Fast Data Parallel Radix Sort Implementation in DirectX 11 Compute Shader to Accelerate Ray Tracing Algorithms

Conference Paper · October 2014

CITATIONS
2

READS
991

4 authors:

Arturo García
Intel
15 PUBLICATIONS **20** CITATIONS
SEE PROFILE

Omar Alvizo
University of Guadalajara
2 PUBLICATIONS **2** CITATIONS
SEE PROFILE

Ulises Olivares-Pinto
Universidad Nacional Autónoma de México
15 PUBLICATIONS **30** CITATIONS
SEE PROFILE

Félix Ramos
Center for Research and Advanced Studies of the National Polytechnic Institute
160 PUBLICATIONS **472** CITATIONS
SEE PROFILE

Fast Data Parallel Radix Sort Implementation in DirectX 11 Compute Shader to Accelerate Ray Tracing Algorithms

Arturo García*‡ Omar Alvizo† Ulises Olivares‡ Félix Ramos‡

* Visual and Parallel computing group, Intel Corporation, e-mail:arturo.garcia@intel.com

†University of Guadalajara, México, e-mail: omar.alvizo@alumno.udg.mx

‡Departament of Electrical Engineering Center for Research and Advanced Studies of the National Polytechnic Institute email: uolivares, framos@gdl.cinvestav.mx

***Abstract*—In this paper we present a fast data parallel implementation of the radix sort on the DirectCompute software development kit (SDK). We also discuss in detail the various optimization strategies that were used to increase the performance of our radix sort, and we show how these strategies can be generalized for any video card that supports the DirectCompute model. The insights that we share in this paper should be of use to any General-Purpose Graphics Processing Unit (GPGPU) programmer regardless of the video card being used. Finally, we discuss how radix sort can be used to accelerate ray tracing and we present some results in this area as well.**

I. Introduction

Sorting algorithms is one of the most heavily researched area in the history of computer science. These algorithms have multiple applications in various areas since they are a necessary step for fast information retrieval. For instance, specialized sorting algorithms exist for external storage (databases) due to their much slower access time.

On the other hand, sorting algorithms that operate on RAM have been extensively studied for decades. The aim of this research is to get greater performance. For instance, in [1], Batcher investigates the use of sorting networks to increase the performance of the bitonic sort. Later, Zagha and Blelloch studied the problem of parallelizing the radix sort algorithm [2].

Nowadays, it is possible for the masses to have access to parallel computing architectures like multi-core processors, General-Purpose GPU (GPGPU) and Many Integrated Core architectures [3]. In the area of sorting algorithms, the Compute Unified Device Architecture (CUDA) data-parallel primitive (CUDPP) library implementation included in the Thrust productivity library [4] and presented by Merrill et al. [5] is currently one of the fastest radix sort algorithms on a GPGPU architecture.

One of the most eye-catching uses for the sorting algorithms on the GPGPU is the implementation of real-time ray tracing. Ray tracing is a technique that simulates how light is transported in a 3D scene and thus can achieve photo-realistic images. However, it requires the computation of several intersections between light rays and objects. Therefore, in order to achieve real-time frame rates with ray tracing, it is necessary to sort the geometry first. In this paper we will exemplify the use of the GPGPU radix sort to implement a real-time ray tracing engine.

As previously mentioned, the CUDPP library has the fastest radix sort implementation for GPGPU [4]. This implementation has several optimizations that are CUDA-specific and, therefore, make it hard to implement this algorithm in other video cards while maintaining the same performance. In this paper we focus on these optimizations and how they can be generalized to remove their dependencies from the CUDA SDK, therefore making it possible to write competitive radix sort algorithms in a multitude of video cards.

The implementation presented in this paper was done on DirectCompute, so it runs on AMD Radeon, Intel and NVIDIA video cards. However, the discussion of the optimizations should allow the implementation of a fast radix sort using other GPGPU programming interfaces like Open Computing Language (OpenCL), and they should also prove useful for the fine-tuning of other GPGPU algorithms.

II. Related Work

Owens et al. present a survey in [6] of a mapping of various algorithms into the GPGPU architectures. This is also a good introductory read to the area of GPU computing as it presents several basic concepts and a multitude of algorithms including sorts.

Hillis et al. [7] describe a data parallel algorithm to sum an array of n numbers that can be computed in time $O(\log n)$ by organizing the addends at the leaves of a binary tree and performing the sums at each level of the tree in parallel. This algorithm is the base for the implementation of an efficient scan algorithm, which in turn is a building block necessary for a fast radix sort algorithm.

Blelloch [8] presents an early specification of the radix sort and scan algorithms for generic parallel architectures. The current radix sort implementations for data parallel computing

architectures are based on the scan algorithms presented in this book.

Harris et al. discuss an efficient implementation of the prefix sum in [9] for CUDA. The prefix sum is an algorithm that can be used to increase the performance of the radix sort by pre-computing the offsets where the elements will be stored. In this way, when elements are swapped during the sort, the algorithm already knows a lower index for each element. Further code and details are provided in [10] where segmented scan, intra-block scan, intra-warp scan and global scan algorithms are presented, again for the CUDA SDK.

In [11], Satish et al. describe the design of a high-performance parallel radix sort for GPGPUs. Their discussion is centered around the CUDA SDK. At the time the paper was published it was the fastest implementation of the radix sort. The core of their optimization relies on reducing the global communication between different threads to a minimum by breaking tasks into sizes that are compatible with the underlying hardware; minimizing the number of scatters to global memory and maximizing the coherence of scatters. The last point is achieved by using an on-chip shared memory to locally sort data blocks.

Merrill et al. [5] superseded the Satish work by presenting a high performance and scalable radix sort implementation where it applies a radix sort strategy based on the fusion of kernels, multi-scan and thread-block serialization reducing the aggregate memory workload and thread synchronizations.

In [12], Eric Young discusses several optimization strategies for the DirectCompute SDK; but it also relies on the CUDA architecture for his discussion. In that presentation are discussed techniques such as coalesced access, proper thread group shared memory usage, memory bank conflicts, maximization of hardware occupancy and proper decomposition of tasks to void global synchronization and other subjects are discussed.

It can be seen that most material on GPGPU optimization is centered on the CUDA SDK. In this paper, we take several points from this material and extend the discussion to other hardware architectures.

III. Data Parallel Computing

General-Purpose computing on graphics hardware can provide a significant advantage to implement efficient data parallel algorithms. Today modern GPGPUs provide high data throughput, memory bandwidth, high-level programming languages to abstract the programmable units. The programing environments abstract these hardware programmable units to allow efficient data parallel processing, data transfer, matching the parallel processing resources and memory system available on the GPU.

While this programmability was first used to accelerate graphic applications, it has transformed the GPGPUs into powerful platforms for high performance computing with a wide variety of applications such as sorting algorithms, matrix multiply, ray tracing, collision detection, linear algebra, etc.

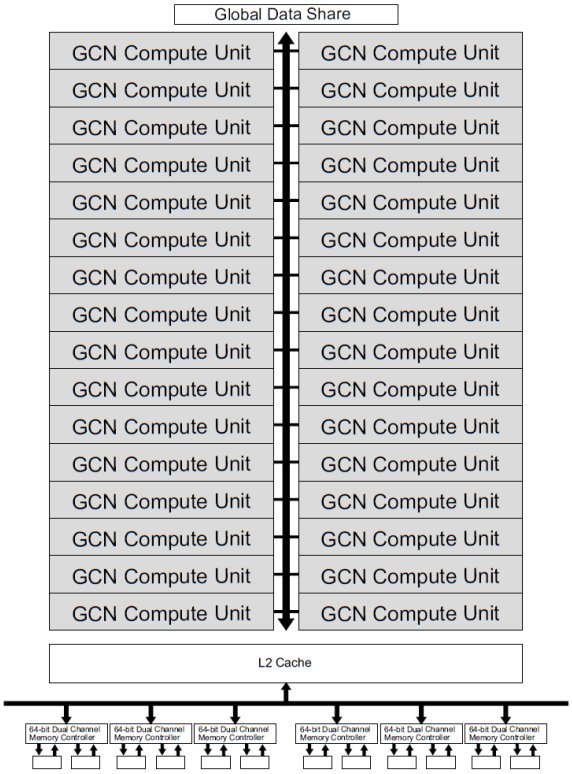

Fig. 1. AMD Radeon HD 7970 Block Diagram [13].

A. GPGPU Architectures

This section presents the HW architecture of three different GPU architectures that can be used for data parallel computing implemented in the DirectX 11 Compute Shader:

1) AMD Radeon HD 7970 Graphics
2) NVIDIA Geforce GTX780
3) Intel HD Graphics 4000

It is not the purpose of this section to compare the performance between these three architectures; instead, we will focus on providing the HW and SW details to implement a portable fast radix sort implementation across these architectures.

1) AMD Radeon HD 7970 Graphics: The AMD Radeon HD 7970 is based on the AMD's Graphics Core Next (GCN) architecture [13]. This video card has 32 compute units (CU). Each CU has 4 SIMD units for vector processing and each SIMD unit is assigned its own 40-bit program counter and instruction buffer for 10 wavefronts and it can execute up to 40 wavefronts. A wavefront groups 64 threads running in parallel. Thus the AMD Radeon HD 7970 video card can issue up to 81,920 work items at a time in 2048 stream processors. Figure 1 shows a block diagram of the Radeon HD 7970 architecture.

2) NVIDIA Geforce GTX780: The NVIDIA Geforce GTX780 video card is based on NVIDIA's Kepler architecture [14] and consists of twelve next-generation Streaming Multiprocessors (SMX) with 192 CUDA cores per SMX. Thus the GeForce GTX 780 implementation has 2304 CUDA cores.

Figure 2 shows a block diagram of the GeForce GTX 780 architecture [14].

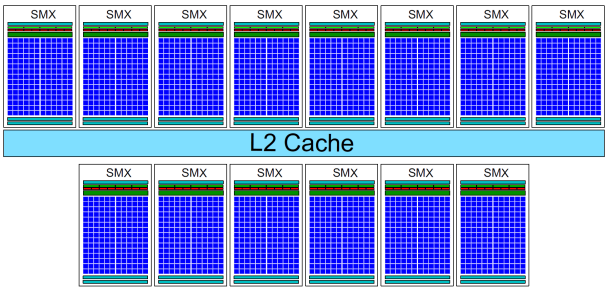

Fig. 2. NVIDIA GeForce GTX 780 Block Diagram [14].

The SMX schedules threads in groups of 32 parallel threads called warps. Each SMX features four warp schedulers and eight instruction dispatch units, allowing four warps to be issued and executed concurrently and two independent instructions per warp can be dispatched on each cycle.

3) Intel HD Graphics 4000: Intel HD Graphics 4000 is a GPU that is integrated in the 3rd generation Intel Core processor [15]. This GPU contains 16 execution units (EU) with 8 threads/EU. Figure 3 shows a block diagram of the Intel Processor Graphics architecture.

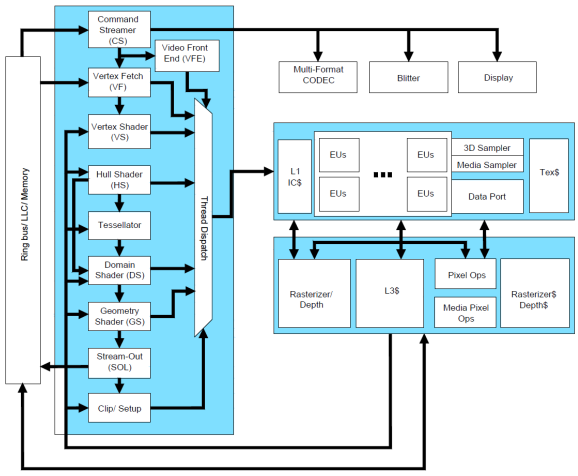

Fig. 3. Overview of the Intel Processor Graphics [15].

B. General-Purpose Programing in GPUs

The main application programing interface (API) environments used to develop general-purpose applications on GPUs are the NVIDIA's CUDA C [16], Microsoft DirectX 11 DirectCompute [17] and OpenCL [18]. These APIs provide a C-like syntax programing environment with tools to easily build and debug complex applications [19].

Unlike DirectCompute and OpenCL, CUDA-enabled GPUs are only available from NVIDIA architecture. We will use DirectCompute for our radix sort implementation in order to enable the execution across different hardware architectures. The DirectCompute is a programmable shader stage that expands Microsoft's Direct3D 11 beyond graphics programming and enable the GPU processing units for general purpose programing. This programmable shader is designed and implemented with HLSL [20]. HLSL abstracts the capabilities of the underlying GPU HW architecture and allows the programmer to write GPU programs in a HW-agnostic way with a more familiar C-like programming language [21].

1) GPGPU programing Model: The programing model used for general purpose computing in GPUs is Single Program Multiple Data (SPMD) which means that all threads execute the same code. Today, GPU's can process hundreds of threads in parallel form, a thread is also called "work item"or element. Threads are grouped into blocks that can be seen as arrays of threads. The blocks are grouped to form a grid of threads in which, each thread has an ID that can be use to calculate its position within a group or the grid by using a 3D vector (x,y,z).

2) GPGPU Memory Model: The understanding of the cost and bottlenecks of the memory access can help improve the performance of a program running in a GPU [22]. The threads in a GPU application perform to the following memory requests:

- Private Memory
  - Fastest access
  - Visible per thread
  - Thread lifetime
- Shared Memory
  - Shared across threads within a Group
  - Very low access latency
  - Block lifetime
- Global Memory
  - Accessible by all threads as well as host (CPU)
  - High access latency and finite bandwidth
  - Program lifetime
- Constant Memory
  - Short access latency
  - Read only
  - Kernel/Dispatch execution lifetime

Based on the memory access characteristics, we can design a strategy to maximize the memory bandwidth utilization, minimize the access latency and improve the performance in memory request such as, in the case of scatter and gather which are one of the most frequent operations in the implementation of sorting algorithms and suffers from low utilization of the memory bandwidth and consequently long memory latency.

IV. Radix Sort

Radix sort is a sorting algorithm that rearranges individual components of the elements to be sorted (called keys) represented in a base-R notation. In high data-parallel computing architectures, it is very efficient to implement radix sort algorithms using integers that can be decomposed in keys represented as binary numbers with $R = 2$ or a power of two $R = 2^s$. It also helps if the keys are aligned with integer boundaries that match the size of the GPU registers. However, this algorithm is not limited to sorting integers, it can be used for sorting any kind of keys including string and floating point values.

There are two different approaches to implement a radix sort algorithm:

1) Most significant digit (MSD) radix sorts and
2) Least significant digit (LSD) radix sorts

The first group examines the digits of the keys in a left-to-right order, working with the most significant digits first. MSD radix sorts process the minimum amount of information necessary to get a sorting job done. The LSD radix sort examines the digits in a right-to-left order, working with the least significant digits first. This approach could spend processing time on digits that cannot affect the result, but it is easy to mitigate this problem and the latter group is the method of choice for most of the sorting applications.

To illustrate how an LSD radix algorithm with $R = 10$ works, consider the input array of 2-digit values shown in Figure 4. The sorting algorithm consists of 2 passes extracting and rearranging the i-th digits of each key in order from least to most significant digit. Each pass uses R counting buckets to store the digits based on the individual values from 0 to $R - 1$ to compute the destination index at which the key should be written. The destination index is calculated by counting the numbers of elements in the lower counting buckets plus the index of the element in the current bucket. Having computed the destination index of each element, the elements are scattered into the output array in the location determined by their destination index.

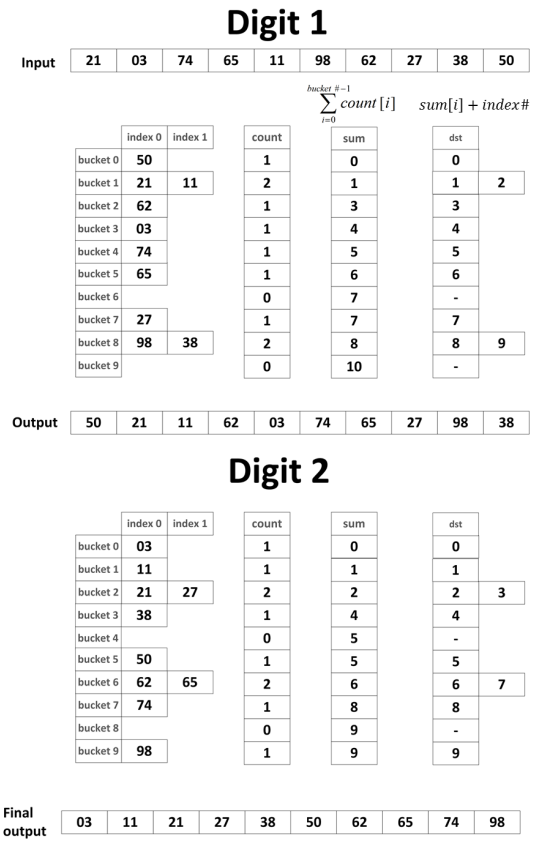

Fig. 4. LSD radix R=10 sort diagram.

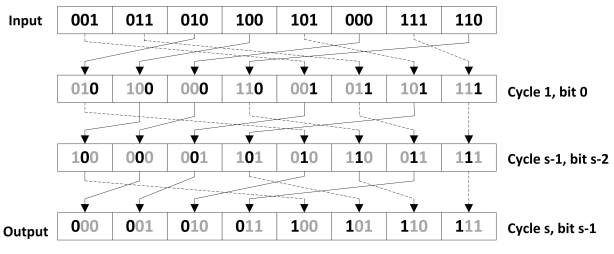

Fig. 5. Radix Sort $R = 2^3$.

A. Radix Sort implementation in DirectX 11

The radix algorithm implementation in the DirectX 11 Compute Shader sorts an input array of 32-bit numbers decomposed in 4-bit keys of integers in a radix base of $R = 2^4$. This requires 16 counting buckets in each pass. In order to parallelize the algorithm in the GPGPU, the input array is divided into blocks that can be processed independently in a processing core. This approach was described by [11] and [23].

The algorithm sorts an input array of 32-bit numbers for the least significant 4-bit digit in 8 cycles. Each cycle uses three dispatches:

1) Each block sorts in local shared memory the 4-bit keys according to the i-th bit using the split primitive described in [8] and [11] and compute offsets for the 16 buckets
2) Perform a prefix sum over the global buckets
3) Compute the destination index into the global array

After the 8 cycles, the output array is transferred to external memory back to the host system.

1) Local Sort Dispatch: The local sort dispatch performs a scan operation in an input array of 4-bit keys. The array is divided in blocks. Each block uses 512 threads and each thread processes one 4-bit key in LSD order. The process uses a local scan in shared memory. At a high level, the local sort dispatch is a radix $R = 2$ sort algorithm that is performed in each block, scanning one i-th bit from each key from least to most significant bit, then the keys are split placing all the keys with a 0 in that digit before all keys with a 1 in that digit . This process is repeated s times to sort the input list of N keys with respect to the i-th bit extracted in each cycle, in order from least to most significant as shown in Figure 5.

In the local sort step, each thread processes one element of the input array, extracting a bit of the key to be passed as the argument "pred"to the split operation. The scanBlock function performs a scan operation to obtain the total number of true predicates and the number of threads with lower groupIndex that have a true value in pred. With these values we can calculate the output position of the key for the current i-th bit-digit value (see Figure 6). The input array of 4-bit keys will be sorted after four successive split and scatter operations in local shared memory.

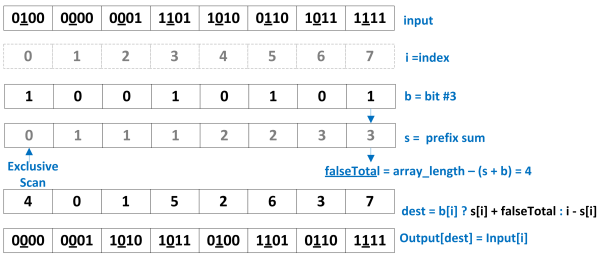

Fig. 6. Split primitive block diagram.

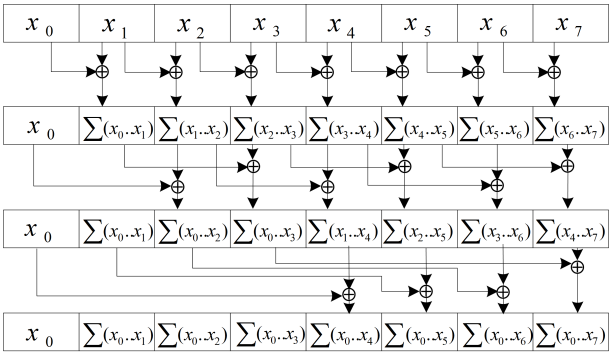

Fig. 7. Memory Access Pattern of the Parallel Scan Operation.

The scan is a common operation for data-parallel computation, it is best known as the "all prefix-sum"operation. Blelloch [8] describes the scan as a primitive operation that can be executed in a fixed unit of time. The exclusive scan primitive takes a binary operator $\oplus$ with identity i, and an ordered array $[a0, a1, ..., an–1]$ of n elements, and returns the ordered array $[i, a0, (a0 \oplus a1), ..., (a0 \oplus a1 \oplus ... \oplus an–2)]$. An "inclusive scan"version of this algorithm also returns an ordered array $[a0, (a0 \oplus a1), ..., (a0 \oplus a1 \oplus ... \oplus an–1)]$.

For example, applying the operator $\oplus$ as an addition to the following array:

$$A = [\ 3 \quad 11 \quad 21 \quad 27 \quad 38 \quad 50 \quad 62\ ]$$

The exclusive scan would return:
$$scan(A, +, excl) = [\ 0 \quad 3 \quad 14 \quad 35 \quad 62 \quad 100 \quad 150\ ]$$

And the inclusive scan output would be:
$$scan(A, +, incl) = [\ 3 \quad 14 \quad 35 \quad 62 \quad 100 \quad 150 \quad 212\ ]$$

In the case of the exclusive scan, the first value is the identity element for the operator that in this case is 0.

Sengupta et al. [10] presented an efficient parallel scan algorithm optimized to take advantage for the CUDA programming model and wrap granularity of the NVIDIA hardware architecture in order to maximize the execution efficiency. By using this approach the threads executed within a block can be organized in groups to shared memory and take advantage of the synchronous execution of threads in a warp to eliminate the need for barrier synchronization.

The algorithm to perform a scan operation with segmented warps/wavefronts of k threads implements the parallel scan algorithm presented by Hillis [7]. Figure 7 depicts the memory access pattern to implement this method. This function has the ability to select either an exclusive or inclusive scan operation via the scanType parameter. This algorithm applies the operator $\bigoplus$, across a block of size b = (THREADS_PER_BLOCK), $O(k \log_2 k)$ times for a warp/wavefront of size k.

Using this Scan primitive we can build a function to scan all the elements in a block arranging all the threads in groups of k threads. The Scan Block function implementation is shown in the Block diagram of Figure 8. This ScanBlock function does $O(b \log_2 k)$ work for a block of size b and a multiple of k containing $\frac{b}{k}$ warps/wavefronts, and each thread processes one value of the block. Then the LocalSort dispatch for a global

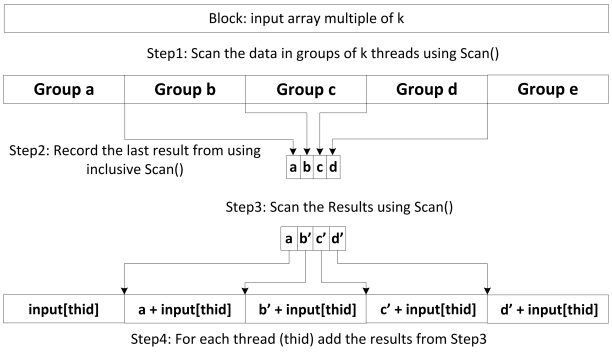

Fig. 8. Constructing a ScanBlock() function.

array of size n will have $\frac{n}{b}$ blocks with a work complexity of $O(n \log_2 k)$.

The offsets function stores the bucket offsets corresponding to the initial location in a block for each radix digit and computes the number of keys in each of the 2^s buckets by counting the sorted digits for each block based on the individual values from 0 to $2^s - 1$. This counting is easily obtained by calculating the difference between adjacent bucket offsets. These values are written into a global buckets matrix that stores the counting values per block.

2) Prefix Sum Over the Global Buckets Matrix: The bucket sizes are obtained by performing a prefix sum dispatch on the buckets matrix. The prefix sum can be done by applying the scan function explained in the LocalSort dispatch section.

3) Compute the Global Destination index: The global scatter dispatch computes the global position of the number locally sorted at its block level with respect to the 4-bit key being processed by adding its local offset in the block to the offset in its bucket.

V. Results

This section analyzes the performance of the Radix Sort implementation in the DirectX 11 Compute Shader running our algorithm in an AMD Radeon HD 7970 and NVIDIA Geforce GTX780 GPU; and compares the execution times to the Radix Sort implementation in the Thrust CUDA library running on the NVIDIA Geforce GTX780. The results obtained in the Intel HD Graphics 4000 GPU are not included in the

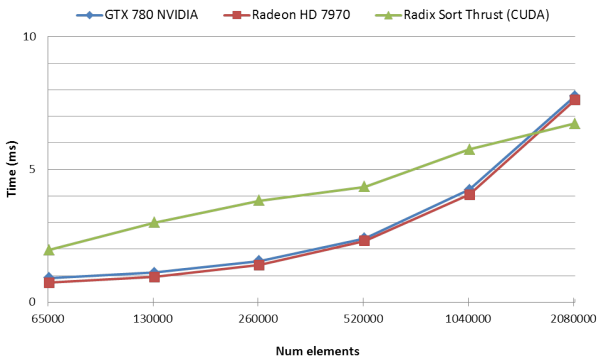

Fig. 9. DirectX 11 Compute Shader Radix Sort Performance in AMD Radeon HD 7970 and NVIDIA Geforce GTX780 GPU's compared with Thrust CUDA radix sort running in the NVIDIA Geforce GTX780 GPU.

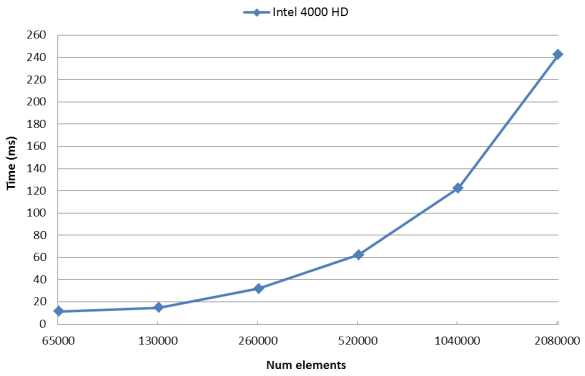

Fig. 10. DirectX 11 Compute Shader Radix Sort Performance in the Intel HD Graphics 4000 GPU.

comparison because it is integrated in a mobile platform. The NVIDA Geforce and AMD Radeon GPU's used are add-in cards for Desktop and workstation platforms where physical space and power consumption are not a constraing allowing higher graphics performance.

The timings were performed with varying number of inputs ranging from 65000 to 16 million elements. The running time measured in milliseconds is averaged over 100 runs, with the inputs redistributed randomly for each run. The running times do not include the data transfer time between CPU and GPU.

As seen in the Figure 9, the radix sort implementation in the Thrust CUDA library gives the best performance and the compute shader implementations running in AMD Radeon HD 7970 gives competitive performance for 2 million elements. With bigger arrays the Thrust CUDA implementation is 1.7× to 2× faster.

It is clear that the radix sort implementation in the Thrust library [4] is highly optimized for CUDA and also applies dynamic optimizations to improve the sorting performance and increase the speed by 2×, which is consistent with the results showed.

Figure 10 shows the execution times on the Intel HD Graphics 4000 GPU with the intention of demonstrating that the algorithm can be executed on any GPU with support for DirectX 11 Compute Shader.

VI. Radix Sort Applied in Ray Tracing

Ray tracing is an advanced illumination technique that simulates the effects of light by tracing rays through the screen on an image plane [24]. However, the computational cost is so high that ray tracing has mostly been used for offline rendering [25]. Nevertheless, real-time ray tracing applications are available today due to constant hardware improvements and algorithmic advances, which yield more efficient algorithms.

Ray tracing is traditionally known to be a trivially-parallel implementation problem given that the color computation for each pixel is independent from its neighbors. On the other hand, both CPUs and GPUs are becoming more powerful year after year; but they are intrinsically different architectures, thus, it is also necessary to have a proper understanding of the interaction between algorithms and hardware in order to achieve a good performance. In any case, memory-handling remains a constant bottleneck [26], and it is one of the key points needed to achieve high frame rates.

To solve the memory bottleneck, efficient data structures are needed. These structures should allow fast memory access and fast ray-geometry intersection discovery. Currently, it is common to use acceleration structures to avoid unnecessary calculations during ray traversal. The most used acceleration structures are kd-trees, Bounding Volume Hierarchies (BVHs) and grids, and their variants [27]. At the core of the acceleration structures based on trees lies the implementation of a sorting algorithm in order to perform a fast tree construction. For dynamic scenes it is usual to rebuild the acceleration structures from scratch in each frame, therefore a fast sorting algorithm is of paramount importance. In most cases, the sorting algorithm used is the radix sort.

A stackless Linear BVH (SLBVH) implementation presented in [28] uses a bitonic algorithm to sort the primitives of the models in order to accelerate the construction of the tree structure. That implementation allows the building of models as big as 2^{18} (262,144) primitives. The use of a radix sort can allow the building of bigger models from scratch on every frame.

Figure 11 shows a ray tracing application [29] executing the Stanford Bunny (69,451 primitives), Stanford Dragon (871,414 primitives) and Welsh Dragon (2,097,152 primitives) models rendering in real time using the SLBVH in AMD Radeon HD 7970 GPU. Our radix sort was used to accelerate the construction of the SLBVH tree. The application is using gloss mapping, Phong shading, one ray for shadows with one light source. Eight cameras with different positions and directions were set-up to measure the frame rate of the scenes.

The whole tree hierarchy is rebuilt in each frame, thus allowing real-time ray tracing with dynamic scenes.

Table I presents the construction times of the SLBVH using our radix sort algorithm compared with a BVH-SAH CPU-based implementation based on the BVH of the PBRT framework [30]. The BVH construction was executed on a 3.19GHz Intel Core i7 CPU with 6Gb of DDR3 RAM compiled as a

Fig. 11. Bunny (69K primitives, 77 FPS), Stanford Dragon (871K primitives, 23 FPS) and Welsh Dragon (2M primitives, 18 FPS) ray traced in real time with full tree reconstruction on each frame using our radix sort algorithm.

TABLE I. ACCELERATION STRUCTURE CONSTRUCTION TIMES

Model	Structure	Construction
Bunny(69K)	BVH CPU	0.142 sec
	SLBVH GPU	0.013 sec
Dragon(871K)	BVH CPU	2.091 sec
	SLBVH GPU	0.043 sec
Welsh-Dragon(2.2M)	BVH CPU	5.161 sec
	SLBVH GPU	0.083 sec

32-bit application. The SLBVH GPU construction is using our radix sort implementation in AMD Radeon HD 7970 GPU. The results showed a very high advantage of the data parallel construction in GPU, this is a key factor in the reconstruction of acceleration structures for real-time ray tracing.

VII. CONCLUSION

A fast data parallel Radix Sort Implementation in the DirectX 11 Compute Shader was presented. As far as the authors know, at the time of writing this paper, it is the fastest implementation published for the Compute Shader. The algorithm implemented several optimization techniques to take advantage of the HW architecture for the AMD, NVDIA and Intel GPUs, such as: taking advantage of kernel fusion strategy, the synchronous execution of threads in a warp/waveform to eliminate the need for barrier synchronization, using shared memory across threads within a group, management of bank conflicts, eliminate divergence by avoiding branch conditions and complete unrolling of loops, use of adequate group/thread dimensions to increase HW occupancy and application of highly data-parallel algorithms to accelerate the scan operations.

Although the results showed that our radix sort implementation gives competitive performance in AMD Radeon HD7970 and NVIDIA Geforce GTX780 GPUs sorting 2 million elements as fast as the Thrust CUDA implementation, our code still has room for improvement like the implementation of dynamic optimizations described in [5] and [4] and the optimization of the gather and scatter operations extensively used in the scan operations of the local sort dispatch and the global scatter dispatch. In [31], He et al. presented a probabilistic model to estimate the memory performance of scatter and gather operations and proposes optimization techniques to improve the bandwidth utilization of these two operations by improving the memory locality in data accesses yielding 2-4$\times$ improvement on the GPU bandwidth utilization and 30-50% improvement in response time. Once these optimizations are in place, we expect to have an even better performance. However, this is the first implementation (that the authors know of) that is competitive against the Thrust CUDA library; with the additional advantage that it can run on any Compute Shader-compliant computing platform.

Finally, the algorithm was also used in a ray tracing application demonstrating its efficiency in the reconstruction of acceleration structures for real-time ray tracing, allowing the rendering of dynamic scenes at interactive frame-rates.

ACKNOWLEDGMENTS

The authors would like to thank the Stanford Computer Graphics Laboratory for the Happy Buddha, and the Stanford Bunny models [32]; and the Bangor University for the Welsh Dragon model [33].

REFERENCES

[1] K. E. Batcher, "Sorting networks and their applications." in *AFIPS Spring Joint Computing Conference*, ser. AFIPS Conference Proceedings, vol. 32. Thomson Book Company, Washington D.C., 1968, pp. 307–314.

[2] M. Zagha and G. E. Blelloch, "Radix sort for vector multiprocessors," in *In Proceedings Supercomputing '91*, 1991, pp. 712–721.

[3] Intel. (2012) Product brief the intel xeon phi product family. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/high-performance-xeon-phi-coprocessor-brief.pdf

[4] N. Bell and J. Hoberock, *A Productivity-Oriented Library for CUDA*. Morgan Kaufmann, 2012, ch. 26.

[5] D. Merrill and A. S. Grimshaw, "High performance and scalable radix sorting: a case study of implementing dynamic parallelism for gpu computing." *Parallel Processing Letters*, vol. 21, pp. 245–272, 2011.

[6] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[7] W. D. Hillis and G. L. Steele, Jr., "Data parallel algorithms," *Commun. ACM*, vol. 29, no. 12, pp. 1170–1183, 1986. [Online]. Available: http://doi.acm.org/10.1145/7902.7903

[8] G. E. Blelloch, *Vector models for data-parallel computing*. Cambridge, MA, USA: MIT Press, 1990.

[9] M. Harris, S. Sengupta, and J. D. Owens, *Parallel Prefix Sum (Scan) with CUDA*. Addison Wesley, 2007, ch. 19, pp. 851–876. [Online]. Available: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html

[10] S. Sengupta, M. Harris, M. Garland, and J. D. Owens, *Efficient Parallel Scan Algorithms for Many-core GPUs*. Taylor & Francis, 2011, ch. 19, pp. 413–442. [Online]. Available: http://www.taylorandfrancis.com/books/details/9781439825365/

[11] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore gpus," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, ser. IPDPS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2009.5161005

[12] E. Young. (2010) Directcompute optimizations and best practices. [Online]. Available: http://www.nvidia.com/content/GTC-2010/pdfs/2260_GTC2010.pdf

[13] AMD. (2012) Amd graphics cores next (gcn) architecture. [Online]. Available: http://www.amd.com/us/Documents/GCN_Architecture_whitepaper.pdf

[14] NVIDA. (2014) Nvidia geforce gtx 780 specifications. [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-780/specifications

[15] Intel. (2012) Directx developers guide for intel processor graphics. [Online]. Available: http://software.intel.com/m/d/4/1/d/8/Ivy_Bridge_Guide2.pdf

[16] NVIDA. (2013) Cuda programming guide. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[17] Microsoft. (2013) Compute shader overview. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/ff476331(v=vs.85).aspx

[18] Khronos. (2013) The open standard for parallel programming of heterogeneous systems. [Online]. Available: https://www.khronos.org/opencl/

[19] N. Kinayman, "Parallel programming with gpus," *Microwave Magazine*, vol. 14, no. 4, pp. 102–115, 2013.

[20] Microsoft. (2013) Hlsl. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/bb509561(v=vs.85).aspx

[21] D. Feinstein, *Hlsl Development Cookbook*. Packt Publishing, Limited, 2013. [Online]. Available: http://books.google.com/books?id=yhC1mgEACAAJ

[22] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 152–163. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555775

[23] L. Ha, J. Krueger, and C. Silva, "Fast 4-way parallel radix sorting on gpus," 2009. [Online]. Available: http://www.sci.utah.edu/publications/ha09/Ha_CGF2009.pdf

[24] I. Wald and P. Slusallek, "State-of-the-Art in Interactive Ray Tracing," in *Eurographics State of the Art Reports*, 2001, pp. 21–42.

[25] I. Wald, P. Slusallek, C. Benthin, and M. Wagner, "Interactive rendering with coherent ray tracing," in *Computer Graphics Forum*, 2001, pp. 153–164.

[26] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley, "State of the art in ray tracing animated scenes," in *Computer Graphics Forum*, 2009.

[27] N. Thrane and L. Simonsen, *A Comparison of Acceleration Structures for GPU Assisted Ray Tracing*. Aarhus Universitet, Datalogisk Institut, 2005. [Online]. Available: http://books.google.com/books?id=lKEIcgAACAAJ

[28] S. Murguia, F. Avila, L. Reyes, and A. Garcia, *Bit-trail Traversal for Stackless LBVH on DirectCompute*, 1st ed. CRC Press, 2013, pp. 319–335. [Online]. Available: http://www.amazon.com/GPU-Pro-Advanced-Rendering-Techniques/dp/1466567430#reader_1466567430

[29] A. Garcia, F. Avila, S. Murguia, and L. Reyes, *Interactive Ray Tracing Using DirectX11 on the Compute Shader*, 1st ed. A K Peters/CRC Press, 2012, pp. 353–376. [Online]. Available: http://www.amazon.com/GPU-PRO-Advanced-Rendering-Techniques/dp/1439887829/ref=sr_1_1?ie=UTF8&qid=1336734285&sr=8-1

[30] M. Pharr and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[31] B. He, N. K. Govindaraju, Q. Luo, and B. Smith, "Efficient gather and scatter operations on graphics processors," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, ser. SC '07. New York, NY, USA: ACM, 2007, pp. 46:1–46:12. [Online]. Available: http://doi.acm.org/10.1145/1362622.1362684

[32] S. University, "The stanford 3d scanning repository," 2013. [Online]. Available: http://www.graphics.stanford.edu/data/3Dscanrep/

[33] B. University, "Eg 2011 welsh dragon," 2011. [Online]. Available: http://eg2011.bangor.ac.uk/dragon/Welsh-Dragon.html