Swarm and Evolutionary Computation ∎ (∎∎∎∎) ∎∎∎–∎∎∎

Contents lists available at SciVerse ScienceDirect

Swarm and Evolutionary Computation

journal homepage: www.elsevier.com/locate/swevo

Regular Paper

Performance investigations of genetic algorithms on graphics cards

Johannes Hofmann *, Steffen Limmer *, Dietmar Fey *

Department of Computer Science 3, Friedrich-Alexander-University Erlangen-Nuremberg, 91058 Erlangen, Germany

ARTICLE INFO

Article history:
Received 21 March 2012
Received in revised form
19 February 2013
Accepted 6 April 2013

Keywords:
GPU
Evolutionary computing
Genetic algorithm
Weierstrass function
Traveling salesman
Fermi

ABSTRACT

Genetic algorithms are one of the most adaptable optimization algorithms. Due to their inherent parallelism they seem well suited for the execution on massively parallel hardware such as graphics processing units. In this paper we put this claim to the test by performing comprehensive experiments. We try to find out how well graphics processing units are suited for the task and what parts of genetic algorithms should be executed on them. We focus especially on the new Fermi generation of Nvidia graphics chips. While it is imperative the fitness function be effectively parallelizable on the GPU, because it is the most computational expensive task of the algorithm, results indicate that if this is the case, speedups of several orders of magnitude are possible compared to conventional multi-core CPUs. Our findings also suggest that, starting with the Fermi architecture, all parts of a genetic algorithm should be carried out on the graphics card instead of only part of it.

1. Introduction

Evolutionary algorithms (EAs), especially genetic algorithms (GAs) [1,3], are a popular form of metaheuristic optimization algorithms. GAs maintain a population of *individuals*, also called candidate solutions. The internal representation of such a candidate solution is called a *chromosome* or *genotype*. The most popular representations are in form of strings of floats, integers or bits (the so-called *genes*). A *fitness function* is used to assign each individual a *fitness value*, indicating the quality of a certain candidate solution. In general, the algorithm repeats the following steps iteratively until a termination condition, for example reaching a fitness threshold, is met:

(i) Based on their quality, *parent individuals* are chosen from the population.
(ii) During *variation*, the *crossover* operator derives a number of *offspring* from the parent population; then, the resulting offspring undergo *mutation*, which introduces random change into their genotypes.
(iii) The newly created offspring are evaluated using the fitness function and, depending on the survivor selection strategy, replace less promising individuals to make up the *next generation*.

Although GAs are very efficient for some kind of optimization problems, they can still be resource-demanding for large problems, evoking the need for efficient parallelization. A vast number of different parallelization strategies can be found in the literature [4,24], the most intuitive of which is the master–worker model: a master instance runs a GA and manages the population as a whole. Whenever a (parallelizable) computational expensive task arises, the master divides this task and distributes the fractions to the workers. It could come to mind to parallelize every stage in the evolutionary cycle because most tasks can be done in parallel:

- Because solutions in the initial population are seeded independently from each other, the task can be performed in parallel.
- Each fitness evaluation takes only one individual into account, thus the fitness of the entire population can be computed in parallel.
- Since most selection schemes work probabilistically and perform random draws from the population, and even encourage multiple draws of the same individual, the draws can be done in parallel.
- Variation operators (crossover and mutation) work on a finite set of genotypes (two in the case of two-parent-crossover, one in the case of mutation) and there exist no dependencies between them, so they can be performed in parallel.

While the first implementations of parallel GAs were designed for multi-core CPUs, clusters, and grids (e.g. ParadisEO [5]), with today's graphics processing unit (GPU) there exists another parallel architecture, offering a high degree of parallelism and huge

* Corresponding authors. Tel.:+49 9131 85 27913.
E-mail addresses: johannes.hofmann@cs.fau.de (J. Hofmann), steffen.limmer@cs.fau.de (S. Limmer), dietmar.fey@cs.fau.de (D. Fey).

2210-6502/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.swevo.2013.04.003

amounts of computational power. As a consequence, it seems natural to employ GPUs for parallel GAs.

In 2005, Wong et al. deduced that GAs are only partly suited for GPUs [6]. This was concluded because the crossover operator examined did not operate on each gene independently, but on the whole chromosome, which could not be implemented efficiently on early GPU architectures.

In 2009, Maitre et al. decided to take another look at running GAs in the master–worker model on the GPU [7]. They found that a master–worker scheme, in which the GPU only evaluated the fitness function, had never been studied, perhaps due to the overhead of data transfer between CPU and GPU required by such a model. Their results, using benchmark functions and a real-world problem of material science, yielded a speedup of around 100 for their master–worker algorithm. They extended EASEA (EAsy Specification for Evolutionary Algorithms) [8], a language designed specifically to help non-expert programmers to make use of EAs, to produce code which can be directly compiled using the CUDA architecture. GPU-realizations of other GA parallelization schemes than the master–worker scheme can be found in the literature [9–11].

Most of the work up to now (e.g. [28–30]) focuses on implementing EAs on older GPUs; even using the previous generations' graphics hardware, some claim to achieve speedups higher than 1000 when comparing their GPU implementations to normal CPUs. Speedups of this magnitude can only be achieved by comparing optimized GPU code to poor CPU implementations. By examination of the existing work it is revealed that all use sequential implementations, which were in no way optimized to take advantage of modern multi-core CPUs (i.e. no multi-threading or vectorization).

The main contribution of our work consists firstly in an evaluation of the most recent Fermi GPU architecture and secondly in a fair comparison of this new architecture to recent multi-core CPUs by using an optimized CPU implementation.

In this work we present comprehensive experimental results to obtain answers to the following questions:

- Are GPUs suited for the master–worker scheme?
- How can GAs benefit from the current Fermi architecture of Nvidia GPUs in comparison to the previous G200 architecture?
- What speedup can be gained with the GPU in comparison to today's high-end multi-core CPUs?
- Which parts of the algorithm should be run on the GPU and which on the CPU?

For this purpose we ran experiments for two test problems. The first one is a mathematical optimization problem often used for benchmarking optimization algorithms: the minimization of the Weierstrass function. The second one is a combinatorial problem which has many real-world applications: the traveling salesman problem (TSP). We compare the performance of the individual parts of GAs for these two problems on the following architectures:

- AMD Opteron 2435 CPU using only one core.
- AMD Opteron 2435 CPU using all six cores.
- Intel Xeon X5650 CPU using only one core.
- Intel Xeon X5650 CPU using all six cores.
- Nvidia G200 GPU.
- Nvidia Fermi GPU.

This paper is organized as follows. The next section briefly describes the G200 and Fermi architectures of GPUs. In Sections 3 and 4, the experiments for the Weierstrass function and the TSP are explained and their results are presented respectively. The last section gives a conclusion of the work.

2. The G200 and Fermi GPUs

A G200 GPU consists of numerous cores, called stream processors (SP) or CUDA cores, that are physically grouped into the so-called streaming multiprocessors (SM).

SMs are SIMD devices, which means that all threads that are simultaneously scheduled on its SPs[1] are executing the same instruction; however, alternate program paths can be taken by different threads of a warp: considering a while loop which evaluates *true* to some part of the processors while *false* to the others, SPs which finished their loop have to wait for the others to complete. This behavior is called *divergence* and can introduce severe performance losses.

Each SM is equipped with 16 kB of fast local memory, which can be shared by all of its cores, 8 kB of texture as well as 8 kB of constant cache, and a separate program counter.

In our experiments we used the Tesla C1060 graphics card as representative for the G200 architecture. It consists of 240 SPs (30 SMs with 8 SPs, each).

One of the main problems of the G200 architecture was, despite the presence of a fast shared memory, that access to global memory was still very slow (about 200 clock cycles). The Fermi architecture solved this by introducing a full cache hierarchy to the GPU: each of the SMs now features a coherent L1 cache of configurable size, and together, the multiprocessors share an additional 768 kB L2 cache.

Each of the SMs now features 64 kB fast SRAM, which can be configured to either 48 kB L1 cache and 16 kB shared memory, or 16 kB L1 cache and 48 kB shared memory. Cache coherency is guaranteed on the multiprocessor level only, and SM caches are backed up by a total of 768 kB of L2 cache, shared and coherent across all multiprocessors. This L2 cache proves extremely useful for applications in which threads from several multiprocessors happen to access the same data (cf. tournament selection, population sorting).

While the introduction of caches will certainly serve to speed up applications, the fact that SRAM has been increased and can be configured will also benefit existing applications which rely on shared memory: because this memory is shared among all thread blocks residing on the same multiprocessor, the block count occupying an SM can be increased, thus improving the ability to hide slow global memory access by scheduling different blocks.

Another welcomed architectural advancement is the improvement of the double-precision performance, which is at the heart of the most high performance computing applications such as linear algebra or simulation. Each SM can now perform up to 16 double-precision fuse-multiply-and-add (FMA) operations per clock cycle, offering up to four times the performance of the G200.

Although one would assume that the availability of the new tools would have acted as a catalyst for evolutionary computing on the GPU, this was not the case. Up until now, there are only a very few publications on running EAs on the new hardware, indicating the benefits of the new architecture (like [13]). This is why we will focus our attention on this architecture in an effort to supplement the existing literature.

The GeForce GTX 480 was used as representative for the Fermi architecture. The card contains 15 SMs, each made up of 32 CUDA cores, resulting in a total of 480 cores (see Fig. 1).

[1] A group of threads that executes concurrently on a given SM is called a *warp*.

Please cite this article as: J. Hofmann, et al., Performance investigations of genetic algorithms on graphics cards, Swarm and Evolutionary Computation (2013), http://dx.doi.org/10.1016/j.swevo.2013.04.003