Obsidian and CUDA programming

Parallel Functional Programming Lab

Bo Joel Svensson Mary Sheeran

May 15, 2016

1 Introduction

This lab consists of 7 tasks. Make sure you have completed all of them. If you have questions or problems during this lab, do not hesitate to contact me at *joels@chalmers.se*.

1.1 Learning objectives

This lab is about GPU programming. It will serve as an introduction to CUDA programming. CUDA is NVIDIA's C dialect for data parallel programming of their GPUs[1]. The level of the CUDA introduction is aimed to suit someone who has never seen a CUDA program before. The lab then introduces you to a library called Obsidian that is currently being actively developed and in a constant state of flux. From this we hope you take with you an appreciation for the embedded language approach to tackling the challenges of new and peculiar emerging parallel hardware.

2 Programming in CUDA

The purpose of this part of the lab is to become familiarised with CUDA and the *nvcc* compiler. We are going to implement a simple CUDA program. A CUDA program consists of *kernels*, programs that run on the GPU, and *glue* code that runs on the CPU.

The role of the glue code is to launch computations on the GPU and to coordinate the transfer of data and allocation of memory.

What you have heard about CUDA in the lecture should be enough to complete these tasks but if you want to go deeper into details, look at the CUDA programming manual [1]. For each CUDA programming task, create a file `taskX.cu` where X is the number of the task you are implementing.

Now it is time to write a small kernel and then get it running on the GPU.

[1] `www.nvidia.com/CUDA`

Task 0: Element-wise addition and glue code

Implement a kernel for element-wise addition of vectors. To get started you may use the template presented below.

```
__global__ void vadd(float *v1, float *v2, float *r){
  unsigned int gtid = blockIdx.x * blockDim.x + threadIdx.x;

  r[gtid] = ...
}
```

Now implement the CUDA code that launches the computation on the GPU. A skeleton of this glue code is given in the slides for the GPU programming lecture. It needs to be adapted for the this particular experiment though. The task is to adapt the glue code to execute the kernel from the previous task or to write your very own glue code.

Task 1: Data generation and timing

Generate input data of various sizes and add code to take timing measurements of the kernel's execution. Try to set up the timing code so that only the execution of the CUDA kernel is timed, not the data transfer or any computation you perform on the CPU (such as data generation). Keep in mind the scale of parallelism on the GPU when designing your timing experiments.

There are tools you can use that performs very much more interesting timing for you. One example is the CUDA profiler. If you decide to use the CUDA profiler or any other method for timing, specify your approach in the report

Task 2: An experiment of your own

The kernel implemented above is very small and uses no shared memory or communication between threads. This task is left very open. You may implement any algorithm you want but try to explore the effects of introducing shared memory. Potential starting points are

- Reduction (for example sum or product).
- Linear algebra operations
  - saxpy
    A Haskell specification of saxpy is
    `saxpy (a :: Float) = zipWith (\x y -> a * x + y)`
  - Matrix multiplication
    `mm xxs yys = [[sum (zipWith (*) xs ys)|ys <- transpose yys] | xs <- xxs]`
- Horner's Method: Evaluation of polynomials.
- Fractal generation: Mandelbrot
- Image processing (Blur or other stencil operations)
- Anything.

Measure the execution time of these kernels for sensible workloads. If you implement more than one version (with and without use of shared memory), measure both and reason about the difference in execution time.

For inspiration you can also look at some of the papers by the Chalmers graphics research group [3, 2]. They are really pushing the envelope of GPU coding.

3 Installing Obsidian

You can install Obsidian from Hackage with the command `cabal install Obsidian`. This should bring down version 0.4.0.0 of Obsidian, which is compatible with this lab. This version of Obsidian should work well under GHC 7.8.3 and 7.10.

We recommend that you use a cabal sandbox. If so the following procedure sets up and install Obsidian in a sandbox:

```
cabal update
cabal sandbox init
cabal install Obsidian
```

After performing the sequence of operations listed above you can issue the `cabal repl` command to enter in an interactive (ghci) session with access to the Obsidian libraries.

4 Programming in Obsidian

In this part of the lab you will write Obsidian code. Each task entails writing kernel code and code for launching that kernel on the GPU. There are two ways to do this, either you generate CUDA kernel code and then write, directly in CUDA, code to launch it, or you use the Obsidian library for launching the kernel from within Haskell. Place all CUDA code and generated kernels in files named `taskX.cu` as before. The Haskell code can be implemented in the provided template, `Lab.hs`. This template is available on the course web page.

Task 3: Reimplement vector addition

As a warm up exercise, reimplement `vadd` in Obsidian. Generate the CUDA code and execute it using the glue code created earlier in this lab (or launch from within Haskell). If you want to see take a look at the CUDA code that Obsidian generates, try the `genKernel` function.

```
genKernel :: ToProgram prg
          => Word32
          -> String
          -> prg
          -> String
genKernel numThreads kernel_name obs_func = ...
```

Import `Obsidian.CodeGen.CUDA` for exposing the `genKernel` function.

Task 4: Reduction

This task is to implement reduction in Obsidian. Reduction operations take arrays as input and produces a single value, think of Haskell's `foldl1` for example. You may chose to implement a general reduction combinator or to reduce for a specific operation (such as +). Your local reduction kernel should be able to reduce arrays that have a length that is a power of two. Use a divide and conquer approach: repeatedly split the array in half use zipWith (+) on the halves (if + was chosen as the operator).

Task 5: Implement dot product

Implement a kernel for performing multiple dot products in Obsidian, one dot product per block.

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{n} a_i * b_i = a_0 b_0 + a_1 b_1 + \ldots + a_n b_n$$

Dot product is a combination of an element-wise operation and a reduction. The two previous tasks can be used as inspiration here. When it comes to writing the CUDA code, you must set it up to first launch the element-wise product kernel followed by the reduction kernel.

It is possible, with a small tweak to convert this computation of multiple dot products to the computation of one gigantic dot product over the entire input arrays. This is left as a voluntary task.

Task 6: An Obsidian experiment of your own

This is, again, a task left very open. You may implement any algorithm you want but try to explore the effects of using shared memory. It is also perfectly ok to reuse the previous Obsidian code in this task, but with the addition of exposing parameters. The task would then be to explore what happens to the performance of the generated kernel at different parameter settings. Other potential starting points are:

- Reduction (for example sum or product).
- Linear algebra operations
  - saxpy
    A Haskell specification of saxpy is
    `saxpy (a :: Float) = zipWith (\x y -> a * x + y)`
  - Matrix multiplication
    `mm xxs yys = [[sum (zipWith (*) xs ys)|ys <- transpose yys] | xs <- xxs]`
- Horner's Method: Evaluation of polynomials.
- Fractal generation: Mandelbrot
- Image processing (Blur or other stencil operations)
- Anything.

Good Luck and have fun!

References

[1] CUDA programming manual. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[2] Markus Billeter, Ola Olsson, and Ulf Assarsson. Efficient stream compaction on wide SIMD many-core architectures. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, New York, NY, USA, 2009. ACM.

[3] Ola Olsson, Markus Billeter, and Ulf Assarsson. Clustered deferred and forward shading. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics*, EGGH-HPG'12. Eurographics Association, 2012.