On Hardware-Accelerated Maximally-Efficient Systolic Arrays

Acceleration and Optimization of Genomics Pipelines Through Hardware/Software Co-Design

Houtgast, Ernst

DOI
10.4233/uuid:b1f4d743-95c1-45ec-ae91-af62224e1d7c

Publication date
2019

Document Version
Final published version

Citation (APA)
Houtgast, E. (2019). *On Hardware-Accelerated Maximally-Efficient Systolic Arrays: Acceleration and Optimization of Genomics Pipelines Through Hardware/Software Co-Design*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:b1f4d743-95c1-45ec-ae91-af62224e1d7c

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

On Hardware-Accelerated Maximally-Efficient Systolic Arrays

Acceleration and Optimization of Genomics Pipelines Through Hardware/Software Co-Design

On Hardware-Accelerated Maximally-Efficient Systolic Arrays

Acceleration and Optimization of Genomics Pipelines Through Hardware/Software Co-Design

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus, prof. dr. ir. T.H.J.J. van der Hagen
chair of the Board for Doctorates
to be defended publicly on
Monday 11 November 2019 at 10:00 o'clock

by

Ernst Joachim Houtgast

Master of Science in Computer Engineering,
Delft University of Technology, the Netherlands
born in The Hague, the Netherlands

This dissertation has been approved by the promotors.

Composition of the doctoral committee:

Rector Magnificus,	Chairman
Prof. dr. ir. K.L.M. Bertels,	Delft University of Technology, promotor
Dr. ir. Z. Al-Ars,	Delft University of Technology, promotor
Independent members:	
Prof. dr. ir. E.E.E. Charbon,	Delft University of Technology
Prof. dr. E. Eisemann,	Delft University of Technology
Prof. dr. C. Witteveen,	Delft University of Technology
Prof. dr. E.P.J.G. Cuppen,	Universitair Medisch Centrum Utrecht
Dr. ir. T.G.R.M. van Leuken,	Delft University of Technology

Keywords: Acceleration, BWA-MEM, FPGA, GPU, Heterogeneous Systems, Pairwise Sequence Alignment, Smith-Waterman, Systolic Array

Printed by: **TBD**

Cover image: **TBD**

ISBN 978-94-6366-203-1

An electronic version of this dissertation is available at
`http://repository.tudelft.nl/`.

"Learning never exhausts the mind."

— Leonardo da Vinci

Summary

DEVELOPMENTS in sequencing technology have drastically reduced the cost of DNA sequencing. The raw sequencing data being generated requires processing through computationally demanding suites of bioinformatics algorithms called genomics pipelines. The greatly decreased cost of sequencing has resulted in its widespread adoption, and the amount of data that is being generated is increasing exponentially, projected to soon rival big data fields such as astronomy. Therefore, acceleration and optimization of such genomics pipelines is becoming increasingly important.

The BWA-MEM genomic mapping algorithm is a critical first step of many genomics pipelines, as it maps the raw input sequences onto a reference genome, thereby reconstructing the sample's original genetic assembly. A major part of overall BWA-MEM execution time is spent performing Seed Extension, an algorithm closely related to the Smith-Waterman pairwise sequence alignment algorithm. The standard approach for the heterogeneous acceleration of the Smith-Waterman algorithm is to map it onto a systolic array architecture to compute elements of the similarity matrix in parallel. In order for systolic arrays to operate at high efficiency, they require long sequences to be aligned to one another. The BWA-MEM algorithm, in contrast, typically generates very short sequences that then require pairwise alignment through the Seed Extension algorithm. Therefore, in this dissertation, various techniques to improve the efficiency of systolic arrays for short sequence lengths are proposed.

The Variable Logical Length, the Variable Physical Length, and the Variable Logical and Physical Length systolic array architectures are proposed to eliminate the dependence of systolic array efficiency on read sequence length. To eliminate its dependence on reference sequence length, a streaming, implicit synchronizing architecture is introduced. Together, these techniques result in a maximally-efficient systolic array. A Seed Extension kernel has been implemented on both FPGA and GPU with a threefold kernel-level improvement to execution time, resulting in the first FPGA-accelerated and the first GPU-accelerated implementation of BWA-MEM with an overall end-to-end twofold application-level speedup. Moreover, a Smith-Waterman implementation has been developed on FPGA using the above efficiency improvements to the systolic array architecture, resulting in an implementation that has a performance of 214 GCUPS and that is able to attain 99.8% efficiency, which is the highest reported efficiency and performance of any FPGA-accelerated Smith-Waterman implementation to date. Finally, various aspects of these designs are evaluated, including power-efficiency and design-time.

Samenvatting

RECENTE ontwikkelingen in sequencing-technologie hebben geresulteerd in drastische verlaging van de kosten voor DNA-sequencing. De onbewerkte gegevens die worden gegenereerd, vereisen verwerking door middel van computationeel veeleisende bioinformatica-algoritmen, de zogeheten genomics-pipelines. De sterk verminderde kosten voor sequencing hebben geresulteerd in wijdverspreid gebruik en de hoeveelheid gegevens die wordt gegenereerd neemt exponentieel toe, met als gevolg een groei die binnenkort big data-velden zoals astronomie zal evenaren. Hierdoor wordt versnelling en optimalisatie van dergelijke genomics-pipelines steeds belangrijker.

Het BWA-MEM genomische mapping-algoritme is een belangrijke eerste stap van veel genomics-pipelines, aangezien het de onbewerkte inputsequenties op een referentiegenoom plaatst, om zodoende de originele genetische assemblage van het testsubject te reconstrueren. Een groot deel van de totale BWA-MEM-uitvoeringstijd wordt besteed aan het uitvoeren van Seed Extension, een algoritme dat nauw verwant is aan het Smith-Waterman-algoritme voor paarsgewijze sequentie-uitlijning. De standaardbenadering voor de heterogene versnelling van het Smith-Waterman-algoritme is om een systolische arrayarchitectuur te gebruiken, waarbij elementen van de similariteitsmatrix in parallel worden berekend. Systolische arrays kunnen uitsluitend op hoge efficiëntie werken wanneer lange sequenties op elkaar worden uitgelijnd. Het BWA-MEM-algoritme daarentegen genereert doorgaans zeer korte sequenties ter uitlijning via het Seed Extension-algoritme. Daarom worden in dit proefschrift verschillende technieken voorgesteld om de efficiëntie van systolische arrays voor korte sequentielengtes te verbeteren.

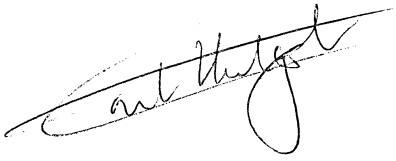
De Variabele Logische Lengte, de Variabele Fysieke Lengte en de Variabele Logische en Fysieke Lengte systolische array-architecturen worden voorgesteld om de afhankelijkheid van systolische array-efficiëntie op invoersequentielengte te elimineren. Om de afhankelijkheid van de lengte van de referentiesequentie te elimineren, wordt een streaming, auto-synchronizerende architectuur geïntroduceerd. Samen resulteren deze technieken in een systolische array met maximale efficiënte. Een Seed Extension-kernel is geïmplementeerd op zowel FPGA als GPU met een drievoudige snelheidsverbetering op kernelniveau, resulterend in de eerste FPGA-versnelde en de eerste GPU-versnelde implementatie van BWA-MEM met in totaal een tweevoudige snelheidsverbetering op applicatieniveau. Bovendien is een FPGA Smith-Waterman-implementatie ontwikkeld met behulp van bovenstaande efficiëntieverbeteringen van de systolische arrayarchitectuur, resulterend in een implementatie met een prestatie van 214 GCUPS en een efficiëntie van 99,8%, de hoogste gerapporteerde efficiëntie en prestatie van enige FPGA-versnelde Smith-Waterman-implementatie tot nu toe. Tenslotte worden verschillende aspecten van deze ontwerpen geëvalueerd, waaronder energie-efficiëntie en ontwerptijd.

Preface

EVER since finishing my Master's degree, I have entertained the thought of pursuing a PhD. However, by the time I had finally obtained my Master's diploma, I had been studying at Delft University of Technology for the better part of a decade, so it seemed prudent to leave academia, if at least for a while. After some years though, my engineering roots started itching again, and I am very grateful for my MSc thesis supervisor and friend Georgi Gaydadjiev for eventually pursuading me to start a PhD.

Furthermore, I would like to thank my promotor prof. Koen Bertels for giving me the opportunity to work for Bluebee while simultaneously being able to do my PhD, and my promotor Dr. Zaid Al-Ars for all his time, guidance and supervision. I really enjoyed our talks together and your door was always open for me. Although I did not spend much time at the university, I would like to thank my colleagues at Computer Engineering for the nice time I had there. In particular, I would like to thank Nauman Ahmed, since every time I went there, he was there available for a chat. I would also like to thank all my colleagues at Bluebee for the very pleasant working environment, and for putting up with me whenever I had another Graduate School course, whenver I had to write a paper, or whenever I had a conference to attend. In particular, I would like to thank Vlad-Mihai Sima for all his time, ideas, and for the discussions we had, and Giacomo Marchiori for implementing all our ideas and designs in actual FPGA hardware.

I would like to thank my friends and family for supporting me through all the time that I was preoccupied with my work and my PhD. Finally, I would like to thank Farnaz for encouraging me and for always believing in me. Thanks again to all of you.



Ernst Joachim Houtgast
Delft, February 2019

Contents

CHAPTER 1

Introduction

"The dinosaurs became extinct because they did not have a space program."

— Larry Niven

REVOLUTIONIZING biology in the process, one of the great scientific breakthroughs in modern history has been the discovery of the structure of DNA by Watson and Crick in the 1950s [1]. However, only recently improvements in sequencing techniques have made DNA sequencing affordable enough to start affecting everyday life. The increased adoption rate of sequencing techniques, combined with the greatly improved capabilities of DNA sequencers have resulted in an exponential increase in the amount of genomics data being generated [2]. Slowly but surely, the bottleneck of sequencing techniques is shifting away from the generation of raw data towards processing it into a useable form, by means of suites of computational bioinformatics tools called genomics pipelines. Therefore, acceleration and optimization of such genomics pipelines is becoming increasingly relevant. This dissertation focuses on improvements to one particular technique used in bioinformatics, namely on improvements to systolic arrays. These improvements are then applied to the BWA-MEM and the Smith-Waterman algorithm.

The remainder of this chapter is organized as follows. In Section 1.1, the motivation for this dissertation is discussed. In Section 1.2, a brief background on the two bioinformatics algorithms mentioned above is given and related work is presented. In Section 1.3, the main contributions of this work are listed. The chapter is concluded with Section 1.4, which explains the organization of the remainder of this dissertation.

1.1. MOTIVATION

The first time a single human genome was sequenced was less than two decades ago, in 2003, as part of the international Human Genome Project [3], a project that took fifteen years and cost about $2.7 billion. Not even twenty years later, sequencing technology has advanced so dramatically that it has become possible to sequence large groups of people at a time, enabling numerous population studies around the world, such as the Precision Medicine Initiative in the US [4], which aims to sequence at least a million

1

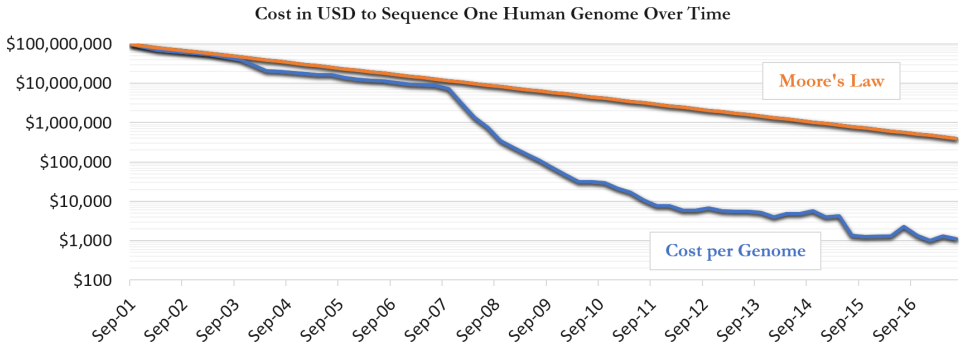


Figure 1.1: The cost of sequencing a single human genome has fallen dramatically over the last twenty years. Nowadays, a single genome can be sequenced for little more than $1,000, a drop of five orders of magnitude. This is even significantly faster than the rate at which Moore's law has revolutionized computing, as the extrapolated line shows. Data taken from [6].

genomes, and the 100,000 Genomes Project in the UK [5], which has even already completed sequencing its one hundred thousand genomes. Amongst others, these studies are expected to result in new treatment options and drugs that are designed to match a patient's unique genetic profile, resulting in improved treatment success rates along with fewer side-effects, in more accurate diagnoses, and in more effective prevention of disease. In stark contrast to the fifteen years it took to sequence the first human genome, a single state-of-the-art Illumina HiSeq X sequencer is able to sequence more than a dozen human genomes every three days, in the process reducing the cost of sequencing a single human genome to around $1,000. This dramatic fall in sequencing cost is best illustrated through a graph (see Figure 1.1). Compared to the well-known Moore's law, which stands at the foundation of revolutionizing the field of computing, the fall in sequencing cost is even more dramatic.

The rapid reduction in sequencing cost has resulted in widespread adoption of sequencing data, to be used in a variety of applications: apart from the uses in healthcare mentioned earlier, DNA sequencing is also used in biotechnology, forensics, virology and many other domains. The result is an explosion in the amount of data being generated, and sequencing data generation is soon expected to rival other big data fields, such as astronomy and online video streaming services [2]. The bottleneck when using sequencing data is slowly shifting away from data generation towards the computational techniques used to process it into a useful, actionable form. For example, a typical WGS cancer data set requires more than three thousand CPU-core hours to process. Hence, a clear need exists for faster, more efficient algorithms to process this deluge of data.

Fortunately, computer technology also advanced at an exponential pace. Although Moore's law seemingly slowed down as of late, on-chip transistor densities still increase and overall cost per transistor still falls. However, a main challenge in recent years has been to put the available transistor budget to good use. Due to limitations in power dissipation, spending the entire budget on a single, high performance monolithic core is infeasible. Instead, a trend can be observed towards multicore and manycore designs, and towards heterogeneous designs consisting of numerous small, highly specialized, highly efficient cores. Such specialized cores can offer huge advantages in both com-

putational power and in power-efficiency. An example is the System-on-a-Chip (SoC) used in any modern cell phone, which might contain not only multiple clusters of low and high power general purpose processors, but also multiple Graphical Processing Unit (GPU) cores, one or more neural network acceleration cores, an image signal processor for accelerating image processing tasks, vector co-processors, large caches, blocks facilitating I/O and memory interface, cores that implement the various wireless communication protocols, and more. Heterogeneous computing is a key technological enabler for increased computational power whilst at the same time improving overall efficiency. Thus, in this dissertation, which is positioned at the intersection of genomics and high performance computing, heterogeneously-accelerated solutions are investigated and applied to the computational algorithms used in the bioinformatics domain.

1.2. BACKGROUND AND RELATED WORK

In this work, heterogeneous architectures are used in order to accelerate the kernels of widely-used bioinformatics algorithms, mapping their most computationally expensive parts onto specialized hardware to obtain significant gains to performance and efficiency. In particular, Field-Programmable Gate Arrays (FPGAs) and Graphical Processing Units (GPUs) are used. A GPU is a specialized chip containing thousands of simple processing elements that excels at computing highly parallel tasks. Originally developed to accelerate the rendering of 2D and 3D images, where each pixel can be computed independently of the others, nowadays they have also become a popular tool for the acceleration of many high performance computing tasks that exhibit abundant parallelism, and are especially well-suited for neural networks. An FPGA, on the other hand, can be compared to a blank canvas, consisting of a large number of look-up-tables (LUTs), flexible routing boxes and memory elements that can be reprogrammed and connected to perform any kind of operation, and thus offering much flexibility to the programmer. Typical FPGAs these days also contain "hardened" elements, such as complete general purpose processors and floating point units. Similar to GPUs, FPGAs can offer huge computational power and very low latency, but also excel at power-efficiency since they can be programmed to exactly implement a certain functionality with little overhead.

The remainder of this section explains two bioinformatics algorithms in more detail, along with related work aimed to accelerate them on heterogeneous hardware.

1.2.1. BWA-MEM GENOMIC MAPPING ALGORITHM

The human genome is contained into long strands of DNA, and encoded in a double-helical sequence of symbols using an alphabet of nucleotides: Adenine (A), Cytosine (C), Guanine (G), and Thymine (T). In total, the human genome contains about three billion base pairs. Unfortunately, most sequencers are unable to read these long DNA strands completely in one long stream. Instead, the raw sequencing data typically consists of short read sequences of perhaps a hundred to one-hundred-and-fifty base pairs. However, these sequencers read each piece of DNA many times over, the so-called coverage. A coverage of 30x is typical, resulting in a raw output of about ninety billion base pairs for one human genome. The job of a genomics pipeline is then to transform this raw sequencing data into useable information.

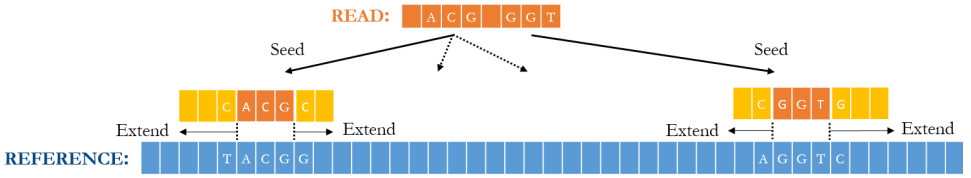


Figure 1.2: The BWA-MEM genomic mapping algorithm maps read sequences onto a reference genome using the Seed-and-Extend paradigm: for each read, Seeds, exactly matching subsequences between the read and the reference, are found through a process called Seed Generation. Then, these Seeds are extended in both directions using a Seed Extension algorithm, which is similar to the Smith-Waterman algorithm and allows for inexact matches. From all of the extended seeds, the alignment with the highest score is selected as output. This dissertation is focused on techniques directly applicable to the Seed Extension kernel of BWA-MEM.

One of the first steps in any genomics pipeline is to reconstruct the original genome from this huge number of short reads. There are two main techniques to accomplish this: de novo assembly and reference-based mapping algorithms. De novo assembly attempts to recreate the original genome from the raw data without using any external information. This can be compared to attempting to put together a puzzle without having an overview image available as help. Unfortunately, in this case where only short reads are available, the de novo assembling technique is impossible to use due to the nature of the genome, as it contains very long repeating sections that cannot be accurately determined without long read sequences to act as a bridge for these regions. Moreover, de novo sequencing is extremely computationally expensive. Therefore, reference-based mapping is the genome assembly technique that is more widely-used in practice. In keeping with the puzzle analogy, reference-based sequencing is similar to putting together a puzzle when one does have a reference image available, that can be used to make one or more initial guesses for the location where a piece might best fit. BWA-MEM then is one of the most widely used reference-based mapping tools [7].

BWA-MEM is one of the more popular reference-based mapping tools, as it is a part of the Broad Best Practices tool chain [8], a framework that gives guidelines on how to best process data from the initial raw data to the final variant call output. However, many other such mapping applications exist, such as Bowtie2 [9], FreeBayes [10], and many more. One characteristic that all contemporary reference-based mapping tools share, is that they use a heuristic called the Seed-and-Extend paradigm to reduce the complexity of the problem of mapping each read sequence onto the reference (see Figure 1.2). In the Seed-and-Extend paradigm, the first step is to determine probable locations for each read sequence, typically using a BWT-based lookup table. This technique places a limitation on seeds, as the BWT-lookup can only find exactly matching sequences between the read and the reference. After one or more seed locations are determined, these seeds are each extended using a Seed Extension algorithm that does allow for differences in the read and reference sequence to determine the overall alignment. Such heuristics are required, as the computational complexity makes it infeasible to use optimal algorithms on the entire reference sequence. From all these extended seeds, the highest scoring one is chosen as the final alignment. In the case of BWA-MEM, the Seed Extension algorithm is based on the Smith-Waterman pairwise sequence alignment algorithm.

1.2.2. SMITH-WATERMAN PAIRWISE SEQUENCE ALIGNMENT ALGORITHM

The Smith-Waterman algorithm, used in many bioinformatics tools, is able to compute the optimal local alignment between two arbitrary sequences [11]. Given a function that assigns scores to matching symbols, mismatching symbols, and to insertions and deletions, it is defined recursively to find the optimal solution. Local alignment in this case refers to the fact that not all symbols of each sequence need to be used in the alignment, if inclusion of additional symbols would yield a lowered score. Contrast this with the highly similar Needleman-Wunsch algorithm [12] for global pairwise sequence alignment, which requires all the symbols of both sequences to be used. The Smith-Waterman algorithm can be expressed as follows:

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + s(a_i, b_j) & : \text{(mis)match} \\ H_{i-1,j} - \textit{gap penalty} & : \text{insertion/deletion} \\ H_{i,j-1} - \textit{gap penalty} & : \text{insertion/deletion} \\ 0 & : \text{local alignment} \end{cases}$$

The collection of all values $H_{i,j}$ is called the similarity matrix, the dimensions of which are determined by the length of the two sequences to be aligned together. The highest value in this matrix denotes the location of the optimal alignment between the two sequences. A traceback procedure can then be followed to track back how this alignment is being generated. Since any value H is only dependent on its left, top-left, and top neighbor, the entire similarity matrix can be computed using a dynamic programming approach, using the fact that anti-diagonals of this matrix are independent of one another and can therefore be computed in parallel.

The Seed Extension algorithm used in BWA-MEM is similar to the Smith-Waterman algorithm, but differs in a few key aspects. Whereas the initial values of the Smith-Waterman algorithm are all set to zero, the Seed Extension algorithm uses the length of the Seed that is to be extended as an initial score. Moreover, the Seed Extension algorithm calculates a few more statistics, besides only the maximum score. For example, it also computes the best global alignment, where all symbols of both sequences are included in the alignment. Finally, the Seed Extension includes a few heuristics aimed at restricting the search in the similarity matrix to only those regions that are expected to contribute to the final score, greatly reducing the amount of computation required.

1.2.3. SYSTOLIC ARRAYS

A typical approach to accelerate the Smith-Waterman algorithm on heterogeneous systems is to utilize the fact that the anti-diagonals in the similarity matrix are independent and can therefore be computed in parallel. Instead of computing the similarity matrix one value at a time, one cell after another, which requires $O(N \times M)$ steps, the entire matrix can be computed in a number of steps equal to the number of anti-diagonals in the matrix, with the number of parallel computations increasing until the full width of the matrix is reached, and then decreasing again until the entire matrix has been computed, greatly reducing the number of steps required to just $O(N + M)$. Such an approach maps naturally onto an architecture called the systolic array. A systolic array consists of a chain of Processing Elements (PEs), in which the output of one PE is used as input to the next.

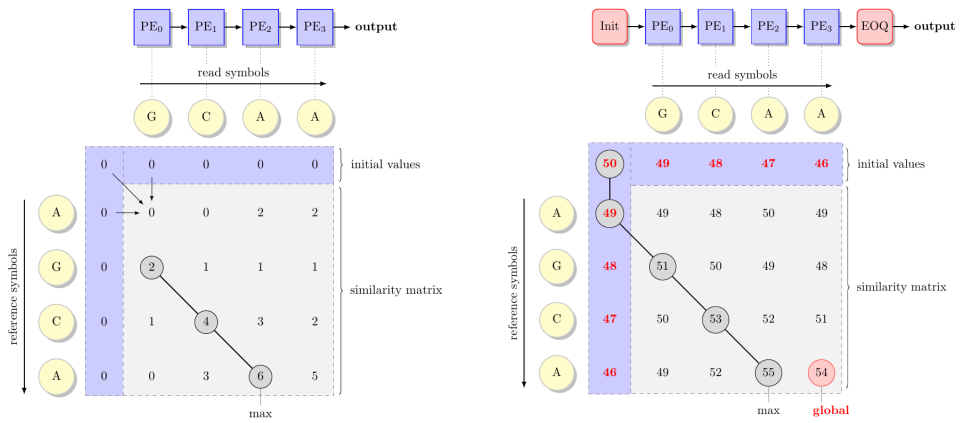


Figure 1.3: The Smith-Waterman algorithm is typically computed using a dynamic programming strategy by the filling of a similarity matrix (left inset). Each cell in the matrix is only dependent on its left, top-left, and upper neighbor, allowing antidiagonals to be processed in parallel. Read symbols are mapped onto PEs. The Seed Extension algorithm (right inset) differs in a few key aspects: non-zero initial values are used, and more values are calculated, including a global maximum.

This is illustrated in Figure 1.3, showing how the similarity matrix is mapped onto the PEs. In this organization, each read symbol is mapped onto a PE, which in effect makes this PE responsible to compute the respective column of the similarity matrix.

Numerous heterogeneously-accelerated versions of the Smith-Waterman algorithm exist that use the systolic array as an underlying architecture, including for example the following FPGA-accelerated Smith-Waterman implementations [13], [14] and [15], and the GPU-accelerated implementation in [16]. A characteristic that all these implementations share, is that the speedup they offer increases with larger similarity matrix dimensions, since the time complexity difference between $O(N \times M)$ and $O(N + M)$ becomes much more pronounced for larger values of N and M. Since the matrix dimensions are directly dependent on the respective sequence's lengths, in effect, this means that performance is better the longer the two sequences to be aligned together are. Conversely, pairwise alignments between shorter sequences may not see much, if any, benefit from such an approach, especially when taking overhead into account.

As mentioned in the previous section, the BWA-MEM Seed Extension algorithm is closely related to the Smith-Waterman algorithm. In Figure 1.3, the Seed Extension similarity matrix is shown for a seed with an initial value of fifty. It is clear that the overall idea is very similar between both algorithms. The differences mainly reside in the fact that the Seed Extension similarity matrix contains non-zero initial values, and that it also computes a global maximum value. One intrinsic characteristic of BWA-MEM is that the length of the sequences to be extended is closely related to the length of the raw input data: for each sequence in the input one or more seeds are found, and, subtracting the seed sequence from the input sequence, both sides are then extended. This means that in practice, the sequences to be extended vary between just a single symbol in length, to up to one-hundred-and-thirty-one symbols. The reference sequence is typically chosen

to be a bit larger than the read sequence to be extended. Therefore, this makes it convenient to map the read symbols onto the PEs of the systolic array, since the array needs to include at least as many PEs as there are symbols, and an upper bound exists on the length of the read sequence. The reference sequence is then fed one-by-one as input to the first PE in the systolic array.

The above highlights the main limitation of systolic array architectures, and in particular of the application of such architectures to accelerate the BWA-MEM Seed Extension algorithm. On the one hand, a systolic array implementation provides a larger speedup the longer the sequences to be aligned are. However, on the other hand, BWA-MEM generates quite short sequences that require to be extended. Therefore, the main objective in this dissertation is to focus on improvements to the systolic array architecture to make them operate efficiently even when applied to short read sequences.

1.3. MAIN CONTRIBUTIONS

In this dissertation, the following contributions are made:

- Various systolic array architectures are proposed to eliminate the dependence of the systolic array's efficiency on the length of the read sequence. Since the symbols of the read sequence are mapped onto the PEs, the length of the read sequence must closely match the number of PEs of the systolic array, otherwise its efficiency will suffer. The impact that these techniques have is evaluated both on relative performance, as well as on the required area.
- Systolic array-based Seed Extension kernels are designed for the FPGA using VHDL and for the GPU using CUDA, obtaining the first FPGA-accelerated and the first GPU-accelerated implementation of BWA-MEM, respectively.
- Two techniques, streamed loading of sequences and implicit synchronization, are introduced that eliminate the dependence of the systolic array's efficiency on the length of the reference sequence. Normally, a systolic array process just a single pairwise sequence alignment at a time. The symbols of the reference sequence are fed one-by-one as input to the systolic array and subsequently flow through all the PEs in the array in order to compute the output. Therefore, feeding short reference sequences as input results in very low efficiency, since most time is spent waiting while this short sequence passes through the array. By allowing any number of pairwise sequence alignments to be performed simultaneously, without requiring to wait for one alignment to finish before starting the next alignment, this source of inefficiency is completely eliminated.
- A Smith-Waterman implementation including both the above techniques is created using OpenCL to create a maximally-efficient systolic array implementation that achieves the highest efficiency and the highest performance to date of any FPGA Smith-Waterman implementation.
- The characteristics of all these implementations are analyzed across a number of other relevant dimensions, namely: power-efficiency, design-time, and the impact of read length on performance.

1.4. THESIS ORGANIZATION

The remainder of this dissertation is organized as follows:

- In **Chapter 2**, a number of systolic array architecture variants are proposed that are designed to eliminate one source of inefficiency of systolic arrays, the dependence of the systolic array on read sequence length. The initial work on the GPU-accelerated and FPGA-accelerated BWA-MEM implementations is discussed.
- In **Chapter 3**, further refinements are proposed for the FPGA and GPU BWA-MEM implementations, resulting in versions that are able to completely hide the Seed Extension kernel execution from overall program execution time, resulting in a two-fold overall application-level speedup. Moreover, improvements to the systolic array architecture are made that eliminate the other source of inefficiency in systolic array operation, its dependence on reference sequence length. The result is an architecture that is maximally-efficient.
- In **Chapter 4**, other key design metrics are considered, such as power-efficiency and the required design-time to create and make changes to heterogeneously-accelerated solutions. Furthermore, the impact of read length on performance is analyzed, as this is an important external requirement from the bioinformatics domain that is expected to increase in the near future.
- In **Chapter 5**, the conclusions and potential avenues for future work are presented.

REFERENCES

[1] J. D. Watson, F. H. Crick, *et al., Molecular Structure of Nucleic Acids,* Nature **171**, 737 (1953).

[2] Z. Stephens, S. Lee, F. Faghri, R. Campbell, C. Zhai, M. Efron, R. Iyer, M. Schatz, S. Sinha, and G. Robinson, *Big Data: Astronomical or Genomical?* PLoS Biology **13** (2015).

[3] International Human Genome Sequencing Consortium, and others, *Initial Sequencing and Analysis of the Human Genome,* Nature **409**, 860 (2001).

[4] National Institutes of Health, *All of Us Research Program,* `https://allofus.nih.gov/`, last visited: 2019-01-08.

[5] Genomics England, *100,000 Genomes Project,* `https://www.genomicsengland.co.uk/`, last visited: 2019-01-08.

[6] K. Wetterstrand, *DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP),* `www.genome.gov/sequencingcostsdata/`, last visited: 2019-01-08.

[7] H. Li, J. Ruan, and R. Durbin, *Mapping Short DNA Sequencing Reads and Calling Variants Using Mapping Quality Scores,* Genome research **18**, 1851 (2008).

[8] M. A. DePristo, E. Banks, R. Poplin, K. V. Garimella, J. R. Maguire, C. Hartl, A. A. Philippakis, G. Del Angel, M. A. Rivas, M. Hanna, *et al.*, *A Framework for Variation Discovery and Genotyping Using Next-Generation DNA Sequencing Data,* Nature genetics **43**, 491 (2011).

[9] B. Langmead and S. L. Salzberg, *Fast Gapped-Read Alignment with Bowtie 2,* Nature methods **9**, 357 (2012).

[10] E. Garrison and G. Marth, *Haplotype-Based Variant Detection From Short-Read Sequencing,* arXiv preprint arXiv:1207.3907 (2012).

[11] T. Smith and M. Waterman, *Identification of Common Molecular Subsequences,* Journal of Molecular Biology **147**, 195 (1981).

[12] S. B. Needleman and C. D. Wunsch, *A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins,* Journal of molecular biology **48**, 443 (1970).

[13] T. Oliver, B. Schmidt, and D. Maskell, *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* (ACM, 2005) pp. 229–237.

[14] P. Zhang, G. Tan, and G. R. Gao, *Implementation of the Smith-Waterman Algorithm on a Reconfigurable Supercomputing Platform,* in *Proceedings of the 1st international workshop on High-performance reconfi- gurable computing technology and applications: held in conjunction with SC07* (ACM, 2007) pp. 39–48.

[15] C. W. Yu, K. Kwong, K.-H. Lee, and P. H. W. Leong, *A Smith-Waterman Systolic Cell,* in *New Algorithms, Architectures and Applications for Reconfigurable Computing* (Springer, 2005) pp. 291–300.

[16] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig, *Bio-sequence database scanning on a GPU,* in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International* (IEEE, 2006) pp. 8–pp.

1

CHAPTER 2

Systolic Array Architectures and the Seed Extension Kernel

"There is nothing permanent except change."

— Heraclitus

NOVEL variants to the systolic array architecture, the main focus of this dissertation, are introduced in this chapter, along with an analysis of their relative performance and area requirements. These optimized systolic arrays are then used to accelerate the Seed Extension kernel of BWA-MEM, a widely used genomic mapping algorithm. Seed Extension is one of the main two kernels in which the BWA-MEM algorithm spends the majority of its execution time. The Seed Extension algorithm is closely related to the Smith-Waterman algorithm, which computes the optimum alignment between two sequences for any given scoring function. The typical approach for hardware-based implementations of the Smith-Waterman algorithm is to use dynamic programming to compute a similarity matrix, through the use of a systolic array. A systolic array consists of a chain of Processing Elements (PEs), where PE_N receives its input from PE_{N-1}, and passes its output to PE_{N+1}. The systolic array architecture is not only useful for dynamic programming tasks, but can address a wide variety of problems, including linear algebra computations and matrix multiplication. In the bioinformatics domain, algorithms that have been accelerated using a systolic array implementation include the Smith-Waterman algorithm for pairwise sequence alignment, BWA-MEM and BLAST for read mapping, and the HaplotypeCaller for variant calling.

To obtain the best possible alignment between two sequences, the Smith-Waterman algorithm computes a similarity matrix, in which the highest value denotes the optimal solution. The dimensions of this similarity matrix are determined solely by the lengths of the two sequences to be aligned. Typically, one sequence is called the reference sequence and the other the read sequence. In the systolic array approach, this read sequence is mapped onto the PEs of the systolic array. The symbols of the reference sequence are fed one-by-one as input to the first PE, and subsequently pass through all PEs of the array to compute the desired function. Systolic arrays have a number of advantages, most notably the fact that values are computed and stored inside the PEs, eliminating the need for external memory and buses, and the fact that all PEs operate independently of one another, allowing for a high degree of parallelism.

A main challenge for any systolic array is to maintain high overall efficiency. Input is fed into the first PE and then flows through the array until it reaches the last PE and produces a result. Keeping all PEs busy is critical to achieve high utilization. However, this is not trivial. The length of the reference and read sequences greatly impacts overall efficiency: a mismatch between read sequence length and systolic array length results in underutilization of the array, as not all PEs are required to compute the similarity matrix. Short reference sequences result in lots of idle cycles while the short input flows through the entire array. In this chapter, measures are proposed to remove the dependence on read sequence length. In the next chapter, techniques are proposed to eliminate the dependence on reference sequence length. Together, these result in an architecture that can operate at maximal efficiency for any input data set.

2.1. MAIN CONTRIBUTIONS

The main contributions of this chapter are as follows:

- The Variable Logical Length, Variable Physical Length, and Variable Logical and Physical Length systolic array design architectures are proposed and analyzed to evaluate their efficiency for processing an input data set with sequences of various lengths. Two measures are introduced to evaluate efficiency: Exit Point Optimality and the Configuration Optimality, respectively measuring how well an exit point configuration approximates the ideal situation where an exit point is available for each read length, and showing how well a configuration approximates optimal load balancing for a particular data set **[SAMOS2015]**.
- A VLL-based Seed Extension implementation offering a threefold improvement to performance is created on FPGA using a combination of VHDL (for the Seed Extension kernel) and HLS (for the integration code), resulting in the first accelerated BWA-MEM implementation **[SAMOS2015]**.
- A GPU-based Seed Extension kernel is created using CUDA, offering a threefold improvement to performance. GPUs have a fixed underlying architecture, on which the systolic array needs to be mapped. The implementation uses various kernels to optimize resource requirements: a single-pass kernel that maps the full width of the systolic array across multiple warps, two multi-pass kernels optimized for shorter and longer reference sequence length, and a single-pass kernel. This results in the first GPU-accelerated implementation of BWA-MEM **[ARCS2016]**.

2.2. RESEARCH ARTICLES

This chapter is based on the following papers:

1. **E.J. Houtgast**, V.M. Sima, K.L.M. Bertels, and Z. Al-Ars, *An FPGA-Based Systolic Array to Accelerate the BWA-MEM Genomic Mapping Algorithm*, 15th International Conference on Embedded Computer Systems (SAMOS), Jul 2015, Samos, Greece.
2. **E.J. Houtgast**, V.M. Sima, K.L.M. Bertels, and Z. Al-Ars, *GPU-Accelerated BWA-MEM Genomic Mapping Algorithm Using Adaptive Load Balancing*, 29th International Conference on Architecture of Computing Systems (ARCS), Apr 2016, Nuremberg, Germany.

An FPGA-Based Systolic Array to Accelerate the BWA-MEM Genomic Mapping Algorithm

Ernst Joachim Houtgast, Vlad-Mihai Sima, Koen Bertels and Zaid Al-Ars
Faculty of EEMCS, Delft University of Technology, Delft, The Netherlands
E-mail: {e.j.houtgast, v.m.sima, k.l.m.bertels, z.al-ars}@tudelft.nl

***Abstract*—We present the first accelerated implementation of BWA-MEM, a popular genome sequence alignment algorithm widely used in next generation sequencing genomics pipelines. The Smith-Waterman-like sequence alignment kernel requires a significant portion of overall execution time. We propose and evaluate a number of FPGA-based systolic array architectures, presenting optimizations generally applicable to variable length Smith-Waterman execution. Our kernel implementation is up to 3x faster, compared to software-only execution. This translates into an overall application speedup of up to 45%, which is 96% of the theoretically maximum achievable speedup when accelerating only this kernel.**

I. INTRODUCTION

As next generation sequencing techniques improve, the resulting genomic data, which can be in the order of tens of gigabytes, requires increasingly long time to process. This is becoming a large bottleneck, for example in cancer diagnostics. Hence, acceleration of algorithms used in genomics pipelines is of prime importance. General purpose processors are not necessarily the best execution platform for such workloads, as many bioinformatics workloads lend themselves well to parallel execution. Use of dedicated hardware, such as GPUs or FPGAs, can then greatly accelerate the computationally intensive kernels to achieve large speedups.

The initial stage for many genomics pipelines is sequence alignment. DNA sequence reads are aligned against a reference genome, producing the best found alignment for each read. Many sequence alignment tools exist, such as Bowtie [5], BWA [8], MAQ [9], and SOAP2 [10]. BWA-MEM [7] is widely used in practice as a de facto sequence alignment algorithm of choice. In this paper, we investigate and propose the first accelerated version of BWA-MEM, using FPGAs to improve performance. The use of FPGAs can yield order-of-magnitude improvements in both processing speed and power consumption, as they can be programmed to include a huge number of execution units that are custom-tailored to the problem at hand, providing much higher throughput than conventional processors. At the same time, they consume much less power, since they operate at relatively low clock frequencies and use less silicon area for the same task.

In this paper, we present the following contributions: we (1) analyze the BWA-MEM algorithm's main execution kernels; (2) propose novel systolic array design approaches optimized for variable length Smith-Waterman execution; (3) implement and integrate the design into the BWA-MEM algorithm; and thus (4) create the first accelerated version of the BWA-MEM algorithm, using FPGAs to offload execution of one kernel.

The rest of this paper is organized as follows. Section II provides a brief background on the BWA-MEM algorithm. Section III describes the details of our acceleration approach. Section IV discusses design alternatives for the systolic array implementation. Section V provides the details on the configuration used to obtain results, which are presented in Section VI and then discussed in Section VII. Section VIII concludes the paper and indicates directions for future work.

II. BWA-MEM ALGORITHM

The BWA program is *"a software package for mapping low-divergent sequences against a large reference genome, such as the human genome. It consists of three algorithms: BWA-backtrack, BWA-SW and BWA-MEM ... BWA-MEM, which is the latest, is generally recommended for high-quality queries as it is faster and more accurate."* [6] A characteristic workload of this algorithm is to align millions of DNA reads against a reference genome. Currently, we align DNA reads of 150 base pairs (bp), a typical output length of next generation sequencers [1], against the human genome.

A. BWA-MEM Algorithm Kernels

The BWA-MEM algorithm alignment procedure consists of three main kernels, which are executed in succession for each read in the input data set.

1. SMEM Generation: Find likely mapping locations, which are called *seeds*, on the reference genome. To this end, a BWT-based index of the reference genome, which has been generated beforehand, is used [8]. Per read, zero or more seeds are generated of varying length;

2. Seed Extension: Chain and extend seeds together using a dynamic programming method that is similar, but not identical, to the Smith-Waterman algorithm [12];

3. Output Generation: Sort and, if necessary, perform global alignment on the intermediate results and produce output in the standardized SAM-format.

The BWA-MEM algorithm processes reads in *batches*. Figure 1 illustrates the order in which these kernels process a batch of reads. The first two kernels, SMEM Generation and Seed Extension, are performed directly after each other for each read. When this is finished for all reads in a batch, Output Generation is performed. BWA-MEM implements multi-threaded execution of all three program kernels.

978-1-4673-7311-1/15/$31.00 ©2015 IEEE

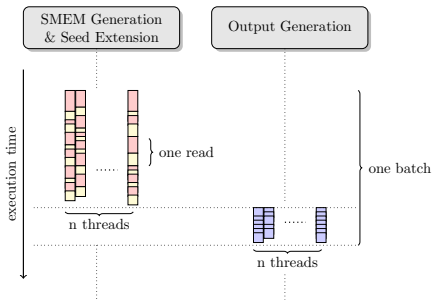


Fig. 1. Execution order of the three main BWA-MEM algorithm kernels. Per batch, execution of SMEM Generation and Seed Extension is intertwined for each read; afterwards, Output Generation is performed.

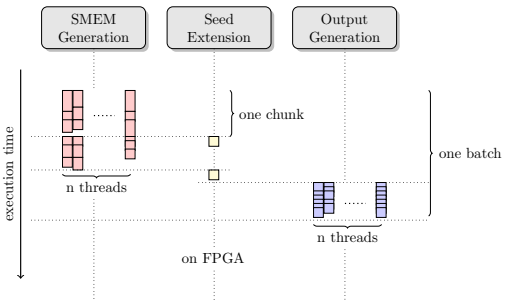


Fig. 2. Our implementation further subdivides batches into chunks. SMEM Generation and Seed Extension are separated and Seed Extension is mapped onto the FPGA; its execution is overlapped with SMEM Generation.

B. Profiling Results

A challenging factor in the acceleration of the BWA-MEM algorithm is the fact that execution time is not isolated in a single computationally-intensive kernel, but is spread over three separate kernels. Hence, speedup of a single kernel will only yield limited overall speedup. To investigate which of these kernels is most suitable for FPGA-based acceleration, we have analyzed and profiled the algorithm with `gprof` and `oprof`. Both yielded similar results. Table I shows the profiling results for a typical workload.[1]

TABLE I
RESULTS OF BWA-MEM ALGORITHM PROFILING

Program Kernel	Time	Bottleneck	Processing
SMEM Generation	56%	Memory	Parallel
Seed Extension	32%	Computation	Parallel
Output Generation	9%	Memory	Parallel
Other	3%	I/O	Sequential

For each kernel, the relative execution time, type of processing and whether it is bound by computation, memory or I/O is specified, based on a combination of profiling and source code inspection. Besides these computationally-intensive kernels, the remaining execution time is comprised of other activities, among them initialization and I/O.

In this paper, we investigate acceleration of the Seed Extension kernel. This work is part of an on-going effort to accelerate execution of the BWA-MEM algorithm. Our rationale to start with the Seed Extension kernel is as follows: although profiling results indicate that the SMEM Generation kernel is more time-consuming, the dynamic programming-type of algorithm used in the Seed Extension kernel is a much better fit for execution on an FPGA. As shown in Table I, Seed Extension requires 32% of overall execution time for this workload. Hence, the maximum speedup we can expect to gain from accelerating this part is 47%.

[1]Single-ended GCAT on the Convey HC-2^{EX} (refer to Section V for more details on the workload and execution platform).

III. IMPLEMENTATION

Our efforts to accelerate the Seed Extension kernel can be divided into two parts: (1) the FPGA-accelerated core that implements the Seed Extension kernel; and (2), integration of this kernel into the BWA-MEM algorithm.

A. Top-Level Accelerated Structure

As shown in Section II-B, the BWA-MEM algorithm consists of three distinct kernels. As illustrated in Figure 1, execution of the SMEM Generation kernel and Seed Extension is intertwined. Directly using this structure to offload execution of the Seed Extension kernel onto an FPGA would require a large number of small data transfers, two per read. The resulting overhead makes such a structure undesirable.[2] Hence, in order to facilitate offloading this kernel onto the FPGA, SMEM Generation and Seed Extension are completely separated from each other, which allows for fewer, but larger transfers of data to and from the FPGA. The modified structure of operation is shown in Figure 2. This approach does require that some temporary data structures are kept in memory longer. In practice, this is not an issue as the data is in the order of tens of megabytes.

The accelerated approach is based on two principles: (1) offloading the Seed Extension kernel onto the FPGA; and (2) overlapping execution of SMEM Generation on the host CPU and Seed Extension on the FPGA, thereby effectively hiding the time required to process this stage. In order to overlap these kernels, the reads in a batch are further subdivided into *chunks*. After a chunk is processed by the SMEM Generation kernel, it is dispatched to the FPGA. Output Generation is performed only after both the kernels finish, as the CPU cores are fully utilized while performing SMEM Generation. Hence, there would be no benefit in overlapping execution of this kernel with the other two.

Reads vary in the amount of temporary data required to process them: some reads generate more SMEMs (i.e., potential

[2]For example, testing reveals that copying 1 Mbyte at once is almost 30x faster than performing a thousand 1 kbyte transfers.

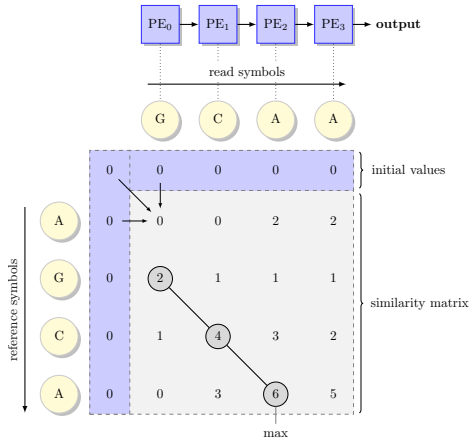


Fig. 3. Smith-Waterman similarity matrix showing the local sequence alignment with maximum score. Each read symbol is mapped onto a Processing Element of the systolic array.

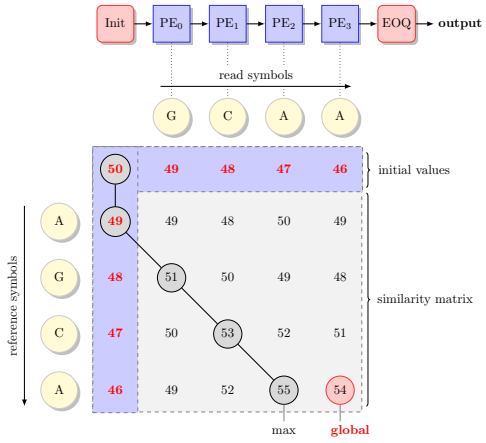


Fig. 4. Seed Extension similarity matrix showing an extension with an initial score of 50. The implications to the systolic array design are highlighted: additional Initialization and End-of-Query blocks; non-zero initial values; and calculation of the global maximum alignment score.

alignment locations) than others, and all alignments need to be kept in memory to be able to select the best overall alignment. Hence, in order to limit the hardware resources required for some extreme cases, not all reads are handled on the FPGA. In practice, we process more than 99% of all alignments on the FPGA. The remaining reads are instead executed on the host, which does not suffer from fixed memory size limitations.

B. Seed Extension Kernel

This section provides more details on the particular function of the Seed Extension kernel. Seeds, as generated by the SMEM Generation kernel, are an exact match of symbols from the read onto the reference (or a subsequence of either). The purpose of Seed Extension is to extend the length of such an exact match while allowing for small differences: mismatches between the read and reference, or skipping symbols on either the read or reference. A typical example of an alignment is given below:

```
                    Seed   Extension
Reference    GCGC AAGCTA GCTGAGGCTAA
Read         ---- AAGCTA AC-GAGG----
```

The Smith-Waterman algorithm [12] is a well-known dynamic programming algorithm that is guaranteed to find the optimum alignment between two sequences for a given scoring system. A *similarity matrix* is filled that computes the best score out of all combinations of matches, mismatches and gaps. This is illustrated by Figure 3. The process by which the similarity matrix is filled contains much inherent parallelism, as each cell only depends on its top, top-left and left neighbor. This implies that all cells on the same anti-diagonal can be computed in parallel.

1) Linear Systolic Arrays: A natural way to map dynamic programming algorithms onto reconfigurable hardware is as a linear systolic array. Many implementations that map the Smith-Waterman algorithm onto a systolic array have been proposed, amongst others [11], [13] and [14]. A systolic array consists of Processing Elements, or PEs for short, that operate in parallel. In the case at hand, we use such an array to take advantage of the available parallelism that exists while filling the similarity matrix, by processing the cells on the anti-diagonal in parallel. As illustrated in Figure 3, we map one read symbol to one PE, which corresponds to one column of the matrix. Each cycle, a PE processes one cell of the matrix and passes the resulting values to the next element. The values typically passed along to calculate the similarity matrix are the current cell's score, the row maximum, the current reference symbol, and the current gap score.

Although systolic array implementations excel in extracting parallelism, they do possess a number of drawbacks. First, the length of the PE-array determines the maximum length of a read that can be processed: one PE is required per read symbol. In this work we consider reads of up to 150 base pairs in length. Hence, we can guarantee that all reads will fit onto an array of a corresponding size.[3] Second, reads shorter than the PE-array still need to travel through it, incurring unnecessary latency and wasting resources by underutilizing the array. Finally, the maximum degree of parallelism is only achieved when all PEs are kept busy, which by virtue of its pipelined organization cannot always be ensured. In Section IV, we show how to deal with these issues.

[3]In practice, as we only consider data with a read length of 150 and the minimum seed length is 19 symbols, an extension can span at most 131 characters. Thus, an array of length 131 suffices.

2) Differences with "Standard" Smith-Waterman: The Seed Extension kernel used in BWA-MEM is similar to the Smith-Waterman algorithm, but the fact that the purpose is not to find the optimal match between two sequences, but instead to extend a seed found beforehand gives rise to three key differences. These differences and the impact they have on the design of the systolic array implementation are discussed below and illustrated in Figure 4.

1. Non-Zero Initial Values: Since the purpose of the Seed Extension kernel is to extend a seed, the match between sequences will always start from the "origin" of the similarity matrix (i.e., the top-left corner). The initial values provided to the first column and row of the dynamic programming matrix are not zero, but depend on the alignment score of the seed found by the SMEM Generation kernel.

Implication: An initial value block is placed in front of the array and initial values are computed and passed from one PE to the next.

2. Additional Output Generation: The Seed Extension kernel not only generates local and global alignment scores, which are the highest score in the matrix and the highest score that spans the entire read respectively, but also returns the exact location inside the similarity matrix where these scores have been found. Furthermore, a *maximum offset* is calculated that indicates the distance from the diagonal at which a maximum score has been found.

Implication: The index of the PE where the maximum is obtained is passed from one PE to the next. An End-of-Query block, which generates the output values by post-processing the results, is inserted at the end of the array.

3. Partial Similarity Matrix Calculation: To optimize for execution speed, BWA-MEM uses a heuristic that attempts to only calculate those cells that are expected to contribute to the final score. Profiling reveals that, in practice, only about 42% of all cells are calculated.

Implication: Our implementation does not use this heuristic, as the systolic array is able to perform all calculations on the anti-diagonal in parallel, which potentially leads to higher quality alignments.

IV. DESIGN SPACE EXPLORATION

Before deciding upon the final design of the Seed Extension kernel, a number of ideas and design alternatives, or *PE-module configurations*, were considered, varying in speedup, FPGA-resource consumption, suitability for certain data sets, and complexity. These are depicted in Figure 5 and will be discussed below. For analysis purposes, a data set with uniformly distributed extension lengths was used. Inspection of a histogram with GCAT seed extension lengths shows that this assumption is reasonable. We also consider that we have the entire data set available at the start for optimal scheduling.

A. Variable Logical Length Systolic Array

The length over which Seed Extension is to be performed is not the entire read length, but shorter, ranging from a single symbol up to the entire read length minus minimum

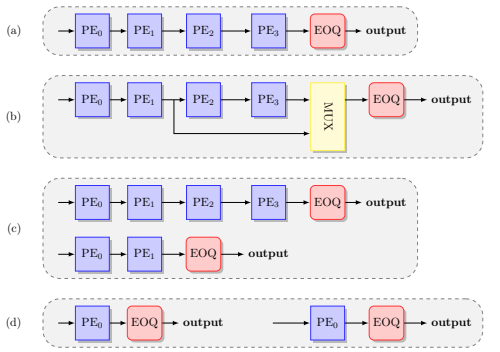


Fig. 5. PE-Module configurations: (a) standard systolic array configuration; (b) Variable Logical Length configuration that can bypass part of the array; (c) Variable Physical Length configuration that matches systolic array length to read length; (d) GPU-like single-PE modules.

seed length. Hence, the alignment that the kernel has to perform varies in its length. As mentioned in Section III-B1, a characteristic of systolic arrays is that processing time is independent of read length, as a read has to travel through the entire PE-array irrespective of its length: i.e., processing time is $O(|PEarray| + |Reference|)$, instead of $O(|Read| + |Reference|)$. Hence, shorter reads incur unnecessary latency and cause the systolic array to be underutilized.

To minimize latency, ideally a read would be processed by a PE-array matching its exact length. However, in practice, this is not achievable, since it would require having a PE-array for each possible read length, which is impractical given the available resources on the FPGA. Therefore, we propose to insert *multiple exit points* into the PE-array, as shown in Figure 5(b). We call this *Variable Logical Length* (VLL). This ensures that shorter reads do not have to travel through the entire array. Only a multiplexer and some control logic to select the correct exit point is needed, so this technique introduces minimal area overhead.

Definition 1 *The Exit Point Optimality measures how well an exit point configuration approximates the ideal situation of having a PE-array matching each read length.*

Of course, the Exit Point Optimality is data set dependent: for a set of reads that are all of the same length, a single module length suffices and the VLL technique can provide no benefit. In the case of a data set with uniformly distributed read lengths, it can be shown that minimal latency is achieved with evenly distributed exit points.

This idea can be further extended by subdividing the systolic array into two or more smaller logical systolic arrays that can operate in parallel. For example, a 150-PE array could present itself as two separate 75-PE arrays. This is similar to the approach suggested in [11]. However, that technique needs substantial additional hardware resources.

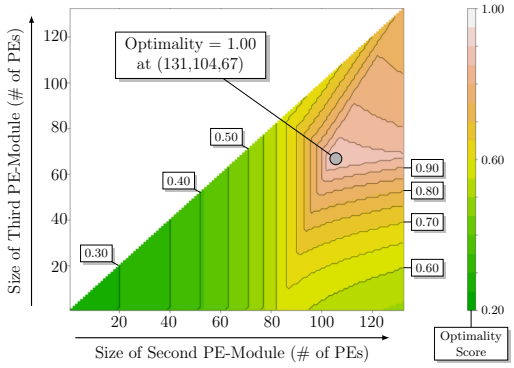


Fig. 6. Configuration Optimality for all three VPL PE-module combinations given a data set with uniformly distributed extension lengths. The indicated point shows the best configuration for the data set, which by definition has a value of 1. Data were obtained through exhaustive search of the design space.

B. Variable Physical Length Systolic Array

Another method to improve performance is using multiple systolic arrays together with *Variable Physical Length* (VPL), as shown in Figure 5(c). Longer reads are processed by the larger PE-arrays, while shorter reads are processed by the smaller arrays. Besides the above-mentioned improvement to execution speed, this has an additional benefit: the additional PE-arrays are physically smaller, which in turn allows for even more modules to be placed on the FPGA, improving speed even more. Combining this idea with the VLL-technique results in a *Variable Logical and Physical Length* (VLPL) array.

Definition 2 *The Configuration Optimality shows how closely a VPL-configuration achieves optimal load-balancing for a given data set, optimum CO being defined as 1.*

Figure 6 shows the Configuration Optimality for all combinations of three PE-modules, given a data set with uniformly distributed extension lengths. The graph shows the relative efficiency of all the configurations in the design space. To derive the relative efficiency, we have modeled the required time to process an entire data set, assuming optimal scheduling of reads onto modules. This is relatively easy given the fact that the processing time for each read is only dependent on the systolic array length of the modules and the length of the read to be processed. The optimal configuration is that configuration with the smallest module sizes (to minimize latency) for which all modules can be kept busy until the end.

Note that the optimal configuration of VLL- and VPL-arrays depends on the specific distribution of extension lengths of the data set at hand. In order to efficiently cope with various input data sets, multiple FPGA bitstreams can be compiled beforehand, each optimally configured for a different distribution. Then, an initial sampling of the input data can be used to select the best matching bitstream.

TABLE II
ESTIMATED RELATIVE PERFORMANCE OF SIMILAR AREA PE-MODULE CONFIGURATIONS GIVEN A DATA SET WITH UNIFORMLY DISTRIBUTED EXTENSION LENGTHS.

Design	PE-Module Configuration	Relative Speed	Relative Area
Standard	2x (131)	100%	100%
VLL	2x (131/87/43)[4]	125%	100%
VPL	1x (131), 1x (104), 1x (67)	181%	114%
VLPL	1x (131/122/113), 1x (104/92/79), 1x (67/45/22)	203%	114%

C. Single-PE Modules

The last alternative (see Figure 5(d)) is technically not a systolic array. One PE processes a similarity matrix entirely by itself. Parallelism is achieved not through intra-read parallelism, but instead by utilizing inter-read parallelism: processing multiple reads in parallel on a ”sea-of-cores”, similar to GPU-accelerated Smith-Waterman approaches such as the method discussed in [3]. Latency is traded for overall throughput.

An advantage of this approach is that it allows the use of the heuristic mentioned in Section III-B2-3, to only calculate relevant parts of the similarity matrix. Hence, as only about 42% of cells are processed, in theory a hundred single-PE modules should be 2.5 times faster than a one-hundred-PE module, not even considering the fact that the latter will often be underutilized. Drawbacks of this method are the considerable overhead this configuration suffers from: in PE-terms, control and other overhead are about equivalent to two PEs in logic cost. Moreover, whereas the systolic array designs implicitly store temporary data inside the array, the single-PE method requires explicit storage of temporary values. Finally, our current top level design can only fit up to six modules (of any kind), due to the resources required per core for input and output data structures.

D. Evaluation of PE-Module Configurations

Table II shows the relative performance of the various systolic array configurations. As we did not implement all the different configurations, we derived these estimations with the same approach as was used in Section IV-B to compute the Configuration Optimality. The configurations all have area requirements similar to a two 131-PE configuration. This will be the area on the FPGA we expect to be able to dedicate to seed extension logic in future implementations that also accelerate other parts of the algorithm on the FPGA. Given the extra resources the single PE configuration requires, we excluded this approach from the comparison.

The results show that the fastest approach is the VLPL configuration, being more than twice as fast as a standard systolic array implementation. The VLL and VPL configurations use different values for their exit points and module

[4]The Exit Point Optimality of this configuration is 0.87.

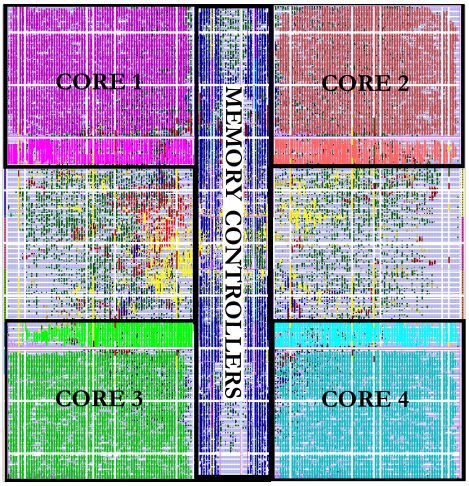


Fig. 7. Floorplan of the implemented FPGA design showing the four identical VLL-based modules and the memory controllers. The remainder of space is taken up by the Convey-to-host interface.

sizes, respectively, as their optimization goal is different: VLL optimizes for average latency, whereas VPL tries to balance execution time between modules. More exit points, or more modules would result in a higher speedup. The single-PE module configuration (not shown in the table) would be more than three times faster than a standard systolic array implementation, as (1) it does not suffer from underutilization of the array, and (2) it can take advantage of the similarity matrix heuristic (refer to Section III-B2-3).

V. METHODS

All tests were run on the Convey HC-2^{EX} platform [2], configured with two Intel Xeon E5-2643 processors (four cores each, HyperThreading disabled) running at 3.3 GHz with 64 GB of DDR3 memory, paired with four Xilinx Virtex-6 LX760 FPGA co-processors (speed grade 1) connected to 64GB of SG-DIMM memory. Each FPGA is programmed with four Seed Extension modules, for a total of sixteen modules across all FPGAs. Modules are VLL-based and contain 131 PEs each.

To accelerate the Seed Extension kernel, we implemented a VLL-based design with four identical modules per FPGA, along with other components, such as memory controllers and the Convey-to-host interface blocks. Figure 7 shows the floorplan of the implemented design. A single module uses about 16% of FPGA resources, while in total approximately 71% of all resources was used. Although with more effort we would be able to place six modules per FPGA, a design with four modules provided sufficient performance to completely hide execution of the Seed Extension kernel. Hence, to reduce planning and routing complexity, we did not attempt to completely fill up the entire FPGA.

Data sets from the Genome Comparison & Analytic Testing (GCAT) framework [4] were used to obtain results for single-ended alignment (150bp-se-small-indel) and pair-ended alignment (150bp-pe-small-indel) of about eight million reads against the reference human genome (UCSC HG19). We used their online sequence alignment quality comparison service to verify that results of our FPGA-accelerated version are indistinguishable from those obtained with the BWA-MEM algorithm. We used BWA-MEM version 0.7.7 [6].

VI. RESULTS

The results are summarized in Table III. Number of chunks indicates how many chunks are sent to the FPGA per batch. A value of one results in serial behavior, as then SMEM Generation and Seed Extension do not overlap. The last column shows the number of base pairs aligned per second.

A. Seed Extension Kernel

The table shows that the FPGA implementation of the Seed Extension kernel is up to three times faster than execution on the CPU, or 1.5 times faster when comparing a single module against one Xeon core. This implementation is fast enough to completely hide the execution of the Seed Extension kernel through overlapping its execution with SMEM Generation. Using a more advanced technique, such as VLPL, would allow us to achieve an even larger performance gain in Seed Extension, up to five-fold as compared to software-only execution (refer to Section IV-D). However, this would only benefit overall performance negligibly.

Note that the executions with only one chunk show slightly higher Seed Extension performance, due to less overhead from the chunking process. However, overall program execution time is lower, as no overlap between the two kernels is realized (refer to Figure 2 for more details).

B. Overall Program Execution

Offloading the Seed Extension kernel onto the FPGA results in an overall improvement to BWA-MEM execution time of up to 45%. Given the fact that BWA-MEM execution time is spread over three kernels (see Section II-B), we manage to attain 96% of the theoretically possible speedup of 47% from accelerating just this one kernel. Different numbers of chunks do not measurably impact performance, as long as overlap is possible between Seed Extension and SMEM Generation. A chunk size of one shows the isolated performance gain from Seed Extension acceleration without overlap. Note that BWA-MEM itself already offers multi-threaded execution.

VII. DISCUSSION

By optimizing one of the three main BWA-MEM kernels, we realized an increase in application performance by up to 45%. Focusing on only one kernel leaves us exposed to the limitations clearly set out by Amdahl's law, limiting the potential gains in performance. Our next efforts are hence focused on accelerating the other kernels.

The Seed Extension kernel proved to be a good fit to port to the FPGA, although it is obvious that even just porting one

TABLE III
EXECUTION TIME AND SPEEDUP FOR THE GCAT ALIGNMENT QUALITY BENCHMARK

			Seed Extension Kernel		Overall Program		
Test	**Platform**	**# of Chunks**	**Execution Time**	**Speedup**	**Execution Time**	**Speedup**	**Throughput**
Single-Ended	CPU only	-	167 s	-	530 s	-	2.3 Mbp/s
Data	CPU+FPGA	11	62 s	2.69x	366 s	1.45x	3.3 Mbp/s
	CPU+FPGA	6	62 s	2.70x	365 s	1.45x	3.3 Mbp/s
	CPU+FPGA	1	61 s	2.73x	412 s	1.29x	2.9 Mbp/s
Pair-Ended	CPU only	-	172 s	-	545 s	-	2.2 Mbp/s
Data	CPU+FPGA	11	63 s	2.75x	402 s	1.35x	3.0 Mbp/s
	CPU+FPGA	6	62 s	2.78x	400 s	1.36x	3.0 Mbp/s
	CPU+FPGA	1	61 s	2.82x	447 s	1.22x	2.7 Mbp/s

kernel has wider implications to the program structure than just replacing a single function call: limitations in memory transfer efficiency forced us to reorder the program execution into batches to allow for larger, more efficient data transfers. Moreover, the acceleration potential of using FPGAs is largely dependent on data size. The huge parallelism an FPGA can offer, granting $O(n)$ scaling as compared to $O(n^2)$ on the host CPU, will become much more apparent at longer read sizes: a read length of 1,000 symbols would result in a ten-fold speedup, compared to the 1.5x speedup we managed to attain. Hence, it is important to have a deep understanding of the data set at hand before applying a general solution. Finally, knowledge of the extension length distribution is required to implement a PE-module design with optimal efficiency.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we have presented the initial results of our efforts to accelerate BWA-MEM. We propose the first accelerated version of BWA-MEM, offloading one of the three main program kernels onto an FPGA and overlapping its execution. We implemented the Seed Extension kernel as a systolic array and achieved performance for this kernel up to three times faster than software-only execution. This translates into an overall improvement to execution time up to 45%, close to the theoretical maximum of 47%, as the kernel's execution time is almost completely hidden. Moreover, we have presented generally applicable techniques to improve the performance of variable length Smith-Waterman systolic arrays by up to three times, with very little area overhead.

Our next efforts will focus on offloading the other kernels of the BWA-MEM algorithm onto the FPGA, for which SMEM Generation is a natural candidate. We will also investigate the implementation of a VLPL-module, mostly as area savings measure, as the gains in speed can be used to reduce the allocated space of the kernel on the FPGA. Successful acceleration of BWA-MEM will bring us one step closer to overcoming the computational bottlenecks inherent in the Next Generation Sequencing genomics pipeline.

IX. ACKNOWLEDGMENTS

The authors would like to thank Bluebee and Convey Computer for kindly making available the resources for testing and implementation work, and in particular thank Giacomo Marchiori from Bluebee for all his efforts.

REFERENCES

[1] H. Cao, Y. Wang, W. Zhang, X. Chai, X. Zhang, S. Chen, F. Yang, C. Zhang, Y. Guo, Y. Liu, et al. A short-read multiplex sequencing method for reliable, cost-effective and high-throughput genotyping in large-scale studies. *Human mutation*, 34(12):1715–1720, 2013.

[2] Convey Computer. The Convey HC-2 architectural overview. http://www.conveycomputer.com. Accessed: 2014-11-04.

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

[4] G. Highnam, J. J. Wang, D. Kusler, J. Zook, V. Vijayan, N. Leibovich, and D. Mittelman. An analytical framework for optimizing variant discovery from personal genomes. *Nature communications*, 6, 2015.

[5] B. Langmead, C. Trapnell, M. Pop, S. L. Salzberg, et al. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol*, 10(3):R25, 2009.

[6] H. Li. Burrows-Wheeler Aligner. http://bio-bwa.sourceforge.net/. Accessed: 2014-11-04.

[7] H. Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv preprint arXiv:1303.3997*, 2013.

[8] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[9] H. Li, J. Ruan, and R. Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome research*, 18(11):1851–1858, 2008.

[10] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[11] T. Oliver, B. Schmidt, and D. Maskell. Hyper custo- mized processors for bio-sequence database scanning on FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 229–237. ACM, 2005.

[12] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[13] C. W. Yu, K. Kwong, K.-H. Lee, and P. H. W. Leong. A Smith-Waterman systolic cell. In *New Algorithms, Architectures and Applications for Reconfigurable Computing*, pages 291–300. Springer, 2005.

[14] P. Zhang, G. Tan, and G. R. Gao. Implementation of the Smith-Waterman algorithm on a reconfigurable supercomputing platform. In *Proceedings of the 1st international workshop on High-performance reconfi- gurable computing technology and applications: held in conjunction with SC07*, pages 39–48. ACM, 2007.

2

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

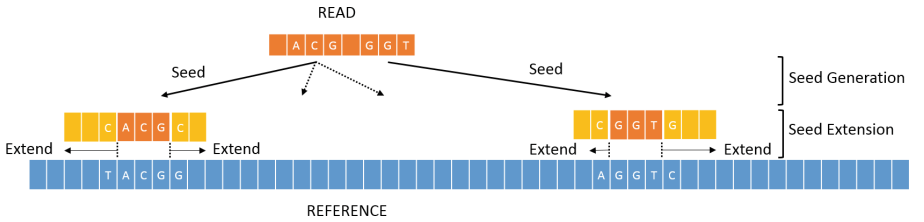


Fig. 1. BWA-MEM processes reads using the Seed-and-Extend paradigm: for each read, likely mapping locations on the reference are found by searching for exactly matching subsequences between the read and the reference, so called *seeds*. Then, these seeds are extended in both directions using a Smith-Waterman-like 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

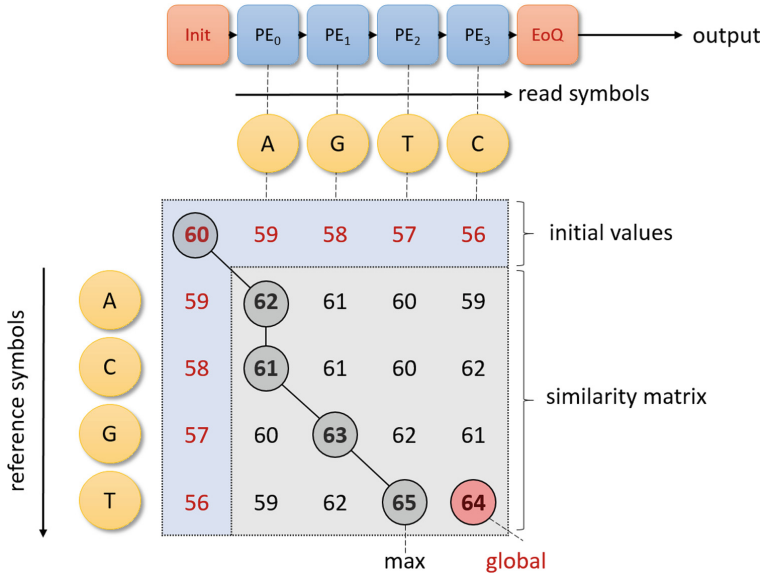


Fig. 2. Extend algorithm similarity matrix with initial score 60 showing local alignment with maximum score and global maximum score. Read symbols map one-to-one onto systolic array Processing Elements. Matrix entries only depend on top, top-left, and left neighbor. Thus, anti-diagonals can be processed 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: $LBF_{old} = LBF_{new}$
3: LBF_{new} = (HostBusyPct / GPUBusyPct) × LBF_{old}
4: $LBF_{new} = \min(1, (LBF_{new} + LBF_{old}) / 2)$
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.

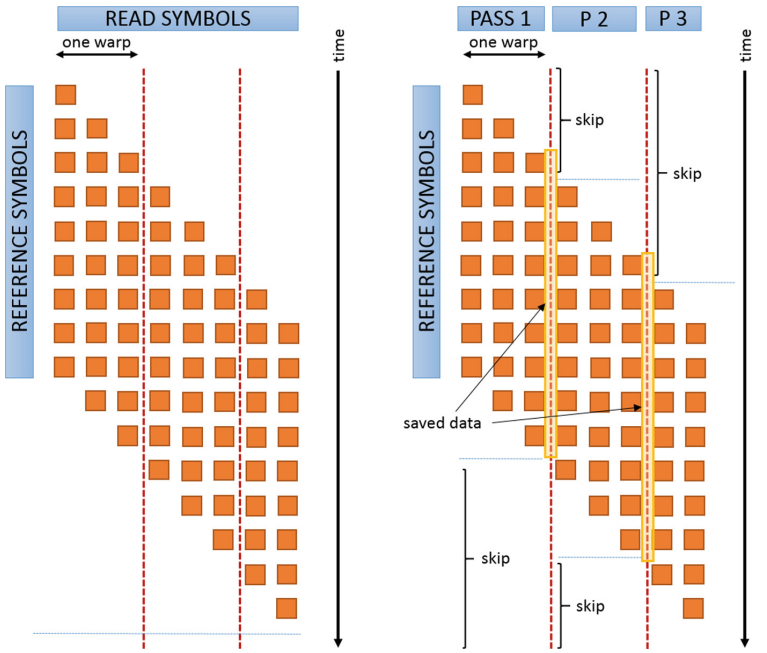


Fig. 3. Systolic array-based GPU Extend kernel implementation. Extension symbols are mapped one-to-one on CUDA cores, reference symbols are fed each iteration of the loop. After each iteration, data is exchanged through shared memory (left). The single warp-based implementation makes multiple passes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

Platform	Test	Extend kernel		Overall program		
		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.

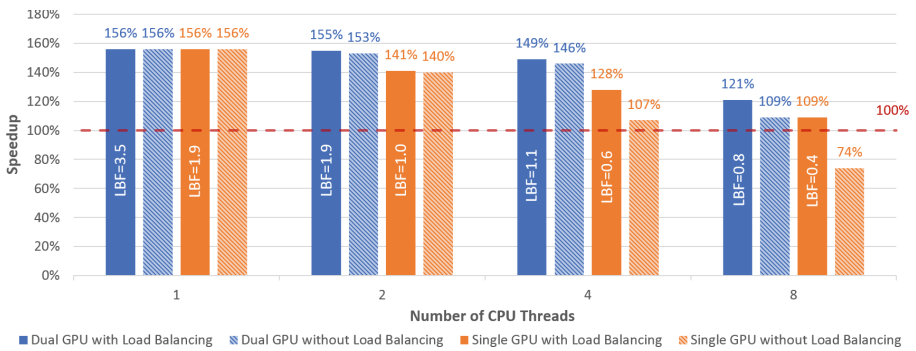


Fig. 4. Overall application speedup for varying number of CPU threads and single and dual GPUs. Results shown with load balancing enabled and disabled. The adaptive load balancing ensures efficient host and accelerator usage and provides an overall application speedup even for GPU-constrained scenarios,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

CHAPTER 3

Optimized Implementations with Efficient Systolic Arrays

"The pessimist complains about the wind;
The optimist expects it to change;
The realist adjusts the sails."

—William Arthur Ward

SOPHISTICATED techniques were proposed in the previous chapter, addressing one source of inefficiency to systolic array performance. Normally, a mismatch between the number of PEs in the systolic array and the read query length results in lowered efficiency as some of the PEs remain idle, since no read symbols are mapped onto them. Through the judicious use of Exit Points, and Variable Logical and/or Physical Length designs, this source of inefficiency can be completely eliminated. In this chapter, the remaining source of inefficiency is addressed. In a standard systolic array implementation, only a single sequence alignment is performed at a time. Thus, three phases during the sequence alignment can be distinguished: a startup phase where the reference sequence symbols are fed into the array and more and more PEs take part in the computation; an active phase, where all PEs join the sequence alignment computation; and a shutdown phase, where more and more PEs are finished and have to wait for the others to complete their computation. Naturally, the array is only at maximum efficiency during the active phase, whereas the other two phases are characterized by reduced efficiency. Worse still, short reference sequences might not even contain an active phase.

In this chapter, a streaming systolic array architecture is proposed that supports an arbitrary number of active sequence alignments simultaneously, thus eliminating the issue of idle PEs during the startup and/or shutdown phases and, in turn, resulting in a maximally-efficient systolic array. Furthermore, two greatly improved FPGA and GPU implementations of BWA-MEM are presented that are able to completely hide the Seed Extension execution time from overall program execution by offloading the Seed Extension kernel onto hardware and overlapping its execution with the remaining BWA-MEM kernels, resulting in the maximum two-fold speedup. The highest efficiency is obtained when the accelerated Seed Extension kernel time and remaining BWA-MEM kernels require an equal amount of time so that neither the host or the accelerator sits idle. This is analyzed in great detail.

3.1. MAIN CONTRIBUTIONS

The main contributions of this chapter are as follows:

- An optimized systolic array architecture is presented that uses streamed loading of sequences and implicit synchronization to fully eliminate the impact of reference sequence length on efficiency. Together with the techniques proposed in Chapter 2, this removes the traditional problem of keeping all PEs of a systolic array active, and the result is an implementation with an overall efficiency of 99.8%. An OpenCL implementation of the Smith-Waterman algorithm is proposed using the above describe optimized systolic array, resulting in the most efficient and most performant FPGA Smith-Waterman implementation to date, reaching 214 GCUPS on a single Intel Arria10 GX development board **[BIBE2017]**.

- Two optimized accelerated BWA-MEM implementations are proposed that each offer the maximum possible two-fold speedup attainable from completely eliminating the Seed Extension kernel from overall program execution time. The Alpha Data FPGA implementation improves on the earlier implementation by being both faster and more power-efficient, while at the same time requiring an equivalent of only about 23% of the FPGA resources. The optimized GPU implementation also contains a number of improvements, including memory access optimizations, the use of a single kernel monolithic kernel, and improved handling of the reference length. Moreover, recommendations are made to further improve the suitability of the GPU for sequence alignment-type problems **[FCCM2016], [CAN2017]**.

- A scalability analysis is performed to estimate the optimal balance in resources between a host machine and an accelerator, be it FPGA or GPU. Since the accelerator performs the Seed Extension tasks, while the host performs all other BWA-MEM tasks, optimal efficiency is achieved when these two require equal amounts of time and when the processing capabilities are suitably matched **[CAN2017]**.

3.2. RESEARCH ARTICLES

This chapter is based on the following articles:

1. **E.J. Houtgast**, V.M. Sima, G. Marchiori, K.L.M. Bertels, and Z. Al-Ars, *Power-Efficient Accelerated Genomic Short Read Mapping on Heterogeneous Computing Platforms*, 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), May 2016, Washington DC, USA.

2. **E.J. Houtgast**, V.M. Sima, K.L.M. Bertels, and Z. Al-Ars, *An Efficient GPU-Accelerated Implementation of Genomic Short Read Mapping with BWA-MEM*, ACM SIGARCH Computer Architecture News 44 (4), p38-43, 2017.

3. **E.J. Houtgast**, V.M. Sima, and Z. Al-Ars, *High Performance Streaming Smith-Waterman Implementation with Implicit Synchronization on Intel FPGA using OpenCL*, 17th International Conference on Bioinformatics and Bioengineering (BIBE), Oct 2017, Washington DC, USA.

2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines

Power-Efficient Accelerated Genomic Short Read Mapping on Heterogeneous Computing Platforms

Ernst Joachim Houtgast*†, Vlad-Mihai Sima†, Giacomo Marchiori†, Koen Bertels* and Zaid Al-Ars*
*Computer Engineering Lab, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands
†Bluebee, Molengraaffsingel 12-14, 2629 JD Delft, The Netherlands
E-mail: {ernst.houtgast, vlad.sima, giacomo.marchiori}@bluebee.com, {k.l.m.bertels, z.al-ars}@tudelft.nl

***Abstract*—We propose a novel FPGA-accelerated BWA-MEM implementation, a popular tool for genomic data mapping. The performance and power-efficiency of the FPGA implementation on the single Xilinx Virtex-7 Alpha Data add-in card is compared against a software-only baseline system. By offloading the Seed Extension phase onto the FPGA, a two-fold speedup in overall application-level performance is achieved and a 1.6x gain in power-efficiency. To facilitate platform and tool-agnostic comparisons, the base pairs per Joule unit is introduced as a measure of power-efficiency. The FPGA design is able to map up to 34 thousand base pairs per Joule.**

Introduction— The extreme scale of genomics data necessitates the use of high performance and power-efficient solutions. BWA-MEM [1] is the de facto standard for mapping DNA short reads onto a reference genome. Following the Seed-and-Extend paradigm, exactly matching seeds are generated for each read, which are subsequently extended using inexact matching similar to the Smith-Waterman algorithm. Here, we accelerate this Seed Extension phase, which forms a major bottleneck in BWA-MEM requiring 30%-50% of total execution time. Few accelerated BWA-MEM implementations exist: two implementations on the Convey HC-2^{EX} platform with four Xilinx Virtex-6 FPGAs, one accelerating only the Seed Extension phase [2] and achieving a 1.5x speedup, and one accelerating multiple phases [3] for an overall 2.6x speedup; and a GPU implementation [4]. Our work extends [2], but is able to achieve a 2.0x speedup using only a single Xilinx Virtex-7 FPGA, which offers about 23% of the resources as compared to the Convey platform.

Approach— Our implementation offloads the Seed Extension phase that performs the inexact matching onto an FPGA, accelerating the Smith-Waterman-like algorithm using a systolic array. Our design contains six modules that are each able to process the Seed Extension phase. The remainder of the logic is filled with arbitration logic to distribute reads, with the PCI-Express interface and with the memory controller. The design is limited by the amount of LUTs available.

Results— Results were gathered using BWA-MEM v0.7.8, which is highly multi-threaded, with the publicly available 150bp-se-small-indel GCAT data set [5]. To measure the power-efficiency, an emonPi energy monitor [6] was used to track the system-level power consumption as measured at the power plug. Performance and power-efficiency results are summarized in Table I. On a kernel-level, the FPGA is 1.8x faster as compared to software-only execution for the Seed Extension phase. In our implementation, this phase is executed in parallel with the other phases, thus resulting into a two-fold improvement to overall application performance. Note that the accelerated Seed Extension phase requires less than half of the overall application execution time. This shows that the FPGA is not fully utilized and can be used to accelerate a more powerful system. Moreover, the FPGA-accelerated implementation is much more power-efficient: it requires only 35 kJ, an energy efficiency improvement of 60%. The power-efficiency would be even greater for a more balanced system. A conservative estimate indicates that the FPGA-accelerated platform is able to achieve an up to 2.1x improvement in power-efficiency and is able to map up to 44 kbp/J.

REFERENCES

[1] H. Li, "Aligning Sequence Reads, Clone Sequences and Assembly Contigs with BWA-MEM," *arXiv preprint arXiv:1303.3997*, 2013.
[2] E. Houtgast, V. Sima, K. Bertels, and Z. Al-Ars, "An FPGA-Based Systolic Array to Accelerate the BWA-MEM Genomic Mapping Algorithm," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*. IEEE, 2015.
[3] N. Ahmed, V. Sima, E. Houtgast, K. Bertels, and Z. Al-Ars, "Heterogeneous Hardware/Software Acceleration of the BWA-MEM DNA Alignment Algorithm," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 240–246.
[4] E. Houtgast, V. Sima, K. Bertels, and Z. Al-Ars, "GPU-Accelerated BWA-MEM Genomic Mapping Algorithm Using Adaptive Load Balancing," Accepted for Publication in Architecture of Computing Systems, 2016.
[5] G. Highnam, J. J. Wang, D. Kusler, J. Zook, V. Vijayan, N. Leibovich, and D. Mittelman, "An Analytical Framework for Optimizing Variant Discovery from Personal Genomes," *Nature comm.*, vol. 6, 2015.
[6] The OpenEnergyMonitor Project, "OpenEnergyMonitor Website," http://openenergymonitor.org/, accessed: 2015-12-14.

TABLE I: POWER CONSUMPTION, PERFORMANCE AND ENERGY-EFFICIENCY FOR SOFTWARE-ONLY AND FPGA-ACCELERATED PLATFORMS (TESTED ON: INTEL CORE I7-4790 @ 3.6 GHZ + 16 GB RAM; ALPHA DATA ADM-PCIE-7V3 WITH XILINX VIRTEX-7 XC7VX690T-2 @ 160 MHZ + 16 GB RAM)

	Kernel Performance		Overall Performance		Power Consumption		Energy Efficiency		
Platform	**Time**	**Speedup**	**Time**	**Speedup**	**Idle**	**Load**	**Power**	**Efficiency**	**Improvement**
Software-Only	237 s	-	552 s	-	37 W	105 W	58 kJ	20.7 kbp/J	-
FPGA-Accelerated	129 s	**1.8x**	272 s	**2.0x**	62 W	129 W	35 kJ	34.1 kbp/J	**1.6x**

978-1-5090-2356-1/16 $31.00 © 2016 IEEE
DOI 10.1109/FCCM.2016.17

An Efficient GPU-Accelerated Implementation of Genomic Short Read Mapping with BWA-MEM

Ernst Joachim Houtgast[1,2], Vlad-Mihai Sima[2], Koen Bertels[1] and Zaid Al-Ars[1]
[1] Computer Engineering Lab, TU Delft, Mekelweg 4, 2628 CD Delft, The Netherlands
[2] Bluebee, Laan van Zuid Hoorn 57, 2289 DC Rijswijk, The Netherlands
Corresponding author: ernst.houtgast@bluebee.com

3

ABSTRACT

Next Generation Sequencing techniques have resulted in an exponential growth in the generation of genetics data, the amount of which will soon rival, if not overtake, other Big Data fields, such as astronomy and streaming video services. To become useful, this data requires processing by a complex pipeline of algorithms, taking multiple days even on large clusters. The mapping stage of such genomics pipelines, which maps the short reads onto a reference genome, takes up a significant portion of execution time. BWA-MEM is the de-facto industry-standard for the mapping stage.

Here, a GPU-accelerated implementation of BWA-MEM is proposed. The Seed Extension phase, one of the three main BWA-MEM algorithm phases that requires between 30%-50% of overall processing time, is offloaded onto the GPU. A thorough design space analysis is presented for an optimized mapping of this phase onto the GPU. The resulting systolic-array based implementation obtains a two-fold overall application-level speedup, which is the maximum theoretically achievable speedup. Moreover, this speedup is sustained for systems with up to twenty-two logical cores. Based on the findings, a number of suggestions are made to improve GPU architecture, resulting in potentially greatly increased performance for bioinformatics-class algorithms.

1. INTRODUCTION

The introduction of Next Generation Sequencing (NGS) techniques has resulted in drastic, ongoing, cost reduction of genomic sequencing, which, in turn, has led to an enormous growth in the amount of genetic DNA data that is being sequenced. High-throughput sequencing facilities are coming online around the world as facilities worldwide embrace NGS [2]. The amount of data being generated is projected to rival, if not outright overtake, other key Big Data-fields, such as astronomy and streaming video services [13].

NGS machines output so-called *short reads*, short fragments of DNA of at most a few hundred base pairs (bp) in length. This data requires extensive processing using a *genomics pipeline*, which typically contain multiple stages with a number of highly complex algorithms. In the case of a DNA sequencing pipeline, first, the millions of short reads generated are mapped onto a reference genome. Then, these mapped reads are sorted and duplicates are marked or removed. Finally, the aligned data is compared at several positions with known possibilities, in order to determine the most probable variant. Only then the data is ready for consumption by the end-user, such as a clinician or researcher. These variants, or mutations, are generally what is of interest, as such a mutation could give insight on which is the most effective treatment to follow for the particular illness a patient has. The mapping stage takes a significant portion of processing time for a typical pipeline execution, around 30%-40%, depending on data set and platform.

A sequencing run on an Illumina HiSeq X, a state-of-the-art NGS sequencer, produces data in the order of 450 GB. For cancer data sets, this data requires multiple days of processing, even on 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. As heterogeneous computing holds great potential to provide large advantages in speed and efficiency, in this paper, we demonstrate the effectiveness of GPU-based acceleration of BWA-MEM, the most widely used tool for the mapping stage of genomics pipelines.

The following contributions are made: 1) an optimized GPU-based implementation of the BWA-MEM Seed Extension phase, resulting in an overall application-level speedup of up to 2x; 2) a thorough discussion of the design space analysis, providing key insight into the requirements of a highly optimized implementation; and 3) recommendations to further improve the GPU architecture that would allow even higher performance for bioinformatics-class applications.

The remainder of this paper is organized as follows. In Section 2, related work is discussed. In Section 3, background information is given on the BWA-MEM algorithm and, in particular, on the Seed Extension phase. In Section 4, the advantages and disadvantages of various implementation architectures are reviewed. In Section 5, the results are presented. Section 6 contains a discussion of the results and recommendations are made to improve the GPU architecture. Section 7 concludes the paper.

2. RELATED WORK

Numerous GPU-accelerated implementations of short read mapping tools exist, notable examples include SOAPv3 [9] and CUSHAW [10]. Similar to BWA-MEM and most other state-of-the-art mapping tools, these consist of an Exact Matching phase using the Burrows-Wheeler transform to find exactly matching subsequences, followed by an Inexact Matching phase. However, these implementations are limited in the flexibility of their Inexact Matching algorithm, allowing only for a small number of mismatches (CUSHAW), or by disallowing gaps in the alignment (SOAPv3).

Using a variant of the Smith-Waterman (SW) algorithm [12] for its Inexact Matching, BWA-MEM does not impose such limitations. For example, gaps do not influence performance. The SW algorithm is a dynamic programming technique able to find the optimal match between two subsequences given a certain scoring scheme. Many accelera-

To appear in the International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies, July 2016, Hong Kong.

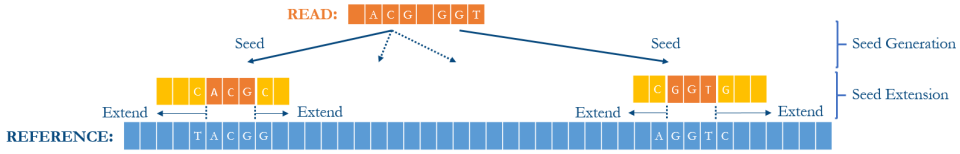


Figure 1: BWA-MEM processes reads using the Seed-and-Extend paradigm: for each read, likely mapping locations on the reference are found by searching for *seeds*, exactly matching subsequences between the read and the reference. These seeds are then extended in both directions using a Smith-Waterman-like approach allowing for inexact matches. The best scoring alignment is selected.

ted implementations of this algorithm exist (e.g., [8], [11]). However, all these implementations perform one complete sequence matching per compute thread, making such an implementation unsuitable for direct application onto BWA-MEM, as it requires batching and sorting of larger groups of work. Section 3.2 explains in more detail why such a parallelization strategy is inapplicable for BWA-MEM.

To the authors' knowledge, only a few accelerated implementations of BWA-MEM exist: two FPGA implementations of BWA-MEM on the Convey supercomputing platform: one offloading the Seed Extension phase onto four Xilinx Virtex-6 FPGAs [4] obtaining a 1.5x speedup, the other accelerating multiple BWA-MEM phases [1] obtaining a 2.6x speedup; and a GPU-accelerated implementation of the Seed Extension phase [5], achieving a 1.6x speedup. This work improves upon [5], obtaining far better results: a two-fold speedup for a system with up to twenty-two logical cores is obtained, compared to an at most 1.6x speedup for a system with up to four cores. Moreover, an NVIDIA GeForce GTX 970 is used, compared to using a setup with dual NVIDIA GeForce GTX TITAN X, equivalent to about one-third of the GPU resources. Note that all these implementations are actual production-quality implementations.

3. BACKGROUND

There are a number of traits that bioinformatics-class algorithms share, making them interesting, but nevertheless challenging candidates for acceleration efforts. The two most important ones are outlined below:

Extreme-Scale Data Size: The data size that many bioinformatics applications deal with are of an enormous magnitude, for example illustrated in the case of NGS sequencing. A single human genome contains three billion base pairs. A base is one out of four possible nucleotides (A, C, G or T). Moreover, the sequencer also provides a quality score for each base, which indicates the confidence with which the nucleotide was read. Finally, as only short fragments are sequenced and this data often contains errors, it is common practice to read the genome multiple times, a *coverage* of 30x or more being typical. This results in a compressed output size of around 100 GB.

Often, this huge amount of data coincides with an abundance of parallelism. For example, in the case of BWA-MEM, short reads can be mapped in parallel, as there exist no dependencies between them.

Complex Multikernel Algorithms: Typical bioinformatics algorithms do not consist of a single phase that dominates execution time, but instead perform a number of time-consuming steps. For example, BWA-MEM processing is spread over three distinct stages, making acceleration of this algorithm more challenging, as not only does it require the adaptation of multiple separate algorithms, but also care has to be taken to not shift the bottleneck to another part of the application, limiting the benefit of any potential speedup as per Amdahl's law. This makes it quite difficult to obtain larger performance gains.

3.1 The BWA-MEM Algorithm

The goal of the BWA-MEM algorithm is to find the best mapping of a short read onto a reference genome [7]. To achieve this, it makes use of the Seed-and-Extend paradigm (refer to Figure 1), a two-step method consisting of an Exact Matching phase and an Inexact Matching phase (for details, see [1]). First, for each short read Seed Generation is performed: exactly matching subsequences of the read and reference called *seeds* are identified using a Burrows-Wheeler Transform-based index. The BWT-method allows for efficient string-lookup and forms the fundament of almost all contemporary state-of-the-art mapping tools. A single short read can have many such seed locations identified. Generated seeds that are found to be in close proximity of each other on the reference genome are grouped into *chains*.

The Seed Generation phase is followed by a Seed Extension phase. Here, seeds found earlier on are *extended* using an algorithm similar to the widely-used Smith-Waterman algorithm, using a scoring system that awards matches and penalizes mismatches, insertions and gaps. Typically, not all seeds are extended. Instead, on average only one seed per chain is extended. Out of all the extended seeds, the highest scoring match is chosen as final *alignment*.

3.2 Seed Extension Phase

BWA-MEM contains three main computational phases: Seed Generation, Seed Extension and Output Generation. During Output Generation the best alignment is selected and the output is written. Seed Extension typically requires between 30%-50% of execution time [5]. As per Amdahl's law, the maximum obtainable speedup for only accelerating this phase is thus limited to a two-fold speedup at best.

This paper focuses on GPU-based acceleration of the Seed Extension phase, which consists of two main parts: an outer loop that loops over all the seeds identified for the read during Seed Generation, and an Inexact Matching kernel, which performs the Smith-Waterman-like extension.

There are no dependencies between reads and thus reads can be processed in parallel. For each read, the groups of chains are processed iteratively, as the check for overlap between earlier found alignment regions introduces a dependency in the program order. This dependency is the main reason why typical Smith-Waterman GPU-implementations are not applicable for the case of BWA-MEM: they ob-

tain their speed by performing many Smith-Waterman alignments in parallel, which are batched and sorted together in larger groups of approximately the same length for load balancing purposes. Due to the highly dynamic nature of the Inexact Matching invocations, this is impractical to achieve for BWA-MEM, and would at the very least require a major algorithm overhaul, if at all possible.

3.3 Inexact Matching Kernel

The Inexact Matching algorithm is similar to the popular Smith-Waterman dynamic programming algorithm, which computes for a given scoring scheme the optimal alignment between two subsequences by filling a similarity matrix, resulting in a maximum score. Backtracking can be used to obtain the actual path through the similarity matrix that results in the optimal alignment. However, the algorithm is computationally expensive, being of $O(read \times reference)$. Therefore, most mapping tools use an initial Seeding-phase to find likely mapping locations, and only then perform localized extension of these seeds.

The Inexact Matching algorithm of BWA-MEM is slightly different from regular Smith-Waterman. Firstly, since the algorithm is used to extend a seed, the initial values used in the similarity matrix are non-zero. Secondly, some additional outputs are generated, most importantly the location of the maximum in the similarity matrix, and a global maximum with its location. Since each value in the similarity matrix only depends on its top, left, and top-left neighbor, the anti-diagonals of the similarity matrix can be computed in parallel, thus making a systolic array a natural implementation approach. Each column of the similarity matrix is processed in parallel by a Processing Element (PE) of the systolic array, thus reducing the processing time from $O(read \times reference)$ to $O(read + reference)$. This results in speedups of several orders of magnitude for longer read and reference sequences. However, as BWA-MEM is typically used for shorter reads of at most a few hundred base pairs, the observed speedup is more modest.

4. DESIGN SPACE EXPLORATION

As explained in Section 3.2, the BWA-MEM Seed Extension phase consists of two very 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. This outer loop performs the sequential tasks of control and branch operations to effectuate the looping over all seeds, proper decoding of the sequence and reference from main memory, and writes the result back to memory. In contrast, the Inexact Matching function is highly computationally intensive and can use as many threads as the systolic array has PEs. Thus, the implementation in [5] on which this work is based 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 are called appropriately. However, our tests show that CUDA Dynamic Parallelism brings about a large initialization penalty, making it unsuitable to use at this scale, where even a single read can generate thousands of calls, resulting in millions of invocations during a typical program execution. Therefore, the implementation here does not make use of Dynamic Parallelism, and instead uses a large monolithic kernel.

4.1 GPU-Based Inexact Matching

The main challenge of the GPU-accelerated Seed Extension function is the implementation of the Smith-Waterman-like Inexact Matching kernel. Typical GPU implementations of Smith-Waterman extract parallelism by performing many sequence alignments in parallel. Here, parallelism is extracted from an individual alignment by harnessing the parallelism residing in the anti-diagonals of the similarity matrix, through use of a systolic array (see Figure 2).

The *warp* is the basic unit of action on an NVIDIA GPU. All threads in a warp perform the same operation, and jobs are always scheduled onto one or more complete warps. Therefore, two types of systolic array implementations were considered. A "wide" systolic array implementation, mapping each PE onto a separate thread and using as many warps as needed, and a single warp implementation, using only a single warp and processes the similarity matrix in multiple passes. These approaches differ in their utilization of Shared Memory (SM), a limited on-chip resource. The amount of SM a thread block requires directly puts an upper bound on the number of thread blocks that can be resident, thus impacting performance.

As data flows through the systolic array, the PEs exchange data with their neighbors to share results and perform their computations. The "wide" implementation uses SM to simulate the data exchange between PEs. After computation of each antidiagonal, each PE passes its results to the next PE in the array. Thus, the amount of SM required depends on the length of the systolic array, which in turn depends on the length of the read. Explicit synchronization between warps is required after each step, which can be costly.

In contrast, the single warp implementation requires storage for the data produced at the "border" of each pass, which is fed back into the array during the next pass. Therefore, the amount of SM required depends on the length of the reference query. A large advantage of using a single warp is that intra-warp shuffle instructions can be used,

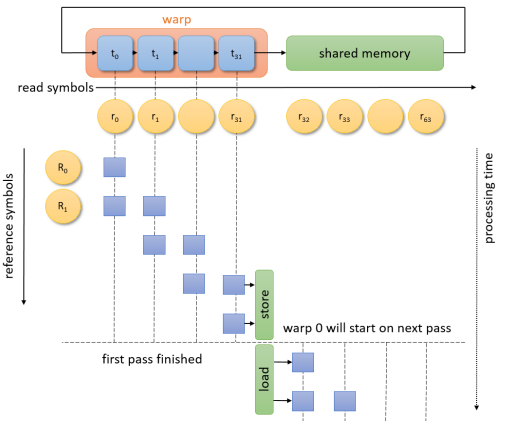


Figure 2: Read symbols map onto the systolic array of threads. Multiple passes are required to process all read symbols. Data exchange between passes is implemented using Shared Memory.

which allow threads within one warp to directly access each other's registers. This eliminates the need for data exchange through SM, saving a huge amount of bandwidth. In this approach, only the first PE in the warp needs to read, and the last PE needs to write temporary data. Another advantage is that intra-thread synchronization within one warp is cheaper. Compared to the "wide" implementation, the single-warp implementation has a secondary benefit as it eliminates a large drawback of systolic arrays: the difficulty of keeping all PEs busy. Depending on the similarity matrix dimensions, many of the PEs can be idle much of the time. With the single warp implementation, this inefficiency is drastically reduced by skipping those parts of the matrix where all the PEs of the pass would be idle.

To implement the Smith-Waterman algorithm, each CUDA thread performs the pseudo code as shown in Algorithm 1: each pass, the warp of threads is assigned a new part of the read symbols to process. Then, all the calculations are performed for this subsection of the similarity matrix. Each cycle, each thread reads its left neighbor's results, this way implementing the systolic array behavior.

4.2 Implementation Architecture

Due to the above reasons, the single-warp systolic array design is implemented. To maximize occupancy, Shared Memory and register usage was carefully balanced. The register count was fixed to use 64 registers per thread. The maximum number of storage between passes was chosen such to ensure that one thread block uses 2 kB of Shared Memory. Hence, up to 32 thread blocks can be resident per multiprocessor. Analysis performed with the NVIDIA Visual Profiler shows that the performance is mostly limited by latency of arithmetic and memory instructions. The memory subsystem is not very much utilized, as the Shared Memory bandwidth is only 158 GB/s and the device memory bandwidth is less than 5 GB/s. The GPU caching is effective, as device memory bandwidth is substantially lower than overall unified cache bandwidth.

Our approach is limited by the fact that the latest NVIDIA GPU architectures (Compute Capability 5.0) can have up to 2048 resident threads active per multiprocessor, but only 32 blocks. For optimal occupancy, thread blocks with at least 64 threads should be used, whereas here only 32 threads are used per block. Hence, occupancy is limited to at most 50%. In practice, up to about 35% occupancy is realized. Earlier Compute Capability versions were even more restrictive, only allowing 16 resident blocks per multiprocessor for Compute Capability 3.0+, or just 8 resident blocks per multiprocessor for earlier architectures. This would have a direct impact on the efficiency of this implementation.

Algorithm 1 Systolic Array CUDA Thread Pseudo Code

```
1: for (each pass) do
2:    Load current read symbol
3:    for (each reference symbol + warp size) do
4:       if (active) then
5:          Load left neighbor values
6:          Perform Seed Extension cell computations
7:       end if
8:    end for
9: end for
```

4.3 BWA-MEM Optimizations

A number of optimizations has been implemented, resulting in much better performance as compared to the GPU-based implementation in [5]:

Single Monolithic Kernel: Although in theory Dynamic Parallelism should help improve occupancy by lowering resource requirements, the incurred performance overhead makes it unfeasible to use when it needs to instantiate kernels dynamically on such an extremely large scale.

Memory Subsystem Optimizations: GPUs contain specialized memory subsystems. The reference and input data are placed inside read-only texture memory to take advantage of locality. Constants are used for parameters such as scoring variables to reduce register count.

Truncated Reference Length: Analysis of the Seed Extension algorithm shows that it is unnecessary to process the part of the similarity matrix where the reference is much longer than the read, given that this would imply numerous gaps or insertions, and thus a low score. The highest score will be found in the upper part of the similarity matrix.

5. EXPERIMENTAL RESULTS

The optimized GPU implementation described here was tested on a system with an Intel Core i7-4790 (3.6 GHz, eight logical cores), SpeedStep and Hyper-Threading enabled, containing 16 GB of DDR3 memory, and an NVIDIA GeForce GTX 970 with 1664 CUDA cores and 4 GB of onboard RAM. CUDA version 7.5 was utilized. Results for the GPU implementation described in [5] were obtained on a system with an Intel Core i7-4790 (4.0 GHz, eight logical cores), SpeedStep and Hyper-Threading enabled, containing 32 GB of DDR3 memory, and two NVIDIA GeForce GTX TITAN X cards, with 3,072 CUDA cores each. The FPGA results were obtained on a system with an Intel Core i7-4790 (3.6 GHz, eight logical cores), SpeedStep and Hyper-Threading enabled, containing 16 GB of RAM and a server-grade Alpha Data ADM-PCIE-7V3 card with a Xilinx Virtex-7 XC7VX690T-2 and 16 GB of on-board RAM, programmed with six Seed Extension modules at 160 MHz [6].

BWA-MEM version 0.7.8 was used with publicly available data sets for single-ended alignment (150bp-se-small-indel) and pair-ended alignment (150bp-pe-large-indel) from the Genome Comparison & Analytic Testing (GCAT) framework [3]. These contain about eight million reads of 150 base pairs, about 1.2 billion base pairs in total. Reads were aligned against the reference human genome (UCSC HG19).

5.1 Performance Results

Performance results are summarized in Table 1, given as execution time in seconds as well as in throughput in millions of base pairs per second. This facilitates cross-data set and cross-platform comparisons. To distinguish the GPU implementations, the implementation from [5] is referred to as *GPU-accelerated*, and the implementation proposed here is called *GPU-optimized*. These two implementations are compared to the FPGA-implementation from [6], which uses the server-grade Alpha Data add-in card. The time for the Seed Extension phase is omitted for the GPU-accelerated implementation, as the number reported there is not directly comparable, as it only includes the Inexact Matching computation time, and not the time required to process the outer loop. Moreover, the GPU-optimized implementation described here processes almost all reads, whereas the GPU-

Table 1: Execution time and speedup for the synthetic GCAT alignment quality benchmark

		Seed Extension Phase		Overall Application		
Test	**Platform**	**Execution Time**	**Speedup**	**Execution Time**	**Speedup**	**Throughput**
Single-Ended Data	Software-Only	237 s	-	552 s	-	2.2 Mbp/s
	FPGA-Accelerated [6]	129 s	1.8x	272 s	2.0x	4.5 Mbp/s
	GPU-Optimized	144 s	1.6x	278 s	2.0x	4.3 Mbp/s
	Software-Only [5]	218 s	-	510 s	-	2.4 Mbp/s
	GPU-Accelerated [5]	N/A	N/A	422 s	1.2x	2.9 Mbp/s
Pair-Ended Data	Software-Only	246 s	-	572 s	-	2.1 Mbp/s
	FPGA-Accelerated [6]	130 s	1.9x	289 s	2.0x	4.1 Mbp/s
	GPU-Optimized	141 s	1.7x	293 s	2.0x	4.1 Mbp/s

accelerated implementation is only able to process about 99.5% of all reads, leaving the most time-consuming reads for the host CPU. The results for the two software-only platforms differ slightly, as their clock frequency is slightly different (4.0 GHz vs 3.6 GHz). Both the GPU-optimized implementation and the FPGA-accelerated implementation are able to reach a 2x speedup, compared to software-only execution. The GPU-accelerated implementation only reaches a 1.2x speedup. The results for the GPU-optimized implementation are much better than for the GPU-accelerated implementation, even though a GPU-subsystem is used with only about 31% of the computational resources.

Note that we deliberately refrain from making a direct comparison between BWA-MEM and other read mapping tools, as in this field, strict reproducibility is critical, making performance of other tools irrelevant.

5.2 Scalability Analysis

Besides overall performance, scalability of the implementations is also important: the number of CPU cores a system can have for which the system is still accelerated with maximum speedup. This is estimated by considering the time required by the accelerator for the Seed Extension phase and regarding this as a lower bound to overall execution time. Assuming overall execution time scales linearly in processor core count, which for BWA-MEM is not unreasonable, the maximum number of logical CPU cores that can be effectively accelerated is thus determined. The results are summarized in Table 2. Results are also included for further optimized implementations that include certain straightforward improvements to scalability behavior, such as further decomposition and pipelining of the data preprocessing step that prepares the data for the GPU or FPGA, indicated as *FPGA-opt* and *GPU-opt* in the table.

The scalability results are also visually depicted in Figure 3, showing speedup compared to a host system with the same number of cores. The increased speedup when using eight threads may be caused by Hyper-Threading, which makes Seed Extension a larger part of overall execution time due to being the least memory-intensive phase, thus benefitting the least from Hyper-Threading.

Table 2: Scalability Analysis of the Implementations

	Execution Time			
Platform	**Seed Ext.**	**Overall**	**Utilization**	**Scalability**
FPGA	129 s	272 s	47%	16.8 cores
FPGA-opt	83 s	272 s	31%	26.1 cores
GPU	144 s	278 s	52%	15.4 cores
GPU-opt	101 s	282 s	36%	22.3 cores

6. DISCUSSION

This section presents lessons learned during the implementation work, and gives some recommendations to improve GPU architecture for bioinformatics-class problems.

6.1 Lessons Learned

The abundance of parallelism in BWA-MEM, and many bioinformatics-class algorithms in general, makes them interesting candidates for GPU-based acceleration. However, the complexity of these algorithms, with execution time distributed over multiple distinct phases, makes obtaining large overall application-level performance gains far from straightforward. Bottlenecks quickly shift towards the non-accelerated parts of the program.

For BWA-MEM Seed Extension, the existence of an outer loop dynamically calling Inexact Matching functions makes it ill-suited to apply typical Smith-Waterman acceleration methods. Therefore, a systolic array approach was used to accelerate the algorithm instead. To avoid large temporary storage requirements for data transfer between systolic array PEs, only a single warp was used, allowing the use of intra-warp shuffle. This greatly reduces Shared Memory bandwidth requirements. However, it puts an upper limit on the achievable occupancy, as only 32 threads are instantiated per thread block, whereas optimal occupancy can only be obtained with at least 64 threads per thread block.

Dynamic parallelism, as used in [5], seems like a valid approach given the two distinct parts of the Seed Extension algorithm. However, in practice, it brings about a large overhead. This shows the importance of testing a wide range of implementations, and not just choosing the approach that in theory should be the best.

6.2 Recommended Architecture Optimizations

The lessons learned during implementation of the GPU-based Seed Extension phase have given key insights in the requirements of bioinformatics-class algorithms. Therefore, here follow a number of suggestions to GPU architecture that could greatly improve performance for such algorithms:

Reduced Dynamic Parallelism Overhead: Reduced Dynamic Parallelism overhead could make this into a valid approach to reduce register and Shared Memory pressure, thus improving occupancy.

Increased Resident Blocks per Multiprocessor: The present limit of 32 resident blocks per multiprocessor limits the maximum obtainable occupancy for single warp thread blocks. Raising this limit to 64 resident blocks per multiprocessor could result in an up to 100% boost to performance, as this limitation is the major obstacle to higher performance in

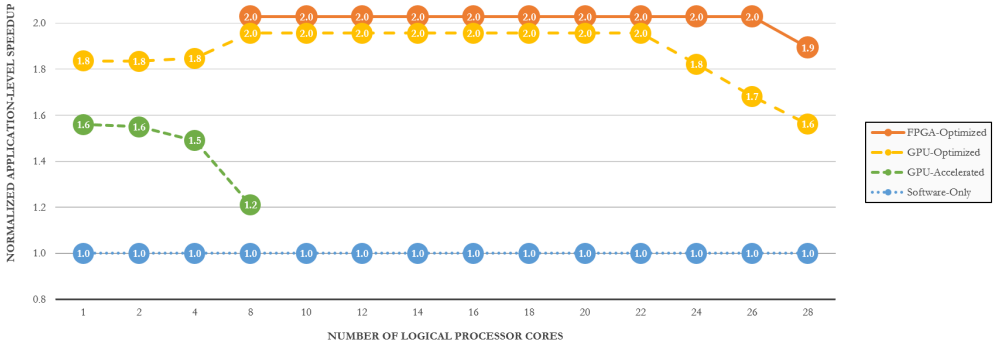


Figure 3: Estimated application-level speedup compared to software-only execution on a system with identical number of CPU cores. The GPU-Optimized implementation is able to sustain a two-fold speedup for systems with up to twenty-two CPU cores, greatly exceeding the results of the GPU-Accelerated implementation, which uses dual NVIDIA GeForce GTX TITAN X compared to the single GeForce GTX 970 here.

this work. Not only the implementation here would benefit, but also other Smith-Waterman implementations, a staple algorithm in bioinformatics. In general, this would improve any implementation that relies on the intra-warp shuffle capability and are thus limited to a single warp.

Native Low Precision Data Formats: Many problems in bioinformatics do not require high precision for their calculations. For example, the maximum value of entries in the Smith-Waterman similarity matrix can be easily determined based on the scoring parameters and length of both sequences. In many instances, even eight bits of precision is sufficient. The current native 32-bits of precision minimum results in wasted register and Shared Memory space, unless tricks are performed that require additional instructions.

7. CONCLUSIONS

In this paper, a GPU-accelerated implementation is described of the BWA-MEM genomic mapping algorithm. 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 twofold speedup in overall application-level performance. Analysis shows that this implementation is able to sustain this maximum speedup for a system with at most twenty-two logical cores. This can save days of processing time on the enormous real-world data sets that are typical of NGS sequencing.

The implementation presented here greatly exceeds the performance of the GPU implementation of [5], offering a higher speedup of 2x for systems with up to twenty-two cores, compared to 1.6x for systems with up to four cores, even while at the same time using a GPU-subsystem that only provides about 31% of the computational capabilities.

Although the work here focuses on BWA-MEM, a widely used genomic mapping tool, the approach used is valid for many similar Seed-and-Extend-based bioinformatics algorithms. Moreover, based on the insights obtained, a number of optimizations to GPU architecture are suggested: reduced Dynamic Parallelism overhead, increased number of resident blocks per multiprocessor, and native low precision data formats. These should greatly improve GPU performance for bioinformatics-class problems.

8. REFERENCES

[1] N Ahmed, V Sima, EJ Houtgast, KLM Bertels, and 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.

[2] James Hadfield and Nick Loman. Next Generation Genomics: World Map of High-throughput Sequencers. `http://omicsmaps.com`, 2016. Accessed: 2016-01-13.

[3] Gareth Highnam, Jason J Wang, Dean Kusler, Justin Zook, Vinaya Vijayan, Nir Leibovich, and David Mittelman. An Analytical Framework for Optimizing Variant Discovery from Personal Genomes. *Nature comm.*, 6, 2015.

[4] EJ Houtgast, V Sima, KLM Bertels, and 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.

[5] EJ Houtgast, V Sima, KLM Bertels, and Z Al-Ars. GPU-Accelerated BWA-MEM Genomic Mapping Algorithm Using Adaptive Load Balancing. In *Architecture of Computing Systems–ARCS*, pages 130–142. Springer, 2016.

[6] EJ Houtgast, V Sima, G Marchiori, KLM Bertels, and 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, May 2016.

[7] Heng Li. Aligning Sequence Reads, Clone Sequences and Assembly Contigs with BWA-MEM. *arXiv preprint arXiv:1303.3997*, 2013.

[8] Lukasz Ligowski and Witold 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*, pages 1–8. IEEE, 2009.

[9] Chi-Man Liu, Thomas Wong, Edward Wu, Ruibang Luo, Siu-Ming Yiu, Yingrui Li, Bingqiang Wang, Chang Yu, Xiaowen Chu, Kaiyong Zhao, and R. Li. SOAP3: Ultra-Fast GPU-Based Parallel Alignment Tool for Short Reads. *Bioinformatics*, 28(6):878–879, 2012.

[10] Yongchao Liu, Bertil Schmidt, and Douglas L Maskell. CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows-Wheeler transform. *Bioinformatics*, 28(14):1830–1837, 2012.

[11] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. CUDASW++ 3.0: Accelerating Smith-Waterman Protein Database Search by Coupling CPU and GPU SIMD Instructions. *BMC bioinformatics*, 14(1):117, 2013.

[12] TF Smith and MS Waterman. Identification of Common Molecular Subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[13] ZD Stephens, SY Lee, F Faghri, RH Campbell, C Zhai, MJ Efron, R Iyer, MC Schatz, S Sinha, and GE Robinson. Big Data: Astronomical or Genomical? *PLoS Biology*, 13(7), 2015.

2017 2017 IEEE 17th International Conference on Bioinformatics and Bioengineering

High Performance Streaming Smith-Waterman Implementation with Implicit Synchronization on Intel FPGA using OpenCL

Ernst Joachim Houtgast[1,2], Vlad-Mihai Sima[1]
[1]*Bluebee Research & Development*
Bluebee BV
Rijswijk, The Netherlands
E-mail: {ernst.houtgast, vlad.sima}@bluebee.com

Zaid Al-Ars[2]
[2]*Department of Computer Engineering*
Delft University of Technology
Delft, The Netherlands
E-mail: {e.j.houtgast, z.al-ars}@tudelft.nl

***Abstract*—The Smith-Waterman algorithm is widely used in bioinformatics and is often used as a benchmark of FPGA performance. Here we present our highly optimized Smith-Waterman implementation on Intel FPGAs using OpenCL. Our implementation is both faster and more efficient than other current Smith-Waterman implementations, obtaining a theoretical performance of 214 GCUPS. Moreover, due to the streaming, implicit synchronizing nature of our implementation, which streams alignments and places no restrictions on the number of alignments in flight, it achieves 99.8% of this performance in practice, almost three times as fast as previous implementations. The expressiveness of OpenCL results in a significant reduction in lines of code, and in a significant reduction of development time compared to programming in regular hardware description languages.**

***Keywords*-FPGA; OpenCL; Smith-Waterman; systolic array**

I. INTRODUCTION

The Smith-Waterman algorithm [1] can be used to find the optimal pairwise alignment between two (sub)sequences of symbols, which in the context of bioinformatics usually means sequences of amino acids (for protein sequences) or nucleotides (for DNA sequences). Given a certain scoring scheme that awards matching symbols and penalizes differences or missing symbols, it uses a dynamic programming approach to calculate the optimal alignment between the sequences. The continued growth of bioinformatics data sets makes optimized and/or accelerated implementations of key algorithms of vital importance.

Field-Programmable Gate Arrays (or FPGAs), with their flexible and reprogrammable substrate, are a natural fit for a computationally intensive algorithm such as the Smith-Waterman algorithm. However, programming FPGAs through hardware description languages such as VHDL or Verilog is difficult, being somewhat comparable to writing software in assembly language. The rise of higher level programming languages such as OpenCL makes FPGA programming a much more accessible venture.

In this paper we present the following contributions:

- An OpenCL FPGA Smith-Waterman implementation that, limited development complexity notwithstanding, outperforms other implementations almost threefold;
- A streaming systolic array architecture that eliminates a key design issue: low utilization of the systolic array.

The remainder of this paper is organized as follows. We review related work in Section II. In Section III, we explain the Smith-Waterman algorithm. In Section IV, the two key features of our implementation, streaming and implicit synchronization, are discussed. Methods and results are presented in Section V and VI, respectively. A discussion follows in Section VII. Section VIII concludes the paper.

II. RELATED WORK

As the Smith-Waterman algorithm [1] is a common algorithm in bioinformatics, it has received much attention to optimize the algorithm's performance, resulting in numerous accelerated implementations. The fastest software-only Smith-Waterman version is SSW [2], which extends the striped Smith-Waterman approach of Farrar [3]. These implementations make pervasive use of SIMD instructions to attain their excellent performance. However, accelerator-based implementations are still able to significantly outperform software-only implementations. For example, the GPU-based CUDASW++ 3.0 [4] is able to attain a performance of 119.0 GCUPS on a GeForce GTX 680. The highest performing FPGA-based implementation is the implementation from Sirasao[5], which attains a performance of 135.4 GCUPS using an AlphaData board with a Xilinx Virtex-7. Moreover, their FPGA-based design is significantly more efficient compared to most GPU-implementation, requiring an order of magnitude less power compared to a GPU-based approach. They measured a power-efficiency of 2.8 GCUPS/Watt compared to 0.24 GCUPS/Watt on the GPU.

Similar to the Sirasao implementation, we use OpenCL as our implementation platform. However, our streaming implementation with implicit synchronization makes pervasive use of OpenCL features such as kernels and channels to allow for a higher performing, and more importantly, a much more efficient Smith-Waterman implementation. Utilization of our design approaches 100%, compared to 57% utilization of the Sirasao design. As a result, our implementation outperforms their implementation by almost three-fold.

2471-7819/17/31.00 ©2017 IEEE
DOI 10.1109/BIBE.2017.00089

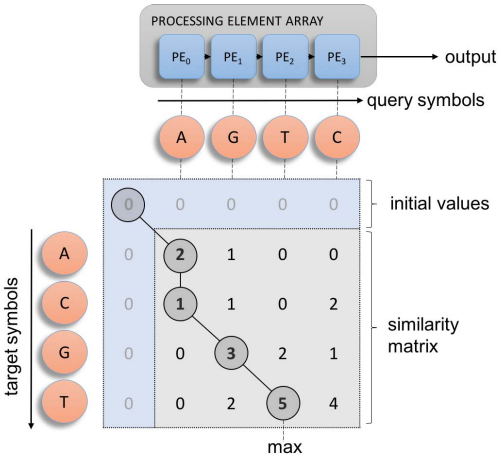


Figure 1. The Smith-Waterman algorithm operates by filling a dynamic programming-based similarity matrix. Cells inside the matrix are only dependent on their top, top-left, and left neighbor, allowing anti-diagonals of the matrix to be processed in parallel. This maps naturally onto a systolic array of Processing Elements. Each Processing Element calculates the column for one of the query symbols. The traceback phase works backwards from the highest scoring cell to produce the actual alignment.

III. Smith-Waterman Algorithm

The Smith-Waterman algorithm is guaranteed to find the optimal pairwise alignment of two sequences. It consists of two phases: first, it uses a dynamic programming approach to fill a similarity matrix, followed by a traceback phase to retrieve the optimal alignment. As the first phase is the most computationally demanding, it is the focus of this work.

The Smith-Waterman equations that govern similarity matrix score calculations are similar to the Needleman-Wunsch algorithm [6], except that disallowing negative values makes the algorithm search for optimal local alignments, as compared to optimal global alignments:

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + s(a_i, b_j) & : \text{(mis)match} \\ H_{i-1,j} - \textit{gap penalty} & : \text{insertion/deletion} \\ H_{i,j-1} - \textit{gap penalty} & : \text{insertion/deletion} \\ 0 & : \text{local alignment} \end{cases}$$

This is illustrated in Figure 1. The highest scoring cell in the similarity matrix indicates the optimal alignment score. From the above equations it is clear that each cell in the similarity matrix only depends on its top, top-left, and left neighbor. Therefore, anti-diagonals in the matrix are independent of one another and can be calculated in parallel: a wavefront of parallelism flows through the similarity matrix. This maps nicely onto a systolic array of Processing Elements, which is the typical approach when implementing the Smith-Waterman algorithm on an FPGA.

IV. Implementation Details

Our implementation uses the Intel FPGA SDK for OpenCL. Two key concepts of OpenCL are kernels and channels. An OpenCL kernel is a function executed on a compute device, so in the case of an FPGA, this is synthesized into actual hardware. Multiple identical kernels are synthesized to work in parallel to achieve higher throughput. An OpenCL channel is a mechanism that implements on-chip low-latency, high bandwidth communication between kernels. Our implementation makes pervasive use of channels, only using the on-board DDR for reading of the input sequences and writing of the output scores.

Figure 2 shows the kernels and channels of a single Smith-Waterman module. The largest area is reserved for the systolic array of Processing Elements, which calculates the Smith-Waterman similarity matrix. For each alignment, each Processing Element has its unique query symbol, whereas the symbols of the target sequence flow through the array. There are two key innovations that work in unison to allow for high utilization of the systolic array:

Implicit Synchronization: Our implementation does not use a dedicated control unit. Instead, control and synchronization is implicitly arranged within each kernel. Control and data signals flow from left to the right: the Input Parser sends packets to the Target and Query Loaders, and to the Result Parser. These packets contain all the information required to know how many iterations this particular alignment requires. This allows each kernel to work independently without explicit synchronization with other kernels.

Streaming: Typically, due to the central control, a systolic array is only able to work on a single alignment at a time. However, the distributed control of our implementation allows for a streaming nature. This is illustrated in Figure 3. A key enabler is the use of Query Buffers, which for each Processing Element hold the query symbols for upcoming alignments. Whenever the current alignment is finished, a New Read token is passed through the array, signaling to a PE that it should reinitialize and load a new Query symbol.

The combined effect of both innovations is that, except for the first alignment, processing time for an alignment only depends on the length of the target sequence, and is independent of query length. The only overhead is the New Read token that is passed to indicate a new alignment. Thus, very short target sequences do slightly impact efficiency. In contrast, efficiency of the systolic array is mostly dependent on the query length as compared to systolic array size, as for shorter queries part of the systolic array will be idle. To alleviate this, it would be possible to use multiple systolic arrays of different size (refer to [7] for more details). Our designs use multiple identically sized modules to utilize all available resources on the FPGA.

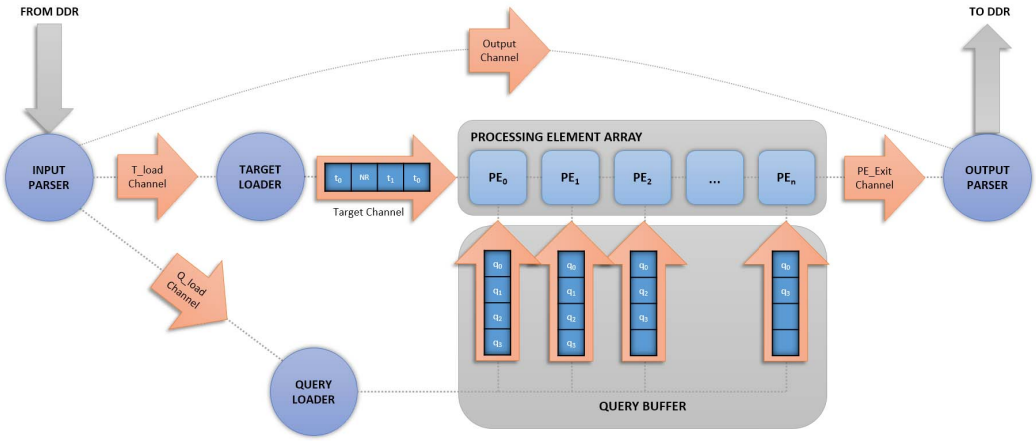


Figure 2. Overview of the kernels and channels of the Smith-Waterman module. Control and data signals flow from left to right through OpenCL channels, with no dependencies or loops, removing any limitation on the number of simultaneous alignments in flight. This way, kernels are decoupled from one another and operate asynchronously. The Query Buffer contains for each Processing Element a separate queue with query symbols for the alignments it needs to process. Whenever the Processing Element encounters the new read token, it checks against the query length to verify if it is active during this alignment; if so, it reads the next query symbol from its queue. Only the Input Parser and Output Parser communicate with the on-board DDR memory.

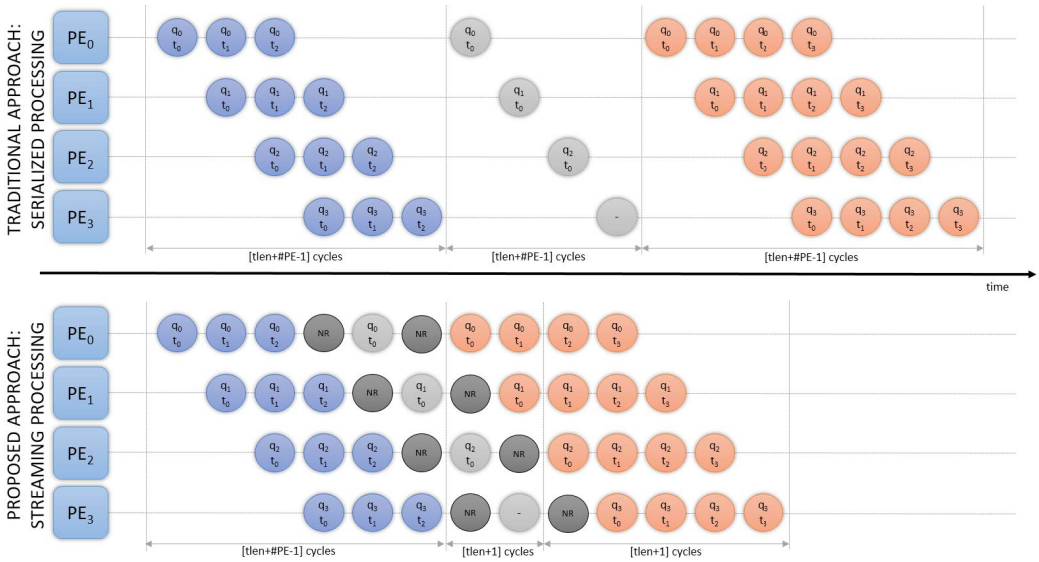


Figure 3. This example illustrates the difference in processing time for Serialized Processing compared to Streaming Processing, with three alignments being performed. A traditional Smith-Waterman systolic array performs serialized processing of the alignments. Each alignment requires [tlen+# of PE-1] cycles. The large amount of white space in the figure is indicative of the fact that large parts of the array are idle during the computation. Streaming processing results in much higher utilization of the systolic array as, except for the first alignment, each alignment only requires [tlen+1] cycles. A New Read token is inserted in-between sequences to signal to a Processing Element that is should proceed onto the next alignment. For ”real” systolic arrays consisting of tens of Processing Elements, the benefits to systolic array utilization are even more pronounced.

Table I
SMITH-WATERMAN KERNEL PERFORMANCE RESULTS

	Design			FPGA Resource Utilization		Performance (GCUPS)		
	Modules	**PEs**	**Frequency**	**Logic**	**RAM Blocks**	**Theoretical**	**Actual**	**Utilization**
BittWare A10PL4	2	131	153 MHz	27%	23%	40.1	40.0	99.8%
BittWare A10PL4	4	131	150 MHz	44%	34%	78.6	78.4	99.8%
BittWare A10PL4	6	131	137 MHz	59%	54%	107.4	107.2	99.8%
Intel A10_REF	2	131	193 MHz	25%	18%	50.5	50.4	99.8%
Intel A10_REF	4	131	188 MHz	41%	40%	98.5	98.3	99.8%
Intel A10_REF	6	131	166 MHz	58%	40%	130.4	130.2	99.8%
Intel A10_REF	8	131	178 MHz	69%	98%	186.5	186.2	99.8%
Intel A10_REF	10	131	164 MHz	91%	98%	214.8	214.4	99.8%
Sirasao[5]	42	32	N/A	N/A	N/A	135.4	77.0	56.9%

V. EXPERIMENTAL SETUP

Results were obtained on a system with an Intel Xeon E5-1650 (six cores, twelve threads @ 3.2 GHz) with 64 GB of RAM. We used a BittWare A10PL4 PCIe board with an Intel Arria 10 GX FPGA (BittWare A10PL4), and the Intel Arria 10 GX FPGA Development Kit Reference Platform board (Intel A10_REF). The only relevant difference between these two boards is that the Intel Reference board uses an Arria 10 GX with higher FPGA fabric speed-grade. Both devices have the same logic density of 1150k logic elements. We used the latest Intel FPGA SDK for OpenCL, which is version 17.0.1 [8]. The results are obtained using a data set of 100'000 pairwise alignment query/target pairs, with a query length of 131 and a target length of 400.

VI. RESULTS

The standardized metric for comparing the performance of Smith-Waterman implementations is by using the GCUPS unit: giga-cell updates per seconds. This number indicates the billions of cell updates that can be performed every second on the Smith-Waterman similarity matrix. The theoretical value can be attained only when all Processing Elements are busy performing useful work. This is not often the case, usually only for very long target sequences. A key innovation of our implementation is that our efficiency is virtually independent of target sequence length. The theoretical maximum GCUPS value is calculated by:

$$\text{Max}_{\text{theo}} = \#\text{ of modules} \times \text{frequency} \times \#\text{ of PEs}$$

The results for the various designs are shown in Table I. We show a number of designs, with increasing number of modules, and for both Arria 10 FPGA boards (A10_REF and A10PL4). The largest 10-module design is able to achieve a theoretical maximum performance of 214 GCUPS. From the results, it is clear that the higher FPGA speed-grade used by the A10_REF board has a significant effect on achievable frequency, improving it by 21-26%. The larger designs show a bit reduced frequency compared to smaller designs, as the FPGA synthesis tool chain has to put more effort into generating a functional design.

We compare our results to the previously highest performing FPGA Smith-Waterman implementation of Sirasao [5]. Sirasao evaluated a variety of designs of which the ratio between number of Processing Elements and number of modules varied. Here, we included only the results for their best performing design, which includes 42 modules of 32 PEs each. Note that the total number of Processing Elements is quite similar to our largest design, with 1344 PEs for Sirasao compared to 1310 for our 10-module design. This 42-module design is able to achieve a theoretical maximum performance of 135 GCUPS. This means that our design has a +58% higher theoretical performance.

However, for traditional Smith-Waterman systolic array designs, the GCUPS value achievable in practice is substantially lower than the theoretical maximum performance. Utilization can be defined as:

$$\text{Utilization} = \frac{(\text{PEs} \times \text{cycles})_{\text{useful}}}{(\text{PEs} \times \text{cycles})_{\text{overall}}}$$

For example, the Sirasao design only achieves 56.9% utilization on their data set, for an actual performance of 77 GCUPS. In contrast, our design achieves almost full utilization, still obtaining 214 GCUPS. Peak performance is only slightly reduced, as for each alignment, one target symbol per alignment is used to indicate a new sequence, thus resulting for our data set with target sequence length 400 in an efficiency of 99.8% (=400/401). In their paper, Sirasao [5] tested with a data set with target length 256 sequences and query length 128, for this the efficiency of our design would be 97.3%. This shows that in practice, our streaming, implicit synchronizing design is almost three times as fast as the fastest previously known implementation.

VII. DISCUSSION

The Streaming architecture significantly improves systolic array utilization. Figure 4 and Figure 5 illustrate how utilization depends on target and query length, respectively.

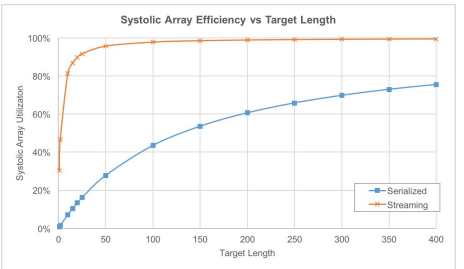


Figure 4. Systolic array efficiency dependence on target length (data set: 100 alignments with query length 131 and variable target length).

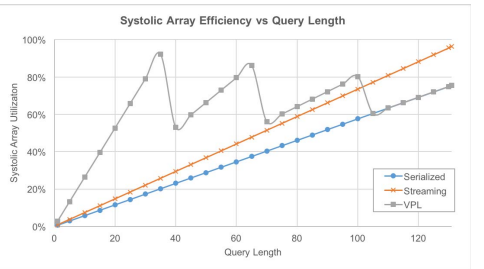


Figure 5. Systolic array efficiency dependence on query length (data set: 100 alignments with variable query length and target length 400).

Whereas efficiency of the Serialized systolic array slowly increases with target length, the Streaming systolic array obtains high efficiency almost immediately. For a data set with so few alignments, the disproportionate long cycle time of the first alignment has a large impact on efficiency; a bigger data set would mask this better. The dependence on query length is similar for both systolic array architectures, showing linear dependence. In [7], various solution are proposed to improve utilization in situations that have an imbalance between systolic array size and query length. For example, the Variable Physical Length (VPL)-systolic array contains multiple sized systolic arrays: alignments with shorter query length go to the smaller arrays. Therefore, the improvements proposed here represent the missing link to achieve a high utilization systolic array.

Compared to using normal hardware description languages, using a high-level language such as OpenCL has two main benefits. First, OpenCL is more expressive (our Processing Element kernel code in OpenCL requires 90 lines of code compared to about 450 lines of VHDL). Second, OpenCL development has more convenient testing and debugging capabilities, such as rapid testing using software emulation, and the ability to use printf statements inside kernels. The end result is a much faster development cycle.

VIII. CONCLUSION

We presented our OpenCL-based FPGA Smith-Waterman implementation that employs two key techniques to greatly improve the utilization of its underlying systolic array architecture. By eliminating centralized control and through the use of Query Buffers, an arbitrary number of alignments can be in flight at the same time, resulting in utilization close to theoretical maximum performance. The techniques presented here are generally applicable to any linear systolic array design, although here we only consider the Smith-Waterman algorithm. Our resulting implementation is both the fastest and most efficient, resulting in a maximum performance of 214 GCUPS and outperforming other Smith-Waterman FPGA implementations almost three-fold.

Compared to typical hardware description languages used for FPGA development, OpenCL simplifies writing code, testing and debugging. The ability to emulate and debug in software allows for a much more agile development cycle, allowing one to test many more different designs.

ACKNOWLEDGMENTS

The authors would like to thank the kind people at Intel and OVH, for providing support on all questions regarding the Intel FPGA SDK for OpenCL, and for providing access to their cloud nodes for development and testing.

REFERENCES

[1] T. Smith and M. Waterman, “Identification of Common Molecular Subsequences,” *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[2] M. Zhao, W. Lee, E. Garrison, and G. Marth, “Ssw library: an simd smith-waterman c/c++ library for use in genomic applications,” *PloS one*, vol. 8, no. 12, p. e82138, 2013.

[3] M. Farrar, “Striped smith–waterman speeds database searches six times over other simd implementations,” *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2006.

[4] Y. Liu, A. Wirawan, and B. Schmidt, “CUDASW++ 3.0: Accelerating Smith-Waterman Protein Database Search by Coupling CPU and GPU SIMD Instructions,” *BMC bioinformatics*, vol. 14, no. 1, p. 117, 2013.

[5] A. Sirasao, E. Delaye, R. Sunkavalli, and S. Neuendorffer, “Fpga based opencl acceleration of genome sequencing software,” *System*, vol. 128, no. 8.7, p. 11, 2015.

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

[7] E. Houtgast, V. Sima, K. Bertels, and 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.

[8] Intel, “The Intel FPGA SDK for Open Computing Language,” https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html, last visited: 2017-08-24.

CHAPTER 4

Power-Efficiency, Design-Time, and Read Length Analysis

Pelorat smiled, "You know, I never considered myself a patriot.
I like to think I recognize only humanity as my nation."

— Isaac Asimov

THE previous two chapters contained a number of improvements to the systolic array architecture that, when combined, result in a maximally-efficient systolic array. However, although efficiency is often a key design metric, others aspects of a design may be equally important. Therefore, in this chapter various other criteria of the proposed implementations are analyzed. How power-efficient are these resulting implementations compared to the non-accelerated software-only implementation? How much time does crafting a heterogeneously-accelerated application require, and how does the required implementation time vary across different programming languages? And, last but not least, what ramifications could expected developments in sequencing hardware have on the implementations described here?

Nowadays, a very important aspect of any design is overall power consumption, and in particular power-efficiency. Mobile devices are greatly constrained by the amount of power they can passively dissipate, and a large part of the total cost of a data center results from active cooling. Therefore, the power-efficiency of the proposed implementations is analyzed, providing a unique insight as some very similar implementations are compared across different hardware architectures. Furthermore, the effects of scaling to more powerful host systems is estimated in order to deduce which system has the overall highest possible efficiency. To ease the effort of programming heterogeneous architectures, over time numerous programming languages have been proposed to facilitate this challenge. These languages differ greatly, amongst others in their abstraction level and in the amount of low-level control they offer. Here, the CUDA, VHDL, and OpenCL implementations are compared to offer a rudimentary idea on how the complexity and design-time varies between these three languages. Finally, developments in the sequencing hardware responsible for generating the raw sequencing data are expected to result in sequencers that output reads that are longer than the currently typical 100 or 150 base pairs, and is expected to grow significantly, which will have a major impact on genomics sequencing pipelines. Here, we investigate its impact to efficiency and performance on the GPU-accelerated BWA-MEM implementation.

4.1. MAIN CONTRIBUTIONS

The main contributions of this chapter are as follows:

- The power-efficiency of the GPU-accelerated and the FPGA-accelerated BWA-MEM implementations is compared to the software-only version. Although both implementations offer a two-fold overall speedup in performance, the FPGA version is also 1.6x as power-efficient when compared to the software version, whereas the GPU-accelerated implementation is not able to offer any efficiency benefit to the system as tested. Extrapolating the results to a more balanced host and accelerator system, the FPGA-accelerated implementation's efficiency further increases to 2.1x, whereas the GPU-accelerated implementation is able to show a 1.4x power-efficiency benefit **[RECONFIG2016]**.

- Comparative analysis of the two BWA-MEM implementations in CUDA and VHDL, and the Smith-Waterman FPGA-implementation in OpenCL shows that there are stark differences in code complexity and design-time. All implementations require vastly more code compared to the software-only baseline implementation. The CUDA and OpenCL implementations require 5-7x as much code, whereas the VHDL implementation requires even 40x as much code. The implementation time is similarly different, with the OpenCL and CUDA implementations being vastly faster to create than the VHDL implementation **[BIBE2018]**.

- An evaluation of the impact of longer read length on the GPU-accelerated BWA-MEM implementation shows that increased read length greatly impacts the ability of the GPU to execute tasks in parallel, emphasizing the need for efficiency improvements to the underlying GPU architecture. The inefficiencies in systolic arrays are categorized in four main classes, and the VPL-based implementation is shown to minimize the negative effects for each of these categories. Load balancing can be used to improve performance by up to 45% when a mismatch exists in the performance of the host and accelerator **[CBC2018]**.

4.2. RESEARCH ARTICLES

This chapter is based on the following articles:

1. **E.J. Houtgast**, V.M. Sima, K.L.M. Bertels, and Z. Al-Ars, *Power-Efficiency Analysis of Accelerated BWA-MEM Implementations on Heterogeneous Computing Platforms*, International Conference on Reconfigurable Computing and FPGAs (ReConFig), December 2016, Cancun, Mexico.

2. **E.J. Houtgast**, V.M. Sima, K.L.M. Bertels, and Z. Al-Ars, *Comparative Analysis of System-Level Acceleration Techniques in Bioinformatics: A Case Study of Accelerating the Smith-Waterman Algorithm for BWA-MEM*, 18th International Conference on Bioinformatics and Bioengineering (BIBE), Oct 2018, Taichung, Taiwan.

3. **E.J. Houtgast**, V.M. Sima, K.L.M. Bertels, and Z. Al-Ars, *Hardware Acceleration of BWA-MEM Genomic Short Read Mapping for Longer Read Lengths*, Computational Biology and Chemistry 75, p54-64, 2018.

Power-Efficiency Analysis of Accelerated BWA-MEM Implementations on Heterogeneous Computing Platforms

Ernst Joachim Houtgast*†, Vlad-Mihai Sima†, Giacomo Marchiori†, Koen Bertels* and Zaid Al-Ars*
*Computer Engineering Lab, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands
†Bluebee, Laan van Zuid-Hoorn 57, 2289 DC Rijswijk, The Netherlands
E-mail: {ernst.houtgast, vlad.sima, giacomo.marchiori}@bluebee.com, {k.l.m.bertels, z.al-ars}@tudelft.nl

***Abstract*—Next Generation Sequencing techniques have dramatically reduced the cost of sequencing genetic material, resulting in huge amounts of data being sequenced. The processing of this data poses huge challenges, both from a performance perspective, as well as from a power-efficiency perspective. Heterogeneous computing can help on both fronts, by enabling more performant and more power-efficient solutions.**

In this paper, power-efficiency of the BWA-MEM algorithm, a popular tool for genomic data mapping, is studied on two heterogeneous architectures. The performance and power-efficiency of an FPGA-based implementation using a single Xilinx Virtex-7 FPGA on the Alpha Data add-in card is compared to a GPU-based implementation using an NVIDIA GeForce GTX 970 and against the software-only baseline system. By offloading the Seed Extension phase on an accelerator, both implementations are able to achieve a two-fold speedup in overall application-level performance over the software-only implementation. Moreover, the highly customizable nature of the FPGA results in much higher power-efficiency, as the FPGA power consumption is less than one fourth of that of the GPU. To facilitate platform and tool-agnostic comparisons, the base pairs per Joule unit is introduced as a measure of power-efficiency. The FPGA design is able to map up to 44 thousand base pairs per Joule, a 2.1x gain in power-efficiency as compared to the software-only baseline.

***Keywords*- FPGA; GPU; Next Generation Sequencing; power-efficiency; read mapping**

I. INTRODUCTION

Next Generation Sequencing (NGS) techniques dramatically decrease the cost of sequencing genetic material. The cost to sequence one complete human genome is rapidly approaching the important $1,000 mark [1]. As a result of these falling costs, the production of genetic data is surging and is projected to rival, if not overtake, other Big Data fields such as streaming video services and astronomy [2]. A single run on a state-of-the-art X Ten sequencing machine [3] can generate up to 1.2 TB of data, which in turn requires multiple days to process, even on a large computing cluster. The extreme scale of data and tremendous computing efforts involved in processing this data necessitates the use of high performance computing solutions to face this challenge. Heterogeneous computing, and in particular reconfigurable computing, offers a large promise as a solution that enables both high performance and power-efficiency compared to traditional computing techniques. Power-efficiency is becoming at least as important as raw performance, as power consumption is an important driver to overall data center cost.

NGS data is typically processed by a complex pipeline of algorithms. In the case of DNA NGS data, the short reads as produced by the sequencer are first mapped onto a reference genome. Then, this output is sorted and duplicates are marked. Finally, mutations in the genetic material as compared to the reference are found during a variant calling stage. Only at this stage does the raw data become usable for further downstream analysis, for example by researchers or medical professionals. To illustrate the immense computational requirements, processing such a data set can easily require multiple days, even on a large cluster.

BWA-MEM is a Burrows-Wheeler Alignment based tool for mapping short reads onto a reference genome [4]. Although many other alignment tools exist (examples include [5], [6]), BWA-MEM is the de facto standard for alignment mapping and is part of the popular BWA-MEM/GATK pipeline, used in organizations around the world [7]. In the example above, BWA-MEM contributes about 36% to the overall processing time, making up a significant portion of the processing time of the entire pipeline. Therefore, it is an important target for acceleration to reduce the overall time, cost and energy of processing NGS data sets.

Similar to other mapping tools, such as [5], BWA-MEM operates using the Seed-and-Extend paradigm (see Figure 1). For each read, seeds, exactly matching subsequences between the read and the reference, are generated. Subsequently, each seed is extended in both directions using an inexact matching algorithm, similar to the popular Smith-Waterman dynamic programming algorithm [8]. The highest scoring extended seed is chosen as final alignment. The Seed Extension phase forms a major bottleneck in the BWA-MEM algorithm, requiring between 30%-50% of total execution time, depending on the computing platform [9], [10]. For example, on the highly multithreaded IBM Power8 platform, the Seed Extension phase requires almost 50% of overall execution time, allowing for an up to two-fold performance improvement, whereas on the Intel Core i7 platform, a performance improvement of up to 1.7x is possible. In this paper, accelerated implementations of the BWA-MEM Seed Extension phase on the Intel Core i7

978-1-5090-3707-0/16/$31.00 ©2016 IEEE

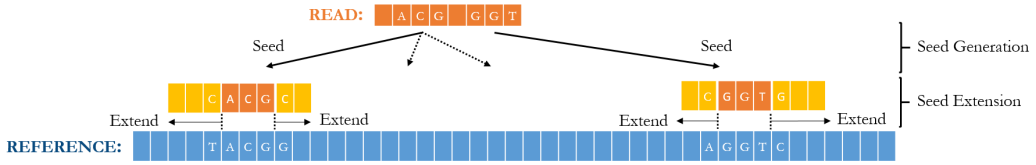


Fig. 1. BWA-MEM processes reads using the Seed-and-Extend paradigm: for each read, likely mapping locations on the reference are found by searching for exactly matching subsequences between the read and the reference (*seeds*). Then, these seeds are extended in both directions using a Smith-Waterman-like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

platform are considered. However, the main idea is generally applicable to other mapping applications as well.

The contributions in this paper include:

- A comparison of both performance and power-efficiency of an FPGA-accelerated implementation to software-only and GPU-based implementations.
- The introduction of the base pairs per Joule (bp/J) unit as measure of power-efficiency, allowing for platform-agnostic comparison of system-level power-efficiency on genomic data sets.

The remainder of this paper is organized as follows. Related work is discussed in Section II. Architectural details of the FPGA and GPU implementations are provided in Section III. The results on performance, scalability and power-efficiency are given in Section IV. Then follows a discussion of the results in Section V. The conclusions are given in Section VI.

II. Related Work

A major part of BWA-MEM execution time is spent in the Seed Extension phase. The inexact mapping algorithm used is similar to the Smith-Waterman algorithm. This algorithm has received much acceleration attention (e.g., [11], [12]). However, for multiple reasons most implementation ideas are not directly applicable to BWA-MEM, the main ones being:

- The highest speedup is generally obtained when performing mapping of very long sequences that contain thousands of bases, whereas NGS reads and BWA-MEM are more focused towards the mapping of short reads of at most a few hundred base pairs.
- For load balancing purposes, these implementations batch alignments of similar length together. In the case of BWA-MEM, this is impractical as the inexact mapping calls are dynamically generated for alignments of varying length, which makes the batching strategy inefficient due to the large communication and temporary data overheads this would require.

Although many accelerated Seed-and-Extend based mapping tools have been proposed (for example [13]), results from these implementations are not directly comparable. In bioinformatics, exactness of results is critical, as larger population studies can take several years to complete and intermediate results need to be comparable. Even a change in version is often unacceptable. Note that BWA-ALN, for which accelerated implementations do exist, is a different algorithm.

A kernel-level acceleration effort specific to the BWA-MEM algorithm is reported in [14], where one of the BWA-MEM kernels, the inexact matching phase, has been accelerated on an FPGA for an up to 26x kernel-level speedup. However, due to the above mentioned reasons, only the kernel-level speedup is reported, and no overall application-level speedup is mentioned. In our experience, accommodating the kernel implementation into a full application is far from a trivial task. The authors acknowledge that the significantly varied input data would pose a challenge for the Smith-Waterman algorithm, but disregard the additional challenges a full-application implementation needs to face.

To our knowledge the only application-level accelerated integrated implementations of BWA-MEM that exist are: an FPGA-accelerated implementation of the Seed Extension phase [15] achieving a 1.5x speedup, further improved in [16] for an overall 2.6x speedup; and a GPU implementation [9], further improved to achieve an up to 2x speedup [17]. The FPGA implementation used here builds on [15], and a comparison of the implementation here is made to the improved GPU implementation. This paper focuses on power-efficiency, besides overall application performance, as for many scenarios, such as processing in a large scale data center, this is at least as important as absolute performance.

III. Architecture Design and Implementation

In this section, first the BWA-MEM algorithm is briefly described, along with the features that both the FPGA and GPU implementations share. Then, the details of the FPGA-accelerated implementation on the Alpha Data card are given. Finally, details of the GPU implementation are briefly discussed (further details can be found in [17]).

The original BWA-MEM algorithm operates in a serial fashion (refer to Figure 2). The input is processed in batches of reads, that are processed one-by-one by two major kernels: Seed Generation, where seeds (exactly matching subsequences) are generated for each read, and Seed Extension, where the generated seeds are extended allowing for inexact matches. This process repeats itself until the input is exhausted. These phases take full advantage of multithreading, as reads are completely independent from each other and can be processed in parallel. To improve the utilization of system resources when using an accelerator, the BWA-MEM algorithm has been reorganized into a fully pipelined organization.

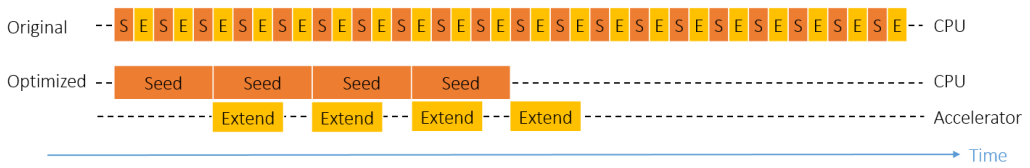


Fig. 2. BWA-MEM processes the input reads one-by-one. For each read, the seeds are generated and then, these seeds are extended. To reduce communication overhead, the accelerated implementations described here use an optimized program architecture, whereby work of multiple reads is batched together. First, seeding is performed for a large group of reads. Then, such a batch is offloaded onto an accelerator that performs the Seed Extension. Its execution is overlapped with work on the CPU. As long as the Seed Extension phase requires less time on the accelerator than the Seeding on the CPU, its execution time is effectively "hidden".

Multiple reads are processed in groups, and the two phases are executed in parallel and are overlapped. Thus, Seed Generation executes simultaneously on the CPU with Seed Extension on the accelerator, resulting in a large performance improvement.

A. FPGA Design and Implementation

The FPGA-accelerated BWA-MEM offloads the Seed Extension phase onto hardware (refer to Figures 1 and 2). After seeds have been generated, they are transferred to the FPGA on-board memory. Typically, a single read will result in multiple seed locations on the reference genome being found. Seeds that are mapped close together on the reference genome are grouped into chains. Seeds are processed one-by-one. Seed Extension for a seed can be skipped depending on the result of previous extensions. On average only one seed per chain requires extension. Moreover, the length of the extension to be performed is dependent on the part of the read that forms the seed. This interdependence between executions makes this phase unsuitable for streaming and, therefore, the FPGA performs both this control logic, as well as the inexact mapping algorithm itself. The resulting mappings are stored in on-board memory and transferred back to the host system.

The Seed Extension module design is based on the design in [15]. The inexact mapping is similar to the Smith-Waterman algorithm. To find the optimal mapping, a 2D similarity matrix is filled. This is implemented as a systolic array, with anti-diagonals of the matrix being calculated in parallel. Each cycle, one processing element (or PE) of the systolic array computes a value of the 2D similarity matrix. Hence, the execution time is reduced from O(M $\times$ N) to O(M+N), where M and N are the length of the reference and the read. This is the reason why certain Smith-Waterman implementations, at least for longer alignments, are able to achieve speedups of several magnitude. In this paper, only short reads of up to 150 base pairs are considered, which is the read length of contemporary sequencers, such as the Illumina HiSeq X. The minimum seed length for BWA-MEM is 19 symbols. Therefore, each inexact mapping engine contains 131 PEs. To improve utilization for shorter reads, *early exit points* as described in [15] are implemented.

The implementation described here uses an Alpha Data add-in card with a single Xilinx Virtex-7 FPGA (details can be found in Section IV). A floorplan of the design is shown in Figure 3. The design contains six modules that are able to process the Seed Extension phase. Within each module, the larger block contains the systolic array logic, the smaller block is filled with control logic. Between the six modules resides the logic that distributes reads over the modules and is responsible for I/O. The rest of the area is taken up by interconnection-related logic, such as the PCI-Express interface and the memory controller.

A significant difference to the design in [15] and [16] is the fact that the Alpha Data card used here contains only a single Virtex-7 FPGA, whereas [15] and [16] use the Convey HC-2^{EX} as implementation platform, which contains four user-configurable Virtex-6 FPGAs. As the design here is limited by the amount of LUTs available, and the Virtex-7 FPGA on the Alpha Data card contains 432,368 LUTs versus 474,240 LUTs per Virtex-6 FPGA on the Convey, this means only about 23% of the resources are available as compared to the Convey platform. This, in turn, requires a careful selection to place only those modules on the FPGA that most benefit from acceleration. Hence, the decision was made to only implement the Seed Extension phase in hardware. In total, about 71% of all LUTs is utilized and the Seed Extension modules run at a clock rate of 160 MHz.

B. GPU Implementation

Similar to the FPGA implementation, the GPU implementation, further described in [17], also accelerates the Seed Extension phase. However, in contrast to the FPGA implementation, which contains six Seed Extension modules, the available execution resources on the GPU depend on the actual GPU model being used. A batch of reads is sent to the GPU as a grid of thread blocks, where each read is mapped onto a single block of 32 threads. This ensures scheduling is automatically taken care of by the GPU, with per-read granularity, based on the available execution resources. In the case of the test platform, an NVIDIA GeForce GTX 970, there are thirteen multiprocessors available. Up to 32 thread blocks can be active per multiprocessor at any single time, due to limits on the amount of registers and shared memory available.

Similar to the FPGA implementation, the Seed Extension phase is logically split between the control logic, which loops over all the seeds of a single read, and the actual inexact mapping. This control logic code is executed by a single

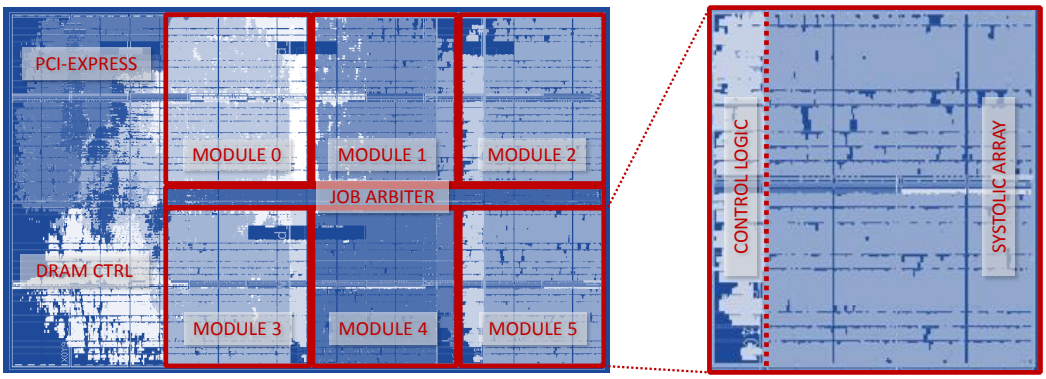


Fig. 3. Floorplan of the FPGA design with six Seed Extension modules. Each module contains control logic to loop over all seeds to be extended, and the Seed Extension systolic array with 131 Processing Elements (see inset). In between the Seed Extension modules resides the arbitrating logic. The remaining area is filled with interconnection-related logic, such as the PCI-Express controller and the DRAM controller.

thread, whereas the inexact mapping code is performed in a systolic array-like manner using 32-PEs at a time. In this case, each CUDA thread acts as a separate PE. Hence, longer extensions require that multiple passes are made over the similarity matrix. The choice to use only 32-threads at a time allows for the use of the intra-warp shuffle functionality, which allows CUDA threads to access each other's registers, to implement data transfer between the PEs, eliminating the need of temporary storage of row data while calculating the similarity matrix elements.

IV. Experimental Results

Tests have been performed using a system with an Intel Core i7-4790 at 3.6 GHz with eight logical cores (four physical cores), SpeedStep and Hyper-Threading enabled. The system contains 16 GB of RAM. Tests were run using CentOS 7.1. To minimize power consumption, no unnecessary services were running. In addition, for the FPGA-accelerated tests, an Alpha Data ADM-PCIE-7V3 card with a Xilinx Virtex-7 XC7VX690T-2 and 16 GB of on-board RAM was added to the system [18]. The Seed Extension modules run at a clock rate of 160 MHz. For the GPU-accelerated tests, 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 was added to the system. The GPU is allowed to go into lower power states when idle to conserve power. The software-only results were gathered without either of these cards installed. An emonPi energy measurement unit from OpenEnergyMonitor [19] was used to measure idle and load power utilization of the entire system under test. The current probe was connected directly to the mains power cord to measure system level power consumption, taking hundreds of samples per second.

BWA-MEM version 0.7.8 was used. Tests were performed using publicly available data from the Genome Comparison & Analytic Testing (GCAT) framework [20]. The single-ended alignment (150bp-se-small-indel) and pair-ended alignment (150bp-pe-large-indel) data sets were used. Each data set contains about eight million reads of 150 base pairs, or about 1.2 billion base pairs in total. The reads were aligned against the reference human genome (UCSC HG19). The GCAT online sequence alignment quality comparison service was used to verify that results of the FPGA-accelerated and the GPU-accelerated versions are indistinguishable from those obtained with the software BWA-MEM algorithm.

In the remainder of this section, performance results on all three platform are shown for the single-ended and pair-ended GCAT tests. The scalability of the heterogeneous platforms is investigated to find the optimal balance between host CPU count and accelerator performance. Finally, the power-efficiency of the various platforms is determined.

A. Performance Analysis

The performance, scalability and power-efficiency results have been gathered using the GCAT data sets. Each contains about eight million reads with about 1.2 billion base pairs, for a file size of about 3 GB for the single-ended data set and 2x 1.5 GB for the pair-ended data set. A typical complete Illumina X Ten sequencing run generates a data set about 400x larger, or approximately 1.2 TB. The implementation has been verified to scale up to work without issues on such larger data sets, but, for reasons of practicality, tests have been performed on the smaller data set.

Performance results on the single-ended and pair-ended GCAT data sets are summarized in Table I. The execution time is shown both as time required for the Seed Extension phase on the accelerator hardware, and for the overall application wall clock time. To facilitate cross-platform comparisons, the results are converted into throughput in millions of base pairs per second. Both the FPGA and GPU implementations are able to achieve a two-fold improvement to performance, the FPGA implementation being slightly faster.

TABLE I
EXECUTION TIME AND SPEEDUP FOR THE GCAT ALIGNMENT QUALITY BENCHMARK

		Seed Extension Phase		Overall Application		
Test	**Platform**	**Execution Time**	**Speedup**	**Execution Time**	**Speedup**	**Throughput**
Single-Ended	Software-Only	237 s	-	552 s	-	2.2 Mbp/s
Data	FPGA-Accelerated	129 s	1.8x	272 s	2.0x	4.5 Mbp/s
	GPU-Accelerated	144 s	1.6x	278 s	2.0x	4.3 Mbp/s
Pair-Ended	Software-Only	246 s	-	572 s	-	2.1 Mbp/s
Data	FPGA-Accelerated	130 s	1.9x	289 s	2.0x	4.1 Mbp/s
	GPU-Accelerated	141 s	1.7x	293 s	2.0x	4.1 Mbp/s

On a kernel-level, the Seed Extension phase itself is up to 1.9x faster on the FPGA and up to 1.7x faster on the GPU, as compared to software-only execution. This is equivalent to a speed of about 15 and 14 logical Intel Core i7 cores, respectively. Since the execution time of the accelerated Seed Extension implementations only requires about half the overall application execution time, it is clear that in both cases, the accelerator is not fully utilized. Therefore, both the FPGA and GPU accelerators can be used to accelerate a system that is more powerful than the system as tested here, which only contains eight logical processor cores. This is explored in the next section.

B. Scalability Analysis

From the differences in execution time between the Seed Extension phase and the overall application, as described in Table I, it is clear that the FPGA and GPU implementations are not fully utilized. The time spent by the accelerators in the Seed Extension phase is only about half the overall application execution time. This implies that the accelerator hardware is not busy 100% of the time. Therefore, a faster host system would still be able to be accelerated for the maximum speedup of 2x. In contrast, if the accelerated implementations could not keep up with the host, overall speedup would fall below 2x. In this case, more Alpha Data cards, or more and/or faster GPUs could be used. Hence, the objective is to design a system for which all system resources (the CPU cores and either an FPGA or GPU-accelerator) are fully utilized.

In Table II, the estimated *scalability* of the accelerated platforms is shown, which is expressed in number of logical CPU cores for which the accelerator is able to provide the maximum two-fold speedup. In order to estimate the optimal ratio of logical CPU cores for each accelerated platforms, the following assumptions have been made. It is assumed that overall application times decreases linearly with additional CPU cores. Furthermore, it is assumed that the maximum speedup is achieved as long as the time required for the Seed Extension phase does not exceed overall application time.

TABLE II
SCALABILITY OF THE ACCELERATED IMPLEMENTATIONS

	Execution Time			
Platform	**Seed Ext.**	**Overall**	**Utilization**	**Scalability**
FPGA (4 modules)	171 s	272 s	63%	12.7 cores
FPGA (5 modules)	146 s	275 s	53%	15.1 cores
FPGA (6 modules)	129 s	272 s	47%	16.8 cores
GPU	144 s	278 s	52%	15.4 cores

The scalability results are presented for three different FPGA designs. These designs vary in the number of Seed Extension modules that were placed onto the FPGA logic. Although the time required for the Seed Extension phase decreases significantly when more modules are available, this proves to only have a slight impact on the overall execution time. The reason for this is the fact that Seed Extension performance is already fast enough. In contrast, scalability results are much improved. The performance and power-efficiency results in the other sections all consider the six module FPGA design. This six module design is able to support a host system with up to sixteen logical Intel Core i7 cores at 3.6 GHz, for example the Intel Core i7-5960X. The GPU implementation is able to support a host system with up to fifteen logical cores.

C. Power-Efficiency Analysis

To measure the power-efficiency of the different platforms, an emonPi energy monitor was used to track the system-level power consumption as measured at the power plug. For the power-efficiency tests, only the single-ended data set was used, although the pair-ended data set should yield similar results, given that the execution profile is similar. In order to minimize the idle power draw, the tests were performed without active GUI, and with a bare system with only an HDD, SSD and optical drive present. In case of the accelerated platforms, the respective accelerator card was added to the system.

To present a quick qualitative overview of the results, an example trace of the power consumption for a single test is shown in Figure 4. As expected, the accelerated platforms use more instantaneous power than the software-only system, both under load and when idle. However, they are also able to map reads at a much higher rate, and both finish in about half the time of the software-only implementation. The results are summarized in Table III. This table gives the power consumption, performance and energy efficiency for each platform. Both the measured data is presented, as well

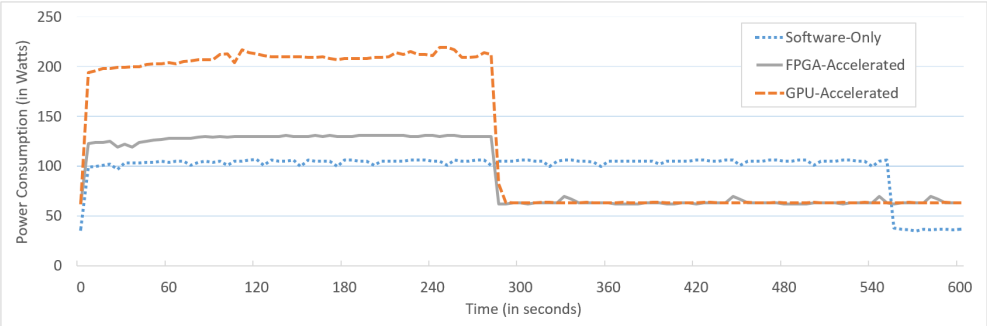


Fig. 4. Example trace of the power consumption over time for the software-only, FPGA-accelerated and GPU-accelerated platforms. The software-only system consumes the least power, both when idle and under load. However, the additional power utilized by the accelerated platforms results in much faster completion of the test, as both implementations finish in much less than three hundred seconds. The FPGA-accelerated platform has the highest power-efficiency.

TABLE III
POWER CONSUMPTION, PERFORMANCE AND POWER-EFFICIENCY FOR SOFTWARE-ONLY AND ACCELERATED PLATFORMS.

			Power Consumption		Performance		Energy Efficiency		
Test	**Platform**	**Logical Cores**	**Idle Power**	**Load Power**	**Execution Time**	**Application Speedup**	**Total Energy**	**Base Pairs Per Energy**	**Efficiency Improvement**
Measured Data	Software-Only	8	37 W	105 W	552 s	-	58 kJ	20.7 kbp/J	-
	FPGA-Accelerated	8	62 W	129 W	272 s	**2.0x**	35 kJ	34.1 kbp/J	**1.6x**
	GPU-Accelerated	8	63 W	210 W	278 s	**2.0x**	58 kJ	20.5 kbp/J	**1.0x**
Estimated Data	Software-Only	15	37 W	164 W	294 s	-	48 kJ	24.8 kbp/J	**1.2x**
	FPGA-Accelerated	15	62 W	188 W	147 s	**2.0x**	28 kJ	43.2 kbp/J	**2.1x**
	GPU-Accelerated	15	63 W	269 W	148 s	**2.0x**	40 kJ	30.0 kbp/J	**1.4x**
	Software-Only	16	37 W	172 W	276 s	-	47 kJ	25.2 kbp/J	**1.2x**
	FPGA-Accelerated	16	62 W	196 W	138 s	**2.0x**	27 kJ	44.1 kbp/J	**2.1x**
	GPU-Accelerated	16	63 W	277 W	144 s	**1.9x**	40 kJ	29.9 kbp/J	**1.4x**

as results for estimated, more well-balanced, systems with a larger number of logical CPU cores. First, the results for the measured data are discussed, afterwards the estimated data.

Similar to the trace shown in Figure 4, as expected the power draw of the software-only system is the lowest, both when idle at 37 W and under load at 105 W. The accelerated platforms show significantly increased idle power consumption, at 62 W for the FPGA platform and at 63 W for the GPU platform, due to the addition of more hardware into the system. Also, both platforms show a higher power consumption under load, at 129 W for the FPGA platform and at 210 W for the GPU platform. However, this is compensated by much improved performance, as each implementation is able to achieve a two-fold speedup as compared to software-only execution. In addition, the FPGA-accelerated implementation is much more power-efficient as compared to the other two platforms: whereas the software-only and GPU-accelerated platforms both require 58 kJ to complete the entire test, the FPGA-accelerated platform requires only 35 kJ, an energy efficiency improvement of 60%. On this system, use of a GPU-accelerator merely provides the user a trade-off between performance and power consumption, as overall power-efficiency remains the same as compared to the software-only platform.

To put the relative power efficiency of the FPGA and the GPU into perspective, consider their respective power consumption when executing the same workload. The difference in load and idle power on the software-only platform implies that the host CPU uses about 68 W of power under load. If it is assumed that the host CPU requires a similar power draw when running the accelerated platforms, then the power consumption under load used exclusively for powering the accelerators can be computed, being 25 W for the FPGA and 105 W for the GPU. Hence, the FPGA power consumption is less than one fourth of the GPU, showing the clear advantage in energy efficiency of the reconfigurable platform. Moreover, the FPGA does not seem to consume any additional power under load, as its logic is always active.

To facilitate comparisons of power-efficiency for mapping a certain data set of reads, the data is also reported as *number of base pairs mapped per Joule of energy*. Using this measure, the power-efficiency for a given data set can be evaluated across platforms, architectures and mapping tools. However, it is important to note that performance of the various tools is not the only measure of interest, as mapping tools can differ greatly in their mapping quality, which is the ability to accurately map reads. As the data set used in these

tests contains about eight million reads, each with 150 base pairs, the resulting power-efficiency for the FPGA-accelerated platform is about 34 kbp/J, a 60% improvement in energy efficiency as compared to the software-only platform.

Based on the scalability results obtained in the previous section, Table III also contains the estimated performance and power-efficiency for two more well-balanced systems that contain more than eight logical CPU cores. The estimated optimum configurations are shown for both the FPGA and the GPU platforms. Note that the optimum configuration for power-efficiency does not necessary need to be able to obtain exactly the maximum two-fold speedup. The following assumptions have been used for the estimation:

- For each platform, system idle power draw remains identical to the measured platform, as no additional idle power draw is assumed for the additional CPU cores (due to perfect power gating of CPU cores);
- Additional power under load required by the additional CPU cores is scaled linearly based on the difference between software-only idle and load power consumption;
- The accelerators do not require additional power for the scaled system, as their workload stays identical;
- Overall application performance scales linearly in CPU core count.

All platforms show a higher energy efficiency as compared to the base eight logical CPU core system. The faster execution results in a lower overall execution time, in turn requiring less power draw of the base system components. Conversely, a system with fewer cores would show lowered power-efficiency. The FPGA-accelerated platform is able to achieve an up to 2.1x improvement in power-efficiency, as compared to the eight core software-only platform, and is able to map up to 44 kbp/J. The GPU-accelerated platform obtains a 46% power-efficiency improvement on the more well-balanced system.

Note that the above results use a conservative estimate for the power-efficiency of the accelerated platforms. These platforms are the most efficient while fully utilized: only in such a situation no unnecessary idle power loss is accumulated while the accelerator is being idle. However, this is not the case for the system on which the results have been gathered, as in that system, the accelerators were idle about half of the overall execution time. Hence, the measured power consumption includes this idle power loss, which means that the results as presented here are a conservative estimate of the power-efficiency for both platforms. In practice, the power-efficiency of a more well-balanced system should be even higher.

To illustrate the dependency of the power-efficiency on the number of logical CPU cores in use, Figure 5 shows the estimated normalized power-efficiency for a system with varying number of cores, compared to the base eight core software-only platform. Under all circumstances, the FPGA platform is the most power-efficient. Both accelerated systems show peak efficiency for a system with about sixteen cores. Hence, a system with an Intel Core i7-5960X processor would be a good matchup. A system with less cores is hampered by under-utilization of the accelerator, whereas for a system with more cores, the accelerator will not be able to keep up with the CPU cores, which, as a result, will be partially idle. In such a situation, load balancing between host and accelerator, as in [9], could improve system resource utilization, resulting in a more graceful drop-off in performance and power-efficiency.

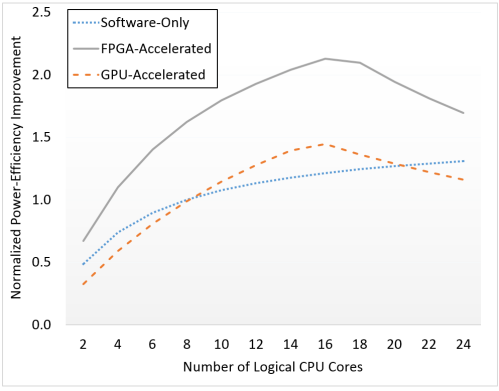


Fig. 5. Estimated power-efficiency for a system with varying CPU core count. The FPGA-accelerated platform is the most power-efficient.

V. DISCUSSION

Accelerate an algorithm such as BWA-MEM, which is characterized by the fact that it contains multiple performance-critical kernels, is always challenging. Hence, as per Amdahl's law, acceleration of a single kernel can only yield limited overall application speedup. In the case of BWA-MEM on the Intel Core i7 platforms, a maximum speedup of 1.7x is implied. Even so, the FPGA and GPU implementations manage to exceed this maximum speedup by at the same time implementing a pipelined program organization, thus achieving a two-fold improvement to execution time. Platforms where the Seed Extension phase consumes a larger part of total application execution time should be able to see an even larger performance increase.

Both the FPGA and GPU-accelerated platforms manage to obtain a two-fold performance improvement. However, it is interesting to compare the nature of both architectures further. The fixed function units on the GPU are able to process large numbers of reads in parallel at a high clock frequency of up to 1.25 GHz. However, a large number of instructions is required to perform a single systolic array cell update, and a large amount of logic is required to provide the massive threading and parallelism. In contrast, the much lower clock frequency and highly customizable nature of the FPGA allows for much more power-efficient processing, resulting in the fact that the FPGA requires less than one fourth of the power consumption of the GPU, while performing the same workload.

The highest power-efficiency of any system including accelerator hardware will only be obtained through a careful balance of system components. Utilizing an accelerator that

is overpowered compared to the rest of the system reduces power-efficiency, whereas a well-balanced system can provide much improved power-efficiency. As shown here, the results for an optimally-balanced system are much better compared to the baseline system. A further improvement to power-efficiency of 29% for the FPGA platform was obtained, and even a 46% improvement to power-efficiency for the GPU platform. This balance of components is especially important when considering cloud-based solutions for data processing, as the available options that can be selected, in particular when accelerators are involved, are restricted to those offered by the cloud provider. The optimal combination may not be always available. It is an ongoing effort to monitor the best available options, making the optimal trade-off between power and power-efficiency.

VI. CONCLUSIONS

In this paper, an accelerated implementation of the BWA-MEM algorithm is introduced that targets the Alpha Data add-in card with a single Xilinx Virtex-7 FPGA. This design is able to provide a two-fold improvement to overall application performance by offloading the Seed Extension phase onto the FPGA and through better pipelining of the application. Scalability analysis shows that the current design is able to provide this maximum two-fold gain in performance for a system with up to sixteen logical CPU cores.

To facilitate platform-agnostic comparison of power-efficiency on mapping genomic data sets, the base pairs per Joule measure is proposed as a unit to express platform power-efficiency. The performance and power-efficiency is compared to a GPU-based implementation, which is also able to obtain a two-fold performance improvement. However, the FPGA implementation is much more power-efficient and can map 34,000 base pairs per Joule of energy, an improvement of 60% compared to the software only and GPU-platforms. Design space exploration shows that a more well-balanced platform, designed to fully utilize the accelerator's potential, can provide an up to 2.1x improvement in power-efficiency, providing a mapping power-efficiency of up to 44,000 base pairs per Joule. Due to the highly customizable nature of the FPGA, its power-efficiency is much higher compared to the GPU and software-only platforms. The FPGA consumes less than one fourth of the power the GPU requires when executing the same workload, showing the large benefit of customizable hardware, as compared to more fixed function hardware.

The authors expect to see increased use of FPGA hardware in the bioinformatics context, as the improvement in speed and power-efficiency will help to reduce the bottleneck of an important part of the widely used BWA-MEM/GATK pipeline. More generally, it offers a large promise of power-efficiency gains for the often extremely large computational challenges in this domain.

REFERENCES

[1] National Human Genome Research Institure, "DNA Sequencing Costs," http://www.genome.gov/sequencingcosts/, accessed: 2015-12-22.

[2] Z. Stephens, S. Lee, F. Faghri, R. Campbell, C. Zhai, M. Efron, R. Iyer, M. Schatz, S. Sinha, and G. Robinson, "Big Data: Astronomical or Genomical?" *PLoS Biology*, vol. 13, no. 7, 2015.

[3] 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.

[4] H. Li, "Aligning Sequence Reads, Clone Sequences and Assembly Contigs with BWA-MEM," *arXiv preprint arXiv:1303.3997*, 2013.

[5] B. Langmead and S. L. Salzberg, "Fast Gapped-Read Alignment with Bowtie 2," *Nature methods*, vol. 9, no. 4, pp. 357–359, 2012.

[6] C.-M. Liu, T. Wong, E. Wu, R. Luo, S.-M. Yiu, Y. Li, B. Wang, C. Yu, X. Chu, K. Zhao, and R. Li, "SOAP3: Ultra-Fast GPU-Based Parallel Alignment Tool for Short Reads," *Bioinformatics*, vol. 28, no. 6, pp. 878–879, 2012.

[7] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, and M. DePristo, "The Genome Analysis Toolkit: a MapReduce Framework for Analyzing Next-Generation DNA Sequencing Data," *Genome research*, vol. 20, no. 9, pp. 1297–1303, 2010.

[8] T. Smith and M. Waterman, "Identification of Common Molecular Subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[9] E. Houtgast, V. Sima, K. Bertels, and 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.

[10] M. J. Jaspers, "Acceleration of Read Alignment with Coherent Attached FPGA Coprocessors," Master's thesis, TU Delft, Delft University of Technology, 2015.

[11] P. Zhang, G. Tan, and G. R. Gao, "Implementation of the Smith-Waterman Algorithm on a Reconfigurable Supercomputing Platform," in *Proceedings of the 1st international workshop on High-performance reconfi- gurable computing technology and applications: held in conjunction with SC07*. ACM, 2007, pp. 39–48.

[12] T. Oliver, B. Schmidt, and D. Maskell, "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*. ACM, 2005, pp. 229–237.

[13] J. Arram, K. H. Tsoi, W. Luk, and P. Jiang, "Reconfigurable Acceleration of Short Read Mapping," in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*. IEEE, 2013, pp. 210–217.

[14] Y.-T. Chen, J. Cong, J. Lei, and P. Wei, "A Novel High-Throughput Acceleration Engine for Read Alignment," in *Field-Programmable Custom Computing Machines, 2015. FCCM 2015. 23rd Annual IEEE Symposium on*. IEEE, 2015.

[15] E. Houtgast, V. Sima, K. Bertels, and 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.

[16] N. Ahmed, V. Sima, E. Houtgast, K. Bertels, and 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*, ser. ICCAD, 2015.

[17] E. Houtgast, V. Sima, K. Bertels, and Z. Al-Ars, "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, July 2016.

[18] 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.

[19] The OpenEnergyMonitor Project, "OpenEnergyMonitor Website," http://openenergymonitor.org/, accessed: 2015-12-14.

[20] G. Highnam, J. J. Wang, D. Kusler, J. Zook, V. Vijayan, N. Leibovich, and D. Mittelman, "An Analytical Framework for Optimizing Variant Discovery from Personal Genomes," *Nature comm.*, vol. 6, 2015.

2018 IEEE 18th International Conference on Bioinformatics and Bioengineering

Comparative Analysis of System-Level Acceleration Techniques in Bioinformatics: A Case Study of Accelerating the Smith-Waterman Algorithm for BWA-MEM

Ernst Joachim Houtgast*†, Vlad-Mihai Sima†, Koen Bertels* and Zaid Al-Ars*
*Quantum&Computer Engineering, Delft University of Technology, Delft, The Netherlands
†Bluebee Research&Development, Bluebee, Rijswijk, The Netherlands
Corresponding e-mail address: e.j.houtgast@tudelft.nl

***Abstract*—Bioinformatics workloads are characterized by huge data sets and complex algorithms, requiring enormous data processing and making high performance heterogeneous computation platforms such as FPGAs and GPUs highly relevant. We compare three accelerated implementations of the widely used BWA-MEM genomic mapping tool as a case study on design-time optimization for heterogeneous architectures: BWA-MEM-CUDA, BWA-MEM-OpenCL, and BWA-MEM-VHDL, each using an optimized Smith-Waterman algorithm implementation. Optimization of design-time is important because of the significant development effort of such implementations: BWA-MEM-CUDA and BWA-MEM-OpenCL require 5-7x more lines of code to express the Smith-Waterman algorithm, while BWA-MEM-VHDL requires more than 40x as many lines of code. Similar differences hold for required implementation time, ranging from one month for BWA-MEM-OpenCL to six months for BWA-MEM-VHDL. The advantages and disadvantages of each implementation are described using both quantitative and qualitative metrics, and recommendations are given for future algorithm implementations.**

***Keywords*-CUDA, Design-Time Optimization, FPGA, GPU, OpenCL, Sequence Alignment, Smith-Waterman, VHDL.**

I. INTRODUCTION

Acceleration of bioinformatics algorithms is critical, due to the huge data sets and complex algorithms involved. However, algorithms and methods are still under active development, resulting in a trade-off between optimal performance and development time. Heterogeneous hardware accelerators such as GPUs, FPGAs or DSPs often have complex and sophisticated architectures, making their programmability critically important. High-level languages can expose the power of these heterogeneous compute devices for relatively little development effort by partly abstracting away the underlying hardware complexity, thus immensely increasing the productivity of a hardware engineer. Many programming models exist for these platforms, each with their own advantages and disadvantages, offering various levels of performance, flexibility and programmability, such as CUDA, OpenCL, VHDL, Verilog, or the various High Level Synthesis (HLS) tools. Here, we compare CUDA, OpenCL and VHDL on a number of metrics, based on our experiences gained from accelerating BWA-MEM, a widely used bioinformatics algorithm. Moreover, we distill some guidelines on when to prefer a certain programming model above others, using our work on BWA-MEM and the Smith-Waterman (S/W) algorithm as a case study.

II. BACKGROUND

The use of heterogeneous platforms to accelerate computation is not new. However, the rate of adoption and ubiquitous nature of current heterogeneous systems, such as cellphones that use SoCs with a variety of specialized hardware blocks, or data centers with nodes containing highly specialized hardware, is a relatively new phenomenon. Recent developments in machine learning are partially enabled through the widescale adoption of massively parallel GPU hardware. Equally important are the improvements in high-level languages. CUDA [1] and OpenCL [2], first released in 2007 and 2009, respectively, made the massively parallel hardware of GPUs available for general purpose problems beyond just graphics rendering, thereby creating the general purpose GPU compute paradigm, also known as GPGPU. Similarly, whereas FPGAs were typically programmed using VHDL, the introduction and ongoing improvement of high-level synthesis tools from 1990s onward [3] made their use more accessible. The more recent introduction of OpenCL for FPGAs [4] again greatly lowers the barrier to utilize such devices. Higher level language may not always be the best tool when the only goal is absolute performance, but these languages do enable a much broader range of applications.

Here, we compare our experiences while creating multiple heterogeneously accelerated implementations of BWA-MEM [5], a widely used genomic sequence mapping tool. To accelerate this application, we offloaded the part of BWA-MEM that performs pairwise sequence alignment using the S/W algorithm [6] onto FPGA and GPU. The S/W algorithm, along with the highly similar Needleman-Wunsch algorithm, is used in many bioinformatics tools, such as BLAST, BWA-MEM and ClustalW. It is able to find the optimal pairwise alignment of two sequences, typically DNA or protein sequences. The acceleration of BWA-MEM and S/W presents us with a good case study, since on the one hand the S/W algorithm consists of a single compact and computationally intensive kernel and is regular, which makes it easy to scale up to utilize the full resources of any device by being able to both accept longer sequences and through the use of multiple modules. On the other hand, the broader effort to accelerate the complete BWA-MEM application gives insight into the additional complexities that arise when including an accelerated kernel into a larger program.

2471-7819/18/$31.00 ©2018 IEEE
DOI 10.1109/BIBE.2018.00053

Table I
METHODS USED FOR THE HETEROGENEOUSLY ACCELERATED BWA-MEM / SMITH-WATERMAN IMPLEMENTATIONS

	CUDA	OpenCL	VHDL/HLS
Developer	NVIDIA	Khronos Group	Multiple
Implementer	NVIDIA	Altera,AMD,Apple,ARM, Intel,Samsung,Xilinx(a.o.)	Altera,Cadence, Xilinx (a.o.)
Scope	Proprietary	Open Standard	Proprietary
Platform(s)	NVIDIA GPU	CPU, DSP, FPGA, GPU	FPGA, ASIC
Key Concepts	Blocks, Threads	Kernels, Channels	Code Transform
Initial Release	2007	2009	1990s
Latest Version	CUDA 9.2	OpenCL 2.2	IEEE 1076-2008 [7]

III. HIGH-LEVEL LANGUAGES & TARGET PLATFORMS

Numerous high-level languages exist to address heterogeneous systems. Table I shows characteristics of those languages we have used to adapt BWA-MEM and the S/W algorithm. Each of these is now described in more detail. Note that this list is not exhaustive; many other languages and frameworks exist that can take advantage of hardware accelerators, such as Metal, SDAccel, or TensorFlow.

CUDA for NVIDIA GPUs: While GPUs have been used for general purpose processing even through graphics-oriented APIs such as OpenGL, the introduction of CUDA [1] started the revolution of GPU usage for mainstream compute. Even though CUDA is only available for NVIDIA GPUs, it is highly relevant in the GPU-compute field. Development of CUDA and further generalization of GPU hardware have caused extraordinary growth in application of general purpose GPU compute. Key concepts in CUDA are *Threads* and *Thread Blocks*. Parallel work is distributed amongst threads, which each perform the same kernel function applied to different data elements. Threads are grouped into thread blocks that describe the layout of the data structure. Thread blocks are dispatched onto the GPU's execution units, with numerous thread blocks executing in parallel. The GPU-accelerated version of BWA-MEM will be referred to as **BWA-MEM-CUDA**. The pairwise sequence alignment phase is similar to the regular S/W algorithm, and is offloaded on the GPU. The S/W algorithm computes the optimal score of an alignment by computing a similarity matrix. Each thread block computes one such matrix, and each thread within the thread block computes a column. More information can be found in [8] and [9].

VHDL+HLS for FPGA: A Field-Programmable Gate Array (FPGA) is an accelerator similar to an ASIC, being programmed to perform a single algorithm in hardware. However, an FPGA can be reprogrammed to change its function whenever required, allowing for greater flexibility compared to an ASIC, which is typically very expensive. FPGAs and ASICs offer high computational capabilities with very low power consumption [10]. However, hardware design with languages such as VHDL or Verilog is complex, and significantly more time-consuming than writing a software program. This resulted in development of High Level Synthesis (HLS) tools, which can be used to generate Register Transfer Level (RTL) representations from higher level languages such as C, C++, greatly reducing the time required to program the hardware. Many compilers that can perform HLS exist. The main FPGA vendors (Intel and Xilinx) each have their own HLS compilers, but many other commercial and academic HLS compilers are also available, differing in the input languages they accept for translation, such as the DWARV compiler [11], which translates a subset of C into VHDL. A disadvantage of any hardware design, either created through VHDL or HLS, is the requirement to place and route logic elements, which is very time-consuming, often taking hours or even days. To create **BWA-MEM-VHDL**, a combination of VHDL and HLS was used for the accelerated implementation of BWA-MEM on FPGA. The S/W kernel was handwritten in VHDL, whereas less critical "glue"-logic was created using HLS. The S/W similarity matrix is computed using a systolic array, where each Processing Element computes a single column of the matrix. More details can be found in [12] and [10].

Intel OpenCL for FPGA: OpenCL is a C-like language for programming a wide range of devices, such as CPUs, GPUs, DSPs, and FPGAs [2]. An OpenCL program utilizes *kernels*, which are executed on one or more *compute units*. These compute units consist of processing elements. In the case of Intel's OpenCL implementation for its FPGAs [4], they also added the concept of a *channel*, which is an FPGA-specific construct that allows two kernels to directly communicate through an on-chip communication channel, instead of having to traverse the memory hierarchy, greatly improving efficiency and performance. Similar to the VHDL implementation, our BWA-MEM implementation in OpenCL, **BWA-MEM-OpenCL**, utilizes a systolic array to calculate the S/W similarity matrix, consisting of multiple kernels that utilize channels pervasively to avoid memory transaction except for the initial data loading and final results storing. More details can be found in [13].

Software-Only Execution: Forgoing the use of an accelerator and simply use software-only execution is a valid design decision. The time required to implement a heterogeneous accelerated solution may not be worthwhile when execution time is not a key design consideration. Also, when a project is still in the early phase of design exploration, software-only execution provides easier algorithm exploration, is easier to debug and troubleshoot, and may sometimes even prove to be the fastest solution, such as in latency bound, conditional branch-heavy code. In any case, the software-only execution case can provide us with a firm baseline for comparison.

IV. CASE STUDY: BWA-MEM AND SMITH-WATERMAN

The high-level languages used for our implementations are compared on implementation complexity, the available development environment, the portability, and performance. Table II summarizes the findings.

Table II
COMPARISON BETWEEN HETEROGENEOUS BWA-MEM / SMITH-WATERMAN IMPLEMENTATIONS ACROSS MULTIPLE METRICS

		BWA-MEM-CUDA	BWA-MEM-OpenCL	BWA-MEM-VHDL	Software-Only
Code Complexity	**Overall**	**++**	**++**	**–**	**++**
	Kernel Code	∼500 lines	∼700 lines	∼3,700 lines	<100 lines
	Kernel Routines	3 functions	10 kernels	17 modules	single function
	Driver Code	∼300 lines	∼200 lines	∼1,700 lines	N/A
	Development Time	∼3 man-months	∼1 man-months	∼6 man-months	N/A
Environment	**Overall**	**++**	**+/-**	**–**	**++**
	IDE/SDK	NVIDIA Nsight	Intel SDK available	many available	many available
	Debugging	normal debug tools: breakpoints, stepping	emulation (fast), then simulation (slow)	VHDL simulation, requires testbenches	normal debug tools: breakpoints, stepping
	Compilation Time	minutes	hours	hours	minutes
Portability	**Overall**	**–**	**+/-**	**+/-**	**++**
	Hardware Architecture	fixed, complex	flexible	flexible	flexible
	Scalability	multiprocessor	FPGA logic elements	FPGA logic elements	CPU cores
Performance	**Overall**	**++**	**++**	**++**	**–**
	Device	GeForce GTX 970	Intel Arria 10 GX DevKit	Alpha Data ADM-PCIE-7V3	Intel Core i7-4790
	Specifications	13 SMXs @ 1,150 MHz	10 modules @ 164 MHz	6 modules @ 160 MHz	4 cores @ 3,600 MHz
	Throughput (SW)	-	215 GCUPS (1-cycle)	42 GCUPS (3-cycle)	-
	Throughput (BWA-MEM)	4.3 Mbp/s	*4.5 Mbp/s*	4.5 Mbp/s	2.2 Mbp/s

Code Complexity: for complexity we use the required development time and the lines of code (LoC) for implementing the kernel, and LoC required to integrate the S/W kernel into the main application, not counting empty lines or comments. Baseline software-only execution requires fewer than a hundred LoC. All accelerated versions require significantly more code, in-line with expectations that accelerated implementations will always require more LoC than software-only execution. However, implementations differ greatly in this regard: BWA-MEM-VHDL requires more than 5,000 kernel and driver LoC, an order of magnitude more than BWA-MEM-CUDA and BWA-MEM-OpenCL. It also required by far the most development time. Differences between BWA-MEM-OpenCL and BWA-MEM-CUDA are smaller. BWA-MEM-OpenCL requires more LoC as the implementation consists of many simple kernels, causing code replication. BWA-MEM-CUDA and BWA-MEM-OpenCL both have many predefined constructs available and offer much higher levels of abstraction compared to BWA-MEM-VHDL: CUDA and OpenCL allow instantiation of kernels from the application without the need for a driver; the memory hierarchy is (partially) abstracted away. Finally, illustrating the complexity involved: a five-line change in the BWA-MEM sequence alignment kernel of version 0.7.9 required a week of implementation time and a week of testing and verification to update BWA-MEM-VHDL as the code is very different from the original source code. In contrast, BWA-MEM-CUDA and BWA-MEM-OpenCL only required minor changes to their implementations.

Maturity and Ease of Development Environment: IDEs and SDKs greatly influence developer productivity. The CUDA development environment is similar to software-only development as NVIDIA Nsight allows debugging facilities such as breakpoints, code stepping, and variable inspection. In contrast, two main challenges were faced during FPGA development of BWA-MEM-OpenCL or BWA-MEM-VHDL: the development cycle is inherently complex and time-consuming due to placement and route. Also, monitoring the internal state of the device is not possible without adding extra outputs to the design. For BWA-MEM-OpenCL, software emulation was used to verify the correctness of functional requirements, but subsequent simulation for performance optimization was very slow. Similarly, for BWA-MEM-VHDL, co-simulation was used for the HLS parts, but the VHDL kernel was debugged using a VHDL simulator, which requires debugging on a signal-level. Moreover, writing VHDL testbenches is very time-consuming.

Portability: Optimization often reduces portability, and accelerators exacerbate this issue. CUDA locks you into the NVIDIA ecosystem, and moreover, taking optimal advantage of a GPU requires full understanding of the underlying architecture, so that code maps efficiently on to the available hardware resources, requiring knowledge of threads, thread blocks, memory access and coalescing rules, and the impact of register usage per thread. A redeeming feature of CUDA is its inherent scalability to wider GPUs, since thread blocks automatically map onto available multiprocessors. However, GPU architectures vary widely, requiring code refactoring for optimal performance: from new features such as warp shuffle, dynamic parallelism and tensor cores, to feature set differences such as the shared memory per multiprocessor, or the steady increase in number of resident threads and warps. In contrast, FPGA development is less constrained, since the underlying substrate is in essence just a large number of logic elements and wires that can be programmed to any desired function. Some knowledge of the underlying technology is still required, but much fewer restrictions are in place compared to the relatively fixed GPU design. Changing between vendors or devices will, however, require a recompilation at the very least.

Performance: Throughput results are shown both at the application level for BWA-MEM, measured in million bases aligned per second (Mbp/s), and at the kernel level for S/W, measured in billions of cell updates (GCUPS). Estimates are shown in italics. Application-level results show that BWA-MEM-VHDL and BWA-MEM-CUDA are twice as fast as software-only execution, at 4.3-4.5 Mbp/s, the maximum speedup obtainable from overlapping the pairwise sequence alignment phase with other parts of BWA-MEM. The BWA-MEM-VHDL design contains six hardware modules to perform S/W [10]. BWA-MEM-CUDA performs about equivalent to a five module design, on a GeForce GTX 970. BWA-MEM-OpenCL performs approximately equal to a thirty module design. Conversely, one module of BWA-MEM-OpenCL is fast enough to obtain a two-fold BWA-MEM speedup. The S/W results show the maximum performance for both BWA-MEM-OpenCL and BWA-MEM-VHDL. The performance difference is due to the fact that the FPGA on the Intel Arria 10 GX used for BWA-MEM-OpenCL is both larger and faster than the FPGA on the Alpha Data card: it has about 50% more logic units, and timing constraints cause BWA-MEM-VHDL to use three cycles to compute a single similarity matrix entry compared to one cycle for BWA-MEM-OpenCL.

V. DISCUSSION

Although VHDL offers complete control over the FPGA design, our experiences with OpenCL and CUDA have shown that these languages are far easier to use: less code needs to be written, and mundane tasks such as starting a kernel from the host program or copying data between host and device memory are unnecessary, resulting in a shortened development cycle. For software still under development this can be particularly relevant, highlighting the trade-off between performance and development time. In this sense, both CUDA and OpenCL seem the better choice. In [14], an implementation of a tsunami simulation both on GPU and FPGA using OpenCL, similar findings are reached. The original Verilog code consisted of over 600,000 lines, whereas the OpenCL code spanned just several hundred lines. Moreover, they could reuse the OpenCL kernel initially developed for the GPU on an FPGA, although it took four implementations to obtain similar performance, showing the advantage in target platforms of OpenCL over both CUDA and VHDL: a single code base can support a variety of hardware architectures.

VI. CONCLUSION

We have compared our experiences in developing three heterogeneously accelerated version of BWA-MEM: BWA-MEM-CUDA targeting the GPU using CUDA, and BWA-MEM-VHDL and BWA-MEM-OpenCL targeting the FPGA using VHDL and OpenCL. Each version implements a highly optimized version of the S/W algorithm. Although all languages offer high performance, the complexity, required effort and the maturity of the programming environments differs greatly. The CUDA and OpenCL implementations are much simpler and required fewer development time than VHDL, requiring 5-7x more LoC compared to the original code and required between 1-3 months of development. In contrast, BWA-MEM-VHDL required 40x the number of LoC, partially due to additional effort required in writing a driver, and six months of development time. Inherent to FPGA design is the slow compilation cycle, which can take hours. Although software emulation and co-simulation can be used for functional tests, performance optimization requires the complete simulation cycle. For bioinformatics tools still under active development, we would recommend against heterogeneous acceleration unless increased performance is absolutely required. Even then, higher-level languages such as CUDA or OpenCL are preferable over VHDL due to the much lower complexity.

REFERENCES

[1] NVIDIA, "NVIDIA CUDA programming guide (version 1.0)," *NVIDIA: Santa Clara, CA*, 2007.

[2] A. Munshi, "The OpenCL specification," in *Hot Chips 21 Symposium (HCS), 2009 IEEE*. IEEE, 2009, pp. 1–314.

[3] D. Knapp, *Behavioral synthesis: digital system design using the Synopsys behavioral compiler*. Prentice Hall PTR, 1996.

[4] Intel, "The Intel FPGA SDK for Open Computing Language," https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html, last visited: 2017-08-24.

[5] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM," *arXiv preprint arXiv:1303.3997*, 2013.

[6] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[7] IEEE Design Automation Standards Committee and others, "Std 1076-2008, IEEE Standard VHDL Language Reference Manual," *IEEE, New York, NY, USA*, 2008.

[8] E. Houtgast, V.-M. Sima, K. Bertels, and 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.

[9] ——, "An efficient GPU-accelerated implementation of genomic short read mapping with BWA-MEM," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 4, pp. 38–43, 2017.

[10] E. Houtgast, V.-M. Sima, G. Marchiori, K. Bertels, and Z. Al-Ars, "Power-efficiency analysis of accelerated BWA-MEM implementations on heterogeneous computing platforms," in *ReConFigurable Computing and FPGAs (ReConFig), 2016 International Conference on*. IEEE, 2016, pp. 1–8.

[11] Y. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, Y. Lu, and S. Vassiliadis, "DWARV: Delftworkbench automated reconfigurable VHDL generator," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*. IEEE, 2007, pp. 697–701.

[12] E. Houtgast, V.-M. Sima, K. Bertels, and 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.

[13] E. Houtgast, V.-M. Sima, and Z. Al-Ars, "High performance streaming Smith-Waterman implementation with implicit synchronization on Intel FPGA using OpenCL," in *2017 IEEE 17th International Conference on Bioinformatics and Bioengineering (BIBE)*. IEEE, 2017, pp. 492–496.

[14] F. Kono, N. Nakasato, K. Hayashi, A. Vazhenin, and S. Sedukhin, "Evaluations of OpenCL-written tsunami simulation on FPGA and comparison with GPU implementation," *The Journal of Supercomputing*, vol. 74, no. 6, pp. 2747–2775, 2018.

Computational Biology and Chemistry 75 (2018) 54–64

Contents lists available at ScienceDirect

Computational Biology and Chemistry

journal homepage: www.elsevier.com/locate/compbiolchem

Research Article

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

ARTICLE INFO

Article history:
Received 7 March 2018
Received in revised form 17 March 2018
Accepted 22 March 2018
Available online 12 April 2018

Keywords:
Acceleration
BWA-MEM
FPGA
GPU
Short read mapping
Systolic array

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.

Sequencers typically output 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.

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 (Stephens et al., 2015). 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 Fig. 1, the mapping phase takes a significant amount of time of the overall genomics pipeline execution time.

* Corresponding author at: Computer Engineering Lab, TU Delft, Mekelweg 4, 2628 CD Delft, The Netherlands.
E-mail address: e.j.houtgast@tudelft.nl (E.J. Houtgast).

https://doi.org/10.1016/j.compbiolchem.2018.03.024
1476-9271/

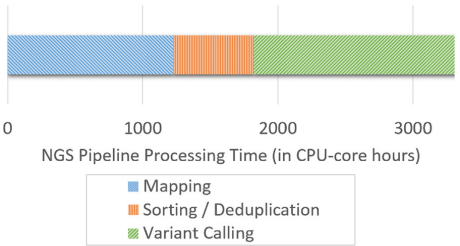


Fig. 1. Breakdown of processing time per NGS pipeline stage for a typical 30× coverage cancer NGS DNA data set. The data set consists of three tumor samples and one normal tissue sample (time given in CPU-core hours).

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 (Illumina, 2015), 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 (Illumina, 2016). 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 2×, 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 Fig. 2. BWA-MEM (Li, 2018) 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 (Liu et al., 2012a) and CUSHAW (Liu et al., 2012b), 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 Ligowski and Rudnicki (2009), Hasan et al. (2011), Manavski and Valle (2008), and Di Tucci et al. (2017). 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 (Houtgast et al., 2015), as well as by using an AlphaData add-in board (Houtgast et al., 2016a,b,c). These implementations were able to achieve an up to two-fold speedup. We also ported our work onto the GPU (Houtgast et al., 2016a,b,c), 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 4600 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

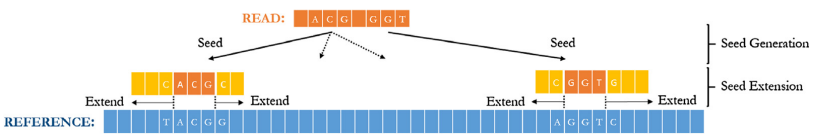


Fig. 2. Most state-of-the-art mapping tools use a paradigm called Seed-and-Extend to map a short read fragment onto a reference genome: first, exactly matching subsequences between the short read and the reference genome are identified, using for example the BWT. These are called *seeds*. Then, these 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.

1.26× speedup (Chang et al., 2016), and the work by Chen, which accelerates the Seed Extension phase and is able to achieve a 1.5× speedup (Chen et al., 2016).

3. BWA-MEM algorithm details

BWA-MEM is a popular short read mapping tool (Li, 2018), 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 Fig. 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 Fig. 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 (Highnam et al., 2015) 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.

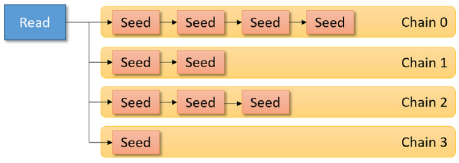


Fig. 3. BWA-MEM Seed Generation can result in many seeds being identified for a single read. Seeds that are located in close proximity of one another on the reference are grouped into chains.

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%	1.2
Seed Extension	40%	40%	39%	38%	1.2
Other	15%	13%	15%	18%	1.2
Total time	656 s	594 s	589 s	612 s	

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 1.7×. We can only achieve higher speedup if other kernels are accelerated as well, similar to what has been done in Ahmed et al. (2015).

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
```

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 Fig. 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

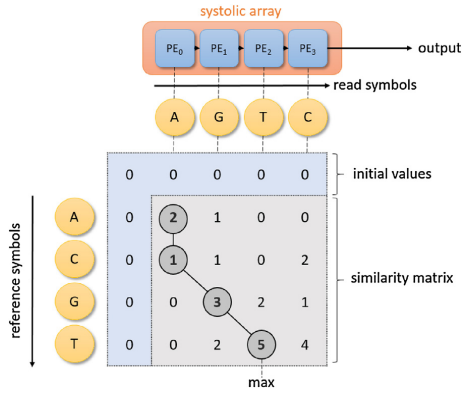


Fig. 4. Smith-Waterman algorithm similarity matrix. The maximum score is indicated. As matrix entries only depend on top, top-left, and left neighbor, anti-diagonals can be processed in parallel. This makes the systolic array a natural implementation choice, whereby each column is processed by one Processing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

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 Houtgast et al. (2016a,b,c).

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 Houtgast et al. (2016a,b,c). 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.

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 (Houtgast et al., 2016a,b,c) 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

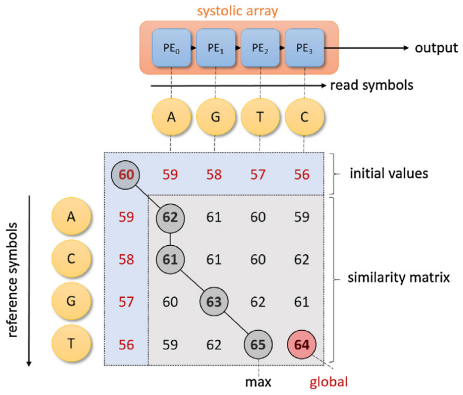


Fig. 5. Inexact Matching algorithm similarity matrix with an initial score of 60. The maximum and global maximum scores are indicated. Differences as compared to the regular Smith-Waterman algorithm include the presence of initial values and computation of a global maximum score. The locations of both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

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 Fig. 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 Fig. 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 Fig. 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

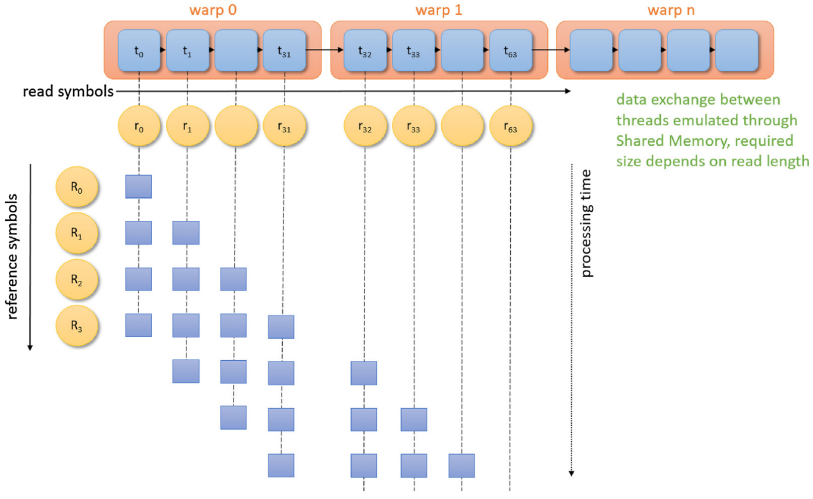


Fig. 6. Overview of the "wide" systolic array implementation, showing active threads when processing the similarity matrix with as many threads as there are read symbols. The data exchanged between the successive threads passes through Shared Memory, resulting in a dependence on the read length for the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

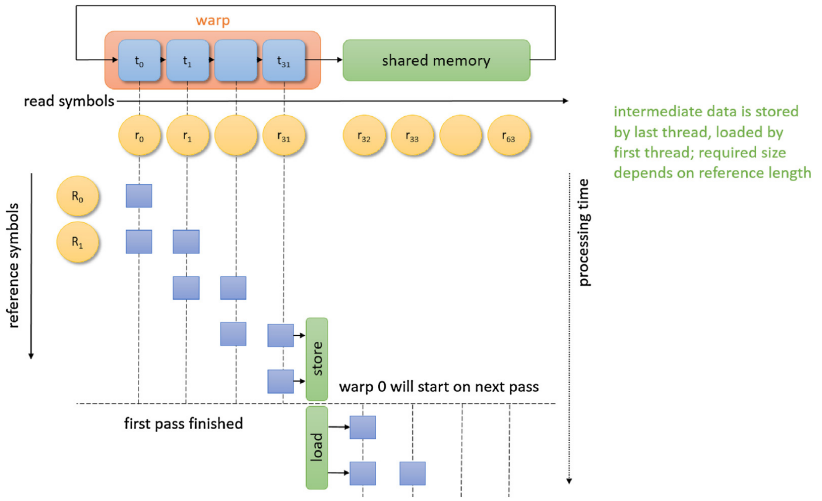


Fig. 7. Overview of the single warp systolic array implementation, showing active threads when processing the similarity matrix with one warp. Multiple passes are made over the similarity matrix. Threads exchange data directly, eliminating the need to store this in Shared Memory. Data exchange between 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.

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 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.

Figs. 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 (NVIDIA, 2016a, b). The results show that the performance is mostly limited by latency of arithmetic and memory instructions. The memory subsystem utilization is shown in Fig. 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

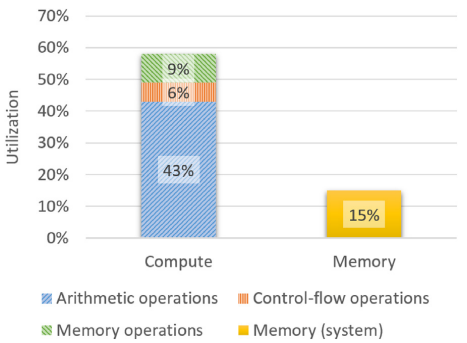


Fig. 8. Output of NVIDIA Visual Profiler Latency Analysis for a test set with two hundred thousand reads of 150 bp. The implementation performance is mostly limited by the latency of arithmetic and memory operations, and by the number of resident blocks per multiprocessor.

	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

Fig. 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.

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 $2 \times 150 \times 2 = 600$ 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 (NVIDIA, 2016a,b). 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 Capability 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 Houtgast et al. (2016a,b,c).

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 Fig. 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

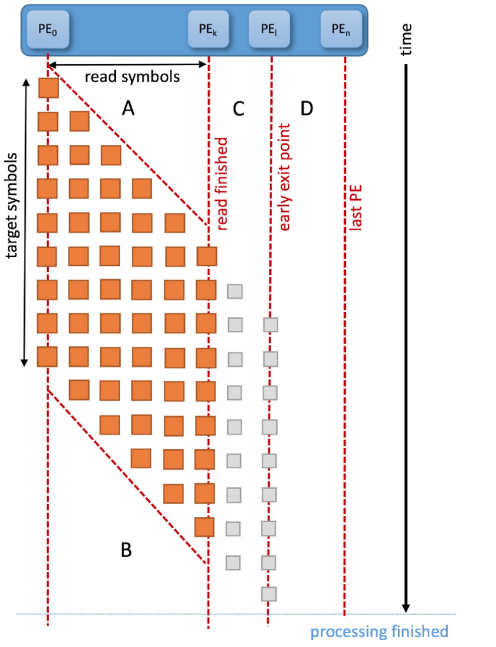


Fig. 10. The efficiency of a systolic array is heavily dependent on the length of the read and target, as compared to the length of the systolic array itself. Areas indicated by A, B, C, and D are areas of inefficiency, where some or all of the Processing Elements are not contributing useful work. Reducing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

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.

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 Houtgast et al. (2015), 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 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 Fig. 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 96×100 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 (Alpha Data, 2015), 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 (Highnam et al., 2015). 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 (Bioplanet.com, 2016) 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 $2\times$ 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 et al. (2016) and Chen et al. (2016). 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 (Chen et al., 2016) accelerates the BWA-MEM Seed Extension phase, using the same AlphaData FPGA board. Chang is able to achieve an overall application-level speedup of $1.26\times$, whereas Chen claims an overall application-level speedup of $3\times$. However, their baseline of comparison is the *Cloud-Scale* BWA-MEM implementation, which performs about 50% slower than regular BWA-MEM (Chen et al., 2015). Moreover, their experimental platform, a dual node Intel

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 (Houtgast et al., 2016a,b,c)	129 s	1.8×	272 s	2.0×	4.5 Mbp/s
	Seed Extension on GPU (Houtgast et al., 2016a,b,c)	144 s	1.6×	278 s	2.0×	4.3 Mbp/s
Chang (Chang et al., 2016)	*Seed Generation on FPGA*	*N/A*	*4×*	*N/A*	*1.26×*	*N/A*
Chen (Chen et al., 2016)	*Software-Only (CW-BWAMEM)*	*N/A*	–	*N/A*	–	*1.2 Mbp/s*[a]
	Seed Extension on FPGA	*N/A*	*10.5×*	*N/A*	*3×*	*3.6 Mbp/s*[a]

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

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 they are able to obtain a 3× 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 250 bp, the full two-fold performance increase is attained. The 400 bp data set only achieves an at most 1.7× 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 $O(\text{READ} + \text{REFERENCE})$, but as $O(\text{READ} \times \text{REFERENCE})$. 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 Houtgast et al., 2015 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 Fig. 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 150 bp 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 400 bp 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 Fig. 11. More importantly, it should prevent an overall application-level slowdown. Under GPU-constrained situations, performance can improve by up to +45%.

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.

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 (100 bp)	gcat42 (150 bp)	gcat46 (250 bp)	gcat50 (400 bp)
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%

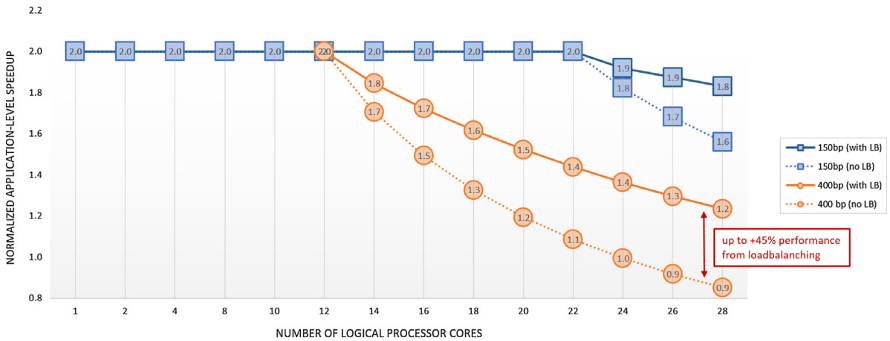


Fig. 11. Estimated application-level speedup depending on CPU core count for 150 bp and 400 bp data set. Load balancing improves performance in GPU-constrained scenarios, ensuring application speedup in all cases.

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% and 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 250 bp can be accelerated with the maximum two-fold application-level speed-up. 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

Ahmed, N., Sima, V., Houtgast, E., Bertels, K., Al-Ars, Z., 2015. Heterogeneous hardware/software acceleration of the BWA-MEM DNA alignment algorithm. Proc. of the IEEE/ACM Intl. Conf. on Computer-Aided Design, ICCAD.

Alpha Data, 2015. Alpha Data ADM-PCIE-7V3 Product Information. . (accessed 14.12.15) http://www.alpha-data.com/dcp/products.php?product=adm-pcie-7v3.

Bioplanet.com, 2016. Genomic Comparison and Analytic Testing. . (accessed 16.11.16) http://www.bioplanet.com/gcat.

Chang, M.-C.F., Chen, Y.-T., Cong, J., Huang, P.-T., Kuo, C.-L., Yu, C.H., 2016. The SMEM seeding algorithm acceleration for DNA sequence alignment. Proc. 24th IEEE International Symposium on Field-Programmable Custom Computing Machines, Washington DC, USA.

Chen, Y.-T., Cong, J., Lei, J., Li, S., Peto, M., Spellman, P., Wei, P., Zhou, P., 2015. CS-BWAMEM: A Fast and Scalable Read Aligner at the Cloud Scale for Whole Genome Sequencing. .

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

Di Tucci, L., O'Brien, K., Blott, M., Santambrogio, M.D., 2017. Architectural optimizations for high performance and energy efficient Smith-Waterman implementation on FPGAs using OpenCL. 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE), IEEE, pp. 716–721.

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

Highnam, G., Wang, J.J., Kusler, D., Zook, J., Vijayan, V., Leibovich, N., Mittelman, D., 2015. An analytical framework for optimizing variant discovery from personal genomes. Nat. Commun. 6.

Houtgast, E., Sima, V., Bertels, K., Al-Ars, Z., 2015. An FPGA-based systolic array to accelerate the BWA-MEM genomic mapping algorithm. Intl. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation .

Houtgast, E., Sima, V., Bertels, K., Al-Ars, Z., 2016a. GPU-accelerated BWA-MEM genomic mapping algorithm using adaptive load balancing. Architecture of Computing Systems-ARCS. Springer, pp. 130–142.

Houtgast, E., Sima, V., Marchiori, G., Bertels, K., Al-Ars, Z., 2016b. Power-efficient accelerated genomic short read mapping on heterogeneous computing platforms. Proc. 24th IEEE International Symposium on Field-Programmable Custom Computing Machines, Washington DC, USA.

Houtgast, E.J., Sima, V., Bertels, K.L.M., Al-Ars, Z., 2016c. An efficient GPU-accelerated implementation of genomic short read mapping with BWA-MEM. Proc. International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies, Hong Kong, China.

Illumina, 2015. HiSeq X Specification Sheet. . (accessed 15.07.15) http://www.illumina.com/content/dam/illumina-marketing/documents/products/datasheets/datasheet-hiseq-x-ten.pdf.

Illumina, 2016. Illumina Sequencing Systems. . (accessed 16.11.16) http://www.illumina.com/systems/sequencing.html.

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

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

Liu, C.-M., Wong, T., Wu, E., Luo, R., Yiu, S.-M., Li, Y., Wang, B., Yu, C., Chu, X., Zhao, K., Li, R., 2012a. SOAP3: ultra-fast GPU-based parallel alignment tool for short reads. Bioinformatics 28 (6), 878–879.

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

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

NVIDIA, 2016a. CUDA C Programming Guide. . (accessed 20.01.16) http://docs.nvidia.com/cuda/cuda-c-programming-guide.

NVIDIA, 2016b. NVIDIA Visual Profiler. . (accessed 14.01.16) https://developer.nvidia.com/nvidia-visual-profiler.

Stephens, Z., Lee, S., Faghri, F., Campbell, R., Zhai, C., Efron, M., Iyer, R., Schatz, M., Sinha, S., Robinson, G., 2015. Big data: astronomical or genomical? PLoS Biol. 13 (7) .

4

CHAPTER 5

Conclusions

"Far better is it to dare mighty things,
To win glorious triumphs, even though checkered by failure..
Than to rank with those poor spirits who neither enjoy nor suffer much,
Because they live in a gray twilight that knows not victory nor defeat."

— Theodore Roosevelt

JUDICIOUS use of the proposals in this dissertation can, when combined, result in an improved systolic array architecture that is able to operate at maximal efficiency. These findings have been used to create accelerated implementations of the Seed Extension kernel of BWA-MEM, a widely-used genomic mapping algorithm, offering a twofold increase to application-level performance. Moreover, an FPGA-accelerated implementation has been created of the Smith-Waterman pairwise sequence alignment algorithm, offering the fastest performance and highest efficiency to date. In this chapter, the work presented in the previous chapters is summarized, and a few suggestions for potential future research directions are offered. In Section 5.1, the main contributions per chapter are briefly revisited. In Section 5.2, opportunities for future work are discussed.

5.1. CONCLUSIONS

CONCLUSIONS FROM CHAPTER 1

In **Chapter 1**, "Introduction", the motivation and the background for this dissertation are presented. Advancements in sequencing techniques and the widespread adoption of sequencing hardware are resulting in increasingly large amounts of data being generated. This data requires processing through so-called genomics pipelines, computationally demanding pipelines of bioinformatics algorithms. Acceleration and optimization of such pipelines is critical in order to keep up with the continuously growing data processing needs. Heterogeneous systems are systems consisting of a variety of different hardware architectures, where each architecture is specialized for a specific domain. Such systems can offer the performance and power-efficiency that will be expected of future bioinformatics applications.

This dissertation focuses on improvements to techniques used for the acceleration of two widely used bioinformatics algorithms: the BWA-MEM genomic mapping algorithm and the Smith-Waterman pairwise sequence alignment algorithm, which are both explained in some detail. These algorithms can be accelerated using the systolic array architecture, as the Smith-Waterman dynamic programming similarity matrix maps well onto this architecture. However, the short read and reference sequences generated by the BWA-MEM algorithm limit the efficiency of a standard systolic array implementation, as systolic array utilization significantly depends on long sequence lengths. This highlights the need for improvements to this architecture that reduce or remove this dependence on read sequence and reference sequence length. The improvements to the systolic array architecture proposed in this dissertation completely remove any such limitations. Furthermore, they are not only useable to improve the performance of the BWA-MEM Seed Extension algorithm, but can also be applied to accelerate or improve the efficiency of a variety of other applications as well.

CONCLUSIONS FROM CHAPTER 2

In **Chapter 2**, "Systolic Array Architectures and the Seed Extension Kernel", one of the two main sources of inefficiency of systolic arrays is addressed, namely the dependence of systolic array efficiency on read sequence length. Due to the fact that the read symbols are mapped one-to-one on the PEs of the systolic array, in the case that a read sequence is to be processed on the array that contains fewer symbols than the number of PEs in the systolic array, part of the PEs will have to remain idle, thus resulting in inefficient use of the array. A number of techniques is proposed, namely Exit Points and three different systolic array architectures: the Variable Logical Length, Variable Physical Length, and Variable Logical and Physical Length systolic array. These can be used to completely eliminate the systolic array's dependence on read sequence length and, therefore, to obtain a systolic array implementation that can operate at maximum efficiency for any data set, regardless of its distribution of read sequence lengths.

This independence is achieved through modification of the systolic array architecture. Exit Points are introduced, which are points in the systolic array that allow shorter read sequences to bypass the remainder of the systolic array, thus reducing the time required to process shorter read sequences. An array including such Exit Points is called a Variable Logical Length systolic array. In contrast, a Variable Physical Length systolic array consists of a collection of differently dimensioned systolic arrays, combined with a simple scheduler to route alignments to the proper unit. This technique avoids that PEs are being idle altogether. Apart from the benefit to execution time, this technique grants an additional advantage, since systolic arrays with fewer PEs require correspondingly fewer area on the FPGA. When combined, these techniques result in a Variable Physical and Logical Length systolic array.

Variable Logical Length-based Seed Extension kernels have been implemented on both FPGA and GPU to respectively create an FPGA-accelerated and a GPU-accelerated version of the widely used genomic mapping algorithm BWA-MEM. These Seed Extension kernels offer a threefold performance increase, when compared to the software-only Seed Extension kernel.

CONCLUSIONS FROM CHAPTER 3

In **Chapter 3**, "Optimized Implementations with Efficient Systolic Arrays", the second source of systolic array inefficiency is addressed: the dependence of the systolic array's efficiency on reference sequence length. A regular systolic array implementation is only able to perform a single pairwise sequence alignment operation at a time. Thus, even when a short reference sequence is used as input to a pairwise alignment, this short reference sequence is required to pass through the entire array before the next pairwise sequence alignment operation can commence. This obviously results in dramatically low efficiency. Through a design in which the read and reference symbols belonging to different pairwise sequence alignments are buffered in Query Buffers and immediately streamed into the systolic array, one directly after the other, as opposed to having to wait for a single pairwise sequence alignment computation to finish before starting the next one, the dependence on reference sequence length can be completely eliminated. The result is a maximally-efficient systolic array. These techniques are used to create the fastest and most efficient Smith-Waterman FPGA implementation available to date, with an overall efficiency of 99.8% and a performance of 214 GCUPS.

Furthermore, improved BWA-MEM implementations are discussed that are able to completely eliminating the Seed Extension kernel from overall program execution time by overlapping the Seed Extension computations on the accelerator with the execution of other functions on the host machine, thus attaining the maximum possible twofold speedup that is possible according to Amdahl's law. The best overall efficiency is obtained only when a host system and its accelerator, be it an FPGA or a GPU, are well-matched with one another, so that neither has to wait for the other to finish. Therefore, a scalability analysis is performed to estimate the best host configuration for both the FPGA and the GPU implementation. This analysis shows that these implementations perform best and are able to scale to host machines with up to 22 cores for the GPU implementation, and up to 26 cores for the FPGA implementation.

CONCLUSIONS FROM CHAPTER 4

In **Chapter 4**, "Power-Efficiency, Design-Time, and Read Length Analysis", a number of other important design metrics are considered. The power-efficiency of the hardware-accelerated BWA-MEM implementations is compared against the baseline software-only version. Both implementations are able to offer a twofold performance improvement. However, only the FPGA implementation is able to also reduce the overall energy consumption for the configuration as tested, resulting in an implementation that is 1.6x as efficient as the software-only version. In contrast, although the GPU implementation is twice as fast as the software-only implementation, it also consumes twice as much power, thus erasing any power-efficiency gains. A scalability analysis is performed to see whether it is possible to improve the power-efficiency further, when the host system and the accelerator are better matched to one another so that their execution time is comparable and neither has to wait for the other. Estimations for such a more balanced system with a faster host processor indicate that, in the case of the FPGA implementation, the power-efficiency improves to 2.1x for a system with 16 cores, and, in the case of the GPU implementation, at least some power-efficiency gains can be achieved, with a power-efficiency that peaks at 1.4x for a system with 15 cores.

The design-time of a heterogeneous solution can be an important aspect to consider, especially in a relatively new field such as bioinformatics where algorithms continue to be improved and replaced. Although heterogeneous designs are able to offer improvements in both absolute performance as well as in power-efficiency, they also require significantly more development effort, even when high-level languages such as CUDA or OpenCL are used. In the case of the Seed Extension kernel and the Smith-Waterman algorithm analyzed here, the CUDA and OpenCL languages required significantly more lines of code to express these algorithms, resulting in a code size increase of a factor 5-7x. The VHDL implementation required even 40x more lines of code. The design-time for these implementations was similarly longer, with an implementation time between 1-3 months for the CUDA and OpenCL implementations, to more than half a year for the VHDL implementation, due to the increased complexity of implementation and testing.

The final aspect under consideration is the influence of read sequence length on the efficiency and performance of the implementations. This is relevant since the sequence length of the raw output generated by DNA sequencers is expected to increase. An analysis of the GPU implementations shows that read length greatly impacts the performance of this implementation, as the number of parallel jobs that can be executed simultaneously is largely determined by the memory requirements for each job, which increase for longer sequence lengths. This highlights the need for improvements to the GPU architecture, for example by supporting data types better suited to applications such as these that require very low precision data formats.

5.2. FUTURE WORK

The following topics can be used as inspiration for future research, as they would make the systolic array architecture more suitable for adoption by other applications:

EXTENSION TO HIGHER-DIMENSION SYSTOLIC ARRAYS

The work in this dissertation has restricted itself to linear, or 1D, systolic arrays, as this is the type of systolic array used to accelerate the Seed Extension kernel and the Smith-Waterman algorithm. However, it would be interesting to extend the work to 2D systolic arrays. These are used, amongst others, to accelerate the GATK HaplotypeCaller, which is another popular bioinformatics application, and for matrix multiplication. Here, the potential gains may be even more profound, as a 2D systolic array would suffer even more from PEs being idle during the startup and shutdown phases.

DYNAMICALLY-ADJUSTABLE SYSTOLIC ARRAY CONFIGURATIONS

The efficiency of the Variable Logical Length, Variable Physical Length, and Variable Physical and Logical Length designs is data dependent and a static analysis of the input data is required to calculate their optimal design. Dynamic runtime reconfiguration could be used, based on a sampling of input data, to automatically adjust the configuration to optimally conform to the input data set. Partial reconfiguration could be used to guarantee a minimum level of service while other parts of the FPGA are being reconfigured to the optimal configuration.

List of Publications

Journal Articles

1. **E.J. Houtgast**, V.M. Sima, K.L.M. Bertels, and Z. Al-Ars, *Hardware Acceleration of BWA-MEM Genomic Short Read Mapping for Longer Read Lengths*, Computational Biology and Chemistry 75, p54-64, 2018.

2. **E.J. Houtgast**, V.M. Sima, K.L.M. Bertels, and Z. Al-Ars, *An Efficient GPU-Accelerated Implementation of Genomic Short Read Mapping with BWA-MEM*, ACM SIGARCH Computer Architecture News 44 (4), p38-43, 2017.

Conference Papers

1. **E.J. Houtgast**, V.M. Sima, K.L.M. Bertels, and Z. Al-Ars, *Comparative Analysis of System-Level Acceleration Techniques in Bioinformatics: A Case Study of Accelerating the Smith-Waterman Algorithm for BWA-MEM*, 18th International Conference on Bioinformatics and Bioengineering (BIBE), Oct 2018.

2. **E.J. Houtgast**, V.M. Sima, and Z. Al-Ars, *High Performance Streaming Smith-Waterman Implementation with Implicit Synchronization on Intel FPGA using OpenCL*, 17th International Conference on Bioinformatics and Bioengineering (BIBE), Oct 2017.

3. **E.J. Houtgast**, V.M. Sima, K.L.M. Bertels, and Z. Al-Ars, *Power-Efficiency Analysis of Accelerated BWA-MEM Implementations on Heterogeneous Computing Platforms*, International Conference on Reconfigurable Computing and FPGAs (ReConFig), December 2016.

4. **E.J. Houtgast**, V.M. Sima, G. Marchiori, K.L.M. Bertels, and Z. Al-Ars, *Power-Efficient Accelerated Genomic Short Read Mapping on Heterogeneous Computing Platforms*, 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), May 2016.

5. **E.J. Houtgast**, V.M. Sima, K.L.M. Bertels, and Z. Al-Ars, *GPU-Accelerated BWA-MEM Genomic Mapping Algorithm Using Adaptive Load Balancing*, 29th International Conference on Architecture of Computing Systems (ARCS), Apr 2016.

6. **E.J. Houtgast**, V.M. Sima, K.L.M. Bertels, and Z. Al-Ars, *An FPGA-Based Systolic Array to Accelerate the BWA-MEM Genomic Mapping Algorithm*, 15th International Conference on Embedded Computer Systems (SAMOS), Jul 2015.

OTHER WORK

1. N. Ahmed, V.M. Sima, **E.J. Houtgast**, K.L.M. Bertels, and Z. Al-Ars, *Heterogeneous Hardware/Software Acceleration of the BWA-MEM DNA Alignment Algorithm*, 34th International Conference on Computer Aided Design (ICCAD), Nov 2015.

2. S. Isaza, **E.J. Houtgast**, and G.N. Gaydadjiev, *Sequence Alignment Application Model for Multi- and Manycore Architectures*, International Journal of Information and Computer Security (IJICS), Sep 2011.

3. S. Isaza, **E.J. Houtgast**, and G.N. Gaydadjiev, *HMMER Performance Model for Multicore Architectures*, 14th Euromicro Conference on Digital System Design (DSD), Sep 2011.

4. S. Isaza, **E.J. Houtgast**, F. Sanchez, A. Ramirez, and G.N. Gaydadjiev, *Scaling HMMER Performance on Multicore Architectures*, Complex, Intelligent and Software Intensive Systems (CISIS), July 2011.

5. **E.J. Houtgast**, *Scalability of Bioinformatics Applications for Multicore Architectures*, MSc Thesis, Nov 2009.

6. Z. Wartell, **E.J. Houtgast**, O.P. Pfeiffer, C.D. Shaw, W. Ribarsky, and F. Post, *Interaction Volume Management in a Multi-Scale Virtual Environment*, Advances in Information and Intelligent Systems (AIIS), 2009.

7. **E.J. Houtgast**, O.P. Pfeiffer, Z. Wartell, W. Ribarsky, and F. Post, *Navigation and Interaction in a Multi-Scale Stereoscopic Environment*, Virtual Reality (VR), 2005.

Curriculum Vitæ

Ernst Joachim HOUTGAST

HOUTGAST was born in 's-Gravenhage, the Netherlands in 1982. He received his Bachelor of Science degree in Computer Science and his Master of Science degree in Computer Engineering from Delft University of Technology. After working for a number of years in finance at Rabobank and Optiver, he decided to return to his engineering roots. He briefly worked at Maxeler Technologies before joining the startup in high performance genomics Bluebee, while at the same time returning to his alma mater Delft University of Technology as a PhD candidate within the Computer Engineering department. His PhD work has been performed under the supervision of Dr. Zaid Al-Ars and Prof. Koen Bertels. His research interests include high performance computing, heterogeneous architectures, and bioinformatics. He currently works as a patent examiner at the European Patent Office in Rijswijk, the Netherlands. Houtgast is a Senior Member of the IEEE.