Computer Physics Communications 207 (2016) 83–90

Contents lists available at ScienceDirect

Computer Physics Communications

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

The Dynamic Kernel Scheduler—Part 1

Andreas Adelmann [a,*], Uldis Locans [a,b], Andreas Suter [a]

[a] Paul Scherrer Institut, Villigen, CH-5232, Switzerland
[b] University of Latvia, 19 Raina Blvd., Riga, LV 1586, Latvia

ARTICLE INFO

Article history:
Received 30 October 2015
Received in revised form
6 April 2016
Accepted 16 May 2016
Available online 26 May 2016

Keywords:
GPU
CUDA
Intel MIC
FFT
Monte Carlo
OPAL
μSR

ABSTRACT

Emerging processor architectures such as GPUs and Intel MICs provide a huge performance potential for high performance computing. However developing software that uses these hardware accelerators introduces additional challenges for the developer. These challenges may include exposing increased parallelism, handling different hardware designs, and using multiple development frameworks in order to utilise devices from different vendors.

The Dynamic Kernel Scheduler (DKS) is being developed in order to provide a software layer between the host application and different hardware accelerators. DKS handles the communication between the host and the device, schedules task execution, and provides a library of built-in algorithms. Algorithms available in the DKS library will be written in CUDA, OpenCL, and OpenMP. Depending on the available hardware, the DKS can select the appropriate implementation of the algorithm.

The first DKS version was created using CUDA for the Nvidia GPUs and OpenMP for Intel MIC. DKS was further integrated into OPAL (Object-oriented Parallel Accelerator Library) in order to speed up a parallel FFT based Poisson solver and Monte Carlo simulations for particle–matter interaction used for proton therapy degrader modelling. DKS was also used together with Minuit2 for parameter fitting, where χ^2 and max-log-likelihood functions were offloaded to the hardware accelerator. The concepts of the DKS, first results, and plans for the future will be shown in this paper.

1. Introduction

In recent years hardware accelerators have become increasingly popular within scientific computing. Based on the Top500 list from June 2015 [1], 90 of the top 500 supercomputers in the world are accelerator based. This includes the top two systems on the list: Tianhe-2 which uses Intel Xeon Phi coprocessors and Titan which uses NVIDIA K20x GPUs. GPU usage for general purpose computing has become even more important, due to the gaming industry. Almost every computer is now equipped with a GPU, but if the application is not exploiting the GPU, it is not using all the available computational power of the system. However, developing software that can take advantage of hardware accelerators can become a challenging task, especially for large existing applications. Each hardware accelerator has its own architecture and memory hierarchy, which must be taken into account to gain the maximum performance out of the device. In addition to hardware differences, there are also varying methods to program these devices. NVIDIA provides the CUDA [2] toolkit for its GPUs, both AMD and NVIDIA support the OpenCL [3] framework, and Intel allows usage of standard tools and languages to program Intel MIC processor [4], but parallelisation and vectorisation of the code is needed to gain the best performance. There are also OpenACC [5] and OpenMP [6] standards that allow the targeting of hardware accelerators by expressing parallelism through compiler directives.

In this work, the Dynamic Kernel Scheduler (DKS) is presented which provides a slim software layer between the host application and the hardware accelerators. DKS separates the accelerator and framework specific code from the host application and provides a simple interface that can be implemented in the host application to offload tasks to the accelerator. DKS provides functions to handle communication and data transfer between host and device, as well as a library of functions written in CUDA, OpenCL, and OpenMP that allow the targeting of different accelerators. The first version of DKS was integrated into OPAL (Object-oriented Parallel Accelerator Library). This DKS version uses CUDA kernels and OpenMP offload pragmas to run OPAL's FFT based Poisson solver and Monte Carlo simulations on a GPU and Intel MIC. DKS was also used together with Minuit2 for parameter fitting, where χ^2

* Corresponding author.
E-mail address: andreas.adelmann@psi.ch (A. Adelmann).

http://dx.doi.org/10.1016/j.cpc.2016.05.013
0010-4655/

and max-log-likelihood functions were offloaded to the hardware accelerator.

In the literature, there are several FFT Based Poisson solvers developed for GPUs using CUDA which use NVIDIA's cuFFT library [7,8]. One can also find research on the use of customised FFTs for asynchronous execution and mapping FFT based Poisson solvers to multi node systems [9–11]. Numerous studies [12–17] have been carried out to show the potential of GPUs and Intel Xeon Phi co-processors for Monte-Carlo simulations for proton and photon transport. These problems are some of the most time consuming parts of the OPAL simulations, and previous research shows that they are good candidates for acceleration on the co-processors.

Many research projects try to focus on improving programmability of hardware accelerators. Apart from compiler directive based approaches, there are a number of vendor supported libraries [18,19] that allow the simplification of offloading specific tasks to accelerators. There has also been work on creating abstractions and providing software layers that would allow to express kernels [20–22] for hardware accelerators, which can be translated to CUDA or OpenCL code that is run on the device.

The ability of DKS to have implementations using different frameworks and libraries, and switch between them from the host application allows the targeting of hardware accelerators of different types and fine tuning of the code to gain the maximum performance from each device. This approach also provides more portability and software investment protection for the host application. In case some hardware architecture is no longer manufactured, or some new architecture or development framework emerges, only DKS needs to be updated.

The rest of the paper is structured as follows: Section 2 describes the concepts and architecture of DKS; Section 3 describes the used hardware; Section 4 describes the concepts of OPAL's FFT based Poisson solver and Monte-Carlo type particle–matter interaction simulations as well as DKS implementation of these functions and the benchmark results; Section 5 explains the DKS and Minuit2 usage and results for parameter fitting using hardware acceleration; and Section 6 provides conclusions and future of the DKS.

2. Concept and architecture of the Dynamic Kernel Scheduler (DKS)

2.1. Concept

The Dynamic Kernel Scheduler (DKS) is a slim software layer between the host application and the hardware accelerator, as depicted in Fig. 1. The aim of the DKS is to allow the creation of fast fine tuned kernels using device specific frameworks such as CUDA, OpenCL, OpenACC and OpenMP. On top of that, DKS allows the easy use of these kernels in host applications without providing any device or framework specific details. This approach facilitates the integration of different types of devices in the existing applications with minimal code changes and makes the device and the host code a lot more manageable.

The architecture of DKS can be split into three main parts:

1. The first part provides communication functions that handle memory allocation and data transfer to, and from, the device. All the memory management is left up to the user. So that the data transfers and memory allocation can be scheduled only when necessary. DKS also supports GPU streams such that asynchronous data transfer and kernel execution can be implemented when possible.

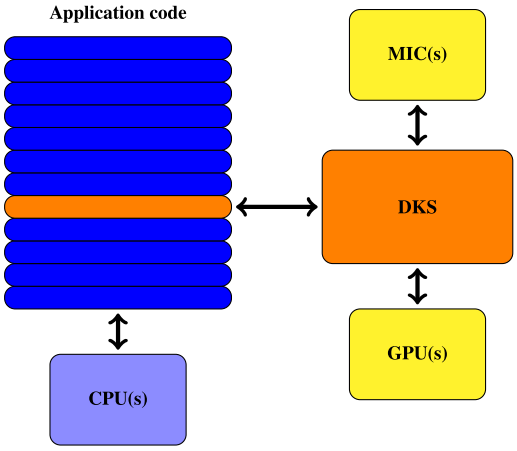

Fig. 1. The Dynamic Kernel Scheduler concept.

2. The second part of DKS consists of a function library, which contains algorithms written in CUDA, OpenCL, and OpenMP to target different devices. DKS can switch between implementations based on the hardware that is available. Writing functions using multiple frameworks results in extra work, but this provides the opportunity to fine tune kernels for each device architecture for maximum performance. That also allows the targeting of systems containing different types of devices. The different implementations of the code are always separated so the code is still easy to manage. Additionally if a host application is targeted at a specific system, implementations that are not needed can be omitted.
3. The third part of DKS is the auto-tuning functionality which will be discussed in a forthcoming paper. The aim of auto-tuning is to select the appropriate implementation of the algorithm and change the launch parameters according to the devices that are available on the system in order to gain the maximum performance. The auto-tuning functionality relies on knowledge of device architecture and benchmark tests that can be run on the system before running the application.

Fig. 2 shows an example code of DKS usage inside a host application to perform Fast Fourier transform. The host application has full control over the memory allocation and data transfer to the device, but there are no device specific details in the host code. DKS evaluates the calls made by host application and chooses the appropriate device to use, and algorithm implementation, to run the code on selected accelerator.

2.2. Architecture

The Dynamic Kernel Scheduler is split into separate modules. Each module contains function implementations using different frameworks. The base class for each module contains functions which handles the device management, memory management, and data transfer, this base class can be extended to cover all the necessary algorithm specific functions. The base class of DKS receives all the calls from the host application and decides which device specific implementation should be used to run the code on the device. Fig. 3 shows the architecture of the first version of DKS, for each module base class can be easily extended to include other algorithms and the base class of DKS can be extended to include other modules to handle different development frameworks.

3. Accelerator hardware

The development and tests of DKS were performed using Nvidia GPUs and Intel Xeon Phi accelerator. The features of all