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 intends to solve efficiently these issues.

We will briefly describe our solution, specifying its main subjects:

1. Tensor Computation Graph Partitioning on Hybrid Systems
2. Data Loading and Preparation
3. Graphs Memory Allocator
4. Parallel Execution of the Graphs on Hybrid Systems
   1. Jobs Initializer
   2. Jobs Concurrency
   3. Sharing Data Between GPU/CPU Jobs

## 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 [1] 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).

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: , where is the set of nodes in the graph and represents the set of directed edges. In order to define the partitioning problem, we adopt the following notations based on Figure 1:

* – how many edges per second can process each device. Research done by Hong et al. [2] shows that , which demonstrates that GPU is fit to massive data parallel computing;
* – communication speed between host and GPU;
* – the percentages of the edges that are partitioned on the host;
* – the percentage of the edges that cross the created partition and the subset of that edges, respectively;
* – the percentage of the edges that are partitioned on GPU;
* – the representation of a partition on a processing element, where can be or .

The time to process a partition of the tensorial graph on a processing unit considers the time for processing the edges in that given partition plus the time for communication through the crossing edges (Equation 1).

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

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:

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

According to Equation 2, will depend only on (the percentage of the edges that are partitioned on the host) if the communication speed , because , as . 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.

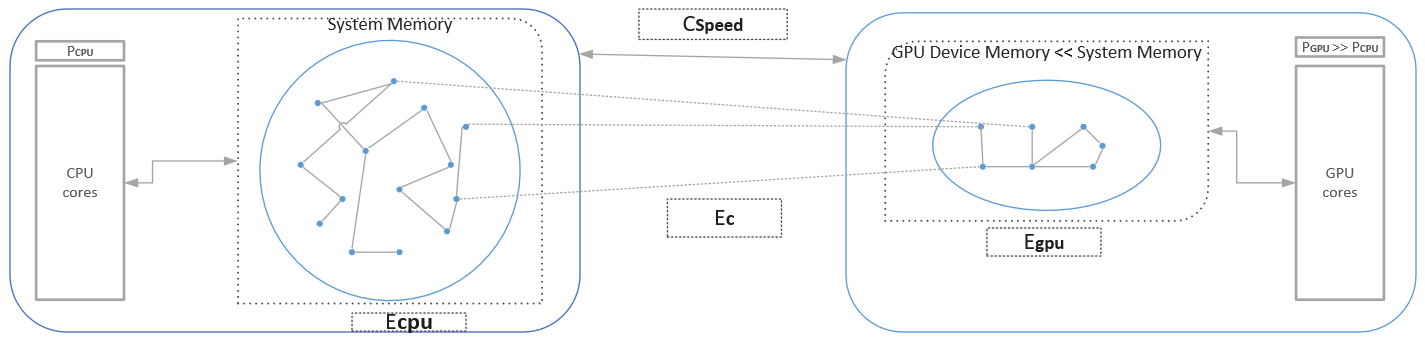
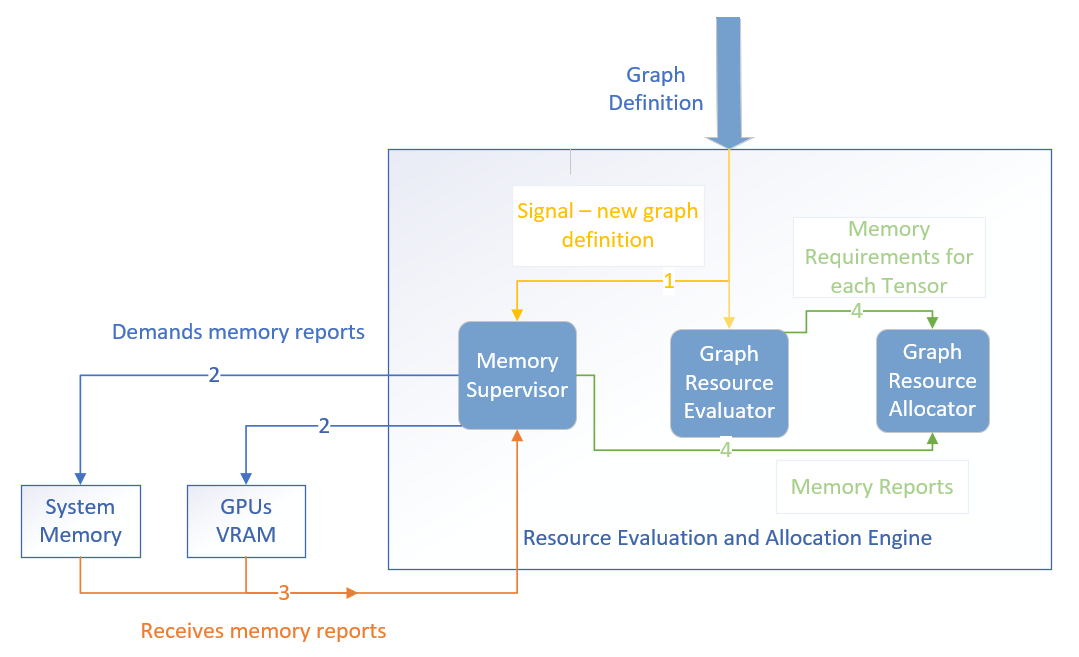


Figure 1 Partitioning the graph in a hybrid system

## Resource Evaluation and Allocation Engine

The first main component in this project 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: (TODO o figura cu astea 3 module)

* 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.



TODO FIGURE

As we already mentioned, 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).

The above 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 resolves this problem in an elegant way which implies three different modules.

The first module 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 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 (TODO subchapter 4.2 Graph Memory Requirements) 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 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 flag received from the second module 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.

## 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. It’s obvious that if the graph does not contain input tensors, there will not be any associated data generator.

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.

## 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 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.

### Jobs Initializer

The 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.

### 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 (TODO resource allocation and evaluation 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.

### Sharing Data between CPU/GPU Jobs

The CPU/GPU jobs work exclusively with sessions that are set up with graphs. As mentioned in (TODO 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 TODO table … 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*.

CAPTION: Tabel

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | JOB ID | STATE | | 1 | RUNNING | | 2 | RUNNING | | 3 | PENDING | | 4 | PENDING | | 5 | RUNNING | |  | |  |  | | --- | --- | | JOB ID | STATE | | 1 | RUNNING | | 2 | FINISHED | | 3 | PENDING | | 4 | PENDING | | 5 | RUNNING | |  | |  |  | | --- | --- | | JOB ID | STATE | | 1 | RUNNING | | 2 | FINISHED | | 3 | PENDING | | 4 | RUNNING | | 5 | RUNNING | |

# Bibliography

|  |  |
| --- | --- |
| [1] | F. Khorasani, R. Gupta and L. N. Bhuyan, "Scalable SIMD-Efficient Graph Processing on GPUs," *International Conference on Parallel Architecture and Compilation (PACT),* 2015. |

https://stackoverflow.com/questions/45419723/python-timer-with-asyncio-coroutine