Accelerating Phylogenetic Inference on GPUs: an OpenACC and CUDA comparison

Lídia Kuan, João Neves, Frederico Pratas*, Pedro Tomás, and Leonel Sousa

INESC-ID, IST, University of Lisbon, Lisboa, Portugal
*Intel Barcelona Research Center - Intel Labs, Barcelona, Spain
{lmlk,jpmn}@sips.inesc-id.pt
{frederico.c.pratas}@intel.com
{pfzt,las}@inesc-id.pt

Abstract. Phylogenetic inference is used to derive a "tree of life" for a collection of species whose DNA sequences are known. While several software packages have already been developed to take advantage of GPUs to accelerate phylogenetic inference, they typically require significant changes to the original code, constraining code maintenance. Recently, the OpenACC API was proposed to minimize the programming efforts on accelerator devices. In this work we evaluate the applicability of the OpenACC API for phylogenetic inference using the most recent MrBayes program (version 3.2.2). A new parallelization strategy is proposed that is specifically adapted to the latest version of MrBayes and minimizes the data transfers between the host (CPU) and the accelerating device (GPU). We further implement the proposed strategy using both the OpenACC and CUDA programming frameworks. Experimental results demonstrate that significant performance gains can be achieved using OpenACC with a reduced amount of programming effort. Comparing with state-of-art GPU's implementations, the proposed OpenACC and CUDA implementations achieve a performance gain of up to 5.2× and 5.7×, respectively. Experimental results indicate that with a reduced amount of programming effort, we achieve a performance that is only 10% inferior to one obtained with CUDA, which uses device specific optimizations.

Keywords: MrBayes, CUDA, OpenACC, Phylogenetic Inference

1 Introduction

In biology, phylogenetics is the study of evolutionary relationships among groups of organisms (e.g. species populations). Evolution is a process whereby populations are altered over time and may split into separate branches, hybridize together, or terminate by extinction. These genealogical relations between the organisms may be represented in a phylogenetic tree, that represents an hypothesis of the order in which evolutionary events are assumed to have occurred.

* The third author performed the work while at INESC-ID.

Phylogenetics trees have many important applications in medical and biological research, such as the analysis of microbial communities in the human gut [6] and in microbial mats [7], infectious diseases like Avian influenza [17, 18], or evolutionary analysis of papillomaviruses [3] that are associated to several types of human cancer, such as cervical cancer.

The mechanics of the evolutionary processes is studied by estimating the rates of nucleotide and amino acid substitutions over time, and by testing models of mutation and selection using sequence data. The problem of phylogenetic trees reconstruction based on molecular sequence data is not new in bioinformatics. However, the high pace at which biological data has been accumulating during the last decade imposes important computational challenges to fulfill the requirements of this type of applications. There are several methods for the reconstruction of phylogenetic trees based on molecular data, namely Maximum Likelihood [2] and Bayesian inference [4].

The method used in the scope of this article, the Maximum Likelihood (ML) model, represents a broadly accepted criterion to score phylogenetic trees. MrBayes [5] is a popular program for Bayesian inference on phylogenetic trees based on the ML model that implements the Metropolis Coupled Markov Chain Monte Carlo (MC^3) sampling method for Bayesian inference of phylogeny. It works by iteratively evaluating H Markov chains, each containing a proposed phylogenetic tree ψ_i. In each iteration, the Markov chains $\psi_1, \cdots, \psi_H$ are randomly perturbed to give rise to another chain of possible trees ψ'_1, ψ'_2, ..., ψ'_H. After tree perturbation, the likelihood of each tree i is evaluated to decide whether or not to replace the initial tree ψ_i with the perturbed tree ψ'_i [2]. The whole procedure is repeated until reaching a stopping criterion, such as convergence, or the maximum number of iterations.

Due to the computational burden of phylogenetic inference [21], limitations are usually imposed to the complexity of the evaluated trees. To overcome this issue, researchers turned to parallel computing in order to speed-up execution. Related work showed that Graphics Processing Units (GPUs) can achieved promising results when compared with multi-core Central Processing Units (CPUs) [14, 13, 8, 1] for this type of computation, since data parallelism can be exploited. Therefore, the proposed work is focused on GPU implementations. To the best of our knowledge the first parallel version of MrBayes MC^3 was proposed by Pratas et al. [14, 13] (gMC^3), which evaluated a set of computational platforms for decreasing the computational time of MrBayes, including multi-core CPUs, the Cell Broadband Engine, and GPUs. An improved parallel version of MrBayes 3.1.2, nMC^3, was proposed by Zhou et al. [22], where a CPU+GPU heterogeneous system is explored to reduce the phylogenetic inference computation time. The resulting speed-up comes mainly from the reduction in host-to-accelerator data transfers, from distributing the computation between the CPU and the GPU, and from overlapping data communication, between the CPU and GPU with computation. In a subsequent version of nMC^3, the authors optimized, the stream order and the thread parallelization strategy for large data sets, achieving further speed-up results. Ling, et al. [8] presented tgMC^3, a new parallel implementation of MrBayes 3.1.2 that integrates multiple functions into a single tight GPU kernel, and perform additional GPU optimizations to further decrease the computational time. In [1], Bao et al. proposed aMC^3, an adaptive implementation of MrBayes 3.1.2, to improve MC^3 on multi-GPU platforms supporting the Compute Unified Device Architecture (CUDA) programming framework. For this, Bao et al. used a dynamic scheme to determine the task granularity to distribute the workload among two GPUs.

The previously described approaches explore the CUDA platform to decrease the time required to compute MrBayes. While results show that substantial speed-ups can be achieved, significant code changes must be performed to the original code. This constrains code adaptation to new models and/or algorithms, since developing for the CUDA programming framework is an expensive and time consuming task. Recently, OpenACC, a new Application Programming Interface (API), was proposed [12, 20] to minimize the programming effort, namely on heterogeneous platforms. With OpenACC, programmer only needs to specify with compiler directives the loops and regions of code that are to be offloaded to the GPU accelerator. This simplifies programming and eases code maintenance, and allows a single source code to be compiled for a single CPU or for a CPU+GPU system.

In this paper we explore the OpenACC API to accelerate the most recent MrBayes 3.2.2 [16]. In particularly we show that by using a proper parallelization scheme, and by minimizing the data transfers, it is possible to decrease computation time by up 8× (regarding the serial CPU version), a performance up to 4x higher than state-of-art implementations. To evaluate the quality of the OpenACC mapping on the GPU, a hand-made CUDA version was implemented, which takes into account complex and time-consuming device-specific optimizations. It allows to increase the processing speed by around 10%, which translates to a 9× speed-up regarding the serial CPU version.

2 Parallel Programming Environments

2.1 Programming with CUDA

CUDA is a computing platform and programming model to explore parallelism in NVIDIA's GPUs. When programming with CUDA it is necessary to take into account several GPU architecture [10, 11] specific factors, such as the limits of the global, shared, constant and texture memories, the number of registers used per kernel and how to efficiently exploit parallelism with the limited available resources. With CUDA, to parallelize a function, a programmer has to rewrite its code according to the single-instruction multiple-data (SIMD) model. Moreover, he has to manually launch the environment to run the function in the GPU; and to be aware of the memory organization and data management mechanisms such as to efficiently map the data into the different memory levels and/or to efficiently copy data to and from the GPU. These steps are necessary to hide inherent microarchitectural inefficiencies and to tune the application.

The CUDA C allows the programmer to define C functions, called *kernels*, that are executed in parallel by N different CUDA *threads*. Each thread is given a unique ID that is accessible within the kernel through the built-in `threadIdx` variable. A kernel function can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks. These multiple blocks are organized into a one, two or three-dimensional *grid* of thread blocks, depending on the compute capability of the GPU. Each block within the grid can be identified by one, two or three-dimensional indexes accessible within the kernel through the built-in `blockIdx` variable.

The CUDA programming model assumes that the *host* (CPU) and the device maintain their own separate memory spaces, referred to as *host memory* and *device memory*, respectively. Therefore, a CUDA program manages the global, constant and texture memory spaces visible to kernels through the CUDA runtime. This includes device memory allocation and deallocation, as well as data transfer between host and device memories. Also, an amount of shared memory is available for each block of threads, which is expected to be much faster than global memory. Any opportunity to replace accesses to global memory by shared memory should therefore be exploited.

2.2 Programming with OpenACC

The OpenACC API describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators. OpenACC allows parallel programmers to provide simple hints, known as "directives", to the compiler, identifying which areas of code to accelerate, without requiring programmers to modify or adapt the underlying code to the specific architecture. By exposing parallelism to the compiler, directives allow the compiler to do the detailed work of mapping the computation onto the accelerator.

When programming with OpenACC, the programmer does not have to be concerned with the specific details of the accelerator device architecture, even though a few parameters are left to allow specific device optimizations (e.g., number of threads per block). Comparing with CUDA, the programmer only has to inform the compiler about the *for loops* to be accelerated and about the involved variables. The compiler is then responsible to tune the application, considering the device architecture characteristics.

As an example, the pseudo-code of the MrBayes `CondLikeDown` function is presented in Algorithm 1, along with the OpenACC compiler directives used to accelerate it. In the given example, the *kernels* directive was used to tell the compiler that the three nested for loops should be translated into a sequence of kernels, to be compiled and mapped on the accelerator device. Furthermore, the *loop* directive was used to inform the compiler that each nested for loop should be executed in parallel in the accelerator. Typically the loops are divided into a parallel domain, and the body of the loop becomes the body of the kernel. The

Algorithm 1: OpenACC parallel example on `CondLikeDown` pseudo-code

```
Input: Conditional likelihood arrays at left and right of the current tree node
       $cl_i$, $cl_j$
Input: Transition probability matrices $tip$
Output: Likelihood of the current tree node $cl_P$
#pragma acc kernels loop independent
foreach discrete rate $r_x$, $x \in \{1, ..., 4\}$ do
    #pragma acc loop independent
    foreach aligment column $c \in \{1, ..., m\}$ do
        #pragma acc loop independent
        foreach $tip_{row}^{r_x}$ do
            $cl_{Pi}$ = Inner Product $i(tip_{row}^{i,r_x}, cl_i^{r_x}(c))$
            $cl_{Pj}$ = Inner Product $j(tip_{row}^{j,r_x}, cl_j^{r_x}(c))$
        end
        $cl_P^{r_x} = cl_{Pi} \times cl_{Pj}$
    end
end
```

independent directive informs the compiler that the loops are data-independent with respect to each other. This allows the compiler to generate code to execute the iterations in parallel without synchronization requirements. When comparing with a CUDA implementation, the programming effort results in adding only 3 lines of code, not requiring to hand-write the *kernel*, launch the kernel environment and copy data into and out of the GPU.

As it can be observed from the example of Algorithm 1, parallelizing a program in OpenACC requires a minimal set of code changes based on the introduction of a set of compiler directives. Nonetheless, it should be noticed that, to enable parallelization, the programmer still has to perform algorithmic changes to the code and/or data structures. However, unlike in CUDA, these code changes are reduced to the minimum, and still allow the resulting code to be executed on a CPU, by signaling the compiler to ignore the directives.

3 MrBayes Parallelization

To compare, and relatively assess, the CUDA and OpenACC programming frameworks, two parallel implementations of MrBayes 3.2.2 were developed, one for each framework. In both cases, the objective is to efficiently explore the GPU's resources and the computational power in a fine-grained parallel model. To guarantee a fair comparison, both implementations follow the same strategy, which is presented in Figure 1. As it can be observed, the implementation efforts were focused on reducing the execution time through the parallelization of the most computation intensive parts of the code, and on reducing the data transfers between the CPU and the GPU.

While nMC^3, tgMC^3 and aMC^3 are parallel implementations of MrBayes 3.1.2, here we use MrBayes 3.2.2 [16]. This new version significantly differs both

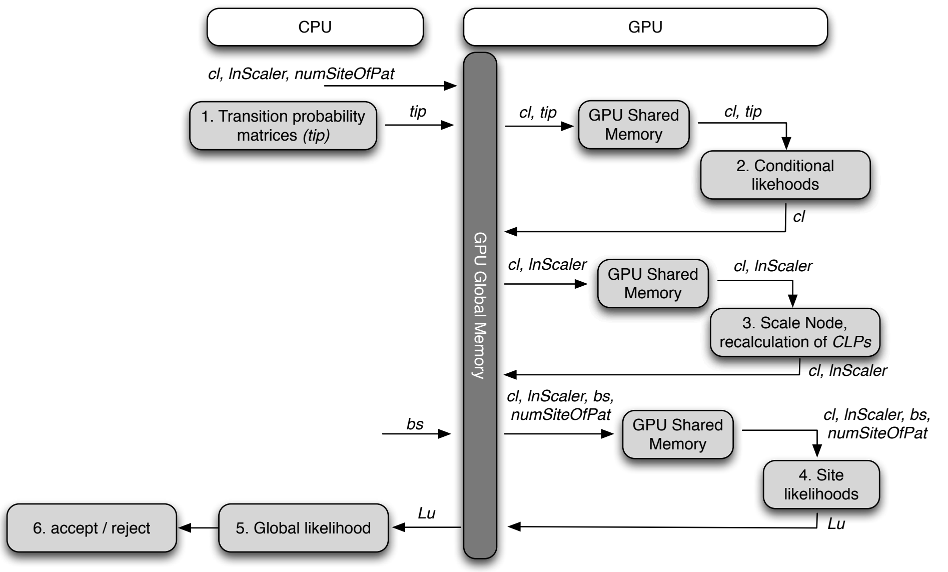

Fig. 1. Proposed implementation flowchart.

in code and in data structures, leading also to different the parallelization strategies to achieve the maximum performance. Thus, the sequential implementation of MrBayes 3.2.2 was profiled using kcachegrind [19] together with valgrind [9] to disclose the most time consuming tasks. Profiling results showed that the main step, in terms of the required computational load, corresponds to the Phylogenetic Likelihood Function (PLF), which is responsible for the calculation of the tree likelihoods using the Felsenstein's algorithm [2]. As in [14, 13], the PLF code includes three main functions: `CondLikeDown`, `CondLikeRoot` and `CondLikeScaler`. Thus, we started the implementation following the parallelization steps in [14], which requires transferring the data in and out of the GPU each time a parallelized function is called. Naturally, this lead to an associated high data communication overhead. Thus, we proceeded by allocating all the required data in the device, transferring data from the host/accelerator to the accelerator/host only when strictly required. To achieve this goal, a thorough code analysis was performed, allowing to conclude that the *condLikes* and *scaler* arrays were modified only by a small set of functions, namely the PLF and the `Likelihood` functions. Moreover, the *numSitesOfPat* array remained unchanged along a Markov Monte-Carlo Chain and was used only by the `Likelihood` function. Based on this information, all the *cl*, *lnScaler* and $numSiteOfPat$ data was allocated and transferred to the GPU only once, when the `RunChain` function is called. Thus, any nested-function requiring these arrays for its computation does not demand any additional data transfer.

While the parallel implementation follows the approach in [14], since MrBayes 3.2.2 significantly differs from the previous versions, different parallel kernels for the PLF and `Likelihood` functions were developed. Also, to decrease memory transfers, the `ResetSiteScalers`, `CopySiteScalers` and the `RemoveNodeScalers` functions were also parallelized and executed on the GPU, such as to guarantee computation correctness without requiring data transfers between the CPU and the GPU. In the final design, only the transition probability matrices and the base frequencies of nucleotide data are transferred between the CPU and the GPU, along the multiple MrBayes iterations.

3.1 OpenACC specific optimizations

As previously mentioned, all the data required for the parallelized functions were allocated in the device and used by the functions. In the OpenACC implementation all the arrays were allocated on the device before the `RunChain` function using the *create* directive. Immediately before calling the `RunChain` function, the conditional likelihood arrays and the *numSitesOfPath* data are copied to the device, using the *update* directives, as shown in Figure 1 by the first arrow from top to bottom. When analyzing Figure 1, it is worth to notice that in OpenACC the usage of the shared and constant memory is not controlled by the programmer. Thus, data copy from the global to the shared memory in Figure 1 is automatically inferred by the compiler.

Finally, in OpenACC, the `Likelihood` function was implemented in a slightly different manner than in CUDA. The final reduction step of the site likelihoods is made in the accelerator using the *reduction* directive, whereas in CUDA it is not made in the GPU.

3.2 CUDA specific optimizations

In the CUDA version, data allocation and transfers are explicitly made through synchronous or asynchronous functions. As in OpenACC, the majority of data transfers are performed when entering the `RunChain` function. However, for the calculation of the conditional likelihoods (cl), besides the transition probability matrices and the base frequencies of nucleotide data, it is also necessary to explicitly transfer the indexes of the cl array in each iteration.

In the proposed CUDA design, the defined kernel environment for the PLF functions uses a 3-dimensional block organized in a 2-dimensional grid (x, y), as shown in Figure 2. In the adopted kernel environment each grid row computes cl elements for a different transition probability matrix (tip). Therefore, the number of grid rows will always be the number of different transition probability matrices.

The thread blocks have a dimension of $b_x \times b_y \times b_z$. Since the tip data structure has 4×4 elements, $b_x = b_y = 4$, which equals the number of elements of tip. Additionally, bz depends on the number of launched threads per block. While the use of the shared memory can tune the application and help controlling the computation in a more fine-grained manner, its usage needs to be explicitly declared and has to be managed by the programmer. In the presented work, each shared memory variable involved in the computation of each block consists in an array with a number of elements equal to the number of threads per block.

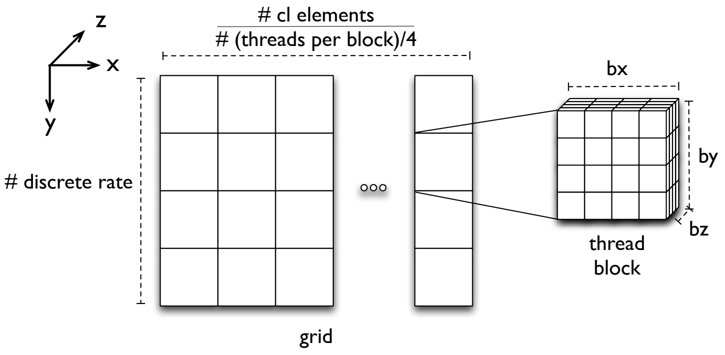

Fig. 2. Kernel thread arrangement for the computation of *cl* elements.

This guarantees that the memory access for the computation of a *cl* element is coalesced.

4 Experimental Results

To evaluate the proposed OpenACC and CUDA implementations of MrBayes 3.2.2, experimental results were obtained for two different computing platforms, both using an Intel Core i7 950@3.07 GHz with 12GB of RAM, and running the Linux operating system. One of the computing platforms is equipped with an NVIDIA GTX 580, with Fermi architecture [10], whereas the second platform is equipped with a Tesla K20c, with Kepler architecture [11]. The sequential version of MrBayes was compiled with GCC 4.6.2 with *-O3* and *-ffast-math* optimization flags. The proposed OpenACC implementation was compiled with the The Portland Group (PGI) Accelerator version 12.6, which uses OpenACC version 1.0 and CUDA 4.2. The proposed CUDA and the state-of-art nMC^3, tgMC^3 were compiled using CUDA 4.2, and the aMC^3 implementation were compiled using CUDA version 5.0.

For comparison purposes, all implementations, including the state-of-art nMC^3, tgMC^3 and aMC^3, were compared with the sequential version of MrBayes 3.2.2. As inputs, simulated DNA data sets of various sizes, generated with Seq-Gen [15], were used. They are a set of 10, 20, 50, and 100 species, each using a data set length between 5000 (5K) and 50000 (50K). For representation purposes the data sets will be hereafter referred to as dS_N, where S is the number of species under analysis and N is the data set length. As reference, a real DNA data set (d20_8543) was also considered to demonstrate the behavior of the algorithm with non-simulated data.

The total execution time in both GPUs is presented in Table 1. As it can be observed, a similar execution time behavior is found for all MrBayes implementations: in all cases, the NVIDIA K20c got slightly lower performance than the NVIDIA GTX 580. The main reason is that the K20c has a different architecture and works at a lower frequency than the GTX 580, thus requiring device specific optimizations that were not taken into consideration. Nonetheless we

Table 1. MrBayes execution time (in seconds) using the GTX 580 and the K20c GPU

	GTX 580					**K20c**				
	tgmc^3	nmc^3	amc^3	OACC†	CUDA	tgmc^3	nmc^3	amc^3	OACC†	CUDA
d10_5K	2.9	2.8	2.9	5.4	3.0	3.8	3.6	3.6	7.3	3.8
d10_20K	11.0	10.9	9.6	11.6	7.2	11.8	12.1	10.4	14.7	7.7
d10_50K	42.0	45.1	40.2	30.1	20.7	43.4	46.1	41.2	35.4	21.0
d20_5K	4.0	3.8	3.7	7.6	5.4	5.4	5.3	5.0	10.1	6.5
d20_20K	20.6	19.4	17.9	15.5	11.3	22.1	22.5	19.3	20.7	12.2
d20_50K	97.5	99.0	89.9	39.3	28.9	99.2	100.3	92.0	49.3	28.9
d20_8543	11.8	11.6	12.0	11.0	8.4	13.1	13.6	13.3	14.3	9.6
d50_5K	7.4	6.5	6.5	11.7	10.6	9.6	9.3	8.4	16.1	12.4
d50_20K	46.7	42.8	40.3	26.5	22.6	48.1	48.3	42.1	34.9	24.1
d50_50K	263.9	265.3	250.5	62.7	53.7	269.1	275.9	253.3	81.5	53.0
d100_5K	12.7	11.1	10.5	17.4	17.0	16.8	15.7	13.5	23.9	19.8
d100_20K	101.0	94.6	88.3	41.5	37.8	106.1	105.5	91.2	54.4	39.3
d100_50K	520.7	516.5	485.4	93.9	85.0	528.7	538.2	488.9	121.3	83.8

† OACC stands for the proposed OpenACC implementation

expect that a better performance could be obtained by rewriting the kernels in CUDA targeting the Kepler architecture, and by using the most recent version of OpenACC. Based on these results, from this point forward the discussion will follow considering only the GTX 580 results.

The total execution time of the implementations is presented in Figure 3 using a logarithmic scale in the y-axis. Overall, the CUDA implementation achieves the highest performance (lowest computation time), except for small data sets. In general, the proposed implementation transfers the majority of data at the beginning of execution. Since this transfer time cannot be hidden by computation time, it represents a computation overhead. However, since this strategy significantly reduces the amount of data transfers after the initial communica-

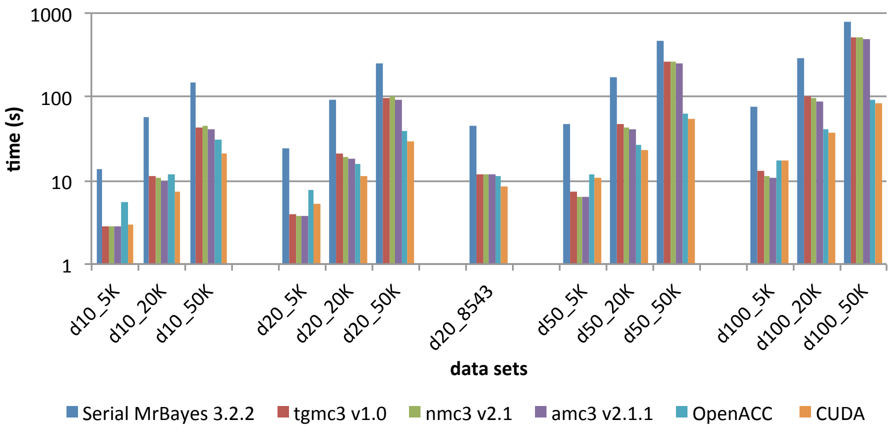

Fig. 3. Total execution time using GTX 580 GPU.

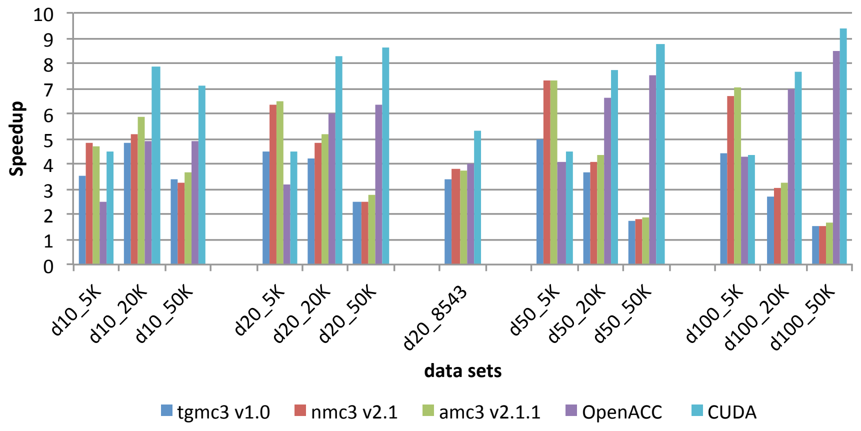

Fig. 4. Speedup achieved with GTX 580 GPU.

tion time, it allows the proposed approaches to scale better when large data sets are used. In those cases, it can be observed that both the proposed OpenACC and CUDA parallel implementations are significantly faster than the state-of-art tgmc^3, nmc^3 and amc^3 implementations.

Additional conclusions can be drawn from Figure 3, which presents the parallel implementations speed-ups regarding the sequential version of MrBayes 3.2.2. Analyzing the figure, it can be concluded that the proposed parallel approach is efficient, especially for large data sets. For the d10_50k, nMC^3, tgMC^3 and aMC^3 reach a speed-up of less than 4×, whereas the OpenACC and CUDA solutions proposed herein reach 5× and 7×, respectively. For larger data sets the advantage is even larger: for the d20_50k data set, the state-of-art implementations have a performance increase of less than 3×, regarding the sequential version, whereas the proposed techniques reach over 6× speed-up. The worst case for the state-of-art implementations regards the larger d100_50k data set, where they reach a speed-up of less than 2x. In contrast our technique achieves a speed-up of up to around 9× and 8× for CUDA and OpenACC, respectively.

Comparing the proposed CUDA and OpenACC parallelization approaches, the obtained results show that the OpenACC framework is indeed particularly interesting for applications requiring significant code maintenance. It allows achieving significant speed-up results with a quite smaller programming effort. While a small price in performance is payed, since the performance is slightly lower than with CUDA, a significant less effort is required for changing the code. In this specific application, these were concerned with the allocation of arrays in the CPU for storing GPU results and with the code structure, which had to be modified in order to ease the compiler job in identifying parallelism. Other than this, to achieve a performance speed-up of up to 8×, it was only necessary to introduce 18 lines of code in order to parallelize 7 functions.

5 Conclusions

The work herein presented evaluates the usage of OpenACC as an accelerator framework for MrBayes, a program for phylogenetic inference. A new parallelization strategy was developed which is specifically tailored for MrBayes 3.2.2. To evaluate the performance obtained with OpenACC, this strategy was also implemented using the CUDA programming framework, which allows using device specific optimizations at the cost of a much higher programming effort. Finally these results were also compared with the state-of-art nMC^3, tgMC^3 and aMC^3 parallel implementations of MrBayes.

When compared with the related work, the proposed implementations reveal a much better scaling with the data set size, allowing to achieve a processing speed-up of up to 8× and 9× with OpenACC and CUDA, respectively. This contrasts with the state-of-art nMC^3, tgMC^3 and aMC^3, which show a speed-up of less than 2× for the larger data sets.

Comparing the results of the proposed OpenACC and CUDA implementations, the later demonstrated a higher performance when compared with the former. However, OpenACC requires much less programming effort, and showed to be a more user-friendly framework to tune applications. With OpenACC, the introduction of a few lines of code in the sequential implementation allowed achieving significant speed-up results. In contrast, at the cost of a much higher programming effort and by hand optimizing the code, CUDA still allowed to decrease the computation time. Even so, it can be concluded that the OpenACC API demonstrates a great potential to tune bioinformatics applications such as MrBayes for the GPU without the necessary effort required for programming these devices with CUDA.

Acknowledgment

The work presented herein was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under projects Threads (ref. PTDC/EEA-ELC/117329/2010), P2HCS (ref. PTDC/EEI-ELC/3152/2012) and PEst-OE/EEI/LA0021/2013, and also with the Ph.D. grant with reference number SFRH/BD/65636/2009.

References

1. Bao, J., Xia, H., Zhou, J., Liu, X., Wang, G.: Efficient implementation of mrbayes on multi-gpu. Molecular biology and evolution 30(6), 1471–1479 (2013)
2. Felsenstein, J.: Evolutionary trees from dna sequences: a maximum likelihood approach. Journal of molecular evolution 17(6), 368–376 (1981)
3. Gottschling, M., Stamatakis, A., Nindl, I., Stockfleth, E., Alonso, Á., Bravo, I.G.: Multiple evolutionary mechanisms drive papillomavirus diversification. Molecular Biology and Evolution 24(5), 1242–1258 (2007)
4. Hastings, W.K.: Monte carlo sampling methods using markov chains and their applications. Biometrika 57(1), 97–109 (1970)

5. Huelsenbeck, J.P., Ronquist, F., et al.: Mrbayes: Bayesian inference of phylogenetic trees. Bioinformatics 17(8), 754–755 (2001)
6. Ley, R.E., Bäckhed, F., Turnbaugh, P., Lozupone, C.A., Knight, R.D., Gordon, J.I.: Obesity alters gut microbial ecology. Proceedings of the National Academy of Sciences of the United States of America 102(31), 11070–11075 (2005)
7. Ley, R.E., Harris, J.K., Wilcox, J., Spear, J.R., Miller, S.R., Bebout, B.M., Maresca, J.A., Bryant, D.A., Sogin, M.L., Pace, N.R.: Unexpected diversity and complexity of the guerrero negro hypersaline microbial mat. Applied and Environmental Microbiology 72(5), 3685–3695 (2006)
8. Ling, C., Hamada, T., Bai, J., Li, X., Chesters, D., Zheng, W., Shi, W.: Mrbayes tgmc3: A tight gpu implementation of mrbayes. PloS one 8(4), e60667 (2013)
9. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM Sigplan Notices 42(6), 89–100 (2007)
10. Nickolls, J., Dally, W.J.: The gpu computing era. Micro, IEEE 30(2), 56–69 (2010)
11. Nvidia: March 8th 2013: http://www.nvidia.com/content/pdf/kepler/nvidia-kepler-gk110-architecture-whitepaper.pdf
12. OpenACC-Standard.org: The openacc application programming interface v2.0 (2013)
13. Pratas, F., Sousa, L.: Applying the stream-based computing model to design hardware accelerators: A case study. In: Embedded Computer Systems: Architectures, Modeling, and Simulation, pp. 237–246. Springer (2009)
14. Pratas, F., Trancoso, P., Stamatakis, A., Sousa, L.: Fine-grain parallelism using multi-core, cell/be, and gpu systems: Accelerating the phylogenetic likelihood function. In: Parallel Processing, 2009. ICPP'09. International Conference on. pp. 9–17. IEEE (2009)
15. Rambaut, A., Grass, N.C.: Seq-gen: an application for the monte carlo simulation of dna sequence evolution along phylogenetic trees. Computer applications in the biosciences: CABIOS 13(3), 235–238 (1997)
16. Ronquist, F., Teslenko, M., van der Mark, P., Ayres, D.L., Darling, A., Höhna, S., Larget, B., Liu, L., Suchard, M.A., Huelsenbeck, J.P.: Mrbayes 3.2: efficient bayesian phylogenetic inference and model choice across a large model space. Systematic Biology 61(3), 539–542 (2012)
17. Salzberg, S.L., Kingsford, C., Cattoli, G., Spiro, D.J., Janies, D.A., Aly, M.M., Brown, I.H., Couacy-Hymann, E., De Mia, G.M., Dung, D.H., et al.: Genome analysis linking recent european and african influenza (h5n1) viruses. Emerging infectious diseases 13(5), 713 (2007)
18. Smith, G.J., Vijaykrishna, D., Bahl, J., Lycett, S.J., Worobey, M., Pybus, O.G., Ma, S.K., Cheung, C.L., Raghwani, J., Bhatt, S., et al.: Origins and evolutionary genomics of the 2009 swine-origin h1n1 influenza a epidemic. Nature 459(7250), 1122–1125 (2009)
19. Weidendorfer, J.: Sequential performance analysis with callgrind and kcachegrind. In: Tools for High Performance Computing, pp. 93–113. Springer (2008)
20. Wienke, S., Springer, P., Terboven, C., an Mey, D.: Openaccfirst experiences with real-world applications. In: Euro-Par 2012 Parallel Processing, pp. 859–870. Springer (2012)
21. Yang, Z., Rannala, B.: Molecular phylogenetics: principles and practice. Nature Reviews Genetics 13(5), 303–314 (2012)
22. Zhou, J., Liu, X., Stones, D.S., Xie, Q., Wang, G.: Mrbayes on a graphics processing unit. Bioinformatics 27(9), 1255–1261 (2011)