Hardware acceleration of BWA-MEM genomic short read mapping for longer read lengths

Houtgast, Ernst Joachim; Sima, Vlad-Mihai; Bertels, Koen; Al-Ars, Zaid

DOI
10.1016/j.compbiolchem.2018.03.024

Publication date
2018

Document Version
Accepted author manuscript

Published in
Computational Biology and Chemistry

Hardware Acceleration of BWA-MEM Genomic Short Read Mapping for Longer Read Lengths

Ernst Joachim Houtgast[a,b,*], Vlad-Mihai Sima[b], Koen Bertels[a], Zaid Al-Ars[a]

[a] *Computer Engineering Lab, TU Delft, Mekelweg 4, 2628 CD Delft, The Netherlands*
[b] *Bluebee, Laan van Zuid Hoorn 57, 2289 DC Rijswijk, The Netherlands*

Abstract

We present our work on hardware accelerated genomics pipelines, using either FPGAs or GPUs to accelerate execution of BWA-MEM, a widely-used algorithm for genomic short read mapping. The mapping stage can take up to 40% of overall processing time for genomics pipelines. Our implementation offloads the Seed Extension function, one of the main BWA-MEM computational functions, onto an accelerator.

Typical sequencer output are reads with a length of 150 base pairs. However, read length is expected to increase in the near future. Here, we investigate the influence of read length on BWA-MEM performance using data sets with read length up to 400 base pairs, and introduce methods to ameliorate the impact of longer read length. For the industry-standard 150 base pair read length, our implementation achieves an up to two-fold increase in overall application-level performance for systems with at most twenty-two logical CPU cores. Longer read length requires commensurately bigger data structures, which directly impacts accelerator efficiency. The two-fold performance increase is sustained for read length of at most 250 base pairs.

To improve performance, we perform a classification of the inefficiency of the underlying systolic array architecture. By eliminating idle regions as much as possible, efficiency is improved by up to +95%. Moreover, adaptive load balancing intelligently distributes work between host and accelerator to ensure use of an accelerator always results in performance improvement, which in GPU-constrained scenarios provides up to +45% more performance.

Keywords: Acceleration, BWA-MEM, FPGA, GPU, Short Read Mapping, Systolic Array.

1. Introduction

Next Generation Sequencing (NGS) has profoundly changed the field of genomics. As the cost of sequencing continues to drop and, in turn, its use is becoming pervasive, the bottleneck is starting to shift from the actual sequencing itself, towards the IT domain. It is projected that NGS will rival, if not overtake, other big data fields such as astronomy and streaming video services within ten years, both in terms of data storage as well as data processing [1]. Hence, acceleration of the algorithms used for genomics data processing is vital to keep up with the projected growth in demand for these services.

A key characteristic of current NGS sequencers is that they cannot read complete chromosomes, or even significantly long stretches of DNA. Instead, only small fragments of DNA called *short reads* are read, for example of 150 base pairs in length. However, the sequencer can produce many millions of such short reads in parallel. Therefore, reproducing the complete genome becomes a bit analogous to reassembling a book that has been torn into very small pieces. The process of reassembling is done through a process called a *genomics pipeline*. Such a pipeline typically starts with a mapping phases. Here, each short read fragment is compared to a reference genome to find the best matching location of where it would fit with the fewest number of differences. Then, after all reads are mapped, a sorting and deduplication phase follows, until, finally, the variant calling phase can be performed. This is the phase where difference between the sequenced genome and the reference genome are discovered. Such differences, or variants, are what the sequencing exercise is all about, because they can indicate phenotypical characteristics such as eye color, but also a propensity towards certain diseases, such as diabetes. As shown in Figure 1, the mapping phase takes a significant amount of time of the overall genomics pipeline execution time.

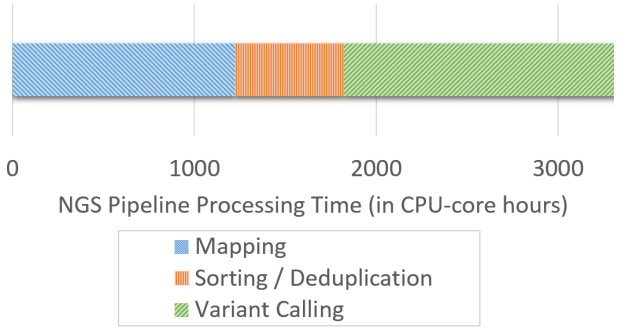

Figure 1: Breakdown of processing time per NGS pipeline stage for a typical 30x coverage cancer NGS DNA data set. The data set consists of three tumor samples and one normal tissue sample (time given in CPU-core hours).

*Corresponding author: ernst.houtgast@bluebee.com

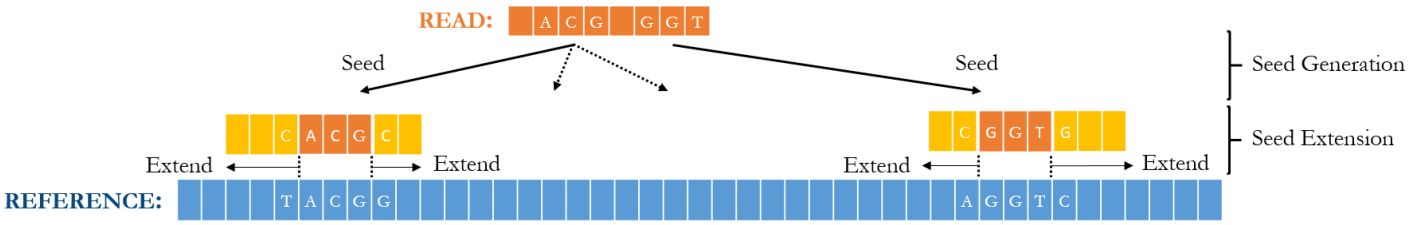

Figure 2: Most state-of-the-art mapping tools use a paradigm called Seed-and-Extend to map a short read fragment onto a reference genome: first, exactly matching subsequences between the short read and the reference genome are identified, using for example the BWT. These are called *seeds*. Then, these subsequences or seeds are further extended using an algorithm such as the Smith-Waterman algorithm that can tolerate mismatches between two sequences. Finally, out of the many seeds that may have been generated and extended, the highest scoring alignment is selected as final output.

Therefore, this paper investigates the acceleration of the mapping phase, in particular for longer read lengths.

A typical sequencing run on an Illumina HiSeq X [2], which is a state-of-the-art NGS sequencer, produces about 1.2 TB of data every two days. For cancer data processing pipelines, this requires multiple days of processing, even when utilizing high performance computing clusters. The extreme scale of data and processing requires enormous computing capabilities to make the analysis feasible within a realistic time frame. Heterogeneous computing holds great potential for large advantages in speed and efficiency, compared to pure software-only execution on general purpose processors.

Most current sequencers output reads with a length of 150 base pairs, examples include the Illumina MiniSeq, NextSeq, and HiSeq series [3]. However, support for longer read lengths is an important consideration as this is the direction that sequencing technology is moving towards. Therefore, in this article we investigate the effectiveness of hardware acceleration of BWA-MEM for a variety of read lengths. We present:

- A GPU-based BWA-MEM Seed Extension kernel that is able to map reads up to 1150 bp, resulting in an overall application-level speedup of up to 2x, which is at least about 25% faster than competing accelerated solutions;
- The effects of short read length on the overall application behavior and performance profile, and on the resulting effectiveness of acceleration;
- A classification of the inefficiencies that are inherent in systolic array designs, in particular for designs with many processing elements;
- Techniques to ameliorate the increased computational load for longer read lengths, through adaptive load balancing and optimizing the underlying systolic array architecture.

The remainder of this article is organized as follows. In Section 2, related work is discussed. Section 3 presents the BWA-MEM algorithm and its functions, in particular the Seed Extension kernel. Section 4 briefly mentions the modification made to the program architecture to improve acceleration potential and the load balancing system. Section 5 discusses the accelerated implementation and its limitations. In Section 6, methods and results are presented. Section 7 contains a discussion of the results. The article is concluded by Section 8.

2. Related Work

The mapping of sequences onto a reference genome is part of a field called sequence alignment. Sequence alignment can be broadly divided into two main categories: pairwise alignment, in which two sequences are to be matched to each other, and multiple sequence alignment, in which the best alignment between a group of sequences is to be found. Many such alignment tools exist, along with numerous accelerated implementations. In the current case, we are only interested in pairwise alignment, since we need to map a short read onto a reference genome. A large number of short read mapping tools exists. As sequence alignment is computationally expensive, the most popular ones all use a heuristic method called Seed-and-Extend. This is explained in Figure 2. BWA-MEM [4] is one of the most widely used tools for short read mapping, as it is able to combine speed with accuracy of finding results.

BWA-MEM differs from most other pairwise alignment tools, such as SOAPv3 [5] and CUSHAW [6], by virtue of the fact that its extend phase offers the most flexibility. For example, SOAPv3 does not allow gaps in the alignment, and CUSHAW only allows for a limited number of mismatches. By utilizing the Smith-Waterman algorithm, BWA-MEM is free from these limitations and is able to find the optimal result for the sections to be extended. This does come at a cost, since the Smith-Waterman algorithm is computationally expensive. Therefore, in our work we focus on accelerating this part of the algorithm. Many accelerated implementations of the Smith-Waterman algorithm exist, for example [7], [8], [9], and [10]. However, the integration of this algorithm into BWA-MEM is far from trivial, as most implementations operate by performing many Smith-Waterman invocations in parallel, which is something that cannot be used in the case of BWA-MEM as will become clear in Section 5.

This work builds upon our prior work on accelerating the BWA-MEM algorithm, which used FPGAs to accelerate the Seed Extension algorithm, both on the Convey supercomputing platform [11], as well as by using an AlphaData add-in board [12]. These implementations were able to achieve an up to two-fold speedup. We also ported our work onto the GPU [13, 14], resulting in a similar performance boost. This work is an extension of our earlier GPU work, which was limited to processing input data sets with short reads of 150 base pairs in length. Here, we focus on the effects of longer read lengths of up to 4'600 base pairs, requiring modified GPU code. We investigate

the bottlenecks and limitations of such greatly increased read lengths. Besides our accelerated implementations, we know of two other accelerated BWA-MEM implementations, both utilizing FPGAs: the work by Chang, which accelerates the Seed Generation phase and is able to achieve a 1.26x speedup [15], and the work by Chen, which accelerates the Seed Extension phase and is able to achieve a 1.5x speedup [16].

3. BWA-MEM Algorithm Details

BWA-MEM is a popular short read mapping tool [4], widely used in genomics pipelines to find for each short read in the input data set a suitable location on the reference genome. This is accomplished through a method called the Seed-and-Extend paradigm, explained in Figure 2. This is a two-step process with an Exact Matching phase and an Inexact Matching phase. For each read, first, exactly matching subsequences called *seeds* are identified using the Burrows-Wheeler Transform. These seeds are then *extended* in both directions using the Smith-Waterman algorithm. This algorithm is able to find the optimal alignment between two sequences given a particular scoring system that awards matching symbols, and penalizes gaps and mismatches. In the case of BWA-MEM, seeds consist of at least nineteen symbols. Seeds that are close to one another on the reference genome are collected together into a longer chain, refer to Figure 3. From all the extended seeds, the one with the highest score is selected as the final *alignment*.

3.1. BWA-MEM Profiling Results

Here we examine the run-time behavior of the BWA-MEM algorithm. The overall execution time of BWA-MEM is spent in three main computational kernels: Seed Generation, Seed Extension and Output Generation. The first two kernels have been mentioned in the previous section. During Output Generation, the final alignment is recomputed using the Needleman-Wunsch global sequence alignment algorithm, and the result is then written to disk. Profiling the application shows a behavior as given in Table 1. For the profiling, freely available input data sets from the GCAT [17] have been used. To investigate the impact of read length on the overall run-time behavior, input data sets with increasingly large read lengths have been used. From this, it is clear that the read length does not significantly affect BWA-MEM behavior. Note that the overall number of base pairs in the input data set is kept stable, which means that the data sets with longer read length contain fewer reads.

Two main candidates for acceleration become obvious: Seed Generation and Seed Extension. As Seed Generation seems to be more memory-bound, we have chosen the Seed Extension kernel as target for our acceleration efforts, as that function is computationally bound. Amdahl's law teaches us that accelerating only this function can provide a speedup of at most 1.7x. We can only achieve higher speedup if other kernels are accelerated as well, similar to what has been done in [18].

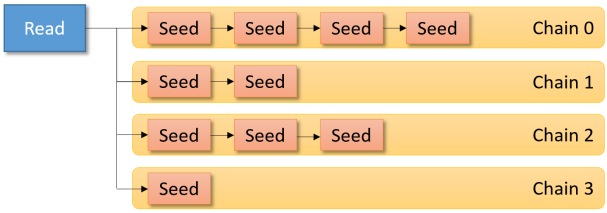

Figure 3: BWA-MEM Seed Generation can result in many seeds being identified for a single read. Seeds that are located in close proximity of one another on the reference are grouped into chains.

Table 1: Results of BWA-MEM algorithm profiling for GCAT data sets with various read length (tests performed on Intel Core i7-4790 @ 3.6 GHz)

	Read Length (in bp)				Total
Program Kernel	**100**	**150**	**250**	**400**	**bp**
Seed Generation	45%	47%	45%	43%	1.2b
Seed Extension	40%	40%	39%	38%	1.2b
Other	15%	13%	15%	18%	1.2b
Total Time	656 s.	594 s.	589 s.	612 s.	

3.2. Seed Extension Functional Details

Accelerating the Seed Extension kernel is an important focus of this article, hence a more in-depth explanation of this phase follows here. The pseudo code of Algorithm 1 describes the main algorithm. The Seed Extension stage consists of two main parts: an outer loop looping over all the seeds identified for the read during Seed Generation, and an Inexact Matching kernel, performing the Smith-Waterman-like functionality as needed.

There are no dependencies between reads and thus, reads can be processed in parallel by the algorithm. For each read, the groups of chains are processed iteratively, as the check for overlap between earlier found Alignment Regions (Line 4) introduces a dependency in the program order. This dependency is the main reason why the method typical Smith-Waterman GPU-implementations rely on is unsuitable in the case of BWA-MEM: these implementations obtain their performance by performing many Smith-Waterman alignments in parallel, which requires the alignments to be batched together in large numbers and, moreover, requires these alignments to be of approximately the same length for load balancing purposes. The highly dynamic nature of the Inexact Matching invocations makes both these requirements impractical to achieve, and would at least require a major algorithm overhaul, if at all possible. Since on average only one seed per chain requires extension, and a typical chain consists of about ten seeds, removing the overlap check (Line 4) and bruteforcing all extensions and selecting the correct ones afterwards would introduce too much overhead.

Algorithm 1 BWA-MEM Seed Extension Pseudo Code

Input: List of Chains of Seeds
Output: List of Alignment Regions
1: **for** (each Chain of Seeds) **do**
2: sort Seeds based on their length
3: **for** (each Seed) **do**
4: **if** (no overlap exists between current Seed and previously found Alignment Regions) **then**
5: perform Inexact Matching Left
6: perform Inexact Matching Right
7: store Alignment Region
8: **end if**
9: **end for**
10: **end for**

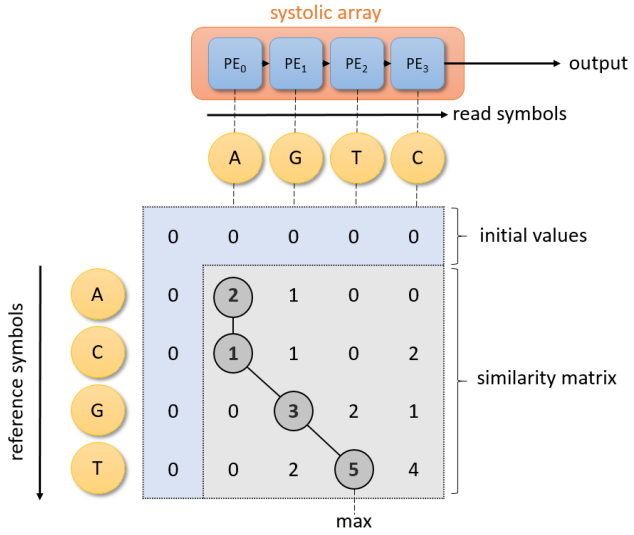

Figure 4: Smith-Waterman algorithm similarity matrix. The maximum score is indicated. As matrix entries only depend on top, top-left, and left neighbor, anti-diagonals can be processed in parallel. This makes the systolic array a natural implementation choice, whereby each column is processed by one Processing Element (or PE).

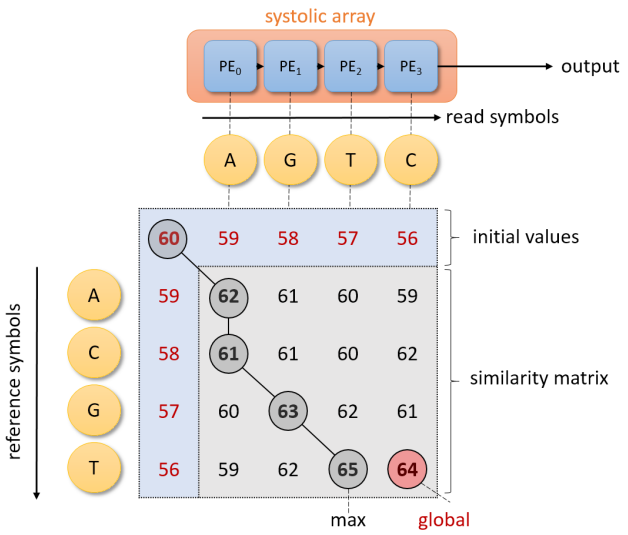

Figure 5: Inexact Matching algorithm similarity matrix with an initial score of 60. The maximum and global maximum scores are indicated. Differences as compared to the regular Smith-Waterman algorithm include the presence of initial values and computation of a global maximum score. The locations of both maxima are also calculated.

3.3. Inexact Matching Kernel

The Inexact Matching algorithm BWA-MEM uses is similar to the widely used Smith-Waterman algorithm. The Smith-Waterman algorithm is able to compute the optimal alignment between two subsequences, given a certain scoring scheme. The dynamic programming algorithm works by filling a similarity matrix. This is illustrated in Figure 4. The end result of the Smith-Waterman algorithm is a maximum score. Backtracking can be used to obtain the actual path through the similarity matrix that results in the final alignment. However, the algorithm is computationally expensive, being of O(READ × REFERENCE), making it infeasible to use the algorithm directly to align a short read to the complete human genome as this would result in unacceptable computation times. Hence, most mapping tools use an initial Seeding-phase to find likely mapping locations, and only then perform localized extension of these seeds.

There are a few key difference between the algorithm BWA-MEM uses and the normal Smith-Waterman algorithm. Two sequences are not compared in isolation; instead, we already have a seed that requires extension. This results in the fact that the initial scores are not set to zero, but have an initial value. Another important difference is that for BWA-MEM, we track a number of additional metrics: most importantly, the global maximum alignment value and the location where the maximum and global maximum are to be found. A nice characteristic of the similarity matrix is that its values only dependent on its top, left, and top-left neighbor. Therefore, values in anti-diagonals of the similarity matrix can be computed in parallel. This maps nicely to an implementation using a systolic array, where each column of the similarity matrix is processed by a Processing Element. The processing time is reduced from O(READ × REFERENCE) to O(READ + REFERENCE).

4. Accelerated Program Architecture

One key characteristic of BWA-MEM is the fact that each short read in the input is processed individually. Seed Generation and Seed Extension is performed in an interleaved fashion for each read. If this mechanism would have been kept in tact for the accelerated version, this would require many small invocations of the accelerated Seed Extension function, in turn resulting in much overhead and hence little (if any) speedup. Therefore, the program structure has been altered to process the input data in larger batches, where for each batch, first Seed Generation is performed for a large number of reads, then Seed Extension, and then Output Generation. Execution of these functions is overlapped with one another. This approach is explained in more details in [13].

4.1. Adaptive Load Balancing Strategy

When using an accelerator to offload a kernel, it is important to properly balance the accelerator with the host machine. If the host is too slow, the accelerator will be idle most of the time; whereas a too slow accelerator will result in the host being idle most of the time. Therefore, in order to maintain a good speedup, even when both accelerator and host are not perfectly balanced, it is important to use a load balancing strategy. This is especially important in computationally complex situations such as the extension of longer reads. An effective load balancing strategy is critical to achieve overall application level speedup. This has been implemented through the use of a Load Balancing Factor (LBF) parameter, which is able to minimize the idle time on the accelerator and the host by offloading only part, or all, of the work to the accelerator. More details can be found in [13]. By using such a load balancing scheme is the use of an accelerator always resulting in a speedup, even if the accelerator itself is relatively slow.

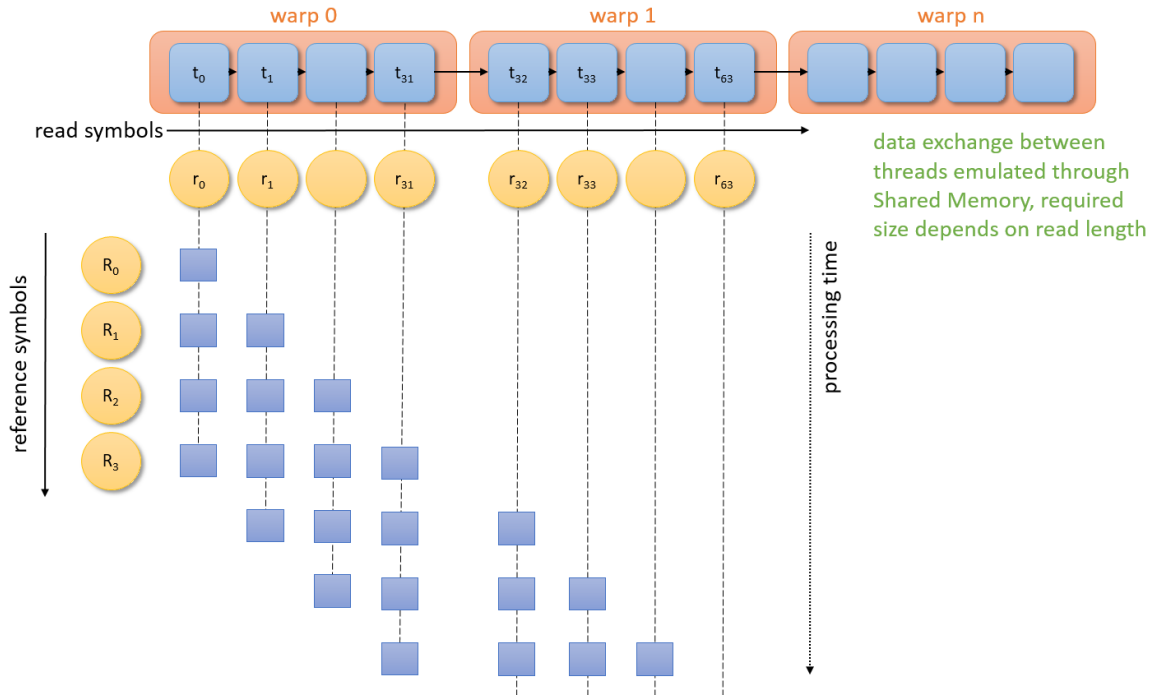

Figure 6: Overview of the ”wide” systolic array implementation, showing active threads when processing the similarity matrix with as many threads as there are read symbols. The data exchanged between the successive threads passes through Shared Memory, resulting in a dependence on the read length for the Shared Memory size. Note that, depending on matrix dimensions, many threads will be idle.

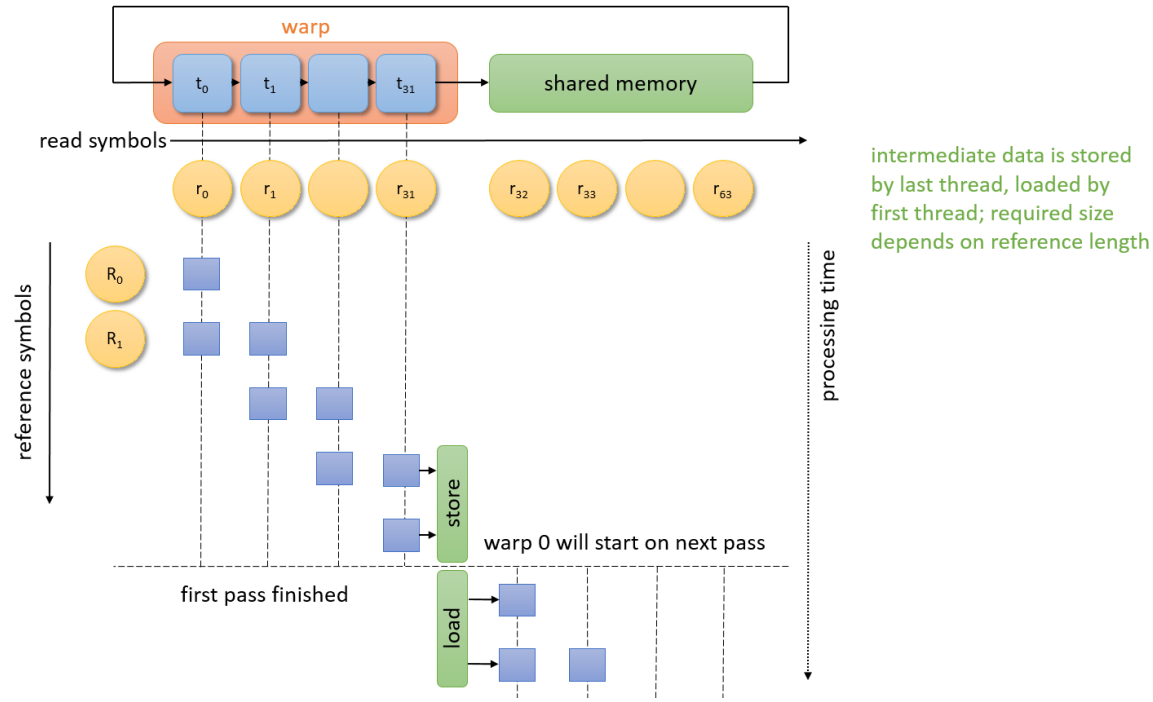

Figure 7: Overview of the single warp systolic array implementation, showing active threads when processing the similarity matrix with one warp. Multiple passes are made over the similarity matrix. Threads exchange data directly, eliminating the need to store this in Shared Memory. Data exchange between passes is stored in Shared Memory, resulting in a dependence on the reference length for the Shared Memory size. Note that threads will be less idle, as processing of parts of the grid can be skipped.

5. Design Space Exploration

In this section, a number of design-related topics are addressed: the GPU implementation is detailed, considering the GPU offloading strategy, the functional split in the Seed Extension function, and the implementation of the Inexact Matching algorithm; the FPGA implementation is briefly shown. Then, the efficiency of systolic array implementations is discussed.

5.1. GPU Implementation

Here, we describe three elements of the GPU-based implementation: the GPU offloading strategy, the functional division of the Seed Extension phase, and the details for the accelerated Inexact Matching function.

5.1.1. GPU Offloading Strategy

To offload work onto the GPU, results from the BWA-MEM Seed Generation phase are grouped into batches of reads (note: this is different from batching Inexact Matching). Each read in the batch of reads is sent to the GPU as a separate *thread block*. Hence, the GPU receives a grid of n thread blocks, where n is the number of reads to be processed. The GPU automatically schedules the reads onto its available execution resources, performing the Seed Extension. Thus, the GPU can be actively processing hundreds of reads at a time.

5.1.2. Seed Extension Functional Division

As explained in Section 3.2, the BWA-MEM Seed Extension phase consists of two distinct parts: the Inexact Matching algorithm, which is implemented as a systolic array, and the Seed Extension main loop, that loops over all the chains of seeds. These two parts are quite different from one another. The outer loop mostly performs control and branch operations to effectuate the looping over all seeds, performs the loading of the sequence and reference from main memory, and writes the eventual result back to memory. These tasks can easily be performed by a single thread, which most likely will be waiting for memory transactions to finish. In contrast, the Inexact Matching function is highly computationally intensive and can use as many threads as the systolic array allows for. Thus, our earlier implementation [13] makes a clear separation between both functions and utilizes CUDA Dynamic Parallelism to dynamically instantiate Inexact Matching kernels as needed. A number of kernels were implemented, each optimized for different matrix dimensions, and called appropriately. The underlying idea was that this should result in lower register and Shared Memory pressure, as each function only needs to allocate as many resources as it needs.

Unfortunately, our tests show that the dynamic kernel instantiation of CUDA Dynamic Parallelism brings about a large initialization penalty, making it unsuitable to use at this extreme scale, as for even a single read it can be called thousands of times, resulting in many millions of invocations during a typical program execution. Therefore, the implementation here does not make use of Dynamic Parallelism, instead executing the Seed Extension as one large monolithic kernel.

5.1.3. GPU-Based Inexact Matching

Although the main Seed Extension loop is interesting in its own right, the main challenge of the GPU-accelerated BWA-MEM Seed Extension function is the implementation of the Smith-Waterman-like Inexact Matching kernel. As discussed before, typical GPU implementations of Smith-Waterman perform many sequence alignments in parallel, mapping one alignment per thread. This facilitates the extraction of parallelism from the problem, but is contingent on the ability to sort and batch work, which is impractical.

Therefore, the other way of extracting parallelism is to make use of the possibility of harnessing the parallelism residing in the anti-diagonals of the similarity matrix, through use of a systolic array. This is the approach followed here. The systolic array Processing Elements (PEs) can be mapped either onto the read symbols (i.e., columns), or onto the reference symbols (i.e., rows) (refer to Figure 5). As careful analysis of BWA-MEM execution has shown that the reads are always shorter than the reference symbols, it is chosen to map PEs onto read symbols. This minimizes the number of PEs required.

Since we use NVIDIA CUDA as an implementation platform, it is important to explain some key concepts underlying the execution model of all NVIDIA GPUs. The basic unit of action in this model is the so-called *warp*, a cluster of typically 32 threads that all perform the same operation in any given clock cycle. Computational jobs are therefore always scheduled onto one or more warps, depending on how many threads they require. Therefore, two execution models were considered to implement our Smith-Waterman systolic array. Either a "wide" systolic array (refer to Figure 6) that uses as many threads as required, one for each processing element in the systolic array. Hence, a job is scheduled across as many warps as needed. The other model (refer to Figure 7) is to use only a single warp, or 32 threads. This in turn requires multiple passes over the similarity matrix to completely calculate all entries.

In a systolic array, during each computation step values are passed from one processing element to the next. Normally, in GPU implementations Shared Memory is used to communicate between threads. A key benefit of the single warp approach is the fact that threads within a single warp are able to access each others registers directly through intra-warp shuffle instructions, foregoing the requirement of communicating through Shared Memory. As Shared Memory is a very limited resource on the GPU, with typically only 64 kB being available per multiprocessor, this is a great benefit. The amount of Shared Memory used by a block of threads puts an upper bound on the number of thread blocks that can concurrently reside on a multiprocessor, so lower Shared Memory requirements directly result in higher performance. The single-warp implementation requires storage of the values on the boundaries of each pass, so that these values can be reused during the next pass over the similarity matrix. Therefore, the required Shared Memory amount is depended on the length of the reference query.

A secondary benefit of the single-warp implementation is that for a typical systolic array implementation, it is impossible to keep all processing elements busy. Depending on the exact

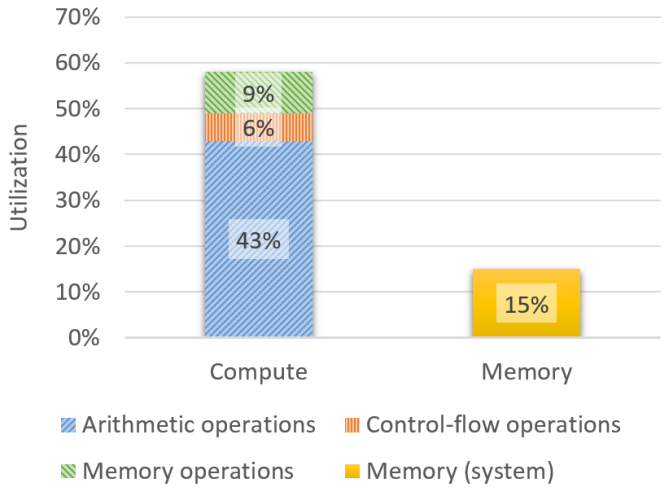

Figure 8: Output of NVIDIA Visual Profiler Latency Analysis for a test set with two hundred thousand reads of 150 bp. The implementation performance is mostly limited by the latency of arithmetic and memory operations, and by the number of resident blocks per multiprocessor.

	Transactions	Bandwidth
Shared Memory		
Shared Loads	752845366	80.064 GB/s
Shared Stores	730457002	77.683 GB/s
Shared Total	1483302368	157.747 GB/s
L2 Cache		
Reads	445401451	11.842 GB/s
Writes	121603141	3.233 GB/s
Total	567004592	15.075 GB/s
Unified Cache		
Local Loads	110882373	2.948 GB/s
Local Stores	65759941	1.748 GB/s
Global Loads	643181064	5.62 GB/s
Global Stores	50114488	1.332 GB/s
Texture Reads	1369309321	36.406 GB/s
Unified Total	2239247187	48.055 GB/s
Device Memory		
Reads	122183405	3.249 GB/s
Writes	54954108	1.461 GB/s
Total	177137513	4.71 GB/s

Figure 9: Output of NVIDIA Visual Profiler Memory Bandwidth Analysis for a test set with two hundred thousand reads of 150 bp. Most of the bandwidth is used during the Inexact Matching by the Shared Memory. Device memory bandwidth utilization is low, as caching through texture memory of the reference and input data is effective.

dimensions of the similarity matrix, many processing elements may not have any useful work to do for large parts of the time. As will be shown in the next sections, a single-warp implementation is able to circumvent, or at least reduce, this problem by skipping parts of the similarity matrix.

5.1.4. Implementation Architecture

Due to the above reasons, the single-warp systolic array design is implemented. This implementation is able to branch between two single-warp Inexact Matching implementations: one function for extensions that fit completely inside a single warp, or in other words, the read symbols for the extension are 32 symbols or less; and one function that can process longer extensions in multiple passes. The benefit of this setup is that the shorter extensions can skip the intermediate data storage step, saving bandwidth and executed instruction, but not Shared Memory, as this is statically allocated on a thread block basis for the Seed Extension function as a whole. As Shared Memory and register usage is the aggregate of all functions in the kernel, it needs to be carefully balanced in order to maximize occupancy. The register count was fixed to use 64 registers per thread. The maximum number of rows that are allowed on the reference was chosen specifically with the input data set in mind, as this influences the amount of Shared Memory each thread block requires. For example, for a data set with reads of 150 bp, the maximum reference read length can be set to 131 symbols, as the maximum seed length is 19 and it can be shown that the part of the similarity matrix corresponding to those reference symbols that exceed the input read length will not contribute to the result. In the case of 131 symbols, one thread block uses 2 kB of Shared Memory. Hence, up to 32 thread blocks can be resident per multiprocessor. If the maximum rows are set to 381, which is required for test data with 400 bp reads, the Shared Memory allocation increases to 5.4 kB per thread block, resulting in only at most 11 resident thread blocks per multiprocessor. Unless specifically mentioned otherwise, our tests use implementations tuned to the specific input read length to optimize occupancy.

Figures 8 and 9 show detailed results from analysis of a smaller test, obtained with the NVIDIA Visual Profiler, a cross-platform profiling tool to help optimize CUDA applications [19]. The results show that the performance is mostly limited by latency of arithmetic and memory instructions. The memory subsystem utilization is shown in Figure 9. Most of the bandwidth is directed onto the Shared Memory subsystem, holding temporary data of the systolic array while calculating the Seed Extension similarity matrix. The GPU caching is effective, as device memory bandwidth is substantially lower than overall unified cache bandwidth. The device memory bandwidth utilization is very low, which corresponds to our expectations for such a computationally-limited application: a Seed Extension algorithm invocation only requires two sequences, which for a read length of 150 bp only amounts to $2x150x2 = 600$ bits. Although in our implementation, the sequences are not ideally packed, this explains the observed low external memory bandwidth requirements.

For the latest NVIDIA GPU architectures offering Compute Capability 5.0+, a multiprocessor can have up to 2048 resident threads [20]. However, since at the same time only 32 blocks can be resident per multiprocessor, this means that optimal occupancy can only be obtained for thread blocks with at least 64 threads. Since this implementation's thread blocks contain only 32 threads, occupancy is limited to at most 50%. In practice, up to about 35% occupancy is realized. Earlier Compute Capa-

bility versions were even more restrictive, only allowing sixteen resident blocks per multiprocessor for architectures with Compute Capability 3.0+, or even only eight resident blocks per multiprocessor for earlier architectures. This would have a direct impact on the efficiency of this implementation.

5.2. FPGA Implementation

The FPGA implementation uses a batching strategy similar to the one used by the GPU as described in Section 5.1.1. Of course, unlike the GPU implementation, which executes the Seed Extension kernel on the underlying GPU substrate, the FPGA implementation consists of a custom bitstream tailor-made for the application. Our design consists of six physical Seed Extension modules, each consisting of a systolic array with 131 Processing Elements. The systolic array contains *early exit points* at Processing Elements 100, 66, and 33. The function of these early exit points will be described in more detail in the next section. Each Seed Extension module is joined by a module that performs the Seed Extension main loop, which loops over the chains of seeds. The rest of the FPGA area is filled with the memory controller, PCI-Express controller, and logic that distributes reads over the modules. More details on the implementation can be found in [12].

5.3. Classification of Systolic Array Inefficiency

The efficiency of a systolic array is heavily dependent on the length of the read and target, as compared to the length of the systolic array itself. Since the read symbols are mapped one-to-one onto systolic array processing elements, a read that is much shorter than the systolic array causes most of the processing elements to remain idle. Moreover, the output still needs to traverse the entire systolic array, causing further inefficiency. A short target sequence causes the systolic array to be occupied until it fully traverses the array. In general, it can be summarized that systolic arrays perform optimally when the read sequence is exactly the same length as the systolic array length, and the target sequence is as long as possible.

This is illustrated by Figure 10, which shows only some parts of the systolic array are contributing to the calculation of the final result. In the figure, each row represents a new time cycle in calculation of the similarity matrix. We can categorize the above-mentioned issues into four categories:

A - Waiting for Input Data: As each Processing Element passes its result onto the next PE, it takes a number of cycles before all Processing Elements can start their calculations. The further along the PE is in the array, the longer it has to wait before it can start its calculations. This area is indicated by area A, the time a PE has to wait for input before it can join the calculations.

B - Waiting for All PEs to Finish: Every cycle, a new symbol of the target sequence is inserted into the systolic array, until all target symbols are inserted. By then, the first PE is finished, however, the overall processing is not. This is indicated as area B, the cycles that while some PEs are already finished, others need to finish as well.

C, D - Imbalanced Read vs Systolic Array Length: Each read symbol is mapped onto a Processing Element. If the read sequence is shorter than the systolic array length, some PEs will remain idle during the entire computation. However, the results still need to flow through the systolic array until the output data can be extracted. Therefore, *early exit points* can be placed inside the systolic array to bypass the need to traverse the entire array. Area C indicates the imbalance between read length and exit point location, area D the remaining portion of the array that remains idle.

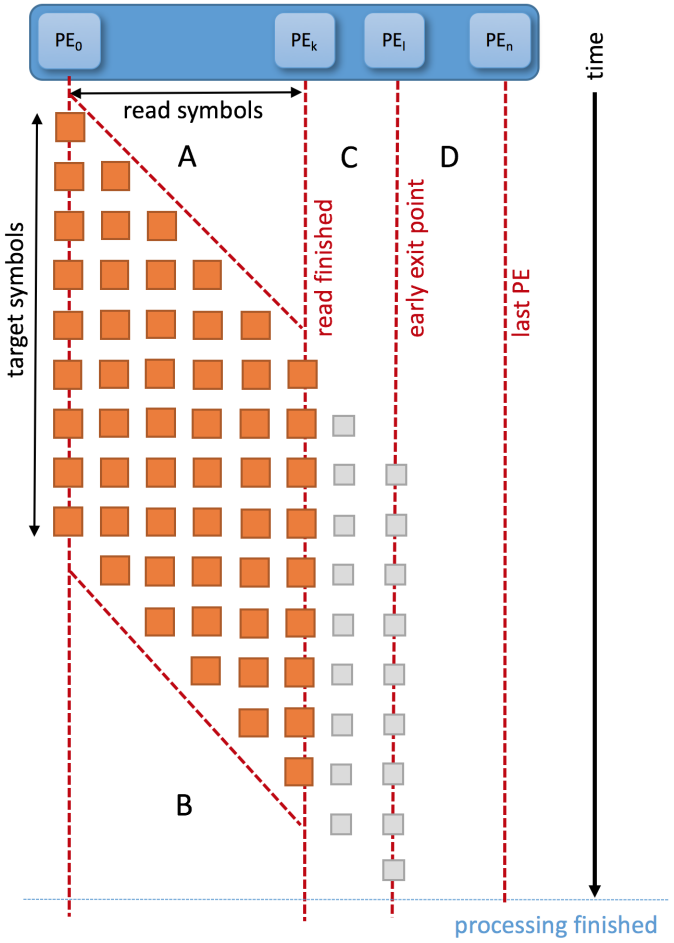

Figure 10: The efficiency of a systolic array is heavily dependent on the length of the read and target, as compared to the length of the systolic array itself. Areas indicated by A, B, C, and D are areas of inefficiency, where some or all of the Processing Elements are not contributing useful work. Reducing these areas can greatly improve systolic array efficiency.

All together, it is clear that there are many situations in which a systolic array operates only at partial capacity. However, having such a categorization allows us to come up with strategies to eliminate or reduce the impact each of these has. In [11], a number of systolic array architectures were introduced: Variable Physical Length (VPL), Variable Logical Length (VLL), and Variable Logical+Physical Length (VLPL). A VPL systolic array is simply to have a number of systolic arrays work in parallel, each with a different number of Processing Elements. This allows us to reduces area D-type inefficiencies, as this inefficiency is caused by the mismatch in systolic array and read length. A VLL systolic array allows a systolic array of a larger

Table 2: Comparison of speedup and throughput of accelerated BWA-MEM v0.7.8 implementations for a data set with 150 bp reads

		Accelerated Phase		Overall Application		
Source	**Platform and Method**	**Execution Time**	**Speedup**	**Execution Time**	**Speedup**	**Throughput**
Our Work	Software-Only (Original BWA-MEM)	237 s	-	552 s	-	2.2 Mbp/s
	Seed Extension on FPGA [12]	**129 s**	**1.8x**	**272 s**	**2.0x**	**4.5 Mbp/s**
	Seed Extension on GPU [14]	144 s	1.6x	278 s	2.0x	4.3 Mbp/s
Chang [15]	*Seed Generation on FPGA*	*N/A*	*4x*	*N/A*	*1.26x*	*N/A*
Chen [16]	*Software-Only (CW-BWAMEM)*	*N/A*	-	*N/A*	-	*1.2 Mbp/s*[1]
	Seed Extension on FPGA	*N/A*	*10.5x*	*N/A*	*3x*	*3.6 Mbp/s*[1]

[1] Reported speedup is 2.4 Mbp/s and 7.2 Mbp/s for 2x Intel Xeon E5-2620v3, which is twice as fast as an Intel Core i7-4790

size to act as if it is of shorter length, by including the above-mentioned early exit points. These are points in the array that are able to output its results, bypassing the need to pass results through the entire array. Part of the array would still be idle during the entire computation, however, the total number of cycles is partially reduced. The VLL-array reduces the area-C. Finally, area A and area B inefficiencies could be circumvented if the Processing Elements of a systolic array were allowed to work on *different* reads, in effect pipelining multiple reads after one another.

The FPGA implementation uses a VLL approach, where six modules are used with 131 Processing Elements, each with early exit points at 131, 100, 66. In contrast, the GPU implementation can be considered to be a VPL implementation, as the multi-pass approach results in an effective systolic array length of any multiple of 32 PEs. Moreover, as can be seen in Figure 7, each pass does not cover the complete 32 PE-wide stripe, but is narrowed down even further by starting at the relevant cycle and stopping as soon as possible, reducing the area A and area B regions. This results for an 96x100 alignment in an 48% efficiency improvement over computing the entire region.

6. Experimental Results

All tests have been performed using a system with an Intel Core i7-4790 at 3.6 GHz with eight logical cores (four physical cores), with both SpeedStep and Hyper-Threading enabled. The system contains 16 GB of DDR3 memory. To obtain the GPU results, we used an NVIDIA GeForce GTX 970 with 1664 CUDA cores with a maximum clock frequency of up to 1.25 GHz and 4 GB of on-board RAM. CUDA version 7.5 was utilized. The FPGA results were obtained using the same base system, but with the server-grade Alpha Data ADM-PCIE-7V3 card with a Xilinx Virtex-7 XC7VX690T-2 and 16 GB of on-board RAM [21], which contains six Seed Extension modules at 160 MHz.

For testing purposes, BWA-MEM version 0.7.8 was used. Tests were performed using data that is freely available from the Genome Comparison & Analytic Testing (GCAT) framework [17]. Pair-ended large indel alignment data sets were used with various read lengths: gcat38 (100bp-pe-large-indel), gcat42 (150bp-pe-large-indel), gcat46 (250bp-pe-large-indel), and gcat50 (400bp-pe-large-indel). Each data set contains about 1.2 billion base pairs. In other words, data sets with more base pairs per read contain fewer reads overall, so that the total amount of base pairs remains the same. The reads were aligned against the reference human genome (UCSC HG19).

As mentioned in Section 2, in bioinformatics, a key requirement is exactness of results. For example, population studies can take many years to complete. For these studies, it is critical that the algorithm does not change over a long period of time. Tests run using the online GCAT portal that allows us to compare read aligner quality [22] show that the results from our implementation are indistinguishable from the software-only BWA-MEM.

6.1. Performance Results

Performance results are summarized in Table 2. Not only execution time is given, but the application performance is also expressed in throughput in millions of base pairs per second, to facilitate cross-algorithm, cross-data set and cross-platform comparisons. Both the GPU-accelerated implementation and the FPGA-accelerated implementation are able to offer an 2x speedup, compared to software-only execution, with the FPGA-accelerated implementation offering slightly higher performance. Most likely, this is due to slightly lower overhead from the FPGA driver as compared to the CUDA driver.

To compare performance between the various accelerated implementations mentioned in Section 2, we also included the results from Chang [15] and Chen [16]. Chang accelerates the BWA-MEM Seed Generation phase using the Intel-Altera Heterogeneous Architecture Research Platform, which contains an Altera Stratix V FPGA. Like our work, Chen [16] accelerates the BWA-MEM Seed Extension phase, using the same AlphaData FPGA board. Chang is able to achieve an overall application-level speedup of 1.26x, whereas Chen claims an overall application-level speedup of 3x. However, their baseline of comparison is the *Cloud-Scale* BWA-MEM implementation, which performs about 50% slower than regular BWA-MEM [23]. Moreover, their experimental platform, a dual node Intel Xeon E5-2620v3, offers about twice the performance as the system used here. In practice, we estimate that their implementation achieves about 80% of the performance obtained by our implementations, when using the same system. The fact that

Table 3: GPU SMM requirements and relative speedup over software-only execution for data sets with increasing read length

	GPU SMM Utilization		Speedup Over Software-Only Execution Per Data Set			
Supported Read Length	**Shared Memory**	**Resident Blocks**	**gcat38 (100bp)**	**gcat42 (150bp)**	**gcat46 (250bp)**	**gcat50 (400bp)**
Up to 100 bp reads	1.3 kB	32	200%	-	-	-
Up to 150 bp reads	2.0 kB	32	201%	197%	-	-
Up to 250 bp reads	3.3 kB	19	200%	194%	198%	-
Up to 400 bp reads	5.4 kB	11	202%	195%	188%	168%
Up to 570 bp reads	8.0 kB	8	179%	194%	174%	127%
Up to 1150 bp reads	16.0 kB	4	150%	160%	113%	75%

they are able to obtain a 3x speedup indicates that the performance profile of CS-BWAMEM is substantially different from regular BWA-MEM, most likely being much more limited by the Seed Extension phase.

The execution time for the Accelerated Phase considers only the kernel execution time, not including data transfer times, as performance in the limiting case will only be determined by the computational part of the Seed Extension. Although in our current implementation we do not overlap data transfer and computation, this would be relative straightforward to implement. Moreover, to illustrate the relative insignificance to this particular application, total data transfer time excluding the transfer of the reference genome, which is done only once at the start of program execution, is less than one second in total.

6.2. Performance Impact of Read Length

As explained in Section 5.1.4, the multi-warp GPU implementation requires Shared Memory directly proportional to the number of rows that can be stored from the similarity matrix. This, in turn, is directly related to the maximum supported read length. The Shared Memory utilization is one of the factors that determines the number of warps that can be scheduled simultaneously onto an SMM, so this directly impacts efficiency. To observe the effect of this, tests have been run with implementations tuned to support different maximum read lengths, against data sets with various read lengths. The results are summarized in Table 3. It is clear that Shared Memory requirements scale proportional to the supported read length. This is inversely proportional to the maximum simultaneous Resident Blocks per SMM. Note, however, that regardless of Shared Memory usage, at most 32 blocks can be resident at any one time.

The impact on the overall application-level speedup is clear: as the supported read length increases, GPU utilization decreases, resulting in worse performance. Processing longer reads is also more GPU-intensive, as only for data sets with up to 250bp, the full two-fold performance increase is attained. The 400 bp data set only achieves an at most 1.7x speedup, and in one case, even results in a slowdown, instead of a speedup. There are two reasons for this behavior. First, the GPU implementation is not a true systolic array, as for longer reads, multiple passes are necessary. Hence, performance scales not as O(READ + REFERENCE), but as O(READ × REFERENCE). Second, the CPU Seed Extension implementation uses a mechanism whereby it only processes a small fraction of the similarity matrix, resulting in more efficient operation (see [11] for details).

6.3. Scalability and Impact of Load Balancing

Apart from overall performance on the test platform, it is also interesting to analyze the scalability of the implementations. Here, scalability is defined as the number of CPU cores that the implementation is effectively able to accelerate while still providing the maximum speedup. In simplified terms, this can be approximated by considering the time required for the Seed Extension phase, which is performed on the GPU and hence insensitive to CPU core count, and regarding this as a lower bound to overall application execution time. Assuming overall execution time scales linearly in processor core count, which has been observed to hold for CPU core count up to at least sixteen cores, the maximum number of logical CPU cores that can be effectively accelerated can thus be estimated.

The scalability results are visually depicted in Figure 11. This graph shows the relative speedup from using the GPU-accelerated implementation compared to execution on a machine with the same number of CPU cores. Note that, obviously, execution on an eight core system will be faster than on a four core system. The graph shows the normalized speedup obtain from using the GPU. For data sets with 150bp reads, maximum speedup is supported for up to twenty-two logical CPU cores. After that, the relative speedup gradually decreases as execution time no longer decreases due to being limited by the GPU-only Seed Extension phase, which is unaffected by CPU core count. For the 400bp data set, only up to twelve logical CPU cores can be supported.

Performance can be improved by using the adaptive load balancing algorithm described in Section 4.1. This ensures optimal benefit from the use of acceleration, by dividing the work between host and accelerator in such a way as to minimize idle times. Thus, it can prevent GPU-constrained situations to result in overall application-level slowdown, by distributing Seed Extension work between the host and the GPU. This results in a more graceful drop-off in performance, as can be seen in Figure 11. More importantly, it should prevent an overall application-level slowdown. Under GPU-constrained situations, performance can improve by up to +45%.

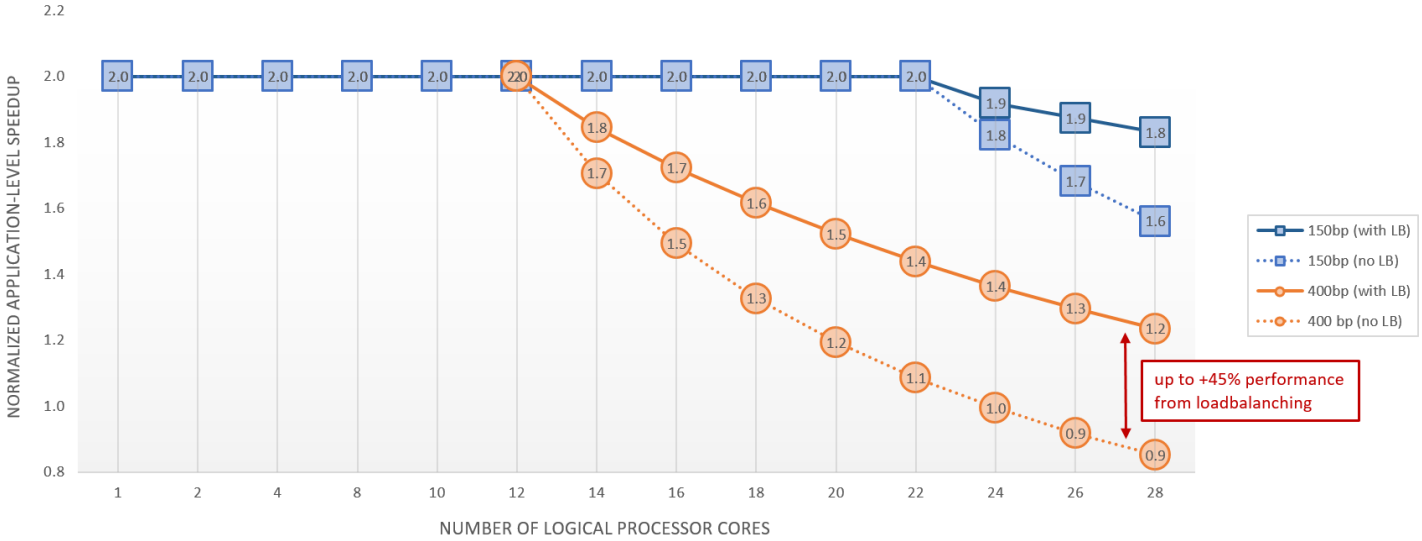

Figure 11: Estimated application-level speedup depending on CPU core count for 150 bp and 400 bp data set. Load balancing improves performance in GPU-constrained scenarios, ensuring application speedup in all cases.

7. Discussion

7.1. Impact of Read Length and Load Balancing

From the above results, it becomes clear how longer read lengths can impact overall GPU-acceleration capability, as the Shared Memory requirements for longer reads greatly reduces the GPU's ability to concurrently execute tasks. The current implementation is able to achieve a maximum two-fold speedup for data sets with up to 250 bp read length. For longer reads, the system is no longer capable to provide this full speedup. A faster GPU model could be used to attain full performance.

Under normal circumstances, the GPU is sufficiently fast to completely hide the Seed Extension phase by overlapping its execution with the other tasks performed by the host CPU. However, the load balancing algorithm can greatly improve performance for scenarios where the system is imbalanced, which is increasingly the case for more strenuous long read data sets. In such a case, the load balancing helps to sustain the acceleration capability of the system.

Finally, batching the work sent to the accelerator in larger groups is often a base requirement to obtain good performance from accelerators to overcome communication overhead. In the case of BWA-MEM, the code transformation whereby Seed Generation results are batched together before being sent to the accelerator to perform Seed Extension is a prerequisite of getting a performance benefit out of a GPU. Depending on the program structure, this can take significant engineering effort. A more closely coupled system, such as the Intel-Altera HARP, could reduce or even eliminate the required effort.

7.2. Systolic Array Efficiency

The importance of improving the efficiency of a systolic array greatly increases with increased read length, as longer systolic arrays suffer much more from the inefficiencies as identified in Section 5.3. Both implementations described here use a different mechanism to improve their efficiency.

The GPU implementation can be considered as a VPL implementation, as the multi-pass approach results in an effective systolic array length of any multiple of 32 PEs. It would have been infeasible to use a single-pass implementation, as such an implementation would require a huge amount of shared memory to emulate the data exchange between Processing Elements. Moreover, for long reads, area A and B-type inefficiencies would be quite large. A multi-pass implementation as used here is able to avoid both these drawbacks. The effectiveness of the VPL-approach is illustrated in Table 4. This VPL-based approach, combined with the technique to only calculate the relevant parts of the stripe, results for increasingly long reads into great improvements in efficiency. Moreover, note that the 50% efficiency the normal systolic array attains is a best case scenario, as an imbalance in systolic array length and read length would greatly reduce efficiency even further.

Table 4: VPL-based systolic array compared to normal systolic array

Read	Target	Useful Cycles	Normal SA Cycles	GPU VPL SA Cycles	Gain
100	100	10'000	20'000	16'896	+18%
150	150	45'000	45'000	29'120	+55%
250	250	62'500	125'000	72'192	+73%
400	400	160'000	320'000	179'712	+78%
570	570	324'900	649'800	346'752	+87%
1150	1150	1'322'500	2'645'000	1'361'664	+94%

In contrast, the FPGA uses a VLL-based approach, where six modules are used with 131 Processing Elements, each with early exit points at 131, 100, 66. This helps reduce area C-type inefficiencies when shorter read lengths are processed. However, for longer read lengths such as the ones considered here, a multi-pass solution can be considered to be almost mandatory. Given that for a typical data set, read length varies considerably. Then, if only a fraction of reads are long reads, this still

requires a systolic array that is able to process reads with the longest length, otherwise a single-pass architecture is unable to process these long reads. Then, apart from the longer processing time, this systolic array would also take up a great amount of the available physical area on the FPGA. For example, instead of six modules of length 131, we would be able to fit only one module with length of about 800 Processing Elements.

8. Conclusion

This article describes a hardware accelerated implementation of the BWA-MEM genomic mapping algorithm, one of the most widely used read mapping tools and a linchpin in many genomics pipelines. The GPU-based implementation has been modified to allow it to process sequences with longer read sizes, a capability that will become necessary as sequencers are expected to generate longer reads in the near future. However, longer read lengths impact the effectiveness of the GPU-based acceleration, as the increased requirements on Shared Memory reduces the GPUs ability to execute tasks in parallel. This makes efficiency improvements to the underlying architecture even more important.

The Seed Extension phase is one of the three main BWA-MEM program phases, which requires between 30%-50% of overall execution time. Offloading this phase onto the GPU provides an up to two-fold speedup in overall application-level performance. For data sets that use the typical read length of 150 bp, the use of the GPU-accelerated implementation can offer this maximum two-fold speedup for a system with up to twenty-two logical cores, as compared to software-only execution. This can save days of processing time on the enormous real-world data sets that are typical of NGS sequencing. Data sets with up to 250bp can be accelerated with the maximum two-fold application-level speedup. Load balancing can be used 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 45%, compared to non-load balanced execution.

A number of inefficiencies is identified common to all systolic array implementations. These inefficiencies are classified into different categories, and ways are shown to ameliorate the drawbacks of each of these categories. The multi-pass based implementation used by the GPU implementation can be considered a Variable Physical Length system, thus circumventing most of the inefficiencies that are related to systolic arrays that contain large numbers of Processing Elements, increasing efficiency by up to 94% compared to a regular systolic array implementation. To further improve systolic array efficiency, we are working on a pipelined read implementation that allows the systolic array to work on more than one read at a time, thus completely eliminating the area A and area B-type inefficiencies that result from Processing Elements waiting on input, or waiting for the processing to finish.

Funding Sources

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors. Bluebee provided the hardware and other testing equipment.

References

[1] Z. Stephens, S. Lee, F. Faghri, R. Campbell, C. Zhai, M. Efron, R. Iyer, M. Schatz, S. Sinha, G. Robinson, Big Data: Astronomical or Genomical?, PLoS Biology 13 (7).

[2] Illumina, HiSeq X Specification Sheet, `http://www.illumina.com/content/dam/illumina-marketing/documents/products/datasheets/datasheet-hiseq-x-ten.pdf`, accessed: 2015-07-15 (2015).

[3] Illumina, Illumina Sequencing Systems, `http://www.illumina.com/systems/sequencing.html`, accessed: 2016-11-16 (2016).

[4] H. Li, Aligning Sequence Reads, Clone Sequences and Assembly Contigs with BWA-MEM, arXiv preprint arXiv:1303.3997.

[5] C.-M. Liu, T. Wong, E. Wu, R. Luo, S.-M. Yiu, Y. Li, B. Wang, C. Yu, X. Chu, K. Zhao, R. Li, SOAP3: Ultra-Fast GPU-Based Parallel Alignment Tool for Short Reads, Bioinformatics 28 (6) (2012) 878–879.

[6] Y. Liu, B. Schmidt, D. L. Maskell, CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows-Wheeler transform, Bioinformatics 28 (14) (2012) 1830–1837.

[7] L. Ligowski, W. Rudnicki, An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases, in: Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, IEEE, 2009, pp. 1–8.

[8] L. Hasan, M. Kentie, Z. Al-Ars, DOPA: GPU-based protein alignment using database and memory access optimizations, BMC research notes 4 (1) (2011) 261.

[9] S. Manavski, G. Valle, CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment, BMC bioinformatics 9 (Suppl 2) (2008) S10.

[10] L. Di Tucci, K. O'Brien, M. Blott, M. D. Santambrogio, Architectural optimizations for high performance and energy efficient smith-waterman implementation on fpgas using opencl, in: 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE), IEEE, 2017, pp. 716–721.

[11] E. Houtgast, V. Sima, K. Bertels, Z. Al-Ars, An FPGA- Based Systolic Array to Accelerate the BWA-MEM Genomic Mapping Algorithm, in: Intl. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation, 2015.

[12] E. Houtgast, V. Sima, G. Marchiori, K. Bertels, Z. Al-Ars, Power-Efficient Accelerated Genomic Short Read Mapping on Heterogeneous Computing Platforms, in: Proc. 24th IEEE International Symposium on Field-Programmable Custom Computing Machines, Washington DC, USA, 2016.

[13] E. Houtgast, V. Sima, K. Bertels, Z. Al-Ars, GPU-Accelerated BWA-MEM Genomic Mapping Algorithm Using Adaptive Load Balancing, in: Architecture of Computing Systems–ARCS, Springer, 2016, pp. 130–142.

[14] Houtgast, EJ and Sima, V and Bertels, KLM and Al-Ars, Z, An Efficient GPU-Accelerated Implementation of Genomic Short Read Mapping with BWA-MEM, in: Proc. International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies, Hong Kong, China, 2016.

[15] M.-C. F. Chang, Y.-T. Chen, J. Cong, P.-T. Huang, C.-L. Kuo, C. H. Yu, The SMEM Seeding Algorithm Acceleration for DNA Sequence Alignment, in: Proc. 24th IEEE International Symposium on Field-Programmable Custom Computing Machines, Washington DC, USA, 2016.

[16] Y.-T. Chen, J. Cong, Z. Fang, J. Lei, P. Wei, When apache spark meets fpgas: A case study for next-generation dna sequencing acceleration, in: 8th USENIX Workshop on Hot Topics in Cloud Computing, 2016.

[17] G. Highnam, J. J. Wang, D. Kusler, J. Zook, V. Vijayan, N. Leibovich, D. Mittelman, An Analytical Framework for Optimizing Variant Discovery from Personal Genomes, Nature comm. 6.

[18] N. Ahmed, V. Sima, E. Houtgast, K. Bertels, Z. Al-Ars, Heterogeneous Hardware/Software Acceleration of the BWA-

MEM DNA Alignment Algorithm, in: Proc. of the IEEE/ACM Intl. Conf. on Computer-Aided Design, ICCAD, 2015.

[19] NVIDIA, NVIDIA Visual Profiler, https://developer.nvidia.com/nvidia-visual-profiler, accessed: 2016-01-14 (2016).

[20] NVIDIA, CUDA C Programming Guide, http://docs.nvidia.com/cuda/cuda-c-programming-guide, accessed: 2016-01-20 (2016).

[21] Alpha Data, Alpha Data ADM-PCIE-7V3 Product Information, http://www.alpha-data.com/dcp/products.php?product=adm-pcie-7v3, accessed: 2015-12-14 (2015).

[22] Bioplanet.com, Genomic Comparison and Analytic Testing, http://www.bioplanet.com/gcat, last visited: 2016-11-16 (2016).

[23] Y.-T. Chen, J. Cong, J. Lei, S. Li, M. Peto, P. Spellman, P. Wei, P. Zhou, CS-BWAMEM: A fast and scalable read aligner at the cloud scale for whole genome sequencing.