Final Project Report: QR Factorization on GPUs

Ryan Synk

December 20, 2021

Introduction

This report outlines my work on implementing a fast QR factorization algorithm on a Graphics Processing Unit (GPU). It closely follows a 2009 paper of Kerr et. al [1] and focuses on writing code to mimic the results found in the paper.

Graphics Processing Units

Graphics processing units (GPUs) are parallel processors initially designed for computing shaders in computer graphics applications. In a Central Processing Unit (CPU), one large, powerful chip performs the computations. GPUs, on the other hand, consist of many small processing units, each capable of performing computations in parallel with one another. These small processing units are referred to by NVIDIA, a GPU manufacturer, as Streaming Multiprocessors (SMs). Associated with each SM is a store of L1 cache memory. Each SM, in turn, can access a global L2 memory cache — see figure 1 for a diagram of a recent GPU. This architecture, with its combination of parallel processors and memory hierarchy, lends itself well to fine-grained parallelization — and in the case of computer graphics, dense linear algebra calculations.

Figure 1: NVIDIA Volta GPU. Source: [2]

Shortly after the advent of modern graphics processing units with programmable shaders, studies in the use of GPUs for scientific computing problems were conducted. In 2001, researchers implemented the first matrix-matrix multiplication algorithm for a GPU [3]. Eventually, certain applications running on GPUs began to outperform their counterparts on CPUs. In a 2005 paper (whose authors include UMD Professor Dinesh Manocha), a GPU implementation of LU factorization with partial pivoting was shown to outperform a CPU. [4]. Scientific computing wasn't the only field impacted by GPUs.

The training of deep neural networks is a process which involves large amounts of dense matrix-vector and matrix-matrix multiplications. The discovery that GPUs are well-suited to accelerate this traning process was instrumental in the revolution of deep learning in the 2010s, helping to turn neural networks from theoretical curiosities into the ubiquitous and powerful models they are today.

Work still continues as researchers in numerical linear algebra and high-performance computing try to find ways that GPUs can accelerate their algorithms. High-performance computing systems are being built with more and more GPU accelerators in them, and the DOE's newest exascale supercomputer, Frontier, will be built with four times as many GPUs as CPUs [5].

QR Factorization

In this report, we will develop a GPU-based method to solve the problem of factorizing an $m \times n$ matrix A as the product $A = QR$. Here, $Q \in \mathbb{R}^{m \times m}$ is an $m \times m$ unitary matrix, and R is an upper-triangular matrix. A number of basic algorithms for finding such a decomposition exist, and they all follow the pattern of successively zeroing out entries of A below the main diagonal using unitary transformations. These methods include

1. Givens rotation factorization
2. Gram-schmidt factorization
3. Householder QR factorization

For the latter, a series of Householder transformations are used to zero out entries below the diagonal of A. Given a vector $x \in \mathbb{R}^n$, a Householder matrix P_x can be generated such that $P_x x = \|x\| e_1$ — in other words, the matrix transforms the vector x into a multiple of the 1^{st} basis vector in $\mathbb{R}^n$.

The transformation P_x is of the form

$$P_x = I - \frac{2}{v^T v} v v^T$$
$$v = x - \|x\| e_1$$

If we choose x_k to be the the vector in $\mathbb{R}^{n-k}$ consisting of entries of the k^{th} column of A, starting at the diagonal and moving downwards ($A(m : k, n)$ in MATLAB notation) we can create a householder transformation P_k which zeros out the entries below the diagonal of the k^{th} column of A. See figure 2 for a diagram.

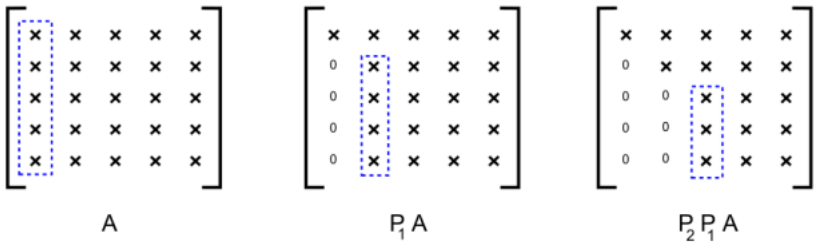

Figure 2: Successive householder transformations acting on A Source: [1]

Importantly, the P matrices don't need to be fully formed. If we define $\beta_k = \frac{2}{v_k^T v_k}$, then

$$\begin{aligned} PA &= (I - \beta_k v_k v_k^T)A \\ &= A - \beta_k v_k (A^T v_k) \end{aligned}$$

So we can perform a rank-one update to A at each iteration.

Putting this together gives the following QR algorithm:

Algorithm 1 QR Factorization Via Householder

Given $A \in \mathbb{R}^{mxn}$
$Q \leftarrow I_m$
$R \leftarrow A$
for $k = 1 : n$ **do**
  $[v, \beta] = \textbf{house}(R[k : m, 1 : n])$
  $R[k : m, 1 : n] = R[k : m, 1 : n] - \beta v v^T R[k : m, 1 : n]$
  $Q[1 : m, k : m] = Q[1 : m, k : m] - \beta Q[1 : m, k : m] v v^T$
end for

This QR algorithm works well, but for our purposes it will require some modification to run quickly on a GPU.

TODO:

1. Explain how to update Q in basic QR
2. Explain how chaining together P matrices gives QR

Algorithm

1. Explanation of why regular QR doesn't work as well (maybe cite a figure)
2. Explanation of block QR update (W, Y matrices)
3. Explanation of full block QR algorithm

Implementation

The algorithm was implemented in C and CUDA — NVIDIA's proprietary API for programming on NVIDIA GPUs. CUDA syntax closely follows that of C, and NVIDIA provides a compiler, called `nvcc` (NVIDIA C Compiler) to allow users to write GPU programs. CUDA divides programs up into "host" and "device" code — host code is run in serial on the CPU while device code is run (potentially in parallel) on the GPU itself. Functions written to be run on the device are usually referred to as "kernels." The CUDA API allows for a programmer to organize and allocate memory on the device from the host, and then launch compute-intensive kernels in parallel.

Since the release of CUDA in 2007, NVIDIA has written a number of libraries and APIs for a variety of GPU applications. One of the earliest they released was a library named CuBLAS [6]. The CuBLAS library adheres to the Basic Linear Algebra Subprograms (BLAS) API, and allows for users to call functions like `cublasDgemm`: CuBLAS double-precision general matrix-matrix multiply. Rather than writing all of the .... a programmer can use CuBLAS and call their kernels to multiply matrices and vectors together.

In the QR implementation, almost all of the kernels launched were calls to the CuBLAS API, with the exception of the computation of the Householder $[v, \beta]$ values, which was written seperately.

TODO:

1. Finish discussing CuBLAS
2. Discuss r value chosen
3. Discuss difference between my implementation and the papers

Results

TODO:

1. Plot of theoretical peak double precision performance on GPU versus actual implementation
2. Plot of Python CPU versus C++ GPU plot (time)

3. Plot of their results vs mine

Conclusion

TODO

1. Sum up what was done
2. Recap main points
3. End with further directions: parallelizing kernels, discussing if it would be useful for writing a least-squares solver

References

[1] A. Kerr, D. Campbell, and M. Richards, "Qr decomposition on gpus," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, (New York, NY, USA), p. 71–78, Association for Computing Machinery, 2009.

[2] L. Durant, O. Giroux, M. Harris, and N. Stam, "Inside volta: The world's most advanced data center gpu." `https://developer.nvidia.com/blog/inside-volta/`. Accessed: 2021-12-20.

[3] E. S. Larsen and D. McAllister, "Fast matrix multiplies using graphics hardware," in *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, (New York, NY, USA), p. 55, Association for Computing Machinery, 2001.

[4] N. Galoppo, N. Govindaraju, M. Henson, and D. Manocha, "Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware," in *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pp. 3–3, 2005.

[5] https://www.olcf.ornl.gov/frontier, 2019.

[6] NVIDIA Corporation, "Cublas."