Parallel programming, convolutions in CUDA

Christian Zanzi
E-mail address
christian.zanzi@edu.unifi.it

Abstract

In this report, we analyze the speedup achievable using a GPU to calculate a convolution on an image. First, we consider a sequential program written in C++ that works on a CPU, then we compare its execution time with a new program written in CUDA that works on NVIDIA GPUs. In this writing, some tricks are shown to make the convolution quicker on a GPU, knowing that the data transfer is the most burdensome operation.

1. Introduction

In image processing and computer vision, image convolutions are widely used to find features of images or to modify them. It is a simple mathematical operation done between matrices or tensors of numeric values.
A filter is applied to each pixel of an image to obtain an output value. The **Figure 1** explains graphically the operation done on a single pixel. In practice is a sum of multiplication between the filter and the pixel values surrounding the pixel in which the filter is centered.

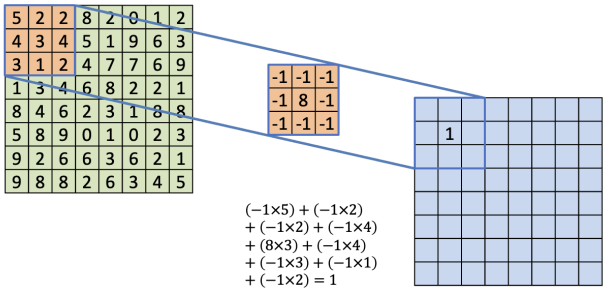

Figure 1. Filter applied on a single pixel.

1.1. Test image and filter

The tests are done using a 2048x2048 pixels image on which an edge detection filter is applied to obtain information on the edges inside the original image.

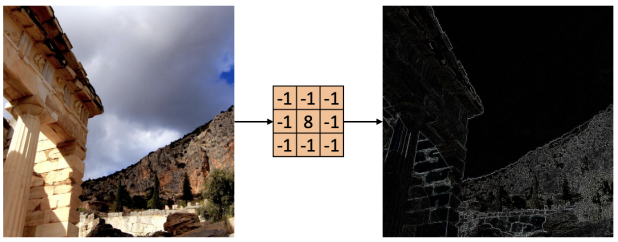

Figure 2. Filter applied on a single pixel.

1.2. Developement metodologies

C++ is used as the programming language for the sequential program's code, and CUDA is used for the parallel version.
The sequential version just calculates linearly for each value of the matrix the convolution operation, while the CUDA program splits the execution across the streaming multiprocessors of a GPU.
Some tricks to speed up the convolution are used to store and load the image in the parallel program:

- use of the constant cache of the streaming multiprocessors to store the filter;
- use of the shared memory to store the tiles assigned from the GPU's scheduler to each streaming multiprocessor;
- 2 batch loading.

1.3. Performance evaluation

The speedup $\frac{t_{seq}}{t_{par}}$ measures the performance improvement obtained from the parallelization of the sequential code. Each program is executed 5 times, and the execution time is recorded with the *chrono* library. The sequential and parallel programs have been timed only considering the convolution operation without taking into account the time spent for the data transfer.

1.4. Test environment

The sequential program is runned on
Model name: MacBook Air
Chip: Apple M3
total number of Cores: 8 (prestazioni 4 ed efficienza 4)
Memory: 16 GB
Using the Mac terminal to run it:

```
clang++ -o sequential_convolution
    sequential_convolution.cpp
./sequential_convolution input.jpg output.jpg
```

For the parallel program, we used the service offered by Amazon Web Services (AWS), called Elastic Compute Cloud (EC2), which allows scalable computational capability.
We launched a G4dn instance that includes the NVIDIA T4 GPU, useful for our problem testing.
Instance name: g4dn.2xlarge
Chip: NVIDIA T4
total number of Cuda Cores: 2560
Memory: 32 GiB
To run the program, we used the instance terminal:

```
nvcc -o cuda_convolution cuda_convolution.cu
./cuda_convolution input.jpg output.jpg
```

2. Programs comparison

In this section, we will discuss the difference between the programs and their implementation.
We used the *stbi* library to load the image and save it after the convolution. In the two programs, the code used for image loading is unaltered

```
int width, height, channels;
unsigned char *input_image = stbi_load(argv[1], &
    width, &height, &channels, 0);
if (!input_image) {
   std::cerr << "Failed to load image\n";
   return 1;
}

size_t image_size = width * height * channels;
float *host_input = new float[image_size];
float *host_output = new float[image_size];
for (size_t i = 0; i < image_size; ++i)
   host_input[i] = input_image[i] / 255.0f;
```

With *stbi_load()* we are obtaining an unsigned char array with each pixel's value. It is common to normalize these values for:

- transforming them to floating points, since GPUs work better with floats;
- increasing the numeric precision and limiting the risk of overflow;
- controlling the results obtained from the convolution with filters with small values.

2.1. Sequential program

The convolution function of the sequential program loops over channels, height rows, and width columns to apply the filter convolution on every pixel.
Before making the sums, the code checks if the index on which the operations should be done is correct.

```
void convolution(float *input, float *output, float *filter, int channels, int width, int height) {
   // image matrix memorized in row-major order, linear memory access to consecutive addresses
   for (int k = 0; k < channels; ++k) {
      for (int y = 0; y < height; ++y) {
         for (int x = 0; x < width; ++x) {
            float sum = 0;

            for (int fy = - FILTER_RADIUS; fy <= FILTER_RADIUS; ++fy) {
               for (int fx = - FILTER_RADIUS; fx <= FILTER_RADIUS; ++fx) {
                  // setting the index position of the image tile with respect to the filter
                  int nx = x + fx;
                  int ny = y + fy;
                  // border handling with replicate padding for the ghost cells
                  if (nx < 0) nx = 0;
                  if (ny < 0) ny = 0;
                  if (nx >= width) nx = width - 1;
                  if (ny >= height) ny = height -1;

                  const int idx = (ny * width + nx) * channels + k;
                  sum += input[idx] * filter[(fx + FILTER_RADIUS) * FILTER_WIDTH + (fy + FILTER_RADIUS)];
               }
            }
            int out_idx = (y * width + x) * channels + k;
            output[out_idx] = sum;
         }
      }
   }
}
```

The program prints the execution time at the end and saves the convoluted image.

2.2. Parallel program

The parallel program has an initial definition of constant variables that do not change during execution.

```
#define FILTER_WIDTH 3
#define FILTER_RADIUS (FILTER_WIDTH / 2)
#define TILE_WIDTH 32
#define w (TILE_WIDTH + FILTER_WIDTH - 1)
#define clamp(x) (min(max((x), 0.0f), 1.0f))
__constant__ float device_filter[FILTER_WIDTH * FILTER_WIDTH];
```

At the beginning, the variable that will be stored in the GPU's constant memory is also defined.

2.2.1 launch_convolution()

To launch the kernel on the GPU, we have defined a function that receives the image width, height, and channels and specifies the space on the device memory (*cudaMalloc*). It transfers the image data from the host to the device with the *cudaMemcpy()*. The filter has to be stored in the constant memory because it does not change during execution and is the same for every convolution operation, and for that reason is used *cudaMemcpyToSymbol()*.
In CUDA, when a kernel is called, the work is divided between multiple blocks of multiple threads. The dimension of the block defines how many threads operate in it. The grid dimension specifies how many blocks the GPU scheduler has to manage and assign to the streaming multiprocessors.

```
void launch_convolution(float *host_input, float *host_filter, float *host_output, int channels, int width, int height) {
   float *device_input, *device_output;
   size_t image_size = width * height * channels * sizeof(float);
   size_t filter_size = FILTER_WIDTH * FILTER_WIDTH * sizeof(float);

   cudaMalloc(&device_input, image_size);
   cudaMalloc(&device_output, image_size);

   cudaMemcpy(device_input, host_input, image_size, cudaMemcpyHostToDevice);
   cudaMemcpyToSymbol(device_filter, host_filter, filter_size);

   dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
   dim3 dimGrid((width + TILE_WIDTH - 1) / TILE_WIDTH, (height + TILE_WIDTH - 1) / TILE_WIDTH);

   std::chrono::time_point<std::chrono::system_clock> start, end;
   start = std::chrono::system_clock::now();
   convolution<<<dimGrid, dimBlock>>>(device_input, device_output, channels, width, height);

   cudaDeviceSynchronize();
   end = std::chrono::system_clock::now();
   std::chrono::duration<double> milliseconds = end - start;
   std::cout << "Total convolution time: " << milliseconds.count() << " ms" << std::endl;

   cudaError_t err = cudaGetLastError();
   if (err != cudaSuccess) {
       std::cerr << "CUDA error: " << cudaGetErrorString(err) << std::endl;
```

```
    }

    cudaMemcpy(host_output, device_output,
     image_size, cudaMemcpyDeviceToHost);
    cudaFree(device_input);
    cudaFree(device_output);
}
```

The function transfers the data back to the host when the convolution is applied by each thread on the original image using *cudaMemcpy()*, and then frees the allocation on the GPU.

2.2.2 convolution()

The kernel called from the *launch_convolution()* contains the instruction that each thread of every block should execute. The first operation of the block is the instantiation of the variable that will memorize the values of the tile assigned to it in the shared memory. We have used the shared memory to reduce the time spent to access the global memory, since it is outside the multiprocessor, indeed, it requires more time to retrieve the data.

```
    __shared__ float N_s[w][w];
```

Using the shared memory is a good choice not only for the access time reduction but also because we can load into it the values outside the tile's border that are needed to compute the convolution and are assigned and used from other blocks for its tile. To load every pixel value that we need, we have implemented a 2-batch loading technique.
The shared memory with size $(TILE\_WIDTH + FILTER\_WIDTH - 1)^2$ must be loaded by the threads with the values of the image inside the global memory before doing the convolution. Since the threads of a block are not able to fully load the shared memory, we need to let them bring more than just one value from the global memory. Fortunately, the number of uploads required for each thread, in our case, where $TILE\_WIDTH >> FILTER\_WIDTH$, is at most two, and usually not for all of them. so for each thread we compute the first loading

```
      int dest = threadIdx.y * TILE_WIDTH +
    threadIdx.x,
          destY = dest / w, destX = dest % w,
          srcY = blockIdx.y * TILE_WIDTH + destY
    - FILTER_RADIUS,
          srcX = blockIdx.x * TILE_WIDTH + destX
    - FILTER_RADIUS,
          src = (srcY * width + srcX) * channels
    + k;

      if (srcY >= 0 && srcY < height && srcX >= 0
     && srcX < width)
         N_s[destY][destX] = input[src];
      else
         N_s[destY][destX] = 0.0f;
```

and the second one

```
      dest = threadIdx.y * TILE_WIDTH + threadIdx
    .x + TILE_WIDTH * TILE_WIDTH;
      destY = dest / w; destX = dest % w;
      srcY = blockIdx.y * TILE_WIDTH + destY -
    FILTER_RADIUS;
      srcX = blockIdx.x * TILE_WIDTH + destX -
    FILTER_RADIUS;
      src = (srcY * width + srcX) * channels + k;

      if (destY < w) {
         if (srcY >= 0 && srcY < height && srcX
    >= 0 && srcX < width)
            N_s[destY][destX] = input[src];
         else
            N_s[destY][destX] = 0.0f;
      }
      __syncthreads();
```

After each thread of a block finishes uploading the pixel values, the convolution can be computed by applying the filter to the pixel assigned to the thread and its neighbours.

```
      float accum = 0.0f;
      for (int fy = 0; fy < FILTER_WIDTH; fy++)
         for (int fx = 0; fx < FILTER_WIDTH; fx
    ++)
            accum += N_s[threadIdx.y + fy][
    threadIdx.x + fx] * device_filter[fy *
    FILTER_WIDTH + fx];

      int y = blockIdx.y * TILE_WIDTH + threadIdx
    .y;
      int x = blockIdx.x * TILE_WIDTH + threadIdx
    .x;
      if (y < height && x < width)
         output[(y * width + x) * channels + k] =
     clamp(accum);
      __syncthreads();
```

Before ending, the normalized result is stored in the global memory. When copied to the host memory, the value can be reconverted to the unsigned char type to obtain the new image.

3. Sperimental results

The execution times of the programs have been collected, considering only the time spent on the convolution without considering the time that we need to move the data from and to the host. We do this to focus our attention on the actual operation of convolution. Practically, when we consider a huge dataset of images, the data transmission cost is much less relevant. The results are shown in *Table 1*.
We can notice how the execution time required for the CUDA convolution is 424 ($Speedup = \frac{T\_seq}{T\_par} = \frac{0.399}{0.00094} = 424,052$) times less than the time spent on the sequential operation. The uploading of the filter into the constant cache and the block's tile into the shared memory is fast, thanks to the 2-batch loading and the leverage of constant memory. The data transfer for just one image is the bottleneck that slows down the execution of the entire CUDA program; the uploading to the GPU takes between 20 and 35 milliseconds, reducing the speedup obtained from the program.
In *Table 2*, we show the time the program takes to upload to the GPU and download the data from it. For just an image is quite burdensome, but especially in machine learning, where there are more images to work with, using batches to load them reduces the total data transfer cost and improves the performance.

Version	t1	t2	t3	t4	t5	t6	mean
Sequential(ms)	0.433983	0.423375	0.380958	0.378117	0.375455	0.402566	0.399075667
Parallel(ms)	0.000800096	0.00078254	0.00118689	0.0011847	0.000970656	0.000721927	0.000941135

Table 1. Speedup results.

	t1	t2	t3	t4	t5	t6
Upload(ms)	0.325928	0.24965	0.24803	0.246484	0.248367	0.296436
Download(ms)	0.0398763	0.0399647	0.0405561	0.0400545	0.0398323	0.0401356

Table 2. GPU data tranfer.

4. Conclusion

We have demonstrated that CUDA convolution is a hundred times faster than convolution performed on a CPU. Additionally, we have implemented several techniques that enhance execution on GPU architecture; using constant cache and shared memory reduces the multiple accesses of streaming multiprocessors to global memory. In a convolution context, 2-batch loading makes filling shared memory extremely beneficial for the operation. The upload cost for one image is the most computationally expensive part of the CUDA program and slows down the process. In future implementation, we should test the programs with more than one image, and perform other tests on the dimension of the block and the number of threads defined for them.