J. Parallel Distrib. Comput. 73 (2013) 86–100

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

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

Accelerated parallel genetic programming tree evaluation with OpenCL

Douglas A. Augusto[a,*], Helio J.C. Barbosa[a,b]

[a] *Laboratório Nacional de Computação Científica, Petrópolis, RJ, Brazil*

[b] *DCC/UFJF, Juiz de Fora, MG, Brazil*

ARTICLE INFO

Article history:
Received 19 April 2011
Received in revised form
26 December 2011
Accepted 24 January 2012
Available online 30 January 2012

Keywords:
OpenCL
GP–GPU
Parallel genetic programming
Accelerated tree evaluation

ABSTRACT

Inspired by the process of natural selection, genetic programming (GP) aims at automatically building arbitrarily complex computer programs. Being classified as an "embarrassingly" parallel technique, GP can theoretically scale up to tackle very diverse problems by increasingly adding computational power to its arsenal. With today's availability of many powerful parallel architectures, a challenge is to take advantage of all those heterogeneous compute devices in a portable and uniform way. This work proposes both (i) a transcription of existing GP parallelization strategies into the OpenCL programming platform; and (ii) a freely available implementation to evaluate its suitability for GP, by assessing the performance of parallel strategies on the CPU and GPU processors from different vendors. Benchmarks on the symbolic regression and data classification domains were performed. On the GPU we could achieve 13 billion node evaluations per second, delivering almost 10 times the throughput of a twelve-core CPU.

1. Introduction

Proposed as a technique to automatically build arbitrarily complex computer programs, genetic programming (GP) [19] is beyond doubt a highly ambitious paradigm from the field of evolutionary computation. Strictly speaking, there are only two requirements that need to be satisfied in order to enable GP as a potential solver for a certain problem: that (i) *the problem's solution could be described as a computer program*, and (ii) *there exists a graduated evaluation metric that given two candidate solutions is capable of indicating – with reasonable accuracy – which one is better*. The first condition says that candidate solutions must be represented on a GP system, hence being able to undergo sustained genetic operations, like crossover and mutation, which is the way programs evolve in GP. The latter allows for the implementation of the principle of natural selection, which is an essential component of every evolutionary computation algorithm.

It follows that, at least in theory, GP can be applicable not only to countless problems, but also to a large range of complexities. Although this fact emphasizes how versatile GP can be, there is a consequence: genetic programming is destined to be computationally demanding, whatever the availability of computational resources, because there will always be a more complex problem to be solved. This does not mean, however, that GP is an inefficient "processor-hungry" technique; being capable of tackling increasingly difficult problems as the computational power advances means that GP is able to take full advantage of an arbitrary amount of resources in order to handle a problem.

Most optimization algorithms can somehow take advantage of the availability of computational resources, but what distinguishes GP from most of them is that GP is able to efficiently explore a wider class of hardware architectures at their full extent. The main reason behind that is the high degree of parallelism endowed by GP, both fine- and coarse-grained [2].

1.1. Parallel architectures

The microprocessor industry players have agreed that keeping up with Moore's Law [23] is becoming increasingly difficult as the manufacturing process involved in making microchips approaches the physical limits of the technology [14]. In other words, unless some breakthrough in the field of manufacturing and materials is accomplished, there is not much room for further improvement on the transistor's density per area; moreover, even before the physical limits of the technology are effectively reached, the costs of producing denser and denser microprocessors may not be commercially viable.

An alternative approach to ensure continuous advance in the processor performance is to soften the focus on the transistor's density *per se* and instead employ many independent "not-so-dense" microchips coupled in a unique processor; that is, the parallel multi-core design. However, this approach does not improve

* Corresponding author.
E-mail addresses: douglas@lncc.br, daaugusto@gmail.com (D.A. Augusto), hcbm@lncc.br (H.J.C. Barbosa).

0743-7315/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2012.01.012

single-core performance, but rather the *total* performance, which is fully achieved when all processor cores operate, simultaneously, at full capacity.

The multi-core design has become ubiquitous in almost every computing platform and so has displaced the dominance of the serial single-core processors. Not merely that, but the demand for computational power has pressed forward all kinds of parallel architectures and has made them mainstream, the future trend.

According to [8], the variety of parallel processors can be categorized into two classes, namely the *latency*-and the *throughput*–oriented processors. The first class concerns those processors whose design is optimized towards the fast processing of sequential tasks, even for multi-core architectures. The second one refers to those processors in which the goal is to process the greatest amount of tasks per unit of time. A representative of the latency-oriented class is the ordinary CPU processor, be it single- or multi-core. As for the throughput-oriented ones, a noteworthy example is the Graphics Processing Unit (GPU) architecture.

1.2. Related work

One of the early attempts to make use of the GPUs to accelerate the execution of evolutionary algorithms was done by Yu et al. [33]. They implemented a parallel genetic algorithm on a Nvidia GeForce 6800GT GPU using the *Cg* language. Not only the fitness evaluation was implemented on the GPU, but also the genetic operators. Compared with the CPU, the authors have reported peak speedup of about 17 for the fitness evaluation. Harding and Banzhaf [11] implemented genetic programming on the GPU using the *Accelerator* toolkit. They reported excellent results on a Nvidia GeForce 7300 GO on different kinds of benchmarks. Langdon and Banzhaf [20] used the *Rapidmind* framework to implement a SIMD interpreter for linear genetic programming on the GPU. They achieved 895 million GPop/s on the Mackey–Glass chaotic time series using a Nvidia 8800 GTX.[1] Robilliard et al. [27,26] have implemented and evaluated different GP population-parallel approaches using the Nvidia's *Compute Unified Device Architecture* (CUDA). Their peak performance on a Nvidia G80 GPU was an impressive 2.8 billion GPop/s. Also using the CUDA framework, Maitre et al. [21] obtained a peak speedup of 250 when comparing the performance of a dual GPU Nvidia GTX-295 (using half of its cores) with an Intel quad-core Q8200 CPU in sequential mode.

Except for a few exceptions, such as the use of the proprietary GPU.NET [12], it seems that most of the recent studies exploiting the GPU power have been built exclusively around the CUDA technology. While CUDA is a powerful and mature toolkit, it is a closed technology and only works on Nvidia GPUs, precluding the use of all the computational power available from other vendors and architectures.

Although the related works found in the literature have shown the remarkable power of the GPU in speeding up the genetic programming execution using different frameworks, so far none of them have implemented and evaluated a parallel genetic programming using the OpenCL specification, which holds the following properties: (i) open standard; (ii) portable across many parallel architectures and vendors; and (iii) allows low-level access to hardware. One of the consequences is that there is neither references detailing how to map a parallel GP into the OpenCL framework nor studies directly comparing the accelerated parallel genetic programming performance on multi-core CPUs and GPUs from different vendors.

[1] GPop/s stands for *genetic programming operations per second*, which in turn usually means the average number of tree nodes evaluated in a second.

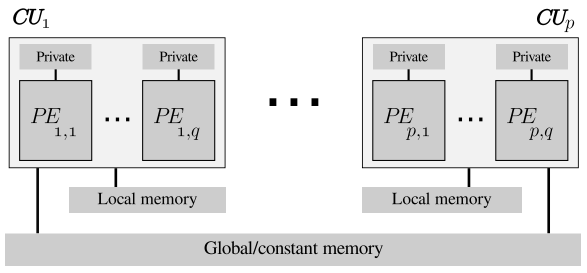

Fig. 1. Conceptual compute device architecture.

1.3. Text organization

This paper is organized as follows. Section 2 introduces OpenCL, a portable multi-vendor language for parallel programming on heterogeneous devices. Different strategies of parallel implementation of a genetic programming system using OpenCL for speeding up tree evaluation on both CPU and GPU devices are described in Section 3. In Section 4, computational experiments comparing raw performance in terms of GPop/s for different class of problems, parallel strategies, hardware architectures, and vendors are presented. Optimization techniques implemented by GPOCL are presented in Section 5. Finally, Section 6 points out conclusions and some directions for future work.

2. Open computing language—OpenCL

OpenCL is an open standard [31] for uniform and portable parallel programming across heterogeneous computing platforms. It aims to provide low and high level access to data- or task-parallel devices, either individually or simultaneously. Besides the conventional multi-core CPUs and GPUs from multiple vendors, OpenCL also supports some other parallel devices, such as Field-Programmable Gate Array (FPGA), digital signal processors (DSP), IBM's Cell Broadband engine [15], and more.

2.1. Programming model

The OpenCL's application program interface (API) exposes the access to the parallel compute devices by querying the *host* system to determine which *platforms* are present. A platform is a particular OpenCL implementation installed on the system. From the platform it is possible to discover which compatible computing *devices* are available and then select the ones according to some criteria defined by the programmer; a criterion, for instance, may be a particular architecture, like CPU or GPU. Given one or more devices, a *memory buffer* can be allocated on them, and a *command queue*, responsible for *kernel* submissions to the devices, can be set up. Roughly speaking, an OpenCL kernel is a C-like function designed to be executed in parallel on each individual processing element (e.g. CPU cores or GPU processors) of a compute device. The kernel has access to many levels of the device memory and can read from, process and write data to it.

2.2. Compute device abstraction

Fig. 1 illustrates a conceptual compute device architecture according to the OpenCL model, which is shared by all supported OpenCL devices. Each device has p compute units (CU), and each compute unit is composed of q processing elements (PE). The conventional CPU has only a single processing element ($q = 1$) per compute unit; p is the number of cores.

The compute device has a *global* memory, which is shared by all the device's processing elements. There is also an optimized global memory for read-only access, the *constant* memory. Each compute