GPU-Based Implementation of the High Payload (7,4) Hamming Code Steganographic Scheme

Ander Lee
National Chiao Tung University

Yueh-Hsin Tu
National Chiao Tung University

Shaoyu Fang
National Chiao Tung University

Abstract

In this project, we propose a GPU-based implementation of the (7,4) Hamming steganographic scheme. The scheme features in a high embedded payload with a trade-off of slightly lower visual quality. We further improve its throughput by applying parallel programming using CUDA. The loop-level and instruction-level parallelism are exploited in order to maximize the speedup. The result shows a considerable speedup compared to its original serial version. While increasing the input image size, the acceleration is still able to sustain.

1 Introduction

The two important parameters of a steganographic technique are the embedding payload and embedding efficiency, the former refers to the amount of secret data that could be loaded with, and the latter addresses to the modification made on the cover image. The (7,4) Hamming steganographic scheme proposed by Chang et al. [1] provides a high embedded payload and a satisfactory embedding efficiency. However, from the viewpoint of throughputs, the full potential of the high payload advantage could not be utilized as the time cost increases proportional to the growth of the cover image size. Hence, we propose to take the advantage of SIMT model and implement Chang's scheme on a Nvidia GPU. The resultant speedup allows more secret data to be embedded into higher resolution image in less time. On the other hands, the visual degradation could as well be compensated by higher number of pixels presented.

1.1 Serial Version

In Chang et al.'s proposed scheme, the parity check matrix H of the (7,4) Hamming code (see Fig. 1.) is used to classify 128 different bit strings of length seven into 16 groups g_v^u. This mapping could be completed by lookup a pre-built coset with syndrome as keys [2]. Each group contains eight different bit strings of length seven. Then, a segment of seven secret bits is computed to match one set of the index (u, v) the corresponding g_v^u would be taken to replace the LSB of the cover pixel segment. This completes the embedding phase.

For the extracting phase, the sender needs to send the parity check matrix H to the receiver via a secure channel. At the receiving side, the receiver calculates the syndromes of the stego segment, then pick up the particular bits of interest from the results, conclude them as parts of secret bits. The total reconstruction of the secret data finished after all cover pixels have been processed. The execution flow of the program is shown in Fig. 2.

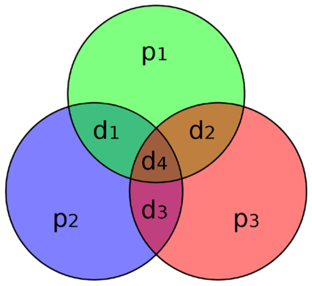

Figure 1. An example of a Hamming (7, 4) parity matrix H with message bits $[d_1, d_2, d_3, d_4]$ and parity bits $[p_1, p_2, p_3]$.

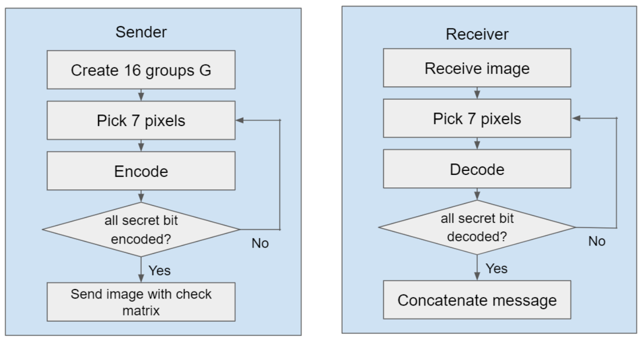

Figure 2. Sender (embedding) & Receiver (extracting)

In the actual C code implementation, we firstly parse the bmp cover image and the secret message in txt file. Then create an array of struct consists of 16 groups G^0, G^1,…, G^{15}. Each contains 8 strings of length seven $g_0^u, g_1^u, g_2^u, g_3^u, g_4^u, g_5^u, g_6^u, g_7^u$. Compute 8×16 coset entries then store in the struct. Once the struct is filled up completely, read seven secret bits $(s_1s_2s_3s_4s_5s_6s_7)$ at a time and embed those bits into each of the seven consecutive pixels. Compute the decimal value $u = (s_3s_5s_6s_7)$ and $v = (s_1s_2s_4)$ in order to retrieve the corresponding coset index of g_v^u. Replace the least significant bits of each seven pixels with the bit string in g_v^u. After embedding all secret bits, reconstruct the image in bmp format. The stego image is ready to be transmitted.

The receiver parses the bmp file and extract the message in the similar manner. In specific, isolate the LSB for each of the seven pixels, $L = (l_1, l_2, l_3, l_4, l_5, l_6, l_7)$. Compute the syndrome vector $Z = (z_1, z_2, z_3)^T = H \times L$, and finally reorder the secret bits such that $s = (z_1, z_2, l_3, z_3, l_5, l_6, l_7)$. Repeat the process until all secret message have been decoded.

2 Proposed Method

This section explains how the parallelism is done on the main three phases of grouping (create coset table), embedding, and decoding. The discussion of some techniques related to optimization for concurrency and memory in CUDA is also given.

2.1 Hardware Specification

In CUDA programming, the choice of hardware will result in great impact on the how grid size and block size should be fined-tuned, as well as the availability of API functions. In our experiment, the relevant spec is shown as the following:

Nvidia GeForce GTX 1060 6 GB
Compute capability: 6.1
Memory bandwidth: 192.2 GB/s
PCIe link rate: 5 Gbit/s

The profiling tool used for analyzing the latency and throughput is the Nvidia Visual Profiler (nvvp) release version 9.1.

2.2.1 Grouping

As mentioned in Section 1.1, the coset is a fixed size C-style struct of 128 entries. Hence the dimBlock is set to be 128 intuitively. The index information of $(\boldsymbol{u}, \boldsymbol{v})$ is stored in each thread's own register for fast accessing. Since the threads do not communicate or read any data from host, no shared memory is needed. The occupancy rate is 100%. See Fig. 3. For the execution flow.

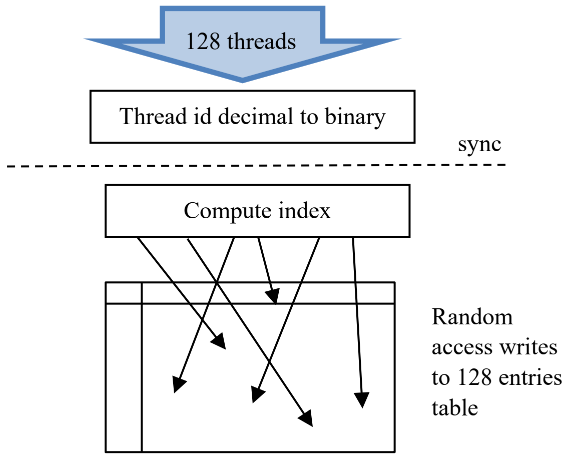
128 threads
Thread id decimal to binary
sync
Compute index
Random access writes to 128 entries table

Figure 3. parallelism for grouping phase

2.2.2 Embedding

Every seven threads are bound to a "team" that processes seven bits secret data, hence it could perform the coset index computation. We use the stride access method (see Fig. 4) for reading the secret data. The stride length is set to 224 (as well as the dimBlock), in which this value is derived from the common multiplier of the "team" size and the warp size, i.e. 7×32 = 224. This configuration ensures that all working threads are aligned to the warp and are fully utilized. The index information is again stored in private registers, similar to that in the grouping phase. Shared memory is used, its advantage will be discussed in more detail in Section 2.3.2. The overall occupancy reaches 98%.

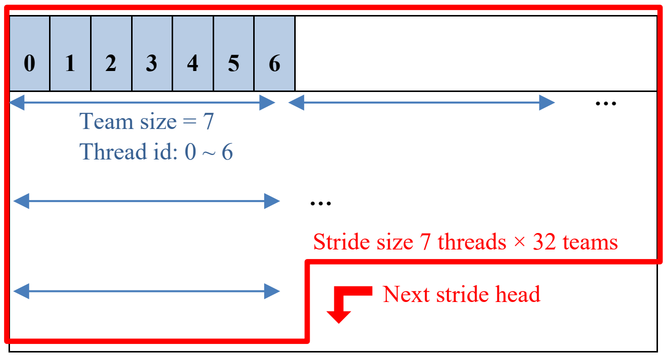

Figure 4. Stride access and threads "team"

2.2.3 Decoding

The same stride access and thread "team" strategy described in Section 2.2.2 apply to the decoding phase as well. Because syndrome vectors need to be computed for every pixel. To avoid data race, each thread keeps its own copy of the parity matrix H in the register. The occupancy is 98%.

2.2.4 Global Sync

In our proposed method, we have parallelized all three of the grouping, embedding, and decoding phases. However, since there is in fact no signal being passed to either between or within each phase, suggesting that out of order (OOO) behavior might occur. Therefore, we have the need to set barrier in order to maintain their sequential consistency. As shown in Fig 5, in the embedding phase, we let each individual thread to handle one pixel/secret independently. But in order to compute the correct value of $u = (s_3 s_5 s_6 s_7)$ and $v = (s_1 s_2 s_4)$, each thread need to wait for all other threads, thus we set a barrier here to ensure it. After computing u and v, we get corresponding g_v^u and replace its string bits to the LSB of each pixel. To make sure every bits are actually embedded successfully, we also set a barrier here before accessing the next seven pixels.

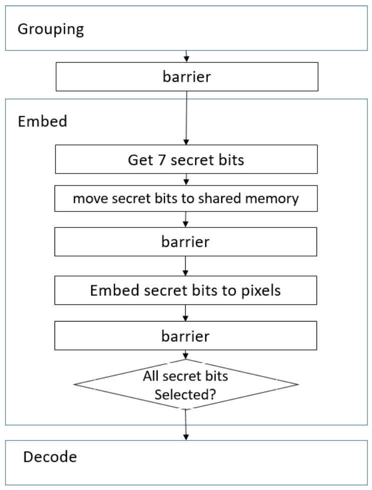
Grouping
barrier
Embed
Get 7 secret bits
move secret bits to shared memory
barrier
Embed secret bits to pixels
barrier
All secret bits Selected?
Decode

Figure 5. The thread global synchronization scheme

2.3 Optimization

This section will explain the method used to exploit instruction-level parallelism (ILP) during the embedding phase. Furthermore, the implement of the CUDA texture memory and shared memory will be discussed. The ultimate goal for adopting this memory hierarchy is to eliminate redundant off-chip memory access.

2.3.1 Optimization through ILP

At the encoder, a read stride has the length of 7×32 while the cover image dimension could be arbitrary $(H \times W)$. The remainder parts of image size / stride size would be the number of pixels at the end of image which are not taken to processing, see Fig. 6.

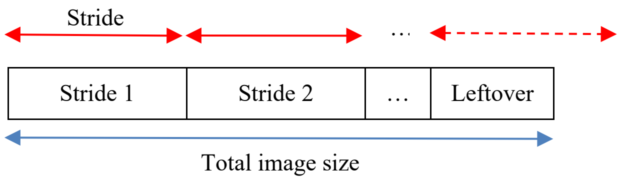

Figure 6. Leftover pixel

An extra embedding function call is needed to deal with the leftover pixels. The strategy used is to apply CUDA stream. Operations in different streams may run concurrently. As each pixel is independently loaded with secret data, the embedding processes of the main part and the leftover part do not have dependency. The only true data dependency occurs between the memory copy phase and the computation phase of individual pixel, in which the latency can still be hidden given the memory copy performs in an asynchronous manner. In other words, memory copy of the adjacent pixel may start as soon as the computation of the previous pixel is finished. The arrangement is shown as Fig. 7.

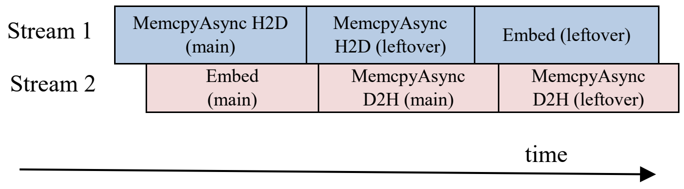

Figure 7. CUDA streams allow better concurrency to hide the memory latency

2.3.2 Optimization for Memory Access

CUDA texture memory is a read-only, on-chip cache, that is optimized for 2D spatial locality [3]. These properties of texture memory are coincidently fitted for the usage on secret data under the context of embedding phase, where the data that stored in consecutive locations are read by threads of the same warp.

To further reduce the memory access latency for secret data, a shared memory space consists of 32×7 entries is created during the embedding phase. Once again, the "32" is correspond to the warp size, in hope of maximize all the active threads in each memory transaction. And the "7" reflects the number of secret bits needed to access in order to compute the index $(\boldsymbol{u}, \boldsymbol{v})$ mentioned in Section 1.1. The overall effect of the shared memory results in the amount of access to off-chip memory reduced by seven times.

To achieve high memory bandwidth for concurrent accesses, shared memory is divided into equally sized memory modules (banks) that can be accessed simultaneously. Shared memory banks are organized such that successive 32-bit words are assigned to successive banks [4]. The thread "team" described in Section 2.2.2 is purposely designed to avoid bank conflict by assigning different thread index to access neighboring array entries, i.e. thread accesses location in a jumping manner. This is illustrated in Fig. 8.

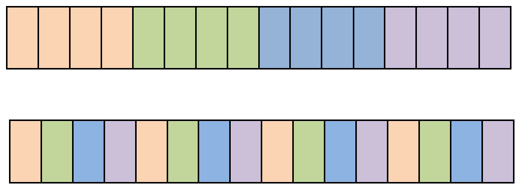

Figure 8. different colors represent different threads in the same warp. The above accessing approach will result in bank conflict, while the bottom will not.

3 Experimental Results

In this section we will examine the correctness, speedup, and scalability of the GPU implementation.

Three different resolutions of gray scale image are put into testing, they are 256*256, 512*512, and 1024*1024 respectively. Fig. 8 shows the example of 256p image before and after embedded with secret messages.

Figure 8. the original cover image (left), and the stego image (right). Human eyes could not distinguish the difference.

In the real system, the embedding process (include the grouping phase) and the decoding usually taking place on separated machines at different locations. Hence our speedup calculation will also treat these two phases separately for practical purpose. Table 1 and Fig. 9 shows the performance speedup.

image size	embed speedup	decode speedup
256P	64	90
512P	70	95
1024P	71	104

Table 1. Speedup with different size of cover image

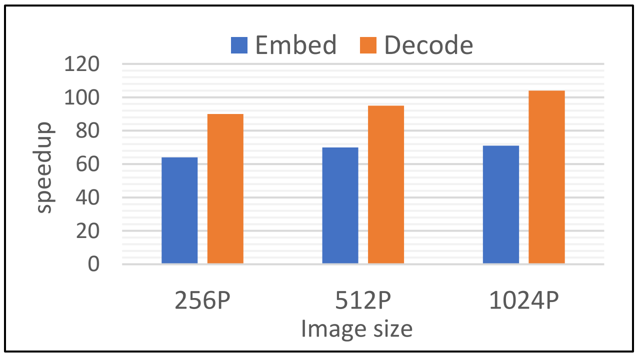

Figure 9. Speedup with different size of cover image

The results from Table 1. suggests that the speedup is lower for smaller size image due to the startup overhead of CUDA is dominating the critical path, such as the *cudaMalloc( )* used to setup the device memory. However, as the image size grows, these overheads would not increase proportionally. Eventually they will disappear once the problem size is sufficiently large.

The decoding achieves more speedup than the embedding. It is because at the decoding phase the syndrome vectors are computed on-line, while the embedding phase relies on accessing the cached coset table to process the secret bits. This agree with the common perception of GPU programming that the memory access is the main bottleneck of performance.

Table 2 shows the scaling capability of our proposed method. As the problem size and the processor number grows almost the same rate, there is still improvement to the speedup. And thus suggests that the implementation is weakly scalable.

image size	256	512	1024
Image growth Gi	1	4	16
processor	292	1170	4681
Processor growth Gp	1	4	16
Gi/Gp	1	1	0.99

Table 2. Scaling ability

4 Related Work

This section discusses briefly about an OpenCL DCT-based steganographic scheme proposed by Poljicak et al. [5].

In encoder part, the cover image is transformed from RGB to YCbCr color space. Then Y component of image is separated into 8×8 blocks. Then each block is transformed to 8×8 coefficient using DCT. Through data embed process, some of coefficients are modified. Then the rest of process are just inverse DCT and YCbCr to RGB. Each of steps above can be parallelized. In decoder part, the first 3 steps are same as encoder, RGB to YCbCr, separated into 8x8 blocks, and DCT. After these 3 steps, the embedded data done by encoder is generated. The final step is to extract the hidden data from embedded data. In the RGB to YCbCr and YCbCr to RGB modules, parallelism is exploited by partitioning the original image, and grouping pixels into work-groups.

DCT and iDCT OpenCL implementation are based on CUDA's DCT8×8 example, which is also fully developed in OpenCL. DCT8×8 transform is optimized by avoiding redundant multiplications and utilizing the separability of 2D transform. Memory exploitation is also performed, such as memory hierarchy usage and bank conflict reduction.

Despite of an extensive implementation of parallelism and optimization, the speedup (maximum 13.07 for encoder and 5.26 for decoder) is still inferior to our method. The primary reason is the difference on steganography algorithm. With DCT-based, it needs many barriers to ensure the sequential data consistency. For example, while doing DCT step, each 8×8 block has to be transformed from RGB to YCbCr in prior because it needs all 8×8 Y component values to calculate each DCT coefficient. Another example is while doing embed step, each 8×8 DCT coefficient must be calculated because the embedded coefficient is related to the summation of 64 DCT coefficient. In contrast, with the Hamming (7, 4) method, the overhead due to barriers is fewer because of the less transformation process and low complexity nature of the algorithm. However, in the aspects other than speedup, the DCT-based steganographic method has its advantages. Firstly, it is robust to attack. In data embedding process, data is not embedded in every block. Instead, the random number generator controlled with the secret key determines which blocks embed data. Thus, it is difficult for attackers to extract data on embedded image without the secret key. Secondly, it has a low impact on the image quality, which presents in lower SNR compared to the Hamming (7, 4) scheme.

5 Conclusion

Although achieving impressive speedup of over 100 times max., our proposed method is heavily fine-tuned to the spec of GPU, in which means the code might not be portable on different platform. The performance may be better or worse depending on the spec of the particular machine it runs on.

Future work could focus on real-time video encode/decode, as proven viable by related work.

Source Code

https://github.com/whollybrewed/stego74-cuda/tree/master

References

[1] C. Chang, T. D. Kieu and Y. Chou, A High Payload Steganographic Scheme Based on (7, 4) Hamming Code for Digital Images, *2008 International Symposium on Electronic Commerce and Security*, Guangzhou City, 2008, pp. 16-21.

[2] MacWilliams, Florence Jessie, and Neil James Alexander Sloane. *The theory of error-correcting codes*. Vol. 16. Elsevier, 1977.

[3] NVIDIA. NVIDIA CUDA C Programming Guide, Version 4.2. 2012.

[4] Mark Harris, *Using Shared Memory in CUDA C/C++*, https://devblogs.nvidia.com/using-shared-memory-cuda-cc/

[5] Poljicak, A., Botella, G., Garcia, C. et al, Portable real-time DCT-based steganography using OpenCL, *J Real-Time Image Proc* 14, 87–99 (2018)