

X265 Motion Estimation

1. Introduction

H.265/High Efficiency Video Coding(HEVC) is one of the most advanced video codec. It is the successor to the H.264, the most popular current format. Compare to H.264, HEVC could double the data compression ratio at the same level of video, or provide substantially improved video quality at the same bit rate. The idea behind compression is that the color a pixel, or a block of pixels, could be predicted from the color of its neighbors within the same frame (intra-frame prediction), or from adjacent frames (inter-frame prediction).

In this project, we are primarily concerned with intra-prediction. Supported by the implementation of HVEC, we could predict a block of pixels as the average of its neighbors (DC mode), a smooth gradient based on its neighbors (Planar mode), and a linear extension of its neighbors in one of 33 directions (Angular mode)[1]. These add up to a total of 35 different intra-prediction modes. Due to the nature of the independence of HVEC's block-based prediction, all the 35 modes' prediction computations on different blocks are independent from each other. So we applied GPU to parallelize those computations, by having each thread responsible for all the computations (including prediction, satd computation, and sorting) of one image block for a specific image block size of a frame.

In the end, we succeeded to implement both the CPU serialized and GPU parallelized version of intra-prediction algorithm, and our final code can match the result of the given reference code. To optimize the parallelized CUDA code, we applied memory coalescing, SATD computation packaging, etc. techniques. The experiments show that our optimized CUDA code can achieve a speedup of about 12X when image size is 3840x2160, and about 1.02X even when the image size is as small as 320x180.

2. Problem Description

Figures 1 through 4 summarized the work flow of the x265 encoding scheme. For simplicity, we assume the input video only has one frame. In a multi-frame case, same work needs to be done for all frames.

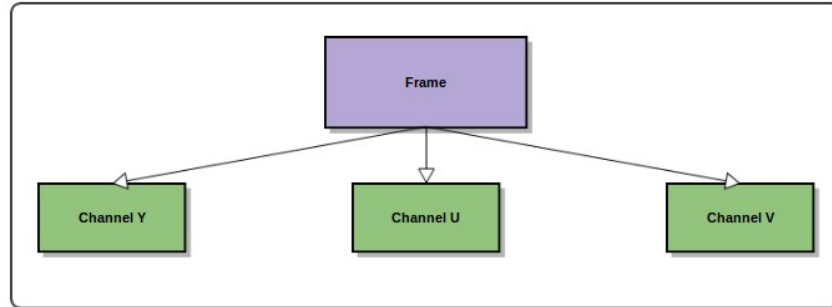


Figure 1: Images divides into Y, U, V channels

The work of the encoder could be viewed as a hierarchical process, and Figure 1 shows the first level of it. First, a frame, which is essentially a large collection of pixels, is divided into 3 channels: Y, U and V, 3 independent arrays of numeric values for the corresponding channel. These 3 arrays can serve as input to our next process.

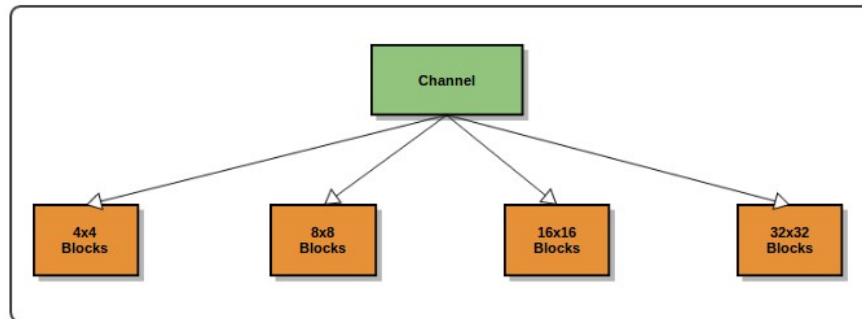


Figure 2: Each channel is broken into 4x4, 8x8, 16x16 or 32x32 blocks

Figure 2 shows the second level of the hierarchical process. In this step, the level one generated channels are further divided into smaller units — blocks. Luma size can be 4, 8, 16 or 32, so that block size could be 4x4, 8x8, 16x16 or 32x32. Each set of blocks are then used as input for the next procedure.

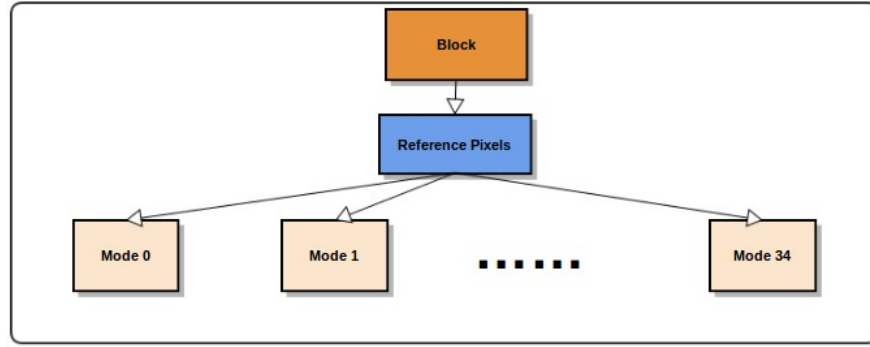


Figure 3: Reference pixel generation and prediction modes

Using the generated blocks for each channel, reference pixels needed for prediction are generated. Since each prediction mode essentially uses the same set of reference pixels, just with different indexing, reference pixels only need to be generated once. A filtered version is also generated for several mode that requires filtered reference pixels. As of now, we have all the information needed for actual prediction.

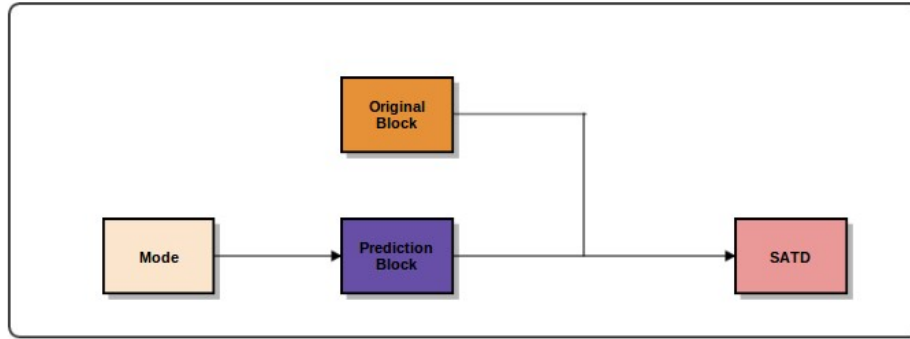


Figure 4: Calculating SATD (Sum of Absolute Difference)

As shown in Figure 4, each mode will produce a prediction block for each original block. Each original/prediction pair could then be compared and SATD could be computed. After SATD of all prediction mode of a given block is computed, the mode-SATD pairs can then be sorted based on the SATD value, thus the best mode for that specific block could be determined.

It is clear that many operations in this process are independent and thus could be executed in parallel to greatly improve performance. The next section would discuss our approach of parallelizing it using a GPU.

3. Implementations

This section first introduces the serial code we wrote by referring to the reference code, and also discusses how our parallelized code works in terms of threading and the traffic to and from GPU, followed by the optimizations we did to improve the performance of our CUDA code.

3.1 Serialized code and its performance

Our objective is to parallelize the reference code and to get as much speedup as possible. However, when we tried to understand the reference code, we found that it is highly object oriented and is written in C++ style. Although the reference code is clean and human friendly, it is not “GPU” friendly. It is not easy to transfer the object oriented reference code directly to GPU and parallelize it . As a result, we decided to develop our own serialized code that does the same work as the reference code but in a more cleaner and GPU friendly way. To achieve this goal, the most of the data structure we used are arrays instead of objects. When we finished our serialized code, we compared its performance with the reference code and our serialized code is indeed faster as we expected. We believe the simplicity of our data structures contributes to this result.

Before we start parallelizing , we ran a timing analysis to give us some idea on the “weight” of each part of the program, and got the following result.

	720*480	1920*1080	3840x2160
REF_PIXEL	1.29%	1.29%	1.44%
MODE_PL	1.52%	1.51%	1.57%
MODE_DC	1.12%	1.17%	1.20%
MODE_AG total	40.57%	40.24%	39.62%
SATD total	47.18%	47.24%	47.66%
SORT	8.32%	8.55%	8.51%

Figure 5 Time weight for different phase with different input image size

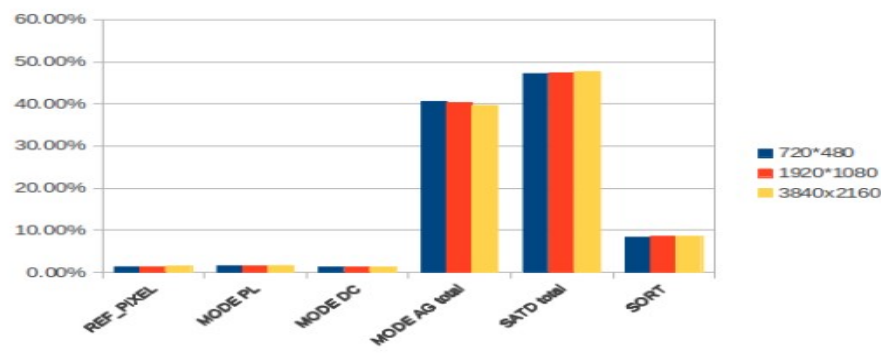


Figure 6: Bar chart for the timing Analysis

The timing analysis give us valuable information, such as the bottleneck of the application, and is used as an important reference as we progress.

3.2 Parallelized code

Kernel Summary:

INPUT: Original block, Unfiltered reference pixels

OUTPUT: Sorted Mode \square SATD pairs

Our parallelized code is quite similar to the serialized code in terms of operations taken, with the only difference being that some part of the work has been shifted to GPU. Just like the serial code, the parallelized code starts off in CPU by having the image divided into 3 channels Y, U and V, and then further dividing into 4x4, 8x8, 16x16 and 32x32 blocks. Reference pixels are generated using these original blocks. Starting here, the parallelized code operates differently as the original block and the reference pixels are then passed to GPU, as shown in Figure 7, rather than continue calculation on CPU.

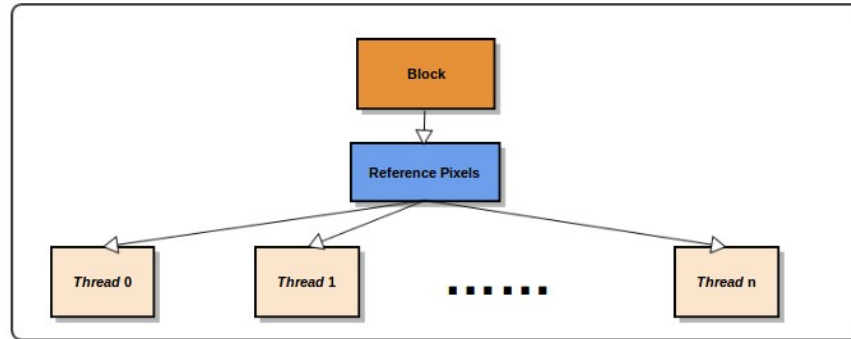


Figure 7: Work assigned to threads

Each thread is in charge of one 4x4, 8x8, 16x16 or 32x32 block, depending on the current value of luma size, and thread number should equal the number of blocks so we are operating on the whole image. Each thread then proceed to filter its reference pixels to create an array of filtered reference pixels, which is later used in some of the prediction modes. At this time, all the information required for prediction is ready. Each thread then start the process of prediction using each of the 35 prediction modes, as shown in Figure 8 and produce a prediction block for that mode. The prediction mode is then compared with the original block and SATD is calculated.

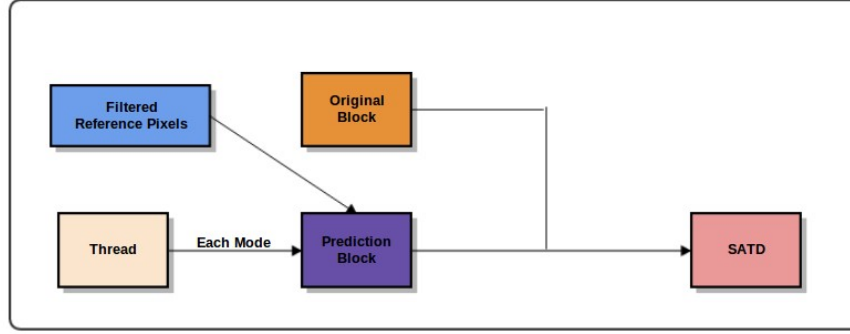


Figure 8: A single thread's prediction process

After a thread has completed calculating the SATD values for all 35 different modes, it sorts the mode-SATD pairs using SATD as the measure. The sorted result is then passed back to CPU, marking the end of GPU's job, as shown in Figure 9

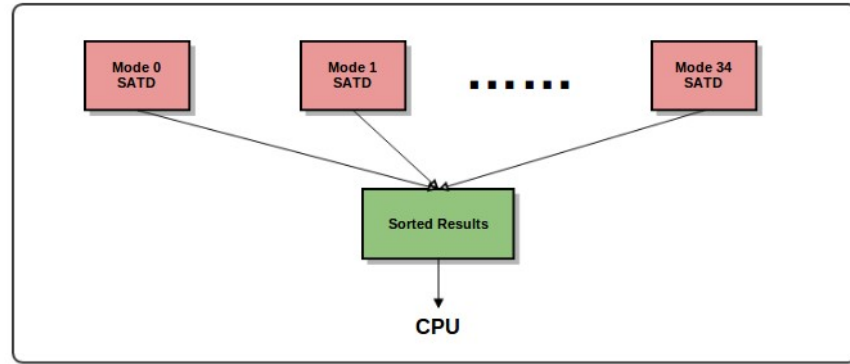


Figure 9: Results passed back to CPU

3.3 Optimizations

In this section, we will talk about several optimization efforts that are made on parallelized code for further improve performance.

3.3.1 memory coalescing

The original GPU parallelized version has similar performance as the serial version of the reference code. We analyzed the reason based on our understanding about the code we wrote. We found that all CUDA threads are independent and each of them is responsible for a separate image block. Therefore, they all need to access the memory at the same time to read the reference pixels to predict, write the predicted pixel block back, and then read both the original and predicted pixel block to compute satd values, and at

last write back the computed satd values. And in the initial CUDA code, all of those memory accesses are uncoalesced.

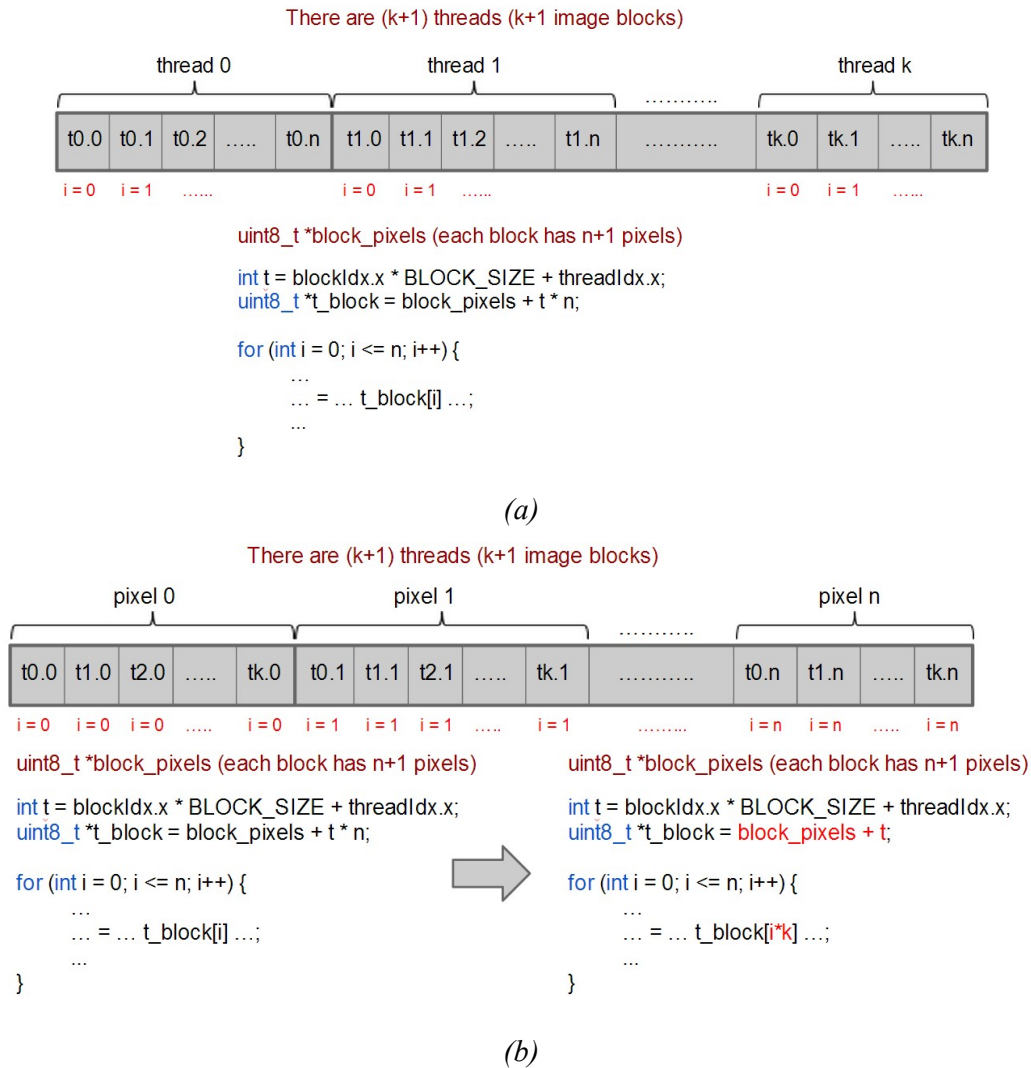


Figure 10: CUDA code example with (a) uncoalesced and (b) coalesced memory accesses

Figure 10 (a) illustrates an example snippet of the initial CUDA code with uncoalesced memory accesses. This example shows the code when all threads try to read the original block pixels from the global memory. For the simplicity and correctness of the initial parallelized code, the original block pixel's memory layout, as shown, is divided into chunks by the thread, so that all pixels used by the first thread (thread 0) are in grouped together in the first consecutive memory chunk, and then all pixels used by the second thread (thread 1) are in the next consecutive memory chunk, and so on. Therefore, in the for loop when all threads try to read each pixel to compute satd value, there is always a stride of n (number of pixels in an image block)


between the memory accesses of consecutive threads. As a result, the global memory bandwidth is greatly underutilized.

To coalesce the memory accesses, we first changed the memory layout of the block pixels. Figure 10 (b) depicts the coalesced memory layout. From the code, we observed that all threads access pixels one by one (first read pixel 0, then pixel 1, pixel 2, ...). Therefore, instead of dividing the memory into chunks by thread, we change it to be divided into chunks by pixels. So the first consecutive pixel chunk includes pixel 0 of all the threads, the second chunk contains pixel 1 of all the threads, and so on. And with the changed memory layout, the kernel is also modified to have the correct memory indexes, as is as shown in Figure 10 (b). Accordingly, whenever the threads try to read the pixels together in the for loop, they will always access consecutive (coalesced) memories.

In the end, we coalesced five global memory variables: reference block pixels (filtered and unfiltered), original and predicted block pixels, and unsorted satd values.

3.3.2 replace multiplication with addition operations

When we start coalescing our memory accesses, a great improvement of performance is observed as expected. However, after we finish coalescing all the memory accesses, the performance is not as good as when we only coalesce a few accesses. Shocked by this unexpected result, we investigated our coalescing algorithm and find out that we have introduced many multiplications.



```
for (int i = 0; i <= n; i++) {  
    ... = ... t_block[i*k] ...;  
    ...  
}  
  
int idx = 0;  
for (int i = 0; i <= n; i++) {  
    ... = ... t_block[idx] ...;  
    ...  
    idx += k;  
}
```

Figure 11: Example of using addition to replace multiplication operations

Knowing that doing multiplication is quite expensive, we began investigating ways to avoid using it while keeping the coalesced memory access pattern. By observation, we found that the multiplication we used can be easily substituted using addition, as shown in Figure 11, to greatly boost performance. Since addition is much cheaper than multiplication, fully coalescing the memory accesses had finally brought us the expected amount of improvement after we implemented this optimization.

3.3.3 pack 2 satd value calculation together

Initially SATD calculation takes two for-loop to compute, and since this function is called many times(each thread has to call it for each prediction mode for each block), we realize that any optimization done on this function would improve the overall performance substantially. We found that due to the nature of the computation, we could pack two computation together by utilizing the low-level support of C language. As a result, since our color has values represented in unsigned 8-bit integer, the sum should use a 16-bit data type. We have made another variable in the type of 64-bit unsigned integer so it could hold two sums concurrently, one in its higher 32bits and the other in its lower 32 bits. Therefore, two calculations can now happen simultaneously as we now has two numbers “packed” into one. This way, we have reduced the number of loops during each SATD calculation to one.

4. Results

After applying all the optimizations, we measured the run time of reference code and our parallel code for different sizes of input image:

Input size	160*90	320*180	640*360	1280*720	1920*1080	3840*2160
run time of reference	0.0115	0.049	0.200	0.803	1.815	7.302
run time of our code	0.413	0.048	0.063	0.111	0.196	0.583
speedup	0.28	1.02	3.17	7.23	9.26	12.52

We calculate our run time by putting a cudaFree(0) out of competition code. Because the first time when we access the GPU device, there is an initialize time, which may take 1.3 second. This additional overhead will make our run time for the first frame inaccurate. So we put cadaFree(0) outside ece408_competition() to calculate the run time more accurately.

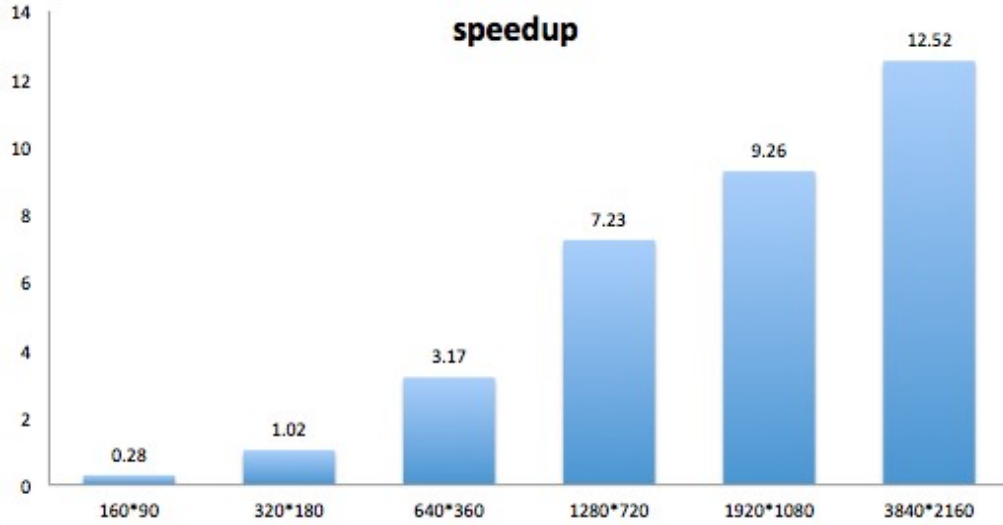


Figure 12: Speedup for different input image size

From the above plot, we can see that when the input image size is small, there is no speedup at all. The reason is that CPU can finish the work very quickly when the input image size is small. While copying data to and from device, launching kernel has some overhead, we can't get any speedup for small input image. As the input image size increases, we can see that the speedup is also increase. For input image size of 3840*2160, the speedup is 12.52.

We also tried to adjust the number of threads in each block. And we found out that 256 is the best configuration. For input image size of 3840*2160, we run our code for 128, 256, and 512 threads in each block.

<i>block size</i>	<i>Run time</i>
128	0.66 s
256	0.58 s
512	0.66 s

5. Challenges

The first challenge we encountered is to produce a serialized code. Since we were not familiar with x265 standard, it took us quite a long time to figure out the what we need to do. When it comes to implementing a serialized code, we found it difficult to do reverse engineer, because the reference code is sometimes obscure for us to understand. It took us more than one week to produce the serialized code and therefore reduced the time we could spend on the parallelization.

The second challenge we encountered is that the reference code keeps changing. At some point, our code could produce the same result as the reference code. However, when an update of reference code occurred, we needed to merge our code manually, for we don't have any strategies to merge code automatically. Sometimes, after merging the code, we found out that our result no longer matched the reference result. In such situation, we didn't know if it was the change of reference code, or our manual merging that caused the mismatch. It would save us a lot of time if the reference code could be fixed at the beginning.

The third challenge we encountered is that sometimes, our submitted job maybe killed. For 90% of the time, our code can run to the end and produce the correct result. However, the job maybe killed occasionally. Much work is put into figuring out the reason and producing a 100% reliable code.

6. Troubleshooting and debugging

Initially, we were printing all the relevant data to the terminal. We need to comment some `printf()` functions back and forth, recompile the code, run the code for multiple times, and compare the output manually. Then we developed a systematic strategy that simplify the process: print the result of reference code and the result of our code to separate files to compare. By doing this, we don't need to comment the `printf()` functions back and forth. Since the reference output will only print to the reference output file, and our code will print to another file. We only need to run the code once, and can do a "vimdiff" to compare the files and find the mismatch instead of finding the mismatch manually.

We have many printing functions that print the intermediate results to file:

- `printin(ece408_frame frame)` will print the input YUV pixels to file. This function helps us to know exactly what the input image is.
- `fprintbefore(FILE* com, uint8_t* ref_above, uint8_t* ref_left, int luma_size)` will print the above and left reference pixel to file "com". This function helps us to make sure that we are preparing the reference pixels correctly.
- `fprintafter(FILE* com, int channel, int cublk_index, int luma_size, uint8_t* pred_block)` will print the predicted pixels to file "com". This function helps us to make sure that we are predicting the pixels in each mode correctly.
- `printout(FILE * outputframe, ece408_intra_pred_result* myref)` will print out the sorted satd values and corresponding modes to file "outputframe". We will call this function twice: one with the

reference result, and one with our result. By comparing the two files, we can know if our result matches the reference code.

After sorting, if our result does not match the reference code, it will be very difficult to find the mismatch. Therefore, most of the time, we will comment out the “sort” part of reference code and our code, and call `printout()` function so that we can identify the mismatch more easily since all modes are in increasing order. When the unsorted code matches, we will add the “sort” back.

6. Conclusion

In this project, we studied, implemented, and parallelized the HVEC intra-prediction algorithm that computes and sorts SATD values of the 35 modes. We started the project by writing a serialized code, making sure it works, and then implemented the GPU parallelized code, making sure it works as well. With the initial result from the parallelized version, we tried to analyze the bottleneck and the reason of the non-ideal result. Then we applied different CUDA optimization techniques, such as memory coalescing, to tackle those bottleneck places and in the end achieved non-trivial speedups (~10X for large images, e.g. 3840x2160) when compared with the reference code. During this process, we encountered some challenges, such as learning the HEVC standard and dealing with changing reference codes. But at the same time, these challenges also helped to expose us to the real engineering industry world, where the specifications and user requirements may change back and forth along the development process. So to deal with those challenges, we developed some systematic approach to print out different debugging information. Overall, this project is a painful, but enjoyable and fruitful learning process for us.

7. Future Work

7.1 Handle small input image size

Our parallel code has no speedup for small input image size. To address this problem, we could pack multiple images together if the input image size is small. Previously, each small input image will call a kernel invocation and the cost is high. However, if we pack multiple input images together, we only need to launch one kernel for multiple images and therefore reduce the overhead for launching kernel. We might get more speedup for a small input image size with a large number of frames.

7.2 Software pipelining:

Currently, there are no parallelism between different frames. For a single frame, we prepare reference pixels, copy host data to device, launching kernel, and copy data from device to host serially. We could actually overlap these works: when we are preparing reference pixels and copy host data to device for frame number n , we can launching the kernel for frame $n-1$, and copy data back from device to host for frame $n-2$. We can potentially get more speedup if we adopt this software pipelining technique.

References

[1] <Project Kickoff reference omitted>