Noah Robinson
Jack Pinkstone
Professor Kandasamy
ECEC 413
13 June 2023

CUDA: Separable convolution

This assignment required the students to use CUDA to write code to parallelize the separable convolution process. We parallelized both rows and columns and then optimized it further with constant memory. CUDA programming improves performance by allowing programmers to offload computationally intensive tasks to the GPU. This can lead to significant speedups, especially for applications that can be parallelized. CUDA programming is relatively easy to learn, and there are many resources available to help programmers get started.

Design

```
size_t mem_size = num_elements * sizeof(float);

/* Allocate memory */
cudaMalloc((void**)&device_result, mem_size);
cudaMalloc((void**)&device_input, mem_size);
cudaMalloc((void**)&device_input_opt, mem_size);
cudaMalloc((void**)&device_kernel, width * sizeof(float));

/* Copy memory from host to device */
cudaMemcpy(device_input, matrix_c, mem_size, cudaMemcpyHostToDevice);
cudaMemcpy(device_input_opt, matrix_c, mem_size, cudaMemcpyHostToDevice);
cudaMemcpy(device_kernel, kernel, width * sizeof(float), cudaMemcpyHostToDevice);

dim3 threads(THREAD_BLOCK_SIZE, THREAD_BLOCK_SIZE);
int dim1 = num_cols/THREAD_BLOCK_SIZE;
int dim2 = num_rows/THREAD_BLOCK_SIZE;
dim3 grid(dim1, dim2);

/* Timing variables */
struct timeval start, stop;
struct timeval start2, stop2;

gettimeofday(&start, NULL);
convolve_rows_kernel_naive<<< grid, threads >>>(device_result, device_input, device_kernel, num_cols, num_rows, half_width);
/* Sync necessary */
cudaDeviceSynchronize();
convolve_columns_kernel_naive<<< grid, threads >>>(device_input, device_result, device_kernel, num_cols, num_rows, half_width);
gettimeofday(&stop, NULL);

float exec_time1 = (float)(stop.tv_sec - start.tv_sec + (stop.tv_usec - start.tv_usec) / (float)1000000);
printf ("[Naive] = %13fs\n", exec_time1);

/* Copy result out of device */
cudaMemcpy(gpu_result, device_input, mem_size, cudaMemcpyDeviceToHost);

gettimeofday(&start2, NULL);
cudaMemcpyToSymbol(kernel_optimized, kernel, width * sizeof(float));
convolve_rows_kernel_optimized<<< grid, threads >>>(device_result, device_input_opt, num_cols, num_rows, half_width);
/* Sync necessary */
cudaDeviceSynchronize();
convolve_columns_kernel_optimized<<< grid, threads >>>(device_input_opt, device_result, num_cols, num_rows, half_width);
gettimeofday(&stop2, NULL);

float exec_time2 = (float)(stop2.tv_sec - start2.tv_sec + (stop2.tv_usec - start2.tv_usec) / (float)1000000);
printf ("[Optim] = %13fs\n", exec_time2);

/* Copy result out of device */
cudaMemcpy(result_with_optimization, device_input, mem_size, cudaMemcpyDeviceToHost);
```

For the host side of this assignment we first allocate memory, then copy the matrices to the device from the host's memory. Then we create our grind and threads. Then we simply call the

kernel to convolve the rows then columns after. A sync is needed in between incase threads from the rows operations don't finish at the same time. This process is then repeated for the optimized kernel calls and the timing output is displayed.

```
__global__ void convolve_columns_kernel_naive(float *result, float *input, float *kernel, int num_cols, int num_rows, int half_width)
{
    int i, i1, j, j1, j2, x, y;

    x = blockDim.x * blockIdx.x + threadIdx.x;
    y = blockDim.y * blockIdx.y + threadIdx.y;

    j1 = y - half_width;
    j2 = y + half_width;

    /* Clamp at the edges of the matrix */
    j1 = max(0, j1);
    j2 = min(num_rows - 1, j2);

    /* Obtain relative position of starting element from element being convolved */
    i1 = j1 - y;

    j1 = j1 - y + half_width; /* Obtain the operating width of the kernel.*/
    j2 = j2 - y + half_width;

    /* Convolve along column */
    result[y * num_cols + x] = 0.0f;
    for (i = i1, j = j1; j <= j2; j++, i++)
        result[y * num_cols + x] += kernel[j] * input[y * num_cols + x + (i * num_cols)];

    return;
}
```

This is the kernel side operations for the naive implementation. We just add the section about indexing to the gold code so that each thread knows its position.

```
__global__ void convolve_rows_kernel_optimized(float *result, float *input, int num_cols, int num_rows, int half_width)
{
    __shared__ float inputShared[(THREAD_BLOCK_SIZE + HALF_WIDTH * 2) * THREAD_BLOCK_SIZE];
    int i, i1, j, j1, j2, x, y;

    x = blockDim.x * blockIdx.x + threadIdx.x;
    y = blockDim.y * blockIdx.y + threadIdx.y;

    int leftHaloIndex = (blockIdx.x - 1) * blockDim.x + threadIdx.x;
    int rightHaloIndex = (blockIdx.x + 1) * blockDim.x + threadIdx.x;

    /* Load input elements into shared memory with halo regions */
    if (threadIdx.x >= (blockDim.x - half_width))
    {
        inputShared[threadIdx.y * blockDim.y + (threadIdx.x - (blockDim.x - half_width))] = (leftHaloIndex < 0) ? 0.0 : input[leftHaloIndex + y * num_rows];
    }

    if (x < num_cols)
        inputShared[threadIdx.y * blockDim.y + (threadIdx.x + half_width)] = input[y * num_rows + x];
    else
        inputShared[threadIdx.y * blockDim.y + (threadIdx.x + half_width)] = 0.0;

    if (threadIdx.x < half_width)
    {
        inputShared[threadIdx.y * blockDim.y + threadIdx.x + (blockDim.x + half_width)] = (rightHaloIndex >= num_cols) ? 0.0 : input[rightHaloIndex + y * num_rows];
    }

    __syncthreads();

    j1 = x - half_width;
    j2 = x + half_width;

    j1 = max(0, j1);
    j2 = min(num_rows - 1, j2);

    /* Obtain relative position of starting element from element being convolved */
    i1 = j1 - x;

    j1 = j1 - x + half_width; /* Obtain operating width of the kernel */
    j2 = j2 - x + half_width;

    /* Convolve along row */
    result[y * num_cols + x] = 0.0f;
    for (i = i1, j = j1; j <= j2; j++, i++)
        result[y * num_cols + x] += kernel_optimized[j] * inputShared[threadIdx.y * blockDim.x + (threadIdx.x + half_width) + i];

    return;
}
```

The optimized version was more tricky, here we had to load elements into shared memory with halo regions to improve efficiency. The rest of this function is just the same as the gold implementation.

Performance Comparison

			Reference	
			Matrix Size	
	Seconds	2048	4096	8192
		0.407606s	1.368606s	5.154757s
			CUDA	
			Matrix Size	
	Seconds	2048	4096	8192
Thread Block	4	0.000818s	0.002882s	0.010991s
	8	0.000346s	0.001051s	0.003745s
	16	0.000314s	0.000883s	0.003238s
	32	0.000312s	0.000835s	0.003060s
			CUDA OPT	
			Matrix Size	
	Seconds	2048	4096	8192
Thread Block	4	0.000317s	0.001155s	0.004457s
	8	0.000146s	0.000449s	0.001645s
	16	0.000151s	0.000437s	0.001669s
	32	0.000215s	0.000702s	0.002649s

There is a MASSIVE improvement in the CUDA implementation here, more than we've ever seen in this course. The optimized case was running over 3000x faster than the serial version for the eight and 16 thread block cases. I notice that lower thread block sizes such as four and higher thread block sizes like 32 were not as efficient for this application as thread counts of eight and 16 where.