Optimization for 2D/3D-Convolution Algorithms in CUDA

Burak Topçu
January, 2022

Overview

- **Problem Definition**
- Proposed Solution/Implementation Details
  - Constant Memory Implementation
  - Shared Memory Implementation
  - Stream Implementation
- Experiment Results, Comparisons and Comments
  - Experiment Setup
  - Experimental Results, Comparisons and Comments
- Future Work

Problem Definition

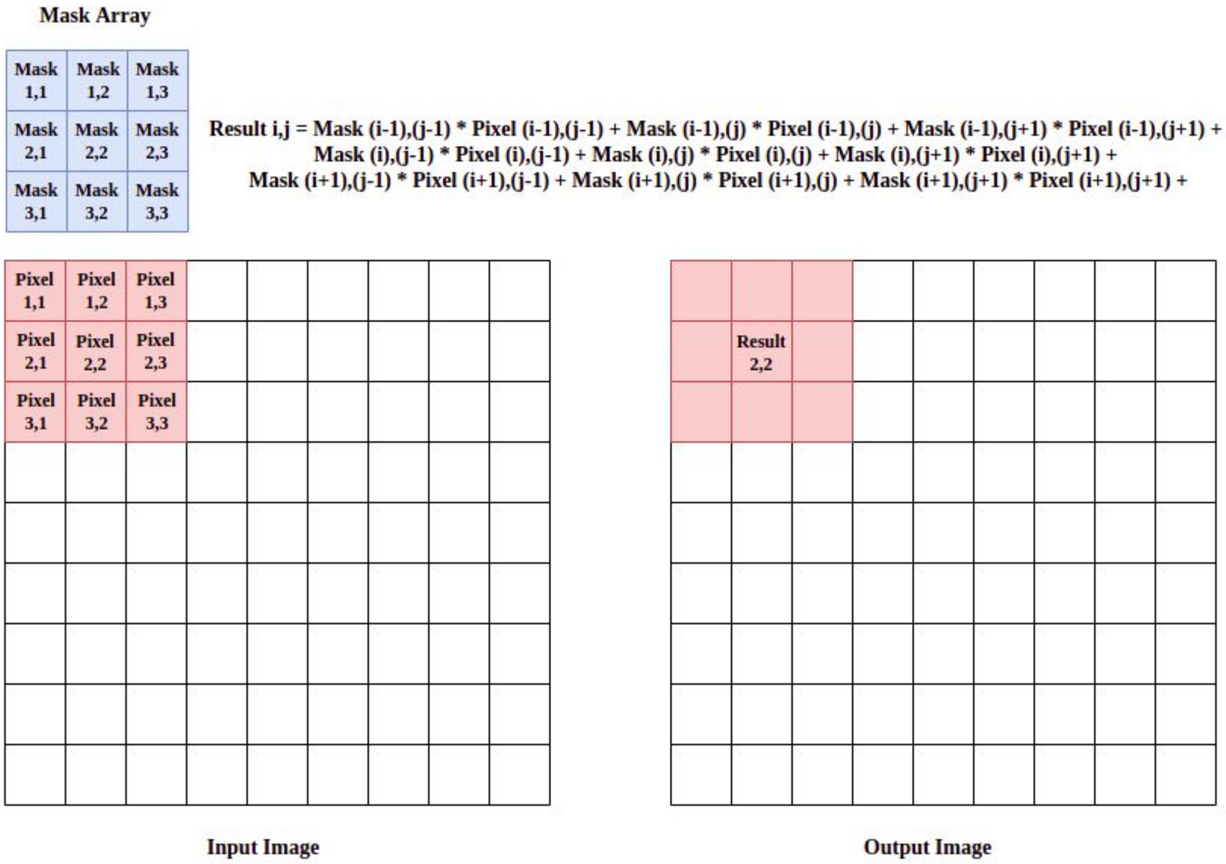


Figure 1: 2DConvolution implementation model

Problem Definition

```
__global__ void naive_kernel(DATA_TYPE *A, DATA_TYPE *B, const int problem_size)
{
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;

    DATA_TYPE c11, c12, c13, c21, c22, c23, c31, c32, c33;

    c11 = +0.2;  c21 = +0.5;  c31 = -0.8;
    c12 = -0.3;  c22 = +0.6;  c32 = -0.9;
    c13 = +0.4;  c23 = +0.7;  c33 = +0.10;

    if ((i < problem_size-1) && (j < problem_size-1) && (i > 0) && (j > 0))
    {
        B[i * problem_size + j] =  c11 * A[(i - 1) * problem_size + (j - 1)] +
                                   c21 * A[(i - 1) * problem_size + (j + 0)] +
                                   c31 * A[(i - 1) * problem_size + (j + 1)] +
                                   c12 * A[(i + 0) * problem_size + (j - 1)] +
                                   c22 * A[(i + 0) * problem_size + (j + 0)] +
                                   c32 * A[(i + 0) * problem_size + (j + 1)] +
                                   c13 * A[(i + 1) * problem_size + (j - 1)] +
                                   c23 * A[(i + 1) * problem_size + (j + 0)] +
                                   c33 * A[(i + 1) * problem_size + (j + 1)];
    }
}
```

Figure 2: 2DConvolution GPU kernel

Problems:

- global memory accesses
- wasting register resource

Problem Definition

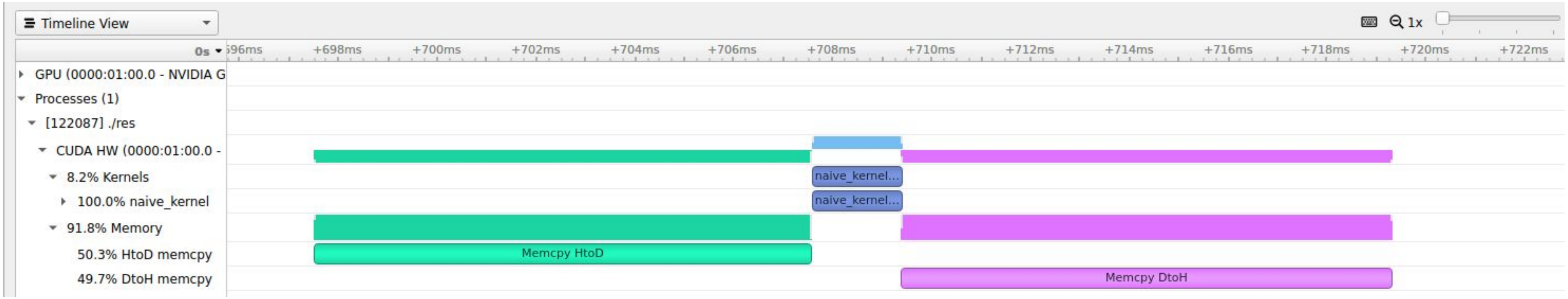


Figure 3: Nsight System profiling for 2DConvolution implemented naively

- Memory copy operations takes lots of time in overall.

Overview

- Problem Definition
- **Proposed Solution/Implementation Details**
  - **Constant Memory Implementation**
  - **Shared Memory Implementation**
  - **Stream Implementation**
- Experiment Results, Comparisons and Comments
  - Experiment Setup
  - Experimental Results, Comparisons and Comments
- Future Work

Proposed Solutions

1) Implementing shared memory utilization for naive GPU kernel of both 2D/3DConvolution.
2) Implementing constant memory utilization for the mask array of the GPU kernels.
3) Implementing a version which works with streams to increase overall performance.

Constant Memory Usage on Convolution Algorithms

```
__constant__ DATA_TYPE c[3][3] = {{0.2, -0.3, 0.4}, {0.5, 0.6, 0.7}, {-0.8, -0.9, 0.1}};

__global__ void const_mem_kernel(DATA_TYPE *A, DATA_TYPE *B, const int problem_size)
{
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;

    if ((i < problem_size-1) && (j < problem_size-1) && (i > 0) && (j > 0))
    {
        B[i * problem_size + j] =  c[0][0] * A[(i - 1) * problem_size + (j - 1)] +
                                   c[1][0] * A[(i - 1) * problem_size + (j + 0)] +
                                   c[2][0] * A[(i - 1) * problem_size + (j + 1)] +
                                   c[0][1] * A[(i + 0) * problem_size + (j - 1)] +
                                   c[1][1] * A[(i + 0) * problem_size + (j + 0)] +
                                   c[2][1] * A[(i + 0) * problem_size + (j + 1)] +
                                   c[0][2] * A[(i + 1) * problem_size + (j - 1)] +
                                   c[1][2] * A[(i + 1) * problem_size + (j + 0)] +
                                   c[2][2] * A[(i + 1) * problem_size + (j + 1)];
    }
}
```

Instead of holding the mask array into registers, I put them into the constant memory.

Figure 4: Constant memory usage for masking array for 2DConvolution.

Shared Memory Usage on Convolution Algorithms

For each thread block where thread block size is 32*32, 34*34 bytes shared memory is allocated.

In each if block a bound is checked to increase accuracy level.

```
__shared__ float shmem[34][34];

short int gl_ty = blockIdx.x * blockDim.x + threadIdx.x;
short int gl_tx = blockIdx.y * blockDim.y + threadIdx.y;
short int lcl_ty = threadIdx.x;
short int lcl_tx = threadIdx.y;

if ((gl_ty > 0) && (gl_tx > 0) && (gl_tx < mat_dim -1) && (gl_ty < mat_dim -1))
{
    shmem[lcl_tx + 1][lcl_ty + 1] = A[gl_tx * mat_dim + gl_ty];

    if(lcl_ty == 0)
        shmem[lcl_tx + 1][0] = A[gl_tx * mat_dim + gl_ty - 1];

    if(lcl_tx == 0)
        shmem[0][lcl_ty + 1] = A[(gl_tx - 1) * mat_dim + gl_ty];

    if(lcl_ty == (b_dim - 1))
        shmem[lcl_tx + 1][b_dim + 1] = A[gl_tx * mat_dim + gl_ty + 1];

    if(lcl_tx == (b_dim - 1))
        shmem[b_dim + 1][lcl_ty + 1] = A[(gl_tx + 1) * mat_dim + gl_ty];
    __syncthreads();

    B[gl_tx * mat_dim + gl_ty] = c11 * shmem[lcl_tx][lcl_ty] +
                                 c21 * shmem[lcl_tx][lcl_ty + 1] +
                                 c31 * shmem[lcl_tx][lcl_ty + 2] +
                                 c12 * shmem[lcl_tx + 1][lcl_ty ] +
                                 c22 * shmem[lcl_tx + 1][lcl_ty + 1] +
                                 c32 * shmem[lcl_tx + 1][lcl_ty + 2] +
                                 c13 * shmem[lcl_tx + 2][lcl_ty ] +
                                 c23 * shmem[lcl_tx + 2][lcl_ty + 1] +
                                 c33 * shmem[lcl_tx + 2][lcl_ty + 2];
}
```

Figure 5: Shared memory usage on 2DConvolution GPU kernel.

Stream Usage on Convolution Algorithms

For stream usage, the naive kernel is kept as the same.

However, kernel is called with the stream amount. Also, memory copy operations from DtoH and HtoD are carried out asynchronously.

```
for(int i = 0; i < num_streams; i++)
{
    cudaStreamCreate(&streams[i]);
}

for(int stream = 0; stream < num_streams; stream++)
{
    const int lower = chunk_size * stream;
    const int upper = min(lower + chunk_size, mat_dim * mat_dim);
    const int width = upper - lower;

    cudaEvent_t start, stop;
    float x = 0;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaMemcpyAsync(A_gpu + lower, A + lower, sizeof(DATA_TYPE) * width, cudaMemcpyHostToDevice, streams[stream]);
    cudaEventRecord(start);
    stream_version_kernel<<<blocksPerGrid, threadsPerBlock, 0, streams[stream]>>>(A_gpu + lower,
                                                                                B_gpu + lower, mat_dim, num_streams);
    cudaEventRecord(stop);
    cudaMemcpyAsync(B_dev + lower, B_gpu + lower, sizeof(DATA_TYPE) * width, cudaMemcpyDeviceToHost, streams[stream]);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&x, start, stop);

    milliseconds += x;

    if(stream)
        cudaStreamSynchronize(streams[stream-1]);
}
for(int i = 0; i < num_streams; i++)
{
    cudaStreamDestroy(streams[i]);
}
```

Figure 6: Stream usage implementation for 2DConvolution.

Overview

- Problem Definition
- Proposed Solution/Implementation Details
  - Constant Memory Implementation
  - Shared Memory Implementation
  - Stream Implementation
- **Experiment Results, Comparisons and Comments**
  - **Experiment Setup**
  - **Experimental Results, Comparisons and Comments**
- Future Work

Experiment Setup

In this work;

I used two different environment.

- GeForce GTX 1650 GPU with CUDA 11.4
- Tesla K80 provided by Google Colab with CUDA 9.1

I completed 7 and 6 different experiments for 2D/3D Convolution algorithms respectively.

- For **2DConvolution:**
  - Problem size : 4096*4096, 8192*8192 and 16384*16384
  - Thread Blocks : 8*8 (64), 16*16 (256) and 32*32 (1024)
  - Number of stream : 8, 16 and 32
- For **3DConvolution**:
  - Problem size : 128*128*128, 256*256*256 and 512*512*512
  - Thread Blocks : 8*8 (64), 16*16 (256) and 32*32 (1024)
  - Number of stream : 4, 8 and 16

Experiment Results (Shared Memory)

- problem size = 16384*16384,
- Thread Block size = 32*32
- # of streams = 8

on Tesla K80 GPU.

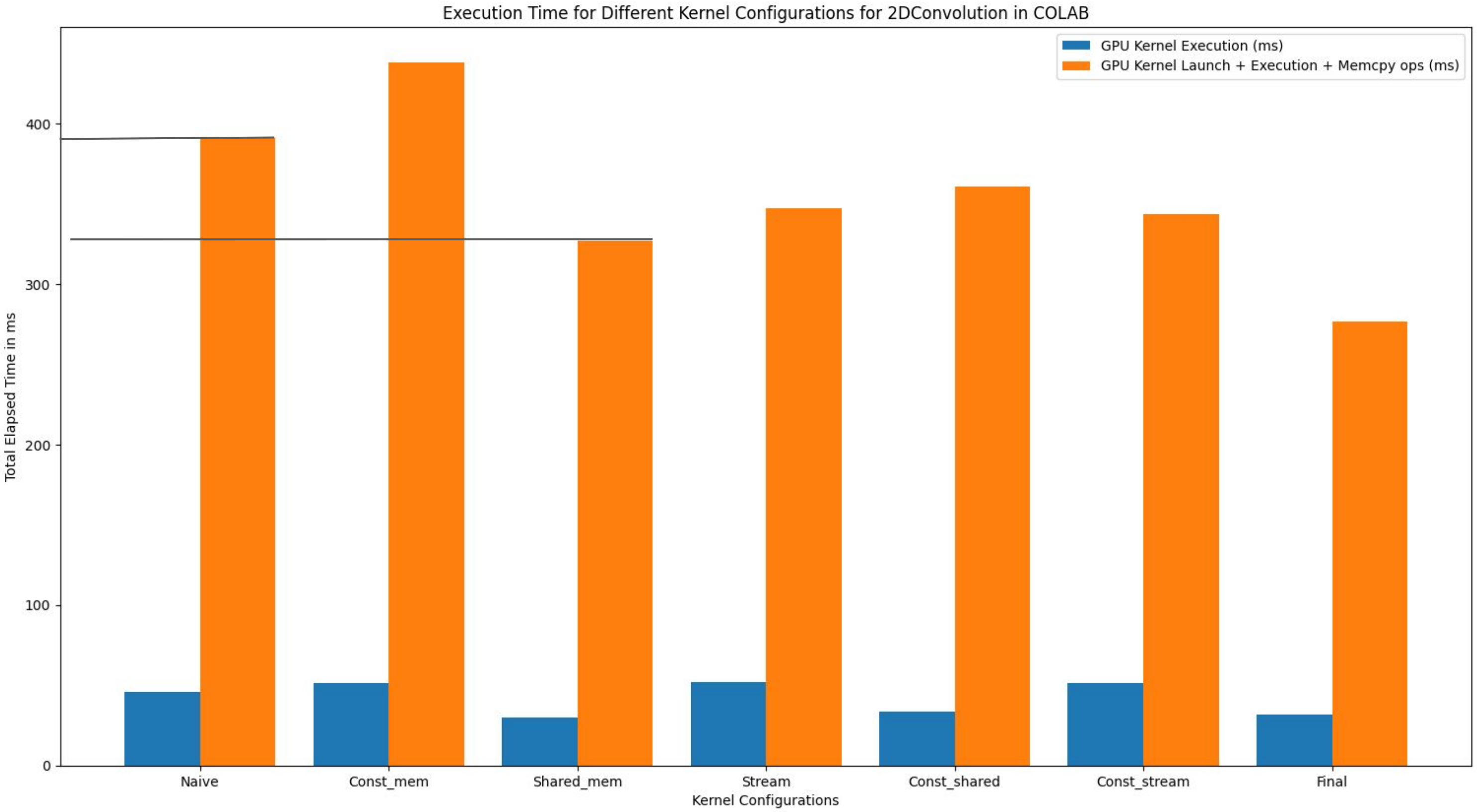
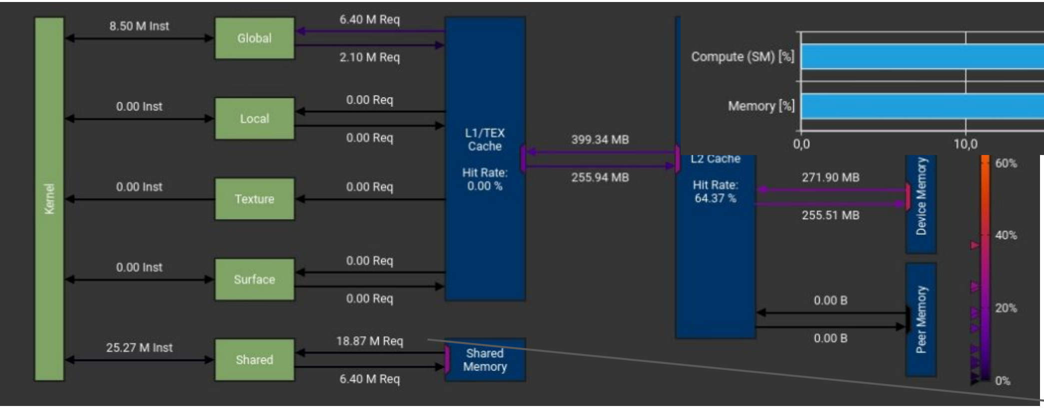
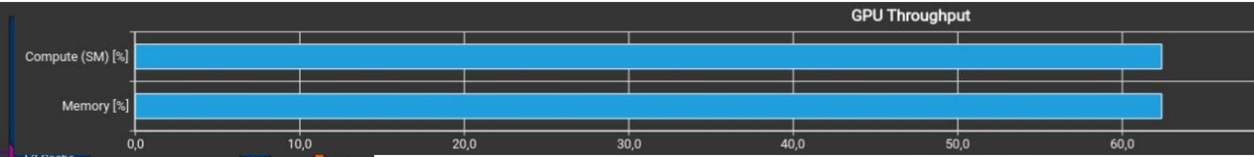


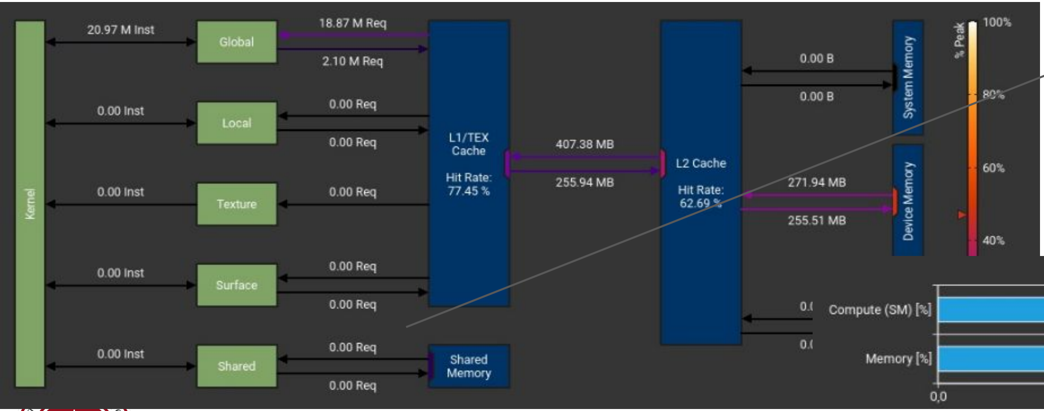
Figure 7: Performance results for 2DConvolution with different configuration parameters.

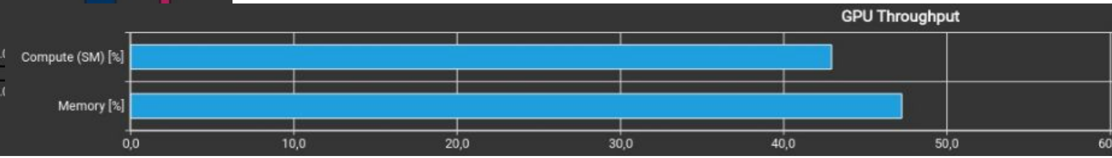
Experiment Results (shared memory)





- Shared memory is utilized.
- Traffic between L1 cache and L2 cache is decreased.
- Also traffic between global memory and L1 is significantly decreased.





Experiment Results (Stream Usage)

- problem size = 16384*16384,
- Thread Block size = 32*32
- # of streams = 8

on Tesla K80 GPU.

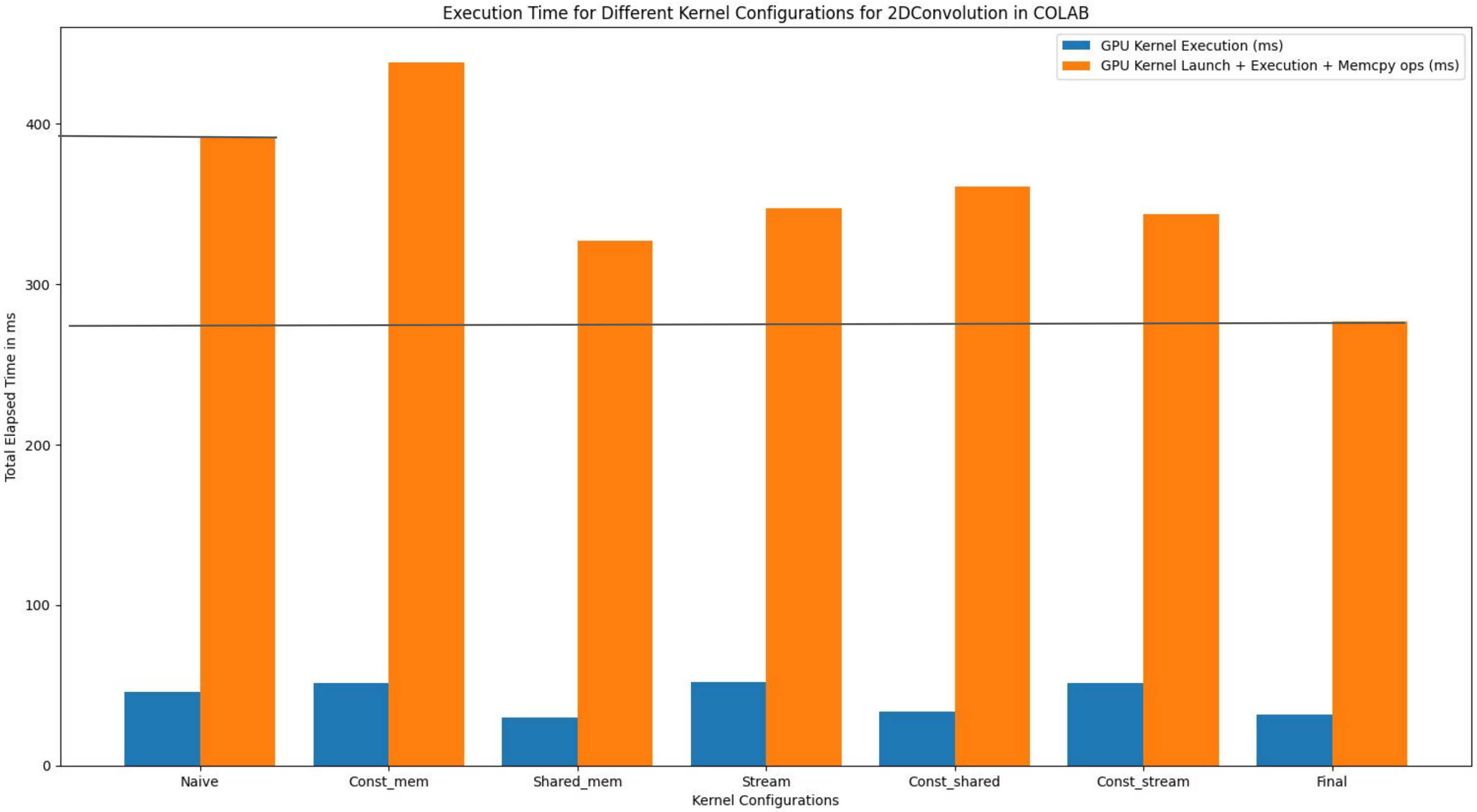


Figure 8: Performance results for 2DConvolution with different configuration parameters.

Experiment Results (Stream Usage)

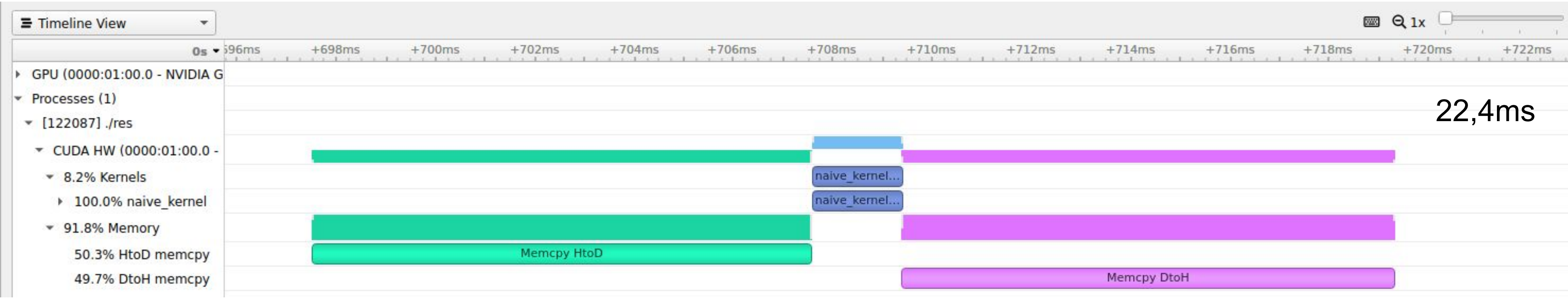


Figure 9: Nsight System profiling for 2DConvolution with naive implementation.

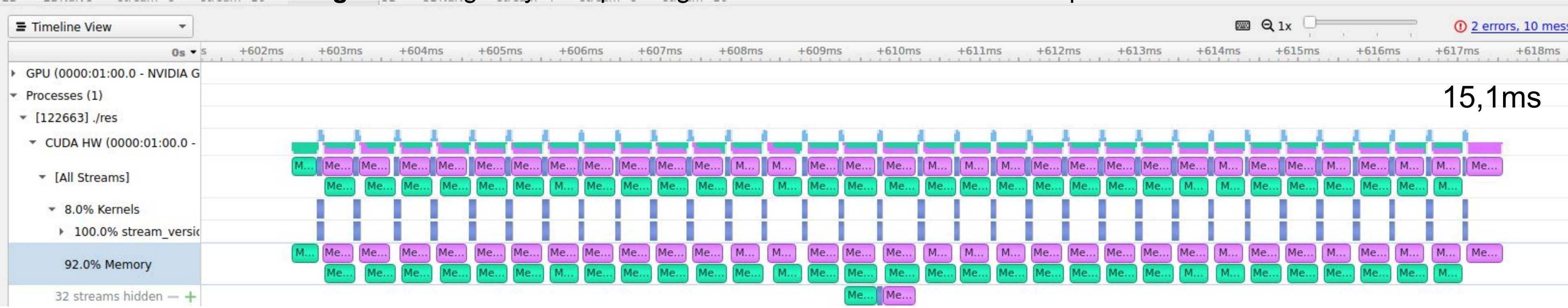


Figure 10: Nsight System profiling for 2DConvolution with 32 streams.

Experiment Results (Constant memory)

- problem size = 16384*16384,
- Thread Block size = 32*32
- # of streams = 8

on Tesla K80 GPU.

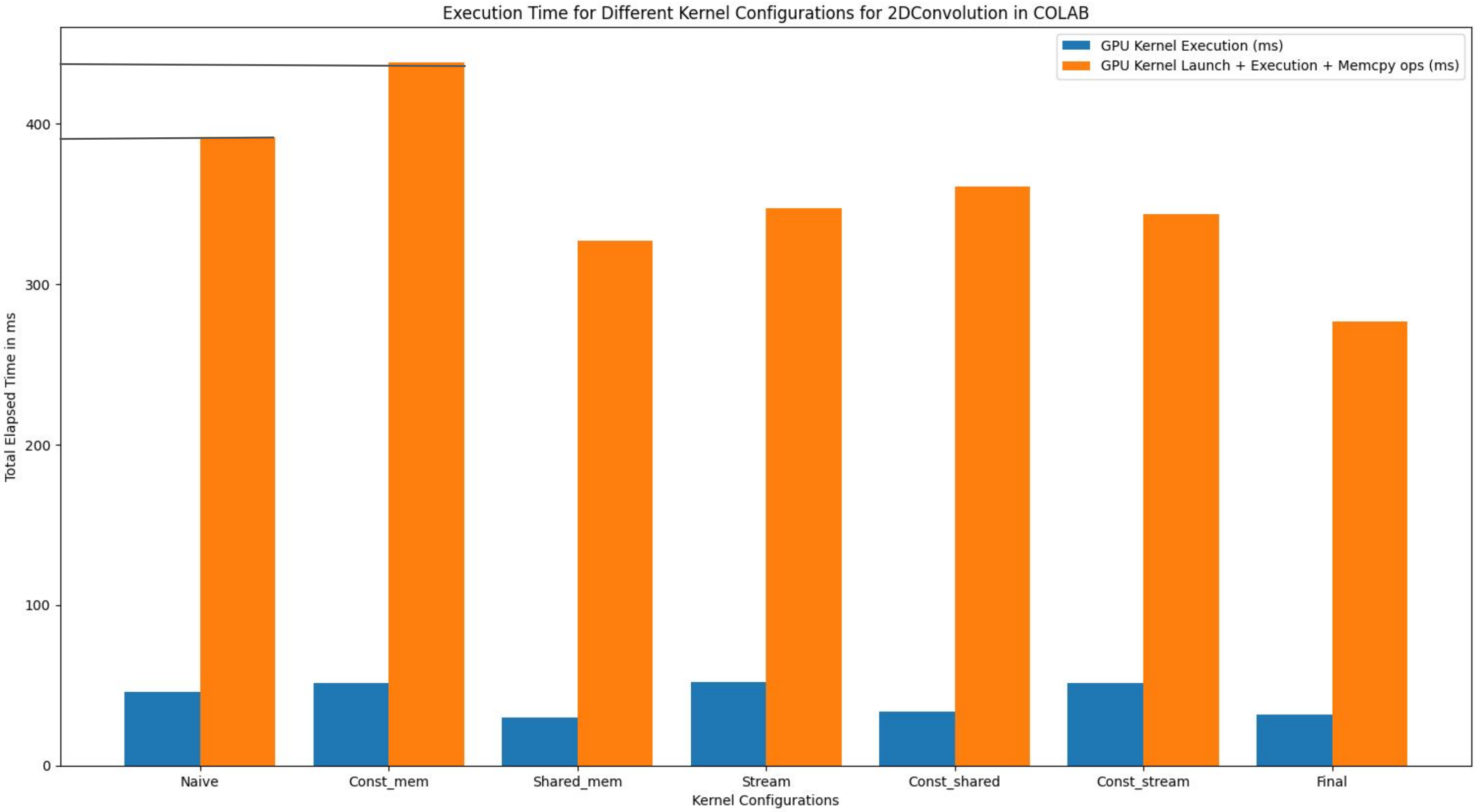


Figure 11: Performance results for 2DConvolution with different configuration parameters.

Experiment Results (Constant Memory)

- problem size = 16384*16384,
- Thread Block size = 32*32
- # of streams = 8

on GeForce GTX 1650 GPU.

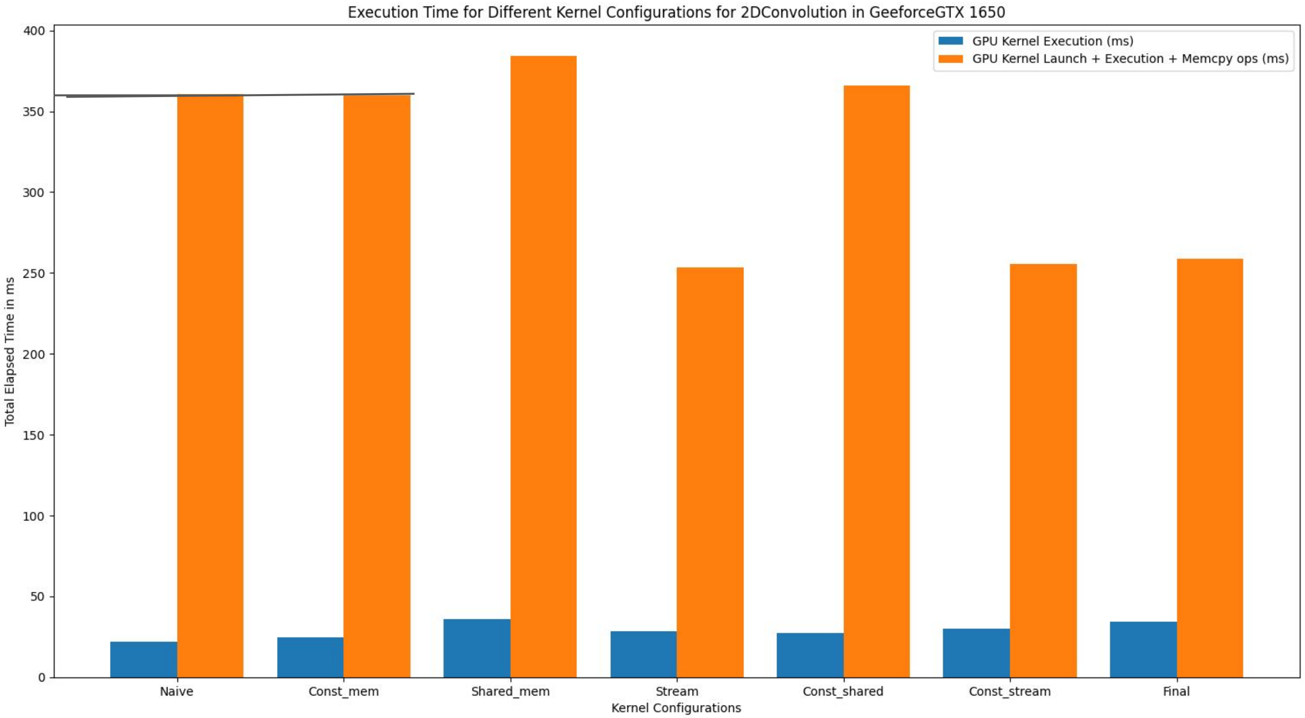


Figure 7: Performance results for 2DConvolution with different configuration parameters.

Future Work

- For future work, the problems that arise after the implementation of the mentioned optimizations can be examined.

1) The decision of the most efficient stream amount depends on the data and belonging architecture resources in terms of peripheral between host and GPU device.
2) The convolution can operate on bigger arrays such as 7*7 or 8*8 instead of 3*3. This will increase the tile size of shared memory. For such a scenario, the user needs to re-implement shared memory usage with respect to the convolution algorithm features and GPU shared memory resource. This also depends on the GPU architecture resources and the data which will be processed.

Similar problems observed after the optimization can be the future work for optimizing 2D/3D Convolution more.

GitHub link

Final report

Thank you for your participation and attention.