Bringing Parallel Performance to Python with Domain-Specific Selective Embedded Just-in-Time Specialization

Shoaib Kamil, Derrick Coetzee, Armando Fox

Abstract—Today's *productivity programmers*, such as scientists who need to write code to do science, are typically forced to choose between productive and maintainable code with modest performance (e.g. Python plus native libraries such as SciPy [SciPy]) or complex, brittle, hardware-specific code that entangles application logic with performance concerns but runs two to three orders of magnitude faster (e.g. C++ with OpenMP, CUDA, etc.). The dynamic features of modern productivity languages like Python enable an alternative approach that bridges the gap between productivity and performance. SEJITS (Selective, Embedded, Just-in-Time Specialization) embeds domain-specific languages (DSLs) in high-level languages like Python for popular computational *kernels* such as stencils, matrix algebra, and others. At runtime, the DSLs are "compiled" by combining expert-provided source code templates specific to each problem type, plus a strategy for optimizing an abstract syntax tree representing a domain-specific but language-independent representation of the problem instance. The result is efficiency-level (e.g. C, C++) code callable from Python whose performance equals or exceeds that of handcrafted code, plus performance portability by allowing multiple code generation strategies within the same specializer to target different hardware present at runtime, e.g. multicore CPUs vs. GPUs. Application writers never leave the Python world, and we do not assume any modification or support for parallelism in Python itself.

We present Asp ("Asp is SEJITS for Python") and initial results from several domains. We demonstrate that domain-specific specializers allow highly-productive Python code to obtain performance meeting or exceeding expert-crafted low-level code on parallel hardware, without sacrificing maintainability or portability.

Index Terms—parallel programming, specialization

Introduction

It has always been a challenge for productivity programmers, such as scientists who write code to support doing science, to get both good performance and ease of programming. This is attested by the proliferation of high-performance libraries such as BLAS, OSKI [OSKI] and FFTW [FFTW], by domain-specific languages like SPIRAL [SPIRAL], and by the popularity of the natively-compiled SciPy [SciPy] libraries among others. To make things worse, processor clock scaling has run into physical limits, so future performance increases will be the result of increasing hardware parallelism rather than single-core speedup, making programming even more complex. As a result, programmers must choose between productive and maintainable but slow-running code on the one hand, and performant but complex and hardware-specific code on the other hand.

The corresponding author is with Department of Computer Science, UC Berkeley, e-mail: skamil@cs.berkeley.edu.

The usual solution to bridging this gap is to provide compiled native libraries for certain functions, as the SciPy package does. However, in some cases libraries may be inadequate or insufficient. Various families of computational patterns share the property that while the *strategy* for mapping the computation onto a particular hardware family is common to all problem instances, the specifics of the problem are not. For example, consider a stencil computation, in which each point in an n-dimensional grid is updated with a new value that is some function of its neighbors' values. The general strategy for optimizing sequential or parallel code given a particular target platform (multicore, GPU, etc.) is independent of the specific function, but because that function is unique to each application, capturing the stencil abstraction in a traditional compiled library is awkward, especially in the efficiency level languages typically used for performant code (C, C++, etc.) that don't support higher-order functions gracefully.

Even if the function doesn't change much across applications, work on auto-tuning [ATLAS] has shown that for algorithms with tunable implementation parameters, the performance gain from fine-tuning these parameters compared to setting them naively can be up to 5×. [SC08] Indeed, the complex internal structure of auto-tuning libraries such as the Optimized Sparse Kernel Interface [OSKI] is driven by the fact that often runtime information is necessary to choose the best execution strategy or tuning-parameter values.

We therefore propose a new methodology to address this performance-productivity gap, called SEJITS (Selective Embedded Just-in-Time Specialization) [Cat09]. This methodology embeds domain-specific languages within high-level languages, and the embedded DSLs are specialized at runtime into high-performance, low-level code by leveraging metaprogramming and introspection features of the host languages, all invisibly to the application programmer. The result is performance-portable, highly-productive code whose performance rivals or exceeds that of implementations hand-written by experts.

The insight of our approach is that because each embedded DSL is specific to just one type of computational pattern (stencil, matrix multiplication, etc.), we can select an implementation strategy and apply optimizations that take advantage of domain knowledge in generating the efficiency-level code. For example, returning to the domain of sten-

cils, one optimization called *time skewing* [Wonn00] involves blocking in time for a stencil applied repeatedly to the same grid. This transformation is not meaningful unless we know the computation is a stencil and we also know the stencil's "footprint," so a generic optimizing compiler would be unable to identify the opportunity to apply it.

We therefore leverage the dynamic features of modern languages like Python to defer until runtime what most libraries must do at compile time, and to do it with higher-level domain knowledge than can be inferred by most compilers.

Asp: Approach and Mechanics

High-level productivity or scripting languages have evolved to include sophisticated introspection and FFI (foreign function interface) capabilities. We leverage these capabilities in Python to build domain- and machine-specific *specializers* that transform user-written code in a high-level language in various ways to expose parallelism, and then generate code for a a specific machine in a low-level language. Then, the code is compiled, linked, and executed. This entire process occurs transparently to the user; to the user, it appears that an interpreted function is being called.

Asp (a recursive acronym for "Asp is SEJITS for Python") is a collection of libraries that realizes the SEJITS approach in Python, using Python both as the language in which application programmers write their code (the *host language*) and the language in which transformations and code generation are carried out (the *transformation language*). Note that in general the host and transformation languages need not be the same, but Python happily serves both purposes well.

Specifically, Asp provides a framework for creating Python classes (*specializers*), each of which represents a particular computational pattern. Application writers subclass these to express specific problem instances. The specializer class's methods use a combination of pre-supplied low-level source code snippets (*templates*) and manipulation of the Python abstract syntax tree (AST, also known as a parse tree) to generate low-level source code in an efficiency-level language (ELL) such as C, C++ or CUDA.

For problems that call for passing in a function, such as the stencil example above, the application writer codes the function in Python (subject to some restrictions) and the specializer class iterates over the function's AST to lower it to the target ELL and inline it into the generated source code. Finally, the source code is compiled by an appropriate conventional compiler, the resulting object file is dynamically linked to the Python interpreter, and the method is called like a native library.

Python code in the application for which no specializer exists is executed by Python as usual. As we describe below, a recommended best practice for creating new specializers is that they include an API-compatible, pure-Python implementation of the kernel(s) they specialize in addition to providing a code-generation-based implementation, so that every valid program using Asp will also run in pure Python without Asp (modulo removing the import directives that refer to Asp). This allows the kernel to be executed and debugged using standard Python tools, and provides a reference implementation for isolating bugs in the specializer.

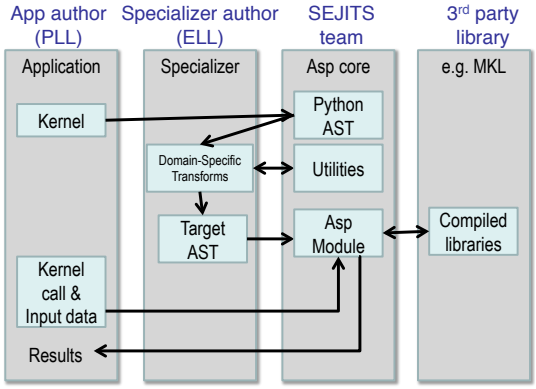

Figure 1: *Separation of concerns in Asp. App authors write code that is transformed by specializers, using Asp infrastructure and third-party libraries.*

One of Asp's primary purposes is separating application and algorithmic logic from code required to make the application run fast. Application writers need only program with high-level class-based constructs provided by specializer writers. It is the task of these specializer writers to ensure the constructs can be specialized into fast versions using infrastructure provided by the Asp team as well as third-party libraries. An overview of this separation is shown in Figure 1.

An overview of the specialization process is as follows. We intercept the first call to a specializable method, grab the AST of the Python code of the specializable method, and immediately transform it to a domain-specific AST, or DAST. That is, we immediately move the computation into a domain where problem-specific optimizations and knowledge can be applied, by applying transformations to the DAST. Returning once again to the stencil, the DAST might have nodes such as "iterate over neighbors" or "iterate over all stencil points." These abstract node types, which differ from one specializer to another, will eventually be used to generate ELL code according to the code generation strategy chosen; but at this level of representation, one can talk about optimizations that make sense *for stencils specifically* as opposed to those that make sense *for iteration generally*.

After any desired optimizations are applied to the domain-specific (but language- and platform-independent) representation of the problem, conversion of the DAST into ELL code is handled largely by CodePy [CodePy]. Finally, the generated source code is compiled by an appropriate downstream compiler into an object file that can be called from Python. Code caching strategies avoid the cost of code generation and compilation on subsequent calls.

In the rest of this section, we outline Asp from the point of view of application writers and specializer writers, and outline the mechanisms the Asp infrastructure provides.

Application Writers

From the point of view of application writers, using a specializer means installing it and using the domain-specific classes defined by the specializer, while following the conventions

```
from stencil_kernel import *

class ExampleKernel(StencilKernel):
  def kernel(self, in_grid, out_grid):
    for x in out_grid.interior_points():
      for y in in_grid.neighbors(x, 1):
        out_grid[x] = out_grid[x] + in_grid[y]

in_grid = StencilGrid([5,5])
in_grid.data = numpy.ones([5,5])
out_grid = StencilGrid([5,5])
ExampleKernel().kernel(in_grid, out_grid)
```

```
1 1 1
1 1 1
1 1 1
  ↓
0 0 0
0 4 0
0 0 0
```

Figure 2: *Example stencil application. Colored source lines match up to nodes of same color in Figure 4.*

outlined in the specializer documentation. Thus, application writers never leave the Python world. As a concrete example of a non-trivial specializer, our structured grid (stencil) specializer provides a StencilKernel class and a StencilGrid class (the grid over which a stencil operates; it uses NumPy internally). An application writer subclasses the StencilKernel class and overrides the function `kernel()`, which operates on StencilGrid instances. If the defined kernel function is restricted to the class of stencils outlined in the documentation, it will be specialized; otherwise the program will still run in pure Python.

An example using our stencil specializer's constructs is shown in Figure 2.

Specializer Writers

Specializer writers often start with an existing implementation of a solution, written in an ELL, for a particular problem type on particular hardware. Such solutions are devised by human experts who may be different from the specializer writer, e.g. numerical-analysis researchers or auto-tuning researchers. Some parts of the solution which remain the same between problem instances, or the same with very small changes, can be converted into *templates*, which are simply ELL source code with a basic macro substitution facility, supplied by [Mako], for inserting values into fixed locations or "holes" at runtime.

Other parts of the ELL solution may vary widely or in a complex manner based on the problem instance. For these cases, a better approach is to provide a set of rules for transforming the DAST of this type of problem in order to realize the optimizations present in the original ELL code. Finally, the specializer writer provides high-level transformation code to drive the entire process.

Specializer writers use Asp infrastructure to build their domain-specific translators. In Asp, we provide two ways to generate low-level code: templates and abstract syntax tree (AST) transformation. For many kinds of computations, using templates is sufficient to translate from Python to C++, but for others, phased AST transformation allows application programmers to express arbitrary computations to specialize.

In a specializer, the user-defined kernel is first translated into a Python AST, and analyzed to see if the code supplied by the application writer adheres to the restrictions of the specializer. Only code adhering to a narrow subset of Python, characterizing the embedded domain-specific language, will be accepted. Since specializer writers frequently need to iterate over ASTs, the Asp infrastructure provides classes that implement a visitor pattern on these ASTs (similar to Python's `ast.NodeTransformer`) to implement their specialization phases. The final phase transforms the DAST into a target-specific AST (e.g, C++ with OpenMP extensions). The Example Walkthrough section below demonstrates these steps in the context of the stencil kernel specializer.

Specializer writers can then use the Asp infrastructure to automatically compile, link, and execute the code in the final AST. In many cases, the programmer will supply several code variants, each represented by a different ASTs, to the Asp infrastructure. Specializer-specific logic determines which variant to run; Asp provides functions to query the hardware features available (number of cores, GPU, etc.). Asp provides for capturing and storing performance data and caching compiled code across runs of the application.

For specializer writers, therefore, the bulk of the work consists of exposing an understandable abstraction for specializer users, ensuring programs execute whether specialized or not, writing test functions to determine specializability (and giving the user meaningful feedback if not), and expressing their translations as phased transforms.

Currently, specializers have several limitations. The most important current limitation is that specialized code cannot call back into the Python interpreter, largely because the interpreter is not thread safe. We are implementing functionality to allow serialized calls back into the interpreter from specialized code.

In the next section, we show an end-to-end walkthrough of an example using our stencil specializer.

Example Walkthrough

In this section we will walk through a complete example of a SEJITS translation and execution on a simple stencil example. We begin with the application source shown in Figure 2. This simple two-dimensional stencil walks over the interior points of a grid and for each point computes the sum of the four surrounding points.

This code is executable Python and can be run and debugged using standard Python tools, but is slow. By merely modifying ExampleKernel to inherit from the StencilKernel base class, we activate the stencil specializer. Now, the first time the `kernel()` function is called, the call is redirected to the stencil specializer, which will translate it to low-level C++ code, compile it, and then dynamically bind the machine code to the Python environment and invoke it.

The translation performed by any specializer consists of five main phases, as shown in Figure 3:

1. Front end: Translate the application source into a domain-specific AST (DAST)
2. Perform platform-independent optimizations on the DAST using domain knowledge.
3. Select a platform and translate the DAST into a platform-specific AST (PAST).
4. Perform platform-specific optimizations using platform knowledge.

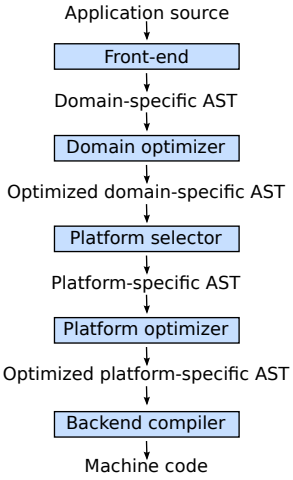

Figure 3: *Pipeline architecture of a specializer.*

5. Back end: Generate low-level source code, compile, and dynamically bind to make available from the host language.

As with any pipeline architecture, each phase's component is reusable and can be easily replaced with another component, and each component can be tested independently. This supports porting to other application languages and other hardware platforms, and helps divide labor between domain experts and platform performance experts. These phases are similar to the phases of a typical optimizing compiler, but are dramatically less complex due to the domain-specific focus and the Asp framework, which provides utilities to support many common tasks, as discussed in the previous section.

In the stencil example, we begin by invoking the Python runtime to parse the `kernel()` function and produce the abstract syntax tree shown in Figure 4. The front end walks over this tree and matches certain patterns of nodes, replacing them with other nodes. For example, a call to the function `interior_points()` is replaced by a domain-specific StencilInterior node. If the walk encounters any pattern of Python nodes that it doesn't handle, for example a function call, the translation fails and produces an error message, and the application falls back on running the `kernel()` function as pure Python. In this case, the walk succeeds, resulting in the DAST shown in Figure 4. Asp provides utilities to facilitate visiting the nodes of a tree and tree pattern matching.

The second phase uses our knowledge of the stencil domain to perform platform-independent optimizations. For example, we know that a point in a two-dimensional grid has four neighbors with known relative locations, allowing us to unroll the innermost loop, an optimization that makes sense on all platforms.

The third phase selects a platform and translates to a platform-specific AST. In general, the platform selected will depend on available hardware, performance characteristics of the machine, and properties of the input (such as grid size). In this example we will target a multicore platform using the OpenMP framework. At this point the loop over the interior points is mapped down to nested parallel for loops, as shown

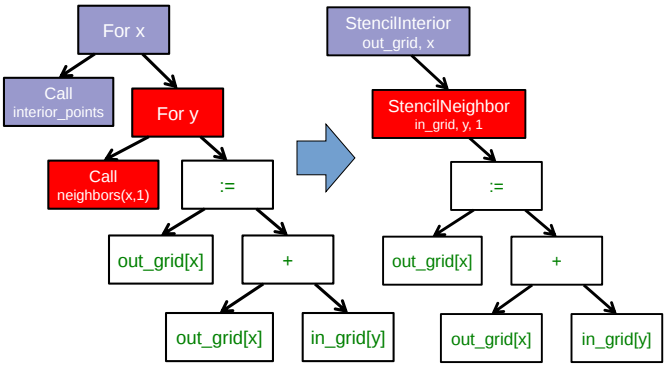

Figure 4: *Left: Initial Python abstract syntax tree. Right: Domain-specific AST.*

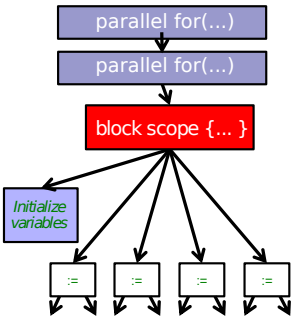

Figure 5: *Platform-specific AST.*

in Figure 5. The Asp framework provides general utilities for transforming arithmetic expressions and simple assignments from the high-level representation used in DASTs to the low-level platform-specific representation, which handles the body of the loop.

Because the specializer was invoked from the first call of the `kernel()` function, the arguments passed to that call are available. In particular, we know the dimensions of the input grid. By hardcoding these dimensions into the AST, we enable a wider variety of optimizations during all phases, particularly phases 4 and 5. For example, on a small grid such as the 8x8 blocks encountered in JPEG encoding, the loop over interior points may be fully unrolled.

The fourth phase performs platform-specific optimizations. For example, we may partially unroll the inner loop to reduce branch penalties. This phase may produce several ASTs to support run-time auto-tuning, which times several variants with different optimization parameters and selects the best one.

Finally, the fifth phase, the backend, is performed entirely by components in the Asp framework and the CodePy library. The PAST is transformed into source code, compiled, and dynamically bound to the Python environment, which then invokes it and returns the result to the application. Interoperation between Python and C++ uses the Boost.Python library, which handles marshalling and conversion of types.

The compiled `kernel()` function is cached so that if the function is called again later, it can be re-invoked directly without the overhead of specialization and compilation. If the input grid dimensions were used during optimization, the input

dimensions must match on subsequent calls to reuse the cached version.

Results

SEJITS claims three benefits for productivity programmers. The first is *performance portability*. A single specializer can include code generation strategies for radically different platforms, and even multiple code variants using different strategies on the *same* platform depending on the problem parameters. The GMM specializer described below illustrates this advantage: a single specializer can produce code either for NVIDIA GPUs (in CUDA) or x86 multicore processors (targeting the Cilk Plus compiler), and the same Python application can run on either platform.

The second benefit is the ability to let application writers work with patterns requiring higher-order functions, something that is cumbersome to do in low-level languages. We can inline these functions into the emitted source code and let the low-level compiler optimize the solution using the maximum available information. Our stencil specializer, as described below, demonstrates this benefit; the performance of the generated code reaches 87% of the achievable memory bandwidth of the multicore machine on which it runs.

The third benefit is the ability to take advantage of auto-tuning or other runtime performance optimizations even for simple problems. Our matrix-powers specializer, which computes $\{x, Ax, A^2x, ..., A^kx\}$ for a sparse matrix A and vector x (an important computation in Krylov-subspace solvers), demonstrates this benefit. Its implementation uses a recently-developed *communication-avoiding* algorithm for matrix powers that runs about an order of magnitude faster than Python+SciPy (see performance details below) while remaining essentially API-compatible with SciPy. Beyond the inherent performance gains from communication-avoidance, a number of parameters in the implementation can be tuned based on the matrix structure in each individual problem instance; this is an example of an optimization that cannot easily be done in a library.

Stencil

To demonstrate the performance and productivity effectiveness of our stencil specializer, we implemented two different computational stencil kernels using our abstractions: a 3D laplacian operator, and a 3D divergence kernel. For both kernels, we run a simple benchmark that iteratively calls our specializer and measures the time for applying the operator (we ensure the cache is cleared in between calls). Both calculations are memory-bound; that is, they are limited by the available bandwidth from memory. Therefore, in accordance to the roofline model [SaWi09], we measure performance compared to measured memory bandwidth performance using the parallel STREAM [STREAM] benchmark.

Figure 6 shows the results of running our kernels for a 256^3 grid on a single-socket quad-core Intel Core i7-840 machine running at 2.93 GHz, using both the OpenMP and Cilk Plus backends. First-run time is not shown; the code generation and compilation takes tens of seconds (mostly due to the speed

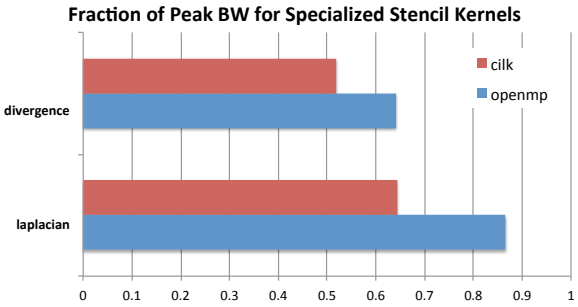

Figure 6: *Performance as fraction of memory bandwidth peak for two specialized stencil kernels. All tests compiled using the Intel C++ compiler 12.0 on a Core i7-840.*

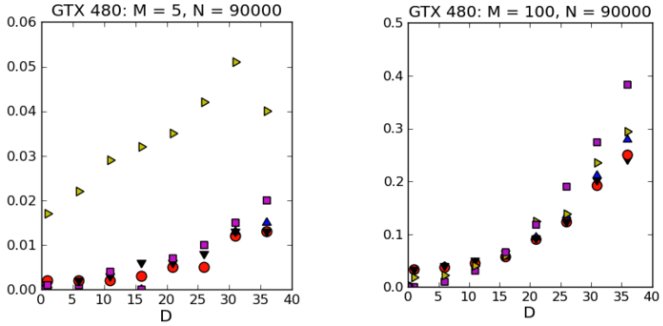

Figure 7: *Runtimes of GMM variants as the D parameter is varied on an Nvidia Fermi GPU (lower is better). The specializer picks the best-performing variant to run.*

of the Intel compiler). In terms of performance, for the 3D laplacian, we obtain 87% of peak memory bandwidth, and 64% of peak bandwidth for the more cache-unfriendly divergence kernel, even though we have only implemented limited optimizations. From previous work [Kam10], we believe that, by adding only a few more tuning parameters, we can obtain over 95% of peak performance for these kernels. In contrast, pure Python execution is nearly three orders of magnitude slower.

In terms of productivity, it is interesting to note the difference in LoC between the stencils written in Python and the produced low-level code. Comparing the divergence kernel with its best-performing produced variant, we see an increase from five lines to over 700 lines--- an enormous difference. The Python version expresses the computation succinctly; using machine characteristics to express fast code requires expressing the stencil more verbosely in a low-level language. With our specialization infrastructure, programmers can continue to write succinct code and have platform-specific fast code generated for them.

Gaussian Mixture Modeling

Gaussian Mixture Models (GMMs) are a class of statistical models used in a wide variety of applications, including image segmentation, speech recognition, document classification, and many other areas. Training such models is done using the Expectation Maximization (EM) algorithm, which is iterative and highly data parallel, making it amenable to execution on GPUs as well as modern multicore processors. However, writing high

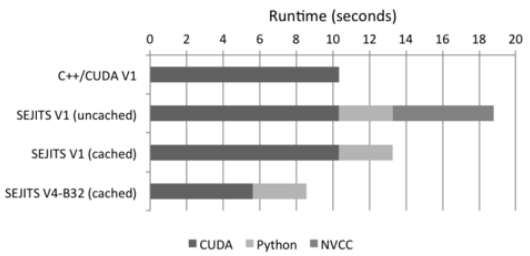

Figure 8: *Overall performance of specialized GMM training versus original optimized CUDA algorithm. Even including specializer overhead, the specialized EM training outperforms the original CUDA implementation.*

performance GMM training algorithms are difficult due to the fact that different code variants will perform better for different problem characteristics. This makes the problem of producing a library for high performance GMM training amenable to the SEJITS approach.

A specializer using the Asp infrastructure has been built by Cook and Gonina [Co10] that targets both CUDA-capable GPUs and Intel multicore processors (with Cilk Plus). The specializer implements four different parallelization strategies for the algorithm; depending on the sizes of the data structures used in GMM training, different strategies perform better. Figure 7 shows performance for different strategies for GMM training on an NVIDIA Fermi GPU as one of the GMM parameters are varied. The specializer uses the best-performing variant (by using the different variants to do one iteration each, and selecting the best-performing one) for the majority of iterations. As a result, even if specialization overhead (code generation, compilation/linking, etc.) is included, the specialized GMM training algorithm outperforms the original, hand-tuned CUDA implementation on some classes of problems, as shown in Figure 8.

Matrix Powers

Recent developments in communication-avoiding algorithms [Bal09]_(AF: need canonical citation here, as well as specific cite for Erin and Nick's CA-matrix powers presentation at EuroSomethingOrOther) have shown that the performance of parallel implementations of several algorithms can be substantially improved by partitioning the problem so as to do redundant work in order to minimize inter-core communication. One example of an algorithm that admits a communication-avoiding implementation is matrix powers [Hoe10]: the computation $\{x, Ax, A^2x, \ldots, A^kx\}$ for a sparse matrix A and vector x, an important building block for communication-avoiding sparse Krylov solvers. A specializer currently under development enables efficient parallel computation of this set of vectors on multicore processors.

The specializer generates parallel communication avoiding code using the pthreads library that implements the PA1 [Hoe10] kernel to compute the vectors more efficiently than just repeatedly doing the multiplication $A \times x$. The naive algorithm, shown in Figure 9, requires communication at each level. However, for many matrices, we can restructure the computation such that communication only occurs every k steps, and before every superstep of k steps, all communication required is completed. At the cost of redundant computation, this reduces the number of communications required. Figure 9 shows the restructured algorithm.

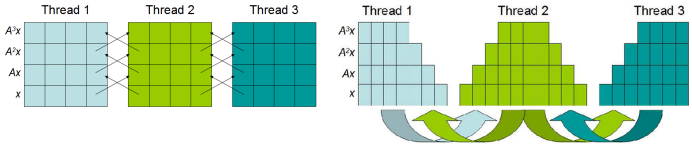

Figure 9: *Left: Naive A^kx computation. Communication required at each level. Right: Algorithm PA1 for communication-avoiding matrix powers. Communication occurs only after k levels of computation, at the cost of redundant computation.*

The specializer implementation further optimizes the PA1 algorithm using traditional matrix optimization techniques such as cache and register blocking. Further optimization using vectorization is in progress.

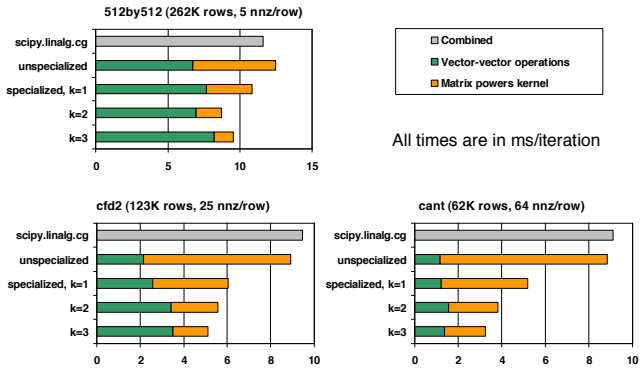

Figure 10: *Results comparing communication-avoiding CG with our matrix powers specializer and SciPy's default solver, run on an Intel Nehalem machine.*

To see what kinds of performance improvements are possible using the specialized communication-avoiding matrix powers kernel, Morlan implemented a conjugate gradient (CG) solver in Python that uses the specializer. Figure 10 shows the results for three test matrices and compares performance against `scipy.linalg.solve` which calls the LAPACK `dgesv` routine. Even with just the matrix powers kernel specialized, the CA CG already outperforms the native solver routine used by SciPy.

Related Work

Allowing domain scientists to program in higher-level languages is the goal of a number of projects in Python, including SciPy [SciPy] which brings Matlab-like functionality for numeric computations into Python. In addition, domain-specific projects such as Biopython [Biopy] and the Python Imaging Library (PIL) [PIL] also attempt to hide complex operations and data structures behind Python infrastructure, making programming simpler for users.

Another approach, used by the Weave subpackage of SciPy, allows users to express C++ code that uses the Python C API as strings, inline with other Python code, that is then compiled

and run. Cython [Cython] is an effort to write a compiler for a subset of Python, while also allowing users to write extension code in C. Another instance of the SEJITS approach is Copperhead [Cat09], which implements SEJITS targeting CUDA GPUs for data parallel operations.

The idea of using multiple code variants, with different optimizations applied to each variant, is a cornerstone of auto-tuning. Auto-tuning was first applied to dense matrix computations in the PHiPAC (Portable High Performance ANSI C) library [PHiPAC]. Using parametrized code generation scripts written in Perl, PHiPAC generated variants of generalized matrix multiply (GEMM) with loop unrolling, cache blocking, and a number of other optimizations, plus a search engine, to, at install time, determine the best GEMM routine for the particular machine. After PHiPAC, auto-tuning has been applied to a number of domains including sparse matrix-vector multiplication (SpMV) [OSKI], Fast Fourier Transforms (FFTs) [SPIRAL], and multicore versions of stencils [KaDa09], [Kam10], [Tang11], showing large improvements in performance over simple implementations of these kernels.

Conclusion

We have presented a new approach to bridging the "productivity/efficiency gap": rather than relying solely on libraries to allow productivity programmers to remain in high-level languages, we package the expertise of human experts as a collection of code templates in a low-level language (C++/OpenMP, etc.) and a set of transformation rules to generate and optimize problem-specific ASTs at runtime. The resulting low-level code runs as fast or faster than the original hand-produced version.

Unlike many prior approaches, we neither propose a standalone DSL nor try to imbue a full compiler with the intelligence to "auto-magically" recognize and optimize compute-intensive problems. Rather, the main contribution of SEJITS is separation of concerns: expert programmers can express implementation optimizations that make sense only for a particular problem (and perhaps only on specific hardware), and package this expertise in a way that makes it widely reusable by Python programmers. Application writers remain oblivious to the details of specialization, making their code simpler and shorter as well as performance-portable.

We hope that our promising initial results will encourage others to contribute to building up the ecosystem of Asp specializers.

Acknowledgments

Henry Cook and Ekaterina Gonina implemented the GMM specializer. Jeffrey Morlan is implementing the matrix-powers specializer based on algorithmic work by Mark Hoemmen, Erin Carson and Nick Knight. Research supported by DARPA (contract #FA8750-10-1-0191), Microsoft Corp. (Award #024263), and Intel Corp. (Award #024894), with matching funding from the UC Discovery Grant (Award #DIG07-10227) and additional support from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Oracle, and Samsung.

References

[ATLAS] R. C. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimization of Software and the ATLAS project. Parallel Computing, vol. 27(1-2), pp. 3–35, 2001.

[Bal09] G. Ballard, J. Demmel, O. Holtz, O. Schwartz. Minimizing Communication in Numerical Linear Algebra. UCB Tech Report (UCB/EECS-2009-62), 2009.

[Biopy] Biopython. http://biopython.org.

[Cat09] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, A. Fox. SEJITS: Getting Productivity and Performance with Selective Embedded Just-in-Time Specialization. Workshop on Programming Models for Emerging Architectures (PMEA), 2009

[CodePy] CodePy Homepage. http://mathema.tician.de/software/codepy

[Co10] H. Cook, E. Gonina, S. Kamil, G. Friedland†, D. Patterson, A. Fox. CUDA-level Performance with Python-level Productivity for Gaussian Mixture Model Applications. 3rd USENIX Workshop on Hot Topics in Parallelism (HotPar) 2011.

[Cython] R. Bradshaw, S. Behnel, D. S. Seljebotn, G. Ewing, et al., The Cython compiler, http://cython.org.

[FFTW] M. Frigo and S. Johnson. The Design and Implementation of FFTW3. Proceedings of the IEEE 93 (2), 216–231 (2005). Invited paper, Special Issue on Program Generation, Optimization, and Platform Adaptation.

[Hoe10] M. Hoemmen. Communication-Avoiding Krylov Subspace Methods. PhD thesis, EECS Department, University of California, Berkeley, May 2010.

[KaDa09] K. Datta. Auto-tuning Stencil Codes for Cache-Based Multicore Platforms. PhD thesis, EECS Department, University of California, Berkeley, Dec 2009.

[Kam10] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An Auto-Tuning Framework for Parallel Multicore Stencil Computations. International Parallel and Distributed Processing Symposium, 2010.

[Mako] Mako Templates for Python. http://www.makotemplates.org

[OSKI] OSKI: Optimized Sparse Kernel Interface. http://bebop.cs.berkeley.edu/oski.

[PHiPAC] J. Bilmes, K. Asanovic, J. Demmel, D. Lam, and C.W. Chin. PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology and its Application to Matrix Multiply. LAPACK Working Note 111.

[PIL] Python Imaging Library. http://pythonware.com/products/pil.

[SaWi09] S. Williams, A. Waterman, D. Patterson. Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures. Communications of the ACM (CACM), April 2009.

[SC08] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. SC2008: High performance computing, networking, and storage conference, 2008.

[SciPy] Scientific Tools for Python. http://www.scipy.org.

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

[STREAM] The STREAM Benchmark. http://www.cs.virginia.edu/stream

[Tang11] Y.Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The Pochoir Stencil Compiler. 23rd ACM Symposium on Parallelism in Algorithms and Architectures, 2011.

[Wonn00] D. Wonnacott. Using Time Skewing to Eliminate Idle Time due to Memory Bandwidth and Network Limitations. International Parallel and Distributed Processing Symposium, 2000.