Implementing an embedded GPU language by combining translation and generation

Calle Lejdfors
calle.lejdfors@cs.lth.se

Lennart Ohlsson
lennart.ohlsson@cs.lth.se

Department of Computer Science
Lund University, Lund, Sweden

ABSTRACT

Dynamic languages typically allow programs to be written at a very high level of abstraction. But their dynamic nature makes it very hard to compile such languages, meaning that a price has to be paid in terms of performance. However under certain restricted conditions compilation is possible. In this paper we describe how a domain specific language for image processing in Python can be compiled for execution on high speed graphics processing units. Previous work on similar problems have used either translative or generative compilation methods, each of which has its limitations. We propose a strategy which combine these two methods thereby achieving the benefits of both.

Categories and Subject Descriptors

D.3.4 [**Programming languages**]: Processors—*Compilers, Code generation, Interpreters*; I.4 [**Image processing and computer vision**]: Miscellaneous; D.2.11 [**Software engineering**]: Software architectures—*Domain-specific architectures, Languages*

Keywords

GPU, image processing, dynamic languages, generative techniques, compilation

1. INTRODUCTION

In this paper we introduce PyGPU, a domain-specific language for writing image processing algorithms embedded in the interpreted, object-oriented, dynamically typed language Python. The PyGPU language consists of a number of classes that allow image processing algorithms to expressed clearly and succinctly. These classes use overloading to provide operations such as multiplying a color by a scalar, and accessing an image, with intuitive semantics. For instance, a function for multiplying every pixel of an image by a scalar can be implemented as:

```
def scalarMul(c=Float, im=Image, p=Position):
    return c*im(p)
```

Furthermore, this function can be compiled to native code executing at very high speeds on the graphics processing unit (GPU). However, as will be described below, the GPU is a very restricted platform, and in order to compile a function type-annotations, as in the above example, are required. The types used are exactly the above classes which here serve the alternate purpose of encoding the restrictions and capabilities of the GPU.

The outline of the rest of this paper is as follows. In Section 2 we introduce the graphics processing unit (GPU) as well as a more extensive example of using PyGPU. In Section 3 we present the implementation of PyGPUs compiler and in Section 4 we finish up with a discussion.

2. GPUS

Most computers come equipped with a powerful 3D graphics card capable of transforming, shading, and rasterizing polygons at speeds in excess of those provided by the CPU alone. These cards are also equipped with a programmable graphics processing unit (GPU) that enable parts of the polygon rasterization process to be programmatically redefined allowing, for instance, many image processing algorithms to be implemented. And, since floating-point performance of the GPU is typically an order of magnitude higher than that of a corresponding CPU [9] it is a very attractive target platform.

The speed advantage of GPUs comes from their highly specific nature; they employ a number of very long pipelines executing in parallel and in order to efficiently use this parallelism the computational model of the GPU is very restricted. There is no dynamic memory allocation. Memory is read-only and may only be accessed in the form of textures containing 4-dimensional floating-point values. Furthermore, flow control structures such as branches and subroutines are not guaranteed to exist even on very modern cards.

2.1 Existing GPU languages

There are a number of specialized language available for programming the GPU. The first generations of GPUs provided limited forms of programmability trough assembler-like languages specific to each vendor and graphics API. With the increased power and maturity of GPUs a number of higher level languages were introduced: Cg by NVIDIA [11], HLSL by Microsoft [6], and GLSL by the OpenGL ARB

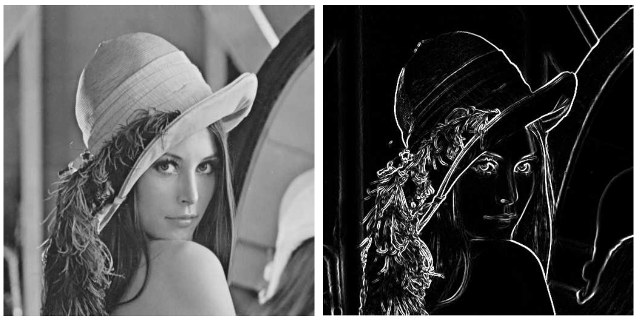
Figure 1: Sobel edge detected lena

[1]. These languages are all syntactic variants of C with some added constructs and data-types suitable for expressing GPU programs.

Other projects aimed at providing embedded languages for programming the GPU are Vertigo [4] and Sh [12]. Vertigo uses Haskell [8] to program the vertex shader functionality of GPUs. Sh is embedded in C++ and uses a generative model for constructing GPU programs at run-time from specification written in C++. Sh also supports, through the use of C++ templates, combining GPU program fragments into new GPU programs [13].

2.2 An image processing example

We will now provide an extended example of PyGPU by implementing an edge detection algorithm. We will construct a general edge detector which will then be used to implementing the well known Sobel edge detector.

2.2.1 *Edge detection in general*

Edge detection is the process whereby sharp gradients in image intensity are identified. In general, this can be implemented as the application of convolution kernels estimating the image intensity gradient in the x and y-directions, respectively. Consider the following function definition:

```
def edgeDetect(kernel, im=Image, p=Position):
    Gx = convolve(kernel, im, p)
    Gy = convolve(transpose(kernel), im, p)
    return sqrt(Gx**2 + Gy**2)
```

This function applies an arbitrary kernel in the x and y-directions (by symmetry the vertical gradient approximation kernel is the transpose of horizontal approximation kernel) and then computes the magnitude of the image gradient.

2.2.2 *Gradient approximation kernels*

There are many examples of gradient approximation kernels. One common choice is the Sobel operator which can be represented by the matrices

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

for the x and y-directions, respectively.

2.2.3 *Complete edge detector*

Using the general edge detection function and the kernel from the previous section we can now create a Sobel edge detector by partially specializing the general edge detection function. To do this we call the PyGPU compiler passing the kernel as a compile-time parameter:

```
sobelEdgeDetGPU = pygpu.compile(edgeDetect,
                                kernel=sobelKernel)
```

The function returned by the compiler runs entirely on the GPU and can be applied to images just as a normal function. However, the position parameter need not be specified, the returned function operates on whole images in parallel:

```
edgesLena = sobelEdgeDetGPU(lena)
```

The result of applying the Sobel edge detector to the standard Lena example image can be seen in Figure 1.

2.2.4 *Example discussion*

Interestingly, the general edge detection function presented above can not be translated to native code on the GPU and the reason lies in the kernel argument. Because the GPU lacks support for dynamic memory allocation translating the kernel argument to the GPU is impossible. PyGPU expresses this by the fact that the kernel argument cannot be given a type in PyGPUs type system. And, since these types encode the capabilities of the GPU, an argument which cannot be typed must be supplied as a value at compile-time.

As a consequence we are allowed to use external libraries even when these libraries cannot be translated to the GPU. For instance, the `transpose` function is taken directly from Python Numeric, an array programming library implemented in C and running on the CPU[14]. Clearly, this function cannot be directly translated but, since the value of `kernel` must be supplied at compile-time, it may still be used to construct PyGPU functions.

3. 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.

3.1 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 affects implementation choices.

3.1.1 *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 [15] and Starkiller [17], 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 [16], a just-in-time compiler for Python.

3.1.2 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 [12], VHDL designs [2], and midi sequencers [7].

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 translated using this approach. Examples of projects using a generative approach are Pan [5], Vertigo [4], and Sh [12]. 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.

3.2 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 makes 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 executing 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.

3.3 The compilation process

As explained above (see Section 2.2.4) 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) [3] for representing the intermediate code. This enables many standard compiler optimization, 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 [11] 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.

3.3.1 Bytecode translation

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 globals 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.

3.4 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 2.2:

```
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 Numeric [14]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.

4. DISCUSSION

We have shown how a compiler for an embedded language can be implemented to combine the advantages of previous methods. By taking advantage of introspection and dynamic code execution features of the host language Python we could implement this compiler very compactly. As an example, The compiler and run-time consists of around 1500 lines of code with the bytecode to flow-graph translator occupying 400 of those lines. By compiling for the GPU, performance in excess to that of optimized CPU code can be obtained. Furthermore, the relative increase in speed for new GPU generations is greater than the corresponding increase for CPUs making the GPU a very attractive platform.

A recent area of research is using the GPU for general purpose computations. Examples of algorithms which have been implemented on the GPU are fluid simulations [10], linear algebra [9], and signal processing [18]. Future work includes extending the PyGPU compiler to allow programming all aspects of the GPU including general purpose numerical algorithms. Also, the presented method ought to be suitable for compiling embedded languages to other platforms such as ordinary CPUs.

Another interesting area for future research is studying how the approach used here integrates with that of PyPy [15]. Of particular interest is reusing parts of the PyPy framework to be able to handle more general examples, including the above general purpose uses of the GPU as well as targeting other platforms.

5. REFERENCES

[1] 3D Labs. *OpenGL 2.0 Shading Language White Paper*, 1.2 edition, February 2002.

[2] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. 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.

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[4] C. Elliott. 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.

[5] C. Elliott, S. Finne, and O. 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.

[6] K. Gray. *DirectX 9 programmable graphics pipeline.* Microsoft Press, 2003.

[7] P. Hudak, T. Makucevich, S. Gadde, and B. Whong. Haskore music notation - an algebra of music. *Journal of Functional Programming*, 6(3):465–483, 1996.

[8] S. P. Jones, editor. *Haskell 98 Language and Libraries.* Cambridge University Press, April 2003. ISBN: 0521826144.

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

[10] Y. Liu, X. Liu, and E. Wu. Real-time 3d fluid simulation on gpu with complex obstacles. In *Proceedings of Pacific Graphics 2004*, pages 247–256, October 2004.

[11] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003.

[12] M. McCool, Z. Qin, and T. Popa. Shader metaprogramming. In T. Ertl, W. Heidrich, and M. Doggett, editors, *Graphics Hardware*, pages 1–12, 2002.

[13] M. McCool, S. D. Toit, T. Popa, B. Chan, and K. Moule. Shader algebra. *ACM Trans. Graph.*, 23(3):787–795, 2004.

[14] Numerical python. http://numpy.org.

[15] Pypy - an implementation of python in python. http://codespeak.net/pypy/.

[16] A. Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26, New York, NY, USA, 2004. ACM Press.

[17] M. Salib. Starkiller: a static type inferencer for python. In *Proceedings of the Europython conference*, 2004.

[18] S. Whalen. Audio and the graphics processing unit. `www.node99.org/projects/gpuaudio/gpuaudio.pdf`, 2005.