

## PYGPU: A HIGH-LEVEL LANGUAGE FOR HIGH-SPEED IMAGE PROCESSING

**Calle Lejdfors** *Department of Computer Science, Lund University, PO Box 118, 221 00 Lund, Sweden*

**Lennart Ohlsson** *Department of Computer Science, Lund University, PO Box 118, 221 00 Lund, Sweden*

### ABSTRACT

Image processing is an area with many computationally demanding algorithms. When implementing an algorithm the programmer has to make the choice of either using a high-level language, thereby gaining rapid development at the expense of run-time performance. Or, using a lower level language, having higher run-time performance but also a higher implementation cost. In this paper we present PyGPU, an embedded language that enables image processing algorithms to be written in the high-level, object-oriented language Python. PyGPU functions are compiled to execute on the graphics processing unit (GPU) present on modern graphics cards, a streaming processor capable of speeds more than a magnitude higher than those of current generation CPUs. We demonstrate a number of common image processing algorithms, showing how these can be implemented succinctly and clearly using high-level abstractions, while at the same time achieving performance close to theoretical peak figures.

### KEYWORDS

Graphics processing units, image processing, high-level languages

## 1. INTRODUCTION

Using a high-level language for writing software comes with many benefits. The code is typically easier to read and understand, making bug spotting easier. The time spent programming is reduced since the programmer need not worry about low level details such as memory management and data storage formats. In the field of image processing, MATLAB is a popular choice of high-level language. It is based on an array programming model in which

algorithms are expressed on whole images instead of their individual pixels. For example, adding two equal sized images A and B is written simply  $A+B$ .

The downside of high-level languages is poor performance. Even though the individual operations have efficient implementations, the overall performance is generally not enough for computationally intensive applications such as real-time motion-tracking or high-resolution video post-processing. To overcome this lack of performance it is often necessary to implement the algorithm in a lower-level language, such as C/C++ or FORTRAN, instead. However, this comes at a substantial increase in implementation cost, mainly in terms of programmer effort. Using a third-party image processing library such as Intel's Integrated Performance Primitives [Intel IPP], the open computer vision library [OpenCV], or Mimas [Amavasai], that provide optimized versions of standard algorithms, it is possible to reduce this cost somewhat. However, the total implementation cost of using a high-performance, lower-level language is typically much greater than when using a higher-level language.

Recently, there has been increased interest in using the graphics processing unit (GPU) present on modern graphics cards as a computational co-processor. The GPU is a highly specialized processor that provides very good performance. On some problems it is capable of outperforming current-generation CPUs by more than a factor of ten [Krüger and Westermann, 2003]. Programming the GPU is done using specialized languages such as NVIDIA's Cg [Mark et al, 2003], Microsoft's HLSL [Gray, 2003], or GLSL by the OpenGL ARB [Kessenich, Baldwin, and Rost, 2003].

Unfortunately, taking advantage of the performance of the GPU requires expressing an algorithm in terms of graphics primitives such as polygons and textures. Doing this requires intimate knowledge of modern real-time graphics programming. Consequently, implementing image processing algorithms to take advantage of GPU comes at a significant implementation cost, even compared to using lower-level languages.

In this paper we present PyGPU, a language for programming image processing algorithms that run on the GPU. It is implemented as an *embedded language* [Hudak, 1996] in the high-level, object-oriented language Python. PyGPU uses a point-wise image abstraction that, together with the high-level features of Python, allows image processing algorithms to be expressed at a high level of abstraction. By using the GPU for execution, PyGPU is able to achieve performance in the order of 2–16 GFLOPS without optimizations even on low-range hardware. This is more than enough to perform real-time edge-detection, for instance, on high-definition video streams.

The rest of this paper is organized as follows: In Section 2 we introduce PyGPU and show a number of example image processing-related algorithms. In Section 3 we discuss performance considerations. In Section 4 we give a description of how the PyGPU compiler is implemented. Section 5 contains an overview and discussion of PyGPU and how the restrictions and capabilities of the GPU affect how algorithms are implemented. Finally, in Section 6 we summarize the contributions made in this paper.

## 2. PYGPU

PyGPU is a domain-specific language for image processing with a compiler that can generate code which executes on the GPU. It is implemented as an embedded language in Python. An embedded language is constructed by inheriting the functionality and syntax of an existing

*host* language. This enables PyGPU to get a lot of high-level language features for free. Python, with its dynamic typing and flexible syntax, allows the embedding to be made very natural manner. Furthermore, using the extensive reflection support of Python, the PyGPU compiler can be implemented very concisely as described in Section 4..

The fundamental abstraction in PyGPU is its image model. An image is modeled as a function from points on a 2-dimensional discrete grid to some space of colors (RGB, YUV, gray-scale, CMYK, etc). As will be shown, this functional model admits expressing image processing algorithms concisely using the high-level language constructs of Python. Also it has the advantage of mapping naturally to the capabilities and restrictions of the GPU.

Below is a small PyGPU function implementing a simple skin detector. It uses the fact that the color of human skin typically lies within a bounded region in the chrominance color plane:

```
@gpu
def isSkin(im=DImage, p=Position):
    y,u,v = toYUV(im(p))
    return inRange(u, uBounds) and \
           inRange(v, vBounds)
```

Looking at the function we see that it has a decorator named `@gpu`. This is a directive to PyGPU's compiler to generate code for the GPU for this function. The default values, `DImage` and `Position`, are type-annotations that are required to compile the function for the GPU.

Apart from these details the function looks like ordinary Python code. The function body shows that to determine if the pixel `p` contains skin we first transform the color value of the pixel `p` in the image `im` to the YUV color space. Then we check if the red and blue chrominance values `u` and `v` both lie within the specified bounds. Applying the skin detector to an image is done by calling it as an ordinary Python function:

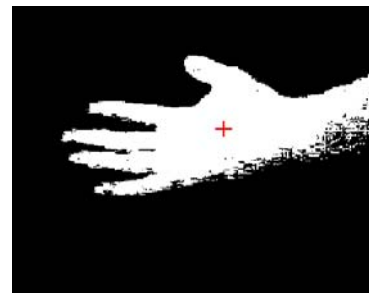
```
skin = isSkin(hand)
```

Note that the position argument is omitted, the skin detector is applied to the whole image. The result is shown in to the right.

The functions `toYUV` and `inRange` are examples of functions from the standard library of PyGPU. This library also provides standard mathematical operations such as basic arithmetic operators, trigonometric functions, and logarithms. These operations work on both scalars and, element-wise, on vectors. PyGPU provides vectors of dimension two, three, or four. Vector operations such as scalar products, and multiplication by scalars are provided through operator overloading, giving an obvious semantics to an expression such as `v+a` where `v` is some vector and `a` either a vector or a scalar.

## 2.1 Convolutions

The skin detector is an example of the most basic kind of image operations where each pixel in the result image only depends on the pixel at the same position in the sources image(s). Many algorithms, however, require access to multiple source image pixels to compute a single



pixel in the result image. Convolution operations, such as differentiations and filters, are typical examples of such algorithms. One example of a convolution is the Sobel edge detector seen below. The edge strength of a pixel is determined as the length of an approximation of the image gradient.

```
@gpu
def sobelEdgeStrength(im=DImage, p=Position):
    Sx = outerproduct([1,2,1], [-1,0,1])
    Sy = transpose(Kx)
    return sqrt(convolve(Sx, im, p)**2 + \
                    convolve(Sy, im, p)**2)
```

The gradient is estimated by the convolution of the so called Sobel kernels, one for the horizontal and one for the vertical direction. One can conveniently be expressed as the outer product of two vectors and by symmetry the other is the transpose of the first one.

This example shows a particularly powerful aspect of PyGPU. The functions `outerproduct` and `transpose` are not PyGPU functions but come from Numpy, an established high performance Python array programming library implemented in C [Numerical Python]. And yet these functions can be used in code that is compiled for the GPU. The reason this works is that the compiler uses generative techniques [Czarnecki and Eisenecker, 2000] to partially evaluate the code at compile-time (see Section 4).

In addition to allowing the use of already available extension libraries, this generative feature makes it possible to use high-level language constructs such as lists and list comprehensions or built-in standard Python functions even though these features cannot be directly translated to the GPU. For example, the `convolve` function used above can be succinctly expressed as:

```
def convolve(kernel, im, p):
    return sum([w*im(p+o)
                for w,o in zip(ravel(kernel),
                              offsets(kernel))])
```

The Numpy function `ravel` is used to compute the column-first linearization of the kernel. Using the built-in Python function `zip` to combine each kernel element with its corresponding offset (computed by the `offsets` helper function), the list of weighted image values can be expressed as a list comprehension. The final result is then computed by the standard Python function `sum`.

## 2.2 Iterative algorithms

The operations presented thus far have been algorithms where the result is computed in a single pass. Many operation use an iterative strategy where successive applications gradually improve the quality of the result. One example of such an algorithm is anisotropic diffusion filtering [Perona and Malik, 1990] that allows efficient removal of noise without simultaneously blurring edges in an image. One step of Perona-Malik anisotropic diffusion can be expressed as:

```
@gpu
def pmAniso(edge=DImage, im=DImage, p=Position):
    offsets = [(1,0), (-1,0), (0,1), (0,-1)]
    return im(p) + 0.25*sum([f(edge, im, p+dp, p)
                            for dp in offsets])

def f(edge, im, x, p):
    return g(0.5*(edge(x)+edge(p)))*(im(x)-im(p))
```

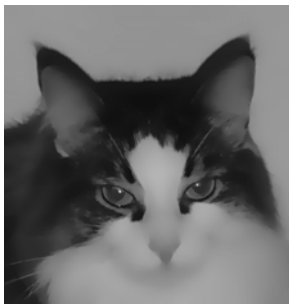


```
def g(x):
    return e**(-x/(K*K))
```

The function `pmAniso` is the main function that is compiled for the GPU and the functions `f` and `g` are helper functions which are generatively evaluated during the compilation process. The function `g` controls the conduction coefficients of the diffusion process with `K` determining the slope. The choice of function names here is the one used in the original paper. Iteratively applying the diffusion operator to an image can either be done by the standard PyGPU function `iterate` or by direct loop:

```
edges = edgeStrength(im)
for i in range(n):
    im = pmAniso(edges, im)
```

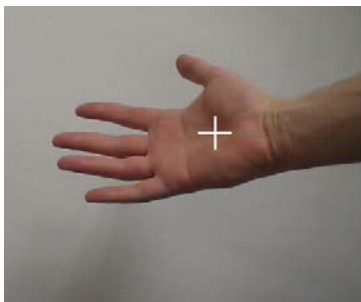
This results in successively more smoothed versions of the original image. The figure to the right shows an example image and the result of applying 400 iterations of the anisotropic diffusion operator using  $K=0.25$ .



## 2.3 Reductions

One common pattern in the above examples is that the result of the operation is always another image. In image analysis, however, it is often the case that the result of an operation is instead some overall property of the image, for example the maximum or average image color. These kinds of operations are called *reductions*, operations which reduce the size of an image down to a single value or set of values. For example, a function which computes the pixel-wise sum of an image can be implemented as:

```
def sumIm(im):
    return reduceIm(add, im)
```



Here, the function `add` is passed as an argument to a general `reduceIm` operation. This function is provided by PyGPU and works analogously to Python's built-in `reduce` but on 2-dimensional images instead of on lists. It is implemented as an iterative algorithm similar to the example in the previous section. Its implementation will be shown in Section 2.7.

A useful example of a reduction is the calculation of the center of mass of a region in a binary image. It can be used, for instance, to approximate the center of a hand or face detected by the skin detector above. The center of

mass is the average position of all pixels in the region and can be computed as:

```
def centerofmass(im):
    return sumIm(pos(im))/sumIm(im)
```

```
@gpu
```



Figure 1. Seamless cloning

```
def pos(im=DImage, p=Position):
    return p*im(p)
```

The result of applying the center of mass detection algorithm to the result of the skin detector above can be seen to the right.

## 2.5 Multi-grid operations

One of the advantages of programming in high-level languages is that the abstraction mechanisms available makes it possible to package complex operations as basic building blocks that can be used to construct even more complex operations. As an example we will show the implementation of an operation from the notion of *Poisson editing* introduced by [Pérez, Gangnet, and Blake, 2003].

The example is called seamless cloning and it is a technique for pasting parts of one image into another in such a way that there is no visible seam between the two images. The idea is to solve the Laplace equation for both images and only replace the differences from these solutions in the pasting operation.

The Laplace equation states that the sum of the second derivatives should be equal to zero. In the case of discrete images this is equivalent to saying that a pixel should be equal to the average of its four nearest neighbors. This average is computed by the following PyGPU function.

```
@gpu
def crossAverage(im=DImage, p=Position):
```

```
offsets = [(1,0), (-1,0), (0,1), (0,-1)]
return sum([im(p+o) for o in offsets])/4
```

Using the standard higher-order PyGPU function `masked`, that applies a function within a given mask and leaves the values outside unchanged, we can express one part of the Laplace equation solver as:

```
x = masked(crossAverage, m)(x)
```

The statement is of the same form as in the anisotropic diffusion example above. It can be used as the basic step in an iterative solver where each iteration yields a successively better solution. The complete implementation of seamless cloning can be expressed succinctly as:

```
def solveLaplace(x, mask):
    return iterate(n, masked(crossAverage, mask), x)

def seamlessCloning(source, target, mask):
    source0 = solveLaplace(source, mask)
    target0 = solveLaplace(target, mask)
    return (source-source0) + target0
```

An example of seamless cloning can be seen in Figure 1.

The Laplace solver above will eventually reach a solution, but it converges very slowly. For the example in Figure 1 it requires on the order of 10 000 iterations to compute `source0` and `target0`, respectively. A standard technique to improve convergence is to use a *multi-grid* approach where solutions are first found at a lower resolution. This approximate solution is then used as input to solving the problem at the higher resolution level, giving a better initial value for the solution and thereby achieving faster convergence. By changing the definition of `solveLaplace` to

```
def solveLaplace(x, mask):
    return maskedMultigrid(n, crossAverage, mask, x)
```

The example instead converges in around 200 iterations. The `maskedMultiGrid` solver is available in the standard library of PyGPU. Its implementation will be shown in Section 2.7.

## 2.6 Sparse operations

The kind of image operations where the parallelism of the GPU is most efficiently used are *dense* operations, where the computations involve all pixels in the image. All operations we have shown so far are all examples of this kind. *Sparse operations* on the other hand operate only on a well chosen subset of points in the images, for example feature points such as detected corners. The irregular access pattern used by sparse methods make them less suitable for implementation on the GPU.

Some kinds of operations use a combination of dense and sparse methods. One class of such operations are active contours or snakes [Kass, Witkin, and Terzopolous, 1988] where a polygon is used to define an image area that is interesting in some sense. The contour can automatically search for its area by iteratively moving the polygon until a local minimum is found on a suitably defined *energy function*. This function typically consists of a weighted average of two separate components: the internal energy and the external energy. The external energy is a measure of the image being analyzed, whereas the internal energy is a measure of the shape of the contour itself, for example its smoothness.

The idea is to sample the neighborhood of each vertex of the snake and if any position in this neighborhood gives the vertex a lower energy it is moved to this position. This step is then repeated as many times as needed. A simple implementation of active contours is:

```

def externalEnergy(im, vs, o, v):
    return im(vs(v)+o)[0]

def internalEnergy(vs, o, v):
    p,x,n = [vs((v+i)%nVerts)[0:2]
              for i in [-1,0,1]]
    x += offset
    m = (p+n)/2
    return norm(x-m)/norm(p-m)

def totalEnergy(wInt, wExt, im, vs, o, v):
    return wInt*internalEnergy(vs, o, v) + \
           wExt*externalEnergy(im, vs, o, v)

@gpu
def energyOptimize(wInt=Float, wExt=Float,
                  im=DImage, vs=DImage, v=Int):
    offsets = array([[0,0], [1,0], [-1,0], [0,1], [0,-1]])
    energies = [totalEnergy(wInt,wExt,im,vs,v,o)
                 for o in offsets]
    return vs(v) + min(zip(energies, offsets))[1]
    
```

Here, the parameters `im` and `vs` contain the image we are optimizing over and the vertices of the polygon, respectively. The weights `wExt` and `wInt` contain the relative weights of the external and internal energy. The use of `min` relies on the fact that comparison between tuples in Python is defined lexicographically. This means that we will find the energy minimum since this is the first member in each tuple. The corresponding offset of that energy minimum is given as the second tuple entry.

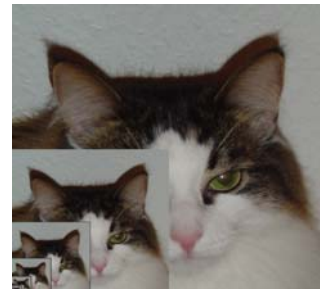
The input image used for the external energy is typically not the image being analyzed but rather some preprocessed version, for example a segmented version with edge enhancements. The internal energy shown here is simply a measure of how far a position is from the midpoint of the two neighboring vertices. This choice will give a “rubber band”-like snake contour where a enclosed region is always convex. Many other variants are possible. The result of applying the snake algorithm is shown to the right.



## 2.7 Implementation of some generic operations

In the previous sections we have used some generic high-level operations such as `reduceIm` and `maskedMultigrid`. Although these are very general and powerful, their implementation in PyGPU is still fairly simple. The reduction operator is implemented by successively applying the base operation to blocks of the image, resulting in smaller and smaller intermediary results. When the size of the image is  $I \times I$  it will contain the sought quantity as illustrated in the figure at the lower right. For a square image having sides that are a power of two, the operation can be implemented in PyGPU as:

```
block = array([(0,0),(0,1),(1,0),(1,1)])
```





```

def reduceIm(f, im):
    @gpu
    def _reduce(im=DImage, p=Position):
        return f([im(2*p+o) for o in block])

    while im.size[0] >= 1 and im.size[1] >= 1:
        im = _reduce(im, _targetSize=im.size/2)

    return im

```

This inner function, which is the one executed on the GPU, successively applies the function `f` to  $2 \times 2$  blocks of the image `im` until it is reduced to a single  $1 \times 1$  image. The actual reduction in image size is achieved by the parameter `_targetSize` which is implicitly made available on all PyGPU compiled functions with a default value of the size of the input image.

A multi-grid solver first finds an iterative solution on a coarse resolution of the image which is then used as the initial value on successively finer resolutions. This masked multi-grid solver in PyGPU can be expressed as:

```

def maskedMultigrid(n, f, mask, x, minSize):
    y = None
    for x, m in reversed(zip(averageR(im, minSize),
                             averageR(mask, minSize))):
        if not y: y = x
        else: y = masked(inflate(y), m)(x)
    y = iterate(n, masked(f, m), y)
    return y

```

The `averageR` helper function generates a sequence of successively coarser representations of an image down to size `minSize`. The function `inflate` does the opposite, i.e., it computes the input to the next higher resolution level.

### 3. PERFORMANCE

PyGPU is built on top of the OpenGL [Shreiner, 2003] and currently uses NVIDIA's Cg [Mark, 2003] as code-generation target. The compiler uses a combination of translative and generative techniques that, together with the introspection features of Python, allows high-level Python code to be translated to the GPU. This process is described in detail in Section 4. The run-time system of PyGPU is centered around OpenGL framebuffer objects. By using framebuffer objects it is possible to avoid unnecessary copying of image data on the GPU, resulting in very good performance.

Although the compiler of PyGPU does not yet implement a number of important optimizations it typically achieve between 0.5 and 4 GPixel operations per second (roughly equal to 2 to 16 GFLOPS) on the examples shown in this paper. This means that a 9-tap convolution filter can be applied to a  $500 \times 500$  RGBA color image in about 13 ms. The examples were run on a NVIDIA GeForce 6600 graphics card, a low-range card at the time of writing.

Table 1 gives a summary of the performance figures for the most representative examples in this paper. The execution times are essentially proportional to the number of pixels times the number of instructions in the compiled shader program to execute for each pixel. They also include a constant overhead for each pass for setting up the graphics cards, passing parameters

to the GPU program, and constructing the result texture. This overhead corresponds roughly to the computation of a couple of thousand pixels, meaning that it is negligible for larger images.

The theoretical peak performance of the NVIDIA 6600 card of our test setup is approximately 4.8 GPixel operations per second (300 MHz core clock, 8 pixel pipelines using instruction co-issuing) with an peak memory bandwidth of 4 GB/s (500 MHz bus clock, 128 bit bus bandwidth, 64 bits per memory access). As we see from the performance figures, programs that perform more computations relative to the number of texture accesses per pixel perform very well. For example, the skin detection algorithm is able to reach 80% of the computational peak performance.

However, programs that perform many texture accesses per computed pixel quickly become bounded by the available memory bandwidth. This is particularly true for the convolution filters that achieve 75% and 85% bandwidth utilization, but with only 11% and 13% computational efficiency, for the  $3\times 3$  and  $7\times 7$  case, respectively.

This figures indicate that the key limiting factor in many GPU programs is memory bandwidth. At present, PyGPU is not optimized for minimizing bandwidth consumption. For example, all computations are carried out on 32-bit floating point 4-tuples, which means that both gray scale and binary images are treated as full four channel RGBA images. By using more compact storage formats, as well as reducing the precision to 16-bits where possible, the bandwidth requirements will be reduced and performance increased further.

Table 1. Performance figures for some of the examples

	No. pixel ops	No. texture accesses	Gpixel ops./s	Texture reads (GB/s)
Convolve ( $3\times 3$ )	27	9	0.56	3.0
Convolve ( $7\times 7$ )	151	49	0.65	3.4
Skin detection	57	1	3.9	1.1
Anisotropic diffusion	43	10	0.58	2.1
Laplace solver	18	4	0.60	2.8

## 4. COMPILER IMPLEMENTATION

The PyGPU compiler is implemented in Python and it is responsible for two major tasks: compiling PyGPU functions to programs running on the GPU, and providing the necessary glue-code allowing these programs to be called as ordinary Python functions. The implementation of the latter is straightforward and will not be covered. The implementation of the translation from Python functions to GPU programs is the focus of this section.

### 4.1 Related work

PyGPU lies at the intersection of two problem areas related to compilation: dynamic languages and embedded languages. It shares a number of problems from both areas all of which must be overcome to allow effective compilation. Furthermore, the restrictions of the target platform greatly affect implementation choices.

#### 4.1.1 Compiling dynamic languages

Compiling dynamic languages is, in general, a very difficult problem. Most of what we know from static languages cease being true: function implementations can be changed at run-time,

arbitrary code can be executed via `eval`, and classes can be dynamically constructed or changed. One approach is to restrict the dynamism of the language. This is used in PyPy [PyPy] and Starkiller [Salib, 2004], two projects targeted at compiling Python. Both projects perform static analysis such as type inferencing to translate general Python code into lower-level compilable code.

Alternatively, the dynamism can be kept by performing *run-time specialization* to compile functions at call-time. This is the approach taken by Psyco [Rigo, 2004], a just-in-time compiler for Python.

#### 4.1.2 Compiling embedded languages

By construction, embedded languages can typically be compiled by the host language compiler. The problem with compiling embedded language however is that they typically target a *different* platform than that supported by the host language. Some examples of such platforms are co-processors [McCool et al, 2002], VHDL designs [Bjesse, 1998], and midi sequencers [Hudak et al, 1996].

The most direct approach for implementing an embedded language compiler is to view the host language merely as syntax for the embedded language. A traditional compiler can then be implemented by reusing the front-end for the host language and implementing a new back end. Such *translative* methods work well when the features of the embedded language closely match the capabilities of the target platform. In such cases translative methods can be implemented fairly directly.

An alternate approach is to use the overloading capabilities of the host language. By implementing a suitable set of abstractions it is possible to *execute* a program in the embedded language in such a way that it generates a program on the target platform. These types of *generative* methods are typically straightforward to implement since much of the existing compiler infrastructure is reused. They are however restricted to translating only those features of the host language that can be overloaded. Conditionals, loops, and function calls, for instance, cannot be overloaded in most languages and consequently cannot be translated using this approach. Examples of projects using a generative approach are Pan [Elliott et al, 2000], Vertigo [Elliott, 2004], and Sh [McCool et al, 2002]. Pan and Vertigo are Haskell domain-specific embedded languages for writing Photoshop plugins and vertex shaders, respectively. Both use a tree-representation constructed at run-time to generate code for their respective platforms. Sh is a GPU programming language embedded in C++ that uses overloading to record the operations performed by a Sh program. This "retained" operation sequence is then analyzed and compiled to native GPU code. We will use a combined approach giving the benefits of both these methods.

## 4.2 Combining translation and generation

Given that we use Python as host language for PyGPU we are faced with a difficult decision. The restrictions of the GPU make direct translation of features such as lists and generators impossible, requiring either restricting the languages or implementing advanced compiler transformations. Using a generative method we are required to supply our own conditionals and loop-construct thereby sacrificing the syntactic brevity of our host language. Ideally one would like to use a translative approach for those features that admit direct translation and a generative approach for those that do not.

We propose that this can be achieved by combining two features commonly found in dynamic high-level languages: *introspection* and *dynamic code execution*. Introspection is the ability of a program to access and, in some cases, modify its own structure at run-time. Dynamic code execution allows a running program to invoke arbitrary code at run-time. For instance, we can use the introspective ability of Python to access the bytecode of a function, where elements such as loops and conditionals are directly represented. This allows using a translative approach where possible. Using dynamic code execution we can reuse large parts of the standard Python interpreter to thereby giving the benefits of translative methods.

### 4.3 The compilation process

As explained above (see Section 2) PyGPU requires that types of all free variables are known at compile-time. Parameter which cannot be given a type must be supplied by value. Hence, for every parameter of a function we know either its type or its value. The compilation strategy thus becomes: if the value is known we evaluate generatively, if only the type is known we perform translation.

The compiler is implemented in the usual three stages: front end, intermediate code generation, and back end. The intermediate code generation and back end stages are implemented using well-known compiler techniques. We use static single-assignment (SSA) [Cytron et al, 1991] for representing the intermediate code. This enables many standard compiler optimizations, such as dead-code elimination and copy propagation, to be implemented effectively. The optimized SSA code is then passed to a back end native code generator. At the moment we use Cg [Mark et al, 2003] as a primary code generation target allowing optimizations of that compiler to be reused.

The front end however, differs from the standard method of implementing a compiler. Instead of using text source code it operates directly on a bytecode representation and it is the front end that implements the above compilation strategy. How this is implemented using the dynamic code execution features of Python will now be described in detail.

The front end parses the stack-based bytecode of Python and translates it to a *flow-graph* which is passed to the intermediate code generator. Throughout this process the types of all variables are tracked allowing the compiler to check for illegal uses as well as performing dispatch of overloaded operations.

Simple opcodes, such as binary operations, are translated directly. More complicated examples such as function calls, that would not be translatable using a generative approach, are handled using the above strategy:

```
elif opcode == CALL_FUNCTION:
    args = stack.popN(oparg)
    func = stack.pop()
    if isValue(args):
        stack.push(func(*args))
    else:
        compiledF = compileFunc(func, args)
        result = currentBlock.CALL(compiledF, args)
        stack.push(result)
```

That is, if all the arguments are values then the function is evaluated directly in the standard interpreter. This is done by using the dynamic code execution abilities of the standard interpreter to call the function via `func(*args)`. This allows the PyGPU compiler to reuse functionality present in external libraries (even compiled ones) generatively. Note that, in

general this kind of constant-folding of function calls is not permitted. The function being called may depend on global values whose value may change between invocations. But, since the GPU lacks global variables PyGPU does not allow global values to be changed after a function has been compiled and consequently this transformation is valid.

If the value of at least one argument is not known then the callee is compiled and a corresponding `CALL`-opcode is added to the current block of the flow-graph. This strategy is not restricted to the case of function calls, it can be used to handle loops as well. Consider the fragment

```
for i in range(n):
    acc += g(i)
```

If `n` is known at compile-time then we may evaluate `range(n)`. Consequently the sequence being iterated over is known and the loop can be trivially unrolled. If `n` is not known the fragment is translated to an equivalent loop in the GPU. The code for handling loops is similar to that of handling function calls albeit slightly more complicated.

## 4.4 An illustrative example

The compilation strategy presented above is very straightforward and it is not obvious how this strategy enables us to translate more complicated examples. Consider the implementation of the `convolve` function used in Section 2.1:

```
def convolve(kernel, im=Image, p=Position):
    return sum([w*im(p+d)
               for w,d in zip(ravel(kernel),
                             offsets(kernel))])
```

The implementation reads: to compute the convolution we first compute the column-first linearization of the kernel using the function `ravel`. The offset to each kernel element is computed and each offset is associated with its corresponding kernel element. The image is accessed at the corresponding locations and the intensities are weighted by the kernel element. Finally the resulting list of intensities is summed and the result returned.

Note that here we use a number of features which cannot be directly translated to the GPU: the compiled Numpy [Numerical Python] function `ravel`, list-comprehensions, and the built-in Python functions `zip` and `sum` both which operates on lists. However, using the above strategy compilation proceeds as follows: The value of `kernel` must be known at compile-time and, consequently, the values of `ravel(kernel)` and `offsets(kernel)` can be computed. Hence the arguments to `zip` are known which implies that it may, in turn, be evaluated at compile-time. The resulting list is used to unroll the list-comprehension resulting in a known number of image accesses which can be directly translated to the GPU. The code for summing these accesses and returning is generated similarly thereby concluding the translation of the above function.

## 5. DISCUSSION

As we have seen the PyGPU language combines high-level programmability with high performance. Being embedded in Python allows functions running on the GPU to be called transparently from Python, greatly facilitating integration of GPU algorithms in larger

applications. Furthermore, since PyGPU functions are, at the same time, valid Python functions GPU programs can be tested on the CPU before being run on the GPU. This allows standard debugging and testing tools to be used for GPU programs also, reducing the need for more specialized GPU debugging tools [Duca et al, 2005].

The performance of the GPU comes from it having a pipelined, highly parallel architecture. This introduces a number of restrictions on what kinds of operations are possible to implement on the GPU. It lacks writable memory. Memory is read only and may only be accessed only in the form of textures containing up to 4-tuples of floating point values. This means that Python features such as lists and objects cannot be used directly on the GPU. But, as we have seen, they may still be used to construct programs.

Also, GPU programs can only write output to a predetermined image location. This means that GPU algorithms must be, using the terminology of parallel computation, written using a *gather*, rather than *scatter*, approach. This restriction is encoded in PyGPU's image model, where algorithms are expressed in a point-wise manner using only gather operations. This is also the reason why the general `reduce` operator, used to do summation for example, is implemented as a iteration over a sequence of progressively smaller images, rather than using a straightforward accumulation loop.

This lack of scatter support sometimes creates difficulties. One such problematic example is computing histograms. This operation is traditionally implemented as a loop over all pixels, having time-complexity linear in the number of pixels. Since the GPU does not support for scattered writes it must instead be implemented as a reduction

```
histogram = reduce(countBins, toBins(im))(0,0)
```

where `toBins` sorts pixels to their respective bins and `countBins` count the number of occurrences in each bin. GPUs only support outputting a limited number of values per pixel, currently 16 floating point values. With a larger number of bins than this the algorithm must be run multiple times resulting in a time-complexity on the order of the number of pixels times the number of bins. This illustrates that not all kinds of image processing algorithms are suitable for the GPU.

## 5.1 Related work

PyGPU was inspired by Pan written by Elliott, Finne, and de Moor [Elliott, Finne, and de Moor, 2000], which is an domain-specific language for image synthesis, embedded in the functional language Haskell [Jones, 2003]. In particular, the functional image model of PyGPU is very similar to that of Pan, but where Pan uses a smooth model, PyGPU focuses on a discrete formulation that allows easier pixel-wise addressing for operations such as convolutions etc.

Other domain-specific languages for using the GPU as a computational co-processor have been proposed. For example, BrookGPU [Buck et al., 2004] is a compiler for writing numerical GPU algorithms in the Brook streaming language, an extension of ANSI C that incorporates *streams* and *kernels*, representing data and operations on data, respectively. The stream and kernel primitives can be mapped to efficient programs running on the GPU. Also, Sh [McCool et al, 2004], for instance, uses C++ templates to provide stream processing abstractions similar to those of Brook. These two projects are based on C and C++, respectively. By using Python, PyGPU is able provide higher-level facilities for writing GPU image processing algorithms than currently possible with these approaches.

Other projects are also targeted at using the GPU as a co-processor for image processing and computer vision. Perhaps the most important among these is [OpenVIDIA], a GPU-accelerated computer vision library. OpenVIDIA provides GPU-implementations of a number of important computer vision algorithms, including Canny edge detection, skin tone tracking, and image compositing. Compared to PyGPU, OpenVIDIA provides only a fixed number of algorithms whereas PyGPU provides a complete, high-level language for implementing many different image processing/computer vision algorithms.

## 5.2 Future work

The current syntax of PyGPU requires the programmer to clearly make the distinction between the parts of the code that should execute on the GPU and the parts that should execute on the CPU. A nice feature would be to have the compiler be able to do this allocation by itself. Apart from relieving the responsibilities of the programmer, it would also allow the compiler to perform more optimizations, both on for storage requirements and also load-balancing.

Also, in order to translate a Python function to the GPU, PyGPU's compiler must know the types of the function parameters. Currently, this information must be provided by the programmer. An interesting improvement would be to remove this requirement and instead have the compiler automatically infer the necessary type information. While PyGPU was initially intended as a language for image processing, it would be usable in other areas as well. Extending the scope of PyGPU to more mathematically oriented applications is an interesting future improvement. For example, both the snake and seamless cloning algorithms presented above, have more general mathematical applications.

## 6. SUMMARY

We have presented PyGPU, a language for image processing on the GPU embedded in Python. The functional programming model used by PyGPU allows algorithms to be translated to efficient code running on the GPU, while still retaining the high-level language features allowing them to be implemented concisely and clearly. The performance of PyGPU is good, allowing many algorithms to be run on real-time streaming video sequences without need for special optimization. This enables the implementer to receive rapid feedback during algorithm development and debugging.

Also, by using language embedding the high-level benefits of Python are transferred onto PyGPU, allowing features such as list comprehensions and higher-order functions to be used in the construction of image processing algorithms. By writing at a higher level of abstraction the code is easier to read and understand. Furthermore, constructing more complex algorithms from simpler building blocks facilitates error detection, making algorithm development and implementation faster and easier.

## REFERENCES

- Amavasai, B., Mimas toolkit, <http://www.shu.ac.uk/mmvl/research/mimas/>.
- Bjesse, P. et al., Lava: hardware design in haskell. In ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming, pages 174-184, New York, NY, USA, 1998. ACM Press.

- Buck, I. et al, 2004. Brook for GPUs: stream computing on graphics hardware, *ACM Trans. Graph.*, Vol. 23, No. 3. pp. 777–786.
- Cytron. R. et al, 1991. *Efficiently computing static single assignment form and the control dependence graph*. ACM Transactions on Programming Languages and Systems, 13(4):451–490, October.
- Czarnecki, K. and Eisenecker, U.W., 2000, *Generative programming: methods, tools, and applications*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.
- Duca, N., et al, 2005. A relational debugging engine for the graphics pipeline, *ACM Trans. Graph.*, Vol. 24, No. 3, pp. 453–463.
- Elliott, C., Finne, S. and de Moor, O., 2000. Compiling embedded languages, *SAIG '00: Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation*, Springer-Verlag, London, UK, pp. 9–27.
- Elliott, C., *Programming graphics processors functionally*. In Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell, pages 45–56, New York, NY, USA, 2004. ACM Press.
- Fung, J. and Mann, S., 2005. OpenVIDIA: parallel GPU computer vision, *MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia*, ACM Press, New York, NY, USA, pp. 849–852.
- Gray, K., 2003. *DirectX 9 programmable graphics pipeline*, Microsoft Press.
- Hudak, P., 1996. *Building domain-specific embedded languages*, ACM Comput. Surv., No. 28, Vol. 4es: 196.
- Hudak, P. et al, 1996, *Haskore music notation - an algebra of music*. Journal of Functional Programming, 6(3):465–483, 1996.
- Intel IPP. Intel integrated performance primitives, <http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/>.
- Jones, S. P. (ed.), 2003. *Haskell 98 Language and Libraries*, Cambridge University Press. ISBN: 0521826144.
- Kass, M., Witkin, A. and Terzopolous, D., 1988. *Snakes: Active countour models*, International Journal of Computer Vision, pp. 321–331.
- Kessenich, J., Baldwin, D. and Rost, R., 2003. *The OpenGL shading language*, 3DLabs, Inc Ltd.
- Krüger, J. and Westermann, R., 2003. *Linear algebra operators for GPU implementation of numerical algorithms*, ACM Trans. Graph., No. 22, Vol. 3, pp. 908–916.
- Mark, W. R. et al, 2003. Cg: a system for programming graphics hardware in a C-like language, ACM Trans. Graph. No. 22, Vol. 3, pp. 896–907.
- McCool, M. et al, 2002. Shader metaprogramming. In T. Ertl, W. Heidrich, and M. Doggett, editors, *Graphics Hardware*, pages 1–12.
- McCool, M. et al, 2004. Shader algebra, *In ACM Trans. Graph.* No. 23, Vol. 3, pp. 787–795.
- Numerical Python. <http://numpy.org>.
- OpenCV. Open source computer vision library, <http://www.intel.com/technology/computing/opencv/>.
- Pérez, P., Gangnet, M. and Blake, A., 2003. Poisson image editing, *In ACM Trans. Graph.* No. 22, Vol. 3, pp. 313–318.
- Perona, P. and Malik, J., 1990. Scale-space and edge detection using anisotropic diffusion, *In IEEE Transactions on Pattern Analysis and Machine Intelligenc*, No. 12, Vol. 7, pp. 629–639.
- PyPy – An implementation of Python in Python. <http://codespeak.net/pypy>.
- Rigo, A., Representation-based just-in-time specialization and the Psycho prototype in Python, *In PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pp.15–26, New York, NY, USA, 2004. ACM Press.
- Salib, M., Starkiller: a static type inferencer in Python. *In Proceedings of the Europython conference*, 2004.
- Shreiner, D. et al. 2003. *OpenGL programming guide*, 4th edn, Addison-Wesley Professional.