

UNIVERSITY OF CALIFORNIA  
SANTA CRUZ

**SCANBOX: A TUNABLE AND PORTABLE GPU PREFIX SCAN  
IMPLEMENTATION IN VULKAN AND WEBGPU.**

A thesis submitted in partial satisfaction of the  
requirements for the degree of

Bachelor of Science

in

COMPUTER SCIENCE

by

**James V. Contini**

June 2025

Copyright © by

James V. Contini

2025

# Table of Contents

<b>List of Figures</b>	<b>v</b>
<b>List of Tables</b>	<b>vii</b>
<b>Abstract</b>	<b>viii</b>
<b>Dedication</b>	<b>x</b>
<b>Acknowledgments</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Contributions . . . . .	3
<b>2 Background</b>	<b>5</b>
2.1 Basic GPU Architecture . . . . .	5
2.2 GPU Programming Model . . . . .	7
2.3 GPU Programming Frameworks . . . . .	9
2.4 Prefix-scan . . . . .	10
<b>3 Methodology</b>	<b>15</b>
3.1 Chained Scan: A Device-wide GPU Scan Strategy . . . . .	15
3.2 Chained Scan Variation: Sequential . . . . .	17
3.3 Chained Scan Variation: Decoupled Lookback . . . . .	19
3.4 Chained Scan Variation: Parallelized Decoupled Lookback . . . . .	20
3.5 Intra-block Scan: Approach and Implementations . . . . .	22
3.6 More Tuning Parameters . . . . .	24
3.7 Why are we tuning? . . . . .	24
3.8 Tuning Approach: Execution Details . . . . .	25
<b>4 Implementation</b>	<b>27</b>
4.1 Toolchain and Frameworks . . . . .	27
4.2 Vulkan: Initial Issues . . . . .	28

4.3	Vulkan: Framework and Toolchain Bugs . . . . .	30
4.4	Vulkan: Formal Verification . . . . .	35
4.5	WebGPU: Initial Issues . . . . .	36
4.6	Vulkan and WebGPU: Implementation Adaptations . . . . .	37
<b>5</b>	<b>Results and Discussion</b>	<b>44</b>
5.1	Roadmap for Results and Discussion . . . . .	44
5.2	ScanBox: Initial Results . . . . .	44
5.3	ScanBox: Fully Tuned Performance Results . . . . .	49
5.4	Performance Variability: The Need for Input-Specific Tuning . . . . .	52
5.5	Leveraging Per-Input Tuning in Resource-Limited Settings . . . . .	55
5.6	ICS for Multiple Devices . . . . .	66
5.7	Final Problem: Generate the Best Kernel for Multiple Devices . . . . .	75
<b>6</b>	<b>Future Work</b>	<b>79</b>
<b>7</b>	<b>Conclusion</b>	<b>81</b>
	<b>Bibliography</b>	<b>82</b>

# List of Figures

2.1	SIMD architecture. . . . .	6
2.2	GPU Thread Hierarchy. . . . .	7
2.3	Serial Scan. . . . .	12
2.4	Hillis and Steele scan algorithm. . . . .	13
2.5	Upsweep & downsweep phases of the Blelloch 1990 scan algorithm. . . . .	14
3.1	Pseudo-code for Chained Scan algorithm. . . . .	16
3.2	Sequential Chained Scan visualization. . . . .	18
3.3	OpenCL-like Sequential Decoupled Lookback pseudo-code. . . . .	21
3.4	Visualization of Parallelized Decoupled Lookback. . . . .	22
4.1	Minimized code example of the Nvidia Vulkan bug. . . . .	31
4.2	Minimized code example of the AMD Vulkan bug. . . . .	32
4.3	Efficient, single value approach. . . . .	38
4.4	Undefined and disallowed subgroup behaviour. . . . .	42
4.5	SIMD Raking is technically undefined in WGSL. . . . .	43

5.1	Initial Nvidia RTX 4070 prefix-sum plot (throughput in GB/s). . . . .	45
5.2	Nvidia RTX 4070 prefix-sum parametrized on Parallelized Decoupled Look- back vs. Sequential Decoupled Lookback (throughput in GB/s). . . . .	47
5.3	AMD XT 7900 varying input data types (throughput in GB/s). . . . .	48
5.5	Nvidia RTX 4070 Vulkan vs. CUB. . . . .	51
5.6	Subset of AMD XT 7900's throughput density distributions plots. . . . .	54
5.7	AMD XT 7900 best overall parameters vs. best single kernel that maximizes the sum of the throughputs at every input size — Throughput in GB/s. . . . .	58
5.8	Nvidia RTX 4070 best overall parameters vs. best single kernel that maximizes the sum of the throughputs (GB/s) at every input size. . . . .	59
5.12	Benchmarked on the Nvidia RTX 4070: The best RTX 4070 parameters vs. The best XT 7900 parameters — Throughput in GB/s. . . . .	66
5.13	Plots the shared ICS for both devices (dotted line), using averaging, against de- vice local ICSs on the AMD XT 7900 and the Nvidia RTX 4070 — Throughput in GB/s . . . . .	67
5.14	Plots the shared ICS for both devices (dotted line), using max, against device local ICSs on the AMD XT 7900 and the Nvidia RTX 4070 . . . . .	68
5.15	Same plot using averaging with variance penalty — Throughput in GB/s . . . .	69
5.19	Minimizing Distance to the 1.25 power — Throughput in GB/s. . . . .	75
5.20	Best kernel across AMD and Nvidia — Throughput in GB/s. . . . .	77

# List of Tables

3.1	Summary of tunable parameters in our prefix-scan framework. . . . .	25
4.1	Summary of device functionality at the beginning of development. . . . .	29
4.2	Summary of device functionality after adding the require full subgroups flag. . .	29
4.3	Summary of device functionality after all aforementioned changes. . . . .	34
5.1	Percentage of memory bandwidth achieved compared to the device ceiling. . .	52

## **Abstract**

Scanbox: A tunable and portable GPU prefix scan implementation in Vulkan and WebGPU.

by

James V. Contini

Prefix-scan is an important algorithmic primitive, it is useful for Radix Sort, Stream Compaction, linear recurrence, quicksort, sparse matrix-vector multiply, tridiagonal matrix solvers, fluid simulation, and more. Since 2016, state-of-the-art GPU implementations of this primitive are so efficient that they are memory-bound; that is, performance is limited by the rate at which data can be transferred to and from the GPU. Extensive research on high performance prefix-scan exists, but it has primarily focused on CUDA implementations. In 2024, 91% of AI papers used Nvidia’s CUDA framework and GPUs, a striking figure given that major competitors such as Apple, AMD, Intel, Qualcomm, Arm, and Google continue to make significant advancements in GPUs. To that end, we have built two portable prefix-scan implementations: one in WGSL and another in OpenCL (for Vulkan backends). This work investigates these implementations outlining issues and solutions with respect to performance and portability across six GPUs from four vendors. We also introduce six key prefix-scan implementation decisions to tune our kernels over. Tuning over these parameters, we have generated scan kernels on AMD and Nvidia that at small sizes outperform and at large sizes come within 1% of vendor optimized implementations like Nvidia’s CUB come within 93% of this ceiling. We investigate implementation quirks across Intel, Apple, Nvidia, and AMD devices but mainly characterize performance on



AMD and Nvidia discrete cards. Through our testing, we find per-input-size tuning not only maximizes performance, but also provides data that's useful for a range of deployment constraints. For example, it allows us to identify best-on-average kernels for space constrained platforms. In this way, tuning supports both peak optimization and practical deployment.

To the UCSC faculty,

and my parents,

and who ever else wants to read it.

## Acknowledgments

I'd like to thank Reese L, Tyler S, Devon M, Rihtik S, Liting H, Yuanchao X, my mom, dad, and brother, Grandma Lucy, Naomi R, Touss M, and Erik C.

I want to thank Reese Levine for introducing me to this project and helping me through it every step of the way. A year and three months ago, he posted a Google Doc asking for undergrads to help with research. Randomly, I chose Prefix-scan not knowing what I was getting into. For the first 30 days, I banged my head on my keyboard until my trackpad broke. But Reese is like the tree of wisdom, not just because he's tall. But because it always felt like, no matter how stuck I was, even with problems that couldn't be fixed (such as framework bugs), they were still manageable. From day one, despite knowing nothing about GPUs, my questions have always felt heard, and I have never once felt like I was being talked down to. Reese has been my main consultant for this thesis and for all implementation things leading up to it. He helped me with multiple rounds of feedback which for a 100 page paper is a lot to ask for anybody. He has been a constant source of support in this work and in my life over the last year and a half. Thank you Reese.

In the fall of 2023, I took Computer Systems with Professor Quinn, where I was introduced to multithreading, which led me to reach out to both Quinn and Tyler. Despite never having met me, Tyler responded warmly to my cold email and took a chance on me. From the beginning, he was incredibly personable and supportive. As I've gotten to know him and understand the lab more, I've been genuinely boggled by how he juggles so many projects,

people, meetings, and moving pieces, and yet still makes time for individual meetings and somehow has the bandwidth. Tyler has helped convince me that I am “cut out”. I certainly would not be writing this if not for relentless thoughtfulness. In addition to emotional support, Tyler has a firm grasp of the field of empirical GPU testing and has been instrumental in guiding the narrative of this thesis. I can’t stress enough how necessary he’s been in helping me start, persevere, and finish this work. Thank you Tyler.

Devon M: You always fix and update the birds so quickly and have given me pointers about them, thank you for making the machines feel less daunting.

Rihit S: Thank you for helping me consolidate my tuning work and giving me pointers on benchmarking, your suggestions have been extremely useful.

Professor Liting & Professor Yuanchao: For co-advising me alongside Tyler. Thank you both for the time and support.

Hernan Ponce De Leon: Thank you for the intriguing conversations and validating my Vulkan implementation.

Jeff Bolz: Thank you for fixing the subgroupBarrier bug. This was absolutely necessary for getting prefix-sum on Vulkan to work.

My mom, dad, and brother: Thank you for always being there for me, accommodating and me in all my endeavors, and being constant supports of love, unwavering support, normalcy and comfortability in my life.

Grandma Lucy: Thank you for being a sounding board and a proof reader simultaneously.

Naomi R: Thank you for sharing your thesis and presentation materials with me, and for helping me navigate the experience of being an undergrad in research.

Touss M: Thank you for convincing me to join a research lab in the first place, and for being, like Tyler, the sound I need to hear when I doubt myself.

Erik C: Thank you for being my #1 housemate for two years. For cooking and eating together. Letting me rubber duck you. Being the one person I know I could see even if I had a midterm due at 10pm and a flight to Virginia at 5am. There will be so many things I miss about our living arrangement.

# Chapter 1

## Introduction

As data volumes grow, GPUs have become increasingly important for computing tasks that benefit from parallelism. Their ability to efficiently process large datasets makes them well-suited for many modern applications such as blockchain, machine learning, and scientific computing [51, 18, 48, 33, 15, 46, 34, 56, 41]. One fundamental operation that enables many parallel algorithms on GPUs is prefix-scan, which computes the running total (or other associative operations) of a sequence of values. Applications of GPU prefix-scan include radix sort, stream compaction, adder design, linear recurrence and tridiagonal solvers, parallel allocation and queuing, DNA Sequencing, quicksort, sparse matrix-vector multiply, fluid simulation, and more [36, 5, 49, 50, 37, 16, 42, 64].

The earliest GPU prefix scan we found was written by Daniel Horn at NVIDIA in 2005 [17]. Since 2012, state-of-the-art GPU implementations of this primitive have become so efficient that they are now memory-bound; that is, performance is limited by the rate at which data can be transferred to and from the GPU [49, 17, 32, 30]. In 2013, another breakthrough

reduced the total number of global memory accesses from  $3N$  to  $2N$ , where  $N$  is the problem size [63]. Most recently, in 2016, researchers from Nvidia introduced an improved global reduction method called Decoupled Lookback, which further reduced latency in global prefix propagation. On Nvidia devices, this strategy was shown to be so efficient that it matches the speed of a memory copy operation at medium to large problem sizes [30].

Incredibly, all four of these breakthrough papers were conducted on Nvidia GPUs while three out of three of these were solely conducted on Nvidia hardware. In 2024, 91% of AI papers were conducted on NVIDIA's GPUs, a striking figure given that major competitors such as Apple, AMD, Intel, Qualcomm, Arm, and Google have recently made significant advancements in GPU performance [58, 3, 61, 27]. Nvidia's early investment in GPGPU (General Purpose GPU) and ecosystem of robust tools, libraries and hardware has resulted in their widespread adoption across both industry and academia. However, CUDA is closed source, and this heavy focus on a single vendor has hindered the broader GPU ecosystem. As a result, alternative platforms remain underdeveloped and under-optimized.

GPGPU programming is a relatively new paradigm, with the first major framework, CUDA, introduced in 2006. Since then, hardware and software ecosystems have evolved in divergent ways across vendors. Differences in compute unit counts, memory hierarchies, thread execution models, and compiler behavior make it challenging to write performance-portable code. These variations complicate tuning strategies, particularly for parallel algorithms like prefix-scan, where optimal parameters can vary significantly depending on the architecture. [28, 24, 57, 45, 65, 11, 59].

## 1.1 Contributions

In this work, we contribute to the cutting edge of GPGPU research by implementing two high-performance, portable prefix-sum kernels: one targeting SPIR-V, and another in WGSL, extending support to Apple devices. Not only do these implementations achieve near peak performance across multiple vendors, but because prefix-scan exercises many complex and often under-tested features of GPU programming models, our work also surfaces several previously unknown bugs in framework implementations. Moreover, we find that code developed within mature frameworks often requires substantial manual adaptation to be ported to newer frameworks, typically at the cost of reduced performance. In particular, WebGPU being the youngest of the three has extremely limited guarantees and support compared to CUDA and Vulkan. To illustrate our point, WebGPU began supporting subgroups 8 months ago, Vulkan 8 years ago, and CUDA 18 years ago [35, 14]. Implementing the same prefix-scan algorithm in WebGPU required numerous adjustments compared to Vulkan, largely due to differences in language expressiveness and feature maturity. Even with similar high-level logic, performance between the two frameworks varies considerably. Nonetheless, consistent with prior work, we find that per-device and per-input-size tuning remains highly effective for extracting near-peak performance across backends, devices and inputs. [65, 11, 59]. To this end, we identified just six key implementation decisions to parameterize our kernels. Despite the small number of tuning parameters, this design achieves strong performance across vendors: it outperforms Nvidia’s vendor-optimized library at small input sizes and reaches up to 92% of global memory bandwidth on both AMD and Nvidia GPUs at large sizes. For reference, Nvidia’s proprietary



implementation achieves 93% of global memory bandwidth. In summary, this thesis presents *ScanBox*, a versatile GPU kernel tuning framework for prefix-scan, along with a detailed investigation of implementation portability across six GPUs and an in-depth characterization of performance portability. We demonstrate its effectiveness in several real-world scenarios:

1. **Single-GPU deployments:** Per-input-size tuning identifies the best configuration for each input size, delivering optimal performance across all inputs on a given device.
2. **Single-GPU, single-kernel deployments:** Useful in environments constrained by storage or setup complexity. This approach finds the best single kernel across all input sizes, requiring performance trade-offs but still achieving near-optimal performance on the target device.
3. **Multi-GPU, single-kernel deployments:** Constrained by both storage and cross-platform compatibility. This approach makes trade-offs across devices and input sizes to provide robust, portable performance across a variety of hardware platforms.

# Chapter 2

## Background

In this section, we first review the fundamentals of GPU architecture, highlighting the principles that make GPUs well-suited for parallel computation. We then examine the major programming models and frameworks that support GPGPU development, focusing on Vulkan and WebGPU. With this foundation, we turn to the parallel prefix-scan problem, presenting various algorithms and strategies that have been proposed for efficient implementation on GPUs.

### 2.1 Basic GPU Architecture

This section will go over the architecture of the GPU. Compared to a CPU, GPUs have thousands of times more cores [44]. To use this abundance of cores effectively, GPUs leverage SIMD (single instruction multiple data) style execution [12]. A SIMD unit executes a single instruction on multiple pieces of data in parallel. The underlying implementation of a SIMD unit varies from vendor to vendor. AMD’s SIMD unit consists of sixteen SIMD lanes (each with their own scalar ALU and register) that execute in lock step. AMD abstracts their plurality

away in documentation and calls them a VALU (vector ALU). ALUs in the same SIMD unit always execute the same instruction in parallel so it is helpful to imagine the ALUs as a single unit as AMD suggests [20, 38, 1, 2, 41].

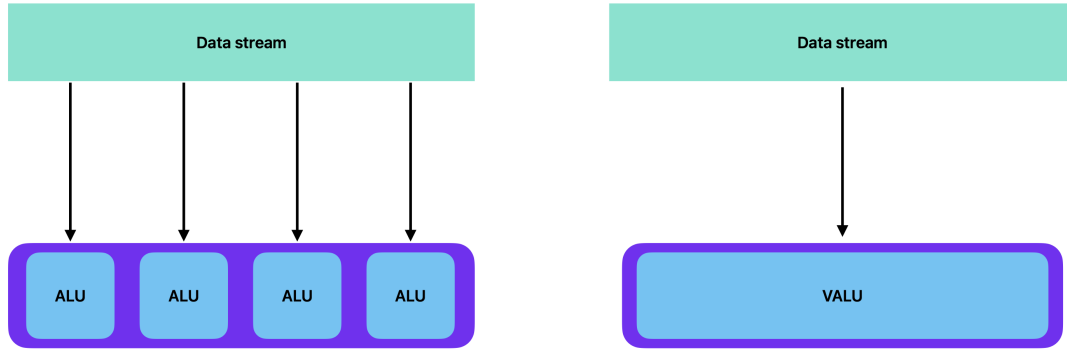


Figure 2.1: SIMD architecture.

Multiple of these SIMD units are grouped into a compute unit (CU). Modern GPUs employ SIMT (single instruction multiple thread). In this paradigm, when an instruction needs to be executed a group of threads (often 16, 32, or 64) called a subgroup execute it all together, in lock step. The subgroup executes this instruction on a CU using one or possibly multiple SIMD units at the same time. SIMT allows for greater flexibility when threads within a subgroup need to diverge from uniform control flow, that is, execute different instructions. Nvidia describes that their architecture deactivates threads that don't need to execute the divergent instruction. AMD explains non-participatory threads of the divergent instruction are masked out during execution. Divergent instruction execution incurs a significant performance penalty. GPUs spend a significant amount of die on ALUs so they lack space for complex control units. This design choice makes them very suited for highly parallel homogenous workloads [60]. Most archi-

textures organize several subgroups into a workgroup. Workgroups are then assigned to a CU. Multiple workgroups may share a single CU. Threads within a workgroup can communicate using special CU memory called shared memory. The CUDA Programming guide says shared memory is equivalent to a cache in memory speed [39]. The mapping between hardware and thread is abstracted but subgroups are functionally equivalent to SIMD execution style from the perspective of the developer [38, 20].

## 2.2 GPU Programming Model

In the GPU programming model, threads and memory are organized hierarchically by intra-group communication speed and memory size [38, 1, 20].

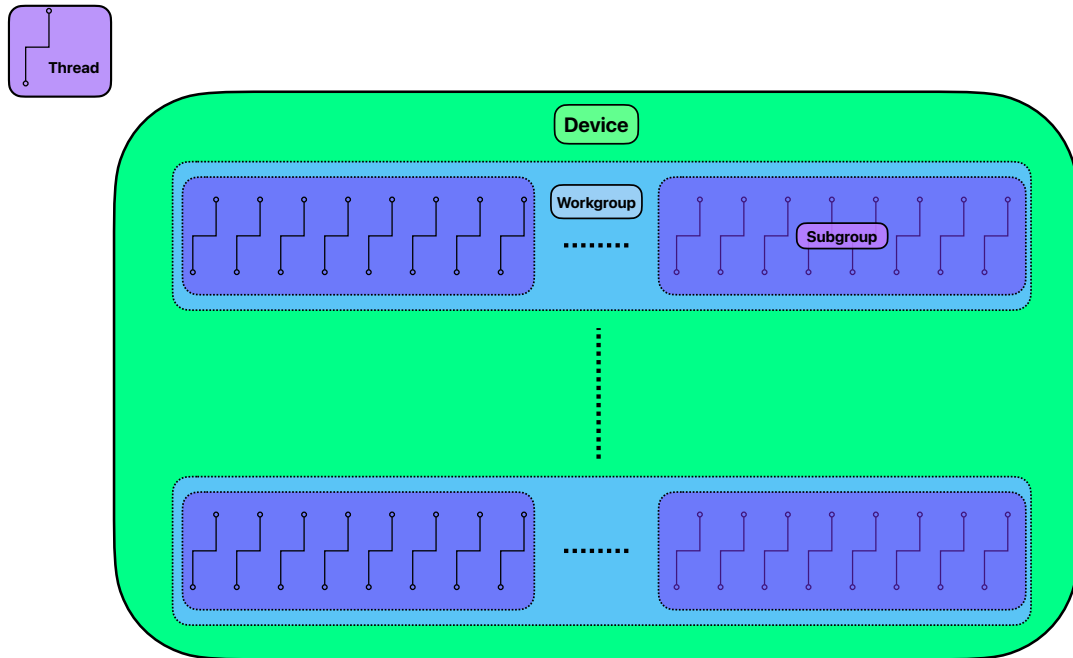


Figure 2.2: GPU Thread Hierarchy.

**Subgroups** Subgroup memory is small and not transparent to the programmer. It is advantageous for threads in the same subgroup to execute the same instructions and avoid divergent control flow so that the subgroup can execute in lock step avoiding non-utilized threads. Threads in the same subgroup have access to special subgroup-level functions. These are SIMD-like functions that enable low-overhead data sharing. They support efficient data-parallel operations and allow for broadcasting values across a subgroup, making them especially useful in high-performance applications.

**Workgroups** Multiple subgroups are organized into a workgroup. Threads in a workgroup can communicate using a local memory region called shared memory. Shared memory is bigger and slower than the subgroup communication functions and operations but unlike subgroup memory it is addressable to the programmer. Threads in the same workgroup communicate through this shared memory region. Unlike subgroups workgroups do not have functions to communicate or compute SIMD like operations. Executing computations or broadcasting data using this shared data requires explicit synchronization to maintain coherence and atomicity between threads of different subgroups.

**Global Memory** Threads in different workgroups can communicate using global memory. Global memory is slow compared to the first two memory spaces.

**GPU Programming Philosophy** Effective use of the GPU involves organizing work to limit communication to the most local levels of the execution hierarchy. Specifically, communication should occur at the narrowest possible level, starting within subgroups, then within work-

groups, and only resorting to global memory when necessary. Subgroup-level execution, in particular, offers highly efficient communication and synchronization. However, until recently, many frameworks did not expose subgroup operations, which placed non-CUDA platforms at a disadvantage. As subgroup functionality becomes more widely supported, programmers are increasingly able to write code that explicitly leverages subgroup execution for the bulk of the computation, particularly in homogeneous workloads.

**GPU-Friendly Workload Patterns** Workloads with non-uniform control dependencies across subgroups often perform poorly on GPUs, as they challenge the SIMD execution model and limit efficiency. To maximize performance, developers should aim to minimize global memory traffic and instead favor shared memory and subgroup intrinsics for fine-grained thread communication.

## 2.3 GPU Programming Frameworks

GPU Programming frameworks attempt to make following this philosophy as easy as possible. Cuda, Vulkan, OpenCL, Metal and WebGPU are examples of GPU programming frameworks. Each of these frameworks provides an interface for the GPU in the form of a programming language. And each of these frameworks has their own specification that outlines the rules for the language. CUDA and Metal are proprietary frameworks that are only built to work on Nvidia and Apple devices respectively. OpenCL, Vulkan, and WebGPU are portable frameworks that can work on a variety of devices. Vulkan and OpenCL are managed by the Khronos Group, a non-profit organization focused on establishing effective open industry standards for

how certain GPU programming languages should behave. These frameworks work on most GPU vendors except Apple [22]. WebGPU is a newer framework managed by W3C. W3C is a non-profit organization focused on effective open standards for the web. There exists multiple WebGPU language implementations but the most mature one, Dawn WebGPU, is spearheaded by Google. Their framework can translate to Vulkan’s language, Apple’s Metal and Microsoft’s HLSL making WebGPU more portable [62]. WebGPU comes with a higher level shading language called WGSL and can be compiled into lower level code for any GPU. OpenCL’s high level language is called OpenCL while the Khronos Group still supports OpenCL, Vulkan is more modern and widely supported. Vulkan doesn’t come with a single high level shading but instead comes with an intermediate representation called SPIR-V. Usually, other high level shading languages like OpenCL, WGSL, and HLSL are compiled to SPIR-V for execution on a Vulkan backend. In this work we focus on WGSL and OpenCL as our high level languages and use CLSPV to compile OpenCL to SPIR-V. And we let WebGPU compile WGSL to SPIR-V on Vulkan machines and Metal on Apple machines. The reason we use OpenCL even though WGSL can compile to SPIR-V is that WGSL is a very new language and has an immature implementation. For example, compiling to SPIR-V through WGSL doesn’t expose certain subgroup features that are necessary for a more performant prefix-scan.

## **2.4 Prefix-scan**

In the following sections we will begin to dive into prefix-scan, known colloquially as scan, first abstractly as an algorithm and then become more specific with how to run it effectively

on GPUs. We begin by defining the reduction operator, which forms the foundation of the scan operation.

**The Reduction Operator** The reduction operator takes in a list as input, performs an operation on the list, reducing it down to a single value. The scan reduction operator computes a scan and only returns the final value. For brevity we will now refer to a scan reduction as just a reduction. Prefix-scan can be thought of as a series of reductions of the input list. To illustrate, let's use  $[1, 2, 3]$  as our example.  $[1, 2, 3] \rightarrow [1, 3, 6]$ . This is prefix-sum i.e., prefix-scan using the addition operator. We compute this through a series of reductions. The first element of the output is the reduction of the first element  $[1] \rightarrow 1$ . The second element of the output is the reduction of the first two elements  $[1, 2] \rightarrow 3$ . The third element of the output is the reduction of all the elements  $[1, 2, 3] \rightarrow 6$ . These three reductions return  $[1, 3, 6]$ , the solution to the initial scan.

**Scan Algorithms: Serial Scan** Many algorithms for scan exist. The first and most straightforward of them all is serial scan. In this strategy reductions are computed serially from start to finish as exemplified in the previous section. This takes  $O(n)$  binary associative operations. In this strategy we leverage  $out[i - 1]$  to compute  $output[i]$ . Reductions are automatically memoized because they are all solutions in the output array. Therefore the memoization is trivial but affords a perfectly work efficient solution for computing prefix-scan, that is, there is no redundant computation. This straightforward implementation also illustrates the lower bound of operations. Prefix-scan is bounded at  $O(n)$ ,  $n$  operations for an array length of  $n$ , i.e. there is no way of calculating a prefix-scan with less than  $n$  operations [54, 49]. After seeing this





Figure 2.3: Serial Scan.

algorithm, it may seem like scan is an inherently serial process given elements depend on immediately adjacent elements but this is far from true. In reality, each reduction does not need to be computed in order, that is, we can compute groups of reductions in any order we want and combine them later. As long as each reduction is computed in order the associative operator affords the ability to compute multiple groups simultaneously.

**Scan Algorithms: Hillis-Steele** Hillis and Steele were the first to introduce a method that did not depend on serial calculation and is therefore parallelizable.

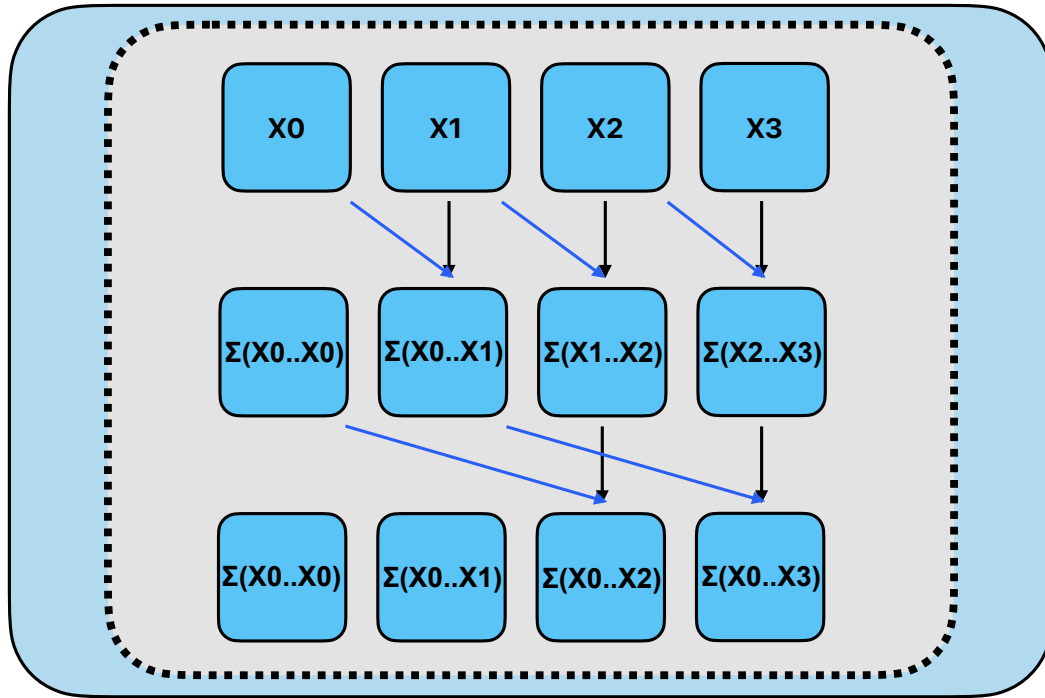


Figure 2.4: Hillis and Steele scan algorithm.

Since within a depth level there are no dependencies, multiple processors can work on each calculation at the same time. This introduces the idea of how parallelization can improve prefix-scan. However, compared to the sequential algorithm it does not scale well, due to poor work efficiency. While serial scan performs  $n$  operations for the entire input, Hillis-Steele performs  $n \log n$  operations, where  $n$  is the input size. The amount of redundant operations can cause the algorithm to perform worse than the serial scan for large inputs [19].

**Scan Algorithms: Blelloch** Four years later in 1990, Blelloch introduced a novel algorithm that solved the problem of work inefficiency. Hillis-Steele and Blelloch's strategies both decompose the input by splitting work up in halves but Blelloch, in his strategy, utilizes a tree

structure to organize an upsweep and downsweep phase that computes prefix-scan at the same time asymptotically as the serial scan greatly improving upon the Hillis-Steele method. In the first phase of Blelloch 1990, a reduction is computed in place leaving the array to have many partial sums. These sums can be added up in a downsweep phase to generate a full prefix-sum [4].

Figure 2.5: Upsweep & downsweep phases of the Blelloch 1990 scan algorithm.

This version solves the problem of work inefficiency and removes adjacent dependencies inviting parallelization. This algorithm takes exactly  $2n + 1$  operations. Other algorithms for scan exist such as Brent-Kung, Sklansky and Kogge-Stone but do not improve upon Blelloch in work efficiency. We also don't dive into these other algorithms because their structures are meant for circuits or other applications but Blelloch's algorithm is the best suited for GPU style parallelism [53, 25, 30, 7]. Even within GPUs, different scan algorithms are employed at different levels of the GPU hierarchy subgroup, workgroup, and device, to best exploit the architectural features and memory models available at each level. We will go into the details of these strategies in the next sections.

## Chapter 3

### Methodology

We begin by surveying various GPU prefix-scan strategies across the GPU hierarchy. We then identify the key parameters that define our implementation and introduce our tuning framework.

#### 3.1 Chained Scan: A Device-wide GPU Scan Strategy

Chained Scan is a general parallel strategy for computing prefix scans efficiently on GPUs. The method breaks the input into blocks, each of which is processed independently by a workgroup. Workgroups compute a “local” scan, that is, a scan of just their block, in fast shared memory, producing both scanned values and a reduction. The reductions from each block are themselves scanned to compute offsets for each block. Finally, each block’s scanned values are adjusted by its corresponding offset to produce the correct global prefix sums. This approach effectively utilizes the unique computing resources and memory types available at each level of the GPU hierarchy. Only the reductions across blocks need to propagate through global

memory. Here is an example of using the trivial case where the input of prefix-scan is just ones and the operator is addition. Using the trivial case as an example makes it easy to see how Chained Scan works abstractly without implementation specific details.

```
# Input:
[1, 1, 1, 1 | 1, 1, 1, 1 | 1, 1, 1, 1 | 1, 1, 1, 1]

# Phase 1 (block scans):
Block 0: [1, 2, 3, 4] → reduction = 4
Block 1: [1, 2, 3, 4] → reduction = 4
Block 2: [1, 2, 3, 4] → reduction = 4
Block 3: [1, 2, 3, 4] → reduction = 4
# Block Reductions:
[4, 4, 4, 4]

# Phase 2 (scan block reductions, these are the "offsets"):
[4, 8, 12, 16]

# Phase 3 (propagate offsets to each block):
Block 0: [1, 2, 3, 4] ← offset (None)
Block 1: [5, 6, 7, 8] ← offset +4
Block 2: [9, 10, 11, 12] ← offset +8
Block 3: [13, 14, 15, 16] ← offset +12

# Output:
[1, 2, 3, 4 | 5, 6, 7, 8 | 9, 10, 11, 12 | 13, 14, 15, 16]
```

Figure 3.1: Pseudo-code for Chained Scan algorithm.

We adopt Chained Scan as our device-wide strategy because of its inherently modular structure. This approach naturally introduces two important parameters to our framework.

1. **Chained Scan Variation:** The strategy by which the GPU aggregates block reductions to complete the full scan.
2. **Inter-block Scan:** The scan implementation used by each workgroup to compute a scan

over its assigned block.

We will first go into Chained Scan variations.

## 3.2 Chained Scan Variation: Sequential

There are three main variations of Chained Scan. We begin with the most straightforward: the *Sequential Chained Scan*.

**Overview** In this approach, the input array is divided into blocks, with each block assigned to a separate workgroup. Each workgroup performs a scan and computes a reduction over its assigned block. The reductions of these blocks are collected and scanned over to add as offsets back to the blocks. The most straightforward way to accomplish this is by storing reductions in an intermediate array as shown in the previous section. For further details, we use StreamScan’s approach.

**Implementation** To begin, an intermediate array is initialized with a flag  $U$  to indicate that a block’s computation is unfinished. Each workgroup first performs a scan over its own block of data. Then, it repeatedly checks the value at `block_id - 1` in the intermediate array, waiting until that entry contains a completed reduction value instead of  $U$ . Once available, the workgroup adds its own block’s reduction to this value and writes the result to its corresponding `block_id` position in the intermediate array. To generalize, this is effectively a serial scan implemented at the workgroup level.

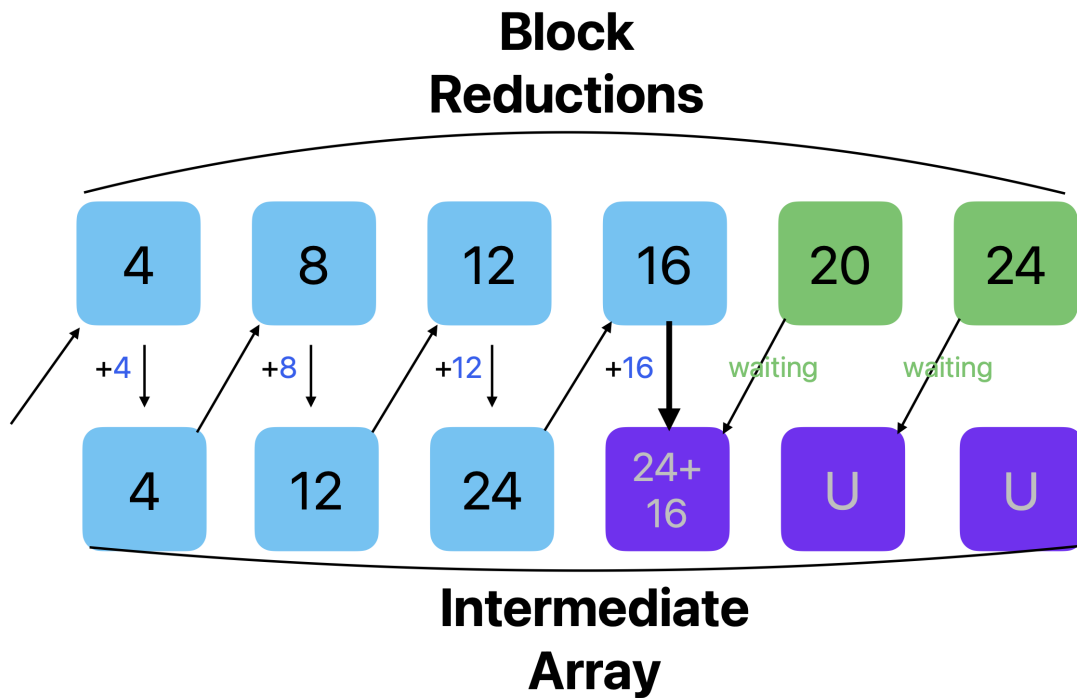


Figure 3.2: Sequential Chained Scan visualization.

**The Problem With Sequential Chained Scan** In large scans, workgroups often idle while waiting for their predecessor's result in global memory. This sequential dependency creates significant latency, as thousands of workgroups stall instead of issuing useful instructions to the pipeline. Because blocks may still be incomplete, adjacent scan values can't be computed in parallel without coordination. If workgroups could detect when their inputs were ready, partial scans could continue concurrently, improving throughput.

### 3.3 Chained Scan Variation: Decoupled Lookback

The Decoupled Lookback strategy from Duane Merrill and Michael Garland’s 2016 Nvidia paper addresses this issue. This version of Chained Scan introduces three key changes.

**Key Changes** First, instead of spinning until the global reduction reaches their index in the intermediate array, workgroups immediately write their own reduction to the array as soon as they finish their local scan. Second, because the intermediate array can now contain values that are not final reductions (i.e., not just the global reduction or  $U$ ), each entry must always include a status flag indicating the status of the reduction. Third, rather than waiting for the global reduction to reach their position, workgroups look back through the intermediate array, starting from the previous index, and accumulate reductions from earlier blocks. This allows workgroups to continue computing partial scans and decouples adjacent dependencies. As more partial reductions are computed, workgroups begin to incorporate earlier reductions into their own. Over time, these partial results accumulate across blocks, and once all block reductions have been combined, the full device-wide scan is complete. This may sound a bit like magic, so let’s fill in some gaps starting with the contents of the intermediate array.

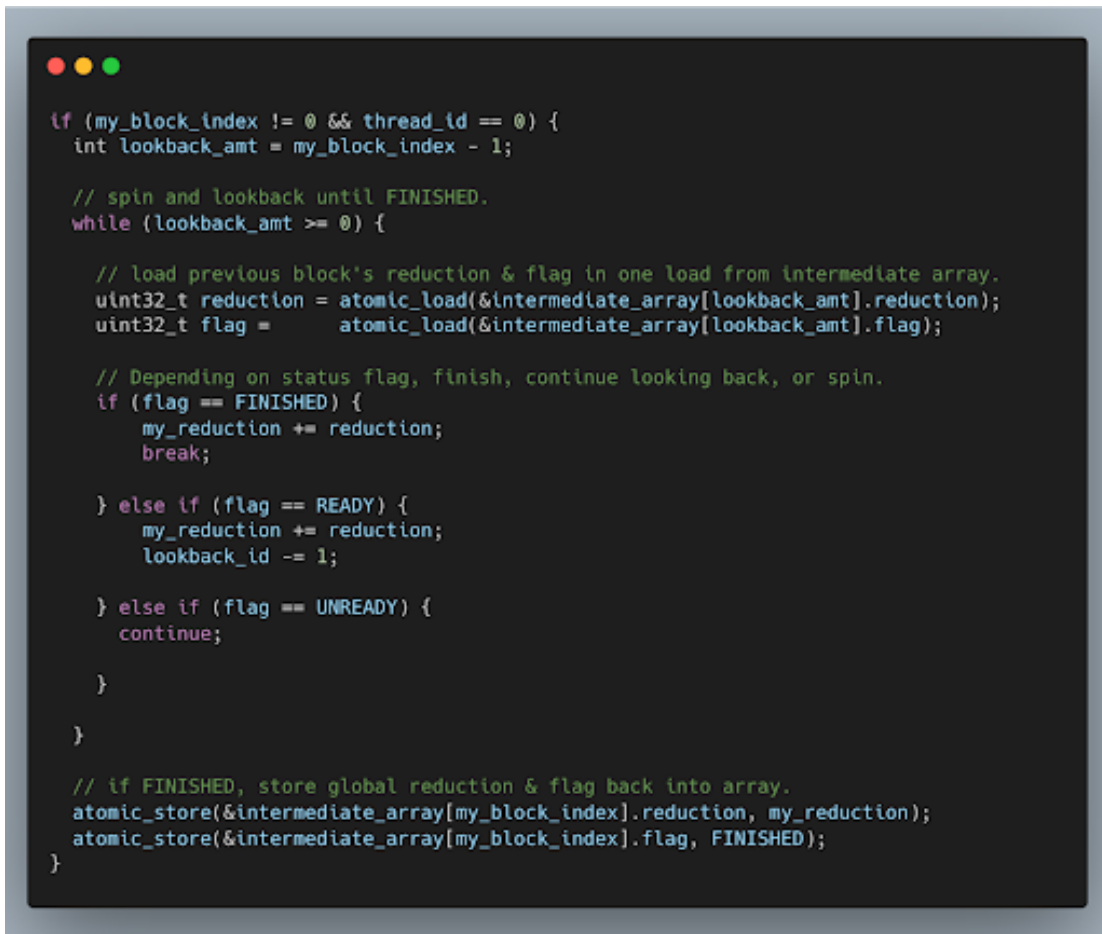
**Implementation** Instead of storing just a reduction *or* a  $U$  flag, each element now holds two pieces of information: a status flag and a reduction value (if available). We’ll cover more details of this process in subsequent sections, but for now, it’s important to understand that the status flag indicates one of three possible states. We’ll begin with the first two. The first state is `UNREADY`, meaning the block-wide scan hasn’t completed yet, and no reduction is available.



The second is `READY`, meaning the block-wide scan has completed, and the reduction has been written to the intermediate array. Once a workgroup reaches the `READY` state, it begins to “look back” in the intermediate array. If the previous block is still `UNREADY`, the workgroup must wait, spinning until the needed reduction becomes available so it can aggregate it into its own. Workgroups may not skip over `UNREADY` blocks and aggregate further reductions because this would change the order of operations and scan operators are not promised to be commutative. The third possible state for a block is `FINISHED` (in contrast to `UNREADY` and `READY`). When a block has the status flag `FINISHED`, it means its reduction value in the intermediate array includes all reductions from preceding blocks. In other words, it is no longer a partial result but a complete scan up to that point. The first block to reach the `FINISHED` state is always block 0, since it has no predecessors and its scan is, by definition, complete. The `FINISHED` state propagates when a `READY` block aggregates a `FINISHED` block’s result, effectively becoming `FINISHED` itself. This process continues as workgroups, in parallel, build on previously finished results. Eventually, all blocks become `FINISHED`, meaning each one contains the complete reduction of all preceding blocks, which is the definition of a prefix-scan.

### 3.4 Chained Scan Variation: Parallelized Decoupled Lookback

Parallelized Decoupled Lookback is an optimization of the standard lookback phase that takes advantage of subgroup-level parallelism. Instead of checking one block at a time sequentially, a full subgroup looks back across a range of consecutive blocks in parallel, speeding up the process of finding and aggregating prior reductions. To take advantage of subgroup paral-



```

if (my_block_index != 0 && thread_id == 0) {
    int lookback_amt = my_block_index - 1;

    // spin and lookback until FINISHED.
    while (lookback_amt >= 0) {

        // load previous block's reduction & flag in one load from intermediate array.
        uint32_t reduction = atomic_load(&intermediate_array[lookback_amt].reduction);
        uint32_t flag =      atomic_load(&intermediate_array[lookback_amt].flag);

        // Depending on status flag, finish, continue looking back, or spin.
        if (flag == FINISHED) {
            my_reduction += reduction;
            break;

        } else if (flag == READY) {
            my_reduction += reduction;
            lookback_id -= 1;

        } else if (flag == UNREADY) {
            continue;

        }

    }

    // if FINISHED, store global reduction & flag back into array.
    atomic_store(&intermediate_array[my_block_index].reduction, my_reduction);
    atomic_store(&intermediate_array[my_block_index].flag, FINISHED);
}

```

Figure 3.3: OpenCL-like Sequential Decoupled Lookback pseudo-code.

lelism, all the blocks being looked back onto must be either `READY` or `FINISHED`. If any block in that range is still `UNREADY`, the subgroup can't proceed, just like in the sequential version, where threads must spin at `UNREADY` blocks before continuing the scan. If a subgroup looks back and finds that all threads are `READY`, it scans the corresponding reductions in parallel, computing partial reductions for multiple blocks at once. If one or more of the blocks is `FINISHED`, the highest `FINISHED` block holds the most complete global reduction so far. Using that as a starting point, the subgroup can scan through the `READY` blocks that follow, adding their reductions

to compute an even more mature `FINISHED` reduction.

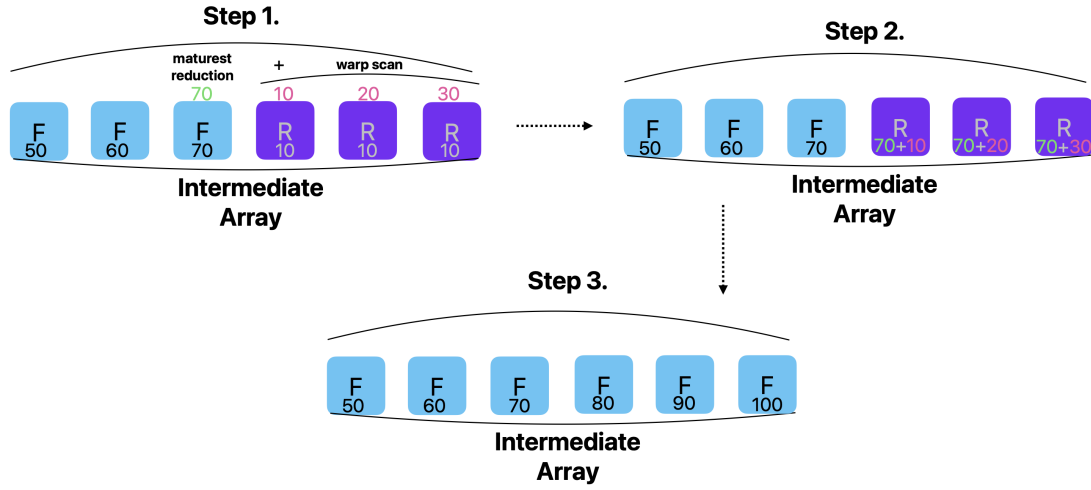


Figure 3.4: Visualization of Parallelized Decoupled Lookback.

### 3.5 Intra-block Scan: Approach and Implementations

This section covers intra-block implementations of scan. Since these scans are limited to the hardware resources of a single workgroup, the design considerations are different from the abstract view of scan algorithms. Intra-block scans are usually small in size, which directly impacts how they're implemented and how they perform. As mentioned, each block is handled by a single workgroup, and at peak utilization, a workgroup can occupy only one compute unit. Because of this, block-level input sizes are often too small to approach the asymptotic limit, so the algorithm's time complexity matters less than how it performs at realistic sizes. Overhead is another consideration that time analysis does not address. Algorithms that maximize parallelism can also incur overhead, as thread scheduling introduces its own costs. With these

considerations in mind we focus on two main implementations for intra-block scan. Blelloch 1990 and Blelloch SIMD Raking.

**Blelloch 1990** We'll start with Blelloch 1990, the same algorithm introduced in the background. As previously explained, it has the same asymptotic efficiency as serial scan, performing  $2N + 1$  operations compared to  $N$  in the serial version. This algorithm maximizes thread parallelism by ensuring all workgroup threads are active and performing an equal share of the work at all times. For smaller global inputs, this can introduce unnecessary overhead due to missed opportunities to use faster intra subgroup communication and by incurring the cost of a workgroup barrier needed at each depth level to keep shared memory coherent.

**SIMD Raking** SIMD raking takes the opposite approach, using a single subgroup to compute the entire block scan [6]. This minimizes overhead for intra-workgroup communication since all threads operate within the same subgroup. SIMD Raking trades abundant parallelism for reduced communication overhead. This technique is related to Brent's theorem, which says that fixed overhead can be mitigated by increasing the amount of work assigned to each thread [8, 23]. In this strategy, each thread in the subgroup is partitioned a chunk of the block to compute serially. Then the reduction of each chunk is scanned using subgroup scan, effectively computing a chained scan of the block. This chained scan doesn't need any fancy propagation strategy because reductions are scanned using subgroup scan. Using subgroup scan enables the lightning fast propagation and computation because memory operations at subgroup level are extremely fast compared to others. Both SIMD Raking and Blelloch (1990) are effective block-level strategies, each with distinct strengths.

### 3.6 More Tuning Parameters

**Workgroup and Thread Count** The number of workgroups and the number of threads per workgroup are key tuning parameters when dispatching GPU kernels. These choices directly impact how the workload is split across the GPU and how efficiently hardware resources like shared memory are used.

**Memory Batching** Another key consideration in prefix-scan is per-thread loads (batch size): determining how many elements each thread should load from global memory at once to optimize performance. Merrill et al. suggest that batch sizes of  $2^0$ ,  $2^1$ , and  $2^2$  elements are reasonable for per-thread loads in the context of a memory copy operation. Initially, we followed this guidance, as prefix-scan is a memory-bound operation at large input sizes. However, later observations suggested that this limitation does not always hold. So we parameterize prefix-scan on batch size as well [31].

**Data Types** Building on this, it's just as important to consider which type of element to load, whether packed types like `vec2` or `vec4`, or scalars like `uin32_t` or `float64`. Performance varies significantly depending on the type used. Because of this, we parameterize the scan implementation based on data type.

### 3.7 Why are we tuning?

With all parameters considered, there are over 3,500 possible kernel configurations. Given the size of this space, identifying the optimal combination for a given device is non-trivial.

Prior work has shown that per-device tuning can significantly improve kernel performance, and when combined with per-input-size tuning, it can even outperform vendor-optimized implementations by up to 230% [29, 65, 11, 59, 47]. To evaluate kernel performance for prefix-scan, we use throughput measured in gigabytes per second (GB/s). Since prefix-scan is memory-bound, throughput gives a direct view of how effectively memory bandwidth is being used. Specifically, we base this on input size, accounting for one full read from global memory ( $N$  reads) and one full write back ( $N$  writes), for a total of  $2N$  memory operations.

### 3.8 Tuning Approach: Execution Details

Each parameter has been introduced in previous sections. Here, we formalize these parameters as part of our tuning framework, outlined in Table 3.1.

Parameter	Values
Batch Size	1, 2, 4, 8, 16
Data Type	vec2, vec4
Workgroup Size	$\{2^i \mid \log_2(\text{subgroup size}) \leq i \leq \log_2(\text{device limit})\}$
Workgroups	$\{2^i \mid 16 \leq i \leq \log_2(\text{device limit})\}$
Lookback Strategy	Parallelized Decoupled, Sequential Decoupled
Block Scan	SIMD Raking, Blelloch (1990)

Table 3.1: Summary of tunable parameters in our prefix-scan framework.

**Generating Configurations** A *configuration* refers to a specific combination of tuning parameters. Each configuration corresponds to a particular input size, since the total number of input elements is determined by multiplying certain parameters together. The input size, measured in number of elements, is computed as follows:

`batch_size * thread per workgroup * workgroups * memory type = number of elements`

If the element is a `uint32_t` then the input size in bytes is calculated by multiplying the above equation by four.

**Parameter Passing** To assist the compiler, parameters are interpolated directly into the source code before compilation rather than passed through buffers. This allows the compiler to treat them as constants, enabling more aggressive optimizations.

**Parameters Constraints** In general, we restrict the number of elements to powers of two ranging from  $2^{10}$  to  $2^{29}$ . Within this range, all configurations are valid, that is, all parameter combinations can execute together without compatibility issues. The only exception is that configurations causing any single element of the scan to exceed  $2^{30}$  are invalid, since our variables require two bits to store the status flag. This detail will be examined further in the following chapter.

**Exhaustive Searching** Rather than focusing solely on finding the single best-performing configuration, our framework performs an exhaustive sweep over the entire parameter space. This comprehensive approach allows us to analyze how performance varies across devices and input sizes, yielding data that informs tuning decisions across a wide range of deployment scenarios. While optimization strategies like genetic algorithms, simulated annealing, or Bayesian optimization have been shown to efficiently discover high-performing configurations, our primary goal is to characterize the broader performance landscape, with an emphasis on portability across hardware platforms and environmental constraints [9].

## Chapter 4

### Implementation

Having established our overall tuning strategy and algorithmic approach, we now turn to the specifics of our implementation. This section begins with an overview of the tools, frameworks, and setup used for each kernel. We then discuss the initial challenges encountered during development. Finally, we describe the device-specific, framework-specific, and GPU-wide adaptations required to achieve correct and efficient execution across a range of hardware.

#### 4.1 Toolchain and Frameworks

This work aims to build a portable and performant prefix-scan. We use WebGPU and Vulkan as our graphics backends, targeting them with two main kernels: one written in OpenCL and the other in WGSL. To run on Vulkan, we use CLSPV, which compiles OpenCL code to SPIR-V. WGSL, on the other hand, works natively with WebGPU, so no transpilation tool is needed in that case. For our Vulkan setup, we use EasyVK, a simplification layer developed by CHPL that makes Vulkan more accessible for lightweight GPU compute tasks [26]. Nor-



mally, setting up Vulkan can take hundreds of lines of boilerplate code, even for something as simple as vector addition. With EasyVK, the same functionality can be achieved in 30 lines of C++ or less. For our WebGPU distribution, we chose the Dawn because it’s currently the only implementation that supports subgroups. We initially developed using WebGPU Native, which enables writing WebGPU setup in C++, but it doesn’t allow selecting a specific GPU on multi-GPU systems. To work around this, we switched to a JavaScript setup and ran the algorithm in the Google Chrome browser, where we could choose the target GPU using Chrome flags. While we successfully built functional versions in both frameworks, WebGPU’s limited subgroup support hindered the correctness of our implementation. As a result, we primarily focus on Vulkan for much of the results section. Vulkan is state-of-the-art and only lacks Apple support compared to WebGPU, making it a solid platform for illustrating our core ideas.

## 4.2 Vulkan: Initial Issues

We began developing a GPU prefix-sum (prefix-scan using the addition operator) implementation in March 2024 using Vulkan, since WebGPU didn’t support subgroups at the time. Our design follows NVIDIA’s prefix-sum paper, using Parallelized Decoupled Lookback for inter-block strategy and SIMD Raking for intra-block reductions. Early in the development we were met with many issues. All five of our test devices failed to run the implementation out of the box. Driver bugs, compiler issues, and vendor-specific quirks all contributed to the lack of functionality [55].

Device	Status
AMD XT 7900	Broken
AMD Radeon Graphics	Broken
Intel ARC A770	Broken
Intel UHD 770	Broken
Nvidia RTX 4070	Broken

Table 4.1: Summary of device functionality at the beginning of development.

**Intel Subgroup Issue** The first fix we found was for our Intel cards. The behaviour was that prefix-sum would non-deterministically produce incorrect results. After thoroughly checking the code and ruling out programming bugs, we discovered that Intel cards don't always enforce full subgroups leading to unexpected behavior. After some digging, we found a flag to pass during shader creation that forces full subgroups. Without this flag:

```
VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT
```

full subgroups aren't guaranteed, which is necessary for our implementation. Adding this flag enabled prefix-sum to work correctly on our Intel GPUs [21].

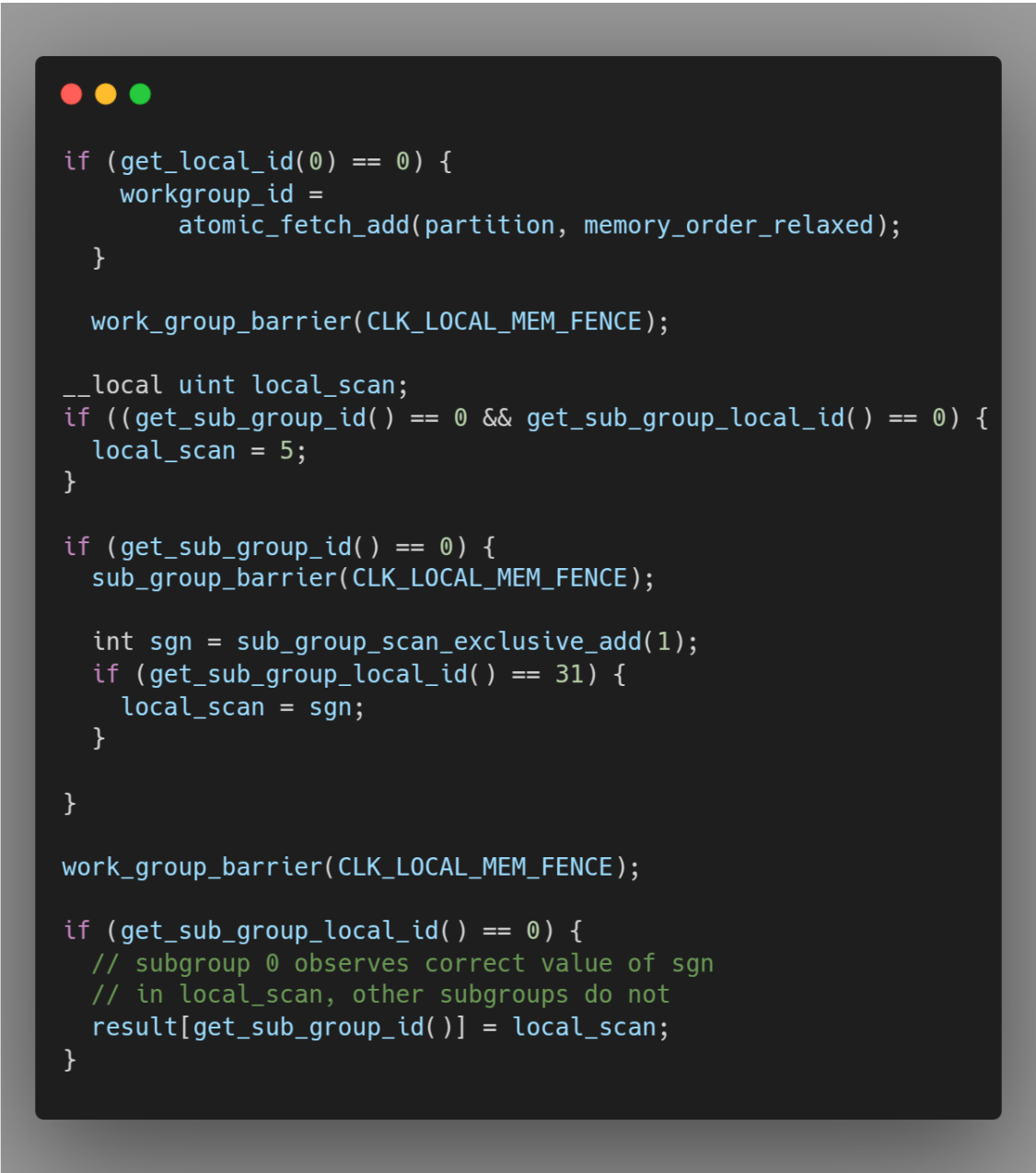
Device	Status
AMD XT 7900	Broken
AMD Radeon Graphics	Broken
Intel ARC A770	Running
Intel UHD 770	Running
Nvidia RTX 4070	Broken

Table 4.2: Summary of device functionality after adding the require full subgroups flag.

### 4.3 Vulkan: Framework and Toolchain Bugs

Next, we addressed our AMD and Nvidia devices. Both consistently failed to produce correct results on any input size when using more than one workgroup. This behavior pointed us toward the Parallelized Decoupled Lookback section which only executes if more than one workgroup is active. However, due to its complex control logic and synchronization, the issue wasn't immediately obvious. On both devices, we attempted to remove unneeded sections of the implementation to isolate the problem. However, seemingly unrelated code continued to affect the result, complicating the debugging process. After reducing the kernel to minimal examples that consistently reproduced the issue, we confirmed that our implementation was correct, indicating the failure was not due to a programming bug on our end.

**Kernel Behavior: NVIDIA** When the kernel executes, thread 31 of subgroup 0 is expected to update `local_scan` with the result of `subgroup_scan_exclusive_add`. However, the workgroup memory barrier fails to propagate this value to higher subgroups, which instead observe the initial value (5). The failure mode of this bug is particularly complex, seemingly unrelated code, such as the dynamic workgroup allocation at the top of the kernel, turns out to be essential for reproducing the issue.



```

if (get_local_id(0) == 0) {
    workgroup_id =
        atomic_fetch_add(partition, memory_order_relaxed);
}

work_group_barrier(CLK_LOCAL_MEM_FENCE);

__local uint local_scan;
if ((get_sub_group_id() == 0 && get_sub_group_local_id() == 0) {
    local_scan = 5;
}

if (get_sub_group_id() == 0) {
    sub_group_barrier(CLK_LOCAL_MEM_FENCE);

    int sgn = sub_group_scan_exclusive_add(1);
    if (get_sub_group_local_id() == 31) {
        local_scan = sgn;
    }
}

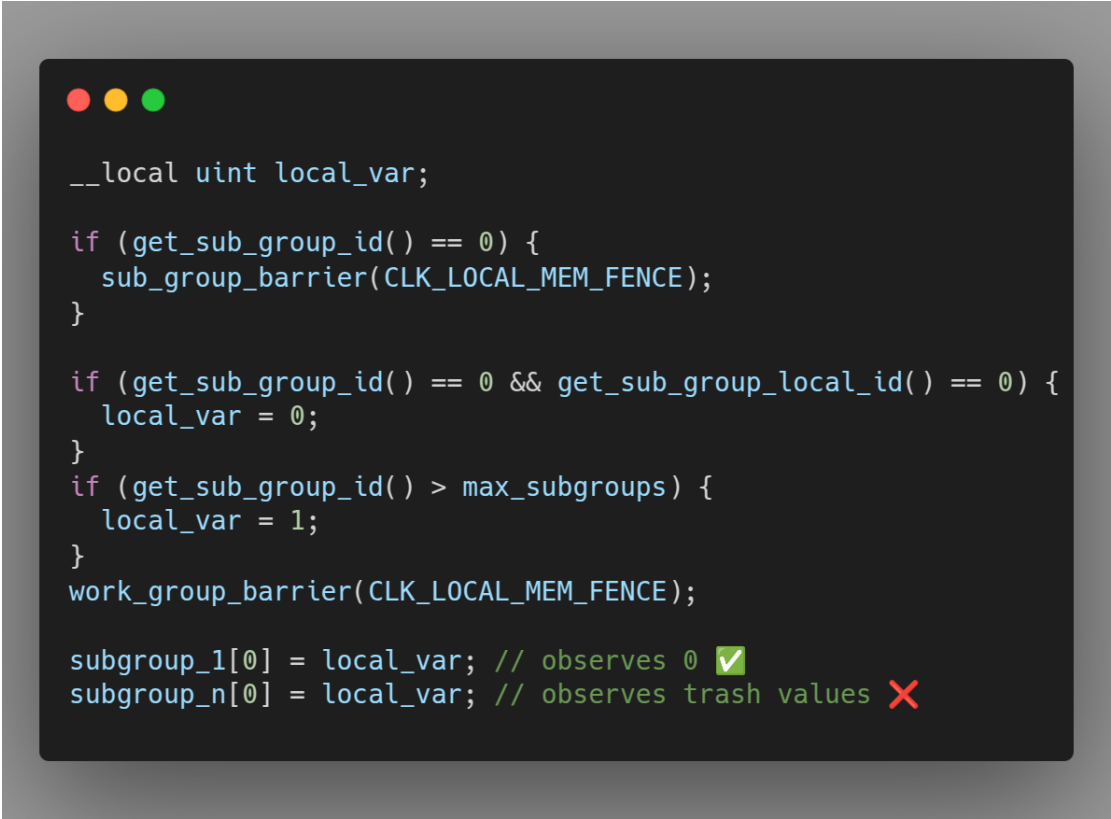
work_group_barrier(CLK_LOCAL_MEM_FENCE);

if (get_sub_group_local_id() == 0) {
    // subgroup 0 observes correct value of sgn
    // in local_scan, other subgroups do not
    result[get_sub_group_id()] = local_scan;
}

```

Figure 4.1: Minimized code example of the Nvidia Vulkan bug.

**Kernel Behavior: AMD** Upon launching the kernel, `local_var` is initialized, and some work happens in the first subgroup that needs synchronization. Then, the first thread of subgroup zero sets `local_var` to zero. Next comes an untaken branch, this branch can be anything as long as it's not simple enough for the compiler to optimize away (if it's a compile-time constant, the bug disappears). The expected behavior is that after the workgroup memory barrier synchronizes `local_var`, every thread in the workgroup should see it as zero. Instead, threads in higher subgroups observe random values. The failure itself is not as complex as Nvidia's but still requires an interesting setup.



```
__local uint local_var;

if (get_sub_group_id() == 0) {
    sub_group_barrier(CLK_LOCAL_MEM_FENCE);
}

if (get_sub_group_id() == 0 && get_sub_group_local_id() == 0) {
    local_var = 0;
}

if (get_sub_group_id() > max_subgroups) {
    local_var = 1;
}

work_group_barrier(CLK_LOCAL_MEM_FENCE);

subgroup_1[0] = local_var; // observes 0 ✓
subgroup_n[0] = local_var; // observes trash values ✗
```

Figure 4.2: Minimized code example of the AMD Vulkan bug.

**Debugging the Toolchain** Given that the issue was not due to our own programming, we turned to the two key tools involved in handling our code. One of them is CLSPV, which compiles our OpenCL source into SPIR-V. It was possible that CLSPV was incorrectly generating SPIR-V during compilation. To verify this, we compiled pairs of nearly identical kernels, one that exhibited the bug and one that did not, and traced their execution to identify any divergence. This allowed us to check whether CLSPV produced only the expected differences between the two. For AMD, we modified the kernel by moving the definition of `local_var` from subgroup zero to subgroup one, for Nvidia we made a very similar change. We include the SPIR-V output for AMD below: the left kernel does not reproduce the bug, while the right one does. Since this is the only difference and it aligns with our intentional change, we conclude that CLSPV is not the source of the issue.

<pre>... uint_1 = OpConstant %uint 1 ...</pre>	<pre>... uint_0 = OpConstant %uint 0 ...</pre>
--	--

With programming errors and CLSPV ruled out, the bug was likely rooted in the Vulkan implementations provided by AMD and Nvidia. Since both bugs originate in the lookback sections of our implementation, developing reliable workarounds would be challenging. As a result, we reported the issues to the respective Vulkan driver teams at AMD and Nvidia in the hope that they could be patched.

**Vulkan Implementation Bug Solutions** On Nvidia’s side they swiftly acknowledged and patched the bug in their Windows 553.22, Linux 550.40.76 Vulkan drivers [40]. However, we never received a response from AMD. Fortunately, AMD’s MESA drivers did not exhibit the

Device	Status
AMD XT 7900	Running
AMD Radeon Graphics	Running
Intel ARC A770	Running
Intel UHD 770	Running
Nvidia RTX 4070	Running

Table 4.3: Summary of device functionality after all aforementioned changes.

same behavior resulting in finding a functioning implementation on all five devices.

From here, we attempted to gather performance results, but the discrete devices were reporting lower throughput than the integrated ones pointing to an issue with our memory handling.

**Device-Local Memory: Benchmarking Limitations** We discovered that the issue stemmed from our use of host-device visible memory. Although convenient, this type of memory is accessible to both the CPU and GPU, and significantly slower than device-local memory. To address this, we implemented a staging process, data is first written into a CPU-accessible buffer, then transferred into faster, device-local memory for execution. This staging strategy enables accurate throughput performance measurements on devices which don't have host-device unified memory.

**Transpilation Issues: CLSPV Bug** After taking an extended break from developing we came back and found that CLSPV was failing to compile our kernel. The kernel however had not been touched for a good amount of time and the compiler was segfaulting so this pointed to an issue with CLSPV as opposed to a bug in our code. In order to be sure this was the case we also tried compiling our program using the CLSPV compiler on `godbolt.org` which hosts the most up to

date versions of various compilers [13]. After observing the online compiler segmentation fault we posted an issue to CLSPV's issue section. According to the investigation by the developers there was a bug with pointer bistreaming. Luckily, the bug was quickly resolved within two days and we were able to continue working.

## 4.4 Vulkan: Formal Verification

In these earlier versions of our scan we were constantly unsure of our program's correctness due to the myriad of bugs we encountered. This was especially true as we started increasing our tuning space, adding new parameters and constructions to support new optimizations. In order to be sure of certain parts of our program we employed Dartagnan. Dartagnan is a tool for checking state reachability under weak memory models [43, 10]. It helps verify that concurrent CUDA or Vulkan programs behave correctly under relaxed-memory conditions. Using this tool, we uncovered hidden race conditions in our Blelloch 1990 implementation, detecting violations between distant lines of code that could have caused future issues. After fixing said issues, we confirmed our Blelloch 1990, memory loading, and Sequential Decoupled Lookback code to be functionally correct. Dartagnan currently supports checking global and workgroup memory handling so checking of Parallelized Decoupled Lookback and SIMD Raking, which use subgroups, is not possible yet. After fixing these issues and months of carefully scrutinizing Parallelized Decoupled Lookback for bugs, we became confident that the rest of our code was race-free and ready to move forward with WebGPU.



## 4.5 WebGPU: Initial Issues

In January 2025, five months after subgroups were added to WebGPU, we began porting our Vulkan implementation. Unlike Vulkan, where toolchain and driver issues slowed us down, WebGPU progressed more smoothly with no framework-level bugs. However, we did encounter a compiler issue with WebGPU's memory safety features.

**Bounds Checking in WGSL** When first building our WGSL implementation, we mistranslated a piece of code from OpenCL causing our WGSL code to be incorrect:

```
if (local_id.x == 0) {  
    atomicStore(&prefix_states[part_id], shared_data[local_id.x - 1]);  
}
```

This WGSL code is incorrect because `shared_data[-1]` is being accessed every time. While the correct line is `shared_data[local_size - 1]`. The interesting part is that this code worked on all the devices for several months even though it relied on the wrong value. While optimizing throughput, we tested a new Chrome Dawn flag, `disable_robustness`, which disables bounds checking. This broke the program and exposed the issue. It turns out that, by default, WebGPU clamps out-of-bounds array accesses, which silently turns `-1` into `array.length()` keeping the program running with the wrong logic.

```
index = 5;  
value = buffer[min(index, buffer.length - 1)];
```

This was curious because under the hood `disable_robustness` removes a `Min()` function so it was initially confusing why the last index was being returned and not zero. The most likely reason is that the incorrect value, `-1` and `UINT_MAX` are being confused. `Min()` likely expects an unsigned number, and `-1` looks like `UINT_MAX` bitwise.

```
index = min(UINT_MAX, buffer.length - 1)
```

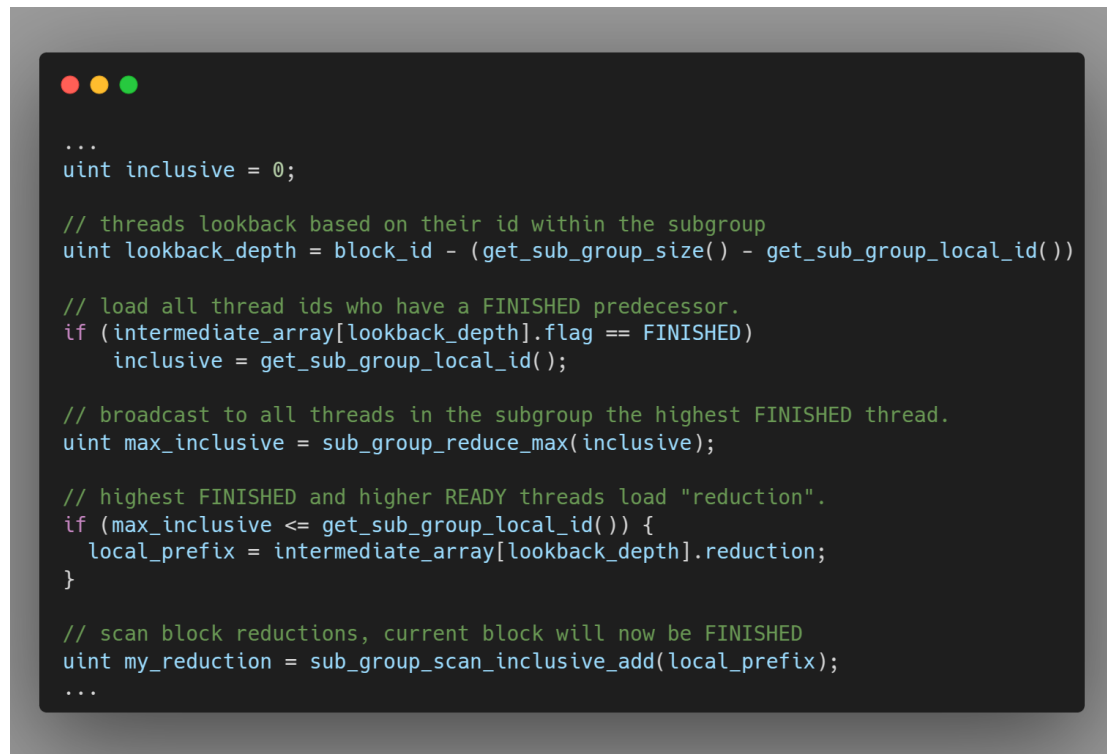
This feature made debugging trickier by masking this bug. While using `min` ensures memory safety, a more transparent approach, like raising errors or returning a defined default for out-of-bounds accesses, could help catch issues earlier and simplify reasoning about program behavior

## 4.6 Vulkan and WebGPU: Implementation Adaptations

For the majority of development we tested our implementation on six devices from four vendors, Intel, Nvidia, AMD and Apple. In the proceeding sections we will enumerate the implementation adaptations we found to be necessary to function on all of our devices and optimizations

**Three-Field Block Descriptor: Overview** Initially, our implementation utilized a three-field block descriptor. As a reminder the block descriptor has been previously described as storing the status flag and reduction of each block in the device wide scan. However, in the Decoupled Lookback paper the original descriptor is described as having three fields, the status flag, the aggregate, and the inclusive prefix. This descriptor holds the block reduction in two separate values, `aggregate` and `inclusive_prefix` depending on if the block's reduction is `FINISHED` or just `READY`. In OpenCL we implemented this using an array of structs where each struct had three fields, `status_flag`, `aggregate` and `inclusive_prefix`. The status flag field must be atomic since it is possible that while a workgroup is updating its block flag, another workgroup might want to look at its flag.

**Two-Field Block Descriptor: Overview** In a three-field block descriptor, threads must choose between loading either the `inclusive_prefix` or `aggregate` field. By merging these into a single `reduction` field, threads within the same subgroup avoid conditional branching, eliminating divergence and saving valuable subgroup cycles during each lookback.

A code editor window with a dark background and light-colored text. The code is in C++ and implements a lookback mechanism for a two-field block descriptor. It starts with a comment and a line to set `inclusive` to 0. Then, it calculates `lookback_depth` based on the block ID and subgroup size. A loop (represented by a single `if` statement in the snippet) checks if the predecessor is `FINISHED` and updates `inclusive`. A `sub_group_reduce_max` call broadcasts the highest `FINISHED` thread's value. Then, threads with `max_inclusive` less than or equal to their local ID load the `reduction` value into `local_prefix`. Finally, `sub_group_scan_inclusive_add` is used to apply the local prefix to the current block's reduction, marking it as `FINISHED`.

```
...
uint inclusive = 0;

// threads lookback based on their id within the subgroup
uint lookback_depth = block_id - (get_sub_group_size() - get_sub_group_local_id())

// load all thread ids who have a FINISHED predecessor.
if (intermediate_array[lookback_depth].flag == FINISHED)
    inclusive = get_sub_group_local_id();

// broadcast to all threads in the subgroup the highest FINISHED thread.
uint max_inclusive = sub_group_reduce_max(inclusive);

// highest FINISHED and higher READY threads load "reduction".
if (max_inclusive <= get_sub_group_local_id()) {
    local_prefix = intermediate_array[lookback_depth].reduction;
}

// scan block reductions, current block will now be FINISHED
uint my_reduction = sub_group_scan_inclusive_add(local_prefix);
...
```

Figure 4.3: Efficient, single value approach.

**Multi-Field Block Descriptor: Memory Ordering Requirements** Whether using a two or three-field block descriptor, explicit atomic memory ordering is required. Without it, the compiler may reorder memory operations, creating a race condition where a later workgroup, during a lookback, observes the status flag as `FINISHED` before the owning workgroup has written the updated reduction value. As a result, the reading workgroup may see a stale or

uninitialized reduction. To prevent this reordering from happening memory fences can be used. Memory fences prevent reordering of memory reads and writes across the line where the fence was written. This strategy works but is actually overkill for our purposes. Instead we can accomplish correctness by utilizing Vulkan's acquire-release memory semantics. Vulkan atomic operations have a second argument which allows you to declare the memory ordering. To formalize the problem, we want to be sure that all stores to the block reduction occur before an atomic write to `status_flag` so that another block can never load a stale reduction value. Similarly, we don't make any writes to `reduction` until after the `status_flag` has been properly loaded so that we never make stores based on a stale flag. Using the flag `memory_order_acquire` prevents loads and stores beneath that memory operation from being reordered while `memory_order_release` prevents loads and stores above it from being reordered. So we can use these semantics to efficiently enforce the necessary memory ordering by marking flag loads with `memory_order_acquire` and flag stores with `memory_order_release`.

**One-Field Block Descriptor Overview and WebGPU Acquire-Release Support** In WebGPU, global memory acquire-release semantics are not supported on global atomic operations. This prevents efficient two-field descriptor implementations from being possible in WebGPU. Thankfully there exists a one-field descriptor implementation which obviates the need for memory ordering semantics at all. In a one-field descriptor, the status flag and the reduction are bit packed into an atomic memory location. Due to flag-reduction loads and stores occurring in the same instruction it is never possible for either value to be stale. Therefore relaxed memory ordering suffices. Bit-masks are used to decompose and re-store the flag and reduction. For

memory location, we use an unsigned 32 bit integer and save the two most significant bits for the flag. This version however does not support values that can't be represented with less than 30 bits since two bits are being used for the flag. Fortunately, this is often unproblematic since, as mentioned in the background, many applications of scan target sparse arrays or use logical operators which don't accumulate values as big as  $2^{30}$ .

**One-Field Block Descriptor: Performance Evaluation** One-field block descriptors are more instruction-efficient than their two-field counterparts. In a two-field setup, each lookback thread must first load the status flag field of the descriptor. If the flag indicates readiness, a second load is issued to retrieve the reduction value. This results in one, and often two, global memory loads per thread. In contrast, a one-field descriptor always requires only a single load per thread, combining both status and data. This simplification reduces the number of global loads per thread during parallelized and sequential lookback, significantly improving lookback latency.

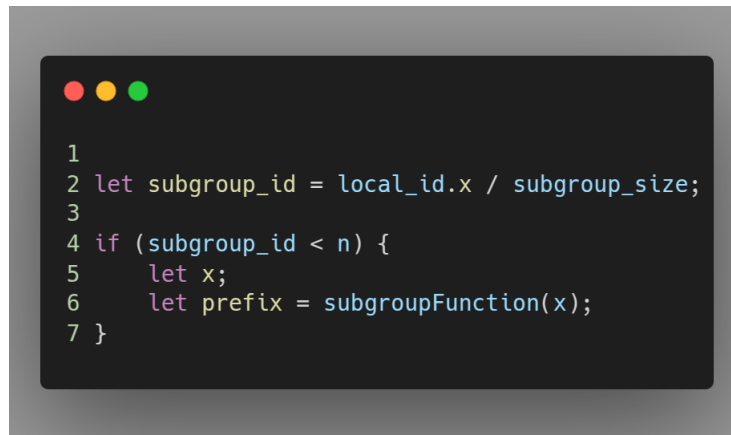
**Workgroup Scheduling: Runtime ID Allocation** Most frameworks' programming models assign each workgroup a numeric rank within the wider device environment. However, the order in which workgroups are actually scheduled is unrelated to this rank. Decoupled Lookback hinges on that rank to determine a block's place in the overall device-wide scan. If the device schedules non-adjacent workgroups and is fully occupied, some blocks may continue polling on UNREADY blocks whose workgroups were never, and will never be, scheduled. This behavior risks deadlock unless the strategy is adapted. To address this, we employ ordered workgroup scheduling by dynamically assigning workgroup IDs. In the order that workgroups are naturally scheduled, the first thread of each workgroup performs an atomic fetch-add on a device-wide

atomic variable, assigning itself a unique ID for the rest of execution. This ensures that adjacent workgroups are scheduled contiguously, improving waiting times and reducing the chance of deadlock.

**WebGPU Limitation: Missing Subgroup ID Support** WebGPU recently added support for subgroups. However, WGSL still doesn't expose `subgroup_id`, which restricts parallel strategies that rely on subgroup indexing. Subgroup indexing is critical in prefix-scan because each subgroup's rank determines its place in the overall block scan. Without access to this ordering, subgroups can compute local scans, but their results can't be properly assembled into a full block-wide scan. Partitioning workgroup threads into subgroup sized chunks is trivial, and can be computed like this.

```
pseudo_subgroup_id = local_id.x / workgroup_size
```

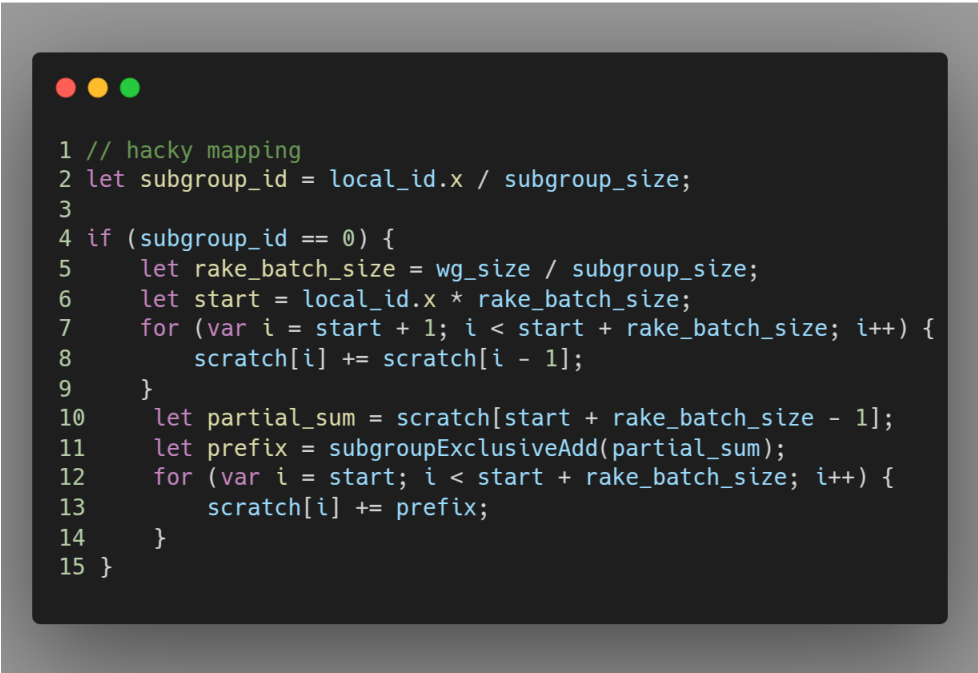
However, this construction is quite hacky, since the mapping it relies on is not guaranteed to be true. Because the WGSL compiler cannot assume that this mapping holds, it treats pseudo-subgroup threads as potentially non-uniform, which is problematic for subgroup functions. Specifically, the WGSL compiler raises a “Non Uniform Control Flow” error when only one pseudo-subgroup is ordered to execute a subgroup-level function. This is due to the fact executing subgroup functions outside of uniform control flow leads to undefined behavior. In general, any kernel using the pattern shown in Figure 2 can lead to undefined behavior triggering a “Non Uniform Control Flow” error.



```
1
2 let subgroup_id = local_id.x / subgroup_size;
3
4 if (subgroup_id < n) {
5     let x;
6     let prefix = subgroupFunction(x);
7 }
```

Figure 4.4: Undefined and disallowed subgroup behaviour.

This significantly impacts our algorithm’s ability to run on WebGPU because both the SIMD Raking block and Parallelized Decoupled Lookback execute with just one subgroup and rely on subgroup-level functions. Our implementation of SIMD Raking throws “Non Uniform Control Flow” on line 11 for this reason.



```

1 // hacky mapping
2 let subgroup_id = local_id.x / subgroup_size;
3
4 if (subgroup_id == 0) {
5     let rake_batch_size = wg_size / subgroup_size;
6     let start = local_id.x * rake_batch_size;
7     for (var i = start + 1; i < start + rake_batch_size; i++) {
8         scratch[i] += scratch[i - 1];
9     }
10    let partial_sum = scratch[start + rake_batch_size - 1];
11    let prefix = subgroupExclusiveAdd(partial_sum);
12    for (var i = start; i < start + rake_batch_size; i++) {
13        scratch[i] += prefix;
14    }
15 }

```

Figure 4.5: SIMD Raking is technically undefined in WGSL.

Fortunately, we found that all of our test devices, except those made by Intel, use the mapping `let subgroup_id = local_id.x / subgroup_size` under the hood. To work around the subgroup uniformity compiler error, we disable subgroup uniformity analysis by adding the following directive to our WebGPU setup:

```
diagnostic(off, subgroup_uniformity);
```

This enables our program to run without the compiler causing issues. Experimentally, our program works across all input sizes for all non-Intel devices. In general, this implementation works but is not guaranteed to be correct because of these limitations.



# **Chapter 5**

## **Results and Discussion**

### **5.1 Roadmap for Results and Discussion**

In this section, we investigate the performance impact of exhaustive per-input-size tuning for GPU prefix-sum. For each device, we define an Ideal Configuration Set (ICS) as the unique set of optimal configurations, one for each input size, identified through exhaustive search. We also determine a best-on-average kernel for each device: a single kernel that performs well across all input sizes, useful in scenarios constrained by storage or setup complexity. Finally, we extend both analyses to a multi-device setting, constructing the ICS and a best-on-average kernel that jointly optimize performance across multiple devices.

### **5.2 ScanBox: Initial Results**

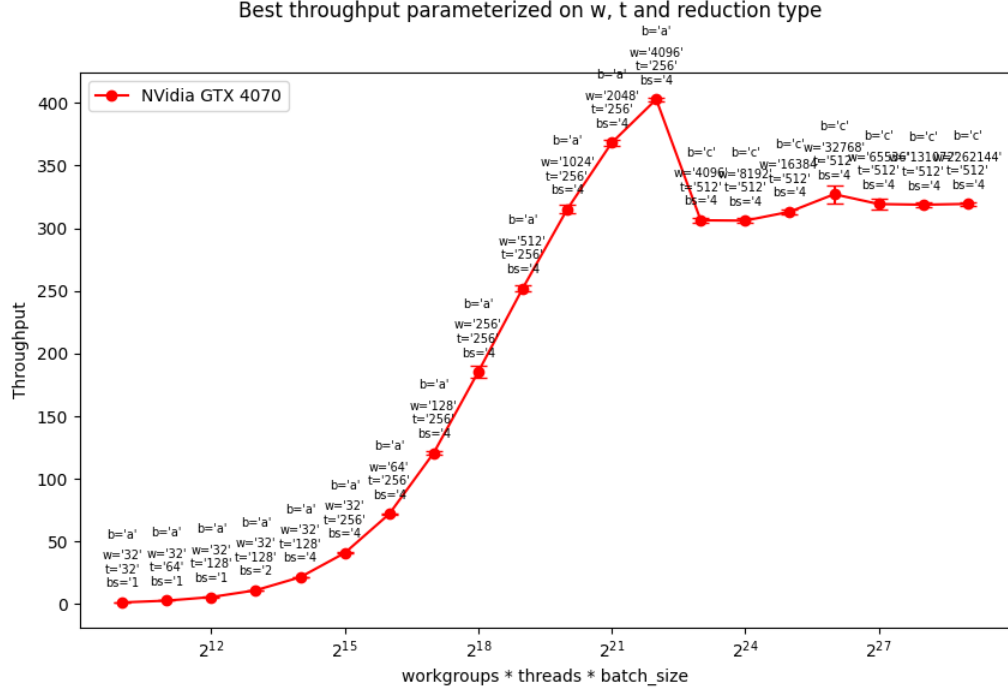


Figure 5.1: Initial Nvidia RTX 4070 prefix-sum plot (throughput in GB/s).

This throughput plot shows the performance of our Vulkan implementation on the RTX 4070. For each input size, we explore a parameter space that includes batch size, work-group size, and local reduction strategy. Parameter **b** represents block scan strategies  $a$  = SIMD Raking,  $c$  = Blelloch 1990. Parameter **w** and **t** represent workgroups and workgroup size (threads) respectively and **bs** represents batch size, i.e. how many unsigned 32 integers we load per workgroup thread. The goal of the implementation is to approach the device's throughput ceiling, which for the RTX 4070 is 504 GB/s. For each input size, we exhaustively searched all combinations of the parameters, and the plot shows the configurations that yielded the highest throughput. We do not include the throughput ceiling in this plot because the 504 GB/s limit lies beyond the plotted range. In upcoming plots, we include this ceiling for refer-

ence. However, the observed throughput does not approach this limit. Our AMD discrete was performing well but irregularly with large throughput differences between runs hence why we do not include it.

**Performance Highlights and Takeaways** Parameter  $a$ , SIMD Raking seems to be more popular in earlier sizes and then after the  $2^{22}$  elements  $c$ , Blleloch 1990 is more effective. Notably, there is a consistent dip in throughput at  $2^{22}$  elements, a pattern that appears in subsequent plots as well. This behavior likely stems from interactions between the device’s cache architecture and our implementation. Similar performance drops can also be observed, to varying degrees, in both memory copy operations and Nvidia’s CUB scan. Building on the insights gained from local scan type selection, we considered expanding the parameter space to include sequential (unparallelized) lookback. Given that different input size regimes favor different strategies, sequential lookback may also prove useful in certain cases.

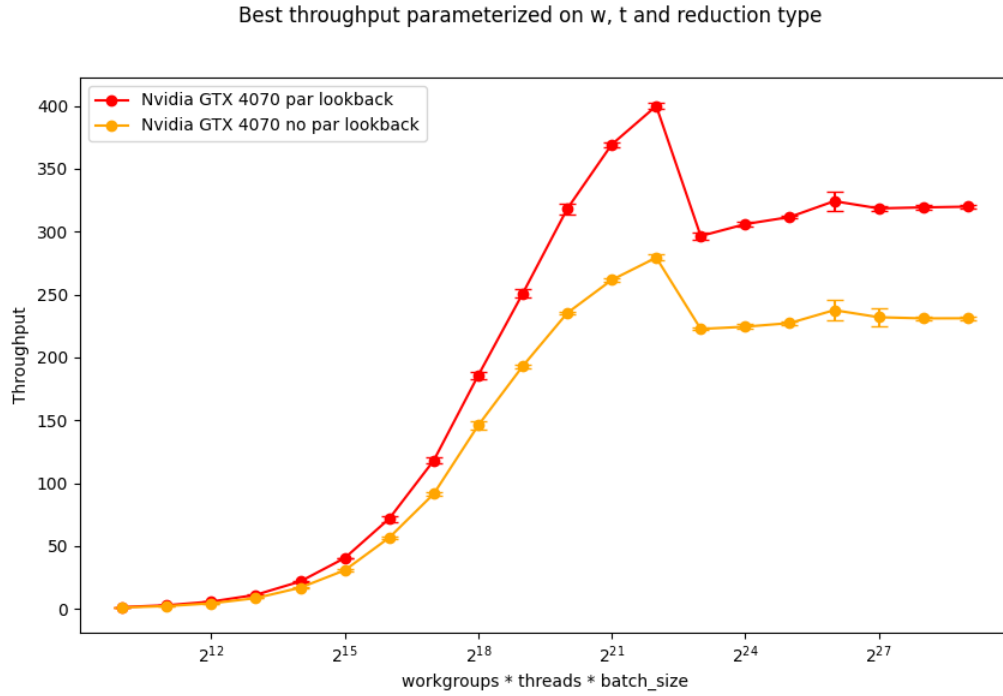


Figure 5.2: Nvidia RTX 4070 prefix-sum parametrized on Parallelized Decoupled Lookback vs. Sequential Decoupled Lookback (throughput in GB/s).

However, with this expanded parameter space, we found that performance was consistently worse across all input sizes. As a side note, this reinforces just how critical subgroup-level memory access is for performance. Parallelized lookback takes advantage of single-subgroup operations, but in WebGPU, the lack of subgroup indexing support makes this optimization both hacky and technically incorrect. After confirming that this direction was unproductive, we shifted focus toward more strategic optimizations. Since prefix-scan is a memory-bound algorithm, we hypothesized that improvements related to memory movement would be the most impactful. This led us to explore loading data types beyond the standard 32-bit unsigned integers.

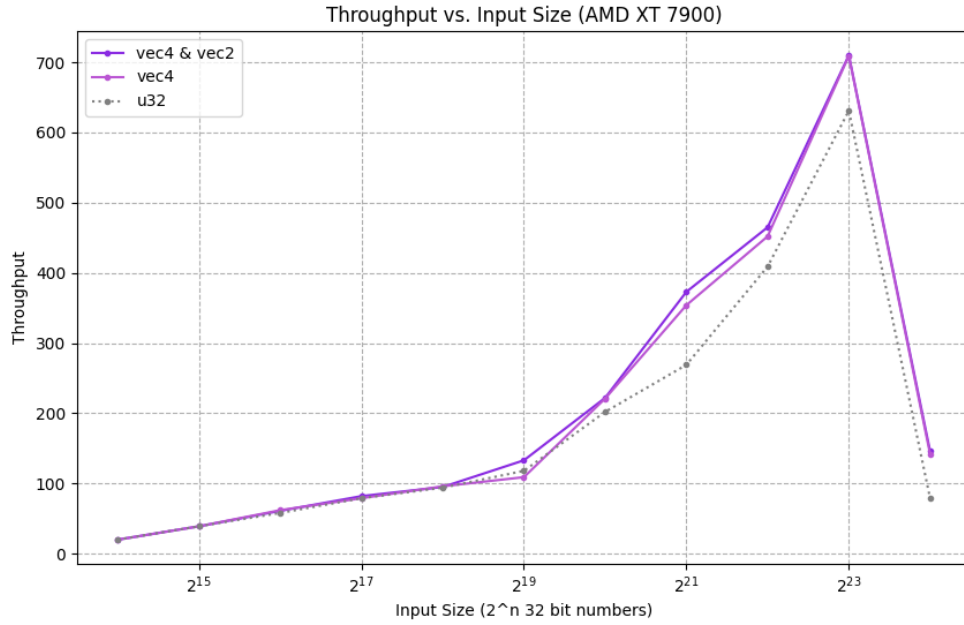
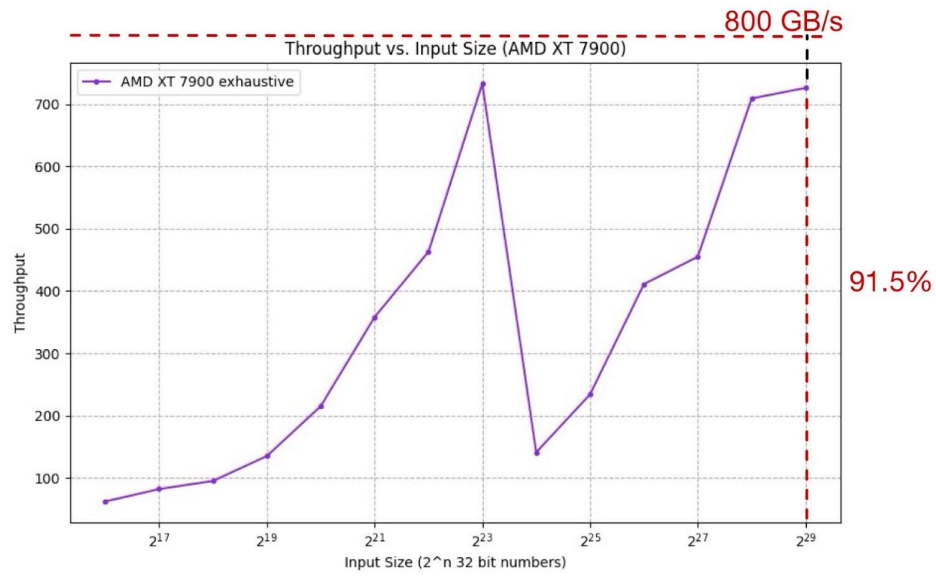


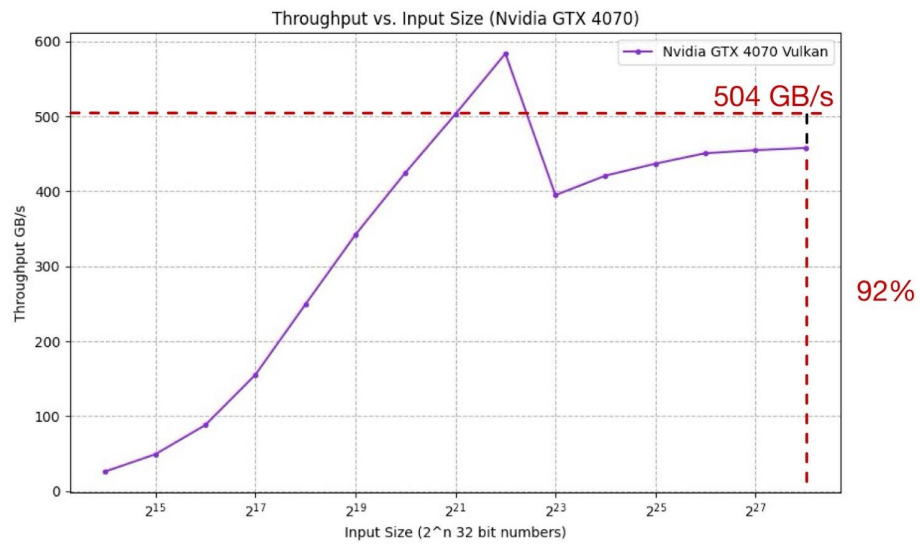
Figure 5.3: AMD XT 7900 varying input data types (throughput in GB/s).

This plot shows results from our AMD RX 7900 XT. For each input size, we exhaustively search the parameter space, including batch size, number of workgroups, workgroup size, lookback preference, and local scan type. The plot lines represent different constraints on memory type. The dotted line uses only 32-bit unsigned integers and performs worse than the colored lines, which leverage vectorized data types. The light purple line uses vec4, while the dark purple line is parameterized over both vec2 and vec4. These results demonstrate that switching between vector types exceeds unsigned 32 bit integers in throughput performance at these input sizes. At  $2^{23}$  elements, we observed up to a 80 GB/s improvement when using vector types compared to non-vectorized implementations. These initial results motivated us to incorporate vector types into our parameter space.

### **5.3 ScanBox: Fully Tuned Performance Results**



(a) AMD XT 7900 exhaustive per input size tuning — Throughput in GB/s.



(b) Nvidia RTX 4070 exhaustive per input size tuning — Throughput in GB/s.

The inclusion of vector data types in our tuning parameters enabled ScanBox (Vulkan) to achieve peak throughputs of 731 GB/s on the AMD GPU (91.5% of theoretical memory bandwidth) and 463 GB/s on the Nvidia GPU (92%) at large input sizes. Compared to Nvidia’s proprietary CUB library, our approach delivers performance that is on par with or better than CUB at the majority of input sizes.

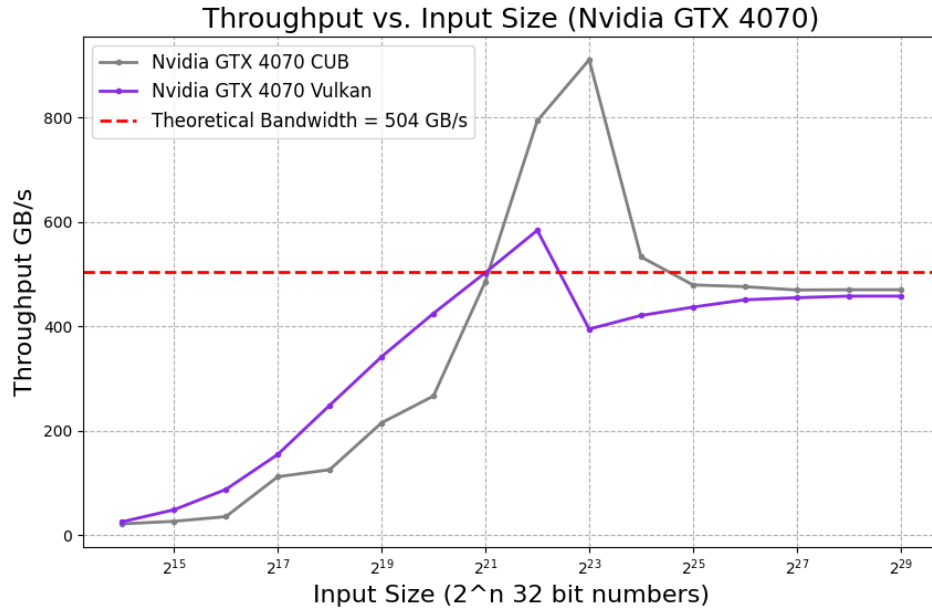


Figure 5.5: Nvidia RTX 4070 Vulkan vs. CUB.

**ScanBox vs. CUB: Cross-Platform Performance Comparison** We divide the plot into three regions: small, medium, and large input sizes, and use the theoretical bandwidth limit as a reference point to assess how well each implementation utilizes available hardware across devices. At small input sizes ( $2^{15}$  to  $2^{21}$ ), ScanBox consistently outperforms Nvidia’s CUB implementation by up to 43%. At large input sizes ( $2^{27}$  to  $2^{29}$ ), ScanBox and CUB are nearly tied on



Nvidia hardware, with CUB holding a slight 1–2% advantage. Notably, when normalized to theoretical bandwidth, this level of efficiency is maintained across vendors: on AMD GPUs, ScanBox reaches within 2–3% of Nvidia CUB’s relative performance. This demonstrates both strong cross-platform portability and competitive performance.

Device	Implementation	Percentage of Bandwidth	Input Size
Nvidia RTX 4070	CUB (CUDA)	93%	$2^{29}$
Nvidia RTX 4070	ScanBox (Vulkan)	92%	$2^{29}$
AMD XT 7900	ScanBox (Vulkan)	91.5%	$2^{29}$

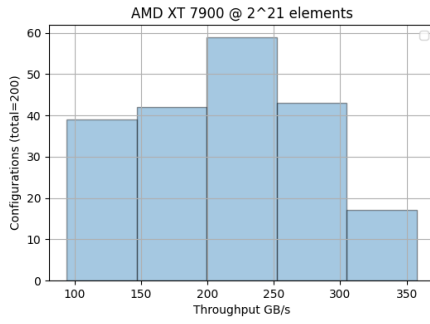
Table 5.1: Percentage of memory bandwidth achieved compared to the device ceiling.

However, at medium sizes ( $2^{22}$  and  $2^{23}$ ), CUB outperforms our method by up to 125%. We suspect this is due to optimizations in CUB that more effectively utilize global memory caching. Since CUB only exceeds our performance in this cache-dominant regime, where observed throughput exceeds the theoretical memory bandwidth limit, its advantage is likely driven by more efficient use of GPU caches.

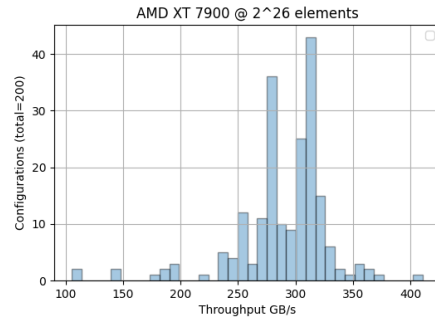
## 5.4 Performance Variability: The Need for Input-Specific Tuning

In the implementations above, we heavily rely on per-input-size tuning, which can be time-consuming to perform. But is this effort justified? Does per-input-size tuning offer significant benefits over device-level tuning? To investigate this, we visualize the distribution of parameter configurations for each input size. Using the AMD RX 7900 XT as a representative device, the figures below show how many configurations fall within specific throughput ranges for each of fifteen input sizes. The six distributions presented here represent the most common

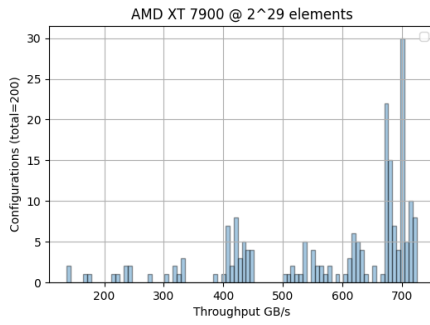
patterns observed.



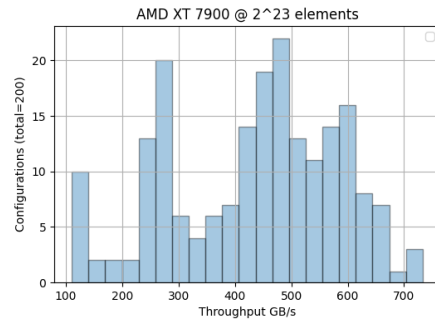
(a) input size:  $2^{21}$



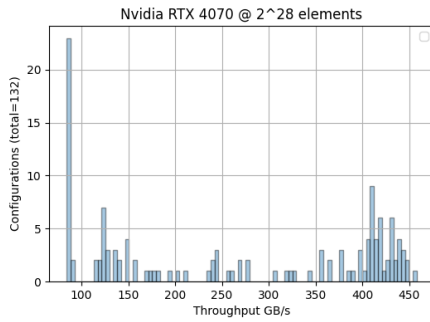
(b) input size:  $2^{26}$



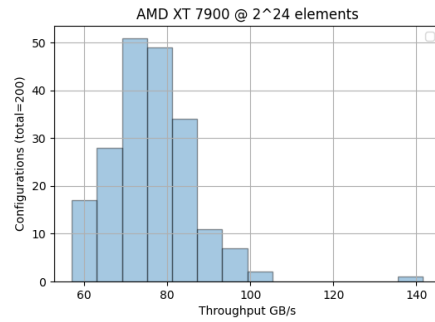
(c) input size:  $2^{29}$



(d) input size:  $2^{23}$



(e) input size:  $2^{28}$



(f) input size:  $2^{24}$

Figure 5.6: Subset of AMD XT 7900's throughput density distributions plots.

**Calculating Throughput Density Bin Sizes** Before we get into the details, to calculate throughput density bin size we use the Shimazaki Shinomoto function which tries to choose the best number of bins  $k$  by minimizing a cost function that balances bias and variance. It’s great for irregular, skewed, or multi-peaked (multimodal) data such as these configuration density plots [52].

**Interpreting Throughput Density Plots** These plots illustrate where the majority of configurations exist. Plots 1, 2, 4 are mostly normally distributed with the majority of performances being mediocre, however plots 2 and 4 are especially sparse in the high performance regions. Plot 3 contains mostly good configurations, with some falling into the moderate performance range. Plot 5 is characterized by a large number of both very poor and very good configurations, while plot 6 is dominated by mostly poor-performing configurations. A device can have roughly 3,500 different parameter configurations, with over 200 applicable to any given input size. Using a systematic framework to explore these variable spaces speeds up development and effectively enumerates options. If the goal is purely to create the most performant algorithm, an input-aware scan implementation will achieve that. However, in environments with memory or processing constraints, what are the practical value of these insights?

## 5.5 Leveraging Per-Input Tuning in Resource-Limited Settings

In a realistic deployment, it may not be practical to use an input-aware kernel or to store dozens of specialized variants. For example, in an embedded a setting where there’s only space, or code complexity budget, for a single kernel. Given the variability in parameter

space and the fact that good configurations can be sparse or input-size dependent, the question becomes: how do we choose the best single kernel?

**Formalizing The Problem** To answer this question, we start by clarifying some assumptions.

In our implementation, a kernel refers to a fixed combination of parameters including batch size, lookback type, memory type, and block scan strategy. These parameters are hardcoded directly into the kernel source code. By contrast, a configuration includes all kernel parameters plus the workgroup size and the number of workgroups. Workgroup size and number of workgroups are set in the pre-kernel boilerplate and usually vary at runtime to control the total input size of the GPU workload. In other words, multiple configurations can exist for a single kernel, each corresponding to different workgroup sizes and counts.

**Pruning The Configuration Space** For this reason the first step in our algorithm is to prune the configurations of a kernel that are lower in throughput than another configuration at the same input size. This way each kernel has one configuration per input size and its the best configuration, with respect to throughput.

**Finding The Kernel of Best Throughput** After this step, we can construct a throughput curve that represents the kernel's optimal performance across the input range. To pick the best performing kernel among our options, the most straightforward approach is to look for the kernel line that maximizes total throughput across all input sizes. We do this by summing throughput values. Since we only include input sizes that have results for every kernel, averaging this sum isn't necessary. We filter kernels that don't have a configuration at every input size because if

we didn't we'd end up unfairly favoring kernels missing from low-throughput regions even if we averaged the sum after. This is because the number of input sizes is relatively small and the throughput range is wide enough that an average would result in a skewed kernel score. We also considered interpolating data that fit the distribution, but that could add noise without giving us much real insight. In practice, we care most about larger input sizes anyway, those that get closest to saturating memory bandwidth, so this approach ends up being effective for our purposes.

**Aside: Why Kernel Selection Isn't Redundant with ICS** One might ask: if we still have to tune by workgroup size and workgroup count, how is this different from ICS tuning, i.e. fully exhaustive tuning? The key distinction lies in implementation complexity. To fully support an ICS, one must either dispatch from a large set of specialized kernels, incurring significant memory overhead, or embed extensive conditional logic within a single GPU kernel, which is notoriously difficult to do efficiently. By contrast, selecting a single kernel and tuning only the workgroup size and count pushes the variability into the pre-kernel boilerplate, where conditional logic is simpler and does not impact performance nearly as severely.

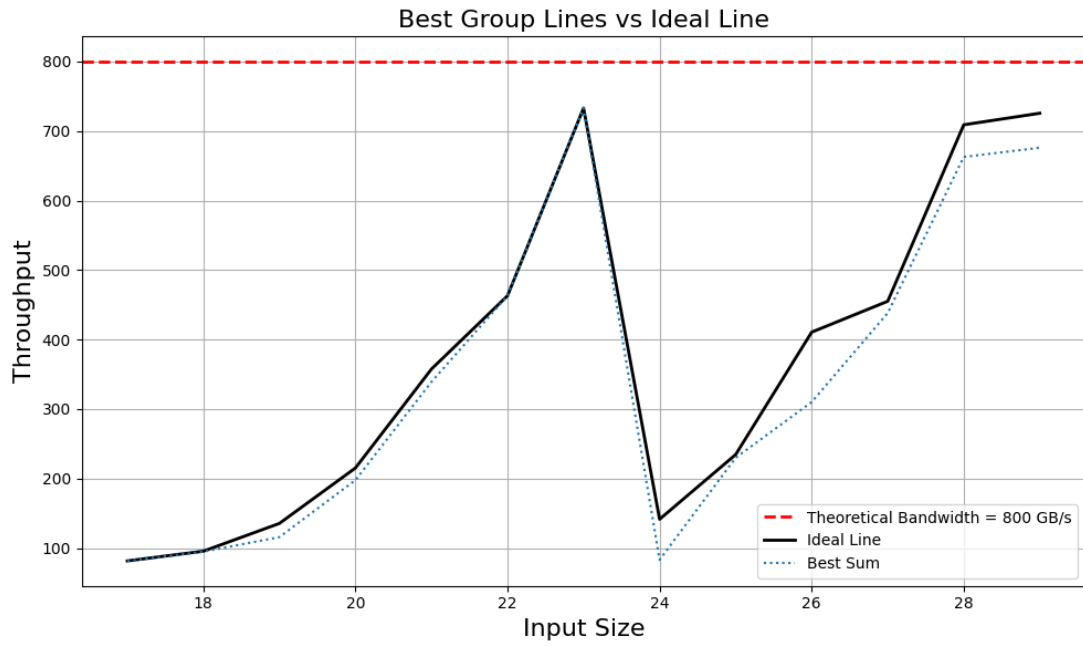


Figure 5.7: AMD XT 7900 best overall parameters vs. best single kernel that maximizes the sum of the throughputs at every input size — Throughput in GB/s.

**Maximizing Throughput: AMD** The blue dotted line represents the highest total throughput total out of every kernel compared to every other kernel. This strategy is effective however we observe significant drops in throughput at  $2^{24}$ ,  $2^{26}$  and  $2^{28}$  and  $2^{29}$  elements.

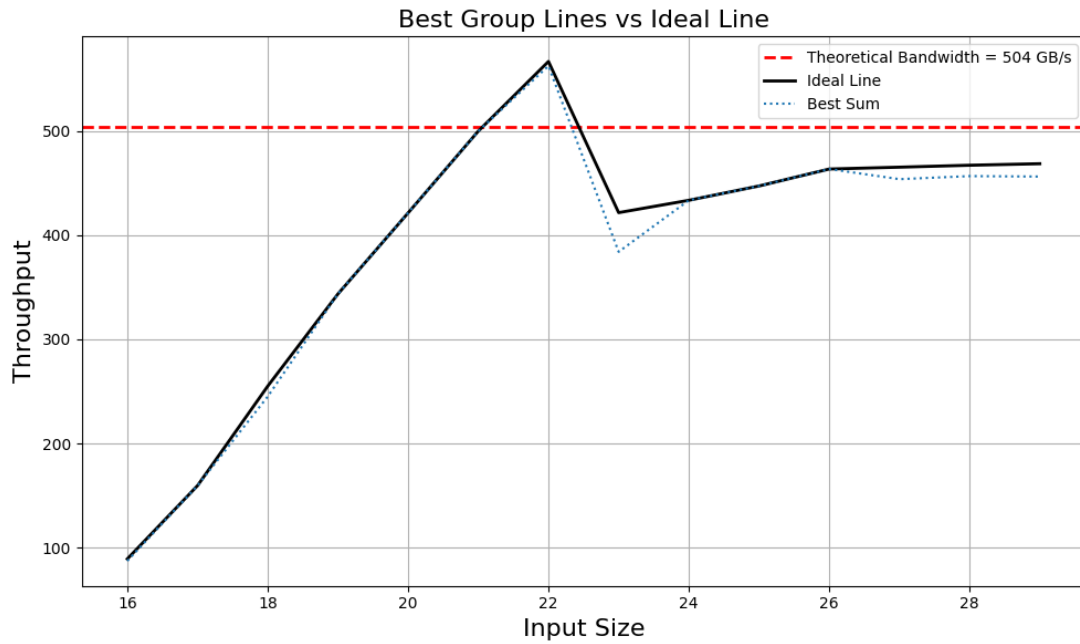
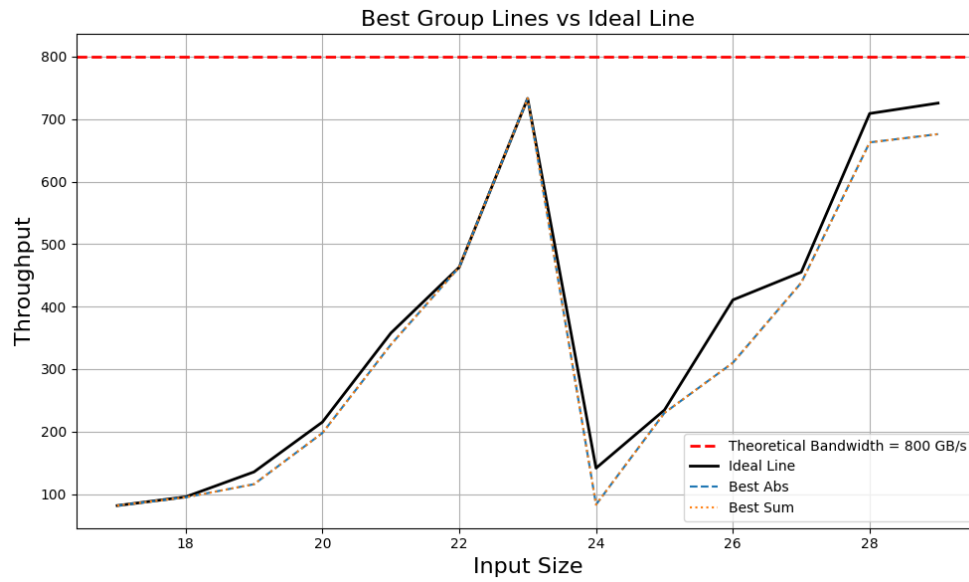


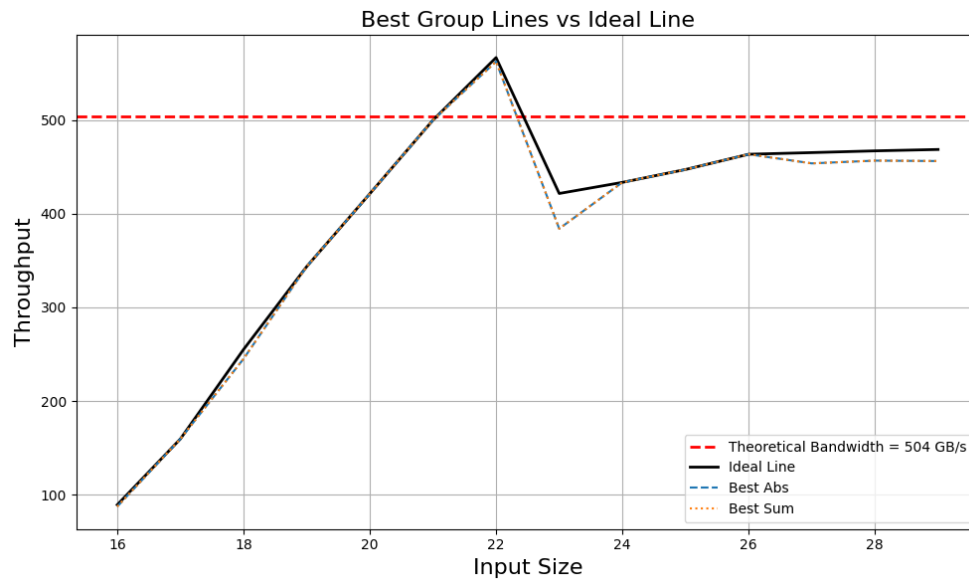
Figure 5.8: Nvidia RTX 4070 best overall parameters vs. best single kernel that maximizes the sum of the throughputs (GB/s) at every input size.

**Maximizing Throughput: Nvidia** For Nvidia there are mostly good options except that we observe a large drop at  $2^{23}$  and minor fall off at input sizes greater than  $2^{26}$ . We can attempt to smooth out these differences by penalizing the space between the lines rather than maximizing throughput directly. Lets minimize the area between the ideal line and our kernel line. Minimizing this area better describes our goal anyway. Additionally, since the graph is discrete, minimizing the sum between each datapoint is equivalent to minimizing the area between two curves. As a side, AMD devices' starting input size is higher because their minimum subgroup size is twice as large as Nvidia's causing there to be less overall kernels.



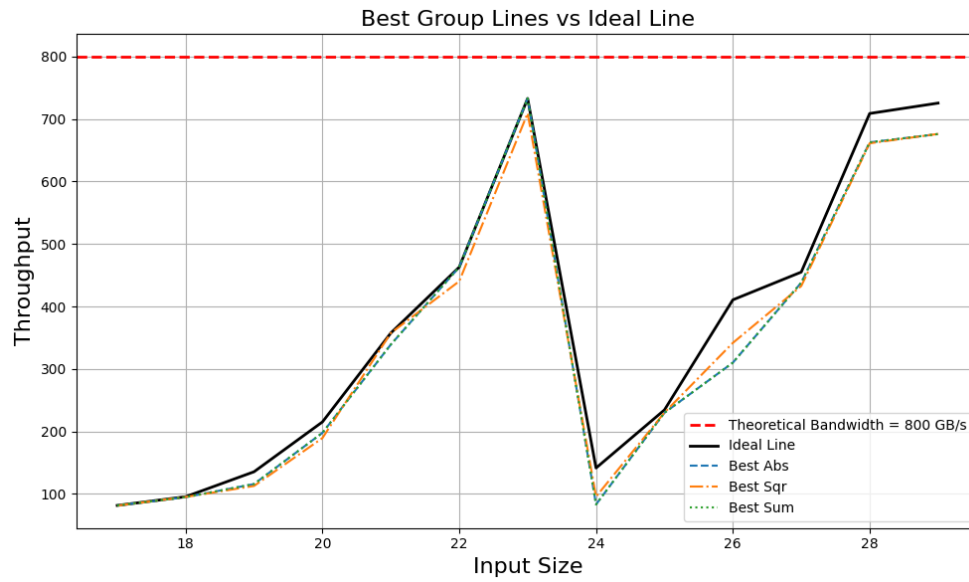


(a) AMD XT 7900 vs. kernel that minimizes area between curves — Throughput in GB/s.

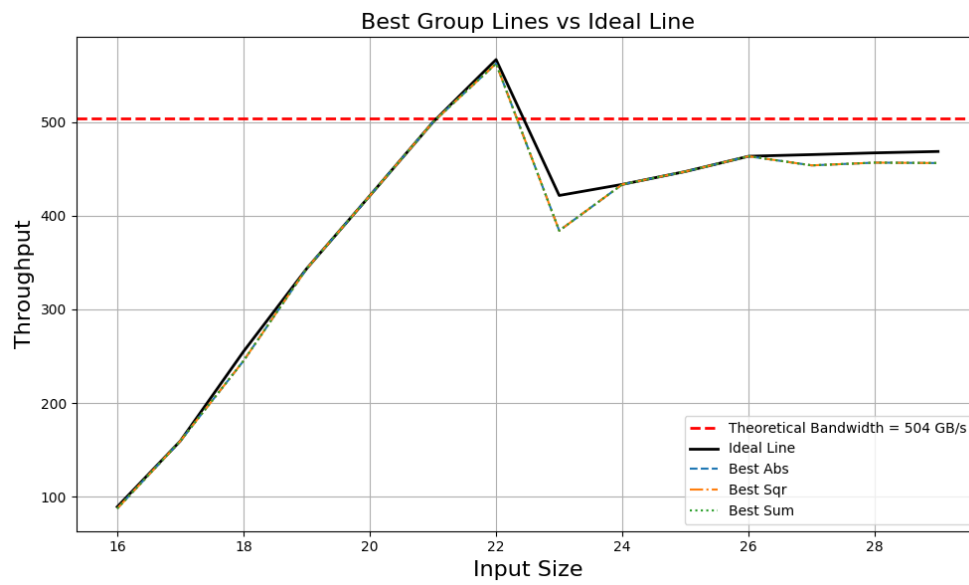


(b) Nvidia RTX 4070 vs. kernel that minimizes area between curves — Throughput in GB/s.

**Impact of the Absolute Distance Metric** For both plots we minimize the absolute distance. Unfortunately, this doesn't have any further effect compared to maximizing the sum. To further increase regularization and penalize the distance between points we employ squared distance as our metric to amplify large differences.



(a) AMD XT 7900 vs. kernel that minimizes squared distance between points —  
Throughput in GB/s.



(b) Nvidia RTX 4070 vs. kernel that minimizes squared distance between points —  
Throughput in GB/s.

**Impact of the Squared Distance Metric** On Nvidia, there is no meaningful difference between minimizing the absolute sum and the squared sum when selecting a best-on-average kernel. However, on AMD, input sizes  $2^{21}$ ,  $2^{24}$ , and  $2^{26}$  show improved performance when minimizing the squared difference rather than the absolute one. At  $2^{22}$  and  $2^{23}$ , small performance drops are observed when using the squared-distance-optimized kernel compared to the ideal configuration. Interestingly, both best-on-average kernels use Parallelized Lookback and SIMD Raking, but differ in other parameters: the squared-sum kernel increases the batch size from 4 to 8 elements and switches the data type from `vec4` to `vec2`.

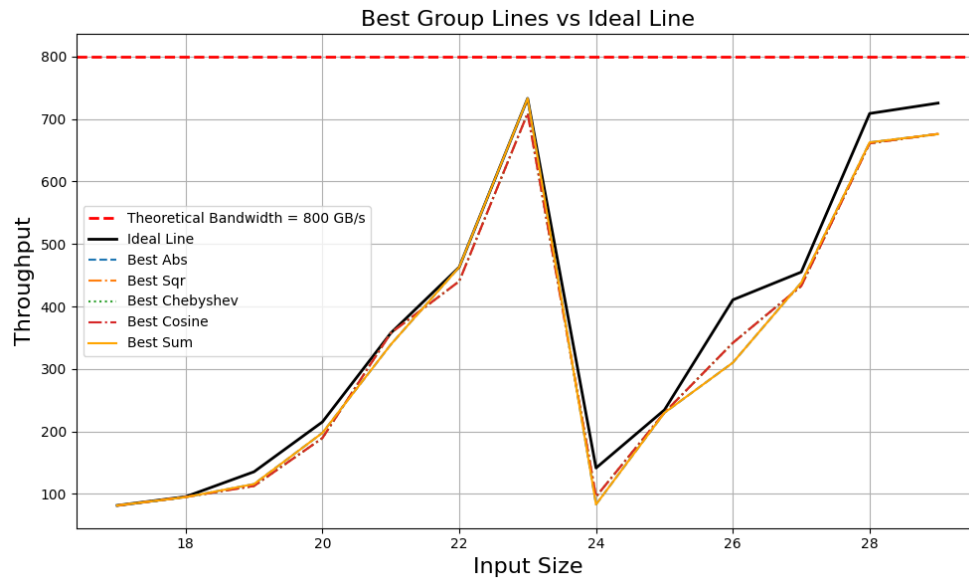
**Divergence Between Best Kernel and ICS at Large Input Sizes** One of the clearest areas where the *best kernels* diverge from the ICS is in the large input size regime on both AMD and Nvidia. In the ICS, we observe a noticeable shift in parameters trends after medium input sizes, where the throughput “mountain” turns into a dip. Specifically, the lookback strategy changes from Parallelized Decoupled Lookback to Sequential Decoupled Lookback.

**Cache-Aware Behavior in Parallelized Lookback** This trend likely stems from cache behavior. At smaller input sizes, when the block descriptors still fit in cache, subgroup threads in Parallelized Decoupled Lookback can benefit from spatial locality. When one thread loads a cache line containing adjacent block descriptors, other threads in the subgroup, who are loading adjacent blocks, can reuse that data, leading to just a single actual global memory access per subgroup Lookback and scan for majority of lookbacks.

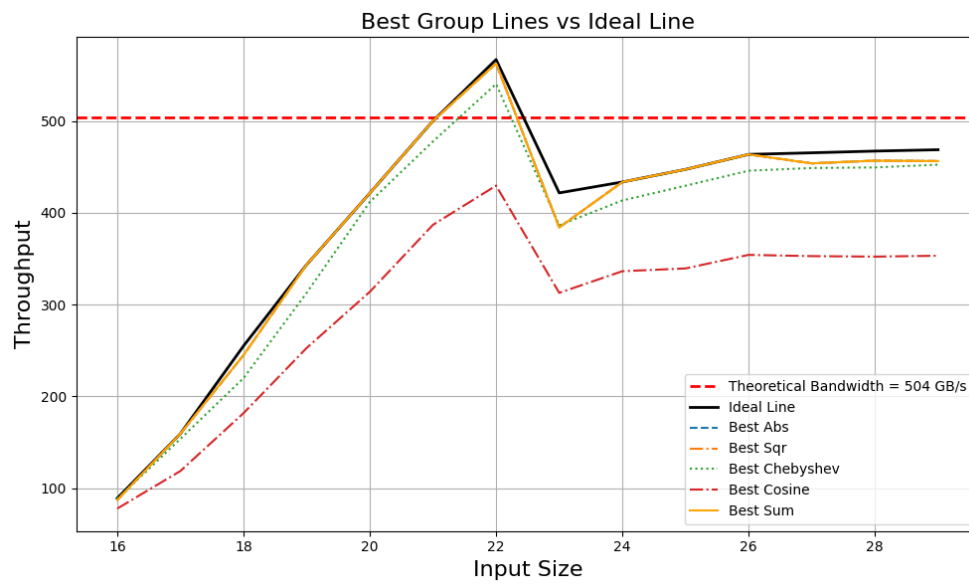
**Global Memory Bottlenecks Undermine Parallelism** However, as input size grows and saturates the cache, this locality advantage disappears. Each subgroup thread must load a new cache line from global memory, and these lines are constantly evicted. This eliminates the cache reuse benefit and subjects every thread to the high latency of global memory.

**Why Sequential Lookback Outperforms in the Memory-Bound Regime** At this point, Sequential Decoupled Lookback becomes competitive. Although threads perform lookbacks serially in the Sequential Decoupled strategy, the high latency of global memory accesses effectively hides the cost of serial computation. In practice, both strategies end up loading block descriptors in a mostly serialized fashion due to memory latency, so the advantage of parallel arithmetic is diminished. Worse, in Parallelized Lookback, all subgroup threads must acquire a `READY` status flag before proceeding with computation. If even a single thread encounters an `UNREADY` flag, the remaining threads, up to `subgroup_size - 1` of them, may have already issued a costly global memory load, only to discard the reduction without doing any useful work. By contrast, Sequential Lookback issues only one memory load at a time, at most one wasted access per iteration, making it more efficient under memory-bound conditions.

**Other Distance Metrics** We also experimented with cosine and Chebyshev distance metrics to address Nvidia’s throughput dip at  $2^{23}$ . However, these alternatives tended to over-regularize the kernel, leading to degraded performance, especially on the RTX 4070, which showed little responsiveness even to the squared-distance metric.



(a) AMD XT 7900 vs. kernel that minimizes Chebyshev and cosine distance between points.



(b) Nvidia RTX 4070 vs. kernel that minimizes Chebyshev and cosine distance between points.

To reiterate our findings. Per input size tuning can help with more than just high performance kernels but can be helpful in specialized scenarios like best-on-average kernels.

## 5.6 ICS for Multiple Devices

Per input size tuning does not carry to other devices.

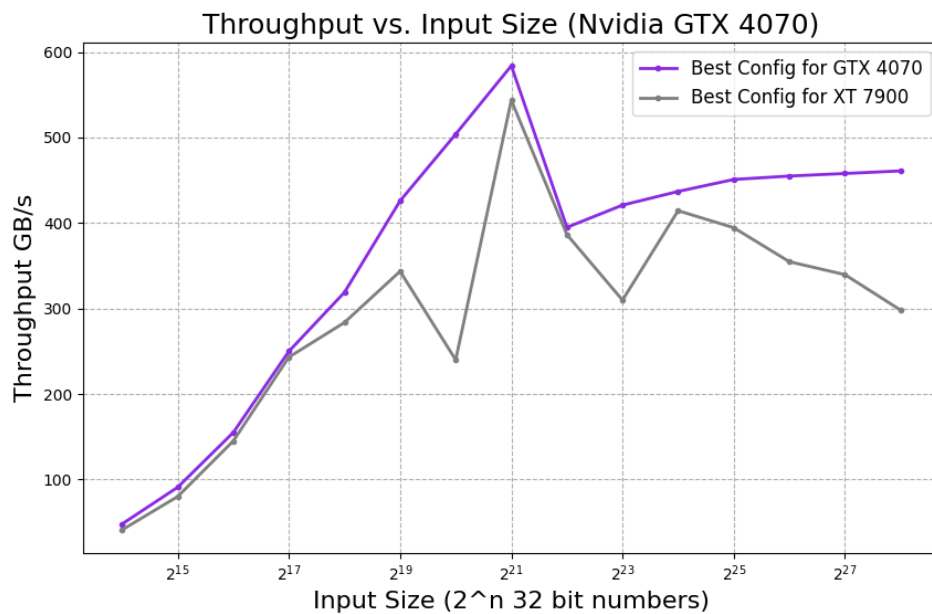


Figure 5.12: Benchmarked on the Nvidia RTX 4070: The best RTX 4070 parameters vs. The best XT 7900 parameters — Throughput in GB/s.

This figure compares the ideal configuration set (ICS), the best per-input-size parameter combinations found via exhaustive tuning, of both the AMD RX 7900 XT and the Nvidia RTX 4070. To highlight the lack of performance portability, all throughput measurements are taken on the RTX 4070. As expected, the ICS tuned for AMD performs poorly on Nvidia

hardware, despite being tuned for input size. This illustrates a key concept, that configurations optimized for one architecture do not necessarily transfer to another. Given this mismatch, we ask: is it possible to construct an ICS that performs well across both devices?

**Initial Approach: Highest Shared Throughput** To identify a shared ICS across devices, we compute the configuration that achieves the highest average throughput at each input size. Since raw throughput varies significantly between devices, we first normalize the throughput values (between 0 and 1) before averaging. This ensures that both devices contribute equally to the selection process, improving fairness.

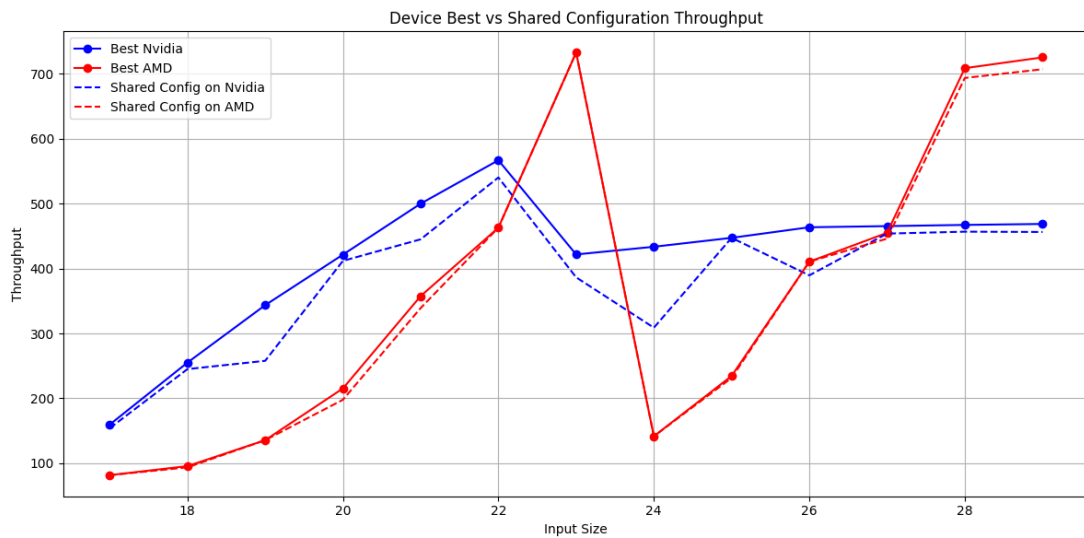


Figure 5.13: Plots the shared ICS for both devices (dotted line), using averaging, against device local ICSs on the AMD XT 7900 and the Nvidia RTX 4070 — Throughput in GB/s

The dotted lines are the performances of the shared configs. This works alright some averages are better at certain inputs and worse at others. There are a significant amount of dips



for the Nvidia device, so let's try to prune these dips by choosing the configuration that has the maximum throughput at that input size on the two devices.

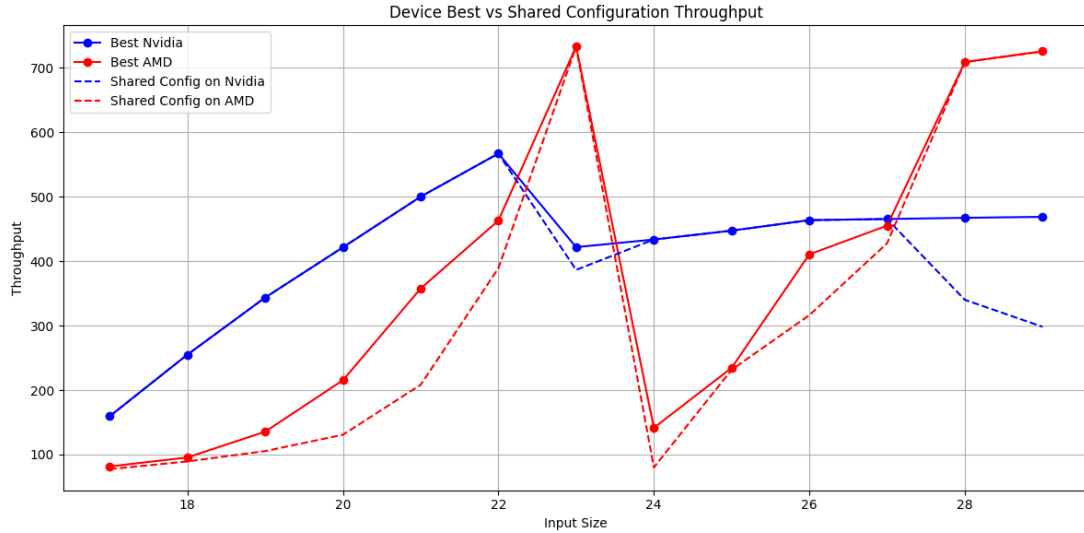


Figure 5.14: Plots the shared ICS for both devices (dotted line), using max, against device local ICSs on the AMD XT 7900 and the Nvidia RTX 4070

**Choosing Shared Configurations by Maximizing Throughput** This is pretty bad and seems to prefer Nvidia over AMD now. We really want to minimize difference between these values to compute a fair shared ICS. Instead of selecting the configuration with the highest mean throughput alone, let's introduce a penalty based on variance: we subtract the standard deviation across devices from the mean. This adjusted score favors configurations that perform consistently across devices, reducing bias toward one architecture and promoting fairness in the shared ICS.

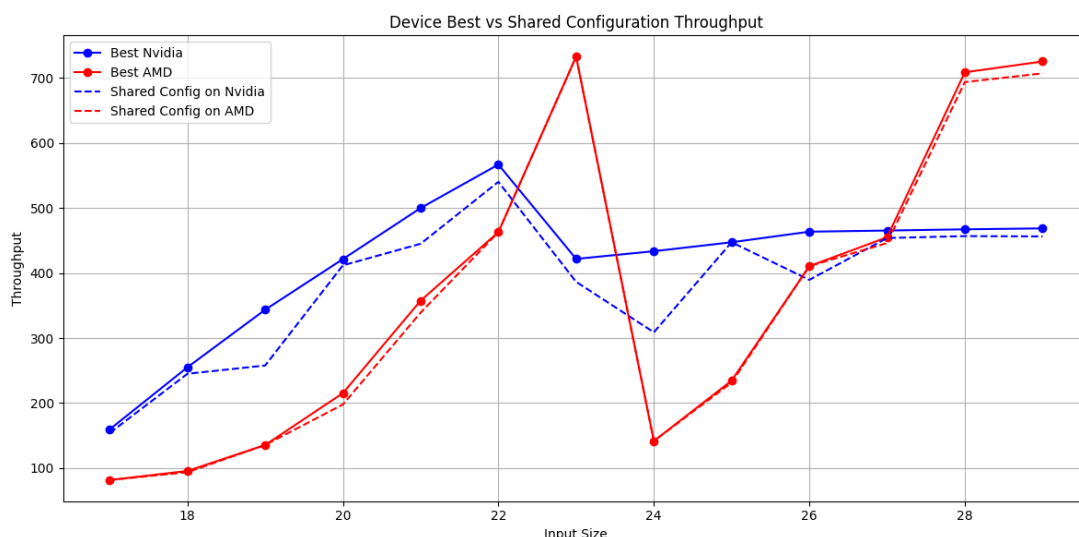
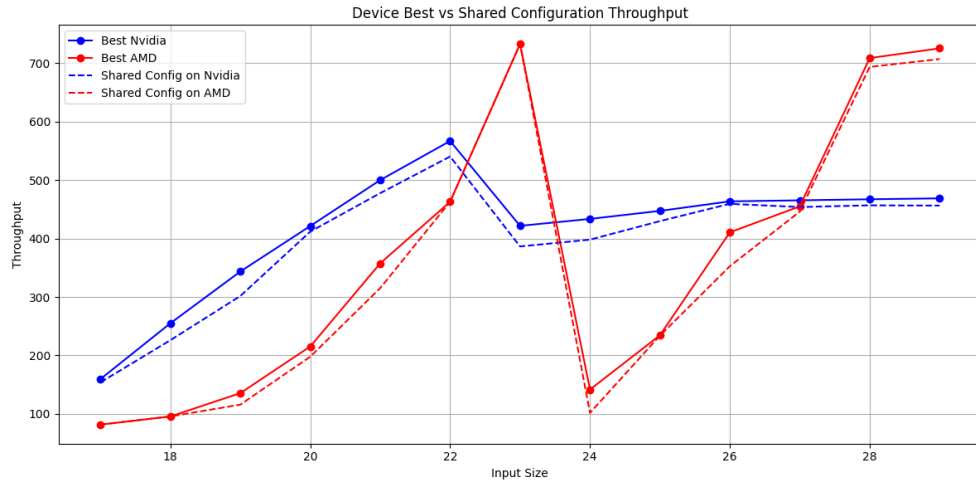


Figure 5.15: Same plot using averaging with variance penalty — Throughput in GB/s

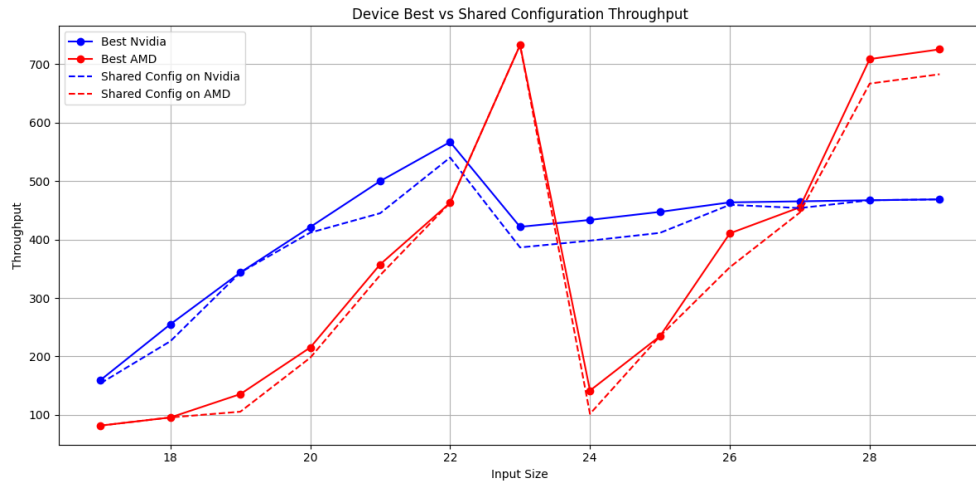
**Choosing Shared Configurations by Relative Rank** This is entirely indifferent from the normal mean so we continue on to our next strategy. Our second approach was spawned out of a realization that normalizing throughputs between to zero and one reduces scale intolerance but only if these devices follow the same distribution which they do not. Normalization improves scale intolerance for averaging but doesn't solve the problem of have a totally different distribution. Averaging, i.e., splitting the gap between configuration performances will not work here. What we really want is for datapoints to be best good relative to their own best lines, this is the actual cieling of their performance. We need a metric that is sensitive to its own score within the device. This sounds like we want the want the best ranked configs within a given device.

**Ranking Metrics** There exist a few metrics for finding the best combined rank across the device. The first one we use is minimizing the sum of their ranks. Summing ranks results in

a simple combined “score” reflecting how well a configuration performs relative to others on both devices simultaneously. The lower the sum, the better the config ranks overall. This favors configurations that are not just good on one device, but also reasonably good on the other. We also including the minimizing the product of ranks which penalizes rank differences even more than minimizing the sum.

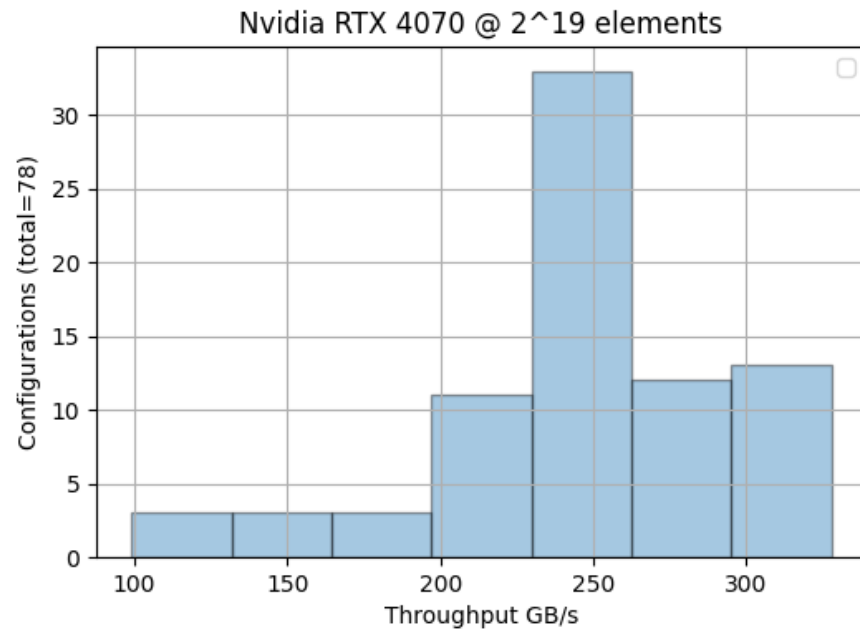


(a) Minimizing Sum of Ranks — Throughput in GB/s.

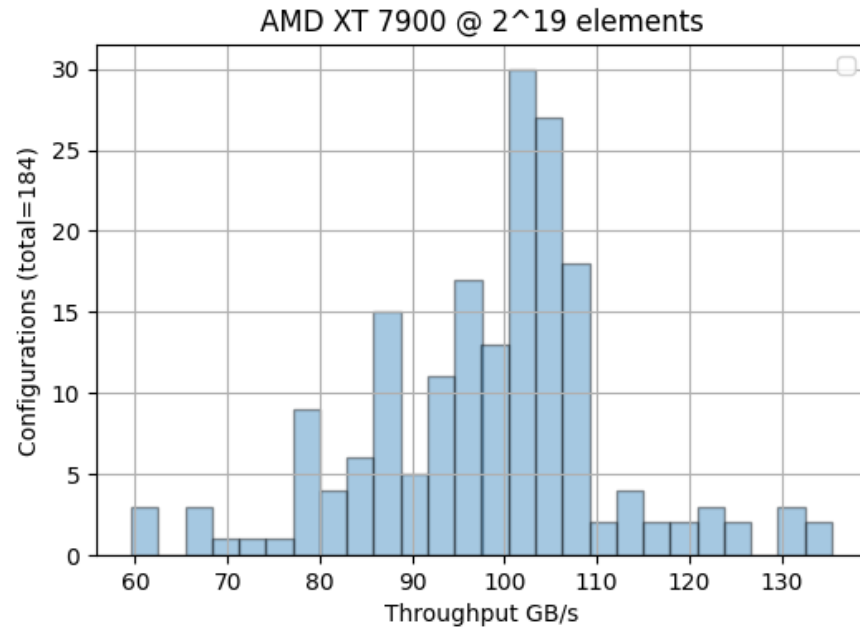


(b) Minimizing Product of Ranks — Throughput in GB/s.

**Rank Based Metric Limitations** This approach yields smoother results, its clear we are on the right path, but rank-based selection remains sensitive to threshold effects. For example, at input sizes  $2^{28}$  and  $2^{19}$ , minimizing the sum of ranks makes a slightly fairer choice (see dotted lines), but at  $2^{19}$ , minimizing the product of ranks yields a better decision. This inconsistency stems from a key limitation: rank-based metrics are insensitive to throughput magnitudes. The configuration density plots confirm this issue. At  $2^{19}$ , AMD's best configuration achieves 110 GB/s and Nvidia's hits 200 GB/s, but both fall within sparsely populated throughput regions.



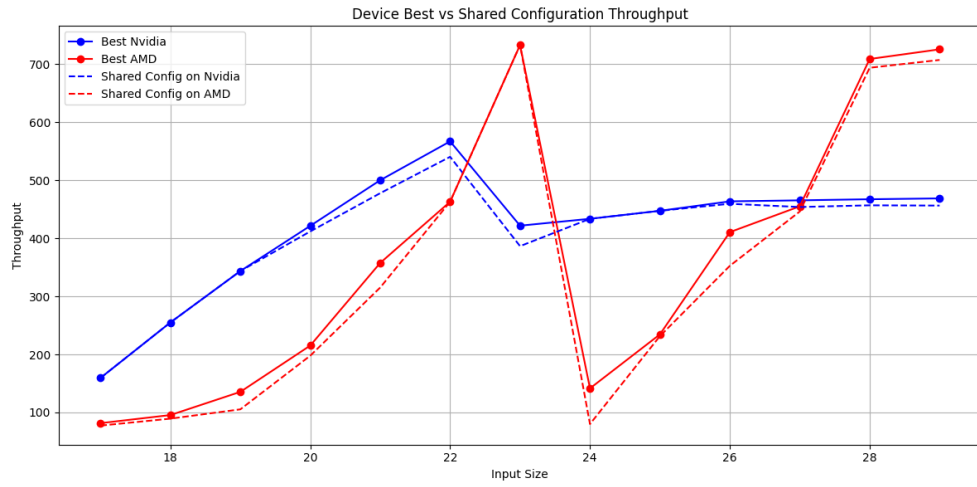
(a) Dip in density at 200 GB/s.



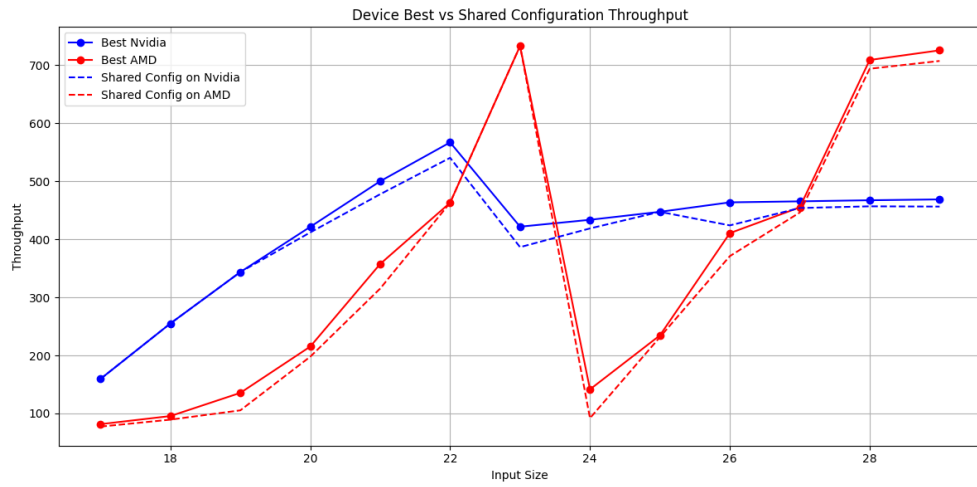
(b) Dip in density at 110 GB/s

**Why Ranking Fails in Sparse Configuration Regions** In such sparse regions, adjacent ranks can differ by large absolute throughput margins, making rank a poor proxy for performance similarity. Ranking tells us only the order, not the cost of deviation. The product of ranks method punishes large rank disparities, but it is overly sensitive and still blind to how much throughput is lost between ranks. To improve fairness, we need a metric that accounts for both relative order and throughput distance.

**The Distance Approach** Instead of ranking we need something that is sensitive to magnitude. Instead of ranking configurations we can compute the distance between each datapoint and its device ideal line like we did in the first section, then sum the two distances which will naturally rank the configurations while preserving their relative magnitudes. Ranking this way solves the threshold problem since ranks are not contiguous but spread by the strength of their difference. Similar to product of ranks we can use the squared distance metric to penalize larger gaps.



(a) Minimizing Distance — Throughput in GB/s.



(b) Minimizing Squared Distance — Throughput in GB/s.

This helps with that process but might be too strong. At  $2^{24}$  product of sums makes the shared lines more even but at  $2^{26}$  overly punishes nvidia. We can tune this slightly and instead of squaring the distance use 1.25 as the exponent.

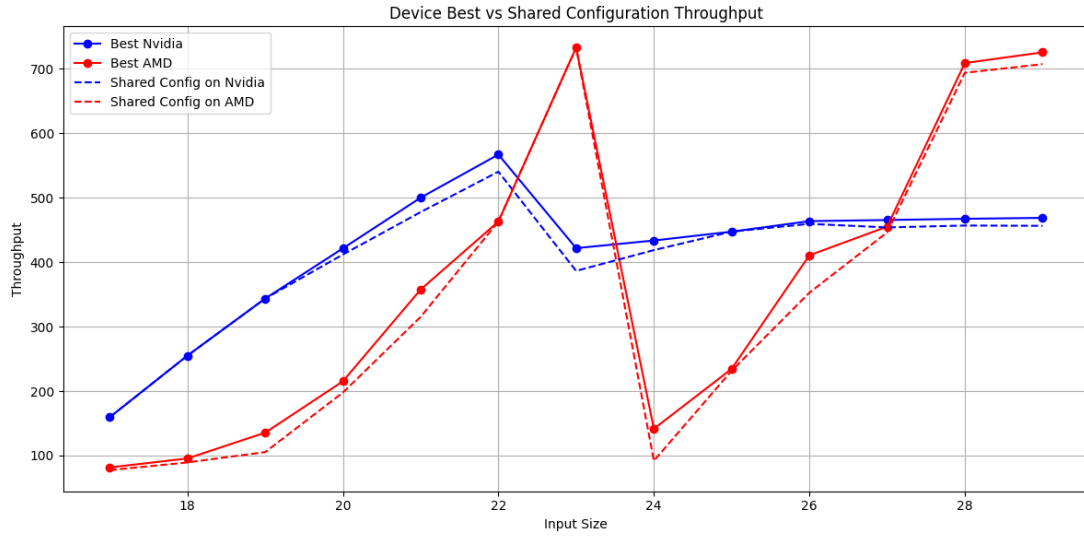


Figure 5.19: Minimizing Distance to the 1.25 power — Throughput in GB/s.

This set prevents the over punishment at  $2^{26}$  while still smoothing  $2^{24}$ . Although any of these three metrics result generate good results.

## 5.7 Final Problem: Generate the Best Kernel for Multiple Devices

Now that we have found a method for comparing devices let's try to solve a more difficult and traditional problem in performance portability. Finding the kernel across multiple devices. This is similar to the problem from section 1 but now with a focus on portability not just single device performance. We have compared kernels before within a single device. Now we want to compare kernel configurations across devices. We now understand the metrics to



both compare kernels within a device and to compare performance of configurations across devices. Lets combine these tactics to find the best standalone kernel across multiple devices. For starters, we don't want to compare these kernels against the best device local kernel because this may not be indicative of the best global kernel, its possible there exists a better kernel across the devices where one input size is higher than the device local kernel though the device local kernel is still on average better locally. Its a safety precaution to choose the actual ceiling so the distance metrics dont get confused when a value exceeds the ceiling. Below, we combine insights from both standalone kernel tuning and cross-device comparisons to develop an algorithm that produces the fairest and most performant standalone kernel for optimized performance across devices.

1. **Group the data** by kernel ID. Each kernel is identified by a tuple of configuration parameters: `[batch_size, local_scan, lookback_type, data_type]`.
2. **Compute best throughput per input size and device.** For each kernel and each input size, find the maximum throughput achieved on each device (sweeping across different workgroup sizes and numbers of workgroups dispatched). This results in a throughput "line" per kernel per device.
3. **Compute distance from the ideal line.** For each kernel, calculate the sum of squared differences between its throughput line and the device's ideal line (the maximum achievable throughput per input size), for both devices. These distances reflect how closely a kernel performs to the best possible.
4. **Aggregate distances across devices.** Add the summed squared distances from both de-

vices to get a single scalar representing the overall deviation from ideal performance.

5. **Select the best kernel.** Identify the kernel with the smallest total deviation. This kernel minimizes the combined performance gap across all input sizes and both target devices.

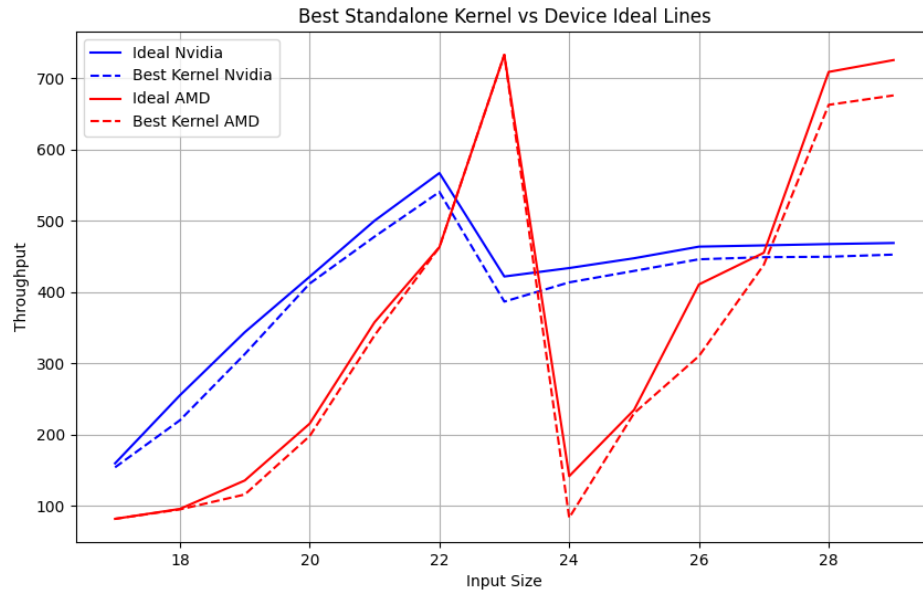


Figure 5.20: Best kernel across AMD and Nvidia — Throughput in GB/s.

This algorithm produces an exceptionally fair and performant portable kernel across devices. Identifying a single performant kernel like this is a challenging problem, especially valuable in scenarios where hardware varies and there are constraints on memory, compute resources, or implementation complexity. Mobile devices, for example vary in hardware and operate under tighter space and resource constraints compared to other platforms. That said, with respect to prefix sum, there appears to be a decent degree of inherent performance portability on these devices. Further testing across a wider range of chips may uncover greater

variation. Still, incorporating this technique into the development process for phones and embedded systems appears to be a promising strategy for portable and efficient GPU kernel design.

## Chapter 6

### Future Work

**A Mountain of "Cache"** One of the most notable patterns in the throughput vs. input size plots across all GPUs is the characteristic peak and subsequent dip at medium input sizes. *ScanBox* outperforms CUB at small input sizes but performs significantly worse at medium sizes. We propose two potential directions for improving performance in this regime:

1. **Introducing a Lookback Range:** In Parallelized Decoupled Lookback, each subgroup currently loads a single block descriptor per iteration. We hypothesize that allowing each subgroup to acquire multiple descriptors per iteration could better utilize the GPU's cache. This is because a single global memory access may cache way more than 128 bytes, exceeding the total amount of data actually used by a subgroup in a single lookback iteration. If more descriptors could be scanned per access, we could maintain the number of true memory loads, while doing more computations.
2. **Vectorizing the Block Descriptor Array:** As an extension of the above idea, vectorizing the memory layout of the block descriptor array could further enhance spatial locality and

caching efficiency. This would introduce two new tuning parameters: a *lookback batch size* and a *lookback data type*. These parameters would control how many descriptors are processed at once and in what format (e.g., `vec2` or `vec4`), potentially improving memory throughput.

Both ideas target the medium input size regime. However, these techniques are likely to underperform at large input sizes, where cache locality breaks down. In those cases, fetching multiple block descriptors would result in redundant memory accesses that are discarded if even a single descriptor is not ready, reproducing the same bottlenecks seen in standard Parallelized Decoupled Lookback.

## Chapter 7

### Conclusion

In this work, we presented ScanBox, a parameterized prefix-scan implementation that approaches peak memory bandwidth across AMD and Nvidia GPUs, rivaling vendor-optimized libraries like CUB. By tuning over six core implementation decisions, we demonstrated that per-input-size tuning not only achieves high performance but also reveals deeper architectural insights across devices. Our results suggest that this tuning methodology can produce fair, portable kernels suitable for hardware-constrained environments like mobile and embedded systems. While performance portability was notably strong on the platforms we tested, this does not diminish the value of our approach. Our strategies consistently produced high-performing standalone kernels, reinforcing per-input-size tuning and our key implementation decisions as powerful tools for maximizing efficiency in prefix-scan and GPU kernel design more broadly.

# Bibliography

- [1] Advanced Micro Devices, Inc. *HIP Programming Guide, Version 6.4.43483*. AMD ROCm Documentation, 2024. Section: Hardware Implementation.
- [2] Paul Bauman, Noel Chalmers, Nick Curtis, Chip Freitag, Joe Greathouse, Nicholas Malaya, Damon McDougall, Scott Moe, René van Oostrum, and Noah Wolfe. Introduction to amd gpu programming with hip. Presentation, 2019. Available from AMD Developer Resources.
- [3] Nathan Benaich and the Air Street Capital team. 91 Accessed: 2025-06-12.
- [4] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1990.
- [5] Guy E. Blelloch. Vector models for data-parallel computing. Technical Report CMU-CS-93-100, Carnegie Mellon University, 1993.
- [6] Guy E. Blelloch, Siddhartha Chatterjee, and Marco Zagha. Solving linear recurrences with loop raking. Technical Report CMU-CS-93-173, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1993.
- [7] Brent and Kung. A regular layout for parallel adders, 1982.
- [8] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, April 1974.
- [9] Sunbal Cheema and Gul Khan. Gpu auto-tuning framework for optimal performance and power consumption. In *Proceedings of the 15th Workshop on General Purpose Processing Using GPU*, GPGPU ’23, page 1–6, New York, NY, USA, 2023. Association for Computing Machinery.
- [10] Hernán Ponce de León et al. Dat3m: A verification framework for weak memory models. <https://github.com/hernanponcedeleon/Dat3M>, 2025. GitHub repository, Accessed: 2025-06-16.
- [11] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing*, 38(8):391–407, 2012. APPLICATION ACCELERATORS IN HPC.

- [12] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.
- [13] Matt Godbolt. Compiler explorer. <https://godbolt.org>, 2025. Accessed: 2025-06-16.
- [14] Google Chrome Developers. New in webgpu 128, Apr 2025. Accessed: 2025-04-16.
- [15] Sayan Goswami, Kisung Lee, Shayan Shams, and Seung-Jong Park. Gpu-accelerated large-scale genome assembly. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, page 814–824. IEEE, May 2018.
- [16] Spencer Green, Eric Enderton, Tim Purcell, and Aaron Lefohn. Onesweep: Faster least-significant-digit radix sort on gpus. In *Proceedings of the ACM SIGGRAPH/Eurographics Conference on High Performance Graphics*, pages 1–11. ACM, 2022.
- [17] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Chapter 36: Parallel prefix sum (scan) with cuda. In Matt Pharr, editor, *GPU Gems 2*. Addison-Wesley, 2005. NVIDIA GPU Gems 2.
- [18] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with cuda. In *2007 Computer Graphics International*, pages 209–217, 2007.
- [19] W. Daniel Hillis and Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.
- [20] Intel Corporation. *oneAPI GPU Optimization Guide, Version 2023.0*. Intel Developer Documentation, 2022. Section: Sub-groups and SIMD Vectorization.
- [21] Khronos Group. Vulkan 1.3.281 Specification: computeFullSubgroups Feature. <https://registry.khronos.org/vulkan/specs/latest/html/vkspec.html#features-computeFullSubgroups>, 2025. Accessed: 2025-06-16.
- [22] Khronos Group. About khronos group. <https://www.khronos.org/about/>, n.d. Accessed: 2025-06-16.
- [23] Matthew Kieber-Emmons. Efficient parallel prefix sum in metal for apple m1: Comparison of optimal m1 gpu scan primitives to vectorized cpu performance, September 2021. Medium article, Accessed: 2025-06-16.
- [24] Jake Kirkham, Tyler Sorensen, Esin Tureci, and Margaret Martonosi. Foundations of empirical memory consistency testing. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.
- [25] Peter M. Kogge and Harold S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, C-22(8):786–793, 1973.
- [26] CHPL Lab. Easyvk. <https://github.com/ucsc-chpl/easyvk/>, 2023.



- [27] Jeremy Laird. Arm reportedly spooling up major new gpu architecture to take on nvidia, Aug 2024.
- [28] Reese Levine, Tianhao Guo, Mingun Cho, Alan Baker, Raph Levien, David Neto, Andrew Quinn, and Tyler Sorensen. Mc mutants: Evaluating and improving testing for memory consistency specifications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '23, page 473–488. ACM, January 2023.
- [29] Puya Memarzia and Farshad Khunjush. An in-depth study on the performance impact of cuda, opencl, and ptx code. *Journal of Information and Computing Science*, 10(2):124–136, 2015. Received: November 30, 2014; Accepted: February 26, 2015.
- [30] Duane Merrill and Michael Garland. Single-pass parallel prefix scan with decoupled look-back. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 577–586. IEEE, 2016.
- [31] Duane Merrill, Michael Garland, and Andrew Grimshaw. Policy-based tuning for performance portability and library co-optimization. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Press, 2012.
- [32] Duane Merrill and Andrew Grimshaw. Parallel scan for stream architectures. Technical report, 2009.
- [33] Shin Morishima and Hiroki Matsutani. Accelerating blockchain search of full nodes using gpus. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 244–248, 2018.
- [34] Mitchell Nelson, Zachary Sorenson, Joseph M. Myre, Jason Sawin, and David Chiu. Parallel acceleration of cpu and gpu range queries over large data sets. *Journal of Cloud Computing*, 9(1), August 2020.
- [35] NVIDIA Corporation. *NVIDIA CUDA Programming Guide Version 1.0*, 2007. Accessed: 2025-06-16.
- [36] NVIDIA Corporation. Parallel prefix sum (scan) with cuda. [https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86\\_website/projects/scan/doc/scan.pdf](https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/scan/doc/scan.pdf), 2007. NVIDIA CUDA SDK Documentation.
- [37] NVIDIA Corporation. Nvidia cuda compute unified device architecture: Programming guide. Technical Report NVR-2008-004, 2008. NVIDIA Technical Report.
- [38] NVIDIA Corporation. *CUDA C++ Programming Guide, Version 12.9*. NVIDIA Developer Documentation, 2024. Section: Hardware Implementation.
- [39] NVIDIA Corporation. *CUDA C++ Programming Guide, Version 12.9*. NVIDIA Developer Documentation, 2024. Section: Performance Guideliens.

- [40] NVIDIADeveloper.
- [41] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.
- [42] Roger Pearce, Maya Gokhale, and Nancy M. Amato. Dynamic graph data structures on gpus. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–11, 2010.
- [43] Hernán Ponce-de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. Dartagnan: Bounded model checking for weak memory models (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems: 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part II*, page 378–382, Berlin, Heidelberg, 2020. Springer-Verlag.
- [44] Shruti K. Ramani, Vaishali J. Desai, and Kruti K. Karia. Gpu - graphics processing unit. *International Journal of Innovative Research in Computer Science & Technology (IJIRCST)*, 3(1):57–60, January 2015. Department of Computer Science, Saurashtra University, Rajkot, Gujarat.
- [45] Burkhard Ringlein, Thomas Parnell, and Radu Stoica. Gpu performance portability needs autotuning, 2025.
- [46] Abill Robert. Comparative analysis of cpu vs. gpu performance in bioinformatics data processing. EasyChair Preprint 13778, EasyChair, 2024.
- [47] Karl Rupp, Peter Tillet, Florian Rudolf, Josef Weinbub, Tibor Grasser, and Ansgar Jüngel. Performance portability study of linear algebra kernels in opencl. In *Proceedings of the International Workshop on OpenCL 2013–2014 (IWOCCL ’14)*, pages 1–11. ACM Press, 2014.
- [48] Markus R Schmidt, Anna Barcons-Simon, Claudia Rabuffo, and T Nicolai Siegel. Smoother: on-the-fly processing of interactome data using prefix sums. *Nucleic Acids Research*, 52(5):e23–e23, January 2024.
- [49] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Chapter 39: Scan primitives for gpu computing. In Hubert Nguyen, editor, *GPU Gems 3*. Addison-Wesley, 2007. NVIDIA GPU Gems 3.
- [50] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. Technical report, University of California, Davis, 2007. Technical Report.
- [51] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision, 2024.
- [52] Hideaki Shimazaki and Shigeru Shinomoto. A method for selecting the bin size of a time histogram. *Neural Computation*, 19(6):1503–1527, June 2007.

- [53] J. Sklansky. Conditional-sum addition logic. *IRE Transactions on Electronic Computers*, EC-9(2):226–231, 1960.
- [54] Marc Snir. Depth-size trade-offs for parallel prefix computation. *Journal of Algorithms*, 7(2):185–201, 1986.
- [55] Tyler Sorensen and Alastair F. Donaldson. The hitchhiker’s guide to cross-platform opencl application development. In *Proceedings of the 4th International Workshop on OpenCL, IWOCL ’16*, page 1–12. ACM, April 2016.
- [56] Tyler Sorensen and Heidy Khlaaf. Leftoverlocals: Listening to llm responses through leaked gpu local memory, 2024.
- [57] Huayou Su, Nan Wu, Mei Wen, Chunyuan Zhang, and Xing Cai. On the gpu-cpu performance portability of opencl for 3d stencil computations. In *2013 International Conference on Parallel and Distributed Systems*, page 78–85. IEEE, December 2013.
- [58] The Verge. Amd radeon rx 9070 series launches with rdna 4 and fsr 4 at ces 2025, 2025. Accessed: 2025-06-12.
- [59] Peter Thoman, Klaus Kofler, Heiko Studt, John Thomson, and Thomas Fahringer. Automatic opencl device characterization: Guiding optimized kernel design. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, pages 438–452, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [60] Jan V. Mcs 572 lecture 16 notes, 2025. Accessed: 2025-06-16.
- [61] Amin Vahdat. Ironwood: The first google tpu for the age of inference, Apr 2025.
- [62] World Wide Web Consortium (W3C). W3c mission: Leading the web to its full potential. <https://www.w3.org/mission/>, n.d. Accessed: 2025-06-16.
- [63] Shengen Yan, Guoping Long, and Yunquan Zhang. Streamscan: fast scan algorithms for gpus without global barrier synchronization. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP ’13, page 229–238. ACM, February 2013.
- [64] Haixiao Zhang, Yu Wang, Mingkun Li, Qinghua He, and Tong Zhang. Sugar-bi: A gpu-accelerated tool for bisulfite sequencing data analysis. *Nucleic Acids Research*, 52(5):e23–e23, 2024.
- [65] Yao Zhang, Mark Sinclair, and Andrew A. Chien. Improving performance portability in opencl programs. In Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer, editors, *Supercomputing*, pages 136–150, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.