Team name: greedybluemoon

Group member: Yichen Zhou, Feifan Zhang

netid: zhou113, fzhang19

Milestone 1:

list of all kernels that collectively consume more than 90% of the program time:

void fermiPlusCgemmLDS128_batched: 34.18%

void cudnn::detail::implicit_convolve_sgemm: 27.09%

void fft2d_c2r_32x32: 12.71%

Sgemm_sm35_ldg_tn_128x8x256x16x32: 8.25%

[CUDA memcpy HtoD]: 6.49%

void cudnn::detail::activation_fw_4d_kernel: 4.08%

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

cudaStreamCreateWithFlags: 37.37%

cudaFree: 28.82%

cudaMemGetInfo: 23.62%

cudaStreamSynchronize: 8.37%

Include an explanation of the difference between kernels and API calls

A kernel is a low level program interfacing with hardware. It is the lowest level program running on computers. An API is a generic term defining the interface developers have to use when writing code using libraries and a programming language. So when we are using API, it is the same as we call the code which are being wrote by other programmer.

Show output of rai running MXNet on the CPU

```
Loading fashion-mnist data...
done
Loading model...
done
New Inference
EvalMetric: {'accuracy': 0.8444}
12.62user 6.40system 0:08.46elapsed 224%CPU (0avgtext+0avgdata 2828716maxresident)k
0inputs+2624outputs (0major+39582minor)pagefaults 0swaps
```

List program run time

```
12.62user 6.40system 0:08.46elapsed 224%CPU (0avgtext+0avgdata 2828716maxresident)k
0inputs+2624outputs (0major+39582minor)pagefaults 0swaps
```

Show output of rai running MXNet on the GPU

```
Loading fashion-mnist data...
done
Loading model...
==342== NVPROF is profiling process 342, command: python m1.2.py
[18:17:30] src/operator/././cudnn_algoreg-inl.h:112: Running performance tests to find
the best convolution algorithm, this can take a while... (setting env variable
MXNET_CUDNN_AUTOTUNE_DEFAULT to 0 to disable)
done
New Inference
EvalMetric: {'accuracy': 0.8444}
==342== Profiling application: python m1.2.py
==342== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 33.90%  118.53ms         9  13.170ms  13.158ms  13.188ms  void
fermiPlusCgemmLDS128_batched<bool=0, bool=1, bool=0, bool=0, int=4, int=4, int=4,
int=3, int=3, bool=1, bool=1>(float2**, float2**, float2**, float2*, float2 const *,
float2 const *, int, int, int, int, int, int, __int64, __int64, __int64, float2 const
*, float2 const *, float2, float2, int)
 26.87%  93.944ms         1  93.944ms  93.944ms  93.944ms  void
cudnn::detail::implicit_convolve_sgemm<float, int=1024, int=5, int=5, int=3, int=3,
int=3, int=1, bool=1, bool=0, bool=1>(int, int, int, float const *, int,
cudnn::detail::implicit_convolve_sgemm<float, int=1024, int=5, int=5, int=3, int=3,
int=3, int=1, bool=1, bool=0, bool=1>*, float const *, kernel_conv_params, int, float,
float, int, float const *, float const *, int, int)
 12.63%  44.153ms         9  4.9059ms  2.7041ms  6.2815ms  void fft2d_c2r_32x32<float,
bool=0, unsigned int=0, bool=0, bool=0>(float*, float2 const *, int, int, int, int,
int, int, int, int, int, float, float, cudnn::reduced_divisor, bool, float*, float*)
  8.16%  28.540ms         1  28.540ms  28.540ms  28.540ms
sgemm_sm35_ldg_tn_128x8x256x16x32
  6.87%  24.035ms        14  1.7168ms  1.5360us  23.208ms  [CUDA memcpy HtoD]
4.06%  14.178ms         2  7.0892ms  252.57us  13.926ms  void
cudnn::detail::activation_fw_4d_kernel<float, float, int=128, int=1, int=4,
cudnn::detail::tanh_func<float>>(cudnnTensorStruct, float const *,
```

```
cudnn::detail::activation_fw_4d_kernel<float, float, int=128, int=1, int=4,
cudnn::detail::tanh_func<float>>, cudnnTensorStruct*, float, cudnnTensorStruct*, int,
cudnnTensorStruct*)
  3.80%  13.297ms         1  13.297ms  13.297ms  13.297ms  void
cudnn::detail::pooling_fw_4d_kernel<float, float,
cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>,
int=0>(cudnnTensorStruct, float const *, cudnn::detail::pooling_fw_4d_kernel<float,
float, cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>, int=0>,
cudnnTensorStruct*, cudnnPoolingStruct, float, cudnnPoolingStruct, int,
cudnn::reduced_divisor, float)
  1.71%  5.9897ms         9  665.52us  499.99us  886.25us  void fft2d_r2c_32x32<float,
unsigned int=0, bool=0>(float2*, float const *, int, int, int, int, int, int, int,
int, int, cudnn::reduced_divisor, bool)
  1.16%  4.0654ms         1  4.0654ms  4.0654ms  4.0654ms
sgemm_sm35_ldg_tn_64x16x128x8x32
  0.37%  1.2819ms         1  1.2819ms  1.2819ms  1.2819ms  void
mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>,
mshadow::expr::Plan<mshadow::expr::ScalarExp<float>, float>>(mshadow::gpu, unsigned
int, mshadow::Shape<int=2>, int=2, int)
0.32%  1.1108ms         1  1.1108ms  1.1108ms  1.1108ms  void
mshadow::cuda::SoftmaxKernel<int=8, float,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>>(mshadow::gpu,
int=2, unsigned int)
  0.05%  175.84us        13  13.525us  2.0480us  74.590us  void
mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>,
mshadow::expr::Plan<mshadow::expr::ScalarExp<float>, float>>(mshadow::gpu, unsigned
int, mshadow::Shape<int=2>, int=2)
0.04%  146.43us         2  73.214us  16.256us  130.17us  void
mshadow::cuda::MapPlanKernel<mshadow::sv::plusto, int=8,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>,
mshadow::expr::Plan<mshadow::expr::Broadcast1DExp<mshadow::Tensor<mshadow::gpu, int=1,
float>, float, int=2, int=1>, float>>(mshadow::gpu, unsigned int,
mshadow::Shape<int=2>, int=2)
  0.04%  130.46us         1  130.46us  130.46us  130.46us
sgemm_sm35_ldg_tn_32x16x64x8x16
  0.01%  22.495us         1  22.495us  22.495us  22.495us  void
mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>,
mshadow::expr::Plan<mshadow::expr::ReduceWithAxisExp<mshadow::red::maximum,
mshadow::Tensor<mshadow::gpu, int=3, float>, float, int=3, bool=1, int=2>,
float>>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=2)
  0.01%  21.344us         1  21.344us  21.344us  21.344us  void fft2d_r2c_32x32<float,
unsigned int=5, bool=1>(float2*, float const *, int, int, int, int, int, int, int,
```

```
int, int, cudnn::reduced_divisor, bool)
 0.00%  9.9510us         1  9.9510us  9.9510us  9.9510us  [CUDA memcpy DtoH]
==342== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 37.40%  1.45873s        18  81.041ms  22.761us  729.21ms  cudaStreamCreateWithFlags
 29.25%  1.14096s        10  114.10ms  1.2160us  302.81ms  cudaFree
 23.47%  915.65ms        27  33.913ms  259.70us  907.34ms  cudaMemGetInfo
  8.30%  323.71ms        29  11.162ms  6.5270us  194.11ms  cudaStreamSynchronize
```

```
  1.24%  48.507ms         9  5.3897ms  8.6400us  23.394ms  cudaMemcpy2DAsync
  0.18%  7.0216ms        45  156.04us  9.0380us  966.62us  cudaMalloc
  0.04%  1.3659ms         4  341.46us  336.29us  356.33us  cuDeviceTotalMem
  0.03%  1.2067ms       352  3.4280us     516ns  113.28us  cuDeviceGetAttribute
  0.03%  1.0783ms       114  9.4580us  1.2580us  406.41us  cudaEventCreateWithFlags
  0.02%  788.60us        53  14.879us  6.4670us  95.007us  cudaLaunch
  0.01%  364.98us         6  60.829us  27.318us  119.28us  cudaMemcpy
  0.01%  356.02us       619     575ns     524ns  1.5840us  cudaSetupArgument
  0.01%  289.79us         4  72.447us  52.258us  92.095us  cudaStreamCreate
  0.00%  124.11us       116  1.0690us     680ns  2.5100us  cudaDeviceGetAttribute
  0.00%  104.53us         4  26.133us  19.246us  29.666us  cuDeviceGetName
  0.00%  93.609us        36  2.6000us     826ns  8.4420us  cudaSetDevice
  0.00%  56.850us        27  2.1050us  1.7510us  3.6070us  cudaStreamWaitEvent
  0.00%  51.082us        53     963ns     533ns  2.7090us  cudaConfigureCall
  0.00%  47.488us         2  23.744us  23.146us  24.342us
cudaStreamCreateWithPriority
  0.00%  26.169us        10  2.6160us  1.5350us  8.3280us  cudaGetDevice
  0.00%  23.543us        12  1.9610us  1.1230us  5.2820us  cudaEventRecord
  0.00%  23.013us         1  23.013us  23.013us  23.013us  cudaBindTexture
  0.00%  21.063us        34     619ns     537ns  1.0390us  cudaGetLastError
  0.00%  13.161us        18     731ns     638ns  1.0900us  cudaPeekAtLastError
  0.00%  7.1140us         6  1.1850us     572ns  2.3200us  cuDeviceGetCount
  0.00%  6.5660us         6  1.0940us     645ns  1.6480us  cuDeviceGet
0.00%  6.5200us         1  6.5200us  6.5200us  6.5200us  cudaStreamGetPriority
  0.00%  6.3120us         1  6.3120us  6.3120us  6.3120us  cudaEventCreate
  0.00%  5.0090us         2  2.5040us  2.0170us  2.9920us
cudaDeviceGetStreamPriorityRange
  0.00%  3.2640us         3  1.0880us     972ns  1.1480us  cuInit
  0.00%  2.8040us         3     934ns     889ns  1.0150us  cuDriverGetVersion
  0.00%  2.5250us         1  2.5250us  2.5250us  2.5250us  cudaEventDestroy
  0.00%  1.8790us         1  1.8790us  1.8790us  1.8790us  cudaUnbindTexture
  0.00%  1.3280us         1  1.3280us  1.3280us  1.3280us  cudaGetDeviceCount
```

List program run time

```
2.31user 1.07system 0:02.84
```

Milestone 2:

```
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 6.585881
Op Time: 19.545969
Correctness: 0.8451 Model: ece408
30.66user 1.58system 0:30.12elapsed 107%CPU (0avgtext+0avgdata 2819428maxresiden
t)k
```

Total Execution Time:

30.66+1.58+30.12 = 62.36s

Operation Times:

First layer Op Time: 6.585881s

Second layer Op Time: 19.545969s

Milestone 3:

In our parallel implemented forward_kernel, its duration is about 705 microseconds, while in the sequential kernel in milestone 2 it costs about 20 seconds. And at the same time, they have the exact same accuracy and same functionality.

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

Duration	704.981 µs

We didn't use shared memory in our forward kernel, so the optimization effect of shared memory doesn't appear in our case.

Shared Memory/Block	0 B

The following is the data for the kernel function we used:

```
Cuda Fuctions :
mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int, int)
Maximum instruction execution count in assembly: 249600
Average instruction execution count in assembly: 72873
Instructions executed for the kernel: 9182080
Thread instructions executed for the kernel: 239962880
Non-predicated thread instructions executed for the kernel: 224787840
Warp non-predicated execution efficiency of the kernel: 76.5%
Warp execution efficiency of the kernel: 81.7%
```

From this we observed that the kernel's warp execution efficiency of 76.5% is less than 100% due to divergent branches and predicated instructions. If predicated instructions are not taken into account the warp execution efficiency for these kernels is 81.7%. This could lead to lower utilization of the GPU's compute resources. The way to optimize this is through reducing the amount of intra-warp divergence and predication in the kernel.

Another thing is the divergent branches. If there exists a high divergence in some warps, the total efficiency of execution would decrease dramatically since divergence means waste of execution resource in each warps. So if we could rearrange each warps and threads to make consecutive threads executing or not executing at the same time, the divergence would be reduced which means that higher optimization in parallel computing kernels.

Memory bandwidth is another important factor that restricts the performances of parallel kernel. But since we didn’t use shared memory here, we didn’t have the best optimized performance due to the restriction in speed of global memory reads and global memory writes.

Transactions	Bandwidth	Utilization	
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Global Loads	2062544	166.377 GB/s	
Global Stores	8840	885.478 MB/s	
Atomic	0	0 B/s	
L1/Shared Total	2071384	167.262 GB/s	Idle Low Medium High Max

Milestone 4:

Our first optimization is the Unroll / shared-memory Matrix multiply. In this function we have two kernels. The first is the unroll function to expand the input feature map into a unrolled version. And then the next step is to use the matrix multiplication function to multiply the X_unrolled with the w input. Shared memory is used in the matrix multiplication.

Also when running dataset of 10, it showed worse performance than the restricted and normal ones.

Here is the timeline of the unroll method.

I first examined the matrix multiplication kernel.

Duration	31.776 µs
Grid Size	[43,1,1]
Block Size	[16,16,1]
Registers/Thread	25
Shared Memory/Block	2 KiB
Shared Memory Requested	112 KiB
Shared Memory Executed	112 KiB
Shared Memory Bank Size	4 B

According to the analysis, our grid size is too small to hide compute and memory latency.The kernel does not execute enough blocks to hide memory and operation latency. Typically the kernel grid size must be large enough to fill the GPU with multiple "waves" of blocks. Since the grid size is determined by the height and width of output and number of maps. This could be much more efficient when dealing with big datasets.

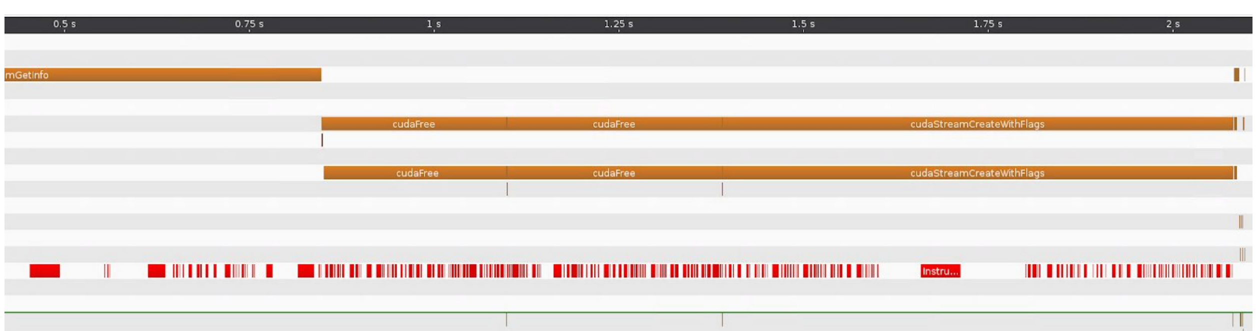
With the use of shared memory, we do not need to perform global read for this kernel. The shared loads and stores are used efficiently.

Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	82560	577.769 GB/s	
Shared Stores	6880	48.147 GB/s	
Global Loads	18477	29.707 GB/s	
Global Stores	940	1.484 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	108857	657.107 GB/s	Idle Low Medium High Max

The next is the unroll kernel. This changed the X input matrix into an unroll matrix. Here is the major parameters.

Duration	19.488 µs
Grid Size	[4,1,1]
Block Size	[1024,1,1]
Registers/Thread	23
Shared Memory/Block	0 B
Shared Memory Requested	112 KiB
Shared Memory Executed	112 KiB
Shared Memory Bank Size	4 B

Our other optimization is using the constant memory and restricted. We use cudaMemcpytoSymbol to copy W matrix into a constant memory. The total execution time is 2s this time.



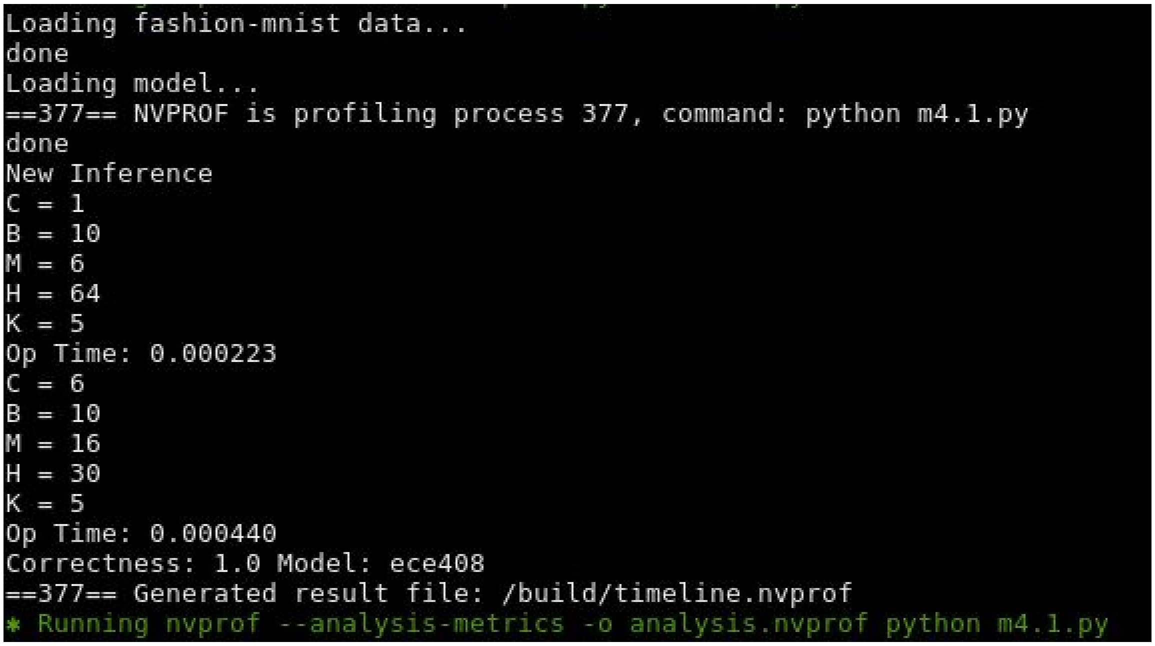
Here is the spec for optimized forward kernel.

Duration	359.484 µs
Grid Size	[10,16,4]
Block Size	[16,16,1]
Registers/Thread	30
Shared Memory/Block	0 B
Shared Memory Requested	112 KiB
Shared Memory Executed	112 KiB

Since we used constant memory in this scenario, the kernel only performs global stores. So it saved time for global reads.

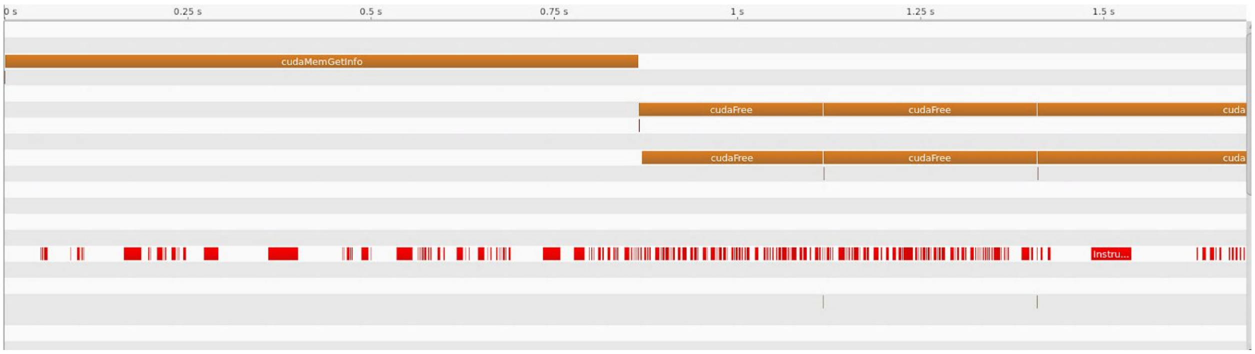
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Global Loads	0	0 B/s	
Global Stores	8840	1.744 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	8840	1.744 GB/s	Idle Low Medium High Max

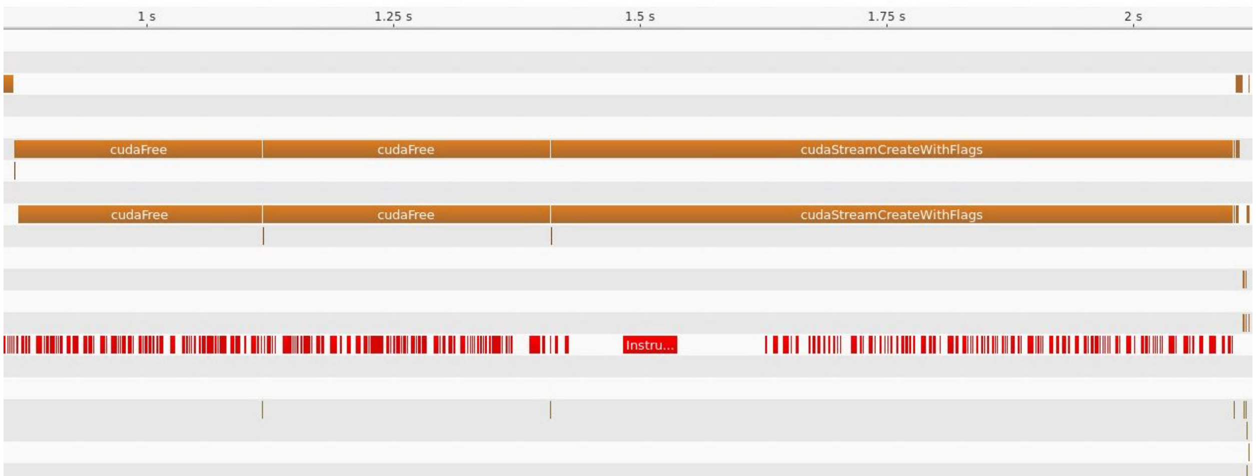
The third optimization we used is Tuning with restrict, loop unrolling. In fact, we combine this optimization with the constant memory optimization together. First we add the __restricted__ signal before three input pointers of matrix. So that the compiler would know that writes and reads to these three pointers would not affect the result of another pointers' value. So that the kernel could be accelerated. Also, we use #pragma unroll just before the calculation for loop in the forward_layer kernel. Thus the compile would unroll all the threads to different threads which would accelerate the operation time. We could know from the picture that the op time is shortened compared to the original non-optimized kernel.



```
Loading fashion-mnist data...
done
Loading model...
==377== NVPROF is profiling process 377, command: python m4.1.py
done
New Inference
C = 1
B = 10
M = 6
H = 64
K = 5
Op Time: 0.000223
C = 6
B = 10
M = 16
H = 30
K = 5
Op Time: 0.000440
Correctness: 1.0 Model: ece408
==377== Generated result file: /build/timeline.nvprof
* Running nvprof --analysis-metrics -o analysis.nvprof python m4.1.py
```

The two following figures are the screen shot from the NVVP, which is the timeline of running the program by this optimized kernel.


0 s
0.25 s
0.5 s
0.75 s
1 s
1.25 s
1.5 s
cudaMemGetInfo
cudaFree
cudaFree
cudaFree
cudaFree
Instru...


1 s
1.25 s
1.5 s
1.75 s
2 s
cudaFree
cudaFree
cudaStreamCreateWithFlags
cudaFree
cudaFree
cudaStreamCreateWithFlags
Instru...