Parallel Computing 52 (2016) 65–77

Contents lists available at ScienceDirect

Parallel Computing

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

Writing a performance-portable matrix multiplication

Jorge F. Fabeiro, Diego Andrade*, Basilio B. Fraguela

Computer Architecture Group, Universidade da Coruña, Spain

ARTICLE INFO

Article history:
Received 27 July 2015
Revised 22 December 2015
Accepted 28 December 2015
Available online 4 January 2016

Keywords:
GPGPU
Heterogeneous systems
OpenCL
Performance portability
Embedded languages

ABSTRACT

There are several frameworks that, while providing functional portability of code across different platforms, do not automatically provide performance portability. As a consequence, programmers have to hand-tune the kernel codes for each device. The Heterogeneous Programming Library (HPL) is one of these libraries, but it has the interesting feature that the kernel codes, which implement the computation to be performed, are generated at run-time. This run-time code generation (RTCG) capability can be used, in conjunction with generic parameterized algorithms, to write performance-portable codes. In this paper we explain how these techniques can be applied to a matrix multiplication algorithm. The performance of our implementation is compared to two state-of-the-art adaptive implementations, clBLAS and ViennaCL, on four different platforms, achieving average speedups with respect to them of 1.74 and 1.44, respectively.

1. Introduction

Performance portability is an open problem in heterogeneous systems. As a consequence, programmers usually have to hand-tune the code of a given algorithm for each platform where it will be executed in order to maximize its performance [1–3]. The Heterogeneous Programming Library (HPL) [4] is a C++ framework that simplifies the portable programming of heterogeneous systems. This library puts emphasis on improving the programmability of these systems by providing an interface that is noticeably simpler than other alternatives like OpenCL [5].

The library provides a programming model similar to OpenCL, where a kernel, which expresses the parallel computation, is spawned to a given device generating several threads. In fact, the current backend of this library is built on top of OpenCL. An interesting characteristic of the library is that the kernel code is translated into OpenCL at run-time. This run-time code generation (RTCG) capability can be used to adapt the code to the properties of the computing device where the code is going to be executed, since they are known at run-time.

This work illustrates the creation of performance-portable HPL kernels. These kernels receive a set of optimization parameters that are used inside the kernel to guide RTCG and generic optimization techniques. For example, RTCG can be used to unroll a loop, being the input optimization parameter the unroll factor. Another example would be a tiling transformation, a generic tile size being the input parameter of this optimization. Our approach complements these RTCG kernels with a genetic algorithm that chooses the values of the input optimization parameters guided by the execution time of the versions generated at run-time.

The usage of some of these techniques has been shown and evaluated in [6]. The current work focuses on the matrix multiplication code, improving [6] in several points: (1) two new techniques, vectorization and instruction scheduling, are

* Corresponding author. Tel.: +34 881016002 fax: +34 981167160.
E-mail address: diego.andrade@udc.es (D. Andrade).

http://dx.doi.org/10.1016/j.parco.2015.12.005
0167-8191/

applied to generate performance-portable kernels, which generates a best-kernel 12 times faster than in [6] (2) the illegal combinations of parameters are discarded during the generation phase of the genetic algorithm, which reduces the search time on average 2.57 times with respect to [6], and (3) the performance of our kernels is compared to two state-of-the-art adaptive implementations, clBLAS [10] and ViennaCL [14]. These two implementations were chosen because (a) they use OpenCL, and thus, they target the same range of platforms as HPL, and (b) they provide adaptive mechanisms to enable performance portability. Our study also covers the OpenCL-based clMAGMA library [7], as it relies on clBLAS for its OpenCL BLAS routines.

Our matrix multiplication implementation is based on existing implementations for NVIDIA GPUs [8], AMD GPUs [9], and any kind of devices supporting OpenCL [10]. This latter implementation also enables performance portability. Our implementation uses not only similar techniques to those introduced in these previous works but also new ones. As a consequence, our implementation turns out to be more effective than those previous ones.

The rest of this paper is organized as follows. Section 2 introduces the basic concepts of the Heterogeneous Programming Library (HPL). Then, Section 3 summarizes the new optimization techniques introduced in this paper with respect to [6]. Section 4 explains the implementation details of our matrix multiplication, and how a genetic search is used to tune its parameters. This is followed by the experimental results in Section 5 and a discussion of related work in Section 6. Finally, Section 7 is devoted to our conclusions and future work.

2. The Heterogeneous Programming Library library

The Heterogeneous Programming Library (HPL), available for download under GPL license at http://hpl.des.udc.es , improves the programmability of heterogeneous systems. Codes written using HPL can be executed across a wide range of devices. In addition, programmers can exploit performance portability on top of HPL using its run-time code generation (RTCG) mechanism. This mechanism is present in HPL kernels, which are written using a language embedded in C++. This code is executed at run-time and it translates the HPL computational kernel into the HPL's intermediate representation (IR), currently OpenCL.

The HPL library supports the same programming model as CUDA and OpenCL. The hardware model is composed of a standard CPU host with a number of computing devices attached. The host runs the sequential parts of the code and it dispatches the parallel parts, which are codified as HPL kernels, to the devices. The CPU of the host can be itself a computing device. Devices are composed of a number of processors that execute SPMD parallel code on data present in the memory of their device. As kernels can only work with data available in the devices, data must be transferred between host and devices, but this process is totally automated by the library.

Several instances of each kernel are executed as threads and they are univocally identified using a tuple of non-negative integers, called global identifiers. These identifiers, and their associated threads, form a global domain with up to 3 dimensions. In turn, these threads can be associated in groups. With this purpose, local domains can be defined as equal portions of the global domain. Threads inside a group are also identified using tuples of local identifiers and they can be synchronized through barriers and share a small scratchpad memory.

The memory model distinguishes four types of memory regions in the devices (from largest to smallest): (1) the global memory, which is read/written and shared by all the processors, (2) the local memory, which is a read/write scratchpad shared by all the processors in a group, (3) the constant memory, which is a read-only memory for the device processors and can be set up by the host, and (4) the private memory, which is only accessible within each thread.

Programmers using HPL have to write a code to be executed in the host, and one or several kernel codes, which will be dispatched to the devices. To do that, the library provides three main components: the host API, the template class `Array` and the kernels. They are now explained in its turn.

The host API. The most important component of this API is the `eval` function, which requests the execution of a kernel `f` with the syntax `eval(f)(arg1, arg2,...)`. The execution of the kernel can be parameterized by inserting methods calls between `eval` and the argument list. For example, by default, the global size is equal to the size of the first argument, whereas the local size is automatically selected by the library. Yet, this default behavior can be overridden by specifying alternative global and local sizes, using methods called `global` and `local` respectively. This way, if we want to define a 200×400 global domain divided into 2×4 local domains, the function `eval` should be invoked as follows `eval(f).global(200, 400).local(2, 4)(a, b)`. Listing 1 contains an HPL implementation of a matrix-vector product. The main procedure of this code contains an example host code for a matrix-vector product, where a global domain of M threads and local domains grouping 10 threads each are defined.

The template class `Array`. The variables used in a kernel must have type `Array<type, ndim [, memoryFlag]>`. This type represents an n-dimensional array of elements of a C++ `type`, or a scalar for `ndim`=0. Scalars and vectors can also be defined with special data types like `Int`, `Float`, `Int4`, `Float8`, etc. The `Array` optional `memoryFlag` either specifies one of the kinds of memory supported (`Global`, `Local`, `Constant` or `Private`). The default value of the `memoryFlag` is `Global`, the exception being the `Arrays` declared inside the body of kernels, which are placed by default in `Private` memory. The elements that compose an array may be any of the usual C++ arithmetic types or a struct. The arrays passed as parameters to the kernels must be declared in the host using the same syntax. These variables are initially stored in the