Solving a System of Equations via QR Decomposition on NVIDIA GPUs provided an Armadillo Sparse Matrix

John Crow

December 2021

Abstract

Armadillo is a Linear Algebra and Scientific Computing library that provides high level syntax for otherwise complicated functions. While the library is accessible, when a user would like to accelerate their code via a GPU the process is non-trivial. This report and the corresponding code demonstrates a process to deconstruct the Armadillo data structure for sparse matrices, sp_mat, and pass these "standard" elements to a library designed for GPU utilization. The solution from the GPU is compared to the Armadillo generated solution and speed up is demonstrated.

1 Introduction

Over the past decade computational science has required both faster more robust code and code with high level syntax that makes development easy. Libraries like NumPy for Python, Armadillo for C++, and recently even entire languages like Julia have tried to balance speed with usability. But there are times that specific processes can be accelerated with a more basic but powerful library that allows for code "closer" to the hardware. cuSOLVER is a library that is part of the CUDA toolkit and provides functional support for the NVIDIA GPU.

2 Problem Description

$$Ax = b \tag{1}$$

Given a system of equations represented by (1) where A is a large highly sparse matrix, solve for x. In Armadillo sparse matrices are represented by a struct sp_mat which abstracts away the actual data structures that store the sparse matrix. The CUDA library cuSOLVER features a function to solve the

above system via QR factorization, cusolverSpDcsrlsvqr. cuSOLVER requires not a single data structure but the actual elements that define a sparse matrix, specifically one stored in Compressed Sparse Row(CSR) format. The Armadillo documentation states that it's sp_mat class utilizes Compressed Sparse Column(CSC) format not CSR and it does not detail how to access this data.

3 Problem solution

3.1 Deconstructing an Armadillo sp_mat

The documentation for the cuSOLVER function requires:

```
int row
int col
int number_of_nonzero_elements
double * values_of_nonzero_elements
int * Row_Pointers
int * Col_Indices
```

Which corresponds to a sparse matrix in CSR format. Armadillo stores its sparse matrices in CSC format but this can be solved by using another CUDA function that converts from CSC to CSR. The problem then becomes grabbing the data out of the sp_mat class and transforming it into the data types required by cuSOLVER. While the Armadillo documentation does not explain how to access this data the above attributes are easily found in the Armadillo header files. The issue is that the arrays are stored as const long long unsigned double/int pointers. In order to convert the arrays the user must cast each element in the array.

3.2 Converting from CSC to CSR

In order to pass the sparse matrix to cusolverSpDcsrlsvqr we must first convert it to CSR format. Normally if a computer scientist were to look at a sparse matrix and construct the CSR or CSC format by hand. The values would be identical and the column/row pointers and row/column indices would be where we see a difference. It is important to note that the cuSPARSE function cusparseCsr2cscEx2 does not operate this way. Instead it keeps the column and row pointers the same but rearranges the values as the transpose of the original. It should also be noted that in order to use cusparseCsr2cscEx2 for CSC to CSR the user should swap the row and column values when passing these values to the function. From here it is relatively simple to call cusolverSpDcsrlsvqr.

4 Results and Discussion

Below in Table 1 I show the results of solving

$$Ax = b \tag{2}$$

utilizing 3 methods. The first is performing QR decomposition and then solving

$$x = R^{-1} * (Q^T * b) \tag{3}$$

utilizing Armadillo. The second is utilizing Armadillos provided x=solve(A,b) utility. The last is the cuSOLVER solution. I also show the L1 norm of the results to demonstrate that we are achieving similar solutions for all 3 methods. The test matrices are sparse matrices of size M by M with an non-zero element density of 3 percent. Time is in seconds.

M	QR	ARMA	L1(QR,ARMA)	CUDA	L1(CUDA,ARMA)
256	0.054532	.007731	5.77049e-11	0.0	2.05643e-11
512	0.235752	0.040988	1.11746e-10	0.0	5.40185e-11
1024	0.777328	0.091157	3.39413e-10	0.1	4.92327e-10
2048	3.56627	0.655714	9.17619e-10	0.7	8.39079e-10
4096	46.3874	8.43671	3.09415e-08	5.9	3.50319e-08
10192	351.07	64.9403	1.09665e-07	38.3	1.12973e-07
20384	-	509.089	-	274.2	1.44178e-07

I also present a graphical view of the timing comparisons as problem size increases in Figure 1. In Figure 1 the timing of the total function call to CUDA is also plotted. This accounts for all of the memory copies and smoothing of the data. This project demonstrates not only how to specifically call a CUDA GPU function from Armadillo but more broadly the process one can take utilizing any library that stores its sparse matrices in a standard format. From both the timing results we can see a speed up factor of approximately 2 for sufficiently large problem sizes.

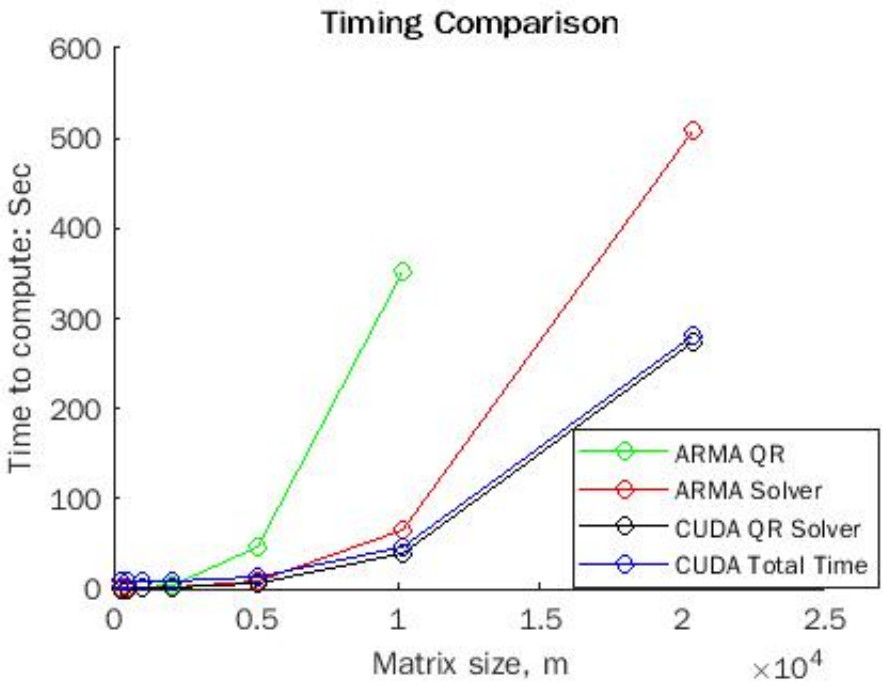

Figure 1: Time vs Matrix size M

5 References

1. \C++ Library for Linear Algebra & Scientific Computing." Armadillo, http://arma.sourceforge.net/.

2. \CUDA Toolkit Documentation v11.5.1." CUDA Toolkit Documentation, https://docs.nvidia.com/cuda/index.html.

3. "Include/armadillo_bits/spmat_meat.HPP · 10.7.x · Conradsnicta / Armadillo-Code." GitLab, https://gitlab.com/conradsnicta/armadillo-code/-/blob/10.7.x/include /armadillo_bits/SpMat_meat.hpp.