Programming Assignment #1

- DUE 5PM Friday, January 30
- Turning in assignment:
  - Use the "handin" program on the CADE machines
  - Use the following command:

    "handin cs6963 lab1 <prob1file> <prob2file>"
  - The file <prob1file> should be a gzipped tar file of the CUDA program and output for Problem 1
  - The file <prob2file> should be a gzipped tar file of the CUDA program and output for Problem 2

How to Develop and Run CUDA programs in Lab 6 (WEB 130)

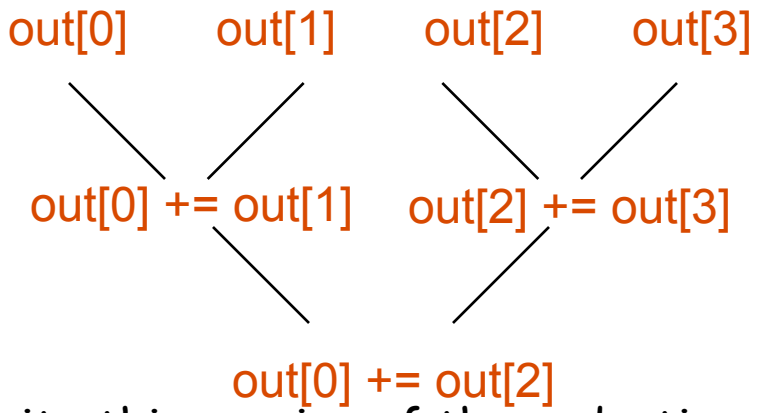
- Run Visual Studio 2005 (not 2008)
- Select "File➔New Project"
- In the window, choose "CUDA WinApp"
- After this, create CUDA programs with .cu extensions

NOTE: Any machine running CUDA 2.0 can be used for this assignment, but this lab has the latest NVIDIA cards in it.

Programming Assignment #1

Problem 1: Reduction

In the "count 6" example using synchronization, we accumulated all results into out_array[0], thus serializing the accumulation computation on the GPU. Suppose we want to exploit some parallelism in this part, which will likely be particularly important to performance as we scale the number of threads. A common idiom for reduction computations is to use a tree-structured results-gathering phase, where independent threads collect their results in parallel. The tree below illustrates this for our example, with SIZE=16 and BLOCKSIZE=4.



Your job is to write this version of the reduction in CUDA. You can start with the sample code, but adjust the problem size to be larger:

```
#define SIZE 256
#define BLOCKSIZE 32
```

Programming Assignment #1

Problem 2: Computation Partitioning & Data Distribution

Using the matrix addition example from Lecture 2 class notes, develop a complete CUDA program for integer matrix addition. Initialize the two input matrices using random integers from 0 to 9 (as in the "count 6" example). Initialize the ith entry in both a[i] and b[i], so that everyone's matrix is initialized the same way.

Your CUDA code should use a data distribution for the input and output matrices that exploits several features of the architecture to improve performance (although the advantages of these will be greater on objects allocated to shared memory and with reuse of data within a block).

- a grid of 16 square blocks of threads (>= # multiproc)
- each block executing 64 threads (> warpsize)
- each thread accessing 32 elements (better if larger)

Within a grid block, you should use a BLOCK data distribution for the columns, so that each thread operates on a column. This should help reduce memory bank conflicts.

Return the output array.