An Approach for Maximizing Performance on Heterogeneous Clusters of CPU and GPU

Nam-Luc Tran, Sabri Skhiri
EURA NOVA
Email: {namluc.tran, sabri.skhiri}
@euranova.eu

Arnaud Schils
Université Catholique de Louvain
Belgium
Email: arnaud.schils@gmail.com

Edgar Isaac Hiroshi Leon Saiki
Universitat Politècnica de Catalunya
Barcelona, Spain
Email: hiroshi.leon@euranova.eu

***Abstract*—Over the past years there has been significant enthusiasm for development of parallel computing on Graphics Processing Units (GPU) which have now become powerful and affordable hardware equipping data centers and research clusters. Our earlier research has explored the ways to exploit the parallel compute performance of the GPU along the CPU in the same cluster. We have proposed a model for processing distributed machine learning tasks leveraging both the CPU and the GPU equipped on the nodes. Still in this direction, we present in this paper our approach for optimizing the performance of the previously proposed framework. We then further present our approach for integrating this processing model into a more general dataflow graph processing framework by extending it with support for GPU tasks and resources. In addition we have developed a k-nearest neighbors implementation demonstrating all the features. We then present our model based on flow networks for the efficient scheduling on this heterogeneous framework.**

***Keywords*—machine learning, Big Data, GPU, scheduling, parallel processing, dataflow graph, distributed processing.**

I. Introduction

We generate huge volumes of data every day even without realizing it. Whether it be in our social interactions, shopping habits, online activity or daily commute, the current state of technology is able to capture these information and process them in a way that eases our lives. For example we might receive real-time itinerary rerouting based on the traffic measured on social networks or even online shopping product recommendation based on personal tastes.

The same goes in the industry where businesses have been generating data at an even greater scale on a daily basis for years. Companies are now building decision models based on the gathered data, using data mining and machine learning techniques, to help them make the right strategic decision in their business.

The high volume of data has led to two effects. First, new paradigms and models are required to enable efficient processing. Among the most notable innovations we can cite the MapReduce [1] processing model and the dataflow graph (DFG) processing model proposed by Dryad [2]. While MapReduce has initiated a great momentum among companies, more and more are abandoning it in favor of the DFG processing model which is more general [3], [4].

The second observable effect is that even simple models are able to produce significant results when applied on high volumes of data as recent researches have pointed out [5]. High volumes of data allow to express models in a simple way.

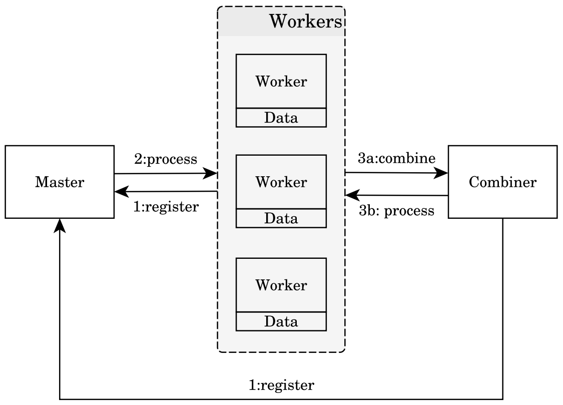


Fig. 1. Architecture of the original framework. Each worker node is equipped with a compute-capable GPU. Steps *3a* and *3b* can be repeated for an iterative algorithm.

The challenge still remains in the processing of these high volumes of data, where hardware compute performance also plays a great part. These past years, High Performance Computing has benefited from developments in parallel computing architecture. We can cite the Graphics Processing Units and the design of many-core architectures as the Xeon Phi or Tilera.

In this way our previous research [6] has focused on two objectives: first, quantifying the added value of, given an existing processing cluster, equipping the nodes with GPUs; and second, defining a suitable processing model for exploiting parallel processing with the GPUs on the nodes along with the CPU. The framework had shown great potential for data intensive applications that require rapid recomputations of decision models for example to take into account fresh data or due to changes in the context [7].

Many challenges were however identified. Designing algorithms on the framework still required ad-hoc development. From the point of view of the delivered performance it did not overwhelmingly outperform other famous frameworks such as Mahout, a data mining and machine learning framework that uses Hadoop MapReduce as backend for running the jobs.[1] Finally, while the processing model is close at some points, it did not offer the full expressiveness of a dataflow graph.

[1]Apache Mahout - https://mahout.apache.org/

```
1: $V \leftarrow dataPoints$                                   ▷ V={5,6,9,10,2,4,7,8,12}
2: init $nbrDimensions$                                        ▷ nbrDimensions=3
3: init $nbrDataPoints$                                        ▷ nbrDataPoints=3
4: $C \leftarrow centroid$                                     ▷ C={2,4,6}
5: $C' \leftarrow$ repeated_range($C$, $nbrDataPoints$)        ▷ range=
   {2,4,6,2,4,6,2,4,6}
6: $V \leftarrow$ subtract_and_square($V$, $C'$) )             ▷ V={9,4,9,64,4,4,25,16,36}
7: $D \leftarrow$ reduce_by_key(linear_index_to_row_index($nbrDimensions$),
   $V$)   ▷ linear_index_to_row_index(3)={1,1,1,2,2,2,3,3,3}, K={1,2,3},
   D={22,72,77}
```

Fig. 2. GPU k-means distance computation strategy.

A solution mentioned then was to extend a DFG processing framework with unified CPU/GPU resources and computations. This also raised further challenges concerning the scheduling of such heterogeneous clusters.

We have pursued the research in this way and the contributions in this paper are as follows. We:

1) optimize further the algorithms in the framework and compare the performance difference with Mahout (§II);
2) present the extension of the DFG processing model to take into account tasks exploiting both batch and streaming processing features of the GPU (§III.A);
3) integrate these operators in an existing DFG processing framework (§III.B-C-D) and demonstrate the features and performance of the operators through a machine learning job (§III.E); and
4) introduce a model for the scheduling on such heterogeneous distributed processing framework (§IV).

The rest of the paper is organized as follows: Sections V describes the future work; Section VI treats the related works; and Section VII concludes the paper.

II. A Framework for Distributed Machine Learning with GPUs

In our previous paper [6] we presented a framework for processing machine learning tasks on a distributed computing cluster where a GPU equips each of the nodes. Figure 1 summarizes the architecture of the original framework.

In the framework each worker node hosts a partition of the total data stored on a distributed filesystem such as the Hadoop Distributed FileSystem (HDFS), the storage layer of the Hadoop ecosystem.[2] Using a distributed filesystem such as HDFS enables data-locality on the workers for the processing. The Master first initiates the job. Each Worker node then performs the processing on its partition of data and produces an intermediate model. These partial models are finally merged by the Combiner to form the final model. The Master can then be triggered to launch subsequent iterations if required. We have shown that this processing model is suitable for a whole family of machine learning and data mining algorithms, referred as *summation forms* [8].

Our contribution in this section is a further optimization that we have developed for the GPU implementation of the k-means algorithm already present in the framework.

[2]Apache Hadoop - http://hadoop.apache.org/

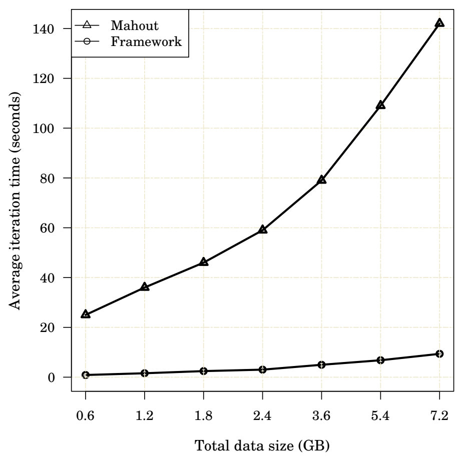


Fig. 3. Average time spent for the computation of the model on a GPU-equipped node and on a CPU node for different sizes of the dataset. Each of the three workers in the cluster locally processes one third of the total data size. GPU: Nvidia GTX Titan. CPU: Intel Core i5-3470@3.2 GHz

A. Optimization of the GPU k-means implementation

The problem of the k-means clustering algorithm can be summarized as, given n data points, finding k cluster centroids such that the average squared distance (or Mean Squared Error - MSE) between the data points and their nearest centroid is minimized:

$$min(\frac{1}{n}\sum_{i=1}^{n}(\min_{k} d^2(x_i, c_k)))$$

where $d(x_i, c_k)$ is the distance (e.g. Euclidean) between the data point x_i and the centroid c_k. In practice, the distance from each data point to every centroid is computed. Each data point is then assigned to the cluster corresponding to the closest centroid. The coordinates of the centroids are then updated as the mean of all the data points belonging to the cluster.

We have further optimized the distance computation phase of the GPU k-means algorithm implementation. All the data points are stored in a one-dimension vector V. For each cluster centroid, the coordinates are subtracted in parallel for each data point and the result squared in V. We then generate an iterator K containing the indices of the data points corresponding to the vector V. The distances for each data point are finally computed using a reduction on the vector V using K as keys to group the data points. Figure 2 illustrates this implementation.

B. Test results discussion

We have tested the performance of the newly optimized k-means clustering implementation along with the performance of a Mahout cluster. This is feasible as both frameworks implement the plain k-means technique with no further algorithmic optimization.

Figure 3 summarizes these results on a dataset with forty features. We can observe that our optimized implementation of k-means using GPUs outperforms Mahout by up to ten times on the same dataset sizes. This is made possible for two reasons: first the $subtract\_and\_square$ and $reduce\_by\_key$ operations are realized in parallel and at once for all data points.

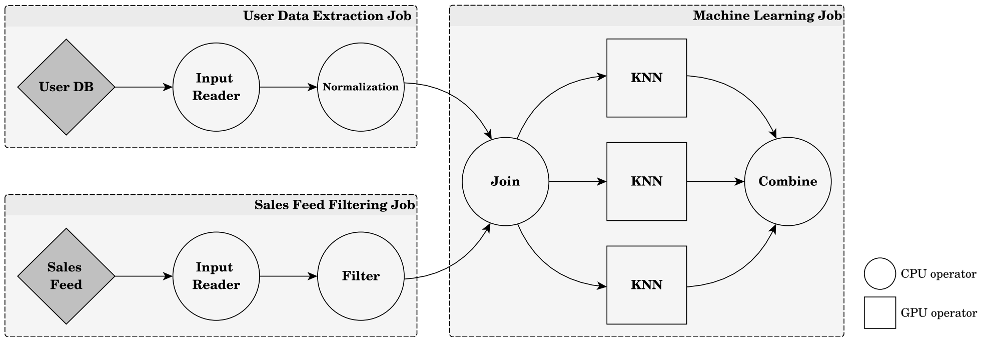


Fig. 4. Dataflow graph representing a global job composed of three chained jobs. The depicted machine learning job fits with the processing model of the framework proposed previously.

Secondly, $repeated\_range$ and $linear\_index\_to\_row\_index$ are iterators generated on the fly. The content of these structures is not allocated in memory as the values are computed on access.

The tests were performed on a cluster of three nodes equipped with i5-3470@3.2 GHz 16GB, two equipped with an Nvidia GTX Titan 6GB and one with an Nvidia Tesla K20 5GB using level 3 `nvcc` optimizations (flag *-O3*)

III. Conciliating the dataflow graph processing model with GPU workloads

The processing model described in the previous section is well similar to a dataflow graph (DFG) processing model. A dataflow graph is an acyclic graph where each node represents an operation on data and the edges represent data dependencies between operations. Figure 4 illustrates a DFG job.

We can actually model a job in the framework as a complete DFG: each worker processes a parallel path of a job graph and the logic of the Combiner is an operator. Each worker further processes a graph of operators, with for instance operators for reading, parsing the data, processing it on the GPU and aggregating the output. The machine learning job described on Figure 4 illustrates this case.

Furthermore, the DFG model easily allows the integration of tasks such as preparation, normalization, aggregation and chaining of other jobs within the same job. Each edge thus models a data channel between operators.

This observation has driven us to extend the dataflow graph processing model to take into account the presence of GPU resources and operators that contain computations targeting the GPU. The presented contribution in this section is, first, the engineering of GPU operators exploiting the batch and streaming processing features of the GPU within DFG jobs; and second, the extension of AROM [9], our in-house dataflow graph processing framework, to integrate these operators.

A. The GPU-enabled DFG operator

The first step in integrating GPU computations in a job graph is the definition of a new type of operator that abstracts and encapsulates computations performed on the GPU. The resulting GPU operator, as any other operator in the DFG model, is then transparently composed with other downstream and upstream operators in a job graph.

The internals of the GPU operator are depicted on Figure 5. It is composed of the host code that prepares and sends the data to the GPU process, and the GPU process that receives the data, processes it, and returns the results to the host code. Figure 5 depicts the sequence of interactions of the GPU operator.

Under the hood, the host code and the GPU process communicate via Inter-Process Communication (IPC).[3] The GPU process receives and returns the data over an IPC socket. This architecture further allows the reuse of data already present in the GPU memory instead of having to retransfer it between two consecutive GPU operators.

We have defined two types of GPU operators: the *batch* and *streaming* operators.

a) Batch GPU operator: this operator waits for all data to be received from upstream operators before transferring the batch to the GPU process. Similarly, the results of the GPU algorithm are transferred at once to the host code.

b) Streaming GPU operator: this operator packs and sends data entry by entry to the GPU process as data is received from upstream. The GPU process then concurrently transfers the data to the GPU global device memory. Similarly, the GPU process can also incrementally return the computed results to the host code. This takes advantage of the CUDA asynchronous memory transfer feature.[4]

During its transfer the data never waits in an intermediate buffer between the upstream operator and the global GPU memory. This global streaming strategy minimizes the impact of the PCI-Express bus. In the best case the GPU kernel execution and the data transfer from upstream operators to the GPU global memory are carried out concurrently.

In order to showcase the capabilities offered by our new GPU operators and their benefits in a distributed processing cluster we have designed a DFG job implementing a k-nearest neighbors (K-NN) machine learning job that exploits all the features described earlier.

B. The k-nearest neighbors algorithm

The k-nearest neighbors is a fundamental machine learning algorithm used for classification or regression [10]. It takes as

[3]through a ZeroMQ (http://zeromq.org/) IPC socket

[4]Compute Unified Device Architecture - https://developer.nvidia.com/cuda-zone

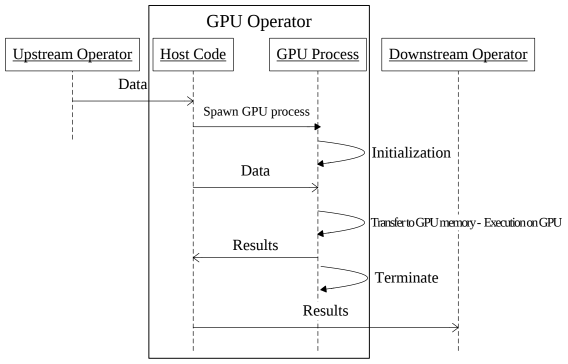


Fig. 5. Sequence diagram of the GPU operator.

input the parameter k and two datasets: one training set and one testing set. The training set contains samples of dimension w already classified. Each training sample is a vector $s_{train} = (x_0, ..., x_{w-2}, t)$ in which t is the target class or variable. The testing set, of dimension $w-1$, contains unlabeled samples $s_{test} = (x_0, ..., x_{w-2})$ for which we want to predict the target variable $t*$.

When receiving a testing sample s_{test} to label, the distances from s_{test} to all s_{train} from the training set are computed and then used to identify the k closest s_{train} neighbors. The distance metric is the Minkowski distance function:

$$d(x, y) = (\sum_{i=0}^{w-2} |x_i - y_i|^p)^{\frac{1}{p}}$$

With $p = 2$ the distance is Euclidean. Since only the relative distances between samples matter, the square root can be removed without altering the outcome of the algorithm. The target $t*$ for s_{test} is then determined by a majority vote on t among its k nearest neighbors in the case of classification. For regression $t* = \frac{\sum_{i=1}^{k} t_i}{k}$.

Because all K-NN computations are delayed to the testing phase, labeling new samples is a computationally intensive task. This phase can be accelerated by the GPU parallel computational power.

C. Distributing the k-nearest neighbors algorithm

By its brute force nature, the original K-NN algorithm described above lends itself easily to distribution. While K-NN distribution is a problem that has already benefited from thorough research we chose to follow a distribution of the brute force strategy since our goal is to illustrate the features of our GPU operator. We position our work with regard to the research performed on the optimization of the algorithm in the Related Work section.

The strategy is based on the distribution of the testing set among the K-NN operators. The whole training set is sent to all the K-NN operators and the testing set is partitioned among the operators. Figure 4 depicts the corresponding machine learning graph job. Each K-NN operator therefore computes the distances from all the instances in the training set to all the testing instances of its data chunk. The majority vote is then carried out by the operator. For each testing sample the operator outputs a $(sampleID, t*)$ tuple containing the ID of the sample and its computed target $t*$. While remaining simple, this strategy is well suited when high volumes of testing samples need to be classified at once.

```
1: init $nbrOfTest, nbrOfTrain, nbrOfEl$   ▷ nbrOfTest=3, nbrOfTrain=3, nbrOfEl=6
2: init $euclideanDist$   ▷ euclideanDist={3,2.5,1,7,2,4.5}
3: $idx \leftarrow sequence(0, nbrOfEl)$   ▷ idx={0,1,2,3,4,5}
4: sort_by_key(euclideanDist, idx)   ▷ euclideanDist={1,2,2.5,3,4.5,7}, idx={2,4,1,0,5,3}
5: $trainIdx \leftarrow$ transform(idx, modulo_by_const_functor(nbrOfTrain))   ▷ trainIdx={2,1,1,0,2,0}
6: $idx \leftarrow$ transform(idx, div_by_const_functor(nbrOfTrain))   ▷ idx={0,1,0,0,1,1}
7: stable_sort_by_key($idx$, $trainIdx$)   ▷ idx={0,0,0,1,1,1}, trainIdx={2,1,0,1,2,0}
```

Fig. 6. GPU K-NN sorting strategy.

D. GPU implementation of the K-NN operator

The GPU implementation of the K-NN operator realizes the following phases:

1) Compute the Euclidean distances: the distance between each testing sample and each training sample is computed and locally stored in a distance matrix $M_{ij} = d(test_i, train_j)$. Let w be the number of attributes in each sample, m the number of training samples and n the number of testing samples, the training and testing matrices dimensions are $m \times w$ and $n \times w$, respectively. Hence, the dimensions of the distance matrix are $n \times m$.

This phase efficiently exploits the parallel computational power of the GPU: threads computes in parallel the square of the Euclidean distance between one testing and one training sample, which corresponds to one element in the distance matrix M. Consequently each thread block of dimension $T \times T$ computes a subset of size $T \times T$ from the distance matrix. Therefore, the total number of thread blocks required for the operation is $\frac{n \times m}{T^2}$. This entire phase is implemented as a pure CUDA kernel.

2) Sort the distances: each row of M is then sorted by increasing distance. First, a global identifier is assigned to each element of the $M_{n \times m}$ distance matrix: $globalId(i, j) = i \times m + j$. The distances are then used as keys to sort the global identifiers by increasing order of distance. The testing identifiers are afterwards retrieved by simply dividing each global identifier by the total number of training samples and the training identifiers are retrieved by taking the modulus after the division of each global identifier by the number of training samples.

Finally, a stable sort is used with the testing indices as the keys and the previously sorted training indices as values. This second sort groups all the distances for a given testing sample in the array and in increasing testing identifier order. The initial order of the training indices, sorted by increasing distance, is preserved thanks to the stability property. Figure 6 illustrates the implementation of this phase.

Since the elements in the device vectors are primitive types, we use the radix sort for both sorts. The radix sort is so far

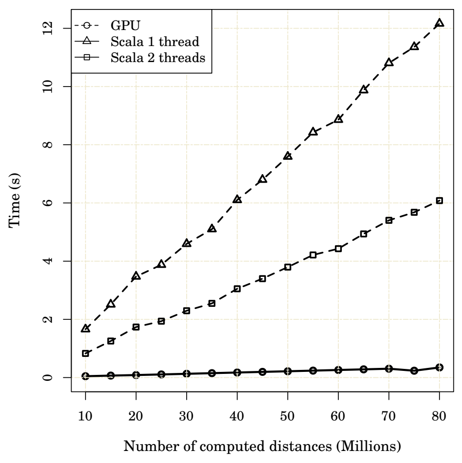


Fig. 7. Comparison of the performance of the distance computation kernel on GPU and Scala implementations. Each vector is of length 40. GPU: Nvidia Geforce GTX 750, CPU: Intel i7-2620M @2.70 GHz.

the fastest sorting algorithm on GPU [11]. We have used the implementation provided by the Thrust library.[5]

3) Majority voting: for each testing sample $test_i$ the k nearest neighbors of the sorted i^{th} row of M are used for the majority voting and to determine $t*$. This phase is performed on the CPU due to the conditional branches.

E. Batch and Streaming GPU K-NN operators

Building on what we have described earlier, we implement a batch and a streaming version of the GPU K-NN operator.

1) Batch GPU K-NN operator: in this implementation the whole data is sent as a single batch to the GPU process. The distance computation kernel is then called on the entire training and testing matrices and the sorting phase is applied. The k nearest neighbors for each testing sample are finally sent back to host process where the majority voting is performed to determine $t*$.

Using this operator, the major issue is the limited amount of memory available on the GPU. Since the datasets are processed at once, the training set, the testing set and the distances matrix must all three fit in global device memory during the distance computation phase. This can however be alleviated by spawning multiple K-NN batch operators in the job graph.

2) Streaming GPU K-NN operator: the streaming GPU K-NN operator addresses the problems of the batch GPU K-NN operator. It is a great example of implementation that fully takes advantage of the streaming features introduced in this work.

a) Incremental and out-of-core computation: we have adapted the K-NN algorithm to enable incremental computation, a requirement for computing the algorithm in a streaming setup. The training and testing sets are split into several chunks. The distance computation is triggered when at least one chunk of each type has been received by the GPU. The resulting distance matrix M is thus also split and computed in chunks.

[5]Thrust - http://thrust.github.io/

This also enables the processing of data sets larger than the GPU memory, a property often referred to as *out-of-core processing* in the literature. We have developed a data serving layer for that purpose that pulls the chunks to the GPU as they arrive and interleaves them for faster processing. This layer also performs the bookkeeping and the memory management of the chunks present in RAM.

We have also developed a similar mechanism for the distance matrix M, computed as a rectangle of chunks and for which out-of-core sorting is performed in passes.

b) CUDA streaming feature: The streaming feature of CUDA enables the overlap of data transfers to the GPU memory with kernel execution. In our case, the transfer to GPU memory, the distance computation and the transfer of the results to the host code are concurrently executed in different streams.

3) Test results discussion: in order to evaluate our approach at each of the steps we have carried out a series of tests on a generated dataset including forty features.

a) Distance computation kernel: Figure 7 shows the performance of our GPU distance computation kernel against optimized CPU implementations of the same task. In order to provide a fair comparison for this test we used CPU and GPU of similar market price (around 150$).

The test highlights a gain of up to 18 and 34 times for the single and dual-threaded version written in Scala, the programming language used by AROM. This is for the same reason as in the k-means algorithm treated earlier: as long as the data fits the GPU memory, all computations in a thread block are performed in parallel and thus performance is only slightly affected by the number of distances to compute compared to the CPU implementation.

b) K-NN operator: Figure 8(a) and 8(b) show the performance of the GPU batch and streaming operators versus an optimized version implemented on the CPU. The CPU implementation computes the target class $t*$ for each testing sample as it arrives.

Compared to the CPU implementation, the streaming GPU operator delivers up to twenty times more performance as, first, computation is divided into chunks of data computed in parallel and, secondly, chunks are continually fed to the GPU. We can interpret the situation like this: the CPU operator processes a stream of data entries while the streaming GPU operator processes a stream of chunks.

The batch operator delivers less performance as it needs to wait for all data to arrive before starting the computation. The streaming GPU operator, while being able to handle larger datasets than the batch operator thanks to the incremental implementation, also yields up to ten times better performance.

4) Implementation parameters discussion: we also conducted a series of tests in order to determine the optimal values for the thread block size, the chunk size and the number of CUDA stream in our implementation.

a) Thread block size: thorough testing and study to determine T, the optimal size of the thread block, have led to 8 as optimal number using level 3 `nvcc` optimizations (flag *-O3*). Increasing further the thread block size does not improve performance.

b) Chunk size: the optimal chunk size is a trade-off in GPU parallelism versus fast streaming of early results. While a bigger chunk size allows filling up the GPU memory and exploiting massive parallelism, smaller chunk sizes allow more fine-grained control over data transfer and processing overlap. In our experiments and with the size of the samples we have measured the best trade-off for the chunk size at around 1200 elements.

c) Number of CUDA streams: further experiments have determined that the optimal number of CUDA streams is two, with a twenty percent increase in performance. Any higher number does not yield better performance.

IV. Scheduling on heterogeneous processing frameworks

The integration of GPU operators performing computations on the GPU in a processing framework such as described in the previous section introduces heterogeneity in the processing. All the resources in the cluster are not equivalent anymore and some tasks of a job now have preference over a certain type of resource. This opens the way to new opportunities for the scheduling of operators on the resources.

We also consider the case where, for a same operator, both CPU and GPU implementations are available, similar to what has been realized in the previous section for the K-NN operator. At runtime, the scheduler decides which implementation to instantiate depending on the costs and the available resources.

Our contribution in this section covers the design of a scheduling model suitable for such heterogeneous processing frameworks. We cover the modeling work in this paper and reserve the testing figures on scenarios at scale for a future work.

A. Task abstraction

In addition to operators that have either a CPU or a GPU implementation and require scheduling on their preferred resource, we add the notion of *heterogeneous task*. This defines a task that has two implementing operators, one targeting the CPU and one targeting the GPU for performance. Both of these operators produce the same results: the only difference lies in their implementation and target architecture. Heterogeneous tasks can thus be scheduled at runtime either on a CPU or a GPU by instantiating the right implementation.

B. Fine-grain scheduling model

We follow the fine-grain scheduling approach proposed by Quincy [12]: the scheduling problem is mapped to a graph representation where edges, weights and capacities model different scheduling policies encompassing competing demands and properties such as data locality. We now add resource preference to this approach.

1) Min-cost flow: using this graph representation and a min-flow formulation of the problem, [12] assumes it is possible to find an exact solution matching the scheduling.

A flow network is a directed graph composed of edges s annotated with a positive capacity y_e and a cost p_e, and nodes v annotated with a supply ϵ where $\sum_v \epsilon_v = 0$. A feasible flow assigns a non-negative integer $f_e \leq y_e$ to each edge such that for each node v:

$$\epsilon_v + \sum_{e \in I_v} f_e = \sum_{e \in O_v} f_e$$

where I_v and O_v are respectively the set of incoming and outgoing edges of v. In a feasible flow the left-hand side of the equation is referred to as the *flow through node* v. A min-cost flow solver can then be used to find the exact matching to the scheduling problem. For a feasible flow each task must find a path for its flow to reach the sink, by "flowing" a unit of scheduling from the task to be scheduled to the sink node S. Figure 9 represents an example of such flow network.

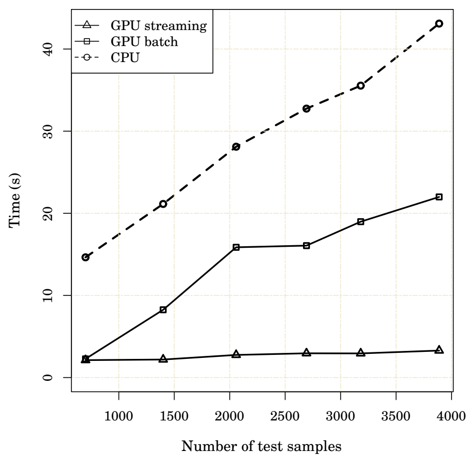
2) Modeling heterogeneous scheduling: applied to our case, we construct a graph encoding the operators ready for scheduling, *i.e.* that have received the end-of-file signal from the upstream operators, as well as the resources in the cluster. We express the preference of the operators, including the nature of the resource (CPU, GPU) and the data-locality, with the edges in the scheduling graph. In the network flow each task T_i has an edge to its preferred resource. Figure 9 depicts an example of the scheduling graph with three tasks ready for scheduling. In order to simplify the problem we consider only one cluster connected in the same rack.

Two particular nodes, X_{CPU} and X_{GPU}, represent the cases where a task is scheduled but not on its preferred resource, either on a CPU or a GPU resource. The associated cost with these edges is the cost of the worst case of scheduling the task on a CPU or GPU resource. Each task has an edge to the X_{GPU} or the X_{CPU} nodes, depending on its target implementation, or both in the case of a heterogeneous task. This is an addition to the original Quincy model where there is only one X node. Both nodes respectively hold an edge to every CPU or GPU resource in the cluster.

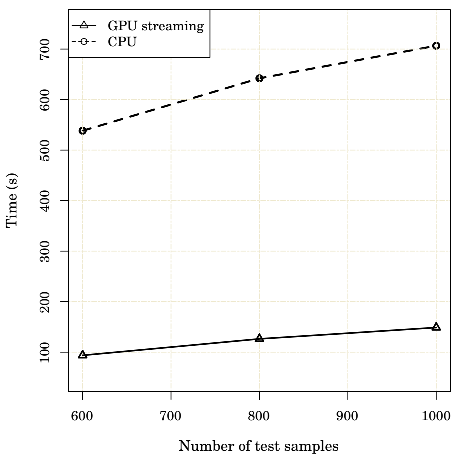
The U node represents the case where a node is left unscheduled. The cost associated with the edge from a task to this node is the cost of scheduling the task on its preferred resource, plus the waiting time for this resource be become available and a constant factor. This allows the task to wait for its preferred resource if it soon becomes available. The S node represents a successful scheduling of a task.

The cost from a task to its preferred resource includes the predicted time for transferring and processing the data on the resource. We infer the data processing and transfer times based on the size of the input data from a linear model built on historical execution times. This feasible on scheduling as the amount of data generated from the upstream operators is known.

Finally, for a heterogeneous task, the GPU or CPU implementation is scheduled according to the solution of the flow network.



(a) Dataset size = 35.000, k = 100



(b) Dataset size = 1.500.000, k = 1

Fig. 8. K-NN operator performance: CPU vs GPU streaming vs GPU batch, GPU: Nvidia GTX Titan, CPU: Intel i7-2620M @2.70 GHz.

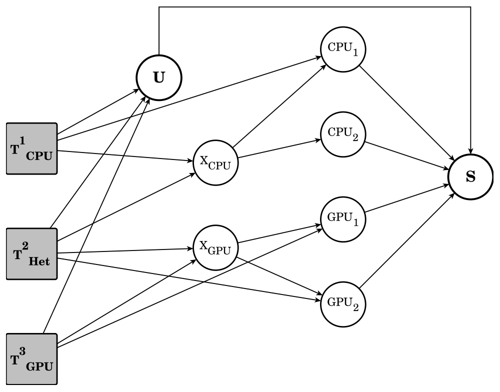


Fig. 9. Scheduler graph showing three tasks ready for scheduling. T^1_{CPU} is a CPU exclusive operator and T^3_{GPU} is a GPU exclusive operator, while T^2_{Het} is a heterogeneous task having a CPU and GPU implementation. Each task has a direct edge to its preferred resource CPU_i or GPU_i. The edges to the X_{CPU} and X_{GPU} nodes represent a task scheduled on the worst case resource. The edges to the U node represents an unscheduled operator. S is the sink node that drains all the flow from the graph. Edge weights are not represented.

V. Future work

In a future work we will present and discuss further the performance and strategies allowed by the proposed scheduling model. Preliminary tests have only validated the feasibility of the model. Implementing different scheduling policies favoring fairness or data locality is possible by varying the costs and weights in the scheduling graph and will also be studied.

Not only many real-life problems can be naturally modeled as graphs (*e.g.*, social, customers network), the current state of the research and technology already allows to store data as graph. A distributed graph database can be a good fit as storage backend of our proposed framework. In a future work we will evaluate the feasibility of integrating our distributed GPU-enabled processing framework within a distributed graph database. Challenges such as defining a proper graph model for both the database and processing on GPU, defining a suitable processing model as well as other possible usages of the GPU (*e.g.* indexing, caching) will need to ba addressed.

VI. Related work

In this section we review the literature related to the presented research.

A. Heterogeneous processing framework

Mars [13] is a MapReduce implementation running on GPU with extensions for supporting multi-GPU. Other MapReduce implementations exist such as [14], [15] but still for a multi-GPU setup on the same machine unlike our approach that targets distributed GPU environments and general-purpose computations. Dandelion [16] on the other hand is a distributed runtime and compiler on heterogeneous nodes equipped with GPUs. The framework takes a job expressed in the high-level query language LINQ as input and compiles it into various architecture code to be executed by the cluster runtime. It however does not seem to support streaming features and hiding of data transfer and execution. With regard to the future work,

B. K-NN optimization

In order to avoid sorting the entire distances matrix, [17] maintains the k nearest neighbors in a heap data structure residing in GPU memory. In this strategy each thread block finds the k nearest neighbors of a single testing sample. Each thread processes the distances with coalesced access to the GPU global memory. If the current distance is lower than the greatest distance residing in the heap, the thread removes the greatest element from the heap and inserts the current element. Since it requires access to a shared data structure, thread synchronization is required when pushing to the heap which is a costly approach compared to our optimized radix sort. In [18] a comb sort algorithm retrieves the k nearest neighbors. The comb sort stops once the k smallest distances and their corresponding training samples identifiers are found. They acknowledge that for $k > 100$ using the radix sort on the entire distances matrix is faster. Our approach, in addition to relying on the radix sort, does not depend on k, which is more stable and allows easier performance prediction.

C. Cluster scheduling

Mesos [19] is a cluster scheduler with a master *offer* level and a framework *demand* level. The implemented *Dominant Resource Fairness* allocation algorithm guarantees a fair allocation of $\frac{1}{n}$ of the dominant resource to each user. Our approach based on Quincy however favors data-locality over fairness, which mitigates the overhead in GPU memory transfers between operators [12]. YARN [20] is the second iteration of the Hadoop scheduler aiming at decoupling the programming model from the resource management infrastructure. YARN however considers resource requests from applications, unlike our approach that considers the demands over the framework. Omega [21] is a distributed cluster scheduler designed for framework and task concurrency that claims at optimizing the global cluster performance rather than the individual framework performance.

VII. Conclusion

In this paper we have presented our approach for optimizing the performance of a cluster of distributed processing with Graphics Processing Units equipping the nodes. Starting from our initial framework we presented a strategy optimizing the GPU implementation of the k-means algorithm.

Once we have acknowledged the performance gain of our proposed solution we have integrated GPU resources and computations in an existing dataflow graph processing framework using the same approach. In order to achieve that we have proposed a model for a GPU operator within the dataflow graph model exploiting the batch and streaming features of the GPU. The streaming GPU operator is furthermore well suited with the data streaming nature of DFG jobs. We have validated the approach on an optimized job implementation of the k-nearest neighbors algorithm.

Finally, we have presented the model for the scheduling on such heterogeneous processing framework, inspired by the Quincy [12] scheduling model. In a future publication we will show experiments and performance figures following thorough testing on the proposed scheduling model.

The code related to the presented contributions as well as the dataflow graph processing framework will be accessible in open source at https://github/nltran/arom.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04, 2004, pp. 10–10.

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, Mar. 2007.

[3] F. Perry, "Sneak peek: Google cloud dataflow, a cloud-native data processing service," http://goo.gl/6iWszM, 2014.

[4] A. Mahout Team, "Adding support for stratosphere as a backend for the mahout dsl," https://issues.apache.org/jira/browse/MAHOUT-1570, 2014.

[5] M. Banko and E. Brill, "Scaling to very very large corpora for natural language disambiguation," in *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, ser. ACL '01, 2001, pp. 26–33.

[6] N.-L. Tran, Q. Dugauthier, and S. Skhiri, "A distributed data mining framework accelerated with graphics processing units," in *Cloud Computing and Big Data (CloudCom-Asia), 2013 International Conference on*. IEEE, 2013, pp. 366–372.

[7] N.-L. Tran, "Accelerate distributed data mining with graphics procesing units." Presented as the 2014 GPU Technology Conference (GTC14), San Jose, CA, 2014.

[8] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Y. Yu, G. Bradski, A. Ng, and K. Olukotun, "Map-Reduce for Machine Learning on Multicore," *Advances in Neural Information Processing Systems*, vol. 19, pp. 281—-288, 2007.

[9] N. Tran, S. Skhiri, A. Lesuisse, and E. Zimanyi, "Arom: Processing big data with data flow graphs and functional programming," in *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, Dec 2012, pp. 875–882.

[10] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[11] D. G. Merrill and A. S. Grimshaw, "Revisiting sorting for gpgpu stream architectures," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10, 2010, pp. 545–546.

[12] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09, 2009, pp. 261–276.

[13] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A mapreduce framework on graphics processors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08, 2008, pp. 260–269.

[14] J. A. Stuart and J. D. Owens, "Multi-gpu mapreduce on gpu clusters," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '11, 2011, pp. 1068–1079.

[15] L. Chen, X. Huo, and G. Agrawal, "Accelerating mapreduce on a coupled cpu-gpu architecture," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, Nov 2012, pp. 1–11.

[16] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly, "Dandelion: a compiler and runtime for heterogeneous systems." SOSP'13: The 24th ACM Symposium on Operating Systems Principles, November.

[17] K. Kato and T. Hosino, "Solving k-nearest neighbor problem on multiple graphics processors," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, ser. CCGRID '10, 2010, pp. 769–773.

[18] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k nearest neighbor search using gpu," in *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, June 2008, pp. 1–6.

[19] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11, 2011, pp. 22–22.

[20] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13, 2013.

[21] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *SIGOPS European Conference on Computer Systems (EuroSys)*, 2013.