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

Abhinav Jangda
aabhinav@cs.umass.edu
University of Massachusetts Amherst
United States

Arjun Guha
a.guha@northeastern.edu
Northeastern University
United States

ABSTRACT

Domain-specific languages that execute image processing pipelines on GPUs, such as Halide and Forma, operate by 1) dividing the image into overlapped tiles, and 2) fusing loops to improve memory locality. However, current approaches have limitations: 1) they require intra thread block synchronization, which has a nontrivial cost, 2) they must choose between small tiles that require more overlapped computations or large tiles that increase shared memory access (and lowers occupancy), and 3) their autoscheduling algorithms use simplified GPU models that can result in inefficient global memory accesses.

We present a new approach for executing image processing pipelines on GPUs that addresses these limitations as follows. 1) We fuse loops to form overlapped tiles that fit in a single *warp*, which allows us to use lightweight warp synchronization. 2) We introduce *hybrid tiling*, which stores overlapped regions in a combination of thread-local registers and shared memory. Thus hybrid tiling either increases occupancy by decreasing shared memory usage or decreases overlapping computations using larger tiles. 3) We present an automatic loop fusion algorithm that considers several factors that affect the performance of GPU kernels. We implement these techniques in PolyMage-GPU, which is a new GPU backend for PolyMage. Our approach produces code that is faster than Halide's manual schedules: 1.65× faster on an NVIDIA GTX 1080Ti and 1.33× faster on an NVIDIA Tesla V100.

CCS CONCEPTS

• **Software and its engineering → Compilers**.

KEYWORDS

Polyhedral Optimizations; Graphics Processing Units; Image Processing Pipelines

ACM Reference Format:
Abhinav Jangda and Arjun Guha. 2020. Model-Based Warp Overlapped Tiling for Image Processing Programs on GPUs. In *Proceedings of the 2020 International Conference on Parallel Architectures and Compilation Techniques (PACT '20), October 3–7, 2020, Virtual Event, GA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3410463.3414649

PACT '20, October 3–7, 2020, Virtual Event, GA, USA

ACM ISBN 978-1-4503-8075-1/20/10…$15.00
https://doi.org/10.1145/3410463.3414649

1 INTRODUCTION

Image processing programs are essential in several domains, including computer vision, embedded vision, computational photography, and medical imaging. These programs run on a variety of platforms, from embedded systems to high-performance clusters that process large amounts of image data. With the increasing demand and sophistication of image processing computations (including real-time requirements), there is a growing need for high-performance implementations of image processing programs.

An image processing program is logically structured as a directed acyclic graph of connected stages, where each stage performs per-pixel data parallel operations on its input image and produces an output image for dependent stages. There are several domain-specific languages (DSLs) for writing image processing pipelines, including Halide [22], PolyMage [19], and Forma [23]. These DSLs allow the programmer to write independent stages in a natural way, but still get high-performance code by applying key optimizations, including *loop fusion* and *overlapped tiling*. Loop fusion allows the program to exploit locality, and is performed on the basis of a schedule that is either specified by an expert [22, 23] or automatically generated using heuristics [6, 14, 17, 18]. After loop fusion, overlapped tiling [19, 22, 23] splits each stage into overlapping regions (known as tiles) that can be processed in parallel without synchronization with other tiles. On a GPU, each tile is mapped to a *thread block*, which stores intermediate results (scratchpad arrays) in shared memory.

These approaches [8, 12, 21–25, 28, 33] to overlapped tiling and automatic loop fusion give suboptimal performance on modern GPUs for three reasons. 1) Processing an overlapped tile per thread block has a high synchronization cost across stages. 2) Smaller tiles have more overlapped regions (and thus require more redundant computation), but larger tiles require more shared memory accesses (and thus lower occupancy). 3) State-of-the-art autoscheduling algorithms for loop fusion and tile-size selection do not employ a rich cost model for GPUs. For example, cost models in [17, 24] do not consider the number of global memory transactions, the ability to hide latency of global memory accesses, and occupancy.

We present PolyMage-GPU (based on PolyMage [19]), a compiler for image processing pipelines that leverages the architecture of modern GPUs to generates high performance code. PolyMage-GPU exploits the fact that all threads in a *warp* can synchronize using warp synchronization, which has significantly lower overhead than thread block synchronization. In addition, modern GPUs have *warp shuffle* [1, Chapter B.16] instructions that allow threads in a warp to read each others' register values. PolyMage-GPU uses warp shuffles to lower shared memory usage and support larger overlapped tiles. Finally, we develop a cost model for GPUs that accounts for

several factors, including the number of global memory transactions, occupancy, and resource utilization. We use this cost model to determine the optimal tile and thread block sizes and loops to fuse, using Dynamic Programming Fusion [14].

To summarize, this paper 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*, which stores portions of a tile in registers instead of shared memory, which reduces the fraction of redundant computations, and reduces shared memory utilization. This improves performance by decreasing global memory loads and increasing occupancy. 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 aforementioned techniques in PolyMage-GPU, which is a new GPU backend for PolyMage [19], which is a DSL embedded in Python for writing image processing pipelines.
(5) Using established benchmarks, we compare our approach to manually written schedules in Halide. On a GeForce GTX 1080Ti, we achieve a speedup of 1.65× over manual schedules and on a Tesla V100, we achieve a speedup of 1.33× over manual schedules.

The rest of this paper is organized as follows. Section 2 discusses the architecture of NVIDIA GPUs, the PolyMage DSL, and Dynamic Programming Fusion. Section 3 presents an overview of our approach. Section 4 presents our technique for running one overlapped tile per warp. Section 5 presents hybrid tiling. Section 6 presents our automatic fusion algorithm. Section 7 evaluates our work. Section 8 discusses related work. Finally, Section 9 concludes.

Our implementation is available at bitbucket.org/abhijangda/polymage-gpu.

2 BACKGROUND

This section first presents the essentials of GPU architecture that are necessary for our work. We then present the PolyMage DSL for writing image processing programs, and two key ideas that it employs: dependence vectors and dynamic programming fusion.

2.1 NVIDIA GPU Architecture

An NVIDIA GPU consists of several Simultaneous Multiprocessors (SM) that execute one or more thread blocks. Each SM consists of several CUDA cores, shared memory, and registers. The number of warps that an SM can execute concurrently depends on properties of the running CUDA kernel: the number of thread blocks it has, the number of threads per thread block, the shared memory used by each thread block, and the registers used by each thread. The *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.

```
1 int val = rand ();
2 for (int offset = 16; offset > 0;offset /= 2)
3   val += __shfl_sync(0xffffffff, val,
4                      threadIdx.x+offset, warpSize);
```

Figure 1: CUDA kernel invoked with 32 threads in x-dimension. At each iteration, each thread add next `offset` thread's `val` to its `val`. At the end of loop, `val` of the first thread contains the sum.

CUDA threads can synchronize in two ways. *Thread block synchronization* synchronizes all threads in a block: until all warps in the block reach the same `__syncthreads` statement, no warp is allowed to proceed. However, as mentioned above, when a warp is stalled on a global memory access, the SM tries to run another warp. Thread block synchronization can force an SM to idle if all warps are waiting for memory accesses to be satisfied. Contemporary current stencil code generators for GPUs use thread block synchronization between producer-consumer stages (Section 8). In contrast, *warp synchronization* synchronizes all threads in a warp, and no thread can proceed until all threads in the warp reach the synchronization point (`__syncwarp`). However, other warps in the same thread block can make progress, thus it is more lightweight than thread block synchronization.

The *warp shuffle* instructions [1, Chapter B.16][5] available in recent AMD and NVIDIA GPUs allow threads to read register values from other threads in the same warp. The `__shfl_sync` instruction takes four arguments: a 32-bit mask of threads participating in the shuffle, the variable stored in the register to read, the index of the source thread containing the register, and 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 [4] of reduction using `__shfl_sync`. For a shuffle to succeed both the calling thread and source thread must execute the instruction.

2.2 PolyMage DSL

PolyMage [19] is a DSL embedded in Python for writing image processing pipelines. The PolyMage compiler transforms programs in the DSL into high-performance code for CPUs. 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 24). 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 19–22). `blury` blurs the output of `blurx` in the y-direction and produces final output (lines 24–26). The PolyMage compiler performs loop fusion on producer-consumer stages to improve locality and provide parallel execution. When fusing two stages, PolyMage performs overlapped 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

```
R,C = Parameter(Int,"R"),Parameter(Int,"C")

# Vars
x = Variable(Int,"x")
y = Variable(Int,"y")
c = Variable(Int,"c")

# Input Image
img = Image(Float,"img",[3,R+2,C+2])

# Intervals
cr = Interval(Int,0,2)
xrow,xcol = Interval(Int,1,R),Interval(Int,0,C+1)
yrow,ycol = Interval(Int,1,R),Interval(Int,1,C)

cond = Condition(x,'>=',1) & Condition(x,'<=',R) &
       Condition(y,'<=',C) & Condition(y,'>=',1)

blurx = Function(([c,x,y],[cr,xrow,xcol]),
                 Float,"blurx")
blurx.defn = [Case(cond,(img(c,x-1,y) + img(c,x,y) +
                         img(c,x+1,y))/3)]
blury = Function(([c,x,y],[cr,yrow,ycol]),
                 Float,"blury")
blury.defn = [Case(cond,(blurx(c,x,y-1) +
               blurx(c,x,y) + blurx(c,x,y+1))/3)]
```

Figure 2: PolyMage DSL specification for *blur*.

```
1  blur_otptb(img[3][R][C], blury[3][R-2][C-2])
2    shared blurx[blockDim.y][tile*blockDim.x+2];
3    c = threadIdx.z
4    y = blockIdx.y*blockDim.y + threadIdx.y
5    for (tx = 0; tx < tile+1; tx++)
6      xx = tx * blockDim.x + threadIdx.x
7      x = (blockIdx.x*blockDim.x)*tx + xx
8      if (xx < tile*blockDim.x+2)
9        blurx[y][xx] = (img[c][y-1][x]+img[c][y][x]+
10                       img[c][y+1][x])/3
11   __syncthreads ();
12   for (tx = 0; tx < tile; tx++)
13     xx = tx * blockDim.x + threadIdx.x
14     x = (blockIdx.x*blockDim.x)*tx + xx
15     blury[c][y][x] = (blurx[y][xx-1]+blurx[y][xx]+
16                       blurx[y][xx+1])/3
```

Figure 3: Equivalent CUDA code generated by Halide for *blur*. Both *blurx* and *blury* are fused in an overlapped tile of size `tile` in x and 1 in y, which is computed by one thread block.

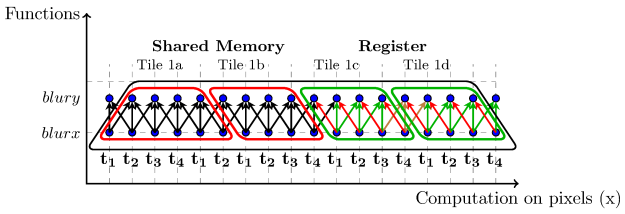

Figure 4: Hybrid Tiling for *blur* program with tile size of 2 in x-dimension and warp size of 4. The overlapped tile is split into four tiles. Tiles in red are stored in shared memory and tiles in green are stored in registers. Each point of `blurx` is computed and stored in the register of same thread represented by t_i. The producer loads in red are from the registers of current thread (Type ①), black are from shared memory (Type ②), green are from the registers of another thread in same parallelogram tile (Type ③), and brown are from registers of another thread in previous parallelogram tile (Type ④).

consumer using small buffers, known as *scratchpads*. A scratchpad is small enough to fit in a CPU cache, or in our work, in GPU shared memory or registers.

2.3 Dependence Vectors

PolyMage uses dependence vectors to encode the dependencies between consumer and producer stages. A *dependence vector* [32] is the difference of the time stamps when a value is consumed and when it is produced. For example, in the *blur* program, the `blury` stage, at `(2,c,x,y)`, consumes values that the `blurx` stage produces at `(1,c,x,y-1)`, `(1,c,x,y)`, and `(1,c,x,y+1)`. This is captured by the dependence vectors (1,0,0,-1), (1,0,0,0), and (1,0,0,1).

2.4 Dynamic Programming Fusion

Dynamic Programming Fusion (DP-Fusion) [14] is an algorithm that performs automatic fusion of image processing pipelines in a few seconds. DP-Fusion finds schedules that are competitive with take days for an autotuner, and are better than a greedy CPU autoscheduler [18]. Instead of using a greedy algorithm and a simple cost function, 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. PolyMage uses DP-Fusion to find the best schedules for image processing programs executing on multi-core CPUs [14]. In this paper, we present a cost model for GPUs that integrates with DP-Fusion.

3 OVERVIEW

Figure 3 shows CUDA code that is equivalent to the code that Halide produces for *blur*. The code fuses both *blurx* and *blury* together and uses overlapped tiles of length `tile` in the x-dimension and unit length in y-dimension. During execution, all threads in a thread block 1) compute *blurx* in parallel by looping over all points in the tile (lines 5–10), 2) store the result of *blurx* in a scratchpad (which is in shared memory), 3) use thread-block synchronization to ensure that all *blurx* values are ready (line 11), and 4) calculates *blury* in parallel, which depends on *blurx* (line 12– 16). On an NVIDIA GTX 1080Ti, this code exhibits its best performance (1.40ms) on a 4096×4096×3 input with 8 tiles and block sizes of 64×4×1. However, thread block synchronization can lower occupancy, so there is room for improvement.

Overlap Tile per Warp (OTPW). We modify the program to assign each overlapped tile to a warp, instead of a thread block. This change allows us to use warp synchronization (`__syncwarp`), which allows the SM to execute a one warp even if another warp is waiting for a memory access. This code exhibits its best performance (1.35ms) with 8 tiles and block sizes of 64×4×1. This is a 1.04× speedup over the prior approach. This choice of tile size produces 0.8%

```
1  blur_otpw_ht(img[3][R][C], blury[3][R-2][C-2])
2    shared blurx[blockDim.y][blockDim.x/warpSz]
3                [tile/2*warpSz+2];
4    y = blockIdx.y * blockDim.y+threadIdx.y;
5    c = threadIdx.z;
6    warpSz = warpSize;
7    warp = threadIdx.x/warpSz;
8
9    for(tx = 0; tx < 2; tx++)
10    for(txx = tx*tile/4;txx<(tx+1)*tile/4+1;txx++)
11     xx = tile*warpSz+threadIdx.x%warpSz;
12     x = (blockIdx.x+1)*blockDim.x*tx+threadIdx.x;
13     if(xx < tile*warpSz+2)
14      blurx[y][warp][xx] = (img[c][y-1][x]+
15      img[c][y][x]+img[c][y+1][x])/3;
16   x = warp_idx+8*warpSz+lane_id_x;
17   blurx_8 = (img[c][y-1][x]+img[c][y][x] +
18              img[c][y+1][x])/3;
19   x = warp_idx+9*warpSz+lane_id_x;
20   blurx_9 = (img[c][y-1][x]+img[c][y][x]+
21              img[c][y+1][x])/3;
22   /*similarly, for all iterations till 15*/
23   syncwarp();
24   for(tx = 0; tx < 2; tx++)
25    for(txx = tx*tile/4; txx <(tx+1)*tile/4; txx++)
26     xx = tile*warpSz+threadIdx.x%warpSz;
27     x = (blockIdx.x+1)*blockDim.x*tx+threadIdx.x;
28     if(xx > 0 and xx < tile/2*warpSz+2)
29      blury[c][y][x] = (blurx[y][warp][xx-1]+
30       blurx[y][warp][xx]+blurx[y][warp][xx+1])/3;
31   blurx_l_2_8 = shfl_up(FULL_MASK, blurx_8, 2);
32   blurx_l_1_8 = shfl_up(FULL_MASK, blurx_8, 1);
33   if(lane_id_x == 0)
34    blurx_l_2_8=blurx[y][warp][7*warpSz+warpSz-2];
35    blurx_l_1_8=blurx[y][warp][7*warpSz+warpSz-1];
36   if(lane_id_x == 1)
37    blurx_l_2_8=blurx[y][warp][7*warpSz+warpSz-1];
38   x = warp_idx+8*warpSz+lane_id_x;
39   blury[c][y][x] = (blurx_l_2_8+blurx_l_1_8+
40                     blurx_8)/3;
41   blurx_l_2_9 = shfl_up(FULL_MASK, blurx_9, 2);
42   blurx_l_1_9 = shfl_up(FULL_MASK, blurx_9, 1);
43   _blurx_l_2_9=shfl(FULL_MASK, blurx_8, warpSz-2);
44   _blurx_l_1_9=shfl(FULL_MASK, blurx_8, warpSz-1);
45   if(lane_id_x == 0)
46     blurx_l_1_9 = _blurx_l_1_9;
47     blurx_l_2_9 = _blurx_l_2_9;
48   if(lane_id_x == 1) blurx_l_2_9 = _blurx_l_1_9;
49   x = warp_idx+9*warpSz+lane_id_x;
50   blury[c][y][x] = (blurx_l_2_9+blurx_l_1_9+
51                     blurx_9)/3;
52   /*similarly, for all iterations till 15*/
```

Figure 5: CUDA code for *blur*, with *blurx* and *blury* fused in an overlapped tile of size `tile` in the *x*-dimension, which is computed by one warp. The first half of the tile is stored in shared memory with latter half in registers. In this code `shfl*` refers to `__shfl*_sync`.

redundant computations per warp. We can achieve fewer redundant computations (0.4%) with tile size 16, but that increases running time (1.45ms) because it consumes far more shared memory (over 16KB). This limits the number of warps that the GPU can run concurrently, i.e., occupancy is only 62.5%.

Hybrid Tiling. To further improve performance, we introduce *hybrid tiling*, which is a technique that decreases the size of the scratchpad buffer in shared memory, by storing some parts of the overlapped tile in registers. In the earlier approaches, we employed the scratchpad to share values between consumers (*blury*) and producers (*blurx*) in a thread block. However, since we now assign each tile to a warp, we can use *warp shuffle* instructions that allow threads in a warp to read register values from other threads in the same warp. This eliminates the need for per thread redundant computation that arise in register blocking. Figure 4 sketches the structure of the computation, assuming four tiles: the first two tiles are stored in shared memory, whereas the latter two tiles are stored in registers. When a *blury* value depends on a *blurx*-value in a register, it can read it directly, using warp shuffles to read across threads if needed.

On a GTX 1080Ti, the code so far only uses 24 registers. With a tile size of 16, we can store half of the tile in registers, which halves the shared memory usage, and leads to 100% occupancy. With hybrid tiling, the code runs in 1.2ms which is 1.13× faster than the *OTPW* approach, and 1.16× faster than the original program.

Figure 5 sketches the CUDA code for *blur* that uses *overlap tile per warp* and *hybrid tiling*. In the figure, the data points of *blurx* for first two tiles are stored in shared memory while the later tiles are stored in the registers. Lines 9–15 processes *blurx* on the first two tiles stored in shared memory using a warp by assigning consecutive data points to consecutive threads in a warp and looping over all points in both tiles. Lines 16–22 unroll the loop and store each data point in registers for two register tiles. Lines 24–30 compute the values of *blury* for first two tiles that are stored in shared memory. Lines 31–32 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 33–37 retrieve the last two values of shared memory for that warp. Line 40 computes each *blury* point for the eighth iteration of the larger overlapped tile. Similarly, for the ninth iteration, lines 41–48 retrieve the values of `blurx_9` from previous threads and for first two threads of warp values of `blurx_8` are retrieved from last two threads of the warp. We generate code for the remaining six iterations in the same manner.

Loop Fusion. The final problem involves choosing tile and block sizes. We present an automatic fusion algorithm that considers key factors affecting the performance of GPU kernels which are not considered in previous work [6, 17, 18]: 1) number of global memory transactions, 2) achieved and theoretical occupancy, 3) GPU resource usage, and 4) fraction of overlapping computations.

We implement OTPW, hybrid tiling, and our new new fusion algorithm in PolyMage-GPU. Figure 6 shows the structure of the compilation pipeline. In summary, our approach uses low cost synchronization, distributes tile in shared memory and registers, decreases shared memory usage, and enables larger tiles to decrease number of overlapping computations without any loss in occupancy. We also address the problem of fusing pipeline stages and choosing tile and thread block sizes automatically.

4 OVERLAP TILE PER WARP

In this section, we describe how we calculate 1) the tile size for each stage, 2) the assignment of input data points to threads, 3) the size of the output scratchpad, and 4) the fraction of overlap.

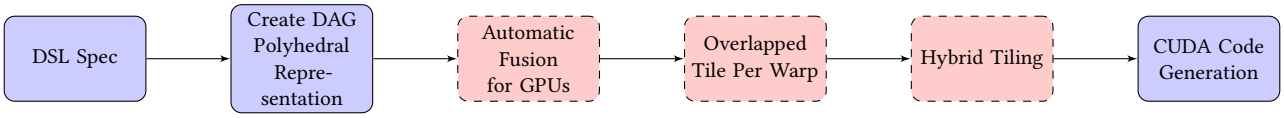

Figure 6: Compilation pipeline of image processing program written in PolyMage-GPU, which is based on PolyMage [19]. The three phases in middle with dashed rectangles are the new phases of PolyMage-GPU (Sections 4–6).

Let (b_x, b_y, b_z) be the coordinates of a thread block (B_x, B_y, B_z) be the thread block size. Consider a group of fused stages with tile sizes (T_x, T_y, T_z) that consumes a three-dimensional input of size (N_x, N_y, N_z), where each dimension is labelled $i \in \{x, y, z\}$. We convert the three-dimensional coordinates of a thread (t_x, t_y, t_z) to a 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. We define warp sizes, W_x, W_y, W_z such that:

$$\begin{aligned} W_x &= \texttt{minimum}(B_x, \texttt{WarpSize}) \\ W_y &= \texttt{minimum}(B_y, \texttt{WarpSize} \div W_x) \\ W_z &= \texttt{minimum}(B_z, \texttt{WarpSize} \div (W_x \times W_y)) \end{aligned}$$

In these equations we assume that number of threads in a thread block are a multiple of `WarpSize`. (We add extra threads as padding if needed.) These warp sizes are the number of threads with distinct IDs of that dimension in a warp. The number of warps in dimension i in a thread block is equal to the ratio of block size to the warp size of that dimension ($\lceil B_i/W_i \rceil$). 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.($\lfloor (b_i \times B_i + t_i)/W_i \rfloor$). Moreover, the lane ID is the remainder ($(b_i \times B_i + t_i) \bmod W_i$). Note that product of all the warp sizes obtained using these equations is equal to `WarpSize`. For 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 the product of the number of points in the given overlapped tile sizes and `WarpSize`. For the given overlapped tile size, the size of the warp overlapped tile is $(T_x \times W_x, T_y \times W_y, T_z \times W_z)$. For example, if the tile size is (8,4,1), block size is (16,8,1), then the warp size will be (16,2,1) and the warp overlapped tile size will be (128,8,1).

Tiling a dimension produces two dimensions: an outer dimension that is iterated from the number of tiles and an inner dimension that is iterated tile size times. We initialize the outer dimension to the warp ID of that dimension, and the inner dimension to the sum of the lane ID and the product of current tile iteration and `WarpSize`. To process each warp tile, we assign consecutive threads in the i^{th} dimension to 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_i^n overlapping computations. We represent the number of data points computed (and the size of the scratchpad) for n^{th} stage as $\prod_{i \in \{x,y,z\}} \lceil B_i/W_i \rceil \times (T_i \times W_i + O_i^n)$.

Since tiling introduces extra conditional branches and arithmetic instructions, we do not perform OTPW in a dimension when the warp size in that dimension is 1. However, as long as the group of stages processes more than one input point, at least one dimension will have warp size greater than 1.

5 HYBRID TILING

In this section we present *hybrid tiling*, which divides a tile between shared memory and registers. Hybrid tiling riles on the fact that each overlapped tile fits in a single wrap. We use *warp shuffle* instructions to allow each thread to access data from other threads in a warp, which eliminates the need for certain redundant computations per thread. Hybrid tiling solves the issues of shared memory only tiling by 1) storing a part of a tile in registers to decrease allocated shared memory, 2) providing extra storage for larger tile sizes, which results in fewer redundant computations, and in turn, fewer global memory loads and total computations; and 3) storing tiles partially in registers, which leads to faster access to data points.

We split the warp overlapped tile over a *split dimension*, into several parallelogram tiles with left tiles stored in shared memory and right tiles stored in registers (Figure 4). These smaller parallelogram tiles are of warp size in the split dimension, and the same size as the 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, which ensures there is no cyclic dependence between two adjacent tiles. The left parallelogram tiles, including the overlap on the left side, are stored in shared memory. Since the right tiles depend on left tiles, we must process the left tiles first.

Since all producer loads by *OTPW* are in the shared memory, we need to convert these loads to access data stored in registers if necessary. Figure 4 shows that there are four types of producer load: ① is a load from a register of the current thread, if the load index is same as the iteration in the split dimension; ② is a load from shared memory, if the load index is less than the lower bound of the register tile in the split dimension; ③ is a load from another thread's register in same tile, if the load index in the split dimension is less than the iteration in the split dimension; and ④ is a load from another thread's register from the previous tile, if the difference between the lane ID of the current thread in the split dimension and the difference between the iteration and load index in the split dimension is less than zero.

We now present the code generation algorithm that uses dependence vectors between producer and consumer stages. Before executing the hybrid tiling algorithm, we use PolyMage's alignment and scaling to make the dependence vectors between each producer-consumer pair constant. Algorithm 1 is our hybrid tiling algorithm. For simplicity, we present the algorithm making two assumptions. First, we assume that the x-dimension is the split

dimension. Second, we assume that the difference between any two dependence vectors in the same dimension after alignment and scaling is less than the warp size. Several of our benchmarks satisfy these assumptions. However, it is straightforward to generalize the algorithm [15, Appendix A].

The arguments to the 2-D-HybridTiling function are the group of stages (G), tile sizes ($T_x \times T_y$), warp sizes ($W_x \times W_y$), and register tile size (*fracReg*) as a fraction of the tile size in the split dimension. The result of the function is CUDA code that does hybrid tiling. First, the algorithm finds a split dimension with tile size greater than 1 (line 15). If no such dimension is found, then tiles must be stored entirely in the shared memory. The rest of the algorithm assumes that the x-dimension is the split dimension. Let ϕ_{rx} and ϕ_{ry} be the right hyperplanes of warp overlapped tiles of G in the x and y dimensions respectively. We first generate the shared memory tile using the PolyMage compiler, and then generate register tiles using the GenRegTile function that takes a stage of the group (H), the hyperplanes (ϕ_{rx}, ϕ_{ry}), the register tile size ($R_x \times R_y$), and the warp sizes ($W_x \times W_y$) as arguments (lines 23–24).

For all the iterations in the register tile, including the overlapping computations, we store each computed value of stage H in a distinct variable, instead of shared memory (line 5). We replace each producer load in the loop is replaced with either a shared memory read or a warp shuffle (lines 6–13). We get the dependence vector between the producer and consumer (line 7) as ϕ_x and ϕ_y. (Note that $\phi_x \leq \phi_{rx}$ and $\phi_y \leq \phi_{ry}$ due to overlapped tiling algorithm.) Figure 7 shows the code generated for three cases that arise when generating code for a load `P[a*x+b][c*y+d]`. The figure shows two types of source lane IDs that contain the register, which stores the value of the producer load: 1) `currTileSrcLane` is the lane ID for a source thread in the current parallelogram tile and 2) `prevTileSrcLane` is the lane ID for a source thread in the previous parallelogram tile. Value of both ids in x-dimension depends on $\phi_x - \phi_{rx}$ and in y-dimension depends on $\phi_y - \phi_{ry}$. We now explain each of the three cases in detail. 1) If $\phi_x = \phi_{rx}$, then the value needed for this load is stored by the current thread's register and we generate the code for Type ① (line 9). 2) When $\phi_x - \phi_{rx} \neq 0$ and the iteration in split dimension, i.e., x-dimension is first iteration of the register tile, then first $|\phi_x - \phi_{rx}|$ threads of warp loads from shared memory (Type ②) and remaining threads loads from registers of threads in same parallelogram tile (Type ③). Figure 7b shows the code generated for this case. The conditional determines whether to load from shared memory or from another thread's register. The `__shfl_sync` function loads the value from the source thread's register. The function `getMask` retrieves the mask of threads that can participate in the warp shuffle. 3) Otherwise, if a thread needs to load from another thread's register that stores value of either the current parallelogram tile (Type ③) or the previous parallelogram tile (Type ④), then we generate the code in Figure 7c. Two warp shuffles are generated that are executed by all threads and a conditional expression selects which loaded value to use.

Instead of generating register array, PolyMage-GPU generates a register access by computing the value of `a*x+b` and `c*y+d` for given register tile iteration {x,y} and converts these values to strings. Hence, it produces explicit variable names for each element of the register array.

```
val = Reg_P[x][c*y + d]
```

(a) Code for a register access from same thread is generated when the source lane ID is the current lane ID, i.e. $\phi_x = \phi_{rx}$ (Type ①).

```
currTileSrcLane = (laneId.x + diffPhi.x) +
  (laneId.y + diffPhi.y)*warpSize.x;
/*Type 3:*/ val = __shfl_sync(getMask(),
            Reg_P[x][c*y+d], currTileSrcLane);
if (laneId.x + diffPhi.x < 0)
  /*Type 2:*/ val = ShMem_P[a*x+b][c*y+d];
```

(b) Code generated when current iteration is the first iteration of register tile and $\phi_x - \phi_{rx} \neq 0$. When the sum of lane index and $\phi_x - \phi_{rx}$ is less than zero, then value is accessed from shared memory tile (Type ②), otherwise value is accessed from register of thread in the same parallelogram tile (Type ③).

```
prevTileSrcLane = (warpSize.x - 1 + diffPhi.x)+
  (warpSize.y - 1 + diffPhi.y)*warpSize.x;
currTileSrcLane = (laneId.x + diffPhi.x) +
  (laneId.y + diffPhi.y)*warpSize.x;
/*Type 3:*/ val = __shfl_sync(getMask(),
              Reg_P[x][c*y+d], currTileSrcLane);
if (laneId.x + diffPhi.x < 0)
  /*Type 4:*/ val = __shfl_sync(getMask(),
              Reg_P[x-1][c*y+d], prevTileSrcLane);
```

(c) Code generated when current iteration is not the first iteration of register tile and $\phi_x - \phi_{rx} \neq 0$. When the sum of lane index and $\phi_x - \phi_{rx}$ is less than zero, then value is accessed from register of last $|\phi_x - \phi_{rx}|$ threads of previous parallelogram tile (Type ④) otherwise value is accessed from register of thread in the same parallelogram tile (Type ③).

Figure 7: Three code generation cases for a producer `p[a*x+b][c*y+d]` at iteration {x, y} of register tile that generates all four load types of Figure 4. Each `p[a*x+b][c*y+d]` of register tile is replaced with `val` and one of the above the code is added. `Reg_P` is the register array storing register tile of `p`. `laneId.x` and `laneId.y` are the lane indices in `x` and `y` dimensions of the current thread. `warpSize.x` and `warpSize.y` are the warp sizes in `x` and `y` dimensions. `diffPhi.x` is the value of $\phi_x - \phi_{rx}$. `diffPhi.y` is the value of $\phi_y - \phi_{ry}$.

Finally, PolyMage-GPU prevents out of bounds accesses in hybrid tiling in two ways. First, the image sizes in the generated CUDA code are treated as parameters that are passed to each CUDA kernel. Hence, the bounds of each stage and the number of tiles depends on the image sizes. Second, before computation of every iteration of each stage, PolyMage's compiler adds conditionals to ensure that for the given image sizes, the tile lies within the correct bounds of current stage. These conditionals will prevent out of bounds accesses if the generated code is used for different image sizes.

Register Blocking. Register blocking [31] is a well-known technique that stores tile in registers of parallel threads. However, it generates one overlapped tile per thread, leading to redundant computations between all threads. In contrast, Hybrid Tiling eliminates these redundant computations by utilizing warp synchronous behavior of threads and warp shuffles to access shared memory and the registers of another thread.

Algorithm 1 Hybrid Tiling

1: **function** GenRegTile(H, ϕ_{rx}, ϕ_{ry}, $R_x \times R_y$, $W_x \times W_y$)
2: **for all** {x, y} ∈ [1, …R_x] × [1, …R_y] **do**
3: Let iteration {x, y} be
4: `H[x][y] = f(P[a*x+b][c*y+d], ...)`
5: Store `H[x][y]` in a register array `Reg_H[x][y]`
6: **for all** loads `P[a*x+b][c*y+d]` ∈ `f` **do**
7: ϕ_x, ϕ_y = dependence vectors between `P[a*x+b][c*y+d]` and `H[x][y]`
8: **if** $\phi_x == \phi_{rx}$ **then**
9: Generate Type① code in Figure 7a
10: **else if** x == 1 **then**
11: Generate Type ② and ③ code from Figure 7b
12: **else**
13: Generate Type ③ and ④ code from Figure 7c
14: **function** 2-D-HybridTiling(G, fracReg, $T_x \times T_y$, $W_x \times W_y$)
15: splitDim = *a dimension with tile size greater than 1*
16: *If no split dimension exists then* **return**
17: *Let* ϕ_{rx} *and* ϕ_{ry} *be right hyperplanes of G in* x *and* y
18: *Let* splitDim is the x*-dimension.*
19: *Create parallelogram tiles in* x*-dim of size* W_x *parallel to* ϕ_{rx}
20: $R_x \leftarrow T_x \times$ fracReg, $S_x \leftarrow T_x \times$ (1 - fracReg)
21: $R_y \leftarrow S_y \leftarrow T_y$
22: **for all** H ∈ G **do**
23: *Gen. Shared Mem Tile with tile size* $S_x \times S_y$
24: GenRegTile(H, ϕ_{rx}, ϕ_{ry}, $R_x \times R_y$, $W_x \times W_y$)

6 AUTOMATIC FUSION FOR GPUS

In this section, we present an automatic fusion algorithm that selects 1) sets of stages to fuse, 2) their tile sizes, and 3) their thread block sizes. Our approach leverages DP-Fusion [14], 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.

The inputs to our algorithm include the register usage and running time of each stage, prior to fusion. We gather this information by generating code for each individual stage, where global memory loads are replaced with shared memory loads, loops perform a single iteration, and the outermost loop is nested inside a loop with a large number of iterations (e.g., one million), to ensure that time measurements are correct. We obtain the time for each iteration by measuring the time taken to execute the kernel by one thread block, with one thread and divide this by the number of loop iterations. We measure the register usage of each stage with `nvcc`.

Algorithm 2 is our cost function, and itt takes four arguments: 1) a group of stages to fuse, G, 2) tile sizes, 3) thread block sizes, 4) fraction of tile stored in registers, and returns the cost. The function refers to the the hardware configuration of a GPU (Table 1). The expression below calls the Cost function for all tile sizes, thread block sizes, and fraction of tile stored in registers including 0.0 (hybrid tiling disabled) and 1.0 (except the overlap in split dimension the complete tile is stored in registers), and global memory transaction size for both L1 and L2 global memory cache, and returns the minimum cost with the appropriate global memory cache enabled, tile sizes, thread block sizes, and the fraction of tile stored in registers:

$$\underset{\substack{\text{tileSize}\in\text{Tile Sizes},\\ \text{tbSize}\in\text{Thread Block Sizes},\\ \text{fracRegTile}\in\{0.0,0.1,\ldots,1.0\},\\ \texttt{GLMemTxSize}\in\{32,128\}}}{\text{argmin}} \quad \text{Cost (G, tileSize, tbSize, fracRegTile)}$$

Model	GTX 1080Ti	Tesla V100
Simultaneous Multiprocessors (`NSMs`)	28	80
CUDA Cores per SM (`CoresPerSM`)	128	64
Global Memory Bandwidth (`GlMemBW`)	484 GBps	898 GBps
Maximum Shared Memory Per Thread Block (`MaxShMemPerTb`)	48 KB	96 KB
Shared Memory per SM (`ShMemPerSM`)	96 KB	
Maximum Warps per SM (`MaxWarpPerSM`)	64	
Maximum Thread Blocks per SM (`MaxTbPerSM`)	16	32
Registers per SM (`RegPerSM`)	65536	
Maximum Registers Per Thread (`MaxRegPerTh`)	256	
Warp Size (`WarpSize`)	32	
Global Memory Transaction Size (`GlMemTxSz`)	32 B for L2 Cache 128 B for L1 Cache	

Table 1: Specifications of the GPUs we use in experiments.

The Cost function determines the cost (line 35) based on 1) the number of global memory transactions per warp, 2) theoretical maximum occupancy, 3) achieved occupancy, 4) shared memory usage, 5) register usage, 6) the fraction of redundant computations, and 7) the load imbalance. We calculate the weighted sum of these factors to determine the cost. The function also ensures the dependence vectors between all stages of a group are constants after alignment and scaling of dependencies (line 2). The function determines the dimension sizes of the group, total threads created, threads per thread block, number of warps per thread block, and warp overlapped tile sizes (lines 3–5). We distribute all thread blocks equally across all SMs (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, the function splits the shared memory tile into two parts and updates the register tile (line 11). We check if the shared memory used per thread block is more than the maximum shared memory (line 12).

The rest of this section describes how we calculate the weight of each component of the cost.

Number of Global Memory Transactions. The cost function estimates the number of global memory transactions that either load input images or inputs to the group (lines 14–17). The number of global memory transactions depends on tile sizes, thread block sizes, and the global memory transaction size. Higher global memory transaction size is beneficial when all values loaded from the global memory are used by the group. If not all loaded values are used in the group, then it is better to use a smaller transaction size.

Algorithm 2 Cost Function

```
1: function Cost(G, tileSize, tbSize, isHybridTile, fracRegTile)
2:     if not constantDependenceVectors(G) then return ∞
3:     totalThreads ← TotalThreads(GetDimSizes(G), tileSize)
4:     warpTileSizes ← WarpTile(tileSizes, WarpSizes(tbSize))
5:     warpsPerTB ← ThreadsPerTB(tbSize) ÷ WarpSize
6:     tbPerSM ← totalThreads ÷ ThreadsPerTB(tbSize) ÷ NSMs
7:     warpTileVol ← ComputeTileVol(G, warpTileSizes)
8:     totalBuff ← NumBuffers(G)
9:     shMemPerTB ← warpTileVol × warpsPerTB × totalBuff
10:    shMemPerTB ← shMemPerTB ×(1 - fracRegTile)
11:    regTile←shMemPerTB×fracRegTile÷tbSize
12:    if shMemPerTB > MaxShMemPerTb then return ∞
13:    totalGLMemTxs ← 0
14:    for all glLoad ∈ GetGlobalMemLoads(G) do
15:        warpLoad ← GLLoadsInWarp(glLoad, tileSize, tbSize)
16:        glTxs ← MinGLTxs(warpLoad, GlMemTxSz)
17:        totalGLMemTxs ← totalGLMemTxs + glTxs × tileVol
18:    maxTBPerSM ← min(ShMemPerSM / shMemPerTB, MaxTbPerSM)
19:    shMemOcc ← min(maxTBPerSM × warpsPerTB, MaxWarpPerSM)
20:    regPerTh ← regTile + ∑_{H∈G} RegUsage(H)
21:    if regPerTh > MaxRegPerTh then return ∞
22:    maxThPerSM ← min(RegPerSM / regPerTh, MaxThPerSM)
23:    regOcc ← maxThPerSM ÷ WarpSize
24:    occupancy ← min(shMemOcc, regOcc) ÷ MaxWarpPerSM
25:    warpBW ← GlMemBW × WarpSize ÷ NSMs × CoresPerSM
26:    memTime ← GlMemTxSz×totalGLMemTxs ÷ warpBW
27:    tileVol ← ComputeTileVol(G, tileSizes)
28:    computeTime ← ∑_{H∈G} TimePerIter(H)×tileVol
29:    shMemPerSM ← shMemPerTB × maxTBPerSM
30:    unallocatedShMem ← 1 − shMemPerSM÷ ShMemPerSM
31:    regPerSM ← regPerTh × MaxWarpPerSM× WarpSize
32:    unusedReg ← 1 − regPerSM× occupancy÷RegPerSM
33:    fracOverlap ← OverlapComputations(G)÷ tileVol
34:    extraTBs ← totalTB % maxTBPerSM
35:    cost = w1×totalGLMemTxs + w2×(1 − occupancy) + w3× memTime
       ÷ computeTime + w4×unallocatedShMem + w5×unusedReg + w6×
       fracOverlap + w7× extraTBs
36:    return cost
```

Using Wolf and Lam [30], we retrieve the loads for each global memory load for all threads in a warp (line 15). We coalesce all memory loads into the minimum number of transactions (line 16). Finally, we calculate the total number of transactions (line 17).

Theoretical Occupancy. We estimate theoretical occupancy based on shared memory and register utilization. We calculate the maximum number of thread blocks supported by an SM based on the shared memory usage and take its minimum with MaxTbPerSM (line 18). Multiplying this value with number of warps per thread block gives the occupancy from shared memory usage (line 19). We sum the register usage of all stages in the group from in *preprocessing* step to get the register usage of the group (line 20). 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 MaxWarpPerSM (lines 22–23). The ratio of minimum of both occupancies to MaxWarpPerSM is the theoretical occupancy (line 24).

Achieved Occupancy. The cost function estimates the number of warps ready to execute at runtime as 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 global 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 25). Hence, this produces the time spent in all global memory transactions (line 26). 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* step 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 28).

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 30–32) when all thread blocks are executing concurrently based on the occupancy.

Fraction of Redundant Computations. The cost function determines the fraction of overlap (line 33).

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 34 determines the extra thread blocks for each SM.

7 EVALUATION

In this section, we investigate the following questions: 1) How fast is our automatic loop fusion algorithm? 2) How does the *OTPW* execution model compare to the state-of-the-art? 3) How do *OTPW with Hybrid Tiling* compare to the state-of-art? 4) Why do *OTPW and Hybrid Tiling* perform well?

Experimental Setup. We use a 3.4 GHz, quad-core Intel i5-4670 CPU with 16GB RAM and two GPUs (each experiment uses a single GPU): an NVIDIA GTX 1080Ti and an NVIDIA Tesla V100 (Table 1 lists their key specifications). For our benchmarks, we use six canonical image processing applications that have appeared in prior work [6, 14, 18, 19, 22]. Table 2 reports the number of stages and the size of the input image for each benchmark. We compare our work to the manually-written schedules present in Halide repository [2], Li et al.'s autoscheduler for Halide [17], Rawat et al.'s code generator [24], and PolyMage's autotuner. We compiled Halide with LLVM 10.0. The execution time that we report for each benchmark is the sum of execution time of all generated CUDA kernels (obtained using `nvprof`), and does not include host and device memory transfer time. We execute each benchmark for three samples with each sample containing 100 runs. We report the minimum of the average running time for each sample.

Benchmark	Stages	Image size (W×H×c)	Fusion
Unsharp Mask (UM)	4	4256×2832×3	0.05s
Harris Corner (HC)	11	4256×2832	0.15s
Bilateral Grid (BG)	7	2560×1536	0.02s
Multiscale Interp. (MI)	49	2560×1536×3	10s
Camera Pipeline (CP)	32	2592×1968	17s
Pyramid Blend (PB)	44	3840×2160×3	28s

Table 2: For each benchmark, the number of stages, size of input, and time taken for loop fusion.

	w_1	w_2	w_3	w_4	w_5	w_6	w_7
GTX 1080Ti	50	0.5	45	20	2	100	1
Tesla V100	50	0.5	60	10	2	100	1

Table 3: Value of weights obtained for both GPUs.

Benchmark	Halide		PolyMage-GPU		Speedup	
	1080Ti	V100	1080Ti	V100	1080Ti	V100
Unsharp Mask	1.50	0.45	1.00	0.39	1.50	1.15
Harris Corner	1.80	0.45	0.80	0.29	2.25	1.55
Bilateral Grid	0.40	0.20	0.32	0.20	1.25	1.00
Multi. Interp.	1.65	0.60	1.26	0.54	1.31	1.11
Camera Pipe.	1.90	0.36	1.04	0.30	1.83	1.23
Pyramid Blend	5.80	2.90	2.90	1.30	2.00	2.23
Geomean					1.65	1.33

Table 4: Execution times (in ms) of benchmarks and speedup of PolyMage-GPU over Halide's manually written schedules on GTX 1080Ti and Tesla V100.

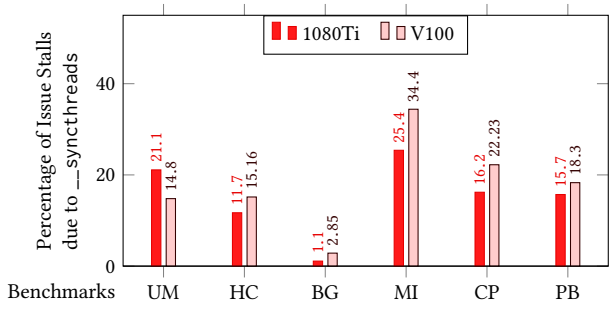

Figure 8: Percentage of instruction issue stalls due to thread block synchronization in OTPTB (Halide) for both GTX 1080Ti and Tesla V100. OTPW execution model does not produce any synchronization based issue stalls.

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 leave-one-out cross validation, since, there are small number of benchmarks. Table 3 shows the weights.

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 [14]. to search for (i) thread block sizes (as a multiple of `WarpSize`), and (ii) tile sizes from 1 to 32 in each dimension. The *Fusion* column in Table 2 shows the time taken, which ranges from less than a second to up to 30 seconds for benchmarks with a few dozen stages. In contrast, the PolyMage autotuner can take up to 20 hours (Section 7.2.2). Thus, our approach to automatic fusion is significantly faster.

Benchmark	Decrease in Global Loads (%)		Increase in Occupancy (%)		Reasons	
	1080Ti	V100	1080Ti	V100	1080Ti	V100
Unsharp Mask	2.51	3.10	0.00	0.00	↓L	↓L
Harris Corner	20.0	31.2	9.10	0.00	↓L+↑O	↓L
Bilateral Grid	4.50	3.60	0.00	0.00	↓L	↓L
Multiscale Interp.	5.30	13.20	0.00	10.0	↓L	↓+↑O
Camera Pipeline	5.21	0.00	1.70	16.6	↓L+↑O	↑O
Pyramid Blend	9.12	7.40	-5.40	13.8	↓L	↓L+↑O

Table 5: Decrease in the number of global memory loads (in %) and increase in achieved occupancy (in %) of code generated using OTPW and Hybrid Tiling over code generated using OTPW on GTX 1080Ti and Tesla V100. Last columns lists the reasons for the increase in performance on both GPUs. ↓L represents decrease in number of global memory loads and ↑O represents increase in the achieved occupancy.

7.2 Performance Evaluation

We now evaluate the performance of OTPW with hybrid tiling and the loop fusion algorithm, which we implement in a tool that we call **PolyMage-GPU**.[1]. We compare our work to the manually-written schedules present in the Halide repository [2]. However, we wrote the schedule for *Pyramid Blend* ourselves, since it was not available.

Table 4 shows the absolute execution times of PolyMage-GPU and Halide and the speedup of PolyMage-GPU over Halide on both GPUs. On every benchmark, PolyMage-GPU is at least as fast as Halide, and in many cases, significantly faster. PolyMage-GPU is faster than manually written schedules in Halide with a geomean speedup of 1.65× and 1.33× on the GTX 1080Ti and Tesla V100 respectively. In general, PolyMage-GPU outperforms Halide because its fusion algorithm chooses better thread block and tile sizes, and the runtime technique has lower synchronization cost, decreased shared memory usage, and improved occupancy. The only exception is the *Bilateral Grid* benchmark on V100, where Halide's manual schedules are competitive with PolyMage-GPU because Halide can fuse the histogram stage, which performs a reduction, with subsequent blurring stages [27], whereas PolyMage-GPU cannot.

7.2.1 Performance Analysis. To study why the OTPW model outperforms the OTPTB, we first investigate instruction stalls due to thread block synchronization. Figure 8 shows that on most benchmarks, a significant fraction of GPU instructions stall due to thread block synchronization. These stalls lead to idle resources which slows down the computation. In contrast, the OTPW model does not employ thread block synchronization at all.

[1]The generated CUDA 10.0 is compiled using `nvcc -O3 -arch=compute_61 -code=sm_61` on the GTX 1080Ti and `nvcc -O3 -arch=compute_70 -code=sm_70` on the Tesla V100.

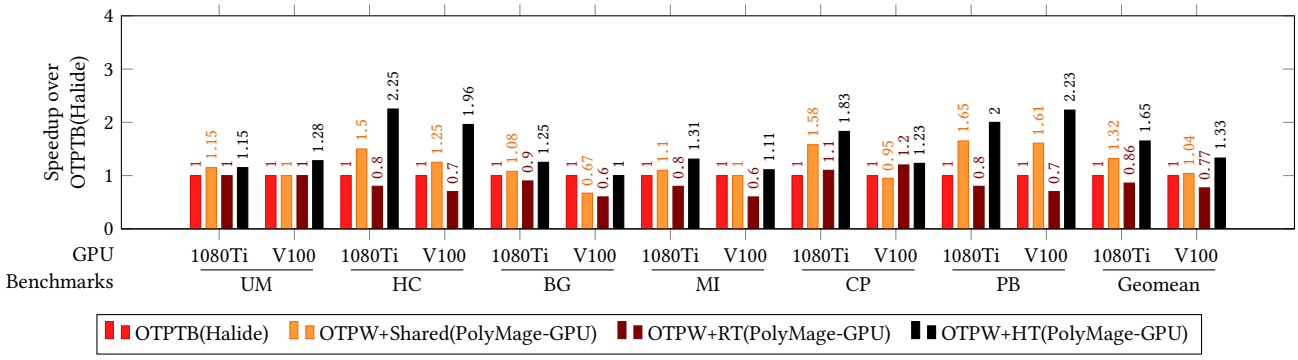

Figure 9: Times relative to OTPTB(Halide) on GTX 1080Ti and Tesla V100. OTPTB(Halide) are the manually written schedules in Halide following *OTPTB* execution model. OTPW+Shared(PolyMage-GPU) is the implementation of *OTPW* execution model in PolyMage-GPU with tiles stored only in shared memory. OTPW+RT(PolyMage-GPU) is the implementation of *OTPW* with tiles stored only in registers in PolyMage-GPU. OTPW+HT(PolyMage-GPU) is the implementation of *OTPW* with Hybrid Tiling in PolyMage-GPU.

Next, we investigate the impact of hybrid tiling. To do so, we modify PolyMage-GPU to disable hybrid tiling: it still uses the OTPW model, but store tiles either entirely in shared memory (**OTPW+Shared**) or entirely in registers (**OTPW+RT**). Figure 9 compares the performance of hybrid tiling (**OTPW+HT**), with the two aforementioned approaches, using thread block tiling (**OTPTB**) as the baseline. On the GTX 1080Ti, *OTPW+Shared* provides a geomean speedup of 1.32× over *OTPTB*: it has no instruction issue stalls, and better grouping with thread block sizes and tile sizes. On the Tesla V100, all benchmarks perform at least as well as *OTPTB* (geomean speedup of 1.04×), with the exception of *Bilateral Grid*. On Bilateral Grid, Halide's manual schedule fuses the reduction stage with the next blurring stage, but the PolyMage compiler cannot. On the V100, *OTPW+Shared* gives the same performance as Halide for *Camera Pipeline* because the manually written schedule performs significant inlining, which the PolyMage compiler cannot do.

Overall, *OTPW+HT* improves the performance of *OTPW+Shared*, with geomean speedups of 1.25× (GTX 1080Ti) and 1.28× (Tesla V100). To investigate further, Table 5 reports how Hybrid Tiling decreases the number of global memory loads, and increases achieved occupancy in contrast to *OTPW+Shared*.

On both GPUs, Hybrid Tiling improves the performance of *Unsharp Mask* and *Harris Corner* by decreasing the number of global memory reads, since hybrid tiling allows larger tile sizes, thereby decreasing the number of overlapping computations. Moreover, Hybrid Tiling increases the occupancy in *Harris Corner* by decreasing shared memory usage. For example, on the GTX 1080Ti, *OTPW+Shared* limits the tile size in *Harris Corner* to 4×1. However, Hybrid Tiling allows 10×1 tiles, with equally divided among shared memory and registers. On both GPUs, the performance of *Bilateral Grid* also improves due to increased tile sizes, and thus fewer overlapping computations, and fewer global memory loads. For the other three benchmarks, the performance improvement is either due to increase in tile sizes, improved occupancy, or both. On the Tesla V100 *Multiscale Interp.* performs better due to a decrease in global memory loads, and an increase in achieved occupancy. The best performing tile sizes of *Multiscale Interp.* decreases the number of overlapping computations but requires more shared memory than the per thread block shared memory limit of Tesla V100, which is decreased to half with Hybrid Tiling. Similarly, on the GTX 1080Ti, opportunity for larger tile size in *Multiscale Interp.* due to Hybrid Tiling decreased the number of overlapping computations. On GTX 1080Ti, Hybrid Tiling decreases the global memory loads and slightly increases the occupancy in *Camera Pipeline*. On Tesla V100, tile sizes of *Pyramid Blend* were increased in Hybrid Tiling due to extra storage for registers available, hence, leading to low overlapping computation, thereby, less global memory loads and increased occupancy. In summary, Hybrid Tiling provides performance improvements due to two major reasons: 1) the extra storage afforded by registers allows larger tiles, which decreases the number of overlapping computations, which in turn, decreases the number of global memory loads, and 2) storing portions of tiles registers decreases the allocated shared memory, hence increases the theoretical and achieved occupancy.

Finally, we note that *OTPW+HT* and *OTPW+Shared* are both faster *OTPW+RT*. Register-only tiles forces PolyMage-GPU to use tiny tiles, which results in a lot of redundant computations.

7.2.2 *Comparison with other techniques.*

Rawat et al.'s Code Generator. We compare PolyMage-GPU to the code generator of Rawat et al. [24]. PolyMage-GPU provides a geomean speedup of 1.6× and 1.7× on the GTX 1080Ti and Tesla V100 respectively. Rawat et al.'s technique has three major drawbacks. First, in their execution model each thread processes exactly one point, whereas PolyMage-GPU does not have this limitation. Thus PolyMage-GPU supports larger tile sizes, and is able to use Hybrid Tiling. Second, since their sliding window technique streams overlapped tiles in one dimension, there is no parallelism in that dimension, thereby leading to significant decrease in total parallelism. Finally, unlike PolyMage-GPU, their cost function is geared towards minimizing the data movement with optimizing shared memory and register usage, hence, does not consider thr number of global memory transactions and achieved occupancy. Hence, our approach decreases the amount of overlapping computations without decreasing in parallelism.

Halide's Gradient GPU autoscheduler. We compare PolyMage-GPU wih Halide's Gradient GPU autoscheduler [17]. To use Halide's latest code generation features, we used the schedules generated by the autoscheduler in the latest Halide version. We found that PolyMage-GPU provides a geomean speedup of 2.42× and 2.35× on the GTX 1080Ti and Tesla V100 respectively. We believe this difference occurs because the thread block sizes picked by PolyMage-GPU are better suited for both GPUs than the hard coded thread block sizes used by Halide's Gradient GPU autoscheduler.

PolyMage's Autotuner. We also compare to PolyMage's image processing autotuner [19]. We added support for OTPW and Hybrid Tiling in the autotuner. The model based autotuner takes tile sizes, thread block sizes, and an overlap threshold. To reduce the search space, PolyMage assigns same tile sizes to all groups and using a greedy approach selects stages to fuse. The greedy approach groups all stages till the fraction of overlap is within a given threshold. Similar to [19], we use same overlap threshold values: 0.2, 0.4, and 0.5. We use tile sizes from 1 to 32 in each dimension, and thread block size of 1 to 512 in each dimension. PolyMage-GPU is 4.5× and 3.3× faster than PolyMage-A on GTX 1080Ti and Tesla V100. PolyMage-A runs till 20 hours to generate these schedules, while PolyMage-GPU runs in seconds. Since, PolyMage-A decreases the search space by selecting the same tile size and thread block sizes for all groups, all schedules are not explored. Hence, PolyMage-A does not find the same schedules as PolyMage-GPU.

8 RELATED WORK

State-of-the-art DSLs for image processing programs all employ loop fusion and overlapped tiling to increase locality between stages [19, 22, 23]. Halide and Forma use GPUs and execute one overlapped tile per thread block. Halide's original CPU autoscheduler [18] uses a greedy algorithm, whereas Dynamic Programming Fusion [14] efficiently enumerates all possible fusion choices for a CPU. Halide has a newer autoscheduler [6] that uses beam search with a learned cost model for CPUs. Halide's Gradient GPU autoscheduler [17] is a GPU autoscheduler for Halide that performs greedy function inlining and loop fusion with hard-coded thread block sizes for each tile. In Section 7.2.2, we compare our work to some of these autoscheduler.

Versapipe [34] exploits both task and data parallelism on GPUs by assigning tasks to persistent threads based on their SM ID. HiWayLib [35] presents a way to efficiently run pipelined computations that require significant communication between CPUs and GPUs. In contrast, our work focuses on improving the performance of image processing pipelines, which are data parallel applications, and we require all data to fit on the GPU. We employ a warp-centric approach and use a cost function to select stages for fusion. The aforementioned approaches would complement our work.

Several techniques support the parallel execution of stencil computations on GPUs, using the *Overlap tile per thread block (OTPTB)* model [12, 24–26, 33]. Rawat et al. [24] use a sliding window on one spatial dimension and overlap tiling on the others to eliminate some redundant computations in Overtile [12]. *Hybrid hexagonal classic tiling* [11] also executes one tile per thread block. Flextended Tiles [33] uses rectangle trapezoid tiling to obtain tighter overlapped tile bounds. Artemis [25] is a DSL that allows an expert to guide challenging code optimizations using bottleneck analysis and tunable code parameters. Artemis and Flextended tiles are complementary to our work. These approach supports expression inlining, which pass the value of producer to consumer through a register within the same thread. However, none of these employ the *overlapped tile per warp (OTPW)* model and *hybrid tiling*, which stores portions of tiles in registers that is shared among threads of a warp.

In 2009, Hong and Kim [13] presented a general analytical model to predict the performance of GPU kernels. However, recent advances in GPU architectures, including changes to their memory hierarchy, have made their model out of date. Prajapati et al. [21] present an analytical model for predicting the runtime of stencil computations on GPUs (tiled using [11]). That model considers shared memory usage, theoretical occupancy, and warp switching. However, it omits several key factors, including register usage, the number of global memory transactions, achieved occupancy, and thread block sizes, which our model considers.

Halide exposes warp shuffle instructions, which makes it possible to store portions of a tile in registers [3]. However, Halide restricts the size of the innermost dimension to be less than warp size, and cannot store tiles in both registers and shared memory. Other systems employ in-register storage and warp shuffles to improve the performance of GPU kernels [7, 9, 10, 16, 20, 29]. Our work allows multiple warps per thread block, allows the innermost dimension to have an arbitrary size, and is a hybrid technique that stores tiles in both registers and shared memory. To the best of our knowledge, this combination has not been presented in prior work.

9 CONCLUSION

This paper presents 1) an execution model for image processing pipelines on GPUs that executes one overlapped tile per warp, 2) *hybrid tiling*, which allows portions of overlapped tiles to be stored in either registers or shared memory, and 3) an automatic loop fusion technique for GPUs that considers several key factors that affect the performance of GPU kernels. These techniques use low cost synchronization, improves occupancy, and allows larger tiles that require fewer overlapping computations. We implement these techniques in PolyMage-GPU, which is a new GPU backend for the PolyMage DSL. Using several benchmarks, we show that our work achieves significant speedups over manually-written schedules.

ACKNOWLEDGEMENTS

This work was partially supported by the National Science Foundation under grant CCF-1717636.

REFERENCES

[1] [n. d.]. CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/
[2] [n. d.]. Halide. https://github.com/halide/Halide/ commit 52da814a2c3c4af78125757385a8a86efdde3234.
[3] [n. d.]. Halide. https://github.com/halide/Halide/ commit 59bca3c8e535f7f99c90efd1d932db934f9c01b6.
[4] [n. d.]. Using CUDA Warp-Level Primitives. https://devblogs.nvidia.com/using-cuda-warp-level-primitives/.
[5] [n. d.]. Warp Shuffle Functions in AMD HIP. https://github.com/ROCm-Developer-Tools/HIP/blob/master/docs/markdown/hip$_k$ernel$_l$anguage.md#warp-shuffle-functions

[6] 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.* (2019).

[7] 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)*.

[8] 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)*.

[9] 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)*.

[10] Simon Garcia De Gonzalo, Sitao Huang, Juan Gómez-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)*.

[11] 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)*.

[12] 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)*.

[13] 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)*.

[14] 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)*.

[15] Abhinav Jangda and Arjun Guha. 2020. Model-Based Warp Overlapped Level Tiling for Image Processing Programs on GPUs. arXiv:cs.PL/1909.07190

[16] 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)*.

[17] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. 2018. Differentiable Programming for Image Processing and Deep Learning in Halide. *ACM Trans. Graph.* (2018).

[18] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically Scheduling Halide Image Processing Pipelines. *ACM Trans. Graph.* (2016).

[19] 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 (ASPLOS '15)*.

[20] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, 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 (ASPLOS '19)*.

[21] Nirmal Prajapati, Waruna Ranasinghe, Sanjay Rajopadhye, Rumen Andonov, Hristo Djidjev, and Tobias Grosser. 2017. Simple, 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)*.

[22] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*.

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

[24] Prashant Singh Rawat, Changwan Hong, Mahesh Ravishankar, Vinod Grover, Louis-Noel 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)*.

[25] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, A. Rountev, L. Pouchet, and P. Sadayappan. 2019. On Optimizing Complex Stencils on GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[26] Prashant Singh Rawat, Miheer Vaidya, Aravind Sukumaran-Rajam, Atanas Rountev, Louis-No"el Pouchet, and P Sadayappan. 2019. On Optimizing Complex Stencils on GPUs. (2019).

[27] Patricia Suriana, Andrew Adams, and Shoaib Kamil. 2017. Parallel Associative Reductions in Halide. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*.

[28] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4 (Jan. 2013).

[29] 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)*.

[30] 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)*.

[31] M. Wolfe. 1989. More Iteration Space Tiling. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing (Supercomputing '89)*.

[32] Michael Wolfe. 1994. The Definition of Dependence Distance. *ACM Trans. Program. Lang. Syst.* (1994).

[33] Jie Zhao and Albert Cohen. 2019. Flextended Tiles: A Flexible Extension of Overlapped Tiles for Polyhedral Compilation. *ACM Trans. Archit. Code Optim.* (2019).

[34] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. 2017. Versapipe: A Versatile Programming Framework for Pipelined Computing on GPU. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*.

[35] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. 2019. HiWayLib: A Software Framework for Enabling High Performance Communications for Heterogeneous Pipeline Computations. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*.