PBA2CUDA - A Framework for Parallelizing Population Based Algorithms Using CUDA

Ioannis Zgeras
Institute of System- and Computer Architecture
Leibniz Universitaet Hannover
30167 Hannover
Email: zgeras@sra.uni-hannover.de

Jürgen Brehm
Institute of System- and Computer Architecture
Leibniz Universitaet Hannover
30167 Hannover
Email: brehm@sra.uni-hannover.de

Michael Knoppik
Institute of System- and Computer Architecture
Leibniz Universitaet Hannover
30167 Hannover

***Abstract*—To increase the performance of a program, developers have to parallelize their code due to trends in modern hardware development. Since the parallelization of source code is paired with additional programming effort, it is desirable to provide developers with tools to help them by parallelizing source code. PBA2CUDA is a framework for semi-automatically parallelization of source code specialized in the algorithm class of Population Based Algorithms.**

I. Introduction

Modern computer architectures consist of a variety of hardware like multi-core CPUs, General-Purpose Graphics Processing Units (GPGPUs) and also specialized hardware like Field Programmable Gate Arrays (FPGAs). With this powerful parallel hardware, new challenges for software developers emerge. Serial programs do not benefit from parallel architectures as the program code is executed sequentially on a single computation node. The program code has to be parallelized to use the computation capabilities of these hardware. This process is not trivial, many constraints have to be fulfilled, like data dependencies and synchronization barriers. Furthermore, the developers have to get used to new frameworks and APIs like OpenMP [1] for multi-core CPUs or CUDA (s. Sec. II-C) for GPGPUs. The memory architecture of modern parallel hardware is complex as well. Particularly, the memories of GPGPUs consist of many different layers with different characteristics and sizes. Using the correct memory with the correct access method is crucial for fast execution of a parallel program.

This paper presents a novel approach for semi-automatically parallelizing serial source code on GPGPUs. Thereby, our tool is specialized on *Population Based Algorithms* (PBAs), an algorithm class from the area of *Biologically-Inspired Algorithms*. The main characteristics of this algorithms are the iterative execution of special functions to solve the given problem by a large number of individuals (Sec. II-B), that makes PBAs suitable for parallelization. Furthermore, by setting the focus on a specific algorithm class, our framework can perform optimization procedures regarding this particular class of algorithms.

The paper is organized as follows: Section II contains a short survey of related work. In Section III we describe the concept and implementation of our parallelization framework. Section IV presents the evaluation results. Finally, Section V concludes with further research opportunities.

II. State of the Art

A. Parallelization Frameworks

There are many parallelization frameworks in the literature transferring serial C/C++ code into CUDA code. The general approach of all frameworks is to mark parts of source code that have be transformed by the specific framework with directives. However, the features of the available frameworks differ. We have created a list of important criteria for PBA2CUDA comparing different frameworks. The criteria for the comparison are:

1) Data transfer between CPU and GPGPU is influenceable.
2) Memory on GPGPU can be allocated and freed manually.
3) Every memory hierarchy of the GPGPU is usable.
4) Loop optimization possible.
5) C++ support
6) Free to use

The results of the comparison are shown in Table I. HMPP and OpenMPC are both supporting the most features. While HMMP supports C++, it is not free to use. We have decided to use OpenMPC, as we wanted to implement an open framework and important C++ features can be supported by implementing C++ to C parser transforming, for example, C++-*Vectors* to C-*Arrays* or C++-*Classes* to C-*Structs*.

B. Population Based Algorithms

PBAs are nature inspired heuristics, all PBAs have similar structures. The main part is the population that consists of a set of solutions for a given problem. These solutions are called individuals, particles or, more general, agents. These agents execute in each iteration of the algorithm different kinds of operations to improve their solution. The quality of a solution is called *fitness*. The function or problem that the agents have to optimize (or solve) is called fitness function. Two representatives of PBAs the so called *Genetic Algorithms* (GAs) and *Particle Swarm Optimization* (PSO) that are used for this paper are now described in more detail.

Criterion	PGI [2]	OpenACC [3]	HMPP [4]	OpenMPC [5]	hiCUDA [6]	R-Stream [7]
1	+	+	+	+	+	+
2	+	+	+	+	+	-
3	-	-	+	+	+	-
4	-	-	+	+	-	-
5	+	+	+	-	-	-
6	-	-	-	+	-	-

TABLE I: Comparison PBA2CUDA criteria

1) Genetic Algorithms:: GAs [8][9] are heuristics based on the idea of natural selection. The population of GAs consists of a set of individuals that represent a solution of a given problem where every solution consists of single chromosomes. As an example the solution of an individual for an *n*-dimensional function would consist of *n* values of this function. The vector representing these values is the chromosome of this individual and every value of this vector is called *gene*. The outer iteration loop of GAs consists of three main operations - *Crossover*, *Mutation* and *Selection* - that are performed after a random initialization process until a break condition (e.g. *finding the minimum*) is achieved.

The crossover operation uses two individuals (parents) to generate new individuals (children) by crossing the solutions of the two parents. There are many different crossover operations, some popular examples can be found in [8]. In the next step, some chromosomes of the individuals are randomly changed, this operation is called mutation. Again, there are different methods for implementing mutation [8]. The individuals are now ranked based on their *fitness value* that indicates the quality of their solution. In the last step, the individuals are selected to become part of the population for the next iteration. Once again, there are many different ways to select the individuals [8]. There are many parallel implementations for GAs, see [10][11][12].

2) Particle Swarm Optimization:: PSO algorithms [13][14] have similarities to GAs but the approach is different. PSO is based on the natural behavior of birds. The population of a PSO algorithm is called *swarm* and the individuals are called *particles*. Every particle is represented by a position and a velocity where the position represents a solution of the problem and velocity the speed and direction this particle changes its position.

In each iteration step, the particles try to approximate better solutions by detecting the best neighbor and updating their velocity and position value taking into account the velocity and position values of the best neighbor. The iteration loop is repeated until a break condition is met. Like for GAs, different parallel PSO algorithms can be found in the literature [15][16].

C. CUDA

As GPUs became more and more complex, their use was no longer limited to tasks belonging to graphical programming. Different programming languages were developed to facilitate the GPUs towards more general purpose processing. The company NVIDIA published CUDA (Compute Unified Device Architecture) as a programming environment for their GPUs. CUDA became a common programming language extending the C language by a small set of instructions, allowing the programmer to develop code running in parallel. The parallel code is executed on so-called CUDA kernels. More details about CUDA can be found in [17][18].

D. ROSE

For our code analysis approach, we have used the ROSE Compiler [19], a source- to-source transformation and analysis tool. ROSE transforms the source code into an Abstract syntax tree (AST) that can be modified and transformed back into compilable source code.

III. PBA2CUDA Framework

This section describes the architecture of the parallelization framework PBA2CUDA. It is divided in three parts, in the first part the general concept of PBA2CUDA is shown while in the second part the actual implementation of the framework is described. Finally, in the third part, the pre-conditions that are necessary for the use of PBA2CUDA are shown.

A. PBA2CUDA Concept

PBA2CUDA is a framework for parallelizing serial PBAs. The generic approach is shown in Fig. 1, showing three main modules, the *Parallelization*-module, the *PBA Optimization*-module and the *CUDA Optimization*-module.

The Parallelization-module is the main module converting a serial PBA into a parallel form that can be executed on a GPGPU. The procedure of the module is semi-automatic and needs information about the parts of the code that have to be converted into CUDA-code. These areas have to be marked by pre-defined directives by the developer. Also, the Parallelization-module provides pre- and post-processing tools for the source-code that are needed by the framework. The PBA-Optimization module gets the parallelized source code as input and executes optimization operations dedicated to PBA. Finally, the CUDA Optimization module executes optimization operations focused on the CUDA specific part to accelerate execution of the code and increase the precision of the results.

B. PBA2CUDA Implementation

Here, the implementation of the generic modules shown in Section III-A is described. The concrete modules representing

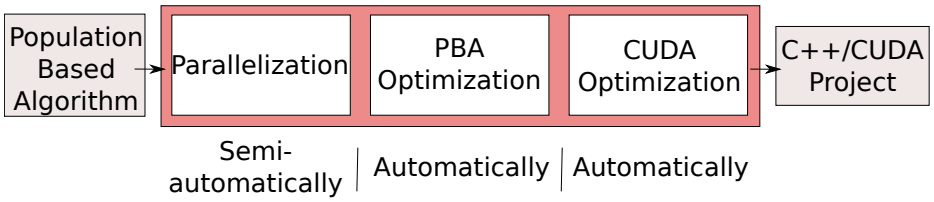

Fig. 1: PBA2CUDA Generic Framework

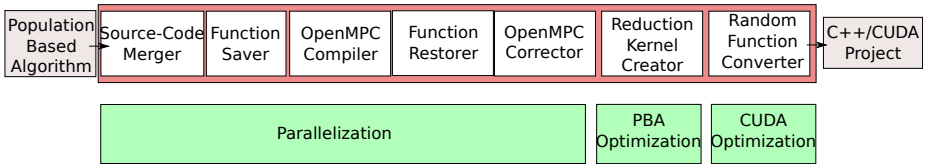

Fig. 2: PBA2CUDA Framework

parts of the generic modules are described and discussed. As shown in Fig. 2, the generic modules consist of several concrete modules specialized on single tasks of PBA2CUDA.

1) Source-Code Merger: A limitation of PBA2CUDA is the necessity to use only a single file when parallelizing source-code, which is unfavorable for larger programs. This limitation raises from the use of ROSE (s. II-D) as our main code parsing tool. To face this limitation we have implemented the *Source-Code Merger* (SCM) module to merge the single files. The SCM module searches for all available header and source files available. Next, the main method is identified and all functions of the remaining source code files are copied in the associated header files. All source-code/header files combinations that belong together are stored in a single header file, respectively. In the last step, the merged header files have to be copied into the main file in the correct order to avoid dependencies. To resolve the dependencies, the main file is parsed recursively and every header include declaration is exchanged by the dedicated merged header file. The included header files are stored to avoid multiple declarations.

2) Function Saver: The used parallelization framework OpenMPC executes optimization operations that raise errors in the source code. As an example, OpenMPC deletes functions that are referenced only as function pointers. However, the references themselves remain in the source code which leads to incorrect code. The *Function Saver* (FS) marks the referenced functions with directives to face this problem. In the following, the marked functions are stored into a separate file and deleted from the source file. Furthermore, the references are removed to avoid errors when parallelizing the code.

3) OpenMPC: We have used OpenMPC (s. II-A) to transform the serial C-code into CUDA code. This step is done fully automatically by OpenMPC and has no further improvements by our framework.

4) Function Restorer: The *Function Restorer* module restores the functions that were removed by the FS module and the parallelization procedure. Subsequently, the FS restores the missing functions and the function pointer references are restored in the source file.

5) OpenMPC Corrector: The OpenMPC compiler generates different errors that can not be corrected by the previous modules and have to be corrected by the *OpenMPC Corrector* (OC) module. In the following, the possible errors are enumerated and the used correction method is described.

1) Sometimes OpenMPC deletes the __host__ declaration in front of a function. This declaration indicates the compiler that the corresponding function can be executed on the host (CPU) system. All __device__ device expressions, that determine a device function, therefore a function that is executed on the GPGPU, are replaced by __host__ __device__ expressions. Functions with this markings can be executed on the GPGPU as well as on the CPU.
2) The NULL pointer is replaced by the expression *(void*)0*. This expression is readable for most compilers but raises an error when parsed by ROSE. Therefore, the OC module transforms the (void*)0 expression back to a standard NULL expression.

6) Reduction Kernel Creator: One of the main tasks in PBAs (s. II-B) is to find the best individual after each iteration. Comparisons between single threads on the GPGPU are not trivial due to synchronization issues and wasted clock cycles while waiting for the last thread to end its task. Generally, a simple search algorithm to find the best value is within the complexity $\mathcal{O}(n)$, where n is the number of different values. Every single value has to be compared iteratively with its neighbours until all values have been compared. As this operation is essential for PBAs, we have implemented the *Reduction Kernel Creator* (RKC) module to speed up comparison operations on GPGPUs. The reduction method is a common approach for finding the best value within a set of values. The main approach is shown in Fig. 3. $n/2$ threads have to be executed in the first iteration step to compare n values. Every thread compares two values and saves the best value for the next iteration step. The next $(n/2)/2$ threads are started and execute the same operation. The algorithm stops when the best value is found.

This approach speeds up the PBA and is in the complexity class $\mathcal{O}(log(n))$. A drawback of this approach is the use

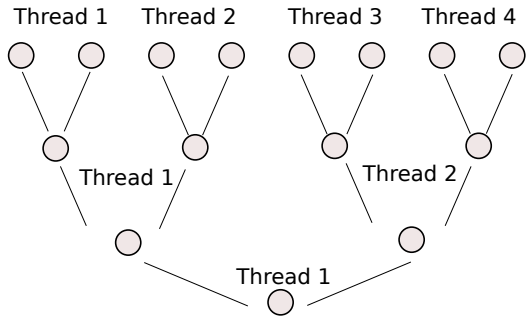

Fig. 3: Reduction procedure

of a small number of threads, that leads to an underutilized GPGPU.

7) Random Function Converter: Random calls are an essential part in PBAs for different operations like mutation or crossover (s. II-B). Unfortunately, the OpenMPC compiler does not consider external function calls like the C *rand()*. We have implemented the *Random Function Converter* (RFC) module to transfer standard C random calls into CUDA random calls using the CUDA library *thrust*. Two options have to be considered: The random call can be found directly in the kernel function or the random call is nested in a device function called by the kernel function. RFC parses the functions recursively to detect every random call and exchanges the call with a thrust random call. A seed is transferred from the host to the device function and concatenated with the thread id of the threads executed on the GPGPU to obtain independent random numbers

C. Preconditions for automatic parallelization

The serial source code that is to be transferred by PBA2CUDA has to meet some preconditions. These preconditions arise out of the used frameworks like ROSE or OpenMPC. The parts of the source code that are to be parallelized have to be written in C without C++ constructs. Furthermore, the parts that are to be parallelized have be marked using OpenMPC directives. We are working on methods to extend PBA2CUDA. Some of our work in progress can be found in Sec. V.

IV. Evaluation

To evaluate to quality of the parallel souce-code generated by PBA2CUDA we have implemented different PBAs (GA and PSO) solving common benchmark functions that can be found in [20]. We then compared the achieved speedup of the automatically parallelized algorithms, manually parallelized implementations and parallelization on the CPU (OpenMP [1]) in relation to serial versions of the algorithms. Due to page limitations, we present in this paper two functions, the *Euclid* function and the more complex *Griewank* function:

1) Euclid: $f(\vec{x}) = \sqrt{\sum_{i=1}^{n}(x_i - 500)^2}$

2) Griewank: $f(\vec{x}) = \frac{1}{4000}(\sum_{i=0}^{n-1} x_i^2) + (\prod_{i=0}^{n-1} \cos(\frac{x_i}{\sqrt{i+1}})) + 1$

We have chosen the suggested parameters from [20] for the dimension size (30 and 100) and have also evaluated a larger dimension size (500). The parameters for the population size are 100 and 500, respectively, covering medium and large population sizes. The test bench consists of an Intel Core-i7-2960XM with 4 cores and a NVIDIA GeForce GTX 580M GPGPU with 384 CUDA-cores.

A. Results: Genetic Algorithm

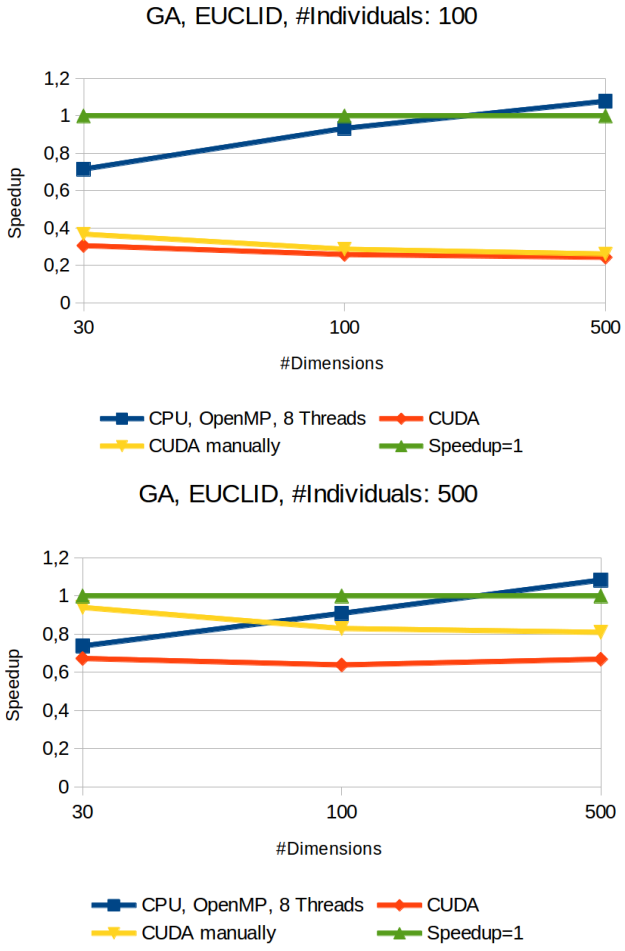

Fig. 4: Genetic Algorithm - Euclid function

1) Euclid Function:: In Fig. 4, the results for the GA algorithm solving the Euclid function are shown. On the left figure, the results using an individual size of 100 are shown while on the right figure, the results for 500 individuals are shown. The parallel versions of the algorithm do not perform well compared to the serial version (Speedup=1 - green curve) due to the simplicity of the Euclid function. The parallelization overhead is much larger than the speedup gained by parallelizing the algorithm. Comparing the parallel versions, the OpenMP version (CPU, OpenMP,8 Threads - blue curve) shows the best results following by the manually implemented CUDA version (CUDA manually - yellow curve) and the PBA2CUDA version (CUDA - red curve).

2) Griewank Function:: The Griewank function is more complex to compute than the Euclid function. In Fig. 5

the results of the GA computing the Griewank function are shown. Even for smaller population sizes (upper figure) a speedup using the parallel versions of the program is achieved. Whereby, the OpenMP version outperforms both the manually implemented CUDA version and the automatically parallelized version by PBA2CUDA. However, for larger population sizes (bottom figure) the CUDA versions outperform the OpenMP version of the program confirming the general assumption that a parallelization of GPGPUs is worthwhile only for large problems. Here again, the manual CUDA implementation shows better results than the PBA2CUDA version. This is achieved by implementing all optimization techniques of PBA2CUDA in our manually optimized implementation of the algorithm.

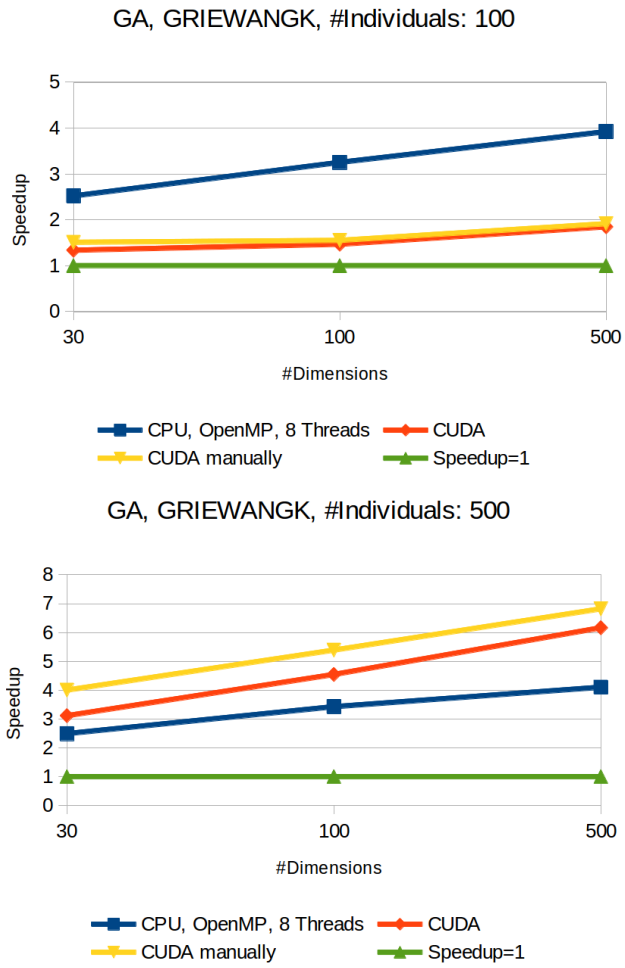

Fig. 5: Genetic Algorithm - Griewank function

B. Results: Particle Swarm Optimization

1) Euclid Function:: The results of the PSO solving the Euclid function show slightly different characteristic than the GA. While the OpenMP version again performs better than the CUDA versions, all parallelized versions perform better than the serial version of the PSO in larger problem sizes (right figure). The reason for this performance is the implemented Reduction Kernel (RK), that is only used for the PSO and can not be used by the GA implementation. The RK is also the reason for the sharp bend in the curves for population sizes greater than 100. While the RK speeds up the calculation of the best particle, it does not utilize the GPGPU very well leaving idle cores (s. Sec. III-B). For larger problem sizes, this leads to a reduction of the RK performance.

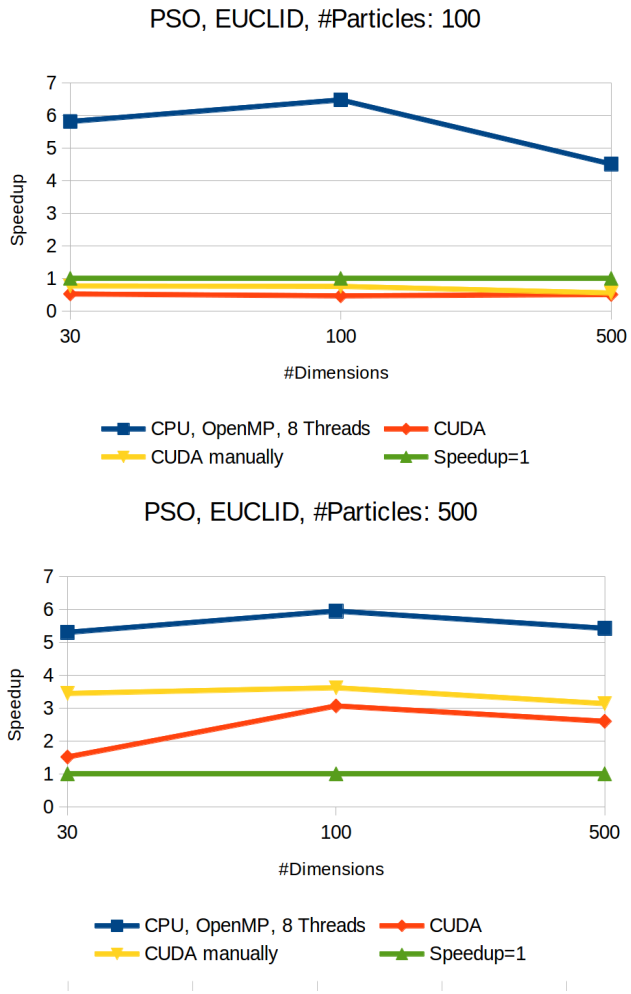

Fig. 6: Particle Swarm Optimization - Euclid function

2) Griewank Function:: Similar to the Euclid function, all parallel versions perform better than the serial version of the PSO for larger problems (right figure). Furthermore, the CUDA versions, both the manually implemented and the PBA2CUDA version, perform better than the OpenMP version. The computation time of the RK has a smaller impact in the general execution time due to the relatively large compute time of the other PSO functions and by this the CUDA versions have no quality fall-off in their performances for population sizes greater 100.

The evaluation results show similar performance of the versions parallelized by PBA2CUDA and the manually parallelized versions of the PBAs. Considering the minimal effort parallelizing a PBA with PBA2CUDA the results look promising and we want to expand the build-in optimization procedures to achieve even better results and support more parallelization patterns.

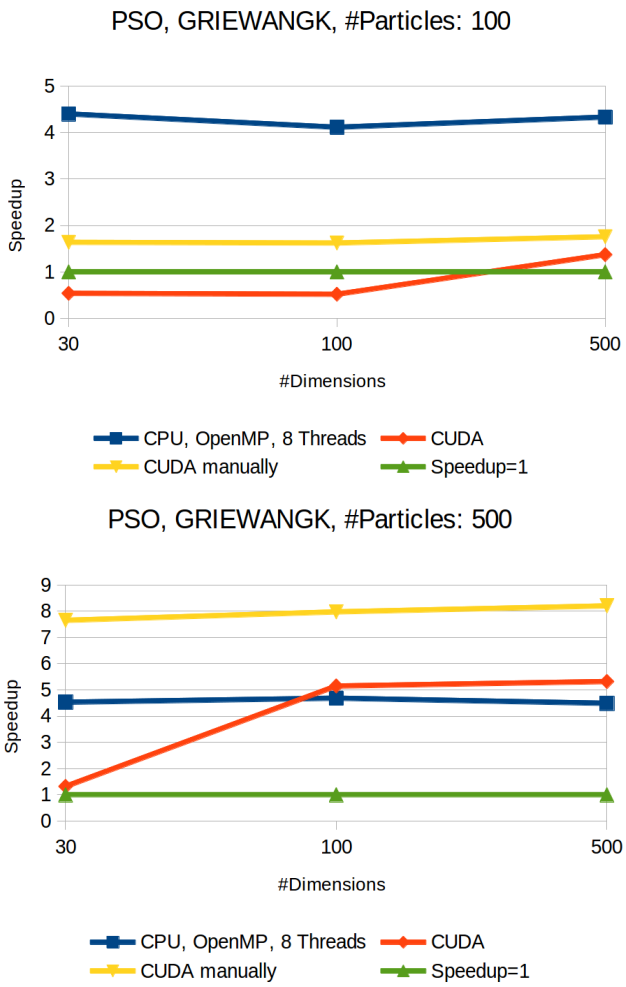

Fig. 7: Particle Swarm Optimization - Griewank Function

V. Conclusion and Future Work

We have shown in our paper a novel approach for half-automatically parallelization of PBAs using CUDA. The PBA2CUDA framework performed in our evaluation scenarios almost as well as a manually optimized CUDA version. We are working on expanding the framework and on making it more user friendly. At the moment, OpenMPC directives have to be inserted in the source code defining the parallel parts and determining on which GPGPU memory the data has to be transferred. We are working on a procedure to simplify these directives and to automatically determine the correct memory for the data. Furthermore, we are working on a decision process based upon speedup analysis mapping the source code on the correct hardware (multi-core/GPGPU) based upon the best predicted speedup automatically. Additionally, we are working on further automatically performed optimization techniques for PBAs similar to the Reduction Kernel shown in this paper.

References

[1] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[2] M. Wolfe, "Implementing the pgi accelerator model," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 43–50.

[3] R. Reyes, I. López-Rodríguez, J. J. Fumero, and F. de Sande, "accull: An openacc implementation with cuda and opencl support," in *Euro-Par 2012 Parallel Processing*. Springer, 2012, pp. 871–882.

[4] S. Bihan, G.-E. Moulard, R. Dolbeau, H. Calandra, and R. Abdelkhalek, "Directive-based heterogeneous programming–a gpu-accelerated rtm use case," in *Proceedings of the 7th International Conference on Computing, Communications and Control Technologies*, 2009.

[5] S. Lee and R. Eigenmann, "Openmpc: Extended openmp programming and tuning for gpus," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–11.

[6] T. D. Han and T. S. Abdelrahman, "hi cuda: a high-level directive-based language for gpu programming," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2009, pp. 52–61.

[7] A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin, "A mapping path for multi-gpgpu accelerated computers from a portable high level programming abstraction," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 51–61.

[8] M. Mitchell, *An Introduction to Genetic Algorithms (Complex Adaptive Systems)*, third printing ed. A Bradford Book, Feb. 1998. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/0262631857

[9] D. E. Goldberg and J. H. Holland, "Genetic algorithms and machine learning," *Machine Learning*, vol. 3, pp. 95–99, 1988, 10.1023/A:1022602019183. [Online]. Available: http://dx.doi.org/10.1023/A:1022602019183

[10] T. V. Luong, N. Melab, and E.-G. Talbi, "Gpu-based island model for evolutionary algorithms," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation - GECCO '10*. New York, New York, USA: ACM Press, Jul. 2010, p. 1089. [Online]. Available: http://dl.acm.org/citation.cfm?id=1830483.1830685

[11] ——, "Parallel hybrid evolutionary algorithms on gpu," *IEEE Congress on Evolutionary Computation CEC*, 2010. [Online]. Available: http://hal.inria.fr/inria-00520466/en/

[12] M. Parrilla, J. Ar, and S. Dormido-canto, "Parallel evolutionary computation: Application of an ea to controller design."

[13] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Neural Networks, 1995. Proceedings., IEEE International Conference on*, vol. 4, nov/dec 1995, pp. 1942 –1948 vol.4.

[14] R. Poli, J. Kennedy, and T. Blackwell, "Particle swarm optimization," *Swarm Intelligence*, vol. 1, pp. 33–57, 2007, 10.1007/s11721-007-0002-0. [Online]. Available: http://dx.doi.org/10.1007/s11721-007-0002-0

[15] Z.-h. Zhan and J. Zhang, "An parallel particle swarm optimization approach for multiobjective optimization problems," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation - GECCO '10*. New York, New York, USA: ACM Press, Jul. 2010, p. 81. [Online]. Available: http://dl.acm.org/citation.cfm?id=1830483.1830497

[16] Y. Zhou and Y. Tan, "Gpu-based parallel particle swarm optimization," in *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, may 2009, pp. 1493 –1500.

[17] NVIDIA, *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[18] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," in *IEEE Micro*, vol. 28, no. 2. Los Alamitos, CA, USA: IEEE Computer Society Press, 2008, pp. 39–55.

[19] D. J. Quinlan, "Rose: Compiler support for object-oriented frameworks," *Parallel Processing Letters*, vol. 10, no. 2/3, pp. 215–226, 2000.

[20] J. Vesterstrom and R. Thomsen, "A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems," in *Evolutionary Computation, 2004. CEC2004. Congress on*, vol. 2, june 2004, pp. 1980 – 1987 Vol.2.