

Acceleration of Support Vector Machine algorithm (SVM) on GPU

Pratyush Agrawal

Department of Electrical Engineering
Columbia University
New York, US
pa2562@columbia.edu

Pei Liu

Department of Electrical Engineering
Columbia University
New York, US
pl2748@columbia.edu

Abstract—Classification of data into different categories is an important step in different applications such as face detection, cell/gene analysis, handwriting recognition and image classification. Support Vector Machine(SVM) is an algorithm that helps to classify the data efficiently. This paper aims to discuss the implementation and challenges involved in accelerating SVM algorithm on GPU using PyCuda by exploring parallelism in terms of how the data can be computed using tiling, shared memory, efficient thread organization and memory allocation to design efficient cuda kernels.

Index Terms—PyCuda, SVM, Machine Learning, Lock-free

I. INTRODUCTION

Support Vector Machine (SVM) is a supervised machine learning algorithm that classifies the data into different categories by finding a hyper-plane in N-dimensions, where N is the number of features. This hyper-plane divides the data into different categories.

The Hyperplane forms a decision boundary and the data points on either side of the plane are classified into a different class. The dimension of the hyperplane depends on the number of feature, for example if there are only two features then the hyperplane is a line, if there are three features then the hyperplane is a 2D plane.

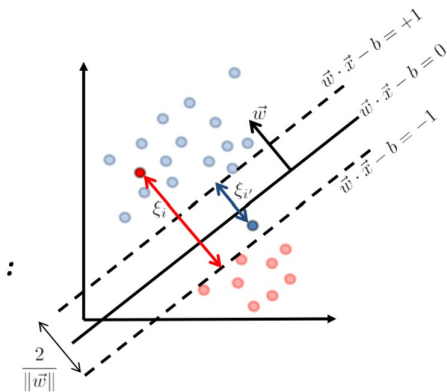


Fig. 1. Equations of hyperplane

In SVM algorithm we try to maximize the distance between the data points and the hyperplane. The loss function that helps

to maximize this margin is called the hinge loss.

For an output, $t = +1$ or -1 and classifier score y the hinge loss of prediction y is defined as $l(y) = \max(0, 1 - t \cdot y)$, where $y = w \cdot x + b$, here w, b are the parameters of the hyperplane and x is the input data point.

$$J(W) = \frac{\lambda}{2} \|W\|^2 + \frac{1}{N} \sum_i \max(0, 1 - y_i (W \cdot x_i + b))$$

Fig. 2. Equation of loss function

Figure 2 shows the equation of loss function that we want to minimize to get the parameters for the hyperplane.

$$\begin{aligned} \text{if } 1 - y_i (W \cdot x_i + b) > 0, \quad \frac{\partial L}{\partial W} &= \lambda \cdot W - \frac{1}{N} \sum_i (-y_i \cdot x_i) \\ \text{else,} \quad \frac{\partial L}{\partial W} &= \lambda \cdot W \end{aligned}$$

Fig. 3. Equation of gradient

Equation in Fig.3 show the gradient which we can calculate from loss function that will be used to update the parameters of the hyperplane.

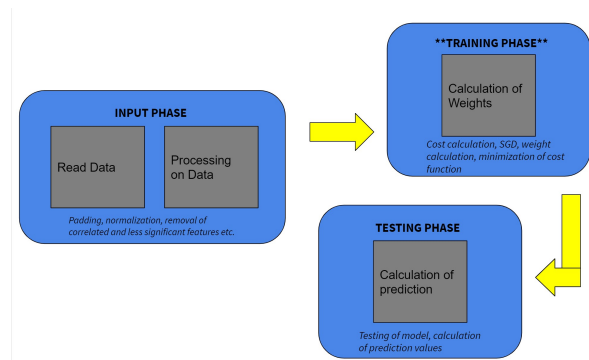


Fig. 4. SVM code flow

Fig.4 shows the code flow for SVM algorithm, it is divided in three phases. The first phase is the "Input Phase" where

we read the dataset and perform pre-processing on data such as, feature reduction, normalization, padding etc.

Phase two is the training phase, where we perform stochastic gradient descent and train the weights with the training data. Phase 3 is the testing phase where we use the trained weights from phase 2 to predict the output on the testing data.

This paper discusses the parallelization aspects and challenges of the training process and different approaches of implementing a CUDA kernel that we explored using "Breast Cancer" (BC) dataset and "CIFAR-10" (CF) dataset for our analysis. Section II discusses the parallelization aspects of our the algorithm, section III discusses the implementation details and different approaches which we followed, section IV discusses the results of the different approaches, section V discusses a detailed analysis of the project and our learnings from the project.

II. PARALLELIZATION ASPECTS

For our project we focused on parallelizing the training part of the code, which trains the weights for the algorithm. For training we use the "Stochastic Gradient Descent" (SGD) algorithm to train weights.

```
while epoch < max_epoch {
    for  $\vec{x}$  in X {
        dw = calculate_cost_gradient ( W,  $\vec{x}_i$ ,  $y_i$  )
        W = W - C * dw

        cost = compute_cost ( W,  $\vec{x}_i$ ,  $y_i$  )
        if cost < cost_threshold
            break
    }
}
```

Fig. 5. Basic flow of SGD

Fig.5 shows the basic flow of SGD, where for every epoch or every training step, we pick one data point from all training examples and then compute the gradient and update the weights, then we pick a second data point and compute the gradient using the new updated weights and then update the weight again, this process goes for each data point in a serial manner. This shows that the SGD is inherently a serial algorithm, in which for each training iteration we serially compute gradient and update weights for each data point.

For the project we decided to explore parallelism in SGD in two methods. In the first method, we explore parallelism in how gradient and weights are computed. In this method we still process each data point serially, but for each data point the computation for each element of gradient and weight happens parallelly. This can be achieved by assigning one thread for every feature, so that each thread can do parallel

computation and calculate each element of weight vector.

In the second method we explored parallelism in how each data point is calculated upon to update the weights, so instead of serially calculating the gradient and updating the weights for each data point we compute the gradient for all the data points and update the weights in parallel lock-free manner [3]. This is achieved by assigning each thread one data point, so now all the threads or all data points operate in parallel and update the weights.

III. IMPLEMENTATION

A. Python SVM Programs

We have three python programs.

- 1) svm_BC-SGD.py
- 2) svm_BC-opencv.py
- 3) train-cifar10-svm.py

We also have 3 corresponding jupyter notebooks for these three python programs having same names.

svm_BC-SGD.py is drafted by referencing the article "SVM-from-scratch-python" [1], and modifying it to make it work for our project. It uses stochastic gradient descent (SGD) algorithm to train a SVM binary classifier by using "breast-Cancer (BC)" dataset. BC dataset has 569 examples, and each example has 31 features. After splitting them as training and testing set, we have 455 Training examples. There are two functions to reduce the dimension of feature space, which are "remove_correlated_features()" and "remove_less_significant_features()". After dimension reduction, our training set has size (455, 12) and testing set is of size (114, 12). Fig.6 shows the pseudo code for the SGD of "svm_BC-SGD.py".

```
function sgd(X, Y):
    for epoch in range(max_epochs):
        X, Y = shuffle(X, Y)
        for every x in X:
            dw = calculate_cost_gradient(W, x, y)
            W = W - (learning_rate * dw)
            if idx == 2^n or epoch == max_epoch - 1
                Idx++;
            cost = compute_cost(W, x, y)
            if (prev_cost - cost)/prev_cost < cost_threshold
                break
            idx += 1
    return W
```

Fig. 6. Pseudo code of naive SGD

svm_BC-opensv.py is modified based on *svm_BC-SGD.py*. We used the same data preprocessing, but the training process implements the library function from OpenCV.

train-cifar10-svm.py is referencing [2]. This program uses cifar-10 dataset. Cifar10 has 49,000 training examples and 1000 testing examples. It has 10 classes of photos, and each photo is of size 32*32*3, thus the weight matrix for this problem is of size (3073, 10). Each column represents the weights trained for one class.

B. PyCUDA program with Breast Cancer Dataset

We have three kernels for BC dataset.

- 1) BC-cuda-naive.py
 - sgd_kernel.cu
- 2) BC-cuda-shared-mem.py
 - sgd_bc_sh_mem.cu
- 3) BC-cuda-lock-free.py
 - sgd_kernel_no_locks.cu

1) *BC-Naive*: This kernel explores parallelism in how each element of weights is calculated, we compute each data point serially however for each data point calculation of each element of gradient and updating of each element of weights happens in parallel.

```
function BC-naive (features, outputs):
    for epoch in range(1, max_epochs):
        X, Y = shuffle(features, outputs)
        For EVERY sample in (X):
            ascent = calculate_cost_gradient(weights, X, Y) //do-parallelly
            weights = weights - (learning_rate * ascent) // do-parallelly
    return weights
```

Fig. 7. Pseudo code of BC naive

Fig.7 shows the pseudo code BC-naive code, since the parallelism is in terms of each element of weights, therefore we only need 12 threads to compute element in parallel. To achieve a high accuracy the input data needs to be shuffle so as to avoid any repeating weight update cycle. Since CUDA doesn't provide a rand() function, to achieve this we explore different approaches; we implemented a hashing function to generate random indexes for the input data, however the indexes generated were not truly random and were often repeating.

To overcome this issue, we kept the epoch loop outside of the kernel, so for every epoch we would first shuffle the data on host and then pass the data to GPU for computation. The weights are stored in the shared memory since they are accessed frequently for computation and once the computation is over we transfer them back to global memory.

2) *BC-Shared-Mem*: This implementation is a tiled version of the naive implementation where we store the inputs into shared memory.

Fig.8 shows the pseudo code of BC shared mem implementation, for every tile we first load the input into the shared memory and then serially process each data point and then we load the next set of data points in the shared memory tile by tile. For our implementation we have take tile width as (64, 12) with block dimensions as (16,64,1).

```
function BC-shared-mem (features, outputs):
    for epoch in range(1, max_epochs):
        X, Y = shuffle(features, outputs)
        For tile in (total tiles):
            x_shared[TW][TW] = x
            y_shared[TW][TW] = y
            for EVERY x in x_shared:
                ascent = calculate_cost_gradient(weights, x, y)
                //do-parallelly
            weights = weights - (learning_rate * ascent)
            // do-parallelly
    return weights
```

Fig. 8. Pseudo code of BC shared mem

3) *BC lock free*: This is the third kernel which we have implemented for BC dataset, this implementation is a lock free approach based on Hogwild! [3] algorithm, where each thread accesses one data point and computes its gradient and update the weights in parallel.

```
function BC-lock-free(features, outputs):
    for epoch in range(1, max_epochs):
        For ALL sample in (X): //do parallelly
            ascent = calculate_cost_gradient(weights, X, Y)
            weights = weights - (learning_rate * ascent)
    return weights
```

Fig. 9. Pseudo code of BC lock free

Fig.9 shows the pseudo code for lock free implementation, here the epoch loop is inside the kernel since all the data points take part in computation in parallel, there isn't a need to shuffle the data for every epoch. Thus, the memory transfer overhead is very small as compared to other kernels where we transfer data from host to GPU for every epoch.

In this implementation all data point uses the weights they currently see and compute their individual gradient and then update the weights in a lock-free manner. Since we have 455 data points, thus, we need only 455 threads and only one block for this implementation.

C. PyCUDA program with Cifar-10 Dataset

We have three kernels for CIFAR-10 dataset.

- 1) CF-cuda-naive.py
 - sgd_cifar_single_blk_normal_mult.cu
- 2) CF-cuda-lock-free.py
 - sgd_cifar_lock_free_2.cu
- 3) CF-cuda-multi-kernels.py -
 - get_w_combo.cu
 - x_dot_w.cu
 - ds.cu
 - delta.cu
 - xT.cu

1) *CF-Naive*: For the cifar naive kernel approach, we have designed the kernel `sgd_cifar_single_blk_normal_mult.cu`. The pseudo code of this kernel is shown in Fig.10 In this kernel code, we have defined 2D block with size (10, 32, 1). The grid size is (1, 1, 1). So there are 320 threads in maximum working for this kernel. When computing $\text{dot}(x, W)$, we are using a tiled approach to compute it. The result of the dot product of one data point with W is stored in `dot_XW[10]` array, which is stored in shared memory. After computation of `dot_XW`, we only have 10 threads left active. Each thread will be working for one column of W matrix. After that we compute distance array, which is also stored in shared memory, by using `dot_XW[10]` array. Then we compute `ds` array and finally we compute `single_dw` matrix, which is of size (3073, 10), and is stored in global memory. We accumulate every element of `dw` for "batchSize" number of examples, then we average them to get `dw` matrix after that we update W for one batch. We also compute one loss for each batch. In order to clearly illustrate how to update W , we intentionally leave out loss computation in pseudo code in Fig.10

```
function sgd_naive(X, Y, W):
    for (x, y) in (X, Y):
        dot_XW[tx] = dot(x, W[:, tx])
        distance[tx] = dot_XW[tx] - dot_XW[y] + 1
        ds[tx] = max(0, distance[tx]) > 0 ? 1 : 0
        ds[y] = 0
        sum_ds = sum(ds)
        ds[y] = -sum_ds
        single_dw += dot(x.T, ds)
    dw = single_dw / batchSize + reg_strength * W
    W = W - learning_rate * dw
```

Fig. 10. Pseudo code of CF-naive

2) *CF-Lock-Free*: For the lock-free kernel, we have "sgd_cifar_lock_free_2.cu" as our kernel code, and "CF-cuda-lock-free.py" as our PyCUDA code. The pseudo code for the kernel is in Fig.11. We let each thread compute one data point. x represents one example, which has size (1, 3073). y is of size (1,), and W is of size (3073, 10). Since each thread block has 1024 threads, if we want to store all x for each thread, then we need the shared memory to be $3073 * 1024 * 4$ bytes = 12 MBytes, which exceeds the capacity of NVIDIA GPU. So we store x in global memory. Weights matrix should also be stored in global memory since it is large to fit in shared memory. During the debugging process, we found out that after calculating dot product of x and W the W stored in global memory becomes NaN. We spent lot of time on this but we haven't figured out the reason. Even though we passed `max_epochs` as 1500 for CF-cuda-lock-free.py program, all weights element in W becomes NaN after the computation of `dot_XW` in the first epoch which results in a poor accuracy.

3) *CF-Multi-Kernels*: This implementation uses a divide and conquer approach where we break down the SGD training code in to multiple kernels which are easier to debug, execute

```
function sgd_lock_free(x, y, W):
    for epoch in range(max_epochs):
        dot_xW = dot(x, W)
        dw = f(dot_xW, y, W)
        dw = calculate_cost_gradient(W, x, y)
        W = W - (learning_rate * dw)
        if epoch == max_epoch - 1
            cost = compute_cost(W, x, y)
```

Fig. 11. Pseudo code of CF-lock-free

and optimize in terms of tiling and memory allocation. To implement the training process we have design five main kernels that execute different functions used in training process.

```
function cf-multi-kernel(x,y):
    for epoch in range(1, max_epochs):
        gpu_func_1 (x_dot_w.cu//x_dot_w_tiled.cu)
        cpu_func   (s_yi)
        gpu_func_2 (delta.cu)
        gpu_func_3 (ds.cu)
        cpu_func   (ds_add)
        gpu_func_4 (xT.cu)
        gpu_func_5 (get_w_combo.cu//get_w_combo_tiled.cu)
    return weights
```

Fig. 12. Pseudo code of CF-Multi-Kernels

Fig.12 shows the pseudo code the CF-Multi-Kernels implementation, the entire training function uses 5 cuda kernel function and 2 native python functions to train the weights. For every epoch all the GPU and CPU functions are executed in series.

The first GPU function performs the dot production of input vector with weights matrix. We have designed two kernels for it; one is naive kernel which performs naive multiplication and another is a tiled version which uses the concept of tiling and shared memory to perform the multiplication.

The second GPU function calculates the delta (distance) used in SVM algorithm, this kernel involves element wise addition and subtraction, since one element is accessed only one time, we have stored the inputs and outputs in global memory.

The third GPU function computes the "ds" matrix which is then used to calculate the gradient. This kernel again performs element wise operation on the matrix calculated from above function. In this kernel since each element is accessed only once we have kept the input and output in global memory only.

The fourth GPU function computes the transpose of input matrix, since each element is accessed once, the input and

output are accessed from global memory only. The fifth GPU function computes the dot product of transpose of input matrix and "ds" matrix then it calculated the gradient and updates the weights, we have implemented a tiled version of this kernel which uses tiling and memory to perform multiplication.

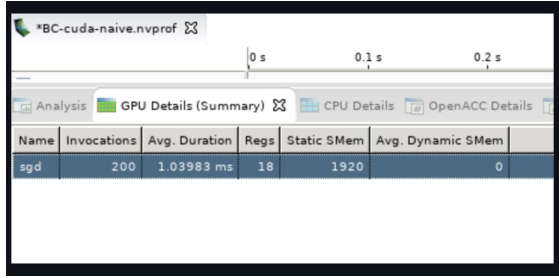
For GPU kernel 1 and GPU kernel 5 we have experimented with different block dimensions and tile widths for the analysis which is discussed in depth in the following section.

IV. RESULTS AND DISCUSSIONS

A. BC Dataset

1) *Python Program*: For serial python implementation we analyzed two program one is written in python using only numpy libraries to perform computation, this program takes 0.84 seconds to execute with an accuracy of 94-97% for 200 epochs, the other python program uses OpenCV library to perform computation, this program takes 0.002 seconds to execute and the accuracy is around 94-97% for 200 epochs.

2) *BC-Naive*: With the first approach using a naive implementation of the kernel, the execution time including memory transfer is 0.27 seconds and execution time excluding memory transfer is 0.19 seconds with an accuracy of 94% for 200 epochs so this approach is approximately 4 times faster than native python code, however it is approximately 100 times slower than OpenCV implementation.



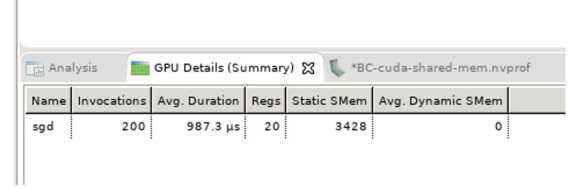
Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem
sgd	200	1.03983 ms	18	1920	0

Fig. 13. GPU summary of BC-naive Implementation

Fig.13 shows the GPU summary result from nvprof, the kernel was invoked 200 times (for each epoch), with average duration 1.03 milliseconds, each thread uses 18 private registers and the kernel used 1.9 Kib of shared memory.

3) *BC-shared-memory*: BC shared memory approach takes 0.27 seconds to execute including memory transfers with an accuracy of approximately 94% for 200 epochs, we can see that even though we used tiling and shared memory for the input data, there was not significant improvement in the execution time as compared to the naive approach. The average time of naive approach and tiled/shared memory approach are almost similar. This is due to the fact that if a data element is used only once then accessing from global memory is faster, if we implement tiling for such cases we

pay double cost of data transfer; first we transfer from global memory to shared memory and then we access the shared memory to read the data.

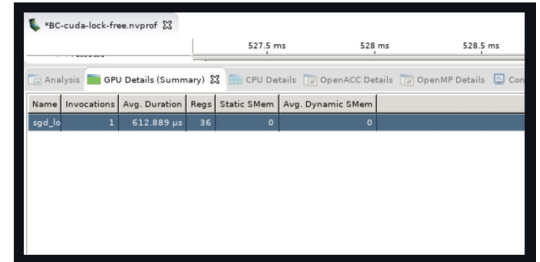


Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem
sgd	200	987.3 μs	20	3428	0

Fig. 14. GPU summary of BC-shared-memory Implementation

Fig.14 shows the GPU summary of the tiled/shared memory implementation, we can see that this implementation uses more registers per thread and more shared memory 3Kib, the average duration of kernel execution being 0.9 milliseconds.

4) *BC-lock-free*: For this lock free implementation, the execution time including memory transfer is 0.002 seconds with an accuracy of approximately 67% for 200 epochs. The accuracy for this implementation is less as compared to naive python or OpenCV implementation, however the execution is approximately 100 times faster than naive python and almost similar to OpenCV implementation.



Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem
sgd_0	1	612.889 μs	36	0	0

Fig. 15. GPU summary of BC-lock-free Implementation

Fig.15 shows the GPU summary of the lock free implementation, since we didn't have to shuffle the data for every epoch, the epoch loop is inside the kernel which results in only one invocation of this kernel, we can see that the average duration is 0.6 milliseconds, this implementation uses 36 registers per thread, which is more as compared to previous implementation, this kernel uses no shared memory.

B. CIFAR-10 Dataset

1) *Python program*: The training time and accuracy of python program is shown in Fig.16 and Fig.17

2) *CIFAR-Naive*: The training time is shown in Fig.18. The training and testing accuracy and loss over max_epochs are shown in Fig.19. We can see that the accuracy is oscillating around 12%. The loss is increasing, the reason is that our kernel code for computing loss and weights has bugs somewhere

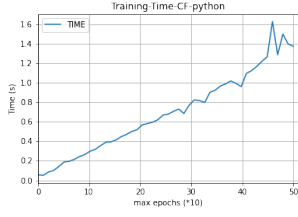


Fig. 16. Training time of CF python

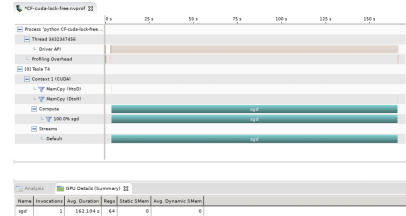


Fig. 20. Profiling of CF-naive

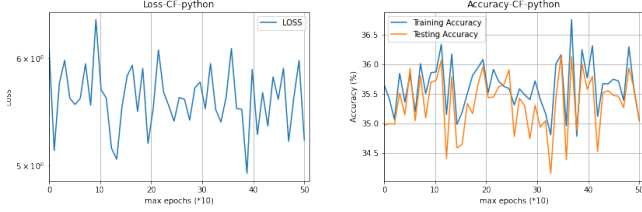


Fig. 17. Training time and Accuracy of CF python program

which we haven't figured out after debugging. The weight matrix computed by this kernel is valid data, which is different from CF-lock-free kernel.

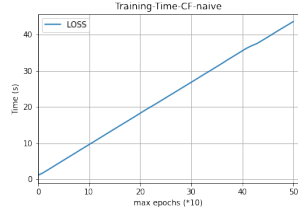


Fig. 18. Training time of CF-naive

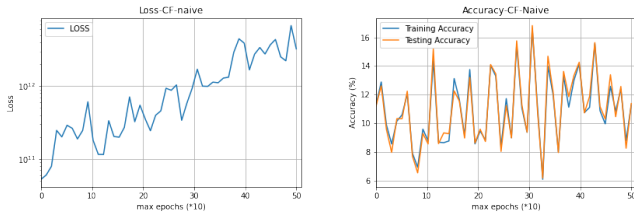


Fig. 19. Training time and Accuracy of CF-naive

3) *CIFAR-Lock-Free*: When we set max_epoch as 500, training time is 162.90 seconds. Training accuracy is 10.03% and testing accuracy is 10.00%. Since the Weight matrix returned by the GPU kernel is "NaN", the prediction is based on the initial weight matrix. Thus, it is not meaningful to compare this accuracy result. The profiling of CIFAR-Lock-Free, shown in Fig.20, this shows us that we only invoke the kernel once and the iteration through max_epochs is done within the kernel. Also, since each thread is working with one data point, the capacity of shared memory is too small to accommodate x or W. So the shared memory usage is not used.

IMPLEMENTATION	BLOCK DIM	TOTAL KERNEL EXECUTION TIME	TOTAL GPU EXECUTION TIME
NAIVE	K1 : 10,100,1 K2: 10,100,1	0.67 secs	6.4 secs
	K1 : 10,100,1 K2: 10,10,1	0.66 secs	6.22 secs
	K1 : 10,10,1 K2: 10,10,1	0.31 secs	5.8 secs
	K1 : 32,32,1 K2: 32,32,1	0.67 secs	6.80 secs
	K1 : 10,100,1 K2: 10,100,1 TW : 100,10	0.59 secs	6.34 secs
	K1 : 10,10,1 K2: 10,10,1 TW : 10,10	0.31 secs	6.28 secs
TILED	K1 : 32,32,1 K2: 10,100,1 TW1 : 32,32 TW2: 100,10	0.47 secs	6.5 secs
	K1 : 32,32,1 K2: 32,32,1 TW1 : 32,32 TW2: 32,32	0.53 secs	6.91 secs

4) *CIFAR-multi-kernels*: Table 1 shows the variation of execution time w.r.t different implementations and configurations. In the table K1, TW1 and K2, TW2 refers to the block dimensions and tile width of GPU kernel 1 and GPU kernel 5 in Fig. 13 respectively, TW refers to the tile width of both of these kernels since these two kernels computationally intensive, we analysed the execution time by varying different parameter and keeping the block dimensions for "delta, ds and xT" kernels as (10,10,1), (10,10,1) and (32, 32,1) respectively.

First we experimented with the naive kernels, we get the fastest execution when the block dimensions of both are kernel are (10,10,1) and execution is the slowest in the case when block dimensions for both the kernel are (10,100,1).

For the tiled kernels the fastest execution was when block dimensions of both the kernel are (10,10,1) and shared memory tiles was (10,10) and slowest when the block dimensions were (10,100,1) with tile width for shared memory as (100,10) for both the kernels.

Fig.21 shows the GPU summary of the naive implementation, we can see that "delta" kernel takes the least amount of time to perform computation and "x_dot_w" takes the maximum time to perform its task it also utilizes more registers per thread as compared to other kernels.

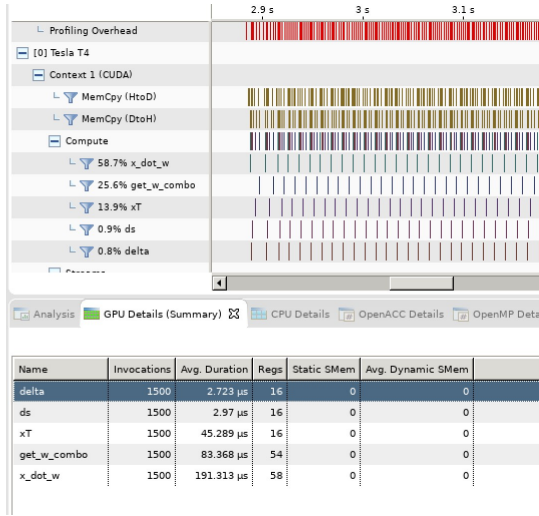


Fig. 21. NVPROF Summary of naive implementation

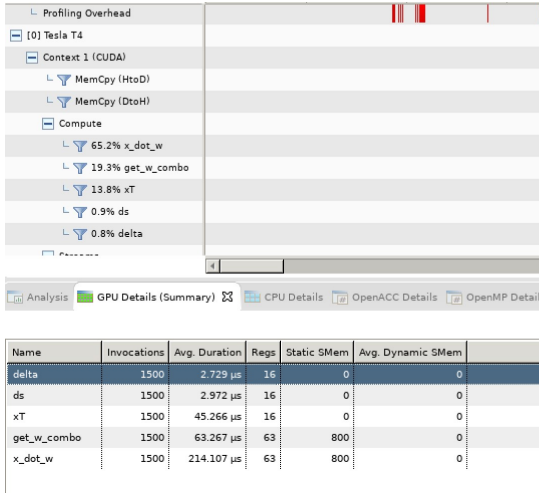


Fig. 22. NVPROF Summary of tiled implementation

Fig.22 shows the GPU summary of the tiled implementation, here we use 0.8Kib of share memory for tiling, which was giving the fastest result as per our analysis with different configurations. One thing to observe is that "get_w_combo" performs faster than naive however there isn't much improvement in execution of "x_dot_w" kernel.

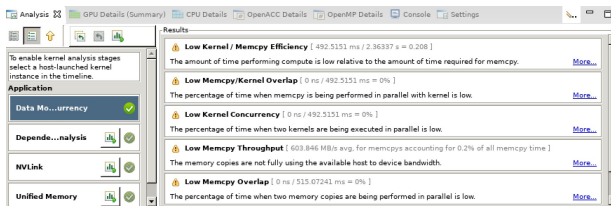


Fig. 23. Data movement analysis

From Fig.23 we can see that the compute to memory transfer ratio is low, there is more overhead in memory transfer which

is also the reason why the total execution time including memory transfer is high.

From our analysis we have selected the dimension configuration which has the fastest training to run the Fig.24. The block dimension for kernel-1 is (10, 10) and the block dimension for the 5-th kernel is also (10, 10). Then we set the max-epoch to different values. We get the training time of CF-cuda-multi-kernel.py, which is shown in Fig.24. We also compare the training time of with python program. We can see that our GPU implementation is faster.

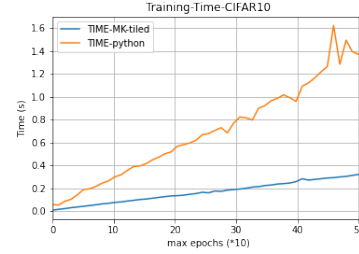


Fig. 24. Running time of CF-multi-kernel and python program

The training and testing accuracy of CF-multi-kernel is shown in Fig.25. We can see that both the training and testing accuracy stop changing after max_epoch reaches 500. So in the following comparison, we will set max_epoch for 500.

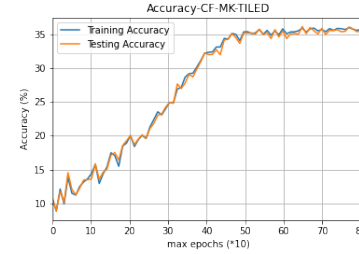


Fig. 25. Accuracy of CF-multi-kernel

The accuracy comparison between multi-kernel and python program is shown in Fig.26. When max_epoch is set to 500, our multi-kernel approach has the similar accuracy as the python program, which is the best accuracy on CIFAR-10 dataset when classified with SVM algorithm reported so far. [3]

V. CONCLUSION AND FUTURE WORK

Through this project we tried to implement the concepts and explore parallelism and experiment with different configuration parameter to design an efficient implementation of the SVM algorithm on GPU. In the project we worked on two data sets different in sizes and complexity, for both the data set we were able to design efficient CUDA kernel which executed faster than the serial python implementation.

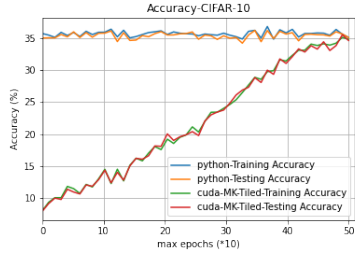


Fig. 26. Accuracy of CF-multi-kernel and python program

During the project we learnt some practical things about parallel programming such as different thread organization and memory allocation and result in different executions, writing multiple kernel is more efficient than writing big kernels, since different computation require different thread organization and memory thus having dedicated structure for different computation can result in efficient execution, however the memory transfer overhead is usually high which we observed from CF-multi-kernel implementation. Our future work would be to try and implement the lock free method and compare the speedup and accuracy w.r.t naive python implementation.

VI. CONTRIBUTION

Proposal discussion	Pei Liu & Pratyush Agrawal
Python code preparation	Pei Liu
kernel code drafting	Pratyush Agrawal
Debugging and discussion	Pei Liu & Pratyush Agrawal
Report writing	Pei Liu & Pratyush Agrawal

VII. ACKNOWLEDGMENT

We would like to thank Prof. Zoran Kostic and course TA Mr. Dwiref Oza for constantly guiding us through out the project and give us an opportunity to implement the concepts and fundamentals of the course in our project.

REFERENCES

- [1] Qandeel Abbassi, SVM From Scratch — Python, <https://towardsdatascience.com/svm-implementation-from-scratch-python-2db2fc52e5c2>
- [2] Zaiyan - SVM with CIFAR 10, <https://www.kaggle.com/zaiyankhan/zaiyan-svm-with-cifar-10>.
- [3] Benjamin Recht, Christopher Re, Stephen Wright, Feng Niu: Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent-Advances in Neural Information Processing Systems 24 (NIPS 2011)