Simulation Modelling Practice and Theory 58 (2015) 79–94

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

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

Static partitioning and mapping of kernel-based applications over modern heterogeneous architectures

J. Daniel García *, Rafael Sotomayor, Javier Fernández, Luis Miguel Sánchez

Computer Architecture Group, Computer Science and Engineering Department, Universidad Carlos III de Madrid, 28911 Leganés, Madrid, Spain

ARTICLE INFO

Keywords:
Parallel computing
Heterogeneous computing
Kernel partitioning

ABSTRACT

Heterogeneous architectures are being used extensively to improve system processing capabilities. Critical functions of each application (kernels) can be mapped to different computing devices (i.e. CPUs, GPGPUs, accelerators) to maximize performance. However, best performance can only be achieved if kernels are accurately mapped to the right device. Moreover, in some cases those kernels could be split and executed over several devices at the same time to maximize the use of compute resources on heterogeneous parallel architectures.

In this paper, we define a static partitioning model based on profiling information from previous executions. This model follows a quantitative model approach which computes the optimal match according to user-defined constraints.

We test different scenarios to evaluate our model: single kernel and multi-kernel applications. Experimental results show that our static partitioning model could increase performance of parallel applications by deploying not only different kernels over different devices but a single kernel over multiple devices. This allows to avoid having idle compute resources on heterogeneous platforms, as well as enhancing the overall performance.

1. Introduction

In recent years, the number of heterogeneous architectures in the top 500 list [1] has been increasing. They are mainly based on a combination of one or several GPGPUs and CPU cores. Those architectures consist of several computing devices that work together, where some are better fitted than others for specific classes of algorithms. The main computing device is a multi-core CPU well fitted for task parallelism, while graphics processing units (GPU) are currently very popular for most data parallel algorithms. However, in general, many applications do not make efficient use of available computing resources, which leads to a reduction of global efficiency for those parallel heterogeneous architectures [2].

The use of accelerators to improve performance of parallel programs is widespread. However, there are still some limitations in the use of those accelerators. For example, the memory-bound problem [3] refers to a general case where a code is limited by memory access [4]. Additionally, "*a data-transfer bottleneck emerges as data to be consumed and produced by the GPU must be transferred from the host CPU memory towards the GPU memory (and vice versa). Data transfers tend to become a performance bottleneck because the computing processors (CPU, GPU or accelerators) are increasing their throughput much faster than the bandwidth of the physical connections. This problem cancels out any gain obtained from the external accelerators (e.g. GPU, Xeon Phi)*" as noted in [22].

* Corresponding author.
E-mail address: josedaniel.garcia@uc3m.es (J.D. García).

http://dx.doi.org/10.1016/j.simpat.2015.05.010
1569-190X/

This problem cancels out any gain obtained from the external accelerators (e.g. GPU, Xeon Phi). Finally, not all algorithms fit well to GPGPU programming model [5], with the multi-core programming model being a better choice for certain parallel applications [6].

Open standards regarding parallel programming models have been proposed for developing new software over heterogeneous architectures, such as OpenACC [7] or the latest version of OpenMP (4.0 or higher) [8]. However, those standards are currently not fully supported by all hardware devices.

The Open Computing Language (OpenCL) [9] is a C-based programming model, used for different computing devices (e.g. CPUs, GPGPUs, DSP, FPGA, accelerators) that has become widely accepted and supported by major vendors. OpenCL is based on parallel code regions, called kernels, that could be executed on a device. OpenCL allows the development of heterogeneous parallel applications that could use more than one computing device, improving application efficiency.

Achieving an efficient mapping of the kernels onto the available computing devices is challenging due to the variation in characteristics and requirements of those kernels. The characteristics from each kernel (e.g. data input and output size, data access pattern or number of branches) and the features of each device result in different performances for different kernel-device combinations. Therefore, some applications include several versions of the same kernel in order to use the one that best fits the device at hand. Moreover, sometimes one kernel can be split into several sub-kernels that can be spread over a set of computing devices. This operation, called partitioning, can improve the performance obtained with only one computing device. This partitioning may affect different sections of the code (task-level) or one or more variables, which are split (over a loop, for instance) to obtain data-level parallelism.

The final goal is the generation of an efficient kernel mapping and partitioning based on a careful selection of the best version for each kernel, a correct management of the dependencies between the selected kernels, and the use of the profiling information about their expected performance.

This paper proposes a partitioning model that allows to describe the different schedules considering kernels, devices, input/output sizes and the relations between them. All these concepts are combined in a representation of the possible schedules for a given programs. The resulting representation is used in an algorithm that performs an off-line partitioning and scheduling of a set of kernels obtained from an application. The partitioning/scheduling is focused on maximizing the performance of the kernels executed in parallel, while observing the dependencies between them. Also, an execution algorithm is presented which allows the application to run the scheduling plan previously obtained.

The rest of the paper is organized as follows: Section 2 presents the scheduling model, the static off-line partitioning/scheduling proposed algorithm and the algorithm to run the execution plan; in Section 3 we show an evaluation of the proposal performed with single kernel and multi kernel examples; in Section 4 we review related work; and finally, in Section 5 we provide conclusions and outline future work.

2. Scheduling model and algorithm

2.1. Model overview

Parallel heterogeneous architectures are those where different computing devices are available (e.g. CPUs, GPUs, DSPs, FPGAs, and other accelerators). In this paper the study is restricted to single node computers, explicitly leaving out multi-node architectures (e.g. clusters). The ultimate objective is to be able to schedule the set of kernels from a codebase into the computing devices in the heterogeneous platform. To this end, a model has been created that allows to represent all different possible schedules. The model is based on four key aspects: kernels, input/output size, devices and transfer rates. Each pair of kernel and input/output size takes a certain time to run. Also related to the data size is the transfer rate. Lastly, each device has its own strengths and limitations, and as such their performance will vary from kernel to kernel.

For the purposes of this study, the combination of a kernel and an specific input size is named an *execution unit*. Thus, a given kernel will be considered a different execution unit when run with an input size of 256 or 512 bytes.

There are two important relationships among execution units: *incompatibility* and *dependency*. Two execution units are *incompatible* if both cannot be executed together on the same application run. The idea behind this is that the algorithm is able to consider different versions of the source code, selecting the best one for each device. For example, it may decide between executing one large execution unit or several smaller execution units when considering the same section of code, or whether to run the outermost or innermost loop as an execution unit. One execution unit *depends* on another if it needs to wait until the latter has finished. This categorization allows to detect independent execution units, which can be run in parallel. Ultimately, this will allow to differentiate between valid and invalid sequences of execution, improving the computation of the best schedule accordingly.

Another measure of interest is the *feasibility* of deploying an execution unit on a specific device. One execution unit is *feasible* for one device if it can be executed on that device. This is required to decide which is the best device for each execution unit considering only valid devices for each execution unit. For example, any execution unit that performs system calls shall be considered unfeasible for a GPGPU device. An execution unit with a sufficiently large input data may be unfeasible for a device with limited memory.

The previous concepts are formalized as follows:

Let E be set of execution units $E = \{e_1, \ldots, e_l\}$, D the set of devices $D = \{d_1, \ldots, d_n\}$ and R the set of execution restrictions $R = \{r_1, \ldots, r_n\}$.