FASTER IMPLEMENTATION OF PROTEIN FOLDING ALGORITHM AND EXTENDED BURROWS WHEELER TRANSFORM

A DISSERTATION

Submitted in partial fulfillment of the requirements for the award of the degree
of
MASTER OF TECHNOLOGY
in
COMPUTER SCIENCE AND ENGINEERING

By

KHALIL SAWANT

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE -247 667 (INDIA)
JUNE, 2009

Candidate Declaration

I hereby declare that the work being presented in the dissertation report titled "**Faster Implementation of Protein Folding Algorithm and Extended Burrows Wheeler Transform**" in partial fulfillment of the requirement for the award of the degree of Master of Technology in Computer Science and Engineering, submitted in the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, is an authentic record of my own work carried out under the guidance of Dr Ankush Mittal and Dr Rajdeep Niyogi, in the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee. I have not submitted the matter embodied in this dissertation report for the award of any other degree.

Dated: 18 JUN 2009

Place: IIT Roorkee.

Khalil BashirAhmed Sawant

Certificate

This is to certify that above statements made by the candidate are correct to the best of our knowledge and belief.

Dated: 18 JUN 2009

Place: IIT Roorkee.

Dr. Ankush Mittal,
Associate Professor,
Department of Electronics
and Computer Engineering.

Dr Rajdeep Niyogi
Assistant Professor,
Department of Electronics
and Computer Engineering.

Acknowledgements

First of all and foremost, I would like to express my deep sense of gratitude and indebtedness to my guide Dr. Ankush Mittal, for his invaluable guidance and constant encouragement throughout the dissertation. His zeal for getting the best out of his students helped me to perform above my par.

I am also grateful to my co-guide Dr Rajdeep Niyogi, for his help with the algorithmic aspects of my dissertation, especially with the formalization and validation of my proposed algorithm.

I would want to express thanks to my colleagues, Salil Sahasrabudhe, Tarun Kumar Gautam, Kshitiz Gupta, Nityam Parakh, Payas Goyal and Binay Kumar Pandey, for their "taken for granted" help with trivial matters, without which I am sure my work might have hit a dead end.

I would like to thank Matthias Schubert and Dominik Buchetmann for developing the uni-processor implementation of the original Protein Folding algorithm that I have modified and parallelized in this work. I would also like to thank Peter Clote and Rolf Backofen for making available the code online.

I also acknowledge Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and the National Science Foundation, for the use of Cell Broadband Engine resources that have contributed to this research.

Khalil BashirAhmed Sawant

Abstract

Over the recent years, there has been an extensive development in the field of bioinformatics. A couple amongst the various works done under this field includes Protein Folding and Genome based Phylogenetic Studies. Protein folding is the physical process by which a polypeptide folds into its characteristic and functional three-dimensional structure. The problem is inherently intractable and hence, non-analytical alternatives for solving the problem exist. Even such alternatives are computationally intensive due to the inherent vastness of the search space. Genome based Phylogenetic studies include the process of matching Mitochondrial DNA of different species to establish their phylogenetic relation. One novel algorithm to achieve this is the Extended Burrows Wheeler Transform. This algorithm also is very compute intensive due to size of mitochondrial genomes used as data. All this necessitates the optimization of such algorithms by parallelization or other means.

The Sony-Toshiba-IBM Cell Broadband Engine is heterogeneous multi core architecture, consisting of a traditional PowerPC based master core meant to run the operating system, and 8 delegate slave processors built for compute intensive processing. Exposure of system level optimization features allows programmer to use algorithm specific tweaks to achieve order of magnitude improvements using Cell-BE. CUDA is a parallel computing architecture developed by NVIDIA. It is a middle-ware compute engine which exposes the power of NVIDIA Graphics Processing Units to software developers through industry standard programming language.

This work introduces a modification on the traditional Protein Folding Algorithm. It describes the implementation of the modified algorithm on Cell-BE and CUDA. Lastly the work describes the implementation of Extended Burrows Wheeler Transform on CUDA and issues involved.

Table of Contents

List of Figures

List of Tables

List of Publications

- K. Sawant, A. Mittal and R. Niyogi, "Search Space Pruning in Protein Folding Simulation of 2D HP Model using Genetic Algorithms", Third International Conference on Information Processing, Bangalore, Aug 2009.

Chapter 1: Introduction

1.1 Protein Folding

Proteins [1] [2] are the functional work-horses of our bodies. They carry out a varied number of tasks in our body ranging from moving muscles, to carrying oxygen in blood, to fighting infection. The proper functioning of the protein is related to its structure, which in turn in related to its constituent chemical molecules. Since proteins are made up varied combination of only twenty different such chemical molecules (amino acids), the structure is related to the combination sequence of these twenty amino acids. This sequence determines the structure of the protein by means of a process called Protein Folding.

Recent discoveries [1] have shown that if protein gets into an improper structure, it could cause a non-functional protein or worse toxic protein that could poison the cell and cause disease like Alzheimer. All this necessitates the understanding of the relation between the structure and constituent sequence of a protein. Simple analysis of protein folding has shown that it is a very compute intensive process. Hence the need to find means of reducing the runtime of the process. Many variations exist in the computational models for protein folding. We try to focus on one such model, which can give us near correct answers for acceptable running times.

1.2 Phylogenetic Analysis

Every living organism is made up DNA and proteins as constituents of its cells which form the organism basic building block [3]. In addition to these molecular constituents defining the organism's outward appearance and biological functions, they also help biologist ascertain the related-ness and non-related-ness of two organisms. It was observed that organism of different species that closely relate show a great deal of similarity in the molecular structure or sequence of chemical components of these biological constituents of the cell.

One such cell constituents is the mitochondrial DNA (mtDNA) which undergoes mutation over generations. The mtDNA is passed only from maternal side, with no

change except mutation any difference between such mtDNA for two species denotes only the mutations accumulated over the time. Such a comparative analysis of mtDNA, help biologist to arrange various species in a tree format with related species represented as more closer branches. The simplistic means of such analysis would be simple string comparison between the two DNA sequences. Such algorithms generally depend upon the product of length of both sequences for their runtime.

However there exists a novel method of having multiple strings to be compared for similarity together, called the Extended Burrows Wheeler Transform (EBWT). This would take less time than having all combination of simple sequence comparison between each of pair of species. The EBWT however still requires long processing time due to the fact that general mtDNA sequences run in the sizes of thousands. This necessitates the reduction in runtime of the EBWT algorithm.

1.3 Multi-Core Architecture

Moore's law had predicted that the chip manufacturing technology would be able to double the transistors on chip roughly every two years, and the prediction had stood good so far. Microprocessors technology had been using this prediction to improve its frequency by various techniques. However in the recent past micro-processors have hit a frequency wall, and not been able to take advantage of the predicted exponential growth. The outcome is the emergence of multi-core processors, which offer the performance benefits of multi-processors on single chip. The presence of such architectures as common desktop processors has made it possible for hitherto time-consuming algorithms to be solved on simple desktop machines.

Another emerging trend has been the use of Graphics Processing Units (GPUs) for general purpose computing. The GPUs model themselves as multi-core processing and expect programs to take advantage of them as raw parallel number-crunchers. The multi-core processors allow program to leverage their computing power by various means like independent threads per core, or allow user to manipulate efficient data flow between cores, or provide a layer of software which manages the scalability of the cores. With the future micro-processor trends likely to increase number of cores as the only means of

their increasing computing power, it becomes necessary to ensure that important algorithms be parallelized to run on next generation of micro-processors.

Thus multi-core processors provide the perfect means of increasing the runtimes of our protein folding simulation and EBWT.

1.4 Problem Statement

In this dissertation, a variation of the protein folding algorithm was studied and profiled for performance bottlenecks. Since the problem portion of the algorithm caused the runtime of the algorithm to go into hours, it was required to find alternatives to be able to speed up the algorithm. This could include parallelization and/or modifying the algorithm completely. Also the extended Burrows Wheeler Transform gave considerable run times due to length of the input data. The objective was to parallelize the algorithms to achieve running time speedup.

1.5 Organization of the Report

The organization of this dissertation report is as follows:

Chapter 2 covers a detailed explanation of the architecture of Cell Broadband Engine and CUDA programming environment, which have been used in this dissertation.

Chapter 3 starts with the explanation of concepts of protein folding, 2D HP model and genetic algorithms. The chapter then discusses the existing algorithm applying genetic algorithms to 2D HP model, and explains in detail our proposed changes to the algorithm. The chapter then discusses parallelization of the modified algorithm as done on Cell-BE and CUDA and the issues faced therein. The results are also discussed.

Chapter 4 starts with the concepts of Burrows Wheeler Transform and its extended version. The chapter then discusses the implementation of the extended Burrows Wheeler transform on CUDA, and the issues faced therein. The results are also discussed.

Chapter 5 concludes the dissertation report and gives suggestion for future work.

Chapter 2: Parallel Processing Architectures

2.1 Cell-BE History and Motivation

The Cell Broadband Engine [4] (henceforth called Cell-BE) is an output of the collaboration between Sony Computer Entertainment Incorporated (SCEI), Toshiba and IBM, which was started in around the summer of 2000. The idea took root as it was determined by SCEI that the then existing traditional architectural organization would not deliver computational power that SCEI sought for its future interactive needs. The collaboration meant that SCEI would be the content provider, IBM to be the microprocessor developer, and Toshiba would be the high-volume manufacturing partner. By the end of the year, an architectural concept was agreed upon, combining the existing 64-bit Power architecture of IBM, along with memory flow control and synergistic slave processors.

The objectives for the new processor were the following

- Outstanding performance, especially on game / multimedia applications
- Real-time responsiveness to the user and the network
- Applicability to a wide range of platforms

2.2 Challenges for the Cell – BE

The objectives for Cell-BE were likely to be challenged by limitation in performance imposed by three major factors namely memory latency and bandwidth, power dissipation and pipeline throughput.

- Memory Latency or "Memory Wall" refers to the fact that the memory (DRAM) speeds have not improved at the same pace as microprocessor speeds, hence memory access has been the processing bottleneck and the gap is widening. Also since large number of multiple memory access is not possible concurrently in current microprocessors, this latency has not been successfully hidden.
- Power dissipation for semiconductors is increasing with increasing density of transistors on chip. The need to keep this power dissipation under control (or "Power Wall") is a major design consideration for microprocessors.

- "Frequency Wall" or the limit onto increasing clock frequency is caused by diminishing returns from pipelined processors. Simply increasing clock frequency causes reduction in pipeline cycle time and causes difficulty in designing substantial work for each stage.

2.3 Hardware Architecture

The Cell-BE[5] consists of nine processors on a single chip, one master processor called PPE and eight slave processors called SPEs, all connected to each other and to external devices by a high-bandwidth, memory-coherent bus. Figure 2.1 shows a block diagram of Cell-BE.

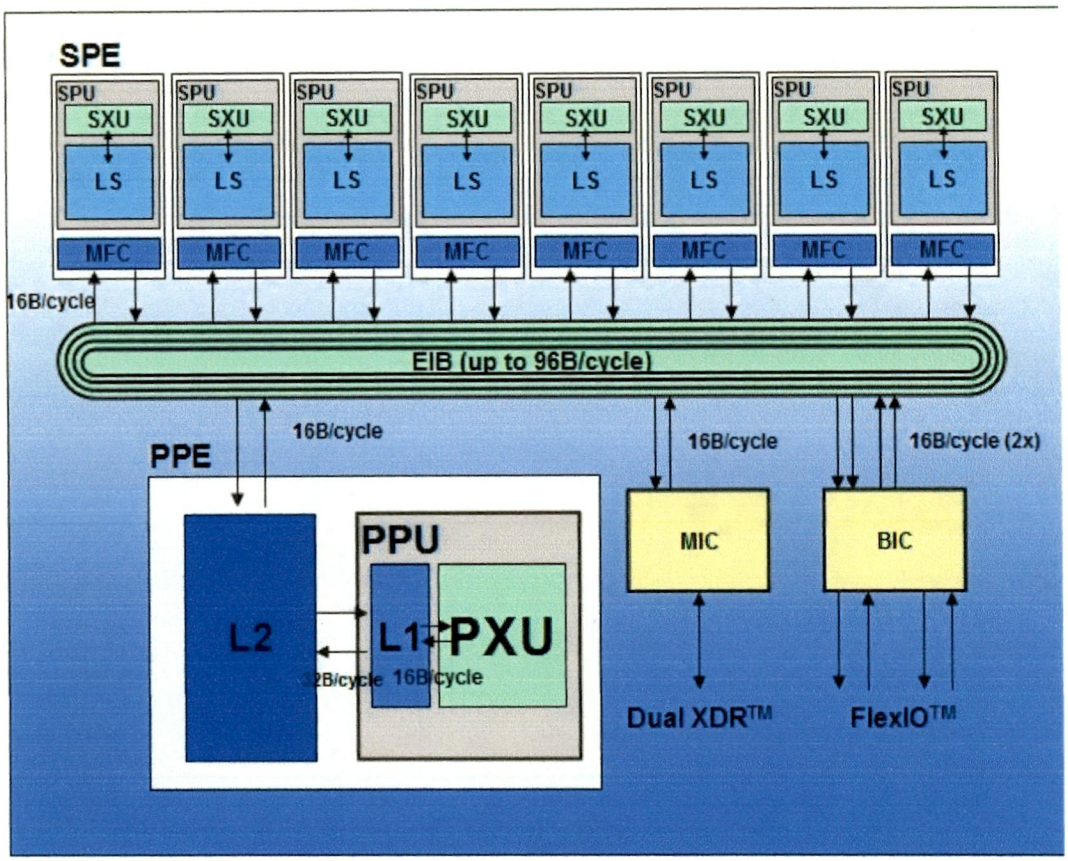

Figure 2.1: The hardware architecture of Cell Broadband Engine[5]

The main components of the architecture are

2.3.1 PowerPC Processor Element (PPE)

The PPE is the main processor. It contains a 64-bit PowerPC architecture based reduced instruction set computer (RISC) core with a traditional virtual-memory subsystem. It runs an operating system, manages system resources, and is intended primarily for control processing, including the allocation and management of SPE threads. It can run legacy PowerPC Architecture software and performs well executing system-control code. It also includes a vector multimedia extension unit, called Single Instruction, Multiple Data (SIMD), so that it can do multiple operations simultaneously with a single instruction. The PPE consists of two main units Power Processor Unit (PPU) and PowerPC Processor Storage Subsystem (PPSS).

The PPU performs instruction execution, and it has level 1 (L1) instruction cache, data cache of 32KB each, and six execution units. The PPU supports two simultaneous threads of execution and can be viewed as a 2-way multiprocessor with shared data-flow. This appears to software as two independent processing units. The PPSS handles memory requests from PPU and external requests to the PPE from SPEs or I/O devices. It has a unified level 2 (L2) instruction and data cache of 512KB.

The primary function of the PPEs is the management and allocation of tasks for the SPEs in a system. When data enters the PPE, this element then distributes it among SPEs, schedules them to be processed on one or more of the SPEs, controls and synchronizes them.

2.3.2 Synergistic Processor Elements (SPEs)

Each of the eight Synergistic Processor Elements (SPEs) is a 128-bit RISC processor specialized for data-rich, compute-intensive SIMD applications. It consists of two main units, the Synergistic Processor Unit (SPU) and the Memory Flow Controller (MFC).

The SPU deals with instruction control and execution. It includes a single register file with 128 registers (each one 128 bits wide), a unified (instructions and data) 256-KB local store (LS), an instruction-control unit, a load and store unit, two fixed-point units, a

floating-point unit, and a channel-and-DMA interface. The SPU implements a new SIMD instruction set, the SPU Instruction Set Architecture, which is specific to the Broadband Processor Architecture.

Each SPU is an independent processor with its own program counter and is optimized to run SPE threads spawned by the PPE. The SPU fetches instructions from its own LS, and it loads and stores data from and to its own LS. With respect to accesses by its SPU, the LS is unprotected and un-translated storage.

2.3.3 Memory Flow Controller (MFC)

The MFC contains a DMA controller that supports DMA transfers. Programs running on the SPU, the PPE, or another SPU, use the MFC's DMA transfers to move instructions and data between the SPU's LS and main storage. (Main storage is the effective-address space as seen by the PPE.) The MFC interfaces the SPU to the EIB, implements bus bandwidth-reservation features, and synchronizes operations between the SPU and all other processors in the system.

To support DMA transfers, the MFC maintains and processes queues of DMA commands. After a DMA command has been queued to the MFC, the SPU can continue to execute instructions while the MFC processes the DMA command autonomously and asynchronously. The MFC also can autonomously execute a sequence of DMA transfers, such as scatter-gather lists, in response to a DMA-list command. This autonomous execution of MFC DMA commands and SPU instructions allows DMA transfers to be conveniently scheduled to hide memory latency. Each DMA transfer can be up to 16 KB in size. Memory-mapped mailboxes or atomic MFC synchronization commands can be used for synchronization and mutual exclusion.

2.4 Programming Features of Cell-BE

2.4.1 SIMD Vectorization:

A vector is an instruction operand containing a set of data elements packed into a one dimensional array. The elements can be integer or floating-point values. Most Vector/SIMD Multimedia Extension and SPU instructions operate on vector operands. Vectors are also called SIMD operands or packed operands. SIMD processing exploits

data-level parallelism. Data-level parallelism means that the operations required to transform a set of vector elements can be performed on all elements of the vector at the same time. That is, a single instruction can be applied to multiple data elements in parallel.

Support for SIMD operations is pervasive in the Cell Broadband Engine. In the PPE, they are supported by the Vector/SIMD Multimedia Extension instruction set. In the SPEs, they are supported by the SPU instruction set. In both the PPE and SPEs, vector registers hold multiple data elements as a single vector. The data paths and registers supporting SIMD operations are 128 bits wide, corresponding to four full 32-bit words. This means that four 32-bit words can be loaded into a single register, and, for example, added to four other words in a different register in a single operation. Similar operations can be performed on vector operands containing 16 bytes, 8 half-words, or 2 double-words. The following figure 2.2 shows such an operation.

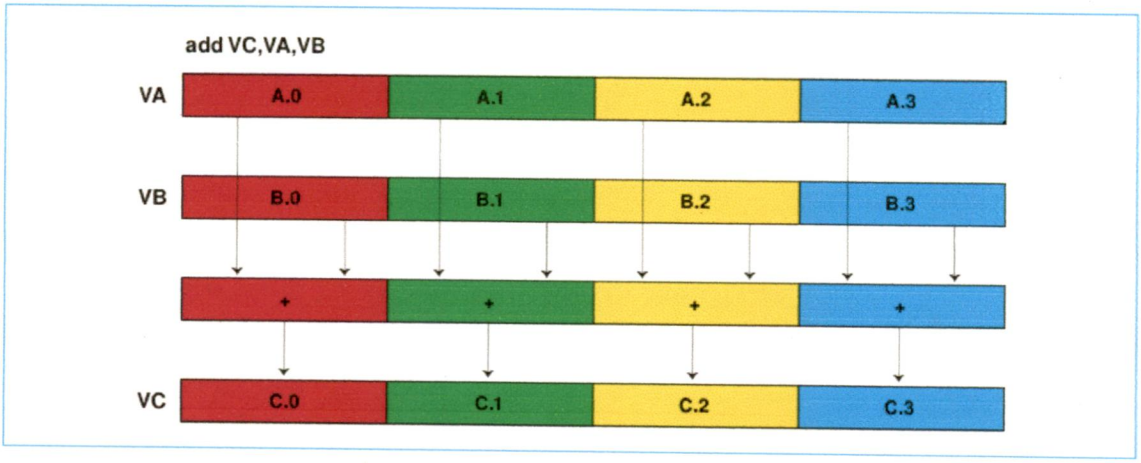

Figure 2.2: SIMD in Cell[5]

2.4.2 DMA and Double Buffering:

MFC supports a set of DMA commands which provide the main mechanism that enables data transfer between the LS and main storage. It also supports a set of synchronization commands which used to control the order in which storage accesses are performed and maintaining synchronization with other processors and devices in the system.

SPE programs use DMA transfers to move data and instructions between main storage and the local store (LS) in the SPE. Consider an SPE program that requires large amounts of data from main storage. The following is a simple scheme to achieve that data transfer:

1 Start a DMA data transfer from main storage to buffer B in the LS.

2 Wait for the transfer to complete.

3 Use the data in buffer B.

4 Repeat.

This method wastes a great deal of time waiting for DMA transfers to complete. We can speed up the process significantly by allocating two buffers, B0 and B1, and overlapping computation on one buffer with data transfer in the other. This technique is called double buffering. The below figure 2.3 shows a flow diagram for this double buffering scheme. Double buffering is a form of multi-buffering, which is the method of using multiple buffers in a circular queue to overlap processing and data transfer.

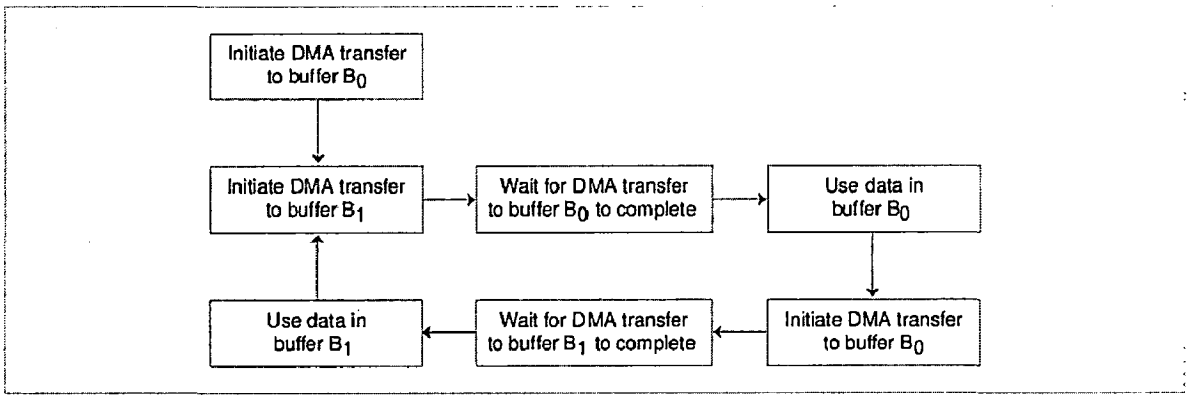

Figure 2.3: Double Buffering [5]

2.5 General Programming on GPU (GPGPU)

The GPU [6] refers to the commodity off-the-shelf 3D Graphics Processing Units, which are specifically designed to be extremely fast at processing large graphics data sets for rendering tasks. GPU designers traditionally have expressed its image-synthesis process as a hardware pipeline of specialized stages which necessarily involve Vector/Matrix Operations. The need for efficient hardware to perform floating-point vector arithmetic

for millions of vertices each second has helped drive the GPU parallel-computing revolution.

GPUs have evolved from a hard-wired implementation of the graphics pipeline to a more programmable one. Fixed-function units for transforming vertices and texturing pixels have been replaced by programmable shaders. These shaders provide units that the programmer can use for performing matrix-vector multiplication, exponentiation, and square root calculations etc. This however necessitates that there should be some means by which general purpose software could be translated into GPU specific primitives.

2.6 CUDA

CUDA (or Compute Unified Device Architecture) is a parallel programming model and software environment developed by Nvidia[7]. It was designed as a middle-ware to allow application software that transparently scales its parallelism on GPU. The core concepts involved with CUDA are a hierarchy of thread groups, shared memories, and barrier synchronization. The thread hierarchy allows user to divide his task in a similar hierarchy, where coarse sub-problems can be solved independently and finer pieces that can be solved cooperatively in parallel using shared memory. CUDA achieves all this using a minimal extension to C thus maintaining a low learning curve for programmers already familiar with the standard programming language.

2.7 General Architecture of GPUs

Whereas CPUs are optimized for low latency, GPUs are optimized for high throughput. Thus applications that do not have requirement for low latency can be ported to GPUs to take advantage of their superior performance. The programmable GPU has evolved into a highly parallel, multi-threaded, many-core processor with tremendous computational horsepower and very high memory bandwidth. There is a widening gap between the raw performance capability of CPUs and GPUs, which is because the GPU is specialized for compute-intensive, highly parallel computation, exactly what graphics rendering is about, and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control. The general architectural difference between CPUs and GPUs is schematically illustrated below in figure 2.4

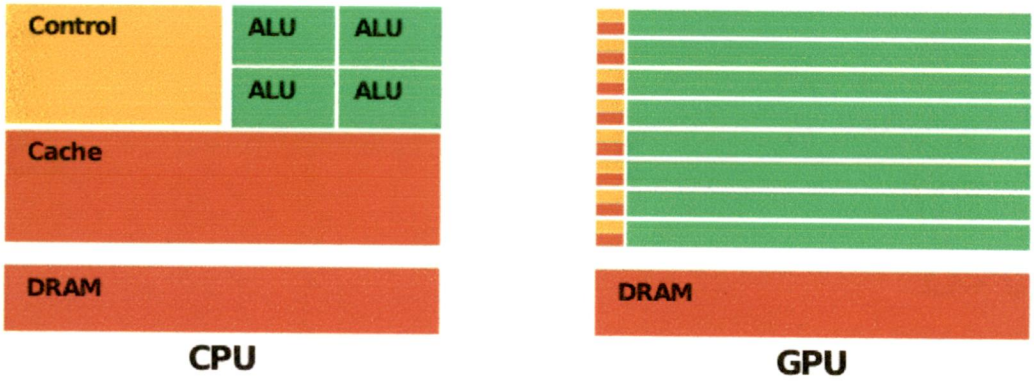

Figure 2.4: General architecture difference between CPU and GPU [8]

More specifically, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations; the same program is executed on many data elements in parallel, with high ratio of arithmetic operations to memory operations. Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control, and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches. The CUDA programming model is very well suited to expose the parallel capabilities of GPUs.

2.8 Programming Constructs and Thread Hierarchy

CUDA extends C[8] by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions.

A kernel is defined using the __global__ declaration specifier and the number of CUDA threads for each call is specified using a new <<<...>>> syntax

```
// Kernel definition
__global__ void vecAdd(float* A, float* B, float* C) {
// Kernel code
}
int main() {
...
    // Kernel invocation
```

```
    vecAdd<<<1, N>>>(A, B, C);
...
}
```

Each of the threads that execute a kernel is given a unique thread ID that is accessible within the kernel through the built-in threadIdx variable. This threadIdx values gives the index of the current thread within its block. In the above code, if the kernel were to add the two vectors A and B of size N and stores the result into vector C, the kernel code would be

```
__global__ void vecAdd(float* A, float* B, float* C) {
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

The logical organization of the thread hierarchy is thus, with the entire set of threads arranged as a two dimensional grid of blocks, with each block containing a three dimensional set of threads, as shown in figure 2.5

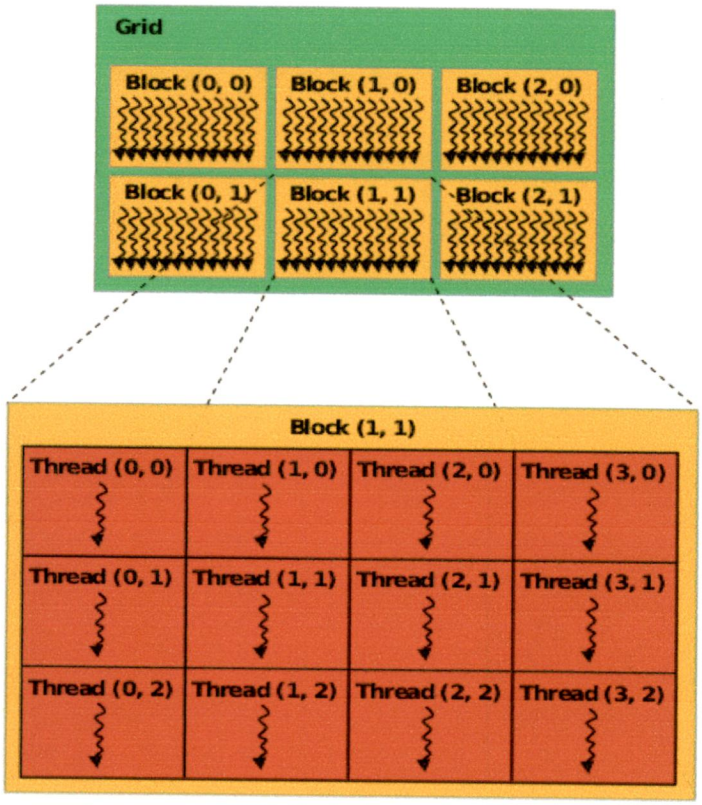

Figure 2.5: Thread Hierarchy in CUDA [8]

Threads within a block can cooperate among themselves by sharing data through some shared memory and synchronizing their execution to coordinate memory accesses. Such synchronization is possible by means of a programming primitive __syncthreads() as exposed by CUDA API. This serves as barrier synchronization. The number of threads per block is restricted by the limited memory resources of a processor core. On NVIDIA Tesla architecture, a thread block may contain up to 512 threads.

In addition to the variable threadIdx, CUDA threads also have a few other built-in variables namely blockIdx and blockDim. The blockIdx variable gives the index of the thread's parent block within the grid, and blockDim which gives the number of threads per block, with the blockDim being supplied in the call to the kernel as the second parameter to the <<<>>> syntax. Since grids are two-dimensional, blockIdx has a x component and y component and since blocks are three-dimensional, blockDim and threadIdx have x, y and z components. If the above code was to be a matrix addition instead of vector addition and was to be processes by a hierarchical arrangement of

threads as shown in the above figure 2.5, with each thread processing one element of the matrix, the code becomes

```
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    // Kernel invocation
    dim3 dimBlock(16, 16);
    matAdd<<<1, dimBlock>>>(A, B, C);
}
```

2.9 Memory Hierarchy

CUDA threads may access data from multiple memory spaces during their execution. Each thread has a private local memory, which is akin to local variable declaration for any normal CPU code. Each thread block has a shared memory visible to all threads of the block and with the same lifetime as the block. Finally, all threads have access to the same global memory. CUDA assumes that both the host and the device maintain their own DRAM, referred to as host memory and device memory respectively. The global memory is persistent across kernel launches by the same application and is allocated in the device memory.

Memory management at runtime on the GPU RAM is done using CUDA API equivalents. The general procedure is to allocate memory on both host and device RAM, using cudaMalloc function call for the device memory. The data contents are copied from host memory to device memory using cudaMemcpy function. Writing data directly onto device memory from CPU code is not possible. The kernel calls are then made to do appropriate processing on the data. The processed data contents are copied back from the device to the host using cudaMemcpy function.

2.10 Architecture

The Tesla architecture is one of the architectures of Nvidia which support CUDA. It is built around a scalable array of multi-threaded Streaming Multiprocessors (SMs). When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors.

A multiprocessor consists of eight Scalar Processor (SP) cores, two special function units for transcendentals, a multithreaded instruction unit, and on-chip shared memory. To manage hundreds of threads running several different programs, the multiprocessor employs a new architecture we call SIMT (single-instruction, multiple-thread). The multiprocessor maps each thread to one scalar processor core, and each scalar thread executes independently with its own instruction address and register state. The multiprocessor SIMT unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. Individual threads composing a SIMT warp start together at the same program address but are otherwise free to branch and execute independently.

When a multiprocessor is given one or more thread blocks to execute, it splits them into warps that get scheduled by the SIMT unit. Every instruction issue time, the SIMT unit selects a warp that is ready to execute and issues the next instruction to the active threads of the warp. A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken; disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Thus the efficiency of CUDA is enhanced if the different threads of a block (particularly of a warp) are executing the same code path but only on different data, thus containing little branching.

Chapter 3: Protein Folding Algorithm

3.1 Protein Folding

Proteins or poly-peptides are organic compounds made of amino acids, arranged in linear chain, joined together by peptide bonds [9]. They support life function by carrying out important biological functions, which are primarily determined by their structures. Each protein exists as an unfolded poly-peptide when translated from a sequence of mRNA to a linear chain of amino acids. Protein folding is the physical process by which these poly-peptides fold into their characteristic and functional three-dimensional structure (called conformation) from an initial unfolded or random structure.

Despite the fact that the theoretical number of possible conformations is astronomical, the actual time to fold is very small, which suggest that proteins use some sort of directed mechanisms to fold. These mechanisms are however not completely understood until now. Yet a few things are certain. These well define conformations are maintained so by a delicate balance of various forces like inter-protein, intra-protein, and hydrogen-bonds with the solvent. Current studies of protein folding involve the interaction and contribution of these various forces to the folding process.

One of the findings of research [1] in the field has been that there exist partially folded structures which form the intermediaries in the process. The protein goes into the final folded state through stages of such intermediaries. Also these intermediary stages have their stability dependent upon temperature. Now if these intermediary stages were subjected to de-stabilizing temperatures, could result in the inability of the protein to pass through these stages as desired and could end in improperly folded proteins. Such improperly folds give rise to dysfunctional proteins like the ones which cause Alzheimer's disease or Mad Cow Disease, or could give rise to desirable phenomenon like boiled eggs (caused by mis-folds of proteins in egg-white during boiling).

3.2 2D HP Model

Various models have been suggested to study the concept of protein folding [10], at various levels of abstraction, from "All Atom Model" to "Lattice Model". The "All Atom Model" works with molecular representation of both the amino-acids and the solvent. This method however works better for smaller sized proteins.

The "Lattice Models" treat the amino-acid of the protein as a unit and study the protein folding process as a function of interaction betweens these amino-acids as a unit. The protein chains have their constituent amino-acids bound together by peptide bonds. These participating atoms form what is called the back-bone of the protein. The bonds within this back-bone provide some degree of flexibility to the entire structure, and determine the extent to which the back-bone and hence the entire structure can be bend at various points.

A simplification of the above idea is to limit the degree of freedom for the back-bone to keep it confined to two-dimensions. The Two-Dimensional Hydrophobic Polar Model (henceforth 2D HP model) was such an idea proposed by Dill [11]. The 2D HP model classifies the constituent amino-acids of the protein into two groups, namely hydrophobic (no affinity for solvent) and polar (having electrostatic affinity for solvent). The protein is thus modeled as a string of H and P. The model thus goes out to place the string on a two dimensional lattice as shown in the figure 3.1 below.

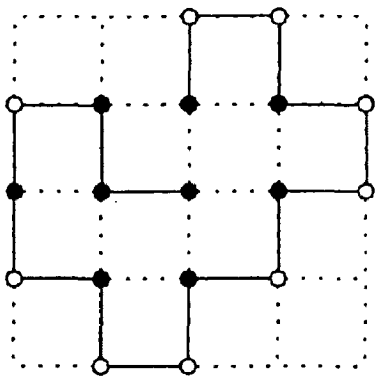

Figure 3.1: 2D HP Model of Protein Folding

The fundamental principle underlying the model is that the tendency of the protein will be to fold in such a manner that the Hydrophobic (H) nodes have minimum contact with external solvent and are surrounded by the Polar (P) nodes which maintain contact with the solvent. Hence the Hydrophobic (H) nodes tend to remain attached to each other and form a core within the center. This can be seen in the figure 3.1 with the black points representing the Hydrophobic (H) nodes and white nodes representing the Polar (P) nodes. This tendency of sticking together of Hydrophobic (H) nodes is used as a criteria for evaluating the energy or fitness of the conformation. The energy of the conformation is calculated by checking the number of Hydrophobic (H) nodes which are not adjacent on the protein chain but occupy adjacent position in the lattice. Such pairs of H nodes are an indication of the tendency of H nodes to stick together. For the above figure the energy is calculated to be 9.

3.3 Genetic Algorithms

Genetic Algorithms (GAs) [12] are a searching paradigm used in computing to find exact or approximate solutions to optimization and search problems. Genetic algorithms are categorized as global search heuristics. Genetic algorithms are a particular class of algorithms called evolutionary computation, that use techniques inspired by evolutionary biology such as inheritance, mutation, selection, and pairing (also called recombination or cross-over).

There are many algorithmic problems which, if they are to be solved by traditional analytical methods, are intractable, i.e. their runtime is an exponential function of the input size. However these same problems are verifiable (check-able whether a given solution is correct/optimal) in polynomial time. Also for some such problems, for practical purposes less than optimal solution are acceptable if they bring about a drastic decrease in running time of algorithm. For such cases we can use genetic algorithms to obtain solution for the problem.

Genetic algorithms are used in computer simulations, wherein the candidate solutions for the problem are encoded in format of a string over a certain alphabet (usually {0,1}) e.g. for Traveling Salesman Problem (TSP) over a graph with n cities, we can encode the

candidate solutions as a number string of length n, giving the order in which the salesman visits the cities.

The general procedure as adapted by genetic algorithm would be to first generate an initial set of random candidate solutions for the problem (e.g. for the TSP example above we generate a set of random permutations of orders in which to visit cities). The solution set can be tested for correctness and then for optimality. The next generation of candidate solutions is generated from the previous generation using a set of simple operators called genetic operators. Only those solutions which are more optimal than the previous generation are retained and the process is continued.

Some of the basic genetic operations are

- Elitism: In this operation, the optimal solution obtained for any generation is carried forward into next generation without any changes. Thus it can be guaranteed that any generation of solution contains one at-least solution which is at-least as good as the best of the last generation, thus ensuring that the subsequent solutions only get better.
- Mutation: In this procedure, the solution of the current generation will be modified slightly to produce the next generation solution. In the TSP example above, we can select two numbers at random in the candidate solution, and exchange them, creating a new permutation. In the actual TSP scenario, this will change the order of visiting the cities.
- Pairing: In this procedure, any two random solutions of the current generation are combined together. In the TSP example, we randomly choose to take cities visiting order from one parent for first k cities and from second parent for the remaining cities.

With each iteration/generation of the genetic algorithm, the fitness of the conformation is checked. The genetic algorithms are repeated till the calculated fitness does not equal the theoretical expected fitness.

3.4 Protein Folding using Genetic Algorithms

Unger and Moult [15] proposed the application of GAs to the 2D HP Model. They have shown the performance improvement using their technique visa-a-vis existing parallel simulation techniques for 2D HP Model using Monte Carlo methods. The methodology as proposed by Unger and Moult consisted of running T independent instances of protein folding simulations, (called as a population of T instances in GA parlance). The process for each instance consists of two phases, namely the Initial Set Generation (ISG) and application of Genetic Operators (GO). We used a public domain implementation of the algorithm done by Schubert, Buchetmann, Clote and Backofen [16].

The algorithm can be formally summed up as follows

T : Number of independent simulation instances
N : Number of amino-acids in the protein (Size of the protein)
M : Number of mutations
optimum : Theoretical optimum energy value
direction[**N**-2] : Array of bends
bestConf : Best conformation at each stage of GO

1. **for** *sim_no* = 1 to **T** do
   1.1. **for** each bend *current* in the protein chain do
      1.1.1. Generate a random bend direction, i.e. one value out of **l**, **r** and **s** (**left**, **right** and **straight**), say *p*, i.e. *direction*[*current*] = *p*
   1.2. $a_1 = (0,0)$
   1.3. $a_2 = (1,0)$
   1.4. **for** each further bend *current* in the protein chain **do**
      1.4.1. calculate $x_{current}$ and $y_{current}$ from $x_{current-1}$, $y_{current-1}$ and direction[current-1]
      1.4.2. if $a_{current} = (x_{current}, y_{current})$ && some $a_m = (x_m, y_m)$ such that $x_{current} = x_m$ and $y_{current} = y_m$, i.e. bend is not possible **then**
         1.4.2.1.reject the conformation, goto step 1.1
2. **while** bestConf.fitness <> **optimum** do **// Genetic Operators**
   2.1. **for** mtns_no = 1 to **M** do
      2.1.1. *bestConf* := best conformation among T instances // elitism
      2.1.2. **for** *sim_no* = 1 to **T** do
         2.1.2.1. apply mutation on conformation *sim_no*
      2.1.3. *bestConf* := best conformation among T instances // elitism
      2.1.4. **for** *sim_no* = 1 to **T** do
         2.1.4.1. apply mutation on conformation *sim_no*
   2.2. *bestConf* := best conformation among T instances // elitism
   2.3. **for** *sim_no* = 1 to **T** do
      2.3.1. apply cross-over on conformation *sim_no*

2.4. *bestConf* := best conformation among T instances // elitism
2.5. **for** *sim_no* = 1 to **T** do
    2.5.1. apply cross-over on conformation *sim_no*

The ISG part (as shown in step 1 in the above algorithm), involved generating an initial set of random conformations, one conformation for each instance. This was done by generating a set of random bends (N-2 bends for a protein of length N), where each bend could either be left, right or straight. The protein conformation was then laid out on a 2D grid in accordance with the generated bends to check for overlap. If an overlap was detected, the conformation was rejected in totality and the entire process of generating a conformation was repeated, regenerating all bends again, the process being continued until a non-overlapping conformation was obtained.

The GO part involved multiple iterations of application of genetic operators to the conformation generated in the ISG. Each iteration of GO consisted of M (M=20 in the implementation [16]) rounds of elitism and mutations followed by 2 rounds of elitism and cross-over. The iterations were continued indefinite till the optimum energy for entire population equaled the theoretical optimum. The independent T instances were taken as a single logical population at each stage of mutation, cross-over and elitism. Thus the conformations for any stage were obtained by applying GO to the population of previous stage only (and not current stage).

The mutation stage consisted of randomly selecting one amino acid out of the entire length of the protein and changing the bend direction of the protein around that amino acid. The transformed (mutated) conformation was checked for overlap. If an overlap occurred, the process of random selection of mutation node and change of bend direction for it was repeated. If no overlap was found for the transformed conformation, the energy value of the protein was calculated. The non-overlapping transformed conformation was accepted if its energy value is greater than average energy of the T conformations. If the energy value non-overlapping transformed conformation was less than the average energy, the conformation was accepted with some probability.

For generating a cross-over conformation, two conformations were randomly selected from the population of the previous stage to act as parent conformations. A random amino acid would be selected in the cross-over conformation which acts as the cross-over node. The cross-over conformation would contain the bend pattern from one parent before the cross-over node and the other parent after the cross-over node. The cross-over node was also mutated. The cross-over conformation was checked for overlap. Like in the mutation stage, the resultant conformation was rejected if it resulted in an overlap. If conformation was rejected, new set of parent conformation and cross-over node were randomly selected for next trial. As with the mutation stage, the non-overlapping cross-over conformation was accepted only if its energy was better than average, or with some probability.

The elitism stage propagated the value of the best and average conformation energy of entire population of T conformations at each stage (mutation/cross-over) to the next stage.

3.5 Search Space Pruning

We observed during the process of profiling the algorithm that the bottleneck for the algorithm was the ISG part. As the size of protein grew, the probability of getting an overlapping conformation increased and hence increased the probability of rejection of conformation and need for regeneration. This caused the increase in run-time of the algorithm.

Our idea was to reduce the run-time of this algorithm-part, by eliminating the possibility of generating an overlapping conformation. We do not generate the entire conformation and check the overlap. Instead we generate the conformations partially, one amino acid (hence on bend at a time) at a time and check overlap in partial conformations. If placing an amino acid is causing overlap, the step is back-tracked, and only the last bend is regenerated. If all three possible bends for a given amino acid placement cause an overlap, we back-track one more step.

We now formally put forth our algorithm and further prove its soundness and completeness.

T : Number of independent simulation instances
N : Number of amino-acids in the protein (Size of the protein)
M : Number of mutations
allowed[**N**-2][3] : boolean(yes/no) array of length **N**-2

1. **for** *sim_no* = 1 to **T** **do**
   1.1.Conformation string **C** = ""
   1.2.a_1 = (0,0)
   1.3.a_2 = (1,0)
   1.4.**for** each further bend *current* in the protein chain **do**
      1.4.1.**if** *allowed*[*current*][0] = **no** **&&** *allowed*[*current*][1] = **no** **&&**
         allowed[*current*][2] = **no** i.e. all bends are disallowed **then**
         1.4.1.1.Mark last bend direction (say *p*) for node *current-1* as disallowed.
            i.e. *allowed*[*current-1*][*p*] = **no**
         1.4.1.2.*allowed*[*i*][*j*] := yes, *i* = *current* to **N**, *j* = 0 to 2
         1.4.1.3.*current* = *current-1* // back-track
      1.4.2.**else**
         1.4.2.1.Generate a random bend direction, i.e. one value out of **l**, **r** and **s**
            (**left**, **right** and **straight**), say *p*
         1.4.2.2.**if** *allowed*[*current*][*p*] = **no** **then**
            1.4.2.2.1. repeat step 1.4.2
         1.4.2.3.**else**
            1.4.2.3.1. if $a_{current} = (x_{current}, y_{current})$ && some $a_m = (x_m, y_m)$ such that
               $x_{current} = x_m$ and $y_{current} = y_m$, i.e. bend is not possible **then**
               1.4.2.3.1.1. reject the bend, *allowed*[*current*][*p*] := **no**
            1.4.2.3.2. else
               1.4.2.3.2.1. ***C*** = ***C***.*p*
               1.4.2.3.2.2. *current* := *current* +*1*
2. Apply genetic operators as in old algorithm.

Like with the original algorithm, in our modified algorithm, the laying out of the conformation starts with placing the first two amino acid nodes at default positions on the grid. We keep track of the entire grid as a matrix data structure, with the occupied nodes in the grid marked out clearly. A square grid matrix of same dimensions as the protein length suffices. We also keep track of the current direction of the partially laid out conformation. With the first two nodes at (0,0) and (1,0), the initial direction of the partial conformation is **east**. As with the existing algorithm, we also keep track of the bends taken so far.

While placing a new amino acid node on the grid, we first generate a random bend. Using the co-ordinates of the last placed amino-acid node on the grid, the current direction of partial conformation and the generated bend, we calculate the would-be co-ordinates for the next amino acid on the grid. From the grid data available to us we can check if the grid co-ordinate is already occupied or empty. If the grid co-ordinate is empty, we go ahead with placing the amino-acid node on the grid and update our grid data structure. We also update the current direction of partial conformation, as this would change for a left or right bend.

We keep track of the set of bends allowed at any given node by means of a boolean matrix, **"allowed"** with three entries per node, one entry for each of the possible bends. When a random bend is generated, we first check if this bend allowed in this **"allowed"** matrix, saving the need for calculating the would-be co-ordinates and checking the grid. Also if after calculating the would-be co-ordinates, we find that the grid is occupied at that position, we update the **"allowed"** matrix, specifying that the specific bend for the node under consideration is not allowed.

If all three bends possibilities for a given amino acid node are closed, means that we have hit a dead end. In this case we need to correct our placement of just-previous amino-acid node. This is done marking the taken bend direction of the previous placed node as now not allowed. Since we have kept track of the co-ordinates of the last placed amino-acid, and also all bend directions taken so far, we would be able to calculate the co-ordinates of previous-to-previous amino-acid, and back-track one step. We would also change the current direction accordingly.

3.6 Algorithm Soundness

We now establish the soundness of our algorithm modification. We specify the loop-invariant and show using it that the modified algorithm does not produce any self-overlapping conformations.

Loop Invariant

The partial conformation string ***C*** does not overlap

Initial Case

At start of loop, ***C*** = "", hence |***C***| = 0, hence does not overlap

Maintenance

For any bend *p* in the current iteration only if ***C.p*** does not overlap, as in step 1.5.2.3.2.1, we go ahead with the bend. Else (as in step 1.5.2.3.1 and step 1.5.1), the conformation ***C*** is not appended with any bend

Termination

At termination, |***C***| = N-2, and as ***C*** does not result in any overlap (as shown in Maintenance), the entire conformation ***C*** (now of length N-2) is non-overlapping

As we have taken care to see that a new accepted bend is appended to the partial conformation string of bends only if an overlap is not generated, it is not possible for overlap to occur in the final conformation.

3.7 Algorithm Completeness

We now prove that the algorithm modification does not cause a loss in reach-ability for any part of the search space.

1. The T conformations generated are all non-overlapping, let S be the set of these T conformations
2. Genetic algorithm methods are applied to the set S, in second part of algorithm.
3. Even if some bias is present in the conformation strings generated, the mutations and pairing process as applied on these strings can remove the bias, as follows.
    a. Assume all T instances of initial conformations start with its first character as "l", the solution set S is thus biased, as it does not include part of search space which

starts with "r" or "s"

b. The probability that the character selected for mutation is not the first character = (N-2)-1/(N-2) = (N-3)/(N-2)
c. The probability of not mutating the first character in any of the T instance = [(N-3)/(N-2)]^T
d. The probability of mutating the first character in at-least of the T instance and thus un-setting of the bias = 1 – [(N-3)/(N-2)]^T which is > 0

3.8 Parallelization on Cell-BE

The parallelization of the Protein Folding simulation continued from our work done on the original algorithm. Since the algorithm involved independent simulation of T instances of protein folding, it was amenable to straight forward parallelization. The T instances of parallel simulations were divided among the 8 SPEs giving T/8 instances to each SPE. Further improvements were obtained, using double buffering. For every simulation instance *i* running on an SPE, undergoing one stage(mutation or cross-over) of GO, the processed data for *i-1* instance is transferred from SPE to PPE, thus causing overlap of computation on SPE with data transfer from SPE to PPE.

Also the algorithm consisted of intermittent stages of elitism which needed to share data between the T simulation instances, i.e. the best conformation data and the total population fitness calculated in each elitism stage had to be passed to each SPE. For the elitism stage, each SPE undertook the elitism operation for its set of T/8 conformations, and the consolidated output for these 8 SPEs was then calculated by the PPE and passed onto each of the SPE for next stage of GO.

A few issues were come across during parallelization of the algorithm. Since the algorithm did not involve any array/matrix calculation on floating point numbers, but rather a complicated logic flow dictated by randomized input, application of SIMD within one simulation instance was not feasible. Also since operations on different simulation instances were not same (due to random nature of simulation), clubbing together of n instance to do n-way SIMD was not possible.

3.9 Parallelization on CUDA

The parallelization of the search-space-pruned version of the protein folding simulation algorithm was similar to that on Cell-BE, as the implementation was amenable for straight forward parallelization. The parallel simulations of T instances of protein folding were distributed among T threads that would run on CUDA.

Since there was no need for much co-operation amongst the parallel instances, it was preferred that their corresponding threads be distributed sparsely over different blocks, with each block containing few threads, rather than a single block with multiple threads. The other major reason for such a division was that the logic for the simulation involved complication flow of control, which depended on randomized inputs. Consequently it was not possible for the different parallel simulations and hence their threads to have similar code paths, thus negating the use of threads from the same block. Since each block is to be scheduled over a single multi-processor containing 8 cores, we distributed the threads as 8 threads per block.

The implementation of the search-space-pruned protein folding algorithm consisted of two stages, namely data allocation and processing. The data allocation stage involved allocating memory for the conformation on the device. The creation of conformation data (ISG) and processing (GO) are all done on the device by parallel threads. The elitism stage is also carried out on the device, but using only a single CUDA thread. This was because the elitism stage consisted of finding the best value in an array of conformations and did not offer any possibility of parallelization. Also the stage was very likely not a bottle-neck for performance.

An obstacle encountered in the implementation of the algorithm in CUDA was the absence of libraries to generate random values on the device. This necessitated the creation of a linear congruential method based random number generator with the device code, with its accuracy comparable to the rand() function as available on normal Unix based system. It was ensured that the random number generated would be independent for each simulation instance/thread.

3.10 Results for Modified Protein Folding on Cell-BE

The timings were calculated for the existing algorithm as run on a Core-2 Duo HP Laptop (2*1.67 GHz). Each set of timings are compared with running the modified algorithm on IBM Cell BE Simulator/Blade Server. The algorithms were run for protein sequences of length 20-64, for standard protein data as obtained from [15].

The algorithm mainly consisted of two parts, namely generation of initial set of conformations (ISG), and then applying genetic operators (GO) on them, which are elitism, mutation and pairing. The timings were calculated separately for both parts and percentage contribution of the ISG was calculated.

The GO stage consisted of a number of iterations of elitism, mutation and pairing, where the iteration count depended upon the speed with which the GO stage converged.

Table 3.1: Protein Sequences Data Used

Sequence Length	*Optimal Energy*	*Sequence*
20	9	hphpphhphpphphhpphph
24	9	hhpphpphpphpphpphpphhh
25	8	pphpphhpppphhpppphhpppphh
36	14	ppphhpphhpppppphhhhhhhpphhpppphhpphpp
48	22	pphpphhpphhpppppphhhhhhhhhhpppppphhpphhpphpphhhhh
50	21	hhphphphphhhhphpppphpppphpppphpppphpppphhhhhphphphhh

Table 3.2: Timing Comparisons for both Algorithms (time in milliseconds)

Sequence Length	*Old Algorithm (Uni-Proc)*			*New Algorithm (Cell-BE)*		
	Energy Obtained	*Initial Set Generation (ISG)*	*Genetic Operations (GO)*	*Energy Obtained*	*Initial Set Generation (ISG)*	*Genetic Operations (GO)*
20	9	133	651	8	14.66	579.86
24	9	409	743	8	14.72	586.47
25	8	509	759	6	14.58	578.89
36	13	10461	1403	8	11.80	626.46
48	22	292224	2581	13	15.00	748.47
50	20	759731	2959	13	14.66	738.53

Table 3.3 Part-Wise and Total Speed Up for Cell-BE

Sequence Length	*Part-wise Speed-Up*		*Iteration Count for GO*		*Percentage contribution of ISG**	*Speed Up*
	Initial Set Generation (ISG)	*Genetic Operations (GO)*	*Old Algorithm (Uni-Proc)*	*New Algorithm (Cell-BE)*		
20	9.07	1.12	72	182	0.11	1.12
24	27.78	1.27	13	656	0.08	1.29
25	34.91	1.31	126	833	0.08	1.33
36	886.52	2.24	1208	660	0.61	7.63
48	19481.6	3.45	718	546	13.62	2656.37
50	51823.39	4	1130	531	18.51	9595.76

Percentage contribution of ISG = Old Time for ISG / (Old Time for ISG + Old Time for GO Worst Count of GO Iterations (Old/New)) expressed as percentage.

3.11 Results for Modified Protein Folding on CUDA

The timings were calculated for the new algorithm as run on a Core-2 Duo HP Laptop (2*1.67 GHz), and then on a CUDA based machine having Intel Xeon CPU (2*3.2 Ghz) and NVIDIA Geforce GTX-280, having 240 cores. The algorithms were run for protein sequences of length 20-36, for standard protein data as obtained from [15], same as those used for Cell-BE performance evaluation.

Table 3.4 Timing Comparisons for Algorithms on Uni-Proc and CUDA

Sequence Length	*(Uni-Proc)*			*(CUDA)*		
	Energy Obtained	*Initial Set Generation (ISG)*	*Genetic Operations (GO)*	*Energy Obtained*	*Initial Set Generation (ISG)*	*Genetic Operations (GO)*
20	8	9.460	308.650	8	2.723	122.401
24	9	13.458	405.869	8	3.302	201.626
25	7	12.099	435.486	7	3.716	206.033
36	10	15.820	921.587	10	5.575	828.261

Table 3.5 Part-Wise and Total Speed Up

Sequence Length	*Part-wise Speed-Up*		*Iteration Count for GO*		*Speed Up*
	Initial Set Generation (ISG)	*Genetic Operations (GO)*	*(Uni-Proc)*	*(CUDA)*	
20	3.474	2.521	15	7	2.521
24	4.075	2.012	13	69	2.012
25	3.255	2.113	9	16	2.113
36	2.837	1.112	21	72	1.112

Percentage contribution of ISG = Old Time for ISG / (Old Time for ISG + Old Time for GO Worst Count of GO Iterations (Old/New)) expressed as percentage.

Chapter 4: Extended Burrows Wheelers Transform

4.1 Introduction to Burrows Wheeler Transform

The Burrows Wheeler Transform [13] is a block-sorting, lossless data compression algorithm, which is used in applications like bzip2. It was developed by Michael Burrows and David Wheeler. A variation of the algorithm was developed by Mantici which extended the concept to a multi-set of words, unlike the original algorithm that worked on single block of text at a time treating it as a single word. A key realization by Mantici et.al. was the applicability of their extended algorithm to the domain of bio-informatics, namely for matching genomic data of species to establish their phylogenetic proximity or non-relatedness.

4.2 Burrows Wheeler Transform

The key idea behind the Burrows Wheeler Transform is to apply a reversible transformation on a block of text so as to convert it into another block of text, by a re-arrangement of characters, a block that is easier to compress. The characters are re-arranged such that the output contains runs of same characters together. Such transformed blocks are then amenable to other compression techniques which take advantage of such runs, like run length encoding or move to front encoding. Such encoding may then be followed by compression encoding like Huffman Encoding. The algorithm produces N conjugates words of the input block text (called primitive word) each of length N, by cyclic shifting the input, one character at a time. These conjugates are then sorted out lexicographically and the last character of each conjugate string is taken. This last column and the position of the original input text in the sorted list form the output of the transform.

It is generally observed in normal language text that certain di-grams or tri-grams occur more frequently than others. Now when texts containing such di-grams or tri-grams are cyclic shifted and sorted lexicographically, closely sorted conjugates are very likely to end in same character, e.g. the trigram "the" is a very common tri-gram, hence all text starting with "he" are very likely to end in a "t". Thus, when such text is sorted, then in

the last column, the "t"s are likely to be clubbed together and hence gives runs of the same character.

Word = "abraca"

Conjugates are
abraca, bracaa, racaab, acaabr, caabra, aabrac
Sorted Conjugates are

1. *aabrac*
2. *abraca*
3. *acaabr*
4. *bracaa*
5. *caabra*
6. *racaab*

Output is last column = ***caraab*** and index of original word = ***2***

Now with the input text and its conjugates sorted, since each of the conjugate string is a cyclic rotation of the original string, hence each character of the original string appears in any given position only once, in any one particular conjugate. Thus for all the sorted conjugates, any column contains all the characters of the original string, each exactly once. Thus the first and the last column both contain all characters of the original stiring. Since the conjugates are all lexicographically sorted, the first column for the sorted matrix, is effectively a sorted list of all characters of the input string. Thus the first column is a sorted form of the last column. Since the conjugates are all cyclic variations of each other, thus for any given conjugate string the character in the first column follows the character in the last column.

We now see the reverse transform, or how we can obtain the original string from the last column and the index of the original word. Since we have the last column as our output from the original transformation, and we can sort it to obtain the first column, we now have both first and last column. And since the first column cyclically follows the last column, we get the relation as to which character follows whom in the original string. Thus this mapping that "character x follows character y" can be used to reconstruct the entire original string. However there is one caveat in this. Since a character may be repeated in the original string and hence in the last and first column, how do we ascertain

which mapping corresponds to which occurrence of the character. It was shown in the original paper that the order in which multiple instance occur in the last column is preserved in the first column and hence any ambiguity regarding this mapping is thus removed.

4.3 Extended Burrows Wheeler Transform (EBWT)

Mantici [14] produced a variation of the Burrows Wheeler Transform and extended it to apply it to a multi-set of words instead of a single block of text. In this case too cyclic conjugates for all the words are produced and the entire lot is sorted. The sorting however is not lexicographic. The paper introduced another form of sorting called ω sorting.

ω sorting

In normal lexicographic sorting, if we encounter two words of different lengths such that one is the prefix of another then, the smaller word is considered lexicographically smaller than the other, and hence is sorted above. However in ω sorting, a word is expanded by repeating the same word over again, creating an infinitely (theoretical) long repeated sequence of original word (called repeat-formation). These repeat-formations of all the words are then sorted lexicographically.

Consider the two words *ab* and *aba*

The conjugates of the word will be
ab and *ba*
aba, *baa*, and *aab*

Sorting them lexicographically gives

1. *aab*
2. *ab*
3. *aba*
4. *ba*
5. *baa*
6.

Creating repeats of the words we get
ab: *abababababab...*
ba: *babababababa...*
aba: *abaabaabaaba...*
baa: *baabaabaabaa...*
aab: *aabaabaabaab...*

Now sorting this repeated string lexicographically, we get the sorted sequence for the original words is

1. *aab*
2. *aba*
3. *ab*
4. *baa*
5. *ba*

Notice that *aba* occurs before *ab* and *baa* occurs before *ba* in ω sorting, in contrast to the normal lexicographic sorting.

It was argued by Mantici that the repetitions need to go on to a length of only $n1 + n2 - \gcd(n1, n2)$ where *n1* and *n2* are the length of the two words involved. Similar to the Burrows Wheeler Transform, the output here too would be the last column, but of the conjugates of the original set of words and not the repeat-formations. Also since the length of the conjugates would be different for each original word, we would take the last character in each. Also like in the Burrows Wheeler Transform, the indices of the original set of words are also part of output.

4.4 Distance Measure in EBWT

Although the primary motivation for the Extended Burrows Wheeler Transform is data compression, which it achieves better than the original algorithm, Mantici found that the algorithm could be used to process of phylogenetic analysis. It is obvious that the sorting process causes interleaving of the conjugates originating from different original words, but it was observed that if two original words are more similar, the interleaving of their conjugates was more pronounced. However if the words were more dissimilar, then conjugates of one word were more likely to occur together in the sorted list. This fact was formalized by a concept called distance as introduced by Mantici.

For each pair of original words, we run through the sorted list and find runs of consecutive conjugates of single word. N consecutive conjugates of same word contribute to a distance of N-1. During comparison of two words in this manner, the other words in the set are ignored. So now if two words are more similar, their pronounced interleaving causes a small distance, whereas if the two words are dissimilar, their conjugates occur together giving a large distance.

Consider EBWT as applied to a multi-set of two words u = bcaa and v = ccbab

After sorting the conjugates' repeat formations, we get

aabc, abca, abccb, babcc, bcaa, bccba, caab, cbabc, ccbab

If we list the original words to which each sorted repeat-formation belongs, we get

u	u	v	v	u	v	u	v	v

The distance for the above case would be 3 as there is one run of u of length greater than 1 and two such runs of v.

This concept is applied to the process of phylogenetic analysis wherein, the original words are mitochondrial DNA of species. Phylogenetically related species give a small distance and unrelated species give a large distance. The paper verified the results of the algorithm with standard data and results from other known algorithms.

4.5 Parallelization of EBWT on CUDA

The implementation of EBWT was done firstly on a uni-processor and then on CUDA and their timings compared. The primary time consuming process within the EBWT was the sorting of the conjugate-repetitions.

4.6 Uni-Processor Implementation

The implementation first reads the gene sequences from a file, one for each species and stores them into a local buffer. This set of gene strings is what we call the set of primitive words. The set of primitive words are used to generate the conjugates for each element in the set. The conjugates are then used to generate repeat-formations of themselves. The maximum width of the repeat formation for successful working of the algorithm is

$$H = \max\{|u_i| + |u_j| - \gcd(|u_i|,|u_j|) \mid i, j = 1..k\} \quad \text{..... (Equation 1)}$$

We use the maximum and second maximum length of the genes that we have. However we skip the process of finding the *gcd* as this is an unnecessary overhead does not affect

the accuracy of the algorithm. Having done this we obtain an array of repeat-formation strings, which we need to now sort.

The sorting of the repeat-formations is done using column wise counting sort as the range of the possible values of the string contents is restricted to four values, namely 'a', 'c', 'g' and 't'. Also since we use one value of max width from above equation, we have all the strings of same width. Thus we sort the string using counting sort on each column, starting from last column up-to the first column (much like radix sort for integers). The approximate algorithmic complexity of the sorting process is O(k*n*n), where n is the length of the gene and k is the number of species. This is so because there are around 2*n columns, calculated from the equation above, and each column has k*n characters to be sorted by counting sort.

We now need to find the distance measure between each pair of species. We run through the sorted list for each pair of species. We consider only those repeat formations in the list which belong to the species-pair under consideration. Using a flag counter, we calculate the runs of repeat formation belonging to only one species in the pair, and we do so for both the species. The total run gives an approximate measure of the phylogenetic distance between the two species. These values were calculated and found to be tallying with results as published by Mantici.

4.7 GPU Implementation

The GPU implementation primarily consists of two main phases, namely data construction and data sorting. The data construction phase consists of allocating memory on the GPU and transferring data onto it from the CPU. The data as required by the algorithms was to be generated and processed on the GPU, with the CPU doing the initial work of reading the gene sequences from a file and have them transferred to the GPU. One of the problems problem encountered in use of CUDA was the absence of string processing libraries on GPUs (since the device is primarily math-intensive), which required that they be written from scratch as device-level user functions.

Another main problem encountered in porting the algorithm onto CUDA was the absence of simple means of allocation multi-dimensional arrays on GPUs. This problem was more

pronounced if the dimension of the multi-dimensional array were not known until runtime. We went ahead with simulating two dimensional arrays of characters (or to be correct, an array of strings, as required for our conjugate and repeat-formations data) using one dimensional array, by setting a practical maximum limit on the length of each row of the logical two-dimensional array (this practical maximum limit was already known to us as max-width using equation). The two dimensional arrays of characters (or array of strings at logical level of abstraction) allocated were of for the primitive words, conjugates and repeat-formations.

Once the allocation of memory on device is done, the process of generation of conjugates and repeat-formations are divided between the thread on the device. Since there are approximately k*n such conjugates/repeat-formations to be generated, that amount to k*16K strings for our practical gene data, the process can be highly parallelized. The repeat-formations are then sorted as in the uni-processor implementation, but using a parallelized form on sorting as discussed in the next section.

4.8 Odd Even Sorting

Cormen et.al. [17] discuss various methods of parallel sorting, or sorting on parallel architectures, clubbed together as a concept called sorting networks. The fundamental means of sorting in parallel can be carried out by means of a sorting network based on primitive blocks like merger or bitonic sorting networks. These methodologies carry out parallel sorting in time of order O(log n). We did not use the methods for primarily two reasons. The methodologies logically involve recursion defining sorting of n numbers as combination of sorting two list of n/2 numbers and then merging them, and recursion was not possible on CUDA device, at-least a recursion that requires co-operation between threads. Secondly since the uni-processor implementation involved an algorithm sort component that runs in O(k*n*n), it would be correct that the parallel implementation for comparison runs in approximately the same time complexity. We chose a similar Odd-Even sorting network, which we explain in context of our work.

The fundamental component of a sorting network is a comparator, which is a two input, two output, logical entity that sorts. The comparator is a basic building block used to

construct larger sorting networks. The below figure 4.1 shows the schematic equivalent used to show a comparator as part of large network.

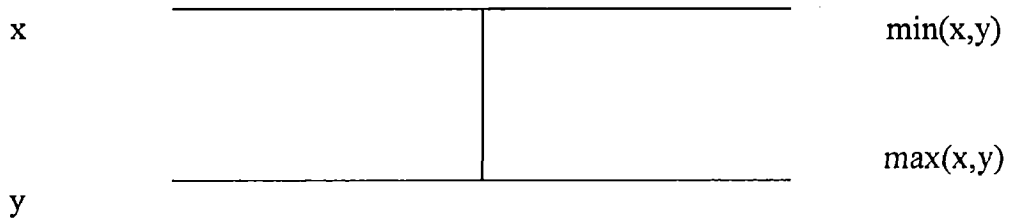

Figure 4.1: Comparator Schematic

The odd-even sorting network is constructed using comparators as shown in the figure 4.2 below. In context of our problem, the inputs of the sorting network are the repeat formations that need to be sorted. Each vertical line in the below figure corresponds to a comparator and each vertical column corresponds to one iteration of parallel sorting. The entire sorting process consists of alternate iteration of odd and even sort. The odd and even sort correspond to respectively comparison of

Odd Sort: a[2*i] and a[2i-1]

Even Sort: a[2*i] and a[2i+1]

This corresponds to simple string comparisons on our code. The number of such stages required as same as the number to be sorted, i.e. ~k*n for EBWT. Cormen[17] argued that the sorting network sorts correctly.

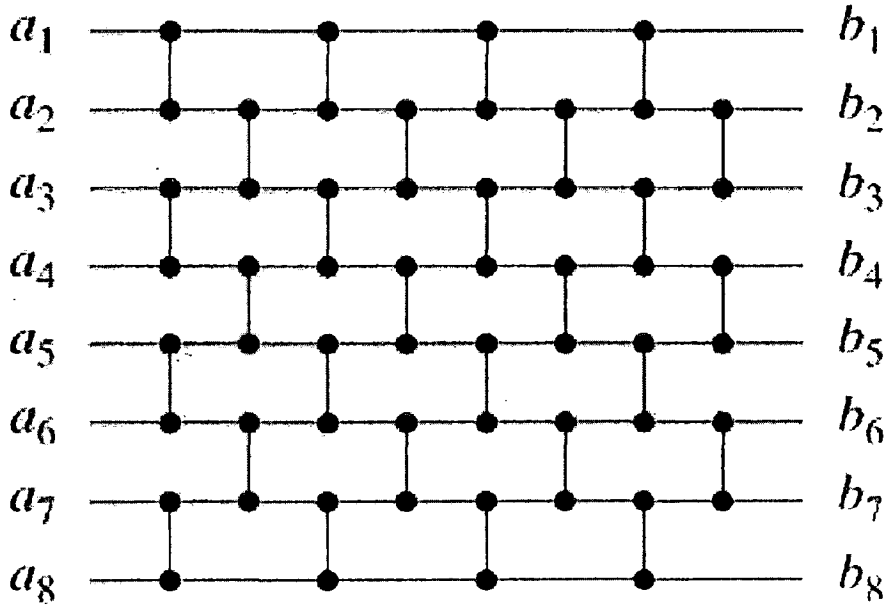

Figure 4.2: Odd Even Sorting [17]

The distance measure was to be carried between each pair of species. This was done by creating k^2 threads, one for each pair.

4.9 Results

The algorithm was used to compare varied number of species together, 2-8 at a time, with data obtained from [18][19]. The timings are as follows.

The machines involved are a Core-2 Duo HP Laptop (2*1.67 GHz), as a uni-processor and a CUDA based machine having Intel Xeon CPU (2*3.2 Ghz) and NVIDIA Geforce GTX-280, having 240 cores.

Table 4.1 Timing comparison* for EBWT

Number of Species Compared	*(Uni-Proc)*		*(CUDA)*		*Speedup*
	Data Generation	*Data Processing*	*Data Generation*	*Data Processing*	
2	0.296	6.693	0.484	0.516	6.98
3	0.432	10.578	0.703	1.234	5.68
4	0.564	15.819	0.921	3.016	4.16
5	0.721	22.410	1.171	4.969	3.76
6	0.878	31.819	1.500	8.109	3.40
7	1.687	46.637	1.906	14.265	2.98
8	1.213	54.168	2.171	20.516	2.44

* Time values in seconds

Chapter 5: Conclusion and Future Work

5.1 Search Space Pruned Protein Folding

The relative proportion of time spent in the ISG part of the algorithm increases with increase in size of the protein to be folded. Thus the importance of optimizing this part and speed-up achieved will be more significant for large sized of proteins. For protein larger than 60 amino-acids, the running time of original algorithm on uni-processor implementation was extrapolated to be more than 10 hours. For such large protein the algorithm modification makes it possible to have the process run in acceptable times.

The approach we used can also be applied to other problems involving use of genetic algorithms (e.g. GA for Travelling Salesman Problem). The conditions for rejection of solution and back-tracking would however be specific to the domain of the problem.

The convergence for the GO depended upon the goodness of the random number generation, and the effect of the algorithm modification on the convergence rate of the GO needs to be studied. Also the GO parameters like the number of mutations and cross-over per generation can be varied and their effect can be studied both on the runtime of the algorithm and the convergence rate of the algorithm to the optimum value.

5.2 EBWT on CUDA

The major bottle neck for performance in the EBWT algorithm was the process of sorting the intermediate data. The performance improvement of EBWT on CUDA was majorly due to the possibility of parallel sorting which took bulk of the runtime in the sequential implementation. The running time of the algorithm can be improved if the sorting part of the algorithm is implemented using O(log n) techniques of parallel sorting. However this would require that such a technique is implemented without recursion.

References

1 Thomasson WA, Unraveling the Mystery of Protein Folding. FASEB's Public Policy, http://www.faseb.org/opa/protfold.pdf (Last Accessed 12 June 2009)

2 http://www.nature.com/horizon/proteinfolding/background/importance.html (Last Accessed 12 June 2009)

3 http://en.wikipedia.org/wiki/Molecular_phylogenetics (Last Accessed 12 June 2009)

4 J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, D. Shippy, Introduction to the cell multiprocessor, IBM Journal of Research and Development, Vol-49 n.4/5, p.589-604, July 2005

5 Cell Broadband Engine Programming Tutorial, Version 2.0, IBM, Dec 2006

6 D. Luebke and G. Humphreys, How GPUs Work, IEEE Computer, Feb 2007, pp 126-130

7 http://www.nvidia.com/object/cuda_home.html (Last Accessed 12 June 2009)

8 CUDA Programming Guide, Version 2.0, NVIDIA, June 2008

9 http://en.wikipedia.org/wiki/Protein_folding (Last Accessed 12 June 2009)

10 Y. Duan and P. A. Kollman, Computational protein folding: From lattice to all-atom, IBM Systems Journal, Vol 40, 2001

11 K. Dill, Theory for Folding and Stability of Globular Protein, Journal Bio-Chemistry, Vol-24, pp 1501-1509, 1985

12 http://www.obitko.com/tutorials/genetic-algorithms/ (Last Accessed 12 June 2009)

13 M. Burrows, D. Wheeler, A block sorting data compression algorithm, Technical report, DIGITAL System Research Center, 1994

14 S. Mantaci, A. Restivo, G. Rosone and M. Sciortino, An extension of the Burrows–Wheeler Transform, Journal of Theoretical Computer Science, pp 298–312, 2007

15 Ron Unger and John Moult, "Genetic Algorithms for Protein Folding Simulations", Journal of Molecular Biology, Vol-231, pp 75-81, 1993

16 Peter Clote and Rolf Backofen, http://www.cs.bc.edu/~clote/ComputationalMolecularBiology/ungerMoult.c (Last Accessed 12 June 2009)

17 T. Cormen, C. Leiserson, R. Rivest, C. Stein, Introduction To Algorithms, 2nd Edition, MIT Press and McGraw-Hill, 2001

18 http://www.ncbi.nlm.nih.gov/ (Last Accessed 12 June 2009)

19 Y. Cao, A. Janke, P.J. Waddell, M. Westerman, O. Takenaka, S. Murata, N. Okada, S. Paabo, M. Hasegawa, "Conflict among individual mitochondrial proteins in resolving the phylogeny of eutherian orders", Journal Molecular Evolution, Vol-47, pp 307–322, 1998