parallelism management. *ACM SIGPLAN Not.* **2010**, *45*, 86–97. [\[CrossRef\]](#)
33. Jog, A.; Kayiran, O.; Nachiappan, N.C.; Mishra, A.K.; Kandemir, M.T.; Mutlu, O.; Iyer, R.; Das, C.R. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. *ACM SIGPLAN Not.* **2013**, *48*, 395–406. [\[CrossRef\]](#)
34. Jeon, H.; Koo, G.; Annaram, M. *CTA-Aware Prefetching for GPGPU*; Computer Engineering Technical Report Number CENG-2014-08; University of Southern California Los Angeles: Los Angeles, CA, USA, 2014.
35. Lakshminarayana, N.B.; Kim, H. Spare Register Aware Prefetching for Graph Algorithms on GPUs. In Proceedings of the IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), Orlando, FL, USA, 15–19 February 2014; pp. 614–625.
36. Koo, G.; Jeon, H.; Liu, Z.; Kim, N.S.; Annaram, M. CTA-Aware Prefetching and Scheduling for GPU. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), Vancouver, BC, Canada, 21–25 May 2018; pp. 137–148.
37. Sethia, A.; Dasika, G.; Samadi, M.; Mahlke, S. APOGEE: Adaptive prefetching on GPUs for energy efficiency. In Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, UK, 7–11 September 2013; pp. 73–82.
38. Jog, A.; Kayiran, O.; Mishra, A.K.; Kandemir, M.T.; Mutlu, O.; Iyer, R.; Das, C.R. Orchestrated Scheduling and Prefetching for GPGPUs. In Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13), Tel-Aviv, Israel, 23–27 June 2013; pp. 332–343.

39. Neves, N.; Tomás, P.; Roma, N. Stream data prefetcher for the GPU memory interface. *J. Supercomput.* **2018**, *74*, 2314–2328. [CrossRef]
40. Oh, Y.; Yoon, M.K.; Park, J.H.; Park, Y.; Ro, W.W. WASP: Selective Data Prefetching with Monitoring Runtime Warp Progress on GPUs. *IEEE Trans. Comput.* **2018**, *67*, 1366–1373. [CrossRef]
41. Oh, Y.; Kim, K.; Yoon, M.K.; Park, J.H.; Annavaram, M.; Ro, W.W. Adaptive Cooperation of Prefetching and Warp Scheduling on GPUs. *IEEE Trans. Comput.* **2019**, *68*, 609–616. [CrossRef]
42. Sadrosadati, M.; Mirhosseini, A.; Ehsani, S.B.; Sarbazi-Azad, H.; Drumond, M.; Falsafi, B.; Ausavarungnirun, R.; Mutlu, O. LTRF: Enabling High-Capacity Register Files for GPUs via Hardware/Software Cooperative Register Prefetching. *ACM SIGPLAN Not.* **2018**, *53*, 489–502. [CrossRef]
43. Guo, H.; Huang, L.; Lü, Y.; Ma, J.; Qian, C.; Ma, S.; Wang, Z. Accelerating BFS via Data Structure-Aware Prefetching on GPU. *IEEE Access* **2018**, *6*, 60234–60248. [CrossRef]
44. Zhang, P.; Srivastava, A.; Nori, A.V.; Kanna, R.; Prasanna, V.K. Fine-Grained Address Segmentation for Attention-Based Variable-Degree Prefetching. In Proceedings of the 19th ACM International Conference on Computing Frontiers (CF '22), Turin, Italy, 17–22 May 2022; pp. 103–112.
45. Forencich, A. Verilog AXI Components Github Repository. 2022. Available online: <https://github.com/alexforencich/verilog-axi> (accessed on 2 October 2025).
46. Aamod, T. GPGPU-Sim Manual. Available online: <http://www.gpgpu-sim.org/manual> (accessed on 2 October 2025).
47. Needleman, S.B.; Wunsch, C.D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.* **1970**, *48*, 443–453. [CrossRef] [PubMed]
48. Huangfu, Y.; Zhang, W. Boosting GPU Performance by Profiling-Based L1 Data Cache Bypassing. In Proceedings of the 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Shenzhen, China, 4–7 May 2015; pp. 1119–1122. [CrossRef]
49. Stoltzfus, L.; Emani, M.K.; Lin, P.; Liao, C. Data Placement Optimization in GPU Memory Hierarchy using Predictive Modeling. In Proceedings of the Workshop on Memory Centric High Performance Computing, Dallas, TX, USA, 11 November 2018.
50. Li, M.; Zhang, Q.; Gao, Y.; Fang, W.; Lu, Y.; Ren, Y.; Xie, Z. Profile-Guided Temporal Prefetching. In Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), Tokyo, Japan, 21–25 June 2025; pp. 572–585.
51. Bakhoda, A.; Yuan, G.L.; Fung, W.W.L.; Wong, H.; Aamodt, T.M. Analyzing CUDA workloads using a detailed GPU simulator. In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, Boston, MA, USA, 26–28 April 2009; pp. 163–174. [CrossRef]
52. Schatz, M.; Trapnell, C.; Delcher, A.; Varshney, A. High-throughput sequence alignment using Graphics Processing Units. *BMC Bioinform.* **2007**, *8*, 474. [CrossRef] [PubMed]
53. Giles, M. Jacobi Iteration for a Laplace Discretisation on a 3D Structured Grid. Available online: <https://people.maths.ox.ac.uk/gilesm/codes/laplace3d/laplace3d.pdf> (accessed on 2 October 2025).
54. Hesthaven, J.S.; Warburton, T. *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*, 1st ed.; Springer Publishing Company, Incorporated: Berlin/Heidelberg, Germany, 2007.
55. Harish, P.; Narayanan, P.J. Accelerating Large Graph Algorithms on the GPU Using CUDA. In Proceedings of the International Conference on High-Performance Computing, Goa, India, 18–21 December 2007; pp. 197–208.
56. Michalakes, J.; Vachharajani, M. GPU acceleration of numerical weather prediction. In Proceedings of the IPDPS 2008: International Symposium on Parallel and Distributed Processing, Miami, FL, USA, 14–18 April 2008; pp. 1–7.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.