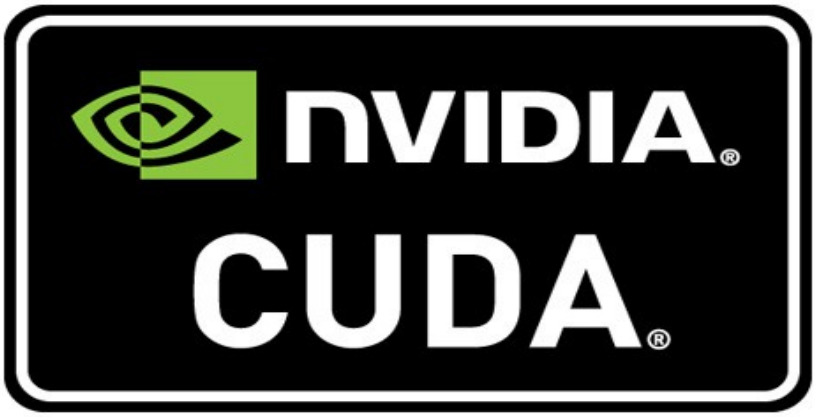
DATA LEVEL PARALLELISM



05/23/17

DLP Project (DUE: Thu. 6/8/2017, 10AM)

Write more serious 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 understanding the GPU memory hierarchy and its effect on performance.

TOOLCHAIN

In this project we will be using NVIDIA's CUDA programming environment. This is installed on the Linux machines in the labs. 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.

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.

As covered in the warmup, 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 the ~/.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 NVIDIA_CUDA_Programming_Guide_3.0.pdf.

BEFORE YOU START

A simple CUDA program (device.cu) is provided on Canvas to query the name and properties of the card you are using. Please use the following command to compile and run it. This will be especially helpful if you do the project on a machine besides those in the lab.

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

SAMPLE CODE

- nvidia-sdk-eec171.tar - Sample sdk tar ball (used in warmup, contains flops project)
- regs.tar – Part 1 sample project
- reduce.tar – Project used in Part 2 and 3

The regs project provides the framework to analyze how register usage affects performance. In the given source code, the kernel (testKernel), reads a data element and computes the final results using a varying number of registers and work.

The reduce project will give you experience with data-parallel algorithms, as well as allow you to explore the memory hierarchy in GPUs.

PART 1 PERFORMANCE SCALING

First download the project regs.tar from Canvas into the GPU/C/src/ directory you created during the warmup. Then unarchive the project using the following command. This creates a new directory, regs, which contains the source code in regs.cu. This is the file you will work with in Part 1.

```
tar xvf regs.tar
```

Here we are interested in finding out how performance scales with register usage. Modify the regs source code so that it does the same math/work but uses more/fewer registers. 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 use 6 registers per thread vs 40 registers per thread (just an example).

Objectives:
1.Determine how performance scales when total number of registers used are varied. You must find a way to compute the same final result (with equal work), however, using less registers.

To compile and run regs, move into the GPU/C directory like you did in the warmup, then run the following commands. The bolded flag following the `make` command tells the NVCC compiler to report how many registers the compiled code will use.

Note 1: running make from the GPU/C directory will compile all projects in the GPU directory tree. You are only interested in the register results for the regs project, so look at the output carefully.
Note 2: the compiler will return register counts for 2 GPU microarchitectures: sm_10 and sm_20. The Quadro 600 GPUs in the ECE lab are sm_20 devices, so use the register count for sm_20.

```
make ptxas=1
./bin/linux/release/regs
```

PART 2 PARALLEL REDUCTION

This next part uses the reduce.tar project you can download from Canvas. The sample program copies an array of 16K numbers to the GPU, then launches a single thread on the GPU that calculates the sum of 16K numbers serially. Your job is to modify the code so that a block of threads does the 16K additions in parallel. 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. Your program should store intermediate calculations in global memory as necessary.

You will need to consider the following issues in your solution, and you will discuss them in your report.

- How will you split up the work between threads?
- Do threads need to synchronize with other threads between different stages of the computation?

- Does the block size affect the performance of your solution? How many threads give the best performance?

Objectives:
1.In your report, you will explain how you parallelized the reduce program, mentioning how you approached the above issues as appropriate.
2.Once you have a single block working correctly, try launching more blocks running the same kernel and data. Note that due to race conditions in global memory, the sums that are returned may not exactly match the reference; this is okay. Try to max out the FLOPS and memory bandwidth of your reduce program. Report these figures in your write-up.

PART 3 GPU MEMORY HIERARCHY

In this part, you will modify your reduce program from Part 2 to store intermediate results in shared memory instead of global memory. The input data and output results should still be in global memory. Recall that shared memory can be accessed only by threads within the same block. This is in contrast to registers (which are only accessible to the local thread) and global memory (which is accessible by all threads in all blocks). Shared memory can be allocated either statically or dynamically. You will need to check the CUDA documentation to determine how to declare and use shared memory within a kernel.

Objectives:
1.After modifying reduce to work with shared memory instead of global memory, try launching more blocks running the same kernel and data. How does the performance of the shared memory version compare to the global memory version? Can you explain why?

Hand in the source code you used in step 1, 2 and 3.

SUBMISSION

Use Canvas to turn in: **1)** the working source code for the regs program and the 2 versions of the reduce program from Parts 2 and 3, and **2)** the report (see report guidelines).

DUE DATE: Thursday 6/8 at 10AM.

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 performance varies as register usage varies. Explain your results and show a chart/graph of performance vs. register usage (use the ptxas=1 option during compile time to get register count).
  - Discuss how you parallelized the reduce operation.
  - Discuss how performance of the reduce program varies when assigning more work per thread and fewer threads vs. less work per thread and more threads.
  - Compare the performance of the reduce program using global memory to shared memory.
- 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 several graphs as mentioned above to better illustrate how your experimental results and support your conclusions.
- Be sure you have thoroughly read the assignment and include all information it asks for.