Efficient Kernel Fusion Techniques for Massive Video Data Analysis on GPGPUs

Asif M. Adnan and Sridhar Radhakrishnan
School of Computer Science
University of Oklahoma
Norman, USA
\{asifmadnan,sridhar\}@ou.edu

Suleyman Karabuk
School of Industrial Engineering
University of Oklahoma
Norman, USA
karabuk@ou.edu

Abstract—Kernels are executable code segments and kernel fusion is a technique for combing the segments in a coherent manner to improve execution time. For the first time, we have developed a technique to fuse image processing kernels to be executed on GPGPUs for improving execution time and total throughput (amount of data processed in unit time). We have applied our techniques for feature tracking on video images captured by a high speed digital video camera where the number of frames captured varies between 600-1000 frames per second.

Image processing kernels are composed of multiple simple kernels, which executes on the input image in a given sequence. A set of kernels that can be fused together forms a partition (or fused kernel). Given a set of Kernels and the data dependencies between them, it is difficult to determine the partitions of kernels such that the total performance is maximized (execution time and throughput). We have developed and implemented an optimization model to find such a partition.

We also developed an algorithm to fuse multiple kernels based on their data dependencies. Additionally, to further improve performance on GPGPU systems, we have provided methods to distribute data and threads to processors. Our model was able to reduce data traffic, which resulted better performance. The performance (both execution time and throughput) of the proposed method for kernel fusing and its subsequent execution is shown to be 2 to 3 times higher than executing kernels in sequence. We have demonstrated our technique for facial feature tracking with applications to Neuroscience.

Keywords—kernel fusion, GPU, parallel image processing, tracking, video analysis

I. INTRODUCTION

Application of High Speed Digital Video (HSDV) for identifying previously unseen patterns has become very popular. HSDV and its analysis have applications in industrial product development and quality control [1], Neuroscience [2], and others [3]. Boss et al. [2] used HSDV for identifying different traits of human facial action movements. One of their distinctive findings is that facial muscles on the left side (controlled by right hemisphere of brain) start moving earlier than the right side, for spontaneous facial expressions. Several such discoveries can be made possible by analyzing the data produced by HSDV.

HSDV results in massive volume of data to store and demands extensive computing power to analyze the data. A one second long video image captured at 600 frames per second (fps) (resp. 1000 fps) with frame dimensions 192 × 432 pixels (resp. 800 × 600 pixels), requires about 190MB (resp. 1.8GB) of storage. Clearly, processing HSDV data using CPUs will be challenging because of its tremendous volume. Real-time or near-real-time processing would be further challenging because of high volume data transfer requirement through the computation components (for example, disk to main memory).

In this paper, the term “Image Processing Algorithm” or IPA will be used to refer to both image and video processing algorithms. A complex IPA, Alg, can be viewed as a sequence of multiple simple algorithms, Alg = \{Alg_i : 1 ≤ i ≤ n\}. The algorithms have to be executed in a sequence on the entire video data \(I\) having dimensions \(N \times M \times T\). Here, \((N \times M)\) refers to image spatial dimension or frame dimensions, and \(T\) refers to temporal dimension or number of frames.

A general CUDA implementation of Alg will be to develop kernels \(K_i\) for each of the algorithms, \(\Lambda_i\), and execute them in sequence. In this paper, we will use the terms algorithms and kernels interchangeably. Each of the \(K_i\) will be distributed among the Streaming Multiprocessors (SMs) and results in the intermediate output. Intermediate output \(I_{\text{out}, i}\) generated by \(K_i\) will be used by the next kernel \(K_{i+1}\) in the sequence. Please note a particular GPU device will have a number of SMs and each SMs have a set number of processors and local memory (called a shared memory). Additionally, each of the SMs and its processor has access to global memory. There are no concurrent accesses to global memory, while there are concurrent read on shared memory by processors within a SM. Thus, keeping as much of data as possible closer to the processors (e.g. shared memory) is better for faster data access.

Image processing algorithms are memory access intensive. A simple sequential execution of kernels \(K_1, ..., K_n\) will result in increased data traffic among different GPU memory units (global and shared), as well as, it will use global memory to store generated intermediate data. The overall performance is related with the volume of data traffic between global and shared memories. Performance will increase with the decrease of data traffic and reduction in global memory usage.

The overall structure of our technique to improve execution time on a set of image processing kernels is as follows. First, we will determine the data access patterns of the given set of kernels. The data access patterns will provide information on the amount of data that needs to be transferred from global memory to shared memory for
execution. Second, based on the data access patterns of each of the kernels, we will partition the set of kernels, where each partition will result in a fused kernel. The set of fused kernels are executed in a sequence consistent with the sequential execution of the set of kernels. Among all possible partitions, our goal is to determine a partition that will reduce the overall data traffic between global memory and shared memory. In order to determine this partition, we have developed a optimization model and have used the Gurobi [4] tool to implement the model. We also would like to point out that once a partition is determined, we have to provide a mechanism to perform the fusing. We have provided one such technique in this paper. Third, once the fused kernel is formed, we will provide techniques to allocate resources (threads, processors, shared memory, global memory) optimally to reduce the total execution time of all fused kernels.

In section II we presented a brief overview of CUDA enabled GPU architecture. Typical image processing algorithm and its implementation on GPU is presented in section III. In section IV we have discussed the data access patterns that are common in image processing applications. Once the threads have been allocated for executing the kernels, there will be data dependencies among the threads and these are discussed in section V. We presented our optimal kernel fusion model and in section VI. In the same section we provided a mechanism to perform the fusing once the set of kernels chosen to be fused have been determined. We also showed how the performance of a fused kernel can be improved by efficient resource allocation and data distribution. Experiments and analysis of the proposed techniques for facial feature tracking is discussed in sections VII and VIII respectively. We discuss this work and compare with the other kernel fusion techniques in section IX and conclude in section X.

II. CUDA Architecture Overview

GPU architecture generally consist of multiple Streaming Multiprocessors (SM), different levels of memory units. Each SM has multiple processors (cores), local memory unit, such as Shared Memory (SHMEM), and registers. Computation heavy and parallelizable tasks are transferred from CPU to GPU, and executed as set of instructions.

CUDA provides a general-purpose parallel programming model which extends C and defines C-like functions, called kernels. Kernels are executed in parallel by different threads on the GPU. Threads are grouped together in blocks, which can be either one, two or three dimensional. All the blocks of threads, form a one, two or three dimensional grid. Threads within a block can co-operate among themselves through the SHMEM and synchronize their execution to coordinate memory accesses. Threads are managed and executed by the multiprocessor in groups. This group is called warp. The usual warp size is 32 threads.

Data communication between CPU and GPU are conducted through the Global Memory (GMEM). All threads in a SM operate in a Single Instruction Multiple Data (SIMD) fashion, while threads belonging to two or more SMs operate in a Single Instruction Multiple Thread (SIMT) fashion. Data from GMEM is accessed by threads for computation. The result of the computation is written back to the GMEM as dictated by the algorithm. SHMEM access is a couple of magnitude faster than GMEM access. Before any algorithm executes, it is desirable to bring the data to SHMEM. The limited space of the SHMEM is a bottleneck to this approach. A more detailed description of CUDA enabled GPUs can be found in [5].

III. Image Processing Kernels

In this section we will present a general image and video analysis application using CUDA enabled GPU devices. A typical feature tracking IPA on a set video frames consist of the application multiple kernels. These include kernels for reducing noise, enhancing pixel quality, image transformation say from RGB to Gray-scale, identifying features (using say Gaussian Filter and Gradient Smoothing), and finally tracking them on video data. Noise reduction and pixel enhancement kernels takes as input raw image data and generates better quality data. The output of this stage is used by other kernels (image transformation, Gaussian filter, gradient smoothing filter, and others) to detect the features. Finally different particle filter (such as the Kalman filter) algorithms are applied to track the features in different frames of the video data.

We modeled a $T$ second long video having $N \times M$ dimensional frames, as, $l([d_x, d_y, d_t])$, where, $d_x = N, d_y = M$, and $d_t = T$. Video segments also have frame rate represented as $R$ frames per second. A $T$ second long frame with $R$ frames per second has total, $F = T \times R$ frames. A pixel of a frame $l([d_x, d_y, d_t])$ at location $(i,j)$, will be represented as $l([i,j])$.

Let's say that a complex algorithm $Alg$ consists of a set of algorithms, $A = \{A_i, 1 \leq i \leq n\}$. A very general implementation of $Alg$ on CUDA devices would be to develop single kernels $K_i$ for each of the algorithms $A_i$. When the kernels $K_i$, $K = \{K_i, 1 \leq i \leq n\}$ finishes execution, the complex $Alg$ completes execution. Fig 1 shows multiple kernels in sequence and along with their data dependencies.

For a sequence of kernel executions, $K_1, K_2, ..., K_n$, processing $l_{in}([d_x, d_y, d_t])$, we get, $l_{out_{K_1}}([d_x, d_y, d_t]) = K_1(l_{in}([d_x, d_y, d_t]))$ $l_{out_{K_2}}([d_x, d_y, d_t]) = K_2(l_{out_{K_1}}([d_x, d_y, d_t]))$ ...

$\ldots$ $l_{out_{final}}([d_x, d_y, d_t]) = K_n(l_{out_{K_{n-1}}}([d_x, d_y, d_t]))$ So, the final output will be generated after executing a sequence of kernels on the given input data, as in, $l_{out_{final}}([d_x, d_y, d_t]) = K_n(...)l_{in}([d_x, d_y, d_t]))$.

IV. Data Access Patterns

Different algorithms have different data access patterns, for example, consider the algorithms mentioned in the previous section. The algorithms for noise reduction and pixel enhancement algorithms relies both on spatial and temporal neighbors of a pixel. Image transformation algorithms such as from RGB to Gray scale transformation relies on a single pixel. Please note that there exists
other transformation algorithms that requires spatial and/or temporal neighborhood values. Gaussian filter and gradient smoothing algorithms that we used for our application are spatial filters, and hence, they require spatial neighbors. The Kalman filter algorithm uses the identified feature position in multiple frames, for tracking. Such a filter relies on a single point in the spatial dimension, but also requires the temporal neighbors.

Computing the value of a pixel \( I_{out}[i, j, t] \) involves a function \( F(\text{in}[d^0_i, d^1_j, d^0_t]) \). Here, \( I_{out}[i, j, t] \) represent a single pixel, and \( \text{in}[d^0_i, d^1_j, d^0_t] \) represent the pixels neighboring to \( I_{in}[i, j, t] \). The function \( F(\text{in}[d^0_i, d^1_j, d^0_t]) \) generates the output pixel with \( \text{in}[d^0_i, d^1_j, d^0_t] \) as input.

To generate an output box (denoted Box \( b \)) of dimension \( x \times y \times t \), the input box (denoted Box \( b_{in} \)) will have dimension \( (x + \delta x) \times (y + \delta y) \times (t + \delta t) \), where \( \delta \) is the small increment in size (the larger box).

The output value of a single pixel \( I_{out}[i, j, t] \) depends on its spatial, temporal, or both spatio-temporal neighbors (Fig. 2). After analyzing commonly used image processing algorithms, we divided them into different types. Table I lists the operation types and shows their data dependency criteria. Table II classifies different simple and basic algorithms into different types based on type of operation and if it involves single or multiple frames (as in the case of video data).

Table I: Different Types of Operations

| Types of Operations          | Data Dependency |
|-----------------------------|-----------------|
| Single-Point Operation      | \(|d_i| = 1, |d_j| = 1, |d_t| = 1\) |
| Rectangular Operation       | \(|d_i| > 1, |d_j| > 1, |d_t| = 1\) |
| Single-Frame Operation      | \(|d_i| = 1\) |
| Multi-Frame Operation       | \(|d_i| > 1\) |
| Spatio-Temporal Operation   | \(|d_i| > 1, |d_j| > 1, |d_t| > 1\) |

V. THREAD DEPENDENCY AND KERNEL FUSION

Each of the kernels, \( K_i \), are executed as a grid of blocks, where each of the blocks will have a set of threads. The total execution time for executing the kernels \( K_1 \) to \( K_K \) corresponding to \( Alg \) will be,

\[
Total^T = \sum_{i=1}^{K} (T_{Access}^i + T_{Compute}^i + T_{Write}^i)
\]

where, for each kernel \( K_i \), \( T_{Access}^i \) is the data access time from GMEM to thread’s local memory, \( T_{Compute}^i \) is the time to compute accessed data, and \( T_{Write}^i \) represents the time to write/store data to GMEM.

Each of the thread block will generate output having certain dimensions. Fig. 3 is showing the division of the video data of dimension \( N \times M \times T \) into boxes \( Box_b \) of dimension \( x \times y \times t \). There will be \( B = \frac{N \times M \times T}{x \times y \times t} \) number of such boxes. As each thread block will perform computation on the data in each of the boxes. Without loss of generality we can assume that there will be \( B \) number of thread blocks. All these thread blocks will be distributed among \( SMs \) number of SMs and executed via warps (or thread groups). To generate an output of each box \( Box_b \), each thread block of the kernels has to take input a box \( Box_{in} \).

A. Thread Dependency Types

Data access patterns discussed in the previous sub-section leads to different types of thread and block-level thread dependencies. More the dependencies less will be the achievable parallelism. In our analysis we found three basic dependency types:

a) Thread to Thread (TT): This occurs when a thread \( T[x, y, t] \) of kernel \( K_i \) depends on the output of the corresponding \( T[x, y, t] \) of kernel \( K_{i-1} \). Every thread in here requires output from the corresponding thread to complete its execution. This results in
high level of parallelism or the smallest amount of dependencies. This is shown in Figure 4(a).

b) Thread to Multi-Thread (TMT): When a thread \( T \{x_i, y_i, t_i\} \) of kernel \( K_i \) depends on multiple threads of kernel \( K_{i-1} \), we say that TMT dependency occurs. A thread has to only wait for the result of the corresponding thread block to complete its execution. Since all blocks are executed in parallel, the dependency is somewhat lower.

c) Kernel to Kernel (KK): When a block of threads that is executing a kernel \( K_j \) depends on the output of multiple blocks of threads executing kernel \( K_{j-1} \), we define it as KK dependency. This is shown in Figure 4(b) As an example, consider the two kernels Center detection to identify the feature and Kalman filter kernel to track features. These two kernels have to be executed in sequence resulting in lower parallelism or higher dependencies. This is shown in Figure 4(c).

### B. Kernel Fusion

Kernel fusion is a source code transformation technique, where new kernels are created by aggregating or merging codes of multiple other kernels. Usually kernels executing on same data are good candidates for fusion.

In case of kernel fusion, a fused kernel’s total execution time will be (in contrast with equation 1), as follows:

\[
Total^T = T_{\text{Access}} + \sum_{i=1}^{n} (T_{\text{Compute}}^{T_i}) + T_{\text{Write}}^{T_i}
\]  

Here, we considered that, for a given set of kernels, \( K \), having \( n \) number of kernels, the necessary input box \( Box_{B_n} \) will be copied from GMEM to SHMEM. Now all the fused kernels will access data from SHMEM and store intermediate data in SHMEM. After the execution of the last kernel in the sequence, the final result will be stored in GMEM.

Fusing multiple kernel reduces total number of data transfers among GPU memory units. Reduction in data transfer to and from GMEM and local memory units also reduces data access from more time consuming global memory access. As the kernels \( K_1, \ldots K_n \) are generating and reusing intermediate data to and from SHMEM or local resources, GMEM is freed up for additional data to be loaded from CPU’s main memory.

### VI. OPTIMAL KERNEL FUSION

Wahib and Maruyama [6] described a technique to estimate the execution time for kernels with heavy data access on GPU systems. We will use this estimate to determine the optimal partition of a set of kernels given in sequence for execution. Each partition will be the fused kernel. For example, consider three kernels that needs to be executed in sequence as follows: \( K_1, K_2, K_3 \). The possible partitions are \( (\{K_1\}, \{K_2\}, \{K_3\}), (\{K_1, K_2\}, \{K_3\}), (\{K_1\}, \{K_2, K_3\}) \). The partition with a minimum total execution time is selected to be the optimal one, where the execution time is determined using the technique in [6].

### A. Identify Fusable Kernel Sets

For a target task, we decompose the task into multiple kernels. After analyzing the thread and block level
dependencies, we generate multiple sets, \( K_1, K_2, \ldots, K_m \) of fusible kernels, where, \( K_k = \{ K_a : 1 \leq a \leq n \} \).

For generating possible fusible kernel sets, we considered kernels having only Thread to Thread and Thread to Multi-Thread dependency in the sequence. We excluded \( KK \) dependent kernels. We developed the following model for kernel fusion, for each set of fusible kernels and was solved using the Gurobi

\[
\text{Number of Algorithms in a fusible set}, K_k, \{A_1, \ldots, A_j\} : n
\]

\[
\text{Number of Possible fused kernel combinations}, \{K_1, \ldots, K_j\} : f = \frac{n(n+1)}{2}
\]

\( a_{ij} \) is a vector, having \( j \) number of elements, \( i = 1 \) and \( 0 \), otherwise. \( X_i = 0,1 \) where, \( X_i = 1 \), if kernel \( K_i \) is selected. Otherwise, \( X_i = 0 \).

\( C_i \) is the predicted execution time for executing \( K_i \).

**Objective:** minimize \( \sum(X_i \cdot C_i) \).

**Figure 5:** Modeling the Objective Function

### B. Algorithm to fuse selected kernels

In the previous step, we generated sets of kernels to be fused. Algorithm 1 will take input as a set of kernels and generate a fused kernel \( K_f \).

**Algorithm 1:** Fuse kernels \( K_1, K_2, \ldots, K_n \) into \( K_f \)

**Data:** Set of kernels, \( K = \{ K_i : 1 \leq i \leq n \} \)

**Result:** Fused kernel \( K_f \)

1. **Copy Input Box**, \( B_{in} \) from Global to shared memory
2. **for each** \( K_i \) **do**
   3. Convert Global Memory access into Shared Memory access.
   4. Insert instruction of \( K_i \) into \( K_f \)
   5. if \( K_i \) and \( K_{i+1} \) are Thread to Multi-Thread dependent then Insert Synchronization.
3. **end**
4. **Copy/Store SHMEM data into GMEM.**

Line 1 of Algorithm 1 copies the required input box from the GMEM to SHMEM. Conversion of GMEM access to local or SHMEM access is explained in the next sub-section. Adjustment of input and output box dimensions are very important and is explained in a subsequent sub-section.

Local synchronization primitives are inserted in line 5 that is based on thread to multi-thread dependencies that are observed. Table III is showing two different types of fusion for three sample kernels. Here, the kernels are synthetic kernels, but has are very close to the ones used in our experiments.

| Table III: Simple and Fused Kernel Samples |
|--------------------------------------------|

RGB2Gray() and Threshold() are two kernels. Threshold() has thread to thread dependency on RGB2Gray(). RGBFusedTh() is the kernel generated by fusing these two kernels. The individual kernel K-Spatial() has the thread to multi-thread dependency on RGB2Gray(). When we fuse these two kernels, we get the kernel RGBFusedK-Spatial(). In both the fused kernels, we calculated the pixel index only once for each thread block to further improve performance. At the very beginning, we copied all the necessary pixels from GMEM to SHMEM. The set of instructions were inserted from corresponding kernels in order, after converting GMEM accesses to SHMEM access (more on this in the next sub-section). After the execution of all the instructions, the fused kernel copies/stores SHMEM data into GMEM.

Let's consider the case wherein the fused kernel \( K_f \) will be executed by a grid of \( B = \frac{N_{x,y,t}}{x \times y \times t} \), and each block will be executed as a set of threads, having \( T_h \times T_y \times T_h \) threads. To access any GMEM element, both block-offset and thread-offset are necessary. Note that block-offset \( = F \text{unctionOf} (\text{BlockID}) \) and thread-offset \( = F \text{unctionOf} (T \text{hreadID}) \).

When, we convert any global access to local shared
memory access, we can omit the block-offset, because all the necessary data is copied into SHMEM from the GMEM. Table III shows examples of such access conversion.

C. Data Distribution for Executing a Fused Kernel

In the previous section, we showed that, to generate an output box Box b (by a thread block T b), it takes as input the box Box b n . For computation on a box Box b with dimensions x × y × t , the dimensions of the input box Box b n will be (x + δ x ) × (y + δ y ) × (t + δ t ) (Fig 6). Algorithm 2 will generate the size of the input box Box b n for a given execution sequence of n kernels K i .

![Figure 6: Input Output Data dependency for generating each Box Box b. Each corresponding kernel will input a box Box b n and generates the box Box b. Dimension of Box b is x × y × t, and dimension of Box b n is (x + δ x ) × (y + δ y ) × (t + δ t ).](image)

**Algorithm 2:** Find input size for generating output image box Box b

| Data: Set of kernels, K = {K i : 1 ≤ i ≤ n} |
|--------------------------------------------------|
| Result: Size of Input Box Box b n |
| 1. Set the dimensions of Box b n as [x, y, t] |
| 2. δ x n = 0 |
| 3. δ y n = 0 |
| 4. δ t n = 0 |
| 5. For each K i (K 1 , ..., K n ) do |
| 6. If one thread in K i requires |
| 2(δ x i + 1) × 2(δ y i + 1) × δ t i , pixels, |
| 7. If (δ x i ≥ δ x n ) δ x n = δ x i |
| 8. If (δ y i ≥ δ y n ) δ y n = δ y i |
| 9. If (δ t i ≥ δ t n ) δ t n = δ t i |
| 10. end |
| 11. The dimensions of Box b n will be |
| [x + δ x n , y + δ y n , t + δ t n ] |

Algorithm 2 considers the data dependency parameters for all the kernels K 1 , ..., K n which are fused in K f to generate the size of the desired input box Box b n . If we do not adjust the input boxes, threads computing the boundary values, will be required to access data from outside of the thread block. As CUDA devices do not allow blocks to share data with other blocks, this will lead to accesses to data GMEM. Our algorithm for computing the size of the input box allows to distribute the boxes such that, no thread has to depend on threads in other blocks.

D. Fused Kernel Reduces Data Traffic

In this subsection, we have shown that the fused kernel has advantage over executing them in a sequence. Let N × M × T be the size of the input that is distributed into x × y × t dimensional boxes. Each such box, Box b is processed by a corresponding thread block, T B b . The number of boxes, B = x × y × t . For n number of kernels, K 1 , ..., K n , total data transfers, will be T transfers = 2 × N × M × x × y × t .

In case of n kernels fused into one kernel, number of thread blocks required is B. In that case, there will be total, Transfers fused = 2 × B × x × y × t + (x × δ x + y × δ y + δ x × δ y ) (t + δ t ) number of transfers.

E. Fused Kernel’s Data Utilization and GPU Occupancy

Data utilization is a measure to determine the amount of space occupied in the shared memory. Since shared memory access if faster it is useful to achieve higher data utilization. The data utilization of each thread block is computed as follows:

\[
\text{DU} = \frac{\text{Output}}{\text{Input}} = \frac{x \times y \times t}{(x + \delta x) \times (y + \delta y) \times (t + \delta t)}
\] (3)

From our observation, we have found that the data utilization will be high when x × y × t is higher. In other words, if we can use maximal SHMEM, we can achieve higher data utilization.

In order to maximize data utilization, it is required to maximize DU A in equation (3). As, x, y are both referring to spatial dimensions, without losing the generality, we can assume x = y . Hence,

\[
\text{DU}_K = \frac{x^2 \times t}{(x + \delta x)^2 \times (t + \delta t)}
\] (4)

As, x × y × t ≤ β Shared (where β Shared is the size of SHMEM), x^2 × t ≤ β Shared . Now maximizing DU K will be equivalent to minimizing

\[
V = (x + \delta x)^2 (t + \delta t)
\] (5)

Solving the equation 5, we found,

\[
x = y = 3 \frac{2 \beta \delta x}{\beta_1 \delta t_1} = 2 \frac{\beta}{\delta x} \left( \frac{\delta x}{\beta} \right) ^ {2/3}
\] (6)

The values found in 6 will provide the minimum value for V . Using these values, our algorithm will decide the size of each window for distribution.

Fig 7 shows the variation of data utilization in different CUDA devices, for different data box sizes. The limited size of SHMEM in different devices is imposing restriction on the achievable data utilization value. K20 and Gtx-750 devices has same maximum amount of SHMEM, where as C1060 allows lesser amount of maximum SHMEM. Not all combinations of data box sizes will result optimal data utilization.

An important measure that is commonly used for compute intensive GPU tasks is GPU occupancy. A high GPU occupancy indicates a high level of parallelism. More formally, GPU occupancy is the ratio of the number of thread blocks (or wraps) and maximum number of thread blocks (or wraps) per streaming multiprocessors. Increasing
GPU occupancy will result in reduced SHMEM space for each of the blocks. Our video analysis algorithms are less compute intensive and requires more memory accesses. It is advantageous to keep have more SHMEM per block and hence we have to reduce GPU occupancy.

VII. EXPERIMENTAL SETUP

A. Data set we used

Ross et al [2] used HSDV to identify facial actions in posed and spontaneous emotional expressions. We used the same data set for our experiments and implemented parallel CUDA kernel mimicking their manual tracking process. Fig. 8 is showing a frame from one of the samples. Fig. 8b is showing the selected rectangles containing the target objects (external markers).

![Frame with external markers](image)

Figure 8: Sample Frame of Tracking Facial Actions in High Speed Video

The frame rate of those videos varied from 600 to 1000 frames per second. The original frame dimension was $432 \times 192$ pixel. For performing experiments with different spatial dimensions, we preprocessed the videos and converted the dimensions into $256 \times 256\times 512 \times 512$ and $1024 \times 1024$ pixels. In case of temporal dimension, we considered computing for 1000 frames.

Our experimental setup consists of three different CUDA devices. The list is, a) Tesla C1060, b) Tesla K20 and c) Gtx 750 Ti. These devices represent two generations of Nvidia architectures. Tesla C0160 and Tesla K20 are from Tesla architecture and Gtx750 Ti is from Maxwell architecture.

We developed individual kernels ($K_1, K_2, K_3, K_4, K_5, K_6$) for each of the algorithms presented in Table II. The algorithms (and hence the kernels) will be executed in the given order in the table.

Table IV is showing the corresponding kernel names and relevant thread level dependencies for each kernel. We will refer each of these kernels as “Simple Kernel”. Execution of these kernels in sequence will be referred as “No Fusion”. From the dependency level, we created two fusible kernel sets, $K_1$ consisting of $K_1, K_2, K_3, K_4, K_5$ and $K_2$ consisting of $K_6$.

By using the process described in section VI-A, we found a fusion solution to fuse all the kernels in $K_1$ into a single kernel, $K_f$ (named “Full Fusion”). As there is only one kernel in $K_2$, it will not require any fusion. For the sake of comparison, we created a non-optimal fusion, where we fused $K_1, K_2$ into $K_f$, and $K_3, K_4, K_5$ into $K_f$. Both of these are referred as “Two Fusion”. In order to compare GPU performance, we also executed the algorithms in the corresponding CPUs. We’ll refer these as “CPU” executions.

All these algorithms in Table II are general enough to be considered as representative algorithms in the image processing domain. Our experiments have shown that kernel fusion is indeed useful. We also calculated the amount of data transfers, data utilization and others for “No Fusion”, “Two Fusion” and “Full Fusion”. We bench-marked the total execution time for different data box sizes while executing “No Fusion” and “Full Fusion” kernels. The results obtained were used to calculate the speed up for various fusion options.

| Algorithms          | Kernel Name (K) | Dependency Type          |
|---------------------|----------------|--------------------------|
| Convert RGBA to Gray| $K_1$          | Thread to Thread         |
| IIR Filter          | $K_2$          | Thread to Thread         |
| Gaussian Smooth Filter| $K_3$     | Thread to Multi-thread   |
| Gradient Filter     | $K_4$          | Thread to Multi-thread   |
| Threshold Computation| $K_5$        | Thread to Thread         |
| Apply Kalman Filter | $K_6$          | Kernel to Kernel         |

We considered the following restrictions while generating the compact kernel, a) the given order of the kernels cannot be violated, b) execution of a kernel ($K_i$) starts after finishing the execution of the previous one ($K_{i-1}$), c) the amount of shared memory each one of the kernels and the resulting $K_f$ are using, can not be greater than the total allocated shared memory size, d) all the kernels will be computing on the same input data, e) no thread in the fused kernels will depend on any thread from other blocks, f) no block will depend on threads from other blocks. It was also reasonable to assume that, all the previously known CUDA kernel optimization techniques were implemented for executing the kernels.

VIII. RESULT ANALYSIS

In this section, we will discuss our findings and explain them. All of the measured times will be in $msec$, unless otherwise mentioned.

We have executed both simple and fused kernels on different devices. Fig 15a is showing execution time for simple and fused kernels. We have computed for three different input sizes, by varying their spatial dimensions. For each of the input size, we benchmarked the execution
time for different data box dimensions \((Box_b \ has \ (x \times y \times t)\) dimensions). The spatial dimensions \((x,y)\) of each of the boxes were also varied as \((16 \times 16), (32 \times 32),\) and \((64 \times 64)\). The temporal dimension for simple kernels was, \(t = 1\). The temporal dimension for the fused kernels were calculated using eq (6). From the figure, it is clear that, for each input size and for each data window size, fused kernels performed better than simple kernels. An increase in input size, increases the execution time.

Figure 9: Simple vs Fused Kernel Execution Times for Different Input Dimensions in Different Devices. The x-axis representing image dimensions \((256 \times 256, 512 \times 512, 1024 \times 1024)\) For each of the input, we bench-marked the execution time for different sizes of data box. Here, \((16 \times 16), (32 \times 32), (64 \times 64)\), representing the spatial dimensions \(x, y\) of a box \(Box_b\). For simple kernels, the temporal dimension was, \(t = 1\) and for fused kernels \(t\) is calculated by equation (6). The y-axis is showing total execution time for computing \(B = \frac{N_{MM}}{N_{nM}}\) blocks.

Fig 10 is comparing the best and worst case GPU timing with the serial execution in the respective CPUs. GPU worst case execution times were corresponding to the computation timing for non-optimal resource allocation for simple kernels. On the contrast, the best times are corresponding to the execution timings for optimal data and resource allocation for fused kernels. In both these two cases, the spatial dimensions were \(32 \times 32\) The CPU times were collected by executing the serial process in the CPUs, hosting the respective CUDA enabled GPUs.

Fig 11 is showing the speed up achieved by fused kernels w.r.t to serial and sequential processes. Fig 11a is the speed up achieved by fused kernel w.r.t to serial process. Similarly, a fused kernel will gain speed up in contrast to the sequential kernel execution. Fig 11b is showing speed up achieved by fused kernels in different devices, in contrast to its sequential counterpart. Here, we compared the speedup, for different input sizes and variations in data box sizes.

We showed in Fig 7 the varying data utilization for different devices, when we are changing the box dimensions. Higher data utilization eventually leads to reduced total data movement. Fig 12 comparing pixel transfer and data utilization. Fig 12a comparing total pixel transfers for a given input of dimension \(256 \times 256 \times 1000\) for different box sizes, when we execute 5 kernels sequentially, two fused kernels and full fusion. In case of no fusion, the number of data movement is constant. For two fusion and full fusion, we see a variation in transfers, for changes in box sizes. Among the data box sizes, for \([8, 8, 8]\) size, two kernel has shown worse performance than no fusion. But for other cases, the data transfers were decreased gradually for an increase in box size.

Fig 12b is showing percentage reduction in data movement for two fusion and full fusion. It also shows the data utilization values corresponding to the data movement reduction. It is evident that, the amount of reduction in data movement is highly correlated with data utilization.

Usage of GMEM in case of “No Fusion”, “Two Fusion”, and “Full Fusion” are shown in Fig 13. For different sizes of input data, both “Two Fusion” and “Full Fusion” has reduced GMEM usage (33% and 44%, respectively).

Fig 15 is showing nvprof profile timing diagram for
simple and fused kernel execution. We used nvprof for generating the images for K20 device. The box sizes for this experiment were $[32 \times 32 \times 16]$ and $[32 \times 32 \times 1]$ for fused and simple kernels respectively. Fig 15a is showing the execution of fused kernel, which is calculating 16 frames in this particular example. In contrast, Fig 15b is showing the execution of single kernels, in a sequence, calculating 1 frame. Time required for fused kernels (for 16 frames) is $\approx 490\mu\text{sec}$, that is, $31\mu\text{sec}$ per frame, where as time required to process 1 frame by simple kernel is $\approx 64\mu\text{sec}$. Computation time for 16 frames will include system overhead time. In reality, we found it took $\approx 1400\mu\text{sec}$ to finish operations on 16 frames.

Fig 14 is showing calculated throughput achieved by simple and fused kernels, for different input sizes executed in different devices. In our input cases, we varied the spatial dimensions, and kept the temporal dimensions same. An increase in input dimension impose extra pressure on computation. It is evident form the figure that, application of kernel fusion would definitely enable us to process more frames in comparison to sequential process.

**IX. RELATED STUDY**

CUDA enabled GPU devices have assisted domain experts in developing faster solutions exploiting parallelism for relevant problems. For designing a CUDA kernel code, developers have to iterate through the following steps: a) Develop, b) Parallelize, c) Optimize and d) Deploy. Optimization took substantial part of developing an efficient CUDA kernel. Researchers have been trying to provide techniques for developing such code. GPU performance improvement via kernel fusion is introduced in many contexts. Guibin et. al [7] proposed kernel fusion technique for energy efficiency. The research was only focusing on reducing the usage of hardware resources, by fusing a small number of kernels. Filipovic et. al [8] showed a library based kernel fusion. Their technique is to generate meta-information of BLAS (basic linear algebra subroutine) functions, represent kernel operations as a high level function. So, when multiple kernels are represented by using those basic functions, an analyzer can analyze the code in meta-information, re-generate new sets of kernels. Fousek et. al [9] also showed a similar map-reduce based performance improvement using kernel fusion. Haicheng et. al [10,11] proposed automatic kernel fusion technique, based on similar principle for Data Base query operations using CUDA device.

In a very recent paper, Wahib et. al [6] presented kernel fusion method for fusing multiple kernels accessing different arrays. In their problem, they initially grouped multiple kernels together based on their data dependency, and later fused the kernels in the same group. Group of kernels were flexible about the sequence of execution. They represented the kernel fusion problem as an optimization problem and proposed a method to solve the problem using their performance prediction method. One of the limitations of their model was, the fused kernels already existing in code base. Also, the fused kernels were not...
restricted to be executed in a strict sequence. For computing multi-dimensional arrays, their model did not mention the data distribution policy for efficient performance.

Image processing domain lacks the existence of set of basic operators or operations, which is universally accepted. As a result, it would be very challenging to represent a given image processing algorithm as combination of basic operators. Also, different algorithms have different data dependency. So, for image processing domain, there are a few major challenges, a) identify the partition of given kernels, b) compose efficient kernels, c) efficiently distribute input data and d) allocate resource, in order to optimally execute the fused kernel(s). Another challenge is, algorithms might be SIMD in nature, but, they have to be executed in a restricted sequence.

The kernel fusion technique is significantly different from kernel tuning technique. Techniques for improving the performance of a single kernel were developed by different researchers [12]–[14]. Application of these techniques for kernel development were shown to improve performance for target GPU devices. Kernel tuning techniques, manages data and resources exclusively used by the kernel. In the presence of multiple kernels executing on multidimensional data, the overall data utilization depends on the data access patterns by the algorithms. So, for multiple kernel executing on various number of data arrays, it is required to develop new technique to manage data movement among the kernels.

X. CONCLUSION

In this paper we studied application of kernel fusion for analyzing large volume of video data. Our fusible kernel identification method is very easy to implement. The kernel fusion algorithm fused kernel, which will reduce data traffic and reduce GMEM usage. The data distribution algorithm further ensures the reduction in data traffic. Overall, our method considers, data access patterns imposed by different algorithms, and available resource parameters to achieve minimal total execution time. Empirical observations verified the effectiveness of the proposed method. We presented the evaluation of the model using a limited set of algorithms. But, the algorithms are representative algorithms form the domain, and our methods can be used for extended set of such algorithms. Our future goal is to develop a set of basic algorithms, which can be used to represent any IPA. This will lead to develop automated fused kernels, in contrast to our manual process.

REFERENCES

[1] T. Luhmann, “Close range photogrammetry for industrial applications,” ISPRS Journal of Photogrammetry and Remote Sensing, vol. 65, no. 6, pp. 558 – 569, 2010. (ISPRS) Centenary Celebration Issue.
[2] E. D. Ross and V. K. Pulusu, “Posed versus spontaneous facial expressions are modulated by opposite cerebral hemispheres,” Cortex, vol. 45, no. 5, pp. 1280–1291, 2013.
[3] J. Kimbell, E. Gross, D. Joyner, M. Godo, and K. Morgan, “Application of computational fluid dynamics to regional dosimetry of inhaled chemicals in the upper respiratory tract of the rat,” Toxicology and Applied Pharmacology, vol. 121, no. 2, pp. 253 – 263, 1993.
[4] http://www.gurobi.com/, “Gurobi optimizer.”
[5] “CUDA Programming Guide.” http://docs.nvidia.com/cuda/cuda-c-programming-guide/.
[6] M. Wahib and N. Maruyama, “Scalable kernel fusion for memory-bound gpus applications,” in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC ’14, pp. 191–202, 2014.
[7] G. Wang, Y. Lin, and W. Yi, “Kernel fusion: An effective method for better power efficiency on multithreaded gpus,” in Proceedings of the 2010 IEEE/ACM Int’l Conference on Green Computing, pp. 344–350, 2010.
[8] J. Filipovic, M. Madzin, J. Fousek, and L. Matyska, “Optimizing CUDA code by kernel fusion—application on BLAS,” CoRR, vol. abs/1305.1183, 2013.
[9] J. Fousek, J. Filipović, and M. Madzin, “Automatic fusions of cuda-gpu kernels for parallel map,” SIGARCH Comput. Archit. News, vol. 39, pp. 98–99, Dec. 2011.
[10] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili, “Kernel weaver: Automatically fusing database primitives for efficient gpu computation,” in Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-45, (Washington, DC, USA), pp. 107–118, IEEE Computer Society, 2012.
[11] H. Wu, G. Diamos, J. Wang, S. Cadambi, S. Yalamanchili, and S. Chakradhar, “Optimizing data warehousing applications for gpus using kernel fusion/fission,” in Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops, pp. 2453–2462, 2012.
[12] A. Davidson and J. D. Owens, “Toward techniques for auto tuning gpu algorithms,” Para 2010: State of the Art in Scientific and Parallel Computing, June 2010.
[13] S. Grauer-Gray, L. Xu, R. Searsle, S. Ayalaosmayajula, and J. Cavazos, “Auto-tuning a high-level language targeted to gpu codes,” in Innovative Parallel Computing (InPar), 2012, pp. 1–10, May 2012.
[14] J. A. Stratton, N. Ansari, C. Rodrigues, N. O. T. L. Chang, and G. D. Liu, “Optimization and architecture effects on gpu computing workload performance,” Innovative Parallel Computing, pp. 1–10, 2012.