GPU-based Sorting in PostgreSQL

Naju Mancheril
School of Computer Science
Carnegie Mellon University
naju@cmu.edu

Abstract

Although the number of transistors per microprocessor core has increased over the last decade, this increase in complexity has caused only a modest increase in application performance. Most performance improvements can be traced to faster clock rates from technology scaling. While complex, wide-issue, superscalar cores provide higher single-thread performance, they are often poorly suited to applications that have a large number of independent threads. In the latter scenario, the best performance is offered by a multi-processor with several simple, independent cores [1]. This paper does not seek to resolve which level of complexity is better suited for the execution of database management systems. Instead, we explore the advantages offered by a hetergeneous *processing environment.*

Specifically, we analyze the the performance improvements offered by using a GPU to aid our sorting operations. To do this, we integrate GPUSort, a fast GPU-based sorting algorithm, into the PostgreSQL database management system. We use GPUSort to implement the key-pointer sort which is used to process the SQL ORDER BY clause. We find that due to the communication overhead of setting up the GPU-based sorting, qsort() is still the faster option for sorting small relations. However, when the relation reaches even a moderate size (on the order of 1MB), GPUSort performs better.

1. Introduction

The number of transistors per microprocessor core has increased over the last decade, but this increase in complexity has caused only a modest increase in application performance. Most performance improvements can be traced to faster clock rates from technology scaling. While more complex cores provide higher single-thread performance, they are often poorly suited to applications that have a large number of independent threads. In the latter scenario, the best performance is offered by a multi-processor with several simple, independent cores [1]. This paper does not seek to resolve which level of complexity is better suited for database management systems. Instead, we explore the advantages offered by a *hetergeneous* processing environment. In particular, we execute our database management system on a general-purpose CPU, but we analyze the possible performance improvements offered by using a GPU to aid our sorting operations.

Modern GPUs (Graphics Processing Units) are structured very differently from genernal-purpose CPUs. GPUs are optimized for a set of operations (i.e. texture-mapping) that have no equivalent hardware support in a CPU. Their memory subsystems are also different since GPUs must manage a high-speed interface to system RAM in addition to the their on-board video memory. However, the single most startling distinction is in the level of hardware support for parallelism. A modern GPU may be clocked at only 400 MHz, but it can have up to 20 independent pipelines. Every clock cycle, a large number of instructions may be retired [2].

As a result of this parallelism, modern GPUS beat even high-end CPUs in terms of raw FLOPS (floating-point operations per second). Recently, there has been much work done on exploiting this processing power and using it to solve complex problems that arise in general-purpose computation [3]. One such problem is sorting. Sorting is the cause of major latencies in modern database and data mining systems. These systems are often reduced to using a CPU-based in-memory quicksort. In this paper, we show that if these systems are allowed to access a graphics processor, they can take advantage of fast parallel sorting algorithm specially suited to the capabilities of modern GPUs.

The GPUSort project at the University of North Carolina, Chapel Hill is devoted to the design and implementation of efficient GPU-based sorting algorithms. The GPUSort implementation we use maps a bitonic sorting network onto a GPU by using OpenGL texture mapping operations. The sorting algorithm also fully utilizes the graphics card's memory bandwidth by using a cache-efficient memory access pattern. Figure 1 shows GPUSort beating qsort() running on a 3.4 GHz Pentium 4 processor system using a one-generation old NVIDIA GPU [4].

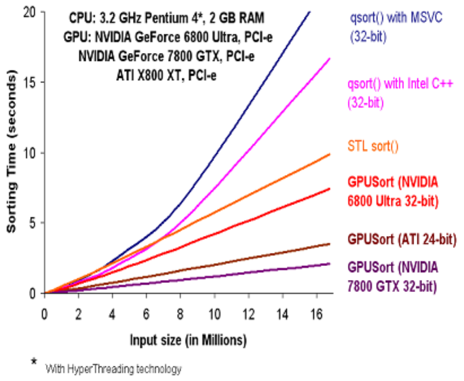


Figure 1: Raw GPUSort vs. raw qsort() [4]

We explore the improvements offered by GPUSort by integrating it with the PostgreSQL database management system. Specifically, we use it to implement the key-pointer sort which is used to process the SQL ORDER BY clause. We find that due to the communication overhead of transferring tuples to the graphics card, qsort() is still the faster option for sorting small relations. However, when the relation reaches even a moderate size (on the order of 1MB), GPUSort performs better.

The remainder of this paper is structured as follows. Section 2 gives a brief overview of bitonic sorting networks. Section 3 discusses the architecture of our modified PostgreSQL system. Section 4 contains the setup for our experimental evaluation. Section 5 contains our experimental results and analysis. Finally, Section 6 contains our conclusions and possible areas for future work.

2. Bitonic Sort

Bitonic sort (or bitonic merge sort) [5] is a fast sorting algorithm that utilizes a sorting network. A sorting network [6] is a special comparison-based parallel sorting procedure in which the sequence of comparisons is not data-dependent [7] [8] [9]. This makes sorting networks ideal functions to parallelize, and possibly even implement in hardware

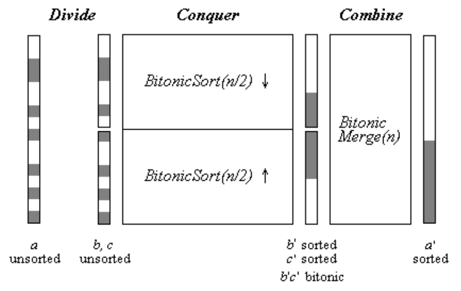
Let a sequence $a = a_0, \ldots, a_{n-1}$ with $a_i \in \{0,1\}, i = 0,...,n-1$ be called 0-1-sequence. A 0-1-sequence is called bitonic, if it contains at most two changes between 0 and 1 bits. i.e. if there exist subsequence lengths $k, m \in \{1, \ldots, n\}$ such that

$a_0, \ldots, a_{k-1} = 0,\ a_k, \ldots, a_{m-1} = 1,\ a_m, \ldots, a_{n-1} = 0$ or
$a_0, \ldots, a_{k-1} = 1,\ a_k, \ldots, a_{m-1} = 0,\ a_m, \ldots, a_{n-1} = 1$

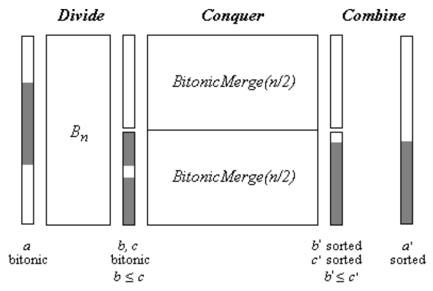
There are two main components to a bitonic sorting network: a comparator network and a merge network. These components are defined recursively. If B_n is a comparator network capable of sorting n-bit numbers, then we have

$$B_n(a) = b_0, \ldots, b_{n/2-1}\ c_0, \ldots, c_{n/2-1}$$

where $b_i \leq c_j$ for $i, j \in \{1, \ldots, n/2-1\}$ and furthermore, $b_0, \ldots, b_{n/2-1}$ and $c_0, \ldots, c_{n/2-1}$ are each bitonic. The merge network combines these two sorted bitonic sequences into one. Now sorting n bits becomes a standard divide-and-conquer algorithm consisting of a divide and sort phase, followed by a merge phase.



(a) Bitonic Sort Network



(b) Bitonic Merge Network

Figure 2: Bitonic Sort [10]

Each sort and merge iteration is called a *stage*. By using an array of n independent processors, bitonic sort lets us sort n elements in $O(\log n)$ stages using $O(\log n)$ steps for each stage. The total running time is $O(\log^2 n)$, much better than the $O(n \log n)$ time offered by quicksort [11].

GPUSort uses OpenGL texture-mapping functions to map a 32-bit floating point comparator network onto the 20 pipelines of our graphics processor. As a result, we can retire about 20 comparison instructions per 400 MHz cycle, as opposed to the 2-3 instructions retired each cycle by our 3.0GHz CPU.

3. Database Architecture

Sorting operations within PostgreSQL are implemented using the tuplesort module. Even though all database operators have an iterator-style interface (open, next, close), sorting correctly requires access to all input tuples. For this reason, the tuplesort interface asks the sort operator code to call puttuple() on every tuple from its input stream, then call performsort() to do the actual sorting, and finally, to call gettuple() on every next() invocation until every tuple has been fetched in sorted order.

The peformsort() function can work in one of two ways. For small datasets (less than 1MB in size), it constructs an in-memory array of tuple pointers, defines a comparator function to order two tuple pointers, and then calls the `libC` qsort() function. For larger datasets, it uses a external sort with a heap sort to build sorted runs.

Figure 3 shows a high-level description of the modifications we made. Once we were able to extract the floating point key from a tuple pointer, we were able to construct our own array of float-pointer pairs and pass this in-memory array directly to the GPUSort code. Upon completion, we used the sorted key-pointer array to permute our original tuple array into sorted order.

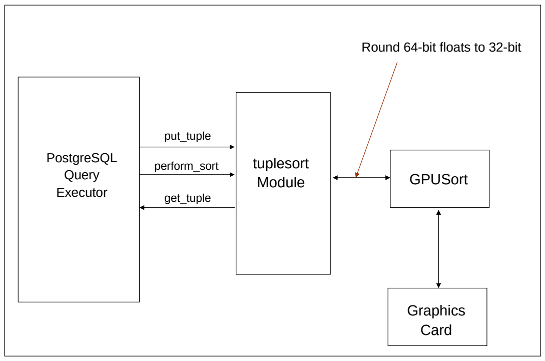


Figure 3: PostgreSQL Sorting Architecture

Even though SQL defines FLOAT to be a 4-byte floating point value, PostgreSQL internally uses a 64-bit (double) data type to represent it. This would not be a problem, except that GPUSort currently only supports 16 and 32-bit floating point numbers. For this reason, we had to round each double value to the nearest float before we could apply GPUSort. This rounding procedure does not change the set of result tuples since the tuples used in our workload did not use more than a few digits of precision. However, the rounding cost should be kept in mind when evaluating the final implementation. Any system that fully adopted GPUSort would either use 32-bit floats or acquire a version of GPUSort which supported 64-bit keys.

Another problem is that our version of GPUSort seemed to be designed specifically for 32-bit machines. So even though we passed an array of FLOAT32-pointer pairs to the sorting routine, the OpenGL code assumed that the total size of each pair was 8 bytes. We worked around this by passing in an array of FLOAT32-offset pairs: rather than the absolute address of each element, we passed in each element's offset in the tuple array. Since the offset is a 32-bit number, we can still sort an array of up to 4 billion tuples, but we hope to remove this limitation in future work.

4. Experimental Setup

4.1. Hardware

All development and evaluation was done on a 3.0GHz Intel(R) Pentium(R) 4 CPU with a 2048KB cache and HyperThreading. The system is equipped with 1GB of main memory. The graphics card used is a eVGA Geforce 7800GT with 256MB of RAM. The core GPU speed is 400 MHz and the on-board memory runs at 500 MHz. The card uses a x16 PCI-Express interface and has a maximum rated memory bandwidth of 54.4GB/sec [2].

4.2. Software

Our sorting library was GPUSort 1.0 from the GPUSort Project at University of North Carolina, Chapel Hill. We integrated it into PostgreSQL 8.2devel. All evaluation is done on Fedora Core release 4 with Linux kernel version `2.6.11-1.1369_FC4smp` optimized for `x86_64`.

4.3. Workload

We use a microbenchmark style evaluation since we only care about measuring the performance of the sorting operator. We used the following table schema for our experiments:

```
TABLE vals(x FLOAT, y FLOAT)
```

We populated this table with one million tuples generated uniformly at random between 0.0 and 100.0. All our queries are derived from the following template:

```
SELECT * FROM vals
WHERE y < y0
ORDER BY y
```

This simple query always yields the two-operator query plan shown in Figure 4.

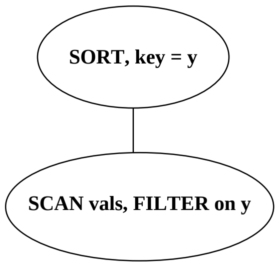


Figure 4: Microbenchmark Query Plan

Since the tuples in our base relation `vals` were generated uniformly at random, we can control the selectivity of the SCAN operator by varying the value of `y0`. For example, by setting `y0` to 10.0, we can perform a sort of $(10.0/100.0) \cdot 1$ million $= 100,000$ tuples.

In our experiments, we vary the relation size as well as the amount of video memory used by the GPUSort code on the graphic card. Another possible parameter to scale would be the tuple size. However, this does not affect the sorting cost since PostgreSQL implements a key-pointer sort. Regardless of the size of each tuple, we are only moving around a float and a word-size pointer (or in the case of GPUSort, a 32-bit offset) when we perform our comparisons and swaps.

5. Results

5.1. Large Inputs

We compared the execution times of three different sorting algorithms: the default PostgreSQL external sort, the PostgreSQL internal sort, and GPUSort. Figure 5 shows the performance of these three algorithms for large input sizes. "Heap Sort" is the PostgreSQL external sorting algorithm. It uses heap sort to create sorted runs of 1MB each and then merges sorted runs together. "qsort" denotes the internal sorting algorithm, which is really just a call to the `libC` qsort() function. For the purposes of our measurement, we increased the amount of memory used by qsort to 1GB. GPUSort 256MB shows the running time of GPUSort using a 256 MB video card.

One important point to note is that the execution time of "Heap Sort" does not monotonically increase with input size. Since the algorithm is based on divide and conquer, it will perform better when the input can be evenly divided several times. Although it is difficult to see in Figure 5, GPUSort also exhibits this behavior. This should be clear from Figure 6, which shows how GPUSort scales with larger inputs. [1]

[1]We generated a 5 million tuple dataset to measure how GPUSort scales, but all comparisons with qsort() are done with 1 million tuple base relation.

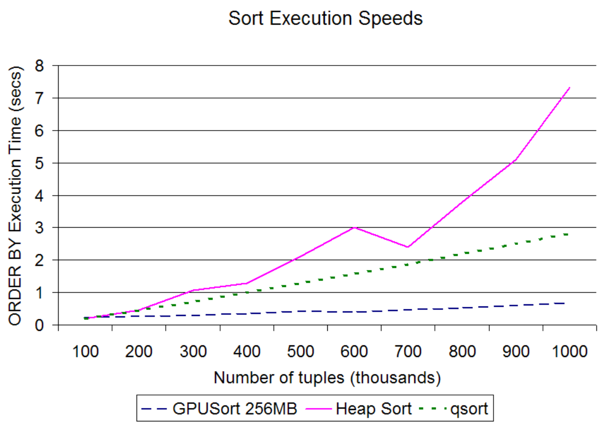


Figure 5: ORDER BY Processing Time

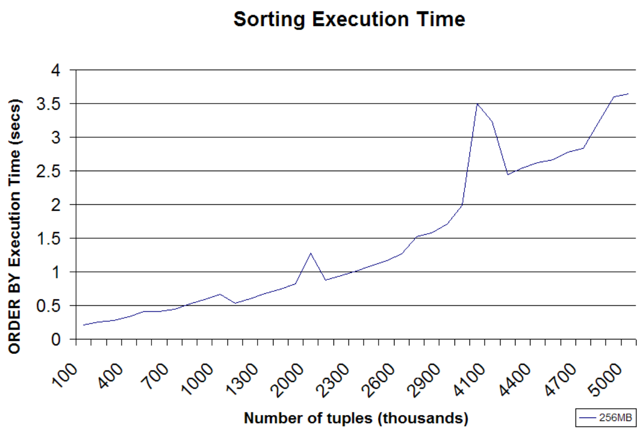


Figure 6: GPUSort ORDER BY Processing Time

We see a drop in sorting time whenever the number of tuples passes a power of 2. We see it for 500,000, 1 million, 2 million, and 4 million tuples. It takes less time to sort 4.2 million tuples than it takes to sort 4 million. This behavior is actually an artifact of how the GPUSort library is implemented. GPUSort will marshall the tuples into an internal matrix before applying its texture-mapping routines. The current implementation will not sort arrays which result in non power-of-two sized matrices completely on the GPU. Instead, it takes the largest power of two smaller than the input array size, sorts the resulting matrix on the GPU, sorts the leftover data with qsort(), and finally merges the two streams together. The code run run slower on non-power-of-two arrays because of the additional CPU usage involved [4].

The good news is that even though we may see such variations in execution time, GPUSort is always faster than qsort() when the dataset gets large. If the query optimizer had to decide between which sort to use, it would not need to specially handle these powers of 2. In fact, for large (greater than 1MB) datasets, GPUSort consistently performs up 4 times as fast as the internal quicksort.

5.2. Small Inputs

We have seen that GPUSort's performance is contingent on how much support it needs from the CPU. Even though Figure 5 shows GPUSort performing better than the CPU-based alternatives, those measurements were done with large inputs, where the execution time was dominated by the time to sort. This need not always be the case. We can break the total time used by our application to perform GPUSort into four components:

$$T_{sort} = T_{array\_build} + T_{network\_build} \\ + T_{transfer} + T_{GPU} + T_{CPU}$$

T_{sort} is the total sorting time, T_{array_build} is the time required to build a key-offset array using the CPU, $T_{network\_build}$ is the time required to build a bitonic sorting network on the GPU, $T_{transfer}$ is the time required to transfer the data to the graphics card, T_{GPU} is the time required to sort on the graphics card, and T_{CPU} is any additional time required to qsort() the remaining data and merge the two sorted streams.

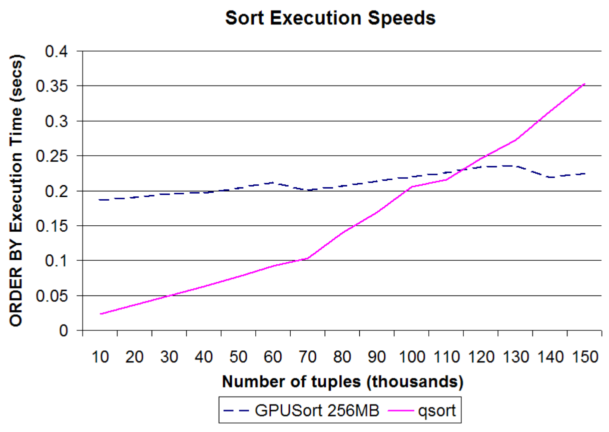


Figure 7: ORDER BY Processing Time

It is possible that the overhead of creating a separate key-pointer array, T_{array_build}, creating a sorting network, $T_{network\_build}$, and transferring tuples to the graphics card, $T_{transfer}$, may be too high for sorting small relations. Figure 7 shows this to be the case. When the number of tuples to sort is less than 100,000, qsort() is actually the faster option.

5.3. Video Memory Size

GPUSort must reserve space within the graphics card's video memory to store the matrices it will be sorting with. By varying GPUSort's esimates of the total video memory size, we control the size of the matrices it generates. We would like to see how GPUSort performs when it has a less powerful graphics processor at its disposal. Figure 8 shows the results of this scaling.

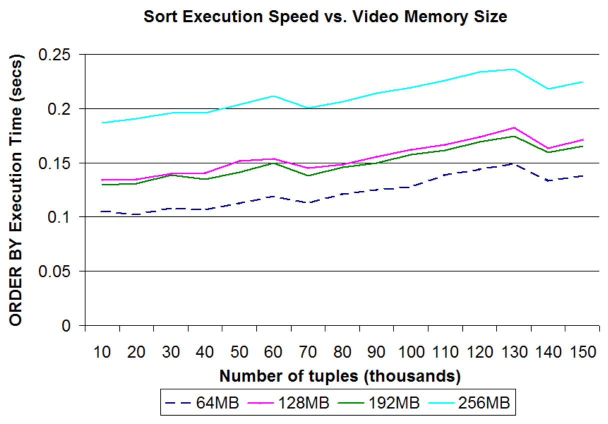


Figure 8: Varying Video Memory

The peculiar result in this graph is that the 256MB graphics card performs the worst. 256MB is the total memory capacity of the video graphics card we used. Originally, we thought that this poor performance may have been the result of thrashing between video memory and system RAM. After all, PostgreSQL must share the graphics card with XWindows and other applications which may be running. However, by running more tests, we were able to find a simpler explanation. Recall the execution time breakdown for GPUSort:

$$T_{sort} = T_{array\_build} + T_{network\_build} \\ + T_{transfer} + T_{GPU} + T_{CPU}$$

It turns out that $T_{network\_build}$ scales with the amount of video memory GPUSort will use. To build our sorting network, we must allocate memory on the graphics card, decide where to place our matrices, and pay any overhead involved with setting up future transfers between system RAM and the graphics card. It turns out that there is no need to build a new sorting network for every sort. We can "precompute" a sorting network on the card and reuse it for each sort. As a result, $T_{network\_build}$ is amortized away and we see much more reasonable execution times:

Figure 9 shows that for large data sets, GPUSort is so fast that it really doesn't matter how much video memory our

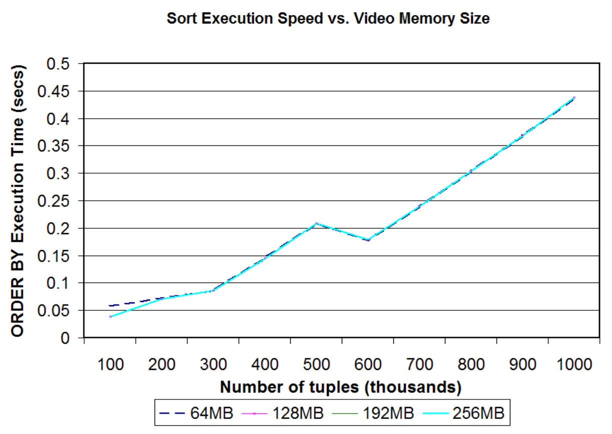


Figure 9: Varying Video Memory, Reusing Sorting Network

GPU has access to. Our performance is only constrained by the speed of the GPU and the bandwidth available to transfer the data from system RAM to video RAM. Measuring performance as a function of these parameters is the topic of future work.

6. Conclusions

This paper explored the improvements offered by GPUSort, a fast GPU-based sorting algorithm, by integrating it with the PostgreSQL database management system. Specifically, it was used to implement the key-pointer sort which is used to process the SQL ORDER BY clause. We found that due to the overhead involved with transferring tuples to the graphics card, qsort() is still the faster option for smaller relations. However, when the size of the relation becomes on the order 1 MB, GPUSort is faster.

Modern GPUs will not replace general-purpose CPUs, but their unique parallel pipeline design gives them enormous potential for improving the performance of computation and memory-bound programs. Even a 64MB graphics card (which can be purchased for less than $100) can offer large improvements over a traditional, CPU-based qsort().

The benefits offered by GPUs will only increase with the speed and number of pipelines within each card. Bus transfers are becoming less of a bottleneck with the advent of PCI Express, a technology that offers twice as many independent transfer channels as the older AGP-8x interface. These advances have led to the creation of many projects, particularly the GPGPU Project, which are dedicated solely to the design and implementation of GPU co-processors libraries for programming general-purpose applications. Although applications based on graphics libraries may be more difficult to develop, they will eventually win out: the performance boost these libraries offer will outweigh the increase in complexity.

Database systems aren't the only applications that have very different resource requirements during different phases of their execution. During "normal" execution, there is a large amount of instruction-level parallelism. For executing such a program, a wide-issue superscalar CPU offers the best performance. However, when the application needs to perform some computationally intensive operations (like sorting), the best performance is offered by a large number of independent pipelines. In the end, the best choice in core complexity may actually be "all of the above": a heterogeneous chip multi-processor with both high- and low-complexity cores that can be activated during different phases of program execution [1].

References

[1] R. Kumar, D. Tullsen, N. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *IEEE Computer*, 38, No. 11:32–38, 2005.

[2] NVIDIA Corporation. *NVIDIA GeForce 7800 GPUs Specifications*.

[3] GPGPU Project. *http://www.gpgpu.org*.

[4] GPUSort Project. *GPUSort Documentation*.

[5] K.E. Batcher. Sorting networks and their applications. volume 32, pages 307–314, 1968.

[6] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press / McGraw-Hill, Cambridge, Massachusetts, 1990.

[7] C.D. Thompson and H.T. Kung. Sorting on a mesh-connected parallel computer. *Communications of the ACM*, 20 4:263–271, 1977.

[8] H.W. Lang, M. Schimmler, H. Schmeck, and H. Schroder. Systolic sorting on a mesh-connected network. *IEEE Transactions on Computers*, C-34, 7:652–658, 1985.

[9] C.P. Schnorr and A. Shamir. An optimal sorting algorithm for mesh-connected computers. pages 255–261, 1986.

[10] Institut fur Medien Informatik und Technische Informatik. *Sequentielle und parallele Sortierverfahren: Bitonic Sort*.

[11] Randima Fernando. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Pearson Education, Boston, Massachusetts, 2003.