Matrix Batch Processing using CUDA

Nguyen Duy Hoang, Harry (A0048242L)
Dandekar, Ashish (A0123873A)
Kuëndig, Adrian (A0134824H)
Ranade, Ketki (A0119950B)
Pham Thanh Tung, Terry (A0120113J)

National University of Singapore,
Singapore

April 16, 2015

Contents

1 Introduction

Earlier, GPUs were used primarily for computer graphics operations such as pixel rendering, image rotations and vertex translations. This involves a large number of very similar computations with no (or very few) inter-dependencies, making them highly parallelizable. Because most graphics computations involve standard matrix and vector operations, engineers and scientists have begun using GPUs for non-graphical calculations in domains like computational chemistry, physics simulations, pattern matching, graph analysis, etc. This has been further accelerated by the evolution of GPUs from being specially designed for computer graphics to todays general-purpose parallel processors with support for APIs in standard languages like C.

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model that was developed by Nvidia for use on its GPUs. CUDA exposes an instruction set for manipulating GPU memory and computational units. The interface is a superset of C and adds a set of constructs that developers can use to write parallelizable programs. Many libraries have been written for various applications, by making use of the primitives CUDA provides. CUBLAS is one such open source library providing an interface to parallelized linear algebraic computations on vectors and matrices.

Many modern applications such as multi-sensor networks require batch processing of large array of small-scale matrices. Leveraging on the massive multi-core architecture, our projects aim to explore the use of GPU to perform various matrix operations effectively and simultaneously. Unlike available libraries, the emphasis is not only on the efficient implementation of the primitives but also on batching the multiple matrices on GPUs. It embarks on first steps of numerous optimization problems like Gaussian Process Model where we conjecture that the massive parallelism of GPU will have cutting edge over CPU.

2 Related Work

Addition, multiplication and inversion are the primitive operations on the matrix, out of which inversion is far more non-trivial than other operations. Different papers have proposed solution to this problem. For example, Sharma[2] et al. have implemented a fast parallel Gauss Jordan algorithm for matrix inversion on GPU. Benner[1] et al.have shown that using the GPU only provides a speedup if the matrix exceeds a certain dimension.

The Gauss-Jordan algorithm discussed by Sharma et al. works with any invertible matrix. But the implementation is constrained by the limitations of CUDA (e.g., maximum of threads that can be launched and the size of the shared memory).

On the other hand, Benner et al. have focused on the SPD matrix inversion problem. The paper talks about two different ways to invert the SPD matrices namely, via Cholesky decomposition and using Gauss Jordan transformation. There is one remarkable result one gets from the paper that inverting the large matrices on GPU is profitable compared to inverting small matrices since the massive parallelism pays off the cost of data transfer.

Along with addition and multiplication, we aim to implement inversion of the matrices catered to the requirements enlisted above. Although Benner et al. have shown that inverting a small ma-

trix on the GPU is not economical, they have not explored the possible performance gains, if any, achievable by batching inversion of multiple matrices. We will study these works and implement matrix operations by utilizing primitives provided by CUDA.

3 Problem Description

We want to probe the embarrassingly parallel GPU architecture to make the Gaussian Process Optimization fast. A Gaussian process is completely determined by its mean and covariance functions. Given a set $\{A, \mathcal{B}, \mathcal{C}, D, E\}$ of which A, D, E are vectors and $\mathcal{B}, \mathcal{C}$ are symmetric positive semidefinite matrices. We want to compute;

$$\begin{aligned} \mu &= A(\mathcal{B}+\mathcal{C})^{-1}D \\ \sigma^2 &= E - A^T(\mathcal{B}+\mathcal{C})^{-1}A \end{aligned}$$

It is patent that this basic computation is rife with matrix operations like addition, multiplication and inversion. Plus in the Gaussian Optimization Problem, we have stream of vectors and matrices of small matrices. Most of the libraries deal with either big matrices or sparse matrices. Here, we want to work with numerous small and dense matrices. So, in this study we implement these primitives so as to make the scheme of batching many matrices in a single launch efficient. To put it objectively, we am to do following things:

- Investigating multi-threaded algorithms for matrix operations
- Developing a toolkit for batched matrix operations
- Conducting experiments to evaluate the efficiency and effectiveness
- Testing the toolkit on a Gaussian Process Optimization problem

4 Algorithms

Addition and multiplication functions have been implemented with very high efficiency in the libraries. It has been observed that implementing such operations again adds an overhead of data transfer. Hence, we choose to use factory functions for addition and multiplication/ Addition and multiplication functions have been implemented with very high efficiency in the libraries. It has been observed that implementing such operations again adds an overhead of data transfer. Hence, we choose to use factory functions for addition and multiplication and focus on non-trivial operation of matrix inversion.

4.1 Gauss-Jordan Method of Inversion

Gauss-Jordan method is the traditional method to invert any kind of matrix. The method makes use of elementary row operations namely multiplication by scalar, linear combination of rows and row shuffling to invert the matrix. In this method, the original matrix(say $\mathcal{A}$) is augmented with identity matrix($\mathcal{I}$) of the same dimension. Row operations are performed so as to convert $\mathcal{A}$ to $\mathcal{I}$. The same set of operations are simultaneously applied to $\mathcal{I}$ which reduced it to $\mathcal{A}^{-1}$.

Gauss-Jordan method is an algorithmic way of finding inverse of the matrix since there is deterministic sequence of operations which yields the inverse of matrix. Let A_i denotes the i^{th} row of the matrix $\mathcal{A}$ and a_{ij}. Following is the set of row operations performed on each row A_i sequentially.

Pivoting For a given row $\mathcal{A}_i$, the element on the principal diagonal of the matrix serves as the pivot element. If a_{ii} is not zero then the control proceeds to the next step. If the $\dashv_{ii}$ is zero then we need to shuffle the rows so as to make a_{ii} non-zero. To do this, we search for the row $j(> i)$ such that a_{ji} is not zero. If such a j is found then $\mathcal{A}_i$ is swapped with $|$. Failure at finding such a j value implies that the matrix $\mathcal{A}$ is not invertible.

Normalization Once a non-zero pivot is found, we divide A_i by a_{ii} to make the diagonal entry 1.

Matrix Transformation After the row A_i has been normalized, we want to make elements a_{ji} for all j but for i. This is achieved by transforming j^{th} row as $A_j \leftarrow A_j - a_{ji}A_i$.

Same operations are applied to $\mathcal{I}$ with the same scalars. When above procedure is applied to $\mathcal{A}$ for every row, the transformed $\mathcal{I}$ reflects the $\mathcal{A}^{-1}$.

Let's look at some caveats in the above procedure which one can exploit to parallelize the method. First of all, as stated earlier, these set of operations have to be performed sequentially on each row. This is the bottleneck in the parallelism which can not be alleviated. All of these operations when performed for a single row entirely changes the matrix and the scalars for the operations on the next row depend on the earlier transition. So, this part of the procedure has to be kept sequential. Despite this, we can indeed parallelize each operation. Let's see each operation one by one:

1. If the element on the principal diagonal is already non-zero then there is no need to find pivot. Search for a non-zero element as pivot has to be sequential again so as to avoid race condition to return result. Once such an element is found, corresponding rows can be swapped parallely.

2. Scalar-vector multiplication can, obviously, be done in parallel since we scale every element of the row by the same scalar.

3. This is a non-trivial parallelization step. In this step, each row must be transformed. So a thread is spawned for each row to handle transformation of the corresponding row.

4. One subtle point which may leave unnoticed in the above analysis is: all of these operations are row-oriented. So, we do not need to spawn more than n(number of rows) threads in a single pass.

4.2 Matrix Inversion using Cholesky Decomposition

5 Experimental Evaluation

In this section we study the performance of the different inversion algorithms. For the experiments we compared the different inversion algorithms with a baseline algorithm implemented in C.

5.1 System Specification

All experiments were conducted on a CentOS 6.6 server machine with XXX hyperthreads XXX GHz each XXX kB L3/L2/L1 cache, XXX GB of memory and a NVIDIA XXX graphics card. The GPU contained XXX blocks, XXX warps, XXX threads per block, XXX kB of shared memory and 6 GB of global memory. The measured memory bandwidth between main memory and GPU memory was a bit more than 3 GB/s. All calculations were executed with single precision floating point math.

5.2 Data Generation

We used a synthetic dataset throughout our experiments. The dataset was generated using MATLAB and consists of multiple sets of square matrices. Each set contains exactly 100 matrices. To benchmark higher numbers of matrices we duplicated the matrices in the set until we got the required number of matrices. The sets of square matrices had dimension of the power of two starting from 8×8 until 256×256.

5.3 Baseline Algorithms

The baseline algorithm is implemented in C using the LAPACK library and it consists of only two calls to the library functions *spotrf* and *spotri*. We chose this algorithm as the sequential (in the sense of thread parallelism) baseline algorithm because it is straightforward to implement and efficient. We also conducted experiments with rudimentary parallelization using OpenMP to see the performance achievable on today's multicore processors. In this parallel algorithm we inverted multiple matrices in parallel using the OpenMP *parallel for* statement.

In addition to the two algorithms implemented in this study we also implemented an inversion algorithm using the two cuda library functions *getrfBatched* and *getriBatched*.

5.4 Benchmarking

Our benchmarks consisted of 10 replicated calculations on the generated dataset. We increased the number of matrices starting from 100 in steps of 100 until 1000. We included the memory transfer time to and from the device into the runtime of each algorithm. Naturally the baseline algorithm did not contain any memory transfer. Still, we argue that memory transfer plays a major role in the decision to offload computation to the GPU since it is a non-negligible time overhead.

In addition to the total runtime of each algorithm, we also report the total working memory required to calculate the inverse for each method. Finally, we also evaluated the numerical stability of the different algorithms. We did this by computing inverses using MATLAB and storing them along the initial matrices. The error we report is the sum of the absolute pairwise distances between the reference and the computed inverse matrix. We also thought about calculating the measuring the distance of the identity matrix and the result of multiplying the inverse matrix with the initial matrix. We discarded this idea because it would also include the error of the final matrix-matrix multiplication.

As a second evaluation method we implemented a mean and variance calculation as used in Gaussian Process Optimization (GPO). We again implemented the baseline algorithm in C as a sequential and as a parallel version using OpenMP. We expected the performance of the GPU to be better in this experiment than in the raw inversion experiment, since there are more operations involved in the mean and variance computation. As with the inversion benchmarks we report the runtime of each algorithm, the used working memory, and the absolute error to the solution obtained from MATLAB.

6 Conclusion and Future Work

References

[1] P. Benner, E. S. Quintana-Orti, P. Ezzatti, and A. Remón. High performance matrix inversion of spd matrices on graphics processors. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 640–646. IEEE, 2011.

[2] G. Sharma, A. Agarwala, and B. Bhattacharya. A fast parallel gauss jordan algorithm for matrix inversion using cuda. *Computers & Structures*, 128:31–37, 2013.