GPU Accelerated Preconditioned Conjugate Gradient

Alex Guldemond

June 2019

1 Introduction

The purpose of this project was to demonstrate the usage of a single GPU to accelerate the computation of the unique solution to the linear equation $Ax = b$, where A is a symmetric positive definite, sparse matrix, and b is a nonzero real vector. All solutions were determined using the preconditioned conjugate gradient (PCG) algorithm, and all matrix operations were accelerated using a GPU. The PCG algorithm is as follows:

Algorithm 1 Preconditioned Conjugate Gradient

Require: A, M, b, ϵ
Ensure: $A > 0, M > 0$
  $x_0 = 0$
  $r_0 = b - Ax_0$
  $z_0 = M^{-1}r_0$
  $p_0 = z_0$
  $k = 0$
  while $r^t r > \epsilon$ **do**
    $\alpha_k = \frac{r_k^t r_k}{p_k^t A p_k}$
    $x_{k+1} = x_k + \alpha_k p_k$
    $r_{k+1} = r_k + \alpha_k A p_k$
    $z_{k+1} = M^{-1} r_{k+1}$
    $\beta_k = \frac{z_{k+1}^t r_{k+1}}{z_k^t r_k}$
    $p_{p+1} = z_{k+1} + \beta_k p_k$
    $k = k + 1$
  end while

The final result is the solution to $Ax = b$. In order to take advantage of the GPU, a preconditioner was needed whose inverse could be directly found and applied, as opposed to one whose inverse requires a sequential algorithm such as forward and backward substitution. For that purpose I chose a sparse ap-

proximation of symmetric successive over relaxation preconditioner. The SSOR preconditioner is given by

$$M = \frac{w}{2-w}(\frac{D}{w} + L)D^{-1}(\frac{D}{w} + L)^T$$

where D and L are the diagonal and lower triangular parts of A respectively, and w is the relaxation parameter between 0 and 2. An approximation of the inverse of M that has the same sparsity pattern as A is given by

$$M^{-1} = w * (2-w)(I - wLD^{-1})^T D^{-1}(I - wLD^{-1})$$

This approximation can be computed directly, and thus the same GPU acceleration can be applied to matrix vector products of M^{-1} and A.

The above algorithm was implemented serially, using no parallelism whatsoever, and with GPU acceleration on all linear algebra operations using CUDA. The end result was that the GPU accelerated version ran significantly faster then the serial version. All CUDA kernels were implemented specifically for this project; no external linear algebra packages were used.

2 Methodology

In order to get the most performance out of the GPU, all initialization and significant computation was done one the GPU. As can be seen in include/LinearSolve.hpp, the flow of control returns to the CPU only to compute scalar values and to perform block level synchronization of the kernels. Since CUDA only provides synchronization at the the level of threads withing blocks, this was necessary in order to ensure the correctness of the computation. However, only scalar values are copied in and out of the GPU memory in order to check the termination criteria, all other computation happens on the GPU side.

The two most critical kernels used are a sparse matrix vector product, and a vector dot product. All kernels used can be seen in include/Parallel.cuh. Since all of the matrices dealt with by this application are sparse, matrices are stored using the compressed sparse row format. The kernel sparseMatrixVectorProduct breaks each CUDA block into thread warps consisting of 32 threads each. Each thread warp is responsible for handling one entry of the final vector. Each thread in the warp caches a single multiplication, then reduction logic is employed over the entire warp to add up the final result.

The dot product kernel works in a similar manner. Each thread over the entire grid first caches a single multiplication into shared memory. A reduction argument is then employed across the entire block to cache the sum of that block's entire cache. That sum is then atomically added to the final dot product value.

For the algorithm to run quickly, it was important that these two kernels run as efficiently as possible. After profiling the code, it is seen that for a 32,768 dimensional problem, with kernels running at 32 blocks per grid, 1024 threads

per kernel, 87% of the algorithm was spent on matrix vector products, and 8% was spent on dot products, taking up the vast majority of the computation time.

The preconditioner inverse M^{-1} was also determined using the GPU. The above expression of M^{-1} can be expanded to find a formula for a given entry of M^{-1}. The kernel ssoraiEntries computes the entry array of M^{-1} by assigning each row in M^{-1} to a single thread. Perhaps a faster implementation would have been to break the blocks into warps as in sparseMatrixVectorProduct, and have each war determine a single row. However, since the preconditioner is only determined once at the beginning of the algorithm, it was not a high priority to accelerate this kernel as much as possible.

All of the major kernels were unit tested to demonstrate correctness. After building the project by following the directions in the readme, the unit tests can be run be executing the ctest command in the build directory, or can be manually run by executing any of the tests in the bin directory. Note that the tests will obviously fail if they are not executed on a GPU node.

One of the more difficult aspects of this project was writing the code in idiomatic C++ while using the very C like CUDA library. It was also necessary that all memory allocation and deallocation happen automatically, without having to explicitly call cudaFree. Furthermore, I wanted the PCG algorithm in LinearSolve.hpp to not require any duplication of code, regardless of whether or not the operations were being carried out on the GPU or CPU. The C++11 standard library provides a smart pointer called std::unique_ptr, which can take a custom deleter as a template argument to call specific deallocation logic. This, along with partial template specialization, were used to guarantee that all memory allocations were automatically deallocated, regardless of memory residing on the host or device. This can be seen by using cuda-memcheck with the leak check option.

Both matrix vector products and vector additions were imlemented to run lazily. This means that if A is a matrix and x is a vector, the product Ax will not actually be evaluated until it is used. This allowed the operations to be written in a more clear syntax without having to allocate any temporary memory. For instance, if these operations were not done lazily, the following code

```
a_direction = mat * direction;
```

would work by first allocating temporary storage for the result of $mat * direction$, then move it into a_direction, and finally delete the old value held by a_direction. However, if the code in DenseVector.hpp and SparseMatrixBase.hpp is examined, one can see that what happens instead is that the memory held by a_direction is instead directly used as the temporary memory to store the result, thus allowing us to skip allocating and freeing memory. Once this lazy evaluation was implemented, the amount of time spent allocating and freeing GPU memory was drastically reduced.

3 Results

The code was compile with both gnu and intel compilers. All tests pass with both comilers. The following results were obtained by running the code after compiling with the gnu compiler version 4.9. The main executable is bin/pcg. The output from all of the test runs can be found in the data directory. All runs were done solving the system $Ax = b$, where b is a vector of all ones, and A is a tridiagonal matrix with 2 down the diagonal and -1 on the off diagonals. Each test was run using single precision floating point arithmetic.

The first job was to find the optimum value of w, the relaxation parameter, which leads to the lowest number of iterations. Relaxation values were tested from 0 to 2, multiplying the dimension by two each test.

Dim	Optimum w
1024	1.7
2048	1.7
4096	1.7
8129	1.8
16,384	1.7
32,768	1.8
65,536	1.7
131,072	1.7

Clearly 1.7 is the optimum value of w for this matrix. The next goal was to compare how the accelerated version compared to the serial version. Relaxation was set to 1.7, dimension was varied, and in the GPU version, the threads per block was set to 1024, and the number of blocks was set to $dim/1024$. The amount of time to compute the inverse of the preconditioner is not included in this calculation.

Dim	Serial Runtime	Parallel Runtime
1024	0.260676	0.084292
2048	1.630144	0.169177
4096	7.637686	0.332503
8129	33.571885	0.601911
16,384	1:05.138690	1.332448

Thus we can clearly see that the accelerated version performs significantly faster than the serial version. The final thing to test was how varying the number of GPU threads used affected the run time. The number of threads per block was fixed at 1024, and the number of blocks per grid was varied.

Dim	1 block	2 blocks	4 blocks	8 blocks	16 blocks	32 blocks	64 blocks
1024	0.085	0.063	0.051	0.036	0.048	0.048	0.053
2048	0.247	0.168	0.125	0.078	0.101	0.099	0.111
4096	0.680	0.481	0.319	0.242	0.236	0.226	0.245
8192	2.072	1.213	0.773	0.616	0.547	0.513	0.515
16,384	7.814	4.444	2.379	1.464	1.165	1.161	1.108
32,768	31.027	16.471	8.682	4.847	3.692	3.682	3.381
65,536	2:11.559	1:08.546	36.201	9.674	12.834	12.471	10.312

This table demonstrates that if we have significantly less threads than the dimension of the problem, the performance takes a serious hit, however once the degree of parallelism reaches the dimension, not much extra performance is gained.

4 Code

The project is structured into 5 important directories. The code primarily exists as a header only library in the include directory. All CUDA kernels are in the Parallel.cuh file. All of the serial linear algebra code, and specialized templates for calling the kernels, are in the DenseVector.hpp, SparseMatrixBase.hpp, and SparseMatrix.hpp. DenseVector.hpp contains all relevent vector code, and SparseMatrixBase.hpp and SparseMatrix.hpp contains all relevant compressed sparse row matrix code. The class hierarchy for matrices consists of a base SparseMatrixBase class, and two derived classes, SparseMatrix, and ProxySparseMatrix. SparseMatrix is a straightforward template for a CSR matrix. ProxySparseMatrix contains a reference to another sparse matrix, along with an array of entries. The purpose of this class is to create a new matrix that shares the same sparsity pattern of another matrix, without replicating the row and column arrays.

The PCG algorithm is in the LinearSolve.hpp header file. This file contains the LinearSolver class. Due to the way the SparseMatrix and DenseVector classes are specialized, making LinearSolve run on the GPU vs the CPU is determined by a single template argument, as can be seen in Main.cu. The final files in the include directory are Error.cuh, gpu_memory.cuh, and allocater.hpp. Error.cuh contains a single macro called checkCuda, which will check the error status returned from any CUDA function, and throw an exception if the status does not indicate success. The gpu_memory.cuh header contains several convenience functions for safely allocating and deleting CUDA memory, creating unique pointers to CUDA memory which will free correctly, and functions for copying data into and out of the GPU. Finally, allocater.hpp contains partially specialized methods for allocating either CPU or GPU memory where appropriate.

The src directory contains the file Main.cu. This is the main executable, and is responsible for running the PCG algorithm with the specified arguments. The test directory contains all the unit tests for this project. Since this code

only exists for the sake of this project, I did not spend as much time writing unit tests as I normally would have. Nevertheless, these tests do demonstrate some level of correctness for various aspects of the algorithm.

The final two directories are the build and bin directories. The build directory contains all of the build objects created by cmake, as well as the downloaded code for googletest. The bin directory contains all tests, as well as the main executable, called pcg. The pcg command comes with a useful help menu that can be invoked by running

```
./pcg --help
```

A typical use of pcg is the following command:

```
./pcg -d $((1024*16)) -g -r 1.7 -p 1024 -b 16
```

This will run the PCG algorithm on a system with dimension 16,384 on the GPU, with relaxation parameter set to be 1.7, the number of threads per block set to 1024, and the number of blocks set to 16.

5 Conclusion

The results show that the PCG algorithm can benefit enormously from GPU acceleration. However, there are still some limitations in the implementation of PCG. The PCG algorithm relies on the set of p_k vectors being A conjugate to each other, i.e $p_i^t A p_j = 0$ if $i \neq j$. The Graham-Schmidt process, along with the fact that the residuals are orthogonal to each other, is used to efficiently compute subsequent search directions. However, the Graham-Schmidt process is numerically unstable, thus after many iterations, the subsequent search directions begin to loose their conjugacy, causing the vector x_k to cease converging to the solution. To mitigate this fact, the threshold ϵ was greatly increased for higher dimensional experiments. Future work could be done to reset the algorithm after a certain number of iterations, and start again, setting x_0 to be the last value obtained previously.

Furthermore, the algorithm further benefit from further splitting of the work. MPI could be used to distribute the matrix vector products and vector dot products across several nodes, each with its own GPU, to further accelerate the computation. Pursuing these goals would be very interesting in any future work. However, the implentation of PCG on a single GPU node clearly demonstrates that GPU acceleration is a viable method for increasing the efficiency of sparse matrix solvers.