Lab Report
Project2
Xuan Hu, Ye Huang

I. Introduction

There is an input file with 24576 elements, each of which has 6*6*6 points inside. Inside each point there are 5 parameters. Our job is to read the last one of the five parameter, executing prediction, quantization, and entropy encoding on these parameters, finally obtaining a compressed data. The purpose of this experiment was to implement the algorithm both in CPU and in GPU.

II. Procedure

A. CPU architecture

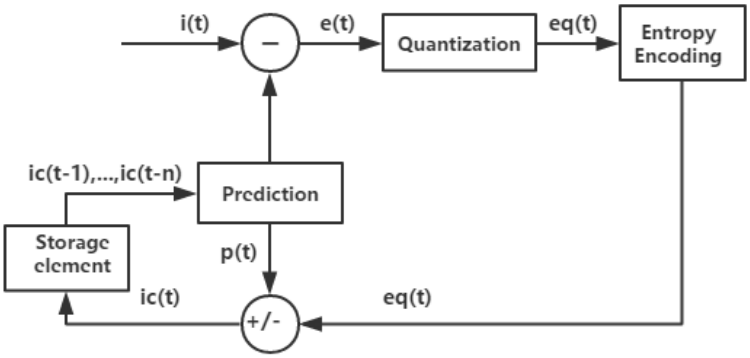

Figure1. Algorithm

In the CPU implementation, we have 10 files; each file has 24576 elements, each element has 6*6*6 points, so the total number of element we have to process is 24576*6*6*6=5,308,416. These elements are encoded in sequence, we have to execute 5,308,416 times.

$eq\ (t)$ =round $(e\ (t)/Q\_FA\ )$ Quantization Q_FACTOR=100

$p\ (t)$ =2 $\times i\_c\ (t-1)\ -i\_c\ (t-2)$

B. How to exploit the parallelism

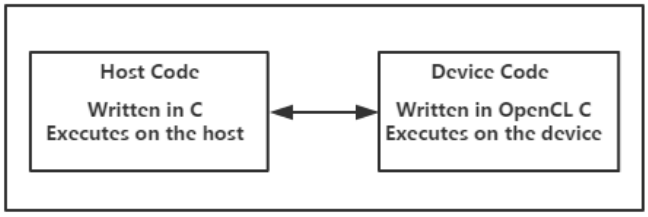

Figure2. Relationship between host code and device code

In CPU, we have to execute the same algorithm 5, 308,416 times sequentially which needs a lot of time. To exploit the parallelism, the basic idea is data-level parallelism, in which multiple processors execute the same instructions on different pieces of data. It allows flow control computation to be shared amongst processors and thus allows more of the hardware to be

devoted to instruction execution.

In the GPU implementation, we have a host code and a device code. Host code sends commands to the Devices to transfer data between host memory and device memories and to execute device code. Serial code executes in a Host (CPU) thread, Parallel code executes in many Device (GPU) threads across multiple processing elements

Firstly, we decompose task into workitems, workitems are grouped into local workgroups. Kernels are executed across a global domain of work-items. Each kernel compute unit can run multiple simultaneous workgroups; workitems grouped in a workgroup are executed in parallel as well. In this way, computation has been parallelized by distributing the data amongst computing nodes.

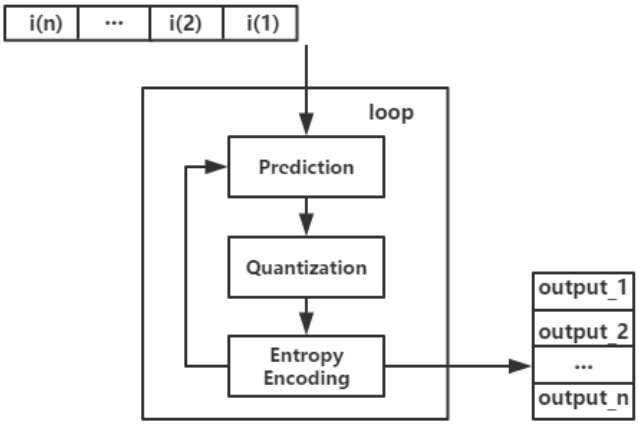

Figure3. CPU execution data flow

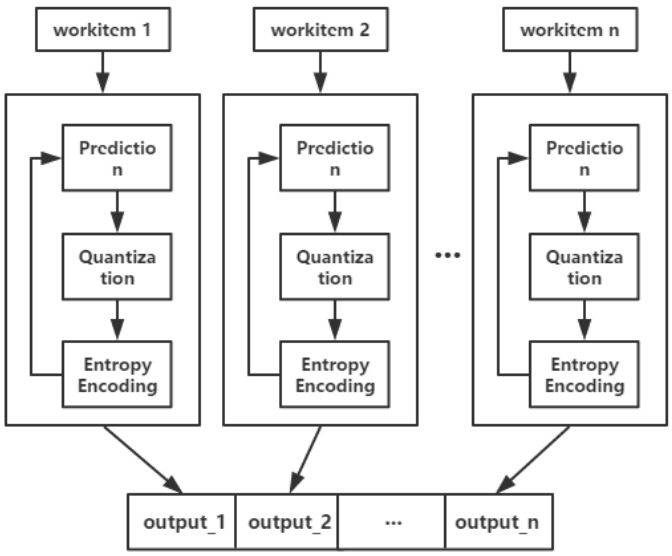

Figure4. GPU execution data flow

Figure3 and figure 4 shows the comparison between the CPU and GPU. In CPU, elements are processed in sequence using a loop. Once is processed, the result is put into an output array,

while in GPU, the workitems are processing in parallel, Kernels represent the computation for 1 work-item. It use the same algorithm as in CPU, the difference is the kernel need no loop, it only have to complete the algorithm only once.

C. GPU architecture

In this experiment, In GPU, we divided the data into 82944 workgroups, each of which contains 64 workitems. Each workitem implement the computation of one element, the execution of the workitem is paralleled so that we can compute the 5,308,416 elements at the same time and then put them in to the output array. The architecture is showed in the following.

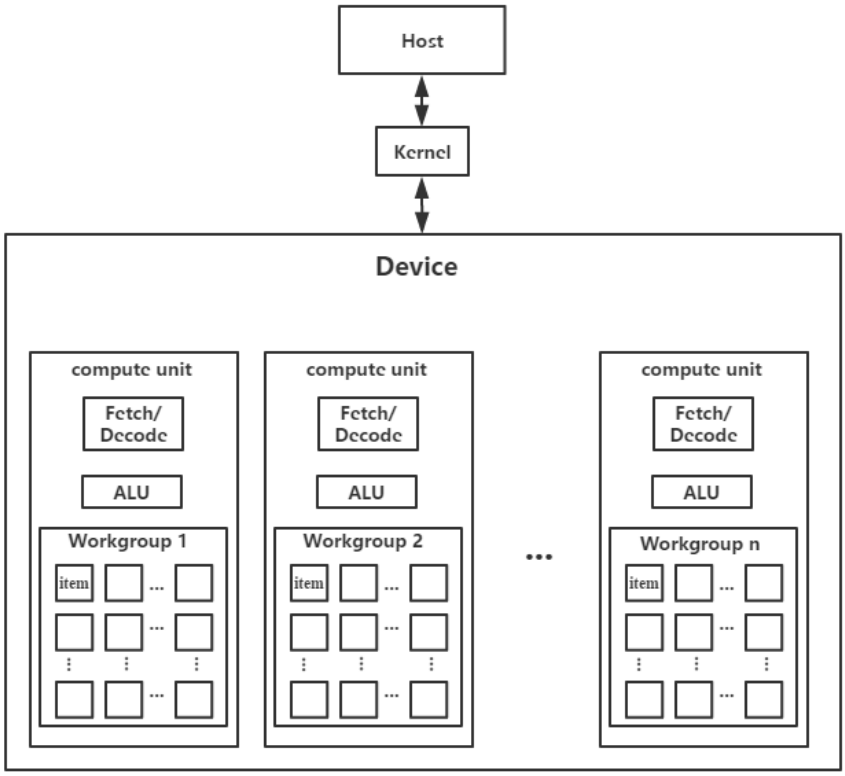

Figure5. GPU architecture

D. Design idea

In the design, we pre-define 4 functions.

1) put_bit

1) Function: The golombcode is variable length coding, which means that only after finishing pushing the first output into the memory, then we can get the start point of second output in memory. When we try to implement parallel computation, we allocate a reasonable space for each output golombcode, which is 2 bytes. And after we get this mid-output, we try to use this *put_bit* function to attach the bits together.

In GPU, because of the parallel computation, the time needed to produce each output is different. If we design another kernel to put the results together, we still don't know the starting point to write an output into the sequence array until the writing of the previous all output are completed. And this additional kernel can only be a sequential loop to attach bits one by one.

Input: unsigned char b (0 or1)

Output: out_buf (an array with element of 8bit)

2) golombcode

Function: implement the golombcode. In the algorithm we set $m=2^k$.

If the output_error needed to be encoded is very large. a long bit string is needed for the result of its golombcode. so in the case that the number of the result bit string is larger than 16, we set 16 bit escape symbols as 0xffff and use the next 32 bit to store the result.

Input: const int x(the error value sended into encoding), int k, int m, const size_t index

Output: return unsigned int output

3) compressionHost

Function: implement the prediction and quantization.

After the prediction, the output_error may be negative, while the golombcode are only suitable for non-negative integers. This will cause information loss. A solution is to process the output_error before encode. If the output_error is positive, output_error = 2*output_error. If the output_error is negative, output_error = | 2*output_error | −1

Input: buffer_p1, buffer_p2, buffer_p3

Output: h_output

4) readata

Function: read the HDF5 files

III. RESULT

TABLE 1

SPEED UP AND COMPRESSION RATIO WITH RESPECT TO THE DIFFERENT QUANTIZATION FACTORS

For k=4, q_factor = 100	FOR K=8, Q_FACTOR = 10	For k=11, q_factor = 1
Context has 1 devices	Context has 1 devices	Context has 1 devices
Running on Hawaii	Running on Hawaii	Running on Hawaii
ReadHDF5Finish	ReadHDF5Finish	ReadHDF5Finish
CPU compute start	CPU compute start	CPU compute start
overflow rate =0.00630131	overflow rate =0.0116298	overflow rate =0.0288732
CPU compute finished	CPU compute finished	CPU compute finished
GPU Computing start	GPU Computing start	GPU Computing start
GPU Computing Finished	GPU Computing Finished	GPU Computing Finished
READ HDF5 Time: 24.802451s	READ HDF5 Time: 8.734907s	READ HDF5 Time: 9.111125s
CPU Time: 0.144077s	CPU Time: 0.135695s	CPU Time: 0.129754s
Memory copy Time: 0.013036s	Memory copy Time: 0.012922s	Memory copy Time: 0.012277s
GPU Time w/o memory copy:	GPU Time w/o memory copy:	GPU Time w/o memory copy:
0.000516s (speedup = 279.219)	0.000463s (speedup = 293.078)	0.000455s (speedup = 285.174)
GPU Time with memory copy:	GPU Time with memory copy:	GPU Time with memory copy:
0.013552s (speedup = 10.6314)	0.013385s (speedup = 10.1378)	0.012732s (speedup = 10.1912)
compressed_size =4192223	compressed_size =6488917	compressed_size =8873029
compression_ratio =10.13	compression_ratio =6.54459	compression_ratio =4.78611
Success	Success	Success

IV. DISCUSS

A. Effect of number of work groups and number work items

Use of the work-groups allows more optimization for the kernel compilers. This is because data is not transferred between work-groups. Depending on used OpenCL device, there might be caches that can be used for local variables to result faster data accesses. If there is only one work-group, local variables would be just the same as global variables which would lead to slower data accesses. Also, usually OpenCL devices use Single Instruction Multiple Data (SIMD) extensions to achieve good parallelism. One work group can be run in parallel with SIMD extensions.

GPUs perform better when they have work-group sizes that are multiples of their SIMD architecture, usually 64, 48, or 32 depending on vendor and model. If you were to use less than those sizes, you would waste clock cycles by underutilizing the GPU's native instruction width. However, an FPGA is an efficiently pipeline architecture so that you have enough work to fill your pipeline, you won't be wasting clock cycles. Also, if you use barriers in your kernel, smaller work-group sizes may yield better results as the latency between the first and last work-item within the same work-group would be lower. There may also be other factors at play with your memory fetch/store instructions,

By taking some experiments to find the optimal balance by trying different work-group sizes. We found that different workgroup sizes affect little on the final result.

B. Resources of the GPU

Knowing how and when to use each type of memory goes a long way towards optimizing the performance of your application. There are different kinds of resources of GPU, Local, Global, Constant, and Texture memory. Data stored in local memory is visible only to the thread that wrote it and lasts only for the lifetime of that thread. Data stored in global memory is visible to all threads within the application, and lasts for the duration of the host allocation.

In the experiment, we made use of shared memory. Because the scope of local memory is local to the thread and it resides off-chip, which makes it as expensive to access as global memory. While shared memory allows communication between threads within the same block which can make optimizing code much easier for beginner to intermediate programmers. It is the most versatile and easy-to-use type of memory.