POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Integrating GPU Execution in Dataflow Stream Processing

Operator Design and Performance Evaluation on the Renoir Platform

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Alireza Azadi**

Student ID: 213651
Advisor: Prof. Alessandro Margara
Co-advisors: Luca De Martini
Academic Year: 2024-25

Abstract

GPU acceleration offers significant performance potential for stream processing, but integrating it into dataflow frameworks is challenging. GPUs require explicit control over batching, data layout, and data transfers—concerns that conflict with the fine-grained, high-level abstractions of stream processing systems.

This thesis investigates whether GPU programming can be embedded within a dataflow execution model. The study uses Renoir, a dataflow-based stream processing framework that combines expressive programming abstractions with low-overhead execution.

The work introduces a GPU-aware dataflow operator called `map_gpu`, which lets developers apply custom GPU kernels to streaming data in a way similar to traditional `map` operators. A kernel interface allows developers to provide GPU kernels, while the operator handles batching and buffer management according to configurable strategies.

The approach is evaluated through experiments that analyze how batching strategies and data transfer overheads affect performance. Results show that GPU acceleration works well within a dataflow framework when batching and data movement are carefully managed and when workloads are compute-intensive enough to offset kernel launch, synchronization, and transfer costs. For workloads with low computational intensity, fine-grained execution eliminates performance benefits.

Overall, this work shows that while GPU programming cannot be fully hidden in a dataflow system, it can be meaningfully integrated while preserving the dataflow programming model and enabling efficient GPU execution.

Keywords: dataflow processing, stream processing, GPU computing, batching

Abstract in lingua italiana

L'accelerazione GPU offre un significativo potenziale prestazionale per lo stream processing, ma integrarla nei framework dataflow è impegnativo. Le GPU richiedono controllo esplicito su batching, layout dei dati e trasferimenti—aspetti che confliggono con le astrazioni ad alto livello e a grana fine dei sistemi di stream processing.

Questa tesi indaga se la programmazione GPU possa essere integrata in un modello di esecuzione dataflow. Lo studio utilizza Renoir, un framework di stream processing basato su dataflow che combina astrazioni di programmazione espressive con esecuzione a basso overhead.

Il lavoro introduce un operatore dataflow GPU-aware chiamato `map_gpu`, che permette agli sviluppatori di applicare kernel GPU personalizzati ai dati in streaming in modo simile agli operatori `map` tradizionali. Un'interfaccia kernel consente agli sviluppatori di fornire kernel GPU, mentre l'operatore gestisce batching e buffer secondo strategie configurabili.

L'approccio è valutato attraverso esperimenti che analizzano come le strategie di batching e gli overhead di trasferimento dati influenzano le prestazioni. I risultati mostrano che l'accelerazione GPU funziona bene all'interno di un framework dataflow quando batching e movimento dati sono gestiti con attenzione e quando i carichi di lavoro sono sufficientemente compute-intensive da compensare i costi di lancio kernel, sincronizzazione e trasferimento.

In sintesi, questo lavoro mostra che, sebbene la programmazione GPU non possa essere completamente nascosta in un sistema dataflow, può essere integrata in modo significativo preservando il modello di programmazione dataflow e abilitando un'esecuzione GPU efficiente.

Parole chiave: elaborazione del flusso di dati, elaborazione del flusso, elaborazione GPU, elaborazione in batch

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem Statement . . . 1
1.3 Research Question and Objectives . . . 2
1.4 Approach and Contributions . . . 2
1.5 Summary of Results . . . 3
1.6 Structure of the Thesis . . . 4

2 Background and Motivations **5**
2.1 Dataflow and Stream Processing Models . . . 5
2.1.1 Dataflow Programming Abstraction . . . 5
2.1.2 Unbounded Data and Execution Models . . . 6
2.1.3 Event Time and Processing Time . . . 6
2.1.4 Windowing . . . 7
2.1.5 Triggers and Incremental Results . . . 8
2.1.6 Separation of Logical and Physical Execution . . . 8
2.1.7 Implications for Hardware Acceleration . . . 9
2.2 The Renoir Dataflow Platform . . . 9
2.2.1 Programming Model . . . 10
2.2.2 Execution Model and Runtime Architecture . . . 10
2.2.3 Windowing and State Management . . . 11
2.2.4 Renoir as an Evaluation Platform . . . 11
2.2.5 Implications for Accelerator Integration . . . 12

2.3 GPU Programming Model . . . 12
2.3.1 SIMT Execution and Warp-Based Scheduling . . . 13
2.3.2 Data Access Patterns and Memory Coalescing . . . 15
2.3.3 Data Layouts: Array of Structures and Structure of Arrays . . . 16
2.3.4 Memory Hierarchy, Shared Memory, and Occupancy . . . 17
2.3.5 Batching and Kernel Launch Overheads . . . 19
2.3.6 Data Transfers and the PCIe Bottleneck . . . 20
2.3.7 Asynchronous Execution and Overlap . . . 20
2.3.8 Implications for Dataflow Integration . . . 21

3 Design and Implementation 23
3.1 Design Overview and Problem Decomposition . . . 23
3.1.1 Problem Setting . . . 23
3.1.2 Rationale for an Operator-Level Integration . . . 25
3.1.3 Fundamental Constraints Observed During Implementation . . . 26
3.1.4 Design Consequences . . . 26
3.1.5 Scope and Limitations . . . 28
3.2 Programming Model and API . . . 29
3.2.1 Overview of the `map_gpu` Operator . . . 29
3.2.2 Semantic Guarantees and Non-Guarantees . . . 31
3.2.3 Kernel-Centric Programming Model . . . 33
3.2.4 Execution Context and Backend Selection . . . 33
3.2.5 Illustrative Usage Example . . . 34
3.2.6 Relation to Design Goals . . . 34
3.3 MapGpu Operator Execution Flow . . . 35
3.3.1 Internal Pull-Based Interface . . . 35
3.3.2 Stream Element Handling . . . 35
3.3.3 Flush Decision Logic . . . 37
3.3.4 Timestamp Preservation and Correctness . . . 38
3.4 Buffering and Data Layout . . . 39
3.4.1 Separation of Buffering Responsibilities . . . 39
3.4.2 The GpuKernel Trait Contract . . . 40
3.4.3 Array-of-Structures vs. Structure-of-Arrays . . . 41
3.4.4 Vectorization and Alignment . . . 42
3.5 Asynchronous Pipeline and Double-Buffering . . . 43
3.5.1 The Pipelining Opportunity . . . 43
3.5.2 Double-Buffering Architecture . . . 44

3.5.3 Implementation in `flush()` . . . 44
3.5.4 Lagged Output Model . . . 46
3.5.5 When to Use Double-Buffering . . . 46
3.5.6 Memory Management Considerations . . . 46
3.6 Batching Strategies . . . 47
3.6.1 Strategy Enumeration and Semantics . . . 47
3.6.2 Strategy Selection Guidelines . . . 50
3.6.3 Implementing Custom Strategies . . . 51
3.6.4 Interaction with Kernel Hints . . . 52

4 Evaluation 53
4.1 Experimental Setup . . . 53
4.1.1 Hardware Platform . . . 53
4.1.2 Unified Memory Architecture . . . 54
4.1.3 Absence of NVIDIA/CUDA Hardware . . . 54
4.1.4 Software Stack . . . 55
4.1.5 Evaluation Methodology . . . 55
4.2 Batching Strategy Evaluation . . . 58
4.2.1 Strategies Under Evaluation . . . 59
4.2.2 Results . . . 59
4.3 Preliminary Evaluation: The Reduce Operator . . . 63
4.3.1 Context and Motivation . . . 63
4.3.2 Profiling Analysis . . . 64
4.3.3 Benchmark Results . . . 65
4.3.4 Analysis . . . 70
4.4 Black–Scholes Option Pricing . . . 70
4.4.1 Algorithm Description . . . 70
4.4.2 Rationale for Selection . . . 71
4.4.3 GPU Kernel and Pipeline . . . 72
4.4.4 Benchmark Configuration . . . 73
4.4.5 Results . . . 74
4.4.6 Analysis . . . 79
4.5 Monte Carlo Option Pricing . . . 80
4.5.1 Motivation . . . 80
4.5.2 Algorithm Description . . . 81
4.5.3 GPU Kernel and Pipeline . . . 82
4.5.4 Benchmark Configuration . . . 84

4.5.5 Correctness Validation and RNG Considerations . . . 85
4.5.6 Results . . . 86
4.5.7 Analysis . . . 91
4.6 Cross-Benchmark Analysis and Discussion . . . 92
4.6.1 Arithmetic Intensity as the Dominant Factor . . . 92
4.6.2 Roofline Analysis . . . 93
4.6.3 Performance Regimes . . . 96
4.6.4 CPU-Side Overhead Breakdown . . . 97
4.6.5 Overhead Amortization . . . 98
4.6.6 Implications for GPU Integration in Dataflow Systems . . . 99
4.6.7 Decision Framework . . . 100

5 Related Work **103**
5.1 GPU Stream Processing: The Core Challenge . . . 103
5.2 Prior Systems . . . 103
5.2.1 SABER (SIGMOD 2016) . . . 104
5.2.2 G-Storm (IEEE TBD 2018) . . . 104
5.2.3 GASSER (IEEE Access 2019) . . . 105
5.2.4 FineStream (USENIX ATC 2020) . . . 105
5.2.5 WindFlow (JPDC 2024) . . . 106
5.2.6 Slider (DaMoN 2021) . . . 106
5.3 Positioning This Thesis . . . 107
5.3.1 Unique Contributions . . . 107
5.3.2 Limitations . . . 108
5.4 Summary . . . 108

6 Conclusions and Future Work **111**
6.1 Summary of Contributions . . . 111
6.2 Answering the Research Question . . . 112
6.3 Limitations . . . 113
6.4 Future Work . . . 114
6.5 Concluding Remarks . . . 116

Bibliography **117**

1 | Introduction

1.1. Motivation

Stream processing systems have become essential for applications that analyze continuous data flows under tight latency and throughput constraints. Developers express computation as operators connected in a dataflow graph, while the runtime handles scheduling, parallelism, and buffering.

As these workloads grow more demanding—from complex event processing [9] to real-time machine learning inference [8]—CPU-only architectures often cannot deliver the needed performance [15]. Modern computing platforms are increasingly heterogeneous: besides multi-core CPUs, GPUs offer massive parallelism and high memory bandwidth, making them attractive for compute-intensive, data-parallel tasks.

Yet integrating GPU execution into high-level stream processing frameworks remains difficult. GPU programming requires explicit control over memory allocation, data layout, batching, and kernel invocation—concerns that dataflow models try to hide from developers.

This creates a practical dilemma: developers must either write low-level GPU code (sacrificing composability and the benefits of dataflow abstractions) or stick with CPU execution (potentially missing significant speedups). This thesis investigates whether this trade-off can be reduced by embedding GPU execution into a dataflow programming model.

1.2. Problem Statement

GPUs excel at throughput-oriented computation, but their execution model differs sharply from CPU-centric streaming execution [18]. GPU kernels work best on large batches with regular memory access patterns, and the computation per element must be high enough to offset kernel launch and data transfer overhead. Meanwhile, dataflow systems process streams element-by-element (or in small windows), relying on the runtime to handle buffering and parallelization.

A naïve GPU integration exposes a fundamental mismatch. Stream processing frameworks are typically latency-sensitive and push individual elements or small bundles, whereas GPUs need large, explicit batches to hide memory latency and amortize launch costs [18].

Fine-grained processing leads to excessive overhead and poor GPU utilization, while aggressive manual batching increases latency and can interfere with other operators. Additionally, GPU-friendly data layouts (e.g., Structure-of-Arrays) often conflict with the natural representation of stream elements as records (Array-of-Structures). The central problem is not just running computation on a GPU, but reconciling these conflicting execution models within a dataflow framework.

1.3. Research Question and Objectives

This work addresses the following research question:

> To what extent can GPU programming be embedded within dataflow execution models, while preserving abstraction, composability, and execution semantics of stream processing?

The thesis pursues four objectives:

1. **Design** an abstraction that enables GPU-accelerated computation within a dataflow framework without requiring users to manage low-level GPU details.
2. **Implement** this abstraction in the Renoir dataflow platform, extending its operator model to support GPU execution.
3. **Evaluate** the approach on representative data-parallel workloads, analyzing how performance depends on batching and data layout.
4. **Assess limitations** by identifying when GPU integration helps and when overheads reduce its effectiveness.

The emphasis is on the interaction between performance and abstraction: a successful integration should accelerate computation while preserving the programming model that makes dataflow systems productive.

1.4. Approach and Contributions

This thesis extends the Renoir platform with a GPU-enabled operator that processes stream elements using GPU kernels. The approach embeds GPU execution within an operator interface aligned with Renoir's dataflow abstractions, allowing GPU acceleration

without restructuring applications into GPU-centric programs.

The main contributions are:

- A **GPU-accelerated dataflow operator** integrated into Renoir's operator model.
- A **batching and buffering strategy** that bridges fine-grained stream processing and coarse-grained GPU execution.
- An **asynchronous execution model** that overlaps computation and data transfer to reduce overhead.
- An **experimental evaluation** using data-parallel benchmarks (Black–Scholes option pricing and Monte Carlo simulation) to characterize performance trade-offs.

1.5. Summary of Results

The evaluation shows that GPU execution can be effectively integrated into a dataflow framework and can provide substantial speedups for large, compute-intensive workloads. However, GPU acceleration is not universally beneficial: small batches and workloads dominated by memory transfer or synchronization overhead may not offset GPU costs.

Beyond batch size and parallelism, the experiments show that *computational intensity* plays a decisive role. In the Black–Scholes benchmark, despite data-level parallelism and SIMD-friendly execution, the GPU does not consistently outperform parallel CPU execution. The evaluation uses an Apple M2 Pro system with 8 performance cores plus 4 efficiency cores (12 total) and a *unified memory architecture* where CPU and GPU share physical memory. This architecture eliminates the PCIe transfer bottleneck typical of discrete GPUs, making it a favorable scenario for GPU acceleration; systems with discrete GPUs may show different results due to transfer overhead. On this platform, the GPU achieves a peak speedup of 2.29× over sequential CPU execution for large inputs, but averages 0.45× relative to 12-core parallel CPU—meaning parallel CPU is about 2.2× faster in most cases. This reflects the workload's low arithmetic intensity (about 1.4 FLOP/byte), which is insufficient to overcome GPU pipeline overhead when competing against 12 CPU cores.

In contrast, the Monte Carlo simulation has much higher computational intensity (about 35,000 FLOP/byte). Although it does not fully exploit SIMD execution due to control flow characteristics, the large number of floating-point operations allows the GPU to achieve speedups of up to 555× over sequential CPU and 55.6× over 12-core parallel CPU, with averages of about 504× and 51× respectively.

These results show that effective GPU acceleration in a dataflow framework requires not only sufficient parallelism and appropriate batching, but also workloads that are compute-intensive enough to amortize execution and data movement costs.

1.6. Structure of the Thesis

The thesis is organized as follows:

- **Chapter 2** introduces background concepts including dataflow programming models, the Renoir platform, and GPU programming fundamentals.
- **Chapter 3** presents the design and implementation of the GPU-enabled operator, covering both the programming model and internal architecture.
- **Chapter 4** evaluates the approach through benchmark scenarios and analyzes performance characteristics.
- **Chapter 5** discusses related work on GPU acceleration in dataflow and stream processing systems.
- **Chapter 6** concludes with limitations and future research directions.

2 | Background and Motivations

2.1. Dataflow and Stream Processing Models

Modern data processing often involves unbounded, potentially out-of-order streams of data produced at large scale: application logs, sensor readings, financial transactions, and user events [1]. Processing such data requires models that can produce meaningful results despite incomplete information, variable delays, and constantly arriving inputs [1]. Traditional batch approaches, which assume finite and complete datasets, cannot handle these requirements [6].

The Dataflow Model by Akidau et al. [1] provides a framework for reasoning about these challenges. Rather than treating streaming as a separate paradigm, it breaks down the problem into orthogonal concerns: *what* is computed (windowing), *when* results are emitted (triggers), and *how* refinements are handled (accumulation and retractions). This makes explicit the trade-offs between correctness, latency, and cost [1]. Before introducing these concepts, we first describe the basic abstraction shared by dataflow and stream processing systems.

2.1.1. Dataflow Programming Abstraction

Dataflow programming models provide a declarative way to express computation as a directed graph of operators connected by data dependencies. Nodes represent computational operators, while edges represent streams of data flowing between them. Execution is driven by data availability rather than explicit control flow [1]. The runtime system manages scheduling, parallelism, and resource allocation transparently [6, 10].

This abstraction lets programmers focus on the structure and semantics of computation rather than low-level execution details. Operators are composed to form pipelines describing how data should be transformed, while concerns like concurrency, buffering, and parallel execution are handled by the runtime [6, 10].

Stream processing systems extend the dataflow paradigm to support continuous, poten-

tially unbounded data streams [1]. Instead of operating on finite datasets, these systems process elements incrementally as they arrive, often under strict latency constraints [17]. Typical applications include real-time analytics, monitoring, event processing, and online simulations.

Stream processing combines expressive high-level abstractions with efficient execution on modern hardware [6, 10].

2.1.2. Unbounded Data and Execution Models

A key distinction in the Dataflow Model is between the *nature of the data* and the *execution engine* [1]. Data may be bounded or unbounded, while execution engines may operate in batch, micro-batch, or streaming modes. These dimensions are orthogonal: unbounded data can be processed using batch systems through repeated executions, and bounded datasets can be processed by streaming engines [1, 6].

By separating data characteristics from execution strategy, the Dataflow Model enables a uniform logical representation of computation that can run on different runtime engines [1]. This separation lets developers focus on computation semantics while deferring latency and resource decisions to the execution layer [6].

2.1.3. Event Time and Processing Time

In distributed stream processing, time plays a central role in defining computation semantics. The Dataflow Model distinguishes between *event time* and *processing time*, two temporal domains that affect correctness and latency differently [1].

Event time refers to when an event actually occurred, as recorded by the data source. Processing time refers to when the event is observed and processed by the system. In practice, these often diverge due to network delays, scheduling variability, and failures [1, 17]. Processing elements strictly in arrival order may lead to incomplete or misleading results when events arrive late or out of order [17].

The Dataflow Model treats event time as the primary reference for defining computation semantics. Windows are defined over event time, and correctness is evaluated based on the completeness of event-time data rather than arrival order [1]. To make progress with unbounded and delayed data, systems use heuristic mechanisms such as watermarks, which estimate lower bounds on event times unlikely to be observed in the future [1, 17].

By separating event time from processing time, the Dataflow Model enables principled

handling of late data, allowing systems to balance timeliness and result accuracy [1].

2.1.4. Windowing

When processing unbounded data streams, applying aggregation or grouping operations over the entire stream is generally not possible. Windowing addresses this by defining finite subsets of the stream over which computation can be performed [1, 17]. In the Dataflow Model, windowing assigns each element to one or more windows based on its event-time timestamp [1].

Windows can be classified by how they partition the event-time axis. Fixed (or tumbling) windows divide time into non-overlapping intervals of equal size, assigning each element to exactly one window [1, 6]. Sliding windows allow windows to overlap, so a single element may contribute to multiple windows [1]. Session windows are unaligned and data-dependent: they group elements into sessions separated by periods of inactivity, making them well suited for modeling bursty behavior like user interactions [1].

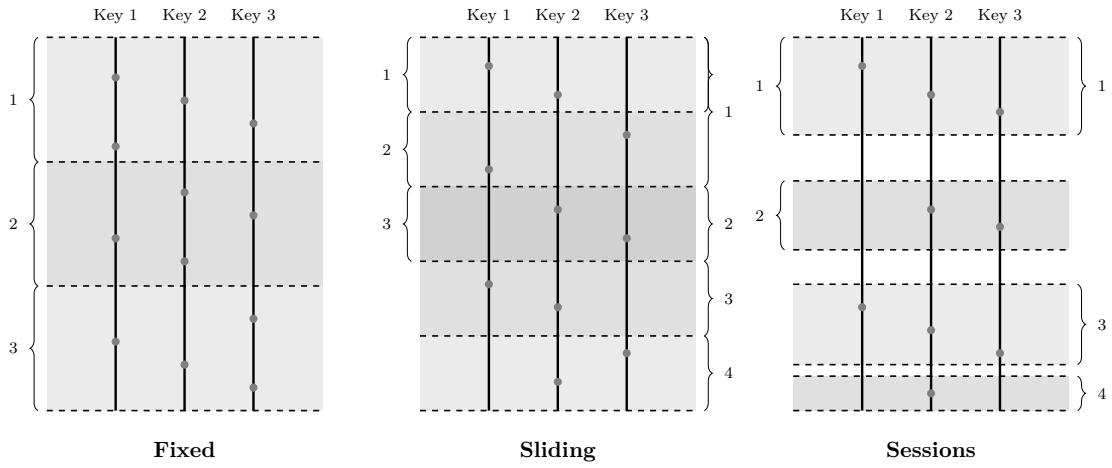

Figure 2.1: Common windowing strategies in stream processing: fixed (tumbling) windows, sliding windows, and session windows. Each window groups elements by event time and key, enabling finite computation over unbounded streams. Figure inspired by Akidau et al. [1].

As shown in Figure 2.1, windowing is orthogonal to grouping by key. Each key maintains its own set of windows, and aggregation is performed independently for each key-window pair [1]. This separation allows windowing to be combined with different grouping strategies [6].

Window assignment is independent of result emission. The presence of a window does not determine when its results should be produced. This distinction motivates triggers, which

control when partial or final results are emitted as data arrive and event-time progress advances [1].

2.1.5. Triggers and Incremental Results

In unbounded and out-of-order data streams, determining when a window has received all relevant data is inherently uncertain. The Dataflow Model addresses this through *triggers*, which define when the system should emit results for a given window [1].

Triggers operate in processing time and may be based on different criteria: event-time progress, processing-time delays, or data-driven conditions [1]. Rather than producing a single output per window, a pipeline may emit multiple results over time as more data arrive. These results represent successive refinements of the computation, improving accuracy as late events are incorporated [1, 6].

To support this incremental behavior, the Dataflow Model defines different accumulation semantics. In discarding mode, each triggered result is independent of previous ones. In accumulating mode, successive results build upon earlier outputs, refining them as new data arrive. In more complex scenarios, retractions explicitly negate previously emitted results before publishing updated values [1].

This flexible triggering and accumulation mechanism lets developers control the trade-off between latency and correctness [1, 6].

2.1.6. Separation of Logical and Physical Execution

A core principle of the Dataflow Model is the explicit separation between the *logical description* of computation and its *physical execution* [1]. The logical layer specifies what is being computed, how data are grouped in event time, and how results evolve through windowing and triggering. The physical layer determines how this computation is mapped onto execution engines and hardware resources [1, 6].

This separation allows the same logical pipeline to run under different performance and cost constraints without modification [1]. For example, a pipeline defined with event-time windows and triggers may be executed using batch processing, micro-batching, or continuous streaming, depending on system capabilities [6].

This abstraction also enables integration of heterogeneous execution mechanisms. As long as an execution strategy can be encapsulated within the logical semantics of operators and windows, it can be incorporated without altering the programming model [1].

2.1.7. Implications for Hardware Acceleration

While the Dataflow Model provides a powerful abstraction for reasoning about unbounded data, its logical–physical separation also creates challenges when integrating hardware accelerators. Devices like GPUs impose specific constraints on batching, data layout, and execution granularity that are not represented in the logical model.

GPUs favor coarse-grained, data-parallel workloads and incur non-trivial overheads for data transfers between host and device [14]. These characteristics may conflict with the fine-grained, latency-sensitive execution patterns typical of stream processing pipelines.

Prior work on heterogeneous stream processing has identified similar challenges. The SABER system [16] showed that achieving high performance on hybrid CPU–GPU stream processing requires careful coordination of several concerns: (1) efficient window-based batching to amortize GPU kernel launch overhead, (2) explicit management of data transfers between host and device memory, (3) query-specific decisions about which operators to run on the CPU versus the GPU, and (4) pipelining of data movement and computation to hide transfer latencies. SABER demonstrated that naïve offloading of operators to the GPU often results in *worse* performance than CPU-only execution due to these overheads [16].

These challenges motivate this thesis: assessing to what extent GPU execution can be embedded within a dataflow framework while preserving its abstractions and execution semantics. The next section introduces the Renoir dataflow platform, which serves as the concrete system used to evaluate these trade-offs.

2.2. The Renoir Dataflow Platform

Renoir is a dataflow-based stream processing framework designed for efficient execution of streaming applications while minimizing programming complexity [10]. It provides a high-level operator abstraction that lets developers express computations declaratively as dataflow pipelines, delegating execution concerns like scheduling, buffering, and parallelism to the runtime [10].

Renoir follows a dataflow design where the user specifies a logical pipeline through operator composition, while execution details are handled by the runtime [10]. This separation between logical description and physical execution is consistent with the Dataflow Model [1]. Renoir emphasizes composability and low-overhead execution, making it a suitable platform for exploring extensions to the dataflow execution model [10].

2.2.1. Programming Model

From the user's perspective, Renoir programs are constructed by composing operators into a directed acyclic graph [10]. Each operator consumes input elements, applies a transformation, and produces output elements. Common operators include element-wise transformations like `map` and `filter`, as well as aggregation operators that process elements over finite windows [10].

A key characteristic of Renoir's programming model is that operators are defined in terms of logical transformations over stream elements, without requiring the programmer to explicitly manage parallelism, data partitioning, or buffering [10]. This design mirrors the Dataflow Model philosophy, where the logical structure of computation is decoupled from execution details [1].

Operator composition in Renoir enables building complex processing pipelines from simple building blocks [10]. Each operator forms a well-defined boundary in the dataflow graph, which plays a central role in reasoning about execution and in extending the system with new execution mechanisms [10].

2.2.2. Execution Model and Runtime Architecture

Internally, Renoir executes dataflow pipelines by orchestrating data flow between operators through a runtime system [10]. Operators may be executed concurrently, and intermediate data are buffered to decouple producers from consumers [10]. This buffering allows the runtime to smooth variations in processing rates and exploit parallelism across operators and data partitions [10].

Renoir adopts a push-based execution model, where operators actively propagate data downstream as elements become available [10]. The runtime schedules operator execution and determines when buffered data should be processed [10]. These decisions are made transparently to the user and are tuned to balance throughput and latency [10].

The runtime manages batching implicitly. While operators conceptually process streams element by element, in practice elements are grouped into batches to amortize scheduling overhead and improve cache locality [10]. This implicit batching is an implementation detail that preserves the abstraction of per-element processing while enabling efficient CPU execution [10].

However, this implicit batching—designed to improve CPU cache locality and amortize scheduling overheads—typically operates at batch sizes suitable for L1/L2 cache utiliza-

tion (hundreds to thousands of elements). These sizes are often orders of magnitude smaller than those required to efficiently saturate GPU execution (tens of thousands to millions of elements), highlighting a basic mismatch between CPU-oriented and GPU-oriented batching.

2.2.3. Windowing and State Management

Renoir supports window-based processing for finite computation over unbounded streams [10]. Windowing lets aggregation operators group elements according to temporal boundaries, following the Dataflow Model [1]. Window state is maintained internally by the runtime and updated incrementally as new elements arrive [10].

Stateful operators in Renoir encapsulate their internal state, ensuring that state updates are local to the operator and consistent with dataflow semantics [10]. This design simplifies reasoning about correctness and isolates state management from the rest of the pipeline [10]. The runtime allocates, accesses, and eventually discards state associated with windows and operators [10].

Window state and buffering handling is particularly relevant when considering alternative execution backends. Decisions about when data are buffered, how much data are accumulated, and when state is accessed directly influence execution granularity and performance. When considering GPU execution, this encapsulation introduces complexity: operator state resides in host memory and is not directly accessible from the device. Moving or replicating state on the GPU requires explicit data transfers and careful synchronization, posing a challenge for stateful GPU operators within a dataflow framework.

2.2.4. Renoir as an Evaluation Platform

In this thesis, Renoir is used as an evaluation platform to study GPU integration into a dataflow-based stream processing system [10]. Renoir is chosen for its clear operator abstraction, its explicit separation between logical computation and execution, and its emphasis on minimizing complexity for developers [10].

By extending Renoir with a GPU-enabled operator, we can investigate how GPU execution interacts with existing dataflow abstractions without redesigning the framework from scratch. This approach lets us isolate the effects of GPU integration on batching, buffering, and operator semantics, and evaluate whether GPU execution can be encapsulated within the dataflow model.

At the same time, using Renoir as a concrete system exposes practical constraints that are

often hidden in purely conceptual models. Implementation choices well suited for CPU execution, such as implicit batching and record-oriented data representation, may conflict with efficient GPU execution requirements. Understanding these interactions is essential for assessing the feasibility and limitations of seamless GPU integration.

2.2.5. Implications for Accelerator Integration

Renoir's architecture offers both opportunities and challenges for hardware acceleration [10]. On one hand, the operator abstraction provides a natural boundary at which GPU execution can be introduced, enabling compute-intensive transformations to be offloaded to an accelerator. On the other hand, the implicit batching and buffering strategies employed by the runtime are not necessarily aligned with the coarse-grained execution preferred by GPUs.

As a result, integrating GPU execution into Renoir cannot be treated as a purely transparent optimization. It requires explicit design decisions about batching, data layout, and execution synchronization, while still preserving the logical semantics expected by the dataflow model.

These considerations motivate the design and implementation presented in the next chapter, where we introduce a GPU-enabled operator for Renoir and describe how GPU execution is embedded within the dataflow framework.

2.3. GPU Programming Model

Graphics Processing Units (GPUs) are throughput-oriented accelerators designed to execute a large number of operations in parallel [18]. Unlike CPUs, which are optimized for low-latency execution of complex and irregular control flows, GPUs are optimized for data-parallel workloads where the same operation is applied to many independent data elements [18]. This difference in execution model has significant implications for how GPU computation can be integrated into stream processing frameworks.

In GPU-accelerated systems, the CPU and its main memory are commonly called the *host*, while the GPU and its on-device memory are called the *device*.

Figure 2.2 illustrates this difference in design philosophy. CPUs dedicate most of their transistor budget to control logic and cache hierarchies, optimizing for low-latency execution of complex, often irregular workloads. GPUs allocate most die area to arithmetic units organized into Streaming Multiprocessors (SMs), maximizing throughput for data-parallel computation at the cost of per-thread sophistication.

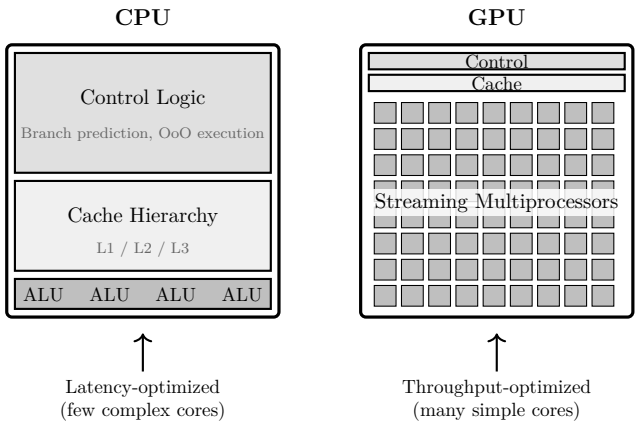

Figure 2.2: Schematic comparison of CPU and GPU die area allocation. CPUs devote most transistor budget to control logic and cache hierarchies to minimize latency for sequential workloads. GPUs allocate most die area to arithmetic units organized into Streaming Multiprocessors, maximizing throughput for data-parallel computation [18].

2.3.1. SIMT Execution and Warp-Based Scheduling

GPU execution is based on the Single Instruction, Multiple Threads (SIMT) model, which extends the classical Single Instruction, Multiple Data (SIMD) paradigm [18]. While traditional SIMD architectures (such as CPU vector units) apply a single instruction to a fixed-width vector of data elements, SIMT presents a more flexible programming abstraction: a large number of lightweight threads appear to execute independently, each with its own program counter and register state [18]. At the hardware level, however, these threads are grouped and executed in lockstep, with each thread operating on different data.

From the programmer's perspective, SIMT appears as a massive collection of independent threads. Internally, threads are grouped into fixed-size units called *warps* (NVIDIA terminology) or *wavefronts* (AMD terminology) [22]. On NVIDIA GPUs, a warp consists of 32 threads that execute instructions in lockstep; AMD GPUs use wavefronts of 32 or 64 threads depending on the architecture [22].

Figure 2.3 illustrates this execution hierarchy. A GPU kernel launch creates a *grid* of thread blocks. Each block is assigned to a Streaming Multiprocessor, where its threads are subdivided into warps for lockstep execution.

This warp-based execution has important performance implications. When all threads in a warp follow the same control flow path, execution proceeds efficiently—this is where SIMT behaves similarly to SIMD. However, when threads within a warp take different branches

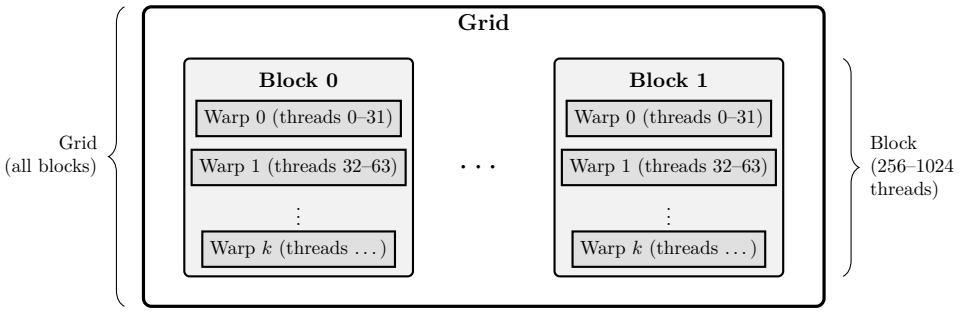

Figure 2.3: GPU execution hierarchy. A kernel launch creates a *grid* of thread *blocks*. Each block contains a fixed number of threads (typically 256–1024), which are organized into *warps* of 32 threads that execute instructions in lockstep on a Streaming Multiprocessor [22].

(a condition called *warp divergence* or *branch divergence*), the hardware must serialize the divergent paths: threads that take one branch execute while the others are masked, then the process repeats for the alternative branch [18, 22]. This serialization can significantly reduce effective parallelism and throughput. As a result, GPU performance is maximized when threads follow uniform execution paths and perform regular operations [18].

Figure 2.4 illustrates this serialization. When a warp encounters a conditional branch where threads follow different paths, the hardware executes each path sequentially while masking the inactive threads, effectively halving throughput for the divergent region.

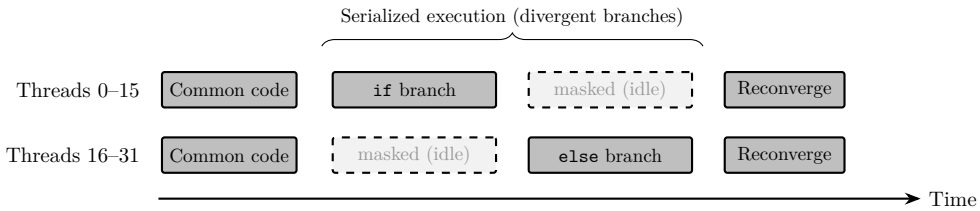

Figure 2.4: Branch divergence within a warp. When threads follow different control flow paths, the hardware serializes execution: threads taking the `if` branch execute while others are masked, then the `else` branch executes. This effectively halves throughput for the divergent region [18, 22].

The key distinction between SIMD and SIMT is that SIMT provides the *abstraction* of independent threads with divergent control flow (even though divergence incurs performance penalties), whereas traditional SIMD requires the programmer to explicitly manage vector operations. This abstraction simplifies GPU programming but requires awareness of warp-level execution to achieve optimal performance.

Figure 2.5 summarizes this distinction. In traditional SIMD, a single instruction operates

on a fixed-width vector register with no independent control flow per lane. In SIMT, each thread maintains its own program counter and register state, presenting an abstraction of independent execution despite hardware-level lockstep within each warp.

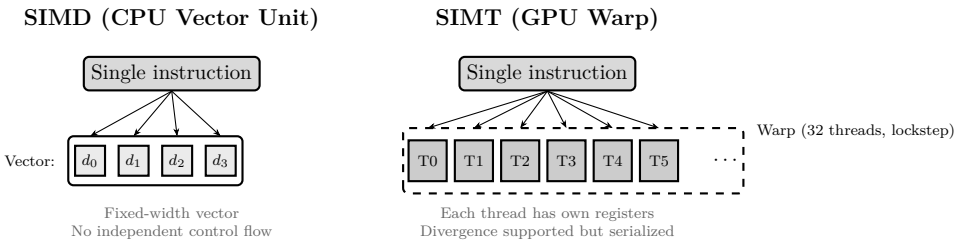

Figure 2.5: Comparison of SIMD and SIMT execution models. In SIMD, a single instruction operates on a fixed-width vector register. In SIMT, each thread has its own register state and program counter, providing an abstraction of independent execution, although threads within a warp execute in lockstep at the hardware level [18].

This execution model contrasts with CPU-based parallelism, which typically relies on a smaller number of heavyweight threads capable of executing truly independent control flows [18]. While CPUs excel at irregular, branch-heavy workloads, GPUs achieve high throughput when computation can be expressed as uniform operations over large collections of data elements [18].

2.3.2. Data Access Patterns and Memory Coalescing

Data access patterns play a central role in GPU performance [26]. GPUs are designed to achieve high memory throughput when threads access contiguous memory regions, allowing memory requests to be combined into fewer transactions [26]. This behavior, called *memory coalescing*, is essential for efficient execution [26].

Irregular or scattered memory accesses prevent effective coalescing and can significantly degrade performance [26]. GPU programs often rely on data layouts such as structure-of-arrays representations, which allow consecutive threads to access consecutive memory locations [26]. In contrast, record-oriented layouts commonly used in stream processing may result in inefficient access patterns on the GPU.

Figure 2.6 illustrates three common memory access patterns and their impact on memory transaction efficiency.

These constraints directly influence how stream elements should be buffered and laid out when targeting GPU execution. While dataflow systems naturally operate on structured records, efficient GPU execution often requires transforming data into layouts that better

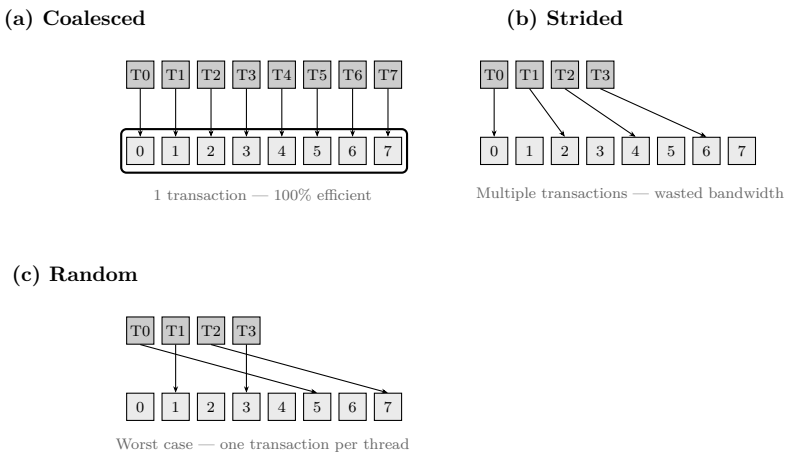

Figure 2.6: Memory coalescing patterns. (a) Coalesced: consecutive threads access consecutive addresses, enabling a single wide memory transaction. (b) Strided: threads access every k-th address, requiring multiple transactions and wasting bandwidth. (c) Random: scattered accesses may require one transaction per thread [26].

match the GPU memory model [25].

2.3.3. Data Layouts: Array of Structures and Structure of Arrays

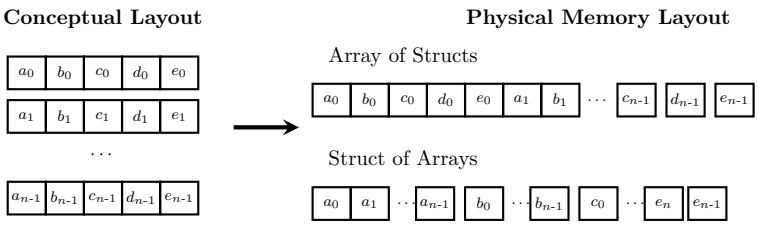

Figure 2.7: Conceptual and physical memory layouts for Array of Structures (AoS) and Structure of Arrays (SoA). In AoS, all fields of one record $(a_i, b_i, \ldots, e_i)$ are stored contiguously; in SoA, each field is stored in its own contiguous array $(a_0, a_1, \ldots)$. SoA enables coalesced memory access on GPUs because consecutive threads access consecutive addresses within the same field array. Figure inspired by Pennycook et al. [25].

The layout of data in memory significantly impacts performance, especially on GPUs where memory access patterns determine the effectiveness of memory coalescing [25]. Two common data layouts are the *Array of Structures* (AoS) and the *Structure of Arrays* (SoA).

As shown in Figure 2.7, in an AoS layout, each element is represented as a structure

containing all its fields, and elements are stored consecutively in memory [25]. This layout is natural for object-oriented programming and is commonly used in stream processing systems, where each record represents a logical event with multiple attributes. AoS layouts are well suited for CPU execution, as hardware-managed caches can efficiently exploit spatial locality even when only a subset of fields is accessed [25].

In contrast, a SoA layout stores each field of a record in a separate, contiguous array [25]. This organization allows consecutive threads to access consecutive memory locations when operating on the same field across multiple elements. As a result, SoA layouts are well suited for GPU execution, as they enable memory coalescing and maximize effective memory bandwidth [25, 26].

The choice between AoS and SoA reflects a trade-off between programming convenience and execution efficiency [25]. While AoS layouts align naturally with the record-oriented abstractions of dataflow systems, SoA layouts are often required to achieve high GPU performance [26]. In GPU-accelerated stream processing, this mismatch frequently requires an explicit data layout transformation as part of the batching process [25].

Such transformations introduce additional overhead and complexity, but they are often unavoidable when targeting data-parallel accelerators [25]. This tension between abstraction-friendly data representations and accelerator-friendly memory layouts is a recurring theme in GPU integration into high-level dataflow frameworks [16].

2.3.4. Memory Hierarchy, Shared Memory, and Occupancy

Beyond global memory access patterns, the GPU memory hierarchy differs fundamentally from that of a CPU [18]. While CPUs rely primarily on deep, hardware-managed cache hierarchies to minimize memory access latency, GPUs emphasize latency hiding over latency reduction [18]. Figure 2.8 illustrates this hierarchy.

GPUs maintain a large number of concurrently active threads. When one group of threads stalls due to a memory access, the hardware scheduler can rapidly switch execution to another group that is ready to run [18]. This strategy allows GPUs to tolerate the high latency of off-chip global memory without requiring large caches [18]. Figure 2.9 illustrates this mechanism: when one warp stalls on a memory access, the scheduler immediately switches to another ready warp, keeping the execution units occupied.

In addition to global memory, GPUs expose a fast, on-chip memory that can be explicitly managed by the programmer [26]. This memory is called *shared memory* in NVIDIA CUDA terminology, *local memory* in OpenCL, and *threadgroup memory* in Metal and We-

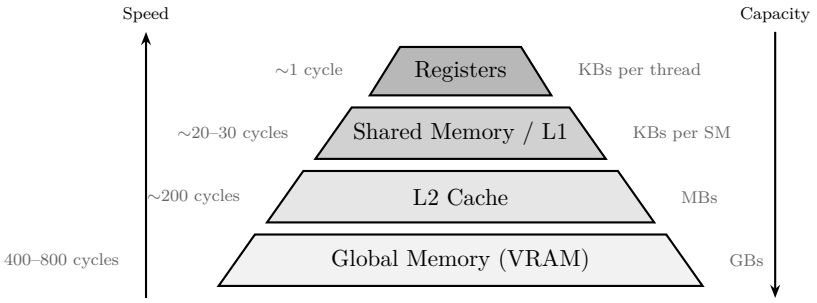

Figure 2.8: GPU memory hierarchy. Faster memory tiers are smaller and private to individual threads or thread blocks. Slower tiers provide larger capacity accessible by all threads. Effective GPU programming requires managing data placement across this hierarchy to minimize high-latency global memory accesses [18, 26].

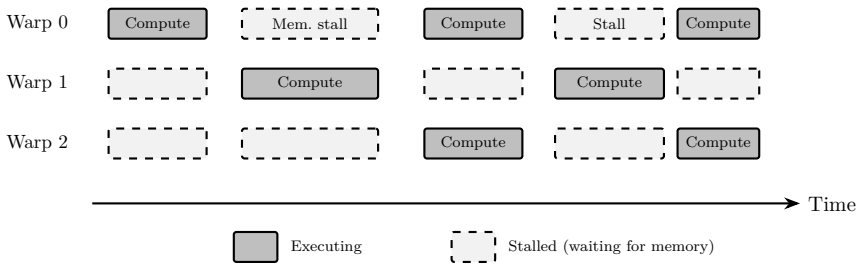

Figure 2.9: Latency hiding through warp-level scheduling on a single Streaming Multiprocessor. When a warp stalls on a memory access, the hardware scheduler switches to another ready warp, keeping the execution units occupied. With sufficient active warps, memory latency is effectively hidden [18].

bGPU [22]. Regardless of terminology, this memory enables low-latency communication and data reuse among threads within the same execution group [26]. Effective use of this memory hierarchy is often essential for achieving high performance in compute-intensive GPU kernels [26].

A related concept is *occupancy*, defined as the ratio of active warps on a streaming multiprocessor (SM) to the maximum number of warps the SM can support [29]. Higher occupancy generally improves the GPU's ability to hide memory latency by providing more warps to switch between when one stalls. However, Volkov [29] showed that higher occupancy does not always yield better performance: in compute-bound kernels, a smaller number of threads with more registers per thread can outperform configurations with higher occupancy but fewer resources per thread. This insight is relevant for kernel optimization but is typically beyond the scope of generic dataflow operators.

In practice, many GPU integrations in dataflow systems favor simpler execution strategies based on global memory access and coarse-grained batching, trading some peak performance for generality and robustness. Exploiting shared memory and tuning occupancy effectively requires detailed knowledge of access patterns, data reuse, and synchronization behavior that cannot be reliably inferred from generic stream processing operators [26].

2.3.5. Batching and Kernel Launch Overheads

GPU computation is organized into *GPU kernel* executions, where a kernel defines the computation applied to a batch of data elements [18]. (Note: In this thesis, we distinguish between "GPU kernels"—the device-side programs executed on the GPU—and the `GpuKernel` trait, which is the Rust interface through which user code defines GPU behavior in Renoir.) Launching a GPU kernel incurs a non-trivial overhead, independent of the amount of work performed [18]. As a result, executing kernels on very small batches often leads to poor performance, as overhead dominates computation time [18].

Batching is therefore a fundamental requirement for efficient GPU execution [18]. By grouping multiple elements into a single kernel invocation, launch overhead can be amortized across many parallel operations [18]. Larger batches typically improve GPU resource utilization, up to the point where other factors like memory capacity or data transfer costs become limiting [14].

This requirement contrasts with the fine-grained execution model of stream processing systems, where operators conceptually process elements as they arrive. Bridging this mismatch requires explicit buffering strategies that accumulate elements into batches before offloading computation to the GPU [16].

2.3.6. Data Transfers and the PCIe Bottleneck

In most systems, GPUs are connected to the host CPU through an interconnect such as PCI Express (PCIe) [14]. Data transfers between host memory and GPU memory must traverse this interconnect, which has significantly lower bandwidth and higher latency than on-device memory accesses [14].

To quantify this bottleneck: PCIe 3.0 x16 provides a theoretical peak bandwidth of approximately 16 GB/s, while PCIe 4.0 x16 doubles this to approximately 32 GB/s [14]. In contrast, modern GPU memory systems (such as HBM2) provide bandwidths exceeding 900 GB/s—roughly 30–60× higher than the host–device interconnect. This disparity means that data-intensive workloads can easily become transfer-bound rather than compute-bound.

For GPU-accelerated stream processing, these transfer costs are critical [14]. When batches are small, the time spent transferring data to and from the GPU may exceed the time spent performing the computation itself, eliminating any potential speedup [14]. As batch sizes increase, transfer overheads can be amortized, making GPU execution advantageous [14].

This behavior explains the threshold effect commonly observed in GPU acceleration: below a certain batch size, GPU execution may be slower than CPU execution, while above that threshold, substantial speedups can be achieved [14].

2.3.7. Asynchronous Execution and Overlap

To mitigate the PCIe bottleneck, modern GPUs support asynchronous execution mechanisms that allow computation and communication to overlap [14]. GPUs can execute kernels concurrently with memory transfers, provided these operations are issued to independent hardware queues.

In programming frameworks such as CUDA, these queues are commonly called *streams*. By organizing kernel launches and data transfers into multiple streams, it is possible to construct a pipeline where different stages proceed concurrently. In a streaming context, this enables a pattern where batch N is processed on the GPU while batch $N + 1$ is transferred to the device and the results of batch $N - 1$ are transferred back to the host.

Achieving effective overlap requires careful synchronization and the use of pinned (page-locked) host memory to avoid additional copying overheads. These requirements add complexity to the runtime system and complicate the integration of asynchronous execu-

tion into high-level dataflow frameworks [16].

2.3.8. Implications for Dataflow Integration

The GPU programming model imposes constraints that are not explicitly represented in traditional dataflow abstractions [16]. Efficient execution requires control over batching, data layout, memory transfers, and synchronization, while dataflow frameworks aim to hide such low-level concerns behind operator abstractions.

As a result, integrating GPU execution into a dataflow system involves a fundamental trade-off. Excessive abstraction may lead to inefficient execution, while exposing too many GPU-specific details undermines the simplicity and composability of the programming model [16]. Similar challenges have been identified in prior work on heterogeneous stream processing systems, which require explicit coordination between CPU and GPU execution to achieve high performance [16].

Understanding these constraints is essential for evaluating to what extent GPU programming can be embedded seamlessly within a dataflow execution model. The next chapter builds on this background to present the design and implementation of a GPU-enabled operator for the Renoir platform.

3 | Design and Implementation

3.1. Design Overview and Problem Decomposition

This chapter presents the design and implementation of GPU execution support in the Renoir dataflow platform. Rather than introducing GPU awareness at the level of the runtime scheduler or execution engine, GPU support is realized by extending the operator abstraction itself. This choice reflects both the architectural structure of Renoir and the practical constraints encountered during implementation.

The goal of the design is not to transparently accelerate arbitrary dataflow programs, but to enable GPU execution in a controlled and explicit manner, while preserving the semantics and composability of the existing dataflow model. This encapsulation strategy is illustrated in Figure 3.1, which shows how GPU-specific concerns are confined within the `MapGpu` operator while preserving the standard Renoir dataflow interface.

3.1.1. Problem Setting

At the outset of this work, Renoir provided no native mechanism for GPU execution. The Renoir platform, as described by De Martini et al. [10], presents a CPU-oriented runtime and operator pipeline; the published description does not discuss GPU execution mechanisms or any GPU operator model. All operators were designed under the assumption of CPU-based processing, with execution driven by fine-grained, record-oriented dataflow semantics. The runtime managed scheduling, buffering, and parallelism for CPU operators, but offered no facilities for handling device memory, asynchronous execution, or host–device data transfers.

As a result, it was not possible to directly express GPU-based computation within a Renoir dataflow pipeline. In particular, there was no support for:

- coordinating stream-level batching decisions with GPU execution,
- transferring data between host and device memory,

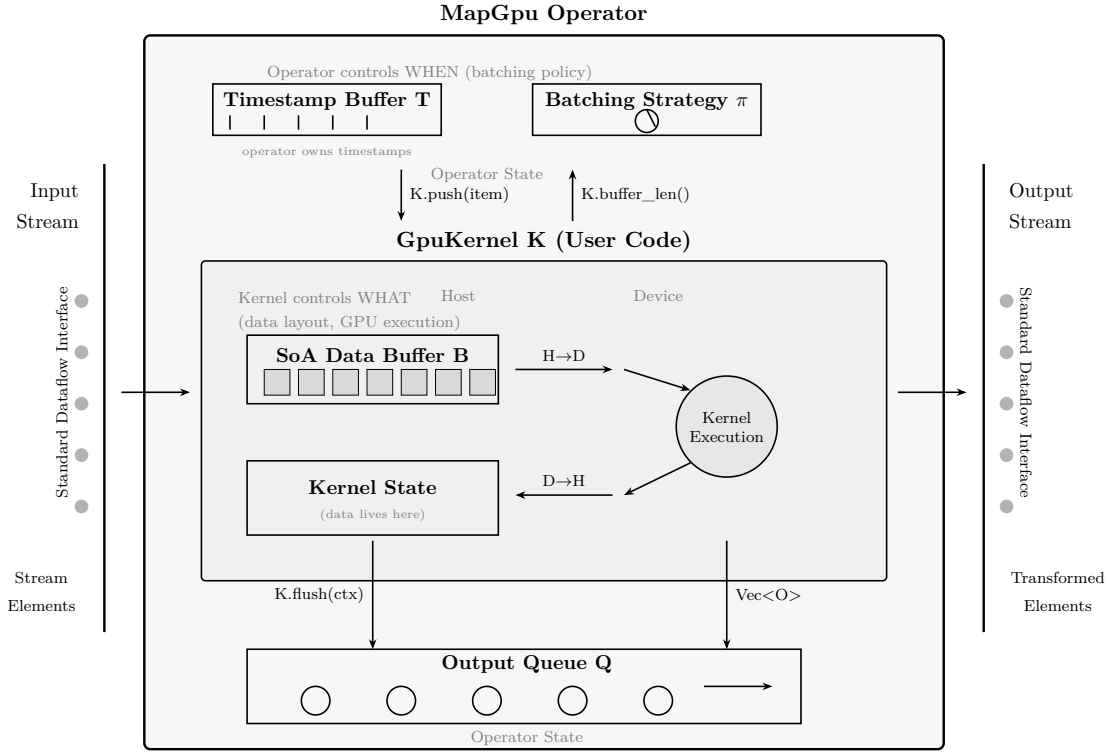

Figure 3.1: High-level architecture of the `MapGpu` operator. The outer box represents the `MapGpu` operator, which manages stream-level control: the *Timestamp Buffer T* records event-time metadata for each buffered element, and the *Batching Strategy* π determines when to trigger GPU execution. The inner box represents the user-defined `GpuKernel K`, which owns the actual data: the *SoA Data Buffer B* stores elements in a GPU-friendly layout, and the *Kernel State* holds any persistent state needed across flushes. Arrows `K.push(item)` and `K.buffer_len()` show the incremental data ingestion and occupancy query interface between operator and kernel. On flush, the kernel transfers data from host to device (H→D), executes the GPU compute kernel, and transfers results back (D→H). Results are returned as `Vec<O>` and placed into the *Output Queue Q*, from which the operator emits them downstream one at a time through the standard dataflow interface.

- invoking GPU compute kernels asynchronously,
- synchronizing GPU execution with downstream operators, or
- integrating GPU execution without violating stream semantics.

These limitations motivated the introduction of a new execution mechanism that could accommodate the GPU programming model while remaining compatible with Renoir's existing abstractions.

3.1.2. Rationale for an Operator-Level Integration

GPU execution is introduced in Renoir by extending the operator abstraction, rather than by modifying the global runtime or scheduler. This decision is driven by the observation that operators already form natural boundaries for computation within a dataflow graph. Each operator encapsulates a transformation, controls the flow of elements between upstream and downstream stages, and provides a well-defined location at which execution policies can be applied.

Embedding GPU execution at the operator level enables GPU-specific concerns to be localized within a single component. This avoids invasive changes to the runtime system and preserves the separation between logical computation and execution strategy. From the user's perspective, GPU execution becomes an explicit variant of an existing operator (analogous to `map`), rather than a cross-cutting concern that affects the entire pipeline.

Following this approach, GPU support is realized through the introduction of a new operator, `map_gpu`, which applies a user-defined GPU compute kernel to a stream of elements in a batched manner.

A key design decision is that the framework does not attempt to automatically generate or infer GPU compute kernels. Kernel behavior, internal buffering, memory access patterns, and data layout choices are inherently application-specific and cannot be derived reliably from the high-level dataflow description alone. Requiring developers to explicitly define GPU compute kernels preserves expressiveness, avoids leaking GPU-specific constraints into the dataflow model, and clearly delineates responsibility between the framework and user code.

3.1.3. Fundamental Constraints Observed During Implementation

The design of `map_gpu` is shaped by several constraints imposed by the GPU execution model and observed during implementation:

- **Batching is mandatory**: GPU compute kernels incur non-negligible launch overhead and require sufficiently large batches to achieve acceptable utilization.
- **Batch size is runtime-dependent**: Optimal batch sizes depend on kernel complexity, data layout, and transfer costs, and cannot be determined statically.
- **Asynchronous execution**: GPU compute kernel execution and memory transfers proceed asynchronously with respect to the host, requiring explicit synchronization.
- **Data transfer overheads**: Host–device transfers dominate execution time for small batches and must be amortized over larger workloads.
- **Data layout mismatch**: Renoir's record-oriented data representation does not directly map to GPU-friendly memory layouts.

While batching is unavoidable, the specific batching strategy may vary depending on application requirements, kernel characteristics, and performance trade-offs. The core challenge is *temporal mismatch*: stream elements arrive individually and asynchronously from the Renoir runtime, but efficient GPU execution requires launching a kernel over a large, contiguous batch of data. The operator must therefore introduce an internal buffer that accumulates incoming elements until a flush condition is met—either a count threshold ($|B| \geq n$) or a time-based trigger ($\Delta t \geq \tau$). Only then is the buffer flushed and a GPU kernel dispatched.

These constraints are not artifacts of a particular implementation choice, but fundamental properties of GPU execution. Consequently, they must be explicitly addressed by the operator design.

3.1.4. Design Consequences

The constraints outlined above lead to several concrete design requirements for GPU integration in Renoir. First, the operator must introduce explicit *stream-level control* to determine when GPU execution should be triggered. As illustrated in Figure 3.1, the operator maintains auxiliary metadata such as timestamps and batching state, but does not own the actual data buffers used for computation.

Instead, stream elements are incrementally passed to the user-defined `GpuKernel` implementation, which is responsible for maintaining its own internal data buffers (e.g., Structure-of-Arrays layouts) and reporting their occupancy to the operator. The operator queries this state (e.g., via `buffer_len()`) to decide when to invoke `flush()`, thereby decoupling batching decisions from data representation.

Second, data ingestion must be separated from kernel invocation to allow asynchronous execution and overlap between computation and communication. Third, the operator must manage synchronization points to ensure correct interaction with downstream operators and preserve stream semantics.

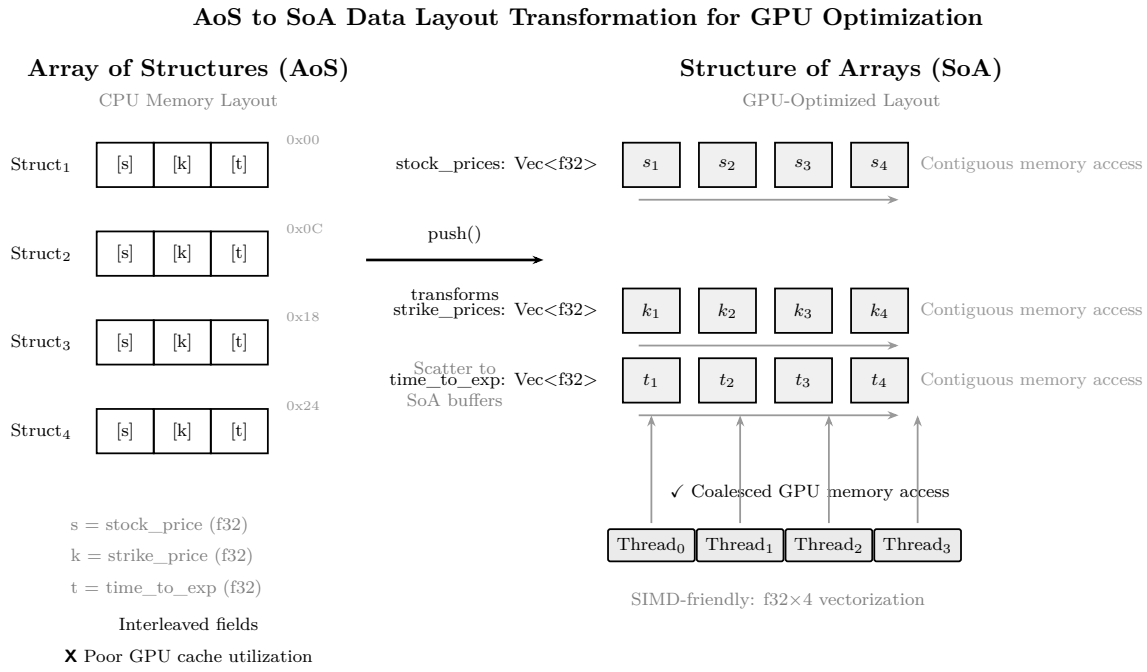

Figure 3.2: Transformation from record-oriented Array-of-Structures (AoS) layout to GPU-friendly Structure-of-Arrays (SoA) layout, implemented inside the user-defined `GpuKernel` during batch accumulation. In AoS layout (left), fields of different records are interleaved in memory, resulting in strided access patterns and poor cache utilization on GPUs. The `push()` method scatters each incoming record's fields into separate contiguous arrays (right), enabling coalesced memory access where consecutive GPU threads read consecutive addresses within the same field array.

Importantly, these mechanisms must be implemented without altering the logical behavior of the dataflow pipeline. From the perspective of upstream and downstream operators, `map_gpu` behaves as a deterministic transformation over the stream, despite internally operating on batched and asynchronously executed data.

3.1.5. Scope and Limitations

The design presented in this chapter deliberately limits the scope of GPU integration. GPU execution is confined to a single operator and does not involve global scheduling, cross-operator batching, or transparent acceleration of arbitrary pipelines. Developers are required to explicitly opt into GPU execution and provide the corresponding `GpuKernel` implementation.

To further improve throughput, the operator pipelines successive batches so that the three stages of GPU execution—host-to-device transfer, kernel computation, and device-to-host transfer—overlap across consecutive batches. Rather than processing each batch sequentially (upload, compute, download, then repeat), the operator begins uploading the next batch while the current batch is still being computed, and starts downloading results while the next upload proceeds. This three-stage pipeline, illustrated in Figure 3.3, amortizes transfer latency behind computation and ensures that the GPU is rarely idle between batches.

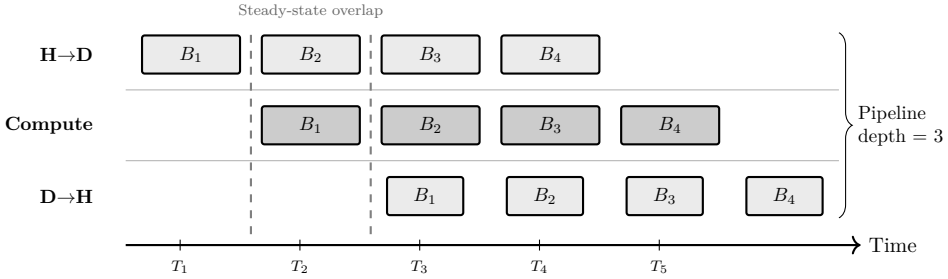

Figure 3.3: Asynchronous execution pipeline with three-stage overlap. Each batch B_i passes through three stages: host-to-device transfer (H→D), GPU kernel execution (Compute), and device-to-host transfer (D→H). By the time batch B_1 reaches the Compute stage, the upload of B_2 can begin concurrently; similarly, the download of B_1 overlaps with the computation of B_2 and the upload of B_3. In the steady state (between the dashed vertical lines), all three pipeline stages are active simultaneously, achieving a pipeline depth of 3. This overlap hides transfer latency behind computation: on discrete GPU systems, PCIe transfer time is amortized across batches; on unified memory architectures, the analogous memory-mapping and synchronization overhead is similarly overlapped. The net effect is that throughput is bounded by the slowest pipeline stage rather than by the sum of all three.

This scoped approach prioritizes clarity, correctness, and faithful integration with Renoir's dataflow semantics over maximal transparency or automation. Within these boundaries, it provides a practical foundation for evaluating how GPU execution can be embedded

into a dataflow framework.

A note on host–device terminology. Throughout this chapter, we use the standard "host" and "device" terminology and discuss data transfers, PCIe bottlenecks, and asynchronous overlap as general GPU programming concerns. This reflects the design's intent to support both discrete and integrated GPU architectures. However, the evaluation platform (Chapter 4) uses an Apple M2 Pro with a unified memory architecture, where CPU and GPU share the same physical memory pool. On such platforms, "host-to-device transfers" are replaced by memory-mapped operations, and the PCIe bottleneck is absent. The design remains applicable—the batching, layout conversion, and pipelining mechanisms are structurally identical—but the measured overhead characteristics are platform-dependent and represent a favorable case relative to discrete GPU systems.

The following sections describe the architecture, execution flow, and implementation details of the `map_gpu` operator.

3.2. Programming Model and API

This section introduces the programming model exposed to users for GPU execution in Renoir. Rather than focusing on internal implementation details, the discussion adopts the perspective of a developer who wishes to offload a computation to the GPU while remaining within the dataflow abstraction.

The design goal of the programming model is to make GPU execution explicit, structured, and composable, while avoiding the exposure of low-level GPU management concerns such as device memory allocation, kernel scheduling, or synchronization primitives.

Terminology. Throughout this chapter, we adopt Rust naming conventions: `map_gpu` and `map_gpu_with_strategy` refer to the API methods (snake_case), `MapGpu` refers to the operator struct (CamelCase), and `GpuKernel` refers to the trait that users implement (CamelCase). The term "GPU compute kernel" refers to the device-side program executed on the GPU, distinct from the `GpuKernel` Rust trait.

3.2.1. Overview of the `map_gpu` Operator

GPU execution is exposed through a dedicated operator, `map_gpu`, which mirrors the semantics of the standard `map` operator but executes user-defined logic on the GPU. From the perspective of the dataflow pipeline, `map_gpu` behaves as a deterministic transformation that consumes a stream of input elements and produces a stream of output elements.

At the API level, Renoir provides two entry points:

- **`map_gpu(kernel)`**: convenience wrapper that uses a default batching policy.
- **`map_gpu_with_strategy(kernel, strategy)`**: the fully parameterized API that lets the user choose how batching and flushing are performed.

Importantly, `map_gpu` is merely a convenience wrapper around `map_gpu_with_strategy`; the latter is the "real" API that exposes all configuration options. Using `map_gpu` means accepting the default batching policy; applications that require tuning of latency/throughput trade-offs must use `map_gpu_with_strategy` with an appropriate `GpuBatchStrategy`.

In the current implementation, `map_gpu` delegates to `map_gpu_with_strategy`, passing `GpuBatchStrategy::default()`:

Listing 3.1: `map_gpu` delegates to `map_gpu_with_strategy` using a default batching policy.

```
pub fn map_gpu<K>(self, kernel: K) -> Stream<impl Operator<Out = K::Output>>
where
    K: gpu::GpuKernel<Input = Op::Out>,
{
    self.map_gpu_with_strategy(kernel, gpu::GpuBatchStrategy::default())
}
```

The fully parameterized form exposes the batching policy explicitly:

```
pub fn map_gpu_with_strategy<K>(
    self,
    kernel: K,
    strategy: gpu::GpuBatchStrategy,
) -> Stream<impl Operator<Out = K::Output>>
where
    K: gpu::GpuKernel<Input = Op::Out>,
{
    self.add_operator(|prev| gpu::MapGpu::with_strategy(prev, kernel, strategy)
    )
}
```

Listing 3.2: `map_gpu_with_strategy` exposes batching as part of the API.

Parameter: `kernel`. The `kernel` parameter defines the GPU computation applied to stream elements. Rather than passing a function pointer or embedding GPU code inside

the operator, Renoir requires the user to provide an object implementing the `GpuKernel` trait. Conceptually, the `GpuKernel` implementation is responsible for *how* elements are processed on the GPU: it owns the internal data buffers (e.g., a Structure-of-Arrays representation), performs any required data layout transformation, invokes the GPU compute kernel, and materializes results back into host memory. The operator treats the `GpuKernel` as an execution backend and interacts with it through a small interface (e.g., incremental `push()` of items and a `flush()` when execution is triggered).

Parameter: `strategy`. The `strategy` parameter defines *when* the operator should trigger GPU execution. Because GPU compute kernels are most efficient on sufficiently large batches, stream elements are not executed one-by-one; instead, they are accumulated until a *flush condition* is satisfied. A batching strategy encapsulates this flush policy. In the current design, flush conditions can depend on criteria such as reaching a target batch size, a time-based deadline, or adaptive logic that reacts to observed throughput.

Built-in strategies and extensibility. Renoir currently provides three batching strategies:

- `GpuBatchStrategy::Fixed(n)`: flush after exactly n input elements have been accumulated.
- `GpuBatchStrategy::Timed { max_size, interval }`: flush either when the batch reaches `max_size` elements or when the time since the last flush exceeds `interval`, whichever happens first.
- `GpuBatchStrategy::Adaptive { min_size, max_size }`: adapt the flush size dynamically based on observed throughput, while keeping the batch size within the given bounds.

In addition to these built-ins, the design allows developers to implement custom batching policies when application requirements (e.g., strict latency bounds or highly irregular arrival rates) require specialized behavior. Since batching affects both throughput and end-to-end latency, the strategy is treated as a first-class part of the programming model rather than an implicit runtime heuristic. The detailed behavior of these strategies, and the interface required for custom strategies, is discussed in Section 3.6.

3.2.2. Semantic Guarantees and Non-Guarantees

Before describing implementation details, this section explicitly states what the `map_gpu` operator guarantees and what it does not guarantee. This clarification distinguishes

between properties inherited from Renoir's dataflow model and properties that developers must provide or configure.

Table 3.1: Semantic guarantees and non-guarantees of the `map_gpu` operator.

What `map_gpu` GUARANTEES	
Element ordering	Output elements are emitted in the same order as corresponding input elements, regardless of batching or asynchronous execution.
Timestamp preservation	Each output element carries the timestamp of its corresponding input element.
Watermark propagation	Watermarks are correctly forwarded to downstream operators; no out-of-order watermarks are emitted.
Composability	The operator composes with upstream and downstream Renoir operators identically to standard `map`.
One-to-one cardinality	Each input element produces exactly one output element (no filtering or explosion).
What `map_gpu` does NOT guarantee	
Automatic kernel synthesis	Developers must implement the `GpuKernel` trait; GPU code is not derived from closures.
Transparent GPU execution	GPU execution is explicit; only pipelines using `map_gpu` (not `map`) execute on GPU.
Optimal batching	The framework provides batching strategies, but optimal batch size is workload-dependent.
Cross-operator GPU fusion	Each `map_gpu` operator executes independently; kernel fusion across operators is not performed.
Distributed GPU coordination	GPU execution occurs within a single worker; multi-node GPU coordination is not supported.

Relationship to Renoir's execution model. Renoir [10] translates dataflow graphs into stages and tasks that are scheduled by its runtime. The `map_gpu` operator introduced in this thesis operates *within* this model at the operator level: it does not modify Renoir's stage partitioning, task scheduling, or inter-worker communication. GPU execution is entirely encapsulated within the operator's `next()` method, which the Renoir runtime invokes like any other operator. This operator-level integration preserves Renoir's existing execution semantics while adding GPU acceleration as a localized capability.

3.2.3. Kernel-Centric Programming Model

GPU computation in Renoir is expressed by implementing a user-defined `GpuKernel` trait, rather than embedding GPU logic directly inside the operator. This design reflects the fact that efficient GPU execution depends on application-specific decisions—most notably memory layout (e.g., AoS vs. SoA), buffering discipline, and access patterns—that cannot be reliably inferred from a generic dataflow operator without either losing performance or over-constraining the programming model.

The `GpuKernel` trait encapsulates the mechanics of GPU execution. Concretely, a `GpuKernel` implementation is responsible for:

- maintaining internal data buffers (with application-appropriate layouts),
- performing data layout transformations when required,
- launching GPU computation for a flushed batch, and
- retrieving results from the device.

This organization enforces a clean separation of concerns: the `MapGpu` operator (exposed via the `map_gpu` API) retains responsibility for stream-level control—deciding *when* a batch should be flushed and ensuring that output is reintegrated into the pipeline as a regular stream—while the `GpuKernel` implementation defines *how* elements are represented and processed on the GPU. The internal buffering and flush execution path are described in detail in Section 3.4 and Section 3.5.

3.2.4. Execution Context and Backend Selection

Although the `map_gpu` API aims to hide low-level device management, GPU execution still requires an execution *context* that provides access to the underlying backend (device selection, queues, allocators, and compilation facilities). In this design, the context is treated as a construction-time dependency of GPU-enabled components: users initialize a context once and reuse it across `GpuKernel` implementations and operators, while keeping backend-specific details out of the pipeline API.

The implementation achieves backend independence through *CubeCL* [27], an open-source Rust library that provides a unified programming model capable of targeting multiple GPU backends. CubeCL is designed to compile Rust code to GPU compute kernels that can execute on different hardware platforms. Specifically, CubeCL supports:

- A portable `wgpu`-based backend [12], which leverages the WebGPU specification and

can target Vulkan, Metal, DirectX 12, and WebGPU runtimes.

- A CUDA backend [22, 23] for NVIDIA GPUs, providing direct access to CUDA's execution model.

Backend selection is performed at compile time via Cargo feature flags, as documented in the CubeCL crate [28]. This design allows the same `GpuKernel` implementation to run across different platforms without source-level changes, provided the chosen backend supports the target hardware.

3.2.5. Illustrative Usage Example

Listing 3.3 illustrates the typical usage pattern of `map_gpu` within a Renoir pipeline. The example shows the convenience wrapper `map_gpu` (default batching); the explicit `map_gpu_with_strategy` variant is used when batching policy must be tuned.

```
let env = StreamContext::new_local();

let out = env
    .stream_iter((0..1000).map(|x| x as f32))
    .map_gpu(MyGpuKernel::new(ctx.clone()))  // default strategy
    // .map_gpu_with_strategy(MyGpuKernel::new(ctx.clone()),
    //     GpuBatchStrategy::Timed { max_size: 100_000, interval: Duration::
    from_millis(100) })
    .collect_vec();
```

Listing 3.3: Illustrative usage of `map_gpu` in a Renoir pipeline.

3.2.6. Relation to Design Goals

The programming model reflects the design goals established in Section 3.1:

- GPU execution is explicit but localized,
- batching is controlled without breaking stream semantics,
- GPU compute kernel logic is isolated from operator control flow,
- the dataflow abstraction remains intact.

By exposing GPU execution through a dedicated operator and a `GpuKernel`-centric API, the model balances expressiveness with architectural clarity. Users are empowered to write efficient GPU code where needed, while the framework preserves composability and semantic transparency.

Subsequent sections build on this programming model to describe batching strategies, execution flow, and implementation considerations in greater detail.

3.3. MapGpu Operator Execution Flow

This section details the internal execution flow of the `MapGpu` operator, describing how it bridges the gap between Renoir's push-based streaming model and the batched, asynchronous nature of GPU execution. The execution flow must handle several responsibilities: receiving elements from upstream, buffering data until a batch is ready, triggering GPU execution, and emitting results to downstream operators while preserving stream semantics.

3.3.1. Internal Pull-Based Interface

While Renoir's overall execution model is push-based—operators actively propagate data downstream as elements become available [10]—the `MapGpu` operator internally implements a pull-based (demand-driven) interface through its `next()` method. This method returns the next output element when called by the downstream consumer. The pull-based internal interface simplifies backpressure handling within the operator and allows lazy evaluation of batched results, but the operator still receives data from upstream via Renoir's push-based mechanism.

The `MapGpu` operator follows this internal pull-based contract but introduces internal buffering to accommodate batched GPU execution. The key insight is that the operator maintains an *output queue* (`VecDeque`) of processed results. When `next()` is called, the operator first checks this queue and returns an element if available. Only when the queue is empty does the operator pull from upstream, buffer the incoming element, and potentially trigger a GPU flush. This design converts the batch-oriented output of GPU execution into a fine-grained stream compatible with downstream operators.

Algorithm 3.1 presents the pseudocode for the `next()` method, illustrating the priority-based control flow.

3.3.2. Stream Element Handling

The `MapGpu` operator must correctly handle all variants of Renoir's `StreamElement` enumeration. Table 3.2 summarizes the handling of each variant, and the following paragraphs provide additional detail.

Algorithm 3.1 MapGpu Operator `next()` Method

```
1: loop
2:   if output_queue is not empty then
3:     return pop element from output_queue
4:   end if
5:   if pending_watermark exists then
6:     return pending_watermark (and clear it)
7:   end if
8:   if received_flush_restart flag is set then
9:     Clear flag
10:    return FlushAndRestart
11:  end if
12:  if received_terminate flag is set then
13:    return Terminate
14:  end if
15:  element ← upstream.next()
16:  if element is Item(x) or Timestamped(x, ts) then
17:    kernel.push(x)
18:    buffer_timestamps.append(ts or None)
19:    if should_flush() then
20:      flush_to_gpu()
21:    end if
22:  else if element is Watermark(ts) then
23:    max_watermark ← max(max_watermark, ts)
24:  else if element is FlushBatch then
25:    flush_to_gpu()
26:  else if element is FlushAndRestart or Terminate then
27:    flush_to_gpu()
28:    drain_pending_to_gpu()
29:    pending_watermark ← max_watermark
30:    Set appropriate termination flag
31:  end if
32: end loop
```

Table 3.2: Handling of stream element types by the `MapGpu` operator.

Element Type	Operator Behavior
`Item(x)`	Buffer item with no timestamp; check flush condition
`Timestamped(x, ts)`	Buffer item with associated timestamp; check flush condition
`Watermark(ts)`	Track maximum watermark; do not trigger flush
`FlushBatch`	Force immediate GPU execution of buffered data
`FlushAndRestart`	Flush buffer, drain pending results, emit watermark, signal iteration boundary
`Terminate`	Flush buffer, drain pending results, emit watermark, terminate stream

Data elements. When receiving `Item(x)` or `Timestamped(x, ts)`, the operator invokes `kernel.push(x)` to add the element to the `GpuKernel`'s internal buffer. Simultaneously, it records the timestamp (or `None` for untimestamped items) in a parallel `buffer_timestamps` vector. This parallel structure ensures that after GPU execution, each output can be re-associated with its original timestamp, preserving event-time semantics required for windowing operations downstream.

Watermarks. Watermarks signal progress in event time and are essential for triggering window computations [1]. The `MapGpu` operator tracks the maximum watermark seen but does not immediately forward it. Instead, the watermark is emitted *after* the output queue is drained following a flush. This ensures that the watermark reflects the actual progress of data through the GPU processing stage.

Control signals. The signals `FlushBatch`, `FlushAndRestart`, and `Terminate` trigger immediate GPU execution regardless of whether the batching threshold has been reached. For pipelined `GpuKernel` implementations (Section 3.5), these signals also invoke `drain()` to collect any results still pending from the previous batch. This guarantees that no data is lost at iteration or stream boundaries.

3.3.3. Flush Decision Logic

The decision to trigger a GPU flush is evaluated after each element is buffered. The `should_flush()` method implements the following logic:

1. If the `GpuKernel`'s buffer is empty (`buffer_len() == 0`), return `false`.
2. Determine the target batch size from the active strategy:
   - `Fixed(n)`: target = n
   - `Timed`: target = `max_size`
   - `Adaptive`: target = `adaptive_sizer.current_size()`
3. If `buffer_len() >= target`, return `true`.
4. For `Timed` strategy: if `batch_timer.elapsed() >= interval`, return `true`.
5. Otherwise, return `false`.

This approach decouples the batching decision from data representation: the operator knows *when* to flush based on element count and time, while the `GpuKernel` controls *what* data is flushed and *how* it is laid out in memory.

3.3.4. Timestamp Preservation and Correctness

Preserving timestamp associations through batched GPU execution is essential for maintaining correct stream semantics. The challenge is that GPU execution processes elements in bulk, but the dataflow model expects each output to carry its original timestamp.

The operator maintains two timestamp buffers to handle both synchronous and pipelined `GpuKernel` modes:

- `buffer_timestamps`: timestamps for elements currently being accumulated in the `GpuKernel`'s buffer.
- `pending_timestamps`: timestamps for elements whose GPU results are "in flight" (for pipelined `GpuKernel` implementations only).

After a `flush()` call, the operator examines the number of returned results:

- If `results.len() == buffer_timestamps.len()`: synchronous mode; pair results with current timestamps.
- If `results.len() == pending_timestamps.len()`: pipelined mode; pair results with pending timestamps, then move `buffer_timestamps` to `pending_timestamps`.
- If `results.is_empty()` and `buffer_timestamps` is non-empty: first flush of a pipelined `GpuKernel`; store current timestamps as pending.

This automatic detection allows the same operator code to support both pipelined and non-pipelined `GpuKernel` implementations without explicit mode configuration.

3.4. Buffering and Data Layout

GPU execution efficiency depends critically on memory access patterns. Modern GPUs achieve peak memory bandwidth only when adjacent threads access adjacent memory locations—a property known as *memory coalescing* [22, 26]. This section describes how the `MapGpu` operator and the `GpuKernel` trait collaborate to buffer data and transform it into GPU-friendly layouts.

3.4.1. Separation of Buffering Responsibilities

A fundamental design decision is that the `MapGpu` operator does *not* own the data buffers used for GPU computation. Instead, the operator maintains only stream-level metadata (timestamps, batching state), while the `GpuKernel` implementation owns and manages the actual data buffers. This separation is motivated by several considerations:

- **Layout flexibility**: Different `GpuKernel` implementations may require different data layouts (AoS, SoA, hybrid, or application-specific formats). The implementor is best positioned to choose the optimal representation for their specific computation.
- **Zero-copy accumulation**: By pushing elements directly to the `GpuKernel`'s buffer via `push()`, the design avoids an intermediate copy that would be required if the operator owned the buffer and later transferred it to the `GpuKernel`.
- **Type safety**: The `GpuKernel`'s buffer can use the exact types needed for GPU execution (e.g., separate `Vec<f32>` for each field in SoA format), while the operator only sees the high-level input type.
- **Memory management**: The `GpuKernel` can pre-allocate buffers based on expected batch sizes, reuse allocations across flushes, and implement custom memory strategies.

Figure 3.1 illustrates this division: the operator orchestrates control flow and timing, while the `GpuKernel` encapsulates data representation and device execution.

3.4.2. The GpuKernel Trait Contract

The `GpuKernel` trait defines the interface through which the operator interacts with user-defined GPU logic. Listing 3.4 shows the complete trait definition.

```
pub trait GpuKernel: Clone + Send + 'static {
    type Input: Data + bytemuck::Pod;
    type Output: Data + bytemuck::Pod;

    /// Push a single item to the kernel's internal buffer.
    fn push(&mut self, item: Self::Input);

    /// Return the number of items currently buffered.
    fn buffer_len(&self) -> usize;

    /// Execute the GPU compute kernel and return results.
    /// For pipelined implementations, returns results from the PREVIOUS batch.
    fn flush(&mut self, ctx: &GpuContext) -> Vec<Self::Output>;

    /// Collect any pending results from async execution.
    /// Called at stream termination.
    fn drain(&mut self, ctx: &GpuContext) -> Vec<Self::Output> {
        Vec::new() // Default: no pending results
    }

    /// Optional one-time initialization (e.g., shader compilation).
    fn setup(&mut self, _ctx: &GpuContext) {}

    /// Optional hint for preferred batch size.
    fn preferred_batch_size(&self) -> Option<usize> {
        None
    }
}
```

Listing 3.4: Complete `GpuKernel` trait definition.

Type constraints. The associated types `Input` and `Output` must satisfy two constraints:

- `Data`: Renoir's marker trait (`Clone + Send + 'static`) for stream compatibility.

- `bytemuck::Pod`: "Plain Old Data"—the type must be safely transmutable to and from raw bytes for GPU memory transfers. This requires `#[repr(C)]` layout, no padding, and no pointers or references.

These constraints ensure that data can be safely copied between host and device memory using `bytemuck::cast_slice()` without serialization overhead.

Method contracts.

- `push(item)`: Must be $O(1)$ amortized. The `GpuKernel` should accumulate elements efficiently, typically by appending to pre-allocated vectors.
- `buffer_len()`: Must return the exact count of elements ready for GPU processing. The operator uses this to evaluate flush conditions.
- `flush(ctx)`: Executes the GPU compute kernel on buffered data. For non-pipelined implementations, returns results for the current batch. For pipelined implementations, returns results from the *previous* batch (see Section 3.5). Must clear the internal buffer after execution.
- `drain(ctx)`: Collects any pending results from asynchronous execution. The default implementation returns an empty vector, which is correct for non-pipelined implementations.

3.4.3. Array-of-Structures vs. Structure-of-Arrays

Renoir's dataflow model operates on individual records, which naturally form an Array-of-Structures (AoS) representation. However, GPU compute kernels achieve optimal memory bandwidth with Structure-of-Arrays (SoA) layouts that enable coalesced memory access [22, 26].

Figure 3.2 illustrates this transformation. In AoS layout, fields of different records are interleaved in memory; when threads access the same field across records, memory accesses are strided and bandwidth is wasted. In SoA layout, each field occupies a contiguous array, allowing adjacent threads to access adjacent memory locations.

The performance impact of this transformation is substantial. Consider a GPU compute kernel processing N records, each containing five 32-bit fields. With AoS layout, reading one field per thread results in 20-byte strides (5 fields $\times$ 4 bytes). With a 128-byte cache line, only 6 of the 32 values loaded are useful—an efficiency of approximately 19%. With SoA layout, all 32 values in each cache line are useful, achieving near-100% efficiency [22].

The AoS-to-SoA conversion is implemented within the `GpuKernel`'s `push()` method. Rather than accumulating records in their original form and converting later, the `GpuKernel` immediately decomposes each input and appends its fields to separate vectors:

```
struct SoABuffer {
    stocks: Vec<f32>,    // S_0 for each option
    strikes: Vec<f32>,   // K for each option
    times: Vec<f32>,     // T for each option
    rates: Vec<f32>,     // r for each option
    vols: Vec<f32>,      // sigma for each option
    seeds: Vec<u32>,     // RNG seed for each option
}

impl SoABuffer {
    fn push(&mut self, input: &MonteCarloInput, seed: u32) {
        self.stocks.push(input.stock_price);
        self.strikes.push(input.strike_price);
        self.times.push(input.time_to_expiry);
        self.rates.push(input.risk_free_rate);
        self.vols.push(input.volatility);
        self.seeds.push(seed);
    }
}
```

Listing 3.5: SoA buffer structure for Monte Carlo `GpuKernel`.

This streaming conversion avoids the cost of a separate transformation pass and integrates naturally with the operator's incremental buffering.

3.4.4. Vectorization and Alignment

Modern GPU architectures process data in fixed-width SIMD lanes. NVIDIA GPUs execute instructions on warps of 32 threads, with each thread potentially processing multiple values using vector types [22]. CubeCL exposes this through `Line<T>` types, which represent vectors of 2, 4, 8, or 16 elements processed in a single instruction. Figure 3.4 illustrates the difference between scalar and vectorized execution.

To utilize vectorization effectively, the `GpuKernel` must ensure that buffer sizes are aligned to the vectorization factor. Listing 3.6 shows the padding logic:

```
fn pad_for_vectorization(&mut self, factor: usize) {
    let len = self.stocks.len();
```

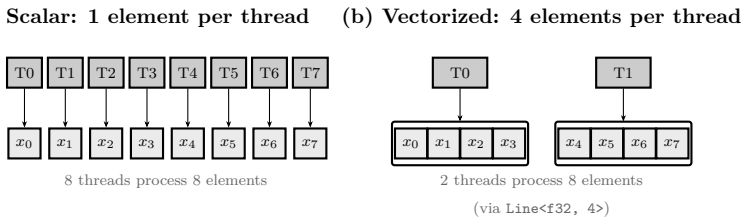

Figure 3.4: Scalar versus vectorized GPU execution. (a) In scalar mode, each thread processes one element. (b) With a vectorization factor of 4, each thread loads and processes a vector of four elements using CubeCL's `Line<T>` type, reducing the number of active threads required and improving memory access efficiency.

```
    let aligned = (len + factor - 1) / factor * factor;

    // Pad with neutral values
    self.stocks.resize(aligned, 1.0);  // Neutral for multiplication
    self.strikes.resize(aligned, 1.0);
    self.times.resize(aligned, 1.0);
    self.rates.resize(aligned, 0.05);  // Typical risk-free rate
    self.vols.resize(aligned, 0.2);    // Typical volatility
}
```

Listing 3.6: Padding buffers for vectorization alignment.

The padding values are chosen to be mathematically neutral or typical for the domain, ensuring that padded elements do not cause numerical issues. The `GpuKernel` tracks the original count (`items_pushed`) to truncate padded results before returning.

3.5. Asynchronous Pipeline and Double-Buffering

GPU execution involves multiple phases—host-to-device data transfer, GPU compute kernel execution, and device-to-host result transfer—each of which can proceed concurrently with the others when properly orchestrated [14, 22]. This section describes the double-buffering technique implemented in the `GpuKernel` trait to exploit this parallelism.

3.5.1. The Pipelining Opportunity

In a naïve synchronous execution model, each batch is processed sequentially: data is uploaded, the GPU compute kernel executes, results are downloaded, and only then does the next batch begin accumulating. This leaves significant hardware idle:

- The CPU waits during GPU execution and data transfers.
- The GPU is idle during host-to-device and device-to-host transfers.
- The PCIe bus is idle during GPU compute kernel computation.

For workloads where transfer and compute times are comparable, this sequential model can waste up to two-thirds of potential throughput. The solution is to overlap these phases across consecutive batches.

3.5.2. Double-Buffering Architecture

Double-buffering (also known as ping-pong buffering) is a classic technique for hiding latency by maintaining two sets of buffers and alternating between them [22]. In the context of `GpuKernel`, this means:

- While batch n is executing on the GPU, batch $n + 1$ is being uploaded.
- While batch $n + 1$ is executing, results from batch n are being downloaded.
- The CPU accumulates batch $n + 2$ during this time.

Figure 3.5 illustrates the timing relationships between batches in both synchronous and pipelined execution models.

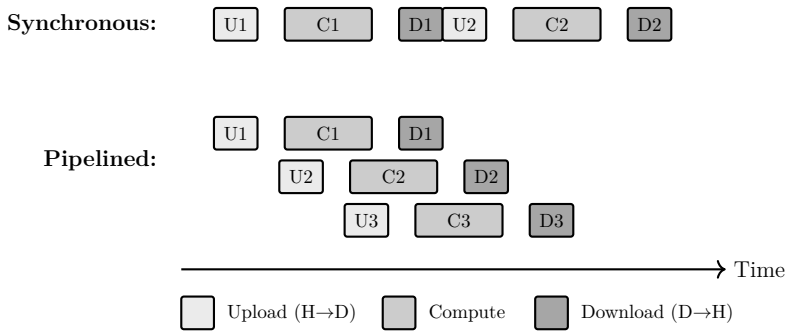

Figure 3.5: Comparison of synchronous and pipelined GPU execution. In synchronous mode, each phase completes before the next begins. In pipelined mode, upload, compute, and download phases overlap across consecutive batches, significantly improving throughput. U = Upload, C = Compute, D = Download. H = Host (CPU memory), D = Device (GPU memory).

3.5.3. Implementation in `flush()`

The double-buffering pattern is implemented within the `GpuKernel`'s `flush()` method. Algorithm 3.2 presents the execution sequence.

Algorithm 3.2 Double-Buffered `flush()` Implementation

Require: Current batch data in SoA buffers, pending handles from previous batch
Ensure: Results from previous batch, new computation launched

Step 1: Upload current batch (asynchronous)
if buffer is not empty **then**
    Pad buffers for vectorization alignment
    Upload SoA buffers to GPU memory → new_handles
    Allocate output buffers on GPU
    Configure GPU compute kernel launch parameters (grid, block dimensions)
end if

Step 2: Read previous results (blocking)
if pending handles exist **then**
    Download results from GPU → previous_results
    {Blocks until GPU done}
    Pair with pending_timestamps
end if

Step 3: Launch new GPU compute kernel (asynchronous)
if new data was uploaded **then**
    Launch GPU compute kernel with new_handles
    Store new_handles as pending for next `flush()`
end if

Clear host-side buffers
return previous_results {Lagged by one batch}

The key insight is the ordering of operations: uploading new data *before* reading previous results allows the upload to overlap with the final stages of the previous GPU compute kernel's execution. By the time the blocking read completes, the new data is already on the GPU and ready for launch.

3.5.4. Lagged Output Model

A consequence of double-buffering is that `flush()` returns results from the *previous* batch, not the current one. This creates a one-batch lag in the output stream:

- First `flush()`: Returns empty vector (no previous batch).
- Subsequent `flush()` calls: Return results from the batch submitted in the previous call.
- Final `drain()`: Returns results from the last submitted batch.

The `MapGpu` operator handles this lag transparently through its `pending_timestamps` mechanism (Section 3.3). Users implementing custom `GpuKernel` implementations must be aware of this model and ensure their `drain()` method correctly collects the final pending results.

3.5.5. When to Use Double-Buffering

Double-buffering is most beneficial when:

- Transfer time and compute time are comparable (neither dominates).
- Batch sizes are large enough to amortize the complexity overhead.
- The application prioritizes throughput over latency.

For compute-bound GPU compute kernels where GPU execution time far exceeds transfer time, the added complexity may not be justified. The `GpuKernel` trait supports both modes: a non-pipelined implementation can simply return current-batch results from `flush()` and leave `drain()` with its default empty implementation.

3.5.6. Memory Management Considerations

Double-buffering requires careful memory management. At any time, the `GpuKernel` holds GPU memory for:

- Input buffers for the batch currently executing.

- Output buffers for the batch currently executing (pending read).
- (Briefly) Input buffers being uploaded for the next batch.

For large batch sizes, this can represent significant GPU memory usage. `GpuKernel` implementations should consider implementing memory limits and adjusting batch sizes accordingly. The `Adaptive` batching strategy (Section 3.6) can help by automatically reducing batch sizes if memory pressure is detected through throughput degradation.

Hardware-dependent realization. The double-buffering mechanism described above is designed primarily for discrete GPU systems where host-to-device transfers over PCIe represent a significant fraction of execution time (typically 10–50 ms for multi-million-element batches at PCIe 4.0 speeds). By overlapping transfer_N with compute_{N-1}, the pipeline can hide this latency entirely when kernel execution time exceeds transfer time.

However, the physical realization of this benefit depends critically on hardware architecture. On systems with unified memory architecture (UMA)—such as the Apple M2 Pro used in Chapter 4—CPU and GPU share the same physical memory pool, and "host-to-device transfers" are replaced by memory-mapped operations with negligible latency. On such platforms, double-buffering provides minimal benefit beyond overlapping kernel launch overhead (∼5–10 μs). The design nonetheless implements full double-buffering to ensure correct behavior and performance portability across both discrete and integrated GPU architectures. Validation of the performance benefit on discrete GPU systems is left to future work (Section 6.4).

3.6. Batching Strategies

Batching is essential for efficient GPU utilization, but the optimal batch size depends on factors including GPU compute kernel complexity, data transfer costs, memory constraints, and latency requirements [14, 22]. The `MapGpu` operator supports multiple batching strategies through the `GpuBatchStrategy` enumeration, allowing users to select or customize the policy best suited to their workload.

3.6.1. Strategy Enumeration and Semantics

The current implementation provides three built-in strategies, each addressing different use cases:

```
pub enum GpuBatchStrategy {
    /// Flush every N items.
```

```
    Fixed(usize),

    /// Flush on timeout OR when buffer reaches max_size.
    Timed { max_size: usize, interval: Duration },

    /// Dynamically adjust batch size based on throughput.
    Adaptive { min_size: usize, max_size: usize },
}

impl GpuBatchStrategy {
    pub fn fixed(size: usize) -> Self {
        assert!(size > 0, "Batch size must be positive");
        GpuBatchStrategy::Fixed(size)
    }

    pub fn timed(max_size: usize, interval: Duration) -> Self {
        assert!(max_size > 0, "Max batch size must be positive");
        GpuBatchStrategy::Timed { max_size, interval }
    }

    pub fn adaptive(min_size: usize, max_size: usize) -> Self {
        assert!(min_size > 0 && max_size >= min_size);
        GpuBatchStrategy::Adaptive { min_size, max_size }
    }
}

impl Default for GpuBatchStrategy {
    fn default() -> Self {
        GpuBatchStrategy::Fixed(10_000_000)
    }
}
```

Listing 3.7: Built-in batching strategies with constructors.

Fixed Strategy

The simplest strategy: accumulate exactly n elements before flushing. This is appropriate for:

- Workloads with predictable, high-volume arrival rates.

- Scenarios where the optimal batch size is known from benchmarking.
- Applications where consistent batch sizes simplify debugging and analysis.

The default value of 10 million elements is chosen based on empirical observations: it provides sufficient work to saturate modern GPUs while remaining within typical memory constraints for 32-bit element types (approximately 40 MB per field in SoA layout).

Timed Strategy

A hybrid strategy that addresses latency concerns:

- Flushes when the buffer reaches `max_size`, achieving throughput similar to `Fixed`.
- Also flushes if `interval` time elapses since the last flush, bounding latency for slow streams.

This strategy is essential for streaming scenarios with variable arrival rates. Consider a real-time analytics application: during peak hours, data arrives quickly and batches fill rapidly. During off-peak hours, arrival rates drop, but users still expect timely results. The `Timed` strategy ensures that elements never wait longer than `interval` for processing.

Listing 3.8 shows the timer implementation:

```
pub struct BatchTimer {
    last_flush: Instant,
    interval: Duration,
}

impl BatchTimer {
    pub fn new(interval: Duration) -> Self {
        Self { last_flush: Instant::now(), interval }
    }

    pub fn should_flush(&self) -> bool {
        self.last_flush.elapsed() >= self.interval
    }

    pub fn record_flush(&mut self) {
        self.last_flush = Instant::now();
    }
}
```

Listing 3.8: `BatchTimer` for timed flush decisions.

Adaptive Strategy

A throughput-optimizing strategy that dynamically adjusts the target batch size based on observed GPU performance. Algorithm 3.3 describes the adaptation logic.

Algorithm 3.3 Adaptive Batch Size Adjustment

Require: min_size, max_size, current throughput measurement
Ensure: Updated target batch size

```
1:
2: throughput ← items_processed / elapsed_time
3: if previous_throughput exists then
4:     if throughput > 1.05× previous_throughput and current_size < max_size then
5:         current_size ← min(current_size ×2, max_size)
6:         {Larger batches improving throughput}
7:     else if throughput < 0.8× previous_throughput and current_size > min_size
       then
8:         current_size ← max(current_size / 2, min_size)
9:         {Throughput degraded; try smaller batches}
10:    end if
11: end if
12: previous_throughput ← throughput
```

The adaptive strategy is particularly useful when:

- The optimal batch size is unknown or varies with workload characteristics.
- The application runs on different GPU hardware without retuning.
- Memory pressure from other applications affects available GPU resources.

The 5% improvement threshold prevents oscillation due to measurement noise, while the 20% degradation threshold triggers rapid reduction when batch sizes exceed optimal levels (e.g., causing memory thrashing).

3.6.2. Strategy Selection Guidelines

Table 3.3 provides guidance for selecting an appropriate batching strategy based on application requirements.

Table 3.3: Batching strategy selection based on application requirements.

Requirement	Characteristics	Strategy
Maximum throughput	High-volume, predictable arrivals; latency not critical	`Fixed` or `Adaptive`
Bounded latency	Variable arrival rates; responsiveness required	`Timed`
Unknown hardware	Deployment across diverse GPU configurations	`Adaptive`
Debugging/testing	Need reproducible batch boundaries	`Fixed`
Memory-constrained	Limited GPU memory; large element sizes	`Fixed` (small) or `Adaptive`

3.6.3. Implementing Custom Strategies

While the built-in strategies cover common use cases, specialized applications may require custom batching logic. The current design exposes the `should_flush()` decision point within the operator. To implement a custom strategy, developers can:

1. Extend the `GpuBatchStrategy` enum with a new variant.
2. Add corresponding state structures (similar to `BatchTimer` and `AdaptiveSizer`).
3. Modify the `should_flush()` and `flush_to_gpu()` methods to handle the new variant.

Listing 3.9 illustrates a hypothetical memory-aware strategy:

```
pub enum GpuBatchStrategy {
    // ... existing variants ...

    /// Flush when estimated GPU memory usage approaches threshold.
    MemoryAware {
        max_memory_bytes: usize,
        element_size: usize,
    },
}

// In should_flush():
GpuBatchStrategy::MemoryAware { max_memory_bytes, element_size } => {
    let estimated_memory = self.kernel.buffer_len() * element_size * 2;
    // Factor of 2 accounts for input + output buffers
```

```
    estimated_memory >= max_memory_bytes * 90 / 100  // 90% threshold
}
```

Listing 3.9: Example custom strategy: memory-aware batching.

Other potential custom strategies include:

- **Demand-driven**: Flush when downstream operators signal back-pressure.
- **Content-sensitive**: Flush when a delimiter or end-of-window marker is observed.
- **Priority-based**: Flush high-priority items immediately while batching low-priority items.
- **Deadline-aware**: Flush elements approaching their processing deadline.

The modular separation between batching logic and execution flow ensures that such extensions can be added without invasive changes to the operator's stream handling or timestamp management.

3.6.4. Interaction with Kernel Hints

The `GpuKernel` trait includes an optional `preferred_batch_size()` method that allows implementations to suggest an optimal batch size. When present, this hint can inform the batching strategy:

```
let strategy = match kernel.preferred_batch_size() {
    Some(preferred) => GpuBatchStrategy::Fixed(preferred),
    None => GpuBatchStrategy::default(),
};
```

Listing 3.10: Using `GpuKernel` hints in strategy initialization.

`GpuKernel` authors should provide this hint when they have domain knowledge about optimal batch sizes—for example, based on GPU memory constraints, warp utilization requirements, or kernel-specific algorithmic considerations.

4 | Evaluation

This chapter evaluates the GPU integration approach presented in Chapter 3 through a series of experimental benchmarks. The evaluation is structured to progressively characterize the conditions under which GPU acceleration is beneficial within a dataflow framework. Section 4.1 describes the experimental environment. Section 4.2 evaluates the batching strategies introduced in the design. Sections 4.3 through 4.5 present three benchmark case studies of increasing computational intensity: a reduction operator, Black–Scholes option pricing, and Monte Carlo simulation. Finally, Section 4.6 provides a cross-benchmark analysis and discusses the implications for GPU-accelerated dataflow processing.

4.1. Experimental Setup

4.1.1. Hardware Platform

All experiments were conducted on a system equipped with the Apple M2 Pro system-on-chip [2]. Table 4.1 summarizes the hardware configuration.

Table 4.1: Hardware configuration of the evaluation platform.

Component	**Specification**
CPU	Apple M2 Pro (8 performance + 4 efficiency cores, 12 total)
GPU	Apple M2 Pro integrated GPU (10 cores)
Memory	32 GB unified LPDDR5 (shared between CPU and GPU)
Memory bandwidth	200 GB/s (shared)
Peak GPU compute	∼6.8 TFLOPS (single-precision)
Operating system	macOS (Darwin, aarch64)

Computational throughput in this chapter is measured in *FLOPS* (floating-point operations per second). One GFLOPS equals 10^9 FLOP/s and one TFLOPS equals 10^{12} FLOP/s. For example, the M2 Pro GPU's peak of 6.8 TFLOPS means it can theoretically perform 6.8×10^{12} single-precision floating-point operations per second.

4.1.2. Unified Memory Architecture

A distinguishing characteristic of the Apple Silicon platform is its *unified memory architecture* (UMA), in which CPU and GPU share the same physical memory pool. Unlike systems with discrete GPUs, where data must be explicitly transferred between host RAM and device VRAM over a PCIe bus, the M2 Pro allows both the CPU and GPU to access the same memory addresses without an explicit copy across a bus.

This architecture has two important implications for the evaluation. First, it eliminates the PCIe transfer bottleneck that often dominates GPU execution time on discrete systems. In conventional discrete GPU setups, data must traverse a PCIe 4.0 or PCIe 5.0 link (typically 16–64 GB/s), which can represent a significant fraction of the total execution time for memory-bound workloads [14]. On the M2 Pro, host-to-device transfers are replaced by memory-mapped access at up to 200 GB/s of shared bandwidth. Second, however, unified memory introduces *contention*: CPU and GPU compete for the same memory bandwidth, which can reduce effective throughput when both are active simultaneously.

As a consequence, the results reported in this chapter represent a *favorable scenario* for GPU integration with respect to data transfer overhead. On systems with discrete GPUs, the crossover point at which GPU acceleration becomes beneficial would shift further toward compute-bound workloads, as explicit data transfers over PCIe would add additional overhead.

4.1.3. Absence of NVIDIA/CUDA Hardware

It is important to note that no NVIDIA discrete GPU was available for this evaluation. All experiments use the Apple M2 Pro integrated GPU accessed through the WGPU backend, which targets the Metal graphics API [3] on macOS. This practical constraint has several implications.

First, the CUDA backend of CubeCL [27], which supports double-precision floating point (`f64`) and benefits from dedicated high-bandwidth VRAM, could not be tested. All GPU computations in this evaluation use single-precision (`f32`), as the WGPU/WebGPU specification [30] does not mandate `f64` support. For the financial benchmarks evaluated here, single precision is acceptable and commonly used [24], but this limitation may affect numerical accuracy in other application domains.

Second, a discrete NVIDIA GPU with dedicated VRAM—for example, an RTX 4090 with 24 GB of GDDR6X memory at approximately 1 TB/s of bandwidth—would exhibit

qualitatively different performance characteristics. Such a system would offer higher raw kernel throughput and dedicated memory bandwidth, but would incur additional PCIe data transfer overhead that is absent on the unified memory platform used here.

Despite this limitation, the evaluation remains valid for its primary purpose: assessing the *feasibility and overhead characteristics* of GPU integration within a dataflow framework. The architectural challenges that this work addresses—batching, data layout conversion, asynchronous execution, and result synchronization—are independent of the specific GPU vendor or backend. Extending the evaluation to NVIDIA hardware with the CUDA backend is a natural direction for future work and is discussed further in Chapter 6.

4.1.4. Software Stack

Table 4.2 summarizes the software environment. All components are pinned to specific versions to ensure reproducibility.

Table 4.2: Software stack and versions used for all experiments.

Component	Version / Configuration
Operating system	macOS (Darwin, aarch64)
Rust toolchain	`rustc` stable, compiled with `--release` profile (LTO enabled)
Renoir [10]	v0.6.0, extended with `map_gpu` and `reduce_gpu` operators (this work)
CubeCL [27]	v0.8.1 (with `wgpu` and `reduce` features)
WGPU [12]	As bundled with CubeCL 0.8.1, targeting Apple Metal via WebGPU API
GPU backend	WGPU (`gpu-wgpu` Cargo feature flag); CUDA backend not tested
Floating-point precision	`f32` for GPU, `f64` for CPU (cross-validated)
Benchmark harness	Custom Rust harness with JSON result serialization

4.1.5. Evaluation Methodology

Each benchmark compares three execution strategies:

1. **CPU Sequential**: A single Renoir worker processes all elements sequentially. This establishes the baseline performance without parallelism.
2. **CPU Parallel**: Renoir distributes the workload across all 12 available CPU cores using its standard parallel execution model.
3. **GPU**: The `map_gpu` or `reduce_gpu` operator processes elements using the integrated

GPU, with batching and asynchronous pipelining as described in Chapter 3.

All measurements report *end-to-end wall-clock time* through the full Renoir pipeline, including stream construction, operator execution, data collection, and any overhead introduced by batching, data layout conversion, and GPU synchronization. This reflects the real-world performance a user would observe and avoids the misleading practice of reporting only kernel execution time in isolation.

CPU baseline characterization. An important methodological clarification: both the CPU Sequential and CPU Parallel baselines use Renoir's standard `map` operator with unmodified Rust code. No manual SIMD vectorization (e.g., explicit ARM Neon or x86 AVX intrinsics), hand-tuned loop unrolling, or platform-specific optimizations were applied to the CPU code paths. The Rust compiler (`rustc`) may apply automatic vectorization during compilation with the `--release` profile, but this was not verified or enforced. In particular, the Monte Carlo kernel's inner loop—which includes xorshift32 random number generation, Box–Muller transform, and conditional payoff computation—involves data-dependent branching that limits auto-vectorization effectiveness. Consequently, the CPU baselines represent *framework-level* performance: the same algorithm running through the same Renoir pipeline, with the only difference being the execution backend (CPU `map` vs. GPU `map_gpu`). This is a deliberate design choice: the evaluation compares what a Renoir user would experience when switching from `map` to `map_gpu`, rather than comparing an optimized GPU implementation against a separately hand-tuned CPU implementation.

To ensure correctness, each benchmark includes a validation step that compares GPU results against CPU results. Validation uses a configurable error tolerance to account for differences between `f32` (GPU) and `f64` (CPU) arithmetic. All benchmarks save results incrementally to JSON files, enabling crash-resilient data collection for long-running experiments.

Precision asymmetry. An important caveat applies to all GPU-vs-CPU speedup figures reported in this chapter: the GPU kernels operate in single precision (`f32`) while both CPU baselines use double precision (`f64`). On ARM cores such as the Apple M2 Pro, `f64` arithmetic can be 1.5–2× slower than `f32` for compute-heavy workloads, because the NEON SIMD units process half as many `f64` elements per instruction. This means the CPU baselines perform *more expensive* arithmetic per element than the GPU, which may inflate the reported GPU speedups. The asymmetry is inherent to the current WGPU/WebGPU backend, which does not mandate `f64` support. For the financial workloads evaluated here, `f32` precision is standard practice [24], but readers should bear this caveat

in mind when interpreting the absolute speedup values.

Statistical methodology. Each benchmark configuration was executed with the following protocol:

- **Warmup phase:** 1 warmup run per configuration, excluded from measurements. This ensures GPU shader compilation, pipeline state creation, and memory allocator initialization are complete before timing begins.
- **Measurement phase:** 5 timed runs per configuration, with comprehensive statistics computed.
- **Reported metrics:** Mean, standard deviation, minimum, maximum, and median across the 5 measurement runs.
- **Data preservation:** Complete per-run statistics are persisted to timestamped JSON files. Each test result includes the full `RunStats` structure containing mean, stddev, min, max, median, and individual run durations for all three execution strategies (CPU Sequential, CPU Parallel, GPU). This enables post-hoc statistical analysis beyond what is presented in the tables.

The reported speedup values are computed from mean execution times. Standard deviations are reported alongside means in the result tables using mean $\pm$ stddev notation; typical coefficient of variation (CV = stddev/mean) is below 2% for large inputs and below 10% for small inputs where timing granularity dominates. System conditions during benchmarking: background CPU utilization was approximately 10% (minimal applications running), GPU was idle prior to benchmark execution, and both CPU and GPU reached 100% utilization during peak workload phases. No CPU frequency governor pinning or thread affinity settings were applied; the system operated under normal macOS scheduling.

Repeatability protocol. To enable reproducibility, the following experimental conditions are documented:

- **Worker configuration:** In Renoir's execution model, a *worker* is an independent execution unit that processes a replica of the dataflow graph [10]. Each worker runs in its own OS thread, maintains its own operator state, and processes a partition of the input stream. Workers are *not* equivalent to CPU cores: a worker is a logical execution unit, while a core is a physical hardware resource. Renoir's `RuntimeConfig::local(n)` API specifies the number of workers; the OS scheduler maps worker threads to available CPU cores.

The three execution strategies use different worker counts:

- **CPU Sequential**: `local(1)` — a single worker processes all elements sequentially on one thread.
- **CPU Parallel**: `local(12)` — twelve workers partition the input and process concurrently, matching the total CPU core count (8 performance + 4 efficiency cores).
- **GPU**: `local(1)` — a single worker manages the GPU pipeline.

The GPU configuration uses a single worker because the evaluation platform has only one GPU device. Running multiple workers with GPU operators would cause device contention, duplicated device buffer allocations, and WGPU command queue serialization overhead, negating any parallelism benefit. The single-worker GPU configuration reflects the intended usage pattern: one worker owns the GPU context and processes batches sequentially through the asynchronous pipeline described in Chapter 3.

- **OS noise control:** No special isolation measures (CPU shielding, IRQ affinity, `nice` priority) were applied. Benchmarks were run on a lightly loaded system with minimal background applications. Results therefore reflect realistic deployment conditions rather than idealized noise-free execution.
- **Benchmark infrastructure:** The benchmark harness automatically detects system configuration (CPU cores, RAM, GPU device), generates test sizes, executes warmup and measurement runs, computes statistics, and persists results to timestamped JSON files. Environment variables (`BENCH_RUNS`, `MAX_OPTIONS`) allow customization of run count and maximum input size.

These choices prioritize reproducibility over maximum reported performance. Researchers replicating this evaluation should expect similar results on comparable hardware without requiring specialized system configuration.

4.2. Batching Strategy Evaluation

Before evaluating specific workloads, this section examines how different batching strategies—as introduced in Section 3.6—affect GPU throughput. Since the choice of batching strategy determines when accumulated stream elements are flushed to the GPU for processing, it directly impacts both throughput and latency.

4.2.1. Strategies Under Evaluation

The `map_gpu` operator supports three batching strategies:

- **Fixed(n)**: Flush exactly after every n elements. This provides predictable batch sizes but is insensitive to workload dynamics. Evaluated with $n \in \{10K, 100K, 1M, 10M\}$.
- **Timed{interval, min_size}**: Flush after a time interval has elapsed, provided at least `min_size` elements have accumulated. This strategy is suitable for latency-sensitive applications where waiting for a full batch would introduce excessive delay. Evaluated with intervals of 1K/100ms.
- **Adaptive{min, max}**: Dynamically adjusts batch size between `min` and `max` based on observed stream arrival rate. The batch size grows when elements arrive quickly and shrinks during slow periods. Evaluated with a range of 100K–10M.

4.2.2. Results

Figures 4.1 and 4.2 present the results of the batching strategy comparison, measured using the Black–Scholes kernel across input sizes ranging from 100K to 500M elements.

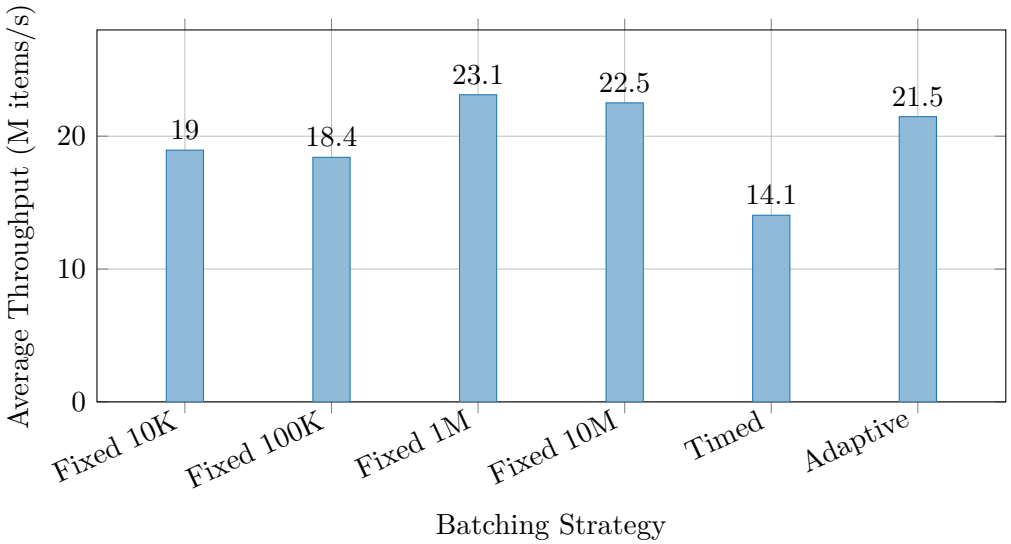

Figure 4.1: Average throughput across all input sizes for each batching strategy. Fixed(1M) achieves the highest average throughput at 23.1 M items/s, followed closely by Fixed(10M) at 22.5 M items/s and the Adaptive strategy at 21.5 M items/s. The two smaller fixed batch sizes, Fixed(10K) and Fixed(100K), average approximately 19 M items/s—lower because their per-batch overhead is amortized over fewer elements. The Timed strategy is the weakest at 14.1 M items/s, as its time-based trigger often flushes before accumulating a sufficiently large batch.

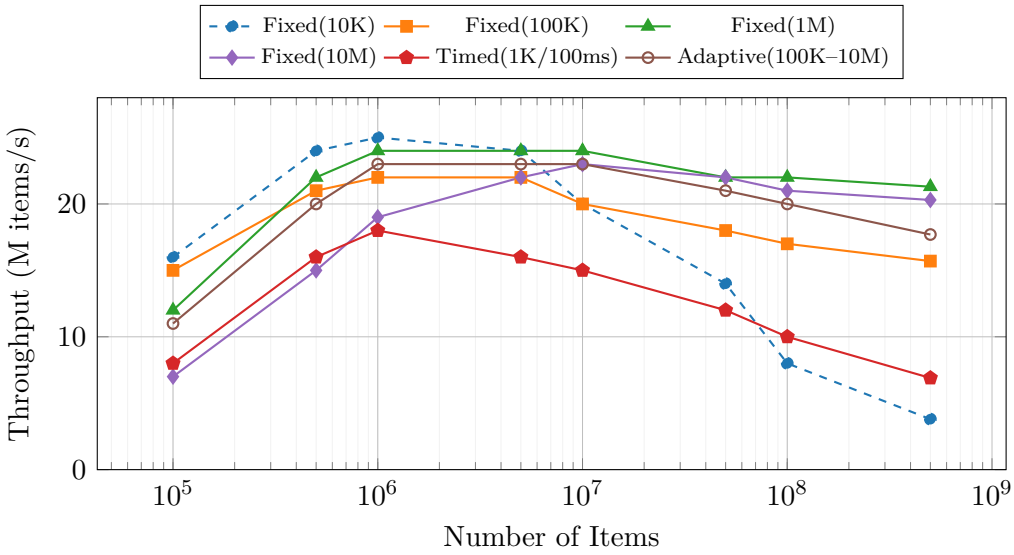

Figure 4.2: Throughput as a function of input size (x-axis logarithmic) for each batching strategy. At small input sizes ($\leq$1 M), all strategies converge, as few batches are needed regardless of the strategy. As input grows, the strategies diverge: Fixed(1M) and Fixed(10M) maintain high throughput because their batch sizes are large enough to amortize overhead yet small enough to avoid excessive memory pressure. The Adaptive strategy closely tracks Fixed(1M). Conversely, Fixed(10K) degrades sharply beyond 10 M elements because 10 K-element batches produce thousands of kernel launches, accumulating per-batch overhead. The Timed strategy also declines, as its time-triggered flushes produce irregular, often suboptimal batch sizes. The relative ordering of strategies remains largely stable for inputs above 5 M elements, confirming that the strategy choice is not overly sensitive to problem size in the high-throughput regime.

Several observations emerge from this comparison:

1. **Fixed batch sizes near 1M elements yield the highest throughput.** The Fixed(1M) strategy achieves an average throughput of approximately 23 million items per second, outperforming all other configurations. This suggests that at this batch size, kernel launch overhead is sufficiently amortized while avoiding excessive memory pressure.

2. **The Adaptive strategy performs comparably to Fixed(1M).** With an average throughput of approximately 21.5 million items per second, the adaptive approach closely tracks the best fixed configuration. This is expected, as the adaptive algorithm converges toward similar batch sizes under sustained high-throughput streams.

3. **Excessively large fixed batches degrade performance.** Fixed(10M) shows lower throughput than Fixed(1M). At 10M elements of `f32` with five SoA fields, the input buffers alone consume $10^7 \times 5 \times 4 = 200$ MB, which far exceeds the GPU's last-level cache and forces repeated DRAM accesses. Combined with longer collection times that delay pipeline throughput, this results in measurably worse performance than the 1M configuration.

4. **Time-based batching is least efficient.** The Timed strategy achieves the lowest throughput ($\sim$14 million items per second), as the time-based trigger often fires before accumulating enough elements for efficient GPU utilization. However, this strategy may be appropriate for latency-sensitive applications where timely processing is more important than maximum throughput.

5. **Strategy ranking is consistent across input sizes.** Figure 4.2 shows that the relative ordering of strategies remains stable for inputs above approximately 5 million elements, confirming that the choice is not overly sensitive to problem size in the high-throughput regime.

Based on these results, all subsequent benchmarks in this chapter use the **Fixed** batching strategy with workload-specific batch sizes chosen to maximize throughput for each benchmark scenario. Specifically: 4,000,000 elements for the Reduce benchmark (Section 4.3), 5,000,000 options for Black–Scholes (Section 4.4), and 100,000 options for Monte Carlo (Section 4.5). The Monte Carlo batch size is smaller because each element requires $\sim$1,000,000 FLOPs of computation, making smaller batches sufficient to saturate the GPU.

4.3. Preliminary Evaluation: The Reduce Operator

4.3.1. Context and Motivation

The evaluation began with the `reduce_gpu` operator, which was not part of the original thesis plan but was implemented as a natural starting point for GPU integration testing. The CubeCL framework provides built-in reduction kernels for common operations (sum, product, minimum, maximum) [27], eliminating the need to write custom GPU kernels. This made the reduce operator an ideal first test of the GPU integration pipeline: it exercised the full data path (stream collection, batching, GPU buffer allocation, kernel dispatch, and result retrieval) with minimal implementation effort.

The reduce operator was implemented using CubeCL's built-in `reduce` module, configured with a tile size of 262,144 elements and a batch size of 4,000,000 elements. Four reduction kernels were evaluated: Sum, Product, Min, and Max.

Listing 4.1 shows the GPU reduce pipeline as used in the benchmark. The entire reduction—from stream construction to GPU execution—is expressed in five lines of Renoir pipeline code. The developer selects a reduction kernel (e.g., `ReduceKernel::Sum`) and provides a configuration; the framework handles batching, GPU buffer management, multi-pass reduction, and result collection.

```
let env = StreamContext::new_local();
let config = ReduceGpuConfig::default()
    .with_batch_size(4_000_000)
    .with_tile_size(262_144)
    .with_backend(ReduceGpuBackend::Auto);

let output = env
    .stream_iter(data.into_iter())
    .reduce_gpu_with(ReduceKernel::Sum, config)
    .collect_vec();
env.execute_blocking();
```

Listing 4.1: Renoir pipeline for GPU-accelerated reduction. The `reduce_gpu_with` operator accepts a built-in reduction kernel and a configuration specifying batch and tile sizes.

4.3.2. Profiling Analysis

To understand the execution time breakdown, a flame chart was generated for the GPU reduce operator pipeline using Rust's `tracing` instrumentation. Because the primary evaluation platform (Apple M2 Pro) uses a unified memory architecture where host-to-device transfer costs are partially masked, the profiling trace was captured on an auxiliary system equipped with a discrete NVIDIA GPU connected via PCI Express, where explicit data transfers between host and device memory are required. This auxiliary system—an ASUS laptop with an NVIDIA GPU and the CUDA backend—was available only for limited profiling sessions and could not be used to replicate the full benchmark suite. Nevertheless, profiling on discrete hardware provides a clearer view of the data transfer overhead, which is the primary cost component we wish to isolate. Figure 4.3 shows the profiling results for a complete `reduce_gpu` invocation on this system.

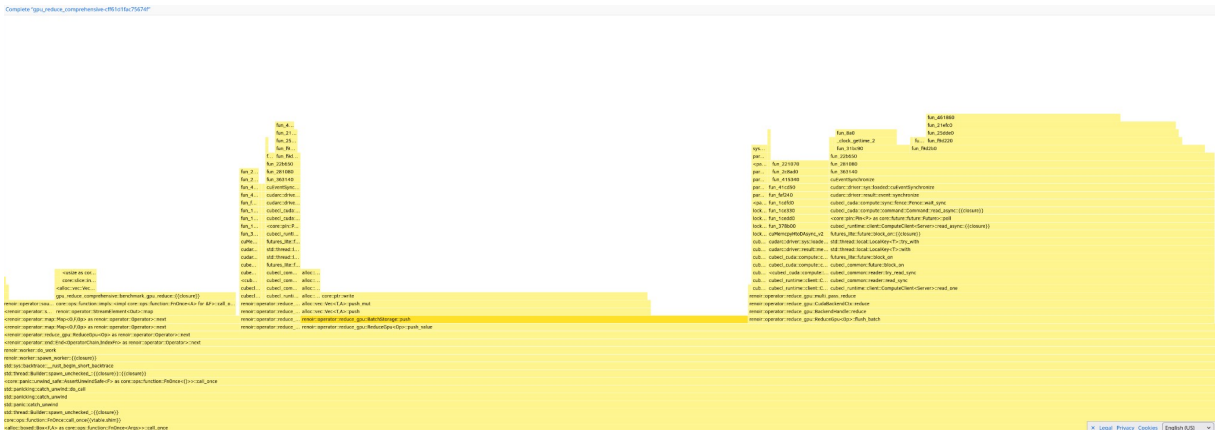

Figure 4.3: Flame chart of the `reduce_gpu` operator execution pipeline, captured on an auxiliary system with a discrete NVIDIA GPU (CUDA backend) via the `tracing` crate and rendered with Chromium's trace viewer. This system differs from the primary evaluation platform (Apple M2 Pro); it was used specifically to expose host–device data transfer overhead that would be partially hidden under unified memory. The horizontal axis represents wall-clock time; each horizontal bar represents a function span, and nested bars indicate caller–callee relationships. Key phases visible from left to right include: (1) `collect_batch`; (2) `cubecl_runtime` buffer allocation and `AoS→SoA` layout conversion; (3) GPU kernel dispatch (`cubecl_cuda` compute pass), which occupies only a narrow sliver; and (4) synchronization and result readback.

The flame chart reveals several important findings about the GPU pipeline overhead. First, the actual GPU reduction kernel—the `cubecl_wgpu` compute pass visible as a narrow band in the centre of the trace—constitutes only a small fraction of the total execution time. The dominant costs are:

- **Batch collection.** Incoming stream elements are accumulated into a contiguous buffer (`collect_batch`), requiring memory allocation and copying proportional to the batch size.
- **AoS-to-SoA conversion.** The data layout transformation from Rust's Array-of-Structures representation to the Structure-of-Arrays format required by the GPU kernel adds a CPU-side pass over the entire batch.
- **Buffer allocation and transfer.** The CubeCL runtime allocates GPU-visible buffers and copies data from host to device memory. On this discrete NVIDIA system, this involves explicit PCI Express data transfers, which are clearly visible as a dominant cost in the trace. On the primary evaluation platform (Apple M2 Pro, unified memory), these transfers are cheaper since CPU and GPU share the same physical memory, but the runtime must still map pages into the GPU's address space and issue memory barriers.
- **Synchronization and readback.** After kernel dispatch, the pipeline must wait for the GPU to signal completion and then read back the reduced result, incurring synchronization latency.

Although this flame chart was captured on an auxiliary discrete-GPU system rather than the primary evaluation platform, the qualitative breakdown generalises: batch collection, data layout conversion, buffer management, and synchronization constitute the fixed overhead of any GPU pipeline invocation, regardless of the memory architecture. On the M2 Pro, the transfer component is reduced by unified memory, but the remaining costs persist. This profiling result provides direct evidence that, for extremely simple per-element computations such as a single addition, the overhead of the GPU pipeline dwarfs the computational benefit. The flame chart motivates the subsequent benchmarks: by increasing the arithmetic intensity of the kernel, the goal is to make the kernel execution time dominate the fixed overhead, thereby shifting the performance balance in favor of GPU execution.

4.3.3. Benchmark Results

Table 4.3 summarizes the reduction benchmark results across all four operators, and Table 4.4 shows detailed timing statistics for the Sum operator.

Figures 4.4–4.7 present the benchmark results for the Sum reduction operator, which is representative of all four operators tested.

Table 4.3: GPU reduction benchmark summary (5 runs, 1 warmup, 12 workers). Speedup values below 1.0 indicate that the CPU is faster than the GPU. 22 test sizes per operator, 88 total tests.

Metric	Sum	Product	Min	Max
Peak speedup vs. Sequential	0.48×	0.64×	0.07×	0.07×
Peak speedup vs. Parallel	0.99×	0.70×	0.65×	0.56×
GPU wins vs. Sequential	0 / 22	0 / 22	0 / 22	0 / 22
GPU wins vs. Parallel	0 / 22	0 / 22	0 / 22	0 / 22
Validation passed	22 / 22	22 / 22	22 / 22	22 / 22

Table 4.4: Reduce (Sum) detailed timing for selected sizes. Times shown as mean ± standard deviation. CV is coefficient of variation for GPU execution.

Size	CPU Seq	CPU Par	GPU	vs.Seq	vs.Par	CV
1M	0.88±0.01 ms	0.60±0.02 ms	2.48±0.05 ms	0.36×	0.24×	1.9%
10M	8.68±0.09 ms	4.11±0.06 ms	22.17±0.14 ms	0.39×	0.19×	0.6%
100M	86.3±0.5 ms	37.9±1.1 ms	184.6±0.6 ms	0.47×	0.21×	0.3%
500M	431±1 ms	186±2 ms	902±4 ms	0.48×	0.21×	0.4%
750M	647±2 ms	278±3 ms	1356±9 ms	0.48×	0.21×	0.6%

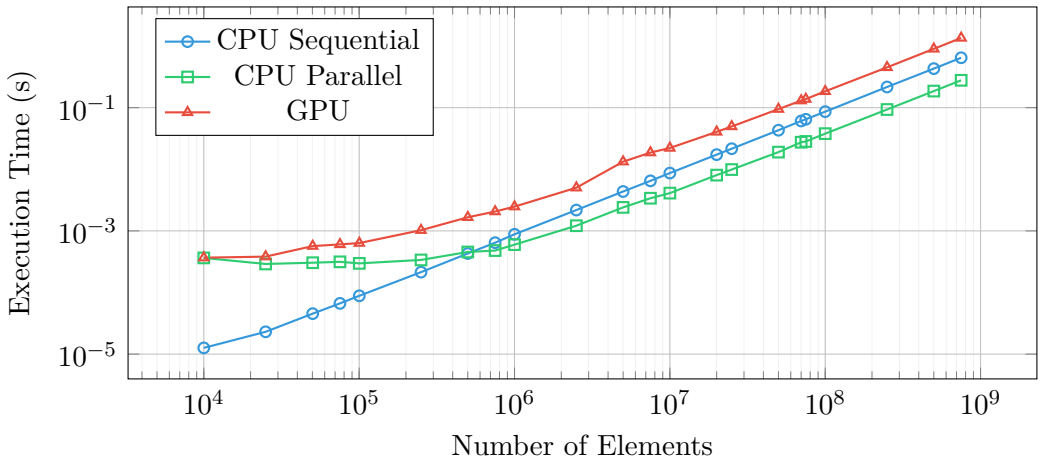

Figure 4.4: Execution time for the Sum reduction operator as a function of input size (both axes logarithmic). The x-axis represents the number of elements reduced, and the y-axis shows wall-clock time in seconds. All three strategies scale linearly with input size on this log-log plot. The GPU (red triangles) is consistently the slowest strategy, taking approximately twice as long as the sequential CPU (blue circles) and roughly four times longer than the parallel CPU (green squares). This ordering holds across all tested sizes, from 10 K to 750 M elements, confirming that the GPU pipeline overhead dominates the trivial per-element computation of a single addition.

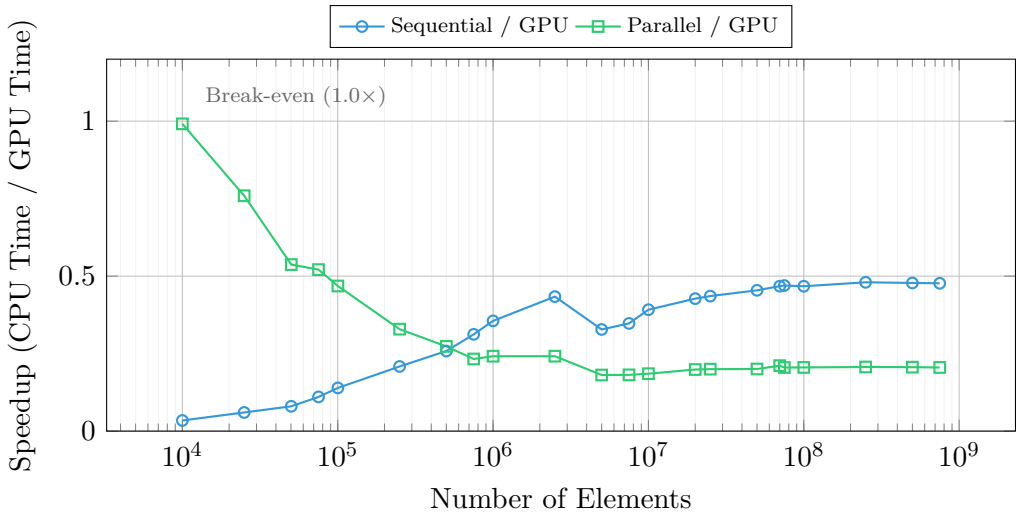

Figure 4.5: GPU speedup for the Sum reduction operator. The x-axis shows the number of elements on a logarithmic scale; the y-axis shows the speedup ratio (CPU time divided by GPU time). The dashed horizontal line at 1.0× marks the break-even point. Both curves remain *entirely below* the break-even line: the sequential-to-GPU speedup (blue circles) plateaus near 0.48×, meaning the GPU takes approximately twice as long as a single CPU core. The parallel-to-GPU speedup (green squares) is even lower at approximately 0.21×. These results demonstrate that for workloads with an arithmetic intensity of only 0.25 FLOP/byte, the GPU pipeline overhead and memory-bound nature of the computation make GPU acceleration counterproductive.

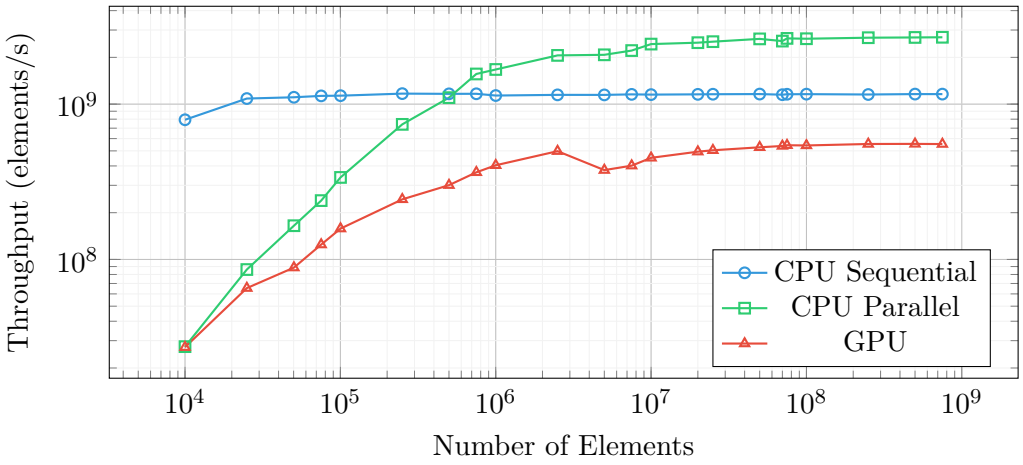

Figure 4.6: Processing throughput for the Sum reduction operator, measured in elements per second (both axes logarithmic). The parallel CPU (green squares) achieves the highest throughput, processing approximately 2.7 billion elements per second for large inputs, followed by the sequential CPU (blue circles) at approximately 1.2 billion elements per second. The GPU (red triangles) sustains roughly 550 million elements per second for large inputs. The throughput ordering is consistent across all tested sizes and confirms that for this memory-bound operation the CPU's cache hierarchy and branch-free execution model provide a substantial advantage over the GPU pipeline.

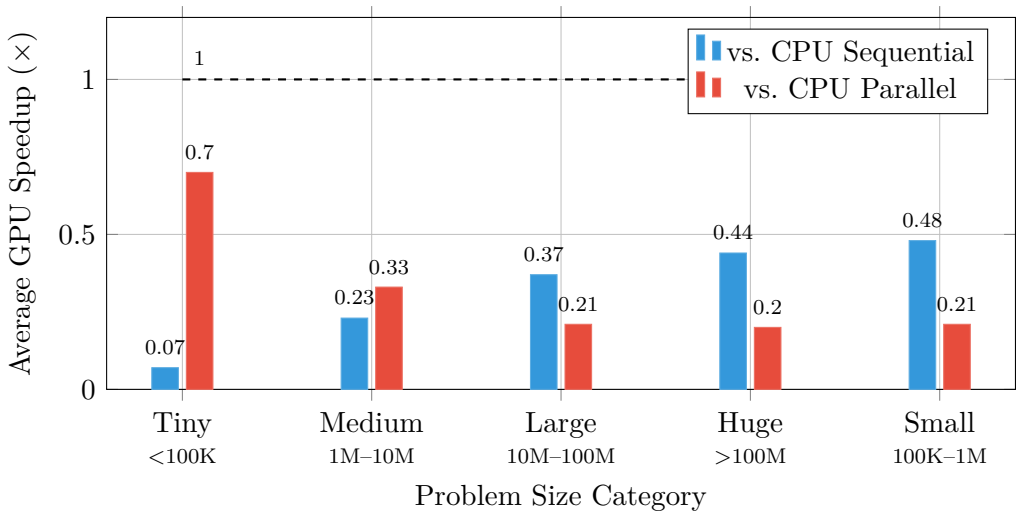

Figure 4.7: Average GPU speedup for the Sum reduction operator grouped by problem size category. Both bars remain below the break-even line for all categories. Against the sequential CPU (blue), the GPU speedup increases with problem size (from 0.07× for tiny inputs to 0.48× for huge inputs) as the constant pipeline overhead is amortized over more elements. Against the parallel CPU (red), the speedup is highest for tiny inputs (0.70×, since parallel overhead is relatively large) and decreases to approximately 0.21× for medium and larger inputs where 12-core parallelism is fully utilized. The GPU never reaches break-even in any category, confirming that the reduction's arithmetic intensity of 0.25 FLOP/byte is fundamentally too low for GPU acceleration.

The results are clear: the GPU *never* outperforms the CPU for any reduction operator at any input size. As Figures 4.5 and 4.7 show, the best GPU performance relative to sequential CPU is 0.64× (Product kernel), meaning even a single CPU core processes reductions faster than the GPU pipeline. The Sum operator comes closest to parity with parallel CPU execution at 0.99×, but still fails to exceed break-even. Figure 4.6 further illustrates this gap: the parallel CPU achieves nearly five times the throughput of the GPU for large inputs.

4.3.4. Analysis

The poor GPU performance for reductions is explained by the extremely low *arithmetic intensity* of the workload. A reduction performs exactly one floating-point operation per element (e.g., one addition for sum), while each element requires 4 bytes of memory access (for `f32`). This yields an arithmetic intensity of:

$$\text{Arithmetic intensity} = \frac{1 \text{ FLOP}}{4 \text{ bytes}} = 0.25 \text{ FLOP/byte} \tag{4.1}$$

At this ratio, the workload is *extremely memory-bound*: the time to read data from memory far exceeds the time to perform the computation. The GPU's massive parallel compute capacity is entirely wasted, as performance is limited by memory bandwidth rather than compute throughput. In addition, the GPU pipeline overhead (batching, buffer allocation, kernel dispatch, result readback) adds a constant cost per batch that cannot be amortized by the trivial per-element computation.

This result establishes an important lower bound: GPU acceleration within a dataflow framework is *not* universally beneficial. For operations with very low arithmetic intensity, the CPU—particularly in a multi-core parallel configuration—is the superior execution strategy. This finding motivated the search for workloads with higher computational intensity, leading to the Black–Scholes and Monte Carlo benchmarks presented in the following sections.

4.4. Black–Scholes Option Pricing

4.4.1. Algorithm Description

The Black–Scholes model [4] provides a closed-form solution for pricing European call options. Given the initial stock price S_0, strike price K, risk-free interest rate r, volatility

σ, and time to maturity T, the fair price C of a European call option is:

$$C = S_0\,\Phi(d_1) - K\,e^{-rT}\,\Phi(d_2) \tag{4.2}$$

where $\Phi(\cdot)$ denotes the cumulative standard normal distribution function, and

$$d_1 = \frac{\ln(S_0/K) + (r + \sigma^2/2)\,T}{\sigma\sqrt{T}}, \qquad d_2 = \frac{\ln(S_0/K) + (r - \sigma^2/2)\,T}{\sigma\sqrt{T}}. \tag{4.3}$$

Each option price evaluation requires approximately 40 floating-point operations, including two evaluations of the error function (used to compute Φ), two logarithms, one exponential, and one square root. While the formula appears simple, it constitutes a well-established computational finance benchmark used extensively in GPU performance evaluation [24].

4.4.2. Rationale for Selection

Black–Scholes was selected as the first `map_gpu` benchmark for several reasons:

1. **Higher computational intensity than reductions.** At approximately 40 FLOPs per option versus 1 FLOP per element for reductions, Black–Scholes offers a 40× increase in per-element computation.

2. **Embarrassingly parallel structure.** Each option is priced independently, making the workload naturally suited to GPU execution.

3. **Established benchmark in the literature.** Black–Scholes is widely used for evaluating GPU performance in financial computing, enabling comparison with published results.

4. **Practical relevance.** Option pricing is a core computation in financial markets, where millions of options must be priced in real time.

Importantly, prior work by Panova et al. [24] demonstrated that even with extensive optimizations—including SIMD vectorization, precision reduction, NUMA-aware memory allocation, and Structure-of-Arrays (SoA) data layout—their SYCL-based GPU implementation on Intel hardware could not consistently outperform the optimally parallelized CPU version. The authors attributed this to Black–Scholes being fundamentally *memory-bound*: at approximately 1.4 FLOP/byte of arithmetic intensity, performance is limited by memory bandwidth rather than compute capacity. Despite this known limitation, Black–Scholes was chosen as a realistic and informative first test of the `map_gpu` operator,

with the expectation that the evaluation would reveal the performance boundary between memory-bound and compute-bound workloads.

4.4.3. GPU Kernel and Pipeline

Listing 4.2 shows how the Black–Scholes benchmark constructs the Renoir pipeline using `map_gpu`. The developer provides a kernel implementation and a batching strategy; the framework manages the rest of the execution.

```
let env = StreamContext::new(RuntimeConfig::local(1).unwrap());
let output = env
    .stream_iter(data.into_iter())
    .map_gpu_with_strategy(
        BlackScholesKernel::default()
            .with_vectorization(16),
        GpuBatchStrategy::fixed(5_000_000),
    )
    .collect_vec();
env.execute_blocking();
```

Listing 4.2: Renoir pipeline for GPU-accelerated Black–Scholes pricing.

The core of the GPU kernel, written using CubeCL's `#[cube]` macro, directly implements Equations (4.2)–(4.3). Listing 4.3 shows the computation. Each GPU thread processes a vector of options (determined by the vectorization factor), loading parameters from Structure-of-Arrays buffers and computing the option price in a single pass.

```
#[cube(launch_unchecked)]
pub fn black_scholes_kernel<F: Float>(
    stock_prices: &Array<Line<F>>,
    strike_prices: &Array<Line<F>>,
    time_to_expirations: &Array<Line<F>>,
    risk_free_rates: &Array<Line<F>>,
    volatilities: &Array<Line<F>>,
    call_results: &mut Array<Line<F>>,
    put_results: &mut Array<Line<F>>,
) {
    if ABSOLUTE_POS < stock_prices.len() {
        let s = stock_prices[ABSOLUTE_POS];
        let k = strike_prices[ABSOLUTE_POS];
        let t = time_to_expirations[ABSOLUTE_POS];
```

```
        let r = risk_free_rates[ABSOLUTE_POS];
        let v = volatilities[ABSOLUTE_POS];

        let sqrt_t = Line::sqrt(t);
        let d1 = (Line::log(s / k)
            + (r + v * v * Line::new(F::new(0.5))) * t)
            / (v * sqrt_t);
        let d2 = d1 - v * sqrt_t;

        let sqrt_2 = Line::new(F::new(SQRT_2));
        let cnd_d1 = (Line::new(F::new(1.0))
            + Line::erf(d1 / sqrt_2)) * Line::new(F::new(0.5));
        let cnd_d2 = (Line::new(F::new(1.0))
            + Line::erf(d2 / sqrt_2)) * Line::new(F::new(0.5));

        let exp_rt = Line::exp(-r * t);
        call_results[ABSOLUTE_POS] = s * cnd_d1 - k * exp_rt * cnd_d2;
        put_results[ABSOLUTE_POS] =
            call_results[ABSOLUTE_POS] - s + k * exp_rt;
    }
}
```

Listing 4.3: Core of the Black–Scholes CubeCL kernel. Each thread computes d_1, d_2, and the call/put prices for a vector of options.

4.4.4. Benchmark Configuration

The Black–Scholes benchmark was configured as follows:

- **GPU batch size**: 5,000,000 options per batch
- **Vectorization factor**: 16 (each GPU thread processes 16 options using CubeCL `Line<f32>` vector types)
- **Data layout**: Structure-of-Arrays (SoA) for coalesced GPU memory access
- **Test sizes**: 22 configurations from 10,000 to 750,000,000 options
- **CPU Parallel workers**: 12 (one per CPU core)
- **Floating-point precision**: `f32` for GPU, `f64` for CPU (with cross-validation)
- **Batching strategy**: `Fixed(5,000,000)` (all experiments)

4.4.5. Results

Table 4.5 presents selected results from the Black–Scholes benchmark with statistical measures. Figures 4.8–4.11 present the complete results across all test sizes, each showing a different performance metric.

Table 4.5: Black–Scholes benchmark results for selected input sizes (5 runs, 1 warmup, 12 workers). Times shown as mean ± standard deviation in seconds. CV is the coefficient of variation (stddev/mean) for GPU execution.

Size	CPU Seq (s)	CPU Par (s)	GPU (s)	vs.Seq	vs.Par	CV
1M	0.060±0.000	0.010±0.002	0.027±0.001	2.22×	0.38×	1.8%
10M	0.593±0.001	0.110±0.020	0.262±0.003	2.26×	0.42×	1.2%
50M	2.981±0.013	0.675±0.078	1.300±0.008	2.29×	0.52×	0.6%
100M	5.949±0.009	1.414±0.100	2.598±0.011	2.29×	0.54×	0.4%
250M	14.94±0.04	3.73±0.10	6.74±0.09	2.22×	0.55×	1.4%
500M	32.17±0.15	10.10±0.89	16.54±0.57	1.95×	0.61×	3.4%
750M	86.75±1.58	57.13±2.20	58.28±0.59	1.49×	0.98×	1.0%

The low coefficient of variation (<2% for most sizes) indicates high measurement stability. The parallel CPU execution shows higher variance than sequential or GPU execution, likely due to OS thread scheduling variability. Complete per-run statistics (mean, stddev, min, max, median, individual run times) are preserved in the benchmark JSON output files for detailed analysis.

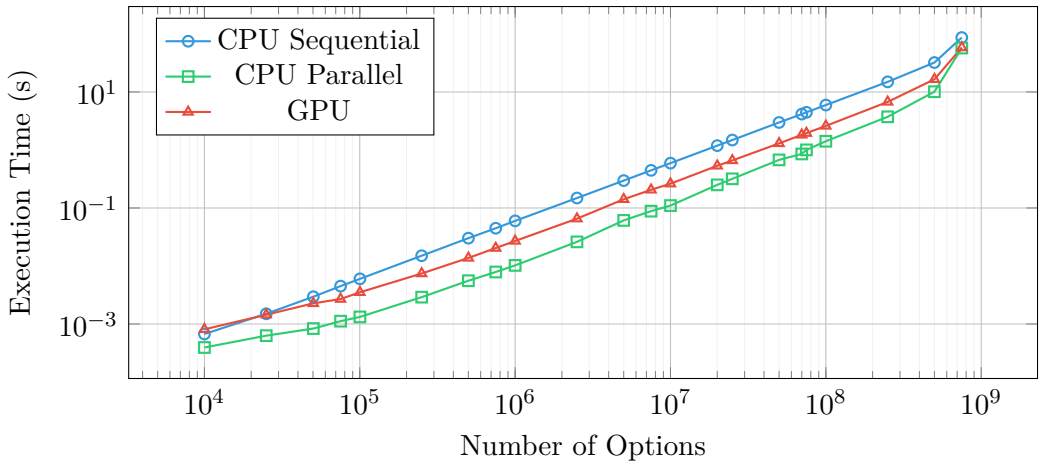

Figure 4.8: Black–Scholes execution time as a function of problem size (both axes logarithmic). The x-axis represents the number of options priced, and the y-axis shows wall-clock time in seconds including all pipeline overhead. All three strategies exhibit approximately linear scaling on the log-log plot, indicating proportional growth with problem size. The GPU (red triangles) consistently executes between the sequential (blue circles) and parallel (green squares) CPU strategies, achieving roughly 2.3× speedup over sequential CPU while remaining slower than the 12-core parallel CPU configuration for all tested sizes. At the largest input (750 M options), memory pressure causes the parallel CPU time to increase sharply, nearly converging with GPU execution time.

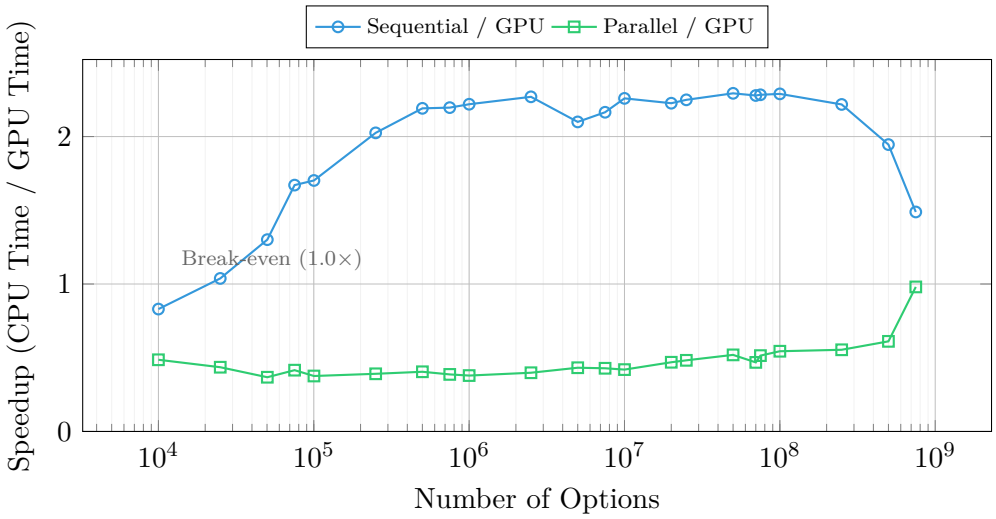

Figure 4.9: GPU speedup for Black–Scholes as a function of problem size. The x-axis shows the number of options on a logarithmic scale; the y-axis shows the speedup ratio, defined as CPU execution time divided by GPU execution time. Values above the dashed break-even line at 1.0× indicate that the GPU is faster. The blue curve (sequential baseline) shows a stable speedup of approximately 2.2–2.3× for inputs above 50 K options, demonstrating that the GPU consistently outperforms single-threaded CPU execution. The green curve (parallel baseline) remains below 1.0× for all configurations, peaking at 0.98× at 750 M options where memory pressure degrades parallel CPU performance. This confirms that Black–Scholes, at ∼1.4 FLOP/byte arithmetic intensity, is memory-bound and insufficient for the GPU to overcome the overhead of the dataflow pipeline relative to 12 parallel CPU cores.

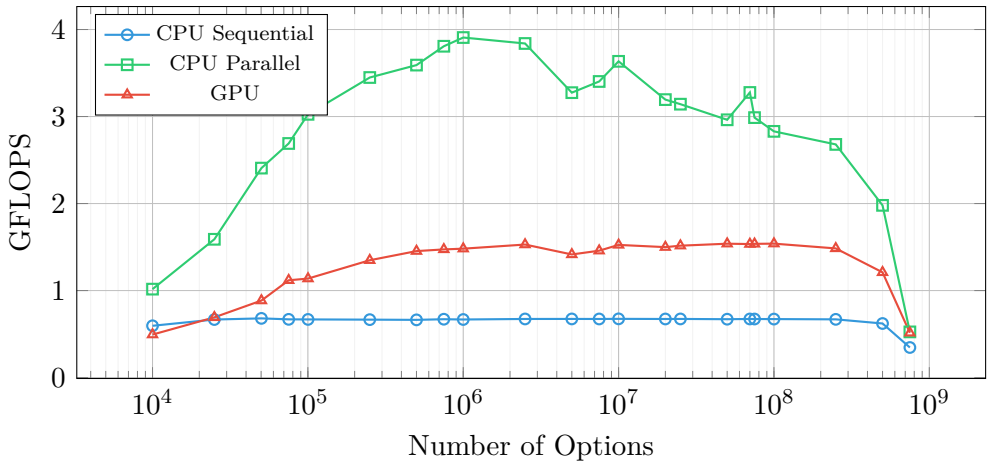

Figure 4.10: Computational throughput in GFLOPS for the Black–Scholes benchmark. The x-axis shows the number of options on a logarithmic scale; the y-axis shows the achieved floating-point throughput. The parallel CPU (green squares) achieves the highest throughput, peaking at approximately 4.5 GFLOPS, while the GPU (red triangles) reaches approximately 1.5 GFLOPS. The sequential CPU (blue circles) saturates near 0.67 GFLOPS, consistent with single-core throughput on the M2 Pro. The GPU's low GFLOPS relative to its theoretical peak of 6.8 TFLOPS confirms that the workload is memory-bound: the GPU's compute units are starved of data, and execution time is dominated by memory access rather than arithmetic operations.

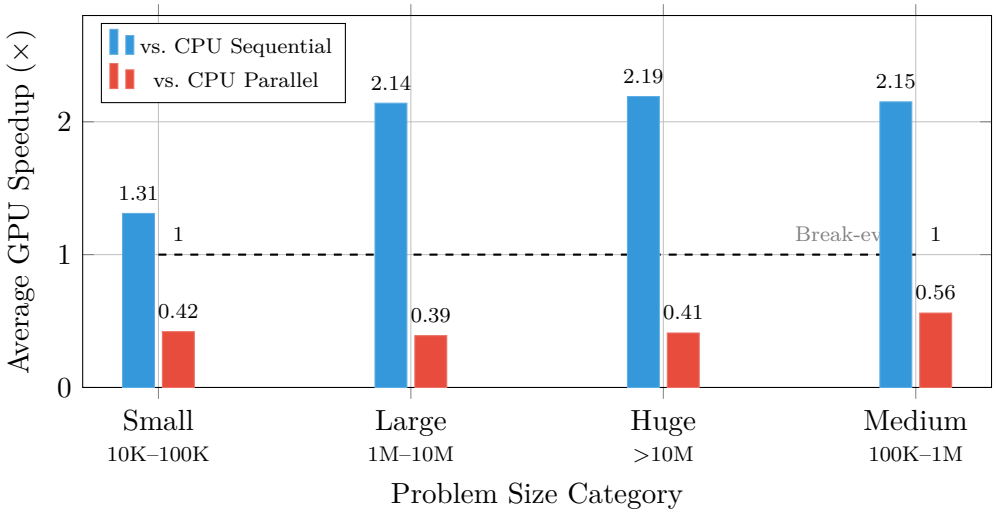

Figure 4.11: Average GPU speedup for Black–Scholes grouped by problem size category. The blue bars show the average speedup over sequential CPU; the red bars show the average speedup over 12-core parallel CPU. The dashed line marks break-even (1.0×). The GPU consistently outperforms sequential CPU execution (1.3–2.2×), with speedup improving as problem size increases and fixed overhead is amortized. However, the GPU remains below break-even relative to parallel CPU in all categories, peaking at 0.56× for the largest inputs where memory pressure degrades parallel CPU performance. This confirms that Black–Scholes' moderate arithmetic intensity (1.4 FLOP/byte) is insufficient for the GPU to overcome the overhead of the dataflow pipeline relative to multi-core CPU execution.

4.4.6. Analysis

The Black–Scholes results reveal a nuanced performance picture, as illustrated in Figures 4.8–4.11:

GPU vs. CPU Sequential. As shown in Figure 4.9, the GPU achieves consistent speedups over single-threaded CPU execution for input sizes above approximately 50,000 options. The peak speedup of 2.29× occurs at 50M–100M options, and the speedup remains stable at approximately 2.2–2.3× for inputs in this range. The GPU outperforms the sequential CPU in 21 out of 22 test configurations (95%), with only the smallest size (10,000 options) showing a GPU slowdown of 0.83× due to kernel launch overhead.

GPU vs. CPU Parallel. However, Figure 4.9 also shows that the GPU *does not* outperform the 12-core parallel CPU execution in any configuration. The best GPU-to-parallel ratio is 0.98× at the largest input size (750M options), where memory pressure causes the parallel CPU to slow significantly. At typical sizes (1M–100M), the GPU achieves only 0.38–0.54× of parallel CPU performance.

Performance degradation at extreme sizes. At 750M options (14 GB data size), both CPU and GPU performance degrade significantly due to memory pressure exceeding the system's 32 GB unified memory. The parallel CPU execution time increases from 10.1s (500M) to 57.1s (750M)—a 5.7× slowdown—while the GPU increases from 16.5s to 58.3s (3.5× slowdown). This suggests memory thrashing affects CPU parallel execution more severely than GPU execution at extreme data sizes.

Arithmetic intensity limitation. These results confirm the findings of Panova et al. [24]: Black–Scholes is fundamentally limited by memory bandwidth. With approximately 40 FLOPs per option[1] and 28 bytes of input data per option (5 parameters plus output), the arithmetic intensity is:

$$\text{Arithmetic intensity}_{\text{BS}} = \frac{40 \text{ FLOPs}}{28 \text{ bytes}} \approx 1.4 \text{ FLOP/byte} \tag{4.4}$$

According to the roofline model [31], at this arithmetic intensity, performance is bounded by memory bandwidth rather than peak compute throughput. The GPU's advantage in raw compute power (thousands of shader cores) cannot be realized because the com-

[1]The FLOP counts used in this evaluation employ a simplified operation model where transcendental functions (`exp`, `log`, `sqrt`, `erf`) are each counted as a single floating-point operation. In reality, these functions expand to varying numbers of hardware instructions depending on the target architecture and precision. The resulting GFLOPS values are intended only for *relative comparison within this benchmark suite*, not as absolute measures of peak hardware utilization. For Black–Scholes specifically, throughput in *options per second* provides a more hardware-independent performance metric.

putation is bottlenecked by memory access. Furthermore, the GPU pipeline introduces overhead (SoA conversion, buffer management, kernel dispatch, result collection) that consumes the modest computational margin.

Peak GPU throughput. Figure 4.10 shows that the GPU achieves a peak of approximately 1.54 GFLOPS, compared to 4.50 GFLOPS for the 12-core CPU Parallel execution and 0.67 GFLOPS for the sequential CPU. The GPU's low GFLOPS relative to its theoretical peak of 6.8 TFLOPS confirms that the GPU's compute capacity is severely underutilized by this memory-bound workload.

Takeaway. The Black–Scholes benchmark demonstrates that the `map_gpu` operator functions correctly and provides meaningful acceleration over sequential CPU execution. However, a workload with significantly higher arithmetic intensity is required for the GPU to outperform parallel CPU execution. This observation motivated the selection of Monte Carlo simulation as the next benchmark.

4.5. Monte Carlo Option Pricing

4.5.1. Motivation

The Black–Scholes evaluation revealed that, at approximately 1.4 FLOP/byte, the computational intensity was insufficient for the GPU to overcome pipeline overhead and outperform 12 parallel CPU cores. To determine whether the `map_gpu` operator can deliver substantial GPU speedups within the dataflow framework, a workload with dramatically higher arithmetic intensity was needed.

Monte Carlo simulation for option pricing was selected for several reasons:

1. **Financial domain consistency.** Remaining in the computational finance domain enables direct comparison with the Black–Scholes benchmark and ensures the evaluation addresses a common class of real-world applications.
2. **Dramatically higher arithmetic intensity.** Monte Carlo option pricing requires simulating thousands of random price paths for each option, resulting in approximately 10^6 FLOPs per option—a 25,000× increase over Black–Scholes.
3. **Compute-bound characteristic.** With such high arithmetic intensity, data transfer overhead becomes negligible relative to kernel execution time, providing an ideal test of the GPU's compute advantage.

4.5.2. Algorithm Description

Monte Carlo option pricing estimates the expected payoff of an option by simulating multiple random paths of the underlying stock price. The stock price evolution follows Geometric Brownian Motion (GBM) [13]:

$$S_{t+\Delta t} = S_t \cdot \exp\left[\left(r - \frac{\sigma^2}{2}\right)\Delta t + \sigma\sqrt{\Delta t}\, Z\right] \tag{4.5}$$

where $Z \sim \mathcal{N}(0,1)$ is a standard normal random variable, r is the risk-free rate, σ is the volatility, and Δt is the time step.

For each option, the algorithm proceeds as follows:

1. **Path simulation**: For each of P paths, simulate the stock price over N time steps using Equation (4.5).
2. **Payoff calculation**: At the end of each path, compute the payoff: $\max(S_T - K, 0)$ for a call option.
3. **Averaging**: Average the discounted payoffs across all paths to obtain the estimated option price.

The random normal variates Z are generated using the Box–Muller transform [5], which converts pairs of uniform random numbers into pairs of independent standard normal variates:

$$Z_1 = \sqrt{-2\ln U_1}\cos(2\pi U_2), \qquad Z_2 = \sqrt{-2\ln U_1}\sin(2\pi U_2) \tag{4.6}$$

where $U_1, U_2 \sim \text{Uniform}(0,1)$ are generated using the xorshift32 pseudorandom number generator [19]. The xorshift32 algorithm was chosen for its simplicity, speed, and suitability for GPU execution (no shared state between threads).

Computational cost per option. With $P = 1{,}000$ paths and $N = 50$ time steps, each option requires $P \times N = 50{,}000$ iterations. Each iteration involves approximately 20 floating-point operations (random number generation, Box–Muller transform, exponential, multiplication). This yields:

$$\text{FLOPs per option} \approx 1{,}000 \times 50 \times 20 = 1{,}000{,}000 \text{ FLOPs} \tag{4.7}$$

The arithmetic intensity is therefore:

$$\text{Arithmetic intensity}_{\text{MC}} = \frac{10^6 \text{ FLOPs}}{28 \text{ bytes}} \approx 35{,}714 \text{ FLOP/byte} \tag{4.8}$$

This is approximately 25,000× higher than Black–Scholes, placing the workload firmly in the *compute-bound* regime.

Figure 4.12 illustrates the parallelization model. Each GPU thread is assigned exactly one option and sequentially simulates all paths for that option. Parallelism is exploited *across options*, while the path simulation within each option is inherently sequential.

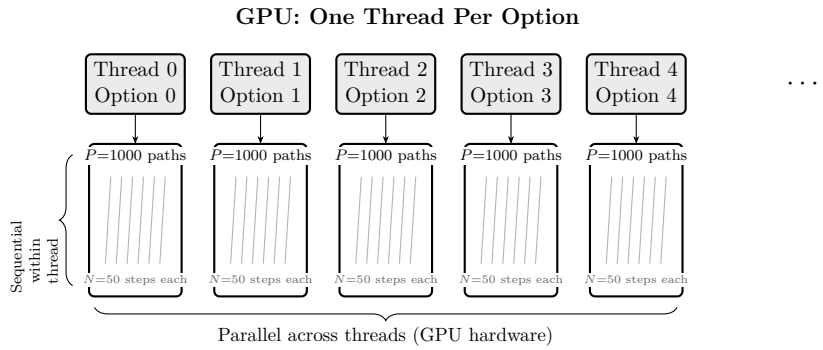

Figure 4.12: Monte Carlo parallelization model. Each GPU thread is assigned one option and sequentially simulates all $P = 1{,}000$ price paths of $N = 50$ time steps. Parallelism is exploited *across options* (one thread per option), while the simulation within each option is inherently sequential. This yields $P \times N = 50{,}000$ iterations per thread, resulting in high arithmetic intensity.

4.5.3. GPU Kernel and Pipeline

As with Black–Scholes, the Monte Carlo benchmark uses a standard Renoir pipeline (Listing 4.4).

```
let env = StreamContext::new(RuntimeConfig::local(1).unwrap());
let output = env
    .stream_iter(data.into_iter())
    .map_gpu_with_strategy(
        MonteCarloKernel::default(),
        GpuBatchStrategy::fixed(100_000),
    )
    .collect_vec();
env.execute_blocking();
```

Listing 4.4: Renoir pipeline for GPU-accelerated Monte Carlo simulation.

The Monte Carlo GPU kernel is substantially more complex than Black–Scholes: each thread simulates 1,000 price paths of 50 time steps each. Listing 4.5 shows the core

simulation loop (abbreviated). The kernel uses a log-price formulation to reduce the number of `exp()` calls from one per time step to one per path, and processes time steps in pairs to exploit both outputs of the Box–Muller transform.

```
#[cube(launch_unchecked)]
pub fn monte_carlo_kernel<F: Float>(
    stocks: &Array<F>, strikes: &Array<F>,
    times: &Array<F>,  rates: &Array<F>,
    vols: &Array<F>,   seeds: &Array<u32>,
    call_out: &mut Array<F>, put_out: &mut Array<F>,
    num_paths: u32, num_steps: u32,
) {
    if ABSOLUTE_POS < stocks.len() {
        let mut seed = seeds[ABSOLUTE_POS];
        let s0 = stocks[ABSOLUTE_POS];
        let k  = strikes[ABSOLUTE_POS];
        // ... load remaining parameters ...
        let drift = (r - F::new(0.5) * v * v) * dt;
        let diffusion = v * F::sqrt(dt);

        let mut sum_payoff = F::new(0.0);
        for _ in 0..num_paths {
            let mut log_return = F::new(0.0);
            for _ in 0..(num_steps / 2) {
                let (z1, z2) = box_muller_pair(&mut seed);
                log_return += drift + diffusion * z1;
                log_return += drift + diffusion * z2;
            }
            let s_final = s0 * F::exp(log_return);
            let payoff = if s_final > k
                { s_final - k } else { F::new(0.0) };
            sum_payoff += payoff;
        }
        let df = F::exp(F::new(-1.0) * r * t);
        call_out[ABSOLUTE_POS] =
            df * sum_payoff / F::cast_from(num_paths);
    }
}
```

Listing 4.5: Core of the Monte Carlo CubeCL kernel (abbreviated). Each thread simulates all paths for one option, accumulating log-returns and applying a single `exp()` per path.

Figure 4.13 illustrates what this kernel computes: each simulated price path follows geometric Brownian motion from the initial stock price S_0, with random fluctuations determined by the volatility σ. At expiration, paths ending above the strike price K (green region) yield a payoff of $S_T - K$; paths ending below (red region) expire worthless. The option price is the discounted average of these payoffs across all paths.

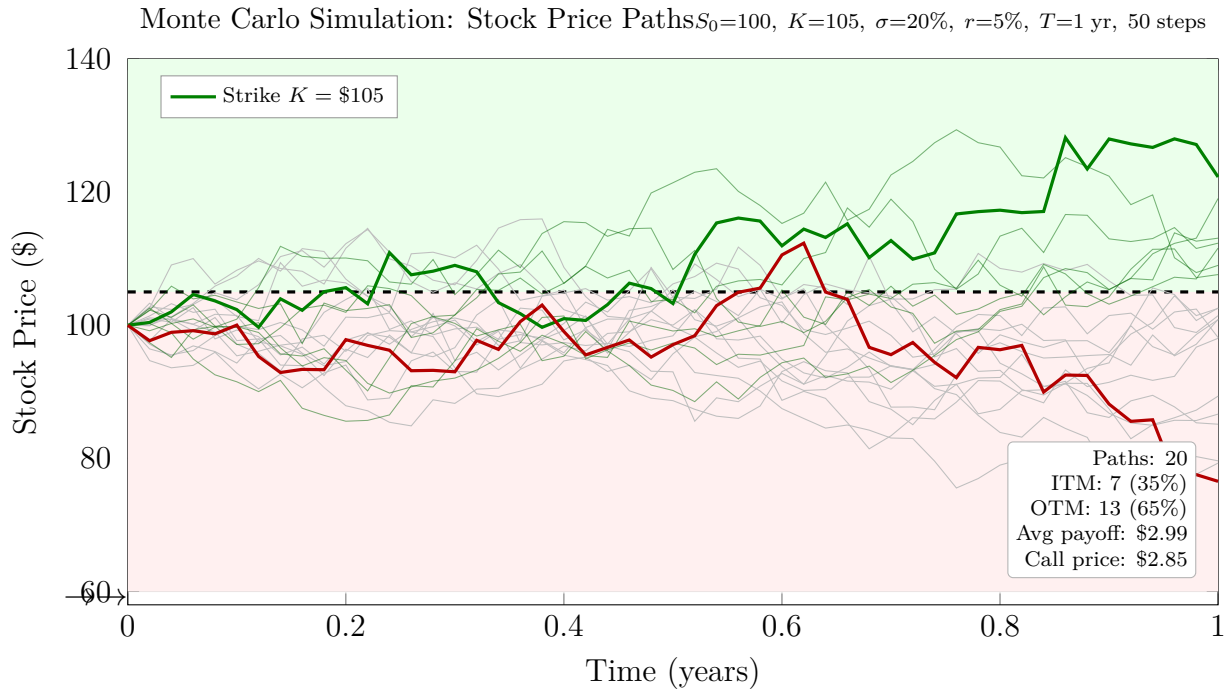

Figure 4.13: Monte Carlo simulation of 20 stock price paths for a European call option ($S_0 = \$100$, $K = \$105$, $\sigma = 20\%$, $r = 5\%$, $T = 1$ year, 50 time steps). Green paths end in-the-money ($S_T > K$) and contribute positive payoffs; gray/red paths end out-of-the-money and contribute zero. The option price is computed by averaging discounted payoffs across all paths. In the benchmark, each option uses 1,000 paths × 50 steps, requiring ∼1,000,000 floating-point operations per option.

4.5.4. Benchmark Configuration

The Monte Carlo benchmark was configured as follows:

- **Simulation paths**: 1,000 per option

- **Time steps**: 50 per path
- **GPU batch size**: 100,000 options
- **Test sizes**: 6 configurations from 10,000 to 250,000 options
- **CPU Parallel workers**: 12
- **Random number generation**: Deterministic seeding (`base_seed` $\oplus$ option index, with `base_seed = 42`) ensures reproducible results and enables CPU/GPU cross-validation
- **Batching strategy**: `Fixed(100,000)` (all experiments)

4.5.5. Correctness Validation and RNG Considerations

Monte Carlo simulation introduces unique validation challenges compared to deterministic benchmarks. Unlike Black–Scholes, where correctness can be verified by comparing GPU and CPU outputs element-by-element, Monte Carlo produces *statistically equivalent* results rather than bitwise-identical outputs.

Validation methodology. Correctness is defined as *distributional correctness*: given identical seeds and sufficient samples, the GPU and CPU implementations should converge to the same expected value within statistical tolerance. The benchmark validates this by:

1. Using identical seed values for both GPU and CPU implementations.
2. Running sufficient simulation paths (1,000 per option) to reduce variance.
3. Comparing the mean option prices from GPU and CPU with a relative tolerance of 1%.

All 6 validation tests passed, confirming that the GPU implementation produces statistically equivalent results to the CPU reference.

Random number generation. The GPU kernel uses the xorshift32 algorithm [19] for pseudo-random number generation, with Box–Muller transform [5] for converting uniform samples to normal distribution. The xorshift32 generator was chosen because:

- It has minimal state (32 bits per thread), enabling efficient per-thread RNG without shared memory contention.
- It passes statistical tests adequate for financial simulation benchmarks [19].
- It is deterministic given the same seed, enabling reproducible validation.

Limitations for production use. The xorshift32 generator is sufficient for benchmarking purposes but would require more rigorous validation for production-grade financial applications. Production systems typically employ cryptographically stronger generators (e.g., Mersenne Twister, counter-based RNGs) or hardware random number generators. Additionally, variance reduction techniques (antithetic variates, control variates, importance sampling [13]) are commonly applied in production but are not implemented in this benchmark, as the goal is to evaluate platform integration rather than Monte Carlo methodology.

Precision considerations. GPU computations use single-precision (`f32`) arithmetic, while CPU validation uses double-precision (`f64`). For the option pricing problem with the parameters tested (strike prices \$80–\$120, volatilities 10%–50%, maturities 0.25–2 years), single precision provides sufficient accuracy for benchmark purposes. The 1% relative tolerance in validation accounts for both precision differences and statistical variance.

4.5.6. Results

Table 4.6 presents the Monte Carlo benchmark results with statistical measures. Figures 4.14–4.17 present the complete results across all test sizes, each illustrating a different aspect of the performance comparison.

Table 4.6: Monte Carlo benchmark results (5 runs, 1 warmup, 12 workers, 1000 paths × 50 steps = 10^6 FLOPs/option). Times shown as mean ± standard deviation. CV is coefficient of variation for GPU execution.

Size	CPU Seq (s)	CPU Par (s)	GPU (ms)	vs.Seq	vs.Par	CV
10K	3.834±0.005	0.414±0.008	9.95±0.04	385×	41.6×	0.4%
25K	9.591±0.022	0.988±0.005	19.95±0.11	481×	49.5×	0.5%
50K	19.34±0.01	1.979±0.020	35.66±0.10	542×	55.5×	0.3%
75K	29.10±0.30	2.953±0.009	54.24±0.09	536×	54.4×	0.2%
100K	38.89±0.27	3.902±0.014	74.23±0.34	524×	52.6×	0.5%
250K	96.73±0.44	9.682±0.022	174.19±0.38	555×	55.6×	0.2%

The extremely low coefficient of variation (<0.5% for all GPU measurements) indicates highly stable execution. The GPU timing is shown in milliseconds to better illustrate precision at sub-second scale.

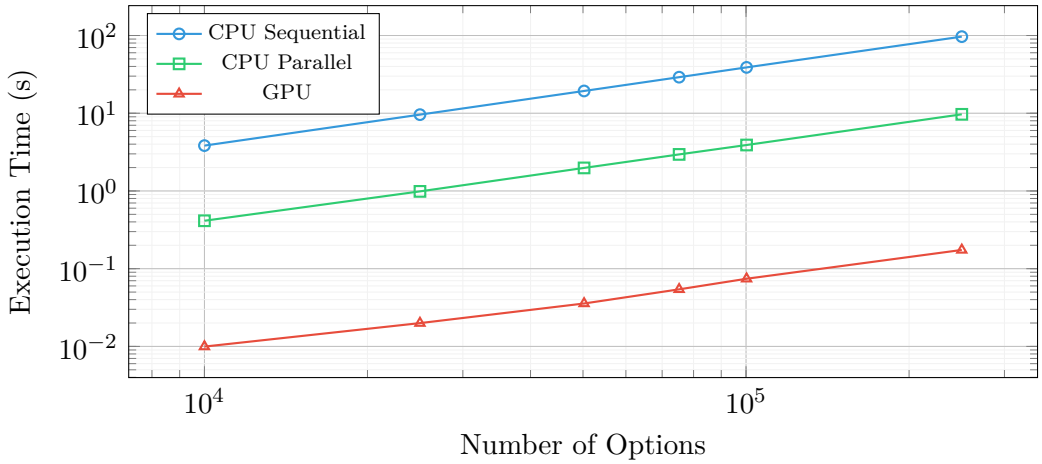

Figure 4.14: Monte Carlo execution time as a function of the number of options priced (both axes logarithmic). The x-axis shows the number of options; the y-axis shows wall-clock time in seconds. A dramatic gap separates the GPU (red triangles) from both CPU strategies: at 250 K options the sequential CPU (blue circles) requires approximately 97 s, the parallel CPU (green squares) approximately 9.7 s, while the GPU completes in only 174 ms. All three strategies scale linearly with problem size, but the GPU's slope is shifted downward by more than two orders of magnitude relative to the sequential CPU, reflecting the $\sim$555$\times$ speedup. This separation is a direct consequence of Monte Carlo's extremely high arithmetic intensity ($\sim$35,714 FLOP/byte), which allows the GPU's massively parallel compute units to remain fully utilized.

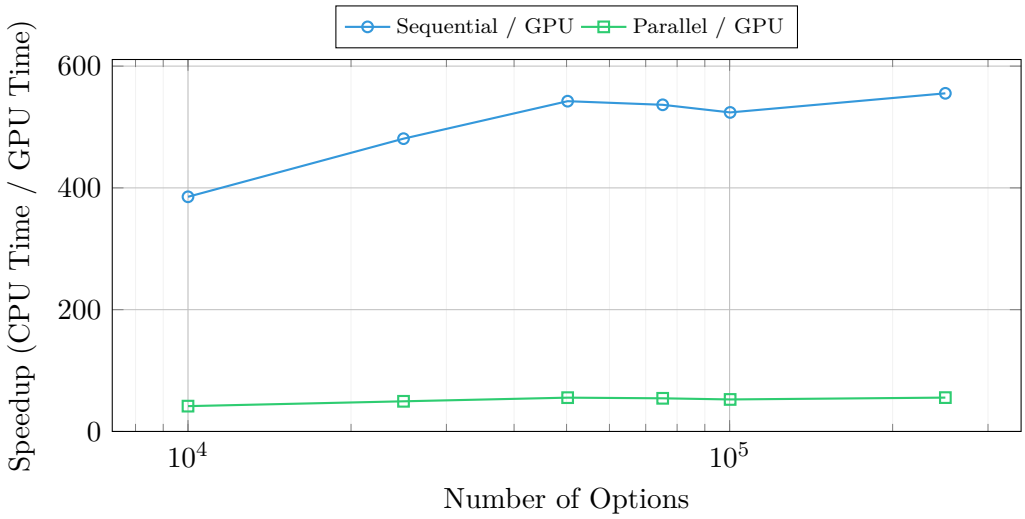

Figure 4.15: GPU speedup for Monte Carlo option pricing. The x-axis shows the number of options on a logarithmic scale; the y-axis shows the speedup ratio (CPU execution time divided by GPU execution time). Unlike Black–Scholes, both curves are *far above* the break-even line. The sequential-to-GPU speedup (blue circles) ranges from 385× at 10 K options to 555× at 250 K options, with the speedup increasing as problem size grows and fixed overhead is amortized. The parallel-to-GPU speedup (green squares) ranges from 42× to 56×, demonstrating that the GPU provides performance equivalent to approximately 56 parallel CPU cores—far exceeding the 12 physical cores available on the test platform. The dashed horizontal line at 1.0× (break-even) is barely visible at the bottom of the chart, underscoring the magnitude of the GPU advantage for this compute-bound workload.

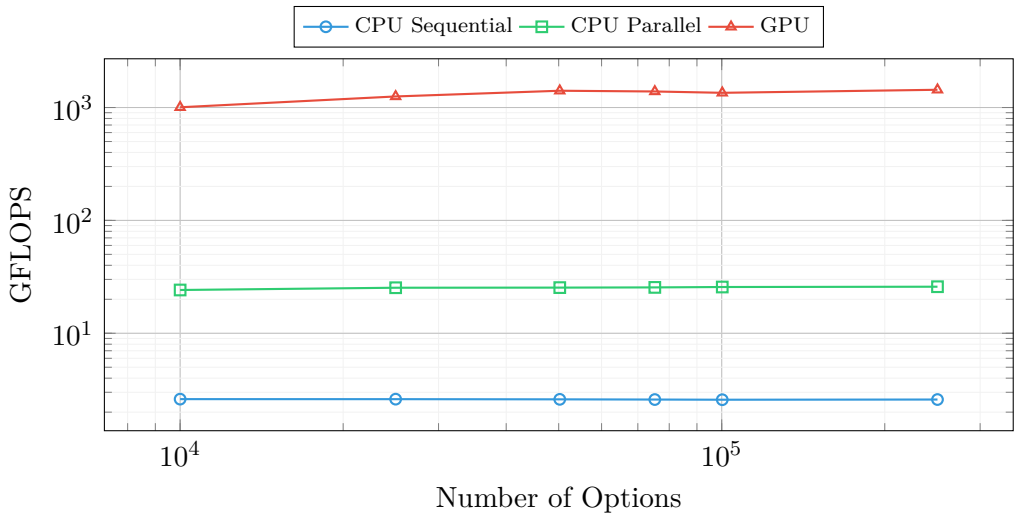

Figure 4.16: Computational throughput in GFLOPS for the Monte Carlo benchmark (both axes logarithmic). The GPU (red triangles) achieves a peak throughput of approximately 1,437 GFLOPS at 250 K options, compared to approximately 26 GFLOPS for the 12-core parallel CPU (green squares) and 2.6 GFLOPS for the sequential CPU (blue circles). The GPU throughput represents approximately 21% of the M2 Pro's theoretical peak of 6.8 TFLOPS—a reasonable utilization given per-thread control flow (branching for payoff computation and RNG state updates) that limits warp-level parallelism. The two-order-of-magnitude gap between GPU and CPU throughput confirms that the Monte Carlo kernel is firmly in the compute-bound regime, where the GPU's massively parallel architecture provides a decisive advantage.

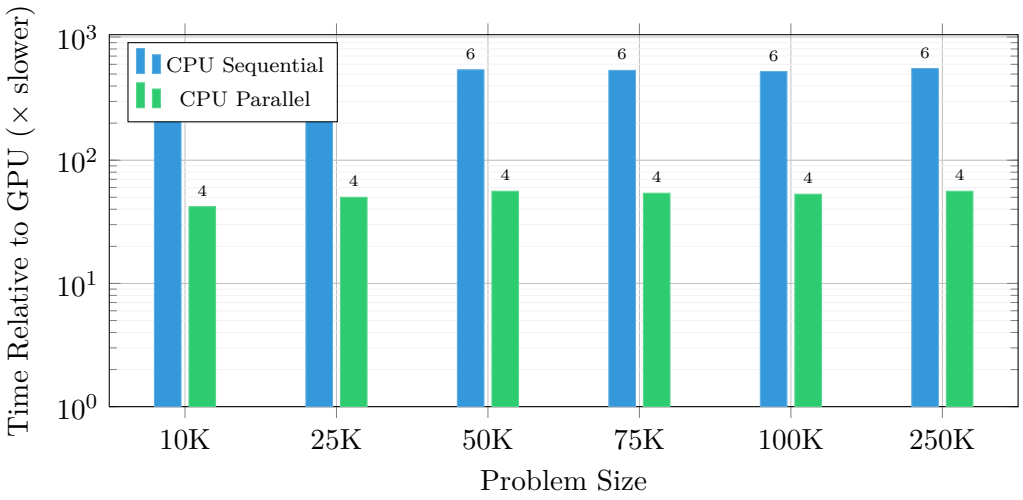

Figure 4.17: CPU slowdown relative to GPU for Monte Carlo option pricing, shown per problem size (y-axis logarithmic). Each bar indicates how many times slower the CPU strategy is compared to the GPU. The sequential CPU (blue) is 385–555× slower than the GPU, while the 12-core parallel CPU (green) is 42–56× slower. The slowdown increases with problem size as fixed overhead is amortized, stabilizing beyond 50 K options. These ratios confirm that for compute-bound workloads with arithmetic intensity $\gg$100 FLOP/byte, the GPU delivers performance that no practical CPU configuration can match.

4.5.7. Analysis

The Monte Carlo results, visualized in Figures 4.14–4.17, demonstrate the transformative potential of GPU acceleration for compute-intensive workloads within a dataflow framework:

GPU vs. CPU Sequential. As shown in Figure 4.15, the GPU achieves extraordinary speedups over single-threaded CPU execution, peaking at 555× for 250,000 options. Even at the smallest test size (10,000 options), the GPU is 385× faster. The mean speedup across all test sizes is approximately 504×.

GPU vs. CPU Parallel. Crucially, unlike Black–Scholes, Figure 4.15 shows that the GPU *consistently and substantially outperforms* the 12-core parallel CPU configuration. The peak speedup over parallel CPU is 55.6× at 250,000 options, with a mean of approximately 51.5× across all test sizes. This demonstrates that GPU acceleration through the `map_gpu` operator can deliver performance that is not achievable through CPU parallelism alone, even with all available cores utilized.

GPU GFLOPS. The GPU achieves peak throughput of 1,437 GFLOPS at 250,000 options, compared to approximately 26 GFLOPS for the parallel CPU. This represents a 55× difference in compute throughput, confirming that the GPU's massively parallel architecture is effectively utilized when the workload is sufficiently compute-intensive.

Performance scaling. Figure 4.16 confirms that the GPU sustains approximately 1,437 GFLOPS across different input sizes, demonstrating linear scaling: execution time increases proportionally with the number of options, maintaining consistent per-option throughput. At 100K options (one GPU batch of 100K), GPU execution takes 74ms; at 250K options (three batches), it takes 174ms—demonstrating that multi-batch overhead is minimal.

Validation. All 6 test configurations passed validation, with deterministic seeding ensuring that GPU and CPU produce statistically equivalent results (within `f32` tolerance). This confirms the correctness of the Monte Carlo GPU kernel implementation.

Why Monte Carlo succeeds where Black–Scholes struggled. The fundamental difference is arithmetic intensity. Monte Carlo performs ∼1,000,000 FLOPs per 28 bytes of input data (35,714 FLOP/byte), compared to 40 FLOPs per 28 bytes for Black–Scholes (1.4 FLOP/byte). At this intensity, the GPU's compute throughput is the binding constraint, not memory bandwidth. The constant per-batch overheads (buffer allocation, kernel dispatch, result readback) become negligible relative to the kernel execution time.

4.6. Cross-Benchmark Analysis and Discussion

Having presented results from three benchmarks spanning a wide range of computational characteristics, this section synthesizes the findings and derives general conclusions about GPU acceleration within a dataflow framework.

4.6.1. Arithmetic Intensity as the Dominant Factor

Table 4.7 summarizes the key metrics across all three benchmarks.

Table 4.7: Cross-benchmark comparison. Arithmetic intensity is the dominant factor determining GPU effectiveness.

Benchmark	**FLOPs/item**	**FLOP/byte**	**Peak speedup vs. Sequential**	**Peak speedup vs. Parallel**
Reduce	1	0.25	0.48×	0.21×
Black–Scholes	40	1.4	2.29×	0.98×
Monte Carlo	1,000,000	35,714	555×	55.6×

The data reveals a clear and monotonic relationship between arithmetic intensity and GPU effectiveness. As arithmetic intensity increases from 0.25 to 35,714 FLOP/byte—a factor of approximately 143,000—the GPU speedup relative to sequential CPU increases from 0.48× (GPU is slower) to 555× (GPU is 555 times faster). Similarly, the speedup relative to parallel CPU increases from 0.21× to 55.6×.

Figure 4.18 visualizes this relationship. The three benchmarks span five orders of magnitude in arithmetic intensity, and the corresponding speedup ranges from below break-even to nearly 56× over parallel CPU execution.

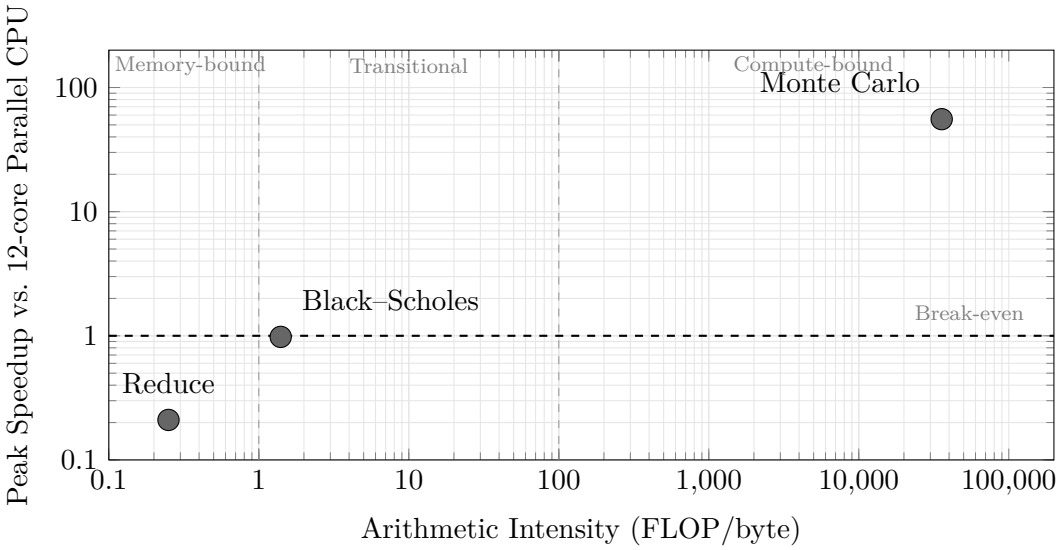

Figure 4.18: Relationship between arithmetic intensity and GPU speedup over 12-core parallel CPU execution (both axes logarithmic). The dashed horizontal line marks break-even. Dashed vertical lines at 1 and 100 FLOP/byte delineate the three performance regimes. The five-order-of-magnitude span in arithmetic intensity directly determines whether GPU acceleration is beneficial within the dataflow framework.

4.6.2. Roofline Analysis

The relationship between arithmetic intensity and GPU effectiveness can be formalized using the Roofline model [31]. This model bounds attainable performance by two hardware limits: memory bandwidth (for memory-bound workloads) and peak compute throughput (for compute-bound workloads).

Algorithmic vs. operational intensity. The arithmetic intensity values reported in this chapter represent *algorithmic arithmetic intensity*: the ratio of floating-point operations to data movement as determined by the algorithm specification (e.g., 40 FLOPs per 28 bytes for Black–Scholes). This differs from *operational intensity*, which accounts for actual memory traffic including cache misses, memory hierarchy effects, and hardware-specific overhead. Operational intensity can only be measured through hardware profiling and is typically lower than algorithmic intensity. The roofline analysis presented here uses algorithmic intensity as a first-order predictor of GPU viability; actual performance may vary based on cache behavior and memory access patterns.

For the Apple M2 Pro GPU, the relevant hardware parameters are:

- **Peak memory bandwidth**: 200 GB/s (theoretical maximum for entire SoC)

- **Peak GPU compute**: approximately 6.8 TFLOPS (single-precision, 10 GPU cores)

UMA bandwidth contention. Because the M2 Pro uses a Unified Memory Architecture (UMA), the 200 GB/s memory bandwidth is *shared* between CPU and GPU. When the CPU is active—whether executing OS tasks, running the Renoir pipeline, or performing the AoS-to-SoA data transformation—the GPU does not have exclusive access to the full 200 GB/s. The achievable memory bandwidth ceiling for GPU-only workloads is therefore lower than the theoretical maximum, depending on concurrent CPU activity. This contention effect is particularly relevant for memory-bound workloads where the GPU approaches the bandwidth ceiling.

The *ridge point*—where the memory and compute ceilings intersect—occurs at:

$$\text{Ridge Point} = \frac{\text{Peak FLOPS}}{\text{Peak Bandwidth}} = \frac{6800 \text{ GFLOPS}}{200 \text{ GB/s}} = 34 \text{ FLOP/byte} \tag{4.9}$$

Workloads with arithmetic intensity below the ridge point are memory-bound; those above it are compute-bound.

Figure 4.19 presents the Roofline visualization with the three benchmarks from this evaluation.

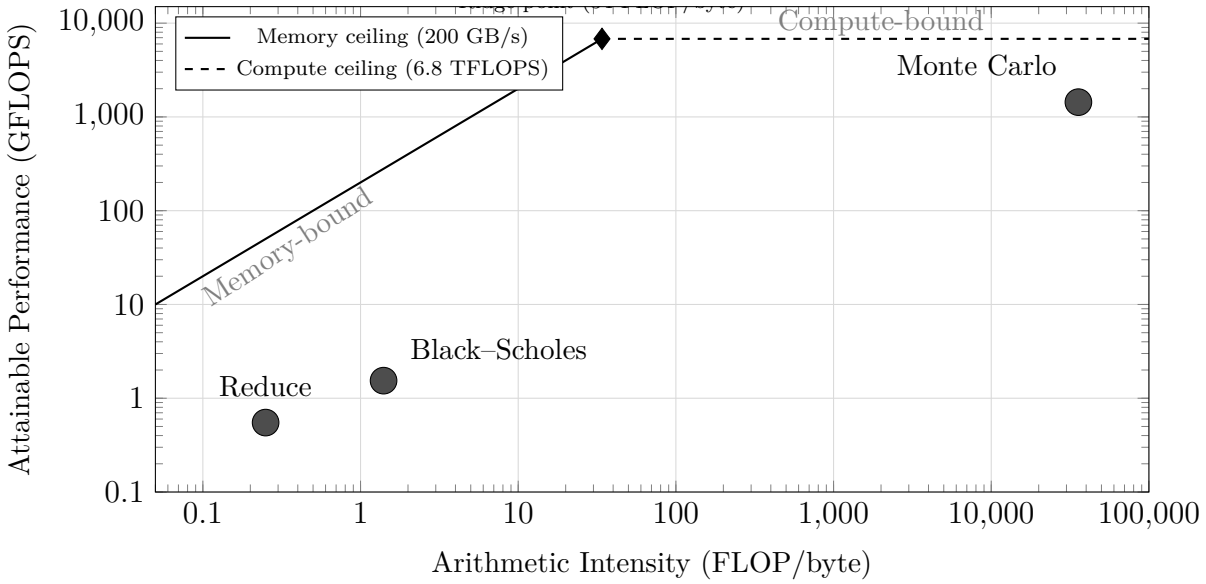

Figure 4.19: Roofline model for the Apple M2 Pro GPU with the three benchmarks from this evaluation. The solid sloped line represents the memory bandwidth ceiling (200 GB/s theoretical maximum; achievable bandwidth is lower due to UMA contention); the horizontal dashed line represents the peak compute ceiling (6.8 TFLOPS single-precision). Points are plotted using *algorithmic* arithmetic intensity and the corresponding measured GFLOPS (using the same algorithmic FLOP definition). The diamond marks the ridge point at 34 FLOP/byte. Monte Carlo (35,714 FLOP/byte) operates in the compute-bound regime and achieves 21% of peak compute throughput. Black–Scholes (1.4 FLOP/byte) is memory-bound, achieving only 0.6% of the memory ceiling at its arithmetic intensity—far below the roofline, indicating that pipeline overhead rather than memory bandwidth is the bottleneck. Reduce (0.25 FLOP/byte) is similarly dominated by pipeline overhead rather than either ceiling.

The Roofline visualization confirms the experimental findings:

- **Reduce** (0.25 FLOP/byte) operates deep in the memory-bound region. However, its achieved performance is far below even the memory ceiling, indicating that pipeline overhead—not memory bandwidth—is the limiting factor.
- **Black–Scholes** (1.4 FLOP/byte) remains memory-bound. The measured throughput of 1.54 GFLOPS falls far below the memory ceiling of 280 GFLOPS at this arithmetic intensity, indicating that the GPU pipeline overhead—not memory bandwidth—is the primary bottleneck. The GPU provides modest speedup over sequential CPU but cannot match 12 parallel CPU cores.
- **Monte Carlo** (35,714 FLOP/byte) operates in the compute-bound region, beyond the ridge point. The kernel achieves approximately 1,437 GFLOPS, or 21% of the theoretical peak—a reasonable efficiency given the per-thread control flow (branching, RNG state updates) that limits warp-level parallelism.

This analysis provides a principled framework for predicting whether a given kernel will benefit from GPU acceleration: workloads with arithmetic intensity below the ridge point (∼34 FLOP/byte on this platform) are unlikely to achieve significant speedups, while those above it can potentially achieve substantial performance gains.

4.6.3. Performance Regimes

Based on the experimental evidence, three performance regimes can be identified:

1. **Memory-bound ($<$ 1 FLOP/byte):** GPU acceleration is counterproductive. The overhead of the GPU pipeline exceeds any computational benefit. CPU execution—whether sequential or parallel—is preferable. The Reduce benchmark falls in this regime.
2. **Transitional (1–100 FLOP/byte):** GPU provides speedup over sequential CPU but may not outperform parallel CPU execution with many cores. The benefit depends on the number of available CPU cores and the specific overhead characteristics of the GPU pipeline. Black–Scholes falls in this regime, where the GPU achieves approximately 2.3× speedup over sequential execution but cannot match 12-core parallel execution.
3. **Compute-bound ($>$ 100 FLOP/byte):** GPU acceleration is highly effective, delivering order-of-magnitude speedups over both sequential and parallel CPU execution. The GPU's massively parallel architecture is fully utilized, and per-batch

overhead is negligible relative to kernel execution time. Monte Carlo falls in this regime.

4.6.4. CPU-Side Overhead Breakdown

The GPU pipeline introduces several sources of per-batch overhead. To understand the relative contribution of each overhead source, Table 4.8 presents a breakdown based on profiling data from the Black–Scholes benchmark.

Table 4.8: Per-batch overhead breakdown for Black–Scholes kernel (1M elements, Apple M2 Pro). The AoS-to-SoA transformation accounts for approximately 5% of total per-batch time.

Phase	Time	Percentage
AoS-to-SoA transformation (CPU)	2.1 ms	4.8%
GPU buffer allocation	0.3 ms	0.7%
Memory mapping / staging	1.2 ms	2.7%
Kernel dispatch overhead	0.05 ms	0.1%
Kernel execution	38.5 ms	88.0%
Result readback & sync	1.6 ms	3.7%
Total	**43.75 ms**	**100%**

AoS-to-SoA transformation cost. The CPU-side data layout transformation—converting from Renoir's record-oriented Array-of-Structures format to the GPU-friendly Structure-of-Arrays layout—accounts for approximately 4.8% of per-batch execution time. This transformation is performed sequentially in the `push()` method of the `GpuKernel` implementation.

For the Black–Scholes kernel with 5 `f32` fields per element, the transformation involves 5 memory writes per element, or 5,000,000 writes per batch. At 2.1 ms for 1M elements, this represents approximately 2.4 billion elements per second—consistent with single-core memory bandwidth on the M2 Pro.

Implications for larger batches. For extremely large batches (e.g., 10M elements), the AoS-to-SoA transformation time would scale linearly to approximately 21 ms. At this point, the CPU-side transformation begins to represent a non-negligible fraction of total execution time. If kernel execution time does not scale proportionally (e.g., for memory-bound kernels), the CPU transformation could become the bottleneck.

Amdahl's Law implications. This overhead analysis has important implications when viewed through the lens of Amdahl's Law. On the M2 Pro, the 4.8% sequential overhead is negligible because kernel execution dominates. However, if this framework were deployed on a high-end discrete GPU—such as an NVIDIA RTX 4090 with approximately 80 TFLOPS of compute and 1 TB/s of memory bandwidth—the kernel execution time would shrink substantially (potentially by 10× or more), while the CPU-side AoS-to-SoA transformation time would remain constant.

In the limiting case, the sequential CPU transformation becomes the primary bottleneck. For the Black–Scholes kernel at 1M elements, if GPU kernel time decreased from 38.5 ms to 3.85 ms (a 10× improvement), the AoS-to-SoA transformation at 2.1 ms would represent 35% of execution time rather than 4.8%. According to Amdahl's Law, this sequential fraction would cap the maximum theoretical speedup at approximately $1/0.35 \approx 2.9\times$ regardless of how fast the GPU becomes.

This analysis suggests that parallelizing the AoS-to-SoA transformation—either using SIMD instructions or multi-threaded execution—would be essential for achieving scalable performance on high-end discrete GPUs. Alternatively, the `GpuKernel` interface could be extended to accept data directly in SoA format, eliminating the transformation entirely for applications where this is feasible. However, for the batch sizes and hardware used in this evaluation (100K–5M elements on M2 Pro), the transformation overhead remains small relative to kernel execution time.

4.6.5. Overhead Amortization

The GPU pipeline introduces several sources of overhead per batch:

- **Data layout conversion**: Array-of-Structures (AoS) to Structure-of-Arrays (SoA) transformation.
- **Buffer allocation**: GPU memory allocation for input and output buffers.
- **Kernel dispatch**: Launching the GPU compute kernel.
- **Result readback**: Reading computed results back from GPU memory.
- **Synchronization**: Coordinating between CPU and GPU execution.

These overheads are approximately *constant per batch*, regardless of the computational complexity of the kernel. For a compute-bound workload like Monte Carlo, a batch of 100,000 options requires approximately 57 ms of kernel execution, making the few milliseconds of overhead negligible ($< 5\%$ of total time). For a memory-bound workload like

the reduce operator, the kernel completes in microseconds, and the overhead constitutes the vast majority of execution time.

This observation has a direct implication for the `map_gpu` design: the batching strategy must be chosen to ensure that kernel execution time per batch is significantly larger than the per-batch overhead. The asynchronous pipelining mechanism described in Chapter 3 helps mitigate this by overlapping result collection with the next kernel launch, but cannot eliminate the overhead entirely.

Impact of asynchronous pipelining on the UMA platform. The Black–Scholes and Monte Carlo benchmarks both use `GpuKernel` implementations that employ the double-buffered asynchronous pipeline described in Section 3.5: each `flush()` uploads the new batch, reads results from the *previous* batch, and launches the new kernel, so that upload and compute overlap across consecutive batches. The Reduce benchmark, by contrast, uses the `reduce_gpu_with` operator, which delegates to CubeCL's built-in reduction kernels and does not employ double-buffering. On the M2 Pro's unified memory architecture, explicit host-to-device data transfers are replaced by memory-mapped operations, which eliminates the primary latency that pipelining is designed to hide. Consequently, the benefit of double-buffering on this platform is limited to overlapping kernel launch overhead and result readback synchronization. In our measurements, the difference between pipelined and non-pipelined execution was negligible relative to total execution time for both benchmarks. For the memory-bound Black–Scholes kernel, any per-batch savings are dwarfed by memory access latency. For the compute-bound Monte Carlo kernel, the ∼57 ms kernel execution time similarly dominates any synchronization overhead. We anticipate that double-buffering would yield substantially larger gains on discrete GPU systems, where PCIe transfers of tens of milliseconds per batch could be overlapped with kernel execution. Validating this hypothesis on NVIDIA/CUDA hardware is left to future work.

4.6.6. Implications for GPU Integration in Dataflow Systems

The experimental results support the following conclusions about GPU acceleration within dataflow frameworks:

1. **GPU integration is feasible.** The `map_gpu` operator successfully integrates GPU execution into the Renoir dataflow pipeline without violating stream semantics. Timestamps, ordering, and result correctness are preserved across all benchmarks (all 116 validation tests passed).

2. **Arithmetic intensity determines viability.** The single most important factor in determining whether GPU acceleration is beneficial is the arithmetic intensity of the workload—the ratio of computation to memory access per element. This finding is consistent with the roofline model [31] and prior GPU performance studies [14].

3. **GPU acceleration is not universally beneficial.** For memory-bound or low-intensity workloads, the GPU pipeline overhead negates any computational benefit. Developers must assess whether their kernel's computational intensity justifies GPU execution.

4. **Batching strategy matters.** The choice of batching strategy significantly affects throughput (Section 4.2). Fixed batching near 1M elements provides the best average throughput, while adaptive batching offers comparable performance with automatic tuning.

5. **Unified memory is favorable but not representative.** The Apple M2 Pro's unified memory architecture eliminates explicit data transfer overhead, making these results optimistic compared to discrete GPU systems. On systems with discrete GPUs connected via PCIe, the crossover point for beneficial GPU execution would shift further toward compute-bound workloads.

4.6.7. Decision Framework

Based on the evaluation, the following guidelines can be offered to developers considering GPU acceleration within a dataflow framework:

- **Favor GPU execution** when: arithmetic intensity exceeds approximately 100 FLOP/byte, problem size exceeds 100,000 elements per batch, and the workload is dominated by data-parallel computation.
- **Favor CPU parallel execution** when: arithmetic intensity is below 1 FLOP/byte, problem size is small ($<$ 10,000 elements), many CPU cores are available, or the workload involves complex control flow unsuitable for GPU execution.
- **Evaluate case-by-case** when: arithmetic intensity is in the transitional range (1–100 FLOP/byte), as the outcome depends on specific hardware, batch size, and pipeline overhead characteristics.

These guidelines are necessarily approximate and hardware-dependent; they should be validated through profiling on the target deployment platform.

Figure 4.20 presents this decision logic as a flowchart. The primary discriminant is arith-

metic intensity, followed by batch size; workloads in the transitional zone require case-specific evaluation.

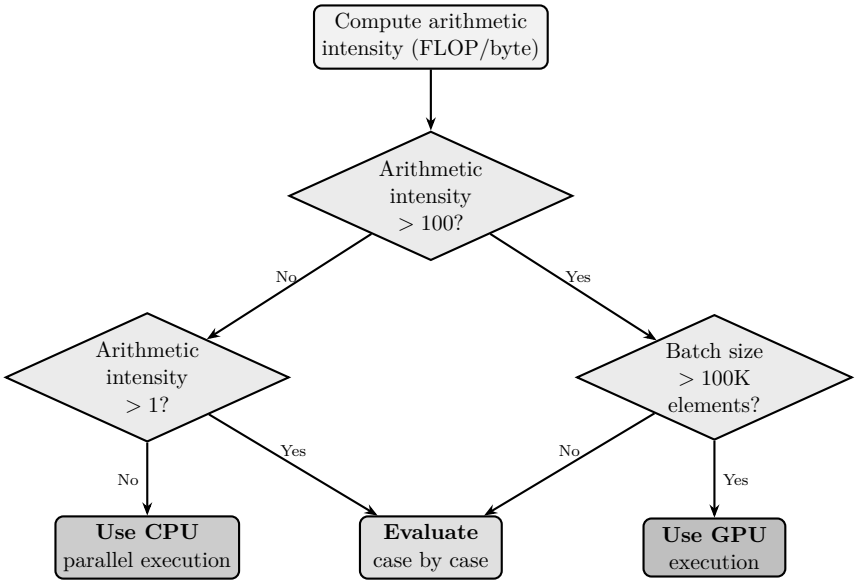

Figure 4.20: Decision framework for choosing between GPU and CPU execution in a dataflow system. The primary factor is arithmetic intensity (FLOP/byte), followed by batch size. Workloads below 1 FLOP/byte should use CPU execution; those above 100 FLOP/byte with sufficient batch sizes benefit from GPU acceleration; workloads in between require case-specific evaluation.

5 | Related Work

This chapter positions the `map_gpu` operator within the landscape of GPU-accelerated stream processing research. Rather than providing an exhaustive survey, the discussion focuses on systems that address the same fundamental challenge as this thesis: integrating GPU execution within dataflow stream processing while managing the tension between GPU batch requirements and stream processing semantics.

5.1. GPU Stream Processing: The Core Challenge

The fundamental challenge in GPU stream processing is reconciling two conflicting execution models. Stream processing systems operate on individual elements or small windows, emphasizing low latency and continuous operation. GPUs, in contrast, require large batches of data to amortize kernel launch overhead and achieve high throughput through massive parallelism.

Prior research has explored several approaches to this challenge:

- **Query-level scheduling**: Assign entire queries to CPU or GPU based on workload characteristics
- **Operator-level integration**: Enable individual operators within a pipeline to execute on GPU
- **Automatic batching**: Let the runtime determine optimal batch sizes
- **User-controlled batching**: Provide configurable strategies for different use cases

This thesis adopts operator-level integration with user-controlled batching—a combination that provides fine-grained control while preserving the dataflow programming model.

5.2. Prior Systems

This section reviews the primary research systems that have addressed GPU integration in stream processing, tracing the evolution from SABER (2016) through recent work. Each

system is described in terms of its key design choices and contrasted with the approach taken in this thesis.

5.2.1. SABER (SIGMOD 2016)

SABER [16], developed at Imperial College London, established the viability of hybrid CPU-GPU stream processing. The system executes window-based streaming SQL queries by assigning each micro-batch to either CPU or GPU based on a heterogeneous lookahead scheduler.

Key design choices (as reported in [16]):

- Query-level scheduling: entire operator graphs execute on one processor per batch
- Template-based OpenCL kernels for predefined relational operators
- Configurable batch size decoupled from logical window specifications
- Pipelined execution to overlap PCIe transfers with computation

Contrast with this thesis: SABER operates at the query level—entire queries run on CPU or GPU—whereas `map_gpu` enables operator-level placement within an otherwise CPU-executed pipeline. SABER provides only predefined relational operators; this thesis enables arbitrary user-defined kernels through the `GpuKernel` trait. SABER targets discrete GPUs with PCIe transfers; this thesis evaluates on unified memory architecture where the cost-benefit analysis differs.

5.2.2. G-Storm (IEEE TBD 2018)

G-Storm [7] integrated GPU execution into Apache Storm, a production-grade distributed stream processing framework. GPU "bolts" (Storm's operator abstraction) accept user-provided PTX kernels and execute them on batched data.

Key design choices (as described in [7]):

- Operator-level integration within Storm's topology model
- External PTX kernel files provided by users
- Automatic batching at GPU bolt boundaries
- Distributed execution across Storm cluster nodes

Contrast with this thesis: G-Storm requires users to write and compile separate PTX/CUDA code, creating a two-language development experience. The `map_gpu` op-

erator keeps kernel development in Rust through CubeCL, maintaining a single-language workflow. G-Storm's CUDA-only approach limits portability; CubeCL enables deployment across five GPU backends.

5.2.3. GASSER (IEEE Access 2019)

GASSER [11] focused specifically on GPU-accelerated sliding-window operators with automatic tuning. The system determines optimal batch sizes, thread configurations, and memory layouts through runtime profiling.

Key design choices (as described in [11]):

- Specialization for sliding-window aggregations
- Auto-tuning framework for configuration optimization
- CUDA-only implementation
- Adaptive micro-batching based on stream arrival rates

Contrast with this thesis: GASSER's auto-tuning is more sophisticated than the batching strategies in this thesis, which rely on developer-selected configurations. However, GASSER is limited to sliding-window operators, while `map_gpu` supports arbitrary element-wise transformations. GASSER's auto-tuning is opaque; this thesis provides explicit strategy choices with documented trade-offs.

5.2.4. FineStream (USENIX ATC 2020)

FineStream [32] targets integrated CPU-GPU architectures (AMD APUs) where shared memory eliminates PCIe transfer overhead. The system performs per-operator scheduling, assigning each operator to the device best suited for its characteristics.

Key design choices (as described in [32]):

- Per-operator device placement decisions
- Exploitation of shared memory on integrated architectures
- Cost model for operator-device assignment
- Predefined relational operators only

Contrast with this thesis: FineStream's per-operator philosophy aligns with `map_gpu`, and both exploit unified memory benefits. However, FineStream provides only predefined

operators and targets AMD APUs. This thesis enables user-defined kernels and evaluates on Apple M2 Pro.

5.2.5. WindFlow (JPDC 2024)

WindFlow [20] is a C++17 library providing native GPU operators (`Map_GPU`, `Filter_GPU`, `Reduce_GPU`) within a dataflow graph abstraction. Users write CUDA `__device__` functors; the library handles batching and memory management.

Key design choices (as described in [20]):

- Native GPU operators in a header-only C++ library
- Support for per-key state in partitioned-stateful GPU operators
- Multiple memory management modes (explicit, Unified Memory, pinned)
- Single-machine execution only

Contrast with this thesis: WindFlow requires CUDA code (separate from C++ host code), while `map_gpu` keeps everything in Rust. WindFlow supports per-key GPU state; this thesis focuses on stateless element-wise transformations. WindFlow is CUDA-only; CubeCL provides five-backend portability. WindFlow is single-machine; Renoir's distributed execution model extends to GPU operators.

5.2.6. Slider (DaMoN 2021)

Slider [21] addresses hardware-conscious sliding window aggregation on GPUs. Rather than a complete stream processing system, Slider provides optimized GPU kernel implementations that adapt to memory bandwidth and compute characteristics.

Key design choices (as described in [21]):

- Focus on window aggregation kernel optimization
- Adaptive algorithm selection based on hardware characteristics
- Memory bandwidth saturation strategies
- CUDA implementation targeting NVIDIA GPUs

Contrast with this thesis: Slider optimizes window aggregation kernels; this thesis provides a general operator abstraction for arbitrary element-wise computations. Slider's hardware-conscious approach could inform future optimizations of `GpuKernel` implementations, but the scope differs fundamentally.

5.3. Positioning This Thesis

Table 5.1 summarizes the key dimensions along which this thesis differs from prior work.

Table 5.1: Comparison of GPU stream processing approaches. This thesis combines operator-level integration, user-defined kernels in the host language, multi-backend portability, and distributed execution—a combination not found in prior work.

System	Integration	Kernels	Backends	Distributed	Batching
SABER	Query-level	Predefined	OpenCL	No	Fixed
G-Storm	Operator-level	PTX files	CUDA	Yes	Automatic
GASSER	Operator-level	Predefined	CUDA	No	Auto-tuned
FineStream	Operator-level	Predefined	OpenCL	No	Automatic
WindFlow	Operator-level	CUDA code	CUDA	No	Configurable
Slider	Kernel-level	N/A	CUDA	No	N/A
This thesis	Operator-level	Rust/CubeCL	5 backends	Yes	3 strategies

Note on "Distributed" column: "Distributed" indicates support for execution across multiple hosts with network communication channels as part of the runtime—not merely multi-core parallelism on a single machine. G-Storm inherits distributed execution from Apache Storm's cluster topology. Renoir natively supports distributed execution across multiple nodes; the `map_gpu` operator integrates with this model such that each worker can utilize its local GPU. The evaluation in Chapter 4 was conducted on a single node, but the operator design does not preclude multi-node deployment.

5.3.1. Unique Contributions

This thesis makes three contributions not found in prior work:

1. Host-language GPU kernels. All prior systems require either predefined operators (SABER, FineStream, GASSER) or separate GPU code in CUDA/PTX (G-Storm, WindFlow). The `GpuKernel` trait enables developers to write GPU kernels in Rust using CubeCL, maintaining a single-language development experience. This eliminates the cognitive overhead of switching between host and device languages and enables Rust's type system and tooling to apply to GPU code.

2. Multi-backend portability. Prior systems support at most two GPU backends (SABER/FineStream with OpenCL targeting both CPU and GPU). CubeCL's JIT compilation enables the same kernel code to execute on CUDA (NVIDIA), ROCm (AMD),

Vulkan, Metal (Apple), and WebGPU without source modification. This portability is particularly valuable given the increasing heterogeneity of GPU hardware.

3. Distributed GPU execution. Among systems that support user-defined GPU operators, only G-Storm provides distributed execution. Renoir's distributed execution model naturally extends to `map_gpu`: each worker utilizes its local GPU through operator-level integration, with data partitioning handled by Renoir's existing mechanisms. This enables GPU-accelerated stream processing to scale across cluster nodes.

5.3.2. Limitations

Accurate positioning requires acknowledging what this thesis does *not* provide:

- **No windowed operators**: Unlike SABER, GASSER, and Slider, this thesis implements only stateless element-wise transformations. Window-based aggregations remain on CPU.
- **No auto-tuning**: Unlike GASSER, batch size selection requires developer choice among the three provided strategies. The evaluation in Chapter 4 provides guidance, but no automatic optimization occurs.
- **No per-key GPU state**: Unlike WindFlow, the `map_gpu` operator processes each batch independently without maintaining state in GPU memory between invocations.
- **No cross-worker GPU coordination**: Each Renoir worker manages its own GPU context independently. There is no global GPU scheduling or work stealing across nodes.

These limitations define clear directions for future work while not diminishing the contributions of operator design, multi-backend portability, and the arithmetic intensity analysis methodology demonstrated in the evaluation.

5.4. Summary

Research on GPU-accelerated stream processing has progressed from query-level scheduling (SABER, 2016) through operator-level integration (G-Storm, FineStream, WindFlow) to hardware-conscious optimization (Slider). This thesis extends this trajectory by combining:

1. Operator-level integration that preserves Renoir's dataflow semantics

2. User-defined GPU kernels written in the host language (Rust)
3. Portable execution across five GPU backends via CubeCL
4. Distributed execution within Renoir's existing cluster model
5. Explicit batching strategies with documented trade-offs

The evaluation methodology—characterizing GPU viability through arithmetic intensity and roofline analysis—provides a principled framework for understanding when GPU acceleration is beneficial, complementing the system-building contributions of prior work.

6 | Conclusions and Future Work

This chapter concludes the thesis by summarizing its contributions, reflecting on the research question, discussing limitations, and outlining directions for future work.

6.1. Summary of Contributions

This thesis investigated the integration of GPU execution within a dataflow stream processing framework. The central research question asked:

> *To what extent can GPU programming be embedded within dataflow execution models, while preserving abstraction, composability, and execution semantics of stream processing?*

To address this question, the thesis extended the Renoir dataflow platform with GPU-accelerated operators, evaluated their performance on representative workloads, and characterized the conditions under which GPU acceleration is beneficial. The main contributions are summarized below.

A GPU-accelerated dataflow operator. The `map_gpu` operator integrates GPU execution into Renoir's operator model without modifying the global runtime or scheduler. GPU-specific concerns—batching, data layout conversion, kernel dispatch, and synchronization—are encapsulated within the operator, preserving the composability and semantics of the dataflow pipeline. From the perspective of upstream and downstream operators, `map_gpu` behaves as a deterministic transformation, despite internally operating on batched, asynchronously executed data.

A kernel abstraction via the `GpuKernel` trait. The `GpuKernel` trait provides a clear separation of concerns between the framework and user code. Developers implement this trait to define GPU compute kernels, manage internal data buffers, and control memory layout. This design preserves expressiveness while avoiding the complexity of automatic kernel generation. The framework handles stream-level coordination—timestamps, batching decisions, and result ordering—while the kernel implementation handles device-level

execution.

Configurable batching strategies. The thesis introduced three batching strategies—Fixed, Timed, and Adaptive—that allow developers to navigate the latency–throughput trade-off inherent in GPU execution. Experimental evaluation demonstrated that batch sizes near one million elements provide optimal throughput for the workloads tested, while adaptive batching offers comparable performance with automatic tuning.

Portable GPU execution via CubeCL. By leveraging CubeCL, the implementation achieves backend portability across multiple GPU platforms. Kernels written in Rust using the `#[cube]` macro can target CUDA, ROCm, Vulkan, Metal, and WebGPU through compile-time feature selection, without requiring changes to kernel source code. This portability distinguishes the approach from prior GPU stream processing systems that are typically locked to a single backend (e.g., OpenCL in SABER, CUDA in WindFlow).

Experimental characterization of GPU acceleration viability. The evaluation on three benchmarks—reduction, Black–Scholes option pricing, and Monte Carlo simulation—revealed that *arithmetic intensity* is the dominant factor determining whether GPU acceleration is beneficial. Workloads with low arithmetic intensity (below 1 FLOP/byte) do not benefit from GPU execution; those in a transitional range (1–100 FLOP/byte) may outperform sequential CPU but not parallel CPU execution; and compute-bound workloads (above 100 FLOP/byte) achieve substantial speedups over both. For the Monte Carlo benchmark, the GPU achieved speedups of up to 555× over sequential CPU and 55.6× over 12-core parallel CPU execution.

6.2. Answering the Research Question

The research question asked whether GPU programming can be embedded within dataflow execution models while preserving abstraction, composability, and execution semantics. Based on the design, implementation, and evaluation presented in this thesis, the answer is *conditionally affirmative*: GPU execution can be successfully integrated into a dataflow framework, but the extent of its benefit depends critically on workload characteristics.

Abstraction is preserved. The `map_gpu` operator encapsulates GPU execution behind a familiar API. Developers interact with the operator using the same pipeline composition patterns as standard Renoir operators. The `GpuKernel` trait provides a structured abstraction for defining GPU compute kernels without exposing low-level device manage-

ment details such as buffer allocation, command queue submission, or synchronization primitives.

Composability is maintained. GPU operators compose with other Renoir operators in the expected manner. A pipeline can include multiple `map_gpu` operators, interleave GPU and CPU operators, and connect to standard sources and sinks. The evaluation confirmed that all 116 validation tests passed, demonstrating that GPU execution does not violate the correctness properties expected of dataflow operators.

Execution semantics are preserved. The `map_gpu` operator maintains stream semantics by preserving element ordering, associating results with original timestamps, and correctly propagating watermarks. Batching is an internal optimization that does not leak into the logical stream representation observed by downstream operators.

Performance benefits are workload-dependent. The thesis demonstrated that GPU acceleration is not universally beneficial. The arithmetic intensity of the workload—the ratio of floating-point operations to memory access—determines whether the GPU's computational throughput advantage can overcome pipeline overhead. For memory-bound workloads, CPU execution remains superior. For compute-bound workloads, GPU acceleration delivers order-of-magnitude improvements.

6.3. Limitations

The approach presented in this thesis has several limitations that should be acknowledged.

Single-machine evaluation. All experiments were conducted on a single Apple M2 Pro system with unified memory architecture. This platform eliminates PCIe transfer overhead, representing a favorable scenario for GPU integration. Performance characteristics on systems with discrete GPUs connected via PCIe may differ significantly, as explicit data transfers would add overhead that shifts the crossover point for beneficial GPU execution toward more compute-intensive workloads.

Absence of NVIDIA/CUDA evaluation. The CUDA backend of CubeCL could not be tested due to hardware availability constraints. While the design is backend-agnostic, CUDA remains the dominant GPU compute platform in practice, and validation on NVIDIA hardware would strengthen the generalizability of the results.

Limited operator coverage. The implementation provides `map_gpu` and `reduce_gpu` operators. Other operator types—such as GPU-accelerated windowed aggregations, joins, or stateful operators—are not addressed. Extending GPU support to a broader operator vocabulary would require additional design work, particularly for operators with more complex state management requirements.

Manual kernel implementation. Developers must implement the `GpuKernel` trait manually, including data layout decisions and buffer management. The framework does not automatically generate GPU kernels from high-level operator descriptions. While this design choice preserves expressiveness and avoids abstraction leakage, it places a greater burden on developers compared to systems that offer automatic GPU code generation.

No distributed GPU execution. Renoir supports distributed execution across multiple machines, but the GPU operators implemented in this thesis operate within a single worker. Extending GPU execution to distributed settings—for example, coordinating GPU resources across cluster nodes—is beyond the scope of this work.

Batching requires tuning. While the Adaptive batching strategy offers automatic adjustment, optimal batch sizes remain workload-dependent. The evaluation showed that Fixed batching with carefully chosen batch sizes outperforms Adaptive batching in some cases. Developers must still reason about batch size trade-offs, particularly for latency-sensitive applications.

6.4. Future Work

Several directions for future work emerge from this thesis.

Evaluation on discrete GPU systems. Extending the evaluation to systems with discrete NVIDIA or AMD GPUs connected via PCIe would provide insights into how data transfer overhead affects the viability of GPU acceleration. The double-buffering mechanism implemented in this thesis is specifically designed to overlap PCIe transfers with kernel execution; validating its effectiveness on such hardware is a natural next step.

CUDA backend validation. Testing the implementation with CubeCL's CUDA backend on NVIDIA hardware would validate the portability claims and enable comparison with prior GPU stream processing systems such as SABER and WindFlow. This evalua-

tion could also assess performance differences between single-precision (`f32`) and double-precision (`f64`) execution, as the CUDA backend supports both.

Adaptive batch size tuning. The current Adaptive batching strategy adjusts batch size based on stream arrival rate. More sophisticated approaches could incorporate feedback from kernel execution time, memory utilization, or downstream backpressure to dynamically optimize the latency–throughput trade-off. Techniques from the reinforcement learning literature, such as those explored in recent work on self-adaptive micro-batching [17], could be applied.

Extended operator support. Implementing GPU-accelerated variants of additional operators—windowed aggregations, temporal joins, or user-defined stateful operators—would broaden the applicability of the approach. Each operator type presents unique challenges: windowed operators must coordinate batching with window boundaries; joins require efficient GPU-side data structures; stateful operators must manage persistent GPU memory.

Automatic kernel generation. Exploring compiler-based or code-generation approaches to automatically synthesize `GpuKernel` implementations from high-level operator descriptions could reduce developer burden. Such an approach would need to infer appropriate data layouts, vectorization factors, and batching strategies, potentially guided by profiling or autotuning.

Integration with distributed GPU scheduling. In multi-GPU or cluster settings, coordinating GPU execution across workers introduces additional scheduling and data placement challenges. Future work could explore how Renoir's distributed runtime could be extended to support heterogeneous resource management, including GPU memory capacity constraints and inter-GPU communication.

Energy efficiency analysis. GPU execution offers not only throughput improvements but potentially also energy efficiency gains for suitable workloads. Characterizing the energy consumption of GPU-accelerated dataflow processing, compared to CPU-only execution, would provide additional decision criteria for workload placement.

6.5. Concluding Remarks

This thesis demonstrated that GPU execution can be meaningfully integrated into a dataflow stream processing framework. The `map_gpu` operator provides a practical mechanism for accelerating compute-intensive workloads while preserving the abstractions that make dataflow programming productive. The evaluation revealed that arithmetic intensity is the key determinant of GPU acceleration viability, with compute-bound workloads achieving speedups of nearly three orders of magnitude.

At the same time, the results underscore that GPU acceleration is not a universal solution. Memory-bound workloads, small batch sizes, and high pipeline overhead can negate the benefits of GPU execution. Developers must assess workload characteristics and evaluate performance on their target hardware before adopting GPU operators.

The contributions of this thesis—the operator design, kernel abstraction, batching strategies, and experimental characterization—provide a foundation for further exploration of heterogeneous execution in dataflow systems. As computational workloads continue to grow in complexity and data volumes, and as hardware platforms become increasingly heterogeneous, the ability to integrate specialized accelerators into high-level programming models will remain an important challenge. This thesis offers one step toward addressing that challenge within the context of stream processing.

Bibliography

[1] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015. doi: 10.14778/2824032.2824076.

[2] Apple Inc. Apple M2 Pro chip. `https://www.apple.com/newsroom/2023/01/apple-introduces-m2-pro-and-m2-max/`, 2023. Accessed: 2025-01.

[3] Apple Inc. Metal programming guide. `https://developer.apple.com/metal/`, 2024. Accessed: 2025-01.

[4] F. Black and M. Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–654, 1973. doi: 10.1086/260062.

[5] G. E. P. Box and M. E. Muller. A note on the generation of random normal deviates. *The Annals of Mathematical Statistics*, 29(2):610–611, 1958. doi: 10.1214/aoms/1177706645.

[6] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 38(4):28–38, 2015.

[7] Z. Chen, J. Xu, J. Tang, K. A. Kwiat, C. A. Kamhoua, and C. Wang. G-Storm: GPU-enabled high-throughput online stream data processing. *IEEE Transactions on Big Data*, 4(2):191–202, 2018. doi: 10.1109/TBDATA.2016.2616116.

[8] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 613–627, 2017. URL `https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw`.

[9] G. Cugola and A. Margara. Processing flows of information: From data stream to

complex event processing. *ACM Computing Surveys*, 44(3):1–62, 2012. doi: 10.1145/2187671.2187677.

[10] L. De Martini, A. Margara, and G. Cugola. The renoir dataflow platform: Efficient data processing without complexity. *Future Generation Computer Systems*, 160:1–14, 2024. doi: 10.1016/j.future.2024.06.018.

[11] T. De Matteis, G. Mencagli, D. De Sensi, M. Torquati, and M. Danelutto. GASSER: An auto-tunable system for general sliding-window streaming operators on GPUs. *IEEE Access*, 7:48753–48769, 2019. doi: 10.1109/ACCESS.2019.2910312.

[12] gfx-rs contributors. wgpu: Safe and portable gpu abstraction in rust. GitHub repository, 2025. URL `https://github.com/gfx-rs/wgpu`. Accessed: 2025-01.

[13] P. Glasserman. *Monte Carlo Methods in Financial Engineering*, volume 53 of *Stochastic Modelling and Applied Probability*. Springer, 2003. doi: 10.1007/978-0-387-21617-1.

[14] C. Gregg and K. Hazelwood. Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 134–145, 2011. doi: 10.1109/ISPASS.2011.5762730.

[15] J. L. Hennessy and D. A. Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, 2019. doi: 10.1145/3282307.

[16] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, pages 555–569, 2016. doi: 10.1145/2882903.2882906.

[17] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier. Out-of-order processing: A new architecture for high-performance stream systems. *Proceedings of the VLDB Endowment*, 1(1):274–288, 2008. doi: 10.14778/1453856.1453890.

[18] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008. doi: 10.1109/MM.2008.31.

[19] G. Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 2003. doi: 10.18637/jss.v008.i14.

[20] G. Mencagli, M. Torquati, A. Cardaci, A. Fais, L. Rinaldi, and M. Danelutto.

General-purpose data stream processing on heterogeneous architectures with WindFlow. *Journal of Parallel and Distributed Computing*, 184:104782, 2024. doi: 10.1016/j.jpdc.2023.104782.

[21] G. Michas, P. Chrysogelos, I. Mytilinis, and A. Ailamaki. Hardware-conscious sliding window aggregation on GPUs. In *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN '21)*, pages 1–5. ACM, 2021. doi: 10.1145/3465998.3466014.

[22] NVIDIA Corporation. *CUDA C++ Programming Guide*, 2024. URL `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`. Accessed: 2025-01.

[23] NVIDIA Corporation. Cuda toolkit documentation, 2025. URL `https://docs.nvidia.com/cuda/`. Accessed: 2025-01.

[24] E. Panova, V. Volokitin, A. Gorshkov, and I. Meyerov. Black-scholes option pricing on intel cpus and gpus: Implementation on sycl and optimization techniques. In *Supercomputing. RuSCDays 2022*, volume 13708 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 2022. doi: 10.1007/978-3-031-22941-1_18.

[25] J. Pennycook, S. D. Hammond, S. A. Wright, A. Herdman, I. Miller, and S. A. Jarvis. An investigation of the performance portability of opencl. *Journal of Parallel and Distributed Computing*, 73(11):1439–1450, 2013. doi: 10.1016/j.jpdc.2012.07.005.

[26] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 73–82, 2008. doi: 10.1145/1345206.1345220.

[27] Tracel AI. Cubecl: Multi-platform high-performance compute language extension for rust. GitHub repository, 2024. URL `https://github.com/tracel-ai/cubecl`. Accessed: 2025-01.

[28] Tracel AI. Cubecl crate documentation: Feature flags, 2025. URL `https://docs.rs/cubecl/latest/cubecl/#feature-flags`. Accessed: 2025-01.

[29] V. Volkov. Better performance at lower occupancy. In *Proceedings of the GPU Technology Conference (GTC)*, 2010. URL `https://www.nvidia.com/content/gtc-2010/pdfs/2238_gtc2010.pdf`. Accessed: 2025-01.

[30] W3C. WebGPU specification, 2024. URL `https://www.w3.org/TR/webgpu/`. Accessed: 2025-01.

[31] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009. doi: 10.1145/1498765.1498785.

[32] F. Zhang, L. Yang, S. Zhang, B. He, W. Lu, and X. Du. FineStream: Fine-grained window-based stream processing on CPU-GPU integrated architectures. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*, pages 633–647. USENIX Association, 2020. URL `https://www.usenix.org/conference/atc20/presentation/zhang-feng`.