Ehi Simon - 14352307
Alisha Augustin - 14389451

ECEC 413-001
06/08/2023

CUDA: Separable Convolution

```
__constant__ float kernel_c[2 * HALF_WIDTH + 1];

__global__ void convolve_rows_kernel_naive(float* result, float* input, float*
kernel, int num_cols, int num_rows)
{
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;

    if (thread_id >= num_rows)
        return;

    for (int x = 0; x < num_cols; x++)
    {
        int j1 = x - HALF_WIDTH;
        int j2 = x + HALF_WIDTH;

        if (j1 < 0)
            j1 = 0;
        if (j2 >= num_cols)
            j2 = num_cols - 1;

        int i1 = j1 - x;

        j1 = j1 - x + HALF_WIDTH;
        j2 = j2 - x + HALF_WIDTH;

        result[thread_id * num_cols + x] = 0.0f;
        for (int i = i1, j = j1; j <= j2; j++, i++)
            result[thread_id * num_cols + x] += kernel[j] * input[thread_id *
            num_cols + x + i];
    }
}
```

```
__global__ void convolve_columns_kernel_naive(float* result, float* input, float* kernel, int num_cols, int num_rows)
{
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;

    if (thread_id >= num_rows)
        return;

    for (int x = 0; x < num_cols; x++)
    {
        int j1 = thread_id - HALF_WIDTH;
        int j2 = thread_id + HALF_WIDTH;

        if (j1 < 0)
            j1 = 0;
        if (j2 >= num_rows)
            j2 = num_rows - 1;

        int i1 = j1 - thread_id;

        j1 = j1 - thread_id + HALF_WIDTH;
        j2 = j2 - thread_id + HALF_WIDTH;

        result[thread_id * num_cols + x] = 0.0f;
        for (int i = i1, j = j1; j <= j2; j++, i++)
            result[thread_id * num_cols + x] += kernel[j] * input[thread_id * num_cols + x + (i * num_cols)];
    }
}
```

Figure 1: Implementation of Naive 2D Convolution Kernels

The first function, *convolve_rows_kernel_naive*, is a CUDA kernel function that performs row-wise convolution on a 2D input matrix using a given kernel. The function takes in several parameters: result is the output matrix where the convolved result will be stored, input is the input matrix on which convolution is applied, kernel is the convolution kernel, num_cols is the number of columns in the input matrix, and num_rows is the number of rows in the input matrix. The kernel function is executed by multiple threads in parallel, and each thread is responsible for processing one row of the input matrix. The thread ID is calculated using the blockIdx.x, blockDim.x, and threadIdx.x variables.

Inside the kernel function, the thread ID is used to determine the row that the thread should process. If the thread ID is greater than or equal to the number of rows, it means there are no more rows to process, so the thread returns. For each row, the function iterates over the columns of the input matrix. It calculates the starting and ending column indices (j1 and j2) for the convolution based on the current column index (x) and the width of the convolution kernel (HALF_WIDTH).

Next, it calculates the starting index (i1) for the input matrix based on j1 and the current column index (x). Then, j1 and j2 are adjusted to be relative to the current column (x) by subtracting x and adding HALF_WIDTH. The result for the current element in the output matrix is initialized to 0.0. Then, a nested loop is used to perform the convolution. The loop iterates over the indices i and j to access the corresponding elements in the kernel and input matrices. It multiplies the kernel value with the corresponding input value and accumulates the result in the output matrix. The second function*, convolve_columns_kernel_naive*, is similar to the first one, but it performs column-wise convolution instead of row-wise convolution. It processes the input matrix column-wise, with each thread responsible for one column. The calculations and nested loops are adjusted accordingly to handle column-wise convolution.

```cpp
__global__ void convolve_rows_kernel_optimized(float* result, float* input, int num_cols, int num_rows)
{
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;

    if (thread_id >= num_rows)
        return;

    for (int x = 0; x < num_cols; x++)
    {
        int j1 = x - HALF_WIDTH;
        int j2 = x + HALF_WIDTH;

        if (j1 < 0)
            j1 = 0;
        if (j2 >= num_cols)
            j2 = num_cols - 1;

        int i1 = j1 - x;

        j1 = j1 - x + HALF_WIDTH;
        j2 = j2 - x + HALF_WIDTH;

        result[thread_id + num_cols * x] = 0.0f;
        for (int i = i1, j = j1; j <= j2; j++, i++)
            result[thread_id + (num_cols * x)] += kernel_c[j] * input[thread_id * num_cols + x + i];
    }
}
```

```cpp
__global__ void convolve_columns_kernel_optimized(float* result, float* input, int num_cols, int num_rows)
{
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;

    if (thread_id >= num_rows)
        return;

    for (int x = 0; x < num_cols; x++)
    {
        int j1 = thread_id - HALF_WIDTH;
        int j2 = thread_id + HALF_WIDTH;

        if (j1 < 0)
            j1 = 0;
        if (j2 >= num_rows)
            j2 = num_rows - 1;

        int i1 = j1 - thread_id;
        j1 = j1 - thread_id + HALF_WIDTH;
        j2 = j2 - thread_id + HALF_WIDTH;

        result[thread_id * num_cols + x] = 0.0f;
        for (int i = i1, j = j1; j <= j2; j++, i++)
            result[thread_id * num_cols + x] += kernel_c[j] * input[thread_id + (num_cols * x) + i];
    }
}
```

Figure 2: Implementation of Optimized 2D Convolution Kernels

The *convolve_rows_kernel_optimized* kernel performs row-wise convolution. It takes in a 2D input matrix represented by the input array, where each row of the matrix is stored in a linearized format. The result array stores the output of the convolution operation. The *num_cols* and *num_rows* parameters specify the dimensions of the input matrix. The kernel

iterates over each row of the input matrix using the *thread_id* variable, which is calculated based on the block and thread indices. It applies a convolution filter to the input matrix by multiplying each element of the filter with the corresponding element in the input matrix. The resulting convolved value is stored in the result array.

The *convolve_columns_kernel_optimized* kernel performs column-wise convolution. It is similar to the row-wise convolution kernel, but the iteration is done over each column of the input matrix. The result and input arrays have a different indexing pattern to handle the column-wise convolution. In both kernels, the indices *j1* and *j2* determine the range of elements in the input matrix that are involved in the convolution operation. The *i1* variable is used to calculate the corresponding index in the input matrix, and the convolution is performed using nested loops over the indices *i* and *j*.

```
void compute_on_device(float *gpu_result, float *matrix_c, float *kernel, int num_cols, int num_rows)
{
    /* Create pointers for computations. */
    float *d_gpu_result = NULL;
    float *d_matrix = NULL;
    float *d_kernel = NULL;
    float *d_row_col = NULL;

    int num_elements = num_rows * num_cols;
    int kernel_size = (2 * HALF_WIDTH) + 1;

    cudaMalloc((void**)&d_matrix, num_elements * sizeof(float));
    cudaMalloc((void**)&d_kernel, kernel_size * sizeof(float));
    cudaMalloc((void**)&d_gpu_result, num_elements * sizeof(float));
    cudaMalloc((void**)&d_row_col, num_elements * sizeof(float));

    cudaMemcpy(d_matrix, matrix_c, num_elements * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_kernel, kernel, kernel_size * sizeof(float), cudaMemcpyHostToDevice);

    dim3 thread_block(THREAD_BLOCK_SIZE, 1, 1);
    int num_thread_blocks = (num_rows + thread_block.x - 1) / thread_block.x;
    dim3 grid(num_thread_blocks, 1);

    struct timeval start, stop;
    gettimeofday(&start, NULL);
    convolve_rows_kernel_naive<<<grid, thread_block>>>(d_row_col, d_matrix, d_kernel, num_rows, num_cols);
    cudaDeviceSynchronize();
    convolve_columns_kernel_naive<<<grid, thread_block>>>(d_gpu_result, d_row_col, d_kernel, num_rows, num_cols);
    cudaDeviceSynchronize();
    gettimeofday(&stop, NULL);

    printf("Naive Execution time = %fs\n", (float)(stop.tv_sec - start.tv_sec + (stop.tv_usec - start.tv_usec) /

    cudaMemcpy(gpu_result, d_gpu_result, num_elements * sizeof(float), cudaMemcpyDeviceToHost);
    cudaFree(d_gpu_result);
    cudaFree(d_matrix);
    cudaFree(d_kernel);
    cudaFree(d_row_col);
}
```

Figure 3: Implementation of CUDA Naive 2D Convolution

The first function, compute_on_device, takes as input the pointers to the GPU result, matrix C, kernel, number of columns, and number of rows. It begins by allocating memory on the GPU for the matrix, kernel, GPU result, and an intermediate row/column buffer. The matrix and kernel data are then copied from the CPU to the GPU device using *cudaMemcpy*. Next, it sets up the thread block and grid dimensions for the GPU kernel launch. The program then launches two GPU kernels sequentially: and *convolve_columns_kernel_naive*. These kernels perform the row-wise and column-wise convolutions, respectively. After each kernel launch, *cudaDeviceSynchronize* is called to ensure that the GPU computations are completed. Finally, the execution time is printed, the GPU result is copied back to the host, and the allocated GPU memory is freed using cudaFree.

```
void compute_on_device_optimized(float *gpu_result, float *matrix_c, float *kernel, int
num_cols, int num_rows)
{

    float *d_gpu_result = NULL;
    float *d_matrix = NULL;
    float *d_row_col = NULL;

    int num_elements = num_rows * num_cols;
    int kernel_size = (2 * HALF_WIDTH) + 1;


    cudaMalloc((void**)&d_matrix, num_elements * sizeof(float));
    cudaMemcpy(d_matrix, matrix_c, num_elements * sizeof(float), cudaMemcpyHostToDevice);

    cudaMalloc((void**)&d_gpu_result, num_elements * sizeof(float));
    cudaMalloc((void**)&d_row_col, num_elements * sizeof(float));

    /* Set up thread block and grid */
    dim3 thread_block(THREAD_BLOCK_SIZE, 1, 1);
    int num_thread_blocks = (num_rows + thread_block.x - 1) / thread_block.x;
    dim3 grid(num_thread_blocks, 1);

    struct timeval start, stop;
    cudaMemcpyToSymbol(kernel_c, kernel, kernel_size * sizeof(float));
    gettimeofday(&start, NULL);
    convolve_rows_kernel_optimized<<<grid, thread_block>>>(d_row_col, d_matrix, num_rows,
    num_cols);
    cudaDeviceSynchronize();
    convolve_columns_kernel_optimized<<<grid, thread_block>>>(d_gpu_result, d_row_col,
    num_rows, num_cols);
    cudaDeviceSynchronize();
    gettimeofday(&stop, NULL);

    printf("Optimized Execution time = %fs\n", (float)(stop.tv_sec - start.tv_sec + (stop.
    tv_usec - start.tv_usec) / (float)1000000));
    cudaMemcpy(gpu_result, d_gpu_result, num_elements * sizeof(float),
    cudaMemcpyDeviceToHost);

    cudaFree(d_gpu_result);
    cudaFree(d_matrix);
    cudaFree(d_row_col);
}
```

Figure 4: Implementation of CUDA Optimized 2D Convolution

The second function, *compute_on_device_optimized*, is a modified version of the first function with some optimizations. It performs the same matrix convolution operation but aims to improve the execution time. Similar to the first function, it allocates memory on the

GPU, copies the matrix data from the host to the device, and sets up the thread block and grid dimensions. Additionally, it uses *cudaMemcpyToSymbol* to copy the kernel data to a constant memory symbol kernel_c, which can provide faster access for the GPU kernels. Then the function launches the optimized GPU kernels *convolve_rows_kernel_optimized* and *convolve_columns_kernel_optimized*, and waits for the GPU computations to complete using *cudaDeviceSynchronize*. The GPU result is copied back to the host and the allocated GPU memory is freed.

Performance (Execution Time) Comparison

	Block Size	**128**	**256**	**512**	**1024**
Matrix Size	**Type**				
2048x2048	**Gold**	0.405787s	0.407196s	0.404237s	0.405211s
	Naïve	0.015844s	0.031121s	0.063407s	0.134686s
	Optimized	0.009179s	0.016990s	0.034147s	0.061577s
4096x4096	**Gold**	1.300282s	1.420355s	1.295892s	1.368509s
	Naïve	0.031601s	0.062138s	0.126320s	0.260707s
	Optimized	0.018434s	0.034065s	0.060574s	0.124092s
8192x8192	**Gold**	5.162479s	5.399398s	5.124732s	5.372020s
	Naïve	0.093498s	0.120811s	0.246221s	0.504534s
	Optimized	0.053040s	0.063203s	0.123225s	0.245698s

Speedup Comparison

	Block Size	**128**	**256**	**512**	**1024**
Matrix Size	**Type**				
2048x2048	**Naïve**	25.6	13.08	6.37	3.01
	Optimized	44.2	23.97	11.84	6.58
4096x4096	**Naïve**	41.15	22.86	10.26	5.25
	Optimized	70.54	41.7	21.39	11.03
8192x8192	**Naïve**	55.2	44.7	20.8	10.65
	Optimized	97.33	85.43	41.59	21.86

The speedup obtained measures the improvement in execution time achieved by running the matrix calculations in parallel compared to running them sequentially. In the given results above, the parallel implementation consistently demonstrates speedup for all matrix sizes and thread configuration. sIn all cases, the optimized implementation outperforms the naïve implementation. This is evident from the higher speedup values across all block sizes and matrix sizes. The optimized implementation shows larger speedup values compared to the naïve implementation, particularly for larger matrix sizes. This indicates that the applied optimizations are effective in enhancing the performance of the convolution algorithm on the GPU.