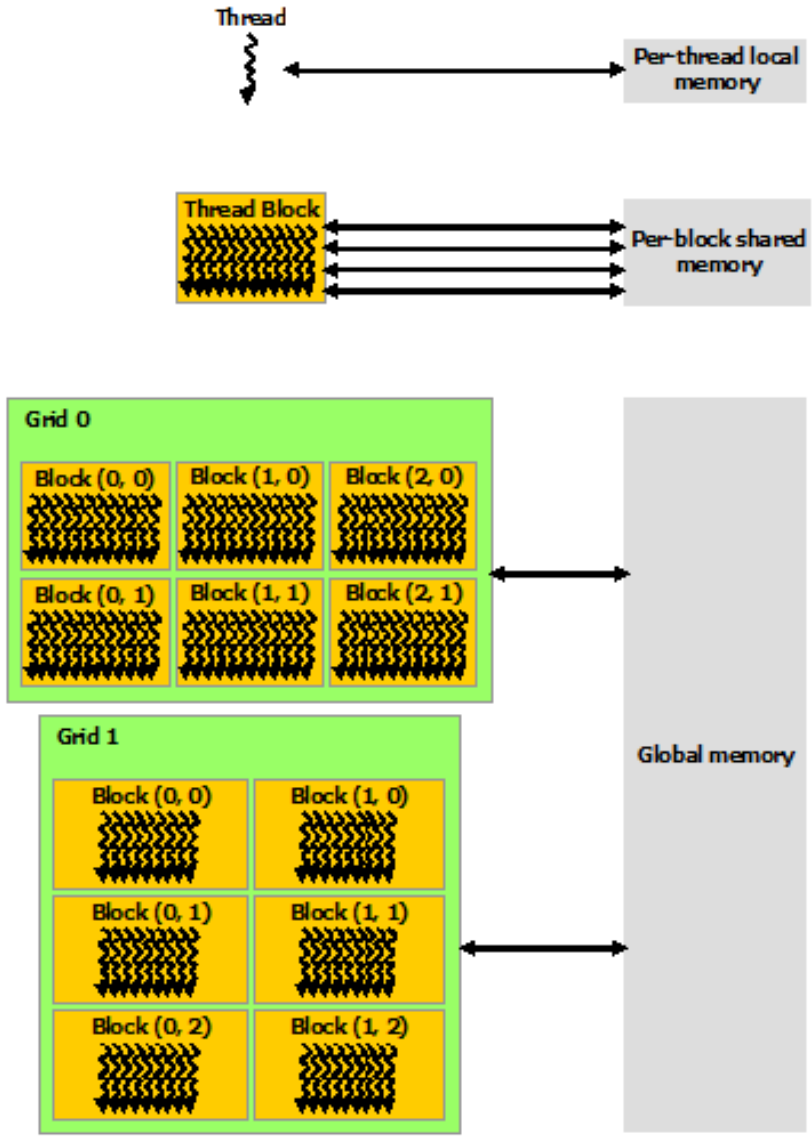
Matrix Multiplication, Data Parallelism, and the smart use of memory Hierarchy



Overview

In this project you will use the NVIDIA CUDA GPU programming environment to explore data-parallel hardware and programming environments. We will do this by 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. The linux machines in lab have new Quadro K620 devices. You'll likely get better numbers from them compared to your own laptop. 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. Important 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 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 for more information.

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

For the lab machines you should get a the following:

```
CUDA Device Query...
There are 1 CUDA devices.

CUDA Device #0
Major revision number:         5
Minor revision number:         0
Name:                          Quadro K620
Total global memory:           2097414144
Total shared memory per block: 49152
Total registers per block:     65536
Warp size:                     32
Maximum memory pitch:          2147483647
Maximum threads per block:     1024
Maximum dimension 0 of block:  1024
Maximum dimension 1 of block:  1024
Maximum dimension 2 of block:  64
Maximum dimension 0 of grid:   2147483647
Maximum dimension 1 of grid:   65535
Maximum dimension 2 of grid:   65535
Clock rate:                    1124000
Total constant memory:         65536
Texture alignment:             512
Concurrent copy and execution: Yes
Number of multiprocessors:     3
Kernel execution timeout:      Yes
```

If you have more than one graphics card in you machine it will report on all cards.

SAMPLE CODE

You can download the following code from canvas
`device.cu`

`matrixMul.cu`

The device.cu file mentioned earlier can be used to give you information about the GPU in your machine that may be useful later. The matrixMul file contains a simple matrix multiplication for both cpu and GPU it will be used as a starting point for your work. Read over the file and make sure you understand how it works.

PART one:

Without using the GPU perform a matrix multiplication of two square million element Matrices. The matrices are initialized with incrementing numbers. How long does it take to perform the multiplication with CPU only? What is the performance in GFLOPS?

The sample code provided also contains a simple CUDA matrix multiplication. Without modification what is its performance? Can you improve the performance by modifying the number of blocks and thread? Remember the full matrix multiplication should still be performed. Blocks*Threads should be a constant. Explore the space of blocks and thread. What is the optimal combination of thread/blocks for your device.

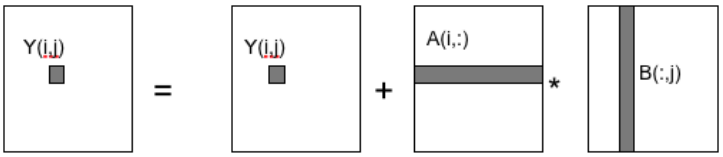
Note: The command line to compile the code as including the needed .h is as follows:

```
nvcc -I /opt/cuda/samples/common/inc/ matrixMul.cu
```

Part two:

In part one both matrices are simply copy to global memory on the GPU. Nvidia graphics cards all have shared memory. This memory is visible to all threads in a block. It is also closer to the operating hardware and much faster to access. Your next assignment is to use of shared memory to improve performance of the matrix multiplication. The speed up here comes from the fact that an element read into shared memory will be available to all thread without being read again from global memory.

To envision how this might be done; imagine Y, the product of matrices A andB, to be comprised of sub blocks. A single block of Y can be computed using a smaller subset of the rows of A and the columns of B. Both the subset, of A, B and Y could be stored entirely in shared memory. Accomplish this by modifying each kernel to read a different subset of A and B to shared memory, synchronizing all threads, and then calculating the an element in the sub-block Y.



As you do this, you should calculate how much of each matrix you can fit into shared memory. How much space will the sub-block of Y take? This information can be used to pick starting values for you to explore thread and block count variation. Report your discoveries and performances.

Part Three:

Recently NVIDIA began to offer a Unified memory model which abstracts away the fact that the GPU and CPU do not share memory. This model allows for allocation of memory which the program can access from either the CPU or GPU without the users actually considering which part of the machine currently has ownership of data or copying it back and forth.

Modify your code to use the simper memory model. (You can use the function cudaMallocManaged() to allocate memory in the unified model). Does it effect performance?

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. Include methodology of exploration of for parts one and two, along with any graphs you feel are enlightening. Discuss how you developed your code for part two. Answer all questions asked in the document along with appropriate discussion. You are encouraged to include any graphs that better illustrate how your experimental results support your conclusions.