DATA LEVEL PARALLELISM

5/13/2009

DLP Warmup (DUE: Wed. 5/20/2009, 5PM)

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

Data Level Parallelism

Thanks to the TA, Marty Nicholes and Prof. Owens, for the prior project handout that I leveraged.

INTRODUCTION

In this project you will use the NVIDIA Compute Unified Device Architecture (CUDA) Graphic Processor Unit (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 G80 hardware. Commands to be entered will be shown in the *`courier italic font`*.

TOOLCHAIN

In this project we will be using NVIDIA's CUDA programming environment. This is installed on 12 machines in the lab (listed here). You are welcome to do the assignment outside of the lab on your own machine, but you would need to have both NVIDIA G80 hardware and the CUDA environment installed on your machine.

Remember the gotcha mentioned in class: 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. Execute the following two lines (or add them to the ~/.cshrc file):

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

NOTE: If the second command gives you an error that the variable is not defined, try the following: *setenv LD_LIBRARY_PATH /opt/cuda/lib*

CUDA documentation is in /opt/cuda/doc/. See the NVIDIA_CUDA_Programming_Guide_1.1.pdf.

I suggest that you build and run your programs in emulation mode (see section on Emulation Mode) in order to work out the bugs without worrying about having hardware, but for credit, you need to make it work on the hardware.

SAMPLE CODE

- flops.tar – Sample project tar ball
- sum.tar – Sample project tar ball

There are two sample projects. The first project, called flops (floating-point operations per second), must be installed first, and the OPTIONAL second project called sum is installed later. You do not have to work with the sum project, but it will be useful for the DLP project you will do next.

The first sample program, flops, calls the routine runTest() from main. The routine runTest performs the following main steps:

1. Initial of the Cuda GPU,
2. Create and start timer,
3. Allocate of host and GPU memory,
4. Copy 2 float values into GPU memory,
5. Call flops_kernel (kernel which executes on the GPU), which:
    a. performs a single add and multiple with the passed in values,
    b. writes result to output variable in GPU, and
    c. returns
6. Copy results from GPU memory back into host memory,
7. Determine run time, and
8. Clean up and exit.

The sum program is very similar, except the kernel performs an addition of two numbers from shared memory, and the main program includes a check on the result.

USING CUDA

Flops Code Install

The file flops.tar contains a directory tree for Cuda SDK projects, with a project called flops. First upload the flops.tar file into your home directory on a snakes system, then untar the directory tree by issuing the following command on a snake system:

tar –xf flops.tar

You should see the directory NVIDIA_CUDA_SDK appear in your home directory. You will find the sample project files for the warmup in this directory: NVIDIA_CUDA_SDK/projects/flops. The files are:

- Makefile
- flops.cu (the source file)

Make sure the sample compiles and runs before doing anything else. Use the following commands (assuming you are in the NVIDIA_CUDA_SDK directory):

make clean
make
bin/linux/release/flops

Sum Code Install (OPTIONAL)

The file sum.tar contains the project subdirectory tree for a Cuda SDK project called sum. First upload the sum.tar file into your NVIDIA_CUDA_SDK/projects directory on a snakes system, then you must cd into your NVIDIA_CUDA_SDK/projects directory and then untar the directory tree by issuing the following command on a snake system:

```
tar –xf sum.tar
```

You should see the directory sum appear in your projects directory. You will find the sample project files for the project in this directory and the files are:

- Makefile
- sum.cu (the main program source file)
- sum_gold.cpp (the test progam)
- sum_kernel.cu (the sum kernel)

Make sure the sample code compiles and runs before doing anything else. Use the following commands (assuming you are in the NVIDIA_CUDA_SDK directory):

```
make clean
make
bin/linux/release/sum
```

EMULATION MODE

Building for device emulation will allow you to debug your kernel with things like print statements, since in emulation mode your kernel executes on the host system. If you want to build for device emulation, then do the following (note the directory for the binary file is different):

```
make emu=1
bin/linux/emurelease/flops
```

In emulation mode, a kernel using multiple blocks will execute very slowly, since the blocks are executed sequentially on the host system. So, you can use something like the following to switch to a single block for emulation mode:

```
#ifndef __DEVICE_EMULATION__
   dim3  grid(256, 1, 1);
#else
   dim3  grid(1, 1, 1); // only one run block in device emu mode or it will be too slow
#endif
```

WARMUP PROBLEMS

Maximize FLOPS Achieved

Your goal is to modify the provided 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 that allows you to count 2 floating-point ops using a single instruction. If the compiler sees a multiply then an add in the form x = a*b+c, it'll compile to MAD(x,a,b,c).

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

SUM 1000 elements (OPTIONAL)

Again, this section is optional, but will save you time when you start the final DLP project. You need to modify the sum project, so that it will add up 1000 elements, instead of only 2. You need to determine a way to parallelize the code. Ideally, you would be able to use 500 threads for the first addition, 250 for the next addition, and so on. You should also remove the print statements, since they will become annoying very fast, with 1000 elements.

SUBMISSION

Pretty easy for the warmup. Use the SmartSite to turn in: **1)** source code for flops and optionally sum; **2)** a pdf file containing the flops results you achieved and optionally how you parallelized sum including the final results; 3) a text file named "README" that describes what you made changes on the source code.

DUE DATE: Wed. 20 May at 5PM.