Use of CUDA Streams for Block-Based MPEG Motion Estimation on the GPU

Mai H. El-Shehaly
Virginia Tech

Denis Gračanin
Virginia Tech

Hicham G. Elmongui
Alexandria University

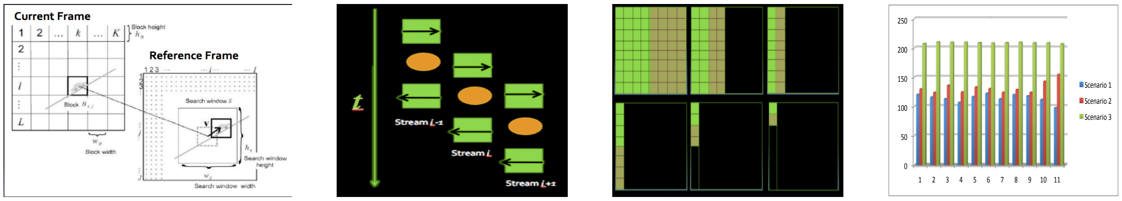

Figure 1: *(a) Block-based SAD calculation (b) Scenario 1: memory transfers (rectangles) and GPU processing (ovals) (c) per block 2D Parallel Reduction to find minimum SAD (d) run times for 3 scenarios.*

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors;

Keywords: MPEG-4, Motion Estimation, GPU computing

1 Introduction

The H.264 standard of MPEG-4 involves motion estimation that takes about 91% of encoding time. Luckily, the problem of block-based motion estimation is highly parallel. Motion vectors are calculated by determining block displacement within an area, typically 32x32 pixels, in a known reference frame. We enhance the GPU-based Sum of Absolute Difference (SAD) calculations of motion estimation, through the use of CUDA streams to hide memory latency by means of different overlapping techniques. A novel implementation strategy is explored that takes advantage of the amount of shared memory available in GPU devices of compute capability 2.x.

2 Our Approach

Each frame in a YUV video sequence, of size $M \times N$ is streamed from page-locked host memory to GPU device, where a grid of $M/4 \times N/4$ blocks of threads is created. Each block contains 32×32 threads (a total of 1024 threads per block). Each block loads from device global memory to the block's shared memory: 4×4 pixels from the current frame and 32×32 pixels from the reference frame. From this point on, all per block calculations are performed locally without having to read/write from/to device global memory. Each thread then calculates one element of a 32×32 matrix of SAD values for each candidate displacement within the search range, evaluated as:

$$SAD_{k,l}(\mathbf{v}) = \sum_{(i,j)\in B_{k,l}} |I^{t+1}(i,j) - I^{t}(i+v_1, j+v_2)|. \quad (1)$$

where $\mathbf{v} = [v_1, v_2]^T$ is the block displacement for block $B_{k,l}$, $I^t(i,j)$ and $I^{t+1}(i,j)$ are image intensities at pixel (i,j) in the reference frame, and in the current frame, respectively.

The resulting matrix is stored in shared memory, where further calculations are performed to obtain SAD values for variable block modes (4×8, 8×4, 8×8) by activating only evenly indexed threads in the x-direction, the y-direction, or both. Once all SAD values are in shared memory, the current thread block decides on the minimum SAD value, by extending the parallel reduction technique described in [Harris 2007] to 2-D.

Experimenting with the above described GPU kernels was done in three different scenarios: In **scenario 1**, memory uploads, kernel launches, and memory downloads are all grouped inside one loop over CUDA streams. Kernels are not executed in parallel. In **scenario 2**, memory uploads, kernel launches, and memory downloads are each performed in their own separate loops. No overlap occurs between memory copies and kernel execution except between the last stream's memory upload with the first kernel launch, and the last kernel launch with the first memory download. In **scenario 3**, (no CUDA streams) frames are transferred one by one to texture memory. The calling thread acquires a lock before sending a frame, and it cannot release the lock until the current frame data have been processed by all blocks in the execution grid. The efficient use of CUDA streams has significantly reduced the amount of time required to process the entire sequence. Scenario 1 performed best, because of the overlap of kernel execution with memory transfers, which is the ideal scenario for latency hiding. Scenario 2 takes on average 15% longer to execute, which can be worsened by higher resolution frames, due to more severe memory transfer latencies. Scenario 3 relies solely on GPU parallelism, and makes no use of CUDA streams. Therefore, on average it takes 83% longer to execute. Our proposed technique achieves almost 3× speedup for variable SAD calculations, over the work presented in [Chen and Hang 2008]. Such speedup could be achieved through the elimination of global memory accesses, and the use of CUDA streams. The proposed latency hiding technique can be used to speedup implementations of more MPEG-4 encoding steps, which will be the focus of our future work.

References

CHEN, W., AND HANG, H. 2008. H.264/AVC motion estimation implmentation on compute unified device architecture (CUDA). In *Proceedings of the 2008 IEEE International Conference on Multimedia and Expo*, IEEE, 697–700.

HARRIS, M. 2007. Optimizing parallel reduction in CUDA. CUDA SDK white paper, NVidia.