AN IMPROVED IMPLEMENTATION OF PARALLEL SELECTION ON GPUs

Darius Bakunas-Milanowski, Vernon Rego, Janche Sang, and Chansu Yu
1Dept. of Electrical Engineering and Computer Science, Cleveland State University, Cleveland, OH, USA
2Dept. of Computer Science, Purdue University, West Lafayette, IN, USA
Emails: d.bakunasmilanowski@vikes.csuohio.edu, {j.sang, c.yu91}@csuohio.edu, rego@cs.purdue.edu

ABSTRACT:
The computing power of current Graphical Processing Units (GPUs) has increased rapidly over the years. They offer much more computational power than recent CPUs by providing a vast number of simple, data parallel, multithreaded cores. In this paper, we proposed an improved implementation of parallel selection and compare the performance of different parallel selection algorithms on the current generation of NVIDIA GPUs. That is, given a massively large array of elements, we were interested in how we could use a GPU to efficiently select those elements that meet certain criteria and then store them into a target array for further processing. The optimization techniques used and implementation issues encountered are discussed in detail. Furthermore, the experimental results show that our advanced implementation performs an average of 2.88 times faster than Thrust, an open-source parallel algorithms library.

KEYWORDS:
Parallel Selection, CUDA, Thrust Library, GPU, Optimization Techniques, SIMT

1. Introduction

In the past few years, modern Graphics Processing Units (GPUs) have been increasingly used together with CPUs to accelerate a broad array of scientific computations in so-called heterogeneous computing [1]. It is now much more convenient to create application software that will run on current GPUs for processing massively large amounts of data, without the need to write low-level assembly language code. Furthermore, a selection of accelerated, high performance libraries allows an easy way of adding GPU-acceleration to the wide array of scientific applications. One can get even more flexibility and speed by writing his or her own GPU-accelerated programs using the CUDA Toolkit, which provides a comprehensive development environment for C and C++ developers.

NVIDIA GPUs consist of a scalable number of streaming multiprocessors (SMs), each containing a group of streaming processors (SPs) to execute the light-weighted threads, warp by warp (groups of 32 threads), using the Single Instruction, Multiple Threads (SIMT) style (term coined by NVIDIA manufacturer). In addition to the main memory on the CPU motherboard, the GPU device has its own off-chip device memory (i.e. global memory). The kernel function, which is executed on the device, is composed of a grid of threads. Note that a grid is divided into a set of blocks and each block contains multiple warps of threads. Blocks are distributed evenly to the different SMs to run. Furthermore, registers and shared memory in a SM are on-chip memory and can be accessed very fast. They are per-block resources and are not released until all the threads in the block finish execution.

Selection, also known as stream compaction or filtering, is a common programming concept that has a wide range of applications in the area of statistics, database software, artificial intelligence, image processing, and simulations [2][3]. It produces a smaller output array, containing only wanted elements from the input array made up of the mixed elements. With the tremendous amount of data elements to be processed, better performance becomes a key factor in implementing these algorithms. Therefore, exploiting the availability and the power of multiprocessors to speed up the execution is of considerable interest.

In this paper, we focused on the design and implementation of an improved parallel selection algorithm and compared its performance with other parallel selection algorithms.
methods on CUDA-enabled GPUs. All tests were performed using CUDA Toolkit on a PC with a consumer grade NVIDIA GeForce GTX 770 GPU and also on the Ohio Supercomputer Center’s newest cluster Ruby, outfitted with professional NVIDIA Tesla K40 GPUs. Both of these cards belong to the NVIDIA® Kepler [5], a cutting-edge high performance computing architecture. The empirical results show that our algorithm, which also preserve the relative order of the input elements, performs much faster than the Thrust library.

The organization of this paper is as follows. Section 2 describes related work. Section 3 goes in to details of our advanced implementation and finally, in Section 4, the experiments and the results for performance evaluation are presented. We give a short conclusion and future work in Section 5.

2. Related Work

Sequential selection is a common function and it is available in many programming languages and/or libraries. However, to implement the parallel selection, the challenging is how to determine the indices of the selected elements in the destination array. In general, the approaches to performing the stream compaction on multiprocessors can be classified into two categories: one is based on the atomic operation, while the other is based on the list ranking using the prefix-sum algorithm [6], as described below in detail.

2.1 atomic operation based approaches

In the former approach, we can use an index counter, which will be incremented by one for each newly selected element. Since many threads share the counter, the addition has to be an atomic operation. This can be done by using CUDA atomicInc() function, as illustrated in Figure 2a. Note that a CUDA atomic function performs a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory.

The main problem with this approach is that these atomic operations become a major bottleneck when the input contains a large amount of elements that pass our selection criteria. This is due to the very large number of elements competing to increment the single counter inside the global memory.

One possible improvement is to use shared memory atomics. This will essentially decrease the number of atomic collisions to a block size. Unfortunately, its performance still suffers from the thread synchronization. As demonstrated in the Experimental Results Section, execution time for both algorithms is directly proportional to the number of passing items.

A modified approach is discussed in the article "CUDA Pro Tip: Optimized Filtering with Warp-Aggregated Atomics" [7], written by researcher A. Adinetz. In addition to using the aggregated atomicAdd(), it uses the primitives __ballot(), __ffs() and __popc() (Compute Capability 2.0 and above) to perform intra-warp scan [8] and also uses the warp shuffle intrinsic [9] which is available on the Kepler and later GPUs (Compute Capability 3.0 and above) to broadcast the group index value to all of the threads within a warp. His implementation of atomic function with warp aggregation is isolated from the rest of the application, and can be used as a drop-in replacement for existing code that use CUDA atomics. Furthermore, since each warp will issue at most one atomicAdd() request, its execution time will not be proportional to the number of passing elements and hence can be reduced greatly.

Note that these three algorithms based on atomic operation do not preserve the relative order of the input elements, thus might not be suitable for certain applications.

2.2 list ranking based approaches

For the latter approach, one such implementation is provided by the Thrust library, specifically a method called copy_if() [10], which is a fairly good implementation, simple to use and may prove to be the best choice for most users needing this operation. By digging into its implementation details, we found that it uses 2-level sums. First, it calculates the number of selected elements within the block (using index counter inside the shared memory) and stores the result in an intermediate array of size $N / \text{block size}$ (N is the total number of input elements) in the global memory. Then it...
performs a parallel prefix sum (See Figure 2b) on this array and uses the outcome in the final phase to determine the indices of output elements. As a result, the relative order of the input elements is also preserved. Furthermore, as shown in the later section, Thrust algorithm execution time does not depend on the number of passing elements.

3. The Improved Implementation

Our algorithm is a list ranking based approach. Before we begin the discussion of our improved algorithm in detail, it would be beneficial to better understand two relatively new CUDA operations, namely __ballot(predicate) as shown in Figure 3 and __popc(unsigned int x). Both instructions were introduced with Compute Capability 2.0, as described. The details are as follows.

- __ballot() operation "evaluates predicate for all active threads of the warp and returns an integer whose Nth bit is set if and only if predicate evaluates to non-zero for the Nth thread of the warp and the Nth thread is active."
- __popc() operation "returns a value between 0 and 32 inclusive representing the number of set bits."

We combined these two instructions in order to eliminate synchronization bottlenecks. Furthermore, since the predicate is evaluated within the __ballot call, it removes an additional 'if' block that may potentially cause warp divergence. This occurs when threads within the warp branch to different execution paths. Since all threads in a warp must execute the same instruction at any given time, warp serially executes both branches, disabling threads that do not take the path – which can potentially result up to 50% performance loss. Warp divergence is an important concept to consider when implementing parallel algorithms on the GPUs.

During the initial implementation of this algorithm, we only allowed each thread to process a single element at a time. That is, each warp would only process 32 elements. Later, we found that increasing the number of elements processed by every warp can increase the computation granularity and hence results in overall better performance.

Our algorithm consists of three phases. In the first phase, we calculated the counter array (see Figure 4) which stores the number of valid elements for each group. For convenience, we set the group size to be 32, i.e., the same as the warp size. As mentioned earlier, each warp runs for 32 iterations processing 32 elements at a time to increase the amount of computation in relation to communication. To determine the number of passing elements inside each 32 element group, we borrowed the idea of intra-warp scan as described in [8] to utilize the combination of __ballot() and __popc() within each iteration. The result was saved into the intermediate counter[] array. Furthermore, we saved the evaluated predicate mask into pred[] array in order to avoid evaluating it for the second time during the final phase. Code listing for the phase 1 kernel is in Figure 6.

In phase 2, we applied a prefix sum operation to the counter array (Figure 5). That is, after the inclusive scan processing, there are counter[k-1] valid elements before the group k. Note that for this operation we simply used Thrust implementation thrust::inclusive_scan(), which was fast enough and sufficient for our purposes.

In the last phase, we used the calculation results from phase 2 to obtain final element indices in the output array. The code listing for the final phase is shown in Figure 8. Similarly to the phase 1, each warp runs for 32 iterations.
During each iteration, we determine an index for each sub-group of elements (see Figure 7). Then by using a pre-evaluated predicate mask, we can get the offset (i.e. stored in the variable output_element_offset) for each of the selected elements inside a group. Finally, for every passing element, its index in the final output array can be determined by adding the global_index and the variable output_element_offset together.

4. Experimental Results

In this section, we compare our improved implementation with the Thrust copy_if() method and three atomic operation based algorithms using single global counter, counter in shared memory, and Adinetz’s warp-aggregated atomics. The following experiments were conducted on one of the nodes in the Ruby cluster provided by the Ohio Supercomputer Center. The GPU used in this particular computing platform is the NVIDIA Tesla K40m, which contains 15 multiprocessors (2880 CUDA cores in total) and 12GB GDDR5 memory. A warp, the scheduling unit in CUDA, has 32 threads that perform SIMT computation on a multiprocessor. The device programs use a CUDA compiler driver 7.0.
In the first experiment, we measured the execution times for all of the algorithms by varying the number of threads per block. This allowed us to select the optimal kernel configuration for our future experiments (Figure 9). An interesting point is that performance for the global counter version does not depend on the kernel configuration, while the algorithm using shared memory could be further optimized by choosing appropriate block size – we found that 128 threads per block worked best for the Tesla K40m device. This also proved to be the case for the improved algorithm.

Next, we measured the execution performance of all five algorithms by varying the number of items being selected (Figure 10). We chose the block size of 128 threads. Note, for this experiment we used uniform random distribution for the input generator, because it gave us better control of the number of passing (filtered) elements and allowed us to measure the algorithm performance by varying the percentage \( p \). It can be seen that the atomic operation based approaches using single global counter and counter in the shared memory can perform better than the Thrust library only when the percentage \( p \) is very small because their execution times are proportional to the number of the selected elements. The running times of the Thrust `copy_if()`, Adinetz’ and our method remain almost unchanged because they are independent of the percentage \( p \) (i.e. the number of valid elements). Most importantly, our

![Figure 8: Code listing for the phase 3 kernel of the improved algorithm](image)

![Figure 9: Performance metrics for various kernel block dimensions tested (\( N = 67,108,864 \))](image)
algorithm performs much better than the Thrust library and almost the same as the Adinetz’s algorithm.

We also measured the execution times for each three phases of our improved algorithm, as illustrated in Figure 11. Phases 1 and 3 were the biggest contributors to overall performance of the algorithm, which is also why we chose not to implement the inclusive scan operation for phase 2 ourselves.

For our final experiment, we measured algorithm performance based on the input size. As Figure 12 shows, the performance of all tested algorithms was directly proportional to the number of elements while the Thrust implementation was affected to a higher degree.

5. Conclusion

Our work represents a fast implementation of a parallel selection algorithm. The experiment results are encouraging, as we were able to achieve 2.88 times better performance than what is possible using Thrust implementation.

Although our algorithm performed slightly slower than the Adinetz’s version, we believe the fact that our algorithm which preserves the order of the elements is more useful for most applications.

Through this process, our research team gained substantial experience working with modern GPUs. In the near future, we intend to implement even faster parallel algorithms by investigating the use of new warp shuffle instructions, which allow the threads within a warp to exchange data in registers, to speed up the computation in phase 1 and phase 3.
Acknowledgments

This research was supported by Cleveland State University 2015 Undergraduate Summer Research Award Program and by allocation of computing time from the Ohio Supercomputer Center.

References


