MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Framework for Parallel Kernels Autotuning

MASTER'S THESIS

Bc. Filip Petrovič

Brno, Spring 2018

Declaration

Bc. Filip Petrovič

Advisor: RNDr. Jiří Filipovič, Ph.D.

Acknowledgement

I would like to thank my supervisor Jiří Filipovič for his help and valuable advice, David Střelák and Jana Pazúriková for their feedback and work on code examples. I would also like to thank my family for their support during my work on the thesis.

Abstract

The result of this thesis is a framework for autotuning of parallel kernels which are written in either OpenCL or CUDA language. The framework includes advanced functionality such as support for composite kernels and online autotuning. The thesis describes API and internal structure of the framework and presents several examples of its utilization for kernel optimization.

Keywords

autotuning, parallel programming, OpenCL, CUDA, kernel, optimization

Contents

1 Introduction

In recent years, acceleration of complex computations using multi-core processors, graphics cards and other types of accelerators has become much more common. Currently, there are many devices developed by multiple vendors which differ in hardware architecture, performance and other attributes. In order to ensure portability of code written for particular device, several software APIs (application programming interfaces) such as OpenCL (Open Computing Language) or CUDA (Compute Unified Device Architecture) were designed. Code written in these APIs can be run on various devices while producing the same result. However, there is a problem with portability of performance due to different hardware characteristics of these devices. For example, code which was optimized for a GPU may run poorly on a regular multi-core processor. The problem may also exist among devices developed by the same vendor, even if they have comparable parameters and theoretical performance.

A costly solution to this problem is to manually optimize code for each utilized device. This has several significant disadvantages, such as a necessity to dedicate large amount of resources to write different versions of code and test which one performs best on a given device. Furthermore, new devices are released frequently and in order to efficiently utilize their capabilities, it is often necessary to rewrite old versions of code and repeat the optimization process again.

An alternative solution is a technique called autotuning, where code includes parameters which affect performance depending on their value, for example a parameter which affects length of a vector type of a particular variable. Optimal values of these parameters might differ for various devices based on their hardware capabilities. Parametrized code is then launched repeatedly using different combinations of parameters to find out the best configuration for a particular device.

In order to make autotuning easier to implement in applications, several frameworks were created. However, large number of these are focused only on a very small subset of computations. There are some frameworks which are more general, but their features are limited and usually only support simple usage scenarios. The aim of this thesis was to develop autotuning framework which would support more complex use cases, such as situations where computation is split into several smaller programs. Additionally, the framework should be written in a way which would allow its easy integration into existing software and possibly combine autotuning with regular computation.

The thesis is split into four main chapters. <Todo: add short description of each chapter.>

2 Compute APIs and autotuning

This chapter serves as an introduction to autotuning technique and includes description of compute APIs which are utilized by KTT (Kernel Tuning Toolkit)[1] – OpenCL and CUDA. Because both APIs provide relatively similar functionality, only OpenCL is described here in greater detail. Section about CUDA is mostly focused on explaining features which differ from OpenCL. It is worth mentioning that CUDA actually consists of two different APIs – low-level driver API and high-level runtime API built on top of the driver API. For the purpose of this thesis, only CUDA driver API will be further described, because the runtime API lacks features which are necessary to implement autotuning in CUDA.

2.1 OpenCL

OpenCL is an API for developing primarily parallel applications which can be run on a range of different devices such as CPUs, GPUs and FPGAs (field-programmable gate arrays). An OpenCL application consists of two main parts. First part is a host program, which is typically executed on a CPU and is responsible for OpenCL device configuration, memory management and launching of kernels. Second part is a kernel, which is a function executed on an OpenCL device and usually contains major part of a computation. Kernels are written in OpenCL C which is based on C programming language.

2.1.1 Host program in OpenCL

Host program is written in a regular programming language, typically in C or C++. Its main objective is to successfully launch a kernel function. OpenCL API defines several important structures which are referenced from host program:

- *cl_platform* – References an OpenCL platform.
- *cl_device* – References an OpenCL device, which is used during context initialization.

1. Name of autotuning framework developed as a part of the thesis.

- *cl_context* – Serves as a holder of resources, similar in functionality to an operating system process. Majority of other OpenCL structures have to be tied to a specific context. Context is created for one or more OpenCL devices.

- *cl_command_queue* – All commands which are executed directly on an OpenCL device have to be submitted inside a command queue. It is possible to initialize multiple command queues within a single context in order to overlap independent asynchronous operations.

- *cl_mem* – Data which is directly accessed by kernel has to be bound to an OpenCL buffer, this includes both scalar and vector arguments. It is possible to specify buffer memory location (device or host memory) and access type (read-only, read-write, write-only).

- *cl_program* – A variable which references OpenCL program compiled from OpenCL C source file. Program can be shared by multiple kernel objects.

- *cl_kernel* – An object used to reference a specific kernel. Consists of an OpenCL program, kernel function name (single program can contain definitions of multiple kernel functions) and buffers which are utilized by the kernel.

- *cl_event* – Serves as a synchronization primitive for individual commands submitted to an OpenCL device. Can be used to retrieve information about the corresponding command, such as status or execution duration.

Execution of an entire OpenCL application then typically consists of the following steps:

- selection of target platform (e.g., AMD, Intel, Nvidia) and device (e.g., Intel Core i5-4690, Nvidia GeForce GTX 970)

- initialization of OpenCL context and one or more command queues

- initialization of OpenCL buffers (either in host or dedicated device memory)
- compilation and execution of kernel function
- retrieval of data produced by kernel from OpenCL buffers into host memory (if data is located in dedicated device memory)

2.1.2 Kernel in OpenCL

Code in a kernel source file is written from perspective of a single *work-item*, which is the smallest OpenCL execution unit. Each work-item has its own *private memory* (memory which is mapped to e.g., CPU or GPU register).

Work-items are organized into a larger structure called *work-group*, from which they all have access to *local memory* (eg. CPU cache, GPU shared memory). Work-group is executed on a single *compute unit* (e.g., CPU core, GPU streaming multiprocessor). It is possible for multiple work-groups to be executed on the same compute unit. OpenCL work-group can have up to three dimensions. Number and size of dimensions affects work-item indexing inside work-group.

Individual work-groups are organized into *NDRange* (N-Dimensional Range). At NDRange level, it is possible to address two types of memory – *global memory* and *constant memory*. Global memory (e.g., CPU main memory, GPU global memory) is usually very large but has high latency. On the other hand, constant memory generally has small capacity but lower latency. It can be utilized to store read-only data. Organization and indexing of work-groups inside NDRange works in the same way as for work-items inside work-group.

Hierarchical organization into NDRange, work-groups and work-items allows for more intuitive mapping of computation tasks onto OpenCL kernels. Tasks are defined at the NDRange level, work-groups represent large chunks of a task which are executed in arbitrary order. The smallest operations (e.g. addition of two numbers) are mapped onto work-items.

Figure 2.1 contains a simple OpenCL kernel, which adds up elements from arrays *a* and *b*, then stores the result in array *c*. Qualifier *__global* specified for the arguments means that they are stored in

```
__kernel void vectorAddition(__global float* a, __global
    float* b, __global float* c)
{
    int i = get_global_id(0);
    c[i] = a[i] + b[i];
}
```

Figure 2.1: Vector addition in OpenCL.

global memory. Function *get_global_id(int)* is used to retrieve work-item index unique for the entire NDRange in specified dimension.

2.2 CUDA, comparison with OpenCL

CUDA is a parallel compute API developed by Nvidia Corporation. It works similarly to OpenCL, but there are also several differences which played an important role during the framework development:

- CUDA is officially supported only on graphics cards released by Nvidia Corporation
- Differences in terminology, same concepts have different terms in OpenCL and CUDA
- Global indexing (i.e., NDRange indexing in OpenCL) works differently in CUDA

Table 2.1 contains terms used in OpenCL and their counterparts in CUDA.

<Todo: Note on indexing differences>

2.3 Autotuning in compute APIs

Todo...

OpenCL term	CUDA term
work-item	thread
work-group	thread block
NDRange	grid
compute unit	streaming multiprocessor
global memory	global memory
constant memory	constant memory
local memory	shared memory
private memory	local memory
cl_platform	N/A
cl_device_id	CUdevice
cl_context	CUcontext
cl_command_queue	CUstream
cl_mem	CUdeviceptr
cl_program	nvrtcProgram / CUmodule
cl_kernel	CUfunction
cl_event	CUevent

Table 2.1: Comparison between OpenCL and CUDA terminology.

3 Conclusion

Todo...

Bibliography

[1] Cedric Nugteren and Valeriu Codreanu. "CLTune: A Generic Auto-Tuner for OpenCL Kernels". In: *MCSoC: 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*. 2015.

[2] Khronos OpenCL Working Group. *The OpenCL 1.2 Specification*. 2012. URL: `https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf` (visited on 25/2/2018).

A Appendix

Todo...