Programming Assignment #5: CUDA

CS222 Spring 2023
10 points
due Saturday, May 6th, 11:59 pm

CUDA Programs

You'll write two CUDA programs. You may work with a partner, but graduate students must work individually.

Do not wait until the last minute! The use of the systems in the VACC is unpredictable, and there could be a surge in use at the end of the semester!

Or, if you have a system with an Nvidia GPU, you can download and install the CUDA development environment and do the work on your own system.

Update: I've thought about this, and I've decided that it's not reasonable to ask you to write the second program in the time we have left. It uses a concept called pitched memory allocation, which is a little tricky, and which we will only talk about in the last week of classes. Do the first program (the dot product). Do the second program if you'd like, for extra credit. Grad students are required to do the second program, but they don't have to do the third program (the matrix multiplication).

1. Dot product

Do the dot-product example from class: compute the dot product of two `float` vectors of length N. Calculate the elapsed time for the GPU computation and for the same computation on the CPU. Calculate the GPU elapsed time two ways: not including the memory copies, and then including the memory copies. Set the values of the u and v vectors to random numbers in the range $(0, 1)$. (Use the Linux `drand48()` function.)

Implement this function:

```
__global__
void dotp( float *u, float *v, float *partialSum, int n );
```

Use 256 threads per thread block and 256 blocks.

Accumulate each thread block's result in shared memory, and do parallel reduction on the shared memory, as described in the lectures.

Also, compute the relative error. The relative error between two scalar values x and y is $|x - y| \,/\, |x|$, with $x \neq 0$. (Compute the relative error on the CPU.)

Do this for $n = 256 * 256$.

You do not need to use a grid-stride loop.

Graduate students: do this with a grid-stride loop. Use $n = 256 * 256 * 256$ and use 256 blocks, with 256 threads per thread block.

Undergraduates may do this for a little extra credit.

2. Matrix-vector product

Implement a matrix-vector product function:

```
__global__
void MxV( float *M, float *x, float *y, size_t pitch, int n);
```

This will do the calculation $y = Mx$, given the $n \times n$ matrix M and the $n \times 1$ vector x.

Each thread will do a dot product: thread i will do the dot product of row i with the vector x and put the result in $y[i]$. (This is a separate program altogether from the first program–you're not using your dot-product function here.)

Don't use a grid-stride loop: assume that n is less than #blocks $\times$ #threads per block. Use 256 threads per thread block, and pick a value of n such as $n = 5000$. Use pitched memory allocation for M.

You'll still use a one-dimensional grid for the GPU computation: each thread only cares about its row number. But because you are using pitched allocation, you'll have to pass the `pitch` variable to the kernel, and the kernel will have to take the pitch into account, as shown in the lecture notes.

(This matrix-vector multiplication will require a huge number of floating-point operations: each entry in y will require n multiplications and n additions; so the total number of multiplications will be n^2, and the total number additions will be n^2.)

Calculate the elapsed time for the GPU computation and for the same computation on the CPU. Calculate the GPU elapsed time two ways: not including the memory copies, and then including the memory copies. Set the entries of M and of x to uniformly distributed random numbers in the range $(0, 1)$.

Write a function to do the matrix-vector product on the CPU, and compute the relative error between your two solutions. The relative error between two vectors u and v is given by

$$relerr = ||u - v|| \,/\, ||v||, \tag{1}$$

assuming v is not zero. $||\cdot||$ is the norm of a vector; the two-norm $||u||$ is given by $||u|| = (\sum_1^n u_i^2)^{1/2}$.

When I run, I see about a 4x speedup with $n = 5000$, and my relative error is on the order of 1×10^{-8}.

Graduate Students

Implement matrix multiplication: the kernel to multiply two $n \times n$ matrices is straightforward:

```
// computing C = A*B
float sum = 0.0;
int col = threadIdx.x + blockDim.x * blockIdx.x;
int row = threadIdx.y + blockDim.y * blockIdx.y;
int numEltsPerRow = pitch / sizeof(float); // #elements in each padded row
if (col < n && row < n) {
  sum = 0.0;
  for (k = 0; k < n; k++)
    sum += A[row * numEltsPerRow + k] * B[k * numEltsPerRow + col];
  C[row * numEltsPerRow + col] = sum;
}
```

This kernel computes a single matrix element in the product.

And as long as $n <= 16$, all of this will fit in a single thread block:

```
dim3 dimGrid(1, 1);
dim3 dimBlock(n, n);
mxmKernel<<<dimGrid, dimBlock>>>( d_A, d_B, d_C, pitch, n);
```

Write a CUDA program to do this. Give A and B random values in the range $(0, 1)$. Used pitched memory allocation. Read the description of pitched memory allocation in the lecture notes carefully.

Compute the relative error between the CPU solution and the GPU solution. The relative error between two matrices A and B is given by

$$relerr = ||A - B|| \,/\, ||A||, \tag{2}$$

assuming $||A||$ is not zero. The norm of a matrix is given by $||A|| = (\sum_1^n \sum_1^n (a_{i,j})^2)^{1/2}$. (This is the Frobenius norm.)

You'll do the error computation on the CPU, after you call the GPU kernel. You'll also have to do the matrix multiplication on the CPU, so that you have a result with which to compare the GPU result.

Then, do this with blocking. Pick a block size $TILESIZE$ such that $n/TILESIZE \leq 16$, and use 256 threads per thread block. Try this with $n = 16, 64, 256$; and make a table comparing CPU runtimes to GPU runtimes for each of these values of n. It's OK to omit the memory copying from the GPU timings. Undergraduates may also do this for a little extra credit.

Now, each thread block will do the multiplication for a submatrix of with dimensions $TILESIZE \times TILESIZE$. The thread at position i, j in the output array still needs to do the dot product of the full i^{th} row of X and the j^{th} column of Y.

So for example, if I have problem size `n=256`, then I must pick `TILESIZE` to be greater or equal to $256/16 = 16$.

Here's how to set this up for blocking:

```
int threadsPerBlock = 256;
assert((n/TILESIZE)*(n/TILESIZE) <= threadsPerBlock);
dim3 dimGrid(n/TILESIZE, n/TILESIZE);
dim3 dimBlock(TILESIZE, TILESIZE);
printf("numBlocks = %d %d\n", dimGrid.x, dimGrid.y);
printf("threadsPerBlock = %d %d\n", dimBlock.x, dimBlock.y);
assert((N/TILESIZE)*(N/TILESIZE) <= threadsPerBlock);
```

Developing and Testing

Use either the computing resources in the VACC or use your own system. See the *CUDA How-To* slides for a description of how to compile and run programs on the VACC machine.

If you use your own system, then get the program `get-device-info.cu` from the course gitlab, compile and run it, and save the output and submit it with your code.

What to Submit

Submit your source code. (No graphs or tables necessary unless you did the third program.)

In addition, forward to me an example email notification that you receive from the slurm scheduler when your dot-product program ends and when your matrix-multiply program ends on the VACC cluster (as shown in the *CUDA How-To* lecture slides).