Dr D. Y. Patil Institute Of Engineering,Pimpri,Pune

Mini Project On

"Run length encoding concurrently on many core GPU"

SUBMITTED BY

Practiksha Chauhan(BCOB05)

Sakshi Gujrathi(BCOB08)

Bhagyashri Chaudhari(BCOB09)

Vinod Muley(BCOB12)

UNDER GUIDANCE OF

Dr. Rachana K. Somkunwar

DEPARTMENTOF COMPUTER ENGINEERING

Academic Year 2020-21

Problem Statement: Generic Compression: Run length encoding concurrently on many core GPU

Objective: To perform Run Length Encoding using GPU

Prerequisite: CUDA and parallel processing concepts

Theory:

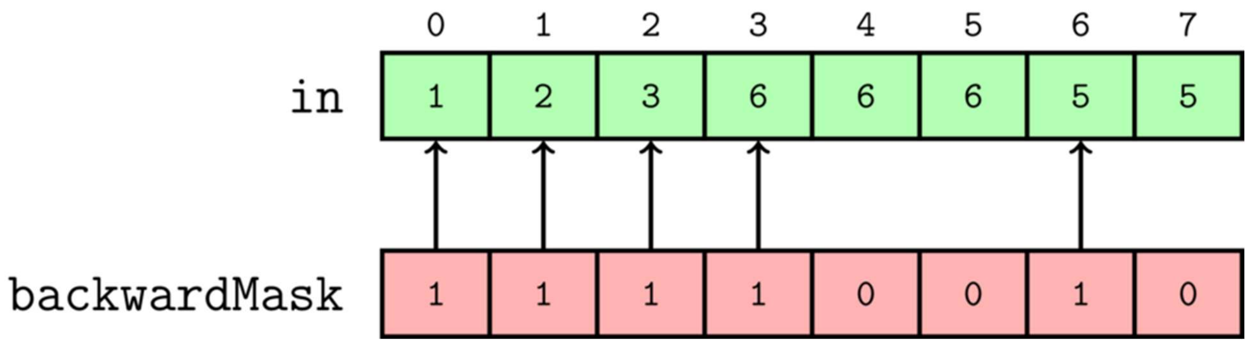
RLE is a very simple compression algorithm. Let us say we have the input data [1,2,3,6,6,6,5,5]. If we run RLE on this data, we obtain the compressed data [(1,1),(1,2),(1,3),(3,6),(2,5)]. As can be seen, by running RLE, we replace runs of repeated data by pairs on the form (x,y), where y is the symbol being repeated, and x is the number of times the symbol is repeated. So the pair (3,6) represents the data [6,6,6].

In our CUDA implementation, we have chosen to split the output pairs into two arrays. The first array contains the x:s, and the second array contains the y:s. So running PARLE on the input data [1,2,3,6,6,6,5,5] will yield the two output arrays [1,1,1,3,2] and [1,2,3,6,5].

Parallel Run-Length Encoding

So, I shall now show how we can implement RLE on the GPU. Ana calls this algorithm Parallel Run-Length Encoding, which we will from now on abbreviate PARLE. Throughout my explanation, I shall through images illustrate what happens to the input array [1,2,3,6,6,6,5,5], as we run it through the compression algorithm. Also, I shall henceforth refer to this input array to as in.

The first step of the algorithm is that we construct a mask from in. We call this mask the backwardMask, and it is constructed as follows: For every element in[i], we assign a thread, and that thread compares the current and the previous elements for equality. If they are not equal, backwardMask[i] will be 1, and otherwise, it is 0. However, the first element does not have a previous element, so, for reasons that will soon become clear, we always set backwardMask[0] to 1. The construction of the mask is also illustrated through the below image.



The numbers over in are the ids of the threads assigned to every element. To give an example, thread number 6 will do the following: It will observe that in[6] == 5, and in[5] == 6, so it will set backwardMask[6] = 1.

From the above illustration, we can draw a simple conclusion; backwardMask encodes the beginning of all the runs. So if backwardMask[i]==1, that means that at i a new repeated runs of characters begin. Now it should be obvious why we said that backwardMask[0] == 1.

For serial algorithms, such the RLE implemented on the CPU above, knowing where to output the next element is often easy; you just output to the next free spot in the array. However, doing the same thing on the GPU is considerably more difficult, because we have lots of threads working at the same time, and we have no idea in which order they finish. As a result, finding the next free spot in the array is often impossible. Instead what you often do, is that you basically twiddle with prefix sums until you get what you want.

Possible Further Optimization

As stated above, when we profiled PARLE in NVidia Visual Profiler, we found that the algorithm is heavily bottlenecked by memory; the arithmetic intensity is very low, so the execution time dominated by the time it takes to read from the global memory. So if you want to implement a faster version of PARLE, you will need to minimize your reads and writes to the global memory. And one way of doing this is, like Ana did, to use shared memory.

Program:

```
#include <iostream>
#include <stdio.h>
#include <unistd.h>
#include <vector>
#include <cstring>
#include <string>
#include <time.h>
#include "hemi/hemi.h"
#include "hemi/kernel.h"
#include "hemi/parallel_for.h"
#include "hemi/launch.h"
#include "cub/util_allocator.cuh"
#include "cub/device/device_scan.cuh"
#include "cub/device/device_run_length_encode.cuh"
using in_elt_t = int;

float parallel_elapsed_time = 0.0;
cudaEvent_t gpu_start, gpu_stop;

template<typename elt_t>
struct array
{
	elt_t *data;
	size_t size; // the number of elt_t elements in data
```

```
static array<elt_t> new_on_device(size_t size)
{
    array<elt_t> d_result{nullptr, size};
    d_result.cudaMalloc();
    return d_result;
}

static array<elt_t> vector_view_on_host(std::vector<elt_t> &v)
{
    return array<elt_t>{v.data(), v.size()};
}

array<elt_t> subview(size_t offset, size_t subview_size)
{
    size_t result_size = std::min(subview_size, size - offset);
    return array<elt_t>{data + offset, result_size};
}

elt_t &operator[](const size_t i)
{
    return data[i];
}

void cudaMalloc()
{
    checkCuda(::cudaMalloc(&data, size * sizeof(*data)));
}
```

```
	void cudaFree()
	{
		checkCuda(::cudaFree(data));
	}
};

void append_partial_result(std::vector<in_elt_t> &out_symbols, std::vector<int> &out_counts,
std::vector<in_elt_t> &full_out_symbols, std::vector<int> &full_out_counts) {
	size_t offset = 0;

	if (full_out_symbols.size() > 0 && out_symbols.size() > 0) {
		size_t prev_full_end{full_out_symbols.size() - 1};
		if (full_out_symbols[prev_full_end] == out_symbols[0]) {
			full_out_counts[prev_full_end] += out_counts[0];
			offset = 1;
		}
	}

	std::copy(out_symbols.begin() + offset, out_symbols.end(),
std::back_inserter(full_out_symbols));
	std::copy(out_counts.begin() + offset, out_counts.end(),
std::back_inserter(full_out_counts));
}

int serial_rle_helper(const in_elt_t* in, int n, in_elt_t* symbolsOut, int* countsOut) {
	if (n == 0)
		return 0; // nothing to compress!
```

```
int outIndex = 0;
in_elt_t symbol = in[0];
int count = 1;

for (int i = 1; i < n; ++i) {
        if (in[i] != symbol) {
                // run is over.
                // So output run.
                symbolsOut[outIndex] = symbol;
                countsOut[outIndex] = count;
                outIndex++;

                // and start new run:
                symbol = in[i];
                count = 1;
        } else {
                ++count; // run is not over yet.
        }
}

// output last run.
symbolsOut[outIndex] = symbol;
countsOut[outIndex] = count;
outIndex++;

return outIndex;
```

```
}

void serial_rle(array<in_elt_t> in, std::vector<in_elt_t> &out_symbols, std::vector<int>
&out_counts, int &out_end) {
        out_end = serial_rle_helper(in.data, in.size, out_symbols.data(), out_counts.data());
}

void inclusive_prefix_sum(array<uint8_t> d_in, array<int> d_out) {
   cub::CachingDeviceAllocator allocator(true);

   void *d_temp_storage = nullptr;
   size_t temp_storage_bytes = 0;
   // Estimate temp_storage_bytes
   checkCuda(cub::DeviceScan::InclusiveSum(d_temp_storage, temp_storage_bytes, d_in.data,
d_out.data, d_in.size));
   checkCuda(allocator.DeviceAllocate(&d_temp_storage, temp_storage_bytes));
   checkCuda(cub::DeviceScan::InclusiveSum(d_temp_storage, temp_storage_bytes, d_in.data,
d_out.data, d_in.size));
}

void paralel_rle_helper(array<in_elt_t> d_in, array<in_elt_t> d_out_symbols, array<int>
d_out_counts, array<int> d_end) {
        auto d_backward_mask = array<uint8_t>::new_on_device(d_in.size);
        auto d_scanned_backward_mask = array<int>::new_on_device(d_in.size);
        auto d_compacted_backward_mask = array<int>::new_on_device(d_in.size + 1);

        hemi::parallel_for(0, d_backward_mask.size, [=] HEMI_LAMBDA(size_t i) {
                if (i == 0) {
                        d_backward_mask.data[i] = 1;
```

```
                return;
            }
            d_backward_mask.data[i] = d_in.data[i] != d_in.data[i - 1];
    });
    inclusive_prefix_sum(d_backward_mask, d_scanned_backward_mask);
    hemi::parallel_for(0, d_in.size, [=] HEMI_LAMBDA(size_t i) {
            if (i == 0) {
                d_compacted_backward_mask.data[i] = 0;
                return;
            }
            size_t out_pos = d_scanned_backward_mask.data[i] - 1;
            if (i == d_in.size - 1) {
                *d_end.data = out_pos + 1;
                d_compacted_backward_mask.data[out_pos + 1] = i + 1;
            }
            if (d_backward_mask.data[i])
                d_compacted_backward_mask.data[out_pos] = i;
    });

    // Not hemi::parallel_for because d_end is only on the device now.
    hemi::launch([=] HEMI_LAMBDA() {
            for (size_t i: hemi::grid_stride_range(0, *d_end.data)) {
                int current = d_compacted_backward_mask.data[i];
                int right = d_compacted_backward_mask.data[i + 1];
                d_out_counts.data[i] = right - current;
                d_out_symbols.data[i] = d_in.data[current];
            }
```

```
	});
	hemi::deviceSynchronize();

	d_compacted_backward_mask.cudaFree();
	d_scanned_backward_mask.cudaFree();
	d_backward_mask.cudaFree();
}

void parallel_rle(array<in_elt_t> in, std::vector<in_elt_t> &out_symbols, std::vector<int>
&out_counts, int &out_end) {
	auto d_in = array<in_elt_t>::new_on_device(in.size);
	auto d_out_symbols = array<in_elt_t>::new_on_device(in.size);
	auto d_out_counts = array<int>::new_on_device(in.size);
	auto d_end = array<int>::new_on_device(1);

	checkCuda(cudaMemcpy(d_in.data, in.data, d_in.size * sizeof(*d_in.data),
cudaMemcpyHostToDevice));

	paralel_rle_helper(d_in, d_out_symbols, d_out_counts, d_end);

	checkCuda(cudaMemcpy(out_symbols.data(), d_out_symbols.data, out_symbols.size() *
sizeof(*out_symbols.data()), cudaMemcpyDeviceToHost));
	checkCuda(cudaMemcpy(out_counts.data(), d_out_counts.data, out_counts.size() *
sizeof(*out_counts.data()), cudaMemcpyDeviceToHost));
	checkCuda(cudaMemcpy(&out_end, d_end.data, sizeof(out_end),
cudaMemcpyDeviceToHost));

	d_in.cudaFree();
	d_out_symbols.cudaFree();
```

```
    d_out_counts.cudaFree();
    d_end.cudaFree();
}

void run_rle_impl(array<in_elt_t> in, std::vector<in_elt_t> &out_symbols, std::vector<int>
&out_counts, int &out_end, bool use_cpu_impl) {
    if (use_cpu_impl)
        serial_rle(in, out_symbols, out_counts, out_end);
  else
    parallel_rle(in, out_symbols, out_counts, out_end);
}

void rle(std::vector<in_elt_t> &in_owner, std::vector<in_elt_t> &full_out_symbols,
std::vector<int> &full_out_counts, size_t piece_size, bool use_cpu_impl, bool verbose) {
    array<in_elt_t> full_in = array<in_elt_t>::vector_view_on_host(in_owner);

    for (size_t start = 0; start < in_owner.size(); start += piece_size) {
        array<in_elt_t> in = full_in.subview(start, piece_size);

        if(verbose)
            std::cout << "Partial in start: " << start << ", size: " << in.size << std::endl;

        // TODO Could actually be allocated once
        std::vector<in_elt_t> out_symbols(in.size);
        std::vector<int> out_counts(in.size);
        int end{0};

        run_rle_impl(in, out_symbols, out_counts, end, use_cpu_impl);
```

```
            out_symbols.resize(end);
            out_counts.resize(end);

            append_partial_result(out_symbols, out_counts, full_out_symbols,
full_out_counts);
      }
}

void parse_input_args(int argc, char* argv[], size_t *input_size) {
   if(argc > 1) {
      *input_size = atoi(argv[1]);
   }
}

int get_single_digit_rand() {
      int singleDigit = rand() % 10;
      return singleDigit;
}

std::vector<in_elt_t> generate_input(size_t size) {
   std::vector<in_elt_t> res{};
   int multiplier = get_single_digit_rand();
   int value = get_single_digit_rand();
   for(int i = 0 ; i < size ; i++) {
      if(multiplier == 0) {
         multiplier = get_single_digit_rand();
         value = get_single_digit_rand();
```

```
        }
        res.push_back(value);
        multiplier--;
    }
    return res;
}

int main(int argc, char *argv[]) {
    srand(time(0));
    size_t input_size = 10000; //default input size
    size_t input_piece_size = 4;
    bool verbose = false;

    parse_input_args(argc, argv, &input_size);

    std::cout<<"Generating Input..."<<std::endl;
    std::vector<in_elt_t> input = generate_input(input_size);

    std::cout<<"Initial Input: "<<std::endl;
    std::cout<<"[";
    for(int i = 0 ; i < input.size() ; i++) {
        std::cout<<input[i]<<" ";
    }
    std::cout<<"]";
    std::cout<<std::endl;

    std::cout<<"Using the CPU implementation (Serial RLE Version)"<<std::endl;
```

```
    std::vector<in_elt_t> out_symbols{};
            std::vector<int> out_counts{};

    rle(input, out_symbols, out_counts, input_piece_size, true, verbose);

    std::cout<<"Output Symbols: "<<std::endl;
    std::cout<<"[";
    for(int i = 0 ; i < out_symbols.size() ; i++) {
        std::cout<<out_symbols[i]<<" ";
    }
    std::cout<<"]";
    std::cout<<std::endl;
    std::cout<<"Count: "<<std::endl;
    std::cout<<"[";
    for(int i = 0 ; i < out_counts.size() ; i++) {
        std::cout<<out_counts[i]<<" ";
    }
    std::cout<<"]";

    std::cout<<std::endl;

std::cout<<"===================================================================
=========="<<std::endl;

            std::cout<<"Using the GPU implementation (Parallel RLE Version)"<<std::endl;
            out_symbols.clear();
            out_counts.clear();
```

```
        rle(input, out_symbols, out_counts, input_piece_size, false, verbose);

    std::cout<<"Output Symbols: "<<std::endl;
    std::cout<<"[";
    for(int i = 0 ; i < out_symbols.size() ; i++) {
        std::cout<<out_symbols[i]<<" ";
    }
    std::cout<<"]";
    std::cout<<std::endl;
    std::cout<<"Count: "<<std::endl;
    std::cout<<"[";
    for(int i = 0 ; i < out_counts.size() ; i++) {
        std::cout<<out_counts[i]<<" ";
    }
    std::cout<<"]";
    std::cout<<std::endl;
    return 0;
}
```

Output:

```
1 !./a.out 1000

Generating Input...
Initial Input:
[8 8 8 8 8 9 9 9 9 9 9 3 3 3 3 3 3 3 3 3 9 9 9 9 9 2 2 2 3 3 3 3 1 1 1 1 1 1 1 1 8 8 8 8 8 8 8 2 9 9 9 9 9 9 9 9
Using the CPU implementation (Serial RLE Version)
Output Symbols:
[8 9 3 9 2 3 1 8 2 9 ]
Count:
[5 6 9 5 3 4 8 7 1 952 ]
=====================================================================
Using the GPU implementation (Parallel RLE Version)
Output Symbols:
[8 9 3 9 2 3 1 8 2 9 ]
Count:
[5 6 9 5 3 4 8 7 1 952 ]
```

```
1 !./a.out 30

Generating Input...
Initial Input:
[5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 ]
Using the CPU implementation (Serial RLE Version)
Output Symbols:
[5 6 4 ]
Count:
[7 8 15 ]
=====================================================================
Using the GPU implementation (Parallel RLE Version)
Output Symbols:
[5 6 4 ]
Count:
[7 8 15 ]
```