A GPU-based Algorithm-specific Optimization for High-performance Background Subtraction

Chulian Zhang, Hamed Tabkhi and Gunar Schirner
Department of Electrical and Computer Engineering
Northeastern University, Boston, MA 02115
zhang.chul@husky.neu.edu, {tabkhi,schirner}@ece.neu.edu

Abstract—Background subtraction is an essential first stage in many vision applications differentiating foreground pixels from the background scene, with Mixture of Gaussians (MoG) being a widely used implementation choice. MoG’s high computation demand renders a real-time single threaded realization infeasible. With it’s pixel level parallelism, deploying MoG on top of parallel architectures such as a Graphics Processing Unit (GPU) is promising. However, MoG poses many challenges having a significant control flow (potentially reducing GPU efficiency) as well as a significant memory bandwidth demand.

In this paper, we propose a GPU implementation of Mixture of Gaussians (MoG) that surpasses real-time processing for full HD (1080p 60 Hz). This paper describes step-wise optimizations starting from general GPU optimizations (such as memory coalescing, computation & communication overlapping), via algorithm-specific optimizations including control flow reduction and register usage optimization, to windowed optimization utilizing shared memory. For each optimization, this paper evaluates the performance potential and identifies architectural bottlenecks. Our CUDA-based implementation improves performance over sequential implementation by 57x, 97x and 101x through general, algorithm-specific, and windowed optimizations respectively, without impact to the output quality.

I. INTRODUCTION

Background subtraction is the first stage in many vision applications, separating foreground pixels from a static background scene. Example applications utilizing background subtraction include video surveillance, industry vision, and patient monitoring systems. Background subtraction algorithms range from history-based realizations to adaptive learning algorithms. For scenes with static camera position, Mixture of Gaussians (MoG) is most frequently used thanks to its high quality and efficiency. MoG is an adaptive algorithm. Multiple Gaussian components (with different learning factors) model each pixel’s background. The model for each pixel is updated with each frame. If none of the Gaussian sufficiently matches the incoming pixel, it is declared as foreground. MoG quality results come at the cost of significant computation complexity and memory bandwidth. Even on a state-of-the-art CPU, MoG for Full-HD operates at 1 frame/s, which is 60 times slower than real-time. Nonetheless, as pixel operations in MoG are independent of each other, MoG is very suitable for parallel implementation.

With the proliferation of many-core platforms, multi- and many-thread implementations gain in importance. Graphics Processing Units (GPUs) have evolved to support general-purpose applications from different market domains, suitable for data-intensive applications with high-throughput requirements and SIMD style parallelism. Consequently, GPUs have also been considered for high-performance vision computing. Since MoG is an embarrassingly parallel application—individual pixel are processed independently—it is a suitable candidate for GPU mapping. Nonetheless, many challenges remain to adjust the MoG algorithm for modern GPU architectures to achieve maximum performance.

GPU programming is challenging as it requires explicit memory management and workload mapping which is typically not needed in CPU-based programming. Furthermore, general GPU optimization approaches have to be augmented with algorithm-specific optimizations to maximize performance. General GPU optimizations include matching the data layout in memory with the many-threaded access (memory coalescing), or overlapping CPU-GPU data transfers with GPU processing. Overall, suitable workload decomposition, thread organization and mapping decision are critical to optimize device utilization. At the same time, algorithm-specific challenges need to be overcome. For example, conditionals and control flow statements hinder efficient GPU execution as parallel threads in a warp should execute an identical path. All these challenges exist in MoG. In addition, MoG’s high memory bandwidth demand poses further challenges to achieve an efficient GPU implementation.

Early mapping approaches of MoG to GPU platforms achieve limited speedup over a single-threaded implementation (around 20×). They mainly apply general GPU-based optimizations, such as coalescing memory accesses and overlapping data transfer with computation, however, leave the actual algorithm largely untouched. In order to achieve considerable performance improvements, a detailed analysis of algorithm to architecture mapping is required involving a thorough understanding of the algorithm itself. In other words, algorithm / architecture co-tuning is necessary to better match the algorithm with the underlying targeted architecture.

In this paper, we propose an efficient GPU implementation of the MoG algorithm that far surpasses real-time requirements. To achieve this, we employ both general and algorithm-specific optimizations. We optimize data layout for Gaussian parameters to facilitate coalesced memory access and overlap data transfer and kernel execution hiding communication latency. We also identify a set of algorithm-
specific optimizations to improve GPU utilization guided by a detailed analysis of architectural bottlenecks exposed by the algorithm. Algorithm-specific optimizations include minimizing the number of conditional statements (branches), adding source-level predicated execution, and minimizing the number of registers per thread. Furthermore, we utilize the performance improvements through local shared memory in a segmented (windowed) MoG that combines processing over multiple consecutive frames. We analyze the trade-off between segment size, shared memory utilization and latency.

Our general GPU optimizations already yield a 57× speedup on a Nvidia Tesla C2075 GPU \([9]\) over single threaded execution on an Intel Xeon E5-2620 \([10]\). Algorithm-specific optimizations increase speedup to 97×. The segmented MoG further increases with up to 101×. On top of that, we investigate the impact of data type and number of Gaussian components on performance. Throughout all optimizations, the output quality remains largely untouched with -4% in foreground quality (MS-SSIM\([11]\)) compared to single-threaded execution.

The remainder of this paper is organized as follows. Section II discusses relevant related work. Section III overviews the MoG algorithm and reviews relevant GPU architecture background. Section IV introduces the optimization methods and evaluates the efficiency. Section V expands the discussion to include the impact of data type and number of Gaussian components. Finally, Section VI concludes the paper and touches on future work.

II. RELATED WORK

With GPUs evolving to support general-purpose applications, researchers have started to adapt vision applications to GPU platforms \([12]\) to improve throughput and efficiency. A subset of OpenCV vision APIs \([13]\) have been realized in CUDA for parallel execution \([14]\). Key algorithms including image segmentation \([15]\), feature extraction \([16]\), object detection and tracking \([17]\) have been mapped to GPUs platforms. Most of these approaches mainly rely on general GPU optimizations (see Section IV-B). Our techniques presented in this paper can help to identify opportunities for algorithm tuning with respect to GPU architecture to further improve performance.

Few research teams focused on a GPU-parallel realization of MoG \([5], [6], [7], [8], [18]\). However, these approaches mainly focused on general GPU optimizations largely independent of the MoG algorithm. Thus, these approaches achieve a limited speedup of about 20× over serial CPU execution. The optimizations applied are almost similar across these approaches.

A common optimization for GPU is overlapping data transfer and kernel execution. In \([5], [6]\), the input frame and output foreground pixels are streamed using DMA in parallel to executing the kernel. However, compare to \([6], [5]\) imposes much more traffic to the system bus, as Gaussian parameters need to be fetched and write back to the main memory per frame execution. For each pixel, MoG reads the Gaussian parameters (with up to 120 bytes assuming 5 Gaussian with 3 double-precision parameters per Gaussian) and writes them back after updating. Three of the four approaches (except \([5]\)) store the Gaussian parameters in GPU global memory avoiding repeatedly transferring Gaussian parameters back and forth.

Many approaches \([5], [6], [8]\) employ memory coalescing to increase memory access efficiency when accessing GPU memory. Furthermore, memory coalescing potentially reduces the volume of transaction to GPU memory, utilizing GPU memory bandwidth. However, even in a perfect GPU memory utilization, updating of Gaussian parameters is still subject to memory access latency, limiting the GPU utilization. Some other approaches \([5], [6], [8]\) improve efficiency by utilizing shared memory to keep part of the Gaussian parameters on-chip avoiding off-chip access and its latency. However, \([5], [6], [8]\) did not reveal shared memory details (e.g. amount) and did not analyze that optimization individually for its performance improvement potential. Our work specifically isolates the effect of shared memory, investigates into the trade-off between segment size, window depth and performance.

The approach \([18]\) aims to reduce the amount of computation in MoG based on the algorithm in \([19], [18]\) eliminates Standard Deviation along with its associated costly operations and utilizes variable number of Gaussian components per pixels. This boosts the performance at cost of quality loss. Overall, using a variable number of Gaussian components seems a promising for a CPU-bound MoG realization. However, when targeted to a GPU, it may only yield limited benefits. The parallel threads in a GPU execute in lock-step mode. All threads perform the same amount of computation even with variable number of Gaussian components. In result, the thread with the most Gaussian components determines the latency of all parallel threads. Furthermore, an unbalanced memory access pattern within the parallel threads potentially reduces the memory access efficiency, limiting the achievable speedup.

Overall, the previous approaches focused on GPU general optimizations not utilizing algorithm-specific optimization opportunities. This leads to limited speedup over serial implementation. In contrast, our work further expands the optimization potential including register usage reduction and reducing MoG branch divergence which require thorough algorithm understanding. In addition, our step-wise optimization provides insight into the potential of each optimization, highlights architectural features involved, and thus provides guidance for future parallelization efforts.

III. BACKGROUND

In order to place our proposed optimizations into perspective, this section first introduces essential aspects of the Mixture of Gaussian (MoG) and highlights its challenges for parallelization on GPUs. Second, this section briefly overviews the targeted GPU to highlight architecture features relevant for our optimization.
A. Mixture of Gaussian

Background subtraction can be realized with different algorithms (see an overview in [1]). Among the options, we selected Mixture of Gaussians (MoG) [2]. MoG offers a very good quality and efficiency in capturing multi-modal background scenes. Furthermore, MoG offers a limited storage requirement since it updates a unified model of the background scene without need to store a history of old frames. MoG is widely used for deployments with fixed camera position.

Fig. 1 illustrates a coarse-grained flow of the MoG. Each pixel’s background value is modeled by three to five Gaussian Components (distribution). The pixel’s Gaussian Components are updated, based on learning factor, taking into account a new incoming pixel value. If none of the Gaussians sufficiently matches the new pixel value, that pixel is declared as a foreground. Each Gaussian Component consists of three parameters: (1) Standard Deviation (sd), (2) Mean (m) and (3) Weight (w).

Algorithm 1 outlines a serial MoG implementation (see details in [20]). The input is a video frame and Gaussian parameters. The algorithm loops through all pixels in the frame (lines 2 – 29). For each pixel, the algorithm first classifies the pixel’s Gaussian components into match or non-match components (line 5). A component matches if the component’s mean is within a certain range $\Gamma_1$ of the current pixel value. Gaussian parameters are updated based on match classification. In case that no match exists (line 12), algorithm creates a new Gaussian component, called virtual component, and replaces the virtual component with the Gaussian component with smallest weight value (line 14). Then, the components are ranked based on their weight over standard deviation ratio (line 16) and sorted by rank (line 19). Starting from the highest rank component, the algorithm declares a pixel to be background based on rank and closeness in matching with the current value (line 23). When finding the first match, the comparison stops and the algorithm continues with the next pixel.

A considerable computation and communication demands are required to realize MoG for full-HD (1080p 60Hz), which exceeds the capabilities of current CPUs. To realize real-time processing, acceleration is necessary. MoG loop iterations over pixels are independent, thus each pixel could be operated on in parallel. Hence, MoG is embarrassingly parallel. A GPU with the massively parallel compute units is a suitable target platform.

Algorithm 1 MoG Pseudo-code

```plaintext
function MoG (in Frame, inout Gaussian, out Foreground)
for i = 0 to numPixel do
  for k = 0 to numGau do
    diff[k] = abs(m[k] - pixel)
    if diff[k] < $\Gamma_1$ then
      match = 1
    else
      non-Match
  end if
end for
for all gaussian do
  Calc Rank for gaussian
end for
for all gaussian do
  Sort Component by Rank
end for
Foreground = 1
for k = HighestRank to 0 do
  if $\omega_k \geq \Gamma_2$ && diff[k]/sd_k < $\Gamma_1$ then
    Foreground = 0
  end if
end for
end function
```

B. GPU Architecture

Modern Graphics Processor Units (GPUs) platform offer a clustered-SIMD architecture with massive thread-level parallelism to support data-intensive high-throughput applications. Fig. 2 conceptually highlights the main architecture elements of a Nvidia GPU. For our experiments, we use a Nvidia Tesla C2075 [9]. Table 1 compares its main architecture features against the selected CPU (Intel Xeon E5-2620 [10]). The Tesla C2075 consists of 14 Streaming Multiprocessor (SM) units. Each SM contains 32 cores (compute units), a shared scheduler, 48kByte shared memory and a large 32K register file (32-bit each) that is shared among the cores in the SM.

![Fig. 2: GPU Architecture Overview](image-url)
TABLE I: HW Configuration

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel Xeon E5-2620</td>
<td>Nvidia Tesla C2075</td>
</tr>
<tr>
<td>Cores</td>
<td>6</td>
<td>448</td>
</tr>
<tr>
<td>Frequency</td>
<td>2.5 GHz</td>
<td>1.15 GHz</td>
</tr>
<tr>
<td>FLOPS (single)</td>
<td>120.3 GFLOPS</td>
<td>1.03 TFLOPS</td>
</tr>
<tr>
<td>FLOPS (double)</td>
<td>(unavailable)</td>
<td>515 GFLOPS</td>
</tr>
<tr>
<td>Cache</td>
<td>L2 (256K), L3 (15M)</td>
<td>L1 (16/48K), L2 (768K)</td>
</tr>
<tr>
<td>Mem. BW</td>
<td>12.8GB/s (DDR3)</td>
<td>144GB/s (GDDR5)</td>
</tr>
</tbody>
</table>

Cores within an SM execute in SIMD fashion. They share a common Program Counter (PC) and are controlled by a common scheduler (two schedulers per SM). Multiple threads (up to 1536) are scheduled in HW by the SM scheduler and a large register file facilitates zero-overhead context switch. This allows to hide the high latency of off-chip access by rapidly switching to another ready-to-run thread. Shared memory offers some on-chip fast access storage for shared data among parallel threads within an SM. The explicitly controlled shared memory helps to reduce accesses to the slower off-chip memory.

![CPU & GPU Interconnection](image)

**Fig. 3: CPU & GPU Interconnection**

Fig. 3 conceptualizes integration of a CPU and a discrete GPU interconnected via system bus (PCIe). More tightly integrated solutions exist (e.g. AMD’s APUs), but high-end (high-performance) GPUs are discrete. In the discrete scenario, CPU and GPU have own memories with dedicated links. The GPU does not have access to the CPU’s memory. To transfer the necessary data to/from CPU memory and GPU memory, the CPU initiates DMA transactions before and after GPU kernel execution.

IV. APPROACH & EVALUATION

This section describes first the base implementation and then our step-wise optimizations. For each optimization, the efficiency is analyzed correlated with underlying hardware characteristics. The optimizations are grouped to general GPU, algorithm-specific, and shared memory optimizations. To motivate and assess benefits of individual optimizations, the evaluation is interleaved throughout this section.

A. Base Implementation

The base implementation is almost a direct translation of the sequential implementation into CUDA (version 4.2 [21]) targeted for execution on Nvidia Tesla C2075 with Fermi architecture (see Table I for characteristics). For execution, the CPU transfers the input frame into GPU global memory, launches the MoG kernel, and transfers the foreground back. Gaussian parameters are initialized once by the CPU and then stored in GPU global memory. Gaussian parameters are updated throughout execution directly in GPU global memory thus do not affect the system bus (PCIs).

Each thread in the GPU operates on an individual pixel independently yielding 2 million threads. We select 128 threads per block which form a thread group. In the base implementation, each thread utilizes 30 registers (as per Nvidia Visual Profiler, v4.2). To achieve the highest quality, we retain the double-precision floating point data type for Gaussian parameters from the sequential implementation. We initially consider 3 Gaussian components per pixel. In this paper, we measure the execution time for processing 450 Full-HD frames (1080 × 1920) and report the achieved speedup of GPU execution over execution on the CPU.

A single-thread double-precision CPU implementation with maximum optimization (-O3 in GCC) requires 227.3s for the 450 frames, about 30× slower than real-time (assuming 60Hz). Customizing the code for SIMD operations reduces the MoG execution time by a factor of 0.28× (to 163s). The SIMD benefit is minimal due to many conditional branches in the MoG algorithm. Even a multi-threaded CPU implementation (8 threads, OpenMP) is with 99.8s far beyond real-time operation. For comparison in this paper, we chose the single core CPU implementation (-O3 optimization) as the reference point. The reported speedup can be scaled (divide by 2.3) to yield a relative comparison against the 8-core CPU implementation.

Our base implementation on GPU requires 17.5s, achieving a moderate speedup of 13× over the single core CPU implementation. The following sections introduce and evaluate our GPU optimizations to increase performance.

B. General Optimizations

Some general optimizations are suitable for almost any algorithm. This section applies to general GPU optimizations, namely memory coalescing and overlapped execution. For brevity of referencing, we refer to the optimizations by single letter as listed in Table II.

**TABLE II: General Optimization Levels**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Implementation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Memory Coalescing</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Overlapped Execution</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

**Memory Coalescing:** GPUs are designed with a simplified memory hierarchy and limited on-chip cache to increase power efficiency in comparison to general-purpose CPUs. In result, fewer accesses are masked by caches. Nonetheless, the local memory is exposed to the programmer, giving mapping flexibility. The Tesla GPU uses GDDR5 DRAM which is optimized for bandwidth (144GB/Byte/s peak). Still, if hundreds of cores simultaneously initiate memory accesses,
the memory can become the bottleneck. The MoG algorithm itself is bandwidth-hungry, requiring transferring 284 MByte (475 MByte) per full HD frame solely for updating the three (five) Gaussian parameters in double-precision.

GPU memory systems can combine (coalesce) memory accesses across threads in an SM to improve the memory access efficiency. Memory access efficiency is the ratio of data used (e.g., read from the core) over volume of data transferred from/to memory (into cache). Partially using the data in a cache line (e.g., through stride memory access) before the cache line is evicted again reduces the memory access efficiency. To increase efficiency, the parallel executing threads should ideally yield a contiguous memory access. Data structures have to be designed accordingly to yield coalesced memory accesses in order to increase memory access efficiency.

Fig. 4 illustrates the data layout for three Gaussian components of different pixels that lead to non-coalesced and coalesced memory access assuming 3 simultaneous threads. Each thread first reads mean (m), then weight (w) followed by standard deviation (sd). In Fig. 4(a), Gaussian components are stored in an array, each containing structures with the individual parameters. Hence in memory m, w, sd are interleaved. Since all threads simultaneously first read m, this leads to a stride access to memory. Due to cache size limitations, the cache line holding the data will evicted while all threads in a group read their m. Although w and sd are also fetched from memory, the core could not use them in time. This reduces memory access efficiency and artificially increases memory bandwidth demands threefold.

Since the access sequence within each thread cannot be changed, coalesced access has to be achieved by optimizing the data layout. Fig. 4(b) shows the optimized layout where the parameters of neighboring pixel’s Gaussian components are adjacent. Basically, each parameter (m, w, sd) is stored in its own array. Now, when all threads collectively read m the first Gaussian component, it results in a contiguous memory access – a coalesced access.

Fig. 4(a) illustrates the benefits of coalesced access in terms of memory access efficiency and memory store transaction. Memory access efficiency improves fourfold from only 17% in A to 78% with coalesced accesses in B. Similarly the number of memory store transaction drops from 13.3 million to 2 million. Memory coalescing increases the speedup threefold from $13 \times$ to $41 \times$ significantly improving performance.

**Overlapping Data Transfer and Execution:** Discrete GPUs do not have direct access to CPU’s main memory. Hence, the CPU initiates data transfers to/from GPU global memory via DMAs. These transfers introduce latency, which in case of limited computation but significant data movement can overshadow the benefits of GPU acceleration. Here the communication delays can even increase total run time over CPU execution [22]. To hide the transfer delay, data transfer and kernel execution can be overlapped.

Fig. 5 illustrates the MoG execution phases both without and with overlapped transfers. In the base implementation, transfers and kernel execution occur sequentially: transfer frame to GPU, execute kernel, and transfer foreground frame back to main memory. Our profiling results show that almost one third of the total execution time is devoted to data transmission. The GPU is idle during this time, which reduces overall efficiency.

To hide transmission delays, data transfers and kernel execution have to be overlapped. Fig. 5 plots the overlapped execution. While the kernel processes frame $i$, the DMA transfers the foreground image of the last frame ($i-1$) to main memory and the input data for the next frame ($i+1$) to GPU memory. The overlapped operation effectively hides the data transfer latency. To allow for concurrent data transfer and processing, we apply double-buffering to separate data that is being transferred and data being accessed by kernel. Overlapping transfers and execution increases the speedup to $57 \times$. Profiling shows that the execution time is now just

**Fig. 5:** Concurrency of data transfer and kernel execution.

**Fig. 6:** Architecture impact of general optimizations.
bounded by the kernel execution time.

To reduce kernel execution time it is important to analyze the kernel’s resource utilization. Along these lines, Fig. 6 shows how many registers are used per thread and the associated SM occupancy (ratio of active threads per SM over maximum possible number of threads). As a side effect of our memory coalescing optimization (B), each thread occupies 6 additional registers for temporary storage. Since all threads in an SM share the same register file, increasing the number of registers per thread can limit the number of concurrent threads. To further improve the performance, SM occupancy as well as thread execution efficiency need to be improved. This requires algorithm-specific optimizations described in the next subsection.

C. Algorithm-Specific Optimization

The general optimizations have already facilitated a 57× performance improvement. To further improve performance a thorough analysis of MoG algorithm is necessary. In result, we developed three algorithm-specific optimizations targeting the number of branches, predicated execution, and register usage. The optimizations are listed in Table III. The next sections describe and analyze these optimizations.

<table>
<thead>
<tr>
<th>TABLE III: Algorithm-Specific Optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>Branch Reduction</td>
</tr>
<tr>
<td>Predicated Execution</td>
</tr>
<tr>
<td>Register Reduction</td>
</tr>
</tbody>
</table>

Eliminating Divergent Branches: Threads within a warp share the same PC counter and program code. This means that parallel threads within a warp execute the same path in lockstep mode. Thread-specific branches may cause branch divergency in a warp, and parallel threads execute different conditional paths. GPUs handle divergent branches through serial execution of all possible conditional paths among parallel threads. The effect of conditional execution is achieved by predicated execution. To guide the hardware, the CUDA compiler inserts a “Set SYNchronization (SSY)” instruction indicating the start of a section with branch divergency. GPU hardware sets an active mask for each thread according to its local branch condition. Only a thread set as active (path taken) commits the final results. The results of threads without the active mask set are discarded.

Branch divergency can lead to an increased thread execution time as all possible paths are executed sequentially. MoG (Algorithm 1) has a considerable number of branches that are thread (pixel) specific: checking the status of Gaussian components (line 5), creating virtual component (line 12), ranking and sorting of Gaussian components (line 16 19), and foreground detection (line 24).

In order to reduce the number of divergent branches, we propose MoG algorithm tuning to match it better to GPU execution characteristics. Ranking and sorting helps in CPU-based execution to eliminate some computation in later steps (ie. at foreground detection). In CPU, since a Gaussian component with a higher rank is more likely to match the current pixel, the loop can be terminated early. Conversely, this typical CPU-bound optimization is inefficient for GPUs as it introduces divergence. To optimize for GPU execution, we replace ranking and sorting with an unconditional checking of all Gaussian components. Algorithm 2 and Algorithm 3 highlight our proposed algorithm tuning, comparing sort and no-sort versions. In the no-sort version, all Gaussian components are compared unconditionally (the order of finding a match does not matter here), which reduces the number of divergent branches.

Fig. 7 visualizes the effect of our algorithm-specific optimizations to architecture characteristics. As shown in Fig. 7(a) the no sort optimization (D) reduces the number of executed branches from 6.7 million to 6.2 million per frame, and improves branch efficiency by 1.5%. Branch efficiency is the ratio of non-divergent branches over total number of executed branches. At the same time D uses 4 fewer registers (Fig. 7(c) which positively affects SM occupancy. As a side effect of eliminating divergent branches the memory access efficiency slightly improves (Fig. 7(b)). By eliminating divergent branches speedup increases from 57x (in C) to 85x (in D).

Source-level Predicated Execution: The GPU compiler often expands (serializes) execution paths of branches and predicates the write back. Serializing branches effectively increases the number of wasted instructions. A further reduction in number of branches and more importantly divergent branches is necessary to further improve performance. Some branches can be avoided by source-level predication in cases the scope is beyond the reach of compiler optimizations. The concept of source-level predicated execution can be applied for updating the parameters according to their component’s match status. Algorithm 4 and 5 compare non-predicated and source-level predicated code. In non-predicated code, the parameters \( w[k], m[k] \) and \( sd[k] \) of a match component are updated (lines 3 6). For a non-match component only \( w[k] \) is updated.

**Algorithm 2 MoG excerpt (sort)**

1: for all gaussian do
2: Calc Rank for gaussian and Sort
3: end for
4: Foreground = 1
5: for \( k = HighestRank \) to 0 do
6: if \( w_k \geq \Gamma_2 \&\& \text{diff}[k]/sd_k < \Gamma_1 \) then
7: Foreground = 0
8: break
9: end if
10: end for

**Algorithm 3 MoG excerpt (no sort)**

1: for \( k = 0 \) to numGau do
2: if \( w_k \geq \Gamma_2 \&\& \text{diff}[k]/sd_k < \Gamma_1 \) then
3: Foreground = 0
4: break
5: end if
6: end for
Algorithm 4 non-Predicated Execution

1: for $k = 0$ to numGau do
2:  
3:  
4:  
5:  
6:  
7:  
8:  
9:  
10: end for

Algorithm 5 Predicated Execution

1: for $k = 0$ to numGau do
2:  
3:  
4:  
5:  
6: end for
(from A to F) on branch efficiency, memory access efficiency and SM occupancy. By combining both general and algorithm-specific optimizations, MoG touches close to 100% in memory access and branch efficiency. More importantly, SM occupancy improves from from 52% at the end of general GPU optimization (C) to 65% at the end of algorithm-specific optimizations (F). With this our MoG optimizations have achieved a very high overall GPU utilization.

D. Shared Memory

Each SM includes a fast on-chip memory (48 KB for Nvidia Tesla C2075) for frequently accessed data that is shared across cores within the SM. The local memory is managed explicitly by the program, controlling data movement, access and lifetime. To offset the overhead of fetching data from global memory, only frequently accessed data should be promoted to local memory.

The most suitable MoG data for shared memory are the Gaussian parameters. However, the required capacity for all Gaussian parameters (149 MB for full-HD resolution with 3 Gaussians and double precision) surpasses by far the shared memory capacity. In addition, the Gaussian parameters are not shared across threads as individual pixels are processed independently. MoG works on a frame basis, meaning that all Gaussian components of all pixels are updated before starting a new frame. In result, swapping Gaussian components in and out of the global memory adds additional overhead without improving performance. A mechanism is needed that Gaussian parameters are reused multiple times to warrant the effort of copying to/from shared memory.

To overcome the size limitation, we split frames into smaller pieces which we call tiles. Tiles are dimensioned so that the Gaussian parameters of a tile’s pixels fit into the shared memory resulting in a 640 pixel tile size. We modify the MoG to operate on tile basis. To realize the benefits of multiple accesses, we order frames into groups. Our tiled MoG operates first on the same tile across each frame of the frame group before shifting to the next tile. Fig. 9 highlights the basic idea of tiled MoG. The kernel first fetches the Gaussian parameters of the first tile to shared memory, processes the tile across all frames in the frame group updating the parameters in shared memory, and stores them back to global memory. The process repeats for all remaining tiles, before shifting to the next frame group.

We have analyzed the efficiency of our tiled implementation in dependency of the frame group size. Fig. 10(a) illustrates that maximum speedup is achieved with a frame group of size 8, yielding 101x speedup over serial CPU execution. Further increasing the group size does not yield better performance. Analyzing SM utilization and memory access efficiency (Fig. 10(b)) reveals the reasons for this effect.

Switching to a tiled implementation even with frame size of 1 lowered the SM occupancy to 40% (compare with 62% in optimization F). With increasing group size occupancy linearly decreases further down to 38% for a group size of 32. Lower SM occupancy is due contention for shared memory capacity as it is shared across all threads within an SM. With each thread requiring more shared memory, fewer can run simultaneously. Furthermore, processing tiles across multiple frame reduces memory access efficiency. Figure 10(a) illustrates that memory access efficiency reduces significantly from more than 90% (group size of 1), down to less than 60% (group size of 32). In summary, a tiled implementation of MoG does not significantly improve performance (increasing from 97x to 101x). In addition, it leads to an increased latency until a frame is completely processed as frame group size increases.

V. Discussion

This section discusses orthogonal aspects to the optimizations presented in Section IV. It first evaluates the effect of optimizations on output quality, investigates how the data type selection (double versus single precision) influences performance and quality, and finally compares how the number of Gaussian components impacts performance.

A. Effect of Optimizations on Quality

To validate correctness of our optimization implementations the resulting foreground decisions have to be compared against a reference model. Since floating point implementations might slightly differ across platforms, even identical implementations may yield slightly different results based on execution order[23], a direct comparison is not expressive.

To assess similarity in output between the sequential implementation and the GPU-based optimizations, we utilize Multi Scalar - Structural SIMilarity (MS-SSIM) index [24]. MS-SSIM quantifies the structural similarity between two images as a value between 0 to 1 where a higher value means closer similarity.1 Different from traditional methods such as mean

\footnotesize{1Implementation of MS-SSIM, example usage, and image comparison can be found at https://ece.uwaterloo.ca/ z70wang/research/ssim/index.html}
TABLE IV: Result Quality for Different Optimization

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Background</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
</tr>
<tr>
<td>Foreground</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
</tr>
</tbody>
</table>

squared error (MSE). MS-SSIM assess image quality based on the similarity of structural information in the scene [24]. It has become a reference measure for assessing quality against a ground truth. The ground truth in our setting is obtained through CPU execution in double precision.

Table IV illustrates the MS-SSIM quality for both background and foreground for all optimization steps. The background is similar to 99% across all optimizations, which indicates an almost identical Background results to the ground truth. The foreground overall quality is also very high. Foreground similarity shows a slight drop for some optimizations, 97% on average and at least 95%. In algorithm-specific optimizations the slight quality drop stems from algorithm tuning (e.g. replacing the sort with parallel comparison). The algorithm modifications have changed the order of floating point operations. Since floating point arithmetic is not associative to an infinite precision, the computation results slightly change. It is surprising that source-level predication did not change quality (it multiplies with approximations of 0 and 1). Conversely, we observe a drop from 97% (E) to 95% (F) when avoiding one register by duplicating a floating point subtraction. At the source level, this has not altered the floating point operation order. To gain further insight assembly-level investigations would be required. Regardless of the small drop, the results for both background and foreground retain a very good quality without perceivable differences.

B. Effect of Number of Gaussian Components

MoG algorithm provides configurability by changing number of Gaussian components per pixel. More components are suitable when dealing with complicated scenes, in cost of higher computation complexity. Our experiments demonstrate a linear increase in execution time on CPU with increasing number of Gaussian components from 3 to 5 (406.6 sec. for 5-Gaussian MoG compared to 227.3 sec in 3-Gaussian). Comparably, we also observe an increase in execution time for 5-Gaussian MoG on GPU across all optimizations. Figure 11(a) compares the achieved speedup across all optimization steps for 3 and 5 Gaussian components. We observe lower speedup by increasing the number of Gaussian components from 3 to 5. For 5-Gaussian optimization, we achieve 44x and 92x speedup for GPU general and algorithm-specific optimizations.

Figure 11(b) summarizes branch efficiency, memory access efficiency and SM occupancy across all optimizations for 5-Gaussian MoG. Compared to 3-Gaussian MoG (summarized in Figure 8(b)), 5-Gaussian MoG has lower SM occupancy due to more registers per thread for calculating five Gaussian equations. However, 5-Gaussian MoG benefits more from algorithm-specific optimizations. Algorithm-specific optimizations can efficiently reduce the number of registers per thread and eliminate divergent branches. This results in a more pronounced improvement in SM occupancy as well as branch efficiency. As an example, while even after employing memory coalescing (B) the branch efficiency is only 60%, it reaches more than 90% efficiency at the end of algorithm-specific optimizations (F).

C. Effect of Data Type

As discussed in Section IV, one determining factor for achieving highest possible speedup is SM occupancy. SM occupancy is bounded by the number of registers per thread. The algorithm-specific optimizations discussed one way to free up registers. An alternative way is to change the data type, which reduces the width of registers. Changing the Gaussian calculation from double-precision (8 byte) to single-precision (4 byte) floating-point cuts the capacity needed for registers in half. On top of that single-precision operations are generally faster than double-precision thus reduce the execution time. This comes at the cost of some loss in quality depending on data ranges occurring during execution. Execution performance on the CPU (Intel Xeon E5-2620) improves about 21% because of the switch to single-precision dropping from 227s to 180s for processing 450 full-HD frames.

Fig. 12(a) compares performance of single-precision and double-precision GPU implementation of MoG across all optimization levels. Both show an almost similar trend of speedup across general and algorithm-specific optimizations. The final optimization (F) is with 105x for single-precision somewhat
faster than double-precision (97x). Reducing register usage (E) to (F) did not yield additional improvements.

Figure [12](b) summarizes branch efficiency, memory access efficiency and SM occupancy across all optimizations for single-precision MoG. Overall, single-precision MoG also benefits from algorithm-specific optimization; memory access efficiency significantly increases (62% in C to 88% in F) along with improvement in branch efficiency (95% in C to 99% in F). Only register usage reduction does not show an impact since capacity in the register file is not a limiting factor for the single-precision implementation. We have also applied the quality assessment to the single-precision implementations still comparing against the ground truth of CPU double-precision implementation. The comparisons are very similar as in the double-precision case. Also the single-precision GPU implementation shows an average loss of around 5%. With this low quality drop, the single precision implementation is clearly preferred.

VI. CONCLUSION & FUTURE WORK
This paper discussed and evaluated a set of optimizations for GPU-based implementation of a widely used background subtraction algorithm: Mixture of Gaussians (MoG). While previous approaches were limited to general GPU optimizations (such as including memory coalescing, data transfer and kernel overlap), our proposed optimizations expand to include algorithm-specific optimizations that require more thorough algorithm understanding: divergent branch elimination and register usage optimization.

Furthermore, we propose frame segmentation to take advantage of SM shared memory to keep Gaussian parameters on-chip. By employing all proposed optimizations we achieved up to 101x speedup in comparison to serial CPU executions. Reaching this speedup required thorough understanding the algorithm and matching to the target architecture (algorithm / architecture co-design). We studied the effect of proposed optimizations over the foreground quality using MS-SSIM and ensured that our optimizations practically have no impact on quality (MS-SSIM reduces by less than 4%).

As a future work, we plan to realize MoG on an embedded GPUs due to the growing interest in adding vision capabilities in mobile devices. With the significantly lower compute power of embedded GPUs, achieving real-time performance will require to trade-off quality for speed.

REFERENCES