Scalable Primitives for Data Mapping and Movement on the GPU

Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science (by Research)
in
Computer Science

by

Suryakant Patidar
200607023
skp@research.iiit.ac.in

International Institute of Information Technology
Hyderabad, India
June 2009

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "Scalable Primitives for Data Mapping and Movement on the GPU" by Mr. Suryakant Patidar (200607023), has been carried out under my supervision and is not submitted elsewhere for a degree.

_____ _____

Date Advisor: Prof. P. J. Narayanan

To Kanchan, My Beloved Mother.

"Be nobody but yourself in a world which is doing its best night and day to make you everybody else. It means to fight the hardest battle which any human being can fight and never stop fighting."

- E. E. Cummings

Abstract

GPUs have been used increasingly for a wide range of problems involving heavy computations in graphics, computer vision, scientific processing, etc. One of the key aspects for their wide acceptance is the high performance to cost ratio. In less than a decade, GPUs have grown from non-programmable graphics co-processors to a general-purpose unit with a high level language interface that delivers 1 TFLOPs for $400. GPU's architecture including the core layout, memory, scheduling, etc. is largely hidden. It also changes more frequently than the single core and multi core CPU architecture. This makes it difficult to extract good performance for non-expert users. Suboptimal implementations can pay severe performance penalties on the GPU. This is likely to persist as many-core architectures and massively multithreaded programming models gain popularity in the future.

One way to exploit the GPU's computing power effectively is through high level primitives upon which other computations can be built. All architecture specific optimizations can be incorporated into the primitives by designing and implementing them carefully. Data parallel primitives play the role of building blocks to many other algorithms on the fundamentally SIMD architecture of the GPU. Operations like sorting, searching etc., have been implemented for large data sets [19, 42].

We present efficient implementations of a few primitives for data mapping and data distribution on the massively multi-threaded architecture of the GPU. The *split* primitive distributes elements of a list according to their category. Split is an important operation for data mapping and is used to build data structures, distribute work load, performing database join queries etc. Simultaneous operations on a common memory is the main problem for parallel split and other applications on the GPU. He et al. [22] overcame the problem of simultaneous writes by using personal memory space for each parallel thread on the GPU. Limited small shared memory available limits the maximum number of categories they can handle to 64. We use hardware atomic operations for split and propose *ordered atomic operations* for stable split which maintains the original order for elements belonging to the same category. Due to limited shared memory, such a split can only be performed to maximum of 2048 bins in a single pass. For number of bins higher than that, we propose an *Iterative Split* approach which can handle billions of bins using multiple passes of stable split operation by using an efficient basic split which splits the data to fixed number of categories. We also present a variant of split that partitions the indexes of records. This facilitates the use of the GPU as a co-processor for split or sort, with the actual data movement handled separately. We can compute the split indexes for a list of 32 million records in 180 milliseconds for a 32-bit key and in 800 ms for a 96-bit key.

The *gather* and *scatter* primitive performs fast, distributed data movement. Efficient data movement is critical to high performance on the GPUs as suboptimal memory accesses can pay heavy penalties. In spite of high-bandwidth (130 GBps) offered by the current GPUs, naive implementation of the above operations hampers the performance and can only utilize a part of the bandwidth. The *instantaneous locality of memory reference* play a critical role in data movement on the current GPU memory architectures. For scatter and gather involving large records, we use collective data movement in which multiple threads cooperate on individual records to improve the instantaneous locality. We use multiple threads to move bytes of a record, maximizing the coalesced memory access. Our implementation of gather and scatter operations efficiently moves multi element records on the GPU. These data movement primitives can be used in conjunction with split for splitting of large records on the GPU.

We extend the split primitive to devide a *SplitSort* algorithm that can sort 32-bit, 64-bit and 128-bit integers on the GPU. The performance of SplitSort is faster than the best GPU sort available today by Satish et al. [40] for 32 bit integers. To our knowledge we are the first to present results on sorting 64-bits and higher integers on the GPU. With split and gather operations we can sort large data records by first sorting their indexes and then moving the original data efficiently. We show sorting of 16 million 128-byte records in 379 milliseconds with 4-byte keys and in 556 ms with 8-byte keys.

Using our fast split primitive, we explore the problem of real time ray casting of large deformable models (over a million triangles) on large displays (a million pixels) on an on-the-shelf GPU. We build a GPU-efficient three dimensional data structure for this purpose and a corresponding algorithm that uses it for fast ray casting. The data structure provides us with blocks of pixels and their corresponding geometry in a list of cells. Thus, each block of pixels can work in parallel on the GPU contributing a region of output to the final image. Our algorithm builds the data structure rapidly using the split operation for each frame (5 milliseconds for 1 million triangles) and can thus support deforming geometry by rebuilding the data structure every frame. Early work on ray tracing by Purcell et al. [35] built the data structures once on the CPU and used the GPU for repeated ray tracing using the data structure. Recent work on ray tracing of deformable objects by Lauterbach et al. [31] handles light models with upto 200K triangles at 7-10 fps. We achieve real-time rates (25 fps) for ray-casting a million triangle model onto a million pixels on current Nvidia GPUs.

Primitives we proposed are widely used and our results show that their performance scales logarithmically with the number of categories, linearly with the list length, and linearly with the number of cores on the GPU. This makes it useful for applications that deal with large data sets. The ideas presented in the thesis are likely to extend to later models and architectures of the GPU as well as to other multi core architectures. Our implementation for the data primitives viz. split, sort, gather, scatter and their combinations are expected to be widely used by future GPU programmers. A recent algorithm by Vineet et al. [47] computes the minimum spanning tree on large graphs used the split primitive to improve the performance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A graphics processing unit or GPU (also occasionally called visual processing unit or VPU) is a specialized processor that offloads 3D graphics rendering from the microprocessor. It is used in personal computers, workstations, game consoles, embedded systems, mobile phones etc. Modern GPUs are very efficient at manipulating images and at computer graphics, and their highly parallel structure makes them more effective than general-purpose CPUs for a range of complex algorithms. In a personal computer, a GPU can be present on a video card, or it can be on the motherboard. More than 90% of new desktop and notebook computers have integrated GPUs, which are usually far less powerful than those on a video card.

GPUs have been increasingly used for a wide range of problems involving heavy computations in graphics, computer vision, scientific processing, etc. The main attraction is the high computation power per unit cost; today's off-the-shelf GPUs deliver 1 TFLOPs of single precision power for under $400. The programming model available on them have also become general purpose with the advent of CUDA [33] and the newly-adopted standard of OpenCL [29]. However, extracting the best performance from the GPU requires a deep knowledge of its internal architecture including the core layout, memory, scheduling, etc. A lot of this information is not publicly available; the architecture also changes more frequently than the architecture of multi-core microprocessors.

1.1 General Processing using GPU

A new concept is to use the GPU as a modified form of a stream processor to allow a general purpose unit. This turns the massive floating-point computational power of a modern graphics accelerator's shader pipeline into general-purpose computing power, as opposed to being hard wired solely to do graphics operations. In certain applications requiring massive vector operations, this can yield several orders of magnitude higher performance than a conventional CPU. The two largest discrete GPU designers, ATI and Nvidia, started to pursue this new market with an array of applications. Both Nvidia and ATI, for example, have teamed with Stanford University to create a GPU-based client in the early 2000s for the Folding@Home distributed computing project (for protein folding calculations). In certain circumstances the GPU is about forty times faster than the conventional CPUs traditionally used for such applications.

Recently, Nvidia began releasing cards supporting an API extension to the C programming language called CUDA (Compute Unified Device Architecture), which allows specified functions from a normal C program to run on the GPU's stream processors. This makes C programs capable of taking advantage of a GPU's ability to operate on large matrices in parallel, while still making use of the CPU where appropriate. CUDA is also the first API to allow CPU-based applications to

access directly the resources of a GPU for more general purpose computing without the limitations of using a graphics API.

1.2 GPU Applications

One way to exploit the GPU's computing power effectively is through high level primitives upon which other computations are built. All architecture specific optimizations can be incorporated into the primitives by designing and implementing them carefully. An application can gain high performance if its computationally intensive parts can be broken down into such primitives.

The high computational rates of the GPU have made graphics hardware an attractive target for demanding applications such as those in signal and image processing. Among the most prominent applications in this area are those related to image segmentation as well as a variety of other applications across the gamut of signal, image, and video processing. The segmentation problem seeks to identify features embedded in 2D or 3D images. A driving application for segmentation is medical imaging. A common problem in medical imaging is to identify a 3D surface embedded in a volume image obtained with an imaging technique such as Magnetic Resonance Imaging (MRI) or Computed Tomography (CT) Imaging. Motivated by the high arithmetic capabilities of modern GPUs, several researchers have developed GPU implementations of many signal processing operations such as convolutions and the fast Fourier transform. Computer vision algorithms now use graphics hardware to accelerate image projection and compositing operations in a camera-based head-tracking system. Yang and Pollefeys [52] used GPUs for real-time stereo depth extraction from multiple images. Variety of applications in the field of audio and signal processing, computational geometry, data parallel algorithms, databases, data compression and data structures have been efficiently coded for the GPU.

With the introduction of CUDA architecture, GPGPU problems are now addressed with a much simpler API for GPU. Operations like sorting, searching and other data structure problems have been efficiently addressed for large data sets [19, 42]. Data structures have applications in multiple fields and efficient implementation of basic primitives have been addressed with their applications in various fields [22].

The map-reduce primitive has recently been very effective in distributing compute intensive applications to a cluster of processors [12]. GPUs essentially follow the data parallel model in which kernels of code are applied on many data elements in parallel. Blelloch defined several data parallel primitives including scan, reduce, and binary split [6, 4]. Sengupta et al. implemented these primitives on the GPU under CUDA [42] which has been made available under the CUDPP library [19]. Several applications have been built using these primitives including kd-trees [55], octrees [54], BVH trees [31], etc.

Efficient parallel implementations of basic data structures is essential for the adoption of parallel architectures like the many-core GPUs and CPUs to general purpose computing. Many applications need fast basic primitives to achieve best performance from accelerators like the GPU. Latest GPUs provide up to 240 programmable cores and deliver 1 TFLOP of performance. CPUs have moved to 4 and 8 cores for high end machines and at least 2 cores on basic machines. We will see many core CPUs that can be specialized for specific application areas, such as the Larrabee from Intel for graphics processing [41]. With the growing number of cores available, we need algorithms which are scalable over a wide range of current and forthcoming parallel architectures.

Algorithm 1 Split (L)

```
1: I Count the number of elements falling into each bin.
2: for each element x of list L do                          [Can be done in parallel]
3:    bin ← category(x)
4:    histogram[bin]++                                      [Multiple elements may clash on a bin]
5: end for
6:
7: II Find the starting index for each bin using a prefix sum
8: startIndex[0] ← 0
9: for each category m do
10:   startIndex[m] ← startIndex[m-1] + histogram[m-1]      [Use Parallel Prefix Sum(Scan)]
11: end for
12:
13: III Assign each element to the output within its position, incrementally
14: localIndex[x] ← 0
15: for each element x of list L do                         [Can be done in parallel]
16:   bin ← category(x)
17:   itemIndex ← localIndex[x]++                            [Atomic Read-and-Increment required]
18:   globalIndex ← startIndex[bin]
19:   outArray[globalIndex + itemIndex] ← x
20: end for
```

1.3 Split Operation

The split primitive is widely used in database operations and for building various data structures. It requires random memory access which is tricky on parallel architectures. Split has been implemented on the GPU by He et al. [22] and was used to perform relational joins on the GPU. Sort is another basic primitive needed by many operations. A number of sort implementations have appeared for the GPU. Algorithms like quick sort, radix sort, merge sort, bitonic sort have been implemented on the GPU recently.

Split can be defined as appending each input element x to a list of the category it belongs to, or performing **append**$(x, List[\textbf{category}(x)])$. *List* holds all elements of a particular category. Split is a function that divides an input relation into a number of partitions. Each element could also be part of multiple categories. This results in non-disjoint partitions and an increased size of the output relation. Multi-split can be handled by replicating input element that are part of multiple categories. A compact function following a split packs the lists of individual categories together so that the input and output lists have the same length.

Problem: We consider the problem of splitting an input list of N elements to M categories or bins. Our split is a length preserving operation. The output is a list of length N with elements of each category stored together and the categories in a sorted order as shown in Figure 1.1.

Split can be implemented using a 3 step process sequentially as given in Algorithm 1. Step I (Line 1) and step III (Line 13) are very similar. Step I limits itself to counting (building the histogram) while step III performs index increment and data copying. Each data element is examined once in step I and step III along with the corresponding binID. Step 2 (Line 7) goes over all the bins. Sequential split thus has a complexity of $O(N + M)$, where N is the length of the input list and M is the number of categories.

Split primitive operates on a list of pairs of element and category. It rearranges the list and

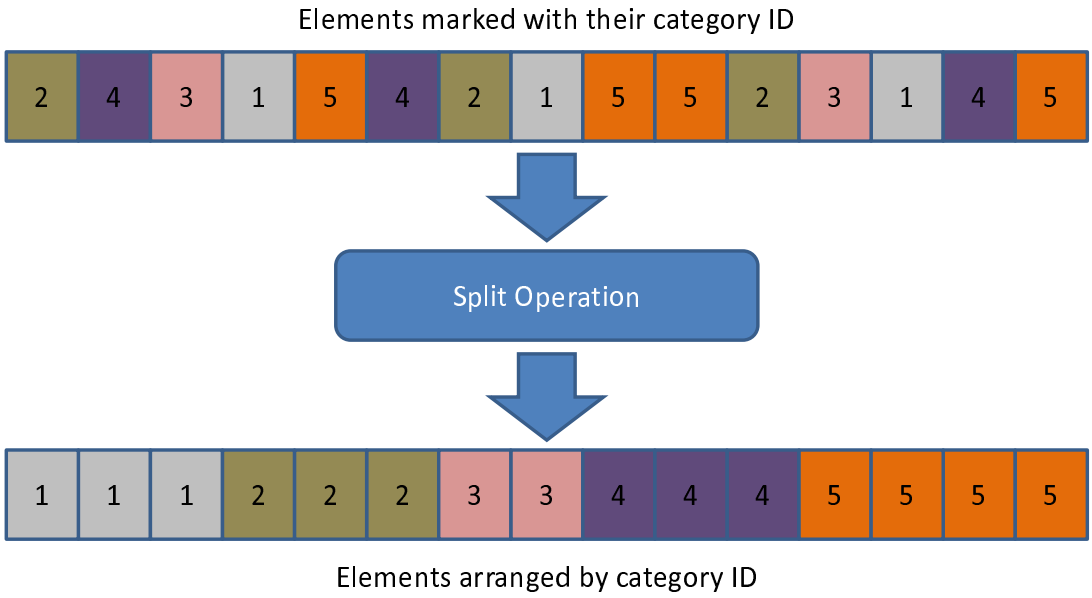


Figure 1.1: Split produces a compact list with the categories in order and elements of each category together

brings together the elements of the same category. Stability of the split operation can be defined in a manner similar to that of sorting. Elements belonging to the same category should appear in the initial order of the input list to ensure the stability of the split operation. Split primitive does not restrict the order of categories, i.e. different categories can be placed in any order in the output list. If these categories are ordered and are placed in ascending/descending order, split essentially reduces to sorting. Implementing split on parallel machines is tricky due to the parallel updating of the category lists. Efficient split operations can be central to several operations, especially those building and manipulating non-trivial data structures. We present the implementation of split on the GPUs using the CUDA model in the rest of this paper.

1.4 Scatter and Gather Operation

Scatter and Gather operation involve movement of records from one location to another. Many primitives like sort process the input and produce new location for each record in the input. This movement of data needs to be performed efficiently on the GPU, by maximizing the bandwidth between the processor and device memory. Peek bandwidth for current state of the art GPU, Nvidia GTX280, is around 130GB/sec which is achieved with coalesced reads and writes to the GPU.

Coalescing is a dual concept of caching on uniprocessors. Caching improves the performance in the presence of temporal locality in memory accesses by the same thread. Coalescing improves the performance when there is *instantaneous locality* in the memory references by a block of consecutive threads, as the accesses are combined into a minimum number of expensive memory transactions. Completely coalesced reads can be a factor 50-100 times faster than a totally random read on current GPUs. Previous generation GPUs (Nvidia G80) would achieve coalesced read/write if consecutive threads access to consecutive locations in the main memory for a half-warp of threads (16 threads). A half-warp could make a transaction of 64 bytes (16 * 4 bytes) or 128 bytes (16 * 8 bytes) with the

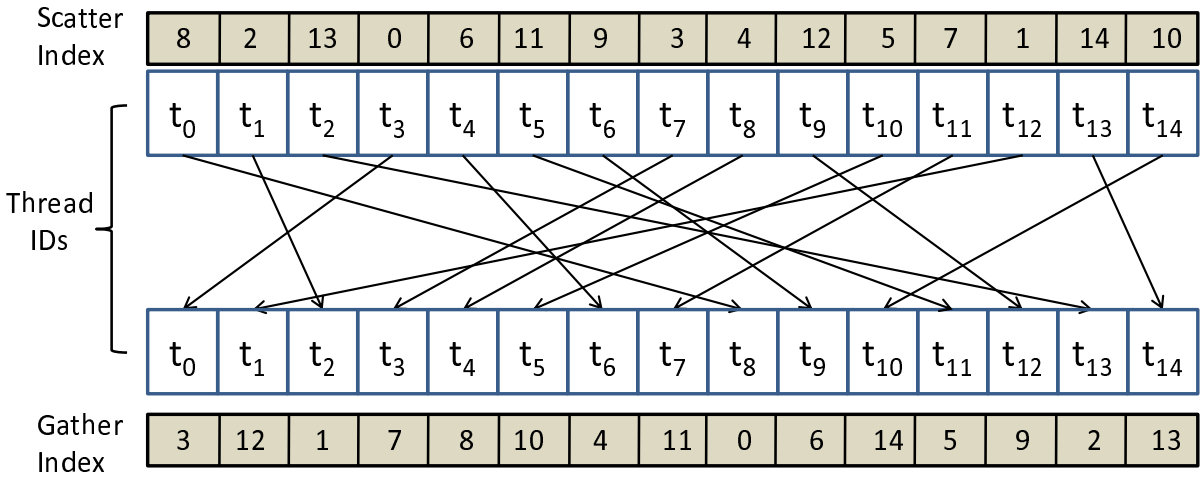


Figure 1.2: Scatter operation forces consecutive threads to write random locations in the memory causing non-coalescing

cost of a single memory fetch if the transaction is coalesced. Current generation GPUs (GTX280 and later) have redefined coalescing as a half-warp reading from a segment of 64 bytes or 128 bytes. Thus, threads need not read/write from consecutive locations but can read/write from any location within the segment.

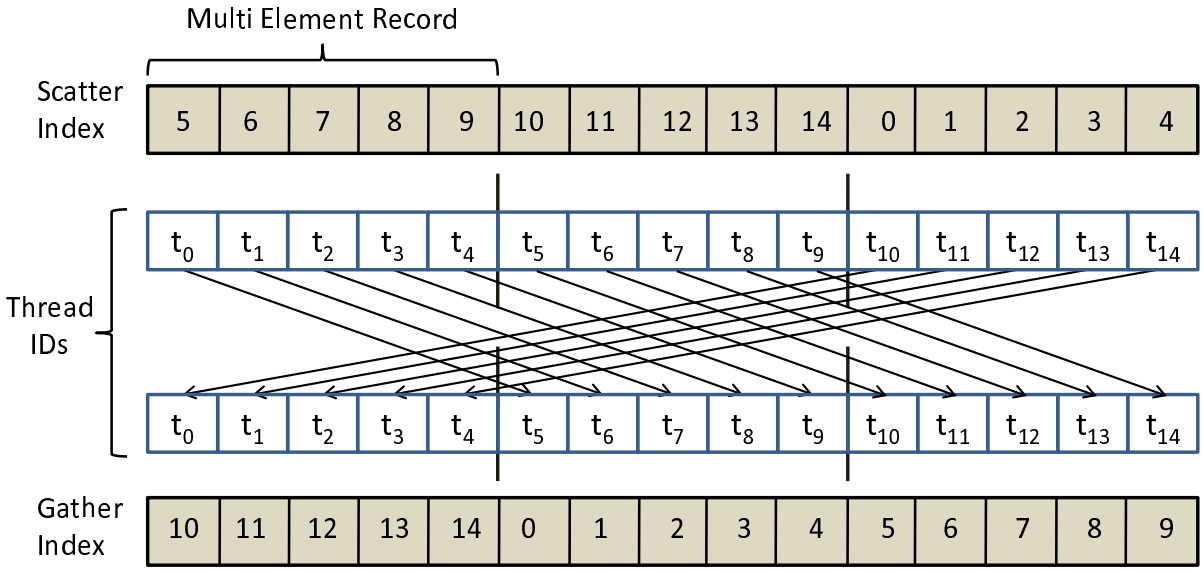


Figure 1.3: Multi element records benefit from the approach where consecutive threads write to contiguous locations in the memory

Performance of non-coalesced reads and writes can degrade given the limited bandwidth of the global memory. A global memory operation costs around 400-500 clock cycles. These transactions are optimally scheduled in order to gain as much performance as possible. Processes which are computationally heavy can successfully hide this cost but memory intensive applications on the

GPU suffer from the slow bandwidth. Figure 1.2 shows how non-coalesced writes are performed during scatter operation for single element records. For large records (multiple elements of 4 or 8 bytes). Figure 1.3 shows data movement can be coalesced by moving elements of a record using consecutive threads. Data movement on the CPU is performed sequentially and would automatically benefit from caching for multi element records. GPUs on the other hand should be explicitly programmed to perform efficient data movement of large records. An alternative approach where consecutive threads move same element of different records would cause non-coalescing even in case of multi element records.

1.5 Contributions of the thesis

In this thesis, we present efficient and scalable implementations of two data mapping primitives with wide applications on the GPU. Data mapping and distribution are used in distributed applications for data structure building, load balancing, etc. We present the *split* primitive that distributes data elements based on a category each belongs to. This is a generalization of the binary split and has been found to be critical for all throughput computing [11]. We also define index-variants of split that defer the actual data movement, which is useful to handle bulky data records. We also present efficient implementations of the data movement primitives *scatter* and *gather* on the GPU that work in conjunction with the index-variants of split. The separation of index computation from data movement enables the use of the GPU as a fast co-processor for split and sort, with data movement handled separately. The GPU performance is highly sensitive to memory access patterns; suboptimal implementation can pay heavy penalty. Optimized primitives are basic building blocks using which many regular and irregular applications can be built.

The main contributions of this thesis are the following:

1. A split algorithm that stores a single copy of the histogram on each block in the shared memory and uses shared memory atomic operations. This is faster than the split by He et al. [22] and extends the number of bins to split from 64 to 2048 for a single pass.

2. A multi-level approach that can split the input list to progressively defined bins. The Multi-Level Hierarchical Split algorithm treats the bins as a hierarchy and splits different sublists independently after the first level. This further extends the number of bins that can be handled efficiently to 2 million. Performance degrades beyond that.

3. Ordered Atomic operations that resolve clashes between multiple processes in the order of a user-supplied priority value. We define the operation and provide a serialized implementation on the current GPUs that may be 5 to 10 times slower than a possible hardware implementation.

4. A Multi-Level Iterative Split algorithm that splits iteratively to sub-bins based on grouping of bits of the bin ID from the right to left. The work done is the same in each iteration and the algorithm scales linearly in the number of bits of the bin ID to any length.

5. A SplitSort algorithm that reduces sort to a iterative splits to a number of bins equal to the range of the input values. This algorithm is linear in the input size, logarithmic in the range of values, and linear in the number of cores available. It can also extend to keys of any length. We show results of sorting up to 64-bit numbers on lists of length up to 64 million in well under one second.

6. We improve and exploit the *instantaneous locality of memory* references to improve the data movement performance on the GPU. The coherence in memory access between different compute elements is critical to memory performance on the GPUs, like caching on the CPUs.

7. We present efficient implementations of the gather and scatter primitives for fast data movement within the GPU, taking advantage of the instantaneous locality of memory references. We can randomly scatter 16 million 128-byte records in about 290 milliseconds and 8 million 256-byte records in under 150 ms.

8. We present efficient and scalable implementations of the split and split-index primitives for the GPU. These can be used to split a list to its category or to sort a list of numbers. We can sort 128 million 32-bit numbers in about 650 milliseconds and 128 million 128-bit numbers in about 4 seconds. We can compute the split-index for 64 million records with 32-bit keys in 350 milliseconds.

9. We show applications of the split and gather primitives for several operations. Sorting large records maps to a split-index followed by a gather. We show sorting of 16 million 128-byte records in 379 milliseconds with 4-byte keys and in 556 ms with 8-byte keys. We also discuss how these primitives can be used to build distributed data structures for applications like ray tracing.

10. We use the split data primitive for building efficient data structure for ray casting of triangulated deformable models. We ray cast a million triangle deformable model onto a million pixel window in real time. CPUs are not capable of performing ray casting of heavy deformable models at real time rates. Even with the introduction of multi core CPUs, it is difficult to achieve the above.

Chapter 2

Background and Related Work

GPUs were originally hardwired for specific graphics tasks. As transistor budgets and demand for flexibility grew, the hardware became more programmable. They still contain special hardware and functional units specific to graphics tasks, but today's GPUs can be considered as compute accelerators.

2.1 GPU Architecture

A typical GPU design, shown in the Figure 2.1, is abstracted from information put out by Nvidia. High performance of the GPUs is due to the large number of thread processors (TP). A set of TPs together with local memory and register file is called a multi processor (MP). The state of the art GPUs by Nvidia are equipped with up to 30 MPs each with 8 TPs. The GPU has a theoretical processing power of up to 1 TFLOPS. Each multiprocessor executes in SIMD mode, i.e., each thread processor in a multiprocessor executes the same instruction simultaneously. Each instruction is quad-clocked with a SIMD width of 4, even though there are only eight thread processors. Nvidia defines these mode of single instruction multiple threads as SIMT execution. In the figure, there are eight thread processors in each of M multiprocessors (MP), for $M * 8$ TPs total.

The multiprocessors execute asynchronously. There is no communication across multiprocessors, although there is a synchronization mechanism for groups of threads running on a multiprocessor. Threads from a multiprocessor can also communicate through the shared memory common to that multiprocessor. Multiprocessors can only communicate via the high latency off-chip global memory. The operations on the common global memory locations by multiple processors do not follow any order, making it hard to communicate. Only one of the multiple simultaneous operations on the memory succeeds, making it non-deterministic in nature. Atomic operations on the hardware are meant to overcome above problem; they guarantee all operations to succeed but no information on ordering of the operations is known.

Each multiprocessor has a special function unit which performs operations like, divide, square root, etc. It is slower compared to other processing units but its infrequently used. Each multiprocessor has a high bandwidth, low latency on-chip local memory (shared memory). Shared memory is a valuable resource but is limited to 16KB on current generation GPUs. There is also a high-bandwidth, high-latency large off-chip global device memory, over 1GB on high-end models. Device memory needs to be filled via the PCIe bus by the host from the host memory. Data on the GPU device memory can be pulled back to the host memory in a similar way. Recent software improvements have enabled the use of host memory from the device, thus extending the amount

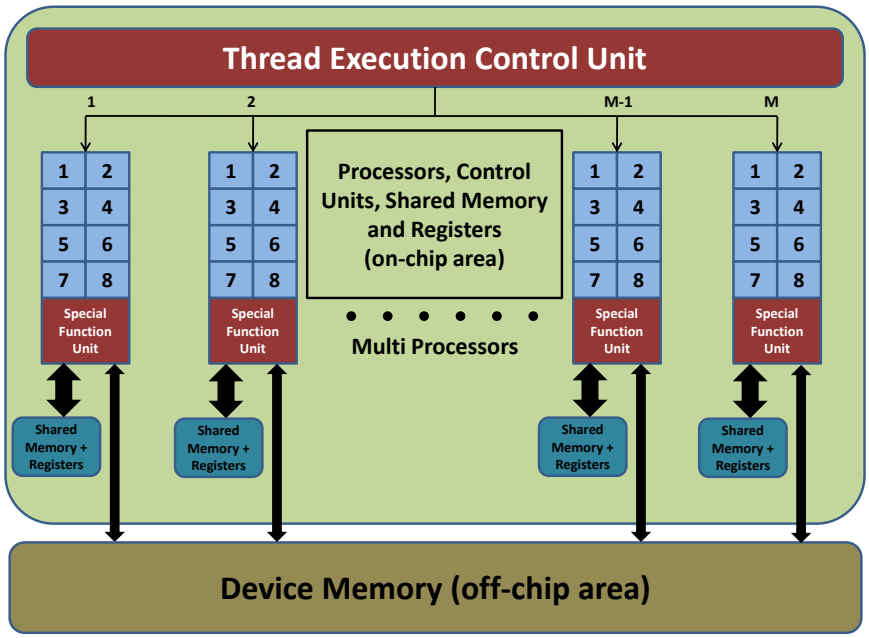


Figure 2.1: GPU Architecture for current generation of Nvidia GPUs. Processors from other vendors have a similar architecture.

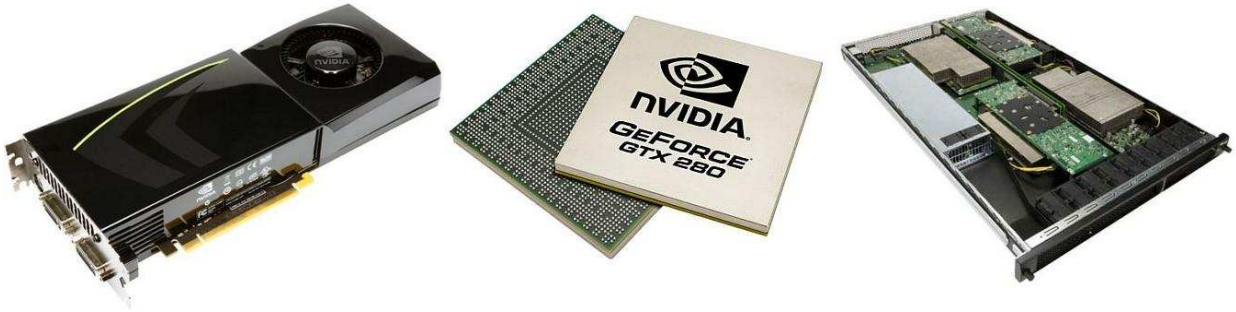


Figure 2.2: Nvidia GTX 280, Nvidia GPU Chip for GTX 200 Series and Nvidia Tesla 4 GPU System shown above. Images obtained from freely available documents provided by Nvidia.

data which can be processed beyond GPU device memory.

Eight thread processors are grouped together to form a multiprocessor and several multiprocessors are combined to form a device in the hardware model. Threads are combined together as *thread blocks* in order to group large number of threads. These thread blocks form a *grid* of blocks, which are processed on the GPU using a kernel. Each thread block is executed on a multiprocessor and thread of the block can communicate through the shared memory of the multiprocessor. Synchronization can also be triggered for the threads of a block using barriers. More than one thread block can occupy a multiprocessor. These thread blocks are time-shared by interleaving *warps* of threads. No ordering of the threads warps is guaranteed. The number of thread blocks which can be handled by a multiprocessor depends on the resources used by each thread block. Private partitions in the shared memory and register file are logically created for each thread block that occupies a multiprocessor.

A large number of threads is required for extracting enough parallelism on the GPU. The SIMD

instructions being executed on each multiprocessor may stall the processors if the instruction takes a long time. A memory request from the global memory may take as much as 500 clock cycles.

Time sharing a large number of threads on a multiprocessor can improve the overall throughput of instructions. Thread warps which are context switched can belong to the same block or different blocks being executed on a multiprocessor.

GPUs are also designed for structured data accesses, as they are designed for graphics processing. The memory is efficient when transactions are performed in an ordered manner by the threads.

The current generation GPUs coalesce as long as all the reads/writes are from a block of memory of 64 or 128 bytes. Shared memory has a low latency but can suffer from bank conflicts. Shared memory of 16KB is divided into 16 banks of 1KB each. Each consecutive word of the memory is placed in consecutive banks, making it possible to read 16 words by a single half-warp (16 threads) without any bank conflicts. In case of a bank conflict, the requests are serialized.

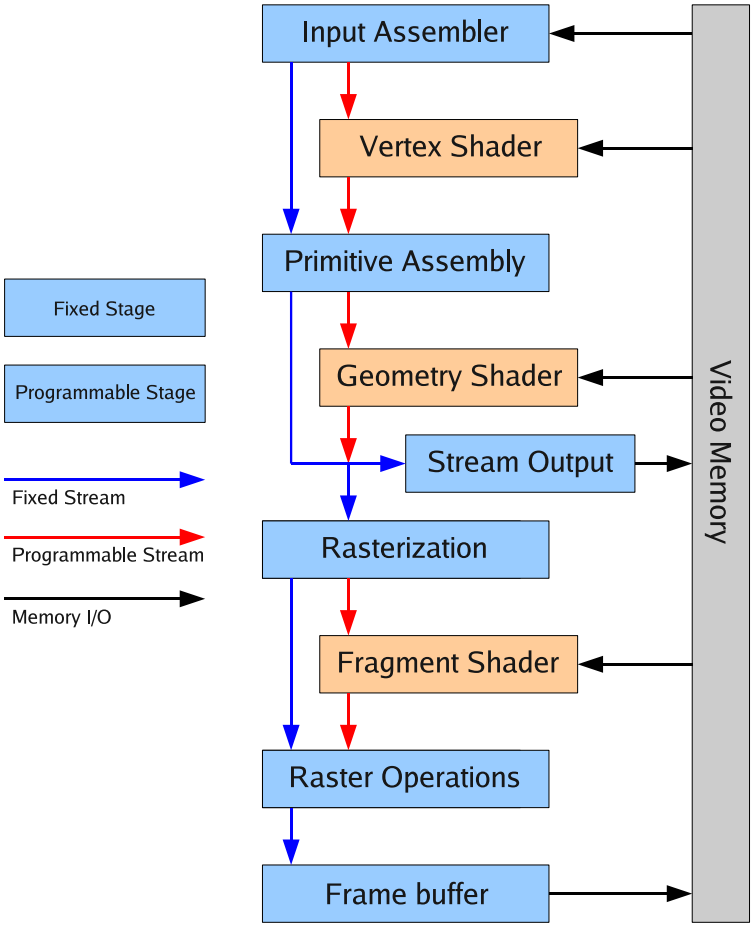


Figure 2.3: Shader Model 4.0 Pipeline. Geometry Shader was introduced as new programmable unit in the pipeline.

Above resources and limitations should be kept in mind for an efficient implementation on the GPU. Data common to a thread block and required for processing more than once should be stored in the shared memory. The life of the data in shared memory is that of the thread block and that data can not be referred by other thread blocks. Data transaction from the global memory should be coalesced in nature to achieve many fold performance as compared to non-coalesced reads and writes.

Each multiprocessor is equipped with a thread control unit (Figure 2.1) which manages the scheduling of threads from multiple blocks assigned to the multiprocessor. All 8 thread processors perform the same instruction in a clock cycle. Multiprocessors also have the synchronization units which allow synchronization of threads from a block. The most common use of synchronization is to maintain data consistency when multiple threads are used to read data from global memory to shared memory, assuming the data brought in can be used by other threads of the block.

2.1.1 GPGPU

Programmability of the GPUs has grown over the last decade. Shader model 3.0 introduced fully programmable vertex and pixel processing units. Support for vector operations on IEEE single precision floating point numbers was introduced. Many high level languages like Cg by Nvidia, open standard GLSL, HLSL by Microsoft etc. for shader programming made it easier to program the GPU and access the computation power. It was studied that GPUs could accelerate some problems by an order of magnitude over the CPU. With the introduction of shader model 4.0, an additional geometry generation unit was added to the pipeline (Figure 2.3). General-purpose application on the graphics processing unit (GPGPU) were mostly addressed through the pixel shader unit and was neither affected nor gained much with the new shader model.

The GPGPU approach could address various non-graphics problems like in-game simulation of physics and computational science. Given the earlier development on the GPUs was focused on graphics applications, the programming environment was tightly constrained. Lack of exposure to the underlying architecture also made it hard for non-graphics developer to port their applications to the GPU. The developer was expected to be an expert in computer graphics in order to make effective use of the GPU.

General purpose computation used a Stream Processing model where a series of operations (kernel functions) are applied to each element from the set of data (a stream). A typical GPGPU problem is mapped as a texture manipulation problem using the graphics pipeline. The main source of input and output data containers are textures which earlier had access only from pixel shaders but now can also be accessed from vertex and geometry shader (Figure 2.3) due to a unified shader model introduced with shader model 4.0. The major limitation of this model was the limited scope of writing an output since scatter was not supported. This approach had a significant learning curve but yet provided opportunities for extreme speedups for selected applications.

2.1.2 CUDA Architecture

CUDA is a programming interface to the parallel architecture of the GPU for general purpose computing. This interface is a set of library functions which is coded as an extension of the C language. A compiler generates executable code for the CUDA device. The CPU sees a CUDA device as a multi-core co-processor. The CUDA design does not have memory restrictions of GPGPU. One can access all memory available on the device with no restriction on its representation though the access times vary for different types of memory. This enhancement in the memory model allows programmers to better exploit the parallel power of the GPU for general purpose computing.

CUDA Hardware Model: At the hardware level the GTX 280 processor is a collection of 30 multiprocessors, with 8 processors each. Each multiprocessor has its own shared memory which is common to all the 8 processors inside it. It also has a set of 32-bit registers, texture, and constant memory caches. In any cycle, each processor of the multiprocessor executes the same instruction on different data. Communication between multiprocessors is through the device memory, which is available to all processors of the multiprocessors.

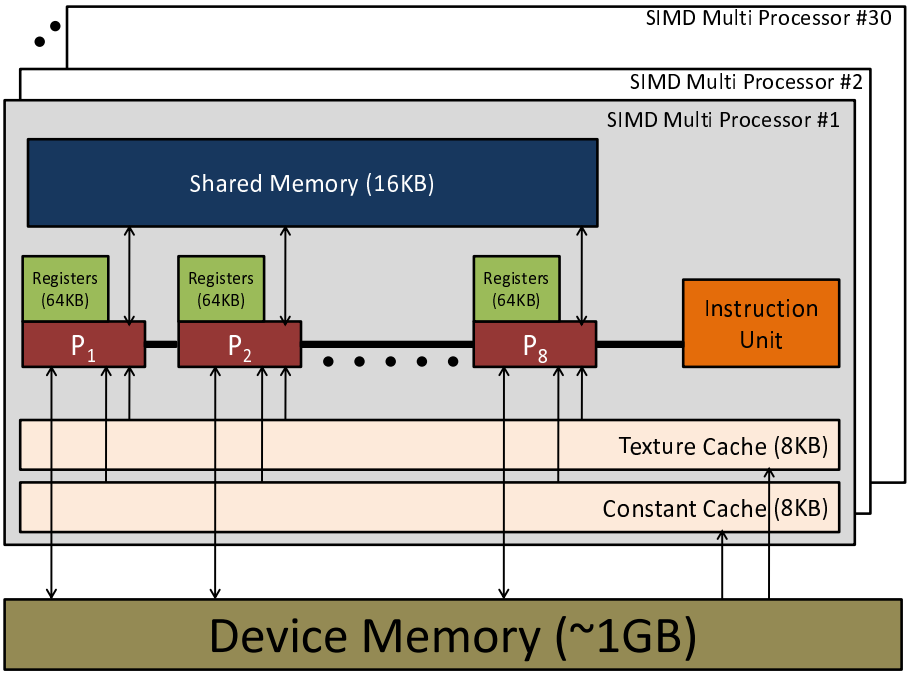


Figure 2.4: Nvidia CUDA Hardware Model.

CUDA Programming Model: For the programmer, the CUDA model is a collection of threads running in parallel. A warp is a collection of threads that can run simultaneously on a multiprocessor. The warp size is fixed for a specific GPU, 32 on present GPUs. The programmer decides the number of threads to be executed. If the number of threads is more than the warp size, they are time-shared internally on the multiprocessor. A collection of threads (called a block) is mapped to a multiprocessor at a given time. Multiple blocks can be assigned to a multiprocessor and their execution is time-shared. A single computation on a device generates a number of blocks. A collection of all blocks in a single computation is called a *grid*. All threads of the blocks mapped to a multiprocessor divide its resources equally amongst themselves. Each thread and block is given a unique ID that can be accessed within the thread during its execution. Each thread executes a single instruction set called the kernel. GPU is a co-processor to the CPU and needs to be initiated by the CPU. A typical CPU/GPU application is executed in the following order when initiated by the CPU.

1. Copy data from main memory to GPU memory
2. CPU instructs the process for GPU execution
3. GPU executes the program in parallel using many cores
4. Copy the results from GPU memory to main memory

GPGPU approach using the graphics pipeline had a steep learning curve due to unfamiliarity of programmers with the graphics APIs. CUDA has several advantages over traditional general purpose computation on GPUs (GPGPU) using graphics APIs.

1. Scattered reads/writes: Code can read from arbitrary addresses in memory.

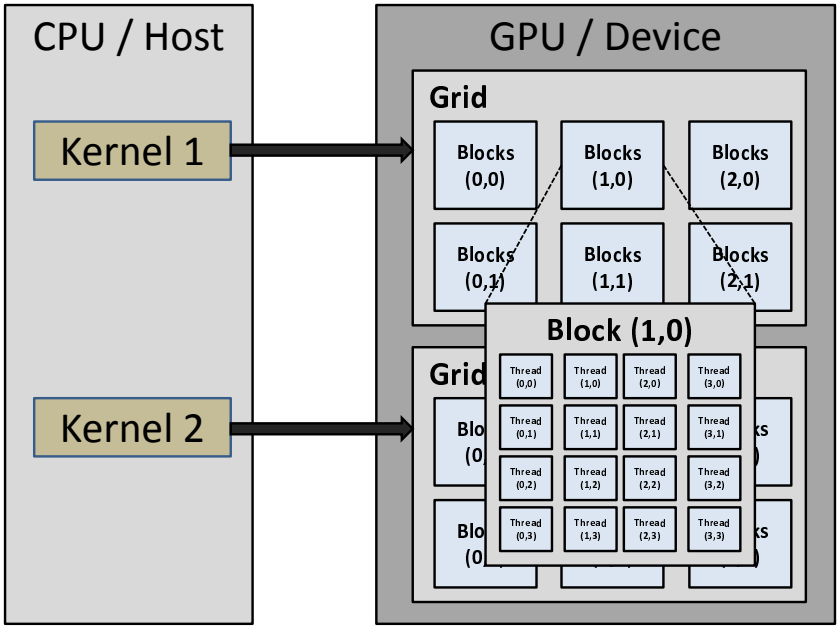


Figure 2.5: Nvidia CUDA Software Model.

2. Shared memory: CUDA exposes a fast shared memory region (16KB in size) that can be shared amongst threads. This can be used as a user-managed cache, enabling higher bandwidth than is possible using texture lookups

3. Faster downloads and read-backs to and from the GPU

4. Full support for integer and bitwise operations, including integer texture lookups.

The following are some of the limitations of CUDA model when compared to sequential programs and the GPGPU.

1. It uses a recursion-free, function-pointer-free subset of the C language, plus some simple extensions.

2. Texture rendering is not supported, although internal copying within GPU memory is fast and can be used.

3. CUDA-enabled GPUs are only available from NVIDIA (GeForce 8 series and above, Quadro and Tesla) but a similar new standard, OpenCL is expected soon which will be a open standard similar to OpenGL and will be supported across all GPU vendors.

2.2 Data Parallel Primitives

As we have seen earlier, the GPU is a data-parallel processor, with thousands of threads processing thousands of data elements. Data parallel primitives play the role of building blocks to many other algorithms on the GPU.

Designing data parallel primitives requires us to partition the data to operate in well-sized blocks. One of the most important resource on current GPUs under CUDA is the shared memory

available to each multiprocessor. A number of data blocks mapped to an MP can extract maximum parallelism and hide the memory latency.

Common parallel primitives include Scan, Reduce, Split, Sort, Gather, Scatter, MapReduce etc. Scan (prefix sum) is a simple and useful parallel primitive for many parallel algorithms like, radix sort, quick sort (segmented scan), string comparison, lexical analysis, stream compaction, run-length encoding, tree operations, histograms etc. Scan was first proposed by Iverson et al. [27] which was further used by Blelloch et al. [5] as a primitive for various parallel algorithms. Horn et al. [25] provided a GPU implementation using the graphics pipeline and was applied to Summed Area Tables by Hensley et al. [24]. Sengupta et al. [42] proposed a CUDA based implementation of the scan operation which is now available as a part of CUDPP library for data parallel primitives [19]. Dotsenko et al. [13] propose a shared memory based approach for performing scan and segmented scan operations on the GPU. Their results show up to 10 times performance over previous implementation by Sengupta et al. [42]. CUDPP [19] library supports a 2-way (1 bit) split operation which can be used to implement radix sort. Multi category split was recently proposed by He et al. [22] which could split an input sequence $n-$ways where n is the number of categories. Dean and Ghemawat [12] proposed MapReduce, a programming model and an associated implementation for processing and generating large data sets. User specified map function processes a key/value pair to generate a set of intermediate key/value pairs, and the reduce function merges all intermediate values associated with the same intermediate key. Recent GPU implementation of the MapReduce framework has been presented by He et al. [20].

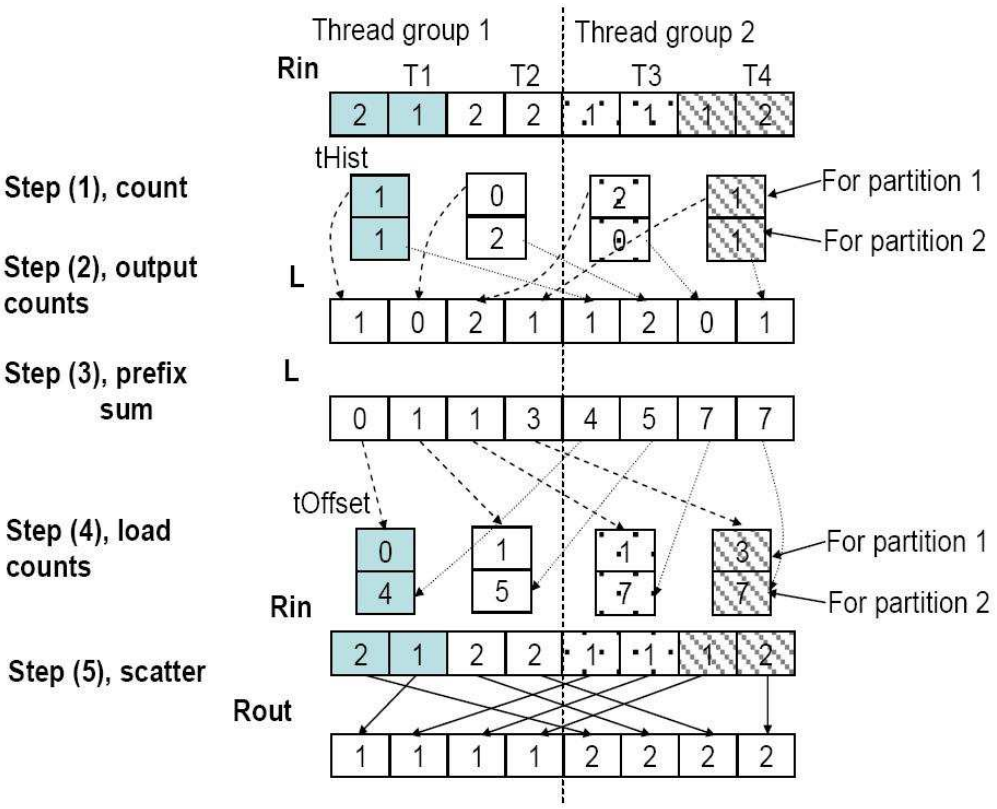


Figure 2.6: Split operation using per-thread histograms in shared memory by He et al. [22]

2.3 Split and Sort

Split has been implemented recently on the GPU by He et al. [22]. They built a per-thread histogram on the GPU to overcome the problem of concurrent writes by multiple threads. They used split as a primitive for implementing a variety of relational join operations for databases on the GPU. Their approach is limited by the available shared memory and limits the number of bins per pass to as low as 64 on current GPUs. Figure 2.6 describes their approach using a small example. The CUDPP library by Harris et al. [19] implements the binary split primitive for two categories. A list of elements tagged with *true* or *false* are split into the two categories. They define and implement *compact* operation which reduces the above kind of list to a smaller list of elements which are tagged true. CUDPP sort is based on this and splits the data 1 bit (2 categories) at a time. Lauterbach et al. [31] use multiple independent split and compact operations, for building a bounding volume hierarchy for ray tracing.

Several other sorting algorithms have been developed for the GPUs. Bitonic sort, a parallel algorithm for sorting was first implemented by Purcell et al. [36]. Govindaraju et al. [15] demonstrated improved sorting performance for external sorting algorithm using graphics processors on large databases. Their implementation of bitonic sort used programmable pixel shaders with OpenGL API. With the introduction of CUDA, sorting algorithms like radix sort and merge sort have been implemented which tend to be faster. Harris et al. [19] propose a bit-wise radix sort approach using CUDA. They partition the data based on a bit starting from least significant bit and moving towards most significant bit. Satish et al. [40] propose a method for bit-wise sorting of data similar to CUDPP. They implement parallel radix sort by increasing the number of bits handled per pass to 4. They also describe a compare based merge sort algorithm. Cederman et al. [9] describe a quick sort based approach for sorting large data on the GPU. Katz et al. [28] implement compare-based sort using bitonic/merge approach.

2.4 Data movement

Govindaraju et al. [16] describe how GPU memory architecture is significantly different from the CPU architecture. GPU cache sizes are smaller in size compared to CPU cache sizes and GPU's video memory has high-bandwidth and high-latency. Scatter operation is an important addition to the programmability of the GPU introduced with CUDA. Efficient methods for scatter and gather operation on the CPU can thus differ from those on the GPU. Govindaraju et al. [21] propose efficient multi-pass scatter and gather operations. They present a probabilistic analysis which accounts for memory access locality to estimate the performance of scatter and gather operations on GPUs. Their approach achieves 2-4 times performance gain with multi-pass scatter.

2.5 Ray Casting and Ray Tracing

The ray casting algorithm for rendering was first presented by Arthur et al. [2]. Ray casting rendered images by tracing a ray from the eye, one per pixel, into the environment. The next important research breakthrough was proposed by Rubin and Whitted [39] in 1979. They extended the idea of ray casting by generating three types of rays: reflection, refraction and shadow, when a ray hits a surface. These techniques have been attempted over years, across architectures like, CPUs, multi-cores, clusters, CellBE, FPGAs etc. Beam Tracing [23] was introduced to exploit the spatial coherence of polygonal environments. Rather than working with high number of rays per image, beam tracing sweeps areas of the scene to form beams. Spatial data structures like kd-trees,

octrees and grids have been used for efficient traversal of large models for ray tracing. Grid data structure was used for one of the first ray tracing implementation on the GPU [35]. The data structure was built once on the CPU and stored as 3-D texture on the GPU for traversal.

Before the introduction of GPU, ray tracing was performed on CPU or on a cluster of CPUs. A single CPU works sequentially on all the rays and finds closest intersections. With the increase in CPU cores and multi threaded architectures, ray tracing could be performed efficiently on a set of processors. MLRTA [38] performs fast ray tracing by allowing a group of rays to start traversing the tree data structure from a node deep inside the tree, saving unnecessary operations. RLOD [53] uses a LOD based scheme which integrates simplified LODs of the model into kd-tree, performing efficient ray-triangle intersections. Wald et al. [48] ray trace deformable objects on the CPU using a bounding volume hierarchy (BVH). They exploit the fact that the topology of the BVH is not changed over time so that only the bounding volumes need be re-fit per frame. In another work, they [50, 49] ray trace animated scenes by rebuilding the grid data structure per frame. They use a new traversal scheme for grid-based acceleration structure that allows for traversing and intersecting packets of coherent ray using an MLRTA-inspired frustum-traversal scheme. Ray tracing has also been performed on non-triangulated models like implicit surfaces [30, 46] and geometry images [18, 8].

Programmable GPUs can perform limited ray tracing due to the constrained programming model [7, 35]. Ray tracing was performed as a multi-pass operation due to insufficient capability of the fragment shaders. With the growth in programmability of the GPU, more efficient methods have emerged which use the looping and conditional operations. Most of the work for ray tracing on GPU uses pre-built data structures, given that the cost of building parallel data structures may be high [26]. A recent work by Zhou et al. [55] builds and ray traces small and medium sized deformable models on the GPU using CUDA. Wie et al. [51] take the alternative approach of non-linear beam tracing on the GPU for deformable objects.

Chapter 3

Atomic and Ordered Atomic Operations

In a parallel computing environment multiple computations are performed simultaneously. It works on the principle that large problems can be divided into smaller concurrent problems. It is believed that parallel programs are more difficult to write than sequential ones, as the concurrent behavior introduces a variety of software complications such as the race condition. Communication and synchronization between multiple small parallel problems are typically one of the greatest obstacles to optimal performance. Concurrency calls for atomic operations to be performed on the memory in order to avoid any kind of read/write hazards (RAW/WAR/WAW). Atomic operations make sure that the concurrent writes from multiple requests are performed in a consistent manner and no corruption of data occurs.

3.1 Atomic Operation

An *atomic operation* is a set of actions that can be combined so that they appear to the rest of the system to be a single operation that succeeds or fails. In concurrent processing, where multiple processes can access a shared memory or register, an atomic operation on the shared location can be implemented by serializing it in some order such that the final results are correct. That is, for an operation $\mathcal{O}$ performed concurrently on a location M by a set S of processes, the resultant value in M is:

$$M \leftarrow \mathcal{O}_t(\mathcal{O}_s(\cdots \mathcal{O}_r(\mathcal{O}_q(\mathcal{O}_p(M))))), \tag{3.1}$$

where $\{p, q, r, \cdots, s, t\}$ is a permutation $\mathcal{P}(S)$ of the processes in S. All permutations will give correct results for associative operations. The atomic operation can be thought of as a serialization of the contending operations in some order, with the final value being the result of all operations applied in that order.

Atomic operations of the *test-and-set* class are needed to build coherent data structures by distributed systems. These operations return to each process the value of the location M immediately prior to its own operation, in one indivisible step. Thus, a process p gets a value

$$m_p \leftarrow \mathcal{O}_{i-1}(\mathcal{O}_{i-2}(\mathcal{O}_{i-3}(\cdots))), \tag{3.2}$$

where i is the order of the process p in the permutation $\mathcal{P}(\mathcal{S})$. Each process gets a potentially different return value m_p based on its order in the permutation.

Definition: An *atomic* invocation of a concurrent operation $\mathcal{O}$ on a shared location M is equivalent to its serialization in an unspecified order within the set S of processes that contend for M. The sequence of steps can be described equivalently as follows.

1. Compute a permutation $\mathcal{P}$ of S as $p_1, p_2, p_3, \cdots, p_{|S|}$. The exact permutation used is unspecified and is usually implementation-dependent.

2. Each process p_i gets the following partial result m_i as its return value

$$m_i \leftarrow \mathcal{O}_{i-1}(\mathcal{O}_{i-2}(\cdots \mathcal{O}_1(M))), \tag{3.3}$$

   where $\mathcal{O}_j$ is the operation of the j'th process in $\mathcal{P}$. Thus, the operation of the first process of the permutation is applied first.

3. The final result in M is given by

$$M \leftarrow \mathcal{O}_{|S|}(\mathcal{O}_{|S|-1}(\cdots(\mathcal{O}_2(\mathcal{O}_1(M))))) \tag{3.4}$$

Example 1: Assume N processes hold a bit b and a data element *data*. Assume n processes have $b = 1$, where $n < N$ is not known ahead of time. The n data elements are to be packed into a shared array of length n.
Solution: The processes with $b = 1$ should write its element to the next free location of the array. This can be performed using a shared variable *count* as:

1. *count* = 0
2. Each processor with $b = 1$ does in parallel
   a) index = *count*++
   b) array[index] = *data*

The "++" operator in Step 2(a) has the semantics of post-increment in C. Thus, `index` has the lower value and the step results in the value being read and incremented in one indivisible step. This is an atomic increment operation of the kind defined earlier. Each process is guaranteed to get a different number starting from 0 to $n - 1$ and will store its data element in a non-conflicting location in Step 2(b).

3.2 Atomic Operations on CUDA

GPUs with compute capability 1.1 (8600 GTS, 8800GT etc.) support global memory atomic operations on the GPU. Current generation GPUs are compute capability 2.0 and higher and can perform atomic operations on the global as well as shared memory. In the following section we explore various ways to use atomic operations on CUDA by considering the histogram computation problem as an exercise.

3.2.1 Global Memory Atomic Operation

We consider a single copy of histogram to be built placed in the global memory of the GPU. Using the atomic operations (increment) provided by CUDA on the global memory we compute the histogram using the following pseudocode.

```
for each thread in parallel
  for each element x assigned to the
      thread, sequentially
  {
    // Get category for element x
    bin = category(x);

    // Increment its count
    atomicInc(globalHist[bin]);
  }
```

3.2.2 Shared Memory Atomic Operation

Atomic operations can be performed on the shared memory also. We describe histogram computation using several forms of atomic operations in this section. These can be used with devices which can not support shared memory atomic operations.

Clash Serial Atomics

One way to simulate parallel atomic writes to the shared memory is by serializing the clashes. This can be done by embedding the thread ID within the warp of threads that are scheduled together with the data [43]. The current Nvidia GPUs schedule 32 threads simultaneously in a warp. 5 bits are needed to identify threads of a warp. Each thread appends its 5-bit tag to the data word before writing to the shared memory (Figure 3.1). A thread knows its write succeeded if its tag is found on a read-back that is done immediately. The capacity of counts is reduced from 2^{32} to 2^{27}. This is not a limitation for split as each block will handle far fewer elements. Each write in warp cycle results in one successful write, serializing the clashes of a warp.

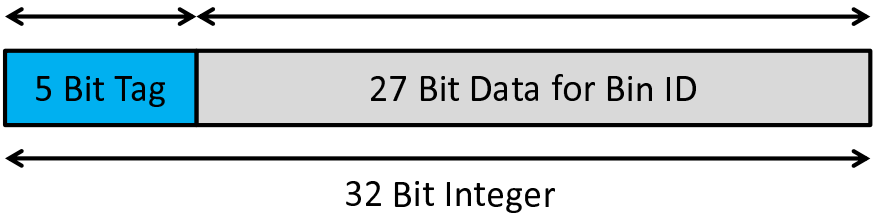


Figure 3.1: Thread ID is used as a tag for clash-serial atomics

```
for each thread in parallel
  for each element x assigned to the
      thread, sequentially
  {
    bin = category(x);

    // Unique Tag for each thread
    threadTag = threadId << (32-WARP_LOG);

    // Write to shared memory until the
```

```
    // read back gives your tag
    do {
      count = sharedMem[bin] & 0x7FFFFFF;
      count = threadTag | (count + 1);
      sharedMem[bin] = count;
    } while (sharedMem[bin] != count);
}
```

The costs of atomic operations is directly proportional to the number of clashes within the warp. In practice, this approach works only when 32 threads are in a block on current GPUs due to the way warps of a block are scheduled. For multiple warps of a block we can use separate histograms for error-free atomic operations.

Thread Serial Atomics

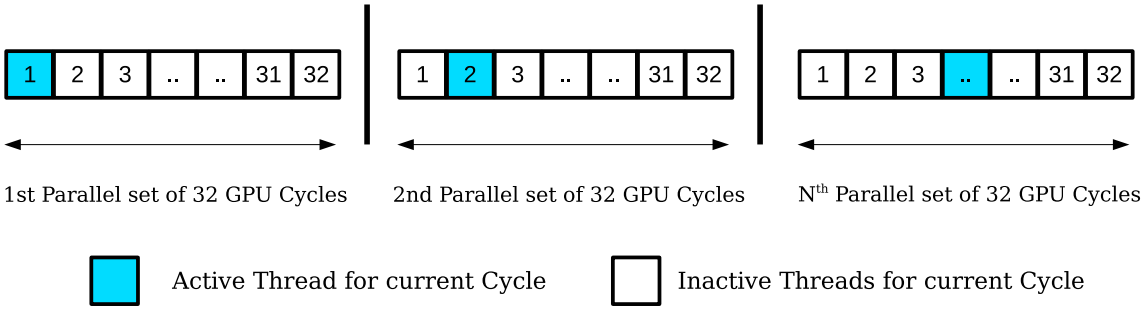


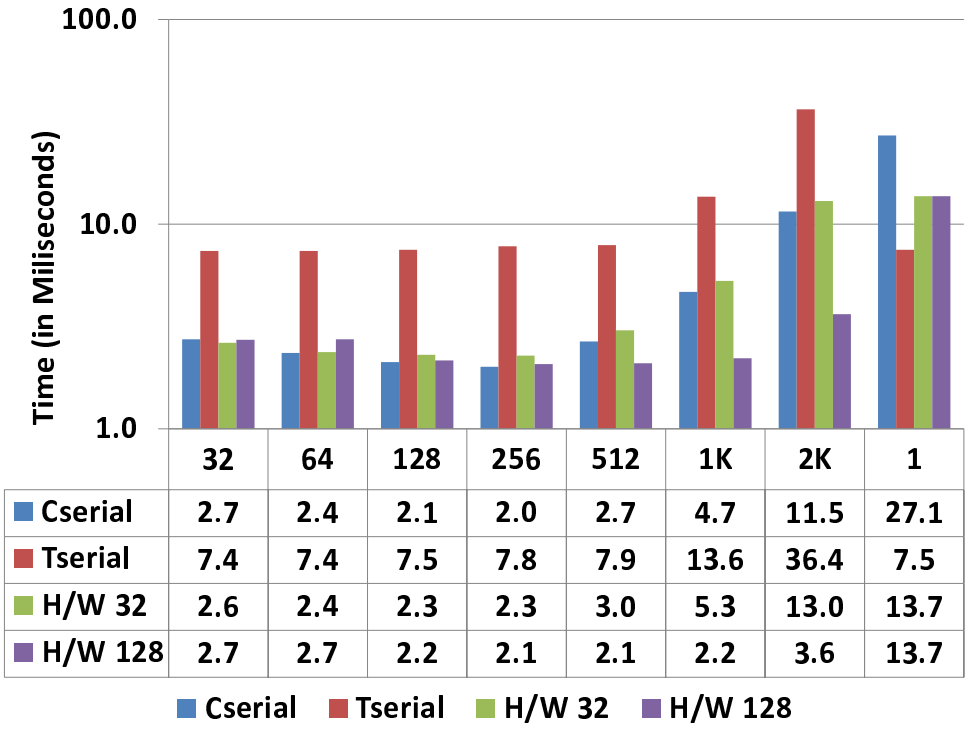
Figure 3.2: Threads are serialized to achieve atomicity with each iteration producing one write to shared memory.

Another approach to perform atomic operations is to serialize the threads of a warp irrespective of clashes. Each thread will wait for its turn to write. This will incur a fixed overhead proportional to the warp size, but does not perform extra shared memory writes.

```
for each thread in parallel
  for each element x assigned to the
      thread, sequentially
  {
    bin = category(x);

    for i = 0 to WARPSIZE-1 sequentially
      if (threadIdx.x == i)
         sharedMem[bin]++;
  }
```

Figure 3.2 shows the state of GPU cycles for this approach for a warp of threads. For each of the 32 cycles, only one thread performs the write on the shared memory thus ensuring the atomicity. In practice, this technique also works only when the block has only one warp. This approach is slow, but can control the order in which clashing threads are serialized. This might be beneficial for some applications, as we will see later.



	32	64	128	256	512	1K	2K	1
Cserial	2.7	2.4	2.1	2.0	2.7	4.7	11.5	27.1
Tserial	7.4	7.4	7.5	7.8	7.9	13.6	36.4	7.5
H/W 32	2.6	2.4	2.3	2.3	3.0	5.3	13.0	13.7
H/W 128	2.7	2.7	2.2	2.1	2.1	2.2	3.6	13.7

Figure 3.3: Timing comparison of histogram using different implementations of shared memory atomic on 16M elements. X-axis gives different number of bins used for histogram operation.

Hardware-Supported Atomics

The latest GPUs support shared memory atomic operations, similar to global memory operations. They work across warps and hence data structures can be shared across blocks of different sizes. This approach is fast and can handle any valid configuration of threads and blocks. Pseudocode for histogram computation using hardware atomics is given below.

```
for each thread in parallel
  for each element x assigned to the
      thread, sequentially
  {
    bin = category(x)
    atomicInc(&sharedHist[bin]);
  }
```

We explore the relative performance of the three methods across a range of bins for $16M$ input data for histogram computation only and for the full split operations. Figure 3.3 gives the performance on the histogram operation. Hardware atomic is tried with 32 threads per block and with 128 threads per block. Hardware atomic operations with 128 threads yields the best performance on hardware that supports it, such as the GTX280. The approach works with higher number of threads too (256, 512), but optimum performance is achieved with 128 threads. On GPUs with no hardware atomic support (such as the G80 series) the Clash Serial method give the best performance. The Thread Serial method gives the worst performance due to its inherent serialization. In these experiments, the input array is divided into chunks, each of which is handled by a block.

The threads of the block collectively process the elements in the chunk. The thread-serial atomic operation has a running time that is independent of the data, however. The last column of table in Figure 3.3 correspond to timings when number of bins is 1. Thread Serial method performs better than both other approaches. Results confirm that Thread Serial approach has a constant overhead independent of the number of bins. The timings when number of bins are 1024 and 2048 is affected due to the overall occupancy of the multiprocessor.

3.3 Ordered Atomic Operations

Definition: An *ordered atomic* invocation of a concurrent operation $\mathcal{O}$ on a shared location M is equivalent to its serialization within the set S of processes that contend for M in the order of a given priority value π. The sequence of steps can be described equivalently as follows.

1. Compute a permutation $\mathcal{P}$ of S as $p_1, p_2, p_3, \cdots, p_{|S|}$ such that
$$\pi(p_1) \leq \pi(p_2) \leq \cdots \leq \pi(p_{|S|}). \tag{3.5}$$
2. Each process p_i gets the following partial result m_i as its return value
$$m_i \leftarrow \mathcal{O}_{i-1}(\mathcal{O}_{i-2}(\cdots \mathcal{O}_1(M))), \tag{3.6}$$
where $\mathcal{O}_j$ is the operation of the j'th in $\mathcal{P}$. The operation of the process with the least priority is applied first.
3. The final result in M is given by
$$M \leftarrow \mathcal{O}_{|S|}(\mathcal{O}_{|S|-1}(\cdots(\mathcal{O}_2(\mathcal{O}_1(M))))) \tag{3.7}$$

Example 2: Assume each process has a unique ID ranging from 1 to N. The operation to be performed is the same as in Example 1, but the data elements need to be stored in the same order as the process IDs. That is, the data from a process i should appear earlier than the data from all processes j if $i < j$.

The atomic operation discussed earlier will not produce the required results if an arbitrary (implementation-dependent) permutation is applied to serialize the computations of processes contending for the same location in Step 2(a). The solution then is to write the b bits to a shared array in order of the process IDs and to perform a parallel prefix sum of the bits. The result in location i gives the index of the data element of process i, using which the data can be written.
Solution: The problem can be solved using the same solution as Example 1, but using an ordered atomic `increment` with the process ID as the priority value in Step 2.

1. *count* = 0
2. Each processor i with $b = 1$ does in parallel
   a) index = `atomicIncrement`(*count*, i)
   b) array[index] = *data*

The operation `increment` returns the first argument value and post-increments it, using the second argument as the priority. This ensures that processes with lower IDs get lower values of `index` than those with higher IDs, resulting in the desired ordering of the data elements in the shared `array`. Ordered atomic operations can be specified by the tuple $<\mathcal{O}, \pi>$ such that the conflicting processes serialized in the order of their π values.

Figure 3.4 describes a case of parallel radix sort by splitting the input one digit per pass starting from least significant digit. Pass 2 needs to perform ordered atomic operations to maintain the order from Pass 1.

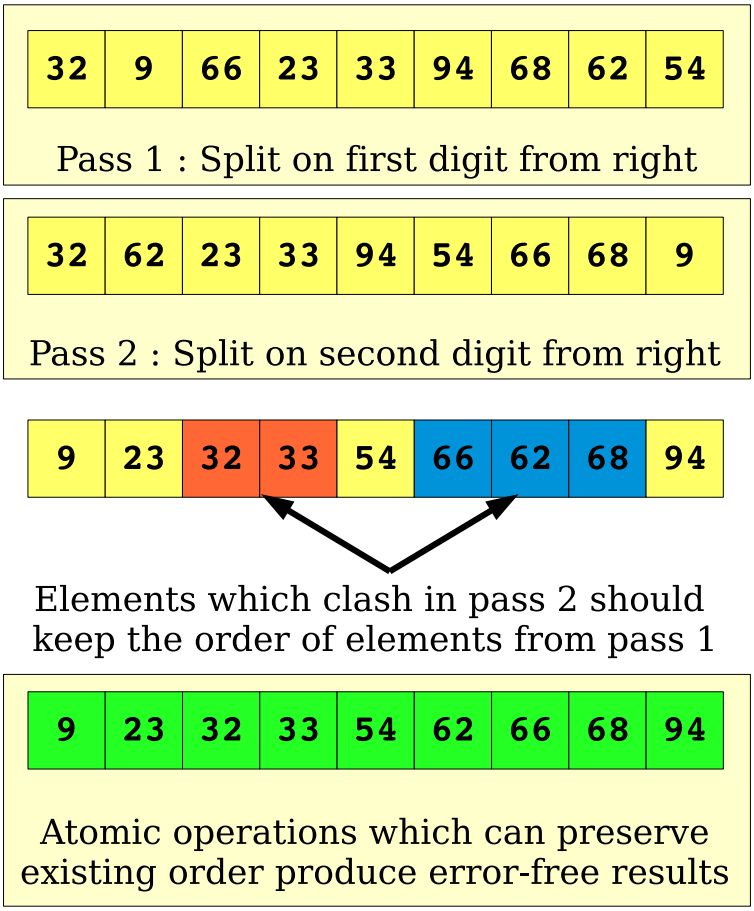


Figure 3.4: An order preserving atomic operation can maintain the existing order for the Iterative Split approach.

3.4 Ordered Atomic Operations on CUDA

Global memory atomic operations on CUDA perform an implementation-dependent serialization and is not guaranteed to preserve the ordering. We described three ways to implement shared memory atomic operation in previous section. Among them, neither the hardware-supported scheme nor the clash-serial scheme guarantees the ordering as both rely on resolving the clashes in some order. Our experiments on the GPU hardware verified this fact; the ordering is not maintained and split done using them produce wrong results.

The thread serial atomic scheme, however, explicitly controls the order in which the operation is taken up by serializing it explicitly. The code fragment in Section 3.2.2 uses equality of the loop variable with the the thread ID as the condition for the write within a warp. This has the impact of using the thread ID as the priority value for the ordered atomic operation. It is easy to see that other priority values can also be used as long similarly. Our experiments on the GPU confirms this fact and the ordering can be preserved for clashes within the same warp of threads. The current GPU hardware does not guarantee ordering of warp scheduling for the sake of greater ability to hide memory latencies. Thus, ordered atomic operations are not guaranteed to work across multiple warps by serializing. We use the thread-serial atomic scheme for correct Iterative Splitting. The

serializing is, however, expensive. It should be noted that the thread serial scheme is 5 to 10 times slower than the hardware-supported scheme (Figure 3.3). The ordered atomic operations can be implemented 5-10 times faster if the serialization of the hardware-supported atomics can be controlled.

Chapter 4

Split Operation

A *split* primitive divides a relation into a number of disjoint partitions according to a given partitioning function as described in Chapter 1. The resulting partitions are stored in the output relation. Splits are used in hash partitioning or range partitioning. For the case of key-value pairs as input, split operation rearranges the data where all the pairs with same keys are contiguous in the output list.

Several steps of Algorithm 1 (Chapter 1) can have clashes when performed in parallel. We consider the data parallel computation model in which a thread – usually mapped to a computing core – handles a small block of list elements with a large number of threads operating in parallel. The counting steps (Line 4, 17) can have clashes as multiple processes or threads may increment the same count. Atomic increments that guarantee correct results under collisions provide one way to overcome this. Atomic operations require hardware support and may be expensive. In the absence of atomic operations, each processor can handle a subset of the list and compute the sub-counts for that part of the data. The sub-counts can then be combined in a separate step to get the global counts. Step II (Line 7-11) maps to a parallel prefix sum or scan operation for which efficient implementations are available [4, 42].

4.1 Parallel Split Operation

The GPU threads have access to two types of memory. The *device* or *global memory* is shared by all processors but has an access latency of 500 clock cycles. The *shared memory* is local to a block of threads and can be accessed in a single cycle. It is, however, limited in size and are not accessible to threads outside of the block. Atomic operations are available on both types of memories on the latest family of GPUs. Earlier GPUs provided atomic operations on the global memory alone or none at all.

We first describe a straightforward approach to split that uses global memory atomic operations. We next describe the approach by He et al. [22] that uses no atomic operations.

4.1.1 Using Global Memory Atomic Operations

The **for** loops of Steps I and III of Algorithm 1 can be performed in parallel. When the number of input elements is very large, each thread can handle a block of elements with multiple threads operating in parallel. A simple approach is to use atomic increment operations at lines 4 and 17. Step I of the algorithm can be written using the global memory atomic operations described in Chapter 3.

The algorithm requires $O(N)$ global memory for the list and $O(M)$ for the sub-counts used in Step II. This method has two drawbacks. First, the global memory access is slow. Second, the performance of atomic operation suffers in the presence of collisions. The cost is too high for a skewed or badly arranged data as multiple threads increment the same memory. Also, the fast shared memory available on each multiprocessor is unused. This approach is slow for small number of bins due to the collisions and the performance improves for larger numbers of bins.

4.1.2 Non-Atomic Split by He et al.

He et al. [22] compute the bin counts without atomic operations on the GPU. In their approach, each thread handles a disjoint block of the input and builds the complete histogram for its block in the shared memory. No collisions occur as the block of elements is processed sequentially by the thread. The partial histograms are written to globally accessible memory to separate locations in a column-major order. A scan over this data gives the required starting points for each bin of each thread. This is used to send each element to its final location. Step I of the algorithm can be written as:

```
for each thread in parallel
  for each element x assigned to the
      thread, sequentially
  {
    // Get category for element x
    bin = category(x);

    // Increment the local histogram
    // for current thread
    localThreadHistogram[bin]++;
  }
```

The limited shared memory on each multiprocessor restricts the number of categories that can be handled at a time. With 16KB of shared memory is available, only 64 bins can be used at a time using 32 threads per block. The approach requires $O(BTM)$ global memory to write the per-thread histograms, where B is the number of blocks used, T the number of threads per block, and M the number of categories. Higher number of bins can be handled by running multiple passes with 64 bins per pass.

4.2 Split Using Atomic Shared Memory Operations

We now present our approach to split. The key idea of our approach is the use of a single copy of the histogram per block of threads. The computations on CUDA are divided into blocks of threads that have read/write access to the limited shared memory. The histogram can thus be built in the shared memory and is updated by all threads. Atomic increment on the shared memory is necessary to handle clashes. The latest generation of GPUs provide hardware support for it. We also described (Chapter 3) two ways of simulating shared memory atomic operations when hardware support is not available.

Algorithm 2 outlines our split algorithm that builds per-block sub-histograms on the shared memory. The above algorithm uses $O(BM)$ global memory as histograms are built per block. Figure 4.1 shows the 3-step algorithm using shared memory for split operation on the GPU. Steps

Algorithm 2 SharedAtomicSplit

1: Compute the histogram for the bins per block
2: Store it bin-wise in global memory in an $M \times B$ sized array
3: Scan the histogram array, giving index of each bin for each block
4: Load part of scan array corresponding to block into shared memory
5: Read x and category(x)
6: Read the scan histogram value for the bin and increment it atomically
7: Write x to value read from the shared memory

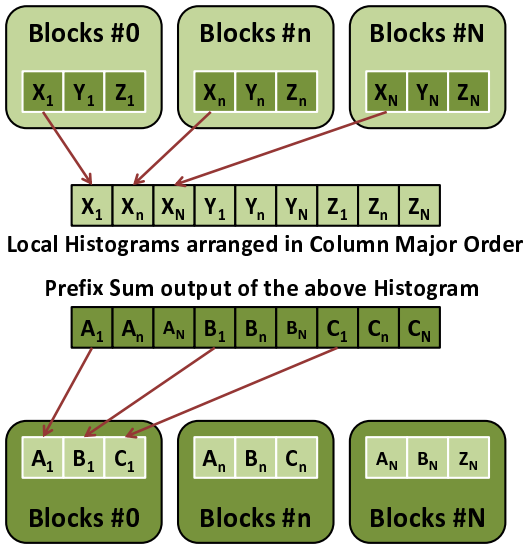


Figure 4.1: The shared atomic split computes one sub-histogram per block which are arranged in order in the global memory. A scan on it gives the starting point for each bin in each block and is used in step III for data insertion.

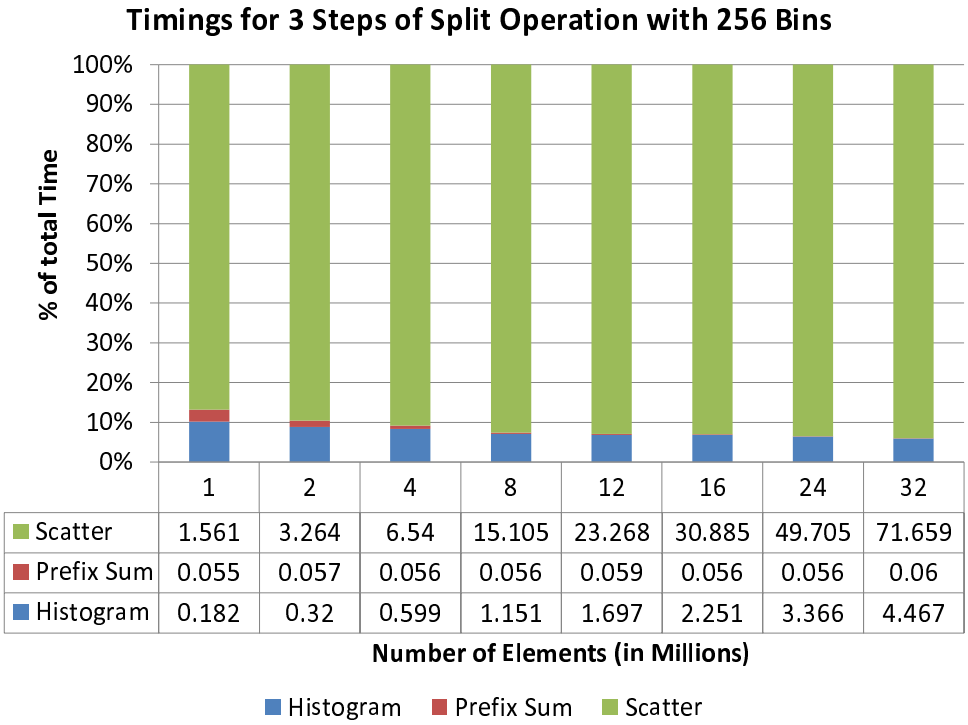
1 and 6 of Algorithm 2 require atomic increment operations for correct results. We compare the results using different histogram computing methods presented in Chapter 3

4.2.1 Comparison of the Methods

No. of categories	Global atomic			Non-Atomic[1]			Shared Memory atomic			CPU		
	1M	4M	16M	1M	4M	16M	1M	4M	16M	1M	4M	16M
32	80	321	1285	2.1	11	117	1.11	5.59	22.2	25	98	396
64	75	301	1206	3.4	14	119	1.40	6.43	25.6	24	98	408
128	54	216	864	12	39	149	1.53	6.70	27.6	24	100	413
256	37	148	591	22	72	277	1.59	7.09	28.9	24	101	419
512	21	86	346	43	140	536	1.64	7.40	30.5	27	115	480
1024	16	65	258	84	277	1055	1.69	7.45	31.1	26	109	464
2048	12	50	207	163	542	2060	1.87	7.53	31.5	27	114	478

Table 4.1: Timing comparison of global memory atomic, non-atomic, and hardware-supported shared memory atomic splits on an Nvidia GTX280 and sequential split on an Intel quad-core CPU. Times are given in milliseconds for lists of 1, 4, and 16 million elements.

Only 16KB of shared memory is available on current GPUs. This limits the number of bins to 2048 for Algorithm 2, as 2K bins will use 8KB of the shared memory to store the histogram. (We assume the number of bins is a power of 2.) Table 4.1 compares the times for different split operations with a sequential CPU implementation, an implementation that uses global memory atomic operations (Section 4.1.1), and the method by He et al. [22] for up to 2K bins. Global memory method performs well for high number of bins due to lower probability of clashes and performs poorly as clashes increase. It maintains a single histogram in the global memory and performs $O(M)$ operations for step II of Algorithm 1. Running time using He et al. [22] grows nearly linearly with the bins. It also overuses the shared memory and needs $O(BTM)$ global memory space. The CPU implementation performs $O(N + M)$ operations and grows linearly with the number of bins.



	1	2	4	8	12	16	24	32
Scatter	1.561	3.264	6.54	15.105	23.268	30.885	49.705	71.659
Prefix Sum	0.055	0.057	0.056	0.056	0.059	0.056	0.056	0.06
Histogram	0.182	0.32	0.599	1.151	1.697	2.251	3.366	4.467

Figure 4.2: Distribution of total time for split using hardware atomic operations. X-axis shows millions of records. Step III of Algorithm 1 consumes 90% or more of the total time due to the scattering to the global memory.

Figure 4.2 shows the times for each step of the split operation from Algorithm 1 using hardware atomic operations on the shared memory. Step I and Step II take less than 10% of the total time. Step III is similar to step I with the additional overhead of writing the output to global memory location.

The random (non-coalesced) writes to the global memory in step III of Algorithm 1 consumes around 90% of the total time of the split operation.

[1] An approach similar to He et al. [22] is used and extended for bins higher than 64.

4.3 Multi Level Split

The limited shared memory available to each block restricts the number of bins for shared memory split. We propose a *multi-level split* scheme for larger numbers of bins. In this scheme, the bins are considered to be arranged at multiple levels. Each bin at a top level is made up of multiple bins at the next lower level, with the original bins at the lowest level. One way to create levels is to divide the binary representation of the bin ID into groups of bits, as shown in Figure 4.3. Hierarchical Split uses leftmost k_1 bits as the sub-bin ID for the first level of splitting. The next level splits the resulting sublists using the next k_2 bits as the next sub-bin ID, etc. This is similar to the MSD radix sort that sorts based on more significant digits first. Iterative Split uses rightmost k_1 bits as the bin ID in the first step, the next k_2 bits in the second step, etc. This is similar to the LSD radix sort that sorts based on less significant digits first. These algorithms are described next.

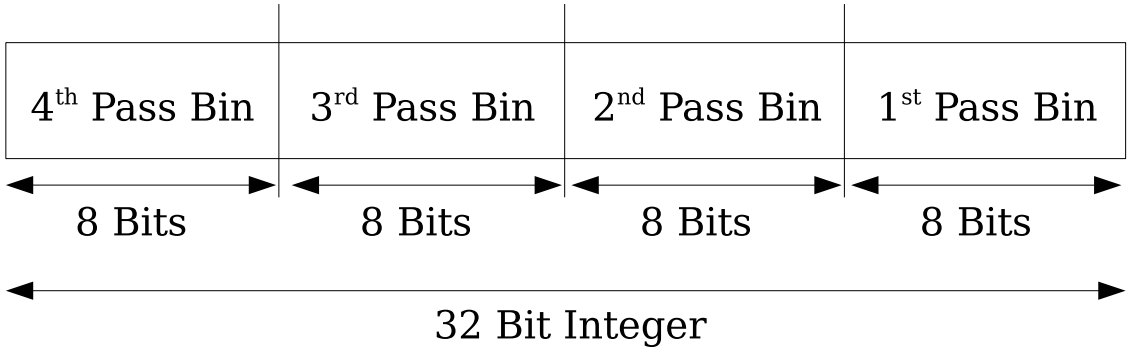


Figure 4.3: Bits are grouped together and treated as sub-bin IDs at each level of multi-level split. Iterative Split approach considers bins starting from right moving towards left.

4.3.1 Hierarchical Split

The Multi-Level Hierarchical Split algorithm divides the bins hierarchically into sub-bins using the bits of the bin ID from the most significant bits to the least. The list is split into M_1 bins using the left most m_1 bits ($M_i = 2^{m_i}$ for all i). In the next step, each sub-list is split independently into M_2 bins using the next m_2 bits. This continues till all $\log_2 M$ bits of the bin ID representation are used up. For example, for 64K bins, the bin ID needs 16 bits. The input list is split to 256 bins using the most significant byte of the bin ID as the category in the first level. In the second level, each of the 256 sub-lists of the first level split are split to 256 bins independently using the least significant byte as the bin ID. This can be repeated if more bits of the bin ID, if present. This approach builds a tree structure (Figure 4.4), which can be used by operations that need them. The number of independent sets to be processed at a level is equal to the number of nodes in the tree (Figure 4.4) at that level. With increasing number of levels the number of disjoint splits required grows rapidly, though the total number of data elements remains the same.

We implement the first level of split as described earlier. The subsequent levels are implemented by allocating a sub-list to a CUDA block. Each block splits its input data to the required number of bins at that level of the tree. The number of blocks increases exponentially with each level.

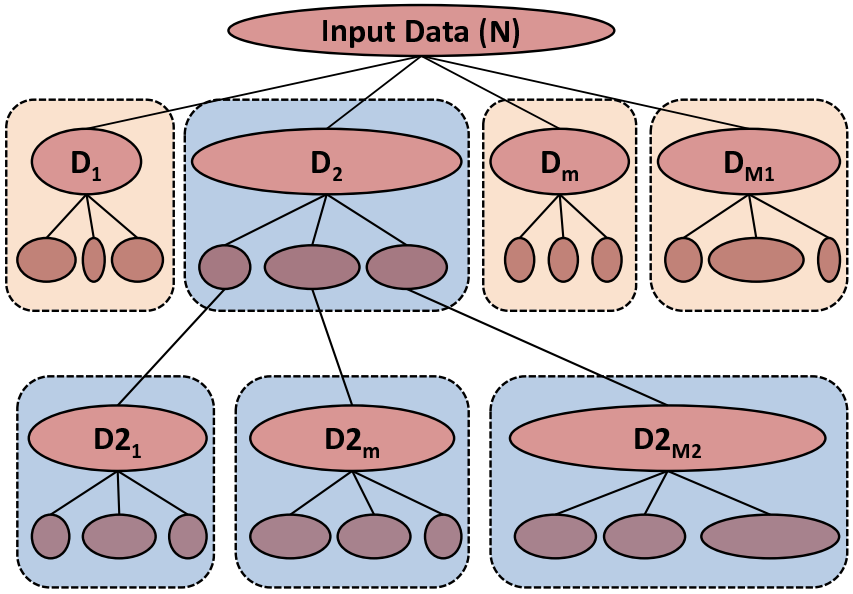


Figure 4.4: The number of independent splits increases at each level of Hierarchical Split algorithm. D_k holds data belonging to bin k in a level and is split independently at the next level.

No. of bins	Global		Hierarchical		CPU	
	1M	16M	1M	16M	1M	16M
4K	10	166	3.01	59.7	28	494
8K	8	139	3.03	59.6	29	508
16K	7	154	3.00	58.6	30	525
32K	7	154	3.04	55.9	33	611
64K	8	159	3.08	54.1	43	1064
128K	9	175	3.17	54.2	60	1577
256K	10	193	3.44	55.1	99	2317
512K	11	212	4.26	55.5	174	2849
1M	13	228	6.3†	58.2	198	3066
2M	14	282	8.5†	72.7	265	3821

Table 4.2: Comparison of the time in milliseconds for global memory atomic and Multi-Level Hierarchical Split on an Nvidia GTX280 with a CPU split. Entries marked † use 3 levels of split and others use 2 levels.

Allocating multiple sub-lists to a CUDA block is less efficient due to the resulting conditional executions. The use of a large number of blocks can degrade the overall performance. This approach, however, is well suited when variable numbers of bins are used dynamically at different levels.

Table 4.2 presents results for the Hierarchical Split operation for large number of bins, needing 2 and 3 levels. Our algorithm outperforms the CPU split and the global memory atomic split. Table 4.3 shows the performance of 1, 2, and 3 level splits for a range of bins. For a single pass of split, a maximum of 2048 bins can be used due to the limited shared memory capacity. For higher number of bins we show how 2 level and further 3 level splits can be used. The performance of the split

depends on the size of input list, the configuration of CUDA blocks, and the bins and sub-bins are organized. Three levels perform better when the number of bins is over one million when the input list has 1M elements. (Partitioning a million elements into a million bins may seem excessive, but can arise.) For a list of 16M elements, three level split catches up only for larger number of bins. Thus, an efficient configuration can be chosen for high number of bins depending on the data size and sub-bins.

No. of bins	Multi-Level Hierarchical Split					
	1M			16M		
	1-L	2-L	3-L	1-L	2-L	3-L
32	1.11	-	-	22	-	-
64	1.40	-	-	25	-	-
128	1.53	-	-	27	-	-
256	1.59	-	-	28	-	-
512	1.64	-	-	30	-	-
1024	1.69	3.37	-	31	57	-
2048	1.87	3.08	-	31	58	-
4096	$*$	3.11	-	$*$	60	-
8192	$*$	3.03	-	$*$	59	-
16K	$*$	3.01	-	$*$	59	-
32K	$*$	3.01	4.8	$*$	59	82
64K	$*$	3.05	4.7	$*$	54	82
128K	$*$	3.20	4.7	$*$	54	81
256K	$*$	3.5	4.9	$*$	55	80
512K	$*$	4.26	5.9	$*$	55	81
1M	$*$	7.99	6.3	$*$	58	82
2M	$*$	24.3	8.5	$*$	72	84

Table 4.3: Times in milliseconds for the Hierarchical Split for different input list lengths, number of categories, and numbers of levels used on an Nvidia GTX280. Configurations of no interest are denoted by a '-' and infeasible configurations are denoted by '$*$'.

4.3.2 Iterative Split

The hierarchical split approach needs to perform a large number of independent splits in later levels. This brings down the overall performance on the GPU when the number of bins exceeds 2 million. We can form sub-bins by grouping bits of the bin ID from the right to the left. For example, the input list can be split to 256 bins using the least significant byte as the category in the first level (Figure 4.3). The entire list can subsequently be split to another 256 bins using the next byte as the category in the second level, etc. This does not induce a hierarchy in the bins. Subsequent passes are, thus, identical to the first pass. The *Multi-Level Iterative Split* algorithm repeats the basic split of the entire list into K sub-lists using $\log_2 K$ bits of the bin ID as the category, starting with the least significant bits. Figure 4.5 shows how this is done. At each level, complete input data is split to K number of bins. The results, however, will be correct only if the items maintain the order from earlier levels, which requires special handling as explained later.

The Iterative Split approach splits the entire data into K bins in each iteration. It can, therefore, choose optimal configuration parameters for efficiency. It scales linearly to arbitrarily high number of levels as the work done in each is identical. The algorithm requires that two elements that fall

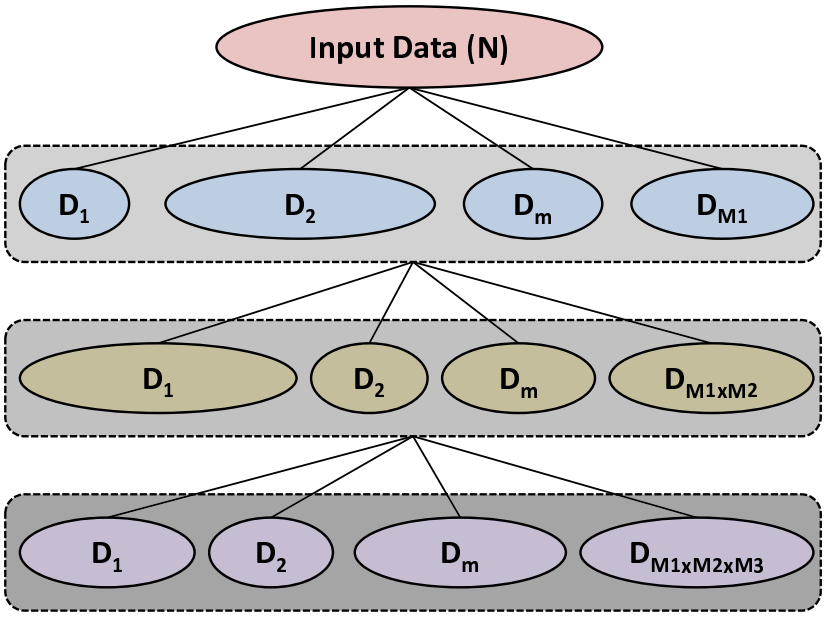


Figure 4.5: The Iterative Split algorithm performs a single split of the whole list to a number of bins at each level. For correctness, the ordering of elements from previous levels need to be maintained.

into the same bin at a particular level maintain their relative order from the previous level. This is ordinarily possible using the direct parallelization of Algorithm 1 as the elements of the list can be allotted to each processor from the left to the right and collected in that order also. The global radix sort in CUDPP [19] sorts numbers by repeatedly assigning them to either the bucket for a 0 in the current bit position or a bucket for 1 using the compact operation, and moving the current bit position from the least to the most significant bit. The 2-way compact performed using global memory scan guarantees proper ordering. The Iterative Split algorithm, however, uses the shared memory for better performance. This makes it challenging to maintain the order from previous levels.

The Iterative Split operation requires to maintain the order from previous splits. Figure 4.6 shows how two elements can belong to the same bin for this iteration of the split and should be inserted in their current order. This is to maintain the split correctness from the previous passes (previous set of bits). Regular hardware atomic operations do no promise to maintain any order in case of a clash. They only guarantee atomicity in no specific order. We described a new class of ordered atomic operation in chapter 3 and its implementation on CUDA that help us achieve this. Figure 4.8 gives results for split operation for bins ranging from 16 to 1024. We see that the performance of split gets better with initial increase in number of bins due to a decrease in number of conflicts. However, for large number of bins (1024) the occupancy on the GPU is hampered and parallelism is not fully utilized. Based on our results we choose 256 bins (8 bits) as the size for basic split operation for Iterative Split method. Thus, we perform multiple passes of 8 bit stable splits in order to achieve split over higher number of bins.

4.3.3 Comparison of Iterative and Hierarchical Split Methods

We compare the two approaches for a large range of bins and input list length in Table 4.4. The ordered atomic operation is simulated by serializing the thread and is 8-10 times slower than

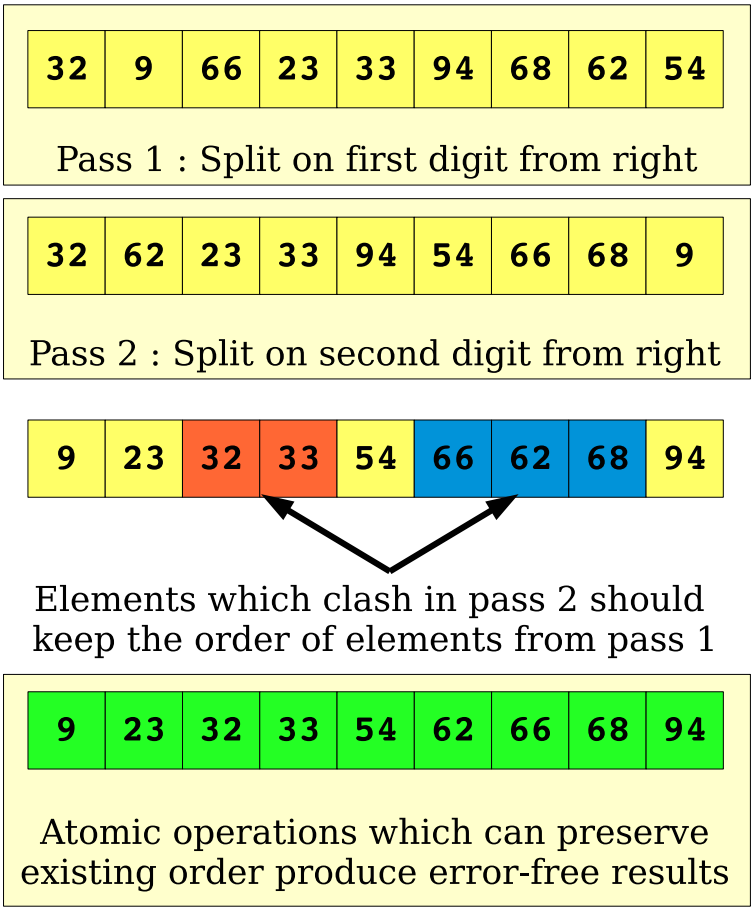


Figure 4.6: An order preserving atomic operation can maintain the existing order for the Iterative Split approach.

hardware atomics as seen in Section 3.2.2. Thus, the Iterative Split method performs only a marginally better or marginally worse than the Hierarchical Split method until the number of bins is 2 million, as seen in Table 4.4. Please note that first level of Iterative Split does not need to use ordered atomics. This is the reason for it being marginally faster for 4K and 8K bins. The partitioning of the bins into sub-bins can be done in multiple ways for large number of bins. For example, 128K bins can be partitioned as 512×256 (that is, split into 512 bins using the leftmost 9 bits in level 1, followed by a split of each of those bins into 256 bins using the rest of the bits) or as 2048×64. Since the second level of hierarchical split performs independent splits using separate CUDA blocks, it is not able to exploit sufficient parallelism to hide memory operations when the first level uses fewer bins. Hence, the 512×32 configuration is faster than the 32×512 one for 16K bins.

We split using 3 levels for both approaches when the number of bins is greater than 2M. Hierarchical split approach prefers larger number of bins among the configurations for the first pass, as the next passes can use more blocks to exploit parallelism. Hierarchical split approach outperforms the Iterative Split method when the number of bins is in the range of 32K to 1M. This is due to the optimal use of the GPU resources or CUDA blocks in the second level of splitting. The exponential increase in the number of independent splits to be performed in later levels degrades

Configuration of bins	Hierarchical						Iterative					
	1M		4M		16M		1M		4M		16M	
	#B	T_{ms}	#B	T_{ms}	#B	T_{ms}	#B	T_{ms}	#B	T_{ms}	#B	T_{ms}
Various combinations for #Bins = 4K												
128×32	30	3.09	30	13.92	960	59.7	120	3.27	960	13.6	1920	54.3
64×64	30	3.01	60	14.72	960	61.4	480	3.00	960	13.68	1920	54.4
32×128	240	3.4	480	15.10	960	61.7	480	2.91	960	13.06	1920	51.3
Various combinations for #Bins = 8K												
256×32	30	3.03	30	12.88	960	59.63	120	3.04	480	14.19	1920	50.2
128×64	30	3.07	30	13.67	960	62.62	240	3.11	960	13.6	1920	56.9
32×256	240	3.4	480	15.09	480	63.73	120	2.82	480	13.0	1920	52.9
Various combinations for #Bins = 16K												
512×32	30	3.01	60	13.08	480	58.63	120	3.08	480	14.52	960	57.1
128×128	30	3.00	240	14.56	30	70.35	240	3.23	960	14.1	1920	56.5
32×512	240	3.47	30	15.23	480	87.01	120	3.18	240	13.93	1920	54.9
Various combinations for #Bins = 32K												
1K×32	30	3.04	30	12.88	240	55.97	120	3.14	480	14.54	480	58.8
256×128	30	3.08	30	13.69	960	62.51	120	3.30	480	14.43	1920	57.5
32×1K	480	4.22	240	16.62	240	91.96	120	4.27	240	17.0	480	54.8
Various combinations for #Bins = 64K												
2K×32	30	3.19	30	12.74	240	55.10	120	3.33	120	13.67	120	58.9
256×256	30	3.08	30	13.51	960	63.12	120	3.31	480	14.65	960	58.1
Various combinations for #Bins = 128K												
2K×64	30	3.17	30	12.71	120	54.24	120	3.41	120	14.43	480	60.5
512×256	30	3.21	30	13.43	240	61.67	120	3.34	120	14.9	960	60.9
Various combinations for #Bins = 256K												
2K×128	30	3.44	30	12.94	240	55.51	120	3.56	120	14.87	480	60.3
512×512	30	3.96	30	13.82	480	61.66	120	3.71	120	15.63	480	62.7
Various combinations for #Bins = 512K												
2K×256	30	4.05	30	13.58	120	56.74	60	3.6	120	15.11	480	61.6
1K×512	30	4.92	120	14.74	240	58.8	120	3.77	240	15.67	480	63.2
Various combinations for 1M =< #Bins =< 2M												
2K×512	30	7.15	30	16.28	120	57.68	60	3.94	120	15.76	480	63.6
1K×1K	30	12.19	60	20.97	240	64.21	60	4.85	240	18.87	480	74.5
2K×1K	60	21.63	30	29.93	120	69.89	60	5.05	60	18.95	240	74.8
Various combinations for #Bins > 2M												
256×256×256	30	45	30	59	120	123	60	5.1	120	21	480	90
512×512×256	30	172	30	184	120	308	60	5.8	120	22	480	92
1K×1K×256	30	604	30	695	120	811	60	6.5	120	25	480	103

Table 4.4: Comparison of Hierarchical and Iterative Split for different configuration of bins and input sizes on an Nvidia GTX 280. Times, in column T_{ms}, are in milliseconds for the optimum number of CUDA blocks, given in columns #B. Iterative Split approach is performed using the thread-serial ordered atomic operations. Hierarchical split is better when only 2 levels are needed. Iterative Split is scalable and is better if more levels are needed.

the performance of the Hierarchical Split approach for even larger number of bins. Iterative Split approach, on the other hand, is immune to this variation and the splitting time depends only on the list size and the number of bins at each level. The dependence on the number of bins can be seen to be weak and the running time grows very slowly with the number of bins for a given list size.

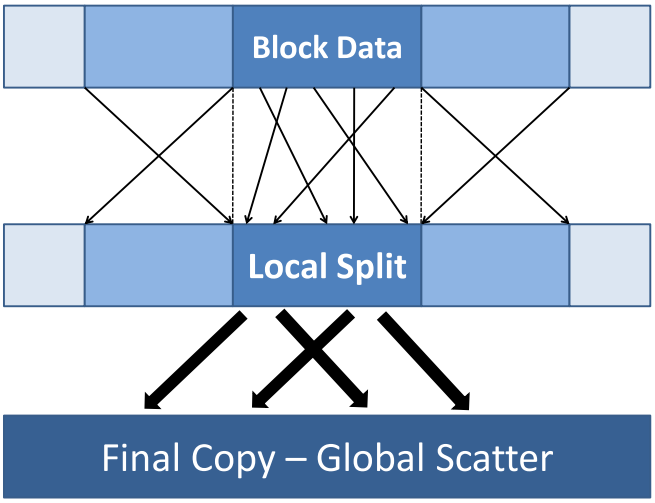


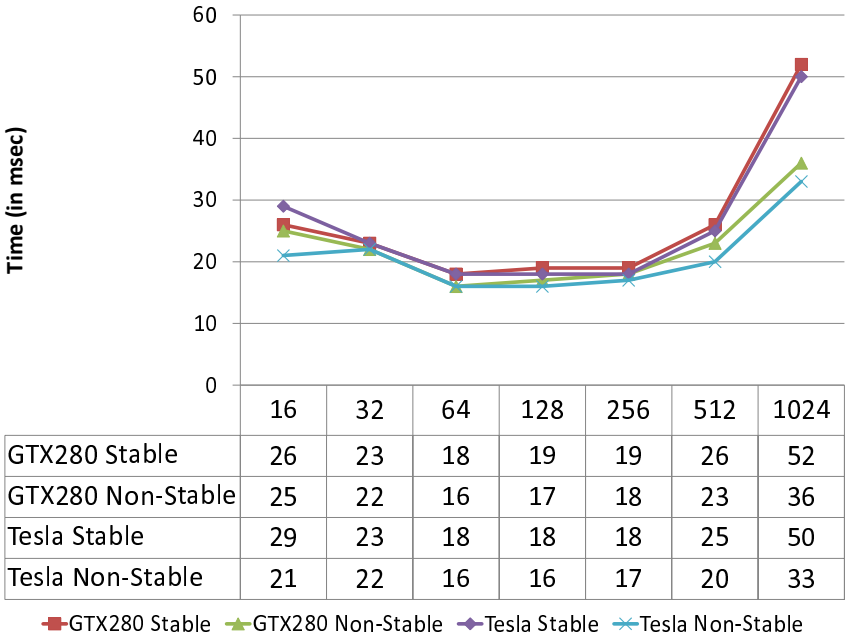
Figure 4.7: The records of a CUDA block are first locally split, followed by a copy to the final location. The instantaneous locality is better for local split than the global split. The final copy has data moving in groups and has high instantaneous locality.

4.4 Split Using a Two Step Scatter

The writing step involves a general scatter of the records. The writing step takes about 90% of the total time if implemented as described above (Figure 4.7. Writing to widely separated global memory locations is very inefficient on the GPUs. The GPU performs coalesced memory operations well. Coalescing is a dual concept of caching on uniprocessors. Caching improves the performance in the presence of temporal locality in memory accesses by the same thread. Coalescing improves the performance when there is *instantaneous locality* in the memory references by a block of consecutive threads, as the accesses are combined into a minimum number of expensive memory transactions. Completely coalesced reads can be a factor 50-100 times faster than a totally random read on current GPUs.

The range of the destination index of each record is the length N of the list in general. The expected value of the instantaneous locality is clearly inversely proportional to the range. We improve the writing speed by performing a two-step scatter operation as given in Algorithm 3. Step 3 splits each record within the segment of records handled by the corresponding CUDA block (Figure 4.7). For this, the local index of each record within its segment is calculated by each thread and the record is written to a temporary list in the global memory at that index. This is a scatter with a range of K. The range is shorter as K is much smaller than N, resulting in a slightly better instantaneous locality than global scatter. In Step 4, threads of a CUDA block read the records from this temporary list, calculates their final indexes, and copies the data to the final location. On the average, $\frac{K}{M}$ records map to the same category. They will be in consecutive positions in the temporary list as well as in the final list (Figure 4.7). High degree of instantaneous locality is ensured in Step 4 if consecutive threads handle consecutive records of the temporary list, if K is significantly smaller M. Split algorithm with 2 step scatter is given below.

As an example, for $N = 16$ million, $M = 256$ and $K = 8192$, the 2-step scatter takes 14 milliseconds, with Step 3 taking 12 ms and Step 4 the rest. The single step scatter on the same data takes 24 milliseconds. The significant speed up is due to the improved instantaneous locality in the local split operation and the high instantaneous locality in the final copy operation. Detailed



	16	32	64	128	256	512	1024
GTX280 Stable	26	23	18	19	19	26	52
GTX280 Non-Stable	25	22	16	17	18	23	36
Tesla Stable	29	23	18	18	18	25	50
Tesla Non-Stable	21	22	16	16	17	20	33

Figure 4.8: Timings for the split of 16 million elements over different number of categories/bins on the X-axis. We choose 256 bins (8 bits) as our basic split size for iterative split method.

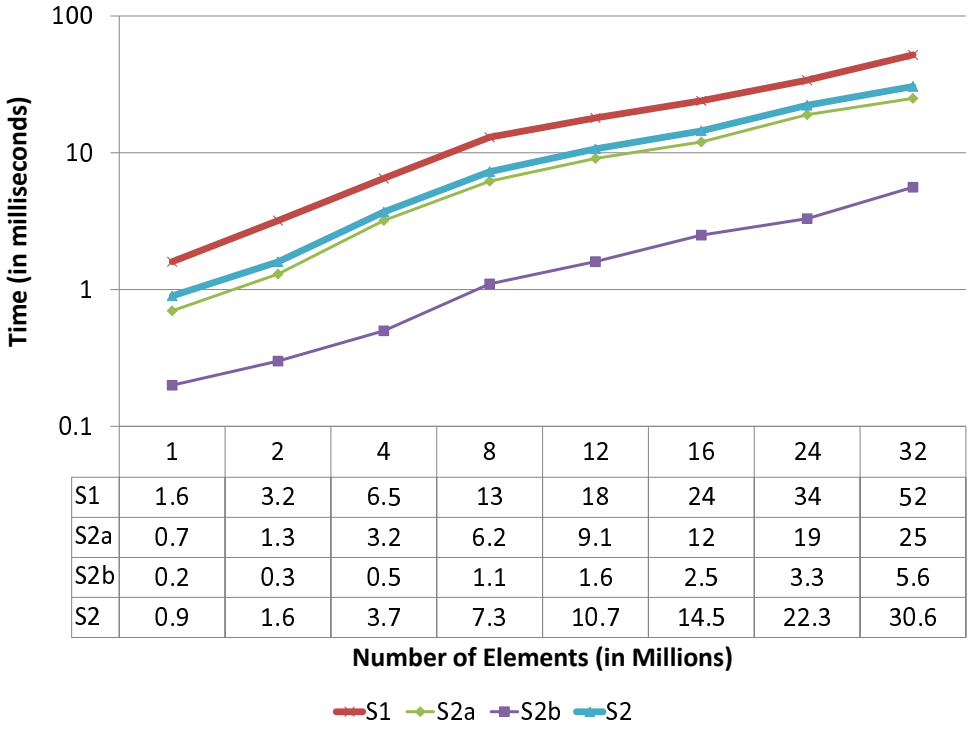
results are presented using one step scatter in Tables 4.4, 4.1, 4.2, 4.3 and Figure 4.2. Figure 4.8 shows results for basic split operation over a range of bins using two step scatter operations. Results shown in Figure 4.9 and later use two step scatter with multiple iterations of basic split operation for best performance of the split operation.

Algorithm 3 BasicSplit: split8()

1: Load the elements of the segment sequentially in each thread. Compute the count for each category per CUDA block using hardware atomic operations on the shared memory.
   – Store them in column major order in blockCount
   – Scan of the count in shared memory and store in localScan
2: Scan the blockCount array, giving the starting index of each bin for each block in globalScan
3: Split the segment locally using ordered atomics and store in localSplit.
4: Scatter localSplit to full range of output array by computing the global scatter index using globalScan and localScan

4.5 Performance of Split

Split can operate on a maximum of 1K bins in a single pass due to shared memory limitations. Figure 4.8 shows times for a single pass of split for different numbers of bins. Split performs best in the central region, where the number of bins is large enough for minimal atomic clashes and small enough for efficient use of shared memory. We use 256 bins or 8-bits of key size as the basic split operation for maximum efficiency. Figure 4.10 gives the times for splitting 64-bit records to different key sizes. All timings in this paper are taken on a single GPU of a Tesla S1070 server, unless otherwise indicated. The figure shows that the split time increases linearly with the key size



	1	2	4	8	12	16	24	32
S1	1.6	3.2	6.5	13	18	24	34	52
S2a	0.7	1.3	3.2	6.2	9.1	12	19	25
S2b	0.2	0.3	0.5	1.1	1.6	2.5	3.3	5.6
S2	0.9	1.6	3.7	7.3	10.7	14.5	22.3	30.6

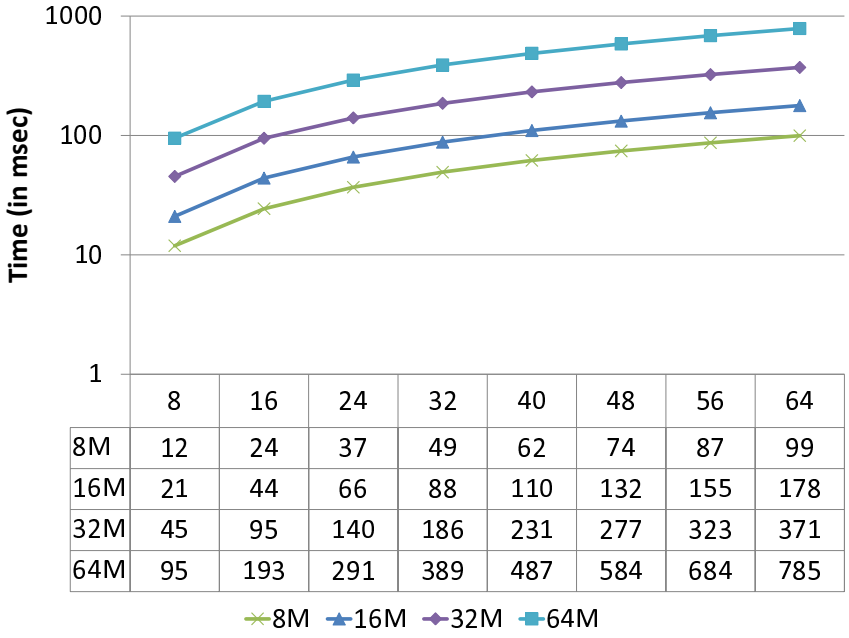
Figure 4.9: Comparison of Scatter operations. S1 performs scatter in a single step while S2 first performs local scatter (S2a) and then a global scatter (S2b). X-axis shows increasing number of elements in millions.

(or logarithmically with the number of categories). We can also see linear increase in split time as the number of records increases. All key values in all our experiments are generated using a system random generator.

Figure 4.12 gives split results for many combinations of keys and values, with the record size varying from 32 bits to 128 bits. Figure 4.13 gives the times for splitting indexes for less than 4 billion records. The index is then a 32-bit number which is used as the value with different key sizes. The dependence on the key size can be observed to be linear when record size is fixed. The dependence on the record size is sub linear in this range as larger records are read using higher access widths.

4.6 Splitting Index Values

Split is often performed on database records that are large in size. Reading and copying of the bulky records can be inefficient and wasteful, especially if split is performed in multiple steps. We can split the *indexes* of the records instead of the records themselves in such situations. The index values are less than the length N of the list. A 32-bit number can store the indexes of a list of 4 billion records, which is sufficient for most problems today. Splitting of the indexes reduces to splitting a new record consisting of the original key value and the 32-bit index value. After the split, the index part of the records will contain the index in the original list for each position. A gather applied to the original list using these indexes will split the input list. The actual data movement may not be needed in many cases as only some records of the split list are needed. For instance,



	8	16	24	32	40	48	56	64
8M	12	24	37	49	62	74	87	99
16M	21	44	66	88	110	132	155	178
32M	45	95	140	186	231	277	323	371
64M	95	193	291	389	487	584	684	785

Figure 4.10: Time to split 64-bit records using key sizes from 8 to 64 bits, on the X-axis, for lists of lengths from 8 to 64 million. Split is scalable in the key size and list length.

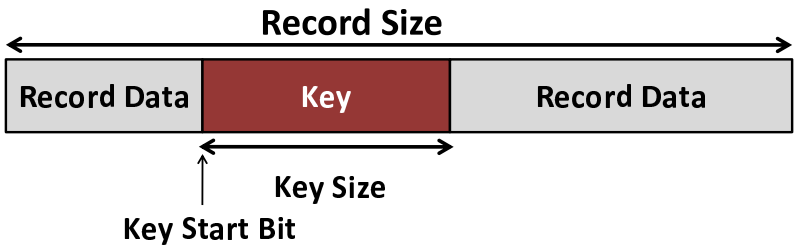
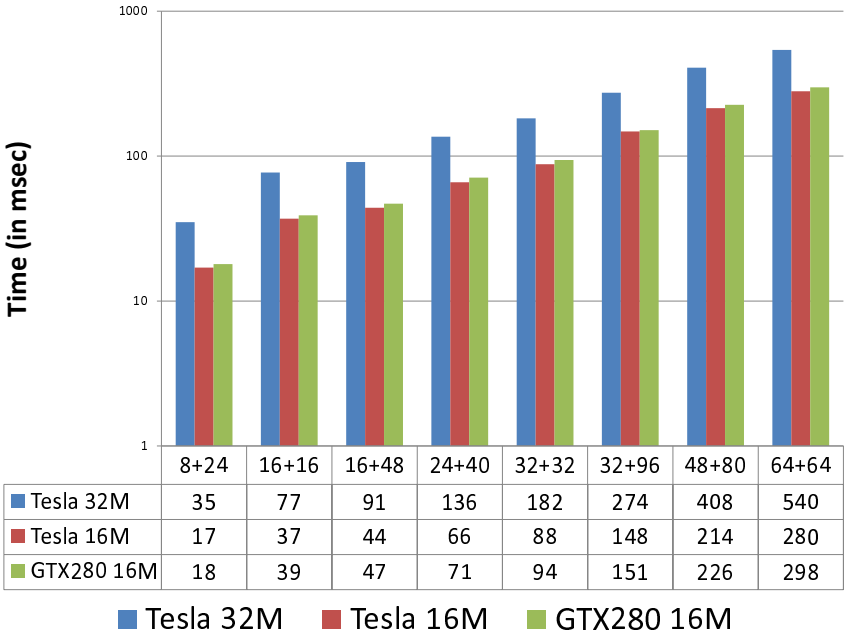


Figure 4.11: Record, key, and start bit for split primitives

only records of a few select categories may be needed after a split on a database table. Costly data movement can be avoided by accessing only the required records using the index values.

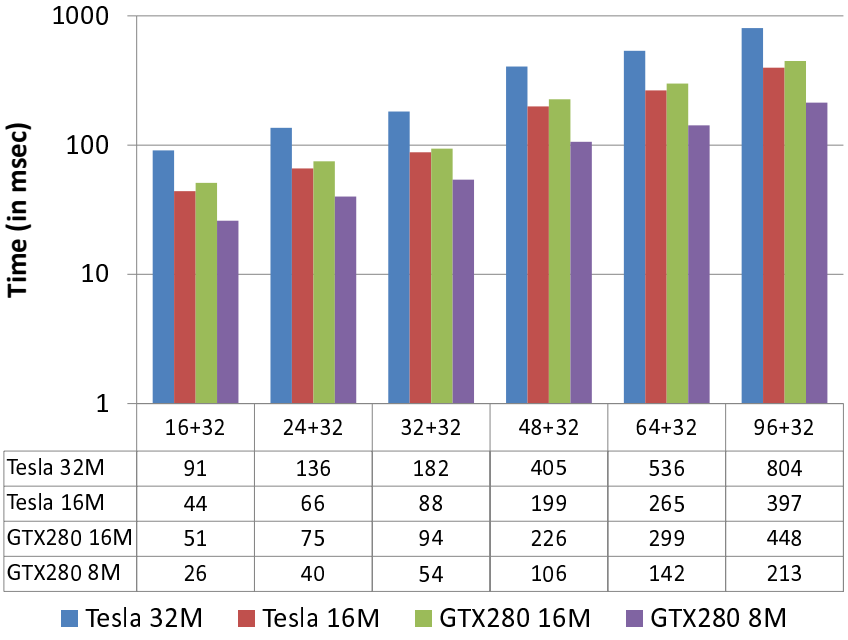
Each record is replaced by its ordinal number. Repeated application of split to exhaust all bits of the key results in the gather index for the records. To get the corresponding scatter index, we need to perform: $sIndex[gIndex[i]] = i$, using the gather index, which results in another random memory operation.

The GPU is a co-processor to the CPU which can perform compute intensive operations very fast. GPU is not good at data movement involving irregular patterns; the bandwidth available to move data between the CPU and the GPU is highly limited. The split-index primitives explicitly enable the use of the GPU as a co-processor that performs the compute intensive part of the split, leaving the data movement to the CPU or another device that is good at that. Figure 4.13 shows results for different key sizes when the value is set as a 32 bit integer. Figure 4.12 gives results for several pairs of key and value. Performance for a size of key value pair depends on the size of the key and total size of the record.



	8+24	16+16	16+48	24+40	32+32	32+96	48+80	64+64
Tesla 32M	35	77	91	136	182	274	408	540
Tesla 16M	17	37	44	66	88	148	214	280
GTX280 16M	18	39	47	71	94	151	226	298

Figure 4.12: Split timings for different record sizes and key sizes given on the X-axis as Key+Value.



	16+32	24+32	32+32	48+32	64+32	96+32
Tesla 32M	91	136	182	405	536	804
Tesla 16M	44	66	88	199	265	397
GTX280 16M	51	75	94	226	299	448
GTX280 8M	26	40	54	106	142	213

Figure 4.13: Time to split a 32 bit index value for different key sizes given on the X-axis as Key+Index.

4.7 Split Primitives on the GPU

Split is very useful in distributed data mapping. Efficient split can be a basic building block to many applications and has a role as a fundamental primitive, like scan and reduce [19]. We provide efficient implementations of the following split primitives.

Split operation requires scratch memory equal to the size of input list in order to perform the intermediate steps. Memory allocation can take significant time given the size, for e.g. allocating 4MB takes 0.6 milliseconds, 64MB takes 6 milliseconds and 512MB takes around 50 milliseconds. User may already have global memory allocated which can be used as the scratch memory. Above calls to the split primitives are overloaded to pass a pointer to the memory (**scratchMem**[]) which is to be used as scratch memory.

1. **split8(list[], rSz, sBt)**: The basic function to split the *list* of records of size *rSz* bytes to 256 bins starting with bit number *sBt* from the left of the start of the record. The output is returned in the same list.

2. **split8ns(list[], rSz, sBt)**: A non-stable version of split8 that is also slightly faster.

3. **split(list[], rSz, kSz, sBt)**: Split the list of *rSz*-byte records using a *kSz*-bit key starting at bit *sBt* of the record (Figure 4.11). This primitive uses split8 iteratively.

4. **splitGatherIndex(list[], rSz, kSz, sBt)**: Split the index values instead of the records. The function returns a list *gindex* of index values for subsequent gathering.

5. **splitScatterIndex(list[], rSz, kSz, sBt)**: Similar, but returns a list *sindex* of index values for scattering. That is, sindex[i] gives the index in the split list for the input record list[i].

6. **split8(list[], scratchMem[], rSz, sBt)**

7. **split8ns(list[], scratchMem[], rSz, sBt)**

8. **split(list[], scratchMem[], rSz, kSz, sBt)**

9. **splitGatherIndex(list[], scratchMem[], rSz, kSz, sBt)**

10. **splitScatterIndex(list[], scratchMem[], rSz, kSz, sBt)**

Chapter 5

Scatter and Gather Operations

Gather and Scatter are basic data movement primitives. They address the data movement aspect of an application in terms of reads (gather) and writes (scatter) form the memory. The nature of the operations is highly parallel but random in terms of memory access pattern. In spite of high-bandwidth (130 GBps) offered by the current GPUs, naive implementation of the above operations hampers the performance and can only utilize a part of the bandwidth.

Gather can be defined as getting data from random memory locations. The operation thus has non-uniform reads but ordered writes to the memory. It can be implemented using a gather index as:

```
outList[threadID] ← inList[gindex[threadID]]
```

ThreadID is a sequence number of each thread. The reading of `gindex` and the writing of `outList` are perfectly coalesced with very high instantaneous locality on current GPUs as consecutive threads access consecutive records. The reading of `inList` follows irregular access pattern and can be very inefficient due to low instantaneous locality. The GPU can handle 4-byte, 8-byte, and 16-byte entities in a single memory access. The above instruction completes the gather for these record sizes.

Scatter can be defined as dispersing the data to random locations in the memory. The operation has ordered reads from the memory but random writes. The operation can be implemented using a scatter index as:

```
outList[sindex[threadID]] ← inList[threadID]
```

In case of scatter, reading of `sindex` and `inList` are perfectly coalesced with very high instantaneous locality although writing of `outList` follows irregular access pattern and can be very inefficient due to low instantaneous locality.

Figure 5.1 shows the data movement using gather and scatter index. In a general scenario the data movement is highly random and is one to one. The one-to-one aspect of it makes it highly data parallel and thus a good candidate for the GPU, while the random behavior affects the performance heavily. The data movement performance of the GPU depends on the memory access patterns. Optimal accesses can be several folds faster than suboptimal ones. Optimal access patterns require deep understanding of the architecture and may not available to every user. We present two primitives for the common data mapping operations on the GPU, namely, gather and scatter, using an index list.

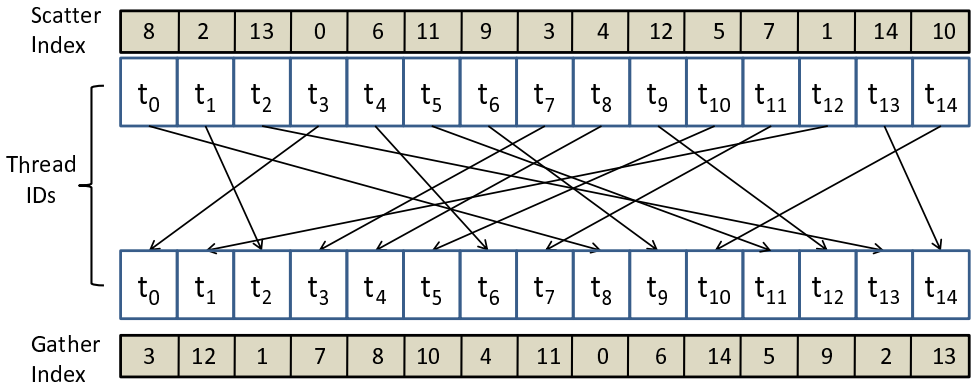


Figure 5.1: Scatter operation forces consecutive threads to write random locations in the memory causing non-coalescing

5.1 Collective Data Movement

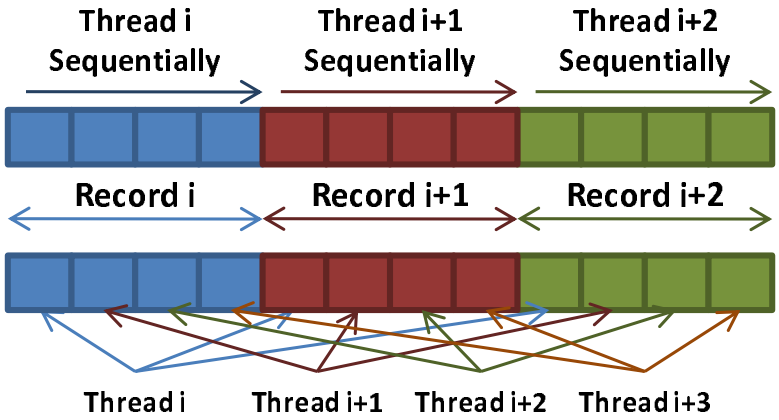


Figure 5.2: Instantaneous locality is low when each thread copies one record element by element (top). Collective copying of records by multiple threads improves the locality (bottom)

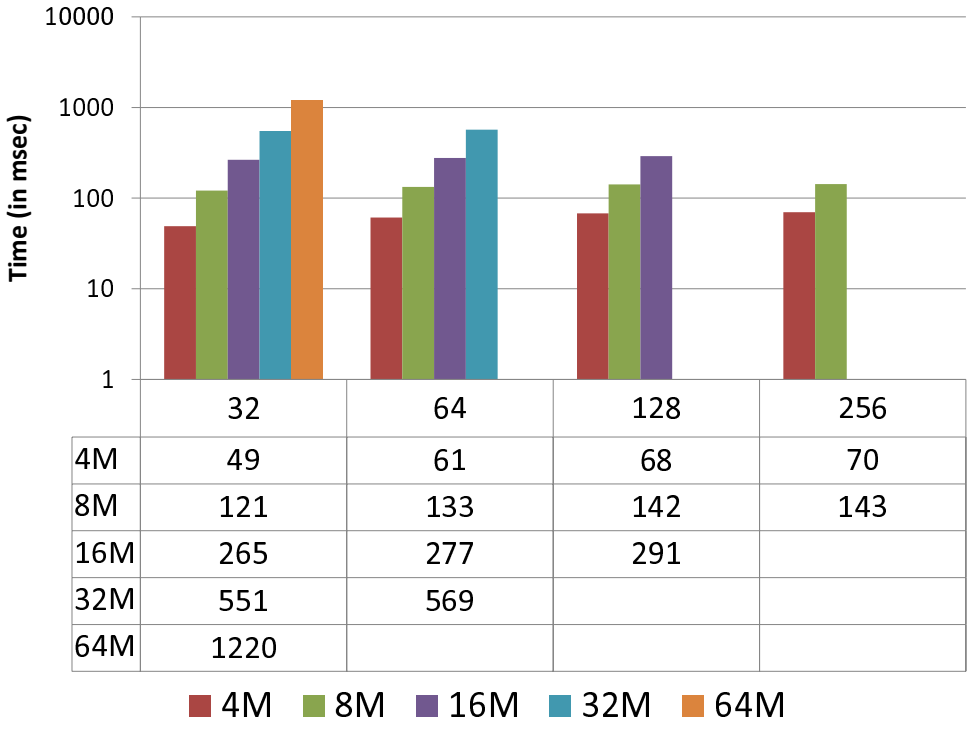
Gather and scatter of large records need to loop over elements of the same record. Since a thread moves a record, the inner loop goes over its elements. This, however, reduces the instantaneous locality of writes of gather by a factor equal to the number of data elements in the record, as the memory accessed by consecutive threads will have gaps between them. The instantaneous locality can be improved by multiple threads copying each record collectively, with consecutive threads reading and writing adjacent data elements. Figure 5.2 demonstrates the approach. In short, an array of structures can be moved most efficiently by multiple threads operating on each structure.

Current GPUs achieve the highest instantaneous locality if 16 consecutive threads (called a half-warp) access adjacent data elements. Thus, best performance is obtained when maximum number of threads cooperate on a single record, if a thread cannot load a record in a single read. The data access width should be set to 4, 8, or 16 bytes accordingly. For example, 8 threads should cooperate using 4-byte accesses on 32-byte records, 16 threads using 4-byte access on 64-byte records, and 16 threads using 8-byte access on 128-byte records, etc. The number of threads that cooperate on a record of size *recSz* bytes and the data-access width are:

$recSz \leq 64$: ($recSz$ / 4) threads and 4-byte accesses record
$recSz \leq 256$: 16 threads and 4, 8 or 16 byte accesses, depending on ($recSz$ / 16)
$recSz > 256$: ($recSz$ / 16) threads and 16-byte accesses

```
//nSubElements = number of elements in a multi-element record
//threadIdx.x = CUDA thread's ID
//blockIdx.x = CUDA block's ID
//NEPB = Number of Records to be handled per CUDA Block

scatterLarge ()
{
    lThreadId1 = threadIdx.x & (nSubElements-1);
    lThreadId2 = (int) threadIdx.x / nSubElements;
    globalIndex = blockIdx.x * NEPB;
    pitch = blockDim.x/nSubElements;
    for ( i = globalIndex; i < globalIndex + NEPB; i+= pitch )
    {
        index = dataIndex[i+lThreadId2];
        dataOut[index].element[lThreadId1]
           = dataIn[i+lThreadId2].element[lThreadId1];
    }
}
```



	32	64	128	256
4M	49	61	68	70
8M	121	133	142	143
16M	265	277	291	
32M	551	569		
64M	1220			

Figure 5.3: Results for random scatter of 4 to 64 million records given in rows of the table, for sizes 32 bytes to 256 bytes given on X-axis and columns of the table.

5.2 Performance of Gather and Scatter

Figure 5.3 shows the times for gather and scatter for combinations of number of records ranging from 4 to 64 million and record size ranging from 32 to 256 bytes, using the collective data movement scheme described above. The dependence on the number of records can be seen to be linear, especially for larger records. The dependence on the record size is highly sub-linear as all memory operations become completely coalesced with high instantaneous locality when moving larger records. From the table in Figure 5.3, the time to move 16 million 128-byte records is twice the time needed to move 8 million 256-byte records, though the total data moved is 2 gigabytes. This is because a record is collectively moved by 16 threads, each accessing 8-byte elements, in the former case whereas the latter case uses 16 threads and 16-byte accesses.

5.3 Data Movement Primitives

We provide implementations of the following data movement primitives on the GPU:

1. **gather(list[], nE, rSz, gindex[])**: Returns a list that is a permutation of the input *list* of *nE* number of records of size *rSz* bytes, with the list *gindex* providing the index to gather from.

2. **scatter(list[], nE, rSz, sindex[])**: Returns a list that is a permutation of the input *list* of *nE* number of records of size *rSz* bytes, with the list *sindex* providing the index to scatter to.

Chapter 6

SplitSort: Sort Using Split Operations

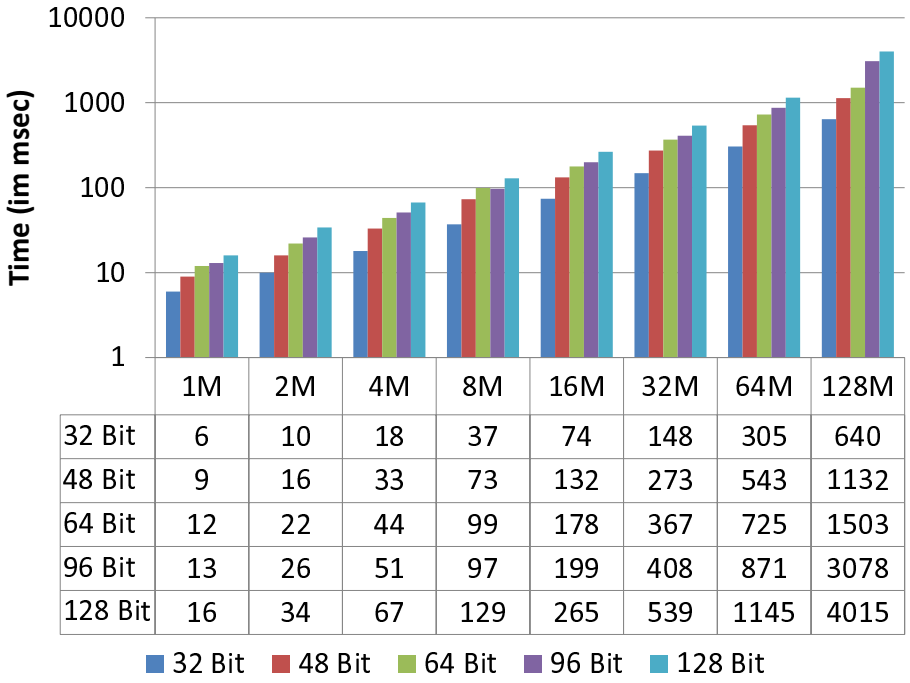
Sorting is a special case of split where the key has ordinal values and is itself interpreted as the category number. Sort also imposes a stronger condition that the categories be ordered. Our split scales linearly in the key size by applying the basic 8-bit split procedure iteratively using ordered atomic operations. Our *SplitSort* algorithm performs sorting using repeated splits, which is akin to LSD radix sort applied to a radix of 256. We describe sorting as iterative split operations in Algorithm 4. We use multiple instances of split operation (Algorithm 3) in order to sort higher number of bits. The first pass of split can use non-stable version of the split, further instances require to maintain the order of elements and needs to use stable split operation.

Algorithm 4 Sort (inputList[], numBits, startBit)

```
1:  nPasses = numBits/MAX_NUMBITS
2:  if nPasses * MAX_NUMBITS < numBits then
3:    nPasses++;
4:  end if
5:  firstPass ← 1
6:  for i = 1 to nPasses do
7:    if numBits >= MAX_NUMBITS then
8:      nBits ← MAX_NUMBITS
9:    else
10:     nBits ← numBits
11:   end if
12:   startBit ← MAX_NUMBITS * (i − 1) + startBit
13:   split8 ( inputList, firstPass, nBits, startBit );
14:   numBits ← numBits − MAX_NUMBITS
15:   firstPass ← 0                          [First pass can use non-stable split]
16: end for
```

The scalability of the basic split makes sorting also highly scalable. Figure 6.1 gives the sorting performance on one GPU of a Tesla S1070. We can clearly see the linear behavior in the number of records as well as on the key size.

The Iterative Split approach can handle arbitrarily large number of bins with a graceful increase in the running time. For instance, splitting to 16 million bins can be done in 3 passes, each of which handles 8 bits of the bin ID starting from the right. Splitting to 4 billion bins will need 4 such passes. Splitting a list of 32-bit numbers to 4 billion ($= 2^{32}$) bins is equivalent to sorting the



	1M	2M	4M	8M	16M	32M	64M	128M
32 Bit	6	10	18	37	74	148	305	640
48 Bit	9	16	33	73	132	273	543	1132
64 Bit	12	22	44	99	178	367	725	1503
96 Bit	13	26	51	97	199	408	871	3078
128 Bit	16	34	67	129	265	539	1145	4015

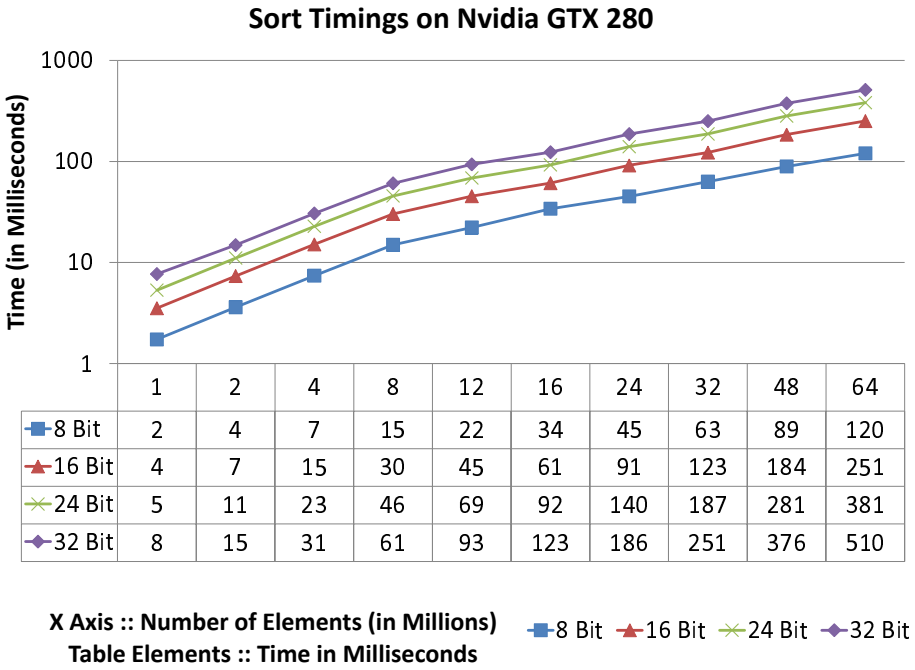
Figure 6.1: Sorting times for list lengths from 1 to 128 million given on X-axis for 32 to 128 bit numbers given in rows of the table.

data. The *Split Sort* algorithm sorts a list of numbers by splitting them to a number of bins equal to the range of the numbers being sorted. This is a type of radix sort over sets of sub-bins, whereas the previous GPU radix sort performs compacting using 1 bit of the bin ID at a time [19].

6.1 Integer Sorting

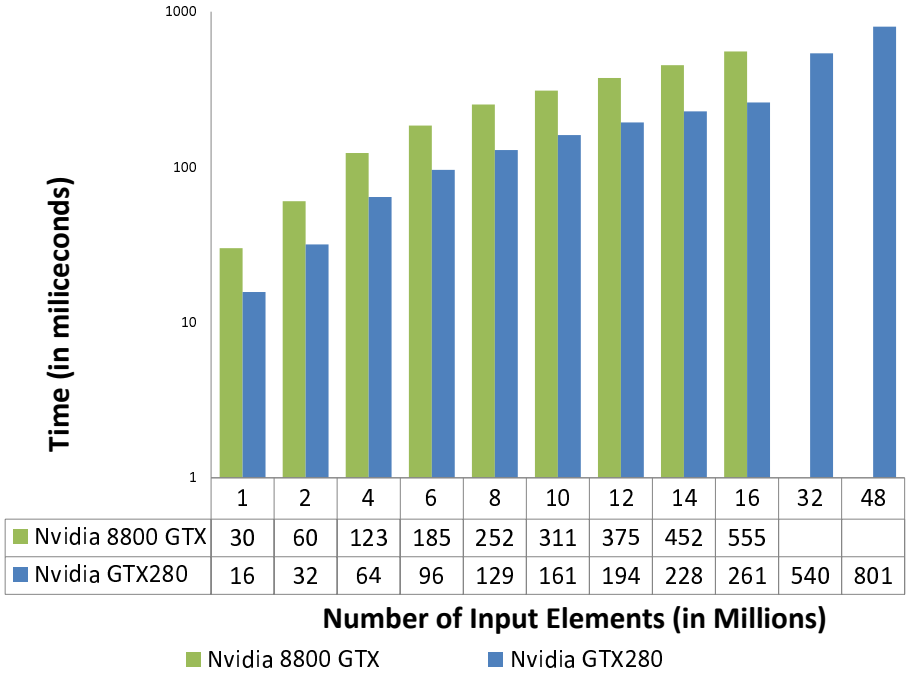
The bits of a 32-bit integer can be partitioned into $11+11+10$ bits resulting in multilevel splits to $2048 \times 2048 \times 1024$ bins. They can also be partitioned as $8+8+8+8$ bits or $256 \times 256 \times 256 \times 256$ bins. The Hierarchical Split approach performs poorly when number of bins exceeds 2 million. The Iterative Split approach can split the list to 2K bins each in the first two levels and 1K bin in the last. In general, the Iterative Split approach is scalable with the range of numbers sorted. The running time is logarithmic in the range of numbers or linear in the number of bits of representation.

Figure 6.2 present the results for different list size and different range of the numbers in the list. The cost of sorting grows nearly linearly with the number of bits or the logarithm of the range of the numbers. The first pass of the algorithm uses the hardware atomic operations as ordered atomic operations are not necessary. The G80 and G200 GPUs show the same trend in performance, with the latter performing nearly twice as fast as the former. The G80 GPU does not support atomic operations on the shared memory, but our serialized implementation of ordered atomic operation does not need it.



	1	2	4	8	12	16	24	32	48	64
8 Bit	2	4	7	15	22	34	45	63	89	120
16 Bit	4	7	15	30	45	61	91	123	184	251
24 Bit	5	11	23	46	69	92	140	187	281	381
32 Bit	8	15	31	61	93	123	186	251	376	510

Figure 6.2: Split Sort times in milliseconds for 4 sizes of the input key on an Nvidia GTX280. X-axis gives the number of elements sorted in millions.

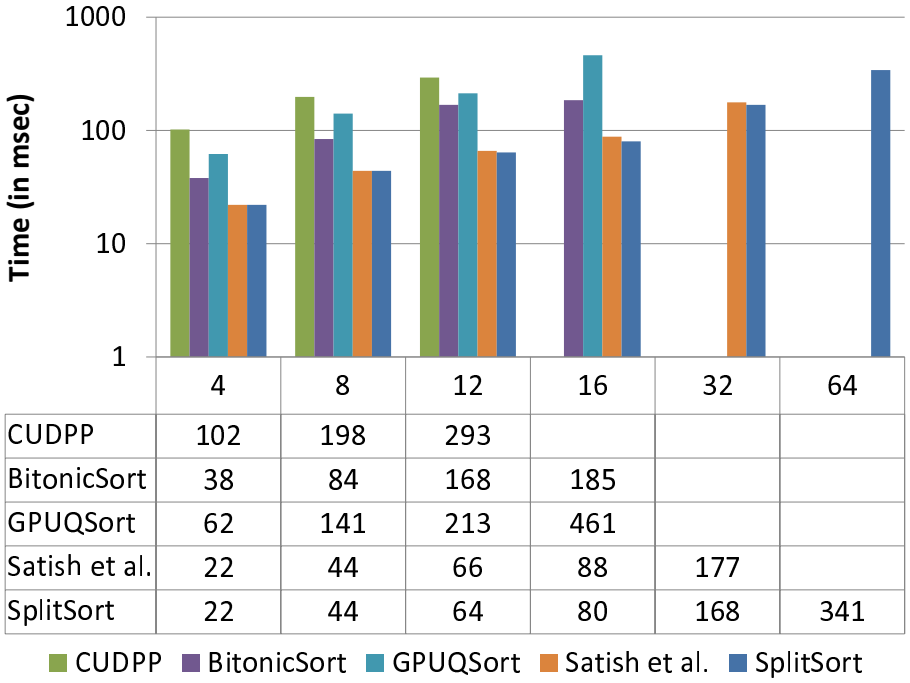


	1	2	4	6	8	10	12	14	16	32	48
Nvidia 8800 GTX	30	60	123	185	252	311	375	452	555		
Nvidia GTX280	16	32	64	96	129	161	194	228	261	540	801

Figure 6.3: Split Sort times in milliseconds for 64-bit keys on 8800 GTX and GTX 280. GTX 280 is nearly twice as fast given it has nearly twice the number of processors. X-axis gives the number of elements sorted in millions.

6.2 Comparison of Sort Implementations on the GPU

Figures 6.4 compare our results with other reported GPU sorts for lists of 32-bit integers of different lengths. The CUDPP sort is a radix sort that divides the data using 1 bit compact operations [19]. GPUQSort is an efficient implementation of Quick Sort on the GPU by Cederman et al. [9]. Satish et al. [40] extend CUDPP [19] sort algorithm by handling 4 bits in a pass of radix sort. The parallel sort by Katz et al. [28] mixes bitonic and merge sorts to obtain final results. Our algorithm is based on the Iterative Split algorithm and scales well. It displays a linear growth in the running time with increase in the input list length and a logarithmic growth in the range of the numbers (Figure 6.2). Linearity in the number of bits of representation makes our scheme applicable to sort using keys of arbitrary number of bits. Figure 6.3 presents the results of sorting 64-bit numbers using 8 levels, each splitting the entire list into 256 bins using the Iterative Split approach. To the best of our knowledge, this is the first reported results on 64-bit numbers on the GPU. The GTX280 supports 64-bit numbers, but at a significant performance penalty. In general, comparison based sorts are slower on the GPU than radix sorts. Our is, however, the first work to demonstrate scalable sorts beyond 32-bit numbers.

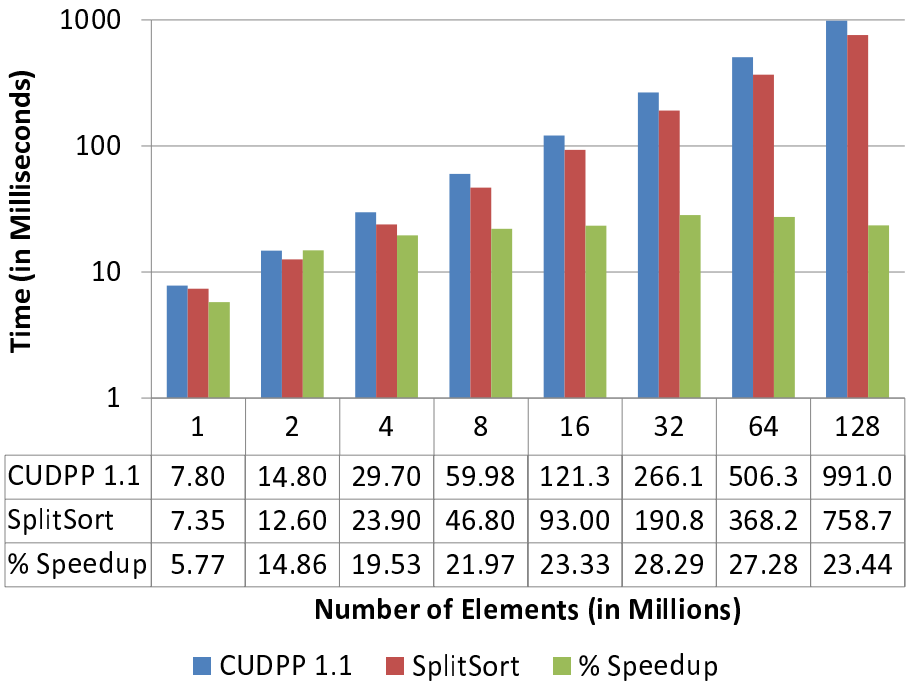


	4	8	12	16	32	64
CUDPP	102	198	293			
BitonicSort	38	84	168	185		
GPUQSort	62	141	213	461		
Satish et al.	22	44	66	88	177	
SplitSort	22	44	64	80	168	341

Figure 6.4: Comparison of our SplitSort with other reported sorts on a GTX280. Satish et al. and SplitSort use key and value pairs of 32 bit each although timings for other techniques correspond to 32 bit integer sorting. X-axis gives number of 32-bit integers sorted in millions. Timings for Satish et al. and SplitSort use 32 bit key-value pairs for sorting.

The main differences between Satish et al.'s implementation, and ours are in important details that are critical to scalable performance. Satish et al. use 1-bit radix sort within each CUDA block for the first level of sorting. This step is restricted by the available shared memory, with a block handling only 1K elements. We, on the other hand, use a 2-step scatter process to implement fast split where in the input data is always streamed in and out rather than storing it. We show 5-10% speedup on list sizes up to 32M (Figure 6.4) which is the largest size reported by Satish et al. We

show results for much larger lists (up to 128M elements, each of 128 bits Figure 6.1). We are sure the performance gap will be much wider for larger lists due to the scalability issues of per-block sorting. Figure 6.5 compares our sorting implementation with the latest release of CUDPP sort. CUDPP v1.1 implements the algorithm proposed by Satish et al. We see that our sort gains around 25% over their implementation.



	1	2	4	8	16	32	64	128
CUDPP 1.1	7.80	14.80	29.70	59.98	121.3	266.1	506.3	991.0
SplitSort	7.35	12.60	23.90	46.80	93.00	190.8	368.2	758.7
% Speedup	5.77	14.86	19.53	21.97	23.33	28.29	27.28	23.44

Figure 6.5: Comparison of SplitSort with CUDPP v1.1 sort which uses algorithm proposed by Satish et al. X-axis shows increasing number input elements in millions.

6.3 Sorting Key-Value Pairs

Database records and web log data deal with a large number of tuples or records. Such data is often arranged as key-value pairs for processing. Records are required to be sorted on the key for such cases. We give results for different sizes of key and value (in bits) in Figure 6.7. The cost of sorting a record is a function of the size of the key and the total size of the record.

Size of the records can be very large as compared to size of the key, for e.g., 128 byte records with 4 byte key. Iterative approaches require to rearrange the data in the memory more than once which can be expensive based on the size of the record. In such cases each data record is replaced with an index which is used for processing during the sorting routine. Index and the key of record form a pair which is now used to sort the records. Figure 6.8 gives performance numbers for a constant index of size 32 bits and varying size of the key. Section 6.5 gives further results for sorting large records.

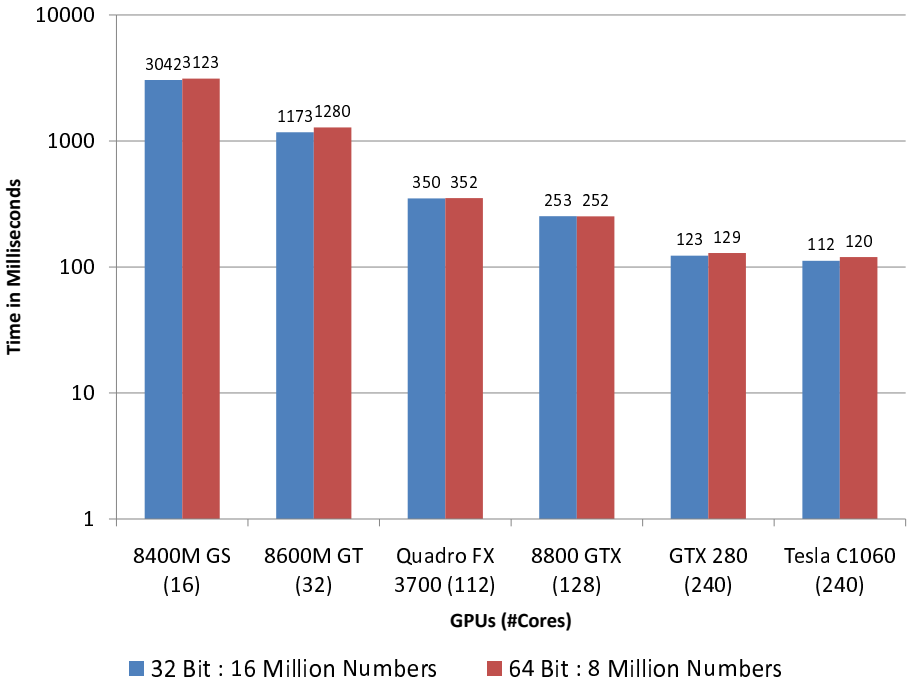
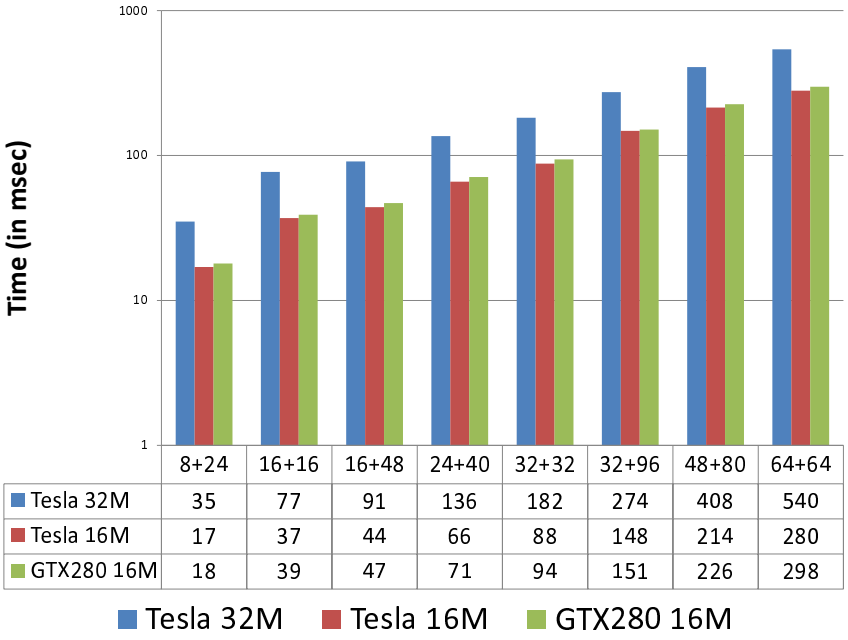


Figure 6.6: Performance of the Split Sort algorithm on different GPUs shows near-linear scaling with increasing number of cores on the GPU, given in parenthesis.

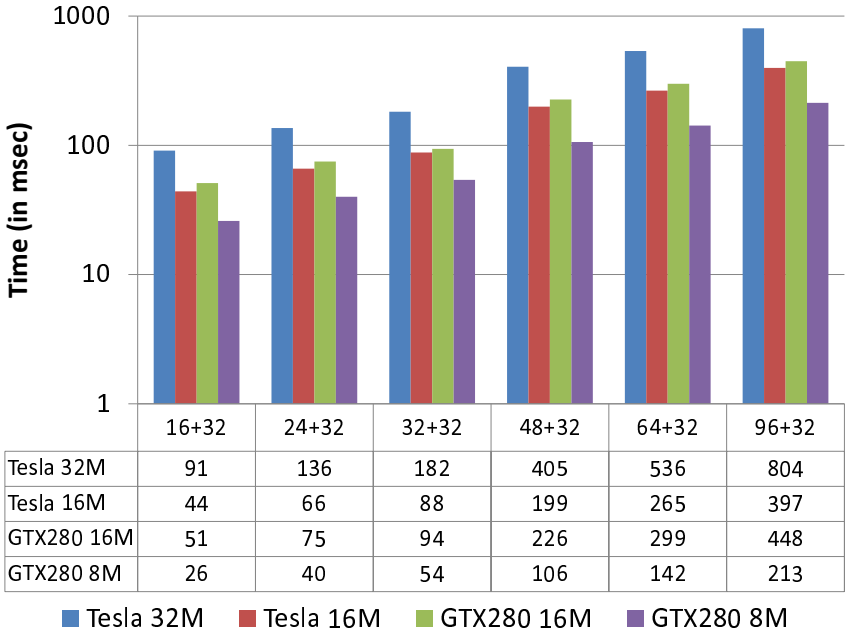


	8+24	16+16	16+48	24+40	32+32	32+96	48+80	64+64
Tesla 32M	35	77	91	136	182	274	408	540
Tesla 16M	17	37	44	66	88	148	214	280
GTX280 16M	18	39	47	71	94	151	226	298

Figure 6.7: Sort timings for different key sizes and record sizes, given as Key+Value pairs on the X-axis.

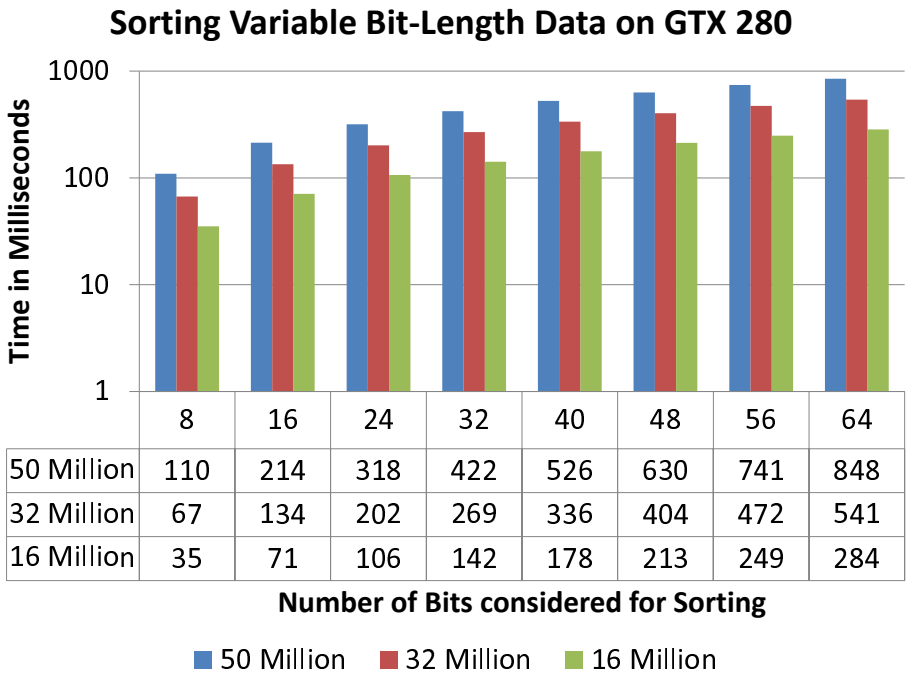
6.4 Scalability in various dimensions

Our approach experimentally shows scalability in the length of the input list (6.2), the range of input numbers (Figure 6.9) and the available number of cores in the GPU (Figure 6.6).



	16+32	24+32	32+32	48+32	64+32	96+32
Tesla 32M	91	136	182	405	536	804
Tesla 16M	44	66	88	199	265	397
GTX280 16M	51	75	94	226	299	448
GTX280 8M	26	40	54	106	142	213

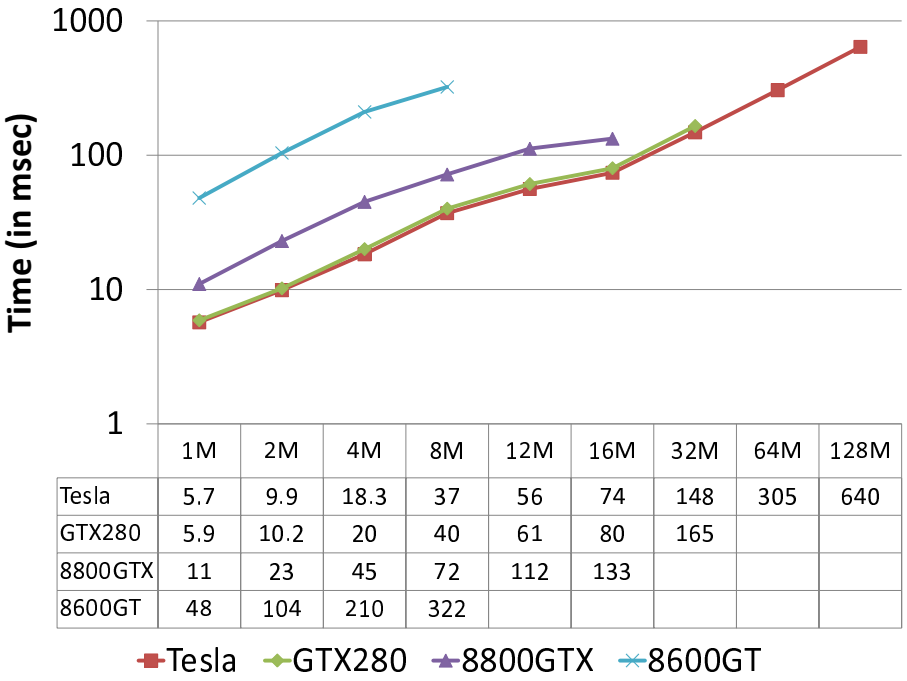
Figure 6.8: Sort timings for 32 bit index value and different key sizes given as Key+Index on the X-axis.



	8	16	24	32	40	48	56	64
50 Million	110	214	318	422	526	630	741	848
32 Million	67	134	202	269	336	404	472	541
16 Million	35	71	106	142	178	213	249	284

Figure 6.9: Performance of Split Sort for different lengths of the key. The time grows linearly with the length of key or logarithmic in the range of values.

Figure 6.10 shows another aspect of scalability of our approach. The figure shows sorting times for lists of different sizes for 32-bit numbers. The sorting time scales linearly with the list length as



	1M	2M	4M	8M	12M	16M	32M	64M	128M
Tesla	5.7	9.9	18.3	37	56	74	148	305	640
GTX280	5.9	10.2	20	40	61	80	165		
8800GTX	11	23	45	72	112	133			
8600GT	48	104	210	322					

Figure 6.10: Comparison of sort times on different GPUs. A roughly linear performance growth can be seen with increase in cores. The Tesla and the GTX280 have 240 cores each. The 8800GTX has 128 cores and the 8600GT has 32 cores.

was observed before. The sorting time also grows approximately linearly with the number of cores available on the GPU. The 8800GTX and 8600GT do not support atomic operations on the shared memory. We simulate it by serializing the clashes.

6.5 Sorting Large Records

Table 6.1 gives the time to sort 8 million to 64 million records of size 32 bytes to 256 bytes, for key sizes of 32 to 64 bits. We can sort 16 million 128-byte records in 379 milliseconds with 4-byte keys and in 556 ms with 8-byte keys. We can also sort 64 million 32-byte records in 1490 ms with 4-byte keys and in 3126 ms with 8-byte keys. These cases use up all of the 4 GB available on a single GPU of the Tesla S1070. Scalability of our approach clearly makes it possible to handle such challenging cases.

Algorithm 5 SortLargeRecords

1: gindex[] ← splitGatherIndex(list[], rSz, kSz, sBt)
2: outList[] ← gather(list[], rSz, gindex[])

Sorting 48-bit and 64-bit numbers finds use in data distribution applications, especially, data structure building. The octree built on the GPU by Zhou et al. [54] was limited to 9 levels as greater than 32-bit sorting was not available. With such scalable sort, the GPUs can be used to sort lists of large records encountered in large databases, etc. The GPU can act as a sorting co-processor if the *splitGatherIndex* primitive is used for index mapping. The data movement is separated by

Record Size	List Length				
	8M	16M	32M	48M	64M
Key size: **4 bytes**					
32B	270	352	733	1102	1488
64B	182	367	752	-	-
128B	194	373	-	-	-
256B	196	-	-	-	-
Key size: **6 bytes**					
32B	210	460	955	1720	2650
64B	230	473	972	-	-
128B	244	489	-	-	-
256B	248	-	-	-	-
Key size: **8 bytes**					
32B	251	530	1080	1980	3124
64B	263	540	1190	-	-
128B	278	555	-	-	-
256B	281	-	-	-	-

Table 6.1: Sorting large records. Times are shown in milliseconds to sort lists of length 8 to 64 million using key sizes of 4 to 8 bytes.

the use of the *gather/scatter* primitives. Sort of such records can be decomposed into a two-step process as shown below.

Chapter 7

Ray Casting of Deformable Models

Ray-casting is a highly parallel operation. In contrast to rasterization which maps the world on to the camera, ray-casting operates on every ray, yielding a highly parallel framework. In the process of ray-casting, each ray needs to process all triangles and identify the one which is closest. For a considerable amount of geometry and large image size, it becomes a computationally heavy operation. To speed up ray casting, we need to reduce the number of ray-triangle intersections per pixel/ray. This is achieved by organizing the triangles based on their position into data structures. Data structures like k-d trees, grids, octrees etc. , that organize the data spatially in the world space are used commonly. Rays traverse the data structure to find a valid subset of intersecting triangles. Cost of building the world space data structures is high. Thus, they are computed at the beginning, making them unsuitable for deformable models. Zhou et al. [55] report real-time k-d tree construction on graphics hardware for small and medium sized models. For a model with 178K triangles the construction time of k-d tree is reported to be 78 milliseconds and consequent rendering achieved 6 fps on the latest GPU. Shevtsov et al. [45] deliver 7-12 fps on models consisting of $200K$ dynamic triangles with shadows and texture.

To ray cast a million triangle model onto a million pixel window, we need a data structure that can be built and processed at real-time rates. We propose a 3-dimensional data structure, with 2-d tiles in the image-space, and depth based slabs in the third dimension. This incorporates the features of beam tracing from the point of view of rendering and restricts the number of triangle intersections per ray to provide a real time ray casting of heavy deformable models.

7.1 Data Structure for Ray Casting

We divide the rendering area into regular tiles which represent a set of rays/pixels (Figure 7.1a). We sort the triangles to the tiles and limit the rays of each tile to intersect with the triangles that fall into it. This produces batches of rays and triangles which can be independently processed on fine grained parallel machines like the GPU, Cell processor etc.

Number of triangles falling into each tile can be excessive to perform ray-triangle intersection with all the rays of the tile. If triangles in each tile are sorted in depth order the intersection can stop at the first occurrence. Sorting triangles of a tile on z completely is costly. We use a middle approach and divide the z-extent into discrete bins called slabs (Figure 7.1b). Each triangle is arranged to a slab based on its nearest z value. Triangles of a slab have no ordering with each other, but triangles from different slabs do have a front to back ordering. For small tiles, this has the potential to exploit the spatial coherence of ray-triangle intersection.

In ray-casting, all rays of a tile operate in parallel. Each ray intersects with all triangles of

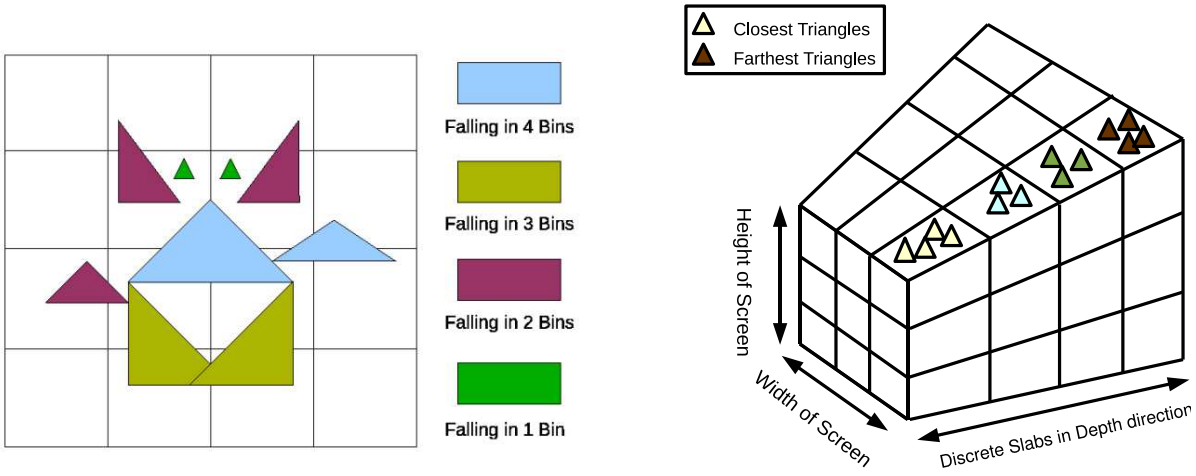


Figure 7.1: (a): 2D view of the data structure for Ray Casting. Image-space is divided into Tiles. (b): 3D view of the data structure. Tiles in the image-space are divided into frustum shaped slabs in z direction.

Dragon Preview				
Rotation Angle	0	30	60	90
RTI (M/frame) 16-Slabs	43.7	44.3	33.2	23.9
RTI (M/frame) 0-Slabs	77.2	77.8	77.3	78.8

Table 7.1: Number of Ray-Triangle Intersections (RTI) performed per frame for Dragon Model($\sim 1M$ triangles after multi-sorting triangles to tiles. With increase in depth complexity of the model, z-slabs tend to deliver better performance.

the next slab. The closest intersection point for each ray is kept track of. Rays which find a valid intersection drop out when a slab is completely processed. The computation ends, if all rays drop out. Otherwise the computation proceeds with the triangles of the next slab. Computation terminates when all slabs are done for all tiles.

7.2 Ray Casting Algorithm

The CUDA algorithm for ray-casting is given in Algorithm 6. The GPU architecture and available resources can place additional constraints on the above process. Under CUDA, we map a tile to a thread-block and each ray to a thread in it. The triangles reside in global memory, which is much slower to access than local shared memory. Since the triangles of a slab are all needed by all threads of a block, we bring the triangles to the shared memory before ray-triangle intersection are computed. The shared memory available to a block is limited on current GPU. All triangles of a slab may not fit into the available shared memory. We, therefore, treat triangles of a slab to be made up of batches which can fit into the shared memory. Triangles are loaded in units of batches. The threads of a block share the loading task equally among themselves when each batch is loaded.

Figure 7.2: Deformed Dragon and Bunny Models

The ray-triangle intersection starts after all triangles of a batch are loaded. Each thread computes the intersection of its ray with each triangle of the current batch and stores the closest intersection in the shared memory. The next batch of the slab is loaded when all threads have processed all triangles in the current batch. This repeats till the slab ends. Each thread determines if its ray found a valid intersection with the slab and sets a local flag, *rayDone* (Algorithm 6). The whole block drops out from the rest of operations, if all the rays are done. This is evaluated using a logical AND of all local flags of the block in a procedure described later. If any ray is not yet done, computation in the block continues with the next slab of triangles. All threads of the block take part in loading the triangles of subsequent batches, but the threads with rayDone set do not participate in the intersection computation.

The threads of a block operate independently. Evaluating aggregate information of data stored in different threads, such as the logical AND of a bit, is difficult and slow. We, however, use a fast technique to compute the logical AND of the individual local ray flags. First, a common memory location in the shared memory is initialized to 1. Every thread that is not done writes a 0 to it and others abstain. CUDA architecture does not guarantee any specific order of writing when multiple threads write to the same shared location simultaneously. It, however, guarantees that one of the threads will succeed. That is sufficient for the above procedure to compute a logical AND in one instruction.

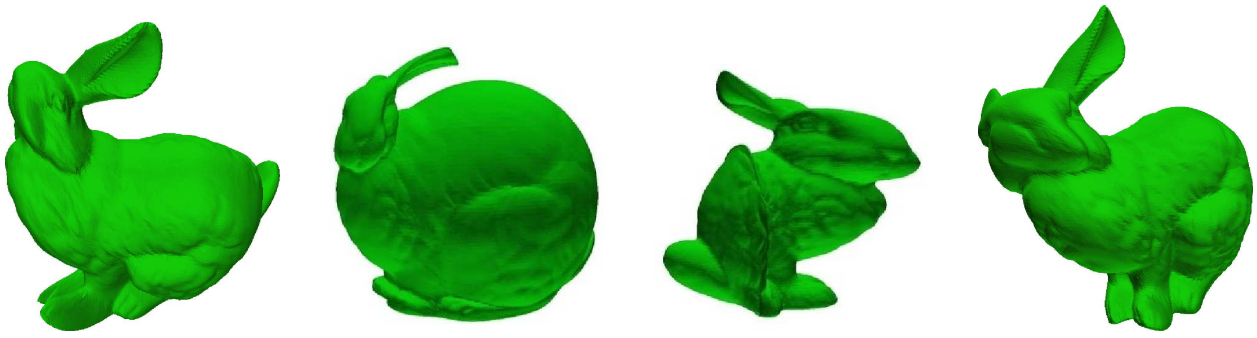
Our ray casting algorithm requires a 3-d data structure which has triangles sorted to tiles in the image space. Triangles in each tile are arranged in z-slabs which are ordered from front to back from the camera. Considering triangles to be elements which can go to more than one tile in the image space, the problem of building the required data structure is similar to performing a multi-split. Building a compact list of triangles which are arranged by tiles and slabs is not straight forward on a parallel hardware. We propose a fast implementation of split and multi-split operation on GPU which can keep up with real-time rates for fast rendering of heavy deformable models.

7.3 Using split primitive for building data structure

Ray Casting requires small tiles in the image space, the order of 8×8. Thus, large number of tiles and moderate number of slabs will work best for an efficient real-time ray casting of heavy models. For a 1024×1024 window/image size we would need 128×128 tiles in the image space along with another 16 slabs in the z-direction, thus making it a $128 \times 128 \times 16$ number of bins. We

Algorithm 6 CUDA_RAYCASTING :: Ray casting by the GPU using 3-D data structure

```
{Each Block executes the following in parallel on the GPU}
for each slab of this tile do
  for batch = 1 to maxBatch(slab) do
    Load currentBatch from global memory
    SyncThreads
    if ( !doneRay ) then
      Perform ray-triangle intersections with all the triangles
      Keep track of closest triangle, minz
    end if
    SyncThreads {All threads sync here to maintain data consistency}
  end for
  doneRay ← 1 if ray intersects
  allDone ← 1
  if ( !doneRay ) then
    allDone ← 0 {All threads in parallel}
  end if
  SyncThreads
  terminate if allDone
end for
Perform lighting/shading using the nearest intersection
```



hierarchically organize the bins and perform a 3 level split to build the required data structure.

We perform first level of split by dividing the image space into a 128 tiles in the x direction and sorting the triangle to these 128 bins. We then perform segmented split over these 128 bins by dividing each of the partitions to 128 y oriented tiles. For second and third level segmented splits, elements of each partition take part. We then perform a third level split considering the distance of triangles from the camera and binning the triangles into 16 different bins.

7.4 Ray casting with Multi Level Split

We consider highly triangulated models of the order of $1M$ for ray casting on a image size of $1M$ $(1K \times 1K)$ pixels. We assume triangles to map to $1 - 20$ pixels on the window and use this assumption to consider a triangle falling into not more than 4 tiles (a tile is 8×8 pixels in the image space). A 2-d tiled data structure in the image space is built which maps rays to CUDA threads and each tile to a CUDA block. Instead of building a data structure in the world space

Algorithm 7 RAYCASTING :: Complete algorithm for per frame building data structure and ray casting, deformable triangulated models.

1: Compute up to four (x, y) tile IDs for each triangle along with minimum z-projection coordinate
2: Perform reduction on z-projection coordinate to find out minimum and maximum z-projection value for current frame
3: First level split (Algorithm 2) is performed by looking up to four x-tile IDs for each triangle
4: Segmented second level split is performed using the y-tile IDs on the output of above step
5: Split in the z-direction is performed by computing a z-tile ID using zMin and zMax for well fitting z-slabs
6: Histogram of triangles falling into tiles and slabs is outputted along with the scan of the histogram from the above step
7: Ray Casting is performed as per Algorithm 6 using the above output

Number of Z-Slabs	DS Time (milliseconds)	RC Time (milliseconds)	Total Time (milliseconds)
1	5.2	52	62
4	5.7	30	36
8	6.4	28	34.4
16	7.5	24	31.5

Table 7.2: Data structure building time and Ray Casting time (Dragon Model) for varying number of Z-Slabs. Z-Slabs=1 corresponds to brute force ray-triangle intersection within a Tile, thus split is not performed in the z direction. Level 1 split is performed on tiles in X direction, second and third are then performed on Y and Z direction respectively.

Models→						
# Triangles	1.09M	870K	70K	346K	97K	641K
Tile Sorting	3.5	2.8	0.3	1.1	0.4	2.1
DS Building	7.5	6.5	0.8	4.2	1.4	9.1
Ray Casting	35	25	8	17	12	27
Frame Rate	22	30	110	45	72	26

Table 7.3: Data Structure building and Ray Casting time for various triangulated models.

and traversing the rays, we build a grid like structure in the image space to bring the geometry to the rays. Thus, a group of rays undergo ray-triangle intersection with a small set of geometry. To reduce the ray-triangle intersection for each ray, we divide the depth for each tile into z-slabs. A third level of hierarchy is thus built for a real time ray casting of heavy models. Multi level split which incorporates multi-split (a triangle mapping to more than one tile) and segmented-splits is described in Figure 7.3. Algorithm 7 describes the steps performed per frame.

For the purpose of ray casting we consider triangles to be sorted to tiles and slabs. Given, each triangle can fall into multiple tiles, we perform a first level of multi-split where each triangle is considered as many times as the number of tiles its falling into. Each triangle is appended to the output list multiple times corresponding to different tiles it falls in. The output size for Level 1

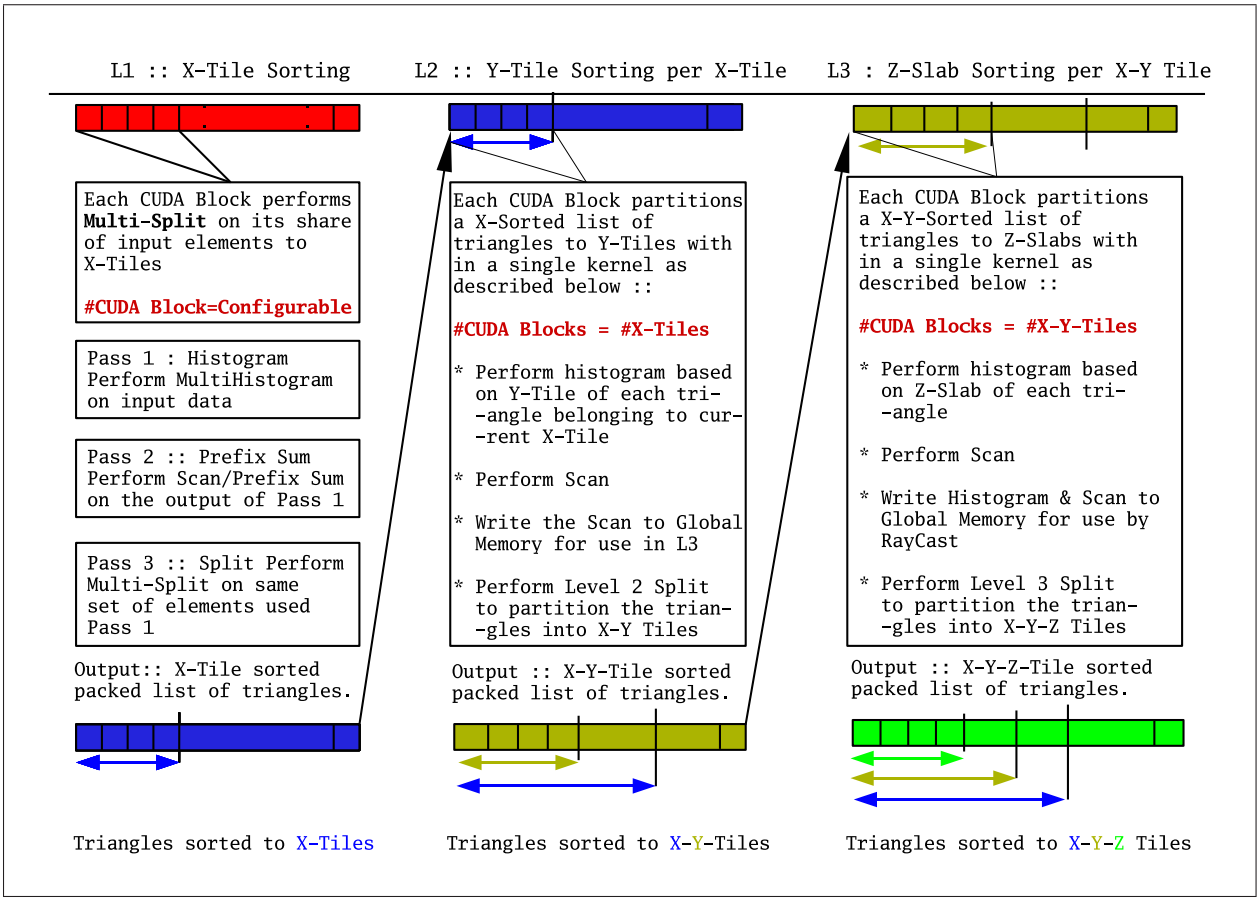


Figure 7.3: Multi Level Split as performed on triangles against X-Y Tiles and Z-Slabs for Ray casting. Level 1 (L1) splits the data and outputs a X-Tile sorted list of triangles, similarly Level 2 (L2) performs a segmented-split on output of L1 to output a X-Y Tile sorted list of triangles. L3 performs segmented split on the above list to obtain the final packed list of triangles sorted by Z-Slabs in each X-Y Tile.

is greater than original number of triangles in a model. Level 1 outputs a X-Tile sorted list of triangles which are further considered for segmented-splits which is single-split as the triangles are now considered only once. At first level of split (L1), number of CUDA blocks are configurable. We consider 64 blocks which equally divide the input number of elements in order to perform the split.

First level of split outputs a X-tile sorted list of triangles with the starting point for each X-tile partition and the number of triangles belonging to the partition. Second level of split (L2) uses the output from L1 and performs a segmented-split on each of the partitions. We now have a hierarchy where each CUDA block performs split on a X-tile partition hence, we have as many CUDA block as there are number of X-tiles. Each block loads the partition of elements corresponding to its block/X-tile and builds the histogram based on triangle's Y-tile. Within the kernel each block performs a scan and uses it to perform the split, thus each block rearranges the X-tile partition by partitioning them into Y-tiles within the X-tile.

Second level of split outputs a X-Y-tile sorted list of triangles along with the histogram and scan of each partition. Third level (L3) divides each of these partitions (# X-Y-Tiles) based on Z-slab of each triangle. Z-slabs are decided based on distance of each triangle from the camera center.

Each triangle is projected on the screen using the ModelView and Projection matrix for the current frame. Projection coordinate range from 0.0 to 1.0. Triangles tend fall into a small range due to non-linear depth distribution. We therefore compute a zMin and zMax which correspond to the z projection coordinate of closest and farthest triangle respectively. We divide the constant number of Z-slabs between zMin and zMax, getting a well fitted and well distributed partition of triangles. With # X-Y-tiles number of CUDA blocks, we partition each of the set into these discrete Z-slabs to output a 3-d data structure along with the counts and starting point for each partition to be used by ray casting process. We use an image size of 1024×1024 which is divided into 128×128 tiles each of size 8×8. Each tile is partitioned into 16 slabs. We perform a $128 \times 128 \times 16$ multi level split to build the desired data structure.

Data structures for ray tracing are used to bring down the number of ray-triangle intersections per pixel. Our data structure divides the geometry in three dimensions resulting in frustum shaped voxels. The effectiveness of our data structure (low number of ray-triangle intersections) is directly associated with the number of these voxels, i.e., the resolution of the three dimensional grid. We currently use a grid size of $128 \times 128 \times 16$ for an image size of 1024×1024. Let us consider the size of the grid in two dimensions as 1×1. For 1 million pixels and 1 million triangles, a brute force approach will take 1 million $\times$ 1 million intersections in the worst case. If we grow the size of the grid to 2×2 and divide the geometry into these cells based on their positions, we are now guaranteed to perform at most $(1million \times 1million)/(2 \times 2)$ ray-triangle intersections. Thus, grid size in 2 dimensions directly affect the maximum number of ray-triangle intersections performed per frame. The division of each of the 2-d grid cells into depth slabs help us terminate a pixel early without performing all the brute force intersections within a cell. From the point of view of CUDA, a grid cell is mapped to a CUDA block and each of its pixels as CUDA threads, thus, with this idea we need to have sufficient number of threads per block also minimizing the number of ray-triangle intersections.

Chapter 8

Conclusions and Future Work

In this thesis, we presented variations of the split primitive which is required for many data processing applications on parallel architectures. Split was efficiently implemented using single histograms per block which allowed us to handle up to 2048 bins in a single pass of split. We used hardware atomic operations on the shared memory to overcome the previous limitation of using separate memory space for each thread. We proposed ordered atomic operations on the shared memory which is required for implementation of stable split operation. Our initial implementation of split used a single step scatter where the scatter operation consumed 90% of the total split time. We described and implemented an efficient 2 level scatter to improve the instantaneous locality of memory references. We achieved a 30% improvement in the performance of split operation using the 2 step scatter. We also presented a SplitSort algorithm that uses iterative splits that is the fastest sort today on the GPU. It can also handle arbitrary sizes of keys. We implemented efficient gather and scatter data movement operation which are used with variants of split and sort to process large, multi-byte records. We presented ray tracing as an applications of the data primitives by building data structure per frame for models with more than 1 million triangles. We also implemented a fast minimum spanning tree algorithm using variants of split primitives.

The architecture of the massively multi-threaded parallel GPUs. High level application on the GPU demands efficient primitive operations. GPU is complex and getting high performance is difficult. We have explained the design and implementation of few such primitives for data mapping and movement on the GPU. Experiments and analysis of the various features like memory transactions, scheduling provided us with valuable insights.

The split and split-index primitives have high device memory requirements: another copy of the input list and extra memory for the histogram. The number of bins is 256 since our basic split splits to at most 256 categories at a time. The number of partitions of the data is in the range of 8000 to 10000. The extra memory requirements can limit the application of the split primitives. We can reduce memory requirements by not storing the input and output in its entirety on the device memory. The input and output can be streamed in and out of the GPU in parts for processing.

We also propose to extend our ray cast algorithm to perform full ray tracing. Shadow rays start from the point of intersection of primary ray and the geometry, and is directed towards the light source. Shadows are rendered per pixel by finding out if the light from the source is reaching the pixel or not. The data structure we use for ray cast can be used as a 3-d grid for secondary rays. Fast enumeration of grid cells intersecting a ray can be done using a 3DDDA algorithm. We traverse these grid cells and perform ray-triangle intersections for the secondary rays. Other secondary rays like reflection and refracted rays also require enumeration of cells to compute the effects.

We have released optimized implementations of these primitives for general use. These primitives

can find a lot of applications in processing irregular data such as graphs and databases on massively multi-threaded architectures like the GPU. We outlined how their use can accelerate applications like data structure building for ray tracing etc.

Related Publications

Suryakant Patidar, P. J. Narayanan. Scalable Split and Sort Primitives using Ordered Atomic Operations on the GPU, High Performance Graphics (Poster), April 2009.

Vibhav Vineet, Harish P K, **Suryakant Patidar**, P. J. Narayanan. Fast Minimum Spanning Tree for Large Graphs on the GPU, High Performance Graphics, April 2009.

Kishore K, Rishabh M, Suhail Rehman, **Suryakant Patidar**, P. J. Narayanan, Kannan S. A Performance Prediction Model for the CUDA GPGPU Platform. International Conference on High Performance Computing, April 2009.

Suryakant Patidar, P. J. Narayanan. Ray Casting Deformable models on the GPU, In Proceedings of the 7th Indian Conference on Computer Vision, Graphics and Image Processing.(ICVGIP 2008).

Shiben Bhattacharjee, **Suryakant Patidar**, P. J. Narayanan. Real-time Rendering and Manipulation of Large Terrains, In Proceedings of the 7th Indian Conference on Computer Vision, Graphics and Image Processing. (ICVGIP 2008).

Soumyajit Deb, Shiben Bhattacharjee, **Suryakant Patidar**, P. J. Narayanan. Real-time Streaming and Rendering of Terrains In Proceedings of the 6th Indian Conference on Computer Vision, Graphics and Image Processing. (ICVGIP 2006).

Suryakant Patidar, P. J. Narayanan. Scalable Split and Sort Primitives using Ordered Atomic Operations on the GPU, IIIT/TR/2009/99, February 2009

Suryakant Patidar, Shiben Bhattacharjee, Jag Mohan Singh, P. J. Narayanan. Exploiting the Shader Model 4.0 Architecture, IIIT/TR/2007/145, March 2007.

Bibliography

[1] S. G. Akl. *Parallel Sorting Algorithms.* Academic Press Inc. U.S., 1990.

[2] Arthur Appel. Some techniques for shading machine renderings of solids. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 37–45, New York, NY, USA, 1968. ACM.

[3] Gianfranco Bilardi and Alexandru Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines. Technical report, Cornell University, Ithaca, NY, USA, 1986.

[4] G. Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, 1990.

[5] Guy Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38:1526–1538, 1987.

[6] Guy Blelloch. Scan primitives as parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, 1989.

[7] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *In Proceedings of Graphics hardware*, 2002.

[8] Nathan A. Carr, Jared Hoberock, Keenan Crane, and John C. Hart. Fast gpu ray tracing of dynamic meshes using geometry images. In *Proceedings of Graphics Interface*, 2006.

[9] Daniel Cederman and Philippas Tsigas. A practical quicksort algorithm for graphics processors. In *Proceedings of the 16th annual European symposium on Algorithms*, 2008.

[10] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *Proc. VLDB Endow.*, 1(2):1313–1324, 2008.

[11] Siggraph Asia Courses. Beyond programmable shading, 2008.

[12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Operating Systems Design and Implementation*, pages 137–150, December 2004.

[13] Yuri Dotsenko, Naga K. Govindaraju, Peter-Pike Sloan, Charles Boyd, and John Manferdelli. Fast scan algorithms on graphics processors. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 205–213, New York, NY, USA, 2008. ACM.

[14] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *In Proceedings of the Graphics hardware*, 2005.

[15] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: High performance graphics co-processor sorting for large database management. In *Proceedings of ACM SIGMOD International Conference on Management of data*, 2006.

[16] Naga K. Govindaraju, Scott Larsen, Jim Gray, and Dinesh Manocha. A memory model for scientific algorithms on graphics processors. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 89, New York, NY, USA, 2006. ACM.

[17] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete fourier transforms on graphics processors. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008.

[18] Xianfeng Gu, Steven J. Gortler, and Hugues Hoppe. Geometry images. *ACM Trans. Graph.*, 21(3), 2002.

[19] Mark Harris, John D. Owens, Shubho Sengupta, Yao Zhang, and Andrew Davidson. Cuda data parallel primitives library, 2007.

[20] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *PACT: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM, 2008.

[21] Bingsheng He, Naga K. Govindaraju, Qiong Luo, and Burton Smith. Efficient gather and scatter operations on graphics processors. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM.

[22] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 2008.

[23] Paul S. Heckbert and Pat Hanrahan. Beam tracing polygonal objects. In *Proceedings of the conference on Computer graphics and interactive techniques*, 1984.

[24] J. Hensley, T. Scheuermann, G. Coombe, M. Singh, and A. Lastra. Fast summed-area table generation and its applications. *Proc. Eurographics*, pages 547–555, 2005.

[25] D Horn. *Stream Reduction Operations for GPGPU Applications*, pages 573–589. Addison Wesley, 2005.

[26] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree gpu raytracing. In *In Proceedings of I3D 2007*, 2007.

[27] K Iverson. *A Programming Language*. Wiley, New York, 1962.

[28] Alan Katz. An implementation of bitonic/merge sort. `http://courses.ece.uiuc.edu/ece498/al1/halloffame.html`, 2008.

[29] Khronos. Opencl : Open compute library, 2009.

[30] A. Knoll, Y. Hijazi, C. Hansen, I. Wald, and H. Hagen. Interactive ray tracing of arbitrary implicits with simd interval arithmetic. *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on*, 2007.

[31] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast bvh construction on gpus. In *Proceedings of Eurographics*, 2009.

[32] Akira Nukada, Yasuhiko Ogata, Toshio Endo, and Satoshi Matsuoka. Bandwidth intensive 3-d fft kernel for gpus using cuda. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008.

[33] Nvidia. Nvidia cuda : Compute unified device architecture, 2008.

[34] Suryakant Patidar and P. J. Narayanan. Ray casting deformable models on the gpu. In *Indian Conference on Computer Vision, Graphics and Image Processing*. IEEE Press, 2008.

[35] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, 21(3):703–712, 2002.

[36] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 258, New York, NY, USA, 2005. ACM.

[37] Erik Reinhard, Brian E. Smits, and Chuck Hansen. Dynamic acceleration structures for interactive ray tracing. In *Proceedings of the Eurographics Workshop on Rendering Techniques*, 2000.

[38] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. *ACM Trans. Graph.*, 24(3), 2005.

[39] Steven M. Rubin and Turner Whitted. A 3-dimensional representation for fast rendering of complex scenes. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 110–116, New York, NY, USA, 1980. ACM.

[40] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *Proceedings of International Parallel and Distributed Processing Symposium*, 2009.

[41] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3), 2008.

[42] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 97–106, 2007.

[43] R. Shams and R. A. Kennedy. Efficient histogram algorithms for NVIDIA CUDA compatible devices. In *International Conference on Signal Processing and Communication Systems*, 2007.

[44] Perumaal Shanmugam and Okan Arikan. Hardware accelerated ambient occlusion techniques on gpus. In *Proceedings of symposium on Interactive 3D graphics and games*, 2007.

[45] Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. *Computer Graphics Forum*, 2007.

[46] Jag Mohan Singh and P. J. Narayanan. Real-time ray-tracing of implicit surfaces on the gpu. *IEEE Transactions on Visualization and Computer Graphics*, 99(1), 2009.

[47] Vibhav Vineet, Harish P. K., Suryakant Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the gpu. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Symposium on High Performance Graphics*, 2009.

[48] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26(1), 2007.

[49] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G. Parker. Ray tracing animated scenes using coherent grid traversal. *ACM Trans. Graph.*, 25(3), 2006.

[50] Ingo Wald, William R. Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G. Parker, and Peter Shirley. State of the art in ray tracing animated scenes. In *STAR Proceedings of Eurographics*, 2007.

[51] Li-Yi Wei, Baoquan Liu, Xu Yang, Chongyang Ma, Ying-Qing Xu, and Baining Guo. Nonlinear beam tracing on a gpu, msr-tr-2007-168. Technical report, Microsoft Research Asia, 2008.

[52] Ruigang Yang and Marc Pollefeys. A versatile stereo implementation on commodity graphics hardware. *Real-Time Imaging*, 11(1):7–18, 2005.

[53] Sung-Eui Yoon, Christian Lauterbach, and Dinesh Manocha. R-lods: Fast lod-based ray tracing of massive models. *Vis. Comput.*, 22(9), 2006.

[54] Kun Zhou, Minmin Gong, Xin Huang, and Baining Guo. Highly parallel surface reconstruction. Technical Report MSR-TR-2008-53, Microsoft Research, April, 2008.

[55] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, 2008.