CS205, Fall 2012
Computing Foundations for Computational Science
Hanspeter Pfister, Cris Cecka

HARVARD
School of Engineering and Applied Sciences

Homework 4 – Due November 9th, 2012 at 11:59pm

This week's homework covers covers more advanced CUDA with shared memory programming, and block / grid optimization.

Download the code for HW4 here.

or

`wget http://www.cs205.org/resources/hw4.zip`

Problem 1 - Introduction

PyCUDA Common Errors

The PyCUDA error reporting is not the most helpful in the world. Heres a quick key to improve your debugging and save the forum from repeated questions.

`Operation failed (dead context maybe?)`

Your CUDA kernel is likely failing. Most commonly, this is due to an index out-of-bounds error.

`Invalid type on parameter #N (0-based)`

The Nth input to your kernel has an invalid type. Make sure the input parameters to the kernel are cast to the correct type:

- `[unsigned] char = numpy.[u]int8`
- `[unsigned] short = numpy.[u]int16`
- `[unsigned] int = numpy.[u]int32`
- `[unsigned] long = numpy.[u]int64`
- `float, double = numpy.float32, numpy.float64`
- pointers (`float*, double*, int*`) from GPU memory allocations

`Python argument types do not match C++ signature`

The block, grid, or shared memory size is likely invalid. Make sure these are integers.

Additional Resources

PyCUDA Documentation

PyCUDA Examples and FAQ

Problem 2 - Fix The Code 1 [20%]

Recall in HW3, we approximated the second derivative of the wave equation with the second-order central difference equation. To speed up the computation even more, we want to implement this operation in CUDA.

Intern Dorothy writes the following kernel for us. Assume the blocks and grid are one-dimensional and that the total number of threads is greater than or equal to N.

```
/** Second order central difference stencil
 *
 * Input: Array a of length N and grid size dx
 * Output: (a[i-1] - 2*a[i] + a[i+1]) / (dx*dx) for all 0 < i < N-1
 */
__global__ void D2x_kernel(double* a, int N, double dx)
{
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if (tid > 0 && tid < N-1) {
        a[tid] = (a[tid-1] - 2*a[tid] + a[tid+1]) / (dx*dx);
    }
}
```

Identify the Problem

Explain what is wrong with the kernel above and a simple way to fix it.

Implement the Solution

Implement your corrected function with PyCUDA. Test your code by applying it to the function $\sin(x)$ sampled with $N = 1048576$ double-precision (`float64`) points in $[0, 1]$. That is, `x = numpy.linspace(0,1,2**20)`. Note that the grid spacing is then $dx = 1/(N-1)$. Plot the result.

Submission Requirements

- `P2.pdf`: Explanations and plots.
- `P2.py`: Code implementation.

Problem 3 - Fix the Code 2 [15%]

Suppose we have a CUDA program which generates 1024 floats that we need to determine the minimum value of. We could copy the data back to the host, but we decide the overhead is undesirable and instead choose to launch a single block of threads to perform this computation. To accomplish this, we will perform a one thread-block reduction with the `min` operator.

Intern Dorothy writes the following CUDA kernel for us and claims that it works.

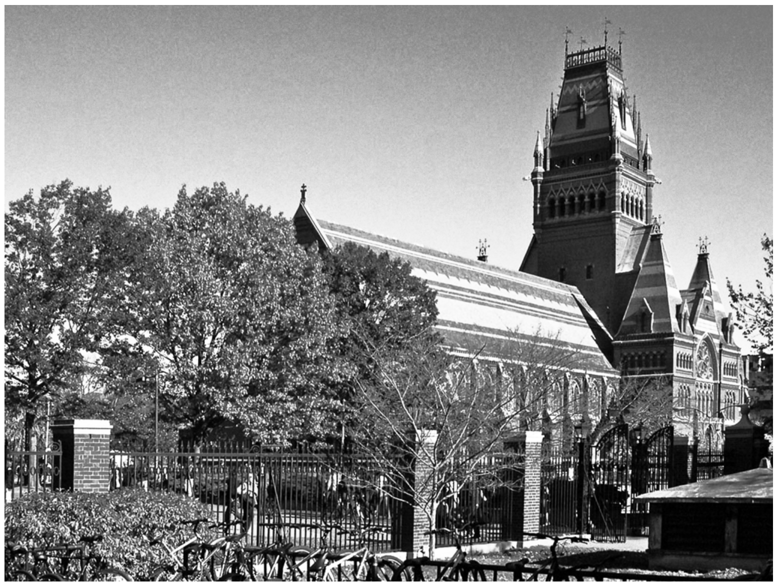
```
/** Single thread-block min-reduction CUDA kernel
 *
 * Input: Array d_data of length 1024
 * Output: Minimum value of d_data stored in array d_min of length 1
 */
__global__ void min_kernel(float* d_data, float* d_min)
{
    // Allocate shared memory in the kernel call
    extern __shared__ float s_data[];

    int idx = threadIdx.x;

    // Copy data to shared memory
    s_data[idx] = d_data[idx];

    // Calculate the minimum value by doing a reduction
    int half = blockDim.x / 2;
    while (half > 0 && idx < half) {
        if (s_data[idx] > s_data[idx + half])
            s_data[idx] = s_data[idx + half];
        half = half / 2;
    }
    // Store the minimum value back to global memory
    d_min[0] = s_data[0];
}
```

You recognize that while this kernel may work some of the time, it uses unsafe operations. Explain three things that could cause errors with the kernel and correct the code. Provide an implementation that verifies your solution is correct.

Submission Requirements

- `P3.pdf`: Explanations.
- `P3.py`: Implementation that compares the result of your `min_kernel` to `np.min()`.

Problem 4 - CUDA Photoshop [35%]



In this problem, we will be manipulating images with a linear filter.

The code in `P4_serial.py` applies a sharpening kernel to the interior pixels of an image with a sharpening coefficient $\varepsilon = 0.004375$, 20 iterations, and the following 9-point stencil:

$$\begin{pmatrix} -1 & -2 & -1 \\ -2 & 12 & -2 \\ -1 & -2 & -1 \end{pmatrix}$$

The boundary pixels are left alone. This stencil acts to amplify edges in the image and some version of this is contained in nearly every photo editing software. The sharpening kernel is inherently unstable and too many applications will destroy the photo. One way to not go too far is to compute the mean and variance of the image and stop the application when the variance becomes too large.

Recall the mean is computed as:

$$mean = \mu = \frac{1}{width * height} \sum_{i,j} image[i,j]$$

And the variance is computed as:

$$variance = \sigma^2 = \frac{1}{width * height} \sum_{i,j} (image[i,j] - \mu)^2$$

Implement With 2D Thread Blocks

Implement the sharpening kernel with PyCUDA using 2D thread blocks and a 2D grid of blocks. Every iteration, compute the mean and variance on the host before sending the image to the GPU for the sharpening kernel. Show that your program executes correctly. Find a good execution configuration (block and grid size) and explain why it is a good configuration.

Mean / Variance

Compute the mean and variance on the GPU as well. You may use any efficient method you like. Now you should be able to transfer the image to the device once before the loop and keep it on the device to compute the sharpening kernel and the mean / variance multiple times.

Your program should output the mean and variance every iteration like the serial version and save the final image as `Harvard_Sharpened_GPU.png` when complete.

HINT: Consider using a CUDA reduce and/or element-wise operations.

Benchmarking

Run your code with the following images:

- `Harvard_Small.png` (512x384)
- `Harvard_Medium.png` (1024x768)
- `Harvard_Large.png` (2048x1536)
- `Harvard_Huge.png` (4096x3072)

Analyze the time for setup (GPU memory allocation and initialization), the time to compute the mean / variance, and sharpening execution time. Compare the performance of the provided serial code with your GPU kernel.

Submission Requirements

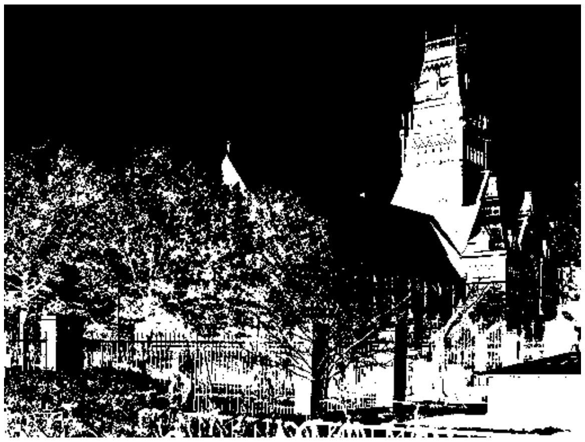
- `P4.pdf`: Explanations.
- `P4.py`: Sharpening code with GPU kernel and variance computation.

Problem 5 - CUDA Photoshop 2 [30%]

Another interesting image processing algorithm is region growing. Suppose we want to find all of the "dark regions" of an image. Perhaps we want to extract shadows for shape matching, balance the dark regions with the light regions, etc. Our first instinct might be to simply filter the image for all the dark pixels. A quick script to do that might look like:

```python
import matplotlib.image as img

in_file_name = "Harvard_Small.png"
out_file_name = "Harvard_SimpleRegion_CPU.png"

# Pixel value threshold that we are looking for
threshold = [0, 0.27];

if __name__ == '__main__':
  # Read image. BW images have R=G=B so extract the R-value
  image = img.imread(in_file_name)[:,:,0]
  # Find all pixels within the threshold
  im_region = (threshold[0] <= image) & (image <= threshold[1])
  # Output
  img.imsave(out_file_name, im_region, cmap='gray')
```

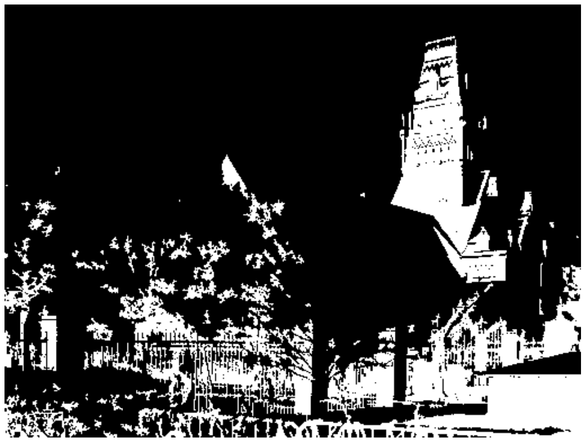
and this results in



which is good... but... really noisy. We were really interested in the dark REGIONS and now we have a lot of individual pixels.

One solution to fix this is a seeded region growing algorithm. Instead of simply searching for pixels, we'll seed our search with a couple of "interesting" pixels and "grow" the region by repeatedly adding neighbors that satisfy our original threshold. This way, the chosen pixels must be connected to a seed pixel through some path of pixels that all satisfy our threshold. That is, these pixels are connected into regions like we desired.

We have provided `P5_serial.py` for seeded region growing on a grayscale image. First, all pixels with values inside a chosen *seed threshold* range are selected. Then, repeated grow operations add adjacent pixels with values inside the original *threshold* range to the image region. This process continues until the regions stop growing. The results are much more satisfying:



However, we may have made a mess of things. The original filter was trivial and embarrassingly parallel, but the region growing algorithm is much more complicated...

Full image region growing

Using the extreme parallelism of CUDA, we can let one thread take responsibility for one pixel on each iteration of the region growing algorithm. That is, the algorithm will look like

- Find seed points – embarrassingly parallel.
- Each thread takes one pixel of the image and determines whether it should be added to the regions.
- Continue until no pixels were added in the last iteration.

You must determine the criteria for step two and determine how to test for convergence (in a parallel-safe way) in step three. Your program should read in the largest of the images, find the seed points, apply the region growing operation, print the total run-time (including all communication), and save the resulting region image as `Harvard_RegionGrow_GPU_A.png`.

Queue-based region growing

The above approach works quite well and is a very "CUDA"-like solution as we use massive parallelism and most of the reads and writes can be coalesced. It is severely different from the serial version though. The serial version keeps track of only the pixels in the newly grown "front" region at each iteration and checks only these pixels for inclusion into the region. This is called a queue-based approach. This approach seems inherently less parallel. Specifically, we don't know how large the front will be at each iteration or how to easily construct the next front in a parallel-safe way.

Implement a queue-based approach with CUDA. You may use the CPU for manipulating the queue if you need to. Print the total run-time (including all communication) and save the resulting region image as `Harvard_RegionGrow_GPU_B.png`.

Is this method faster than the "CUDA"-like solution above? Discuss.

Extra Credit

Improve one or both of the above seeded region growing algorithms GPU algorithms. Fully explain your approach and reasoning and provide implementations that show your improvements.

HINT: Try to reduce the number of kernel launches required and the amount of communication between the CPU and GPU. You will probably want to use parallel scan and/or reduction operations (documented at http://documen.tician.de/pycuda/array.html) to minimize the CPU workload/communication.

Submission Requirements

- `P5.pdf`: Explanations.
- `P5A.py`: Full image region growing.
- `P5B.py`: Queue-based region growing.
- `P5AExtra.py`: Faster region growing.
- `P5BExtra.py`: Faster region growing.