HIGH PERFORMANCE HISTOGRAMMING ON MASSIVELY PARALLEL PROCESSORS

BY

GREGORY JUSTIN ROSS

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Adviser:

Professor Wen-Mei W. Hwu

ABSTRACT

Histogramming is a technique by which input datasets are mined to extract features and patterns. Histograms have wide range of uses in computer vision, machine learning, database processing, quality control for manufacturing, and many applications benefit from advance knowledge about the distribution of data.

Computing a histogram is, essentially, the antithesis of parallel processing. Without the use of slow atomic operations or serial execution when contributing data to a histogram bin in an input-driven method, there would likely be inaccuracies in the resulting output. An output-driven method would eliminate the need for atomic operations but would amplify read bandwidth requirements, reduce overall throughput, and result in a zero or negative gain in performance.

We introduce a method to pack multiple bins into a memory word with the goal of better utilizing GPU resources. This method improves GPU occupancy relative to earlier histogram kernel implementations, increases the number of working threads to better hide the latency of atomic operations and collisions while maintaining reasonable throughput. This technique will be demonstrated to improve performance of histogram functions of various sizes with a variety of inputs, including a study on a particular application. While the results are heavily driven by dependancies on input data patterns, the conclusions gathered in this thesis will outline that the packed atomics histogramming kernel can and usually does outperform other implementations in all but a select number of exceptions.

ACKNOWLEDGMENTS

I would like to express my appreciation and thanks to my adviser, Wen-Mei Hwu, for his support and guidance. Secondly, I would like to thank Dr. Volodymyr Kindratenko and Dr. Sanjay J. Patel of the University of Illinois as well as Mark Roulo and Eliezer Rosengaus of KLA-Tencor for providing both inspiration and unique case studies that led to the creation of this thesis.

Over the last few years there have been a number of people who have been an influential part of my academic and professional careers. The hard work and dedication that all of you demonstrated set an example that I shall learn from for many years to come.

In no particular order thank you all: Joshua Cochrane, Keith Campbell, Sam Cornwell, Nady Obeid, John Stratton, Neil Crago, Wojciech Truty, Puskar Nada, Daniel Johnson, Marie-Pierre Lassiva-Moulin, and many more from the Rigel Group, the ECE 190 family, and NSCA whose names are too numerous to mention.

Special thanks also goes out to Assistant Professor Yih-Chun Hu who has always encouraged me to make time to complete my research despite the responsibilities of being a teaching assistant for a truly awesome class, ECE 190.

Thank you also to my fellow Purdue faculty, graduate students, and alum whom were a part of my journey. In particular thank you to: Dr. Mark C. Johnson, Associate Professor Dr. Matthew Ohland, and Paul Griffin.

Last and certainly not least I would like to thank my grandmother, my extended family, and my wife and fellow graduate student Christine Ross.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

1080P	A resolution of 1920×1080 pixels
4K	In the context of an image, a resolution of 3840×2160 pixels
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit
GPGPU	General Purpose Computing on Graphics Processing Units
HDR	High-Dynamic Range Images
Hz	Hertz, a unit of frequency
IC	Integrated Circuit
ILP	Instruction Level Parallelism
OpenCL	Open Compute Language, similar to CUDA
PTX	Parallel Thread Execution, a pseudo-assembly language in CUDA
RGB	Red/Green/Blue image format
SIMD	Single Instruction Multiple Data
SM, SMX	Streaming Multiprocessor
KB	Kilobyte (1024^1 bytes)
GB	Gigabyte (1024^3 bytes)
Gpixel	Gigapixel (1024^3 inputs)
Gpixel/s	Gigapixels per second

CHAPTER 1

INTRODUCTION

Since CUDA's introduction in 2006 and OpenCL's introduction in 2008, GPUs have been regularly re-purposed for supercomputing platforms, data processing applications, and consumer image and video applications. The SIMD architecture of graphics cards from both nVidia and AMD lend themselves well to efficiently processing complex calculations over large sets of data.

Where massively parallel architectures falter, and where there are limited solutions available, is in applications requiring a high degree of synchronization where many threads will only contribute to a handful of outputs. Expensive atomic operations lead to large serialization of parallel processing which ultimately defeats the purpose of having a many-core system to begin with.

An architecture that lends itself well to serial execution will perform well on histogramming applications. However, depending on the specific needs of the user, histogramming on GPGPU systems may become a necessity. By creating an algorithm capable of providing sufficient throughput in histogramming applications, we enable GPGPU developers to avoid the need to manage or design systems that require coordinated execution between the traditional CPU and GPU. The main contribution of this thesis will be a scalable histogramming algorithm using packed atomic operations to accelerate histogram applications of various bin counts, ranging from 256 bins up to 2^{21} bins.

Chapter 2 introduces histogramming on GPGPUs and discusses its applicability to a variety of systems. Chapter 3 covers the architecture of a massively parallel CUDA device and the CUDA programming model. Chapter 4 introduces the benchmarks used in evaluating the algorithm. Chapter 5 outlines the algorithm, how it can be adapted for a number of applications. Chapter 6 discusses the performance of the algorithm. Finally, Chapter 7 contains the conclusion.

CHAPTER 2

BACKGROUND

To establish a need for the histogramming kernel introduced in this thesis, three generalized applications are discussed that will form the basis of the benchmarks established in Chapter 4.

2.1 Image Processing Applications

The need for efficient GPU histogramming is well established. Arguably the most recognized application for histogramming can be found in the field of image and video manipulation. Histogram equalization is an image processing technique where the overall distribution of pixel intensities (contrast) are adjusted with the goal of enhancing contrast between light and dark portions of the input image. This technique can be used to improve the contrast or the tonal range of images produced for a customer, used as a post-processing step in real-time graphics to improve the quality of video presented, or used to map high dynamic range images into a color space that can be displayed on a computer monitor that would otherwise not be capable of rendering HDR images.

In 2007, Scheuermann and Hensley [1] illustrated that histogramming a 1024×1024 pixel image required a significant amount of time to process (taking well over 3 milliseconds per image in a best case scenario). Though Lee et al. [2] established in 2010 GPU histogram throughputs had been dramatically increased, even in the best case scenarios a computation could still take well over a millisecond. That is a significant computational cost given a real-time video rendered on a personal computer allocates at most 17 milliseconds for processing each frame (60Hz refresh rate). As display resolutions move beyond 1080P high definition resolutions and into ultra high definition 4K images, and "3D" display techniques mandate 120Hz refresh rates, processing time becomes an ever more limited resource.

Medical imaging could benefit from image enhancement techniques as well. As written in [3], the insight segmentation and registration toolkit (ITK) is one of the most widely used open-source software toolkits for "the processing of medical images from CT and MRI scanners." Histogramming was originally implemented in ITK for the express purpose of normalizing gray-scale values of multiple medical images so that more valid comparisons of scans could be made. These imaging enhancement techniques have already been established as useful tools in other applications as well [4, 5, 6, 7]. It is also established in [3] that histogram processing on a small/local scale can aid in high resolution imaging sensors and image reconstruction technology.

Image histograms tend to process RGB images with three 8-bit color channels and generally consist of 256 bins. Should an application choose to histogram all three color channels that would produce histograms consisting of up to $2^{24} = 16\text{M}$ bins. With the onset of advanced display technology (HDMI 1.3, DisplayPort, etc) image processing applications may soon make frequent use of "Deep Color" (30-bit pixels or larger) inputs. It would be difficult with modern technology to calculate histograms of 2^{30} bins, but 2^{10} bin histograms certainly have near term applicability.

2.2 Computer Vision

Frequency distributions of an input's color data is useful for more than just human vision. It is a widely used tool in the field of computer vision and image recognition. Perhaps the most useful real-time application for histogramming is that of image tracking and image monitoring.

Consider the possibilities when a computer is able to track, in real-time, the end-user's head from the input of a camera or other video recording device. Birchfield [8] discusses the potential usefulness of this tracking ability yet it is specifically mentioned that improvements in histogramming would be useful to address algorithm shortcomings. Part of the bottleneck negatively impacting the algorithm was the refresh rate of the histogram – the histogram could not be computed fast enough for it to be resilient against sudden scene changes. Further study into this paper indicated that while the algorithm did work in real-time, the input video streams were far below high definition resolution.

Other real-time applications are discussed in Comaniciu et al. [9], Sizintsev et

al. [10], and Yoon and Kweon [11] where any generic object can be tracked to follow objects in motion or find objects whose color characteristics are known. These methods are computationally expensive and, as discussed in Section 2.1, if applied to high definition achieving real-time throughput could be difficult.

2.3 Advanced Image Processing, Statistical Analysis, Defect Detection

Most of the image processing applications discussed in this section have focused on small bin count histograms with usually no more than 256 output bins, there are actually a wide range of statistical analysis techniques that require several orders of magnitude more output bins. Mutual information of two random variables helps us to measure the dependence between two seemingly random variables. These two-dimensional histograms help users to determine joint probability density functions that could represent anything from pixel information (for advanced image processing techniques) to data regarding aspects of a manufacturing process (quality control and statistical analysis). Currently there are only a small number of efficient CUDA histogram methods, none of which advertise high throughputs when computing histograms of this size [12, 13, 14, 15]. This is of particular concern when the input data streams are extremely large and require real-time statistical analysis for process control.

Few references exist that indicate a need for histogram sizes exceeding 10,000 bins but there has been at least one case study for process control that involved the computation of joint probability distribution functions with inputs exceeding 2^{21} bits (over two million bins) [16]. It is this specific use-case that led to the development of this thesis.

CHAPTER 3

GPU ARCHITECTURE

From an algorithms standpoint a histogram is, fundamentally, a computation in which the output memory access pattern is unpredictable and requires atomic operations to avoid read-after-write, write-after-read, and write-after-write hazards and memory corruption. As a result histogramming does not map easily to GPGPU architectures.

The architecture we focus on in this thesis is a graphics processing unit, more specifically, the NVIDIA GTX 780 GPU. In this chapter, we will examine the details of the Kepler GK110 architecture. Full details on the GPU device, and the source for the figures included in this chapter, can be found in the GK110 whitepaper [17].

3.1 CUDA Programming Model

CUDA is an extension of the C/C++ programming language with added syntax to express the concepts of parallel execution. A program that runs on the GPU is called a *kernel*, which is loaded into the GPU by this host (i.e. the CPU) and commanded to run. This command to run specifies how many *thread-blocks* to launch, where each thread-block is a group of *threads*.

3.2 GPU Architecture

A single kernel can consist of any number of thread-blocks that are individually assigned to a *streaming multiprocessor* (SM). This thread-block assignment is performed in real-time as each multiprocessor advertises to the GPU scheduler that sufficient resources are available to execute another thread-block on the SM. The kernel's thread-blocks are automatically spread out across as many SMs as possible to improve overall performance.

3.2.1 Streaming Multiprocessor

The Kepler GK110 contains the newest incarnation of NVIDIA's streaming multiprocessor (herein referred to as an SMX for Kepler devices). The streaming multiprocessor is similar to a central processing unit within a desktop PC. The GTX 780 has twelve of these devices. As illustrated in Figure 3.1, each SMX contains 192 CUDA cores (each of which is a fully functional arithmetic unit), four dual-issue *warp* schedulers, and a large *shared memory* (visible to all cores in the SMX) which can also serve as an L1 cache.

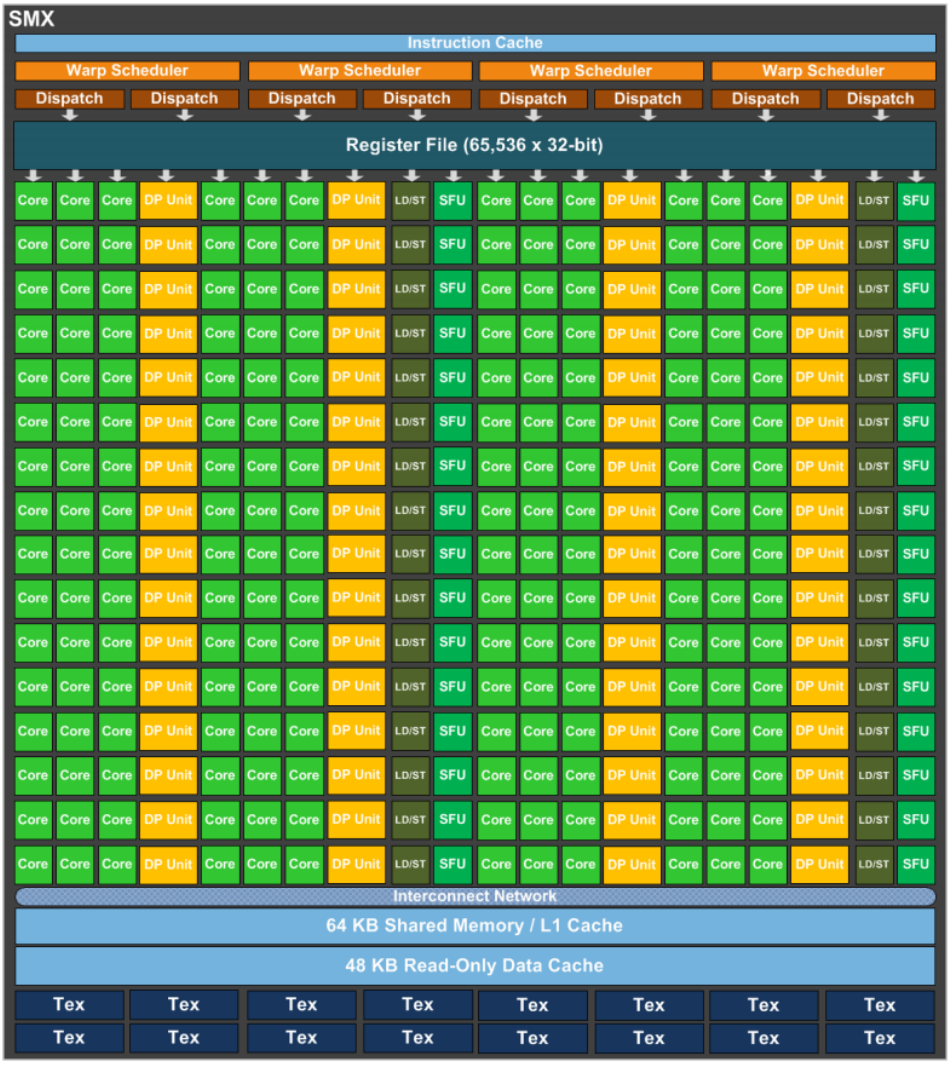


Figure 3.1: Streaming Multiprocessor

3.2.2 Warps and Threads

The SMX schedules threads of the kernel on a per-*warp* basis. A warp consists of a group of 32 threads within the thread-block that are all executed in parallel. Each group of 32 threads is assigned to 32 CUDA cores that execute in lock step (similar to a SIMD architecture).

The operation of a warp is of special concern for histogram applications. In order to achieve high performance the goal is to ensure that all 32 threads within a warp are performing useful tasks at all times. This is known as the SIMD execution model. If there is *divergence* (not all 32 threads execute the same code), the warp scheduler will execute each code path in a serial fashion. If each thread of a warp evaluates the same conditional but only four the threads within the warp compute a value of TRUE for that conditional, the warp scheduler will execute the IF code block with four threads (while leaving the other 28 threads idle) and then execute the ELSE code block with the other 28 threads (while leaving the first four threads idle). Care must be taken to avoid divergence when possible.

3.2.3 Warp Scheduler

Each Kepler SMX contains four warp schedulers, which are simple units responsible for fetching, decoding, and dispatching instructions. Each warp scheduler is dual issue, meaning that two independent instructions can be executed simultaneously within a warp. This execution flow is illustrated in Figure 3.2.

The challenge, therefore, is to keep all warp schedulers busy whenever possible. Each thread-block should have a sufficient number of threads ready to dispatch to utilize as many CUDA cores as possible. Because the warp scheduler is dual-issue, the compiled code would (ideally) be friendly to out-of-order execution and take advantage of instruction level parallelism. (For example, the code "$A = B + C$; $D = E + F$" has a higher degree of ILP, while code "$A = B + C$; $D = A + E$" does not have as much ILP due to dependencies.)

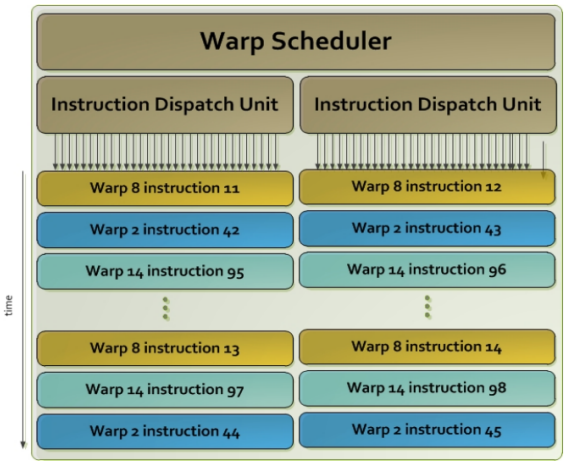


Figure 3.2: Warp Scheduler

By carefully examining the SMX system, one will note that the SMX is capable of dispatching eight instructions per clock and require 256 CUDA cores to process yet there are only 192 CUDA cores per SMX. In this scenario the dispatch system can only dispatch six instructions per clock, leaving the other two instructions available to dispatch on the next clock cycle. This could have a positive effect in that the memory subsystem would require fewer resources to prepare for each clock cycle, assuming the code path is more "deep" than "wide." This indicates that techniques such as loop unrolling and/or thread coarsening may be useful when optimizing the kernel.

On the other hand, more in-flight threads will put additional strain on the register file and shared memory subsystem. In the most recent iterations of CUDA architectures, NVIDIA has actually decreased the ratio of shared memory to CUDA cores (Tesla in 2008: 16 KB per 8 cores, Fermi in 2010: 48 KB per 32 cores, Kepler in 2012: 48 KB per 192 cores); this is not a friendly development for large scale histogramming.

The Fermi warp scheduler is similar, but smaller, and consists of a two single-issue warp schedulers per SM. As a result ILP-friendly code does not have as much of an impact on performance, however, to achieve high throughput it is necessary to launch thread-blocks sizable enough that there is normally going to be two warps that have instructions available to execute.

3.2.4 Memory System

There are several memory subsystems within the Kepler architecture that are of interest for GPU histogramming and shown in Figure 3.3. Global memory, shared memory, and register files are all of interest to histogramming applications.

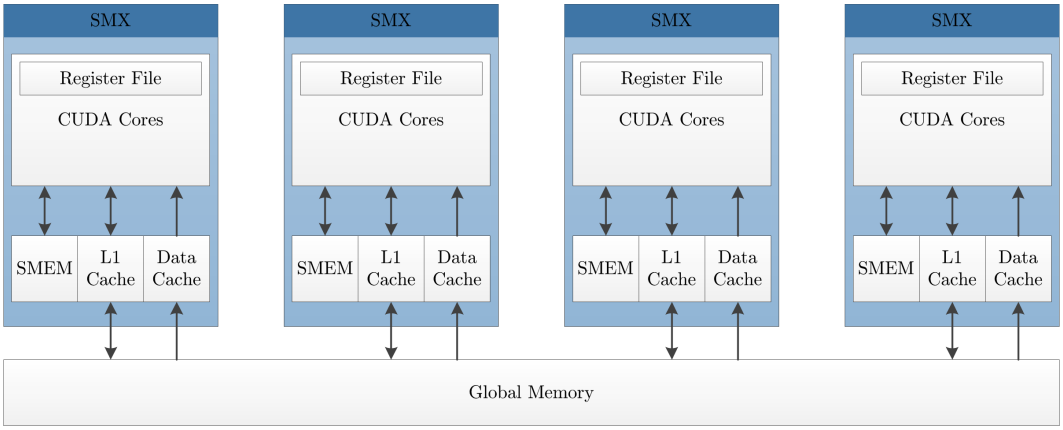


Figure 3.3: Memory Architecture

Other memory subsystems such as the L1 cache have little impact on the performance of a histogram application, aside from the requirement that we must keep all CUDA cores fed with memory bandwidth as they execute load and store instructions.

The L2 cache will serve an important, though somewhat unnoticed, role in the histogramming kernel as it is the primary buffer between off-chip DRAM and the SMXs. All global memory atomics take place in this space, and all data loaded is cached here until it is no longer needed or must be evicted. Kepler contains 1536 KB of L2 cache, which is plenty for most histogram applications. Fermi contains 640 KB of L2 cache. Both quantites of L2 are sufficient for most histogram kernels, with the exception of large histograms **if** the data is binned into widespread patterns that cause the L2 cache to frequently need to evict previously updated histogram bins to make room for bins being incremented in the present.

GLOBAL MEMORY

The first memory subsystem is the global memory system, which is a high-latency off-chip DRAM memory which is visible to all threads in flight across all multiprocessors. Read and write accesses are "slow" compared to other memory systems. Atomic operations in global memory lock memory locations for long periods of time while the data is loaded, incremented, and stored back into DRAM.

The Kepler whitepaper establishes that atomic operations are vastly improved compared to previous generation architectures. Atomic operations to a common global memory address will have a throughput of **one operation per clock**, compared to Fermi's **nine clocks per operation**. When the destination address is not common, the Kepler architecture can achieve 64 operations per clock, up from Fermi's 24 operations per clock [18]. While this advancement reduces the cost of using global atomics, the very nature of a histogram ensures collisions will frequently get in the way. This is especially evident in applications with small histogram bin counts.

When it comes to global memory access patterns, the histogram application has about as ideal a read-access pattern as possible. In each pass of the input data successive threads will load coalesced and aligned memory addresses; this is friendly to both DRAM and L1/L2 caching systems which are capable of supporting up to 288 GB/s on the GTX 780. As long as we ensure each thread accesses 32-bit or larger data types, enough threads exist to saturate the memory bus with requests and hide the latency of DRAM access. The global memory subsystem will not usually be a critical resource for histogramming.

SHARED MEMORY

The next memory store in the system is the shared memory block. This is an on-chip SRAM memory that is "extremely fast" and supports over 2.5 TB/s of throughput after summing the capabilities of all SMXs. Every SMX has its own shared memory block, which serves up a scratchpad visible to all threads within the thread-block. If a thread-block requests a larger scratchpad, fewer blocks can run on the same SMX in parallel.

In order to ensure high utilization of shared memory, accesses to shared memory should ideally be free of *bank conflicts*. A thread-block's scratch-pad is divided into 32

banks of memory. Each bank is four bytes (GF110, GK104) or eight bytes (GK110) wide. If multiple threads access the same bank they must also access byte(s) within the same word in order for the access to be considered free of conflicts (Figure 3.4).

Otherwise if two threads access different words in the same bank, those accesses are serialized and performance would be similar to divergent execution. Throughput of the instruction would be reduced by a factor equal to the maximum number of conflicts (i.e. maximum number of conflicting access to any one bank). In histogramming we also see serialization of execution when atomic operations access the same word; instruction latency is increased due to the extra clock cycles necessary to process serial access.

Note there are no published specifications on the behavior of shared memory when atomic operations access the space, but results in Chaper 6 demonstrate that shared memory atomics can operate at better than 83 Gpixels/s.

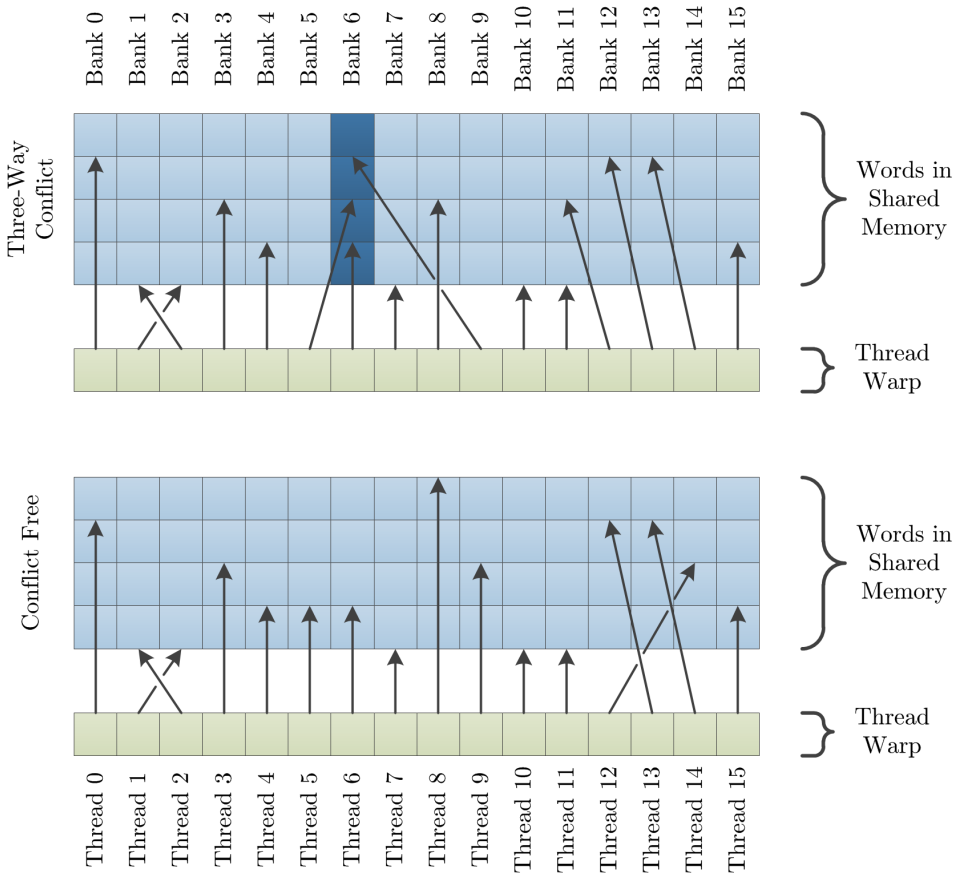


Figure 3.4: Shared Memory Accesses with and without Bank Conflicts

REGISTER FILE

The closest memory space is a private register file whose registers are assigned to each thread in one or more thread-blocks. Even with the most recent iteration of the SMX, each thread is limited to a maximum of 255 registers. In the heavily optimized histogramming kernels, having a larger register file provides the means to store additional data.

L1, L2, READ-ONLY DATA CACHES

The caching system on the Kepler architecture uses an L1/L2 cache system that operates very similarly to a traditional CPU – with the exception that memory buses tend to be much wider to support the bandwidth needs of massively parallel workloads.

From a histogramming standpoint, the only potential use of the cache subsystem is to accelerate loading input data multiple times if this becomes necessary in computing many-bin histograms. The Kepler "GK110" architecture contains 1536 KB of on-chip L2 cache. Prior iterations of Kepler and Fermi architectures had no more than 512 KB of L2 cache, while Tesla architectures only had a texture cache that was similar to an read-only data cache.

CHAPTER 4

DESCRIPTION OF BENCHMARKS

In this thesis, we use the hardware configuration from Table 4.1 to evaluate CUDA kernels in all benchmarks. The testbed system is utilizing NVIDIA's v332.88 drivers and the v6.0 CUDA Toolkit on a 64-bit Windows 7 installation.

Table 4.1: Hardware Configurations

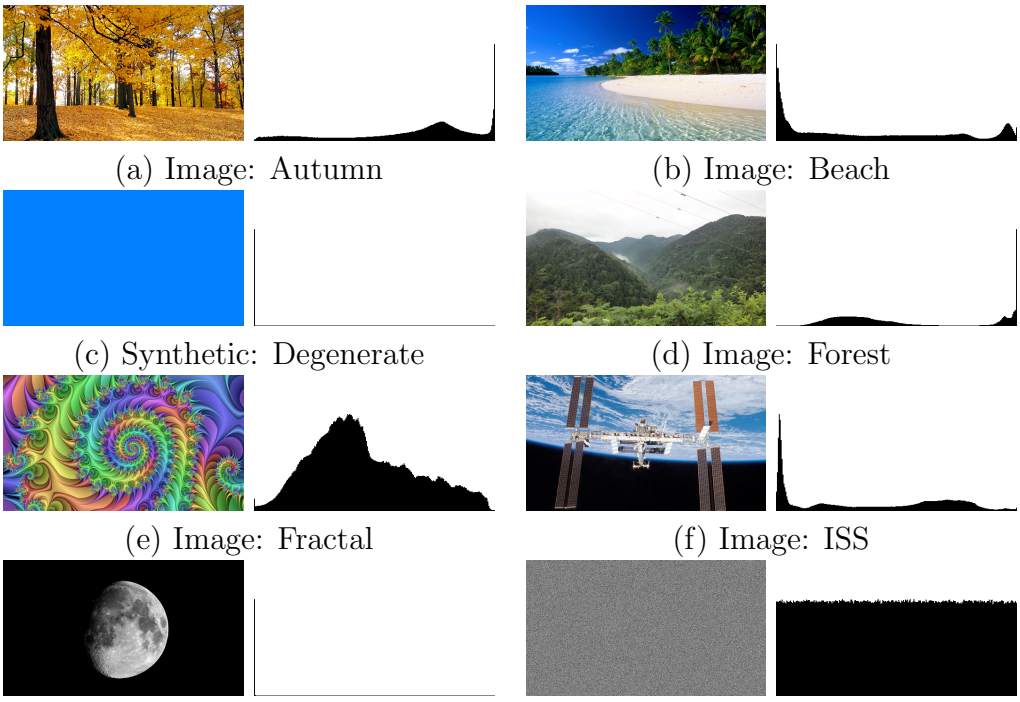
	Configuration 1	Configuration 2
Host CPU	Intel i7-4770K	Intel i7-4770K
GPU	eVGA GTX 780	eVGA GTX 570
Compute Capability	3.5	2.0
CUDA Cores	2880	480
GPU Frequency	941 MHz	732 MHz
GPU Bandwidth	288 GB/s	152 GB/s
GPU Streaming Multiprocessors	12	15
Cores per SMX	192	32
Shared Memory per SMX	48 KB	48 KB
L1 Cache per SMX	16 KB	16 KB
Read-Only Data Cache per SMX	48 KB	None
L2 Cache	1536 KB	640 KB

4.1 Image Processing

For kernels calculating 256-bin histograms, a variety of real-life and synthetic inputs are used to evaluate the performance of each kernel. For this set of benchmarks, 24-bit RGB images with a resolution of 1920×1080 and 3840×2160 pixels are used. Only the 8-bit red color channels are used during testing.

The images used and their output histograms are shown in Figure 4.1. These inputs include uniform distributions of data, randomized data, degenerate data that all map

to the same output bin, and real photographs.



(a) Image: Autumn (b) Image: Beach

(c) Synthetic: Degenerate (d) Image: Forest

(e) Image: Fractal (f) Image: ISS

(g) Image: Moon (h) Synthetic: Random

Figure 4.1: Color Images for Benchmarks

It may also be useful to know the average number of memory bank conflicts the GPU will have to contend with if/when an algorithm utilizes shared memory for sub-histograms. For every set of 32 pixels the number of bank conflicts that would be generated was computed, averaged, and the results included in Table 4.2. Even though several of the images have similar frequency distributions, the overall pattern of distribution can impact kernel throughput. A kernel that utilized shared memory could be expected to perform worse when inputs such as the "MOON" dataset, or better with the "FRACTAL" dataset.

Table 4.2: Bank Conflict Statistics for Input Images

Image	Average Bank Conflicts	Std. Dev.
Autumn	4.46	2.02
Beach	6.42	4.18
Degenerate	32	0
Forrest	8.49	8.36
Fractal	4.23	2.01
ISS	6.43	3.53
Random	3.53	0.74
Moon	26.36	11.17

4.2 Data Analysis in Wafer Fabrication

Consider a wafer fabrication facility that produces silicon die used in a variety of ICs such as processor units, memory chips, and graphic processing units. Semiconductor fabrication plants such as the 300 mm wafer fabrication facility in the Central Taiwan Science Park is expected to output more than 100,000 wafers per month.

One 18-month-long study from a firm that supports IC design teams cited that the average die size from their customers using the 65 nanometer manufacturing process comes out to 2.13 mm $\times$ 2.13 mm [19]. Given a 300 mm diameter wafer we compute that more than 15,000 die can fit onto a single wafer. The largest die cited in the study was 20.253 mm $\times$ 20.253 mm; a 300 mm wafer could hold approximately 140 die of that size.

In other words, a single 300 mm wafer fabrication facility can output more than 1.5 billion average sized die per month or 14 million of the largest designed seen in the study.

Having established the volume expected in a facility, we shall discuss processor yield. Each defect in a wafer has the potential to render one or more die useless. IC defects can originate from a defect in the wafer itself or a defect in the process itself.

To detect defects during the manufacturing process for a wafer, image data is collected by scanning the wafer in between manufacturing steps as illustrated in Figure 4.2. These wafer images are broken into images of the individual die which are then compared amongst each other to find any die that substantially differ from the rest of the set.

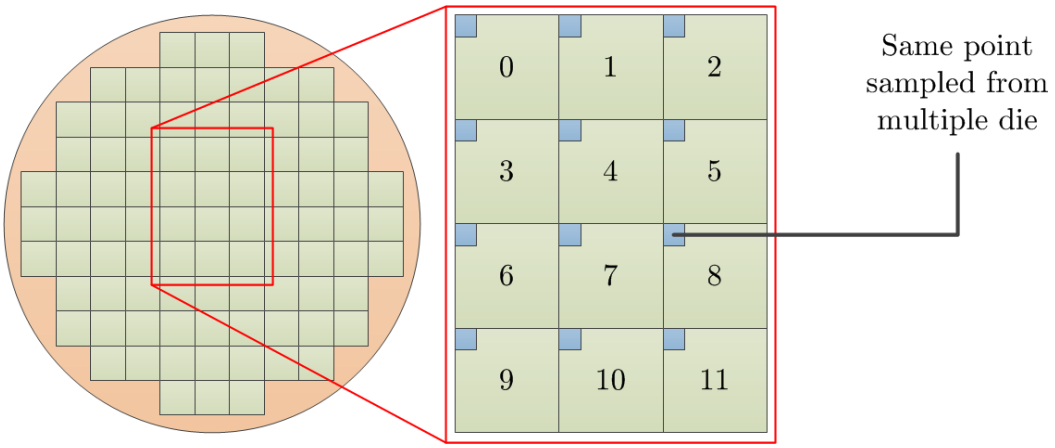


Figure 4.2: Breakdown of Silicon Wafer

This comparative analysis is where histogramming is used in a defect analysis. The point $p_i(x, y)$ is sampled from each image i. The arithmetic mean of the point from all images is computed. In this particular application, M is an 8-bit unsigned value ranging from 0 to 255.

$$M(x, y) = \frac{1}{n} \sum_{i=1}^{n} p_i(x, y)$$

The mean value M is then compared against all the points $p(x, y)$ sampled from the input images $i_1, i_2, ..., i_{imageCount}$. Then the *variance* is computed based on a scaled difference of pairs of points from image i and j and the mean of that point from all images in the set.

$$N(x, y, i, j) = variance\left(M, p_i(x, y), p_j(x, y)\right)$$

The *variance* value N is a 12-bit or 13-bit unsigned value (the bit width of the variance depends on the requirements of the wafer manufacturing process and precision of the instruments used to examine the wafers in the manufacturing line).

The final histogram is then considered to be a two-dimensional histogram of 256×4096 or 256×8192 bins. Each bin has an output value equal to:

$$bin_{m,n} = \sum_{x=1}^{imageSize_x} \sum_{y=1}^{imageSize_y} \sum_{i=1}^{imageCount} \begin{cases} 1, & \text{if } m = M(x, y) \text{ and } n = N(x, y, i, j) \\ 0, & \text{otherwise} \end{cases}$$

The remainder of the comparative analysis algorithm processes the resulting histogram to look for outliers which may or may not indicate a faulty die within the wafer. Those routines are outside the scope of this thesis.

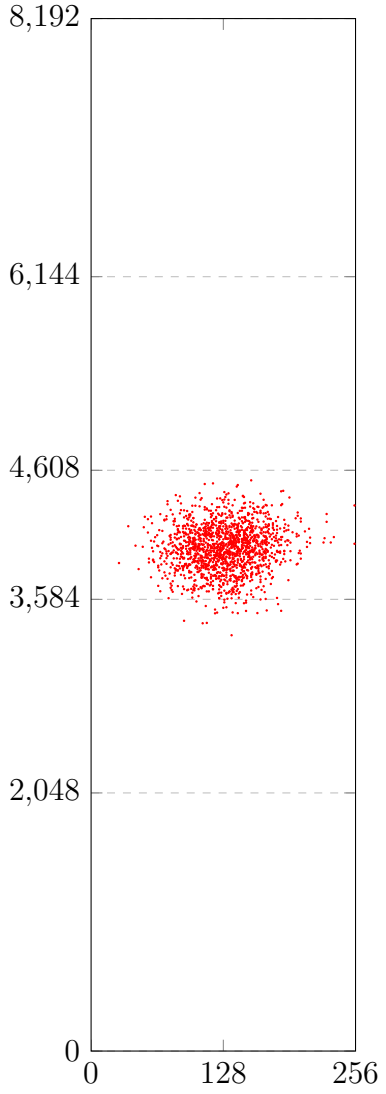


Figure 4.3: Output Histogram

Authentic data captured from silicon manufacturing lines was not available, however, KLA-Tencor has provided routines that generate synthetic data. These synthetic datasets approximate the distributions of data seen in the field.

An example of an output image is shown in Figure 4.3. As can be seen in the data the output histogram is sparse; more than 99% of input data falls within the middle 10% (200,000) of the histogram bins.

In this particular use-case the histogram kernel sits right in the middle of a series of kernels that must be executed on the image data collected. Transferring the histogram inputs to the CPU, computing the histogram, and transferring the results back to the GPU was not advised in their architecture.

In this particular benchmark, histogram consisting of 256×4096 and 256×8192 bins will be used and represent real-world applications for the packed atomics histogramming kernel.

A full depiction of the input datasets provided please see Appendix A.

4.3 Generalized Use

In statistical analysis a histogram is a representation of the distribution of data and, hence, a useful indicator of trends or a precursor to data binning. To represent potential applications, several datasets will be used to give a general idea kernel performance.

Uniformly Distributed Data

A random number generator will be used to create inputs ranging from $0, \ldots, N-1$ and shall be uniformly distributed across the range.

Degenerate Distribution of Data

All inputs will map to a single bin.

Clustered Data

A random number generator will be configured to create inputs with a Gaussian distribution such that 99% of all inputs map to a region filling approximately 12.5% of the histogram bins. When benchmarking this application, the kernel shall not make any assumptions regarding the clusters of data. For testing purposes, the dataset used to produce the distribution seen in Figure A.3 of Appendix A will be used.

CHAPTER 5

PACKED ATOMICS HISTOGRAMMING

5.1 Problem Statement and Design Goal

Prior research into histogram throughout has largely focused on developing algorithms that reduce the overall number of atomic operations involved in the histogram computation [1, 15, 20, 21, 22].

The most common technique, and one this thesis builds upon, is to use faster shared memory atomics to process the input data prior to contributing the shared memory histogram into the final histogram stored in global memory.

Atomic operations on the CUDA architecture operate only on 32-bit or 64-bit integers. This typically limits shared memory kernels to storing no more than 4,096 bins per SM on a Tesla GPU or 12,288 bins per SMX on a Kepler GPU.

Compression or packing routines can fit more bins but then require duplication of bins such that atomic operations are not necessary; this is counter productive and leads to a zero or negative gain.

Thus it is the design goal of this kernel to improve the net count of histogram bins in shared memory and keep GPU occupancy high enough to aid in masking latencies of atomic operations. For sufficiently large histograms, such as the two million bin use case, the algorithm should scale well and maintain throughput as best as possible.

5.2 Histogram Kernel: Data Design

Each thread-block in the histogram kernel is responsible for computing its own shared memory sub-histogram. Bins are allocated 4-bits (or 8-bits) of shared memory depending on the application, hence, the shared memory bin cannot exceed a value of $2^B - 1$ (15 or 255).

Shared memory atomic operations only operate at a 32-bit and/or 64-bit granularity, therefore each addition into shared memory has the potential to impact up to multiple bins in the block histogram. When overflow does occur (where one bin rolls over to 0), subsequent bins in the word can be falsely incremented.

In order to counteract overflows, two options exist. The first involves limiting thread-block sizes and requires heavy use of synchronization; this severely reduces throughput due to low GPU occupancy. The second option, and the one we elect to use, is to address overflows by tracking them. The kernel uses the histogram as a storage space to hold correction values for each bin in the histogram. This correction factor, when added to the thread-block sub-histogram outputs, results in an accurate output value. The general dataflow of the kernel is shown in Figure 5.1.

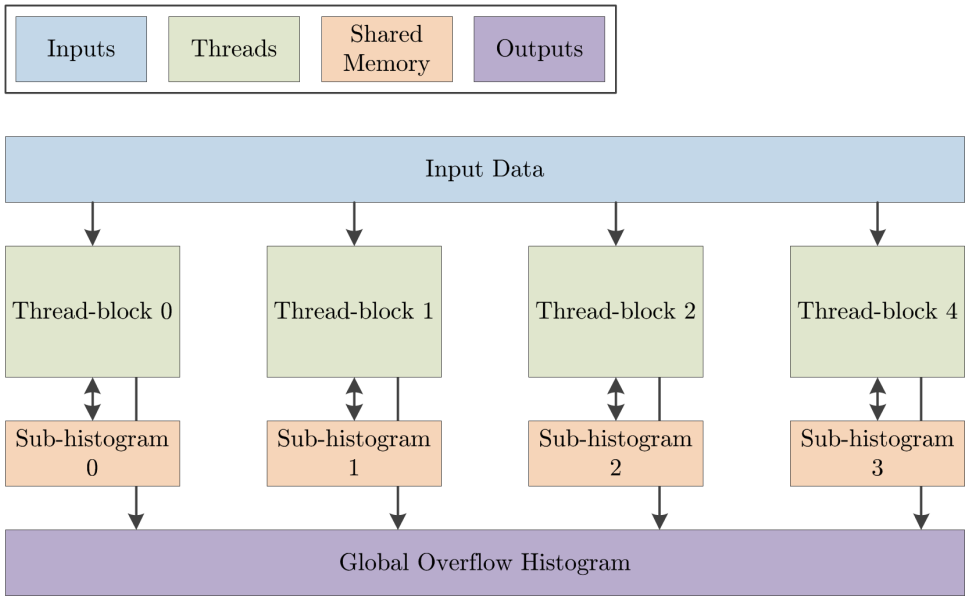


Figure 5.1: Thread-Block Histogram Kernel

5.3 Histogram Kernel: Method

Atomic operations are required to ensure proper incrementing of histogram bins. Unfortunately these operations only operate on four byte words; any rollover from 0xFF to 0x00 has the potential to impact all more significant bytes in the shared memory word. The completed kernel code is capable of compensating for this

problem, but the compensating code requires a number of clock cycles to execute. To detect overflow into the next byte in the word, the kernel takes advantage of the fact that the **atomicAdd()** returns the previous value of the word. The high level code is shown in Figure B.1 of Appendix B.

In the typical kernel configuration where eight bits are assigned to each bin ($B = 8$), between one and $32/B$ corrective actions need to be taken upon arithmetic overflow. The global histogram bin_N is incremented by 2^B. Subsequent bins in the integer must then be examined to determine if a corrective value must be added into the global histogram. This is done by comparing the previous value of the integer (the return value of the atomicAdd() function) and the resulting value of the integer on a bin-by-bin basis. When there is no change in a bin's value, no corrective action is taken. If there has been a change in a bin's value, either -1 or $(2^B - 1)$ must be added to the global histogram bin so that the final histogram value is still correct.

To illustrate, examples are given in Sections 5.3.1 and 5.3.2. These examples show how the code listing in Figure B.2 of Appendix B operates.

By forming a thread-block sub-histogram, the total number of global atomic operations is reduced from *bins* to the number of byte overflows that occur during kernel execution, plus the number of bins held in shared memory. Overflow is a factor that cannot be controlled and is entirely dependent on the input data.

As a general note, experiments show it is actually more efficient to directly add the thread-block sub-histogram into the global histogram via atomic operations on the Kepler and Fermi architectures. This is due to the improved efficiency of global atomic operations relative to older GPU architectures such as the Tesla architecture. Older GPU architectures with poor global atomic throughput should store the shared memory histograms into a global memory then launch a reduction kernel to compute the final output.

5.3.1 Example 1: Simple Overflow in Least Significant Bin

The first example in Figure 5.2 shows how the kernel handles the most simple arithmetic overflow. The least significant bin within the shared memory integer is incremented, subsequently overflows, and corrective action is taken. Note all values are in hexadecimal notation.

	0x	000000FF	Original Value in Shared Memory
	0x	......FF	Original bin_N
	0x	....00..	Original bin_{N+1}
	0x	..00....	Original bin_{N+2}
	0x	00......	Original bin_{N+3}
+	0x	01	Atomic Add to Increment bin_B
	0x	00000100	Result of Atomic Add in Shared Memory
	0x	00	Resulting bin_N
	0x	....01..	Resulting bin_{N+1}
	0x	..00....	Resulting bin_{N+2}
	0x	00......	Resulting bin_{N+3}
	0x	100	Correct Value of bin_N
	0x	....00..	Correct Value of bin_{N+1}
	0x	..00....	Correct Value of bin_{N+2}
	0x	00......	Correct Value of bin_{N+3}
	0x	100	Corrective Value of bin_N to add to global histogram
	0x	....-1..	Corrective Value of bin_{N+1} to add to global histogram
	0x	..00....	No addition to global histogram for bin_{N+2}
	0x	00......	No addition to global histogram for bin_{N+3}

Figure 5.2: Simple Overflow in Least Significant Bin

When bin_N is incremented by 1, it rolls over from 255 to 0. This results in the kernel adding 256 to the global histogram. This results in the sum of the resulting shared memory bin_N and resulting global memory bin_N equaling the sum of the original shared memory bin_N and original global memory bin_N plus 1.

The next significant bin in the integer, bin_{N+1}, is falsely incremented from 0 to 1. Because the original value of the bin was **not** 255, the thread adds -1 to the global memory bin_{N+1}. The total sum of bin_{N+1} between the global memory and shared memory histogram is thus left unchanged.

5.3.2 Example 2: Multiple Overflows from Least Significant Bin

This next example in Figure 5.3 shows how the kernel handles arithmetic overflow that impacts most of the bins within the integer. The least significant bin within the shared memory integer is incremented, subsequently overflows into two bins, and different corrective actions are taken in each case. Note all values are in hexadecimal notation.

	0x	0001FFFF	Original Value in Shared Memory
	0x	......FF	Original bin_N
	0x	....FF..	Original bin_{N+1}
	0x	..01....	Original bin_{N+2}
	0x	00......	Original bin_{N+3}
+	0x	01	Atomic Add to Increment bin_B
	0x	00020000	Result of Atomic Add in Shared Memory
	0x	00	Resulting bin_N
	0x	....00..	Resulting bin_{N+1}
	0x	..02....	Resulting bin_{N+2}
	0x	00......	Resulting bin_{N+3}
	0x	100	Correct Value of bin_N
	0x	....FF..	Correct Value of bin_{N+1}
	0x	..01....	Correct Value of bin_{N+2}
	0x	00......	Correct Value of bin_{N+3}
	0x	100	Corrective Value of bin_N to add to global histogram
	0x	....FF..	Corrective Value of bin_{N+1} to add to global histogram
	0x	..-1....	Corrective Value of bin_{N+2} to add to global histogram
	0x	00......	No addition to global histogram for bin_{N+3}

Figure 5.3: Multiple Overflows from Least Significant Bin

When bin_N is incremented by 1, it rolls over from 255 to 0. This results in the kernel adding 256 to the global histogram. This results in the sum of the resulting shared memory bin_N and resulting global memory bin_N equaling the sum of the original shared memory bin_N and original global memory bin_N plus 1.

The next significant bin in the integer, bin_{N+1}, is falsely incremented from 0 to 1. Because the original value of the bin was **not** 255, the thread adds -1 to the global memory bin_{N+1}. The total sum of bin_{N+1} between the global memory and shared memory histogram is thus left unchanged.

5.3.3 Probability of Overflow

As this code path involves a large number of operations, the probability of overflow must be examined. Under the following assumptions and given a uniformly distributed set of inputs:

$$w = \text{Threads per Warp} = 32$$
$$B = \text{Bits per Bin in 32-bit Word} = 8$$
$$N = \text{Bits per Input} = 8$$

The probability of any single warp experiencing an overflow event is:

$$P(overflow) = w \times \frac{1}{2^B} \times \frac{1}{2^N}$$
$$P(overflow) = 32 \times \frac{1}{256} \times \frac{1}{256}$$
$$P(overflow) = 0.0488\% \ = 0.05\%$$

For a single 1920×1080 input image, that amounts to more than 1,000 overflow events per image that slow down execution of the kernel. Assuming uniform distribution of overflows across all thread-blocks there would be approximately 43 overflow events per thread-block. Each overflow event stalls the SMXs ability to retire that thread-block, thus drawing out the total kernel run time and reducing overall throughput.

5.4 Extension 1: Thread Coarsening

Thread coarsening is a method by which each worker thread processes multiple inputs per iteration in order to amortize loop overhead over multiple inputs. By doubling or quadrupling the number of items processed per thread there is potential to take advantage of addtional global memory bandwidth (in the form of wider memory accesses which can make more efficient use of the L1 cache or DRAM memory bandwidth) and better hide the latency associated with global memory reads and writes (because that approximately 800 clock cycle wait for a load to complete is amortized over two or four inputs instead of a single input).

As can be seen in Figure 5.4, the relative performance benefits of thread coarsening are consistent for small to medium histograms. A few select sample inputs were run through optimized histogramming kernels with the relative results displayed. The baseline implementation has a thread coarsening factor of 1, meaning it only loads one 8-bit input per iteration, and establishes a baseline of 1X on our graph. All other kernels performance figures are relative to that (e.g. an optimized kernel showing 1.25X has 25% more throughput given the same inputs). Realized performance will vary on the pattern of the data.

Thread coarsening actually *worsens* performance for small and medium histogram kernels. With 100% GPU occupancy and a relatively large number of threads competing for a relatively small number of shared memory words, throughput is better served with a balance between atomic operations and shared memory atomics.

For histogram kernels with tens of thousands of output bins as shown in Figure 5.5, we obtain opposing trends with respect to thread coarsening. The kernel makes use of casting techniques to load 2×32-bit (2 inputs, 32-bit input, 64-bit loads) or 4×32-bit values (2 inputs, 32-bit input, 128-bit loads).

The key to this dichotomy is in GPU occupancy figures. For histogram kernels with 24K bins and less the histogram kernel is able to achieve 100% GPU occupancy; this is why a balance between shared memory atomic operations and global memory access results in higher performance. When the histogram sizes exceed 24K bins the histogram kernel immediately drops to a 50% occupancy; with fewer threads available to hide global memory access latency, better performance is obtained by grouping accesses together.

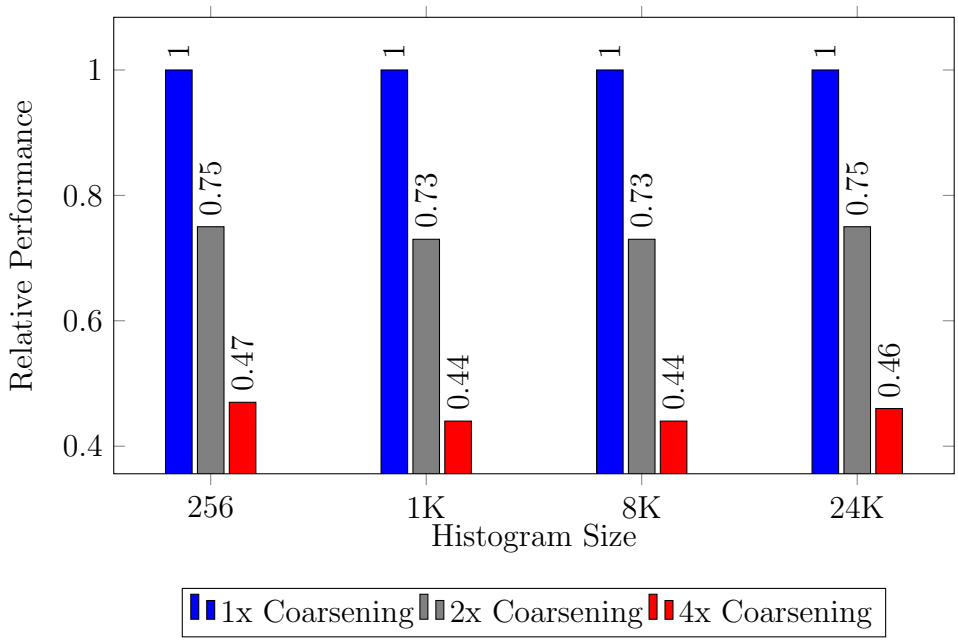


Figure 5.4: Extension 1, Small Histograms

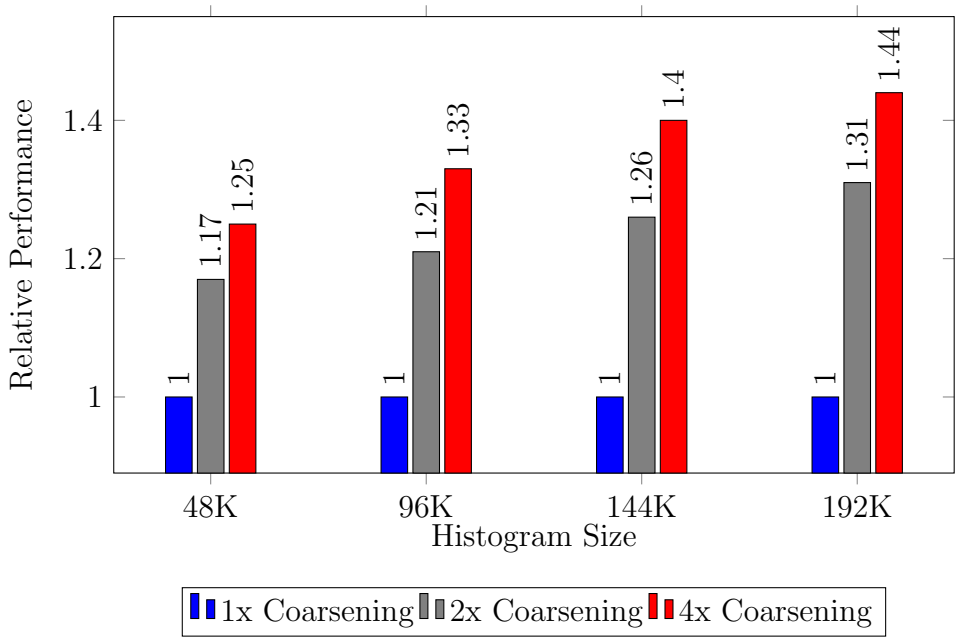


Figure 5.5: Extension 1, Large Histograms

5.5 Extension 2: Multiple Shared Memory Histograms

For histograms containing 24×1024 or fewer bins, the packed atomic histogram kernel can afford to launch two thread-blocks per SMX. This almost always improves performance as the kernel will enjoy a GPU occupancy factor of 100%, up from 50%. However as the histogram size decreases multiple atomic operations will collide more often and serialize accesses between competing threads. This is particularly painful in a standard 256-bin histogram kernel; 1024 threads would all be competing for the same 256 atomic locks leading to a dramatic reduction in throughput.

Extension 2 introduces the concept of **duplicating** shared memory histograms in each thread-block and maintaining as close to 100% usage on shared memory resources. Experiments indicated optimal performance was achieved when the number of histograms per thread-block is set to a power of 2. Table 5.1 outlines how many shared memory histograms are to be used in each thread-block, given a specific number of bins.

This optimization has substantial impact on smaller histogram kernels, as can be shown in Figure 5.6. The benefit, however, wears off no additional performance gains are achieved as the histogram size exceeds 4K with uniformly distributed inputs.

For datasets that are not expected to be uniformly distributed, the performance advantage of this extension for small histograms is made even more clear. Figure 5.7 shows that with a Gaussian distribution of inputs where atomic contention (same number of inputs, tighter distribution of outputs) is partially canceled out by the duplication technique. In general, the benefit of this duplication again wears out around the 4K histogram mark where it achieves parity with the non-optimized version of the kernel.

Table 5.1: Mapping Multiple Histograms per Thread-Block

	Bins	Sub-Histograms
$\leq 24 \times 1024$	$= 24,576$	1
$\leq 12 \times 1024$	$= 12,288$	2
$\leq 6 \times 1024$	$= 6,144$	4
$\leq 3 \times 1024$	$= 3,072$	8
$\leq 1.5 \times 1024$	$= 1,536$	16
$\leq 0.75 \times 1024$	$= 768$	32
$\leq 0.375 \times 1024$	$= 384$	64

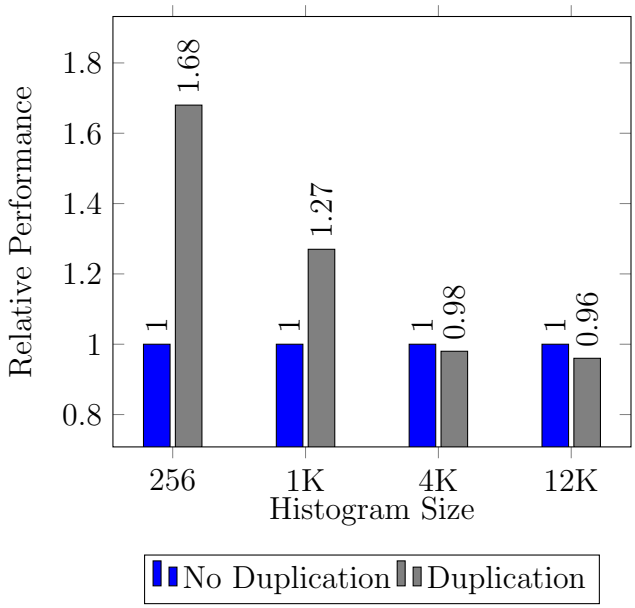


Figure 5.6: Benefits of Extension 2, Uniformly Distributed Inputs

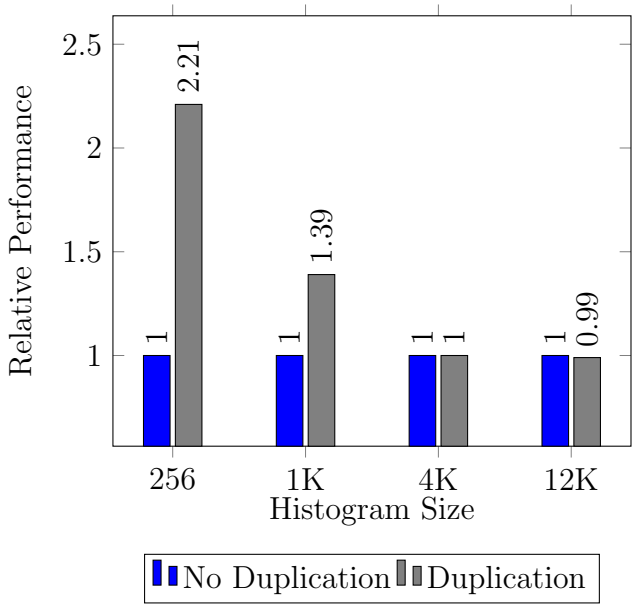


Figure 5.7: Benefits of Extension 2, Gaussian Distribution of Inputs

5.6 Extension 3: Pre-Processing Histogram Inputs

For large histograms that exceed 48K bins, it is necessary to add vertical tiling into the algorithm. The function responsible for incrementing shared memory bins and addressing any overflows has to be expanded to accommodate tiling.

The idea to vertical tiling is simple. A block with $blockIdx.y == 0$ would create a histogram for bins ranged from [0, 48K), $blockIdx.y == 1$ scans the same input sets to handle bins ranged from [48K, 96K), so on and so forth. Each thread must, therefore, determine which tile the bin belongs in by dividing the input by the number of bins that can be held in shared memory. The thread would then process the input only if the data falls into the range assigned to that particular $blockIdx.y$.

The additional division step as shown in Figure B.3 in Appendix B can negatively impact throughput; CUDA cores do not have efficient division capabilities unless the division is by a power of two (which simplifies into a binary shift). By re-arranging the steps, introducing a kernel to swizzle input data and eliminating the lines of code that are called-out in Figure B.3, the number of expensive division operations are $O\,(inputs)$ rather than $O\,(threads)$.

In the tiled histogram kernel, the increment function is modified to load these pre-processed values per the code in Figure B.4 of Appendix B. The PTX code generated is full of binary shifts and logical operations that may or may not be more efficient than leaving the division code in the histogramming kernel itself.

Note that this extension only becomes useful as the number of bins, and thus the number of passes required to process the input data, increase. Even then, the performance improvement is minuscule and does not factor in the kernel launch that converts the input from a raw to a more optimal format. This speed-up is illustrated in Figure 5.8. Note that some architectures, particularly those with weak division capabilities, may still benefit from this extension. Otherwise this extension is only useful for extracting out the last bit of performance for extremely large histogram kernels.

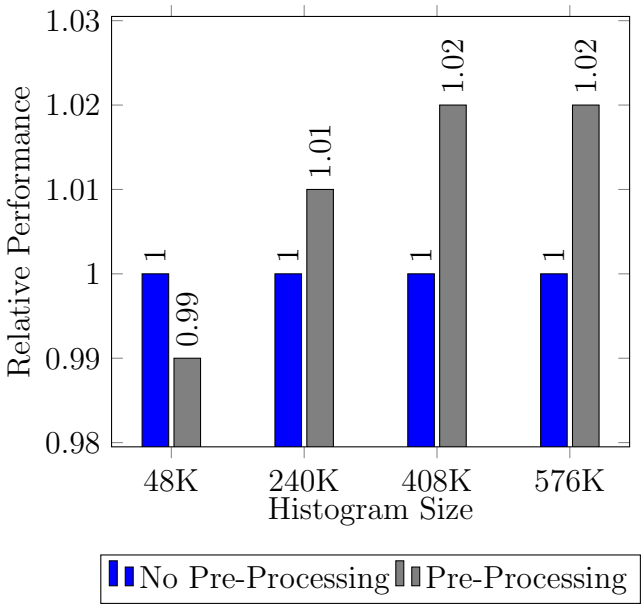


Figure 5.8: Benefits of Extension 3, Gaussian Distribution of Inputs

5.7 Extension 4: Multiple Thread-Blocks per Histogram Segment

For most small histogram kernels, or kernels with unusual histogram sizes, the kernel launched does not always make good use of all SMXs. We need at least 12 thread-blocks in order to achieve 50% occupancy on histogram kernels computing outputs of 48K bins and more, or at least 24 thread-blocks for histogram outputs of 24K bins and less to reach 100% occupancy. Failure to do this leaves CUDA cores idle during kernel execution and, hence, wastes throughput.

To balance out the workload across additional SMXs, horizontal tiling is introduced. Multiple thread-blocks with $blockIdx.x = 0, 1, \ldots, blockDim.x - 1$ will produce sub-histograms for the same vertical tile. Hence each sub-histogram only covers a fraction of the input, as per the code in Figure B.5 of Appendix B.

The benefits of launching additional thread-blocks is twofold. While adding additional blocks in the X dimension increases the overall number of global atomic operations at the end of the histogram kernel, in practice the technique does not negatively impact throughput due to the improved global atomic throughput for non-colliding operations. This technique primarily improves resource utilization as

additional SMXs can be utilized to process the input dataset.

Secondly as *blockDim.x* is increased, the kernel divides the inputs into more and more thread-blocks thus decreasing the possibility of overflow events. When a shared memory bin overflows there is a substantial penalty in terms of execution speed – minimizing the occurrences of overflow penalties generally improves overall throughput.

A visual decomposition would look something like Figure 5.9. In this figure, however, the diagram is simplified by assuming a thread-block consists of only 16 threads and four thread-blocks are launched to cover an entire 256 bin image.

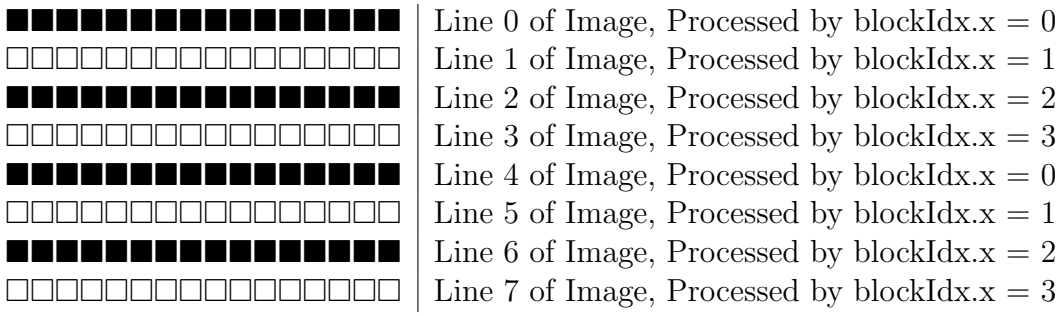


Figure 5.9: Decomposition for Scanning Inputs

The **subdivision** of tasks also had another ancillary benefit. By dividing the work among additional workers, the kernel reduces the potential maximum execution time of any single thread-block. If one particular thread-block is doing more than its fair share of work, that execution time impacts the entire run-time of the kernel. This unfair workload would occur when inputs are somewhat degenerate, thus causing a select group of thread-blocks to have to perform more atomicAdd() operations and correct more frequent overflow events.

The top illustration in Figure 5.10 shows the original program flow without horizontal tiling. A large number of inputs map to the bins assigned to the thread-block within a particular SMX. This in turn causes the thread-block to perform almost half the work given to the entire kernel and drag out execution time for the entire kernel.

The bottom illustration in Figure 5.10 shows how throughput is improved by launching 4X more thread-blocks. While the total execution time is still the same, the additional granularity improves load balancing by minimizing the maximum execution

time of any one thread-block. The GPU scheduler, in turn, can more tightly pack the work and reduce the net execution time of the kernel. This can result in significant improvements in throughput per Figure 5.11.

The divide-and-conquer approach is simply an excellent tool to demonstrate the need to keep SMXs tasked with useful work. In smaller histograms, we see very large speed-ups as the kernel continues to populate the SMXs with thread-blocks. As we approach ($48 \times 1024 \times \texttt{SMX} = 576\texttt{K}$) bin kernels, the benefits of Extension 4 wash out.

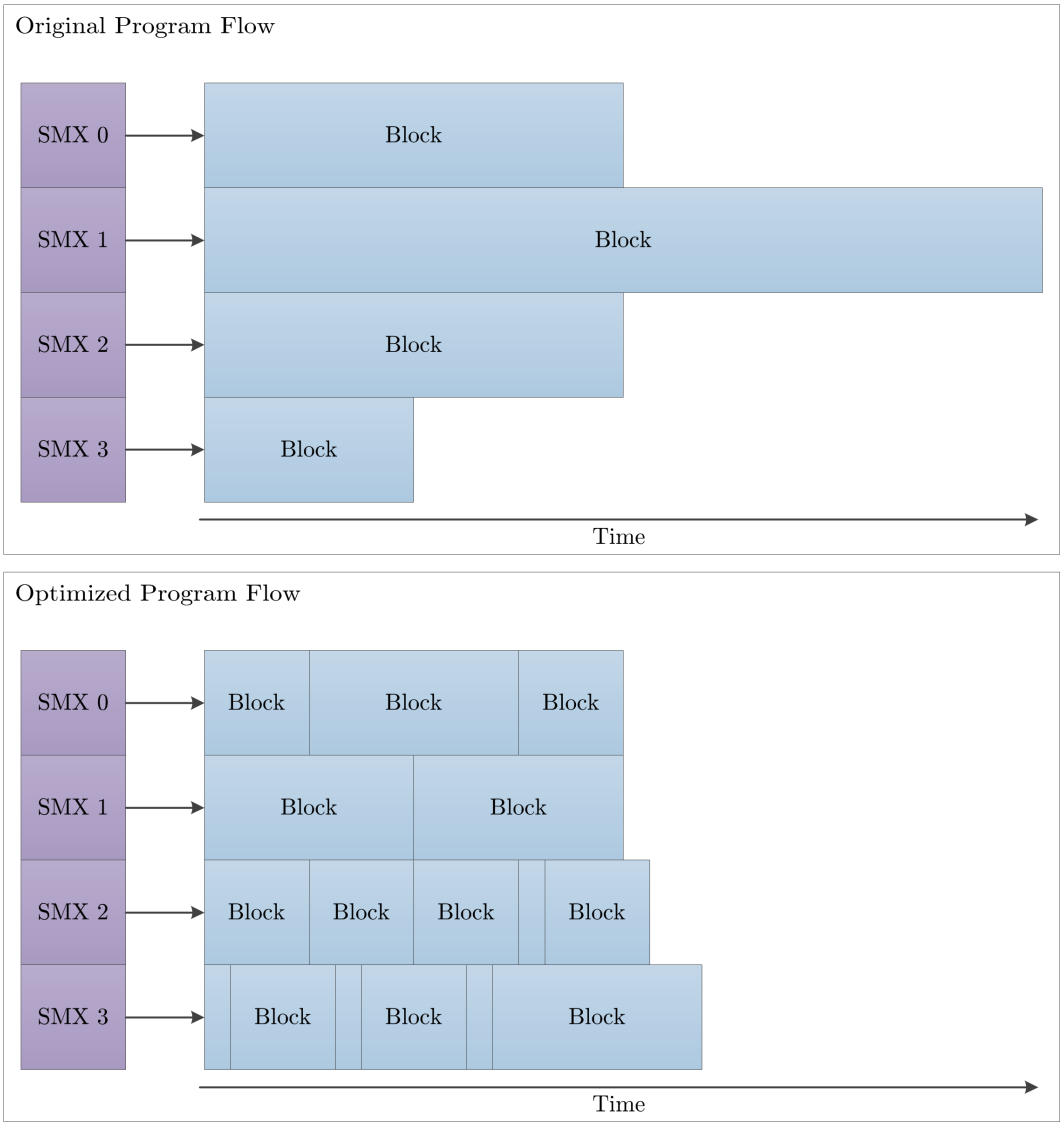


Figure 5.10: Improved Critical Path

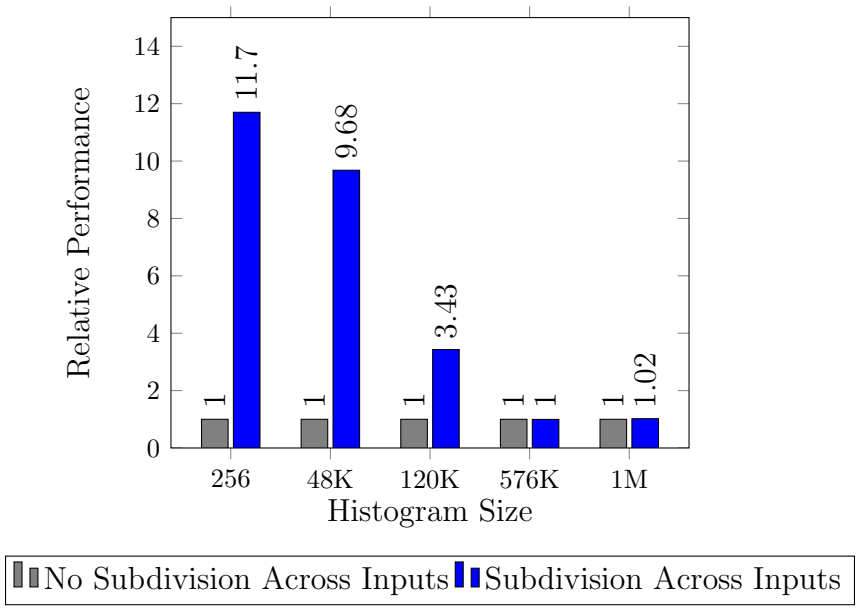


Figure 5.11: Benefits of Extension 4

5.8 Extension 5: Additional Packing, Partitioning, CUDA Streams

As the output histogram increases to 512K bins and larger, performance of the packed atomic histogramming kernel encounters a point where a baseline histogram kernel running on modern hardware has higher throughput. This is not a statement of the inefficiency of the packed atomic histogram kernel, but rather of how NVIDIA has improved the throughput of global memory atomics since the original revision of the packed atomic histogramming kernel was created [23].

While the packed atomic histogramming kernel has a positive effect on throughput by offloading a majority of the atomic operations into shared memory, the kernel also has a negative effect in that it requires increased memory bandwidth to support the additional passes of the input data required to divide the output histogram into tiles.

To this end three more tools are introduced in this section to combat the increasingly negative effects of processing larger and larger histograms.

5.8.1 Increasing Bin Density

Up until now this thesis has covered exclusively "4X" packed atomic operations (four bins per 32-bit word). In an effort to squeeze additional performance out of many-bin histogram kernels, the algorithm is modified to support "8X" packed atomic operations.

Upon revisiting Section 5.3, we derive alternative values for a few key parameters. Under the following assumptions and given a uniformly distributed set of inputs these are parameters that change:

$$
\begin{aligned}
B &= \text{Bits per Bin in 32-bit Word} = 8 \\
N &= \text{Bits per Input} = 22 \\
P(overflow) &= w \times \frac{1}{2^B} \times \frac{1}{2^N} \\
P(overflow) &= 32 \times \frac{1}{256} \times \frac{1}{2\mathtt{M}} \\
P(overflow) &= 0.000006\%
\end{aligned}
$$

Due to the sheer size of the histogram, the probability of an overflow event decreases to the point that we can also decrease B without a substantial risk to throughput. This has the effect of doubling the number of bins that can be stored in shared memory per thread-block, and cuts in half the global memory bandwidth required to support the histogramming kernel. The updated values are as follows:

$$
\begin{aligned}
B &= \text{Bits per Bin in 32-bit Word} = 4 \\
N &= \text{Bits per Input} = 22 \\
P(overflow) &= w \times \frac{1}{2^B} \times \frac{1}{2^N} \\
P(overflow) &= 32 \times \frac{1}{16} \times \frac{1}{2\mathtt{M}} \\
P(overflow) &= 0.000095\%
\end{aligned}
$$

Even though the probability of an overflow has increased by 16X due to packing eight bins into a single 32-bit word, $P(overflow)$ is still improved over the 0.05% calculation outlined in Section 5.3. The relative performance comparison compiled in Figure 5.12 show that this optimization usually, but does not always, result in a net-gain.

Kernels computing extremely large histograms benefit from this extension. The performance figures shown were obtained by varying the number of bits per bin, while leaving other factors constant. The input data given produces the sparsely filled histogram outlined in Section 4.2.

5.8.2 CUDA Streams

CUDA streams are yet another layer of parallelism that can be utilized to improve performance. In Section 5.7, it was illustrated that increasing the number of thread-blocks that handle a specific tile of the output histogram does not significantly improve performance for large histogramming. By using CUDA streams in-lieu of Extension 4, we can still achieve parallel operation across input datasets.

The concept is relatively straightforward. With the ability to execute multiple CUDA kernels simultaneously, the CUDA device will execute as many kernels as it can to fully utilize the SMXs. If one kernel is only fully utilizing six SMXs, and another CUDA stream invokes a second kernel, the CUDA device will start processing that kernel as well by utilizing the available SMXs.

For extremely large histograms, most of the performance improvement comes from one stream starting up with the resources no longer used by the previous stream at the tail end of its execution (remember – a few thread-blocks will likely still be active as most inputs are not going to guarantee perfect load balancing).

The benefits are much like those illustrated in Figure 5.10, except we are discussing CUDA streams across SMXs instead of thread-blocks across SMXs. Experimentally it was determined that launching four CUDA streams produced optimal results for the histogramming kernel. Throughput improvements, shown in Figure 5.13, typically hover around 20%.

When mapping the kernel to the GPU, the host CPU code sets launch parameters such that:

$$\begin{aligned} \text{Streams} &= 1, 2, \ldots, \text{or } 5 \\ \text{gridDim.x} \times \text{Streams} &= \text{Streaming Multiprocessors} \\ \text{gridDim.y} &= \text{Bins/Shared Memory} \end{aligned}$$

This allocation method (1) works well regardless of the histogram size and (2) there are some benefits, however small, to subdividing the inputs across $gridDim.x$.

In particular, the risk shall be minimized that an overflow will occur. For the GTX 780, the host CPU code would launch four streams with the first dimension of the grid size equal to three. The second dimension of the grid size depends on the size of the final histogram.

5.8.3 Partitioning

When the histogram outputs have a known pattern, such as those discussed in Section 4.2, the packed atomic histogram kernel can be optimized further to (1) reduce the total number of passes across the input data and (2) reduce the total number of global atomic operations by launching fewer thread-blocks that ultimately do very little work. Keep in mind that with that particular use-case, the histogram was extremely sparse; only about 0.125% of the output bins were populated. Only a fraction of bins outside of those densely populated area(s) have non-zero values. Launching thread-blocks to cover those tiles of the output histogram simply wastes time and throughput.

Hence we introduce a scheme whereby the kernel is given a region to histogram using its shared memory resources; this information is inferred given a single input parameter. If an input is loaded that is *outside* of the dense histogram region, then the thread will simply increment the bin in global memory space much like the baseline histogram implementation. Keep in mind that usually less than 1% of inputs fall outside of the dense region. In most cases less than 2% of inputs fall outside of the main cluster. Ultimately, the number of passes on the input data required are lessened at the small cost of a few additional global memory atomic operations. For

computations of histograms of any size, Figure 5.14 shows the optimization results in a net-gain.

Without the use of this divide-and-conquer technique, the number of thread-blocks that have to be launched to cover a single histogram is increased by the factor of eight. This in turn places additional burdens on the memory system, reducing the number of kernels that can be run simultaneously via different streams, and reducing overall throughput.

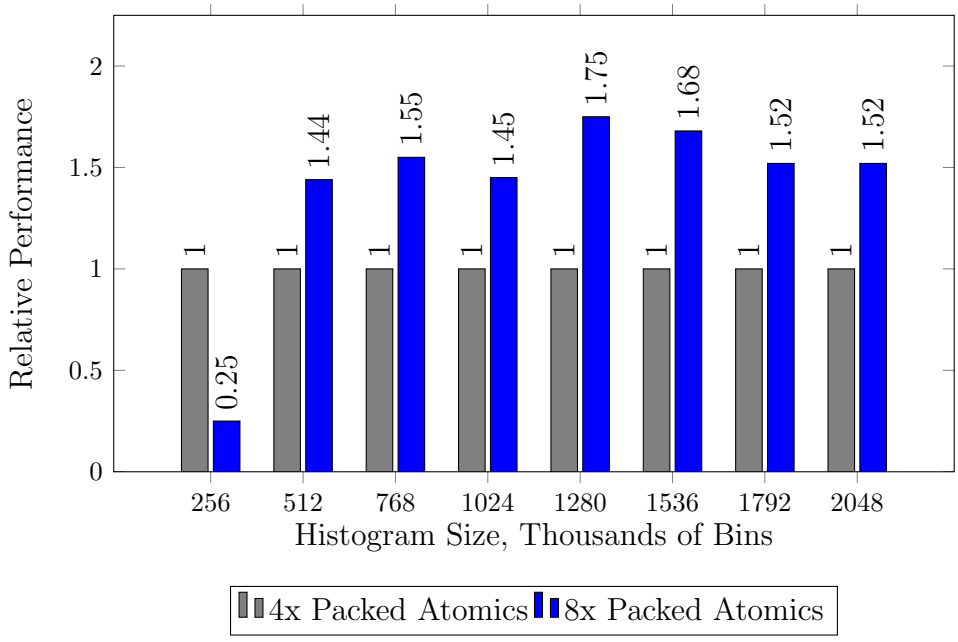


Figure 5.12: Benefits of Extension 5, Packing

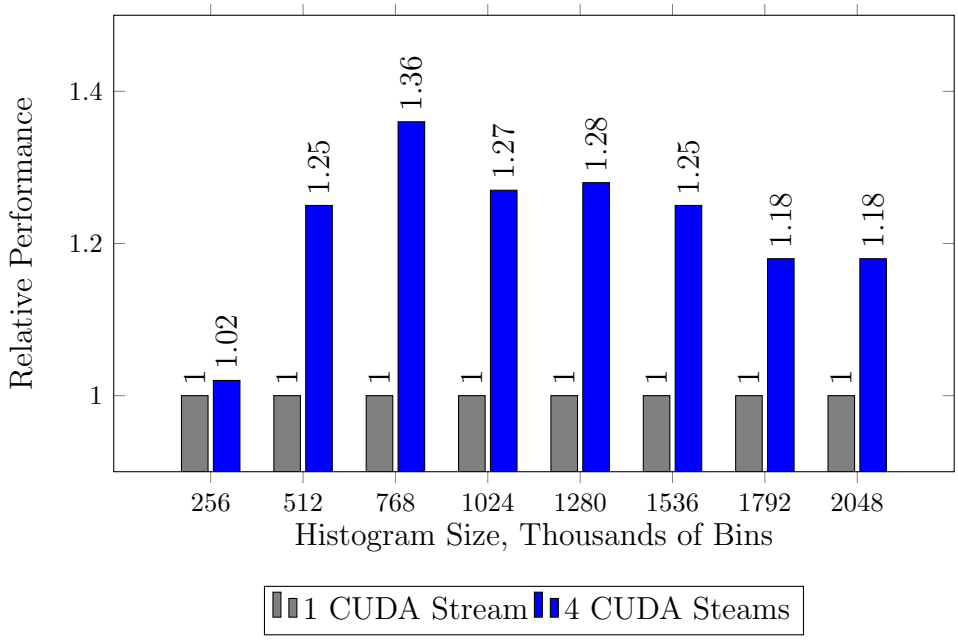


Figure 5.13: Benefits of Extension 5, CUDA Streams

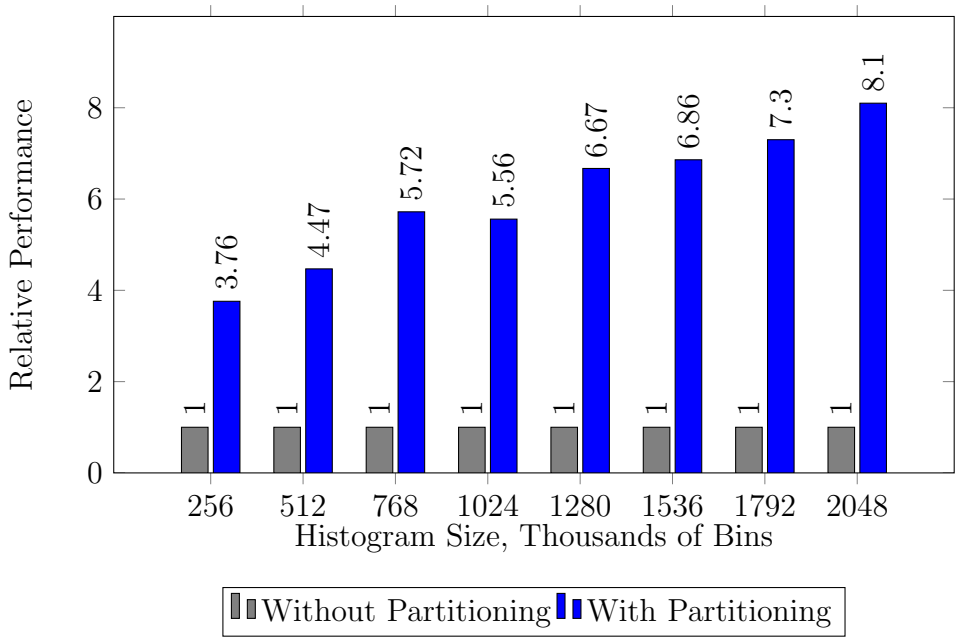


Figure 5.14: Benefits of Extension 5, Partitioning

CHAPTER 6

PERFORMANCE EVALUATION

Having reviewed the histogramming kernel, discussed the extensions to the algorithm, and measured relative performance of those extensions under a variety of configurations, these packed atomic histogramming kernels configurations are used in the final performance evaluation.

Histograms with 256 to 48K Bins

Extension 1: Thread Coarsening Factor Set to 1
Extension 2: Duplication Enabled
Extension 3: Pre-Processing Disabled
Extension 4: Subdivision Enabled
Extension 5: CUDA Streams Disabled

Histograms with 48K to 256K Bins

Extension 1: Thread Coarsening Factor Set to 4
Extension 2: Duplication Not Possible
Extension 3: Pre-Processing Disabled
Extension 4: Subdivision Enabled
Extension 5: CUDA Streams Enabled

Histograms with 256K+ Bins

Extension 1: Thread Coarsening Factor Set to 4
Extension 2: Duplication Not Possible
Extension 3: Pre-Processing Disabled
Extension 4: Subdivision Enabled
Extension 5: CUDA Streams Enabled

6.1 Reference Implementations for Comparison

When evaluating the packed atomic histogramming kernel, two reference implementations are included in the comparison. When possible we optimize these kernels by using thread coarsening (Extension 1) and subdivision of inputs (Extension 4).

6.1.1 Baseline Histogram

The baseline implementation establishes a minimum performance metric on the GPU in which all kernels can be compared to. Each thread performs coalesced reads from the input stream, atomically increments the appropriate bin in global memory, and exits. This kernel takes advantage of the improved global memory atomic capabilities on the GTX 780 [17]. As the packed atomic kernel moves toward computing very large output histograms, this kernel becomes the primary competition. In the worst case scenario this kernel can execute at 1.0 Gpixel/s and has been demonstrated to be capable of processing as much as 12.0 Gpixel/s on Kepler.

6.1.2 Shared Memory Histogram

This reference implementation is a simplified version of the packed atomic histogramming kernel. Each thread performs the same histogramming but utilizes 32-bit shared memory bins to take advantage of shared memory atomic operations. When the histogram size exceeds 12K bins, the maximum supported by the standard shared memory histogram kernel, we augment this kernel using the tiling technique so that we can continue to measure this kernel's performance.

Testing demonstrates that the shared memory histogramming kernel achieves a peak throughput of approximately 21.0 Gpixels/s on Kepler.

6.2 Image Processing

When it comes to processing 256 bin histograms with 8-bit image data as inputs, the packed atomic histogram kernel on Kepler achieves higher throughput than the reference kernels on a Kepler GPU. The kernel is on average 6.4X faster than the baseline implementation and 7.9X faster than the shared memory histogram implementation. Performance tends to increase as the size of the input datasets increase, but the relative performance delta is maintained as shown in Table 6.1.

The data in Table 6.2 illustrates a similar story for the packed atomic histogramming kernel and the shared memory histogramming kernel on a Fermi GPU. The packed atomic histogramming kernel on average performs 7.5X faster than the better reference implementation. Note the very weak showing from the baseline global memory histogramming application. The Fermi architecture has weak global atomic throughput, which significantly impacts the results throughout this chapter.

On both architectures this kernel is making the most use of the shared memory atomic resources. We can infer some architectural differences between the two cards from the data given. For one, the Kepler architecture shared memory atomic bandwidth is vastly improved. Despite the fact that the GPU only has 12 SMXs against Fermi's 15, the GPU still manages to roughly double its performance on shared memory kernels compared to its counterpart. The baseline histogram kernel is also vastly improved by roughly 7–8X.

Regardless of the target architecture, the packed atomics histogramming kernel performance is very portable across generations for histograms of this size. Shared memory atomics are virtually guaranteed to be more efficient than global memory atomics in future architectures. Hence, by offloading as much of the processing to the shared memory banks as this kernel does, we ensure a strong performance delta in favor of this design.

Table 6.1: Image Histogram Performance on Kepler, (Gpixels/s)

Image	Baseline	Shared Memory	Packed Atomics	Speedup
Autumn 1080P	6.34	6.69	43.18	6.45X
Beach 1080P	4.92	6.47	26.34	4.07X
Degenerate 1080P	0.96	0.38	6.37	6.64X
Forest 1080P	4.12	3.14	23.18	5.63X
Fractal 1080P	8.12	17.20	55.05	3.20X
ISS 1080P	5.32	5.70	26.23	4.60X
Moon 1080P	1.17	0.47	7.94	6.79X
Random 1080P	10.46	20.96	68.22	3.25X
Autumn 4K	7.62	8.19	52.90	6.46X
Beach 4K	5.59	7.13	28.30	3.97X
Degenerate 4K	1.03	0.39	6.63	6.44X
Forest 4K	3.95	3.20	24.69	6.25X
Fractal 4K	8.56	18.56	61.75	3.33X
ISS 4K	5.28	5.62	26.81	4.77X
Moon 4K	1.27	0.49	8.20	6.46X
Random 4K	9.26	21.18	83.32	3.93X

Table 6.2: Image Histogram Performance on Fermi, (Gpixels/s)

Image	Baseline	Shared Memory	Packed Atomics	Speedup
Autumn 1080P	0.85	5.42	27.51	5.08X
Beach 1080P	0.85	4.88	18.59	3.81X
Degenerate 1080P	0.08	0.32	5.90	18.40X
Forest 1080P	0.62	2.60	17.17	6.60X
Fractal 1080P	1.13	13.28	30.94	2.33X
ISS 1080P	0.72	4.23	18.04	4.26X
Moon 1080P	0.10	0.42	7.32	17.40X
Random 1080P	1.10	16.73	36.09	2.16X
Autumn 4K	0.91	6.18	31.40	5.08X
Beach 4K	0.84	5.18	19.48	3.76X
Degenerate 4K	0.08	0.32	6.13	19.20X
Forest 4K	0.61	2.40	17.97	7.49X
Fractal 4K	1.19	13.09	32.75	2.50X
ISS 4K	0.71	4.20	18.74	4.46X
Moon 4K	0.10	0.40	7.58	18.95X
Random 4K	1.18	16.57	42.66	2.57X

6.3 Generalized Use

When the input data clusters is relatively uniform, the packed histogramming kernel achieves higher throughput on architectures without a strong global atomics capability. But beyond that specific case the performance benefits of the kernel largely depend on the distribution of data and size of the resulting histogram.

In Figure 6.1, there is a cutoff point in the Kepler performance chart that shows the packed atomics kernel where is global atomics simply outperform kernels that perform most of the atomic operations in shared memory. This cutoff point is where the cost of additional read operations becomes large enough to cancel out the benefits of more efficient shared memory atomics. Even in the best case, uniformly distributed inputs, NVIDIA's improvements on global atomic throughput outperform other implementations as the histogram size exceeds 96K bins.

The performance curve of the packed atomic kernel is very much the same between the two architectures. The last three quarters of Figure 6.1, from 64K bins onward, shared memory atomic performance becomes the limiting factor of the algorithm. When the output falls below 64K bins on Kepler, the limiting resource is the compute throughput of the CUDA cores.

For the degenerate distribution of Figure 6.2, once again it is seen that shared memory kernels are performance-portable. Global atomic throughput makes a strong showing on Kepler in this use case, with a throughput 4X larger than the competition.

The same pattern is seen in Figure 6.3. In the 64K to 96K bin range, the packed atomics kernel begins to achieve better performance compared to the non-baseline implementation on Kepler. On Fermi cards, and by extension architectures without reasonable global atomic performance, the bin packing technique results in tangible benefits so long as there is sufficient memory bandwidth to support the additional reads of the input data.

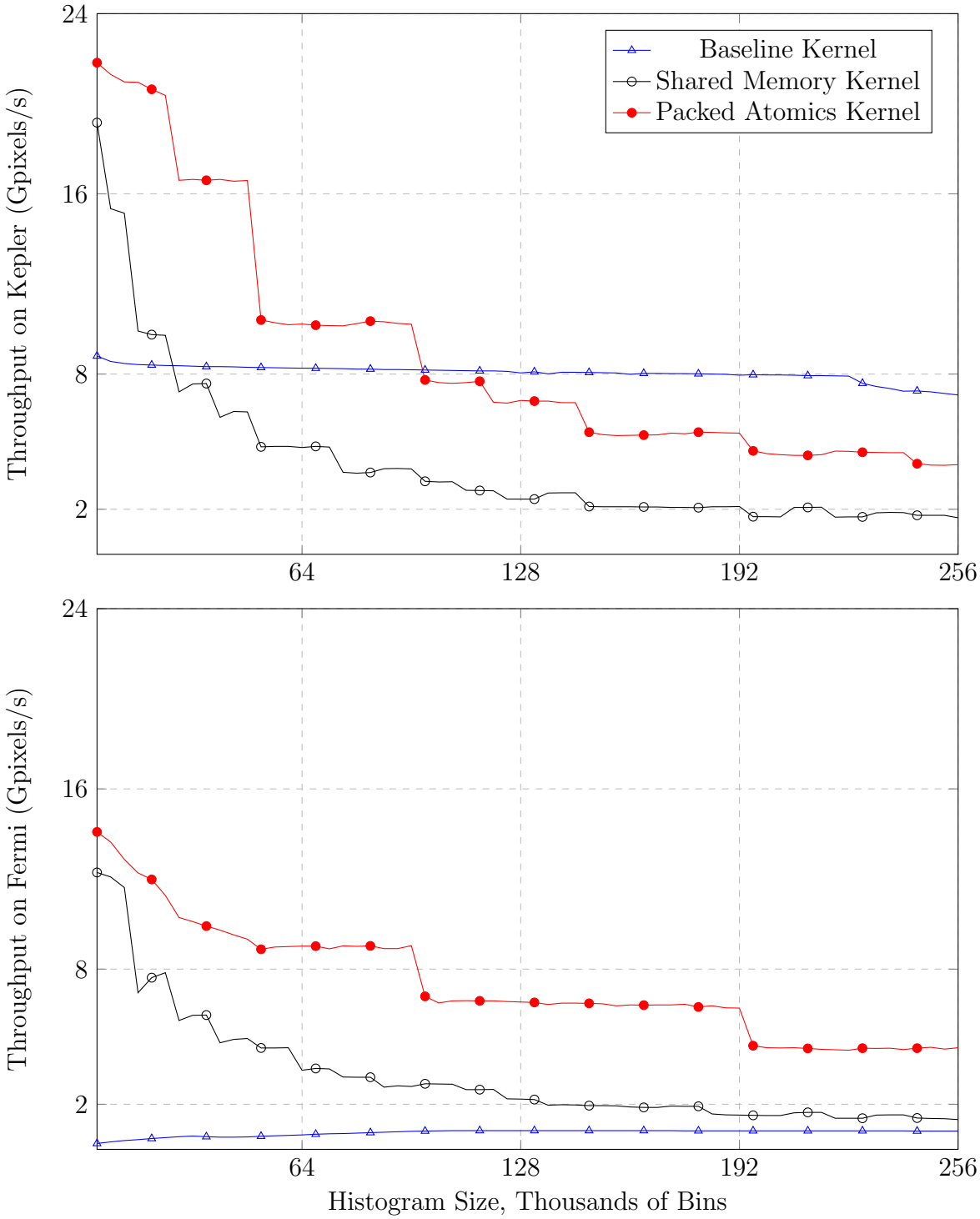


Figure 6.1: Performance with Uniformly Distributed Inputs

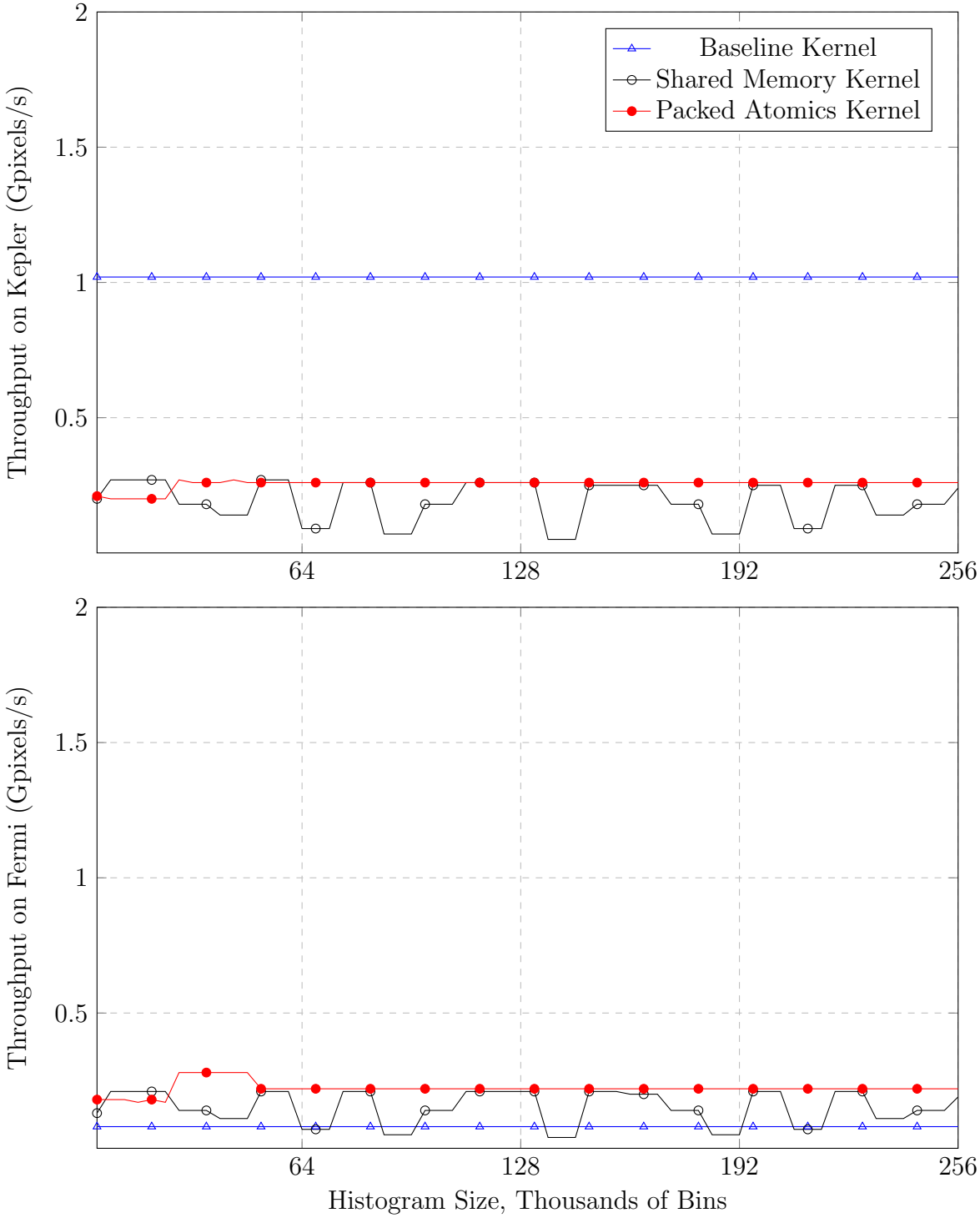


Figure 6.2: Performance with Degenerate Distribution of Inputs

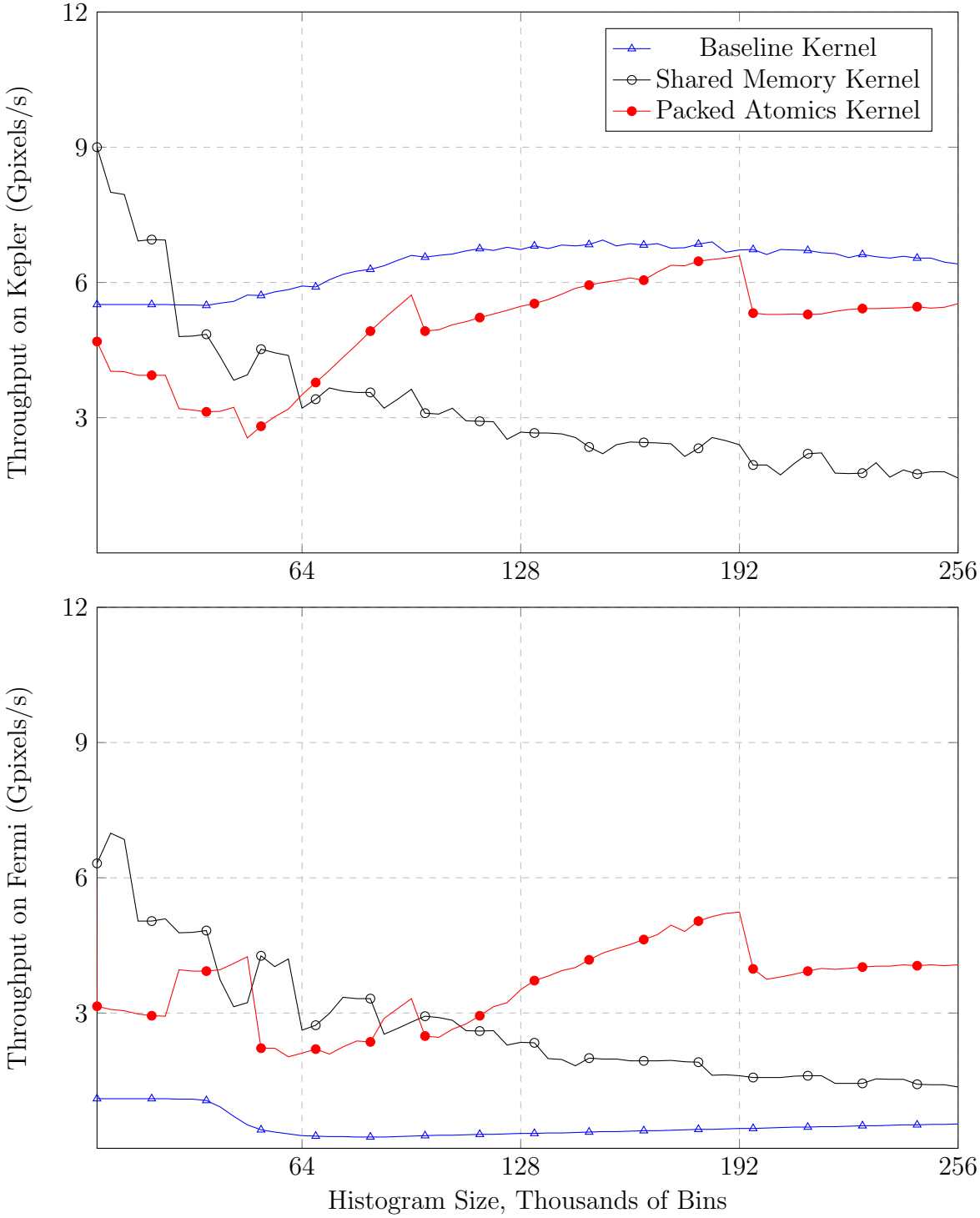


Figure 6.3: Performance with Clustered Distributions of Inputs

6.4 Data Analysis in Wafer Fabrication

In Section 6.3, the trends were clear for Kepler and Fermi architectures. As the histogram size exceeded 96K, the global atomics histogram on Kepler and the packed atomics kernel on Fermi enjoyed the best throughputs. The significant difference between the two trends is that GK110 cards have superior global atomic throughput.

To emphasize this point, we reference Table 4.1 and Figure 6.1. Relative to Fermi, Kepler has almost double the global memory bandwidth but *roughly 8X the global atomics throughput*.

The fact that one characteristic of the GPU was improved by 4X relative to the other essentially cancels out the improvements on throughput the packed atomics histogram kernel demonstrates on the Fermi architecture.

After carefully reviewing Figures 6.4 and 6.5, it is clear that this key architectural difference is responsible for the contrasting results seen in this section.

On Kepler cards, for a majority of inputs, the packed atomics kernel almost keeps pace with global atomic throughput. Because the kernel depends less on global atomics and more on memory throughput, it is possible though speculative that future CUDA designs will shift the balance back in favor of the packed atomics kernel.

Such a shift to favor the design established in this thesis is seen in the previous generation Fermi architecture. The performance ratio between global memory throughput and global atomics throughput is such that either the generic packed atomics kernel (where no selective partitioning of work is done) and the optimized packed atomics kernel (where 87.5% of the atomic operations are atomically updated in global memory) significantly outperforms other implementations. Typically we see 2X to 4X improvements on throughput.

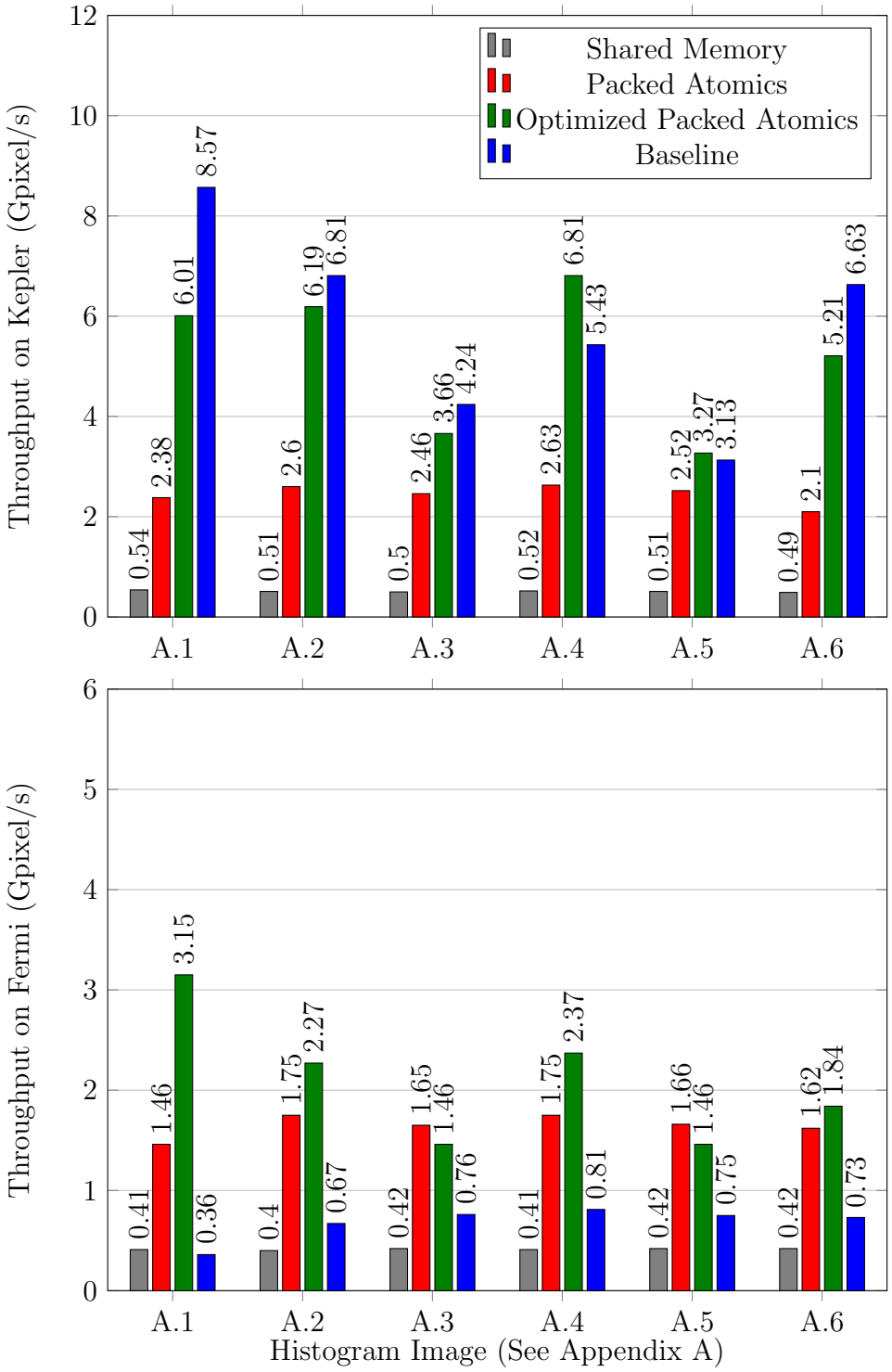


Figure 6.4: Kernel Performance with 1M Bin Histograms

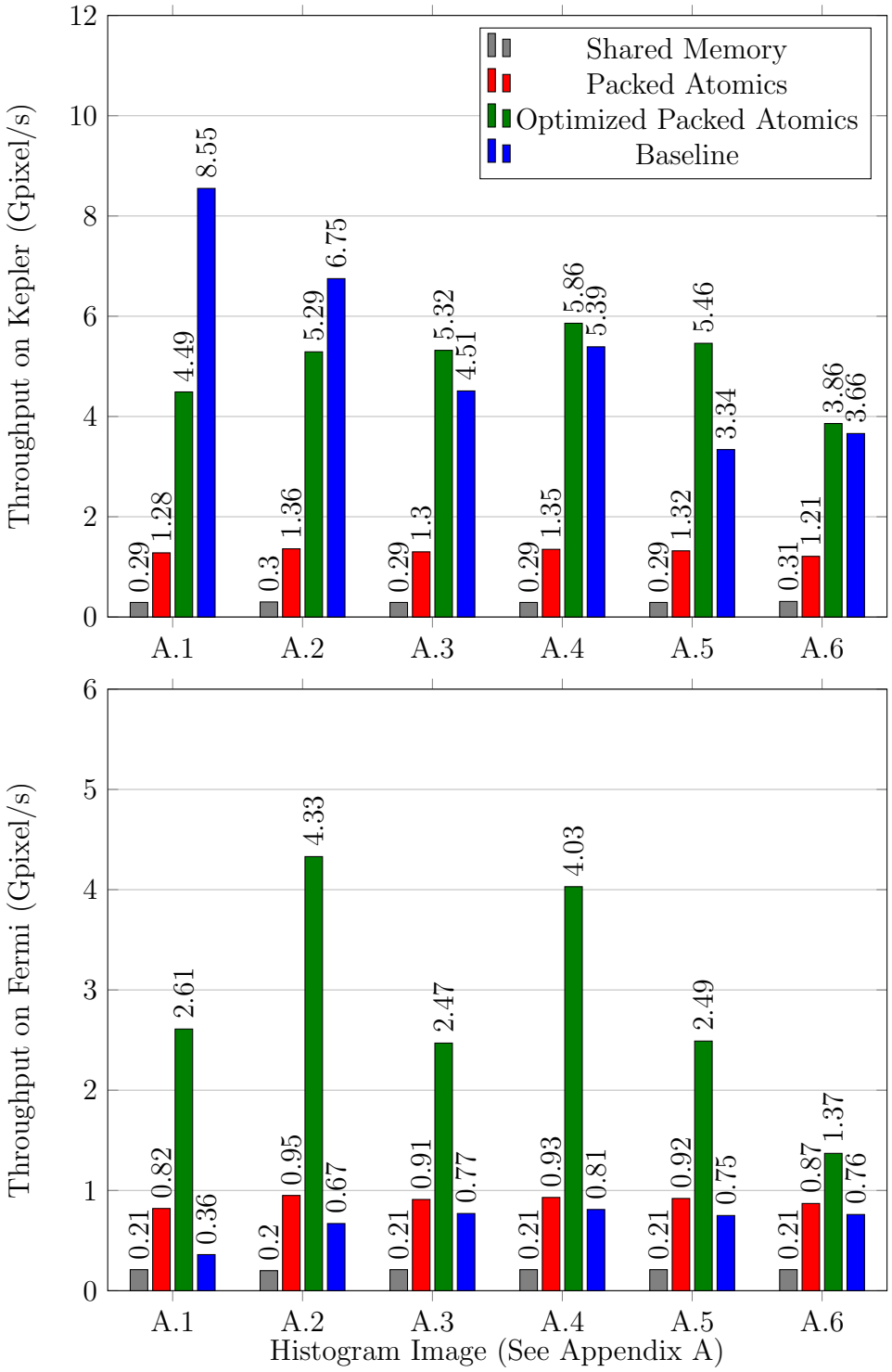


Figure 6.5: Kernel Performance with 2M Bin Histograms

All of these results, however, are very much data-dependent. Take, as an example in Figure 6.6, a set of inputs that are uniformly distributed across a fraction of the bin range. These inputs are more uniform than the histogram datasets previously used. The packed atomics kernel in *this* case outperforms other implementations on Kepler if only barely. The same outcome can also be seen in Figure 5.14 in the Kepler performance results for the inputs from Figure A.5 and Figure A.6 from Appendix A.

Therefore, the primary weakness of the algorithm is input datasets that are more degenerate than not. That having been said, that weakness will be present in any histogram kernel that does not compute sub-histograms on a per-thread basis.

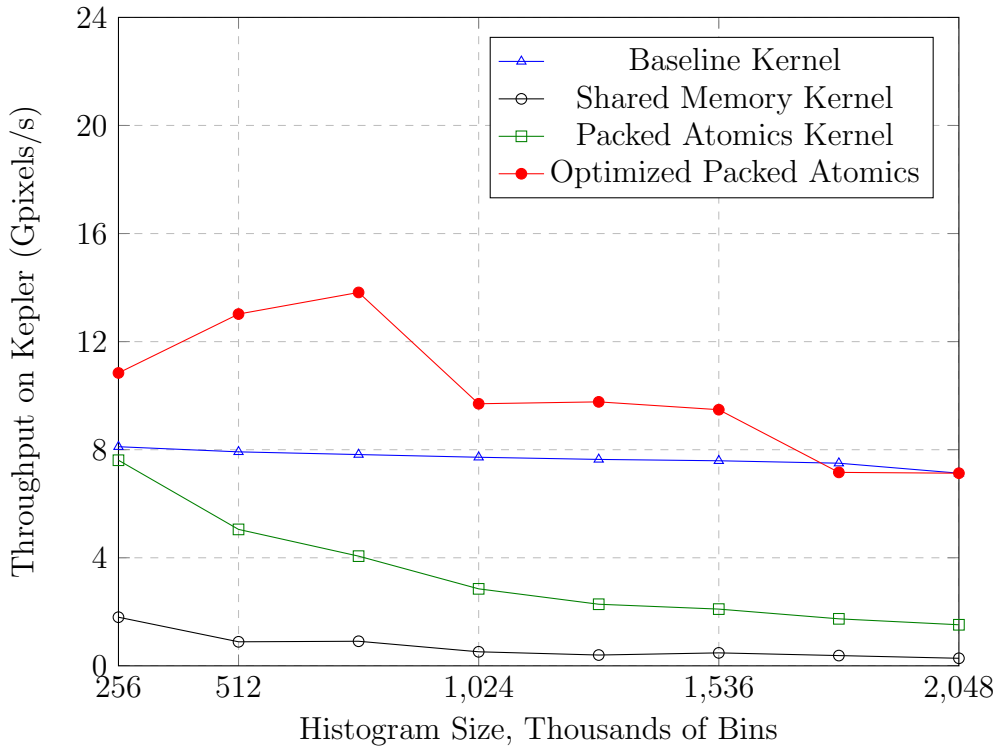


Figure 6.6: Performance with Uniformly Clustered Distribution of Inputs

CHAPTER 7

CONCLUSION

The packed atomic histogramming kernel started out as an optimization to improve very large histogramming kernels on Tesla and Fermi architectures. As summarized in Table 7.1, this thesis demonstrates that this design has several additional uses outside of the original application it was designed for, despite the fact that advances in GPU architecture have allowed naive histogram kernels to catch up with the packed atomic histogramming kernel for very large histograms.

Table 7.1: Summary of Histogram Throughputs on Kepler (Gpixel/s)

	Baseline	Shared Memory	Packed	Speedup
256 Bin, Beach	5.59	7.13	28.30	4.0X
256 Bin, Degenerate	1.03	0.39	6.63	6.4X
256 Bin, Fractal	8.56	18.56	61.75	3.3X
8K Bin	8.56	15.34	21.3	1.4X
32K Bin	8.35	7.56	16.64	2.0X
96K Bin	8.19	3.79	10.21	1.25X
1M Bin, Clustered Input A.1	8.57	0.54	6.01	0.70X
1M Bin, Clustered Input A.4	5.43	0.52	6.81	1.25X
2M Bin, Clustered Input A.1	8.55	0.29	4.49	0.53X
2M Bin, Clustered Input A.4	5.46	0.29	3.34	1.63X

It was originally expected that the kernel would only outperform other histogram implementations in the specific 1-2M bin histogram applications, when the general data distributions are known in advance. Upon further examintion it is clear that assumption is false. This kernel provides means to reduce atomic lock contention via duplication, pack bins together to reduce global memory bandwidth requirements, and also make better use of shared memory thus leading to higher GPU occupancy. All of these traits can benefit applications whereby many contributors are storing data to a relatively small count of outputs.

As with any histogram application, however, the results can be impacted by the data distribution of the input and the GPU architecture. For GPU architectures where atomic throughputs are relatively low (e.g. Fermi), the packed atomics histogramming kernel performs well in most applications. For GPU architectures like Kepler, with improved global atomic throughput, no particular implementation is always going to be the better performer as the histogram size exceeds 100K bins. In a majority of applications, however, the packed atomics histogramming kernel is a strong performer.

REFERENCES

[1] T. Scheuermann and J. Hensley, "Efficient histogram generation using scattering on GPUs," in *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, ser. I3D '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1230100.1230105 pp. 33–37.

[2] V. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU," pp. 451–460, 2010. [Online]. Available: http://doi.acm.org/10.1145/1815961.1816021

[3] W. M. Hwu, *GPU Computing Gems*, Emerald Edition ed. Morgan Kaufmann, 2011.

[4] National Library of Medicine, *ITK API Documentation v4.2.0*, 2012. [Online]. Available: http://www.itk.org/Doxygen/html/classitk_1_1HistogramMatchingImageFilter.html

[5] M. Sundaram, K. Ramar, N. Arumugam, and G. Prabin, "Histogram based contrast enhancement for mammogram images," in *Signal Processing, Communication, Computing and Networking Technologies (ICSCCN), 2011 International Conference on*, July 2011, pp. 842–846.

[6] N. Sengee, B. Bazarragchaa, T. Y. Kim, and H. K. Choi, "Weight clustering histogram equalization for medical image enhancement," in *Communications Workshops, 2009. ICC Workshops 2009. IEEE International Conference on*, June 2009, pp. 1–5.

[7] J. Y. Kim, L. S. Kim, and S. H. Hwang, "An advanced contrast enhancement using partially overlapped sub-block histogram equalization," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 11, no. 4, pp. 475–484, April 2001.

[8] S. Birchfield, "Elliptical head tracking using intensity gradients and color histograms," in *Computer Vision and Pattern Recognition, 1998. Proceedings. 1998 IEEE Computer Society Conference on*, June 1998, pp. 232–237.

[9] D. Comaniciu, V. Ramesh, and P. Meer, "Kernel-based object tracking," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 25, no. 5, pp. 564–575, May 2003. [Online]. Available: http://dx.doi.org/10.1109/TPAMI.2003.1195991

[10] M. Sizintsev, K. G. Derpanis, and A. Hogue, "Histogram-based search: A comparative study," in *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, June 2008, pp. 1–8.

[11] K. J. Yoon and I. S. Kweon, "Artificial landmark tracking based on the color histogram," in *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, vol. 4, 2001, pp. 1918–1923 vol.4.

[12] F. Maes, A. Collignon, D. Vandermeulen, G. Marchal, and P. Suetens, "Multimodality image registration by maximization of mutual information," *Medical Imaging, IEEE Transactions on*, vol. 16, no. 2, pp. 187–198, April 1997.

[13] P. Viola and W. M. Wells III, "Alignment by maximization of mutual information," in *Computer Vision, 1995. Proceedings., Fifth International Conference on*, June 1995, pp. 16–23.

[14] R. Shams, P. Sadeghi, and R. A. Kennedy, "Gradient intensity: A new mutual information-based registration method," in *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on*, June 2007, pp. 1–8.

[15] R. Shams and R. A. Kennedy, "Efficient histogram algorithms for NVIDIA CUDA compatible devices," in *Proc. Int. Conf. on Signal Processing and Communications Systems (ICSPCS)*, Gold Coast, Australia, December 2007, pp. 418–422.

[16] M. Roulo, "KLA-Tencor challenge," 2010. [Online]. Available: http://courses.engr.illinois.edu/ece498/al/mps/mp5/MP5-README.txt

[17] NVIDIA, "NVIDIA's next generation CUDA compute architecture: Kepler GK110," 2012. [Online]. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[18] NVIDIA, "NVIDIA GeForce GTX 680: The fastest, most efficient GPU ever built." 2012. [Online]. Available: http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf

[19] C. Maxfield, "How many silicon chips are there on a 300 mm wafer?" 2007. [Online]. Available: http://www.eetimes.com/author.asp?section_id=14&doc_id=1282825

[20] C. Nugteren, G. J. van den Braak, H. Corporaal, and B. Mesman, "High performance predictable histogramming on GPUs: Exploring and evaluating algorithm trade-offs," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1964179.1964181 pp. 1:1–1:8.

[21] V. Podlozhnyuk, "Histogram calculation in CUDA," November 2007, NVIDIA. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/histogram64/doc/histogram.pdf

[22] S. Koppaka, D. Mudigere, S. Narasimhan, and B. Narayanan, "Fast histograms using adaptive CUDA streams," *CoRR*, vol. abs/1011.0235, 2010.

[23] G. Ross and J. Stratton, "Saturating histogram," 2010. [Online]. Available: http://impact.crhc.illinois.edu/Parboil/parboil.aspx

APPENDIX A

HISTOGRAMS IN DATA ANALYSIS BENCHMARK

The histogram benchmark code provided by KLA-Tencor programmatically creates synthetic inputs to approximate the same characterizations found in real-world data. The actual input data is too large to include in this document; the six sets of input data produce the following histograms.

Each histogram is 256×4096 bins in size, with a majority of the inputs gathering within the centers of the histograms. Input parameters control the general spread of the output data. Marker lines are drawn to indicate the middle 12.5% of the histogram. For histogram benchmarks other than 2M bins, the input data is scaled vertically so that $256 \times height = bins$.

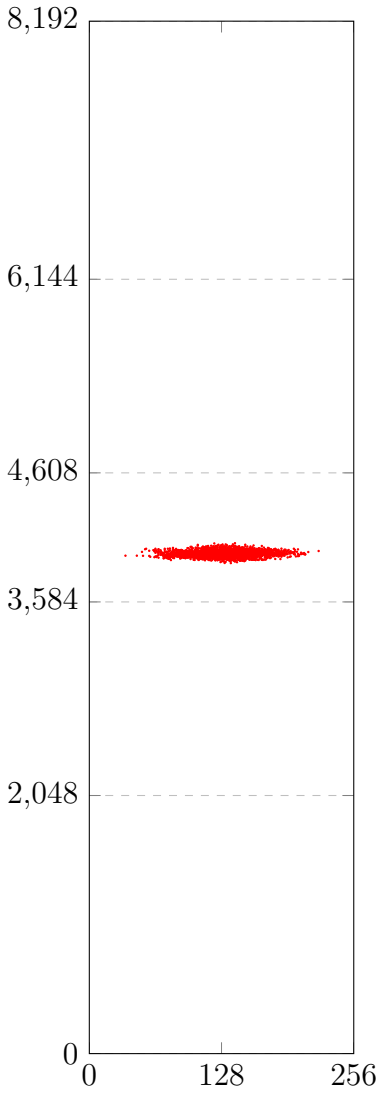


Figure A.1: Histogram 1

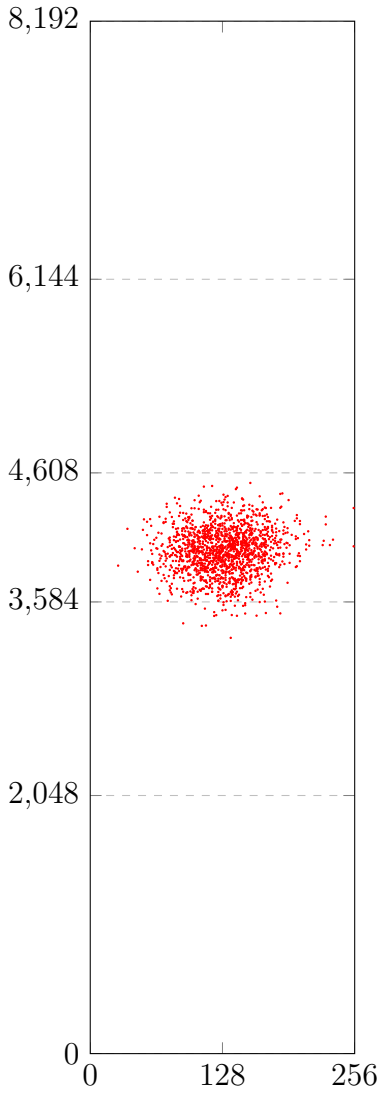


Figure A.2: Histogram 2

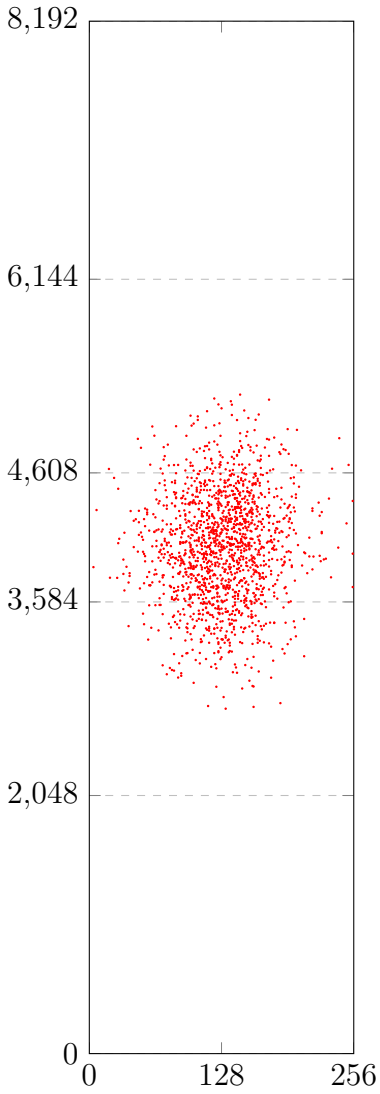


Figure A.3: Histogram 3

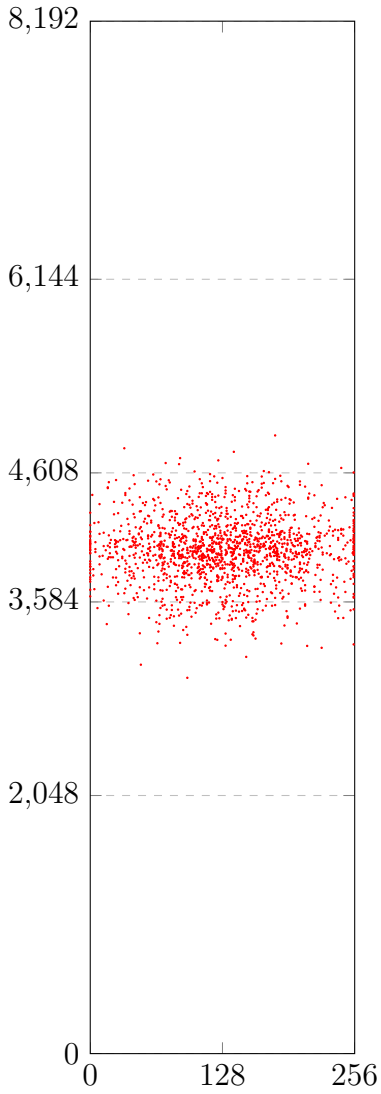


Figure A.4: Histogram 4

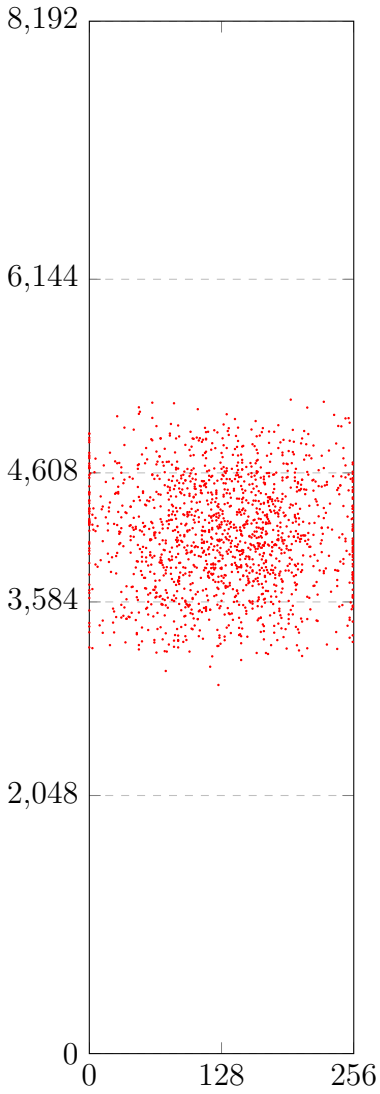


Figure A.5: Histogram 5

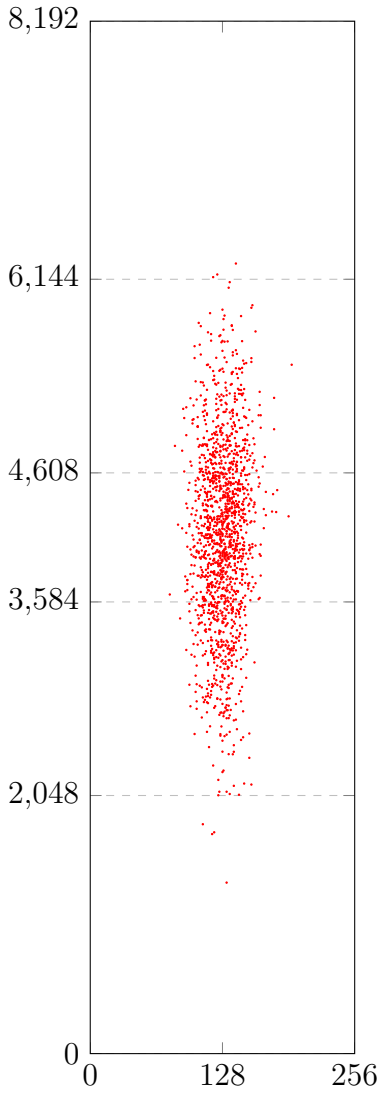


Figure A.6: Histogram 6

APPENDIX B

CODE LISTINGS

```
template<typename INPUT>
__device__ void increment_local (
        INPUT bin,
        unsigned int * global_overflow,
        unsigned int * smem_histo )
{
    const unsigned int index = ( bin >> 2 );
    const unsigned int offset = ( bin % 4 ) * 8;

    /* Atomically increment shared memory */
    unsigned int add = (unsigned int)(1 << offset);
    unsigned int prev = atomicAdd (&smem_histo[index], add);
    unsigned int curr = prev + add;

    /* Check if current bin overflowed */
    unsigned int prev_bin_val = (prev >> offset) & 0x000000FF;

    /* If there was an overflow, record it and record if it cascaded into
       other bins */
    if (prev_bin_val == 0x000000FF)
    {
        /* Code contributing to overflow histogram here */
    }
}
```

Figure B.1: Shared Memory Histogram, 48K Bins and Less

```
template<typename INPUT>
__device__ void increment_local (
        INPUT bin,
        unsigned int * global_overflow,
        unsigned int * smem_histo )
{
    /* ... code ... */
    if (prev_bin_val == 0x000000FF)
    {
        global_overflow += bin_mapping.tile * TILE + ( bin_mapping.index )
            * 4 + ( bin_mapping.offset / 8 );

        prev_bin_1_val = (prev >> (bin_mapping.offset + 8)) & 0xFF;
        curr_bin_1_val = (curr >> (bin_mapping.offset + 8)) & 0xFF;
        overflow_bin_1 = prev_bin_1_val != curr_bin_1_val;
        bin_1_add = (prev_bin_1_val < 0x000000FF) ? 0xFFFFFFFF : 0xFF;

        atomicAdd (global_overflow++, 256);
        if (overflow_bin_1) atomicAdd (global_overflow++, bin_1_add);
    }
    /* ... code ... */
}
```

Figure B.2: Shared Memory Histogram, Method of Correcting Overflow

```
template<typename INPUT>
#define TILE ( 48 * 1024 )

__device__ void increment_local (
       unsigned int bin,
       unsigned int * global_overflow,
       unsigned int * smem_histo )
{
    /*** INEFFICIENT IN CUDA ***/
    unsigned int tile = ( bin / TILE );
    unsigned int index = ( bin - ( tile * TILE ) );
    unsigned int offset = ( index % 4 ) * 8;
    index /= 4;
    /*** INEFFICIENT IN CUDA ***/

    /* Determine if this thread-block should increment */
    if ( tile == <...> )
    {
        /* Atomically increment shared memory */
        unsigned int add = (unsigned int)(1 << offset);
        unsigned int prev = atomicAdd (&smem_histo[index], add);
        unsigned int curr = prev + add;

        /* Check if current bin overflowed */
        unsigned int prev_bin_val = (prev >> offset) & 0x000000FF;

        /* If there was an overflow, record it and record if it cascaded
           into other bins */
        if (prev_bin_val == 0x000000FF)
        {
            /* Code contributing to overflow histogram here */
        }
    }
}
```

Figure B.3: Shared Memory Histogram, 48K Bins and More

```
#define TILE ( 48 * 1024 )
typedef union {
    unsigned int tile : 8;
    unsigned int offset : 8;
    unsigned int index : 16;
} bin_map_t;

static __device__ bin_map_t precompute_bin_mapping ( unsigned int bin ) {
    bin_map_t map;
    map.tile = bin / TILE;
    map.index = bin - ( map.tile * TILE );
    map.offset = 8 * ( map.index % 4 );
    map.index >>= 2;

    return map;
}

__device__ void increment_local_bin_mapping (
    bin_map_t map,
    unsigned int * global_overflow,
    unsigned int * smem_histo ) {

    if ( map.tile == blockIdx.y ) {
        /* Atomically increment shared memory */
        unsigned int add = (unsigned int)(1 << map.offset);
        unsigned int prev = atomicAdd (&smem_histo[map.index], add);
        unsigned int curr = prev + add;

        /* Check if current bin overflowed */
        unsigned int prev_bin_val = (prev >> map.offset) & 0x000000FF;

        /* If there was an overflow, record it and record if it cascaded
           into other bins */
        if (prev_bin_val == 0x000000FF) {
            /* Code contributing to overflow histogram */
            < ... >
        }
    }
}
```

Figure B.4: Pre-Processing Histogram Inputs

```
static __global__ void packed_histogram4 (unsigned int *device_image,
    unsigned int pixels, unsigned int *global_histogram)
{
    extern __shared__ unsigned char smem_histo[];

    unsigned int global_threads_X = blockDim.x * gridDim.x;
    unsigned int global_thread_id_X = blockIdx.x * blockDim.x +
        threadIdx.x;

    unsigned int pixelsPerLoad = sizeof(uint4) / sizeof(unsigned int);

    /*** Code to initialize shared memory ***/
    __syncthreads();

    uint4 * device_image_vector = (uint4 *) device_image;
    for ( int i = global_thread_id_X ; i < pixels / pixelsPerLoad ; i +=
        global_threads_X )
    {
        uint4 data = device_image_vector[i];
        increment_local_bin(data.x, global_histogram, smem_histo);
        increment_local_bin(data.y, global_histogram, smem_histo);
        increment_local_bin(data.z, global_histogram, smem_histo);
        increment_local_bin(data.w, global_histogram, smem_histo);
    }

    __syncthreads();
    /*** Code to save results in shared memory to global histogram ***/
}
```

Figure B.5: Code for Horizontal Tiling