ISSN 2281-4299

DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA UNIVERSITÀ DI ROMA

Design, Implementation and Performance Evaluation of a Stochastic Gradient Descent Algorithm on CUDA

Emanuele De Falco

Technical Report n. 11, 2015

Design, Implementation and Performance Evaluation of a Stochastic Gradient Descent Algorithm on CUDA

Emanuele De Falco

Sapienza University of Rome
emanuele.defalco@gmail.com

Abstract

Stochastic Gradient Descent, a stochastic optimization of Gradient Descent, is an algorithm that is used in different topics, like for example for linear regression or logistic regression. After the Netflix prize, SGD start to be used also in recommender systems to compute matrix factorization. Considering the large amounts of data that this kind of system has to elaborate, adapt the algorithm on a distributed system or parallelize it is a good idea to improve performance. One way to do this is by using GPGPU, that thanks to its characteristics it's now days a good solution for parallelize an application. With this work, we are interested in analyze how SGD works on a GPGPU environment that is designed with a CUDA architecture, starting from an existing implementation for parallel environments and then adapting it to exploits all characteristics that a GPU of this kind provide.

Keywords: Stochastic Gradient Descent, CUDA, GPGPU, Recommender Systems, Parallel Programming

Design, Implementation and Performance Evaluation of a Stochastic Gradient Descent Algorithm on CUDA

Emanuele De Falco

Sapienza University of Rome
emanuele.defalco@gmail.com

Abstract

Stochastic Gradient Descent, a stochastic optimization of Gradient Descent, is an algorithm that is used in different topics, like for example for linear regression or logistic regression. After the Netflix prize, SGD start to be used also in recommender systems to compute matrix factorization. Considering the large amounts of data that this kind of system has to elaborate, adapt the algorithm on a distributed system or parallelize it is a good idea to improve performance. One way to do this is by using GPGPU, that thanks to its characteristics it's now days a good solution for parallelize an application. With this work, we are interested in analyze how SGD works on a GPGPU environment that is designed with a CUDA architecture, starting from an existing implementation for parallel environments and then adapting it to exploits all characteristics that a GPU of this kind provide.

1 Introduction

Stochastic Gradient Descent is commonly applied in different scenarios where we need to minimize an objective function, for example to implement the k-means algorithm or for linear and logistic regression. One of these are also recommender systems, where SGD is used to realize a particular implementation called collaborative filtering (that starting from existing relations between users and items tries to found new relations between them) through matrix factorization. Indeed, this could be described as a minimization problem where we want minimize the error between the real matrix and the approximation matrix. Therefore, SGD could be applied to solve this problem.
SGD is implemented in several open-source data analysis libraries and frameworks likes Weka, Apache Mahout, GraphLab, ecc. Some of these are distributed, and in that case get benifits by working in batch on large clusters. However, these are complex and expensive architectures, and consequently are unsuitable to perform interactive computations.

On the contrary, GPGPUs offer performance of a super-computing-like on a single workstation thanks to the possibility to execute an high number of threads in parallel and offered with a cheap architecture.
Therefore, our goal is study to which extent GPGPUs can empower an iterative exploration of large datasets with SGD on a single machine equipped with a CUDA architecture.
Actually, Kareem et al. in [1] provide a solution to implement SGD on a GPGPU by using OpenCL. However, in this solution it is used a graph representation of the rating matrix, and accesses on the GPU global memory that are performed don't take advantages of *coalescing*, that can easily lead to poor performance.
In our solution we used a linear representation of the rating matrix, in *COO format* [2] to have a low memory occupancy and take advantages of coalesced memory accesses, that as we will see it's a strength point of this work.

2 Stochastic Gradient Descent

SGD, Stochastic Gradient Descent, is a stochastic optimization of the Gradient Descent algorithm, a well known technique to solve minimization problems. The basic idea behind GD is that because we want to minimize a value of a function, we compute the gradient. Indeed, this is a vector that identify a direction where the function increases. So, starting from a random initial value, we will move in the opposite direction of it, until we reach the minimum of the function.
Formally, given a function of the form

$$F(x) = \sum_{i=1}^{n} F_i(x) \tag{1}$$

in each step, gradient descent update the value of x as follow:

$$x \leftarrow x - \mu \nabla F(x) = x - \mu \nabla \sum_{i=1}^{n} F_i(x) \tag{2}$$

where μ is what is called *learing rate*: it's a positive parameter that is used to define the step size in the gradient descent search.
Computing the gradient on a small training set is fine, but when this grow to much could be much expensive.

This is the reason why we need an approximation like SGD. Indeed, in this case we don't compute the gradient on all the training set, but we approximate it in an incremental way by computing the gradient on a single sample:

$$x \leftarrow x - \mu \nabla F_i(x) \tag{3}$$

and then iterate over all of them.
Each iteration over all the training samples could be computed multiple times, until a condition of convergence is reached. Each of this iteration is called *epoch*.

3 SGD in Recommender Systems

SGD could be used to perform matrix factorization[3], a collaborative filtering technique used in recommender systems. The idea is as follow: from ratings, we can make a sparse matrix, called rating matrix and indicated as R, where rows of this are users and columns are items. Matrix factorization founds two matrix, each one with dimensionality less than R, whose dot product approximate the rating matrix, by using only non-empty entries in the computation. This two matrices, that we indicate as P and Q, are respectively representations of users features and items features.
This could be described as a minimization problem, where we want to minimize the error between the rating matrix and the approximation matrix. The objective function of the problem that we could use is the *regularized square loss*, expressed as:

$$L(P, Q) = \sum_{(i,j) \in P,Q} (r_{ij} - p_i q_j)^2 + \lambda(||P||_F^2 + ||Q||_F^2) \tag{4}$$

where $||\cdot||_F$ is the Frobenius norm and $\lambda \geq 0$ is a regularization factor to avoid overfitting[4].
Because this is a minimization problem, we can apply SGD to solve it. This is performed by firstly computing the error between a training point in the rating matrix and its approximation:

$$e_{ij} = r_{ij} - p_i q_j \tag{5}$$

and then using this value to update matrices P and Q:

$$p_i = p_i + \mu(e_{ij} q_j - \lambda p_i) \tag{6}$$

$$q_j = q_j + \mu(e_{ij} p_i - \lambda q_j) \tag{7}$$

This operations are performed for each training point in the rating matrix for each epoch.
Implement a parallel version of this algorithm is not an easy operation. Indeed, each update of P and Q influences subsequent updates.
Hogwild![5] provide a lock-free approach to implement SGD in a parallel environment. Parallel thread will randomly select elements (one for each thread) from the rating matrix,

and will compute stochastic updates on them. Due to the fact that R is highly sparse, the probability that two independent threads will select two elements that share the same row or column is very small. Then, we can have a very small error introduction into the computation.

4 SGD on CUDA

Our solution to perform SGD on a CUDA GPU is a version of Hogwild! adapted to exploit the underlying architecture and very similar to the one in *Apache Mahout*.
In each epoch, first of all we randomly shuffle the elements in the rating matrix to perform randomization of accesses. This is an important step, not only for having a small error introduction, but also to obtain a faster convergence[6]. Then, for each training point, we will have a thread that perform a kernel function for the stochastic update. In more details, we will launch multiple times a number of blocks equals to the number of Streaming Multiprocessors in the GPU, each one with a number of threads equals to the number of cores per SM.
When an epoch is complete, we compute the value of the loss function. This is required to verify the convergence of the algorithm (we define that the algorithm is converged when the variation of the loss function in two consecutive epochs is less than 0.01%) and to implement an adaptation of the learning rate μ with the bold driver heuristic[7]. This technique is very important, thanks to which we are able to obtain a faster convergence.
The main problem of this solution is given by global memory accesses. Indeed, due to the shuffling of the training points, needed to have a small error introduction, it's impossible to perform coalesced memory accesses. This aspect has a big impact on performance that can be achieved. Indeed, if we compare the time required to complete an epoch in a solution that perform coalesced accesses (for example by removing the shuffling and performing sequential accesses into the rating matrix) with the solution with random accesses, we can see that we have a great advantage when we are able to perform coalescing. In the example provided with coalescing, each thread choose in deterministic way the training point where perform the computation. Results are showed in Figure 1.
However, even if we speedup the execution time of each single epoch, by removing the randomness of accesses we converge slower than previous solution and with a bigger error introduction.
So, we need to find a sweet-spot in the tradeoff between having coalesced accesses and randomized accesses. For this reason we have designed partial randomization schemes, that are based on logically dividing the matrix in blocks and randomizing the order in which these ones and elements that belong to them are computed. In particular, we have defined three main partial randomization schemes:

- *interblock randomization*, where we randomize the order of elements inside a block
- *intrablock randomization*, where we randomize the order of blocks

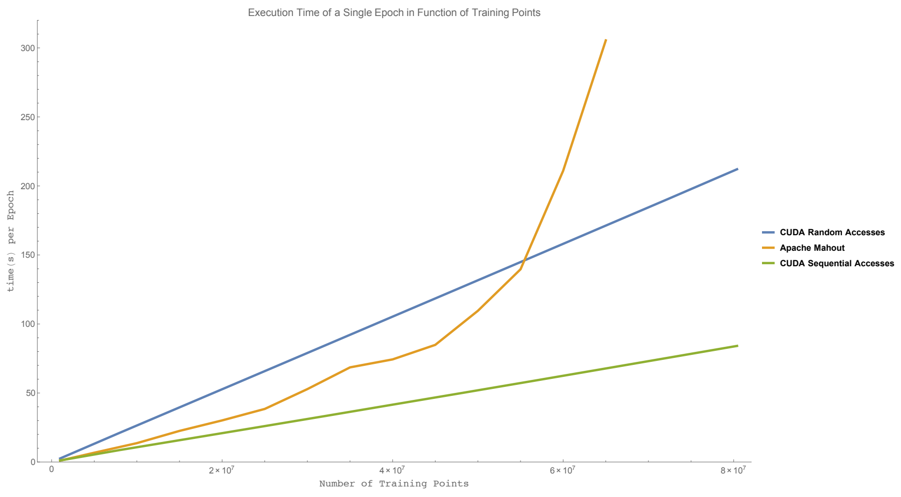

Figure 1: Comparison between coalesced memory accesses and uncoalesced memory accesses in terms of time required per epoch at the variation of the number of training points

- *interblock + intrablock randomization* that combines the two previous schemes.

In Figure 2 we can analyse the different results achievable by these schemes. It's interesting to see that in all cases, thanks to coalescing, we obtain results very close to a sequential accesses scheme, and consequently faster than a complete randomization scheme.

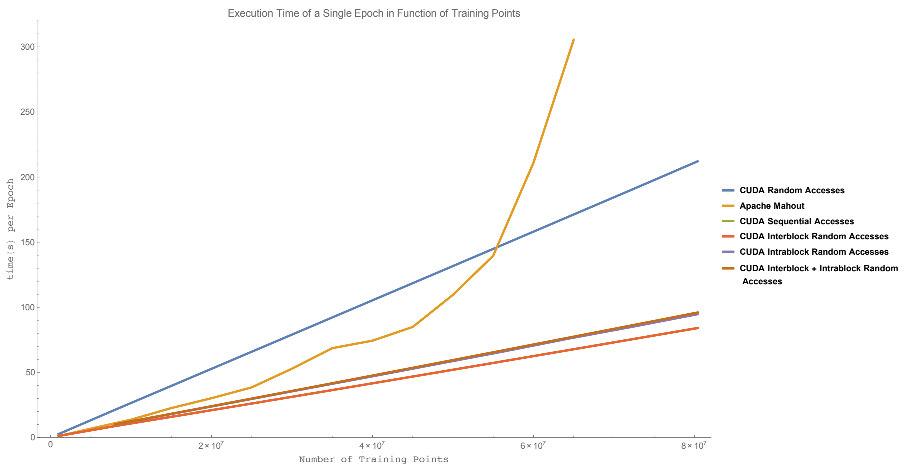

Figure 2: Comparison between different partial randomization schemes

5 Experimental Evaluation

We are not only interested to obtain a fast computation for each epoch but also a fast convergence with a small error introduction.
To analyse the error introduction in the computation we have used the *RMSE* as main metric.
Then, we have performed several tests on the netflix prize dataset[8], divided into a training set (with 80% of training points) and a test set (with 20% of training points) on a GPU Nvidia GeForce GT 540M with 2GB DDR3 VRAM and 2 Streaming Multiprocessor equipped each one with 48 cores.
In Figure 3 is possible to see how using partial randomization schemes without any expedient, even if we perform faster each single epoch, we obtain a slower convergence in addition to an high error introduction.

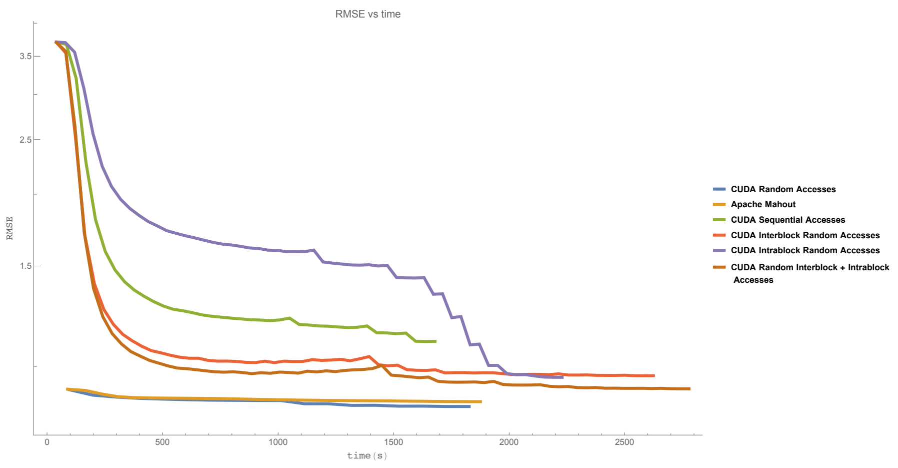

Figure 3: Comparison between different partial randomization schemes in terms of RMSE variation in time

To improve partial randomization schemes, we have defined four main parameters to drive them:

- n, the size of a threads block
- N, the size of a matrix block
- r, define the level of interblock randomization
- s, define the level of intrablock randomization

What we have obtained is that, if variating r and s in the range [0,1] don't provide significant results (therefore its advisable to choose or not a particular randomization scheme),

this is not true for parameter n and N. Indeed, by decreasing the value of n we decrease the number of threads in each SM, and consequently reducing the time necessary to the warp scheduler to switch from a warp to another. Be careful that we have a lower bound limit for n to avoid waste of resources, that is equals to the number of cores per SM. Therefore, this value depends from the specific GPU.
Instead, by variating the parameter N we also influence the amount of randomization that we perform, until we get closer to a complete randomization scheme, that could be generalized by assuming N equals to 1.
In Figure 4 we can see results of these considerations, where using the correct configuration we obtain an RMSE very close to the one achieved with a complete randomization scheme but with a faster convergence.

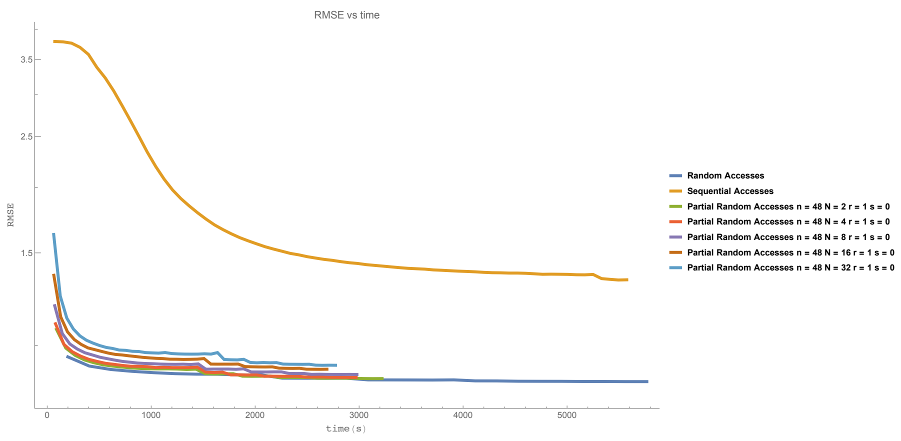

Figure 4: Comparison between different partial randomization schemes

6 Conclusions

In this work we have analyzed how SGD could be implemented on CUDA. We have seen that, caused by how memory accesses are done into a CUDA GPU, implement SGD in parallel way as Hogwild! describe don't bring any advantages in terms of epoch performance. However, thanks to the possibility to implement in a fast parallel way a module that analyze the behaviour of the algorithm during the epochs, it's easy to implement an adaptive learning rate like bold driver. Thanks to this one, we are able to achieve a faster convergence.
To solve the problem of memory accesses, we have seen that we can exploit coalescing accesses into memory by changing the randomization scheme. Indeed, a completely randomization make unfeasible coalescing accesses, that are instead done with the three schemes designed: interblock randomization, intrablock randomization, interblock + intrablock ran-

domization. Each of these allows us to obtain better performance in terms of time needed per epochs and thanks to this, even if in some cases we need more epochs to converge, we converge faster.
All of this schemes are also based on division of the rating matrix in blocks, and variating their size we can improve the final RMSE value and the convergence of the algorithm, reaching results very closer to the complete randomization scheme.
Obviously, at variation of blocks dimensions corresponds a variation of the importance of each partial randomization schemes: at an increasing of N corresponds an increasing importance of the intrablock randomization and a decreasing importance of the interblock randomization; instead, at a decreasing of N correspond an increasing importance of the interblock randomization and a decreasing importance of the intrablock randomization.
In conclusion, we can say that to implement SGD on a CUDA architecture we have to use partial randomization strategies, in particular the interblock randomization with a small block size. The use of the intrablock randomization could be avoided if this size is near to 2.

7 Future Works

After all of this tests, continues to be difficult to say if implement SGD on a GPU is a good idea. Indeed, is true that, with all the adjustments done and thanks to a fast computation of the value of the loss function, we are able to achieve a fast convergence of the algorithm compared for example to Mahout, but is also true that all of this tests are done on a single machine. So, if for Mahout we can exploit the advantages of working on a platform like Hadoop when we need to scale on multiple machines, this is not so immediately when we are working on a GPU. Indeed, even if exists algorithms that provide the possibility to distribute SGD over multiple machines, this ones required coordination between nodes of the cluster. This, in a GPU environment, means data transferring between device memory and host memory, that has to be reduced as much as possible.
So, will be interesting in future to analyze this aspects, considering different mechanisms of coordination and matrix partitioning to reduce as much as possible data transferring between the two memories and determining the scalability of this implementation on multiple machines.

References

[1] Rashid Kaleem, Sreepathi Pai and Keshav Pingali *Stochastic Gradient Descent on GPUs* GPGPU-8,2015.

[2] Nathan Bell and Michael Garland *Efficient Sparse Matrix-Vector Multiplication on CUDA* NVIDIA Technical Report NVR-2008-004, 2008.

[3] Y. Koren, R. Bell, and C. Volinsky *Matrix factorization techniques for recommender systems Computer* vol. 42, pp. 30,37, 2009.

[4] Fabio Petroni and Leonardo Querzoni *GASGD: Stochastic Gradient Descent for Distributed Asynchronous Matrix Completion via Graph Partitioning* RecSys'14, 2014

[5] F. Niu, B. Recht, and C. Re *Hogwild!: A lock-free approach to parallelizing stochastic gradient descent* 2011

[6] F. Makari, C. Teflioudi, R. Gemulla, P. Haas, and Y. Sismanis, *Shared- memory and shared-nothing stochastic gradient descent algorithms for matrix completion* 2013

[7] R.Battiti and F.Masulli *The bold driver method* International Neural Network Conference vol. 2, p. 758, 1990.

[8] http://www.lifecrunch.biz/archives/207.