Programming Assignment 4 - Optimized Convolutions

- Shantanu Mukhopadhyay[200541793]

Implementation of Kernels: conv2dV1 (without Shared memory optimization)

During Assignment 4, I had to change the kernel to accommodate batch of input images and multiple filters of size 3x3. This approach is using global memory without using shared memory optimizations

Input and Output Dimensions

- **Input:** N images, each of size H × W. These are given by the first 3 lines of the input.txt files
- **Filter:** K filters, each of size 3 × 3, these are given by the first 2 lines of the filter.txt files.
- **Output:** Each input image is convolved with each filter, producing N × K output feature maps of size (H−2) × (W−2) after valid padding – since the image passed to the kernel is padded image of paddedheight= H+2 (ie. height + 2* padding of 1) and paddedwidth= W+2 (i.e width + 2* padding of 1).

Add **1-pixel zero-padding** on all sides of the input images to ensure that the convolution results are not truncated at the borders.

CUDA Kernel Design: conv2d_kernel

Thread Mapping

```
int w = threadIdx.x + blockIdx.x * blockDim.x;
int h = threadIdx.y + blockIdx.y * blockDim.y;
int k = blockIdx.z % K;
int n = blockIdx.z / K;
```

- Each thread computes **one output element** at (n, k, h, w) where:
    - int w = threadIdx.x + blockIdx.x * blockDim.x;

    ➔ threadIdx.x: x-coordinate of the thread within its block.
    ➔ blockIdx.x: x-coordinate of the block in the grid.
    ➔ blockDim.x: number of threads per block in the x-direction.
    ➔ this gives the **global x-index** (i.e., column) for the thread.

    - int h = threadIdx.y + blockIdx.y * blockDim.y;k = filter index

    ➔ It gives the **global y-index** (i.e., row) for the thread

- int k = blockIdx.z % K;

  ➔ K: the number of filters (output channels),
  ➔ N: the number of input images (batch size).
  ➔ So blockIdx.z ranges from 0 to N * K - 1.

To extract which filter (i.e., output channel) this thread should work on, used:

  ➔ k = blockIdx.z % K: This gives the filter index.

- Int n = blockIdx.z /K;

  ➔ This gives the index of images (since N*K was passed from Host code)

Each CUDA block is BLOCK_SIZE × BLOCK_SIZE (16*16 threads, and each thread is assigned a unique (h, w) coordinate in the output feature map.

Convolution Computation

```
    if (n < N && k < K && h < Hout && w < Wout) {
        float sum = 0.0f; // setting the value to 0 for sum
        for (int i = -1; i <= 1; i++) {
            for (int j = -1; j <= 1; j++) {
                sum += input[n * H * W + (h + 1 + i) * W + (w + 1 + j)] *
                       filter[k * 3 * 3 + (i + 1) * 3 + (j + 1)];
            }
        }
        int out_index = n * K * Hout * Wout + k * Hout * Wout + h * Wout + w;
        output[out_index] = sum;
```

For each output pixel (h, w) in the (n, k) output:

- The corresponding 3×3 patch is extracted from the input image n.
- It is element-wise multiplied with the filter k, and the results are accumulated into sum.
- The final result is written to output[n * K * Hout * Wout + k * Hout * Wout + h * Wout + w].

Memory Management

- Input images are **manually padded** in the host code before copying to device.
- Device memory is allocated for:
  - d_input (padded input of size N × (H+2) × (W+2))
  - d_filter (filters of size K × 3 × 3)
  - d_output (results of size N × K × (H−2) × (W−2))

Memory transfers are performed using cudaMemcpy before and after kernel execution.

Implementation of Kernels: conv2dV2 (with Shared memory optimization)

Similar to conv2dV1, the values remains the same for conv2dV2

Kernel Implementation Details

The kernel conv2d_kernel is launched with a 3D grid configuration:

int tx = threadIdx.x;

int ty = threadIdx.y;

- These retrieve the thread's local **x** and **y** indices **within a thread block**.
- tx and ty are used for indexing inside shared memory and for assigning output coordinates.

int w_out = tx + blockIdx.x * blockDim.x;

int h_out = ty + blockIdx.y * blockDim.y;

- These compute the **global output coordinates** (w_out, h_out) that this thread is responsible for computing.
- Each block handles a tile in the output grid, and each thread in a block computes one output pixel. So w_out and h_out correspond to the **(x, y)** position in the output feature map

int n = blockIdx.z / K;

int k = blockIdx.z % K;

- This handles the **3D grid launch**, where blockIdx.z encodes both:
    - n: Index of the input image in the batch (from N total images),
    - k: Index of the filter to apply (from K total filters).

int shared_W = BLOCK_SIZE + 2;

- This calculates the width of the **shared memory tile**.
- Since a 3x3 filter needs a 1-pixel border on all sides, each tile must be 2 pixels wider and taller than the actual output tile it computes (to get necessary neighbors).
- BLOCK_SIZE is the number of output pixels per dimension per block, so:

Each block processes a **BLOCK_SIZE × BLOCK_SIZE** tile of the output. However, since a 3×3 filter is used, each block must access a **(BLOCK_SIZE + 2) × (BLOCK_SIZE + 2)** tile of the input to perform valid convolution on the borders.

Use of Shared Memory

Shared memory is used to reduce the number of global memory accesses by collaboratively loading a tile of the input into on-chip memory. Here's how it works:

Allocation

```
extern __shared__ float tile[];
```

A dynamically allocated shared memory array named tile is declared with size (BLOCK_SIZE + 2) × (BLOCK_SIZE + 2).Since now I have not passed pre padded images.

Filling Shared Memory

Each thread participates in loading a portion of the input tile into shared memory:

```
// Load input tile into shared memory
for (int i = ty; i < BLOCK_SIZE + 2; i += blockDim.y) {
    for (int j = tx; j < BLOCK_SIZE + 2; j += blockDim.x) {
        int row = blockIdx.y * BLOCK_SIZE + i;
        int col = blockIdx.x * BLOCK_SIZE + j;

        if (row < H && col < W) {
            tile[i * shared_W + j] = input[n * H * W + row * W + col];
        } else {
            tile[i * shared_W + j] = 0.0f;
        }
    }
}
```

Threads compute their positions within the padded input and load the corresponding elements. Out-of-bounds accesses are safely handled using zero-padding.

After loading, I use:

__syncthreads();

to ensure all threads have completed the shared memory fill before any thread begins computation.

Convolution Computation

Once the shared tile is ready, each thread computes a single output value:

```
if (h_out < Hout && w_out < Wout && n < N && k < K) {
    float sum = 0.0f;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            sum += tile[(ty + i) * shared_W + (tx + j)] *
                   filter[k * 9 + i * 3 + j];
        }
    }

    int out_index = n * K * Hout * Wout + k * Hout * Wout + h_out * Wout + w_out;
    output[out_index] = sum;
}
```

This 3×3 region in shared memory corresponds to the receptive field for the output pixel at (h_out, w_out).

Finally, the computed result is written to global memory:

int out_index = n * K * Hout * Wout + k * Hout * Wout + h_out * Wout + w_out;

output[out_index] = sum;

Summary

- Shared memory was used to cache local tiles of the input data, significantly reducing redundant global memory access.
- Threads collaborated to load a larger region (BLOCK_SIZE+2 × BLOCK_SIZE+2) to account for the 3×3 filter's overlap.
- The result is a highly parallel, memory-efficient 2D convolution implementation that processes multiple images and filters simultaneously.

RESULTS:

Conv2dV1 vs Conv2dV2 on GPGPU SIM with **SM7_QV100** configuration

Conv2dV1

```
gpu_sim_cycle = 6624
gpu_sim_insn = 307200
gpu_ipc =       46.3768
gpu_tot_sim_cycle = 6624
gpu_tot_sim_insn = 307200
gpu_tot_ipc =       46.3768
gpu_tot_issued_cta = 16
gpu_occupancy = 10.9131%
gpu_tot_occupancy = 10.9131%
max_total_param_size = 0
gpu_stall_dramfull = 0
gpu_stall_icnt2sh    = 0
```

Conv2dV2

```
gpu_sim_cycle = 7416
gpu_sim_insn = 451200
gpu_ipc =       60.8414
gpu_tot_sim_cycle = 7416
gpu_tot_sim_insn = 451200
gpu_tot_ipc =       60.8414
gpu_tot_issued_cta = 16
gpu_occupancy = 12.4255%
gpu_tot_occupancy = 12.4255%
max_total_param_size = 0
gpu_stall_dramfull = 0
gpu_stall_icnt2sh    = 0
```

After the shared memory optimization we utilize the shared memory structure which is linked with the L1 Cache in GPU architecture, this increased the IPC to 60.8414 (31.2% increase from baseline), which is naturally can also be reflected by the occupancy data is the way the data is stored.

The increased occupancy indicates improved thread-level parallelism, which allows more warps to be active and helps in better latency hiding

```
  L1D_total_cache_accesses = 8192
  L1D_total_cache_misses = 1664
  L1D_total_cache_miss_rate = 0.2031
  L1D_total_cache_pending_hits = 0
  L1D_total_cache_reservation_fails = 0
  L1D_cache_data_port_util = 0.255
  L1D_cache_fill_port_util = 0.035
```

```
  L1D_total_cache_accesses = 2736
  L1D_total_cache_misses = 1632
  L1D_total_cache_miss_rate = 0.5965
  L1D_total_cache_pending_hits = 0
  L1D_total_cache_reservation_fails = 0
  L1D_cache_data_port_util = 0.029
  L1D_cache_fill_port_util = 0.023
```

This also reduces the L1D total cache access to 2736 (around 66% reduction), as the data are stored in L1D in tiles and doesn't require unnecessary L1D cache accesses.

```
gpgpu_n_tot_thrd_icount = 311296
gpgpu_n_tot_w_icount = 9728
gpgpu_n_stall_shd_mem = 5760
gpgpu_n_mem_read_local = 0
gpgpu_n_mem_write_local = 0
gpgpu_n_mem_read_global = 896
gpgpu_n_mem_write_global = 512
gpgpu_n_mem_texture = 0
gpgpu_n_mem_const = 0
gpgpu_n_load_insn  = 73728
gpgpu_n_store_insn = 4096
gpgpu_n_shmem_insn = 0
gpgpu_n_sstarr_insn = 0
gpgpu_n_tex_insn = 0
gpgpu_n_const_mem_insn = 0
gpgpu_n_param_mem_insn = 28672
gpgpu_n_shmem_bkconflict = 0
gpgpu_n_cache_bkconflict = 0
gpgpu_n_intrawarp_mshr_merge = 0
gpgpu_n_cmem_portconflict = 0
```

```
gpgpu_n_tot_thrd_icount = 554496
gpgpu_n_tot_w_icount = 17328
gpgpu_n_stall_shd_mem = 2464
gpgpu_n_mem_read_local = 0
gpgpu_n_mem_write_local = 0
gpgpu_n_mem_read_global = 896
gpgpu_n_mem_write_global = 512
gpgpu_n_mem_texture = 0
gpgpu_n_mem_const = 0
gpgpu_n_load_insn  = 42048
gpgpu_n_store_insn = 4096
gpgpu_n_shmem_insn = 42048
gpgpu_n_sstarr_insn = 0
gpgpu_n_tex_insn = 0
gpgpu_n_const_mem_insn = 0
gpgpu_n_param_mem_insn = 28672
gpgpu_n_shmem_bkconflict = 0
gpgpu_n_cache_bkconflict = 0
gpgpu_n_intrawarp_mshr_merge = 0
gpgpu_n_cmem_portconflict = 0
```

We can see that the shared mem instruction are now being used and is 42048 and the load instruction have reduced to 42048 (around 43% reduction).

The use of tiled storage in shared memory enables data reuse across multiple threads in a block, which is not possible with direct global memory access, leading to better temporal locality.

By utilizing shared memory more effectively, Conv2dV2 significantly reduces global memory traffic, which not only improves latency but also frees up bandwidth for other operations