Count Sort for GPU Computing

Weidong Sun [1,2] *Zongmin Ma* [2]
1. School of Computer, Shenyang Institute of Aeronautical Engineering, Shenyang, China;
2. School of Information Science and Engineering, Northeastern University, Shenyang, China.
Email: sunweidong@syiae.edu.cn)

Abstract

Counting sort is a simple, stable and efficient sort algorithm with linear running time, which is a fundamental building block for many applications. This paper depicts the design issues of a data parallel implementation of the count sort algorithm on a commodity multiprocessor GPU using the Compute Unified Device Architecture (CUDA) platform, both from NVIDIA Corporation. The full parallel version runs much faster than any serial implementation on CPU with the loss of stability due to the limitation of the massive threads parallel model. But the thread-level parallel implementation still provides an efficient parallel sort primitive for many applications, which do not require stable sort or can be adapted for unstable subroutines.

1. Introduction

Counting sort is a simple, stable and non comparison sort with linear running time of $\Theta(n+k)$, where n and k are the length of the input position array a and the counting array c respectively. The corresponding values of a are stored in rank array r, r is used as the indices of c to count the occurrences of each rank value of a, another array b is used to hold the output position values. Counting sort is often used for processing individual digits in radix sort, and it also serves as a fundamental building block for many other applications. More specially, we will talk about the parallel implementation issues of count sort in some applications, where stable sort is not necessary, for example, in suffix array construction [1] and exact repeats finding [2].

The suffix array is a lexicographically sorted array of all suffixes of a string, which is used widely in string matching, genome analysis [3] and text compression [4]. Many suffix array construction algorithms employs count sort as a fundamental function block to sort or order the suffixes [1, 5, 6], for its simplicity and efficiency. Furthermore, parallel suffix array construction [7] has emerged as an interesting research field to meet high-performance computing requirement in full text index, data compression and bioinformatics. However, we are not aware of any data parallel version exists. Parallel implementation of count sort is the first step to conquer the problem in data parallel way. This is the subject of the present paper. We introduce data parallel model in Section 2, readers familiar with CUDA programming can simply pass this section. We describe the parallelization method and issues in Section 3 and experiment results in Section 4. Section 5 concludes with the discussion of the advantages and limitations of the data parallel implementation of the counting sort.

2. CUDA Parallel processing model

Most personal computers today are equipped with hardware for 3D graphics acceleration called Graphics Processing Units (GPUs). The GPUs provide tremendous memory bandwidth and computational horsepower not only for vertex and pixel processing pipelines but also for non-graphical, general purpose (GPGPU) applications [8]. The high-performance computing capability of GPUs in commodity now can match with the computational power of high-end servers only available in computer centers [9].

In this paper, we particularly discuss our data parallel implementation on CUDA introduced by NVIDIA [10]. CUDA is a development toolkit, supported on all NVIDIA GeForce, Quadro, and Tesla hardware architecture, allows programmers to develop GPGPU applications using the extended C programming language instead of early GPGPU platform using graphics APIs.

1521-9097/09 $26.00 © 2009 IEEE
DOI 10.1109/ICPADS.2009.30

The parallel programming model and software develop environment of CUDA are designed to transparently scale GPU parallelism while providing low learning curve and various debugging tools for C programmers. The core of CUDA consists of three rank abstractions – a hierarchy of thread groups, shared memories, and barrier synchronization-which is especially well-suited to solve problems that can be expressed as data-parallel computations – the same code is executed on many data elements in parallel. To date, CUDA has got more widespread application than other GPGPU solutions.

In CUDA, the GPU is viewed as a compute device suitable for data parallel applications. It has its own device memory and may run a huge number of parallel threads (Fig.1). Threads are grouped in blocks and blocks are further aggregated to a grid. In current CUDA threads schedule model, 3D block index and 2D thread index are supported for generating enough number of threads and adopting for application data dimension. For CUDA 1.0, the maximum number of threads per block is 512 and the maximum size of each dimension of a grid of thread blocks is 65535. These hierarchical sets of threads are executed on SIMT (single-instruction, multiple-thread) multiprocessor as Kernels, also called device code comparing to code executed on CPU (named host code), blocks of threads are mapped to virtually arbitrary number of streaming multiprocessors (SMs). The multiprocessor creates, manages, and executes the huge number concurrent threads in hardware with zero scheduling overhead comparing to CPU threads. A single instruction barrier synchronization together with lightweight thread creation and zero-overhead thread scheduling efficiently support fine-grained parallelism. Now GPU provides from several to many such SIMT multiprocessors, furthermore, the parallel unit number continues to scale with Moore's law and different design philosophy of GPU.

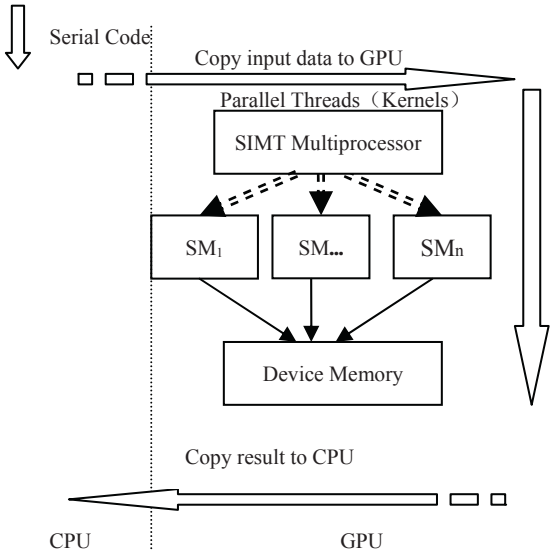

Figure 1. **Simplified CUDA thread model**

Threads may access data from various device memory spaces on GPU during their executions (Fig.2). Each thread has private registers and local memory, each thread block has shared memory visible to all threads of the block, and all threads of the grid have global memory accessible by all threads and host code. All the memory types mentioned above are random access memory, *i.e.* read/write, small amount of memory are read-only, they are constant and texture memory, both of them are cached, which provide highly efficient access for read-only data. On-chip shared memory provides much faster access rate than other types of memory which reside in device memory (DRAM).

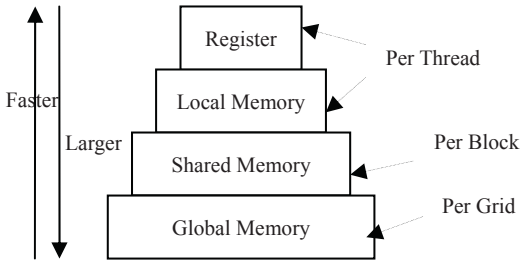

Figure 2. **Simplified CUDA memory model**

3. Parallelization method and issues

To demonstrate the parallelization process and effect, the count sort algorithm is divided into four steps, as in Table 1. The step 1 and 3 only occupy a small amount of time during the execution of the whole algorithm due to fast direct addressing mode on CPU, while the step 2 and 4 are the most time consuming operations since indirect addressing instruction is used several times and complex caching mechanism of CPU do not work well in this situation, which severely decrease the performance of the algorithm.

Table 1. **Count sort steps and corresponding proportion in serial version**

Steps	Statement	Function	Quota
1	for (int i = 0; i <= K; i++) c[i] = 0;	initialize count array	3%
2	for (int i = 0; i < n; i++) c[r[a[i]]]++;	count each rank value occurrences	18%
3	for (int i = 0, sum = 0; i <= K; i++) {int t = c[i]; c[i] = sum; sum += t;}	exclusive prefix sums	6%
4	for (int i = 0; i < n; i++) b[c[r[a[i]]]++] = a[i];	sort	73%

After the analysis of the serial algorithm, the key parallel kernel design philosophy is clear, that is to maximize the parallel threads number to hide the memory access latency. In addition, indirect addressing instructions in step 2 and 4 make fast shared memory inapplicable in the scenario, so high latency global memory is the only choice which can also be hidden by the great parallelism of the kernel code. When lots of threads increase the count array, the threads with the same rank value will conflict with each other, as show in Fig.3, so read-modify-write atomic operations is necessary, which is supported by CUDA as device runtime component [10].

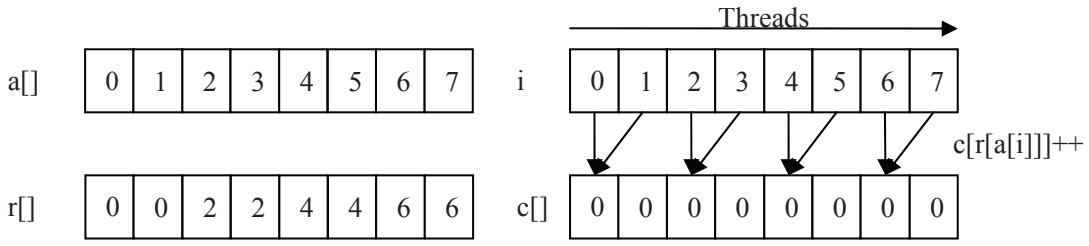

Figure 3. **Memory access conflicts**

3.1. Parallelization process and method

The step 1 is the simplest code block for parallelization, which is a typical scatter operation in GPGPU programming, *i.e.*, each thread should set one count array element zero. The set zero operation can run very fast on GPU, if we set the threads number equals the size of count array *c*.

The step 2 and 4 apply the same parallelization technique, atomic operations, since memory access conflicts occur as shown in Fig.3. In CUDA, since atomic functions are only supported when maximal active threads number is less than or equal to the product of physical multiprocessor number and maximal threads number per block, the atomic accesses of count array should be done in segmentation for large size input array. Step 2 needs one atomic add operation, but step 4 needs two atomic operations, first atomic add operation with the original value of the count array element is stored in local memory of each thread, then atomic assign the value of $a[i]$ to $b[t]$, where t is the stored private value. The atomic access of the same count array element will cause performance loss, since conflict threads have to be serialized. If the number of distinct rank values of r is very small, the overall efficiency of the parallel

algorithm will be severely played down. The last and important issue needs to be mentioned is that if duplicate rank values exists, the paralleled sort block will not to be stable, *i.e.*, the position orders of these rank values are not kept due to the scheduling undeterminism of parallel threads.

The step 3 is the *all-prefix-sums* operation that seems inherently sequential, but for which there is an efficient parallel algorithm [11]. The all-prefix-sums operation is defined as follows in [11]:

Definition: The all-prefix-sums operation takes a binary associative operator $\oplus$, and an array of n elements

$$[a_0, a_1, \ldots, a_{n-1}]$$

and returns the array

$$[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus, \ldots, \oplus a_{n-1})]$$

This type of all-prefix-sums is commonly known as *inclusive scan*, the other type of all-prefix-sums is commonly known as *exclusive scan* [11].

Definition: The exclusive scan operation takes a binary associative operator $\oplus$ with an identity element I, and an array of n elements

$$[a_0, a_1, \ldots, a_{n-1}]$$

and returns the array

$$[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus, \ldots, \oplus a_{n-2})]$$

In this paper, we employ the CUDA exclusive scan code implemented by Mark Harris [12], The scan algorithm consists of two phases: the up-sweep phase and down-sweep phase. The up-sweep phase only computing partial sums (it performs (n-1) adds) [11], the down-sweep using the partial sums and swap operations to build the exclusive scan (it performs (n-1) adds and (n-1) swap) [11]. The algorithm works fine to scan an array inside a thread block, maximal elements number is limited to 1024 due to threads number limitation per block. Fast share memory is used in both two phases for data locality in a block. For arbitrary (non-power-of-two) size array, the algorithm is first applied recursively for each scan block to generate an array of block increments, then each scan block is uniformly added by block increments[11]. This scan algorithm performs O(n) operations with great parallelism, therefore it works more efficient than serial code for large array. For small arrays, it is executed on GPU not efficient as on CPU, but it avoids the slow memory copy operation from device to host, so it is appropriate choice for our parallel count sort implementation. In addition, the scan algorithm should be called unsegmented scan more precisely, since it includes recursive host function call, *i.e.*, not all parts of the code runs on GPU. The full parallel implementation is named segmented scan [13], which is not of our interest, since it is about several times slower than unsegmented scans for large array and occupies more memory space [13].

3.2. Implementation issues

Since we define three different dimension sizes of block and thread in steps 1, 3, and 2 and 4, and step 2 and 4 are not consecutive code blocks, each step is implemented separately for clarity and efficiency. They are *zero-count-array*, *count-each-rank-value-occurrences*, *exclusive-prefix-sum* and *sort*. The count array c used in these steps only exists in device memory, only the sort result b needs to be transferred back into host memory. Though several steps form the whole parallel sort algorithm, since kernel call cost is so low that it can be omitted, it will not decrease the whole algorithm performance and makes us easy to analyze performance gain in each step.

Step 2 and 4 are the most time consuming part of the algorithm, and they both need atomic operations. The atomicity is only kept among the threads that can be scheduling by the physical GPU, *i.e.*, we must know the multiprocessor number *MN* of the GPU running the program and maximal thread number per multiprocessor *TN*, both of them can be got at run time by device management functions[14]. So the maximal thread number per iteration *ITN* can be defined as

$$ITN = MN * TN$$

For large size input data of length n, at least ITN/n iterations are executed to accomplish the two steps. Since the count sort algorithm deals with one dimension array, we only use x dimension of 2D thread index, which is set to equal *TN*. Two dimensions of 3D block index are used

to scale up the thread number for large *ITN*. The *ITN* guarantees the full utilization of the underlying GPU and atomicity of operations. The exclusive scan codes [11] are modified to make use of two dimensions of 3D block index for the same reason. The same scaling method is also applied for large size input data in step 1 to achieve highest parallelism, for step 1 is the only code block totally running on GPU.

Since stable serial sort and unstable parallel sort return different result array, we develop a tailored program to test the correctness of the two results instead of using standard CUDA comparison API. For terseness, we do not discuss it in the paper.

4. Experiment results

We have tested the serial and parallel solutions on a workstation, having the AMD Athlon 64 X2 3800+ 2.00 GHz processor and NVidia GeForce 9600GSO graphic cards. The 9600GSO has 256 MB of on-board RAM with 126 bit memory bandwidth and a G82 with 12 1.45GHz multiprocessors. At the time of paper writing, the retail price of the 9600GSO video card is about $70, and the AMD Athlon 64 X2 CPU is about $35. The machine was running Microsoft Windows XP Professional Edition SP3 with CUDA 2.0 and Microsoft Visual Studio 2005. The rank values are integers generated randomly by standard C++ pseudo-random function, the number of duplicate rank values is calculated to show the memory conflicts rate. The relative performances of count sort are firstly measured by comparing the total run time of the GPU and CPU version code with about a half memory conflicts, as shown in Table 2.

Table 2. **Comparison of experiment results**

#elements	#duplicates	CPU Time(ms)	GPU Time(ms)	Speedup
10000	4943	0. 55	0.79	0.70
50000	24957	3.63	1.73	2.10
100000	50076	10.22	2.81	3.64
500000	250083	96.76	13.75	7.04
1000000	500121	216.39	28.08	7.71
5000000	2507557	1192.85	147.54	8.08

The maximal speedup of GPU version over CPU version are about 8 times for large size of input data elements. For small size genome the GPU implementation is slower than CPU implementation, since the startup overheads of the kernel calls are relatively high and small enough data size prohibits the full utilization of GPU computing power. The number of duplicate rank value also has a strong impact on the data parallel count sort algorithm, since duplicates will reduce the parallelism of threads. For instance, the number of input data elements is 5000000 with 2507557 duplicates, if we change number of duplicate rank values to 0 and 4999999, the parallel code executing time will alter to 93.99ms and 1607.467 accordingly.

To analyze the performance gain of each step in parallel version, the time proportion of each step is given in Table 3. The exclusive prefix sum step occupies about a half of total time, which is quite different from the serial code. More efficient algorithm should be developed to remove this performance bottleneck, which forms the crucial path of the whole sort algorithm. Surprisingly the sort step executes much faster than count step, though they share almost the same quota in the parallel code. The speedup of sort step reveals that GPU can do complicated indirect memory accesses better than CPU, since GPU do not have complex caching mechanism as in CPU, which may malfunction in the complicated memory accesses. The initialization step gets the maximal acceleration due to broadcast enabled architecture of GPU, which provides high bandwidth for memory access.

Table 3. **Speedup of each parallel step**

Steps	Function	Quota	Speedup
1	initialize count array	<0.1%	>100
2	count rank value occurrences	28.1%	5.1
3	exclusive prefix sums	41.9%	1.2
4	Sort	29.9%	19.6

5. Conclusion

The count sort is a simple and powerful building block for a broad range of applications. In this paper we first depicted the efficient data parallel implementation of the algorithm on CUDA platform. The parallel code achieves a significant performance gain over the sequential implementation on CPU, which shows that various kinds of existing programs can be readily imported to the GPGPU domain with higher performance and lower cost [15].

Our implementation also reveals an intrinsic drawback of thread parallel computing model, the nondeterminism of threads execution. The parallel thread scheduling mechanism of CUDA can not guarantee the execution order of the threads, which changes the count sort algorithm from stable to unstable. But for many application without the need of stable sort, the data parallel code still serve as an efficient and off the shelf building block. In the future, we will test the parallel count sort code for solving more complex problems (suffix array construction, exact repeats finding, etc.).

References

[1] Manber U., Myers G., "Suffix arrays: A new method for on-line string searches", SIAM Journal of Computing 22(5), 1993, 935–948.

[2] Weigong Sun, Zongmin Ma, "A fast exact repeats search algorithm for genome analysis", In Proc. 9th International Conference on Hybrid Intelligent Systems(HIS-2009), IEEE Computer Society Press, 2009, 427-430.

[3] Abouelhoda M.I., Kurtz S., Ohlebusch E., "The enhanced suffix array and its applications to genome analysis", In Proc. 2nd Workshop on Algorithms in Bioinformatics, Springer, 2002, 449–463.

[4] Burrows M., Wheeler D.J., "A block-sorting lossless data compression algorithm", Technical Report. 124, Digital Systems Research Center, Palo Alto, California, 1994.

[5] Kärkkäinen J., Sanders P., "Simple linear work suffix array construction", In Proc. 30th International Conference on Automata, Languages and Programming, Springer, 2003, 943–955.

[6] Puglisi S. J., Smyth W. F., Turpin A., "A taxonomy of suffix array construction algorithms", ACM Computing Surveys 39(2), 2007, 1–31.

[7] Fabian Kulla, Peter Sanders, "Scalable parallel suffix array construction", Parallel Computing 33(9), 2007, 605-612.

[8] Owens J., Luebke D., Govidaraju N., Harris M., Kruger J., Lefohn A., Purcell T., "A survey of general-purpose computation on graphics hardware", Computer Graphics Forum 26(1), 2007, 80-113.

[9] NVIDIA Tesla Personal Supercomputer, [http://www.nvidia.com/object/personal_supercomputing.html].

[10] CUDA Programming Guide Version 2.0, Technical report, NVIDIA Corporation, 2008.

[11] Guy E. Blelloch, "Prefix sums and their applications", In John H. Reif (Ed.), Synthesis of Parallel Algorithms, Morgan Kaufmann, 1993, 35-60.

[12] Harris M., Sengupta S., Owens J. D., "Parallel prefix sum (scan) with CUDA". In GPU Gems 3, Nguyen H., (Ed.), Addison Wesley, 2007, 851-876.

[13] Sengupta S., Harris M., Yao ZH., Owens J. D., "Scan primitives for GPU computing", Graphics Hardware, 2007, 97-106.

[14] NVIDIA CUDA Compute Unified Device Architecture Reference Manual Version 2.0, Technical report, NVIDIA Corporation, 2008.

[15] Nickolls J., Buck I., Garland M., Skadron K., "Scalable parallel programming with CUDA", ACM Queue 6 (2), 2008, 40-53.