Math 525: HPC Semester Project

Cuda GPU Accelerated: Support Vector Machine

Justin Merkel

Contents

Introduction

Support Vector Machine (SVM) is a type of Machine Learning algorithm that is designed around finding the highest margin decision boundary (hyperplane) that classifies the data points. The intuition of using the highest margin hyperplane is that the further the data point is from the decision boundary, the more confident the classification of that data point is.

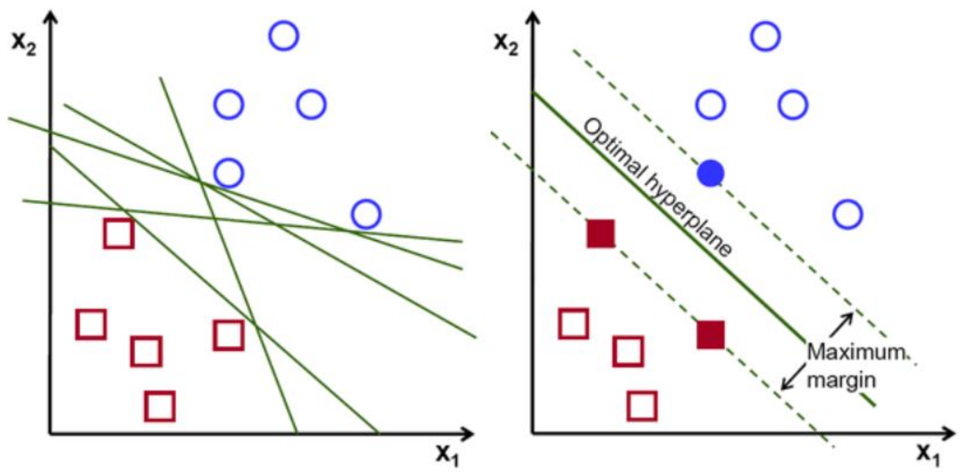


Figure 1: On the left are some possible hyperplanes while on the right is the optimal hyperplane with the maximum margin.

The input to the algorithm is a vector $[x_1, x_2, ...x_n] = x_i$ and a label y_i and the hyperplane is represented as a series of linear weights $[w_1, w_2, ...w_n] = w$. The prediction of the model is $sign(w^T x)$. Since the purpose of SVM is to minimize the margin, the definition of margin needs to be defined. Margin (γ) is the distance from the decision boundary hyperplane which is $\gamma_i = \frac{y_i(w^T x_i)}{||w||}$.

The problem now becomes how to minimize γ.

γ can be minimized by setting up a lagrangian function where the $\alpha_i \geq 0$

$$\mathcal{L}(w, \alpha) = \frac{1}{2} w^T w + \sum_{i=1}^{N} \alpha_i (1 - y_i (w^T x_i))$$

To begin solving this function we must take the partial derivative with respect to w and set the result to 0.

$$\frac{\partial \mathcal{L}(w, \alpha)}{\partial w} = w + \sum_{i=1}^{N} \alpha_i (-y_i x_i) = 0$$

$$w = \sum_{i=1}^{N} \alpha_i y_i x_i$$

Then, substituting back into the original lagrangian, the resulting function is dubbed the "Dual Problem" where result is solely determined by α

$$\mathcal{L}(w, \alpha) = -\frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j x_i^T x_j + \sum_{i=1}^{N} \alpha_i$$

If the weights are known, the αs are known and vice versa. The SVM algorithm initializes alphas to random numbers and that steps through the gradient to determine the resulting weights. If the data is linearly separable, SVM will find the highest margin hyperplane.

In order to use SVM in the non-separable cases, there are two modifications that can be employed. The first modification is using "soft" margin rather than hard margin. Soft margin additionally limits the α to be less than or equal to a parameter C. This prevents points that are grossly miss-classified from dominating the decision hyperplane. The second is to use a kernel function. The kernel trick can be used to map the data to a higher dimensional space using less computation. This is due to the fact that the SVM algorithm only relies on the inner product of the vectors which is computationally cheap. For the polynomial case to map 2 features to quadratic space (x_i, x_j) we must calculate $(1, x_i, x_j, x_i^2, x_j^2, x_i x_j$ but using a kernel is only $(x_i^T x_j + 1)^2$.

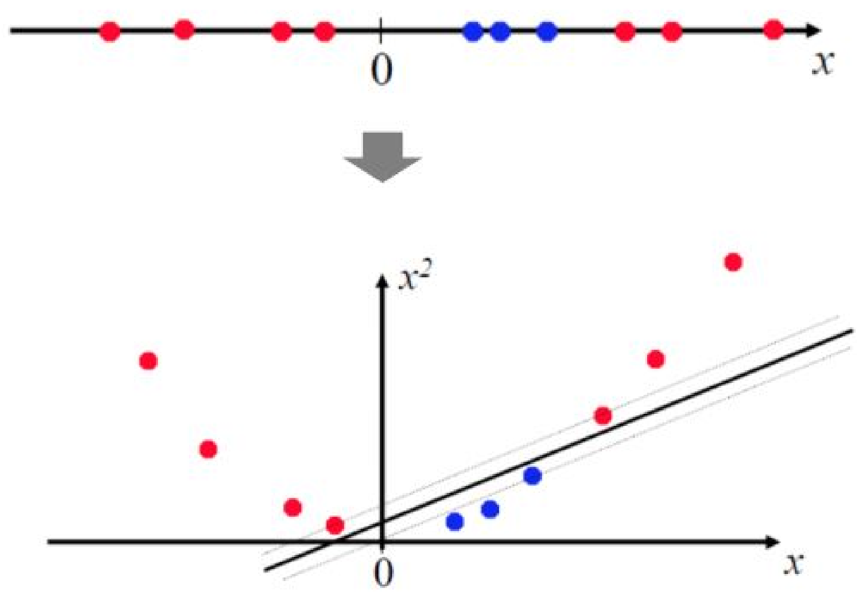
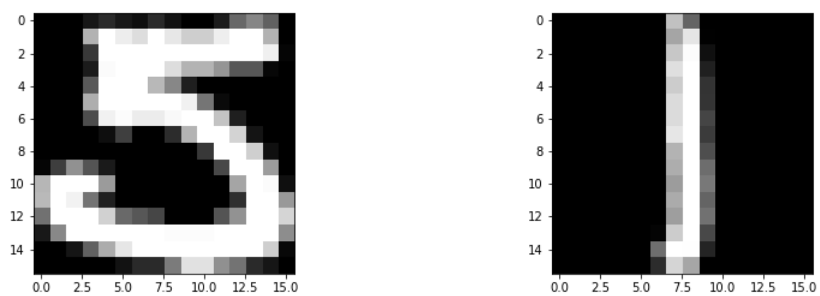


Figure 2: Mapping input vector to higher dimensional space to separate.

The overall computational work for SVM is high since the alphas must be individually recalculated based on the gradient and each cell of the kernel matrix does not depend on any other cells while being of dimension datapoints x datapoints. This leads to the SVM algorithm to be a prime candidate for the acceleration of a GPU in the CUDAFortran language. The dataset in use for the demonstrations of the algorithm is a set of 1560 images in gray scale values from -1 to 1 for 256 pixels. There is a corresponding label (1 or 5) that the algorithm will classify.



(a) A 5 datapoint plotted (b) A 1 datapoint plotted

Figure 3: Example data from both classes

Data Distribution

The data distribution component when working in cuda fortran has a distinct problem when working with cuda threads and the host cpu thread. The latentcy of a data transfer from the host to the device and vice versa can be a bottle neck if there are frequent small transfers of data. Ideally, any data that is sent to the gpu is never needed again in the host thread until the processing is done. In the first draft of the gpu code prior to data optimizations, the loss of each training epoch was being calculated on the cpu side. This is a problem for the execution speed of the cuda kernel as there exists a data dependency every epoch that must be fulfilled. By either removing this line or calculating the loss on the kernel will and did increase the speed of this program significantly.

The Memory profiling results from nvprof showed this and the visual profiler was showing a lot of GPU page faults and memory thrashing which can clearly be seen by the green bar in the image.

```
==150107== Unified Memory profiling result:
Device "GeForce RTX 2070 SUPER (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
  728004  27.323KB  4.0000KB  0.9961MB  18.97026GB  2.537655s  Host To Device
  131570  151.41KB  4.0000KB  0.9961MB  18.99832GB  1.650468s  Device To Host
   34283         -         -         -           -  6.111599s  Gpu page fault
       groups
     164  4.0000KB  4.0000KB  4.0000KB  656.0000KB          -  Memory thrashes
Total CPU Page faults: 65627
Total CPU thrashes: 164
```

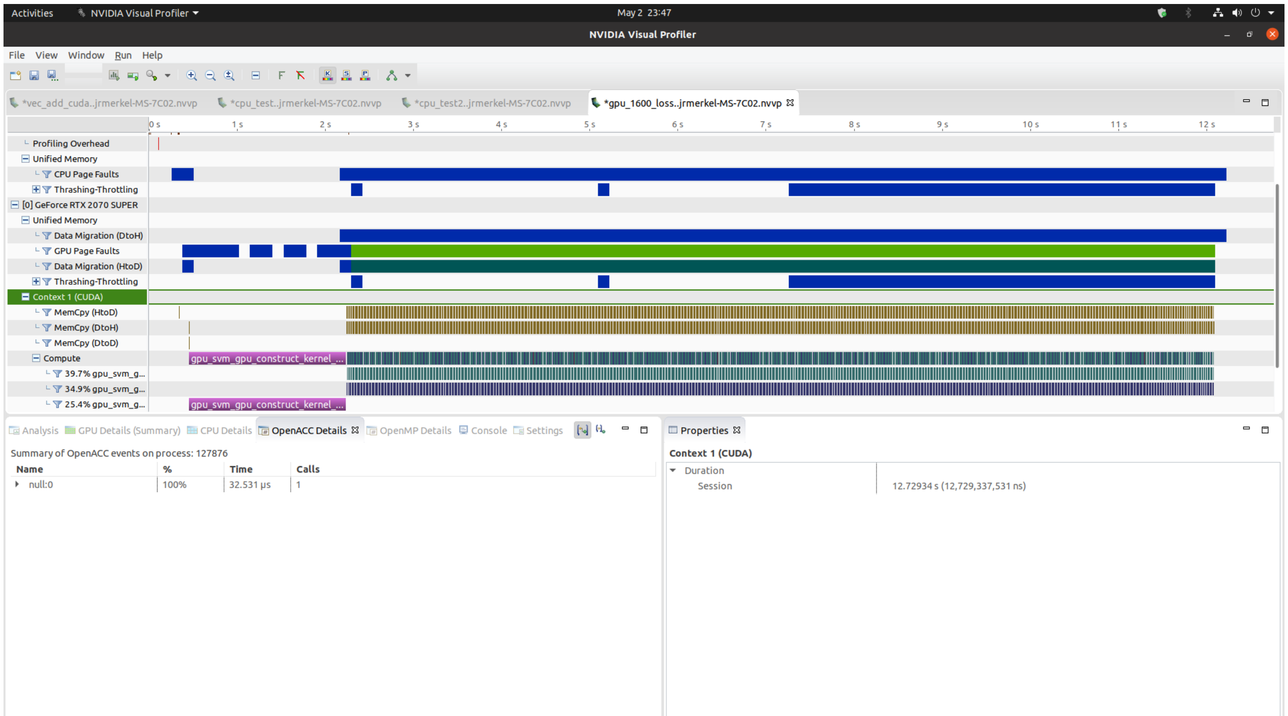


Figure 4: Nvidia Visual Profiler of Memory Problems

As we can see from the output, the amount of time that the program was spending transferring data was really high in comparison to the total execution time which was 12.729. Originally, I was very confused by such a high amount of transferred memory at almost 19GB. The dataset itself should only be a few MB so there had to be something. Eventually I noticed the loss calculation line was the only one that was not in a kernel. By removing that line the profiling results look like this instead.

```
==150202== Unified Memory profiling result:
Device "GeForce RTX 2070 SUPER (0)"
```

```
  Count  Avg Size Min Size Max Size Total Size Total Time Name
     74  141.84KB 4.0000KB 0.9922MB 10.25000MB 934.7800us Host To Device
    138  145.16KB 4.0000KB 0.9961MB 19.56250MB 1.812210ms Device To Host
    114         -        -        -          -  9.456863ms Gpu page fault
        groups
Total CPU Page faults: 123
```

Since the data is not being sent back to the CPU every epoch, that allows us to avoid the large latentcy associated with the gpu memory transfer. This means sending all of the data and keeping it there for the full training is really the only way to distribute the data for performance and most datasets will not exceed the amount memory on the GPU. What this looks like in the actual code is that we have managed arrays that are associated with the module that the kernel subroutines will modify.

```
module gpu_svm
    use cudafor
    use cublas
    real, save, managed, dimension(:,:), allocatable ::
        kernel_matrix,orig_dataset
    integer,save, managed:: rows, features, max_iter,
        num_cudathreads, type, num_ex,num_blocks
    real,save, managed :: learning_rate, C
    real, save, dimension(:), managed, allocatable :: labels
```

Then the cuda compiler will optimize when the memory is used so that we do not have to worry about the cuda memory functions.

Serial Optimization

The kernel construction function was originally implemented as a small subroutine that does depend on the loop index but is not many lines of code. This means that the subroutine call can use inlining to find more optimization opportunites. This is done by adding a compiler argument to the nvfortran compiler of -Minline=gpu_kernel which will allow the compiler to optimize out this call.

```
attributes(global) subroutine gpu_construct_kernel_matrix()
       real, dimension(rows) :: rowA, rowB

   do i=1, num_ex
       do j=1, rows

              rowA = orig_dataset(((i-1)*num_cudathreads)+
                 threadidx\%x + (blockidx\%x - 1) *num_blocks ,:)
              rowB = orig_dataset(j,:)

              call gpu_kernel(rowA, rowB,
                 kernel_matrix((i-1)*num_cudathreads +
                 threadidx\%x+ (blockidx\%x - 1) *num_blocks,j))
       enddo
```

```
    enddo

    end subroutine
```

In the performance section, we can see that the gpu_kernel function does not show up which means that this function is correctly being optimized out of its own call and is being inlined. Another aspect of the serial optimization was to pre-calculate the number of executions that each gpu thread would have to use so that each time the kernel is called, the gpu thread does not have to determine that value which can take some cycles off for each kernel call. Since there are 4 kernel calls per epoch and 1000 epochs this adds up to a sizeable time change. There is still a possibility to create stride 1 that would require a large refactor by storing the matrices in a transposed matrix.

Load Balancing

Load balancing is another aspect of using cuda fortran that should be considered. Each gpu process should be doing the same amount of work to ensure that the exectution is as fast as possible. In order to do this, any time a GPU kernel is used, the number of executions is the same and is either implicitly done by the cuda compiler if you use !cuf directive or explicitly in kernel subroutines. For the explicit kernel subroutines, I used num_ex which was pre-calculated to increase the serial speed and is calculated by the number of rows divided by the ceiling of the number of cuda threads. Then to determine which iteration the thread should be doing, we use (i-1)*num_cudathreads)+ threadidx%x + (blockidx%x - 1) *num_blocks to spread out the load evenly and still completing every iteration. Since cuda takes care of most of the load balancing this was not a significant amount of the project yet from the Nvidia Visual Profiler we can still see that all of the threads we are allocating are doing the same work.

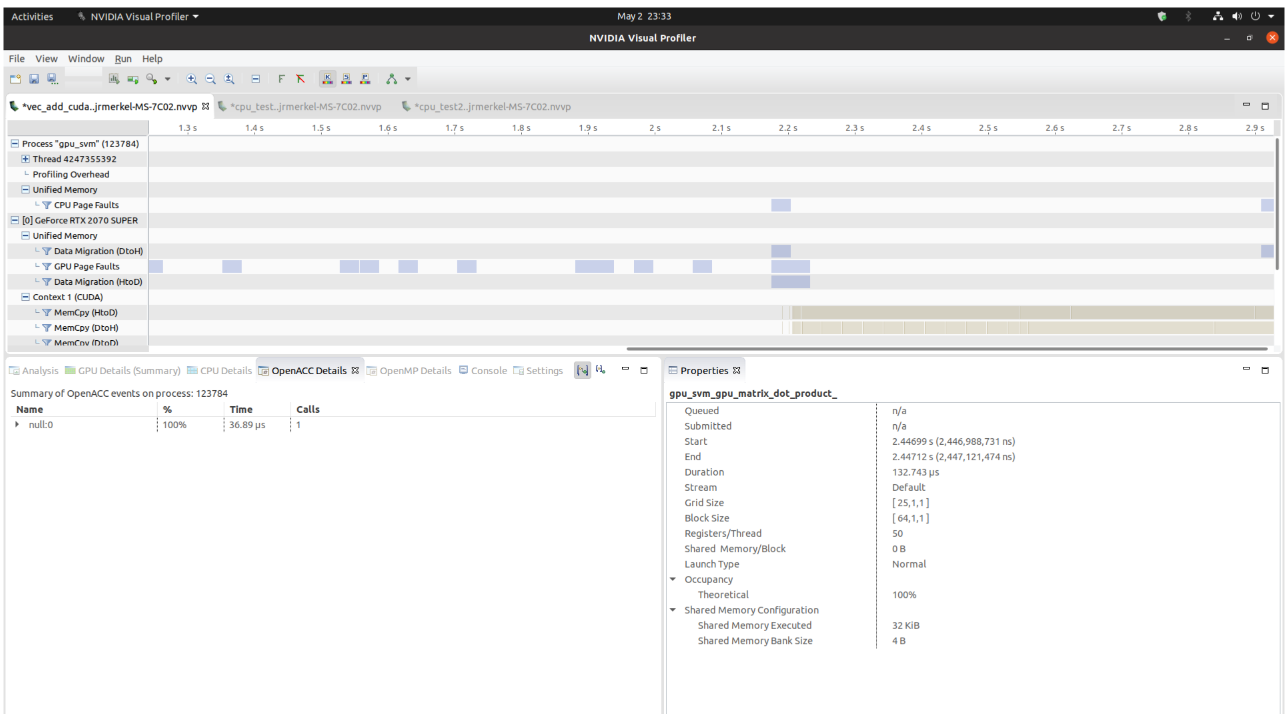


Figure 5: Nvidia Visual Profiler of 100% usage of threads

From this figure we can see that the 25 blocks of 64 threads are being executed with 100% occupancy of the threads.

Performance/Scalability

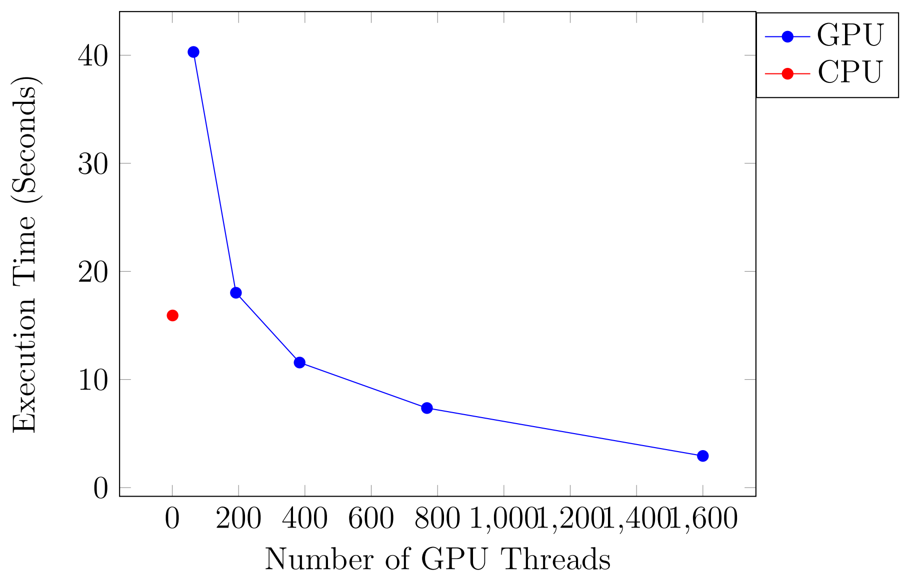


Figure 6: Execution Time by GPU Threads

GPU Threads	Execution time
64	40.31
192	18.026
384	11.569
768	7.36
1600	2.93

Figure 7: Table of GPU Execution Time

In terms of the GPU threads, the performance graph looks exactly how you would expect a highly parallel process to look. As we double the amount of GPU processes, the Execution time nearly halves. This is because nearly every aspect of the computation for the GPU version of the SVM algorithm is all in the various kernels. Of course as we increase the number of processes significantly, there still exists some time that gets wasted by overhead. The more threads you allocate, the more amount of overhead is added, along with the serial portions that cannot be optimized.

One thing that was done to improve the scalability is that all of the code for each epoch is done in the gpu kernel and the number of calls to the gpu kernel is minimized. Originally this snippet of code looked like the following.

```
  !$cuf kernel do <<<num_cudathreads/num_blocks, 64>>>
 do j = 1, rows
              gradient(j) = 1.d0 - y_x_alpha(j)
              alpha_arr(j)= alpha_arr(j) + learning_rate *
                 gradient(j)
enddo
istat = cudaDeviceSynchronize()
!For soft margin limit alpha by C
!$cuf kernel do <<<num_cudathreads/num_blocks, 64>>>
          do j = 1, rows
                 if(alpha_arr(j) > C) then
                    alpha_arr(j) = C
                 else if (alpha_arr(j) < 0) then
                    alpha_arr(j) = 0
                 endif
              enddo
```

```
          istat = cudaDeviceSynchronize()
```

One of the kernel calls can be removed by instead replacing it with this.

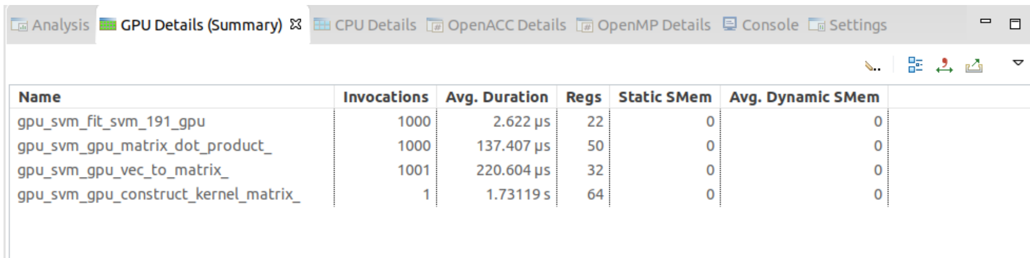
```
!$cuf kernel do <<<num_cudathreads/num_blocks, 64>>>
          do j = 1, rows
              gradient(j) = 1.d0 - y_x_alpha(j)
              alpha_arr(j)= alpha_arr(j) + learning_rate *
                  gradient(j)
!For soft margin limit alpha by C
              if(alpha_arr(j) > C) then
                  alpha_arr(j) = C
              else if (alpha_arr(j) < 0) then
                  alpha_arr(j) = 0
              endif
          enddo

          istat = cudaDeviceSynchronize()
```

This removes 1 overhead call to device synchronize and the setup of the kernel in general which will improve performance over the 1000 epochs that the program is ran. From the profiling report by nvprof, the time spent in the GPU activities and specifically in the subroutines that I created which are the gpu_matrix_dot_product, gpu_vec_to_matrix, and gpu_construct_kernel_matrix.

```
==150107== Profiling application: ./gpu_svm
==150107== Profiling result:
          Type  Time(\%)     Time     Calls      Avg       Min       Max  Name
 GPU activities: 41.41\% 2.89805s     1000  2.8981ms 2.1803ms 6.9813ms
     gpu_svm_gpu_matrix_dot_product_
                 34.55\% 2.41756s     1001  2.4151ms 1.1209ms 5.8735ms
                      gpu_svm_gpu_vec_to_matrix_
                 23.85\% 1.66932s        1  1.66932s 1.66932s 1.66932s
                      gpu_svm_gpu_construct_kernel_matrix_
                  0.09\% 6.0573ms     3002  2.0170us 1.1200us 5.7610us [CUDA memcpy DtoH]
                  0.06\% 4.0054ms     5014    798ns    736ns 2.6560us [CUDA memcpy HtoD]
                  0.04\% 2.4738ms     1000  2.4730us 2.2400us 3.5520us
                       gpu_svm_fit_svm_191_gpu
                  0.01\% 799.29us        2  399.64us 341.48us 457.81us [CUDA memcpy DtoD]
     API calls:  74.04\% 5.58224s     5003  1.1158ms 1.6500us 220.01ms cudaFree
                 22.50\% 1.69667s     3002  565.18us 1.1600us 1.66932s cudaDeviceSynchronize
                  0.70\% 52.625ms     5004  10.516us 3.2100us 2.4818ms cudaMemcpy
                  0.59\% 44.629ms     1000  44.628us 27.199us 78.148us cudaMemcpyAsync
                  0.55\% 41.327ms     1000  41.327us 5.4100us 383.19us cudaStreamSynchronize
                  0.54\% 41.087ms     2002  20.522us 10.890us 55.850us cudaMemcpyFromSymbol
                  0.38\% 28.821ms     5003  5.7600us 2.1200us 181.54us cudaMalloc
                  0.35\% 26.260ms     3002  8.7470us 4.3400us 233.09us cudaLaunchKernel
                  0.27\% 20.268ms        1  20.268ms 20.268ms 20.268ms cuMemAllocManaged
                  0.02\% 1.7507ms     2000    875ns    280ns 2.8900us
                       cudaDeviceGetAttribute
                  0.02\% 1.3691ms     1001  1.3670us    570ns 2.7100us cudaGetDevice
                  0.01\% 797.89us        1  797.89us 797.89us 797.89us cuMemHostAlloc
                  0.01\% 784.67us        3  261.56us 175.09us 390.27us cuDeviceTotalMem
                  0.00\% 241.72us      103  2.3460us    270ns 99.348us cuDeviceGetAttribute
                  0.00\% 183.38us       12  15.281us 6.3500us 68.859us cudaMemcpyToSymbol
                  0.00\% 35.429us        1  35.429us 35.429us 35.429us cuDeviceGetName
                  0.00\% 20.170us        1  20.170us 20.170us 20.170us cudaStreamCreate
                  0.00\% 5.8800us        1  5.8800us 5.8800us 5.8800us cudaSetDevice
                  0.00\% 5.2600us        1  5.2600us 5.2600us 5.2600us cuDeviceGetPCIBusId
```

```
                 0.00\% 3.6700us        1  3.6700us 3.6700us 3.6700us cudaDeviceSetLimit
                 0.00\% 2.5400us        4     635ns    370ns 1.1400us cuDeviceGetCount
                 0.00\% 2.0700us        1  2.0700us 2.0700us 2.0700us cudaGetFuncBySymbol
                 0.00\% 1.6900us        1  1.6900us 1.6900us 1.6900us cudaGetDeviceCount
                 0.00\% 1.5100us        1  1.5100us 1.5100us 1.5100us cuInit
                 0.00\% 1.3700us        3     456ns    280ns    710ns cuCtxSetCurrent
                 0.00\% 1.2700us        3     423ns    310ns    600ns cuDeviceGet
                 0.00\%    750ns        1     750ns    750ns    750ns cuFuncGetModule
                 0.00\%    650ns        1     650ns    650ns    650ns cudaGetSymbolAddress
                 0.00\%    630ns        2     315ns    200ns    430ns cudaRuntimeGetVersion
                 0.00\%    610ns        1     610ns    610ns    610ns cuCtxGetDevice
                 0.00\%    570ns        1     570ns    570ns    570ns cuCtxGetCurrent
                 0.00\%    470ns        1     470ns    470ns    470ns
                        cuDeviceComputeCapability
                 0.00\%    450ns        1     450ns    450ns    450ns cuDeviceGetUuid
                 0.00\%    350ns        1     350ns    350ns    350ns cuDriverGetVersion
                 0.00\%    220ns        1     220ns    220ns    220ns cudaDriverGetVersion
OpenACC (excl): 100.00\% 32.740us       1  32.740us 32.740us 32.740us acc_device_init
```

Since the GPU is spending most of its time in the highly parallel subroutines that I created, this project is highly scalable and in fact would see greater performance gains if the dataset was bigger and used more features. The amount of threads is almost overkill since from this snippet of the nvidia visual profiler we can see that the average call to the subroutines I wrote are less than a millisecond each and the cuf directive kernel is blindingly fast at 2.622 microseconds. If the dataset was instead much larger than the around 1600 samples in the training data, the setup for those kernels would be roughly the same but the durations would be much longer. This longer duration would mean more time is spent actually computing instead of going from kernel to kernel.



Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem
gpu_svm_fit_svm_191_gpu	1000	2.622 μs	22	0	0
gpu_svm_gpu_matrix_dot_product_	1000	137.407 μs	50	0	0
gpu_svm_gpu_vec_to_matrix_	1001	220.604 μs	32	0	0
gpu_svm_gpu_construct_kernel_matrix_	1	1.73119 s	64	0	0

Figure 8: Average Kernel Execution Time

The last thing I'll touch on in this section is that the SVM algorithm was very efficient at classifying the datapoints after modifying the learning rate so that our gradient descent did not overfit the training data. This resulted in a prediction of 98.58% accuracy in classifying between the 5 and 1 image inputs.

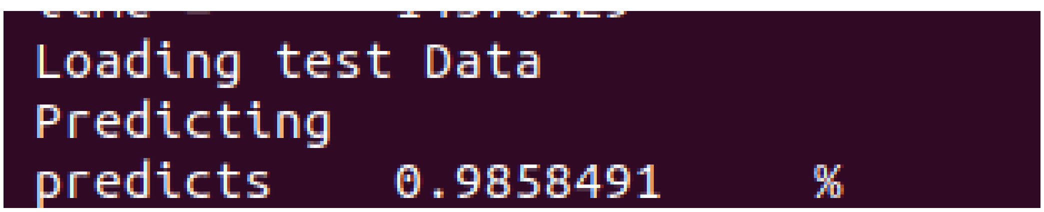


Figure 9: Code Output

Conclusion

Overall, the project was successful at implementing the SVM algorithm using cuda fortran and actually was able to classify the dataset with 98.58% accuracy. There are still more optimizations that could be made in the code that I did not have time to attempt. I think the efficacy of using a GPU as a hardware accelerator were clearly shown since the cpu speed was improved by a factor of 5.5x speed. This would likely be exacerbated by a larger dataset with more features. If I had more time I would refactor the code to use the transpose of the kernel matrix as well as implementing more kernel types other than just the linear kernel and polynomial kernel. A multivariate SVM would be one additional thing to increase the use cases of the algorithm in general.

References

- The dataset was taken from the COMS 573 HW on SVM and lecture notes from that class were also used.
- The implementation of SVM used help from http://www.adeveloperdiary.com/data-science/machine-learning/support-vector-machines-for-beginners-kernel-svm/
- Images from https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47

Code

CPU SVM

```
module svm
  contains

  subroutine kernel(x_i, x_j, type, transform)
    integer, intent(in):: type
    real, dimension(:) , intent(in):: x_i, x_j
    real, intent(out) :: transform
    real :: temp
    select case (type)
      !linear kernel x_i^T * x_j
      case(0)
        transform = dot_product(x_i,x_j)
      ! polynomial kernel (x_i^T * x_j)^2
      case(1)
        temp = (dot_product(x_i,x_j) + 1)
        transform = temp * temp * temp
      ! rbf kernel
      case(2)

      case default
        ! print*, "ERROR: NOT SUPPORTED KERNEL TYPE"
        ! call exit(0)
    end select
    return
  end subroutine

  subroutine construct_kernel_matrix_prediction(datasetA, datasetB, result, rows,
     features, type, result_dim)
    integer, intent(in) :: rows, features, type, result_dim
    real, dimension(:,:), intent(in) :: datasetA, datasetB
```

```fortran
    real, dimension(:,:), intent(out) :: result

    ! Perhaps there is a efficiency here either in dataset access or loop iters
    do i=1, result_dim
        do j=1, rows
            ! call kernel(dataset(i,1:features), dataset(j,1:features), type,
                 kernel_matrix(i,j))
            call kernel(datasetA(j, :), datasetB(i, :), type, result(i,j))
        enddo
    enddo
end subroutine



subroutine cpu_construct_kernel_matrix(dataset, kernel_matrix, rows, features, type)
    integer, intent(in) :: rows, features, type
    real, dimension(:,:), intent(in) :: dataset
    real, dimension(:,:), intent(out) :: kernel_matrix
    ! call syncthreads()
    ! Perhaps there is a efficiency here either in dataset access or loop iters
    do i=1, rows
        do j=1, rows
            ! call kernel(dataset(i,1:features), dataset(j,1:features), type,
                 kernel_matrix(i,j))
            call kernel(dataset(i,:), dataset(j,:), type, kernel_matrix(i,j))
        enddo
    enddo
end subroutine

subroutine vec_to_matrix(matrix, vector, n)
    integer, intent(in) :: n
    real, dimension (:), intent(in) :: vector
    real, dimension (:,:), intent(out) :: matrix
    integer :: i
    do i = 1, n
        do j = 1, n
```

```
            matrix(i,j) = vector(i) * vector(j)
        enddo
    enddo
end subroutine

subroutine matrix_dot_product(matrixA, VecB, n, outVector)
    real, dimension(:,:), intent(in) :: matrixA
    real, dimension(:), intent(in) :: VecB
    integer, intent(in) :: n
    real, dimension(:), intent(out) :: outVector

    real :: acc
    integer :: i
    do i = 1, n
        acc = 0.d0
        do j = 1,n
            acc = acc + (matrixA(i,j) * VecB(j))
        enddo
        outVector(i) = acc
    enddo

end subroutine

subroutine fit_svm(kernel_matrix, labels, rows, features, max_iter, learning_rate,
     alpha_arr, loss, bias)
    !Inputs
    integer, intent(in) :: rows, features, max_iter
    real, intent(in) :: learning_rate
    real, dimension(:), intent(in) :: labels
    real, dimension(:,:), intent(in) :: kernel_matrix
    !Local Vars
    real, dimension(:,:), allocatable :: labels_matrix
    integer :: i, j
    real   :: C
    real, dimension(:,:), allocatable :: y, alpha_matrix
```

```fortran
real, dimension(:), allocatable :: bias_arr, y_x_alpha, gradient
! Inout since alphas change
real, dimension(:), intent(out) :: alpha_arr
real, intent(inout) :: bias
! Outputs
real, dimension(:), intent(out) :: loss
print*, "Init alphas"
C = 1.d0
! Randomize initial alphas
call random_number(alpha_arr(1:rows))
print*, "Allocating"
! Init bias, ones, and gradient
allocate(gradient(rows))
bias = 0.d0
allocate(bias_arr(rows))
bias_arr(1:rows) = 0.d0
allocate(y(rows, rows))
allocate(labels_matrix(rows,rows))
allocate(alpha_matrix(rows,rows))
!Temp vars
allocate(y_x_alpha(rows))

! calculate matrix form of labels
call vec_to_matrix(labels_matrix, labels, rows)

y(1:rows, 1:rows) = labels_matrix(1:rows, 1:rows) * kernel_matrix(1:rows, 1:rows)
! Training
do i = 1, max_iter
   print*, "Epoch = ", i
   ! Calculate gradient and update alpha
   call matrix_dot_product(y, alpha_arr, rows, y_x_alpha))
   gradient = 1.d0 - y_x_alpha
   alpha_arr= alpha_arr+ learning_rate * gradient

   !For soft margin limit alpha by C
```

```
        do j = 1, rows
            if(alpha_arr(j) > C) then
                alpha_arr(j) = C
            else if (alpha_arr(j) < 0) then
                alpha_arr(j) = 0
            endif
        enddo
        ! Loss = sum alpha - 1/2 sum_i sum_j alphasij yij K Not doing to keep
             consistent with GPU
        !call vec_to_matrix(alpha_matrix, alpha_arr, rows)
        !loss(i) = sum(alpha_arr) - 0.5 * sum(alpha_matrix * y)
    enddo

    !Calc bias = avg if alpha within range, yi - aiyiK
    y_x_alpha(1:rows) = alpha_arr(1:rows) * labels(1:rows)
    ! call matrix_dot_product(kernel_matrix, alpha_x_y, rows, alpha_x_y_dot_kernel)

    do j = 1, rows
        if(alpha_arr(j) > 0 .and. alpha_arr(j) < C) then
            bias_arr(j) = labels(j) - dot_product(y_x_alpha, kernel_matrix(:, j))
        endif
    enddo

    bias = sum(bias_arr) / rows

end subroutine

!yhat = sign(sum alphai yi kernel(xi)T kernel(zi) + b)
!Figure out how to make modules save arrays
subroutine predict(orig_dataset, orig_labels, orig_rows, features, alpha_arr, bias,
     samples, samples_dim, type, predictions)
    !!!!!!!!!!!!!!!!!!!!!!Inputs!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    real, dimension(:,:), intent(in) :: orig_dataset, samples
    real, dimension(:), intent(in)   :: orig_labels, alpha_arr
    real, intent(in)                 :: bias
```

```
    integer, intent(in)              :: orig_rows, features, samples_dim, type
    !!!!!!!!!!!!!!!!!!!!!!Outputs!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    integer, dimension (:), intent(out) :: predictions
    !!!!!!!!!!!!!!!!!!!!!!Local!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    real, dimension(:), allocatable             :: y_x_alpha
    real, dimension(:,:), allocatable           :: result

    allocate(y_x_alpha(orig_rows))
    allocate(result(samples_dim, orig_rows))
    y_x_alpha(1:orig_rows) = alpha_arr(1:orig_rows) * orig_labels(1:orig_rows)

    call construct_kernel_matrix_prediction(orig_dataset, samples, result, orig_rows,
         features, type, samples_dim)
    do i = 1, samples_dim
       predictions(i) = sign_func(dot_product(y_x_alpha, result(i,:)))
    enddo

end subroutine

integer function sign_func(input) result (out)
    real :: input

    if(input > 0) then
       out = 1
    else
       out = -1
    endif
end function

real function score(labels, prediction, n) result (out)
    integer, dimension(:) :: prediction
    real, dimension(:)  :: labels
    integer             :: i, num_correct
    num_correct = 0
    do i = 1, n
```

```
            if(int(labels(i)) == prediction(i)) then
                num_correct = num_correct + 1
            endif
        enddo
        out = dble(num_correct) / dble(n)
    end function

end module svm

!!!!!!!!!!!!!!!!!!!!!!!!!Main!!!!!!!!!!!!!!!!!!
use cudafor
use cublas
use svm
use data_load
integer, parameter :: kernel_type = 1, n = 400, max_iter = 1000
real, parameter :: learning_rate = 0.0000000002, C = 5.0
integer :: t1,t2,t3,t4
real, dimension(:,:), allocatable :: dataset, raw_dataset, test_dataset, raw_test_dataset
real, dimension(:,:), allocatable :: kernel_matrix
real, dimension(:), allocatable :: labels, alpha_arr, loss, test_labels
integer, dimension(:), allocatable :: predictions
integer :: rows, features, count_max, count_rate, test_rows
real :: bias
!Setup the cuda devices
istat = cudaSetDevice(0)
call system_clock(count_max = count_max, count_rate=count_rate)
rows =
     determine_row_num("/home/jrmerkel/Documents/Fortran/SVM_CudaFortran/SVM_Example/data/train.txt")
features =
     determine_feature_size("/home/jrmerkel/Documents/Fortran/SVM_CudaFortran/SVM_Example/data/train.txt")
device_rows = rows
device_features = features
! allocate the dataset and kernel matrices on host and devices
allocate(raw_dataset(rows,features))
```

```fortran
print*, "Loading Dataset"

call
    load_data("/home/jrmerkel/Documents/Fortran/SVM_CudaFortran/SVM_Example/data/train.txt",
    raw_dataset, rows, features)

! allocate arr and matrices
allocate(labels(rows))
allocate(kernel_matrix(rows,rows))
allocate(alpha_arr(rows))
allocate(loss(max_iter))

print*, "Parsing labels"
call split_labels(raw_dataset, dataset, labels, rows, features)
call normalize_labels(labels, rows)

print*, "Constructing Kernel"
call cpu_construct_kernel_matrix(dataset, kernel_matrix, rows, features, kernel_type)

print*, "fit Svm"
call fit_svm(kernel_matrix, labels, rows, features, max_iter, learning_rate, alpha_arr,
    loss, bias)

! load test dataset
test_rows =
    determine_row_num("/home/jrmerkel/Documents/Fortran/SVM_CudaFortran/SVM_Example/data/test.txt")
! allocate the dataset
allocate(raw_test_dataset(test_rows,features))
allocate(test_dataset(test_rows,features))
allocate(predictions(test_rows))
print*, "Loading test Data"

call
    load_data("/home/jrmerkel/Documents/Fortran/SVM_CudaFortran/SVM_Example/data/test.txt",
    raw_test_dataset, test_rows, features)
```

```
print*, "Predicting"
! Get the prediction array
allocate(test_labels(test_rows))

call split_labels(raw_test_dataset, test_dataset, test_labels, test_rows, features)
call normalize_labels(test_labels, test_rows)

call predict(dataset, labels, rows, features, alpha_arr, bias, test_dataset, test_rows,
     kernel_type, predictions)
print*, "Writing"

! Score
print*, "predicts ", score(test_labels, predictions, test_rows), "%"

end
```

Data Load Helper

```
module data_load
   integer :: MAX_FEATURE_SIZE = 260
   integer :: MAX_LINE_CHARACTER_LEN = 4500
   integer :: FILE_UNIFIER = 20
   real :: DUMMY_VAL = -99999.d22
   contains
   ! NOTE this subroutine reads in the data is (row,col)
   ! Consider Transposing for efficiency
   subroutine load_data(filepath, data_array, row_num, feature_size)
      character(len=*), intent(in) :: filepath
      integer, intent(in) :: row_num, feature_size
      real, dimension(:,:), intent(out) :: data_array
      ! Read in the data
      open(FILE_UNIFIER, FILE = filepath)
      do i = 1, row_num
```

```fortran
            read(FILE_UNIFIER, *) data_array(i, 1:feature_size)
        enddo
        ! Transpose matrix Fortran is Col
        !transpose(data_array)
        close(FILE_UNIFIER)
    end subroutine

! Changes the labels to be 1 or -1 so that SVM can be run
    subroutine normalize_labels(label_array, n)
        real, dimension(:), intent(inout) :: label_array
        integer, intent(in) :: n
        integer :: i

        do i = 1, n
            if(label_array(i) < 1.00001 .and. label_array(i) > 0.9999) then
                label_array(i) = 1.d0
            else
                label_array(i) = -1.d0
            endif
        enddo

    end subroutine

!splits the labels from the dataset because that needs to be separate to train/test the
     data
    subroutine split_labels(data_array, feature_array, label_array, row_num, feature_size)
        real, dimension(:,:), intent(in) :: data_array
        integer, intent(in) :: row_num, feature_size
        real, dimension(:,:), intent(out) :: feature_array
        real, dimension(:), intent(out) :: label_array

        label_array = data_array(1:row_num, 1)
        feature_array(1:row_num, 1:feature_size) = data_array(1:row_num, 1:feature_size)
        feature_array(1:row_num,1) = 1.d0
    end subroutine
```

```
!Determines the number of datapoints
   integer function determine_row_num(filepath) result (row)
      integer :: row, i, ios
      character(len=*) :: filepath
      character(len=200) :: huh
      real, dimension(2000) :: data_array
      ! Open file
      open(FILE_UNIFIER, FILE = filepath)
      row = 0
      do i=1, 2000
         read(FILE_UNIFIER, '(ES10.2)', iostat=ios), data_array(i)
         if (ios /= 0) then
            close(FILE_UNIFIER)
            return
         endif
         row = row + 1
      enddo
      row = -1
      return

   end function

! Determines how many features are in use
   integer function determine_feature_size(filepath) result (feature)
      integer :: feature, i, ios
      character(len=*) :: filepath
      character(len=MAX_LINE_CHARACTER_LEN) :: Line
      real, dimension(MAX_FEATURE_SIZE) :: data_array
      ! Open file
      open(FILE_UNIFIER, FILE = filepath)
      feature = 0
      !init dummy vals
      data_array(1:MAX_FEATURE_SIZE) = DUMMY_VAL
      !Read line and then into array
```

```
        read(FILE_UNIFIER, '(A)'), Line
        read(Line, *, iostat=ios) data_array

        do i = 1, MAX_FEATURE_SIZE
            if(data_array(i)/= DUMMY_VAL) then
                feature = feature + 1
            else
                close(FILE_UNIFIER)
                return
            endif
        enddo
        close(FILE_UNIFIER)
        return
    end function

end module
```

GPU SVM

```
module gpu_svm
    use cudafor
    use cublas
    real, save, managed, dimension(:,:), allocatable :: kernel_matrix,orig_dataset
    ! real, save, dimension(:,:), managed, allocatable :: test
    integer,save, managed:: rows, features, max_iter, num_cudathreads, type,
         num_ex,num_blocks
    real,save, managed :: learning_rate, C
    real, save, dimension(:), managed, allocatable :: labels
    contains
!impelments the row by row kernel calculations in gpu
    attributes(device) subroutine gpu_kernel(x_i, x_j, transform)
        ! integer, device, intent(in):: type
        real, device, dimension(:) , intent(in):: x_i, x_j
```

```
        real, device, intent(out) :: transform
        real, device:: temp
        select case (type)
            !linear kernel x_i^T * x_j
            case(0)
                transform = dot_product(x_i,x_j)
            ! polynomial kernel (x_i^T * x_j)^2
            case(1)
                temp = (dot_product(x_i,x_j) + 1)
                transform = temp * temp * temp
            case default
                ! print*, "ERROR: NOT SUPPORTED KERNEL TYPE"
                ! call exit(0)
        end select
        return
    end subroutine

!impelments the row by row kernel calculations in cpu
    subroutine kernel(x_i, x_j, type, transform)
        integer, intent(in):: type
        real, dimension(:) , intent(in):: x_i, x_j
        real, intent(out) :: transform
        real :: temp
        select case (type)
            !linear kernel x_i^T * x_j
            case(0)
                transform = dot_product(x_i,x_j)
            ! polynomial kernel (x_i^T * x_j)^2
            case(1)
                temp = (dot_product(x_i,x_j) + 1)
                transform = temp * temp * temp
            ! rbf kernel
            case(2)

            case default
```

```
            ! print*, "ERROR: NOT SUPPORTED KERNEL TYPE"
            ! call exit(0)
        end select
        return
    end subroutine
    !Constructs the kernel matrix for the kernel trick of svm
    attributes(global) subroutine gpu_construct_kernel_matrix()
        real, dimension(rows) :: rowA, rowB

    do i=1, num_ex
        do j=1, rows

            rowA = orig_dataset(((i-1)*num_cudathreads)+ threadidx%x + (blockidx%x - 1)
                 *num_blocks ,:)
            rowB = orig_dataset(j,:)
            call gpu_kernel(rowA, rowB, kernel_matrix((i-1)*num_cudathreads + threadidx%x+
                 (blockidx%x - 1) *num_blocks,j))

        enddo
    enddo

    end subroutine

!Utility function to test kernel
    attributes(host) integer function write_kernel() result (out)

        open(23,FILE = "./orig_data.txt")
      write(23, *), orig_dataset(1,:)
      close(23)
      out = 1
      return
    end function
```

```
    subroutine construct_kernel_matrix_prediction(datasetA, datasetB, result, result_dim)
        integer, intent(in) :: result_dim
        real, dimension(:,:), intent(in) :: datasetA, datasetB
        real, dimension(:,:), intent(out) :: result

        do i=1, result_dim
            do j=1, rows
                ! call kernel(dataset(i,1:features), dataset(j,1:features), type,
                     kernel_matrix(i,j))
                call kernel(datasetA(j, :), datasetB(i, :), type, result(i,j))
            enddo
        enddo
    end subroutine

!Implement Np.outer for vector
    attributes(global) subroutine gpu_vec_to_matrix(matrix, vector)
        real, device,dimension (:), intent(in) :: vector
        real, device,dimension (:,:), intent(out) :: matrix
        integer :: i,j
        !num_ex = ceiling(real(rows) / real(num_cudathreads))
    do j = 1, num_ex
        ! print*, " j = ", (j-1)*(num_cudathreads) + ((blockidx%x - 1) *num_blocks) +
             threadidx%x
              do i = 1, rows
                  matrix(i, (j-1)*(num_cudathreads) + ((blockidx%x - 1) *num_blocks) +
                       threadidx%x) = vector(i) * vector((j-1)*(num_cudathreads) +
                       ((blockidx%x - 1) *num_blocks) + threadidx%x)
      !  matrix(i,((j-1)*(num_cudathreads + (blockidx%x - 1) *num_blocks)+ threadidx%x)) =
           vector(i) * vector((j-1)*(num_cudathreads + (blockidx%x - 1) *num_blocks) +
           threadidx%x)
              enddo
          enddo
    end subroutine

!Dot product of a matrix and a vector
```

```fortran
attributes(global) subroutine gpu_matrix_dot_product(matrixA, VecB, outVector)
    real, device, dimension(:,:), intent(in) :: matrixA
    real, device, dimension(:), intent(in) :: VecB
    real, device, dimension(:), intent(out) :: outVector

    real :: acc
    integer :: i

! print*, "blockidx", blockidx%x, "threadidx", threadidx%x
    do i = 1, num_ex
        acc = 0.d0
        do j = 1, rows
            acc = acc + (matrixA((i-1)*num_cudathreads + threadidx%x+ (blockidx%x - 1)
                 *num_blocks,j) * VecB(j))
        enddo
        outVector((i-1)*num_cudathreads + threadidx%x+ (blockidx%x - 1) *num_blocks) =
             acc
    enddo

end subroutine

attributes(host) subroutine fit_svm(alpha_arr, loss, bias)
    !Inputs


    !Local Vars
    real, dimension(:,:), managed, allocatable :: labels_matrix
    integer :: i, j
    real, dimension(:,:), managed, allocatable :: y, alpha_matrix
    real, dimension(:), managed, allocatable :: bias_arr, y_x_alpha, gradient
    ! Inout since alphas change
    real, dimension(:), managed, intent(out) :: alpha_arr
    real, intent(inout) :: bias
    ! Outputs
    real, dimension(:), intent(out) :: loss
```

```
    print*, "Init alphas"
    ! Randomize initial alphas
    call random_number(alpha_arr(1:rows))

    ! Init bias, ones, and gradient
    allocate(gradient(rows))

    bias = 0.d0
    allocate(bias_arr(rows))

    bias_arr(1:rows) = 0.d0
    allocate(y(rows, rows))
    allocate(labels_matrix(rows,rows))
    allocate(alpha_matrix(rows,rows))

    !Temp vars
    allocate(y_x_alpha(rows))

    ! calculate matrix form of labels
    call gpu_vec_to_matrix<<<num_cudathreads/num_blocks,num_blocks>>>(labels_matrix,
         labels)

istat = cudaDeviceSynchronize()
    y(1:rows, 1:rows) = labels_matrix(1:rows, 1:rows) * kernel_matrix(1:rows, 1:rows)

    ! Training
    do i = 1, max_iter
        print*, "Epoch = ", i
        ! Calculate gradient and update alpha
        istat = cudaDevuceSynchronize
   call gpu_matrix_dot_product<<<num_cudathreads/num_blocks, num_blocks>>>(y,
        alpha_arr, y_x_alpha)

        istat = cudaDeviceSynchronize()
       !$cuf kernel do <<<num_cudathreads/num_blocks, 64>>>
```

```
        do j = 1, rows
           gradient(j) = 1.d0 - y_x_alpha(j)
           alpha_arr(j)= alpha_arr(j) + learning_rate * gradient(j)
           if(alpha_arr(j) > C) then
              alpha_arr(j) = C
           else if (alpha_arr(j) < 0) then
              alpha_arr(j) = 0
           endif
        enddo

        istat = cudaDeviceSynchronize()
        ! Loss = sum alpha - 1/2 sum_i sum_j alphasij yij K Loss is not necessary to
             calculate
        !call gpu_vec_to_matrix<<<num_cudathreads/num_blocks,
             num_blocks>>>(alpha_matrix, alpha_arr)
   ! loss(i) = sum(alpha_arr) - 0.5 * sum(alpha_matrix * y)
     enddo
     !Calc bias = avg if alpha within range, yi - aiyiK
     y_x_alpha(1:rows) = alpha_arr(1:rows) * labels(1:rows)
     print*, "bias calc"
     do j = 1, rows
        if(alpha_arr(j) > 0 .and. alpha_arr(j) < C) then
           bias_arr(j) = labels(j) - dot_product(y_x_alpha, kernel_matrix(:, j))
        endif
     enddo

     bias = 1.d0

  end subroutine

  !yhat = sign(sum alphai yi kernel(xi)T kernel(zi) + b)

!Predict for the incoming dataset
  subroutine predict(original_dataset, orig_labels, orig_rows, alpha_arr, bias, samples,
       samples_dim, predictions)
```

```
      !!!!!!!!!!!!!!!!!!!!!!Inputs!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      real, dimension(:,:), intent(in) :: original_dataset, samples
      real, dimension(:), intent(in)   :: orig_labels, alpha_arr
      real, intent(in)                 :: bias
      integer, intent(in)              :: orig_rows, samples_dim
           !!!!!!!!!!!!!!!!!!!!!Outputs!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      integer, dimension (:), intent(out) :: predictions
      !!!!!!!!!!!!!!!!!!!!!!Local!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      real, dimension(:), allocatable            :: y_x_alpha
      real, dimension(:,:), allocatable          :: result

      allocate(y_x_alpha(orig_rows))
      allocate(result(samples_dim, orig_rows))
!Calculate alpha * Labels
      y_x_alpha(1:orig_rows) = alpha_arr(1:orig_rows) * orig_labels(1:orig_rows)

      call construct_kernel_matrix_prediction(original_dataset, samples, result,
           samples_dim)
      do i = 1, samples_dim
         predictions(i) = sign_func(dot_product(y_x_alpha, result(i,:)))
      enddo

   end subroutine

!Sign function is used to determine which label is the prediction.
   integer function sign_func(input) result (out)
      real :: input

      if(input > 0) then
         out = 1
      else
         out = -1
      endif
   end function
```

```
!Score the predictions
   real function score(labels, prediction, n) result (out)
      integer, dimension(:) :: prediction
      real, dimension(:)  :: labels
      integer             :: i, num_correct, n
      num_correct = 0
      do i = 1, n
         if(int(labels(i)) == prediction(i)) then
            num_correct = num_correct + 1
         endif
      enddo
      out = dble(num_correct) / dble(n)
   end function

!Allocate datasets
   integer function allc() result (out)
      allocate(kernel_matrix(rows,rows))
  allocate(labels(rows))
  allocate(orig_dataset(rows, features))

   end function

!Utility function to test
   attributes(host) integer function write_dataset(in_data, in_labels) result (out)
  real, dimension(:,:) :: in_data
  real,dimension (:) :: in_labels
      labels = in_labels
  orig_dataset = in_data
   end function

!Sets the dimensional values that will be used in fit
   integer function set_dim(r, f, m, num_cuda, lr, svmC, t, tpb) result(out)
      integer :: r,f, m, num_cuda, t, tpb
      real :: lr, svmC
      rows = r
```

```
        features = f
        max_iter = m
        num_cudathreads = num_cuda
        learning_rate = lr
        C = svmC
        type = t
        num_ex = ceiling(real(rows) / real(num_cudathreads))
    num_blocks = tpb
     end function
end module gpu_svm
! integer, parameter :: kernel_type = 1, n = 700, max_iter = 10
! real, parameter :: learning_rate = 0.0000000002, C = 5.0
!!!!!!!!!!!!!!!!!!!!!!!!!Main!!!!!!!!!!!!!!!!!!
use cudafor
use cublas
use gpu_svm
use data_load
implicit none
integer, parameter :: kernel_type = 1, main_n = 1600, main_max_iter = 1000, main_type = 1,
     threadsPerBlock = 64
real, parameter :: main_learning_rate = 0.0000000002, main_C = 5.0
integer :: t1,t2,t3,t4
real, dimension(:,:), allocatable :: dataset, raw_dataset, test_dataset, raw_test_dataset
! real, device, dimension(:,:), allocatable :: device_kernel_matrix
real, dimension(:), allocatable :: main_labels, main_alpha_arr, main_loss, test_labels
integer, dimension(:), allocatable :: predictions
! real, allocatable, device :: device_dataset(:,:)
! real, allocatable, device :: device_kernel_matrix(:,:)
integer :: main_rows, main_features, count_max, count_rate, test_rows
integer :: istat
integer(8) :: heap
real :: main_bias

!Increase heap size
heap = 16 * 2000 * 2000
```

```
!Setup the cuda devices
istat = cudaSetDevice(0)
istat = cudaDeviceSetLimit(cudaLimitMallocHeapSize, heap)

print*, istat
call system_clock(count_max = count_max, count_rate=count_rate)
main_rows =
     determine_row_num("/home/jrmerkel/Documents/Fortran/SVM_CudaFortran/SVM_Example/data/train.txt")
main_features =
     determine_feature_size("/home/jrmerkel/Documents/Fortran/SVM_CudaFortran/SVM_Example/data/train.txt")
istat = set_dim(main_rows, main_features, main_max_iter, main_n, main_learning_rate,
     main_C, main_type, threadsPerBlock)
istat = allc()
! allocate the dataset and kernel matrices on host and devices
allocate(raw_dataset(main_rows,main_features))
print*, "main_rows", main_rows
print*, "Loading Dataset"

call
     load_data("/home/jrmerkel/Documents/Fortran/SVM_CudaFortran/SVM_Example/data/train.txt",
     raw_dataset, main_rows, main_features)
! allocate arr and matrices
allocate(dataset(main_rows,main_features))
allocate(main_labels(main_rows))
allocate(main_alpha_arr(main_rows))
allocate(main_loss(main_max_iter))

call split_labels(raw_dataset, dataset, main_labels, main_rows, main_features)
call normalize_labels(main_labels, main_rows)

call gpu_construct_kernel_matrix <<<main_n/num_blocks, num_blocks>>>()

istat = cudaDeviceSynchronize()
istat = write_kernel()
```

```
print*, "fit Svm"
call fit_svm( main_alpha_arr, main_loss, main_bias)

call system_clock(t2)
print*, "time = ", t2-t1
print*, "fit done"
! load test dataset
test_rows =
    determine_row_num("/home/jrmerkel/Documents/Fortran/SVM_CudaFortran/SVM_Example/data/test.txt")
! allocate the dataset
allocate(raw_test_dataset(test_rows,main_features))
allocate(test_dataset(test_rows,main_features))
allocate(predictions(test_rows))
print*, "Loading test Data"

call
    load_data("/home/jrmerkel/Documents/Fortran/SVM_CudaFortran/SVM_Example/data/test.txt",
    raw_test_dataset, test_rows, main_features)
print*, "Predicting"
! Get the prediction array
allocate(test_labels(test_rows))

call split_labels(raw_test_dataset, test_dataset, test_labels, test_rows, main_features)
call normalize_labels(test_labels, test_rows)

istat = cudaGetLastError()
call predict(dataset, main_labels, main_rows, main_alpha_arr, main_bias, test_dataset,
    test_rows, predictions)
print*, "Writing"

istat = cudaGetLastError()
print *, cudaGetErrorString(istat)
! Score
print*, "predicts ", score(test_labels, predictions, test_rows), "%"
```

```
istat = cudaGetLastError()
print *, cudaGetErrorString(istat)

end
```

Unit tests

```
module unittests
   use svm
   use gpu_svm
   use cudafor
contains

   integer function test_matrix_dot_product() result (out)

      real, dimension(3,3) :: matrixA
      real, dimension(3) :: VecB, outVector
      integer :: n, i, j
      n = 3
      do i = 1, n
         do j = 1, n
            matrixA(i,j) = i*3 -3 + j
         enddo
         VecB(i) = i + 1
      enddo
      call matrix_dot_product(matrixA, VecB, n, outVector)
      if(outVector(1) == 20.00000 .and. outVector(2) == 47.00000 .and. outVector(3) ==
          74.00000) then
         out = 1
      else
         out = 0
      endif
```

```
end function

integer function test_vect_to_matrix() result (out)

real, dimension(3,3) :: outMatrix
real, dimension(3) :: VecA
integer :: n, i, j
n = 3
do i = 1, n
   VecA(i) = i + 1
enddo
call vec_to_matrix(outMatrix, VecA, n)
do i = 1, n
   if(outMatrix(i,1) == (i+1) * 2 .and. outMatrix(i,2) == (i+1) * 3 .and.
        outMatrix(i,3) == (i+1) * 4) then
      out = 1
   else
      out = 0
      return
   endif
enddo
end function

attributes(host) integer function gpu_test_vect_to_matrix() result (out)

real,managed, dimension(3,3) :: outMatrix
real,managed, dimension(3) :: VecA
integer :: n, i, j
n = 3
do i = 1, n
   VecA(i) = i + 1
enddo
x = set_dim(3, 3, 1, 10, 0.001, 1.00, 1, 1)
```

```
call gpu_vec_to_matrix<<<1,3>>>(outMatrix, VecA)
do i = 1, n
   if(outMatrix(i,1) == (i+1) * 2 .and. outMatrix(i,2) == (i+1) * 3 .and.
       outMatrix(i,3) == (i+1) * 4) then
      out = 1
   else
      out = 0
      return
   endif
enddo
end function

attributes(host) integer function gpucpu_test_vect_to_matrix() result (out)

real,managed, dimension(77,77) :: outMatrix_cpu, outMatrix_gpu
real,managed, dimension(77) :: VecA

integer :: n, i, j
n = 77
call random_number(VecA(i:n))
call gpu_vec_to_matrix<<<5,10>>>(outMatrix_gpu, VecA)
call vec_to_matrix(outMatrix_cpu, VecA, n)
i = cudaDeviceSynchronize()
do i = 1, n
   do j = 1, n

  if(outMatrix_gpu(j,i) == outMatrix_cpu(j,i)) then
     out = 1
  else
     print*, j, i
     out = 0
     return
  endif

enddo
```

```
 enddo
 end function

 attributes(host) integer function gpucpu_test_matrix_dot() result (out)

 real,managed, dimension(77) :: outVector_cpu, outVector_gpu
 real,managed, dimension(77) :: VecB
real, managed, dimension(77,77) :: MatrixA
 integer :: n, i, j
 n = 77
  do i = 1, n
     Vecb(i) = i + 1
do j = 1, n
   MatrixA(j,i) = i+1
enddo
 enddo
 i = cudaDeviceSynchronize()

i = cudaGetLastError()
print *, cudaGetErrorString(i)
 call gpu_matrix_dot_product<<<5,10>>>(MatrixA,VecB,outVector_gpu)
 call matrix_dot_product(MatrixA,VecB,n,outVector_cpu)
 i =  cudaDeviceSynchronize()

i = cudaGetLastError()
print *, cudaGetErrorString(i)
 do i = 1, n
if(outVector_gpu(i) == outVector_cpu(i)) then
   out = 1
else
   out = 0
   return
endif
 enddo
 end function
```

```
end module

use svm
use gpu_svm
use unittests
integer :: x,y
x = cudaSetDevice(0)
y = set_dim(3, 3, 1, 10, 0.001, 1.00, 1, 1)

x = 1
if(test_matrix_dot_product() == 0) then
   print*, "Matrix Dot Product fails"
   x=0
endif

if(test_vect_to_matrix() == 0) then
   print*, "Vect to Matrix (np.outer) fails"
   x = 0
endif
y =  set_dim(77, 3, 1, 10, 0.001, 1.00, 1,1)

if(gpu_test_vect_to_matrix() == 0) then
   print*, "GPU Vect to Matrix (np.outer) fails"
   x = 0
endif
y = set_dim(77, 3, 1, 50, 0.001, 1.00, 1, 10)


if(gpucpu_test_vect_to_matrix() == 0) then
   print*, "GPU and CPU Vect to Matrix (np.outer) mismatch"
   x = 0
endif

if(x == 1)print*,"All Tests Pass"
end
```