Our work relates to several different areas with most emphasis on GPU optimized massive parallel computing and also on predictive analytics area. Basically, the second focus area that our thesis relies on results from the actual production grade systems and experimental results of our project. Nevertheless, the main focus of our present work is related to the current state-of-the-art research and technologies for GPU-based parallel numeric computing. As a result, we will briefly describe the following subjects that relate to our work:

1. Parallel computing and processing of tensor graphs
   1. Tensor graphs
   2. State-of-the-art in tensor graph computation engines
   3. State-of-the-art in numerical parallel computing based on GPU
   4. Performance considerations
   5. State-of-the-art tensor graph architectures memory requirements
2. Predictive analytics related work
   1. Real life problems
   2. Predictive analytics graph architectures
   3. Baseline analytics

## Parallel computing and processing of tensor graphs

### Tensors

Mathematically, tensors are N-dimensional arrays that have various applications in numerical calculus, linear algebra, big-data or any other field where mathematical computation is involved. Therefore, Google and The Theano Development Team developed **Tensorflow** [1] and **Theano** [2], respectively, two frameworks used for defining tensor computational graph models and for executing such models on hybrid environments (CPUs and GPUs using **CUDA** [3]).

The power of these framework consists in representing the computation as a directed graph where each node is an operation, whereas each edge is a tensor. In other words, all the data flows through tensors and each node computes a mathematical operation on the edges that reach its input. Their intelligent graph pruning mechanism leads to evaluation only of the part of the graph that is involved in a certain operation, avoiding redundant mathematical computation and optimizing run-time memory usage.

### State-of-the-art in tensor graph computation engines

#### Tensorflow

Google developed this framework in order to replace DistBelief [4], which was successfully used in production since 2011 for their AI tasks that involved Neural Networks. DistBelief was designed as a framework in which neural networks are seen as DAGs of layers, whereas Tensorflow represents each mathematical operation as a node in the computation graph, being suitable for any scientific computation task. An important aspect is that the Tensorflow’s directed acyclic graphs can be executed on a large variety of platforms such as mobile devices, multiple-GPU machines or clusters of multiple-GPU machines (distributed implementation). Besides the CPU implementation of the operations, Tensorflow provides GPU kernels for most of them, which leads to a highly parallelizable environment that uses CPU and GPU cores.

Tensorflow handles itself the execution in hybrid environments (where exist also GPU cards) by placing the nodes onto each existing device and then choosing the device where the estimated running time is the lowest. Also, the framework cares of the communication between the subgraphs that are mapped on different devices.

As we mentioned, Tensorflow provides also distributed capabilities, but this thesis will focus on the single-machine environment, because we concentrate on allocation and optimization of directed acyclic graphs in hybrid environments where scarcity of memory and computation cores might be an issue.

#### Theano

The main aspect that dissociates Theano of Tensorflow is the flexibility that this framework brings for single-machine computing. One main drawback of Theano is that it do not posses multiple GPU implementation, which is essential in order to create a highly parallelizable environment for computation graphs. Moreover, it is used with predominance for experiments, whereas Tensorflow is employed in all production-grade systems of Google and other big companies like Intel, NVIDIA, Airbnb or Snapchat.

Our analysis of tensor computation engines can be summarized in the following table (Table 1):

Table 1

|  |  |  |
| --- | --- | --- |
| Framework | Pros | Cons |
| Tensorflow | 1. Abstraction based on computational graphs 2. Faster compile time than Theano 3. Visualization tools like Tensorboard 4. Parallel execution 5. Multiple CPUs and multiple GPUs support 6. Distributed implementation 7. Google continous development – better for production-grade systems | 1. It runs dramatically slower than other frameworks |
| Theano | 1. Abstraction based on computational graphs 2. Parallel execution 3. More flexible when it comes to single-machine computing | 1. Longer compile time than Tensorflow 2. Single GPU support 3. Theano models cannot be deployed on mobile devices |

To further substantiate our related work we have also analyzed the research done by Bahrampour et al. [5] and presented their research in the following table (Table 2):

Table 2

|  |  |  |
| --- | --- | --- |
| Framework | Pros | Cons |
| Tensorflow | 1. Multi-threaded CPU using Eigen library 2. Multi-GPU support | 1. Low performance on a single GPU 2. Only CUDA support for GPU computing |
| Theano | 1. Multi-threaded CPU using Blas and OpenMP 2. Most flexible for single GPU computing 3. CUDA and OpenCL support for GPU computing | 1. Single GPU support |

Although there are multiple tensor computation engines similar to Tensorflow, they do not posses the GPU allocation, computation and distribution capabilities, this being the main reason for our choice of state-of-the-art technology.

### State-of-the-art in numerical parallel computing based on GPU

Nowadays, the most widely used programming options for numerical parallel computing based on GPU are **CUDA** [3] and **OpenCL** [6]. There are slightly differences between these two GPU acceleration options, with the main one being that CUDA is created by NVIDIA and OpenCL is open-source. Therefore, CUDA integration for NVIDIA GPU cards provides better speedup than OpenCL integration.

The core entities of these state-of-the-art frameworks for GPU acceleration are the *kernels,* which describe the instructions that will be run in a parallel fashion on the *devices.* The code snippets (**Code 1** and **Code 2**) define the CUDA and OpenCL kernels that compute the dot product between two vectors (Equation 1) for the particular case of matrix dot product. Very important to mention that matrix dot product represents the core of all predictive analytics tasks ranging from basic linear regression models to complex recommender systems.

|  |  |  |
| --- | --- | --- |
|  |  | (1) |

**Code 1** Two Matrix dot product – CUDA kernel

\_\_global\_\_ void

MatDotKernel(Matrix A, Matrix B, Matrix C)

{

int tx = blockIdx.x \* blockDim.x + threadIdx.x;

int ty = blockIdx.y \* blockDim.y + threadIdx.y;

if(ty < A.height && tx < B.width) {

float sum = 0;

for (int k = 0; k < A.width; k++) {

float elementA = A.elements[ty \* A.width + k];

float elementB = B.elements[k \* B.width + tx];

sum += (elementA + elementB);

}

C.elements[ty \* C.width + tx] = acc;

}

**Code 2** Two Matrix dot product – OpenCL kernel

\_\_kernel void

MatDotKernel(\_\_global float\* C,

\_\_global float\* A,

\_\_global float\* B,

int wA, int wB)

{

int tx = get\_global\_id(0);

int ty = get\_global\_id(1);

float sum = 0.0 // stores the element that that is computed by the thread

for (int k = 0; k < wA; k++) {

float elementA = A[ty \* wA + k];

float elementB = B[k \* wB + tx];

sum += (elementA + elementB);

}

C[ty \* wA + tx] = sum;

}

The above two kernels represent the computational engine on GPU cards for various predictive analytics tasks which require matrix multiplications. Our work do not include kernel implementations for various mathematical operations, because the tensorial computation framework that we use, namely Tensorflow, provides GPU CUDA kernels for most of them.

### Performance considerations

One important subject that our research and development work relates to is that of performance analysis and optimization when employing off-CPU jobs for numerical computing on GPU devices. Basically, in this area we have two different issues:

* numerical computation offloading to GPU
* evaluation of hidden operations within graphs

Numerical computation offloading to GPU, of jobs that usually and traditionally are processed either in multi-threaded CPU environments or in multi-processing CPU-based environment, has a very important potential bottleneck: GPU offload latency. The GPU offload latency simply consists in the time required to load the CUDA kernel and the actual data to be processed summed with the time required to download from GPU the results. This tensor load/download operation is applied to various tensor operations: from large tensors (sometimes over 1GB worth of data) down to small tensors. Intuitively the main issue here is to find the right balance of the time to upload/download data to the GPU device vs the speed gain resulted from the running of numerical operations on the GPU rather than CPU. Basically, we need to have where the variables are defined in below equation:

|  |  |  |
| --- | --- | --- |
|  |  | (2) |

As argued in one of our referenced researches by Lustig et al. [7] there are multiple factors that cooperate to our variable (with its two components: the upload and download times) as follows:

* the GPU kernel launch command itself requires time for allocation and preparation of execution regardless of data size
* there is a delay between the when data physically arrives on the destination GPU and the when data is ready for processing by the GPU kernel (due to specific synchronization issues).
* finally, there is a delay introduced by the actual API that makes the low-level upload/download operations of both data and kernel code on and from the GPU device.

The second issue related to the performance considerations when using GPU devices for massive parallel numerical tensor computations is that related to the actual evaluation of tensor graph memory and computing requirements. For the particular case of our experiments where we use deep acyclic graphs for predictive business analytics tasks we strongly depend on both data structure and graph execution/optimization algorithm. As an intuitive explanation we can think that simply looking at one's defined DAG we can infer the required memory allocation and thus we can simply allocate and transfer the data tensors on the GPU device memory. Nevertheless, most directed acyclic graphs optimization algorithms require more than one so-called copy of the entire graph and that leads to 2x, 3x or even 4x memory requirements for the whole process of graph execution and optimization. To finish our intuition presentation, we can conclude that in most cases we can either decide to allocate partially the required graph memory – certainly at the cost of additional GPU offload latency - or allocate the whole required memory for all operations.

While the first previously mentioned issue is mostly covered by latest researches and development of NVidia and particularly Nvidia CUDA technology that we are strongly relying upon, the second issue - that of hidden tensor graph allocation requirement based on optimization algorithm – falls entirely in our task of research, development and experimentation.

### State-of-the-art tensor graph architectures memory requirements

With the development of big-data and the need of processing this huge amount of data in order to obtain state-of-the-art results in the area of predictive analytics, it was demonstrated that fully-connected neural directed acyclic graphs [8], as well as recurrent neural directed acyclic graphs [9] are known to be the heart of the most efficient predictive analytics systems, namely recommender systems [10] [11] [12] [13]. It is worth mentioning that our system can handle any tensorial directed acyclic graph for parallel computing, but this chapter comes to demonstrate our research in CPU-GPU parallel computing and memory optimization specifically using these state-of-the-art tensor graph architectures.

Every neural network based tensorial computation graph is composed of a set of nodes (operations) and a set of tensors (parameters) which are trained to learn a given task (input data). A cost function (error function) represents the way a neural network based tensorial graph is evaluated. In order to update the parameters with respect to the computed cost function, in the computation graph is added another node called optimizer. The most efficient method of optimizing the parameters is known as Gradient Descent [14]. Gradient Descent optimizer calculates the gradients of each parameter with respect to the cost function and then update them accordingly with the computed gradients (Equation (*3)*). Gradient Descent method has been improved over time and therefore momentum-based gradient optimizers [15] have been discovered: Adagrad [16], Adam [17] etc. These momentum-based gradient optimizers update the parameters by adding a fraction of the previous gradients to the current ones (Equation (*4)*) in order to reach the global minimum of the cost function faster.

|  |  |  |
| --- | --- | --- |
|  |  | (3) |

|  |  |  |
| --- | --- | --- |
|  |  | (4) |

A fully-connected neural network based computational graph is represented in Figure 1. Each block in composed of a set of operations (*MatMul, Add* or any nonlinear transformation [18]) that are applied for a set of three tensors ( – input, – parameters). For each parameter, there exist the corresponding gradients . Therefore, the **total amount of necessary memory** for evaluating such a graph using **single-precision floating-point format calculus, i.e. 32 bits per parameter**, is presented in the below equation and it depends on the method used for parameters update.

|  |  |  |
| --- | --- | --- |
|  |  | (5) |

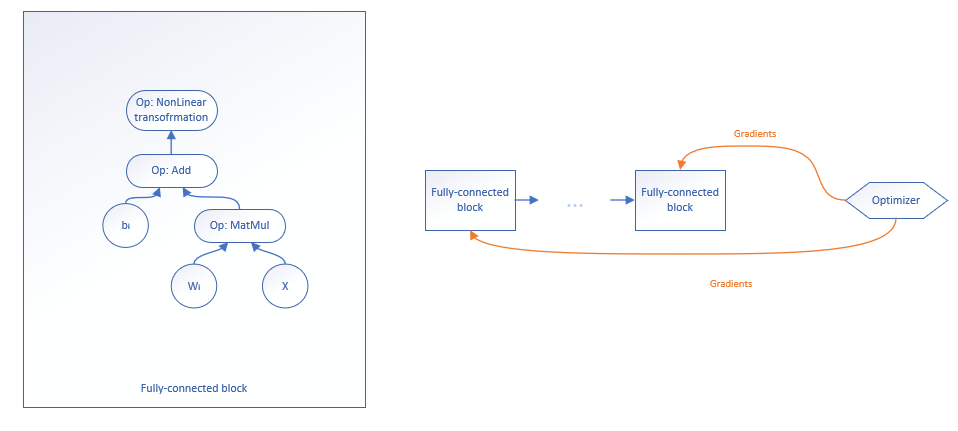


Figure 1 Fully-connected NN-based computational graph

A recurrent neural network based tensorial graph can process time series, keeping internally an accumulator which is a memory off all past time steps. More specifically, if a certain input (that represent a time step in a time series ) is fed to the graph, the recurrent graph will process this input also based on its memory and will update the memory accordingly in order to pass it to the next time step. Therefore, the algorithm that process a time series is presented below, using natural language:

* The memory is initialized with a vector of zeros, i.e. the graph does not have any information from previous events;
* For each timestep , the computational graph presented in Figure 2 is evaluated in order to compute the value of the cost function. Moreover, the memory is transferred to the next time step;
* The optimization process, which is applied after the computational graph is evaluated for the last time step, employs the same method of optimization of parameters as for fully-connected DAGs – gradient descent – but with a slightly difference: backpropagation through time [19] which introduce a new gradient of loss with respect to the hidden state.

To resume, the memory consumers for this particular case of tensor graph architecture are: – input, – parameters, – memory and the corresponding gradients . Consequently, **the total amount of necessary memory** for evaluating this particular graph using **single-precision calculus,** is presented in the below equation and it also depends on the method used for parameters update.

|  |  |  |
| --- | --- | --- |
|  |  | (6) |

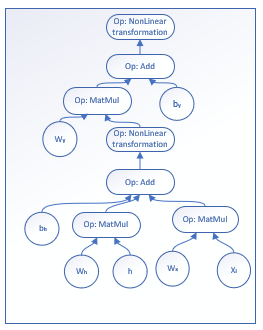


Figure 2 Computational graph of operations for a recurrent block

## Predictive analytics related work

The actual research in the area of computer systems’ architecture started from a real production-grade product involving the execution of computational graphs for business predictive analytics tasks – real time preparation of market basket analytics. For this particular case we already researched and developed our own two different Deep Learning architectures (as presented in our research paper „..........”) that are capable to analyze unstructured transactional information and construct a complete relational graph between any two different entities. To be more precise, our architectures generate different product “meta-maps” which describe the relationship between each and every product and provide our clients insights about how to generate market baskets of complementary (very likely to be bought together for a certain need) products. The computational graphs resulted from the presented architectures in our mentioned research paper will be run using our developed system parallel and highly optimized fashion.

### Real life problems

The main need of such a predictive analytics model that can easily recommend market baskets just by computing cosine similarity [20] in the latent space of the products comes from the increasing evolution of online advertisements that should target very well the users in order to reach their demands. This is an important aspect that make people to visit again the sites or to use the applications without being annoyed by not interesting advertisements. Consequently, the financial success of retailers is now strongly related to their users’ retention, which is an effect of tailoring for their tastes every single moment.

Thereby, our predictive analytics models resolve this real life problem and our parallel numerical computation system is capable to produce a proof-of-concept in real-time and provide our clients with the confidence that the product delivers the promised features without any ETL process.

### Predictive analytics graph architectures

Our predictive analytics graphs architectures employ, besides the already presented state-of-the-art tensor graph architectures, an additional subgraph which is a representation of embedding concept [10]. This technique maps factor variables, such as products, to a latent space of vectors of real numbers. More specifically, each factor variable is embedded into a vector , which is a precise representation in a semantical multi-dimensional space and which defines the relationship with every different factor variable (product).

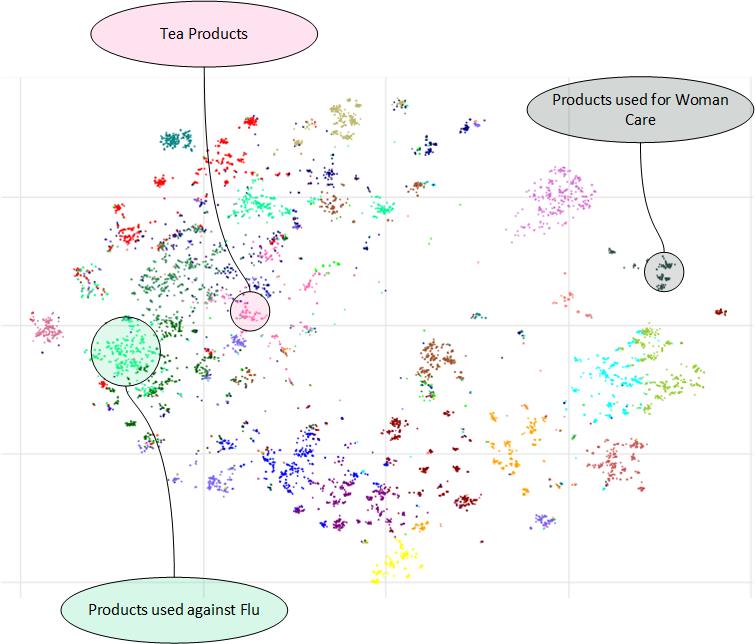
Thereby, the memory required by the computational graph will be extended with an additional number of bytes , where is the total number of factor variables and represents the dimension of an embedding.

As we mentioned, we desire to generate six latent spaces of embeddings using the two presented tensor graph architectures for three different seasonal transactional datasets. Each embedding representation of products is then transformed in a two-dimensional “meta-map” employing a technique called t-Distributed Stochastic Neighbor Embedding (t-SNE) [21]. More specifically, our system is able to run, for this particular case, 6 Tensorflow sessions, each on a different job that allocates the tensors of the computational graph both on GPU and CPU, considering the memory requirements for each job and the memory limitations for each device (more details in Proposed Solution and Implementation Details chapters).

TODO memory for all 6 graphs!!!

### Baseline analytics

Our predictive analytics architectures were successfully tested on a pharma-retail dataset of over 200 million transactions and an actual output of the “meta-maps” generated using t-SNE algorithm over our multi-dimensional embedding space is presented in TODO Figure. This particular output of the experiment is just one of the six expected outputs. In the figure, we have highlighted three different regions that actually contain strong semantic meanings: tea products, products used for woman care and products used against flu.



In order to obtain these results, the parallel computing system basically needs to feed the three seasonal datasets to the mentioned graph architectures in six parallel execution sessions. Our tensorial computation system will evaluate the maximum required GPU resources for each of the six parallel sessions and fit all them optimally in the available GPU resources and use CPU capabilities complementary. Nevertheless, it is essential to mention that in our goal we will take into consideration the execution of highly parallelizable numeric computations on GPU and the execution of sequential-like operations on CPU.

# Bibliography

|  |  |
| --- | --- |
| [1] | M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, L. Kaiser, M. Kudlur, J. Levenberg and X. Zheng, "TensorFlow : Large-Scale Machine Learning on Heterogeneous Distributed Systems," 2015. |
| [2] | R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov and others, "Theano: A Python framework for fast computation of mathematical expressions," arXiv preprint arXiv:1605.02688, 2016. |
| [3] | J. Ghorpade, J. Parande and M. Kulkarni, "GPGPU PROCESSING IN CUDA ARCHITECTURE," *Advanced Computing: An International Journal ( ACIJ ),* vol. 3, 2012. |
| [4] | J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang and A. Ng, "Large scale distributed deep networks," in *proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12), F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.), Vol. 1. Curran Associates Inc., USA, 1223-1231.* , Lake Tahoe, Nevada, 2012. |
| [5] | S. Bahrampour, N. Ramakrishnan, L. Schott and M. Shah, "Comparative Study of Deep Learning Software Frameworks," arXiv preprint arXiv:1511.06435v3, 2016. |
| [6] | J. Stone, D. Gohara and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science & Engineering,* vol. 12, 2010. |
| [7] | D. Lustig and M. Martonosi, "Reducing GPU Offload Latency via Fine-Grained CPU-GPU Synchronization," *HPCA '13 Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA) ,* pp. 354-365, 2013. |
| [8] | J. Schmidhuber, "Deep Learning in Neural Networks: An Overview," *Neural Networks DOI: 10.1016/j.neunet.2014.09.003,* vol. 61, pp. 85-117, 2015. |
| [9] | S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Journal Neural Computation,* vol. 9, no. 8, pp. 1735-1780 , 1997. |
| [10] | T. Mikolov, I. Sutskever, K. Chen, G. Corrado and J. Dean, "Distributed representations of words and phrases and their compositionality," in *proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'13), C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger (Eds.), Vol. 2. Curran Associates Inc., USA, 3111-3119*, Lake Tahoe, Nevada, 2013. |
| [11] | Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32 (ICML'14), Eric P. Xing and Tony Jebara (Eds.), Vol. 32. JMLR.org II-1188-II-1196*, Beijing, China, 2014. |
| [12] | J. Pennington, R. Socher and C. Manning, "Glove: Global Vectors for Word Representation," in *EMNLP. 14. 1532-1543. 10.3115/v1/D14-1162*, 2014. |
| [13] | E. Smirnova and F. Vasile, "Contextual Sequence Modeling for Recommendation with Recurrent Neural Networks," arXiv preprint arXiv:1706.07684, 2017. |
| [14] | J. Kiefer and J. Wolfwitz, " Stochastic Estimation of the Maximum of a Regression Function," *Ann. Math. Statist.,* vol. 23, no. 3, pp. 462-466, 1952. |
| [15] | S. Ruder, "An overview of gradient descent optimization algorithms," arXiv preprint arXiv:1609.04747, 2017. |
| [16] | J. Duchi, E. Hazan and Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," *Journal of Machine Learning Research 12,* pp. 2121-2159, 2011. |
| [17] | D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," arXiv preprint arXiv:1412.6980, 2017. |
| [18] | P. Ramachandran, B. Zoph and Q. V. Le, "Searching for Activation Functions," *CoRR abs/1710.05941,* 2017. |
| [19] | Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," in *proceedings of the 22nd ACM international conference on Multimedia (MM '14). ACM, New York, NY, USA, 675-678. DOI: https://doi.org/10.1145/2647868.2654889* , 2014. |
| [20] | M. Grbovic, V. Radosavljevic, N. Djuric, N. Bhamidipati, J. Savla, V. Bhagwan and D. Sharp, "E-commerce in Your Inbox: Product Recommendations at Scale," in *proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '15). ACM, New York, NY, USA, 1809-1818. DOI: https://doi.org/10.1145/2783258.2788627* , Sydney, NSW, Australia, 2015. |