Fusing Convolution Kernels through Tiling

Mahesh Ravishankar Paulius Micikevicius Vinod Grover
NVIDIA Corporation, U.S.A
mravishankar,pauliusm,vgrover@nvidia.com

Abstract

Image processing pipelines are continuously being developed to deduce more information about objects captured in images. To facilitate the development of such pipelines several Domain Specific Languages (DSLs) have been proposed that provide constructs for easy specification of such computations. It is then upto the DSL compiler to generate code to efficiently execute the pipeline on multiple hardware architectures. While such compilers are getting ever more sophisticated, to achieve large scale adoption these DSLs have to beat, or at least match, the performance that can be achieved by a skilled programmer.

Many of these pipelines use a sequence of convolution kernels that are memory bandwidth bound. One way to address this bottleneck is through use of tiling. In this paper we describe an approach to tiling within the context of a DSL called Forma. Using the high-level specification of the pipeline in this DSL, we describe a code generation algorithm that fuses multiple stages of the pipeline through the use of tiling to reduce the memory bandwidth requirements on both GPU and CPU. Using this technique improves the performance of pipelines like Canny Edge Detection by 58% on NVIDIA GPUs, and of the Harris Corner Detection pipeline by 71% on CPUs.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Optimization

Keywords Fusion, Tiling, Domain Specific Languages, Convolutions, GPU

1. Introduction

Recently many Domain Specific Languages (DSLs) have been developed that allow application developers to easily specify image processing pipelines [3, 9, 10, 12, 13]. These DSLs provide constructs that not only allow easy specification of common operations in image processing, like stencils and convolutions, but can also generate code to target multiple architectures, like multi-core CPUs and GPUs, from the same specification. It also allows the DSL compiler to capture the producer-consumer relationship between different stages of the pipeline and perform optimizations, like fusion of multiple stages, that are outside the scope of traditional compilers like GCC and NVCC.

ARRAY'15, June 13, 2015, Portland, OR, USA
© 2015 ACM. 978-1-4503-3584-3/15/06...$15.00
http://dx.doi.org/10.1145/2774959.2774965

```
1  stencil blurx(vector#2 float X){
2     return (X@[-1,0] + X + X@[1,0])/3.0;
3  }
4  stencil blury(vector#2 float Y){
5     return (Y@[0,-1] + Y + Y@[0,1])/3.0;
6  }
7  parameter M,N;
8  vector#2 float input[M,N];
9  temp = blurx(input);
10 output = blury(temp);
11 return output;
```

Listing 1. Blur in Forma

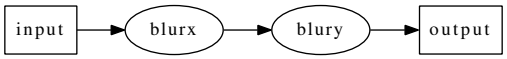
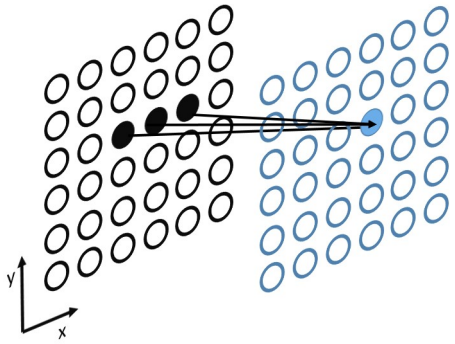


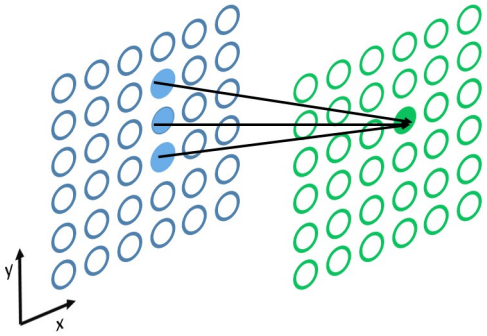
Figure 1. DAG of producer-consumer relationship of Forma program in Listing 1

While DSLs are capable of generating good quality code, there is still much scope for improvement. Typically stencil computations are bandwidth bound. To get around this issue application developers have used strategies like tiling that utilize hardware features like caches on CPUs and shared memory buffers on GPUs to reduce the bandwidth requirement of computations. Some DSLs have adopted similar techniques to improve the performance of the generated code. PolyMage [10] uses sophisticated polyhedral analysis to increase reuse across pipeline stages on CPUs. Halide [12] allows users to specify a schedule that implements tiling and uses auto-tuning to determine the tile sizes. In this paper, we explore the use of tiling to improve performance of a sequence of convolution kernels within Forma [13]. This DSL can generate CUDA code to target NVIDIA GPUs, as well as C code with OpenMP pragmas to target multi-core CPUs. While our main goal is to improve performance on GPUs, the developed approach shows a considerable benefit on CPUs as well.

Listing 1 shows the specification of a simple image processing pipeline in Forma. A stencil computation is specified as a function prefixed with the keyword `stencil` that is to be applied over the image passed as argument to the function. For example, the function `blurx` specifies a stencil that operates on a 2D image or `vector#2`. Applying this function to an image `input` at Line 9, performs the stencil computation on `input` to generate the image `temp`. The data type used for storing the image, as well as its size is computed automatically by the compiler based on the specification of the stencil. For example, the image `temp` is deduced to be a 2D image of `float`s with its size being same as the size of `input`. The body of a stencil function specifies the computation that gives the value at a point (i, j) of the result image. Use of the `@` operator



(a) blurx : `input` → `temp`



(b) blury : `temp` → `output`

Figure 2. Naive execution of Listing 1

allows access to neighboring points of the image passed as argument to the function. For example, `X@[-1,0]` refers to the value at a point $(i-1, j)$ of `X` while computing the value at point (i, j) of the result. References to the argument without use of the `@` operator are analogous to the expression `X@[0,0]`. The Forma compiler can also capture the producer-consumer relationship between the different stages of the image processing pipeline in form of a Directed Acyclic Graph (DAG). Figure 1 shows the different stages involved in computing the image `output` from the image `input` using the pipeline specified in Listing 1.

A naive execution of the computation shown in Listing 1 on the GPU would be to have two separate kernels, one to perform the computation corresponding to `blurx`, and another to perform the computation corresponding to `blury`. Within each of these kernels, a thread on the GPU would be responsible to compute the value of a single point of the result image. For the first kernel (Figure 2a), each thread would read 3 elements of the image `input` from global memory and perform 1 global memory write of the result at a point in image `temp`. The same would happen for the second kernel (Figure 2b). Therefore, computing the value at a single point of the final image (`output`) would require 6 global memory loads and 2 global memory writes. This results in significant pressure on the bandwidth to global memory. While hardware mechanisms like vectorized loads of coalesced accesses and caching accesses within a thread block ease some of this pressure, they are less effective for larger stencils. For example a simple 5×5 Gaussian blur kernel issues 25 loads per thread and 1 global memory store. The same is true for a naive execution on the CPU. The computation for each of the stencil is performed within loop nests that iterate over the output image and compute the value of each pixel. The value at points of the intermediate image `temp` that are along the same column (along y-axis) are reused while computing the value at points along the same column of the `output` image. When image sizes are large, these values are evicted from cache before they can be reused resulting in a performance degradation.

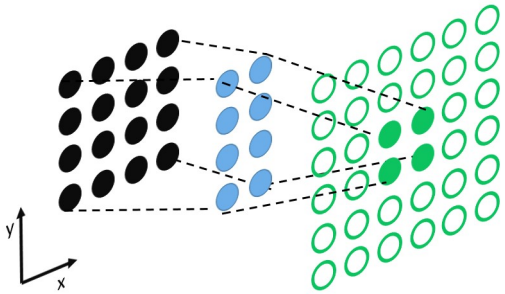


Figure 3. Tile of `input` loaded into shared memory → Tile of `temp` computed in shared memory → Tile of output computed

In this paper we use a combination of tiling and fusion to reduce the number of global memory accesses required to compute the final image after applying a sequence of convolution kernels, like the ones shown in Listing 1. Unlike optimizing compilers like PluTo [1] which operate on C codes and are constrained by the manner in which intermediate results of the pipeline are computed/stored by the input code, the Forma compiler only needs to generate the output image in memory. It is free to compute/store intermediates in a manner that would help improve the performance of the generated code. While a domain expert can manually implement such schemes, doing so is extremely tedious and error-prone, especially while handling boundary conditions.

The rest of the paper is organized as follows. Section 2 describes the code generation algorithm that implements our code generation scheme that can be refered to as *fusion through tiling*, for both CPUs and GPUs. Section 3 evaluates the performance of the generated code on three common image processing pipelines. Section 4 describes other related work on stencil optimizations with Section 5 summarizing the contributions of this paper.

2. Evaluating the DAG in a Tiled Fashion

In this section we describe the tiled execution of the code generated by Forma on GPUs and the algorithm to generate that code. We then describe how a similar approach with a some modifications is used to generate tiled code on the CPU as well.

2.1 Tiling and Fusion on the GPU

To reduce the bandwidth requirement, a combination of tiling and fusion is used. Consider the GPU execution schematic shown in Figure 3. The output image is computed using the stencil function `blury`. On the GPU, each tile is evaluated by a thread-block such that each thread computes the value at a single point. For example in Figure 3, a tile of size 2×2 of the image `output` is computed by a single thread block. From the DAG of the computation (Figure 1), the compiler knows that the argument to the function `blury` is itself computed using the stencil function `blurx`. So the compiler generates code to compute in shared memory, the tile of the result of the stage `blurx` needed to compute a tile of the result of stage `blury`. Since the stencil `blury` access neighboring points at a distance 1 and -1 along the y-direction, the size of this intermediate tile has to be increased by 1 along the positive and negative y-directions.

To compute the values of this intermediate tile, the function `blurx` is applied to a region of the image `input`. Since this image is already in memory, the compiler generates code to read the required region of this image into shared memory. The stencil `blurx` accesses neighboring points at a distance 1 and -1 along

Algorithm 1: ComputeTile(G,v,$\vec{T}$)

```
Input : G : DAG of the computation
        v : Node in the DAG being analysed
        T⃗ : The tile size along each dimension used to compute the
result of v
1  begin
2    Consumed = φ ;
3    foreach p ∈ v.Predecessor do
4      Ē = MaxPositive(v,p) - MaxNegative(v,p) ;
5      T⃗_p = T⃗ + Ē ;
6      if p.IsStencil then
7        p_val = ComputeTile(G,p,T⃗_p) ;
8        Consumed = Consumed ∪ p_val ;
9      else
10       ComputeInMemoryIfUnavailable(G,p) ;
11       p_val = LoadTileToShared(p,T⃗_p) ;
12       Consumed = Consumed ∪ p_val ;
13   R = ConsumeFromSharedMem(v,T⃗,Consumed) ;
```

Algorithm 2: ComputeInMemoryIfUnavailable(G,v)

```
Input : G : DAG of the computation
        v : Node in the DAG being evaluated
1  begin
2    if ¬ v.IsComputed then
3      if v.IsStencil then
4        T⃗ = DEFAULT_TILE_SIZE ;
5        R = ComputeTile(G,v,T⃗) ;
6        I = ComputeTileIndex(T⃗,v.Size) ;
7        StoreTileToGlobalMemory(R,I) ;
8      else
9        foreach p ∈ v.Predecessor do
10         ComputeInMemoryIfUnavailable(G,p) ;
11       ComsumeFromGlobalMem(G,v) ;
12     v.IsComputed = true ;
```

the x-direction of the input image. Consequently, the size of the region needed is the size of the intermediate tile increased by 1 along the positive and negative x-direction.

The entire computation is performed within a single kernel with most threads reading one element from global memory and storing one value to the global memory. Therefore the sequence of convolution kernels have been fused into a single kernel. However, it should be noted that neighboring blocks of the kernel execution end up computing values along the extended regions of intermediate tiles in redundant fashion. This approach is similar to the overlapped-tiling approach to tiling developed in [7] for iterative stencil applications. There the authors showed that on GPUs, the benefit of reducing global memory accesses outweighs the cost of performing redundant computations. PolyMage [10] uses a similar approach of tiling on CPUs.

Algorithm 2 describes the procedure to generate the tiled code on GPU. To generate the code for computing the result of the pipeline the compiler invokes the function *ComputeInMemoryIfUnavailable* on the node in the producer-consumer DAG that computes the result image. If this stage is a stencil [1], the compiler starts with a default tile size (the size of the thread block on the GPU), and invokes the tiling code generation function *ComputeTile*. This function is described in Algorithm 1. For each predecessor in the producer-consumer DAG the size of the intermediate tile used to store the output of the predecessor node is computed at Line 4. This size is the current tile size increased by the maximum positive and negative offsets used to accesses the result produced by the predecessor node within the stencil body of the current node. If the predecessor is a stencil operation itself, a recursive call is made to generate the code that computes the intermediate tile in shared memory. If not, the entire image corresponding to the output of the predecessor is evaluated in global memory by calling the function *ComputeInMemoryIfUnavailable* described in Algorithm 2, followed by the required tile loaded into shared memory. Since all the values needed to compute a tile of the output is now in shared memory, the compiler generates code to use these values and generate the output tile in shared memory as well.

On returning from the call to *ComputeTile* at Line 7, the compiler generates code to compute the index of the tile evaluated (*ComputeTileIndex*) and to store the computed tile to global memory (*StoreTileToGlobalMemory*). Note that for this final step, instead of computing the result tile in shared memory and later writing it to global memory, it is more efficient to write the values directly to global memory. The description of this has been left out for the sake of clarity. Finally, if the function *ComputeInMemoryIfUnavailable* is invoked on a stage that is not a stencil, the code generator falls back to the default mode of first computing the result of all the predecessor nodes in global memory and consuming them to produce the result of this stage.

2.1.1 Adjusting Shared Memory Usage for Valid Kernel Execution

The amount of shared memory required to execute the fused kernel is a function of the block size used for the kernel launch. For example, say the block size used for the fused version of the kernel for the computation in Listing 1 is `bx, by`. The size of the shared memory used would be `(bx+2)*(by+2)` for the input image, and `bx*(by+2)` for the intermediate image. The compiler generates code to adjust the block size so that the launched kernel does not use more shared memory than what is available on the device. It does so by iteratively halving the block size along each dimension till the constraints of shared memory usage are met. Since shared memory usage always decreases on decreasing the block size, this approach is guaranteed to find a configuration that is valid for the given device. While this might affect device occupancy during kernel execution, for the experiments described in Section 3 this was not an issue.

2.1.2 Handling Boundary Conditions in Presence of Tiling

The syntax of Forma allows the developers to easily specify boundary conditions that are to be used while applying a stencil. For the example in Listing 1, mirror boundary conditions can be applied by changing the stencil application at Line 9 to `blurx(input:mirror)` and at Line 10 to `blury(temp:mirror)` [2]. If tiling has to be used in such situations, while computing along the edges of the output image, the values in the intermediate tiles have to computed while respecting these boundary conditions. Our code-generator automatically generates code to handle such boundary conditions appropriately. The tedious details of the implementation are skipped here, but such a feature demonstrates the advantage of using a DSL like Forma. Manually implementing tiling algorithms like the one described in this section while handling

[1] Forma provides other operations like *compose* and *interpolation /extrapolation* that are not targeted by the present approach

[2] Other boundary conditions like *clamped* and *constant* can also be specified [13]

boundary conditions appropriately requires considerable effort on part of application developers. The Forma compiler can automatically handle boundaries even while generating tiled code.

2.2 Reducing Branching Overhead on GPUs

The code generation algorithms described in Section 2.1 required that intermediate tiles computed in shared memory to be of size bigger than the size of the output tile being computed to account for the accesses to neighboring points in the body of the stencil function. Since the size of the thread block used is same as the size of the output tile, some threads have the additional work of computing (or initializing from global memory) the value of points along the extended regions of the intermediate tiles. This serializes the computation of some parts of the intermediate tiles. One way to address this issue is to increase the size of the thread block to be as big as the size of the biggest intermediate tiles. So each thread would either have to compute the value at either one or no points. As the kernel computes all the intermediate tiles and the final output tile, threads become inactive but none of the computation is serialized. This approach increases the register usage per kernel and could potentially hurt the achieved occupancy on the GPU, but in practice this was not found to be a limiting factor. An additional tweak that helped improve the performance was to ensure that thread with higher index values become inactive as the computation proceeds so that all inactive threads are collected within a single warp.

2.3 Code Generation on the CPU

The code generation algorithm described in Section 2.1 can also be used on the CPU with some modifications. Without tiling, the Forma code generator for CPUs generates loop nests that iterate over the size of the result of each stage (`output`) and computes the value of each point. For large image sizes there would be no reuse across the multiple uses of values of the input image. Instead, using a smaller buffer which stores only those values of an intermediate image needed to compute a tile of the output would reduce the cost of multiple accesses to these values when the buffer fits in some level of the hardware cache.

To use this approach, a tiled loop nest that iterates over the size of the result image is generated. The outer loops corresponds to those that iterate over the tiles of the result, while the inner loops iterate over points within a tile. At a point before these inner loops, the compiler generates code that computes the value of intermediate tiles used for computing the final output. The size of the result tile computed is chosen such that all the intermediate buffers fit in L2 cache simultaneously. Padding is used to avoid conflict misses. Unlike the GPU code generation scheme, there is no need to explicitly move elements of images that have already been fully computed into a smaller buffer. The caching mechanism on the hardware provides this functionality automatically. The tile sizes for the loop have to be adjusted so that there is enough space for caching elements of these images as well.

3. Experimental Evaluation

In this section we evaluate the improvement in performance obtained by fusion through tiling algorithm described in Section 2. For evaluation we used three benchmarks, one is the simple blur computation described in Listing 1. The other two are the Canny-edge detection pipeline [2] and the Harris Corner Detection [5] algorithm.

3.1 GPU Performance

The Forma compiler targets NVIDIA GPUs by generating CUDA code. The code generator was modified to incorporate Algorithms 1 and 2. NVCC-6.5 was used for compiling the generated code. Since the amount of shared memory used by a block has to be less than 48KB, the backend also generates a function to calculate the amount shared memory used by a thread block for a particular block size. Using this, the generated code adjusts the block size to achieve maximum occupancy.

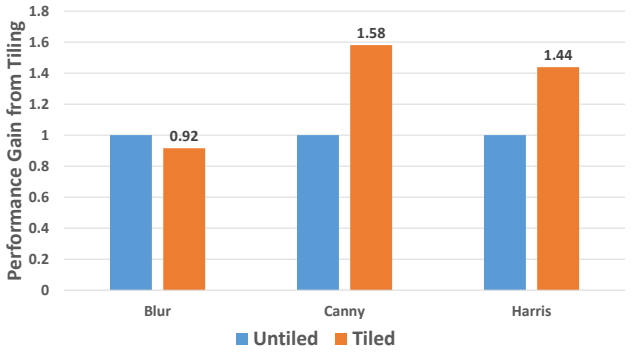


Figure 4. Performance Improvement from Fusion through Tiling on GPU (Tesla K20c)

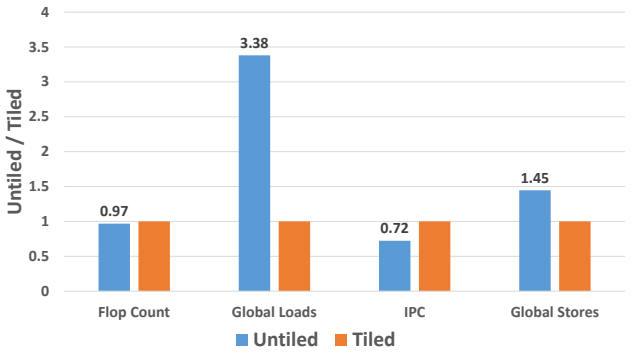


Figure 5. GPU Performance Metrics for Blur

The benefit of tiling was evaluated on an NVIDIA Tesla K20c. Figure 4 shows the improvement in the total kernel execution times observed for the three benchmarks. While there is a slight loss of performance with Blur, with Canny and Harris there is a significant improvement. Figures 5, 6 and 7 provide an explanation for the performance observed. Tiling for Canny (Figure 7) leads to a significant drop in the number of global memory loads. The relatively low Instructions Per Cycle (IPC) for the untiled code compared to the tiled code shows that this was indeed a bottleneck. For Harris kernel (Figure 6), the number of global memory loads is reduced by only a factor of 4.35, but the IPC of the tiled code is almost twice that of the untiled code suggesting that the loads from global memory were still a bottleneck for the kernel execution. With Blur however (Figure 5), even though the number of global memory loads reduced by a factor of 3.38, the IPC of the untiled code was already pretty high which suggests that for this kernel global memory loads were not a significant bottleneck. Since the tiling approach used here increases the number of floating point operations, the performance of the Blur kernel dropped by a small amount.

3.2 CPU Performance

The performance on CPU was measured on a quad-core Intel Core i7-4820K. Figure 8 shows the speed up obtained. For Blur and Harri, an improvement of 79% and 71% respectively was observed.

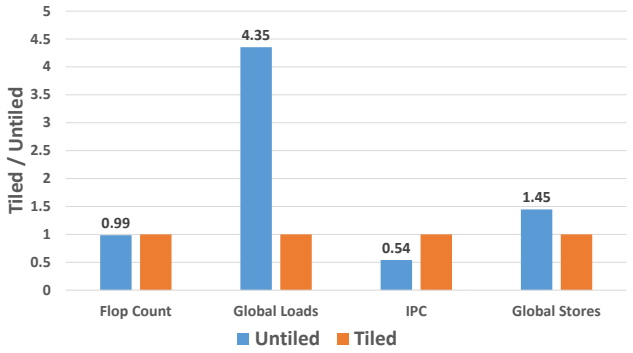


Figure 6. GPU Performance Metrics for Harris

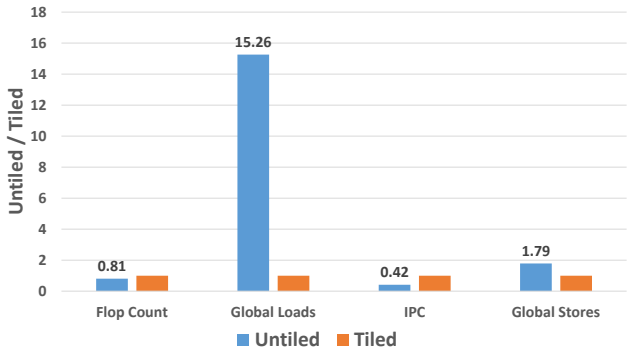


Figure 7. GPU Performance Metrics for Canny

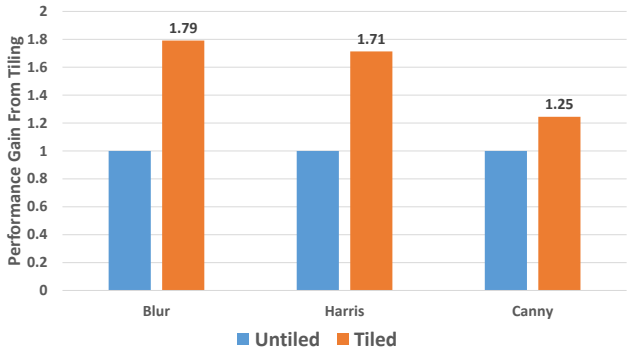


Figure 8. Performance Improvement from Fusion through Tiling on CPU (i7-4820K)

For Canny, the improvement was 25%. The reason for the higher speed up obtained on CPU is that bandwidth to main memory is typically lower than that available on the GPU, whereas bandwidth to caches is pretty high. The Forma C backend was modified to modify the tile sizes based on the cache sizes automatically. As described in Section 2.3, use of padding for the tiles helps reduce conflict misses resulting in better cache utilization.

4. Related Work

The approach of using tiling to improve data locality is a classic transformation. PLuTo [1] implements state-of-the art loop transformations to enhance data locality through fusion and tiling. While PLuTo is source-to-source compiler that targets a wide range of applications written in C, PolyMage [10] from the same authors uses similar techniques but is targeted towards image processing pipelines. AlphaZ [14] also groups together operations that have a producer-consumer relationship but operates at the granularity of individual statements, and not at the granularity of images as is done in this paper. Several other compiler techniques [4, 6–8] have been developed to target iterative stencil computations where tiling is done across the time stepping loop in order to improve data locality. These techniques are target low level C code and cannot eliminate the use of large temporary buffers as was done in this paper through fusion.

Previous work have also explored the used of tiling to improve performance on other architectures. KernelGenius [9] uses tiling to improve performance of image processing applications on embedded architectures. Pouchet et.al. [11] have used tiling to improve the performance of stencil applications on FPGAs.

In context of image processing applications, Halide [12] is the most popular DSL. In Halide the programmer splits the computation in two parts, a specification and a schedule. The schedule allows programmers to specify the manner in which the specified computation is executed. This is where a programmer can make use of techniques like tiling and vectorization to improve the performance of the pipeline. The optimal tile size can be deduced through auto-tuning. The approach in this paper is to free the programmer from even having to decide when and how to tile the computation. Instead, the DSL compiler automatically uses tiling to improve the performance of the generated code when it finds an opportunity to do so.

5. Conclusion

In this paper we have developed a code generation algorithm within the context of a DSL for image processing pipelines, that uses the high level description of such pipelines to fuse a sequence of convolution kernels using tiling. On GPUs this reduce the global memory bandwidth requirements by utilizing the fast shared memory. The same technique can be used to improve the performance on CPUs as well by making use of the hardware caching mechanism. On GPUs this lead to an improvement of 58% on the Canny edge detection pipeline and 44% on the Harris Corner pipeline. On CPUs an improvement of 25% and 71%, respectively, was observed for the same pipelines. The approach described is not specific to image processing but can be applied to any domain, like machine learning, where a sequence of convolutions are used.

References

[1] U. Bondhugula, J. Ramanujam, and et al. Pluto: A practical and fully automatic polyhedral program optimization system. In *PLDI*, 2008.

[2] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1986.

[3] C. Chiw, G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer. Diderot: A parallel DSL for image analysis and visualization. In *PLDI*, 2012.

[4] A. Cohen, T. Grosser, P. H. J. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege. Split Tiling for GPUs: Automatic Parallelization Using Trapezoidal Tiles to Reconcile Parallelism and Locality, avoiding Divergence and Load Imbalance. In *GPGPU 6*, Houston, États-Unis, 2013.

[5] C. Harris and M. Stephens. A combined corner and edge detector. In *Fourth Alvey Vision Conference*, 1988.

[6] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector SIMD architectures. In *ICS*, 2013.

[7] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *ICS*, pages 311–320, 2012.

[8] S. Krishnamoorthy, M. M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *PLDI*, pages 235–244, 2007.

[9] T. Lepley, P. Paulin, and E. Flamand. A novel compilation approach for image processing graphs on a many-core platform with explicitly managed memory. In *CASES*, 2013.

[10] R. T. Mullapudi, V. Vasista, and U. Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *ASPLOS*, 2014.

[11] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong. Polyhedral-based data reuse optimization for configurable computing. In *FPGA*, 2013.

[12] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*, 2013.

[13] M. Ravishankar, J. Holewinski, and V. Grover. Forma : A DSL for image processing applications to target gpus and multi-core cpus. In *GPGPU*, 2015.

[14] T. Yuki, V. Basupalli, G. Gupta, G. Iooss, D. Kim, T. Pathan, P. Srinivasa, Y. Zou, and S. Rajopadhye. Alphaz: A system for analysis, transformation, and code generation in the polyhedral equational model. Technical report, Colorado State University, 2012.