Accelerating a Bayesian Phylogenetic Inference Application with OpenACC

João Pedro de Matos Neves

Thesis to obtain the Master of Science Degree in Electrotechnic and Computer Engineering

Examination Committee
Chairperson: Doutor Nuno Cavaco Gomes Horta
Supervisor: Doutor Pedro Filipe Zeferino Tomás
Members of the Committee: Doutor Hérve Miguel Cordeiro Paulino

October 2013
Acknowledgments

I would like to say a big “Thank You” to Frederico Pratas. This work would not have been possible without his invaluable help.

I would also like to thank professor Pedro Tomás for the great assistance that he gave to me during the development process.

Professor Leonel Sousa, I’d like to thank you for the opportunity of working with GPUs in INESC-ID.

Last but not least, I dedicate this work to my girlfriend Mariana Gomes, to my mother Manuela Neves and my father Vitor Neves. This would not have been possible without their love and support.
Abstract

The need for faster computing has been around ever since the birth of the first computers. Faster hardware will almost always guarantee faster computing but occasionally the rate of hardware development is not enough for some programs to deal with the vast information they need. When these programs need to be accelerated, algorithmic optimizations have to be done that typically require changes to the program structure, in order to take advantage of parallel architectures, such as Graphics Processing Units (GPUs).

Several frameworks have been developed to take advantage of the GPUs available parallelism. However, this typically requires major changes to the original program as well knowledge about the target GPU architecture. OpenACC is a recent technology which targets the simplification of this process by giving compiler hints about the parallelization strategy.

This thesis targets the acceleration of an important bioinformatics application using OpenACC. Thus, two important results are taken: a) a performance comparison with CUDA; and b) a performance comparison with other parallel implementations of the program.

Results show: that CUDA can have up to 2 times the performance of OpenACC regarding a single kernel, using OpenACC an overall 4.1 speedup is achieved over the original serial MrBayes and that this implementation introduces some overhead when compared to the state of the art but scales much better for larger datasets than the latter.

Keywords

GPGPU programming frameworks, Phylogenetic Inference, OpenACC, MrBayes.
Resumo

Computação mais rápida é uma necessidade existente praticamente desde o nascimento dos primeiros computadores. Hardware mais rápido geralmente garante computação mais rápida, mas o ritmo de desenvolvimento do hardware por vezes não é rápido o suficiente para acompanhar as necessidades de processamento de grandes quantidades de dados de alguns programas. Quando estes programas precisam de ser acelerados, optimizações a nível do algoritmo são necessárias, que tipicamente envolvem modificações na estrutura do programa para tirar partido de arquiteturas paralelas como os Graphics Processing Units (GPUs).

Existem várias frameworks desenvolvidas para tirar partido do paralelismo disponível nos GPUs. No entanto, isto tipicamente envolve grandes modificações no programa original e também conhecimento sobre a arquitetura do GPU alvo. O OpenACC é uma tecnologia recente que visa simplificar este processo através de pistas que se dão ao compilador sobre a estratégia de paralelização.

Esta tese tem como alvo a aceleração de uma aplicação bioinformática importante usando OpenACC. Como tal, dois resultados importantes são retirados: a) uma comparação de performance com CUDA; e b) uma comparação de performance com outras implementações paralelas do programa.

Os resultados mostram: que o CUDA tem até o dobro da performance do OpenACC relativamente a um único kernel, usando OpenACC obteve-se um speedup global de 4.1 sobre o o programa série original é obtido e que a implementação obtida nesta tese introduz algum overhead sobre as outras implementações do estado da arte mas escala muito melhor para datasets maiores do que as últimas.

Palavras Chave

GPGPU programming frameworks, Phylogenetic Inference, OpenACC, MrBayes.
Contents

1 Introduction
   1.1 Motivation .................................................. 2
   1.2 Objectives .................................................... 3
   1.3 Main contributions ............................................. 4
   1.4 Dissertation outline ......................................... 4

2 State of the art
   2.1 GPUs .......................................................... 8
      2.1.1 Fermi Architecture ....................................... 8
      2.1.2 Kepler Architecture .................................... 11
   2.2 Parallel APIs .................................................. 14
      2.2.1 CUDA ...................................................... 14
      2.2.2 OpenACC ............................................... 15
   2.3 Performance evaluation of parallel systems ................. 21
      2.3.1 Speedup ................................................. 21
      2.3.2 Amdahl's law ........................................... 21
   2.4 MrBayes Application .......................................... 21
   2.5 Parallel Implementations of MrBayes ......................... 24
      2.5.1 Parallel Phylogenetic Likelihood Function ............ 24
      2.5.2 nMC$^3$ ................................................. 24
      2.5.3 oMC$^3$ ................................................ 25
      2.5.4 tgMC$^3$ ............................................... 26
   2.6 Testing framework ............................................ 27
   2.7 Profiling of the Original MrBayes .......................... 28
   2.8 Summary ...................................................... 30

3 From OpenACC to CUDA: a performance comparison .......... 31
   3.1 The Program .................................................. 32
   3.2 Parallelization Strategy ..................................... 34
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Fermi Architecture Overview</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Fermi SM and CUDA Core Overview</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Kepler Architecture Overview</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>Kepler’s SMX Overview</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>Simple CUDA kernel declaration and launch part 1.</td>
<td>16</td>
</tr>
<tr>
<td>2.6</td>
<td>Simple CUDA kernel declaration and launch part 2.</td>
<td>17</td>
</tr>
<tr>
<td>2.7</td>
<td>Simple CUDA kernel declaration and launch part 3.</td>
<td>18</td>
</tr>
<tr>
<td>2.8</td>
<td>Simple OpenACC example.</td>
<td>20</td>
</tr>
<tr>
<td>2.9</td>
<td>Rooted and Unrooted Phylogenetic Trees</td>
<td>22</td>
</tr>
<tr>
<td>2.10</td>
<td>Parallel Phylogenetic Likelihood function results</td>
<td>24</td>
</tr>
<tr>
<td>2.11</td>
<td>MrBayes serial and nMC³ algorithm comparison</td>
<td>25</td>
</tr>
<tr>
<td>2.12</td>
<td>oMC³ architecture overview [15]</td>
<td>26</td>
</tr>
<tr>
<td>2.13</td>
<td>Original MrBayes overview [16]</td>
<td>27</td>
</tr>
<tr>
<td>2.14</td>
<td>tgMC³ algorithm overview</td>
<td>28</td>
</tr>
<tr>
<td>2.15</td>
<td>Serial MrBayes execution time</td>
<td>29</td>
</tr>
<tr>
<td>2.16</td>
<td>Serial MrBayes execution percentage</td>
<td>29</td>
</tr>
<tr>
<td>3.1</td>
<td>Profiled functions with Kcache.grind</td>
<td>32</td>
</tr>
<tr>
<td>3.2</td>
<td>Kcache.grind profiling graph</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>CondLikeDown vs CondLikeRoot</td>
<td>33</td>
</tr>
<tr>
<td>3.4</td>
<td>Original CondLikeDown_NUC4</td>
<td>35</td>
</tr>
<tr>
<td>3.5</td>
<td>Original CondLikeRoot_NUC4</td>
<td>36</td>
</tr>
<tr>
<td>3.6</td>
<td>First modification of CondLikeDown_NUC4</td>
<td>37</td>
</tr>
<tr>
<td>3.7</td>
<td>First modification of CondLikeRoot_NUC4</td>
<td>38</td>
</tr>
<tr>
<td>3.8</td>
<td>First modification time graphic</td>
<td>39</td>
</tr>
<tr>
<td>3.9</td>
<td>First modification total speedup</td>
<td>40</td>
</tr>
<tr>
<td>3.10</td>
<td>Regular Dynamic Matrix Allocation</td>
<td>42</td>
</tr>
<tr>
<td>3.11</td>
<td>Optimized Dynamic Matrix Allocation</td>
<td>43</td>
</tr>
<tr>
<td>3.12</td>
<td>Master Data Region</td>
<td>43</td>
</tr>
</tbody>
</table>
List of Figures

3.13 Second modification of CondLikeDown_NUC4. ................................................. 44
3.14 Second Modification Execution Time. ................................................................. 45
3.15 Second Modification Functions Speedup. ......................................................... 46
3.16 CUDA vs. OpenACC ......................................................................................... 47

4.1 The original CondLikeScaler_NUC4 ..................................................................... 52
4.2 The first modification made to CondLikeScaler_NUC4 ........................................ 54
4.3 The first modification made to CondLikeScaler_NUC4 ........................................ 55
4.4 Execution Time for the third modification made to MrBayes. ............................... 55
4.5 Third modification speedup ................................................................................. 56
4.6 Second modification versus Third modification .................................................. 57
4.7 The original Likelihood_NUC4 function .............................................................. 58
4.8 The modification made to the Likelihood_NUC4 function ..................................... 59
4.9 Modified Likelihood Execution Time. ................................................................. 60
4.10 The Modified Likelihood Speedup. ................................................................. 61
4.11 The modification made to the RemoveNodeScalers function .............................. 61
4.12 The modification made to the CopySiteScalers function ..................................... 62
4.13 The modification made to the ResetSiteScalers function ..................................... 62
4.14 Second modification made to the CondLikeScaler_NUC4_OpenACC function . 62
4.15 Initialization of the single buffer on CPU ......................................................... 63
4.16 Allocation of the single buffer on GPU ............................................................... 63
4.17 Final Execution Time. ....................................................................................... 64
4.18 Detailed Final Execution Time ......................................................................... 65
4.19 Time Comparison between final and previous versions ..................................... 66
4.20 Detailed Final Speedup. ................................................................................... 67

5.1 Overview of the nMC$^3$. ...................................................................................... 71
5.2 nMC$^3$ v1.0 time comparison ............................................................................ 72
5.3 nMC$^3$ v2.1 Time Comparison ........................................................................... 73
5.4 tgMC$^3$ Execution Time Comparison .................................................................. 74
5.5 Comparison among all implementations ............................................................ 75

A.1 Final version of CondLikeDown_NUC4_OpenACC ............................................. 86
A.2 Final version of CondLikeRoot_NUC4_OpenACC .............................................. 87
A.3 Final version of CondLikeScaler_NUC4_OpenACC Part 1 .................................... 88
A.4 Final version of CondLikeScaler_NUC4_OpenACC Part 2 .................................... 89
A.5 Final version of Likelihood_NUC4_OpenACC ................................................... 90
A.6 Final version of RemoveNodeScalers_OpenACC ............................................... 90
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.7</td>
<td>Final version of CopySiteScalers_OpenACC</td>
<td>90</td>
</tr>
<tr>
<td>A.8</td>
<td>Final version of ResetSiteScalers_OpenACC</td>
<td>90</td>
</tr>
<tr>
<td>B.1</td>
<td>nMC$^3$ v1.0 Execution Time</td>
<td>92</td>
</tr>
<tr>
<td>B.2</td>
<td>nMC$^3$ v1.0 Speedup Comparison</td>
<td>92</td>
</tr>
<tr>
<td>B.3</td>
<td>nMC$^3$ v2.1 Execution Time</td>
<td>93</td>
</tr>
<tr>
<td>B.4</td>
<td>nMC$^3$ v1.0 and v2.1 Time Comparison</td>
<td>93</td>
</tr>
<tr>
<td>B.5</td>
<td>nMC$^3$ v2.1 Speedup Comparison</td>
<td>94</td>
</tr>
<tr>
<td>B.6</td>
<td>tgMC$^3$ Execution Time</td>
<td>94</td>
</tr>
<tr>
<td>B.7</td>
<td>tgMC$^3$ vs nMC$^3$ v2.1</td>
<td>95</td>
</tr>
<tr>
<td>B.8</td>
<td>tgMC$^3$ Speedup Comparison</td>
<td>95</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Number of floats transferred between CPU and GPU per iteration. . . . . . . . . . . 50
1 Introduction

Contents

1.1 Motivation .................................................. 2
1.2 Objectives ................................................... 3
1.3 Main contributions ........................................... 4
1.4 Dissertation outline ........................................ 4
1. Introduction

1.1 Motivation

Moore's Law has impacted the world of computing since its appearance in 1965 [17]. Roughly, Moore's Law states that the number of transistors in a chip doubles every 18 months [21]. For the last three decades, Moore's Law along with Dennard scaling [31] and device, circuit, microarchitecture, architecture, and compiler advances led to an exponential growth of microprocessor's performance [10]. Up to 2001, all of these microprocessors were single-core. Due to power and temperature constraints [37], the only way to take advantage of the increasing number of transistors and to keep the rate of performance growth experienced was to shift to a parallel multi-core architecture. The first multi-core microprocessor was IBM's POWER 4 [36] launched in 2001.

Accelerating software using more than one core to do so is called parallelism, in this sense a program accelerated with this technique would be called a parallel program. Taking advantage of the multi-core computer architectures to accelerate software (i.e. parallelism) is a problem not entirely solved that is currently being investigated. In this area of investigation, the same solution does not apply to different software.

When developing parallel software, one must be aware of its many pitfalls. Pitfalls such as race conditions and overheads due to synchronization, load balancing and memory transfers make it difficult to produce efficient parallel software. Race conditions can be very hard to find and fix. Overheads may be unavoidable and kill the acceleration. “In addition, many experienced software developers are more afraid of implementation faults such as memory errors than bad programming logic.” [30].

Despite the fact that microprocessors (CPUs) shifted to a core level parallel architecture in 2001 there is another component that had already shifted to a parallel architecture due to the nature of this chip's purpose: the GPU. The GPU was first designed to accelerate computer graphics computation. Calculating the colour of each pixel in a screen is a job with parallel nature and GPUs began to have parallel architectures during the 1990's [20]. The increased complexity of GPU architectures have lead them to go from specialized to programmable, to multi-core chips, with the recent ability to perform General Purpose Computation (GPGPU). Its parallel architecture has certain advantages over the CPUs', such as: modern GPUs have hundreds of cores[1] and are able to achieve 1 TFLOPS for single-precision (SP) arithmetic and over 80 GFLOPS for double-precision (DP) calculations [38]. GPGPU can deliver substantial performance gains over CPU computing alone but it requires the application itself to be adapted to the architecture of the target GPU and the application itself must be parallelizable enough [34]. Successfully taking advantage of the GPU's architecture to accelerate a general purpose application is a great challenge. Phylogenetics applications (e.g. MrBayes) possess the aforementioned characteristics to harness the parallel computing power of the modern GPU.

[1] The number of GPU cores in a common GPU is much higher than the number of CPU cores in a common CPU. A GPU core is much simpler than a CPU core though.
1.2 Objectives

Every organism that exists on Earth descend from a common ancestor. This means that, according to modern evolutionary theory, every set of species are related in spite of being already extinct or not. Phylogeny is the name given to this relationship between species. A graphical representation of phylogenetic relationships is called a phylogenetic tree. The scientific field that infers phylogenetic trees from the species’ morphological, physiological, and molecular characteristics is called phylogenetics. A phylogenetic tree containing every species that still exists or is already extinct is called the “tree of life”. A more specialized phylogenetic tree (that may contain molecular phylogeny instead of species) is useful to support comparative studies, test biogeographic hypotheses, evaluate mode and timing of speciation, infer amino acid sequence of extinct proteins, track the evolution of diseases, and even provide evidence in criminal cases [5].

MrBayes is a bioinformatics program to reconstruct phylogenetic trees. MrBayes has to deal with a very large amount of data and therefore it takes a significant amount of time to compute the larger inputs that are required. In order to accelerate MrBayes related research it is necessary to accelerate the MrBayes program itself. There are several ways of accelerating a program and choosing the right one can be a problem in itself, however this thesis will use a particular framework and technology to accelerate MrBayes.

Since MrBayes has the aforementioned relevance, it is only natural that this is not the first attempt to accelerate this program. In fact, many attempts were made to accelerate MrBayes using mostly multi-core CPUs, networks of computers and GPU computing with CUDA. From the mentioned technologies, CUDA has had the most success [25].

CUDA is an extension of the commonly used C language and allows the programmer to write general purpose code to run on the GPU in a SIMD fashion. This requires the programmer to understand CUDA a write separated code for CPU and GPU. OpenACC is a recent programming framework that lifts the burden of having to write code for the GPU. OpenACC works with compiler directives which the programmer should use to tell the compiler which parts of the code are to be executed on the GPU [32]. This saves time to the programmer and make it easier for one to get used to GPGPU.

1.2 Objectives

The proposed objectives for this thesis are described bellow:

- Accelerate MrBayes using OpenACC technology.

- Find and reduce as much as possible the bottlenecks in the parallelization of MrBayes.

- Compare OpenACC technology with CUDA technology.
1. Introduction

- Compare the implemented parallel MrBayes with other state of the art parallel implementations.

1.3 Main contributions

The main contributions of this thesis include using the new and emerging technology of OpenACC and explaining what type of OpenACC features are best suited to this kind of work and which do not fit so well. This thesis also details where the overall speedup comes from and in which sections of the program speedup can be achieved with OpenACC. A comparison in terms of performance between the OpenACC technology and the CUDA technology is presented. The memory transfers between CPU and GPU are minimized and some details of some of the implementation issues that the programmer needs to be aware of with OpenACC are presented. A good profile of MrBayes is presented for both its serial and parallel versions. This thesis compares the implementation achieved with other CUDA based state of the art implementations. An overall speedup of 4.1 over the original serial MrBayes was achieved.

1.4 Dissertation outline

This thesis is organized in the following manner:

- **Chapter 2** - An overview of every technological aspect that the reader should be familiar with to understand the contents presented later on. This overview includes: Hardware - CPUs and GPUs; Software - the available software that can be useful for a work of this type; Evaluation Metrics; The Application under study (MrBayes 3.2.1) and already existent similar parallelization work (state of the art).

- **Chapter 3** - MrBayes is thoroughly studied, analysed, profiled and the bottleneck is identified. With this information, the first parallelization strategy is outlined. This strategy represents a two part implementation done in Sections 3.3 and 3.4. These Sections describe the implementation procedures and present the most relevant results obtained with these procedures. At the end of Chapter 3, a comparison between OpenACC and CUDA is done.

- **Chapter 4** - This Chapter is entirely dedicated to improving the overall performance of the program. To do this, the new bottleneck is identified. With this information an implementation is performed in three parts. Each of these parts present the implementation procedures and the most relevant results much like what is done in Chapter 3.

- **Chapter 5** - The necessary information to comprehend the tests done in this thesis is presented. This information contains:
  - Testing framework - a description of the hardware of the computer used for testing.
1.4 Dissertation outline

- Inputs - a small description of the input files that were used for testing.
- Original MrBayes - an extensive test done to the original MrBayes 3.2.1. This test is the basis for every speedup calculation in the thesis.

In this Chapter, a comparison between the final version of this thesis and some of the most similar parallelizations done with MrBayes is also presented.

• Chapter 6 - The conclusions from this thesis and future work on how to improve the results obtained in the final version of this thesis are presented.
1. Introduction
2. State of the art

The Central Processing Unit (CPU) is one the most important components in a computer as it is responsible for processing everything the whole system needs. It is also the component used whenever science and research demand information processing.

Current CPU chips usually have more than one CPU inside them. It is commonly called CPU to the whole package and core to each of the processing elements inside the same chip. Thus a CPU with more than one core is called a multi-core CPU and a CPU with just one core is called single-core CPU.

Notwithstanding that the CPU be a powerful information processing device, there are other devices on a computer that, when used in conjunction with a CPU, can yield better results than a single multi-core CPU. Certain applications have a great potential for parallelism and a GPU can take better advantage of this great parallelism potential. This is due to the fact that modern GPUs have hundreds of cores, even though each core is simpler and slower than a CPU core [3]. The next Section provides greater detail on GPU architectures.

2.1 GPUs

Since NVIDIA GPUs will be used, this section will describe two important NVIDIA GPU architectures: Fermi and Kepler. Both of these architectures are divided into several small processors (cores) called CUDA cores. These CUDA cores are then grouped into Streaming Multiprocessors (SM). The main reason behind the usage of NVIDIA GPUs is the fact that NVIDIA’s GPUs are the only ones that support both CUDA and OpenACC.

In modern computer architectures, the GPU is connected to the CPU by means of a PCI-E interface, and act as specialized accelerators. Normally, GPUs are used for graphics processing, however, due to architectural improvements, they can now be used to accelerate general purpose applications. The PCI-E interface is currently in its 3.0 version. All of the memory transfers between CPU and GPU are done through the PCI-E interface. The memory transfer rate of PCI-E is, however, much slower than the GPU accessing its own memory [18].

2.1.1 Fermi Architecture

The Fermi architecture was first introduced by NVIDIA in 2010 with the GeForce 400 series graphics cards. This architecture is composed of a total of up to 512 CUDA cores distributed among 16 Streaming Multiprocessors (SMs) with 32 cores each [23]. These 32 CUDA cores in each SM are further divided into two blocks of 16 CUDA cores that are called execution blocks. Besides the CUDA cores, each SM also has 16 Load and Store units (LD/ST) and 4 Special Function Units (SU). These SU execute special mathematical functions such as sin, cosine, reciprocal, and square root [23] but only four of these operations can be issued per clock cycle. It is possible to use the load and store instructions with memory references in terms of $x$ and $y$.
2.1 GPUs

Figure 2.1: An overview of NVIDIA’s Fermi Architecture [23].

values to access two-dimensional arrays.

Each SM has its own 64 KiB of memory which can be split between L1 cache and shared memory in the following manner: 16KiB/48KiB or 48KiB/16KiB. Shared memory supplies low latency memory to the CUDA cores within the same SM. The decision to allocate more shared memory or cache is dependent on whether the kernel (code segment that runs on the GPU) requires more shared memory or makes more frequent accesses to the global memory. The global memory is usually the off-chip DRAM. Data format can be converted (e.g. from float to double or vice-versa) at the full rate when it is transferred from the core registers to DRAM or vice-versa. These are optimizations that can only be found on GPUs since they are not worthwhile in CPUs [8].

The SMs are grouped around an up to 768KiB L2 common cache as it is shown in Figure 2.1. This L2 cache implements a set of atomic read-modify-write operations. Because these atomic operations are uninterruptible, they are ideal for thread synchronization and for preventing data races. The atomic read-modify-write operations feature is faster on GPUs since on CPUs this has to be done in a two step process: an atomic operation (test-and-set) is used by the CPU to open a semaphore and the semaphore then manages the memory location. In addition to the L2 cache the Fermi architecture also has up to 6GiB of off-chip GDDR5 DRAM.

Each CUDA core in a SM has its own Integer Unit (INT Unit) and Floating Point Unit (FP Unit) and the operations of the latter follow the IEEE 754-2008 floating-point standard. Besides integer and floating operations, a CUDA core also supports a Fused Multiply-Add (FMA) operation. It is possible to complete a single-precision FMA in a single clock cycle or a double-precision FMA in two clock cycles. FMA operations also follow the IEEE 754-2008 standard. An overview of the
2. State of the art

Figure 2.2: An overview of the Fermi’s SM and CUDA Core [23].

SM architecture and a single CUDA core architecture can be seen in Figure 2.2.

A maximum of 1536 threads can be run on the up to 512 CUDA cores. These threads are grouped into thread blocks. A single SM executes all of the threads that belong to the same thread block. This allows the threads in the same thread block to share memory (shared memory or L1 cache) and cooperate among themselves. The user has no control over the execution sequence of the various thread blocks and they can run either sequentially or concurrently. However, it is possible for the thread blocks to coordinate the use of global shared memory. Both thread blocks and threads have unique IDs which can be used by the programmer to determine what part of the memory a certain thread will access. Within each thread block there is a maximum of 32 threads (one per CUDA core) and a set of threads of the same thread block is called a warp. A warp can contain up to 32 threads. It is possible to issue two separate warps from different thread blocks at the same time and have them execute concurrently [8].

A group of thread blocks is called a grid. Each grid can only execute a single kernel at a time. A kernel is a general purpose code segment to run on the GPU that executes on a grid. The Fermi architecture is able to execute more than one kernel at the same time on the same GPU. This was done to maximize the GPU usage and prevent a kernel with a small thread count from
rendering most of the device useless during its execution. There is a limit of 16 concurrent kernels to be executed at one time [23].

There is dedicated hardware in the Fermi architecture to manage thread scheduling. This hardware is called GigaThread and can be seen in Figure 2.1. GigaThread is another feature where the CPU and GPU architectures differ. CPU architectures do not have dedicated thread schedulers and any thread can execute on any available CPU core at any given time. This grants the CPUs with the general processing power they need but Fermi's approach does not require a complex processing unit for this purpose. On the other hand, Fermi architecture is able to have dedicated hardware schedulers because the intended applications, principles of stream processing, and the kernel and thread model, were all known in advance, which is not the case with CPUs. GigaThread also contains two streaming data-transfer engines. It is common practice to use on the engines to transfer data from CPU to GPU and the other to transfer data from GPU to CPU [8].

2.1.2 Kepler Architecture

The Kepler architecture was first introduced by NVIDIA in 2012 with the GeForce 600 series graphics cards and has some key new features which translate into some advantages over the Fermi architecture:

- Dynamic Parallelism - this new feature enhances the GPU with the capability to generate new work for itself, synchronize on results and control the scheduling of that work via dedicated, accelerated hardware paths [24]. The GPU is able to perform these tasks without the assistance of the CPU.

- Hyper-Q - this feature allows for more than one CPU core to launch kernels on the same GPU at the same time. This maximizes GPU utilization and minimizes CPU idle times.

- Grid Management Unit (GMU) - the GMU is the hardware that enables Dynamic Parallelism. The GMU schedules the grids to be executed on the GPU and can queue and pause grids until they are ready to execute.

- NVIDIA GPUDirect™- this feature allows multiple GPUs in the same computer or multiple GPUs in different servers in the same network to transfer data directly between them. This doesn’t require for the data to pass through CPU/system memory. In addition a Remote Direct Memory Access (RDMA) feature, that GPUDirect™has, gives third party devices direct access to the memory of the various GPUs residing in the same system. This greatly reduces the latency of send and receive messages either to or from the GPU that MPI uses.

\[\text{This feature is only available in the GK100 GPU chip or later.}\]
2. State of the art

Figure 2.3: An overview the NVIDIA’s Kepler architecture [24].

This architecture has up to 2880 CUDA Cores distributed among 15 Streaming Multiproces-
sors (now called SMX) with 192 CUDA cores each [24]. The SMXs are grouped around an L2 cache much like the SMs of the Fermi architecture. This can be seen in Figure 2.3.

The SMX is somewhat different from the SM. There are still CUDA Cores, LD/ST Units and SFUs but a new Double Precision Unit (DP Unit) is introduced. The CUDA Core has the same base architecture as the Fermi’s CUDA Core and the Kepler architecture’s single and double-precision operations are still fully compliant with the IEEE 754-2008 standard. Each SMX has 192 CUDA Cores, 64 DP Units, 32 SFUs and 32 LD/ST Units. The SMX also features four Warp Schedulers instead of two in the Fermi architecture. An overview of the SMX can be seen in Figure 2.4.

The memory hierarchy of the Kepler architecture is organized in a similar manner of the one of the Fermi architecture. Each SMX has 64KiB of dedicated memory. Much like the Fermi architecture, this 64KiB can be configured/distributed between shared memory and L1 cache, using the following shared memory/L1 cache configurations: 48KiB/16KiB, 16KiB/48KiB and 32KiB/32KiB. In addition to this, the SMX also features 48KiB of cache that is read-only for the duration of the utilization. This cache already existed in the Fermi architecture but it could only be accessed by the Texture unit. In the Kepler architecture this cache can be accessed freely by the SMX thus allowing its usage for GPGPU. The L2 cache is up to 1536KiB which is up to twice as much as in the Fermi architecture [24].
2.1 GPUs

Figure 2.4: An Overview of Kepler's SMX [24].
2. State of the art

2.2 Parallel APIs

In order to facilitate the use of emerging parallel architectures, several APIs (Application Programming Interface) have been developed, including CUDA, OpenCL and OpenACC. In the following sections CUDA and OpenACC will be described in detail: CUDA because it was developed by NVIDIA to fully take advantage of the GPU processing power and OpenACC because it greatly simplifies programming.

2.2.1 CUDA

CUDA (Compute Unified Device Architecture) is a “parallel computing platform and programming model” developed by NVIDIA [25]. CUDA is available for C/C++ and Fortran and enables the programmer to write parallel code for the GPU.

CUDA is a high level language (an extension of the ANSI C language [1]) which allows the programmer to write functions\(^2\) that run on each CUDA core of the GPU in an SIMD (Single Instruction Multiple Data)\(^3\) fashion. This means that every CUDA core will process the same instruction but on different data. In addition to this, CUDA has some built-in functions to transfer memory from and to the GPU.

A kernel in CUDA can be declared using the “\_global\_” declaration specifier. The number of threads a kernel will launch can be specified with the “<<<...>>” syntax. The unique thread ID mentioned in Section 2.1.1 can be accessed through the variable “threadIdx”. As an example, a simple kernel and its launching procedure is shown in Figures 2.5, 2.6 and 2.7. The kernel itself is represented only in the first few lines of Figure 2.5. Variable “blockIdx” is used to access the blocks unique IDs and variable “blockDim” contains the dimension of the block. Thus variable “i” contains the index of the array that each thread calculates and each thread access a specific part of the arrays [26]. The rest of Figures 2.5, 2.6 and 2.7 represent the necessary steps needed to use CUDA. In short, these steps are:

1. Allocate and initialize the arrays on the host side (CPU).
2. Allocate the arrays on the device side (GPU) - using the “cudaMalloc” function.
3. Transfer the necessary data from the host to the device - using the “cudaMemcpy” function.
4. Launch the CUDA kernel to process the data on the GPU - using the following line of code: “vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);”. The “blocksPerGrid” variable contains the number of thread blocks to be launched and the “threadsPerBlock” variable contains the number of threads each block should contain. A CUDA kernel can have several arguments if needed just like a regular C function.

\(^2\)These GPGPU functions that run on the GPU are called kernels.
\(^3\)NVIDIA, calls this SIMT (Single Instruction Multiple Thread) rather than SIMD but the meaning of both are similar [33].
5. Transfer the results from the device back to the host - using the “cudaMemcpy” function.

6. Free the allocated device memory - this is done with the “cudaFree” function.

7. Free the allocated host memory.

CUDA related functions usually have their names begin with “cuda”. The “cudaMemcpy” is used for transferring data both from and to the GPU. The distinction between the two cases is done with the last argument of the function: “cudaMemcpyHostToDevice” defines a memory transfer from the CPU to the GPU and “cudaMemcpyDeviceToHost” defines a memory transfer from the GPU to the CPU. More detailed information for the various features of CUDA can be found in [26].

2.2.2 OpenACC

Much like what OpenMP [4, 29] does for threads, OpenACC does for CUDA. OpenACC allows the programmer to not have to worry about coding in parallel. The only thing the programmer needs to do is to give some hints to the compiler (called pragmas) as to indicate the regions of the serial code that can be parallelized to run on the GPU. These hints are a “collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators” [27]. This allows the programmer to parallelize code much faster than with CUDA, but like OpenMP one looses some control over what the GPU might be executing.

As it shall be seen later in this document (Section 2.4), MrBayes is a rather large program and requires quite some functions to be worked on. Because of this and considering that a single GPU has more potential for parallelism than a single multi-core CPU, this thesis is focused on the use of OpenACC along with NVIDIA GPUs.

OpenACC first appeared as a standard in November, 2011 with its 1.0 version specification [28]. In this specification there are the most basic pragmas available to be able to write a parallel kernel. Some of the most important pragmas are:

- parallel - allows to transform a single for loop into a parallel kernel. Can be used with the loop construct to specify how the parallelization is done.

- kernels - similar to the parallel pragma but attempts to parallelize the inner nested for loops as well. Can be used with the loop construct to specify how the parallelization is done.

- data - allows management of data between the CPU and GPU within the surrounding region of code. It is used with the following directives:

---

4 Today’s most advanced CPU’s have at most 8 hyper threaded cores (as seen on the latest Intel Xeon [11]) versus the up to 2880 CUDA cores of a GPU, as seen in Section 2.1.2.

5 A pragma is a compiler directive that informs the compiler of the program of something. In the case of OpenACC, it can inform the compiler that a section of code is to be executed on the GPU or that some data needs to be moved from or to the GPU.

6 As mentioned earlier, a kernel is the name given to a General Purpose function that runs on the GPU.
2. State of the art

```c
/**
 * CUDA Kernel Device code
 * Computes the vector addition of A and B into C. The 3 vectors have the same
 * number of elements numElements.
 */
__global__ void vectorAdd(const float *A, const float *B, float *C, int numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < numElements)
    {
        C[i] = A[i] + B[i];
    }
}

/**
 * Host main routine
 */
int main(void)
{
    // Error code to check return values for CUDA calls
    cudaError_t err = cudaSuccess;

    // Print the vector length to be used, and compute its size
    int numElements = 50000;
    size_t size = numElements * sizeof(float);
    printf("[Vector addition of %d elements]\n", numElements);

    // Allocate the host input vector A
    float *h_A = (float *)malloc(size);
    // Allocate the host input vector B
    float *h_B = (float *)malloc(size);
    // Allocate the host output vector C
    float *h_C = (float *)malloc(size);

    // Verify that allocations succeeded
    if (h_A == NULL || h_B == NULL || h_C == NULL)
    {
        fprintf(stderr, "Failed to allocate host vectors!\n");
        exit(EXIT_FAILURE);
    }

    // Initialize the host input vectors
    for (int i = 0; i < numElements; ++i)
    {
        h_A[i] = rand()/(float)RAND_MAX;
        h_B[i] = rand()/(float)RAND_MAX;
    }

    // Allocate the device input vector A
    float *d_A = NULL;
    err = cudaMalloc((void **)&d_A, size);
    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to allocate device vector A (error code \%s)!\n", cudaMemcpyErrorString(err));
        exit(EXIT_FAILURE);
    }

    // Allocate the device input vector B
    float *d_B = NULL;
    err = cudaMalloc((void **)&d_B, size);
    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to allocate device vector B (error code \%s)!\n", cudaMemcpyErrorString(err));
        exit(EXIT_FAILURE);
    }
}
```

Figure 2.5: A simple CUDA kernel declaration and launch. This kernel simply adds two vectors [26], Part 1.
2.2 Parallel APIs

// Allocate the device output vector C
float *d_C = NULL;
err = cudaMalloc(&d_C, size);
if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to allocate device vector C (error code %s)\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}

// Copy the host input vectors A and B in host memory to the device input vectors in
// device memory
printf("Copy input data from the host memory to the CUDA device\n");
err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to copy vector A from host to device (error code %s)\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}
err = cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to copy vector B from host to device (error code %s)\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}

// Launch the Vector Add CUDA Kernel
int threadsPerBlock = 256;
int blocksPerGrid = (numElements + threadsPerBlock - 1) / threadsPerBlock;
printf("CUDA kernel launch with %d blocks of %d threads\n", blocksPerGrid, threadsPerBlock);
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);
err = cudaGetLastError();
if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to launch vectorAdd kernel (error code %s)\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}

// Copy the device result vector in device memory to the host result vector
// in host memory.
printf("Copy output data from the CUDA device to the host memory\n");
err = cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to copy vector C from device to host (error code %s)\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}

Figure 2.6: A simple CUDA kernel declaration and launch. This kernel simply adds two vectors [26]. Part 2.
2. State of the art

```c
// Free device global memory
err = cudaFree(d_A);
if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to free device vector A (error code %s)!\n",.cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}
err = cudaFree(d_B);
if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to free device vector B (error code %s)!\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}
err = cudaFree(d_C);
if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to free device vector C (error code %s)!\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}

// Free host memory
free(h_A);
free(h_B);
free(h_C);

// Reset the device and exit
err = cudaDeviceReset();
if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to deinitialize the device! error=%s\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}
printf("Done\n");
return 0;
```

Figure 2.7: A simple CUDA kernel declaration and launch. This kernel simply adds two vectors [26]. Part 3.
2.2 Parallel APIs

- **copy** - allocates the memory on the GPU side, copies the memory to the GPU when entering the data region and copies the data back to the CPU when exiting the data region.

- **copyin** - similar to the **copy** directive but does not copy the data back to the CPU when exiting the data region.

- **copyout** - similar to the **copy** directive but does not copy the data into the GPU when entering the data region.

- **present** - informs the compiler that the specific data is already on the GPU. If, for some reason, the data is not on the GPU, a runtime error is yield and the program exits.

- **create** - allocates the data on the GPU but does not perform any other action.

- **update** - allows to manually transfer data to or from the GPU. It is always used with additional directives.
  - **host** - transfers memory from the GPU to the CPU.
  - **device** - transfers memory from the CPU to the GPU.

It is worth noting that if the parallel or the kernels pragma is introduced in the code without any further specification, the compiler will attempt to perform the best parallelization it can. However, experimental results obtained during the implementation of this thesis suggest that it is almost always better to specify how the parallelization is done manually.

In order to have a better understanding of how OpenACC works, a simple example is presented in Figure 2.8. Simply put, this code generates a matrix with floats, transforms this matrix and then sends the results back to the CPU. The first pragma in the program is a data directive that defines a data region where memory can be transferred and managed on the GPU. Here the compiler is told that the *matrix* variable is to be allocated in the GPU and that the data should be transferred from the CPU to the GPU at the beginning of the data region and back to the CPU at the end of the region. The compiler should allocate the *scaler* variable and transfer the results back to the CPU at the end of the region and it should also allocate the *aux_scaler* variable.

Inside the data region there are two outer for loops to be parallelized. On the first loop the **private** clause is telling the compiler that each GPU thread should have its own copy of the *aux_scaler* variable in order to avoid unnecessary data synchronism. The **reduction** clause on the inner loop is telling the compiler that it should chose the maximum value obtained for the *aux_scaler* variable for each of the outer loops. The second main loop is much simpler and the **independent** clause simply tells the compiler to override its check for data dependencies between loops and assume that the loops are data independent. This is necessary because in complex loops the compiler is unable to verify that some loops are indeed data independent. When this compiler check fails no parallelization is done whatsoever.
Figure 2.8: A simple OpenACC example.
2.3 Performance evaluation of parallel systems

Evaluation is an important topic in almost every area of engineering. Computational performance is no exception to this “rule” and this thesis needs a way of evaluation and comparison with other similar work. Thus a less detailed and a more detailed version of the same method of parallelism evaluation is presented in this Section.

2.3.1 Speedup

Speedup is probably the most common way of evaluating an acceleration work. As the name itself implies, speedup is the number of times a program is faster than the original. Let $S$ be the speedup, $t_{\text{Original}}$ the time of execution of the original program and $t_{\text{Parallel}}$ be the time of execution of the parallel program. The speedup is given by:

$$S = \frac{t_{\text{Original}}}{t_{\text{Parallel}}} \quad (2.1)$$

Although Eq. 2.1 describes the speedup of an entire program, it can also be used to compute the relative performance of a program section or function.

2.3.2 Amdahl’s law

Amdahl’s law is a more detailed take on Speedup and it is mostly used to verify the impact of the serial part of a parallel program in the global performance of the program. The Amdahl’s law for parallel computation is as follows [2]:

$$\text{Speedup} = \frac{1}{r_s + \frac{r_p}{n}} \quad (2.2)$$

In equation 2.2, $r_s$ is the serial fraction of the program, $r_p$ is the parallel fraction of the program and $n$ is the number of processors (i.e. cores) that are available. Note that $r_s$ and $r_p$ must always verify $r_s + r_p = 1$. It is important to notice that this kind of analysis is more suitable when using only CPUs since the GPU cores are much simpler that a CPU core and thus a GPU core cannot be directly compared with a CPU core.

2.4 MrBayes Application

The history and origin of species can be described through phylogenies. Phylogenies not only give a deeper insight of the species’ past but can also be used to predict their future [35].

MrBayes is a bioinformatics application. Bioinformatics consists in the use of computational knowledge and power to deal with the great amount of data that biology research needs to process nowadays. Due to the great amount of data this kind of research has to process, most of today’s challenges in biology are also challenges in computing [22]. Biology is “the study of living
organisms, divided into many specialized fields that cover their morphology, physiology, anatomy, behaviour, origin, and distribution” [14].

MrBayes uses the DNA information of multiple species and attempts to reconstruct the phylogenetic tree of that group of species. Phylogenesis is the branch of biology that studies how a certain species, group of organisms or a particular feature of an organism evolves and diverges in the course of time. Thus a phylogenetic tree is a schematic representation in a form of a tree of the process of evolutionary development and diversification just mentioned. In other words, it is the representation of the ancestral relationships among a group of organisms through a graph [9].

There are two types of phylogenetic trees: rooted and unrooted. A rooted tree is a better representation of the phylogenetic history of species and an unrooted tree is used best to represent the correlation between species [19]. There’s an example of both a rooted and an unrooted tree in Figure 2.9 where tree A is an unrooted tree and tree B is a rooted tree.

Phylogenetic tree reconstruction is the act of attempting to obtain the original phylogenetic tree, for a certain group of species, through an approximation that is based on the data available nowadays, such as corresponding DNA sequences [9]. The problem with phylogenetic tree reconstruction is that a very small number of species can lead to a very large number of different possible trees. For example, the number of trees \( B(s) \) in a rooted-tree like structure composed of \( s \) different species is [3]:

\[
B(s) = \frac{(2s-3)!}{2^{s-2}(s-2)!}
\]

Which leads to a total of \( \sim 3 \times 10^7 \) possible trees for 10 species, or \( \sim 2 \times 10^{14} \) trees for 15 species.

There are several different methods to perform phylogenetic tree reconstruction also referred to as phylogenetic inference. Some examples of such methods are: the parsimony method, distance methods, maximum likelihood and Bayesian inference. There are some advantages in using Bayesian over the other methods such as easy interpretation of results. Phylogenetic trees' posterior probabilities are the base for the actual phylogenetic inference in the Bayesian inference method. Bayes theorem can be used to calculate these posterior probabilities. For example, the \( i \)th phylogenetic tree \( (\tau_i) \) conditional on an alignment of DNA sequences \( (X) \) has a posterior
probability of:

\[ f(\tau_i | X) = \frac{f(X | \tau_i) f(\tau_i)}{\sum_{j=1}^{B(s)} f(X | \tau_j) f(\tau_j)}, \quad (2.4) \]

where \( B(s) \) is the same mentioned on equation 2.3.

\[ f(X | \tau_i) = \int_{v} \int_{\theta} f(X | \tau_i, v, \theta) f(v, \theta) \, dv \, d\theta, \quad (2.5) \]

\( v \) represents the combinations of branch lengths, \( \theta \) the substitution parameters, \( f(\tau_i) \) is the prior for phylogenetic trees, which is usually set to \( f(\tau_i) = \frac{1}{B(s)} \), and \( f(v, \theta) \) is the prior on branch lengths and substitution parameters. \( f(X | \tau_i, v, \theta) \) is the likelihood function and its calculation is done assuming that a time-homogeneous Poisson Process accurately describes the occurrence of the substitutions. Both the maximum likelihood analysis and the Bayesian inference methods can use the same models for DNA substitution [13].

Alas, the summations and integrals presented in equations 2.4 and 2.5 (required in the Bayesian analysis) cannot be evaluated analytically. Thus MrBayes obtains an approximate result for the trees’ posterior probabilities using Markov Chain Monte Carlo (MCMC). Fundamentally, the MCMC algorithm has the following steps:

1. Resorting to a stochastic mechanism, a new state for the chain is proposed.
2. Each new state has its acceptance probability calculated.
3. A random variable is drawn (between values 0 and 1) and the acceptance of this new state depends on it. The new state is accepted only if the number drawn is lower than the acceptance probability, otherwise it is rejected.

This process occurs millions of times. Throughout the computation of a chain every tree is visited a certain amount of time. If a single tree is taken into account, then the proportion of time this tree was visited, when the chain was being computed, is a valid approximation of this trees’ posterior probability.

In addition to the MCMC algorithm, MrBayes also implements another algorithm called Metropolis-coupled Markov chain Monte Carlo or (MC)\(^3\) for short. This is a variation of the MCMC algorithm that runs \( n \) chains from which \( n - 1 \) are heated. The steady state distribution for the heated chains is \( f(\tau_i | X)^\beta \). Incremental heat is used. The \( i \)th chain has an applied heat of \( \beta = \frac{1}{1 + (i - 1)T} \), where \( T \) is a user set heating parameter. After every chain has completed one step, an attempt is made to exchange two chains that were randomly selected. The two randomly selected chains will only switch states if the exchange is accepted. Only the states for the cold chain (\( \beta = 1 \)) are used to base inferences on. The heated chains have greater freedom to explore the tree space. The objective of having heated chains is to lower the peaks and fill in valleys. It is worth noting that there is the possibility of the cold chain jumping across deep valleys if a successful exchange occurs between the cold chain and a heated chain. It is possible to verify experimentally that better results are obtained using the (MC)\(^3\) algorithm [13].
2. State of the art

Figure 2.10: Frequency scaled total time for all systems, real data set [7].

MrBayes’ interface is a regular command line where it is possible to tweak some parameters of the analysis. The input files for MrBayes contain an aligned matrix of DNA or amino acid sequence. This matrix comes in the standard NEXUS format. MrBayes has a $4 \times 4$ DNA substitution model implemented that is as general as it can be. It also has some substitution models for amino acids, but this thesis will focus on DNA only.

2.5 Parallel Implementations of MrBayes

There are many more parallel implementations of MrBayes than the ones discussed here. However, since this thesis is focusing on GPU parallelization, all of the parallelization strategies presented here are for (or at least work with) GPUs.

2.5.1 Parallel Phylogenetic Likelihood Function

The first parallel implementation of MrBayes was performed by Pratas et. al. [7]. In this work, the authors, tried to compare the performance of different computing architectures for philogenetic inference, namely: multi-core CPUs, IMB’s Cell processor and NVIDIA’s GTX 285 GPU.

Only the Likelihood function was parallelized which represents important work. Due to the amount of communication between the CPU and the GPU and the lack of further parallelization, it resulted in a reduced overall performance (see Figure 2.10).

2.5.2 nMC$^3$

The nMC$^3$ version of MrBayes is an improvement made over the previously referred implementation. The overall structure of the algorithm compared to the serial MrBayes can be seen in Figure 2.11.
2.5 Parallel Implementations of MrBayes

(a) Serial MrBayes

(b) nMC$^3$ parallel version.

Figure 2.11: Comparison between the serial MrBayes with nMC$^3$’s parallel algorithm [12].

Essentially, the nMC$^3$ algorithm has the following improvements:

- Instead of having either the CPU or the GPU performing computations, nMC$^3$ has both the CPU and the GPU working at the same time.

- nMC$^3$ uses a pipelining strategy to keep both CPU and GPU working for the maximum time possible.

- nMC$^3$ reduces the number of total CPU-GPU transfers by transferring stages 1 and 3 each as a single batch. This reduces the number of memory transfers to 2 per iteration.

- On stage 3, the site likelihoods are also computed by the GPU, effectively postponing the necessary data transfer needed and reducing the amount of data that needs to be transferred.

All of this causes nMC$^3$ to have an overall much better performance than the implementation on Section 2.5.1 [12]. As we have seen in this Section, memory transfers are a very important topic in GPGPU. A careful use of memory transfers is required to obtain good results.

2.5.3 oMC$^3$

oMC$^3$ is an improvement over nMC$^3$ and utilizes OpenMP, MPI and CUDA to join the benefits of nMC$^3$ with the advantages of hMC$^3$. Since this algorithm is using MPI, it targets multiple computers each with a multi-core CPU and at least one GPU. Each one of these computers is

---

7GPGPU refers to General Purpose GPU or General Purpose Graphics Processing Unit.
8hMC$^3$ is the name the authors of [15] gave to a parallelized MrBayes algorithm detailed in [39]. However, this algorithm does not use the GPU in any way.
2. State of the art

One of the advantages of this algorithm over nMC$^3$ is that a situation where a CPU core would remain unused for the entire run of the program no longer happens. The balanced work distribution between CPU cores and GPUs allows this algorithm to harvest more of the computational power of a single node than nMC$^3$.

In spite of this algorithm’s better results when compared to nMC$^3$ there’s a disadvantage to this approach. In order for this algorithm to achieve these results, a manual profiling of the system in question is necessary in order to the workload division to work properly. It is also worth mentioning that the results presented were obtained with the TianHe-1A supercomputer [15].

2.5.4 tgMC$^3$

Much like oMC$^3$, tgMC$^3$ is based on the nMC$^3$ algorithm, but it improves it substantially. The main idea behind this algorithm is to fuse all of the kernels$^9$ used in nMC$^3$ into a single kernel in order to prevent excessive global to local memory transfers on the GPU side [16]. According to the authors the overview of the original MrBayes algorithm can be seen in Figure 2.13. The main

---

$^9$Kernel is the name given to a general purpose function that runs on the GPU.
2.6 Testing framework

MrBayes is a rather large program and can be executed in very different modes. In this thesis all the tests are made with MrBayes in `DEBUG,NOSHORTCUTS` mode. This prevents MrBayes from taking some shortcuts in the calculations. Also for the purpose of this thesis MrBayes

architecture proposed for this algorithm can be found in Figure 2.14.

There are a few key advantages of the `tgMC³` algorithm:

- `tgMC³` combines several functions into a single kernel which lowers the complexity of kernel launching. This was achieved by analysing the data dependencies present in MrBayes `MC³`.

- Memory transfers regarding the `tip` and `prelike` matrices were optimized from the implementation done in `nMC³`.

- Several improvements were made regarding the scaling step of the program. Some branches were avoided to speedup the computation.

This is one of the most recent algorithms published to run MrBayes on GPU.

![Figure 2.13: Original MrBayes overview](image)
is used for analysing DNA sequences in a 4 by 4 nucleotide substitution model. A computer with a quad-core CPU and one GPU is used to run all of the tests presented. This computer has the following specifications:

- CPU - Intel Core i7 950 @ 3.07GHz.
- RAM - (2x) Corsair XMS DDR3 3x2GB @ 2GHz.
- Motherboard - ASUS P6T SE.
- GPU - (MSI) NVIDIA GeForce GTX 580.

For evaluating the results, several datasets were used, considering the analysis of a set of 10, 20, 50 or 100 species, each using dataset lengths between 5000 and 50000. For representation purposes the datasets will be referred as to dSn, where S determines the number of species under analysis and N the dataset length. For example, the dataset d50_5000, is used to analyse the DNA sequences of 500 species, with datasets of size 5000.

### 2.7 Profiling of the Original MrBayes

The MrBayes presented here is the basis for every speedup calculation done throughout this entire thesis. As stated before, this is the 3.2.1 version of the program and it is always run in `DEBUG_NOSHORTCUTS` mode. The original execution time can be seen in Figure 2.15. “Down/Root”, “Scaler”, “Likelihood”, “RemoveNode”, “CopySite” and “ResetSite” all refer to their respective functions. “Serial Part” refers to the serial part of the program. Even though “RemoveNode”, “CopySite” and “ResetSite” appear in the label, their total execution time was too small to appear in the actual graphic. It is also possible to see the same graphic in percentage form in Figure 2.16.

---

10 This means that the results from this Section are the $t_{Original}$ of equation 2.1.
2.7 Profiling of the Original MrBayes

Figure 2.15: The execution time of the original MrBayes with no shortcuts in serial mode. This was run by the CPU of the aforementioned testing framework.

Figure 2.16: The percentage of execution of the original serial MrBayes with no shortcuts activated. The percentage of time spent in functions Likelihood, RemoveNode, CopySite and ResetSite is 0 due to the fact that these functions are extremely fast and their time cannot be counted. This was run by the CPU of the aforementioned testing framework.
2.8 Summary

In this Chapter an overview of the various technological aspects needed to comprehend this thesis as well as some of the most recent work in this area was presented. The technological aspects viewed were:

- The GPU and its features for parallelism, including two of the most recent GPU architectures.
- The most commonly used APIs for parallel computing and the advantages and disadvantages of each one.
- The overall structure and characterization of the MrBayes program.
- State of the art parallel implementations of MrBayes.
- The framework and the input files used for testing.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 The Program</td>
<td>32</td>
</tr>
<tr>
<td>3.2 Parallelization Strategy</td>
<td>34</td>
</tr>
<tr>
<td>3.3 Parallelization of the most time consuming MrBayes functions: CondLikeDown and CondLikeRoot</td>
<td>35</td>
</tr>
<tr>
<td>3.4 Optimization considering memory accesses on GPU</td>
<td>40</td>
</tr>
<tr>
<td>3.5 Comparison with Parallel Phylogenetic Likelihood Function</td>
<td>46</td>
</tr>
<tr>
<td>3.6 Summary</td>
<td>47</td>
</tr>
</tbody>
</table>

3. From OpenACC to CUDA: a performance comparison
3. From OpenACC to CUDA: a performance comparison

Figure 3.1: MrBayes’ functions profiled with Kcacheigrind.

In order to select the best parallelization strategy of MrBayes and obtain an OpenACC implementation, this chapter starts by presenting the application structure. Following an approach similar to the one used in [7], a strategy was developed and implemented using OpenACC. This also allows to compare the performance of the computing system when using CUDA and OpenACC, as will be shown latter in this chapter. Further optimizations will then be described in Chapter 4.

3.1 The Program

MrBayes is a program for Bayesian inference whose latest version is MrBayes 3.2.1 released on February 2012 and used in this thesis. MrBayes can be used in various ways to analyse a set of different information. This thesis however focuses on analysing DNA in a 4 by 4 nucleotide substitution method. In addition to this, the program is set to run in DEBUG,NOSHORTCUTS mode which disables all computing shortcuts for processing phylogenetic trees. This allows to guarantee that the obtained acceleration is not due to avoiding calculations.

In a preliminary approach to the source code, it is easily noticeable that it is quite large. It is composed of 28 source files of which the most important file is over 45000 lines of code. Thus reading the code would prove to be a rather inefficient strategy to get familiar with the program and to be able to discern where the parallelization opportunities are.

Naturally, the correct way to tackle this problem is to profile the program to disclose the most time consuming tasks. Using kcachegrind together with valgrind produced a very informative profiling output. This output can be seen in Figure 3.1 and Figure 3.2.
3.1 The Program

LogLike is the first function called more than once during the program, has seen in Figure 3.1. This function has a surrounding for loop and it is one of the primary functions responsible for the computation of the MCMC algorithm. The LogLike function then calls two other very important functions: CondLikeDownNUC4 and the CondLikeRootNUC4. The relevance of these functions is due to the fact that they are some of the most called functions during the program, as seen in Figure 3.1 and they’re the most time consuming functions. In fact, CondLikeDownNUC4 is the most time consuming function of the entire program. However, for each tree node (iteration), either the CondLikeDownNUC4 or the CondLikeRootNUC4 function is called and never both. This can be seen in line 15056 of the mcmc.c source code file and in Figure 3.3. These functions are also somewhat similar and so it wouldn’t make sense to parallelize one without the other.

The CondLikeDownNUC4 function can be found on line 2975 of the mcmc.c file and the CondLikeRootNUC4 function can be found on line 4905 of the same file. Reading their code reveals that they are very similar functions with minor differences between them. It makes no sense to parallelize the CondLikeDownNUC4 function without the CondLikeRootNUC4 func-

1The tree referred is the phylogenetic tree which is represented as a tree (an acyclic graph) in the program.
tion because the percentage of times the CondLikeRoot\_NUC4 function is chosen over the CondLikeDown\_NUC4 function varies from input file to input file.

From every stated reason it is possible to conclude that the best functions to parallelize are the CondLikeDown\_NUC4 function and the CondLikeRoot\_NUC4 function. A short summary of these reasons follows:

- The CondLikeDown\_NUC4 and the CondLikeRoot\_NUC4 functions consume about 50% of the execution time (as seen in Figure 2.16) when running the MrBayes program. This means that there is no other functions where there can be a greater overall time gain than these.

- For each node of the tree either the CondLikeDown\_NUC4 function or the CondLikeRoot\_NUC4 function is called and never both. This means that the most time consuming block of code is either calling CondLikeDown\_NUC4 or CondLikeRoot\_NUC4.

- CondLikeDown\_NUC4 and CondLikeRoot\_NUC4 are very similar functions and the same parallelization strategy will work on both of them.

There are some challenges associated with the parallelization of both CondLikeDown\_NUC4 and CondLikeRoot\_NUC4 functions. Both functions are indeed very similar. The most relevant part of these functions to parallelize can be seen in Figure 3.4 and in Figure 3.5. This is the most relevant part because these for loops are the most computational heavy code in these functions. Furthermore this is also the most parallelizable part, since the rest is mostly serial code.

It is worth noting that the CondLikeDown\_NUC4 and the CondLikeRoot\_NUC4 functions are also the same functions that were parallelized in [7]. By doing this, this thesis follows the same strategy as the aforementioned state of the art which ensures that there is a fair comparison between these two frameworks later in this Chapter.

### 3.2 Parallelization Strategy

The parallelization strategy adopted for this first part of this thesis will be similar to the one used in Pratas et. al. [7] which is part of the state of the art. The required memory for each iteration is transferred only when it is needed. This means that each iteration begins by transferring its required data from the CPU to the GPU. Then, the most time consuming functions are executed on the GPU. Finally the results are transferred back to the CPU.

In what regards workload balancing, the strategy adopted in Pratas et. al. [7] was to have each thread calculating a position of the results' vector. This ensures that each thread's data is independent and that there is no synchronization needed among threads. Another workload optimization performed was to assign 4 threads to each block of data within the same array. Since
3.3 Parallelization of the most time consuming MrBayes functions: CondLikeDown and CondLikeRoot

Like most pipelining optimization problems, one of the best strategies to obtain a good overall acceleration in a program is to find the bottleneck and accelerate it. When a particular part of the program no longer is the bottleneck then one should move to the part of the program that has become the bottleneck. This strategy can be iterated until a satisfactory result is achieved.

3.3.1 Implementation

Since CPU and GPU have physically separated memory the first concern when trying to parallelize code using GPUs should be to transfer the necessary data to the GPU so that it can perform the necessary computations. Logically the first step is to create a data region for each function surrounding the outer for loops. To do this it is necessary to use the copyin and the copyout directives to transfer the data to the GPU and transfer the results back to CPU. It is also necessary to find the explicit length of each array and inform the compiler that a certain variable contains the length of said memory array. This is a more complex data region than the one shown in the example of Figure 2.8. That simple example was dealing with static memory which means that the compiler can easily determine the length of each array and allocate it in the GPU. This is not the case with MrBayes. MrBayes uses dynamic memory allocated to meet the needs of each input file. This means that the data length is only known during runtime and the compiler is unable to know its length beforehand and thus the aforementioned variable is needed.
3. From OpenACC to CUDA: a performance comparison

Once the data region is set up, the next step needed in order to run the program on the GPU is to tell the compiler how the parallelization is done. Since there are nested loops in both functions, the correct pragma to use is kernels. There are, however, further modifications needed: the loop itself must be modified and the indexes of the arrays must be calculated explicitly by the programmer. This is necessary because, much like the data region, this parallelization is not as simple as it was with the example of Figure 2.8. In this simple example the for loops are all very straightforward and easy for the compiler to understand. And once again this is not the case with the for loops of both these functions. Not only are they more elaborate but also use pointer arithmetic. The pointer arithmetic issue is critical because not only this is not a good practice with the GPU but also it is completely impossible for the compiler to comprehend and parallelize pointer arithmetic. Logically follows that pointer arithmetic cannot appear anywhere in the code that needs to be parallelized with OpenACC which in turn means that the absolute index for each variable must be calculated manually and inserted in the code as already mentioned.

The modifications described so far were not enough to achieve a decent parallelization. An additional modification to the for loops was made: an extra inner for loop was added to optimize the parallelization. This is necessary because the compiler still had some problems parallelizing these functions since the parallelization it was doing was not very effective. The reason for this is that the compiler was trying to distribute each loop iteration through the available CUDA cores and the inner loop’s work could be further split into smaller work units for a better workload balancing. With the extra for loop that was added, the compiler was able to understand that it was possible...
3.3 Parallelization of the most time consuming MrBayes functions: CondLikeDown and CondLikeRoot

```c
numGammaCats = m->numGammaCats;
numChars = m->numChars;
tiPL = pL;
tiPR = pR;
#pragma acc data pcopyin(tiPL[16*numGammaCats], cLL[4*numChars*numGammaCats], tiPR[16*numGammaCats],
clR[4*numChars*numGammaCats]), copyout(clP[4*numChars*numGammaCats])
#pragma acc kernels loop independent gang(numGammaCats)
for (k=0; k<numGammaCats; k++)
{
    #pragma acc loop independent gang(numChars/numGammaCats) vector(NTHREADS)
    for (c=0; c<numChars; c++)
    {
        #pragma acc loop independent vector(4)
        for (i=0; i < 4; i++)
        {
            register int indice = k*numChars*4+c*4;
            register int indice2 = k*16+i*4;
            c1P[indice+i] = (tiPL[indice2+AA]*cLL[indice+A] + tiPL[indice2+AC]*cLL[indice+C] +
                tiPL[indice2+AG]*cLL[indice+G] + tiPL[indice2+AT]*cLL[indice+T])*(tiPR[indice2+AA]*cLR[indice+A] +
                tiPR[indice2+AC]*cLR[indice+C] + tiPR[indice2+AG]*cLR[indice+G] + tiPR[indice2+AT]*cLR[indice+T]);
        }
    }
}
```

Figure 3.6: The first modification done to the most relevant part of the CondLikeDown_NUC4 function.

...to distribute the inner loop's work as well, as seen in Figure 3.6 and 3.7.

CondLikeDown_NUC4 and CondLikeRoot_NUC4 with every single modification described can be seen in Figure 3.6 and in Figure 3.7, respectively. Both of these modified functions can be found on a separated library created to prevent the modifications to damage the original MrBayes code. This new library contains the mbopenacc.c file and the mbopenacc.h file. The functions are now called CondLikeDown_NUC4_OpenACC and CondLikeRoot_NUC4_OpenACC, respectively.

NTHREADS refers to a define written in the beginning of the file in order to facilitate the testing with different number of GPU threads. The variables numGammaCats and numChars were added to solve an aforementioned limitation of the OpenACC compiler. The compiler, as mentioned earlier, cannot read pointer arithmetic on pragmas and thus it couldn’t understand something like #pragma acc data pcopyin(tiPL[16 * m – >numGammaCats]) because this pragma attempts to read the memory position in m – >numGammaCats.

There is yet a “hidden” bottleneck that is present in most acceleration problems using GPUs and was not addressed so far: memory transfers. The PCI-E communication interface between CPU and GPU is particularly slow for data transfers. There are some precautions to take with CPU to GPU (and vice-versa) memory transfers [18]:

- Minimize the amount of data transferred between host and device when possible, even if that means running kernels on the GPU that get little or no speed-up compared to running them on the host CPU.
3. From OpenACC to CUDA: a performance comparison

```c
numGammaCats = m->numGammaCats;
numChars = m->numChars;
tiPL = pL;
tiPR = pR;
tiPA = pA;
#pragma acc data pcopyin(tiPL[16*numGammaCats], clL[4*numChars*numGammaCats],
tiPR[16*numGammaCats], clR[4*numChars*numGammaCats],
tiPA[16*numGammaCats], clA[4*numChars*numGammaCats]),
copyout(clP[4*numChars*numGammaCats])

#pragma acc kernels loop independent gang(numGammaCats)
for (k=0; k<numGammaCats; k++)
{
    #pragma acc loop independent gang(numChars/numGammaCats) vector(NTHREADS)
    for (c=0; c<numChars; c++)
    {
        #pragma acc loop independent vector(4)
        for (i=0; i<4; i++) {
            register int indice = k*numChars*4+c*4;
            register int indice2 = k*16+i*4;
            clP[indice+i] = (tiPL[indice2+AA]*clL[indice+A] +
                            tiPL[indice2+AC]*clL[indice+C] +
                            tiPL[indice2+AG]*clL[indice+G] +
                            tiPL[indice2+AT]*clL[indice+T]) *
                            (tiPR[indice2+AA]*clR[indice+A] +
                            tiPR[indice2+AC]*clR[indice+C] +
                            tiPR[indice2+AG]*clR[indice+G] +
                            tiPR[indice2+AT]*clR[indice+T]) *
                            (tiPA[indice2+AA]*clA[indice+A] +
                            tiPA[indice2+AC]*clA[indice+C] +
                            tiPA[indice2+AG]*clA[indice+G] +
                            tiPA[indice2+AT]*clA[indice+T]);
        }
    }
}
```

Figure 3.7: The first modification done to the most relevant part of the CondLikeRoot.NUC4 function.

- Higher bandwidth is possible between the host and the device when using page-locked (or “pinned”) memory.
- Batching many small transfers into one larger transfer performs much better because it eliminates most of the per-transfer overhead.
- Data transfers between the host and device can sometimes be overlapped with kernel execution and other data transfers.

The second point is more CUDA related since there is not the same level of control of memory allocation with OpenACC. When memory transfers are optimized in this manner there is a high probability of having a great performance improvement.

3.3.2 Results

This implementation is a good starting point but it is not without its problems. As it is possible to see in Figure 3.1, the CondLikeDown.NUC4 is called over 32000 times on the small input file MrBayes was analysing. This means that the GPU memory associated with the CondLikeDown.NUC4 function is allocated and transferred to and from the GPU over 32000 times. This represents a huge overhead to the GPU usage and undermines any overall speedup the program might had. This is confirmed by Equation 2.1 since $t_{\text{Parallel}} = t_{\text{SerialPart}} + t_{\text{ParallelPart}} + t_{\text{CommunicationOverhead}}$. 

3.3 Parallelization of the most time consuming MrBayes functions: CondLikeDown and CondLikeRoot

Figure 3.8: A graphic showing the time taken to execute several inputs for the first modification made to MrBayes.

On Figure 3.8 it is possible to see the time taken by the program to analyse several inputs. This graphic takes into account the time consumed by the serial part of the program and the parallel part of the program. The “Serial Part” refers to the time consumed by the serial part of the program and the “Down/Root” refers to the time consumed by the functions CondLikeDown_NUC4_OpenACC and CondLikeRoot_NUC4_OpenACC. The parallel part is still taking a lot of time because of the aforementioned problems and because the “Down/Root” part takes into account the time used for memory transfers.

With the data of Figure 3.8 it is possible to obtain the speedup of both the entire program and the CondLikeDown_NUC4_OpenACC and CondLikeRoot_NUC4_OpenACC functions. This can be seen in Figure 3.9. As expected the speedup of the whole program rarely exceeds 1 and the speedup for the CondLikeDown_NUC4_OpenACC and CondLikeRoot_NUC4_OpenACC functions is not much better due to the excessive memory allocation and the fact that the memory transfer are taken into account.

A small explanation about the input files, the testing machine and the basis for calculating the speedup can be found in Chapter 2.
3. From OpenACC to CUDA: a performance comparison

Figure 3.9: The speedup obtained with the first modification made to MrBayes program.

3.4 Optimization considering memory accesses on GPU

The first modification made to MrBayes was a good preliminary attempt to parallelize MrBayes with OpenACC. Unfortunately some shortcomings of the implementation itself meant that the performance was below its full potential. In this Section some of the previous version’s problems are tackled and the parallelized functions’ performance is greatly increased.

3.4.1 Implementation

As already mentioned before, the previous implementation had some problems. These problems can be summarised in the following:

- Excessive memory allocations on the GPU side.
- Lack of optimization of the kernel itself.
- Excessive memory transfers between CPU and GPU.

The first two points of the list have greater impact on the performance of the parallelized functions known as CondLikeDown_NUC4_OpenACC and CondLikeRoot_NUC4_OpenACC and the last point impacts mostly on the overall performance. Since reducing the amount of data transferred between the CPU and GPU is a greater problem that affects a large portion of the MrBayes program, this Section will only focus in tackling the first two points.
In order to prevent the excessive memory allocation on the GPU, one must only allocate the memory once and instruct the GPU to reuse that same space. There are two main conceptual ways this could be done:

- Allocate every single array needed for each function and reuse them on each function call.

- Understand that these arrays are part of a larger matrix of memory and allocate the entire memory.

The simpler solution is to simply allocate the memory arrays needed for each function call and reuse them. However, even though it is simpler, there are two major reasons for not doing this:

- If, later on, the memory transfer problem is tackled, the entire memory matrix is going to be needed on the GPU.

- OpenACC does not deal well with pointers that point to only part of a larger array of memory.

Several tests were made regarding the second reason and the only way to avoid this would be to allocate smaller arrays on the CPU side and copy the data from the larger matrix of data to these smaller arrays on the CPU side and finally copy the data of these smaller arrays from the CPU to the GPU. Naturally this would be a rather inefficient solution.

On the other hand, allocating the entire matrix of data is not without its own problems. The biggest problem that could be faced with this approach would be the GPU not having enough memory to accommodate the required data. Luckily this is not a problem since the maximum size these matrices can have for the inputs available represent a total of 236825344 floats which represent about 900 MiB. The available memory on the GPU used in this thesis is 1536 MiB\(^2\). However, should there be a larger input file that would exceed the available memory then a different memory management strategy would be needed since the matrices would not be able to fit in. This would require to transfer part of the memory to the GPU and rotate the part of the memory that is in the GPU according to the program needs. These matrices had to be found, of course, and they are the \textit{tiProbs} matrix and the \textit{condLikes} matrix.

In order to achieve only one memory allocation it is necessary to create a data region that surrounds the entire loop that is responsible for analysing an input file. As it is possible to perceive from Figure [3.1] and Figure [3.2] \textit{RunChain} is the last function called only once. \textit{RunChain} then calls \textit{LogLike} multiple times. Since the objective is to allocate the GPU memory only once, one of the best places to place the data region is in the \textit{DoMcmc} function surrounding the call of the \textit{RunChain} function. An attempt to place the data region within the \textit{RunChain} function surrounding the for loop was made but the compiler had some problems in perceiving this so the data region ended up surrounding the entire \textit{RunChain} function as stated before. \textit{RunChain}, \textit{LogLike} and \textit{DoMcmc} can all be found on the mcmc.c file.

\[^2\]More information about the machine and GPU used for testing can be found in the second Chapter.
With the aforementioned strategy, the OpenACC compiler was able to perceive the data region without any problems. However, MrBayes’ memory allocation strategy is not optimized for GPU usage thus this strategy had to be modified so that the matrices were guaranteed to be in a contiguous array of memory. The reason for this modification is that the `tiProbs` and `condLikes` variables are indeed dynamic 2D data matrices. This means that they’re dynamically allocated depending on the memory needs of each input file. The most common way of allocating a 2D dynamic matrix in memory is depicted in Figure 3.10. The `Base` is a double pointer of the data type needed. An array of pointers is then allocated with one pointer per matrix line. Finally an array of the data type needed is allocated with the length equal to the number of columns. This last array is allocated multiple times (one per matrix line). This is how MrBayes allocates its matrices. The major problem with this approach is that the data arrays are not guaranteed to be contiguous, which is a problem for OpenACC, since it has issues detecting non-contiguous data matrices. Thus the aforementioned modification is to use a much better memory allocation strategy that guarantees the memory is contiguous while allowing the user to access the data like a regular dynamic matrix. This strategy is depicted in Figure 3.11. The base and the first array of pointers is similar to the previous method but there’s only one array of data with the length of the entire matrix (`number of columns × number of lines`). The pointer array points to the beginning of each line in the larger array. This strategy allows OpenACC to recognise the matrix and the rest of the program continues to access the data in the same way as it did before. Two new variables were created: `tiProbs_gpu` and `condLikes_gpu`. These variables simply point to the beginning of the data array of `tiProbs` and `condLikes`, respectively. In this manner OpenACC simply perceives `tiProbs` and `condLikes` as two very large and contiguous arrays.
3.4 Optimization considering memory accesses on GPU

```c
#ifndef OACC_ENABLED
    numGammaCats = (&modelSettings[0])->numGammaCats; numChars = (&modelSettings[0])->numChars;
    m = &modelSettings[0];
tiProbs_gpu = &m->tiProbs[0][0];
    condLikes_gpu = &m->condLikes[0][0];
    tiProbs_x = m->numTiProbs;
    tiProbs_y = m->tiProbLength;
    condLikes_x = m->numCondLikes;
    condLikes_y = m->condLikeLength;
#endif
```

```c
#pragma acc data create(tiProbs_gpu[tiProbs_x*tiProbs_y], condLikes_gpu[condLikes_x*condLikes_y])
#endif
/* Run the Markov chain. */
rc = RunChain(&seed);
```  

Regarding the kernel itself, the experimental results showed that the best value for \textit{NTHREADS} was 16, thus optimizing the workload distribution. Another optimization done, was to set a fixed length to the \textit{tiProbs\_gpu} and \textit{condLikes\_gpu} arrays on the GPU side. It was not possible to obtain a reason for this, but the performance of the kernels was much better when the GPU memory had a fixed size. The fact that the GPU is accessing the entire matrix instead of just the arrays it needed means the every single index had to be recalculated to take the entire matrix into account.

The modifications made to the \textit{DoMcmc} function can be seen in Figure 3.12. It worth noting that the \texttt{#if defined (OACC\_ENABLED)} was added to easily turn off OpenACC usage and its data region. The modifications made to the \textit{CondLike\_Down\_NUC4\_OpenACC} function can be seen in Figure 3.13. Since the modifications to the \textit{CondLike\_Root\_NUC4\_OpenACC} function are very similar to those of the \textit{CondLike\_Down\_NUC4\_OpenACC} function, just the code for the latter
Figure 3.13: The second modification done to the most relevant part of the CondLikeDown_NUC4 function.
3.4 Optimization considering memory accesses on GPU

3.4.2 Results

The improvements made to the code in this section had a greater impact in the functions \texttt{CondLikeDownNUCA\_OpenACC} and \texttt{CondLikeRootNUCA\_OpenACC} as it was expected. The overall performance of the program remained more or less the same since the memory transfers between CPU and GPU are a big bottleneck that was not addressed. This can be seen in Figure 3.14 where “Serial Part” refers to the serial part of the program, “Total memory transfers” refers to the time consumed by the memory transfers associated with both functions and “Total Down/Root” refers to the time consumed by the \texttt{CondLikeDownNUCA\_OpenACC} and \texttt{CondLikeRootNUCA\_OpenACC} functions. However there was a performance improvement of both the \texttt{CondLikeDownNUCA\_OpenACC} function and the \texttt{CondLikeRootNUCA\_OpenACC} function. The maximum speedup achieved for both functions can be seen in Figure 3.15. In this Figure both functions’ time was summed and compared to the summed time of the functions of the original MrBayes. The memory transfers were not taken into account in this Figure. As shown in this Figure there is a maximum speedup of about 9 for both functions.

A small explanation about the input files, the testing machine and the basis for calculating the speedup can be found in Chapter 2.
3. From OpenACC to CUDA: a performance comparison

Figure 3.15: The speedup of both CondLikeDown_NUC4_OpenACC and CondLikeRoot_NUC4_OpenACC functions when compared to the original MrBayes.

3.5 Comparison with Parallel Phylogenetic Likelihood Function

With this level of development, this thesis is now ready to be compared with a very similar CUDA implementation. The parallel implementation of MrBayes discussed in Section 2.5.1 is indeed very resemblant of this implementation. Both implementations focused on parallelizing the CondLikeDown_NUC4 and CondLikeRoot_NUC4 functions.

Since the implementation of Section 2.5.1 restricted its parallelization to these two functions, it is a good choice to make the comparison between CUDA and OpenACC. The other implementations of Section 2.5 usually modify much more than just these two functions.

The comparison chart between CUDA and OpenACC can be seen in Figure 3.16. This chart shows both the speedup of the functions as well as the speedup of the whole program for both CUDA and OpenACC. “Full OpenACC Speedup” (dark blue) refers to the speedup of the whole program using OpenACC. “Full CUDA Speedup” (purple) refers to the speedup of the whole program using CUDA. “Down/Root OpenACC Speedup” (green) refers to the speedup of the CondLikeDown_NUC4 and CondLikeRoot_NUC4 functions parallelized with OpenACC. “Down/Root CUDA Speedup” (light blue) refers to the speedup of the CondLikeDown_NUC4 and CondLikeRoot_NUC4 functions parallelized with CUDA.

As it is possible to see through the functions’ speedup, OpenACC has nearly half the performance of CUDA and in some cases it is even worse. The reason behind this involves some fine
3.6 Summary

Figure 3.16: Comparison between CUDA and OpenACC.

Tuning that it is only possible to do when the kernel is coded with CUDA, but mostly it is due to the differences between the two frameworks and the automatism that OpenACC introduces when generating the kernels. Thus some loss of performance was to be expected.

3.6 Summary

This Chapter started by detailing more the technological aspect of OpenACC as well as the main functionalities it had including its pragmas. In order to simplify the reader’s comprehension of OpenACC a simple example was given in Figure 2.8 along with a detailed explanation of it.

After discussing the technological aspect of OpenACC, it was presented a programmer’s overview on MrBayes and its configuration parameters. Furthermore a characterization of the program was performed as presented in Figure 3.1 and in Figure 3.2. This allowed the identification of the most time consuming function (*CondLikeDown*, NUC4) and the bottleneck on MrBayes.

Following, it was discussed the parallelization strategy, including general guidelines for acceleration problems as well as some typical problems when accelerating applications on GPUs.

After this introduction to the technological aspects regarding this thesis, the first parallelization attempt was presented. This first attempt focused only on the bottleneck identified. This provided some acceleration to the functions belonging to the bottleneck but it had a very poor overall performance due to some problems that had been found (excessive memory allocation, lack of kernel optimization and excessive memory transfers). There were also some guidelines regarding OpenACC, and its limitations and how to surpass them.
3. From OpenACC to CUDA: a performance comparison

All of the work described, led to a very similar implementation of parallel MrBayes to the one described in Section 2.5.1. The main difference is that this thesis’ implementation is done with OpenACC instead of CUDA. This means that a fair performance comparison between CUDA and OpenACC was done. This allowed to assess the performance difference between OpenACC and CUDA.
# Bottlenecks and optimizations: an overall performance improvement

## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 The Bottleneck</td>
<td>50</td>
</tr>
<tr>
<td>4.2 Parallelization of the second most time consuming function: CondLikeScaler</td>
<td>51</td>
</tr>
<tr>
<td>4.3 Memory Output Reduction</td>
<td>56</td>
</tr>
<tr>
<td>4.4 Memory Transfers Optimization</td>
<td>61</td>
</tr>
<tr>
<td>4.5 Summary</td>
<td>66</td>
</tr>
</tbody>
</table>
4. Bottlenecks and optimizations: an overall performance improvement

Table 4.1: Number of floats transferred between CPU and GPU per iteration.

<table>
<thead>
<tr>
<th></th>
<th>Smallest Input</th>
<th>Largest Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>tiProbs CondLikeDown</td>
<td>512 bytes per iteration</td>
<td>512 bytes per iteration</td>
</tr>
<tr>
<td>condLikes CondLikeDown</td>
<td>192000 bytes per iteration</td>
<td>9600000 bytes per iteration</td>
</tr>
<tr>
<td>tiProbs CondLikeRoot</td>
<td>768 bytes per iteration</td>
<td>768 bytes per iteration</td>
</tr>
<tr>
<td>condLikes CondLikeRoot</td>
<td>256000 floats per iteration</td>
<td>12800000 bytes per iteration</td>
</tr>
</tbody>
</table>

In the previous Chapter a decent parallelization of the bottleneck functions was achieved with a maximum speedup of about 9. In spite of this, the overall performance of the program was somewhat poor. Thus, it is possible to obtain better results regarding the program’s overall performance and that is exactly what this Chapter focuses.

4.1 The Bottleneck

The true bottleneck preventing this thesis’ implementation from having good overall performance had already become evident in the previous Chapter: memory transfers between CPU and GPU (as seen in Figure 3.14). Overcoming this bottleneck is not, however, an easy task. There are two ways of accelerating memory transfers between CPU and GPU: eliminate the memory transfers all together or to overlap the memory transfers with computation.

A closer analysis to the code reveals that the condLikes matrix only needs some initial values and every other result is calculated during the data processing. The same applies to the tiProbs matrix which means that it is not necessary to overlap memory transfers with computation since it is possible to remove them. Removing memory transfers cannot be done just by simply eliminating the update clauses in the code. The GPU needs that data to perform its calculations. However, the vast majority of memory transfers is related to the condLikes matrix. It is possible to see how many bytes are transferred for each matrix in each function in table 4.1. The number of floats shown here are per iteration of the main loop, i.e. per function call.

In order to remove the data transfer related to the condLikes it is necessary to parallelize every single function that processes data in this matrix so that it will run on the GPU. Examining the code reveals that only the CondLikeScaler_NUC4 modifies the condLikes matrix apart from the two functions already parallelized. Thus CondLikeScaler_NUC4 is the next function that is necessary to parallelize. However, there is yet another function that simply reads from the condLikes matrix without modifying it. This function has an output of just a single float which means that instead of sending an entire array of data to the CPU each time a function modifies the condLikes matrix, it is possible to send just a single float if this function is parallelized. This function is called Likelihood_NUC4. Doing this also addresses the yet large serial portion of the program as seen in Figure 3.14.
4.2 Parallelization of the second most time consuming function: CondLikeScaler

The parallelization of the function CondLikeScaler, which original code is shown in Figure 4.1, presented new challenges that were not found in the previous functions.

This function requires for the for loop to be modified substantially. The the two inner loops and the work done by the outer loop are fairly independent, this large loop is thus be subdivided into 3 smaller loops:

- A first double for loop containing the first inner loop.
- A second double for loop containing the second inner loop.
- A third for loop containing the work done by the outer loop.

The main reason behind this division is that the OpenACC compiler finds it difficult to understand and parallelize complex for loops. This loop in particular is rather complex since it has two different inner loops inside the same outer loop and the outer loop itself is doing some calculations. This level of complexity caused the compiler to be unable to discern whether the loops are data independent or not and thus it fails any attempt of parallelization.

This simplifies the loops for the compiler, allowing it to be able to fully understand and successfully parallelize the loops. However, to perfect this simplification for the compiler, several intermediate transformations require the use of extra memory: the intermediate results need to be stored in an array so that the next loop can continue where the previous one left off.

As stated before, the main objective is to reduce the memory transfers associated with the condLikes matrix. Due to the fact that only the first two smaller loops need and transform data from the condLikes matrix it is a good strategy to only parallelize the first two smaller loops for this specific objective.

The parallelization strategy for these two smaller loops is much similar to the one used on Section 3.3. The first thing needed is a data region surrounding the loops detailing how the arrays are transferred/allocated in the GPU. For this particular case it was necessary to inform the compiler that the condLikes matrix was already on the GPU memory and that it was necessary to allocate auxiliary variables to accommodate the intermediate results. Once this was done, it was simply a matter of using a kernels pragma to attempt to parallelize the loops.

Still, these for loops required one last modification to run smoothly. The inner loop of the first smaller loop was manually unrolled to prevent a compiler misunderstanding of variable behaviour to ruin the results. Further explanation of this compiler bug follows.

As soon as the the first attempt of this parallelization was done, the results from MrBayes began to be all wrong. In spite of the simplification done the loops, the compiler was still misunderstanding part of the code. Since, to date, it is not possible to access data from GPU in a
4. Bottlenecks and optimizations: an overall performance improvement

/* This function is part of a mechanism MrBayes implements to prevent underflow derived from very low probabilities. It begins by finding the highest probability in a set (the scaler) and then it divides the results by that scaler. Since the probabilities (and the scaler) are values between 0 and 1, this division will always yield a value equal or higher than the original. Finally the scaler is stored so that the original value can be retrieved later on. */

/* rescale values */
for (c=0; c<m->numChars; c++)
{
    scaler = 0.0;
    for (k=0; k<m->numGammaCats; k++)
    {
        if (clP[k][A] > scaler)
            scaler = clP[k][A];
        if (clP[k][C] > scaler)
            scaler = clP[k][C];
        if (clP[k][G] > scaler)
            scaler = clP[k][G];
        if (clP[k][T] > scaler)
            scaler = clP[k][T];
    }
    #if defined (FAST_LOG)
    frexp (scaler, &index);
    index = 1-index;
    scaler = scalerValue[index];
    #endif
    for (k=0; k<m->numGammaCats; k++)
    {
        clP[k][A] /= scaler;
        clP[k][C] /= scaler;
        clP[k][G] /= scaler;
        clP[k][T] /= scaler;
    }
    #if defined (FAST_LOG)
    scP[c] = logValue[index];  /* store node scaler */
    lnScaler[c] += scP[c];  /* add into tree scaler */
    #endif
    #else
    scP[c] = (CLFlt) log(scaler);  /* store node scaler */
    lnScaler[c] += scP[c];  /* add into tree scaler */
    #endif
}

Figure 4.1: The most important part of original CondLikeScaler_NUC4.
4.2 Parallelization of the second most time consuming function: CondLikeScaler

direct manner, the only way to understand what was going wrong was to transfer the memory to the CPU and analyse it there. This strategy confirmed that the problem was indeed from the kernel created for the CondLikeScaler_NUC4 function but it did not pinpoint exactly what was the problem.

The problem resided within the first loop. The first loop starts by setting a variable named scaler to zero and then it finds the highest possible value in another array to attribute to that same variable. The compiler was having problems understanding how this variable was supposed to behave among GPU threads. The objective was for the compiler to understand that this variable should be private for each thread and that simply selecting the highest value at the end should suffice. Since this was not working, then it was necessary to force a single thread to execute the whole loop to guarantee that the variable was not shared with any other thread. A good way of forcing the single thread processing is to manually unroll the inner loop. Since the inner loop only executed for times per outer loop cycle, this was not complicated. This solution may not be the best or the most efficient, but it made the compiler parallelize this function correctly.

Another function, containing all of the aforementioned modifications, was added to the library this thesis is developing: CondLikeScaler_NUC4_OpenACC. The most important part of this function can be found in Figures 4.2 and 4.3.

4.2.1 Results

These modifications only targeted the CondLikeScaler_NUC4 function. Even though they are not enough to remove the memory transfers, they represent an important step towards that goal.

Since there was no modifications made to the CondLikeDown_NUC4_OpenACC function and CondLikeRoot_NUC4_OpenACC function, the results regarding them should not have changed. It is however expectable to observe some speedup relative to the CondLikeScaler_NUC4 function. The execution time for this implementation can be seen in Figure 4.4. The “Serial Part” refers to the serial portion of the program, the “Scaler Memory Transfers” refers to the time consumed by the memory transfers in the CondLikeDown_NUC4_OpenACC function, the “Scaler Total Time” refers to the time consumed by the CondLikeDown_NUC4_OpenACC function without memory transfers, the “Down/Root Memory Transfers” refers to the time consumed by the memory transfers associated with the CondLikeDown_NUC4_OpenACC and CondLikeRoot_NUC4_OpenACC functions and the “Down/Root Total Time” refers to the time consumed by the same functions without memory transfers. In this Figure it is possible to observe that the parallelization of the CondLikeScaler_NUC4 scaler introduces non neglectable memory transfers of its own. Furthermore, since there are no memory transfers removed, the overall performance is still very poor.

The speedup obtained for the various parts of the program can be seen in Figure 4.5. The “Full Program” refers to the speedup of the entire program, the “Down/Root” refers to the speedup of
4. Bottlenecks and optimizations: an overall performance improvement

```c
#pragma acc data present(condLikes_gpu[condLikes_length]) create(aux_scaler) pcreate(scalers_gpu[scalers_length])
{
    /* rescale values */
    #pragma acc kernels loop independent gang(numChars) private (aux_scaler)
    for (c=0; c<numChars; c++)
    {
        aux_scaler = 0.0;
        register int indexA = clPtr_index * clPtr_length + 4*c + A;
        register int indexC = clPtr_index * clPtr_length + 4*c + C;
        register int indexG = clPtr_index * clPtr_length + 4*c + G;
        register int indexT = clPtr_index * clPtr_length + 4*c + T;
        if (condLikes_gpu[indexA] > aux_scaler)
            aux_scaler = condLikes_gpu[indexA];
        if (condLikes_gpu[indexC] > aux_scaler)
            aux_scaler = condLikes_gpu[indexC];
        if (condLikes_gpu[indexG] > aux_scaler)
            aux_scaler = condLikes_gpu[indexG];
        if (condLikes_gpu[indexT] > aux_scaler)
            aux_scaler = condLikes_gpu[indexT];
        indexA += clPtr_increment;
        indexC += clPtr_increment;
        indexG += clPtr_increment;
        indexT += clPtr_increment;
        if (condLikes_gpu[indexA] > aux_scaler)
            aux_scaler = condLikes_gpu[indexA];
        if (condLikes_gpu[indexC] > aux_scaler)
            aux_scaler = condLikes_gpu[indexC];
        if (condLikes_gpu[indexG] > aux_scaler)
            aux_scaler = condLikes_gpu[indexG];
        if (condLikes_gpu[indexT] > aux_scaler)
            aux_scaler = condLikes_gpu[indexT];
        indexA += clPtr_increment;
        indexC += clPtr_increment;
        indexG += clPtr_increment;
        indexT += clPtr_increment;
        if (condLikes_gpu[indexA] > aux_scaler)
            aux_scaler = condLikes_gpu[indexA];
        if (condLikes_gpu[indexC] > aux_scaler)
            aux_scaler = condLikes_gpu[indexC];
        if (condLikes_gpu[indexG] > aux_scaler)
            aux_scaler = condLikes_gpu[indexG];
        if (condLikes_gpu[indexT] > aux_scaler)
            aux_scaler = condLikes_gpu[indexT];
        indexA += clPtr_increment;
        indexC += clPtr_increment;
        indexG += clPtr_increment;
        indexT += clPtr_increment;
        if (condLikes_gpu[indexA] > aux_scaler)
            aux_scaler = condLikes_gpu[indexA];
        if (condLikes_gpu[indexC] > aux_scaler)
            aux_scaler = condLikes_gpu[indexC];
        if (condLikes_gpu[indexG] > aux_scaler)
            aux_scaler = condLikes_gpu[indexG];
        if (condLikes_gpu[indexT] > aux_scaler)
            aux_scaler = condLikes_gpu[indexT];
        scalers_gpu[c] = aux_scaler;
    }
#pragma acc update host(scalers_gpu[scalers_length])
```

Figure 4.2: The most important part of the first modification made to CondLikeScaler_NUC4.
4.2 Parallelization of the second most time consuming function: CondLikeScaler

```c
#pragma acc kernels loop independent gang(numChars)
for (c=0; c<numChars; c++)
{
    #pragma acc loop independent gang(1) vector(numGammaCats)
    for (k=0; k<numGammaCats; k++)
    {
        register int indexA = k * clPtr_increment + clPtr_index * clPtr_length + 4*c + A;
        register int indexC = k * clPtr_increment + clPtr_index * clPtr_length + 4*c + C;
        register int indexG = k * clPtr_increment + clPtr_index * clPtr_length + 4*c + G;
        register int indexT = k * clPtr_increment + clPtr_index * clPtr_length + 4*c + T;
        condLikes_gpu[indexA] /= scalers_gpu[c];
        condLikes_gpu[indexC] /= scalers_gpu[c];
        condLikes_gpu[indexG] /= scalers_gpu[c];
        condLikes_gpu[indexT] /= scalers_gpu[c];
    }
    #pragma acc update host(condLikes_gpu[clPtr_index * clPtr_length : numGammaCats * clPtr_increment])
}

/* Extra loop to do the remaining work done at the end of the previous loop*/
for (c=0; c<m->numChars; c++)
{
    scP[c] = (CLFlt) log(scalers_gpu[c]); /* store node scaler */
    lnScaler[c] += scP[c]; /* add into tree scaler */
}
```

Figure 4.3: The most important part of the first modification made to CondLikeScaler_NUC4.

Figure 4.4: The detailed execution time for the third modification made to MrBayes
4. Bottlenecks and optimizations: an overall performance improvement

Figure 4.5: The speedup for the various parts of the program for the third modification made to MrBayes.

the CondLikeDownNUC4OpenACC and CondLikeRootNUC4OpenACC functions without memory transfers, the “Scaler” refers to the speedup of the CondLikeScalerNUC4OpenACC function without memory transfers and the “Functions Speedup” refers to the speedup of the three functions together. In this Figure it is possible to observe that the CondLikeDownNUC4OpenACC and CondLikeRootNUC4OpenACC functions maintained more or less the same performance (as expected) and that the CondLikeScalerNUC4OpenACC function has a maximum speedup of about 7.

When compared to the previous implementation (section 3.4) the overall performance is about the same with exception to some of the larger inputs where this version has better results. These results can be seen in Figure 4.6.

A small explanation about the input files, the testing machine and the basis for calculating the speedup can be found in Chapter 2.

4.3 Memory Output Reduction

In order to truly avoid data transfers related to the condLikes matrix, it is necessary to parallelize the LikelihoodNUC4 function in addition to the CondLikeScalerNUC4 function, as previ-
4.3 Memory Output Reduction

Figure 4.6: Execution time comparison between the second modification and the third modification made to MrBayes.

Previously stated in Section 4.1.

The original code of the Likelihood\_NUC4 function can be seen in Figure 4.7. From this Figure it is possible to perceive that there are two main challenges when parallelizing this function:

- There is a problem with the like variable similar to the problem in the previous implementation (Section 4.2).

- There is need of a reduction clause to sum all of lnL.

The first point is tackled in a similar manner to the one used in the previous Section 4.2. The only workaround to the compiler bug is to manually unroll the loop. Once again the inner loop only executed 4 times per outer loop.

To implement the second point, it is necessary to do something counter intuitive: the lnL variable must only appear in the reduction clause and never appear in a copyout clause. Originally, in the data region created for this function, there was a copyout clause to the lnL variable in addition to the reduction clause in the kernels. This caused the lnL variable to always have the value zero on the CPU side after the kernel's execution. After some investigation and contacting the help forums from PGI\(^1\), the source of the problem was found. If a data clause (such as copyout) is used in conjunction with a reduction clause, the data clause will overwrite the result from the reduction clause.

\(^1\)PGI was the compiler used to compile OpenACC pragmas.
4. Bottlenecks and optimizations: an overall performance improvement

/* This function is responsible for calculating the final chain likelihood for each iteration. This final value is the probability used to decide whether or not to keep the chain. */
for (c=0; c<m->numChars; c++)
{
    like = 0.0;
    for (k=0; k<m->numGammaCats; k++)
    {
        clP[k] += 4;
    }
    like *= freq;
    /* check against LIKE_EPSILON (values close to zero are problematic) */
    if (like < LIKE_EPSILON)
    {
        MrBayesPrint ("%s WARNING: In LIKE_EPSILON - for division %d char %d has like = %1.30lf\n",
                      spacer, division+1, c+1, like);
        (*lnL)=MRBFLT_NEG_MAX;
        return ERROR;
    }
    else
    {
#if defined (FAST_LOG)
        f = frexp (like, &index);
        index = 1-index;
        (*lnL) += (lnScaler[c] + logValue[index]) * nSitesOfPat[c];
        for (k=0; k<(int)nSitesOfPat[c]; k++)
            likeAdjust *= f;
#else
        (*lnL) += (lnScaler[c] + log(like)) * nSitesOfPat[c];
#endif
    }
}

Figure 4.7: The most important part of the original Likelihood_NUC4 function.
4.3 Memory Output Reduction

```c
#pragma acc data present(condLikes_gpu[condLikes_length],
scalers_gpu[scalers_length*scalers_height], numSitesOfPat[numSitesOfPat_length])
copyin(bs[4], logFreq, clPtr_index, clPtr_length, clPtr_increment) create(like_gpu[4], like)
{
    #pragma acc kernels loop independent reduction(+ : lnL_gpu) private(like)
    for (c=0; c<numChars; c++)
    {
        register int indexA = indexAux1 + 4*c + A;
        register int indexC = indexAux1 + 4*c + C;
        register int indexG = indexAux1 + 4*c + G;
        register int indexT = indexAux1 + 4*c + T;


        indexA += clPtr_increment;
        indexC += clPtr_increment;
        indexG += clPtr_increment;
        indexT += clPtr_increment;

        indexA += clPtr_increment;
        indexC += clPtr_increment;
        indexG += clPtr_increment;
        indexT += clPtr_increment;

        indexA += clPtr_increment;
        indexC += clPtr_increment;
        indexG += clPtr_increment;
        indexT += clPtr_increment;

        indexA += clPtr_increment;
        indexC += clPtr_increment;
        indexG += clPtr_increment;
        indexT += clPtr_increment;

        indexA += clPtr_increment;
        indexC += clPtr_increment;
        indexG += clPtr_increment;
        indexT += clPtr_increment;
        lnL_gpu += (scalers_gpu[scalers_index*numChars + c] + log(like)*logFreq) * numSitesOfPat_gpu[numSitesOfPat_index + c];
    }
    *lnL = lnL_gpu;
}
```

Figure 4.8: The most important part of the modification made to the Likelihood_NUC4 function.

The clause and cause the loss of data that was experienced. All of the aforementioned modifications can be seen in Figure 4.8 which represents the Likelihood_NUC4_OpenACC function added to the library being created. An auxiliary variable named lnL_gpu was created to assist the GPU calculations.

After this implementation it was possible to remove every single memory transfer related to the condLikes matrix. However, as seen in Figure 4.4 the scaler function also introduced its own memory transfers which in turn introduced a non negligible overhead. It would be good to eliminate these memory transfers as well. Resorting once again to KCachegrind and inspecting the original MrBayes code reveals three more functions that use the same memory used in the CondLikeScaler_NUC4_OpenACC function:

- RemoveNodeScalers.

59
4. Bottlenecks and optimizations: an overall performance improvement

Figure 4.9: The execution time of the modified Likelihood_NUC4 function.

- CopySiteScalers.
- ResetSiteScalers.

Instead of removing the memory transfers associated with the condLikes matrix right now, Section 4.4 will focus on parallelizing these three remaining functions and removing practically all of the bottleneck data transfers between CPU and GPU.

4.3.1 Results

Since the memory transfer reduction was not yet made, the results in this Section will only reflect the modifications made to the Likelihood_NUC4 function. A comparison between the original Likelihood_NUC4 function and the modified version can be seen in Figure 4.9. The “Modified Likelihood” refers to the Likelihood_NUC4_OpenACC execution time and the “Original Likelihood” refers to the Likelihood_NUC4 execution time. The speedup of the Likelihood_NUC4_OpenACC function relative to the Likelihood_NUC4 can be found in Figure 4.10.

As can be observed in Figures 4.9 and 4.10, the performance of the Likelihood_NUC4_OpenACC function is only good for the larger inputs, however the time gained when removing the memory transfers will most likely compensate this fact.

A small explanation about the input files, the testing machine and the basis for calculating the speedup can be found in Chapter 2.
4.4 Memory Transfers Optimization

As already stated before, in order to remove the memory transfers between CPU and GPU associated with the CondLikeScaler_NUC4_OpenACC function it is necessary to parallelize 3 more functions: RemoveNodeScalers, CopySiteScalers and ResetSiteScalers. These functions do not require all of the techniques used in the example of Figure 2.8. To parallelize these functions a data region associated with a kernels pragma is needed. These for loops are simple enough for the compiler to fully understand them and for both the data region and the kernels pragma be joined in the same pragma. The modified versions of RemoveNodeScalers, CopySiteScalers and ResetSiteScalers functions can be seen in Figure 4.11 in Figure 4.12 and in Figure 4.13 respectively.

In addition to these functions it was also necessary to parallelize the part that was not parallelized in Section 4.2 of the CondLikeScaler_NUC4_OpenACC function. This last smaller loop is slightly more complex than the functions of Figures 4.11, 4.12 and 4.13. Again, data region and a
4. Bottlenecks and optimizations: an overall performance improvement

```c
#pragma acc kernels loop independent, present(scalers_gpu_aux[scalers_max_length], scalers_gpu[scalers_size])
for (c=0; c<numChars; c++)
{
  CLFlt aux;
  aux=(CLFlt) log(scalers_gpu_aux[c]);
  scalers_gpu[scP_index*numChars + c] = aux; /* store node scaler */
  scalers_gpu[scalers_index*numChars + c] += aux; /* add into tree scaler */
}
```

Figure 4.14: The most important part of the second modification made to the CondLikeScaler_NUC4_OpenACC function.

Before removing the data transfers, however, it was necessary to merge every array allocated on the GPU into one single contiguous memory array to prevent contiguous memory shortages on the GPU side. As the GPU is already executing a considerable portion of the program, there were some GPU memory issues to be attended. The program was having a runtime error that informed that some of the arrays used in the GPU were not entirely on the GPU. The total GPU memory used by the largest input of the program is now about 942 MiB which is still far away from the maximum 1536 MiB available on the used graphics board. On the other hand, since each GPU array is indeed quite large, there is a shortage of sequential memory for at least one of the arrays. Thus, as stated before, it required to allocate one single array with the size of 942 MiB to accommodate all of the other arrays. This is achieved by allocating the larger array containing all of the needed space on the CPU and then making the pointers point to specific parts of this larger array so that all of the necessary arrays are contained in the larger array. Then, when allocating space on the GPU with OpenACC, only the larger array needs to be allocated. This meant that the code in Figure 4.15 was added to the initialization of the program to deal with the CPU part of the problem. Then the data region created on Section 3.4 was modified accordingly. This can be

```c
#pragma acc kernels loop independent gang(numChars), present(scalers_gpu[scalers_length*scalers_height])
for(i=0; i<numChars; i++)
  scalers_gpu[to_index*numChars + i] = scalers_gpu[from_index*numChars + i];
```

Figure 4.12: The most important part of the modification made to the CopySiteScalers function.

```c
#pragma acc kernels loop independent gang(numChars), present(scalers_gpu[scalers_length*scalers_height])
for (c=0; c<numChars; c++)
  scalers_gpu[lnScaler_index + c] = 0.0;
```

Figure 4.13: The most important part of the modification made to the ResetSiteScalers function.

`kernels` pragma are enough to parallelize this particular for loop. The modifications made can be seen in Figure 4.14.

More details about the hardware used for testing can be found on Chapter 2.
4.4 Memory Transfers Optimization

```c
gpu_buffer_size = tiProbs_length + condLikes_length + numSitesOfPat_length + scalers_size;
gpu_buffer = (CLFlt*)malloc(gpu_buffer_size * sizeof(CLFlt));
if (gpu_buffer == NULL) {printf("Error: not enough space to allocate gpu_buffer on the CPU side!\n"); exit(-6);}
tiProbs_gpu = gpu_buffer;
condLikes_gpu = gpu_buffer + tiProbs_length;
scalers_gpu = condLikes_gpu + condLikes_length;
numSitesOfPat_gpu = scalers_gpu + scalers_size;
```

Figure 4.15: The initialization of the single buffer on the CPU side.

```c
#pragma acc data create(gpu_buffer[tiProbs_length + condLikes_length + numSitesOfPat_length + scalers_size], scalers_gpu_aux[scalers_max_length])
{
    #pragma acc update device(numSitesOfPat_gpu[numSitesOfPat_length], condLikes_gpu[condLikes_x*condLikes_y])
    /* Run the Markov chain. */
    rc = RunChain (&seed);
}
```

Figure 4.16: The allocation of the single buffer on the GPU side.

seen in Figure 4.16. This strategy allows to retain all of the previous indexes since the individual pointers to the GPU memory continue to exist.

After doing this, all of the memory transfers associated with the `condLikes` matrix and the `CondLikeScaler_NUC4_OpenACC` function were removed. All but the initial values needed that are being transferred only once in the beginning of the program as seen in Figure 4.16.

Finally there were some final optimizations to be done to the code. There was a buggy present clause in the `CondLikeScaler_NUC4_OpenACC` function that was preventing the correct parallelization of that same function. There were also some excessive memory allocations of smaller variables on the GPU side that were removed. The final version of every parallelized function can be found on Appendix A.

4.4.1 Results

This time, all of the memory transfers that can be seen in Figure 4.4 were greatly reduced. This means that it is expected that the overall performance is greatly improved. The execution time for the final implementation can be seen in Figure 4.17. A more detailed version can be seen in Figure 4.18. “Down/Root Total/Kernel”, “Scaler Total/Kernel”, “Likelihood Kernel”, “RemoveNode”, “CopySite” and “ResetSite” refer to the execution time of their respective functions without memory transfers. “Down/Root Update” and “Likelihood Data” refer to the memory transfers associated with their respective functions. “mcmc total” and “mcmc update” refer to the initial memory allocation and transfer of the initial data. “OpenACC init” refers to the initialization of OpenACC and “Serial Part” refers to the serial part of the program.

An execution time comparison between this implementation and the implementation of Section 4.2 can be seen in Figure 4.19. “Current Version” refers to the final implementation and “Previous Version” refers to the implementation of Section 4.2.
4. Bottlenecks and optimizations: an overall performance improvement

Figure 4.17: The execution time of the final version.
Figure 4.18: A detailed version of the final execution time.
4. Bottlenecks and optimizations: an overall performance improvement

Figure 4.19: An execution time comparison between the final version and the previous version (section 4.2).

The speedup for the final version can be seen in Figure 4.20. “MrBayes GPU” refers to the speedup of the whole program of this thesis. “Down/Root”, “Scaler” and “Likelihood” refer to the speedup of their respective functions. “Total functions” refers to the speedup of “Down/Root”, “Scaler” and “Likelihood” when their summed time is compared to the summed time of the original functions. From this Figure it is possible to see that the overall maximum speedup has now increased to 4.1, thus providing the program with a decent overall performance.

A small explanation about the input files, the testing machine and the basis for calculating the speedup can be found in Chapter 2.

4.5 Summary

This Chapter focused solely in improving the overall performance of the program. In order to do this, it was necessary to decrease or remove completely the memory transfers between CPU and GPU, which required the parallelization of the CondLikeScaler_NUC function. This function’s original implementation had a for loop. To simplify this loop so that the compiler could parallelize it, it was subdivided into three smaller and simpler loops. To guarantee that this parallelization would not, introduce additional memory transfers between the CPU and the GPU, the Likelihood_NUC function was also parallelized. This function had a somewhat complex loop
Figure 4.20: The detailed speedup for the final version of this thesis.
4. Bottlenecks and optimizations: an overall performance improvement

that was modified in order to make it easier for the compiler to understand and parallelize.

Finally, to remove the memory transfers 3 small functions were also parallelized, namely: RemoveNodeScalers, CopySiteScalers and ResetSiteScalers. To allow better memory management, the whole memory used on the GPU had to be allocated as a single array.

This allowed the reduction of the memory transfers which, along with additional kernel optimizations, allowed achieving an overall speedup of 4.1.
5

Comparisons with State of the Art

Contents

5.1 Comparison with nMC v1.0 ............................................. 70
5.2 Comparison with nMC v2.1 ............................................. 72
5.3 Comparison with tgMC .................................................. 73
5.4 Summary ................................................................. 74
5. Comparisons with State of the Art

In order to assess the performance improvements made in the scope of this thesis, the performance results are compared with those obtained with the state of art implementations, namely, both the first and the latest versions of nMC\(^3\) and tgMC\(^3\).

5.1 Comparison with nMC\(^3\) v1.0

This is the very first version of nMC\(^3\) and the one described in Section 2.5.2. This is the closest implementation to the one done in this thesis and thus this is the fairest comparison that is possible to be made. Nonetheless, additional comparisons with a more recent version is made in Section 5.2. In order to have a fair comparison the nMC\(^3\) was run in DEBUG, NOSHORTCUTS mode as well.

nMC\(^3\) computes the various likelihoods of the nodes in the GPU just like what this thesis has done. However, the global likelihood of the tree is computed on the CPU side. In this thesis, the global likelihood of the tree is also computed on the GPU side, effectively reducing the amount of GPU to CPU communication to a single float per iteration. The overall nMC\(^3\) can be observed in Figure 5.1. nMC\(^3\) also implements pipelining in order to reduce the time consumed with memory transfers and to keep the CPU busy while the GPU works. This allows for nMC\(^3\) to overlap node likelihood calculations (in the GPU) with global likelihood calculations (in the CPU). This thesis does not implement a similar strategy because the time consumed by memory transfers is already reduced. Though pipelining could be used in a different manner in this thesis, such a possibility would require some analysis which was not done due to the time already consumed by the implementation done. This possibility is further discussed in Section 6.1.

The execution time of this program can be seen in Figure B.1. “Total Down/Root”, “Scaler” and “Likelihood” refer to their respective functions. “Serial Part” refers to the serial part of the program which includes memory transfers between CPU and GPU. As it is possible to observe the serial part consumes practically all of the execution time of nMC\(^3\).

A time comparison between this thesis and nMC\(^3\) can be seen in Figure 5.2. In this Figure it is possible to observe that this thesis outperforms nMC\(^3\) v1.0 in the larger inputs but it does not perform so well in the smaller inputs. This is partially due to a matrix compression that nMC\(^3\) performs. In the larger inputs this matrix compression takes a large portion of the execution time and it is responsible for the enormous amount of serial time observed in Figure B.1. Another reason for this is that this thesis has less GPU to CPU transfers, as stated before, and thus in the larger inputs these transfers begin to consume more time that the computations that nMC\(^3\) does in parallel and cause the program to stall and wait for these transfers to complete. In addition to this, this thesis introduces an overhead over the nMC\(^3\) (regarding the smaller inputs) since there is an extra kernel to perform the global likelihood computation (likelihood of the entire tree) and this computation is not overlapped with node likelihoods computation in the way nMC\(^3\). This represents 70
Figure 5.1: An overview of the parallelization strategy used in nMC$^3$. 
5. Comparisons with State of the Art

5.2 Comparison with nMC$^3$ v2.1

This is the latest version available that still only implements nMC$^3$. There’s a more recent version but it begins implementing a new parallelization strategy that is not yet finished. This version had a lot of work on top of what was done to version 1.0 which was essentially optimizing kernels and debugging the previous versions.

The execution time for this version can be seen in Figure B.3. This is very similar to what was presented in Figure B.1. The similarity can also be seen in Figure B.4 which compares both versions of nMC$^3$.

An execution time comparison between this thesis and nMC$^3$ v2.1 can be seen in Figure 5.3. Again this is very similar to what was shown in the previous Section. The speedup comparison seen in Figure B.5 further confirms this similarity.
5.3 Comparison with tgMC

This implementation is the state of the art when it comes to parallelizing MrBayes using GPUs. tgMC has a similar parallelization strategy to calculate the tip to the one used in nMC. However, unlike nMC, tgMC does not employ a pipelining strategy. tgMC also requires for the tip matrix to be transformed (similar to the matrix compression performed by nMC) but this transformation is done on the GPU side. Like nMC, tgMC also computes the global likelihood on the CPU side thus having larger GPU to CPU memory transfers than this thesis. The main difference between tgMC and nMC is that tgMC does not require for the CPU to launch the individual kernels for the various functions of the program. tgMC has one single kernel to deal with this and uses CUDA specific functions that allow for a kernel to call another kernel directly within the GPU.

tgMC’s execution time can be seen in Figure 5.6. “Down”, “Root” and “Scaler” refer to their respective functions. “Serial Part” refers to the serial part of the program. tgMC no longer has a Likelihood function. It was merged with the other functions. Much like the previous comparisons, this implementation also has a rather large serial part. This is due once again to a matrix compression that this thesis does not require. tgMC also has a similar performance with nMC because tgMC was designed to be executed in several computers at the same time, but these tests were made in a single machine as stated in Section 2.6. This similarity can be seen in Figure B.7.

A time comparison between this thesis and tgMC can be seen in Figure 5.4. As it is pos-
5. Comparisons with State of the Art

![Execution time comparison between this thesis and tgMC^3.](image)

It is possible to see, the compared performance is similar to what was obtained previously. A speedup comparison can be seen in Figure B.8.

Finally, an execution time comparison among all of the implementations and this thesis can be seen in Figure 5.5.

5.4 Summary

This Chapter presented some comparisons with other parallel implementations of MrBayes, namely: nMC^3 v1.0, nMC^3 v2.1 and tgMC^3. Testing these implementations on the computer described in Section 2.6 revealed that they all have similar performances. It is worth noting that they should not have similar performances when testing with a network of computers.

From these tests it is possible to observe that this thesis’ implementation introduces some overhead when compared to the state of the art, namely for smaller datasets. On the other hand, this thesis’ implementation scales better than the state of the art when the larger datasets are observed. This is valid for all of the comparisons made in this Chapter and is due to existent differences in the parallelization strategy used.

This ends the current Chapter, hopefully the reader has gained deeper insight on the performance achieved with this thesis and the performance of the other implementations.
5.4 Summary

Figure 5.5: Execution time comparison among nMC$^3$ v1.0, nMC$^3$ v2.1, tgMC$^3$ and this thesis.
5. Comparisons with State of the Art
6 Conclusions
6. Conclusions

MrBayes is an important bioinformatics application and the acceleration of this application is important to its related research. The main purpose of this thesis is to accelerate MrBayes using the OpenACC parallel programming framework along with the parallel architecture of GPUs. Such a task typically involves identifying the program’s bottleneck, changing the program structure to adapt to a parallel architecture and managing the various overheads parallel programming introduces.

Similar works to the one performed in this thesis usually use the CUDA framework. OpenACC is a newer framework and as such thesis presented which OpenACC features are best suited for the development performed. Since both CUDA and OpenACC are frameworks for parallel GPGPU programming, this thesis presented a performance comparison between the two. Results show that CUDA can be up to twice as fast than OpenACC regarding a single kernel. This thesis reduced as much a possible the communication between the CPU and the GPU by transferring only the initial values needed to the GPU and then guaranteeing that the CPU never had to access the same data. This allows for the CPU to be focused traversing the tree and the GPU to be focused in the most time consuming calculations. At the end of each iteration a single values is transferred from the GPU to the CPU which is used to accept or reject the tree.

The development process for this thesis encountered some obstacles with OpenACC, namely:

1. The compiler cannot parallelize complex or data dependent for loops.
2. The compiler is generating a kernel with poor workload balancing.
3. The compiler does not recognise a closed data region.
4. OpenACC cannot access a regular dynamically allocated matrix of data.
5. Results started to have the value 0 after using both the copyou and the reduction clause.
6. OpenACC reports that a certain array is only partially present on the GPU when it should be in its full extent.

And the solutions found for these problems are as follows:

1. Simplification of the for loops is required and it is necessary to inform the compiler that the for loops are data independent with the independent clause. The loops cannot use pointer arithmetic and in this case it is necessary to calculate the index manually.
2. It is necessary to adjust the values of gang, workers or vector clauses and create an extra inner for loop for the compiler to understand that further parallelism is possible.
3. This can occur if a function has a possible exit point outside of the data region. The solution is to include all possible exit points in the data region or to include the whole function call in the data region.
4. OpenACC works best with contiguous data arrays, thus it is a matter of transforming the several allocated arrays into one single array of data.

5. If a variable appears both in a \texttt{copyou} and a \texttt{reduction} clause, the \texttt{copyou} clause overrides the result from the \texttt{reduction} clause. Removing the variable from the \texttt{copyou} clause solves this issue.

6. This can happen for two reasons: the data region is not being correctly detected by the compiler or there is lack of contiguous memory on the GPU to accommodate the array. There is already an aforementioned solution for the first reason. The second reason requires to check if the total memory used on the GPU does not surpass the target GPU's available memory. If it indeed surpasses then it is necessary to transfer only part of the array to the GPU at a time and perform the calculations only for the part of the memory that is present on the GPU. If, on the other hand, the used memory is bellow the available memory on the GPU, then it is necessary to join all of the arrays in the GPU memory into one single large array.

The implementation performed in this thesis achieved an overall speedup of 4.1 over the original serial MrBayes 3.2.1. As for the state of the art implementations, it is observable that this thesis’ implementation as an additional overhead. This overhead, however, is diluted in inputs with larger datasets. This means that this thesis scales better than the state of art. This is due to some differences in the parallelization strategy used by this thesis and the state of the art: this thesis tackled the communication between CPU and GPU problem by reducing the memory transfers as much as possible whereas the state of the art presents more communication between CPU and GPU and tackled the problem with a pipelining strategy to overlap communication with computation. This pipelining strategy is further aided by a matrix compression mechanism to reduce the amount of memory transferred between CPU and GPU. For larger datasets, the matrices are quite larger and the matrix compression mechanism backfires by consuming almost the totality of the execution time and slowing down the performance considerably.

### 6.1 Future work

The strategy described in Section 3.2 is still valid to continue the work done so far. From Figure 4.18 it is possible to see that the kernel associated with the \texttt{CondLikeDown} function is the one that takes most time. Some analysis should be done upon this kernel to see if it is possible to reduce it is execution time. The same applies to the summed time of the \texttt{CondLikeScaler} kernels.

Another possible strategy would be to eliminate completely the memory transfers between CPU and GPU. Though it would not provide the same improvement seen in Figure 4.19 it might still improve the overall performance. To do this it would be necessary to parallelize various other functions in the same manner as done in Chapter 4.
6. Conclusions

The use of pipelining might also prove useful. It wouldn’t be done in the same way as nMC\textsuperscript{3} though. But since the machine used for testing has 2 GPUs it would be possible to use them both to pipeline the kernels. Doing this, however, would have the drawback of memory synchronization among the GPUs.

It would also be possible to optimize the kernels to take advantage of both GPUs instead of just one of them. On the other hand, dividing the phylogenetic tree between both GPUs might prove to be a better strategy.

Regarding the phylogenetic tree itself, there are some optimizations that could be done. For instance, in the case of an unrooted phylogenetic tree, it is possible to develop an heuristic to choose an optimal fictional root to improve the workload distribution if the tree was distributed through some GPUs.

Finally it would also be possible to start studying and implementing the parallelization done in a cluster of computers. This was done in the case of tgMC\textsuperscript{3}. 
Bibliography


Bibliography


Final Version of the Parallelized Functions
A. Final Version of the Parallelized Functions

```c
#pragma acc update device(tiProbs_gpu[pL_index*tiProbs_y:16*numGammaCats],
 tiProbs_gpu[pR_index*tiProbs_y:16*numGammaCats])
{
#pragma acc kernels loop independent gang(numGammaCats),
present(condLikes_gpu[clL_index*condLikes_y:4*numChars*numGammaCats],
 condLikes_gpu[clR_index*condLikes_y:4*numChars*numGammaCats],
 tiProbs_gpu[pl_index*tiProbs_y:16*numGammaCats], tiProbs_gpu[pR_index*tiProbs_y:16*numGammaCats])
{
#pragma acc loop independent gang(numChars/numGammaCats) vector(NTHREADS)
for (c=0; c<numChars; c++)
{
#pragma acc loop independent vector(4)
for(i=0; i < 4; i++){
    register int indice = k*numChars*4+c*4;
    register int indice2 = k*16+i*4;
    register int clP_index2 = clP_index*condLikes_y;
    register int pl_index2 = pl_index*tiProbs_y;
    register int clL_index2 = clL_index*condLikes_y;
    register int pl_index2 = pl_index*tiProbs_y;
    register int clR_index2 = clR_index*condLikes_y;
    register int pl_index2 = pl_index*tiProbs_y;
    register int clR_index2 = clR_index*condLikes_y;

    condLikes_gpu[clP_index2+indice+i] =
    (tiProbs_gpu[pl_index2+indice2+AA]*condLikes_gpu[clL_index2+indice+A]
    + tiProbs_gpu[pl_index2+indice2+AC]*condLikes_gpu[clL_index2+indice+C]
    + tiProbs_gpu[pl_index2+indice2+AG]*condLikes_gpu[clL_index2+indice+G]
    + tiProbs_gpu[pl_index2+indice2+AT]*condLikes_gpu[clL_index2+indice+T])
    *(tiProbs_gpu[pR_index2+indice2+AA]*condLikes_gpu[clR_index2+indice+A]
    + tiProbs_gpu[pR_index2+indice2+AC]*condLikes_gpu[clR_index2+indice+C]
    + tiProbs_gpu[pR_index2+indice2+AG]*condLikes_gpu[clR_index2+indice+G]
    + tiProbs_gpu[pR_index2+indice2+AT]*condLikes_gpu[clR_index2+indice+T]);
}
}
```

Figure A.1: The most important part of final version of the CondLikeDownNUC4_OpenACC function.
#pragma acc update device(tiProbs_gpu[pl_index*tiProbs_y:16*numGammaCats],
tiProbs_gpu[pR_index*tiProbs_y:16*numGammaCats],
tiProbs_gpu[pA_index*tiProbs_y:16*numGammaCats])
{
    #pragma acc kernels loop independent gang(numGammaCats),
present(condLikes_gpu[clL_index*condLikes_y:4*numChars*numGammaCats],
condLikes_gpu[clR_index*condLikes_y:4*numChars*numGammaCats],
condLikes_gpu[clA_index*condLikes_y:4*numChars*numGammaCats],
tiProbs_gpu[pl_index*tiProbs_y:16*numGammaCats],
tiProbs_gpu[pR_index*tiProbs_y:16*numGammaCats],
tiProbs_gpu[pA_index*tiProbs_y:16*numGammaCats])
    for (k=0; k<numGammaCats; k++)
    {
        #pragma acc loop independent gang(numChars/numGammaCats) vector(NTHREADS)
        for (c=0; c<numChars; c++)
        {
            #pragma acc loop independent vector(4)
            for (i=0; i<4; i++)
            {
                register int indice = k*numChars*4+c*4;
                register int indice2 = k*16+i*4;
                register int clP_index2 = clP_index*condLikes_y;
                register int pl_index2 = pl_index*tiProbs_y;
                register int clL_index2 = clL_index*condLikes_y;
                register int pL_index2 = pL_index*tiProbs_y;
                register int clR_index2 = clR_index*condLikes_y;
                register int pR_index2 = pR_index*tiProbs_y;
                register int clA_index2 = clA_index*condLikes_y;
                register int pA_index2 = pA_index*tiProbs_y;

                condLikes_gpu[clP_index2+indice+i] =
                {tiProbs_gpu[pl_index2+indice2+AA]*condLikes_gpu[clL_index2+indice+A] +
                tiProbs_gpu[pl_index2+indice2+AC]*condLikes_gpu[clL_index2+indice+C] +
                tiProbs_gpu[pl_index2+indice2+AG]*condLikes_gpu[clL_index2+indice+G] +
                tiProbs_gpu[pl_index2+indice2+AT]*condLikes_gpu[clL_index2+indice+T]}*
                {tiProbs_gpu[pR_index2+indice2+AA]*condLikes_gpu[clR_index2+indice+A] +
                tiProbs_gpu[pR_index2+indice2+AC]*condLikes_gpu[clR_index2+indice+C] +
                tiProbs_gpu[pR_index2+indice2+AG]*condLikes_gpu[clR_index2+indice+G] +
                tiProbs_gpu[pR_index2+indice2+AT]*condLikes_gpu[clR_index2+indice+T]}*
                {tiProbs_gpu[pA_index2+indice2+AA]*condLikes_gpu[clA_index2+indice+A] +
                tiProbs_gpu[pA_index2+indice2+AC]*condLikes_gpu[clA_index2+indice+C] +
                tiProbs_gpu[pA_index2+indice2+AG]*condLikes_gpu[clA_index2+indice+G] +
                tiProbs_gpu[pA_index2+indice2+AT]*condLikes_gpu[clA_index2+indice+T]};
            }
        }
    }
}

Figure A.2: The most important part of final version of the CondLikeRoot_NUC4_OpenACC function.
A. Final Version of the Parallelized Functions

```c
/* rescale values */
#pragma acc kernels loop independent gang(numChars),
present(condLikes_gpu[condLikes_length], scalers_gpu_aux[scalers_max_length])
for (c=0; c<numChars; c++)
{
    CLFlt aux_scaler;
    aux_scaler = 0.0;
    register int indexA = clPtr_index * clPtr_length + 4*c + A;
    register int indexC = clPtr_index * clPtr_length + 4*c + C;
    register int indexG = clPtr_index * clPtr_length + 4*c + G;
    register int indexT = clPtr_index * clPtr_length + 4*c + T;
    if (condLikes_gpu[indexA] > aux_scaler)
        aux_scaler = condLikes_gpu[indexA];
    if (condLikes_gpu[indexC] > aux_scaler)
        aux_scaler = condLikes_gpu[indexC];
    if (condLikes_gpu[indexG] > aux_scaler)
        aux_scaler = condLikes_gpu[indexG];
    if (condLikes_gpu[indexT] > aux_scaler)
        aux_scaler = condLikes_gpu[indexT];
    indexA += clPtr_increment;
    indexC += clPtr_increment;
    indexG += clPtr_increment;
    indexT += clPtr_increment;
    if (condLikes_gpu[indexA] > aux_scaler)
        aux_scaler = condLikes_gpu[indexA];
    if (condLikes_gpu[indexC] > aux_scaler)
        aux_scaler = condLikes_gpu[indexC];
    if (condLikes_gpu[indexG] > aux_scaler)
        aux_scaler = condLikes_gpu[indexG];
    if (condLikes_gpu[indexT] > aux_scaler)
        aux_scaler = condLikes_gpu[indexT];
    indexA += clPtr_increment;
    indexC += clPtr_increment;
    indexG += clPtr_increment;
    indexT += clPtr_increment;
    if (condLikes_gpu[indexA] > aux_scaler)
        aux_scaler = condLikes_gpu[indexA];
    if (condLikes_gpu[indexC] > aux_scaler)
        aux_scaler = condLikes_gpu[indexC];
    if (condLikes_gpu[indexG] > aux_scaler)
        aux_scaler = condLikes_gpu[indexG];
    if (condLikes_gpu[indexT] > aux_scaler)
        aux_scaler = condLikes_gpu[indexT];
    indexA += clPtr_increment;
    indexC += clPtr_increment;
    indexG += clPtr_increment;
    indexT += clPtr_increment;
    if (condLikes_gpu[indexA] > aux_scaler)
        aux_scaler = condLikes_gpu[indexA];
    if (condLikes_gpu[indexC] > aux_scaler)
        aux_scaler = condLikes_gpu[indexC];
    if (condLikes_gpu[indexG] > aux_scaler)
        aux_scaler = condLikes_gpu[indexG];
    if (condLikes_gpu[indexT] > aux_scaler)
        aux_scaler = condLikes_gpu[indexT];
    indexA += clPtr_increment;
    indexC += clPtr_increment;
    indexG += clPtr_increment;
    indexT += clPtr_increment;
    if (condLikes_gpu[indexA] > aux_scaler)
        aux_scaler = condLikes_gpu[indexA];
    if (condLikes_gpu[indexC] > aux_scaler)
        aux_scaler = condLikes_gpu[indexC];
    if (condLikes_gpu[indexG] > aux_scaler)
        aux_scaler = condLikes_gpu[indexG];
    if (condLikes_gpu[indexT] > aux_scaler)
        aux_scaler = condLikes_gpu[indexT];
    scalers_gpu_aux[c] = aux_scaler;
}
```

Figure A.3: The first part of most important part of final version of the CondLikeScaler_NUC4_OpenACC function.
#pragma acc kernels loop independent gang(numChars),
present(condLikes_gpu[condLikes_length], scalers_gpu_aux[scalers_max_length])
for (c=0; c<numChars; c++)
{
   #pragma acc loop independent gang(1) vector(numGammaCats)
   for (k=0; k<numGammaCats; k++)
   {
      register int indexA = k * clPtr_increment + clPtr_index * clPtr_length + 4*c + A;
      register int indexC = k * clPtr_increment + clPtr_index * clPtr_length + 4*c + C;
      register int indexG = k * clPtr_increment + clPtr_index * clPtr_length + 4*c + G;
      register int indexT = k * clPtr_increment + clPtr_index * clPtr_length + 4*c + T;

      condLikes_gpu[indexA] /= scalers_gpu_aux[c];
      condLikes_gpu[indexC] /= scalers_gpu_aux[c];
      condLikes_gpu[indexG] /= scalers_gpu_aux[c];
      condLikes_gpu[indexT] /= scalers_gpu_aux[c];
   }
}

#pragma acc kernels loop independent,
present(scalers_gpu_aux[scalers_max_length], scalers_gpu[scalers_size])
for (c=0; c<numChars; c++)
{
   CLFlt aux;
   aux=(CLFlt) log(scalers_gpu_aux[c]);
   scalers_gpu[scP_index*numChars +c] = aux;/* store node scaler */
   scalers_gpu[scalers_index*numChars +c] += aux;/* add into tree scaler */
}

Figure A.4: The second part of most important part of final version of the
CondLikeScaler_NUC4_OpenACC function.
A. Final Version of the Parallelized Functions

```
#pragma acc data present(condLikes_gpu[condLikes_length],
scalers_gpu[scalers_length*scalers_height], numSitesOfPat[numSitesOfPat_length])
copyin(bs[4], logFreq, clPtr_index, clPtr_length, clPtr_increment) create(like_gpu[4], like)
{
#pragma acc kernels loop independent reduction(+ : lnL_gpu) private(like)
for (c=0; c<numChars; c++)
{
    register int indexA = indexAux1 + 4*c + A;
    register int indexC = indexAux1 + 4*c + C;
    register int indexG = indexAux1 + 4*c + G;
    register int indexT = indexAux1 + 4*c + T;

    like = (condLikes_gpu[indexA] * bs[A] + condLikes_gpu[indexC] * bs[C] +

    indexA += clPtr_increment;
    indexC += clPtr_increment;
    indexG += clPtr_increment;
    indexT += clPtr_increment;

    lnL_gpu += (scalers_gpu[scalers_index*numChars + c] +
               log(like) + logFreq) * numSitesOfPat_gpu[nSitesOfPat_index + c];
}
}
```

Figure A.5: The most important part of final version of the Likelihood_NUC4_OpenACC function.

```
#pragma acc kernels loop independent reduction(+ : lnL_gpu) present(scalers_gpu[scalers_length*scalers_height])
for (c=0; c<numChars; c++)
{
    scalers_gpu[lnScaler_index*numChars + c] -= scalers_gpu[scP_index*numChars + c];
}
```

Figure A.6: The most important part of final version of the RemoveNodeScalers_OpenACC function.

```
#pragma acc kernels loop independent reduction(+ : lnL_gpu) present(scalers_gpu[scalers_length*scalers_height])
for (i=0; i<numChars; i++)
{
    scalers_gpu[to_index*numChars + i] = scalers_gpu[from_index*numChars + i];
}
```

Figure A.7: The most important part of final version of the CopySiteScalers_OpenACC function.

```
#pragma acc kernels loop independent reduction(+ : lnL_gpu) present(scalers_gpu[scalers_length*scalers_height])
for (c=0; c<numChars; c++)
{
    scalers_gpu[lnScaler_index + c] = 0.0;
}
```

Figure A.8: The most important part of final version of the ResetSiteScalers_OpenACC function.
State of the Art Graphics
B. State of the Art Graphics

Figure B.1: The execution time of the nMC\textsuperscript{3} v1.0.

Figure B.2: Speedup comparison between this thesis and nMC\textsuperscript{3} v1.0, regarding the original Mr-Bayes implementation.
Figure B.3: The execution time of the nMC\textsuperscript{3} v2.1.

Figure B.4: Execution time comparison between v1.0 and v2.1 of nMC\textsuperscript{3}.
B. State of the Art Graphics

Figure B.5: Speedup comparison between this thesis and nMC\textsuperscript{3} v2.1.

Figure B.6: The execution time of tgMC\textsuperscript{3}.
Figure B.7: Execution time comparison between tgMC$^3$ and nMC$^3$ v2.1.

Figure B.8: Speedup comparison between this thesis and tgMC$^3$. 
B. State of the Art Graphics