Forma: A DSL for Image Processing Applications to Target GPUs and Multi-core CPUs

Mahesh Ravishankar
NVIDIA Corporation
Redmond, Washington
USA
mravishankar@nvidia.com

Justin Holewinski
NVIDIA Corporation
Charlotte, North Carolina
USA
jholewinski@nvidia.com

Vinod Grover
NVIDIA Corporation
Redmond, Washington
USA
vgrover@nvidia.com

ABSTRACT

As architectures evolve, optimization techniques to obtain good performance evolve as well. Using low-level programming languages like C/C++ typically results in architecture-specific optimization techniques getting entangled with the application specification. In such situations, moving from one target architecture to another usually requires a reimplementation of the entire application. Further, several compiler transformations are rendered ineffective due to implementation choices. Domain-Specific Languages (DSL) tackle both these issues by allowing developers to specify the computation at a high level, allowing the compiler to handle many tedious and error-prone tasks, while generating efficient code for multiple target architectures at the same time.

Here we present Forma, a DSL for image processing applications that targets both CPUs and GPUs. The language provides syntax to express several operations like stencils, sampling, etc. which are commonly used in this domain. These can be chained together to specify complex pipelines in a concise manner. The Forma compiler is in charge of tedious tasks like memory management, data transfers from host to device, handling boundary conditions, etc. The high-level description allows the compiler to generate efficient code through use of compile-time analysis and by taking advantage of hardware resources, like texture memory on GPUs. The ease with which complex pipelines can be specified in Forma is demonstrated through several examples. The efficiency of the generated code is evaluated through comparison with a state-of-the-art DSL that targets the same domain, Halide. Our experimental result show that using Forma allows developers to obtain comparable performance on both CPU and GPU with lesser programmer effort. We also show how Forma could be easily integrated with widely used productivity tools like Python and OpenCV. Such an integration would allow users of such tools to develop efficient implementations easily.

GPGPU-8, February 7, 2015, San Francisco, CA, USA
Copyright 2015 ACM 978-1-4503-3407-5/15/02...$15.00
http://dx.doi.org/10.1145/2716282.2716290

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Specialized Application Languages*

Keywords

Image processing, DSL, Stencils, multi-core, GPGPU

	Auto Mem. Manag-ment	**Auto Schedul-ing**	**Whole Prog. Opt.**	**Auto GPU Offload-ing**
NumPy	✔	✘	✘	✘
OpenCV	✔	✘	✘	✘
Halide	✔	✘	✔	✔
Forma	✔	✔	✔	✔

Figure 1: Comparison of Forma with existing frameworks for developing image processing applications

1. INTRODUCTION

Image processing techniques have contributed significantly to the widespread use of technology in our day-to-day lives. From mundane applications like face-recognition in digital cameras, to critical applications like medical imaging to screen for cancer and other neuro-degenerative diseases, image processing techniques form an important part of innovation everywhere. With the advent of smart phones and the increasing resolution of images captured by its cameras, processing these images on a phone using its limited hardware resources quickly and in a power efficient manner is important for a good end-user experience. As a result, significant time and effort has been spent in manual optimizations of these applications. This often leads to the specifics of the computation being intertwined with code transformations necessary to get good performance on a specific hardware. To port such an application to different architectures usually requires a significant change to the code. With the wide range of constantly evolving architectures currently used in computing devices, maintaining all architecture specific optimizations for each application can be quite cumbersome.

Domain-Specific Languages (DSLs) [12, 25, 10, 9] have traditionally been used to separate the architecture specific details from the specification of the computation. They provide domain specific abstractions that allow the programmer to easily specify the computation. Simultaneously, compilers can exploit the semantics of these abstractions to generate optimized code for different hardware. The challenge

```
temp(x,y) = (image(2*x-1,y)
     + 2*image(2*x,y) + image(2*x+1,y)) / 4;
output(x,y) = (temp(x,2*y-1)
     + 2*temp(x,2*y) + temp(x,2*y+1)) / 4;
```

Listing 1: Mathematical description of a sampling filter

```
temp(2*x,y) = (3*image(x,y) + image(x-1,y))/4;
temp(2*x+1,y) = (3*image(x,y) + image(x+1,y))/4;
output(x,2*y) = (3*temp(x,y) + temp(x,y-1))/4;
output(x,2*y+1) = (3*temp(x,y) + temp(x,y+1))/4;
```

Listing 2: Mathematical description of an extrapolation filter

in designing DSLs is to devise a set of abstractions that are expressive enough while still exposing optimization opportunities to the compiler.

In this paper we present Forma, a DSL for image processing applications that is capable of targeting both multi-core CPUs and GPGPUs. Listing 1 provides a mathematical description of a simple image-processing pipeline involving two steps to sample a 2D image. The first step uses a stencil to sample along the x-dimension of the input image, and the second step uses a similar stencil to sample along the y-dimension. The resulting image (`output`) is half the size of the input image along each dimension. An image processing DSL should provide syntax to allow the user to specify such stencils easily, while allowing the user to chain together, different stages of a pipeline.

Listing 2 shows a more involved pipeline which computes an extrapolated image from a given input image. Here, the output image is computed by first extrapolating along the x-dimension followed by an extrapolation along the y-dimension. The difference here is that for every dimension the even and odd points are computed through different stencils.

The implementation methodology that would achieve good performance would depend on the target architecture. For example, on CPU it might be better to apply both the odd and even stencil within a single loop-nest to increase locality. On the GPU, if each thread is operating on one pixel of the output, making this decision for each pixel would result in thread divergence within the kernel. In addition, using low-level programming languages like C++/CUDA would require the developer to manage tedious details such as allocating buffers of appropriate sizes for intermediate images, handling boundary conditions while applying a stencil, copying data to and from the device while targeting GPUs, etc. The syntax of Forma allows application developers to specify image processing pipelines and the stencil to be applied at every stage in a concise manner. Tedious details of memory management, applying boundary conditions and data transfers are left to the compiler to manage. The syntax also allows the compiler to optimize the generated code without being constrained by the implementation details.

This paper highlights the ease with which complex applications can be easily specified using the syntax provided by Forma, while achieving good performance on both CPUs and GPUs. The rest of the paper describes the syntax of Forma and demonstrates the ease with which complex image processing pipelines can be expressed in this language. We evaluate the developed DSL in terms of ease of programming and performance with several existing frameworks, including NumPy [20], OpenCV [22], and Halide [24]. Figure 1 summarizes the benefits of using Forma over these frameworks. In particular, NumPy can only accelerate individual array operations, and does not support offloading arbitrary computations to accelerators such as GPUs. Similarly, OpenCV does not optimize across the entire program. In the general use case, OpenCV is just a library and the compiler only sees function calls to OpenCV routines without knowledge of the implementations of these routines. It supports GPU offloading for a limited set of pre-existing operations and not arbitrary code written using the OpenCV library. Halide comes closest to Forma in feature set, but requires the user to explicitly specify the execution schedule of the generated code. This requires the user to have intimate knowledge of the target hardware in order to achieve optimal performance. As shown later in this paper, Forma provides a useful framework to provide whole program optimization and automatic GPU offloading of arbitrary computations to applications using NumPy and OpenCV.

The rest of the paper is organized as follows. Section 2 describes the syntax of Forma programs and their semantics. Section 3 describes the internal representation of the pipeline derived by the compiler from a Forma program and optimizations that are possible due to this representation. It also describes the architecture specific optimizations implemented by the back-end while targeting multi-core CPUs and NVIDIA GPUs. Section 4 describe several simple to complex image processing pipelines and shows the ease with which they could be expressed, along with the performance of the generated code on both CPUs and GPUs. Halide [24] another DSL used for image processing, is used as a basis for performance comparison of the generated code. Section 5 explains how Forma can be integrated with productivity tools like Python and OpenCV while highlighting the potential gains in performance. Section 6 describes other related research in this area. Sections 7 presents some future directions that we wish to explore based on the described framework with Section 8 stating the conclusion that are drawn from the work described here.

2. SYNTAX AND SEMANTICS OF FORMA PROGRAMS

Forma provides syntax to define and manipulate orthogonal domains, where each point in the domain is associated with some data. Basic data types supported are 8-bit integers (`int8`), 16-bit integer (`int16`), 32-bit integer (`int`), single-precision floating point number (`float`) or double-precision floating point number (`double`). User defined data structures with fields of basic data types are also supported. Therefore, an RGB image can be viewed as a rectangular domain of points, each point associated with a structure having three fields of type `int8`.

Listing 3 shows a simple Forma program. The input domain is declared at line 5 as a 2D domain of size $M \times N$, with each point containing an RGB value. Temporary variables can be used to store intermediate results within the pipeline. Forma uses a single-assignment paradigm, i.e. once assigned a variable cannot be reassigned. This ensures that the size of the domain represented by variables, once set, does not

```
struct RGB{
  int8 r; int8 g; int8 b;
}
parameter M,N;
vector#2 RGB input[M,N];
output = input;
return output;
```

Listing 3: Hello World in Forma

```
1 stencil jacobi2d(vector#2 int X){
2  y_pt = 4*X - (X@[-1,0]+X@[0,-1]+X@[1,0]+X@[0,1]);
3  return y_pt;
4 }
5 parameter M,N;
6 vector#2 int x[M,N];
7 y = jacobi2d(x:constant(0));
8 return y;
```

Listing 4: Stencils in Forma

change during the program execution. The compiler checks that every variable used within the program is either an input or has been defined previously. It is not necessary that the generated code allocates memory for every variable within the program, allowing it to inline/fuse multiple stages. The program ends with a `return` statement which specifies the result of the computation. Section 3.2.3 describes the interface that allows the user to pass the input domain and parameter values to the generated code and to receive the result of the pipeline.

2.1 Stencil

Stencils represent the basic operations of an image processing pipeline. In Forma, stencils are specified as functions that are *applied* on domains. Listing 4 shows a specification of 2D-Jacobi stencil. Stencil function definitions are prefixed with the keyword `stencil`. The arguments to the function specify the type and dimensionality of the domains operated on. In Listing 4 the input argument is a 2D domain of integers. When applied to a domain (at Line 7), the computation specified within the stencil function is performed at every point of the domain.

Within the stencil specification, Forma allows the use of an *offset operator*, '`@[...]`', to access neighboring points of input domains. The list of integers specified within the square braces represent the offset of the point whose value is to be used. Therefore `X@[-1,0]` refers to the value at point $(i-1, j)$ while performing the computation at point (i, j). Dropping the offset operator is a short-hand to refer to the value at point (i, j) itself. The right-hand side of statements within a stencil function definition can contain mathematical operations, math functions, or ternary expressions. Loops and conditionals are not allowed within a stencil definition. The `return` value is stored at point (i, j) of the output domain.

To allow the specification of stencil operations like the one described in Listing 1, a *scaling factor* can be used for every dimension in the offset operator. Instead of a list of integers, one can specify a list of tuples of the form (a, b) to refer to a point $a + i \times b$ while computing at the i^{th} point along that dimension. Listing 5 shows the Forma specification of the filter in Listing 1. Using an integer instead of a tuple for a

```
stencil sample_x(vector#2 int X){
 x_pt= (X@[(-1,2),0]+2*X@[(0,2),0]+X@[(1,2),0])/4;
 return x_pt;
}
stencil sample_y(vector#2 int Y){
 y_pt= (Y@[0,(-1,2)]+2*Y@[0,(0,2)]+Y@[0,(1,2)])/4;
 return y_pt;
}
parameter M,N;
vector#2 int image[M,N];
temp = sample_x(image);
return sample_y(temp);
```

Listing 5: Stencils with non-unit scaling factor

dimension, sets the scaling factor as 1 for that dimension. Using a scaling factor along a dimension reduces the size of the output domain along that dimension by the scaling factor. Therefore, the size of `temp` is half that of `image` along the x-dimension. The size of the output is half of the input `image` along both dimensions.

Boundary Conditions: Applying the stencil in Listings 4 and 5 along the edges of the output domain would be erroneous. By default, the computation is done only at those points of the output domain where applying the offset results in a valid point of the input domain. Forma also provides language support for the following standard boundary conditions:

- constant(val) : The values accessed outside the domain are set to a constant value, *val*
- clamped : The values accessed outside the domain are set to the value at the closest edge
- wrap : The domain is treated as being wrapped along all dimensions
- mirror : The values outside the domain are computed by treating the closest edge as a mirror.

Boundary conditions are specified while applying the stencil as shown in Listing 4, which applies the constant boundary condition. Since the stencil definition is decoupled from the boundary conditions to be used, the same stencil specification can be used with different boundary conditions at different points of a Forma program. In case of multiple input domains to a stencil function, each argument can have a different boundary condition.

2.1.1 Extrapolation

To support the extrapolation operation (also refered to as upsample) shown in Listing 2, Forma allows the use of the offset operator outside of a stencil definition. Listing 6 shows the syntax to specify the first step of the extrapolation operation. At line 7, use of the offset operator on the left-hand side of the assignment statement specifies that the value at a point (i, j) of the domain on the right-hand side (which is the result of a stencil application) is assigned to the point $(2i, j)$ of the domain represented by `temp`.

2.2 Compose Operation

While the operations described in Section 2.1 allows you to specify individual steps of the filters shown in Listing 1 and 2, the single assignment paradigm of Forma would not allow the programmer to assign to different points of a domain in successive statements. The *compose operation* can be used in such situations to specify the operations used to compute different portions of a domain. Line 7 of Listing 7

```
stencil upsample_x_even(vector#2 int X){
 x_even = (3*X + X@[-1,0]) / 4;
 return x_even;
}
parameter M,N;
vector#2 int image[M,N];
temp@[(0,2),0] = upsample_x_even(image);
```

Listing 6: Extrapolation

```
stencil upsample_x_even(vector#2 float X){
  return (3*X + X@[-1,0]) / 4;
}
stencil upsample_x_odd(vector#2 float X){
  return (X@[1,0] + X*3)/4;
}
out = ( @[(0,2),0] = upsample_x_even(in);
        @[(1,2),0] = upsample_x_odd(in); );
```

Listing 7: `Compose` Expression

shows an example of this operation used to specify the computation for the odd and even points along the x-dimension. Each statement within the compose operation is assumed to specify the evaluation of disjoint portions of the output domain, which can be computed independent of each other. In case where the regions are not disjoint, the behavior is unspecified.

2.3 Cropping and Stacking of Images

Forma allows the programmer to also extract an orthogonal sub-domain of points by using the *crop operator*. The region to be extracted is specified within '`[ ]`'. For example, at Line 2 of Listing 8, the crop operation is used to extract a rectangular region formed between the points $(3,5)$ and $(25,29)$. The size of the resulting domain (`y`) is 23×25.

A similar operator is the *padding operator* which is used on the left-hand side of a statement, an example of which is shown in Line 3 of Listing 8. It is similar to the crop operator with the upper-bound unspecified. The output domain is padded by the amount specified for each dimension. In the example shown, the values from the domain represented by `x` are copied to the domain `z` after being padded by 3 along the first dimension and 5 along the second. The resulting domain is of size $(M+3) \times (N+5)$. The number of elements in the list of the padding operator can be greater than the dimensionality of the right-hand side expression. The resultant domain has the same dimensionality as the number of elements in this list. When used in conjunction with the compose operation, this allows stacking of images as shown in Listing 9. The domains represented by `x`, is placed along the 0$^{\text{th}}$ index of the domain `stack`, with the elements of the former used to initialize the points along the inner dimensions. Domains `y` and `z` are used to initialize the points along the 1$^{\text{st}}$ and 2$^{\text{nd}}$ indices of the outermost dimensions of `stack`.

```
1 vector#2 int x[M,N];
2 y = x[3..25,5..29];
3 z[3..,5..] = x;
```

Listing 8: Crop Operation

```
vector#2 int x[M,N];
vector#2 int y[M,N];
vector#2 int z[M,N];
stack = ( @[0..,0..,0..] = x;
          @[1..,0..,0..] = y;
          @[2..,0..,0..] = z; );
```

Listing 9: Intersection Example

```
 1 stencil jacobi1D(vector#2 int X){
 2   return (X@[-1,0] + 2*X + X@[1,0])/4;
 3 }
 4 stencil bdy0(vector#2 int X){
 5   return (3*X + X@[1,0])/4;
 6 }
 7 stencil bdyN(vector#2 int X){
 8   return (3*X + X@[-1,0])/4;
 9 }
10 parameter N;
11 vector#2 int a[N,N];
12 b=([0..,0..] = bdy0(a[0..1,0..N-1]);
     [1..,0..] = jacobi1d(a)[1..N-2,0..N];
     [N-1..,0..]= bdyN(a[N-2..N-1,0..N-1]); );
```

Listing 10: Custom Boundary Conditions

Custom Boundary Conditions : The crop, padding and compose operator can be used to specify custom boundary conditions. For example, consider the `jacobi1D` stencil shown in Listing 10. This stencil is applied at all points of an image except the first and last column, where the stencils `bdy0` and `bdyN` are to be applied instead. Line 12 extracts the first and last column of the domain `a`, applies the respective stencils on these and places them along the first and last column of the output domain `b`. The interior points of `b` are computed by applying the stencil `jacobi1D` with default boundary conditions. The result of the stencil application is cropped at the edges and placed in the output with a padding of 1 along the x-dimension. The compose operator is used to specify these operations simultaneously in keeping with the single-assignment paradigm.

2.4 Vector Functions and Loops

The previous sections highlighted operations on domains that are supported within Forma that allow the user to specify a stage in an image processing pipeline. In this section we discuss a constructs that allow programmers to express complex pipelines succinctly by chaining together multiple stages.

2.4.1 *Vector Functions*

It is common for a sequence of filters to be applied repeatedly over the course of a computation. As an example consider the pipeline shown in Figure 2, which is used to combine multiple RGB images at different exposures to get an HDR image [19]. The filters **`exposure`**, **`make_grey`**, **`laplacian`**, **`saturation`** and **`multiply`** are applied in the same sequence for every input image. This sequence computes a weight to be used for each input image while combining them (`combine`) to get the final HDR image. Vector functions encapsulate a sequence of operations into a single function. In Forma, vector functions are defined by adding the keyword `vector` to the function definition as shown in Listing 11. Vector functions allow the Forma programs to be more concise, and easier to maintain. (The term vector

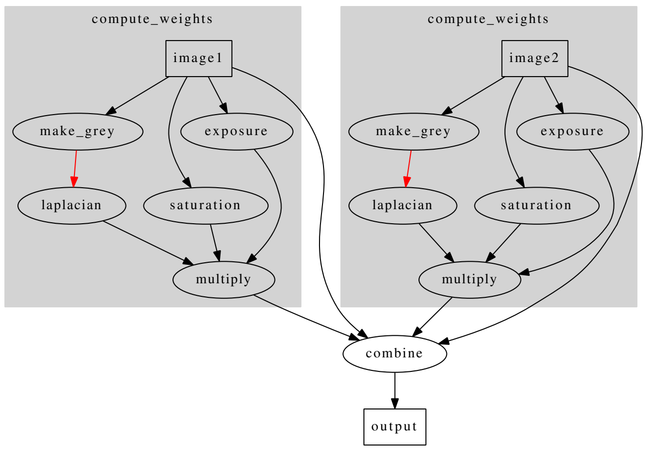


Figure 2: Basic Exposure Fusion algorithm

```
...
vector compute_weights(vector#2 rgb X){
  grey = make_grey(X);
  laplacian = laplacian(grey);
  saturation = saturation(X);
  well_exposed = well_exposed(X);
  return multiply(laplacian,
            saturation,well_exposed);
}
...
weights1 = compute_weights(image1);
weights2 = compute_weights(image2);
return combine(weights1,image1,weights2,image2);
```

Listing 11: Vector Functions

functions refers to operations on image objects, or vectors, as a whole. It does not refer to mapping operations to SSE vector instruction/intrinsics.)

2.4.2 *Loops and Qualified Variables*

For some image processing pipelines, loops provide a natural way of expressing the computation. A prime example of this is the Laplacian Pyramid [7] computation. Here, two image pyramids are constructed from the input image, namely the *Gaussian pyramid* and the *Laplacian Pyramid*. The input image forms the base of the gaussian pyramid. Successive level are constructed by sampling from the images at a lower level. The laplacian pyramid is constructed by extrapolating the images at a level of the gaussian pyramid and subtracting the image at the immediately lower level. Each level of gaussian and laplacian pyramids can be computed using loops.

Listing 12 shows the specification of the laplacian pyramid computation in Forma, with the use of loops at line 17. Loops within Forma can only have numeric bounds and increments of $+1$ or -1. The variables `gaussian` and `laplacian` are tagged by an additional parameter specified within '`< >`'. Such variables are refered to as *qualified variables* and represent a list of domains, each of which are of the same type but can have different sizes. To maintain the single-assignment property, all statements within a loop body can only assign to variables that are qualified. The compiler also ensures that a particular domain within a qualified variable has been defined before it is used. For example, in Listing 12

```
 1 ...
 2 vector sample(vector#2 float Z){
 3   return sample_y(sample_x(Z));
 4 }
 5 vector upsample_x(vector#2 float X){
 6   return
       (@[(0,2),(0,1)] = upsample_x_even(X);
        @[(1,2),(0,1)] = upsample_x_odd(X); );
 7 }
 8 vector upsample_y(vector#2 float Y){
 9   return
       (@[(0,1),(0,2)] = upsample_y_even(Y);
        @[(0,1),(1,2)] = upsample_y_odd(Y); );
10 }
11 vector upsample(vector#2 float Z){
12   return upsample_y(upsample_x(Z));
13 }
14 parameter M,N;
15 vector#2 float image[M,N];
16 gaussian<0> = image;
17 for i = 1..3
18    gaussian<i> = sample(gaussian<i-1>);
19    laplacian<i> = subtract(gaussian<i>,
                        upsample(gaussian<i-1>));
20 endfor
21 ...
```

Listing 12: Laplacian Pyramid Computation

the compiler checks that $(i-1)^{th}$ domain of qualified variable `gaussian`, has been defined before its use. As a result, the 0^{th} domain of `gaussian` has to be defined outside the loop-body for the program to be correct.

3. COMPILE-TIME ANALYSIS AND CODE GENERATION

The previous sections outlined the syntax and semantics of various operations that allow the programmer to express image processing pipelines in a compact manner. The Forma compiler uses this high-level specification to optimize across pipeline stages and generate code that can target both multicore CPUs and GPUs. These aspects are discussed in this section.

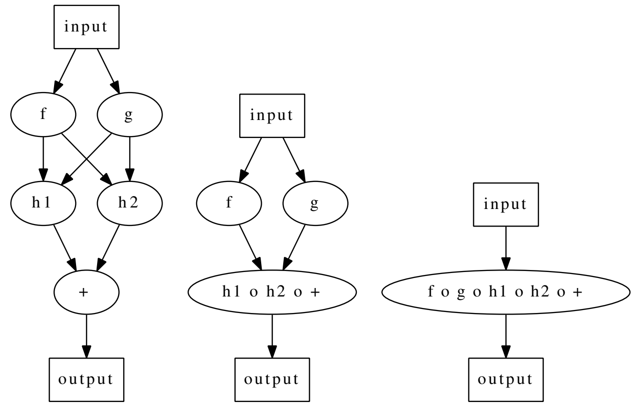
3.1 Computation DAG and Fusing Stages of the Pipeline

Directed Acyclic Graphs (DAGs) represent a convenient abstraction to represent the dependences between the different stages of an image processing pipeline. Figure 2 accurately captures the dependence between the different stages of a pipeline to compute HDR images. The single-assignment paradigm used in Forma allows the compiler to build a similar computation DAG for all Forma programs. Every stencil or vector function application is represented by a node in this DAG. An edge is added from one node to another when the result of a function application represented by the former is used as an argument to function represented by the latter. A separate DAG is built for each vector function. Before building a DAG, each loop is completely unrolled. This representation can be leveraged to merge (or fuse) multiple stages of the image processing pipeline. This not only reduces the number of intermediate buffers (and hence the memory footprint) needed for the computation, but also reduces the total memory bandwidth requirements.

Within Forma, fusion is performed only when it doesn't result in redundant computation. Consider the DAG shown in Figure 2. The filters `laplacian` and `make_grey` are shown

```
stencil make_grey(vector#2 rgb X){
  return (0.02126f * X.r + 0.7152f * X.g
           + 0.0722f * X.b) / 255;
}
stencil laplacian(vector#2 float X){
  return 4 * X - ( X@[-1,0] + X@[0,-1] +
                   X@[1,0] + X@[0,1] );
}
```

Listing 13: Laplacian Filter



(a) Initial DAG (b) First Pass (c) Second Pass

Figure 3: Multiple passes of Fusion

in Listing 13. If these two filters were combined, before using the value at the neighboring points to compute the laplacian, the grey value at all the neighboring points would have to be computed as well. This would result in redundant computation. To avoid this, stages are only combined when the function represented by the target node doesn't refer to neighbors of the result of the function represented by the source node. Since boundary conditions are meaningful only when neighboring points are accessed using a stencil, the fusion optimization does not have to consider handling boundaries.

Further, nodes are merged only if the source node has only one outgoing edge, i.e., the result at the source node is not used in multiple stages of the pipeline. Under these two constraints, the nodes `saturation`, `exposure` and `laplacian`, which do not refer to neighboring points of their input domains, can be merged with the node `multiple`. The node `make_grey` cannot be merged with `laplacian`. To fuse as many stages as possible, Forma inlines all vector functions beforehand.

Fusing two nodes might unearth further fusion opportunities. Consider a computation DAG shown in Figure 3a where none of the stages apply stencils that refer to neighboring points of input domains. Under the constraints described earlier, only the stages `h1` and `h2` can be fused with their successor, resulting in a computation DAG shown in Figure 3b. This results in the nodes `f` and `g` now having only one successor and can be fused with it. Therefore, to minimize the number of stages in the pipeline, the Forma compiler does multiple passes over the computation DAG, terminating only when no candidates for fusion are found. Figure 4 shows the computation DAG obtained as a result of the fusion optimization starting with the DAG in Figure 2.

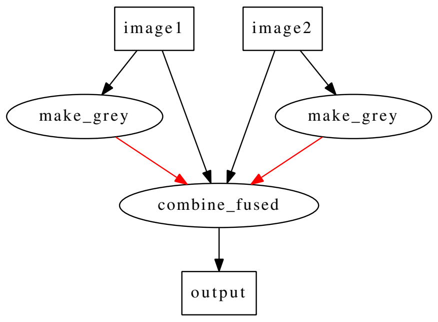


Figure 4: Fused computation DAG for exposure fusion algorithm

The number of stages in the computation reduces from 11 to 3, reducing the number of intermediate buffers needed from 10 to 2.

3.2 Code Generation

Forma compiler generates C code with OpenMP pragmas to target multi-core CPUs and CUDA code to target NVIDIA GPUs. Here we discuss architecture specific optimizations that are implemented within Forma to enhance the performance of the generated code.

3.2.1 *Targeting CPUs*

For multi-core CPUs, after fusing as many stages as possible (as discussed in Section 3.1), for each stage Forma generates loop nests to iterate over the output domain and apply the specified filter or stencil function. The size of all intermediate buffers are automatically computed by the compiler from the program specification, freeing the programmer from the burden of explicit memory management. By design, for each loop-nest generated, there is no loop carried dependence at any level. This allows the use of OpenMP pragmas to parallelize the outermost loop, at the same time enabling vectorization of the innermost loop through use of vectorization pragmas (like `#pragma ivdep`, `#pragma vector always` in ICC). Therefore the generated C code is able to exploit both coarse-gained parallelism across cores, and fine-grained parallelism through use of vector functional units on each core. Both of these are critical to obtain good performance on modern CPU architectures.

When no boundary conditions are specified for stencil function arguments, the generated loop-nests compute only points along the interior where applying the offsets to the input domains results in a valid memory location. To handle boundary conditions, additional loop-nests are generated to apply the appropriate boundary condition along the edges. Since the loop-nest for the interior points and the boundary points are disjoint, the code for the interior points avoids the use of conditionals that adversely affect vectorization. This is analogous to loop-peeling and allows the application of boundary conditions with relatively minimal overhead.

3.2.2 *Targeting GPUs*

For NVIDIA GPUs, Forma generates separate kernels for each stage in the image processing pipeline (after stages have been fused). Each thread on the device is responsible for computation of a single point of the output domain. Apart from handling memory allocation and deallocation on the device, Forma also generates code to move data to and from the device. The kernel code is generated such that the access to global memory are unit-stride, as far as possible, to enable coalesced accesses across threads. Similar to the CPU code-generation, when no boundaries are specified for arguments of a stencil function, the generated kernel ignores the points along edges of the output domain. Boundary conditions are handled by generating separate kernels for edges, and for the interior points. This avoids branches within the kernel for the latter, avoiding the cost of branch divergence.

Additionally, a compile-time option can be specified to utilize the texture units on the device. Buffers that are read-only within a kernel, and are pitch-linear in global memory can be mapped to the texture fetch units of the device[1]. These units exploit spatial locality of accesses from multiple threads to reduce the cost of global memory reads. In addition, the texture fetch units provide hardware support for handling the boundary conditions supported by Forma, i.e., constant, clamped, wrap and mirror. Using texture allows the GPU backend to implement these without generating separate kernels for the boundaries.

3.2.3 *Interface to the Generated Code*

The generated Forma code (for GPU or CPU) is encapsulated within a C-function. A header file is also generated with the signature of this function. The arguments to this functions are

- Variables that are defined as *input* within the Forma program,
- Parameters used within the program, and
- Pointers to memory location which should contain the result of the pipeline (specified along with the `return` expression in the program).

The header file also has a description of the size of the output buffer needed, which is computed by the Forma compiler but has to be allocated by the calling function. On returning from the Forma generated function, the output buffer contains the result of the pipeline.

4. CASE STUDIES AND PERFORMANCE EVALUATION

In this section we discuss several image processing pipelines that have been ported to Forma. These range from simple stencils stencils to complex multi-step pipelines which use multiple pyramids. Apart from demonstrating the ease with which these pipelines can be specified, we compare the performance of the C and CUDA code generated by the Forma backend with Halide[2]. ICC 14.1[3] was used for compiling the C code, while NVCC 6.5[4] was used for compiling the CUDA code. A few of these pipelines were ported from examples that are packaged with Halide. For such examples, the schedule tuned for x86 architectures was used. For those which are not in Halide, we developed the schedule ourselves by exploring many possibilities. All schedules for GPUs were also developed by us after significant effort to obtain the best execution times.

The experimental evaluation of the CPU back-end was done on an Intel i7-4820k which is based on the IvyBridge architecture and an Intel i5-4570 which is based on the Haswell architecture. Figure 5 shows the speed up of the Forma generated C code for different number of hardware threads with respect to the sequential execution time of the Halide generated CPU code. Also shown is the speed up of Halide generated CPU code for 2 and 4 threads. The performance of the code generated by the GPU backend was evaluated on two GPUs, a GTX 680 (compute capability 3.0) and a Tesla K20c (compute capability 3.5). For all the GPU codes the total execution time of all the kernels was obtained by using NVProf[5]. For each of the pipelines, the speed up of the Forma generated CUDA code with respect to the Halide generated GPU code is shown. The specifics of the data presented in Figures 5 and 6 will be discussed below.

```
stencil emboss(vector#2 float X){
  return X@[1,1] + X@[0,1] + X@[1,0] -
         X@[-1,0] - X@[0,-1] - X@[-1,-1];
}
```

Listing 14: Emboss filter in Forma

```
stencil blurx(vector#2 float X){
  return X@[-1,0] + X + X@[1,0];
}
stencil blury(vector#2 float Y){
  return Y@[0,-1] + Y + Y@[0,1];
}
return blury(blurx(input));
```

Listing 15: Blur Filer in Forma

4.1 Simple Stencils

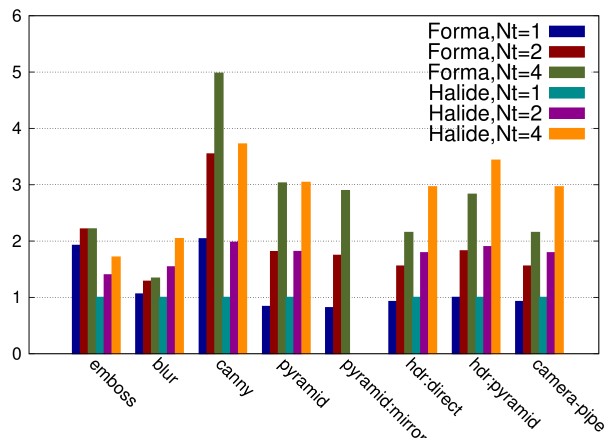
We start with two simple stencil applications, *emboss* and *blur*. The Forma code for these are shown in Listings 14 and 15, respectively. The former is a simple 5-pt stencil. The latter contains two stages. The first stage applies a 3-pt stencil along x-direction, followed by a 3-pt stencil along y-direction. For evaluation, these filters were applied to an input of 2048×2048 32-bit floating point values.

The GPU results show that the Halide generated code and the Forma generated code are on-par. The use of texture units results in significant improvements in the execution times for the emboss kernel. For blur as well, the performance of the code generated by Forma is comparable or better than that generated by Halide. On CPU, the Forma generated code is effectively vectorized by the host compiler, resulting in better performance than the Halide generated CPU code. For blur, the Halide schedule that interleaves the two stages delivers good performance, especially on the IvyBridge machine due to reduced bandwidth requirements. On the Haswell machine the performance of the Forma generated code and Halide generated code are comparable.

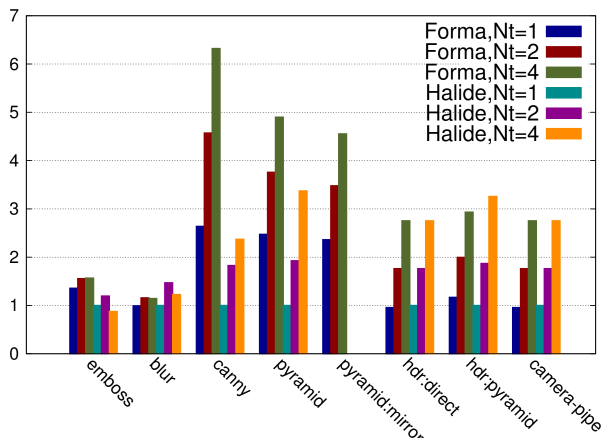
4.2 Canny Edge Detection

Canny edge detection in an image processing pipeline used to detect edges of images [8]. This was implemented in both Halide and Forma. The Forma specification is done in less than 10 lines, and without having to pay attention to intermediate buffer sizes or boundary computations. While the current implementation used a 5-pt gaussian blur stencil in the first step, it can be easily adapted to implement more complex stencil functions without significant programming

[1]https://docs.nvidia.com/cuda/cuda-c-programming-guide
[2]http://halide-lang.org/
[3]https://software.intel.com/en-us/c-compilers
[4]https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc
[5]https://docs.nvidia.com/cuda/profiler-users-guide



(a) Intel Core i7-4820k



(b) Intel Core i5-4570

Figure 5: CPU execution times: Speed up with respect to sequential Halide CPU code

effort. Making similar changes to a C/CUDA code would require a change to the entire computation since the sizes of the boundaries depend on the stencil used.

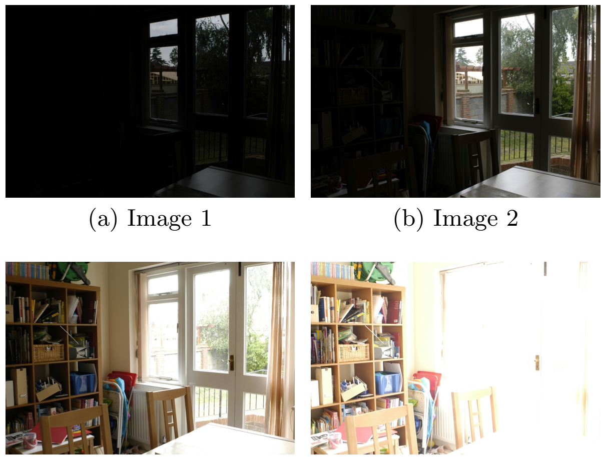
The CPU results in Figure 5 show that the Forma generated C-code performs significantly better than the Halide code for both Ivybridge and Sandybridge machines. The GPU results indicate that the use of textures gives a significant performance boost for this benchmark. This is because the most expensive step, the gaussian blur, is ideal for using texture fetch units on the GPU.

4.3 Laplacian Pyramid

Laplacian Pyramids were briefly introduced in Section 2.4.2. Implementation of pyramids in a low-level language, like C++/CUDA, is challenging since the images at different levels of the pyramid have different sizes. Further, they depend on the filter used for sampling and extrapolation. The application developer would have to compute these sizes manually and allocate memory for these intermediate levels. Using a filter with different sampling rates would require a change at all these levels.

In Forma, the pyramid computation can be expressed using just 26 lines of code, making it easy to debug and maintain. Since the compiler automatically figures out the sizes for each level of the pyramid from the program specification, the filters used can be easily modified. Handling boundary conditions adds an additional challenge while programming in low-level languages, while these can be specified easily in Forma with minimal changes to the code. Effect of different boundary conditions can also be explored easily.

The bars labeled *pyramid* in Figures 5 compare the performance of the Forma generated C-code with Halide for this pipeline. The schedule for Halide was obtained from examples available with the Halide package. The Forma generated code performs significantly better than the Halide generated code for the Haswell machine and is on-par for the Sandybridge machines. The bars labeled *pyramid:mirror* show the speed-up of the Forma C-code which implements mirror boundary conditions for all stencil applications, with respect to the sequential Halide CPU code for *pyramid*, i.e., without any boundary conditions. These results illustrate the negligible costs incurred while handling boundaries on the CPU while using Forma.



(a) Image 1 (b) Image 2

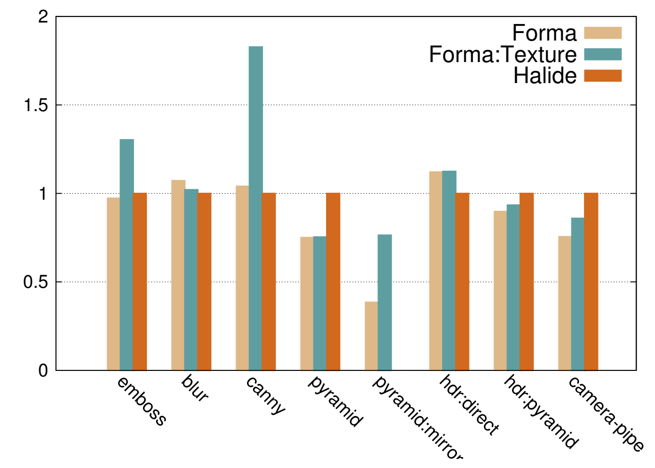
(c) Image 3 (d) Image 4

Figure 7: Inputs used for the Exposure Fusion [19]

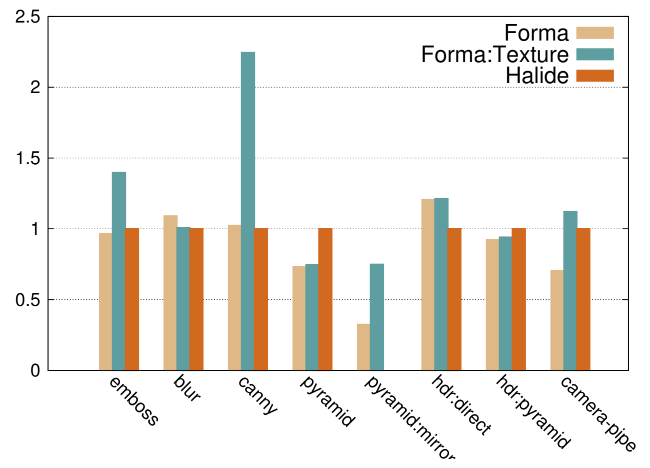
On GPUs, the benefit of using textures is illustrated by comparing the normalized execution times of *pyramid* and *pyramid:mirror* in Figure 6. Both the bars show the speed up achieved by the Forma code when compared to the Halide GPU code for *pyramid*. Without using textures, the default CUDA code uses additional kernels to handle boundary conditions, while the texture code uses hardware support for implementing them. Consequently, implementing the standard boundary conditions supported by Forma comes for free while using the texture fetch units.

4.4 Exposure Fusion

Section 3.1 introduced a simple technique to generate HDR images. In some cases using this scheme results in seam effects. Exposure Fusion [19] is method developed to address this issue. It starts by building 3 laplacian pyramids and 3 gaussian pyramids for each input image, one for each color. The laplacian pyramids for each color are combined using the method described in Section 3.1 to get the laplacian pyramid for each color of the output image. Finally the

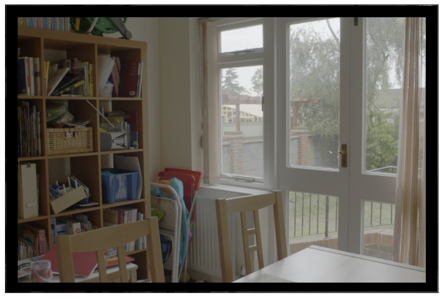


(a) GTX 680

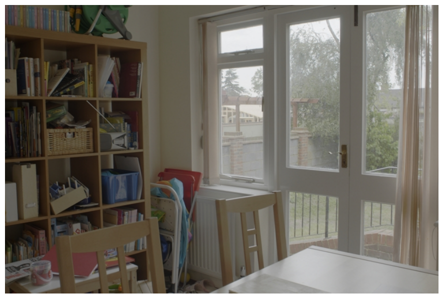


(b) Tesla K20c

Figure 6: GPU execution times: Speed up with respect to Halide GPU code



(a) Without Boundary Conditions



(b) With Mirror Boundary Conditions

Figure 8: Effect of improper boundary conditions

gaussian pyramid of the output image is computed to get the final HDR image. The Forma implementation of this method combines the 4 images shown in Figure 7, resulting in the construction of 30 pyramid objects. Improper handling of boundary conditions results in edge effects shown in Figure 8a. Figure 8b shows the final HDR images constructed when using mirror boundary conditions for all stencil applications. This complex image processing pipeline can be expressed in Forma with just 184 lines of code, using 25 stencil functions, 9 vector functions and 2 loops.

The bars labeled *hdr:direct* and *hdr:pyramid* in Figures 5 and 6 show that the performance of the Forma generated code is comparable to that of Halide for both CPU and GPU.

4.5 Camera Pipe

Camera pipeline is a benchmark from [24], and is used in many cameras to convert raw data from the sensors into an image. An implementation from the Halide package was used as a reference to develop the Forma implementation. The original Halide implementation used 32 functions and 22 different stencils. The Forma implementation used 12 `stencil` functions (it was easier to express the computation in Forma by combining some of the stencils) and 1 `vector` function (to define the `demosaic` stack used in the computation). Developing a schedule for such a complex pipelines is quite challenging even for an expert programmer, since the ideal choice of where to compute and store intermediate buffers used in the pipeline requires inter-procedural reasoning. Further, this has to be repeated for every target architecture. For this specific example, while the Halide implementation comes with a schedule tuned for x86 architectures, the GPU schedule was developed by the authors. The schedule that gave the best performance was obtaining by aggressive inlining, a strategy that diverges significantly from the CPU schedule.

In Forma, the same schedule is used to generate both CPU and GPU code. Even with the relatively simple scheduling strategy used in Forma, performance of the generated texture code on Tesla K20c and of the C code on Haswell CPU is comparable to that obtained from Halide after significant effort.

5. INTEGRATING FORMA WITH EXISTING TOOLS

5.1 Using Forma with NumPy/SciPy

The NumPy/SciPy [20, 14] ecosystem for Python has emerged in recent years as a high-productivity platform for image processing and scientific computing. It provides a high-level abstraction, and familiar development environment and language for engineers and researchers while achieving performance on-par with tuned native routines for many common operations. It does this by exposing a generic n-dimensional array type and many associated operations to

Table 1: Execution time for NumPy/SciPy and Forma (ms).

Operation	Python	NumPy/ Scipy	Forma
Emboss	9335.371	20.806	4.033
Exposure Fusion	84158.143	142.704	15.307
Rician Denoise 2D	9526.096	7.889	0.706
Rician Denoise 3D	10149.112	9.669	1.091

```
def emboss_arr(inp, outp):
  outp[1:-1,1:-1] = inp[2:,2:] + inp[1:-1,2:]
      + inp[2:,1:-1] - inp[0:-2,1:-1]
      - inp[1:-1,0:-2] - inp[0:-2,0:-2]
```

Listing 16: Emboss filter in NumPy

Python while implementing much of the functionality in optimized native code. Highly-tuned libraries such as BLAS [4] and LAPACK [2] are used to accelerate matrix operations, allowing Python code to take advantage of tuned implementations for a number of target platforms including supercomputer clusters and GPGPU devices.

Unfortunately, as NumPy and SciPy are just Python libraries, they are not able to accelerate user code that introduces new basic operations. Users that want to write new algorithms that cannot be efficiently composed of NumPy primitives must either (a) write NumPy array code in Python and pay the performance cost of executing in the Python interpreter, or (b) write their algorithm in optimized C, C++ or Fortran code and expose it as a native Python module. Ideally, a user could express the computation at a higher level and also benefit from any accelerator hardware available. Even if the user opts for choice (b), new code must be written for each type of target to ensure the best possible performance.

Consider the emboss filter written in NumPy shown in Listing 16. This implementation uses NumPy's vectorized access expressions to achieve decent performance on CPU targets. However, the Python interpreter executes each arithmetic operator sequentially. Even though adding/subtracting two arrays may be accelerated behind the scenes in optimized C code, there is no possibility of fusion across operations. In contrast, we can write the same algorithm in Forma (Listing 14) and expose the generated code as a Python function. Users can leverage this to write high-level code without sacrificing performance in a Python environment. NumPy is used as a data format to allow users to leverage Forma within the vast amount of NumPy application already written.

Table 1 shows the performance advantage of using Forma to generate NumPy kernels from a high level description. For each operation, the kernel is implemented using Python loops, using NumPy vectorized array expressions, and using Forma to generate a Python extension module. The times shown are the wall clock execution times for the kernel on a Core i7-4820, averaged across 20 runs, as timed by the calling Python code.

The advantage of this approach is that users can write high level descriptions of the operations they wish to perform while taking advantage of the underlying hardware. The Forma compiler takes care of generating optimal code and exposing a simple Python interface to the user. Accelerators such as GPGPUs can even be leveraged transparently by the Forma compiler. The results show that the code generated by Forma is much more efficient than the corresponding Python NumPy implementation. While the vectorized array expression syntax in NumPy can greatly accelerate Python array code compared to standard Python loops, it does not enable optimization across individual array operations. The operations may be highly optimized in the NumPy library, but they are executed sequentially by the Python interpreter. Forma, on the other hand, has global knowledge of the entire Forma program and can perform useful optimizations like fusion and parallelization across different parts of the computation.

Table 2: Operation execution time for OpenCV and Forma (ms).

Operation	OpenCV	Forma
Emboss (`filter2D`)	4.065	2.701
Rician Denoise 2D	66.558	15.120
Rician Denoise 3D	49.314	9.120

5.2 Using Forma with OpenCV

OpenCV [22] is a popular C++ library for computer vision. Like NumPy, it provides an n-dimensional array type and many pre-optimized implementations of image processing operations. Users can combine operations to form implementations of more complex algorithms, and automatically benefit from the tuning work done within OpenCV. However, for users writing new image processing filters that are not expressible as lower-level OpenCV operations, OpenCV provides little performance and productivity advantage. The developer must write loop code in C++ and either optimize it by hand or rely on an optimizing compiler to parallelize and tile the code sufficiently. Most C++ compilers are also unable to optimize across OpenCV operations as they are just pre-compiled functions.

Forma provides an easy way to write higher-level image operations that are automatically optimized for the target machine. Since the OpenCV `Mat` data type allows users to obtain a direct pointer to image memory, Forma-generated code can be easily adapted to operate on OpenCV `Mat` variables. Table 2 shows the performance of Forma-generated code used with OpenCV data types compared to the same operations written in C++ and optimized by the host compiler (GCC 4.8.2). Simple stencil operations like the *emboss* operation are implemented using the OpenCV `filter2D` function that applies a kernel matrix to an image. Though this function is optimized in OpenCV, it is still a general implementation that must work with arbitrary kernel matrices. Forma on the other hand generates code that is specialized for the embossing operation and can be better optimized. The rician denoising operations have complex per-pixel interactions and do not have a corresponding OpenCV implementation that we can use. In this case, the Forma-generated code is much faster than the hand-written loop code. In both cases, the generated code fits nicely into larger OpenCV applications.

6. RELATED WORK

Domain-Specific Languages are an attractive framework that allows programmers to raise the level at which a computation is specified. As a result the compiler is less encumbered by specific programming methodology used for the implementation and is free to choose an optimal strategy. Many research groups have developed DSLs to target different applications or computation paradigms. Due to the wide-spread use of stencils, many DSLs have been developed to address stencil computations.

The Pochoir [25] stencil compiler embeds a DSL within C++. The DSL syntax can be parsed by standard C++ compilers using a template library, or by the Pochoir compiler that generates high-performance C++ code using cache-oblivious algorithms to generate efficient tiling. SDSL [12, 13] is a stencil DSL that addresses the issue of time-tiling using the approach described in [15] to target both SIMD architecture and GPGPUs. Unlike Forma, both these DSLs focus on stencil computations that are iterative in nature with values in buffers modified and used repeatedly. Forma, on the other hand, focuses on computations that are more naturally represented as a DAG. Patus [10] allows the programmer to decouple the stencil specification from the scheduling strategy for efficient execution. The programmer can also specify auto-tuning parameters for the compiler. Partans [17] is a stencil DSL that focuses on optimizations for multi-GPUs. Other DSLs like Diderot [9] and Spiral [23] raise the level of abstraction even higher by allowing the programmer to specify computations in terms of mathematical operations.

Two frameworks that target the same space of applications are Hipacc [18] and KernelGenius [16]. Hipacc is a DSL embedded in C++, that allows specification of simple convolutions and targets both CPU and GPU. It supports much of the boundary conditions that are supported by Forma, but does not support features like sampling, extrapolation, or pyramid objects. KernelGenius on the other hand uses a similar DAG abstraction for modeling impage processing pipelines. The same abstraction is then used for implementing tiling strategies. The input specification, while more verbose, can be used to express many of the same applications targeted by Forma. Due to the nature of hardware targeted by this framework, the focus was more on optimizing data movement and tiling, rather than on exploiting parallelism. Some of the same scheduling strategies are being explored in Forma as well for improving the performance of the generated code.

The work that comes closest to Forma is the Halide [24] image processing compiler. Embedded within C++, it provides a functional abstraction for images with the value of a pixel viewed as the result of a function application. Intermediate images are functions, and image processing pipelines are formed by composing expressions of functions. Halide uses many optimization techniques like spatial tiling and sliding-window optimizations to generate high-performance code on CPUs and GPUs, but relies on the user to specify the parallelism and tiling scheme. For complex benchmarks described in this paper, it can be quite challenging to specify an optimal schedule, especially across multiple functions. While Halide can auto-tune for various parameters like tile-sizes, unroll factors, etc. for many applications the search space can become intractable. In this paper, we have shown that for many representative benchmarks performance equivalent to that of a tuned Halide implementation can be achieved without auto-tuning or having the user specify a schedule. This was achieved with a relatively naive scheduling strategy used within Forma, by generating code in a form that can be aggressively optimized by device-specific compilers like ICC and NVCC. Section 7 lists some of the other scheduling/target specific optimizations that will be added to Forma in future iterations. Further, in Forma the boundary condition to be applied is not tied to the variable but is specified at the time of stencil application. This approach allows the same intermediate image to be used with different boundary conditions at various program points.

The polyhedral compilation framework [11, 1, 21] provides a powerful abstraction for affine programs, viewing statement instances within loops as points in a convex polyhedron. Several compiler frameworks use this abstraction to enhance reuse through tiling [6], to exploit shared-memory parallelism and to target GPGPUs [3]. AlphaZ [26] is a DSL that allows the user to setup producer-consumer relationships between points in an iteration space. While semantics of Forma are inspired by such abstractions, the dependences are tracked at a granularity of domains. Indeed, polyhedral techniques could be used to improve the code generated by Forma through use of techniques like tiling and vectorization.

7. FUTURE WORK

In the presented work, we focused on the design of the language and semantics of Forma. The aim was to make the language expressive enough to allow specification of generic image processing pipelines in a convenient manner. While our code-generation uses OpenMP to utilize multiple cores of the CPU and optimizes for the data-accesses on GPUs, we want to incorporate more aggressive optimizations, like tiling and other loop transformations. The code generated from Forma programs is affine and can therefore be represented using the polyhedral compilation model. Our goal is to use this to help generate efficient tiled, parallelized, and vectorized code. This will allow us to leverage the large body of research that has gone into polyhedral code optimization. We also want to address hybrid code generation where different stages of an image pipeline are executed on either the CPU or GPU device. Such functionality is imperative for taking advantage of the growing number of heterogeneous architectures coming to both the workstation and mobile markets.

8. CONCLUSION

This paper describes the syntax and semantics of Forma, a new DSL for image processing applications that allows programmers to conveniently express several complex image processing techniques. The ease of programming in this DSL has been demonstrated through several complex and real-world image processing applications. These implementations were developed by the authors based on a description of the computation in literature in a matter of days if not hours. The developed code is easy to read and maintain. We have also shown how the compiler can use this high-level description of the computation to target both multi-core CPUs and GPGPUs, while making use of specific architectural features of the target such as texture units on GPUs. The achieved performance is in many cases better than current state of the art DSLs targeting image processing application. We also describe how Forma can be incorporated into existing tools like Python and OpenCV, and show the potential benefits of such an integration. In future, the DSL compiler would be linked with other traditional optimizing compilers (like the polyhedral compilers [21, 5]) to generate better quality code, further improving the performance of the applications without needing any change in its specification.

9. ACKNOWLEDGEMENTS

We would like to thank the following people from NVIDIA for their inputs and contributions : Sean Lee, Yuan Lin, Thibault Lutz, Alberto Magni, Kari Pulli, Yun-Ta Tsai, Alejandro Trocolli, David Pajak, Shervin Emami and Thierry Lepley. We would also like to thank Prof. P. Sadayappan from Ohio State University for his comments. Finally, we thank the reviewers of this paper for their helpful comments regarding related work and possible enhancements.

10. REFERENCES

[1] Omega project: Frameworks and algorithms for the analysis and transformation of scientific programs. `http://tinyurl.com/lxeeu29`.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK users' guide, 1999.

[3] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In *CC*, 2010.

[4] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (blas). *ACM Trans. Math. Softw.*, 2002.

[5] U. Bondhugula. PLuTO:An automatic parallelizer and locality optimizer for multicores.

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

[7] P. Burt and E. Adelson. The laplacian pyramid as a compact image code. *Communications, IEEE Transactions on*, 1983.

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

[9] C. Chiw, G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer. Diderot: a parallel dsl for image analysis and visualization. *PLDI*, 2012.

[10] M. Christen, O. Schenk, and H. Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *IPDPS*, 2011.

[11] P. Feautrier. Dataflow analysis of array and scalar references. *IJPP*, 1991.

[12] 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.

[13] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on GPU architectures. In *ICS*, 2012.

[14] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python.

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

[16] 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.

[17] T. Lutz, C. Fensch, and M. Cole. Partans: An autotuning framework for stencil computation on multi-GPU systems. *TACO*, 2013.

[18] R. Membarth, F. Hannig, J. Teich, M. Korner, and W. Eckert. Generating device-specific gpu code for local operators in medical imaging. In *IPDPS*, 2012.

[19] T. Mertens, J. Kautz, and F. Van Reeth. Exposure fusion. In *CGA*, 2007.

[20] T. E. Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 2007.

[21] L.-N. Pouchet. Pocc : the polyhedral compiler collection.

[22] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov. Real-time computer vision with OpenCV. *Communications of the ACM*, June 2012.

[23] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 2005.

[24] 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. *PLDI*, 2013.

[25] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *SPAA*, 2011.

[26] 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.