COMPACT BINNING FOR PARALLEL PROCESSING OF LIMITED-RANGE FUNCTIONS

BY

NADY M. OBEID

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, 2010

Urbana, Illinois

Adviser:

Professor Wen-Mei W. Hwu

ABSTRACT

Limited-range functions are domain-level optimizations to a class of applications where all input elements contribute to all output elements, based on the distance between two given elements. When the contribution of an input element to the output is inversely proportional to the distance, a limited range can be applied, which approximates the contribution to zero beyond a certain cutoff distance. Introducing a limited-range function to the application reduces the computation complexity from $O(N^2)$ to $O(N)$.

Processing multiple input elements in a limited-range function in parallel can lead to data races without the use of expensive synchronization. That is why a preferred approach is an output-driven one, where multiple output elements are processed in parallel, all sharing the input data set for reads. Typically the input data set is unstructured, which without the use of binning, would result in every output element in the output-driven approach reading all of the input elements to determine which ones fall within its cutoff. Binning is a preconditioning step that sorts the input elements into predetermined bins that are easily accessible by the output, thus allowing the output to only access the bins relevant to its computation.

Traditionally, bins were created with uniform size and capacity to enable easy access to them; however, making the bins regular can have severe side-effects on memory requirements to maintain these bins. We propose a technique to allow the bins to vary in capacity in order to reduce the memory overhead, at the cost of added accessing overhead. In this work, we will compare regular binning and our approach, compact binning. We will demonstrate that compact bins can in fact improve the execution performance of limited-range functions executed in parallel.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my adviser Wen-Mei Hwu for his constant support and guidance. He has truly influenced and motivated my work, and has helped me grow professionally in the two years he has been my adviser. Secondly, I would like to thank Daniel Liu who helped a lot with the execution of this work. I would also like to thank Ian Atkinson whose collaboration on an MRI project led to the inception of this work.

Over the last two years, I have met a lot of people who have really influenced my work. I would like to thank I-Jui Sung who I shared a cubicle with those two years. The countless discussions I have had with him have challenged me to think outside the box and as a result improve the quality of my research. I also want to thank all the members of our research group who have lent their support countless times. In no particular order, thank you to: Chris Rodrigues, Sara Baghsorkhi, John Stratton, Alex Papakonstantinou, Xiao-Long Wu, Victor Huang, Deepthi Nandakumar, Hee-Seok Kim, Nasser Anssari, Li-Wen Chang, Tim Wentz, and Steven Wu. And of course, nothing would be possible without the help of our tremendous staff who were always very helpful. Thank you to: Marie-Pierre Lassiva-Moulin, Laurie Talkington, Andrew Schuh, Umesh Thakkar, and Xiaolin Liu.

A good work/life balance was crucial to maintaining my sanity, so I would like to thank all the friends who stood by me in the last two years. And last but not least, I would like to thank all my family, in particular, my Mom, my Dad, and my sister Nay.

Thank you everyone.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

With the advancement and ubiquity of high performance computing, applications from various scientific domains have emerged that try to model and simulate the interactions between large sets of elements in physical systems. These applications seek to measure anything from the gravitational forces between many bodies of mass to the electric field in space due to the presence of charged atoms, or even signal propagation between any two points in a space. This information is simulated by measuring the interactions between every pair of points in the system. For example, in order to determine the electric field at a certain point in space, we need to compute the electric field effect of every atom in the space onto that point, and similarly every atom in the space contributes to the electric field of every point in the space. However, it is generally the nature of these interactions that the effect of one element on another decreases as the distance increases between them. Since the computation required to simulate these $O(N^2)$ systems for large data sets is very expensive, scientists often take advantage of the decreasing effect to accelerate the computation. They do so by neglecting the interactions between two elements when the distance between them causes the effects on one another to be insignificantly small. In other words, they approximate all the effects beyond a certain cutoff distance to be zero. In some application, the distant contributions are computed using a different method. By doing so, they reduce the complexity of the algorithm from $O(N^2)$ to $O(cN)$ where c is the constant-sized cutoff distance beyond which no interactions are computed. Applying a cutoff to the computation in order to reduce the complexity of the algorithm results in what we define as a limited-range function, because, as the name

suggests, we confine the effects of each input element to a limited set of output elements that fall within its range.

Typically, the input data to these applications are irregularly distributed and do not follow any uniform distribution pattern, and that may be a result of the way these input elements are collected (e.g., samples collected by an MRI scanner), or simply the natural distribution of these elements in their medium (e.g., atom cloud in space). On the other hand, when simulating or processing these input elements we often wish to compute their effects on a regularly structured output set (e.g., the electric field at every point in a regular grid) where the output data set is much larger than the input data set. These properties are true for all of the applications we analyze in this work, except for one where the input element set and the output element set are the same, and both are irregular. The relative sizes of the input and output, the regularity of the output, and irregularity of the input are necessary considerations when optimizing the computation of these systems.

1.1 Sequential Implementation

When computing limited-range functions on a CPU, the program iterates over all the input elements, and computes the contributions of each input onto the output elements. Because the inputs are not ordered in any uniform way, their location in the space cannot be inferred or computed. Instead, each input element holds its own coordinates explicitly. On the other hand, if the output is a regular grid, the coordinates of every output point can be computed. That is why it often makes more sense to take an input driven approach rather than an output driven one when computing limited-range functions. Based on the input's coordinates, a neighborhood is determined by computing a sphere centered at the

input's coordinates with a radius equal to the cutoff distance. Every output element that intercepts this neighborhood region is therefore a neighbor of the input element and is contributed to by this element. Figure 1 shows a two-dimensional example of a neighborhood around one of the input elements. Since the neighboring output to a given input point can be predetermined, it is unnecessary to visit any output elements that fall outside of the neighboring region. Multiple input elements may contribute to the same output point, as shown in Figure 2; however, since the processing of input elements is done sequentially, no update conflicts occur.

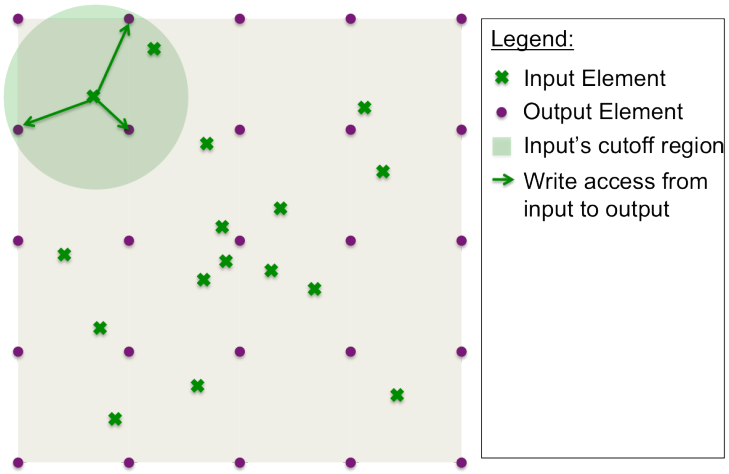


Figure 1. Depiction of sequential execution of limited-range functions

1.2 Parallel Implementation

Limited-range functions are inherently parallel. Every input element computes its contributions to the output independently from other input points. Similarly, every output point can be computed independently of all other output points. However, several factors can hinder their performance on parallel architectures. For instance, if we were to naively port the input-driven sequential algorithm to a parallel execution model, one of the biggest problems we face is write contention by input elements onto the output. Specifically, if all input elements are processed in parallel, inputs attempting to update the same output

element may suffer from data races, leading to incorrect results. The two input elements highlighted in Figure 2 may suffer from a data race if they both attempt to update their shared output simultaneously (contention shown in red). Since updating an element requires multiple instructions (read, modify, write), data races occur when the update instructions of one processing thread are interleaved with the update instructions of another thread, causing one of the threads' updates to be lost. One way to avoid data races is to make updates atomic, that is, guarantee that the three instructions from one thread cannot be interrupted, and that a processing thread cannot start updating an element until another thread that is already in the process of updating that element has finished. However, ensuring this synchronization is costly and can deteriorate the computing performance, especially when several threads try to simultaneously update the same element, since atomicity causes threads' updates to be serialized.

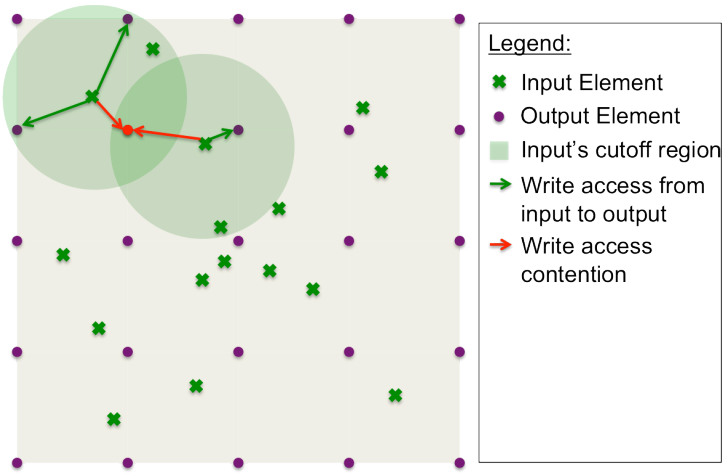


Figure 2. Parallel implementation of the scatter approach

Another way to avoid data races is to privatize each output to a single writer: instead of having each thread compute the contributions of an input element onto all the neighboring output elements, we let each thread compute exclusively the value of an output element by calculating the contributions of its neighboring input elements. The

definition of the neighborhood is the same for the input-driven approach as well that the output-driven approach, since the cutoff distance is the same whether seen from point A to point B or B to A. By privatizing the output among the threads, multiple output may end up reading the same input elements (Figure 3); however, since read accesses do not modify the input elements' values, no synchronization is needed. This output-driven approach is called a "Gather" approach whereas the input-driven one is called a "Scatter" approach. The names are symbolic of the methods of computation: gather is a collection of multiple input contributions onto one output element, whereas scatter takes one input and generates its contribution onto multiple outputs.

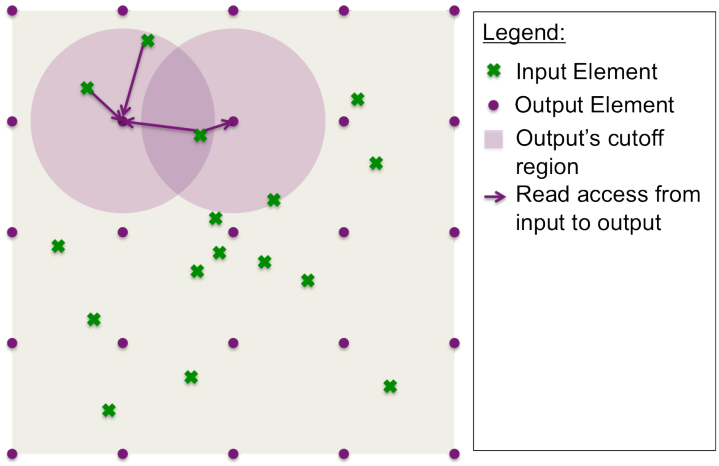


Figure 3. Parallel implementation of the gather approach

One difficulty that arises with the gather approach is that, as we mentioned earlier in this chapter, input elements are typically unstructured, and need to explicitly maintain their coordinate information. As a result, every output element has to iterate over all the input elements and determine for each whether they fall within its cutoff distance before computing their contributions, as seen in Figure 4. Having to evaluate all the input elements negates the benefits of introducing a cutoff in the first place, as the resulting algorithm once again becomes $O(N^2)$.

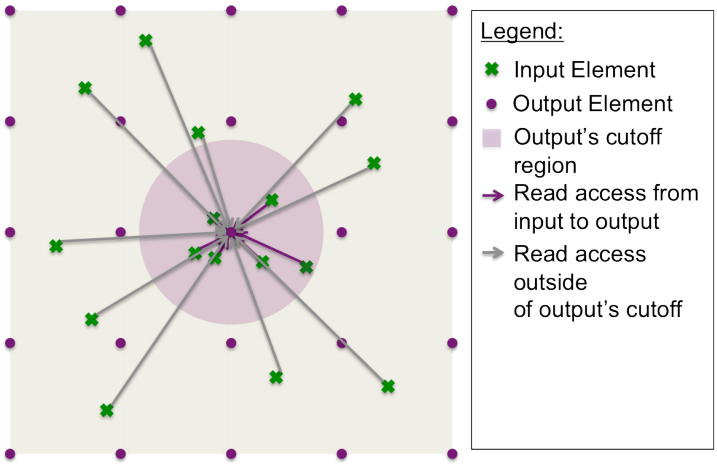


Figure 4. Side effects of the gather approach without binning

1.3 Parallel Implementation with Binning

Binning is one technique we can use to reduce the complexity of a gather algorithm from $O(N^2)$ back to $O(N)$. A bin is a container corresponding to a sub-region of the total space containing all of the input elements that fall within this space. These containers have known characteristics, such as the size of the sub-regions they cover and their element capacity, and this makes them easier to access than individual input elements. We enable easy access to input elements by placing them within the bins. Instead of each output element having to traverse the array of all the input elements, it only needs to access the bins that fall within its cutoff to get to the neighboring input elements. Performing binning on the input data reduces the complexity of the computation from $O(N^2)$ back to $O(N)$. Figure 5 depicts the execution of the gather approach with binning. Note that some elements that fall within a neighboring bin may not themselves be neighbors of the output element, so it is still necessary to calculate their distance from the output before computing their contribution. In fact binning cannot completely prevent an output from reading input

elements that are outside of its cutoff region, but it can reduce the number of these occurrences significantly.

One simple way to make all the bins easily accessible is to make them all identical. That includes making all the bins represent an equal portion of the space (size), as well as making each bin contain the same number of elements (depth). By doing so, the starting index of every bin within the data structure containing the bins can be computed using the index of the sub-region that bin represents and the capacity (or depth) of each bin.

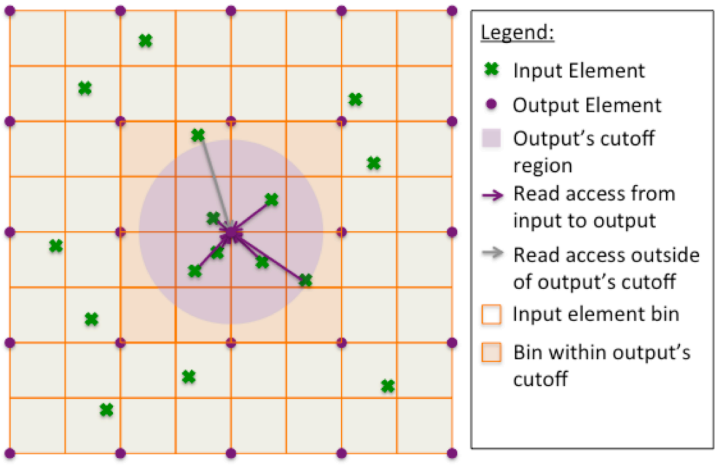


Figure 5. Gather implementation with binning

Assigning an equal portion of the space to each bin can be achieved (assuming a regular space) by simply dividing the total region evenly among all the bins. Guaranteeing that each bin contains the same number of elements, on the other hand, is a more challenging task, since the number of elements that go into a bin is dependent on the input data, and can vary from one dataset to another. One way to achieve uniform bin capacity is to make every bin contain as many elements as the largest bin. In other words, we determine the maximum capacity required by any bin, and make the capacity of all the bins be equal to that maximum. In a situation where the elements are evenly distributed in the space, and every sub-region contains the same number of elements, the maximum capacity

will be the same as the average capacity. Figure 6 is an example of a uniformly distributed input in 1-D space. The integer shown for each element in the input array corresponds to the bin that element belongs to. As we can see, every bin contains exactly two input elements, which makes it easy to achieve uniform bin capacity. However, as soon as the number of elements in each bin starts to vary, maintaining a uniform size for all the bins will require padding for the bins that have fewer elements than the maximum capacity. Padding is the use of mock elements in every bin to make up for the missing elements when the number of real elements in the bin is smaller than the maximum bin depth. If the fifth element in the array from Figure 6 were a 3 rather than a 1 (shown in Figure 7), the bin depth would no longer be uniform, which means that in order to maintain a uniform depth in all the bins, we would have to pad all the bins than have fewer than three elements in them (shown as "X" in Figure 7). In essence, padding makes all the bins equal in capacity at the cost of increasing the memory requirement for these bins by introducing dummy elements into the bin array. These dummy elements are not computed for when an output reads a bin, since they do not represent real input elements.

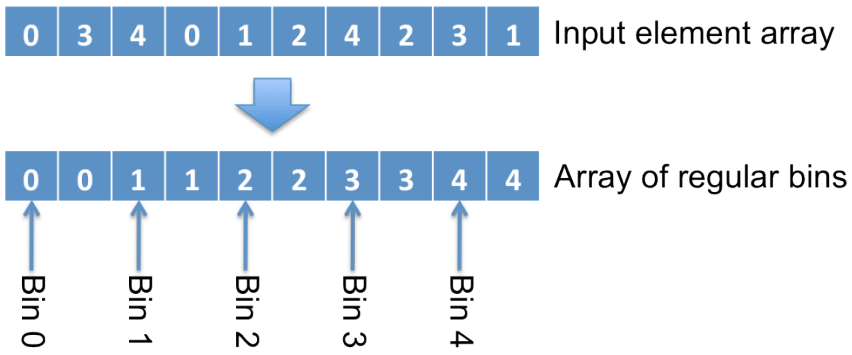


Figure 6. Example of regular binning with uniform distribution

Regular-sized binning has a space complexity O(CB), where C is the capacity of every bin and B is the total number of bins. Increasing the maximum capacity by 1

increases the amount of space needed for the bin data structure by B elements. This becomes increasingly expensive as the disparity between the average capacity and maximum capacity increases (Figure 7). When B and C both become very large, the strain on the memory due to binning may become the limiting and sometimes disabling factor in performing the computation. We will demonstrate such cases in our benchmarks. The motivation of this work is to come up with a solution that makes binning a feasible solution even for highly unbalanced problems.

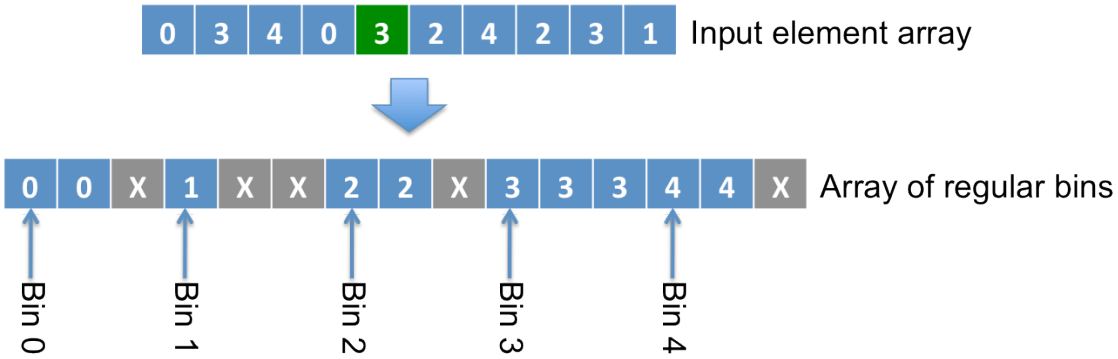


Figure 7. Example of regular binning with non-uniform distribution

1.4 Parallel Implementation with Compact Binning

In this work, we propose compact binning, a method of performing binning with a space complexity of O(N), where N is the number of input elements, independently of the number of bins and the capacity of each bin. The main idea behind compact binning is to allow each bin to have its own bin depth regardless of all the other bins. As a result, we eliminate the need for padding, and the size of the bin data structure becomes only as large as the number of input elements (Figure 8). The variable bin depth and elimination of padding come at the expense of more complicated access methods to these bin. Since the size of each bin is independent of all the other bins, accessing a bin can no longer be

computed as a function of the bin index and the bin capacity. Therefore, additional overhead is incurred in trying to determine the starting offset of each bin. The added overhead stems from the need to pre-compute the starting index of every bin and store it in an array which will then be used as a look-up table when trying to access the bins during the limited-range function computation.

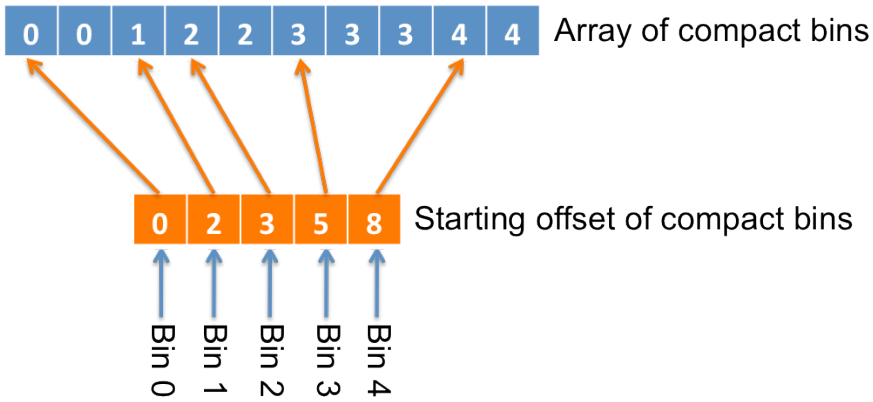


Figure 8. Example of compact binning using the input array from Figure 7

In reality, when the input data is highly non-uniform, it is advantageous to partition the input across multiple data structures. In the case of regular binning, since the space requirement is a factor of the number of bins and the capacity of each bin, programmers often place a cap on the bin capacity to reduce the size of the bin array. Bins that exceed this cap size "spill over" their excess to another data structure. Bins that have fewer elements than the cap are still padded to achieve regularity. In this situation, the cap size is chosen to maximize the number of elements that get placed in bins, while simultaneously balancing the amount of padding required. However, when the variance from the average bin depth becomes too large, there may no longer be a bin depth that maximizes the number of input elements in the bins without incurring a large overhead.

Though padding is not a concern for compact binning, partitioning the input data across the bins and the spill-over array can in fact improve the overall performance due to better load balance among bins. In Chapter 5, we discuss a method for partitioning the input data and examine how varying the cap value affects the execution in the regular and compact binning cases.

The remainder of this work will be dedicated to comparing regular and compact binning in the context of limited-range function applications. We will discuss the different methods of implementing each type of binning and will evaluate their effect on four different applications each with a different input distribution pattern: MRI gridding, cutoff Coulombic potential, Blinn's blob, and N-body simulation. All except for the last application have non-uniform input data, with varying degrees of non-uniformity, and a uniform output grid. In the case of N-body, the input data set is also the output data set, and therefore both are non-uniformly distributed within the space; however, we will demonstrate how our technique of compact binning can still applied to this application without hurting its performance on GPUs.

We will not, however, discuss in this work when to use cutoff and how to determine an appropriate cutoff distance, since cutoff is a domain level optimization and not a programming optimization. In other words, cutoff is a property of the application's domain and is introduced as an optimization to the computation only when some loss of accuracy in the output can be tolerated. If no loss of accuracy can be tolerated by the application, the programmer cannot choose to introduce a cutoff as a programming-level optimization. For that reason, we will be comparing the use of regular and compact binning assuming that the application allows the use of a limited-range function.

The remainder of this work will be organized as follows. Chapter 2 will describe GPUs, the architecture on which this work was conducted. Chapter 3 will describe the four applications used for the analysis of this work. Chapter 4 will discuss the trade-offs between regular and compact binning. Chapter 5 will discuss partitioning as an orthogonal optimization to binning. Chapters 6 will discuss the similarities between limited range functions and the different representations of sparse matrices in the linear algebra domain, and Chapter 7 will conclude the work.

CHAPTER 2

THE GPU ARCHITECTURE

As shown in Chapter 1, limited-range function applications are inherently parallel since the computation of each input's contributions to the output set is independent from all other input points, and similarly, the computation of each output element based on the inputs' contributions is independent from all other output points. The amount of parallelism in these computations is on the order of the number of input elements for the scatter approach and the number of output elements for the gather approach. This large amount of parallelism makes limited-range functions a good fit for massively parallel architectures and though the techniques we describe in this work can be applied to any parallel architecture, they are best suited for these kinds of architectures that execute many fine-grained threads simultaneously. The architecture we focus on is a graphics processing unit (GPU), more specifically, the NVIDIA GTX280 GPU. In this chapter, we will describe the details of the architecture and the programming model as relevant to this work. Full details on the GPU devices and their programming model can be found in the Programming Guide published by NVIDIA [1].

2.1 CUDA Programming Model

Compute Unified Device Architecture (CUDA) is the programming language used to program NVIDIA GPUs. CUDA is based on the C programming language, with added constructs to explicitly describe parallelism. The explicit parallelism constructs are used to specify how a function is executed on the GPU. A function that runs on the GPU is called a *kernel*. The kernel is launched from the host (i.e., the CPU), with a specified number of *threads*,

all of which execute the same kernel code. GPU threads are lightweight, and a single kernel typically invokes hundreds or thousands of threads that are scheduled onto the GPU and executed as computing resources become available. Figure 9 shows the organization of the various processing elements. Threads are grouped in *blocks*, which in turn are grouped in a *grid*. A grid therefore is the entire set of all processing threads that carry out the execution of a kernel. Blocks within a grid have two-dimensional indices (x and y), which are used to determine the section of the work that each block is responsible for. Similarly, every block is made of threads with three-dimensional indices (x, y, and z), for determining which part of the work within the block every thread computes. GPUs support single program multiple data (SPMD) computation models: every block can execute a different path through the kernel code (paths are determined by conditional branches) independently of all the other blocks. Therefore, even though all the blocks execute the same kernel, different blocks may execute different sets of instructions within the kernel. Threads in each block are further grouped into *warps* of 32 threads, where a warp is the atomic vector unit of execution. All threads within a warp execute in the single instruction multiple data (SIMD) computation model, which means that all the threads execute the same set of instructions of a kernel; however, different warps with the same block are free to execute different paths within the kernel. In the event that threads within a warp need to execute different paths of the kernel based on their data values (this event is called *thread divergence*), all the threads in the warp have to execute all the paths taken by any of the threads that constitute that warp, but only commit the results of the path that is relevant to them. Thread divergence can be costly, first because the different paths are serialized (example shown in Figure 10), and secondly because it results in threads performing unnecessary computation, thus occupying computing resources only to discard the results in the end.

Though all threads within a warp share the same state (e.g., program counter, execution schedule, etc.), each thread maintains its own set of private registers for computing its data (shown in Figure 9). Registers are the fastest type of memory available to threads. In addition to registers, all threads within a block have access to a shared memory space that can be used to read and write common data. This space is managed explicitly in software (by declaring a variable or array with the __shared__ keyword appended to its data type) and is commonly used to store shared data among threads locally to avoid replicating accesses to the main memory. At the highest level, and with the highest access latency, is global memory, which is viewable by all

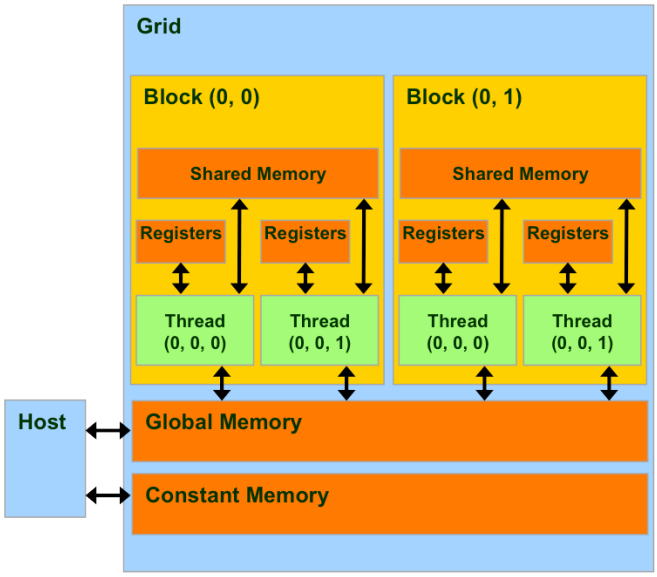


Figure 9. CUDA programming model

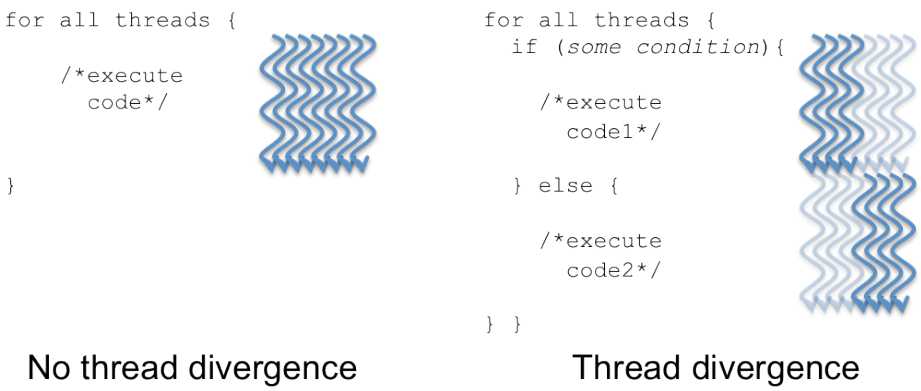


Figure 10. Effects of thread divergence on warp execution

the threads across blocks, as well as the host processor. Constant memory is a subspace within global memory that is read-only, and is cached closer to the SMs for faster re-access to the data by the threads.

All threads within a block can be synchronized using the __syncthreads() function. Synchronization enables the use of the shared memory by guaranteeing that all threads have finished writing data into it before it is subsequently read, and conversely, data in shared memory is read by all threads that need it before it is overwritten by others. In the general case, however, threads across blocks cannot be synchronized except by ending the kernel execution.

The host processor (typically the CPU) controls the computation on the GPU (also referred to as the device). The host launches the kernels to be executed on the GPU with the corresponding grid and block configurations. Kernel launches are asynchronous, meaning that once the host launches a kernel, it can continue executing its own workload without waiting for the GPU kernel to complete execution. The kernel is synchronized once the data it computes on the device is requested back on the host. Alternatively, the kernel can be made synchronous using API calls provided by the language. In addition, because the GPU and CPU have different memory address spaces, the CUDA language also provides APIs for dynamically allocating and freeing memory on the device, as well as transferring data to and from the device (using DMA transfers). These calls are usually synchronous, but their asynchronous equivalents are also available.

2.2 GPU Architecture

As one would expect, there is a duality between the GPU's hardware organization and the programming model. Figure 11 shows a simplified diagram of the GTX 280 architecture. The

GTX 280 features 240 cores (called streaming processors or SPs). Each processor is a single-instruction in-order processor with one floating point and integer arithmetic unit. Every eight SPs are grouped into a simultaneous multiprocessor (SM), for a total of 30 SMs. All the SPs in an SM share a single instruction fetch and decode unit, effectively making the SM an eight-wide vector processor, with each SP processing one of the eight elements. Blocks are assigned to single SMs for execution, and every SM can maintain contexts and execute up to eight blocks simultaneously. Every warp within a block that is scheduled on an SM executes instructions for its 32 threads in four consecutive cycles. Scheduling multiple blocks (and by association warps) on every SM allows the GPU to hide the long latency of global memory accesses such that when one warp makes an access to memory and has to wait for the request to return, another warp can be executed in the mean time.

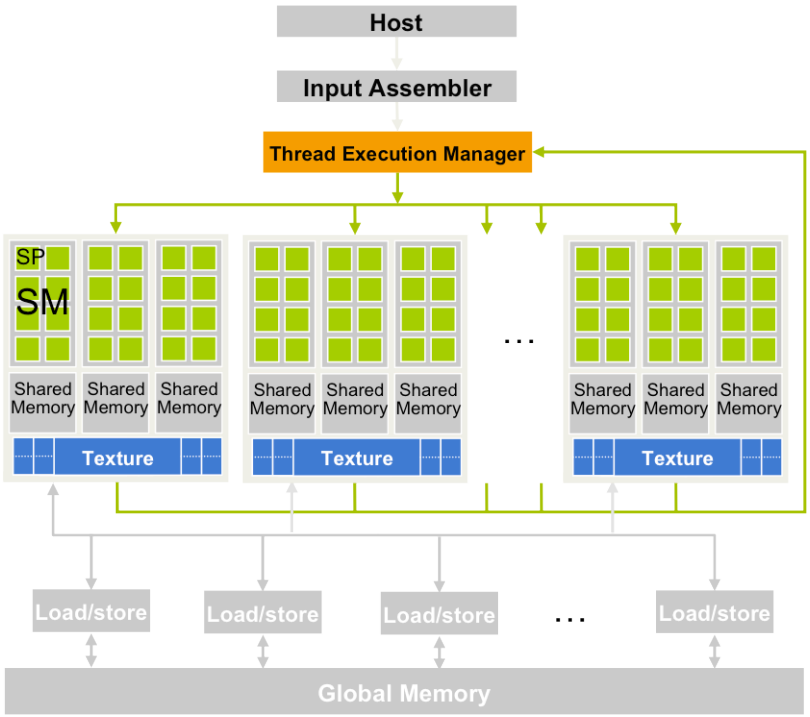


Figure 11. GTX 280 architecture

Every SM is attached to its own shared memory, which has a separate address space from the other shared memories and global memory. Shared memory is a scratchpad memory,

meaning that it is explicitly managed by software and is not guaranteed to be consistent with the contents of global memory. Though multiple blocks may run on an SM simultaneously, each block can only access its own equal portion of shared memory. The amount of shared memory needed by each block can also determine how many blocks can be scheduled simultaneously on an SM.

Global memory is a high-latency off-chip DRAM memory attached to the GPU and is accessible by all the SMs. The DRAM technology makes read and write accesses into memory very slow, so one way to improve the efficiency of such memory is to increase the amount of data returned by each access, thus amortizing the latency [2]. This collection of data returned by a single access is called a burst. In order to utilize the data returned in a given burst, GPUs combine accesses of threads within a half-warp if those accesses are made to the same burst. When all threads in a half-warp access data in the same burst, we call that a *coalesced* access (Figure 12.a). If the requests are not coalesced (example in Figure 12.b), every thread will issue a separate request and receive a full burst, of which it will only extract the data that it needs. Performance can degrade significantly as a result of non-coalescing.

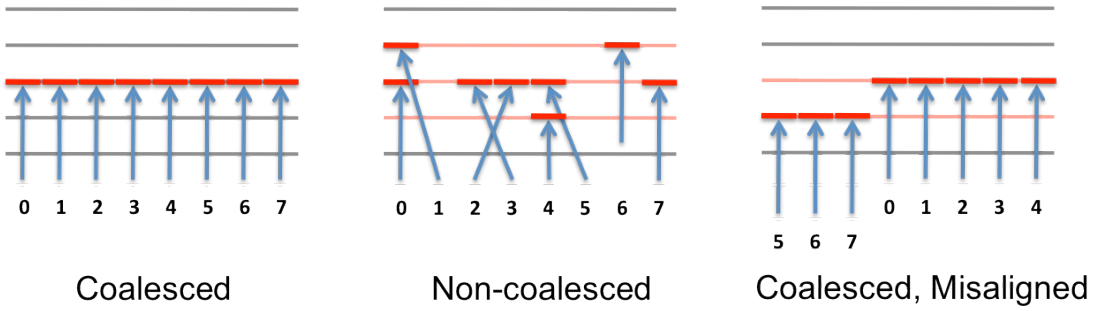


Figure 12. Effects of coalescing and alignment on global memory accesses

The GTX 280 supports three burst sizes: 32 bytes, 64 bytes, and 128 bytes, corresponding to 2-byte, 4-byte, and 8-byte data types respectively. Alignment is another factor that can affect memory performance. Alignment occurs when the starting address of a memory request by a

half-warp coincides with the start of a burst. If the request does not start at the beginning of the burst, the misalignment may cause the request to be split into multiples, thus wasting memory bandwidth, increasing memory latency, and resulting in performance degradation (Figure 13).

Because each bin in compact binning is allowed to have an arbitrary size, misalignment becomes a concern when accessing these bins. This motivated us to try and better understand the effects of misalignment on the kernel's performance. To that end, we wrote a micro benchmark that simulates the accesses into regular bins that were initially aligned, and recorded the runtime as we varied the amount of misalignment. Figure 13, shows the results of that simulation. As we can see, misalignment increases the runtime by nearly 60% when threads load consecutive floats from the bin array. The runtime improves slightly when the misalignment is 32 bytes since that coincides with a 32-byte burst boundary. The effects of misalignment can be reduced if threads load a float2 short vector type element from the array rather than a single float. Float4 vector types also improve runtime compared to single float types but only in certain cases, and they fail to outperform float2 accesses.

We used profiling counters that recorded the number of accesses made to each of the three burst sizes to further explain the change due to misalignment, seen in Figure 13. Based on the counters' values, we have deduced the model shown in Figure 14. The model shown corresponds to a half-warp loading 256 consecutive bytes. This corresponds to 4 separate load instructions for float, two for float2, and one for float4 data types. Note that even though it only takes one instruction to load 256 bytes of float4 data, it takes two memory accesses of the largest burst size to satisfy the request. However, those two memory accesses are treated as a single unit and cannot be scheduled separately.

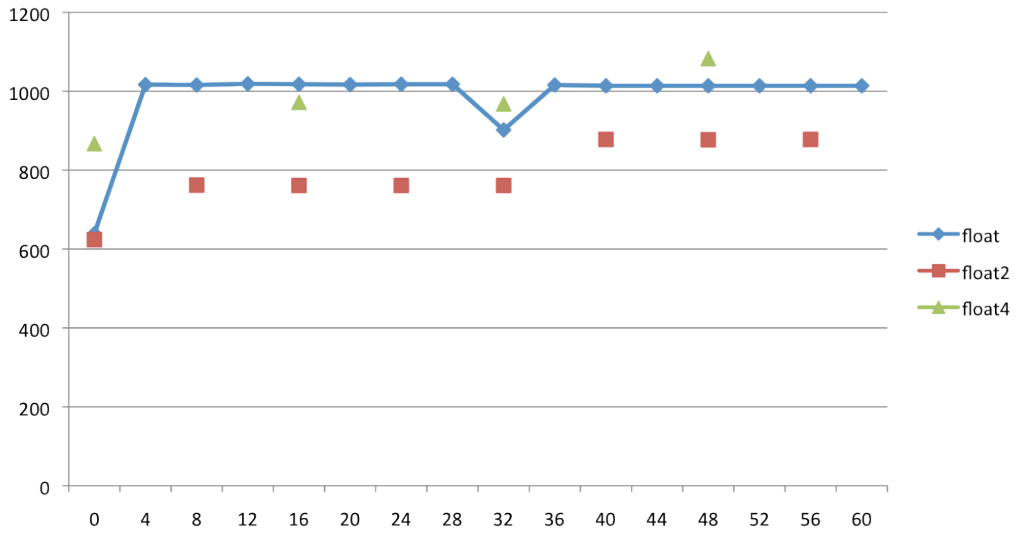
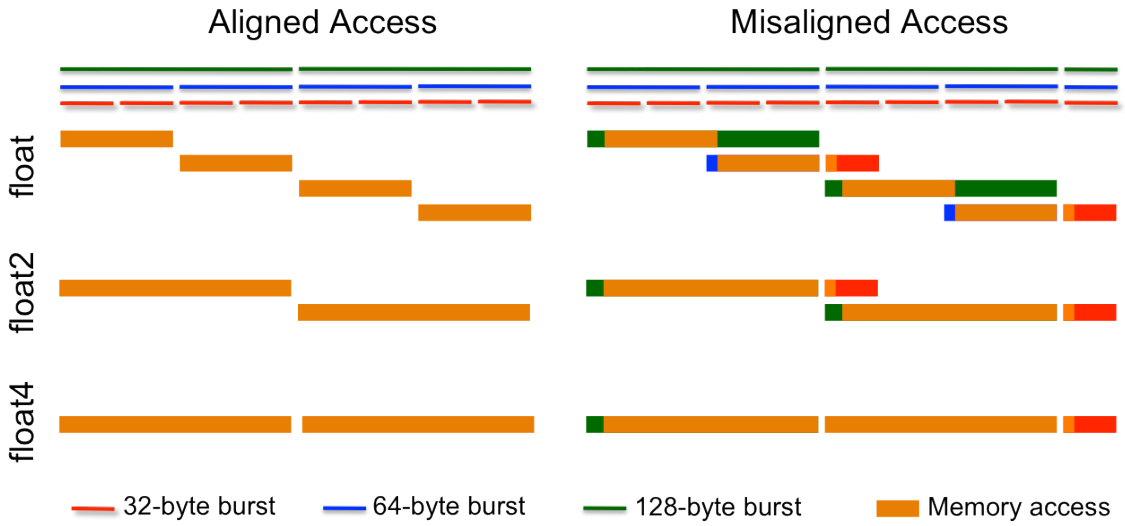


Figure 13. Effects of misalignment on float and float vector types



Total Number of Accesses

	Aligned	Misaligned
float	4	6
float2	2	4
float4	2	3

Figure 14. Memory accesses due to misalignment

Misalignment had the greatest effect on single float types as it results in six memory accesses of various burst sizes to load all 256 bytes. Furthermore, a misaligned access to an array

of floats causes the largest waste of burst data (everything that is not orange in the misaligned float diagram), which inevitably reduces the effective memory bandwidth. Misaligned float2 accesses waste significantly less bandwidth, despite a slight increase in the number of accesses, and that could explain the behavior in Figure 13. Misaligned float4 accesses are the most efficient both in the number of added accesses and in the amount of wasted bandwidth; however, they perform worse than float2. We believe this to be the result of scheduling since the three accesses in the misaligned case have to be scheduled simultaneously, likely resulting in memory bank conflicts (for a more thorough study of bank conflicts, please refer to [2]).

CHAPTER 3

DESCRIPTION OF BENCHMARK APPLICATIONS

In this chapter we will introduce the four benchmarks that we use to compare regular and compact binning. We will use these benchmarks in Chapters 4 and 5 to provide quantitative analysis for the different aspects of the comparison. The four benchmarks are: MRI reconstruction gridding step, cutoff Coulombic potential, Blinn's blob, and N-body simulation.

3.1 MRI Reconstruction

Magnetic resonance imaging (MRI) is a common, non-invasive technique used in radiology to analyze the internal structure of the human body, and is used for a wide range of applications where precise information is desired due to its image resolution compared to other imaging techniques like computed tomography (CT) and x-ray. The scanner used for MRI data acquisition collects samples in the frequency domain. An inverse fast Fourier transform (IFFT) is then applied to the acquired data to transform it back to the image domain.

Because of the need to perform an FFT operation during reconstruction, traditional acquisitions collected data along a Cartesian path with uniform spacing between data points. The result, however, was a very slow acquisition that presented physical challenges to the patient, who had to lie in the scanner for approximately 20 minutes without moving. More recently, MRI acquisition has been performed on non-Cartesian paths, which saves both time and data [3]. Research has shown that fewer samples can be collected while still maintaining enough data to reconstruct the image without quality degradation. The time and data saved at acquisition time, however, come at the expense of added complexity and time needed to reconstruct the images. There have been many efforts to speed up the reconstruction of non-Cartesian data. One such

effort involves treating the input data as a linear system and solving it using an iterative method such as least-squares or conjugate gradient, as shown on the right-hand side of Figure 15. Wu et al. implemented a GPU version of this approach [4]. Another approach is the gridding technique shown on the left-hand side of Figure 15. As the name suggests, the idea behind gridding is to map the non-Cartesian input data onto a Cartesian grid in the same domain (i.e., the frequency domain), then proceed with IFFT as is done in the classical method. One motivation for using gridding instead of the iterative method is that the former has O(N log N) complexity compared to the $O(N^2)$ complexity of the iterative method. The trade-off comes in the slightly poorer quality of the gridding image due to some noise being introduced by the gridding step itself.

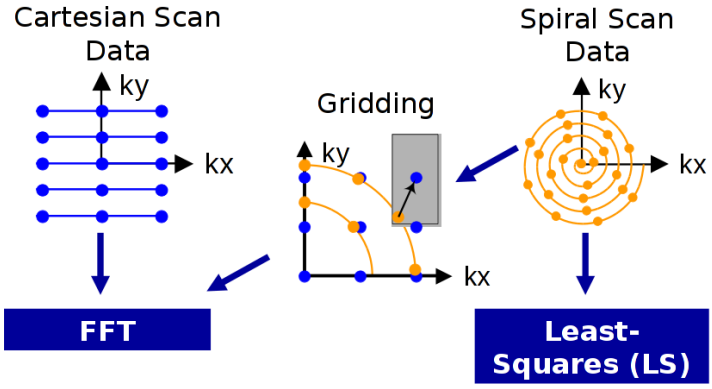


Figure 15. Reconstruction techniques for non-Cartesian MR sampling

At the heart of the gridding step is an application of limited-range function. Every input point, also known as a sample point in the 3D frequency domain space, is mapped onto a 3D Cartesian grid of the same space, using a Kaiser-Bessel function [5]. The Kaiser-Bessel function is used to determine the weight of the contribution of a sample point onto a grid point, based on the distance between the two. Because the weight of the contribution becomes insignificant beyond a certain distance between the two points, a hard cutoff is imposed on the kernel beyond which the contribution is considered to be zero. The cutoff distance for the Kaiser-Bessel function is called the "kernel length."

Figure 16 shows a sample acquisition trajectory. The acquisition starts in the center of the space and moves outward in a conical shape with varying angles of the cone. One can see from the figure that the data density is higher in the center than it is on the outside. Figure 17 is a plot of the data density along the space. This better shows the large variation in data distribution throughout the space. For a large data set like the one shown in Table 1, approximately 24 million sample points lie in the horizontal band shown at the bottom of the curve. The average density of points in that region is approximately 4 sample points per 1 unit^3 bin. The peak density

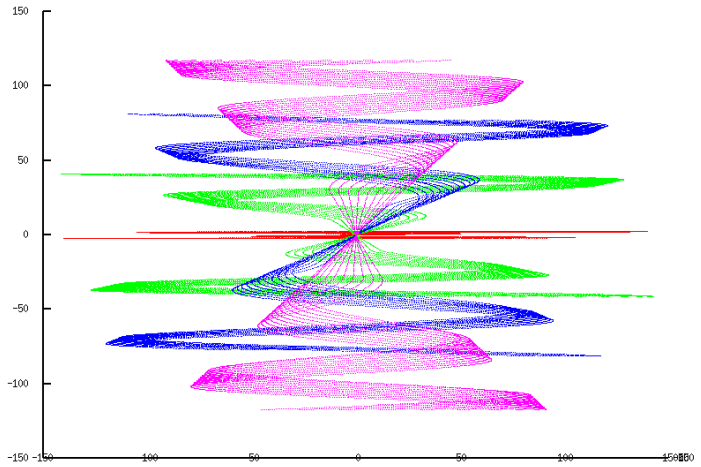


Figure 16. Acquisition trajectory of non-Cartesian MR sampling

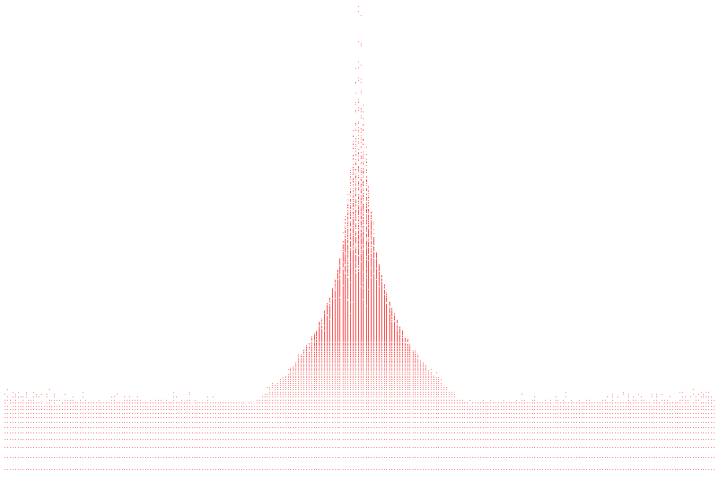


Figure 17. Sample density for the trajectory shown in Figure 16

in the middle is 391k points in a single bin, and decreases sharply moving away from the center. It is very inefficient, and sometimes infeasible, to apply regular binning to this kind of data distribution. Because the mean of the data density is too high and the variance too large, there is no bin size that would map the majority of the input points to the GPU and minimize the amount of spill-over data to the CPU without causing very large data bloats. In this situation, compact binning is more than just an optimization technique; it is an enabling one. We will use the small data set for comparison in Chapters 4 and 5 since it can be represented using regular binning if the bin depth is capped at 9 samples per bin. Figure 18 shows the output of the small data set.

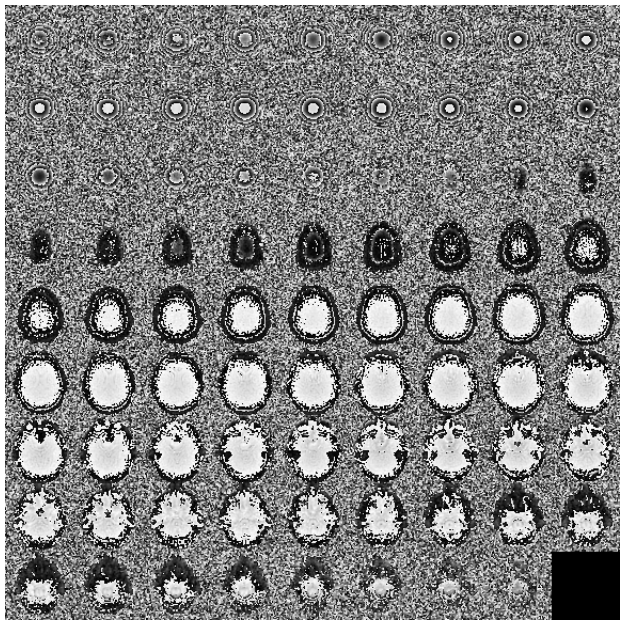


Figure 18. Sample reconstructed MR image

Table 1. MRI data statistics

	Small	Large
No. of Samples	2655910	30144488
No. of Bins	16777216	191102976
Min Bin Depth	0	0
Max Bin Depth	11560	391536
Avg Bin Depth	0.158305	0.07316
StdDev	2.86096	28.861158

3.2 Cutoff Coulombic Potential

A biomolecular modeling system seeks to simulate the interactions between atoms in a medium. There are two types of interactions in such a system: the interactions among chains of covalently bonded atoms (such as proteins) and the interactions between non-bonded atoms. These interactions obey Newton's second law of motion with the forces in the system generated by Coulomb's law of electrostatic interaction. Computing these simulations is computationally expensive. It is on the order of O(N) for the covalently bonded atoms, and $O(N^2)$ for all the pairs of unbonded atoms. Furthermore, because a truly continuous simulation is impossible to achieve, we approximate the motion of atoms in the space by breaking down the simulation's time window into many consecutive discrete time steps. For each time step we compute the forces exerted on all the atoms in the space, and based on those forces update the velocity and position of each atom for the next time step. Depending on the duration of time being simulated and the length of each time step, a full simulation's runtime can be on the orders of hours, weeks, or even years.

Another aspect of the biomolecular system that is useful for simulation and visual rendering (example in Figure 19) is to determine the electrostatic potential map for that system. The electrostatic potential map is a grid of equally spaced points, and the potential of each point on the grid is calculated by accumulating the potential of each atom at that point. Atoms are modeled as point charges with each atom i at position r_i holding a fixed charge q_i. The potential of a map point at position r is computed using the following equation:

$$V(\vec{r}; \vec{r}_1, \vec{r}_2, \ldots, \vec{r}_N) = \sum_{i=1}^{N} \frac{q_i}{4\pi\varepsilon_0 |\vec{r} - \vec{r}_i|} s(|\vec{r} - \vec{r}_i|),$$

where ε_0 is the dielectric constant of the medium, and $s(r)$ is a unitless scaling factor between 0 and 1. When $s(r)$ equals 1, the electrostatic potential for every output element is computed by

iterating over all the atoms in the space, resulting in an $O(N^2)$ algorithm. Hence, to improve the algorithm's complexity, $s(r)$ is chosen in such a way to yield a cutoff distance r_c beyond which the contribution's value is insignificant and can be approximated to zero. One choice for $s(r)$ is

$$s(r) = \begin{cases} (1 - r^2/r_c^2)^2, & \text{if } r < r_c, \\ 0, & \text{otherwise,} \end{cases}$$

With this equation for $s(r)$, the potential of an atom onto a map point diminishes gradually to 0 as it approaches r_c, the cutoff radius, and is zero beyond r_c.

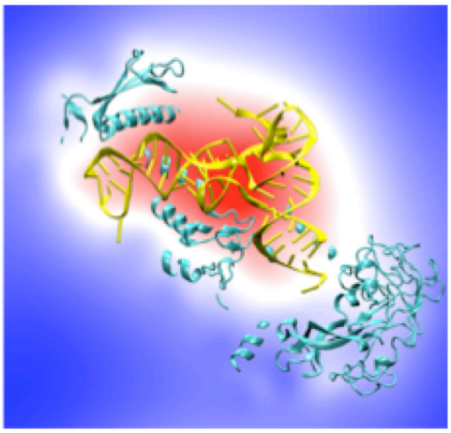


Figure 19. Example of a rendering of protein cells and their potential map

When $s(r)$ is less than 1, the computation pattern is effectively a limited-range function, and can benefit from binning to maintain a computational complexity of O(N) when executed in fine-grained parallelism. Because molecules have a fairly uniform density of about 1 atom per 10 $Å^3$, regular binning works well for this computation. Rodrigues et al. [6] implemented a highly optimized version of the electrostatic potential map computation for GPUs using regular binning. By using regular binning they were able to control the alignment and coalescing of memory accesses when reading bins into shared memory. In this work we will compare their regular binning implementation with compact binning and demonstrate that even for well distributed input data, compact binning can be a viable solution. Table 2 shows the statistics of

two input data sets for CP. For our experiments, we will use the large dataset. Compared to the MRI dataset, the CP large data set has a much more uniform distribution (smaller standard deviation).

Table 2. Coulombic potential data statistics

	Small	Large
No. of Atoms	5943	570348
No. of Bins	4913	140608
Min Bin Depth	0	0
Max Bin Depth	12	14
Avg Bin Depth	1.209648	4.056298
StdDev	2.528611	3.342421

3.3 Blinn's Blob

The Blinn's blob algorithm is very similar to the electrostatic potential map computation. It too can be used for the image rendering of a point cloud [7], with an example shown in Figure 20. Blinn's blob creates a density map by accumulating the density contributions of all atoms to a particular point on the grid. The contributions depend on the distance of the atom from the grid point, the radius of the atom as well as the blobbiness that is desired. In addition, because the density function exponentially decreases with the increase in distance between the atom/grid point pair, every atom only affects a small neighborhood of grid points, beyond which its contributions are negligible and can be approximated to zero. Table 3 shows three example data sets for Blinn's blob. All three of these data sets exhibit a very sparse distribution of the input elements in the space (average bin depth < 0.2), which means that in the case of regular binning, the majority of the bins will only contain padding elements. As a result we expect to see a noticeable improvement in performance and memory usage with the use of compact binning.

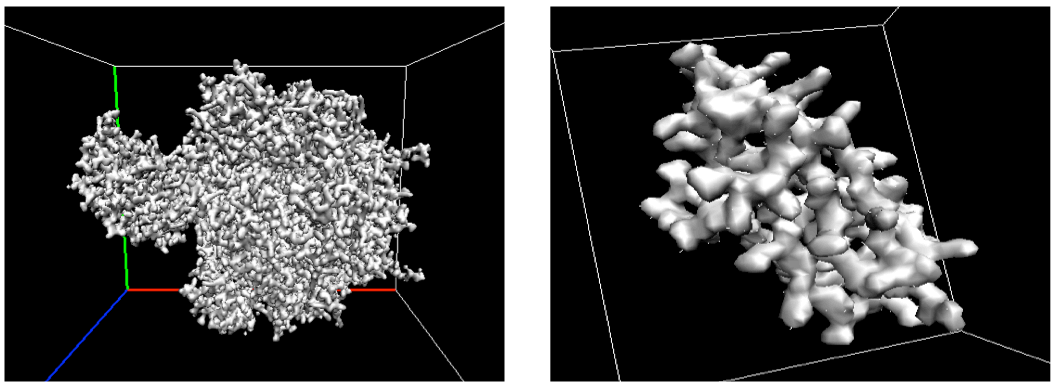


Figure 20. Examples of Blinn's blob rendering for atom clouds

Table 3. Blinn's blob data statistics

	Small	Large	Random
No. of Atoms	1739	26318	500000
No. of Bins	262144	1179648	23887872
Min Bin Depth	0	0	0
Max Bin Depth	360	3	3
Avg Bin Depth	0.006634	0.02231	0.020931
StdDev	0.708008	0.15885	0.144142

3.4 N-Body Simulation

N-body simulations are common tools used to model astrophysical systems and their evolution. Due to the very large number of elements and the long period of time for which these system are simulated, good performance is critical for the feasibility of these simulations. The N-body simulation, among other things, seeks to simulate the motions of objects, such as stars, galaxies, and planets, through space based on the gravitational forces they exert on each other. The objects are typically modeled as points in space with mass, position and velocity attributes, and do not represent physical objects. Similar to the Coulombic potential, the motion over time is simulated by computing the positions and velocities of all the objects for a given discrete time step. The most direct and most accurate approach is the particle-particle simulation, which computes the gravitational forces of every pair of objects, but its complexity grows quadratically

with the number of elements, and thus is computationally infeasible for large simulations. Another method is the particle-mesh method (PM) which partitions the space into meshes for which a fast Fourier transform is computed to solve Poisson's equations, therefore reducing the computational complexity to O(M log M), where M is the number of meshes. However meshes have to be regular to satisfy the constraints of the FFT algorithm, and mapping the particles onto the mesh introduces noise into the system, therefore sacrificing some accuracy in the final results.

A combination of PP and PM combines the benefits of both approaches [8]. If we partition the total force on a particle as the sum of nearby forces and distant forces, we can use the PP method to compute the nearby forces, where accuracy matters, and PM for the distant forces where approximated results can be tolerated. In this setting, computing the PP forces becomes an application of limited-range functions. N-body simulation is different from the other three applications in that its output is not a regular grid. Since the quantities being computed are the forces of the bodies among themselves, the input and output in fact consist of the same data set. Input binning is still useful to reduce the number of elements each output object needs to access; however, there is little to no locality to the output when every thread block is given an equal number of output elements, because the output is non-uniformly distributed in space. For that reason, data sharing in the shared memory is not applicable, and may in fact hurt performance. Instead, every thread computing the position and velocity of a body reads its relevant bins directly from global memory. This access pattern differs vastly from the other three, which makes it less relevant for parts of the discussion in Chapters 4 and 5. However, since N-body is the most general type of limited-range application, it is important to analyze how different binning techniques affect its performance. The statistics of the input data for N-body, as shown in Table 4, are vastly different from the other three in part because it is the only

application with as many inputs as outputs, and more inputs than number of bins. Figure 21 is a visual representation of the simulation of the data set shown in Table 4 for a given time step.

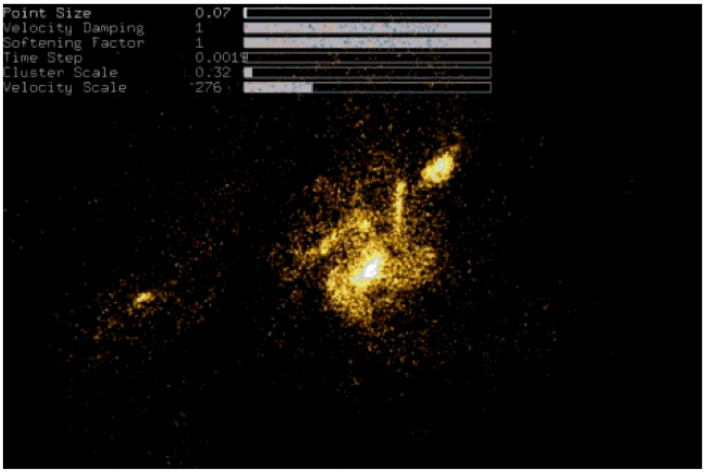


Figure 21. Example of an N-body simulation from the CUDA SDK

Table 4. N-body data statistics

	Random
No. of Bodies	131072
No. of Bins	32768
Min Bin Depth	0
Max Bin Depth	99
Avg Bin Depth	4
StdDev	6.52

CHAPTER 4

COMPARING REGULAR AND COMPACT BINNING

In this chapter we will compare regular binning and compact binning. We have discussed in Chapter 1 how regular binning provides ease of access to the bins, and better control over coalescing and alignment of memory accesses, at the cost of large memory requirements when the bin densities vary. We have also explained how compact binning eliminates the overhead of memory padding at the cost of creating and having to use an additional array for accessing the bins. Furthermore, with compact binning, it is more difficult to maintain alignment when accessing the data in the bins. In this chapter, we evaluate qualitatively and quantitatively both binning approaches. We begin by explaining the algorithm for performing and using each binning technique, then proceed to analyzing the differences.

4.1 Regular Binning Algorithm

Step1: Determining the size of the largest bin

Determining the size of the largest bin can be done either sequentially or in parallel. Either way, a zero-initialized integer array for all the bins needs to be maintained, and as each input element is visited and its bin index determined, the integer corresponding to that bin is incremented by 1. When performed in parallel, generating the integer array (which is effectively a histogram) is most simply done using atomic updates into the array. Once the histogram is generated, we use it to determine the size of the largest bin, which can be done by using a reduction computation with a max operator, and the final access to determine the max can be done using a reduction computation with a max operator [9]. This step can be omitted if the bin size is known statically (e.g., applications where the bin size does not change for different input data). Coulombic

potential is an example of such an application because the density of atoms in space is fairly regular across data sets.

Step2: Binning the input elements

Once the maximum bin depth has been determined and the data structure allocated accordingly, we can perform the actual distribution of input elements into the bins. This step can also be performed sequentially or in parallel since it is not very computationally expensive. In order to perform binning, we need to maintain another zero-initialized integer array of offsets into each bin, which is used to determine the offset within the bin at which to place a given input element. For each input element, we determine once again the bin it belongs to, place it at the current offset within the bin, then increment the offset. If performed in parallel, binning can be achieved by atomically incrementing the offset counter, and the effects of this atomicity are not too severe, since the only contention is between elements trying to update the same bin, and all other bins can be populated in parallel.

Step 3: Performing the limited-range function computation

In order to perform the computation of the limited-range function, the output grid is first divided into tiles, where each tile is a subset of spatially local output elements. Each tile is assigned to a thread block where every thread computes exclusively the result of one or more output elements from that subset. The spatial locality of the output in a tile is important to maximize sharing of input data among threads within the block. Figure 22 shows the pseudo code for the limited-range computation. `sharedLocalBin` is an array in shared memory that is accessible by all the threads within a block. In the code in Figure 22, each thread is shown to compute only one output element and compute that output's index based on the 2D `blockIdx` and 3D `threadIdx` (both of which are CUDA constructs). Since every thread computes an output

element exclusively, the result can be accumulated in a local register (line 2).

Every output element is computed by a single thread exclusively, and that thread can compute the value of that element locally (line 2). Every block iterates over all the bins that its output tile intersects: zLo to zHi, yLo to yHi, and xLo to xHi are the 3D bounds of the region intersected by a given tile. For each bin that is visited, all of its elements are loaded cooperatively into shared memory by all the threads in the block. Note that a bin is visited if at least one of the outputs within the block's tile intersects that bin; however, that bin may fall outside the cutoff region of other outputs in the tile. That is why it is still necessary to check whether a given input point is within the cutoff distance of the output point before computing its contribution to that output (line 15). Once all the bins and all the elements within them have been visited, and their contributions added, each thread writes its privately computed output to the global array that is the final result.

```
00 __shared__  inElem sharedBinCache[/*max size*/];
01 outputIdx index = computeOutputIndex(blockIdx, threadIdx);
02 outElem myOutElem = initOutElem(index);
03 int zLo = z0 - cutoff;
04 int zHi = z0 + blockDim.z + cutoff;
05 // compute yLo, yHi, xLo, xHi similarly
06 for (z: zLo → zHi){
07   for(y: yLo → yHi){
08     for(x: xLo → xHi){
09       int count = binCount[z][y][x];
10       if(threadIdx < count){
11         localBinCache[threadIdx] = globalBinArray[z][y][x][threadIdx];
12       }
13       __syncthreads();
14       for(i: 0 → count){
15         if(|localBinCache[i].coords - myOutElem.coords| < cutoff){
16           /*compute the contribution of this input onto the output*/
17         }
18       }
19     }
20   }
21 }
22 globalOutputGrid[index] = myOutElem;
```

Figure 22. Pseudo code for parallel limited range function kernel with regular bins

BinCount in line 9 is the histogram generated in step 2 when performing the input binning and is used to determine how many elements are in a given bin to avoid unnecessarily loading the padding elements. Alternatively, we can load the entire content of the bin regardless of the number of real elements within the bin, and as we traverse shared array of elements, break out of the loop upon the first occurrence of a padding element (line 14). Effectively, the padding elements behave as sentinels in this situation. There are advantages to both approaches. If the number of actual elements in each bin is not much smaller than the maximum bin capacity, loading the padded elements into shared memory will likely be less costly than reading the binCount (which requires an extra global memory access). Alternatively, if the number of elements per bin varies significantly, it may be more effective to only load the elements needed, by first figuring out how many real elements there are in each bin. Figure 23 compares the two alternatives for all four benchmarks. For each benchmark the runtime of the limited-function execution is plotted for the sentinel checking method and the count method. For MRI, Blinn, and CP, count always performs better than sentinel checking. This is likely an indication of a large number of empty bins or bins with fewer than bin depth elements, which causes the sentinel method to read more data than the count method, resulting in worse performance. N-body (Figure 23.d), on the other hand, performs slightly better with sentinel checking up to a bin depth of 10, after which count starts performing better. Overall, despite the additional access to global memory to retrieve the size of each bin, checking the element count seems to perform better than loading the entire bin into on-chip memory and checking for the sentinel locally.

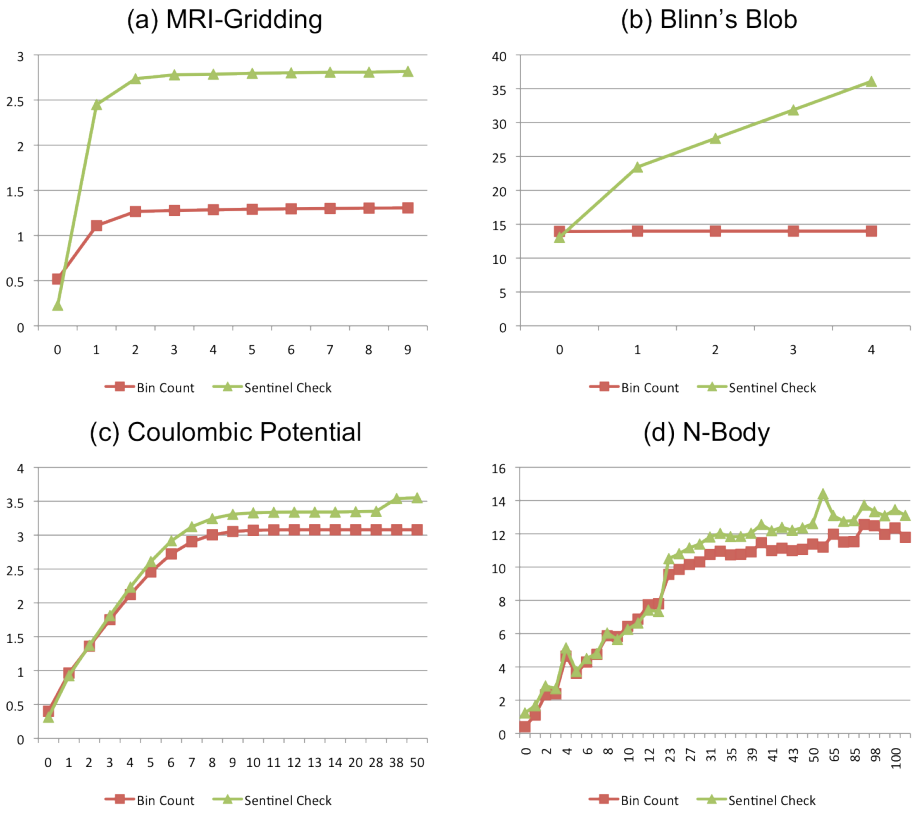


Figure 23. Comparing the runtime of count vs. sentinel checking for varying bin depths

4.2 Compact Binning

Step 1: Determining the size of each input bin

This step is identical to step 1 of the regular binning algorithm. The purpose of this step in compact binning, however, is slightly different: The histogram built in this step will be used to determine the start of each bin rather than the max depth of the bins.

Step 2: Determining the start of every bin

Using the histogram generated in step 1, we can determine the start of every bin. The operation that achieves this is called a prefix sum. The prefix sum computes, for every element at index *i* in an array, the sum of all the elements from index 0 to index *i*-1 (the value at index 0 is zero). Since every element in the array corresponds to the size of a bin, computing the starting offset of

a bin corresponds to the sum of the sizes of all the bins that precede it. Prefix sum (also known as a scan operation), can be efficiently performed in parallel [10].

Step 3: Binning the input elements

This step is also similar to step 2 of the regular binning algorithm. The only difference is that the starting offset of each bin has to be looked up from the array generated in the previous step, since it cannot simply be computed, as is the case with regular binning. Just like in regular binning, another array needs to be maintained that keeps a count of the number of elements that have gone into a bin, to determine the position of every input element that was placed in the bin. Despite the fact that bins have varying sizes, each bin can be populated in parallel with other bins since the start of every bin can be independently known by reading the starting offset from the array of bin offsets.

Step 4: Performing the limited-range function computation

In its simplest form, the computation of the limited-range function using compact bins does not look much different from its regular equivalent. The only difference is the need to access the bin offset array to determine the start and end of a bin, rather than computing its starting offset using x, y, and z (lines 9, 10 and 12 in Figure 24). Furthermore, when loading the elements into shared memory, the boundary test depends on the variable size of the bin rather than a predetermined constant bin size (lines 11 and 15 in Figure 24).

```
00 __shared__  inElem sharedBinCache[/*max size*/];
01 outputIdx index = computeOutputIndex(blockIdx, threadIdx);
02 outElem myOutElem = initOutElem(index);
03 int zLo = z0 - cutoff;
04 int zHi = z0 + blockDim.z + cutoff;
05 // compute yLo, yHi, xLo, xHi similarly
06 for (z: zLo → zHi){
07   for(y: yLo → yHi){
08     for(x: 0 → xLo → xHi){
09       int start = binOffsetArray[z][y][x];
10       int end   = binOffsetArray[z][y][x+1];
11       if(threadIdx < end-start){
12         localBinCache[threadIdx] = globalBinArray[start+threadIdx];
13       }
14       __syncthreads();
15       for(i: 0 → end-start){
16         if(|localBinCache[i].coords - myOutElem.coords| < cutoff){
17           /*compute the contribution of this input onto the output*/
18         }
19       }
20     }
21   }
22 }
23 globalOutputGrid[index] = myOutElem;
```

Figure 24. Pseudo code for parallel limited range function kernel with the compact bins

4.3 Comparing Regular and Compact Binning

As mentioned previously, using either regular or compact bins involves a tradeoff. Regular bins enable us to compute the starting offset of each bin rather than having to pre-compute it and store it in an array for look-up during the computation. Furthermore, since the bins have the same size, we can better control the layout of these bins in memory, thus ensuring aligned accesses. However, the use of padding may sometimes result in a largely inflated bin data structure, which can limit the size of the problem that can be computed by a single kernel. In addition, because of padding, further checks need to be made to avoid computing unnecessarily for those padding elements. One method discussed in Section 4.1 is to maintain an array of the element count per bin as shown in Figure 22. An alternative method is to load the entire bin into shared memory and check for a sentinel value signifying the end of the valid data in a bin. Either

method incurs a certain amount of overhead.

Compact binning, on the other hand, eliminates the need for padding, therefore guaranteeing that all data loaded into shared memory is valid data that will be consumed by at least one thread. Yet due to the variable size of the bins, we incur the overhead of needing to look up the start and end indices of each bin, which requires additional accesses to global memory. Another side effect of the variable size of the bins is the difficulty of controlling the alignment of bins in memory. In the following subsections, we will evaluate the effects on performance due to the binning overhead, the cost of the various element count methods and the effects of misalignment.

4.3.1 Binning overhead

Computing the size of every bin as the first step of both the compact and the regular binning incurs the same computation overhead. Both are O(N) computations, and both have the same access pattern into the bin counters, based on the values of the inputs. Both also suffer to the same extent from the serializing effects of atomic operations into the bin counter. The use of these arrays differs for the two binning approaches. Regular binning uses this array to determine the max bin depth using a reduction operation that takes O(log M) steps and O(M) comparisons when performed in parallel (M is the number of output elements). On the other hand, the prefix sum used in compact binning to determine the start of every bin takes twice as many steps and performs twice as much computation. Note that this computation does not depend on the number of input elements or their values, but only depends on the number of output elements. However, even for the largest output of any dataset we have analyzed (i.e., the large MRI gridding dataset in Table 1), the runtime is 96 ms and 136 ms for reduction and scan respectively, and the

difference of 40 ms constitutes less than 4% of the total runtime of the algorithm. The final binning step also varies slightly for compact and regular binning. The compact implementation requires a look-up of the bin's starting index, which increases the number of memory requests in comparison with the regular binning algorithm that computes the starting index. Overall, it takes longer to perform compact binning than it does to perform regular binning, but as we will demonstrate later on in the work, the difference is not large enough to negate the benefits of using compact bins to execute the limited range function.

4.3.2 Added access overhead

We have shown in Section 4.1 that maintaining the number of real elements per bin improves the execution time of the limited range function as compared to loading the entire bin, despite requiring an additional access to global memory (Figure 23). With that in mind, the access pattern for compact binning is not much different from the count approach for regular bins. The only differences are highlighted in red in Figures 22 and 24. Instead of loading the number of elements in a bin, the compact algorithm loads the starting offset of the bin, and the starting offset of the bin that follows, and from the difference infers the size of a particular bin. This results in an extra global memory access for every visited bin. Figure 25 shows the relative performance of various access patterns as compared to regular binning (blue series). We can see from the figure that the extra global memory access for compact can have a negative effect on performance (green series). Blinn's blob is the only exception, as it sees a slight performance improvement for compact versus regular. The difference, however, is not significant and could be attributed to more efficient memory accesses (in terms of simultaneous accesses to the different memory banks), and that is beyond the control of the programmer.

One optimization that can be performed to compact binning is to access an entire range of contiguous bins simultaneously rather than accessing each bin separately. More specifically, for any given z and y bin coordinates, bins xL through xH, which occupy consecutive memory locations, can all be loaded simultaneously since compact bins do not contain any padding; all the elements between the start of xL and xH are in fact useful to the computation and all need to be loaded into on chip memory. For that reason, rather than simply reading the start of each bin and the one following it to determine the range of a single bin in x, we can read the start and end indices of the entire range in x once, and load all the elements within that range into on chip memory. The benefits of this optimization are three-fold. First, the number of accesses to the bin offset array is reduced from two accesses per bin, to two accesses amortized over the number of bins within the range. Second, the access into the bins is more efficient as we better utilize memory bursts by not breaking bins' bounds. Finally, by accessing entire ranges rather than individual bins, we get rid of the loop for the x dimension (line 8 in Figure 24), thereby reducing the overall number of iterations within the kernel. As a result of this optimization, we see a significant improvement of the performance of compact binning over regular binning as shown by the purple series in Figure 25. Since N-body does not utilize shared memory to share the input data among all the threads within a block, the optimization of range accesses does not apply to it.

The range optimization can also be applied to the regular binning implementation, but its overall effects are detrimental to the kernel's performance. The reason is that by accessing an entire range we have to inevitably load padding elements into shared memory, which unnecessarily consumes memory bandwidth. The red series in Figure 25 shows the performance of the range optimization on regular binning. In the best case, it breaks even with the performance of regular binning for Blinn. In the case of MRI, however, we see a 6.82X

slowdown that can most likely be attributed to the large number of zero elements that end up being unnecessarily loaded into on-chip memory. The range optimization could not be applied to regular binning in the case of N-body because the implementation does not use shared memory to cache the bins.

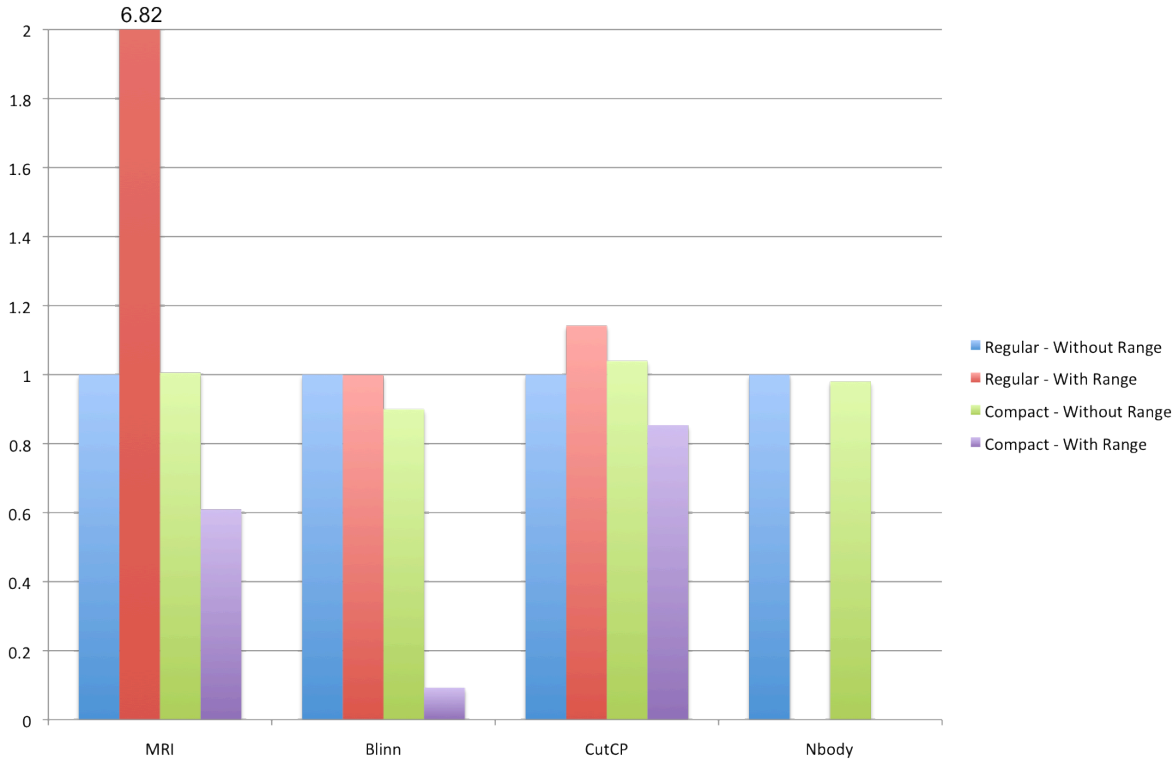


Figure 25. Comparing regular and compact bin accesses

4.3.3 Effects of misalignment

One of the potential drawbacks of compact binning is the resulting misalignment of bins in memory. In this section, we study these effects. First of all, we propose three techniques to fix misalignment. The first one is to pad each input element individually so that it can satisfy the alignment requirements. For all of the benchmarks we studied, padding each element to 8 floats satisfied the lowest requirement of 32-byte alignment with minimal memory bloating (25% overhead for MRI, 50% for the others). Since the padding is done per input element, bins that do not contain any elements in them do not contribute to the padding overhead. Furthermore, since

every element is aligned, by extension, every bin will be aligned as well regardless of the number of elements it contains. The only drawback to this technique is that the padding of each element will reduce the effective bandwidth from global memory when the elements are read.

The second approach is to lay out the input elements in the form of arrays of float vector types (float2 or float4). As demonstrated in Chapter 2, the effect of misalignment on float2 arrays is less severe than on single float arrays. This approach involves a reorganization of the bin data structures from arrays of structures to structure of arrays. Sung et al. discuss the benefits of this transformation in their work [2]; however, unlike the strided access pattern they discuss, if all the elements within the structures are of the same type (in the case for all the benchmarks we analyzed, all the elements are floats), we can have every thread load a single float element from within the structure to shared memory, thus maintaining a coalesced access since the stride of the access is one (see Figure 26). Since the accesses into the array of structures are already coalesced, laying out the data in a structure of array format is not expected to significantly impact the performance. However, if we laid out the data in a structure of short vector arrays, we would expect to see better performance for misaligned accesses as shown in Figure 13 of Chapter 2.

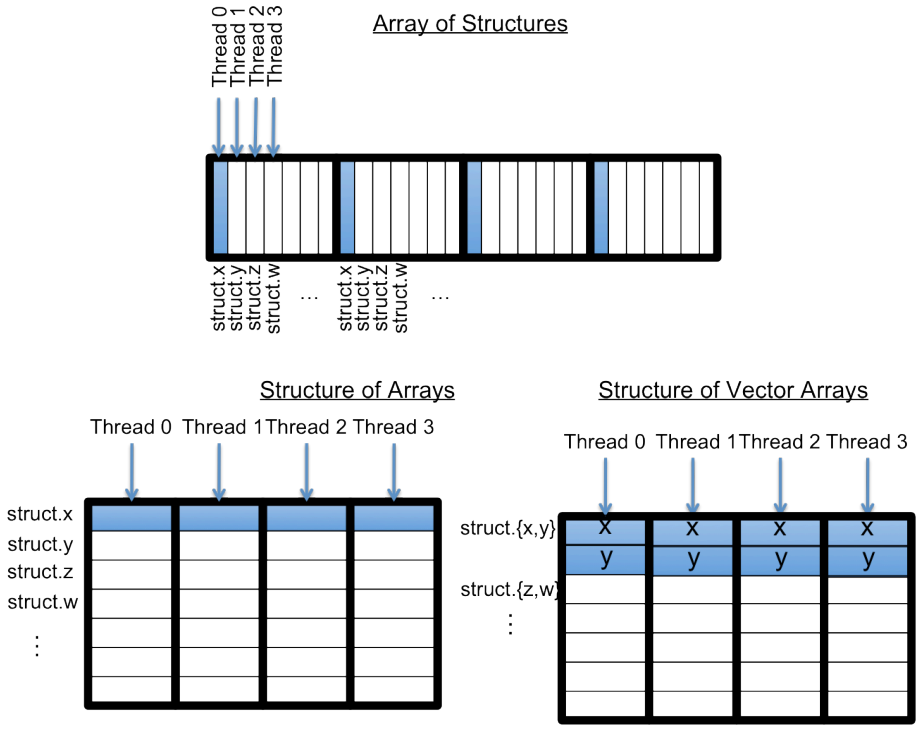


Figure 26. Array of structures, structure of arrays, structure of vector arrays

The third and final approach to fixing misalignment is to pad every bin to the nearest alignment boundary. This is different from the first proposed method of padding each element, since the padding is essentially amortized over the number of existing elements in the bin rather than being incurred for each element. In other words, rather than padding each element separately to an alignment boundary and incurring a padding overhead for each, we pad the entire bin to an alignment boundary to guarantee that the next bin will be aligned; however, the elements within a bin may still be misaligned if their data type is not itself aligned. With this technique, bins that have zero elements in them (which constitute the majority of bins in our studied benchmarks) do not incur any padding. However, the side effect of this approach is the reintroduction of padding into the data structure. As shown in Figure 23, loading this data into shared memory can greatly affect the performance of the kernel. The only way to avoid loading the padding data into shared memory is to load each bin separately rather than loading an entire

range of bins, and this severely impacts the performance as well, as shown in Figure 25.

Finally, we measured the effects of each of the techniques discussed in this section that aim to improve alignment, and the results are shown in Figure 27. The results have been normalized to the runtime of the limited range function executed with regular binning. We can see that in fact any effort to reduce or eliminate misaligned accesses seems to impact the performance negatively, and that the optimal performance is achieved with a simple array of structures layout without any padding. We conclude from this that in the real kernels, the effects of misalignment are not as severe as shown in the micro benchmark in Chapter 2. One of the reasons could be a healthier ratio of computation to memory accesses, which means that memory accesses can be partially or fully overlapped with computation.

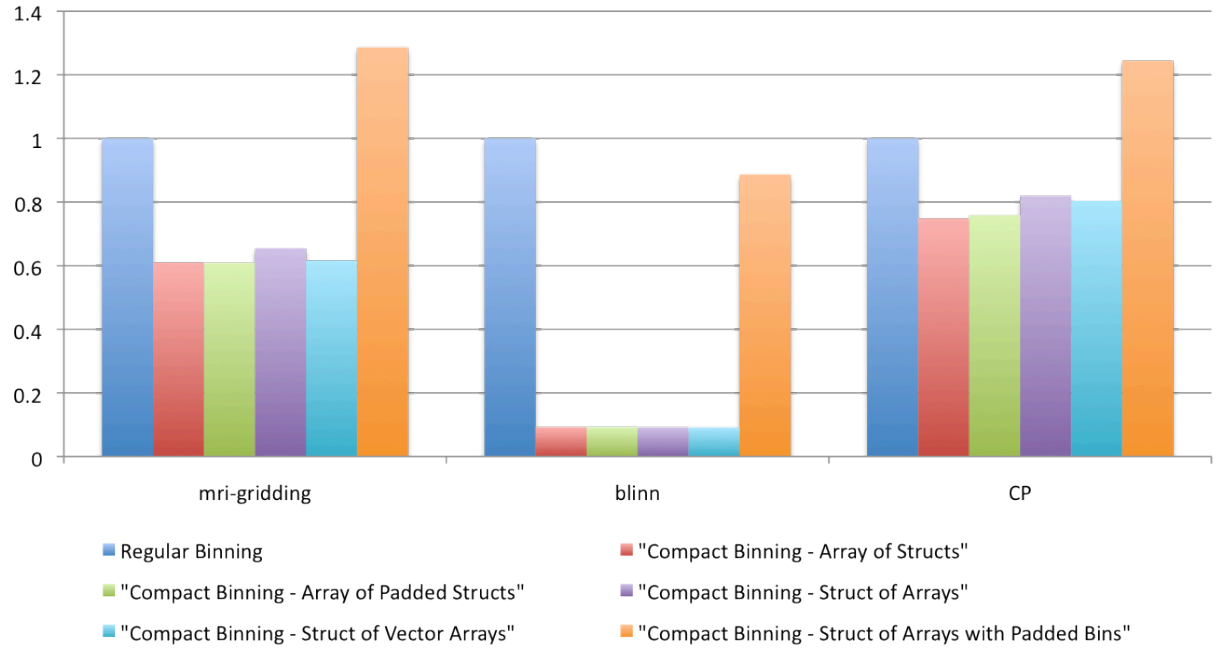


Figure 27. Comparing various techniques for eliminating misalignment of compact bins

CHAPTER 5

PARTITIONING

One of the advantages of using compact bins is to relieve the pressure on memory due to padding when the sizes of the bins vary. While this also improves the performance of the limited-range function as demonstrated in Chapter 4, it does not resolve the issue of load balancing. In essence, due to the varying number of elements per bin, some blocks have to compute more elements than other blocks. Figure 28.a best illustrates this load imbalance. Blocks that reconstruct the center of the space compute significantly more than blocks that reconstruct the edges of the space, due to the high concentration of sample points in the center. To reduce the effect of this load imbalance, we propose in this chapter a simple technique that balances execution by partitioning the work between the CPU and the GPU. Instead of binning all of the input elements for execution on the GPU, we determine a bin depth that achieves the optimal balance between CPU and GPU execution, and offload all of the elements that exceed this bin depth to the CPU when performing binning. Since kernel execution on the GPU is asynchronous to the CPU, the optimal bin depth is defined as that which results in equal execution time on the GPU and CPU. Figure 28.b illustrates our proposed technique for load balancing as applied to the MRI input data. The remainder of this chapter will describe the implementation of this load balancing technique and its effects on regular and compact binning. Currently the only limitation of this work is that the bin depth used for partitioning needs to be provided by the user, and cannot be determined by the program based on input data distribution. This requires the user to know the input data distribution in order to choose a bin depth that yields good performance.

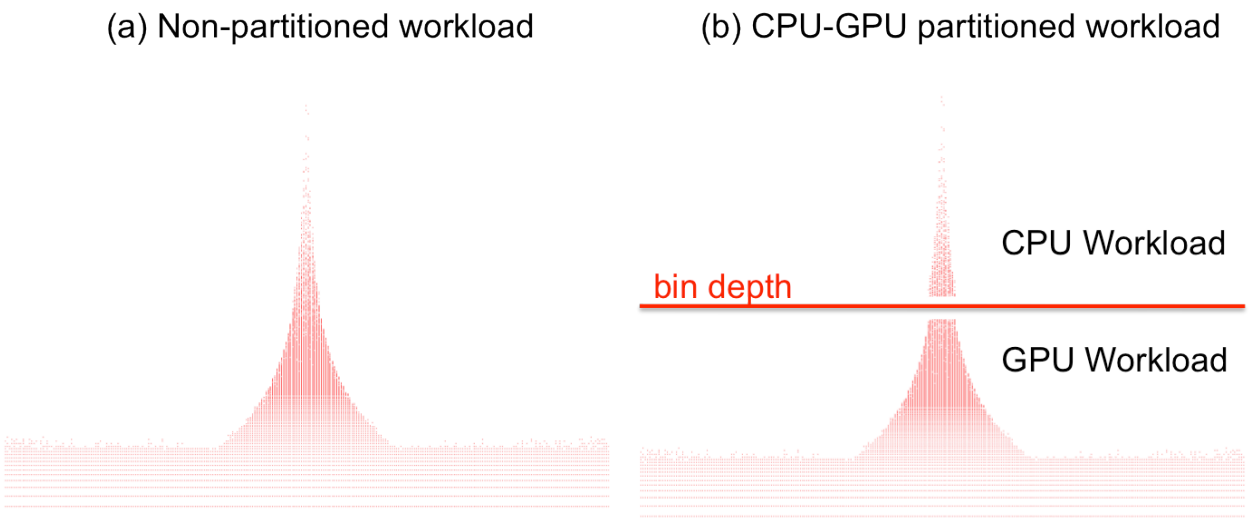


Figure 28. Applying partitioning to the MRI data

5.1 Implementing Partitioning

To implement partitioning we only need to slightly modify the algorithms described in Sections 4.1 and 4.2. In dense binning, the first step becomes unnecessary since the user provides the desired bin depth rather than using the max bin depth determined in this step. We need to add a check in step 2 that verifies that the bin is not already at the bin depth limit, before adding an element to it. If the bin is not full, the thread proceeds to adding the input element, just as it would in the non-partitioning method. If the bin is found to be full, the element is instead placed in the CPU bin. The CPU bin may be a separate array or an extension of the bin data structure, and since this array is the overflow array from all the GPU bins, it cannot be bounded by bin depth. When adding an element to the CPU bin, a counter needs to be maintained for that array, which every thread increments atomically to determine where to insert its overflowed element.

If the binning is performed on the GPU, the CPU bin will need to be transferred back to the CPU, where it will be processed using a scatter approach into the CPU's copy of the output array. Simultaneously, the GPU performs the gather approach for all of its bins, and in the end

the CPU and GPU copies of the output array are combined either on the GPU or the CPU as a straightforward vector addition. This vector addition is in fact an added overhead that is the result of partitioning the work between CPU and GPU, but we will show that despite this overhead, the performance of a partitioned execution is better than that of a unified execution on the GPU or CPU alone.

Enabling partitioning for the compact bin approach requires modifying the first step of the algorithm, which is to determine the size of every bin. Rather than simply accumulating the number of elements that go into each bin, we need to saturate the bin size at bin depth. Therefore, every time we atomically update a bin counter, we need to verify that the number of elements in that bin have not exceeded the bin depth limit, by reading the returned value of the atomic operation. If the returned value is greater than or equal to bin depth, it means that the bin has already overflowed, and therefore we need to atomically subtract one element from it to bring it back to maximum capacity. The reason for requiring an exact count per bin is because the histogram generated in step 1 is later fed into the prefix sum step that determines the starting address of every bin. And since the starting address of a bin is determined by summing the number of elements in all the previous bins, we need to maintain the exact number of elements that go into each bin, and that should not exceed the maximum bin depth. In addition, similar to the regular binning case, a CPU bin needs to be maintained at binning time which collects the overflow from all the GPU bins to execute them on the CPU simultaneously with the GPU kernel. Figure 29 depicts the execution model for the partitioned regular and compact bins.

Since the optimal bin depth is the one for which the runtime of the overflow on the CPU is the same as the runtime of the GPU execution, the optimal bin depth may vary depending on the runtime of the CPU and GPU kernels, and modifying either one may require retuning the bin

depth to maintain equilibrium. In fact the optimal bin size may even vary for the same code if run on an environment with a different CPU and/or GPU. Ideally, the bin depth should be computed automatically based on some performance model of the GPU and CPU, but this is not an easy task, and we consider it to be beyond the scope of this work.

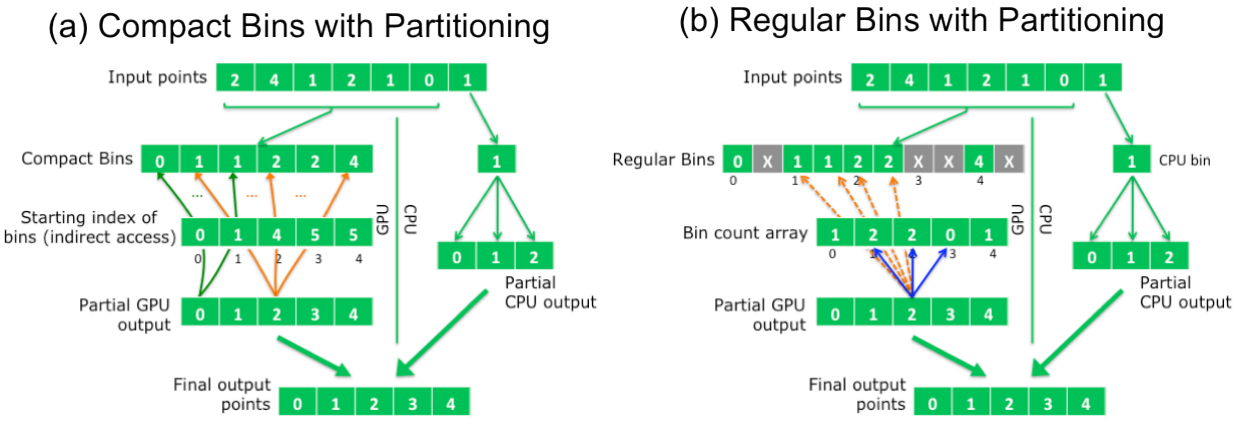


Figure 29. Partitioned execution of limited-range functions

5.2 Effects of Bin Depth on GPU Execution

In this section, we will analyze the effects of varying the bin depth on the runtime of regular and compact limited-range functions. A larger bin depth signifies more work is being put on the GPU rather than the CPU. In the case of regular binning, a bigger bin depth means more padding of the regular bins.

We plotted the runtime of the limited range function for regular and compact binning with varying bin depths. Figure 30 shows those results for all four benchmarks. As would be expected, the runtime increases with the increasing bin depth since more work is being performed by the kernel. Beyond that, padding does not seem to degrade the performance of the regular binning kernel. The regular binning implementation shown here is the one that uses the element count array, which means that regardless of the amount of padding, only the real

elements in every bin will be loaded into on-chip memory. There is a constant runtime gap between the regular bin kernel and the compact bin kernel, and that is most likely due to the access to the count array for every bin for the former, versus the one-time bound checking for each range in the latter (as shown in Section 4.3.2).

Due to the large output size for the MRI benchmark and the large imbalance in the input's distribution, any bin depth greater than 9 causes the regular bin data structure to exceed the global memory capacity. That is not the case for compact binning since the maximum size of the bin data structure is equal to the number of input elements, regardless of the maximum bin depth specified, and thus we can vary the bin depth arbitrarily as shown in Figure 30.a. That is in fact an important advantage of compact binning: the maximum bin depth for regular binning can often be limited by the size of memory, even if that bin depth does not achieve the optimal load balancing between the CPU and the GPU. One such case is the large MRI data set, which is not shown Figure 30. The number of bins for this data set is 576^3, and yet the majority of these bins are empty. Representing these bins in a regular format, even with a bin depth of 1, requires 4.27 GB of memory, which is more than the 4 GB available in the C1060 GPU. On the other hand, in the compact bin case, choosing a bin depth that is large enough to bin all of the input elements only occupies 0.67 GB of memory. As a result, the user is capable of better choosing a bin depth that balances the execution runtime between the CPU and GPU when representing bins in a compact format.

The performance improvement seen for Blinn's blob is the same shown in Figure 25. Because the Blinn's blob data set is so sparse, with the majority of bins having zero elements in them, compact binning achieves a large speedup compared to regular binning because zero-element bins do not consume any computation or bandwidth overhead in the former, whereas

they do in the latter.

It is worth observing how the varying bin depth affects the performance of N-body. Unlike the other three benchmarks, N-body does not demonstrate the same steady increase in runtime as the bin depth increases. The reason is that N-body does not preload bin contents into shared memory; rather, every thread loads the data that it needs immediately from global memory before using it. The resulting access pattern into global memory is a lot less regular than the other three benchmarks. Varying the bin depth simply randomizes the access pattern further and for some bin depths may result in better coalescing, whereas for others it may result in worse coalescing. The randomization effects are more noticeable for regular binning since the increase in bin depth causes the elements from two adjacent bins to move farther apart, due to padding, than in the compact case.

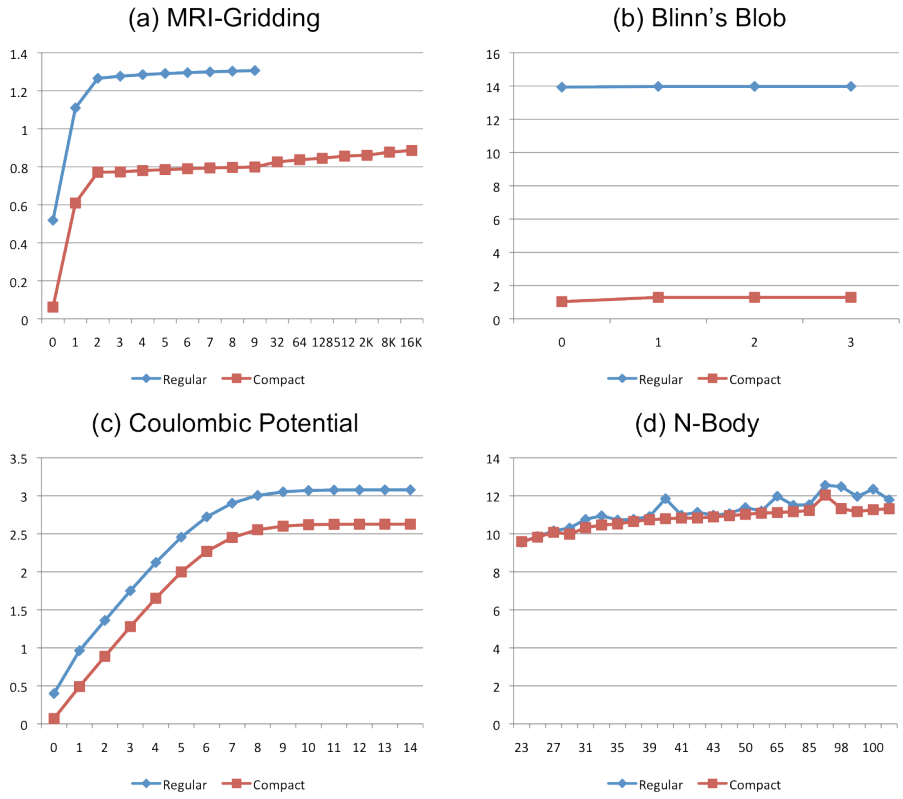


Figure 30. Regular vs. compact runtime for varying bin depths

5.3 Best Overall Performance

We have shown in Section 5.2 how increasing the bin size affects the execution runtime of the limited-range kernels. In this section, we will look at the overall performance of the overlapped execution of the CPU and GPU kernels. As mentioned previously, since the GPU kernel executes asynchronously with respect to the CPU, computing the two partial results can be done in parallel, and the overall runtime is equal to the greater of the two runtimes. That is why choosing a bin depth that makes the two runtimes equal yields the best overall performance. Figures 31, 32, 33, and 34 show the overall execution time for the limited-range function using regular and compact bins for all four benchmarks. A bin depth of 0 for all of them signifies that all the execution is performed on the CPU. Note that a bin depth of zero does not result in a zero runtime for the GPU since the kernel still needs to be launched and the size of each bin needs to be checked before realizing that there is no work to be done. The largest bin size shown on all graphs corresponds to all the input elements being assigned to the GPU (except for regular binned MRI, which exceeds memory capacity beyond a bin size of 9). Regardless of the binning format, a given bin depth results in the same number of elements being executed on the CPU, and therefore the same runtime. The column highlighted in red corresponds to the bin depth that yields the best performance for each case. The curves shown for compact and regular GPU execution are the same as the ones shown in Figure 30, and for the optimal bin depth, the speedup achieved is the equal in magnitude to the performance gap shown in Figure 30 (the performance of compact and regular bins is roughly the same for N-body).

We notice that for all the benchmarks, the GPU runtime is more resilient to an increase in workload than the CPU, in part due to the GPU's massive parallelism and greater number of resources compared to the CPU. This implies that despite the lack of an automated method for

determining the optimal bin depth, the users need not be exact in their choice of bin depth as long as they choose a bin depth large enough to reduce the CPU runtime below that of the GPU.

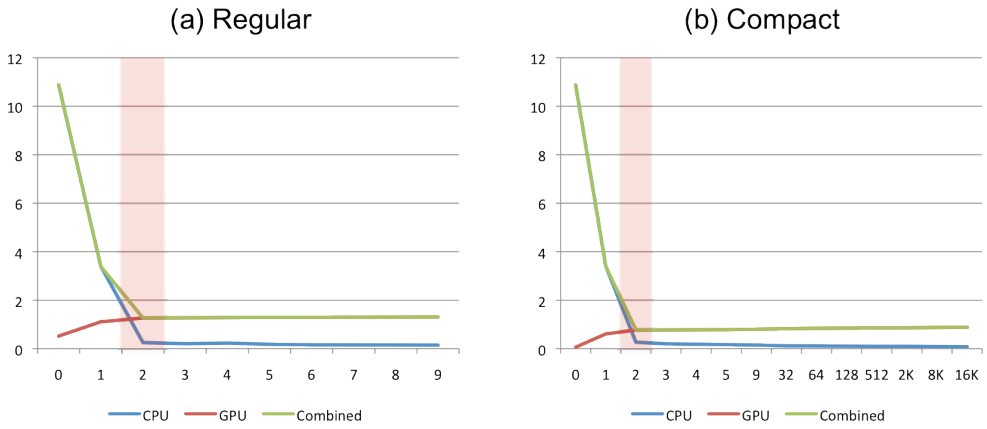


Figure 31. Overall MRI gridding runtime for various bin depths

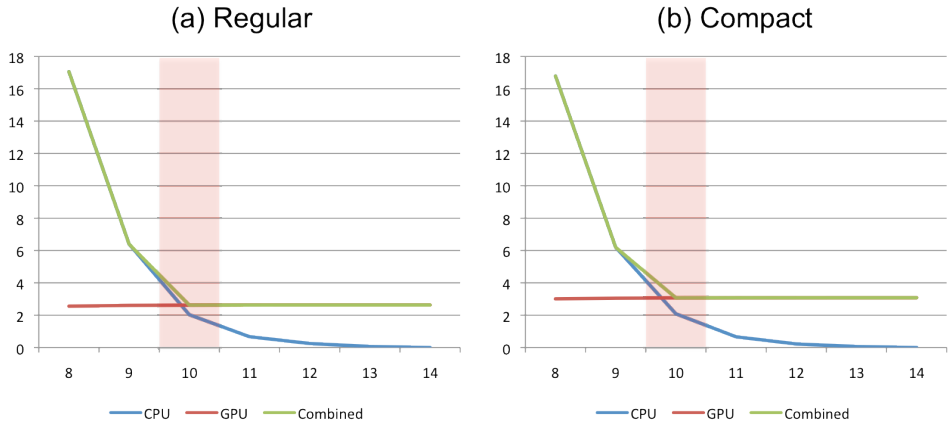


Figure 32. Overall Coulombic potential runtime for various bin depths

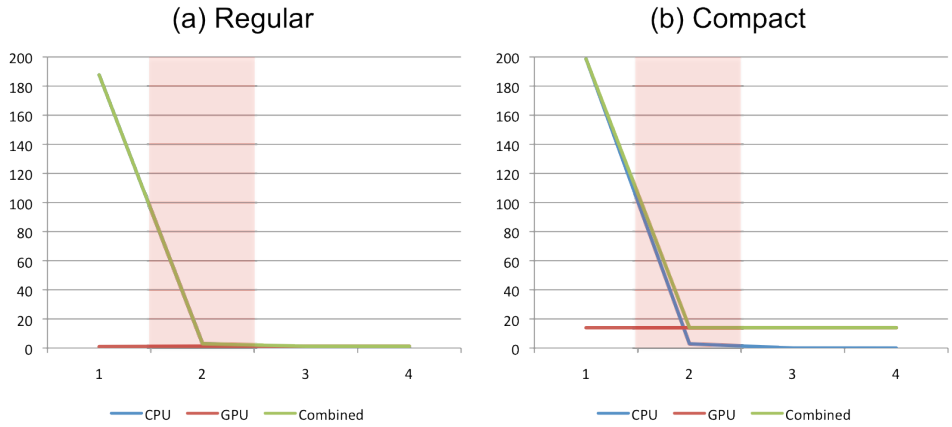


Figure 33. Overall Blinn's blob runtime for various bin depths

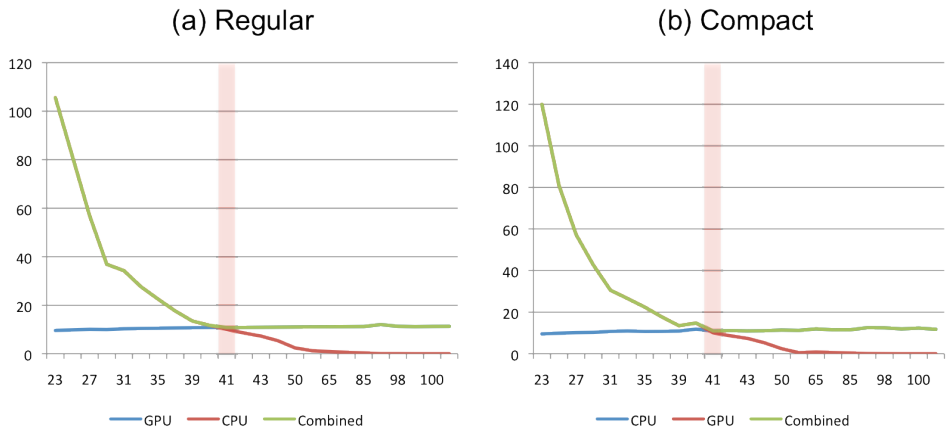


Figure 34. Overall N-body runtime for various bin depths

CHAPTER 6

COMPACTION IN RELATION TO SPARSE MATRICES

The binning concepts introduced in this work are not new concepts, but simply new applications of existing concepts. In particular, compaction and partitioning are techniques borrowed from sparse matrix representation. In this chapter, we will highlight the similarities and differences between the use of these techniques in the domains of sparse matrix multiplication and parallel limited-range functions. For an in-depth analysis of sparse matrix representations and their performance on GPUs, please refer to the paper written by Nathan Bell and Michael Garland on the topic [11]. For the purpose of comparing the two domains, we will only highlight the concepts that are relevant to the discussion.

The most rudimentary way to represent a sparse matrix is to store, for each non-zero element, its value, its column index, and its row index, in three arrays. This is known as the COO format (COO for coordinates). A sample matrix and its COO representation are shown in Figure 35. COO is in fact the most explicit way of representing sparse data since it maintains all the information of all the elements. A more efficient way to present the matrix data is the CSR format, which maintains the value and column index of each element, but sorts the elements based on their row index, and rather than maintaining a separate row index value for each element, simply maintains a starting index for all the elements of the same row. The row value of each element is therefore implied based on the offset range it belongs to (Figure 36). Compact binning is in fact closely related to the CSR format. All elements that fall into a bin are sorted in such a way that they are in contiguous memory locations, and accessing a certain bin, similar to accessing a row in CSR, is done by determining the starting offset of the bin, and the starting offset of the bin that follows.

In a matrix-vector multiplication kernel, each thread computes one element in the output vector, and the value of that vector corresponds to the dot product of one row of the matrix and the vector it is being multiplied by, and having each thread access one row causes memory requests to be non-coalesced. That is why it is more efficient to have an entire warp or block handle each row, such that threads are accessing consecutive elements in the sparse matrix data structure. In the end, the partial results from all the threads handling the same row are reduced down to a single value that corresponds to the value of the output element. Threads in a limited range function are also made to access consecutive elements in a bin for both an array of structures and a structure of arrays layout, as discussed in Section 4.3.3. The only difference between the two kernels is that threads within a warp in sparse matrix multiplications together compute the value of a single output element, whereas threads within a warp in a limited range kernel together load an input bin that they all need to compute different output elements.

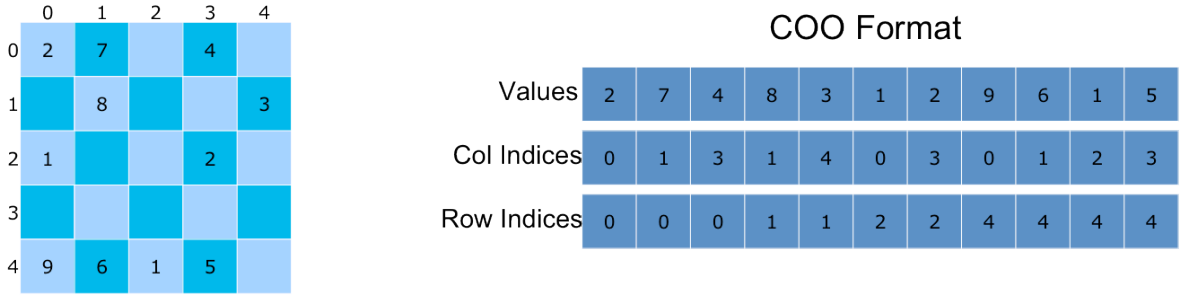


Figure 35. Dense matrix and COO representation

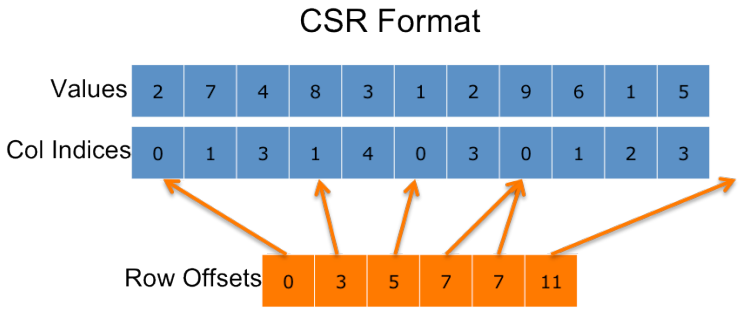


Figure 36. CSR format

Another efficient layout of sparse matrix data is the ELLPACK format (Figure 37). ELLPACK seeks to ensure that threads working on consecutive rows access the data within those rows in a coalesced manner. Since elements within a warp execute in lockstep, every thread will access the first non-zero element of its row simultaneously, then the second element simultaneously, and so on. Therefore to make sure that accesses are coalesced, all first elements need to be placed in consecutive memory locations, followed by all second elements, etc. Effectively, the ELLPACK format transposes the sparse matrix so that elements in consecutive rows become elements in consecutive columns. However, to achieve this transposition, all rows have to have the same number of elements in them; otherwise, the access to the transposed elements of the original row becomes difficult. To achieve this uniform row size, all rows need to be padded up to the maximum row size before performing the transpose. ELLPACK is not as useful in the context of limited-range kernels; however, the pre-transpose structure does resemble the dense binning representation, where the largest number of non-zero elements in a row corresponds to the maximum bin depth, and all the rows that have fewer than max depth elements in them are padded to achieve regularity.

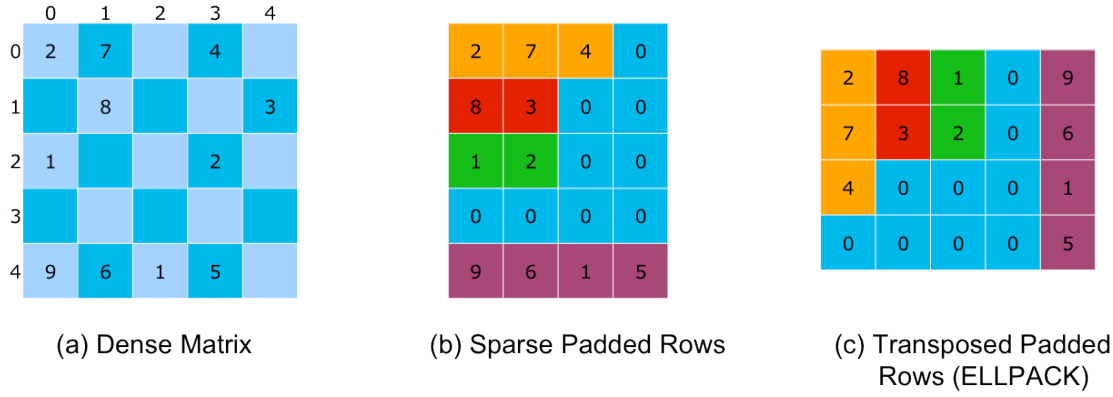


Figure 37. ELLPACK format

In fact, similar to dense binning, a large variance in the number of non-zero elements in each row of the sparse matrix causes a large memory bloat in ELLPACK due to padding

elements. The HYB (for hybrid) format seeks to reduce the overhead of padding. The HYB format as described in [11] is a combination of ELLPACK and COO (Figure 38.a). Rather than extending each row to the maximum row size, we instead find an average row size that keeps as many of the elements in ELLPACK format while minimizing the amount of padding needed. All elements that exceed this average row size get stored in a separate COO data structure, which can be executed by a separate CPU or GPU kernel. In addition to reducing the padding overhead, the HYB format improves load balancing for the execution of the ELLPACK data structure, as it reduces the variation in row size. Partitioning in limited-range applications is in fact a hybrid format, and achieves the same benefits of reduction of padding overhead and load balancing. Partitioning in compact binning is equivalent to a CSR/COO hybrid format (Figure 38.b).

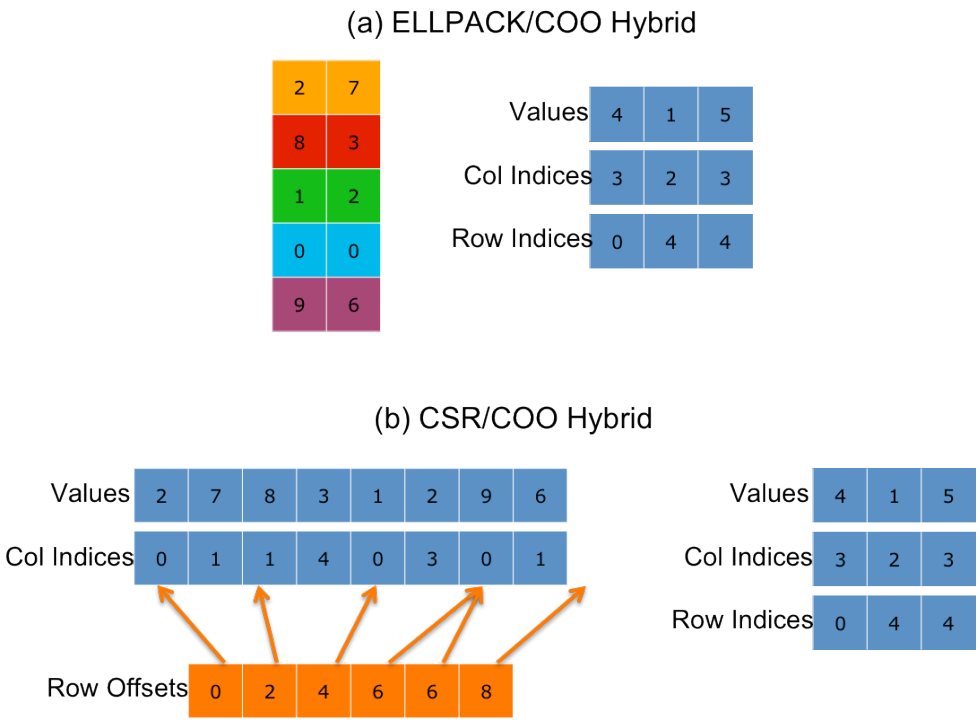


Figure 38. Hybrid format representations

CHAPTER 7

CONCLUSION

To say that we expected the results of this work would be untrue. Intuitively, one would expect that the added complexity of compact binning, while it may benefit certain applications and datasets, would prevent this approach from outperforming regular binning for all applications. However, upon further analysis, we were able to explain why compact binning can in fact outperform regular binning. One of the key factors in this speedup is the reduction of the number of loops and memory accesses due to the iteration over ranges of bins rather than individual bins when the bin data is compact. The only characteristic of compact bins that made this optimization possible, and the reason why the same optimization is disadvantageous for regular binning, is the elimination of padding elements from the bin data structure. Furthermore, we have demonstrated in Chapter 5 how compact binning can enable better load balancing between the CPU and GPU by overcoming the memory capacity barrier encountered with regular binning. Table 5 compares the results of the full execution (binning and partitioning, limited range computation, and CPU/GPU output reduction) for the best regular binning implementation with the best compact binning implementation. For regular binning, the best implementation consists of using a count array to determine the number of real elements in each bin before loading that bin into shared memory, as well as determining the best partitioning bin depth that balances the work done on the CPU and on the GPU. For compact binning, the best implementation consists of using range accesses within the smallest dimension of the space, using an array of structures for the input data, and finally, similar to regular binning, determining the best partitioning bin depth that balances work on the CPU and GPU. The first three applications all see an improvement in performance, and in the case of Blinn's blob, the speedup

is approximately a factor of 8x over regular binning. For N-body, which does not use any shared memory and therefore does not take advantage of range accesses into the bins, we at least do not see any loss of performance despite the added complexity of compact binning.

Table 5. Summary of compact and regular execution runtimes

	Regular	Compact	Speedup
CutCP	3.91	3.47	1.13x
MRI	1.47	0.98	1.50x
Blinn	14.42	1.81	7.98x
N-body	14.90	14.60	1.02x

REFERENCES

[1] "NVIDIA CUDA C Programming Guide," NVIDIA Inc., 22 Oct. 2010. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf

[2] I. J. Sung, J. Stratton, and W. M. Hwu, "Data layout transformation exploiting memory-level parallelism in structured grid many-core applications," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT) 2010*, Vienna, Austria, September 11-15, 2010.

[3] J. Benedetto and H. Wu, "Non-uniform sampling and spiral MRI reconstruction," in *SPIE-Wavelet Applications in Signal and Image Processing VIII*, vol. 4119, pp. 130-141, 2000.

[4] Y. Zhuo, X. L. Wu, J. Haldar, W. M. Hwu, Z. P. Liang, and B. Sutton, "Accelerating iterative field-compensated MR image reconstruction on GPUs," in *International Society for Magnetic Resonance in Medicine (ISMRM) 2010*, Rotterdam, The Netherlands, 2010.

[5] J. I. Jackson, C. H. Meyer, D. G. Nishimura, and A. Macovski, "Selection of a convolution function for Fourier inversion using gridding [computerized tomography application]," *IEEE Transactions on Medical Imaging*, vol. 10, no. 3, pp. 473-478, 1991.

[6] C. Rodrigues, D. Hardy, J. Stone, K. Schulten, and W. M. Hwu, "GPU acceleration of cutoff pair potentials for molecular modeling applications," in *CF'08: Proceedings of the 2008 Conference on Computing Frontiers*, 2008, pp. 273-282.

[7] J. M. Singh and P. J. Narayanan, "Real-time ray tracing of implicit surfaces on the GPU," *IEEE Transactions on Visualization and Computer Graphics*, vol. 99, pp. 261-272, 2009.

[8] J. Waltz, G. L. Page, S. D. Milder, J. Wallin, and A. Antunes, "A performance comparison of tree data structures for N-body simulation," *Journal of Computational Physics*, vol. 178, no. 1, pp. 1-14, 2002.

[9] N. Bell, "Optimizing parallel reduction in CUDA," NVIDIA Inc., n.d. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf

[10] S. Sengupta, A. Lefohn, and J. Owens, "A work-efficient step-efficient prefix sum algorithm," in *Proceedings of the Workshop on Edge Computing Using New Commodity Architectures*, 2006, pp. 26-27.

[11] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009, pp. 1-11.