Masaryk University
Faculty of Informatics

Framework for Parallel Kernels Autotuning

Master's Thesis

Bc. Filip Petrovič

Brno, Spring 2018

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Filip Petrovič

Advisor: RNDr. Jiří Filipovič, Ph.D.

Acknowledgements

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 **1**

2 Compute APIs and autotuning **3**
2.1 *OpenCL* 3
2.1.1 Host program in OpenCL 3
2.1.2 Kernel in OpenCL 5
2.2 *CUDA, comparison with OpenCL* 6
2.3 *Possibilities for autotuning in compute APIs* 7
2.3.1 Work-group (thread block) dimensions 8
2.3.2 Usage of vector data types 8
2.3.3 Data placement in different types of memory . . 8

3 Comparison of autotuning frameworks **9**
3.1 *CLTune* 9
3.2 *Kernel Tuner* 11
3.3 *OpenTuner* 12

4 KTT framework **14**

5 Conclusion **15**

Bibliography **16**

A Appendix **17**

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 program code contains 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 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 five 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, identical or similar 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. Due to several differences in design, some terms do not have an equivalent term in the other API.

Difference in global indexing plays an important role during addition of tuning parameters which affect either grid dimensions or block dimensions. In OpenCL, the NDRange size is specified as total number of threads in a dimension. However, in CUDA the grid size is specified as number of threads in a dimension divided by number of blocks in that dimension. This would be rather inconvenient for porting autotuned programs from one API to the other. The problem will be further elaborated upon in chapter 4.

OpenCL term	CUDA term
compute unit	streaming multiprocessor
NDRange	grid
work-group	thread block
work-item	thread
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.

2.3 Possibilities for autotuning in compute APIs

Design of previously described APIs allows for a wide range of optimization opportunities, both inside kernel and host code. These optimizations can be implemented with usage of tuning parameters. While some of the parameters can be utilized only in limited range of applications, there are also several ones which can be implemented in larger number of computation tasks. This section provides a list of some of the most common parameters and explanation why they are used for autotuning.

2.3.1 Work-group (thread block) dimensions

Work-group dimensions specify how many work-items are included in a single work-group. Work-groups are executed on compute units, which are mapped onto, for example CPU cores or GPU multiprocessors. Performance of these devices may be vastly different and manually finding an optimal work-group size is difficult, which is a reason why this parameter usually makes an ideal candidate for autotuning.

2.3.2 Usage of vector data types

Modern processors contain vector registers that allow concurrent execution of a single instruction over multiple data which leads to a significant speed-up of certain types of computations. Kernel compilers attempt to automatically utilize these registers in order to speed up computation without manual code modification. However, automatic vectorization is not always optimal. There is an option to perform manual vectorization by using vector data types which are available in both OpenCL and CUDA. It is possible to control vector length with a tuning parameter, e.g., by using type aliases.

2.3.3 Data placement in different types of memory

Section 2.1.2 described various memory types available in OpenCL, similar memory hierarchy can also be found in CUDA. In many cases, there are more valid memory types to choose from for data placement. The choice can have an effect on performance, for example accessing data from OpenCL local memory is usually faster than using global memory. The problem is that local memory capacity is limited and while on certain devices the data could fit into it, on other devices it would be necessary to use global memory instead. Having a single version of kernel which would utilize only global memory would be inefficient for large number of devices. This can be solved by using tuning parameter which controls the data memory placement.

3 Comparison of autotuning frameworks

This chapter presents several generic autotuning frameworks for parallel kernels. Each section describes one framework, its advantages, disadvantages and an example of its usage. Frameworks which are not publicly available or focus only on a specific subset of computations are not discussed here. The final section of this chapter includes motivation for development of KTT framework.

3.1 CLTune

CLTune [1] is a framework for autotuning of OpenCL and CUDA kernels. It is freely available in form of a library and provides C++ interface for writing host programs. It is relatively easy to use and provides capabilities for tuning of single kernels, multiple configuration search strategies and result validation in a form of reference kernels.

However, it also has several limitations. Among the most significant ones are lack of support for composite kernels, limited argument handling options and inability to validate kernel results with C or C++ function. The framework is no longer actively developed, so it is unlikely that new features will be introduced.

Basic tuner configuration in CLTune consists of several main steps, which are listed below. KTT functionality, in its simplest form, is based on the same idea. Figure 3.1 contains part of a program written in CLTune, which includes all the main steps.

1. Initialization of tuner by specifying target platform and device.
2. Addition of tuned kernel.
3. Addition of reference kernel for output validation.
4. Definition of tuning parameters.
5. Setup of kernel arguments.
6. Launch of the tuning process.
7. Retrieval of results.

```
cltune::Tuner tuner(platformIndex, deviceIndex);
size_t kernelId = tuner.AddKernel({"path/to/kernel.cl"},
    "kernelName", ndRangeDimensions, workGroupDimensions);
tuner.SetReference({"path/to/reference_kernel.cl"},
    "referenceKernelName", ndRangeDimensions,
    workGroupDimensions);

tuner.AddParameter(kernelId, "VECTOR_TYPE", { 1, 2, 4, 8
    });
tuner.AddParameter(kernelId, "USE_CONSTANT_MEMORY", { 0, 1
    });

tuner.AddArgumentInput(bufferA);
tuner.AddArgumentInput(bufferB);
tuner.AddArgumentScalar(helperVariable);
tuner.AddArgumentOutput(bufferResult);

tuner.Tune();
tuner.PrintToScreen();
```

Figure 3.1: Host program written in CLTune.

```
#if USE_CONSTANT_MEMORY == 0
#define MEMORY_TYPE __global
#elif USE_CONSTANT_MEMORY == 1
#define MEMORY_TYPE __constant
#endif

__kernel void tunedKernel(MEMORY_TYPE float* bufferA, ...)
{
    ...
}
```

Figure 3.2: Adding support for autotuning to kernel via preprocessor macros.

In order to support tuning parameters, kernel source file needs to be modified as well. In case of CLTune, the tuner exports parameter values from given configuration to kernel by using preprocessor definitions. Code needs to be modified, so that the values have intended effect. Simple example of such modification is shown in figure 3.2.

3.2 Kernel Tuner

Kernel Tuner [2] is another open-source kernel autotuning framework. It supports tuning of OpenCL and CUDA kernels as well as regular C functions, though in the last case, user is responsible for measuring execution duration. API is provided for Python. Compared to CLTune, it provides more utility methods, for example ability to set kernel compiler options, measuring execution duration in multiple iterations to increase accuracy and validating output with user-defined, precomputed answer rather than being restricted to reference kernel.

Disadvantages include again lack of support for composite kernels and inability for integration into existing software. However, Kernel Tuner is still actively developed and some of these shortcomings may be eventually amended.

Host program has to be written in similar fashion as in CLTune, preprocessor definitions for parameters are exported in the same way.

Figure 3.3 contains major portion of host program written for Kernel Tuner.

3.3 OpenTuner

Todo...

```
def tune():

with open('stencil.cl', 'r') as f:
kernel_string = f.read()

problem_size = (4096, 2048)
size = numpy.prod(problem_size)

x_old = numpy.random.randn(size).astype(numpy.float32)
x_new = numpy.copy(x_old)
args = [x_new, x_old]

tune_params = OrderedDict()
tune_params["block_size_x"] = [32*i for i in range(1,9)]
tune_params["block_size_y"] = [2**i for i in range(6)]

grid_div_x = ["block_size_x"]
grid_div_y = ["block_size_y"]

return kernel_tuner.tune_kernel("stencil_kernel",
    kernel_string, problem_size,
args, tune_params, grid_div_x=grid_div_x,
    grid_div_y=grid_div_y,
verbose = True)


if __name__ == "__main__":
tune()
```

Figure 3.3: Host program written in Kernel Tuner. Source: [4]

4 KTT framework

Todo...

5 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] Ben van Werkhoven. *Kernel Tuner: A Search-Optimizing GPU Code Auto-Tuner*. 2018.

[3] Jason Ansel et al. "OpenTuner: An Extensible Framework for Program Autotuning". In: *International Conference on Parallel Architectures and Compilation Techniques*. Edmonton, Canada, Aug. 2014. URL: `http://groups.csail.mit.edu/commit/papers/2014/ansel-pact14-opentuner.pdf`.

[4] Ben van Werkhoven. *Kernel Tuner Example*. 2018. URL: `https://github.com/benvanwerkhoven/kernel_tuner/blob/master/examples/cuda/expdist.py` (visited on 8/3/2018).

[5] 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...