

Programming Assignment 4: Conjugate Gradients on a GPU

Parallel and Distributed Computing

Due: Wednesday, April 13
Note the change in due date

1 Introduction

Recall the method of conjugate gradients:

```
 $k = 0; x_0 = 0; r_0 = b$ 
while ( $\|r_k\|^2 > \text{tolerance}$ ) and ( $k < \text{max\_iter}$ )
     $k++$ 
    if  $k = 1$ 
         $p_1 = r_0$ 
    else
         $\beta_k = \frac{r_{k-1} \cdot r_{k-1}}{r_{k-2} \cdot r_{k-2}}$  // minimize  $\|p_k - r_{k-1}\|$ 
         $p_k = r_{k-1} + \beta_k p_{k-1}$ 
    endif
     $s_k = Ap_k$ 
     $\alpha_k = \frac{r_{k-1} \cdot r_{k-1}}{p_k \cdot s_k}$  // minimize  $q(x_{k-1} + \alpha p_k)$ 
     $x_k = x_{k-1} + \alpha_k p_k$ 
     $r_k = r_{k-1} - \alpha_k s_k$ 
endwhile
 $x = x_k$ 
```

Here, A is a symmetric, positive definite, n -dimensional matrix, and r_k, p_k, s_k, x_k, b , and x are n -dimensional vectors. The other variables are scalars. When the algorithm converges, it finds the solution to the linear system $Ax = b$.

When we parallelized this for programming assignment 2, we used a *data parallel* solution: we divided the data — the matrix and the vectors — among the MPI processes, and each process applied essentially the same instructions to its parts of the data.

Nvidia's API for GPGPU programming, CUDA, is ideally suited to data parallelism. So for programming assignment 4 you should write a CUDA program that implements the method of conjugate gradients for solving linear a symmetric positive definite system. For a brief introduction to using CUDA, see <http://www.cs.usfca.edu/peter/cs625/using-cuda.pdf>.

2 Parallelization

As with the MPI implementation of conjugate gradients, In order to parallelize conjugate gradients with CUDA, we just need to implement some operations from basic linear algebra: SAXPY¹, Dot-product, Matrix-vector product, and maybe one or two other operations. The remainder of the program can be carried out on the host. Thus, a program might begin by allocating and initializing storage on the device: storage for the matrix and the various vectors. Then the program can execute by having the host carry out scalar operations and calling kernels on the GPU when it needs to carry out some linear algebra.

With this approach, you should only need to copy nonscalar data between the host and the device at the beginning and at the end of the algorithm. Intermediate vector results should only need to be copied to the host for debugging.

3 Details

3.1 Input

Your program should take the same command-line arguments used by the MPI conjugate gradient program:

- The order of the system,
- The tolerance,
- The maximum number of iterations,
- An optional fourth argument: the character ‘n’ if the user wants to suppress printing of the solution vector.

The arguments will be given in this order on the command line. Note that the optional fourth argument should not suppress printing of all output: only the solution vector should be omitted; the other information — elapsed time, etc. — should always be printed.

After getting the command line arguments, and setting up storage on the host, the program should read the matrix A and the right-hand side b from `stdin`. For this program, the matrix can be stored as an ordinary dense matrix in the usual one-dimensional format. Our GPUs don’t have support for doubles. So you should use floats for the matrix, the various vectors, and the scalars that are used on the GPU.

After getting the input, you can allocate storage for A on the device and copy its contents from the host.

3.2 Taking Timings

After copying the matrix, you can start the timer. The reasoning here is that if the CG solver is part of a larger program, it’s likely that the input to the solver, the matrix A , will already be stored on the device when the solver is called. However, it may be the case that the various temporary vectors, x_k , p_k , s_k , and r_k , will not have storage allocated. If this is the case allocation, initialization and freeing of these data structures should be included

¹SAXPY is the single-precision version of DAXPY: our GPUs only provide single-precision floating point.

in the timing. Of course, it is possible that these structures will already be allocated and initialized. So you may want to add a second timer that doesn't include this part of the code. (This is not required, but if you do use a second timer, be sure it's clear in your output which time is which.)

After the solver terminates, you can stop the second timer, but don't stop the first timer until after the temporary storage has been freed. Before taking the final time(s), be sure that the device has actually finished execution by calling `cudaThreadSynchronize()` on the host.

Cuda provides a class called `cudaEvent_t` and objects of this class can be used for timing. However, I've found that the results they give can be unreliable on some systems. So I recommend that you use `timer.h` and the macro `GET_TIME`. Note that the argument to `GET_TIME` should be a double, *not* a float. This shouldn't cause any problem, since the timing will be done on the CPU, not the GPU.

3.3 Threads

For each kernel call your program should start a total of at least n threads, where n is the order of the system. So you can assume that n will be no larger than the maximum number of threads the device supports. The threads can be organized into blocks in any way you want, subject, of course to the restrictions of our systems — e.g., no more than 512 threads in a block. You can also start different numbers of threads for the different kernels. For example, it may make sense to start more threads for the matrix-vector multiplication than for SAXPY or Dot.

As a rule of thumb, you should use at least as many blocks as there are multiprocessors. The number of multiprocessors can be obtained with the following code run on the host:

```
cudaDeviceProp props;  
  
cudaGetDeviceProperties(&props, 0);  
int min_blocks = props.multiProcessorCount;
```

The second argument to `cudaGetDeviceProperties` is the device number. Since all of our systems only have one device, this will always be 0 for us.

3.4 Output

You should print all the output that you printed for the MPI implementation of CG:

- The number of iterations,
- The elapsed time (can be two numbers here — see above),
- The solution vector (unless the user included the 'n' command line option),
- The square of the norm of the residual calculated by CG,
- The square of the norm of the residual calculated from the definition,

In addition you should print

- The number of blocks and the number of threads per block for each kernel.

4 Program Development

Since the heart of the program is the operations from linear algebra, it would be a good idea to start by writing independent CUDA programs that carry out these operations: a program that carries out a SAXPY, a program that implements dot-product, etc. Once these are fully debugged and optimized, you can be fairly confident that when they're incorporated into the conjugate gradients program, they won't introduce new bugs.

As we noted in the Brief Guide, debugging CUDA programs can be painful. However, the conjugate gradients program should be relatively easy to debug since

- You'll be developing the most important parts of the program as separate small programs, and
- The structure of the conjugate gradients algorithm and the use of CUDA provide a natural organization for the examination of intermediate results.

To understand the second bullet observe that after calculating an intermediate result — e.g., calculating p_k — you can add an `#ifdef DEBUG . . . #endif` block in which the result of interest is copied back to the host and printed. So it should be relatively easy to identify problems.

Note that the use of floats instead of doubles in the data structures will have (at least) two effects on the CUDA program:

1. Arithmetic with floats is faster than arithmetic with doubles. So the CUDA CG program may seem faster than a serial CG program, when it might actually be slower if both used floats.
2. The precision of calculations with floats is much lower. So, for example, it may be difficult or impossible to achieve a tolerance that you were able to achieve with the program that used doubles.

5 Reduction

The implementation of SAXPY and matrix-vector multiplication with n threads and n -dimensional vectors is quite straight-forward in CUDA: each thread can be responsible for computing one component of the result vector. On the other hand, dot product involves a reduction whose implementation is less obvious.

The simplest solution is to use one of CUDA's atomic operators, `atomicAdd`:

```
float tmp;
int i = blockDim.x*blockIdx.x + threadIdx.x

if (i < n) {
    tmp = x[i]*y[i];
    atomicAdd(dot_p, tmp);
}
```

The idea here is that we’re forming the dot product of the vectors x and y . We start by forming the product of `x[i]` and `y[i]` on each thread. Then each thread adds its part of the dot product into the shared location pointed at by `dot_p`. The function `atomicAdd` does the addition “atomically.” That is the load, update, and store of `*dot_p` is executed as if it were a single operation so that there won’t be a race condition. When the dot product is complete, the result can be copied from the device to the host.

Adrian pointed out that `atomicAdd` with floating point arguments isn’t available for our Nvidia cards. Fortunately, the CUDA Programming Guide gives an implementation in terms of other functions. So you should use `atomicAddf`, which can be downloaded from the class website.

6 Optimization

The implementation of `Dot` that we just discussed isn’t very efficient, since it effectively serializes access to the shared variable `dot_p`. If n is large, there are *much* better methods. For example, we can use a binary tree structure to reduce the number of stages from n to $\log_2(n)$. This requires the use of the synchronization function `__syncthreads()` when the additions are carried out within a block of threads, and, as we noted in class, in order to synchronize across different thread blocks it’s necessary to return from a kernel and start a new kernel. Similar ideas can be used for the individual dot products in matrix-vector multiplication. Another possible optimization is to try to use shared memory instead of global memory for at least parts of the vectors.

On the other hand, you should keep in mind that the order of the vectors we can use is fairly small — probably less than 10,000 — so it may be that on shorter vectors, it might be faster to simply serialize access to memory locations.

Although once the algorithm has started, there should be very little communication between host memory and device memory, it might be advantageous to use page-locked mapped memory to reduce the cost of the initial and final data transfers.

Adrian also pointed out that the CUDA shared libraries may not be initialized before a kernel is run. So another optimization might be to start a “dummy” kernel before starting the code that actually carries out the CG method.

None of these optimizations is required. Indeed, they may not improve overall performance. However, they should be checked if you’re interested in getting the extra credit.

7 Documentation

Follow the standard rules for documentation. Describe the purpose and algorithm, the parameters and variables, and the input and output of each routine.

Be sure to carefully document any optimizations you’ve made to the basic algorithm.

8 Grading

Correctness will be 80% of your grade. Does your program correctly implement the conjugate gradient method for CUDA? Does it accept input and command line options in the required format? Does it output the correct information?

Quality of solution will be 15% of your grade. Is the implementation reasonably efficient? Is the program well-designed and easy-to-read?

Static features such as source format and documentation will be 5% of your grade?

The fastest correct program will get 10 points extra credit. The next two programs will get 5 points each.

9 Collaboration

You may *discuss* all aspects of the program with your classmates. However, you should never show any of your code to another student, and you should never look at anyone else's code — regardless of its source.

10 Submission

The program is due on *Wednesday, April 13*. By 10 am, you should have copied your source files, makefiles, and any additional files to your `cs625/prog4` SVN directory. Put a hardcopy of your source code in my mailbox in Harney 545 by 2 pm.