SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization

9 authors, including:

Bryan C. Catanzaro
NVIDIA

Yunsup Lee
University of California, Berkeley

John Shalf
Lawrence Berkeley National Laboratory

Katherine Yelick
University of California at Berkeley and Lawr…

SEJITS: Getting Productivity ***and*** Performance With Selective Embedded JIT Specialization

Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, Krste Asanovic, James Demmel,
Kurt Keutzer, John Shalf*, Kathy Yelick, Armando Fox
UC Berkeley Parallel Computing Laboratory and *Lawrence Berkeley Laboratory

Parallel programming must be accessible to domain experts without requiring them to become experts in parallel hardware architecture. While domain experts today prefer to use high-level "productivity" scripting languages with domain-appropriate abstractions, achieving high performance still requires expertise in lower-level "efficiency" languages (CUDA, CILK, C with OpenMP) that expose hardware-level programming models directly. We bridge this gap through the use of *embedded just-in-time specialization*: domain experts write in high-level scripting languages, but at runtime, we *specialize* (generate, compile, and execute efficiency-language source code for) an application-specific and platform-specific subset of the productivity language. This enables invisible and *selective* optimization of only those application-level abstractions that enjoy a large performance advantage when expressed in an efficiency language on the available hardware and will be executed many times, amortizing the overhead of specialization. Because the specialization machinery is implemented in the productivity language, efficiency programmers can easily extend our system by adding new specializers for specific additional domain abstractions or new hardware, transparently to the productivity-language programmers. Our approach results in competitive performance on real applications with a fraction of the programming effort on the part of the domain expert. We argue that the separation of concerns enabled by embedded JIT specialization allows research to proceed in parallel on both the productivity and efficiency layers, and is therefore uniquely suited to the problem of making different parallel hardware architectures more accessible to domain-expert programmers with a fraction of the programmer time and effort.

1 Motivation

As computational sciences continue to increase in importance, more programming is done by specialists in various fields instead of by expert programmers. In the quest to minimize time-to-solution, these specialists are increasingly turning to very-high-level scripting and domain-specific languages such as Python and MATLAB, which focus on programmer productivity over hardware efficiency. Besides offering abstractions well-matched to the domains, these *productivity-level languages* (PLLs) often provide utilities for debugging and visualization that are compelling to domain experts. While we are not yet aware of large-scale longitudinal studies on the productivity of such languages compared to traditional imperative languages such as C, C++ and Java, individual case studies have found that such languages allow programmers to express the same programs in 3–10x fewer lines of code and in $1/5$ to $1/3$ the development time [19, 4, 8].

However, at the same time, the well documented move towards parallel processing [1] motivates increased focus on low-level computational problems involving specifics of parallel hardware platforms. PLLs usually offer insufficient performance for large problem sizes and are unable to take full advantage of parallel processors such as today's multicore CPUs and manycore Graphics Processors. Consequently, many applications are eventually rewritten in *efficiency-level languages* (ELLs) such as C with parallel extensions (CILK, OpenMP, CUDA). Because ELLs expose hardware-supported programming models directly, they can achieve multiple orders of magnitude higher performance than PLLs on emerging parallel hardware [3]. However, the performance comes at high cost: the abstractions provided by ELLs are a poor match to those used by domain experts, and moving to a different hardware programming model requires rewriting the ELL code, making ELLs a poor medium for exploratory work, debugging and prototyping.

Ideally, domain experts could use high-productivity domain-appropriate abstractions *and* achieve high performance in a single language, without rewriting their code. This is difficult today because of the *implementation gap*

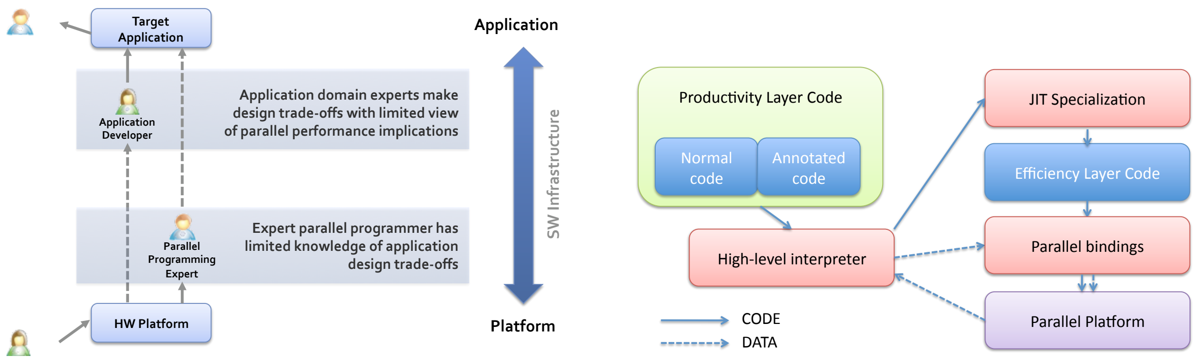

Figure 1: **Left:** Implementation gap between productivity-level languages (PLL) and efficiency-level languages (ELL). **Right:** Specialized embedded just-in-time specialization (SEJITS) schematic workflow.

between hardware targets and high-level domain abstractions: as depicted in Figure 1, PLLs fail to be hardware-efficient, while ELLs fail to be productive.

Not only is the implementation gap already a problem, but it is also getting worse: domains are specializing further (leaving domain experts even less time to become programming experts on multiple platforms), and available target hardware is getting more heterogeneous, with manycore GPUs, hyperthreaded multicore, heterogeneous manycore, and message-passing systems all exposing radically different programming models.

In this paper we observe that the metaprogramming and introspection facilities in modern scripting languages such as Python and Ruby can bridge the gap between the ease of use of PLLs and the high performance of ELLs. Specifically, *just-in-time specialization* of PLL code, which consists of dynamically generating source code in an ELL within the context of a PLL interpreter, allows the application to be written using PLL abstractions but automatically generating and compiling ELL source code at runtime. Unlike most conventional JIT approaches, our JIT specialization machinery is *selective*, allowing us to pay the overhead of runtime specialization only where a significant performance advantage can be realized and leaving the rest of the code in the PLL. It is also *embedded* in the PLL itself, making it easy to extend and add new specializers, while taking advantage of PLL libraries and infrastructure.

We describe our approach in more detail in Section 2. We present two case studies using with early results in Section 3; we test our approach on part of a state-of-the-art problem in computer vision. Section 4 contextualizes the results and lessons from this early work. In Section 5 we place our work in the context of existing work and discuss our future work, concluding and looking forward in Section 6.

2 Making JIT Specialization Selective and Embedded

The key to our approach, as outlined in Figure 1 (right), is *selective embedded just-in-time (JIT) specialization*. The domain programmer expresses her code in a PLL using provided class libraries of domain-appropriate abstractions. Rather than executing computations directly, however, the library functions generate source code at runtime in a lower-level ELL, such as C with parallel extensions. This *specific subset* of the code is then JIT-compiled, cached, dynamically linked, executed via a foreign-function interface (on possibly exotic target hardware), and the results returned to the PLL, all at runtime and under the control of the PLL interpreter. From the domain programmer's view, the process is indistinguishable from doing all computation directly in the PLL, except (ideally) much faster.

SEJITS inherits standard advantages of JIT compilation, such as the ability to tailor generated code for particular argument values or function compositions, or for other characteristics known only at runtime. However, as the name suggests, SEJITS realizes specific benefits for both the PLL and ELL programmer by being selective and embedded.

Selective. A SEJITS specializer targets a particular function or set of functions *and* a particular ELL platform (say, C+OpenMP on a multicore CPU, or CUDA [13] on a GPU). Specialization occurs only for those specific functions, and only if *all* of the following are true: (1) function specializers exist for the target platform, (2) the ELL specialization of the function is much faster than the PLL implementation, (3) the function is likely to be executed many times (e.g. an inner loop), amortizing the one-time overhead of specialization and reducing overall running time. While conventional

JIT compilers such as HotSpot [17] also make runtime decisions about what to specialize, in SEJITS the benefit of specialization is not just avoiding overhead at runtime, but also completely avoiding *any* additional mechanism for nonspecialized code by falling back to the PLL when no appropriate $\langle$function, ELL platform$\rangle$ specializer exists. We can therefore sidestep the difficult question of whether PLL language constructs outside the specialized subset can be JITted efficiently.

The programmer can also explicitly disable all specialization in order to use the PLL's debugger or other features during exploratory work, in which case all computations are performed in the PLL directly.

Embedded. Embedding in a modern PLL helps both the efficiency programmer and the productivity programmer. From the efficiency programmer's point of view, the specialization machinery is implemented in the PLL itself by exploiting modern PLL features, as we describe in Section 4.1. As a result, we avoid rebuilding JIT compiler infrastructure (parser, analyzer, etc.). The effect is that writing new specializers is much easier, and integrating them more seamless, than if the JIT machinery were outside the PLL interpreter. We expect this to enable rapid evolution and adaptation of a collection of specializers for a PLL, invisibly to the productivity programmer.

From the productivity programmer's point of view, modern PLL features such as iterators, abstract classes, and metaprogramming allow the specializable abstractions to look more like language extensions or a mini-embedded-DSL [7] than procedural libraries. In effect, SEJITS makes it possible to provide abstractions that resemble a domain-specific language embedded in Python/Ruby with an efficient implementation for some parts.

3 Case Studies

SEJITS is most easily illustrated by example. We have prototyped two specializers, both of which rely on the introspection features of modern PLLs (Python and Ruby, in our work) to perform specialization. We have used these specializers, in conjunction with Python and Ruby libraries, to test our approach on real problems from high performance computing and computer vision. The problems are noteworthy because the original implementations of the algorithms by domain researchers used productivity languages, but ultimately the algorithms had to be rewritten in efficiency languages to achieve acceptable performance by exploiting parallel-computing hardware.

Both case studies focus on providing high-level abstractions for *stencils*, an important class of nearest-neighbor computations used in signal and image processing and structured-grid algorithms [10]. In a typical stencil kernel,[1] each grid point updates its value based on the values of its nearest neighbors, as defined by the stencil shape. For example, a three-dimensional 7-point stencil examines each point in a 3D grid and computes a new value for that point based on the properties of its 7 nearest neighbors. This operation and its typical corresponding memory access pattern are depicted in Figure 2.

In the first case study, Ruby classes and methods providing stencil abstractions are JIT-specialized to C code annotated with OpenMP pragmas. In the second, Python functions providing the abstractions are JIT-specialized to CUDA [13] code for execution on Nvidia Graphics Processors. In both case studies, the introspection and function-interposition features of the PLLs are used to effect specialization, including using information about the actual arguments at runtime to make the generated code more efficient.

In our early experiments, we focus on the following questions:

- How does the performance and scalability of JIT-specialized code compare to ELL code handcrafted by an expert programmer? This provides an upper bound on how well we can do.
- Which aspects of JIT specialization overhead are fundamental and which can be mitigated by further engineering? This tells us how close we can expect to come to the upper bound.
- How does the approximate programmer effort required to write PLL code compare to the effort required for an expert to code the same functionality in an ELL? This helps quantify the tradeoff between raw performance and programmer productivity, highlighting the fact that "time to solution" is often as important as achievable peak performance.

We first describe how each prototype performs specialization and execution and presents its abstractions to the domain programmer. We then discuss the results for both case studies together.

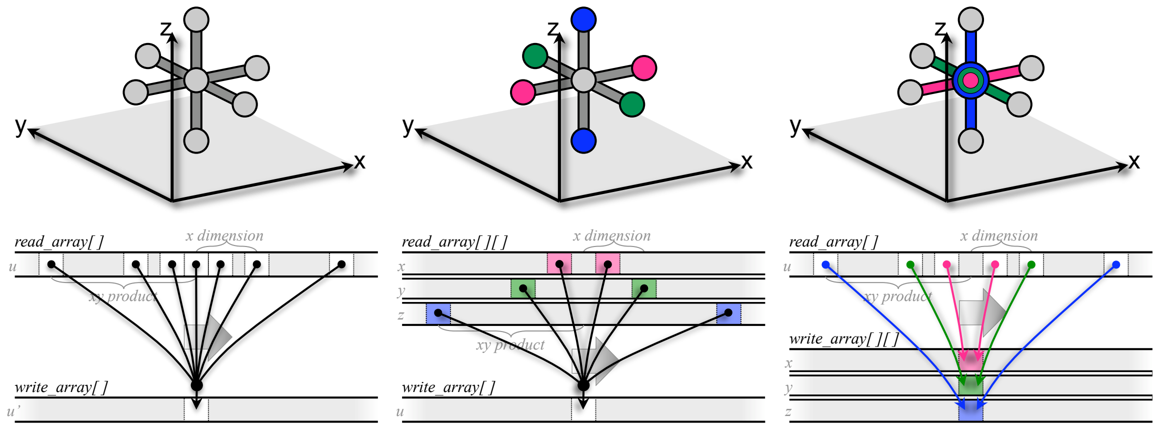

Figure 2: Graphical representation (top) and Memory access pattern (bottom) of the three stencil kernels under study: Laplacian (left), Divergence (middle), and Gradient (right). Figure by Sam Williams.

```
class LaplacianKernel < JacobiKernel
def kernel
<<EOF
  def kernel(in_grid, out_grid)
    in_grid.each_interior do |center|
      in_grid.neighbors(center,1).each do |x|
        out_grid[center] = out_grid[center]
        + 0.2 * in_grid[x]
      end
    end
  end
EOF
end
end
```

```
VALUE kern_par(int argc, VALUE* argv, VALUE self) {
unpack_arrays into in_grid and out_grid;

#pragma omp parallel for default(shared)
private (t_6,t_7,t_8)
for (t_8=1; t_8<256-1; t_8++) {
 for (t_7=1; t_7<256-1; t_7++) {
  for (t_6=1; t_6<256-1; t_6++) {
   int center = INDEX(t_6,t_7,t_8);
   out_grid[center] = (out_grid[center]
      +(0.2*in_grid[INDEX(t_6-1,t_7,t_8)]));
   ...
   out_grid[center] = (out_grid[center]
      +(0.2*in_grid[INDEX(t_6,t_7,t_8+1)]));
;}}}
return Qtrue;}
```

Figure 3: Example of a Laplacian kernel implemented in the Ruby stencil framework. Source in Ruby (top left) is parsed into a tree that gets transformed to contain stencil-specific nodes (bottom left). The code generation phase emits inlined C code (right). Note that the code defining neighbors is not shown.

3.1 Case Study 1: Ruby and OpenMP

Abstractions. Our first case study provides Ruby `JacobiKernel` and `StencilGrid` classes whose methods can be JIT-specialized to C with OpenMP pragmas (annotations) for parallelizing compilers. `StencilGrid` implements an n-dimensional grid as a single flat array indexed based on the actual dimensions of a the grid instance. `JacobiKernel` provides the base class that the programmer subclasses to implement her own stencil kernel; the programmer overrides the `kernel` function[2], which accepts a pair of `StencilGrid` objects, to define the desired stencil computation. As the code excerpt in Figure 3 shows, `StencilGrid` provides a `neighbors` function that returns a point's neighbors based on a user-supplied description of the grid topology (function not shown), and Ruby iterators `each_interior` and `each_border` over the interior and border points of the grid, respectively.

The Ruby programmer must subclass from `JacobiKernel` and use our iterators; other than that, the function to be specialized can contain arbitrary Ruby code as long as any method calls are reentrant.

Specialization. When the user-provided kernel method is called, the `JacobiKernel` instance parses the method's

[1]Computational scientists use the term *kernel* to denote a core or "inner loop" computation that is an essential element of a larger computation.

[2]Currently, the `kernel` method must return the actual body of the kernel as text, hence the $<<$ (here-document) notation in the Ruby code of Figure 3, but this implementation artifact will soon be eliminated.

code using the RubyParser library [22], which returns a symbolic expression (Sexp) representing the parse tree. The parse tree is then walked to generate ELL code using information about the kernel method's arguments (which are instances of `StencilGrid`) to build an efficient parallel C implementation. In our initial implementation, the ELL language is C with OpenMP [15] pragmas that a compiler can use to parallelize the code in a target-architecture-appropriate way. An example Ruby kernel function and the corresponding generated ELL code are shown in Figure 3.

Execution. Using RubyInline [21], the C code is invisibly compiled into a shared object file, dynamically linked to the interpreter, and called using Ruby's well-documented foreign function interface. Since the generated kernel operates on Ruby data structures, there is no overhead for marshalling data in and out of the Ruby interpreter. RubyInline also attempts to avoid unnecessary recompilation by comparing file times and function signatures; the Ruby specializer machinery also performs higher-level caching by comparing the parsed code with a previously cached parse tree to avoid the overhead of ELL code regeneration.

Experiments. We implemented three stencil kernels using the Ruby framework: Laplacian, Divergence, and Gradient. Graphical representations of the three kernels are shown in Figure 2. We ran these on both a 2.6 GHz Intel Nehalem (8 cores, with 2-way SMT for a total of 16 HW threads) and 2.3 GHz AMD Barcelona (8 cores). For comparison, we also ran handcrafted C+OpenMP versions of the three kernels using the StencilProbe [24] microbenchmark. For both implementations, NUMA-aware initialization is used to avoid deleterious NUMA effects resulting from the "first-touch" policy on these machines, whereby memory is allocated at the first core's memory controller. We discuss results of both casestudies together in Section 3.3.

3.2 Case Study 2: Python and CUDA

Abstraction. In our second case study, we provide abstractions in Python and generate ELL code for CUDA [13]. Our `stencil` primitive accepts a list of filter functions and applies each in turn to all elements of an array. A filter function can read any array elements, but cannot modify the array. This constraint allows us to cache the array in various ways, which is important for performance on platforms such as a GPU, where caches must be managed in the ELL code. Our `category-reduce` primitive performs multiple data-dependent reductions across arrays: given an array of values each tagged with one of N unique labels, and a set of N associative reduction operators corresponding to the possible labels, `category-reduce` applies the appropriate reduction operator to each array element. If there is only one label, `category-reduce` behaves like a traditional reduction.

Specialization. Our prototype relies on *function decorators*, a Python construct that allows interception of Python function calls, to trigger specialization. The Python programmer inserts the decorator `@specialize` to annotate the definitions of the function that will call `stencil` and/or `category-reduce` as well as any filter functions passed as arguments to these primitives. The presence of the decorator triggers the specializer to use Python's introspection features to obtain the abstract syntax tree of the decorated function. Decorated functions must be restricted to the embedded subset of Python supported by our specializer. Specifically, since our efficiency layer code is statically typed, we perform type inference based on the dynamic types presented to the runtime and require that all types be resolvable to static types supported by NumPy [14]. Type inference is done by examining the types of the input arguments to the specialized function and propagating that information through the AST. In addition, we must be able to statically unbox function calls, i.e. lower the code to C without the use of function pointers. As development proceeds, we will continue expanding the supported subset of Python. If the specializer can't support a particular Python idiom or fails to resolve types, or if no decorators are provided, execution falls back to pure Python (with an error message if appropriate).

Execution. If all goes well, the specializer generates CUDA code, and arranges to use NumPy [14] to ease conversion of numerical arrays between C and Python and PyCUDA [9] to compile and execute the CUDA code on the GPU under the control of Python. The specializer runtime also takes care of moving data to and from GPU memory, as well as compiling and executing the CUDA code.

Experiments. We used these two primitives to implement three computations that are important parts of the *gPb* (Global Probability of Boundaries) [11] multi-stage contour detection algorithm. *gPb* was an interesting case study for two reasons. First, this algorithm, while complicated, provides the most accurate known image contours on natural images, and so it is more representative of real-world image processing algorithms than simpler examples. Second, the algorithm was prototyped in MATLAB, C++, and Fortran, but rewriting it manually in CUDA resulted in a 100x speedup [3], clearly showing the implementation gap discussed in Section 1.

The stencil computations we implemented correspond to the *colorspace conversion, texton computation,* and *local*

```python
@specialize
def min(a, b):
  if a > b: return b
  else: return a

@specialize
def colMin(array, element, [height, width],
      [y, x]):
  val = element
  if (y > 0):
    val = min(val, array[y-1][x])
  if (y < height-1):
    val = min(val, array[y+1][x])
  return val

@specialize
def kernel(pixels):
  return stencil(pixels, [filter], [])
```

```
__device__ int min(...);
__global__ void colMin(int height,
                   int width,int* dest, float* array)
{
  const int y=blockIdx.y*blockDim.y+threadIdx.y;
  const int x=blockIdx.x*blockDim.x+threadIdx.x;
  int element=array2d[y*width+x];
  int* ret=&dest[y*width+x];

  int val = element;
  if (y > 0) {
    val = min(val, array[(y-1)*width+x]);
  }
  if (y < height - 1) {
    val = min(val, array[(y+1)*width+x]);
  }
  *ret = val;
}
```

Figure 4: Illustration of simple kernel. Source in Python (top) calls the `stencil` primitive with functions decorated with `@specialize`, which then generates CUDA code for functions called inside our parallel primitives.

cues computations of *gPb*. The Python code for local cues, the most complex of the three, requires a total of five stencil filters to extract local contours out of an image: quantize, construct histograms, normalize/smooth histogram, sum histograms, and χ^2 difference of histograms. We show results on two different Nvidia GPUs: the 16-core 9800GX2 and the 30-core Tesla C1060.

The one-time specialization cost indicates the time necessary to compile the PLL into CUDA. The per-call specialization cost indicates time needed to move data between the Python interpreter and the CUDA runtime, and the execute time reflects GPU execution time. Not shown are pure Python results without specialization, which took approximately 1000× slower than our JIT specialized version on the C1060. For simple functions, like the colorspace conversion function, we approach handcoded performance. Our most complex code, the local cue extractor, ran about 4× slower than handcoded CUDA, which we feel is respectable. We also note good parallel scalability as we move to processors with more cores, although it's important to note that some of that scaling came from architectural improvement rather than parallelism.

3.3 Results and Discussion

Performance. Table 1 summarizes our results. For each JIT-specializer combination, we compare the performance of SEJITS code against handcrafted code written by an expert; the *slowdown* column captures this penalty, with 1.0 indicating no performance penalty relative to handcrafted code. Where SEJITS is slower, we report both the fixed overhead (generating and compiling source code) and the per-call overhead of calling the compiled ELL code from the PLL. For example, the second row shows that when running the Laplacian stencil using our Ruby SEJITS framework on the 16-core Nehalem, the running time of 0.614 seconds is 2.8 times as long as the 0.219-second runtime of the handcrafted C code. The same row shows that of the total SEJITS runtime, 0.271 seconds or 44% consists of fixed specialization overhead, including source code generation and compilation; and 0.12 seconds or 20.2% is the total overhead accrued in repeatedly calling the specialized code.

Several aspects of the results are noteworthy. First, the Ruby examples show that it is possible for SEJITS code to achieve runtimes no worse than 3 times slower than handcrafted ELL code. In fact, the Barcelona results show that once specialized, the Laplacian and Gradient kernel performance is not only comparable to handcrafted C, but in some cases *faster* because the JIT-specialized kernels contain hardcoded array bounds while the C version does not. On Nehalem, all kernels are slower in Ruby, due in part to the different code structure of the two in the ELL; as the code generation phase is quite primitive at the moment, a few simple changes to this phase of the JIT could result in much better performance.

The Python examples overall perform substantially worse than Ruby, but a larger percentage of the slowdown is due to specialization overhead. Most of this overhead is coming from the CUDA compiler itself, since in our

Language & computation	CPU, #cores	Hand coded	SEJITS	Slow-down	Spec. overhead	Exec. overhead
Ruby Laplacian	Barcelona, 8	0.740	0.993	1.34	0.25 (25%)	.003 (0.3%)
Ruby Laplacian	Nehalem, 8	0.219	0.614	2.80	0.271 (44%)	0.12 (20.2%)
Ruby Divergence	Barcelona, 8	0.720	0.973	1.35	0.273 (28%)	0 (0%)
Ruby Divergence	Nehalem, 8	0.264	0.669	2.53	0.269 (40%)	0.136 (20.3%)
Ruby Gradient	Barcelona, 8	1.260	1.531	1.22	0.271 (18%)	0 (0%)
Ruby Gradient	Nehalem, 8	0.390	0.936	2.40	0.268 (29%)	0.278 (29.7%)
Python Colorspace	GX2, 16	0.001	0.469	469.00	0.448 (96%)	0.02 (4.3%)
Python Colorspace	C1060, 30	0.001	0.596	596.00	0.577 (97%)	0.018 (3.0%)
Python Textons	GX2, 16	2.294	10.00	4.36	2.462 (25%)	5.249 (52.5%)
Python Textons	C1060, 30	0.477	9.015	18.90	3.397 (38%)	5.141 (57.0%)
Python Localcues	GX2, 16	0.565	5.757	10.19	2.6 (45%)	2.592 (45.0%)
Python Localcues	C1060, 30	0.263	3.323	12.63	2.235 (67%)	0.825 (24.8%)

Table 1: Performance results (times are in seconds) comparing SEJITS vs. handcrafted ELL code showing what fraction of SEJITS execution time is due to SEJITS overhead. See text for detailed discussion of overheads. Ruby results reflect 10 iterations of the stencil ("inner loop").

prototype we specialize functions that may be called very few times. The colorspace conversion example shows this: the execution overhead is less than 0.02 seconds, whereas the specialization overhead is essentially the time required to run the CUDA compiler.

More importantly, our category reduction primitive is currently not optimized, which is why the Texton computation runs about 12× slower with SEJITS than handcoded CUDA. More specifically, The choice of strategy for implementing parallel category reductions on CUDA depends strongly on the parameters of the particular category reduction being computed. If the number of categories is one, the computation is a straightforward reduction and can be handled straightforwardly. If the number of categories is high, the best-performing implementations, which we use in our handcrafted CUDA code, use atomic memory transactions on on-chip memory structures to deal with bin contention efficiently. The size of data being accumulated is also very important: in our case the texton bins consist of a struct of 34 elements each, which has major implications on how intermediate reduction data structures fit in on-chip memory.

All of this information is available to our SEJITS framework at runtime, but due to time constraints, our flexible yet simplistic code generator for category reductions neither exploits this information nor uses fast on-chip (GPU) memory. While we expect to improve our code generator shortly, the broader principle is that specialization allows these details to be encapsulated in library elements, and then instantiated at runtime using efficient code generation.

There are some other overheads that could be mitigated through futher software engineering. In our current prototype, although PyCUDA caches translations of CUDA source to CUDA compiled code, our Python machinery does not (yet) cache translations of function signatures to generated source (corresponding to the higher-level caching of function signatures done by our Ruby prototype, as described in Section 3.1). Hence, the functions to be specialized are re-analyzed (type inference, walk the AST) on every call, which is expensive. Python has the necessary introspection mechanisms to eliminate this overhead and we expect to report substantial improvements in the camera ready version of this paper.

We do not show results for running the PLL-native versions of the computations. Python was about three orders of magnitude slower than handcrafted C, and Ruby about two orders of magnitude slower. This is not surprising, but it emphasizes that SEJITS is much closer to the performance of handwritten code than it is to the performance of the PLL itself.

Programmer effort. All in all, these are useful results for domain programmers. The original rewrite of *gPb* in CUDA [3] took many engineer-months of work by a researcher who is both a domain expert and a CUDA expert. The difficulty lay in using the GPU memory hierarchy properly, partitioning the data correctly, and debugging CUDA code without the high-level debugging tools provided by PLLs. Using our Python SEJITS framework and Python's debugging tools, it took one afternoon to get all three kernels running reasonably fast on the GPU. Similarly, the Ruby stencils took only a couple of hours to write with SEJITS, compared to 3–4 days for OpenMP. Besides consisting of fewer lines of code, the PLL code was developed with the full benefits of the Ruby debugging facilities (interactive command prompt, breakpoint symbolic debugger, etc.) These results encourage us that it is indeed possible to get competitive performance from PLL source code in a programmer-invisible and source-portable manner.

4 Discussion

While these two examples are not sufficient to generalize, we believe SEJITS presents an opportunity for significant leverage. For example, even handling the thirteen computational "motifs" that recur in many applications that would benefit from parallel computing [1] would be a productive step forward. Here we discuss the opportunities and challenges of pursuing such a path.

4.1 Why Now? (Or: PLL Features to Support SEJITS)

In 1998, John Ousterhout [16] made the case for using scripting languages for higher-level programming because they are designed to glue together components in different languages while providing enough functionality to code useful logic in the scripting language itself. In particular, good "glue facilities" include the ability to dynamically link object code created by other compilers, make the entry points available to the scripting language via a foreign function interface, and support translating data structures back and forth across the boundary.

In 1998, the most widespread scripting languages were Tcl and Perl. Tcl was expressly designed as a glue language and succeeds admirably in that regard, but with no type system and few facilities for data abstraction or encapsulation, it is not rich enough to support the domain experts we target. Perl is considerably more powerful, but both it and Tcl fall short in introspection support, which is necessary for the *embedding* aspect of our approach that leads to ease of extensibility. Java supports introspection, but has weak glue facilities and a low level of abstraction—the same program typically requires 3–10x as many lines to express as in typical scripting languages [19]. Popular domain-specific languages like MATLAB and R have weak glue facilities and introspection; while JIT specialization could technically be applied, it could not be embedded, and new specializers would be more difficult to write and integrate, requiring recompiling or relinking the PLL interpreter or runtime for each change.

Modern scripting languages like Python and Ruby finally embody the *combination* of features that enable SEJITS: a high level of abstraction for the programmer, excellent introspection support, *and* good glue facilities. For these reasons, they are ideal vehicles for SEJITS. Although the interception/specialization machinery can be implemented in any language with aspect-oriented support, we exploit specific Python and Ruby features for both ease of extensibility and better performance. In Ruby, when a method is successfully specialized, the instance on which the method was called is converted to a singleton. This allows all fixed overheads associated with specialization to be eliminated on subsequent invocations—the only check necessary is whether the method must be re-specialized because the function signature has changed or because (in our example) the `StencilGrid` arguments have different sizes. We also exploit the fact that Ruby 1.8 exposes the parse tree for any Ruby function, since the Ruby interpreter has to create this data structure anyway; unfortunately, this may disappear in Ruby 1.9. In Python we used the quite general function decorator mechanism to intercept function calls; our current lack of a cache for generated source code results in penalties on subsequent invocations, although Python has the necessary functionality to support such a cache, and we expect to have one implemented and report improved results by camera-ready.

4.2 Benefits to Efficiency Programmers

Although SEJITS clearly benefits productivity programmers, less obvious is the benefit to efficiency programmers, who are often asked to adapt existing code to run efficiently on new hardware. Because the specializer machinery (function call interception, code introspection, orchestration of the compile/link/run cycle, argument marshalling and unmarshalling) is *embedded* in the PLL, an efficiency programmer wishing to create a new specializer for some class method M merely has to determine what to do at each node of the abstract syntax tree of a call to M (a straightforward instance of the Visitor design pattern [6]). Furthermore, this code is written in the PLL, which typically has excellent debugging and prototyping support. This encourages rapid experimentation and prototyping of new specializers as new hardware or ELL platforms become available, all without contaminating the domain expert's application source code written in the PLL. Indeed, if an efficiency programmer has to code a particular abstraction in an ELL anyway, it should require minimal additional work to "plug it into" the SEJITS framework.

In addition, since we emit source code, efficiency programmers can immediately leverage the vast previous work on optimization, autotuning [2], parallelizing optimizing compilers, source transformation, etc. in an incremental fashion and without entangling these concerns with the application logic or contaminating the application source code.

4.3 Drawbacks

Dynamically-generated code is much harder to debug than static code. Our current prototype actually generates C source code, so debugging of JIT-specialized code could be eased by saving that code for inspection. But we recognize that the emission of source code, while a secondary benefit, is not fundamental to our approach.

An additional complication is that floating-point-intensive numerical codes may behave nondeterministically due to the non-associativity of floating-point operations. The nature of JIT specialization is such that it is highly likely that floating-point computations will be refactored or reordered as they are mapped down to the ELL, and that this transformation is decided at runtime and highly platform-dependent but deliberately kept invisible to the productivity programmer. While developers of numerical codes are accustomed to dealing with such problems, we recognize that our introduction of an extra level of indirection may exacerbate it.

5 Related and Future Work

JIT approaches. Early work by Engler and Proebsting [5] illustrated the benefits of *selective* JIT compilation. Products such as Sun's HotSpot JVM [18] performs runtime profiling to decide which functions are worth the overhead of JITting, but must still be able to run arbitrary Java bytecode, whereas SEJITS does not need to be able to specialize arbitrary PLL code. In this way SEJITS is more similar to PyPy [20], which provides an interpreter for a subset of Python written in Python itself to allow experimenting with the implementation of interpreter features. Our approach is also in the spirit of Accelerator [25], which focuses on optimizing specific parallel kernels for GPU's while paying careful attention to the efficient composition of those kernels to maximize use of scarce resources such as GPU fast memory. We anticipate that as our efforts expand we will encounter the opportunity to bring to bear the substantial literature on code-generation and code-optimization research.

Data marshalling/unmarshalling and copying across the PLL/ELL boundary is a significant part of the per-call overhead in our Python prototype. We are looking at approaches such as DiSTiL [23] for ideas on how to optimize data structure composition, placement, and movement.

Approaches to building PLLs. Hudak and others [7] argued for Domain-Specific Embedded Languages (DSELs), an approach in which a domain-specific language is implemented within the constructs provided by some host language. This embedding effectively allows the productivity programmer to mix DSL code with libraries and code in the host language, avoiding a well-known limitation of DSLs. As well, the DSL can be easily evolved as the domain evolves, since its implementation is in terms of a PLL. Our motivation for embedding the specializer machinery parallels this argument: SEJITS does not restrict the PLL programmer since non-specializable functions fall back to the PLL, and it eases the efficiency programmer's job by allowing new specializers to be implemented directly in the PLL.

Stanford's Pervasive Parallel Lab is focusing on DSLs to improve domain-expert programming productivity. In general, their approach assumes the construction of a complete toolchain from the DSL to the hardware. In contrast, SEJITS would accommodate DSLs via embedding in the PLL (DSELs) and thereby allow them to be specialized incrementally using the machinery we describe; Ruby's language features and flexible syntax are particularly well suited to DSEL prototyping [12].

6 Conclusions

Emerging architectures such as manycore processors and GPU's have much to offer applications that can benefit from economical high-performance computing. Unfortunately the gap between the productivity-level languages at which domain experts would like to program (PLLs) and the efficiency-level languages one *must* use to get performance (ELLs) is large and growing. Selective embedded JIT specialization bridges this gap by allowing selective, function-specific and platform-specific specialization of PLL code at runtime via JIT source code generation, compilation and linking. SEJITS can be implemented incrementally and invisibly to the productivity programmer and allows research on efficiency-layer techniques to proceed independently of the languages used by domain experts.

In two case studies, we provided similar abstractions in two different PLLs (Ruby and Python) and targeting different emerging architectures (multicore x86 and multicore GPU). Applying SEJITS to real stencil-code algorithms from a state-of-the-art vision algorithm yields competitive performance to approaches requiring much higher programmer effort. These early results encourage us to further investigate SEJITS as a low-friction framework for rapid uptake of efficiency-programmer techniques by productivity programmers.

In addition to our ParLab colleagues, we thank Jim Larus and Yannis Smaragdakis for valuable feedback on an early draft of this paper. This research is supported by the Universal Parallel Computing Research Centers (UPCRC) program via Microsoft (Award #024263) and Intel (Award #024894), and by matching funds from the UC Discovery Grant (Award #DIG07-10227).

References

[1] K. Asanović, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] J. Bilmes, K. Asanović, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of International Conference on Supercomputing*, Vienna, Austria, July 1997.

[3] B. Catanzaro, B. Su, N. Sundaram, Y. Lee, M. Murphy, and K. Keutzer. Efficient, high-quality image contour detection. *International Conference on Computer Vision, 2009. ICCV 2009*, September 2009.

[4] J. Chaves, J. Nehrbass, B. Guilfoos, J. Gardiner, S. Ahalt, A. Krishnamurthy, J. Unpingco, A. Chalker, A. Warnock, and S. Samsi. Octave and python: High-level scripting languages productivity and performance evaluation. In *HPCMP Users Group Conference, 2006*, pages 429–434, June 2006.

[5] D. R. Engler and T. A. Proebsting. Dcg: an efficient, retargetable dynamic code generation system. In *ASPLOS 1994*, pages 263–272, New York, NY, USA, 1994. ACM.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Prentice-Hall, 1995.

[7] P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28:196, December 1996.

[8] P. Hudak and M. P. Jones. Haskell vs. Ada vs. C++ vs. Awk vs...an experiment in software prototyping productivity. Technical Report YALEU/DCS/RR-1049, Yale University Department of Computer Science, New Haven, CT, 1994.

[9] A. Klöckner. PyCUDA, 2009. `http://mathema.tician.de/software/pycuda`.

[10] Lawrence Berkeley National Laboratory. Chombo. http://seesar.lbl.gov/ANAG/software.html.

[11] M. Maire, P. Arbeláez, C. Fowlkes, and J. Malik. Using contours to detect and localize junctions in natural images. *Computer Vision and Pattern Recognition, 2008. CVPR 2008.*, pages 1–8, June 2008.

[12] E. M. Maximilien, H. Wilkinson, N. Desai, and S. Tai. A domain specific-language for web apis and services mashups. In *5th International Conference on Service Oriented Computing (ICSOC)*, Vienna, Austria, 2007.

[13] Nvidia. Nvidia CUDA, 2007. `http://nvidia.com/cuda`.

[14] T. E. Oliphant. Python for scientific computing. *Computing in Science and Engineering*, 9(3):10–20, 2007.

[15] OpenMP Specification for Parallel Programming. `http://openmp.org/wp/`.

[16] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, March 1998.

[17] M. Paleczny, C. Vick, and C. Click. The java HotSpot server compiler. In *Java Virtual Machine Research and Technology Symposium (JVM '01)*, April 2001.

[18] M. Paleczny, C. Vick, and C. Click. The java hotspot server compiler. In *JVM'01*, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association.

[19] L. Prechelt. An empirical comparison of seven programming languages. *IEEE Computer*, 33(10):23–29, Oct 2000.

[20] PyPy: A Highly Flexible Python Implementation. `http://codespeak.net/pypy`.

[21] RubyInline. `http://www.zenspider.com/ZSS/Products/RubyInline/`.

[22] RubyParser. `http://parsetree.rubyforge.org/ruby_parser/`.

[23] Y. Smaragdakis and D. Batory. Distil: a transformation library for data structures. In *In USENIX Conference on Domain-Specific Languages*, pages 257–270, 1997.

[24] StencilProbe: A Stencil Microbenchmark. `http://www.cs.berkeley.edu/~skamil/projects/stencilprobe/`.

[25] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: Using data parallelism to program gpus for general-purpose uses. In *ASPLOS 2006*, pages 325–335, 2006.