182.731 GPU Architectures and Computing - Exercise 1

Group 1:
Hiti Mario, 01327428
Gollmann Christian, 01435044
Hinterseer Simon, 09925802
Loch Adam, 12044788

April 7, 2022

Contents

1 Task description

A histogram of the pixel colors of a picture is to be assembled using CUDA kernels. In particular the use of the `atomicAdd` function is to be examined in terms of performance, depending on the number of colors that occur in the picture.

2 Compilation and usage

To compile the code use the following commands:

```
cd Ex1
mkdir build && cd build
make -D CMAKE_BUILD_TYPE=Release ..
```

To run the code use:

```
cd Ex1/build
make run
```

3 Kernels

All kernels can be found in the file **main.cu**. They can also be found in the appendix of this document. The kernels can be called in any grid configuration (except for the linear kernel, see description later). The number of threads does not have to be equal to the problem size.

Each kernel takes the following arguments:

1. device pointer to the array holding the results (the color buckets). This array needs to be initialized to zeros before the kernel is called.
2. device pointer to the array holding the pixel colors
3. integer that holds the number of entries in the pixel array

3.1 Original 1

This kernel is used as a reference. It is much like the suggested kernel in the lecture slides. The only difference is, that it uses a loop to go over all the pixels even if the number of threads is less than the number of pixels. This change was made so that all kernels can be called using the same grid configuration. Using this kernel, `atomicAdd` will be called once per pixel.

This strategy is expected to be slow, when the number of colors is small, because many calls to `atomicAdd` will be queued up to operate on the same memory address.

3.2 Block partition 2

In this kernel, every thread block owns a shared array, called local_buckets. Similar as in the original, in a loop over all threads, color value occurrences in the input array get added to the block local local_buckets via `atomicAdd`. After this process has finished, a loop over the block has threads add the values in the local_buckets to the global output array via `atomicAdd`.

The difference to the original here is that since there are many local_buckets (one per block), contention in using `atomicAdd` is much less. Therefore, this strategy is expected to perform better than the original, regardless of the input. However, it is apparent that a high number of distinct color values in the input will shorten the execution time.

3.3 Thread-local buckets (tlr) 3

In this kernel, each thread uses a local array of color buckets, that gets filled by the same loop as in the original kernel. After filling the buckets, each thread will loop over the buckets and call `atomicAdd` once per bucket to write to the global results array. In this way the number of `atomicAdd` calls equals the product of the number of threads and the number of possible colors in the picture. E.g. for a typical grey scale image, `atomicAdd` will be called $N_{threads} \cdot 256$ times.

This strategy is expected to be faster than the reference since there are fewer calls to `atomicAdd`. This effect is expected to be stronger the fewer colors occur in an image.

3.4 Thread-local buckets with block-level reduction (tlr_blr) 4

In this kernel, each thread uses a local bucket and fills it, as it was described in section 3.3. After filling the thread local buckets, a block level shared array is used to calculate the sum over each bucket.

The sum is computed by a block level reduction strategy, that ends up holding the sum in the first entry of the array after $log_2(blockDim)$ steps. This sum is added to the global bucket by having one thread per block call `atomicAdd`.

Using this strategy, the number of calls to `atomicAdd` is $N_{Blocks} \cdot 256$ for a standard grey scale image.

This strategy is expected to be even faster than the kernel above (tlr) since there are even fewer calls to `atomicAdd`.

3.5 One color per block (linear) 5

In this kernel, every thread block looks at the whole input array and only counts the color that is equal to its block-Id. In a loop over the block, every thread sums up locally for itself the color occurrences of the color value that is equal to the block-Id. Once this process is finished, a reduction inside the block adds up the thread-local results. After that, one thread from each block writes to the output array.

This strategy works without any atomics. However, it is apparent, that at least as many blocks as color values are needed. If this kernel is called with 256 blocks for an input with only two distinct colors, 254 blocks will contribute nothing to the end result. In general, it is not expected that this strategy will yield fast execution times. It was just an idea we came up with and can now be regarded as an example for a lower bound.

4 Results

In the following we present the results of the kernel benchmarks. Every kernel evaluation was performed several times and the median execution time was taken. In addition to that, kernels also get checked for correct results by comparing to a host-calculated solution.

The performance of the _tlb and _tlb_blr kernels could be negatively affected by register spilling as every thread has its own list of buckets. For a standard gray scale image, these arrays will each require 1024 bytes of memory. This strategy may work better for smaller buckets.

Overall the block-partition kernel achieves the best results.

4.1 Benchmarking with simulated images

Benchmarks were performed using randomly generated color arrays of different size with various numbers of distinct colors.

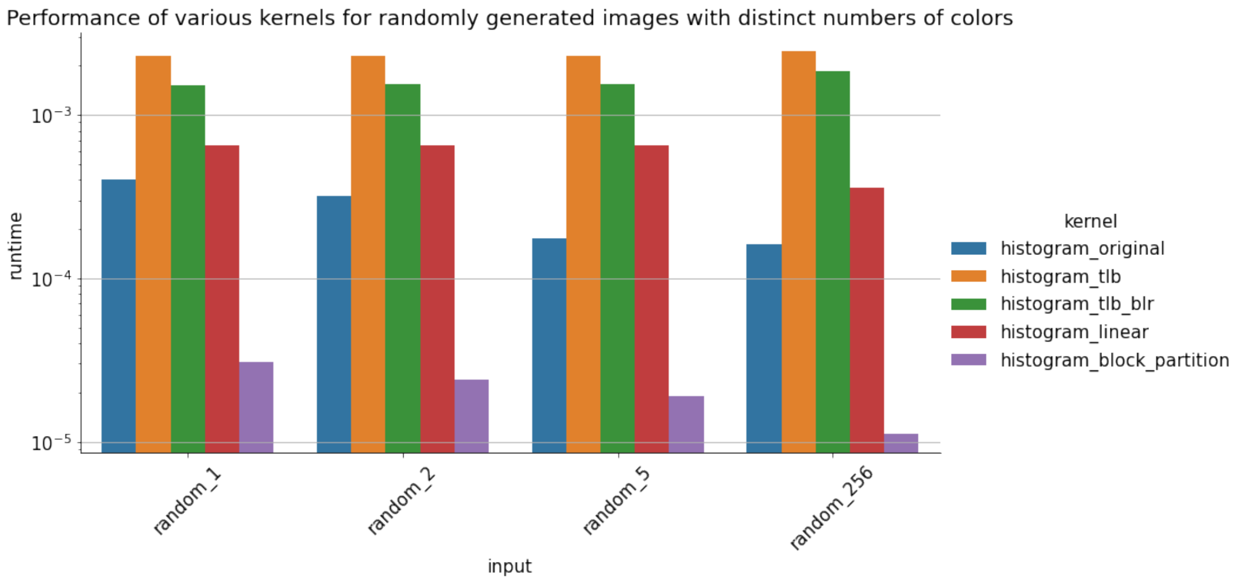

Fig. 1: Randomly populated arrays of size 640x426 were used as input. Numbers of colors per image were 1, 2, 5 and 256

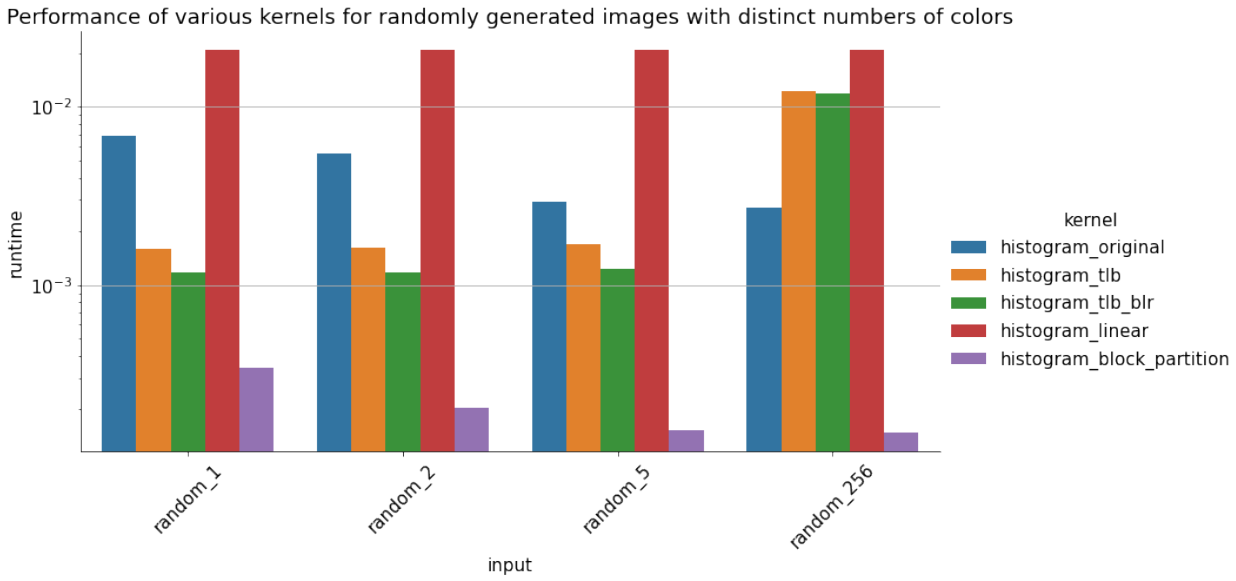

Fig. 2: Randomly populated arrays of size 3840x2160 were used as input. Numbers of colors per image were 1, 2, 5 and 256

4.2 Benchmarking with real-world grayscale images

The following benchmark was performed with a gray scale image in two different resolutions.

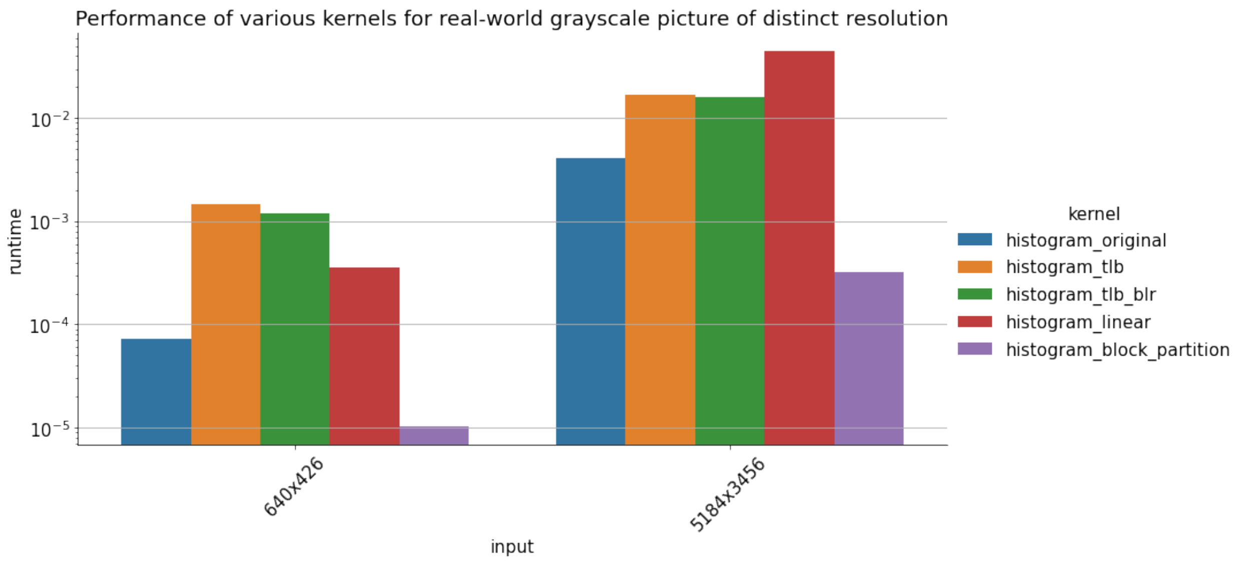

Fig. 3: Execution times for histogram kernels on real grayscale images

5 Appendix

Algorithm 1: histogram_original

```
__global__ void histogram_original(int *buckets, int *colors, size_t n_colors)
{

  size_t n_threads = blockDim.x * gridDim.x;
  size_t thread_id = blockIdx.x * blockDim.x + threadIdx.x;

  for (size_t i = thread_id; i < n_colors; i += n_threads)
  {
    int c = colors[i];
    atomicAdd(&buckets[c], 1);
  }
}
```

Algorithm 2: histogram_block_partition

```
__global__ void histogram_block_partition(int *buckets, int *colors, size_t
    n_colors)
{
  int global_idx = threadIdx.x + blockDim.x * blockIdx.x;
  int num_threads = gridDim.x * blockDim.x;

  // shared memory within the block
  __shared__ int local_buckets[RGB_COLOR_RANGE];
  for(size_t i = threadIdx.x; i < RGB_COLOR_RANGE; i+=blockDim.x)
  {
      local_buckets[i] = 0;
  }

   __syncthreads();
  for (unsigned int i = global_idx; i < n_colors; i += num_threads)
  {
    int c = colors[i];
    atomicAdd(&local_buckets[c], 1);
  }

  __syncthreads();
  {
    for(size_t i = threadIdx.x; i < RGB_COLOR_RANGE; i+=blockDim.x)
    {
      atomicAdd(&buckets[i], local_buckets[i]);
    }
  }
}
```

Algorithm 3: histogram_tlb

```
__global__ void histogram_tlb(int* buckets, int* pixels, int num_pixels) // tlb:
    thread-local buckets
//__global__ void histogram_tlb(int* pixels, int num_pixels, int* buckets) // tlb:
     thread-local buckets
{
    int global_idx = threadIdx.x + blockDim.x * blockIdx.x;
    int num_threads = blockDim.x * gridDim.x;

    int loc_buc[RGB_COLOR_RANGE];

    for(int i = 0; i < RGB_COLOR_RANGE; ++i)
        loc_buc[i] = 0;

    int c;
    for(int i = global_idx; i < num_pixels; i += num_threads){
        c = pixels[i];
        loc_buc[c]++;
    }

    for(int i = 0; i < RGB_COLOR_RANGE; ++i)
    {
        size_t index = (i+threadIdx.x)%RGB_COLOR_RANGE;
        atomicAdd(&buckets[index],loc_buc[index]);
    }
}
```

Algorithm 4: histogram_tlb_blr

```
__global__ void histogram_tlb_blr(int* buckets, int* pixels, int num_pixels)  //
    tlb_blr: thread-local buckets, block-level reduction
//__global__ void histogram_tlb_blr(int* pixels, int num_pixels, int* buckets) //
    tlb_blr: thread-local buckets, block-level reduction
{
    int global_idx = threadIdx.x + blockDim.x * blockIdx.x;
    int num_threads = gridDim.x * blockDim.x;
    int loc_buc[RGB_COLOR_RANGE];
    int c;

    for(int i = 0; i < RGB_COLOR_RANGE; ++i)
        loc_buc[i] = 0;

    // fill thread - local buckets
    for(int i = global_idx; i < num_pixels; i += num_threads){
        c = pixels[i];
        loc_buc[c]++;
    }

    // perform block level reduction for each color
    // ---------------------------------------------
    // this shared array can only hold one single color at once - so we recycle it
         once for each color (we do not have enough SM-local memory for a 256x256
        array)
    __shared__ int block_buc[BLOCK_SIZE];
    for(int col_idx = 0; col_idx < RGB_COLOR_RANGE; ++col_idx){

        // initialize shared array with thread-local resuls
        block_buc[threadIdx.x] = loc_buc[col_idx];

        // do the block-level reduction
        for(int stride = blockDim.x/2; stride>0; stride/=2){
            __syncthreads();
            if (threadIdx.x < stride)
                block_buc[threadIdx.x] += block_buc[threadIdx.x + stride];
        }

        // one atomicAdd() by the index-0-thread
        if(threadIdx.x == 0)
            atomicAdd(&buckets[col_idx], block_buc[0]);
    }
}
```

Algorithm 5: histogram_linear

```
__global__ void histogram_linear(int *buckets, int *colors, size_t n_colors)
{
  // shared memory within the block
  __shared__ int local_sums[BLOCK_SIZE];

  // thread local variable
  int sum = 0;

  for (unsigned int i = threadIdx.x; i < n_colors; i += blockDim.x)
  {
    if (colors[i] == blockIdx.x)
      sum += 1;
  }

  local_sums[threadIdx.x] = sum;

  for (unsigned int range = blockDim.x / 2; range > 0; range /= 2)
  {
    __syncthreads();
    if (threadIdx.x < range)
      local_sums[threadIdx.x] += local_sums[threadIdx.x + range];
  }

  if (threadIdx.x == 0)
    buckets[blockIdx.x] = local_sums[0];

}
```