DATA LEVEL PARALLELISM

5/20/2009

DLP Project (DUE: Wed. 6/3/2009, 5PM)

Write more serious 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 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.

You will probably want to consult the NVIDIA CUDA Programming Guide extensively for this assignment (NVIDIA_CUDA_Programming_Guide_2.1.pdf, also in /opt/cuda/doc).

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

2. If you are a bash user, replace the word `setenv` with `export` in your command.

CUDA documentation is in `/opt/cuda/doc/`. See the NVIDIA_CUDA_Programming_Guide_2.1.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 still need to perform the real tests on the hardware.

SAMPLE CODE

- flops.tar – Sample project tar ball (see warmup for description)
- sum.tar – Sample project tar ball (see warmup for description)
- branch.tar – Sample project tar ball (see below for description)

The branch project provides the framework to analyze branch effect on performance. The main routine is very basic, but the kernel, testKernel, branches based on threadId and calls one of two functions, called

bigfunctiona and bigfunctionb. Note that the two functions have the __device__ qualifier, which makes the functions callable on the GPU only.

PART 1 PERFORMANCE SCALING

In order to understand what type of hardware support is present in the GPU for parallel data processing, you need to utilize the program you created during the DLP warmup based on the flops sample code. For part 1 of the project, you need to see how much work you can get out of the GPU with various configurations of threads and blocks. The GPU has only so much hardware, so your goal is to determine where the hardware support starts to fall off, i.e. if you launch 10,000 threads you may not see much more performance than with 5,000 threads (numbers made up to protect the innocent).

Tasks:

1. Determine how performance in GFlops varies as a function of the number of threads per block, and
2. Determine how performance in GFlops varies as a function of the number of blocks.

PART 2 DATA ELEMENTS PER THREAD

The file sum.tar contains the project subdirectory tree for a CUDA SDK project called sum. If you have not installed and modified the sum project as specified in the optional section of the warmup, you must do that before continuing on in the project.

The goal of your sum project is to add 1024 elements together in a data parallel way. You should use shared memory to perform this sum to allow different threads to communicate. Make sure that the result from each sum step in each thread ends up affecting the final sum, or the compiler will throw out the code.

Tasks :

1. See how performance scales as you vary the amount of work per thread. In order to get interesting results, you need to launch many blocks (1024?) that are using the same kernel and data. Your main program will receive a final sum from each of the blocks.
2. Does the performance track with your expectations, given the GPU architecture? Explain why or why not.

NOTE: Make sure that you follow the sum example, and move all the data into shared memory, otherwise you will be bound by the bandwidth of reading out of global memory. Global memory should only be used to pass sums between blocks.

PART 3 BRANCH EFFECT ON PERFORMANCE

In this part of the project we are exploring the cost of a branch with respect to the GPU's SIMD architecture (for warps) and SPMD architecture (across warps). Recall that the 32 threads in a warp run in lockstep (SIMD), and diverging threads within a warp will cause the GPU to run the different paths of execution sequentially. To observe this behavior, we need to have a lot of work per thread, as we will see below.

First ftp the branch.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 branch.tar
```

You should see the directory named "branch" appear in your projects directory. You will find the sample project files for the project in this directory: NVIDIA_CUDA_SDK/projects/branch. The files are:

- Makefile
- branch.cu (the main program source file)

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/branch
```

The branch project is setup to give you the first data point; the performance when there is one branch made in the kernel. The branch in the kernel decides whether to call bigfunctiona or bigfunctionb. You need to extend the branch code, so that you can measure performance at all the other branch granularities: 256, 128, 64, 32, 16, 8, 4, 2, and 1. For example, in the case with 256 branches, you might consider code like the following in the kernel:

```
testKernel() {
  if (threadIdx.x == 0) {
    bigfunctiona();
  } else if (threadIdx.x == 1) {
    bigfunctionb();
  } else if (threadIdx.x == 2) {
    bigfunctionc();
  } else ...
}
```

So, you can see that you will need up to 256 unique big functions. The unique big functions can be created by shuffling the function calls in one of the sample bigfunctions we provide, ensuring that no two bigfunctions are the same:

```
__device__ float bigfunctiona()
{
return
(expf(sqrtf(exp2f(exp10f(expm1f(logf(log1pf(sinf(cosf(tanf(float(th
readIdx.x))))))))))));
}
```

These results should be displayed graphically, and you should provide some explanation of the results, based on your understanding of the GPU architecture.

One thing to note about `bigfunctiona`: the GPU has two different flavors of each function (look at the CUDA Programming Guide for more details), one flavor being lengthy in runtime and more accurate, one flavor being faster and less accurate. For instance, __sinf takes 32 cycles to complete, but (according to the documentation) sinf is "much more expensive". You may wish to only use the __ (faster) variants in your bigfunctions.

Tasks:

1. See how performance scales with branch granularity, from 1 to 256 branches.
2. Save the source code for the version of your code that does 256 branches for handing in.

SUBMISSION

Use the SmartSite to turn in: **1)** the working source code for the sum program, and the branch program (256-branch kernel), **2)** the report (see report guidelines), and **3)** a text file named "README" that describes what you made changes on the source code.

REPORT GUIDELINES:

- Your report should be targeted at a manager who has some understanding of the GPU architecture, but does not understand the interaction between the way the code is written and way the GPU performs. Remember to explain your methodology.
- Report Content:
  - Discuss how the flops performance varies with the number of threads, blocks, etc.
  - Discuss how you perform the sum operation with multiple threads.
  - Discuss the performance characteristics of sum, as you vary: the number of elements per thread, the number of threads per block, and the number of blocks.
  - Discuss how performance with sum varies when assigning more work per thread and fewer threads vs. less work per thread and more threads.
  - Discuss the performance impact of branches on the branch project. How does the performance vary as branch granularity changes?
- The report should be no more than 3 pages in PDF format with 1 inch margins and no smaller than 10 point type.
- You are encouraged to include graphs as mentioned above to better illustrate how your experimental results and support your conclusions. However, don't feel like you have to submit every single graph. You should definitely use a few graphs to underscore important points that you are making.
- Organize your report in a way that is easy to follow.
- Do not just write it as a sequence of responses to each section! Put the problem in context, have a conclusion, and so on ... everything you've learned about writing an essay in English class applies here.
- Be sure you have thoroughly read the assignment and include all information it asks for.

- **DUE DATE: Wednesday 6/3 at 5PM.**