Parallel Lexicographic Names Construction with CUDA

Weidong Sun 1,2  Zongmin Ma 1

1. School of Information Science and Engineering, Northeastern University, Shenyang, China;
2. School of Computer, Shenyang Institute of Aeronautical Engineering, Shenyang, China.

Email: sunweidong@syiae.edu.cn

Abstract

Suffix array is a simpler and compact alternative to the suffix tree, lexicographic name construction is the fundamental building block in suffix array construction process. This paper depicts the design issues of first data parallel implementation of the lexicographic name construction 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. The thread level parallel code block provides an efficient primitive for building a high performance suffix array construction program and many other applications.

1. Introduction

The suffix array [1] is a lexicographically sorted array of all suffixes of a string, which is used widely in string matching, genome analysis [2] and text compression [3]. For many applications, the suffix array is a simpler and more compact alternative to the suffix tree [4, 5, 6]. Further more, 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.

Since parallel suffix tree construction is quite complicated, we are not aware of any implementation. Lexicographic names construction is one of the fundamental steps in suffix array construction process, which builds the unique rank value of each suffix according to its alphabetic order, as shown in Fig.1 a. The naïve method to build these rank values is directly sorting all the suffixes, but it is not efficient. The alternative way is building the rank values of the prefixes of the suffixes in a step by step process from the computed rank values of previous step [1]. The computed rank values are typically generated by sorting algorithms. There are plenty of parallel sorting algorithms, which is beyond the scope of this paper. The lexicographic names construction process can be depict abstractly by the C codes shown in Fig.1 b, when we only consider one rank value comparison, which shares the same mechanism as the two rank values comparison. The code block is regarded as the central element in the paper, we will refer it in other sections. We introduce data parallel model in Section 2, readers familiar with CUDA programming can simply pass this section. We describe the parallelization method 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.

### Figure 1. Lexicographic names construction

<table>
<thead>
<tr>
<th>Suffixes</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>banana</td>
<td>4</td>
</tr>
<tr>
<td>anana</td>
<td>3</td>
</tr>
<tr>
<td>nana</td>
<td>6</td>
</tr>
<tr>
<td>ana</td>
<td>2</td>
</tr>
<tr>
<td>na</td>
<td>5</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
</tr>
</tbody>
</table>

a. Unique Rank Values of Suffixes

k=0;
r_out[0] = 0;
for (int i = 1;  i < n;  i++)
    if (r_in[i] != r_in[i-1]) r_out[i] = ++k;
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 [7]. 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 [8].

In this paper, we particularly discuss our data parallel implementation on CUDA introduced by NVIDIA [9]. 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. 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 paralleled threads (Fig.2). 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 2. Simplified CUDA thread model](image_url)
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).

3. Parallelization method and issues

The code block seems inherently sequential from the first look, since the rank value $k$ is generated step by step, i.e., if the current rank value is different from the previous one, then $k$ will be incremented by 1. For example, if the input rank values are shown as $r_{in}$ in Fig.3 a, then the output rank values (lexicographic names) are numbers in $r_{out}$, which represent the different values in $r_{in}$. The current output rank value is determined by the comparison result of current and previous input rank values and previous output rank value. It should be pointed out that the data dependence on previous output rank value prohibits the data parallel execution mode.

To overcome the data dependency, we can first do the comparisons without considering the previous output rank value, which decouples the serial data dependency. The temporal comparison results are denoted by 0 and 1, i.e., 0 for duplication, 1 for distinctness. After the comparison, the intermediate results in the output rank value array are shown in Fig.3 b. The comparison phase is easy to be paralleled, since no data dependency exists in calculating the intermediate results. But the next problem is how to use the intermediate results to finish the whole lexicographic name construction, i.e., how to paralleled generate the final output rank values as same as in the sequential calculation mode, which will be explained in section 3.1.

![Figure 3](image)

**Figure 3. Sequential calculation and parallel comparison**

3.1. Parallelization process and method

If we add the intermediate results array from left to right sequentially, then we will find the two final results are identical. But these operations also seem inherently sequential. Before we introduce the full efficient parallel algorithm, we will first give some definitions and explanations to make the problem clear and easy to be solved.

**Definition:** The all-prefix-sums 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 $[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})]$. 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 $\ominus$ with an identity element $I$, and an array of $n$ elements $[a_0, a_1, \ldots, a_{n-1}]$ and returns the array $[I, a_0, (a_0 \ominus a_1), \ldots, (a_0 \ominus a_1 \ominus \ldots \ominus a_{n-2})].$

The following serial addition of intermediate results can be easily transformed to *exclusive scan*, when the
identity element $I$ holds 0. In this paper, we employ the CUDA exclusive scan code implemented by Mark Harris [11]. 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) [10], the down-sweep using the partial sums and swap operations to build the exclusive scan (it performs $(n-1)$ adds and $(n-1)$ swap) [10]. 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 [10]. 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 paralleled implementation is named segmented scan [12], which is not of our interest, since it is about several times slower than unsegmented scans for large array and occupies more memory space [12].

3.2. Implementation issues

After the analysis, the parallel version program consists of two steps, neighbor-comparison and exclusive-prefix-sum. The two steps has different dimension sizes of block and thread, each step is implemented separately, i.e., two function (kernel) calls.

The neighbor-comparison step is the most time consuming part (about 99%) in the serial version, but it is very simple to parallelize. The key parallel kernel design philosophy in this step is to maximize the parallel threads number, since high parallelism can utilize the full underlying GPU computation capacity. Each thread except 0 thread compares the corresponding input data element with the previous element, the corresponding output data element is set to 0 or 1 according to the comparison result. Using fast block shared memory can improve the comparison performance, but special mapping mechanism is required. Since shared memory is visible only within one thread block, each block shared memory need to load additional adjacent values, which will make the code lengthy and elusive. In our implementation, only global memory is used for compact code and clarity, which means the implementation is not the fastest one. In addition, using shared memory can reduce about a quarter of the running time, and our code can be readily adapted for better performance with little modification. Readers can do it by yourselves according to the application requirement.

The neighbor-comparison results are kept in device memory for the next step processing--exclusive-prefix-sum. The exclusive scan codes [11] are adopted from CUDA example projects with minor modification. The scan algorithm to scan large arrays of arbitrary size is based on recursively dividing the large array into blocks that each can be scanned by a single thread block, scanning the blocks, writing the total sum of each block to an auxiliary array of block sums, and scanning the block sums. The recursive function calls is running on host, since the device can not executes recursive code. More specially, the host code recursively calls exclusive-prefix-sum kernel codes on GPU, but no data transfer between GPU and CPU during the scan phase, only the final result need to be transferred back to host memory, which has little side-effect of the overall performance, since frequent data transfers between host and device will severely play down the parallel execution efficiency. This is one of the drawbacks of current CUDA model.
4. Experiment and 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 input rank values are integers generated randomly by standard C++ pseudo-random function. The relative performances of parallel lexicographic names construction are measured by comparing the total run time of the GPU and CPU version code, as shown in Table 1.

<table>
<thead>
<tr>
<th>#elements</th>
<th>CPU Time (ms)</th>
<th>GPU Time (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.02</td>
<td>0.52</td>
<td>0.04</td>
</tr>
<tr>
<td>32768</td>
<td>0.50</td>
<td>0.69</td>
<td>0.72</td>
</tr>
<tr>
<td>65536</td>
<td>1.14</td>
<td>0.77</td>
<td>1.48</td>
</tr>
<tr>
<td>131072</td>
<td>4.55</td>
<td>0.99</td>
<td>4.60</td>
</tr>
<tr>
<td>262144</td>
<td>9.05</td>
<td>1.47</td>
<td>6.16</td>
</tr>
<tr>
<td>524288</td>
<td>18.13</td>
<td>2.49</td>
<td>7.28</td>
</tr>
<tr>
<td>1048576</td>
<td>36.32</td>
<td>4.34</td>
<td>8.08</td>
</tr>
<tr>
<td>2097152</td>
<td>72.77</td>
<td>8.11</td>
<td>8.37</td>
</tr>
<tr>
<td>4194304</td>
<td>146.63</td>
<td>15.52</td>
<td>9.45</td>
</tr>
<tr>
<td>8388688</td>
<td>289.10</td>
<td>30.50</td>
<td>9.48</td>
</tr>
<tr>
<td>16777216</td>
<td>584.71</td>
<td>60.28</td>
<td>9.70</td>
</tr>
</tbody>
</table>

The maximal speedup of GPU version over CPU version are about 10 times for large size of input data elements. For small size input data 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 neighbor-comparison code block only occupy less than 10% running time in the whole parallel version, which means that neighbor-comparison step can be fully paralleled efficiently. While the exclusive-prefix-sum forms the critical path of the whole algorithm, the overall performance is decided by the efficiency of the scan kernel.

There are only very simple operations (comparison, addition, and assignment) in the algorithm, the GPU computing power exploited by data parallel code execution are 10 times higher than serial CPU, which exhibits the great performance potentials of GPU programming in non-numerical computing fields and better cost performance gain, keeping in mind that GPU computing power and data transfer rate are growing much faster than CPU [8].

5. Conclusion

The lexicographic names construction is a fundamental building block for suffix construction and many other applications. In this paper we first explained the efficient data parallel implementation of the algorithm on CUDA platform according to our knowledge. The parallel code achieves a significant performance gain over a sequential implementation on CPU, which shows that various kinds existing serial
programs can be readily imported to the GPGPU domain with higher performance and lower cost [13].

The lexicographic names construction can also be used as a common function for other applications, for example, generating a small range of input values for non comparison sort algorithms, if the the transfer rate between host and device has a significant boost or the fusion of CPU and GPU becomes reality. Future programming model may be the hybrid of current serial CPU and data parallel GPU execution paradigm for high performance computing.

Furthermore, our data parallel implementation of the lexicographic names construction and count sort code blocks [14] provides the possibility for parallel suffix array construction. In the future, we will try to employ these codes as the fundamental functions to solve more complex problems – exact repeats finding [15] and data parallel suffix array construction, etc., which can be used for many bioinformatic and text processing applications to meet the high computing power requirements in these areas.

References


