UNIVERSITATEA POLITEHNICA BUCUREȘTI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

LUCRARE DE LICENȚĂ

Execuția Optimizată a Grafurilor Directe Aciclice în Medii Hibride CPU-GPU
Iulie 2018

Laurențiu Gheorghe Piciu

Coordonator științific:

Prof. dr. ing. Nicolae Țăpuș

BUCUREŞTI

2018

UNIVERSITY POLITEHNICA OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE DEPARTMENT

BACHELOR THESIS

Optimized Execution of Directed Acyclic Graphs on Hybrid CPU and GPU Environments
July 2018

Laurențiu Gheorghe Piciu

Thesis advisor:

Prof. dr. ing. Nicolae Țăpuș

BUCHAREST

2018

TABLE OF CONTENTS

SINOPSIS

Această lucrare propune un sistem de procesare paralelă în medii hibride CPU-GPU a grafurilor computaționale aciclice, denumit HELIX[1], având ca obiectiv principal acela al alocării și execuției lor eficiente în medii și infrastructuri computationale mobile ce posedă resurse limitate de calcul paralel. Sistemul experimental rezultat din proiect și utilizat într-o aplicație comercială utilizează motorul și biblioteca de calcul tensorial TensorFlow - framework-ul pe care Google l-a dezvoltat și lansat Open Source in 2016 – acesta reprezentând mediul de rulare în producție al tuturor serviciilor inteligente realizate de această companie în ultimii ani. Ca aplicație reală, lansată in productie în mediu comercial in anul 2018, ce utilizează principiile execuției optimizate a grafurilor computaționale, va fi prezentat un sistem de analiză, inferență și predicție a propensității de cumpărare de produse a clienților unui mare lanț de farmacii. De menționat faptul că sistemul rezultat în urma dezvoltării experimentale se bazează pe o serie de componente inteligente ce utilizează grafuri computaționale aciclice ce descriu modele de analiză predictivă proprii domeniului Învățării Automate. Totodată trebuie menționat faptul că, deși TensorFlow reprezintă cel mai folosit framework de calcul tensorial în domeniul Deep Learning la acest moment atât în mediul academic, cât și în mediul comercial, acesta nu este optimizat pentru alocarea eficientă a grafurilor pe sisteme cu resurse computaționale limitate și cu atât mai mult pentru procesarea paralelă a mai multor grafuri de tensori în astfel de medii. Astfel, în cadrul acestui proiect de cercetare-dezvoltare, prin sistemul propus și prezentat in aceasta lucrare rezolvăm într-un mod eficient aceste probleme in contextul dat.

[1] Hyperloop graph Execution engine for Limited Infrastructure conteXts

ABSTRACT

This thesis presents the research and experimental development that targeted the objective of optimally allocating and executing multiple Directed Acyclic Graphs in a parallel fashion on heterogeneous CPU-GPU computing infrastructures that possess limited resource. The proposed experimental system, named HELIX[2], is using an Open Source low-level tensor computation engine, namely TensorFlow [1], a state-of-the-art framework that is able to represents, evaluate and optimize computations based on data-flow graphs. This low-level tensor computation engine has been employed by all Google AI-based production-grade services since 2016. Our proposed system is part of a large commercial product developed by a team of data scientists that has the purpose of applying Machine Learning and Deep Learning, in particular, in order to solve real-life problems in the area of Business Predictive Analytics. The actual real-life problems solved by the commercial product are related to the inference and prediction of customer product propensity based on various approaches, tasks that have to be done with mobile computing devices such as notebooks/laptops. Even though TensorFlow has become widely used for Deep Learning tensor-based computation, it is not optimized for highly efficient allocation of tensor graphs on hybrid limited resource environments or for executing multiple tensor graphs in a parallel fashion in such systems. Therefore, in this thesis we will argue that our research and resulted experimental system solves efficiently these proposed real-life issues.

[2] Hyperloop graph Execution engine for Limited Infrastructure conteXts

ACKNOWLEDGEMENTS

I would like to express my very great appreciation to professor Nicolae Țăpuș and to PhDc Andrei Ionuț Damian for continous support and constructive suggestions during the planning and development of this work. Also, I would like to acknowledge 4E Software SRL that supported the development of the products presented in this thesis and last but not least, the Data Science team that includes as junior Data Scientist one of my fellow colleague Simion-Constantinescu Andrei.

1 INTRODUCTION

1.1 Context of Computational Graphs Execution

The whole research started from an actual concrete production-grade commercial product involving the execution of computational graphs for business predictive analytics tasks such as inferring or predicting pharma retail business customer buying propensities.

More precisely, during the implementation of the case study, several shortcomings of the current state-of-the-art tensor processing frameworks were discovered, particularly when working with TensorFlow, probably the most advanced tensorial computation and optimization framework currently on the market. Those shortcoming, as it will be further detailed in the following chapters, derive from the need of using limited resources for tasks that traditionally and usually require large numeric computational resources both in terms of parallel computing power and memory allocation.

1.2 Problem – Parallel DAG Computing on Limited Infrastructures

As mentioned in the previous paragraphs of the thesis, the main issue tackled is related to the efficient allocation and execution of tensor computational graphs in limited resources environments and particularly in environments where GPU resources are available although with severe limitations in terms of numerical parallel computing cores and VRAM. TensorFlow is currently the most used and the most advanced tensor computation framework in production-grade systems - as a side note, the most important horizontal industry where TensorFlow is employed is Artificial Intelligence and Deep Learning in particular – however the mentioned tensor computation engine has a few limitations. TensorFlow is not optimized for highly efficient allocation of tensor graphs in hybrid environments where scarcity of memory and computation cores might be an issue – such as mobile computing devices that possess dedicated GPU capabilities. Further, as previously mentioned, in our research and development experiments and in our commercial product development process, we discovered the clear need of executing multiple computational tensor graphs in a parallel fashion with partial (or full) data sharing between the various graph evaluation and optimization processes. We also have to mention that, based on our best knowledge and on our researched further presented in this thesis, there are no available tensor graph computation engines on the market that solve the proposed problem.

1.3 Objectives

The main objective of the project research and development was to create a state-of-the-art approach for computational graph resource evaluation and determination of optimal strategy for execution of the mentioned tensor graphs on limited resources GPU infrastructure. This objective has been based on the actual real-life objective of running on-site experiments without having access to large computational infrastructures due to data confidentiality measures. We can further define our objective with the following three sub-objectives as follows:

a) on-the-fly evaluation of computational graphs required resources;
b) allocation and parallel execution with shared or non-shared data;
c) application of these strategies to a real-life production grade problem.

To further detail the challenge and the approach presented in this thesis, a basic use case will be described in detail depicting the real-life problem that has been addressed both directly and indirectly by this work. As previously mentioned, in the presented scenario the proposed system should be able to execute multiple tensor-based graph computations in parallel on a limited resource system such as a mobile computing device. As a result, the system used in this scenario is a usual portable professional computer that offers a basic amount of CPU computation power and an entry level GPU computation device with limited capability of performing parallel numerical computations.

1.4 Solution Overview

As mentioned within the objectives section, the main purpose of the real-life experiment is to allow execution on premises, with no cloud computing support nor local high-performance resources, of multiple experiments in the area of predictive analytics for the industrial vertical of pharma retail.

Based on the research and experimentation presented in this thesis, we argue that the researched and developed solution (HELIX) solves the above presented problems within the real-life use-case. The solution has the following two main components:

- *Directed acyclic graphs resource evaluation sub-system*. This proposed sub-system has the purpose of evaluating a given DAG resource requirements before the tensor computation engine initiates an evaluation or optimization session. This approach gives us a solution for having enough in-process knowledge in order to allow or not the tensor computation engine to further allocate GPU resources. This particular sub-system basically analyses „off-line" multiple scenarios of evaluation or optimization for a given DAG and generates all the potential resource requirements for each scenario;

- *DAG allocation and execution system*. Based on the knowledge gained by applying the DAG resource evaluation heuristics the execution system uses multi-threaded flow scheduling in order to asynchronously fit the maximum number of heterogenous jobs on the available resources. The DAG jobs execution system bases its heuristics on multi-source knowledge such as: DAG requirements for a given task, external input data resource requirements, inter-DAG data sharing, current available physical resources and finally the existing running jobs.

In the related predictive analytics tasks are employed deep learning techniques based on tensor-graph computation. The deep learning models are composed of Directed Acyclic Graphs (DAGs) combining multiple architectures such as Fully-Connected Neural Networks, Convolutional Neural Networks, Recurrent Neural Networks and skip connections for optimized numerical gradient flowing. The research and experimentation details of the deep learning models, including the commercial business related key performance indicators, are not entirely in the scope of this thesis and are presented in a separate paper called *„Deep recommender engine based on efficient product embeddings neural pipeline"* - currently in draft stage available on arXiv [2]. Regardless the architectural details of the proposed predictive analytics experiment, the actual challenges and solutions proposed in this thesis can be analyzed as a totally separated work entirely related to parallel computation optimization in GPU-CPU environments.

Without going into further details of the deep learning DAGs, we have to analyze their actual running environment – namely the TensorFlow computation engine. Nevertheless, for a quick reference of the proposed directed acyclic graphs architectures, we mention that they are using similar techniques and architecture with that of with word2vec [3], doc2vec [4], prod2vec [5], meta-prod2vec [6], GloVe [7] and also sequence-based inference and prediction approaches. Within the presented scenario (an actual real-life example of inferring customer buying propensities), the system has to process on-site a massive amount of data (namely transactional information) in order to extract and present an inferential and predictive analytics report in near-real-time and on-the-spot. The presented study case has an input dataset of well over 200 million transactions summing over 50 GB of data (more than any mobile infrastructure system memory) and the computational graphs have to analyze all transactional information and compute in parallel graph-maps that describe the relationship of each entity (in our case, can be product, customer, receipt, product category etc.) with each and every other entity.

1.5 Results of the Proposed Graph Execution Engine

Based on our research, experimental development and deployment of the commercial products, the following results were achieved:

- A reliable approach of obtaining tensor computational graph resource requirements before the actual execution of the graph evaluation or optimization within the tensor computation engine;
- A reliable methodology of pre-execution analysis and also managing multiple tensor graph execution sessions, all wrapped around the TensorFlow framework and thus providing support and potential benefits for other academic and commercial projects that use this state-of-the-art framework;
- Ability to run in the final product multiple predictive analytics experiments in parallel that otherwise would have been ran sequentially on a limited resource mobile computing environment. We have achieved comparable overall experiment pipeline execution time to that of running the experiment pipeline in a multi-GPU or in a computing cluster environment. Finally, this product-level ability of our research and development allows the users or consultants that employ the system to provide on-the-spot and almost real-time analyses.

1.6 Thesis Structure

This thesis will describe in the next chapter which were the requirements and the motivation for this proposed solution, then will present all the technologies and concepts that were taken into consideration for the development of this solution. Further, the approach, how it was implemented, as well as concrete results based on this approach will be debated.

2 REQUIREMENTS ANALYSIS AND MOTIVATION

The main motivation of the project has been based on an actual real-life experiment implementation and its particular issues: how to execute multiple jobs of actual computational graph evaluation and optimization on a limited resources mobile computing device by maximizing the available resource usage and actually "squeezing" as much computation and memory allocation as possible. This motivation has generated the list of project requirements that had to address both the scientific and the commercial aspects of the project.

The main motivation and the actual requirements specifications of the proposed researched innovation is not derived from potential customer needs nor from potential use-case scenarios but rather from actual existing customers that are using this innovation in a production-grade system. The proposed innovation that is presented within this thesis is part of a large scale predictive analytics system that has multiple separate pipelines ranging from our real-life use case - that of real time preparation of market basket analytics/inferring customers buying propensities, based on tensor graph parallel computation, customer retention predictive analytics models (done by other junior data scientist), recommendation engines, event prediction. All these modules have been researched and developed together with an additional ETL team of two engineering and the overall supervision of a senior data scientist (PhD candidate). It is mandatory to mention that our real-life use-case is the main source for the proposed research and innovation presented in this thesis. Even more, the presented research resulting innovation is totally mandatory for the actually functioning and commercial application of the production grade use-case.

The actual real-life production grade use-case of the predictive analytics systems involves, as previously mentioned, the parallel processing of massive amounts of data using tensor computational graphs. Basically, the real-life goal of the system is to provide in a short amount of time the final user/customer with various analysis of the available commercial transactional information, thus constructing a complete relational graph between any two different entities. In order to have even more in-depth information, we can imagine a pharmaceutical retail company that needs a product similarity analysis, product complementarity matching, products composition in encapsulated products, decomposition of compel products, affection-based micro or meta clustering of all its products - all of these features based entirely on customer transactional behavior without any other features. For this hypothesis we have researched and developed several Deep Learning models (as presented in the research paper [2]) that use NLP-related method in order to generate, with two different approaches, many different products „meta-maps”. As stated, detailed information about our research in the area of Deep Learning for Predictive Business Analytics is beyond the scope of this thesis and can be further analyzed in the mentioned paper, however we will briefly explain the real-life end-customer problem that resulted in the need for the proposed innovation in a step-by-step approach:

1. The end-user of the commercial system is presented with the feature of our system: that of constructing advanced analytics for the so called classic „Market Basket Problem”
2. In order to produce a proof-of-concept in real-time and provide the customer with the confidence that the proposed product delivers promised feature, we make the analysis on-the-spot without any cloud-backed resources, without any process of ETL and without any need for customer data export to our internal infrastructure.
3. For the previous mentioned simple proof-of-concept is used fully confidentially transactional data that contains only TIMESTAMP, TRANSACTION_UNIQUE_ID, PRODUCT_ID and CUSTOMER_ID.
4. By employing the mentioned Deep Learning models based on tensor computational graph architectures, there will be created several latent vector multi-dimensional spaces where the products are defined with l numbers based on 32-bit floats (is not necessary double-precision computation). The generated latent multi-dimensional spaces can then be used to produce the previously mentioned end-user analytics.
5. In order to run the previously defined experiment, is needed an environment that will be capable of executing multiple graph computation with multiple data-sources in parallel in order to minimize the experiment running time. Due to the nature of the required tensor computations, basically large matrix multiplications and additions, it is obviously required to employ a simple setup of massive parallel computing that could be used based on mobile computing infrastructure. Thus, a basic setup of GPU capabilities will be used. At this point two hypotheses can be defined:

a. The execution of the tensor computation should be minimized by using the array of numerical cores
b. The total execution time of the whole experiment should be minimized by "squeezing" all the tensor computational graphs and required input data within the available infrastructure

Finally, at this point we arrived at the conclusion that a special GPU parallel computation allocation engine is required that will augment the existing capabilities of the state-of-the-art tensor computation frameworks (TensorFlow in this particular case) with the proposed memory and computation "squeeze" feature. This basic requirement resulted in the following main analysis objectives that provided the fundament of our proposed innovation:

1. The requirement to analyze existing state-of-the-art with emphasis on our particular real-life problem with regard to memory allocation strategy for tensor computational graphs;
2. Analysis of a potential engine architecture capable of finding a good strategy for parallel execution of tensor computational graphs based on optimal allocation of CPU/GPU resources.

3 RELATED WORK

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 this thesis relies on results from the actual production grade systems and experimental results of our real-life project. Nevertheless, the main focus of this present work is related to the current state-of-the-art research and technologies for GPU-based parallel numeric computing.

We will briefly describe the following subjects that relate to this work. First of all, this chapter will focus on parallel computing and processing of tensor graph, as well as on predictive analytics related work based on the real-life project objectives. The first main topic of discussion includes details about tensor graphs, state-of-the-art in tensor graph computation engines (TensorFlow [1] and Theano [8]), state-of-the-art in numerical parallel computing based on GPU (CUDA [9] and OpenCL [10]), GPU memory load and offload performance considerations and state-of-the-art tensor graph architectures. The second main topic of discussion presents the results of the predictive analytics related work for the industrial vertical of pharma retail.

3.1 Parallel Computing and Processing of Tensor Graphs

3.1.1 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** [8] , respectively, two frameworks used for defining tensor computational graph models and for executing such models on hybrid environments (CPUs and GPUs using **CUDA** [9]).

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.

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

3.1.2.1 TensorFlow

Google developed this framework in order to replace DistBelief [11], 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.

3.1.2.2 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 does not possess 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 Table 1. To further substantiate our related work, we have also analyzed the research done by Bahrampour et al. [12] and presented their research in Table 2.

Table 1 – Our analysis of tensor computation engines

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

Table 2 – Bahrampour et al. [12] research on tensor computation engines

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

3.1.3 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** [9] and **OpenCL** [10]. 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.

$$V_1 \cdot V_2 = \sum_{i=1}^{n} v_{1i} * v_{2i}\,, with\, V_1 = (v_{11} \quad v_{12} \quad \cdots \quad v_{1n})\; and\; V_2 = \begin{pmatrix} v_{21} \\ v_{22} \\ \vdots \\ v_{2n} \end{pmatrix} \qquad (1)$$

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 does 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.

3.1.4 Performance considerations

One important subject that our research and development work relate 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 $TotalGPU_T < TotalCPU_T$ where the variables are defined in below equation:

$$TotalGPU_T = GPUMemory_T + ParallelNumericalComputing_T \tag{2}$$
$$GPUMemory_T = (InputTensorUpload_T + OutputTensorDownload_T)$$
$$TotalCPU_T = \min(ThreadBased_T, MKLBased_T)$$

As argued in one of the referenced researches by Lustig et al. [13] there are multiple factors that cooperate to our variable $GPUMemory_T$ (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 $TGPU_1$ when data physically arrives on the destination GPU and the $TGPU_2$ 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.

3.1.5 State-of-the-art tensor graph architectures

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 simple computational directed acyclic graphs [14], as well as sequence-based directed acyclic graphs [15] are known to be the heart of the most efficient predictive analytics systems, namely recommender systems [3] [4] [7] [16]. The representations of the mentioned graphs can be visualized in Figure 1 and Figure 3.

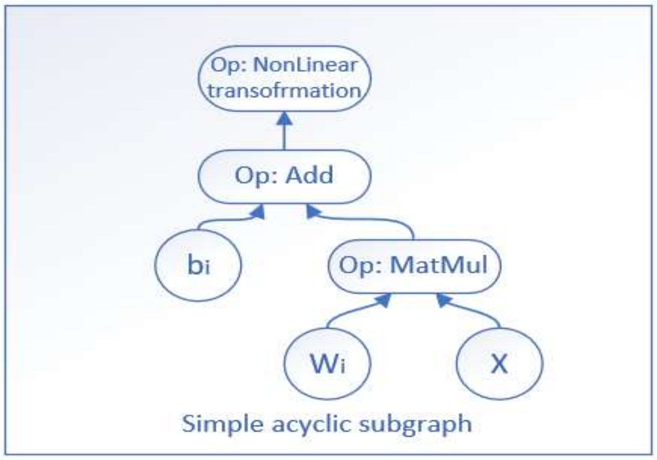


Figure 1 Simple acyclic subgraph

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 evaluation or optimization of a tensorial computation graph is composed of a set of multiple chained operations (nodes) and multiple sets of tensors (parameters). In the particular case of applying the evaluation or optimization of directed acyclic graphs in the area of deep learning, we are either supposed to obtain the result of a function $\hat{y} = f(X|W)$ where X is the representation of the given task (input data) and W are the tensors defined in the graph or to optimize (Equation (3)) a cost function (error function) J that compares the result $\hat{y}$ to a real target y in order to obtain the best parameters that resolve the problem.

$$\underset{W}{\operatorname{argmin}} J(f(X|W)) \tag{3}$$

In order to update the parameters with respect to the computed cost function, in the computation graph is added another node called **optimizer** (Figure 2). The most efficient method of optimizing the parameters is known as Gradient Descent [17]. 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 (4)). Gradient Descent method has been improved over time and therefore momentum-based gradient optimizers [18] have been discovered: Adagrad [19], Adam [20] etc. These momentum-based gradient optimizers update the parameters by adding a fraction of the previous gradients to the current ones (Equation (5)) in order to reach the global minimum of the cost function faster.

$$\begin{aligned} g_{it} &= \alpha * \nabla_{W_i} J(W) \\ W_i &= W_i - g_{it} \end{aligned} \tag{4}$$

$$\begin{aligned} g_{it} &= \beta * g_{it-1} + \alpha * \nabla_{W_i} J(W) \\ W_i &= W_i - g_{it} \end{aligned} \tag{5}$$

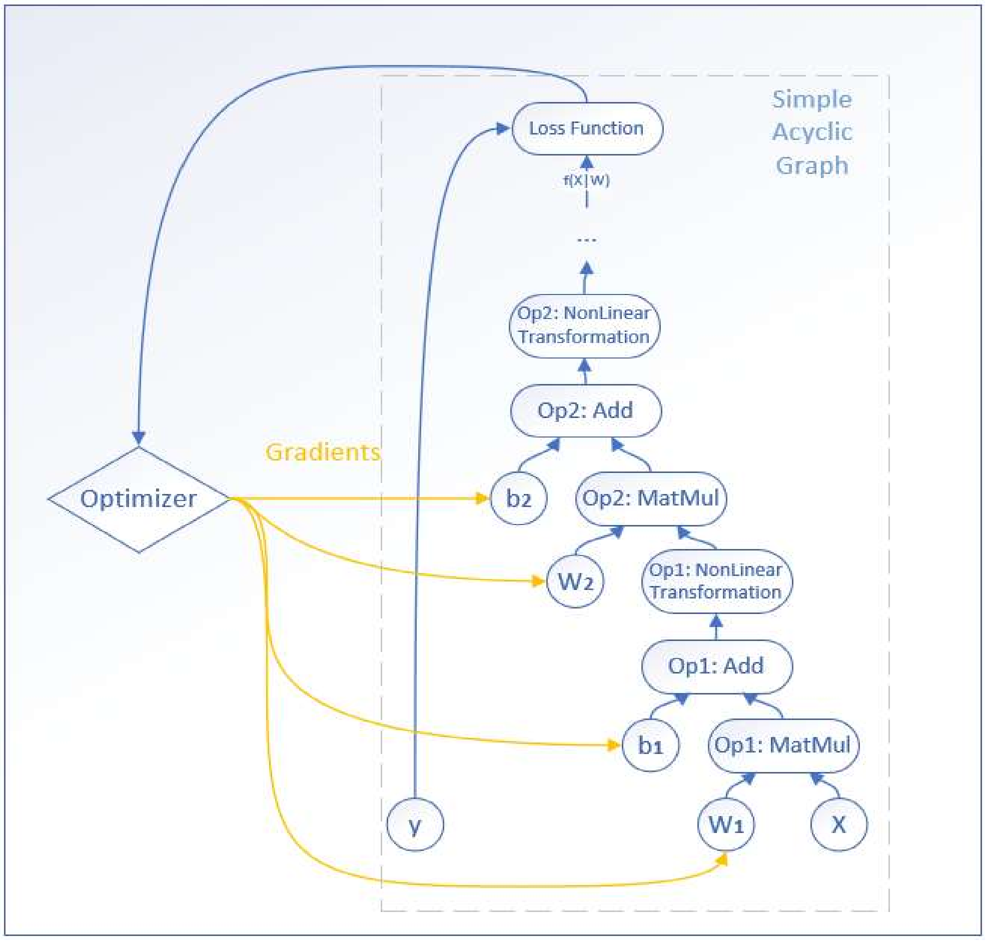


Figure 2 Evaluation and optimization of a simple acyclic graph

A sequence-based tensorial graph can process time series, keeping internally an accumulator h which is a memory off all past time steps. More specifically, if a certain input X_i (that represent a time step in a time series X) 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 $h \in \mathbb{R}^d$ is initialized with a vector of zeros, i.e. the graph does not have any information from previous events;
- For each time step $X_i \in \mathbb{R}^{m\,x\,n}$, the computational graph presented in Figure 3 is evaluated in order to compute the value of the cost function. Moreover, the memory h 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 [21] which introduce a new gradient of loss with respect to the hidden state.

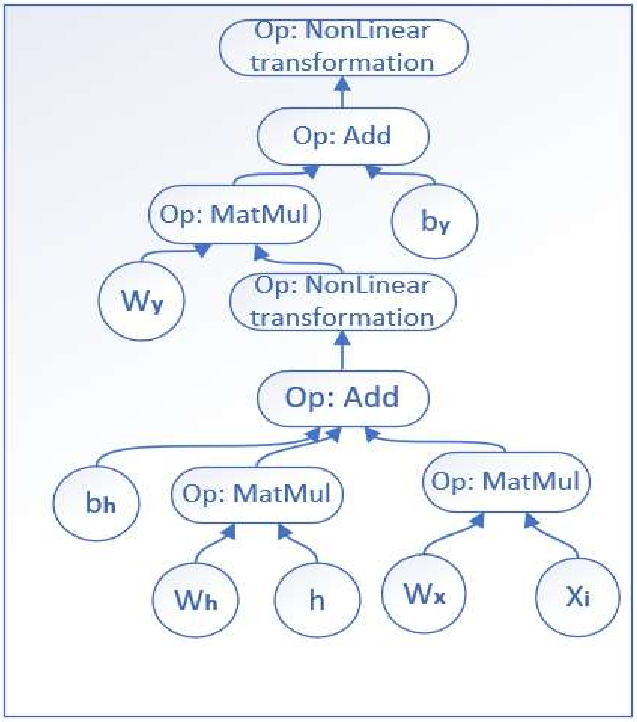


Figure 3 Computational graph of operations for a recurrent block

3.2 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 [2]) 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.

3.2.1 Real life problems

The main need of such a predictive analytics model that can easily recommend market baskets just by computing cosine similarity [22] in the latent semantic 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.

3.2.2 Predictive analytics graph architectures

Our predictive analytics graphs architectures [2] employ, besides the already presented state-of-the-art tensor graph architectures, an additional subgraph which is a representation of embedding concept [3]. This technique maps factor variables, such as products, to a latent space of vectors of real numbers. More specifically, each factor variable i is embedded into a vector $e_i \in \mathbb{R}^l$, which is a precise representation in a semantical multi-dimensional space and which defines the relationship with every different factor variable (product).

As we mentioned, we desire to generate multiple latent spaces of embeddings using the two presented tensor graph architectures for 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) [23]. 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).

3.2.3 Baseline analytics

The predictive analytics architectures fully described in [2], were successfully tested on a pharma-retail dataset of over 200 million transactions and an actual output of the behavioral maps generated using t-SNE algorithm over our multi-dimensional embedding space is presented in Appendix 1. This particular output of the experiment is just one of the multiple expected outputs. In the figure are 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 multiple parallel execution sessions. Our tensorial computation system will evaluate the maximum required GPU resources for each of the 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.

4 HYPERLOOP GRAPH EXECUTION ENGINE FOR LIMITED INFRASTRUCTURE CONTEXTS

Our proposed solution is part of a large scale predictive analytics ecosystem that has multiple separate pipelines. We concentrate our current work to tensor graph parallel computation and integrate this work within the presented ecosystem for real time preparation of market basket analytics.

We discovered the need for such a solution when implementing our case-study using the current state-of-the-art tensor DAG processing framework – TensorFlow – which is not optimized for highly efficient allocation of tensor graph in hybrid environments where scarcity of memory and computation cores might be a problem. Moreover, TensorFlow do not possess capabilities of running multiple computational tensor graphs in a parallel fashion with or without partial (or full) data sharing and therefore, our system, called HELIX, intends to solve efficiently these issues by implementing:

- A memory allocator which calculates the memory required by each graph, decides if the graph can be executed and maps the tensors on the available devices;
- A functionality that starts in a parallel fashion CPU and GPU jobs (each job being in charge of a certain graph), manages the jobs concurrency on the available devices and is capable to share data between the jobs that run at a certain time.

This chapter will describe also the data structures that are useful for data loading and data preparation and will show that our system can handle any type of tensorial computational graph that is fed with any type of structured/unstructured data.

We will briefly describe the solution, specifying its main subjects. First of all, the problem of tensor computation graph partitioning on hybrid systems will be tackled, then the memory requirements for various graphs will be analyzed. The first main component of HELIX - resource evaluation and allocation engine – will be also discussed. Very important to mention that the presented methodology to calculate the memory requirements is taken into consideration for the implementation of this engine. Moreover, the chapter will present how the data is loaded in order to feed the computational graphs and least but not last, the second main component of the HELIX solution – proper hypervisor that can start multiple threads in order to execute in a parallel fashion the graph on hybrid systems.

4.1 Tensor Computation Graph Partitioning on Hybrid Systems

With the recent advanced innovations for GPU acceleration (CUDA, OpenCL), GPU cells offer the advantage of executing intensive computational tasks in a highly parallelized fashion. A GPU architecture uses SIMD (single instruction multiple data) paradigm [24] that assumes quick threads context switch and orientation towards massive data parallel computing. The graphic processing unit has a private memory space, which is often much smaller than the host memory size (2GB vs 16GB, for example). The memory hierarchy of a GPU device is simplified comparatively to that processor core x86/ARM and thus, can result in memory access latency. This drawback can be hidden by highly intensive computation, which is handled very well by a GPU, as against a CPU device.

Our tensorial directed acyclic graphs often require much more memory than the available GPU VRAM and therefore they should be partitioned on GPUs and CPUs. We demonstrate the insight that this partitioning is more effective than processing the graphs only on host.

A directed graph can be mathematically represented as a collection of nodes and directed edges that describe the flow between nodes: $G = (N, E)$, where N is the set of nodes in the graph and E represents the set of directed edges. In order to define the partitioning problem [25], we adopt the following notations based on Figure 4 Partitioning the graph in a hybrid system:

- P_{CPU}, P_{GPU} – how many edges per second can process each device. Research done by Hong et al. [26] shows that $P_{GPU} >> P_{CPU}$, which demonstrates that GPU is fit to massive data parallel computing;
- C_{Speed} – communication speed between host and GPU;
- r – the percentages of the edges that are partitioned on the host;
- q, E_c – the percentage of the edges that cross the created partition and the subset of that edges;
- $1 - r - q$ – the percentage of the edges that are partitioned on GPU;
- $G_p = (N_p, E_p)$ – the representation of a partition on a processing element, where p can be CPU or GPU;
- $card(E)$ – the cardinality of the set E.

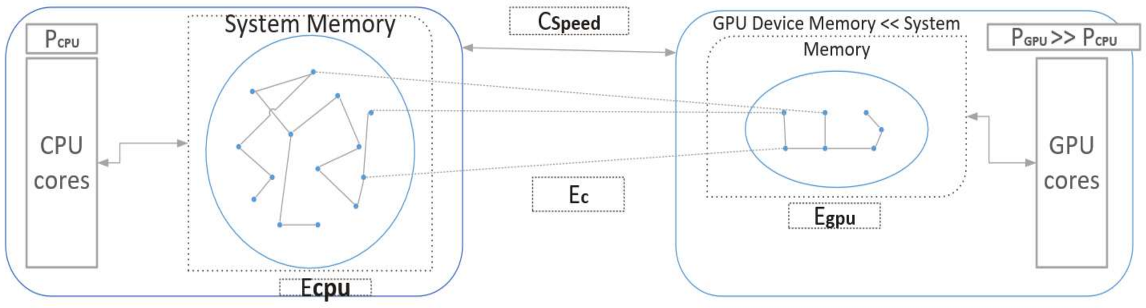


Figure 4 Partitioning the graph in a hybrid system

The time to process a partition of the tensorial graph G on a processing unit p considers the time for processing the edges in that given partition plus the time for communication through the crossing edges (Equation (6)).

$$t(G_p) = \frac{card(E_p)}{P_p} + \frac{card(E_C)}{C_{Speed}} \tag{6}$$

Considering that the performance of a hybrid system is affected by its slowest device (**the host**), the speedup of computing a tensorial graph in a hybrid environment, compared to computing it only on the host, can be defined as follows:

$$Speedup(G) = \frac{\frac{card(E)}{P_{CPU}}}{\frac{card(E_{CPU})}{P_{CPU}} + \frac{card(E_{GPU})}{P_{GPU}} + \frac{card(Ec)}{C_{Speed}}} \Leftrightarrow \tag{7}$$

$$Speedup(G) = \frac{\frac{card(E)}{P_{CPU}}}{\frac{r * card(E)}{P_{CPU}} + \frac{(1 - r - q) * card(E)}{P_{GPU}} + \frac{q * card(E)}{C_{Speed}}} \Leftrightarrow$$

$$Speedup(G) = \frac{P_{GPU} * C_{Spee}}{r * P_{GPU} * C_{Speed} + (1 - r - q) * P_{CPU} * C_{Speed} + q * P_{GPU} * P_{CPU}} \Leftrightarrow$$

$$Speedup(G) = \frac{1}{r + \frac{P_{CPU}}{P_{GPU}} * (1 - r - q) + \frac{P_{CPU}}{C_{Speed}} * q}$$

According to Equation (7), $Speedup(G)$ will depend only on r (the percentage of the edges that are partitioned on the host) if the communication speed $C_{Speed} \to \infty$, because $\frac{P_{CPU}}{P_{GPU}} \to 0$, as $P_{GPU} >> P_{CPU}$. This assumption implies that our insight is validated, because surely a partitioned directed acyclic graph will be computed faster on a hybrid system rather than on host only.

4.2 Graph Memory Requirements

As we already mentioned in subchapter 3.2.2, the predictive analytics graph architectures [2] employ the state-of-the art tensor DAGs, adding extra subgraphs for creating the relational graphs of the entities. Therefore, we analyzed graph memory requirements for the state-of-the-art tensor graph architectures from the perspective of evaluating and optimizing them.

A simple directed acyclic computational graph is represented in Figure 1 Simple acyclic subgraph. Each block in composed of a set of operations (*MatMul, Add* or any nonlinear transformation [27]) that are applied for a set of three tensors ($X \in \mathbb{R}^{m\,x\,n}$ – input, $W_i \in \mathbb{R}^{n\,x\,d}, b_i \in \mathbb{R}^d$ – parameters). If there is an optimization task, for each parameter there exist the corresponding gradients $\frac{\partial J(W,b)}{\partial W_i} \in \mathbb{R}^{n\,x\,d}, \frac{\partial J(W,b)}{\partial b_i} \in \mathbb{R}^d$.

The **total amount of necessary memory** for evaluating/optimizing such a graph using **single-precision floating-point format** calculus**, i.e. 32 bits per parameter**, is presented in the following equation - (8). For the particular case of optimizing the computational graph, the required amount of memory depends on the method used for parameters update [17] [18] [19] [20].

$$Mem(G) = 4\left[m * n + k\left((n+1) * d\right)\right] bytes, \text{where} \quad (8)$$

$$k = \begin{cases} 1, & evaluation\ task \\ 2, & optimization\ task\ using\ basic\ opt. \\ 4, & optimization\ task\ using\ momentum\ opt. \end{cases}$$

The memory consumers for the particular case of sequence-based tensor graph architecture are: $X_i \in \mathbb{R}^{m\,x\,n}$ – input, $W_x \in \mathbb{R}^{n\,x\,d}$, $W_h \in \mathbb{R}^{d\,x\,d}, W_y \in \mathbb{R}^{d\,x\,o}, b_h \in \mathbb{R}^d, b_y \in \mathbb{R}^o$ – parameters, $h \in \mathbb{R}^d$ – memory and the corresponding gradients, if there is an optimization task, $\frac{\partial J(W,b,h)}{\partial W_x} \in \mathbb{R}^{n\,x\,d}, \frac{\partial J(W,b,h)}{\partial W_h} \in \mathbb{R}^{d\,x\,d}, \frac{\partial J(W,b,h)}{\partial W_y} \in \mathbb{R}^{d\,x\,o}, \frac{\partial J(W,b,h)}{\partial b_h} \in \mathbb{R}^d, \frac{\partial J(W,b,h)}{\partial b_y} \in$

$\mathbb{R}^{o}, \frac{\partial J(W,b,h)}{\partial h} \in \mathbb{R}^{d}$. Consequently, **the total amount of necessary memory** for evaluating/optimizing this particular graph using **single-precision** calculus**,** is presented in the following equation - (9). It is again mandatory to mention that if the computational graph needs to be optimized, then the total amount of memory depends also on the optimization method.

$$Mem(G) = 4\left[m * n + k\left((n + d + 1) * d + (d + 1) * o\right)\right] bytes, \text{ where} \quad (9)$$

$$k = \begin{cases} 1, & evaluation\ task \\ 2, & optimization\ task\ using\ basic\ opt. \\ 4, & optimization\ task\ using\ momentum\ opt. \end{cases}$$

With these definitions of the memory requirements, we can state that the predictive analytics graphs will be extended with an additional number of bytes (Equation (10)) caused by the employing of embedding technique [3] for creating the relational graph between all entities.

$$Bytes_{Ext} = 4 * k * v * l \quad (10)$$

In the above equation, k has the meaning as in equations (8) and (9), namely depending on the task our system is in charge of at a certain time: evaluation or optimization; v is the total number of factor variables and l represents the dimension of an embedding.

It is mandatory to mention that the presented methods both for tensorial graph evaluation and for optimization are generally valid for any kind of DAG that can support these operations.

4.3 Resource Evaluation and Allocation Engine

The first main component in this project, called HELIX, is the resource evaluation and allocation engine, which plays a very important role in this system because it manages the resources of the hybrid CPU-GPU environment. This efficient management results from:

- Real-time interrogation of the available host memory and the available GPU devices' VRAM;
- Very precise computation of tensor directed acyclic graph memory requirements;
- Efficient allocation of tensor graphs in the available GPU or CPU memory.

As we already mentioned (subchapter 1.1), this project is built over the state-of-the-art tensor graph computation framework, TensorFlow, which do not possess efficient allocation of tensor graphs in hybrid environments with very limited resources. As a consequence, the presented engine represents a powerful tool also in the related predictive analytics ecosystem, but also in every domain that requires massive parallel processing of DAGs using environment with scarce resources.

The main TensorFlow's drawback that this solution aims to correct to a great extent is that of poor evaluation of memory resources. Every time TensorFlow handles a tensorial computation graph that needs to be allocated and that requests more memory than the actual available GPU resources, the framework will throw `ResourceExhaustedError: OOM (Out of Memory).`

In Table 3 is presented an exact output of this error, when trying to allocate 3 tensors that occupy respectively, 2GB, 1GB and 3GB, on a NVIDIA GeForce GTX 1060 GPU device, with total memory 6.00GB, but only 4.915GB available. From the displayed output, it can be noted that the TensorFlow backend manages to allocate memory for the first 2 tensors, but when it is trying to map the last tensor on GPU, OOM error will be thrown. As a detail, the float tensor with shape `[268435456, 3]` occupies exactly 3GB (Equation (11)).

Table 3 – ResourceExaustedError: OOM example in TensorFlow

```
Found device 0 with properties:
name: GeForce GTX 1060 major: 6 minor: 1 memoryClockRate(GHz): 1.6705
pciBusID: 0000:01:00.0
totalMemory: 6.00GiB freeMemory: 4.915GiB

Allocator (GPU_0_bfc) ran out of memory trying to allocate 3.00GiB.  Current
allocation summary follows
     Summary of in-use Chunks by size:
1 Chunks of size 1073741824 totalling 1.00GiB
1 Chunks of size 2147483648 totalling 2.00GiB
Sum Total of in-use chunks: 3.00GiB

ResourceExhaustedError: OOM when allocating tensor with shape[268435456,3]
and   type   float   on   /job:localhost/replica:0/task:0/device:GPU:0   by
allocator GPU_0_bfc
```

$$\frac{268435456 * 3 * 4\ bytes}{1024^3} = 3GB \tag{11}$$

The presented situation would result in a non-productive system which needs non-proficiency workarounds that would lead to correct evaluation or optimization of the graph using pure TensorFlow. Nevertheless, the proposed memory evaluation and allocation engine that is part of the whole project HELIX resolves this problem in an elegant way which implies three different modules.

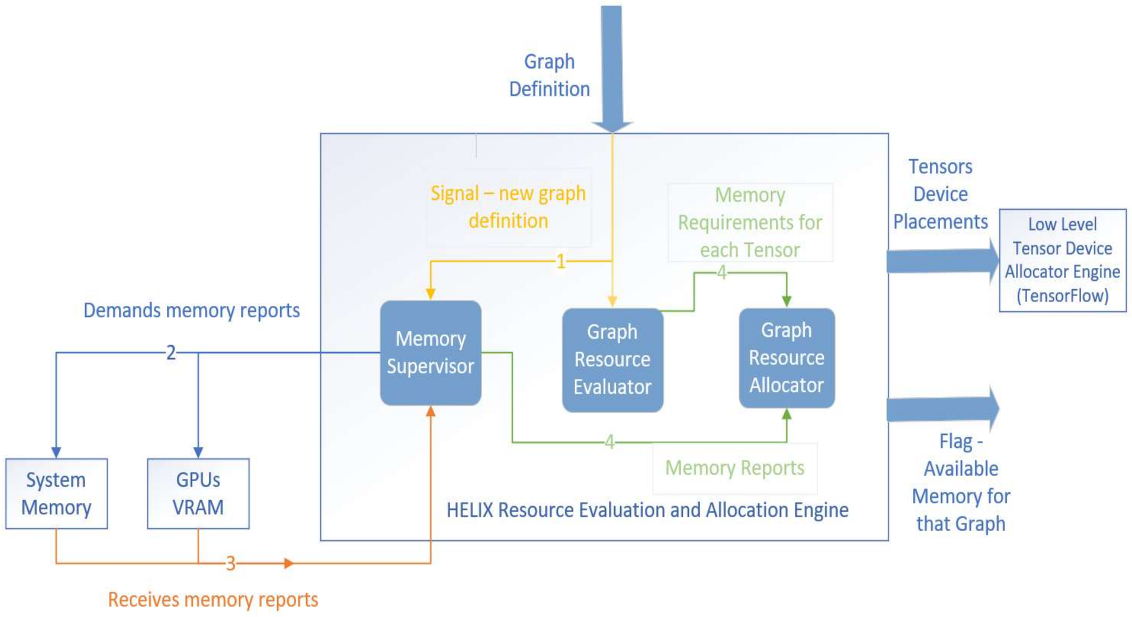

Graph Definition
Signal – new graph definition
Memory Requirements for each Tensor
Tensors Device Placements
Low Level Tensor Device Allocator Engine (TensorFlow)
Demands memory reports
Memory Supervisor
Graph Resource Evaluator
Graph Resource Allocator
1
2
3
4
4
System Memory
GPUs VRAM
Memory Reports
HELIX Resource Evaluation and Allocation Engine
Flag - Available Memory for that Graph
Receives memory reports

Figure 5 Functionality of the Resource Evaluation and Allocation Engine – the numbers on the arrows represent the flow in this engine

The first module (Figure 5) is practically a supervisor which inspects every time a new tensorial graph should be allocated the available host memory and the available GPU memory. This module can also offer precise information about CPU core utilization or statistics about memory consumers that were previously allocated by the engine which will be logged to the end-user whenever it is mandatory.

The second module (Figure 5) evaluates the resources that a directed acyclic graph requires by traversing (Depth-First Search or Breadth-First Search) **its definition** and computing the memory demand for that specific graph using the methods presented in subchapter 4.2 both for graph evaluation and graph optimization. Once determined, this module interrogates the memory supervisor and it responds with a complete real-time report of the hybrid environment memory availability and decides if the tensorial graph can or cannot be fit. The module passes then this information to the last module that completes this engine.

The third module (Figure 5) receives from the previous module the report of memory availability, the list with memory requirements for each tensor in the graph and based on these inputs sets a flag that specifies if the tensorial graph is able to be fit in the current free CPU/GPU memory. Having a precise overview both on the current graph definition and on the current state of the hybrid environment, this module will create effectively the graph, specifying the device where each tensor will run, but without allocating them. More precisely, this is a pseudo-allocation mechanism which allow just to specify where a tensor will run, without loading it on that device. Further, the multi-threading engine that will be presented in the following subchapter will handle the TensorFlow sessions and will allocate the memory, if the internal flag is set. Otherwise, there exists another mechanism that enables to keep the not yet allocated graph in a waiting state until the necessary amount of memory for it will be free.

4.4 Data Loading and Preparation

A tensor DAG can be fed with data either for an optimization task or for an evaluation task if it contains special tensors (inputs) that are not initialized with any value and are just filled with data. In order to build a general architecture that can load and prepare data for every tensorial computation graph, not only for the particular case of predictive analytics graphs, the project proposes a solution which implies that for each DAG there should exist a reference to a data generator that has access to a file handle (where the data resides) and iteratively generates chunks of data from that file (Figure 6). The chunk size is chosen by the end-user and it’s recommended to be small in order to load from disk to RAM only the data is being processed once. It’s obvious that if the graph does not contain input tensors, there will not be any associated data generator.

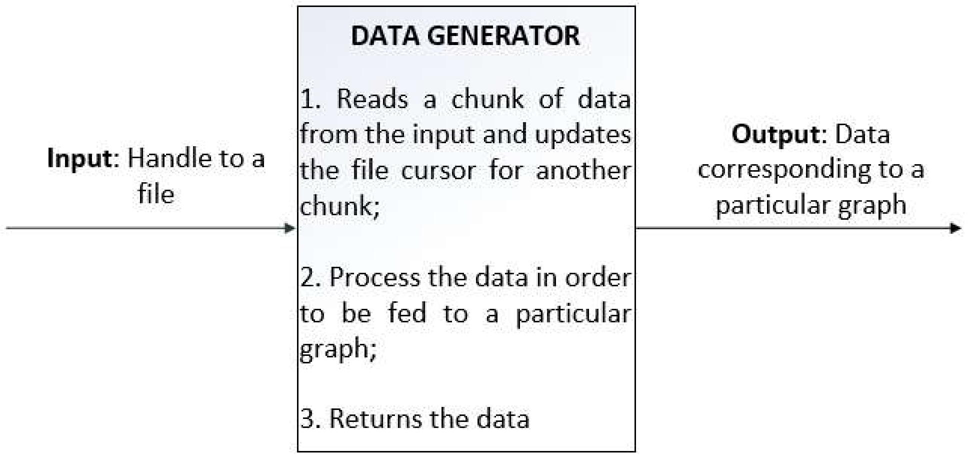

DATA GENERATOR
1. Reads a chunk of data from the input and updates the file cursor for another chunk;
2. Process the data in order to be fed to a particular graph;
3. Returns the data
Input: Handle to a file
Output: Data corresponding to a particular graph

Figure 6 Data Generator inputs/outputs

This way of abstraction leads to an implementation of the multi-threading engine used for parallel processing of the graphs that is agnostic of how the data is loaded and processed. Therefore, the engine can be easily used in any other ecosystem where tensorial graph parallel optimization/evaluation is required.

4.5 Multi-Threaded Execution of the Graphs on Hybrid Systems

TensorFlow obtains very high performance for systems where distributed capabilities exist or for systems with multi-GPU devices. In such systems, each GPU device can be used for memory loads and offloads only for a single tensorial directed acyclic graph and therefore each GPU device is associated to execution of intensive tensorial computations for a single specific task. The simple structures that possess very low GPU capabilities are not suited for executing multiple tensorial graphs in a highly parallelized fashion using TensorFlow. Therefore, the proposed hypervisor targets these simple structures through efficient memory management and parallel execution of the graphs using multi-threading even if there exists one or few GPU cards with low facilities.

Nowadays, most environments, even those with limited resources, offer multi-core computation support and that’s why multi-threading should be the work-horse for every system which intends to resolve a massive computational task using parallel jobs.

The proposed hypervisor that is part of the whole project HELIX is capable to manage multi-threading processing, by assigning a TensorFlow session to a specific thread. Each thread sets up a TensorFlow session with a single computational graph and then allocates effectively either the host memory or GPU memory for that DAG. As already mentioned, every tensor that is allocated on GPU will be loaded, computed and then off-loaded on the GPU card entirely in a transparent way using TensorFlow's backend mechanism and kernel implementations.

Figure 7 describes this project, without all the implementation details, in the sense that it defines the flow between all its main components. The hypervisor receives jobs described by a tuple of three elements: graph definition, the data generator that feeds the graph and the configuration file for that tensorial graph. The configuration file is JSON formatted and it indicates which are the inputs that should be fed with data and which are the operations that are run both for optimization and evaluation.

JSON Formatted Graph Configuration File

```
{
    "INPUT_TENSORS"  : [tensor_input1, tensor_input2 …],
    "EVALUATION"     : {
         "OPERATIONS":  [eval_op1, …]
    },
    "OPTIMIZATION"   : {
         "OPERATIONS": [opt_op],
         "NR_ITERATIONS": 5
    }
}
```

The HELIX hypervisor passes the graph definition to the resource evaluator and allocator which returns a list of tensors device placements and a flag that specifies if there is available memory in the system (either on CPU or GPU). If the flag is not set, then the job will be assigned in a priority queue, which is managed by an asynchronous task which operates regularly and establish when a pending job from the priority queue can be fit in memory. Otherwise, if the flag is set, then the TensorFlow graph is created based on its definition and on the device- placements returned by the resource evaluator and allocator, a thread is initialized and a session is instantiated. The session allocates internally the memory and then the hypervisor runs, using this session, the operations specified in the configuration file, feeding data from the generator. The last process in this pipeline refers to an end-user who should collect the needed information.

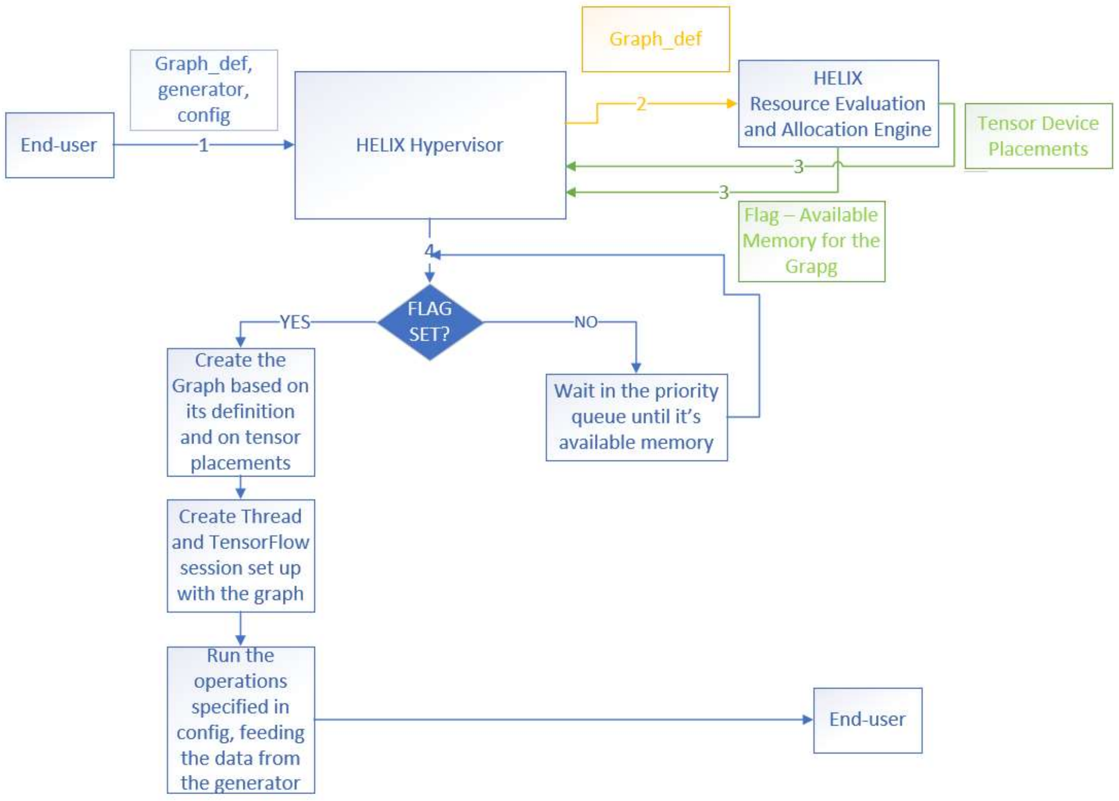

Graph_def
Graph_def, generator, config
End-user
1
HELIX Hypervisor
2
HELIX Resource Evaluation and Allocation Engine
Tensor Device Placements
3
3
Flag – Available Memory for the Grapg
4
FLAG SET?
YES
NO
Create the Graph based on its definition and on tensor placements
Wait in the priority queue until it's available memory
Create Thread and TensorFlow session set up with the graph
Run the operations specified in config, feeding the data from the generator
End-user

Figure 7 Functionality of the proposed project

4.5.1 Jobs initializer

The HELIX hypervisor uses the multi-threading facilities of the working environment and initializes a thread for each specific job, which can be either tensorial graph execution job, or optimization job. The thread will be able to manage a TensorFlow session which is set up with a specific computational graph and to run the specific operations for each tensorial graph, using the environment's resources concurrently with other threads. Besides the proposed project's memory evaluator and allocator, which is a very useful tool for the predictive analytics ecosystem, in particular, but also for every scarce environment that needs parallel processing of DAGs, this facility provides another big improvement by the simple fact that it resolves a necessity in many fields that are based on parallel numerical computing - sharing a GPU card for multiple graphs.

4.5.2 Jobs concurrency

The tensorial DAGs and particularly those used in the related predictive analytics imply massive computations. Therefore, the initialized jobs are CPU-bound and they should be preempted in order to ensure jobs concurrency and high performance of the environment under parallel computing circumstances.

Another aspect that the proposed solution considers is the integration of a professional logging module which is present in each part of the related predictive analytics ecosystem. This logger represents a tool that prints all the logging messages to standard output, as well as in a unique identified file kept in cloud. This approach of work has many advantages ranging from very easy debugging of any issue that could occur to reproducibility of experiments which last very long times. The integration of the logging module is assured through synchronized access of all initialized threads to the same logger object: every time a thread sends a message using the logging module, it will need to acquire a synchronization object that guarantees that no other thread is using the logger at that certain moment.

In the description of the Resource Evaluation and Allocation Engine we mentioned that this solution also takes into consideration the case when a tensorial graph cannot be fit in the current free CPU/GPU memory. For that, each graph that corresponds to this situation is enqueued in a memory-ascending **priority queue**, which is also protected through a synchronization mechanism. It is obvious that the hypervisor should find the timely moment when the available memory can accommodate the first graph in the priority queue. The hypervisor implements this functionality by asynchronously interrogating the available memory. If the available memory is greater than the requirements of the first tensorial graph in the priority queue, then a new thread that handles a TensorFlow session set up with that graph is initialized.

4.5.3 Sharing data between CPU/GPU jobs

The CPU/GPU jobs work exclusively with sessions that are set up with graphs. As mentioned in Data Loading and Preparation subchapter, these graphs can be fed with data. The data can reside in any format on disk, bringing up that the pre-defined data generator should permit working with that file format.

Two or more graphs could require loading data using the same file handle. For this situation of sharing data between the jobs that process these specific graphs, a synchronization mechanism (mutex) is implemented in this project. For each file handle is associated a different mutex object, such that, in the situation two or more threads demand (through their data generators) access to file handles in the same time, the hypervisor triggers the synchronization mechanism just for the jobs that require same file handle.

The last aspect taken into consideration for this specific topic refers to a mechanism that would let the end-user to know the status for each created job any time. It's obvious that the engine cannot be interrogated about the state of the jobs while they are running and thus, the hypervisor (master thread) will create a report file (comma separated values format, for example) whose access will be synchronized through a mutex mechanism. Every time a new job is enqueued in the priority queue, the hypervisor will acquire the mutex and will update the report file, specifying that the job is in a *PENDING* state, and every time an initialized thread will update its state (*RUNNING* or *FINISHED*), will also try to acquire the mutex and update the report file. In Table 4 are presented three possible outputs of the report file at three different moments. For the first moment, graphs managed by job IDs 1,2 and 5 can be fit in memory and run and their threads update the report file accordingly, whereas job IDs 3 and 4 are categorized by the hypervisor as being *PENDING.* Later, job ID 2 finish its evaluation/optimization task, updates the report file, frees the memory, the hypervisor detects available memory for one of the two jobs that are *PENDING* and initialize the thread which also updates the report file, specifying that it is *RUNNING*.

Table 4 – Report file concurrently update

JOB ID	STATE		JOB ID	STATE		JOB ID	STATE
1	RUNNING		**1**	RUNNING		**1**	RUNNING
2	RUNNING		**2**	FINISHED		**2**	FINISHED
3	PENDING	→	**3**	PENDING	→	**3**	PENDING
4	PENDING		**4**	PENDING		**4**	RUNNING
5	RUNNING		**5**	RUNNING		**5**	RUNNING

5 IMPLEMENTATION DETAILS

This chapter emphasizes the implementation details of the proposed solution, how it was successfully applied in the related predictive analytics case study, as well as the working principles of TensorFlow that were applied in this project.

The entire project is implemented using Python language programming [28] and other libraries useful for:

- scientific computing in Python – NumPy [29];
- big-data manipulation and analysis – Pandas [30];
- evaluation and optimization of computations based on dataflow graphs – TensorFlow [1].

5.1 TensorFlow Graphs and Sessions

Before the parallel evaluation/optimization of tensorial computation graphs using this current project, the dependencies between all the operations should be defined. TensorFlow provides an elegant and also efficient programming model for defining such graphs. The following code snippet (**Code 3**) presents a concrete example of creating the simple acyclic graph shown in Figure 1 (replacing the generic "NonLinear transformation" with SELU – scaled exponential linear units [27]), using Python programming interface and NumPy scientific computing library for parameters initialization. Simultaneously, the code snippet illustrates the placement on the particular GPU device *'/gpu:0'.*

Code 3 Graph Definition in TensorFlow using Python programming interface

```
import tensorflow as tf
import numpy as np

# the parameters are chosen from a Gaussian distribution
np_W = np.random.normal(size = (nr_features, nr_hidden))

tf_graph = tf.Graph()
with tf_graph.as_default(), tf.device('/gpu:0'):
    tf_X = tf.placeholder(dtype = tf.float32,shape = (nr_observations, nr_features),
                          name = 'input_X')
    tf_W = tf.Variable(initial_value = np_W, dtype = tf.float32,name = 'param_W')
    tf_b = tf.Variable(initial_value = tf.zeros([nr_hidden]),dtype = tf.float32,
                       name = 'param_b')
    tf_matmul = tf.matmul(tf_X, tf_W)
    tf_add = tf.add(tf_matmul, tf_b)
    tf_nonlinear = tf.nn.selu(tf_add)
    tf_init = tf.global_variables_initializer() # operation that initializes the
variables for this specific computation graph

# write the graph definition that is loaded in the proposed graph evaluator
# and allocator
tf.train.write_graph(graph_or_graph_def = tf_graph, logdir = '.',
                     name = 'simple_acyclic_graph.pb', as_text = False)
```

A TensorFlow Session represents the communication pipe between the programming interface and the C++ runtime that executes all the instructions. This session is set up with a TensorFlow graph and provides an operation (*Run*) which takes a list of outputs that should be processed and a list of input tensors that should be fed with data in order to compute the output tensors.

The proposed HELIX hypervisor is able to run a list of outputs, either for optimization or for evaluation tasks, using the configuration file that is mandatory for each job and to feed data for the input tensors using the generator. **Code** snippets **4** and **5** briefly present the routines that are executed by each thread for both use-cases: evaluation and optimization.

Code 4 Summary of Thread Routine for Evaluation Task

```
tf_session  = tf.Session(graph=tf_graph)

# signals the C++ runtime to allocate memory for the variables on the corresponding
# device (for example: '/gpu:0')
tf_session.run(tf_init)

feed_tensors = [tf_graph.get_tensor_by_name(n) for n in config['INPUT_TENSORS']]
output_tensors = [tf_graph.get_tensor_by_name(n) for n in
                  config['EVALUATION']['OPERATIONS']]
generator_obj = generator()
all_results = []
while True:
    try:
        feed_data = generator_obj.next()
    except StopIteration: # all the chuncks were processed
        break
    feed_dict = dict(zip(feed_tensors, feed_data))
    all_results.append(session.run(output_tensors, feed_dict=feed_dict))
return all_results
```

Code 5 Summary of Thread Routine for Optimization Task

```
tf_session  = tf.Session(graph=tf_graph)

# signals the C++ runtime to allocate memory for the variables on the corresponding
# device (for example: ’/gpu:0’)
tf_session.run(tf_init)

feed_tensors = [tf_graph.get_tensor_by_name(n) for n in config[’INPUT_TENSORS’]]
output_tensors = [tf_graph.get_tensor_by_name(n) for n in
                  config[’OPTIMIZATION’][’OPERATIONS’]]
all_results = dict()
# The optimization is executed iteratively
for iteration in config[’OPTIMIZATION’][’NR_ITERATIONS’]:
    all_results[iteration] = []
    generator_obj = generator()
    while True:
        try:
            feed_data = generator_obj.next()
        except StopIteration: # all the chuncks were processed
            break
        feed_dict = dict(zip(feed_tensors, feed_data))
        all_results[iteration].append(session.run(output_tensors,
                                                  feed_dict=feed_dict))

return all_results
```

5.2 Data Analysis and Generation for the Study Case in Predictive Analytics Area

The particular case analyzed in this thesis comes from the predictive analytics area and it refers to a real-life example of inferring customer buy propensities based on market basket analytics using tensorial graphs computation on massive amount of transactional information. These real commercial applications were released in 2018 for a large pharma retail company and the struggle (treated in this thesis) was to create a parallel tensorial graph numerical computation system that has to process all the graph architectures on resource limited mobile computing environments in a short amount of time.

Besides the development of such a system (named HELIX) composed of a) the hypervisor which initializes threads that runs session in a parallel and synchronized fashion and b) the resource evaluator and allocator engine, the project needed to adopt a general method for data generation. The solution proposed that for each computation graph to exist a generator – provider of data, and the hypervisor to call this generator independent of the structure of the tensorial DAG.

For this particular study case, the transactional information dataset has well over 200 million observations summing over 50GB of data. Loading all the data into the virtual memory of the host is not even possible because, nowadays, the limited structures can rarely reach 16GB RAM. Therefore, a general solution is proposed which use Pandas library that can easily manipulate big-data by reading the files incrementally (chunks). **Code** snippet **6** presents the approach for reading data from the transactional dataset, which is applied for any dataset. The *yield* mechanism in Python results in a generator which does not keep all the values in memory, generating them "on-the-fly".

Code 6 Data Generation for Market Basket Analysis

```
import pandas as pd

# handle to the transactional dataset file.
pd_reader = pd.read_csv('transactions.csv', chunksize = chunksize,
               names = [TIMESTAMP', 'TRAN_UNIQUE_ID', 'PRODUCT_ID', 'CUSTOMER_ID']

def mba_generator(file_handle):
    for pd_chunk in file_handle:
        inputs = process_chunk(pd_chunk)
        yield inputs

# reference that is passed to the HELIX hypervisor along with the graph and
# the configuration JSON for that graph
generator = mba_generator(pd_reader)
```

6 EVALUATION

This chapter will present the results obtained using the researched and developed parallel DAG computation and resource allocation system, namely HELIX, for the particular case of predictive analytics tensor computation graphs.

Using the Deep Learning techniques presented in [2], the resulting graphs generate semantical and behavioral information about the products (Appendix 1) and the users of the big pharma retail company used for the case study. The Deep Learning techniques are meant to discover common behaviors between people, products that can be placed in the same market baskets (recommendations), buying propensities of each customer for any of the existing products, or to predict the next basket that a customer will prefer.

Before the research and development of this project, all the graphs resulted from the particular presented Deep Learning techniques were either run in a sequentially manner on a low resource computing equipment or were parallelly executed in a High-Performance Computing cluster (CPU and/or GPU). Therefore, in a particular system deployment case where, on-the-spot and on-site processing/computing is required, there was no other option than all mentioned computational graph jobs sequentially. With the completion of this work, they can be evaluated and optimized in a parallel fashion with partial or full data sharing, based on two main components: a) the hypervisor which initializes threads that runs session in a parallel and synchronized fashion and b) the resource evaluator and allocator engine.

This research and development leads to the ability of presenting in almost real-time, using local/mobile limited resource infrastructures, the results of the Deep Learning based analytics for any potential client or project. This also mitigates the issue of having highly confidential data (for example, customer confidential transactional data) which cannot be processed with cloud support.

For the real-life commercial pharma-retail scenario, there are $P = 26{,}719$ pharmaceutical products ever marketed and $U = 4{,}344{,}882$ people who have been involved in at least a purchase. Three different graphs which use the embeddings technique [3] [4] [16] are defined for this scenario and they are further presented:

- P2V (Product2Vector): represents all $P = 26{,}719$ products on a multi-dimensional space of l_1 embeddings. The graph is composed of an embedding subgraph and a simple acyclic subgraph which receives an input with l_1 features (Figure 1);
- U2V (User2Vector): represents all $P = 26{,}719$ products and all $U = 4{,}344{,}882$ users on two multi-dimensional spaces of l_1 and l_2 embeddings. The graph is composed of 2 embedding subgraphs and a simple acyclic subgraph which receives an input with $l_1 + l_2$ features;
- NBP (Next Basket Prediction): applies the same principles as above using two multi-dimensional spaces of embeddings, but they are processed using then a sequence-based tensorial graph (Figure 3).

Table 5 presents the HELIX resource evaluator results for each of these graphs, for different values of l_1 and l_2, based on equations (8) (page 28) (9) (page 29) and (10) (page 29). The optimization task is run using a basic optimizer [17] in order to "squeeze" as many graphs as possible in the available memory. The number of observation is chosen experimentally $m = 512$.

Table 5 - HELIX Memory Evaluator results for diverse tensorial graphs

ARCH.	**TASK**	$l_1 = 64$	$l_1 = 128$	$l_1 = 64; l_2 = 32$	$l_1 = 128; l_2 = 32$
P2V	*Evaluation*	13.27 MB	26.44 MB		
	Optimization	26.42 MB	52.63 MB		
U2V	*Evaluation*			546.97 MB	560.14 MB
	Optimization			1093.76 MB	1119.98 MB
NBP	*Evaluation*			590.56 MB	597.34 MB
	Optimization			1180.94 MB	1194.37 MB

Each of these three graph architectures were tested (in diverse scenarios) using this project on three different mobile infrastructures:

1. **Host** – Intel(R) Core(TM) i7-7700HQ CPU @ 2.8GHz (8 CPUs) and 16GB of DDR4 RAM, **GPU Device** – NVIDIA GeForce GTX 1060 (Pascal architecture), 6GB VRAM and 1280 CUDA cores;
2. **Host** – Intel(R) Core(TM) i7-6700HQ CPU @ 2.6GHz (8CPUs) and 16GB of DDR4 RAM, **GPU Device** – NVIDIA GeForce GTX 960M (Maxwell architecture), 4GB VRAM and 640 CUDA cores;
3. **Host** – Intel(R) Core(TM) i7-6820HQ CPU @ 2.7GHz (8CPUs) and 16GB of DDR4 RAM, **GPU Device** – NVIDIA GeForce 940MX (Maxwell architecture); 2GB VRAM and 368 CUDA cores.

The graphs presented in Table 5 were separatelly tested, using the intrinsic methods of TensorFlow, on the mobile infrastructures – presented above – whose GPU capabilities differ in terms of video memory and parallel processors. All the graphs were successfuly run on each GPU infrastructure because it can be noted that none of them do not require more than 1.2GB of memory and the most minimalist structure possesses 2GB of VRAM. The results, in terms of running time, based on a transactional dataset of over 200 millions observations that sum over 50GB, are presented in the below table.

Table 6 – Individual running time for each predictive analytics graph in evaluation/optimization processes (l_1 denotes embedding space for products, l_2 denotes embedding space for users)

GRAPH	**TASK**	**NVIDIA GPU GTX 1060 1280 cores**	**NVIDIA GPU GTX 960M 640 cores**	**NVIDIA GPU 940 MX 368 cores**
P2V $l_1 = 64$	*Evaluation*	38’	40’	43’
	Optimization	1h 25’	1h 31’	1h 54’
P2V $l_1 = 128$	*Evaluation*	41’	42’	46’
	Optimization	1h 29’	1h 32’	2h 01’
U2V $l_1 = 64, l_2 = 32$	*Evaluation*	43’	46’	48’
	Optimization	1h 38’	1h 45’	2h 03’
U2V, $l_1 = 128, l_2 = 32$	*Evaluation*	49’	55’	1h 02’
	Optimization	1h 57’	2h 05’	2h 20’

GRAPH	TASK	NVIDIA GPU GTX 1060 1280 cores	NVIDIA GPU GTX 960M 640 cores	NVIDIA GPU 940 MX 368 cores
NBP $\boldsymbol{l_1 = 64, l_2 = 32}$	*Evaluation*	1h 04’	1h 20’	1h 28’
	Optimization	2h 19’	2h 47’	3h
NBP $\boldsymbol{l_1 = 128, l_2 = 32}$	*Evaluation*	1h 14’	1h 28’	1h 46’
	Optimization	2h 44’	3h 02’	3h 30’

We defined several scenarios of running in a parallel fashion, using HELIX, multiple graphs chosen from those presented in Table 6 in order to measure the speedup of our solution as against the sequencial run of the scenario:

- **Scenario 1:**
  - P2V $l_1 = 64$ (optimization task);
  - P2V $l_1 = 128$ (optimization task);
  - U2V $l_1 = 64$, $l_2 = 32$ (optimization task);
  - U2V $l_1 = 128$, $l_2 = 32$ (optimization task);
  - NBP $l_1 = 64$, $l_2 = 32$ (optimization task);
  - NBP $l_1 = 128$, $l_2 = 32$ (optimization task).
- **Scenario 2:**
  - P2V $l_1 = 64$ (evaluation task);
  - NBP $l_1 = 64$, $l_2 = 32$ (optimization task).
  - NBP $l_1 = 128$, $l_2 = 32$ (optimization task).
- **Scenario 3:**
  - U2V $l_1 = 64$, $l_2 = 32$ (evaluation task);
  - U2V $l_1 = 128$, $l_2 = 32$ (evaluation task).
- **Scenario 4:**
  - P2V $l_1 = 128$ (evaluation task);
  - U2V $l_1 = 128$, $l_2 = 32$ (optimization task);
  - NBP $l_1 = 128$, $l_2 = 32$ (evaluation task).

Based on the results presented in Table 6, the total sequential running time for each scenario can be calculated as follows (Table 7):

Table 7 – Sequential running time for the presented scenarios

	NVIDIA GPU GTX 1060 1280 cores	NVIDIA GPU GTX 960M 640 cores	NVIDIA GPU 940 MX 368 cores
Scenario 1	11h 32’	12h 42’	14h 48’
Scenario 2	5h 41’	6h 29’	7h 13’
Scenario 3	1h 32’	1h 41’	1h 50’
Scenario 4	3h 52’	4h 15’	4h 42’

Based on the memory evaluator results for the tensorial graphs (Table 5) and on the total GPU memory of the presented mobile infrastructures, we can present for each scenario how many complete tensorial graphs can be “squeezed” on the GPU card (Table 8):

Table 8 – Maximum number of graphs that can be allocated in parallel on GPU by HELIX

Scenario	Number of graphs in scenario	Memory required	NVIDIA GPU GTX 1060 6GB VRAM	NVIDIA GPU GTX 960M 4GB VRAM	NVIDIA GPU 940 MX 2GB VRAM
Scenario 1	6	4.58GB	**6**	**5**	**3**
Scenario 2	3	2.38GB	**3**	**3**	**2**
Scenario 3	2	1.1GB	**2**	**2**	**2**
Scenario 4	3	1.73GB	**3**	**3**	**3**

We obtained, running the graphs for each scenario using the proposed system - HELIX, the following running times, with the associated speedups towards the sequencial run (Table 9):

Table 9 – Speedups for parallel running of “squeezed” graphs vs sequential running

	NVIDIA GPU GTX 1060 1280 cores, 6GB	**NVIDIA GPU GTX 960M 640 cores, 4GB**	**NVIDIA GPU 940 MX 368 cores, 2GB**
Scenario 1	3h 21’ (**3.44x**)	4h 47’ (**2.65x**)	12h 35’ (**1.17x**)
Scenario 2	3h 02’ (**1.87x**)	3h 45’ (**1.72x**)	7h (**1.03x**)
Scenario 3	49’ (**1.87x**)	1h (**1.68x**)	1h 20’ (**1.37x**)
Scenario 4	2h 10’ (**1.78x**)	2h 43’ (**1.56x**)	2h 58’ (**1.58x**)

7 CONCLUSIONS AND FURTHER WORK

7.1 Conclusions

Based on the whole research and development, we succeeded in finalizing the project Hyperloop graph Execution engine for Limited Infrastructure conteXts (HELIX) as part of a larger commercial-grade project (Hyperloop) and thus accomplishing the goal of actually running up to six different parallel computing jobs within a quarter time of the traditional approach.

We can conclude that our proposed solution manages to innovate the area of parallel numeric computing, and particularly when working with tensorial directed acyclic graphs, through the following main approaches/metodologies:

- A methodology that precisely evaluate the memory requirements of tensorial DAGs, which can be integrated in any system that uses TensorFlow as tensor computation engine. However, this methodology has a very important role, particularly on limited infrastructures where we need to know from the very beginning how many graphs can be “squeezed” for parallel computation on the available GPU memory. Up until now, all state-of-the-art tensor computation engines, based on their lazy evaluation approach, did not offer to the researcher or the developer the option of having prior knowledge of the required tensor graphs resources. As we previously mentioned in chapter 4.3, there are various issues such as the simple issue of having runtime OOM errors, which can interrupt the execution flow of the experiments.

- Although computing devices and particularly GPUs are improving day after day, the area of mobile device computing and particularly, parallel computing for mobile devices, will continue to have severe limitations and thus, we foresee an important contribution of the HELIX project in this area.

7.2 Further Work

The work on HELIX project will continue and, short-term it will consists mainly on experiments and new graph architecture deployment within the proposed tensor computation and job evaluation & allocation engine. Currently, based on the work presented in the paper [2], we are experimenting several others graph architecutes such as [7].

Finally, we plan to continue our research and development in the area of other small computing devices such as NVIDIA Jetson family (TX2, TK1) and up to smartphone devices where the GPU-based parallel numerical computational resources are even more limited.

8 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] L. Piciu, A. I. Damian and N. Tapus, "Deep recommender engine based on efficient product embeddings neural pipeline," arXiv preprint, Bucharest, 2018.

[3] 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.

[4] 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.

[5] 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.

[6] F. Vasile, E. Smirnova and A. Conneau, "Meta-Prod2Vec: Product Embeddings Using Side-Information for Recommendation," in *proceedings of the 10th ACM Conference on Recommender Systems (RecSys '16). ACM, New York, NY, USA, 225-232. DOI: https://doi.org/10.1145/2959100.2959160* , 2016.

[7] J. Pennington, R. Socher and C. Manning, "Glove: Global Vectors for Word Representation," in *EMNLP. 14. 1532-1543. 10.3115/v1/D14-1162*, 2014.

[8] 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.

[9] J. Ghorpade, J. Parande and M. Kulkarni, "GPGPU PROCESSING IN CUDA ARCHITECTURE," *Advanced Computing: An International Journal (ACIJ),* vol. 3, 2012.

[10] J. Stone, D. Gohara and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science & Engineering,* vol. 12, 2010.

[11] 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.

[12] S. Bahrampour, N. Ramakrishnan, L. Schott and M. Shah, "Comparative Study of Deep Learning Software Frameworks," arXiv preprint arXiv:1511.06435v3, 2016.

[13] 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.

[14] 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.

[15] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Journal Neural Computation,* vol. 9, no. 8, pp. 1735-1780 , 1997.

[16] E. Smirnova and F. Vasile, "Contextual Sequence Modeling for Recommendation with Recurrent Neural Networks," arXiv preprint arXiv:1706.07684, 2017.

[17] 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.

[18] S. Ruder, "An overview of gradient descent optimization algorithms," arXiv preprint arXiv:1609.04747, 2017.

[19] 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.

[20] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," arXiv preprint arXiv:1412.6980, 2017.

[21] P. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE. 78. 1550 - 1560. 10.1109/5.58337,* 1990.

[22] F. Rahutomo, T. Kitasuka and M. Aritsugi, "Semantic Cosine Similarity," *The 7th International Student Conference on Advanced Science and Technology,* 2012.

[23] L. van der Maaten and G. Hinton, "Visualizing Data using t-SNE," *Journal of Machine Learning Research 9,* pp. 2579-2605, 2008.

[24] F. Khorasani, R. Gupta and L. N. Bhuyan, "Scalable SIMD-Efficient Graph Processing on GPUs," *International Conference on Parallel Architecture and Compilation (PACT),* 2015.

[25] A. Gharaibeh, T. Reza, E. Santos-Neto, L. B. Costa, S. Sallinen and M. Ripeanu, "Efficient Large-Scale Graph Processing on Hybrid CPU and GPU Systems," *CoRR, vol. abs/1312.3018,* 2013.

[26] S. Hong, S. K. Kim, T. Oguntebi and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *16th ACM symposium on Principles and practice of parallel programming (PPoPP '11).*, New York, 2011.

[27] P. Ramachandran, B. Zoph and Q. V. Le, "Searching for Activation Functions," *CoRR abs/1710.05941,* 2017.

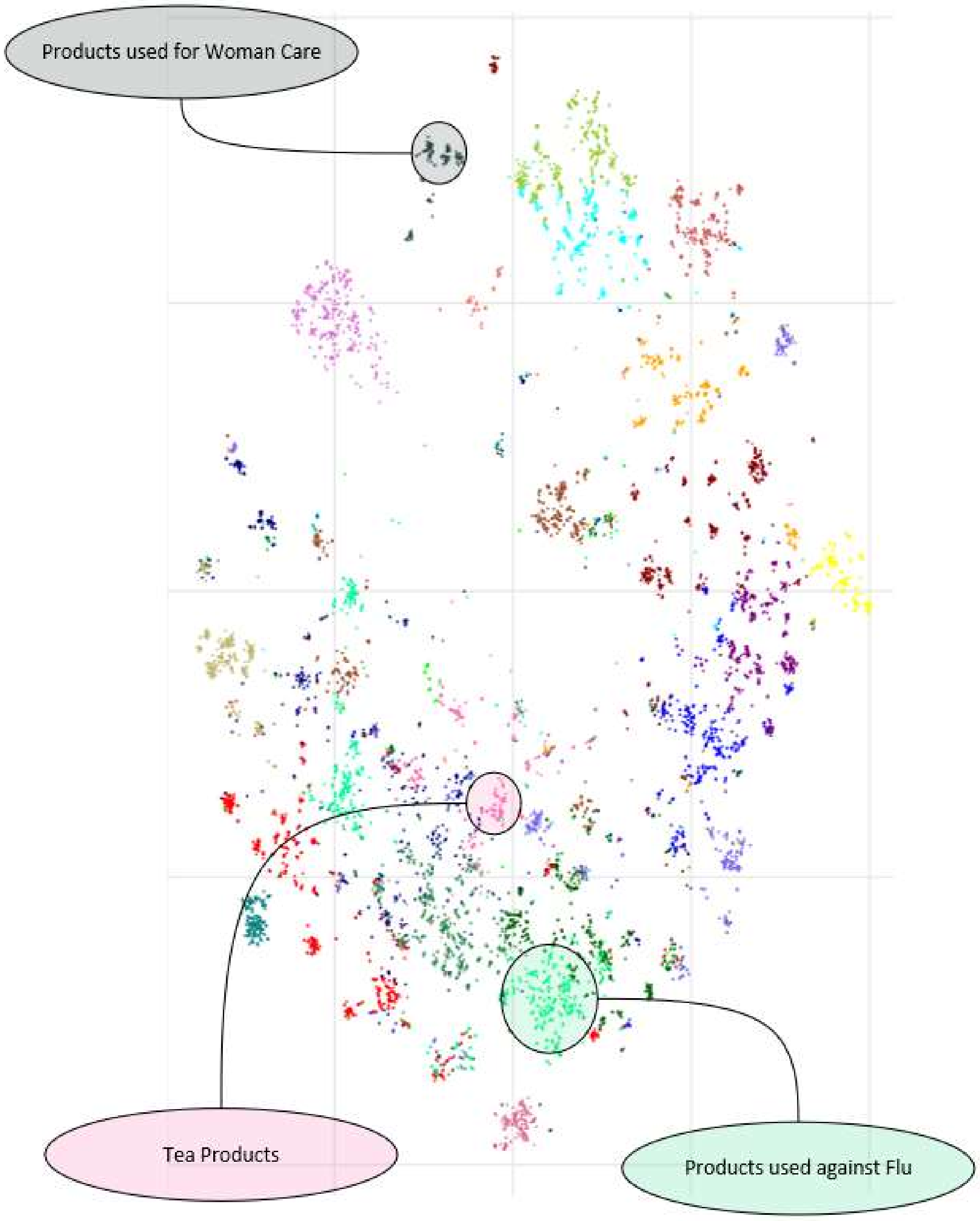
[28] G. van Rossum, "Python tutorial," in *Technical Report CS-R9526, Centrum voor Wiskunde en Informatica (CWI)*, Amsterdam, 1995.

[29] D. Ascher, P. F. Dubois, K. Hinsen, J. Hugunin and T. Oliphant, "Numerical Python," in *Lawrence Livermore National Laboratory*, Livermore, California, USA, 2001.

[30] W. McKinney, "pandas: a foundational Python library for data analysis and statistics," in *Python for High Performance and Scientific Computing, 1-9*, 2011.

9 LIST OF FIGURES

10 APPENDENCIES



Appendix 1 t-SNE representation of products embeddings of a pharma retail company