Journal of Computer and System Sciences 81 (2015) 16–37

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Heterogeneous programming with Single Operation Multiple Data ☆

Hervé Paulino *, Eduardo Marques

CITI / Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2829-516 Caparica, Portugal

ARTICLE INFO

Article history:
Received 8 February 2013
Received in revised form 31 July 2013
Accepted 22 January 2014
Available online 27 June 2014

Keywords:
Data parallelism
Single Operation Multiple Data
Multi-cores
GPUs

ABSTRACT

Heterogeneity is omnipresent in today's commodity computational systems, which comprise at least one Central Processing Unit (CPU) and one Graphics Processing Unit (GPU). Nonetheless, all this computing power is not being harnessed in mainstream computing, as the programming of these systems entails many details of the underlying architecture and execution models. Current research on parallel programming is addressing these issues but, still, the system's heterogeneity is exposed at language level. This paper proposes a uniform framework, grounded on the Single Operation Multiple Data model, for the programming of such heterogeneous systems. We designed a simple extension of the Java programming language that embodies the model, and developed a compiler that generates code for both multi-core CPUs and GPUs. A performance evaluation attests that, despite being based on a simple programming model, the approach is able to deliver performance gains on par with hand-tuned data-parallel multi-threaded Java applications.

1. Introduction

The landscape of computing systems has altered in the last few years, with the shift from frequency to core scaling in CPU design, and the increasing popularity of General Purpose computation on GPUs (GPGPU) [1]. The architecture of current commodity computational systems is quite complex and heterogeneous, featuring a combination of, at least, one multi-core CPU and one GPU. This is further aggravated by the distinct nature of the architectural and execution models in place.

Therefore, the programming of these heterogeneous systems as a whole raises several challenges: which computations to run on each kind of processing unit; how to decompose a problem to fit the execution model of the target processing unit; how to map this decomposition in the system's complex memory hierarchy; among others. Tackling them efficiently requires true knowledge of parallel computing and computer architecture. However, mainstream software developers do not wish to deal with details, inherent to the underlying architecture, that are completely abstracted in classic sequential and even concurrent programming. Consequently, most of the available computing power is not really exploited.

The definition of high-level programming models for heterogeneous computing has been the driver of a considerable amount of recent research. Existing languages, such as X10 [2], Chapel [3], and StreamIt [4] have incorporated GPU support, and new languages, such as Lime [5], have been proposed altogether. Of these works, we are particularly interested in X10

☆ This work was partially funded by FCT-MEC in the framework of the PEst-OE/EEI/UI0527/2011 – Centro de Informática e Tecnologias da Informação (CITI/FCT/UNL) – 2011–2012.

* Corresponding author. Fax: +351 212948541.
E-mail addresses: herve.paulino@fct.unl.pt (H. Paulino), eduardo_raf@gmail.com (E. Marques).
URL: http://asc.di.fct.unl.pt/~herve (H. Paulino).

http://dx.doi.org/10.1016/j.jcss.2014.06.021
0022-0000/

and Chapel, since, to the best of our knowledge, they are the only to target heterogeneous systems at node and cluster level. However, as will be detailed in Section 2, their approach is not platform independent, exposing details of the target architecture at language level.

In this paper, we propose the use of the Single Operation Multiple Data (SOMD) model, presented in [6], to provide a uniform framework for the programming of this range of architectures. SOMD introduces the expression of data parallelism at subroutine level. The calling of a subroutine in this context spawns several tasks, each operating on a separate partition of the input dataset. These tasks are offloaded for parallel execution by multiple workers, and run in conformity to a variation of the Single Program Multiple Data (SPMD) execution model.

This approach provides a framework for the average programmer to express data parallel computations by annotating unaltered sequential subroutines, hence taking advantage of the parallel nature of the target hardware without having to program specialized code. In [6] we debated that this approach is viable for the programming of both shared and distributed memory architectures. In this extended version, we broad this claim to the GPU computing field, thus addressing the issue of heterogeneous computing. To this extent, our contributions are: i) a more detailed presentation of the SOMD execution model and a refinement of its programming model, featuring new constructs; ii) the conceptual realization of this execution model on shared memory architectures, distributed memory architectures, and GPU accelerated systems; iii) the effective compilation process for tackling heterogeneous computing, namely targeting multi-core CPUs and GPUs; iv) the evaluation of the code generated by our current prototype against hand-tuned data parallel multi-threaded applications.

The remainder of this paper is structured as follows: the next section provides a detailed motivation for expressing data parallelism at subroutine level, when compared to the usual loop-level parallelism. We also make evidence that approaches such as X10 and Chapel are bound to the underlying architecture. Section 3 overviews the SOMD execution and programming models. Section 4 presents our conceptual realization of the model on multiple architectures. Sections 5 and 6 describe, respectively, the compilation process for generating code for both shared memory multi-core CPUs and GPUs, and the required runtime support. Section 7 evaluates our prototype implementation from a performance perspective. Finally, Section 8 presents our concluding remarks.

2. Background and related work

Data-parallelism is traditionally expressed at loop-level in both shared and distributed memory programming. Regarding the former, OpenMP [7] is the most popular parallel computing framework. It provides a mix of compiler directives, library calls and environment variables, being that data-parallelism is expressed by annotating with directives the loops suitable for parallel execution. More recently, similar approaches have made their way into GPGPU. The most notorious example is OpenACC [8], a directive-based specification for offloading computation to GPUs and managing the associated data transfers.

Another representative system for shared-memory parallelism is Cilk [9], a C language extension that offers primitives to spawn and synchronize concurrent tasks (C functions). In Cilk, data-parallelism is expressed by programming a loop to spawn the desired number of tasks, each receiving as argument the boundaries of the subset of the problem's domain it will work upon. A C++ derivation of the extension was recently integrated in Intel's parallel programming toolkit [10]. This derivation adds some new features to the system, among which a special loop construct (`cilk_for`) for loop-level parallelism.

Intel Threading Building Blocks (TBB) [11] is a C++ template library that factorizes recurring parallel patterns. Data-parallelism in TBB may be expressed through the specialization of loop templates, supplying the task's body and, eventually, a partitioning strategy.

Loop-level data parallelism also prevails in distributed memory environments. di_pSystem [12] applied Cilk-like parallelism to distributed environments, providing a uniform interface for the programming of both shared and distributed memory architectures. For that purpose, the spawned tasks could be parametrized with the data to work upon and operate over explicitly managed shared variables. Single Program Multiple Data (SPMD) languages, such as UPC [13], require a special `forall` construct to express loops that work upon data distributed across multiple nodes. In the particular case of UPC (and Co-Array Fortran [14]) the Partitioned Global Addressing Space (PGAS) model provides the means for the programmer to explore the affinity between data and computation and hence reduce the communication overhead.

X10 [15] extends the PGAS model with notion of asynchronous activity, providing a framework for both task and data parallelism. Data distribution is performed at runtime upon a set of places (abstractions of network nodes). In this context, data parallelism is expressed through loops that iterate over this set of places, working only on the data residing locally at each place. Inter-place parallelism is expressed much in the same way as in the original Cilk. More recently, the language has also been extended to support the offload of computation to GPUs [2], which are presented as sub-places of the original place, the node hosting the GPU. This approach adequately exposes the isolation of the GPU's memory relatively to node's main memory. However, the X10 programming model is very imperative, forcing the programmer to be also aware of the underlying execution model. In order to be suitable for GPU execution an X10 asynchronous task must begin with two `for` loops: one to denote the distribution of the work per thread-groups, and a second to denote the distribution of the work within a group. Moreover, the enclosed code must fulfill several restrictions, such as the absence of method invocations.