GPU-Accelerated BWA-MEM Genomic Mapping Algorithm Using Adaptive Load Balancing

Ernst Joachim Houtgast[1(✉)], Vlad-Mihai Sima[1], Koen Bertels[2], and Zaid Al-Ars[2]

[1] Bluebee, Delft, The Netherlands
{ernst.houtgast,vlad.sima}@bluebee.com
[2] Faculty of EEMCS, Delft University of Technology, Delft, The Netherlands
{k.l.m.bertels,z.al-ars}@tudelft.nl

Abstract. Genomic sequencing is rapidly becoming a premier generator of Big Data, posing great computational challenges. Hence, acceleration of the algorithms used is of utmost importance. This paper presents a GPU-accelerated implementation of BWA-MEM, a widely used algorithm to map genomic sequences onto a reference genome. BWA-MEM contains three main computational functions: Seed Generation, Seed Extension and Output Generation. This paper discusses acceleration of the Seed Extension function on a GPU accelerator.

The GPU-based Extend kernel achieves three times higher performance and, by offloading the kernel onto an accelerator and overlapping its execution with the other functions, this results in an overall improvement to application-level execution time of up to 1.6x.

To ensure that using an accelerator always results in an overall performance improvement, especially when considering slower GPUs, an adaptive load balancing solution is introduced, which intelligently distributes work between host and GPU. This provides, compared to not using load balancing, up to +46 % more performance.

Keywords: Acceleration · BWA-MEM · GPU · High performance genomics

1 Introduction

Genomics information proves to be a valuable source of information to clinicians and researchers alike. The amount of data generated by Next Generation Sequencing (NGS) techniques is increasing at an explosive rate and will soon rival, if not overtake, other Big Data fields such as astronomy [14]. The raw sequenced data is processed by a complex pipeline of algorithms, a so-called genomics analysis pipeline. This data processing can require many days, even on a large cluster, and is becoming a bottleneck for applications dependent on

F. Hannig et al. (Eds.): ARCS 2016, LNCS 9637, pp. 130–142, 2016.
DOI: 10.1007/978-3-319-30695-7_10

genetic information. Hence, the challenges in genomics are shifting from sequencing towards data processing. Therefore, acceleration of bioinformatics algorithms is vital to relieve these bottlenecks.

One step in genomics analysis pipelines is to reconstruct the original genome from the millions of short reads produced using NGS. The purpose of subsequent steps in the pipeline is to find differences in the sequenced genetic material as compared to annotated reference material. The reconstruction step of a typical pipeline is represented by the mapping of the short reads onto a reference genome. BWA-MEM [9] is widely used in practice to this end.

This paper presents the following contributions:

- The first GPU-based implementation of the BWA-MEM algorithm.
- A load balancing algorithm to distribute reads between host and accelerator.
- A comparison of kernel and system-level results to an FPGA implementation.

The rest of this paper is organized as follows: Sect. 2 places this work into its context within related work. Section 3 discusses the BWA-MEM program operation and its main functions. Section 4 explains the modification of program scheduling to improve the acceleration potential. Section 5 describes the load balancing system. Section 6 discusses the GPU implementation. Section 7 presents the methods and results. The paper is concluded by Sect. 8.

2 Related Work

Although BWA-MEM [9] is one of the most popular mapping tools, there are numerous other mapping tools available. Most state-of-the-art mapping tools, such as [7], follow the Seed-and-Extend paradigm, explained below. Mapping tools generally differ in their mapping quality and speed. BWA-MEM offers a good compromise between mapping speed and quality. Many accelerated Seed-and-Extend-based mapping tools exist. However, in the field of bioinformatics, exactness of results is critical. To the authors' knowledge, the only accelerated versions of BWA-MEM are [1,5]. In [5], one of the BWA-MEM kernels is mapped onto a FPGA-based systolic array. This is further improved upon in [1], in which multiple BWA-MEM kernels are accelerated. The work here is similar to [5], but implements the systolic array on a GPU-based platform instead.

3 BWA-MEM Algorithm

BWA-MEM [9] is used to map sequenced reads onto a reference genome, such as the human genome. To illustrate the data sizes involved, a single run on a currently state-of-the-art sequencing platform, the Illumina HiSeq X, generates up to six billion pair-ended reads of 150 base pairs (or bp) in less than three days [6]. Even on a cluster, processing this data can take multiple days.

BWA-MEM is based on the Seed-and-Extend paradigm (refer to Fig. 1). For each read, *seed* locations on the genome are determined, exactly matching subsequences of the read and the reference. Then, Seed Extension is performed: an

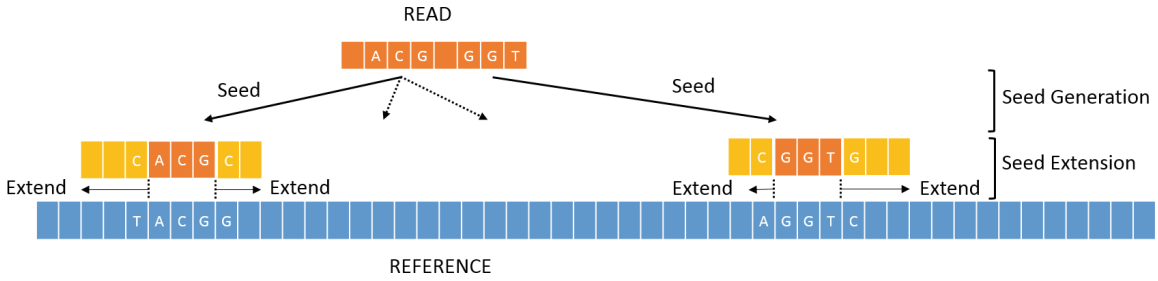

Fig. 1. BWA-MEM processes reads using the Seed-and-Extend paradigm: for each read, likely mapping locations on the reference are found by searching for exactly matching subsequences between the read and the reference, so called *seeds*. Then, these seeds are extended in both directions using a Smith-Waterman-like dynamic programming approach that allows for inexact matches. From all of these extended seeds, the best scoring alignment is selected.

attempt to extend these seeds in both directions using an alignment algorithm that allows for inexact matches. The best scoring alignment is chosen from all the resulting alignments. The final score is obtained by performing global alignment over the entire read against the chosen reference region.

3.1 BWA-MEM Profiling Results

The BWA-MEM algorithm main functions are: Seed Generation, Seed Extension, and Output Generation. To investigate the acceleration potential of BWA-MEM, the application has been profiled using the GCAT data set. The results are shown in Table 1, which reveals that acceleration of BWA-MEM is not trivial: processing is divided over multiple functions. As per Amdahl's law, speedup resulting from acceleration of any single function is limited. Greater speedup can only be achieved when accelerating multiple functions, such as in [1]. The table also shows that Seed Extension is the function limited by a computational bottleneck. For this reason, the Seed Extension function was chosen as initial optimization target. The other functions are not further analyzed in this paper.

Table 1. Results of BWA-MEM algorithm profiling (tests performed on Intel Core i7-4790 @ 4 GHz with the GCAT 150bp-se-small-indel data set)

Program kernel	Time	Bottleneck	Processing	Max speedup
Seed generation	46 %	Memory	Parallel	1.85x
Seed extension	43 %	Computation	Parallel	1.75x
Output generation	4 %	Memory	Parallel	1.04x
Other	7 %	I/O	Sequential	1.08x

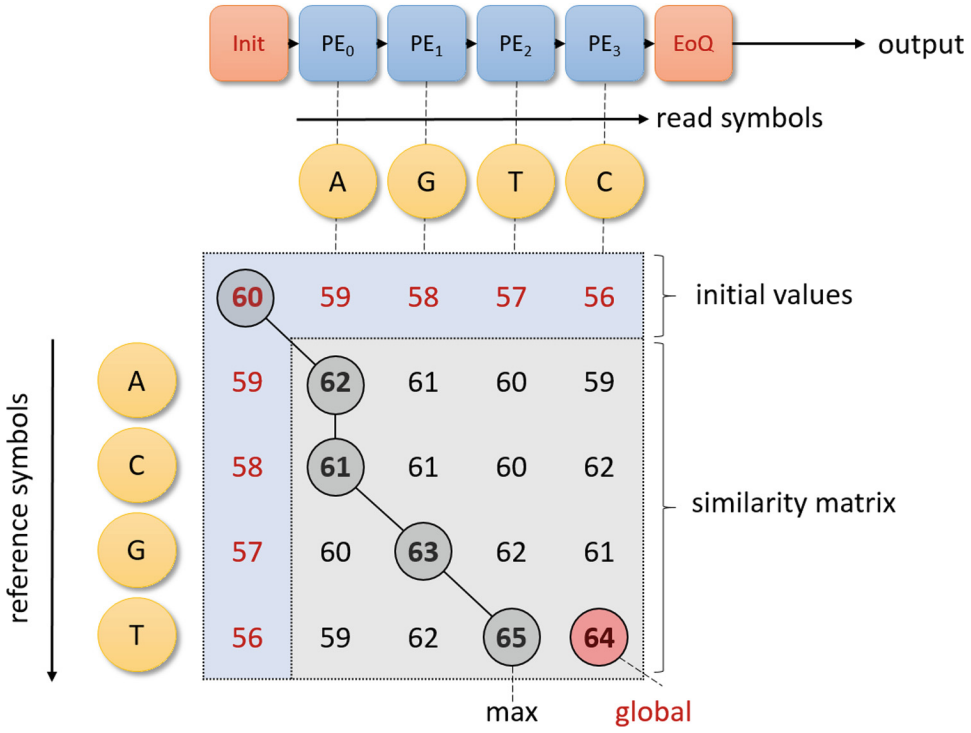

Fig. 2. Extend algorithm similarity matrix with initial score 60 showing local alignment with maximum score and global maximum score. Read symbols map one-to-one onto systolic array Processing Elements. Matrix entries only depend on top, top-left, and left neighbor. Thus, anti-diagonals can be processed in parallel. Differences compared to regular Smith-Waterman are: additional Initialization and End-of-Query blocks, non-zero initial values, and additional outputs, such as the global maximum (from [5]).

3.2 Seed Extension Functional Details

After Seed Generation, Seed Extension is invoked, which consists of two separate components: an outer function that loops over all seeds and determines whether it should be extended or not; and the actual Extend kernel. The number of times the Extend kernel is performed depends on the number of seeds found, which can range from none to thousands of seeds per short read. Since seeds generally only encompass a subsequence of the read, they may be extended in either direction, unless the seed includes the first and/or last symbol of the read.

The outer function keeps track of all earlier found extensions belonging to one read. If a later seed overlaps previous extensions by a certain amount, the seed is ignored. Seeds that are located close together on the reference are grouped into *chains*. Profiling shows that, in general, only one seed per chain is extended. Hence, a dependency exists between the extension of seeds. This dependency makes Extend less suitable for parallel execution: speculatively performing all extensions in parallel would cause significant work that would outweigh any benefit of parallelization. Therefore, Extend kernel executions for a read are performed serially. Instead, parallelism is extracted on two other levels: on the read-level by processing multiple reads at the same time, and by utilizing the parallelism inherent in the extension algorithm itself.

The extension algorithm is similar to the well-known Smith-Waterman dynamic programming algorithm [13], used to align two sequences to each other. Its basic operation is illustrated in Fig. 2. To compute the optimal alignment, a similarity matrix is filled, thus computing all possible alignments. The value of one cell in this matrix is only dependent on its top, top-left, and left- neighbor. Hence, anti-diagonals can be computed in parallel. A systolic array implementation is a natural way to map the problem onto Processing Elements when using an acceleration platform [10,15].

Most GPU-based Smith-Waterman acceleration efforts operate by mapping the complete processing of one alignment to a single core, in effect doing hundreds of sequence alignments in parallel [3,11]. As on a GPU the cores operate in lock-step, optimal performance is contingent on balancing the workload per core. Hence, alignments are sorted beforehand. Unfortunately, for BWA-MEM this method is unsuitable as Extend kernel invocations are generated dynamically and can have very different lengths. Therefore, to extract parallelism, the implementation described here operates in a systolic array-like manner.

As the Extend kernel is used to extend an earlier found match, in contrast to simply aligning sequences in complete isolation, it differs from regular Smith-Waterman in three ways, explained below. These differences are also illustrated in Fig. 2. The result is that the Extend kernel implementation is more complex than a normal Smith-Waterman implementation.

Non-zero Initial values: The initial values of the similarity matrix are not zero, but depend on the score of the seed that is being extended. Therefore, an Initial Value block is added in front of the systolic array.

Additional Outputs: The Extend kernel produces more outputs than the normal Smith-Waterman algorithm. Therefore, to obtain these, the output is post-processed by an additional End-of-Query block.

Partial Similarity Matrix Calculation: The algorithm uses a heuristic to restrict the similarity matrix calculations to only those cells that are likely to influence the end result.

4 Accelerated Program Architecture

In this section, changes made to enable an accelerated implementation of the BWA-MEM algorithm are described. The main goal was to drastically reduce the number of Seed Extension invocations. The original BWA-MEM algorithm works as shown in Algorithm 1. The input data is processed in batches. For each read in a batch, Seed Generation is performed first; then, Seed Extension; and finally, Output Generation. Note that for each read, Seed Generation and Seed Extension are performed directly after one another.

Applying heterogeneous acceleration of the Seed Extension function call directly to this structure would imply accelerator invocation for every individual read, along with the accompanying data transfers from and to the device's memory. As typically many millions of reads are processed, the resulting overhead

Algorithm 1. Original Program Structure

Input: a batch of n reads
Output: n aligned reads
1: **for** $i = 1$ to n **do**
2: Seed Generation(read i)
3: Seed Extension(read i)
4: **end for**
5: **for** $i = 1$ to n **do**
6: Output Generation(read i)
7: **end for**

would be likely to nullify any gains resulting from more efficient execution. Moreover, acceleration of a single alignment may not even be faster than processing it on the host. Often, speedup is obtained by leveraging the massive parallelism inherent in the data set to be processed, which accelerators are able to exploit.

Therefore, the BWA-MEM program structure has been refactored in order to be more receptive to heterogeneous execution. The refactored structure is given in Algorithm 2. Note that the workload has been subdivided into chunks of reads. For each chunk, first, Seed Generation is performed for all reads in the chunk. Then, the Seed Extension function is executed for all the reads in the chunk. Then, the algorithm proceeds to the next chunk. After all chunks are finished, Output Generation is performed. This setup requires temporary data storage, which is in the order of tens of megabytes. However, this approach is far more suitable to acceleration, as in this situation a single accelerator invocation suffices to perform Seed Extension for the entire chunk, as opposed to one invocation per read. The reduction in number of invocations is on the same order of magnitude as the chunk size, which is typically in the order of tens of thousands.

Algorithm 2. Refactored Program Structure

Input: a batch of n reads
Output: n aligned reads
1: **for** $i = 1$ to $n/chunksize$ **do**
2: **for** $j = 1$ to $chunksize$ **do**
3: Seed Generation(read $j + (i - 1) \times chunksize$)
4: **end for**
5: **for** $j = 1$ to $chunksize$ **do**
6: Seed Extension(read $j + (i - 1) \times chunksize$)
7: **end for**
8: **end for**
9: **for** $i = 1$ to n **do**
10: Output Generation(read i)
11: **end for**

Note that Algorithm 2 has been implemented in such a way that Seed Generation and Seed Extension are overlapped in a pipelined fashion. Hence, ideally,

the execution of Seed Extension is almost completely hidden, resulting in a maximum theoretical speedup of 1.75x, as predicted by Amdahl's law.

5 Adaptive Load Balancing Strategy

To accelerate the Seed Extension function (lines 5–7 of Algorithm 2), a GPU is used to assist the host in processing the Seed Extension work. To ensure optimal speedup, even for slower GPUs, an adaptive load balancing strategy is used to determine the optimal division of work between host and GPU, controlled by a Load Balancing Factor parameter (LBF). The goal of this algorithm is to minimize the idle time on both host and GPU. Otherwise, simply offloading all the work onto a slower GPU might result in an application slowdown, instead of in an application speedup. The LBF is recalculated after each batch of reads as shown in Algorithm 3. As the amount of work per batch seems mostly stable, idle time is minimized by measuring the host and the GPU processing times to determine their respective busy percentage during the previous batch and modifying the LBF accordingly (similar to [2]). Given a sufficiently fast GPU, all the work can be offloaded from the host. However, for a slower GPU, only part of the work may be performed on the GPU, hence LBF will be less than 1. The load balancing should result in a speedup in all cases though. The algorithm uses smoothing in order to prevent oscillations of the LBF.

Algorithm 3. Adaptive Load Balancing Strategy

Input: HostBusyPct, GPUBusyPct, LBF_{old}
Output: LBF_{new}
1: **for** each batch of reads **do**
2: $\quad \text{LBF}_{\text{old}} = \text{LBF}_{\text{new}}$
3: $\quad \text{LBF}_{\text{new}} = (\text{HostBusyPct} \,/\, \text{GPUBusyPct}) \times \text{LBF}_{\text{old}}$
4: $\quad \text{LBF}_{\text{new}} = \min(1, (\text{LBF}_{\text{new}} + \text{LBF}_{\text{old}}) \,/\, 2)$
5: **end for**

6 Implementation Details

The GPU implementation of Seed Extension consists of an outer loop and the actual Extend kernel. These have been implemented as separate kernels using the NVIDIA CUDA Runtime API. In this section, the GPU kernels and the optimizations that were applied are described in more detail.

6.1 Seed Extension Function Kernels

As discussed before (see Algorithm 2), reads are sent in large batches to the GPU. Each read is processed independently by the outer loop kernel, a control function that loops over the seeds and, using CUDA Dynamic Parallelism (available from

Table 2. Summary of NVIDIA CUDA compiler & profiling information

CUDA kernel	# Calls	Time	Registers	Shared memory	Threads
Outer loop	1	66 %	78	0 kB	1
Extend multipass long	24657	17 %	34	2.9 kB	32
Extend wide	17912	11 %	54	3.3 kB	1–131
Extend multipass short	9695	3 %	34	1.7 kB	32
Extend single pass	17640	3 %	30	0.5 kB	32

CUDA Compute Capability 3.5 onward), instantiates Extend kernels as needed. This function only runs as a single thread. For the Extend kernel itself, four versions of the kernel have been implemented to optimize register and shared memory usage to improve occupancy. These are described in the next section. Table 2 provides some information on the CUDA kernels in use.[1] From the table, it is clear that most time is spent in the outer loop, which is characterized by random memory accesses and branching operations.

6.2 Extend Systolic Array Kernels

The basic idea of all the Extend kernels is their implementation as a systolic array, similar to [5]. The largest advantage of using a systolic array is the possibility to extract the available parallelism on anti-diagonals while calculating the similarity matrix. Using a systolic array, calculation of the entire array takes $O(|\text{Reference}| + |\text{Extension}|)$ execution time, instead of $O(|\text{Reference}| \times |\text{Extension}|)$. For larger problem sizes, this can result in a large speedup compared to a serialized implementation. The drawback of a systolic implementation is the often low overall efficiency: in general, not all the Processing Elements (or PEs) of the array can be kept busy. Full utilization is only attained during calculation of the "widest" diagonals of the matrix. For the other diagonals, PEs at the start and/or at the end of the array will be idle, lowering overall efficiency. Moreover, for physically implemented systolic arrays, unnecessary latency is incurred when processing reads shorter than the array itself. Also, the number of PEs determines the maximum length of the extension that can be processed, as one PE is required for each read symbol that is to be extended. Longer reads can be processed by making multiple passes over the matrix, with temporary data stored between passes, as in [12]. The GPU implementation does not suffer from these issues as the systolic array length is dynamically instantiated.

The GPU implementation maps read symbols onto the systolic array PEs, similar to Fig. 2. The PEs are implemented as CUDA cores, where a CUDA thread performs the calculations of that PE. CUDA threads are grouped into blocks of 32 threads, a *warp*, which all perform exactly the same instruction.

[1] These numbers are obtained while executing the first 50,000 reads of the GCAT 150bp-se-small-indel data set using the *nvprof* and *nvcc* tools.

A warp is the basic unit of action in an NVIDIA GPU. The *Ext. wide* kernel is the most straightforward systolic array implementation. On the left of Fig. 3, is shown how the similarity matrix is processed over time. As many warps as necessary are allocated to process the matrix. After each cycle, PEs exchange data through the on-chip Shared Memory cache. For larger extension lengths, this can require a large amount of shared memory. Moreover, from Fig. 3 it is clear that many PEs will be idle for much of the time.

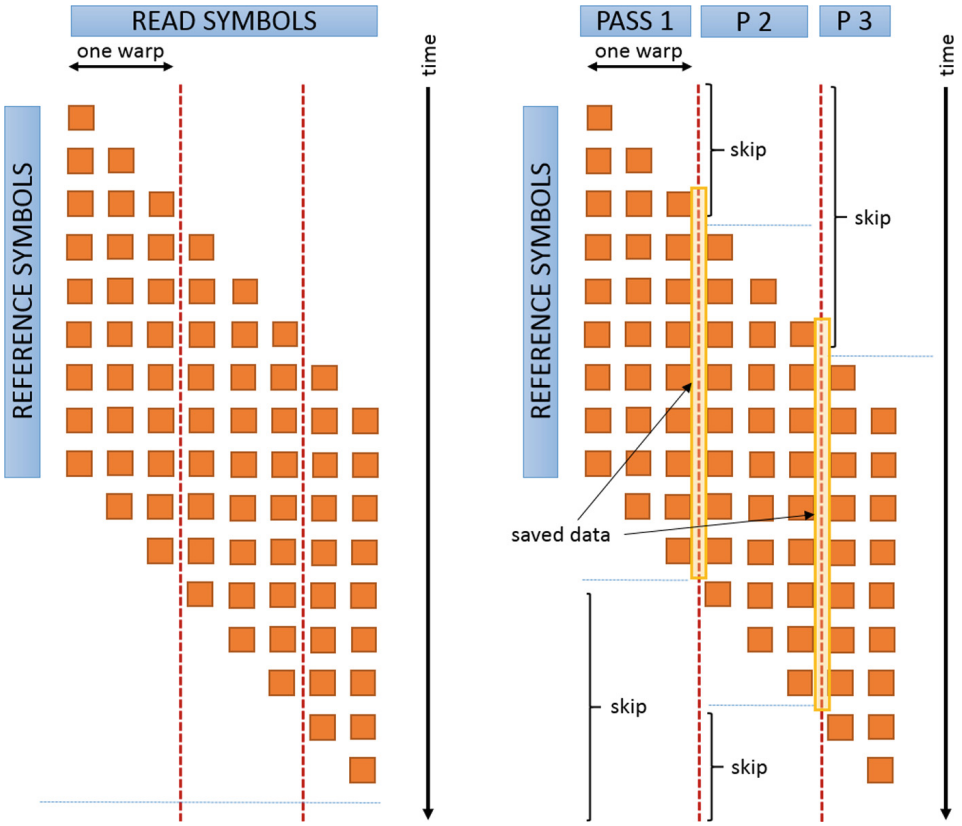

Fig. 3. Systolic array-based GPU Extend kernel implementation. Extension symbols are mapped one-to-one on CUDA cores, reference symbols are fed each iteration of the loop. After each iteration, data is exchanged through shared memory (left). The single warp-based implementation makes multiple passes over the array (right). Unnecessary iterations are skipped over and per-pass temporary data is saved in shared memory.

Therefore, a number of kernels have been implemented designed to process the matrix on a single warp, which corresponds to 32-symbol wide columns. This is shown on the right of Fig. 3. Multiple passes are made over the matrix, with intermediate data between passes saved into shared memory. Data exchange between cores is implemented using shuffle instructions, avoiding the use of shared memory. Unnecessary iterations per pass are skipped, drastically reducing idle time. For example, extending a size 150 reference against a size 100 extension would, in the simple implementation, result in on average 60 cores out of 100 being busy; however, for the warp-based implementation, 27 out of 32 are

busy. Efficiency is 40 % higher. The *Ext. multipass long* and *Ext. multipass short* kernels differ in the available amount of statically allocated storage space. The *Ext. single pass* kernel is used when the entire matrix fits within a single warp (i.e., 32 read symbols or smaller), and hence only one pass is needed. In this case, no intermediate data from the matrix needs to be saved in shared memory. The use of the different kernels provides a 20 % improvement to performance.

6.3 GPU-Specific Optimizations

Apart from the multiple Extend kernel implementations, the following optimizations were applied and are worth mentioning:

Coalesced Memory Access: Memory accesses are coalesced as much as possible. In contrast to a normal systolic array, reference symbols are loaded in one large coalesced access. Read symbols are obtained similarly.

Shuffle Instructions: Shuffle instructions are used to remove the need to use shared memory for data exchange between PEs. This is only possible within a warp, hence the need for a multiple pass implementation.

Dynamic Parallelism: To reduce register pressure, the outer controlling function uses only a single thread, subsequently invoking Extend kernels with as many threads as needed using CUDA Dynamic Parallelism.

7 Results

Profiling and performance tests were performed on a machine with an Intel Core i7-4790 (four cores, Hyper-Threading enabled) running at 4.0 GHz, with 32 GB of DDR3 memory. The system contains two NVIDIA GeForce GTX TITAN X cards, with 3,072 CUDA cores each, running at up to 1,076 MHz, and offering Compute Capability 5.0. The GPU implementation requires at least Compute Capability 3.5 in order to be able to use dynamic parallelism. NVIDIA CUDA Runtime API version 7.5 was used.

The 150bp-se-small-indel data set from the Genome Comparison & Analytic Testing (GCAT) framework [4] was used to map about eight million 150 base pair reads onto the UCSC HG19 reference human genome. The GCAT online sequence alignment quality comparison service was used to verify that results of the GPU-accelerated version are similar to those obtained with the original BWA-MEM algorithm. BWA-MEM version 0.7.7 was used [8].

7.1 Performance Results and Scaling

Table 3 shows the Extend kernel execution time and overall application performance for single and dual GPU execution using eight CPU threads. The results of the FPGA implementation from [5] are also given. As the platforms are non-identical (they use 2x Intel Xeon E5-2643 at 3.3 GHz), relative Extend

Table 3. Execution time and speedup on the GCAT alignment quality benchmark

		Extend kernel		Overall program		
Platform	Test	Time	Speedup	Time	Speedup	Throughput
GPU-Accelerated	CPU only	218 s	-	510 s	-	2.4 Mbp/s
	CPU+Single GPU	118 s	1.9x	468 s	1.09x	2.6 Mbp/s
	CPU+Dual GPU	73 s	3.0x	422 s	1.21x	2.9 Mbp/s
FPGA-Accelerated	CPU only	167 s	-	530 s	-	2.3 Mbp/s
	CPU+FPGA	62 s	2.7x	365 s	1.45x	3.3 Mbp/s

kernel times differ, mostly due to the different CPUs. Results are normalized to throughput in base pairs per second, to facilitate comparison of numbers.

The Extend dynamic programming kernel is three times faster compared to CPU-only execution. Even though execution of this kernel is overlapped with the functions executed on the host CPU, the results show that, in contrast to the FPGA implementation, the GPU-accelerated version is unable to hide the entire Seed Extension function time, due to the large overhead of the outer function. Performance results for varying CPU thread counts are given in Fig. 4. The dual GPU setup is able to achieve a speedup of 1.6x for up to two threads, or 1.5x for four threads. The maximum speedup of 1.75x is not achieved, due to batching overhead and since GPU on-chip memory limitations allow only 99.5 % of Seed Extensions to be processed on the GPU. The remaining reads, with thousands of seeds, are processed on the host and still require about 4 % of overall host execution time, reducing the maximum achievable speedup accordingly.

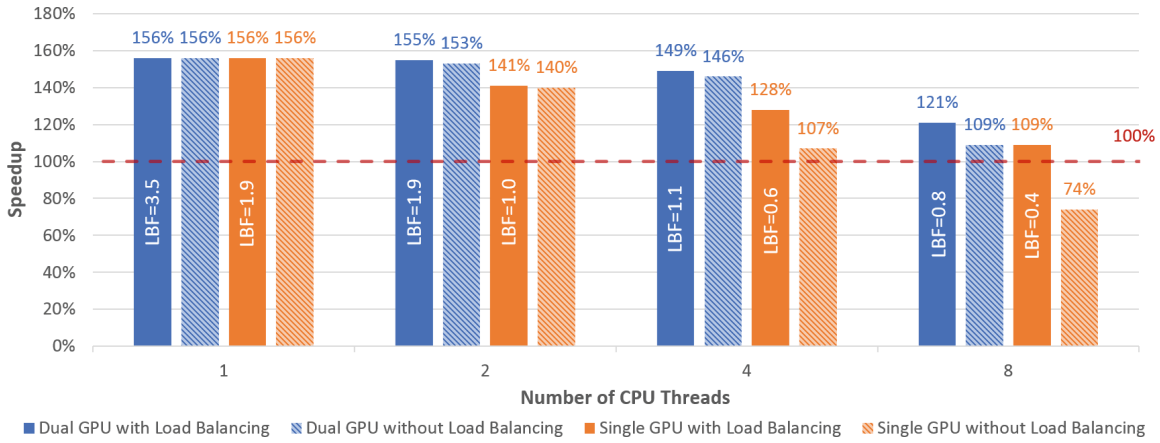

Fig. 4. Overall application speedup for varying number of CPU threads and single and dual GPUs. Results shown with load balancing enabled and disabled. The adaptive load balancing ensures efficient host and accelerator usage and provides an overall application speedup even for GPU-constrained scenarios, which might otherwise result in an overall application slowdown.

7.2 Load Balancing Results

An adaptive load balancing algorithm was implemented to ensure optimal benefit from the use of acceleration. Figure 4 shows that the load balancing is effective: for increasing number of CPU threads, the load balanced single GPU scenario provides similar or better performance as compared to non-load balanced execution, improving performance by up to 46 %. Note that execution using eight threads results in a slowdown on the non-load balanced situation, due to a mismatch in host and accelerator performance. For a dual GPU setup, load balancing still provides a benefit, but only when using eight threads. The unbounded LBF value is also given. This shows that the dual GPU setup is able to perform up to 90 % more work than a single GPU setup.

8 Conclusion

This paper describes a GPU-accelerated version of the BWA-MEM genomic mapping algorithm. It was possible to hide the execution time of the Seed Extension function, one of the three main computational functions, by overlapping its execution with the other program functions for up to four CPU threads. Speedup of up to three times is achieved for the Extend kernel, which translates in an overall improvement to BWA-MEM execution time of up to 1.6x. This can save days of processing time on real-world data sets.

A generally applicable adaptive load balancing strategy was implemented to ensure an efficient division of work between the host and the GPU, improving performance and ensuring application speedup even for mismatched host and accelerator performance. The load balancing algorithm provides an improvement to performance of up to 46 %, compared to non-load balanced execution.

Although the work here focuses on BWA-MEM, a widely used genomic mapping tool, the approach is valid for many similar Seed-and-Extend-based bioinformatics algorithms. Future work will focus on the reorganization of the outer Seed Extension function to make it better suitable towards parallel execution, and will also focus on porting other parts of BWA-MEM onto the GPU.

Acknowledgments. The authors would like to thank the people at the Neuroscience Department of the Erasmus Medical Center for kindly granting access to their computing facilities for performance tests.

References

1. Ahmed, N., Sima, V.M., Houtgast, E., Bertels, K., Al-Ars, Z.: Heterogeneous hardware/software acceleration of the BWA-MEM DNA alignment algorithm. In: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2015, pp. 240–246. IEEE Press, Piscataway, NJ, USA (2015). http://dl.acm.org/citation.cfm?id=2840819.2840854

2. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency Comput. Pract. Experience **23**(2), 187–198 (2011)
3. Hasan, L., Kentie, M., Al-Ars, Z.: DOPA: GPU-based protein alignment using database and memory access optimizations. BMC Res. Notes **4**(1), 261 (2011)
4. Highnam, G., Wang, J.J., Kusler, D., Zook, J., Vijayan, V., Leibovich, N., Mittelman, D.: An analytical framework for optimizing variant discovery from personal genomes. Nature Comm. **6** (2015)
5. Houtgast, E., Sima, V., Bertels, K., Al-Ars, Z.: An FPGA-based systolic array to accelerate the BWA-MEM genomic mapping algorithm. In: International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (2015)
6. Illumina: HiSeq X Specification Sheet. http://www.illumina.com/content/dam/illumina-marketing/documents/products/datasheets/datasheet-hiseq-x-ten.pdf. Accessed 15 July 2015
7. Langmead, B., Salzberg, S.L.: Fast gapped-read alignment with Bowtie 2. Nat. Methods **9**(4), 357–359 (2012)
8. Li, H.: Burrows-Wheeler Aligner. http://bio-bwa.sourceforge.net/. Accessed 04 November 2014
9. Li, H.: Aligning Sequence Reads, Clone Sequences and Assembly Contigs with BWA-MEM. arXiv preprint arxiv:1303.3997 (2013)
10. Liu, W., Schmidt, B., Voss, G., Schroder, A., Muller-Wittig, W.: Bio-sequence database scanning on a GPU. In: 20th International Parallel and Distributed Processing Symposium, 2006, IPDPS 2006, p. 8. IEEE (2006)
11. Liu, Y., Wirawan, A., Schmidt, B.: CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. BMC Bioinformatics **14**(1), 117 (2013)
12. Oliver, T., Schmidt, B., Maskell, D.: Hyper customized processors for bio-sequence database scanning on FPGAs. In: Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays, pp. 229–237. ACM (2005)
13. Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences. J. Mol. Biol. **147**(1), 195–197 (1981)
14. Stephens, Z., Lee, S., Faghri, F., Campbell, R., Zhai, C., Efron, M., et al.: Big data: astronomical or genomical? PLoS Biol. **13**(7), e1002195 (2015)
15. Yu, C.W., Kwong, K., Lee, K.H., Leong, P.H.W.: A Smith-Waterman systolic cell. In: Lysaght, P., Rosenstiel, W. (eds.) New Algorithms, Architectures and Applications for Reconfigurable Computing, pp. 291–300. Springer, Heidelberg (2005)