EFFICIENT SEQUENCE COMPARISONS USING THE EXTENDED BURROWS WHEELER TRANSFORM

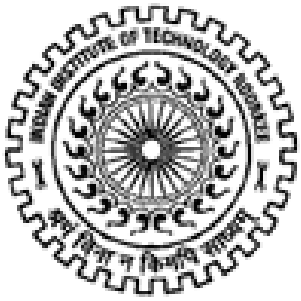
A TECHNICAL REPORT

Submitted in the partial fulfillment of the requirements of

MICROSOFT AWARD FOR SUMMER INTERNSHIP

BY

SHASHANK SRIKANT
NATIONAL INSTITUTE OF TECHNOLOGY, KURUKSHETRA



DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE - 247 667 (INDIA)
JULY, 2010

Candidate Declaration

I hereby declare that the work being presented in this technical report titled **"Efficient sequence comparisons using the Extended Burrows Wheeler Transform"** in partial fulfillment of the Microsoft Award for Summer Internship, submitted in the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, is an authentic record of my work done during my stay the period May 31 2010 to July 22 2010, carried out under Dr. Rajdeep Niyogi, Assistant Professor, Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, India.

Dated :

Place : IIT Roorkee **Shashank Srikant**

Certificate

This is to certify that the above statements made by the candidate are correct to the best of my knowledge.

Dated :

Place : IIT Roorkee

Dr Rajdeep Niyogi

Assistant Professor,
Department of Electronics and Computer Engineering
IIT Roorkee

Acknowledgement

At the outset, I would like to thank the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee for allowing me to spend quality time here and providing all the resources needed to work on good, challenging problems in the domain of High Performance Computing.

In particular, I would like to profusely thank Dr. Rajdeep Niyogi, my guide here at Roorkee, for allowing me to be a part of his team and providing the direction and the right ambience to work on a fresh problem in the world of Computer Science. His advice, insight into a problem, patience and the care he showed is something which eased my settling in here at Roorkee.

I would also like to thank Mr. Binay Kumar Pandey, Research Scholar, Department of Electronics and Computer Engineering, IIT Roorkee for his constant supervision and care in seeing to it that the least of inconvenience came my way. I would like to express my deep sense of gratitude to him for having advised me in the direction of my work, created a congenial work-environment and been ready to help me out at any given time.

I would also like to thank Dr. Navneet Gupta, In-charge, Institute Computer Centre, for allowing me to make use of the Research Scholars' lab at ICC.

I thank Stanley Seibert, Los Alamos Laboratory, United States of America and Nathan Bell and Jared Hoberock, NVIDIA Research for providing me valuable suggestions regarding programming on NVIDIA CUDA.

Lastly, I thank Achin for being the good company he was during my stay at Roorkee.

Abstract

Over the recent years, there has been an extensive development in the field of Bioinformatics. A couple amongst the various works done under this field includes Genome based phylogenetic studies. Genome based phylogenetic studies include the process of matching mitochondrial DNA of different species to establish their phylogenetic relation. One novel algorithm in order to achieve this is the Extended Burrows Wheeler Transform.

The Extended Burrows Wheeler Transform is a new development in the field of Computer Science which discusses a good strategy to compress multiple strings efficiently. However, the transform operations are compute-intensive due to the sheer size of the mitochondrial genomes used as input data. All this necessitates an optimization of such algorithms by parallelization or other means.

Our research looks at this particular application in detail, exploring the various aspects of the transformation which could help efficiently parallelize the operations in order to perform them quickly. The parallelization has been performed on Nvidia CUDA. CUDA is a parallel programming architecture which acts as a middle-ware compute engine which exposes the computational power of the NVIDIA Graphics Processing Units to software developers through industry standard programming.

The work done here analyses the working of EBWT on CUDA and describes an efficient implementation model. The implementation model described here manages a 5X speed up on some of the most compute-intensive parts of the operation. This speed-up proves credibility to the fact that this transformation technique may indeed help in the real-time realization of comparing genomic sequences. Given this credibility, new applications based on EBWT are discussed as future scope of the performed work.

Table of Content

Chapter 1: Introduction

1.1 Phylogenetic Analysis

Every living organism is made up of DNA and proteins as constituents of its cells which form the organism's basic building block. In addition to these molecular constituents defining the organism's outward appearance and biological functions, they also help biologists ascertain the related-ness or non- related-ness of two organisms. It was observed that organisms of different species that closely relate show a great deal of similarity in the molecular structure or sequence of chemical components of these biological constituents of the cell.

One such cell constituent is the mitochondrial DNA (mtDNA) which undergoes mutations over generations. The mtDNA is passed only from the maternal side, with no change except mutation. Such a comparative analysis of mtDNA help biologists to arrange various species in a tree forma with related species represented as closer branches. The simplistic means of such an analysis would be a simple string comparison between the two DNA sequences. Such algorithms generally depend upon the product of length of both sequences for their runtime.

However, there exists a novel way of having multiple strings compared for similarity, which is an application of the Extended Burrows Wheeler Transform (EBWT). This would take less time than having all combination of simple sequence comparison between each pair of sequences. The EBWT however still requires long processing times due to the sheer sizes of mtDNA data. This necessitates the reduction in runtime of the EBWT algorithm.

1.2 Multi-Core Architectures

Moore's Law had predicted that the chip manufacturing technology would be able to double the transistors on chip roughly every two years, which has stood good so far. Microprocessor technology has been using this prediction to improve its frequency by various techniques. However, in the recent past, micro-processors have hit a frequency wall, and not been able to take advantage of the predicted exponential growth. The outcome of this wall is the emergence of multi-core processors, which offer the performance benefits of multi-processors on a single chip. The presence of such architectures as common desktop processors has made it possible for hitherto time-consuming algorithms to be involved on simple desktop machines.

Another emerging trend has been the use of Graphics Processing Units (GPUs) for general purpose computing. The GPUs model themselves as multi-core processing and expect programs to take advantage of them as raw parallel number-crunchers. The multi-core processors allow programs to leverage their computing power by various means like independent threads per core or allow users to manipulate efficient data flow between cores, or provide a layer of software which manages the scalability of the cores. With the future micro-processors the trend is likely to increase the number of cores to increase computation power. Hence, it becomes important for algorithms to be parallelized to run on these next-generation micro-processors.

Thus, multi-core processors provide the perfect means of increasing the runtimes of our analysis and implementation of the EBWT applied to sequence comparisons.

1.3 Problem Statement

In our work, algorithms used in the EBWT were studied and profiled for bottlenecks. Since the problem portion of the algorithm caused its runtime to go into hours, it was required to find alternatives to be able to speed up the algorithm. This could include parallelization and modifying the algorithms used.

The objective was to parallelize the algorithms and find out if there was any improvement in the run-time performance.

1.4 Organization of the Report

This report is organized as follows:

Chapter 2 covers a detailed explanation of the CUDA programming environment which was used in this dissertation.

Chapter 3 introduces the concepts of the Extended Burrows Wheeler Tranform. It then discusses the implementation on CUDA and the issues faced therein. The results are also discussed and analyzed.

Chapter 5 concludes this report and ponders over work which might follow the results obtained.

Chapter 2: Parallel Processing Architectures

2.1 GPU

A **graphics processing unit** or **GPU** (also occasionally called **visual processing unit** or **VPU**) is a specialized processor that offloads 3D graphics rendering from the microprocessor. It is used in embedded systems, mobile phones, personal computers, workstations, and game consoles. Modern GPUs are very efficient at manipulating computer graphics, and their highly parallel structure makes them more effective than general-purpose CPUs for a range of complex algorithms. In a personal computer, a GPU can be present on a video card, or it can be on the motherboard. More than 90% of new desktop and notebook computers have integrated GPUs, which are usually far less powerful than those on a video card.

GPUs are massively multi-threaded many-core chips. They are mainly designed to handle billions of pixels, millions of polygons per second. The contain thousands of processor cores compared to the traditional CPU which contains 2, 4 or 8 cores. Scalability in these processors is very high and there exist hundreds of scalable processors. In general, they are able to run tens of thousands of concurrent threads. Some of the best computing times clocked on modern-day GPUs are close to 1 T Floating point Operations per Second (FLOPS). They implement fine grained data-parallel computation and are able to implement algorithms developed for the exotic computers.

2.2 CUDA[1][2]

NVIDIA is best known for motherboard chip sets as well as for outstanding graphics processors that have become popular as the basis for graphics cards. In the quest for maximum speed, NVIDIA's GPUs (Graphics Processing Units) have evolved far beyond single processors. Modern NVIDIA GPUs are not single processors but rather are parallel supercomputers on a chip that consist of very many, very fast processors. Contemporary NVIDIA GPUs range from 16 to 256 stream processors per card, delivering incredibly powerful computing bandwidth. The card shown above, for example, provides 256 stream processors.

Although the market impetus behind the creation of such supercomputers on a plug-in board has been the computational demands of the PC gaming market, such "graphics" boards have become so powerful that the scientific computing community has begun

using them for general purpose computing. It turns out that many mathematical computations, such as matrix multiplication and transposition, which are required for complex visual and physics simulations in games are also exactly the same computations that must be performed in a wide variety of scientific computing applications, including GIS.

NVIDIA has supported this trend by releasing the **CUDA™** (Compute Unified Device Architecture) interface library to allow applications developers to write code that can be uploaded into an NVIDIA-based card for execution by NVIDIA's massively parallel GPUs. This allows applications developers to plug in a 500 gigaflop, 256-processor, NVIDIA-based card and upload applications to run within the NVIDIA GPU at far greater speed than possible on even the fastest general purpose CPU on the motherboard. For a mere few hundred dollars we can use CUDA to achieve true, supercomputer performance on the desktop.

CUDA offers such tremendous performance gains that many functions within Manifold have been re-engineered to execute as parallel processes within CUDA if such a card is available. If we have a CUDA-capable NVIDIA graphics card installed in our system, Manifold can take advantage of the phenomenal power of massively parallel NVIDIA stream processors to execute many tasks at much greater speed.

The following section discusses some key features in the design and architecture of the NVIDIA CUDA which makes it a much better computing platform than the rest of the General Purpose GPUs available.

2.3 General Architecture

CUDA is a parallel programming model and software environment that leverages the computational horsepower of GPU (graphics processing unit) for non-graphics computing. CUDA technology was developed with several design goals in mind:

- CUDA is essentially a small set of extensions to the C programming language that enable a straightforward implementation of parallel algorithms. With CUDA, programmers can focus on the design of parallel algorithms rather than spending time on the implementation.

- CUDA also supports heterogeneous computation where applications use both the CPU and GPU. Serial portions of applications are run on the CPU, and parallel

portions are offloaded to the GPU. As such, CUDA can be incrementally applied to existing C applications. The CPU and GPU are treated as separate devices that have their own memory spaces. This configuration also allows simultaneous computation on both the CPU and GPU without contention for memory resources.

CUDA -capable GPUs have hundreds of cores that can collectively run thousands of computing threads. Each core has shared resources, including registers and memory. The on-chip shared memory allows parallel tasks running on these cores to share data without sending it over the system memory bus.

2.4 CUDA Thread Organization

Since all threads in a grid execute the same kernel function, they rely on unique coordinates to distinguish themselves from each other and to identify the appropriate portion of the data to process. These threads are organized into a two-level hierarchy using unique coordinates, called blockId and threadId, assigned to them by the CUDA runtime system. The blockId and threadId appear as built-in variables that are initialized by the run-time system and can be accessed within the kernel functions. When a thread executes the kernel function, references to the blockId and threadId variables return the appropriate values that form coordinates of the thread.

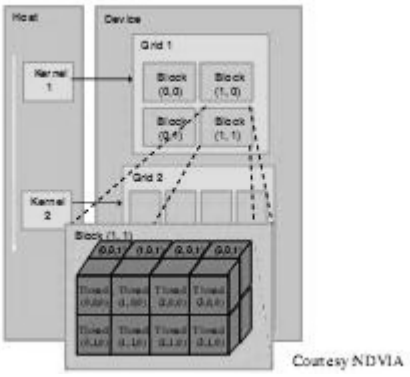


Fig 2.1- Thread Organization

At the top level of the hierarchy, a grid is organized as a two dimensional array of blocks. The number of blocks in each dimension is specified by the first special parameter given at the kernel launch. For the purpose of our discussions, we will refer to the special parameters that specify the number of blocks in each dimension as a struct variable gridDim, with gridDim.x specifying the number of blocks in the x dimension and gridDim.y the y dimension. The values of gridDim.x and gridDim.y can be anywhere between 1 and 65,536. The values of gridDim.x and gridDim.y can be supplied by run-time variables at kernel launch time. Once a kernel is launched, its dimensions cannot change in the current CUDA run-time implementation. All threads in a block share the

same blockId values. The blockId.x value ranges between 0 and gridDim.x-1 and the blockId.y value between 0 and gridDim.y-1.

2.5 Execution Model

CUDA is made up of several clusters of what Nvidia calls Texture Processor Clusters. An 8800GTX, for example, has eight clusters, an 8800GTS six, and so on. Each cluster, in fact, is made up of a texture unit and two streaming multiprocessors.

These processors consist of a front end that reads/decodes and launches instructions and a backend made up of a group of eight calculating units and two SFUs (Super Function Units) where the instructions are executed in SIMD fashion: The same instruction is applied to all the threads in the warp.

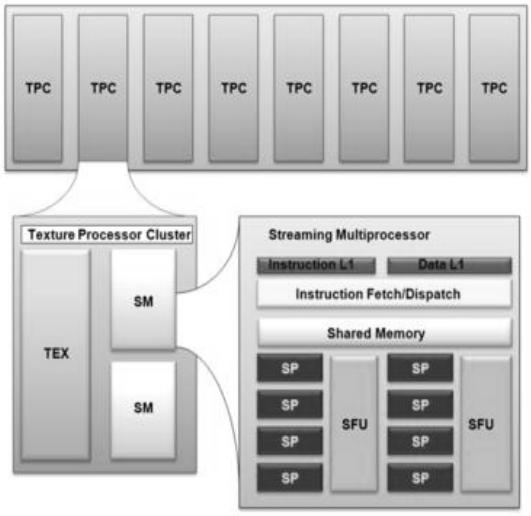


Figure 2. 2: Architecture Overview

Nvidia calls this mode of execution SIMT (for single instruction multiple threads). It's important to point out that the backend operates at double the frequency of the front end. In practice, then, the part that executes the instructions appears to be twice as "wide" as it actually is (that is, as a 16-way SIMD unit instead of an 8-way one).

The streaming multiprocessors' operating mode is as follows:

At each cycle, a warp ready for execution is selected by the front end, which launches execution of an instruction. To apply the instruction to all 32 threads in the warp, the backend will take four cycles, but since it operates at double the frequency of the front end, from its point of view only two cycles will be executed. So, to avoid having the front end remain unused for one cycle and to maximize the use of the hardware, the ideal is to alternate types of instructions every cycle – a classic instruction for one cycle and an SFU instruction for the other.

Each multiprocessor also has certain amount of resources that should be understood in order to make the best use of them. They have a small memory area called 'Shared Memory' with a size of 16 KB per multiprocessor. This is not a cache memory – the programmer has a free hand in its management. As such, it's like the Local Store of the SPUs on Cell processors. This detail is particularly interesting, and demonstrates the fact that CUDA is indeed a set of software and hardware technologies. This memory area is not used for pixel shaders.

2.6 Memory Layout

This memory area provides a way for threads in the same block to communicate. It's important to stress the restriction: all the threads in a given block are guaranteed to be executed by the same multiprocessor. Conversely, the assignment of blocks to the different multiprocessors is completely undefined, meaning that two threads from different blocks can't communicate during their execution. That means that using this memory is complicated. But it can also be worthwhile, because except for cases where several threads try to access the same memory bank, causing a conflict; the rest of the time, access to shared memory is as fast as access to the registers.

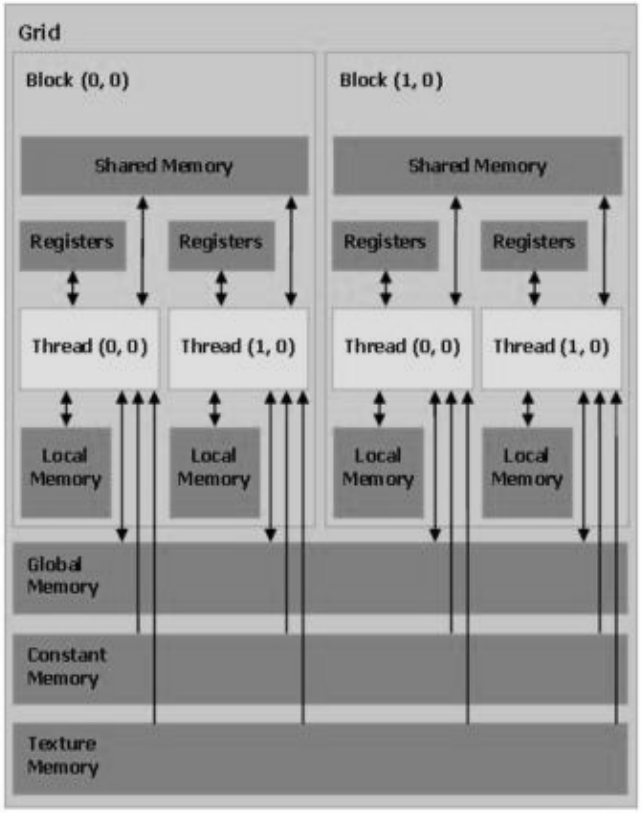


Figure 2.3: Memory Layout

The shared memory is not the only memory the multiprocessors can access. Obviously they can use the video memory, but it has lower bandwidth and higher latency. Consequently, to limit too-frequent access to this memory, Nvidia has also provided its multiprocessors with a cache (approximately 8 KB per multiprocessor) for access to constants and textures.

The multiprocessors also have 8,192 registers that are shared among all the threads of all the blocks active on that multiprocessor. The number of active blocks per multiprocessor can't exceed eight, and the number of active warps are limited to 24 (768 threads). So, an 8800GTX can have up to 12,288 threads being processed at a given instant. It's worth

mentioning all these limits because it helps in dimensioning the algorithm as a function of the available resources.

Optimizing a CUDA program, then, essentially consists of striking the optimum balance between the number of blocks and their size – more threads per block will be useful in masking the latency of the memory operations, but at the same time the number of registers available per thread is reduced. What's more, a block of 512 threads would be particularly inefficient, since only one block might be active on a multiprocessor, potentially wasting 256 threads. So, Nvidia advises using blocks of 128 to 256 threads, which offers the best compromise between masking latency and the number of registers needed for most kernels.

2.7 Software Model

From a software point of view, CUDA consists of a set of extensions to the C language, and a few specific API calls. Among the extensions are type qualifiers that apply to functions and variables. The keyword to remember here is *__global__*, which when prefixed to a function indicates that the latter is a kernel – that is, a function that will be called by the CPU and executed by the GPU. The *__device__* keyword designates a function that will be executed by the GPU (which CUDA refers to as the "device") but can only be called from the GPU (in other words, from another __device__ function or from a __global__ function). Finally, the *__host__* keyword is optional, and designates a function that's called by the CPU and executed by the CPU – in other words, a traditional function.

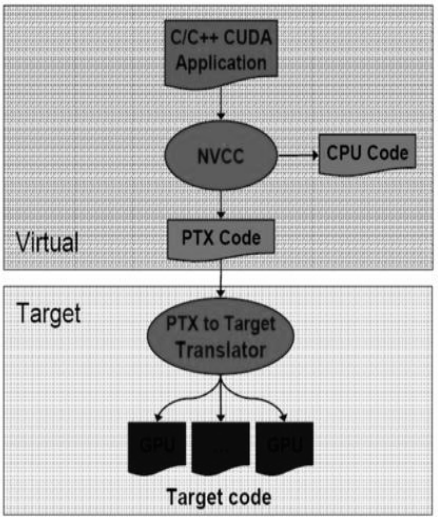


Figure 2.4: Software Hierarchy

There are a few restrictions associated with __device__ and __global__ functions: They can't be recursive (that is, they can't call themselves) and they can't have a variable number of arguments. Finally, regarding __device__ functions resident in the GPU's memory space, logically enough it's impossible to obtain their address. Variables also have new qualifiers that allow control of the memory area where they'll be stored. A variable preceded by the keyword *__shared__* indicates that it will be stored in the streaming multiprocessors' shared memory. The way a __global__ function is called is

also a little different. That's because the execution configuration has to be defined at the time of the call – more concretely, the size of the grid to which the kernel is applied and the size of each block. Take the example of a kernel with the following signature:

```
__global__ void Func(float* parameter);
```

which will be called as follows:

```
Func<<< Dg, Db >>>(parameter);
```

where `Dg` is the grid dimension and `Db` the dimension of a block. These two variables are of a new vector type introduced by CUDA.

The CUDA API essentially comprises functions for memory manipulation in VRAM: *cudaMalloc()* to allocate memory, *cudaFree()* to free it and *cudaMemcpy()* to copy data between RAM and VRAM and vice-versa.

Chapter 3: An Extension to the Burrows Wheeler Transform

3.1 Introduction to the Burrows Wheeler Transform

The Burrows Wheeler's transform[3] is a block-sorting, lossless data compression algorithm, which is used in applications such as bzip2. It was developed by Michael Burrows and David Wheeler. A variation of this algorithm was developed by Mantici et al.[4]5] which extended the concept to a multi-set of words, unlike the original algorithm which worked on a single block of text. A key realization by Mantici et al. was the applicability of their extended algorithm to the domain of bio-informatics, namely the matching of genomic data of species to establish their Phylogenetic proximity or non-relatedness.

3.2 Burrows Wheeler Transform

The Burrows-Wheeler transform was given jointly by Burrows and Wheeler in 1994. It is also known by another name called block-sorting. The very basic job of Burrows wheeler transform is to sorts the block of characters, according to a lexical ordering of their following context. This process can be thought as a sorting a matrix containing all cyclic rotations of the string. An example matrix shown in Figure 3.1 is constructed for the input string *mississippi*.

Each row is consists of one of the eleven rotations of the input, and then each rows have been sorted lexicographically. The first column (F) of this matrix contains the first characters of the each rotated string, and the last column (L) denotes the permuted characters of the string that form the output of the BWT.

It is also necessary to transmit the position of the original string in the sorted matrix is shown in fifth row of Figure 1. Therefore, the complete BWT output for the string $mississippi$ is the pair$\{pssmipissii, 5\}$, as it is find that only a few characters are likely to appear in any given context, that why move-to-front transform encoding scheme is consider as an ideal scheme for encoding, which replaces the recently seen symbols with shorter codes.
The random nature of the permuted text, make it almost impossible to recover the original text without any other information, but on the hand a reverse transformation can be performed easily on the given permuted string *L*, the first character index, and the sorted matrix *F*, which can be obtained from the permuted string is in $O(nlog|\Sigma|)$ time.

This is infer from the two important considerations: first one is that, the sorted matrix is constructed by performing cyclic rotations, so that each character in *L* is immediately

followed by the corresponding character in *F*, and secondly, that the instances of each distinct character appear in the same order in both arrays *F* and *L*. In order to develop a deep understanding an example given in which the third occurrence of the letter i in *L* corresponds to the third occurrence of i in *F*, and so on. In this example, the first character is m at position 5 of *F*, and that this must correspond to m at position 4 of L, since there is only one m in the text.

The next character is i at position 4 of *F*, since characters in *F* immediately follow the characters at the same index in *L*. This is the fourth i in *F*, and so it corresponds to the fourth i in *L* at position 11, which in turn must be followed by an s. Continuing in this way, it is possible to decode the whole string in $O(n \log |\Sigma|)$.[6]

If it is not necessary to decode the entire string at once, a transform array *W* can be computed in linear time with a single pass through the *L* and *F* arrays, such that

$$\forall i: 1 \leq n, T[i] = L[W^i[id]]$$

Where $W^0[x] = x, W^{i+1}[x] = W[W^i[x]]$, and id *is* the index of the original text in the sorted matrix. By traversing L with W, we have a means of decoding arbitrary substrings of the text.

.

S. NO	FIRST COLUMN LETTER (F)	PERMUTED STRING	LAST COLUMN LETTER (L)		S. NO	LAST COLUMN LETTER (L)	FIRST COLUMN LETTER (F)	SORTED LIST OF SUBSTRINGS
1	i	mississip	p		1	p	i	
2	i	ppimissis	s		2	s	i	ppi
3	i	ssippimis	s		3	s	i	ssippi
4	i	ssissippi	m		4	m	i	ssissippi
5	m	ississipp	i		5	i	m	ississippi
6	p	imississi	p		6	p	p	i
7	p	pimississ	i		7	i	p	pi
8	s	ippimissi	s		8	s	s	ippi
9	s	issippimi	s		9	s	s	issippi
10	s	sippimiss	i		10	i	s	sippi

Figure 5.1

3.3 Extended Burrows Wheeler Transform[7]

Mantici et al. produced a variation in of the Burrows Wheeler Transformation and extended it to a multi-set of words instead of a single block of text. In this case too, cyclic conjugates for all the words are produced and the entire lot is sorted. The sorting however is not lexicographic. The paper introduced another form of sorting called the ω sorting.

ω Sorting[4]

In normal lexicographic sorting, if we encounter two words of different lengths such that one is the prefix of another, then the smaller word is considered lexicographically smaller than the other, and hence is sorted above. However, in ω sorting, a word is expanded by repeating the same word over again, creating an infinitely (theoretical) long repeated sequence of original word (called, repeat-transformation). These repeat transformations of all the words are then sorted lexicographically.

Consider two words *ab* and *aba*

The conjugates of the word will be
ab and *ba*
aba, *baa* and *aab*

Sorting them lexicographically gives

1. *aab*
2. *ab*
3. *aba*
4. *ba*
5. *baa*

Creating repeats of the words we get
ab: abababababababab…
ba: babababababababa…
aba: abaabaabaabaaba…
baa: baabaabaabaabaa…
aab: aabaabaabaabaab…

Sorting these strings lexicographically gives

1. *aab*
2. *aba*
3. *ab*
4. *baa*
5. *ba*

3.4 Distance Measurement

The notion of distance between two sequences .Such a notion is based on the following intuitive idea. Given two sequences *u* and *v*, we consider the sorted list of the conjugates of *u* and *v*, obtained in the first step of the computation of E({*u*,*v*}). If the same segment s appears both in *u* and *v*, then the conjugates of *u* and *v* starting with s are likely to be close in the above list. The greater is the number of segments shared by the two sequences *u* and *v*, the greater is the number of alternations in the above list between the elements coming from *u* and those coming from *v*. Thus, we define a distance that takes into account the alternation of the symbols coming from different sequences in the output of the transformation E.

More formally, let S = {*u*,*v*} and let $w_1, w_2, w_3, \dots, w_m$ be the sorted list of conjugates of *u* and *v* obtained in the first step of the computation of E({*u*,*v*}). Consider the new alphabet Σ = {*U*,*V*} and the map γ that associates to each sequence w_i in the list, a symbol of Σ as follows:

$$\gamma(w_i) = \begin{cases} U, & \text{if } w_i \text{ is a conjugate of } u \\ V, & \text{if } w_i \text{ is a conjugate of } v \end{cases}$$

Let $\tau(u,v) = \gamma(w_1)\gamma(w_2)\dots\gamma(w_1)$

If $u, v \in A^*$ be two sequences and $\tau(u,v) = U^{n_1}V^{n_2}U^{n_3}\dots V^{n_k}$.
The distance is then defined as:

$$\delta(u,v) = \sum_{\substack{i=1 \\ n_i \neq 0}}^{k} (n_i - 1)$$

3.5 EBWT on CUDA[8][9]

The EBWT logic on CUDA was distributed into phases of functionality, with each phase being handled either by the CPU or the GPU, based on the feasibility to parallelize that phase.

The phases involved were:

- Accepting a weighted sequence and pattern
- Converting the weighted sequence into solid sequences and patterns, based on a threshold value.

- Generating conjugates for each of the generated solid sequence and pattern.
- Sort the repeated conjugates
- Calculate the distances based on the sorted data.

The conversion of weighted sequence to solid sequences is a typical exponential problem. Due to the nature of brute-computation required in enumerating the various sequences formed, parallelizing the operation was not found feasible.

However, in order to reduce the overhead of constants involved, an intuitive approach of efficiently selecting only those nodes that satisfy the threshold value was followed. In order to do this, the weighted positions were arranged in the descending order of the probabilities. Of these, those nodes satisfying the threshold value were marked 'Black' and a chain of such 'Black nodes' was created.[10]

The conjugate-generation of these solid sequences was parallelized. Every sequence was allotted a block, wherein, each thread in the block was responsible for the generation of one conjugate.
As an illustration, let's examine how the fifth conjugate of the first sequence was formed, assuming the sequence length to be 10.
The thread which would be assigned to generate conjugate number 5 would be the fifth thread in the block to which the first word is assigned to. The fifth thread, based on its threadIdx.x would simply copy the content of the first word from position 6 to 10 to a temporary memory location. It would then be followed by the copying of content of location 0 to 5 of the first word and concatenate it to the temporary location, thereby forming the required conjugate.

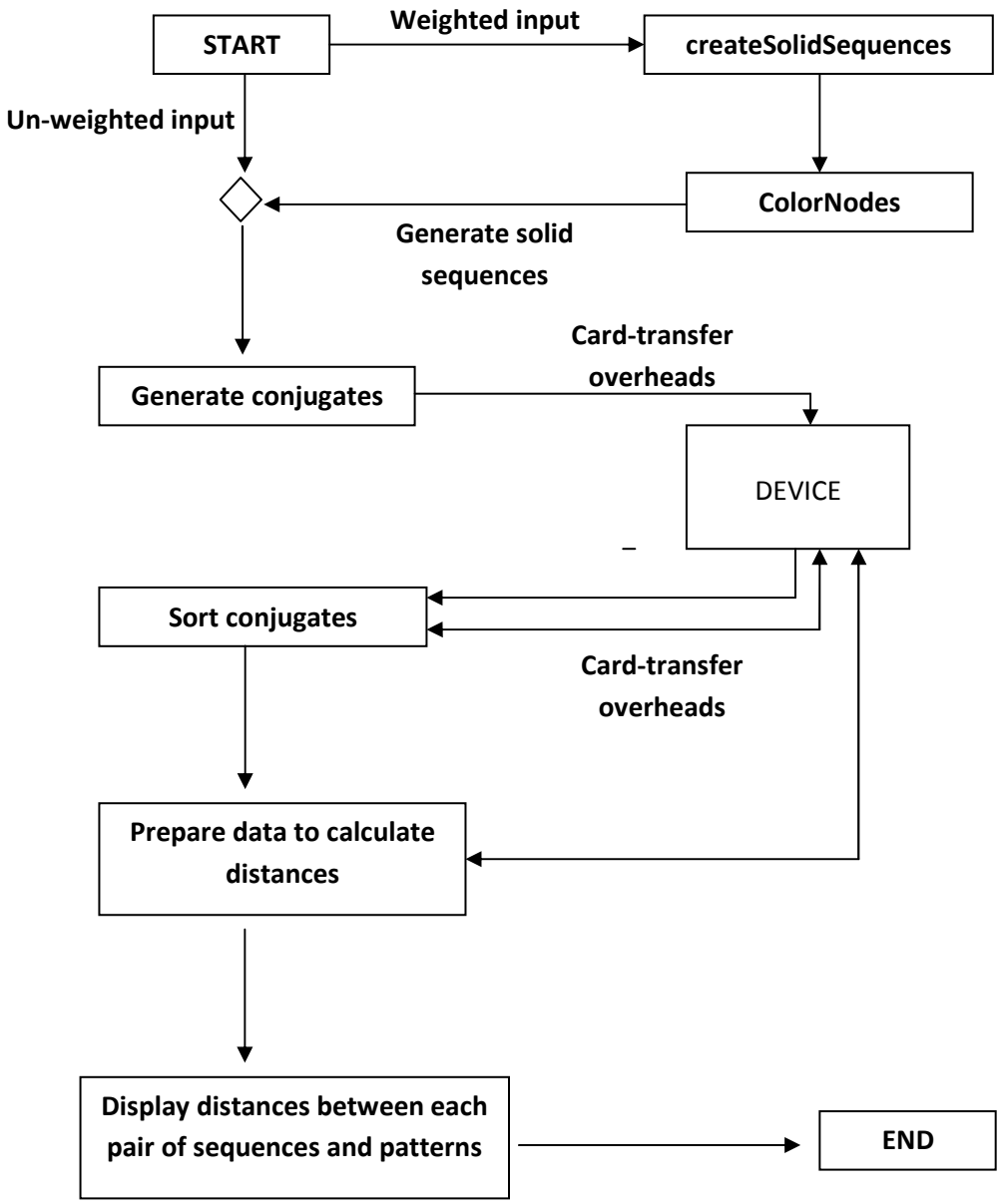
The total number of threads required for this operation would be the number of sequences and patterns times the maximum width a sequence or pattern, since, the number of conjugates generated in each pattern or sequence would be equal to the number of characters contained in it.

The conjugate-generation operation was followed by the sorting of these generated conjugates. Sorting in parallel has been a widely discussed topic. However, due to the strict constraints posed by the underlying architecture of the GPU, sorting efficiently was quite a challenge. The traditional Bitonic sort [11] proved to be too inefficient considering execution-time. As a move to improve this, *Thrust* [12] a set of libraries providing STL functionality in the CUDA framework was made use of. These libraries make use of an efficient Radix Sort as given by Satish et al. in the paper *Designing Efficient Sorting Algorithms for Manycore GPUs*.[13]

Once the sorted, the distances of the conjugates are then calculated. The distance calculation when linearly implemented, is invariably an $O(n^2)$ operation, where n is the number of generated conjugates. And given the low-inter communication required in

the calculation, this operation when parallelized produced promising results. The operation when parallelized makes one thread in a block responsible for the distance calculation of one sequence-pattern pair. Hence, a total of $O(n^2)$ threads, running in parallel, are able to calculate the distance measure.

To summarize the work, the following flow diagram shows the control of transfer in the execution of the EBWT logic:



Chapter 4: Analysis and Results

INPUT

- **Length of weighted sequence** : $4.\log_4(m)$
- **Length of weighted pattern** : $4.\log_4(n)$
- **Threshold parameter** : K **(Note: Threshold =** $^1/_K$ **)**

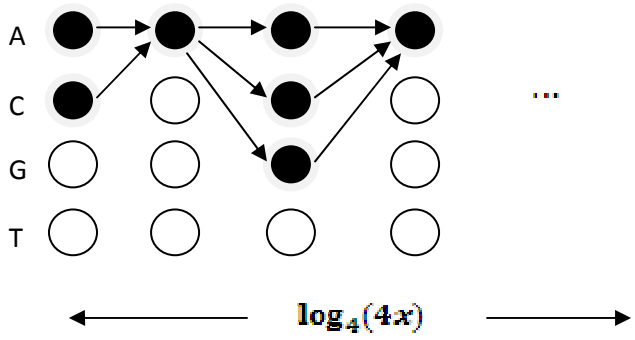
---

1. Conversion of weighted sequences to solid sequences:
Performed on Host

- In the following discussion, let x represent either m or n.
- m and n represent the number of positions in the input weighted sequence and pattern respectively.

Sub-operations and timing

1. Identify Black nodes, based on K: $O(4.\log_4 x) = O(\log_4 x)$



2. Construct Black-node graph: $O(4^{\log_4 x}) = O(x)$
3. Form solid sequences from above constructed graph: $O(4^{\log_4 x}) = O(x)$

Resulting data generated:

- Total number of solid sequences, φ_m : $O(4^{\log_4 m}) = O(m)$
- Total number of solid patterns, φ_n : $O(4^{\log_4 n}) = O(n)$
- Width of each solid sequences, W_m : $\log_4 m$
- Width of each solid pattern, W_n : $\log_4 n$

2. Initialize offsets and generate conjugates of each solid sequence
Performed on Device

Sub-operations

1. Generate conjugates of every solid sequence:

   Each solid word is assigned one block. Each thread in that block is assigned one character of that word. Each thread thus is responsible for creating one conjugate. The resulting conjugates are stored in a contiguous locations of a char[] data structure.

   - Conjugate generation cost by each thread : $O(1)$
   - Number of active threads $= \varphi_m . W_m + \varphi_n . W_n$
   - Number of active blocks $= \varphi_m + \varphi_n$
   - Responsible code:

```
dev_strncpy(conjugate_of_my_concern, pw_of_my_concern +
conjugate_no, pw_len gth[blockIdx.x]-conjugate_no);

dev_strncpy(conjugate_of_my_concern + pw_length[blockIdx.x] -
conjugate_no,p w_of_my_concern, conjugate_no);
```

   - Moreover, efficient utilization of card bandwidth is achieved while writing elements to the global memory space, due to the contiguous nature of memory calls.

   - The gather operations and string operations, are, however, not efficient.

2. Keep track of every conjugate's parent solid word : $O(1)$

   - Maintain a *from[]* array to keep in store the parent index.
   - Maintained as and when a conjugate is created.

3. Lexicographic Sorting of generated conjugates

Performed on Device

Sorting is the performance bottle-neck in our implementation. A Bitonic sort implemented in CUDA resulted in a degradation of performance as compared to Uni-processor performance.

For a Bitonic Sort, which makes use of the Odd-Even sorting principle, the results on the runs were:

Length of solid sequence	**Time taken to perform Bitonic Sort** (in s)	
	GPU	**CPU**
50	0.235	0.2500
60	0.341	0.3750
70	0.473	0.3280
80	3.226	2.188
90	27.148	25.172
100	171.97	120.468
110	235.184	155.690
120	314.291	199.469
130	411.521	269.570
140	513.339	298.2797
150	639.562	384.219
160	782.565	421.141
170	904.870	512.000
180	1600.894	598.370

190	1242.666	678.969
200	1433.8454	729.078
210	1648.092	883.219
220	1882.680	974.359
230	2134.472	1043.281
240	2422.185	1179.797
250	2442.583	1310.688
260	4086.381	1452.141
270	3359.381	1553.157
280	15610.748	6818.651
290	17331.973	8629.375
300	--	--

The data has been processed in the following environment:

- GPU used : NVIDIA Quadro 3700 FX
- CPU used : Intel Pentium 4 uniprocessor
- Compiling environment used: Microsoft Visual Studio 2005, enabled with NVCC, available through CUDA 3.0 SDK
- Threshold value for weighted sequence, K = ⅕

Shortcoming in the implementation of Bitonic Sort

- The implementation of the Odd-Even Sort or Bitonic Sort is divided into 3 stages, as seen from the perspective of the device. The stages are:

Stage 1:

In the device, each thread is assigned to handle two `ptr_d[ ]`* elements. In accessing these two elements, the threads create an un-coalesced gather operation from the global memory of the device, thereby greatly decreasing capable memory bandwidth and increasing the access time by a few hundred clock cycles.

Stage 2:

Once the values of `ptr_d[i]` and `ptr_d[i+1]` are obtained by a thread, the characters pointed to by these values are accessed through a global-memory access. (This is done by passing these values in `__device__ dev_strcmp()`* method)

Since this gather operation is more scattered than the previous operation, there exists a further inefficiency in the bandwidth utilization, thereby costing a few hundred more clock cycles for the operation.

Stage 3:

Once the characters from the sequences are read and compared, appropriate changes to their positions' data is calculated. This calculation is reflected onto `ptr_d[ ]` by making a global-write.

Since these writes too are scattered in nature, no attempt to utilize the full-bandwidth is done.

- The above stages highlight a high-degree of scattering while making global memory calls.

In addition to this, the number of kernel calls generated by the CPU for Bitonic Sort too is $O(\texttt{tot\_length}^*)$. In a typical run, $\texttt{tot\_length} \approx 10^6$.
Hence, the CPU overhead of launching these many kernels is also high.

An Improvement:

In the implementation of Bitonic Sort, it may be noted, that Stages 1 and 3 are unavoidable, no matter what our approach to the whole sorting operation is. An optimization, if any, may be considered for Stage 2 of this process.

Satish et al., in their paper *Designing Efficient Sorting Algorithms for Manycore GPUs*, discuss an efficient implementation of the Radix Sort on CUDA. This implementation is being incorporated on Stage 2 of our sorting process.

The theory discussed in the paper, however, has a limitation while being implemented. Its strategy to improve performance involves message passing among thread blocks. However, a clear description of the implementation details is missing. It maybe noted that CUDA explicitly does not support message passing between blocks of thread.

These shortcomings aside, the approach promises a considerable speed-up as compared to Bitonic Sort, if implemented. The features contributing to the projected speed-up are:

1. The number of kernel calls needed to sort the entire `all_repeat[ ]`* array would be of the order O(`max_width`*), in contrast to an order O(`tot_length`) required by Bitonic Sort.

2. The implementation makes use of the fast-caches, local to each thread-block to perform calculations, as opposed to using the global memory. The access to these caches is significantly faster than global memory accesses.

This shared memory is not used at all, in the implementation of Bitonic Sort.

---

* Please refer appendix for a description of the variables

Lexicographic Sorting using Thrust Libraries and consequences

Thrust is a library which provides an STL interface to coding in CUDA. Along with a user-friendly library data structures, it also provides highly-optimized implementation of standard algorithms, such as sorting and searching.

We attempted to implement our ideas using *Thrust. Thrust* does not provide any ready-made implementation of lexicographic sorting. In order to achieve this, an intermediate class had to be written, the objects of which imitated the list of solid words. The objects were then passed to the library defined sorting functions, keeping in regard the syntax and data-structure requirements of these functions. This helped achieve a significant improvement in our performance. The library sort functions use the ideas discussed in the paper by Satish et al.

The executions timed against the CPU performance along with the user-defined class are tabulated below.

Results on implementing Lexicographic Sort using Thrust

Length of solid sequence	**Time taken to perform Bitonic Sort** (in s)	
	GPU	**CPU**
50	0.016	0.2500
60	0.016	0.3750
70	0.031	0.3280
80	0.046	2.188
90	0.140	25.172
100	0.360	120.468
110	0.422	155.690
120	0.531	199.469

130	0.593	269.570
140	0.719	298.2797
150	0.812	384.219
160	0.921	421.141
170	1.079	512.000
180	1.265	598.370
190	1.375	678.969
200	1.531	729.078
210	1.688	883.219
220	1.906	974.359
230	2.047	1043.281
240	2.250	1179.797
250	2.422	1310.688
260	2.703	1452.141

Results on implementing Lexicographic Sort using Thrust with improvised CPU code

(in seconds)

Length of sequence	**GPU**			**CPU**		
	Sort	**Distance**	**Total**	**Sort**	**Distance**	**Total**
50	0.016	0.016	0.594	0.031	0.470	0.109
60	0.016	0.015	0.625	0.031	0.032	0.063

70	0.031	0.016	0.735	0.031	0.470	0.078
80	0.046	0.078	1.562	0.062	0.359	0.484
90	0.140	0.772	4.016	0.172	2.779	3.062
100	0.360	5.50	13.07	0.500	23.844	24.515
110	0.422	6.00	14.82	0.500	25.734	26.437
120	0.531	6.563	15.922	0.593	28.39	29.265
130	0.593	7.093	16.656	0.657	30.50	31.422
140	0.719	7.625	18.203	0.703	33.094	34.094
150	0.812	8.188	19.36	0.875	35.437	36.625
160	0.921	8.797	20.719	0.860	37.89	39.14
170	1.079	9.219	22.0	0.953	40.359	41.734
180	1.265	9.783	23.485	1.046	42.078	43.531
190	1.375	10.344	25.672	1.187	44.984	46.625
200	1.531	2.953	26.50	1.297	47.282	49.078
210	1.688	11.453	28.00	1.39	49.641	51.953
220	1.906	11.979	29.359	1.453	51.984	54.593
230	2.047	12.578	30.937	1.516	53.843	56.093
240	2.250	13.094	32.672	1.65	57.172	59.516
250	2.422	13.61	33.625	1.75	59.532	63.0
260	2.703	14.141	35.203	1.843	61.563	64.469

It may be observed in the previous results that GPU outperforms the CPU through a 5X speed up. However, the total time, which is not the sum of the times taken to sort the conjugates and calculate the distances between them, is reduced from the 5X mark due to the card's overhead of moving data to and from the RAM, hence, bringing down the speed up to 2X.

The implementation now brings in to focus the constraints posed by the ranges of the data-types used in the implementation. Given good sorting speeds, there is an upper limit to the amount of data which can be crunched by the current implementation. An analysis of this follows –

1. The maximum file size possible to be tested in current implementation

- In the course of the implementation, we define `tot_length` to keep track of the sum of all string lengths of the weighted sequences generated.

`tot_length` is assigned type `unsigned int`.

On a 64-bit machine, this translates to maximum capacity of the variable to be: $2^{4x8=32}$ since `unsigned int` is assigned 4 bytes.

- Let n be the string length of the Weighted input sequence. Then, the number of strings generated, in the worst case, are $O(4^n)$

In this case, the sum of all string lengths generated would be $O(n.4^n)$

- For correct operability, $n.4^n \leq 2^{32}$
  Solving for n, we get $n \approx 14.5$

As a result, we immediately get an upper bound to the size file usable for comparison of sequences.

Assuming each character consumes one byte, a typical input file of the pattern may not exceed 500 bytes (given, there exists a fixed format for the input files)

Moreover, there also exists a bound on the memory spaces `Malloc()` can allocate, the number of elements the STL data type `vector<>` can contain and the total number of blocks a kernel can launch.

Conclusion and Future Work

The results have shown that an efficient implementation on CUDA may result in a significant increase in performance. This increase encourages us to explore new applications wherein the functionalities of EBWT can be used to solve real-life problems.

As food for thought, we would like to see how EBWT would perform if applied to the problem of matching images. Image identification and matching is a problem with massive real-life application. If EBWT succeeds to imitate existing algorithms in recognizing and matching two images, we shall have found a plausible alternative in approaching the problem. Moreover, its impact onto application domains such as robotics, reverse-image searching etc. shall also be interesting to study.

Moreover, if the functionality of EBWT is able to translate to the domain of matching images, it shall help analyze many other open-ended problems such as matching of audio-samples, matching of motion pictures etc. These problems, if solved efficiently, promises tremendous impact on everyday applications.

Appendix

1. Table of some key variables used in the implementation

Name of variable	Data Type	Size/Bound	Description
`max_width`	`unsigned int`	`< 500`	It contains the maximum length a solid sequence may have
`tot_length`	`unsigned int`	`< sizeof(int)`	It contains the sum of the string lengths of all solid sequences generated
`all_repeats`	`char*`	`max_width*tot_length`	It contains all the generated conjugates/ padded conjugates needed for the EBWT, placed contiguously in the array.
`ptr_d`	`char*`	`tot_length`	It contains the address of the beginning of every solid sequence stored in `all_repeats[]`

2. Class definition to imitate a string on the device

```
#include <thrust/device_ptr.h>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <thrust/copy.h>

#define POOL_SZ (100*1024*1024)
using namespace std;

class device_string
{
public:
    int cstr_len;
    char* raw;
    thrust::device_ptr<char> cstr;

    static char* pool_raw;
    static thrust::device_ptr<char> pool_cstr;
    static thrust::device_ptr<char> pool_top;

     // Sets the variables up the first time its used.
    __host__ static void init()
        {
            static bool v = true;
            if( v )
            {
                v = false;

                pool_cstr = thrust::device_malloc(POOL_SZ);
                pool_raw  = (char*)raw_pointer_cast( pool_cstr
);
                pool_top = pool_cstr;
            }
        }
     // Destructor for device variables used.
    __host__ static void fin()
        {
            init();
            thrust::device_free(pool_cstr);
        }

     // Parametrized constructor to copy one device_string to
another.
    __host__ device_string( const device_string& s )
```

```
        {
            cstr_len = s.cstr_len;
            raw = s.raw;
            cstr = s.cstr;
        }

     // Parametrized constructor to copy a std::string to
device_string type
    __host__ device_string( const std::string& s )
          {
               cstr_len = s.length();
               init();
               cstr      = pool_top;
             pool_top += cstr_len+1;
             raw = (char *) raw_pointer_cast(cstr);
             cudaMemcpy( raw, s.c_str(), cstr_len+1,
cudaMemcpyHostToDevice );
          }

     // Default constructor.
     __host__ __device__ device_string()
          {
               cstr_len = -1;
               raw = NULL;
          }

     // Conversion operator to copy device_string type to
std::string
     __host__  operator std::string (void)
          {
            thrust::host_vector<char> temp(cstr_len);
            thrust::copy(cstr, cstr + cstr_len, temp.begin());
            std::string result(temp.begin(), temp.end());
            return result;
          }
};
char* device_string::pool_raw;
thrust::device_ptr<char> device_string::pool_cstr;
thrust::device_ptr<char> device_string::pool_top;

// User-defined comparison operator
bool __device__ operator < (device_string lhs, device_string
rhs)
{
    char* l = lhs.raw;
    char* r = rhs.raw;
```

```
    for( ; *l && *r && *l==*r; )
    {
        ++l;
        ++r;
    }
    return *l < *r;
}
```

References

1 NVIDIA Corporation, NVIDIA CUDA Programming Guide, Nov. 2007. Version 1.1.

2 NVIDIA Corporation, NVIDIA OpenCL Best Practices Guide Version 1.0, July 10, 2009.

3 M. Burrows and D.J. Wheeler, A Block-sorting Lossless Data Compression Algorithm, SRC Research Report, DIGITAL Systems Research Center, Palo Alto, May 10, 1994.

4 S. Mantaci, A. Restivo, G. Rosone, M. Sciortino, An extension of the Burrows–Wheeler Transform, Theoretical Computer Science 387, Pp. 298–312, 2007.

5 Sabrina Mantaci, Antonio Restivo, G. Rosone, and Marinella Sciortino, An Extension of the Burrows Wheeler Transform and Applications to Sequence Comparison and Data Compression, Lecture Notes in Computer Science, Springer 3537/2005, Pp. 178-189, May 2005.

6 Jamie Simpson, Simon J. Puglisi, Words with simple Burrows-Wheeler Transforms, The Electronic Journal of Combinatorics, Volume 15(1), 2008, #R83, 2008.

7 Matt Powell, Compressed-Domain Pattern Matching with the Burrows-Wheeler Transform, Honours Project Report, Canterbury Corpus, New Zealand, 2001.

8 Gregory Diamos, Sudnya Padalikar, Exploring The Latency and Bandwidth Tolerance of CUDA Applications, NFinTes Tech Report, December 2009.

9 Lung-Sheng Chien, Hand-Tuned SGEMM on GT200 GPU, Technical Report, Department of Mathematics, Tsing Hua university, R.O.C. (Taiwan), 2009.

10 Costas S. Iliopoulos, Christos Makris, Yannis Panagis, Katerina Perdikuri, Evangelos Theodoridis, The Weighted Suffix Tree: An Efficient Data Structure for Handling Molecular Weighted Sequences and its Applications, Fundamenta Informaticae, 71(2-3), Pp. 259-277, 2006.

11 T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, MIT Press, Second edition, Sept. 2001.

12 Jared Hoberock and Nathan Bell, Thrust: A Parallel Template Library, version 1.2, http://www.meganewtons.com/, 2009.

13 Nadathur Satish, Mark Harris, Michael Garland, Designing Efficient Sorting Algorithms for Manycore GPUs, Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium, May 2009.