On GPU Acceleration of the Vector Quantization Image Compression Algorithm

Kris Manohar, Patrick Hosein, Duc Kieu and Alana Sankar
The University of the West Indies, St. Augustine, Trinidad
{kris.manohar, patrick.hosein, duc.kieu, alana.sankar}@uwi.edu

***Abstract*—Historically, the Vector Quantization (VQ) image compression algorithm was designed for single-core processors. Despite its simplicity, impressive bit rates, and good reconstructed image quality, the VQ algorithm is limited by a runtime complexity of $\mathcal{O}(N)$ where N is the size of the main codebook. As each input pixel block can be processed independently, the traditional VQ algorithm does not fully exploit modern multi-core architectures. We propose novel GPU-accelerated implementation for both the VQ encoder and decoder that exploits multi-core architectures. We present the design of CUDA kernels for distributing the computations across multiple GPU threads. Although these kernels do not fundamentally change the theoretical runtime complexity of encoder and decoder, it does reduce their respective constant factors which yeilds significantly lower execution times. Specifically the VQ encoder is improved by 61x, 44x, 34x and 29x for main codebook sizes $N = 128, 256, 512,$ and 1024 while the GPU accelerated VQ decoder is at least 200x faster across all codebook sizes. These improvements make VQ algorithms more practical for real-time applications such as multi-media streaming and reversible data hiding.**

***Index Terms*—CUDA, GPU acceleration, image compression, Vector Quantization, Side Match Vector Quantization.**

I. Introduction

Today, the internet is ubiquitous. As a communication channel, large amounts of data are generated, stored and transmitted daily. Over the years, computing hardware has evolved to keep pace with the increasing need to process large volumes of digital data [1]. Over the past few decades the widespread adoption of general-purpose multi-core processors has introduced parallel computing into most (if not all) modern consumer devices such as desktops and laptops [2].

General-purpose multi-core processors have penetrated several industries from signal processing, embedded devices, data bases, and graphics processing [3]. In 1999, NVIDIA released the GeForce 256 graphics processing unit (GPU). This GPU was the first dedicated processor for real-time graphics. Researchers developed strategies for exploiting these early programmable GPUs for executing computationally intensive scientific applications. As the GPUs evolved, so did the number of available programmable cores [4]. Today, GPUs have found utility in accelerating, simulations, operations research, database, data science, large language models, artificial intelligence, and image processing algorithms and applications [5]–[7].

Vector Quantization (VQ) [8] and Side Match Vector Quantization (SMVQ) [9] are two popular image compression techniques. These techniques are desirable due to their simplicity, impressive bit rates, and good reconstructed image visual quality. These properties make these algorithms ideal candidates for reversible data hiding (RDH) techniques. These techniques focus on concealing secret communications within digital objects (such as texts, videos, audios, images etc.) so as not to arouse suspicion [10]–[15].

A common characteristic of RDH based on the SMVQ compression algorithm is its long execution times [1], [16], [18], [19]. For each input pixel block X, the SMVQ algorithm must sort the N codewords of the main codebook D. This step introduces a time complexity of $\mathcal{O}(Nlog_2N)$ per input pixel block. Compared to VQ the time complexity per input pixel block is $\mathcal{O}(N)$ since only searching for the most similar codeword to X is required. In 2022, Manohar and Kieu [20] demonstrated how the SMVQ time complexity could become linear by defining an order preserving map between the side match distortions and an integer sequence. Such a transformation allows the SMVQ algorithm to exploit integer sorting algorithms with a linear time complexity.

Both the VQ and SMVQ algorithms were designed when single-core processors were the dominant architecture which is not optimal in today's multi-core environment. To the best of our knowledge no work has been done on parallelizing these algorithms. In this paper we proposed a GPU accelerated implementation of the VQ image compression algorithm. Specifically, we define Compute Unified Device Architecture (CUDA) kernels for the VQ's encoder and decoder which are exploited to distribute the computations of across multiple processors on an NVIDIA GPU. Our experimental results confirm speeds ups of 61x, 44x, 34x and 29x for main codebook sizes $N = 128, 256, 512,$ and 1024 for the VQ encoder and at least 200x across all codebook sizes for the VQ decoder. The remainder of this paper is organized as follows. Section 2 discusses the related works, Section 3 presents the proposed GPU accelerated algorithms for the VQ encoder and decoder and Section 4 documents the experimental results and discussions. Finally, section 5 summarizes the contributions of this paper.

II. Related Work

A. Vector Quantization

The Vector Quantization (VQ) algorithm is a lossy data compression algorithm proposed by Gray in 1984 [8]. This

image compression technique has three components: the codebook generator, the VQ encoder, and the VQ decoder. The codebook generator utilizes a clustering algorithm such as the LBG algorithm [21] to derive a main codebook D. This main codebook contains N, K-dimensional codewords CW_f's, where $CW_f = (cw_{f,1}, cw_{f,2}, ..., cw_{f,K})$ and $K = p \times q$. Given an non-overlapping input pixel block $X = (x_1, x_2, ..., x_q, x_{q+1}, x_{q+2}, ..., x_{2q}, ..., x_{(p-1)q+1}, \ldots, x_K)$ sized $p \times q$, the VQ encoder searches D for the best matched codeword CW_x and replaces X with x in a VQ index table, VIT sized $\frac{H}{p} \times \frac{W}{q}$. Specifically, the codeword CW_x has the smallest Euclidean distance from X. The Euclidean distance between X and CW_f is defined by:

$$ED(X, CW_f) = \sqrt{\sum_{i=1}^{p}\sum_{j=1}^{q}\left(X_{i,j} - cw_{f,q(i-1)+j}\right)^2} \quad (1)$$

where $X_{i,j}$ is the jth component in the ith row of the pixel block X, CW_f is the fth codeword in D, and $cw_{f,z}$ is the zth component of CW_f with $z = q \times (i-1) + j$.

Repeating this process an collecting the VQ indices x's over all non-overlapping input pixel blocks X, in raster scan order, generates the VIT. The VQ decoder reconstructs the compressed image. It replaces each VQ index x with the codeword CW_x from D. The complete VQ encoder is presented in Algorithm 1.

Algorithm 1 VQ Algorithm Encoder

Input: A grayscale cover image G size $H \times W$, pixel block size $p \times q$, codebook D sized $N \times K$, where $K = p \times q$
Output: VQ index table VIT sized $\frac{H}{p} \times \frac{W}{q}$
Create output VQ index table VIT sized $\frac{H}{p} \times \frac{W}{q}$
Divide G into non-overlapping pixel blocks X's sized $p \times q$
for $r = 1$ to $\frac{H}{p}$ **do**
  for $c = 1$ to $\frac{W}{q}$ **do**
    $min\_cw\_dist \leftarrow \infty$
    $x \leftarrow 0$
    for $f = 1$ to N **do**
      $X \leftarrow$ pixel block at row r column c
      $dist \leftarrow ED(X, CW_f)$
      if $dist < min\_cw\_dist$ **then**
        $min\_cw\_dist \leftarrow dist$
        $x \leftarrow f$
      end if
    end for
    $VIT[r, c] \leftarrow x$
  end for
end for

The VQ decoder uses VQ index table VIT sized $\frac{H}{p} \times \frac{W}{q}$ to generate reconstructed input image G' sized $H \times W$. It expands each VQ index into a reconstructed pixel block where the pixel values are the components of the corresponding codeword. Specifically, the decoder reads VQ index x located a row r and column c and uses the components of codeword CW_x to reconstruct pixels $G'_{i,j} = cw_{x,k}$ where $k = 1, \ldots, K$ and $i = (r-1)p + \lceil \frac{k}{p} \rceil$ and $j = (c-1)q + \mod (q+1)$. The completed VQ decoder is presented in Algorithm 2.

Algorithm 2 VQ Algorithm Decoder

Input: Pixel block size $p \times q$, codebook D sized $N \times K$, where $K = p \times q$ and VQ index table VIT sized $\frac{H}{p} \times \frac{W}{q}$
Output: Reconstructed grayscale cover image G' sized $H \times W$
Create and initialized an output image G' sized $H \times W$
for $r = 1$ to $\frac{H}{p}$ **do**
  for $c = 1$ to $\frac{W}{q}$ **do**
    $x \leftarrow VIT[r, c]$
    for $k = 1$ to K **do** ▷ Reconstruct X' with CW_x
      $i \leftarrow (r-1)p + \lceil \frac{k}{p} \rceil$
      $j \leftarrow (c-1)q + k \mod (q+1)$
      $G'[i, j] \leftarrow CW_x[k]$
    end for
  end for
end for

III. Proposed Scheme

Traditionally, the Vector Quantization (VQ) algorithm compresses the input image's pixel blocks in raster scan order. Single-core machines were the dominant computational model when Grey developed the VQ which made the raster scan order optimal as only one input pixel block could be processed at time. However, with the prevalence of multi-core CPUs and GPUs, we can exploit these additional cores to process several input pixel blocks in parallel. In this section we propose the implementation of a CUDA VQ kernel which compresses single input pixel block. By executing this kernel on multiple GPU threads, we can now compress many input pixel blocks in parallel which significantly speeds up compression time. The following subsections discuss these details and propose GPU-accelerated VQ compression algorithm.

A. GPU Accelerated VQ Encoder

Given an non-overlapping input pixel block X, sized $p \times q$ from the input image G sized $H \times W$, our VQ-kernel computes the position of the best-matched codeword x from the main codebook D. This operation is independent for each X. Therefore, theoretically, the VQ compression of all $M = \frac{H}{p} \times \frac{W}{q}$ pixel blocks can occur simultaneously. The limiting factor for this approach would be the number of parallel processing threads T available on the GPU. Since each thread would execute the VQ-kernel, which performs the VQ compression independently, the maximum speed up is M. The python code for the proposed VQ-kernel is presented in Algorithm 3.

The VQ-kernel accepts matrices d_I, d_{CB} and array d_{VIT}. The ith non-overlapping input pixel block X of input image G, in raster scan order corresponds to the ith column of the matrix d_I sized $K \times M$, where $K = p \times q$. Similarly the jth codeword of the main codebook D, which contains N,

Algorithm 3 Python Code for the proposed VQ-Kernel

```
from numba import cuda
@cuda.jit()
def vq_kernel(d_I, d_CB, d_VIT):
    idx = cuda.grid(1)

    x = 0
    min_cw_dist = 0
    for _i in range(0, d_I.shape[0]):
        min_cw_dist += (d_I[_i, idx] - d_CB[_i, x])**2
    min_cw_dist = min_cw_dist**0.5

    for _x in range(1, d_CB.shape[1]):
        cw_dist = 0
        for _i in range(0, d_I.shape[0]):
            cw_dist += (d_I[_i, idx] - d_CB[_i, _x])**2
        cw_dist = cw_dist**0.5

        x = _x if cw_dist < min_cw_dist else x
        min_cw_dist = min(min_cw_dist, cw_dist)

    d_VIT[idx] = x
```

K-dimensional codewords, corresponds to the jth column of the matrix d_{CB} sized $K \times N$ and the zth VQ index of the output VQ index table VIT, sized $\frac{H}{p} \times \frac{W}{q}$, in raster scan order, corresponds to the zth element of array d_{VIT} sized M. We selected these dimensions so the VQ-Kernel's I/O will be coalesced. Coalesced reads and writes occurs when adjacent threads access/write to adjacent locations in memory. Therefore by storing the input pixel blocks and codewords in a columnar format in d_I, adjacent locations within in each row will be accessed by adjacent threads.

At the start of the VQ-kernel each thread determines its index idx in he overall grid. This index allows each thread executing in parallel to determine which of the M input pixel blocks it has to compress and where to write the corresponding VQ index. For example, executing the VQ-kernel on two adjacent threads identified by indices $idx_1 = 0$ and $idx_2 = 1$ respectively, will read the input pixel blocks $d_I[:, 0]$, and $d_I[:, 1]$, compute the VQ indices x_1 and x_2 and write them to adjacent locations $d_{VIT}[0]$ and $d_{VIT}[1]$, respectively.

The VQ-kernel computes the VQ index of its current executing thread's pixel block $d_I[:, idx]$ as follows. Thread idx calculates the euclidean distance cw_dist between its pixel block $d_I[:, idx]$ and every codeword in d_{CB} while tracking the overall minimum euclidean distance min_cw_dist and its corresponding codeword index x. At the end of the loop x will be the VQ-index with the smallest distance from $d_I[:, idx]$. Although this procedure seems simple, there are two important implementation challenges to consider.

The first challenge is deciding on how adjacent threads will read codewords from d_{CB}. Here we will consider two implementations. The first implementation is using a coalesced memory access pattern, where adjacent threads will read all codewords each starting a different offset. For example, consider two adjacent threads identified by indices $idx_1 = 0$ and $idx_2 = 1$ respectively. Thread $idx_1 = 0$, will loop through the codewords in the order $d_{CB}[:, 0], d_{CB}[:, 1], d_{CB}[:, 2], \ldots, d_{CB}[:, N-1]$, while simultaneously thread $idx_1 = 1$, will loop through the codewords $d_{CB}[:, 1], d_{CB}[:, 2], \ldots, d_{CB}[:, N-1], d_{CB}[:, 0]$. Since these threads are adjacent, for each iteration of the loop, both threads will access adjacent codewords. While these reads are coalesced, there is a significant amount of redundant fetches as each thread will eventually fetch the same N codewords. Another implementation will be for adjacent threads to fetch the same codewords. That is, thread $idx_1 = 0$, will loop through the codewords in the order $d_{CB}[:, 0], d_{CB}[:, 1], d_{CB}[:, 2], \ldots, d_{CB}[:, N-1]$, while simultaneously thread $idx_1 = 1$, will also loop through the codewords $d_{CB}[:, 0], d_{CB}[:, 1], d_{CB}[:, 2], \ldots, d_{CB}[:, N-1]$. This type of memory access pattern results in the complied code using a broadcasting mechanism, where a single codeword is fetched from memory and is then copied to each thread. This mechanism replaces several more expensive fetches to global memory with faster register copys. Experimentally, with codebook sizes $N = 256$ and 1024 the second implementation was 13% and 14% faster respectively.

The second challenge is tracking the best matched codeword index within each thread. The usual if statement which updates the best VQ-index x based on the minimum euclidean distance is prone to warp divergence and register pressure. On the GPU a group of threads, called a warp (usually 32 threads) execute instructions physically at the same time. A wrap achieves maximum performance when all threads within the wrap execute the same instructions. If-then-else statements have the potential create divergent execution paths for threads based on the evaluation of the statement. This warp divergence forces the GPU to serialize execution for those divergent paths, impacting performance. Essentially, all threads where the if statement evaluates to true execute first, followed by the remaining blocks, which consumes two cycles, and impacts the warps performance. Another challenge for the GPU threads is the amount active variables required to determine index of the codeword with the minimum euclidean distance and its VQ index (e.g., idx, x, min_cw_dist, loop counters etc.). When the total number of these variables exceeds capacity of the limited, super-fast registers on the GPU, the compiler is forced to spill them to much slower local memory. This constant moving of data to and from slow memory drastically slows down the VQ-kernel.

There are techniques which can help mitigate the impact of warp divergence and register pressure. Typically, a minimum-reduction technique to find the minimum codeword distance can theoretically prevent wrap divergence while reducing register pressure would require rethinking how the parallelization occurs.

Our current implementation achieves speeds up of 60x, 44x, 34x and 29x for main cookbook sizes 128, 256, 512, and 1024 respectively. In future works we will explore addressing the challenges with the current VQ-kernel to achieve speeds closer to the theoretical maximum of $M = \frac{H}{p} \times \frac{W}{q}$. Algorithm 4 describes the accelerated VQ encoder's algorithm.

B. GPU Accelerated VQ Decoder

The VQ decoder's kernel has much simpler job than its VQ encoder counterpart. The VQ decoder kernel maps a pixel

Algorithm 4 GPU Accelerated VQ Encoder

Input: A grayscale cover image G size $H \times W$, pixel block size $p \times q$, codebook D sized $N \times K$, where $K = p \times q$ and parameter $threads\_per\_block$ default is 64
Output: VQ index table VIT sized $\frac{H}{p} \times \frac{W}{q}$
Create output VQ index table VIT sized $\frac{H}{p} \times \frac{W}{q}$
Divide G into non-overlapping pixel blocks X's sized $p \times q$
Create matrix $blocks$ sized $K \times M$ where $M = \frac{H}{p} \times \frac{W}{q}$
For the ith pixel block X in raster scan order, copy its kth pixel in raster scan order to the kth row of the ith column in $blocks$
$d_{CB} \leftarrow$ cuda.to_device(D^T) ▷ Copy from Host to GPU
$d_{VIT} \leftarrow$ cuda.to_device(VIT)
$d_I \leftarrow$ cuda.to_device($blocks$)
$blocks\_per\_grid \leftarrow \frac{M}{threads\_per\_block}$
Launch `vq_kernel` with context $blocks\_per\_grid$, and $threads\_per\_block$ and parameters d_I,d_{CB},d_{VIT}
$result \leftarrow d_{VIT}$ ▷ Copy from GPU to Host
Reshape $result$ from a 1-D array sized M to 2-D array sized $\frac{H}{p} \times \frac{W}{q}$, as save it as VIT

location onto a codeword's component. Specifically, given a pixel position y_{idx}, x_{idx}, in the reconstructed grayscale image image G', the kernel extracts the corresponding VQ index x from VIT and determines the specific codeword component k to reconstruct this pixel. Applying this kernel to each pixel location generates reconstrcuted image G' sized $H \times W$.

Since adjacent VQ indices in VIT may not map to adjacent codewords, there is a high probability of uncoalesced memory accesses by adjacent threads which will degrade performance. To mitigate this impact, each block of threads loads the entire main codebook in to shared memory location that all threads within the block can access. This shared memory is faster than global memory which reduces the impact of the uncoalesced memory access pattern.

Another consideration is the launch configuration of the VQ decoder's kernel. The main idea is for each GPU thread will reconstruct a single element of the 2D pixel array, hence a 2D organization fo the the GPU's blocks and threads is a natural choice. Experimentally we found (8×8) blocks per grid with (32×32) threads per block to be optimal. In this launch configuration, the number threads is likely to be less than the number of pixels. Therefore, we used a grid stride loop to efficiently distribute the work of reconstructing multiple pixels to a single thread. The optimal launch configurations may differ for other GPUs. Algorithm 6 describes the accelerated VQ decoder's algorithm.

IV. Experimental Results

This section compares the performance of the the CPU-VQ and the GPU-VQ algorithms. Both algorithms were coded in the python programming language version 3.9.13. We implemented CPU-VQ using the numpy library version 1.21.5. The numpy library is a the fundamental package for scientific computing in python. The GPU-VQ was implemented using

Algorithm 5 Python Code for the proposed VQ-Decode-Kernel

```
from numba import types, cuda
@cuda.jit()
def vq_decode_kernel(d_G_prime, d_CB, d_VIT, p, q):
#Copy codebook in the thead block's shared memory to
#reduce the amount of uncoalesced memory access
    cb = cuda.shared.array(shape=(16,1024+1),
                           dtype=types.int16)
    t = cuda.threadIdx.y*cuda.blockDim.x
        + cuda.threadIdx.x

    if t < d_CB.shape[1]:
        for x in range(t, d_CB.shape[1],
                       cuda.blockDim.x*cuda.blockDim.y):
            for k in range(0, d_CB.shape[0]):
                cb[k, x] = d_CB[k, x]
    cuda.syncthreads()
#Loop grid over all pixels positions yidx's and xidx's
    for yidx in range(cuda.blockIdx.y*cuda.blockDim.y
                      + cuda.threadIdx.y,
                      d_G_prime.shape[0],
                      cuda.gridDim.y*cuda.blockDim.y):
        for xidx in range(cuda.blockIdx.x*cuda.blockDim.x
                          + cuda.threadIdx.x,
                          d_G_prime.shape[1],
                          cuda.gridDim.x*cuda.blockDim.x):
#Map pixel location yidx, xidx on to codeword component k
            r = yidx//p
            c = xidx//q
            x = d_VIT[r, c]
            k = p*(yidx % p) + xidx % q
            d_G_prime[yidx, xidx] = cb[k, x]
```

Algorithm 6 GPU Accelerated VQ Decoder

Input: Pixel block size $p \times q$, codebook D sized $N \times K$, where $K = p \times q$ and VQ index table VIT size $\frac{H}{p} \times \frac{W}{q}$
Output: Reconstructed grayscale cover image G' sized $H \times W$
Create and initialized an output image G' sized $H \times W$
$d_{CB} \leftarrow$ cuda.to_device(D^T) ▷ Copy from Host to GPU
$d_{VIT} \leftarrow$ cuda.to_device(VIT)
$d_{G'} \leftarrow$ cuda.to_device(G')
$threads\_per\_block \leftarrow (32 \times 32)$
$blocks\_per\_grid \leftarrow (8 \times 8)$
Launch `vq_decode_kernel` with context $blocks\_per\_grid$, and $threads\_per\_block$ and parameters d_{CB},d_{VIT}, $d_{G'}$
$G' \leftarrow d_{G'}$ ▷ Copy from GPU to Host

the python CUDA toolkit version 10.1. Both implementations were executed on Laptop with 12th Gen Intel(R) Core(TM) i7-12650H, 2.30 GHz, 16 GB RAM, and NVIDIA RTX 3050Ti GPU.

All experiments used standard cover images sized 512×512, with the main codebooks sizes $N = 128, 256, 512,$ and 1024. The LBG algorithm [21] was used to generate these main codebooks. Each codebook contained N, 16-dimensional codewords.

We evaluated the performance based on average runtime (in miliseconds), after executing each test case 100 times. Since both implementations generated identical output, the Bit rate (BR) and peak signal-to-noise ratio (PSNR) of the compressed images were the same in all test cases.

Tables I and II compare the runtime (in miliseconds) of the CPU and GPU implementations of the VQ encoder across all main codebook sizes. The work done by both implementations increases as the main codebook size N increased because more time was required to search the longer codebooks for the best match codeword for each input pixel block. In all test cases the GPU accelerated VQ encoder achieved better execution times and its CPU counterpart. This speedup, was do to multiple threads compressing input pixel blocks in parallel rather than a single CPU thread processing input blocks serially.

Theoretically, we can estimate the maximum speed $\mathcal{M}$ up as follows. The work done by both implementations are dominated by the search for the best match codeword, which requires $\mathcal{O}(N)$ comparisons in the worst case for main codebook size N. Though both implementation are in the same complexity class, their constants are different since the GPU implementation reduces the time to do the work by factor of T as it can execute the VQ-kernel on T threads simultaneously. Specifically, the runtime of the CPU encoder is $M \times \mathcal{O}(N)$ where $M = \frac{H}{p} \times \frac{W}{q}$, while the GPU encoder's runtime is $\frac{M}{T} \times \mathcal{O}(N)$. Thus the maximum speed up $\mathcal{M} = T$. As proposed GPU encoder assigns all the work to compress an input pixel block X to a single thread, T cannot exceed the total number of blocks M which is 16384 for our test cases.

Table IV captures the observed speed up for all main codebook sizes. On average the GPU VQ encoder achieved speed ups of 61x, 44x, 34x and 29x for main codebook sizes $N = 128, 256, 512,$ and 1024. respectively. Although these improvements are significant there is still much room for improvement. Specifically, the proposed VQ-kernel is impacted by the issues of thread divergence and register pressure. These issues scale with the size of the main codebook N, as each GPU thread needs to track the live values of the best matched VQ index and its codework distance across more iterations, which requires more registers increasing the likely hood of registers spilling over in to local memory. Figure 1 illustrates this phenomenon as the VQ encoder speeds increases at a faster rate as N decrease since the amount of thread divergence and register spillage reduces as N decreases.

Unlike the VQ-kernel for the encoder, its counter part, the VQ-kernel for the decoder is not affected by these issues. In general, the decoder simply takes a VQ index x and reconstructs the pixels of a pixel block using the components of codeword CW_x. Therefore, its runtime for both the CPU and GPU implementations is $\mathcal{O}(K)$, as the work done to reconstruct every non-overlapping pixel block X' of G' is proportional to the length of its codeword CW_x. Because the decoder's VQ-kernel is effectively mapping codeowords components to pixel locations, no if statements or large amount of registers are required which completely avoids the issues of thread divergence and register pressure. Table III compares the execution times of both implementations. Across all main codebook sizes, the average execution times were similar for both the CPU and GPU implementations. Table IV shows that

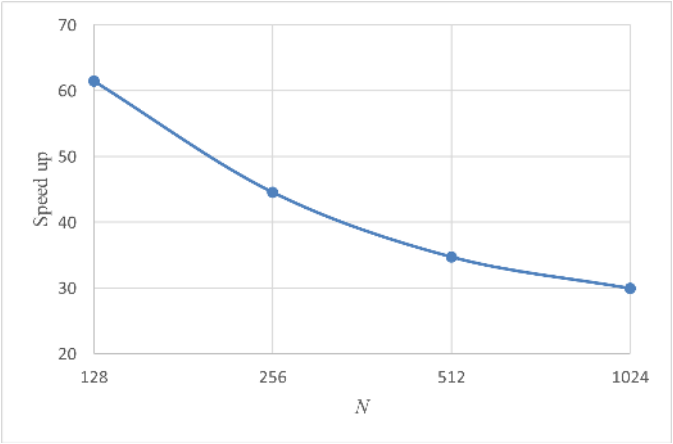

Fig. 1. Average speed up of the VQ Encoder with main codebook sizes $N = 128, 256, 512, 1024$, calculated as $\frac{\text{CPU VQ Encoder time}}{\text{GPU VQ Encoder time}}$

the speed up of 200x was similar for all main codebook sizes. This improvement is an order of magnitude greater than the VQ encoder's speed up due to the lack of thread divergence and register pressure. These results serve a motivation to address these issues by re-thinking the task parallelism for the VQ encoder in future works.

V. Conclusion and Future Work

In this paper we discuss the GPU accelerated implementations of the Vector Quantization (VQ) encoder and decoder. The traditional raster-scan order is optimal only in a single-core architecture as every non-overlapping input pixel block is eligible for VQ compression. Thus, the traditional approach is sub-optimal in an environment where multiple cores are available. For both the VQ encoder and decoder we defined GPU accelerated algorithms which utilized CUDA kernels to parallelized their tasks. Although these implementations do not change their respective time complexity classes it does distribute the work across multiple threads which theorectically reduces their constants and improves performance.

The experimental results indicate that runtime of the VQ encoder is improved by 61x, 44x, 34x and 29x for main codebook sizes $N = 128, 256, 512,$ and 1024 while the GPU accelerated VQ decoder is at least 200x faster across all codebook sizes. These results demonstrate that the proposed GPU accelerated algorithms make more efficient use of today's modern hardware and makes these algorithms more practical for real-time applications such as multimedia streaming and reversible data hiding.

We believe that there is room to improve these numbers by further exploiting hardware, architecture and kernel features. We plan to provide these enhancements in a GitHub repository for use by the public.

References

[1] P. F. Gorder, "Multicore Processors for Science and Engineering," in Computing in Science & Engineering, vol. 9, no. 2, pp. 3-7, March-April 2007, doi: 10.1109/MCSE.2007.35.

TABLE I
AVERAGE EXECUTION TIME IN (MS) OF THE CPU VQ ENCODER WITH MAIN CODEBOOK SIZES $N = 128, 256, 512$ AND 1024

Image	128	256	512	1024
Baboon	187.120	264.484	410.719	678.040
Barbara	202.475	256.992	407.262	679.685
Boats	182.662	260.291	408.356	697.246
GoldHill	176.312	265.671	396.035	692.758
JetF16	183.462	266.655	418.922	697.352
Lena	186.520	256.591	400.908	708.485
Peppers	180.758	265.766	403.528	693.777
Sailboat	179.556	259.452	401.013	692.605
Tiffany	179.558	269.656	405.887	706.226
Toys	178.655	260.608	409.983	676.630
Zelda	180.864	262.410	397.021	674.503
Average	183.449	262.598	405.421	690.664

TABLE II
AVERAGE EXECUTION TIME IN (MS) OF THE GPU VQ ENCODER WITH MAIN CODEBOOK SIZES $N = 128, 256, 512$ AND 1024

Image	128	256	512	1024
Baboon	2.944	5.900	11.675	23.213
Barbara	2.969	5.892	11.664	23.071
Boats	3.008	5.906	11.588	23.052
GoldHill	3.022	5.917	11.631	23.038
JetF16	2.975	5.944	11.661	23.036
Lena	2.977	5.910	11.654	23.018
Peppers	3.012	5.900	11.814	23.069
Sailboat	2.975	5.872	11.797	23.064
Tiffany	2.960	5.897	11.645	23.169
Toys	2.986	5.874	11.727	23.229
Zelda	3.024	5.884	11.745	23.232
Average	2.987	5.900	11.691	23.108

TABLE III
AVERAGE EXECUTION TIME IN (MS) OF THE CPU AND GPU VQ DECODER'S IMPLEMENTAITONS WITH MAIN CODEBOOK SIZES $N = 128, 256, 512$ AND 1024

Implementation	128	256	512	1024
CPU	68.700	69.202	72.853	77.739
GPU	0.340	0.259	0.274	0.306

TABLE IV
SPEEDUP OF THE VQ ENCODER AND DECODER WITH MAIN CODEBOOK SIZES $N = 128, 256, 512$ AND 1024, CALCULATED AS $\frac{\text{CPU VQ ENCODER TIME}}{\text{GPU VQ ENCODER TIME}}$ AND $\frac{\text{CPU VQ DECODER TIME}}{\text{GPU VQ DECODER TIME}}$ RESPECTIVELY

Algorithm	**128**	**256**	**512**	**1024**
VQ Encoder	61.434	44.511	34.680	29.890
VQ Decoder	202.192	267.190	265.510	253.641

[2] Moore, C., Conway, P. (2009). General-Purpose Multi-core Processors. In: Keckler, S., Olukotun, K., Hofstee, H. (eds) Multicore Processors and Systems. Integrated Circuits and Systems. Springer, Boston, MA. https://doi.org/10.1007/978-1-4419-0263-4_6

[3] G. Blake, R. G. Dreslinski and T. Mudge, "A survey of multicore processors," in IEEE Signal Processing Magazine, vol. 26, no. 6, pp. 26-37, November 2009, doi: 10.1109/MSP.2009.934110.

[4] W. J. Dally, S. W. Keckler and D. B. Kirk, "Evolution of the Graphics Processing Unit (GPU)," in IEEE Micro, vol. 41, no. 6, pp. 42-51, 1 Nov.-Dec. 2021, doi: 10.1109/MM.2021.3113475

[5] P. Shantharama, A. S. Thyagaturu and M. Reisslein, "Hardware-Accelerated Platforms and Infrastructures for Network Functions: A Survey of Enabling Technologies and Research Studies," in IEEE Access, vol. 8, pp. 132021–132085, 2020, doi: 10.1109/ACCESS.2020.3008250

[6] A. Ozsoy and M. Swany, "CULZSS: LZSS Lossless Data Compression on CUDA," 2011 IEEE International Conference on Cluster Computing, Austin, TX, USA, 2011, pp. 403-411, doi: 10.1109/CLUSTER.2011.52

[7] W. Tang, T. Chen and M. Armstrong, "GPU-accelerated parallel all-pair shortest path routing within stochastic road networks", International Journal of Geographical Information Sci-ence, 53-85 (2024)

[8] R. Gray, "Vector quantization," in IEEE ASSP Magazine, vol. 1, no. 2, pp. 4-29, April 1984, doi: 10.1109/MASSP.1984.1162229

[9] T. Kim, "Side match and overlap match vector quantizers for images," in IEEE Transactions on Image Processing, vol. 1, no. 2, pp. 170-185, April 1992, doi: 10.1109/83.136594

[10] W. He, K. Zhou, J. Cai, L. Wang, G. Xiong, Reversible data hiding using multi-pass pixel value ordering and prediction-error expansion, Journal of Visual Communication and Image Representation 49 (2017) 351–360

[11] D. An, X. Pu, D. Hao, R. Zhao and Y. Zhang, "Reversible Data Hiding with Flexible Extraction for Thumbnail-Preserving Encryption," in IEEE Transactions on Consumer Electronics, doi: 10.1109/TCE.2025.3545959

[12] Sanjay Kumar, Gurjit Singh Walia, Anjana Gupta, "An efficient technique for reversible data hiding using bidirectional histogram shifting and multistage embedding", Engineering Applications of Artificial Intelligence, 156, (2025), https://doi.org/10.1016/j.engappai.2025.110986

[13] Ren, Y., Qin, J., Xiang, X. et al. Reversible data hiding in encrypted images based on Lasso regression predictor and dynamic secret sharing. Appl Intell 55, 95 (2025). https://doi.org/10.1007/s10489-024-05929-6

[14] Chi, H., Chang, C. C., & Liu, Y. (2020). An SMVQ compressed data hiding scheme based on multiple linear regression prediction. Connection Science, 33(3), 495–514. https://doi.org/10.1080/09540091.2020.1852179

[15] Chi, H.-X., Horng, J.-H., Chang, C.-C., & Li, Y.-H. (2022). Embedding Biometric Information in Interpolated Medical Images with a Reversible and Adaptive Strategy. Sensors, 22(20), 7942. https://doi.org/10.3390/s22207942

[16] Manohar, K., Kieu, T.D. A centroid based vector quantization reversible data hiding technique. Multimedia Tools Appl 78, 25273—25298 (2019). https://doi.org/10.1007/s11042-019-7631-3

[17] C. C. Chang, W. -l. Tai and C.C. Lin, "A Reversible Data Hiding Scheme Based on Side Match Vector Quantization," in IEEE Transactions on Circuits and Systems for Video Technology, vol. 16, no. 10, pp. 1301–1308, Oct. 2006, doi: 10.1109/TCSVT.2006.882380

[18] Wang, Y., Pan, Z., Li, R. et al. New SMVQ scheme with exactly the same PSNR of VQ by introducing extend state codebook. Multimedia Tools Appl 77, 21571–21588 (2018). https://doi.org/10.1007/s11042-017-5584-y

[19] C.C. Lin, X.L. Liu, S.M. Yuan, Reversible data hiding for VQ-compressed images based on search order coding and state-codebook mapping, Information Sciences 293 (2015) 314–326.

[20] Manohar, K., & Kieu, T. D. (2020). A Strategy for the Linear Time Implementation of the SMVQ Algorithm. IETE Technical Review, 39(2), 270–275. https://doi.org/10.1080/02564602.2020.1842258

[21] Y. Linde, A. Buzo and R. Gray, "An Algorithm for Vector Quantizer Design," in IEEE Transactions on Communications, vol. 28, no. 1, pp. 84-95, January 1980, doi: 10.1109/TCOM.1980.1094577