

Name : Srushti Vikas Hembade
SID: 862395839
Email Id - shemb001@ucr.edu

Project Report

Option 1: Parallelizing serial C code with CUDA

Backpropagation Network

(Source: <https://courses.cs.washington.edu/courses/cse599/01wi/admin/Assignments/bpn.html>)

OVERVIEW :

In order to get beyond a single device's memory constraints and further shorten execution times, the utilization of many GPUs concurrently is becoming more and more common. The issue can be divided into separate calculations so that many GPUs can work on it concurrently.

The main function to be parallelized here as a part of the Back propagation network is the TrainNet, which is in charge of neural network training. Have parallelized the training of the sunspots data over 10 epochs to independently compute and store the adjusted weights calculated by every block. Ahead working with the TestNet function to calculate the TrainError and TestError followed by calculating the SaveWeights and Restoring the Weights to stop the training.

About the C code from the source given - The backpropagation neural network for time-series forecasting, namely for estimating the yearly number of sunspots, is implemented in this C code. The network is trained via the backpropagation technique and has momentum and bias terms. The program makes predictions about future values by using a dataset of sunspot values. The network's performance is assessed on both training and test sets, and the training procedure is repeated. When the test error goes down, the network's weights are preserved; if the test error goes up too much, training is terminated.

During training, the Normalized Mean Squared Error (NMSE) on the training and test sets is displayed in the simulator's output. Training is continued until the test error begins to rise; at that time, it is halted and the weights producing the greatest results are put back. Lastly, the network is assessed on an independent assessment set, and the sunspot values are contrasted with the predictions.

A series of comparatively tiny input data operations are available that we distribute to the kernel, allowing us to execute them separately on several GPUs. The work will then be completed faster by several GPUs.

The code for this project does multi-GPU training of the neural network with numerous kernel invocations using streams. The calculation is divided into four parts by the software.

How is the GPU used to accelerate the application?

Implementation details :

Details related to the parallel algorithm design / implementation:

Here, the main focus of parallelization is the TrainNet function, which is in charge of neural network training. Created a **data_declare.h header file** for all the declarations which is included in the main.cu and kernel.cu file where ever the declarations are needed.

Using CUDA memory allocation functions, the data—including **neural network Layer, Units, Output, Error, Weight, WeightSave, dWeight, and Input Sunspots array, and Main Output** — CPU to GPU using the **cudaMallocManaged()** i.e Unified memory which is accessible by both CPU and GPU as per their needs during the computations. The Network generated and initialized during the start of the training is also copied from CPU to GPU using the **cudaMalloc()** and **cudaMemcpy()** from host to device and back from device to host for the output generated in the kernel computations of all the thread blocks.

```
printf("Allocating device variables..."); fflush(stdout);
startTime(&timer);

cudaMalloc((void**)&d_Net, sizeof(NET));
cudaMalloc((void**)&d_Input, sizeof(REAL) * NUM_YEARS);
cudaMalloc((void**)&d_Output, sizeof(REAL) * M);

cudaDeviceSynchronize();
stopTime(&timer); printf("%f s\n", elapsedTime(timer));

printf("Copying data from host to device..."); fflush(stdout);
startTime(&timer);

cudaMemcpy(d_Net, &Net, sizeof(NET), cudaMemcpyHostToDevice);
cudaMemcpy(d_Input, &Input, sizeof(REAL) * NUM_YEARS, cudaMemcpyHostToDevice);

cudaDeviceSynchronize();
stopTime(&timer); printf("%f s\n", elapsedTime(timer));

size_t free_byte, total_byte;
free_byte, total_byte = cudaMemGetInfo(&free_byte, &total_byte);
printf("Free GPU memory: %zu bytes, Total GPU memory: %zu bytes\n", free_byte, total_byte);

printf("Launching kernel...");
startTime(&timer);
```

```

INT *Units;
cudaMallocManaged(&Units, NUM_LAYERS * sizeof(INT));

Net->Layer = (LAYER**) calloc(NUM_LAYERS, sizeof(LAYER*));
cudaMallocManaged(&Net->Layer, NUM_LAYERS * sizeof(LAYER*));

cudaError_t err = cudaMallocManaged(&(Net->Layer), NUM_LAYERS * sizeof(LAYER*));
if (err != cudaSuccess) {
    printf("Error: %s\n", cudaGetErrorString(err));
}

for (l=0; l<NUM_LAYERS; l++) {
    Net->Layer[l] = (LAYER*) malloc(sizeof(LAYER));
    cudaMallocManaged(&Net->Layer[l], sizeof(LAYER));
    Net->Layer[l]->Units = Units[l];
    cudaMallocManaged(&Net->Layer[l]->Units, Units[l] * sizeof(INT));

    Net->Layer[l]->Output = (REAL*) calloc(Units[l]+1, sizeof(REAL));
    cudaMallocManaged(&Net->Layer[l]->Output, Units[l]+1 * sizeof(REAL));

    Net->Layer[l]->Error = (REAL*) calloc(Units[l]+1, sizeof(REAL));
    cudaMallocManaged(&Net->Layer[l]->Error, Units[l]+1 * sizeof(REAL));

    Net->Layer[l]->Weight = (REAL**) calloc(Units[l]+1, sizeof(REAL*));
    cudaMallocManaged(&Net->Layer[l]->Weight, Units[l]+1 * sizeof(REAL*));

    Net->Layer[l]->WeightSave = (REAL**) calloc(Units[l]+1, sizeof(REAL*));
    cudaMallocManaged(&Net->Layer[l]->WeightSave, Units[l]+1 * sizeof(REAL*));

    Net->Layer[l]->dWeight = (REAL**) calloc(Units[l]+1, sizeof(REAL*));
    cudaMallocManaged(&Net->Layer[l]->dWeight, Units[l]+1 * sizeof(REAL*));

    Net->Layer[l]->Output[0] = BIAS;
    cudaMallocManaged(&Net->Layer[l]->Output[0], BIAS * sizeof(REAL));

    if (l != 0) {
        for (i=1; i<=Units[l]; i++) {
            Net->Layer[l]->Weight[i] = (REAL*) calloc(Units[l-1]+1, sizeof(REAL));
            cudaMallocManaged(&Net->Layer[l]->Weight[i], Units[l-1]+1 * sizeof(REAL));

            Net->Layer[l]->WeightSave[i] = (REAL*) calloc(Units[l-1]+1, sizeof(REAL));
            cudaMallocManaged(&Net->Layer[l]->WeightSave[i], Units[l-1]+1 * sizeof(REAL));
        }
    }
}

```

Created the **kernel wrapper function of the TrainNet()** that is to be called in the main.cu file from the kernel.cu and the kernel code in the **kernel function as TrainNet_krn1()**. CUDA kernels should then be used to **parallelize** the **TrainNet** function's training loop. The loop of **Epochs * TRAIN_YEARS** is assigned to GPU threads and iterates over training Sunspots Year Input. The CUDA kernel structure is designed to accommodate the dependencies between iterations, ensuring proper synchronization during execution.

```

printf("Launching kernel...");
startTime(&timer);

INT epochs = 10;

do {
    TrainNet(d_Net, epochs, d_Input);
    fprintf(f, " - stopping Training ...");
    Stop = TRUE;
} while (NOT Stop);

cuda_ret = cudaDeviceSynchronize();
if(cuda_ret != cudaSuccess) printf("Unable to launch kernel");
stopTime(&timer); printf("%f s\n", elapsedTime(timer));

```

```

void TrainNet(NET* Net, INT epochs, REAL* Input) {
    INT totalIterations = epochs * 150;

    dim3 dim_Grid, dim_Block;
    dim_Block.x = BLOCK_SIZE;
    dim_Block.y, dim_Block.z = 1;

    dim_Grid.x = (totalIterations-1)/BLOCK_SIZE+1;;
    dim_Grid.y = dim_Grid.z = 1;

    printf("dim_Block.x: %zu , dim_Grid.x: %zu \n", dim_Block.x, dim_Grid.x);

    printf("Executing TrainNet kernel...");

    for (INT i = 0; i < epochs; i++){
        TrainNet_krn1<<<dim_Grid, dim_Block>>>(Net, Input, 30, 179, 30, 1, 150);
    }

    cudaDeviceSynchronize();
    cudaError_t err = cudaGetLastError();
    if (err != cudaSuccess) {
        printf("Error launching kernel: %s\n", cudaGetErrorString(err));
        exit(1);
    }
}

```

```

INT epochs = 10;

do {
    TrainNet(d_Net, epochs, d_Input);
    printf("Copying data from device to host for TrainNet..."); fflush(stdout);
    startTime(&timer);
    // Copy the output data back to the CPU
    cudaMemcpy(&Net, d_Net, sizeof(NET), cudaMemcpyDeviceToHost);

    cudaDeviceSynchronize();
    stopTime(&timer); printf("%f s\n", elapsedTime(timer));

    TestNet(d_Net, d_Input, d_Train_error, d_Test_error);

    printf("Copying data from device to host for TestNet..."); fflush(stdout);
    startTime(&timer);
    // Copy the output data back to the CPU
    cudaMemcpy(&Train_error, d_Train_error, sizeof(REAL), cudaMemcpyDeviceToHost);
    cudaMemcpy(&Test_error, d_Test_error, sizeof(REAL), cudaMemcpyDeviceToHost);

    printf("Kernel code calculated Train Error: %f , Test error: %f \n", Train_error, Test_error);
    printf("\nMSE is %0.3f on Training Set and %0.3f on Test Set",
        Train_error / TrainErrorPredictingMean,
        Test_error / TestErrorPredictingMean);

    cudaDeviceSynchronize();
    stopTime(&timer); printf("Completion of kernel code for Training and Test....%f s\n", elapsedTime(timer));
}

```

Copied the calculated values of the GPU thread back from device to host for Network and the TrainError and TestError.

How is the problem space partitioned into threads / thread blocks:

The problem space is partitioned into threads and thread blocks to perform training iterations for a neural network.

One dimensional grid (dim_Grid.x) and one dimensional block (dim_Block.x) are launched with the kernel. In this scenario, **BLOCK_SIZE (256)** is defined as the block size i.e number of threads per block. Both the block size and the total number of iterations (totalIterations) determine the total number of blocks in the grid.

```

void TrainNet(NET* Net, INT epochs, REAL* Input) {
    INT totalIterations = epochs * 150;

    dim3 dim_Grid, dim_Block;
    dim_Block.x = BLOCK_SIZE;
    dim_Block.y, dim_Block.z = 1;

    dim_Grid.x = (totalIterations-1)/BLOCK_SIZE+1;;
    dim_Grid.y = dim_Grid.z = 1;
}

```

Based on its location inside the block (threadIdx.x) and the block's position within the grid (blockIdx.x), each thread in the kernel is given a unique index (idx). The product of block sizes and grid size yields the total number of threads.

```
__global__ void TrainNet_krnl(NET* Net, REAL* Input, INT lwb, INT upb, INT n, INT m, INT tr_yeras)
{
    INT idx = threadIdx.x + blockIdx.x * blockDim.x;
    INT Year;
    REAL Output[M];

    if (idx >= tr_yeras) {
        return;
    }
}
```

Which stage of the software pipeline is parallelized :

The training epochs are represented by the outer loop (for (INT i = 0; i < epochs; i++)). With the given grid and block size, the TrainNet_krnl kernel is launched on each iteration of the loop. A portion of the issue space is processed by the kernel according to the thread and block indices.

```
for (INT i = 0; i < epochs; i++){
    TrainNet_krnl<<<dim_Grid, dim_Block>>>(Net, Input, 30, 179, 30, 1, 150);
}

cudaDeviceSynchronize();
cudaError_t err = cudaGetLastError();
if (err != cudaSuccess) {
    printf("Error launching kernel: %s\n", cudaGetErrorString(err));
    exit(1);
}
```

We are aware that a grid is composed of threads blocks and that blocks are composed of threads. The block index is the same for every thread in the same block. We employ the following "if" condition to make sure the extra threads don't go out of bounds and to ensure that the extra threads do not do any computations..

```
if (idx < tr_yeras) {
    Year = RandomEqualINT_krnl(lwb, upb);
    if (Year >= 30 && Year < NUM_YEARS) {
        printf("Entered if year ....");
        SimulateNet(Net, &Input[Year - 30], Output, &Input[Year], TRUE);
    }
}
```

Within the total number of iterations (totalIterations), each thread is in charge of processing a particular iteration (idx).

The TrainNet_krnl kernel chooses random years to simulate the neural network (Year = RandomEqualINT_krnl(TRAIN_LWB, TRAIN_UPB)) depending on the thread index.

Ahead will execute the SimulateNet kernel method to independently compute the below steps for each thread executuing with the year generated as a index to the sunspots input

array with Input (year -30) and target value to be considered to train on as the Input(year) along with the Network copied from the host.

```
__device__ void SimulateNet (NET* Net, REAL* Input, REAL* Output, REAL* Target, BOOL Training)
{
    // printf("Entered SimulateNet method ....");
    SetInput(Net, Input);
    PropagateNet(Net);
    GetOutput(Net, Output);

    ComputeOutputError(Net, Target);
    if (Training) {
        BackpropagateNet(Net);
        AdjustWeights(Net);
    }
}
```

Ahead of this each thread will perform the function SimulateNet, which is identified as a device function in GPU code, is in charge of simulating the neural network on each thread for a specific set of input data. First, the code uses the SetInput function to set the input values. Then, using the PropagateNet function, the neural network is transmitted forward through its layers. Next, we use the GetOutput method to retrieve the output data from the network. The ComputeOutputError function is then used by the function to calculate the error between the goal output and the obtained output. The function backpropagates the error via the network using the BackpropagateNet function and modifies the weights using the AdjustWeights function if the simulation is a part of a training process (Training flag is true). This entire simulation and, if applicable, training process are executed independently on each thread, contributing to the parallelized computation of the neural network simulation and training tasks on the GPU.

OPTIMIZATION of TrainError and TestError calculations IMPLEMENTATION :

Have parallelized the TrainError and TestError calculations in the TestNet function called by managing to calculate the values for the Train years bound and Test years bounds parallelly on the same thread as they are not interdependent.

```

__global__ void TestNet_krn1(NET* Net, REAL* Input, INT no_yeras, INT Year, REAL* Train_error, REAL* Test_error)
{
    INT idx = threadIdx.x + blockIdx.x * blockDim.x;
    REAL Output[M];

    REAL TrainError = 0;
    REAL TestError = 0;
    // printf("Enetered TestNet method....");
    Net->Error = 0;
    if (idx >= no_yeras) {
        return;
    }

    if (idx < no_yeras) {
        // printf("Enetered if loop....");
        SimulateNet(Net, &Input[Year-30], Output, &Input[Year], FALSE);
        TrainError += Net->Error;
        *Train_error = TrainError;
        // printf("TrainError :   %f , Train_error_Host:  %f\n", TrainError, *Train_error);
        __syncthreads();
        if (Year + 150 <= 259 ){
            SimulateNet(Net, &Input[Year-30 + 150], Output, &Input[Year + 150], FALSE);
            TestError += Net->Error;
            *Test_error = TestError;
            // printf("TestError :   %f , Test_error_Host:  %f\n", TestError, *Test_error);
        }
        __syncthreads();
    }
}

```

```

TestError :   0.009774 , Test_error_Host:  0.009774
TestError :   0.009774 , Test_error_Host:  0.009774
TestError :   0.009774 , Test_error_Host:  0.009774
TestError :   0.009774 , Test_error_Host:  0.009774
TestError :   0.009774 , Test_error_Host:  0.009774
TestError :   0.009774 , Test_error_Host:  0.009774
TestError :   0.009774 , Test_error_Host:  0.009774
TestError :   0.009774 , Test_error_Host:  0.009774
TestError :   0.009774 , Test_error_Host:  0.009774
TestError :   0.009774 , Test_error_Host:  0.009774
TestError :   0.009774 , Test_error_Host:  0.009774
TestError :   0.009774 , Test_error_Host:  0.009774
TestError :   0.009774 , Test_error_Host:  0.009774
TestError :   0.009774 , Test_error_Host:  0.009774
TestError :   0.009774 , Test_error_Host:  0.009774
TestError :   0.009774 , Test_error_Host:  0.009774
TestError :   0.009774 , Test_error_Host:  0.009774
TestError :   0.009774 , Test_error_Host:  0.009774
TestError :   0.009774 , Test_error_Host:  0.009774
TrainError :   0.000973 , Train_error_Host:  0.000973
TrainError :   0.000973 , Train_error_Host:  0.000973
TrainError :   0.000973 , Train_error_Host:  0.000973
TrainError :   0.000973 , Train_error_Host:  0.000973
TrainError :   0.000973 , Train_error_Host:  0.000973
TrainError :   0.000973 , Train_error_Host:  0.000973
TrainError :   0.000973 , Train_error_Host:  0.000973
TrainError :   0.000973 , Train_error_Host:  0.000973
TrainError :   0.000973 , Train_error_Host:  0.000973
TrainError :   0.000973 , Train_error_Host:  0.000973
TrainError :   0.000973 , Train_error_Host:  0.000973
TrainError :   0.000973 , Train_error_Host:  0.000973
TrainError :   0.000973 , Train_error_Host:  0.000973
TrainError :   0.000973 , Train_error_Host:  0.000973
TrainError :   0.000973 , Train_error_Host:  0.000973
TrainError :   0.000973 , Train_error_Host:  0.000973
TrainError :   0.000973 , Train_error_Host:  0.000973
TrainError :   0.000973 , Train_error_Host:  0.000973
TrainError :   0.000973 , Train_error_Host:  0.000973

```

What libraries were used?

The standard C libraries `<stdlib.h>`, and `<stdio.h>` are utilized, which offer a collection of functions and data types for handling integers, memory allocation, and input/output procedures.

In C, `<sys/time.h>` header is specifically used to work with time-related functions. Mostly for measuring elapsed time, setting timeouts, or handling various time-related operations. The mathematical functions like the exponential function `exp()`, `sum()`, which is employed in the sigmoid activation function of the neural network, are probably obtained from the `<math.h>` library. Specifically for below code lines used in the struct of Network and Layer computations and updates.

```
__device__ void PropagateLayer(NET* Net, LAYER* Lower, LAYER* Upper)
{
    INT i,j;
    REAL Sum;

    for (i=1; i<=Upper->Units; i++) {
        Sum = 0;
        for (j=0; j<=Lower->Units; j++) {
            Sum += Upper->Weight[i][j] * Lower->Output[j];
        }
        Upper->Output[i] = 1 / (1 + exp(-Net->Gain * Sum));
    }
}
```

FOLDER STRUCTURE :

Referred to professors code as we have for our lab assignments previously. Integrated my code in the same format. Created a `data_declare.h` header file with all the declarations of variables. Then have the main function in the `main.cu` file along with the serialised methods that run on the CPU itself at the beginning that execute only once before training initiation. Have the Kernel code in `kernel.cu` file with the other device specific methods as well.

Created a Makefile to compile the `main.cu` and `support.cu` (evaluation of the network values) file and generate a back propagation execution file.

Documentation on how to run your code

Please follow the instructions below to generate and perform the Back propagation on Network for implemented parallelism and acceleration using GPUs:

1. Install Bender with supporting 4 GPUs. Login into your engr account on the bender using the below command example for your username and password.

```
ssh -l shembade bender.engr.ucr.edu (replace the username with your username)/
```

ssh shembade@bender.engr.ucr.edu (replace the username with your username)

Password: ***** (your password)

2. Use the command **git clone** to create a copy of the source code repository on Bender. The code will be downloaded on your bender account folder named as finalproject-f23-srushti-hembade.

3. Open the source code directory(<**cd folder_name**> command) and use the **make** command to begin the code's compilation. In order to run the software, these compilation files will be created named as **back-propagation**.

4. Use the command **./back-propagation** to start the program. By default, this will execute the program using a 280 years Sunspots data and 10 epochs loop and validate the outcome.

Evaluation/Results :

The Code ran with GPU code is faster when Training and testing phase for saving the weights with minimum test error as compared to the serialized C code as given. Below is the screenshot depicting the time taken for allocating memory, copying from host to device, kernel computations and the memory copy from device to host.

```

Setting up the problem...
Available Number of GPU: 4
0.017505 s
    Net: 56 x 2240
    Output: 8 x 1
    Input: 2979787469 x 1702539733
Input array:
0.026200 0.057500 0.083700 0.120300 0.188300 0.303300 0.151700 0.104600 0.052300 0.041800 0.015700 0.000000 0.000000 0.010500 0.057500 0.141200 0.
245800 0.329500 0.313800 0.204000 0.146400 0.136000 0.115100 0.057500 0.109800 0.209200 0.407900 0.638100 0.538700 0.381800 0.245800 0.183100 0.05
7500 0.026200 0.083700 0.177800 0.366100 0.423600 0.580500 0.528200 0.381800 0.209200 0.104600 0.083700 0.026200 0.057500 0.115100 0.209200 0.3138
00 0.423100 0.436200 0.249500 0.250000 0.160600 0.063800 0.050200 0.053400 0.170000 0.248900 0.282400 0.329000 0.449300 0.320100 0.235900 0.190400
0.109300 0.059600 0.197700 0.365100 0.554900 0.527200 0.426800 0.347800 0.182000 0.160000 0.036600 0.103600 0.483800 0.807500 0.658500 0.443500 0.
356200 0.201400 0.119200 0.053400 0.126000 0.433600 0.690400 0.684600 0.617700 0.470200 0.348300 0.313800 0.245300 0.214400 0.111400 0.083700 0.063
3500 0.021400 0.035600 0.075800 0.177800 0.235400 0.225400 0.248400 0.220700 0.147000 0.052800 0.042400 0.013100 0.000000 0.007300 0.026200 0.063
800 0.072700 0.185100 0.239500 0.215000 0.157400 0.125000 0.081600 0.034500 0.020900 0.009400 0.044500 0.086800 0.189800 0.259400 0.335800 0.35040
0 0.370800 0.250000 0.143800 0.044500 0.069000 0.297600 0.635400 0.723300 0.539700 0.448200 0.337900 0.191900 0.126600 0.056000 0.078500 0.209700
0.321600 0.515200 0.652200 0.503600 0.348300 0.337300 0.282900 0.204000 0.107700 0.035000 0.022500 0.118700 0.286600 0.490600 0.501000 0.403800 0.
309100 0.230100 0.245800 0.159500 0.085300 0.038200 0.196600 0.387000 0.727000 0.581600 0.531400 0.346200 0.233800 0.088900 0.059100 0.064900 0.01
7800 0.031400 0.168900 0.284000 0.312200 0.333200 0.332100 0.273000 0.132800 0.068500 0.035600 0.033000 0.037100 0.186200 0.381800 0.445100 0.4079
00 0.334700 0.218600 0.137000 0.139600 0.063300 0.049700 0.014100 0.026200 0.127600 0.219700 0.332100 0.281400 0.324300 0.253700 0.229600 0.097300
0.029800 0.018800 0.007300 0.050200 0.247900 0.298600 0.543400 0.421500 0.332600 0.196600 0.136500 0.074300 0.030300 0.087300 0.231700 0.334200 0.
360900 0.406900 0.339400 0.186700 0.110900 0.058100 0.029800 0.045500 0.188800 0.416800 0.598300 0.573200 0.464400 0.354600 0.248400 0.160000 0.0
85300 0.050200 0.173600 0.484300 0.792900 0.712800 0.704500 0.438800 0.363000 0.164700 0.072700 0.023000 0.198700 0.741100 0.994700 0.966500 0.831
600 0.587300 0.281900 0.196100 0.145900 0.053400 0.079000 0.245800 0.490600 0.553900 0.551800 0.546500 0.348300 0.360300 0.198700 0.180400 0.08110
0 0.065900 0.142800 0.483800 0.812700
Starting the Back propagation code...Completed InitializeRandoms...Completed GenerateNetwork...Completed RandomWeights...Completed InitializeAppli
cation...Allocating device variables...0.000347 s
Copying data from host to device...0.000093 s
Free GPU memory: 7026900992 bytes, Total GPU memory: 0 bytes
Launching kernel...dim_Block.x: 256 , dim_Grid.x: 6
Executing TrainNet kernel...Copying data from device to host for TrainNet...0.000024 s
dim_Block.x: 256 , dim_Grid.x: 2
Executing TestNet kernel...Copying data from device to host for TestNet...Kernel code calculated Train Error: 0.075714 , Test error: 0.553884

NMSE is 0.044 on Training Set and 0.411 on Test SetCompletion of kernel code for Training and Test....0.000056 s
- saving Weights ... - stopping Training ...dim_Block.x: 256 , dim_Grid.x: 2
Executing TestNet kernel...Copying data from device to host for Final TestNet...Kernel code calculated Train Error: 0.075714 , Test error: 0.5538
84

NMSE is 0.044 on Training Set and 0.411 on Test SetCompletion of kernel code for final TestNet....0.000053 s
Copying data from device to host...Completion of whole kernel code....0.000010 s
bender /home/cegrad/shembade/finalproject-f23-srushti-hembade $ ls
back-propagation BPN.txt data declare.h kernel.cu main.cu Makefile README.md support.cu support.h support.o
bender /home/cegrad/shembade/finalproject-f23-srushti-hembade $ cat BPN.txt

```

The size of the input and output set along with the input array and the Network generated after the Application Initiation method.

```

bender /home/cegrad/shembade/finalproject-f23-srushti-hembade $ ./back-propagation

Setting up the problem...0.000000 s
    Net: 56 x 2240
    Output: 8 x 1281310176
    Input: 261051085 x 1702463359
Input array:
0.026200 0.057500 0.083700 0.120300 0.188300 0.303300 0.151700 0.104600 0.052300 0.041800 0.015700 0.000000 0.000000 0.010500 0.057500 0.141200 0.
245800 0.329500 0.313800 0.204000 0.146400 0.136000 0.115100 0.057500 0.109800 0.209200 0.407900 0.638100 0.538700 0.381800 0.245800 0.183100 0.05
7500 0.026200 0.083700 0.177800 0.366100 0.423600 0.580500 0.528200 0.381800 0.209200 0.104600 0.083700 0.026200 0.057500 0.115100 0.209200 0.3138
00 0.423100 0.436200 0.249500 0.250000 0.160600 0.063800 0.050200 0.053400 0.170000 0.248900 0.282400 0.329000 0.449300 0.320100 0.235900 0.190400
0.109300 0.059600 0.197700 0.365100 0.554900 0.527200 0.426800 0.347800 0.182000 0.160000 0.036600 0.103600 0.483800 0.807500 0.658500 0.443500 0.
356200 0.201400 0.119200 0.053400 0.126000 0.433600 0.690400 0.684600 0.617700 0.470200 0.348300 0.313800 0.245300 0.214400 0.111400 0.083700 0.063
3500 0.021400 0.035600 0.075800 0.177800 0.235400 0.225400 0.248400 0.220700 0.147000 0.052800 0.042400 0.013100 0.000000 0.007300 0.026200 0.063
800 0.072700 0.185100 0.239500 0.215000 0.157400 0.125000 0.081600 0.034500 0.020900 0.009400 0.044500 0.086800 0.189800 0.259400 0.335800 0.35040
0 0.370800 0.250000 0.143800 0.044500 0.069000 0.297600 0.635400 0.723300 0.539700 0.448200 0.337900 0.191900 0.126600 0.056000 0.078500 0.209700
0.321600 0.515200 0.652200 0.503600 0.348300 0.337300 0.282900 0.204000 0.107700 0.035000 0.022500 0.118700 0.286600 0.490600 0.501000 0.403800 0.
309100 0.230100 0.245800 0.159500 0.085300 0.038200 0.196600 0.387000 0.727000 0.581600 0.531400 0.346200 0.233800 0.088900 0.059100 0.064900 0.01
7800 0.031400 0.168900 0.284000 0.312200 0.333200 0.332100 0.273000 0.132800 0.068500 0.035600 0.033000 0.037100 0.186200 0.381800 0.445100 0.4079
00 0.334700 0.218600 0.137000 0.139600 0.063300 0.049700 0.014100 0.026200 0.127600 0.219700 0.332100 0.281400 0.324300 0.253700 0.229600 0.097300
0.029800 0.018800 0.007300 0.050200 0.247900 0.298600 0.543400 0.421500 0.332600 0.196600 0.136500 0.074300 0.030300 0.087300 0.231700 0.334200 0.
360900 0.406900 0.339400 0.186700 0.110900 0.058100 0.029800 0.045500 0.188800 0.416800 0.598300 0.573200 0.464400 0.354600 0.248400 0.160000 0.0
85300 0.050200 0.173600 0.484300 0.792900 0.712800 0.704500 0.438800 0.363000 0.164700 0.072700 0.023000 0.198700 0.741100 0.994700 0.966500 0.831
600 0.587300 0.281900 0.196100 0.145900 0.053400 0.079000 0.245800 0.490600 0.553900 0.551800 0.546500 0.348300 0.360300 0.198700 0.180400 0.08110
0 0.065900 0.142800 0.483800 0.812700
Starting the Back propagation code...Completed InitializeRandoms...Completed GenerateNetwork...Completed RandomWeights...Completed InitializeAppli
cation...Network Information:
Number of Layers: 3
Alpha: 0.500000
Eta: 0.050000
Gain: 1.000000

Layer 0:
Units: 30
Unit 0:
    Output: 1.000000
    Error: 0.000000
Unit 1:
    Output: 0.000000
    Error: 0.000000

```


BPN.txt file → Complete prediction output using parallel TrainNet and TestNet GPU code.

```
bender /home/cegrad/shembade/finalproject-f23-srushti-hembade $ cat BPN.txt
```

Year	Sunspots	Open-Loop Prediction	Closed-Loop Prediction
1960	0.572	0.296	0.296
1961	0.327	0.297	0.296
1962	0.258	0.302	0.303
1963	0.217	0.309	0.312
1964	0.143	0.305	0.307
1965	0.164	0.306	0.303
1966	0.298	0.307	0.304
1967	0.495	0.321	0.321
1968	0.545	0.327	0.324
1969	0.544	0.316	0.316
1970	0.540	0.306	0.304
1971	0.380	0.293	0.295
1972	0.390	0.303	0.307
1973	0.260	0.309	0.310
1974	0.245	0.302	0.305
1975	0.165	0.297	0.306
1976	0.153	0.300	0.292
1977	0.215	0.308	0.302
1978	0.489	0.314	0.307
1979	0.754	0.308	0.295

```
bender /home/cegrad/shembade/finalproject-f23-srushti-hembade $ █
```

Status of your project :

In my project, parallelized the C source code given for the back propagation simulator as Time-Series Forecasting Prediction of the Annual Number of Sunspots by providing the loop of **training** phase for epoch * train years to multiple threads by dividing the training computations among each and every thread within a thread block of size 256 that will run parallel and return the train error data needed ahead to predict the values. Followed by the **TestNet()** function in which both of the methods include the same **SimulateNet** method including its sub methods as the device methods.

I have written code for separating the loop among the thread blocks in the grid and then execute the SimulateNet method on each and every step for 10 epochs. It helped speeding up the training process than executing it in the loop on CPU in serialization due to the parallel thread executing it resulting in increased performance.

Also, have added code for the four different kernels in which I am able to get the results I wanted for the predictions of the years more faster than single GPU.. The Status of the project is that I am able to see that the algorithm is able to do the training using 4 different GPU. The Feature for scaling to the multiple GPU works well.

CHALLENGES OF THE PROJECT :

I am getting the desired output for the predictions on the years of sunspots after using the GPUs. Limitations on the project would be the size epochs to be ran on the GPUs for training as it is dependant on the threads utilized, won't support more than 10 epoch loops of training. In each version, the memory could only handle a small amount of data due to 2 byte memory of each GPU—resulted in Segmentation faults so used the Multi-GPUs. I used the existing documentation and implementation of the multi-GPU back propogation when constructing the project as per professor provided slide steps. Additionally, initially during the implementation was unable to allocate memory for each and every layer in GPU facing out of memory error, so used Unified memory for not taking more space in GPU and reducing the transfer of CPU to GPU that helped in increasing the performance. Tried using the multi- gpu to improves its performance, but faced some issue with the illegal memory allocation so ended up doing everything on one GPU.

CONCLUSION :

The procedure is executed faster when many GPUs are used for back propagation in a network, hence it makes sense to employ numerous GPUs instead of only one. This Cuda Streams feature allows us to run numerous independent jobs on different GPUs at the same time. This enables us to implement algorithms more quickly.

CONTRIBUTIONS :

Task	Contribution
Implementation of CUDA GPU code, Report, Presentation Video	Srushti Hembade - 100%

Presentation VIDEO -Shared on Yuja/ Media with Professor. Daniel Wong and TA's
<https://drive.google.com/file/d/1RQvVFdzg-8ccHgRcOVXemKG3MFsJiXgD/view?usp=sharing>

23

uncements

Media/Storage


ments


ius

4

scope

xtbooks

 **Media Library**

 **Manage Media**

Search

NEW FOLDER

NEW PLAYLIST

UPLOAD

GO BACK

MORE ACTIONS

PERSONAL

My Media

Default Collecti...

Shared With Me

Favorites

Shared With Others

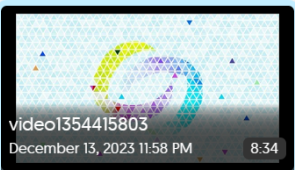
SHARED

EnterpriseTube

Shared Folders

All Channels

Default Collection



video1354415803
December 13, 2023 11:58 PM 8:34