DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ACCELERATING BIOINFORMATICS AND BIOMEDICAL APPLICATIONS VIA MASSIVELY PARALLEL AND RECONFIGURABLE SYSTEMS

DOCTOR OF PHILOSOPHY DISSERTATION

AGATHOKLIS PAPADOPOULOS

2014

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ACCELERATING BIOINFORMATICS AND BIOMEDICAL APPLICATIONS VIA MASSIVELY PARALLEL AND RECONFIGURABLE SYSTEMS

DOCTOR OF PHILOSOPHY DISSERTATION

AGATHOKLIS PAPADOPOULOS

A Dissertation Submitted to the University of Cyprus in Partial Fulfilment of the Requirements for the Degree of Doctor of Philosophy

October, 2014

VALIDATION PAGE

Doctoral Candidate: Agathoklis Papadopoulos

Doctoral Thesis Title: Accelerating Bioinformatics and Biomedical Applications via Massively Parallel and Reconfigurable Systems

The present Doctoral Dissertation was submitted in partial fulfilment of the requirements for the Degree of Doctor of Philosophy at the Department ***of Electrical and Computer Engineering*** *and was approved on* ***22 October 2014*** *the by the members of the* ***Examination Committee***.

Examination Committee:

Committee Chair: _____

(Constantinos Pitris, Associate Professor, University of Cyprus)

Research Supervisor: _____

(Theocharis Theocharides, Assistant Professor, University of Cyprus)

Committee Member: _____

(George Ellinas, Associate Professor, University of Cyprus)

Committee Member: _____

(Constantinos Pattichis, Professor, University of Cyprus)

Committee Member: _____

(Vasilis Promponas, Assistant Professor, University of Cyprus)

Committee Member: _____

(Apostolos Dollas, Professor, Technical University of Crete)

DECLARATION OF DOCTORAL CANDIDATE

The present doctoral dissertation was submitted in partial fulfillment of the requirements of the degree of Doctor of Philosophy of the University of Cyprus. It is a product of original work of my own, unless otherwise mentioned through references, notes, or any other statements.

Agathoklis Papadopoulos

_____

ΠΕΡΙΛΗΨΗ

Οι συνεχείς προσπάθειες για βελτίωση της παρεχόμενης ιατροφαρμακευτικής περίθαλψης μέσω καλύτερων τεχνικών διάγνωσης, παρακολούθησης και θεραπείας και κατά συνέπεια της βελτίωσης του βιοτικού επιπέδου της σύγχρονης κοινωνίας, δεσμέυονται άρρηκτα με τα υπάρχοντα υπολογιστικά σύστηματα υψηλής απόδοσης. Τα συστήματα αυτά είναι απαραίτητα για την εκτέλεση των υπολογιστικά απαιτητικών εφαρμογών βιοπληροφορικής και βιοϊατρικής που χρησιμοποιούν ερευνητές παγκόσμια για να δημιουργήσουν καλύτερα φάρμακα, νέες μη-επεμβατικές τεχνικές διάγνωσης και υψηλότερης ακριβείας μεθόδους πρόληψης. Τα συστήματα αυτά όμως βασίζονται σε ακριβές και υψηλής ενεργειακής κατανάλωσης τεχνολογίες, όπως φάρμες υπολογιστών και συμπλέγματα ομογένων υπολογιστικών κόμβων. Νέες αναδυόμενες τεχνικές επιτρέπουν πλέον την εγκατάσταση μονάδων αναδιατασσόμενης λογικής (FPGAs) και μονάδων επεξεργασίας γραφικών (GPUs), που μπορούν να συνεργάζονται με κλασσικούς επεξεργαστές (CPUs) για να πετύχουν ακόμη καλύτερα αποτελέσματα. Τα ετερόγενη υπολογιστικά συστήματα μπορούν να συνδυάσουν την σχεδιαστική ευελιξία των μονάδων αναδιατασσόμενης λογικής και την υψηλή ταχύτητα υπολογιστών των μονάδων επεξεργασίας γραφικών με την ευχρησία των πολυνυματικών επεξεργαστών. Οι εφαρμογές βιοπληροφορικής και βιοϊατρικής μπορούν να κερδίσουν υψηλότερες επιδόσεις εκμεταλλευόμενες τις δυνατότητες των συστημάτων αυτών.

Η διατριβή αυτή παρουσιάζει την μεθοδολογία που έχει αναπτυχθεί για να αξιολογήσει τις πραγματικές δυνατότητες των πολυνυματικών επεξεργαστών, των μονάδων αναδιατασσόμενης λογικής και των μονάδων επεξεργασίας γραφικών ως υπολογιστικούς επιταχυντές για εφαρμογές βιοπληροφορικής και βιοϊατρικής. Στα πλαίσια της διατριβής αυτής, έχουν υλοποιηθεί κώδικες λογισμικού και αρχιτεκτονικές υλικού της εφαρμογής CAST (ανάλυση περιοχών χαμηλής πληροφορίας σε πρωτεΐνες) για όλες τις τεχνολογίες επιτάχυνσης για να έχουμε δίκαιη σύγκριση κάθε τεχνολογίας με τις υπόλοιπες. Στόχος είναι να παρουσιαστούν οι δυνατότητες και οι περιορισμοί της κάθε νέας τεχνολογίας όταν επιλεχθούν ως υπολογιστικές μονάδες για τις εφαρμογές - CAST κ.α. -που μελετήσαμε. Η διατριβή κλείνει με την παρουσίαση της πρώτης υβριδικής CPU/GPU/FPGA πλατφόρμας για εφαρμογές βιοπληροφορικής και βιοϊατρικής. Τα ετερογένη αυτά υβριδικά συστήματα, προβλέπονται να προσφέρουν πολύ περισσότερη ευελιξία και υπολογιστική ισχύ σε χαμηλότερη κατανάλωση ενέργειας από τα υφιστάμενα υπολογιστικά συστήματα.

ABSTRACT

The quest for advancement in healthcare diagnosis, monitoring and therapy, all key aspects of improving the quality of life, becomes heavily dependent on high-performance, real-time computation systems. These systems are necessary to perform extremely complicated algorithms used by bioinformatics engineers and doctors worldwide that will lead to the development of more effective drugs, non-invasive diagnostic techniques and preventive medicine. Such technology though, comes in the form of homogeneous supercomputers or large clusters, which are large, expensive and energy devouring, as they rely on software flexibility to facilitate the execution of multiple applications. However, recent advancements allow the integration of reconfigurable hardware (FPGAs) and graphics processing units (GPUs), alongside with general-purpose processors (CPUs). Such heterogeneous systems can combine the flexibility of the FPGA hardware fabric, the raw speed of GPUs, and the generality of CPUs. Biomedical and bioinformatics applications can therefore take advantage of the inherent flexibility and available parallelism, with significant performance gains.

This thesis presents a methodology for designing and evaluating multi-core CPUs, GPUs, and FPGAs as parallel and reconfigurable accelerators for bioinformatics and biomedical applications. Software implementations and hardware architectures for a selected case study application (CAST for low-complexity region analysis in proteins) were designed and implemented for all computing engines. We highlight the benefits and issues of our parallelization methodology through an overview of the challenges we faced when mapping CAST and the other selected case study applications on each computing platform. A multi-threaded software version of CAST was implemented to fairly evaluate state-of-the-art multi-core CPUs against GPUs and FPGAs. Then computation structures were developed for accelerating CAST on GPUs. Furthermore, a key contribution of this thesis is to show how reconfigurable hardware platforms can be utilized to achieve higher performance per watt for bioinformatics applications. Multi-FPGA implementations were explored as well. Lastly, a guide to build a hybrid CPU/GPU/FPGA heterogeneous system for bioinformatics and biomedical applications is provided, as it is anticipated that the integration of FPGAs and GPUs alongside with multicore CPUs, can provide a blend of flexibility and processing power, an advantage not found in existing systems.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my thesis advisor, Dr. Theocharis Theocharides for his excellent guidance and support in my research all these years. Also, I would like to express a warm thank you to the members of the examination committee: Dr. Ellinas, Dr. Pitris, Dr. Pattichis, Dr. Dollas and Dr. Promponas, for their kind support, feedback and insights which helped me achieved this thesis.

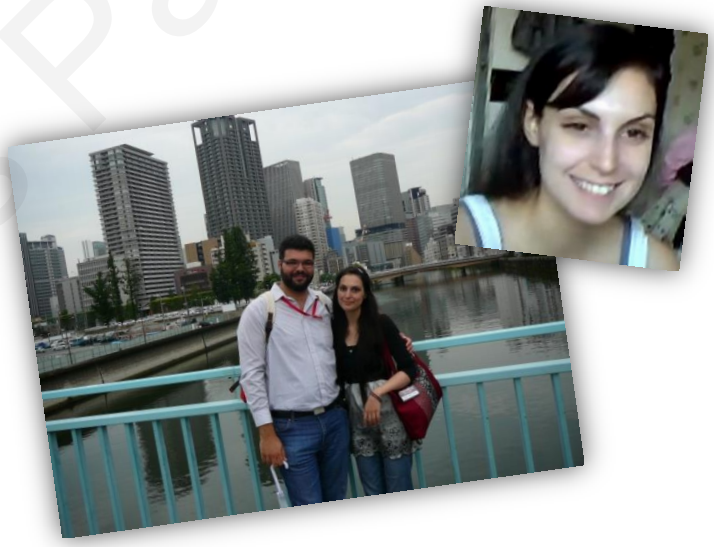
My many thanks to my research collaborators: Dr. Vasilis Promponas, Dr. Manolis Christodoulakis and Ioannis Krirmitzoglou, as our collaboration allowed me to complete the multi-disciplinary research work presented in this thesis.

Also, I express my gratitude to my lab-mates and partners: Christos Kyrkou and Christos Ttofis for their teamwork spirit and mostly for their unconditional friendship. I would like to thank Demetrios Eliades as well, for all of our meaningful discussions over the years.

Last but not least, a warm thank you to my wonderful wife Maria, my parents and family for their unconditional love and continuous support.

Dedication

This dissertation is dedicated to my beloved wife Maria, being the inspiration in my life!
No words are enough to express my feelings of gratitude
towards the people closest to my heart.



Biography: Agathoklis Papadopoulos received his 5-year Engineer Diploma (Dipl.-Ing.) in Electrical and Computer Engineering from National Technical University of Athens in 2009. During his studies at NTUA, he specialized at the fields of Hardware Design, Microprocessor and Microcontroller Systems, Database Systems and Bioengineering. Since 2009 Agathoklis is pursuing his PhD in Computer Engineering at the University of Cyprus, under the supervision of Dr. Theocharis Theocharides.

His research interests include embedded systems design and application-specific hardware systems for biomedical and bioinformatics applications. He is involved in various projects funded by the European Commission and the Research Promotion Foundation of Cyprus. Agathoklis currently is a researcher at KIOS Research Center for Intelligent Systems and Networks and a member of Embedded & Application Specific Systems-On-Chip Laboratory (EASoC).

Publications stemming from this thesis

Journal Articles

[J1] **A. Papadopoulos**, T. Theocharides et al. " Towards a Hybrid CPU/GPU/FPGA Platform for Accelerating Bioinformatics and Biomedical Applications", *IEEE Design and Test Magazine, under review*

[J2] T. Theocharides, **A. Papadopoulos**, et al. "Reconfiguring the Bioinformatics Computational Spectrum: Challenges and Opportunities of FPGA-Based Bioinformatics", *IEEE Design and Test Magazine: Special Issue on Hardware Acceleration in Computational Biology, Vol.31, Issue.1, pp.62-73,* IEEE, 2014

[J3] **A. Papadopoulos**, I. Kirmitzoglou, V. J. Promponas, T. Theocharides, "FPGA-based hardware acceleration for local complexity analysis of massive genomic data*," HardBio2012 - Special Issue on Hardware for Bioinformatics Applications, Integration, The VLSI Journal,* Vol.46, Issue.3, pp.230-239, Elsevier, 2013

Conference Proceedings

[C1] **A. Papadopoulos**, I. Kirmitzoglou, V.J. Promponas, T. Theocharides, " GPU technology as a platform for accelerating local complexity analysis of protein sequences", 35th Annual International Conference of the IEEE Engineering in Medicine and Biology Society - EMBC2013, Osaka, July 2013

[C2] G. Chrysos, E. Sotiriades, C. Rousopoulos, A. Dollas, **A. Papadopoulos**, I.Kirmitzoglou, V.Promponas, T.Theocharides, G. Petihakis, J. Lagnel, P. Vavylis, G. Kotoulas, "Opportunities from the Use of FPGAs as Platforms for Bioinformatics Algorithms", *12th IEEE International Conference on Bioinformatics and Bioengineering - BIBE2012,* November 2012

[C3] **A. Papadopoulos**, V.J. Promponas, T. Theocharides, "Towards systolic hardware acceleration for local complexity analysis of massive genomic data", ACM/IEEE Great Lakes Symposium on VLSI - GLSVLSI 2012, May 2012

Workshops & Conferences without Proceedings

[W1] **A. Papadopoulos**, T. Theocharides, "Towards a low-cost GPU/FPGA hybrid computation platform for bioinformatics and biomedical applications", 2nd Hellenic IEEE Student Branch & GOLD Conference - IEEE HSBC2013, Nicosia, November 2013

[W2] **A. Papadopoulos**, V.J. Promponas, T. Theocharides, "GPU_CAST: GPU technology as a platform for accelerating low complexity region detection in protein sequences", 7th Symposium of the Hellenic Society for Computational Biology & Bioinformatics - HSCBB12, Heraklion, October 2012

[W3] **A. Papadopoulos**, V.J. Promponas, T. Theocharides, "FPGA-accelerated CAST for low-complexity region detection in amino acid sequences", 6th Symposium of the Hellenic Society for Computational Biology & Bioinformatics - HSCBB11, Patras, October 2011

Other Publications

Journal Articles

[J4] D. Koukounis, C. Ttofis, **A. Papadopoulos**, T. Theocharides, A High Performance Hardware Architecture for Portable, Low-Power Retinal Vessel Segmentation*," Integration, The VLSI Journal, Vol.47, Issue.3, pp.377-386,* Elsevier, 2014

[J5] C. Ttofis, **A. Papadopoulos**, T. Theocharides, M. Michael, D. Doumenis, "An MPSoC-based QAM modulation architecture with Run-Time Load Balancing," *EURASIP Journal of Embedded Systems,* vol. 2011, 2011

Conference Proceedings

[C4] **Agathoklis Papadopoulos**, Theocharis Theocharides, Maria K. Michael "Towards Optimal CMOS Lifetime via Reliability Modeling and multi-Objective Optimization," Proceedings of IEEE International *Symposium on Circuits and Systems - ISCAS 2011*, Rio de Janeiro, May 2011

[C5] Christos Ttofis, **Agathoklis Papadopoulos**, Theocharis Theocharides "A Reconfigurable MPSoC-based QAM Modulation Architecture," Proceedings of the 18th IEEE/IFIP International Conference on VLSI and Systems-on-Chip - VLSI-SoC, Madrid, Sept 2010

Other Publications

[O1] **A. Papadopoulos**, (Advisor: K. Pekmestzi), "Design and Implementation of a domestic power monitoring embedded system using ZigBee technology", Engineering Diploma Thesis (in Greek language), National Technical University of Athens, Greece, July 2009 http://artemis.cslab.ntua.gr/Dienst/UI/1.0/Display/artemis.ntua.ece/DT2009-0110

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

1.1 Biology and Medicine: An Ever Increasing Need for Computational Performance

Genomics – the study of the complete genomes of biological species (including human) – has revolutionized the way biological research is currently performed. This is further evident with the novel high-throughput sequencing technologies that enable whole genome DNA sequencing of simple unicellular organisms in a matter of days [1]. Moreover, anticipating the "$1000 genome", researchers expect that genomic science will soon become a central part of clinical practice [2].

Bioinformatics algorithms traditionally require extensive computational resources. Following the revolutionary advent of molecular biology, life sciences are being transformed to a quantitative, data-rich scientific domain. A new form of high-throughput biology has emerged, mainly based on large volumes of heterogeneous data regarding biological macromolecules (e.g. nucleotide and protein sequence data, protein structures, gene expression data) and bio-systems in general. The unprecedented volume of biological data needs to be decoded and transformed to knowledge regarding biological systems at multiple levels. The aforementioned goals have established bioinformatics and computational biology in a central position of modern biological sciences [3]. Nevertheless, the overwhelming accumulation of data, paired with the introduction of novel data types and biological questions of increasing complexity, continuously challenges established computational approaches in handling

biological data. Inevitably, the so-called next-generation biology can only be realized using next-generation computing paradigms.

As huge amounts of sequence data are currently being produced worldwide at an increasing pace, extensive downstream computational analysis is required. Typical computational pipelines for genomics feature a computationally intensive sequence comparison component; this is justified by the empirical observation that genes and proteins with similar sequences usually perform similar functions. Therefore, sequence similarity search serves for inferring functional and structural analogy for biological macromolecules [4][5].

Current computational challenges in the field of computational biology and bioinformatics stem from diverge sub-disciplines, including genomics, proteomics and structural biology. Using as an illustrative example the modern (next-generation) sequencing platforms, a single experiment produces *terabytes* of raw data. Initial analysis, followed by sequence assembly for *de novo* genome sequencing, genome re-sequencing or transcription profiling, and downstream annotation, require dedicated computational infrastructure. To further extract more information out of these data, researchers use comparative genomics approaches via complete genome alignments, exhaustive pairwise sequence similarity searches, and/or phylogenetic reconstructions; all are resource-intensive tasks in terms of data processing, memory access and storage.

With the cost of sequencing dropping to less than $0.1 per Megabase of DNA sequence [6], the doubling time of the volume of nucleotide sequence data is estimated to be approx. 5 months, while the same figure for data storage capacity can be estimated to be 3 times higher [3]. To cope with this data deluge, the bioinformatics community constantly seeks novel efficient tools (e.g. hardware and software) or even entirely revolutionary computing platforms. Such data-driven computation therefore could only be achieved in high-end supercomputers which traditionally have been developed as high-performance computing clusters and multiprocessor systems [7]. A notable example is IBM's BlueGene supercomputer, designed solely for bioinformatics [8]. However, the specialized nature of algorithms targeting bioinformatics, limits the number of end-users for such platforms, thus the costs of building and maintaining such supercomputers tend to be extremely high. As such, alternative technologies that can better balance the cost and performance constraints can be more efficient for targeting the bioinformatics research communities.

Medicine is considered to be one of the most important fields of humanity's knowledge. Advancements in this field not only enrich our view of the world around and inside of us, but further improves our longevity and our quality of life through new diagnostic methods, new treatments and new drugs. Biomedical engineering is the field of life sciences that attempts to fill the gap between traditional medicine and engineering. Biomedical applications efficiently combine the problem solving and solution designing of engineering with the knowledge stemming from medicine and biological sciences in order to advance healthcare, including diagnosis, monitoring and therapy [9].

Biomedical applications cover a broad band of fields: from commonly recognisable and wide-spread medical imaging techniques such as MRI [10], to physiological systems modelling where researchers attempt to develop models which will allow for fine-tuning new drug production or better understand the ways a specific medical condition or disease progress in a human body[11]. Different biomedical applications use varied raw data types and diverse processing techniques. For example, medical imaging uses computer imaging methods and psychological system modelling uses computational intelligence. However, most of biomedical applications have to conduct computationally heavy processing over the available data to extract the desired medical knowledge.

This processing mainly is performed on clusters [7]. The number of nodes required for each biomedical application depends on the application constrains (for example, MRI images should be generated from the magnetic reads in near real-time) and the amount of raw data to be processed. The high costs of installing and maintaining such clusters can lead to researchers straggle to extract useful information from computing clusters with inadequate resources due to financial issues. Moreover, the heavy computational load makes most of biomedical applications to be considered power hungry.

Emerging technologies such as general purpose computing on graphics processing units (GPUs) and domain-specific hardware accelerators running on field-programmable gate arrays (FPGAs) are prime candidates for improving performance of bioinformatics and biomedical applications. GPGPU paradigm offers a powerful alternative to traditional cluster computing, as it allows for executing instructions on massive amounts of data in parallel. A GPU-based implementation of the BLAST algorithm showed that execution time can be cut to a fourth of the time needed on a multicore system [12]. Moreover, the GPGPU programming paradigm

follows a similar coding scheme as traditional multi-threading/cluster coding, making it easy for developers to migrate from clusters to GPGPU.

However, reconfigurable hardware implementations have been proven capable of outperforming high-end GPU-based systems, even when running on low-budget FPGA boards [13][14], especially when taking the power consumption into consideration. Despite clock speeds that are typically 1/10th of those in cluster and traditional computing [15], by exploiting parallelism at all levels and through custom-designed hardware blocks, speedups of up to several orders of magnitude can be achieved. These speedups are gained over execution of the same algorithms on clusters.

FPGAs can greatly benefit biomedical and bioinformatics applications since even a 2x speedup are considerable due to the immense computational times of these algorithms. A typical protein sequence alignment can take several days to perform over proteomic databases consisting of millions of sequences. FPGA-based reconfigurable systems have the potential to cut the required processing time to just a few hours. The cost per computation and joules per computational load (or watts during computation time) are quite favorable for reconfigurable computing, and hence it is worth examining as a platform for bioinformatics and biomedical applications. Given the cost-performance benefits of FPGAs, recent advances in computational biology suggest that reconfigurable domain-specific hardware might be indeed a powerful alternative to cluster-based supercomputers [16].

In this thesis, we are not trying to answer the question: "GPUs or FPGAs to replace traditional computing platforms?". We strongly believe that the future holds something new and interesting. Through our work on massively parallel accelerators for bioinformatics and biomedical applications, we arrived to the conclusion that hybrid systems that combine general-purpose computers, alongside with Graphics Processing Units (GPUs) and FPGAs can offer more benefits than systems having a single type of accelerators. We built and evaluated such a hybrid platform which emphasizes on the availability of a conventional CPU, two highly parallel GPUs and multiple FPGA devices. All computation engines are interconnected with the system's main memory via high-speed data transfer protocols. The proposed system offers integrated solutions for the execution of I/O- and memory-intensive problems, in which the FPGAs and the GPUs form tightly coupled co-processors to the conventional one.

This thesis discusses the opportunities for solving biomedical- and bioinformatics-related data-intensive problems on large-scale multi-FPGA systems, as well as highlighting open research challenges dealing with hybrid systems comprised of FPGAs and GPUs alongside powerful multicore CPUs. We strongly believe that reconfigurable hybrid systems can be the basis for future bioinformatics hardware acceleration, providing high performance within a minimal carbon footprint. Most of our work have been presented in [17][18][19][20][21][22]; this thesis features a thorough discussion related to the characteristics of bioinformatics algorithms and the continuous quest for advanced computing platforms, and describes why reconfigurable hardware architectures and GPU-based implementations can become a viable, if not dominant platform. Moreover, this thesis discusses how large-scale reconfigurable hybrid CPU/GPGPU/FPGA platforms can become an even better solution.

1.2 Thesis Scope and Contributions

This thesis attempts to use massively parallel and reconfigurable systems as accelerators for bioinformatics and biomedical applications. State-of-the-art parallel computation engines such as multicore CPUs, GPUs and FPGAs can potentially be used for accelerating bioinformatics and biomedical applications. Our results showed that there is no single answer whether one engine is better than the other. As such, hybrid high-performance computing is considered as an acceleration option in the last chapters of this thesis. Previous works on heterogeneous massively parallel systems usually focus on using FPGAs as co-processors for executing computationally intensive segments of the applications [23]. The last contribution of this multi-disciplinary thesis, is the evaluation of our own custom-built hybrid CPU/GPU/FPGA heterogeneous system as a massively parallel engine for bioinformatics and biomedical applications.

First, we had to establish a strong partnership with accredited bioinformatics and biomedical research groups in order to identify the computational needs in their areas of expertise. Algorithms having the most common computation characteristics of their area where selected through our meetings and thorough literature reviews. These algorithms had their traits analysed as well in order to determine how hybrid high-performance computing can help in increasing their performance while maintaining a green energy footprint.

We have developed a number of multi-threaded, GPU-based and FPGA-based versions of the selected case study algorithms in order to fairly evaluate how each computing platform can benefit them. Moreover, during this thesis we followed a strategy for performing a fair comparison of multicore CPUs, GPUs and FPGAs by implementing software and hardware versions of the CAST algorithm for *all* state-of-the-art processing engines.

A short list of our implementations includes:

- We implemented and later optimized a multi-threaded version of the CAST bioinformatics algorithm with improved performance up to 20x against the original software tool.

- We investigated the GPGPU programming paradigm by implementing a CUDA-enabled GPU-based version of CAST. GPU_CAST showed speedups up to 11x compared to the multi-threaded version. Also, this GPU-based implementation showed that even in worst case scenario, it can perform at least on par with the multi-threaded version. The limitations of the GPGPU paradigm – such as memory transfer overheads - when used in the field of bioinformatics were investigated as well.
- An FPGA-based hardware architecture for the CAST algorithm has been designed and implemented to be compared against the CAST implementations for other state-of-the-art computation technologies in order to extract information about the benefits and issues of using FPGAs as accelerators. This information we believe can be used as a roadmap for accelerating other applications in biology and medicine.
- An FPGA-based prefix doubling suffix array construction application has been implemented in order to evaluate reconfigurable hardware architectures for de novo DNA sequence assembly.

Additionally, we have explored the scalability of the proposed FPGA-based architectures and FPGA-based CAST has been evaluated by implementing a quad- and an octal-unit system. The necessary PCI-Express I/O interfaces were developed as well, in order to test the system under real world scenarios.

Last but not least, a prototype hybrid heterogeneous CPU/GPU/FPGA computing platform have been designed and built for evaluating our implementations and tools. This high-end system facilitates multiple FPGAs and two high-end GPU processing cards alongside an Intel Xeon processor. A real world benchmark bioinformatics application was evaluated on this high-end hybrid system and allowed us to identify the benefits and issues of using such a system as a massively parallel and reconfigurable platform for accelerating bioinformatics and biomedical applications.

1.3 **Thesis Organization**

This thesis is organized as follows: Chapter 2 is a discussion on how we can achieve higher performance through parallelization of the selected bioinformatics and biomedical applications. An introduction to the biological problem each of the selected algorithms is attempting to solve is provided as well. Chapter 3 includes an introduction to parallel computing and has a high-level description of the current state-of-the-art computational engines like multi-core processors, GPUs and FPGAs and discusses their uses in accelerating bioinformatics applications through its related work section.

In Chapter 4, the use of parallel programming on multi-core processors is explored for accelerating local complexity analysis algorithms. In Chapter 5, GPU_CAST is presented as an attempt to utilize the GPU programming paradigm though the CUDA framework as an acceleration engine. A GPU-based psychological system model (LVN modelling for sub thalamic deep brain stimulation) is presented as well. The benefits, as well as the issues, stemming from using graphic cards for accelerating bioinformatics and biomedical applications are then discussed.

In Chapter 6, the FPGA-based hardware architectures for accelerating low-complexity regions masking in protein sequences and a proof-of-concept architecture for implementing a part of a de novo DNA sequence assembly pipeline, are described and discussed. The CAST FPGA-based architecture is expanded and tested as a multi-FPGA application. In Chapter 7, we describe our attempt to build a hybrid CPU/GPU/FPGA computing platform for accelerating biology and medicine applications.

Chapter 8 holds our research plans for the future and how the results presented in this thesis can be used to further advance computing engines for bioinformatics. This chapter concludes this thesis by summing up its contributions, results and gained knowledge.

Appendix A gives a complete overview of the proteomic benchmarks used to evaluate the work in this thesis. Appendix B provides information on available communication schemes which allow data communication between the different processing units of hybrid systems.

2

Challenges on Accelerating Bioinformatics and Biomedical Applications

In this chapter, we will discuss applications we selected as case studies since among them share the common traits of the popular sequence analysis domain of bioinformatics. These traits include but not limited to FASTA format input, dynamic programming, multi-step processing of massive amounts of data, use of optimization methodologies etc. Moreover, a application from the computationally-heavy psychological system modelling biomedical domain is selected as case study as well.

The selected case study applications - Local Complexity Analysis of protein sequences, Subthalamic Nucleus Deep Brain Stimulation Modelling for Parkinson's disease and de novo DNA sequence assembly - are introduced each in a different section of this chapter. The most common and popular algorithms designed to solve each application are presented and their characteristics are discussed as well. Moreover, we will demonstrate how we can improve the performance of these algorithms by providing a step-by-step analysis of how parallelization techniques can be used for accelerating them on state-of-the-art computation engines.

2.1 **Local Complexity Analysis**

Sequence comparison is usually performed by aligning sequences, i.e. by algorithmically identifying the optimal correspondence of individual positions (residues) between the compared sequences, given a scoring scheme. Alignment-based pair-wise comparison of biological macromolecular sequences is a routine computational procedure practiced daily in

most biological research laboratories throughout the world. Thus, a large set of tools has been developed over the years trying to provide improved solutions to the sequence comparison problem over the traditional dynamic programming algorithms. Rapid and sensitive algorithms, such as the BLAST heuristic algorithm [24], have become the most widely used tools for homology searches in sequence databases. The wide utilization of the BLAST suite of programs is clearly reflected by the fact that the two key methodological papers describing the methods [24][25] have been collectively cited more than 78000 times [26].

When a newly identified protein sequence is submitted for comparison against a database of already known proteins, the similarity score alone is not sufficient to pinpoint important biological relationships for functional inference. Thus, robust statistical computations [27] have been used to reliably identify those similarities that are unlikely to have arisen simply by chance in a haystack of unrelated hits. Such measures involve complex data processing, with unpredictable data flow behavior; data dependencies involved in such algorithms are therefore a significant drawback when employing traditional superscalar and multi-threaded or multicore CPU architectures. Furthermore, memory bandwidth and memory management is another issue as well.

From a biological perspective, some of the basic assumptions related to such applications do not hold for a significant fraction of known proteins. For example, for obtaining reliable estimates of the statistical significance of scores observed in a series of pair-wise sequence comparisons, it is postulated that the compared sequences are random; they are generated by sampling from an amino acid residue distribution, i.e. the distribution of the protein sequence database. However, real biomolecular sequences deviate from this “ideal” distribution, especially in cases where skewed local composition is observed [28][29]. The latter observation was based on a working definition of the so-called Low Complexity Regions (LCRs) in amino acid proteomic sequences. More specifically, a measure of local compositional complexity was introduced, calculated by applying an entropy-like computation on composition vectors. This approach was followed in the SEG suite of programs as a two-pass procedure, where candidate LCRs are identified during the first pass, followed by an optimization step [29].

Based on the concept of LCRs, important improvements have been suggested by the bioinformatics community for improving the computational behavior of sequence comparison

algorithms. In particular, identification of LCRs is usually followed by a procedure known as masking, with the objective of cancelling out the effect of local compositional bias on scoring protein sequence comparisons. It is worth mentioning that proteins with LCRs seem to play important roles in several natural biological processes (including human disease) [30][31]. However, this thesis mainly focuses on the application of massive LCR detection in large protein datasets for masking, as a pre-processing step in sequence database search.

Alternative approaches for LCR detection and masking have been proposed, although not necessarily based on the same definition. A popular method is the CAST algorithm [32]. Masking protein sequences with CAST has been empirically shown to result into similarity searches of superior specificity (low false positive rate) without sacrificing sensitivity [32]. However, a major obstacle against the wider use of the CAST algorithm is its relatively low computational performance (compared to SEG) due to its iterative nature. With the exponential growth of sequence databases [33], even the fastest algorithms for sequence comparison fall short.

2.1.1 CAST Algorithm

The CAST algorithm [32] is an iterative method for identifying and masking LCRs in protein sequences and has shown higher quality results over other proposed methods, such as SEG [29]. In principle, the CAST algorithm compares protein sequences against an artificial database consisting of 20 degenerate protein sequences of arbitrary length, each one being a homopolymer based on one of the 20 natural amino acid residue types. CAST identifies LCRs in a single linear pass for each amino acid type, with memory and computations required for each pass being linear to the input size. This feature is a consequence of the careful reformulation of the Smith-Waterman local sequence alignment algorithm [34], by taking into account that homopolymers do not carry positional information, and gaps are not permitted. The most remarkable feature of CAST is that not only it detects LCRs, but also identifies the type of amino acid residue causing the bias. Thus, selective masking can be performed in a more subtle and specific manner compared to other masking methods (e.g. [29]).

The LCR detection step requires the use of a substitution matrix, where matching non-identical residue types may give positive scores as well. Thus, an LCR biased in one residue type may lead to high scores for biases of a similar but not identical type (e.g. arginine/lysine).

```
Algorithm CAST
Input: A protein sequence S
Output: The sequence masked for LCRs

residues  <- (A, C, D, .., Y)
hscore    <- (0, 0, 0, .., 0)
from      <- 0
to        <- 0
neutral   <- X

do
       for each res in residues
              (hscore[res], from, to) <- detectBias(S, res)

       hscoreMAX <- max(hscore[A], hscore[C], .., hscore[Y])
       lcrType <- residues[ argmax(hscore[A], hscore[C], .., hscore[Y]) ]

       if (hscoreMAX>=Threshold)
              S <- mask(S, lcrType, from, to)

while (hscoreMAX>=Threshold)

function detectBias(Sequence: S, Residue:r)
       (maxscore, from, to) <- alignSmithWaterman(S, poly-residue)
       return (maxscore, from, to)

function mask(Sequence: S, Residue:lcrType, Start:from, End:to)
       for pos in (from .. to)
              if (S[pos] equals lcrType)
                     S[pos] <- neutral
       return (S)
```

Figure 2.1: Pseudocode for the CAST algorithm.
LCR detection is performed by iteratively comparing input S with degenerate homopolymers (poly-residues) of the 20 naturaly occuring amino acids.

Therefore, CAST iteratively performs LCR detection and masking steps to prevent unnecessary masking due to cross-dependencies between amino acid residue types. An empirically defined threshold value *T* for the similarity score serves as a LCR selection criterion. With the use of the BLOSUM62[1] substitution matrix, the optimal value $T = 40$ is used. In practice, a variant of BLOSUM62 serves as the default scoring matrix: the scores of each residue type against the neutral type 'X', are computed as the mean value of the amino acid substitution scores for the respective residue type.

1 BLOSUM62 is one of the substitution matrices commonly used for calculating scores between evolutionarily divergent protein sequences.

While is possible to use any other substitution matrix, BLOSUM62 is the substitution matrix of choice for our work.

The algorithm shown in Figure 2.1 receives as input a protein sequence, and, searches for the LCR candidates (highest scoring segments- HSS) of each natural amino acid type. It then selects the HSS with the maximum score, and if that score is less than the threshold *T*, it ends outputting discovered LCRs; otherwise, it replaces each occurrence of the max scoring residue type in the highest scoring segment region with an 'X' (i.e. a neutral amino acid) and iterates through the updated sequence. For each discovered LCR its residue type, the sequential position (start and end) and computed score are reported. Further details of the algorithm can be found in [32].

2.1.2 CAST Analysis: How we Can Achieve Parallelization

We will be using CAST's pseudocode in Figure 2.1 for analysing its behaviour during execution and how we can extract information on the potential performance gains of its potential parallel implementations. In this section, important code will be highlighted and analysed for potential parallelization.

- *Line 7*: **for each** res **in** residues
  (hscore[res], from, to) ← **detectBias**(S, res)

CAST iterates and calculates *hscore* for every naturally occurring amino acid by calling the *detectBias* function on the input sequence *S*. It is clear to note that we can calculate *hscore* in parallel for each amino acid if we can access *S* in parallel. How we can meet the latter condition depends heavily on the selected computation platform we use for execution. For example, if enough memory per processing core is present, we can have multiple copies of *S*, one for each core. In this case, we can achieve improvement of this code section directly related to the number of available cores. In the case where more than twenty processing cores are available, resource allocation schemes need to be utilized in order to partitioned multiple input sequences in batches running on groups of twenty cores for even higher performance.

- *Line 9*: hscoreMAX←**max**((hscore[A], hscore[C], .., hscore[Y])
  Line 10: lcrType←residues[(**argmax**(hscore[A], hscore[C], .., hscore[Y])]

The next computation step requires to have available the *detectBias* results from all amino acids. As such, *hscoreMAX* can be calculated only after the loop of *Line 7* is completed. This code segment is executed sequentially after the completion of the previous one. Optimizations can be implemented, but only in relation to the selected computation platform.

- *Line 11*: **if** (hscoreMAX>=Threshold)

S ← **mask**(S, lcrType, from, to)

This computation step is mainly focused on executing the *mask* function over the input sequence *S*. The *mask* function requires full access over sequence *S*, as it will updating the sequence for the next iteration. As such, special care should be taken in case of creating multiple copies of *S* when parallelizing *Line 7*.

- *Line 13*: **while** (hscoreMAX>=Threshold)

The condition that keeps CAST iterating until met. Essentially it is a single comparison per iteration over a single data variable.

Extensive profiling of the CAST application yielded identical results with the above analysis. The heaviest processing is done by the *detectBias* calls of *Line 7*'s code while the rest of code segments are executed in constant time and cost only <2% of the overall processing for the benchmarks in *Appendix A: Proteomic Benchmarks*.

To summarize our profile and analysis of CAST application: parallelization attempts should be focusing on *Line 7*'s loop and the *detectBias* function. However, special consideration should be given in the critical code segment of *Line 11*. Failing to update all instances of *S* will result to unpredictable behaviour and wrong output.

Various implementations of CAST for multicore CPUs, GPUs and FPGAs have been designed and extensively evaluated in this thesis since CAST is a prime case study algorithm for the field of sequence alignment and processing. Different techniques and methods were used to achieve parallelization of the identified code segments for each computation platform. For example, we used shared memory to hold the input sequence S for the multi-threaded implementations running on multicore processors, while we used a specialized, custom-build streaming pipeline architecture for the FPGA-based implementation.

Extensive details and evaluation results can be found in latter chapters: the multi-threaded CAST software is discussed in *Chapter* 4, the GPGPU implementation using *Nvidia's CUDA* in *Chapter* 5, while the FPGA-based architecture is presented in *Chapter 6*.

2.2 Subthalamic Nucleus Deep Brain Stimulation Modelling

Extracellular recordings in the area of the subthalamic nucleus (STN) of Parkinson's disease patients undergoing deep brain stimulation comprise fast events, Action Potentials and slower events, known as Local Field Potentials (LFP) [35]. Previous studies utilized pre-defined LFP features to predict spiking from simultaneously recorded LFP, and have reported good prediction of spike bursts but only moderate accuracies for individual spikes [36][37]. The LFP is believed to represent the synchronized input into the observed area, as opposed to the spike data, which represents the output. It is shown before that there is an input-output relationship between these two components in the STN [35].

2.2.1 STN Modelling Using Volterra-Laguerre Networks

Researchers from University of Cyprus extended the above observations by using LFP-driven Volterra models and the Laguerre expansion technique to estimate nonlinear dynamic models which are able to predict the recorded spiking activity [37]. The team managed to utilize a data-driven approach, without relying on feature selection, to predict individual spike times. The relationship between LFPs and multi-unit spike trains in monkey early visual cortex during passive viewing of grating stimuli was analyzed using a variant of the general Volterra approach, the Laguerre-Volterra network (LVN) which has been successfully applied to modeling of dynamic physiological systems [38].

An LVN with 2 inputs and 1 output is illustrated in Figure 2.2. The network is preceded by a number of Laguerre Filter Banks (LFB). Each LFB consists of a number of filters that each input value is exposed in succession. Each filter deeper in the LFB is receiving as input the output of the previous ones plus some LFB variables. The filters' output are weighted and summed. The result is then fed into the neural network. The neural network can consist of any number of layers. Our example LVN of Figure 2.2 only has a single layer. Its weights can be tuned by a learning algorithm.

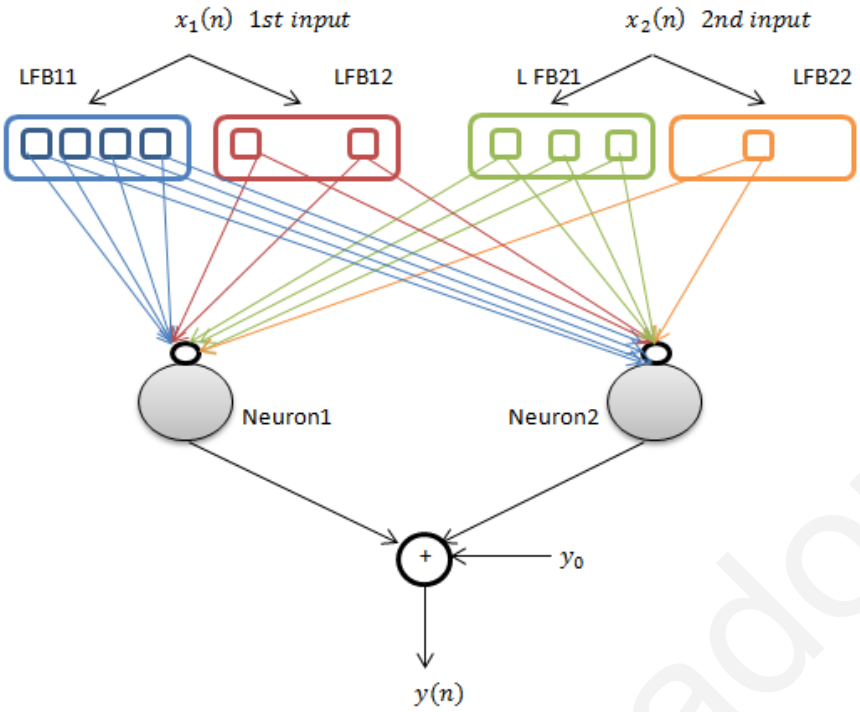


Figure 2.2: A two-input single output fully connected LVN with 2 neurons.
Each input is preprocessed by 2 Laguerre filter banks (LFB).
Each filter bank is characterized by a unique Laguerre parameter (a11, a12, a21, a22).
The two LFBs for the 1st input contain L11=4 filters and L12=2 filters and
the two LFBs for the 2nd input L21=3 and L22=1 filters.

The developed model for STN modelling is based on piecewise stationary analysis of the data over time, i.e. partition the non-stationary data into small time frames, where we assume that in each frame the data is stationary. Sampling frequency is the factor that determines the number of frames - input samples - to be processed. The model has the following specifications:

- **Inputs**: 2
- **Maximum capacity**: 1500 samples
- **Maximum Number of LFBs for each input**: 2
- **Maximum Number of filters (L) for each LFB**: 10
- **Maximum Number of neurons**: 4
- **Maximum nonlinearity**: Q=3

The code for the developed model is given in Figure 2.3. Parameter *i* denotes the input, f_i is the total number of LFBs for the input *i*. *Q* expresses the nonlinearity of the model, i.e. the order of the polynomial functions of the neurons present in the model. For instance, *Q*=2

denotes a second order polynomial function for the neurons, in the form of $c_1 u(n) + c_2\, u^2\,(n)$.

n is denoting the input sample processed; *N* is the total number of input samples and is identical for all input *i*. LFBs are represented by *j* and L_{ij} is the total filters for the LFB *j* for each input *i*; *m* identifies the neurons and $\boldsymbol{w_{i,j,m}}=[w_{i,j,m}(1) \ldots w_{i,j,m}(L_{ij})]$ are the weights of the filters in LFB *j* of input *i* which are used to produce the input of neuron *m*. $\boldsymbol{c_m}=[c_{m,1} \ldots c_{m,Q}]$ are the coefficients of the polynomial function of neuron m, y_o is constant for every *n* and $T = {}^{1}/_{f_s}$ is the sampling period and frequency.

As we can easily infer, the LVN structure is modified as we change the values of f_i, L_{ij}, Q, and the number of neurons. The model needs to be optimized for the STN problem under consideration. The optimization process can be described as the process we need to follow to identify the values of f_i, L_{ij}, Q, and the number of neurons that minimize the equations below:

$$\boldsymbol{e} = \boldsymbol{y}(\boldsymbol{ignore}:\boldsymbol{end}) - \hat{\boldsymbol{y}}(\boldsymbol{ignore}:\boldsymbol{end})$$

$$J = \sum_{t=1}^{N} e^2(t).$$

$$BIC(d) = \frac{N}{2}\log\left(\frac{J}{N}\right) + \frac{d}{2}\log N$$

$$AIC(d) = \frac{N}{2}\log\left(\frac{J}{N}\right) + d$$

$\boldsymbol{N}$ corresponds to the total samples on the inputs; $\boldsymbol{y}$ the measured STN value when our inputs are observed; $\hat{\boldsymbol{y}}$ is the predicted value by the STN model using LVN; ***ignore*** is a user defined parameter that helps ignoring some initial input points; $\boldsymbol{d}$ is the total number of the model parameters:

$$d = \left(\sum_{i=1,j=1}^{i=2,j=f_i} L_{ij} + Q\right) \times neurons$$

For n=1… N

For i=1:2

For j=1:fi

If (n==1)

$$v_{i,j,1}(n) = T\sqrt{1 - a_{i,j}}\; x_i(1)$$

For k=2… L_{ij}

$$v_{i,j,k}(n) = \sqrt{a_{i,j}}\, v_{i,j,k-1}(1)$$

End

Else if (n>1)

$$v_{i,j,1}(n) = \sqrt{a_{i,j}}\, v_{i,j,1}(n-1) + T\sqrt{1 - a_{i,j}}\, x_i(n)$$

For k=2… L_{ij}

$$v_{i,j,k}(n) = \sqrt{a}\left(v_{i,j,k}(n-1) + v_{i,j,k-1}(n)\right) - v_{i,j,k-1}(n-1)$$

End

End

For m=1….neurons

$$p_{i,j,m}(n) = \boldsymbol{w_{i,j,m}} \begin{bmatrix} v_{i,j,1}(n) \\ \vdots \\ v_{i,j,L_{ij}}(n) \end{bmatrix}$$

End

End

End

For m=1….neurons

$$u_m(n) = \sum_{i=1}^{i=2} \sum_{j=1}^{f_i} p_{i,j,m}(n)$$

$$z_m(n) = \boldsymbol{c_m} \begin{bmatrix} u_m(n) \\ \vdots \\ (u_m(n))^Q \end{bmatrix}$$

End

$$y(n) = y_0 + \sum_{m=1}^{neurons} z_m(n)$$

End

Figure 2.3: Pseudocode for the Volterra-Laguerre Model for STN.

2.2.2 Particle Swarm Optimization

The search for the values of parameters f_i, L_{ij}, Q, and number of neurons that yield optimal performance for the STN model can be done either by brute force exploration or by using optimization techniques such as particle swarm optimization (PSO) [39][40] or any type of evolutionary algorithm [41]. PSO has shown to accurately generate the values required [37].

Function PSO (S, b_{lo}, b_{up}) **returns** g

inputs: S, the swarm size

b_{lo}, b_{up}, the search space boundaries

outputs: g, global optimal solution

loop for *i* **from** 1 **to** S **do**

Initialize position, $x_i \leftarrow$ Uniform Random Selection (b_{lo}, b_{up})

Initialize local best, $p_i \leftarrow x_i$

Initialize global best,

if (i=1)

then $g \leftarrow p_i$

if ($f(p_i) < f(g)$)

then $g \leftarrow p_i$

Initialize velocity, $v_i \leftarrow$ Uniform Random Selection ($-|b_{up}-b_{lo}|$, $|b_{up}-b_{lo}|$)

end

repeat

loop for *i* **from** 1 **to** S **do**

r_p, $r_g \leftarrow$ Uniform Random Selection (0,1)

loop for d **from** 1 **to** n

$\mathbf{v}_{i,d} \leftarrow \omega\, \mathbf{v}_{i,d} + \varphi_p\, r_p\, (\mathbf{p}_{i,d}-\mathbf{x}_{i,d}) + \varphi_g\, r_g\, (\mathbf{g}_d-\mathbf{x}_{i,d})$

end

$\mathbf{x}_i \leftarrow \mathbf{x}_i + \mathbf{v}_i$

If ($f(\mathbf{x}_i) < f(\mathbf{p}_i)$)

then $\mathbf{p}_i \leftarrow \mathbf{x}_i$

If ($f(\mathbf{p}_i) < f(\mathbf{g})$)

then $g \leftarrow p_i$

end

until <*criteria*> **are met**

return *g*

Figure 2.4: A general PSO algorithm layout.

A general PSO algorithm layout is shown in Figure 2.4: A general PSO algorithm layout.Figure 2.4. PSO is an optimization method that tries to optimize the solution of a given problem using a population of potential solutions – particles – that move around in the search space of the problem. Each particle is moving by simply following the equations of its velocity and last position. The optimization process occurs as each particle's movement is affected by its own local best known position and the global best known position reached by the swarm – the population of all particles [39].

PSO begins by randomly initializing the position of all particles of the swarm. However, these random initial positions must be within the boundaries of the search space $\mathbf{b_{lo}}, \mathbf{b_{up}}$. Each random initial value is used to initialize the local known position of each particle and the global swarm's position. The velocity of each particle in the swarm is randomly initialized as well.

The optimization process is then started and iterates until the termination criteria are met. These criteria differ from application to application and can be the number of iterations, an error threshold etc.

Each particle's speed is stochastically updated using the last known local position, the global best known position **g**, the user-defined inertia constants $\boldsymbol{\omega}$, $\boldsymbol{\varphi_p}$, $\boldsymbol{\varphi_r}$, and the uniform random variables $\boldsymbol{r_p}$, $\boldsymbol{r_g}$. The general algorithm in Figure 2.4 is updating each particle's velocity one dimension $\boldsymbol{d}$ at a time since each problem under consideration can be in any *n*-dimensional space. The position of each particle is then updated, followed by the updates of the local and global best-known positions updates. The global best-known solution $\boldsymbol{g}$ holds the best solution (global minimum within the search space) found so far by the swarm, by the end of each swarm's iteration.

2.2.3 STN LVN Model Analysis: How we Can Achieve Parallelization

Let's assume that the selected computation platform can support building a fully connected LVN taking into account the maximum values of the parameters f_i, L_{ij}, Q, and number of neurons. If a fully connected LVN is possible, then we can emulate a smaller LVN by simply disconnecting connections (set the respective weights at zero).

As we can see from the STN model pseudocode of Figure 2.3, the LVN model is calculated by iterating through the network using loops. Parameters are defined by the STN problem formulation and their values are bounded in: $L_{ij} \in [1,10], Q \in [1,3], n \in [1,4]$ as integers, while α_{ij} are real numbers and bounded in $\alpha_{ij} \in [0.1,0.99]$. The optimization of the model requires that we identify the optimal values for $\boldsymbol{w_{i,j,m}}$, $\boldsymbol{c_m}$, α_{ij} and y_0. The LVN structure is changing as one or more values of $\boldsymbol{w_{i,j,m}}$, $\boldsymbol{c_m}$, α_{ij}, y_0 are set to be zero.

A PSO module provides the values of $\boldsymbol{w_{i,j,m}}$, $\boldsymbol{c_m}$, α_{ij} and y_0 that we need to check if they produce the optimal results by minimizing the equations discussed in the previous section. As such, the PSO module is an important part of the STN model optimization as well.

We will first analyse the pseudocode of Figure 2.3 in order to identify potential parallel regions of the code that we will later use for parallelizing the STN model.

- *Line 1*: **for** n = 1…N
  for i=1…2
  for j=1…fi

These three nested *for* loops construct all of LFBs of the LVN. The first loop is allowing the model to iterate though all of the input samples *n*. The second nested loop is responsible to calculate the model's response for all inputs *i*. Lastly, the inner most loop is calculating the responses of all LFBs *j* on the their respective input *i*.

Simply by reading the rest of the LVN's code, we can see that there are no data dependencies between the filters and LFBs for different inputs. As such, we can move the *i*-loop before the *n*-loop without changing the output of the LVN code and then use a technique known as *loop unrolling* for parallelizing the *i*-loop. That means that we can execute each instance of the *i*-loop on different processing units in parallel. In this case, i=1 instance will run on processing unit A, while i=2 instance will run on processing unit B of the selected computation engine.

Each LFB of each input can be calculated independently as well since there are no data dependencies between the LFBs of the same input and a_{ij} is remains the same for each LFB. As such, we can move the *j*-loop just after the *i*-loop and incorporate *loop unrolling* on it too. This means that if there are available resources on the selected computation platform, LFBs can be calculated in parallel as well.

- *Line 4*: **if (**n == 1)
- *Line 9*: **else if (**n > 1)

Studying the above code segments we can see that we cannot use the same techniques for the *n*-loop. The conditions in line 4 and line 9 let us understand that the first sample of each input requires different calculations than the rest. Assuming that *N=1500* samples per input, we can see that the first input is an exception <1% of the total calculations of each input. Each computation platform allows for methods to "hide" these kinds of exceptions and allow for parallelizing the rest. For example, each thread on a multi-threaded STN modelling software can check its assigned n sample and execute an exception routine for n=1.

- *Line 7*: **for** k=2…Lij

$$v_{i,j,k}(n) = \sqrt{a_{ij}} v_{i,j,k-1}(1)$$

- *Line 10*: $v_{i,j,i}(n) = \sqrt{a_{ij}} v_{i,j,1}(n-1) + \cdots$

The main issue that restricts our parallelization efforts is the data dependencies between the filters of the same LFBs and the data dependencies between adjacent input samples as shown from lines 7-8 and l0. These dependencies can be an issue to deal with in case of selecting GPU as the computation platform, as they limit the level of achievable data parallelism. In case of a potential FPGA-based approach, these dependencies can be solved by using a pipelined architecture where sample data are streamed through adjacent processing units. Each unit will receive the required previous computation data through the pipeline.

- *Line 15*: **for** m=1…neurons

$$p_{i,j,m}(n) = \cdots$$

Calculating the inputs of the neural network requires the outputs of all the LFBs to be available. Techniques such as *thread synchronization* need to be used in case we are aiming for a multi-threaded approach for a multicore system, in order to assure that all data needed is available for computing the loop of line 15. Implementing FPGA-based hardware architecture of the STN model allows for using *accumulators* as means to gradually build $p_{i,j,m}$ on-the-fly while the pipelines calculate the results for each LFB.

- *Line 20*: for m=1…neurons

$$u_m(n) = \cdots$$

$$z_m(n) = \cdots$$

- *Line 24*: $y(n) = \cdots$

The loop of Line 20 and the code of Line 24 deal with the layers of the neural network. Assuming that the required values of $p_{i,j,m}$ are all available, we can calculate u_m and z_m each on a single processing step. If this is not the case, special synchronization case should be taken in case of a software implementation or implement some sort of specialized accumulators in an FPGA-based approach. Line 24 can calculate the output of the neural network in a single processing step, assuming that the inner layers of the network are finished their calculations on the same time step.

- $J = \sum_{t=1}^{N} e^2(t)$.
- $BIC(d) = \frac{N}{2}\log\left(\frac{J}{N}\right) + \frac{d}{2}\log N$
- $AIC(d) = \frac{N}{2}\log\left(\frac{J}{N}\right) + d$

The optimization equations are single step computations, except of *J* where its results are a summation of the square errors for all input samples. However, *J* is calculated only once and as such, no effort is necessary to be allocated for reducing its computation time (See *Amdahl's law* in Chapter 3.1.2).

We will now analyse the PSO optimization algorithm and try to achieve higher performance though parallelization as well. We will use the general algorithm of Figure 2.4 for our analysis.

- *Line 1*: **for** i=1…S
  Initialize $x_i \sim U(b_{lo}, b_{up})$
- *Line 7*: $v_i \sim U(-|b_{up}-b_{lo}|, |b_{up}-b_{lo}|)$

The first loop of the algorithm only runs once and its purpose is to initialize the positions and velocities of every particle *i* of the swarm *S*. The only code that can cause an issue is the random generator U. Each call of the function U must be able to generate an *n*-dimensional vector of position and later velocity for a respective particle. Since each dimension *d* of each uniformly generated vector is independent from the rest, we can simply clone the random generator U for one dimension and parallelize the vector generation on the selected processing platform. Another option is to use already available highly-parallel random generator libraries such as the cuRAND library for CUDA-enabled GPU cards [42].

- *Line 9*: **while** (<criteria> not met)

The loop of Line 9 is the critical code segment of the PSO algorithm. This loop will continue to iterate until the termination criteria are met. The loop continuously updates the velocities, positions and statistics for all particles of the swarm. Our efforts for parallelization should be focused here.

- *Line 10*: **for** i=1…S

The loop of Line 10 is responsible for updating all particles of the swarm. As we can see from the code of Figure 2.4, each particle is independent of each other. However, all particles do share the global best-known position *g*.

We can modify the algorithm to update g as soon as the loop of Line 10 finishes by moving the conditional update of Line 18 in a new loop that executes after the Line 10 loop finishes. This modification allows for calculating the updated features (velocity, position, local best-known position) of each particle *in parallel* without having to deal with the critical code section of Line 18 that updates the shared variable *g*. The level of achievable parallelism is directly related to the selected computation platform.

A multi-threaded software approach for a *x*-core processor can create *x* threads (i.e. on a quad-core CPU, will create 4 threads); each thread will be responsible for updating particles selected from a globally shared FIFO queue. A GPGPU-based approach however, can utilize *hundreds* of processing elements – equal to the total population of the swarm; each element will handle just one particle.

A note should be made about the proposed modification and their effect on the termination criteria. In the case that the termination criteria are focused on a metric regarding the global best-known position (for example, $|g_n|-|g_{n-1}|<0{,}01$) and not on general iterations etc., there is a strong possibility that the modified algorithm will need to iterate once more before reaching the terminal condition since the *g* value is not updated on-the-fly but after *all* particles are updated.

- *Line 12*: for d = 1…n

$$v_{i,d} \leftarrow \omega\, v_{i,d} + \varphi_p\, r_p\, (p_{i,d}-x_{i,d}) + \varphi_g\, r_g\, (g_d-x_{i,d})$$

The loop of Line 12 is calculating the updated speed for each dimension of the particle under update. This code segment can be parallelized quite easily as there are no data dependencies

among the dimensions of each particle. The selected computation platform affects the level of parallelism in this case as well. For example, creating threads for each dimension in multi-threaded software approaches is not a wise decision, as the OS is already busy scheduling the threads for the particle updates discussed above. Creating more threads when there are no available resources hinders performance, as the OS has to switch between threads thus increasing overheads. Using an FPGA-based approach, on the other hand, allows for *massively* parallelizing this code segment as we can just clone the respective processing unit as long as we have available hardware fabric on the selected device.

The last issue we need to address while we analyze the LVN-based STN model is how we can transfer data between the PSO optimizer module and the LVN module. When using software for implementing multi-threaded versions, we can just combine the code together. The PSO will call the LVN functions and vise-versa. The GPGPU approach works in a similar way, as a PSO kernel will pass data to a LVN kernel and vise-versa. Developer's only consideration should be to adequately synchronize the two software components both for a multi-threaded and a GPU-based software approach. If we select an FPGA-based hardware architecture for the application, the integration of the two modules is coming down to passing the necessary data and control signals between the two.

Integrating different platforms for each component on a hybrid system is a totally different issue. Let's say we selected PSO to be implemented on a GPU and the LVN model to be implemented either on an FPGA or on an multi-core CPU. The data transfers between the two modules need to be done through PCI-Express Bridge and this fact adds significant overheads to this potential design. A streaming system that continuously transfers data between the units can hide some percentage of the transfer overheads. However, CUDA framework requires all kernel calculations on the GPU to be finished before returning data through the PCI-Express; as such a streaming system layout is not easily achievable.

On the other hand, transferring data from an FPGA to a multi-core processor through PCI-Express can be done by simply loading the FPGA as PCI-Express device on the OS. The multi-threaded software running on the multi-core system can now "see" and access data from the FPGA memory through the OS memory map.

2.3 **De Novo DNA Sequence Assembly**

Genomics, i.e. the science of analysing DNA and extract useful information for advancing medicine and our understanding of living organisms is usually bound by one of its most important problems: obtain a complete genome sequence using millions of fragments – called *reads* – extracted from laboratories around the world [43].

Sanger sequencing technology was the prominent method of extracting genome information from living organisms until 2005, when next-generation sequencing techniques where commercially introduced. Sanger can produce large reads (500-1000 base pairs) of genomes under examination; however, its high cost could not allow for large number of genome projects to be completed.

Next-generation sequencing (NGS) completely changed the picture of genomics. These new sequencing techniques drastically reduce the cost per sequenced nucleotide and can produce reads at astonishing speeds – orders of magnitude faster than Sanger. Nowadays, gigabytes of genome data are generated every day and the issue now lies on how we can use this extreme amount of data to extract actual genome information. Next-generation sequencing is producing a burst of small reads (30-300 base pairs) and using these reads to build large *newly-discovered* genomes (lengths larger than 100 million base pairs) is increasingly hard.

Taking the above facts into account, new algorithms and methods were needed to be developed for allowing reliable *de novo* sequence assembly for next-generation sequencing. State-of-the-art de novo sequencing algorithms are computationally intensive algorithms that run over extremely large amounts of data. Parallel computation techniques on the newly available processing engines will definitely help increasing performance of these tools and further reduce the sequencing cost towards the milestone of $1000 per complete genome [2].

We will now give a brief introduction to the available de novo sequencing tools and then discuss their potential parallelization. Some of the initial sequencing algorithms were just adaptions of older tools created to work over Sanger-retrieved reads. Those algorithms often evolved to become strong tools for next-generation de novo sequencing as well. Some other methods were developed exclusively for next-generation sequencing. In order to avoid lengthy descriptions of these algorithms, we present the following overview:

SaSSY [147]
Class: **Overlap-Layout-Consensus** Year: **1995**
Features:
- Paired read info during graph formation
- Novel heuristics
- Avoid all-versus-all comparison during overlap calculation

Issues:
- Memory Intensive

EULER [130]
Class: **De Bruijin** Year: **2001**
Features:
- Uses **De Bruijin** graphs
- Assembler based on **Euler path** solver

GPU_Euler [131]
Class: **De Bruijin** Year:**2011**
Features:
- CUDA-enabled implementation
- Uses Awerbuch parallel Euler path solver [132]

Advantages:
- 4X speedups compared to the original

SSAKE [137]
Class: **Greedy** Year: **2006**
Features:
- *First* to be released for NGS
- Assembles reads of uniform length
- Uses De Bruijin graph
- K-mer search / 3' extension
- Uses **prefix tree / hash tables**

Issues:
- No pair reads / Fails to identify repeats

Parallel Velvet H [142]
Class: **De Bruijin** Year: **2011**
Features:
- Cluster version of VELVET
- Uses hp-MPI
- Improves k-mer reads
- Partitions k-mer hash table in multiple ranks

Advantages:
- Assembles a human genome in 2 hours

VELVET [148]
Class: **De Bruijin** Year: **2008**
Features:
- *Most popular* set of NGS assembly tools
- Uses **De Bruijin** graphs
- K-mers organized in **hash table**
- Uses pair reads

Issues:
- Needs massive amount of shared memory for large genomes

VCAKE [140]
Class: **Greedy** Year: **2007**
Features:
- **Prefix tree** approach
- K-mer extension / 3' read extension
- Voting mechanism
- Handles erroneous reads

Issues:
- No paired reads

EDENA [139]
Class: **Overlap-Layout-Consensus** Year: **2008**
Features:
- *First* NGS assembler to use traditional **overlap-layout-consensus** approach
- No error correction before graph building phase
- Three error correction steps before assembly

Issues:
- Messy sequence graph (no error correction before building it)

ALLPATHS [136]
Class: **De Bruijin** Year: **2008**
Features:
- Uses **De Bruijin** graphs
- Hybrid approach

Advantages:
- **Most accurate** method available

Issues:
- Requires massive memory
- Time consuming method

SHARGCS [146]
Class: **Greedy** Year: **2007**
Features:
- **Prefix tree** approach
- builds three naïve approaches with three different filters
- tries to merge them for higher quality results

Issues:
- No paired reads

QSRA [145]
Class: **Greedy** Year: **2009**
Features:
- Extension of VCAKE
- **Prefix tree** approach

Advantages:
- Quality scores

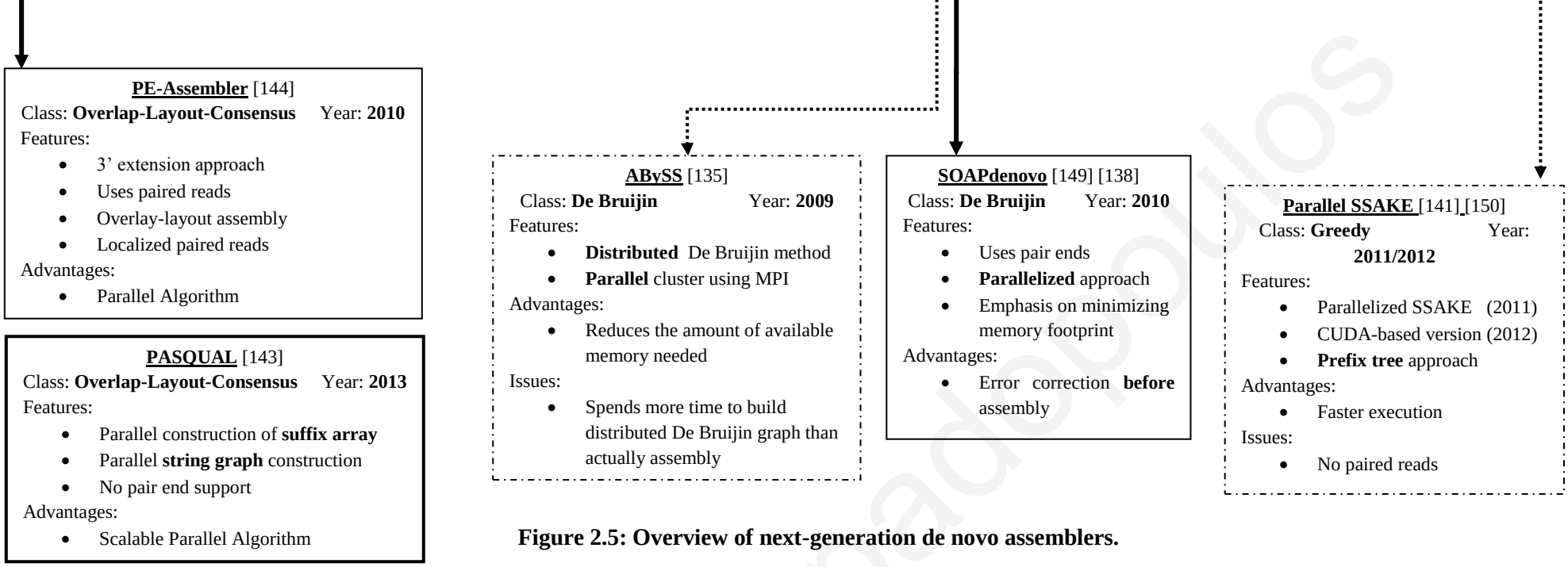


Figure 2.5: Overview of next-generation de novo assemblers.

Algorithm	*Steptococcus suis* s.suis genome		*Helicobacter acinonychis* *h.acinonychis* genome		*Steptococcus aureus* s.aureus genome		*Escherichia coli* *e.coli* genome		Human genome	
	Total Time	Total Memory	Total Time	Total Memory	Total Time	Total Memory	Total Time	Peak Memory	Total Time	Peak Memory
ABySS					13m	2.6 GB	43m	2.9 GB	87h	16.0 GB
ALLPATHS2							3h 47m	29.7 GB		
EDENA	6m	0.5 GB	20m	1.5 GB	10m	0.9 GB	188m	2.8 GB		
PASQUAL							<5m	N/A		
PE-Assembler					17m	1.9 GB	21m	2.3 GB		
QSRA	55m	1.9 GB								
SHARGCS			8h 0m	50.0 GB	17h 0m	20.0 GB				
SOAPdenovo							<5m	5.9 GB	42h	140.0 GB
SSAKE	1h 39m	2.1 GB	16h 0m	2.5 GB	1h 30m	1.2 GB				
VCAKE	2h 02m	1.7 GB								
VELVET	3m	0.7 GB	11m	1.2 GB	5m	0.4 GB	10m	2.9 GB	Unable to run	Unable to run

Figure 2.5 shows an overview of the NGS assemblers with their main features, their advantages and their open issues. Figure 2.5 is accompanied with the available execution times and memory requirements for some of the most popular DNA benchmark genomes. The data in the aforementioned table are gathered through the literature and the references mentioned in the figure. The data in the table is sparse, since we show the time required for running the assemblers only in the cases reported to use the same computational platforms for all the runs. We believe that this data can prove useful insights on the merits of each mentioned assembler. An 2011's survey paper which attempted to gather information on the capabilities of the available de novo assemblers by comparing their performance on a number of genomic benchmark sequences [44], recommends different assemblers for difference genome assembly problems: For very small reads assembly of a microorganism's genome on a machine with less than 16GB RAM, the authors suggest to use SSAKE,QSRA or EDENA; For reads less than 36bp of a large genome on a system with more than 16GB RAM, they suggest SOAPdenovo, while for reads 75bp ALLPATHS is superior on the same system. As we can see, there is no perfect assembler for all de novo genome assembly problems.

The assemblers described in Figure 2.5 can be categorized in three distinct classes: *Greedy*, *Overlap–Layout–Consensus (OLC)* and *de Bruijn graph*. The choice of which type of assembler to use, depends on the characteristics of the data being assembled. For example, de Bruijn graph assemblers have been successful in assembling highly accurate short reads (<~100 bp), whereas OLC-based methods are mostly used for longer, more inaccurate data (>200 bp) [45]. Unfortunately, both OLC and de Bruijin graph de novo sequence assembly algorithms are proven to be *NP-hard*[2] [46]. This factor increasingly affects execution times as the genomic data analysis needs continue to grow in exascale (10^{18}) – i.e. a quintillion floating-point operations per second – a scale where the HPC community is not yet comfortable to address [47].

However, as the NGS technology improves, new assembling algorithms are implemented and the methods used are continuously evolving to match the features and the scale of the newly

[2] NP-hard (Non-deterministic Polynomial-time hard) is a class of computation problems where there exists a polynomial-time reduction that can map an NP problem to the aforementioned NP-hard one [133]. An informal definition could be that NP-hard problems are at least as difficult to solve as an NP problem, i.e. there is no polynomial-time solution for an NP-hard problem identified *yet*.

generated data. For example, de Bruijn graph assemblers are evolved and can now process longer reads by using pre-processing stages to correct errors, while OLC assemblers have been modified and now are able to cope with shorter length reads. Moreover, the assembly parameters chosen by the end-user can greatly impact the performance of the assembler.

A thorough literature review revealed that OLC assemblers are usually *computationally intensive*, while most of the de Bruijin graph assemblers are *memory intensive*. Greedy approaches lay in between. These traits can help us identify potential parallelism to exploit in order to further improve performance of NGS de novo assemblers. We describe each class of de novo sequence assembly algorithm's characteristics in the section below.

2.3.1 De Novo Sequence Assembly Problem Characteristics

NGS machines provide multiple copies of the genome under examination, each randomly fragmented. De novo sequencing means that we need to construct the initial genome sequence from scratch – i.e. having no reference for how the complete genome looks like. Assuming that we have enough reads to cover every position of the genome multiple times, we can say that if there is similarity between the beginning of one read and the end of another, these two reads most probably originate from overlapping fragments in the original genome.

The de novo sequence assemblers try to take advantage of the aforementioned overlapping possibility. Firstly, they build a data structure – usually an overlap graph or a prefix tree – which holds the information on the available reads. An overlap graph has the reads in its vertices and the edges hold the information of discovered overlaps between them.

The assembly problem is then reduced to finding the optimal traverse of the graph that connects most of the vertices together. The assembly problem is not as simple as it looks since we have no *a priori* knowledge of the genome. The problem can be mapped on the *shortest common superstring (SCS)* problem, *Hamiltonian Path* problem or even the *Travelling Salesman* problem. These problems are *NP-hard* and as such the optimal solution is difficult to be found.

2.3.2 Greedy De Novo Assembly Algorithms

Greedy algorithms can be employed to solve a non-optimal variant of the assembly problem. Given an overlap graph of the reads, the algorithm "greedily" chooses the longest remaining overlap in every step and merges its two ends. Greedy algorithms are not guaranteed to choose overlaps yielding the optimal SCS. However, the yielding superstring will not be more than 2.5 times longer than the true SCS [48]. Pairwise alignment - another greedy option - can be done between all available reads of the genome and the reads with the largest overlap can be merged iteratively to construct the new genome.

Special characteristics of the genome such as repetitive regions or circularity, and errors stemming from the technology used or possible mutations are greatly affecting the quality of the results of greedy assemblers. Techniques such as maximum likelihood of the reads to force the solution to be consistent with uniform coverage or even ignore regions of the genome that are deemed irresolvable is incorporated in greedy assemblers in an attempt to increase the quality of the results while having lower execution times than the initial assembly problem formulations. This is the case of OLC and de Bruijin assemblers which both handle irresolvable repeats by breaking the assembly into fragments - called *contigs* - rather than attempting to assemble the whole genome.

2.3.3 Overlap-Layout-Consensus Assembly Algorithms

Overlap-Layout-Consensus (OLC) algorithms have three distinct steps. Firstly, the available reads are scanned and compared to identify *overlaps* based on all pairwise comparisons. This information is used to construct an overlap graph. An overlap graph is a data structure where the reads are represented as nodes and the edges represent the overlaps between them.

The OLC method continues with its second step: *layout*, where it looks for segments of the genome in the constructed overlap graph. This is done by trying to find the *Hamiltonian Path*,- i.e. how all nodes of the graph can be visited exactly once [49]. The discovered path is used to construct the *contig* for the reads under analysis.

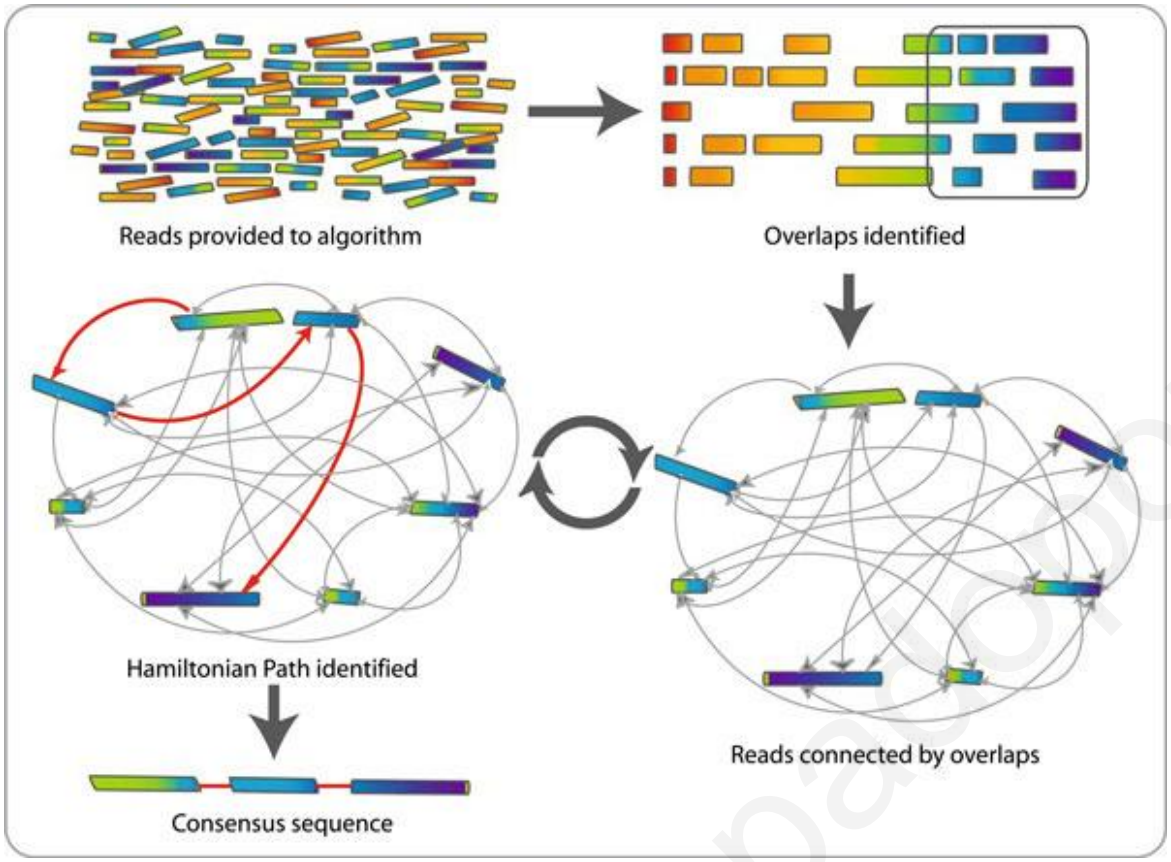


Figure 2.6: Overlap-Layout-Consensus Assembly Algorithm overview [134].

In cases where multiple edges between nodes do exist in the overlap graph, multiple sequence alignment can help reaching the *consensus* sequence (the *contig* corresponding to the input reads) by selecting the most abundant nucleotide for each position, however a sufficient number of reads is required to ensure a statistically significant consensus. During the *consensus* process, reading errors are corrected as well. The workflow for a general OLC de novo assembler is shown in Figure 2.6.

OLC-based de novo assemblers seem to work well for a number of different genome assembly cases [44]. Their main drawback is the use of the *Hamiltonian Path* in order to identify newly discovered contigs in the to-be-assembled genome sequence. The *Hamiltonian Path* is as already mentioned an *NP-hard* problem and as such there is no polynomial time solver for this problem. Moreover, the computations needed to construct the overlap graph and identify its Hamiltonian paths are intensive and hinder the overall performance of the OLC assembly algorithms.

2.3.4 De Bruijin Assembly Algorithms

The de Bruijin-based algorithms for de novo sequence assembly were introduced in order to improve the time needed to identify new contigs by transforming the problem of finding a *Hamiltonian Path* to finding an *Eulerian Path*. The *Eulerian Path* problem is similar to the *Hamiltonian Path* problem as both are used to traverse graphs. The *Eulerian Path* problem tries to traverse all edges of a graph by using each edge just once [50]. Traversing all edges once instead of traversing nodes, is much easier and this algorithmic contrast motivated the design of Euler-based genome assemblers [51].

De Bruijin assembly is an counterintuitive method, the genome sequence reads are partitioned in a large number of much smaller fragments and the genome is recreated from these fragments [52]. This class of De Bruijin-based assemblers are - in general - following a two-step process. First, the reads are partitioned in smaller *k*-mers - substrings of a user-defined fixed length *k* - and use them to construct the *De Bruijin graph* of the genome reads. *De Bruijin graphs* is a directed data structure that holds *k-1*-length suffixes and prefixes as nodes. The k-mers are represented by the directed edges of the graph and each k-mer is actually an *k-2* overlap of the substrings represented by the connected nodes.

The assembly process is completed by traversing the De Bruijin graph looking to identify its *Eulerian Path*. The discovered path is considered to be a contig of the newly assembled

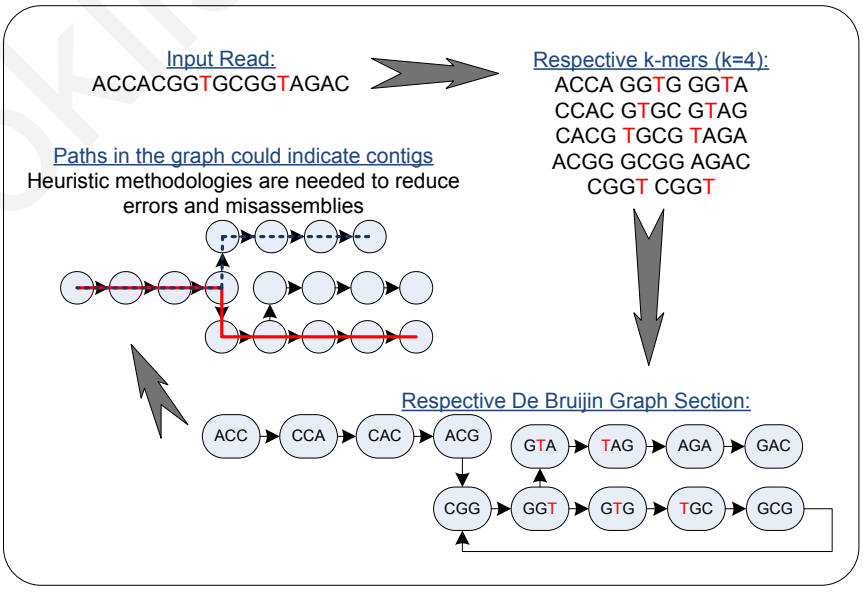


Figure 2.7: De Bruijin Assembly Algorithm overview for one read.

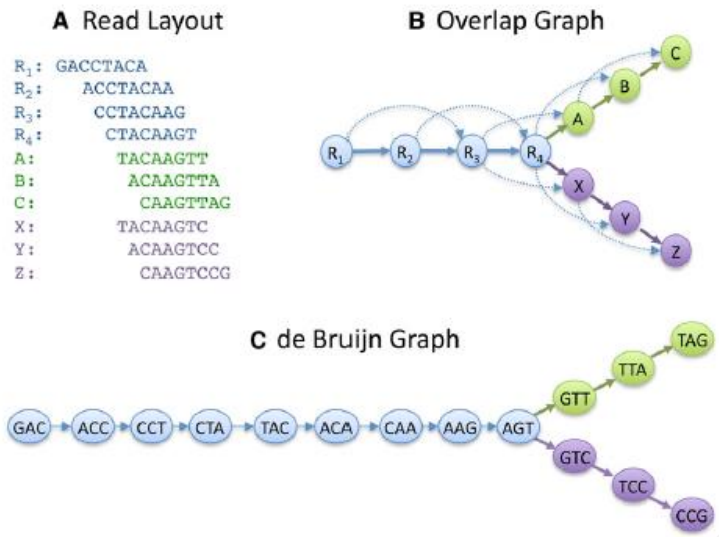


Figure 2.8: Comparison of Ovelap Graphs versus De Bruijin Graphs [54].
10 input reads (A) are used to create an overlap graph (B) with 10 nodes.
The respective De Bruijin Graph (*k*=4) requires much larger memory space to store its 15 nodes.
OLC-assembly is considered Computationally intensive while De Bruijin assembly is considered memory intensive.

genome. Despite the fact that the Eulerian Path of an Eulerian graph can be found in *linear time* directly related to the number of its edges [53], the assembly process still is an *NP-hard* problem [46]. Errors in sequencing can produce erroneous reads that can lead to a non-Eulerian graph. As such, to correctly identify contigs on this non-optimal data structure, we need to rely on approximation algorithms and heuristics in order to remove errors, redundancies and overall simplify the graph before traversing it.

Another drawback to the de Bruijin assemblers is the loss of information caused by partitioning a read into *k*-mers. De Bruijin-based algorithns create multiple *k-1*-length nodes for each read, and these nodes may not form a linear path once edges from other reads are added [54]. Furthermore, unlike the overlap graph, the De Bruijn graph is not read coherent [55], meaning there may be paths through the graph that form a sequence that is not supported by the underlying reads.

Last but not least, constructing the De Bruijin graph requires massive amount of available memory as the *k*-mers are in much more volume than their respective reads used in overlap graphs [54]. This fact is demonstrated with the a simple example of Figure 2.8. Another interesting difference between the two prominent de novo assembly classes is that De Bruijn graphs are able to represent efficiently genomes with repeats, while overlap graphs mask repeats that are longer than the read length [51]. Selecting the best assembler for each newly discovered issue is a trade-off between available computing resources and memory, required

accuracy and speed that needs to take into account the available sequencing technology that generates the reads.

2.3.5 Hardware Architectures for De Novo Sequence Assembly

Various research groups have been working into creating parallel versions of the most popular assemblers. These software-based attempts are depicted with dotted boxes in Figure 2.5. We choose to include them in our NGS de novo assemblers' overview diagram in order to show where the community's main efforts for higher performance are directed so far and how well they are doing.

The FPGA-based approaches however are not presented in Figure 2.5 as could not provide a fair comparison due to the specialized nature of these hardware architectures. The FPGA hardware architectures for NGS de novo assembly first presented in [56] and then improved in [57] showing speedups of 11x over Velvet software implementations. The works in this non-inclusive list of architectures as well as the rest available in the literature are *not* addressing the de novo sequence assembly problem *comprehensively* as they focus on parts of the assembly processes or just on performance estimations [58]. NGS assembly despite being a "perfect" problem to solve on FPGAs as it is both computational and memory intensive, its accelerating solutions is still in infancy due to bandwidth, memory and algorithm limitations.

2.3.6 De Novo Genome Sequence Assembly Analysis: How to Achieve Parallelization

The genome de novo assembly problem as mentioned in the previous sections is an open research topic and new algorithms of each class (greedy, OLC, or de Bruijin) are presented every few months. Hybrid approaches combining elements of more than one class have been introduced with promising performance results [59]. Attempts to achieve parallelization though multi-threading or MPI programming are depicted in Figure 2.5 as dotted boxes and their workflow description and performance results can be found in the mentioned references. It is worth to mention however, that the recorded attempts to parallelize one step of the

assembly procedure in software or in FPGA-based hardware can potentially cause great bottlenecks in the next step due to memory and/or resources limitations.

We have selected to focus our analysis in the pre-processing early stages of the assembly process, as we can positively benefit the overall execution time without falling under the limitations of *Amdahl's law*. Mainly, we will attempt to analyse a subset of algorithms and methods used in constructing the necessary assembly process graphs. More specifically, we will try to identify potential parallelism in the *suffix tree construction* - a crucial pre-processing step for creating the De Bruijin graph.

Suffix trees is an efficient and well researched data structure used for indexing the *k*-mers created from reads [60]. Various research groups have looked into parallel algorithms that can efficiently create suffix structures. *Suffix arrays* where introduced in early 90s [61] as an attempt to improve memory space requirements of *suffix trees.* Despite the fact that a suffix tree requires $\Theta(n)$ space, practically a suffix array uses less space. For example, a suffix tree to store *n* integers of *4 bytes* each requires *20n* bytes of memory in total, while the corresponding suffix array requires just *4n* bytes [62]. *Suffix arrays* are directly related with their respective *suffix trees*, as the array is the leafs of the tree given they are read in lexicographical order. Moreover, possibly the fastest way to construct a *suffix tree* is to construct its *suffix array* first and then transform it to *suffix tree*.

Suffix arrays have been the structure of choice for the most prominent De Bruijin graph assemblers and their construction algorithms are generally partitioned in the following classes: *Prefix doubling*, *Recursive*, and *Induced copying* [63].

Prefix doubling [64] is based on assigning order to suffixes using prefixes, while assigning order *preserving names*. The data is scanned for identifying unique prefixes: If not all prefixes are unique, the assessed prefix length is doubled and the algorithm iterates; If all prefixes are unique, the array is sorted and each prefix provides the rank of the associated suffix.

Recursive algorithms follow the approach proposed in [65] or similar to sort a subset of suffixes into a intermediate suffix array. The sorted subset is used recursively to create the suffix array of the remaining suffixes. Both intermediate arrays are then merged together into the final suffix array. A well-known linear time *recursive* algorithm for integer alphabets is the

Function PrefixDoubling(<S>, n) **returns** S_A

inputs: <S>, concatenated input sequence
n, inputs sequence length

outputs: S_A, the suffix array of <S>

Note: 1) T_i^l is the **prefix** of the **suffix** T_i of length l when $|T_i| < l$
2)**Order Preserving name:** N_i^l is an integer [0..n] such that $N_i^l \leq N_j^l$ *if and only if* $T_i^l \leq T_j^l$

repeat
Sort T_i^l using N_i^l to hold the index
Use the pairs (N_i^l , N_{i+l}^l) to represent $T_i^{2l} = T_i^l T_{i+l}^l$
until $l > n$ **or** [other *terminal conditions such as: unique index for all* $N_i^{l/2}$]
return N^1 , The final N^1 array is S_A, the suffix Array of the Input Sequence

Figure 2.9: Prefix doubling algorithm

DC3/skew algorithm [66].This algorithm has been used as the basis for both a parallel software version and a external memory suffix array construction algorithm [67] [68].

Inducing copying methods for constructing suffix arrays are similar to the recursive algorithms mentioned above as both use already sorted subsets in order to quickly sort the remaining suffixes. This class of algorithms differ from the previous one in the sense that they favor iteration over recursion to sort the selected suffix subset[63].

We will explore the general *prefix doubling* algorithm as prefix doubling is the *Ockham's razor*[3] option to suffix array constructing. As already mentioned above, this method uses the prefixes of the input string in order to construct the array. The pseudocode of the general prefix doubling method is given in Figure 2.9. Using the word *"banana$"* as an example, the algorithm will generate the following data:

The ***l*=1** sort is equivalent to sorting individual characters and generates T^1={b,a,n,a,n,a,$}, N^1={2,1,3,1,3,1,0} since $ is considered to be the smallest possible character in the alphabet Σ={$=0,a=1,b=2,n=3} that *"banana"* uses.

[3] A problem solving principle stating that the simpler solution to a problem is - in most cases - the better.

The $\boldsymbol{l}$**=2** sort will generate T^2={ba,an,na,an,na,a$,$}, N^2={3,2,4,2,4,1,0}. The prefixes to be sorted in this round are generated by doubling the length of the prefixes used on the previous round. For example, the prefix in position *i=0* is $T_0^2 = T_0^1 T_1^1 = \{b\}\{a\} = \{ba\}$.
The pair has the preserved name $(N_0^1 , N_1^1) = (2,1)$ which has the new index 3 in Σ={$=(0,0)=0,a$=(1,0)=1,an=(1,3)=2,ba=(2,1)=3,na=(3,1)=4}

The code continues to iterate for $\boldsymbol{l}$**=4** before reaching its terminal conditions ($\boldsymbol{l}$**=8 > n=7**). The generated arrays are: T^4={bana,anan,nana,ana$,na$,a$,$}, $\mathbf{N^4}$**={4,3,6,2,5,1,0}** where $\mathbf{N^4}$ is the suffix array of the word *"banana"*.

As we can see from our example execution of the prefix doubling algorithm of Figure 2.9, this application is *memory-intensive*. Computations needed are the prefix doubling and the integer index sorting, both over a massive string (*n* is in the orders of hundred thousand bases for de novo sequence assembly). The sorting can be done in linear time by a number of efficient methods such as radix sort, ternary radix sort and their variants. As such, attempts to extract parallelization should be focusing on the prefix doubling part of the algorithm.

The difficulty for massive strings is the extraction of the new sorting indexes needed for the array *N*. Calculating the index of DNA nucleotide bases (Σ={A,C,G,T,$}) needs to sort $|\Sigma|^l = 5^l$ indexes which doubles for every iteration. We can see that eventually we are reaching to a massive number of *preserved names* indexes to sort and then we need to pass and assign these indexes to *n* prefixes every iteration.

We decided to use FPGAs as a potential parallel computation engine because we have the flexibility to work with variable bit size to hold the variables for each iteration. The ability to use registers with more than 64bit length (the maximum available variable size for both CPUs and GPUs) allows for constructing the array $\boldsymbol{N}$ for multiple $\boldsymbol{l}$ concurrently using hardware-only available techniques.

De novo sequence assembly is indeed one of the most important open research topics in molecular biology. Our initial results showed that this biology problem can greatly benefit from hybrid CPU/GPU/FPGA computing platforms. Our proof-of-concept CPU-aided FPGA-based implementation for prefix doubling is discussed in *Chapter 6.2*.

2.4 **Closing Remarks**

In this chapter, we presented a number of different bioinformatics and biomedical applications which were selected as benchmark cases in order to investigate the potential performance gains by running on state-of-the-art accelerators and processing platforms such as multicore CPUs, GPUs and FPGAs.

Each of the selected applications was analyzed under the scope of extracting parallelism by modifying their execution flow for mapping them efficiently on each processing engine. The extracted information on potential parallelization is put to the test in latter chapters of this thesis. The following chapter provides an overview of parallel computing and how it is been used in this thesis.

3
Parallel Computing Fundamentals Related Work

This chapter is an overview of the fundamental computing theory we have based our high-performance parallel software and hardware solutions for accelerating the bioinformatics and biomedical applications presented in *Chapter* 2. We discuss the benefits and the limitations of parallel computing and we present the state-of-the-art computing platforms of today.

Moreover, the related work using these platforms for accelerating other bioinformatics and biomedical applications is considered in this chapter as well; the results in the literature are used to further elaborate the merits of using parallel computing on hybrid heterogeneous systems in biological sciences and medicine.

3.1 **An Introduction to Parallel Computing**

"Parallel computing" stands for the ability of computer systems to perform multiple operations *simultaneously*. The main principle behind parallel computing is the fundamental idea that large problems can be divided to smaller ones which can be then solved in parallel - i.e. executed concurrently on the available computing resources [69]. The quest for fully utilizing the available computing resources has started with the dawn of computing itself as achieving higher parallelism on one's application can increase performance, decrease wall-clock execution time, decrease performance-per-watt and save energy and resources.

Parallelism can be achieved in different forms, each form bounded by the technology limits of its era. Computer engineers have been trying to utilize as much parallelism as possible directly on hardware since the early 70s. This approach helped the parallelisation process to remain transparent to the software developers, thus allowing them to follow the same programming paradigms for decades while the performance kept improving.

However, physical limitations on silicon CMOS technology are preventing further relying on low-level hardware-based parallelism techniques. Scaling down transistor size is top priority for chipmakers; however, new challenges arise as the CMOS devices reach deep in the nanoscale regions.

Process variation - the natural random variation of transistor attributes in manufacturing - becomes extremely important at smaller scales (<65nm CMOS technology). These variations can be now a major percentage of the intended transistor features, thus altering the intended transistor's attributes. Thus, process variation can cause variation in output performance of the CMOS devices.

Reliability issues arise as well. Nanoscale CMOS devices have increasingly more chances to fail, as the downsizing of transistor features – length, width, oxide thickness – and the attempts to drive down operational voltage both negatively affect their predicted lifetime [70]. Naturally occurring phenomena that lead to failure are well-known since the 60s; however their effects become more and more prominent as CMOS technology scales down. Chipmakers and researchers are now studying these phenomena and how we can predict and improve the expected lifetime for any given CMOS device. Details on these phenomena and our work on how we can optimize CMOS lifetime can be found in [71].

Taking the above into consideration, application developers should now turn to the *multi-threading parallel programming paradigm* and new *emerging computing technologies* to achieve higher performance and lower power consumption for their application needs.

3.1.1 Types of Parallelism

Bit-level parallelism: This form of parallelism is based on doubling the processor word[4] size. Increased bit-level parallelism means faster execution of arithmetic operations for large numbers. For example, an 8-bit processor needs two cycles to execute a 16-bit addition while a 16-bit processor just one cycle. This level of parallelism seems to have come to an end with the introduction of 64-bit processors.

Instruction-level parallelism (ILP): This type of parallelism is trying to exploit the potential overlap between instructions in a computer program. Most forms of ILP are implemented and realized on each processor's hardware:

- Instruction Pipelining – Instructions partially overlap, thus making full use of idle resources.
- Superscalar Processors – Utilize multiple execution units in a single processor die, thus following instructions can be executed without waiting on complex preceding instructions to finish executing.
- Out-of-order execution – Instructions not violating any data dependencies are executed when a unit is available even when preceding instructions are still executed.
- Speculative execution / Branch prediction – The processor tries to avoid stalling while control instructions (branches - i.e. if or case commands) are still resolved by executing the most probable flow of the program.

[4] *Processor word* is the natural unit of data of each processor and actually is a fixed-sized set of bits. The hardware units of processors are built to handle processor words and their respective instruction sets are using processor words as units.

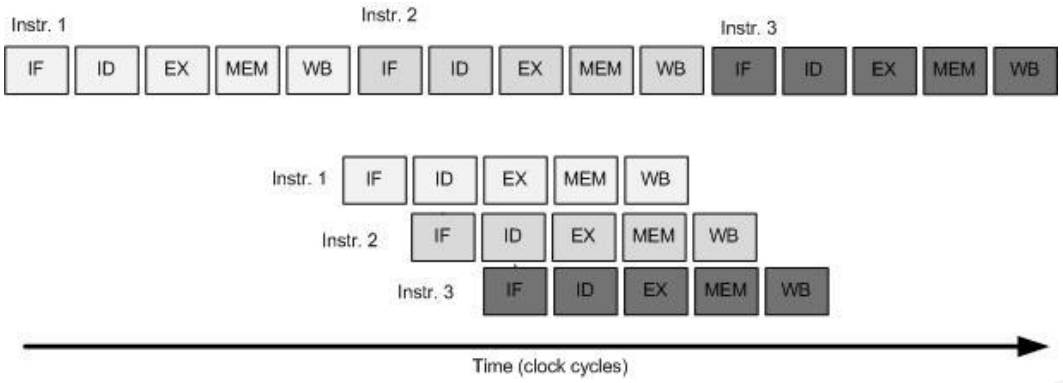


Figure 3.1: An example of ILP.
A simple 5-stage processor needs 15 clock cycles to execute 3 instructions.
However, a pipelined version of the same processor needs only 7 clock cycles for the same instructions.
Utilizing ILP decreased the execution time by: $\mathbf{1 - \frac{New\ Ex.Time}{Old\ Ex.Time} = \frac{(\#\ of\ Instructions - 1) + (\#\ of\ Stages)}{(\#\ of\ Instructions) * (\#\ of\ Stages)} = 54\%}$.

Most processors use a combination of the above ILP techniques. For example, Intel Pentium series used all of the above techniques to achieve higher performance with each product generation[72]. An example of the advantages of ILP is shown in Figure 3.1.

Static ILP parallelism on the other hand, can be achieved on software level by specialized compilers which are used by the specialized *Very long instruction word* (VLIW) processors. VLIW processors have multiple execution units organized in multiple pipelines and require the compilers to prepare the parallel instruction streams to fully utilize their resources[73].

Task / Thread-level parallelism: This high-level form of parallel computing is focusing on partitioning the application to be executed in distinct *tasks* or *threads*, that can be then executed simultaneously on different computation units. Threads can work on independent data fragments or even share data between them.

The developer has to use the *multi-threaded programming* paradigm in order to fully utilize the capabilities of the nowadays available multicore processors. This new programming paradigm requires a shift of mentality while programming new applications, which is somewhat difficult, as until recently programming was done *sequentially* (a single-thread holding the whole application).

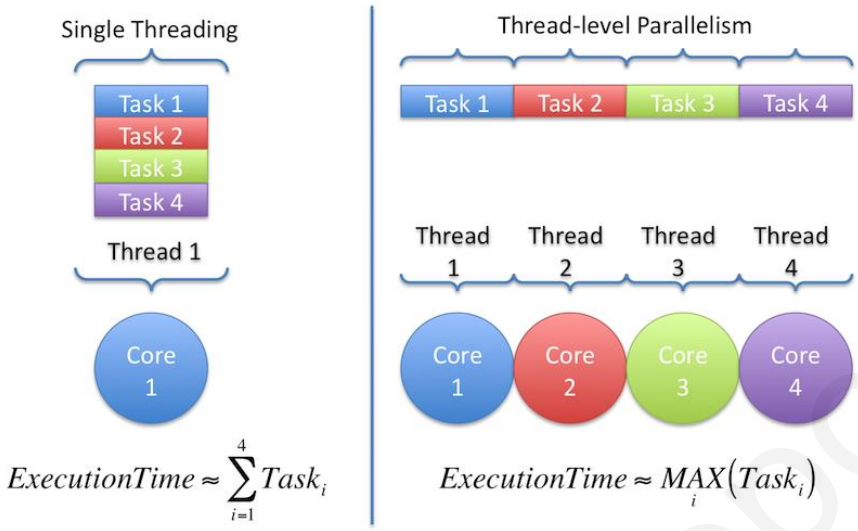


Figure 3.2: Thread-level Parallelism on a quad-core processor.
Executing a single-threaded application results on using just one of the available cores.
However, partitioning the application to four tasks via *multi-threading programming,*
all of the available resources are used and the execution time decreases considerably.

Modern Operating Systems (OS) have been used to provide a transparent utilization of all of the available cores of modern multicore processors by scheduling different processes[5] on different cores. However, this is not the case for executing efficiently complex bioinformatics applications, because the computation load of each process cannot be distributed on the available cores by the OS. As such, these applications must be re-developed with multi-threading in mind in order to achieve higher level of parallelism though thread-level parallelism and improve their performance. An example of this type of parallelism benefits on a quad-core processor is given in Figure 3.2.

Data parallelism: A form of high-level parallelism that in contrast to thread-level parallelism tries to partition the data to different available computation units instead. The cores execute the *same* task code over the data assigned to each. This form requires advanced developing skills to achieve, as the code executed should be executed independently on each data fragment. In addition, it is applicable to specific problems only.

[5] Process: A term used in Computing to refer to a computer application/program called for execution. Processes usually are handled by the OS. For example, double-clicking on MS Word application results on creating a process of MS Word. Multiple processes of the same application can be executing concurrently. For example, opening multiple browser windows etc. Each process is a independent instance of its respective program.

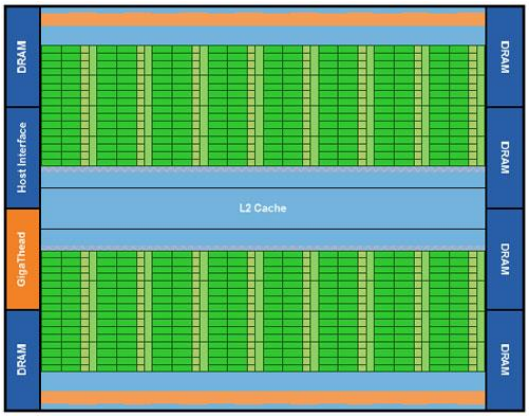


Figure 3.3: NVIDIA Fermi GPU Architecture Overview.
Note the hundreds of cores (green). This architecture has been developed with data parallelism in mind.

Data parallelism is the only available option for high-level parallelism in computer graphics because the graphic processor units (GPUs) are designed to execute each graphic processing task as fast as possible by partitioning each frame in regions. The task on command is then executed independently on each data region by their hundreds of processing units.

3.1.2 Limitations of Parallel Computing

Computer engineers keep pushing the limits of available parallelism by designing architectures with more resources and more processing cores – there are plans to build CPUs with 1000 cores [74]; however, no actual benefit will be yielded even in the case that a 1000-core processor is available tomorrow. Moving to hundreds and thousands of cores requires a radical rethinking in how software is designed [75].

Software is the actual roadblock that limits the full utilization of the available computing resources of today as well. The industry has moved towards a clear thread-level parallelism model since the early 2000s, however the existence of legacy software and the lack of parallel programming training for developers have not allowed for high-end computation systems to fully embrace the available parallelism.

Algorithms used in biomedical and bioinformatics applications have mostly been designed and implemented while using the traditional sequential way-of-thought. As such, these implementations fail - in most cases - to achieve the best performance possible on the

available state-of-the-art computing technologies. Attempts to parallelize these algorithms are bounded by *Amdahl's law* [76]:

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1)\left(B + \frac{1}{n}(1-B)\right)} = \frac{1}{B + \frac{1}{n}(1-B)} \qquad (1)$$

where *n* is the number of executing threads of the algorithm implementation, *B* is the percentage of the algorithm that is serial, *T* is the execution time, and *S(n)* is the speedup of the sequential (single-threaded) implementation versus the *n*-threaded one.

Amdahl's law is actually modelling the expected improvement of parallelized implementations of an already existing algorithm implementation. Equation 1 clearly shows that the execution time of the parallel version is limited by the percentage of the sequential (non-parallelized) code. For example, if 90% of an algorithm is parallelizable, the overall improvement can be no more than 10x.

An assumption made by *Amdahl's law* is that the problem size remains the same when the problem is parallelized. However, this is not the norm and Equation 1 just describes the *best-case scenario*. Splitting data for processing between many different processing units, results in I/O and memory overheads. These overheads negatively affect overall performance.

It is well known that an optimal assignment of threads to processing units is NP-hard, even when threads compete for a single processing unit, and the demands for this resource are known a priori [77]. In practice, the assignment problem is exaggerated by the unpredictable and dispersive nature of the resource requirements of the threads, as well as by memory, I/O or inter-process communication between threads of the same process [78].

Reducing the I/O and memory overheads as well as the communication delays between the threads is an open research problem. However, there is no optimal solution that can minimize the aforementioned overheads for all problems. Each application needs to be extensively profiled and an appropriate memory management and a resource allocation scheme need to be defined.

Memory can be shared between threads – i.e. all threads access the same data – or distributed – i.e. all memory is local for each thread and sharing is done by transferring data between

units or even a mixture of both. Shared memory offers a unified address space; however is prone to race conditions for multiple processing units request/access the same chunk of data. Distributed memory excludes race conditions, but forces the developer to think about data distribution and how the units need to communicate between them. A hybrid distributed/shared solution allows for scalable designs; however despite hiding the mechanism of data communication, it does not hide the delays.

The design and implementation of a resource allocation scheme for a particular computation problem is heavily impacted by the targeted computation engine's architecture and memory scheme as well [78]. Options and designs are different for multicore processors, clusters and dedicated hardware implementations. Our work on a resource allocation scheme for optimizing performance of a high-throughput Network-On-Chip communication architecture for QAM modulation can be found in [79].

Using new emerging computation platforms such as GPUs and FPGAs can offer hundreds of processing cores for executing inherently *computationally intensive* applications [17]. However, limitations do exist. Extreme parallelization of applications means that many processing cores will need data to work on. Thus, such applications are been transformed into *memory intensive* due to the use of parallelism [17].

GPUs have limited amount of memory per core and FPGAs have limited amount of on-chip memory. As such, memory becomes critical to the parallelized applications. Developers must rely on external memories such as DRAMs. However, these large external memory systems still lag behind in feeding the hundreds of available processing units in time, as the existing design practices (DRAM DIMMs for example) limit the available data bandwidth. For the aforementioned reasons, memory and I/O transferring rates pose a limit on performance gains when mapping applications on these now-emerging computation platforms.

3.1.3 Classification of Parallel Systems

In order to be able to differentiate between the various parallel systems and their potential, we will need *Flynn's taxonomy* [80][81]:

Single Instruction / Single Data (SISD): The most basic form of computing. A single instruction stream – i.e. a program – is executed one operation at a time on a single data stream. Traditional single-core processors fall under this category. No form of parallelism is exploited either in the instruction stream or the data stream.

Single Instruction / Multiple Data (SIMD): A system that executes a single instruction over multiple data streams *simultaneously*. SIMD systems exploit parallelism over independent data using data parallelism. The best example of this category is the GPU processors.

Multiple Instruction / Single Data (MISD): A rare type of computer system that is mostly used for fault tolerance. Multiple instructions operate on a single data stream. Results from all heterogeneous processing units must agree in order for the result to be committed back in memory. As such, MISD systems do not improve performance. Specialized processors used in space applications and systolic array architectures are falling under MISD class.

Multiple Instruction / Multiple Data (MIMD): Multiple processing units execute different instruction streams on different data streams. Modern day multicore processors and distributed systems fall under MIMD classification. MIMD systems either use shared or distributed memory space as discussed in *Section 3.1.2*. Computing experts further divide the MIMD class:

- ***Single Program / Multiple Data (SPMD):*** Multiple processing cores execute the same program. Tasks of the program are allocated on different cores with different input data in order to produce results faster. Most distributed computer systems are classified as SPMD and utilize distributed memory space and message passing for transferring information between their processing units.
- ***Multiprocessing Systems:*** Multiple identical processing cores are connected to a single, shared main memory and have access to all I/O via a system bus or a crossbar. Most modern day multicore processors are falling under this subcategory of MIMD systems. Processing is divided into multiple threads and each thread is executed on a

different core concurrently. The shared memory scheme may lead to deadlocks and unpredictable results. As such, special precautions need to be taken in the software implementation to ensure that data is accessed in a predictable way. These measures include semaphores, locks and barriers that allow the different threads to synchronize.

3.2 State-of-the-Art Computing Technologies

Recent technology advancements, gave birth to a number of promising computing technologies that help us move beyond the traditional computation platforms such as single-die processors and custom-built application specific integrated circuits (ASICs). In this chapter, we present the most prominent - already commercially available – technologies which can be utilized for achieving lower performance per watt and lower execution times for bioinformatics applications.

3.2.1 Multicore Processors

Processors (CPUs) are the dominant computation engine both in academia and industry. Almost all PCs, laptops, tablets etc. run on CPUs, as well as the high-end processing farms and cloud computing hardware consist of massive amounts of CPUs interconnected in a cluster. CPUs follow the *von Neumann* data processing paradigm in which standardized commands are issued and executed one by one [82]. Applications are executed as processes and are developed in software. The processes are scheduled on the available resources of a

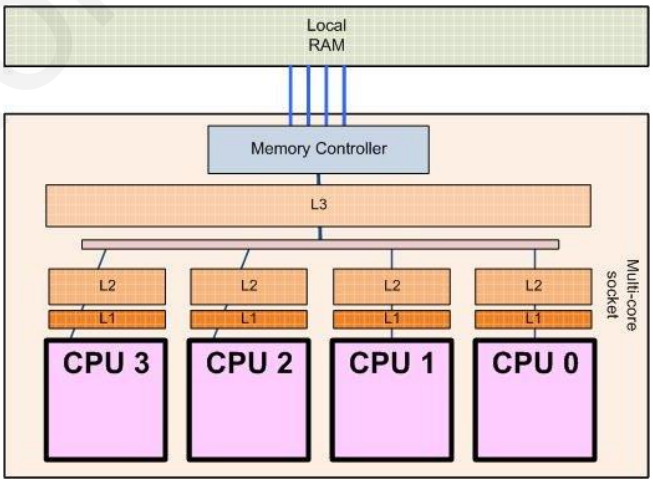


Figure 3.4: Outline of a quad-core processor.

processor-based system by an Operating System (OS) [83].

Performance has been improving by increasing the operational frequency while decreasing the size of the transistors, i.e. simply by following *Moore's law* [84] - stating that the number of transistors in integrated circuits will be doubling every two years. However, chipmakers understood that this paradigm can improve performance and energy efficiency no more. Further increase in frequency, offers diminishing results as three factors negatively affect performance:

- The *ILP wall*: Designing new processors with increased single stream instruction parallelism has become exceptionally difficult.
- The *power wall*: Power consumption is increasing exponentially while increasing the operational frequency of the processor. This can lead to additional problems such as overheating ICs and increased cooling budget.
- The *memory wall*: Memories are inherently slower than the CPUs connected to them. Faster CPUs (operating in higher frequency) would waste even more time waiting for data from memories.

Taking the above reasons into considerations, chipmakers have moved into multicore CPUs [85]. Modern multicore CPUs (a quad-core processor example is shown in Figure 3.4) consist of smaller processing units (cores) which essentially means that multiple code sections can executed in parallel (MIMD). The multi-threaded programming paradigm allows each process to contain a number of threads performing instructions over data (shared or independent) using the parallel computing paradigm. Parallel computing techniques help taking advantage of the available processing units (cores) of modern day multi-core processors.

Parallel computing on multicore CPUs, shows - in most cases – improvement over traditional sequential programming. However, limitations such as limited resources and the memory wall, do exist. Also, the OS still hinders performance and a multi-threaded application does not always take full advantage of the algorithms behind it. Nonetheless, using multicore processors as a computational engine can show performance degradation issues for real-time applications.

3.2.2 Graphic Processor Units

GPGPU (General Purpose computing on Graphic Processing Units) is a programming model that enables the use of the processing capabilities of the graphic cards (typically, SIMD systems, like the architecture shown in Figure 3.5) present in modern-day computers for executing code that traditionally was executed on CPUs.

However, the processing elements of GPU cards are designed specifically for graphics and thus are very restrictive in operations and programming. As such, GPUs are only effective for problems that can be solved using stream processing and their hardware can be used only when using specialized APIs such as OpenCL and nVidia CUDA libraries.

CUDA is the programming model developed by nVidia and implemented by the company's graphic cards [86]. CUDA libraries give access to a virtual instruction set for industry-standard programming languages (such as C/C++ and Fortran) that allows software developers to efficiently map applications to be executed on GPU cards.

A typical CUDA processing flow is: a) Copy data from CPU memory to GPU memory; 2) CPU instructs the GPU to start processing the desired code - *kernel*; 3) GPU executes the kernel code *in parallel*; 4) Copy the results from GPU memory back to CPU memory. The processing flow of the CUDA framework is shown graphically in Figure 3.6.

Figure 3.5: Outline of a nVidia Graphic Processor Unit (GPU).

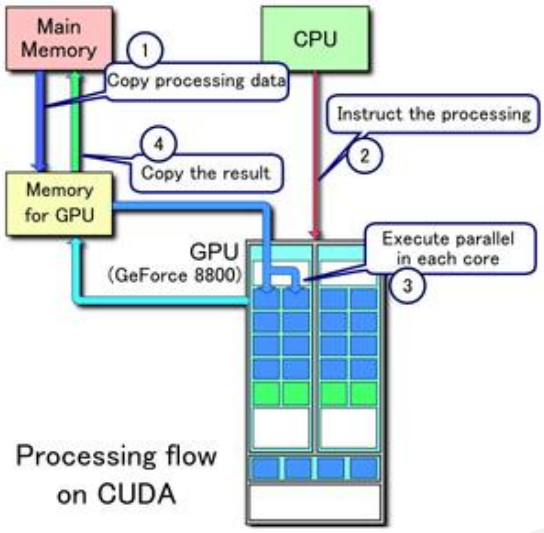


Figure 3.6: nVidia CUDA Processing Flow

GPUs can only process independent data fragments, but can process many of them in parallel. The GPGPU approach is effective when the programmer wants to process a large amount of data in the same way. GPUs process data sets *independently* and there is no way to have shared or static data. Algorithms must be rewritten to accommodate this peculiarity.

Using the GPGPU programming paradigm has other limitations, such as: unavoidable memory transfer overheads between CPU and GPU memories, limited memory resources per GPU core and adjustments needed for double floating number operations. However, it provides a tool that enables developers to harness the processing capabilities of the hundreds of cores present on nVidia's graphic cards.

3.2.3 Field-Programmable Gate Arrays

Field-Programmable Gate Arrays (FPGAs) are designed to be configured by the customer or designer after manufacturing and applications are developed as hardware architectures similarly with ASIC implementations [87]. FPGAs are closing the “gap” between processors and ASICs because as their name suggests they are an array of logic, RAM(BRAM) and I/O blocks.

FPGAs offer programmable interconnections between blocks and logical functions can be configured on the logic fabric. As such, everything is running concurrently like an ASIC

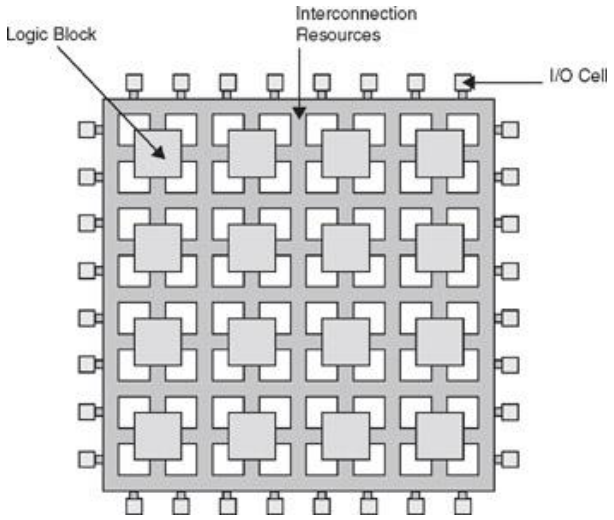


Figure 3.7: Outline of an FPGA.

implementation, while the logic fabric can be reconfigured with an updated architecture much like a software application can be updated on a CPU-based system. A typical FPGA architecture is shown in Figure 3.7. FPGAs are reported to offer high performance results for almost every application tested. Moreover, architectures can easily combine a custom-built core with soft CPUs for more efficiency and can be reconfigured in real-time, offering online fault-correction and more flexibility than ASIC.

However, implementing an application on FPGAs needs adequate knowledge of hardware description languages and needs longer time to be developed than a software solution. Nevertheless, FPGAs are a powerful computational platform that can offer orders of magnitude performance improvement over CPUs and GPUs in a number of applications. For this reason, FPGAs have already appeared as the main building blocks for specialized industrial and emulation platforms and as coprocessors for high-end mainframes.

3.2.4 Hybrid Systems

Mainstream FPGA integration is emerging through the use of standard bus protocols such as *PCI-express*, enabling FPGA platforms to be integrated into generic computing systems. More importantly, this enables integration alongside with GPUs, with their extremely high computational capabilities and parallel processing as GPGPU paradigm is proven an efficient software-based acceleration platform for processing massive blocks of independent data [88].

A number of attempts to develop such hybrid systems do exist. Axel [89] is a heterogeneous computing cluster architecture where each node consists of one CPU, one GPU and one FPGA. Each node runs its own Linux OS and these nodes are interconnected using traditional cluster architectures (Gigabit Ethernet and OpenMPI). QP Multi-Accelerator Cluster [90] is comprised of 16 AMD dual-core nodes running RedHat Linux; each node is equipped with four GPU cards and a Nallatech FPGA accelerator.

The "Chimera" desktop hybrid computing platform [91] has been developed as a first attempt to build a hybrid platform using commercially available components over a standard PCI-Express communication scheme. The "Chimera" system runs on a modified Linux OS in order to support direct memory access to the interconnected components.

To the best of our knowledge, there are no commercial ready-to-use heterogeneous systems that facilitate hybrid computational capabilities for bioinformatics and biomedical applications. As such, we have been actively building two heterogeneous systems comprising of CUDA-enabled GPU cards, FPGA boards and a host system with a multi-core general purpose processor and DRAM memory in order to test the capabilities that hybrid systems can offer for bioinformatics applications.

We have designed and built a high-end custom-built heterogeneous computing platform for bioinformatics and biomedical applications, called EVAGORAS in order to harness the benefits of hybrid systems for life sciences, medicine and biology [88]. We decided to pursue a highly modular design that will allow for continuous updates of the selected processing engines.

After consideration of the commercially available system boards, we decided to use Dini Group's DNBFC Cluster. This particular system board running a quad-core Intel Xeon® E3-1275 processor, met all of our specifications with the added benefit of being fully compatible with the Dini Group's DNBFC_S12_PCIe custom-built FPGA cards (13 Xilinx Spartan-6 FPGA chips each). Our hybrid system's complete hardware part list includes two of the aforementioned DNBFC_S12_PCIe multi-FPGA cards, two CUDA-capable Nvidia TESLA C2075 Computing Processors, 32GB DDR3 RAM and two SATA hard disks. Communication between accelerators is achieved through PCI-Express DMA links. The system has been decided to run Linux. A photo of EVAGORAS is shown in Figure 3.8.

A moderate low-cost hybrid system using off-the-shelf desktop components has been developed in our lab as well. This system called ASCLEPIUS has been designed to be a sandbox for our hybrid bioinformatics and biomedical applications. Moreover, this system should be an affordable - but on the same time powerful and efficient - solution for allowing small biology laboratories to contact breakthrough research despite not having the means to acquire high-end equipment. This system facilitates an Intel i7 processor, two nVidia GTX690 CUDA-enabled graphic cards and an ML605 FPGA Evaluation Board (Xilinx Virtex-6 FPGA-chip). All accelerators are interconnected with the system board through PCI-Express. The low-cost system has been build to run both Linux and Windows as we would like to run and evaluate hybrid applications using tools available for either OS. A picture of ASCLEPIUS desktop system is shown in Figure 3.9.

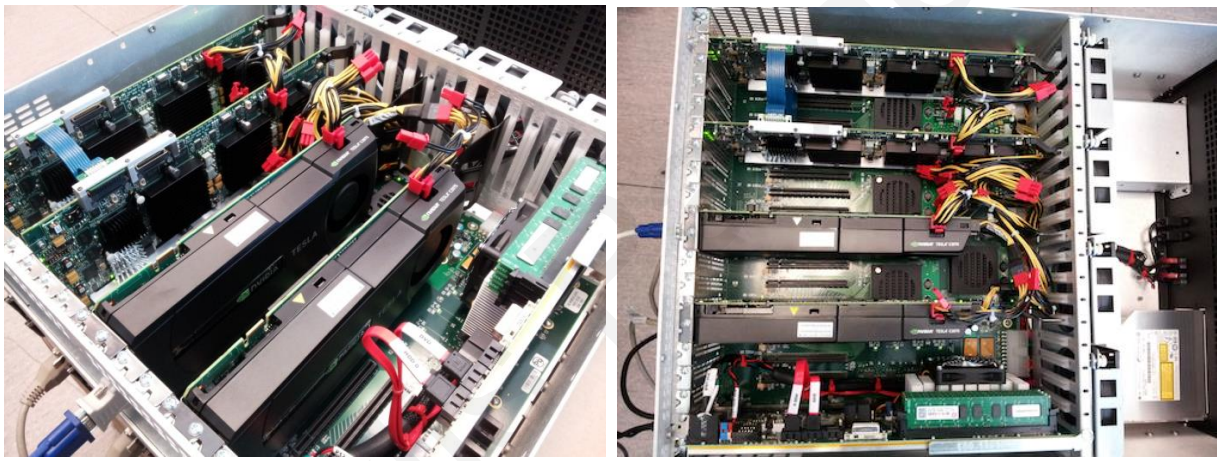


Figure 3.8: The state-of-the-art EVAGORAS hybrid platform.

Figure 3.9: Screenshots of the ASCLEPIUS hybrid platform

3.3 **Related Work**

3.3.1 Using GPUs for Bioinformatics and Biomedical Applications

The *General-purpose computing on graphics processing units* programming paradigm - i.e. using a graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU) - has been explored as a promising platform for bioinformatics; despite GPU's relatively high power demands, as they offer significant speedups over general-purpose systems. GPGPU-enabled GPU cards are composed of a massive amount of processing nodes and can only process independent data fragments, but can process many of them in parallel.

The GPGPU approach is effective for algorithms that process large amounts of data in the same way. GPGPU programming is an appealing solution for achieving higher performance while having reduced development times, as it can be utilized using a software programming style similar to API-based parallel programming used for multicore CPUs and clusters.

GPGPU programming is extensively used in bioinformatics since its introduction [86], as most of the applications used in the field are computationally complex and execute on massive datasets - characteristics that allow extensive utilization of the GPGPU programming capabilities. While not all algorithms perform well when executed on GPUs [92], previous attempts on developing CUDA-enabled implementations for bioinformatics applications showed promising results. The GPU-based implementation of the Smith-Waterman sequence alignment algorithm [93] showed speedups of 2x-30x over any previous implementation, while CUDASW++ implementation of Smith-Waterman showed speedups of 16x [94]. GPU-BLAST [12] showed speedups between 3x-4x for all test cases. Moreover, SOAP3 an alignment implementation using the Burrows-Wheeler Transform (BWT), showed speedups between 7.5x-20x [95]; while MUMmerGPU, another GPU-based tool for local pairwise alignment of complete genomes proposed in [96] showed 10-fold speedup as well.

GPU technology is proven effective in other bioinformatics applications as well. When used in Markov clustering, it showed speedups of 3x-8x [97]. Moreover, a RAxML implementation used for Phylogenetic Tree Construction showed improvements over the traditional software [98]. GBOOST for gene-to-gene interaction analysis showed 40x improvements [99]. Last but

not least in this non-exclusive list of such GPU-based implementations are the CUSHAW implementation for short-read assembly that showed improvements between 2x-10x [100], and CUDA-MEME program for motif discovery with speedups of 20x [101]. Taking the above non-exhaustive list into consideration, we can indeed deduct that algorithms used in bioinformatics applications are prime candidates for developing GPU-based versions.

3.3.2 Using FPGAs for Bioinformatics and Biomedical Applications

The protein database has been growing exponentially over the years, and the execution time of existing tools implemented traditionally in software grows exponentially even on high-end computer systems [102]. Recently, however, application-specific reconfigurable hardware solutions, utilizing FPGAs have emerged as promising alternatives. Several researchers proposed architectures for bioinformatics on FPGAs that exhibit performance improvements.

The reconfigurable architecture of the *BLAST* algorithm proposed in [103] showed speedups of over 45x over the *NCBI BLAST* software. The architecture proposed for *BLASTp* in [104] showed remarkable speedups up to 1400x over the software implementations of the same algorithm. Along the same lines, the FPGA-based implementation of *PSI-BLAST* proposed in [105] showed speedup over 20x compared to the existing software solutions.

In a relevant sequence analysis problem, an FPGA accelerator of the *GOR* protein secondary structure prediction algorithm [106], showed speedup factors above 430x over the original *GOR* and more than 110x over the multi-threaded software version [107]. Recently, a Network-on-Chip-based hardware accelerator for biological sequence alignment reported speedup over three orders of magnitude over traditional CPU architectures [16].

Preprocessing tools implemented on hardware can further reduce execution times since major tools – such as *BLAST* – have already been implemented on reconfigurable fabric with astonishing results. A prefiltering approach for further improving performance of *BLAST* is already implemented on reconfigurable fabric [108].

Other bioinformatics applications implemented as FPGA-based architectures include but are not limited to:

- MAFFT algorithm [18] - a progressive multiple sequence alignment method based on the Fast Fourier Transform (FFT) - which showed speedups of 10x-50x.

- T-coffee [18] - a progressive constraint-based method for multiple sequence alignment - with speedups up to 10x.
- Zuker [18] - an RNA secondary structure prediction algorithm - with speedups of 3x-10x.
- Predator [18] - a method which predicts the secondary structure of a protein sequence - with speedups of 30x to 50x.
- and an FPGA-based implementation of phylogenetic tree reconstruction which is a tree-like structure that shows the inferred evolutionary relationships among biological species [18], showed speedups up to 8x when compared against its multi-threaded software counterpart.

The implementations mentioned above, illustrate that reconfigurable technology offers significant advantages on the execution performance of CPU-intensive bioinformatics algorithms. More specifically, FPGA technology offers substantial speedups vs. conventional computers. Moreover, results in[104][21] indicate that reconfigurable technology also offers significantly lower energy requirements for the entire calculation when compared to cluster or grid platforms running the same algorithms. However, there are still several open issues before FPGA technology can be widely adopted by the bioinformatics community. These issues are not inherent problems of FPGA devices but of FPGA based platforms.

These platforms are typically sold with minimal software support, usually device drivers and proprietary OS. As such, popular bioinformatics algorithms and their software implementations, require extensive reverse engineering, so that the gap between the platform middleware and the algorithm is bridged. This does not happen in an easy manner, resulting several times in functional errors, either due to a wrongly designed hardware unit or misinterpretation of the algorithm's task flow graph when mapped on the hardware.

Furthermore, inherent FPGA capabilities, such as fast I/O interfaces, are not fully exploited yet, and, while internal on-chip FPGA memory (BRAM) is the critical resource of most of the proposed architectures, external memory (DRAM) still needs to be used in order to offer higher levels of parallelization. As such, even with the vast I/O bandwidth available to FPGAs, the existing external memory design practices (DRAM DIMMs for example) that result in the associated I/O memory problems appearing in general-purpose computing, still limit the performance of FPGA-based implementations.

However, recent multi-FPGA systems, with dedicated platforms designed to fulfil domain-specific computing demands, precisely to support the disadvantages observed in single FPGA boards, are becoming increasingly powerful and popular, and are promising revolutionary benefits when it comes to bioinformatics [17].

4

Parallelizing Software for Bioinformatics Applications

In this section the mCAST parallel software implementations are presented. These implementations were developed in order to evaluate the performance of multicore processors while executing bioinformatics applications and serve as a base for comparison against GPGPU and FPGA-based implementations. The original CAST implementation presented in [32] could be an initial comparison reference; however, for obvious reasons, comparing parallel software/hardware implementations with a sequential single-threaded software implementation is unfair. Thus, we developed a multi-threaded version of CAST. The work presented in this section has been published and presented in [19].

4.1 **mCAST Parallel Software**

The multi-threaded versions of CAST have been developed for fully utilizing modern OS thread scheduling capabilities by using C++ threading libraries for both Linux and Windows operating systems. The multi-threaded software takes advantage of the parallel nature of the first step of the CAST algorithm where the input sequence(s) are compared against the twenty amino acid residue types in order to find the respective highest scoring segments (See Figure 2.1 at Section 3.2). The search for those segments can be done in parallel since there are no data dependencies. Thus, the main program can create up to twenty threads for calculating the high scoring segments.

4.1.1 Evaluation and Results

The multi-threaded CAST (*mCAST*) was compared against the sequential CAST (*sCAST*) presented in [32]. We used both versions for comparison purposes, as the sequential CAST (*sCAST*) is optimized for uniprocessor systems and multi-threaded CAST (*mCAST*) is optimized for multicore systems. The two software versions were executed on two different computer systems: an *Intel Core2 Duo E8400@3GHz/4GB RAM* running *Windows XP*, and an *Intel Core i7 720@2.66GHz/8GB RAM* running *Windows Server 2008* for the sequence databases listed in *Appendix A: Proteomic Benchmarks*.

The application was built in Visual Studio 2012 and compiled using the integrated Microsoft C++ compiler. The execution times for sCAST and mCAST have been calculated while suppressing I/O system calls for fair comparison with future hardware architectures; the software under these conditions focuses on reading inputs and computing the algorithm, similarly with a hardware approach.

The reported evaluation results are the averaged execution time of twenty different executions of each software implementation over the same datasets. This measure was necessary for accurately acquiring the effect of the operating system on the overall execution. A windows batch script was developed to instruct all executions over all datasets and report the results in the dedicated ASCII output files.

TABLE 1: SINGLE-THREADED VS MULTI-THREADED SOFTWARE (AVERAGE WALL CLOCK EXECUTION TIMES)

	Intel Core2 Duo8400 @3GHz/4GBRAM Windows XP			Intel Core i7 720 @2.66GHz/8GB RAM Windows Server 2008		
	sCAST (ms)	mCAST (ms)	X[a]	sCAST (ms)	mCAST (ms)	X[a]
Case.I	238	288	1.23	133	85,1	1.56
Case.II	552	1185	0.46	532	63,6	8.36
Case.III	1031	673	1.53	1090	365.6	2.98
Case.IV	541	671	0.80	549	376.1	1.46
Case.V	1502	1797	0.83	931	535.5	1.74
Haem	4.2s	4.8s	0.85	4.0s	2.4s	1.66
p.falciparum	960s	1166s	0.82	528s	313s	1.69

a. Speedup of mCAST over sCAST

The wall clock execution times, presented in Table 1, indicate that *mCAST* offers minimal performance improvements for the first system, as the *Intel Core2 Duo E8400* supports up to two threads in parallel. The remaining threads have to wait to be scheduled and the resulting continuous context switch rather burdens performance than helping. The second system supports *mCAST* much better, as its processor supports up to eight simultaneous threads.

The speedup however does not reach the ideal 8x (except Case.II) because of the frequent memory accesses and the time used by the OS. Thus, the speedup for small-to-average sized sequences is not significant as the performance gain stemming from the parallel execution is of the same order as the time consumed by the OS for creating and scheduling the threads. The results also indicate that multithreading does not help for datasets having longer sequences. For Case.II and Case.III datasets, the speedup reaches above 2x, since the sequences in these datasets need more iterations – larger number of LCRs – and the time spent on memory accesses and by the OS for context switch, becomes less important. mCAST therefore, still does not achieve full potential for several datasets due to memory and control issues.

4.1.2 Optimizations

The initial *mCAST* presented in [19] was a straightforward parallelization of the single-threaded algorithm of [32]. In [19], the identification of the HSS for each amino acid type was a two-step process. The first forward, step is the actual comparison of the sequence against the homopolymer and it results in a vector of scores for each position in the sequence.

During the second, backward step, regions spanning from a score of zero to a local peak score are detected and homogenized; this results in a vector containing regions of uniform scores out of which the one with the higher score is picked as the HSS.

In the *mCAST2.0*, we optimized the execution path and the backward step is only executed if the maximum score detected in the forward step is equal or greater than the score threshold. This almost halves the execution time for the detection of each HSS and since CAST is an iterative algorithm the overall speed-up is, in many cases, much better. Moreover, additional optimization of memory operations and dynamic thread handling control further improve wall clock execution time of the application.

TABLE 2: MCAST VS MCAST 2.0 (AVERAGE WALL CLOCK EXECUTION TIMES)

	Intel Core i7 720@2.66GHz/8GB RAM Windows Server 2008				
	sCAST (ms)	mCAST (ms)	mCAST 2.0 (ms)	X[a]	X[b]
Case.I	133	85	9	14.8	9.44
Case.II	532	64	239	2.2	0.3
Case.III	1090	366	472	2.3	0.8
Case.IV	549	376	290	1.9	1.3
Case.V	931	536	229	4.1	2.3
Haem	4.0s	2.4s	1.7s	2.4	1.4
p.falciparum	528s	313s	15s	35.2	20.9
Uniprot.Sprot	571s	310s	110s	5.2	2.8

a. Speedup of mCAST 2.0 over sCAST
b. Speedup of mCAST 2.0 over mCAST

We tested the efficiency of the improved multi-threaded *mCAST2.0* on a *Intel Core i7 720 @ 2.66GHz/8GB RAM* system running *Windows Server 2008*. A Windows batch script was developed to execute twenty runs of each algorithm for every dataset. The resulting wall clock execution times were averaged and are presented in Table 2.

The mCAST2.0 outperformed the initial multi-threaded software by halving the wall clock execution time on average, even reaching a 20x speedup in the difficult (over 70% of its sequences having LCRs) p.falciparum dataset. As such, we selected to use the superior *mCAST2.0* software in order to fairly evaluate future massively parallel software implementations and reconfigurable hardware architectures.

4.2 **Using the OpenMP Parallel Programming API**

OpenMP (Open Multi-Processing) is an Application Programming Interface (API) that supports multi-platform shared memory multiprocessing programming in C/C++ and Fortran. OpenMP is managed by the OpenMP Architecture Review Board consortium which includes computing companies such as Intel, AMD, IBM, HP, Fujitsu, Nvidia, Texas Instruments and Oracle Corporation [109].

OpenMP uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer [110]. The OpenMP API consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior [111].

4.2.1 Introduction to OpenMP Programming

The OpenMP API is implementing multi-threading by allowing a master thread to fork a number of slave threads that execute different tasks on the available computing resources. The slave threads run concurrently allocated in different processing cores while controlled by the Operating System (OS). As soon as all slave threads complete their tasks, runtime control returns to the master thread where it continues onward until all of its tasks are executed. An example of OpenMP application is shown in Figure 4.1.

OpenMP allows for the programmer to mark the sections of the code needed to be executed in parallel using a specialized compiler specific pre-processing directive (*#pragma*) allowing of the formation of the parallel slave threads and their control. Each thread has an id attached to it which facilitates tuning and synchronization among the slave threads. The runtime environment allocates threads to processors depending on usage, machine load and other factors [109].

The slave threads execute their tasks completely independent of each other by default. Work and data sharing constructs are available to be used in order to partition a task among different threads or how each thread has access to shared data. Both task parallelism and data parallelism can be achieved using OpenMP in this way [111].

Thread synchronization is extremely crucial when threads share data among them; and as such OpenMP offers synchronization constructs that allow to fine tune data access and make sure that the threads are affecting shared data with the intended order.

Using the OpenMP API libraries for implementing multi-threaded applications has both advantages and disadvantages. A major advantage of using OpenMP is that we can create portable multithreading code that can be executed in a number of different platforms and OSs without the need of using platform-specific primitives. Learning OpenMP has a much smaller learning curve than learning to use platform-specific multi-threading programming and its use is much simpler as task decomposition and data is handled automatically by directives. Moreover, OpenMP is scalable: no changes are needed for an OpenMP application to execute on new multicore processors featuring an increased number of processing cores [111].

Existing sequential applications can easily be modified to be multi-threaded through the use of OpenMP for a number of reasons. Firstly, the original sequential code does not need to be modified when parallelized with OpenMP thus reducing the chance of introducing bugs. An application can be parallelized in stages, as the master thread approach used in OpenMP allows for sequential code to exist alongside with parallelized sections. Moreover, OpenMP applications can run on legacy systems since its constructs are treated as comments in sequential compilers.

Despite having major advantages over traditional multi-threading programming, OpenMP has a number of important disadvantages as well that can greatly affect performance gains. OpenMP API has been proven to run efficiently only on shared-memory processing systems

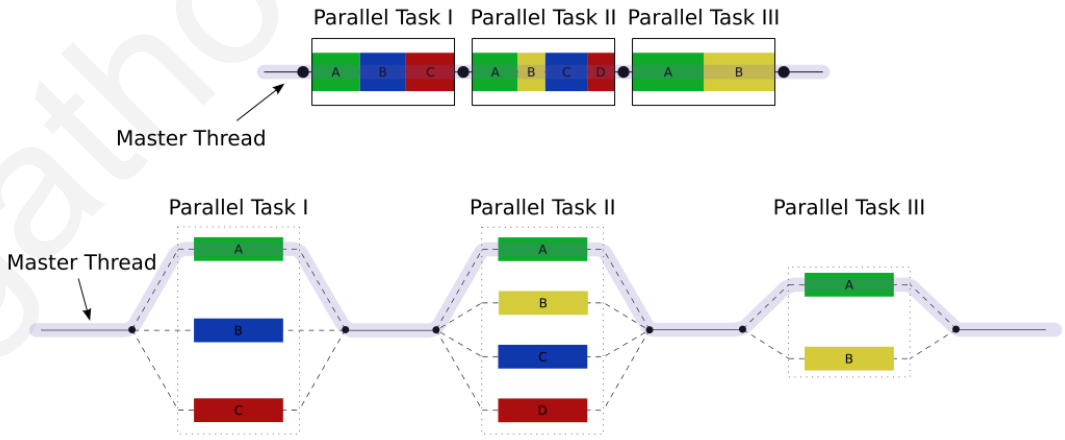


Figure 4.1: Outline of an OpenMP multi-threaded application.

and requires a compiler that supports it in order to use it. Another disadvantage is that the scalability of OpenMP applications is greatly affected by the memory architecture of the multicore system running it.

OpenMP lacks of fine-grain thread control mechanisms and does not support compare-and-swap [112] data sharing among threads. There is no reliable mechanism to handle errors and OpenMP construct-using approach can lead to wrong sharing of tasks or data among threads, or even to difficult-to-detect synchronization bugs and race conditions [113].

Last but not least, multi-threaded applications can have worse start-up time than their original sequential versions as the OS has to create and handle a number of different threads on every parallel code section instead of handling just one. If the parallelized code sections of an OpenMP application are small enough, we could get worse execution time than the execution time of the sequential code for the same application.

4.2.2 OpenMP CAST Implementation

We evaluated the OpenMP API for its potential in parallelizing bioinformatics and biomedical applications by using it to parallelize the CAST algorithm. We selected CAST as we have already gained experience in its parallelization potential though the use of traditional multi-threading approaches which we have presented extensively in Section 4.1.

```
for(counter=0;counter<20;counter++)
    {
        param[counter]=counter;
        #ifdef WIN32
            handle[counter] = (HANDLE) _beginthreadex(NULL, 0, &ThreadProc, &param[counter], 0, &threadID[counter]);
        #else
            handle[counter]= pthread_create(&threadID[counter], NULL, &ThreadProc, (void*) &counter);
        #endif
    }

    #ifdef WIN32
        WaitForMultipleObjects(20,handle,TRUE,INFINITE);
     #endif
     for(counter=0;counter<20;counter++)
     {
       #ifdef WIN32
            CloseHandle(handle[counter]);
       #else
            pthread_join(threadID[counter], NULL);
       #endif
     }
```

Figure 4.2: Code segment of a traditional multi-threaded software implementation.

Comparing an OpenMP version of CAST against the optimized *mCAST 2.0* should result to interesting information that can help us understand the benefits and issues of using the OpenMP approach in other bioinformatics applications as well. We developed the OpenMP-based CAST software using the OpenMP3.0 API in C++. We transferred all optimizations we have done in *mCAST 2.0* in OpenMP-based parallel CAST as well in order to be fair in our evaluation. We used this unique opportunity to evaluate all available task and data sharing constructs of OpenMP in order to evaluate not only its performance, but its ease to use for developing parallel multi-threaded software.

A code segment of the traditional multi-threaded programming approach used in *mCAST 2.0* can be seen in Figure 4.2. This code segment shows why traditional parallelizing techniques are not portable: *mCAST 2.0* needed to be able to run on both Windows and Linux -based systems, however each OS handles threads differently. Windows, for example, uses the *_beginthreadex()* function calls while Linux uses *pthread_create()* for creating new threads. As such, a cross-platform application like mCAST 2.0 needed to have compiler directives for both OSs. Threads are executing the *ThreadProc* task (identify HSS for every amino acid residue in parallel) and upon completion the runtime returns to the main thread to complete the computation for each iteration.

```
/* Processing Threads */
#pragma omp section
{
    printf("STARTING processing Thread, Thread ID = %d\n",omp_get_thread_num());

    #pragma omp parallel shared(fileIn,queue,thread,Sequences,seq_finish,allAA,Matrix) private(h,ok) num_threads(No_of_Treads-2)
    {
        printf("STARTING CheckAA Thread, Thread ID = %d\n",omp_get_thread_num());
        while (seq_finish==0)
        {
            ok=0;
            h=0;
            #pragma omp critical(thread)
            {
            if (thread.size>0)
            {
                h=thread.pop(&thread); //Extract next available sequence
                ok=1;
            }
            }

            if(ok==1)
            {
                detectBiased(Sequences[h].Sequence, Sequences[h].Title,Sequences[h].Length);
                #pragma omp critical(queue)
                {
                    queue.push(&queue,h);
                }
            }
        }
    }
}
```

Figure 4.3: Code segment of an OpenMP software implementation.

A code segment of the OpenMP-based CAST implementation is shown in Figure 4.3. The programming style can be very different as OpenMP creates and handles threads autonomously. The whole code segment of Figure 4.2 is replaced with the single construct:

#pragma omp parallel

This construct is enough to handle the creation and management of threads while the code remains portable among different OS. However, this coarse-grain parallelism is allowing for synchronization bugs to appear for the data that the slave threads share between them (defined in the *shared* parameter in the *#pragma omp parallel* construct).

Extra caution needs to be taken in order to assure that the data is accessed and modified correctly. As such, we had to use the *#pragma omp critical* construct to denote the critical code sections where each thread has to wait for its turn to access important shared data.

Each slave thread executed the *detectBiased* task which is used to identify the LCRs in each input sequence. The high-level abstraction of OpenMP allows for creating parallel applications that are indeed readable without the need to rewrite every piece of code.

4.2.3 Evaluation and Results

We evaluated the implemented OpenMP-based parallel CAST software using the same *Intel Core i7 720 @ 2.66GHz/8GB RAM* system running *Windows Server 2008* that we used for evaluating *mCAST* software versions. This machine has been selected since is an average off-the-shelf multicore computation engine and we will be able to have fair results for both multi-threaded programming approaches.

The OpenMP CAST was developed in Visual Studio 2012 (just like mCAST 2.0) using the integrated support for OpenMP directives alongside with the Microsoft C++ Compiler. A windows batch script was responsible for acquiring and extracting the averaged execution times of twenty executions over each dataset.

We run both OpenMP and *mCAST 2.0* multi-threaded software implementations with the proteomic benchmarks of *Appendix A: Proteomic Benchmarks* as selected test cases. The results were greatly affected by the datasets:

For the *haem* proteomic benchmark database, *OpenMP outperformed mCAST 2.0 with a speedup of 1.73x.* On the other side of the spectrum was the results for *plasma* database; *mCAST 2.0 has been proven to be quite faster with a speedup of 3.3x over its OpenMP counterpart*.

The reasons for these difference in performance results simply rely with how each approach manages the available computing resources. Traditional multi-threading allows for fine-grain control of both the threads and shared data. This can be very helpful on computationally heavy cases such as the *plasma* benchmark database.

On the contrary, OpenMP parallelism works on a coarser level and each slave thread handling an input sequence with large numbers of LCRs can cause the rest to wait much longer on the synchronization barriers of the master thread. This can greatly affect performance. On the other hand, a database like *haem* with average LCR numbers, can greatly benefit from leaving the thread handling to the OS in the case of OpenMP-based parallel programming.

To conclude, selecting a multi-threaded API like OpenMP for parallelizing bioinformatics applications has proven benefits such as smaller developing times, ease of use over existing software and still yield higher performance results. However, trade-offs do exist! There are cases where using traditional fine-grained multi-threading techniques can help achieving even higher performance results through better task and data sharing control but on the cost of tedious programming and extensive code rewriting. The final decision is left with the reader, as no approach is proven to clearly overwhelm the other.

4.3 **Closing Remarks**

In this chapter, we presented our attempt to design and implement multi-threaded versions of a popular bioinformatics application in order to explore the potential of using multicore CPUs as a platform for computing bioinformatics and biomedical applications.

We have implemented multi-threaded versions of CAST that outperformed the original sequential CAST [32] by twenty times. The multi-threaded versions of CAST were created using both traditional multi-threading techniques and the OpenMP API library. Our results showed that despite no multi-threading technique was able to differentiate its performance

form the other, the trade-offs we identified could make one more attractive than the other depending on the application at hand.

Multi-threading on multicore CPUs is an attractive solution for accelerating bioinformatics and biomedical applications. However, we need to explore GPUs as a processing platform as well. The next chapter is discussing GPU-based implementations of CAST and how the GPGPU paradigm can positively impact the application's performance.

5 GPGPU: A Paradigm for Bioinformatics and Biomedical Applications?

Modern GPGPU-enabled graphic cards offer hundreds of processing elements that as we discussed in *Chapter* 3 can potentially be an attractive computing platform for accelerating applications in the fields of medicine and biology. In order to evaluate the benefits and issues rising from using GPUs for biomedical and bioinformatics applications, we have implemented CUDA-enabled versions of the CAST algorithm and the subthalamic nucleus simulation LVN model, both presented in *Chapter* 2.

5.1 **GPU_CAST**

In this section, we present the first attempt to accelerate the performance of CAST using the GPGPU programming paradigm. The GPGPU-enabled version developed in this work is compared against the optimized multi-threaded version of CAST presented in earlier work in order to evaluate the proposed implementation. The work presented in this section has been published and presented in [20].

5.1.1 Proposed GPGPU Implementation

GPU_CAST follows a similar workflow with the optimized multi-threaded software presented in [21]. However, some fundamental differences do exist in order to allow higher performance when executed on the GPU. The workflow of our implementation is shown in Figure 5.1.

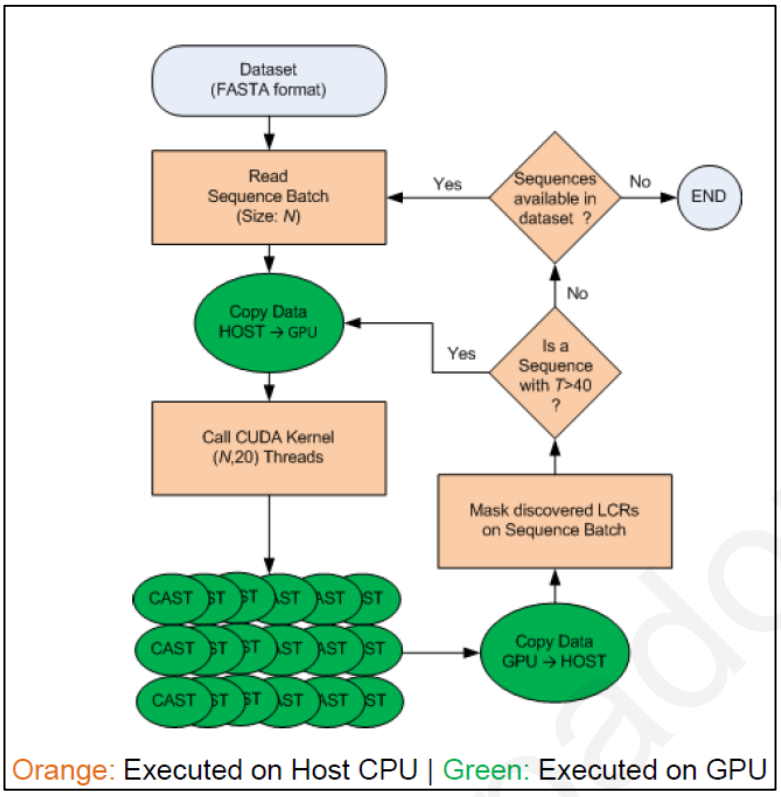


Figure 5.1: GPU_CAST Execution Flow

The sequence data are fetched in batches of predefined size (N=50, due to GPU shared memory size limitations). For each batch, a kernel execution is prepared by the host CPU. Firstly, the batch's data is transferred from the main memory to the GPU memory and stored in the GPU's global memory. Then, the kernel is invoked.

The *GPU_CAST* calculation is executed in 20*N threads on the GPU card's cores. Each set of 20 threads, corresponds to the CAST execution over the homopolymers for each one of the N sequences processed in parallel, thus making efficient use of the GPU card’s processing elements. The data resulting from each set of threads, will then be synchronized in the GPU's shared memory and used by thread 0 of the set to calculate the current iteration's LCR.

Upon kernel completion - all threads terminated - the host CPU will regain control and mask each sequence in the batch based on the kernel results. This process iterates if needed; the kernel is invoked on the updated batch in order to discover the remaining LCRs of its sequences. Next, the host CPU proceeds to fetch the next batch of N sequences. The GPU_CAST developed in this work, was programmed to maintain the same input/output interface as the original tool while ensuring it produces identical results.

The proposed *GPU_CAST* tool was evaluated against the optimized multi-threaded version – *mCAST 2.0* – proposed in [19]. We used a computer system having: *Intel Core i7 3960X @ 3.3GHz, 32GB RAM* running Microsoft Windows 7, equipped with a *nVidia GeForce GTX690* graphic card.

The GPU_CAST application was compiled using the nVidia's nvcc compiler driver integrated in Visual Studio 2012. The reported wall clock execution times were averaged over twenty different executions on every dataset using a windows batch script - just like the mCAST 2.0 evaluation. We compared the execution times, while executing the benchmark datasets described in *Appendix A: Proteomic Benchmarks*.

5.1.2 Results

The wall clock execution times of *mCAST 2.0* and *GPU_CAST* are shown in Figure 5.2 expressed in milliseconds. GPU_CAST when executed on the nVidia GeForce GTX690 graphic card shows no significant execution time differences for the relatively small *haem* and *p.falciparum* (plasma) datasets.

The GPGPU implementation paradigm shows its benefits when GPU_CAST executes over *Viral.1.Protein* (viral) and *Uniprot.sprot* datasets and **speedups between 5x-11x are achieved**.

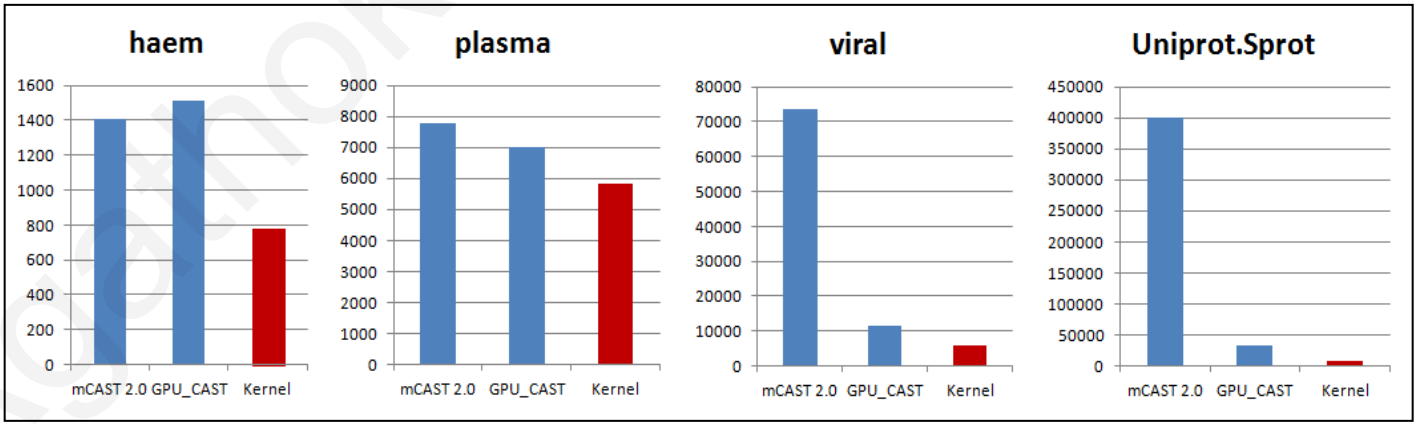


Figure 5.2: Wall Clock Execution Times (ms) of mCAST 2.0 and GPU_CAST . Total and Actual Kernel execution time.

We can see that the CAST algorithm is accelerated when executed on the GPU card; however the speedup greatly depends of the dataset. For example the execution time is cut in tenth for *Uniprot.sprot* but for the smaller *haem* dataset *GPU_CAST* yields similar (yet slower) execution times as *mCAST 2.0*.

While GPU_CAST is proven to be more efficient for executing the CAST algorithm over large datasets, a comparison between Figure 5.2's *GPU_CAST* and kernel columns, shows that a significant amount of time is consumed in tasks different than actually performing CAST (shown by the kernel column). As such, further analysis is required for identifying these overheads.

Executing software programs directly on the GPU card, the data has to be present on the GPU memory, as already discussed in Section 3.3.2.2. As such, the data needed for calculating CAST - the information of the sequences in the dataset and the substitution matrix - must be transferred to the GPU memory before the CAST kernel executes, and move the results from the GPU back to the host memory as well.

Taking the above into consideration, memory transfers between the CPU and the GPU have to be evaluated as potential performance bottlenecks, in order to determine their actual impact on GPU_CAST execution times. Figure 5.3 shows the time consumed in CUDA API-related tasks such as memory transfers and kernel execution.

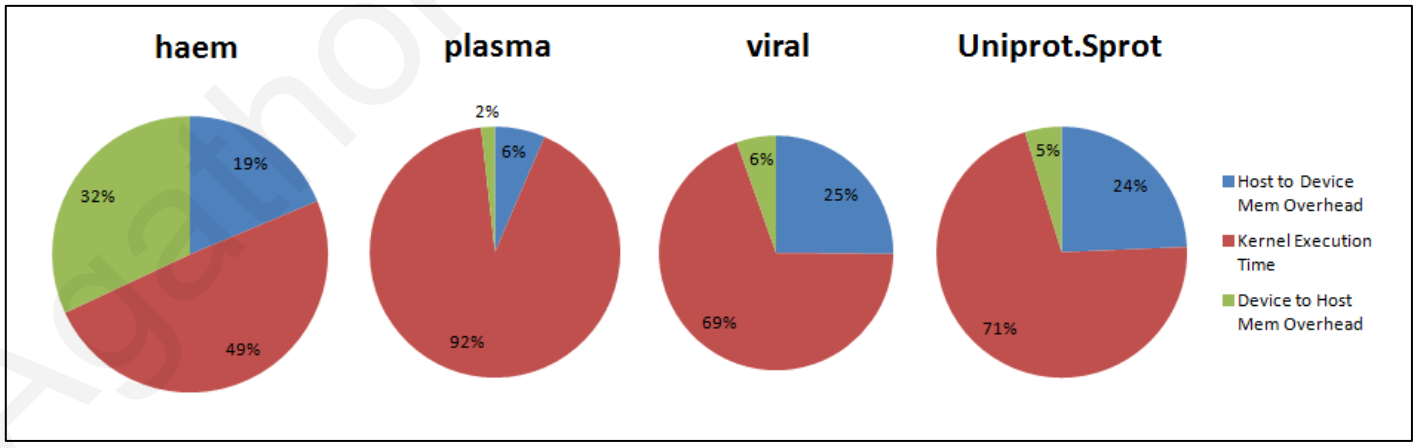


Figure 5.3: CUDA API Times (ms) Breakdown (Memory Transfer Overheads and Kernel Execution Times)

5.1.3 Memory Transfer Overheads

Memory transfer overheads sum up to 51% of the CUDA tasks for *haem*, 31% for *viral.1.protein*, and 29% for *Uniprot.sprot* dataset respectively, while remain less than 10% for executing the *plasma* dataset. We can understand that the memory overheads are greatly affected by the size and the characteristics of the dataset.

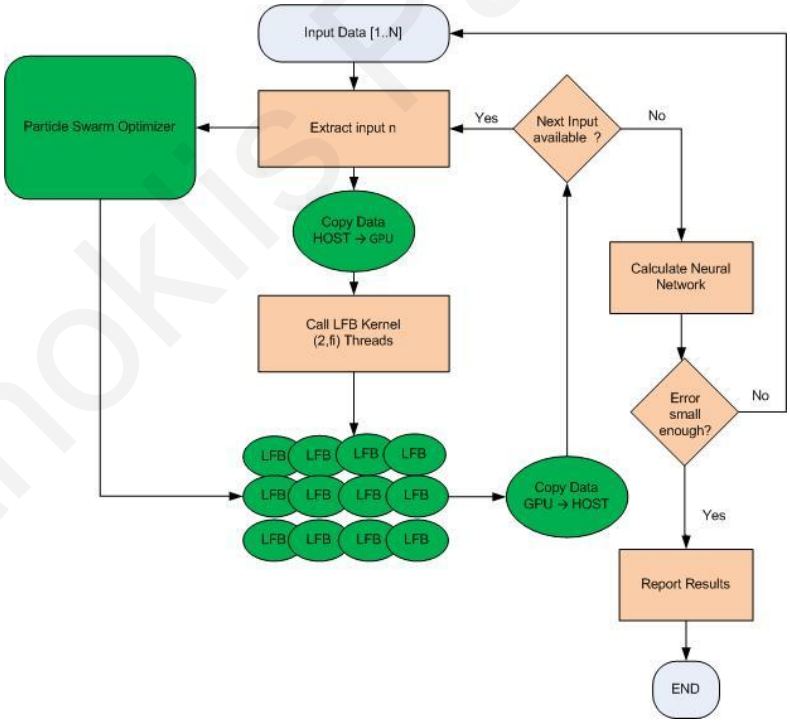
For instance, the low memory overheads for plasma dataset can be justified by the fact that more than 70% of the sequences in the dataset have at least one LCR (as shown in Table I), thus CAST kernel has to iterate more. The dynamic nature of CAST algorithm makes difficult to accurately predict memory overheads, however *the average memory overheads of all the datasets is calculated to be 30% which is a significant portion of the overall execution time.*

5.2 GPU-based LVN Modeling

We have designed a GPU-based implementation for the STN LVN model using nVidia's CUDA framework. This design is based on the analysis on the application's available parallelism reported in *Chapter 2.2.3*. The LVN model has two distinct components: the Particle Swarm Optimizer (PSO) and the actual LVN component integrating both the LFBs and the neural network modules.

5.2.1 Proposed GPGPU Implementation

The application is initialized with all input data present on the host system's main memory. An overview of out implementation is provided in Figure 5.4. As already discussed in *Chapter 2.2*, the model's simulator relies on a particle swarm optimization component to generate the weights necessary for the LFBs and the neural network. This PSO component - depicted as a green square in Figure 5.4 - is described later in this chapter.



Green: Executed on GPU | **Orange:** Executed on host CPU

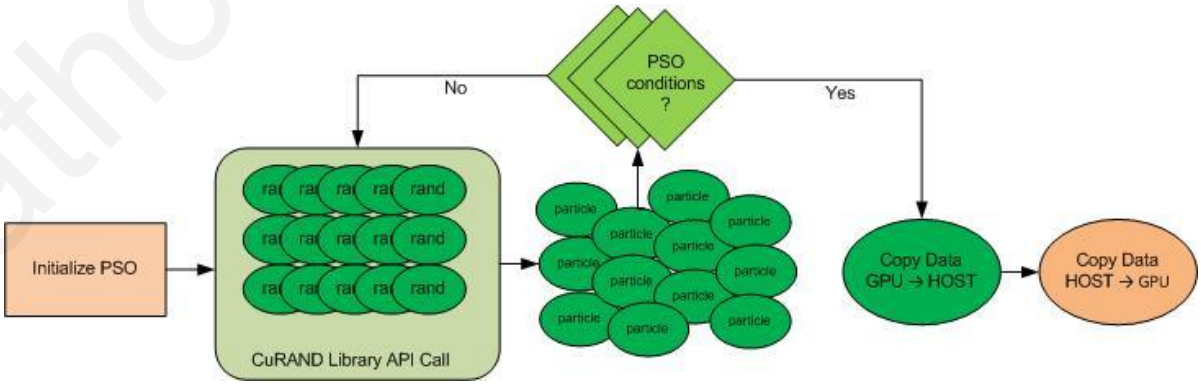
Figure 5.4: GPU-based LVN model implementation.

Following the analysis of *Chapter 2.2*, we have designed a GPU kernel responsible for calculating the LFBs of the model. The kernel invokes enough GPU threads to calculate all LFBs for each input in parallel. The data dependencies between *n* and *n-1* inputs do not allow for further increasing concurrent calculations. As such, the LFB kernel is called iteratively for each input *n*. The invoked GPU threads need to receive the weights generated by the PSO component in order to complete their calculations.

PSO, which is executed by a second GPU kernel, should transfer its output into the LFB kernel. Unfortunately, CUDA does not allow sharing data directly between kernels. There are two options for bypassing this issue. First option is sharing the data using the GPU's global memory as a buffer. This option however, is not certain to work with all GPU card configurations currently and is not fast enough. Second option is to transfer the PSO kernel's results to the LFB kernel though the host CPU's memory. We used this option in our implementation since the data always arrive correctly. However, the additional memory overheads and the fact that LFB kernel cannot be invoked before PSO kernel returns its results to the main memory, can both be performance degradation issues.

The neural network is calculated just as the LFB kernel is executed over all available input data. Since the network is small enough, there is no need to attempt designing a GPU kernel for it. As such, neural network calculations are carried out on the host system's CPU.

If the LVN model results are justifying the desired error margins, the application ends. If this is not the case, the application iterates once again through PSO and LFB kernels until the error is less than the desired predefined threshold.



Green: Executed on GPU | **Orange:** Executed on host CPU

Figure 5.5: GPU-based PSO module.

The PSO module's outline is shown in Figure 5.5. The module is initialized though a host CPU call and uses the already available CuRAND library for generating the required uniformly random initial features for each particle. The PSO kernel invokes enough threads for calculating position and velocity for all particles in the swarm in parallel. Each thread calculates the position and speed of its assigned particle using CuRAND function calls to generate the required random variables. Our design does not exploit the dimensions of the swarm. This feature is left as future work.

The global best position is stored in shared memory while the local best position is stored locally for each particle. All GPU threads update the global best position and check if PSO conditions are met upon completion of each iteration. If one GPU thread reaches the PSO conditions, it rises a shared flag variable that allows all threads to know that PSO reached the desired conditions. The GPU threads continue to iterate in order to improve their respective particle's position. However, GPU threads terminate their flow as soon the flag is raised. The PSO results are then transferred to the host CPU's memory in order to become available for the LFB kernel.

5.2.2 Evaluation

The GPU-based LVN model described in the previous section was implemented in Visual Studio 2012 and compiled using the nVidia's nvcc compiler driver integrated in Visual Studio 2012. The execution times were averaged over twenty different executions on every dataset using a windows batch script.

Existing STN LVN models are implemented and executed in the MATLAB environment. This approach allows for the biomedical engineers to use the model in a familiar programming environment. However, it does not allow for fair comparison between the existing implementations and the proposed GPU-based version for the following reasons:

MATLAB does offer multi-threading and CUDA-based execution; however, the need for higher performance was evident in all of the meetings with the researchers using the existing MATLAB model. MATLAB has significant Java environment overheads and offers only

coarse-grain parallelization. On the other hand, our GPU-based approach uses a different development environment and uses fine-grained parallelization techniques.

The proposed GPGPU implementation's results have been tested against the MATLAB generated results for a number of real world data samples and the proposed GPU-based model is faster than the MATLAB equivalent for all runs.

We were able to confirm that the GPU generated results have - in average - less than 2% value difference at 7 decimal places accuracy for all input datasets. This is considered adequate since no two set of results for the same input are identical even in MATLAB due to the randomly generated weight values and the relaxed terminal conditions.

Despite the proposed GPU-based model is faster than the MATLAB equivalent for all runs, fairly evaluating the performance of the proposed GPGPU design against a multi-threaded CPU implementation is on-going work since we need to develop a multi-threaded version in the C++ programming language.

5.3 **Closing Remarks**

In this chapter, we explored the possibility of using the GPGPU programming paradigm for developing high-performance implementations tackling problems in biology and medicine. GPUs offer hundreds of processing elements that can be used to accelerate data-intensive applications. However, limitations do exist; the data to be processed needs to be present on the GPU memory, and data chucks processed in parallel should be independent from each other.

The GPU_CAST implementation showed that a GPU-based version of CAST is at least on par with the multi-threaded mCAST2.0 and there are cases where the GPGPU paradigm allows for speedups of 10x over mCAST2.0. Memory transfer overheads are calculated to account for 30% of the overall execution time.

Future work should include a combination of multi-threaded CUDA-enabled version of CAST since we believe that partitioning the data between CPU threads that each drives a different CUDA-enable device will further diminish memory overheads and achieve even higher

performance. If this attempt proves to be successful, it will be another step towards our goal of building a truly hybrid CPU/GPU/FPGA system for biology and medicine.

The last step towards a fair comparison of the available computing technologies should be the exploration of FPGAs as platforms for bioinformatics and biomedical applications. The next chapter presents and discusses our FPGA-based implementations.

6 Reconfigurable Hardware as a Computing Platform for Bioinformatics Applications

Reconfigurable hardware, such as FPGAs, are already proven to be successful alternatives for achieving higher performance for biomedical and bioinformatics applications (*Chapter 3.3*). However, we need to fairly compare these performance gains against the results yielded from GPU-based and multi-threaded software for the same applications. As such, we designed and implemented an FPGA-based reconfigurable CAST hardware architecture which we evaluated against the software implementations presented in previous chapters of this thesis. Moreover, our first attempt to develop an FPGA-based suffix array construction algorithm for building the data structures needed in de novo sequence assembly is presented in this chapter as well.

6.1 **FPGA-based CAST Hardware Architecture**

In this section, the first attempt to implement a hardware acceleration architecture that can be used for identifying and masking LCRs in protein sequences, is presented. The work presented in this section has been published in [19] where the main module has firstly been presented. We expanded our architecture by developing a complete system prototype that can *massively* identify and mask LCRs in [21]. The necessary I/O communication and sequence allocation schemes have been developed as well in order to optimize the overall system performance in actual real world trials.

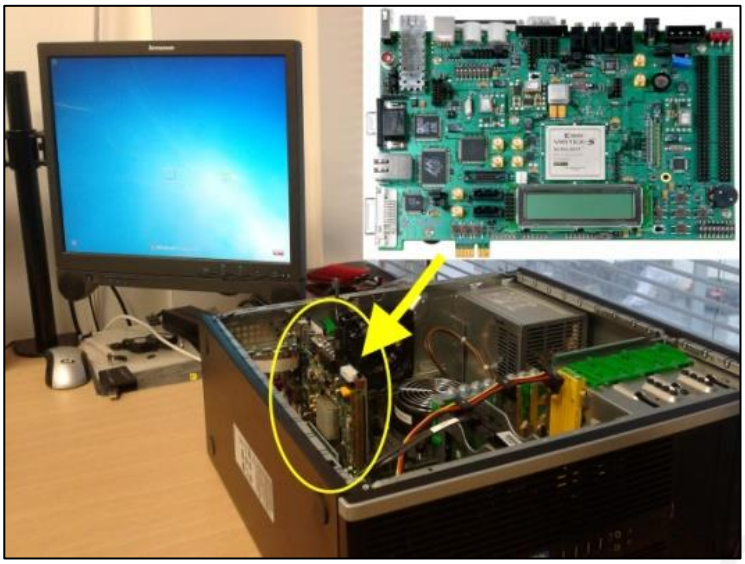


Figure 6.1: An example setup of the proposed architecture. The proposed architecture is running on an FPGA board connected to the host system via PCI-Express.

The proposed hardware architecture is based on custom-built modular components as building blocks. This design approach makes the proposed architecture highly scalable and implementable on other platforms, such as custom ASICs. As such, the system can be easily expanded and upgraded to address emerging computational and algorithmic needs, stemming from both enhanced datasets and algorithmic adjustments/improvements. The proposed architecture is designed to be seamlessly integrated in existing computational systems already used by current end-users. The protein sequences and the results are transferred-to and received-from the proposed architecture by utilizing a standard, high-speed communication channel such as PCI-Express or Gigabit Ethernet. An example setup where the FPGA board configured with the proposed architecture is installed on the PCI-Express bus of a host system is shown in Figure 6.1.

The CAST algorithm can be executed over multiple protein sequences in parallel by using multiple instances of the hardware architecture initially presented in[19]. These *Sequence Processing Units* (SPUs) receive data streams prepared by a specialized *allocation unit* in order to achieve high load balancing and minimized total execution time. The system is receiving the protein sequence dataset from the host PC via a memory-mapped high-speed I/O channel (for example PCI-Express). This I/O scheme is selected because it is natively supported by modern operating systems and provides the end-users (who usually do not have computer engineering background) with a friendly programming environment.

6.1.1 Sequence Processing Unit - SPU

A SPU of the proposed hardware architecture –shown in Figure 6.2 – consists of a series of interconnected processing elements (PEs) that take advantage of the natively parallel characteristics of the CAST algorithm. The SPU receives a stream of protein sequence represented in FASTA format[6], compares the stream with the twenty degenerate sequences, extracts current iteration's LCR, masking it (if appropriate) and iterates until all LCRs are discovered. As soon as all LCRs are discovered, the SPU outputs each of the sequence's LCRs score and position in the sequence.

An SPU consists of three different types of PEs: a front-end unit, a number of CAST processing units and a back-end unit. The input sequence under consideration is streamed into the SPU through an I/O receiver and each unit of the SPU propagates the necessary signals to the next one in a pipelined manner until the propagated stream reaches the back-end unit. The back-end unit then sends it to the I/O transmitter or redirects the updated stream back for the next iteration.

6.1.1.1 Front-end Unit

The front-end unit (FEU) is responsible for receiving one ASCII symbol of the input sequence per cycle from the allocation unit, and to generate the control signals needed by the CAST processing units. FEU is responsible for generating the signals required for communicating with the allocation unit as well. The received symbol is re-coded by the *ASCII decoder unit* to a lesser bit representation in order to reduce the hardware resources needed. The *Control*

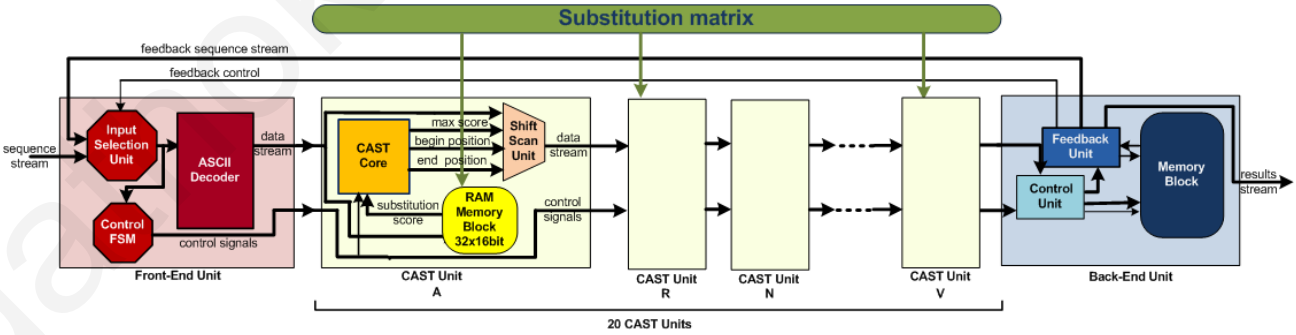


Figure 6.2: Sequence Processing Unit (SPU)

6 FASTA format is using English alphabet letters to represent amino acids in proteins, represented in ASCII text [129].

Finite State Machine (FSM) of the FEU checks each received symbol and if it marks the beginning or the ending of a sequence, the unit generates the appropriate control signals which are propagated to the CAST units. The FEU also facilitates the logic necessary to control the number of iterations needed. Whenever the need for another iteration occurs, the unit signals the allocation unit to hold any new pending sequences and instead re-feeds the updated sequence received from the back-end unit back to the SPU architecture.

6.1.1.2 CAST Unit

The CAST processing units perform the CAST algorithm computations. Each SPU has twenty identical CAST units, each one responsible for executing the CAST algorithm for one of the twenty degenerate homopolymers. They are interconnected in a row-wise pipelined manner. Each unit calculates the *high scoring segment* (HSS) – this region is indicating a potential LCR – of the input sequence for their respective amino acid residue type. A CAST unit consists of a *local memory block* holding the subset of the substitution matrix referring to its respective amino acid residue, a *CAST core* performing the score calculations needed to detect the boundaries of the HSS, and a *shift/scan unit* for propagating the data and control signals needed by the adjacent CAST unit.

The *local memory block* is initialized with the substitution values from a centralized memory block holding the complete matrix prior the arrival of the first input sequence. The substitution matrices used in Bioinformatics are rather small in size (i.e. the BLOSUM62 variant used in this work, requires less than 500 bytes) and can be stored in a centralized multi-ported memory block. However, the number of accesses to the centralized memory block to fetch the substitution values needed in the calculations for all of CAST units during each cycle is a potential bottleneck. Hence, the decision to store the substitution values in small local memory blocks in every CAST unit was made, so that the substitution score data can be accessed asynchronously and as fast as possible. This design approach eliminates the memory access times of the centralized memory, as the data needed is fetched and processed in one cycle.

The *CAST core* – shown in Figure 6.3– derives the score for the sequence streamed in and holds the statistics needed for identifying the HSS and its score. The score accumulates every cycle the substitution scores of each amino acid in the sequence processed so far. If the accumulated score falls below zero, the score is registered as zero and remains zero until the cycle where a positive substitution score arrives. This score value is compared against the

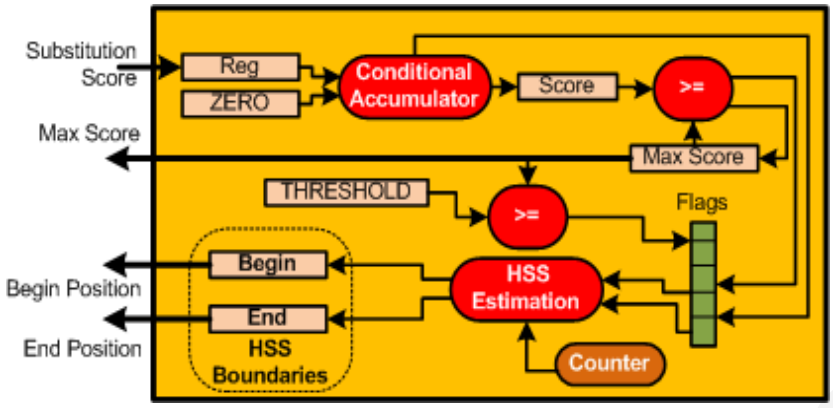


Figure 6.3: CAST core architecture

current max score every cycle as well. The max score register value is increased if needed, and it is also compared against the threshold set by the CAST algorithm. The HSS boundaries can be retrieved just by storing HSS's beginning and ending positions in the sequence. A counter measuring the position of the currently processed symbol in the input sequence is used for that purpose. The *CAST core* has modules which are used for marking the HSS boundaries using a sliding window approach. The end position of the HSS is the position where the current max score has retrieved. When the max score is increased, the position counter's value is used to determine the newly identified HSS's ending. The beginning of the HSS can be determined by evaluating the scoring patterns of the sequence. Each time the current score is set to zero is an indication that the current region of interest (marked as the currently selected HSS) is ending. Thus, the next positive score is showing the beginning of a potential high scoring series of amino acid residues in the sequence and the position of that positive score is stored as a probable beginning of the actual HSS. If this newly discovered region of interest does score higher than the current max score, this stored position is set to indicate the beginning of the currently selected HSS. This process continues to re-evaluate the boundaries of sequence's HSS until all symbols in the sequence are processed. Those boundaries and the max score are fed to the *Shift & Scan* unit.

The *Shift & Scan unit* is responsible for propagating the signals needed by the adjacent CAST units. The actions of all Shift & Scan units are decided by the control signals propagated through their respective CAST Unit. Every Shift & Scan unit operates in two modes; *shift* and *scan*. When in *shift* mode, the unit propagates the stream without changes while the SPU is computing the HSS. When in *scan* mode, the unit propagates the highest value between its CAST Unit's max score and the propagated max score received from the left adjacent CAST

unit. This method allows the *back-end unit* to receive the sequence's maximum HSS score as soon as possible. Whenever the feedback unit informs the corresponding CAST unit that its respective amino acid HSS is actually the current iteration's LCR, the *shift & scan* unit transmits the boundaries of the LCR, first the beginning and then the ending position.

6.1.1.3 Back-end Unit

The back-end unit (BEU) decides whether the analysis of the current sequence is complete and propagates the results to the external I/O controller or updates the sequence and redirects it back to the FEU for calculating the next LCR. This decision is taken by the BEU's control unit and is based on the current iteration having an amino acid residue HSS with higher value than the threshold parameter of the CAST algorithm. The symbols streamed by the last CAST unit while in calculation mode, are stored in a local *Line Memory* of a size equal to the size of the maximum sequence length used in the software implementations of the algorithm. This memory's size can be easily increased to accommodate larger protein sequences. *Line Memory* is implemented as a dual-port memory block. The stream representing the sequence is deemed necessary to be stored locally as the nature of the CAST algorithm requires a number of iterations *dynamically* decided at runtime.

Upon receiving confirmation that all symbols of the sequence are processed, the control unit waits for the HSS with the maximum score to be propagated through the architecture and marks it as current iteration's LCR. It then transmits the necessary signals to the feedback unit to start reading the sequence from the Line Memory and stream it either back to the FEU for another round of calculations, or to the SPU output connected to an external I/O transmitter. The sequence is dynamically updated by the feedback unit as the symbols corresponding to the highest scoring amino-acid are replaced with 'X' while passing through the feedback unit.

6.1.2 Multi-SPU System

The SPU described above is efficiently executing the CAST algorithm over each sequence stream received. The performance of the hardware can be further boosted by integrating multiple SPUs in a single system. The inherit modularity makes the system highly scalable and a system featuring four SPUs is presented in Figure 6.4. The number of SPUs present in each system is limited only by the I/O scheme and the available hardware resources – in the

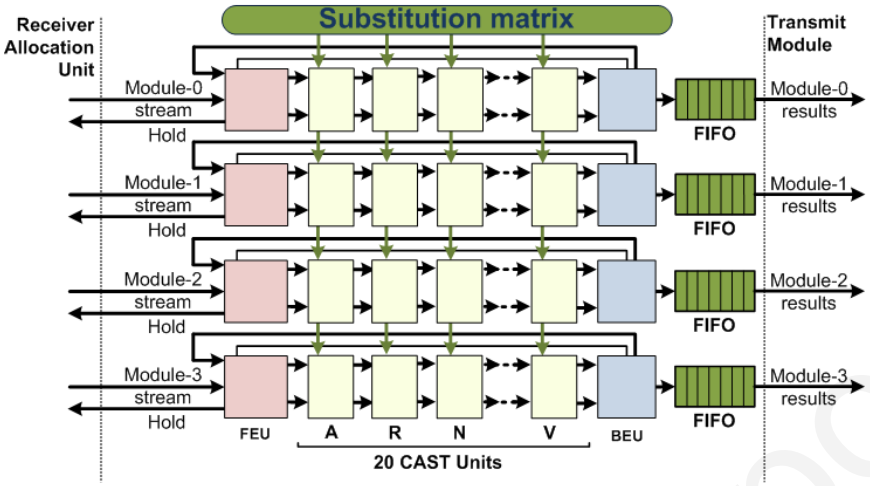


Figure 6.4: Quad-SPU architecture

case of using FPGAs. As such, systems having eight or more SPUs are feasible as well. The SPUs are independent of each other and each one receives a different protein sequence stream. Each SPU is initially loaded with the substitution matrix values, executes the CAST algorithm over the protein sequence stream presented to its input port as described in the previous section, and exports the results in a dedicated FIFO which is used to present the results to the host PC via the high-speed I/O channel. A system of the proposed architecture featuring multiple SPUs needs to have a specialized *allocation unit* to properly partition the received protein dataset, and to prepare each SPU's input stream.

6.1.2.1 Data Allocation Unit

A multi-SPU system integrated with a high-speed I/O communication channel capable of providing multiple symbols per cycle needs an efficient way to partition the received protein sequence data among the SPUs. As such, the host operating system (OS) can be programmed to transmit the protein data in a scheme that is efficient and requires minimum modifications by the proposed architecture. Assuming that the protein sequences are received in a predetermined format through the high-speed I/O communication channel, the system's allocation unit is responsible to divide the sequences to the SPUs. The allocation scheme used, affects the multi-SPU system's execution time for each protein database, as unbalanced load sharing between the SPUs can hinder performance.

The allocation unit can be designed as the one shown in Figure 6.5 where a simple allocation unit, designed using FIFOs, for a system having four SPUs is presented. This allocation unit was designed with emphasis on fitting on the targeted FPGA board. The host OS arbitrarily

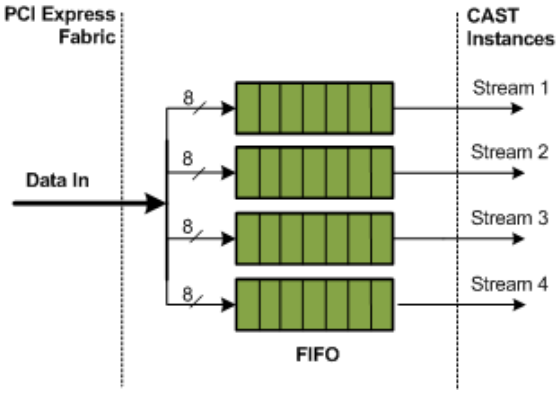


Figure 6.5: A possible allocation unit for four SPUs

splits the protein sequence database in four parts and provides an ASCII symbol taken simultaneously from each part per cycle on the PCI-Express communication channel. As such, four different sequences are fetched in parallel and are stored in input FIFOs driving each SPU. Each FIFO has all necessary control signals like read/write enable, full and empty connected either on the I/O receiver module or the SPU; however those signals are omitted from Figure 6.5 for clarity. The abovementioned allocation scheme is also used in evaluating the proposed hardware architecture.

Higher-complexity resource allocation schemes can take better advantage of the inherent parallelism; however, these schemes can fit only on larger FPGAs. The design of such higher-complexity units is a challenge, as the execution time of a given dataset is greatly affected be the number of iterations needed for each sequence and that is not known beforehand. As such, a static allocation scheme is not efficient, and intuitively an arbitrary partition of the dataset among the SPUs can cause great variations in execution times. The effects of arbitrarily partitioning the data among the SPUs using the resource allocation unit of Figure 6.5, is extensively discussed in the results section below.

6.1.2.2 Overall System Data Flow

The local memory blocks belonging to the CAST units of each SPU are first initialized with their respective substitution matrix subsets. As soon as the initialization is completed, the system is ready to receive data from the high-speed I/O communication channel. The host system begins transmitting the protein sequences to the FPGA, and the allocation unit stores them locally. As soon as an SPU requests a sequence for processing, the allocation unit

allocates an unprocessed sequence to the idle SPU following the selected allocation scheme. The newly allocated sequence is then propagated as input stream to the SPU.

Each stream received by an SPU is processed one ASCII symbol per cycle. The FEU of the SPU recognizes the beginning of the sequence, sets the appropriate control signals and starts propagating the symbols to the adjacent CAST units. When a new sequence is streamed in these CAST units, each unit first resets and waits for the symbols to arrive. Next, each unit searches through its local memory block for the symbol's substitution value, and it then updates the necessary internal registers for the scores and HSS boundaries for each symbol propagated through.

The BEU of the SPU stores the streamed sequence to its Line Memory block and waits for the end of sequence signal. When this happens, the BEU generates the appropriate signals for the CAST units and waits to receive the maximum scoring HSS. The maximum score is propagated through the architecture, and, when it reaches the BEU, is marked as current iteration's LCR. The CAST unit having that LCR then, propagates the region boundaries to BEU. If the current iteration's LCR score is higher than the CAST algorithm's threshold, the BEU updates the sequence stored in its Line Memory and signals the FEU to hold any new incoming sequences. The updated sequence is then fed back to the FEU for the next iteration. However, if the threshold has not been reached, the BEU sends the LCR boundaries and scores to the SPU's output FIFO. In the latter case, the SPU is free to receive the next protein sequence from the allocation unit.

6.1.3 Experimental Platform

A prototype of the proposed hardware architecture was implemented for a Virtex-5 LX110T FPGA running at a 100MHz clock, and evaluated using real protein datasets (see *Appendix A*) . The initial evaluation results of a single SPU design were also presented in [19]. In this work, we evaluate a *complete* multi-SPU system, which includes the I/O module and the host machine. We implemented a PCI-Express communication channel using the *Xilinx LogicCORE EndPoint Block Plus for PCI-Express* IP core [114]. The allocation unit was built using the *Xilinx CoreGenerator* FIFO cores [115], following the scheme presented in Figure 6.5. The complete system was downloaded on a Xilinx University Program XUP-V5-LX110T

Evaluation Platform (cost ~$750) and the board was mounted on a host PC (Intel i3/4GB RAM), as shown in Figure 6.1.

Every SPU has the local memory blocks of its corresponding CAST units mapped as distributed RAM on the FPGA fabric for more flexibility and speed. The line memory of each BEU is implemented as a standard dual-port memory and is mapped on the BRAM resources of the FPGA. The FPGA system was integrated to the host PC through the PCI-Express channel interface (see Figure 6.1). Obviously, the use of the external communication channel adds significant overheads; however, we strongly believe that giving results for a complete multi-SPU system strengthens the actual impact of the proposed architecture and the presented results. It is of course implied that implementation of custom-built (but not necessarily standard) communication modules, optimized for the proposed CAST hardware architecture, will minimize the impact of communication overheads on the execution time, but the design of such systems is left as future work.

The original CAST implementation presented in [32] provides an initial comparison reference; however, comparing a highly parallel hardware architecture with a sequential single-threaded software implementation can be considered unfair. Thus, we compared the proposed hardware architecture against the optimized *mCAST2.0* software implementation using the datasets listed in *Appendix A: Proteomic Benchmarks* [19]. Moreover, we run the SEG algorithm on the *Intel Core i7* system for all test datasets. Comparing the results of the proposed architecture against SEG algorithm is essential, as SEG algorithm is a common option for LCR masking used by the BLAST suite.

6.1.4 Hardware Evaluation

We evaluated the proposed hardware architecture designed for executing the CAST algorithm by using three different system builds: a single-SPU architecture, a Quad-SPU and finally an Octal-SPU system. All evaluation configurations have a complete 64bit/per cycle PCI-Express I/O communication channel. We evaluated the octal-SPU system – which consumed almost all available resources on the FPGA evaluation platform – in order to observe the scalability of the proposed architecture.

We compare the results of the proposed hardware architecture configurations against the software *mCAST2.0* implementation and *SEG* running on the *Intel Core i7* system. The execution times for both *mCAST2.0* and SEG have been calculated while suppressing I/O system calls for fair comparison with the proposed hardware architecture; the software under these conditions focuses on reading inputs and computing the algorithm, similarly with the hardware approach.

Table 3 shows the comparisons between the execution time achieved by *mCAST2.0* and SEG, against the execution time of the three evaluation configurations of the proposed architecture. The performance results of all hardware configurations outperform both the optimized *mCAST2.0* and SEG despite running at a lower clock of 100MHz. In fact, results are orders of magnitude better for all test sequences. The merits of designing a hardware architecture for accelerating the masking of LCRs in protein sequences are made clear when comparing the execution time results for the *Haem*, *p.falciparum*, *Viral.1.Protein* and *Uniprot.sprot* databases. These cases are actual protein datasets which may be used for benchmarking bioinformatics applications. The single-SPU configuration yields speedups higher than 30x over SEG and over two orders of magnitude over *mCAST2.0*. The quad-SPU architecture

TABLE 3: SINGLE-SPU VS QUAD-SPU VS OCTAL-SPU SYSTEM

	Single SPU H/W (ms)	Quad SPU H/W (ms)	Octal SPU H/W (ms)	X^a	X^b	mCAST2.0 (ms)	X^c	X^d	X^e	SEG (ms)	X^f	X^g	X^h
Case.I	0.07	0.07	0.07	1	1	9	129	129	129	1	14	14	14
Case.II	0.93	0.26	0.13	3.58	7.15	239	257	919	1838	58	62	223	446
Case.III	1.85	0.50	0.26	3.70	7.11	472	255	944	1815	30	16	60	115
Case.IV	0.94	0.24	0.15	3.92	6.26	290	309	1208	1933	29	30	121	193
Case.V	1.28	0.40	0.23	3.20	5.56	229	179	573	996	33	26	83	143
Haem	6.74	1.68	0.90	4.01	7.49	1721	255	1024	1912	209	31	124	232
p.falciparum	162	42	22	3.86	7.36	15031	93	358	683	4357	27	104	198
Viral.1.Protein	452	113	58	4.00	7.79	110062	243	974	1898	13105	29	116	226
Uniprot.Sprot	3043	768	388	3.96	7.84	2054892	675	2675	5296	59363	20	77	153

a. Speedup of Quad-SPU H/W over Single-SPU H/W architecture
b. Speedup of Octal-SPU over Single-SPU H/W architecture
c. Speedup of Single-SPU H/W over mCAST 2.0
d. Speedup of Quad-SPU H/W architecture over mCAST 2.0
e. Speedup of Octal-SPU H/W architecture over mCAST 2.0
f. Speedup of Single-SPU H/W architecture over SEG
g. Speedup of Quad-SPU H/W architecture over SEG
h. Speedup of Octal-SPU H/W architecture over SEG

outperforms SEG with two orders of magnitude speedup and *mCAST2.0* is outperformed by several thousand times for most test cases. The octal-SPU configuration has shown astonishing results as well, proving that the proposed architecture is indeed highly scalable as this configuration does halve the execution times when compared to the quad-SPU configuration.

Results on the *p.falciparum* database, which has a large number of compositionally biased regions, demonstrate the capabilities of the hardware approach, as the proposed quad-SPU hardware has a 350x speedup over *mCAST2.0* and 104x speedup over SEG. *Results of the quad- and octal-SPUs configurations on the over-half-a-million-sequences long Uniprot.sprot database, show over 2500x and over 5000x speedups over the mCAST 2.0 implementation, respectively.* These astonishing results prove that the proposed system for identifying and masking LCRs is capable of handling efficiently the massive protein datasets nowadays analyzed in several biology labs worldwide.

Table 3 indicates the limitations of LCR masking on general-purpose high-end processors, as they cannot efficiently execute algorithms used in bioinformatics due to memory bandwidth issues and limitations in available levels of parallelism, despite the coding improvements made to the software implementations. On the other hand, a custom-built hardware system running even on a cheap FPGA board can significantly accelerate performance. Moreover, a future ASIC implementation of the proposed architecture will exhibit even higher performance gains.

6.1.4.1 Allocation Unit - Observations and Discussion

The allocation scheme for partitioning the received sequences to the SPUs is a design aspect that needs to be evaluated as well. A poor allocation scheme will cause significant variation between the SPUs execution times. Unbalanced input streams will cause some of the SPUs to finish their load early while other SPUs will be struggling under heavier loads; this hinders the overall execution time.

Table 4 provides the execution times of every SPU when the proposed architecture configurations follow the allocation scheme of Figure 6.5. We also calculated the utilization percentage for each SPU and the average utilization percentage relative to the longest running SPU for each configuration.

The simple allocation unit shown in Figure 6.5 is proven adequate to justify the proposed system, despite the fact that the arbitrary allocation of the sequences caused unbalanced utilization of the CAST instances and negatively affected performance results, as shown in Table 4. For instance, the load of the quad-SPU configuration is greatly unbalanced while it was executing the *haem* dataset. The arbitrary partition of the dataset by the OS of the host PC has placed heavy load on SPU-0 and SPU-1 while the rest SPUs are underutilized (utilization 13% and 14% for SPU-1 and SPU-2 respectively). The overall execution time for the dataset is calculated as 1.68ms (time needed for all SPUs to finish their load) while a properly balanced allocation should have yielded execution time below 0.9ms.

The simple allocation scheme used in the evaluation of the proposed hardware architecture offers more balanced SPU utilization as we increase the number of SPUs in the system. Table 4 shows that the octal-SPU configuration is well balanced in most cases. Splitting each dataset in a larger number of partitions statistically increases the probability to achieve even load

TABLE 4: INDIVIDUAL SPU PERFORMANCE AND UTILIZATION PERCENTAGE FOR EACH CONFIGURATION

	Quad-SPU Architecture						Octal-SPU Architecture									
	SPU 0 (ms)	SPU 1 (ms)	SPU 2 (ms)	SPU 3 (ms)	Total (ms)	Average Utilization	SPU 0 (ms)	SPU 1 (ms)	SPU 2 (ms)	SPU 3 (ms)	SPU 4 (ms)	SPU 5 (ms)	SPU 6 (ms)	SPU 7 (ms)	Total (ms)	Average Utilization
Case.II	0.26	0.22	0.23	0.24	0.26	91.25%	0.13	0.12	0.14	0.13	0.13	0.10	0.09	0.10	0.14	83.88%
	100%	85%	88%	92%			93%	86%	100%	93%	93%	71%	64%	71%		
Case.III	0.50	0.41	0.44	0.50	0.50	92.5%	0.26	0.23	0.25	0.25	0.24	0.19	0.19	0.26	0.26	89.75%
	100%	82%	88%	100%			100%	88%	96%	96%	92%	73%	73%	100%		
Case.IV	0.24	0.23	0.24	0.24	0.24	99%	0.13	0.12	0.10	0.12	0.12	0.11	0.15	0.12	0.15	80.88%
	100%	96%	100%	100%			87%	80%	67%	80%	80%	73%	100%	80%		
Case.V	0.40	0.32	0.26	0.30	0.40	80%	0.17	0.16	0.13	0.12	0.23	0.16	0.13	0.17	0.23	69%
	100%	80%	65%	75%			74%	69%	57%	52%	100%	69%	57%	74%		
Haem	1.68	1.28	0.22	0.24	1.68	50.75%	0.86	0.90	0.88	0.84	0.84	0.83	0.80	0.90	0.90	95.13%
	100%	76%	13%	14%			96%	100%	98%	93%	93%	92%	89%	100%		
p.falciparum	39.91	41.58	38.56	0.14	41.58	72.3%	19.10	20.38	18.67	21.99	20.87	21.26	19.94	20.98	21.99	93%
	96%	100%	93%	0.3%			87%	93%	85%	100%	95%	97%	91%	96%		
Viral.1.Protein	112.49	113.08	113.43	112.84	113.43	99.25%	57.69	56.11	56.94	56.95	55.81	57.98	57.50	56.91	57.98	98.25%
	99%	99%	100%	99%			99%	97%	98%	98%	97%	100%	99%	98%		
Uniprot.Sprot	767.63	766.01	762.18	763.88	763.63	99.63%	385.1	385.0	384.5	383.3	387.6	386.4	383.2	385.9	387.6	99.38%
	100%	99.8%	99.2%	99.5%			99.4%	99.3%	99.2%	98.9%	100%	99.7%	98.9%	99.6%		

among the SPUs. It worth mentioning this observation is reversed for the small datasets (*Case.I* to *Case.V*). Table 4 shows that the octal-SPU configuration has worse load balancing than the quad-SPU configuration for small datasets. Probability to uniformly partition a small dataset in equal chucks is smaller than uniformly partition large datasets. However, if we take into account that most real world scenarios use large datasets, a system featuring eight or more SPUs has no need for a complex allocation unit.

6.1.5 Power Consumption

The average power consumption was estimated using the Xilinx XPower tool over the proposed architecture's post-route simulation model. The results show that an 4-SPUs system consumes less than 1.8W of dynamic power on average. Taking the minimized execution time into account as well, the proposed system's energy consumption gains rate well in the range of the other FPGA-based architectures studied in [14] where FPGAs are proven to use orders of magnitude less energy than their GPU- and CPU-based counterparts.

6.1.6 Hardware Synthesis Results for Xilinx Virtex-5 FPGA

We lastly give the hardware synthesis results in Table 5, in order to give a complete picture of the required hardware resources needed for mapping the proposed architecture on a relatively cheap, off-the-shelf FPGA.

We provide the FPGA resources needed to map each SPU, the simple allocation unit of Figure 6.5 and the PCI-Express communication modules while following the memory mapping considerations discussed in the results section above. The synthesis results for the complete quad-SPU and octal-SPU systems are also provided.

The proposed architecture accelerates over 100x the execution time of the improved multi-threaded CAST algorithm using just 10% of the LX110T FPGA for implementing a single-SPU system. Such overheads are small enough to allow other FPGA-based bioinformatics algorithms such as BLAST to be combined with CAST on the same FPGA, further enhancing the performance of such systems. The quad-SPU system yields average speedups of over 500x

TABLE 5: SYNTHESIS RESULTS FOR THE XILINX VIRTEX-5 LX110T FPGA

FPGA Resources	Used / Available	Usage Percentage
Sequence Processing Unit (SPU)		
Slice Registers	4542 out of 69120	6%
Number of Slice LUTs	8259 out of 69120	11%
Block Rams	6 out of 148	4%
DSP48Es	0 out of 64	0%
Simple Allocation Unit for Quad-SPU System		
Slice Registers	695 out of 69120	1%
Number of Slice LUTs	417 out of 69120	1%
Block Rams	2 out of 148	1%
DSP48Es	0 out of 64	0%
PCI-Express Communication Module		
Slice Registers	2768 out of 69120	4%
Number of Slice LUTs	4105 out of 69120	5%
Block Rams	6 out of 148	4%
PCIE Lanes	1 out of 1	100%
I/O Bandwidth	64bits/cycle	
Quad-SPU Architecture		
Slice Registers	21647 out of 69120	31%
Number of Slice LUTs	37603 out of 69120	54%
Block Rams	14 out of 148	9%
Frequency	114MHz	
Octal-SPU Architecture		
Slice Registers	43298 out of 69120	63%
Number of Slice LUTs	69114 out of 69120	100%
Block Rams	28 out of 148	19%
Frequency	100MHz	

and occupies around 50% of the FPGA used in the evaluation process, which leaves enough resources for mapping other bioinformatics software tools on the FPGA fabric as well.

The PCI-Express communication module integrated in the proposed architecture allows the utilization of multiple FPGA boards on the same host PC, each running either an instance of a multi-SPU architecture or other FPGA-based bioinformatics applications like BLAST.

6.1.7 Scalability Test for the Proposed Architecture

The capabilities of the highly scalable FPGA-based CAST architecture are also limited only by the amount of reconfigurable fabric present on the FPGA used, and the speed of the I/O communication channel that propagates the protein sequence streams from the host PC into the FPGA fabric.

To offset these limitations, multiple instances of the CAST hardware architecture where loaded on the BeeCube Hardware Emulation platform [116] as a case study. BeeCube is a full-

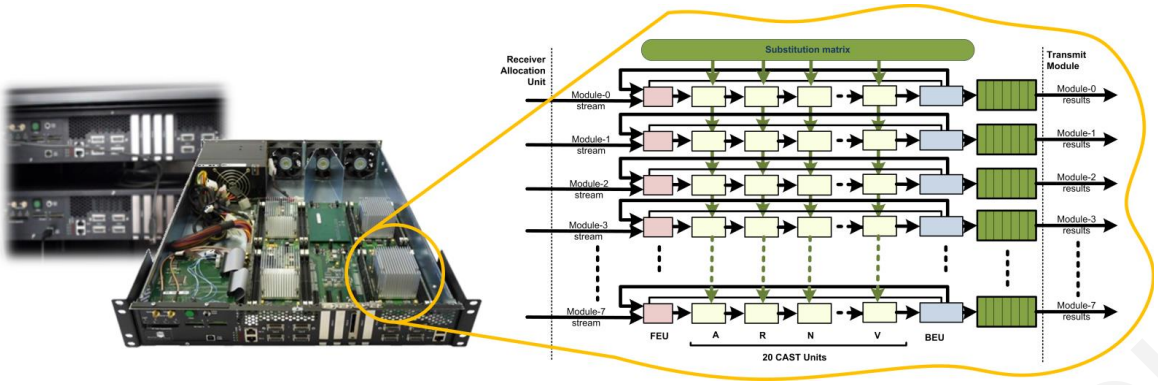


Figure 6.6: CAST Execution on the BeeCube Hardware Emulation Platform

speed FPGA prototyping platform which facilitates 4 Xilinx-5 FPGAs and can achieve more than 4000GOPS or emulate more than 64 RISC processor cores concurrently at 100MHz. A single BeeCube FPGA module can accommodate up to eight instances of CAST when adding the necessary I/O logic needed (FIFO buffers) to propagate the protein sequence streams to each CAST instance.

A case study for using BeeCube in in high-performance bioinformatics applications has been presented in [17]. Eight instances of the CAST hardware architecture were loaded on each FPGA. The protein sequences were loaded using a Compact Flash memory, and transferred to the BeeCube memory structure through an integrated MicroBlaze soft processor. Initial results show a near-linear speedup over the standalone FPGA-based implementation [18][20].

We obtained a near-linear speedup on the BeeCube implementation over the single chip implementation demonstrated in [20] and prove that higher performance is indeed achievable when the design is scalable [17]. However, a full linear speed-up was not reached, as a more optimized protein sequence partitioning scheme was not considered in this initial study. Arbitrarily partitioning the sequences into streams to drive each instance fails when applied to the CAST algorithm, in which the amount of iterations for each sequence is dynamically determined at runtime.

Given that the algorithm's performance relies heavily on the way that the input sequences are fed to the system, an optimal resource allocation algorithm could jointly be developed amongst biologists and engineers that will enable linear speedups. This further emphasizes the potential for research and development in the particular field as such large-scale systems can indeed be used as computationally efficient high-end platforms for genomic data processing.

6.2 **FPGA-based Prefix Doubling Architecture**

In this section, we will present our first attempt to use FPGAs for implementing prefix doubling suffix array construction for de novo sequence assembly applications. FPGAs offer reconfigurable fabric which we can use to create computation modules with *arbitrary* I/O bit length. Prefix doubling - as already discussed in *Section 2.3.6* - can benefit of this feature of FPGAs since a hardware architecture for this application is not restricted by the maximum 64bit variable size for CPU- and GPU-based software implementations. The critical code segment of Figure 2.9 is:

While $l > n$ or [other *terminal conditions such as: unique index for all* $N_i^{l/2}$]
  Sort T_i^l using N_i^l to hold the index
  Use the pairs (N_i^l , N_{i+l}^l) to represent $T_i^{2l} = T_i^l T_{i+l}^l$
End

It is already mentioned that despite this code segment having low computational complexity, the difficulty lays with the extraction of the new sorting indexes needed for the array *N*. Calculating the index of DNA nucleotide bases (Σ={A,C,G,T,$}) needs to sort $|\Sigma|^l = 5^l$ indexes which doubles for every iteration. We can see that eventually we are reaching to a massive number of *preserved name* indexes to sort. These indexes are then needed to be assigned to all of the *n* generated prefixes of every iteration. The proposed architecture is attempting to offer fast computation of the (N_i^l , N_{i+l}^l) pairs for multiple iterations in a single run.

6.2.1 Systolic Pair Calculator

The main building block of the proposed systolic architecture (SPR) is the *N-core.* Each instance of *N-core* is responsible for creating the preserved name for its assigned iteration for the input sequence reads. We are using basic bitwise operations to create the preserved name in each *N-core* unit using the following intuitive formula:

Given the initial nucleotide base alphabet Σ={$,A,C,G,T}, we first assign *l*=1 indexes using a look-up table. The data then is propagated through the architecture into the *2l*-assigned *N-cores*. Each N-core is responsible for creating the preserved name for the *doubled prefix* by *doubling* the size of its output by concatenate its two inputs *together*.

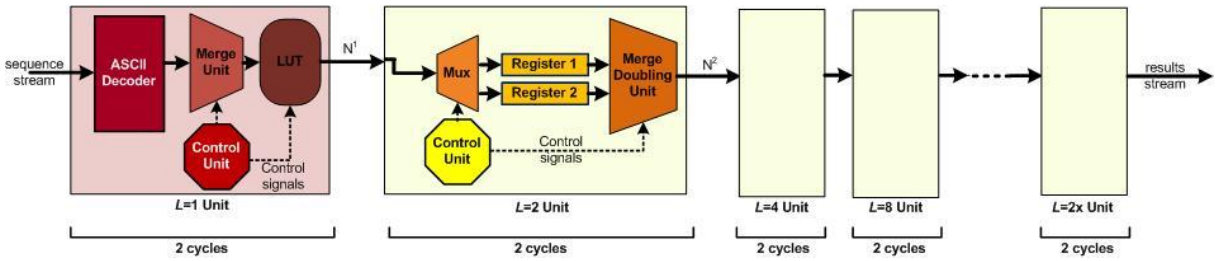


Figure 6.7: Systolic (N_i^l , N_{i+1}^l) pair Calculator

This is an intuitive solution that can be explained using arithmetics: Assuming that we have the preserved name's indexes *x* and *y* for *l=i* iteration, the prefix doubling pair name for the next iteration - i.e for *l=2i* - will be *(x,y)*. This pair must be then sorted in order to extract its index for the iterations to come. Since we use integers to express these indexes, sorting is normally done by integer sorting algorithms such as radix sort and its variants.

We are solving the preserved name indexing problem using concatenation: Assuming *x* and *y* are expressed in base *r*, we concatenate these numbers using the same numeric base *r* and the result is the new number *xy* expressed as an integer in base *r*: $xy = x \cdot r^1 + y \cdot r^0$.

Suppose we have the preserved names *(x,y)* and *(w,z)* where *x>w* and {x,y,w,z} are considered to be integers in base *r* . Using the above concatenating method, the prefix doubling indexes expressed as *xy* and *wz*, always respect *xy>wz*. As such, the concatenated doubled values shall maintain the *same* sorting order as we could get by following the original procedure: we sort the pairs first, then execute prefix doubling and then sort them again.

As we can see, we can eliminate two processing steps with one concatenation. Using concatenations in succession, we can reduce the operations needed to perform preserved naming prefix doubling *exponentially*. This is the idea behind the proposed SPC architecture shown in Figure 6.7.

This idea is only applicable for FPGA-based architectures since the required variable length will exceed the available 64bits on CPUs and GPUs in the end of just two iterations. We will now describe the proposed architecture and discuss each unit in Figure 6.7 in detail.

6.2.1.1 L=1 Unit

The sequence reads are propagated into the architecture as ASCII characters {A,C,G,T} through the *L=1 Unit* which is responsible for decoding the received nucleotide bases to the binary numbers of 3 bit width representing them. The unit holds every data for two cycles each in order to generate all necessary prefix doubling data for the matrix N^1. The data is then merged together and encoded though an LUT before the generated data propagates deeper in the SPC architecture.

6.2.1.2 L Unit

The doubled prefix data continuous to propagate deeper into the architecture until we reach a preselected *L unit* limit. Each instance of this unit works in the same manner; however each unit's output bit width is double its input bit width. This means that *L units* connected deeper into the pipeline have to deal with larger and larger bit widths. For example, *L=4 unit* has input ports of 12bits and output ports of 24bits.

Each instance of *L unit* holds each input data for two cycles before its merge/doubling unit produces the doubled output data. The two cycle latency is hidden as all of the SPC architecture's units start to fill with data and after a number of cycles, a new prefix doubled data for *L=max(2 x # of L units)* is generated every cycle.

The number of *L units* present in the architecture is greatly affected by the available reconfigurable fabric and the number of I/O pins available on the FPGA selected for the architecture to be mapped on. The generated paired preserved names data is then streamed outside the architecture and into a CPU for sorting using *insertion sort*.

6.2.1.3 CPU-based Insertion Sort

The SPC architecture generates data that even with the concatenation technique eventually needs to be sorted. We can design and implement an FPGA-based sorter which can be connected as the last stage of the proposed architecture, however the length of the input DNA sequence is the prohibitive factor.

The suffix array used to create the necessary graph used in de novo sequence assembly applications is the suffix array of all the reads concatenated together in a massive string. This string's length can be in the order of hundreds of thousands. Unfortunately, available FPGAs do not have enough memory resources for this array to be stored. We remind that the suffix

```
<L> is the List holding the already sorted data
J is the new received data

i = 0
While  i < L.size or J < L(i)
        i = i + 1
End
If  i < L.size
        Add J to the end of the list
else
        Add J between  L(i) and L(i + 1)
End
```

Figure 6.8: Insertion Sort for prefix doubling sorting

array is of the same length as the input string. As such, we decided to use a CPU to sort the SPC architecture's generated data.

Insertion sort despite being slow compared to other sorting algorithms - such as heap sort, merge sort and quick sort - is suitable for our demonstration since it is implemented with just a handful of commands and *it can sort a list as it receives it*. This is ideal for our streaming application. New data is generated from a systolic architecture such as SPR every cycle in a single output port. CPU is working at orders of magnitude higher frequencies, thus - on average - insertion sort will be able to add the received data in a sorted list before the next data is available. The pseudocode for insertion sort used in the proposed system is shown in Figure 6.8.

6.2.1.4 Overall Data Flow

The SPC architecture and the accompanied CPU are receiving the massive concatenated string of the DNA sequence reads which is stored on the host systems main memory. First, the data starts to be transferred one ASCII symbol at a time using the preselected data transfer scheme such as PCI-Express. The SPC architecture starts generating one *x-bit* entry belonging to the N^L array where *L=max(2 x # of L units)* per cycle as soon as the initialization latency is over.

The data output of the SPC architecture is then transferred back to the host system through the data transfer link, one entry at each cycle. The difference of clock frequency between the FPGA and the host system's CPU will allow for the entry to be transferred and added to the sorted list using insertion sort by the CPU, before the next entry arrives from the FPGA.

The sorted list now holds the N^L array preserved name indexes which can be used in two different ways. In case where the suffix array terminal conditions of Figure 2.9 are met, the application reports the indexes are the suffix array of the input sequence. Otherwise, the system simply redirects the indexes into the SPC architecture and the process iterates until the criteria are met.

6.2.2 Performance Evaluation & Results

We evaluated the proposed architecture using the ASCLEPIUS hybrid platform we have built in our laboratory at the University of Cyprus. The selected host system runs on an *Intel i7 3960 Xtreme processor@32GB RAM* and has a *Virtex6 FPGA ML605 Evaluation board* attached through PCI-Express.

Since the proposed SPC architecture serves as a proof-of-concept implementation, we used a series of randomly generated sequences of preselected length sizes (*N=1.000, 10.000, 100.000*) using the DNA nucleotide base ASCII characters {A, C, G, T, $} as the alphabet.

The random sequences were loaded on the host system's main memory and their data was streamed into the FPGA through the PCI-Express interface application we have developed. The FPGA-based SPC architecture has configured to have six *L units*. This means that we can calculate preserved name indexes of 7 iterations on a single pass.

The evaluation results shown in Table 6 state that the SPC architecture actual execution times are less than 1% of the overall execution time for each configuration. This makes clear the fact that the actual computing time is consumed by the PCI-Express link and the insertion sort

TABLE 6: SPC ARCHITECTURE - PERFORMANCE RESULTS

FPGA Resources	Execution Time	Percentage
N=1.000		
Total Execution Time:	16s	
SPC Architecture	<0.01s (2 passes)	<0.1%
N=10.000		
Total Execution Time:	193s	
SPC Architecture	0.79s (2 passes)	<0.1%
N=100.000		
Total Execution Time:	1841s	
SPC Architecture	12.01s (3 passes)	<0.1%
Frequency	200MHz	

software running on the CPU. A more detailed profile of the CPU-based part of the application revealed that the bottleneck is actual the PCI-Express link. Data remains in the PCI-Express transmit/receive FIFOs more than it was anticipated during the design phase of this proof-of-concept application. Reasons behind this behaviour can be traced to the way the host operating system uses the FPGA PCI-Express driver we developed. Moreover, the data transfer was implemented to be extremely fine-grained in order to allow for the streaming data the FPGA-based SPC architecture needs.

Future implementations need to utilize DMA data transfers through the PCI-Express link which allow for reaching the theoretical bandwidth throughput limits by directly transmitting bulk of data from the main memory to the FPGA without using CPU resources.

6.2.3 Synthesis Results

The synthesis results for the 7-level SPC architecture are presented in Table 7. The SPC architecture uses just 1% of the available FPGA resources. However, we cannot add more levels - i.e. additional *L units* - since we have reached the I/O limits of the device. The SPC architecture connects to the outside world through PCI-Express modules which can only transfer 64bits/cycle on each PCI-Express lane.

The proposed hardware architecture can be clocked up to 859MHz, four times higher than the ML605 board clock frequency. However, despite being an ultra-fast solution for prefix doubling, the FPGA remains extremely underutilized (it consumes 1% of the resources). Scaling the architecture to accommodate more than a single index at a time is not an option because of the FPGA board's available I/O bandwidth limitations.

TABLE 7: SYNTHESIS RESULTS FOR THE XILINX VIRTEX-6 LX240T FPGA ON THE ML605 BOARD

FPGA Resources	Used / Available	Usage Percentage
Systolic (N_i^l , N_{i+l}^l) pair Calculator (SPC)		
Slice Registers	964 out of 301400	1%
Number of Slice LUTs	970 out of 150720	1%
PCI-Express Lanes	2 out of 2	100%
Block Rams	0 out of 416	0%
DSP48Es	0 out of 768	0%
I/O Bandwidth	128bits/cycle in each direction	
Frequency	200MHz	

6.3 **Closing Remarks**

In this chapter, we presented our FPGA-based hardware implementation for the low-complexity region masking CAST algorithm and a proof-of-concept suffix array construction method for de novo DNA sequence assembly.

A single instance of the FPGA-based CAST implementation has speedups of over 100x times when compared to the multi-threaded software implementation mCAST2.0. Moreover, the design is proven to be scalable by achieving near-linear speedups when multiple instances where loaded on the BeeCube multi-FPGA platform. The octal-SPU CAST offers more than three orders of magnitude performance improvement over mCAST2.0 while yielding identical results with the original software.

On the other hand, the SPC architecture for prefix doubling suffix array construction is proven to be non-trivial. The nature of the application forbid us to fully use the available resources on the FPGA due to either lack of large enough memory on the FPGA or either to the I/O limitations of the PCI-Express bus used to transfer data between the SPC architecture and the host system's CPU where the last part of the computation was executed.

While the FPGA part of the application had adequate performance, the data transfer overheads restricted the overall performance gains. These results may indicate that the selected prefix doubling algorithm may not be a good candidate for an FPGA implementation, unless I/O issues are handled by developing a custom I/O solution, rather than relying on PCI-Express.

To conclude, bioinformatics and biomedical applications can benefit from the use of FPGAs in cases where the application can be mapped efficiently on the available resources. There is no available methodology to determine the performance of an FPGA architecture early enough in the design phase and this can lead to misuse of developing time and effort.

The inability of knowing the performance results before developing the complete FPGA-based architecture and the fact that there are applications needed to be accelerated on strict deadlines and still yield performance gains, lead to the use of software implementations for exploiting parallelism either on multicore CPUs or GPUs. However, in most cases we have studied, FPGAs did offer performance acceleration in the applications or processing steps mapped on them.

Combining the above observations together, lead us to believe that there is a need for hybrid computing systems consisting of CPUs, GPUs and FPGAs. FPGAs should be an integral part of such systems because of their impressive acceleration results for most bioinformatics and biomedical applications. GPUs are vital for such systems as well, since they offer faster developing times and our results show that the GPGPU programming paradigm allows for higher performance. Such a hybrid CPU/GPU/FPGA system which has been built in the University of Cyprus, is explored in the next chapter of this thesis.

7

Towards a Hybrid CPU/GPU/FPGA Platform for Accelerating Bioinformatics and Biomedical Applications

The major drawback of utilizing a hybrid CPU/GPU/FPGA computing platform is the lack of appropriate tools for producing optimized code for each of the available resource type from a single framework or a toolchain.

Hybrid heterogeneous systems despite their advantage of combining all of available processing engines, are not deemed as practical computing platforms due to the fact that they require complex designs and advanced programming methods in order to be utilized [117].

In this chapter, we first discuss the available tools for hybrid platforms, and then present the communication modules and the hybrid application programming model we developed for the hybrid CPU/GPU/FPGA heterogeneous computing platform we have built in the University of Cyprus. The results yielded by loading the first to be reported CPU/GPU/FPGA implementation on our custom-built hybrid platform are discussed as well.

7.1 Communication Protocols

Firstly, all available state-of-the-art inter-module communication schemes had to be studied and evaluated in order to select the most applicable one for being used as the base communication protocol for our computation platform. The evaluation results are presented in *Appendix B: Communication Schemes Evaluation*.

PCI-Express has been selected as the base communication scheme between the FPGA boards, the GPU cards and the main multicore CPU, as it allows for fast and robust data transfers between the heterogeneous building blocks of the platform. Also, it is a widely accepted communication standard and as such there exist off-the-shelve products that can allow a cheap integration of this project's prototype. A top-level diagram of the proposed PCI-Express-based platform is presented in Figure 7.1.GPU cards are connected to the main processor via PCI-Express by default. Also, the nVidia's CUDA framework is providing functions that can be used for transferring and receiving data from each GPU. As such, we decided to code and compile our user interface using the CUDA compiler which allows for inherit use of both the data transfer functions and kernel execute commands.

The FPGA boards however, require the development of new custom PCI-Express hardware modules which will be running on the reconfigurable fabric alongside with the custom-built accelerators for the bioinformatics and biomedical applications. Also, new specialized drivers that will allow the FPGA devices to be access and configured through the operating system are required to be developed as well.

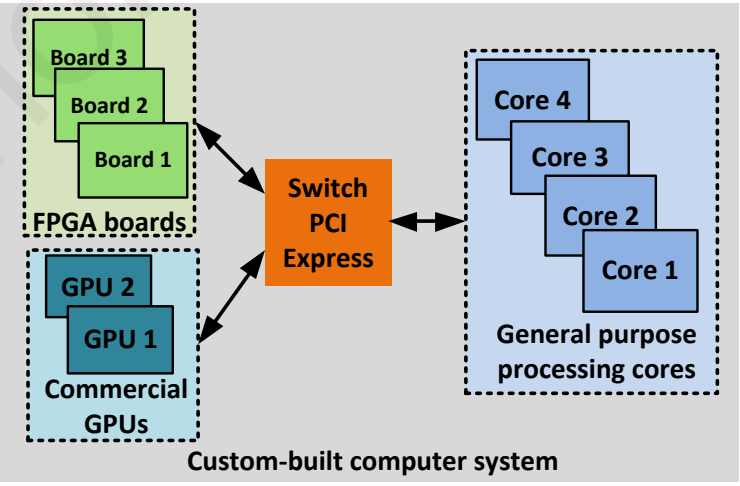


Figure 7.1: Proposed Hybrid Platform Diagram

7.1.1 PCI-Express Communication Modules for FPGAs

A first version of the PCI-Express communication module has been used in transferring the data from the main memory of the host computer to the attached FPGA-board during the performance evaluation of the CAST algorithm accelerator presented in [21]. This simple allocation unit is shown in Figure 7.2.

In this implementation, the host OS arbitrarily splits the protein sequence database in four parts and provides an ASCII symbol - which describes an amino acid of the protein - taken simultaneously from each part per cycle on a 32-bit PCI-Express communication channel. As such, four different sequences are fetched in parallel and are stored in input FIFOs driving each instance of the accelerator. The evaluation results showed that specialized allocation schemes must be used in order to fully utilize the FPGA resources.

Our attempt to design an allocation unit that allows better partitioning of the protein sequences among the FPGA resources is shown in Figure 7.3. Each protein sequence in the dataset is stored as a linked list in the main memory and the operating system transmits the next symbol address over the PCI-Express channel along with the current symbol and the current write address. The allocation unit receives the data and stores it in a multi-ported memory block. Each memory address of the memory block can store an ASCII symbol and the address of the next symbol just like a linked list structure implemented in a high-level programming language.

The memory block has as many read ports as the number of accelerator instances present in the system and one write port which is connected to the PCI-Express communication channel. The allocation unit uses the current symbol's next address tag to propagate the next symbol to

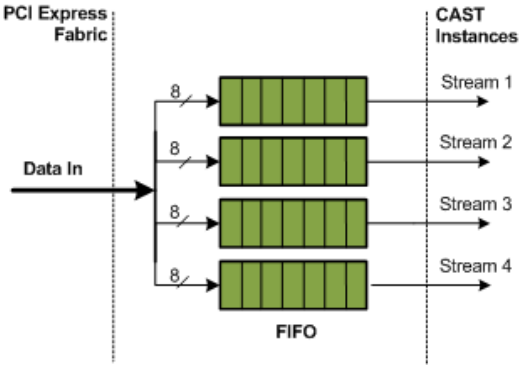


Figure 7.2: Simple Allocation Unit

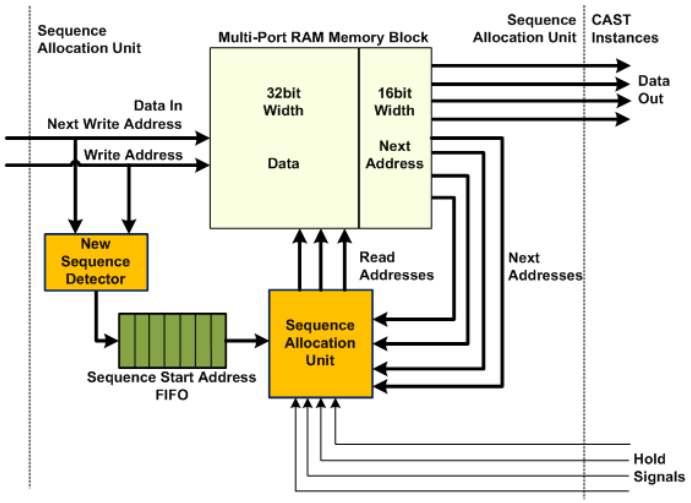


Figure 7.3: Optimal Allocation Unit

the corresponding read port for each instance upon request. In the case where an instance finished processing the current protein sequence, the allocation unit extracts the first ASCII symbol's memory address of a new unprocessed sequence from the *start address FIFO*. This proposed allocation unit is allowing optimal load balancing among the instances as it allocates the protein sequences *dynamically* to the idle instances.

The implemented PCI-Express communication backbone was been evaluated on our custom-built ASCLEPIUS hybrid platform (introduced in *Chapter 3.2.4*) while transferring data for a GPU-based and a FPGA-based CAST implementation from Windows 7 operating system. The GPU application showed that we can fully utilize the resources of a GTX690 GPU card using the CUDA PCI-Express transfer functions from the developed user interface. The FPGA application using the initial simple PCI-Express module showed promising results. However, the optimal PCI-Express allocation unit cannot efficiently mapped on the Xilinx Virtex-5 FPGA used in our evaluation, as the on-board BRAM memory resources can only facilitate up to two ports each. As such, the optimal utilization of the FPGA resources of our hybrid platform is still an open research topic.

7.2 **Tooling and Programming Methodologies**

Ideally, a hybrid application developer would exhaustively try all possible configurations in order to select the optimal one that meets all of the application's requirement on performance, power and resource utilization. However, this approach is not feasible in most of the real world's problems as each configuration leads to different tasks to be mapped on different components. Thus, the developers will need to spend excessive amounts of time just to develop different software code or hardware modules for exploring each possible configuration.

Since the exhaustive design space exploration is not an option, developers are seeking tools and formal performance analysis methods that can help them decide on an acceptable configuration even though not necessarily the optimal one [118]. The issue of developing a hybrid application aggravates by the different coding environment options for each computation engine. While modern multicore CPUs and GPUs can be programmed using any multi-threading/CUDA capable language such as C/C++ or Fortran, producing efficient FPGA architectures requires the use of hardware description languages such as VHDL or Verilog.

Each of the aforementioned CPU/GPU/FPGA hybrid systems in literature (See *Section 3.2.4*) tries to solve the resource allocation, task mapping and developing platform issues, using its own toolchain since no standardized method exists. For example , the Axel hybrid cluster [89] uses a map-reduce tool over its own Hardware Abstraction Model to decide on the task allocation on the available different-type resources.

The "Chimera" system on the other hand relies on a manual task allocation by the user in order to assign each task to the appropriate computing resource [91]. On both systems, the tasks targeting the GPUs and FPGAs of the cluster need to be developed using the appropriate languages and tools and then combined together to produce the hybrid application. This scheme might seem appealing; however there is enormous developing time difference between programming a CUDA program (matter of hours) and implementing an FPGA-based architecture (weeks or even months).

QP cluster nodes [90], on the other hand, are programmed entirely by a single framework. In [90] the first attempt to develop a comprehensive tooling framework, called Phoenix where a user can develop complete hybrid applications using the C programming language, is presented as well. The Phoenix framework trades-off on having an optimal hybrid system configuration by using Nallatech's C-to-VHDL function generator for generating the necessary FPGA-based hardware architectures executed alongside with task allocation tools in C.

This approach is reasoned on the fact that eventually the performance of optimal ultra-fast accelerators will be limited by the effects of *Amdhal's Law* and is reported to allow for seamless application development without the need to use specialized HDL tools. However, no CPU/GPU/FPGA hybrid application is reported to be generated by the Phoenix framework yet, as only CPU/GPU and CPU/FPGA applications have been reported to be implemented.

The hybrid system presented in [119] has been implemented to use OpenCL (the open-source alternative to nVidia CUDA) and the High-Level Synthesis (HLS) tool ROCCC to produce its GPU kernel code and its FPGA bitmap, respectively. This HLS tool is reported to perform so well, outperforming even handwritten VHDL counterparts. However, no fine-tuning optimization is done to the handwritten modules. An exploration of using roofline performance model for estimating performance of various accelerators during task allocation is given in [119] as well with promising results.

Considering all the above, we have designed a compact hybrid application developing framework which is mainly focusing on reducing the number of tools needed to compile the desired application. The whole process in done through a high-level C language framework where all coding for all the different processing engines is done by running a single executable.

We selected popular and well-tested compilers: *gcc* for multi-threaded CPU applications, and the NVIDIA CUDA Toolkit's *nvcc* for GPU kernels. The gcc compiler supports multi-threading - through pthreads or OpenMP - and *nvcc* can use *gcc* as host compiler thus we can achieve hybrid CPU/GPU applications though a single compilation. For instance, invoked CPU child threads can each execute GPU kernels concurrently on different GPU devices.

As already mentioned, we can implement FPGA hardware modules either through HDL tools or through available HLS tools. We focused on the HDL option as we believe it offers fine-

grained control over the components under development and more configuration options for optimizing performance later on. Moreover, we and our collaborators have developed an already impressive number of modular hardware architectures that target almost any type of bioinformatics and biomedical problems [88]. We are continuously expanding this library of architectures with more modules as well. As such, the extraction of specific modules for reusing them for executing assigned tasks in a hybrid CPU/GPU/FPGA system, is easy to perform.

The FPGA-based architectures are developed in the Xilinx ISE Design Suite and the respective bitstreams to be downloaded on the accelerators, are generated through Xilinx tools as well. The only issue at hand is how we can combine the bitstreams with the CPU/GPU code to create a truly hybrid application. Instead of manually configuring each FPGA of our hybrid system and then execute the CPU/GPU software, we modified the Dini's Group AETEST software - provided with the DNBFC_S12_PCIe multi-FPGA boards.

AETEST is a tool that allows for configuring each FPGA through PCI-Express and even allows for DMA data transfer between the FPGAs and the host CPU. Our modification allows for integrating the AETEST libraries and drivers into any C application we develop. As such, we are now able to program the FPGAs directly though the main hybrid application and do even more; Each application is a single hybrid CPU/GPU/FPGA executable which can transfer data from and to any processing engine present to our system.

While we have implemented some of the first presented truly hybrid bioinformatics applications, having an automated resource allocation tool for partitioning the tasks among the available CPU/GPU/FPGA resources of our hybrid system is still an open research topic. Several options are being explored with no clear winner so far, since partitioning an existing bioinformatics algorithm in distinct tasks needs an extensive amount of profiling and tests.

Each task is then classified in multiple and diverse classes. Each class response to a characteristic of the task under consideration: percentage of execution time, I/O operations, memory accesses, used memory's spatial/temporal locality, power, available parallelism and its type, scalability and data/control dependencies on other tasks. Moreover, this process is getting more complicated as inherently compute-intensive tasks can be transformed to memory-intensive tasks through exploring parallelism.

This is an open issue in task allocation problems since memory-intensive tasks can - for example - greatly benefit from FPGAs, while compute-intensive problems can execute on CPUs with adequate performance. Taking all the above facts into account, finding a optimal solution to the task allocation problem of hybrid heterogeneous systems is not something that can be achieved through heuristics alone.

The developer's experience in developing such heterogeneous applications *still* is a crucial part of the equation leading to the optimal solution. As such, a degree of old-school design space exploration is needed to be performed before zeroing into an acceptable task allocation among the available resources.

7.3 **Filtered Sequence Alignment as a CPU/GPU/FPGA Application**

We have extensively studied bioinformatics algorithms for protein sequence alignment in the past [88] and we have developed a series of multi-threaded and GPU-based software implementations; complemented with a number of single-FPGA and multi-FPGA architectures as well. As such, our first to report hybrid CPU/GPU/FPGA application is naturally to be in a bioinformatics field that we have experience at.

We selected the filtered protein sequence alignment problem as it is solved with a process consisting of distinct steps: first the input sequences are filtered for low-complexity regions that can negatively impact the alignment results with a tool such as CAST; the sequences are then masked; and lastly the filtered sequences are aligned against a database of already known proteins for identifying similarities between them with a tool such as BLAST. Two sequences which are similar enough, are likely to have similar function [120]. As such, sequence alignment is crucial in further expanding our knowledge in biology and medicine.

The individual tasks of the filtered sequence alignment have been studied and profiled in previous works [88][104][21][20][18]. CAST is a popular filtering tool for low-complexity regions that is reported to have superior quality results than other similar tools. We have developed CAST versions for all available computing engines as an attempt to evaluate each of their merits.

BLAST is one of the most popular bioinformatics tools and our collaborators in Technical University of Crete have developed extensive FPGA-based implementations that yield identical results as well as a multi-FPGA implementation. Moreover, we have access to the official multi-threaded BLASTp software since it is available from NCBI website [121] and we can evaluate BLAST on GPUs using the open-source GPU- BLAST software [12].

We have modified the open-source, third-party, software implementations to use them as pipeline modules for the various hybrid configurations we used for evaluating EVAGORAS heterogeneous platform. The outline of the hardware resources available on EVAGORAS is shown in Figure 7.4.

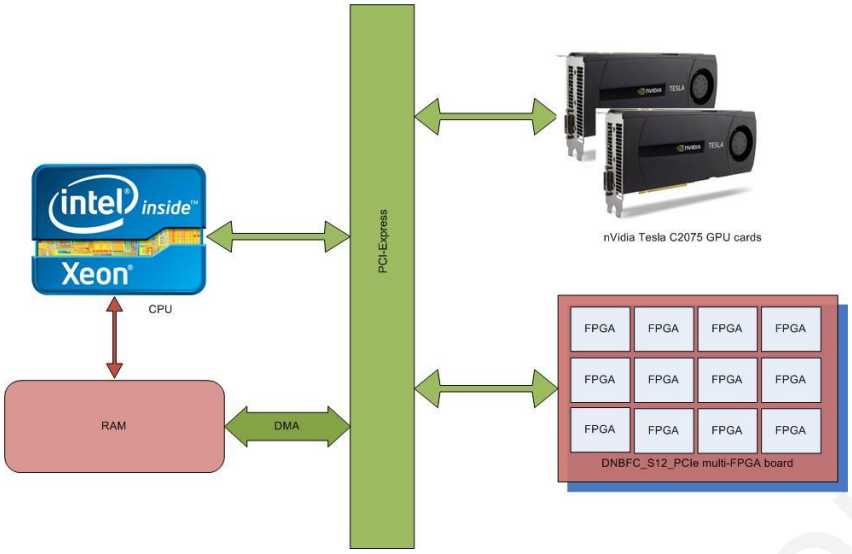


Figure 7.4: University of Cyprus EVAGORAS high-end CPU/GPU/FPGA Hybrid Platform

As already mentioned above, the filtered sequence alignment problem tasks (*LCR detection, LCR masking, Sequence alignment*) can be efficiently partitioned in two distinct applications: CAST followed by BLAST. Each application can be executed either on multicore CPUs or GPUs or FPGAs, using modules based on the original software and hardware implementations, we modified for EVAGORAS. These modules are described in the list below:

- *mCAST2.0:* A variant of the multicore software presented in *Chapter 4.1*.
- *GPU_CAST:* The GPGPU software presented in *Chapter 5.1*. As a reminder, LCR detection is performed on the GPUs and masking is done by the multicore CPU.
- *FPGA_CAST:* The reconfigurable hardware architecture presented in *Chapter 6.1*.

- *BLASTp:* The official BLAST multi-threaded software for protein sequence alignment from NCBI.
- *GPU-BLAST:* A variant of the open-source GPU-based software. The original code was downloaded from the website of [12].
- *FPGA-BLAST:* The reconfigurable hardware architecture developed by our collaborators in Technical University of Crete [104].

7.3.1 EVAGORAS Hybrid Configurations

The capabilities of EVAGORAS hybrid system as a computation platform for bioinformatics applications were explored by conducting a comprehensive design-space exploration for the filtered sequence alignment problem.

We used the various CAST and BLAST modules mentioned above, to build our configurations. The modules are interconnected through data transfer FIFOs, implemented either through the PCI-Express bus or as local memory structures. These configurations shown in Table 8, were then executed on our EVAGORAS hybrid CPU/GPU/FPGA system as a proof-of-concept for the merits of using such a system in bioinformatics.

TABLE 8: CONFIGURATIONS FOR EVAGORAS PLATFORM EVALUATION

mCAST2.0 & BLASTp	
LCR detection	CPU
LCR masking	CPU
Sequence Alignment	CPU
mCAST2.0 & GPU-BLAST	
LCR detection	CPU
LCR masking	CPU
Sequence Alignment	GPU
mCAST2.0 & FPGA-BLASTp	
LCR detection	CPU
LCR masking	CPU
Sequence Alignment	FPGA
GPU_CAST & BLASTp	
LCR detection	GPU
LCR masking	CPU
Sequence Alignment	CPU
GPU_CAST & GPU-BLAST	
LCR detection	GPU
LCR masking	CPU
Sequence Alignment	GPU
GPU_CAST & FPGA-BLASTp	
LCR detection	GPU
LCR masking	CPU
Sequence Alignment	GPU
FPGA_CAST & BLASTp	
LCR detection	FPGA
LCR masking	FPGA
Sequence Alignment	CPU
FPGA_CAST & GPU-BLAST	
LCR detection	FPGA
LCR masking	FPGA
Sequence Alignment	GPU
FPGA_CAST & FPGA-BLASTp	
LCR detection	FPGA
LCR masking	FPGA
Sequence Alignment	FPGA

The *GPU_CAST & FPGA_BLASTp* configuration of Table 8, is shown in Figure 7.5 as an example. The dataflow of the application is controlled by the system CPU for all nine different configurations. The *GPU_CAST & FPGA_BLAST* configuration works as follows:

The system's CPU first partitions the input sequences into batches (having *N* sequences, default=50) that then transfers to the GPU for executing GPU_CAST [20]. As soon as the filtering is finished for each batch, the low-complexity regions discovered are transferred back to the CPU where the masking is done. The updated sequences are then transferred to an idle FPGA which runs the FPGA-based BLAST developed in TUC. Lastly, the results are then evaluated on the CPU and the batch is marked as completed. The application has data transfer FIFOs that hold all data transferred through the PCI-Express link. The system is capable of running up to two batches *concurrently* as it has two Tesla C2075 graphic cards and twelve Spartan6 FPGAs installed.

The task partitioning between the available resources for each different configuration is shown in Table 8. PCI-Express is the default data transfer protocol interconnecting two task executed in different processing engine. In the cases where two or more consecutive tasks are executed in the same engine (an example should be the *mCAST2.0 & BLASTp* configuration) the data transfer between the modules is handled by local memory FIFO structures.

We decided not to merge successive modules running on same type resources, since we aim to prove that a hybrid platform can efficiently use third-party software or already-developed hardware IP cores as modules for building bioinformatics applications of higher complexity. In our evaluation, filtered sequence alignment serves as a real-world test case scenario.

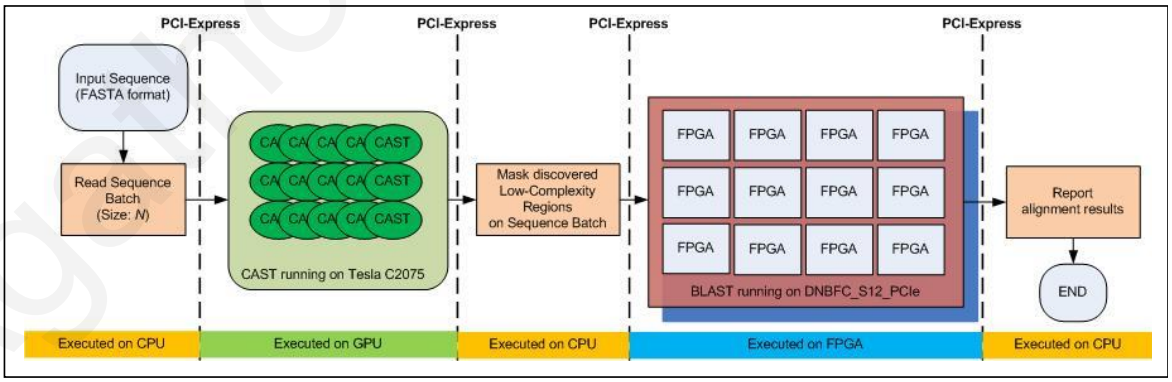
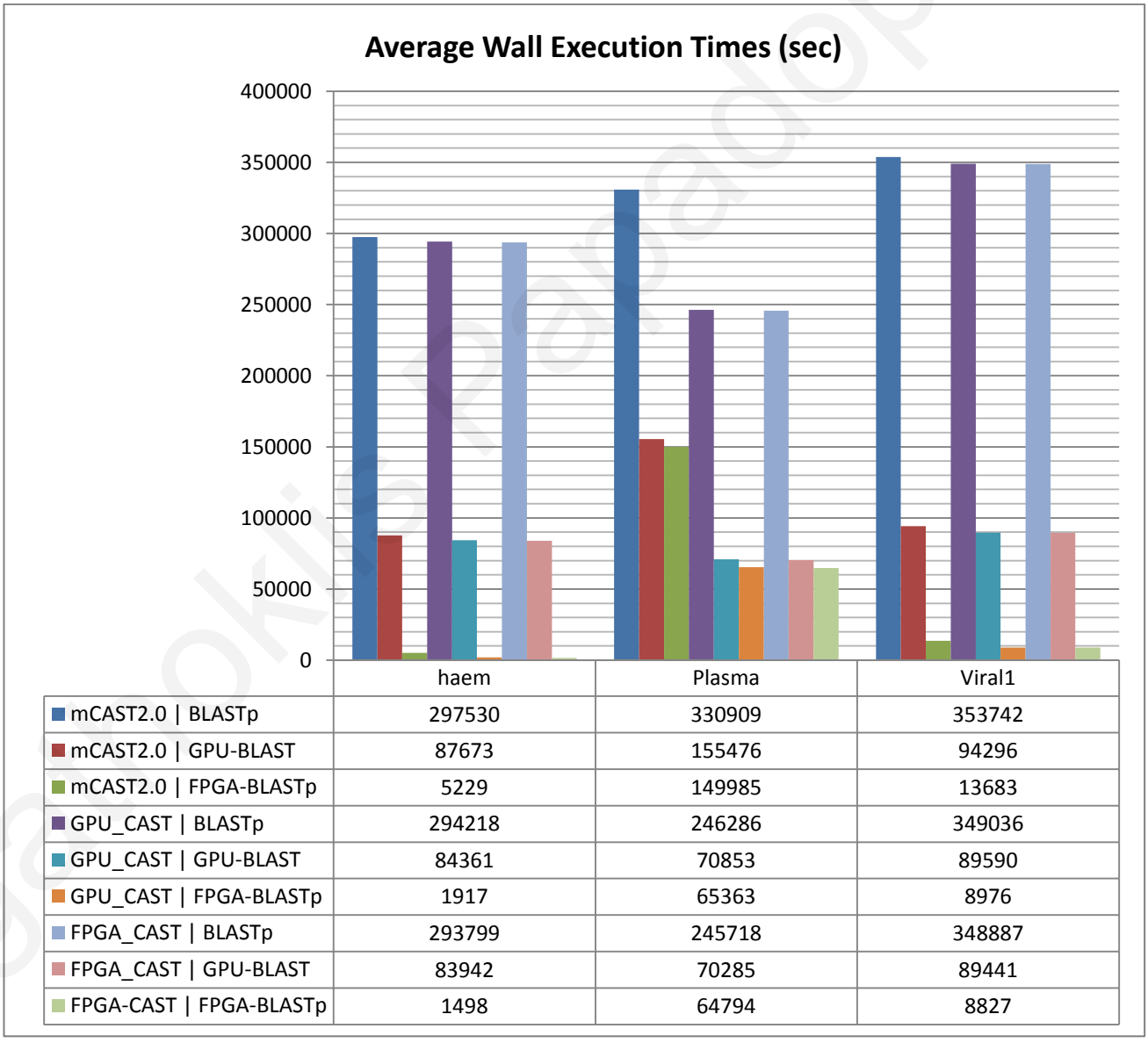


Figure 7.5: GPU_CAST&FPGA-BLASTp Filtered Sequence Alignment. Task Partitioning and Workflow for every Sequence Batch

7.3.2 Evaluation & Results

We evaluated EVAGORAS hybrid CPU/GPU/FPGA system as a computation platform for bioinformatics applications by running each configuration of filtered sequence alignment twenty times. We used the proteomic datasets for *Appendix A: Proteomic Benchmarks* as input. In each execution of one configuration, the application first identified the LCR regions in its input dataset, masked them, and then aligned the masked dataset against the same dataset. The averaged execution times for all configurations (with *batch size = 50*) of each proteomic benchmark dataset are shown in Figure 7.6.



	haem	Plasma	Viral1
mCAST2.0 | BLASTp	297530	330909	353742
mCAST2.0 | GPU-BLAST	87673	155476	94296
mCAST2.0 | FPGA-BLASTp	5229	149985	13683
GPU_CAST | BLASTp	294218	246286	349036
GPU_CAST | GPU-BLAST	84361	70853	89590
GPU_CAST | FPGA-BLASTp	1917	65363	8976
FPGA_CAST | BLASTp	293799	245718	348887
FPGA_CAST | GPU-BLAST	83942	70285	89441
FPGA-CAST | FPGA-BLASTp	1498	64794	8827

Figure 7.6: Average Wall Clock Execution Times for all Filtered Sequence Alignment Configurations.

All hybrid configurations of our application yielded identical results with the official multi-threaded software counterparts. The multi-threaded configuration *mCAST2.0 & BLASTp* was found to be the slowest for all three datasets. This was expected since GPGPU-based implementations and reconfigurable hardware architectures offer higher performance because they allow for massively parallel implementations.

We use *mCAST2.0 & BLASTp* configuration as our evaluation baseline and we evaluated the rest configurations against its results. The average speedups for each configuration against the selected *mCAST2.0 & BLASTp* configuration are shown in Figure 7.7. Moreover, the percentage of the overall execution time used by the CAST and the BLAST modules for each configuration is shown in Figure 7.8.

Figure 7.6 clearly shows that all configurations using *BLASTp* for the sequence alignment task are much slower than the rest, for all benchmark datasets. Figure 7.8 shows that all *BLASTp* configurations spend more than 95% of their execution time running BLAST. BLAST is responsible the heavier computation load of the filtered sequence alignment application. As such, *Amdahl's Law* states that accelerators focusing only on CAST and not

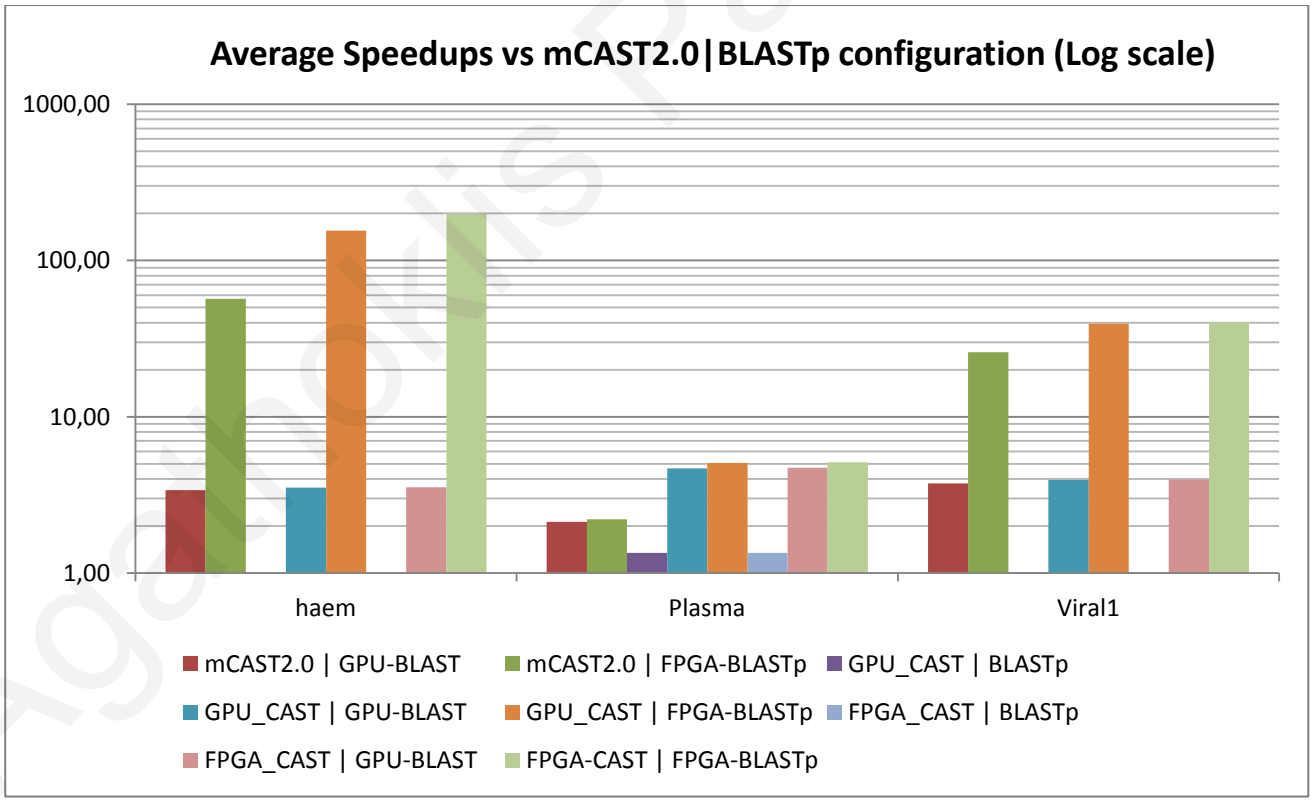


Figure 7.7: Speedups of all Configurations against the mCAST2.0&BLASTp Configuration.

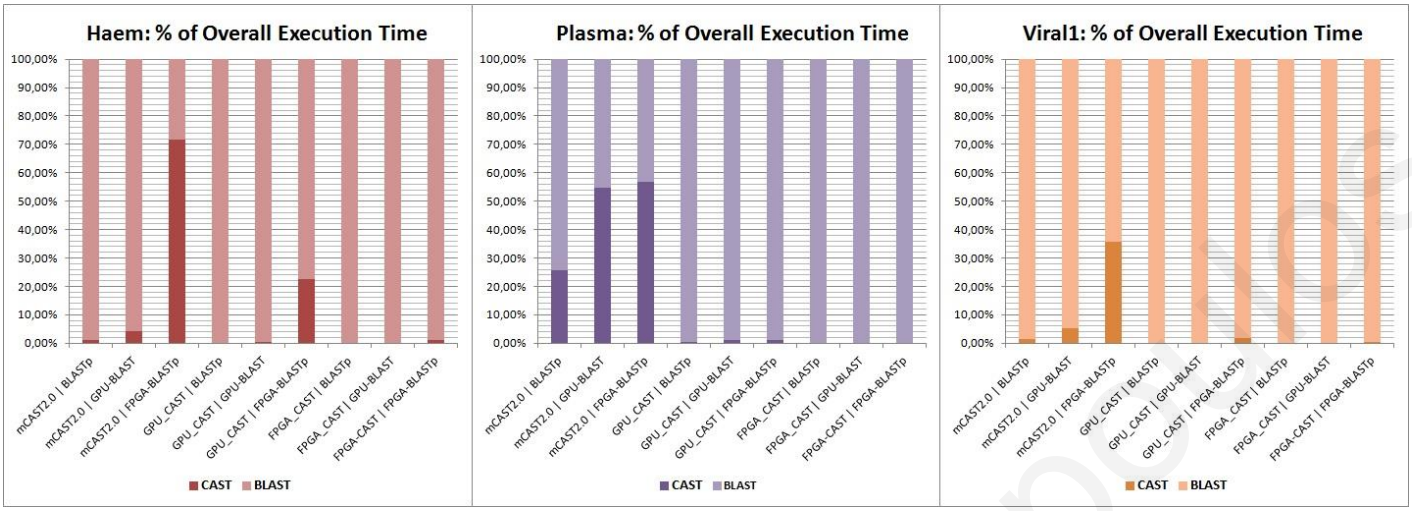


Figure 7.8: CAST and BLAST Execution Percentages for all Configurations.

BLAST will not perform well. This fact is evident since configurations with *BLASTp* have \almost no speedups. Figure 7.7 shows that BLASTp configurations have a negligible speedup of 1.34x only on the LCR-heavy *plasma* dataset.

Accelerating just BLAST execution through GPU-BLAST, allows for speedups between 2x-4x depending on the dataset. The *mCAST2.0 & FPGA-BLASTp* configuration shows speedups up to 56x. Faster execution of BLAST leaves the burden to mCAST2.0 execution. Figure 7.8 shows that CAST can occupy up to 55% of the overall execution time in these configurations.

Accelerating CAST as well can lead to further performance gains. For example, GPU_CAST configurations combined with GPU-BLAST or FPGA-BLASTp allow for ~4x speedups. FPGA_CAST barely offers any additional performance gains when combined with BLAST accelerators. Figure 7.7 shows in FPGA_CAST configurations we have slightly higher speedups than the GPU_CAST configurations.

These results show that a hybrid CPU/GPU/FPGA system offers *the option to gain almost equal performance gains with a full FPGA-based hardware architecture* (like our *FPGA_CAST & FPGA-BLASTp* configuration) by accelerating the heavier tasks on FPGAs and less heavier tasks on GPUs. This is faster to build than implementing hardware architectures for all tasks - computationally heavy or not - for running them on a multi-FPGA system.

The presented results were bounded by the PCI-Express bandwidth limitations. The batches of protein sequences must be transferred between the various accelerators. Each configuration has its own data transferring bottlenecks that differ from the rest.

Taking the *GPU_CAST & FPGA_BLASTp* configuration as an example, we have identified that the PCI-Express link between CAST and BLAST is a bottleneck because the FPGA accelerators running BLAST were consuming the masked batches produced by CAST too fast. This issue seems to lessen as we increased the batch size (almost double speedups for N=100); however the performance starts degrading once again for larger batch sizes since the CPU-to-GPU link becomes the bottleneck of the system. This is a well-known problem of GPGPU computing applications as all data needs to be present on the GPU memory for a GPU kernel to start computing. Increasing the batch size more than N=100 simply means that more data needs to be transferred before GPU-CAST starts and as such latter faster processing units (such as the FPGAs running FPGA_BLASTp) are deprived from data to process.

7.4 **Closing Remarks**

In this chapter, we showcased the benefits of using a CPU/GPU/FPGA hybrid system in the fields of bioinformatics. Our custom-built hybrid system is able to execute hybrid applications consisting of different software and hardware modules communicating through a PCI-Express communication stack which includes our custom-built PCI-Express modules for the FPGA fabric.

The first - in our knowledge - reported truly CPU/GPU/FPGA application of filtered sequence alignment provided more than enough results to justify the use of such a system. We have tested various hybrid configurations for executing the filtered sequence alignment problem on our hybrid system with promising results.

Open issues - such as partitioning an application to tasks and the task allocation to each type of processing engine, as well as the need for standardized application developing tools - are discussed as well. There is open research - conducted by various groups around the world - that tries to answer these challenges that need to be addressed before hybrid computing becomes the norm for HPC.

8 Overview of Contributions, Conclusions and Impact

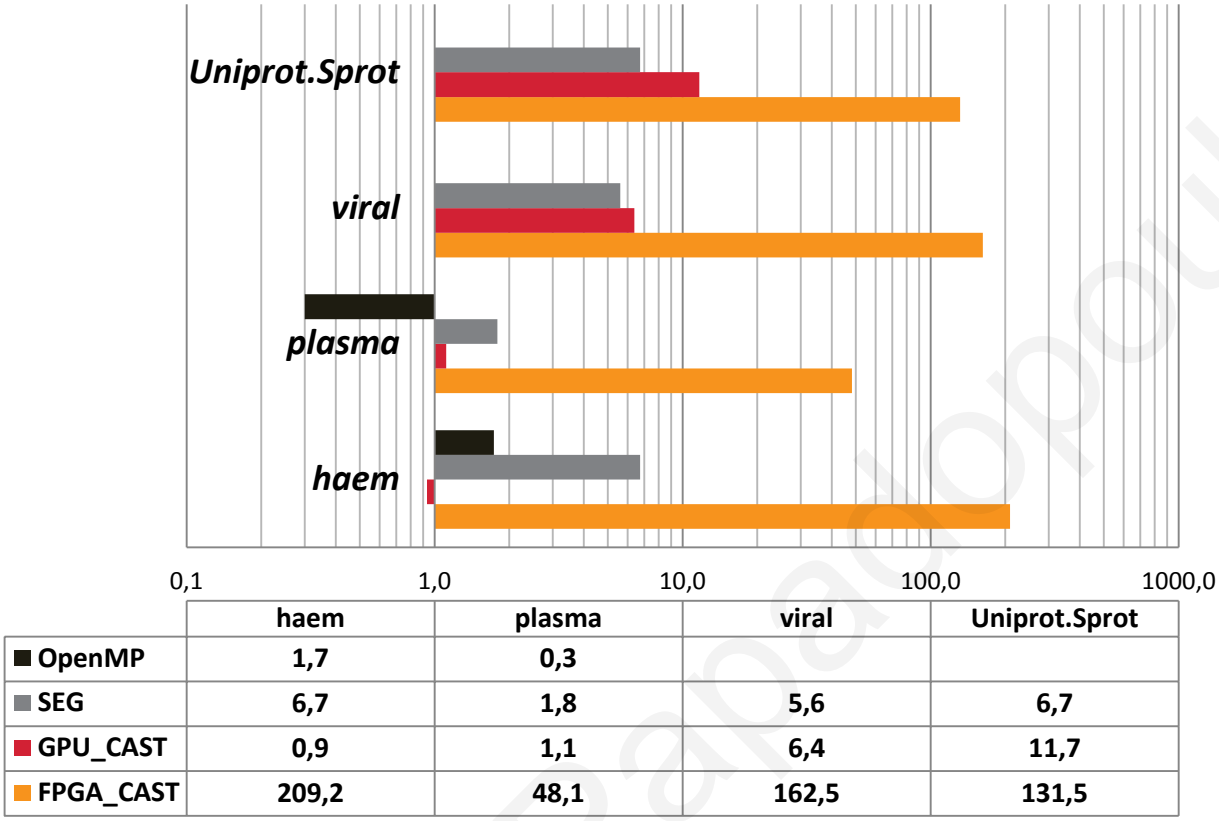
In this final chapter, we provide an overview of this thesis results. The benefits and issues of using multicore CPUs, GPUs and FPGAs as accelerators of bioinformatics and biomedical applications are discussed as well. Moreover, the merits of combining all different paradigms in a single powerful hybrid heterogeneous systems are presented.

Open challenges in the field of heterogeneous high performance computing and future research plans are presented in this chapter as well. This thesis is concluded with the author's thoughts on accelerating computationally intensive applications in the fields of biomedical engineering and biology.

8.1 **Remarks on Computation Engines**

The advantages and disadvantages of using each of multicore CPUs, GPUs, and FPGAs as accelerators for biomedical and bioinformatics applications have become evident from the research results presented in earlier chapters of this thesis.

No single computation engine can be named "best" for the applications we have targeted and this is due to technology or programming paradigm limitations. The speedups of the various CAST implementations against mCAST 2.0 multi-threaded software are presented in Figure 8.1. These results are already discussed in previous chapter, however a comprehensive overview of the qualities and issues of each approach is presented here.



	haem	plasma	viral	Uniprot.Sprot
OpenMP	1,7	0,3		
SEG	6,7	1,8	5,6	6,7
GPU_CAST	0,9	1,1	6,4	11,7
FPGA_CAST	209,2	48,1	162,5	131,5

Figure 8.1: Overview of CAST accelerators
Software results were extracted from a high-end *Intel i7-3960X@3.33GHz/32GB RAM* desktop system with *Nvidia GTX690 GPUs*
Hardware results were extracted from a single CAST instance configured on a *Xilinx Virtex 5* FPGA board

8.1.1 Multi-threading Applications on Multicore CPUs

We have shown that multi-threading programming can take full advantage of the available parallelism in the case of CAST algorithm. We have demonstrated that mCAST implementations greatly outperform the original software (sCAST). However, the available resources on multicore CPUs limit the extracted parallelism.

This is evident from Figure 8.1 in which all other accelerators have reported speedups against the optimized mCAST 2.0. SEG - the heuristic-based alternative to CAST - still performs faster than the multi-threaded CAST. So, in cases where an analysis over a massive dataset which typically can take days, researchers may choose to trade-off CAST quality for SEG speed. SEG needs only one day to perform local complexity analysis on a dataset which mCAST 2.0 requires six.

OpenMP parallel programming framework despite offering faster developing times for implementing multi-threaded applications, the coarse-grain parallelization paradigm used is not performing adequately well for all benchmark datasets running the OpenMP-based CAST.

As we can see on Figure 8.1, the black bar indicating wall clock time execution of the OpenMP CAST despite performing very well for *haem* dataset, is three time slower than mCAST 2.0 on the LCR-heavy *plasma* dataset. The decision on which multi-threading programming paradigm should be used in each application, is left with the reader.

OpenMP offers faster implementation and easier use compared with the traditional multi-threading paradigm, however leaves the thread handling to the operating system and this may hurt the actual performance of computationally-heavy datasets.

Multicore CPUs do offer resources that allow extracting parallelism from bioinformatics and biomedical applications; however the limited number of processing cores present in the commercially available CPUs does not allow for taking full advantage of the available parallelism of these applications.

For instance, the high-end *i7 extreme* processor used to evaluate our implementations offers 12 logical cores while CAST achieves maximum performance when all 20 homopolymers are calculated simultaneously. As such, mCAST 2.0 does not take full advantage of the inherent parallelism of CAST, since can concurrently execute only a subset of the necessary threads on the *i7 extreme* multicore processor.

On the other hand, multicore CPUs have better memory transfer schemes. Systems hosting such processors offer the fastest available schemes for transferring data to be processed from the system's main memory. This is a clear benefit of CPUs against all other accelerators since PCI-Express (or other) connections have less transfer throughput and as we showed with our CPU/FPGA implementation for suffix array construction in the de novo DNA sequence assembly, I/O limitations can massively hinder the achieved performance despite the fact that FPGAs theoretically can have hundreds of specialized processing elements.

8.1.2 GPGPU Programming on GPU Cards

Evaluation results stemming from the implementation presented in *Chapter* 5, have demonstrated the capabilities of the GPGPU programming paradigm in accelerating applications in biomedical engineering and biology. We have demonstrated that GPU_CAST is at least on par with mCAST 2.0 on smaller dataset and showed that the performance benefits increase as the dataset grow in size [20].

For example, GPU_CAST in Figure 8.1 is shown to outperform mCAST 2.0 by an order of magnitude for the *uniprot* dataset. This means that a massive dataset that requires three days of processing with mCAST 2.0, using GPU cards as accelerators the required wall clock time drops to just under four hours.

Moreover, GPUs can be a powerful accelerator that provides high quality results in reasonable time, since now GPU_CAST and the heuristic-based SEG are comparable; GPU-accelerated CAST even outperforms SEG for massive datasets such as *uniprot*.

The implementation of GPU_CAST and the GPU-based LVN model for STN deep brain stimulation revealed some issues of using the GPGPU programming paradigm for accelerating bioinformatics and biomedical applications. Despite that GPUs have hundreds of processing elements, limitations in their use do exist.

Memory transfer overheads are negatively affecting overall wall clock time execution, averaging at 30% for the overall execution time of GPU_CAST. Special precautions need to be taken by the developer in order to minimize their impact on performance. The algorithm needs to be rewritten in order to accommodate the GPGPU paradigm's peculiarity of needing all the data to be processed to be present on the GPU card's memory.

Other issues we have faced when developing our GPU-based implementations include the lack of data-sharing among different GPU kernels. This issue translates to further increase of memory transfer overheads since the data needs to be transferred through the host system. Last but not least issues is the high cost of acquiring a high-end GPU card and the energy cost of running applications on it.

Scaling GPGPU-based computation platforms to clusters with hundreds of GPU cards can lead to massively increased power consumption. On the other hand, small platforms with a handful

of GPU cards can have a relatively small power consumption increase when compared to a traditional CPU-based system.

GPGPU programming paradigm is offering enough benefits to run targeted GPU-based biomedical and bioinformatics applications that have minimized data dependencies and the data to be processed can efficiently partitioned in smaller batches in order to minimize the impact of the small-sized shared memory among GPU threads invoked in a single kernel. In this thesis, we showed that memory overheads can be minimized for large enough or computationally-heavy enough datasets, thus allowing GPGPU paradigm to offer significant performance gains.

8.1.3 FPGA-based Reconfigurable Architectures

In this thesis, FPGAs have been evaluated as a mean to run massively parallel reconfigurable hardware architectures for bioinformatics applications. We have demonstrated the performance benefits, as well as the issues arising from using such FPGA-based architectures through the evaluation of FPGA_CAST architecture as well as the FPGA-based suffix array construction for de novo DNA sequence assembly.

The benefits of using massively parallel hardware architectures are quite evident from Figure 8.1. As we can see, a single instance of the FPGA-based CAST architecture implemented with a very conservative hardware design approach outperforms all other accelerators with orders of magnitude speedups [19].

Moreover, FPGA_CAST is 25x faster that the heuristic-based SEG software. The performance results for multiple instances of the proposed hardware architecture outperform optimized multi-threaded implementations of the CAST algorithm by 100x-5000x times [21].

Our results concur with the results of other FPGA-based bioinformatics and biomedical accelerators found in literature. This fact shows that FPGAs can be powerful options for massively parallel implementations that take full advantage of the available parallelism found in most applications in life sciences and medicine.

On the other hand, there are cases such as our FPGA-based suffix array implementation that despite having all available parallelism extracted, the resulting implementation is not quite achieving the expected performance. In the aforementioned case, a major issue has the lack of the necessary memory to store the massive concatenated input sequence. As such, we had to use a CPU/FPGA hybrid streaming implementation.

The CPU/FPGA implementation for suffix array construction despite initially considered a promising solution, was the reason for identifying another major issue of such heterogeneous architectures. Despite having an ultra-fast reconfigurable hardware architecture, the design does not perform as expected due to not having the necessary I/O bandwidth.

8.1.4 Data Transfer between accelerators and System Memory

Even the fastest available communication schemes such as PCI-Express might fail to provide enough data transfer rate to fully exploit the available parallelism of bioinformatics and biomedical applications. This issue is evident from the cases discussed above and the hybrid CPU/GPU/FPGA system evaluation of Chapter 7 as well. The bottlenecks of the system were the PCI-Express links between the accelerators.

This thesis is providing enough evidence to suggest that hybrid systems could provide a computation platform that can adapt and exploit the most of algorithms used in medicine and life sciences. However, data transfer between accelerators is an open research topic. No available communication scheme is proven to be able to achieve adequate transfer rate for fully exploiting the available parallelism in bioinformatics and biomedical applications.

Current memory hierarchies are built in favour of each host system's CPU. Caches and DMA data transfer schemes and virtual memory allow for applications running on multicore CPUs to use data as optimal as possible even if the massive data is stored in hard drives. Since GPUs and FPGAs were not considered as computation accelerators until recently, no care was taken for enhancing the memory management system both in hardware and OS level.

This lack of memory systems that target all types of computational engines leads to GPGPU-accelerated applications and reconfigurable FPGA-based architectures with significant memory and I/O transfer overheads that negatively affect performance. As we already

mentioned, GPU-CAST has 30% of the overall wall clock time execution time consumed by memory transfer overheads and the FPGA-based suffix array construction architecture underutilizes the FPGA resources since there is no adequate PCI-Express bandwidth for transferring data from- and to- the host system's memory.

8.2 Open Research Challenges

The research results presented in this thesis showed that there are several open research challenges needed to be addressed while attempting to generalize the extraction of parallelism from bioinformatics and biomedical applications.

Multicore CPUs, GPUs and FPGAs have been proven to offer performance improvements when executing parallel and massively parallel version of existing algorithms. Moreover, hybrid heterogeneous systems can potentially be the norm of exploiting available parallelism for most of life science problems by combining the benefits of all accelerators.

Open research challenges do exist in the field. These challenges revolve around addressing the issues identified throughout this thesis. Crucial challenges for advancing the field include:

Standardized developing tools: Developing frameworks and toolchains for developing massively parallel implementations for popular applications is an essential step for further advancing the field. These tools will allow for rapid development of parallel applications with significant performance gains thus minimizing the design and development times for new biological and biomedical applications.

Parallelization and design space exploration tools: There is a need for developing tools for evaluating the available parallelism of a new bioinformatics and biomedical applications and how each accelerator performs on extracting the available parallelism for each application. These tools can be quite useful for selecting the ideal computation engine for accelerating the application under consideration while minimizing the manual design exploration by the developers. Moreover, these tools ideally will provide enough inside for which applications will benefit by a hybrid implementation instead of using just a single accelerator type.

Hybrid application modelling and task partitioning: A modelling language that enables users to efficiently partition algorithm sections and map them either on the reconfigurable platforms, or the GPUs using an automated procedure. The developing of such tools will boost performance of hybrid applications while minimizing design and development time for new bioinformatics and biomedical applications. These tools need to be able to take into account not only the computation steps of each algorithm, but the memory operations needed and the limits on data transfer for each accelerator as well.

Performance metrics: Metrics need to be also defined better, as the cost and energy become more important and contradict the performance benefits. Standardized metrics will allow for researchers to accurately evaluate new accelerators, new hybrid systems and their configurations and the actual performance of massively parallelized applications in medicine and biology.

To conclude, we are cautiously optimistic that these obstacles can be overcome and the developing of new massively parallel bioinformatics and biomedical applications will become easier while gaining significant performance gains.

Moreover, addressing these open research challenges will allow to combine the positives of the all computation paradigms, thus creating a truly hybrid computational platform that integrates performance and flexibility, along with energy-efficiency and cost.

8.3 **Future Work**

The ground for advancing the field of accelerating bioinformatics and biomedical applications is set through this thesis. The research results presented here provided enough evidence for the benefits of using parallel accelerators such as multicore CPUs, GPUs and FPGAs, as well as heterogeneous systems combining all paradigms in medicine and biology.

We will continue to pursue the advancement of this field through extensive hybrid implementations of other bioinformatics and biomedical algorithms. These implementations will allow for extracting additional information on how each identified issue affects a hybrid application's performance. The gained knowledge will allow us for laying out plans for tackling the task partitioning issue of hybrid applications.

8.4 **Conclusions**

In this thesis we present our results and research for utilizing CPUs, GPUs and FPGAs as massively parallel computational engines for bioinformatics and biomedical applications. We selected to accelerate algorithms - such as CAST, a dynamic programming methodology executed over massive amounts of data - whose traits are typical for most of these applications. We mapped the selected applications on state-of-the-art computational engines of all available processing technologies.

The FPGA-based hardware acceleration architecture for accelerated masking of LCRs in protein sequences showed a speedup is over two orders of magnitude with a very conservative hardware design approach. The GPGPU implementation proposed for LCR discovery and masking provided results which indeed show improved execution times when compared to CPU-based software. However, their performance still lacks orders of magnitude behind, in most cases, from their FPGA-based counterparts.

GPGPU implementations have limitations such as: unavoidable memory transfer overheads between the host CPU and the GPU - since the data to be processed must be present on the GPU memory, limited memory resources per GPU core, adjustments that are necessary for double floating number operations. As such, the GPGPU paradigm cannot keep up with the highly optimized FPGA-based hardware architectures like the ones discussed above [17]. Taking CAST as an example, GPU_CAST while faster than the multithreaded version [20] for large benchmarks such as the *viral.1* and the *uniprot* datasets, it was still orders of magnitude slower than the FPGA-based implementation [21].

However, the answer whether the FPGAs or the GPUs are suitable for bioinformatics is not strictly one-dimensional. State-of-the-art methodologies for analysing and processing biological and medical data are consisted of enough processing steps with widely different processing characteristics and complexity that can make the selection of the "right" acceleration platform non-trivial.

The future of high-performance computing (HPC) lays in the use of hybrid CPU/GPU/FPGA heterogeneous computing systems. We showcased the benefits of using such a hybrid system

in the fields of bioinformatics and biomedicine. Our hybrid system has been built to be at least on par with the rest of the systems reported in literature.

The first - to our knowledge - reported truly CPU/GPU/FPGA application provided more than enough results to justify the use of such a system in biology and medicine. Open issues - such as partitioning an application to tasks and the task allocation to each type of processing engine, as well as the need for standardized application developing tools - are discussed as well. There is open research - conducted by various groups around the world - that tries to answer these challenges that need to be addressed before hybrid computing becomes the norm for HPC.

Looking into the future, a "marriage" between a highly parallel GPGPU platform and a flexible and reconfigurable FPGA platform alongside with powerful multicore processors, can merge the positives from all computing paradigms and perhaps provide a powerful and cost-effective paradigm for accelerating, among others, bioinformatics and biomedical applications.

References

[1] M. L.Metzker, "Sequencing technologies - the next generation," *Nature Reviews Genetics,* vol. 11, no. 1, pp. 31-46, 2008.

[2] E. R. Mardis, "Anticipating the $1,000 genome," *Genome Biology,* vol. 7, no. 7, p. 112, 2006.

[3] C.A. Ouzounis, "Rise and Demise of Bioinformatics? Promise and Progress," *PLoS Computational Biology,* vol. 8, no. 4, p. e1002487, 2012.

[4] R. Rost, B. Nair, "Sequence conserved for subcellular localization," *Protein Science,* vol. 11, no. 12, pp. 2836-2847, 2002.

[5] A. J. Enright, S. van Dongen, and C. A. Ouzounis, "An efficient algorithm for large-scale detection of protein families," *Nucleic Acids Research,* vol. 30, no. 7, pp. 1575-1884, 2002.

[6] "National Human Genome Research Institute (NHGRI)," [Online]. Available: http://www.genome.gov/sequencingcosts/. [Accessed Semptember 2014].

[7] A. E. Darling, L. Carey, and W. Feng-chun, "The design, implementation, and evaluation of mpiBLAST," *ClusterWorld,* vol. 2003, pp. 13-15, 2003.

[8] A. Gara et al., "Overview of the Blue Gene/L system architecture," *IBM Journal of Research and Development,* vol. 49, no. 2, pp. 195-212, 2005.

[9] J. Denis Enderle, J. D. Bronzino, Introduction to Biomedical Engineering, Academic Press, 2012.

[10] European Magnetic Resonance Forum, "Facts and Figures," in *Magnetic Resonance, a critical peer-reviewed introduction (online),* The Round Table Foundation, 2012.

[11] N. Ty Smith, Kenton R. Starko, "Physiological Systems Modeling," in *Encyclopedia of Medical Devices and Instrumentation*, John Wiley & Sons, 2006.

[12] P. Vouzis and N. Sahinidis, "GPU-BLAST: Using graphics processors to accelerate protein sequence alignment," *Bioinformatics,* vol. 2010, no. 27, pp. 182-188, 2010.

[13] C. Fletcher, I. Lebedev, N. Asadi, D. Burke, and J. Wawrzynek, "Bridging the GPGPU-FPGA efficiency gap," in *19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '11),* New York, 2011.

[14] J. Fowers, G.Brown, P.Cooke, G. Stitt, "A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications," in *ACM/SIGDA international symposium on Field Programmable Gate Arrays*, New York, 2012.

[15] C. Eddington, B. Ray, "Multi-Gigahertz FPGA Signal Processing," *Synopsys Insight Newsletter ,* vol. 1, p. Online, 2013.

[16] S. Sarkar, G. R. Kulkarni, P. P. Pande, and A. Kalyanaraman, "Network-on-Chip Hardware Accelerators for Biological Sequence Alignment," *IEEE Transactions on Computers,* vol. 59, no. 1, pp. 29-41, 2010.

[17] T. Theocharides, A. Papadopoulos, et al., “Reconfiguring the Bioinformatics Computational Spectrum: Challenges and Opportunities of FPGA-Based Bioinformatics,” *IEEE Design and Test Magazine: Special Issue on Hardware Acceleration in Computational Biology,* vol. 31, no. 1, pp. 62-73, 2014.

[18] A. Papadopoulos et al., “Opportunities from the use of FPGAs as platforms for bioinformatics algorithms,” in *12th IEEE International Conference on Bioinformatics & Bioengineering (BIBE)*, 2012.

[19] A. Papadopoulos, V. J. Promponas, T. Theocharides, "Towards systolic hardware acceleration for local complexity analysis of massive genomic data," in *ACM/IEEE Great Lakes Symposium on VLSI - GLVLSI 2012*, 2012.

[20] A. Papadopoulos, et. al, “GPU technology as a platform for accelerating local complexity analysis of protein sequences,” in *35th Annual International Conference of the IEEE Engineering in Medicine and Biology Society - EMBC2013*, 2013.

[21] A. Papadopoulos, I. Kirmitzoglou, V. J. Promponas, T. Theocharides, “FPGA-based hardware acceleration for local complexity analysis of massive genomic data,” *Integration, The VLSI Journal, Elsevier,* vol. 46, no. 3 - Special Issue on Hardware for Bioinformatics Applications, Integration, pp. 230-239, 2013.

[22] A. Papadopoulos, T. Theocharides et al., “Towards a Hybrid CPU/GPU/FPGA Platform for Accelerating Bioinformatics and Biomedical Applications,” *IEEE Design & Test Magazine,* p. (under review), 2014.

[23] S. Zierke, J.D. Bakos, “FPGA acceleration of the phylogenetic likelihood function for Bayesian MCMC inference methods,” *BMC Bioinformatics,* vol. 11, no. 184, p. Online, 2010.

[24] S. F. Altschul et al, “Basic local alignment search tool,” *Journal of Molecular Biology,* vol. 215, no. 3, pp. 403-410, 1990.

[25] S. F. Altschul et al., “Gapped BLAST and PSI-BLAST: A new generation of protein database search programs,” *Nucleic Acids Research,* vol. 25, no. 17, pp. 3389-3402, 1997.

[26] September 2014. [Online]. Available: http://www.scopus.com. [Accessed September 2014].

[27] S. Karlin and S. F. Altschul, “Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes,” *Proceedings of the National Academy of Sciences of the United States of America,* vol. 87, no. 6, pp. 2264-2268, 1990.

[28] J. C. Wootton, “Sequences with ‘unusual’ amino acid compositions,” *Current Opinion in Structural Biology,* vol. 4, no. 3, pp. 413-421, 1994.

[29] J. C. Wootton and S. Federhen, “Statistics of local complexity in amino acid sequences and sequence databases,” *Computers & Chemistry,* vol. 17, no. 2, pp. 149-163, 1993.

[30] W. Haerty and G. B. Golding, “Low-complexity sequences and single amino acid repeats: not just ‘junk’ peptide sequences,” *Genome,* vol. 53, no. 10, pp. 753-762, 2010.

[31] G. Gill, E. Pascal, Z.H. Tseng, R. Tjian, “A glutamine-rich hydrophobic patch in transcription factor Sp1 contacts the dTAFII110 component of the Drosophila TFIID complex and mediates transcriptional activation,” *Proceedings of the National Academy of Sciences, USA,* vol. 91, no. 1, pp. 192-196, 1994.

[32] V. J. Promponas et al., "CAST: an iterative algorithm for the complexity analysis of sequence tracts," *Bioinformatics,* vol. 16, no. 10, pp. 915-922, 2000.

[33] D. Benton et al., "GenBank," *Nucleic Acids Research,* vol. 40, pp. D48-D53, 2012.

[34] T.F. Smith, M.S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology,* vol. 147, no. 1, pp. 195-197, 1981.

[35] K.P Michmizos, K.S Nikita, "Can we infer subthalamic nucleus spike trains from intranuclear local field potentials?," in *Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC2010),* 2010.

[36] K.P. Michmizos, D. Sakas, K.S. Nikita, "Prediction of the timing and the rhythm of the parkinsonian subthalamic nucleus neural spikes using the local field potentials," *IEEE Transactions of Information Technology in Biomedicine,* vol. 16, no. 2, pp. 190-197, 2012.

[37] K. Kostoglou, G.D. Mitsis et al., "Prediction of the Parkinsonian subthalamic nucleus spike activity from local field potentials using nonlinear dynamic models," in *12th International Conference on Bioinformatics & Bioengineering (BIBE 2012),* 2012.

[38] T-S. Kim, R. E. N. Shehada, V. Z. Marmarelis, "Nonlinear modeling of ultrasonic transmit-receive system using Laguerre-Volterra networks," in *Medical Imaging 2003: Ultrasonic Imaging and Signal Processing,* 2003.

[39] J. Kennedy, R. Eberhart, "Particle Swarm Optimization," in *IEEE International Conference on Neural Networks* , 1995.

[40] J. Kennedy, R.C. Eberhart, Swarm Intelligence, Morgan Kaufmann, 2001.

[41] D. Ashlock, Evolutionary Computation for Modeling and Optimization, Springer, 2006.

[42] nVIDIA, September 2014. [Online]. Available: https://developer.nvidia.com/cuRAND.

[43] P. Flicek, E. Birney, "Sense from sequence reads: methods for alignment and assembly," *Nat Methods,* vol. 2009, no. 6, pp. S6-12, 2009.

[44] W. Zhang et al., "A Practical Comparison of De Novo Genome Assembly Software Tools for Next-Generation Sequencing Technologies," *PloS one,* vol. 6, no. 3, p. e17915, 2011.

[45] N. Nagarajan, M. Pop, "Sequence assembly demystified," *Nature Reviews Genetics,* no. 14, pp. 157-167, March 2013.

[46] P. Medvedev, et al., "Computability of Models for Sequence Assembly," in *Algorithms in Bioinformatics,* Springer Berlin Heidelberg, 2007, pp. 289-301.

[47] M. Dublin, "The Exciting World of Exascale," 2010.

[48] D. Gusfield, Algorithms on Strings, Trees and Sequences - Computer Science and Computational Biology, Cambridge University Press, 1997.

[49] T.H. Cormen, C.E. Leiserson, R.L.Rivest, C. Stein, Introduction to Algorithms, McGraw-Hill, 2003.

[50] G. Chartrand, Introductory Graph Theory, Dover Publications, 1984.

[51] P. E.C. Compeau, P. A. Pevzne, G. Tesler, "How to apply de Bruijn graphs to genome assembly," *Nature biotechnology,* vol. 29, no. 11, pp. 987-991, 2011.

[52] Z. Li, Y. Chen et al., "Comparison of the two major classes of assembly algorithms: overlap-layout-consensus and de-bruijn-graph," *Briefings in Functional Genomics,* vol. II, no. 1, pp. 25-37, 2011.

[53] P.A. Pevzner, Computational Molecular Biology: An Algorithmic Approach, MIT Press, 2000.

[54] M. C. Schatz, A.r L. Delcher, S. L. Salzberg, "Assembly of large genomes using second-

generation sequencing," *Genome Research,* vol. 9, no. 20, pp. 1165-1173, 2010.

[55] E.W. Myers, "The fragment assembly string graph," *Bioinformatics,* vol. 2005, no. 21, p. ii79–ii85, 2005.

[56] B.S.C. Varma, K. Paul, M. Balakrishnan, D. Lavenier, "FAssem: FPGA Based Acceleration of De Novo Genome Assembly," in *IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM2013)*, 2013.

[57] B.S.C. Varma, K. Paul, M. Balakrishnan, "Accelerating Genome Assembly Using Hard Embedded Blocks in FPGAs," in *27th International Conference on VLSI Design & 13th International Conference on Embedded Systems*, 2014.

[58] S. Aluru, N. Jammula, "A Review of Hardware Acceleration for Computational Genomics," *IEEE Design & Test,* vol. 31, no. 1, pp. 19-30, 2014.

[59] B. Schmidt, R. Sinha, B. Beresford-Smith, S. J. Puglisi, "A fast hybrid short read fragment assembly algorithm," *Bioinformatics,* vol. 25, no. 17, pp. 2279-2280, 2009.

[60] B. Cazaux, T. Lecroq, E. Rivals, "From Indexing Data Structures to de Bruijin Graphs," 2014.

[61] U. Manber, G. Myers, "Suffix arrays: a new method for on-line string searches," in *1st Annual ACM-SIAM Symposium on Discrete Algorithms*, 1990.

[62] S. Kurtz, "Reducing the space requirement of suffix trees," *Software: Practice and Experience,* vol. 29, no. 13, p. 1149–1171, 1999.

[63] S. J. Puglisi, W.F. Smyth, A. H. Turpin, "A Taxonomy of Suffix Array Construction Algorithms," *ACM Computing Surveys,* vol. 39, no. 2, p. Article No. 4, 2007.

[64] R. M. Karp, R. E. Miller, A. Rosenberg, "L.: Rapid identification of repeated patterns in strings, trees and arrays," in *4th Symposium on Theory of Computing* , 1972.

[65] M. Farach, "Optimal suffix tree construction with large alphabets," in *38th Annual Symposium on Foundations of Computer Science,* 1997.

[66] J. Kärkkäinen, P. Sanders, "Simple Linear Work Suffix Array Construction," *Automata, Languages and Programming. Lecture Notes in Computer Science,* vol. 2719, no. 2003, pp. 945-955, 2003.

[67] F. Kulla, P. Sanders, "Scalable parallel suffix array construction," *Parallel Computing,* vol. 33, no. 9, pp. 605-612, 2007.

[68] R. Dementiev, J. Kärkkäinen, J. Mehnert, P. Sanders, "Better external memory suffix array construction," *Journal of Experimental Algorithmics,* vol. 12, no. 1, p. Article No. 3.4, 2008.

[69] J. L. Hennessy, D. A. Patterson, Computer Architecture: A Quantitative Approach., Elsevier, 2012.

[70] A. W. Strong, E. Y. Wu, R-P. Vollertsen, J. Sune, G. La Rosa, T. D. Sullivan, S. E. Rauch, III, Reliability Wearout Mechanisms in Advanced CMOS Technologies, Willey-IEEE Press, 2009.

[71] A. Papadopoulos, T. Theocharides, M.K. Michael, "Towards optimal CMOS lifetime via unified reliability modeling and multi-objective optimization," in *IEEE International Symposium on Circuits and Systems - ISCAS 2011*, 2011.

[72] Intel Corporation, "Intel ARK: Your source for Intel product information," [Online]. Available: http://ark.intel.com/. [Accessed September 2014].

[73] Philips, Inc., "Introduction to VLIW Computer Architecture," Philips Semiconductors , 1997.

[74] ITRS Consortium, “International Technology Roadmap for Semiconductors,” 2013.

[75] Fox News - Technology, 11 2010. [Online]. Available: http://www.foxnews.com/tech/2010/11/24/intel-talks-core-processor/. [Accessed September 2014].

[76] G. M. Amdahl, “Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities,” in *AFIPS Conference Proceedings*, 1967.

[77] M.R. Garey, D.S. Johnson, A Guide to the Theory of NP-Completeness, W.H. Freeman and Co., 1979.

[78] K. Arie, A. Barak, “Opportunity cost algorithms for reduction of i/o and interprocess communication overhead in a computing cluster,” *IEEE Transactions on Parallel and Distributed Systems,* vol. 14, no. 1, pp. 39-50, 2003.

[79] C. Ttofis, A. Papadopoulos, T. Theocharides, M. Michael, D.Doumenis, “An MPSoC-based QAM modulation architecture with Run-Time Load Balancing,” *EURASIP Journal of Embedded Systems,* vol. 2011, p. Article ID 790265, 2011.

[80] M.J. Flynn, “Some Computer Organizations and Their Effectiveness,” *IEEE Transactions on Computers,* Vols. C-21, no. 9, p. 948–960, 1972.

[81] R. Duncan, “A survey of parallel computer architectures,” *IEEE Computer,* vol. 23, no. 2, pp. 5-16, 1990.

[82] J. von Neumann, “First Draft of a Report on the EDVAC,” 1945.

[83] A. Silberschatz, P.B. Galvin, G. Gagne, A. Silberschatz, Operating system concepts, Addison-Wesley, 1998.

[84] G. E. Moore, “Cramming More Components onto Integrated Circuits,” *Proceedings of the IEEE,* vol. 86, no. 1, pp. 82-85, 2002.

[85] D. Geer, “Chip makers turn to multicore processors,” *Computer*, vol. 38, no. 5, pp. 11-13, 2005.

[86] J. Nickolls, “Nvidia GPU parallel computing architecture,” in *IEEE Hot Chips, IEEE Technical Committee on Microprocessors and Microcomputers*, Stanford, 2007.

[87] W. Wolf, FPGA-Based System Design, Prentice Hall, 2004.

[88] G. Chrysos, E. Sotiriades, C. Rousopoulos, K. Pramataris et al., “Reconfiguring the Bioinformatics Computational Spectrum: Challenges and Opportunities of FPGA-Based Bioinformatics,” *EEE Design and Test Magazine: Special Issue on Hardware Acceleration in Computational Biology,* vol. 31, no. 1, pp. 62-73, 2014.

[89] K.H. Tsoi, W. Luk, “Axel: A Hteregeneous Cluster with FPGAs and GPUs,” in *18nd ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2010),* 2010.

[90] M. Showerman, J. Enos, A. Pant et al., “QP: A Heterogeneous Multi-Accelerator Cluster,” in *10th LCI International Conference on High-Performance Clustered Computing*, 2009.

[91] R. Inta, D. J. Bowman, S. M. Scott, “The "Chimera": An Off-The-Shelf CPU/GPGPU/FPGA Hybrid Computing Platform,” *Hindawi International Journal of Reconfigurable Computing,* vol. 2012, p. Article No. 2, 2012.

[92] J.M. Elble et al., “GPU computing with Kaczmarz's and other iterative algorithms for linear systems,” *Parallel Computing,* vol. 36, no. 5-6, pp. 215-231, 2010.

[93] S. Manavski, G. Velle, “CUDA compatible GPU cards as efficient hardware acceleratos for

Smith-Waterman sequence alignment," *BMC Bioinformatics,* vol. 9 , no. Suppl.2, p. S10, 2008.

[94] Y. Liu, Adrianto Wirawan, Bertil Schmidt, "CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions," *BMC Bioinformatics,* vol. 2013, no. 14, p. 117, 2013.

[95] C.M.Liu et al., "SOAP3: Ultra-fast GPU-based parallel alignment tool for short reads," *Bioinformatics,* vol. 28, no. 6, pp. 878-879, 2012.

[96] M.C. Schatz, C. Trapnell, A.L Delcher, A. Varshney, "High-throughput sequence alignment using Graphics Processing Units," *BMC Bioinformatics,* vol. 8, no. 474, p. Online, 2007.

[97] A. Bustamam, K. Burrage, N.A. Hamilton, "Fast Parallel Markov Clustering in Bioinformatics Using Massively Parallel Computing on GPU with CUDA and ELLPACK-R Sparse Format," *IEEE/ACM Transactions of Computational Biology and Bioinformatics,* vol. 9, no. 3, pp. 679-692, 2012.

[98] A. Stamatakis et al., "RAxML Version 8: A tool for Phylogenetic Analysis and Post-Analysis of Large Phylogenies," *Bioinformatics,* vol. 2014, p. Online, 2014.

[99] L.S. Yung, C. Yang, X. Wan, W. Yu, "GBOOST: a GPU-based tool for detecting gene-gene interactions in genome-wide case control studies," *Bioinformatics,* vol. 27, no. 9, pp. 1309-1310, 2011.

[100] Y. Liu, B. Schmidt, D. L. Maskell, "CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows-Wheeler transform," *Bioinformatics,* vol. 28, no. 14, pp. 1830-1837, 2012.

[101] Y. Liu, B. Schmidt, W. Liu, D.L. Maskell, "CUDA-MEME: accelerating motif discovery in biological sequences using CUDA-enabled graphics processing units," *Pattern Recognition Letters,* vol. 31, no. 14, pp. 2170 - 2177, 2014.

[102] R. C. Edgar, "Search and clustering orders of magnitude faster than BLAST," *Bioinformatics,* vol. 26, no. 19, pp. 2460-1, 2010.

[103] M. Herbordt, J. Model, B. Sukhwani, Y. Gu and T. VanCourt, "Single pass streaming BLAST on FPGAs," *Parallel Computing Journal,* vol. 48, no. 3, pp. 189-208, 2007.

[104] E. Sotiriades and A. Dollas, "A General Reconfigurable Architecture for the BLAST Algorithm," *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology,* vol. 48, no. 3, pp. 189-208, 2007.

[105] S. Kasap, K. Benkrid, and Y. Liu, "A high performance fpga-based implementation of position specific iterated blast," in *ACM/SIGDA international symposium on Field programmable gate arrays - FPGA '09*, 2009.

[106] J. Garnier, J. Gibrat, B. Robson, "GOR method for predicting protein secondary structure from amino acid sequence," *Methods in Enzymology,* vol. 266, no. 1996, pp. 540-553, 1996.

[107] F. Xia, Y. Dou, G. Lei, and Y. Tan, "FPGA accelerator for protein secondary structure prediction based on the GOR algorithm," *BMC bioinformatics,* vol. 12 Suppl 1, no. 1, p. S5, 2011.

[108] P. Afratis, E. Sotiriades, G. Chrysos, S. Fytraki, and D. Pnevmatikatos, "A rate-based prefiltering approach to blast acceleration," in *International Conference on Field Programmable Logic and Applications, 2008*, 2008.

[109] OpenMP ARB Consortium, "The OpenMP® API specification for parallel programming," [Online]. Available: http://openmp.org/. [Accessed September 2014].

[110] N. Chapman, G. Jost, R. van der Pas, Using OpenMP Portable Shared Memory Parallel Programming, MIT Press, 2007.

[111] OpenMP ARB Consortium, "OpenMP Application Program Interface Specifications 4.0," OpenMP APB Consortium, 2013.

[112] S. Blair-Chappell, Intel Corporation, "Becoming a Parallel Programming Expert in Nine Minutes," in *Key Note at ACCU 2010 conference*, 2010.

[113] A. Karpov, "32 OpenMP traps for C++ developers," Intel Corporation, Developer Zone Article, 2008.

[114] Xilinx, 2014 September. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/pcie_blk_plus_ug341.pdf/.

[115] Xilinx, "LogiCORE IP FIFO User Guide," [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/fifo_generator_ug175.pdf. [Accessed Jan 2014].

[116] "BeeCube Hardware Emulation Platform," [Online]. Available: http://beecube.com. [Accessed September 2014].

[117] J. Carabano, F. Dios, M. Daneshtalab, M. Ebrahimi, "An Exploration of Heterogeneous Systems," in *8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, 2013.

[118] A. Hamann, M. Jersak, K.Richter, "A framework for modular analysis and exploration of heterogeneous embedded systems," *Real-Time Systems*, vol. 33, no. 1-3, pp. 101-137, 2006.

[119] B. da Silva, A. Braeken, E. H. D'Hollander, A. Touhafi, J. G. Cornelis, J. Lemeire, "Comparing and combining GPU and FPGA accelerators in an image processing context," in *23rd International Conference on Field Programmable Logic and Applications (FPL 2013)*, 2013.

[120] G.A. Petsko, D. Ringe, "From Sequence to Function: Case Studies in Structural and Functional Genomics," in *Protein Structure and Function*, New Science Press, 2004, pp. 130-162.

[121] "National Center for Biotechnology Information," [Online]. Available: http://www.ncbi.nlm.nih.gov/. [Accessed September 2014].

[122] "Dryad Digital Repository," [Online]. Available: http://datadryad.org/. [Accessed September 2014].

[123] IEEE, "802.3u-1995 - IEEE Standards for Local and Metropolitan Area Networks: Supplement to Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications Media Access Control (MAC) Parameters, Physical Layer,," IEEE Standards, 1995.

[124] IEEE, "IEEE 802.3 Ethernet - 2012," IEEE Standards.

[125] Compaq, HP, Intel,Microsoft, Lucent, NEC, Phillips, "Universal Serial Bus Specification Revision 2.0," 2000.

[126] HP, "USB 3.0 Technology," 2010.

[127] PCI-SIG, "PCIe Base Specification 2.0," 2007.

[128] PCI-SIG, "PCIe Base Specification 3.0," 2010.

[129] R. Bittner, "Speedy Bus Mastering PCI Express," in *22nd International Conference on Field Programmable Logic and Applications (FPL 2012)*, 2012.

[130] "IUPAC code table," [Online]. Available: http://www.dna.affrc.go.jp/misc/MPsrch/InfoIUPAC.html. [Accessed Jan 2014].

[131] PA Pevzner, HX Tang, MS Waterman, "An Eulerian path approach to DNA fragment assembly," *Proceedings of National Academy of Sciences,* no. 98, pp. 9748-9753, 2001.

[132] S.F. Mahmood, H. Rangwala, "GPU-Euler: Sequence Assembly Using GPGPU," in *13th IEEE International Conference on High Performance Computing and Communications (HPCC),* 2011.

[133] B. Awerbuch, A. Israeli, and Y. Shiloach, "Finding euler circuits in logarithmic parallel time," in *16th annual ACM symposium on Theory of computing,* 1984.

[134] D.S. Johnson, "A Catalog of Complexity Classes," in *Handbook of Theoretical Computer Science,* Amsterdam , Elsevier , 1998, pp. 67-150.

[135] J. Commins, C. Toft, M.A. Fares, "Computational biology methods and their application to the comparative genomics of endocellular symbiotic bacteria of insects," *Biological Procedures Online,* no. 11, pp. 52-78, 2009.

[136] JT Simpson, K Wong, SD Jackman et al., "AbySS: a parallel assembler for short read sequence data," *Genome Research,* vol. 6, no. 19, pp. 1117-1123, 2009.

[137] J. Butler, I. MacCallum, M. Kleber et al., "ALLPATHS: de novo assembly of whole-genome stogun microreads," *Genome Research,* vol. 5, no. 18, pp. 810-820, 2008.

[138] RL Warren, GG Sutton, SJM Jones et al., "Assembling millions of short DNA sequences using SSAKE," *Bioinformatics,* vol. 4, no. 23, pp. 500-501, 2007.

[139] RQ Li, HM Zhu, J Ruan et al., "De novo assembly of human genomes with massively parallel short read sequencing," *Genome Research,* vol. 2010, no. 20, pp. 265-272, 2010.

[140] D Hernadez, P Francois, L Farinelli et al., "De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer," *Genome Research,* vol. 2008, no. 18, pp. 802-804, 2008.

[141] WR Jeck, JA Reinhardt, DA Baltrus et al., "Extending assembly of short DNA sequences to handle error," *Bioinformatics,* vol. 23, no. 21, pp. 2942-2944, 2007.

[142] D. D'Agostino, I. Merelli et al., "Parallelization of the SSAKE Genomics Application," in *19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP '11),* 2011.

[143] N. Joshi, S. Shekhar Srivastava, M. Milner Kuma et al., "Parallelization of Velvet,"a de novo genome sequence assembler"," in *IEEE International Conference on High Performance Computing (HiPC),* 2011.

[144] X. Liu, P.R. Pande, H. Meyerhenke, D.A. Bader, "PASQUAL: Parallel techniques for next generation genome sequence assembly," *IEEE Transactions on Parallel and Distributed Systems,* vol. 24, no. 5, pp. 977-986, 2013.

[145] P.N. Ariyaratne, W-K. Sung, "PE-Assembler: de novo assembler using short paired-end reads," *Bioinformatics,* vol. 2010, no. 27, pp. 167-174, 2010.

[146] D.W. Bryant Jr., W.K. Wong, T.C. Mockler, "QSRA: a wuality-value guided de novo short read assembler," *BMC Bioinformatics,* vol. 10, no. 69, p. Online, 2009.

[147] JC Dohm, C Lottaz, T Borodina et al., "SHARCGS, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing," *Genome Research,* vol. 2007, no. 17, pp. 1697-1706, 2007.

[148] E.W Myers, “Towards simplifying and accurately formulating fragment assembly,” *Journal of Computational Biology*, vol. 2, no. 2, pp. 275-290, 1995.

[149] D.R. Zerbino, E. Birney, “Velvet: algortihms for de novo short read assembly using de Bruijin graphs,” *Genome Research*, vol. 18, no. 5, pp. 821-829, 2008.

[150] R.Q. Li, YR Li, K. Kristiansen et al., “SOAP: short oligonucleotide alignment program,” *Bioinformatics*, vol. 24, no. 5, pp. 713-714, 2008.

[151] W. Tang, W. Wang, et al., “Accelerating Millions of Short Reads Mapping on a Heterogeneous Architecture with FPGA Accelerator,” in *IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM2012)*, 2012.

[152] D. D'Agostino, A. Clematis et al., “A CUDA-based Implementation of the SSAKE Genomics Application,” in *20th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP '12)*, Washington,DC, 2012.

Appendix A:
Proteomic Benchmarks

To compare the proposed implementations, we used a number of datasets stemming from four actual protein sequence databases [121].

The first database belongs to *Haemophilus influenza* bacteria (*haem database*) and the second consists of protein sequences from the malaria parasite genome *Plasmodium falciparum* (p.falciparum database).

The third and fourth databases are a reference viral database and a unified protein database from NCBI [121] used for benchmarking bioinformatics applications (*viral.1.protein, uniprot.sprot*). The datasets used in the evaluation of the proposed system can be download from [122].

Each dataset has been selected for its unique characteristics regarding LCRs in order to adequately evaluate the performance of the proposed implementations under various real-life scenarios. *Table A.1* lists all the test databases used during evaluation process.

TABLE 9: PROTEOMIC DATASETS USED FOR EVALUATING CAST IMPLEMENTATIONS

Dataset Name	# of sequences	Average length	% LCRs*	Details
Case.I	1	1695	100 (12.51)	Longest sequence of *haem* database
Case.II	236	311.6	7.62 (0.59)	Random sequences from *haem* database
Case.III	478	299.2	8.57 (0.69)	Random sequences from *haem* database
Case.IV	258	292.3	7.75 (0.58)	Random sequences from *haem* database
Case.V	113	764	23.89 (1.28)	Sequences longer than 600 from *haem* database
Haem	1,743	305.1	7.97 (0.64)	All sequences from *haem* database
plasma.falciparum (plasma)	5,491	755.9	72.81 (14.42)	All sequences from *p.falciparum* database
Viral.1.Protein	101,537	274.4	16.56 (2.05)	All sequences from *viral.1.protein* reference database
Uniprot.sprot	537,593	354.9	33.80 (0.16)	All sequences from *uniprot.sprot* reference database

* % of sequences having at least one LCR (% of residues masked by CAST)

Appendix B:
Communication Schemes Evaluation

There are a number of available standardized high-speed communication protocols on off-the-shelf motherboards and accelerator cards which can allow data to be transferred between the main CPU-controlled memory and the peripheral's memory present on commercial GPU cards and FPGA boards.

Options include: Gigabit Ethernet, USB and PCI-Express; each with its advantages and disadvantages. Each available communication technology's characteristics, benefits and issues for using it to transfer data to each computing platform are reviewed in the tables below.

TABLE 10: ETHERNET TECHNOLOGY OVERVIEW

Communication Technology		
100Base-X [123] / Gigabit Ethernet [124]		
Standardized	Max Speed	
Yes - IEEE 802.3 / IEEE 802.3-2008	100Mbps / 1Gbps	
Platforms		
CPU	GPU	FPGA
Standard peripheral on motherboards Typically connected via the Southbridge chip	Not supported directly Connected indirectly via the CPU and main memory	Most of the commercially available FPGA boards support direct connection

TABLE 11: USB TECHNOLOGY OVERVIEW

Communication Technology		
USB2.0 [125] / USB3.0 [126]		
Standardized	Max Speed	
Yes	480Mbps / 4Gpbs	
Platforms		
CPU	GPU	FPGA
Standard peripheral on motherboards Typically connected via the Southbridge chip	Not supported directly Connected indirectly via the CPU and main memory	Some FPGA boards support USB connections Not widely used No reference designs or sources

TABLE 12: PCI-EXPRESS TECHNOLOGY OVERVIEW

Communication Technology		
PCI - Express [127][128]		
Standardized	Max Speed	
Yes	500Mbps per lane (v2.x) Max available: 32 lanes → 16Gbps 985Mbps per lane (v3.x) Max available: 32 lanes → 31Gbps	
Platforms		
CPU	GPU	FPGA
Standard communication protocol for connecting to off-dice devices (excluding RAM)	Default communication protocol Typically connected via the Northbridge chip	Most FPGA boards support PCI-Express connections Number of lanes typically differ depending on each board's price range

PCI-Express is the communication scheme of choice on almost all available hybrid systems since allows for accelerators to be integrated on commercially available motherboards without the need for extra hardware. Truth is that PCI-Express communication is cumbersome since each computation engine (CPU, GPU and FPGA) has its own drivers and rules for communication.

For instance, GPU cards must always be the bus masters in any transfer it is involved. GPU/CPU communication and its setup can be handled directly by the related API (such as CUDA) as it sets up and allocates both the CPU-controlled main-memory and the needed GPU memory space and makes the data transfer by executing specialized system and function calls. FPGA/CPU communication can be achieved using already available FPGA-based PCI-Express modules. There is a need however, for drivers (open source drivers are available) that allow the OS running on the CPU to recognize the FPGA as a PCI-Express DMA device and initialize the data transfer with the FPGA as bus master.

Most hybrid systems in literatue use indirect PCI-Express communication to transfer data between the GPU and the FPGA as both are needed to be bus masters. As such, the data is transferred using the CPU main-memory as an intermediate. This fact hinders performance of FPGA/GPU applications, as unnecessary memory transfer overheads are introduced. Microsoft Research's Speedy PCI-Express FPGA core [129] is the first attempt to implement direct GPU/FPGA communication by modifying a FPGA core to act as PCI-Express bus slave and then use already available CUDA commands to allow the FPGA's memory to be page-locked as virtual memory by the GPU card. This approach allows direct data transfer between the FPGA and the GPU memory using standard CUDA memory transfer commands.