# Parallel Multidata-HMM: A Study on GPU and Parallel Implementation of Hidden Markov Models for Speech Recognition

Mateo Bonilla L.[1]

[1]Electrical and Computer Engineering Department, New York University Tandon School of Engineering

## ABSTRACT

The purpose of this paper is to study and analyze different execution approaches of HMM for speech recognition, and determine a framework that can be used to extract high levels of performance in terms of speed, energy consumption, and CPU utilization. Three different approaches are proposed for this research which are: Single Core, Multi Core (Model and Data Parallel), and GPU implementation. For these configurations, analysis are done for a single data and a data parallel approaches. After the single data approach is developed, several sources of inefficiency are determined and tackled in the data parallel and GPU approaches. The results showed a speedup in execution for Data Parallel and GPU of 2.1x and 2.01x with energy efficiency of 2.52x and 3.01x respectively. Therefore, the Data Parallel and GPU configurations showed the expected improvement over the Single Core configuration for the HMM algorithm in a speech recognition framework.

## 1. INTRODUCTION

Hidden Markov Models (HMM) are statistical sequential data representations that are used to represent a randomly changing system [3]. The purpose of a HMM is to predict the future value of the certain observation, considering the information of the current state and working with the probability of different new outcomes to be observed in the future. Therefore, considering prediction and classification as the main objectives of HMM, the applications for this Machine Learning Algorithm go from Speech Recognition to predict and classify certain characters up to Computational Finance to predict changes in the stocks [1]. The amount of data that all these applications manage for performing good predictions is massive, and everyday requires more computational effort to operate in real-time scenarios. This increasing amount of data for machine learning algorithms requires the implementation of viable solutions for accelerating the process of data processing and evaluation of different models.

Taking in consideration this necessity of performing operations in larger amounts of data, this requires that the machine learning algorithms and systems become more effective and able to handle this larger amounts of data to achieve higher levels of performance and accuracy of the overall model [15]. Therefore, several different solutions have been presented to solve this problem and allow the implementation of machine learning models using parallel computation and GPU programming, which are common techniques that are used to increase computational performance in different algorithms. In the particular case of HMM algorithms, there are several complexities inside the process that can be solved using parallel computation.

Considering this previously mentioned information and necessity that appears in the area of Machine Learning and Real-time big data analysis, this research project studies different data and model parallel approaches to implement HMM in parallel and increase the overall performance of the algorithm. Therefore, to address the previously mentioned necessity of parallel machine learning configurations for HMMs, and the necessity of processing larger amounts of data to improve accuracy and performance, this paper presents a study of model parallel and data parallel implementation of HMMs in both CPU and GPU configurations. These configurations are proposed to improve the overall performance of the algorithm in terms of execution time, energy consumption, power, and CPU/GPU utilization. The main contributions of this study are:

- Introduce the big data analysis in HMM for speech recognition applications and real-time machine learning by studying the effects of the application of a big data model in a single core implementation of HMM.
- Optimize the single core HMM implementation to run in data and model parallel configurations to reduce the inefficiency and computational complexity of the model.
- Determine the different sources of inefficiency inside HMM models in terms of execution time and power consumption. Create a GPU implementation of the sources of inefficiency inside the model to optimize the HMM overall execution.

## 2. BACKGROUND AND RELATED WORK

Taking into account previous research and implementations in the area of machine learning acceleration and parallel Hidden Markov Model applications, this section is dedicated

to introduce additional details about the execution and operations that have to be performed inside this algorithm (both training and testing sections), as well as some additional topics about Machine Learning Acceleration and related work in the area of parallel HMMs.

## 2.1 Hidden Markov Models

According to the previously stated definitions on Hidden Markov Models, this algorithm is used as a probabilistic model, which has the main purpose of represent a sequence of observations that can be either time dependant or independent [3]. The origin of this algorithm goes back to 1960, where Leonard Baum and Lloyd Welch published the first theory on how this algorithm works, and since then, HMMs have been used for different applications along the filed of engineering (Speech Recognition, signal and systems analysis) and many other fields such as biology and finance [18].

Considering the previous stated definition, we can conclude that the main purpose HMM is to predict or classify different data according to the sequence of observation that it is available at a certain point in time [5]. As any other machine learning algorithms, the HMM algorithm requires to implement two different stages to obtain the most probable sequence of observations to predict future values inside the model. These two different stages are the training and testing algorithms that are required for the model to obtain the basic weights and parameters about the behaviour of the model and then use this parameters to test the model and obtain the most probable sequence of states for the prediction model, which for this study, is going to be used for a speech recognition process that represent one of the most important and computer expensive applications for this algorithm [18].

For the case of HMM, there are two popular models that are used for the training and testing process. For the case of the training, the Baum-Welch Algorithm is used to produce the expectation and maximization steps that are required to obtain the initial state probabilities of the model as well as the transition matrix [17]. On the other hand, the testing step is performed using the Viterbi Algorithm, which calculates the maximum a posteriori probability estimate of the model to obtain the most probable sequence of states considering the previous calculations from the training step [13]. The following subsections are going to explain both algorithms and calculations in detail.

### *2.1.1 Baum Welch Algorithm*

The Baum-Welch Algorithm, as previously mentioned, it is an algorithm used to calculate the initial probabilities and transition matrix of the model considering the initial data that it is used to train the model [16]. To calculate the transition matrix and initial probabilities of the model, the following calculations are required:

Consider a model:

$$S = \{S_1, ..., S_N\}$$

Where $S$ are the $N$ different states inside the model and there are $M$ observations inside it. To calculate the initial probabilities, the alpha and beta loop operations need to be performed. Therefore, consider the $A = \{\alpha_{ij}\}$ where $\alpha_{ij}$ is the probability of the state at time $t+1$. Therefore, the $\alpha_{ij}$ probability can be calculated as:

$$\alpha_{ij} = P_r\{q_{t+1} = S_j | q_t = S_i\}, 1 \leq i, j \leq N$$

This probabilities must satisfy the following normal stochastic constraint:

$$\alpha_{ij} \geq 0, 1 \leq i, j \leq N and \sum_{j=1}^{N} \alpha_{ij} = 1, 1 \leq i \leq N$$

Consequently, the beta calculation has to be performed to obtain the final value of the forward-backward algorithm to obtain the expectation maximization step and obtain the transition and initial probabilities matrix [17]. Consider $B = \{\beta_j(v_k)\}$ where $\beta_j(v_k)$ is the probability of symbol $v_k$ to be emitted in state $S_j$. Therefore the following expressions can be obtained:

$$\beta_j(v_k) \ = P_r\{o_t = v_k | q_t = S_j\}, 1 \leq j \leq N, 1 \leq k \leq N$$

This probabilities must satisfy the following normal stochastic constraint:

$$\beta_j(v_k) \geq 0 and \sum_{k=1}^{M} \beta_j(v_k) = 1, 1 \leq j \leq N, 1 \leq k \leq M$$

Therefore, the initial probability matrix can be calculated by considering these previously calculated values, and also the transition matrix is obtained from the maximization step as it is presented below:

$$\pi_i = P_r\{q_1 = S_i\}, 1 \leq i \leq N, \sum_{i=1}^{N} \pi_i = 1$$

Where $\pi_i$ represents the initial states probabilities of the model and, with the calculation of these probabilities, the transition matrix of the model is defined as $\lambda(A, B, \pi)$ which is a function of the previously calculated values.

### *2.1.2 Viterbi Algorithm*

On the other hand, the Viterbi Algorithm is used for the testing section of the HMM algorithm considering the previously calculated values in the Baum-Welch Algorithm. Therefore, this algorithm is used to calculate the maximum a posteriori probabilities to obtain the most probable sequence of states considering the previously given transition matrix and initial probabilities.

To calculate the maximum a posteriori probabilities considering the previously calculated values in the Baum-Welch Algorithm, the following function calculation has to be performed:

$$\gamma_t(i) = \frac{\alpha_i(i) A_i B_j(O_i) \beta_t(j)}{Pr[O|\lambda]}$$

It is important to mention that, for the case of the Viterbi Algorithm, this $\gamma$ function is calculated inside a loop, and requires additional expectation maximization calculations inside the loop as well as the probability density function

calculation to obtain the most probable function of states. Refer to Algorithm 2 to observe the complete operation of the Viterbi Algorithm.

## 2.2 Machine Learning Acceleration

Machine Learning Acceleration is one of the most relevant topics in hardware and software design nowadays. The main reason for this, is that humans produce even more amounts of data and information, which can be used and have many different applications in different industries. In addition, the industry efforts are now aiming for the creation of machine learning frameworks based on servers and cloud computing, which are easier to manage and can be accessible for larger amounts of people [9]. To be able to provide this accessibility and speed on machine learning frameworks, different software and hardware designs must be improved to accelerate machine learning task. Therefore, machine learning acceleration is currently being studied in different engineering areas to obtain even better results of the already existent models. To accelerate the models, there are two techniques that are being currently studied: adapt existing computer architectures to Machine Learning tasks and develop brand-new computer architectures and ASICs for Machine Learning [14]. In previous research work and literature [7] [4] [6] [10], different examples of machine learning accelerators had been presented and used to accelerate matrix multiplications inside neural networks or other algorithms that require extensive operations inside its implementations.

Considering these previously mentioned research papers, this study proposes a machine learning acceleration technique based on the utilization of commercial hardware and software. By improving the resource allocation and parallel code, the HMM machine learning algorithm is accelerated.

## 2.3 Related Work

In the last ten years, there has been different efforts for the parallel implementation and acceleration of HMMs algorithms using different software and hardware techniques, and also applied to different research and industry applications. For instance, Imad Sassi and et al. [16] propose a ParaDist-HMM implementation, which implements HMMs in a parallel distributed configuration model using SparK architecture and also proposing a Big Data approach for the acceleration and real-time execution of HMMs.

On the other hand, Jun Li and et al [11] proposes a parallel implementation (model parallel) of the forward-backward algorithm used inside HMMs, which can be used to evaluate several HMMs and fit them to a single observation sequence. This allows that multiple data can be evaluated at the same time using a parallel implementation of the model inside different CPU machines. In the case of GPU implementations of HMMs, Chuan Liu and et al [12] propose a CUDA implementation of the HMM algorithm for data with 512 states and 512 sequences, which results in large matrix multiplications. This study showed speedups of 180x for the Baum-Welch Algorithm in comparison to a CPU implementation.

Finally, in the area of HMMs algorithms for speech recognition and word analysis, Leiming Yu and et al [19] proposed a parallel GPU implementation of a HMM for isolated word speech recognition.

# 3. METHODOLOGY

For the methodology that was used in this study, different analysis regarding execution time, power, and accuracy were performed to determine which implementation of the Machine Learning Hidden Markov Model algorithm was better. To perform the study, Matlab software was used to create the source codes for each algorithm and for each of the proposed implementations. On the other hand, the main hardware that was used for this implementation was the Intel i5-11400F 6-core processor running at 2.9GHz and the NVIDIA GeForce GTX 1650 GPU which has 856 CUDA Cores running at 1665MHz.

As it was previously mentioned, the HMM model that was implemented in this study uses two main algorithms that are used to train and test the complete Markov Chain for speech recognition. The first is the Baum Welch Algorithm, which is used to train the HMM by finding the hidden parameters inside the Markov Chain. Consequently, the results of this algorithm are used as inputs of the Viterbi algorithm, which is used to determine the most probable sequence of states using the data of the hidden parameters that were previously calculated.

Using these two different algorithms, the most probable sequence of states is calculated for each of the observations in the model. To determine the optimized configuration for implementing these algorithms, four different approaches are studied: Single Core Implementation, Model Parallel Implementation, Data Parallel Implementation, and GPU Implementation. Therefore, using Matlab as the programming environment, the four configurations are implemented to determine under which configuration the proposed HMM model has better performance results.

## 3.1 Single Core Configuration

For the Single Core Implementation, both Baum Welch and Viterbi Algorithms are implemented using common sequential code which is executed inside one core following the order in which each section of the code was declared. To understand the overall implementation of the algorithms, we will consider each of the previously mentioned calculations to create the source code to implement the Baum Welch and Viterbi Algorithm for a single core implementation.

Considering that we only have one core to perform both algorithms and test the proposed HMM, the code will execute sequentially inside the machine, which means that computing expensive operations and data management will take most of the time to process. Therefore, to optimize some of the required calculation, vectorized code was used as well as different embedded Matlab functions. Therefore, as it was previously mentioned, the Baum Welch algorithm requires three computer expensive calculations which are: density function calculation, alpha and beta recursion for forward-backward algorithm, and the expectation step for estimation of the state probabilities. Therefore, the following code was implemented for these tasks.

Algorithm 1 presents the most important calculations that are performed inside the Baum Welch Algorithm that were previously explained. Observing the reported algorithm, it clearly shows how the calculation of each parameter is being done sequentially and in most of the cases, the data de-

**Algorithm 1: Baum Welch Algorithm - Single Core**

```
 % Probability density function
for i=1:N
  B(:,i) = mvnpdf(X,Mu(i,:),Cov(:,:,i));
end
B(B==0)=1e-200;

% Initial alpha step
alpha(1,:)=Pi.*B(1,:);
s(1)=sum(alpha(1,:));
alpha(1,:)=alpha(1,:)./s(1);

% Alpha recursion
for t=2:T
   for i=1:N
    alpha(t,i)=alpha(t-1)*T(i))*B(t,i);
   end
   s(t)=sum(alpha(t));
   alpha(t)=alpha(t)./s(t);
end

% Beta recursion
beta(T,:)=ones(1,N)/scale(T);
for t=T-1:-1:1
   for i=1:N
     beta(t,i)=T(i)*B(t+1)*beta(t+1)/(t)
   end
end

% E (Expectation step)
for t=1:T-1
 for i=1:N
  for j=1:N
  E(t,i,j)=Al(t,i)*T(i,j)*B(t,j)*Be(t,j)
  end
 end
 a=E(t,:,:);
 E(t,:,:)=a./sum(a(:));
end
Gamma=(sum(E,3));
```

**Algorithm 2: Viterbi Algorithm - Single Core**

```
%Compute the observation probabilities
for i=1:p
  x(i) = mvnpdf(obs,Mu(i,:),Cov(:,:,i))
end

%Gamma recursive calculation
for t=2:length(obs)
 for state=1:p
  delta(s)=delta(:,t)+(T(:,s)))+x(t);
  r(s,t)=r(:,t-1).*T(:,s)*x(s,t);
  mgamma=delta(:,t-1)+log(T(:,s));
  gamma(s,t)=mgamma;
 end
 scale(t)=sum(r(:,t));
 r(:,t)=r(:,t)./scale(t);
end

%Map Calculation
s_MAP=zeros(1,length(obs(:,1)));
[~,s_MAP(end)]=max(delta(:,end));
Prob_max=max(delta(:,end));

%Backtracking
for t=length(obs)-1:-1:1
    s_MAP(t)=gamma(s_MAP(t+1),t);
end
max_ind=s_MAP;
```

pends on the previous result which is a certain limitation that this machine learning algorithm has for implementing it in a parallel execution. After the results from the Baum Welch Algorithm are obtained, the Viterbi algorithm is performed to obtain the most probable sequence of states. For this algorithm, the calculation of the probability density function, gamma, and the sequence map are the computer expensive calculations that this algorithm requires. Therefore, consider the following code specified in Algorithm 2.

Using Algorithms 1 and 2, the Single Core configuration is performed and used for in the experimentation section to obtain the total execution time, energy, and CPU utilization for each of the cases to then create the Multi Core and GPU configurations.

## 3.2 Model Parallel Configuration

In Machine Learning implementations and parallel computer context, a Model Parallel Implementation is a technique in the area of distributed training and learning in which it is not possible to fit the entire model on a single machine or processor, which the model to be deployed on different machines to take advantage of executing operations in parallel for better performance [8]

For the Model Parallel Implementation configuration, each of the previously mentioned algorithms are optimized to run different tasks in parallel and take advantage of the multiple cores that the used processor has in its configuration. To create the Model Parallel Implementation, both algorithms are optimized, which allows certain operations to be calculated in parallel to obtain the results faster and using less power. Therefore, some of the previously mentioned calculations are now being performed in parallel.

Observing the previously mentioned code, it can be determined that certain parts of the code for the Baum Welch Algorithm are running in parallel to optimize the complete execution the complete framework. The same optimizations can be performed for the Viterbi Algorithm in the testing implementation of the model.

Therefore, observing the previously presented code optimizations, the model is now deployed in parallel to obtain better performance in both execution time and energy as well as hardware utilization of this specific processor. This is because these proposed code optimizations use the *parfor* loop implementation, which causes is used to implement parallel loops inside the 6 different cores of the processor and take advantage of a parallel calculation of these portions of code. It is important to mention that the allocation of workers in

**Algorithm 3 - Model Parallel: Baum Welch Algorithm - Operations that can be executed in parallel**

```
%Mean-Covariance for each observation
parfor i=1:N
    Cov(:,:,i) = CovCell{i}
    Mu(i,:)= MeanCell{i}
end

% Probability density function
parfor i=1:N
  B(:,i) = mvnpdf(X,Mu(i,:),Cov(:,:,i))
end
B(B==0)=1e-200

%Final Mean/Covariance
parfor i=1:N
  Mu(i)=X(1:end)*Gamma(i))/sum(Gamma(i)
  d=X(1:end,:)-Mu(i);
  Cov(i)=(d*Gamma(i)*ones(p))*d/Gamma(i)
end
```

**Algorithm 4 - Model Parallel: Viterbi Algorithm - Operations that can be executed in parallel**

```
%Compute the observation probabilities
parfor i=1:p
  x(i) = mvnpdf(obs,Mu(i,:),Cov(:,:,i))
end

%Initial Probability Initialization
parfor i=1:p
    delta(i,1)=log(Pi(i))+log(x(i))
    prb(i)=Pi(i)*x(i);
end
```

this case is done automatically by the software, which can represent certain limitations inside a more interactive way of allocating work inside the different cores, which is something that it is performed in the next configuration.

Nevertheless, there are some sections of code in both algorithms that cannot exploit good levels of parallelism. The main reason for this is the dependence that the calculation of the future value has with the existence of the present value. This means that there are some values that depend on the present value to calculate the next one. These type of dependencies cause that the certain parts of the code cannot be calculated in parallel, because they will cause additional communication overhead that can downgrade the overall performance of the configuration.

## 3.3 Data Parallel Configuration

Contrary to the Model Parallel Implementation, the parallel computing machine learning models propose the Data Parallel Implementation method, which is used to deploy different operations in parallel using different portions of data and setting their independent calculation in a different core using the single core implementation [2]. After the calculation in each core is performed, all the results of the partitioned data are used to determine the gradient of data, which is the one that it is used for the further implementation of the algorithms [20].

For the Data Parallel configuration, more design and interactive parallelization techniques were used to obtain better results in terms of performance and hardware utilization. Given the nature of the data parallel configuration, the complete HMM model framework for a single core implementation was deployed inside the 6 different cores that the used CPU has in its configuration. To control which worker did which part of the model, the *spmd* block in Matlab is used for the supervised allocation of work inside each of the cores. The *spmd* block in Matlab stands for Single Process Multiple Data, which allocates an specific index identifier for each of the available workers (cores) in the processor, and allows to manually allocate each model and data inside each of the cores.

**Algorithm 5 - Data Parallel: Worker allocation for both Baum Welch and Viterbi Algorithms**

```
%Worker Allocation
spmd(6)
 if labindex == 1
  [M,C,Ma,P]=BaumWelch(train{1},HS);
  MuStruct = M;
  CovStruct = C;
  MatrixStruct = Ma;
  PiStruct = P;
 elseif labindex == 2
  [M,C,Ma,P]=BaumWelch(train{2},HS);
  MuStruct = M;
  CovStruct = C;
  MatrixStruct = Ma;
  PiStruct = P;
 elseif labindex == 3
  [M,C,Ma,P]=BaumWelch(train{3},HS);
  MuStruct = M;
  CovStruct = C;
  MatrixStruct = Ma;
  PiStruct = P;
 elseif labindex == 4
  [M,C,Ma,P]=BaumWelch(train{4},HS);
  MuStruct = M;
  CovStruct = C;
  MatrixStruct = Ma;
  PiStruct = P;
 elseif labindex == 5
  [M,C,Ma,P]=BaumWelch(train{5},HS);
  MuStruct = M;
  CovStruct = C;
  MatrixStruct = Ma;
  PiStruct = P;
 elseif labindex == 6
  [M,C,Ma,P]=BaumWelch(train{6},HS);
  MuStruct = M;
  CovStruct = C;
  MatrixStruct = Ma;
  PiStruct = P;
 end
end
```

Observing the reported code under Algorithm 5, the parallel worker allocation allows to execute the model inside each of the cores of the machine that it is being used to implement the Data Parallel approach. After the implementation of the partitions of data inside each of the workers, the gradient of the data is averaged to obtain the values of the mean, covariance, transition matrix, and initial probability matrix which corresponds to the training of the model. Consequently, the testing dataset is launched in parallel as presented in the Algorithm 6 code snippet. In this case, as the values of the training are average of the gradient values, the Viterbi Algorithm is launched in parallel to optimize the calculation of the most probable sequence.

**Algorithm 6 - Data Parallel: Viterbi Algorithm - Operations that can be executed in parallel**

```
%Compute the sequence of states
parfor i = 1:NumTrain
 [in,d,p]=Parviterbi(test{i},M,C,P,T)
 max_indStruct{i} = max_ind;
 deltaStruct{i} = delta;
 prbStruct{i} = prb;
end
```

## 3.4 GPU Implementation

Considering each of the previous configurations for deploying the machine learning model and obtain the best performance results, an analysis of the sources of inefficiency inside the code is performed to determine which parts of code can be improved *see section 5.4 for inefficiency analysis*. By determining these different sources of inefficiency inside the code, the GPU implementation of the algorithms is proposed to improve the overall performance of the proposed model.

The GPU Implementation takes place using the GPU Coder Application that the Matlab software provides to generate CUDA Code from a given Matlab source code. This configuration is used to optimize the parts of code that take the longest amount of time and power on execution. Therefore, this configuration is used to overcome the sources of inefficiency that the the previously mentioned algorithms have in its execution.

Using the previously mentioned application inside the Matlab software, different sections of the code are transformed to a basic representation in CUDA programming that allows the execution of these parts of code directly in the GPU. Using the different cores that are available inside the GPU (approximately 856 CUDA Cores in the used GPU) and the high bandwidth memory, the implementation of these parts of code are expected to take advantage of these new resources and obtain better performance results that improve the proposed HMM model for speech recognition.

# 4. EXPERIMENTATION

Using the previously mentioned methodology for each of the proposed configurations, different experiments are performed in each one using a common framework, which will allow us to determine which configuration is the most suitable to perform real-time analysis and speech recognition using HMM algorithms. The common model that it is used for experimentation is presented in Figure 1. This figure represents the four hidden states in the Markov Chain and the 58 different observations of Mel Frequency Cepstral Coefficients (MFCCs), which are used to characterize each of the emotions that were present inside the dataset of the studied people.

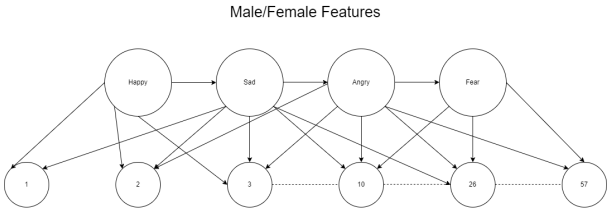


**Figure 1: Hidden Markov Model for speech recognition. Each hidden state represent a emotion that can be represented by a set of 58 different MFCCs**

Therefore, using the general model previously presented each of the configurations are tested on an equal framework to determine which has better performance in terms of total execution time, power, and accuracy. It is important to mention that the experimentation requires to perform both training and testing, which are the frameworks that are presented in the following subsections.

## 4.1 Training Framework

For the training framework of the experiment, the dataset was divided using cross-validation to obtain random sets of data and use it to predict the emotions on the speech of each subject. Therefore, the random permutation of the data was included in the experimental training framework to obtain the divided data. To perform a fair analysis using cross validation and the data and model parallel configurations, the permuted data varies from 1 to 6 training and testing datasets. By ensuring this division process, the training framework in the model and data parallel configurations can have 100% utilization of the hardware, which allows a fair comparison between the single and multi core configurations for this model.

After data is divided inside that specified range, the Baum Welch Algorithm is implemented using the training datasets as input observation inputs inside the code and also declaring the total number of states in the model (4 states which represents each emotion of the database). The obtained results of this algorithm is the mean, covariance, initial condition matrix, and transition matrix of the model, which are later used in the testing model to determine the most likely emotions that the different subjects have considering the measured observations.

## 4.2 Testing Framework

For the testing framework, the output values of the training algorithm is taken as input values for this algorithm as well as the testing dataset that it is going to be used to determine the emotions of the analyzed subjects. The mean of the normal distribution for each observation, the covariance, the initial condition matrix, and the transition matrix are used to perform the Viterbi Algorithm and obtain the values of the most

probable sequence. For this specific task, the experimentation framework also deals with the data that was previously partitioned using cross validation between 1 and 6 for maximum hardware utilization. After the execution of the testing framework, the values of total execution time, power, and accuracy are obtained for each configuration to determine under which one of them the model performs the best.

## 5. RESULTS

For this section, the overall performance results for each configuration is presented in terms of total execution time, total power, and accuracy. Consequently, this results are compared between each other to determine the which proposed configurations shows the best performance for Machine Learning acceleration using commercial hardware.

### 5.1 Single Core Configuration

For the Single Core Implementation of the proposed configuration, the results for the total execution time are presented in Figure 2 for the 6 different cross validation techniques that are implemented for the training and testing of the proposed Machine Learning Model.

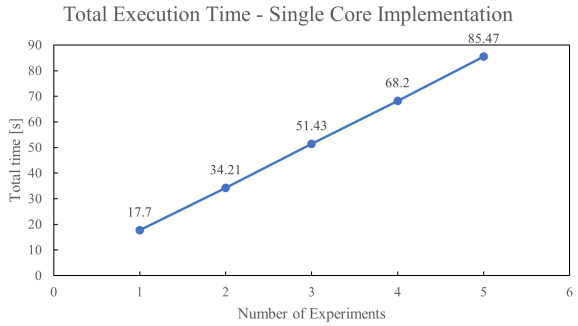


**Figure 2: Total Execution Time for the complete HMM Model on a Single Core Configuration**

Figure 2 represents how the proposed HMM model behaves using the cross validation technique for the training and testing of the HMM. Observing the figure, the overall execution time for the different cross validations is presented. In general, it can be seen that it is taking 85.47s for the complete model to execute when 5 cross validations are being performed for the training of the model. Considering the amount of data that it is being used for the implementation - 40 thousand data observations in 5 different cross validation models - it is not a good overall total execution time considering nowadays requirements for big data processing and prediction processes.

### 5.2 Model Parallel Configuration

For the Model Parallel Configuration, as it was previously mentioned in the Methodology section, both training and testing models are divided and executed inside the different cores (workers) that are available inside the Intel i5 processor.

Observing the results presented in Figure 3, the total execution time for the Model Parallel implementation of the proposed HMM model considerably decreases for the same

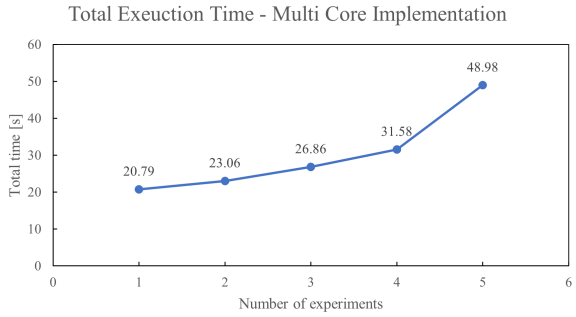


**Figure 3: Total Execution Time for the complete HMM Model on a Model Parallel Configuration**

number of cross validated data that it is used for the training method. For 5 cross validated datasets, the time reduces from 85.47s to 48.98s which is considerable better for big data and real-time machine learning scenarios.

| | Single Core | | | Multi Core | | |
|---|---|---|---|---|---|---|
| Experiments | Baum Welch [s] | Viterbi [s] | Overall [s] | Baum Welch [s] | Viterbi [s] | Overall [s] |
| 1 | 14.37 | 0.1765 | 17.7 | 3.58 | 0.0338 | 20.79 |
| 2 | 14.09 | 0.1482 | 34.21 | 3.915 | 0.0346 | 23.06 |
| 3 | 14.10 | 0.1408 | 51.43 | 4.17 | 0.0481 | 26.86 |
| 4 | 14.02 | 0.127675 | 68.2 | 4.70 | 0.0522 | 31.58 |
| 5 | 14.07 | 0.12962 | 85.47 | 4.55 | 0.04658 | 48.98 |

**Table 1: Total Execution Time comparison between Single core and Model Parallel configurations.**

The result reported in Table 1, represent the breakdown of execution times between the single core and model parallel configurations for the same amount of cross validation datasets that are implemented for the model. There is a clear improvement in the execution times for each algorithm in the studied cases. The following figure (Figure 4) summarizes this improvement.

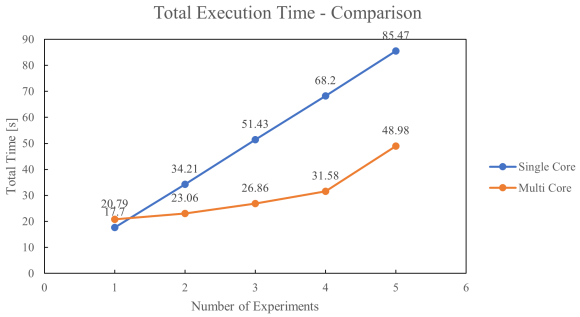


**Figure 4: Total Execution Time comparison between Single Core and Model Parallel Configurations**

### 5.3 Data Parallel Configuration

For the Data Parallel configuration, as it was previously mentioned in the Methodology section, it takes larger amounts of data and divided it to work on all the available workers (6 workers) in the CPU that it is being used. Therefore, this configuration is used to analyze bigger sets of data - with 80 thousand observations instead of 40 thousand - and allows

to deploy the single core models in each of the workers and analyze this data independently to later calculate the gradient of data for the testing portion of the model. The obtained results for this configuration is presented in the following figure.

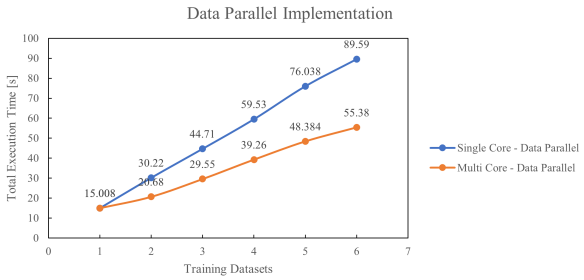


**Figure 5: Total Execution Time comparison between Single Core and Data Parallel Configurations**

Observing the reported results in Figure 5, it can be determined that the overall execution time of the data parallel configuration presents an improvement in comparison to the single core configuration for the same amount of training datasets. According to the obtained results, the total execution time for 6 training dataset with 80 thousand observations is 55.38s while for the single core implementation working only for 40 thousand observations is 89.59s, which shows a relevant decrease in the execution time and demonstrates how the data parallel configuration is able to work with larger amounts of data and achieve better performance for real-time machine learning application.

## 5.4 Sources of Inefficiency

Observing the different proposed configurations for execution of the parallel HMM algorithm, some bottlenecks and sources of inefficiency are found that cause an increase in time of the overall execution of the algorithm. These sources of inefficiency appear inside both single and multi core implementations (model parallel) that were studied in the previous sections of this paper. In addition, these sources of inefficiency cause that the overall energy consumption and CPU utilization are not optimal for the real-time machine learning framework that it is being proposed.

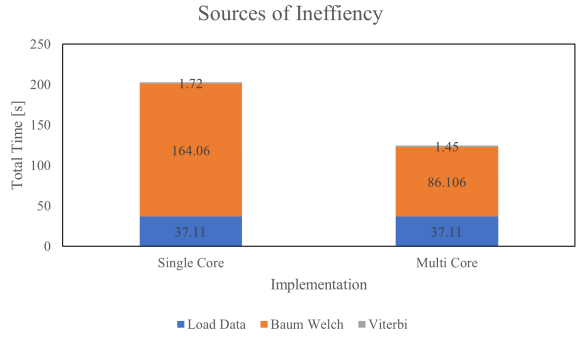


**Figure 6: Sources of Inefficiency inside the HMM algorithm execution**

Observing Figure 6, the results of the sources of inefficiency are presented in terms of total execution time. It is important to observe that the bottleneck in the execution of the system is the Baum-Welch algorithm used for training the model.The main reason for these inefficiencies in the design are the result of the dependencies that are inside the execution of the algorithm. These dependencies inside the calculations of the Baum-Welch algorithm cause that the code cannot be fully parallel in the model parallel approach, causing a reduction of the overall performance that can be obtained in the case of a fully parallel code. Therefore, these sources of inefficiency that cannot be fully parallel and that require extensive matrix multiplications, are tackled by creating the GPU CUDA code for the execution inside the GPU.

## 5.5 GPU Configuration

Finally, the GPU CUDA configuration is performed for extracting better performance out of the proposed HMM model. Therefore, using both GPUs - the NVIDIA GTX 1650 and the NVIDIA Tesla V100 - the CUDA code is executed and timed to observe the differences between the total execution times between the CPU and GPU implementations of the code and observing if the parts of the code that were inefficient improve in their execution.

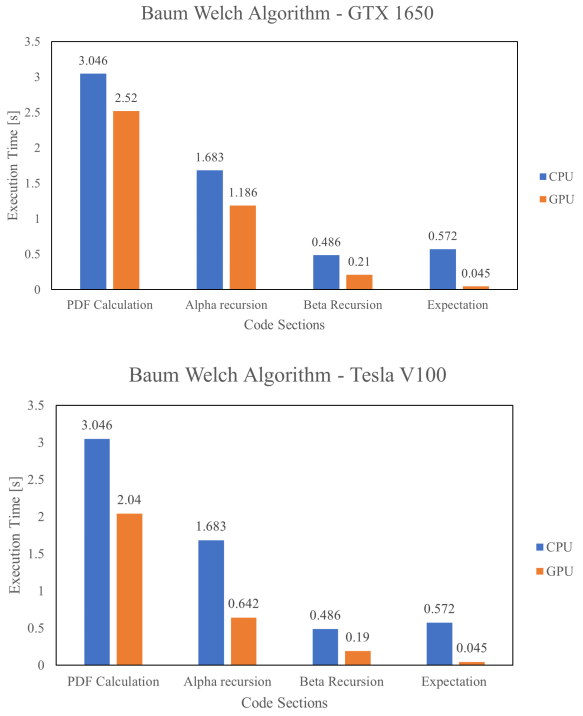


**Figure 7: Total Execution Time for GPU implementations in NVIDIA GTX 1650 (above) and NVIDIA Tesla V100 (below)**

Figure 7 presents the total execution time comparison for the identified operations that are inefficient inside the Baum-Welch Algorithm between the CPU and GPU configurations. Clearly, the GPU CUDA code allows better execution times for the cases in which the the matrix multiplications take

longer times, which are the PDF calculation, alpha and beta recursion, and the expectation calculation. In addition, the results show that the NVIDIA Tesla V100 GPU has better performance results than the NVIDIA GTX 1650, which shows that better execution and performance can be obtained out of this high performance NVIDIA GPU (for further explanation about double-precision matrix multiplication please refer to ANNEXES section Figure 10-11).

## 5.6 Energy, Power and CPU Utilization

Finally, the energy, power and CPU utilization are analyzed for each of the implemented configurations inside this study. The results presented in Figure 8 show the total energy and power consumed by each of the configurations as a function of the elapsed time that takes the complete execution of the algorithm inside each configuration.

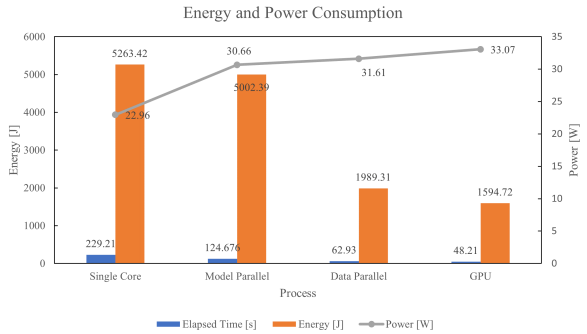


**Figure 8: Total Energy and Power Consumption comparison between the different configurations**

Therefore, it can be determined that the data parallel and GPU configurations show better performance results in the three parameters that are presented in the previous figure. This shows that these two implementations have the optimal results for a real-time analysis of data inside the HMM algorithm for speech recognition. It is important to mention that the presented power is calculated by dividing the total energy by the elapsed time that takes the algorithm to execute inside each configuration. Therefore, power consumption is going to be higher for the data parallel and GPU configurations given that these have less execution time.

| Process | Elapsed Time [s] | Energy [J] | Power [W] |
|---|---|---|---|
| Single Core | 229.21 | 5263.42 | 22.96 |
| Model Parallel | 124.676 | 5002.39 | 30.66 |
| Data Parallel | 62.93 | 1989.31 | 31.61 |
| GPU | 48.21 | 1594.72 | 33.07 |

**Table 2: Total Energy and Power Consumption of the different proposed configurations**

In addition, the total CPU utilization is obtained by using the Intel Power Gadget that it is available for the characterization of all Intel's processors in the market. The results for the CPU utilization are presented in the following figure.

Considering all the previously mentioned configurations for parallel execution of the HMM algorithm, the CPU utilization is presented for each of these configurations as a function

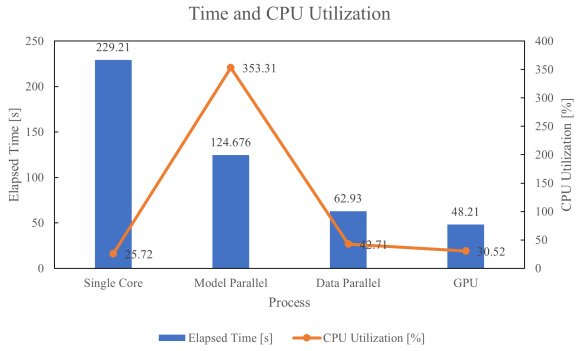


**Figure 9: Percentage of CPU utilization as a function of total execution time for all the proposed configurations**

of the total elapsed time that it is required for the complete execution of the algorithm to complete. The reported results show that the GPU and Data Parallel configurations show good CPU utilization percentages, while the Model Parallel configuration shows a higher utilization considering the less time that it takes for it to execute all the algorithm. For the case of the Single Core, the CPU utilization is clearly lower than the others and it is the one that takes the most time to implement.

| Process | Accuracy [%] |
|---|---|
| Single core | 92.3 |
| Model Parallel | 93.8 |
| Data Parallel | 97.3 |
| GPU | 97.8 |

**Table 3: Percentage of Accuracy for the different proposed configurations**

Considering all the previously mentioned configurations, the percentage of accuracy of the proposed algorithm is measured for each of the configurations, and can be obtained that the GPU and Model Parallel configurations are the ones that have the highest levels of accuracy, which is an expected behaviour given that these two cases are using more data than the others.

# 6. DISCUSSION

The proposed implementations for a parallel HMM demonstrated an overall improvement in the classical one core sequential programming approach. Observing the used and proposed methodology for the implementations, we can observe that there are some limitations in the overall process of creating a parallel implementation of the algorithm. The main reason for these limitations is the dependence between the past and current values for the calculation of the future values inside the alpha and beta recursions in the expectation maximization process of the Baum Welch Algorithm for the training process of the problem. Therefore, these dependencies between values cause a limitation for the code to be complete deployed in parallel and, as a result of this low parallel code, the complete performance enhancement

technique cannot be fully exploited.

Nevertheless, the results presented an overall improvement for the parallel configurations (model and data parallel) in comparison to the classical single core implementation that it is used in most of the proposed cases right now. In general, the Multi Core configuration shows an execution 2.1x times faster than the Single Core configuration considering total execution time. By breaking down into the algorithm execution, the Data Parallel configuration is 2.5x times faster than the Model Parallel configuration while executing the Baum-Welch Algorithm and 4.2x times faster than the Single Core configuration under the same algorithm. Despite the fact that there are good improvements in terms of total execution time for the comparison between Multi Core and Single Core configurations, these improvements are still limited by the dependencies that there exist inside the code that does not allow a fully parallel configuration in the case of the Model Parallel configuration. If the configuration was fully parallel, the communication overheads will be bigger, causing an increase of execution time.

For the GPU configuration, the results clearly show that the CUDA implementations of the sources of inefficiency allows to exploit better levels of parallelism inside the code and also faster matrix-matrix multiplication operations, which is the most important operation that has to be accelerated to obtain the appropriate results. Using the NVIDIA GeForce GTX 1650 GPU, the Baum-Welch Algorithm is 1.46x times faster than the Multi Core Configuration. On the other hand, the NVIDIA Tesla V100 GPU is 2.01x times faster than the Multi Core configuration, which clearly show how different GPUs are able to provide better levels of efficiency between the operations and how they can be used to increase performance of a Machine Learning Algorithm execution.

Finally, an analysis about energy, power, CPU Utilization, and accuracy is performed for all the previously proposed configurations. In terms of energy, the Data Parallel configuration is 2.52x times more energy efficient than the Model Parallel configuration, and 2.7x times energy efficient than the Single Core configuration. Considering the GPU configuration, this is the one that presents the most important improvements in terms of energy efficiency, given that it is 3.3x times more energy efficient than the Single Core configuration and 3.1x times more energy efficient than the Multi Core configuration. In terms of CPU utilization, the Model Parallel configuration presents a better usage of the CPU, given that is the one that maximizes utilization with having a high execution time. Finally, in terms of accuracy in the speech recognition process, the GPU and Data Parallel configurations presents the best results, showing accuracy levels up to 97.8% and 97.3% respectively.

## 7. CONCLUSIONS

This research shows an extensive study of the different configurations that can be used for the implementation of HMMs in commercial hardware to accelerate the overall execution and obtain better levels of performance for this algorithm in the area of speech recognition. HMM algorithm is one of the most used algorithms for speech recognition and Machine Learning and it has been constantly being adapted to work for a Big Data perspective and in the area of real-time Machine Learning. Therefore, this study shows a promising approach to optimize the implementation of this algorithm for speech recognition and adapt it for a parallel execution that allows the algorithm take better advantage of the current available hardware and resources to execute faster and with higher levels of precision and accuracy. In addition, this study proposes a Big Data implementation of the HMM algorithm, which shows how this algorithm can be modified and adapted for an execution in parallel CPU and GPU that allows larger amounts of data to be processed at the same time, resulting on an improvement for the overall execution of the algorithm.

Considering all the previously presented results for the different proposed configurations, we can observe that there are good improvements by comparing the Multi Core (Model and Data Parallel) configurations with the Single Core configuration. In general, the Multi Core configuration presents and speedup of 2.1x times in execution time over the Single Core implementation with 2.52x times better energy efficiency. On the other hand, the GPU implementation showed overall speedups of up to 2.01x times faster than the Multi Core configuration with a energy efficiency 3.1x times better than its CPU counterpart. In terms of efficiency, it can be concluded that the Data Parallel and GPU configurations show better results given that they manipulate larger amounts of data than the Model Parallel and Single Core configurations.

Finally, it can be concluded that this study shows a how the Data Parallel and GPU configurations for implementing the HMM algorithm for speech recognition can be scaled to execute in a Big Data approach and can obtain better levels of precision and accuracy for this specific case. Therefore, this scalability in the design also allows to determine other applications of the model for different research and industry areas. As a future goal of this research, the design and creation of a novel custom architecture for Machine Learning and HMMs can be created using hardware design.

## ACKNOWLEDGMENTS

This study was successful thanks to the ECE 9413 Parallel and Customized Computer Architectures course and lessons, which were used to learn more about the topics of machine learning acceleration and custom architectures for high performance computing.

## REFERENCES

[1] E. Alpaydin, *Introduction to Machine Learning*, 3rd ed. Cambridge, Massachusetts: MIT Press, 2014.

[2] T. Be-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," in *Department of Computer Science Zurich University*, 2018.

[3] C. Bishop, *Pattern Recognition and Machine Learning*, 2nd ed. Cambridge, UK: Sringer, 2006.

[4] Y. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *43th International Symposium on Computer Architecture (ISCA)*, 2016.

[5] G. Grimmet and D. Stirzaker, *Probability and Random Processes*, 3rd ed. Oxford, UK: Oxford University Press, 2001.

[6] U. Gupta, B. Reagen, L. Pentecost, M. Donato, T. Tambe, A. Rush, G. Yeon, and D. Brooks, "Masr: A modular accelerator for sparse rnns," in *International Conference on Parallel Architectures and Compilation Techniques*, 2019.

[7] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. Horowitz, and W. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *ACM SIGARCH Computer Architecture News*, 2016.

[8] V. Hegde and S. Usmani, "Parallel and distributed deep learning," in *Stanford University*, 2016.

[9] U. Holzle, "Mlperf benchmark establishes that google cloud offers the most accessible scale for machine learning training," in *Google Cloud Newsletter*, 2018.

[10] N. Jouppi and et al., "In-datacenter performance analysis of a tensor processing unit," in *44th International Symposium on Computer Architecture (ISCA)*, 2017.

[11] J. Li, S. Chen, and Y. Li, "The fast evaluation of hidden markov models on gpu," in *IEEE International Conference on Intelligent Computing and Intelligent Systems*, 2009.

[12] C. Liu, "cuhmm: a cuda implementation of hidden markov model training and classification," in *Available: http://liuchuan.org/pub/cuHMM.pdf*, 2006.

[13] I. McDonald, W. Zucchini, and R. Langrock, *Hidden Markov models for time series: an introduction using R*, 1st ed. Oxford, UK: CRC Press, 2017.

[14] A. Roytel, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Survey of machine learning accelerators," in *MIT Lincoln Laboratory Supercomputing Center*, 2009.

[15] I. Sassi, S. Anter, and A. Bekkhoucha, *An Overview of Big Data and Machine Learning Paradigms*, 1st ed. Cambridge, UK: Sringer, 2018.

[16] I. Sassi, S. Anter, and A. Bekkhoucha, "Paradist-hmm: A parallel distributed implementation of hidden markov model for big data analytics using spark," in *(IJACSA) International Journal of Advanced Computer Science and Applications*, 2021.

[17] L. Welch, "Hidden markov models and the baum-welch algorithm," in *IEEE Inform Theory Soc Newsletter*, 2003.

[18] D. Westhead and M. Vijayabaskar, *Hidden Markov Models Method and Protocols*, 1st ed. New York, New York: Springer Science and Bussiness Media, 2017.

[19] L. Yu, Y. Ukidave, and D. Kaeli, "Gpu-accelerated hmm for speech recognition," in *43rd International Conference on Parallel Processing Workshops*, 2014.

[20] H. Zheng, "Data parallel deep learning," in *Data Science Group at ALFC*, 2019.

# ANNEXES

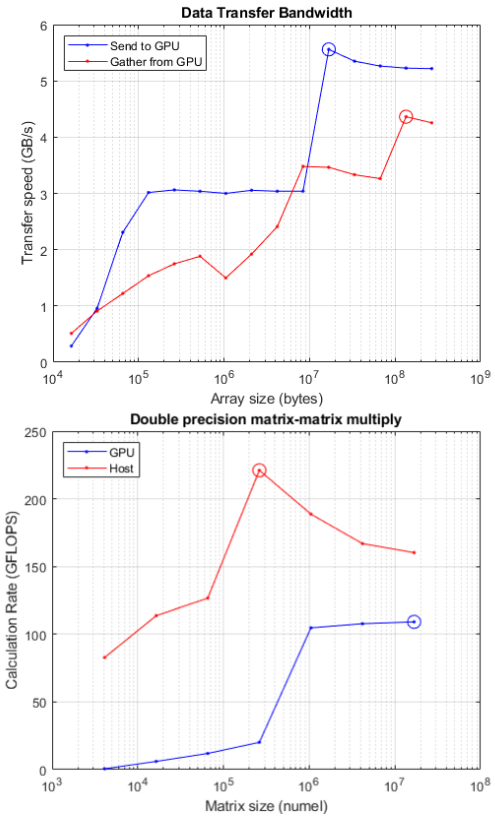


**Figure 10: NVIDIA GeForce GTX 1650 Data Transfer Rate and Double-Precision Matrix Multiplication Characterization**

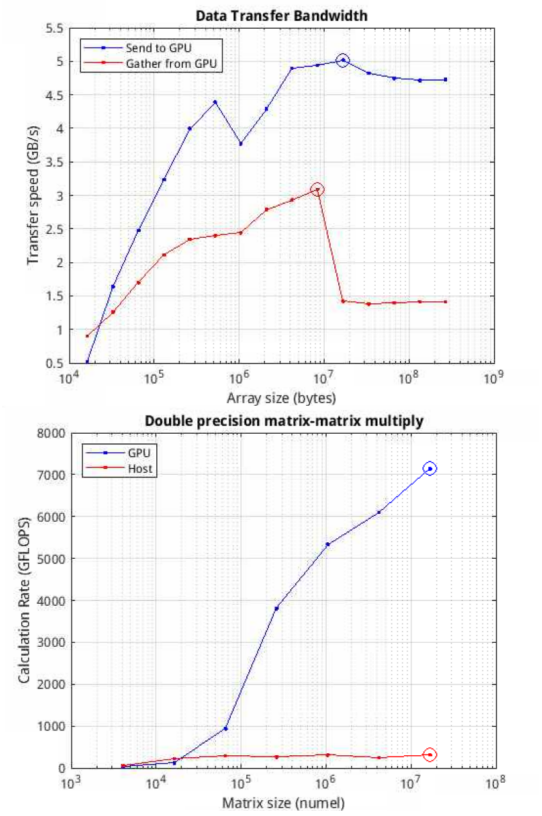


**Figure 11: NVIDIA Tesla V100 Data Transfer Rate and Double-Precision Matrix Multiplication Characterization**

| | Double precision results (in GFLOPS) | | | Single precision results (in GFLOPS) | | |
|---|---|---|---|---|---|---|
| | MTimes | Backslash | FFT | MTimes | Backslash | FFT |
| Tesla V100-PCIE-32GB | 6884.95 | 563.73 | 728.71 | 13727.99 | 1210.42 | 1365.11 |
| TITAN V | 6779.73 | 674.40 | 534.65 | 13515.42 | 1336.39 | 985.36 |
| Tesla P100-PCIE-12GB | 4510.03 | 929.00 | 357.65 | 8435.34 | 1647.83 | 687.13 |
| Tesla K40c | 1189.54 | 677.12 | 135.88 | 3187.76 | 1334.17 | 294.86 |
| Tesla K20c | 1004.06 | 641.42 | 106.09 | 2657.01 | 1230.28 | 235.20 |
| TITAN Xp | 421.00 | 369.32 | 209.45 | 10823.05 | 1272.06 | 797.17 |
| GeForce RTX 2080 SUPER | 373.37 | 345.32 | 164.30 | 10813.12 | 1330.64 | 746.20 |
| GeForce GTX 1080 | 280.84 | 223.05 | 137.66 | 7707.01 | 399.37 | 424.60 |
| **Your GPU** (NVIDIA GeForce GTX 1650) | **107.35** | **100.82** | **43.55** | **3062.80** | **1180.31** | **206.10** |
| **Your CPU** | **208.78** | **111.35** | **15.03** | **523.79** | **249.00** | **45.61** |
| Quadro K620 | 25.45 | 22.77 | 12.75 | 716.71 | 350.31 | 75.00 |

(Sort the results by clicking on any column title. To see detailed performance data, click on an individual result or a device name.)

**Figure 12: GPU Characterization performed in Matlab for to observe difference between execution times in computer expensive algorithms**

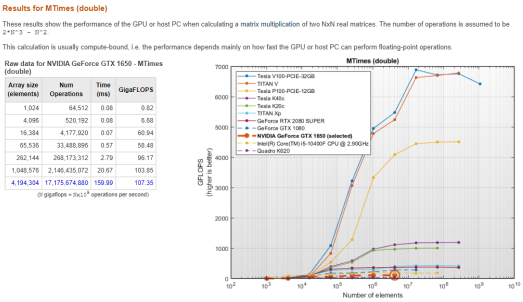


**Figure 13: GPU Characterization: Double Precision Matrix multiplication for different GPUs**

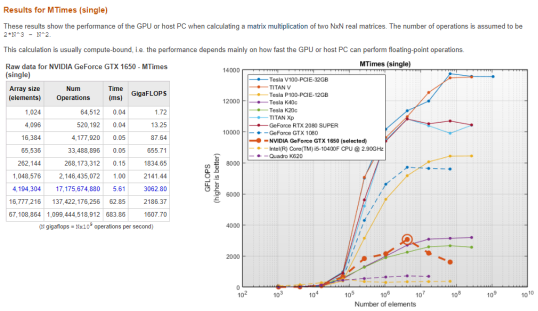


**Figure 14: GPU Characterization: Single Precision Matrix multiplication for different GPUs**