Final Project Report: QR Factorization on GPUs

Ryan Synk

December 21, 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.

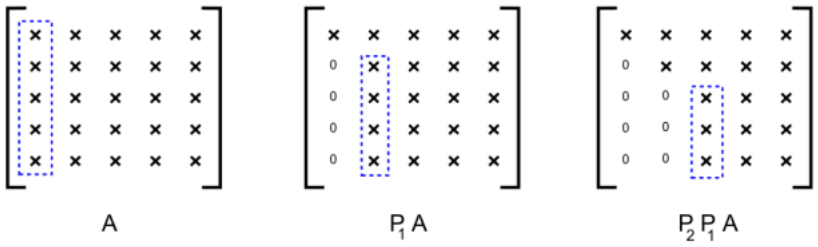

Figure 2: Successive householder transformations acting on A Source: [1]

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. We can string these updates together, and, since each of the P_k transformations are unitary, we preserve the original matrix:

$$A = (P_1^T P_2^T \dots P_k^T)(P_k \dots P_2 P_1)A$$

And our QR decomposition is

$$\begin{aligned} Q &= (P_1^T P_2^T \dots P_{n-1}^T) \\ R &= (P_{n-1} \dots P_2 P_1 A) \end{aligned}$$

So at each iteration, we make the following updates:

$$\begin{aligned} R &\leftarrow (I - \beta_k v_k v_k^T)R \\ Q &\leftarrow Q(I - \beta_K v_k v_k^T) \end{aligned}$$

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.

Algorithm

As was mentioned before, GPUs can offer speedups to many different problems. However, given the physical distance between host memory and the GPU, there is a nontrivial cost in moving data from the CPU to the GPU. In many problems, the data in question is small enough to fit onto CPU cache, and thus can be done in the time it takes to move the data onto the GPU itself. Care must be taken in writing GPU code to make sure that, when possible, all data that is moved onto the GPU stays on the GPU for the duration of the the computation. And, before writing any code, the size of the problem in question should be examined to ensure that a GPU will offer an increased effiency. In this way, GPUs will offer much greater speedups in computing QR factorizations for **large** matrices, and our algorithm is tailored around that idea.

Additionally, when data is on the GPU, in order to maximize efficiency, as many computations as possible should be performed in parallel. In this way,... As [1] states, in the above algorithm, "the amount of computation per memory element fetched from global memory is quite low." Currently, the algorithm... So if we want to take advantage of the speedups that a GPU has to offer, we will have to give the GPU a sufficient amount of work to do. In this way, we'd like to transition our computation from operating solely in terms of matrix-vector products, and shift it to focusing on matrix-matrix products.

It is with this in mind that we turn to a 1985 result of Bischof and Van Loan [6]. They showed that applying a series of rank-1 updates

$$\begin{aligned} P &= P_1 P_2 \dots P_r \\ &= (I - \beta_1 v_1 v_1^T) \dots (I - \beta_r v_r v_r^T) \end{aligned}$$

Is equivalent to applying a rank-r update of the following form:

$$\begin{aligned} P_{wy} &= P_1 P_2 \dots P_r \\ &= I + Y W^T \end{aligned}$$

For $m \times r$ matrices W and Y, which are constructed from the vectors and scalars (β_i, v_i). Their construction is outlined in the following algorithm:

So, given some set of r Householder updates, we can generate matrices W and Y to update the rest of our matrix with. This motivates Kerr et.

Algorithm 2 Construction of W, Y matrices

Given $v_1 \in \mathbb{R}^n, v_2 \in \mathbb{R}^{n-1} \dots v_r \in \mathbb{R}^{n-r+1}$
Given $\beta_1, \dots \beta_r \in \mathbb{R}$
$Y = v_1$
$W = -\beta_1 * v_1$
for $j = 2 : r$ **do**
    $z = -\beta_j v_j - \beta_j W Y^T v_j$
    $W = [Wz]$
    $Y = [Yv]$
end for

al's Block Householder algorithm. We first partition our matrix A into block matrices of size $m \times r$:

$$A = \begin{pmatrix} A_1 & A_2 & \dots & A_{\frac{n}{r}} \end{pmatrix}$$

For a given block A_k, we compute the necessary r Householder updates, and zero out the proper entries below the main diagonal in that block, giving us $v_1, \dots v_r$ and $\beta_1 \dots \beta_r$. Instead of updating the rest of the matrix using each individual v_i, we instead form the matrices W_k and Y_k, and update $A_{k+1}, \dots, A_{\frac{n}{r}}$ by multiplying each by $I + Y_k W_k^T$. This process is repeated over all blocks, until Q and R are formed. This gives us the following algorithm:

Algorithm 3 Block Householder QR

1: Given $A \in \mathbb{R}^{m \times n}$, $r \in \{1, \dots, n\}$ Partition $A = \begin{bmatrix} A_1 \dots A_{\frac{n}{r}} \end{bmatrix}$
2: **for** $k = 1$ to n/r **do** ▷ Loop over all blocks
3: $s = (k-1)r + 1$ ▷ index of first column of A_k
4: **for** $j = 1$ to r **do** ▷ Loop over each column in block
5: $u = s + j - 1$ ▷ index of j^{th} column of A_k
6: $[v_j, \beta_j] = \textbf{house}(A[u : m, u])$
7: $A_k \leftarrow (I - \beta_j v_j v_j^T) A_k$
8: $V[:, j] \leftarrow v_j$
9: $B[j] \leftarrow \beta_j$
10: **end for**
11: $W, Y = \textbf{constructWY}(V, B)$
12: $[A_{k+1} \dots A_{n/r}] \leftarrow (I + YW^T)[A_{k+1} \dots A_{n/r}]$
13: $Q \leftarrow Q(I + WY^T)$
14: **end for**

As a small aside, the matrix V is $m \times r$, while the vector v_j is in $\mathbb{R}^{m-u}$. So implicitly during the step $V[:, j] \leftarrow v_j$ on line 8, the vector v_j is first padded with zeros at the top.

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 [7]. 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 kernels for the block QR algorithm by hand, this implementation makes use of the pre-written CuBLAS calls to perform matrix-matrix and matrix-vector multiplications.

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.

Kerr's original paper set the r value parameter (size of each block) to be 64. They mention in the paper that "r should be chosen to minimize total runtime on the target architecture." In experiments, the code ran slightly faster with higher values, of 128 and 256. Experiments for varying r levels are shown below.

In the original implementation, Kerr et. al did not use the CuBLAS kernel for matrix-vector multiply. They cited the fact that their own implementation could achieve a speedup over the CUDA implementation. For the purposes of this report, an original `gemv` (generalized matrix-vector multiply) kernel was not given. This has not been tested, but there is a good chance that in the 12 years since the publication of Kerr's paper, NVIDIA has since

tuned their CuBLAS library to be very performant — CuBLAS had only existed for a couple years at the time of their publication.

All code for the project can be found in the following github repository: `https://github.com/ryansynk/final_project_amsc763`. Along with a CUDA/C implementation of the block QR algorithm, the following were also implemented along the way:

1. A python (serial/CPU) implementation of non-blocked Householder QR
2. A python (serial/CPU) implementation of blocked Householder QR
3. A C/CUDA (GPU) implementation of non-blocked Householder QR
4. A C (serial/CPU) implementation of non-blocked Householder QR

Results

TODO:

1. Description of hardware tests are ran on.
2. Plot of time for varying sizes of r.
3. Plot of theoretical peak double precision performance on GPU versus actual implementation
4. Plot of Python CPU versus C++ GPU plot (time)
5. 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. 7178, Association for Computing Machinery, 2009.

[2] L. Durant, O. Giroux, M. Harris, and N. Stam, "Inside volta: The worlds 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] C. Bischof and C. Van Loan, "The wy representation for products of householder matrices," *SIAM J. Sci. Stat. Comput.; (United States)*, vol. 8:1, 1 1987.

[7] NVIDIA Corporation, "Cublas."