2010 13th IEEE International Conference on Computational Science and Engineering

GPU-RMAP: Accelerating Short-Read Mapping on Graphics Processors

Ashwin M. Aji, Liqing Zhang and Wu-chun Feng
Department of Computer Science
Virginia Tech
Blacksburg, Virginia 24060, U.S.A
{aaji, lqzhang, feng}@cs.vt.edu

***Abstract*—Next-generation, high-throughput sequencers are now capable of producing hundreds of billions of short sequences (reads) in a single day. The task of accurately mapping the reads back to a reference genome is of particular importance because it is used in several other biological applications, e.g., genome re-sequencing, DNA methylation, and ChiP sequencing. On a personal computer (PC), the computationally intensive short-read mapping task currently requires several hours to execute while working on very large sets of reads and genomes. Accelerating this task requires parallel computing. Among the current parallel computing platforms, the graphics processing unit (GPU) provides massively parallel computational prowess that holds the promise of accelerating scientific applications at low cost.**

In this paper, we propose GPU-RMAP, a massively parallel version of the RMAP short-read mapping tool that is highly optimized for the NVIDIA family of GPUs. We then evaluate GPU-RMAP by mapping millions of synthetic and real reads of varying widths on the mosquito (Aedes aegypti) and human genomes. We also discuss the effects of various input parameters, such as read width, number of reads, and chromosome size, on the performance of GPU-RMAP. We then show that despite using the conventionally "slower" but GPU-compatible binary search algorithm, GPU-RMAP outperforms the sequential RMAP implementation, which uses the "faster" hashing technique on a PC. Our data-parallel GPU implementation results in impressive speedups of up to 14.5-times for the mapping kernel and up to 9.6-times for the overall program execution time over the sequential RMAP implementation on a traditional PC.

***Keywords*-short-read mapping; sequence analysis; graphics processing unit (GPU); RMAP; CUDA.**

I. Introduction

Next-generation high-throughput sequencing instruments, like Illumina's Solexa IG sequencer, Applied Biosystem's SOLiD system, and Roche's (454) GS FLX Genome Analyzer, are capable of producing billions of short sequence data (about 25-100 bases each) within a single day. In the field of genomics, a very important problem is to map these short sequences, or *reads*, to a reference genome. This highly compute-intensive mapping phase forms an integral link in the computational pipeline of several applications, such as genome re-sequencing, DNA methylation, transcriptome sequencing, and ChiP sequencing [1], [2]. Mapping billions of reads to huge reference genomes, like the human genome that has about 3-billion bases, may take several hours if executed sequentially on a personal computer (PC). While traditional local sequence alignment tools, e.g., BLAST [3] and PatternHunter [4], can be used to map reads, they are *not* optimized to align a large number of very short reads. Many specialized tools have been developed to solve the short-read mapping problem [2], [5]–[7], but the rate of data analysis is much slower than the unprecedented rate at which the sequence data is being generated. This has led to the development of accelerated parallel versions of the short-read mapping tools on a multitude of high-performance computing platforms [1], [8], [9].

Today, gains in computational horsepower are no longer driven by increases in clock speeds. Instead, the gains are increasingly achieved through parallelism, both in traditional multi-core architectures as well as the many-core architectures of the graphics processing unit (GPU). Amongst the most prominent many-core architectures are the GPUs from NVIDIA and AMD/ATI, which can support general-purpose computation on the GPU (GPGPU). Thus, GPUs have evolved from their traditional roots of graphics pipeline models into programmable devices that are suited for accelerating scientific applications, such as sequence alignment and fast N-body simulations [8], [10]–[12], at very affordable prices. Further speedups can be achieved by adding the GPU to an existing personal computer (PC).

In this paper, we introduce and evaluate *GPU-RMAP*, a massively parallel version of the RMAP short-read mapping tool [2] that we have designed and optimized for the GPU thread and memory architecture of NVIDIA Tesla via CUDA programming platform [13]. In this design, each GPU thread efficiently maps the given set of reads to a unique genome segment and collectively returns all the chosen reads and the mapped genome sites. Race conditions arise when multiple GPU threads update the map results simultaneously, but we explain how our multi-staged algorithm prevents this classic problem from occurring. In addition, we show that despite using the conventionally "slower" binary search method at the core of the GPU mapping algorithm, GPU-RMAP outperforms the sequential RMAP implementation, which uses the "faster" hashing technique for mapping reads on a PC. We also present further optimizations, where we explicitly cache the repeatedly accessed data elements in the faster on-chip memory of the GPU.

978-0-7695-4323-9/10 $26.00 © 2010 IEEE
DOI 10.1109/CSE.2010.29

We evaluate GPU-RMAP by performing a wide range of experiments, followed by rigorous data analysis. We map millions of real and artificially generated reads of varying widths onto the mosquito (Aedes aegypti) and human genomes. We then profile the GPU-RMAP code and discuss the effects of the chosen optimization techniques on the different phases of the RMAP algorithm. Next, we present the effects of RMAP's various input parameters (e.g., read width, number of reads, and chromosome size) on the performance of GPU-RMAP. Finally, we show that all the efficient design considerations of GPU-RMAP result in impressive speedups of up to $14.5\times$ for the mapping kernel and up to $9.6\times$ for the overall program execution time over the sequential RMAP implementation on a traditional PC.

The rest of the paper is organized as follows: Section II presents the related work. Section III describes the NVIDIA Tesla C1060 architecture and the CUDA programming model. Section IV introduces the sequential RMAP algorithm. Techniques to accelerate RMAP on the GPU using CUDA are described in Section V. Section VI compares and analyzes the performance of the GPU-RMAP. Section VII concludes the paper.

II. Related Work

Recent parallel short-sequence mapping has been carried out on 64-node clusters [1], where the authors have analyzed the difference in the performances between multiple parallelization techniques, including partitioning the reads, partitioning the genome, and partitioning both the reads and the genome. While their performance results demonstrate good scalability of their code, the hardware is expensive and not largely available to the bioinformatics community. Moreover, they have parallelized their own SOLiD algorithm that performs short sequence mapping by using covering designs. In contrast, we have improved the performance of the RMAP algorithm, which claims to be more useful than other short sequence-mapping tools because of its ability to map paired-end reads and bisulfite-treated reads [2]. Also, we have implemented GPU-RMAP on commodity hardware, i.e., a desktop PC with a graphics card (containing a GPU), thus providing a very inexpensive and massively parallel computing platform for the short-read mapping problem.

MUMmerGPU v1.0 [8] and v2.0 [14] parallelize the MUMmer short sequence-mapping program on the CUDA programming platform and report total application speedups of $3.5\times$ and $13\times$, respectively, over a CPU implementation. MUMmer represents the target genome as a suffix tree and maps the incoming input reads, while RMAP uses a hash table of the reads to quickly match various genome segments. The authors of MUMmer and RMAP claim several advantages of one tool over another, and clearly, both of these tools are very much relevant to the bioinformatics community. To the best of our knowledge, no other literature presents the parallelization of RMAP on commodity graphics processors.

CloudBurst [9] is a parallel implementation of RMAP on distributed memory architectures that uses Google's MapReduce [15] framework. This program achieves more than $100\times$ speedup by executing on a remote compute cloud with 96 cores. As previously mentioned, our program just uses a single commodity GPU for accelerating RMAP, and therefore, we achieve a much better performance-cost ratio. Moreover, CloudBurst parallelizes an older version of RMAP, where the reads are partitioned into multiple *seeds* corresponding to the number of allowed mismatches. We have chosen to accelerate a newer version of RMAP, where *layered seeds* are used for maximum search sensitivity, while maintaining good execution times [16].

III. NVIDIA GPU Architecture and the CUDA Programming Model

The NVIDIA Tesla C1060 GPU (or *device*) consists of a set of 30 single-instruction, multiple-data (SIMD) streaming multiprocessors (SMs), where each SM consists of eight scalar processor (SP) cores running at 1.2 GHz with 16-KB on-chip shared memory (cache), and a multi-threaded instruction unit. The SMs on the GPU can simultaneously access the *device memory*, which consists of 4 GB read-write global memory, 64 KB of read-only constant memory, and read-only texture memory. However, all the device memory modules can be read or written to by the *host* processor. Each SM has on-chip memory, which can be accessed by all the SPs within the SM and will be one of the following four types: a set of registers; 16 KB of 'shared memory', which is a software-managed data cache; a read-only constant memory cache; and a read-only texture memory cache. The global memory space is not cached by the device.

CUDA (Compute Unified Device Architecture) [13] is the parallel programming model and software environment provided by NVIDIA to run applications on their GPUs. It abstracts the architecture to parallel programmers via simple extensions to the C programming language. CUDA follows a code off-loading model, where compute-intensive portions of applications that normally run on the host processor are off-loaded onto the GPU device. The *kernel* is the portion of the program that is compiled to the instruction set of the device and then off-loaded to the device before execution.

IV. The RMAP Package

The RMAP program accurately maps reads from next-generation sequencing technology. Although RMAP was originally designed for mapping Illumina reads, it can also be used to map Roche/454 and ABI SOLiD reads [2]. The typical inputs to the RMAP program are (1) a set of millions of short reads and (2) the target genome or a file containing the path to a set of target genomes (FASTA format). The reads have to be quickly mapped onto different regions of

the target genomes. The output of the program is a set of mapped reads, the sites on the target genome at which each of them is best mapped, and the strand (forward / reverse) of the genome. RMAP also allows the end user to configure some execution parameters, including the maximum read width, number of mismatches, and number of non-unique (ambiguous) reads. In addition, there are no limitations on any of the parameters. Currently, the RMAP package (v2.02) contains three mapping programs, where each program pre-processes the set of reads and creates a hash structure, and then scans the genome to find potential high-scoring maps by doing a hash table lookup. Apart from the basic `rmap` program, the RMAP package contains the `rmappe` and the `rmapbs` programs for mapping 'paired-end' reads and bisulfite-treated reads, respectively.

The reads for each of the above programs can be specified in their entirety (FASTA format) or can be specified as a set of quality scores for each position within the read. These quality scores are derived from some probabilistic model, which allots a certain confidence level for finding a particular base-pair in the corresponding position of the read.

In this paper, we parallelize the basic `rmap` program and choose the FASTA format as the input for the reads and the genomes. We note that our design principles can also be directly applied to the other programs in the RMAP package.

A. The Sequential Algorithm

The RMAP algorithm can be broadly thought of as a methodology to match a set of patterns (reads) onto a large text (genome). The algorithm first indexes the reads by creating a hash table.[1] Collisions in the hash table are resolved by chaining, where the chain indicates the set of reads that have resolved to the same hash key. Then, the genome is scanned at every site to check for any match in the hash table. If there is a match, then the genome segment is *scored* against all the reads that correspond to the respective hash table entry. If the score is within a certain threshold (constrained by the number of allowed mismatches and non-unique mappings), the read, along with the mapped genome site and the genome strand (forward / reverse) are added to the final map (i.e., results).

To make this process more efficient, RMAP v2.02 employs the concept of *layered seed structures* [16] for constructing the hash table. Seed structures specify sets of locations in the reads that are required to match the genome exactly at any site where the read can map. In other words, the seeds can be considered to be bit masks that have to be applied (i.e., bitwise AND'ed) to the reads before adding them as keys to the hash table. The result of applying the seed to the read will be a 64-bit unsigned integer, with the bases of the read corresponding to $1's$ in the seed. Multiple reads can therefore produce the same hash key, and collisions in the hash table are resolved by chaining, where the chain indicates the set of reads that have the same bases at the positions determined by the seed structure. The same seed (bit mask) must then be applied to all the sites of the genome, when scanning them for matches in the hash table. If we choose more seeds, then the sensitivity of the algorithm will be higher, but too many seeds or a bad choice of seed structures can potentially hurt the execution time of the program. However, the seeds are designed to be more accurate in RMAP, and therefore, there will be fewer full comparisons while scanning the genome [16].

In summary, the execution of `rmap` can be broadly characterized to consist of the following phases:

- Initialization – read all the input reads and genome data from disk, process the command line parameters, initialize the seed structures and the final result structures.
- Hash table construction – apply the seed structure to the set of reads before creating the hash table.
- Genome mapping – scan the genome and lookup the hash table for matches, calculate the scores and choose the highest scoring reads.
- Best maps reporting – collect and present the final mapped reads and the respective mapped sites, chromosome strands and the scores.

The hash table construction and genome mapping steps are carried out for every seed in the seed structures. In this paper, we accelerate the genome mapping step because it consumes more than 98% of the total execution time, and hence, the code section to be parallelized.

V. GPU-RMAP: Design and Implementation

In this section, we describe the techniques used to accelerate the RMAP algorithm on the NVIDIA Tesla C1060 GPU by using the CUDA programming platform. As discussed in the previous section, we focus on parallelizing only the *genome mapping* part of the algorithm because it is the most computationally expensive portion of the program. Genome mapping includes the following phases: (1) *Scanning and hash table lookup:* The entire genome and the hash (lookup) table for the reads are transferred from the CPU memory to the device memory of the GPU. Next, all the segments of the genome are inspected by the GPU threads and the potential matches in the hash table are found; (2) *Scoring:* The number of mismatches (score) between the genome segment and the corresponding matched reads is computed by the GPU threads; and (3) *Selection:* For every read, the best set of mapped genome sites, the corresponding scores, and the strand information are added to the final map result structure on the GPU (filtered by a pre-determined score threshold). The final map results are transferred back to the CPU memory for further processing or reporting.

[1] Reads are indexed, and not the genome, probably because the genome index structure would have been a few orders of magnitude larger than that of the reads. The RMAP authors have also likely chosen to optimize memory in the "memory vs. time" tradeoff.

A. High-Level Mapping of RMAP onto the GPU

Figure 1 depicts the high-level design of GPU-RMAP. After the genome and the lookup table data is transferred to the GPU's device memory, the genome is partitioned into several independent segments that marginally overlap each other. These segments are then distributed for processing among all the threads on the GPU so that each thread can independently (i.e., in parallel) map the complete set of reads onto a unique portion of the genome.

However, given that each thread independently processes a unique portion of the genome, the set of reads is common to all the GPU threads. Consequently, different threads may map their genome segments to the same read at the same time, and thus, may produce different 'best' scoring sites for the same set of reads. The selection of the best-scoring genome sites for each read will therefore depend on the order in which the threads execute, thus leading to a classic *race condition*, as shown in Figure 1. Moreover, CUDA does not yet fully support global inter-thread communication [13],[2] which could have potentially solved this problem.

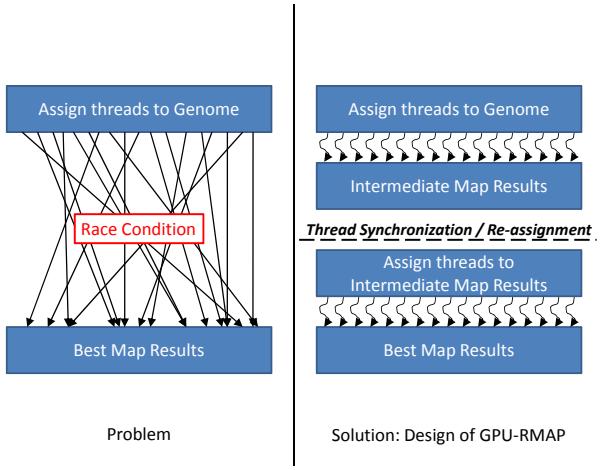
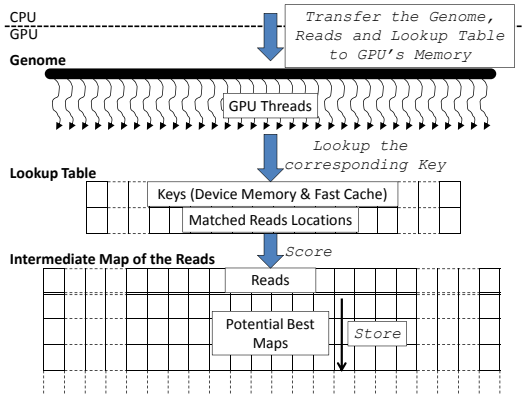


Figure 1. GPU-RMAP: High-Level Design.

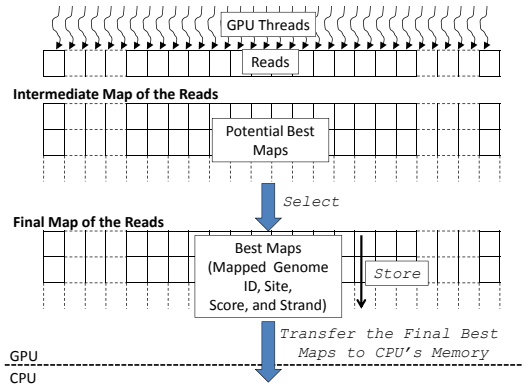
To address the race condition, we divide the algorithm into two stages, where the first stage performs the parallel genome scanning and scoring and the second stage does the selection. At the end of the first stage, each read entry has a list of *potential* 'best map' genome sites, which represent the the intermediate map results. Next, we synchronize all the threads, followed by re-distributing the intermediate results among the GPU threads for the second stage, i.e. the final selection phase, as shown on the right-hand side of Figure 1. The thread "synchronization / re-assignment" step solves the race condition problem because we have decoupled the dependency of the final selection phase from the execution order of the GPU threads in the parallel genome scanning and scoring phase.

Global thread synchronization can be implicitly implemented in CUDA by launching a new CUDA kernel. So, in GPU-RMAP, we launch *two* CUDA kernels, as shown in Figure 2.

As shown in Figure 2a, *Kernel_1* executes the scanning, table lookup, and scoring phases. Each thread independently chooses the set of reads that could be potentially mapped to the genome segment that is assigned to the thread. It then appends the score and mapped site information to the list of *potential* best maps, corresponding to each chosen read. The `atomicAdd` instruction, which is provided by the CUDA SDK, allows each thread to calculate the next available index of the list in a thread-safe manner, where the intermediate scores and potential mapped genome sites of the respective reads can be stored. Each read still needs to inspect its intermediate list of potential best maps and make a final selection of the best-mapped genome sites. The termination of the first kernel acts as an implicit synchronization point for all the GPU threads, and further processing can be safely done by the second kernel without any race conditions.



(a) Kernel_1



(b) Kernel_2

Figure 2. GPU-RMAP: Detailed Design.

[2]CUDA only supports a basic set of atomic operations, which we have used in our design.

Figure 2b shows that the second kernel performs the selection phase by distributing all the *reads* among the GPU threads for processing. Each thread independently inspects (i.e., in parallel) the list of potential sites (obtained from *Kernel_1*), corresponding to the respective reads, and chooses the best-mapped sites and scores. There cannot be any race conditions with this approach because the best-map results for each read are collected independently by a different thread.

B. Lookup Table Optimization: Hashing vs. Binary Search

The sequential RMAP algorithm uses the C++ Standard Template Library's (STL) `unordered_multimap` data structure, which implements a hash table to store and lookup the keys. The hash table of the `unordered_multimap` allows multiple keys to be grouped together without sorting them. STL also implements a very efficient hash algorithm for the quick insertion and retrieval of the key-value pairs into the hash table. While there exist hashing implementations designed for the GPU [17], [18], they aim to accelerate a single instance of the hashing problem, i.e., they use all the threads on the GPU to lookup a *single key* in the hash table efficiently. On the other hand, our design targets enhancing the throughput of search by performing a coarse-grained parallel execution, where each thread on the GPU should independently search for a different key in the table. Moreover, an inefficient hashing function will result in a worst-case key insertion or retrieval time of $O(n)$, where n is the number of keys in the hash table, thus resulting in larger number of memory accesses on the GPU, which in turn, results in terrible performance if the access patterns do not follow a set of alignment rules [13].

In GPU-RMAP, we modified RMAP and replaced the `unordered_multimap` with the `multimap` data structure, where the keys are all *sorted* and similar keys are obviously grouped. Adding the keys to an ordered map is marginally more expensive than adding the keys to an unordered map on the CPU. However, upon transferring the keys to the GPU's device memory, the lookup operation on the keys can be done by using a simple binary-search algorithm, where the lookup time will always be logarithmic to the table size. In section VI, we show that this initial optimization of GPU-RMAP, which uses binary search, produces an impressive mapping speedup of $7\times$ to $10.5\times$ over the sequential RMAP implementation on PC, which uses hashing as its primary search algorithm. While binary search ($O(\log n)$) is a *slower* search algorithm on the CPU than the conventional hashing technique ($O(1)$), we show that the binary-search algorithm is a much better choice for the GPU.

Further Optimization using Faster Cache Memory.: All the GPU threads perform binary search repeatedly over the same binary search tree of the keys. So, the top few levels of the binary search tree will be accessed multiple times throughout the execution of table lookup kernel, (i.e., *Kernel_1*). We used this knowledge and made an additional optimization by moving the top few levels of the search tree to the faster cache (shared) memory of each processing unit on the GPU, as explained in Figure 3. However, the current NVIDIA GPUs only have a cache capacity of 16 KB, and therefore, we could transfer only about 10 to 15 levels of the search tree to fit into the cache.

Accessing the GPU cache consumes approximately the same number of clock cycles as accessing a register, whereas an access to the device memory takes 400 to 600 GPU clock cycles. Our optimization method reduces several slower device memory accesses and produces a mapping speedup of up to $14.5\times$ over the sequential RMAP implementation on the CPU. Additional details are provided in Section VI.

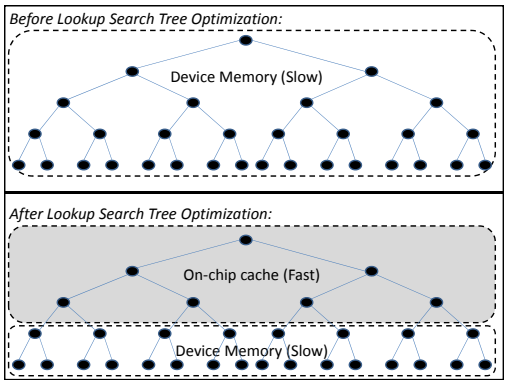


Figure 3. GPU-RMAP: Optimization of the keys' Lookup Table (search tree) using the Faster Cache.

VI. Performance Analysis and Discussion

We first explain our experimental setup, followed by a detailed discussion about the performance of GPU-RMAP.

A. Experimental Setup

For our experiments, we chose a simulated set of reads that were generated by taking random segments of predefined widths from existing human chromosomes and randomly modifying some of the bases within them. We used read widths of 32, 48 and 64 base-pairs (bp) and generated sets of 1, 2, 4 and 8 million reads for each read width, resulting in a total of *twelve* sets of reads as input data. We also used a real set of reads for our experiments but found that the trends and results were similar to the synthetic reads. Hence, we do not present those results in this paper for brevity.

For the target genomes, we chose to map all these reads onto (1) the human genome, which is made up of 23 chromosomes, totaling around 3 billion bp, and (2) the available mosquito genome (*Aedes aegypti*), which currently has around 1.4 million bp sequenced. We downloaded the human genome from

http://hgdownload.cse.ucsc.edu and approximately 4800 supercontigs of the Aedes aegypti genome assembly from http://aaegypti.vectorbase.org. Because the complete mosquito genome is not available and contains three chromosomes, we concatenated the supercontigs and generated three chromosomes of fairly equal sizes. Our experimental genome data thus included 23 chromosomes from the human genome and 3 chromosomes from the mosquito genome. Moreover, to analyze the effect of the chromosome size / length on the performance of GPU-RMAP, we performed additional experiments and mapped the same set of reads on the following chromosomes of the human genome: chromosome 1 (chr1), which has about 250 million bp, chromosome 12 (chr12), which has about 135 million bp and chromosome 22 (chr22), which has about 50 million bp. These chromosomes display the required variety in size / length.

RMAP maps only nucleotide reads, which can be represented by the alphabet set {A, C, G, T, N[3]}. The RMAP tool encodes each nucleotide base by using a structure of 3 bits (for the 5 characters). So, any read with the width up to 64 bp will be stored in the computer as a structure of three 64-bit words. This storage mechanism proves to be very efficient for the scoring phase (to count the number of mismatches between the chromosome segment and the read), where the score can be computed in time logarithmic to the read length by using a series of bitwise operations, as described by [19]. However, if the read width is greater than 64, each read should be represented by a list of 64-bit words. This means that more computation is needed to convert each read into this format, and hence, the overall speedup may be reduced. For all our experiments, we have chosen to map the reads with width not greater than 64 because we can easily store each read as only three 64-bit words on the GPU. Maintaining a list of 64-bit words on the GPU to represent larger read widths proves to be computationally expensive on the GPU.

Our parallel execution environment for running all of the above experiments was the NVIDIA Tesla C1060 GPU with CUDA v2.3 as our programming interface to the GPU. Our results are shown for the best possible CUDA execution configuration, where the mapping kernels have 480 blocks of 256 threads running across the entire GPU. The Tesla GPU card has 4-GB global device memory and each of its cores runs at 1.2 GHz. The above GPU was placed in a PC that contained an AMD quad-core processor (each core running at 1.2 GHz) as the host CPU. The host CPU had 8-GB RAM. All the chosen sets of reads and genomes fit well into both the host and device memory. We ran the sequential implementation of RMAP, with which we compared the results of GPU-RMAP, on one of the cores of the host AMD processor.

[3] 'N' stands for an unknown base or error.

B. Results and Discussion

1) Analysis of Speedup: Figure 4 shows the RMAP execution time being partitioned into its different phases and compared to the sequential RMAP algorithm. This figure shows a couple of our several experimental runs, where we map 1-million reads on the entire mosquito genome and map 8-million reads on the complete human genome. Also, we compare performance by varying the read widths with 32, 48 and 64 bp. Although we have run this experiment for all the reads-genomes combinations, we present only the above results in this paper due to space constraints. In the examples shown, we show that the mapping phase has been accelerated by about $9.2\times$ – $14.5\times$, and the total execution time has been improved by $5.5\times$ – $9.6\times$, when compared to the sequential RMAP.

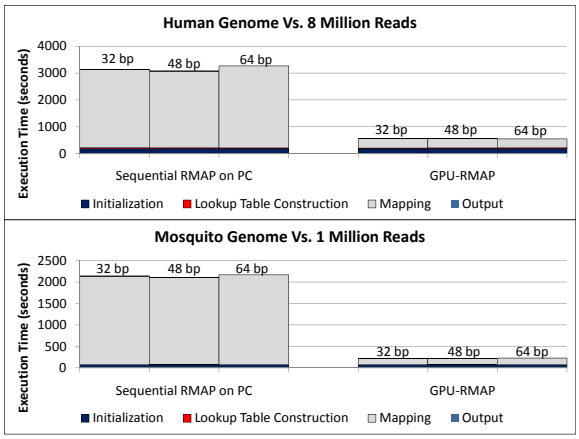


Figure 4. GPU-RMAP vs. Sequential RMAP: Analysis of Speedup. Note: The Lookup Table Construction and the Output generation times are negligible when compared to the overall execution time of the program, and hence, they may not visible in the chart.

Effect of the Fast Cache Memory Optimization: Figure 5 presents the partition of the execution times of the sequential RMAP on PC, along with GPU-RMAP with and without the faster cache memory optimization (section V-B). This particular experiment maps 1-million 64-wide reads on the chromosome 1 of the human genome. The figure shows that the mapping speedup increases from $10.53\times$ to $13.82\times$, and the total program execution improves from $5.93\times$ to $6.77\times$ over the sequential RMAP execution on PC. The initialization phase of RMAP (reading the reads and the chromosome from the disk, and allocating the required memory) is unavoidable and cannot be parallelized. Moreover, the initialization phase takes a substantial chunk of the overall execution time in all the versions of RMAP and GPU-RMAP, and therefore, the total program speedup is measurably less than the mapping speedup alone. The figure also shows that the design of GPU-RMAP is highly efficient, because the most significant amount of the mapping time is taken up by the execution of the CUDA kernels,

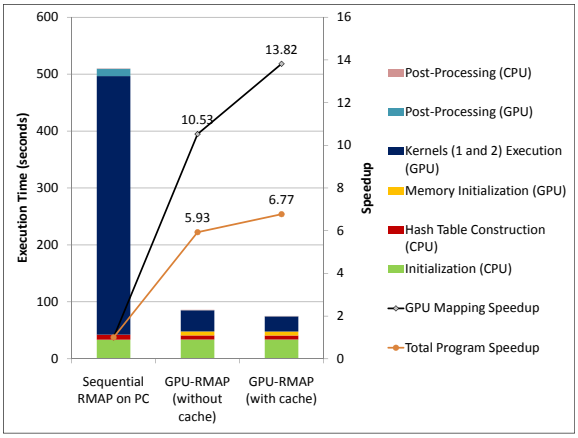


Figure 5. Code Profile of GPU-RMAP: Effect of Fast Cache Memory Optimization (`chromosome 1` Vs. 1 million 64-wide Reads).

while the GPU memory initialization, GPU-to-CPU memory transfers, and the miscellaneous post-processing phases are quite negligible.

2) Effect of Read Width: All the charts in the Figure 6 show the impact of the read width on the mapping phase and the overall execution speedup of GPU-RMAP. The speedup values do *not* change when the width of the reads changes because the computational overhead for scoring the chromosome segments against the reads will be not change if the read width is < 64 bp. If we had used larger-sized reads, then we could expect a decrease in performance because of the more complicated storage overhead, as explained in Section VI-A. However, since the reads generated by the modern sequencing instruments are very short (typically 25-100 bases each), we believe that the above limitation is a reasonable one.

3) Effect of Chromosome Size: Figure 6a shows the impact of the chromosome size on the mapping speedup of GPU-RMAP. The speedup remains almost the same for all the chromosome sizes but is less for `chr22` (smallest in size) and the entire human genome (largest in size). The `chr22` is very small to keep all the GPU threads busy, i.e., GPU resources are wasted when trying to map smaller chromosomes. On the other hand, the entire human genome contains several chromosomes. The additional overhead of repeatedly moving different chromosomes to and from the GPU device memory thwarts the performance of GPU-RMAP when we map the reads against the whole human genome.

4) Effect of Number of Reads: Figures 6b and 6c show the impact of the number of reads on the mapping speedup of GPU-RMAP on the complete human genome and the mosquito genome, respectively. While the execution time increases for larger number of reads (data not shown for brevity), the speedup factor decreases for larger number of reads. This means that when the number of reads increases, the table lookup time (binary search) in the GPU-RMAP code increases *at a higher rate* than the table lookup time (hashing) in the sequential RMAP code. While the above results may indicate that the binary search implementation on the GPU is a potential bottleneck for larger sets of input data, GPU-RMAP (using binary search) shows speedups of up to $14.5\times$ over the sequential RMAP implementation (using hashing) on a PC.

VII. Conclusions

In this paper, we accelerated an extremely popular short-read mapping application called RMAP onto the GPU. To demonstrate how we achieved up to $14.5\times$ mapping speedup and $9.6\times$ total program execution speedup over the sequential RMAP implementation on a traditional PC, we presented a detailed design of GPU-RMAP, along with an efficient fast cache memory optimization.

We then performed a detailed experimental analysis by mapping millions of reads of different widths on the mosquito and the human genomes. We discussed in detail about the effects of various input parameters on the performance, like the read width, genome size, and number of reads. Next, we presented and discussed the profiled code and the speedups achieved by the different phases of the GPU-RMAP algorithm and show that our design decisions are very efficient. In addition, we presented our idea of dividing the RMAP algorithm into two stages to avoid the classic race condition problem, so that correctness is ensured. We also show that despite using the conventionally *slower* binary search algorithm, GPU-RMAP out-performs the sequential RMAP implementation, which uses the *faster* hashing technique on a PC.

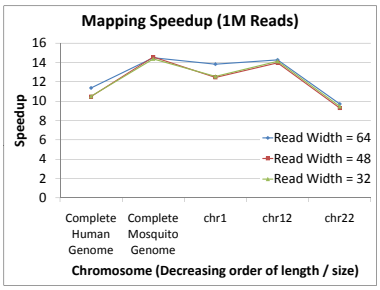
As future work, we would like to develop detailed performance models for the GPUs and evaluate the actual performance of GPU-RMAP against the model. We also hope to implement other sequence search or sequence alignment algorithms that use hashing as a core computational component (e.g., BLAST) on the GPU and evaluate them against the techniques used in GPU-RMAP.

Acknowledgments

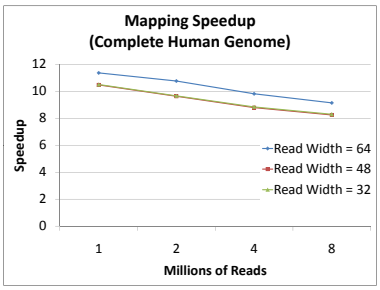
We would like to thank Heshan Lin for the inception of this project, the immense technical support in the project's initial design stages, and the detailed feedback on this manuscript. This work is supported in part by NSF grants IIP-0804155 and IIS-0710945 and an NVIDIA Professor Partnership Award.

References

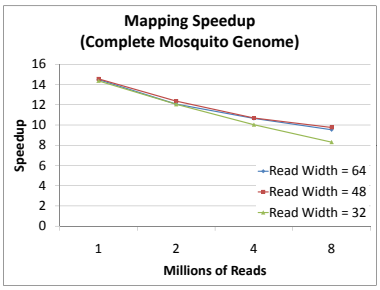
[1] D. Bozdag, C. C. Barbacioru, and U. V. Catalyurek, "Parallel Short Sequence Mapping for High Throughput Genome Sequencing," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–10.



(a) Effect of Chromosome Size on Speedup



(b) Effect of number of reads on Speedup (Human Genome)



(c) Effect of number of reads on Speedup (Mosquito Genome)

Figure 6. GPU-RMAP: Summary of Results.

[2] A. Smith, Z. Xuan, and M. Zhang, "Using Quality Scores and Longer Reads Improves Accuracy of Solexa Read Mapping," *BMC Bioinformatics*, vol. 9, no. 1, p. 128, 2008. [Online]. Available: http://www.biomedcentral.com/1471-2105/9/128

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic Local Alignment Search Tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403–410, October 1990. [Online]. Available: http://dx.doi.org/10.1006/jmbi.1990.9999

[4] B. Ma, J. Tromp, and M. Li, "PatternHunter: Faster and more Sensitive Homology Search," 2002. [Online]. Available: citeseer.ist.psu.edu/ma02patternhunter.html

[5] J. W. Kent, "BLAT–the BLAST-like Alignment Tool," *Genome Res*, vol. 12, no. 4, pp. 656–664, April 2002.

[6] H. Li and R. Durbin, "Maq: Mapping and Assembly with Qualities." [Online]. Available: http://maq.sourceforge.net/

[7] S. M. Rumble, P. Lacroute, A. V. Dalca, M. Fiume, A. Sidow, and M. Brudno, "SHRiMP: Accurate Mapping of Short Color-space Reads," *PLoS Comput Biol*, vol. 5, no. 5, pp. e1 000 386+, May 2009. [Online]. Available: http://dx.doi.org/10.1371/journal.pcbi.1000386

[8] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney, "High-Throughput Sequence Alignment Using Graphics Processing Units," *BMC Bioinformatics*, vol. 8, no. 1, p. 474, 2007. [Online]. Available: http://www.biomedcentral.com/1471-2105/8/474

[9] M. C. Schatz, "CloudBurst: Highly Sensitive Read Mapping with MapReduce," *Bioinformatics*, vol. 25, no. 11, pp. 1363–1369, June 2009. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btp236

[10] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, "Accelerating Molecular Modeling Applications with Graphics Processors," *Journal of Computational Chemistry*, vol. 28, pp. 2618–2640, 2007.

[11] L. Nyland, M. Harris, and J. Prins, "Fast N-Body Simulation with CUDA," *GPU Gems*, vol. 3, pp. 677–695, 2007.

[12] S. Xiao, A. M. Aji, and W. Feng, "On the Robust Mapping of Dynamic Programming nto a Graphics Processing Unit," in *15th International Conference on Parallel aond Distributed Systems (ICPADS)*, Shenzhen, China, December 2009.

[13] NVIDIA, "NVIDIA CUDA Programming Guide-2.3.1," 2009, http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf.

[14] C. Trapnell and M. C. Schatz, "Optimizing Data Intensive GPGPU Computations for DNA Sequence Alignment," *Parallel Computing*, vol. 35, no. 8-9, pp. 429 – 440, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/B6V12-4WK485P-1/2/da460c952ad271ff7aba6d40ad6734b9

[15] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[16] A. D. Smith, W.-Y. Chung, E. Hodges, J. Kendall, G. Hannon, J. Hicks, Z. Xuan, and M. Q. Zhang, "Updates to the RMAP Short-Read Mapping Software," *Bioinformatics*, vol. 25, no. 21, pp. 2841–2842, November 2009. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btp533

[17] L. J. Gosink, K. Wu, E. W. Bethel, J. D. Owens, and K. I. Joy, "Bin-Hash Indexing: A Parallel Method For Fast Query Processing," Laurence Berkeley National Laboratories, Tech. Rep. LBNL-729E, 2008.

[18] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Real-Time Parallel Hashing on the GPU," *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2009)*, vol. 28, no. 5, Dec. 2009. [Online]. Available: http://idav.ucdavis.edu/~dfalcant/research/hashing.php

[19] Henry S. Warren, *Hacker's Delight*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.