Names: Yen-Wen Lu, Ming-Cheng Liao, Yu-Tang Su
Netid: ywlu2, mcliao2, ytsu2
rai ids: 5d97b1e588a5ec28f9cb9493
5d97b1df88a5ec28f9cb9488
5d97b21188a5ec28f9cb94e5
Team name: dpp
School affiliation: on-campus

1. Optimization Approach and Results

- **Optimization 4 (in Line 559 in new-forward.cuh): Unroll + shared-memory Matrix multiply + Exploiting parallelism in input images, input channels, and output channels.**

  - **how you identified the optimization opportunity**

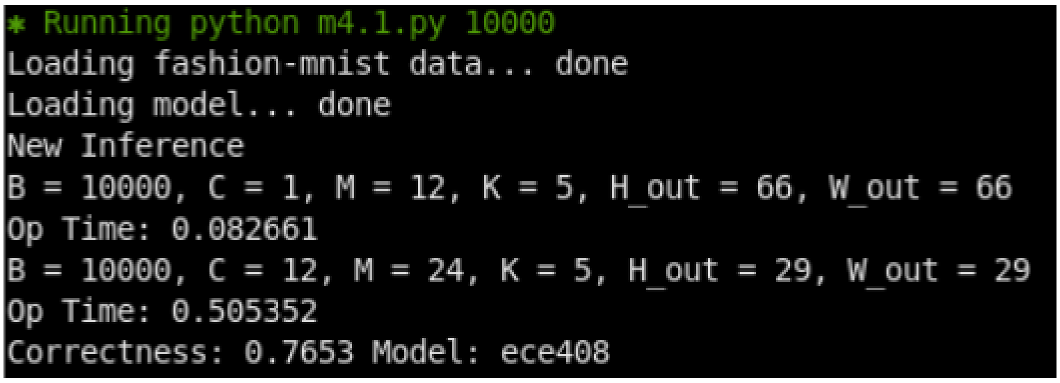
    In milestone2, Matrix multiplication was implemented with three four loops, which indicated that lots of multiplication was not paralleled. Therefore, we think that Unroll X is able to reduce for loops and then accelerate the computation. Moreover, we tried to launch a kernel with two dimensional grid (X dimension is the size of unroll x and Y dimension is the size of batch) to calculate unroll X with all batch simultaneously, which is also the exploiting parallelism in input images optimization. We think it would faster than using for loop to compute each images in each batch. In addition, in milestone2, we accessed data from global memory which was slow. It would be faster if we accessed data from shared memory when doing matrix multiplication.

  - **Why do you thought the approach would be fruitful**

    Unrolling X is able to effectively reduce lots of iteration in kernel function. Therefore, we believe unrolling X is a fruitful method.
    Weight Matrices and unrolling X were stored in shared memories when doing matrix multiplication, which can be accessed faster by threads than those stored in global. Exploiting parallelism in input images, input channels, and output channels is faster since threads implemented each elements wise multiplication simultaneously than using for loop of each input images, input channels, and output channels to implemented matrix multiplication.

  - **Result**

    Fig1. Indicates that this method is slower compared to milestone 2.1. The possible reason is that global memory is not enough to store unroll X. We had to use three kernels to load part of batch images separately. We separated 10000 batch size into 4000, 4000, 2000 to do the Unroll + shared-memory Matrix multiplication. Since we called kernel function for three times, op time will be slower than the op time in milestone 2.1. From Fig2, though kernel function run quite fast, three cudamalloc and cudaFree operation spent about 0.4s, which is the bottle neck in all operation time and it is the main reason why optimization 4 is slower than milestone2.



```
✱ Running python m4.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
B = 10000, C = 1, M = 12, K = 5, H_out = 66, W_out = 66
Op Time: 0.082661
B = 10000, C = 12, M = 24, K = 5, H_out = 29, W_out = 29
Op Time: 0.505352
Correctness: 0.7653 Model: ece408
```

Fig 1. Op time of Optimization 4

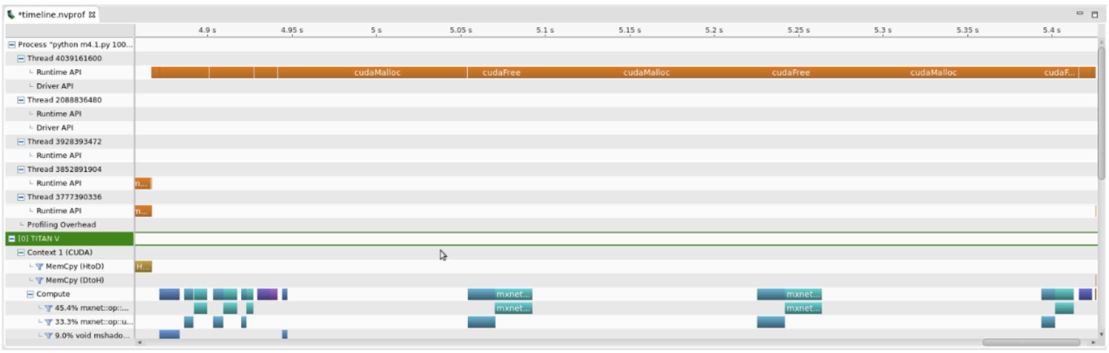


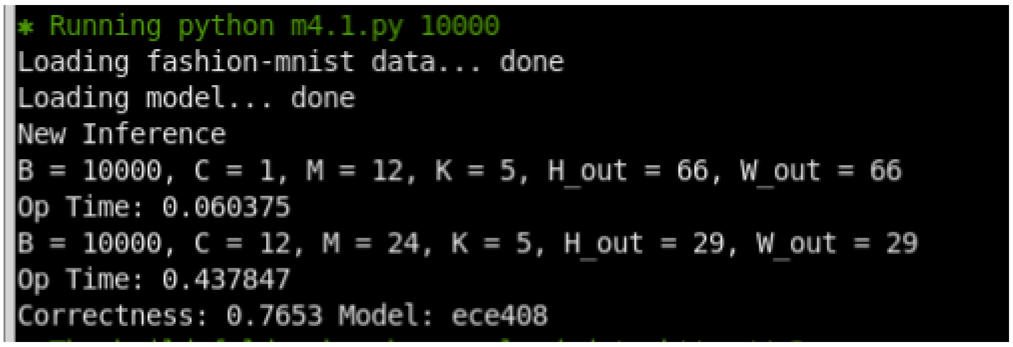
Fig 2. NVPP Profile of Optimization 4

- **Optimization 5 (in Line 627 in new-forward.cuh): Unroll + <u>Register-Tile</u> Matrix multiply + Exploiting parallelism in input images, input channels, and output channels.**
  - **how you identified the optimization opportunity**

    From lecture slides of "Lecture 3: Thread Coarsening and Register Tiling" in ECE508[1], Registers do not require memory access instructions, which indicates high throughput. Moreover, this optimization is implemented with thread coarsening, so the computation from merged threads can now share registers.
  - **Why do you think the approach would be fruitful**

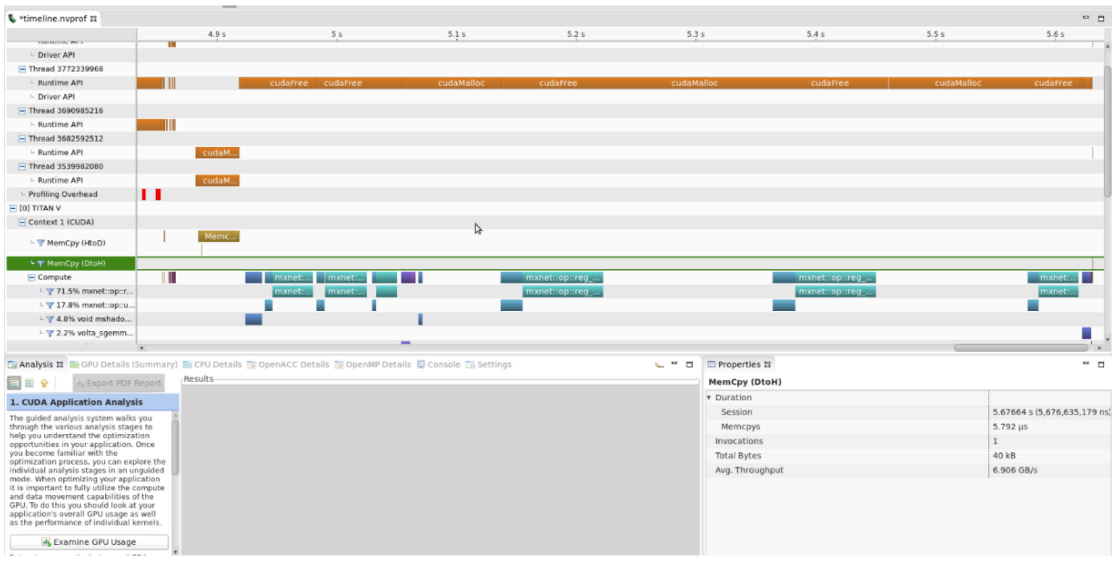
    Since registers is not capable of all data from global memory. What we did is loading weight matrix in register and loading x into shared memory and then doing the matrix multiplication. Since registers' throughput is extremely high, this approach would be fruitful.
  - **Result**

    In theory, register should be much faster. However, the result in Fig 3. Showed that optimization 5 is slower than Optimization 4. From Fig 4. , bottle neck is executing kernel function, which spent longer time than Optimization 4. The possible reason for longer execution time is that the kernel spent too much time loading unroll X into shared memory, although the kernel accessed weight matrix fast form registers.



```
* Running python m4.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
B = 10000, C = 1, M = 12, K = 5, H_out = 66, W_out = 66
Op Time: 0.060375
B = 10000, C = 12, M = 24, K = 5, H_out = 29, W_out = 29
Op Time: 0.437847
Correctness: 0.7653 Model: ece408
```

Fig 3. Op time of Optimization 5



1. CUDA Application Analysis

The guided analysis system walks you through the various analysis stages to help you understand the optimization opportunities in your application. Once you become familiar with the optimization process, you can explore the individual analysis stages in an unguided mode. When optimizing your application it is important to fully utilize the compute and data movement capabilities of the GPU. To do this you should look at your application's overall GPU usage as well as the performance of individual kernels.

MemCpy (DtoH)

▾ Duration	
Session	5.67664 s (5,676,635,179 ns)
Memcpys	5.792 μs
Invocations	1
Total Bytes	40 kB
Avg. Throughput	6.906 GB/s

Fig 4. NVPP Profile of Optimization 5

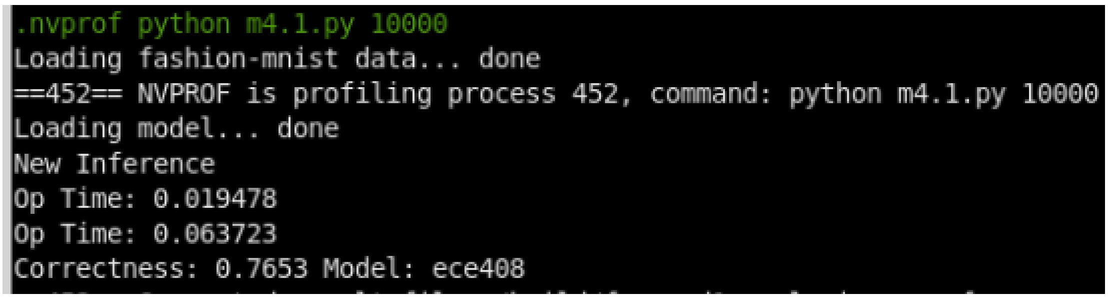
- **Optimization 6 (in Line 689 in new-forward.cuh): Kernel fusion for unrolling and matrix-multiplication (unrolling X into shared memory directly)**
  - **How you identified the optimize opportunity**

    We thought that combining unrolling X kernel and matrix multiplication into one kernel would be faster since this only did memcpy from host and device and memcpy from device to host one time. Moreover, we were able to do unrolling X, store part of X into shared memory and matrix multiplication in a loop.
  - **Why do you think the approach would be fruitful?**

    Kernel fusion combines two kernels (unrolling and matrix-multiplication) into one kernel. This method only takes one time to allocate memory on device and announce block number and block size. Therefore, it is able to accelerate the computation compared to Optimization 4 which implemented two kernels

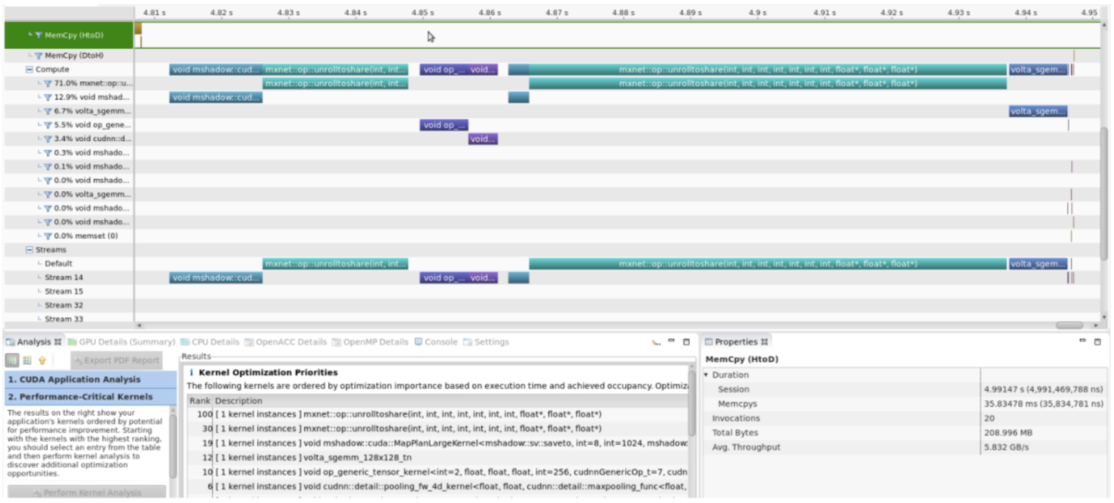
- **Result**

  Fig5. indicates that this kernel fusion method is faster than milestone 2. There are two reasons why this optimization method is much faster. First, we loaded part of data into data for one time and compute part of elements, so there will not be insufficient memory problem. Second, our kernel fusion implemented unrolling and matrix multiplication in the same loop, which is faster than separating unrolling X and matrix multiplication.



```
.nvprof python m4.1.py 10000
Loading fashion-mnist data... done
==452== NVPROF is profiling process 452, command: python m4.1.py 10000
Loading model... done
New Inference
Op Time: 0.019478
Op Time: 0.063723
Correctness: 0.7653 Model: ece408
```

Fig 5. Op time of Optimization 6



Kernel Optimization Priorities

The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimiz

Rank	Description
100	[1 kernel instances] mxnet::op::unrolltoshare(int, int, int, int, int, int, int, float*, float*, float*)
30	[1 kernel instances] mxnet::op::unrolltoshare(int, int, int, int, int, int, int, float*, float*, float*)
19	[1 kernel instances] void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024, mshadow:
12	[1 kernel instances] volta_sgemm_128x128_tn
10	[1 kernel instances] void op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGenericOp_t=7, cudn
6	[1 kernel instances] void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float,

MemCpy (HtoD)

Duration	
Session	4.99147 s (4,991,469,788 ns)
Memcpys	35.83478 ms (35,834,781 ns)
Invocations	20
Total Bytes	208.996 MB
Avg. Throughput	5.832 GB/s

Fig 6. NVPP Profile of Optimization

- **Optimization 7 (in Line 507 in new-forward.cuh): Sweeping various parameters to find best values (Optimization 7 is the best method)**

  - **How you identified the optimization opportunity**

    Certain block sizes or other parameters may avoid control divergence problem or memory not coalescing. Moreover, if setting the block size too large, it may cause illegal memory access problem. Therefore, sweeping various parameters to find best values is able to prevent above problems to occur and improve computation speed.

  - **Why do you thought the approach would be fruitful**

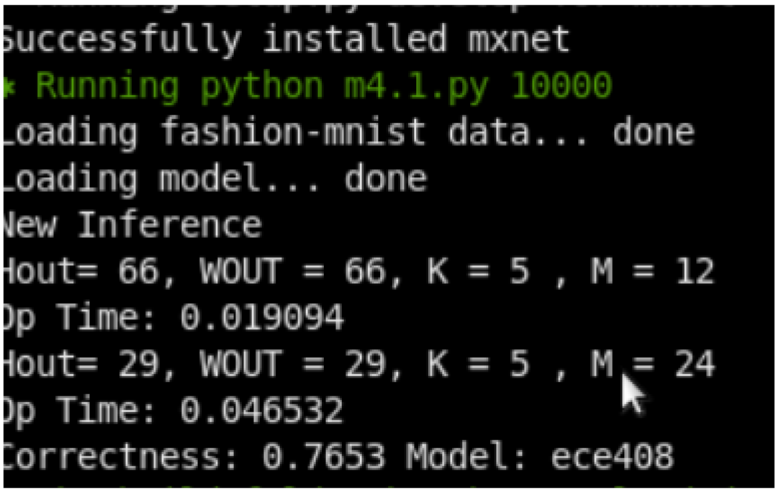
We found that Op1 and Op2 s' best block size for Op time are different. Our method was to test each Ops' best block size. Since Op1 is for input channel equaling 1 and Op2 is for input channel equaling 12. We used if statement to control different block size for different Op. Furthermore, we found that using __restrict__ to one parameter for the whole kernel function is better for optimization than using __restrict__ to all parameters. That is why we think this approach is fruitful.

■ Result

Table 2. shows that our all testing with different block size and optimization method. From the "Total Op time" we can see that kernel fusion method with Block size equaling 24 and kernel fusion with Block equaling 32 had the shortest Op time. However, from Fig 9. And Fig 10. , which is divergent branches in nvpp profile, it shows that our code exists control divergence. Therefore, Op time may be shorter if we could solve this control divergence problem.

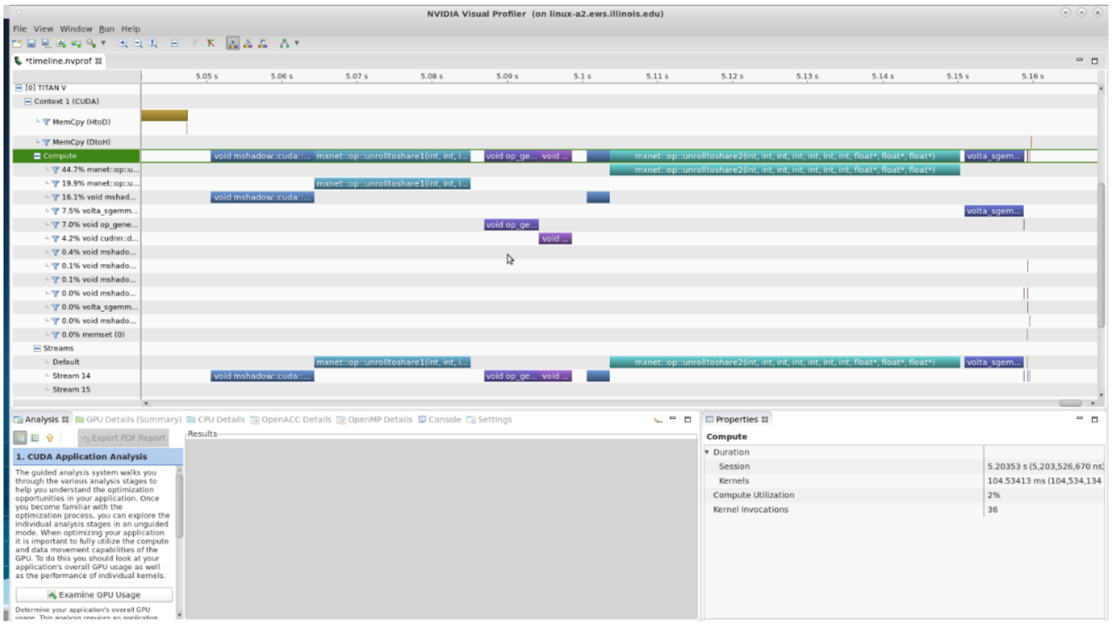
	Optimization Method(Op1)	**Optimization Method(Op2)**	**Block Size(Op1)**	**Block Size(Op2)**	**Total Op Time(sec)**
Test 1	**Baseline method**	**Baseline method**	**16**	**16**	**0.115309**
Test 2	**Baseline method**	**Baseline method**	**16**	**32**	**0.085582**
Test 3	**Fusion method**	**Fusion Method**	**16**	**32**	**0.087712**
Test 4	**Fusion method**	**Fusion method**	**6**	**6**	**0.223641**
Test 5	**Fusion method**	**Fusion method**	**12**	**12**	**0.091421**
Test 6	**Fusion method**	**Fusion method**	**18**	**18**	**0.104445**
Test 7	**Fusion method**	**Fusion method**	**24**	**24**	**0.089376**
Test 8	**Fusion method**	**Fusion method**	**32**	**32**	**0.105804**
Test 9	**Fusion method**	**Fusion method**	**16**	**32**	**0.078843**
Test 10	**Fusion method**	**Fusion method**	**16**	**24**	**0.065626**

Table 2. Different parameters test



```
Successfully installed mxnet
* Running python m4.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Hout= 66, WOUT = 66, K = 5 , M = 12
Op Time: 0.019094
Hout= 29, WOUT = 29, K = 5 , M = 24
Op Time: 0.046532
Correctness: 0.7653 Model: ece408
```

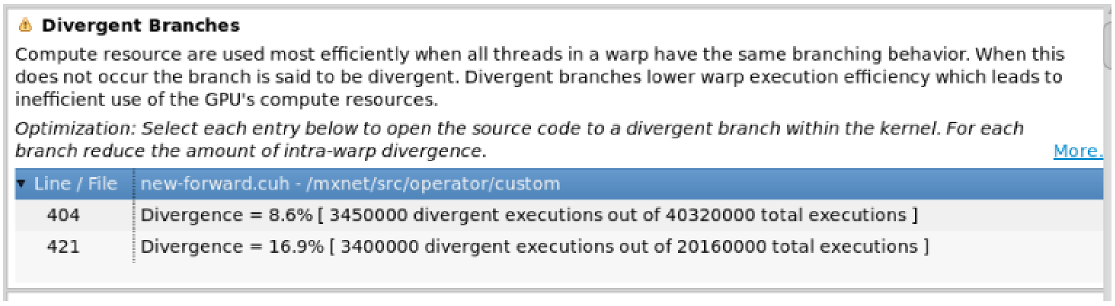
Fig 7. Op time of Optimization 7



Compute

▾ Duration	
Session	5.20353 s (5,203,526,670 ns)
Kernels	104.53413 ms (104,534,134
Compute Utilization	2%
Kernel Invocations	36

Fig 8. Op time of Optimization 7



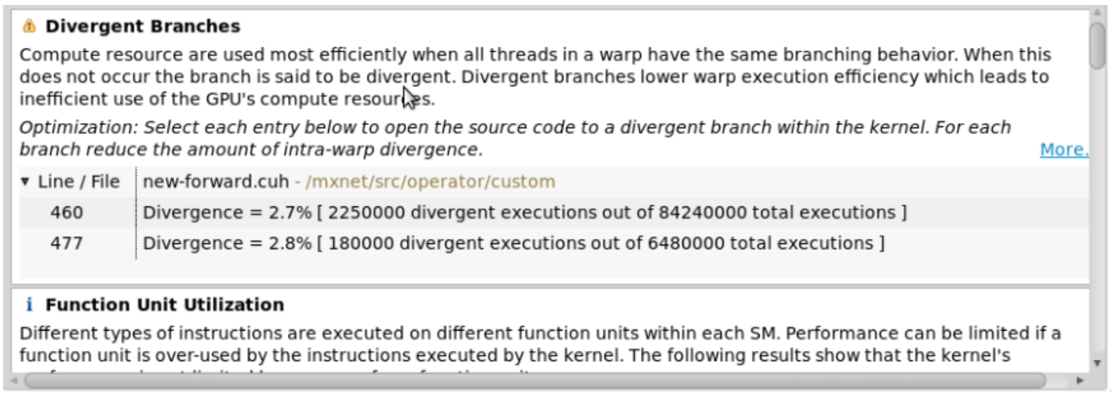
Divergent Branches

Compute resource are used most efficiently when all threads in a warp have the same branching behavior. When this does not occur the branch is said to be divergent. Divergent branches lower warp execution efficiency which leads to inefficient use of the GPU's compute resources.

Optimization: Select each entry below to open the source code to a divergent branch within the kernel. For each branch reduce the amount of intra-warp divergence. More...

▾ Line / File	new-forward.cuh - /mxnet/src/operator/custom
404	Divergence = 8.6% [3450000 divergent executions out of 40320000 total executions]
421	Divergence = 16.9% [3400000 divergent executions out of 20160000 total executions]

Fig 9. Divergent Branches for Op1


Divergent Branches

Compute resource are used most efficiently when all threads in a warp have the same branching behavior. When this does not occur the branch is said to be divergent. Divergent branches lower warp execution efficiency which leads to inefficient use of the GPU's compute resources.

Optimization: Select each entry below to open the source code to a divergent branch within the kernel. For each branch reduce the amount of intra-warp divergence. More.

Line / File	new-forward.cuh - /mxnet/src/operator/custom
460	Divergence = 2.7% [2250000 divergent executions out of 84240000 total executions]
477	Divergence = 2.8% [180000 divergent executions out of 6480000 total executions]

Function Unit Utilization

Different types of instructions are executed on different function units within each SM. Performance can be limited if a function unit is over-used by the instructions executed by the kernel. The following results show that the kernel's

Fig 10. Divergent Branches for Op2

1.1 implemented Optimization

Table 2. shows which optimization methods listed on github were implemented in our optimization kernels.

Table 3. marks each optimization's position in new-forward.cuh.

Optimization / Optimization method in ECE 408 project github	Optimization 4	Optimization 5	Optimization 6	Optimization 7
Unroll + shared-memory Matrix multiply	**V**			
Kernel fusion for unrolling and matrix-multiplication			**V**	**V**
register-tiled		**V**		
Sweeping various parameters to find best values				**V**
Exploiting parallelism in input images, input channels, and output channels	**V**	**V**	**V**	**V**

Table 2. Optimization List

Optimization	**Optimization 4**	**Optimization 5**	**Optimization 6**	**Optimization 7**
Line number in new-forward.cuh	**559**	**627**	**689**	**507**

Table 3. Line number of each Optimization in new-forward.cuh

2. **References** (as needed)

[1] W. Hwu, “Lecture 4: Joint Register and Shared-Memory Tiling” ECE 508 Lecture Slides, 2019.

[2] W. Hwu, “Application Case Study –Machine Learning” ECE 408 Lecture Slides, 2019.

3. Contribution: How your team organized and divided up this work

Team member	Contribution
Ming-Cheng Liao	Optimization 4, Optimization 5, Optimization 6, Optimization 7
Yen-Wen Lu	Optimization 4, Optimization 6, Optimization 7, Final Report
Yu-Tang Su	Optimization 4, Optimization 7

4. Previous Report

1. Include a list of all kernels that collectively consume more than 90% of the program time

CUDA memcpy HtoD
volta_scudnn_128x64_relu_interior_nn_v1
volta_gcgemm_64x32_nt
fft2d_c2r_32x32
volta_sgemm_128x128_tn
op_generic_tensor_kernel
fft2d_r2c_32x32

2. Include a list of all CUDA API calls that collectively consume more than 90% of the program time.

cudaMemGetInfo
cudaStreamCreateWithFlags
cudaFree

3. Explanation of the difference between kernels and API calls

Kernels are the function that executed on GPU. API calls are the functions used to interact between host and device.

4. Show output of rai running MXNet on the CPU

```
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8154}
```

5. List program run time

4.97s

6. Show output of rai running MXNet on the GPU

```
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8154}
```

7. List program run time

4.66s

8. Create a CPU implementation

```
void forward(mshadow::Tensor<cpu, 4, DType> &y, const
mshadow::Tensor<cpu, 4, DType> &x, const
mshadow::Tensor<cpu, 4, DType> &k)
{
    /*
    Modify this function to implement the forward pass
described in Chapter 16.
    The code in 16 is for a single image.
    We have added an additional dimension to the tensors
to support an entire mini-batch
    The goal here is to be correct, not fast (this is the
CPU implementation.)
    */

    const int B = x.shape_[0];
    const int M = y.shape_[1];
    const int C = x.shape_[1];
    const int H = x.shape_[2];
    const int W = x.shape_[3];
    const int K = k.shape_[3];

    for (int b = 0; b < B; ++b) {
        for(int m = 0; m < M; ++m){
            for(int h = 0; h < H; ++h){
                for(int w = 0; w < W; ++w){
                    y[b][m][h][w] = 0;
                    for(int c = 0; c < C; ++c){
                        for(int p = 0; p < K; ++p){
                            for(int q = 0; q< K; ++q){
                                y[b][m][h][w] +=
x[b][c][h + p][w + q] * k[m][c][p][q];
                            }
                        }
                    }
                }
            }
        }
    }
}
```

9. List whole program execution time

m1.1.py:
user time = 24.66
system time = 5.48
elapsed time = 0:11.35

m1.2.py:
user time = 4.99
system time = 3.34
elapsed time = 0:04.73

m2.1.py:
user time = 105.00
system time = 10.65
elapsed time = 1:33.88

10. List Op Times
optime1: 11s
optime2: 78s

Implement a GPU Convolution

```
#ifndef MXNET_OPERATOR_NEW_FORWARD_CUH_
#define MXNET_OPERATOR_NEW_FORWARD_CUH_

#include <mxnet/base.h>

# define TILE_WIDTH 16
namespace mxnet
{
namespace op
{

__global__ void forward_kernel(float *y, const float *x, const float *k, const int B,
const int M, const int C, const int H, const int W, const int K)
{

    /*
    Modify this function to implement the forward pass described in Chapter 16.
    We have added an additional dimension to the tensors to support an entire mini-
batch
    The goal here is to be correct AND fast.
    We have some nice #defs for you below to simplify indexing. Feel free to use
them, or create your own.
    */

    const int H_out = H - K + 1;
    const int W_out = W - K + 1;
    (void)H_out; // silence declared but never referenced warning. remove this line
when you start working
    (void)W_out; // silence declared but never referenced warning. remove this line
when you start working

// An example use of these macros:
// float a = y4d(0,0,0,0)
// y4d(0,0,0,0) = a
#define y4d(i3, i2, i1, i0) y[(i3) * (M * H_out * W_out) + (i2) * (H_out * W_out) +
(i1) * (W_out) + i0]
#define x4d(i3, i2, i1, i0) x[(i3) * (C * H * W) + (i2) * (H * W) + (i1) * (W) + i0]
```

```
#define k4d(i3, i2, i1, i0) k[(i3) * (C * K * K) + (i2) * (K * K) + (i1) * (K) + i0]
    int W_grid = ceil((float)W_out / TILE_WIDTH);
    const int b = blockIdx.x;
    const int m = blockIdx.y;
    const int h = blockIdx.z / W_grid * TILE_WIDTH + threadIdx.y;
    const int w = blockIdx.z % W_grid * TILE_WIDTH + threadIdx.x;
    if(h < H_out && w < W_out){
        float val = 0;
        for(int c = 0; c < C; c++){
            for(int p = 0; p < K; p++){
                for(int q = 0; q < K; q++){
                    val += x4d(b, c, h+p, w+q) * k4d(m, c, p, q);
                }
            }
        }
        y4d(b, m, h, w) = val;
    }



#undef y4d
#undef x4d
#undef k4d
}

/*
   This function is called by new-inl.h
   Any code you write should be executed by this function.
   For ECE408, we only expect the float version of the operator to be called, so here
we specialize with only floats.
*/
template <>
void forward<gpu, float>(mshadow::Tensor<gpu, 4, float> &y, const
mshadow::Tensor<gpu, 4, float> &x, const mshadow::Tensor<gpu, 4, float> &w)
{

    // Use mxnet's CHECK_EQ to do assertions.
    // Remove this assertion when you do your implementation!
    //CHECK_EQ(0, 1) << "Remove this line and replace with your
implementation";

    // Extract the tensor dimensions into B,M,C,H,W,K
    // ...



    const int B = x.shape_[0];
    const int C = x.shape_[1];
    const int H = x.shape_[2];
    const int W = x.shape_[3];

    const int M = y.shape_[1];
    const int K = w.shape_[3];

    int H_out = H - K + 1;
    int W_out = W - K + 1;

    int W_grid = ceil(1.0*W_out / TILE_WIDTH);
    int H_grid = ceil(1.0*H_out / TILE_WIDTH);

    int Z = H_grid * W_grid;

    // Set the kernel dimensions
    dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1);
    dim3 gridDim(B, M, Z);



    // Call the kernel
    //forward_kernel<<<gridDim,
blockDim,0,s>>>(y.dptr_,x.dptr_,w.dptr_,B,M,C,H,W,K);
    forward_kernel<<<gridDim,
blockDim,0,0>>>(y.dptr_,x.dptr_,w.dptr_,B,M,C,H,W,K);
    // Use MSHADOW_CUDA_CALL to check for CUDA runtime errors.
    MSHADOW_CUDA_CALL(cudaDeviceSynchronize());

}
```

```
/*
	This tells mxnet how to do an op when it's not a float.
	This is not used in the ECE408 project
*/
template <typename gpu, typename DType>
void forward(mshadow::Tensor<gpu, 4, DType> &y, const mshadow::Tensor<gpu, 4,
DType> &x, const mshadow::Tensor<gpu, 4, DType> &w)
{
	//CHECK_EQ(0,1) << "Remove this line and replace it with your
implementation.";
}
}
}

#endif
```

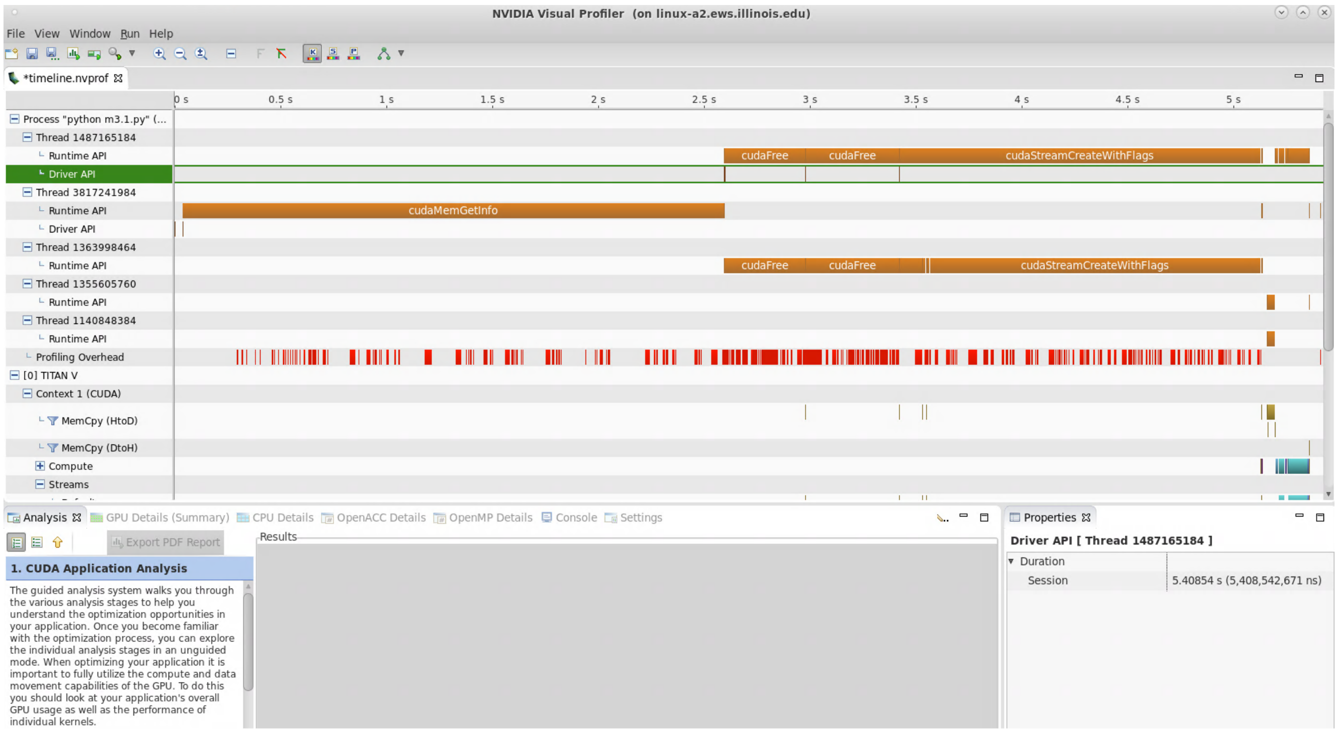
Correctness and timing with 3 different dataset sizes

```
✱ Running python m3.1.py 100
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.000270
Op Time: 0.000931
Correctness: 0.76 Model: ece408
✱ Running python m3.1.py 1000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.002707
Op Time: 0.009456
Correctness: 0.767 Model: ece408b
✱ Running python m3.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.028302
Op Time: 0.087000
Correctness: 0.7653 Model: ece408
```

∗ Running nvprof python m3.1.py

```
Loading fashion-mnist data... done
==265== NVPROF is profiling process 265, command: python m3.1.py
Loading model... done
New Inference
Op Time: 0.032408
Op Time: 0.097270
Correctness: 0.7653 Model: ece408
```

Demonstrate nvprof profiling the execution



1. CUDA Application Analysis

The guided analysis system walks you through the various analysis stages to help you understand the optimization opportunities in your application. Once you become familiar with the optimization process, you can explore the individual analysis stages in an unguided mode. When optimizing your application it is important to fully utilize the compute and data movement capabilities of the GPU. To do this you should look at your application's overall GPU usage as well as the performance of individual kernels.

Driver API [Thread 1487165184]

Duration	
Session	5.40854 s (5,408,542,671 ns)

From this picture, we can directly see the timeline of each api like cudaFree and cudaMemGetinfo. Therefore, we can easily understand which part is the bottleneck of the process. In this case, it is cudaMemGetInfo.

Milestone4:

Implement final GPU optimizations

Optimization our team used :

a. Shared Memory convolution
b. Weight matrix (kernel values) in constant memory
c. Tuning with restrict and loop unrolling

Implementation (we combined these 3 method into one kernel function and used new-forward.cuh):

```
ifndef MXNET_OPERATOR_NEW_FORWARD_CUH_
#define MXNET_OPERATOR_NEW_FORWARD_CUH_
```

```
#include <mxnet/base.h>

namespace mxnet
{
namespace op
{

#define y4d(i3, i2, i1, i0) y[(i3) * (M * H_out * W_out) + (i2) * (H_out * W_out) +
(i1) * (W_out) + i0]
#define x4d(i3, i2, i1, i0) x[(i3) * (C * H * W) + (i2) * (H * W) + (i1) * (W) + i0]
#define k4d(i3, i2, i1, i0) kernel[(i3) * (C * K * K) + (i2) * (K * K) + (i1) * (K) + i0]
//#define MIN( a, b ) ( (a < b) ? a : b )


constexpr int TILE_WIDTH = 8;
__constant__ float kernel[14112];
__global__ void forward_kernel(float * __restrict__ y, const float * __restrict__ x,
const int B, const int M, const int C, const int H, const int W, const int K, int W_grid)
{



    // use stratege 3
    __shared__ float x_tile[TILE_WIDTH][TILE_WIDTH];

    int H_out = H - K + 1;
    int W_out = W - K + 1;

    int tx = threadIdx.x; int ty = threadIdx.y;
    int bx = blockIdx.x; int by = blockIdx.y;

    int w = blockIdx.z % W_grid * TILE_WIDTH + tx;
    int h = blockIdx.z / W_grid * TILE_WIDTH + ty;

    int tile_begin_x = blockIdx.z % W_grid * TILE_WIDTH;
    int tile_end_x = tile_begin_x + TILE_WIDTH;
    int tile_begin_y = blockIdx.z / W_grid * TILE_WIDTH;
    int tile_end_y = tile_begin_y + TILE_WIDTH;

    int shift = K/2;

    float val = 0;
    #pragma unroll 2
    for (int c = 0; c < C; c++) {

        //Loading core part of input into shared matrix
        if (w < W_out && h < H_out) {
            x_tile[ty][tx] = x4d(bx, c, h + shift, w + shift);
        } else {
            x_tile[ty][tx] = 0;
        }
        __syncthreads();



        //Set Boundary Condition
        int input_begin_x = tile_begin_x + shift;
        int input_end_x = tile_end_x + shift;
        int input_begin_y = tile_begin_y + shift;
        int input_end_y = tile_end_y + shift;

        for (int i = 0; i < K; i++) {
            for (int j = 0; j < K; j++) {
                int input_glob_x = ((w-shift)+shift) + i;    //first shifting to input
coord., and then staring convolution from the left top corner
                int input_glob_y = ((h-shift)+shift) + j;    //first shifting to input
coord., and then staring convolution from the left top corner
                int input_share_x = tx+i-shift;
                int input_share_y = ty+j-shift;
                if (input_glob_x < W && input_glob_y < H) {
                    int x_min = ( (input_end_x < W_out + shift) ? input_end_x :
W_out + shift );
                    int y_min = ( (input_end_y < W_out + shift) ? input_end_y :
W_out + shift );

                    //using shared memory data
                    if (input_glob_x >= input_begin_x && input_glob_x <
x_min && input_glob_y >= input_begin_y && input_glob_y < y_min) {
                        val += x_tile[input_share_y][input_share_x] * k4d(by,
c, j, i);
                    }
                    else {
                        //using global memory data
                        val += x4d(bx, c, input_glob_y, input_glob_x) *
k4d(by, c, j, i);
                    }
                }
            }
        }
        __syncthreads(); // make sure all threads finish its computation before
loading new data into shared memory
    }

    if (w < W_out && h < H_out) {
        y4d(bx, by, h, w) = val;
    }

}

/*
   This function is called by new-inl.h
   Any code you write should be executed by this function.
   For ECE408, we only expect the float version of the operator to be called, so here
we specialize with only floats.
*/
template <>
void forward<gpu, float>(mshadow::Tensor<gpu, 4, float> &y, const
mshadow::Tensor<gpu, 4, float> &x, const mshadow::Tensor<gpu, 4, float> &w)
{

    // Extract the tensor dimensions into B,M,C,H,W,K
    const int B = x.shape_[0];      // batch size
    const int C = x.shape_[1];      // input channel
    const int H = x.shape_[2];      // height of the input feature map
    const int W = x.shape_[3];     // width of the input feature map
    const int M = y.shape_[1];     // output channel
    const int K = w.shape_[3];     // the size of the filter (assuming squared filter)
    //printf("K[0]=%d\nK[1]=%d\nK[2]=%d\nK[3]=%d\n", w.shape_[0],
w.shape_[1], w.shape_[2], w.shape_[3]);


    cudaMemcpyToSymbol(kernel, w.dptr_, sizeof(float) * w.shape_[0] *
w.shape_[1] * w.shape_[2] * w.shape_[3], 0, cudaMemcpyHostToDevice);
    // Set the kernel dimensions
    int H_out = H - K + 1;
    int W_out = W - K + 1;

    int H_grid = ceil(H_out / (1.0 * TILE_WIDTH));
    int W_grid = ceil(W_out / (1.0 * TILE_WIDTH));
    int Z = H_grid * W_grid;

    dim3 gridDim(B, M, Z);
    dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1);

    forward_kernel<<<gridDim, blockDim>>>(y.dptr_, x.dptr_, B, M, C, H, W, K,
W_grid);
    // Use MSHADOW_CUDA_CALL to check for CUDA runtime errors.
    MSHADOW_CUDA_CALL(cudaDeviceSynchronize());

}

/*
    This tells mxnet how to do an op when it's not a float.
    This is not used in the ECE408 project
*/
template <typename gpu, typename DType>
void forward(mshadow::Tensor<gpu, 4, DType> &y, const mshadow::Tensor<gpu, 4,
DType> &x, const mshadow::Tensor<gpu, 4, DType> &w)
{
    CHECK_EQ(0,1) << "Remove this line and replace it with your
implementation.";
}
```

```
}
}

#endif
```

Describe and analyze the optimizations:

We integrated three optimization into on kernel. First, we loaded kernel matrix into constant memory, and then we implemented strategy 3 for convolution. For pointer y and pointer x, we added __restricted__ to do tuning with restrict. For for loop in the kernel function, we implemented #pragma unroll 2 to do loop unrolling.

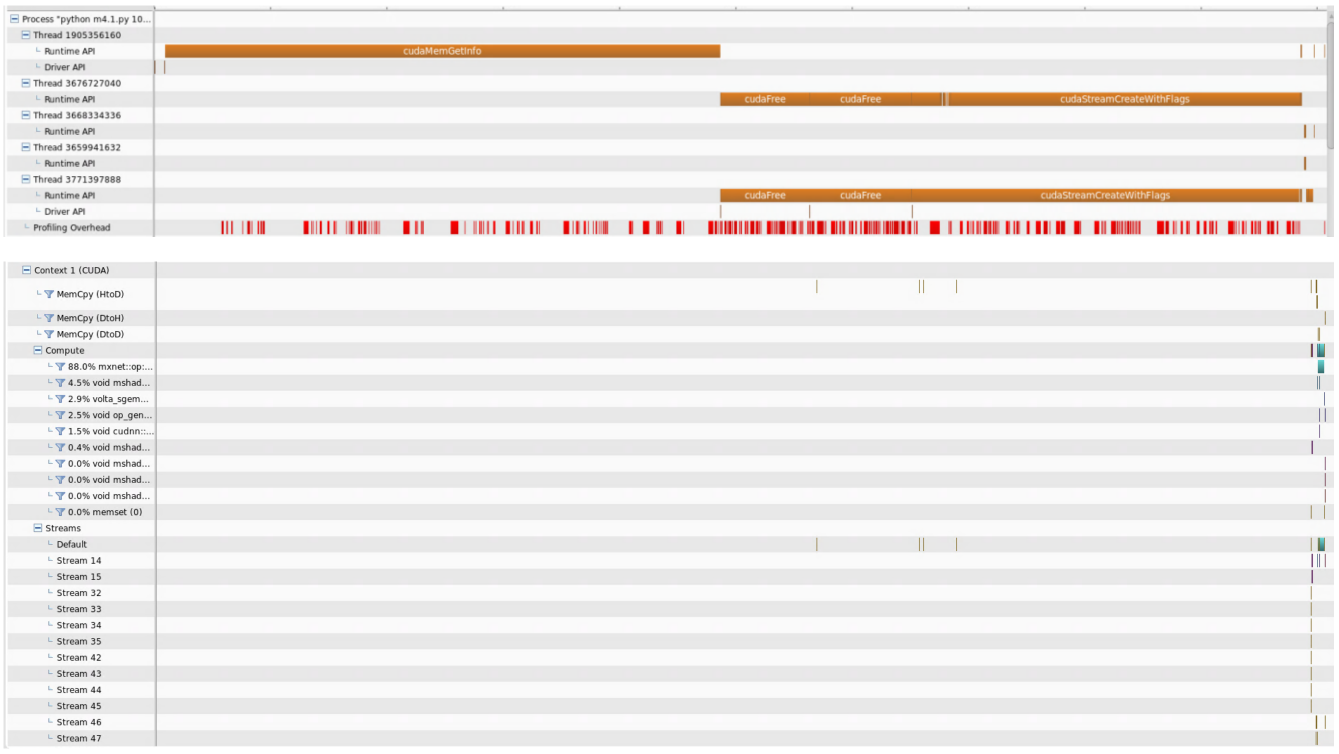
Correctness and timing with 3 different dataset sizes

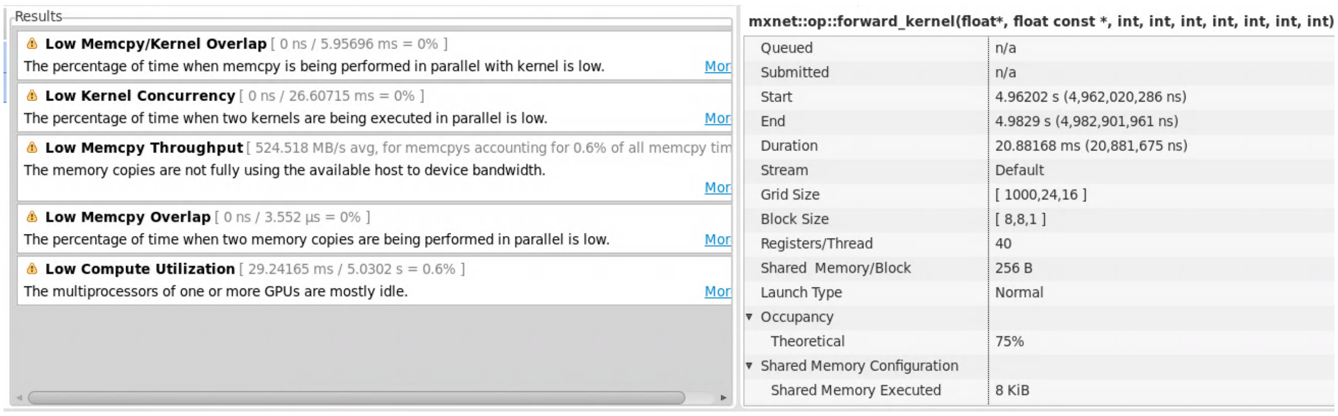
```
✱ Running python m4.1.py 100
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.000459
Op Time: 0.001862
Correctness: 0.76 Model: ece408

✱ Running python m4.1.py 1000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.004259
Op Time: 0.018774
Correctness: 0.767 Model: ece408

✱ Running python m4.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.057039
```

Op Time: 0.190717

Correctness: 0.7653 Model: ece408

use NVVP to analyze your optimization





Results

Low Memcpy/Kernel Overlap [0 ns / 5.95696 ms = 0%]
The percentage of time when memcpy is being performed in parallel with kernel is low. Mor

Low Kernel Concurrency [0 ns / 26.60715 ms = 0%]
The percentage of time when two kernels are being executed in parallel is low. Mor

Low Memcpy Throughput [524.518 MB/s avg, for memcpys accounting for 0.6% of all memcpy tim
The memory copies are not fully using the available host to device bandwidth. Mor

Low Memcpy Overlap [0 ns / 3.552 µs = 0%]
The percentage of time when two memory copies are being performed in parallel is low. Mor

Low Compute Utilization [29.24165 ms / 5.0302 s = 0.6%]
The multiprocessors of one or more GPUs are mostly idle. Mor

mxnet::op::forward_kernel(float*, float const *, int, int, int, int, int, int, int)

Queued	n/a
Submitted	n/a
Start	4.96202 s (4,962,020,286 ns)
End	4.9829 s (4,982,901,961 ns)
Duration	20.88168 ms (20,881,675 ns)
Stream	Default
Grid Size	[1000,24,16]
Block Size	[8,8,1]
Registers/Thread	40
Shared Memory/Block	256 B
Launch Type	Normal
▾ Occupancy	
Theoretical	75%
▾ Shared Memory Configuration	
Shared Memory Executed	8 KiB

From “Examine Individual Kernels” in NVPP, There is no control divergence in my code. However, memcpy spent most of time and the parallelism of memcpy is not good. This may be the reason why our team’s optimization were not up to our expectation.