DATA LEVEL PARALLELISM

05/19/14

DLP Warmup (DUE: Fri. 5/23/2014, 5PM)

Start to play with programs utilizing data level parallelism executing on a graphics processor.

Data Level Parallelism

INTRODUCTION

In this project you will use the NVIDIA CUDA GPU programming environment to explore data-parallel hardware and programming environments. The goals include exploring the space of parallel algorithms, understanding how the data-parallel hardware scales performance with more resources, and utilizing the data-parallel programming model.

The warmup is designed to get you familiar with the CUDA tools and the NVIDIA Quadro hardware.

TOOLCHAIN

In this project we will be using NVIDIA's CUDA programming environment. This is installed on all Linux machines in Kemper 2107 and 2110. The toolkit is installed in /opt/cuda. You are welcome to do the assignment outside of the lab on your own machine, but you would need to have both NVIDIA CUDA-capable hardware and the CUDA environment installed on your machine.

Note: if you are logged in remotely, you will have problems executing your code if someone else is physically at the machine. Use the *`who`* command to see if you are alone.

To set up your machine for using CUDA you must place both the CUDA binaries (compiler) and libraries in your path. Add the following two lines to your ~/.cshrc file:

- *`setenv PATH ${PATH}:/opt/cuda/bin`*
- *`setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:/opt/cuda/lib`*

CUDA documentation is in `/opt/cuda/doc/`. See the `CUDA_C_Programming_Guide.pdf`.

SAMPLE CODE

You will be looking at a simple sample project called flops. The routine runTest( ), called from main( ), performs the following main steps:

1. initialize the CUDA GPU
2. allocate host and GPU memory
3. copy 2 float values into GPU memory
4. start timer
5. call flops_kernel (kernel which executes on the GPU), which:
    a. performs a single add and mult with the passed in values
    b. writes result to output variable in GPU
    c. returns
6. stops timer and determines run time
7. copies result from GPU memory back into host memory
8. cleans up and exits

USING CUDA

Before working on the flops program, download the program device.cu from SmartSite **Resources / Projects / DLP**. Compile it by running:

```
nvcc -o devQuery device.cu
```

Then run the program to see information about the installed GPU hardware. Save the output to turn in.

Flops Code Install

The file **nvidia-sdk-eec171.tar** contains a directory tree for CUDA SDK projects, with a project called flops and another called sum. Download this file (from **Resources / Projects / DLP** on SmartSite) and then untar it into your home directory using the following command:

```
tar xvf nvidia-sdk-eec171.tar
```

You should now have the directory *GPU* in your home directory. You will find the sample project files for the warmup in this directory: *GPU/C/src/flops*. The files are:

- Makefile
- flops.cu (the source file)

Make sure the sample compiles and runs before doing anything else. Run the following commands from the *GPU/C* directory:

```
make clean
make
./bin/linux/release/flops
```

Note: Don't forget to re-compile your project (using make) every time you make any changes to the source code.

WARMUP PROBLEM

Maximize FLOPS Achieved

Your goal is to modify the flops program in order to **measure and maximize** the number of floating-point operations per second (FLOPS). Currently the kernel only performs one floating-point add and multiply operation and returns, which may not lead to a high delivered FLOPS rate. To maximize FLOPS, you need to run many operations in parallel. For example, you may increase the number of threads per block, increase the number of blocks, and have kernels with a lot of compute and low control complexity (HINT: maybe unrolled loops). You also might consider utilizing the MAD instruction (multiply-and-add) since it allows you to count 2 floating-point ops using a single instruction. If the compiler sees a multiply followed by an add in the form x = a * b + c, it compiles to MAD(x,a,b,c). You should be able to calculate both your achieved FLOPS rate and your achieved memory bandwidth. Compare those two figures against the theoretical maximum for this GPU, which you should be able to find online (Wikipedia has good information). **Is this program compute or memory bound?**

Make sure that any computations you perform in the kernel will eventually influence results that are returned to the host processor; otherwise the compiler will optimize those results by itself. The compiler is smart and aggressive.

To calculate the FLOPS for your program, multiply the number of FP operations per thread by the total number of threads the kernel launches. Divide this by the program run time.

To calculate the memory bandwidth, count the number of **global** memory accesses made by each thread (Only the memory allocated by cudaMalloc( ) in the main routine is GPU global memory – the other variables in your thread are generally registers and should not be counted as global memory accesses). Then total up the number of memory accesses across all threads and divide by the run time.

SUBMISSION

Turn in to SmartSite: 1) output of the device.cu program; 2) your modified flops.cu file; 3) report the maximum FLOPS and memory bandwidth you achieved with the flops program, and state whether the program is compute- or memory-bound.

DUE DATE: Friday, May 23 at 5PM.