Parallelizing Alignment in Bowtie Using GPU

Salvador Aguinaga, Olivia Choudhury, Fazle Faisal, Yuriy Hulovatyy

Department of Computer Science and Engineering
University of Notre Dame, Notre Dame, IN

***Abstract*—Advancements in Next Generation Sequencing (NGS) technologies has led to the generation of massive datasets at an unprecedented rate. To keep pace with this high-throughput technology, it is imperative to develop efficient genome alignment tools for comparing different genome sequences. Sequence comparison can shed light on the structural, functional, or evolutionary relationships between different genomes. Although running aligners in multiple threads can reduce the computation time, further performance improvement could be achieved by leveraging the benefit of graphics processing unit (GPU), a fact supported by recently developed GPU-implemented alignment tools. This work focused on the challenges of implementing the GPU-based version of *Bowtie*, a widely-used short-reads alignment tool. We addressed the issue of random memory access and diverging branches and achieved a significant performance gain over the CPU-version of its global alignment procedure. In particular, the GPU-implementation of the global alignment procedure outperforms its CPU-version by 17×.**

I. Introduction

DNA sequencing has emerged as a promising technique in the domain of molecular biology for providing the "blueprint" of an organism by generating the sequence of its hereditary information, as encoded in the DNA. The advent of NGS [1] technologies has facilitated its implementation in molecular medicine, anthropology, evolution, human migration, DNA forensics, agriculture and bioprocessing and several other areas of research. This technique results in the generation of large-scale dataset, which necessitate development of efficient genome alignment tools for comprehending the underlying structural, functional or evolutionary information among different sequences. Genome alignment is the procedure of determining the position of a shorter query genome relative to a longer reference genome. This has been illustrated in Fig. 1. Here, insertion or deletion (indel) of a nucleotide in a sequence is represented by '–' whereas matching nucleotide pairs are connected with '|'.

```
A: C A T - T C A - C
   |   |       |   |
B: C - T C G G A G C
```

Fig. 1: Genome alignment between two sequences A and B

The alignment of short sequences or reads generated by NGS technology with respect to a reference sequence often requires not only more time but also higher computational resources. As we discuss later in detail, Bowtie [2] has been proved to be one of the fastest genome alignment tools currently available. The advantage of *Bowtie* also lies in its small memory footprint that enables it to work for the human genome of size 3 billion base-pairs even on a computer with 2 GB RAM. Although the GPU-enabled short read aligners like CUSHAW [3], SOAP3 [4] and SARUMAN [5] perform better than *Bowtie* in terms of overall computation time, their memory requirement is still a matter of concern. Hence, improving the computation time of *Bowtie* by levering the benefits of GPU, in addition to preserving its low memory footprint requirement, will enable *Bowtie* to be the most efficient genome alignment tool in comparison to the GPU-enabled aligners as well as their CPU-counterparts.

The initial step of alignment in *Bowtie* involves indexing the reference genome sequence using the data compression algorithm Burrows Wheeler Transform (BWT) [6] and Ferragina and Manzini (FM) index [7]. The data structure thus formed is then used for locating regions of similarity between the reference sequence and a given query sequence.

In this project, we have focused on the alignment phase, not the indexing phase, for parallelization due to the following reasons. First, for a given reference sequence we do not need to build the index before every alignment as the index is reusable. Second, as supported by experimental results (see Table III) the indexing phase does not require as much time as the alignment phase does. Third, for the same reference sequence, the overall execution time will be determined by the query sequence and hence the alignment time involved for it.

We have identified the major challenges in utilizing GPU programming for the alignment procedure of *Bowtie*. Firstly, the BWT-based pattern searching involves considerable random memory access, which is not suitable for GPU. Secondly, diverging branching cannot be determined until run-time. Finally, a particular challenge with the alignment algorithm of *Bowtie* is its backtracking logic, which unless modified may cause a major bottleneck in achieving a significant performance gain.

In addition to describing the challenges, we also discuss our proposed solutions and present experimental data to support them. Considering the scope of the project and the time constraint, we have mainly focused on the first issue (that is, memory accesses) and implemented memory coalescing to ensure reads from the query sequence, loaded in the global memory of GPU, are accessed in a way such that a higher memory throughput is achieved. Memory access patterns have

a significant impact on the performance, and thus considering this issue is critical for achieving performance gain.

The report is organized as follows. In the next section, we review and discuss related works. In particular, we describe existing short read aligners, including GPU-based tools, and review their features and limitations. Section III presents methods used in our project. It starts with a short overview of CUDA, which includes discussion of its hardware and software architecture. Then we introduce challenges related the CUDA-based version of *Bowtie* and our proposed solutions them. In this section, we also describe test dataset, evaluation metrics, and tools used for our experiments. Section IV contains the experimental results and their discussion. It describes the approaches we used and the methods of their evaluation. We end this report with a discussion of the conclusions we made based on the results of our implementations and their analyses. Finally, we consider advantages and limitations of our approach, and discuss directions for future work.

II. Related Works

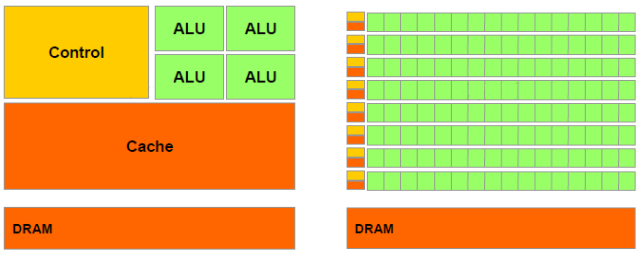
SOAP [8] is one of the earliest short read aligners, which has been designed and implemented by taking into account the short reads generated by next generation sequencing technology. One of the major limitations of SOAP lies in its huge memory usage. Then, SOAP2 [9] has been implemented as an improved version of SOAP, which reduced the memory usage by more than 3 times and improved the alignment speed by around 20-30 times. In particular, SOAP2 uses BWT compression indexing to reduce the memory usage of the reference genome.

Bowtie is a classical memory efficient short read aligner. The memory-efficient design of this aligner requires only 1.3 Gb memory footprint to align more than 25 million short reads with whole human reference genome. The bottleneck of *Bowtie* lies in its backtracking algorithm for alignment. The drawback has been overcome by Bowtie2 [10] through the implementation of hardware-accelerated dynamic programming. In addition, the inclusion of full-text minute index has improved the memory efficiency of Bowtie2. In summary, Bowtie2 is the best non-GPU short read aligner both in terms of performance and memory efficiency.

SARUMAN [5] is the first GPU-based short read aligner. It distributes the alignment of short reads with reference genome among multiple cores using GPU hardware. The limitation of this aligner lies in capability of aligning only small-scale genomes and lack of support for pair-end mapping.

CUSHAW [3] is a sophisticated GPU-based short read alignment tool that overcomes the limitations of the previous aligner by aligning short reads to whole human reference genome and facilitating pair-end mapping. CUSHAW has few limitations. First, the alignment quality of CUSHAW has not been widely tested. Second, CUSHAW does not support paired-end mapping.

SOAP3 [4] is a modified version of SOAP2 that has been implemented in GPU to take advantage of the explicit parallelization over multiple processor cores. Keeping the full functionality of SOAP2, SOAP3 takes less than 30 seconds to align one million short reads with human reference genome, which is eventually 7.5 times faster than BWA and 20 times faster than *Bowtie*. SOAP3 re-designs the data structure of SOAP2 to reduce memory accesses among threads running on different cores. Furthermore, SOAP3 performs parametric study to identify a pattern of having "too many" branches, groups them together and eventually reduces the idle time of each core.



(a) (b)

Fig. 2: **CPU and GPU designs.** CPUs and GPUs have different design philosophy. (a) CPUs have latency-oriented design. (b) GPUs have throughput-oriented design.

As the non-GPU version of *Bowtie* runs faster than the non-GPU version of SOAP, it is possible that the GPU version of *Bowtie* may outperform the GPU version of SOAP. In addition, the low memory usage of *Bowtie* has created an opportunity to align longer genome sequences. In this project, we aim to understand the feasibility of GPU implementation of *Bowtie* and implement a part of short read alignment algorithm used by *Bowtie* to understand the benefit of parallel computing to improve performance. In particular, we focus on the migration of *Bowtie* from CPU to GPU. We consider the full implementation of *Bowtie* as a potential future work and we provide a baseline towards that direction.

III. Methods

A. Overview of CUDA

1) GPU programming: In order to use CUDA effectively, it is essential to understand the underlying hardware of CUDA-enabled GPUs. GPUs, invented by NVIDIA in 1999, are probably the most pervasive parallel processors up to date [11]. In contrast to CPUs that are designed to execute relatively few large tasks sequentially, GPUs are focusing on parallel execution of many small tasks (see Fig. 2). Due to their main responsibility to excel at graphics, GPUs have evolved into processors with unmatched floating-point performance and programmability. Thus, they greatly outperform CPUs in arithmetic throughput and memory bandwidth, which provides a promising way to accelerate a variety of data parallel applications.

However, comparing to the CPU's traditional data processing pipeline, performing general-purpose computations on a GPU is a relatively new concept [12]. Initially, despite promising high arithmetic throughput of GPUs, the programming model was still far too restrictive for any critical mass of developers to form. There were tight resource constraints,

since programs could receive input data only from a handful of input colors and a handful of texture units. There were serious limitations on how and where the programmer could write results to memory, so algorithms requiring the ability to write to arbitrary locations in memory (scatter) could not run on a GPU [13].

To address these issues, NVIDIA introduced Compute Unified Device Architecture (CUDA), a software and hardware architecture that enabled the GPU to be programmed with a variety of high level programming languages. This architecture included several new components designed strictly for GPU computing and aimed to alleviate many of the limitations that prevented previous graphics processors from being legitimately useful for general-purpose computation [13]. NVIDIA took industry standard C and added a relatively small number of keywords in order to harness some of the special features of the CUDA Architecture. So, users are no longer required to have any knowledge of the OpenGL or DirectX graphics programming interfaces, and they do not have make their problems look like computer graphics tasks.

To this end, many applications in various areas, e.g., medical imaging, computational fluid dynamics, and environmental science, were built using CUDA [13], [14]. These applications have orders-of-magnitude performance improvement over the previous state-of-the-art implementations. Furthermore, applications running on NVIDIA graphics processors enjoy superior performance per dollar and performance per watt than implementations built exclusively on traditional central processing technologies.

2) Hardware architecture: A CUDA-enabled GPU is a fully configurable scalable processor (SP) array organized into a set of streaming multiprocessors (SM) [15]. The device contains 16 SMs with each SM comprising 32 SPs. Each SM has 32-bit registers with a total size of 32 KB and has a configurable shared memory size from the 64 KB on-chip memory, which can be configured at run-time for each CUDA kernel. Each thread has 512 KB of local memory. There is also local and global memory caching through a L1 cache of configurable size per SM and a unified L2 cache of size 768 KB per device. This cache hierarchy can provide a significant improvement of the access performance to the external device memory compared to direct accesses. Since it is possible to disable the global, but not local memory caching in the L1, it is important to determine the best combination for a kernel: 16 KB or 48 KB L1 cache with or without global memory caching in L1 and with more or less usage of local memory [16]. For example, kernels having more local and global memory access can potentially benefit from a larger L1 cache.

A hierarchy of processors on the GPU corresponds to the hierarchy of threads: a GPU executes one or more kernel grids, a streaming multiprocessor (SM) executes one or more thread blocks, and CUDA cores and other execution units in the SM execute threads. A SM executes threads in small groups of 32 parallel threads, called *warps*, which are scheduled in a single instruction, multiple thread fashion. While understanding of warp execution is not necessary for functional correctness, it is nevertheless important to take it into account. If threads in a warp execute the same code path and access memory in nearby addresses, the performance will be greatly improved [11].

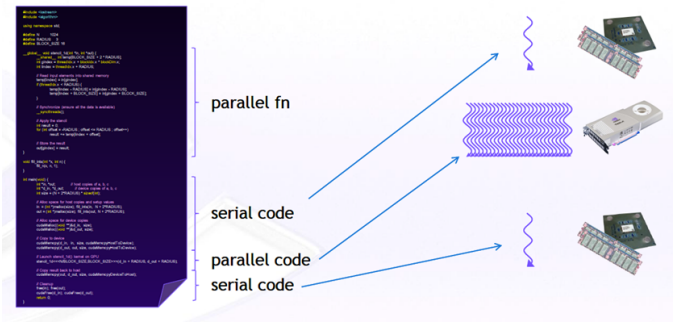


Fig. 3: **CUDA programming model.** Serial C code executes in a host thread. Parallel kernel C code executes in many device threads across multiple processing elements.

3) Software architecture: NVIDIA took industry standard C and added a relatively small number of keywords in order to harness some of the special features of the CUDA Architecture. So, users are no longer required to have any knowledge of the OpenGL or DirectX graphics programming interfaces, and they do not have make their problems look like computer graphics tasks.

In CUDA, the GPU is regarded as a coprocessor that executes data-parallel kernel code. A CUDA program contains code for the *host* (CPU and the system memory) and for the *devices* (GPUs and their memory). Thus, it consists of multiple phases executed on either the CPU or the GPU. The phases with little or no data parallelism are implemented in host, which is expressed in ANSI C and compiled with the host C compiler (see Fig. 3). On the other hand, the phases with rich data parallelism are implemented as functions in the device code that are compiled by the NVIDIA CUDA C compiler and the kernel GPU object code generator [17]. The host, to transfer data to and from the global memory of GPU, must use API calls.

A function that executes on the device is typically called a *kernel*. A kernel cannot contain any recursion, static variable declarations, and must have a non-variable number of arguments [17]. It is launched over a set of lightweight parallel threads on GPUs. The threads are organized into a grid of thread blocks.

B. Challenges

1) Memory accesses: One of the main challenges in using GPUs in *Bowtie* is related to memory accesses. BWT is a sophisticated compressed indexing data structure, and therefore pattern searching using BWT requires a huge number of random memory accesses. That is, there will be a lot of random access to the index and the auxiliary data structures (see Fig. 4), which, given that accessing GPU global memory is expensive, can negatively affect performance.

Based on how the reads have been loaded in the global memory of GPU, the alignment procedure can cause a significant overhead in data transaction. Thus, in order to achieve peak memory performance, it must be ensured that the computation involves data obtained from sequential memory access.

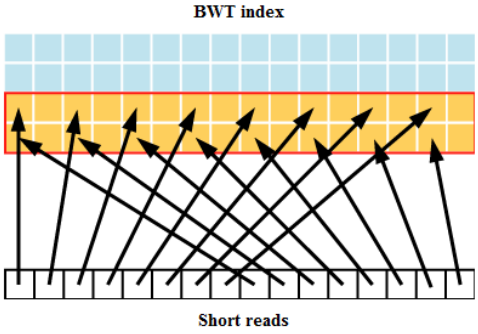


Fig. 4: **Random memory accesses.** Pattern searching using BWT involves a lot of random accesses to the index and the auxiliary data structures.

A direct implementation of the CPU version on GPU would induce heavy memory contention among different threads and thus considerably reduce the overall performance. One way to deal with this problem is to redesign the data structure of the index to maximize locality.

In addition to this, with significant memory traffic, we may expect a high cache miss rate, which again would negatively impact performance. A potential way to solve this problem is to use cached global memory instead of cached texture memory to cache reference genome index. This strategy should improve the performance for two reasons. First, global memory cached in L1 performs faster than texture memory. Second, cached global memory gets rid of the overhead of cache coherency.

It is also important to achieve the benefit of coalesced memory access, since this can make huge difference in terms of performance [11]. Access time of global memory is very sensitive to memory access patterns, and effective bandwidth can vary by an order of magnitude depending on them. In particular, memory accesses that are coalesced within a warp are generally faster than those that are not. In order for memory accesses to be coalesced within a warp, they need to access nearby locations in the memory in such a way that no two threads in the warp access the same memory location.

Since the patters to be aligned are loaded into the global memory and accessed frequently during alignment, we need to make sure that they are stored in a specific way to allow coalesced memory accesses. Specifically, the reads can be partitioned in the groups of the warp size and arranged in such a way that the threads in a warp simultaneously access, for example, the first words of the reads. In this way, the memory locations will be accessed form a contiguous 128-byte segment and 32 memory accesses will be coalesced and done in a single memory transaction. As a result, this will help to achieve a high memory throughput.

2) Branching effect: Another difficulty comes from the fact that GPU works in a single-instruction multiple-thread mode. That is, processors in the same unit must execute the same instruction. Consequently, too many diverging branches in the execution path would force some of the processors to idle, but it cannot be determined until run-time exactly how many diverging branches a pattern may introduce. So, the divergence of alignment paths for threads in a warp becomes a challenge.

In particular, during the alignment, some patterns can generate more than one range due to mismatches, and thus executing them will require more computations. Processing such patterns together with patterns having less ranges may have a negative impact on the elapsed time of the whole warp. Thus, separating patterns into different warps will help to reduce branch divergences.

The problem is, however, to determine these complicated patterns before they are actually aligned. It can be decided in run-time whether a pattern would introduce too many branches by using the number of ranges it generates [4]. That is, if the number of ranges is larger than a certain threshold, we will consider a pattern to be complicated.

After identifying such patterns, they can be collected, and then be attempted to execute in the second round of the alignment, after patterns with fewer ranges. Since in the second round there will more allocated space for each pattern, since they are known to generate more ranges. If, however, a pattern has too many ranges even for the second round (that is, more than a threshold value), it can be terminated and passed to CPU. Given that the number of such patterns will be relatively small, processing them on the host will not significantly impact performance, while it will help to reduce branch divergence. The way of executing patterns in rounds according to their complexity is illustrated on Fig. 5.

In addition to this, diverging branches may be caused by the fact that *Bowtie* relies on backtracking for alignment. In particular, it uses a quality-aware backtracking algorithm that allows mismatches and favors high-quality alignments [2]. While this is beneficial for the alignment quality, excessive backtracking may have a negative effect on performance [18].

To avoid this, *Bowtie* uses a technique called “double indexing”. That is, it produces two indices of the genome (forward and mirrored) and proceeds in two stages corresponding to which half of the read contains the mismatch. However, this will not prevent excessive backtracking in case alignments are permitted to have two or more mismatches. In this case, we can make a limit on the number of backtracks allowed before a search is terminated. While this may prevent us from reporting low-quality but legitimate alignments, this is an acceptable trade-off for most cases.

C. Test Dataset

The test dataset was collected from Dr. Pfrender’s laboratory in the Biological Science department in University of Notre Dame. It comprises the genome sequences of *Daphnia pulex* and two lanes of *Daphnia magna*. We considered *Daphnia magna* to be our reference sequence and *Daphnia pulex* as our query sequence. We used these for running the original *Bowtie* and CUSHAW tools. Since SOAP3 does not support paired-end alignment, we used only one of the lanes as our query sequence. Due to the huge size of the files, we tested our GPU-implemented global alignment procedure on a subset of the query dataset, with number of reads varying between 1000 and 100000, as illustrated in Section IV.

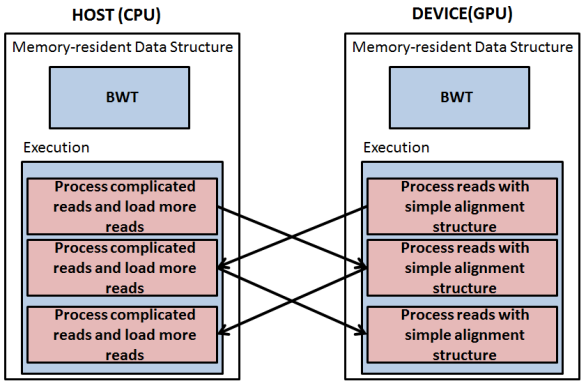


Fig. 5: **Processing reads in rounds.** Initially, the reads are attempted for execution on the device. However, if they are too complicated, they are processed on the host.

D. Implementation plan

1) Evaluation Metrics: Understanding the already complex nature of the *Bowtie* source code in order to decipher what and which parts to parallelize via the GPU devices, our work focused on profiling the run-time (or execution time) and use of memory.

- **Execution Time:** Execution time or run-time was profiled by modifying the source code or using built-in time-reporting features of the describe applications. Other methods to confirm execution time involved using performance tools like **perf**, **valgrind**, and **qsub**. In III-D.2) we discuss the tools in more detail.

- **Memory Profiling:** Memory utilization is obtained using **TotalView** and by submitting a job to the CRC queue (qsub). Table IV shows a baseline of CPU time, wall-clock time, and overall peak virtual memory utilization for *index building* and *alignment* processes. In this work, we explored the parallelization across multiple-nodes using Message Passing (MPI). Thus, we used the batching system **qsub** to send a job to be processed over many nodes (not just multi-core), a process that could have been profiled using CRC's **xymon** tool. However, we quickly learned that the code itself needed to be ported to execute in such manner. We abandon that approach to focus on parallelizing the execution across a group of graphics computing cores. Other tools used are described further below (in III-D.2).
  - **Heap:** A report showing what objects in the source code are allocate and hold memory during run-time. The objective is to use this information to understand memory usage during pattern matching. We expect this information will yield insights into whether memory coalescence can be used for performance enhancement. **Valgrind** also allowed us to profile the cache and get a picture of what objects (functions in the source code) execute transactions to cache and memory. This information was crucial to our understanding of how Bowtie works and how memory is utilized, since that is one of the major strengths of this application.

2) Tools: In Table I we list the profiling tools used to obtain baseline information needed to gain insights that directed our implementation plan.

TABLE I: Profiling Tools Used

Tool	Description	Purpose
TotalView	Memory Usage	Memory debugging and usage
Perf	Performance counters subsystem in Linux	Memory counters i.e. page faults, IPC, branch misses, etc.
Valgrind	Instrumentation framework for debugging and profiling	Detailed cache information
VisualProfiler	Performance profiling tool for CUDA	Optimizing our CUDA C/C++ implementation
Xymon	Profiling tool for monitoring servers, applications and networks.	Memory utilization for a given system

The development and evaluation environment for this work was fixed Table II shows the details. Both hardware and software resources (profiling tools) are from Univ. of Notre Dame's Center for Research Computing (CRC). This table provides high and low-level system details.

TABLE II: Development and Evaluation Environment

Host: fermife.crc.nd.edu	
System Characteristics:	
IBM I-Dataplex dx360 M3 graphics front end server	
Dual Six-Core Intel Xeon E5645	
Running at 2.4GHz processors and with 24GB RAM	
GPU Devices:	**2 NVIDIA Tesla M2070**
Total global memory:	**5375 MBytes**
Multiproc x CUDA Cores/MP:	**(14) x (32) = 448 CUDA Cores**
GPU Clock Speed:	**1.15 GHz**
System Cache Details:	
Instructions (I1) cache:	32768 B, 64 B, 4-way assoc.
Data (D1) cache:	32768 B, 64 B, 8-way assoc.
L2 (LL) cache:	12.58 KB, 64 B, 24-way assoc.

IV. Experimental Results

We evaluate the performance of *Bowtie* using the following metrics: execution time, memory, and cache utilization. In addition, we evaluate the performance of related GPU-enabled short-read aligners such as **CUSHAW** and **SOAP3**. The objective is to establish a performance target for our GPU implementation of *Bowtie*.

TABLE III: Baseline *Bowtie* Indexing and Alignement Performance (totalview, single thread)

	Execution Time (hh:mm:ss)	Wall-clock time (hh:mm:ss)	Peak VMem (MB)	Phase (Queue)
Bowtie	03:32	04:21	375.215	**Indexing** (short@dqcneh029)
Bowtie	02:17:03	02:22:10	340.086	**Alignment** (short@dqcneh017)

First, we obtained the performance characteristics for the index generation procedure, but quickly learned that it does not require significant portion of the total execution time. Table III provides the execution time and memory utilization for **Bowtie's** index and alignment phases. Thus, we focused on the alignment process and obtained both an execution time and memory utilization profile.

Next, we looked at how *Bowtie* scales when we increase the number of threads (it utilizes pthreads for multi-threading). Table IV and Fig. 6 show how run-time goes down linearly as a function of thread number. GPU enabled **CUSHAW** is executed with one GPU device (a NVIDIA TESLA device in this case) and is able to achieve some improvement over **Bowtie's** single-threaded equivalent, at a significantly higher memory cost.

TABLE IV: Execution Time and Memory Usage: During Paired-End Alignment (Unless Noted Otherwise)

	Cores	CPU Time (hh:mm:ss)	Virtual Mem (MB)	Heap (MB)	Runs
Bowtie	1	2:16:16	288.26	197.0	3
Bowtie	4	0:28:29	665.25	586.3	1
Bowtie	8	0:16:44	1185.40	1106.4	1
Cushaw[a]	1	1:19:45	10700	76.5	1
Bowtie[b]	8	00:39:59	647	1404	1
SOAP3[c]	1	00:02:43	16700	22	1

[a]GPU enabled
[b]In single-ended alignment
[c]In single-ended alignment and GPU enabled

The results obtained using **perf**, as shown in Table V, reinforces our implementation plan. While these data are superficial (more measurements could potentially paint a different picture), these offer sufficient information. For example, we can see that *Bowtie* yields less Page Faults for sufficiently large input files compared to the other GPU implementations. Future implementations would leverage any techniques inherent in the design of *Bowtie* to maintain a low number of Page Faults per million of instructions. These data also show that there is room for improvement to achieve higher IPC and potentially improve the run-time costs due to branch misses through further exploration of the GPU hardware.

Table IV shows measurement results of run-time and the memory utilization using the data-set in paired-end mode (using both reference input files). Separated by a line is *Bowtie*

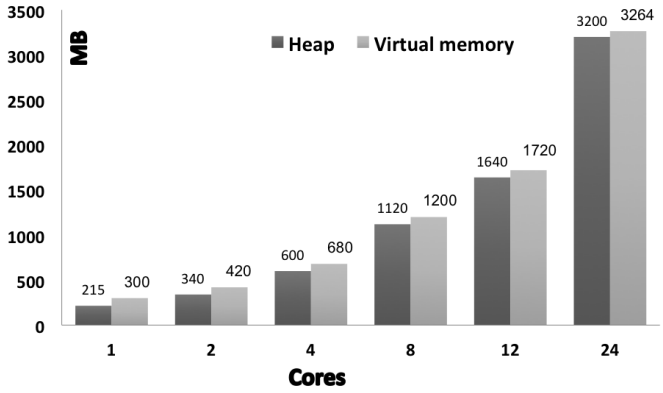


Fig. 6: Bowtie memory usage for various threads/cores.

TABLE V: Memory Statistics Using Perf

	$\frac{Page\ Faults}{Million\ of\ Instr}$	IPC	Branch-misses (% of all branches)
Bowtie	3.1004e-4	1.38	7.24
CUSHAW	0.021961	1.45	3.13
SOAP3	0.058407	1.78	0.09

measured against **SOAP3**, because the latter cannot do paired-end mode. In single-ended mode, *Bowtie* takes more than double the time as in paired-end mode, but half the virtual memory. On the other hand, the memory requirement for the GPU enabled version is significantly higher than **CUSHAW**, which is performing alignment in paired-end mode (two input files rather than one). From these results, we can see that our target performance is the combination of both GPU enabled applications. For execution time we need to design our implementation using the insight gained from **SOAP3**, but our memory utilization target is to beat **CUSHAW**. Therefore, we implemented **SOAP3's** memory coalescing and elimination of diverging branches to implement the GPU-version of global alignment in *Bowtie*.

GPU-implementation of global alignment of Bowtie: In this project, we have implemented the GPU version of Needleman-Wunsch algorithm [19] in *Bowtie*. *Bowtie* represents the reference genome as suffix trees for data compression, which requires a backtracking algorithm to align short reads with a reference genome. In our GPU implementation, we did not keep the reference genome in compressed format as backtracking algorithms usually cannot be efficiently implemented on GPU. Hence, we tested the performance of Needleman-Wansch algorithm of *Bowtie*, keeping all other functionalities unchanged framework, for the CPU as well as GPU versions. Moreover, we varied the parameters like block size and number of threads in each block to analyze their impact on the performance of the alignment procedure. Fig. 7 shows a profiling trace session of our GPU implementation using VisualProfiler for 15 consecutive runs identifying CUDA API tracing and GPU time of the memory (pageable) data transfers from host to device.

Effect of threads: Increasing the number of threads per

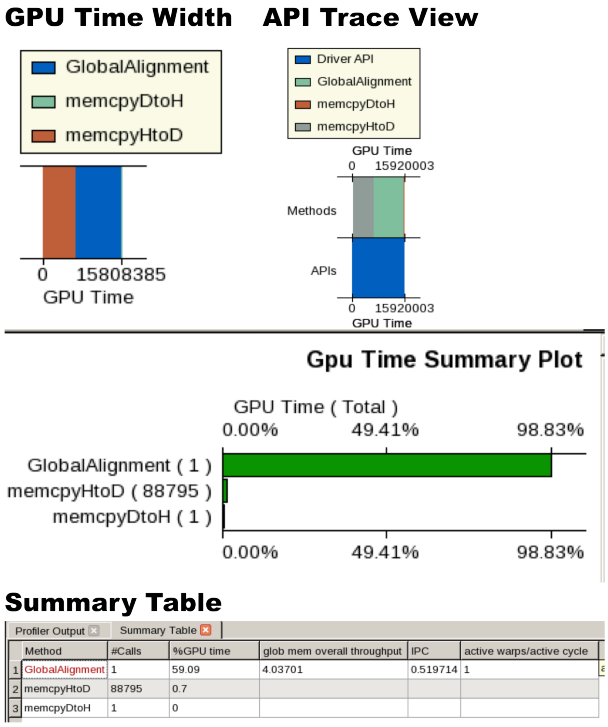


Fig. 7: NVIDIA VisualProfiler tracing of pageable data transfer from host to device and vice-versa including a summary table and timing plots of CUDA objects GPU profile for our basic implementation.

block helps improving the computation time of Needleman-Wansch algorithm until a certain threshold, beyond which we might not be able to achieve a performance gain. Fig. 8 depicts the performance of GPU implemented Needleman-Wunsch algorithm running on 10,000 short reads using 16 blocks and varying number of threads. This is because excessive threading causes the overhead of additional memory contention. As showed, for our dataset, we achieve the best performance by using 128 threads per block.

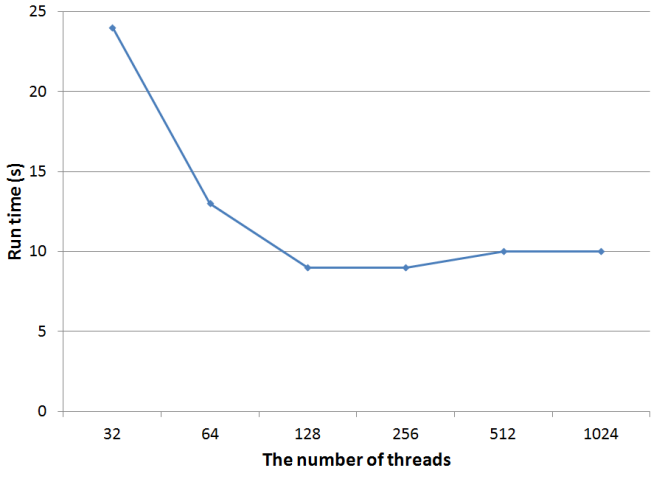


Fig. 8: The performance of Needleman-Wunsch algorithm running on various numbers of threads.

Effect of blocks: Increasing the number of blocks helps improving the computation time of Needleman-Wunsch algorithm. As with the case of varying number of threads per block, using a large number of blocks do not necessarily lead to better performance. Fig. 9 shows the performance of GPU implemented Needleman-Wunsch algorithm running on 10,000 short reads using 128 threads per block and varying number of blocks. Although we proved using 64 blocks would enable best performance, the performance improvement is negligible when we used more than 16 blocks. This is because utilizing several blocks would entail the overhead of distributing the job among several GPU cores and then retrieving the results from individual core back to the host. This justifies our use of 16 blocks and 128 threads per block while executing the GPU implemented Needleman-Wunsch algorithm.

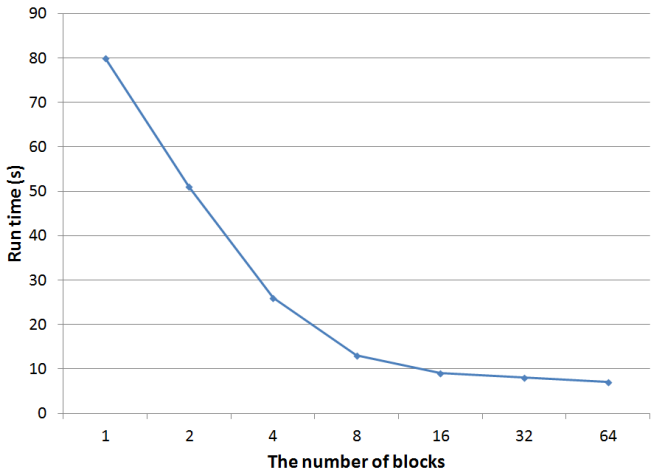


Fig. 9: The performance of Needleman-Wunsch algorithm running on various number of blocks.

Effect of the number of short reads: We executed both the CPU and GPU implemented Needleman-Wunsch algorithm on various number of short reads to compare their relative performance. We executed the GPU version on the combination of number of blocks and threads per block, as proposed above. Fig. 10 It is evident that the GPU implementation outperforms the CPU implementation by 17x. We could achieve this significant performance gain by parallelizing the alignment of the short reads against the reference sequence by letting them run in individual cores and threads of GPU. In this regard, an optimized selection of block number and threads per block further proved to be advantageous. Our implementation of memory coalescing eliminated the issue of random memory access. By storing the short reads in the global memory in 32 blocks, we ensured that during alignment, reads are fetched from consecutive memory location, thus reducing the overhead in time caused by random memory access. Varying number of blocks and threads per block consolidated our idea of implementation. Accessing reads from consecutive locations also eliminated the issue of diverging branches. Thus, we did not have to wait until runtime to resolve these branches, that also helped improve overall computation time.

Hence, resolving the challenge of random memory access and diverging branches through memory coalescing and usage of appropriate number of blocks and threads per block, we enabled a considerable speedup for our GPU-implementation of **Bowtie's** global alignment procedure.

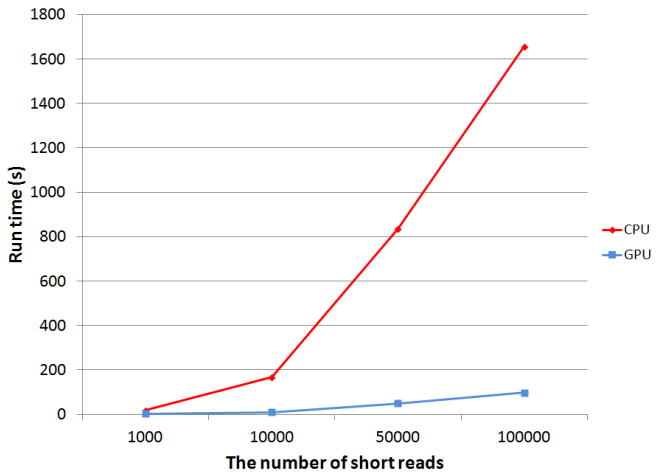


Fig. 10: The performance of Needleman-Wunsch algorithm running on various numbers of blocks.

V. Conclusions

A. Discussion

In this project we laid the foundation for implementing a CUDA-based version of *Bowtie*. We identified the main challenges in exploiting the power of GPUs for short read alignment and analyzed them from the hardware perspective. We proposed and discussed solutions to them. To support our conclusions, we have performed a number of benchmarking tests of both the original and modified versions of our alignment implementation. This helped us understand the underlying challenges and the validity of our approach.

Considering the time constraints and scope of the project, we mainly focused on the issues of random memory access and diverging branches. We implemented our approach in CUDA and analyzed it in terms of computation time and memory utilization, and compared against its CPU counterpart. The experimental results indicate that our CUDA-enabled alignment achieved a performance gain of 17x in terms of overall alignment time over the CPU-based method. Furthermore, analyzing the results obtained by varying parameters (for example, number of threads and blocks) of our implementation helped us to better determine their impact and identify the most effective combination that maximizes performance.

B. Advantages and Limitations

It is important to consider the limitations of the proposed approach. Due to the nature of short read alignment and the specific algorithmic implementations used by the tool, the potential speedup of *Bowtie* achievable by using GPU programming is limited. In particular, the major concern is the pattern of searching and data transaction while using BWT. Although memory coalescing might involve the issue of memory traffic, the high performance gain overpowers it. Moreover, the use of backtracking by *Bowtie* also limits the potential speedup from using GPU programming approach.

Yet, as our experimental results indicate, *Bowtie* possess a considerable amount of parallelism which can be effectively exploited using CUDA. The analysis of the existing CUDA-based aligners suggests that GPUs can still significantly improve performance of short read alignments. In addition to this, analysis of our implementations also point towards positive effect of using CUDA on *Bowtie* performance. One of our main targets in implementing CUDA-based alignments was to achieve effective memory access patterns, since this, as we have seen, has an impact on performance. By changing memory access pattern, we were able to achieve substantial speedups by using GPUs over corresponding CPU implementations.

C. Future directions

In our implementation, we have primarily focused on coalescing memory accesses and avoiding diverging branches. While addressing these issues has a huge impact on the overall performance, there are also other ways to further improve it. For example, sampled data structures can help to reduce memory traffic and using load balancing can help with effective backtracking. Moreover, in this work we have used only shared and global memory leaving us to wonder if utilization of other types of memory, such as those available on CUDA-enabled GPUs (e.g. constant or texture memory) could bring positive results. It would be also interesting to consider some of the new features available in CUDA 5 and new generation hardware such as dynamic parallelism, which is an extension of the CUDA programming model that enables kernels to create and synchronize with new work directly and in turn reducing the need to transfer execution control between host and device [20].

References

[1] J. S. Reis-Filho, "Next-generation sequencing," *Breast Cancer Research*, vol. 11, pp. 1–7, 2009. [Online]. Available: http://dx.doi.org/10.1186/bcr2431

[2] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome," *Genome Biology*, vol. 10, no. 3, p. R25, 2009. [Online]. Available: http://genomebiology.com/2009/10/3/R25

[3] Y. Liu, B. Schmidt, and D. L. Maskell, "CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows-Wheeler transform," *Bioinformatics*, vol. 28, no. 14, pp. 1830–37, 2012.

[4] C.-M. Liu, T. Wong, E. Wu, R. Luo, S.-M. Yiu, Y. Li, B. Wang, C. Yu, X. Chu, K. Zhao, R. Li, and T.-W. Lam, "SOAP3: ultra-fast GPU-based parallel alignment tool for short reads," *Bioinformatics*, vol. 28, no. 6, pp. 878–879, 2012.

[5] J. Blom, T. Jakobi, D. Doppmeier, S. Jaenicke, J. Kalinowski, J. Stoye, and A. Goesmann, "Exact and complete short read alignment to microbial genomes using GPU programming," *Bioinformatics*, vol. 27, no. 10, pp. 1351–58, 2011.

[6] W. D. Burrows M, "A block sorting lossless data compression algorithm," Technical Report, Digital Equipment Corporation, 1994.

[7] E. Siragusa, D. Weese, and K. Reinert, "Fast and sensitive read mapping with approximate seeds and multiple backtracking," *arXiv preprint arXiv:1208.4238*, 2012.

[8] R. Li, Y. Li, K. Kristiansen, and J. Wang, "SOAP: short oligonucleotide alignment program," *Bioinformatics*, vol. 24, no. 5, pp. 713–714, 2008.

[9] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang, "Soap2: an improved ultrafast tool for short read alignment," *Bioinformatics*, vol. 25, no. 15, pp. 1966–1967, 2009.

[10] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with Bowtie 2," *Nature Methods*, vol. 9, no. 4, pp. 357–359, 2012.

[11] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," 2009.

[12] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2011.

[13] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[14] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros, "A massively parallel adaptive fast multipole method on heterogeneous architectures," *Commun. ACM*, vol. 55, no. 5, pp. 101–109, may 2012. [Online]. Available: http://doi.acm.org/10.1145/2160718.2160740

[15] NVIDIA, "NVIDIA GeForce GTX 680," 2012.

[16] Y. Liu and B. Schmidt, "Evaluation of gpu-based seed generation for computational genomics using burrows-wheeler transform," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, 2012, pp. 684–690.

[17] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 2008, pp. 73–82.

[18] J. Jenkins, I. Arkatkar, J. D. Owens, A. Choudhary, and N. F. Samatova, "Lessons learned from exploring the backtracking paradigm on the GPU," in *Euro-Par 2011 Parallel Processing*. Springer, 2011, pp. 425–437.

[19] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, 1970.

[20] M. Harris, "CUDA 5 and Beyond," 2012.