Obsidian and CUDA programming

Parallel Functional Programming Lab B

Bo Joel Svensson Mary Sheeran

April 15, 2013

1 Introduction

This lab consists of 5 tasks. Make sure you have completed all of them. The lab also has 6 voluntary exercises. You are free to do any, all or none of these. However, select one or two of the voluntary exercises for serious treatment if you plan to approach us for a potential master thesis related to functional GPU programming.

If you have questions or problems during this lab, do not hesitate to contact me at *joels@chalmers.se*.

1.1 How to report this lab

Hand in all your code and a short written report with answers to question that are not easily answered as code or comments. The report and all source code should be submitted using the FIRE system `https://fire.cs.chalmers.se:8069/cgi/Fire-pfp`.

1.2 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 vecAdd(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 lecture 5. 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

Voluntary 0: An experiment of your own

The kernel implemented above is very small and uses no shared memory or communication between threads. This voluntary 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]`
- Image processing (Blur or other stencil operations)
- Anything.

For inspiration you can also look at some of the papers by the Chalmers graphics research group [6, 2]. They are really pushing the envelope of GPU coding.

3 Programming in Obsidian

Obsidian is work in progress and the version of it that is presented to you here has very recently gone through some major revisions. Be patient with any bugs you discover. Actually we would be very grateful if you make note of any bugs you find in your report.

In this part of the lab you will write both Obsidian and CUDA code. Each task entails generating some CUDA code from Obsidian code and the implementation of CUDA glue code to launch it. Place all CUDA

code and generated kernels in files named `taskX.cu` as before. The Haskell code can be implemented in the provided template, `LabB.hs`. This template is available on the course web page.

Task 2: Reimplement vector addition

As a warm up exercise, reimplement `vecAdd` in Obsidian. Generate the CUDA code and execute it using the glue code created earlier in this lab.

Task 3: 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 4: Implement dot product

Implement dot product in Obsidian.

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

Dot product is a combination of an element-wise operation and a reduction. We suggest that you implement these as two separate kernels. 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.

This concludes the obligatory part of the lab.

Voluntary 1: Single-kernel dot product

Reimplement dot product but using only a single kernel.

Voluntary 2: Implement matrix multiplication

Implement matrix multiplication for fixed (static) size matrices in Obsidian.

Voluntary 3: Experiment of your own

Go above and beyond what is specified in the Obsidian tasks. Be clever and see what happens. Use shared memory. You can also here try to implement your favourite data parallel algorithm.

Voluntary 4: Bug hunting

Did you find any bugs while working with Obsidian? If you have been experimenting outside of the borders of this lab assignment, it is very likely that you have. Pointing them out makes us grateful but proposing constructive insights on how to fix them merits you greatly.
Possible bugs you can discover are:

- Wrong CUDA code is generated. That is, the generated code does not correspond with your intuition of what the Obsidian represents.
- Missing class instances, type family instances etc.
- Missing cases in any of the compilation passes.

Voluntary 5: Propose an extension

This voluntary task is about proposing an extension. Suggest propositions should not just be "add X" or "do Y instead of Z", they should be feasible and thought out ideas with a sketch of an implementation plan. This does not need to be so detailed as to include any actual code, though.

One possible direction is to use Obsidian as a code generating back-end for some a DSL specialised to narrower domain. An example of such an approach is [4].

For inspiration you can also look at the references [3, 5], that describe Obsidian.

Good Luck and have fun!

4 Hints

4.1 About array sizes

In the current version of Obsidian an array may have either a static or a dynamic length. This is controlled by the first type parameter to a `Pull`. If it is a `Pull Word32 a` array, the length is static and if it is `Pull EWord32 a` it is dynamic. The general guideline is that local computations should be performed with static lengths. This enables the code generator to figure out things like number of threads needed and amount of shared memory required.

To make the types a little bit cleaner aliases exist

```
type SPull = Pull Word32
type DPull = Pull EWord32
```

and similarly for push arrays.

```
type SPush t a = Push t Word32 a
type DPush t a = Push t EWord32 a
```

4.2 Useful Obsidian library function

- `force :: SPull a -> BProgram (SPull a)`
- `force :: SPush Block a -> BProgram (SPull a)`
- `halve :: ASize l => Pull l a -> (Pull l a, Pull l a)`
- ```
  mapG :: ASize l
         => (SPull a -> BProgram (SPull b))
         -> Pull l (SPull a) -> Push Grid l b
  ```
- `mkPullArray :: s -> (Exp Word32 -> a) -> Pull s a`
- `replicate :: s -> a -> Pull s a`
- `zipWith :: ASize l => (a -> b -> c) -> Pull l a -> Pull l b -> Pull l c`
- ```
  zipWithG :: ASize l
             => (SPull a -> SPull b -> BProgram (SPull c))
             -> Pull l (SPull a) -> Pull l (SPull b) -> Push Grid l c
  ```

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] Koen Claessen, Mary Sheeran, and Bo Joel Svensson. Expressive Array Constructs in an Embedded GPU Kernel Programming Language. In *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*, DAMP '12, 2012.

[4] A. Cole, A. McEwan, and G. Mainland. Beauty And The Beast: Exploiting GPUs In Haskell, 2012.

[5] Joel Svensson. Obsidian: GPU Kernel Programming in Haskell. Technical Report 77L, Computer Science and Enginering, Chalmers University of Technology, Gothenburg, 2011. Thesis for the degree of Licentiate of Philosophy.

[6] 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.