Matrix Multiplication with CUDA — A basic introduction to the CUDA programming model Assessment

Robert Hochberg

April 5, 2012

Assessment of Student Knowledge

1. **Understanding the Theory:** A student who can answer the **Test of Concepts** question as well as the **Practice 1** and **Practice 2** questions in the text has likely understood the idea of counting paths in graphs, matrix multiplication, and how they are related. No CUDA knowledge could be inferred, though.

2. **Basic Understanding of CUDA:** Students who can solve **Largest Matries** and **Test of CUDA Acuity** have mastered the basic ideas of programming in CUDA. Students who successfully solve **Explorations with Pascal's Triangle** problem have also demonstrated proficiency at C programming and the idea of putting jobs on the card for the card to solve. To test the ability to write original programs, it is recommended that you continue with the module "Dynamic Programming with CUDA - Part I" freely available at the same site as this module.

3. **Comfort with the Environment:** Students who can work through **Practice 3** in the text and **Largest Matrices** are probably comfortable with the modify-compile-run cycle of development.

Solutions

1. **Test of Concepts:** Here is the matrix showing the number of paths of length 4 between the pairs of vertices of a graph:

$$\begin{pmatrix} 9 & 3 & 6 & 11 & 1 \\ 3 & 15 & 8 & 7 & 11 \\ 6 & 8 & 8 & 8 & 6 \\ 11 & 7 & 8 & 15 & 3 \\ 1 & 11 & 6 & 3 & 9 \end{pmatrix}$$

   - How many vertices does the graph have?
   - Assume that the vertices of the graph are labeled A, B, C, ... and that the rows and columns are given alphabetically. Show that there are 208 paths of length 8 from A to B in that graph.

> The number of vertices in the graph is the same as the number of rows and columns of the matrix, which is 5. To find the number of paths of length 8, simply square the"paths of length 4" matrix. The result is shown below.
>
> $$\begin{pmatrix} 248 & 208 & 220 & 336 & 120 \\ 208 & 468 & 324 & 340 & 336 \\ 220 & 324 & 264 & 324 & 220 \\ 336 & 340 & 324 & 468 & 208 \\ 120 & 336 & 220 & 208 & 248 \end{pmatrix}$$

2. **Largest Matrices:** The program 'deviceQuery' is included with the CUDA SDK. It gives you specifications on the CUDA-enabled card(s) installed in your system. Run this program to find out how much RAM your card has, and use this to determine the theoretical largest value N for which you can multiply two $N \times N$ matrices on your card, and hold the product. Then compare this theoretical value with the actual largest N for which `multShare N N N` returns with no errors.

> It is the amount of global memory that restricts the sizes of the matrices that you can multiply. For square matrices of size $N \times N$ with integer entries, you'd need (Number of matrices) * (Entries / Matrix) * (Bytes / Entry) $= 3 \times N^2 \times 4 = 12N^2$ bytes of memory. So on my Mac, for example, running an NVIDIA GeForce GT 330M with 256MB of memory, I could multiply square matrices of size $\sqrt{256,000,000/12} \approx 4618$.

3. **Test of CUDA Acuity:** Modify the code for the shared memory version of the program so that it works even if the dimensions of the matrices are not multiples of the size of the thread blocks. Use the code for the no-shared memory version as a model.

   Since each kernel thread is responsible for computing one element in the product matrix, the only modification needed is to add a test at the start of each thread whereby the thread tests whether it lies within the matrix or not. For example, if the block size is 16×16 and we are multiplying matrices of size 50×50, then we'd launch a 4×4 grid of blocks. At startup, each thread will check which element of the product matrix it corresponds to. For example it would evaluate `blockIdx.x * blockDim.x + threadIdx.x` to find which column it belongs to, and if it is 50 or greater, the thread will terminate immediately. A similar computation would be done for the row. The result is that only those threads corresponding to matrix elements will perform their calculations and save to the product matrix in global memory.

4. **Explorations with Pascal's Triangle** The matrix on the left below contains the entries of Pascal's Triangle. The left column and the main diagonal consist of ones, all entries above the main diagonal are zeros, and each other entry is the sum of the entry directly above it and the entry diagonally above and to the left. The matrix on the right is the same, except that the diagonals are alternately positive and negative.

   - What is the product of the two matrices given here?
   - Modify `multShare.c` so that it generates larger versions of these two matrices and verify that the program produces the correct product.
   - How large can your matrices get while still using integers?
   - modify the code to use long ints or doubles, so that you can multiply larger matrices. Now how large can your matrices be and still produce the correct result?

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 3 & 3 & 1 & 0 & 0 & 0 & 0 \\ 1 & 4 & 6 & 4 & 1 & 0 & 0 & 0 \\ 1 & 5 & 10 & 10 & 5 & 1 & 0 & 0 \\ 1 & 6 & 15 & 20 & 15 & 6 & 1 & 0 \\ 1 & 7 & 21 & 35 & 35 & 21 & 7 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 3 & -3 & 1 & 0 & 0 & 0 & 0 \\ 1 & -4 & 6 & -4 & 1 & 0 & 0 & 0 \\ -1 & 5 & -10 & 10 & -5 & 1 & 0 & 0 \\ 1 & -6 & 15 & -20 & 15 & -6 & 1 & 0 \\ -1 & 7 & -21 & 35 & -35 & 21 & -7 & 1 \end{pmatrix}$$

The product of these matrices will always be the identity matrix, that is, the matrix having ones on the main diagonal and zeros everywhere else. With 64-bit integers or floats, it is possible to get to about the 66th row, depending on implementation.