

Rafael Vital Rodrigues – 815993

Synthesis of the project developed for the course advanced programming for scientific computing

Project:

Speed up of an extremely randomized tree algorithm using GPU

Objective:

Implements a code that uses the graphic processor unit (GPU) to speed up the creation of an extremely randomized tree for supervised classification and regression problems.

The version for central processor unit (CPU) already exist and the job is developed a high parallelized version that can be applied on a GPU. To accomplish this we used the parallel computing platform and programming model CUDA, invented and provided by NVIDIA.

The extremely randomized tree algorithm

The extremely randomized tree algorithm, or extra tree ensemble algorithm, is an extension of the Extra Tree algorithm, where we don't build only one Extra Tree, but a forest of **M** Extra Tree.

Each Extra Tree is constructed in a randomized way, where the splitting parameters, attribute and cut-point, are chosen randomly. Not exactly random, because we select a set of **K** randomly pair of attribute and cut-point, and pick the pair which the reduction of variance is largest.

Here is a pseudo code of the algorithm: (Extracted from [1])

Build an extra tree ensemble(S).

Input: a training set S.

Output: a tree ensemble $T = \{t_1, \dots, t_m\}$

- For $i = 1$ to M
 - Generate a tree: $t_i = \text{Build an extra tree}(S)$;
- Return T.

Build an extra tree(S).

Input: a training set S.

Output: a tree t.

- Return a leaf labeled by class frequencies (or average output, in regression) in S if
 - (i) $|S| < n_{\min}$, or
 - (ii) all candidate attributes are constant in S, or

(iii) the output variable is constant in S

- Otherwise:

1. Select randomly K attributes, $\{a_1, \dots, a_k\}$, among all (non constant in S) candidate attributes;
 2. Generate K splits $\{s_1, \dots, s_k\}$, where $s_i = \text{Pick a random split}(S, a_i)$, $\forall i = 1, \dots, K$;
 3. Select a split s^* such that Score (s^* , S) is maximum
 4. Split S into subsets S_l and S_r according to s^* ;
 5. Build $t_l = \text{Build an extra tree}(S_l)$ and $t_r = \text{Build an extra tree}(S_r)$ from these subsets;
 6. Create a node with the split s^* , attach t_r and t_l as left and right subtrees of this node;
- Return the resulting tree t.

Pick a random split(S, a)

Input: a training set S and an attribute a.

Output: a split.

- Compute the maximal and minimal value of a in S, denoted respectively by a_{\min} and a_{\max} ;
- Draw a cut-point a_c uniformly in $[a_{\min}, a_{\max}]$;
- Return the split $[a < a_c]$.

The Fitted library

The Fitted library is a library developed by Matteo Pirotta and Marcello Restelli at Politecnico di Milano where was implemented the pseudo code in C++.

Further the library contains a series of Class that are responsible to read a Dataset from a file and manage it in order to obtain good train result with low bias and variance. The idea is extended this library in such way we can enjoy the power of GPU to speed up the train, but remains with the same interface of the current library.

The most important classes of this library are:

Dataset: The dataset is vector that store the samples. Where each samples is a tuple of input attributes and a tuple of output, usually only one output. The method of dataset are able to:

- Read and write dataset on a file.
- Obtain different dataset starting from the original dataset, the segmentation can be by inputs or by samples, i. e., we can obtain a reduced dataset that have less input attribute than the original one or have a subset of samples. This is very useful to do the cross validation, where we split the dataset in two subset, one to train, and another to validate.
- Calculate Mean and Variance of the outputs.

Regressor: Is a class that manage a generic type of regressor. The method are:

- Train: Train the Regressor given a dataset.
- Evaluate: Compare the result of the trained Regressor with the real output.
- ComputeTrainError.

- ComputeResiduals.
- CrossValidate.

ExtraTree: A class that build a Tree according with algorithm described, this class inherits from regressor.

ExtraTreeEnsemble: A class that build a tree ensemble according with algorithm described, this class inherits from regressor, but not from ExtraTree, it uses ExtraTree for build the singles tree.

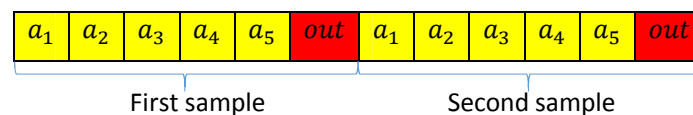
rtANode, rtINode, rtLeaf: Classes that define a ExtraTree with its internal node (attribute and split value) and leaf node (class frequencies or average output). So when we evaluate an extra tree, we traverses the tree following the internal node instructions until arrive on a leaf node that assert the result. On an extra tree ensemble tree, the evaluation is the mean (or mode) of the evaluation of each tree.

The extra tree on the library is described by an rtANode called root. This node points for its children node and so on.

The Extended Fitted library:

We will starting doing a brief description of the new classes created, after we will do a deep explanation of how each task is done.

DataCu: This class do a copy of all dataset to the device memory. Instead of the class Dataset, this class don't have methods that segments the dataset. The dataset on the device is stored as a unique vector ordered by sample, i.e., the first position assert the first attribute of the first sample, the second position, the second attribute, and so on, until arrive to the last input. After we list the output of the first sample and restart the process for the other samples.



NodeCu: Each object of nodeCu represents a subset of samples of the dataCu. To accomplish it, the object have a vector that indicates which samples of the dataset belong to the nodeCu. Further this class has the most important method that analysis the dataset of the node and decide if this dataset is "leaf dataset" or "an internal dataset", if it is an internal, the method will search for the best random split and will divide this dataset in other two dataset following the same rules of serial algorithm, the difference, of course, is that this method do it in parallel using the GPU processor.

ExtraTreeCuda: This class is a specialization (inherit) of the class ExtraTree in which the only difference is how the train of the dataset is done.

ExtraTreeEnsembleCuda: Differently from the ExtraTreeCuda, it is not a specialization of the class ExtraTreeEnsemble, but another type of the regressor. Although the major features available on ExtraTreeEnsemble are available also on ExtraTreeEnsembleCuda too, for instance, train and evaluate method.

LinkCu: This class do a link between the nodeCu and the node of the tree.

The Parallel Train of an Extra Tree and Ensemble Extra Tree

Differently from the serial version, the tree is built with a strategy similar to Breadth-first. We created a list with the nodes that need to be processed. So, we computed the nodes and add their children, if exist, to the list. This process continue until the list ends.

Here when we say node we want refer to a node of the tree, but a node of the tree don't know what subset of dataset it express, so we need a way to connect the node of the tree and a subset of dataset, this is the job of the object of the class LinkCu, that do a connection between the nodeCu (subset of dataset) and the tree nodes. So, our list is a list of object LinkCu.

The Ensemble Extra Tree train is made exactly in the same way of the Extra Tree train, we put the nodes that need be processed in only one list. Thus, this list has the nodes of all single tree.

LinkCu process

When we are processing a node of type LinkCu, in the hood, we are processing a node of type NodeCu and use the decision of the NodeCu, if the dataset is a leaf or not, to create the tree.

The pseudo code is the following:

LinkCu process (LinkCu).

Input: a LinkCu with contains a **nodeCu** (subset of the data) and a **node of a Tree**

Output: none or two children LinkCu

- Process nodeCu
- If nodeCu is a leaf
 - 1) Fit the output data (mean or mode)
 - 2) Declare the node of the tree as leaf
 - 3) Return none child

- If nodeCu is internal node
 - 1) Declare the tree node as leaf
 - 2) Create two new LinkCu object using two new nodeCu children received and the tree node location
 - 3) Return both LinkCu children

NodeCu process

The use of the GPU for speed up the train happens in this method. This method decide if the subset of the dataset is “leaf dataset” or “an internal dataset” , if it is an internal dataset, the method will generate a random split for **all** the input attribute and find the best pair (attribute, random split) that reduce the variance of the split dataset. At the end, the method will return the two new nodeCu children created with the split pair chosen.

Here is a big difference with respect to the serial algorithm, the parameter **K** is always equal to the number of inputs attributes.

Ahead is the pseudo code:

NodeCu process (NodeCu).

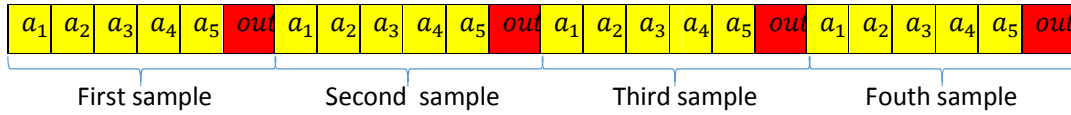
Input: a NodeCu

Output: none or two children NodeCu, and the split pair

- Verify if the size of NodeCU \leq nmin, if yes, this nodeCu is a leaf and return none.
- Launch a kernel on GPU that verify if all input attribute or the output are constant, if yes, this nodeCu is a leaf and return none.
- Launch a kernel to identify the maximum and minimum of each attribute.
- Chosen a random (uniformly distributed) split for each attribute, based on the maximum and minimum.
- Launch a kernel that do the sum and the sum square of all output elements in which its input attribute is smaller than the split value.
- Calculate the “variance” using the value of sum and sum square.
- Chose the split pair that gives a better score.
- Launch a kernel that divide (partitions) the “map” of the current nodeCu in two news “maps” using the split pair.
- Create two new nodeCu object using these news “maps”.
- Return the two object and the split pair

Maps, stride and sample

First of explain how the kernel works, is better explain what is map, stride and samples, and the relationship between then. A figure is a good way to do it.



Imagine that this vector represents all the dataset of the problem and suppose that the nodeCu, is the subset that contain the second and fourth samples. So $map = \{6, 18\}$.

The map store the address of the first input of each samples. A simple way to know what is the index, is do the operation $index = (sample\ number - 1) * stride$, where stride is the sum of number the input with the number of output. Therefore, all nodeCu object is defined by two things: a map (a simple vector) and a pointer that points to the complete dataset stored on the device (this pointer is equal to “all” nodeCu object).

Score

To decide what the better split is we use the following formula.

$$score = 1 - \frac{Var_{minor} * Size_{minor} + Var_{major} * Size_{major}}{Var_{total} * Size_{total}}$$

Where Var means Variance, total represents the complete subset, minor, the subset where the selected input is smaller than the split value and major, the subset where the selected input is greater than the split value.

In practice we don't calculate the variance nor the major part of the formula. We calculate the value of $Var_{minor} * Size_{minor}$ direct, using the equation $Var_{minor} * Size_{minor} = sumSquare_{minor} - (sum_{minor})^2 / size_{minor}$.

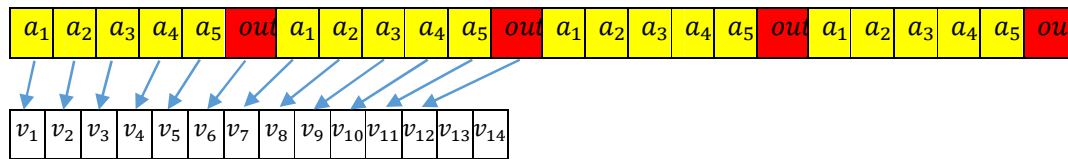
The sum_{major} parcel we obtain extrapolation through the equation $sum_{major} = sum_{total} - sum_{minor}$.

The Kernels

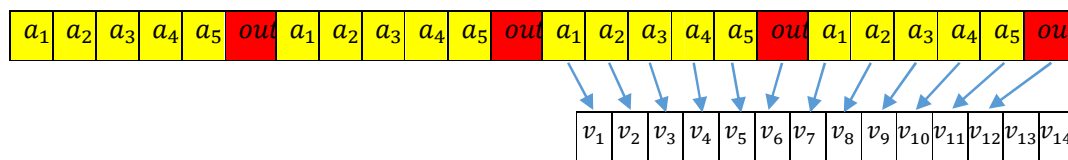
The kernel is a function that take place on the GPU. A GPU function need to be designed to work in such way that thousands of thread performance their jobs simultaneously. Further when the kernel need to access the device memory, thread that are closest of each other, should preferably access near memory. So we can't do the operations by input, because if we do it, we will access memory that are not coalesced and lost speed, so we need to do the operation by samples. (We store the dataset by samples because the operations was been planned to work by samples).

In the figure bellow we can see the dataset, represented by the long vector, the variables of each thread, represented by the white vector, and the threads, represented by the arrows, we suppose in this case that we have only 14 thread. In the first occasion, the thread operates on the first two sample and store the result in the local variable of the tread. In the second occasion, operates on the third and fourth samples and store the result in the local variable, these occasion repeat until we arrive the end of the dataset.

First Occasion



Second Occasion

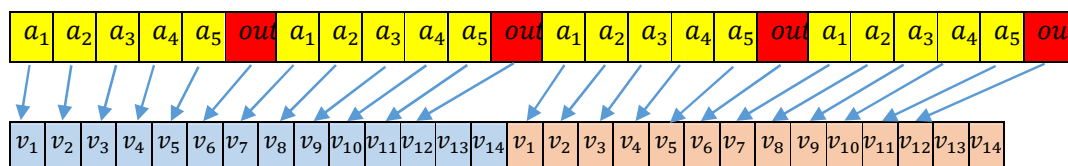


After we scan all the vector, we remain only with the local thread variable, in this example we have two instance for each input and output. So our next step, is operate between them in such way to obtain only one instance per attribute. This operations is well known and is named Reduce.

This figure is only a small example, because in real cases we have around tens of thousands samples and the thousands of threads. Another abuse that we did on the diagram is that the group of thread is divided into blocks, which was not taken into consideration.

Depending on how many thread per blocks we have, the kernel behavior change a little. We have two situations, one when the number of thread per blocks is greater than the stride, another when it's smaller.

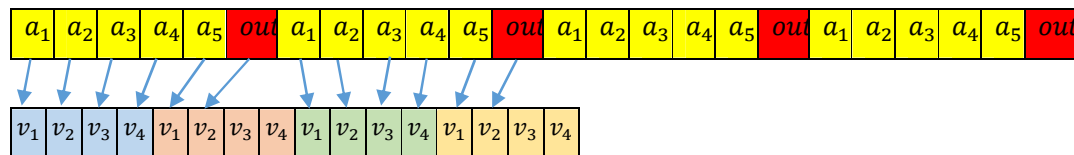
Diagram of first case:



In the first case, we have more thread per block than attribute, so each block (indicated by the colors) is responsible for analyzing two samples. As we have two blocks, each occasion reads four samples. After scanning all the occasions, each block reduces itself to get only one instance of each attribute.

Although, we have two blocks, so we need one more step that operates between the blocks. This task is done by the CPU, because in real cases the number of blocks is in the order of dozens, thus it's a short computational task.

Diagram of second case:



In the second case we have more attributes than threads per block, so each block (indicated by the color) is responsible for analyzing only part of the sample. In this example, we need two blocks for each sample. As we have four blocks, each occasion reads two samples. After scanning all the occasions, we don't need to do the block reduce, because certainly each block has only one instance of each attribute.

However, the CPU needs to join the information that is spread on different blocks, further to do the operation between the blocks.

Kernel Occupancy

When we launch a block we need to decide how much blocks, threads per blocks, and which case we want to use. These values must take account the computation capability of the device and the type of the dataset, in such way to obtain the maximum occupancy.

The occupancy is limited by four factors.

- Number of blocks
- Number of total threads launched
- Number of shared memory
- Number of registers

In our kernels the great problem are the number of registers, so to decrease these numbers, we pass some parameters at compile time using templates. However, this method brings rigidity to the code. (One way to decrease the rigidity, is to create a factory for the kernel launch).

The algorithm have four kernels, the first three are done as described, unless the kernel `verifyConstant` that have a little difference, because if the kernel identify that the attributes aren't constant the kernel finish before scan all the dataset.

The last kernel, that partitions the current map, don't follow these rules and its implementation is done using the external library CUB.

The Kernel Control Variable

Now we will explain the meaning of each control variable of the kernel.

For the kernel of first case, the control variable are:

$\text{unsigned readOnly} = (\text{blockSize} / \text{stride}) * \text{stride}$: This variable indicate what threads on the block are used (all thread whose `threadIdx` is lesser than `readOnly` are valid). In all examples presented, we can see that some thread wasn't used, it is normal, but we need choose a good value of `blockSize` (or Thread per block) to minimize the number of useless thread.

$\text{unsigned mod} = \text{threadIdx.x} \% \text{stride}$: Each thread correspond to an attribute, this variable indicate which one.

$\text{unsigned step} = (\text{blockIdx.x} * \text{readOnly} + \text{threadIdx.x}) / \text{stride}$: Each thread correspond to a sample, this variable indicate which one.

$\text{unsigned stepIdx} = \text{readOnly} * \text{gridDim.x} / \text{stride}$: Show how much samples are read on each occasion.

So to access an element of the dataset we use the expression: `data[map[step]+mod]` that can be translate to: Read the attribute **mod** of the **step**-th samples of the subset of dataset described by **map**.

For the kernel of first case, the control variable are:

$\text{unsigned mod} = \text{threadIdx.x} + \text{blockIdx.x} \% \text{block_per_sample} * \text{blockSize}$: Each thread correspond to an attribute, this variable indicate which one.

$\text{unsigned step} = (\text{blockIdx.x} / \text{block_per_sample})$: Each thread correspond to a sample, this variable indicate which one.

$\text{unsigned stepIdx} = \text{gridDim.x} / \text{block_per_sample}$: Show how much samples are read on each occasion.

To access an element of the dataset we use the expression: `data[map[step]+mod]` that have the same mean of the other case. In this case there are useless thread as well, but we don't need another variable to control it, is enough respect the condition **mod < stride**.

Fit data

Until now, we cover almost all the points of the extended library, missing say only about the process of Fit data, executed by the LinkCu's method processLink during the creation of a leaf.

Who do this job is the CPU and it is done in parallel with the tree building. To accomplish it, we create a thread on the CPU that is responsible to fit data, while the main thread continue to manage the tree building, i.e., expanding the internal node, in this way the calculation of the mean or mode don't interfere with the remainder tree.

Would be also possible force the GPU to do this task, however as leaf have few data, the GPU wouldn't use all its capacity.

CPU building the Tree

Following the same idea of the fit data, when the node have few data, the GPU efficient feel and can be more efficient build the tree with the CPU. Hence, when the size of the dataset is small enough, the extended library launch a new CPU thread that build a tree with remainder dataset and add this tree to the bigger one that are being processed by the GPU.

Version 2

The first version of the code have a problem where independently of the number of samples on the dataset, the kernel use all the GPU source to process this data, but sometimes, this source don't work with a satisfactory computation power.

In other words, if I launch a kernel with the maximum GPU occupancy and this kernel is computationally small, each thread will do a small work and some thread will do nothing, so the global efficiency will be smaller.

To solve this problem, we can launch two kernel simultaneously, so each kernel alone don't use all the GPU resource, however, the used threads do a bigger work, reaching better results.

This technique, multi kernel launches, can be used in our problem, but we prefer don't use it. Because if we use it, the CPU need synchronize, launch and control the kernels that can be computationally costly.

Therefore, we used a "virtually multiple kernels", i.e., we created newer functions and kernels where instead of process only one NodeCu, they process two or more nodeCu. However, these nodeCu need to have approximately the same number of samples, because the GPU resource are divided equally. To accomplish it, we changed three things:

- The first one was change the list order. Before, our list was similar to a breath-first list, now it is based on the number of samples. Thus, we changed the type of the list to the standard ordered set.

- The second change was add a chosen parameter that decide when we process one or multiple nodeCu (linkCu). The number of the samples that delimit the frontier between to process one single node, or two node, or three node and so on, depends of the number of attributes and of the graphic card capacity (number of thread, multiprocessor, maximum number of blocks). Thus, it need to be optimized "by hand" with a tuning parameter OTI that represent these capacity.

- The third change was create new functions and kernel that instead of receive a pointer that represents the data of one single node, receive a vector of pointer, one pointer for each node, and process the vector of pointer simultaneously.

The function processLink of linkCu, that is responsible for call the function processNode and create the tree, had only a small difference: We create **for loop** and put all the instruction that operated on linkCu object into these loops, therefore we scan the vector of linkCu object. The new functions is called processVectorLink.

The function processNode, called by the processLink function, was replaced by the function processVectorNode. This function is similar to processNode with **for loops**, however, some barrier complicate its structure.

The first obstacle is declared and allocate on the GPU the pointer of pointer (the vector that contains the pointers to the correspondent data) required by the kernel. The pointer of pointer was allocated in this way:

```
unsigned** d_mapVec = NULL;
unsigned** h_mapVec = NULL;
sizeAlloc = sizeof(unsigned*)*nNode;
cudaMalloc((void**)&d_mapVec,sizeAlloc);
h_mapVec = (unsigned**)malloc(sizeAlloc);
for(unsigned i=0; i< nNode; ++i){
    h_mapVec[i] = nodeCuVector[i]->map;
    // nodeCuVector[i]->map is a pointer for an address memory of the GPU that had already been allocated
}
cudaMemcpy(d_mapVec, h_mapVec,sizeAlloc,cudaMemcpyHostToDevice);
```

Here is important note that h_mapVec is a vector on the host memory (CPU) that contains a pointer for the device memory (GPU) and d_mapVec is a vector on the device memory (GPU) that contains a pointer for the device memory (GPU).

The other difficult is merge the reply vector of the kernel, for accomplish it we create the control variable **nodeId** that indicate the position of the n^{th} nodeCu in the reply vector.

The kernel functions that search the **maximum and minimum** values of each attribute and the kernel that calculate the **sum and sum square** are unique called kernels, i.e., each processVectorNode function call each kernel once (one call per Vector, not per Node). To achieve this task new kernel were created, in such way, the total number of blocks launched are equally divided into **N** node groups and each group process separately each node. On these kernel some changes on the control variables are required. The major change is the creation of a new variable responsible to indicate the “true” blockIdx.x and modify the other control variables to use it. These kernel receive into their call pointers of size and pointer of pointer of data.

In the other hand, the kernels responsible for **divided (partitioned)** the “map” are unchanged, and are called on time per node. It can be modified to work like the other kernel, however, the benefit are low, because if we bring the execution time of this task to zero, the overall execution time will be reduced approximately by five percent.

The kernel that verify if the attribute are constant don't exist anymore because is not efficiently launch it. Suppose that we launch a kernel to verify if the attribute of three different node are equally, thus, if the kernel discover that the first node isn't constant, the kernel stop scan the first node, however the second and third node are still being verified. So, the GPU will not use all its resource until the end of the work.

Don't verify the constant attributes, don't bring any damage to the algorithm, since the score function is "zero" when the attribute are constant and it will not be chosen as the better split.

Results

The object of the project is develop an extended library that is faster than the original library, we are not interested in the performance of the algorithm, i.e., if the algorithm learn the dataset well, with low bias and standard derivation. Our main goal is obtain a code that learn the dataset, arriving to the same result, in less time.

We tested the extended library on three dataset with different number of samples and input attributes. The test was executed on a machine with processor Intel core i7-2GHz-8GB of RAM and GPU Nvidia GFORCE-720M-2GB. The dataset are:

	Number of Samples	Number of Inputs	Number of Outputs
Data1	2310	18	1
Data2	61680	18	1
Data3	43897	442	1

The **first** dataset was used to verify the correctness of the extended library with respect to the original library, thus we divided the dataset in five sub datasets and run the extremely randomized tree algorithm on each one, doing a cross validation between them.

We run the algorithm with parameters M (number of tree) equal 50, k (number of tested attributes) equal 18 - remember of the "limitation" of the extended library on which k is always equal the number of inputs - and nMin (number of samples in the leaf node) equal 1.

To compare the performance of extended and original library we used the index RMSE (Root Mean Square Error), MAE (Maximum Error/standard derivation), the time of execution and the structure of the result tree, confirmable through the number of the tree nodes created consequently the length of the log file.

	Extended Library	Original Library
MSE	0.591666	0.607657
MAE	1.7334	1.86023
Time (s)	1.98	0.39
Output File (KB)	126	128

Comparing the table we can conclude that the extended library is working well and the small difference exist because of the random feature of the algorithm. The time of execution on this experience of the extended library are five timer greater, it happens because the GPU is not working with all its capacity. It is expected since the number of samples is small.

The **second** dataset is equal to first dataset with its samples replicated in such way we obtain a larger dataset. Here, we did almost the same analysis of the first samples, however we used M equal 20 instead of 50.

	Extended Library	Original Library
MSE	0.000438382	0.0286809
MAE	0.00766629	0.164994
Time (s)	1.71	6.01
Output File (KB)	55	53

In this table, we can see that the execution time of extended library is four time smaller than the original library. The difference of the MSE and MAE occurs because the cross validation take distinct samples.

The third dataset is a particular dataset, it has 441 inputs. Since the dataset is huge and the execution time is high, we executed the test for only one tree (M = 1).

	Extended Library	Original Library
MSE	0.0200571	0.0200571
MAE	49.8943	49.8943
Time (s)	1.30	45.7512
Output File (KB)	12.6	14.6

The comparison table show a huge difference on the execution time, around a speedup of 35 times. This example demonstrate the powerful of the GPU when we have a huge number of data.

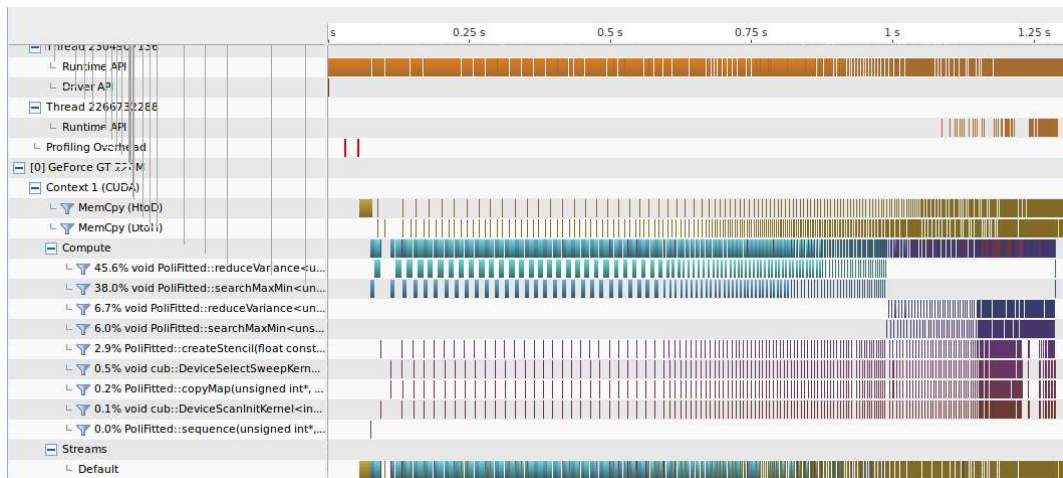
However, this is not a fair comparison because the serial code was note optimized and was running with only one core.

Performance Features

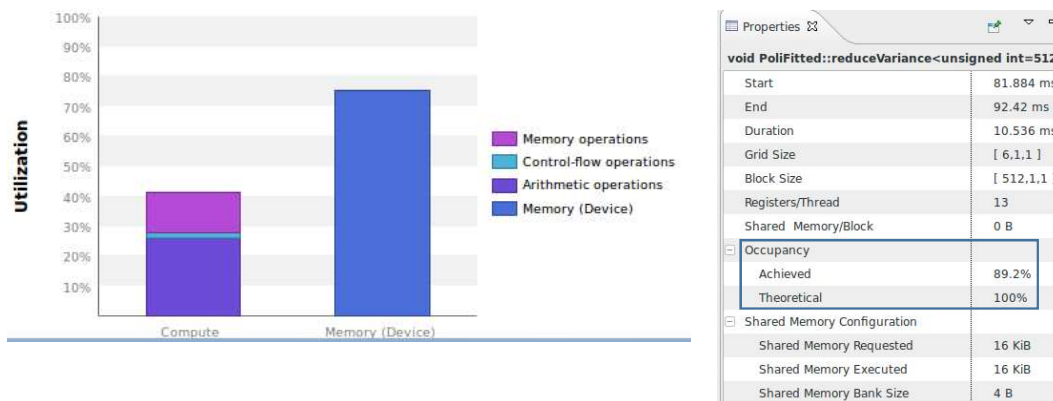
To examine the code is necessary use some tools that present information of how the code works and where its weak points is.

The feature that we used is the NVIDIA Visual Profiler, this tools do a profiler of all functions and show how each kernel (function) are running, what are the bottleneck, which kernel are more expensive and how optimize it.




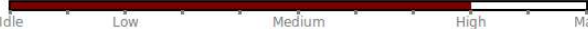
The first scream show us what kernel is more costly. We have two kernel that dominate the program, the kernel serchMaxMin and reduceVariance. So, to reduce the execution time we need to improve these functions.



On the next figure, we will analyze the kernel `reduceVariance` to identify what is its bottleneck. The operation `reduceVariance` is memory limited, because we have tens of calculations for each memory access. Furthermore, the memory access is slower, around hundreds times slower, thus the algorithm bottleneck is the memory transfer. Moreover, we can see that the occupancy is high.



Once the bottleneck is the memory transfer, is important check if all the bandwidth are being used or not. In our case the use of bandwidth is high, thus we are limited by the hardware. So to improve even more the execution time of the algorithm, using this hardware, is necessary to reduce the number of memory tranfers.

Results			
	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Global Loads	2414356	37.442 GB/s	
Global Stores	420	4.687 MB/s	
Atomic	0	0 B/s	
L1/Shared Total	2414776	37.447 GB/s	
L2 Cache			
L1 Reads	2680304	10.392 GB/s	
L1 Writes	1209	4.687 MB/s	
Texture Reads	0	0 B/s	
Atomic	0	0 B/s	
Total	2681513	10.396 GB/s	
Texture Cache			
Reads	0	0 B/s	
Device Memory			
Reads	2436310	9.446 GB/s	
Writes	1001	3.881 MB/s	
Total	2437311	9.449 GB/s	

Conclusion

The objective of the project was reached, in the best of the case the speed up is 35 times, although the comparison is not so fair because the serial code was not optimized.

Furthermore, the test isn't enough to conclude if the extended library is efficiently as it seen, because the GPU used for testing was developed for graphic processing, thus its calculus performance is doubtful and maybe the code was optimized in the wrong way, i.e, the code operate in the peak performance for this GPU and not for a generic GPU.

Hence, a crucial next step is test the algorithm performance on a workstation with a Tesla GPU, specialized on scientific calculus, and verify if the extended library exploit all the capacity of this device.

Bibliography

[1] Extremely randomized trees, Pierre Geurts, Damien Ernst, Louis Wehenkel, 2005.