Exploiting Multigrain Parallelism in Pairwise Sequence Search on Emergent CMP Architectures

Ashwin Mandayam Aji

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Wu-chun Feng, Chair
Dimitrios S. Nikolopoulos, Committee member
Kirk W. Cameron, Committee Member

30 May, 2008
Blacksburg, Virginia

Keywords: Bioinformatics, Smith-Waterman, Cell Broadband Engine, GPGPU

Exploiting Multigrain Parallelism in Pairwise Sequence Search on Emergent CMP Architectures

Ashwin Mandayam Aji

(ABSTRACT)

With the emerging hybrid multi-core and many-core compute platforms delivering unprecedented high performance within a single chip, and making rapid strides toward the commodity processor market, they are widely expected to replace the multi-core processors in the existing High-Performance Computing (HPC) infrastructures, such as large scale clusters, grids and supercomputers. On the other hand in the realm of bioinformatics, the size of genomic databases is doubling every 12 months, and hence the need for novel approaches to parallelize sequence search algorithms has become increasingly important. This thesis puts a significant step forward in bridging the gap between software and hardware by presenting an efficient and scalable model to accelerate one of the popular sequence alignment algorithms by exploiting multigrain parallelism that is exposed by the emerging multiprocessor architectures. Specifically, we parallelize a dynamic programming algorithm called Smith-Waterman both within and across multiple Cell Broadband Engines and within an nVIDIA GeForce General Purpose Graphics Processing Unit (GPGPU).

Cell Broadband Engine: We parallelize the Smith-Waterman algorithm *within* a Cell node by performing a blocked data decomposition of the dynamic programming matrix followed by pipelined execution of the blocks across the synergistic processing elements (SPEs) of the Cell. We also introduce novel optimization methods that completely utilize the vector processing power of the SPE. As a result, we achieve near-linear scalability or near-constant efficiency for up to 16 SPEs on the dual-Cell QS20 blades, and our design is highly scalable to more cores, if available. We further extend this design to accelerate the Smith-Waterman algorithm *across* nodes on both the IBM QS20 and the PlayStation3 Cell cluster platforms and achieve a maximum speedup of 44, when compared to the execution times on a single Cell node. We then introduce an analytical model to accurately estimate the execution times of parallel sequence alignments and wavefront algorithms in general on the Cell cluster platforms. Lastly, we contribute and evaluate TOSS – a Throughput-Oriented Sequence Scheduler, which leverages the performance prediction model and *dynamically* partitions the available processing elements to simultaneously align multiple sequences. This scheme succeeds in aligning more sequences per unit time with an improvement of 33.5% over the naïve first-come, first-serve (FCFS) scheduler.

nVIDIA GPGPU: We parallelize the Smith-Waterman algorithm on the GPGPU by optimizing the code in stages, which include optimal data layout strategies, coalesced memory accesses and blocked data decomposition techniques. Results show that our methods provide a maximum speedup of 3.6 on the nVIDIA GPGPU when compared to the performance of the naïve implementation of Smith-Waterman.

Dedication

I dedicate my thesis to my parents, brother, sister and family for their unconditional love, support and encouragement to help me obtain a post-graduate degree in the field of my choice.

Acknowledgments

I would like to express my heart-felt gratitude to the following people, (in no particular order) without whom this thesis would not have been possible:

- Dr. Wu-chun Feng, my advisor, who always had faith in me, and supported and encouraged me at the right times even before I landed in Blacksburg. I have learnt a lot of valuable lessons from him during my stay here at Virginia Tech.
- Dr. Dimitris Nikolopoulos, who provided excellent guidance for our Cell-SWat publication and his great teaching skills motivated me to take up his Parallel Computation courses.
- Dr. Kirk Cameron, for his valuable inputs, time and patience.
- Dr. Yong Cao and his team, for their valuable suggestions related to GPGPU programming.
- Dr. Eli Tilevich, for his funny comments on life, in general, and they proved to be great stress relievers during the long nights of research at the CRC.
- Filip Blagojevic, a very good friend and colleague, for his unmatched patience with all of us who seeked guidance from the *Cell Guru*.
- Ganesh, Jeremy, Tom, Song and Ajeet for making an excellent 'SyNeRGistic' team.
- Jaishankar, Lokeya, Ajith, Sanket, Lakshman, Shvetha, Sneha and Anusha for all the fun times in Blacksburg.
- Ashok Holla, a great friend, who motivated me to pursue post-graduation studies.
- Tilottama Gaat, who has been of immense support to me at all times and whose mere presence fills me with a unique sense of completeness.

Thank you, one and all.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Until recently, computational horsepower doubled every 18-24 months simply by increasing the clock speed of a processor. However, that era is now over with processor clock speeds having stalled out in the sub-4.0 GHz range. Instead, while computational horsepower continues to double, it does so via a doubling of the number of cores per processor in both multi-core and many-core architectures.

Commodity processors that are based on replicating scalar cores are arguably plagued by limitations in performance and power consumption. Consequently, this has led to a bit of unconventional thinking on the part of Sony, Toshiba, and IBM with their Cell Broadband Engine (BE), and nVIDIA with their GeForce General Purpose Graphics Programming Unit (GPGPU) cards. The Cell B.E contains heterogeneous cores and specialized accelerators on the same chip and drives the Sony PlayStation 3 game console, while the present day GPGPUs comprise of simplistic, yet several tens of accelerator cores inside one chip to

tremendously enhance the performance of graphics-based applications.

The new pieces of computing hardware boast of extremely impressive aggregate single-precision floating-point performances, thus providing the necessary computational horse-power for scientific computing. However, the integration of accelerators with more conventional parallel programming paradigms and tools is currently an active research area [5, 13, 21, 9]. The emergent unconventional architectures many times require platform-specific programming interfaces and complete redesigning of existing algorithms to fully exploit the hardware. Furthermore, scheduling code on accelerators and achieving efficient parallel execution and data transfers between host processors and accelerators is a challenging problem [13].

Among scientific applications related to bioinformatics, sequence-search algorithms are used extensively in a wide range of areas from estimating evolutionary histories to predicting the behavior of newly found genes to identifying possible drugs to curing prevalent diseases. As of April 2005, the NCBI BLAST server received about $400,000$ queries per day to search against their massive genome databases [39]. This trend has been doubling every 12-18 months and would potentially double at an even faster rate if NCBI had the capacity to support it. However, the exponential growth in the nucleotide and protein databases has made the optimal sequence search algorithms, such as the Needleman-Wunsch [37] and Smith-Waterman algorithms [42], impractical to search on these databases because of their quadratic time and space complexity.

As a result, this led to the development of algorithmic heuristics such as FASTA [31] and the BLAST [3] family of algorithms that sacrificed sensitivity for speed, i.e, they are much faster but missed some fraction of good sequence homologies that the optimal algorithms would have found. More recent innovations have led to heuristic algorithms that attempt to bring the sensitivity of sequence alignment as close to Smith-Waterman as possible while

achieving a more reasonable time and space complexity [33, 30].

The time and space complexity of Smith-Waterman also led to innovations on the non-algorithmic front — specifically, special-purpose hardware solutions on FPGAs [46, 44, 29] and linear processor arrays [27]. However, such solutions are quite expensive.

In summary, to address the ever-increasing need to more quickly search biological databases that are doubling in size every 9-12 months, and hence, growing at a rate faster than we can compute on them with a single compute node, computational scientists have proposed a plethora of faster homology sequence searches *but* at the expense of using heuristics that reduce the sensitivity of the searches or using expensive hardware to produce ideal sensitivity. In this thesis, we attempt to achieve high speeds while retaining ideal sensitivity by choosing to parallelize the Smith-Waterman algorithm on the emerging (and arguably, commodity) chip multi-processors like the Cell Broadband Engine and the nVIDIA GeForce GPGPU.

In addition, parallelizing the Smith-Waterman algorithm is significant because it follows the dynamic programming paradigm, which is one of the 13 Dwarfs[1] [5] of parallel programming. Thus, a thorough understanding of how this application maps onto the existing novel parallel computing platforms provides solid insights to design and evaluate future parallel architectures and programming models.

In this thesis, we first present highly efficient methodologies to parallelize the Smith-Waterman pairwise sequence alignment algorithm within a Cell chip [1], where we achieve near-constant efficiency for up to 16 SPEs on the dual-Cell QS20 blades, and our approach is highly scalable to more cores, if available. However, by using only one Cell processor for aligning a pair of sequences, we limit the problem space to aligning sequences smaller than 8KB on the QS20 Cell blade and smaller than 3.5KB on a PS3 console, due to the inherent memory constraints of the algorithm. This prevents about 200,000 large-sized sequences from the

[1] A dwarf is an algorithmic method that captures a pattern of computation and communication.

NT nucleotide database from being aligned, and NT is just one among the many existing sequence databases. To alleviate the problem of aligning arbitrarily long sequences, coupled with the ever-increasing need to accelerate sequence search, we introduce and describe a scalable model that uses multiple nodes in a cluster of Cell processors to align a single pair of nucleotide or protein sequences. By simultaneously exploiting multiple layers and granularities of parallelism within the cluster, we achieve maximum speedups of 44× on the IBM QS20 and 26× on the PS3 Cell cluster platforms, where the base measurement for speedup is the execution time recorded on a single PPE-SPE combination. We show that our design for executing parallel wavefront computations *within* a Cell node serves as a remarkably generic design template, which can thus be recursively applied to every layer of parallelism in the Cell cluster. We also present an analytical model to accurately estimate the execution times of parallel sequence alignments within and across multiple Cell nodes. We achieve an error rate of less than 3% for sequence alignments within a Cell node and error rates of less than 10% for alignments across Cell nodes on an average.

We then contribute and evaluate TOSS – a Throughput-Oriented Sequence Scheduler, which follows the greedy paradigm and spatially distributes the available computing resources *dynamically* among simultaneous sequence alignments to achieve better throughput in sequence search. We use the data generated by the analytical model in deciding the optimal input configuration for TOSS and achieve an improvement of 33.5% on the QS20 Cell cluster and about 13.5% on the PS3 cluster over the naïve FCFS scheduler. Thus, we can align arbitrarily large-sized sequences at high speeds by simultaneously exploiting the multigrain parallelism of the Cell clusters.

Lastly, we parallelize the Smith-Waterman on the CUDA platform of the nVIDIA GPGPU by incrementally optimizing the code in five stages, including optimal data layout strategies, coalesced memory accesses and blocked data decomposition techniques to provide an efficient

mapping of the algorithm to the data parallel architecture of the GPGPU. We show that our optimizations yield a speedup of 3.6× over the serial implementation of Smith-Waterman.

Thus, we explore various layers of parallelism – across nodes, within nodes and within cores, which are exposed by the Cell B.E and the nVIDIA GPGPU, and we efficiently parallelize the optimal pairwise sequence alignment algorithm on each of them. Our design methodology, combined with the experimental results, help in providing insights into developing future programming models that help in porting existing legacy codes by effectively utilizing the potential of emergent CMP architectures to the fullest.

1.2 Related Work

Numerous recent research efforts have explored application development, optimization methodologies and new programming environments for the Cell B.E. In particular, recent studies investigate Cell versions of applications including particle transport codes [40], numerical kernels [2], FFT [7], irregular graph algorithms [8], computational biology [41], sorting [23], query processing [24], and data mining [15]. Our research departs from these earlier studies in that it models and optimizes a parallel algorithmic pattern that is yet to be explored thoroughly on the Cell B.E, namely wavefront algorithms, which is a generalization of the Smith-Waterman algorithm. The work closest to our research is a recent parallelization and optimization of the wavefront algorithm used in a popular ASCI particle transport application, SWEEP3D [40], on the Cell B.E. Our contribution differs in three aspects. First, we consider inter-tile parallelism during the execution of a wavefront across the Cell SPEs, to cope with variable granularity and degree of parallelism within and across tiles or blocks. Second, we provide an analytical model for tiled wavefront algorithms to guide the parallelization, granularity selection, and scheduling process for both single and multiple wave-front

computations executing simultaneously. Third, we consider throughput-oriented execution of multiple wavefront computations on the Cell B.E, which is the common usage scenario of these algorithms in the domain of computational biology.

Our research also parallels efforts for porting and optimizing key computational biology algorithms, such as phylogenic tree construction [11] and sequence alignment [41]. The work of Sachdeva et. al [41] relates to ours, as it explores the same algorithm (Smith-Waterman), albeit in the context of vectorization for SIMD-enabled accelerators. We present a significantly extended implementation of Smith-Waterman that exploits pipelining across multiple accelerators in conjunction with vectorization and optimizes task granularity and multiple query execution throughput on the Cell. We also extend this work through a generic model of wavefront calculations on the Cell B.E, which can be applied to a wide range of applications using dynamic programming for both performance-oriented and throughput-oriented optimization.

Recently proposed programming environments (languages and runtime systems) such as Sequoia [21], Cell SuperScalar [9], CorePy [36] and PPE-SPE code generators from single-source modules [19, 45], address the problem of achieving high performance with reduced programming effort. Our work is oriented towards simplifying the effort to achieve high performance from a specific algorithmic pattern on the Cell B.E, and is orthogonal to related work on programming models and interfaces. An interesting topic for future exploration is the expression of wavefront algorithms with high-level language constructs, such as those provided by Sequoia and CellSs, and techniques for automatic optimization of key algorithmic parameters in the compilation environment of high-level parallel programming languages.

Efforts to parallelize the Smith-Waterman algorithm across a cluster of workstations to address the speed and memory problems have exploited only a single layer of parallelism at the node level [14, 48]. We provide a comprehensive methodology to extract the maximum

potential out of each layer of parallelism in the multiple levels that are available on the Cell cluster.

While the contribution of Filip et. al [12] is the only other work that models a cluster of Cell nodes by exploiting the inherent multiple layers of parallelism, our work is specific to modeling wavefront algorithms and the Smith-Waterman algorithm in particular, thereby providing a higher prediction accuracy of the execution times for the algorithms in the domain of computational biology. To the best of our knowledge, no other work exploits multigrained parallelism to effectively schedule sequence alignments dynamically to achieve large throughputs.

Smith-Waterman has previously been implemented on the GPGPU by using graphics primitives [47, 32], and recently on the CUDA platform [34]. While the older implementations that use graphics primitives report good speedups over the serial implementations, they are now obsolete and we do not learn much from them in terms of developing general programming models on the latest GPGPUs, which mostly use regular C-style libraries. The CUDA implementation of Smith-Waterman too reports impressive speedup values, but suffers from the following limitations – it follows a coarse-grained parallelization approach of assigning a single sequence alignment to each thread on the device, and this severely restricts the maximum sequence size that can be aligned. Moreover, it is not clear if only the alignment score is computed or the actual alignment is generated. Our implementation follows fine-grained parallelization by distributing the task of aligning a single sequence among all the threads on the GPGPU, followed by actually generating the optimal sequence alignment.

1.3 Document Overview

The rest of this thesis is organized as follows: Chapter 2 introduces the sequence alignment concepts and popular sequence search algorithms that are relevant to our study. Chapter 3 describes the architecture and programming model employed in the emergent chip multiprocessors – in particular the Cell B.E and the nVIDIA GeForce GPGPU. Chapter 4 introduces and elaborates our strategies to exploit multigrain parallelism on the emergent CMP architectures to enhance the performance of pairwise sequence search. Chapter 5 concludes the thesis.

Chapter 2

Pairwise Sequence Search

2.1 Sequence Alignment Basics

A biological sequence is a succession of letters representing the structure of a real or hypothetical DNA or protein molecule or strand. The alphabet set representing the sequence is different for the nucleotides and the proteins. In this thesis, we have based our studies on aligning DNA sequences that comprises of the alphabet set {A, C, T, G}, but the same underlying principles can be applied to protein alignment as well. Sequences can be derived from the biological raw material through a variety of *sequencing* methods.

Sequence alignment is a process of arranging a group of nucleotide or protein sequences to determine similar regions, so that useful insights into the functional, structural, or evolutionary relationships between the sequences can be provided. In this thesis, we focus on *pairwise* sequence alignment rather than *multiple* sequence alignment, i.e. our focus is on aligning a single pair of sequences.

The process of aligning entire sequences against each other is called as *global alignment*. This

Table 2.1: Global Alignment Vs. Local Alignment.

Original Sequences	GTGTACGCCATAT GTACCCAAT
Global Alignment	GTGTACGCCATAT | ||| ||| || G__TAC_CCA_AT
Local Alignment	GTGTACGCC_ATAT |||| || || GTAC_CCAAT

approach is useful if the sequences are closely related to each other and most of the residues (a single character in a sequence) match against each other. If the sequences are loosely related to each other, it might be more interesting to find smaller regions of similarity within the sequences, and this process is called *local alignment*.

Table 2.1 shows examples of local and global alignments. If the sequences are sufficiently similar to each other, there is no significant difference between local and global alignments. Each alignment is quantified with an alignment *score*, and the aim of any sequence alignment algorithm should be to optimize the alignment so that the score is maximized. The score of an alignment is calculated by using a scoring system, that typically comprises of two subsystems:

- *The substitution matrix, M:* Each entry in the substitution matrix, $M(i, j)$, indicates the score of aligning the characters i and j. If $M(i, j)$ is positive, then there is a match between i and j, and the score is referred to as a reward. A higher positive score indicates a better match. If $M(i, j)$ is negative, then it is a mismatch between i and j, and the score is a penalty. PAM [18] and BLOSUM [25] substitution matrices are typically used for protein alignment, while DNA alignment typically uses a single match-score and a single mismatch-score.

- *The gap-scoring scheme:* Gaps are introduced between the amino acid or nucleotide residues so that similar characters get aligned to potentially increase the alignment score. These gaps are usually denoted by a '–' in the output alignment. They are considered to be a type of a mismatch and incur some penalty. We consider the system with affine gap penalties which means that there are two types of gap penalties:

  1. Gap-open penalty (o): This is the penalty for starting (or opening) a gap in the alignment
  2. Gap-extension penalty (e): This is usually a less severe penalty than the gap-open penalty. It is imposed for extending a previously existing gap in the alignment by one unit.

  Thus, if there are k consecutive gaps in an alignment, then the total gap penalty incurred by that gap is $o + k \times e$.

2.2 Optimal Sequence Alignment Algorithms: Needleman-Wunsch and Smith-Waterman

Several sequence alignment algorithms have been proposed in the past, among which we discuss the optimal algorithms in this section, and we briefly introduce the faster, but less accurate heuristic algorithms in the next section.

The *dynamic programming* algorithmic paradigm is used to produce optimal global alignments via the Needleman-Wunsch algorithm [37], and optimal local alignments via the Smith-Waterman algorithm [42]. Due to the high similarity in the execution pattern of the above two algorithms, we explain their core strategy in a combined fashion, while deviating to discuss the minor differences as and when necessary. The algorithms can be partitioned into

two phases: (1) matrix filling – to find the optimal alignment score and (2) backtracing – to generate the optimal alignment.

- *Matrix filling:* The optimal alignment score is computed by filling out a dynamic programming matrix, starting from the northwest corner and moving towards the southeast corner, following the wavefront pattern. The matrix is filled based on the scoring system, as discussed previously. The recursive data dependence of the elements within the dynamic-programming matrix can be explained by the following equations:

$$DP_N[i,j] = e + max \begin{cases} DP_N[i-1,j] \\ DP_W[i-1,j] + o \\ DP_{NW}[i-1,j] + o \end{cases} \tag{2.1}$$

$$DP_W[i,j] = e + max \begin{cases} DP_N[i,j-1] + o \\ DP_W[i,j-1] \\ DP_{NW}[i,j-1] + o \end{cases} \tag{2.2}$$

$$DP_{NW}[i,j] = M(X_i, Y_j) + max \begin{cases} DP_N[i-1,j-1] \\ DP_W[i-1,j-1] \\ DP_{NW}[i-1,j-1] \end{cases} \tag{2.3}$$

Equations (2.1), (2.2), and (2.3) indicate the presence of three weighted matrices and also imply a three-dimensional (3D) dependency among the elements of the matrix as shown in Figure 2.1.

The elements of the matrix DP_N are dependent only on the northern neighbors of the three available weighted matrices. Similarly, DP_W and DP_{NW} have elements that depend only on their respective western and northwestern neighbors of the available

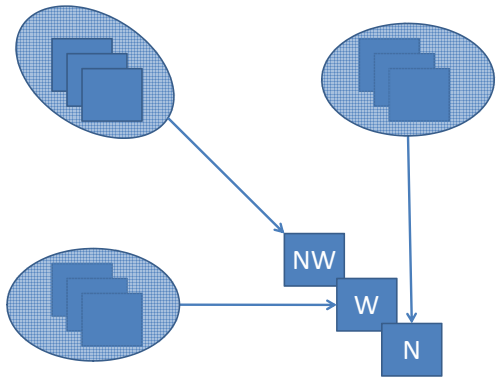


Figure 2.1: Optimal Sequence Alignment: 3D dependencies

three weighted matrices, thereby maintaining the wavefront pattern.

The maximum value in the matrix denotes the optimal local alignment score, while the element at the southeast corner of the matrix (not required to be the maximum) indicates the optimal global alignment score.

- *Backtracing:* This stage of the algorithm yields the highest scoring (local or global) alignment. The backtrace begins at the matrix cell that holds the optimal alignment score and proceeds in a direction opposite to that of the matrix filling, until a cell that satisfies the terminating condition is encountered. Backtrace for global alignment terminates only at the northwest cell of the matrix, while backtrace for local alignment stops when a cell with score zero is encountered. In both the cases, the path traced by this operation generates the optimal alignment.

Backtracing sequential post-processing operation and requires the entire matrix to be stored in memory before-hand. Based on this background, we define *robustness* of an alignment in the following way – if the algorithm performs both the phases (matrix filling and backtrace)

of a sequence alignment, then the alignment it termed to be *robust*. The memory requirement of a robust alignment is more severe than that of a non-robust alignment.

2.3 Heuristic Algorithms

While the dynamic programming methods generate the optimal sequence alignments, they suffer from quadratic space and time complexity. This has rendered the above algorithms useless in practical scenarios where query sequences are regularly searched against ever-expanding sequence databases. Many approximation algorithms have thus been developed to work around the sequence search problem.

2.3.1 BLAST: Basic Local Alignment Search Tool

The original BLAST algorithm [3] performs three basic steps:

1. In the first step, BLAST searches for exact matches or *seeds* of a small fixed length W between the query sequence and every sequence in the database. By default, $W = 11$ is used for the initial seeds when aligning DNA sequences.

2. In the second step, BLAST performs an ungapped alignment (alignment that does not consider gaps) by extending the seed in both directions until the score drops beyond a pre-determined threshold score. If a high-scoring ungapped alignment is found, the algorithm passes the query-database sequence pair to the third step.

3. In the third step, BLAST performs a gapped alignment between the query and the database sequence using a variation of the Smith-Waterman algorithm. Statistically significant alignments are then returned as output.

Although BLAST gains over its predecessors relative to speed, it sacrifices on sensitivity (defined in Appendix A), i.e. BLAST misses onfinding several optimal sequence alignments that the ideal Smith-Waterman algorithm would havefound [6]. Several variations of the original BLASTalgorithm have emerged to improve the sensitivity of the sequencesearch while maintaining the high speed [28, 4, 10, 16].

2.3.2 FASTA

FASTA [31] is a local sequence alignment software package that differs from BLAST in the algorithmic front but provides the same result, i.e. aligns DNA or protein sequences quickly by compromising the sensitivity of the sequence search. The FASTA algorithm initially observes the pattern of word-to-word matches (or *identities*) of a given length, and marks potential matches before performing a more time-consuming optimized dynamic programming based algorithm like Smith-Waterman. This approach ensures that the sequence database has been substantially pruned to get rid of irrelevant sequences, and finds more practical significance in searching extremely large sequence databases. The length of the identity controls the sensitivity and speed of the program. The FASTA algorithm can be divided into four stages:

1. Identify the identities in each sequence comparison.

2. Recalculate the scores of the regions by using the scoring matrices. Trim the ends of the region to include only the parts that contribute to the highest score.

3. In an alignment if several initial regions with scores greater than a *threshold* value are found, check whether the trimmed initial regions can be joined to form an approximate alignment with gaps.

4. Use the ideal Smith-Waterman algorithm to calculate an optimal score for the chosen

alignment.

2.3.3 PatternHunter

The PatternHunter [33, 30] family of algorithms aim to resolve the sensitivity issue of BLAST while maintaining its high speed. While BLAST initially looks for k consecutive residue matches as seeds, the PatternHunter algorithm uses *non-consecutive* k letter matches as seeds. This 'spaced-seed model' is represented as a 0-1 string, where ones in the model indicate a required match, while zeroes indicate 'dont-care' positions.

The authors of PatternHunter extended their idea of spaced seeds to use *multiple* spaced seed models and created PatternHunterII [30]. In such an approach, a set of several seed models are selected first. Then all the hits produced by all the seed models are examined to produce local alignments. This obviously increases the sensitivity because more hits are generated than by using single seed models. In their paper, the authors have also described methodologies to generate the optimal spaced seeds that generate highly sensitive local alignments.

The software solutions to the sequence search problem presented thus far have tried to speedup the sequence alignment process by trading-off sensitivity. However, in this thesis, we choose to improve the performance of the Smith-Waterman algorithm [1] on emergent hardware accelerators, such as the Cell Broadband Engine and the GPGPU, so that ideal sensitivity is maintained.

[1]The speedup strategy for the Smith-Waterman local alignment algorithm is also applicable to the Needleman-Wunsch global alignment algorithm

Chapter 3

Emergent CMP Architectures

Until recently, computational horsepower doubled every 18-24 months simply by increasing the clock speed of a processor. However, that era is now over with processor clock speeds having stalled out in the sub-4.0 GHz range. Instead, while computational horsepower continues to double, it does so via a doubling of the number of cores per processor in both multi-core and many-core architectures.

Commodity processors that are based on replicating scalar cores are arguably plagued by limitations in performance and power consumption. Consequently, this has led to a bit of unconventional thinking on the part of Sony, Toshiba, and IBM with their Cell Broadband Engine (BE), which contains heterogeneous cores and specialized accelerators on the same chip and drives the Sony PlayStation 3 game console[1]. The Cell BE possesses an aggregate single-precision floating-point performance of 204.8 Gflops, thus providing the necessary computational horsepower for scientific computing such as the pairwise sequence searching found in Smith-Waterman.

Until recently, Graphics Processing Units (GPUs) were used mainly for processing images

[1]This computer is a game console that currently costs a mere $399.

and geometric information at high speeds. The steadily increasing demand for advanced graphics within common software applications has made high-performance graphics systems ubiquitous and affordable. Now that GPUs have evolved into fully programmable devices and key architectural limitations have been eliminated, they have become an ideal resource for acceleration of many arithmetic and memory bandwidth intensive scientific applications [43]. In this chapter, we briefly describe the architecture of the two emergent Chip-Multiprocessors (CMP) – the Cell Broadband Engine and the GPGPU belonging to the NVIDIA GeForce family, along with the programming model that we employ to execute the Smith-Waterman algorithm on them.

3.1 Cell Broadband Engine (B.E)

3.1.1 Architecture

Figure 3.1 illustrates the various modules within the Cell Broadband Engine. The Cell is a heterogeneous processor which integrates a total of 9 cores: a two-way SMT PowerPC core (the Power Processing Element or PPE), and 8 tightly coupled SIMD-based processors (the Synergistic Processing Elements SPEs) [20]. The components of the Cell processor are connected via a high bandwidth Element Interconnect Bus (EIB). The EIB is a 4-ring structure, capable of transmitting 96 bytes per cycle, for a maximum theoretical memory bandwidth of 204.8 Gigabytes/second. The EIB can support more than 100 outstanding DMA requests.

The PPE is a 64-bit SMT processor running the PowerPC ISA, with vector/SIMD multimedia extensions. The PPE boasts two levels of on-chip cache, L1-I and L1-D with a capacity of 32 KB each, and L2 with a capacity of 512 KB. Each SPE has two main components,

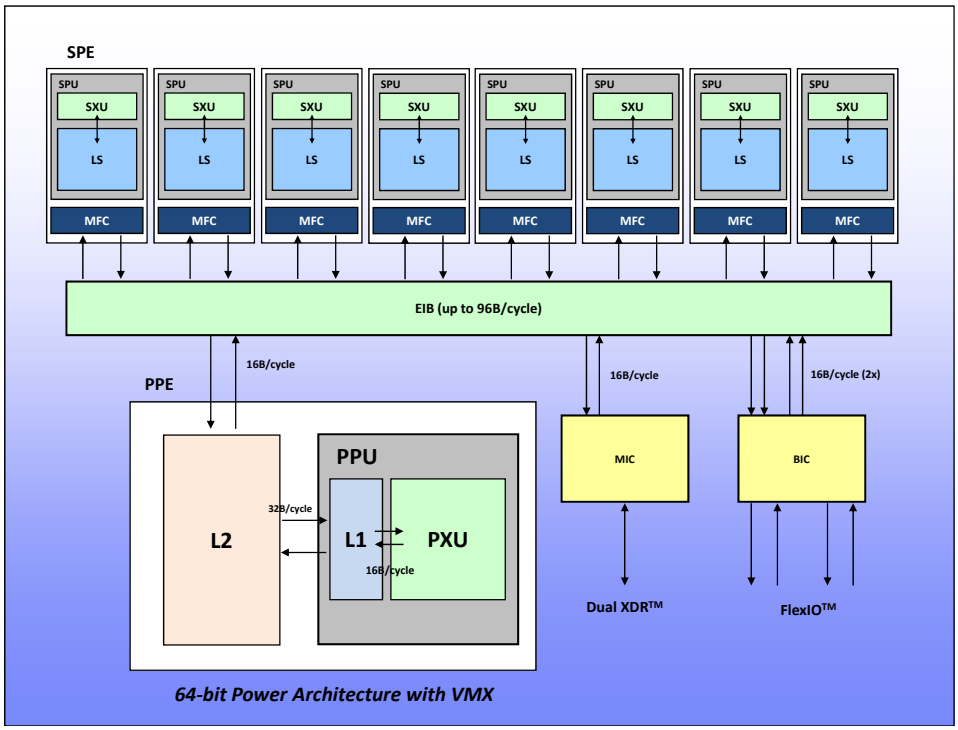


Figure 3.1: The Cell Broadband Engine architecture. Source – IBM Corporation. [35]

the Synergistic Processor Unit (SPU) and the Memory Flow Controller (MFC). The SPU has 128 registers, each 128 bits wide, and 256KB of software-managed local storage. Each SPU can access only local storage with direct loads and stores and main memory through the MFC by using DMAs. The SPU has a different ISA than the PPE, and leverages vector execution units to implement Cell-specific SIMD intrinsics on the 128-bit wide registers. The MFC is used for performing memory transactions between the local storage and main memory. The SPU and MFC are decoupled enough to enable partial or total computation-communication overlap. Single-precision floating point operations are dual-issued and fully pipelined on the SPEs, whereas double-precision floating point operations have a 13-cycle latency, with only the last 7 cycles pipelined. No other instructions can be issued in the same

instruction slot with double-precision floating point instructions and no instructions of any kind are issued for 6 cycles after a double-precision instruction is issued. These limitations reduce severely the performance of Cell/BE when it performs double-precision floating point arithmetic. Theoretical performance peak of the Cell processor with all eight SPUs active and fully pipelined double precision FP operation is 21.03 Gflops. In single-precision FP operation, the Cell BE is capable of a peak performance of 230.4 Gflops [17].

3.2 NVIDIA GeForce 8800 GTS 512MB

This section summarizes the many-core architecture of the NVIDIA Geforce GPUs and the CUDA programming model from the CUDA Programming Guide [38]. The NVIDIA GeForce 8800 series GPGPU cards can be programmed in the C language by using the CUDA (Compute Unified Device Architecture) library and API (Application Programming Interface). When programmed through CUDA, there is no need to map the function primitives to graphics API like before; the GPU is simply viewed as a highly data-parallel *compute device*. The data-parallel, compute-intensive portions of applications running on the *host* processor are off-loaded onto the GPU. The *kernel* is the portion of the program that is compiled to the instruction set of the device, and then downloaded to the device before execution. Data can be copied from the *host memory* to the *device memory* or vice versa through optimized Direct Memory Access (DMA) calls.

3.2.1 Architecture

The many-core design of the device is implemented as a set of 16 multiprocessors as illustrated in Figure 3.2. At any given clock cycle, each of the 8 processors in the multiprocessor executes

the same instruction, but operates on different data (SIMD – Single Instruction, Multiple Data architecture). Each multiprocessor has on-chip memory of the following four types

1. A set of 8192 local 32-bit registers per processor
2. 16KB of parallel shared data cache that is common to all the processors
3. A read-only constant cache that speeds up data reading from the constant memory that is present on the device
4. A read-only texture cache that speeds up data reading from the texture memory that is present on the device

The device memory consists of 512MB of read-write global memory and 64KB of read-only constant memory, the threads on the device can only read from the constant and texture memories but can read and write to the global memory. The local and global memory spaces are not cached.

3.2.2 Programming Model

The host launches multiple kernels onto the device in succession. Each kernel is executed as a bunch of threads organized as a grid of thread blocks as shown in Figure 3.3. The dimensions of the blocks and the grid of thread blocks are specified before the kernel launch. The ID of the blocks and the threads within each block can be retrieved by the CUDA built-in 2- or 3-dimension structures. Threads can only be synchronized within a thread block but not across blocks.

A thread that is executed on the device can access only the on-chip and device memory modules as shown in Figure 3.3. A grid of thread blocks is executed on the device such that

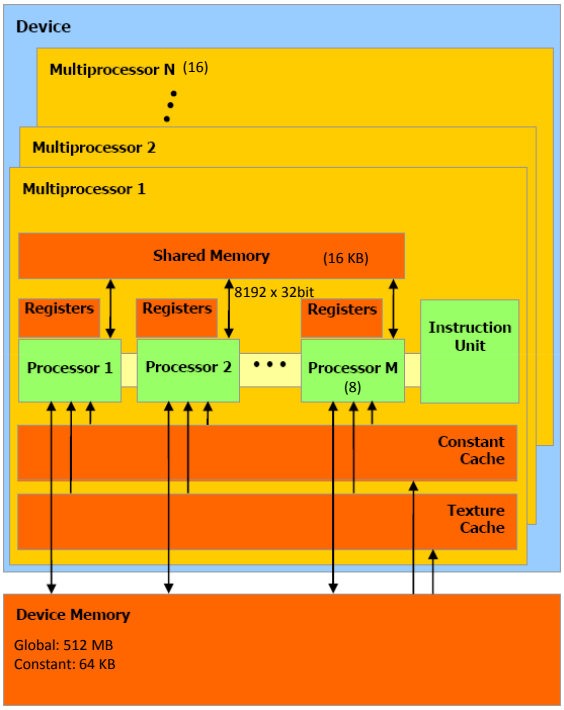


Figure 3.2: The nVIDIA GeForce 8800 GTS 512MB architecture. Source – CUDA Programming Guide. [38]

each multiprocessor processes batches of blocks one batch after the other. A thread block can be mapped to execute on only one multiprocessor, but a single multiprocessor can execute multiple thread blocks at any time. The on-chip shared memory is common to the threads within a block, while the global, constant and texture memory modules are shared across all the the thread blocks in the grid. The device memory can be read from or written to by the host via DMA calls and are persistent across kernel launches by the same application.

Since there is no synchronization mechanism between blocks, threads from two different blocks of the same grid cannot safely communicate with each other through global memory during the execution of the kernel. The exitof a kernel provides an implicit barrier to all the threads that are executed on thedevice.

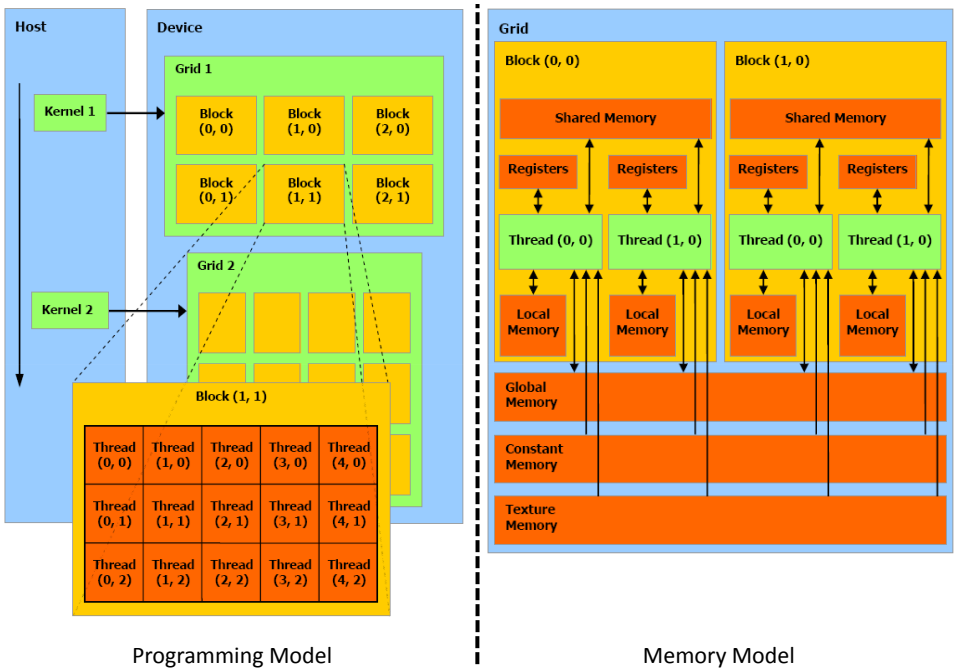


Figure 3.3: The GPGPU Programming and Memory model. Source – CUDA Programming Guide. [38]

Chapter 4

Exploiting Multigrain Parallelism for Pairwise Sequence Search

This chapter presents efficient techniques of parallelizing the Smith-Waterman sequence alignment algorithm on the Cell B.E and the GPGPU compute platforms. We first present Cell-SWat – a highly scalable parallel design of the Smith-Waterman that exploits multigrain parallelism across and within a cluster of Cell-based nodes. We then introduce CUDA-SWat – a multi-layered parallel implementation of Smith-Waterman using CUDA on the nVIDIA GeForce 8800 GTS 512MB graphics platform. In both the cases, we first describe the experimental platform and then proceed to explain our parallel design techniques and finally present the experimental results.

4.1 Cell-SWat: Smith-Waterman on the Cell B.E.

4.1.1 Experimental Platform

Our experiments were conducted on two disjoint clusters of compute nodes hosting the Cell processor. One of them is a cluster of 20 PlayStation 3 nodes built at Virginia Tech for about $8,000, and the other is the *Cellbuzz* cluster of 14 QS20 dual-Cell blades located at Georgia Tech. While most of the PS3 nodes were available to us in a dedicated mode, we had restricted access of up to 7 QS20 blades on the Cellbuzz cluster due to its public and busy nature. We therefore report numbers for up to the same number of nodes on both the clusters to provide a fair performance comparison between the platforms. The nodes on both systems are connected through a GigE switch and communicate with each other via the MPICH2-1.0.7-rc2 MPI library calls. We used the IBM Cell SDK 2.1 for developing parallel codes within the Cell B.E. Each PS3 and QS20 node runs Linux FC5, compiled for the 64-bit PowerPC architecture. Only six of the eight SPEs are accessible to the programmers on the PS3, as one SPE is reserved by the proprietary hypervisor and another is hardware disabled. Also, each PS3 console is provided with less than 256MB of main memory. Each QS20 blade comprises of two 3.2 GHz Cell processors sharing 1 GB of XDRAM (512 MB per processor) based on the NUMA architecture. This unique processor setup enables the threads on the 2 PPE cores to share 16 SPEs, thereby giving 10 additional accelerators per Cell node, as compared to a PS3 console.

4.1.2 Design, Implementation and Results

In this section, we discuss how the multigrain parallelism that is exposed by our experimental platform is effectively exploited to deliver optimal performance, in terms of both speed and

robustness of pairwise sequence alignment. The processing elements within the cluster of Cell nodes can be hierarchically grouped together at three different levels of computational granularity, i.e. Cell nodes, SPEs within each node, and vector processing units within each SPE, as shown in Figure 4.1.

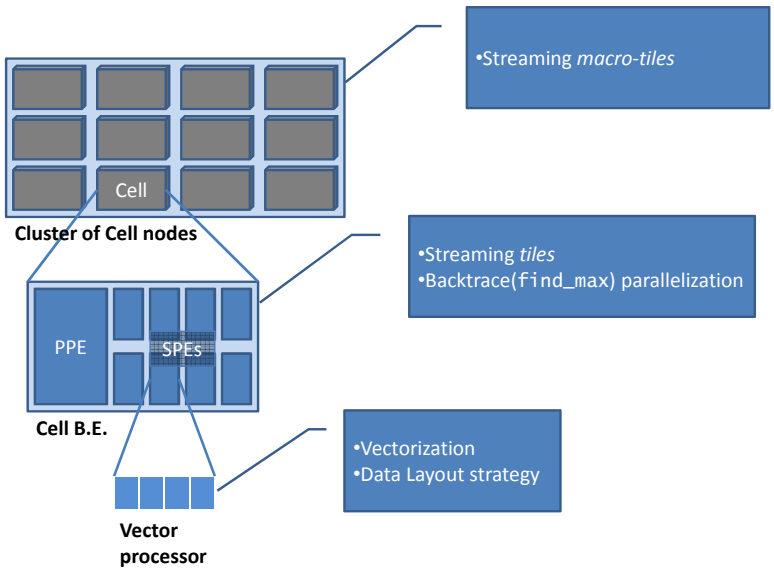


Figure 4.1: Multigrained parallelism exposed by the Cell cluster and our optimizations of parallel sequence alignment at each layer.

We first introduce and explain our strategy to parallelize the Smith-Waterman sequence alignment algorithm in particular, and wavefront algorithms in general, within and across the SPEs on a single Cell node. We also develop an analytical model that accurately predicts the performance of aligning sequences within a Cell node [1].

However, to address the ever-increasing need for higher speeds and larger memory footprints in the sequence search and wavefront computations, we need to simultaneously extract performance from every layer of the multi-grained parallel architecture. Hence, we leverage our

techniques to parallelize within a Cell node and introduce optimization techniques that exploits parallelism at the cluster level in conjunction with the lower micro-architectural layers, as depicted in the Figure 4.1. To emphasize on the generality of the problem, our discussions in this section will henceforth refer to parallelizing wavefront algorithms. But, it must be noted that our design methodology is in particular applicable to the Smith-Waterman sequence alignment algorithm. We also extend our earlier analytical model to accurately predict the execution time of the wavefront algorithms for different input and system configurations. We show that our design for executing parallel wavefront computations *within* a Cell node serves as a remarkably generic design template, and can thus be recursively applied to every layer of parallelism within the Cell cluster.

Parallel Wavefront within a Cell Node

The wavefront algorithm is an important pattern utilized in a variety of scientific applications, including particle physics, motion planning, and computational biology [26]. Computation proceeds like a wavefront filling a matrix, where each cell of the matrix is evaluated based on the values of cells computed earlier. The algorithm advances through the matrix by computing all anti-diagonals starting from the northwest corner, as shown in Figure 4.2(a). The computation carries dependencies across anti-diagonals, that is, each element of the matrix depends on its respective northern, western, and northwestern neighbors, as shown in Figure 4.2(b). The Smith-Waterman algorithm directly falls into the category of wavefront algorithms.

While consecutive anti-diagonals are dependent, the cells lying on the same anti-diagonal are independent and can be processed in parallel. Processing individual matrix elements in parallel incurs high communication overhead which can be reduced by grouping matrix cells into large, computationally-independent blocks, which are more suitable for parallel

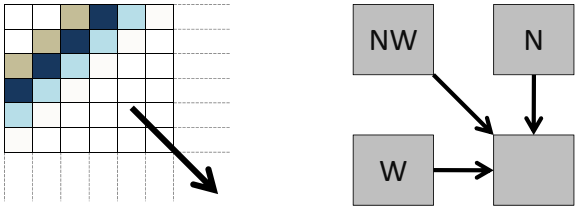


Figure 4.2: A general wavefront algorithm (a) and its dependencies (b).

processing. This common optimization strategy is outlined in Figure 4.3. We refer to each block of matrix cells as a *tile*. The coarsened basic unit of work does not change the properties of the wavefront algorithm – the algorithm advances through the matrix by computing anti-diagonals which are composed of multiple tiles.

The most important aspects of the wavefront algorithm are tile computation and communication among processes which perform computation on different tiles. We describe each of the two steps and their implementation on the Cell/BE in more detail next.

Tile Computation The Cell/BE contains multiple accelerator cores capable of performing independent asynchronous computation. To map the wavefront algorithm to the Cell/BE we assign independent tiles for processing on different SPEs. Assuming the matrix is divided in square tiles, as presented in Figure 4.3, the execution starts by processing tile t_1. Due to the computational dependencies across anti-diagonals, the tiles lying on the anti-diagonal t_2 can be processed only after t_1 has been computed. Although the described behavior limits the amount of parallelism exposed by the application, the utilization of the SPE cores increases as the algorithm advances through the matrix. Starting with the anti-diagonal t_8, the number of tiles available for parallel processing is equal to or exceeds the number of SPEs on a single Cell chip, and all SPEs can be used for tile processing.

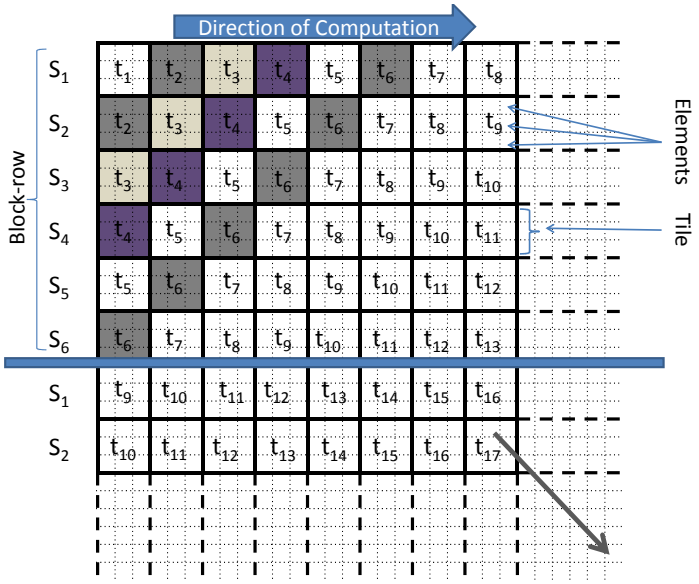


Figure 4.3: Tiled wavefront on a PS3 with 6 SPEs.

While different scheduling strategies can be used for assigning the units of work to SPEs, we focus on predetermined tile-SPE assignment in this study. Our scheduling scheme achieves perfectly balanced SPE work assignment, while at the same time enables complete utilization of the Cell chip. We change the algorithm computation direction, and instead of computing entire anti-diagonals, the algorithm advances through the *block-rows*, as shown in Figure 4.3. The height of each block-row is equal to the total number of SPEs. For anti-diagonals which contain more tiles than the number of available SPEs, the part of the anti-diagonal which belongs to the block-row is computed, and the computation shifts to the next anti-diagonal. Note that this is legal execution since the computation of each tile depends on its north, west, and northwest neighbor. The same process repeats until the algorithm reaches the right edge of the matrix, after which the computation continues in the next block-row. The matrix is split into multiple block-rows and possible underutilization of the Cell processor might occur only in the last row, if the height of the row is smaller than the number of SPEs.

This can easily be avoided by resizing the tiles in the last block-row.

While working on an anti-diagonal in a block-row, each SPE is assigned a single tile. Along with the algorithm, the SPEs advance through the block-row towards the right edge of the matrix. After reaching the edge, each SPE continues processing the tiles contained in the next block-row. No two SPEs reach the edge of the matrix at the same time, which causes computation overlap of consecutive block-rows, which is shown in Figure 4.4 (processing the end of the first block row overlaps with the beginning of the second block row). Simultaneous processing of different block-rows enable high utilization of the Cell processor – the idle SPEs are assigned work units from the next block-row.

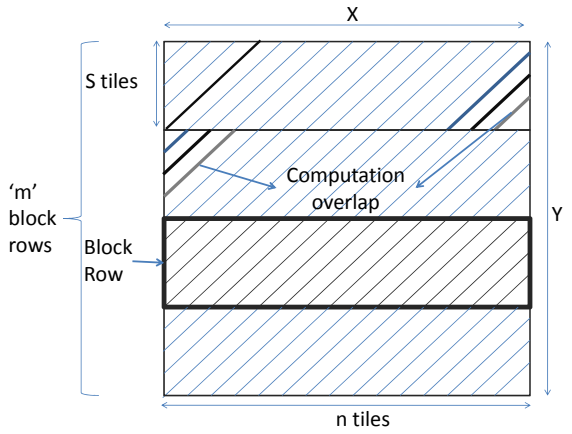


Figure 4.4: Matrix divided into block rows.

Computation-Communication Communication patterns that occur during the tile computation are shown in Figure 4.5. We describe step-by-step communication-computation mechanism performed by each SPE while processing a tile:

1. To start computing a tile, an SPE needs to obtain boundary data from its west, north, and northwest neighbor. The boundary elements from the northern neighbor

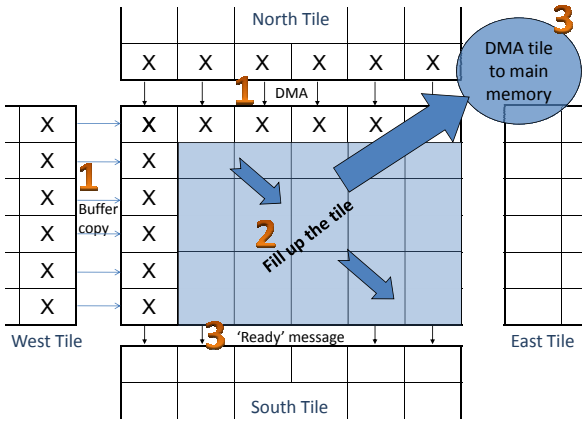


Figure 4.5: Computation-Communication pattern between tiles.

are fetched to the local storage from the local storage of the SPE which was processing the northern neighbor. The boundary elements of the west neighbor do not need to be fetched due to the fact that each SPE advances through a tiled row, and therefore each SPE already contains the required data. The necessary boundary elements of the northwestern neighbor also reside in the local storage of the SPE which processed the northern neighbor, and are fetched along with the boundary elements from the northern neighbor.

2. In the second step, the SPE proceeds with the tile computation.

3. Finally, the SPE moves the tile to main memory for post-processing and notifies the SPE which works on the south neighboring tile that the boundary elements are ready for transfer.

The above steps describe the processing of non-boundary tiles. Boundary conditions can be easily checked and the redundant steps can be avoided.

The Performance Prediction Model To capture the performance of the wavefront algorithm on the Cell/BE, we developed an analytical model capable of accurately predicting the total execution time of the algorithm. As an input, the model takes several application specific parameters and estimates execution time on a variable number of cores. We start the discussion about the model by introducing the following equation:

$$T = T_F + T_{serial} \tag{4.1}$$

In Equation (4.1), T represents the total time taken by the algorithm, T_F is the time the algorithm uses to fill the matrix, and T_{serial} is the time taken by the inherently sequential part of the algorithm.

If we denote the time needed to compute one tile as T_{Tile} and the time used to fetch and commit the data necessary for tile computation as T_{DMA}, then the total time spent processing a single tile can be represented as $T_{Tile} + T_{DMA}$. Since all elements of a single anti-diagonal (in a block-row) are processed in parallel, the time to process the entire anti-diagonal (in a block-row) can also be represented as $T_{Tile} + T_{DMA}$. The total number of tiled anti-diagonals should be carefully counted, since it involves overlaps of anti-diagonals between adjacent block rows. By inspecting Figure 4.4, we estimate the total number of anti-diagonals to be $(n \cdot m) + S$, where m represents the total number of block-rows, n represents the number of anti-diagonals per block-row, and S is the number of anti-diagonals containing less tiles than the number of SPEs and therefore their processing does not utilize the entire Cell chip. In Figure 4.4, these diagonals are represented in the upper left corner of the matrix. S depends on the number of SPEs, and therefore we can denote it as $S(N_{spe})$, where N_{spe} is the number of available SPEs. From the above discussion, we represent the total time T

from Equation (4.1) as:

$$T = (T_{Tile} + T_{DMA}) \cdot [(m \times n) + S(N_{spe})] \tag{4.2}$$

We can further decompose m as:

$$m = \frac{Y}{T_{Size} \cdot N_{spe}} \tag{4.3}$$

where Y represents elements (not tiles) in the y-dimension of the matrix, T_{Size} represents the size of a tile, and N_{spe} is again the number of available SPEs (equal to the height of the block-row). Also, we can decompose n as:

$$n = \frac{X}{T_{Size}} \tag{4.4}$$

where X represents elements (not tiles) in the x-dimension of the matrix,

Combining Equations (4.1), (4.2), (4.3), and (4.4), we derive the final modeling equation:

$$T = (T_{Tile} + T_{DMA}) \cdot \left(\frac{X \cdot Y}{T_{Size}^2 \cdot N_{spe}} + S(N_{spe}) \right) + T_{serial} \tag{4.5}$$

To employ the model as a run-time tool capable of determining the most efficient execution configuration, we need to estimate all parameters included in the model. The parameters T_{Tile} and T_{DMA} need to be measured before they can be plugged into the model. The measurement can be performed during a short *sampling* phase, which would occur at the beginning of the program execution or via offline microbenchmarks. By knowing T_{Tile} and T_{DMA} for a single tile size (T_{size}), we can accurately estimate the same parameters for any tile size. This is due to the fact that each tile is composed of the matrix cells that require equal amount of processing time. X and Y depend on the input data set and can be determined statically. N_{spe}, $S(N_{spe})$ and T_{size} are related to the number of SPEs used for parallelization

and the tile size. These parameters can iterate trough different values, and those that provide the most efficient execution will be used for the algorithm execution. Parameter T_{serial} does not influence the parallel execution of the program, and we can disregard this parameter while searching for the most efficient parallel configuration.

Implementation Details and Optimizations

Tile representation Each tile is physically stored in memory, as a 1D array, by storing adjacent anti-diagonals next to each other. This is depicted in Figure 4.6. This arrangement makes it easier to perform vector operations on the tile by taking one anti-diagonal at a time.

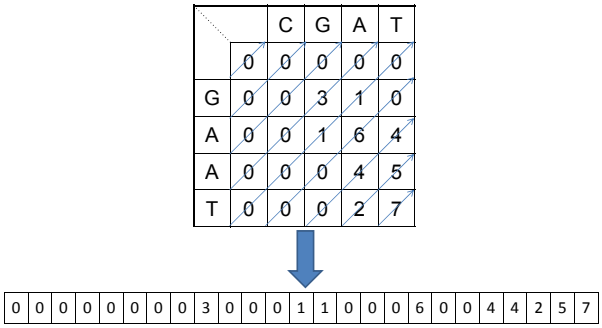


Figure 4.6: Tile representation in memory.

Vectorization of the tile for the SPE We assign each tile to execute on individual SPEs. To extract the true potential of the SPEs, the data has to be vectorized before being operated upon. The vectorization process that we follow is described by Figure 4.7. A two-dimensional (2D) representation is shown in the figure (instead of 3D) for the sake of simplicity. During a tile vectorization process, we process one anti-diagonal at a time following the wavefront

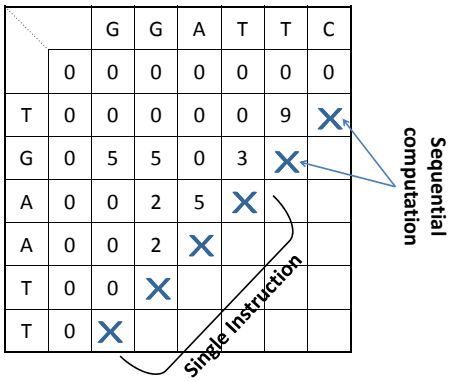
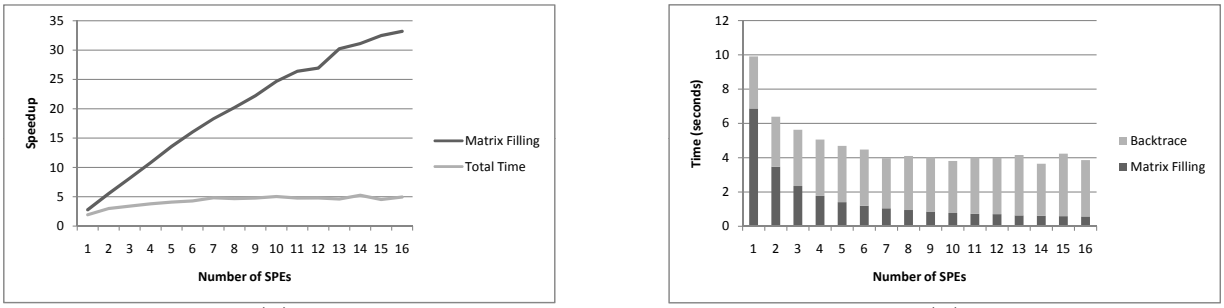


Figure 4.7: Tile vectorization.

pattern. To effectively utilize the SIMD capabilities of the SPE, the anti-diagonal must be divided into as many vectors as possible. The number of elements on the anti-diagonal keeps changing for every anti-diagonal and cannot be perfectly partitioned into vectors in some cases. In these cases, the remainimg elements undergo a serial computation. Upon vectorization, we obtained the speedup and execution time curves shown in Figure 4.8(a) and (b), respectively. These timings were recorded for input sequence lengths of 8 KB. Figure 4.8 indicates reduced speedup when the number of SPEs exceeds 6. The reason is the backtrace phase, which is completed solely on the PPE and does not depend on the number of SPEs. The sequential backtrace calculation on the PPE is the next bottleneck for optimization.

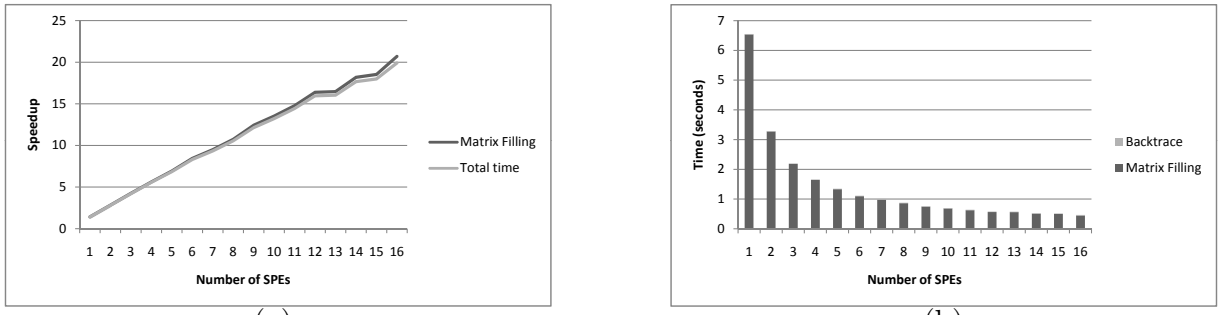
The backtrace optimization The backtrace begins at the matrix cell that holds the largest alignment score; therefore, a find_max operation is needed. Initially, our implementation executed this function on the PPE after the entire matrix was filled up. To reduce the high PPE overhead caused by the backtrace operation, we optimized find_max by parallelizing it across SPEs. The local optimum score calculated by each SPE is passed on to the PPE at the end of the matrix filling phase. From this data, the PPE calculates the overall



(a) (b)

Figure 4.8: Speedup (a) and timing (b) charts before optimizing the backtrace operation.

optimum score by performing at most S if checks, where S is the number of SPEs used. This optimization had a considerable impact on the achieved speedups (as shown in Figure 4.9).



(a) (b)

Figure 4.9: Speedup (a) and timing (b) charts after optimizing the backtrace operation.

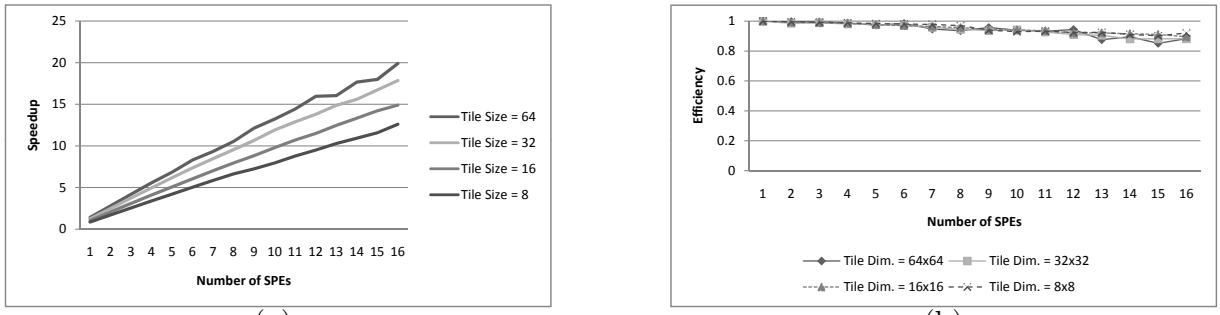
Experimental Results

We present results from experiments on a single, dedicated dual-Cell/BE QS20 blade. We chose the QS20 blade over the PS3 for this experiment because of the availability of more SPE cores on the QS20, and thus our design can be tested extensively. We conducted these

experiments by aligning sequences of realistic sizes as are currently present in the NCBI Genbank nucleotide (NT) database. There are approximately 3.5 million sequences in the NT database. Of those, approximately 95% are 5 KB in size or less [22]. For the tests, we chose eight randomly generated sequence pairs of sizes varying from 1 KB to 8 KB in increasing steps of 1 KB, thus covering most of the realistic sequence sizes. We randomly generated the input sequences because the complexity of the Smith-Waterman algorithm is dependent only on the sequence length and not on the sequence contents. We repeated the tests for the above sequence lengths by varying the number of SPE threads from 1 to 16 to test the scalability of our implementation on up to two fully utilized Cell processors. To measure the effect of tile granularity on the execution times, we repeated all of the above experiments for tile sizes of 8, 16, 32 and 64 elements. To measure the speedup of our implementation, we executed the serial version of Smith-Waterman on a machine with a 2.8-GHz dual-core Intel processor and 2-GB memory, and we used one of the two cores present on the chip. We believe that using the Intel processor as a basis for calculating speedup on the Cell is more realistic than using the PPE core, which has very limited computational capacity compared to the SPEs. Using the PPE core as a basis for speedup calculation would only inflate the results with not much added value.

Speedup Figure 4.10(a) and (b) illustrates the achieved speedup and efficiency with different numbers of SPEs. Similar curves were observed for all eight sequence sizes.

The speedup curves indicate that our algorithm delivers perfect linear speedup or near-constant efficiency for up to 16 SPEs, irrespective of the tile size, and it is highly scalable for more cores if they are available on the chip. The figure also shows that as the tile size increases, more speedup is achieved. This is because more data is locally available for each SPE to work upon, and there is less communication overhead between the SPEs. We were

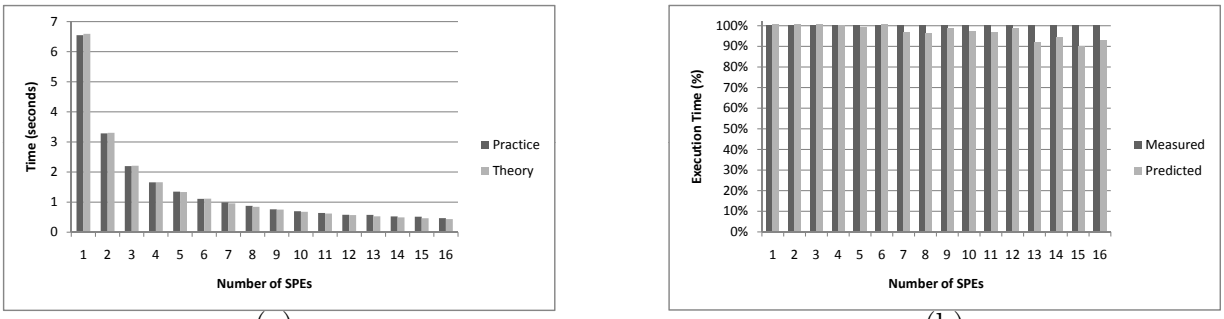


(a) (b)

Figure 4.10: The obtained (a) speedup and (b) efficiency for input sequences of length 8KB. The number of SPEs varies from 1 to 16.

not able to choose a tile size of more than 64 elements because the memory required to work on a single tile exceeded the capacity of the local store of the SPE.

Model Verification To verify our model, we initially experimentally measured T_{tile}, T_{DMA} and T_{serial} by varying the other parameters of Equation (4.5). We chose an example sequence pair of 8KB in size and tile size of 64 for this experiment. The measured values for this configuration were $T_{tile} = 0.00057s, T_{DMA} = 10^{-6}s$ and $T_{serial} = 0.015s$. By varying S from 1 to 16, we generated a set of theoretically estimated execution times. The theoretical estimates from our wavefront model was then compared to the actual execution times, as seen in Figure 4.11(a) and (b). Similar results were observed for all the other sequence sizes and tile sizes as well. This shows that our model estimates accurately the execution time taken to align two sequences of any size, using any number of SPEs or any tile size. The model error is within a range of 3% on average.



(a) (b)

Figure 4.11: Chart showing theoretical (a) timing estimates and (b) normalized times of our model (labeled theory) against the measured execution times (labeled practice).

Parallel Wavefront across Multiple Cell Nodes

In this section, we leverage the design presented in the previous section to execute wavefront algorithms on multiple SPEs spread across a cluster of Cell nodes. The purpose of the new design is twofold: (1) to speedup the current implementations further by utilizing more synergistic processing elements and (2) to accommodate problem sizes that do not fit into the main memory of a single node.

The methodology that we discuss here is similar to the tiled-wavefront approach, but it works at a coarser granularity of computation, communication and data elements. We decompose the original matrix into multiple equal-sized sub-matrices called *macro-tiles*. Each macro-tile is further divided into multiple tiles, where each tile by itself contains a bunch of individual matrix elements as shown in Figure 4.12. Each macro-tile will be executed on a single node in the Cell cluster, thus remaining faithful to the one-one mapping between the larger data granularity and the processor size.

The data representation at the highest level of granularity will now consist of many *macro-tile-rows*, *macro-tile-columns* and *macro-tile-diagonals*. The bigger basic unit of work has not

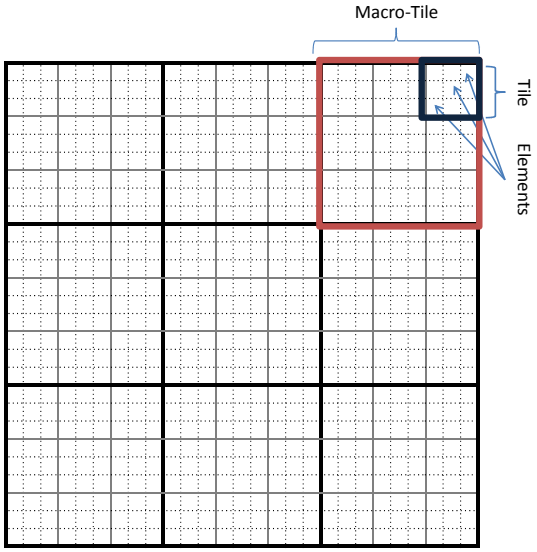


Figure 4.12: Recursive decomposition of the matrix into macro-tiles and tiles.

changed the wavefront properties of the algorithm – the macro-tiles on the same macro-tile-diagonal are computationally independent and can be processed concurrently on different nodes. The algorithm advances through the matrix by processing anti-diagonals that comprise of many macro-tiles. We have already shown a highly efficient model to process the tiles within a macro-tile across the SPEs in a single Cell system in the previous section. We now present the scheduling scheme for computing the various macro-tiles of the matrix on a cluster of Cell systems, and discuss the computation-communication pattern of the active nodes in the cluster next.

Macro-tile Scheduling We assign the computation of macro-tiles on different nodes in the cluster in a manner that is similar to scheduling tiles on different SPEs. The macro-tile-rows of the matrix are cyclically assigned to the available nodes for computation as shown in Figure 4.13. Execution begins by processing macro-tile m_1 on node N_1. The

subsequent iterations process the macro-tiles along the anti-diagonals labeled m_2, m_3, m_4, etc concurrently on increasing number of nodes, until all the available nodes are busy. The anti-diagonals are processed in a pipelined fashion that induces an unavoidable initial pipeline-setup latency, which can be significant for certain input configurations of realistic datasets. This overhead is one of the main parameters to be evaluated before selecting the optimum computation configuration, and is faithfully characterized in the execution model that we present later in this section. Beginning from the iteration in which the anti-diagonal contains more macro-tiles than the number of available nodes, the computation moves across to the next anti-diagonal along the same *macro-block row*, as shown in Figure 4.13.

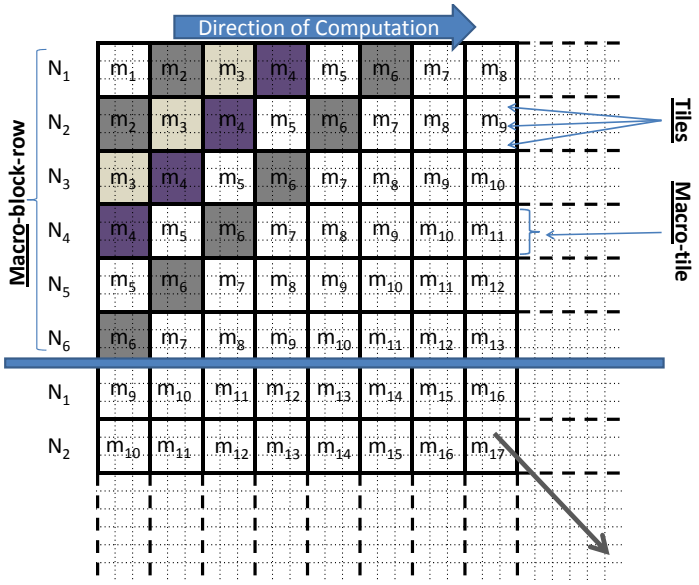


Figure 4.13: Macro-tiled wavefront on a PS3 with 6 SPEs.

Computation-Communication Communication between the nodes processing the macro-tiles is done through the interconnection network via explicit message passing routines. We can observe that as the granularity of the computed data increases with that of the processing

elements, the granularity of communication follows suit. The computation-communication patterns that can be observed while processing a macro-tile is shown in Figure 4.14, and categorically explained below:

1. Before computing the macro-tile, each node fetches the required boundary data elements from the node that processed the north macro-tile. The communication between nodes is done via explicit message passing routines, and the tag associated with the message will suffice as the synchronization mechanism between nodes processing adjacent macro-tiles. The required boundary elements from the west macro-tile are already present in the main memory of the node because the same node computes an entire macro-tile-row.

2. In the second step, the node processes all the tiles within the macro-tile as explained before in the section on parallel wavefront algorithms within a Cell node.

3. Finally, the processed tile is transferred to other forms of storage for future processing. [Optional]

The above steps are repeated by all the nodes until the entire matrix is computed.

The Performance Prediction Model We begin our discussion about the execution model by comparing the execution and system parameters of the wavefront computations that are executed on stand-alone Cell systems against those on a cluster of Cell nodes, as shown in Table 4.1.2.

From the above-mentioned table, we observe that as we move up the processor hierarchy, the model performs identical operations, but on coarser units of computation, communication and memory. Hence, the execution model that we developed to accurately predict the running times of wavefront algorithms within a single Cell node can be generalized to wavefront

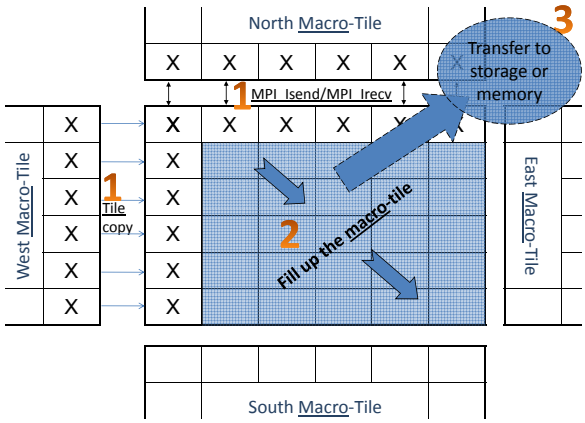


Figure 4.14: Computation-Communication pattern between macro-tiles.

Table 4.1: Comparison of the the execution parameters in the model that performs wavefront computations within a Cell system against those on a cluster of Cell nodes.

	Within the Cell B.E.	*Across a cluster of Cell B.Es*
Stream Processors	SPE	PPE
Basic Unit of work	Tile	Macro-tile
Parallel Execution Code	SPE thread	Individual process
Medium for Communication	Bus (DMA)	Interconnection Network (MPI)
Storage for intermediate results (cache)	Local Storage of the SPE	Main memory

computations across multiple Cell machines by simply replacing the fine-grained parameters with the coarse-grained counterparts.

We first review our analytical model for predicting the execution times of wavefront computations of a single macro-tile on a stand-alone Cell system as shown in Equation (4.6):

$$T_{MTile} = (T_{Tile} + T_{DMA}) \cdot \left(\frac{X \cdot Y}{T_{Size}^2 \cdot N_{spe}} + S(N_{spe}) \right) + T_{ppe\_misc} \tag{4.6}$$

where,

T_{MTile} = Total time to compute a single macro-tile

T_{Tile} = Time to compute a single tile

T_{DMA} = Time to fetch and commit the necessary data for tile computation

T_{Size} = Size of the tile along the X or Y dimension

N_{spe} = Number of SPEs used

$S(N_{spe})$ = Function of the number of SPEs used

T_{ppe_misc} = Time taken by the inherently sequential (miscellaneous) part of the algorithm

X, Y = Input sequence lengths

As discussed, we replace the fine-grained parameters from Equation (4.6) with coarse-grained ones that are specific to wavefront computations on a Cell cluster as shown in Equation (4.7).

$$T = (T_{MTile} + T_{Comm}) \cdot \left(\frac{X \cdot Y}{MT_{Size}^2 \cdot N_{nodes}} + S(N_{nodes}) \right) + T_{misc} \tag{4.7}$$

where,

T = Total time taken

T_{MTile} = Time to compute a single macro-tile (Equation (4.6))

T_{Comm} = Time to communicate the necessary data for macro-tile computation and transfer of intermediate results to storage

MT_{Size} = Size of the macro-tile along the X or Y dimension

N_{nodes} = Number of nodes used

$S(N_{nodes})$ = Function of the number of nodes used – this parameter characterizes the initial pipeline setup latency

T_{misc} = Time taken by the inherently miscellaneous part of the algorithm

X, Y = Input sequence lengths

While we include the parameter T_{misc} in our model for completeness, we disregard it while using our model for finding the optimal system configuration because it bears no influence on the running times of our parallelization strategy.

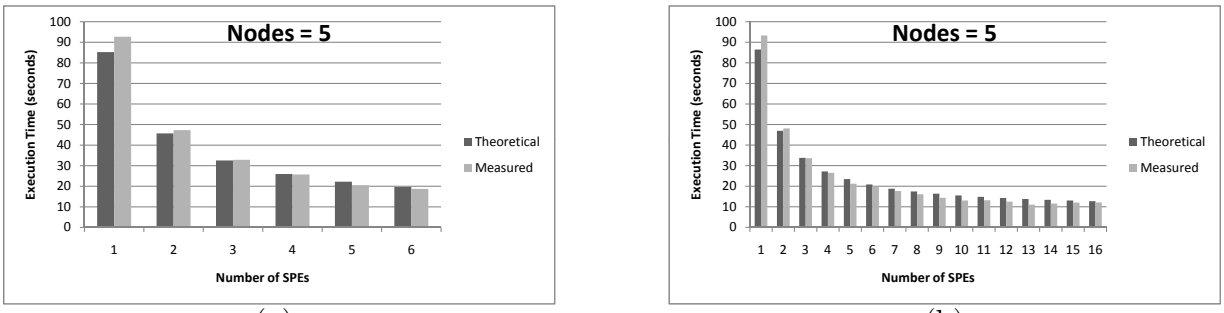
To make use of our model to predict the running times, we need to measure the parameters T_{Tile}, T_{ppe_misc}, T_{DMA} and T_{Comm} before they can be plugged into the model. This is done during a short *sampling* phase, which would occur at the beginning of the program execution or by running micro-benchmarks, such as the `dmabench` program that ships with the Cell SDK, before actually deploying the algorithm on the execution platform. If we accurately measure T_{Tile} and T_{DMA} for a single tile size T_{size}, and measure T_{Comm} for a single macro-tile size MT_{size}, we can accurately estimate the same parameters for any tile and macro-tile size respectively. This is because the DMA/communication times are proportional to the size of the data array that is transferred, and the tile computation time is proportional to number of elements in the tile. The remaining variables can iterate through different values, and the optimal configuration can be used for purposes of deployment.

Experimental Results

Model Verification In this section, we verify the accuracy of the execution model that we discussed in the previous section by running the Smith-Waterman algorithm across the cluster of Cell processors.

To verify the model, we initially measured T_{Tile}, T_{DMA}, T_{ppe_misc} and T_{Comm} for $T_{Size} = 64$ and $MT_{Size} = 3150$ for both the QS20 and PS3 clusters. The input query sequences were both 51KB in length. We could access up to 7 QS20 Cell blades for the purpose of this experiment, and hence we show validation results for up to the same number of nodes on the PS3 cluster to have a fair comparison of our model verification between the two platforms.

Also, we do not perform the post-processing step for the following experiments because of the overhead of writing large amounts of the intermediate results to disk, and hence trade-off robustness for speed. We generated a set of theoretically estimated times by iterating N_{spe} from 1 to 16 on the QS20 blades and from 1 to 6 on the PS3. We varied N_{nodes} from 1 to 7 on both the platforms for each value of N_{spe}. The theoretically generated times were then compared against the actual execution times as shown in Figures 4.15 and 4.16.



(a) (b)

Figure 4.15: Chart comparing theoretically estimated times of our model against the measured execution times for sequence alignment across 5 nodes of (a) PS3 cluster and (b) QS20 cluster.

The model error is consistently within an average error range of 5% on the PS3 cluster and 10% on the QS20 cell cluster. Similar consistent results were observed for the other different configurations as well.

Speedup Figure 4.17 shows a 3-dimensional illustration of the speedup obtained by executing the parallelized Smith-Waterman for the different combinations of the number of SPEs and nodes in the PS3 and QS20 Cell clusters. The speedup ranges are shown as color-coded contours on the chart surface, and we can see that the PS3 cluster scales very well for more nodes and more SPEs achieving a 26× maximum speedup, while the QS20 Cell cluster

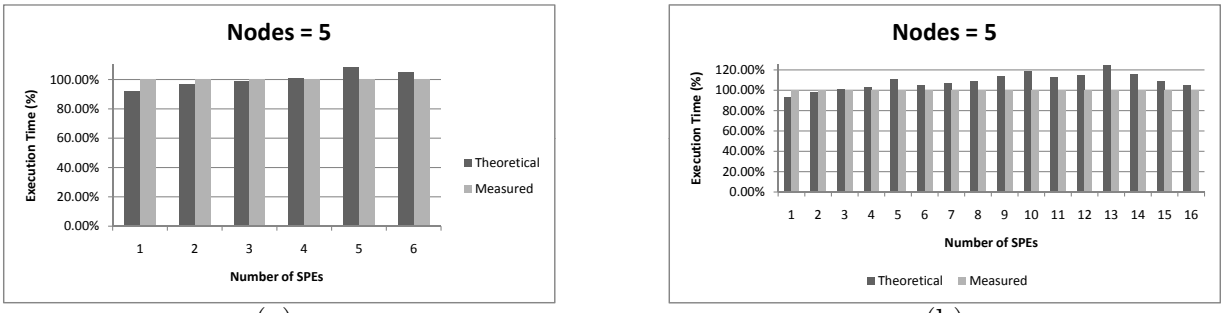
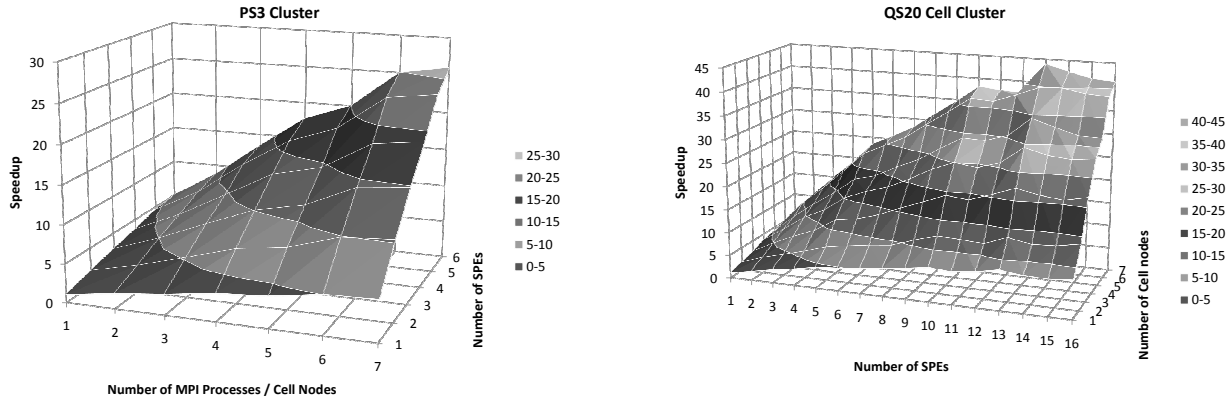


Figure 4.16: Chart comparing normalized theoretically estimated times of our model against the measured execution times for sequence alignment across 5 nodes of (a) PS3 cluster and (b) QS20 cluster.

reports a 44× maximum speedup, where the base measurement for speedup is the execution time recorded on a single PPE-SPE combination. The scalability of the QS20 cluster drops beyond an SPE count of 13 because the gain achieved by the extra SPEs is negligible when compared to the overhead of the MPI communication layer. We use the execution time of running the parallel Smith-Waterman on a single Cell node, using a single SPE as the basis for speedup. We observe similar speedup patterns for other input and system configurations as well.

4.1.3 Throughput Oriented Sequence Search

In this section, we consider a realistic scenario for the use of Smith-Waterman by computational biologists, where more number of pairwise sequences need to be aligned per time, i.e. we target to achieve higher sequence throughput. The straightforward approach is to align the sequence pairs, one pair at a time, in a first-come-first-served (FCFS) fashion – where each alignment uses all the available compute resources and achieves maximum parallelism



(a) (b)

Figure 4.17: Color-coded contour chart showing speedup ranges on (a) PS3 cluster and (b) QS20 cluster.

within each sequence alignment. However, we can also achieve parallelism *across* sequence alignments where many sequence pairs are aligned at the same time, and each pair uses lesser computing resources. We analyze the tradeoffs between the above two approaches by leveraging the performance prediction model that was introduced in Section 4.1.2, for both within and across compute nodes in a Cell cluster. Lastly, we contribute and evaluate TOSS – a Throughput-Oriented Sequence Scheduler, which follows the greedy paradigm and spatially distributes the available computing resources *dynamically* among simultaneous sequence alignments to achieve better throughput in sequence search. By using TOSS, we achieve an improvement of 33.5% on the QS20 Cell cluster and about 13.5% on the PS3 cluster over the naïve FCFS scheduler.

Static Sequence Scheduler within a Cell Node

The FCFS approach to align sequence pairs within a Cell node is to simultaneously use all 16 SPEs. By using all the 16 SPEs for one alignment, we achieve maximum parallelism

within each sequence alignment. However, we can also achieve parallelism across sequence alignments where many pairs are aligned at the same time, and each pair uses less than 16 SPEs. A simple experiment was conducted by executing 2, 4 and 8 pairs of sequences in parallel, and this was compared against the FCFS approach. The results are as shown in Figure 4.18. The results indicate the processing multiple sequences in parallel achieves higher throughput than processing each sequence separately using all available SPEs. More specifically, sacrificing some parallelism within each sequence alignment can be traded off profitably for increasing the number of sequence alignments processed in parallel, via spatial partitioning of the Cell SPEs.

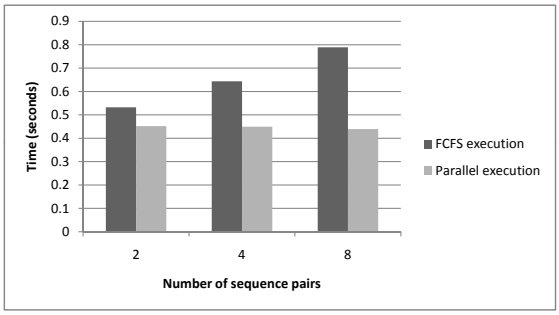


Figure 4.18: Comparison of FCFS execution and parallel strategies where 2, 4, and 8 pairs of sequences are processed in parallel.

We first create a static scheduling algorithm for achieving sequence throughput by deciding the set of sequence pairs that have to executed in parallel. To test the described strategy, we obtained the distribution of sequences in the nucleotide (NT) database. However, running our static scheduler on the entire NT database would take several days, and would not have any added value in validating our scheduling algorithm. Therefore, we first computed the distribution of the sequences in the database based on their sizes, as shown in Table 4.2. We then randomly generated 100 sequence pairs based on the scaled-down distribution, thereby

Table 4.2: Distribution of nucleotide sequences in the NT database.

Sequence size (KB)	Count	Count scaled-down to (0–100)
0 – 0.96	553,030	9
0.96 – 1.6	2,239,588	36
1.6 – 2.1	1,389,550	22
2.1 – 3.2	1,229,649	20
3.2 – 30	782,024	12
30 – 60	16,553	0
$\geq$ 60	77,208	1
Total	6,287,602	100

imitating the actual NT database. Since the performance of the Smith-Waterman algorithm depends only on the sequence size, we can easily extrapolate our results to the entire dataset. Further, we disregard sequences larger than 3.2KB because the corresponding matrix will not fit into the available memory within a single Cell node.

Our static scheduling scheme takes equal-sized sequence pairs in batches and executes them in parallel, provided that they not overflow the available memory. While this scheme is by no means optimized, it can show the potential of our model by taking into account the estimated speedup and scalability slopes for each sequence length, while scheduling multiple alignments. We evaluate the tradeoffs of the FCFS approach versus our scheduling approach for the experimental work set and the results are shown in the Table 4.1.3. The analytical model we developed and described in Section 4.1.2 can be used to analyze the different tradeoffs between FCFS and the various sequence scheduling policies accurately. In the case of our static scheduling scheme, we are able to improve throughput compared to FCFS and execution of alignments at the maximum level of available concurrency by 8%.

Table 4.3: Performance comparison between the FCFS approach and parallel execution on a realistic data set.

FCFS execution	Parallel execution
26.67792s	24.552805s

TOSS: Throughput-Oriented Sequence Scheduler

While there are multiple combinations of possible configurations that can be evaluated before optimizing the throughput of sequence search, we follow the Greedy algorithmic paradigm to design a very efficient dynamic sequence alignment scheduler for this study.

Here are some key assumptions and guidelines that we follow while designing TOSS:

- For the sake of simplicity, each sequence in the database will be aligned against a similar-sized query sequence. The dynamic programming matrix will therefore have equal number of elements in both the dimensions.
- All nodes have the same amount of physical memory.
- All nodes have the same number of active SPEs.
- The Smith-Waterman algorithm comprises of two phases: (1) matrix filling phase, where a dynamic programming matrix is filled following the wavefront pattern and (2) backtrace phase, which is a sequential post-processing operation and requires the entire matrix to be stored in memory before this operation is performed. Chapter 2 defines *robustness* of an alignment in the following way – if the algorithm performs both the phases (matrix filling and backtrace) of a sequence alignment, then the alignment it termed to be *robust*. The memory requirement of a robust alignment is more severe than that of a non-robust alignment. We prioritize to perform robust alignments if possible in the following way:

- If the memory requirement for a robust sequence alignment is within the physical memory capacity of a single node, we will *not* align that sequence-pair across nodes. The matrix filling and the backtrace operations can both be optimally executed, and thus the robustness of the alignment is maintained.
- If the memory requirement for a robust sequence alignment exceeds the capacity of the physical memory within a single node, then the sequence will be aligned across multiple nodes in the cluster. The backtrace operation for that alignment would require extremely slow and inefficient disk accesses multiple times for storing the intermediate results. In this case, the backtrace operation will not be performed as we trade off the robustness of the alignment for speed.

We first discuss the pre-processing steps that have to be executed before deploying TOSS, and present the pseudo-code for TOSS in Figure 4.19.

Pre-processing steps:

1. Distribute the given sequences into β different categories, based on the memory that is required to perform a robust alignment for each of the sequences.
   (a) Label the categories from A_1 through A_β in the increasing order of memory requirements, i.e. A_1 will have the sequences with the least memory requirements while A_β will have the sequences with the highest memory requirements, while maintaining robustness.
2. Split the categories further into two sub-categories: A_1 to A_α and $A_{\alpha+1}$ to A_β, such that the first α groups contain sequences whose memory requirements enable them to be aligned within a single node, and the next $(\beta - \alpha)$ groups contain the remaining larger sequences that have to be aligned across nodes.

The 'Greedy' paradigm

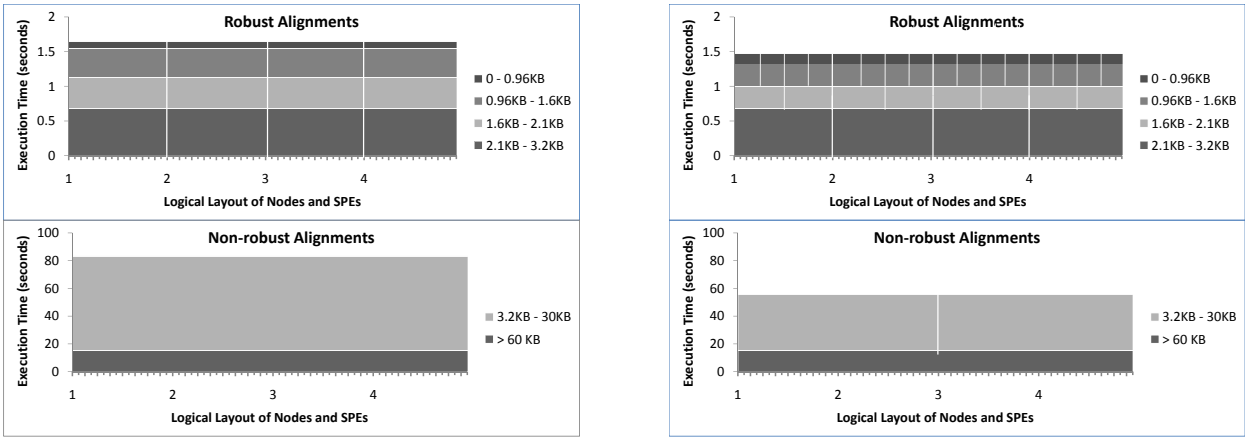
```
1  j <- number of available nodes
2  N_spe <- number of available SPEs
3  per node

4  for each group A in (A_1 to A_α)
5  {
6     distribute the sequences of A
7     equally to the available
8     nodes.
9
10    n <- number of sequences to be
11    aligned in each of the nodes;
12
13    w_A <- number of SPEs in the
14    speedup chart where the
15    efficiency (performance per
16    SPE) is maximum to align a
17    single sequence of category A;

      /*w_A can be calculated by using
      a combination of our model and
      the memory requirements for a
      robust alignment of sequences
      in the category A. For
      example, on applying our model
      to the sequence set, we may
      discover that aligning 4
      sequences in parallel such
      that each sequence works upon
      2 SPEs is more efficient than
      executing the 4 sequences in a
      first-come-first-served (FCFS)
      fashion by using all 8 SPEs
      for each sequence alignment.*/
18
19    s_A <- maximum sequences of
20    category A that can be
21    optimally aligned in parallel
22    within a single node;
23    s_A <- N_spe / w_A;
24
25    Execute the following
26    concurrently on nodes (1..j)
27    {
28        n' = n
29        while(n > s_A)
30        {
31           concurrently align s_A
32           sequences;
             /*each sequence uses w_A
             SPEs for alignment*/
33           n = n - s_A;
34        }
          /*Optimization*/
35        Align the remaining
36        sequences (n'mod s_A) in
37        parallel by using
38        (N_spe/(n'mod s_A)) number of
39        SPEs for each alignment
40    }
41 }
42

43 for each group A in (A_α+1 to A_β)
44 {
45    w_A <- number of nodes in the
46    speedup chart where the
47    efficiency (performance per
48    node) is maximum to align a
49    single sequence of category A;
50
      /*w_A can be calculated by using
      a combination of our model
      applied to the sequence
      category A. For example, on
      applying our model to the
      sequence set, we may discover
      that aligning 2 sequences in
      parallel on 4 nodes such that
      each sequence works on 2
      nodes, is more efficient than
      executing the 2 sequences one
      after another in a FCFS
      fashion by using all 4 nodes
      for each sequence alignment.*/

      B <- the set of sequences that
      are executed concurrently in a
      single batch on the j
      available nodes
51    Clear(B);
52
53    w <- the cumulative node count
54    that is required to align the
55    batch of sequences in B
56    w <- 0;
57    for each sequence a in A
58    {
59        if(w < j)
          /*we have not yet reached
          the node limit, so we can
          execute more sequences in
          parallel*/
60        {
61          add a to B;
62          w = w + w_A;
63        }
64        else
65        {
66          execute sequences in B
67          concurrently across j
68          nodes;
69          clear(B);
70          w <- 0;
71        }
72     }
73
       /*Optimization*/
74    execute the final batch of
75    sequences in B across j nodes;
      /*i.e. the sequences whose
      node count did not add up to w
      in the final batch*/
76 }
```

Figure 4.19: TOSS: Throughput-Oriented Sequence Scheduler.

Experimental Results We validate the TOSS algorithm on the scaled-down nucleotide (NT) database as discussed in the previous section, but make use of all the sequence size ranges because we are now dealing with multiple Cell nodes and can align sequences of arbitrary lengths.

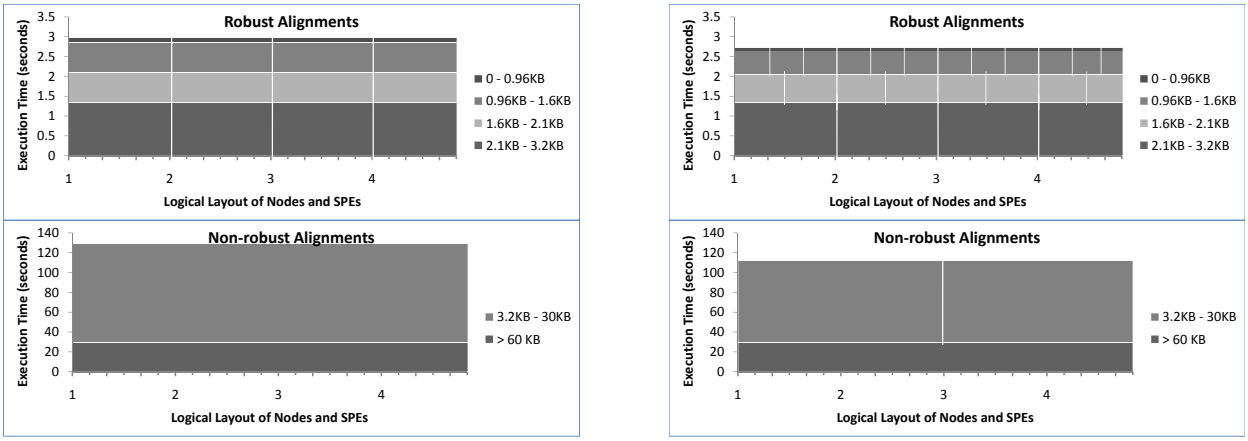
We executed TOSS from Figure 4.19 separately on the dual-Cell QS20 *Cellbuzz* cluster at Georgia Tech, and on the dedicated 20-node PS3 cluster at Virginia Tech. While we had dedicated access to the entire PS3 cluster, we could manage to access only four nodes of the Cellbuzz cluster for the purposes of this experiment, due to the public and busy nature of the Cellbuzz resource at Georgia Tech. To provide a fair comparison of the performances between the two clusters, we report the results of our validation experiment for up to four nodes on both the platforms. We compared the performance of the TOSS scheme against the naïve first-come-first-served (FCFS) sequence scheduling scheme on each of the above mentioned platforms.

TOSS uses spatial partitioning of the computing resources to enable concurrent sequence alignments, while the FCFS scheme utilizes the maximum available computing resources for every sequence alignment. Figures 4.20 and 4.21 compare the performances of our scheduling algorithm against the naïve FCFS scheduler on both the PS3 and QS20 Cell clusers. The horizontal axis of the charts provides the logical view of the layout of the computing elements at two levels of processor granularity - the Cell nodes and the SPEs within them. The vertical axis represents the cumulative execution time taken by the corresponding scheduling algorithm. Each block in the chart area denotes a category of sequence alignments based on the size of the sequence strings, as indicated by the legend on the right. The width of each block represents the range of processing elements that collectively process the corresponding sequence alignments, and the height of the blocks denote the time taken to align the sequences in the respective category.



(a) (b)

Figure 4.20: Spatial layout of the processing elements on the QS20 cluster Vs. cumulative execution times taken for (a) FCFS and (b) TOSS.



(a) (b)

Figure 4.21: Spatial layout of the processing elements on the PS3 cluster Vs. cumulative execution times taken for (a) FCFS and (b) TOSS.

From the Figures 4.20 and 4.21, we show that on realistic datasets, the QS20 cluster outperforms the PS3 cluster in both the FCFS and the TOSS schemes due to the ready availability of more SPE cores per Cell node, which is the true workhorse of the parallel Smith-Waterman algorithm. Also, the two PPE cores on the dual-Cell QS20 blade support the

execution of up to four concurrent MPI processes, while the PPE core on the PS3 can handle only up to two simultaneous thread excutions. However, the TOSS algorithm in conjunction with our execution models for general wavefront algorithms capture the underlying hardware characteristics accurately and enhances the throughput of sequence search by about 33.5% on the QS20 Cell cluster, and by about 13.5% on the PS3 cluster, when compared against the naïve FCFS scheduler.

4.2 CUDA-SWat: Smith-Waterman on the CUDA platform

4.2.1 Experimental Platform

Our experiments were conducted on a host machine with a dual-core Intel processor, each core operating at 2.2GHz and running Ubuntu 4.1.2 with the Linux kernel version 2.6.20-16. The machine is provided with 4 GB of main memory. The device chosen was the nVIDIA GeForce 8800 GTS 512MB graphics unit and the architectural details are already discussed in Chapter 3. We used CUDA version 1.1 as the software interface for developing the parallel Smith-Waterman for the nVIDIA GeForce card.

4.2.2 Design, Implementation and Results

The parallelization strategy for CUDA-SWat leverages the lessons learnt from the design of Cell-SWat and helps us to choose key optimization techniques over the others. We will begin with the serial implementation and its performance for different combinations of the input. We then present a series of five optimization techniques and compare the performance of

each of the schemes against the baseline naïve implementation.

Naïve Implementation The nVIDIA GeForce card contains 512MB of global memory which restricts the size of the matrix and hence limits the maximum sequence length to 6KB. For all the tests, we chose eight randomly generated sequence pairs of sizes varying from 1 KB to 6 KB, thus covering most of the realistic sequence sizes as discussed in Section 4.1.2. Figure 4.22 shows the exeution times for the various sequence sizes.

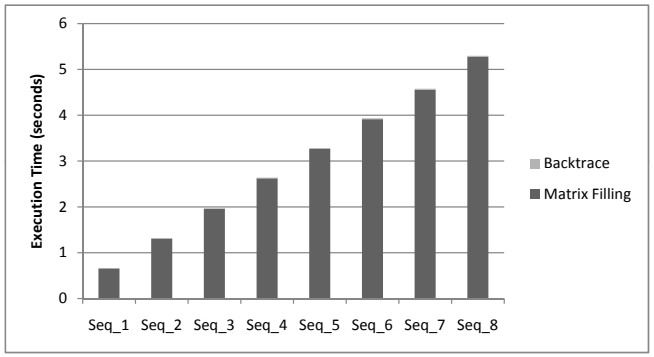


Figure 4.22: Chart showing execution times for aligning 8 different sequence sizes (Naïve Implementation).

Diagonalized Data Layout: Host

The first level of optimization is to arrange the data in a way that is suitable for vector processing as shown in Figure 4.6 in Section 4.1.2. We then execute the Smith-Waterman algorithm directly on the host and observe the results as seen in Figure 4.23.

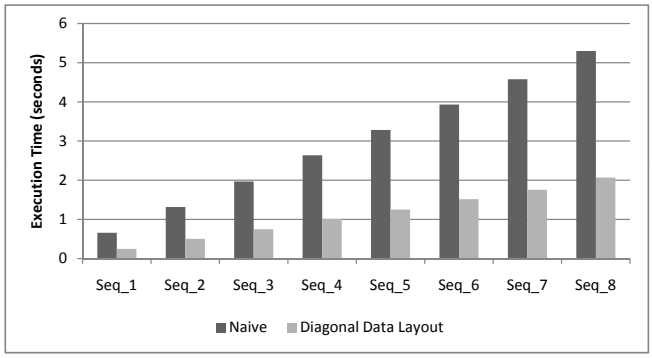


Figure 4.23: Chart comparing the performance of the Naïve Implementation and the Diagonalized Data Layout scheme on the host memory.

Diagonalized Data Layout: Kernel Offload

The next step in the optimization process is to offload the matrix filling part onto the device to efficiently extract the parallelism of the many-core architecture. We follow the wavefront pattern where anti-diagonals are filled by the device starting from the northwest corner to the southeast corner of the matrix. We launch the kernel as 1-Dimensional grid of thread blocks, where each block further contains a single dimension of threads as shown in Figure 4.24. We distribute the elements on each anti-diagonal among the total threads in the grid of thread blocks. The dependence between consecutive anti-diagonals force a synchronization operation after computing each anti-diagonal. However, only the threads within a block can be synchronized and communication between thread blocks is not currently supported by CUDA. All the global memory transactions by the device threads can safely be assumed to be complete only when the kernel exits from the device. This forces us to launch one kernel for each anti-diagonal of the matrix. Figure 4.25 shows that a maximum of 3.2× speedup over the naïve implementation can be achieved by offloading the kernel to the device in conjunction with the diagonalized data representation. This approach achieves a

24% improvement over the host-run implementation that was discussed in the previous unit.

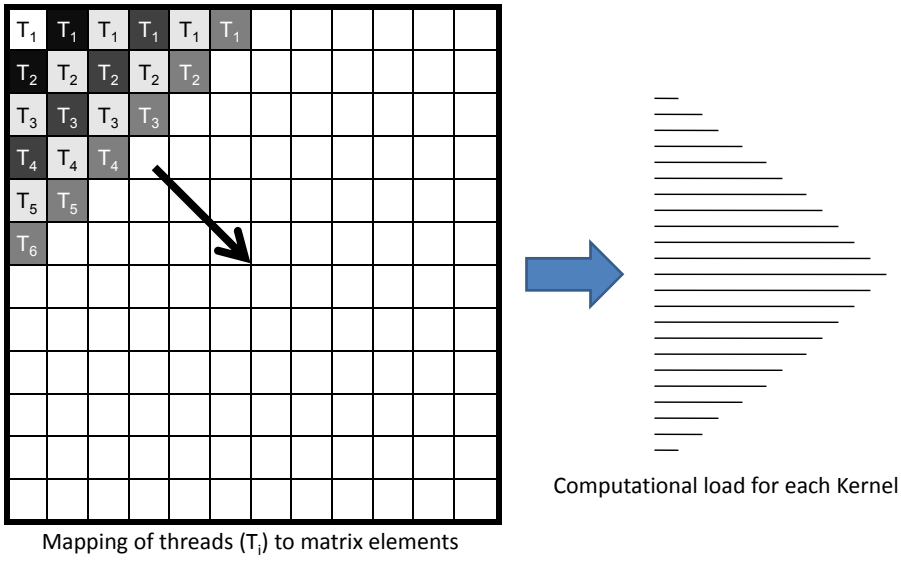


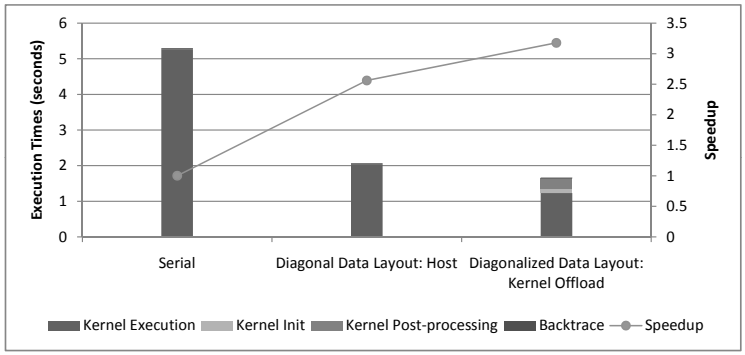
Figure 4.24: (Left) Mapping of threads to matrix elements and the (Right) variation of the computational load that is imposed on successive kernels. It also denotes the non-coalesced data representation of successive anti-diagonals in memory.

Once the entire matrix if filled in the device memory, it is completely transferred to the host memory before the backtrace operation is performed.

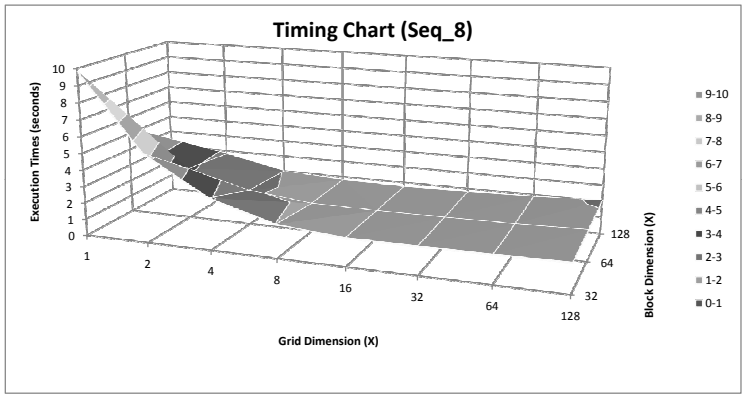
Coalesced Global Memory Access

Although the previous optimization shows some improvement over the host-run code, the approach is still plagued with two major problems:

- Multiple kernel launches – A kernel launch per anti-diagonal means that a typical application will have thousands of kernel invocations. We developed a microbenchmark



(a)



(b)

Figure 4.25: (a) Chart comparing the performance of the Diagonalized Data Layout scheme on the device memory with the other two methods and (b) Execution times for various execution configurations.

that launches empty kernels multiple times onto the device to characterize the effect of kernel invocations in an application. The results show that, for the chosen problem sizes, around 41% of the total execution time is currently being spent in kernel launch alone.

- Non-coalesced global memory access: The CUDA Programming Guide [38] specifies

that the effective global memory bandwidth significantly depends on the memory access pattern because global memory is not cached unlike the other memory spaces on the device.

We deal with the non-coalesced global memory access in this unit and discuss solutions for the multiple kernel launch problem in the next unit.

Coalesced memory accesses require that (1) only 32-bit, 64-bit, or 128-bit words should be read from global memory into the registers by each thread, (2) the global memory addresses simultaneously accessed by consecutive threads during the execution of a single read or write instruction should be arranged so that the memory accesses can be coalesced into a single contiguous, aligned memory access and (3) when accessing x-byte words from global memory, the address location that is accessed by the thread with ID $= 0$ should be a multiple of $16 \times x$. The bandwidth for non-coalesced global memory accesses is found to be around an order of magnitude lower than for coalesced accesses when the accesses are 32-bit words.

Since we currently launch separate kernels to compute every anti-diagonal and we are accessing integers (4-byte words), we make sure that the start address of every anti-diagonal is aligned to 64-byte boundaries to ensure that all the writes are coalesced as shown in Figure 4.26. The skewed dependence between the elements of neighboring anti-diagonals restrict the degree of coalescing among the reads from global memory. Moreover, we make sure that the dimension of the blocks and grid of blocks are all powers of 2, to enable all the blocks to enjoy coalesced memory accesses.

Figure 4.27 shows that a maximum of 3.52× speedup over the naïve implementation can be achieved by coalescing the global memory accesses in conjunction with the previously discussed optimizations. This approach achieves a 11% improvement over the non-coalesced memory access implementation that was discussed in the previous unit.

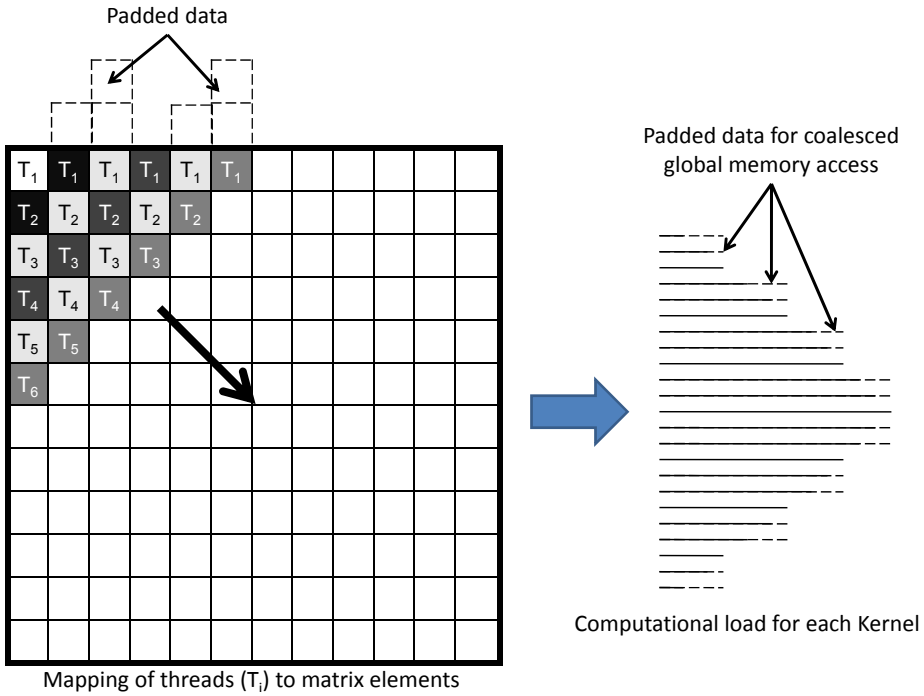
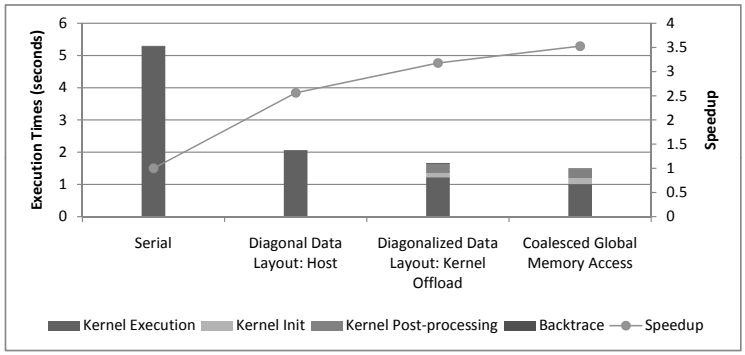


Figure 4.26: (Left) Mapping of threads to matrix elements and the (Right) variation of the computational load that is imposed on successive kernels. It also denotes the *coalesced* data representation of successive anti-diagonals in memory.

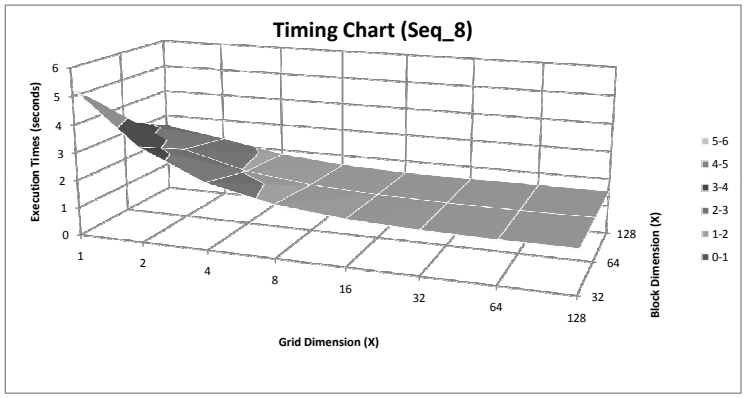
Tiled Wavefront

This unit discusses the problem of multiple kernel launches and our solution to the problem. Microbenchmark results reveal that 41% of the total execution time is spent in kernel invocations. To minimize this overhead, we apply the tiled-wavefront design from Section 4.1.2 to the GPGPU architecture. The differences between the tiled-wavefront approaches followed in Cell-SWat and CUDA-SWat are discussed next.

Tile Scheduling Our scheduling scheme assigns a thread block to compute a tile, and consecutive tile-diagonals are computed one after another from the northwest corner to the



(a)



(b)

Figure 4.27: (a) Chart comparing the performance of the Coalesced Global Memory Access scheme with the other three methods and (b) Execution times for various execution configurations.

southeast corner of the matrix. Entire tile-diagonals are processed at once rather than computing block-rows that is followed by Cell-SWat. Although there are more number of tiles (or thread blocks) to be computed than the number of multiprocessors, the CUDA thread scheduler, which is transparent to the developers, will optimize the thread occupancy on the chip. Hence, our tile scheduling strategy leverages the thread scheduling scheme of

the CUDA library.

Computation-Communication Communication patterns that occur during the tile computation are shown in Figure 4.28. We describe step-by-step communication-computation mechanism performed by each thread block while processing a tile:

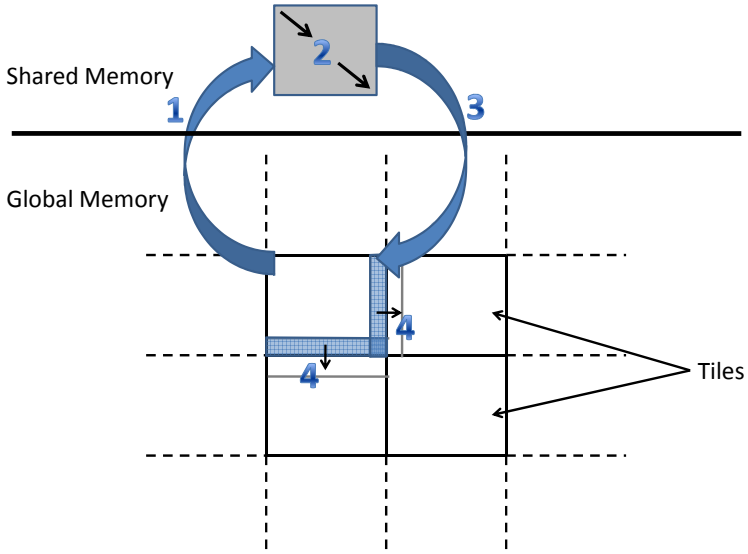


Figure 4.28: Computation-Communication pattern between tiles in global memory (unoptimized).

1. To start computing a tile, a thread block transfers the corresponding elements from global memory to the on-chip shared memory. This memory transfer will be coalesced because we handcraft the allocation of each tile to follow the rules for coalesced memory accesses. The boundary elements that are required for tile computation are assumed to be already present within the tile at this stage. We will later explain how we populate the tile boundary with the appropriate elements.

2. Next, the thread block proceeds with the tile computation within shared memory. The threads use block synchronization primitives after computing every anti-diagonal within the tile.

3. The processed tile is transferred back to its location in global memory.

4. Finally, boundary elements from the current tile are transferred to the boundaries of the neighboring tiles on the west and south. Memory is thus moved within the global memory by a single thread in the block, and is thus a non-coalesced memory transfer.

The above steps describe the processing of non-boundary tiles. Boundary conditions can be easily checked and the redundant steps can be avoided.

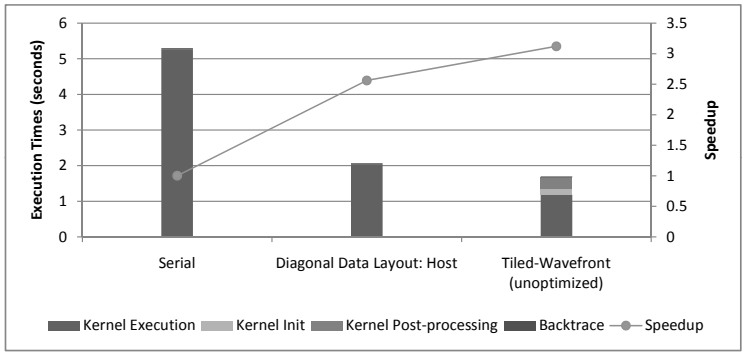
Figure 4.29 shows that a maximum of 3.12× speedup over the naïve implementation can be achieved by applying the tiled wavefront design. This approach achieves a 22% improvement over the host-run implementation with the diagonalized data representation that was discussed before.

Optimized Tiled Wavefront

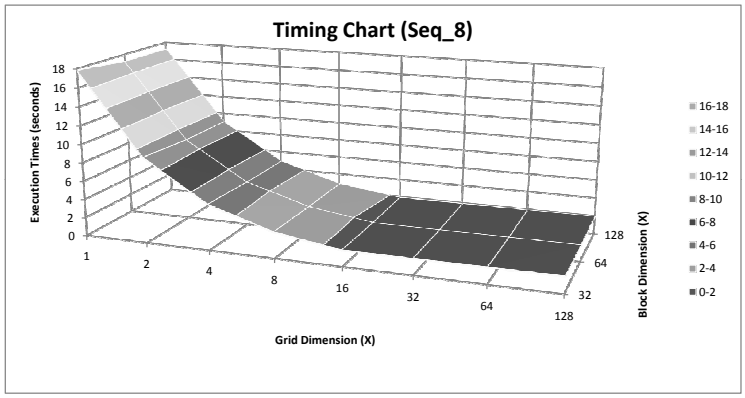
This section talks about how the non-coalesced global memory transfer of the boundary elements between neighboring tiles can be replaced with on-chip buffer copies followed by coalesced global memory transfers as shown in Figure 4.30.

While we retain most of the computation-communication model from the previous section, we modify the final step as follows:

4a. In shared memory, the east and south boundary elements of the current tile are first copied to separate buffers.



(a)



(b)

Figure 4.29: (a) Chart comparing the performance of the Tiled wavefront scheme with the two naïve methods and (b) Execution times for various execution configurations.

4b. The memory allocated for the current tile in shared memory now assumes the role of its east neighbor and all the elements of the tile are initially 'cleared' or assigned the value 0. The east buffer is then copied to the tile's west boundary. The tile (representing the east neighbor) is then OR'ed with the east neighboring tile in global memory and since we are transferring the entire tile, coalesced memory access is assured. The OR operation - in combination with the non-boundary elements of the tile being 0, populates

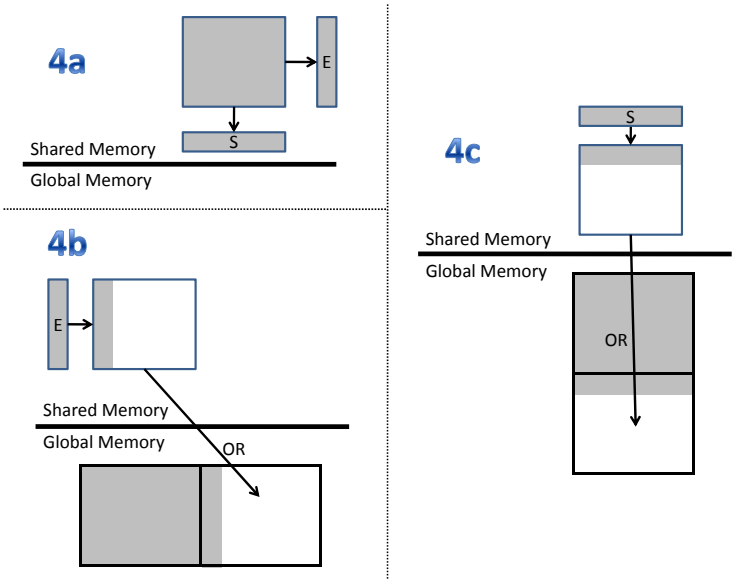


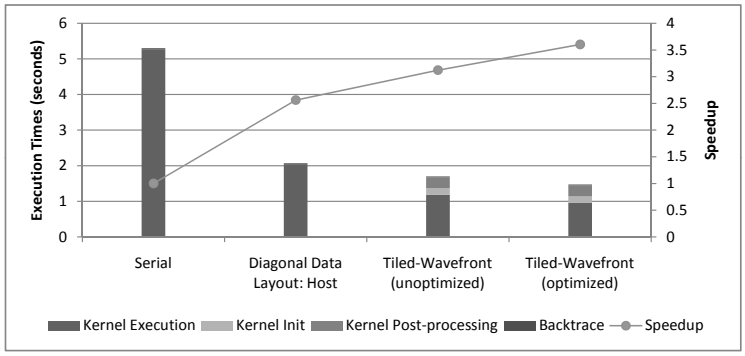
Figure 4.30: Computation-Communication pattern between tiles (optimized).

only the the relevant boundary elements in the east neighbor in global memory.

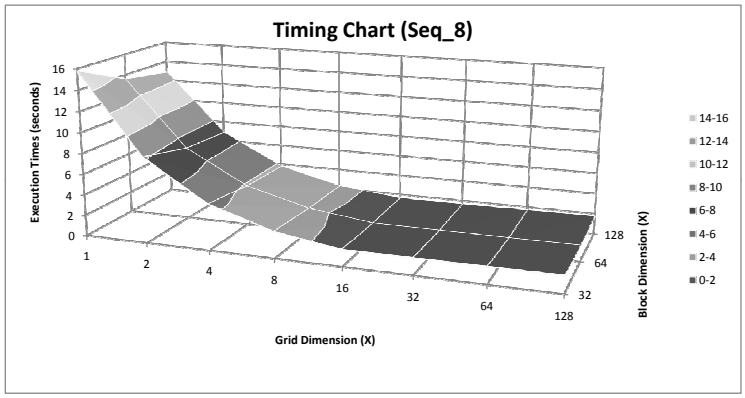
4c. The memory allocated for the current tile in shared memory now assumes the role of its south neighbor and all the elements of the tile are initially 'cleared' or assigned the value 0. The south buffer is then copied to the tile's north boundary. The tile (representing the south neighbor) is then OR'ed with the south neighboring tile in global memory and since we are transferring the entire tile, coalesced memory access is assured. The OR operation - in combination with the non-boundary elements of the tile being 0, populates only the the relevant boundary elements in the south neighbor in global memory.

The above steps convert the non-coalesced memory transfers to coalesced ones. Figure 4.31 shows that a maximum of 3.6× speedup over the naïve implementation can be achieved by performing on-chip buffer copies in conjunction with the tiled wavefront optimization. This

approach achieves a 15.4% improvement over the non-optimized tiled wavefront implementation that was discussed in the previous unit.



(a)



(b)

Figure 4.31: (a) Chart comparing the performance of the Tiled wavefront scheme with the two naïve methods and (b) Execution times for various execution configurations.

4.2.3 Discussion

Figure 4.32 provides all the results from Section 4.2.2 in a combined form for comparing the various optimization approaches against each other before choosing the best strategy.

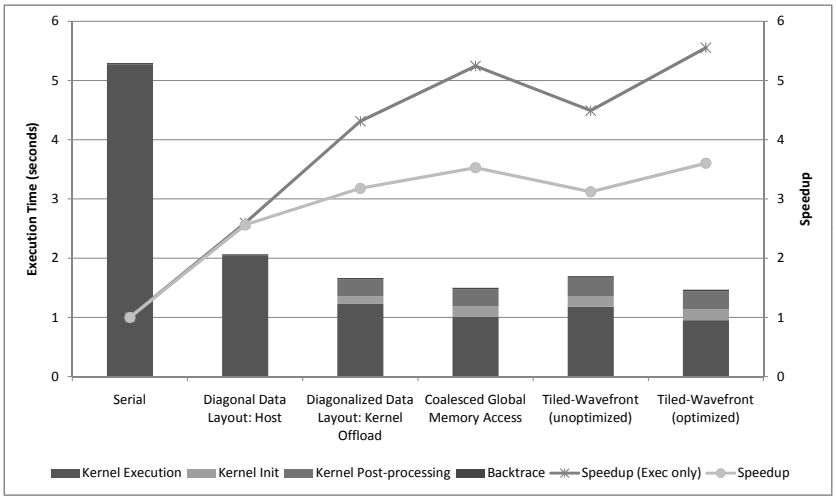


Figure 4.32: Comparison of the benefits of all the discussed optmization techniques.

We see from the figure that although we achieve good speedups when compared to the serial implementation, there is no incremental improvement in the performance when the optimizations are applied in succession. In fact, when we try to reduce the number of kernel launches by tiling the computation (Unoptimized Tiled-Wavefront), we introduce more non-coalesced global memory accesses and hence the performance drops. The optimized tiled wavefront then improves the performance by on chip buffer copies, but the values are comparable to the results of the Coalesced Global Memory Access optimization technique.

Also, the speedup is constant beyond 16 thread blocks for all the optimization techniques. This is because the high utilization of shared resources per block forces only one active block

per multiprocessor, and this means that the performance of a kernel that has more thread blocks than the number of available multiprocessors will not be better than the one with 16 thread blocks (where the chip has 16 multiprocessors).

Moreover, the kernel pre- and post-processing times amount to 35% of the total execution time, and this is a severe, but unavoidable cost. The backtrace operation has negligible effects as expected.

We attribute the moderate performance of CUDA-SWat to the inherent problem-architecture combination – the nVIDIA GPGPU is best suited for data-parallel applications, while the Smith-Waterman algorithm has data parallelism only withing each anti-diagonal. However, there are some unaddressed optimization problems that can improve CUDA-SWat further, which we consider as future work:

- The Smith-Waterman algorithm calls the find_max function multiple times during its execution. Currently each thread compares the calculated value against the value in a location in global memory that is unique to the thread, and updates the global memory variable with the maximum of the two values. The above transaction consists of multiple non-coalesced global memory reads and writes, which causes a loss in the performance.

- The shared memory access patterns should be investigated further to remove memory bank conflicts, if any.

Chapter 5

Conclusion

5.1 Concluding Remarks

With the Cell Broadband Engine and the GPGPU platforms delivering unprecedented high performance within a single chip, and making rapid strides toward the commodity processor market, they are widely expected to replace the multi-core processors in the existing high-performance computing infrastructures, such as large scale clusters, grids and supercomputers. This motivation, in conjunction with the ever growing need for speed in the homology search domain has led us to come up with highly efficient mappings of the optimal Smith-Waterman wavefront algorithm separately on to the nVIDIA GeForce 8800 GTS 512MB card and a cluster of the Cell hybrid multicore processors. We leveraged the generality of our *tiled-wavefront* design for executing parallel wavefront computations *within* a Cell node and recursively applied the same design to every granularity of parallelism in the Cell cluster. However, we traded-off robustness for speed when the sequences were aligned across the Cell nodes. We exploited the multiple layers of parallelism within the platforms to achieve near-linear scalability on both the IBM QS20 and the PS3 Cell platforms, and a max-

imum speedup of 3.6× on the GPGPU. We incrementally optimized the Smith-Waterman code in five stages to enhance its performance on the GPGPU.

We also presented a highly accurate analytical model to estimate the execution times of parallel sequence alignments, and wavefront algorithms in general, within and across multiple Cell nodes. We achieve an error rate of less than 3% for sequence alignments within a Cell node and error rates of less than 10% for alignments across Cell nodes on an average. Finally, we presented TOSS – a Throughput Oriented Sequence Scheduler, that achieved an improved sequence throughput of of 33.5% on the QS20 Cell cluster, and 13.5% on the PS3 cluster, over the naïve FCFS scheduler.

5.2 Future Work

We intend to investigate the integration of our Smith-Waterman implementation on the Cell and GPGPU into existing sequence alignment toolkits. We would also like to extend our modeling and implementation methodologies to other wave-front algorithms in general. On the GPGPU front, we plan to explore further optimization techniques to parallelize sequence search algorithms within and across multiple GPGPU cards. We then want to understand the design and scalability issues of exploiting multigrain parallelism that is available on the emergent CMP architectures to efficiently speedup other sequence search algorithms, such as BLAST and PatternHunter.

Appendix A

Performance Metrics of Sequence Search Algorithms

In this section, we define and discuss the metrics that we use to estimate the performance of sequence-search algorithms in general.

Sequence-search algorithms can be measured on many dimensions such as execution time (or speed), sensitivity, complexity of implementation, and cost of deployment. While measuring the execution time of an algorithm and estimating the cost of deployment of the complete system are straightforward, there is no existing definition that clearly defines and quantifies the 'sensitivity' of a sequence-search algorithm, which we discuss below.

A.1 Sensitivity of Sequence Search

Previous work defined and measured sensitivity in an unconvincing and informal fashion [33, 30]. To address this, we propose a formal definition for sensitivity in the following way.

Homology search methods are similar to web search algorithms. In the web search domain, an input query or keyword is searched against a large known document collection. The output will be a set of relevant web pages that are sorted by closeness or rank. Similarly, in the realm of homology search, an input query sequence is searched against a large known sequence database. The output will be a set of relevant sequences similar to the query, which are sorted by the alignment score or corresponding statistical quantifiers such as E-Value and P-Value of the alignment. Given the analogy between the homology search and web search methods, we first explore some of the many definitions and metrics that have been proposed to measure the performance of information retrieval systems. We then analyze their relevance to sequence search, and later, modify and adapt those definitions in order to quantify sensitivity. The information retrieval metrics that are of interest are as follows:

- *Precision:* Among all the retrieved documents, the fraction of documents that is relevant to the user's information need is termed as *precision*. It gives an indication of the percentage of false-positives that are included in the final result set of the search.
- *Recall:* Among all the documents relevant to the query, the fraction of the documents that is successfully retrieved is termed as *recall* and can be represented by Equation (A.1). It gives an indication of the percentage of false-negatives that are not included in the final result set of the search. In other words, recall denotes the power of the search algorithm to retrieve all the relevant documents.

$$recall = \frac{|all\ relevant\ documents \bigcap retrieved\ documents|}{|all\ relevant\ documents|} \tag{A.1}$$

Relevance to homology search: With respect to sequence-search algorithms, the universal relevant document group or the *absolute result set* corresponds to all the sequence alignments that are generated by the optimal Smith-Waterman algorithm, for a given threshold score.

Different threshold scores generate different absolute result sets of sequence alignments. False positives are created by assigning a score that is greater than the optimum to a typically low-scoring (irrelevant) alignment, which may cause the irrelevant alignment to cross the threshold score and appear in the final result set. However, no sequence-search algorithm assigns a score that is higher than the optimum to any alignment. Therefore, false positives cannot be generated by this class of algorithms, thus eliminating 'precision' as a relevant metric to compare sequence-search algorithms.

False negatives, on the other hand, can be generated by those heuristic algorithms that are willing not to output some high-scoring (relevant) sequences in order to obtain large speed improvements. This is typically the case with heuristics such as BLAST, FASTA, and PatternHunter, for example. Therefore, we consider 'recall' as a relevant metric to compare homology search methods. With this background, we can now define and quantify the term *sensitivity*.

Definition Among all the sequence alignments that are generated by the Smith-Waterman algorithm for a given threshold score, the fraction of the alignments that is successfully generated for the same threshold score by the algorithm under test is denoted as the *sensitivity* of that algorithm for that threshold score.

Let χ represent the set of scores of all the statistically significant alignments[1] that are generated by the Smith-Waterman algorithm. If we consider each element of the set χ as a potential threshold score, then the sensitivity of the test algorithm at the different threshold scores in χ can be represented by the Equation (A.2).

$$sensitivity_i = \frac{|S_i \bigcap T_i|}{|S_i|} \tag{A.2}$$

[1]The statistical significance of an alignment can be inferred by examining the corresponding E-values and P-values.

where,

$$\begin{aligned} i &\in \chi, \text{ the set of threshold scores} \\ sensitivity_i &= \text{sensitivity at the threshold score } i \\ S_i &= \text{result set generated by Smith-Waterman with alignment scores} \geq i \\ T_i &= \text{result set generated by the test algorithm with alignment scores} \geq i \end{aligned}$$

Since no sequence-search algorithm generates false positives, the result set generated by the test algorithm is contained in the absolute result set generated by Smith-Waterman, i.e. $T_i \subseteq S_i$. Therefore, Equation (A.2) becomes

$$sensitivity_i = \frac{|T_i|}{|S_i|} \tag{A.3}$$

Sensitivity is therefore a function of the threshold score. To assign a unified sensitivity value to a sequence-search algorithm, we take the mean of sensitivity values at all the threshold scores in χ, as shown in Equation (A.4). Empirical results show that the sensitivity values for different threshold scores have very low variance [6], and therefore, their mean value gives a good estimate of the sensitivity of the algorithm.

$$Sensitivity = \frac{\sum\limits_{i \in \chi} sensitivity_i}{|\chi|} \tag{A.4}$$

Therefore, the target for any sequence-search algorithm is to provide a result set that is identical to that of Smith-Waterman, thereby achieving a perfect sensitivity of 1. If the sensitivity value is less than 1, it means that the sequence-search algorithm has *missed* generating significant alignments.

Bibliography

[1] Ashwin M. Aji, Wu chun Feng, Filip Blagojevic, and D. S. Nikolopoulos. Cell-SWat: Modeling and Scheduling Wavefront Computations on the Cell Broadband Engine. In *Proc. of the ACM International Conference on Computing Frontiers*, May 2008.

[2] S. Alam, J. Meredith, and J. Vetter. Balancing Productivity and Performance on the Cell Broadband Engine. In *Proc. of the 2007 IEEE Annual International Conference on Cluster Computing*, September 2007.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J Mol Biol*, 215(3):403–410, October 1990.

[4] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped blast and PSI–BLAST: a new generation of protein database search programs. *Nucleic Acids Res.*, 25:3389–3402, 1997.

[5] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[6] Ashwin M. Aji and Wu-chun Feng. Revisiting the Speed-versus-Sensitivity Tradeoff in Pairwise Sequence Search. Technical Report TR-08-07, Department of Computer Science, Virginia Tech, March 2008.

[7] David Bader and Virat Agarwal. FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine. In *Proc. of the 14 th IEEE International Conference on High Performance Computing (HiPC), Lecture Notes in Computer Science 4873*, 2007.

[8] David A. Bader, Virat Agarwal, and Kamesh Madduri. On the design and analysis of irregular algorithms on the cell processor: A case study of list ranking. In *Proc. of the 21st International Parallel and Distributed Processing Symposium*, pages 1–10, 2007.

[9] Pieter Bellens, Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. Memory - cellss: a programming model for the cell be architecture. In *Proc. of Supercomputing'2006*, page 86, 2006.

[10] R.D. Bjornson, A.H. Sherman, S.B. Weston, N. Willard, and J. Wing. Turboblast : a parallel implementation of blast built on the turbohub. *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 183–190, 2002.

[11] F. Blagojevic, A. Stamatakis, C. Antonopoulos, and D. S. Nikolopoulos. RAxML-CELL: Parallel Phylogenetic Tree Construction on the Cell Broadband Engine. In *Proc. of the 21st IEEE/ACM International Parallel and Distributed Processing Symposium*, March 2007.

[12] Filip Blagojevic, Matthew Curtis-Maury, Jae seung Yeom, Scott Schneider, and Dimitrios S. Nikolopoulos. Scheduling asymmetric parallelism on a playstation3 cluster. The 8th IEEE International Symposium on Cluster Computing and the Grid, Lyon, France, May 2008.

[13] Filip Blagojevic, Dimitris S. Nikolopoulos, Alexandros Stamatakis, and Christos D. Antonopoulos. Dynamic multigrain parallelization on the cell broadband engine. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 90–100, New York, NY, USA, 2007. ACM.

[14] Azzedine Boukerche, Alba Cristina Magalhaes Alves de Melo, Mauricio Ayala-Rincon, and Thomas M. Santana. *Experimental and Efficient Algorithms*, chapter Parallel Smith-Waterman Algorithm for Local DNA Comparison in a Cluster of Workstations, pages 464–475. 2005.

[15] G. Buehrer and S. Parthasarathy. The Potential of the Cell Broadband Engine for Data Mining. Technical Report TR-2007-22, Department of Computer Science and Engineering, Ohio State University, 2007.

[16] M. Cameron, H.E. Williams, and A. Cannane. Improved gapped alignment in blast. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 1(3):116–129, July-Sept. 2004.

[17] Thomas Chen, Ram Raghavan, Jason Dale, and Eiji Iwata. Cell broadband engine architecture and its first implementation. *IBM developerWorks*, Nov 2005.

[18] M.O. Dayhoff, R.M. Schwartz, and B. C. Orcutt. *Atlas of protein sequence and structure*, volume 5, chapter A model of evolutionary change in proteins., pages 345–352. National Biomedical Research Foundation, Silver Spring, Maryland, 1978.

[19] A. E. Eichenberger et al. Using Advanced Compiler Technology to Exploit the Performance of the Cell Broadband Engine Architecture. *IBM Systems Journal*, 45(1):59–84, 2006.

[20] J. A. Kahle et al. Introduction to the Cell multiprocessor. In *IBM Journal of Research and Development*, pages 589–604, Jul-Sep 2005.

[21] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Memory - sequoia: programming the memory hierarchy. In *Proc. of Supercomputing'2006*, page 83, 2006.

[22] J.; Heshan Lin; Xiaosong Ma Gardner, M.K.; Wu-chun Feng; Archuleta. Parallel genomic sequence-searching on an ad-hoc grid: Experiences, lessons learned, and implications. *Supercomputing, 2006. SC '06. Proceedings of the ACM/IEEE SC 2006 Conference*, pages 22–22, 11-17 Nov. 2006.

[23] Bugra Gedik, Rajesh Bordawekar, and Philip S. Yu. Cellsort: High performance sorting on the cell processor. In *Proc. of the 33rd Very Large Databases Conference*, pages 1286–1207, 2007.

[24] Sandor Heman, Niels Nes, Marcin Zukowski, and Peter Boncz. Vectorized Data Processing on the Cell Broadband Engine. In *Proc. of the Third International Workshop on Data Management on New Hardware*, June 2007.

[25] Steven Henikoff and Jorja G. Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences of the United States of America*, 89(22):10915–10919, 1992.

[26] A. Hoisie, O. Lubeck, and H. Wasserman. Performance and Scalability Analysis of Teraflop-scale Parallel Architectures using Multidimensional Wavefront Applications. Technical Report LAUR-98-3316, Los Alamos National Laboratory, August 1998.

[27] R. Hughey. Parallel hardware for sequence comparison and alignment, 1996.

[28] W. J. Kent. Blat–the blast-like alignment tool. *Genome Res*, 12(4):656–664, April 2002.

[29] D. Lavenier. Dedicated hardware for biological sequence comparison. 2(2):77–86, 1996.

[30] M. Li, B. Ma, D. Kisman, and J. Tromp. Patternhunter ii: Highly sensitive and fast homology search. 2003.

[31] D. J. Lipman and W. R. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, March 1985.

[32] Weiguo Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig. Bio-sequence database scanning on a gpu. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8 pp.–, 25-29 April 2006.

[33] B. Ma, J. Tromp, and M. Li. Patternhunter: faster and more sensitive homology search. 2002.

[34] Svetlin A Manavski and Giorgio Valle. Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC Bioinformatics*, 2008.

[35] Michael Gschwind and Bruce D'Amora and Alexandre Eichenberger. Cell BE enabling density computing for data rich environments. *URL:http://www.research.ibm.com/people/m/mikeg/papers/cell_isca2006.pdf. Accessed: 2008-08-06. (Archived by WebCite at http://www.webcitation.org/5ZrdhV79j).*

[36] Chris Mueller, Ben Martin, and Andrew Lumsdaine. CorePy: High-Productivity Cell/B.E. Programming. In *Proc. of the First STI/Georgia Tech Workshop on Software and Applications for the Cell/B.E. Processor*, June 2007.

[37] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, March 1970.

[38] NVIDIA Corporation. NVIDIA CUDA Programming Guide. *URL: http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf. Accessed: 2008-08-06. (Archived by WebCite at http://www.webcitation.org/5ZrcTQQeV)* .

[39] James Ostell. Databases of Discovery. *ACM Queue*, 3(3), April 2005.

[40] Fabrizio Petrini, Gordon Fossum, Juan Fernández, Ana Lucia Varbanescu, Michael Kistler, and Michael Perrone. Multicore surprises: Lessons learned from optimizing sweep3d on the cell broadband engine. In *Proc. of the 21st International Parallel and Distributed Processing Symposium*, pages 1–10, 2007.

[41] Vipin Sachdeva, Michael Kistler, William Evan Speight, and Tzy-Hwa Kathy Tzeng. Exploring the viability of the cell broadband engine for bioinformatics applications. In *Proc. of the 6th IEEE International Workshop on High Performance Computational Biology*, 2007.

[42] Tf Smith and Ms Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147:195–197.

[43] John E. Stone, James C. Phillips, Peter L. Freddolino, David J. Hardy, Leonardo G. Trabuco, and Klaus Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, 28:2618–2640, 2007.

[44] TimeLogic Biocomputing Solutions. DeCypherSW. *URL:http://www.timelogic.com/downloads/decyphersw.pdf. Accessed: 2008-02-02.(Archived by WebCite at http://www.webcitation.org/5VK0AyWiI).*

[45] Ana Lucia Varbanescu, Henk J. Sips, Kenneth A. Ross, Qiang Liu, Lurng-Kuo Liu, Apostol Natsev, and John R. Smith. An effective strategy for porting c++ applications

on cell. In *Proc. of the 2007 International Conference on Parallel Processing*, page 59, 2007.

[46] Yoshiki Yamaguchi, Tsutomu Maruyama, and Akihiko Konagaya. High speed homology search with fpgas.

[47] John Johnson Yang Liu, Wayne Huang and Sheila Vaidya. Gpu accelerated smith-waterman. In Peter M.A. Sloot Vassil N. Alexandrov, Geert Dick van Albada and Jack Dongarra, editors, *Computational Science – ICCS 2006*, volume 3994 of *LNCS*, pages 188–195. Springer, 2006.

[48] Fa Zhang, Xiang-Zhen Qiao, and Zhi-Yong Liu. A parallel smith-waterman algorithm based on divide and conquer. *Algorithms and Architectures for Parallel Processing, 2002. Proceedings. Fifth International Conference on*, pages 162–169, 2002.