Model-Based Warp-Level Tiling for Image Processing Programs on GPUs

Abhinav Jangda
University of Massachusetts Amherst
USA

Arjun Guha
University of Massachusetts Amherst
USA

Abstract
The efficient execution of image processing pipelines on GPUs is an area of active research. The state-of-art involves 1) dividing portions of an image into overlapped tiles, where each tile can be processed by a single thread block and 2) fusing loops together to improve memory locality. However, the state-of-the-art has two limitations: 1) synchronization between all threads across thread blocks has a nontrivial cost and 2) autoscheduling algorithms use an overly simplified model of GPUs.

This paper presents a new approach for optimizing image processing programs on GPUs. First, we fuse loops to form overlapped tiles that can be processed by a single warp, which allows us to use lightweight warp synchronization. Second, we introduce hybrid tiling, which is an approach that partially stores overlapped regions in thread-local registers instead of shared memory, thus increasing occupancy and providing faster access to data. Hybrid tiling leverages the warp shuffling capabilities of recent GPUs. Finally, we present an automatic loop fusion algorithm that considers several factors that affect the performance of GPU kernels. We implement these techniques in a new GPU-based backend for PolyMage, which is a DSL embedded in Python for describing image processing pipelines. Using standard benchmarks, our approach produces code that is 2.15× faster than manual schedules, 6.83× faster than Halide’s GPU auto-scheduler, and 5.16× faster than autotuner on an NVIDIA GTX 1080Ti.

1 Introduction
Image processing is an important part of several fields of science and engineering, including computer vision, embedded vision, computational photography, and medical imaging. Image processing programs execute on a variety of platforms, from embedded systems in cameras to high-performance clusters that process large amounts of image data. With the increasing demand and sophistication of image processing computations, including the real-time requirements of certain domains, there is a need for high-performance implementations of image processing programs. The last decade has generated a lot of interest in efficiently executing stencils and image processing programs on GPUs [6, 11, 20–23, 25].

An image processing program is logically structured as a pipeline, where the pipeline is a directed acyclic graph of connected stages, and each stage performs per-pixel data parallel operations on the input image and produces output for dependent stages. A pipeline may have a few point-wise functions or hundreds of stencil operations and reductions. There are several domain-specific languages for writing image processing pipelines, including Halide [21], PolyMage [18], and Forma [22]. These DSLs allow programmers to express the pipeline in a natural way, but still get high-performance code by applying key optimizations, including loop fusion and overlapped tiling. Fusing stages together allows the program to exploit locality. Loop fusion is performed on the basis of a schedule that is either specified by an expert [21, 22] or automatically generated using heuristics [4, 13, 16, 17]. Overlapped tiling [18, 21, 22] splits the stage into overlapping regions that can be processed in parallel without synchronization with other tiles. On a GPU, each tile is mapped to a thread block, which stores the intermediate scratchpad arrays in shared memory. However, synchronization is still required across stages, to ensure that all produced values are ready before they are consumed by the next stage.

However, current approaches to overlapped tiling and automatic loop fusion do not fully leverage the capabilities of modern GPUs. Processing an overlapped tile per block has a high synchronization cost across stages, can require a significant amount of shared memory, and does not leverage new architectural features, such as the ability of threads in a warp to read each other’s registers [1, Chapter B.16]. Moreover, state-of-the-art autoscheduling algorithms for loop fusion and tile-size selection do not employ a rich cost model for GPUs [16]. In fact, we found that hand-written schedules significantly outperform automatically produced schedules. The fundamental problem with today’s auto-schedulers is that they are greedy algorithms that do not account for GPU-specific details, such as the number of global memory transactions, occupancy, shared memory, and register usage.

In this paper, we first present a new GPU-based execution model for image processing pipelines. Our approach leverages the architecture of modern GPUs in several ways. 1) We exploit the fact that all threads in a warp can synchronize using low overhead warp synchronization. 2) We exploit warp shuffle [1, Chapter B.16] instructions, which are available in recent GPUs that allow threads in a warp to read each others’ register values. This allows us to eliminate certain redundant
computations that are necessary in overlapped tiling, and store portions of tiles in registers instead of shared memory. 3) We present a fusion and tile size selection algorithm that uses Dynamic Programming Fusion [13] with a novel cost model to automatically select optimal fusion choices according to the cost model along with tile sizes and thread block sizes. Our cost model accounts for several factors that affect the performance of GPU kernels, including the number of global memory transactions, theoretical and achieved occupancy, and GPU resource utilization.

This paper thus makes the following contributions.

1. We present an approach to overlapped tiling on GPUs that executes one overlapped tile per warp, which significantly decreases synchronization costs (Section 4).

2. We present hybrid tiling, an approach that stores portions of a tile in registers instead of shared memory, reducing the fraction of redundant computations, and shared memory utilization, all of which can improve performance. Hybrid tiling relies on warp shuffle instructions available in recent GPUs (Section 5).

3. We present a fast automatic fusion and tile size selection algorithm that considers key factors affecting the performance of an image processing pipeline on a GPU, including the number of global memory transactions, fraction of redundant computations, and occupancy (Section 6).

4. We implement the above in a new GPU backend for PolyMage [18], which is a DSL embedded in Python for writing image processing pipelines. (The current implementation of PolyMage only supports CPUs.)

5. Using established benchmarks, we compare our approach to expert-written schedules for Halide and Halide’s autoscheduler for GPUs [16]. On a GeForce GTX 1080Ti, we achieve a speedup of 2.15× over autoscheduler, and 8× over autotuner.

The rest of this paper is organized as follows. Section 2 discusses the architecture of NVIDIA GPUs, CUDA, the PolyMage DSL, and Dynamic Programming Fusion. Section 3 presents an overview of our approach with a canonical example. Section 4 presents our technique for running one overlapped tile per warp technique. Section 5 presents hybrid tiling. Section 6 presents our automatic fusion algorithm. Section 7 evaluates our work in comparison to state of the art Halide DSL and PolyMage’s autotuner. Finally, Section 8 discusses related work and Section 9 concludes.

2 Background

2.1 NVIDIA GPU Architecture

An NVIDIA GPU consists several Simultaneous Multiprocessors (SM) that execute one or more thread blocks. Each SM consists of 1) several CUDA cores with 32 cores executing one warp, 2) shared memory, and 3) registers. The number of warps that a SM can execute concurrently depends on the shared memory usage of each thread block and register usage of each thread. Occupancy is the ratio of the number of concurrently executing warps to the maximum number of warps supported. When a warp accesses global memory, its execution is delayed due to memory access latency. To hide this latency, the warp scheduler switches execution to another warp that is ready to execute while memory is loaded.

Current stencil computations code generators for GPUs use thread block synchronization between each producer-consumer stage. The __syncthreads function synchronizes all threads in a block. Therefore, until all warps of the thread block reach __syncthreads, no warp is allowed to proceed. Therefore, the SM must switch to another warp when the current warp performs global memory access, hence, thread block synchronization can force an SM to idle if all other threads are waiting for their memory access to be satisfied.

Warp Synchronization using __synccwarf synchronizes all threads in a warp, and no thread can proceed until all threads reach the synchronization point. Warp synchronization is lightweight compared to thread block synchronization and does not force an SM to idle, because the SM does not switch warps to synchronize.

Warp Shuffle The warp shuffle instructions [1, Chapter B.16] available in recent GPUs allow threads to read register values from other threads in the same warp. __shfl_sync takes four arguments: 1) a 32-bit mask of threads participating in the shuffle, 2) a variable stored in the register to read, 3) the index of the source thread containing the register, and 4) the warp size. Similarly, __shfl_down_sync and __shfl_up_sync read registers from a thread with an index immediately before or after the calling thread. Figure 1 shows an example from [3] of reduction using __shfl_sync. For a shuffle to succeed both the calling thread and source thread must execute the instruction.

2.2 PolyMage: A DSL for Image Processing Pipelines

PolyMage [18] is a DSL embedded in Python for writing image processing programs. The PolyMage compiler transforms programs in the DSL into high-performance code for CPUs. A PolyMage program is expressed as a multi-stage...
pipeline, where each stage is a function that transforms its input into output for other stages. A pipeline is a directed acyclic graph (DAG) of stages, connected with producer-consumer relationship.

Figure 2 shows an image blurring program (blur) with two stages (blurx and blury). The parameters to the pipeline are the number of rows and columns in the image (line 1). The program first feeds the input image (img on line 9) to blurx, and then the output of blurx to blury. Each stage is a function mapping a multi-dimensional integer domain to values representing intensities of image pixels (lines 19 and 21). The domain of the function is defined at lines 12–14. blurx takes the image as input and blurs it in the x-direction (lines 19–20). blury blurs the output of blurx in the y-direction and produces the output of this pipeline (lines 21–22). The PolyMage’s compiler performs loop fusion on producer-consumer stages to improve locality and provide parallel execution. When fusing two stages, PolyMage performs overlapping tiling using polyhedral transformations. Two adjacent tiles perform redundant computations to ensure that all the data required to compute the output of a tile (known as liveouts) is available within that tile, providing parallel execution of all tiles. Within a tile, the output of a producer stage is transferred to its consumer using small buffers called scratchpads. A scratchpad is small enough to fit in a CPU cache, or in the case of our work, a scratchpad fits in GPU shared memory or registers.

### 2.3 Dynamic Programming Fusion

Dynamic Programming Fusion (DP-Fusion) [13] is an algorithm that performs automatic fusion of image processing pipelines in a few seconds. The schedules found by DP-Fusion are competitive with those produced by an autotuner after days and better than Halide’s CPU autoscheduler [17].

![Figure 2. PolyMage DSL specification for blur.](image)

![Figure 3. Equivalent CUDA code generated by Halide for blur pipeline. Both blurx and blury are fused in an overlapped tile of length tile in x-direction and 1 in y-direction, which is computed by one thread block.](image)

![Figure 4. Hybrid Tiling for blur program with tile size of 2 in x-dimension and warp size of 4. The overlapped tile is split into two tiles. Tile in red is stored in shared memory and tile in green is stored in registers. Blue line separates first and second iteration of tile. Each point of blurx is computed and stored in the register of same thread represented by t. The producer loads in red are from the registers of current thread (Case 1), black are from shared memory (Case 2), green are from the registers of another thread in same tile iteration (Case 3), and brown are from the registers of another thread in previous tile iteration (Case 4).](image)

Instead of using a greedy algorithm and a simple cost function of autoscheduler, DP-Fusion enumerates all valid fusion possibilities and uses dynamic programming combined with an analytic cost function to significantly decrease the runtime of a combinatorial algorithm. Among all fusion possibilities, DP-Fusion finds the best fusion choices on the basis of the cost of candidate fused loops. The cost of fused loops is calculated using a cost function that also uses a model to determine tile sizes.

### 3 Overview

In this section, we explain the limitations of previous work and provide an overview of our technique. Figure 3 shows CUDA code that is equivalent to the code that Halide produces for blur pipeline. The code fuses both blurx and blury together and uses overapped tiles of length tile in the x-dimension and length 1 in y-dimension. Moreover, it creates a scratchpad for blurx in shared memory and assigns each
Figure 5. CUDA code for blur pipeline with blurx and blury fused in an overlapped tile of size tile x in-dimension, which is computed by executing all threads in warp and first half of the tile is stored in shared memory with other half stored in registers.

tile to one thread block. During execution, all threads first compute blurx in parallel by looping over all points in the tile (lines 5–9) and then blury in similar fashion (lines 11–14). This code uses thread-block synchronization to ensure that all values of blurx in a tile are ready before the computation of blury starts (line 10). For tile = 2 and block sizes of 32 x 2 x 3 this code gives the best performance and runs in 2ms for a 4096 x 4096 x 3 image on NVIDIA GTX 1080Ti. However, recall from Section 2 that thread block synchronization is costly because it can force an SM to idle.

Our technique performs overlapped tiling and execute each overlapped tile per warp (OTPW). First, it processes blurx by assigning consecutive data points to consecutive threads in a warp, and then blury in the same manner. Synchronization is done by calling __syncwarp between blurx and blury. For tile = 16 and block sizes of 64 x 4 x 3 this code gives best performance and runs at 1.5 ms, hence, gives a speedup of 1.33x over the code in Figure 3. However, this approach (and the approach taken by Halide) suffers from high shared memory usage and low occupancy. For example, the code for OTPW has shared memory usage of more than 16KB leading to 62.5% occupancy.

To address this problem, we introduce hybrid tiling, which is a technique that stores tiles in both registers and shared memory. Moreover, each warp processes one tile, hybrid tiling can use warp shuffle to allow one thread to read data points stored in other threads’ registers. This eliminates the need for per thread redundant computation as well. Storing tiles in registers increases performance by 1) decreasing the shared memory usage, thereby, increasing the occupancy, and 2) decreasing the access time because register access is faster than shared memory access [14]. Since, nvcc reports that above code uses only 24 registers and on GTX 1080Ti we can store 8 data points in registers and half the shared memory usage, leading to 100% occupancy. Figure 5 shows the code with hybrid tiling for tile = 16. As shown in Figure 4, data points of blurx for one half of the overlapped tile is stored in shared memory while the other half is stored in the registers. Lines 6–10 processes blurx on the first half of the overlapped tile stored in shared memory using a warp by assigning consecutive data points to consecutive threads in a warp and looping over all points in one tile. Lines 12–16 unroll the loop and store each data point in registers. Lines 18–22 compute the values of blury for half of the tile that is stored in shared memory. Lines 24–25 retrieve the values of blurx from other threads using warp shuffle. Since, the first two values for the first thread in a warp are the values produced and stored in shared memory by last two threads of that warp, lines 26–28 retrieve the last two values of shared memory for that warp. Line 30 computes each blury point for the eighth tile. Similarly, for the ninth tile, lines 32–38 retrieve the values of blury_9 from previous threads and for first two threads of warp values of blury_8 are retrieved from last two threads of the warp. Similarly, code is generated for the remaining six iterations. Since, the code uses only eight extra registers and decreases the shared memory usage by half, the theoretical occupancy is increased to 100%. This code runs at 1.3ms which is 1.15x faster than the code using OTPW and 1.54x faster than the code in Figure 3.

In summary, our approach uses low cost synchronization, stores tiles in registers, and decreases shared memory usage. We also target the problem of fusing pipeline stages and choosing tile and thread block sizes with an automatic fusion algorithm. Our algorithm considers key factors affecting the performance of GPU kernels which are not considered in previous work [4, 16, 17]: (i) number of global memory transactions, (ii) achieved and theoretical occupancy, (iii) GPU resource usage, and (iv) fraction of overlapping computations.
4 Overlap Tile per Warp

In this section, we present our execution model that executes one Overlap Tile per Warp (OTPW). We present an algorithm to determine 1) the tile sizes for each overlap tiles based on thread block sizes, 2) assignment of consumers data points to each thread, 3) the size of scratchpad buffers, and 4) the fraction of overlap. We follow these notations: x, y, and z are the three dimensions; \( i \) is a dimension from \( \{x, y, z\} \); \( (N_x, N_y, N_z) \) is the input size; \( (B_x, B_y, B_z) \) is the thread block size; a thread's coordinates in a thread block is \( (t_x, t_y, t_z) \); and the tile sizes are \( (T_x, T_y, T_z) \) where \( (O^n_x, O^n_y, O^n_z) \) is the overlap in each dimension for the \( n^{th} \) stage.

A three-dimensional thread coordinate is converted to the linear thread ID \( t_x + B_x \times t_y + B_x \times B_y \times t_z \). The Warp ID of a thread is the thread ID divided by WarpSize and the index of a thread in a warp (known as its lane ID) of a thread is the remainder of this division. We define warp sizes, \( W_x, W_y, W_z \) such that:

\[
W_x = \min(B_x, \text{WarpSize}) \\
W_y = \min(B_y, \text{WarpSize} \div W_x) \\
W_z = \min(B_z, \text{WarpSize} \div (W_x \times W_y))
\]

In above equations we assume that number of threads in a thread block are a multiple of WarpSize. (We can pad extra threads if needed to ensure that this is the case.) These warp sizes are the number of threads with distinct ids of that dimension in a warp. The number of warps in dimension \( i \) is equal to the ratio of block size to the warp size of that dimension, i.e., \( \left\lfloor \frac{B_i}{W_i} \right\rfloor \). The warp ID of a thread in a dimension is the floor of division of the thread's ID in that dimension to the warp size of that dimension, i.e., \( \left\lfloor \frac{t_i}{W_i} \right\rfloor \) and lane ID is the remainder of the division, \( t_i \mod W_i \). Note that product of all warp sizes obtained from above equation is equal to WarpSize. For a given overlapped tile sizes, we create a warp overlapped tile by extending the tile sizes of each dimension to cover exactly one warp. The total number of points in a warp overlapped tile excluding the redundant computations is a product of number of points in the given overlapped tile sizes and WarpSize. For the given overlapped tile size, size of the warp overlapped tile is \( (T_x \times W_x, T_y \times W_y, T_z \times W_z) \).

Tiling a dimension produces two dimensions: an outer dimension that is iterated from the number of tiles and an inner dimension that is iterated the tile size times. The outer dimension is initialized to the warp ID of that dimension. The inner dimension is initialized to the lane ID in that dimension and the product of current tile iteration and WarpSize. To process each warp tile, the consecutive threads in \( i^{th} \) dimension are assigned consecutive data points in an outer loop that runs for \( T_i \) times.

The size of each scratchpad for a stage is exactly the number of data points computed by the thread block for that stage. For the \( n^{th} \) stage, each warp computes two types of data points in \( i^{th} \) dimension: 1) \( T_i \times W_i \) computations for the tile, and 2) \( O^n_i \) overlapping computations. We represent the number of data points computed (and the size of the scratchpad) for \( n^{th} \) stage as \( \prod_{i \in \{x, y, z\}} \left( \frac{B_i}{W_i} \times (T_i \times W_i + O^n_i) \right) \).

Performing tiling produces extra conditional branches and arithmetic instructions, we do not perform OTPW in a dimension when the warp size in that dimension is 1. Therefore, if there are no dimensions with warp sizes greater than one, we do not perform OTPW. There will be at least one dimension that will have warp size greater than 1, if the group processes more than one point in its input.

The fraction of overlap in OTPW for a 2-D \( n^{th} \) stage is

\[
\frac{\sum_{n}(W_x \times T_y + O^n_y) \times O^n_x + O^n_y \times (W_x \times T_x + O^n_z))}{\sum_n(W_y \times T_y + O^n_y) \times (W_y \times T_x + O^n_z)}
\]

We now present a hybrid tiling technique that uses the fact that overlapped tiles are computed in a warp.

5 Hybrid Tiling

In this section we present hybrid tiling, which divides a tile between shared memory and registers. We use warp shuffles to allow each thread to access required data from other threads in a warp, which eliminates the need for redundant computations within a tile. Hybrid Tiling solves the issues of shared memory only tiling. 1) Storing a part of a tile in registers decreases allocated shared memory, that might increasing the occupancy and lower the data transfer cost. 2) With larger tile sizes the fraction of redundant computations per tile is decreased. 3) Since register accesses using warp shuffles are faster than shared memory, storing tiles in registers leads to faster accesses of data points.

To implement such a hybrid scheme, we split the warp overlapped tile over a split dimension, into two tiles with one tile stored in the shared memory and other tile stored in registers as shown in Figure 4. Within a warp overlapped tile, we create two parallelogram tiles of half of size of warp overlapped tile in the split dimension and the same size as that of warp overlapped tile in other dimensions. The slope of the parallelogram tiles are parallel to the right hyperplane of the warp overlapped tile in the split dimension. If the parallelogram tiles are parallel to the left hyperplane, then two adjacent tiles will have cyclic dependencies. To avoid this problem, the left parallelogram tile including the overlap on the left side is stored in shared memory and right parallelogram tile is stored in registers. Since, the right tile is dependent on left tile tile on the right tile, the left tile must be generated before the right tile. Since all producer loads by OPTW are in shared memory, we need to convert these loads to access data stored in registers.

Figure 4 shows the type of producer loads in register tiles. Each producer value used in the loop is loaded from one of the four locations: Case (1) a register of the current thread, if the load index in the split dimension is same as the iteration...
in the split dimension, Case 2 from shared memory, if the load index is less than the lower bound of register tile in split dimension, Case 3 from a register of another thread, if the load index in any dimension is less than the iteration of that dimension, and Case 4 from a register of previous tile iteration of another thread, if the load index in any dimension is less than the iteration of that dimension.

Algorithm 1 is our hybrid tiling algorithm. The HybridTiling function takes the group, tile sizes, and warp sizes as parameters and generates code for hybrid tiling. For given tile sizes and warp sizes, the split dimension is found that must have tile size of more than 1 (line 31). If no split dimension is found then tiles are stored only in the shared memory. We first generate the shared memory tile using the existing PolyMage’s compiler and then generate register tiles using GenRegTile function takes the stage of group, split dimension, and tile size as argument (lines 35–36).

For all the iterations, the loops are unrolled and each computed value is stored in a distinct variable instead of shared memory (lines 22–23). We unroll loop for all iterations for the given stage including the overlapping computations for all dimensions other than split dimension because the tile stored in register is a parallelogram in the split dimension. The source lane is set to the difference between current lane id, and difference of iteration index and load index of split dimension (line 27). We also set the index of lane id for previous tile iteration to the difference of warp size and difference of iteration index and load index of split dimension (line 28). Consequently, when the load index and iteration index are the same, the source lane ID of a dimension is set to current dimension and previous tile iteration lane id is set to warp size. Each producer load expression in the loop is replaced with suitable register reads using warp shuffle or shared memory reads (line 29).

UpdateProducerLoad takes the load indices, warp sizes, prev tile iteration lane indices, and source lane indices in each dimension. If all source lane ids are set to current lane id then register access from same thread is generated (Case 1 of Figure 4). Otherwise, we expand the multi-dimensional source lane index and prev lane index into an expression by multiplying each index by size of next dimension and summing all (lines 5–8). We generate __shfl1_sync with a mask, variable name, and the expression to determine the target thread (Case 3 of Figure 4) (line 9). GetMask retrieves the mask expression. If there are conditionals in a computation then the mask has to be determined based on the condition used by the conditional otherwise a full mask is returned. Since, all source and target threads must participate in the warp shuffle, generated warp shuffle is executed by all threads specified in the mask. A shared memory access is required only if the current iteration in the split dimension is the first iteration of the register tile (Case 2 of Figure 4) (line 11). In this case, an if expression is generated that updates the value of variable based on the lane ID in split dimension and difference of load index and iteration index (lines 11–13). Otherwise, another shuffle instruction must be executed by all threads is generated to access load index of previous iteration and the result of value is stored in a new variable (Case 4 of Figure 4) (lines 16–18). An if expression is generated that selects between the value of two shuffles, if the difference of load index and iteration index is negative.

### 6 Automatic Fusion for GPUs

In this section, we present an automatic fusion algorithm that select stages to fuse with tile sizes and thread block sizes for each fused loop. Our approach leverages DP-Fusion [13], which is an algorithm that efficiently enumerates all fusion
possibilities, given a cost function. We introduce a cost function that calculates the minimum cost of a sequence of fused loops, along with optimal tile sizes and thread block sizes.

**Preprocessing** Our algorithm requires register usage and computation time of one iteration of each stage as parameters. We obtain both by first generating code for each stage such that no two stages are fused, all global memory loads are replaced with shared memory loads, all loops perform single iteration, and outermost loop is nested inside a loop of one million iterations. We obtain the per iteration time by getting the time taken to execute the kernel by one thread block with one thread and dividing this by one million. We get register usage using `nvcc`.

Algorithm 2 is our cost function that takes four arguments: (i) a group of stages to fused, G, (ii) tile sizes, (iii) thread block sizes, and (iv) whether to do hybrid tiling or not, and returns the cost. Cost uses hardware configuration of a GPU listed in Table 1 with the name of variables used in the algorithm. Below expression calls Cost function for all tile sizes and thread block sizes and returns the minimum cost with tile sizes, thread block sizes, and if hybrid tiling is done:

\[
\text{argmin}_{\text{tileSize}, \text{tBlockSize}, \text{isHybridTile}} \text{Cost}(G, \text{ tileSize}, \text{ blockSize}, \text{ isHybridTile})
\]

Cost determines the cost (line 36) based on seven key criteria: (i) number of global memory transactions per warp, (ii) theoretical occupancy, (iii) achieved occupancy, (iv) shared memory usage, (v) register usage, (vi) amount of redundant computations, and (vii) load imbalance. Each of these components are multiplied by weights and summed to calculate the cost. We check if the dependence vectors between all stages of a group are constants after alignment and scaling of dependencies (line 2). We determine dimension sizes of group, total threads in the grid, threads per thread block, number of warps per thread block, and warp overlapped tile sizes (lines 3–5). We distribute all thread blocks equally to each SM (line 6). We retrieve the volume of each tile, the number of intermediate buffers, and multiply them with number of warps per thread block to determine shared memory usage per thread block (lines 7–9). If hybrid tiling is used, shared memory tile is split in half and updates register tile value (line 12). We check if the shared memory used per thread block is more than the maximum shared memory (line 13). Below we explain how we determine each of the cost components:

| Specification | GTX 1080Ti | Tesla V100 |
|---------------|------------|------------|
| Simultaneous Multiprocessors (NSMs) | 28 | 80 |
| CUDA Cores per SM (CoresPerSM) | 128 | 64 |
| Global Memory Bandwidth (GMemBW) | 484 GBps | 898 GBps |
| Warp Size (WarpSize) | 32 | 32 |
| Maximum Shared Memory Per thread block (MaxShMemPerTB) | 48 KB | 48 KB |
| Shared Memory per SM (SHMemPerSM) | 96 KB | 96 KB |
| Maximum Warps per SM (MaxWarpPerSM) | 64 | 64 |
| Maximum Thread Blocks per SM (MaxTbPerSM) | 16 | 16 |
| Registers per SM (RegPerSM) | 65536 | 65536 |
| Maximum Registers Per Thread (MaxRegPerTh) | 256 | 256 |
| Global Memory Transaction Size (GMemTxSz) | 32 KB | 32 KB |

**Table 1.** Hardware configuration for NVIDIA GTX 1080Ti and Tesla V100 GPU

**Algorithm 2 Cost Function**

1. function Cost(G, tileSize, tbSize, isHybridTile) if not constantDependenceVectors(G) then return \(\infty\)
2. totalThreads \(\leftarrow\) totalThreads(GetDimSizes(G), tileSize)
3. warpTileSizes \(\leftarrow\) WarpTile(tileSizes, WarpSizes(tbSize))
4. warpsPerTB \(\leftarrow\) ThreadsPerTB(tbSize) \(\div\) WarpSize
5. tbPerSM \(\leftarrow\) totalThreads \(\div\) ThreadsPerTB(tbSize) \(\div\) NSMs
6. WarpTileVol \(\leftarrow\) ComputeTileVol(G, warpTileSizes)
7. totalBuff \(\leftarrow\) NumBUFFERS(G)
8. regTile \(\leftarrow\) 0
9. if isHybridTile then
10. shMemPerTB \(\leftarrow\) warpTileVol \(\times\) warpsPerTB \(\times\) totalBuff
11. regPerTh \(\leftarrow\) regTile \(\times\) shMemPerTB \(\div\) 2
12. if shMemPerTB \(>\) MaxShMemPerTB then return \(\infty\)
13. totalGLMemTxs \(\leftarrow\) 0
14. for all gLoad \(\in\) GetGlobalMemLoaded(G) do
15. warpLoad \(\leftarrow\) GLLOADSBnWarp(gLoad, tileSize, tbSize)
16. gTxs \(\leftarrow\) MinGLTxs(warpLoad, GMemTxSz)
17. totalGLMemTxs \(\leftarrow\) totalGLMemTxs \(+\) gTxs \(\times\) tileVol
18. maxTBPerSM \(\leftarrow\) min(shMemPerTB, MaxTbPerSM)
19. shMemOcc \(\leftarrow\) \(\min(\frac{shMemPerTB}{MaxTbPerSM}) \times\) warpsPerTB, MaxWarpPerSM)
20. regPerTh \(\leftarrow\) regTile \(\times\) H \(\times\) RegUsage(H)
21. if regPerTh \(>\) MaxRegPerTh then return \(\infty\)
22. maxThPerSM \(\leftarrow\) min(RegPerSM \(\div\) regPerTh, MaxThPerSM)
23. occupancy \(\leftarrow\) min(shMemOcc, regOcc) \(\times\) MaxWarpPerSM
24. warpBW \(\leftarrow\) GMemBW \(\times\) WarpSize \(\div\) NSMs \(\times\) CoresPerSM
25. memTime \(\leftarrow\) GMemTxSz \(\div\) totalGLMemTxs \(\div\) WarpSize
26. tileVol \(\leftarrow\) ComputeTileVol(G, tileSize)
27. computeTime \(\leftarrow\) ComputeTimePerIter(H) \(\times\) tileVol
28. shMemPerSM \(\leftarrow\) shMemPerTB \(\times\) MaxTBPPerSM
29. unallocatedShMem \(\leftarrow\) 1 \(\times\) shMemPerSM \(\div\) ShMemPerSM
30. regPerSM \(\leftarrow\) regPerTh \(\times\) MaxWarpPerSM \(\times\) WarpSize
31. unusedReg \(\leftarrow\) 1 \(\times\) regPerSM \(\times\) occupancy \(\times\) Regusage
32. fracOverlap \(\leftarrow\) OVERLAPPComputation(G) \(\times\) tileVol
33. cleanupTBs \(\leftarrow\) (totalTB \(\times\) maxTBPerSM \(-\) 1) \(\%\) maxTBPerSM
34. cost \(=\) \(w_1\) \(\times\) totalGLMemTxs \(+\) \(w_2\) \((1 \times\) occupancy \(+)\)
35. \(\times\) memTime \(\times\) computeTime \(+\) \(w_3\) \(\times\) unallocatedShMem \(+\)
36. \(w_4\) \(\times\) unusedReg \(+\) \(w_5\) \(\times\) fracOverlap \(\times\) cleanupTBs
37. return cost
• **Number of Global Memory Transactions**: The cost function estimates the number of global memory transactions that either loads input images or inputs to the group (lines 15–18). The number of global memory transactions depends both on tile sizes and thread block sizes. Using [27], we retrieve the loads for each global memory load for all threads in a warp line (line 16). We coalesces all memory loads into the minimum number of transactions (line 17). Finally, we calculate the total number of transactions (line 18).

• **Theoretical Occupancy**: The cost function estimates theoretical occupancy based on shared memory and register usage. We calculate the maximum number of thread blocks supported by an SM based on the shared memory usage and take its minimum with \( \text{MaxTpPerSM} \) (line 19). Multiplying this value with number of warps per thread block gives the occupancy from shared memory usage (line 20). We sum register usage of all stages in the group obtained in preprocessing step to get the register usage of the group (line 21). We obtain the occupancy from register usage by determining the maximum number of warps supported based on register usage and taking the minimum with the \( \text{MaxWarpPerSM} \) (lines 23–24). Ratio of minimum of both occupancies to \( \text{MaxWarpPerSM} \) is the theoretical occupancy (line 25).

• **Achieved Occupancy**: The cost function estimates achieved occupancy based on the ratio of time spent in global memory loads to the time spent in computations. This ratio must be decreased, since, theoretical occupancy cannot be reached at runtime if warps spent most of their time waiting for memory requests to be fulfilled and an SM’s compute resources are idle. To determine the time spent in global memory loads, we divide the theoretical global memory bandwidth equally among all SMs, and then among all warps that can execute in parallel (line 26), hence, obtain the time spent in all global memory transactions (line 27). We do not use a cost model to obtain the computation time because GPU uses optimizations like pipelining instead we obtain the execution time of each stage as mentioned in preprocessing and then determine the computation time for the group by the summing the computation time for individual stages and multiplying that by the tile size (line 29).

• **Shared Memory and Register Usage**: The cost function maximizes the shared memory and register usage in addition to occupancy because while higher occupancy can imply lower shared memory or register usage, high shared memory or register usage can lead to lower occupancy. We calculate per thread block shared memory usage and register usage (line 31–33) when all thread blocks are executing concurrently based on the occupancy.

• **Fraction of Redundant Computations**: The cost function determines the fraction of overlap using Equation 1 (line 34).

• **Load imbalance**: The cost function minimizes the load imbalance due to when the number of thread blocks per SMs are not always a multiple of number of thread blocks executing concurrently per SM based on the occupancy. Line 35 determines the cleanup threads for each SM.

### 7 Evaluation

In this section we evaluate our work. We investigate the following questions: 1) Does our automatic loop fusion algorithm run sufficiently fast? 2) How does the OTPW execution model compare to the state-of-the-art? 3) How does OTPW with Hybrid Tiling compare to the state-of-art?

**Experimental Setup** We use a 3.4 GHz, quad-core Intel i5-4670 CPU with 16GB RAM and two GPUs: an NVIDIA GTX 1080Ti and an NVIDIA Tesla V100. Table 1 lists several key attributes of both GPUs. For our benchmarks, we use six canonical image processing applications that have appeared in prior work [4, 13, 17, 18, 21]. Table 2 reports the number of stages and the size of the input image for each benchmark. We compare our work to the latest version of Halide, using both expert-written schedules [2] and its GPU autoscheduler [16]. We compiled Halide with LLVM 8.0. We executed each benchmark for 3 samples with each sample containing 100 runs. We report the minimum of average of each sample.

**Cost Function Weights** The cost function that we use for automatic fusion requires several weights that are GPU-dependent. We determine the best weights empirically using three benchmarks (Unsharp Mask, Harris Corner Detection, and Multiscale Interpolate). Table 3 shows the weights for both GPUs.

#### 7.1 Automatic Fusion Time

We first measure the time it takes for automatic fusion to process each benchmark program to find an optimal schedule. We use Bounded DP Fusion with \( \infty \) as the limit [13]. Our cost model searches for thread block sizes in at maximum three dimensions, such that number of threads in a thread block is a multiple of \( \text{WarpSize} \). Moreover, searches for tile
Benchmark & H-Manual & H-Auto & PolyMage-A & PolyMage-GPU & Speedup of PolyMage-GPU over  
1080Ti & V100 & 1080Ti & V100 & 1080Ti & V100 & 1080Ti & V100 & 1080Ti & V100 & 1080Ti & V100 & 1080Ti & V100 & 1080Ti & V100 & 1080Ti & V100  

Unsharp Mask & 3.71 & 0.60 & 11.8 & 9.08 & 16.0 & 1.01 & 3.70 & 1.50 & 11.8 & 22.7 & 16.0 & 22.7 & 12.7 & 38.2 & 16.0 & 28.2 & 16.0 & 28.2  
Harris Corner & 2.42 & 0.50 & 19.9 & 17.1 & 9.20 & 8.50 & 0.72 & 0.22 & 3.42 & 2.27 & 28.0 & 78.0 & 12.7 & 38.2 & 10.0 & 38.2 & 12.7 & 38.2  
Bilateral Grid & 2.53 & 2.23 & 8.51 & 7.32 & 5.40 & 3.40 & 2.43 & 2.30 & 1.04 & 0.97 & 3.54 & 3.20 & 2.22 & 1.47 & 1.22 & 1.70 & 1.22 & 1.70  
Multiscale Interp. & 3.21 & 2.00 & 12.0 & 12.0 & 6.50 & 6.00 & 1.25 & 1.25 & 2.66 & 1.60 & 10.0 & 9.60 & 5.20 & 4.80 & 10.0 & 9.60 & 5.20 & 4.80  
Camera Pipeline & 8.13 & 6.56 & 5.42 & 6.73 & 7.50 & 7.50 & 4.43 & 2.00 & 1.84 & 4.10 & 1.22 & 4.21 & 1.70 & 3.75 & 10.0 & 9.60 & 5.20 & 4.80  
Pyramid Blend & 8.13 & 6.56 & 5.42 & 6.73 & 7.50 & 7.50 & 4.43 & 2.00 & 1.84 & 4.10 & 1.22 & 4.21 & 1.70 & 3.75 & 10.0 & 9.60 & 5.20 & 4.80  

Geomean Speedup & 2.15 & 1.73 & 6.83 & 12.67 & 5.16 & 8.02 & 

Table 4. Execution times (in ms) of benchmarks and speedup of PolyMage-GPU on GTX 1080Ti and Tesla V100.

Figure 6. Times relative to OTPTB(H-Auto) on GTX 1080Ti and Tesla V100. OTPTB(H-Manual) are the expert written schedules in Halide and OTPTB(H-Auto) is Halide autoscheduler following OTPTB execution model. OTPW(PolyMage-A) is PolyMage's Autotuner. OTPW(PolyMage-GPU) is the implementation of OTPW execution model in PolyMage-GPU. OTPW+HT(PolyMage-GPU) is the implementation of OTPW with Hybrid Tiling in PolyMage-GPU. Absolute execution times can be determined from Table 4.

sizes from 1 to 32 in each dimension. The Fusion column in Table 2 shows time taken, which ranges from less than a second to up to 30 seconds for benchmarks with a few dozen stages. Therefore, we conclude that our approach to automatic fusion is sufficiently fast.

7.2 Performance Evaluation

We now evaluate the runtime performance of one tile per warp (OTPW) and OTPW with hybrid tiling. For comparison, we run the same benchmarks in Halide using both its autoscheduler and expert-written schedules. We run all benchmarks in four ways:

1. **PolyMage-GPU** is our implementation of OTPW with hybrid tiling and uses the loop fusion algorithm evaluated above.¹

2. **PolyMage-A** uses OTPW and Hybrid Tiling, but employs the PolyMage autotuner [18] instead of our loop fusion algorithm. We used three threshold values of overlap, 0.2, 0.4, and 0.5, tile sizes from 1 to 32 in each dimension, and thread block size of 1 to 512 in each dimension.

3. **H-Manual** is Halide with expert-written schedules, taken from the Halide repository.

4. **H-Auto** is Halide with automatically generated schedules [16] and thread block sizes of 16×16×4.

Table 4 shows the absolute execution times of all four experimental configurations on both GPUs. Moreover, the last group of values in the table report the speedup of PolyMage-GPU over the other three configurations. To summarize, on every benchmark, PolyMage-GPU is at least as fast as almost all the alternative configurations, and in many cases, significantly faster. The geometric speedup over Halide’s autotuner (H-Auto) is 6.83× and 12.67× on the 1080Ti and V100 respectively. We believe this different occurs because the thread block size that H-Auto uses is better suited for the 1080Ti. Halide’s expert-written schedules (H-Manual) fare

---

¹ We generated CUDA 10.0 using nvcc -O3 -arch=sm_61 on the 1080Ti and nvcc -O3 -arch=sm_70 on the V100.
better, but PolyMage-GPU is still faster with a geomean speedup of 2.15× and 1.73× on the 1080Ti and V100 respectively. (The expert-written schedules were optimized for the V100.) In general, PolyMage-GPU outperforms the other approaches because its fusion algorithm chooses better thread block and tile sizes, and the runtime technique has lower synchronization cost, decreased shared memory usage, improved occupancy, and lower data access times with hybrid tiling. The only exception is the Bilateral Grid benchmark, where Halide’s manual schedules are competitive with PolyMage-GPU. In this case, Halide can fuse its reduction stage, which PolyMage-GPU cannot do.

PolyMage-GPU is also faster than PolyMage-A. Moreover, the loop fusion algorithm in PolyMage-GPU runs in seconds, PolyMage-A takes up to 20 hours because PolyMage-A selects the same tile size and thread block sizes for all groups, hence, not exploring several high performance schedules.

Figure 6 shows performance improvement of OTPW with automatic fusion in PolyMage-GPU and Hybrid Tiling in OTPW with automatic fusion in PolyMage-GPU over OTPTB model in Halide. For 5 out of 6 benchmarks OTPW performs better than OTPTB model in both GPUs. For the Unsharp Mask and Harris Corner PolyMage-GPU, H-Auto, and H-Manual use exactly the same grouping, but different thread block and tile sizes. On the other benchmarks, OTPW (PolyMage-GPU) performs better, because it produces groupings, with thread block sizes and tile sizes, and warp synchronization. There is no significant speedup for Pyramid Blending because the hard-coded thread block sizes in H-Auto are close to those found by our approach. However, PolyMage-GPU does group stages differently. Even though the expert-written schedule for Camera Pipeline inlines several functions that PolyMage-GPU does not, OTPW (PolyMage-GPU) still gives a significant speedup due to different grouping with thread block sizes and tile sizes.

Hybrid tiling makes OTPW faster by 1.2× (geomean). Both OTPW and H-Manual group all functions together in Harris Corner and Unsharp Mask, but use different tile and thread block sizes. In Harris Corner, OTPW achieves an occupancy of 93.75% (limited due to due to shared memory usage of 6.375 KB per thread block). However, hybrid tiling achieves 100% occupancy. In Bilateral Grid, hybrid tiling is not performed because shared memory usage of best performing tile size did not decrease occupancy to less than 100%. In Pyramid Blending, hybrid tiling has a minor effect because shared memory usage does not affect occupancy for most groups.

8 Related Work
The state-of-the-art DSLs for image processing programs [18, 21, 22] all employ loop fusion and overlapped tiling to exploit locality between producer and consumer stages. Halide and Forma leverage GPUs and execute one overlapped tile per thread block. Halide’s original CPU autoscheduler [17]

uses a greedy algorithm. Dynamic Programming Fusion [13] efficiently enumerates all possible fusion choices for a CPU. Halide has a newer autoscheduler [4] that uses beam search with a learned cost model for CPUs. Halide also has a GPU autoscheduler [16] that performs greedy function inlining and loop fusion with hard-coded thread block sizes for each tile.

Several techniques [6, 7, 10, 11, 23–25] support parallel execution of stencil computations on GPUs. Overlap tile per thread block (OTPTB) model is followed by techniques performing overlap tiling [11, 23, 24]. Rawat et al. [23] reduce the number of redundant computations in Overtile [11] by using a sliding window along one spatial dimension and performs overlapped tiling along other dimensions. Hybrid hexagonal classic tiling of [10] also executes one tile per thread block. None of the aforementioned works use OTPW model or store any portions of tiles in registers, which is a key component of the our hybrid tiling approach.

Hong and Kim [12] present a general analytical model to predict the performance of GPU kernels. However, advances in GPU architectures, including changes to their memory hierarchy, have made their model out of date. Prajapati et al. [20] present an analytical model for predicting the runtime of stencil computations tiled by [10] on GPU. Their model consists of shared memory usage, theoretical occupancy, and warp switching. However, they also did not consider other key factors: register usage, number of global memory transactions, achieved occupancy, and thread block sizes.

Other research projects have employed in-register storage and warp shuffles to improve the performance of GPU kernels [5, 8, 9, 15, 19, 26]. In contrast, our work is not restricted to one warp per thread block and is a hybrid technique that stores tiles in both registers and shared memory. To the best of our knowledge, this approach has not been presented in prior work.

9 Conclusion
In this paper, we presented (i) an execution model for image processing pipelines on GPUs that executes one overlapped tile per warp, (ii) hybrid tiling technique that stores half overlapped tile in shared memory and other half in registers and uses warp shuffles to prevent generating redundant computations for each thread, (iii) an automatic fusion technique for GPUs that considers several key factors affecting the performance of GPU kernels. On a GeForce GTX 1080Ti, we achieve a speedup of 2.15× over expert written schedules, 6.83× over autoscheduler, and 5.16× over autotuner. On a Tesla V100, we achieve a speedup of 1.72× over expert written schedules, 12.67× over autoscheduler, and 8.02× over autotuner. Our technique generate these schedules with in a few seconds while an autotuner takes up to 20 hours.
References

[1] CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[2] Halide. https://github.com/halide/Halide/ commit 60880e6cd8f632e4215063a7ce5328527c7add.

[3] Using CUDA Warp-Level Primitives. https://devblogs.nvidia.com/using-cuda-warp-level-primitives/.

[4] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michael Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Fredo Durand, and Jonathan Ragan-Kelley. 2019. Learning to Optimize Halide with Tree Search and Random Programs. ACM Trans. Graph. 38, 4, Article 121 (July 2019), 12 pages.

[5] Karan Aggarwal and Uday Bondhugula. 2019. Optimizing the Linear Fascicle Evaluation Algorithm for Many-core Systems. In Proceedings of the ACM International Conference on Supercomputing (ICS ’19). 425–437.

[6] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2008. A Compiler Framework for Optimization of Affine Loop Nests for Gpgpus. In Proceedings of the 22nd Annual International Conference on Supercomputing (ICS ’08). 225–234.

[7] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. 2010. Automatic C-to-CUDA Code Generation for Affine Programs. In Compiler Construction. CC 2010. Lecture Notes in Computer Science (CC ’10).

[8] Eli Ben-Sasson, Matan Hamilis, Mark Silberstein, and Eran Tromer. 2016. Fast Multiplication in Binary Fields on GPUs via Register Cache. In Proceedings of the 2016 International Conference on Supercomputing (ICS ’16). Article 35, 12 pages.

[9] Simon Garcia De Gonzalez, Sitao Huang, Juan Gomez-Luna, Simon Hammond, Onur Mutlu, and Wen-mei Hwu. 2019. Automatic Generation of Warp-level Primitives and Atomic Instructions for Fast and Portable Parallel Reduction on GPUs. In Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2019). 73–84.

[10] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. 2014. Hybrid Hexagonal/Classical Tiling for GPUs. In Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO ’14). Article 66, 10 pages.

[11] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. 2012. High-performance Code Generation for Stencil Computations on GPU Architectures. In Proceedings of the 26th ACM International Conference on Supercomputing (ICS ’12). 311–320.

[12] Sunpyo Hong and Hyesoon Kim. 2009. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. In Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA ’09). 152–163.

[13] Abhinav Jangda and Uday Bondhugula. 2018. An Effective Fusion and Tile Size Model for Optimizing Image Processing Pipelines. In Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’18). 261–275.

[14] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. CoRR abs/1804.06826 (2018). arXiv:1804.06826

[15] Ang Li, Weifeng Liu, Linnan Wang, Kevin Barker, and Shuaiwen Leon Song. 2018. Warp-Consolidation: A Novel Execution Model for GPUs. In Proceedings of the 2018 International Conference on Supercomputing (ICS ’18). 53–64.

[16] Tzu-Mao Li, Michael Gharbi, Andrew Adams, Fredo Durand, and Jonathan Ragan-Kelley. 2018. Differentiable Programming for Image Processing and Deep Learning in Halide. ACM Trans. Graph. 37, 4, Article 139 (July 2018), 13 pages.

[17] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically Scheduling Halide Image Processing Pipelines. ACM Trans. Graph. 35, 4, Article 83 (July 2016), 11 pages.

[18] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. In Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPOC ’15). 429–443.

[19] Prithivaya Mangoo Photilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emmina Tordak, and Rastislav Bodik. 2019. Swizzle Inventor: Data Movement Synthesis for GPU Kernels. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPL ’19). 65–78.

[20] Nirmal Prajapati, Waruna Rasasinghe, Sanjay Rajopadhye, Rumen Andonov, Hristo Djidjev, and Tobias Grosser. 2017. Accurate, Analytical Time Modeling and Optimal Tile Size Selection for GPGPU Stencils. In Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’17). 163–177.

[21] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédé Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In Proceedings of the 54th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’13). 519–530.

[22] Mahesh Ravishankar, Justin Holewinski, and Vinod Grover. 2015. Forma: A DSL for Image Processing Applications to Target GPUs and Multi-core GPUs. In Proceedings of the 8th Workshop on General Purpose Processing Using GPUs (GPGPU-8). 109–120.

[23] Prashant Singh Rawat, Changwan Hong, Mahesh Ravishankar, Vinod Grover, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. 2016. Resource Conscious Reuse-Driven Tiling for GPUs. In Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT ’16). 99–111.

[24] Prashant Singh Rawat, Miheer Vaidya, Aravind Sukumaran-Rajam, Atanas Rountev, Louis-Noël Pouchet, and P. Sadayappan. 2019. On Optimizing Complex Stencils on GPUs. (2019).

[25] Sven Verdoolaege, Juan Carlos Juega, Alberto Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. 2013. Polyhexed Parallel Code Generation for CUDA. ACM Trans. Archit. Code Optim. 9, 4, Article 54 (Jan. 2013), 23 pages.

[26] J. Wang, X. Xie, and J. Cong. 2017. Communication Optimization on GPU: A Case Study of Sequence Alignment Algorithms. In 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). 72–81.

[27] Michael E. Wolf and Monica S. Lam. 1991. A Data Locality Optimizing Algorithm. In Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI ’91). 30–44.