GPU Computing

Patterns for massively parallel programming (part 2)

Histograms

Edwin Carlinet, Joseph Chazalon

`firstname.lastname@epita.fr`

Spring 2024

EPITA Research Laboratory (LRE)

Slides generated on March 26, 2024

Lab reminder

During the practice session, you ~~will have~~ had to compute the **cumulated histogram** of the image.

There are two major steps:

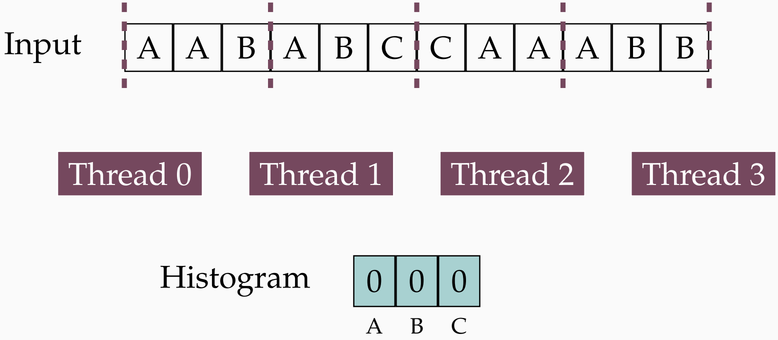
1. Compute the histogram H:
   count the number of occurrences of each value within the image.
2. Compute the cumulated histogram C:
   sum histogram values such that $C[i] = \sum_{k=0}^{i} H[k]$.

We will see how to compute those elements with **efficient parallel algorithms**.

Simple parallel histogram

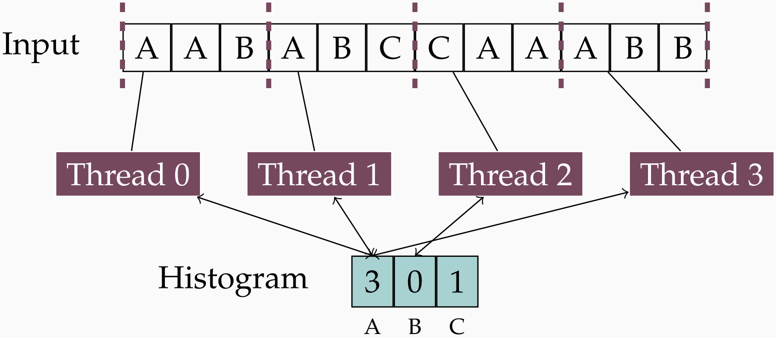
The very wrong way

A wrong approach consists in **sectioning** the input,
i.e. assigning a chunk of the input to each thread.



The very wrong way

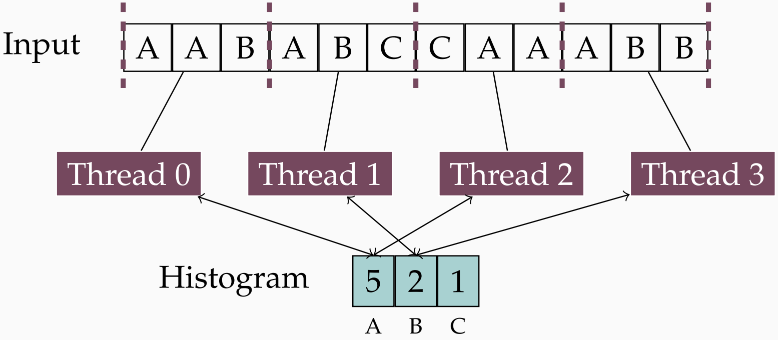
A wrong approach consists in **sectioning** the input,
i.e. assigning a chunk of the input to each thread.



What is the issue?

The very wrong way

A wrong approach consists in **sectioning** the input,
i.e. assigning a chunk of the input to each thread.



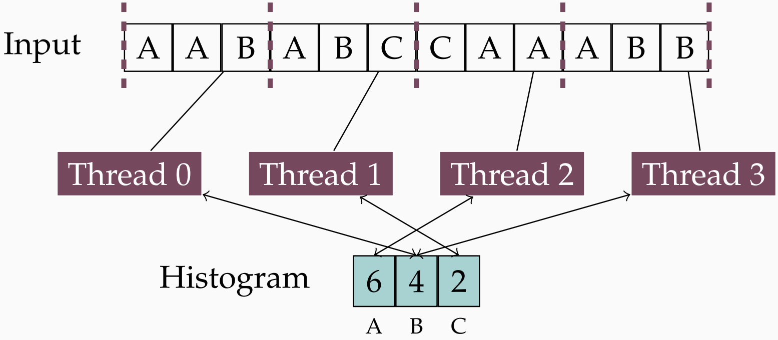
What is the issue?

Inefficient, non-coalesced memory access.

- Does not leverage cache
- Does not make use of full RAM burst

The very wrong way

A wrong approach consists in **sectioning** the input, i.e. assigning a chunk of the input to each thread.



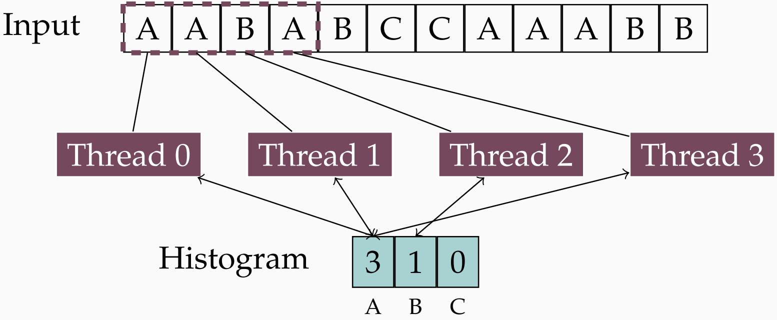
What is the issue?

Inefficient, non-coalesced memory access.

- Does not leverage cache
- Does not make use of full RAM burst

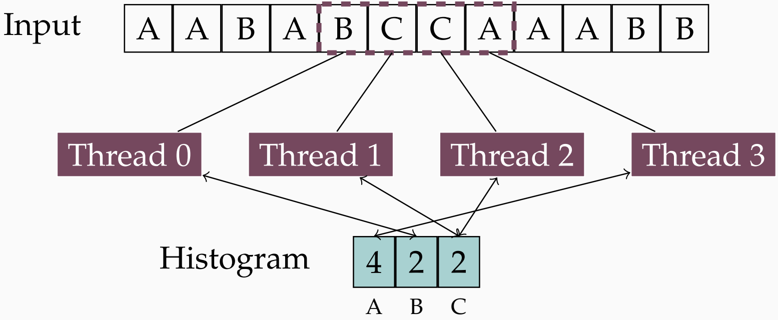
Interleaved partitioning of input

This enables coalesced memory accesses.



Interleaved partitioning of input

This enables coalesced memory accesses.



Much more efficient, coalesced memory access.

- All threads process a contiguous secion of elements
- They all move to the next section and repeat

First code sample

```
__global__ void histo(int* buf, int w, int h, int pitch, int hist_size, int* hist)
{
  int x = blockDim.x * blockIdx.x + threadIdx.x;
  int y = blockDim.y * blockIdx.y + threadIdx.y;
  if (x >= w || y >= h) return;
  int cellValue = getValue(buf, x, y, pitch);
  hist[cellValue]++;  // This is wrong!
}
```

What is the issue?

```
__global__ void histo(int* buf, int w, int h, int pitch, int hist_size, int* hist)
{
  int x = blockDim.x * blockIdx.x + threadIdx.x;
  int y = blockDim.y * blockIdx.y + threadIdx.y;
  if (x >= w || y >= h) return;
  int cellValue = getValue(buf, x, y, pitch);
  hist[cellValue]++;  // This is wrong!
}
```

What is the issue?

Data race!

A correct naive version

```
__global__ void histo(int* buf, int w, int h, int pitch, int hist_size, int* hist)
{
  int x = blockDim.x * blockIdx.x + threadIdx.x;
  int y = blockDim.y * blockIdx.y + threadIdx.y;
  if (x >= w || y >= h) return;
  int cellValue = getValue(buf, x, y, pitch);
  atomicAdd(&(hist[cellValue]), 1);
}
```

Parallel algorithm using output privatization

Reminder about *output privatization*

A simple solution called **“output privatization”** works by proceeding in two steps:

1. compute a local histogram for each block in shared memory
   cost to read and write: 1 cycle each
2. at the end of the block, flush each local histogram to global memory

Local histogram: initialization

Shared memory must be initialized.

This can be done with the “comb-like” pattern.

```
__global__ void histo(int* buf, int w, int h, int pitch, int hist_size, int* hist)
{
  extern __shared__ int localHist[];
  // linear thread id and block dim to init the 1D histogram
  int i = blockDim.x * threadIdx.y + threadIdx.x;
  int bs = blockDim.x * blockDim.y;
  for (; i < hist_size; i+=bs)
    localHist[i] = 0;
  // Wait for all block's threads before next stage
  __syncthreads();
}
```

Warning: we need synchronization after this stage.

Local histogram: computation

Like previous code, but with local atomics!

```
__global__ void histo(int* buf, int w, int h, int pitch, int hist_size, int* hist)
{
  // ...

  int x = blockDim.x * blockIdx.x + threadIdx.x;
  int y = blockDim.y * blockIdx.y + threadIdx.y;
  if (x >= w || y >= h) return;
  int cellValue = getValue(buf, x, y, pitch);
  atomicAdd(&(localHist[cellValue]), 1);
  // Wait for all block's threads before next stage
  __syncthreads();
}
```

Warning: we need synchronization after this stage.

Local histogram: commit to global memory

It is the same pattern as for the initialization.

We use a global atomic here.

```
__global__ void histo(int* buf, int w, int h, int pitch, int hist_size, int* hist)
{
  // ...

  // linear thread id and block dim to copy the 1D histogram
  int i = blockDim.x * threadIdx.y + threadIdx.x;
  int bs = blockDim.x * blockDim.y;
  for (; i < hist_size; i+=bs)
    atomicAdd(&(hist[i]), localHist[i]);
}
```

Summary

Histogram summary

Performance boosters:

- coalesced accesses
- output privatization
- (not seen here: cascading)

Requirements:

- atomics
- synchronization

Limitations:

- histograms smaller than the size of the shared memory
- overhead to allocate and initialize private copies, then commit them to global memory