Spring 2020 ME/CS/ECE759 Final Project Report
University of Wisconsin-Madison

CUDA Accelerated Implementation of Naive Bayes and it's variants

Sandeep Kumar Ramani
Shashank Verma

May 6, 2020

Abstract

"Naive Bayes" is a famous machine learning (ML) algorithm for classification problems, and it is widely used especially for document classification tasks. Various CPU-based implementations are well supported in popular libraries like Scikit learn. However, the popular CuML library (CUDA accelerated ML in Python) does not yet support GPU accelerated versions of the various variants of Naive Bayes (Multinomial , Gaussian, Bernoulli, Complement), except Multinomial. Doing a quick web search doesn't reveal any open-source and clean CUDA/C++ implementations easily. Motivated by this, we implement high performance versions of these Naive Bayes' algorithms in both CUDA and OpenMP. We do so in order to be able to compare both CPU and GPU-accelerated versions, and give a performance analysis.

Link to Final Project git repo: https://github.com/shashank3959/me759-final-project

Contents

1. General information

- Your home department: Electrical and Computer Engineering
- Current status: MS
- Individuals working on the Final Project
    - **Sandeep Kumar Ramani**: Literature Survey, Multinomial and Gaussian variants of Naive Bayes in CUDA, OpenMP and OpenMP profiling.
    - **Shashank Verma:** Literature Survey, implementing Bernoulli and Complement variants of Naive Bayes in CUDA, OpenMP and CUDA profiling.
- I release the ME759 Final Project code as open source and under a BSD3 license for unfettered use of it by any interested party.

2. Problem statement

Different variants (CPU-based) of the Naive Bayes algorithm exist in sklearn (Multinomial , Gaussian, Bernoulli, Complement), however, GPU accelerated versions of all these variants do not exist in CuML (Nvidia RAPIDS ([5]) Machine Learning library) other than Multinomial yet. There are several research publications that discuss a parallelized implementation of the Naive Bayes algorithm, but the authors of these (eg. [1]) works haven't released an open-source version of their implementations either. And more so, very few authors even talk about the comparison of a GPU based CUDA ([3]) version and a CPU based OpenMP ([4]) version.

The motivation for this project are as follows:

- The idea originated from a feature request for the same [5] asking for implementations of these in CuML. Our long term objective is to be able to use the CUDA accelerated implementation from this project, and add suitable abstraction in Python to be able to contribute to this open source project.
- Both Sandeep and Shashank are pursuing an MS specializing in Machine Learning / Data Science. We were both curious to solve this, and use the knowledge we gained from coursework in ME759 in this project. In general, we gained a deeper understanding of how an ML algorithm can be accelerated in hardware.

3. Solution description

In this project, we experimented using the popular **IMDb** movie reviews dataset [7] containing movie reviews from users and a corresponding label to classify whether a review is good or bad. Firstly, we preprocessed this text data into numerically appropriate representations such as "one-hot" and "bag-of-words" using Python libraries like the natural language processing toolkit (NLTK; [6]) and the Scikit-learn library ([2]). This basically involved cleaning the text by removing stop words, symbols, stemming the word, and splitting the dataset into dedicated test and train sets. This data is then stored in the CSV file format and are used as inputs to the classification algorithms.

The other dataset we used was the **Iris** Dataset [8]. This has continuous features (unlike word frequencies in IMDb) that we could use for testing the Gaussian Naive Bayes algorithm. The Gaussian Algorithm is typically used when we are dealing with continuous data that is roughly distributed as a Gaussian distribution. The Iris dataset has 4 features related to attributes of flowers, which can be used to classify flowers in 3 distinct categories.

Introduction to Naives Bayes Classifier

The Naive Bayes classifier is a famous machine learning predictive modeling algorithm widely used for classification tasks (especially document classification) and is based on the principles of Bayes' probability theorem.

The algorithm assigns probability to each class $\mathbf{c_i}$ based on the given data $\mathbf{d_i = (a_1, a_2, a_3, \ldots, a_v)}$ where $\mathbf{a_i}$ is a binary or weighted vector representing the frequency of terms in the document $\mathbf{d_i}$. Here, **v** is the size of the vocabulary.

Assume that we have **k** total classes. By taking the maximum of this conditional probability $\mathbf{P(c_i \mid d_i)}$ across all the classes $\mathbf{c_i}$ where i={0, 1, 2, 3, 4, .., k}, we get a class with maximum probability. The posterior probability is given by,

$$P(c_i|d_t) = \frac{P(c_i)P(d_t|c_i)}{P(d_t)}$$

where, the prior probability,
$P(c_i)$ = (Number of instances belonging to class c_i) / Total number of instances among all classes

In order to calculate the probability of each class $\mathbf{c_i}$ where **i = {0, 1, 2, 3, 4, .., k}**, we need to calculate $\mathbf{P(d_t \mid c_i)}$ which is **computationally expensive** as $\mathbf{d_t}$ tends to have many features (a large vocabulary). One "**Naive**" approach to solve this issue is to assume that the features are independent of each other. This is equivalent to assuming that the frequency of occurrence of a given term in a document is independent to that of all other terms. This allows us to write the probability of $P(d_t \mid c_i)$ in terms of the product of probability of each term/feature x_i using the Independence rule. This makes the calculations viable.

Therefore, the above equation is reduced to

$$P(c_i|d_t) = \frac{P(c_i)\prod_{\forall j \in d_t} P(a_j|c_i)}{P(d_t)} \qquad \text{[Equation 1]}$$

The different variants of Naive Bayes

In this section, we introduce the different variants of the Naive Bayes algorithm and discuss **OpenMP** and **CUDA** solutions to the algorithm. Note that the independence assumptions make our life much easier, because as the features are assumed to be exclusive to each other, we can compute and use them independently in **parallel**. This often leads to some embarrassingly parallel sections in code.

Multinomial Naive Bayes

In Multinomial Naive Bayes, the conditional probability $P(a_j \mid c_i)$ is defined as

$$P(a_j|c_i) = \frac{Tc_i(a_j) + 1}{\sum_{v \in V} Tc_i(v) + 1}$$

where,

- 1 is an additive Laplace smoothing parameter, V (vocabulary) is the number of unique words in the training data.
- $Tc_i(a_j)$ is the sum of frequencies of word a_j present in class c_i from all documents in the training set.
- $\Sigma_v Tc_i(v)$ is the sum of all term frequencies that belong to class c_i in the training dataset.

To avoid division by zero, an additive Laplace smoothing term of 1 is added to both numerator and denominator. Substituting this conditional probability into Equation (1) yields the posterior probability whose maximum value among all the classes yields the required label.

Flow of the algorithm

The training phase of the algorithm consists of calculating the conditional probability and prior probability from the given data. These calculations can be divided into two stage processes, first we calculate the number of occurrences of each term a_j and sum of occurrences of each term for each class by looping over all features in the training data. We then use this to calculate the class probability.

In the testing phase, we uses this class probability and prior probability along with the test data to predict the class C_pred using the negative log probability as shown below,

$$C\_pred = \text{argmax}_i \left[\log P(c_i) + \Sigma_v \log P(a_j \mid c_j) \right]$$

where, V is the number of terms in the document. Here we used negative log probability so that we avoid the use of double precision as probability are very small and solve underflow issues.

Gaussian Naive Bayes

This algorithm is very similar to the definition above, except with the variation that Gaussian Naive Bayes handles continuous variables instead of discrete. Thus the conditional probability is modeled in terms of the gaussian density function with parameters mean and variances defined below,

$$p(x = v \mid C_k) = \frac{1}{\sqrt{2\pi\sigma_k^2}} e^{-\frac{(v-\mu_k)^2}{2\sigma_k^2}}$$

These parameters for mean and variance are calculated per feature for each class. During the test phase, we simply use this form of conditional probabilities to do the predictions.

Bernoulli Naive Bayes

If the data are represented in one hot/binary representation, every feature in the document is associated with a 1 or 0 corresponding to whether the feature exists or not. If the feature vector has **m** dimensions, where **m** is the number of words in the vocabulary, then the conditional probability is modeled using the Bernoulli trials as defined below:

$$P(\mathbf{x} \mid \omega_j) = \prod_{i=1}^{m} P(x_i \mid \omega_j)^b \cdot (1 - P(x_i \mid \omega_j))^{(1-b)} \quad (b \in 0, 1).$$

Note that the variable w_j above represents a distinct class j.
The maximum-likelihood estimate (MLE) $P(x_i \mid w_j)$ for a particular word x_i occuring in the class w_j is defined as,

$$\hat{P}(x_i \mid \omega_j) = \frac{df_{xi,y} + 1}{df_y + 2}$$

where, $df_{xi,y}$ is the number of documents in the training dataset that contain the feature x_i belonging to the class w_j. d_{f_y} is the number of documents in the training dataset that belong to class w_j. The constants +1 and +2 are additive for Laplace smoothing for a more stable division.

Complement Naive Bayes

The Complement Naive Bayes (CNB) algorithm is a twist on the Multinomial Naive Bayes. This algorithm is known to produce more stable parameter estimates especially in datasets with an imbalance between classes. It often outperforms the Multinomial Naive Bayes algorithm in such situations.

CNB uses statistics from the complement of each class. For instance in the training phase, the parameters θ are estimated using the following steps:

$$\hat{\theta}_{ci} = \frac{\alpha_i + \sum_{j:y_j \neq c} d_{ij}}{\alpha + \sum_{j:y_j \neq c} \sum_k d_{kj}}$$

$$w_{ci} = \log \hat{\theta}_{ci}$$

$$w_{ci} = \frac{w_{ci}}{\sum_j |w_{cj}|}$$

$\mathbf{d_{ij}}$ represent the frequency of term/feature **i** in class **j**. Here the summations are over all documents **j** that do not belong to the same class **c** (hence the complement). The constant **α** is used to make the division more stable. The normalization step of the parameters smoothes out the tendency of larger dataset (or documents) to dominate the parameter calculations.
In the testing phase, the parameter estimates of θ are used as follows:

$$\hat{c} = \arg\min_c \sum_i t_i w_{ci}$$

where the prediction ĉ is the class that gives the minimum sum of frequency-weight products for all features/terms. The weights are from the learning phase and the frequencies are from the test document term frequencies.

Optimization using OpenMP

In this section, we explain how we use OpenMP ([4]) acceleration techniques to parallelize the Naive Bayes algorithm. We experimented with 10 core CPUs (on Euler) with a maximum of 20 threads. An important observation in general among all the algorithms is that we have nested loops which iterate over the entire training data, and for each instance (or data point) to calculate the frequency terms which we plug-in to find the conditional probability.

We enabled loop parallelism using a “static” scheduler and “collapse” the loops which resulted in reduction in the execution time. Here we used a static scheduler since the workload is balanced. We parallelized all levels of loops instead of just the outer one using the collapse method.

The independence assumption of the Naive Bayes algorithm makes it suitable for parallelization. We optimized the sequential version using reordering the calculations of terms, avoiding critical sections and using 2D structure of array representation instead of 1D. We loaded the training and test data and labels onto the memory using the **OpenMP Sections** method since data loading tasks can be executed in parallel. This resulted in speed-up of execution time of the code overall. We found that since these probabilities are very small, in order of 10^{-6} , in order to get accurate results we need to store these data in **double** precision format. But this leads to poor cache hit and many read cycles from memory. Thus we modified the algorithm to single precision floating point representation using the concept of log probability where we calculate the probability in **log space**. This too contributed significantly to the speed up. Here is a small snippet of the code using OpenMP constructs:

```
/* How many documents contain each term (per label) */
#pragma omp for collapse(2)
for (int i = 0; i < train_size; ++i) { /* For each example */
        for (j = 0; j < n_features_; ++j) { /* For each feature*/

                feature_probs_[labels[i]][j] += data[i][j];
                if (j==0)
                   {
                   class_count_[labels[i]] += 1;
                   }
        }

}
```

Fig. 1

Optimization using CUDA

The independence assumption of the Naive Bayes algorithm makes it suitable for parallelization. The algorithm (and it's variants) in general has a few distinct parts. Some of these parts, with some intelligent reordering of code can be made embarrassingly parallel.

For instance, let's take the case of Multinomial Naive Bayes (MNB). The algorithm can roughly be divided into 3 parts: Feature Aggregation, Parameter Estimation, and Testing phase. Given our assumption of independence of features, we can enable parallelism using separate kernels for each phase.

- For the **feature aggregation** phase, we wrote a kernel that launches a dedicated thread for each single feature. This kernel simply aggregates (see `MultinomialNBCalcKernel` in code) the total occurrences of each feature for each class. In effect, each thread only has to run a loop iterating through all the classes. There are no critical sections, and this is embarrassingly parallel.

- For the **parameter estimation** phase (see `MultinomialNBLearnKernel` in code) we launch a dedicated thread per feature. We have one weight for each feature per class. This gives us a weights matrix of (num_feature x num_classes). Each thread can take up the calculations or the "learning" for one feature iterating through all classes. There are minimal bank conflicts and unaligned accesses. There are no critical sections, and this again is embarrassingly parallel.

- For the **testing phase**, (see `MultinomialNBTestKernel` in code) we launch a CUDA thread for each test sample. This means the scoring of all samples is parallelised. Since test samples and testing calculations are independent, this is also embarrassingly parallel. We note that there is a potential for use of **Shared Memory** here, since all threads share the learned parameters. However, we couldn't find time to fully explore this option in the time scope of this project.

To give an overview of calculations and memory operations, we provide the following illustration:

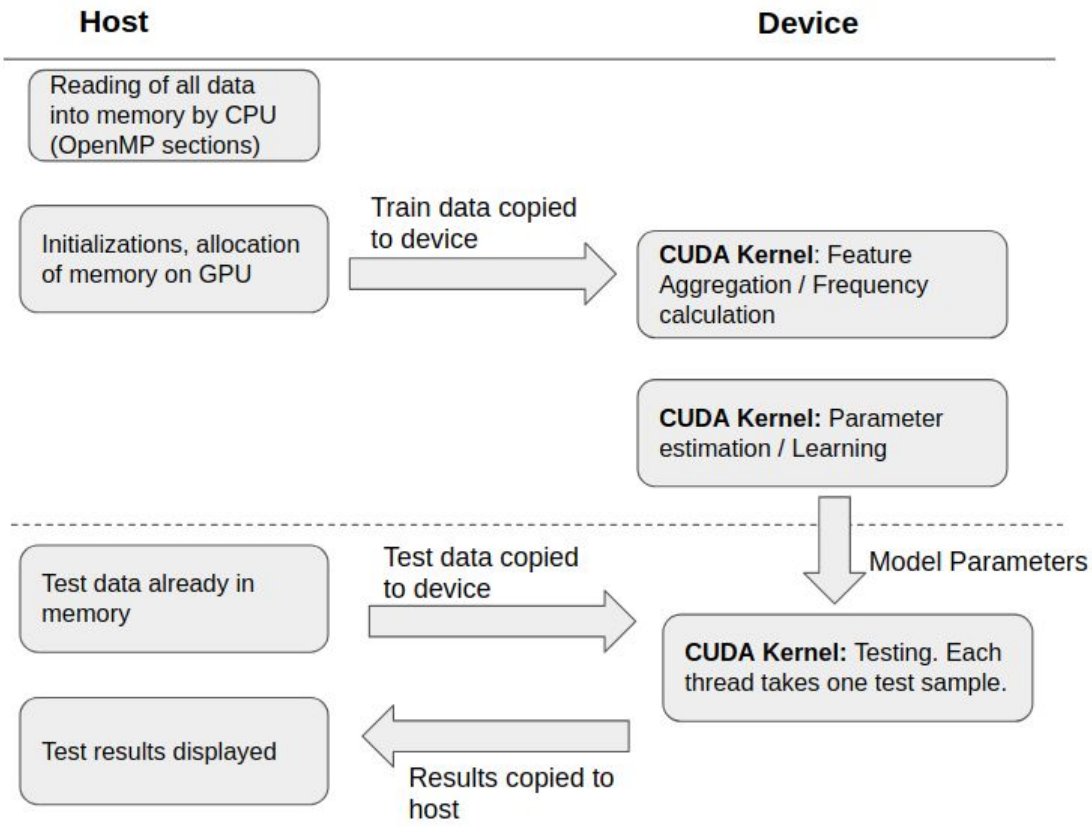

Fig. 2

We use similar ideas to exploit parallelism in all the variants. For example, our Gaussian NB implementation has 4 distinct CUDA kernels for calculations of sum, mean, variance, and the testing phase. We note that because of the design of our approach, we can potentially gain more performance as the number of features, or the number of test samples are increased in data! We also use some basic operations from the thrust library directly like *thrust::reduction* and *thrust::sort* in an attempt to improve efficiency ([9]).

As probably expected, the biggest bottlenecks are memory operations. We try to minimize data movement between the CPU and the GPU, but can't avoid copying the training and testing data over at least once. Since our training and testing data are huge, memory operations are a definite bottleneck. Since our data is very sparse, in future there is potential in the option of copying compressed data to the device, and using device computational resources to uncompress and use the data.

NOTE: All our experiments with CUDA were done with a **GTX 1080 Ti**

Functional Verification

In order to verify our functionality, we used scikit-learn [2], a machine learning library in Python. We observed 96% accuracy scores on the test set (using Iris) in Gaussian Naive Bayes and approx. 84% in all other variants (using IMDb) of Naive Bayes algorithm. We got similar results in both OpenMP and CUDA implementation, thus making them functionally correct.

4. Overview of results. Demonstration of your project

4.1 OpenMP Plots

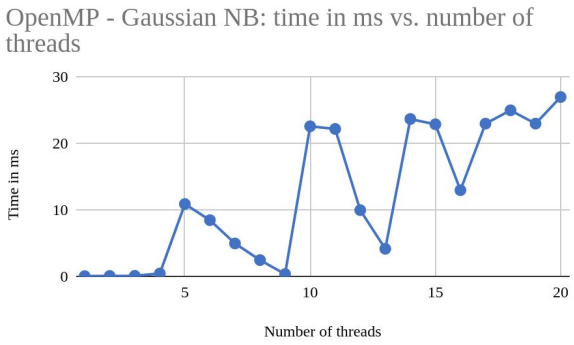

Fig 3.

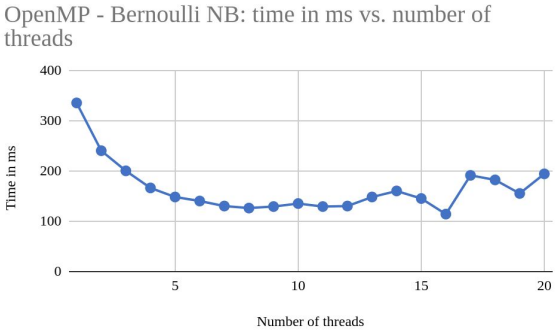

Fig. 4

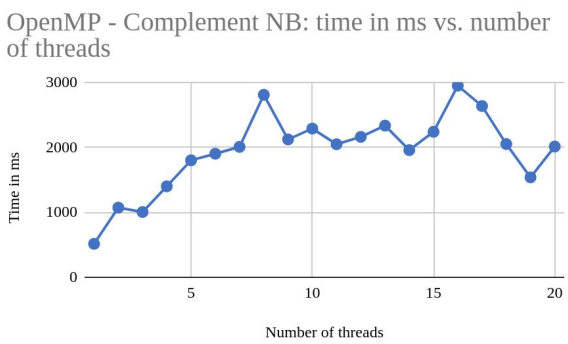

Fig. 5

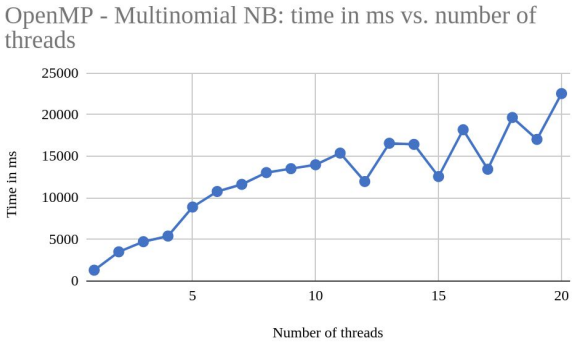

Fig. 6

We found that the execution time of Gaussian Naive Bayes increased after the number of threads equaled 4 as shown in the Fig.3. This is because we think that since we had only 4 features in the training data as we increased the number of threads it was not helping in parallelizing. In the case of Bernoulli Naive Bayes, the execution time decreased with increase in the number of threads. In Multinomial Naive Bayes, single threaded execution was better than multi-threaded because we found by profiling that the critical section (to find the feature count per label) was causing significant delay. In the future work, we would like to remove this critical section.

4.2 CUDA Plots

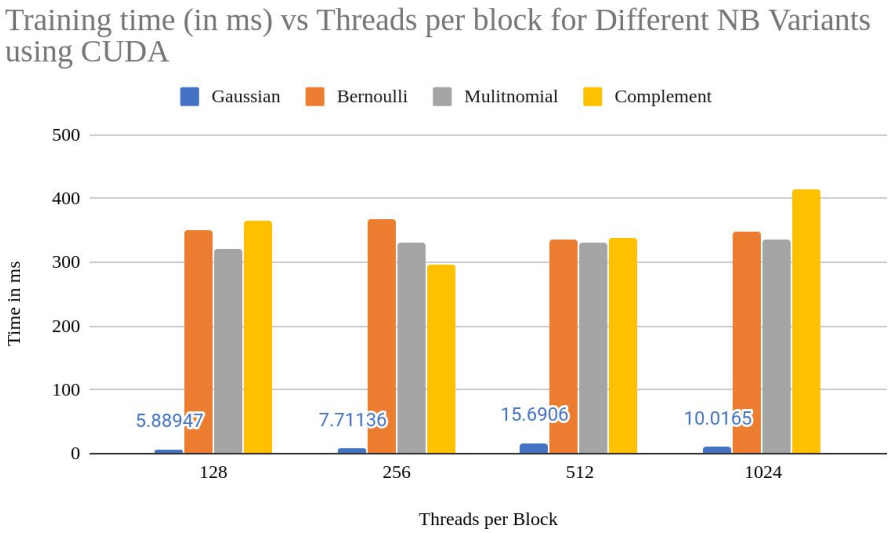

Fig. 6

From Fig. 6 we see that Gaussian takes much less time than other variants. This is because it uses the Iris dataset (as mentioned before), which is much smaller than the IMDb dataset. The other three variants (Bernoulli, Multinomial, Complement) take roughly similar amounts of training

time over runs. Interestingly, as the number of threads per block increases, the training time remains consistent (not much overall increase or decrease for all three). We think this might be because transferring data to devices might be the biggest bottleneck, and not the lack of SM occupancy.

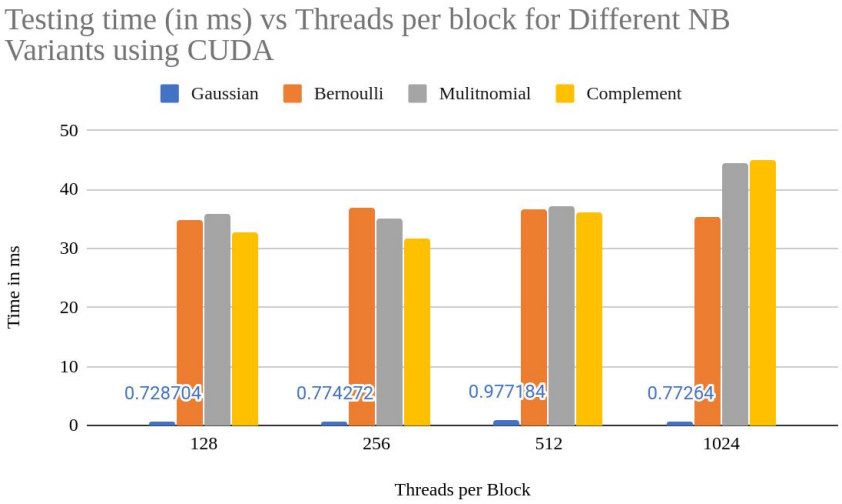

Fig. 7

We see similar results for test time as we had for training times. There is a mild positive trend in test times vs threads per block. However, the trend was not strong enough over many runs to draw a significant conclusion.

4.3 Comparison of OpenMP vs CUDA

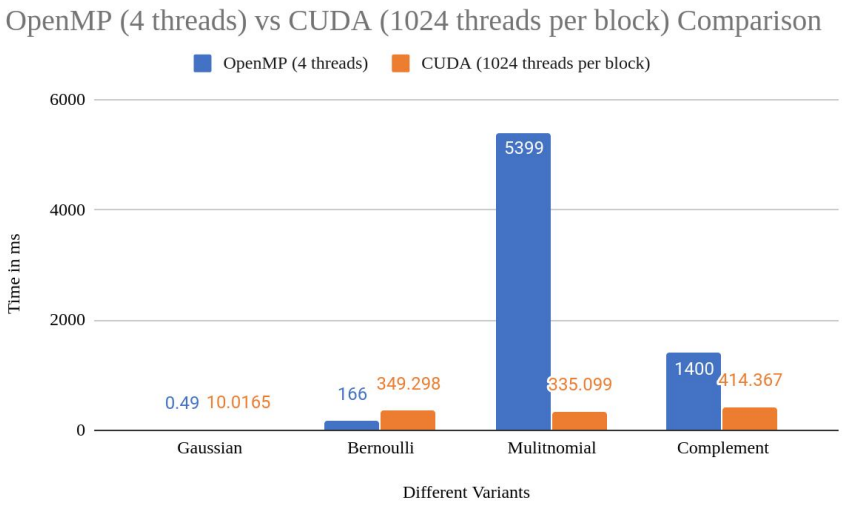

Fig. 8

Best case comparisons between the best case (threads per block) of CUDA vs best case (num. threads) of OpenMP for each variant:

a. **Gaussian:** CUDA (128 threads per block) gives a ~65x slowdown as compared to single threaded OpenMP.
b. **Bernoulli:** CUDA (512 threads per block) gives a ~2.5x slowdown as compared to OpenMP (8 threads)
c. **Multinomial:** CUDA (128 threads per block) gives a ~4x speedup over single thread OpenMP (sequential code)
d. **Complement:** CUDA (256 threads per block) gives ~2x speedup over single thread (sequential code)

5. Deliverables

a. The project report is available on Canvas
b. The accompanying code is present at this github location. We have provided a instructions README with the following:
    i. Scripts to obtain the already preprocessed dataset (or scripts to preprocess it yourself)
    ii. Instructions to compile and run the code for both OpenMP and CUDA versions on both Euler and Windows/Mac.
    iii. Instructions to check the functionality (or correctness) of our C++ based OpenMP /CUDA implementation compared with the standard Python ML scikit-learn library.

6. Conclusions and Future Work

We had the following observations:

- Except in the Case of Bernoulli NB, OpenMP actually failed to give better results when the number of threads were increased. We suspect this may be because of indirection, and critical sections in a part of the code. Fixing this is part of our future work as well.
- In small datasets (like Iris; Gaussian NB), the overhead of launching the many kernels trumps the potential performance gain due to GPU acceleration.
- Memory transactions are the most expensive operations, and our implementations are memory bound.

In future, we would like to do the following:

- Exploring results on other bigger datasets. We think our CUDA kernels may give better efficiency when the number of features, or number of test samples are higher.
- Remove the usage of critical sections in OpenMP Multinomial code. We found (using timers) that the one critical section used in code is the reason for the rapid performance drop as the number of threads are increased.
- To save on memory transactions, we may consider moving compressed data (since it's usually sparse in nature) to the CUDA device and figuring out a way to uncompress it on the device.
- A thorough check for unaligned accesses and bank conflicts to improve CUDA speedup.
- Exploring more optimization tricks for OpenMP code (like simd, etc.)

References

[1] Andrade, G., Viegas, F., Ramos, G. S., Almeida, J., Rocha, L., Gonçalves, M., & Ferreira, R. (2013, October). GPU-NB: a fast CUDA-based implementation of naive bayes. In *2013 25th International Symposium on Computer Architecture and High Performance Computing* (pp. 168-175). IEEE.

[2] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Vanderplas, J. (2011). Scikit-learn: Machine learning in Python. *the Journal of machine Learning research*, *12*, 2825-2830.

[3] Nvidia, C. U. D. A. (2011). Nvidia cuda c programming guide. Nvidia Corporation, 120(18), 8.

[4] Chandra, R., Dagum, L., Kohr, D., Menon, R., Maydan, D., & McDonald, J. (2001). *Parallel programming in OpenMP*. Morgan kaufmann.

[5] Nvidia Rapids AI (https://github.com/rapidsai/cuml/issues/1666)

[6] Loper, E., & Bird, S. (2002). NLTK: the natural language toolkit. *arXiv preprint cs/0205028*.

[7] IMDB dataset: https://ai.stanford.edu/~amaas/data/sentiment/

[8] Iris dataset: https://archive.ics.uci.edu/ml/datasets/iris

[9] Bell, N., & Hoberock, J. (2012). Thrust: A productivity-oriented library for CUDA. In GPU computing gems Jade edition (pp. 359-371). Morgan Kaufmann.