Parallel Implementation of Simple Neural Networks for Image Recognition on Fashion-MNIST dataset using CUDA

Sesha Phani Deepika Vadlamudi[1]

Stevens Institute of Technology
svadlam1@stevens.edu

Abstract. These days neural network applications are found everywhere because of their ability to mimic the human brain operations to recognize patterns and relationships between humongous amounts of data. Some of the applications need real-time predictions by analyzing thousands to millions of data points. One such example is that of a self-driving car. Therefore, there is a dire need to parallelize the existing algorithms for better and faster performance. In this project, we implement a simple neural network architecture to demonstrate the efficacy of parallel implementations over sequential implementations. We use the Fashion-MNIST dataset to evaluate the performance of our implementations. The parallel implementation has achieved a 35x speedup over the sequential version.

*Appendix A contains the code snippets of kernel functions that use shared memory and Appendix B contains the code snippets of kernel functions that compute convolution and max pool operations together.

1 Introduction

Neural networks are a major area of research in the field of Machine Learning and Artificial Intelligence. Their agile nature to learn almost every kind of concept makes them the most preferred way to address problems. This is because they have thousands and most often millions of parameters that they fine-tune during their training phase. Since there are so many parameters and calculations involved and some applications have to perform real-time analysis, this is a perfect setting for parallelizing neural nets. Tesla, for example, takes in input from 8 different cameras located around the car, has to process these continuous inputs, and has to calculate the steering angle accurately in real-time.

We implement a parallel version of a neural network that has an architecture similar to that of AlexNet. Firstly, we train the neural network using TensorFlow on the Fashion-MNIST dataset and obtain weights for each layer. We then use these weights to test a given image. We compare the time taken by the sequential version to that of the parallel version. The goal remains to make use of different concepts and techniques to achieve maximum speedup.

2 Dataset

For our project, we take the Fashion-MNIST dataset to evaluate the performance. This dataset consists of 28x28 pixels, grayscale images. It has 70,000 images of fashion products from 10 categories with a label indicating the correct garment. Also, there are 7,000

images per category. The training set has 60,000 images and the test set has 10,000 images.

Fashion-MNIST dataset is an alternative to the MNIST dataset and was created to

Fig. 1. Sample Images from the Fashion MNIST dataset, source - ResearchGate

serve as replacement for the original MNIST dataset to benchmark algorithms in the Machine Learning domain. The reason for selecting Fashion-MNIST over the original MNIST is because the original dataset is overused, and is often regarded as too easy. However, the size of the images and the size of the training and test sets in both the datasets are same.

3 Neural Network Architecture

Neural Networks are used for a wide range of problems such as text processing, image recognition, stock price prediction, machine translation, bioinformatics, drug design etc. Different kind of neural networks have been created over the years for different kind of purposes. Recurrent Neural Networks for example are used to process sequential data. They are used for forecast models. Convolutional Neural Networks, a different kind of neural nets, are used for Computer Vision tasks and these are the kind of neural nets that we will be using in our project.

Our neural network is an adaptation of the AlexNet. It comprises a convolution base and a classifier. The convolution base has two sets of convolution layers each followed by a max-pool layer. The first convolution layer has a filter of 32 kernels, each having a size of $(5, 5)$. The second convolution layer has a filter of 64 kernels, each of size $(5, 5, 32)$.

The reason for choosing this specific filter size is to avoid zero-padding which consumes

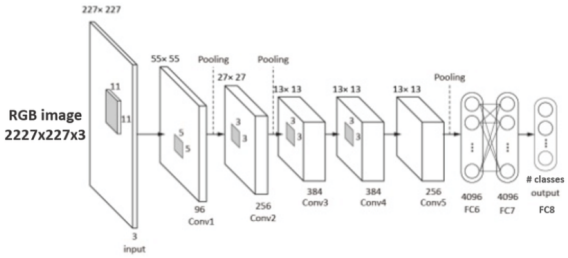

Fig. 2. Architecture of AlexNet, source: Alexander Khvostikov, ResearchGate

additional computational power. Let us consider an example where the input is of dimensions (x, y). When we apply a convolutional operation using a filter whose dimension is (a, b), the output of the conv layer is of dimensions $((x - a + 1), (y - b + 1))$. Therefore, if we happened to choose a filter with dimension $(3, 3)$, the output of the first convolutional layer would be $(26, 26)$. And the input to our second convolutional layer would be $(13, 13)$. The output of the second conv layer would be $(11, 11)$. This output is supposed to be passed to the max pool layer which operates on a $(2, 2)$ patch. And hence, this would require us to zero pad our input or look for other ways.

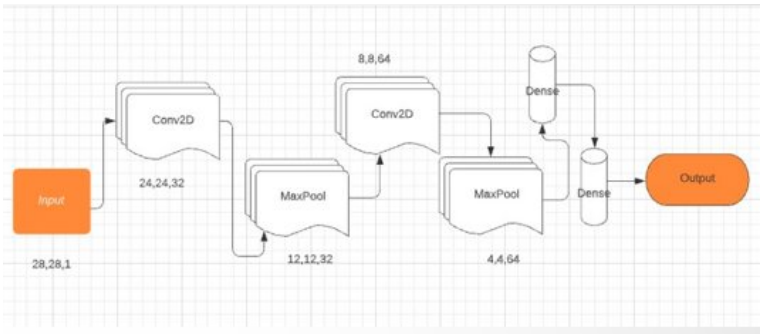

Fig. 3. Architecture of the Neural Network used for this project

All the convolution operations have stride 1 and are 2D operations, which means that the filter moves in only two directions. The two max pool layers after the convolution layers operate on a patch of $(2, 2)$ and have a stride of 2. For the classifier part of our network, we have two dense layers. The first dense layer has 64 units, and the second dense layer has 10 units. Since we store all our inputs and outputs as a 1D array, we don't need to implement a flatten layer which is usually implemented in between the convolution base and classifier.

4 Implementation

I have implemented this project using google colab. Colab has assigned me a Tesla K80 GPU with compute capability of 3.7. All the results obtained are from the same session.

4.1 First Stage

The first stage of this project was to train a neural network and obtain the weights. I have trained the neural network and ran it for 200 epochs and got an accuracy of about 91.1%. This is a reasonable accuracy given that our architecture has only two convolutional layers. After training, we save the weights of our model to a file so that we can use them while performing testing on our sequential and parallel program.

4.2 Second Stage

The second stage of this project was to implement a sequential version of the neural network. The code is sequential, we only have one function for the max pool layer where it takes in the respective four elements and outputs the maximum among those elements. The sequential algorithm was executed in 0.01 seconds.

Since the size of the kernel is $(5, 5)$ for the kernel in the filter of the first convolutional layer, instead of having a for loop going over the elements, I rolled out the loop and programmed it to be one big equation. The second convolutional layer has a kernel of size $(5, 5, 32)$. So for this code block, I programmed it such that there is a for loop going over from 0 to 31, and one big equation that computes the sum of 25 products of the kernel and the respective patch of the input.

The dense layer calculates the inner product between the input and a weight matrix. For the code block for dense layers we had three nested for loops. The number of times the outer loop runs is the same as the number of units of our dense layer, i.e, for the first dense layer, there are 64 units. The output of the first dense layer is (64,1) which is the input to the second dense layer which has 10 units. The weight matrix of the first dense layer has the dimensions of (1024, 64) since the input to this layer is of dimensions (1024,1) and the dimensions of the weight matrix in the second dense layer are (64,10) since the input to this layer is the output of the previous layer which has dimensions of (64,1).

4.3 Third Stage

In this stage, we implemented the first GPU version of the neural network architecture. This version mostly uses global memory. Only the filter for the first convolutional layer is stored in constant memory. Each layer has a kernel of its own. Therefore, there are six kernels in total. Although the max pool kernel could be reused, there are two separate kernels for the two different layers offering a tailored fit to the problem. Also, this will not harm any performance since both the kernels are never executed at the same time, so there will not be any kernel overhead.

For the first convolutional layer, the grid size is (32) and the block size is (32,32). Since there are 32 kernels in the filter, each block computes the output of one kernel. Therefore, each block has a total number of 576 (24×24) computations. The number of global

writes is 576 as well for each block. Note that the filter of this layer is stored in constant memory

The second kernel is that of the first max pool layer. The grid size is (32) and the block size is (16,16). Implementing the same technique as before, each block calculates one layer of the input of dimensions (24,24,1). Hence, each block gives an output of dimension (12,12,1) which are then stacked together to form the input to the next layer, the second convolutional layer, of dimensions (12,12,32).

The third kernel computes the second convolutional layer. The grid dimensions are (64,1) and the block dimensions are (8,8). Each block calculates the output of one kernel in the filter which of dimensions (5,5,32). This kernel is slightly different from that of the first. The first kernel has a filter of dimensions (5,5,32) whereas the second conv layer has a filter of dimensions (5,5,32,64). So, this kernel has an additional loop that goes over the third dimension (0 to 31). Each thread calculates one element of this layer. Therefore, each thread computes a total of 800 calculations (the threads are quite heavy). The output of this layer is of dimensions (8,8,64).

The fourth kernel which computes the second max pool layer is the same as the second kernel. The input to this kernel is (8,8,64) and the output is (4,4,64). The grid dimensions and block dimensions are the same as the previous layer.

The fifth kernel is that of the dense layer. The grid size is (1,1) and the block dimensions are (64,1). Each thread calculates one element of the output which is the inner product of the output of the previous layer with one column of the weight matrix. The weight matrix is of dimensions (1024,64). The input to this layer is of dimensions (1024,1). The threads are computationally heavy in this kernel as well.

The sixth kernel is the same as the previous kernel. It takes in the input of dimensions (64,1) and the weight matrix is of dimensions (64,10). The output is of dimensions (10,1).

4.4 Fourth Stage

In this final stage of the project, we convert one kernel at a time to use shared memory and record the time taken by the program at each stage the findings of which we shall discuss in the Analysis section. The grid dimensions and block dimensions are the same as before unless specified otherwise.

As before, the first kernel computes the first convolutional layer. The image is in constant memory, the filter is of dimensions (5,5,32). Each block loads one layer of the filter (5,5,1) into shared memory and each block computes one layer of the output which is of dimensions (24,24,1). The final output of this layer is (24,24,32).

The second kernel computes the max pool layer. Each block loads one layer of the in-

put (24,24,1) into the shared memory and calculates one layer of the output which is of dimension (12,12,1). The output of this layer is (12,12,32).

The third kernel computes the second convolutional layer. Each block loads one kernel of the filter which is of dimension (5,5,32) and the input of this layer. Each block outputs one layer of the output which is of dimension (8,8,1). The output of this layer is (8,8,64).

The fourth kernel calculates the second max pool layer. Same as before, each block loads one layer of the input (8,8,1) and calculates one layer of the output (4,4,1). Therefore, the output of this layer comes down to (4,4,64) since there are 64 kernels in the filter.

The fifth kernel computes the operation that of a dense layer. The grid dimensions are (64,1) and the block dimensions are (32,32). Firstly, each thread loads one element of the input and one element of the corresponding column of the weight matrix into shared memory. After loading the required elements, we create another array with the dimensions that of the input and compute the product of the input element and the corresponding weight element and store it in this temporary output array. After all the elements of this array are computed, we go over all the elements of this array and sum them. This sum is then written to the global output array.

The sixth kernel is that of the second dense layer. The grid dimensions for this kernel are (10,1) and the block dimensions are (64,1). The operation of this kernel is the same as the previous kernel.

5 Occupancy calculations

Following are the occupancy calculations for the kernels using shared memory:

- Conv1 Kernel - Has 1024 threads allocated, 100% occupancy
- Maxpool1 Kernel - Has 256 htreads allocated, 100% occupancy
- Conv2 Kernel - Number of threads allocated for this kernel are 64, 50% occupancy
- Maxpool2 Kernel - Number of threads allocated are the same as previous kernel, 50% occupancy
- Dense1 Kernel - Even this kernel has 64 threads allocated to it, 100% occupancy
- Dense2 Kernel - This kenel has only 64 threads allocated, 50% occupancy

We shall analyze these results in the next section.

6 Experiments, Results, and Analysis

The initial stage of experimentation involved looking at the speedup gained as each of the kernels switched from global memory to shared memory. The results are shown in the table below:

S no.	Layers in the net						Time taken
	Conv1	Maxpool1	conv2	Maxpool2	Dense1	Dense2	
1	NA	NA	NA	NA	NA	NA	0.013744 sec
2	GM	GM	GM	GM	GM	GM	0.000706 sec
3	SM	GM	GM	GM	GM	GM	0.000723 sec
4	SM	SM	GM	GM	GM	GM	0.000724 sec
5	SM	SM	GM	SM	GM	GM	0.000724 sec
6	SM	SM	SM	SM	GM	GM	0.000708 sec
7	SM	SM	SM	SM	SM	GM	0.000629 sec
8	SM	SM	SM	SM	SM	SM	0.000613 sec

Different rows correspond to different experiments performed in the project. Each column corresponds to the respective layer and indicates if the kernel corresponding to the layer used Global Memory (GM) or Shared Memory (SM).

It is observed that the GPU version that uses just the global memory performs 14 times faster than the CPU version and the GPU version where all the kernels use shared memory performs 18 times faster than the sequential CPU code. Also, it can be noted that the relative speed up gained by the program where all the kernels used Shared memory is not much when compared to the program which used only global memory. This maybe a possible cause of kernel overhead. this can also be attributed to the occupancy calculations that were mentioned in the previous section. It can be duly noted that the GPU was not 100% used while some kernels were running. Changes can be made to address this possibility of better and faster performance.

Acknowledging this, we performed further experiments. The first one was trying to load in the filters that are required for the convolutional layers into constant memory. Although, loading both the filters was not possible because of the memory restrictions of constant memory. We got the following error :

```
ptxas error   : File uses too much global constant data (0x338c0 bytes, 0x10000 max)
```

Fig. 4. Error for exceeding limits of constant memory

Therefore, we now have only the initial image and the filter corresponding to the first convolutional layer in constant memory. Running this program as is, the program took 0.000593 seconds.

Next, since we attributed the minimal speedup to kernel overhead, we combine the operation of max pool layers with that of the convolutional layers. I recorded the time taken by the program when only the first max pool kernel was combined with that of the first con-

volutional layer and all the other kernels (conv2, max pool2, dense1, and dense2) remain as is and this program took about 0.000465 seconds. Next, I combined the second max pool layer with the kernel computing the second convolutional layer. Now the program has four kernels - the first kernel computing the first convolutional and maxpool layer operations, the second kernel computes the second convolutional and maxpool operations, the third computes the first dense layer, and the fourth kernel computes the second dense layer. This version of program took 0.000392 seconds to complete.

Now, we can see a significant speedup when compared to the sequential program. This final version of the code runs 35 times faster than the sequential CPU version.

7 Future Improvements

While going through this project, following are some techniques or methods I could think of to achieve better performance.

- *Prefetching* - This technique can be used in the second kernel which computes convolution and max pool operation (the first kernel uses constant memory). Since each kernel of the filter is applied multiple times on the input, this would be a perfect setting to use the technique of prefetching.

- *Reduction* - This method serves as a perfect solution to the operations of dense layers. We used a single thread to sum over a temporary output array. Reduction techniques can be used here so as to employ more threads and therefore reduce the time taken to sum the elements.

- *Streams* - Streams can also be incorporated to achieve faster performance. Since each convolution operation has multiple filters which iterate over the input, this could be parallelized such that multiple computations can be done at the same time.

- Experiments can also be done by varying the sizes of grids and blocks to determine the optimal values and possibly achieve 100% occupancy for all the kernels.

Appendix A Kernels Using Shared Memory

```
/*Kernel corresponding to first convolutional layer that uses shared memory*/
__global__ void conv1(int *filterd, int *resultd){

  int xsize = 28;
  int filterdim = 5;

  __shared__ int fil[25];
  int i,j,l;
  int sum, offset;
  i = threadIdx.y;
  j = threadIdx.x;
  l = blockIdx.x;
  offset = l*25;
  sum =0;
  if(i<filterdim && j<filterdim){
    fil[i*filterdim+j] = filterd[offset + i*filterdim+j];

  }
  __syncthreads();
  if(i<(xsize -filterdim +1)&& j<(xsize -filterdim +1)){
    sum = fil[0]*PIC[ xsize * (i) + j ]
    + fil[1]*PIC[ xsize*(i) + (j+1) ]
      + fil[2]*PIC[ xsize * (i)+(j+2)]
      + fil[3]*PIC[xsize * (i)+(j+3)]
      + fil[4]*PIC[ xsize * (i)+(j+4)]
      + fil[5]*PIC[ xsize*(i+1)+(j) ]
      + fil[6]*PIC[ xsize * (i+1) + (j+1) ]
      + fil[7]*PIC[ xsize*(i+1) + (j+2) ] +
      fil[8]*PIC[ xsize*(i+1) + (j+3) ]
      + fil[9]*PIC[ xsize*(i+1) + (j+4) ] +
      fil[10]*PIC[ xsize*(i+2) + (j) ]
      + fil[11]*PIC[ xsize * (i+2) + (j+1) ] +
      fil[12]*PIC[ xsize*(i+2) + (j+2)]
      + fil[13]*PIC[ xsize*(i+2) + (j+3)]
      +fil[14]*PIC[ xsize*(i+2) + (j+4)]
      + fil[15]*PIC[ xsize*(i+3) + (j)]
      + fil[16]*PIC[ xsize*(i+3) + (j+1)]
      + fil[17]*PIC[ xsize*(i+3) + (j+2)]
      + fil[18]*PIC[ xsize*(i+3) + (j+3)]
      + fil[19]*PIC[ xsize*(i+3) + (j+4)]
      + fil[20]*PIC[ xsize*(i+4) + (j)]
      +fil[21]*PIC[ xsize*(i+3) + (j+1)]
      + fil[22]*PIC[ xsize*(i+4) + (j+2)]
      + fil[23]*PIC[ xsize*(i+4) + (j+3)]
      + fil[24]*PIC[ xsize*(i+4) + (j+4)];

      resultd[l*(xsize -filterdim +1)*(xsize -filterdim +1)
      + i*(xsize - filterdim +1)+j] = sum;

  }
}
```

```
/*Kernel Corresponding to First Maxpool layer that uses shared memory*/
__global__ void maxpooling(int *maxip1d, int *maxop1d){

  int i,j,l,offset;
  i = threadIdx.y;
  j = threadIdx.x;
  l = blockIdx.x;
  int xsize = 24;
  int filterdim = 5;
  offset = l*xsize*xsize;

  __shared__ int max[576];

  if(i<12 && j<12){
    max[i*2*xsize + j*2] = maxip1d[offset + i*2*xsize + j*2];
    max[i*2*xsize+j*2+1] = maxip1d[offset + i*2*xsize +j*2+1];
    max[i*2*xsize+j*2+24] = maxip1d[offset + i*2*xsize +j*2+24];
    max[i*2*xsize+j*2+25] = maxip1d[offset + i*2*xsize +j*2+25];
    // printf("i: %d,\t j: %d,\t l: %d,\t max1: %d,\t max2: %d,\t max3: %d,\t
    max4: %d\n",i,j,l,max[i*xsize + j],max[i*xsize+1],max[i*xsize+24],max[i*
    xsize+25]);

  }

  __syncthreads();

  if(i<12 && j<12){
    int max1, max2;
    if(max[i*xsize + j]>=max[i*xsize + j+1]){
      max1 = max[i*xsize + j];
    }
    else{
      max1 = max[i*xsize + j+1];
    }
    if(max[i*xsize + j+24]>=max[i*xsize + j+25]){
      max2 = max[i*xsize + j+24];
    }
    else{
      max2 = max[i*xsize + j+25];
    }
    if(max1>=max2){
      maxop1d[l*144 + i*12+j]=max1;
      // printf("Max1 : %d\t l: %d \t i: %d\t j: %d\n",max1,l,i,j);
    }
    else{
      maxop1d[l*144 + i*12+j] = max2;
      // printf("Max2 : %d\n",max2);
    }
    // printf("Maxpool1d[%d][%d]:%d\n",l,i*12+j,maxop1d[l*144 + i*12+j]);
  }
}
```

```
/*Kernel corresponding to second conv layer that uses shared memory*/
__global__ void conv2(int *cip2d, int *filter2d, int *cop2d){
  int i,j,l,sum;
  i = threadIdx.y;
  j = threadIdx.x;
  l = blockIdx.x;
  int lstar;
  lstar = l*800;

  __shared__ int fil2[800];
  __shared__ int cip2[4608];
  if(i<5 && j<5){
    for(int m = 0; m<32; m++){
      fil2[m*25 + i*5 +j] =  filter2d[lstar + m*25+ i*5 + j];
    }
  }
  if(i<8 && j<8){
    for(int m =0; m<32; m++){
      cip2[m*144 + i*12 + j] = cip2d[m*144 + i*12 + j];
    }
  }
  if(i<4 && j<4){
    for(int m = 0; m<32;m++){
      cip2[m*144 + i*12 + j+8] = cip2d[m*144 + i*12 + j+8];
      cip2[m*144 + (i+4)*12+j+8] = cip2d[m*144 + (i+4)*12 + j+8];
      cip2[m*144 + (i+8)*12 + j] = cip2d[m*144 + (i+8)*12 + j];
      cip2[m*144 + (i+8)*12 + j+4] = cip2d[m*144 + (i+8)*12 + j+4];
      cip2[m*144 + (i+8)*12 + j+8] = cip2d[m*144 + (i+8)*12 + j+8];

    }
  }
  __syncthreads();

  sum = 0;
  int di = 12;
  int disquare = di*di;
  int m;
  int k =0;
  if(i<8 && j<8){
    for(m = 0; m<32; m++){
      sum = sum + fil2[k]*cip2[(m*disquare)+ (di*i) + j] + fil2[k+1]*cip2[(m*
    disquare)+ di*(i) + (j+1)]
          + fil2[k+2]*cip2[(m*disquare)+ di*(i)+(j+2)]+fil2[k+3]*cip2[(m*disquare)
    + di*(i)+(j+3)]
          + fil2[k+4]*cip2[(m*disquare)+ di*(i)+(j+4)]+fil2[k+5]*cip2[(m*disquare)
    + di*(i+1)+(j)]
          + fil2[k+6]*cip2[(m*disquare)+ di*(i+1)+(j+1)]+fil2[k+7]*cip2[(m*
    disquare)+ di*(i+1)+(j+2)]
          + fil2[k+8]*cip2[(m*disquare)+ di*(i+1)+(j+3)]+fil2[k+9]*cip2[(m*
    disquare)+ di*(i+1) +(j+4)]
          + fil2[k+10]*cip2[(m*disquare)+ di*(i+2)+(j)]+fil2[k+11]*cip2[(m*
    disquare)+ di* (i+2) + (j+1)]
          + fil2[k+12]*cip2[(m*disquare)+ di*(i+2)+(j+2)]+fil2[k+13]*cip2[(m*
    disquare)+ di*(i+2)+(j+3)]
          + fil2[k+14]*cip2[(m*disquare)+ di*(i+2)+(j+4)]+fil2[k+15]*cip2[(m*
    disquare)+ di*(i+3)+(j)]
```

```
        + fil2[k+16]*cip2[(m*disquare)+ di*(i+3)+(j+1)]+fil2[k+17]*cip2[(m*disquare)+ di*(i+3)+(j+2)]
        + fil2[k+18]*cip2[(m*disquare)+ di*(i+3)+(j+3)]+fil2[k+19]*cip2[(m*disquare)+di*(i+3)+(j+4)]
        + fil2[k+20]*cip2[(m*disquare)+ di*(i+4)+(j)]+fil2[k+21]*cip2[(m*disquare)+ di*(i+3)+(j+1)]
        + fil2[k+22]*cip2[(m*disquare)+ di*(i+4)+(j+2)]+fil2[k+23]*cip2[(m*disquare)+ di*(i+4)+(j+3)]
        + fil2[k+24]*cip2[(m*disquare)+ di*(i+4)+(j+4)];

      k+=25;
    }
    cop2d[l*64+i*8+j] = sum;
    // printf("resultdevice[%d][%d]:%d\n",l,i*8+j,cop2d[l*64+i*8+j]);
  }
}
```

```
/*Kernel corresponding to the second maxpool layer that uses shared memory*/
__global__ void maxpool(int *maxip2d, int *maxop2d){

  int i,j,l;
  i = threadIdx.y;
  j = threadIdx.x;
  l = blockIdx.x;
  int offset;
  offset = l*64;
  int xsize = 12;
  __shared__ int max2[64];

  if(i<8 && j<8){
    max2[i*8 + j] = maxip2d[offset + i*8 +j];
  }

  __syncthreads();

  if(i<4 && j<4){
    int a,b,c,d, m1, m2;
    // index = threadIdx.x*2 + threadIdx.y*2*8;
    a = max2[i*16 + j*2];
    b = max2[i*16 + j*2 +1];
    c = max2[i*16 + j*2+8];
    d = max2[i*16 + j*2 + 9];
    if(a>=b){
      m1 = a;
    }
    else{
      m1 = b;
    }
    if(c>=d){
      m2 = c;
    }
    else{
      m2 = d;
    }
    if(m1>=m2){
      maxop2d[l*16 + i*4+j]=m1;
    }
```

```
    else{
      maxop2d[l*16 + i*4+j] = m2;
    }
    // printf("maxop2d[%d][%d]: %d\n",l, i*4+j, maxop2d[l*16 + i*4+j]);
  }
}
```

```
/*Kernel Corresponding to the first dense layer using shared memory*/
__global__ void dense1(int *denseip1d, int *weight1d, int *denseop1d){
  int i,j,l,sum;
  i=threadIdx.y;
  j=threadIdx.x;
  l = blockIdx.x;
  int k;
  int length, offset;
  length = 64*4*4;
  offset = l*64*4*4;
  sum = 0;
  __shared__ int dip1[1024];
  __shared__ int wei1[1024];
  __shared__ int dop1[1024];

  if(i<32 && j<32){
    dip1[i*32 + j] = denseip1d[i*32 + j];
    wei1[i*32 + j] = weight1d[offset + i*32+j];

  }
  __syncthreads();

  if(i<32 && j<32){
    dop1[i*32 + j] = dip1[i*32 + j]*  wei1[i*32 + j];

  }
  __syncthreads();

  if(i==0&&j==0){
    for(k=0;k<length;k++){
      sum+= dop1[k];
    }
    denseop1d[l] = sum;
  }
  // printf("denseop[%d] : %d\t i: %d\t j: %d\n",l,sum,i,j);

}
```

```
/*Kernel corresponding to the second dense layer that uses shared memory*/
__global__ void dense2(int *denseip2d, int *weight2d, int *denseop2d){
  int i,l;
  i = threadIdx.x;
  l = blockIdx.x;
  int k,sum;
  int length;
  length =64;
  sum = 0;

  __shared__ int dip2[64];
  __shared__ int wei2[64];
  __shared__ int dop2[64];
```

```
14
15    if(i<64){
16      dip2[i] = denseip2d[i];
17      wei2[i] = weight2d[l*10 + i];
18    }
19    __syncthreads();
20
21    if(i<64){
22      dop2[i] = dip2[i]*wei2[i];
23    }
24    __syncthreads();
25
26    if(i==0){
27      for(k=0;k<length;k++){
28        sum+=dop2[k];
29      }
30      denseop2d[l] = sum;
31    }
32  }
```

Appendix B Updated Kernel for better Performance

```
/*Updated Conv layer that computes the convolution and maxpool operations*/
__global__ void conv1(int *maxop1d){
  int xsize = 28;
  int filterdim = 5;

  __shared__ int res[24*24];
  int i,j,l;
  int sum, offset, offset1;
  i = threadIdx.y;
  j = threadIdx.x;
  l = blockIdx.x;
  offset = l*25;
  offset1 = xsize -filterdim +1;
  sum =0;

  if(i<offset1&& j<offset1){
    sum = FIL[0]*PIC[ xsize * (i) + j ] + FIL[1]*PIC[ xsize*(i) + (j+1) ]
      + FIL[2]*PIC[ xsize * (i)+(j+2)] + FIL[3]*PIC[xsize * (i)+(j+3)]
      + FIL[4]*PIC[ xsize * (i)+(j+4)]+ FIL[5]*PIC[ xsize*(i+1)+(j) ]
      + FIL[6]*PIC[ xsize * (i+1) + (j+1) ] + FIL[7]*PIC[ xsize*(i+1) + (j+2) ]
      + FIL[8]*PIC[ xsize*(i+1) + (j+3) ] + FIL[9]*PIC[ xsize*(i+1) + (j+4) ]
      + FIL[10]*PIC[ xsize*(i+2) + (j) ]  + FIL[11]*PIC[ xsize * (i+2) + (j+1) ]
      + FIL[12]*PIC[ xsize*(i+2) + (j+2)] + FIL[13]*PIC[ xsize*(i+2) + (j+3)]
      + FIL[14]*PIC[ xsize*(i+2) + (j+4)] + FIL[15]*PIC[ xsize*(i+3) + (j)]
      + FIL[16]*PIC[ xsize*(i+3) + (j+1)] + FIL[17]*PIC[ xsize*(i+3) + (j+2)]
      + FIL[18]*PIC[ xsize*(i+3) + (j+3)] + FIL[19]*PIC[ xsize*(i+3) + (j+4)]
      + FIL[20]*PIC[ xsize*(i+4) + (j)] +FIL[21]*PIC[ xsize*(i+3) + (j+1)]
      + FIL[22]*PIC[ xsize*(i+4) + (j+2)] + FIL[23]*PIC[ xsize*(i+4) + (j+3)]
      + FIL[24]*PIC[ xsize*(i+4) + (j+4)];
      res[ i*offset1+j] = sum;
  }
  __syncthreads();
  int offset2 = offset1/2;
  if(i<offset2 && j<offset2){
    int max1, max2;
    if(res[i*offset1*2 + j]>= res[i*offset1*2 + j+1]){
      max1 = res[i*offset1*2 + j];
    }
    else{
      max1 = res[i*offset1*2 + j+1];
    }
    if(res[i*offset1*2 + j+24]>= res[i*offset1*2 + j+25]){
      max2 = res[i*offset1*2 + j +24];
    }
    else{
      max2 = res[i*offset1*2 + j+25];
    }
    if(max1>max2){
      maxop1d[l*144 + i*12+j]=max1;
    }
    else{
      maxop1d[l*144 + i*12+j]=max2;
    }
  }
}
```

```
/*Updated kernel that computes the second convolutional operation along with the
    second maaxpool layer*/
_global__ void conv2(int *cip2d,int *filter2d, int *maxop2d){
  int i,j,l,sum;
  i = threadIdx.y;
  j = threadIdx.x;
  l = blockIdx.x;
  int lstar;
  lstar = l*800;

  __shared__ int fil2[800];
  __shared__ int cip2[4608];
  __shared__ int res2[64];
  if(i<5 && j<5){
    for(int m = 0; m<32; m++){
      fil2[m*25 + i*5 +j] =  filter2d[lstar + m*25+ i*5 + j];
    }
  }
  if(i<8 && j<8){
    for(int m =0; m<32; m++){
      cip2[m*144 + i*12 + j] = cip2d[m*144 + i*12 + j];
    }
  }
  if(i<4 && j<4){
    for(int m = 0; m<32;m++){
      cip2[m*144 + i*12 + j+8] = cip2d[m*144 + i*12 + j+8];
      cip2[m*144 + (i+4)*12+j+8] = cip2d[m*144 + (i+4)*12 + j+8];
      cip2[m*144 + (i+8)*12 + j] = cip2d[m*144 + (i+8)*12 + j];
      cip2[m*144 + (i+8)*12 + j+4] = cip2d[m*144 + (i+8)*12 + j+4];
      cip2[m*144 + (i+8)*12 + j+8] = cip2d[m*144 + (i+8)*12 + j+8];

    }
  }
  __syncthreads();

  sum = 0;
  int di = 12;
  int disquare = di*di;
  int m;
  int k =0;
  if(i<8 && j<8){
    for(m = 0; m<32; m++){
      sum = sum + fil2[k]*cip2[(m*disquare)+ (di*i) + j] + fil2[k+1]*cip2[(m*
    disquare)+ di*(i) + (j+1)]
        + fil2[k+2]*cip2[(m*disquare)+ di*(i)+(j+2)]+fil2[k+3]*cip2[(m*disquare)
    + di*(i)+(j+3)]
        + fil2[k+4]*cip2[(m*disquare)+ di*(i)+(j+4)]+fil2[k+5]*cip2[(m*disquare)
    + di*(i+1)+(j)]
        + fil2[k+6]*cip2[(m*disquare)+ di*(i+1)+(j+1)]+fil2[k+7]*cip2[(m*
    disquare)+ di*(i+1)+(j+2)]
        + fil2[k+8]*cip2[(m*disquare)+ di*(i+1)+(j+3)]+fil2[k+9]*cip2[(m*
    disquare)+ di*(i+1) +(j+4)]
        + fil2[k+10]*cip2[(m*disquare)+ di*(i+2)+(j)]+fil2[k+11]*cip2[(m*
    disquare)+ di* (i+2) + (j+1)]
        + fil2[k+12]*cip2[(m*disquare)+ di*(i+2)+(j+2)]+fil2[k+13]*cip2[(m*
    disquare)+ di*(i+2)+(j+3)]
```

```
49          + fil2[k+14]*cip2[(m*disquare)+ di*(i+2)+(j+4)]+fil2[k+15]*cip2[(m*
    disquare)+ di*(i+3)+(j)]
50          + fil2[k+16]*cip2[(m*disquare)+ di*(i+3)+(j+1)]+fil2[k+17]*cip2[(m*
    disquare)+ di*(i+3)+(j+2)]
51          + fil2[k+18]*cip2[(m*disquare)+ di*(i+3)+(j+3)]+fil2[k+19]*cip2[(m*
    disquare)+di*(i+3)+(j+4)]
52          + fil2[k+20]*cip2[(m*disquare)+ di*(i+4)+(j)]+fil2[k+21]*cip2[(m*
    disquare)+ di*(i+3)+(j+1)]
53          + fil2[k+22]*cip2[(m*disquare)+ di*(i+4)+(j+2)]+fil2[k+23]*cip2[(m*
    disquare)+ di*(i+4)+(j+3)]
54          + fil2[k+24]*cip2[(m*disquare)+ di*(i+4)+(j+4)];
55
56        k+=25;
57      }
58      res2[i*8+j] = sum;
59    }
60    __syncthreads();
61    if(i<4 && j<4){
62      int max1, max2;
63      if(res2[i*16+j*2]>=res2[i*16+j*2+1]){
64        max1  = res2[i*16+j*2];
65      }
66      else{
67        max1  = res2[i*16+j*2+1];
68      }
69      if(res2[i*16+j*2+8]>=res2[i*16+j*2+9]){
70        max2  = res2[i*16+j*8];
71      }
72      else{
73        max2  = res2[i*16+j*2+9];
74      }
75      if(max1>=max2){
76        maxop2d[l*16 + i*4+j]=max1;
77      }
78      else{
79        maxop2d[l*16 + i*4+j] = max2;
80      }
81    }
82  }
```