Gradient-Boosted Generalised Additive Models with GPU Acceleration

Antoni Sieminski
a.m.sieminski@ed.ac.uk

Johnny MyungWon Lee
johnny.myungwon.lee@ed.ac.uk

7th April 2023

1. Introduction

In this paper, we advance the computation of gradient-boosted generalised additive models (GAMs) with ℓ_2 penalty through graphics processing unit (GPU). Computational bottlenecks are addressed by parallelisng the most expensive parts of the algorithm, that is matrix-vector multiplications for each base learner and boosting iteration.

Gradient boosting involves creating an ensemble model by iteratively fitting "weak" estimators known as base learners to the model's current errors. These estimators can have the form of individual penalised GAM effects resulting in an interpretable model with enough complexity to accurately predict future data. The bottom-up nature of the gradient boosting also increases interpretability. Irrelevant base learners do not make it into the final model, because the fitting process is usually stopped before they are selected, thus resulting in a sparse final model.

However, this does not stop these base learners from being considered, thus still affecting the computational complexity of the algorithm. The base learners in this framework are, in general, computationally expensive as every covariate with a nonlinear effect is fitted using a multi-column basis matrix. One of the main parts of GAMs is matrix-vector multiplication – a fundamental operation in linear algebra and has numerous applications in fields such as physics, engineering, and computer science. It involves multiplying a matrix by a vector to produce a new vector.

GPUs are a popular choice of processor unit to accelerate expensive calculation. They are highly useful particularly for matrix-vector multiplications due to their ability to perform many parallel computations (Sørensen, 2012). Hence, our starting point for GPU acceleration is the following idea from (Bainville, 2010) which benchmarks the `gemv` algorithm from `BLAS` package (Blackford et al., 2002). We can break down the computation into tasks and work on parallel threads when performing matrix-vector multiplication on a GPU. This leads to a significant speedup compared to performing the same operations on a CPU (Central Processing Unit). With the current development of CPUs, it achieves similar speed with GPUs in lower-dimensional settings. However, with the use of GPUs, we can achieve significant speedup for large-dimension matrix-vector multiplications.

In Chapter 2, we introduce the preliminary materials that link to the idea of gradient boosting, GAMs and the ℓ_2 penalty – a special case of penalised likelihood. In Chapter

3, we describe the algorithms of gradient boosting and the multi-threaded matrix-vector multiplication for GPUs. Chapter 4 explains our simulation study on the described algorithms and compares their performance under different conditions. Shortcomings and future directions are described in the Conclusion.

2. Preliminary

2.1 Generalised Additive Models

Generalised Additive Models (GAMs) are an extension of linear and generalised linear models (GLMs), which allow for fitting complex non-linear relationships between independent predictors $\boldsymbol{x}_j$ and the random response variable $\boldsymbol{y}$, which comes from a Gaussian distribution $\mathcal{N}$ with mean $\boldsymbol{\mu}$, and variance σ^2.

The resulting model is typically of the form:

$$\boldsymbol{y} \sim \mathcal{N}(\boldsymbol{\mu}, \sigma^2 I_n) \tag{1}$$
$$\boldsymbol{\mu} = \beta_0 + f_1(\boldsymbol{x}_1) + \ldots + f_p(\boldsymbol{x}_p) \tag{2}$$

where each function f_j depends on one or more variables in $\boldsymbol{x}_j$, thus allowing for relatively easy interpretation of partial effects on the mean $\boldsymbol{\mu}$.

2.1.1 Smoothing splines

In GAMs, each smooth non-linear function f_j can be represented as a linear combination of rank-reduced k basis functions $b_i(.)$ multiplied by their fitted coefficients γ_i, such that

$$f_j(\boldsymbol{x}_j) = \sum_{i=1}^{k} b_i(\boldsymbol{x}_j)\gamma_i = \mathbf{B}_j\boldsymbol{\gamma}_j \tag{3}$$

where $\mathbf{B}_j$ is the b-spline basis matrix created from the predictor variable $\boldsymbol{x}_j$. Penalisation terms $\lambda_j\boldsymbol{\gamma}_j^T\mathbf{S}_j\boldsymbol{\gamma}_j$ are typically added to the loss function for each term f_j, where $\mathbf{S}_j$ is the smoothing matrix defining the penalty type, and the penalty parameter λ_j controls its influence over the final fit of the coefficients $\boldsymbol{\gamma}_j$. These steps are taken to avoid overfitting and ensure smoothness of the function f_j. For this study, we used a simple ℓ_2 penalty, which sets $\mathbf{S}_j = \mathbf{I}_j$, thus resulting in $||\boldsymbol{\gamma}_j||^2$ penalty.

However, there are many other possible penalties to choose from. For example, $\mathbf{S}_j$ can be a cross-product $\mathbf{D}_j^T\mathbf{D}_j$ of two second-order difference matrices $\mathbf{D}_j$ resulting in a P-Spline penalty $\sum_{i=2}^{k-1}(\gamma_{i-1} - 2\gamma_i + \gamma_{i+1})^2$ (Eilers and Marx, 1996), which penalises the wiggliness of the function f_j.

2.1.2 Effective degrees of freedom

Effective degrees of freedom (edf) of penalised regression models can be defined as the trace of the symmetric hat matrix $\mathbf{H}$ – the matrix mapping the response vector $\boldsymbol{y}$ to the predicted values $\boldsymbol{\mu}$, such that $\mathbf{H}\boldsymbol{y} = \boldsymbol{\mu}$. Setting $\mathrm{Tr}(\mathbf{H})$ to some value ω (e.g., $\omega := 1$) allows for finding the penalty parameter λ constraining the model fit and is an important part of gradient boosting in the GAM setting.

2.2 Gradient Boosting

GAMs of the form described in (1) can be fitted in a wide variety of ways, which allow them to select the right form of the partial predictors f_i. One of such methods is Gradient boosting (Tutz and Binder, 2006), which is an ensemble learning method allowing many "weak" estimators $h^{[i]}(\mathcal{X})$ to create a complex model $F^{[m]}(\mathcal{X}) = \sum_{i=0}^{m} h^{[i]}(\mathcal{X})$, where $\mathcal{X} = \{\boldsymbol{x}_j : j \in 1, \ldots, K\}$ is just a set of all predictors $\boldsymbol{x}_j$. These estimators known as *base learners* aim to correct their predecessors' errors defined as the gradient of the loss function $L(\boldsymbol{y}, \boldsymbol{\mu})$.

In the Gaussian distribution, $L(\boldsymbol{y}, \boldsymbol{\mu})$ can be defined as the residual sum of squares $||\boldsymbol{y} - \boldsymbol{\mu}||^2$ and the linear predictor at iteration m as $F^{[m]}(\mathcal{X}) = \boldsymbol{\mu}^{[m]}$. Taking the gradient of the latter, results in values proportional to the vector of raw residuals ϵ:

$$\frac{\partial L(\boldsymbol{y}, F^{[m]}(\mathcal{X}))}{\partial F^{[m]}(\mathcal{X})} = 2(\boldsymbol{y} - \boldsymbol{\mu}^{[m]}) \propto \boldsymbol{\epsilon}^{[m]}$$

Gradient boosting can also be viewed as a functional version of the gradient descent algorithm. One of the main differences, though, is that gradient boosting uses early stopping to induce shrinkage and reduce the generalisation error of the model. The gradient boosted GAM algorithm is further explained in section 3.1.

2.2.1 Base learners

In gradient boosting, any estimator can be a base learner. However, there are certain properties that make these estimators more or less desirable for use.

Base learners require high training-data bias – i.e. be "weak" – so that they don't overshoot the minimum during the boosting process. Furthermore, this bias should be the same for all base learners, so that no base learner gets preferential treatment over the other ones (Hofner et al., 2014). Functional gradient descent is responsible for selecting the optimal functional form from the given base learner options and should, therefore, not select any of the learners over others based on arbitrary factors.

2.2.2 Splines as base learners

In order to create an interpretable model as displayed in (1), the base learners should follow its structure, which means that none of them should be able to fit an interaction between variables from vectors $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$, where $i \neq j$. Furthermore, if $f_j(\boldsymbol{x}_j)$ is to be a smooth function, then so should be the base learners composing it.

Base learners also need to be fast to fit and have a high training-data bias. In the case of splines, this is achieved by fitting them via penalised likelihood as described in 2.1.2. The smoothing parameter λ is computed by setting the number of degrees of freedom for each base learners to some arbitrary value, such as $\omega := 1$ (Hofner et al., 2014) and is done only once during the initialisation stage. Note that despite fixing λ to some relatively high value, gradient boosting can still fit arbitrarily wiggly functions due to the residual updating scheme inherent in gradient boosting.

3. Algorithm

3.1 Gradient Boosting of Generalised Additive Model

3.1.1 Initialisation

Given a set of continuous predictors $\boldsymbol{x}_j \in \mathcal{X}$ and a response vector $\boldsymbol{y}$, create a set of basis matrices $\mathbf{B}$. The basis matrices $\mathbf{B}_j$ are the design matrices of ℓ_2-penalised regressors, which form the model's set of base learners. In order to fit them, each regressor j needs to have a smoothing parameter λ_j, which sets the effective degrees of freedom (edf) of each base learner to a preset number $\omega \in \mathbb{R}^+$. We use the edf = $\mathrm{Tr}(\mathbf{H})$ formulation, where $\mathbf{H} = \mathbf{B}(\mathbf{B}^T\mathbf{B} + \lambda\mathbf{I})^{-1}\mathbf{B}^T$. For computational reasons, we also precompute $(\mathbf{B}_j^T\mathbf{B}_j + \lambda_j\mathbf{I})^{-1}$ for each base learner j and store the result in local memory. The detailed equation of the algorithm for edf is shown in Appendix A.

Independently to the above steps, the offset term β_0 from (1) is fitted to the response $\boldsymbol{y}$ by simply taking its average $\bar{\boldsymbol{y}}$ and becomes the linear predictor $\boldsymbol{\mu}$ at iteration $m = 0$.

Data: $\boldsymbol{y}, \mathbf{B}, \omega$
for i *in* $[1, \ldots, K]$ **do**
  $\lambda_i \leftarrow \underset{\lambda}{\text{uniroot}}\,(edf[\mathbf{B}_i, \lambda] - \omega)$;
  $\text{learner}_i \leftarrow \{\mathbf{B}_i; (\mathbf{B}_i^T\mathbf{B}_i + \lambda_i\mathbf{I})^{-1}\}$;
end
$\boldsymbol{\mu}^{[0]} \leftarrow \bar{\boldsymbol{y}}$;

Algorithm 1: Initialise Gradient Boosted GAM

3.1.2 Iteration for Gradient Boosted GAM

After the initialisation process, the fitting algorithm proceeds as follows: the gradient vector is computed from the previous iteration's predictions $\boldsymbol{\mu}^{[m-1]}$. Then, each of the K learners is fitted to this gradient $\mathbf{g}^{[m]}$ and the one with the lowest residual sum of squares (RSS) is selected. The selected learner i has its fitted coefficients shrunk by learning rate ν yielding $\boldsymbol{\delta}_i^{[m]}$, which is then added to the final model's vector of coefficients $\boldsymbol{\gamma}_i$ for the base learner. The prediction vector $\boldsymbol{\mu}$ is updated accordingly, and the process starts all over again until $m = m_{stop}$.

Data: $\boldsymbol{\mu}^{[0]}, \boldsymbol{y}, \{\text{learner}_i : i \in 1, \ldots, K\}, m_{stop}, \nu$
for m *in* $[1..m_{stop}]$ **do**
  $\mathbf{g}^{[m]} \leftarrow \boldsymbol{y} - \boldsymbol{\mu}^{[m-1]}$
  $i \leftarrow \arg\min_i \mathrm{RSS}(\text{learner}_i, \mathbf{g}^{[m]})$
  $\boldsymbol{\delta}_i^{[m]} \leftarrow \nu \cdot \mathrm{MatVecMul}(\mathrm{MatVecMul}(\mathbf{B}_i^T\mathbf{B}_i + \lambda_i\mathbf{I})^{-1}, \mathrm{MatVecMul}(\mathbf{B}_i^T, \mathbf{g}^{[m]}))$
  $\boldsymbol{\gamma}_i^{[m]} \leftarrow \boldsymbol{\gamma}_i^{[m-1]} + \boldsymbol{\delta}_i^{[m]}$
  $\boldsymbol{\mu}^{[m]} \leftarrow \boldsymbol{\mu}^{[m-1]} + X_i\boldsymbol{\delta}_i^{[m]}$
end

Algorithm 2: Iterate Gradient Boosted GAM

3.2 Matrix-Vector Multiplication

3.2.1 Naive Implementation

Let us define a matrix $\mathbf{A}$ and a vector $\boldsymbol{x}$ with dimensions of $(M \times N)$ and $(N \times 1)$ respectively. The resulting product of the matrix and the vector has dimension of $(M \times 1)$ and we denote this vector as $\boldsymbol{y}$ where $y_i = \sum_j A_{ij} x_j$. The naive implementation of matrix-vector multiplication is presented in (3).

Data: $\mathbf{A}, \boldsymbol{x}, \boldsymbol{y}$
for i *in* $[1, \ldots, M]$ **do**
  $\boldsymbol{y}[i] \leftarrow 0$;
  for j *in* $[1, \ldots, N]$ **do**
    $\boldsymbol{y}[i] += \mathbf{A}[i][j] * \boldsymbol{x}[j]$;
  end
end

Algorithm 3: Naive Implementation of Matrix-Vector Multiplication

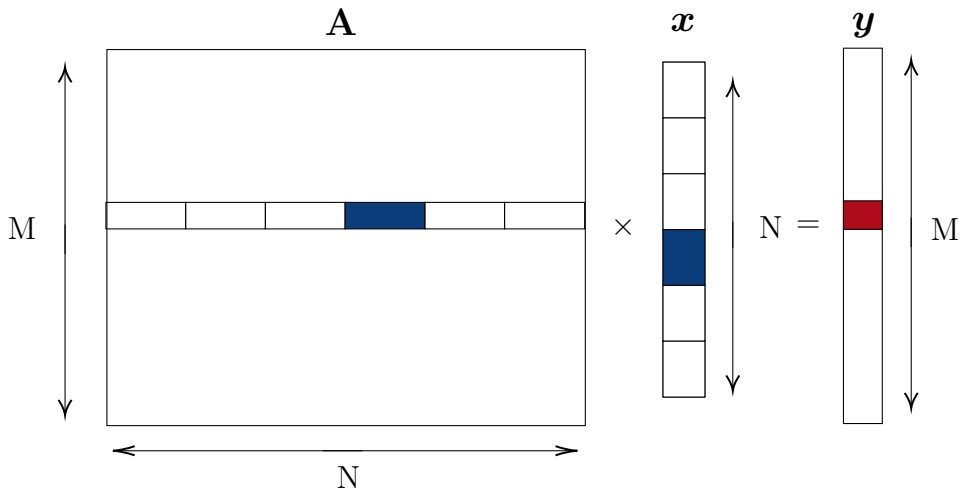

Figure 1: Matrix-Vector Multiplication in GPU

Fig. 1, illustrates the implementation for GPU where the thread identifiers in the kernel from the above algorithm are shown (Appendix B). Each thread takes one row of matrix $\mathbf{A}$ and vector $\boldsymbol{x}$ from global memory, stores the dot product back to vector $\boldsymbol{y}$ in the global memory. Here, P threads were employed and each thread produces $\boldsymbol{y}_i$ by computing the dot product between $\mathbf{A}[i, 1:N]$ and $\boldsymbol{y}$.

This implementation has its advantage in simplicity but it poses a major disadvantage where the inner loop requires N multiplications resulting in $O(n)$ time complexity assuming $P > M$. Also, its performance speed in the CPU is limited up to a certain dimension of the input matrix $\mathbf{A}$. As a result, we will take a step further and introduce two different versions of matrix-vector multiplication inspired from (Bainville, 2010) that could potentially outperform the naive implementation by full exploitation of the parallel threads of GPU.

3.2.2 P Threads per Dot Product

Naive Implementation (3) computed one thread per dot product, instead computation is done through out P threads in a GPU. Under the 2D blocks setting, we form a $(M' \times P)$ threads as shown in Fig. 2. Hence, the mathematical expression below represents the total work done by the threads.

$$\sum_{i \in M'} \sum_{j \leq p} A_{ij} x_j + \sum_{i \in M'} \sum_{p < j \leq 2p} A_{ij} x_j + \cdots + \sum_{i \in M'} \sum_{P-p < j \leq P} A_{ij} x_j \tag{4}$$

The two long dark blue coloured segments are computed and stored in a work group of size $(M' \times P)$. For each row, the P threads communicate and reduces the working group by computing the sum of their respective contributions to y_i that is shown in the red box. Thus, we are taking the advantage where the same work group share some of the local memory.

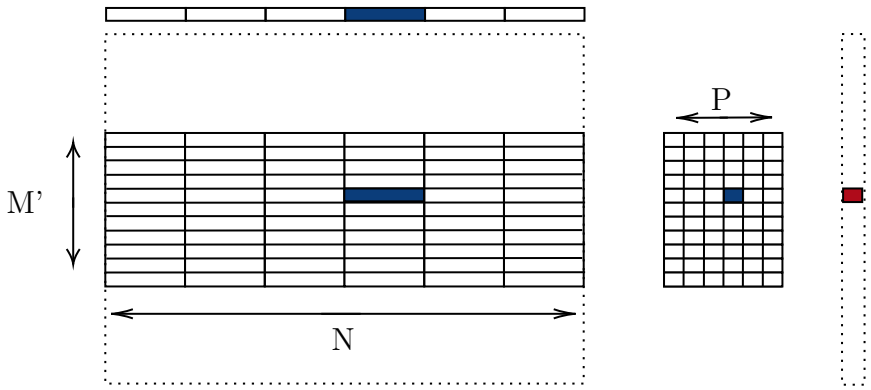

Figure 2: P Threads per Dot Product

3.2.3 P Threads per Row

We now run P threads per row, where each thread preloads the corresponding segment of vector $\boldsymbol{x}$ in a local memory, to make it available to all threads of the group. The result is stored in global memory in an $(M \times P)$ matrix. Here we denote $\mathbf{A}'$ as a working group sub-block of matrix $\mathbf{A}$. Then we have,

$$\sum_{j=1}^{P} \sum_{i=1}^{M'} \mathbf{A}'_i x_j$$

$$\text{with } \boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_p)^T, \quad \mathbf{A} = \begin{pmatrix} \mathbf{A}_{11} & \cdots & \mathbf{A}_{1p} \\ \vdots & \ddots & \vdots \\ \mathbf{A}_{M1} & \cdots & \mathbf{A}_{Mp} \end{pmatrix}$$

After the computation of the partial dot products, we reduce this matrix by column with a separate kernel. The cost of the reduction is negligible when P is small compared to the computation of all partial sums thus, this gives an advantage of diving the kernel into two parts. Looking at Fig. 3, a working group (in the middle) is defined to have the size

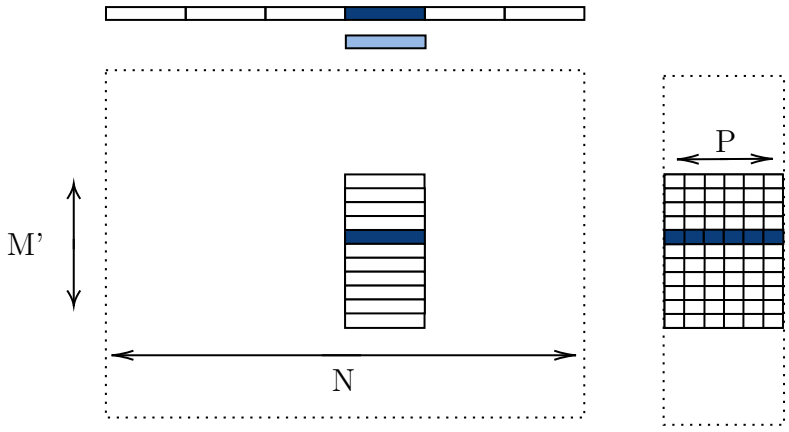

Figure 3: P Threads per Row

of $(M' \times 1)$ threads. Each group copies a segment of $\boldsymbol{x}$ in local memory (in light blue), then each thread computes its partial dot product (in dark blue). The result is stored in a cell of $\boldsymbol{y}$ (in dark blue on the right), which has been extended to P columns. The columns of $\boldsymbol{y}$ are added together using a second kernel that does addition.

Now, the requirement in local memory for each working group is small (either $\lfloor N/p \rfloor$ or $\lceil N/p \rceil$ elements), allowing more works to be done simultaneously in each thread. Another advantage is that we can maximise the reuse of the portions of $\boldsymbol{x}$ copied to local memory. This is because the working group width is now one thread and we do not need all threads affected to the same row to communicate thus assigning the total work group capacity to the computation of rows.

4. Simulation

In this chapter, we illustrate the simulation conducted with `pyopencl` under `[<pyopencl.Device 'NVIDIA GeForce MX550' on 'NVIDIA CUDA'>]` for the GPU computation and `numpy.dot()` under `12th Gen Intel(R) Core(TM) i7-1265U 1.80 GHz` for the CPU computation. We then present the results from the simulation and discuss its result. Note that the performance timings do not include transfer of data between the host and GPU and the computation was done in single precision.

4.1 Generating data

Each explanatory variable $\boldsymbol{x}_j$ with length of $\{10^3, 10^4, 10^5\}$ was sampled from an independent and identical uniform distribution and, in total, collected 100 explanatory variables. We randomly selected 20 explanatory variables from $\mathcal{X}$, $\{\boldsymbol{x}_j : j \in G | G \subset [1..100] | \#G = 20\}$. They have an additive effect of $10\sin(2\pi x)$ on the mean of the response $\boldsymbol{y}$, and the rest had no effect on $\boldsymbol{y}$. An intercept term $\beta_0 = 7$. The response $\boldsymbol{y}$ was generated with a low amount of random noise drawn from the normal distribution

with $\sigma^2 = 10^{-3}$.

$$y_i \sim N(\mu_i, 10^{-3}), \quad x_{ji} \sim U(0,1)$$
$$\boldsymbol{\mu} = 7 + \sum_{j \in G} f_j(\boldsymbol{x}_j), \quad f_j(\boldsymbol{x}_j) = 10\sin(2\pi \boldsymbol{x}_j)$$

After generating the simulated dataset, we used the `patsy` package in Python to create B-spline basis matrices $B_j(\boldsymbol{x}_j)$ with equidistant knots. We have set the number of basis to equal 16, 32 or 64 for benchmarking purposes.

4.2 Results

Algorithm (2) requires at least 3 matrix-vector multiplication needs to be performed three times. We conducted Monte Carlo simulation by repeating the algorithm 50 times and recorded the total time taken of the matrix-vector multiplication. The average value of the total time taken is then presented in the result below.

Fig. 4 shows the performance of our two implementations in comparison to the `numpy.dot()` function. We selected `numpy.dot()` for benchmarking due to the widespread use of the `numpy` package for CPU-based linear algebra operations. Our first version corresponds to the implementation where we used P threads per dot product and the result is coloured in purple. In addition, the second version corresponding to P threads per row is shown with blue colour. To that, the benchmark `numpy.dot()` is coloured in orange.

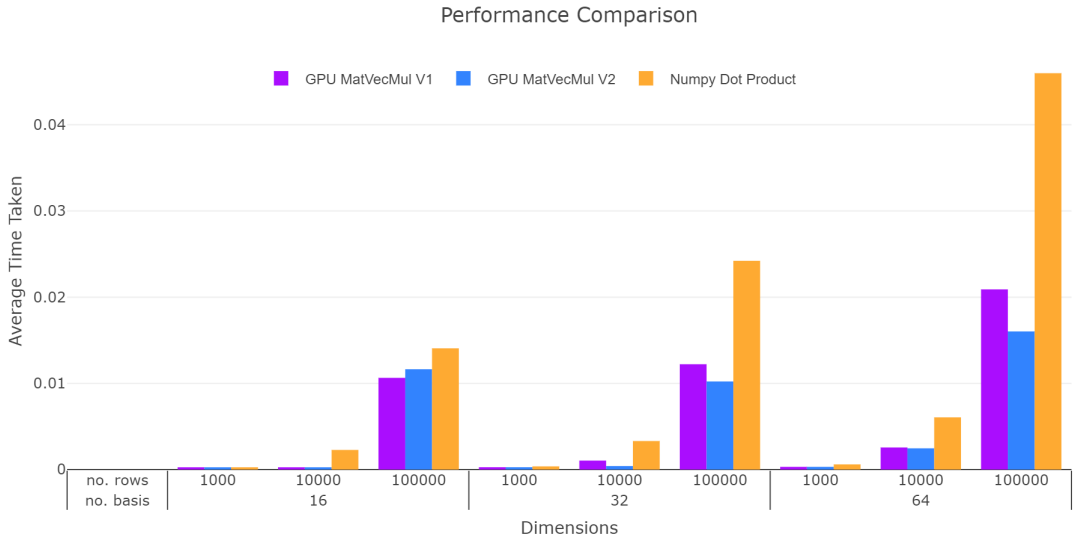

Figure 4: Performance Comparison

Looking at the result, when the dimension of the input matrix $\mathbf{A}$ size was small, we did not observe huge difference in the time taken. However, the GPU matrix-vector multiplication is noticeably faster than `numpy.dot()` on matrices with larger dimension, which suggests that GPU acceleration can indeed increase the speed of gradient-boosted GAMs. Moreover, the two versions of our implementation displayed similar average time

taken as the difference was lower than 0.01 in most cases. We would also like to point out that with similar trend, the performance speed of the second version is higher compared to the first version as the dimension gets larger gradually.

5. Conclusion

We conducted series of numerical simulation for different problem sizes and its performance were measured in both GPU and CPU settings. Our results indicated that GPU acceleration can be a useful method for reducing fitting times of gradient boosted GAMs by parallelising one of the main computational bottlenecks, that is matrix-vector multiplications.

The results are currently limited to the savings obtained exclusively by parallelising matrix-vector multiplications. However, future research should aim to develop a holistic approach of GPU accelerated gradient boosting algorithm to avoid transferring the computed values between the host and the GPU. In addition, performance could be compared against state-of-the-art matrix-vector multiplication libraries, such as the `gemv` function from `cuBLAS` developed by NVIDIA.

Lastly, the project still requires major works to be usable in industrial and research applications. A major concern with GPU-accelerated software is the ease of setting it up across common hardware and operating system setups, which has not been addressed in this project. Furthermore, a user-friendly setup of a wide variety of base learners would be needed along with other important utility functions and documentation to increase the popularity of these methods.

References

[1] E. Bainville. Gpu matrix-vector product, 2010. URL http://www.bealto.com/gpu-gemv_intro.html.

[2] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[3] P. H. Eilers and B. D. Marx. Flexible smoothing with b-splines and penalties. *Statistical science*, 11(2):89–121, 1996.

[4] B. Hofner, A. Mayr, N. Robinzonov, and M. Schmid. Model-based boosting in r: a hands-on tutorial using the r package mboost. *Computational statistics*, 29:3–35, 2014.

[5] H. H. B. Sørensen. High-performance matrix-vector multiplication on the gpu. In *Euro-Par 2011: Parallel Processing Workshops: CCPI, CGWS, HeteroPar, HiBB, HPCVirt, HPPC, HPSS, MDGS, ProPer, Resilience, UCHPC, VHPC, Bordeaux, France, August 29–September 2, 2011, Revised Selected Papers, Part I 17*, pages 377–386. Springer, 2012.

[6] G. Tutz and H. Binder. Generalized additive modeling with implicit variable selection by likelihood-based boosting. *Biometrics*, 62(4):961–971, 2006.

Appendix A.

We can approximate edf from the trace of hat matrix, H.

$$\begin{aligned} \text{edf} &\approx tr(H) \approx tr(H^T H) \approx tr(2H - H^T H) \\ \text{where } \; tr(H) &= tr(X[\mathbf{B}^T W \mathbf{B} + \lambda I]^{-1} \mathbf{B}^T W) \end{aligned}$$

We employ the segment R from a QR decomposition,

$$\begin{aligned} (RR^T)^{-1} &= U \Lambda U^T \text{ where } QR = \tilde{\mathbf{B}} = \sqrt{W}\mathbf{B} \\ \Rightarrow \underbrace{RR^T}_{\text{Symmetric}} &= U \Lambda^{-1} U^T \end{aligned}$$

With the above result, we can compute the base learn as follow,

$$\begin{aligned} \tilde{\mathbf{B}}^T \tilde{\mathbf{B}} + \lambda I &= R^T (I + \lambda R^{-T} I R^{-1}) R \\ \tilde{\mathbf{B}}^T \tilde{\mathbf{B}} + \lambda I &= R^T U (I + \lambda \Lambda) U^T R \\ tr(\sqrt{W}^{-1} \tilde{\mathbf{B}} [\tilde{\mathbf{B}}^T \tilde{\mathbf{B}} + \lambda I]^{-1} \tilde{\mathbf{B}}^T \sqrt{W}) &= tr(\sqrt{W}^{-1} Q R R^{-1} U (I + \lambda \Lambda)^{-1} U^T R^{-T} R^T Q^T \sqrt{W}) \\ &= tr(\sqrt{W}^{-1} \sqrt{W} Q Q^T U U^T [I + \lambda \Lambda]^{-1}) \\ &= tr([I + \lambda \Lambda]^{-1}) = \sum_i 1/(1 + \lambda \Lambda_{ii}) \end{aligned}$$

The above formulation can be differentiated and used for finding the penalty parameter λ, such that edf $= \omega$.

Appendix B.

Version 1.

```
__kernel void mv1(int m,int n, __global const float * a,
                  __global const float * x,
                  __global float * y
                  ){
  // Compute partial dot product
  __local float work[16];
  float sum = 0;
  for (int k=get_global_id(COL_DIM);k<n;k+=get_global_size(COL_DIM))
    {
      sum += a[get_global_id(ROW_DIM)+m*k] * x[k];
    }
  // Each thread stores its partial sum in WORK
  int rows = get_local_size(ROW_DIM); // rows in group
  int cols = get_local_size(COL_DIM); // initial cols in group
  int ii = get_local_id(ROW_DIM); // local row index in group, 0<=ii<rows
  int jj = get_local_id(COL_DIM); // block index in column, 0<=jj<cols
  work[ii+rows*jj] = sum;
```

```
  barrier(CLK_LOCAL_MEM_FENCE); // sync group
  // Reduce sums in log2(cols) steps
  while ( cols > 1 )
    {
      cols >>= 1;
      if (jj < cols) work[ii+rows*jj] += work[ii+rows*(jj+cols)];
      barrier(CLK_LOCAL_MEM_FENCE); // sync group
    }
  // Write final result in Y
  if ( jj == 0 ) y[get_global_id(ROW_DIM)] = work[ii];
}
```

Version 2.

```
__kernel void mv2(int m,int n, __global const float * a,
                  __global const float * x,
                  __global float * y
                  ){
  // Compute partial dot product
  __local float work[4];
  float sum = 0;
  for (int k=get_global_id(COL_DIM);k<n;k+=get_global_size(COL_DIM))
    {
      sum += a[get_global_id(ROW_DIM)+m*k] * x[k];
    }
  // Each thread stores its partial sum in WORK
  int rows = get_local_size(ROW_DIM); // rows in group
  int cols = get_local_size(COL_DIM); // initial cols in group
  int ii = get_local_id(ROW_DIM); // local row index in group, 0<=ii<rows
  int jj = get_local_id(COL_DIM); // block index in column, 0<=jj<cols
  work[ii+rows*jj] = sum;
  barrier(CLK_LOCAL_MEM_FENCE); // sync group
  // Reduce sums in log2(cols) steps
  while ( cols > 1 )
    {
      cols >>= 1;
      if (jj < cols) work[ii+rows*jj] += work[ii+rows*(jj+cols)];
      barrier(CLK_LOCAL_MEM_FENCE); // sync group
    }
  // Write final result in Y
  if ( jj == 0 ) y[get_global_id(ROW_DIM)] = work[ii];
}
```