Algorithmic Techniques for Many-core Processors
PASI Summer School

January 10, 2011

1 Introduction

The purpose of the hands-on labs is to assist the PASI short course that teaches proven algorithmic techniques for many-core processors. This manual includes an introductory lab, followed by the 7-point stencil computation discussed in the lectures.

In order to cater to students with different levels of understanding of CUDA programming, we provide one hands-on lab, with multiple performance optimizations that the student needs to implement. The labs are designed with levels of different difficulties. Due to limited time in a hands-on lab section, we suggest that you choose the difficulty level that best fits your programming skills. If you have time, you can take on another level.

2 Lab 0: Package Download and Environment Setup

1. Objective

   The purpose of this lab is to check your environment settings and to make sure you can compile and run CUDA programs on the NCSA AC cluster. The objectives of this lab are summarized below:

   - To download the assignment package, unpack it and walk through the directory structure.
   - Set up the environment for executing the assignment.

2. Preliminary work

   Step 1 Go to `http://goo.gl/zCZzQ` and find your login information.

   Step 2 Use an SSH program to login to ac.ncsa.uiuc.edu[1], using the training account login and initial password given to you. Your home directory can be organized in any way you like. To unpack the SDK framework including the code for all of

[1] If there are connection issues with ac.ncsa.uiuc.edu, try the backup host ac1.ncsa.uiuc.edu

the lab assignments, execute the unpack command in the directory you would like the SDK to be deployed.

```
$> tar -zxf ~tra108/PASI_stencil.tgz
```

Note: **If you are a remote user, we recommend you use an FTP program with the SFTP protocol and your account/password to retrieve the source files on ac.ncsa.uiuc.edu for editing because the network may be unstable and disconnected during lab sections.**

Step 3 Go to the lab0 directory and make sure it exists and is populated.

```
$> cd PASI_stencil/benchmarks/deviceQuery/src/cuda
```

There should be at least two files:

(a) deviceQuery.cu

(b) Makefile

3. Make the first CUDA program and execute it on the AC cluster.

The execution steps of this lab are listed below. You shall use the commands listed here in the remaining labs in the manual.

Step 1 Set environment variables. **You have to do this step whenever you start a new terminal window.**

```
$> cd PASI_stencil.
$> source env.sh
Parboil library path is set:  /home/ac/tra108/PASI_stencil
```

Step 2 Compile the lab.

```
$> ./parboil compile deviceQuery cuda
```

Step 3 Execute the lab.

```
$> ./parboil run deviceQuery cuda default
```

You shall see a similar message as follows:

```
There is 1 device supporting CUDA

Device 0: "GeForce GTX 480"
  Major revision number:                         2
  Minor revision number:                         0
  Total amount of global memory:                 1610285056 bytes
  Number of multiprocessors:                     15
  Number of cores:                               120
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per block:           1024
```

```
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             513 bytes
  Clock rate:                                    1.40 GHz
  Concurrent copy and execution:                 Yes

Test PASSED
Parboil parallel benchmark suite, version 0.1
```

4. Understanding the Parboil framework

   In the hands-on labs for the ManyCore Processors course, we use a framework called Parboil to organize the labs. The Parboil framework is developed by the IMPACT Research group at the University of Illinois, and is meant to provide an easier interface to manipulate benchmarks and measure performance on a GPU platform.

   All lab assignments shall be compiled and run by the parboil script under the unzipped lab package directory, as listed in the previous sections. The parboil script is designed to submit jobs to the AC cluster in a batch mode fashion.

   The unzipped lab package is composed of the following directory structures. For simplicity, here only lists the information you may need to know:

   File "env.sh" The environment settings file.

   File "parboil" The main script to compile and run the labs.

   Directory "benchmarks/" This directory stores the labs.

   Subdirectory list of benchmarks : deviceQuery/ stencil/

   Each benchmark directory contains the following directories: "build", "input", "output", "src", and "tools". "build" stores the executable of each lab. "input" stores the problem parameters and input data sets. "output" stores the problem output data sets and golden results for comparison. "src" stores the source files of the problem of different versions or phases. "tools" stores the tools to assist the lab result comparison.

   Directory "common/" This directory contains libraries shared by all labs.

   Subdirectory "src/" : This directory contains the main parboil library source file. If you plan to port the lab package to another environment, you should rebuild the parboil library at this place.

   Directory "driver/" : Here lists the related tools to manipulate the compilation and execution of the labs.

   The output log of each lab execution shall list the execution time taken for each of the following sections, IO, GPU, Copy, and Compute. Their meanings are listed below.

IO Time spent in input/output.

GPU Time spent computing on the GPU, recorded asynchronously.

Copy Time spent synchronously moving data to/from GPU and allocating/freeing memory on the GPU.

Driver Time spent in the host interacting with the driver, primarily for recording the time spent queuing asynchronous operations

Copy_Async Time spent in asynchronous transfers

Compute Time for all program execution on the host CPU other than parsing command line arguments, I/O, GPU, and copy.

CPU/GPU Overlap Time double-counted in asynchronous and host activity: automatically filled in, not intended for direct usage.

3 Lab 1: 2-D blocking and Register Tiling

1. Objective
   The purpose of this lab is to provide a real application environment to explore the performance effects of 2-D blocking with data reuse and register tiling transformations. This lab introduces the 7 point stencil probe application, a micro-benchmark and test-bed for large stencil grid applications.

   This lab will draw on the lecture material on:

   (a) Increasing locality in dense arrays (with a focus on tiling).

   (b) Improving efficiency and vectorization in dense arrays.

   The source file directory for this lab contains two options for students:

   Difficulty Level 1 Students who are experienced in parallel program optimization and have a good understanding of the lecture material should choose this section. Students choosing Lab 1.1 will need to implement Optimizations 1, 2, and 3 detailed in Section 3 below in the file **kernels1.1.cu**.

   Difficult Level 2 Students who are not very experienced in parallel programming and/or are not very familiar with the lecture material should choose this section. Students choosing Lab 1.2 will need to implement only Optimizations 2 and 3 detailed in Section 3 below in the file **kernels1.2.cu**.

2. Examine and Understand the 7 pt. Stencil Kernel
   The parboil benchmark 'stencil' contains the data and source code, which should compile and run correctly with the parboil interface as it is. Note that the output comparison step will take several seconds.

   ```
   $> ./parboil run stencil cuda default
   ```

   All source code for the lab can be found in the benchmarks/stencil/src/cuda subdirectory of the provided lab package. The 7-pt stencil probe application is an example of nearest neighbor computations on an input 3-D array. Every element of the output array is described as a weighted linear combination of itself and 6 neighboring values as shown in the lecture notes.

   The **main** function in the file **main.cu** contains the major components of kernel setup and execution including global memory allocation, input data copied to global memory, kernel launch and output data copied back into host memory. The input data is generated in the **generate_data** function in the file main.cu.

   The naive kernel **block2D_naive** is provided for reference in **kernels.cu**, and is invoked to compute the output grid as a weighted combination of elements of the input grid. The kernel configuration parameters and launch are shown in the **main.cu** file in the main function.

```
dim3 block (tx, ty, 1);
dim3 grid ((nx+tx-1)/tx, (ny+ty-1)/ty,1);
block2D_naive<<<grid, block>>>(fac,d_A0,d_Anext,nx,ny,nz);
```

Note that each thread block processes a **BLOCK_SIZE_X** by **BLOCK_SIZE_Y** block, within a for-loop that iterates in the z-direction. In the given kernel **block2D_naive**, each thread computes a single output point, by a weighted combination of 7 global memory elements that from the nearest neighbors.

To simplify boundary conditions, the outer boundary of each x-y plane (topmost and bottommost rows, and leftmost and rightmost columns) is initialized to zeros, and the thread blocks process the elements within this boundary. Thus, the thread blocks that form the outer boundary of the x-y plane have a fraction of threads that are idle.

4 Modifying the Kernel

1. Difficulty Levels

   This lab is organized into two difficulty levels, which students can choose from, depending on their experience in parallel program optimization and/or their confidence and understanding of the lab material. The students will be required to implement the required optimizations in a manner that best exploits the performance potential of the application. The functions for Optimizations 1, 2, and 3 (detailed below) are declared as **block2D_opt_1**, **block2D_opt_2**, and **block2D_opt_3** in the file **kernels.cu**.

   Lab 1.1 Students choosing this lab will need to implement Optimizations 1, 2, and 3 in the file **kernels1.1.cu**. The naive kernel **block2D_naive** (explained in Section 2) is provided for reference. Make sure that the file **kernels1.1.cu** is included in the **main.cu** file.

   Lab 1.2 Students choosing this lab will be provided with the kernels **block2D_naive** and **block2D_shared** for reference. The students will need to implement only Optimizations 2 and 3 detailed below in the file **kernels1.2.cu**. Make sure that the file **kernels1.2.cu** is included in the **main.cu** file. Additionally, make sure that the kernel invoked in **main.cu** is **block2D_shared**.

2. Optimization 1

   Analyze the given kernel in detail. Every thread loads 7 global memory elements, thus meaning that even neighboring threads that share data points load the same point repetitively. Thus, this indicates good potential for performance improvement from data reuse. Provided below is a rough outline of the logical steps needed to arrive at an optimized kernel. Note that multiple lines of code may be necessary to implement each of these logical steps.

   Step 1 Declare a shared memory structure to be used for data sharing. Determine the appropriate size from your analysis of the problem.

   Step 2 For each frame along the z-axis, threads should load data points in collaborative fashion into the shared memory structure. Ensure that synchronization is effectively used to ensure data consistency.

   Step 3 For each frame along the z-axis, compute the weighted combination of nearest neighbor elements for the output point.

   Step 4 Change the kernel launch commands within the main function to invoke kernel **block2D_opt_1**.

   Step 5 Once you get a working implementation of the above methodology, experiment with different combinations to find the best performing solution.

The following pseudo code for Optimization 1 shows Step 2 and Step 3.

```
//Pseudo Code: Load elements into shared memory {
for(each frame along z-axis) {
  if(indices within range)
    shared_mem[index1] = data[index2];
  syncthreads();

  if(indices within range)
    output[index] = Weighted combination of neighboring elements;
  syncthreads();
}
```

	Before Optimization 1	After Optimization 1	Speedup
GPU Time			

3. Optimization 2

Analyze the kernel you are working with in detail. Note that for each z-frame k, you load the $(k-1), k, and (k+1)^{th}$ frame (alternatively called the bottom, current and top frames). This means that for the next frame, only the $(k+2)^{th}$ frame needs to be loaded from global memory, as the k and $(k+1)^{th}$ frame were already fetched in the previous iteration. Provided below is a rough outline of the logical steps needed to arrive at an optimized kernel. Note that multiple lines of code may be necessary to implement each of these logical steps. **Note that the kernel block2D_opt_1 (defined in kernels.cu) is provided as an additional reference, for those students choosing Lab 1.2**.

Step 1 Determine the appropriate size of the declared shared memory structure.

Step 2 Insert a prologue that loads the very first 2 required z-frame values from global memory into registers/shared memory respectively. This is to make sure that the first iteration of the inner loop has all the required z-frames. You may also decide to store all frames(top, bottom and current) in shared memory in which case, your code will look different from the example pseudo codes shown.

The following pseudo code is for Optimization 2, Step 1 and 2

```
//Declaring shared memory structures to hold 3 frames
__shared current;
float bottom, top;

//Prologue
if(indices within range){
```

```
  bottom = data[index]; //Corresponding data from Frame 0

  //Threads load collaboratively into shared memory
  current frame = {Load frame 1};
  __syncthreads();
}
```

Step 3 Within the inner for-loop that iterates over the frames on the z-axis, load the next required value from the top frame into a register before computation. After computation, for the next iteration, the current frame becomes the bottom one, and the top frame becomes the current frame. Thus, note that within each iteration of the for-loop, only the required top frame needs to be loaded from global memory.

The following pseudo code is for Optimization 2, Step 3

```
for(each frame along z-axis) {
  if(indices within range)
    top = data[index]; //Load next frame data from global mem

  syncthreads();

  if(indices within range)
    output[index] = Weighted combination of neighboring elements;

  syncthreads();

  bottom = Current[index];
  //Threads copy data collaboratively into shared memory
  current frame = {Copy data from top};
}
```

Step 4 Change the kernel launch commands within the main function to invoke kernel **block2D_opt_2**.

	Before Optimization 2	After Optimization 2	Speedup
GPU Time			

4. Optimization 3

In the kernel that you are working with, note that each thread computes exactly one output point. By having each thread compute multiple points, we would be implementing the register tiling optimization detailed in the lecture. In this optimization, the student will modify the kernel to compute 2 output points instead of 1.

Step 1 Make sure that the sizes of the declared shared memory elements are large enough to hold input data for 2 output points.

Step 2 Modify the loads from global memory to shared memory to load the additional data points required for computing output point 2. If required, you may also need to modify the prologue and the frame update for the next iteration.

The following pseudo code is for Optimization 3, Steps 1 and 2

```
//Declaring shared memory structures to hold 3 frames for 2 output points

__shared current_combined;
float top1, top2, bottom1, bottom2;

//Prologue
if(indices within range) {
  bottom1 = data[index1];//Load Frame 0 for output point 1
  bottom2 = data[index2];//Load Frame 0 for output point 2

  //Threads load collaboratively into shared memory
  Current_combined = {Load frame 1 for o/p points 1 and 2};
}
```

Step 3 Add additional computation steps to load the top frames corresponding to the 2 output points compute the 2nd output point.

The following pseudo code is for Optimization 3, Step 3

```
for(each frame along z-axis) {

  if(indices within range)
    top1 = data[index1]; //Load data from next frame for o/p point 1
    top2 = data[index2]; //Load data from next frame for o/p point 2

  syncthreads();

  if(indices within range) {
    output[index1] = Weighted combination of index1 neighboring elements;
  }

  if(indices within range) {
    output[index2] = Weighted combination of index2 neighboring elements;
  }

  syncthreads();

  bottom1 = Current[index1];
  bottom2 = Current[index2];
```

```
  //Threads copy data collaboratively from top1 and top2
  Current_combined = {Copy data from top1 and top2 collaboratively};
}
```

Step 4 Change the kernel launch commands within the main function to invoke kernel **block2D_opt_3**, and also make necessary changes (if required) to the kernel configuration parameters, such as **block**, **grid** etc.

	Before Optimization 3	After Optimization 3	Speedup
GPU Time			

5 Questions

Of all the configurations you tested, which ones performed best? A few questions that could get you to analyze the performance better are: What is the optimal size of the shared memory structure? Which are the nearest neighbor elements that have a high degree of sharing with neighboring threads? How can global memory loads be performed so as to make best use of the underlying memory infrastructure? (Hint: coalesced accesses). For Optimization 1, what patterns of the global memory loads performed best. Why? Can you think of other ways to load data into a 2-D shared memory structure that could give better performance? For Optimization 3, how did you choose the 2 output points that a single thread computes? Are there other combinations that you could have chosen?