Large Matrix inversion on GPU's

Yuval Mandel[1]

[1] Department of Computer Science and
the Department of Computer and Electrical Engineering
– Technion, Haifa, Israel

Abstract. As seen when looking at the Gauss–Jordan implementation in the paper "A fast parallel Gauss Jordan algorithm for matrix inversion using CUDA", In theory, each elementary row transformation can run in parallel within itself, processing n cells in the matrix. More than that, the algorithm can run each row needed elementary row transformation in parallel to each other, so in theory the algorithm can process n^2 in parallel. We will call the entire process as a matrix update. Each matrix update can potentially be $O(1)$ and we need updates as the amount of rows, therefore the algorithm can be $O(n)$. This is all good in theory. In practice, the hardware resources needed to perform this operation are very demanding to say the least. In order to calculate a 2048 on 2048 matrix, $2^{22} = 4,194,304$ threads are needed to run simultaneously, and if the used data types is float32, $2^{22} * 4 = 16,777,216$ bytes of data are needed. Which might present a bottleneck for the possible calculations. The CUDA implementation of thealgorithem and other related code is provided at https://github.com/YuvalMandel1/GPUMatrixInversion

Keywords: Matrix Inversion · GPU.

1 Introduction

Matrix inversion can be used in multiple machine-learning applications for finding the optimal weights of a solution, like in linear-regression such as LLS and more. As of right now, matrix inversion algorithms, like Gauss–Jordan, have a complexity of $O(n^3)$ in a serial way. The project goal is to take of advantage of the Gauss–Jordan inherit parallelism to improve the complexity to O(n), and to run it on multiple GPU's simultaneously.

The rest of the paper is organized as follows: **??** describes our proposed method, **??** provides our experimental results, and Section 7 concludes the paper.

2 Related work

The main paper which will we be referenced is "A fast parallel Gauss Jordan algorithm for matrix inversion using CUDA" by Girish Sharma, Abhishek Agarwala and Baidurya Bhattacharya. However, with the improvment of GPUs we show that even for larger matrices the complexity of the parallel algorithem can still be linear.

3 Theory

3.1 Making the algorithm parallel

The traditional Gauss-Jordan method for computing the inverse of a matrix A with dimension n starts by augmenting the matrix with an identity matrix of size n:

$$[C] = [A|I]$$

Subsequently, by performing elementary row transformations on matrix C, the left half of C is systematically transformed column by column into the unit matrix. This process can be divided into two steps for each column of the left half matrix. The first step involves converting the element a_{ii} to 1 using the transformation:

$$R_i \leftarrow R_i / a_{ii}$$

The second step is to eliminate all the other elements in the jth column, except for the jth row, using the following transformation:

$$R_i \leftarrow R_i - R_j \cdot a_{ij}$$

$$[C'] = \left[\begin{array}{ccccc|ccccc} 1 & a_{12}/a_{11} & a_{13}/a_{11} & \cdots & \cdots & 1/a_{11} & 0 & 0 & \cdots & 0 \\ 0 & a_{22} - a_{21} \times a_{12}/a_{11} & a_{23} - a_{21} \times a_{13}/a_{11} & \cdots & \cdots & -a_{21}/a_{11} & 1 & 0 & \cdots & 0 \\ 0 & a_{32} - a_{31} \times a_{12}/a_{11} & a_{33} - a_{31} \times a_{13}/a_{11} & \cdots & \cdots & -a_{31}/a_{11} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2} - a_{n1} \times a_{12}/a_{11} & a_{n3} - a_{n1} \times a_{13}/a_{11} & \cdots & \ddots & -a_{n1}/a_{11} & 0 & 0 & \cdots & 1 \end{array}\right]$$

Fig. 1. The matrix C after the first update.

Now we shall redesign the algorithm so it shall run in parallel. First, we will update all the elements in a single row in parallel to each other, since they are independent from one another, As seen in Figure 2.

And we can notice that the rows are also independent from one another, so we will update all the rows (the entire matrix) in parallel, as seen in Figure 3.

3.2 The challenges of synchronization

There are 2 approaches in order to synchronize the n^2 threads: The first is a single barrier for all of them, so each thread runs a single update on its assigned element, and wait for all the threads to finish. The second is having each element

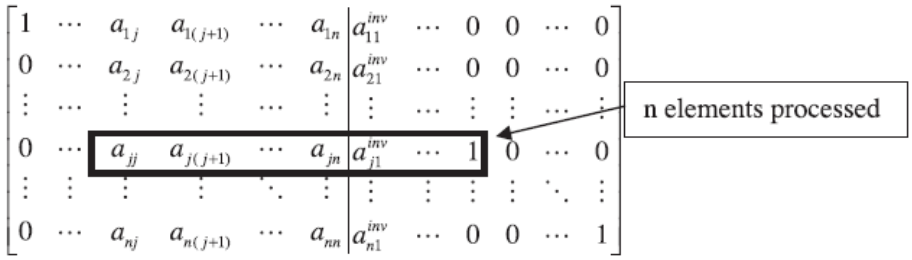

Fig. 2. The elements of a row, processed in parallel.

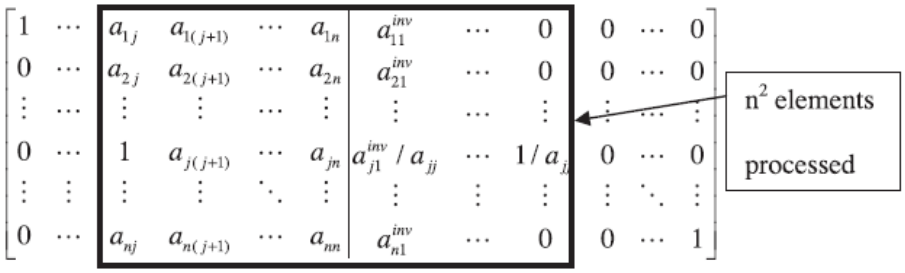

Fig. 3. The elements of the matrix in a single update, processed in parallel.

have a history buffer the size of n with an atomic variable to match. This way, all the threads wait only for the relevant elements to be updated.

The first approach requires minimal memory resources, so the space complexity is $O(1)$, but the second approach has a space complexity of $O(n^3)$ (each one of the n^2 elements requires a buffer of size n, making it irrelevant for large matrix inversion on a single GPU.

4 Implementation

4.1 Synchronization

As discussed in Section 3.2, we shall use the first method with a single barrier for all threads. Since the GPU we are using has no support of barriers between thread blocks, we will invoke a GPU kernel for a single update, let the threads of the kernel update the matrix and only then continue.

4.2 Memory management

One of the issues implementing the algorithm on the GPU was a data-race between different threads. It could occur when a thread has updated it's assigned element, therefore overriding the previous value, which other threads need for their computation. This issue happened for matrices larger than 205^2, most likely because the GPU can run 205^2 in parallel, not with each other.

This issue was solved using two arrays, a double buffer, so one is used as the source matrix and one as the result, and each iteration/kernel invocation, we alternate between them. You can view the invocation of the kernel in listing 1.1.

Listing 1.1. Kernel invocation, allowing synchronization and memory swapping.

```
for (int i = 0; i<n; i++){
 if(i % 2 == 0){
  matrix_inversion_kernel<<<numBlocks, threadsPerBlock>>>(
      d_A1, dI1, d_A2, dI2, n, i);
 }
 else{
  matrix_inversion_kernel<<<numBlocks, threadsPerBlock>>>(
      d_A2, dI2, d_A1, dI1, n, i);
 }
}
```

4.3 Kernel

The kernel itself is quite simple, as based on the thread 2D indexes, it updates the both matrices A and C. some pieces of code are not necessary, but are kept for debug. You can view the implementation in listing 1.2.

5 Results

5.1 Limits

The GPU implementation does succeed with matrices of 8191^2 in size, but cannot manage 8192^2. It seems to be a part of the hardware limitations.

5.2 Linearity

We will show that the compute time for our algorithm is linear, $O(n)$, by looking at matrices of size $n = 2$ to $n = 205$ as in in figure 4.

5.3 Example of invocation (Nsight)

We can view an example of an algorithm run on a matrix of $n = 2048$ using Nsight, as seen in figure 5. The part we are intersted in is the grey part under "CUDA API". It holds the 2048 kernel invocations of the algorithm. We can view that time for the general setup and synchronization outside the algorithm can have a significant run-time, making the implantation overall run-time increase significantly.

Listing 1.2. The Kernel implementation

```
__global__ void matrix_inversion_kernel(double *input_A,
    double *input_I, double *output_A, double *output_I, int
    n, int i){
 int x = blockIdx.x * blockDim.x + threadIdx.x;
 int y = blockIdx.y * blockDim.y + threadIdx.y;
 int block_num = blockDim.x*blockDim.y;
 double temp_A;
 double temp_I;

 if (x < n && y < n){
  if(x==i){
   if(x==y){
    temp_A = 1;
    temp_I = 1/input_A[i*n + i];
   }else{
    temp_A = input_A[i*n + y]/input_A[i*n + i];
    temp_I = input_I[i*n + y]/input_A[i*n + i];
    }
  }else{
   temp_A = input_A[x*n + y] - input_A[i*n + y]*input_A[x*n +
        i]/input_A[i*n + i];
   temp_I = input_I[x*n + y] - input_I[i*n + y]*input_A[x*n +
        i]/input_A[i*n + i];
  }
 }
 __syncthreads();
 if (x < n && y < n){
  output_A[x*n + y] = temp_A;
  output_I[x*n + y] = temp_I;
 }
}
```

6 Comparisons

6.1 Classic algorithm on CPU

For a matrice in size of $n = 205$, we have witnessed that a classical Gauss–Jordan algorithem has a run-tine of about $56ms$, and our implemantion requires $4.5ms$

6.2 Pytorch with GPU and CPU

We can see in figure 6 that our performance for matrices of size $n = 2$ to $n = 205$ are worse than that of Pytorch. In the project there was an unsuccessful attempt to access the relevant code in the Pytorch repository, making it impossible to understand how Pytorch performs better in this task.

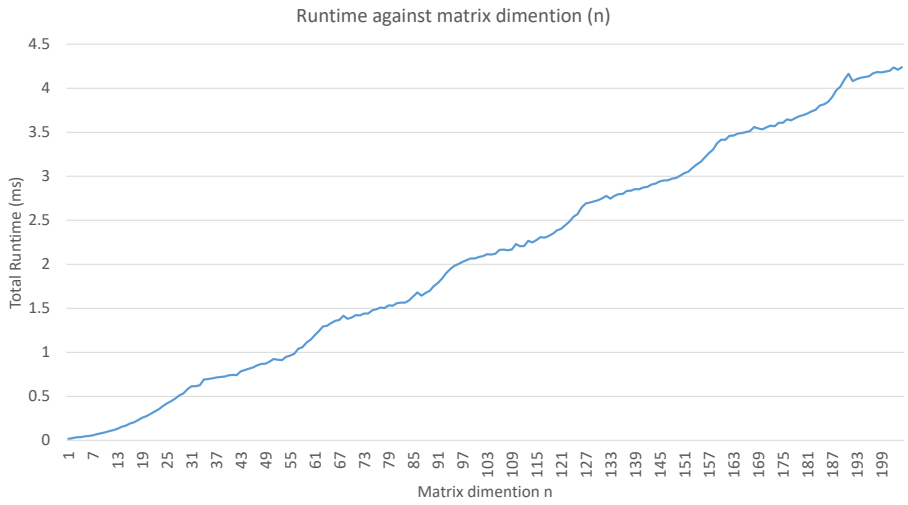

Fig. 4. Our implementation run-time against the size of the matrix.

7 Conclusions

This project tried to tackle the issue of matrix inversion long run-time, which makes the use of matrix inversion as a tool for machine learning algorithms very difficult.

We have managed to show an implementation that has the time complexity of parallel $O(n)$ instead of the classic $O(n^3)$.

However, this implementation does not improve the results that Pytorch achieves on bot GPU and CPU. We attempted to access the code for the matrix inversion of Pytorch, but without any success.

The most challenging aspect of the project was to make the CUDA code work, and dealing with the issue of synchronization.

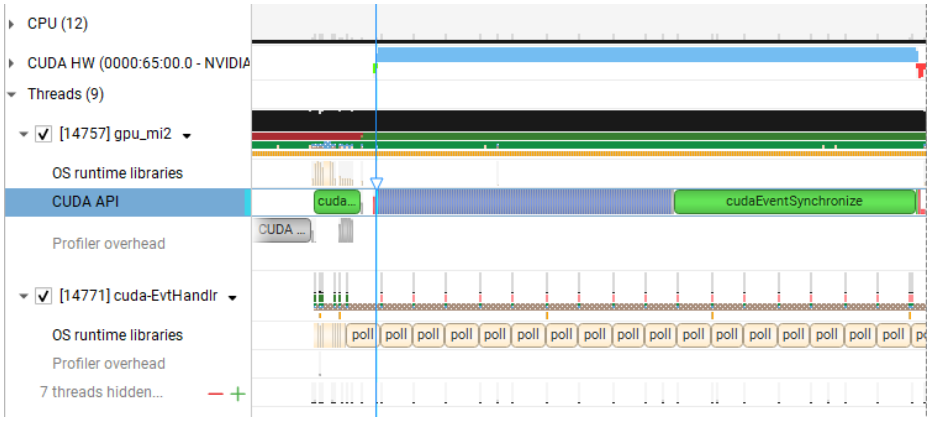

Fig. 5. Our implementation run-time against the size of the matrix.

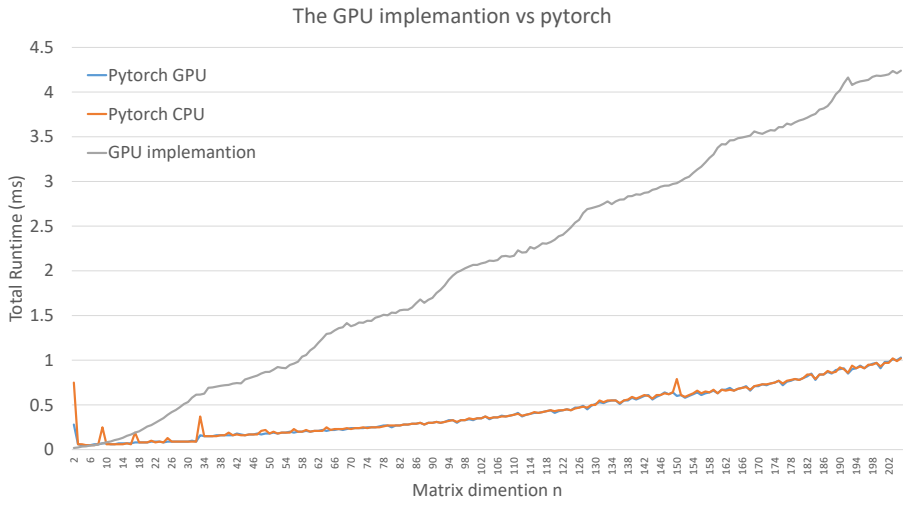

Fig. 6. Comparison between Pytorch on CPU and GPU vs our implementation.