GPU COMPUTING

Mattia Ferraretto [00072A]

Academic year 2022-2023

Contents

1 Project requirements

The project consists of developing the PCA (Principal Component Analysis) technique using CUDA runtime API in C. The purpose of the project is to show the speedup and the correctness obtained with parallel solution compared to the sequential solution.

2 Introduction to PCA

Principal Component Analysis, also called PCA, is one of the most popular data-mining technique used for dimensionality reduction. Basically, it consists of taking a dataset of tuples representing data points in high dimensional space and replacing them with their projection onto the most important axes. More formally, original data, represented by the matrix M, are transformed by the eigenvectors of the matrix $M^T M$, where the first-largest eigenvector corresponds to the axis on along which the variance of the data is maximized, the second-largest eigenvector is the axis along which the variance of distances from the first axis is greatest and so on. In the end, original data are approximated by data with many fewer dimensions and that summarize original data well. [1]

Follow PCA pseudo-code:

Algorithm 1 Principal Component Analysis

Require: Dataset as matrix M

$M^T :=$ transpose(M)
$COV :=$ matMul(M^T, M)
$E :=$ eigenvectors(COV)
$M' :=$ matMul(M, E)

As can be seen, first the transposed matrix of original data is computed, then the covariance matrix is computed as the matrix multiplication between the transposed and original matrix, subsequently eigenvectors of covariance matrix are found, and finally data are transformed with a matrix multiplication between original data and eigenvectors.

2.1 Finding eigenpairs

In literature there are several methods to compute eigenvectors and eigenvalues of a symmetric matrix. Due to the simplicity, I decide to provide a from scratch implementation of **power iteration method** in my project.

Let's now start with the definition of eigenvalues and eigenvectors: let M be a square matrix, λ a constant and e a nonzero column vector with the same number of rows as M. Then λ is an *eigenvalue* of M and e is the corresponding *eigenvector* if $Me = \lambda e$.

In power iteration method we start by computing the principal eigenvector multiplying the matrix M with any nonzero column vector, we then modify the matrix removing the fist-largest eigenvector in order to obtain a new matrix whose its principal eigenvector is the second-largest one

of the original matrix. The process proceeds until the desired number of eigenvectors is computed. Formally speaking, let M be a square matrix and x_k any nonzero vector, then:

$$x_{k+1} = \frac{Mx_k}{\|Mx_k\|}$$

where $\|Mx_k\|$ is the Frobenius norm of the vector. That is, we multiply the vector x_k by the matrix M until it converges. Let x be the vector obtained after the iteration, x is the first-largest eigenvector of M. To obtain the corresponding eigenvalue we compute $\lambda_1 = x^T Mx$. Now, to compute the second eigenpairs, we create a new matrix $M^* = M - \lambda_1 xx^T$ and then we use the power iteration on M^* to compute its largest eigenvector, which corresponds to the second-largest one of the original matrix M. As just mentioned, termination is reached when the desired number of eigenvectors are computed. [2]

3 Project structure

The project is dived into two main parts:

1. `clib` folder containing the libraries `ndarry` and `linalg`
2. `cudalib` folder containing the library `cudalinalg`

In addition there are three files `main.cu`, `test.cu`, `profiling.cu` used to define PCA algorithm, unit tests, profiling tests on CUDA-accelerated functions respectively.

3.1 `clib`

The `clib` folder contains libraries developed in ANSI C. Inside it, you find `ndarray` and `linalg` libraries. Each library consists of two files: the header (`.h` file extension) and the implementation (`.cpp` file extension).

ndarray library The purpose of the `ndarray` library is to provide structures and functions to easily manage an n-dimensional array. Each one is represented by a C struct containing the shape and data. Then, several basic functions are defined to properly model n-dimensional arrays. The most important functions are: `new_ndarray` used to allocate a new n-dimensional array and `free_` used to free the memory. As fundamental element of the project this library is used in both the developed solution, sequential and parallel.

linalg library The purpose of `linalg` library is to provide building blocks of dense linear algebra necessary for the implementation of the PCA algorithm. That is, in the library are present vector-scalar, matrix-scalar, matrix-vector and matrix-matrix operations. All of them are implemented sequentially.

3.2 `cudalib`

The `cudalib` folder consists of CUDA-accelerated libraries. In particular, it contains the `cudalinalg` library. As well as in C libraries, it is composed by two files: header (`.cuh` file extension) and the implementation (`.cu` file extension).

cudalinalg library The purpose of the library, again, is to provide the fundamental building blocks of dense linear algebra essential for the implementation of the PCA algorithm. So, linear algebra operations implemented are the same developed in the `linalg` library, the main and most important difference is that they are accelerated with CUDA runtime API.

In more details, such library includes primitive functions useful for transferring memory from host to device and device to host, kernel definitions and dedicated host functions needed to handle memory transfer and kernel declarations. All of the linear algebra operations implemented support the memory transfer overlap by means of CUDA streams, except for the matrix multiplication, that due to its complexity in handling efficient memory transfer, a streamed version of it has not been developed. The operations supported by CUDA stream have the following memory transfer scheme:

Algorithm 2 Memory transfer scheme

Require: nTile (number of memory tiles)

Compute number of elements/rows per memory tile
Compute number of stream needed

for each stream **do**

Create the stream
Transfer memory tile to the device
Launch the kernel
Transfer memory tile to the host

end for

Device synchronize

for each stream **do**

Destroy the stream

end for

In order to improve the performance in reading and writing operations, the library uses the pinned memory. It is also important to remark that all the resources allocated in the device are then deallocated after the computation of the device is termined.

3.3 `main.cu`

The `main.cu` file implements PCA algorithm. As already said in the Section 1, in order to reach the achievement, sequential and parallel solution are shown. To ensure the correctness of both solution, a simply test was written. The test consist of measuring the average error between the two results found by the proposed solutions, when the error tends to zero the solutions are very similar.

3.4 `test.cu`

The `test.cu` file contains unit tests. Basically, each unit test consist of checking the correctness of the sequential solution compared to the parallel one for each function implemented in `linalg` and `cudalinalg` libraries. A unit test is passed when the solutions are not too much different, with a tolerance error lower than 1e-2.

3.5 `profiling.cu`

The `profiling.cu` file is simply used to create an executable file. This file launches all the kernels implemented and is then profiled by the Nsight Compute tool developed by Nvidia. For more details about kernels profiling, see Section 5.

4 Performance evaluation

In this section I'll show you the performance obtained by testing each single functions and PCA algorithm, but first let me introduce the test environment. Results were obtained by running the test file on Google Colab free tier. The free tier offers a virtual machine with an Intel Xeon CPU with two vCPUs and 13 GB of RAM. In addition, when changing the default runtime, Colab allows the use of a Tesla T4 GPU with 16 GB of VRAM. In the following table results are shown:

Functions	Size (MB)	CPU time (s)	GPU time (s)	Speed up	Passed
Vector-scalar div.	256	0.3185	0.0527	6x	YES
	512	0.6318	0.1013	6x	YES
	1024	1.4574	0.2040	7x	YES
	2048	2.4497	0.4048	6x	YES
Euclidean dist.	256	0.4667	0.0613	8x	YES
	512	0.7980	0.1177	7x	YES
	1024	1.5728	0.2360	7x	YES
	2048	3.1742	0.4105	8x	YES
Matrix-scalar prod.	256	0.3135	0.0551	6x	YES
	512	0.6006	0.1089	5x	YES
	1024	1.6568	0.2070	8x	YES
	2048	2.8328	0.3894	7x	YES
Matrix transpose	256	2.3023	0.0575	40x	YES
	512	2.6405	0.1145	23x	YES
	1024	9.4109	0.2224	42x	YES
	2048	12.8485	0.4364	29x	YES
Matrix-matrix sub.	256	0.3379	0.0829	4x	YES
	512	0.6973	0.1600	4x	YES
	1024	1.4538	0.3105	5x	YES
	2048	2.7849	0.5894	5x	YES
Matrix-matrix prod.	8	43.2721	0.0192	2259x	YES
	16	260.6750	0.0517	5040x	YES
	32	441.7003	0.1360	3248x	YES
	64	4212.5503	0.3354	12561x	YES

As can be seen from the table [1] above, results are pretty clear: time needed, for each function computed by the GPU, is lower than one second [2]. The speed up reached is also remarkable with a peak of 12561 times faster than sequential solution in matrix-matrix product. An unexpected outlier is the matrix transpose operation, where the sequential solution is not performing very well. More expected was the incredible performance reached by the parallel version of the matrix-matrix product compared with the sequential one, in fact, in order to achieve results in considerable time the size of the matrices involved was reduced a lot. In conclusion, in sequential solutions the time required to compute the linear algebra functions proposed increases, roughly, linearly with respect to the size of data structures involved. The single exception is the matrix-matrix product where time needed increases very fast, maybe we might say exponentially with respect respect the size of the data. On the other hand, parallel versions are performing very well, of course, time increases when data grow but they permit to handle even more than 2 GB of data, obtaining results in a considerable time. (See charts at the end of the report)

Let's now evaluate the performance reached by the Principal Component Analysis algorithm. In this case results are obtained running the `main.cu` file. Performances are evaluated by collecting the execution time on different data size. Benchmark data are generated randomly. In the following table results are shown:

Algorithm	Size (MB)	CPU time (s)	GPU time (s)	Speed up	Error
PCA	4	179.3397	13.4208	13x	0.004260
	8	354.9910	24.5382	14x	0.009630
	16	903.7424	46.7842	19x	0.018666
	32	1751.9310	79.8038	22x	0.083864
	64	6795.9517	147.5944	46x	0.333195

The table shows that the execution time increases exponentially, with respect to the size of the data, in the sequential version of Principal Component Analysis. In the parallel one the execution time increases, but much more slowly than the sequential version, in fact, the GPU was 46 times faster than the CPU in the peak. In the last column of the table, can be seen the average error computed on the transformation of the data obtained from sequential a parallel versions, the error increases as data increases but sill tends to zero meaning that both transformations are good. Finally, I want to notice that the execution time of PCA is strictly related, of course, to the size of the data, but even with the number of eigenvectors computed, their precision, the number of maximum iteration set, and in parallel version, the number of memory transfers performed. In my opinion, seeing the results obtained from both versions, I can conclude by saying that the sequential approach is not suitable when huge amount of data have to be handled, since the execution time, literally, explodes. In such cases the parallel approach is recommended. The chart of the execution time of PCA is shown in Figure 1.

[1] The last column in the table indicates if the results obtained from sequential and parallel solution are equivalent or not

[2] GPU's results showed are obtained transferring a single memory tile, transferring more than one, time increases due the overhead introduced.

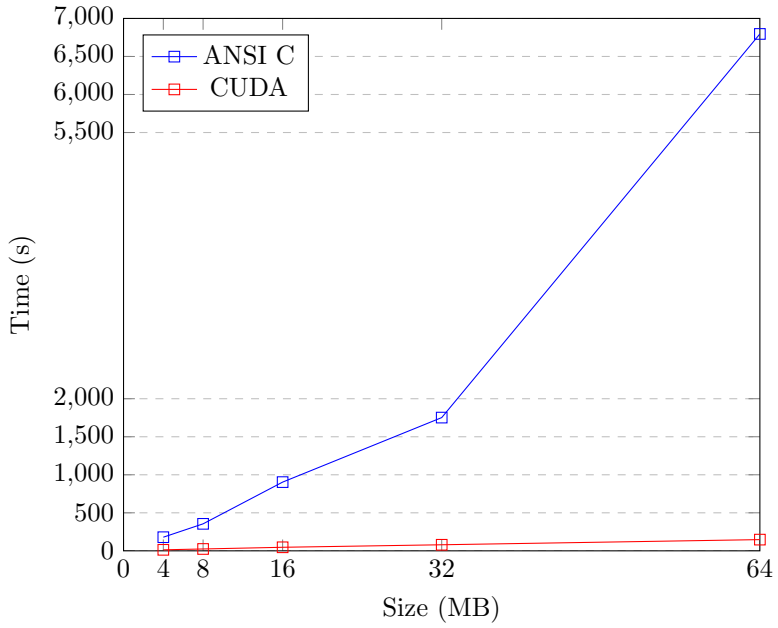

Figure 1: PCA execution time

5 Profiling results

Kernel profiling is the process of collecting data about the execution of the kernel, such as which functions are being called, how often they are being called, and how much time they are spending executing. This data can be used to identify performance bottlenecks in the kernel and to optimize its performance. Several metrics are made available by Nsight Compute tool, each one is focused on particular aspect of the kernel execution, from the occupancy achieved per streaming multiprocessor, to branch divergence, bank conflict, memory patterns such as global loads and stores and so on. In my project I payed attention to the following metrics [3]:

1. execution time: execution time of the kernel
2. achieved_occupancy: ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor
3. branch_efficiency: ratio of branch instruction to sum of branch and divergent branch instruction
4. gld_efficiency: ratio of requested global memory load throughput to required global memory load throughput expressed as percentage.
5. gld_throughput: global memory load throughput
6. gld_transactions_per_request: average number of global memory load transactions performed for each global memory load.

7. gst_efficiency: ratio of requested global memory store throughput to required global memory store throughput expressed as percentage.

8. gst_throughput: global memory store throughput.

9. gst_transactions_per_request: average number of global memory store transactions performed for each global memory store

Since in Nsight Compute tool the name of the metrics have been changed, at this link you will find the conversion table. The results shown in the table below were obtained through kernel profiling on randomly generated benchmark data with a size of 1 GB:

Kernel	#1 (ms)	#2 %	#3 %	#4 %	#5 GB/s	#6 sec/req	#7 %	#8 GB/s	#9 sec/req
cudaVSDivision	10.38	81.43	100	100	103.45	4	100	103.45	4
cudaEDistance	30.12	91.55	100	100	71.31	4	12.50	1.11	1
cudaMSProduct	11.43	82.41	100	82.50	117.39	2.50	100	93.91	4
cudaMTranspose	25.94	87.72	0	70.83	62.10	2	25	165.60	16
cudaMMSub	14.36	85.49	100	90.28	168.24	3	100	74.77	4
cudaMMProduct	23610	99.99	100	99.97	92.43	3.96	100	0.0424	4

In general, a good kernel should maximize device resources utilization, minimize divergence between threads, and ensure coalescent and aligned memory accesses. The last aspect is very important because memory accesses can become the main bottleneck of a kernel. To understand when the resources of a device are well exploited, may be useful observing the achieved_occupancy metric (#2) and then, from the outcomes reached, we can deduce that kernels developed are using the majority of the device's resources. Branch divergences occur at the warp level and can cause a performance degradation since each branch is then executed sequentially. The branch_efficiency metric (#3) is used to keep track of branch divergences, and when its value tends to 100%, it means that the most threads follow the same execution path. Therefore, seeing values in the table, we can say that all kernels, except cudaMTranspose, create no divergences or very small ones. Why the branch_efficiency metric of cudaMTranspose is equals to zero is not clear to me, but I think the reason is that reading data in "a certain direction" and then writing it in "another" causes, in somehow, a huge divergence between threads in a warp. Memory access patterns might have an important impact on kernel's performance, both global memory loads and stores should be analyzed, and interesting metrics for doing so are: gld_efficiency (#4), gld_throughput (#5), and gld_transactions_per_request (#6) for global memory loads; while gst_efficiency (#7), gst_throughput (#8), and gst_transactions_per_request (#9) for global memory stores. Looking at the metrics of global memory loads (#4, #5, #6) the outcomes indicate that loads performed are coalescent. The first fact supporting my statement is that gld_efficiency metrics (#4) is quite high tending to 100%, and the second one, is that the average number of memory transactions required to satisfy a memory load request by threads in a warp (#6) is quite low (around 4). Focusing now on effective memory bandwidth (#5) scores, it can be said that peaks reached by kernels are quite interesting, but maybe some improvement can be performed since the maximum memory bandwidth is 320 GB/s according to Nvida T4 specs [4]. Analyzing global memory stores results (#7, #8, #9), again, in general, kernels achieve satisfactory performances. Some discrepancies can be observed in cudaEDistance and cudaMTranspose kernels: in the first one, gst_efficiency (#7) and

gst_throughput (#8) scores are low due to the fact that one threads per block writes the partial sum of the parallel reduction to global memory, while, in the second one, gst_efficiency is low due to the high average number of memory transaction needed to satisfy a memory store request (#9).

To summarize, in my opinion, the overall kernels performances are good. Of course some improvements might be made, in particular some effort may be spent on optimizing memory throughput. However, the cudaMTranspose kernel does not have very good performance, so, a complete kernel review is necessary.

References

[1] Principal-Component Analysis; Mining of Massive Dataset 3rd edition, Chapter 11, Section 11.2 page 412; http://infolab.stanford.edu/ ullman/mmds/ch11.pdf

[2] Eigenvalues and Eigenvectors of Symmetric Matrices; Mining of Massive Dataset 3rd edition, Chapter 11, Sections 11.1.1, 11.1.3 page 406; http://infolab.stanford.edu/ ullman/mmds/ch11.pdf

[3] Metrics for Capability 7.x; Nvida CUDA documentation; https://docs.nvidia.com/cuda/profilerusersguide/index.html#metricsforcapability7x

[4] Next-Level Acceleration Has Arrived; Nvida website; https://www.nvidia.com/en-us/data-center/teslat4/

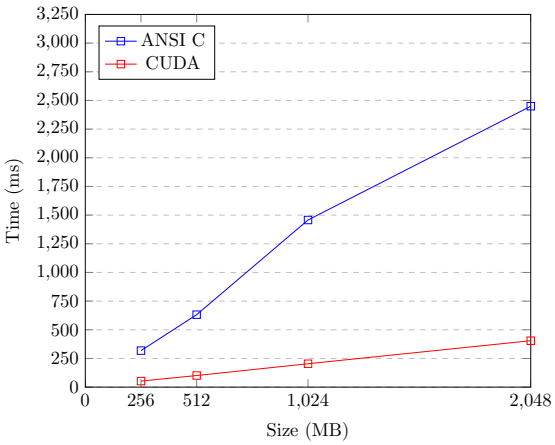

Figure 2: Vector-scalar division

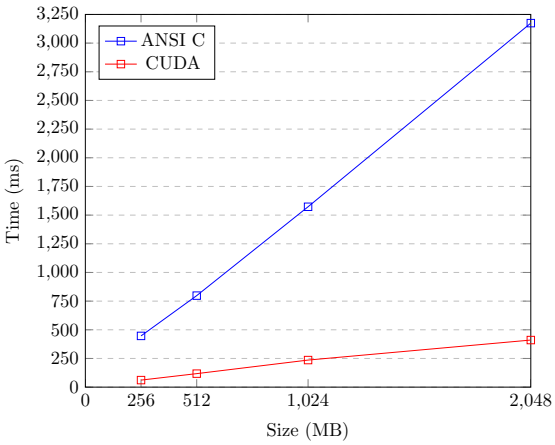

Figure 3: Euclidean distance

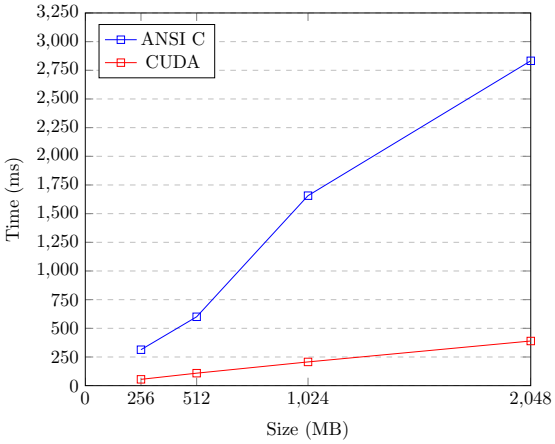

Figure 4: Matrix-scalar product

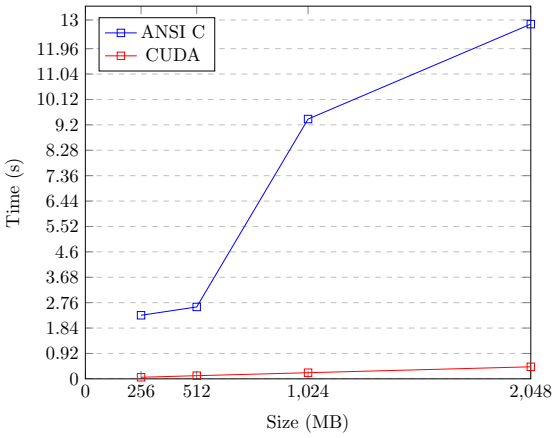

Figure 5: Matrix transpose

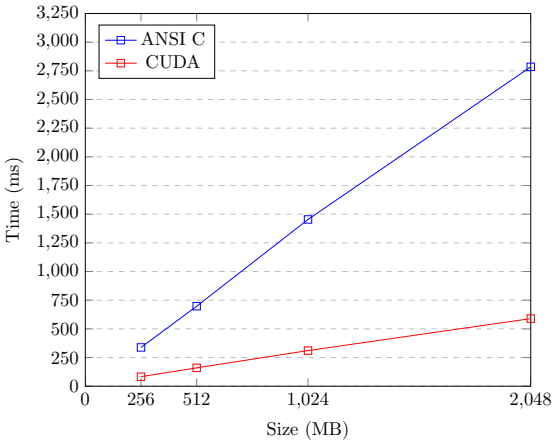

Figure 6: Matrix-matrix subtraction

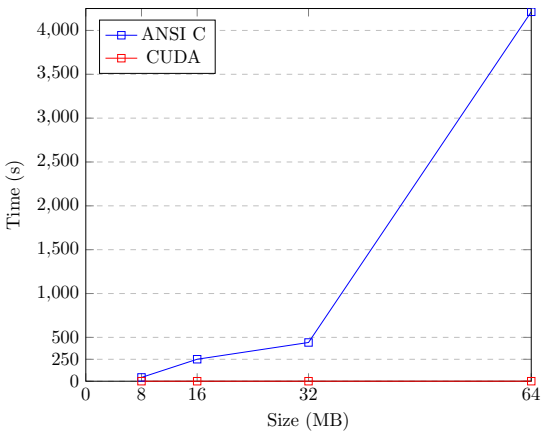

Figure 7: Matrix-matrix product