HIGH PARALLEL COMPUTING

ECOLE CENTRALE LILLE

MASTER DATA SCIENCE

Lab3: The power method on GPU

Authors:
Ayoub Youssoufi
Yassin El Hajj Chehade

Professor:
Pierre Fortin

March 25, 2023

1 Introduction

The power iteration is an eigenvalue algorithm: given a diagonalizable matrix A, the algorithm will produce a number λ , which is the greatest (in absolute value) eigenvalue of A, and a non-zero vector v, which is a corresponding eigenvector of λ , that is Av= λ v. The issue is that for large matrices the computation time becomes too long. That is the reason why the GPUs has been introduced to this problems. In the following lab, we will be implementing the power method algorithm using the GPU.

2 Work description

We want to create a sequential algorithm using GPU to compute the eigenvalues of matrix of size n x n. The error estimated from the computation should be less than 10^-5 . In order to achieve this result, we use 5 kernels :

- Kernel to compute the product A.x
- Kernel to compute the norm $\|A.x\|^2$
- Kernel to compute y = $\frac{A.x}{\|A.x\|}$
- Kernel to compute the squared error = $\|x - y\|^2$
- Kernel to update X such that $X = Y$

Algorithm 1 Sequential algorithm

while $error > threshold = 10^-5$ **do**
  y $\leftarrow \frac{A.x}{\|A.x\|}$
  error $\leftarrow \|x - y\|$ and x = y
end if

Algorithm 2 Algorithm with cuda GPU

while error $> ERROR\_THRESHOLD$ **do**
  error $\leftarrow$ 0;
  norm $\leftarrow$ 0;
  cudaMemcpy(d_norm, &norm, float, HostToDevice);
  cudaMemcpy(d_error, &error, float, HostToDevice);
  matmulkernel<<<gridSize, threadsPerBlock>>>(d_A, d_X, d_Y, n);
  cudaDeviceSynchronize();
  normkernel<<<gridSize, threadsPerBlock>>>(d_norm, d_Y, n);
  cudaDeviceSynchronize();
  divkernel<<<gridSize, threadsPerBlock>>>(d_Y, d_norm, n);
  cudaDeviceSynchronize();
  difkernel<<<gridSize, threadsPerBlock>>>(d_Y, d_X, d_error, n);
  cudaDeviceSynchronize();
  switchx<<<gridSize, threadsPerBlock>>>(d_Y, d_X, n);
  cudaDeviceSynchronize();
  cudaMemcpy(&error, d_error, float, DeviceToHost);
  error $\leftarrow \sqrt{error}$;
end while

2.1 Kernel explanation

As we just mentioned, we used 5 kernels to adapt the algorithm to the Cuda GPU. By using these kernels, we transform the algorithm from the sequential version to a parallelized algorithm. Instead of using loops, we used the GPU threads, and thus the algorithm was much faster. For the first kernel, we used the x-dimension of the variables **blockDim.x**, **blockIdx.x** and **threadIdx.x** to be able to parallelize the algorithm and so for the other kernels. For the calculation of the norm and the calculation of the error, we used the atomic function **atomicAdd** to be able to avoid the problems of memory call.

Algorithm 3 Matrix multiplication kernel 1

```
i ← blockDim.x*blockIdx.x + threadIdx.x;
if i < n then
  d_Y[i] ← 0;
  for int K ← 0; K < N; K ← K + 1 do
    d_Y[i] ← d_Y[i] + d_A[i*n + k]*d_X[k]
  end for
end if
```

In algorithm 3, which presents the kernel of matrix multiplication, the idea is to use the variables of the cuda GPU, threadIdx for example, which indicates the rank of the thread that will parallelize the access to the matrix and then accelerate the execution of the code. For the other kernels we used the same idea to parallelize the code.

2.2 Parallel algorithm

After defining the kernel functions we are able to do the work and calculate the desired matrix.

2.2.1 Variables

In order to facilitate the computation, we used a grid of dimension 1 gridSize which is the quotient of n the length of X over block_size the number of threads per block.

In addition, to adapt the memory transfer between the GPU and the CPU, we used the following variables:

- The matrices A, X and Y in the CPU.
- The matrices d_A, d_X and d_Y in the GPU.
- The matrices d_norm and d_error of norm 1 to compute the norm and error in the GPU.

First, we initialize the matrices A and X. Then we start the parallel calculation. We use the while loop until we reach or exceed the error threshold.

We start by setting the norm and the error to 0 using the memcpy function, then we use consecutively our 5 already defined kernels. Between each kernel we add a function **cudaDeviceSynchronise** to assert that all threads have finished their work before moving on to the next kernel. At the end, we use a GPU CPU synchronous transfer to update the error and check the condition.

3 Optimization

Regarding the code implemented in the algorithm 2, the essential optimization was made on reducing the transfer memory between the CPU and the GPU. Instead of transferring the data, after the 4^{th} kernel where we compute the error, using 2 memcpy, one to copy d_Y in X and the other one to copy X in d_X, we preferred to use a fifth kernel and do this operation in the GPU.

Furthermore, we could apply the same logic with the norm and error computation since we need that all threads finished their work before the calculation. But we chose the atomic function, atomicAdd which speeds up the computation and organizes the access to the variable between the different threads.

In addition, one idea was to use a thread to parallelize the multimatrix kernel instead of using a for loop, but we would find ourselves in a situation with a memory access problem for $Y[i]$.

4 Performance

We start by running the code using $n = 32768$ and with $block\_size = 256$. In terms of time, the GPU version took 60 seconds to complete, and the sequential version took 527 seconds. Table 1 shows the time taken to run the Algorithm 2 with $n = 32768$. As we can see the best result was obtained with 512 threads per block. We made some changes to the grid size, but we still roughly reach the same time execution.

5 Testing the algorithm

To run the code in Grid5000:

Length of matrix X	BlockSize	Time(in seconds)
32768	128	68
	256	60
	512	55
	1024	54

Table 1: Time required in algorithm 2 with different block_size

5.1 Algorithm1

See the following commands :

```
$ gcc sequential.c -o test -lm
$ ./test [n]
```

Here we test the first algorithm, the sequential version. The variable between 2 brackets is necessary in order to execute the code, **n** here is the length of the X matrix.

5.2 Algorithm2

Before we have to go to a GPU environment using the following command:

```
$ oarsub -l /cpu=1/gpu=1/core=2/,walltime=2:0:0
-I -p "gpu_model = 'GeForce GTX 1080 Ti'"
```

See the following commands to execute the code:

```
$ nvcc --generate-code arch=compute_61,code=sm_61 -O3 -o gsql seq_gpu.cu
$ ./gsql [n] [block_size]
```

Here we test the second algorithm with block_size threads. The variable between 2 brackets are necessary for the execution of the function. The variable **n** tends for the length of the matrix X and the variable **block_size** for the number of threads per block.