Parallelizing H.264 Motion Estimation Algorithm using CUDA

Lawrence Chan, Jae W. Lee, Alex Rothberg, and Paul Weaver
Massachusetts Institute of Technology

Abstract

Motion estimation is currently the most computationally intensive portion of the H.264 encoding process. Previous attempts to parallelize this algorithm in CUDA showed sizable speedups but sacrificed quality by ignoring the motion vector prediction (MVP). We demonstrate the viability of a hierarchical (pyramid) motion estimation algorithm in CUDA. This solution addresses the MVP problem while still taking advantage of the CUDA framework.

1 Introduction

H.264 (MPEG-4 Part 10/MPEG-4 AVC) is a state-of-the-art video compression standard. H.264 is used in many video storage and transmission applications like on Apple’s iPods. Compared to the previous-generation standards such as XviD and DivX, the compression ratio of H.264 is reported to be about 2× higher. However, a lot more computation power is required to encode the video, and it often takes hours to encode DVD-quality video clips in H.264 even on a high-performance desktop machine.

This project presents accelerating H.264 video encoding using NVidia’s CUDA language and tools on general-purpose graphics processing unit (GPGPU). To achieve best speedup, the motion estimation function, which is known to be the most computationally intensive section of the code, was parallelized. Figure 1 shows execution time breakdown for H.264 encoding, and the motion estimation algorithm accounts for as much as 86% of the total execution time.

A motion estimation algorithm exploits redundancy between frames, which is called temporal redundancy. A video frame is broken down into macroblocks (each macroblock typically covers 16x16 pixels), each macroblock’s movement from a previous frame (reference frame) is tracked and represented as a vector, called motion vector. Storing this vector and residual information instead of the complete pixel information greatly reduces the amount of data used to store the video. Among many motion estimation algorithms, we adopted the pyramid (or hierarchical) search algorithm, which is more suitable for parallelization than other algorithms by breaking dependency chains. Our CUDA implementation resulted in a 56× speedup of the pyramid search compared to a CPU-only reference code implementing the same algorithm.

2 Background

In this section, we first present an overview of the H.264 video compression standard, followed by an explanation of the GPGPU/CUDA platform.

2.1 H.264 Overview

H.264 (and other video codecs) are based on three redundancy reduction principles: spatial redundancy reduction, temporal redundancy reduction, and entropy coding. Figure 2 illustrates spatial and temporal redundancy. As shown in Frame 1, a block usually has a similar pattern with its neighbors (spatial redundancy), and this redundancy can be reduced by predicting the pixel values of one block from its neighboring blocks and compensate errors with residuals. This block is called intra-coded block. Intra-coded blocks do not require much computation, but the compression efficiency is relatively low.

On the other hand, temporal redundancy is reduced by motion estimation. In video, pixel values often do not change much across scenes (frames), and the pixel values of a block can be predicted from the values of a similar block in a previous frame (reference frame). The motion estimation is an operation of finding a motion vector, which indicates the similar block in the reference frame, and residuals to compensate prediction errors. To find the motion vector, the motion estimation algorithm finds the best match in a certain window of the reference frame. The block coded with motion estimation is called inter-coded block. Inter-coded blocks yield a high compression ratio, but require heavy computations to find the best match.

Finally, the entropy coding further reduces the encoded video size by assigning more frequently appearing symbols to shorter codes. In the H.264 standard, Context-Adaptive
Variable-Length Code (CAVLC), Context-Adaptive Binary-Adaptive Code (CABAC), and Exp-Golomb code are used.

Figure 3 shows a block diagram of H.264 encoder. The blocks in the shaded region are used to find a good prediction to the current input block, either from previous frames or from neighboring blocks in the same frame. The entropy coding is performed right before the final output. As said, the execution time of encoding is often dominated by the motion estimation algorithm, which will be our target for parallelization using CUDA.

### 2.2 GPGPU/CUDA

Traditionally, CPU has been optimized to maximize the performance of a single-thread execution, and GPU to achieve a high throughput for a small number of fixed graphics operations. GPUs were generally good at processing a program with rich data parallelism, while CPUs were good at handling a program with irregular parallelism. However, recent trends show convergence of both architectures–more energy-efficient throughput computing on CPUs and better programmability on GPUs. Figure 4 explains this trends. These days, such GPUs are called general-purpose GPU (GPU) because they can be used for non-graphics processing.

Still, the programmability is one of the biggest challenges in GPGPU computing, CUDA is an extention to C language to ease parallel programming on NVidia’s GPGPUs. We do not cover the details of CUDA in this paper, but interested readers are referred to [5].

### 2.3 Previous Implementations

#### 2.3.1 x264

x264 is an open source encoder for H.264 written in C and hand-coded assembly. It is considered the fastest encoder available and as of August 2008, x264 implements more H.264 features than any other encoder[1]. It has been opti-
mized to take advantage of MMX, SSE2, SSE3 and SSE4 instructions. It has multithreaded support, although the threading it uses is on the order of tens of threads. While it is fast compared to other H.264 encoders, it is still slow compared to the last generation encoders such as XviD and DivX. This has lead to a delayed adoption of the H.264 standard.

2.3.2 CUDA Attempts

There have been several articles published on using CUDA (or previously GPGPU code) for H.264 encoding. All of these approaches targeted motion estimation portion of the encoding process for parallelization. In [2] for example, the authors were able to achieve over 10 times speed up in the encoding process when using CUDA for motion estimation.

In [6], the authors were able to achieve 1.47 times speed up, however they acknowledge “A serial dependence between motion estimation of macroblocks in a video frame is removed to enable parallel execution of the motion estimation code. Although this modification changes the output of the program, it is allowed within the H.264 standard.”

3 Implementation

3.1 Motion Estimation Algorithms

Motion Estimation is the process of finding the lowest cost (in bits) way of representing a given macroblock’s motion.

An example of entropy encoding that will cause problems when we attempt to perform motion estimation in parallel is the transmission of the motion vectors. Similar to the ideas of spatial redundancy described above, nearby areas are more likely to move in the same direction than in different direction. We can use this fact to aid in the compression process. Rather than transmitting the entire motion vector for a given macroblock, we only need to transmit how it differs from what we predict it would be, given how the blocks around it moved. The “blocks around it” are defined as the macroblock above (1), to left (2), and above and to the right of the given block (3). Our motion vector prediction (MVp) is the median of these three vectors. We only need to transmit how a given block’s motion vector differs from the MVp. Using entropy encoding we can then more efficiently encode this differential.

The “cost” takes into account the cost of encoding the residual (usually approximated using the Sum of Absolute
Differences (SAD) as well as the cost of encoding the motion vector. This cost calculation usually looks something like:

\[
\text{cost} = \text{SAD} + \lambda \left[ C(MV_x - MV_{px}) + C(MV_y - MV_{py}) \right]
\]

where \( \lambda \) is an empirically determined value and \( C(.) \) is the cost of encoding a motion vector differential of that length.

In order to solve the problem of performing motion estimation in a massively parallel environment we needed to deal with the issue of needing the MVp to find the optimal motion vector. Simply ignoring this component of the cost calculation will lead to larger files in order to maintain the same level of quality. The H.264 specification deals with how the decoder functions so it leaves many of details of the encoder to the implementer.

### 3.2 Motion Estimation in CUDA

We considered two solutions to solving the motion estimation on a massively parallel scale: (1) deal with the dependency between blocks and (2) approximate the MVp. The first approach, a common technique in parallel programming referred to as “wavefront” iterations, involves analyzing which macroblocks do not have inter-dependencies and then executing these blocks simultaneously. While this solves the dependency problem and still allows some level of parallelization, it significantly limits our ability to take advantage of the CUDA hardware. In the case of HD video (1080p), at its widest point, this approach would allow us to process only 34 macroblocks in parallel. In total it would require 306 iterations. The other approach was to give up on strict dependency and attempt to approximate what the MVp would be. While this will cause a slight decrease in encoding quality, it could greatly improve our cost estimates (over ignoring this component of the cost).

We chose to solve the problem of needing the MVp’s in order to properly estimate the cost of a given motion vector by using an estimate of what the MVp will actually be. In order to estimate what the MVp would be for a given region, we used pyramid (or hierarchical) motion estimation. This process involves performing motion estimation on a significant downsampled version of the image. The vectors found in this iteration are used as estimates for the MVp’s in motion estimation of a slightly less-downsampled image. This process is repeated until the motion estimation is performed on the full-resolution image.

Our implementation started at a level of sixteen times downsampling and doubled the resolution to eight times. It continued doubling until the motion estimation was performed for the full resolution image. For the purpose of testing we performed the downsampling on the GPU.

We executed one kernel per level of hierarchy. After the kernel was done executing the motion vectors found are left on the device for the next kernel call. This allows us to minimize the number of host-device memory transfers needed.

We assigned one block per macroblock. We had one thread check each search position within the search window. By having one block per macroblock we were able to take advantage of thread synchronization to perform an

---

\[\text{The actual MVp is needed for the final encoding stage, but this can be obtained in a linear (raster-order) operation once all of the motion estimation is complete.}\]
argmin reduction to find the lowest cost motion vector for each macroblock. In the future, using one block will allow us to use shared memory to preload the pixel values that we will be using. The 512 thread limit on the earlier CUDA-enabled cards does impose a limit on the window size (the window must be $\leq 11$). This problem can be addressed either by using newer cards or by having each thread check more than one position.

While our current implementation does not take advantage of shared memory to preload the needed pixel values, we do store both the current frame and the reference (previous) frame in texture memory. This helps to avoid global memory access. It also allows for cheap sub-pixel interpolation (searching for non-integer motion vectors) because the texture provides interpolated access in hardware. Further, the H.264 specification states that when searching for a block’s motion vector, the area beyond the edge of the frame is to be treated as the edge duplicated to infinity. We get this behavior for free from textures with clamping enabled.

4 Evaluation

4.1 Test Infrastructure

Because H.264 is a decoding standard, we have some degree of flexibility in the encoding process and there is no gold standard that is necessarily correct. Therefore, we need to verify that our results are acceptable by some other means, and one way to do this is to compare the encoded video with the original. We can calculate objective measures such as the peak signal-to-noise ratio (PSNR), and we can also look at subjective differences between the two videos.

In this work, however, we are not emphasizing the quality of the encoded video, but rather the viability of a CUDA-enabled motion estimation algorithm. To check that the results are good, we apply our motion estimation code to a video and compare the results numerically with both x264 outputs and our own gold standard outputs. We also output the motion vector fields as an overlay on the frames to visually judge the outputs. It is common for the outputs to differ numerically but still be close to optimal, so it is important to look at the visual output in judging the algorithm.

To facilitate easier plug-and-play motion estimation algorithm swapping, we wrote python wrappers to allow us to generate visual output from our C code. Our main python test bed takes care of reading the video files, generating the luminance data, sending this data to the motion estimation code, and producing the visual output with matplotlib. Since it is written in python, it is very flexible and it is simple to add extra plots, images, or statistics.

Once we verified that our results were good, we timed our code using the built-in CUDA timer.

4.2 Results

<table>
<thead>
<tr>
<th>Code</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gold</td>
<td>203.3</td>
</tr>
<tr>
<td>CUDA</td>
<td>3.6</td>
</tr>
<tr>
<td>x264</td>
<td>11.6</td>
</tr>
</tbody>
</table>

Figure 5: Results

In 4.2 we see that the CUDA code ran 56 times faster than the Gold code (written in C). It is not appropriate to compare the CUDA time to the x264 time as x264 is performing a more accurate search.

5 Conclusion and Future Work

H.264’s motion estimation in CUDA is viable, but will not be easy. Any attempt will need to be enough faster than CPU implementations to justify any sacrifices in quality. Further, the current CPU code is very well-written and highly optimized. At this point it seems that effort should be solely focused on the motion estimation segment of the encoding process. It represents the majority of the encoding time and is most suitable for parallelization. The other parts of the encoding process have complex control flows and many serial dependencies making them poor choice for CUDA.

The purpose of this project was as a proof of concept that motion estimation in CUDA was viable. It did not seek to implement the full set of motion estimation feature that an encoder, such as x264, uses. A lot is needed to make CUDA motion estimation ready for widespread use in encoding. The most important task is improving the estimate for the MVp. If the estimates for the MVp are too far from the actually values, any increase in speed will not justify the loss in quality. There is also a lot of room to pipeline the data transfers. We can begin transferring the content for the next frame to the graphics card while the current frame is encoding. There is also the question, which we did not answer, as to whether the down sampling should be performed on the GPU or on the CPU. While two sequential frames cannot be encoded in parallel due to the need to have the previously encoded frame available during motion estimation as a reference frame, frames reasonably far apart can be encoded in parallel. This would allow us to achieve higher occupancy on the GPU.

3Test were conducted on a video with 384 x 288 resolution. Pyramid search was only done with 2 level of hierarchy.
While working on this project we found CUDA is great for what it attempts to do: opening up the GPU for general purpose calculations. In less than a months time we were able to accomplish quite a lot. We did find the documentation was sparse, NVIDIA provides three documents, and in many cases were left looking through SDK code examples. The forums though helpful usually required a lot of reading to find the correct solution to a given problem. In some cases there was no clear consensus on what the correct solution was (or even what the trade offs between the various options are). Emulation mode was great for debugging, but in many cases it was prohibitively slow to run the code in it. Further, while CUDA is marketed as rack-mounted approach to super computing, we often found out node locked and required a system reboot (super user access needed).  

Acknowledgement

Dark_Shikari (x264 developer) and various other people in #x264 channel @ Freenode.net. Youngmin Yi at UC Berkeley.

References


---

4This might have been due to using the beta drivers. We did not test this problem on other systems with release drivers