Department Informatik

Technical Reports / ISSN 2191-5008

Steffen Limmer, Dietmar Fey, Johannes Jahn

GPU Implementation of a Multiobjective Search Algorithm

Technical Report CS-2011-03

April 2011

Please cite as:

Steffen Limmer, Dietmar Fey, Johannes Jahn, "GPU Implementation of a Multiobjective Search Algorithm," University of Erlangen, Dept. of Computer Science, Technical Reports, CS-2011-03, April 2011.

GPU Implementation of a Multiobjective Search Algorithm

Steffen Limmer¹, Dietmar Fey¹, Johannes Jahn²

Computer Architecture¹, Applied Mathematics II²

Dept. of Computer Science¹, Dept. of Mathematics², University of Erlangen, Germany

{Steffen.Limmer,Dietmar.Fey}@informatik.uni-erlangen.de, jahn@am.uni-erlangen.de

Abstract—In this paper we observe the possibility to accelerate a search algorithm for multiobjective optimization problems with help of a graphics processing unit. Besides an implementation we present test results for it and the conclusions that can be drawn from these results.

Index Terms—multiobjective search algorithm, GPGPU, optimization, parallelization, CUDA

I. INTRODUCTION

A multiobjective optimization problem is an optimization problem of the following form:

$$\min \begin{pmatrix} f_1(x) \\ \vdots \\ f_m(x) \end{pmatrix}$$

subject to the constraints

$$g_1(x) \leq 0$$

$$\vdots$$

$$g_p(x) \leq 0$$

$$x \in \mathbb{R}^n$$

with given functions $f_1, \dots, f_m, g_1, \dots, g_p: \mathbb{R}^n \rightarrow \mathbb{R}$. Let $S \subset \mathbb{R}^n$ denote the non-empty set of feasible points. The solution of such an optimization problem is in general not a single point, but a whole set of so called *Pareto optimal* points. A point $\bar{x} \in S$ is Pareto optimal if there exists no $x \in S$ with

$$f_i(x) \leq f_i(\bar{x}) \text{ for all } i \in \{1, \dots, m\}$$

and

$$f_j(x) < f_j(\bar{x}) \text{ for at least one } j \in \{1, \dots, m\}.$$

The set of all Pareto optimal points is also called *Pareto set*. One method to determine an approximation of

this set for a given problem are for example evolutionary algorithms [1]. Another approach, which is simple to adopt and does not require any deeper knowledge about the optimization problem, is the multiobjective search algorithm with subdivision technique, presented in [2]. This algorithm consists of four parts from which the in general most compute intensive one seems suitable for the parallelization on a graphics processing unit (GPU).

Modern GPUs are more and more used for applications other than graphics processing (also termed as general-purpose computing on graphics processing units). They contain a lot of computational units and offer a high peak performance at a comparatively low price and power consumption. The development of programming APIs like CUDA and OpenCL made them conveniently applicable for general applications, especially in the area of high performance computing [3].

Present GPUs contain multiple multiprocessors. Each multiprocessor in turn contains multiple compute cores, the so called shaders, which work according to the SIMD principle and a dedicated memory (also called shared memory because it is shared among the shaders of a multicore). Additionally a GPU contains a global memory which is accessible by all shaders.

For the programming of a GPU in CUDA, subroutines denoted as kernels are implemented. These kernels are executed by threads running on the various shaders. The started threads are divided into smaller subgroups, the so called thread blocks. All threads of a thread block are scheduled to the same multiprocessor, thus having all access to the shared memory of this multiprocessor.

This paper is structured as follows: The optimization algorithm is described in the next section. Section III presents a GPU implementation of it and test results that were achieved with help of three test problems. Finally, Section IV will conclude the present paper with a short summary.

II. OPTIMIZATION ALGORITHM

The multiobjective search algorithm with subdivision technique consists of the following four steps:

- 1) Determine a rough approximation $U \subseteq S$ of the Pareto set with images $V = f(U)$. For this paper this is done with the Graef-Younes random search with backward iteration (see [2] for more details), which first determines an approximation of the Pareto set with a fixed number of points and then reduces these points to the minimal points among them.
- 2) Subdivide the from step 1 resulting set U in a number of k subsets U_i by defining an appropriate covering of U by a set of n -dimensional intervals $I_i \subset \mathbb{R}^n, i = 1, \dots, k$ ¹ and setting $U_i := I_i \cap U$. Let V_i denote the image set $f(U_i)$ of U_i .
- 3) Choose an $l \in \mathbb{N}$ and extend each U_i to l elements in the following way:
for $i = 1, \dots, k$ do:
 while $(\#U_i < l)$ do:
 choose a random point $x \in I_i \cap S$
 if $(f(x) \notin V_i \ \& \ f(x) \not\leq y \ \forall y \in V_i)$ then
 $V_i := V_i \cup \{f(x)\}, U_i := U_i \cup \{x\}$
 end if
 end while
end for
- 4) Form $U := U_1 \cup \dots \cup U_k$ and determine all Pareto points in U (with corresponding image points).

The first step determines an overview over the Pareto set which is then refined in the subsequent steps. The third step of the algorithm seems suitable for a parallelization. The next section describes a GPU implementation of it.

III. GPU IMPLEMENTATION

We implemented the third step of the algorithm as a CUDA kernel. The parallelization is done in the following way: For each interval I_i a thread block b_i consisting of 512 threads is started. Each b_i performs the following steps (a)-(c) until U_i (respectively V_i) contains the desired number l of points:

- (a) Each thread of the block generates random points $\in I_i$ until it has found a point x that is in the set S of feasible points (those points, fulfilling the constraints of the optimization problem). The result is a set U_c of 512 candidate points $\in S$ (one for each thread).
- (b) Each thread computes the image point $f(x)$ of its random point and tests it against the currently in V_i

contained image points. If there is a point $y \in V_i$ with $y \leq f(x)$, then x is no longer a candidate for the insertion in U_i . Let U'_c be the set of points that still come in question for insertion, after this step.

- (c) Now the images of the points in U'_c are compared among themselves. For this reason each thread whose random point is contained in U'_c , compares the corresponding image point with images from all points in U'_c that were generated by threads with a lower thread id. In this way the algorithm produces the same result as if all random points were generated and evaluated in a serial manner. The after this step remaining points are finally inserted in U_i respectively V_i .

By this form of parallelization the thread blocks are completely independent from each other and thus no shared data in the global memory is required. Inside a thread block the steps (a) and (b) can be done independently by all threads without synchronization. But after step (b) and during step (c) synchronization is required since some threads potentially need data which is written respectively modified by other threads and it has to be insured that all threads work with the actual data, at all times. Especially, it has to be guaranteed, that no point is inserted in U_i , for which the necessary evaluation is not finished, yet.

The generation of the random points is done with help of the CUDA library CURAND, since the call of the C function `rand()` from inside a CUDA kernel is not permitted. The different seeds for all the threads are generated in advance on the host CPU per `rand()`.

A. Tests

We compared a serial CPU implementation of step 3 with the GPU implementation in order to test the GPU implementation. Both versions were compiled with optimization level 3. The CPU implementation was executed on a 2.6 GHz Intel Xeon processor and the GPU implementation on an NVIDIA Tesla C2050 with 448 shaders (14 multiprocessors with 32 shaders each). Both implementations were tested with help of three test problems that were also used in [2]. For the run-time determination both implementations were run five times. The runtimes of the GPU implementation contain the required time for the transfer of data from the host memory to the device memory and vice versa. The first test problem (a modification of Example 11.4 from [4]) is the following:

¹ $\{I_1, \dots, I_k\}$ is a covering of U if $U \subseteq \bigcup_{i=1, \dots, k} I_i$.

$$\min \begin{pmatrix} -x_1 \\ x_1 + x_2^2 - \cos(50x_1) \end{pmatrix}$$

subject to the constraints

$$x_1^2 - x_2 \leq 0$$

$$x_1 + 2x_2 - 3 \leq 0$$

$$(x_1, x_2) \in [-1.5, 1] \times [0, 2.25] \subset \mathbb{R}^2$$

We used the following optimization parameters for that problem: in step 1 5000 points are created and backward iteration is applied to them. The domain $[-1.5, 1] \times [0, 2.25]$ is decomposed in 20×20 regular intervals. On those intervals, containing points from step 1, step 3 is applied which generates 5000 points per interval (i.e. $l = 5000$).

The average runtime of step 3 on the CPU was 204.04 s while it was 32.40 s on the GPU. That makes a speed up of 6.3. There are several reasons why no higher speed up could be achieved on the GPU. One of them is that the workload is not evenly distributed over the intervals and thus not over the thread blocks as well. The number of random points that have to be created differs much from interval to interval. This is because the intervals contain different numbers of points from step 1 before step 3 is applied to them and first of all because they cover different sized regions of the set S of feasible points and of the Pareto set. Thus, dependent on the size of these regions different numbers of random points have to be generated and evaluated before one of the set S is found and different numbers of points of S have to be generated and evaluated until the required number l of points in U_i is reached. Figure 1 shows the set of feasible points for test problem 1 in blue and the intervals, that where handled in step 3 in one test run. The points resulting from step 3 are shown in light green. As one can see, the intervals cover differently sized regions of S and the generated points are differently wide spread.

In five test runs the average total number of generated random points in step 3 was 4781840644 and the average maximal number for an interval was 1265010093. There were always 19 intervals that were handled in step 3. Thus, in average there was one of the 19 intervals resp. blocks that had to handle 26 % of the total number of points.

Another problem besides the imbalance of the workload among the intervals is, that there are a lot of divergent branches for the threads of a block (respectively a

Fig. 1. The intervals, the set S of feasible points (blue) and the in step 3 generated points (light green) for test problem 1.

thread warps). Not only the number of points that have to be generated until a point of S is found, differs from thread to thread, but also the number of comparisons that have to be done for a point to decide if it can be inserted in U_i or not. These divergent branches can have a big impact on the performance on the GPU.

A further problem that limits the speed up, is that the size of the shared memory (48 kByte) is not sufficient to hold all the points of V_i (80000 Bytes for the test problem). Thus, a lot of accesses to the global memory have to be done in order to compare a generated random point with the points, already contained in U_i . Global memory accesses on a GPU are relatively slow.

The second test problem (proposed by Tanaka et al. [5]) is the following:

$$\min \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

subject to the constraints

$$-x_1^2 - x_2^2 + 1 + 0.1 \cos(16 \arctan \frac{x_1}{x_2}) \leq 0$$

$$(x_1 - 0.5)^2 + (x_2 - 0.5)^2 - 0.5 \leq 0$$

$$(x_1, x_2) \in [0, 1.3] \times [0, 1.3] \subset \mathbb{R}^2$$

We tested it with the following settings: 5000 points are generated in step 1, the domain $[0, 1.3] \times [0, 1.3]$ is partitioned in 20×20 intervals and in step 3 1000 points are determined for each of these intervals, that contains points from step 1. With these parameters the CPU implementation took in average 273.45 s and the GPU implementation 39.58 s, meaning a speed up of 6.9. So the speed up is a little bit higher compared to test problem 1. This result seems peculiar since the

imbalance of the intervals respectively thread blocks is even higher like for problem 1. In average 1938553286 random points were generated in total and the maximal number of generated points for a block was in average 937238083, meaning that this block had to generate and process 48 % of all points. The total number of processed intervals was 27 (illustrated in Figure 2).

Fig. 2. The intervals, S (blue) and the in step 3 generated points (light green) for test problem 2.

The reason why the speed up compared to problem 1, where the block with the maximal number of random points had to process only 26 % of all points, is nevertheless higher, seems to be a better ratio of computations compared to memory accesses. For problem 1 the CPU took about 0.0427 s for one million points and for each point 1.6 comparisons with already in U_i contained points had to be done in average. These comparisons require expensive global memory accesses on the GPU. For problem 2 the CPU took more than 3 times as much time to process 1 million points, namely circa 0.1411 s , because the check of the constraints is relative compute intensive. But the average number of comparisons for one point was only 0.77 (about a half of the comparisons for problem 1). With another subdivision of the domain it is possible to improve the load balance among the blocks. With a partition in 22×22 intervals it becomes much better, resulting in an improvement of the speed up to about 12.

The third and last test problem we used (and which is

proposed by Kursawe [6]) is the following:

$$\min \left(\frac{-10 \sum_{i=1}^2 e^{-0.2 \sqrt{x_i^2 + x_{i+1}^2}}}{\sum_{i=1}^3 |x_i|^{0.8} + 5 \sin(x_i^3)} \right)$$

subject to the constraints

$$(x_1, x_2, x_3) \in [-5, 5] \times [-5, 5] \times [-5, 5] \subset \mathbb{R}^3$$

Here we used the following optimization parameters: 500 points are generated in step one, the interval $[-5, 5] \times [-5, 5] \times [-5, 5]$ is partitioned in $5 \times 5 \times 5$ subintervals and for each of them 1000 points are determined in step 3. The average runtime on the CPU was 111.30 s and on the GPU 5.93 s . Thus, a speed up of 18.77 was reached. This is due to the fact that in the different test runs only 5 or 6 intervals were processed in step 3, meaning that 9 or 8 multiprocessors were not used (but a more fine grained decomposition results in an excessive runtime for that problem). Furthermore the imbalance under the thread blocks is again very high. There was in average a thread block that had to process 43 % of all points. But the ratio of computations to memory accesses is again advantageous for the GPU computations. The CPU took in average 0.76 s for 1 million points (more than 5 times as much as for problem 2) and in average 1.5 comparisons are required for a point (2 times more like for problem 2). Another reason for the comparable good speed up for the third problem is, that a generated random point always fulfills the constraints, since the random points are already generated in that range. That reduces the diversity under the threads. Figure 3 shows plots of the image sets V resulting from the optimizations for all the three test examples.

IV. CONCLUSION

It could be shown that the described multiobjective optimization algorithm could be accelerated compared to a serial execution on the CPU by the usage of GPUs for a part of its computations. For the three test problems speed ups between 6 and 19 were reached for step 3 of the algorithm. The exact value highly depends on the optimization problem and on the decomposition of the domain. As higher the rate of needed computations compared to memory accesses, as higher the achievable speed up. Furthermore the workload of the intervals should be as balanced as possible. If it is possible to find a subdivision technique that guarantees this, one can gain more benefit from the usage of GPUs for this

Fig. 3. The image sets V resulting from the optimizations for test problem 1 to 3 (from left to right).

type of optimization. Another way to increase the speed up can be the employment of multiple GPUs for the optimization.

REFERENCES

- [1] C. A. C. Coello, "A comprehensive survey of evolutionary-based multiobjective optimization techniques," *Knowledge and Information Systems*, vol. 1, pp. 269–308, 1998.
- [2] J. Jahn, "Multiobjective Search Algorithm with Subdivision Technique," *Computational Optimization and Applications*, pp. 161–175, March 2006.
- [3] G. Chen, G. Li, S. Pei, and B. Wu, "High performance computing via a gpu," in *Information Science and Engineering (ICISE), 2009 1st International Conference on*, 2009, pp. 238–241.
- [4] J. Jahn, *Vector optimization: theory, applications, and extensions*. Berlin / Heidelberg: Springer, 2011.
- [5] M. Tanaka, H. Watanabe, Y. Furukawa, and T. Tanino, "Ga-based decision support system for multicriteria optimization," in *Systems, Man and Cybernetics, 1995. Intelligent Systems for the 21st Century., IEEE International Conference on*, vol. 2, Oct. 1995, pp. 1556–1561 vol.2.
- [6] F. Kursawe, "A variant of evolution strategies for vector optimization," in *Parallel Problem Solving from Nature. 1st Workshop, PPSN I, volume 496 of Lecture Notes in Computer Science*. Springer-Verlag, 1991, pp. 193–197.