PyGPU: A high-level language for high-speed image processing

Calle Lejdfors Lennart Ohlsson
Dept. of Computer Science
Lund University
{calle.lejdfors | lennart.ohlsson}@cs.lth.se

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 runtime 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 high performance.

Keywords: Image processing, Image analysis, GPGPU, GPU, Embedded languages, 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 spotting bugs 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 [1] is a popular choice of high-level language. MATLAB is based around 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 (IPP) [2], OpenCV [3], or Mimas [4], 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 [5]. Programming the GPU is done using specialized languages such as NVIDIA's Cg [6], Microsoft's HLSL [7], or GLSL by the OpenGL ARB [8].

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* [9] in the high-level, object-oriented language Python [10]. PyGPU using 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 0.5–4 GFLOPS without optimizations even on mid-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. Section 4 contains an overview and discussion of PyGPU and how the restrictions and capabilities of the GPU affect how algorithms are implemented. In Section 5 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 [11].

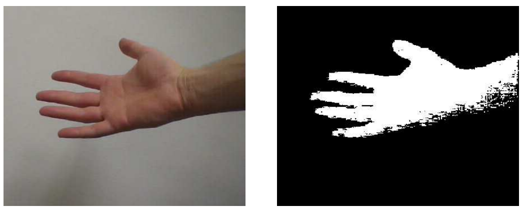
Figure 1: Skin detection

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 Figure 1.

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 outerproduct 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 Numarray, an established high performance Python array programming library implemented in C [12]. 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 [13] to partially evaluate the code at compilation time [11].

In addition to allowing the use of third-party 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 Numarray 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 [14] 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))
```

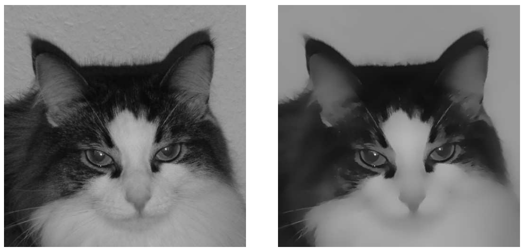
Figure 2: Perona-Malik anisotropic diffusion.

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 here is one of the functions 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 as shown below:

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

This results in successively more smoothed versions of the original image. Figure 2 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

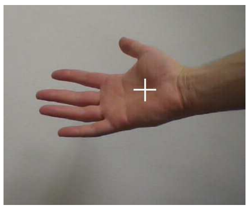
Figure 3: Center of mass

an iterative algorithm similar to the example in the previous section. Its implementation will be shown in Section 2.6.

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
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 in Figure 3.

2.4 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 in [15]. 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 derivates 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.

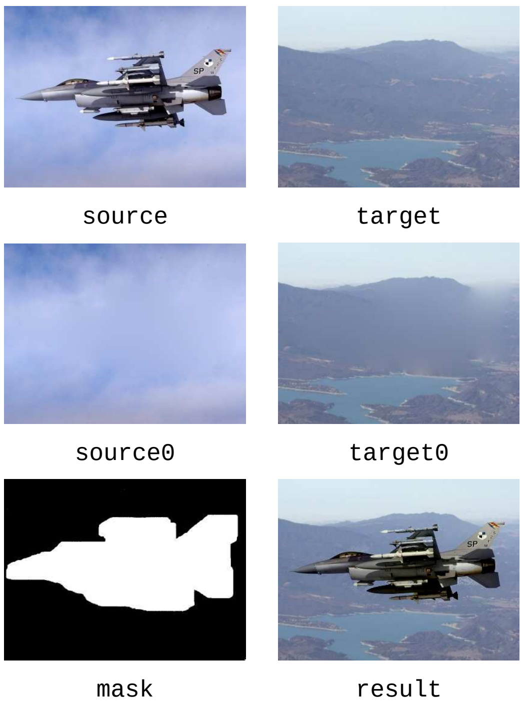

Figure 4: Seamless cloning

```
@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 leave 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 4.

The Laplace solver above will eventually reach a solution, but it converges very slowly. For the example in Figure 4 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 *multigrid* 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 2.6.

2.5 *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 [16] 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):
```

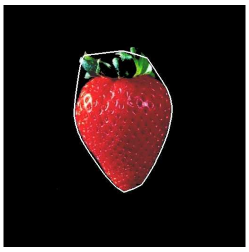
Figure 5: Contour detection using the snake algorithm

```
    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 in Figure 5.

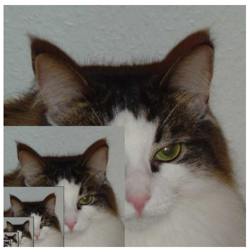
Figure 6: Block-wise reductions

2.6 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 1×1 it will contain the sought quantity as illustrated in Figure 6. 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 2×2 blocks of the image `im` until it is reduced to a single 1×1 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

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 GFLOPS) on the examples shown in this paper. This means that a 9-tap convolution filter can be applied to a 500×500 RGBA color image in about 13 ms. The examples were run on a NVIDIA GeForce 6600 graphics card, a lower mid-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 3×3 and 7×7 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. These improvements, as well as trying to locate other bottlenecks in the processing pipeline, are things which will be incorporated in future versions of PyGPU.

4 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 [17].

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 be used to construct programs. For information on how this is achieved, see [11].

Also, GPU programs can only write output to a predetermined image location. This means that GPU algorithms must be, using the termi-

	No. pixel ops.	No. texture accesses	Gpixel ops./s	Texture reads (GB/s)
Convolve (3×3)	27	9	0.56	3.0
Convolve (7×7)	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

Table 1: Performance figures for some of the examples

nology 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.

4.1 Related work

PyGPU was inspired by Pan written by Elliott *et al.* [18], which is an domain-specific language for image synthesis embedded in the function language Haskell [19]. 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 by Buck et al. [20] 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 by McCool *et al.* [21], 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.

4.2 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 executon 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.

5 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 implementor to receive rapid feed-back 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

[1] Matlab. `http://www.mathworks.com/`.

[2] Intel integrated performance primitives. `http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/`.

[3] Open source compter vision library. `http://www.intel.com/technology/computing/opencv/`.

[4] Bala Amavasai. Mimas toolkit. `http://www.shu.ac.uk/mmvl/research/mimas/`.

[5] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Trans. Graph.*, 22(3):908–916, 2003.

[6] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003.

[7] Kris Gray. *DirectX 9 programmable graphics pipeline*. Microsoft Press, 2003.

[8] John Kessenich, David Baldwin, and Randi Rost. The opengl shading language. `http://developer.3dlabs.com/documents/index.htm`. 3DLabs, Inc Ltd.

[9] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996.

[10] The Python language. `http://www.python.org/`.

[11] Calle Lejdfors and Lennart Ohlsson. Implementing an embedded gpu language by combining translation and generation. *To appear in SAC'06 Programming Language track*, 2006.

[12] Perry Greenfield, Jay Todd Miller, Jin chung Hsu, and Richard L. White. numarray: A new scientific array package for python. PyCon DC 2003, March 2003.

[13] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[14] Pietro Perona and Jitendra Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(7):629–639, July 1990.

[15] Patrick Pérez, Michel Gangnet, and Andrew Blake. Poisson image editing. *ACM Trans. Graph.*, 22(3):313–318, 2003.

[16] Michael Kass, Andrew Witkin, and Demetri Terzopolous. Snakes: Active countour models. *International Journal of Computer Vision*, pages 321–331, 1988.

[17] Nathaniel Duca, Krzysztof Niski, Jonathan Bilodeau, Matthew Bolitho, Yuan Chen,

and Jonathan Cohen. A relational debugging engine for the graphics pipeline. *ACM Trans. Graph.*, 24(3):453–463, 2005.

[18] Conal Elliott, Sigbjorn Finne, and Oege de Moor. Compiling embedded languages. In *SAIG '00: Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 9–27, London, UK, 2000. Springer-Verlag.

[19] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries*. Cambridge University Press, April 2003. ISBN: 0521826144.

[20] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[21] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. *ACM Trans. Graph.*, 23(3):787–795, 2004.