Accelerating a Bayesian Phylogenetic Inference Application with OpenACC

João Pedro de Matos Neves

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, an overall 4.1 speed-up 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.

Index Terms—GPGPU programming frameworks, Phylogenetic Inference, OpenACC, MrBayes.

1 Introduction

MOORE's Law has impacted the world of computing since it is appearance in 1965 [Mack, 2011]. Roughly, Moore's Law states that the number of transistors in a chip doubles every 18 months [MOORE, 1998]. For the last three decades, Moore's Law along with Dennard scaling [Robert H. Dennard, 1974] and device, circuit, microarchitecture, architecture, and compiler advances led to an exponential growth of microprocessor's performance [Hadi Esmaeilzadeh, 2011]. The strive for greater performance led computer engineers to adopt parallel multi-core architectures for CPUs, and to enable general purpose features on GPUs instead of just the usual graphical features (GPGPU).

The main contributions of this document include using the new and emerging technology of OpenACC and explaining what type of OpenACC features are best suited for GPGPU computing. A comparison in terms of performance between the OpenACC technology and the CUDA technology is presented. A good profile of the target program is presented for both its serial and parallel versions. This document compares the implementation achieved with other CUDA based state-of-the-art implementations. An overall speed-up of 4.1x over the original serial target program was achieved.

The target application is MrBayes 3.2.1. MrBayes is a bioinformatics application. MrBayes uses the DNA information of multiple species and attempts to reconstruct the phylogenetic tree of that group of species [John P. Huelsenbeck, 2001]. MrBayes can greatly benefit from parallel computing in general and the power offered by GPGPU in particular because the number of possible phylogenetic trees grows exponentially with the number of species as shown by the following Equation [Felsenstein, 1978]:

$$B(s) = \frac{(2s-3)!}{2^{s-2}(s-2)!}, \tag{1}$$

where $B(s)$ represents the number of trees and s the number of species. This means that there are a

• *J. P. M. Neves is with Instituto Superior Técnico (IST) from Universidade de Lisboa.*
E-mail: joao7neves@gmail.com

total of $\sim 3 \times 10^7$ possible trees for 10 species, or $\sim 2 \times 10^{14}$ trees for 15 species.

2 Related Work

To the extent of the author's knowledge, currently there are the following parallel implementations of MrBayes that take advantage GPUs to accelerate the application's execution:

2.1 Parallel Phylogenetic Likelihood Function

The first parallel implementation of MrBayes was performed by Pratas et. al. [Frederico Pratas, 2009]. 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. This work foccused on the parallelization of the Likelihood function. This function represents most of the computational time spent by the application, and the natural target for acceleration on parallel architectures. Nevertheless, the authors have also shown that the huge memory bandwidth requirements for the data transfers between the CPU and the GPU result in a reduced performance, and other algorithmic changes should be introduced in order to further improve the execution.

2.2 nMC^3

In the sequence of the above mentioned work, an improved version of MrBayes (nMC3) has been proposed in [Jianfu Zhou and Wang, 2011]. This version has the following improvements:

- nMC^3 exploits extra parallelism between the CPU and the GPU.
- nMC^3 uses a pipelining strategy to overlap as much as possible the execution on the CPU and GPU.
- nMC^3 reduces the number of total CPU-GPU transfers by transferring them as a single batch.
- The site likelihoods are also computed by the GPU.

All of this causes nMC^3 to have an overall much better performance than the aforementioned implementation [Jianfu Zhou and Wang, 2011].

2.3 oMC^3

hMC^3 [Zhou et al., 2010] and oMC^3 [Jun Chai, 2013] propose coarser-grain parallel approaches. They introduce further improvements to nMC3, namely by using OpenMP, MPI, and CUDA. oMC^3 targets multiple computers (called nodes) each with a multi-core CPU and at least one GPU. It begins by distributing the Markov chain Monte Carlo among the nodes. Each node has some CPU cores dedicated to computation and other CPU cores to serve as GPU hosts. The first CPU cores use OpenMP to distribute the workload among them and the others use CUDA to send some of the work to the GPU [Jun Chai, 2013].

2.4 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 used in nMC^3 into a single kernel in order to prevent excessive global to local memory transfers on the GPU side [Ling et al., 2013].

3 Frameworks

3.1 CUDA

CUDA (Compute Unified Device Architecture) is a "parallel computing platform and programming model" developed by NVIDIA [nVidia, 2013]. 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 [Ali Bakhoda and Aamodt, 2009]) which allows the programmer to write functions that run on each CUDA core of the GPU in an SIMD (Single Instruction Multiple Data)[1] 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 each thread has can be accessed through the variable "threadIdx". Variable "blockIdx" is used to access the blocks unique IDs and variable "blockDim" contains the dimension of the block.

CUDA related functions usually have their names begin with "cuda". The "cudaMemcpy" is

1. NVIDIA, calls this SIMT (Single Instruction Multiple Thread) rather than SIMD but the meaning of both are similar [S. Huang, 2009].

used for transfering data both from and to the GPU. The distinction between the two cases is done with the last argument of the function: “cudaMemcpy-HostToDevice” defines a memory transfer from the CPU to the GPU and “cudaMemcpyDeviceToHost” defines a memory transfer from the GPU to the CPU.

3.2 OpenACC

Much like what OpenMP [Bob Kuhn, 2010], [OpenMP Architecture Review Board, 2013] does for CPU threads, OpenACC does for GPU threads. OpenACC reduces some of the complexity inherent to parallel programming, thus 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” [OpenACC, 2013]. 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.

OpenACC first appeared as a standard in November, 2011 with its 1.0 version specification [OpenACC-Standard.org, 2011]. This specification describes the features of the different basic pragmas available and required 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’s used to create a data region.

It is worth noting that OpenACC allows for two types of parallelization approaches: i) automatic parallelization, where the compiler attempts to parallelize the regions identified by the programmer with the basic available pragmas (”parallel” and ”kernels”) without the need for any other information; and ii) manual parallelization, that much like in OpenMP, allows the programmer to have an extra level of control over the execution by providing additional hints to the compiler. However, experimental results obtained in the study presented herein suggest that it is almost always better to specify how the parallelization is done manually. When the compiler attempts to parallelize a loop, it checks if the loop is data independent. If this condition is not verified no parallelization is done whatsoever.

4 Parallelization Strategy

In order to devise the best parallelization strategy for MrBayes under OpenACC, we first present an approach similar to the one used in [Frederico Pratas, 2009]. This allows to compare the performance of the computing system when using CUDA and OpenACC, as will be shown later in this document.

In order to have comparable results between the different implementations and different runs of the target application (MrBayes v3.2.1), we disabled some computation shortcuts and CPU optimizations, and we have fixed the random seeds to a fixed number. The target program (MrBayes 3.2.1) is set to run in $DEBUG\_NOSHORTCUTS$ mode which disables all computing shortcuts for processing phylogenetic trees.

To better understand MrBayes’ structure and to find its bottleneck, the application was profiled with KCachegrind. This revealed the number of times each function is called and the percentage of execution time it consumes. $LogLike$ (used to compute the likelihood of a tree) is the first function called more than once during the program. 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: $CondLikeDown\_NUC4$ and the $CondLikeScaler\_NUC4$. The relevance of these functions is due to the fact that they are called more frequently during the execution and represent most of the computational burden of he algorithm. In fact, $CondLikeDown\_NUC4$ is the most time consuming function of the entire program. However, for each tree[2] node

2. The tree referred is the phylogenetic tree which is represented as a tree (an acyclic graph) in the program.

(iteration), either the $CondLikeDown\_NUC4$ or the $CondLikeScaler\_NUC4$ function is called and never both. It makes no sense to parallelize the $CondLikeDown\_NUC4$ function without the $CondLikeRoot\_NUC4$ function because the percentage of times the $CondLikeRoot\_NUC4$ function is chosen over the $CondLikeDown\_NUC4$ function varies from input file to input file. Both these functions serve to calculate node likelihoods and are composed of a for loop nested in another for loop. These loops read data from the $tiProbs$ matrix and store the results in the $condLikes$ matrix.

It is worth noting that the $CondLikeDown\_NUC4$ and the $CondLikeRoot\_NUC4$ functions are also the same functions that were parallelized in [Frederico Pratas, 2009]. By doing this, this study 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 document.

Other than the two functions mentioned there is also the $CondLikeScaler\_NUC4$ function. This function comes second to the $CondLikeDown\_NUC4$ and the $CondLikeRoot\_NUC4$ functions in terms of execution time consumption but it is important to achieve a better result later in this study.

5 CONDLIKEDOWN AND CONDLIKEROOT

5.1 Implementation

Since CPU and GPU have physically and logically separated memory spaces, 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 will be to create a data region for each function surrounding parallel computation part of the code (the outermost for loops). However, MrBayes uses dynamic allocated memory to meet the needs of each input file. This means that the data length is only known at runtime and the compiler is unable to know its length beforehand. While this may seem a simple issue, that could be solved by using explicit data allocations and transfers to/from the GPU at runtime, it is not because handling this type of information in OpenACC is done in an implicit way. Therefore, to solve this issue, it is necessary to associate a variable, containing the size of the data arrays, to the data regions, such that the compiler is able to automatically extract this information. With this in mind, the correct OpenACC directives to be used in this case are "copyin" for the data that must be passed to the GPU memory and "copyout" for the results produced on the GPU and that must be copied back to the CPU.

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 going to be done. Since there are nested loops in both functions, the correct pragma to use is "kernels". These loops use pointer arithmetic and this issue is critical because not only this is not a good practice with the GPU but also it is harder for the compiler to comprehend and parallelize pointer arithmetic. This is related for example with the fact that it is much more difficult to automatically identify and follow dependencies within the code. 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 manually calculated and inserted in the code.

Even after removing the pointer arithmetic from the code the compiler still had some problems parallelizing these functions. The parallelization it was doing was not very effective because it was trying to distribute each loop iteration through the available CUDA cores and the inner loop's work can be further split into smaller work units for a better workload balancing. What this means in practice is that an extra for loop was added so that the compiler could understand that it was possible to distribute the inner loop's work as well.

5.2 Optimization considering memory accesses on GPU

The parallelization strategy described above had some problems. These problems can be summarised as:

- 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 only focuses 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. As mentioned before this is not a simple task due to the nature of OpenACC and how it implicitly tries to capture the information provided by the programmer. The memory required for this case are two matrices which are used to store the conditional likelihoods and the transitional probabilities.

Allocating a large enough space to accommodate the entirety of the data in the application has 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. In this particular case, this does not represent a significant issue since the maximum size these matrices can have for the input test cases considered represent a total of 236825344 floats, i.e., about 900 MiB. The available memory on the GPU used in this thesis is 1536 MiB. 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. The matrices mentioned above are: the $tiProbs$ matrix and the $condLikes$ matrix.

6 Overall Acceleration and Memory Transfers Optimization

6.1 CondLikeScaler

This function is composed of a main for loop witch contains two separated inner loops and some calculations. It is used to prevent underflow and 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.

This simplifies the loops for the compiler, allowing it to be able to fully understand and successfully parallelize the loops. However, to perform 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.

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.

For this particular case it was necessary to guarantee that the compiler is able to understand that the $condLikes$ matrix is already on the GPU memory and that it is necessary to allocate auxiliary variables to accommodate the intermediate results. This is done by creating a data region surrounding both loops with the array for 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 the compiler from assuming that a variable should be shared among the threads when it should be private. This misunderstanding was ruining the results.

6.2 Memory Output Reduction

In order to truly avoid data transfers related to the $condLikes$ matrix, it is necessary to parallelize the $Likelihood\_NUC4$ function in addition to the $CondLikeScaler\_NUC4$ function. This function consists of a for loop that calculates the final likelihood of the tree which is stored in the lnL variable.

There are two main challenges when parallelizing this function:

- The compiler assumes that the $like$ variable should be shared among the threads when it should be a local variable for each thread.
- 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. As for the second point, a counterintuitive approach much be followed, namely: the lnL variable must only appear in the $reduction$ clause and never appear in a $copyout$ clause. 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 and cause the loss of data that was previously stored in that variable.

By applying this last technique it was possible to remove every single memory transfer related to the *condLikes* matrix. However the scaler function also requires its own memory transfers which in turn introduced a non neglectable overhead. Resorting to KCachegrind and inspecting the original MrBayes code reveals three other functions that use the same data as in the $CondLikeScaler\_NUC4\_OpenACC$ function:

- $RemoveNodeScalers$.
- $CopySiteScalers$.
- $ResetSiteScalers$.

These functions will be tackled in the next section.

6.3 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 other functions: $RemoveNodeScalers$, $CopySiteScalers$ and $ResetSiteScalers$. To parallelize these functions a data region associated with a *kernels* pragma is needed. Each of the three functions mentioned is composed of one for loop. 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.

In addition to these functions it was also necessary to parallelize the remainder of the already partially parallelized $CondLikeScaler\_NUC4\_OpenACC$ function that was not parallelized in Section 6.1. Again, data region and a *kernels* pragma are enough to parallelize this particular for loop.

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 could not be stored on the GPU memory. Applying the techniques mentioned throughout this document requires additional memory space on the GPU. This variation is due to several intermediate results that are now kept on the GPU side. For example, for the largest input test used in this study, the memory used has grown up to 942 MiB. Although this value is still far away from the maximum 1536 MiB available on the GTX 580 graphics card used, there is not enough continuous memory space to accommodate the largest arrays. This issue can be overcome by allocating a single 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 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. The only memory transfer left is the initial values needed but this only occurs once per program execution. 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 arrays on the GPU side that were removed.

7 Comparison Between CUDA and OpenACC

With the level of development of Section 5.2, this document is able to compare it with a very similar CUDA implementation. The parallel implementation of MrBayes discussed in Section 2.1 is indeed very resemblant of this implementation. Both implementations focused on parallelizing the $CondLikeDown\_NUC4$ and $CondLikeRoot\_NUC4$ functions.

Since the implementation of Section 2.1 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 usually modify much more than just these two functions.

The comparison chart between CUDA and OpenACC can be seen in Figure 1. This chart shows both the speed-up of the functions as well as the speed-up of the whole program for both CUDA and OpenACC. "Full OpenACC Speed-up" (dark blue) refers to the speed-up of the whole program using

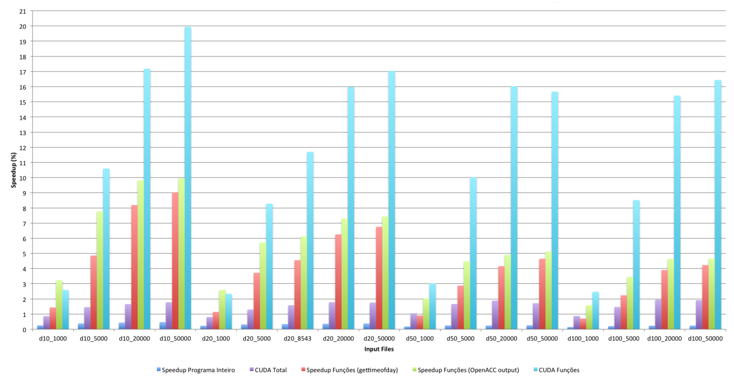

Fig. 1. Comparison between CUDA and OpenACC.

OpenACC. "Full CUDA Speed-up" (purple) refers to the speed-up of the whole program using CUDA. "Down/Root OpenACC Speed-up" (green) refers to the speed-up of the $CondLikeDown\_NUC4$ and $CondLikeRoot\_NUC4$ functions parallelized with OpenACC. "Down/Root CUDA Speed-up" (light blue) refers to the speed-up of the $CondLikeDown\_NUC4$ and $CondLikeRoot\_NUC4$ functions parallelized with CUDA.

As it is possible to see through the functions' speed-up, OpenACC has nearly half the performance of CUDA and in some cases it is even worse. The reason behind this involves some fine 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.

8 Results

All of the memory transfers between CPU and GPU 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 2. A more detailed version can be seen in Figure 3. "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

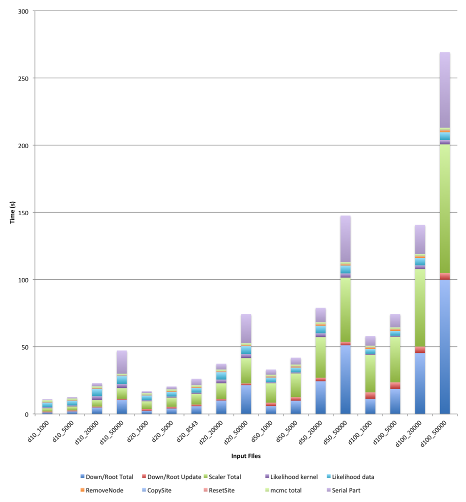

Fig. 2. The execution time of the final version.

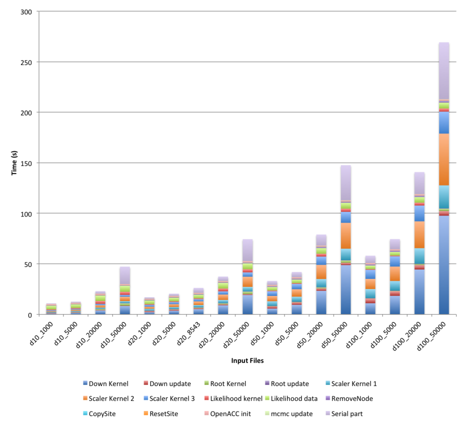

Fig. 3. A detailed version of the final execution time.

OpenACC and "Serial Part" refers to the serial part of the program.

The speed-up for the final version can be seen in Figure 4. "MrBayes GPU" refers to the speed-up of the whole application as considered in this thesis. "Down/Root", "Scaler" and "Likelihood" refer to the speed-up of their respective functions. "Total functions" refers to the speed-up of "Down/Root", "Scaler" and "Likelihood" when their accumulated time is compared to the accumulated time of the original functions. From this Figure it is possible to see that the overall maximum speed-up has now increased to 4.1, thus providing the program with a decent overall performance.

9 Conclusions

This document presented a performance comparison between OpenACC and CUDA.

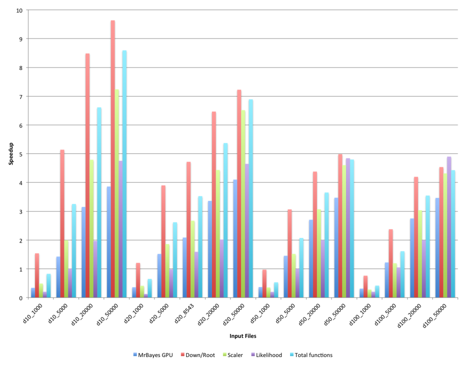

Fig. 4. The detailed speed-up for the final version of this thesis.

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

1) The compiler cannot parallelize complex or data dependent *for* loops.
2) The compiler generates kernels 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) The application returns wrong results when using both the *copyout* 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) Simplify the *for* loops and use the *independent* clause. Manually calculate array indexes.
2) Adjust the values of *gang*, *workers* or *vector* clauses and create extra inner *for* loops as necessary, thus providing finer-grained parallelization possibilities.
3) Include all possible exit points in the data region or include the whole function call in the data region.
4) Transform non-contiguous into contiguous memory regions.
5) If a variable appears both in a *copyout* and a *reduction* clause, the *copyout* clause overrides the result from the *reduction* clause. Removing the variable from the *copyout* clause solves this issue.
6) It is necessary to: make sure the data region is correct, either join all of the GPU arrays into one or transfer only part of the array at a time.

The implementation performed in this thesis achieved an overall speed-up of 4.1x over the original serial MrBayes 3.2.1.

References

[Ali Bakhoda and Aamodt, 2009] Ali Bakhoda, George L. Yuan, W. W. L. F. H. W. and Aamodt, T. M. (2009). Analyzing CUDA Workloads Using a Detailed GPU Simulator.

[Bob Kuhn, 2010] Bob Kuhn, Paul Petersen Kuck & Associates, I. E. O. C. C. C. (2010). OpenMP versus Threading in C/C++.

[Felsenstein, 1978] Felsenstein, J. (1978). The Number of Evolutionary Trees. *Systematic Zoology*, 27(1):27–33.

[Frederico Pratas, 2009] Frederico Pratas, Pedro Trancoso, A. S. L. S. (2009). Fine-grain Parallelism using Multi-core, Cell/BE, and GPU Systems: Accelerating the Phylogenetic Likelihood Function. *IEEE Computer Society*, pages 9–17.

[Hadi Esmaeilzadeh, 2011] Hadi Esmaeilzadeh, Emily Blem, R. S. A. K. S. D. B. (2011). Dark Silicon and the End of Multicore Scaling. *Proceedings of the 38th International Symposium on Computer Architecture*.

[Jianfu Zhou and Wang, 2011] Jianfu Zhou, Xiaoguang Liu, D. S. S. Q. X. and Wang, G. (2011). MrBayes on a Graphics Processing Unit. *Bioinformatics*, 27(9):1255–1261.

[John P. Huelsenbeck, 2001] John P. Huelsenbeck, F. R. (2001). MRBAYES: Bayesian inference of phylogenetic trees. *Bioinformatics Applications Note*, 17(8):754–755.

[Jun Chai, 2013] Jun Chai, Huayou Su, M. W. X. C. N. W. C. Z. (2013). Resource-efficient utilization of CPU/GPU-based heterogeneous supercomputers for Bayesian phylogenetic inference. *Springer*.

[Ling et al., 2013] Ling, C., Hamada, T., Bai, J., Li, X., Chesters, D., Zheng, W., and Shi, W. (2013). Mrbayes tgmc¡sup¿3¡/sup¿: A tight gpu implementation of mrbayes. *PLoS ONE*, 8(4):e60667.

[Mack, 2011] Mack, C. A. (2011). Fifty Years of Moores Law. *IEEE TRANSACTIONS ON SEMICONDUCTOR MANUFACTURING*, 24(2):202–207.

[MOORE, 1998] MOORE, G. E. (1998). Cramming More Components onto Integrated Circuits. *PROCEEDINGS OF THE IEEE*, 86(1):82–85.

[nVidia, 2013] nVidia (2013). http://www.nvidia.com/object/cuda_home_

[OpenACC, 2013] OpenACC (2013). http://openacc.org/.

[OpenACC-Standard.org, 2011] OpenACC-Standard.org (2011). The OpenACC^{TM} Application Programming Interface.

[OpenMP Architecture Review Board, 2013] OpenMP Architecture Review Board (2013). http://openmp.org/wp/.

[Robert H. Dennard, 1974] Robert H. Dennard, Fritz H. Gaensslen, H.-N. Y. V. L. R. E. B. A. R. L. (1974). Design of Ion-Implanted MOSFETS with Very Small Physical Dimensions. *IEEE JOURNAL OF SOLID-STATE CIRCUITS*, sc-9(5):256–268.

[S. Huang, 2009] S. Huang, S. Xiao, W. F. (2009). On the Energy Efficiency of Graphics Processing Units for Scientific Computing.

[Zhou et al., 2010] Zhou, J., Wang, G., and Liu, X. (2010). A new hybrid parallel algorithm for mrbayes. In *Proceedings of the 10th international conference on Algorithms and Architectures for Parallel Processing - Volume Part I*, ICA3PP'10, pages 102–112, Berlin, Heidelberg. Springer-Verlag.