Chapter 12

Getting Productivity and Performance with Selective Embedded Just-in-Time Specialization

Armando Fox

Productivity was a keyword in the original Par Lab vision: non-experts in computing (whom we call *productivity programmers*) would be able to write source-portable, performance-portable, correct, high-performance programs without requiring expert help for each one. Initially, we envisioned them using a new language, possibly a derivative of Titanium [9]. However, we quickly realized that language adoption is historically very difficult, especially for non-expert programmers; and even if we assumed they'd adopt a good language, the difficulty of writing parallel code would remain a major obstacle.

Instead, inspired by the rise to prominence of so-called *scripting languages* such as Python and Ruby for general-purpose computing, Professors Dave Patterson, Krste Asanović, Koushik Sen, and Armando Fox decided to teach a graduate seminar course exploring the possibility of modifying existing scripting languages to work in parallel or to create a new, parallel scripting language. Modifying an existing non-threadsafe interpreter to allow first-class parallelism was a daunting task, as these languages were designed without any parallelism in mind. Instead, therefore, two projects decided to leverage features of modern scripting languages—so-called productivity-level languages or PLLs—to build domain-specific parallel constructs into the languages without modifying the existing interpreters. These two projects eventually evolved into Copperhead and Asp, two different frameworks that allowed performant yet productive code to be written in Python, a scripting language enjoying wide and rapid adoption by non-expert programmers such as the scientists and application experts we were targeting. The two frameworks had different goals but shared similar philosophical underpinnings.

The main idea in both was that certain performance-critical parts of Python apps would, at runtime, cause code to be generated in a lower-level but high-performance language (a so-called efficiency-level language, or ELL), while non-performance-critical parts would be executed by the Python interpreter as usual. The ideas of embedded domain-specific languages (DSELs) would determine what parts of the Python code would result in runtime code generation. In Copperhead, a subset of Python that uses Python's list syntax to express data parallel operations, is translated into CUDA, the language used by NVIDIA Graphics Processing Units, which is similar to OpenCL. The knowledge of how to perform these transformations lies entirely in the Copperhead runtime compiler, which was created by an expert programmer, Bryan Catanzaro. Asp instead took a modular approach: an expert programmer would create the runtime compiler for a single computational pattern, such as structured grid or graph traversal, and use generic facilities in the Asp framework, such as syntax tree manipulation, interrogation of hardware capabilities, and so on, to assist in code generation and compilation. Each such pattern-specific runtime compiler is called a *specializer*; Shoaib Kamil wrote most of Asp and the first specializer for structured grids, based on his extensive previous work optimizing those codes in Fortran using a combination of code generation and autotuning [6]. The early prototypes of both Copperhead

and Asp demonstrated that each approach yielded both productivity gains and performance comparable to hand-tuned code, and the common philosophy underpinning them was published in an early workshop paper [4].

Copperhead [3] provides a small set of data parallel functions such as `map()` and `reduce()`, along with infrastructure that compiles programs using these functions to parallel implementations in CUDA and OpenMP. To program using Copperhead, the user writes Python programs in which certain functions are selectively annotated as complying with the restrictions of Copperhead. When such a function is called from the Python interpreter, the runtime compiles the function and executes it on a parallel platform. Selectively embedding Copperhead programs allows the entire application to be written in the same language, using the vast array of Python libraries to create the application, and annotating computationally expensive portions of the program as Copperhead compliant to provide efficient parallel execution. To avoid bottlenecks due to runtime compilation, Copperhead developed infrastructure for binary caching, which was later reused in Asp. To demonstrate the utility of the Copperhead framework, we wrote several applications inspired by the Content Based Image Retrieval project, including support vector machine training and a solver for optical flow computation. We found that Copperhead programs provided 45-100% of the performance of hand-written CUDA programs, while requiring significantly less programmer effort.

While originally targeting GPUs, Copperhead has since been extended to include support for parallel CPUs. We also improved the quality of code produced by the Copperhead compiler, allowing it to use on-chip memory in certain cases. Copperhead continues as an open source project sponsored by NVIDIA.

The Copperhead compiler has full responsibility for coordinating function composition, and requires a full type inference procedure to interoperate with Python, whose types are implicit. This gives Copperhead great flexibility for implementation: for example, the compiler decides which functions should be fused together, and which should be separated; the compiler decides which functions should be executed in parallel and which should be executed sequentially. However, it also means that the scope of the compiler and runtime is fairly large. Asp, a SEJITS project that began after Copperhead, chose a more targeted approach, envisioning large numbers of very specific mini-compilers and a community of mini-compiler writers.

In Asp, programmers write Python as usual but have access to a library of Python classes implementing particular patterns, such as structured grid, graph algorithms, and so on. Asp took Copperhead's ideas further by separating the general machinery required for runtime code generation—syntax tree analysis, mapping high-level DSEL operations to lower-level operations that could be expressed in ELLs such as C/OpenMP, Cilk+, and so on—from the individual mini-compilers or specializers, allowing specializers to share this common machinery. Also, the transformation process from DSEL code to compilable PLL code follows a sequence of well-defined phases in Asp, so individual specializer phases could be reused as well. For example, one specializer for audio signal processing computations [5] has back ends for both CUDA and C/OpenMP, and automatically selects both the appropriate specializer depending on what hardware is available at runtime and (in the case of OpenMP) the appropriate code-generation variant depending on the dimensions of the problem to be solved. Another specializer for the Bag of Little Bootstraps sampling algorithm [8] has back ends for OpenMP, Cilk+, and cloud computing, the latter based on generating code for the Spark [10] distributed computing framework. Finally, the specializers themselves are written in Python, so the expert programmer creating a specializer has access to the same level of productivity, selection of libraries, and so forth as the application writer who will eventually use that specializer. Several highly successful specializers were created using Asp and some are in production use in research projects unrelated to the Par Lab [7]. One important lesson learned during the development of these specializers is that our runtime DSEL-based approach also simplified the creation of autotuned libraries, because of our reliance on runtime introspection in each specializer. As a result, we developed several autotuned libraries packaged using Asp, enabling runtime autotuning that would otherwise require large amounts of infrastructure.

SEJITS became one of the big bets of the Par Lab because it allowed us to assume that design space exploration would no longer be limited to using toy benchmarks, or require standing up a full software stack on new hardware: a new specializer exploiting new hardware is much easier to write (existing specializers are a few hundred to a couple of thousand lines of Python), and plays well with the rest of the runtime system. Indeed, when we integrated Asp technology with the existing Knowledge Discovery Toolbox (KDT) graph algorithm framework, we eliminated the $80\times$ slowdown penalty incurred when scientists chose to express their graph algorithm primitives in Python rather than C++ [2]. (That work is an ongoing collaboration with UC Santa Barbara and Lawrence Berkeley National Laboratory.) The fast-prototyping philosophy enabled by SEJITS is a cornerstone of the follow-on ASPIRE project and was part of the inspiration for CHISEL [1], which uses DSELs embedded in Scala to synthesize high-performance software simulators for new hardware designs.

Though much progress has been made during the scope of the Par Lab, much work remains. The two most pressing open research problems are debugging and composition. Building DSEL compilers remains challenging, in part because bugs can manifest in the specializer, in the application code (e.g. if it uses the specializer in a manner that violates the specializer's semantics), or at the platform level (e.g. if the specialized code contains race conditions that the specializer writer did not anticipate). We have done some exploratory work on multi-level debugging to help detect and trace the source of such errors, but debugging will be a major focus going forward.

Composition of DSEL compilers is difficult, and not restricted to SEJITS: in autotuning, it is well known that the optimal co-tuning of two composed computations may not match the composed optimally-tuned individual computations. General composition is complex and perhaps unnecessary: we have begun approaching the problem by building frameworks for related kernels that can be composed together in useful, though restricted, ways [5]. The hardware/-software co-tuning activities of ASPIRE should provide some interesting problems that we can use as drivers to better understand composition.

Bibliography

[1] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. Chisel: Constructing hardware in a scala embedded language. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1212–1221, 2012.

[2] A. Buluç, E. Duriakova, A. Fox, J. R. Gilbert, S. Kamil, A. Lugowki, L. Oliker, and S. W. Williams. High-productivity and high-performance analysis of filtered semantic graphs. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS'13) (to appear)*. IEEE Computer Society, 2013.

[3] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming (PPoPP '11)*, New York, NY, USA, 2011.

[4] B. Catanzaro, S. Kamil, Y. Lee, K. Asanović, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox. SEJITS: Getting productivity and performance with selective embedded JIT specialization. In *Workshop on Programming Models for Emerging Architectures (PMEA 2009)*, Raleigh, NC, October 2009.

[5] E. Gonina, G. Friedland, E. Battenberg, M. Driscoll, P. Koanantakool, E. Georganas, and K. Keutzer. Pycasp: Scalable multimedia content analysis on parallel platforms using python. *TOMCCAP - ACM Transactions on Multimedia Computing, Communications and Applications*, 2013.

[6] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 1–12, 2010.

[7] S. Kamil, D. Coetzee, and A. Fox. Bringing parallel performance to python with domain-specific selective embedded just-in-time specialization. In *10th Python in Science Conference (SciPy 2011)*, Austin, TX, July 2011.

[8] A. Prasad, D. Howard, S. Kamil, and A. Fox. Parallel high performance bootstrapping in python. In *Proc. Eleventh Annual Scientific Computing with Python*, 2012.

[9] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: a high-performance java dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, 1998.

[10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

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

Shoaib Kamil, Derrick Coetzee, and Armando Fox

This work originally appeared as "Bringing Parallel Performance to Python with Domain-Specific Selective Embedded Just-in-Time Specialization" by Shoaib Kamil, Derrick Coetzee, and Armando Fox, for the *10th Python for Scientific Computing Conference*, 2011. Reprinted by permission of the authors.

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 [18]) 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.

1 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 [13] and FFTW [8], by domain-specific languages like SPIRAL [19], and by the popularity of the natively-compiled SciPy [18] 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 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 [1] 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 $5\times$. [17] Indeed, the complex internal structure of auto-tuning libraries such as the Optimized Sparse Kernel Interface [13] 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) [4]. 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 stencils, one optimization called *time skewing* [22] 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.

2 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.

One of Asp's primary purposes is separating application and algorithmic logic from code required to make the

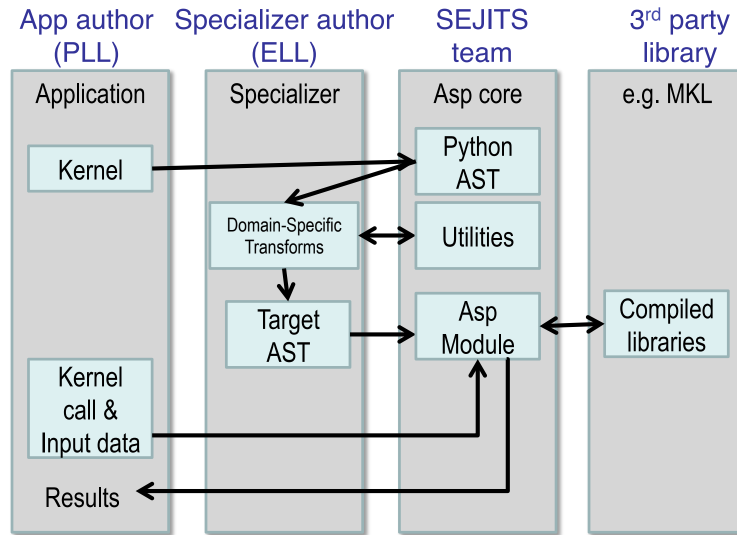

Figure 1: Separation of concerns in Asp. App authors write code that is transformed by specializers, using Asp infrastructure and third-party libraries.

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 [5]. 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.

2.1 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 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.

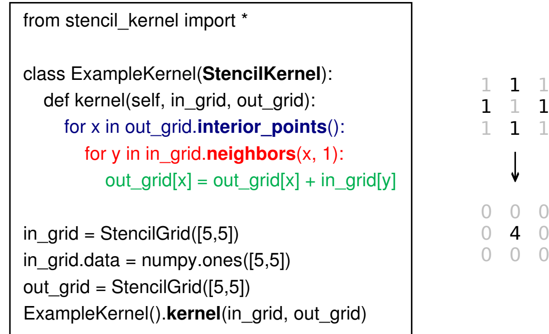

Figure 2: Example stencil application. Colored source lines match up to nodes of same color in Figure 4.

2.2 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 [12], 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.

3 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

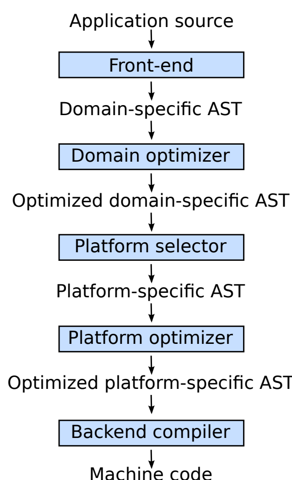

Figure 3: Pipeline architecture of a specializer.

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.
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 in Figure 5. The Asp framework provides

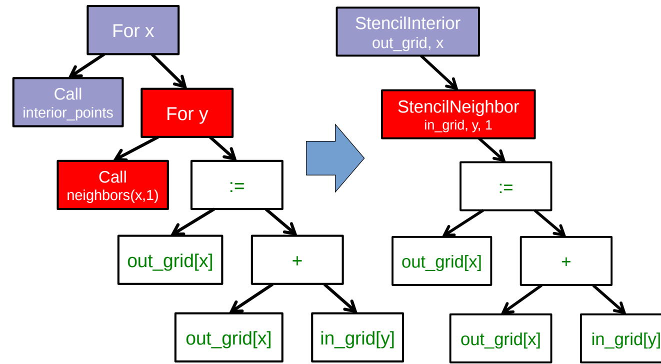

Figure 4: Left: Initial Python abstract syntax tree. Right: Domain-specific AST.

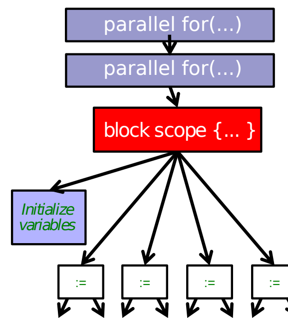

Figure 5: Platform-specific AST.

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.

4 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

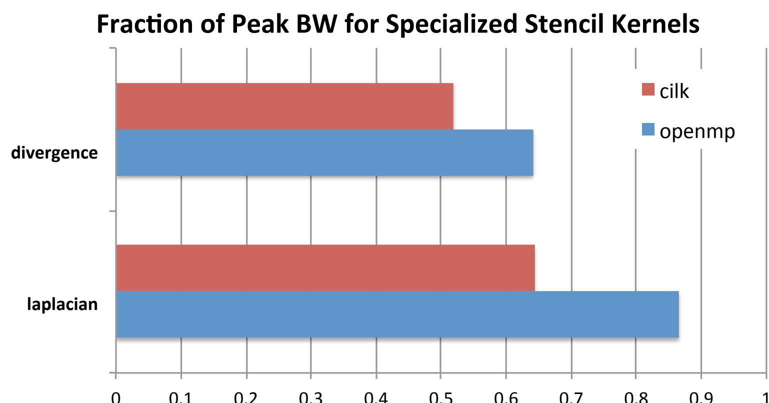

Figure 6: Performance as fraction of memory bandwidth peak for two specialized stencil kernels. All tests compiled using the Intel C++ compiler 12.0 on a Core i7-840.

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.

4.1 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 [16], we measure performance compared to measured memory bandwidth performance using the parallel STREAM [20] 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 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 [11], 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.

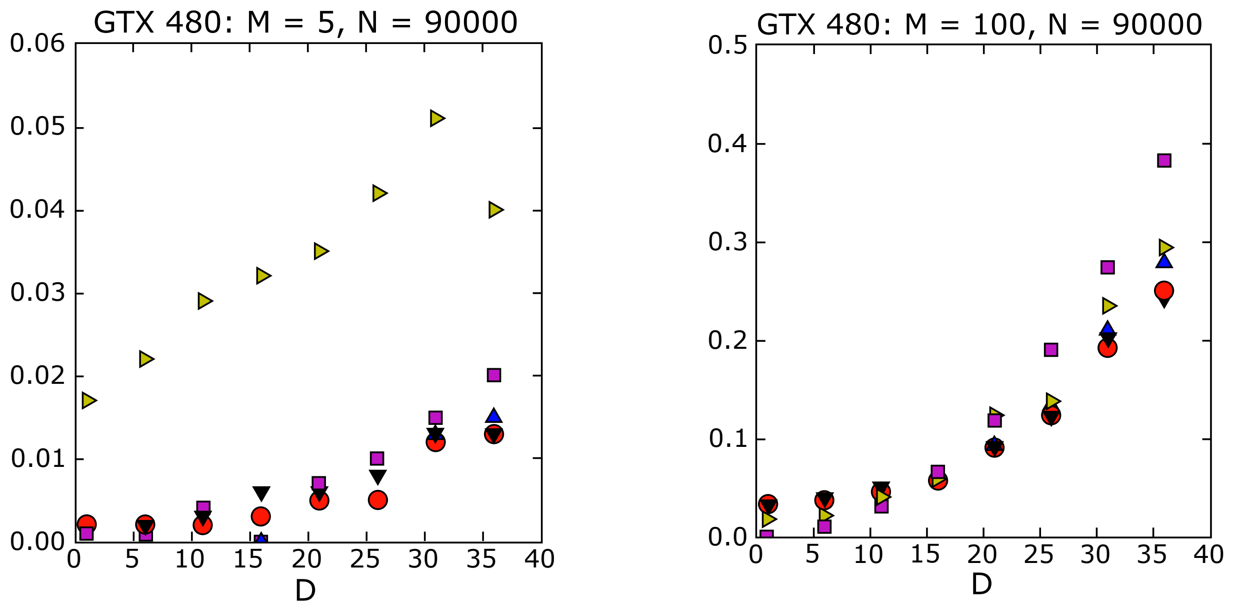

Figure 7: Runtimes of GMM variants as the D parameter is varied on an Nvidia Fermi GPU (lower is better). The specializer picks the best-performing variant to run.

4.2 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 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 [6] 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.

4.3 Matrix Powers

Recent developments in communication-avoiding algorithms [2] 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 [9]: the computation $\{x, Ax, A^2x, ..., A^kx\}$ 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 [9] 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

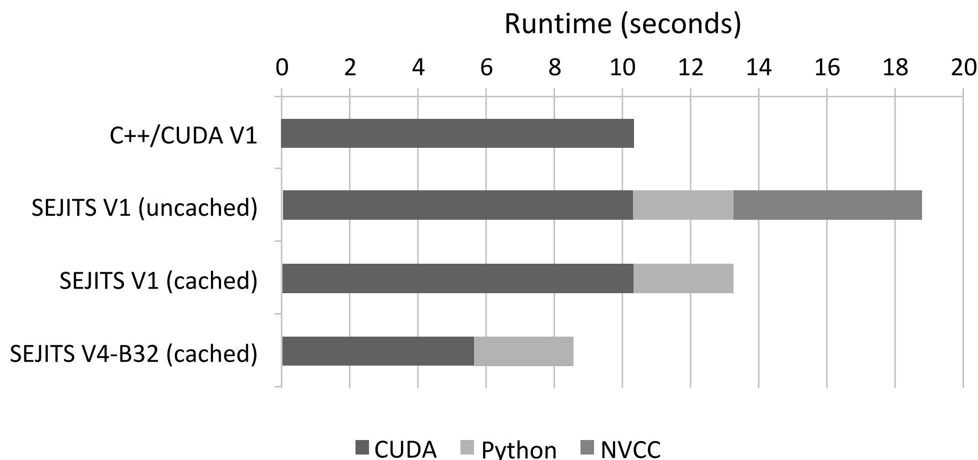

Figure 8: Overall performance of specialized GMM training versus original optimized CUDA algorithm. Even including specializer overhead, the specialized EM training outperforms the original CUDA implementation.

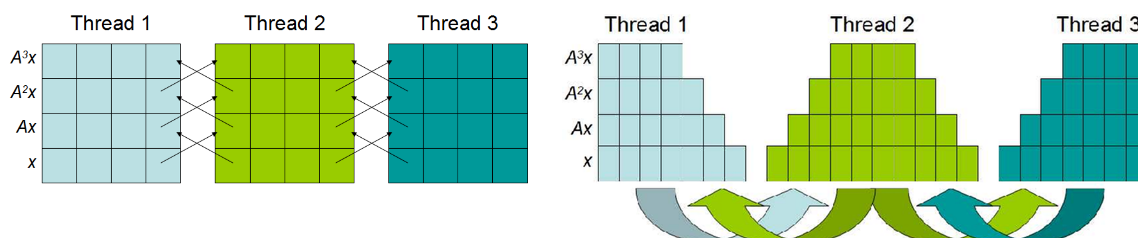

Figure 9: Left: Naive $A^k x$ computation. Communication required at each level. Right: Algorithm PA1 for communication-avoiding matrix powers. Communication occurs only after k levels of computation, at the cost of redundant computation.

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.

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.

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.

5 Related Work

Allowing domain scientists to program in higher-level languages is the goal of a number of projects in Python, including SciPy [18] which brings Matlab-like functionality for numeric computations into Python. In addition, domain-specific projects such as Biopython [3] and the Python Imaging Library (PIL) [15] 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 [7] 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 [4], 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

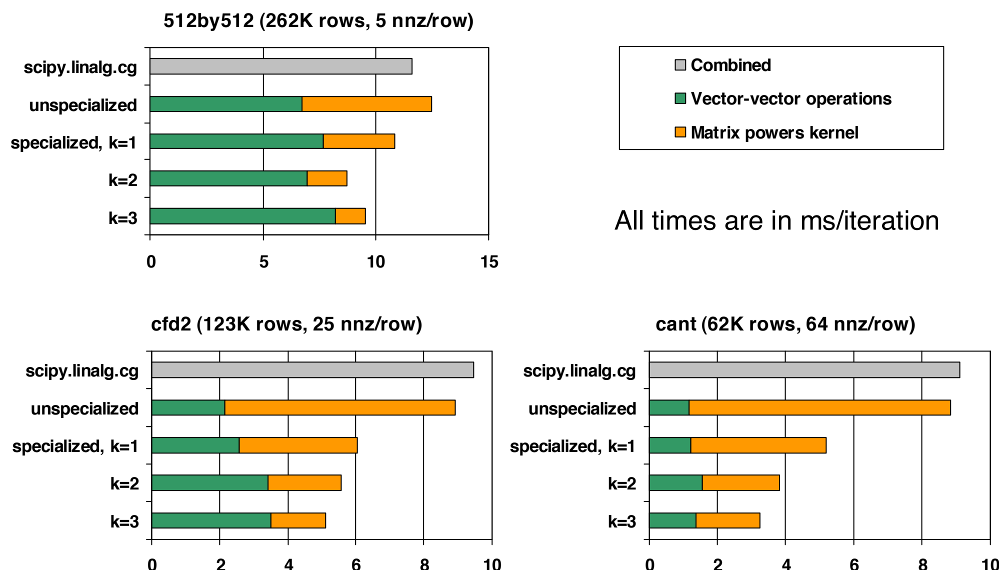

Figure 10: Results comparing communication-avoiding CG with our matrix powers specializer and SciPy's default solver, run on an Intel Nehalem machine.

auto-tuning. Auto-tuning was first applied to dense matrix computations in the PHiPAC (Portable High Performance ANSI C) library [14]. 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) [13], Fast Fourier Transforms (FFTs) [19], and multicore versions of stencils [10], [11], [21], showing large improvements in performance over simple implementations of these kernels.

6 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.

7 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.

Bibliography

[1] 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.

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

[3] Biopython. `http://biopython.org`.

[4] 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

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

[6] 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.

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

[8] 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.

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

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

[11] 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.

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

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

[14] 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.

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

[16] 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.

[17] 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.

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

[19] 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".

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

[21] 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.

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

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

Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, Krste Asanović, James Demmel, Kurt Keutzer, John Shalf, Kathy Yelick, Armando Fox

This work originally appeared as "SEJITS: Getting Productivity and Performance with Selective Embedded JIT Specialization" by Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, Krste Asanovic, James Demmel, Kurt Keutzer, John Shalf, Kathy Yelick, and Armando Fox, for the Workshop on Programming Models for Emerging Architectures (PMEA), 2009. Reprinted by permission of the authors.

Abstract

Today's "high productivity" programming languages such as Python lack the performance of harder-to-program "efficiency" languages (CUDA, Cilk, C with OpenMP) that can exploit extensive programmer knowledge of parallel hardware architectures. We combine efficiency-language performance with productivity-language programmability using *selective embedded just-in-time specialization* (SEJITS). At runtime, we *specialize* (generate, compile, and execute efficiency-language source code for) an application-specific and platform-specific subset of a productivity language, largely invisibly to the application programmer. Because the specialization machinery is implemented in the productivity language itself, it is easy for efficiency programmers to incrementally add specializers for new domain abstractions, new hardware, or both. SEJITS has the potential to bridge productivity-layer research and efficiency-layer research, allowing domain experts to exploit different parallel hardware architectures with a fraction of the programmer time and effort usually required.

1 Motivation

With the growing interest in computational science, more programming is done by experts in each application domain instead of by expert programmers. These domain experts increasingly turn to scripting languages and domain-specific languages, such as Python and MATLAB, which emphasize programmer productivity over hardware efficiency. Besides offering abstractions tailored to the domains, these *productivity-level languages* (PLLs) often provide excellent facilities for debugging and visualization. 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 3–10× fewer lines of code and in $1/5$ to $1/3$ the development time [17, 4, 8].

Although PLLs support rapid development of initial working code, they typically make inefficient use of underlying hardware and provide insufficient performance for large problem sizes. This performance gap is amplified by the recent move towards parallel processing [1], where today's multicore CPUs and manycore graphics processors require careful low-level orchestration to attain reasonable efficiency. 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*

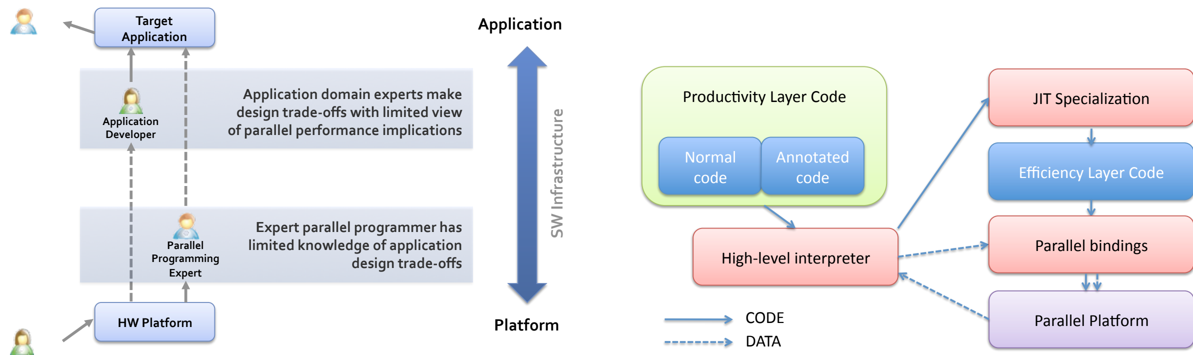

Figure 1: Left: Implementation gap between productivity-level languages (PLL) and efficiency-level languages (ELL). Right: Specialized embedded just-in-time specialization (SEJITS) schematic workflow.

between high-level domain abstractions and hardware targets, as depicted in Figure 1. This implementation gap not only already a problem, but is further widening. Domains are specializing into sub-disciplines, and available target hardware is becoming more heterogeneous, with hyperthreaded multicore, manycore GPUs, 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. We propose the use of *just-in-time specialization* of PLL code, where we dynamically generate source code in an ELL within the context of a PLL interpreter. Unlike most conventional JIT approaches, our JIT specialization machinery is *selective*, allowing us to pay the overhead of runtime specialization only where performance can be improved significantly, leaving the rest of the code in the PLL. Further, our JIT machinery is *embedded* in the PLL itself, making it easy to extend and add new specializers, while taking advantage of PLL libraries and infrastructure. Selective embedding of specialized JITs supports the rapid development and evolution of a collection of efficient code specializers, made accessible to domain experts through domain-appropriate abstractions.

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 additional benefits 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 [12] 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 [16] 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 therefore sidestep the difficult question of whether PLL language constructs outside the specialized subset can be JIT-ed 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 PLL provides both better productivity-level abstractions and simpler efficiency-level implementations. Modern PLL features such as iterators, abstract classes, and metaprogramming allow specialized abstractions to appear to the domain expert as language extensions or mini-embedded-DSLs [7] rather than as procedural libraries.

Embedding also helps the implementers of the specialized JITs, because the specialization machinery is implemented in the PLL itself by exploiting modern PLL features, as described 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.

3 Case Studies

Our SEJITS approach is most easily illustrated by example. We have prototyped two specializers, one in Python and one in Ruby, both of which rely on the introspection features of modern PLLs to perform specialization, and we have tested our approach on real problems from high-performance computing and computer vision. These 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 on parallel 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, 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 computes a new value for each point in a 3D grid based on the values of its 7 nearest neighbors.

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 [12] code for execution on Nvidia Graphics Processors. In both case studies, the introspection and function-interposition features of the PLLs effect specialization, including using information about the actual arguments at runtime to generate more efficient code.

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.

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 the grid instance. `JacobiKernel` provides the base class that the programmer subclasses to implement her own stencil kernel; the programmer overrides the `kernel` function[1], which accepts a pair of `StencilGrid` objects, to define the desired stencil computation. As the code excerpt in Figure 2 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.

[1]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 2, but this implementation artifact will soon be eliminated.

```
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 2: Example of a Laplacian kernel implemented in the Ruby stencil framework. Source in Ruby (left) passes through the specialization system to generate inlined C code (right). Note that the code defining neighbors is not shown.

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 code using the RubyParser library [20], 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 [14] 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 2.

Execution. Using RubyInline [19], 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, implemented as 7-pt stencils on a 3D grid. The stencils differ in which points are vector quantities and which are scalars; in each case, we use separate input and output grids. We ran these on both a 2.6 GHz Intel Nehalem (8 cores, with 2-way SMT for a total of 16 hardware threads) and a 2.3 GHz AMD Barcelona (8 cores). For comparison, we also ran handcrafted C+OpenMP versions of the three kernels using the StencilProbe [22] 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 case studies 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 [12]. 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

```
@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 3: 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.

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 [13]. 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 [13] 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.

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 100× 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 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× longer 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 performance boost came from architectural improvements (e.g. a better memory coalescer).

Table 1: Performance results (all times in seconds) comparing SEJITS vs. handcrafted ELL code. Ruby results reflect 10 iterations of the stencil ("inner loop").

Language & computation	CPU, #cores	Hand coded	SEJITS	Slow-down	Specialization Overhead		Execution Overhead	
Ruby Laplacian	Barcelona, 8	0.740	0.993	1.34	0.250	(25%)	0.003	(0.3%)
Ruby Laplacian	Nehalem, 8	0.219	0.614	2.80	0.271	(44%)	0.120	(20.2%)
Ruby Divergence	Barcelona, 8	0.720	0.973	1.35	0.273	(28%)	0.000	(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%)	0.000	(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%)	0.020	(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	7.226	3.15	2.470	(34%)	2.462	(34.1%)
Python Textons	C1060, 30	0.477	5.779	12.12	3.224	(56%)	2.077	(35.9%)
Python Localcues	GX2, 16	0.565	5.757	10.19	2.600	(45%)	2.592	(45.0%)
Python Localcues	C1060, 30	0.263	3.323	12.63	2.235	(67%)	0.825	(24.8%)

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. 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 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 parallel primitives are currently not optimized, which is why the Localcues and Texton computation runs $3 - -12\times$ slower with SEJITS than handcoded CUDA. For example, the read-only input data for a stencil filter could be stored in the GPU's texture cache, eliminating copying of intermediate data between filter steps. As another example, the implementation strategy for parallel category reduction on CUDA depends strongly on the parameters of the particular reduction: for large numbers of categories, our handcrafted CUDA code uses atomic memory transactions to on-chip memory structures to deal with bin contention. As well, the size of data being accumulated dictates how intermediate reduction data is mapped to the various GPU on-chip memory structures.

Although we have experience hand-coding such scenarios, we have not yet incorporated this knowledge into the specializer, though all the necessary information is available to the SEJITS framework at runtime. Our broader vision is that specialization allows these details to be encapsulated in specializers to enable runtime generation of efficient code.

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 an hours to write with SEJITS, compared to a day 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 a significant opportunity. For example, even handling the thirteen computational "motifs" that recur in many applications [1] would be a productive step forward. Here we discuss the opportunities and challenges of pursuing such a path.

4.1 Why Now?

In 1998, John Ousterhout [15] 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 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 3–10× as many lines to express than in typical scripting languages [17]. Popular domain-specific languages like MATLAB and R have weak glue facilities and introspection; while JIT specialization could 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. In Python we used the 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.

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 [16] perform runtime profiling to decide which functions are worth the overhead of JIT-ing, 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 [18], 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 [23], 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 [21] for ideas on how to optimize data structure composition, placement, and movement.

Approaches to building PLLs. Domain-specific languages (DSLs) have long been used to improve domain-expert programmer productivity, but a complete toolchain from DSL down to the hardware makes DSLs expensive to build and modify. As an alternative, Hudak and others [7] proposed Domain-Specific Embedded Languages (DSELs), an approach in which the DSL is implemented within the constructs provided by some host language. This embedding allows the productivity programmer fall back to host-language code when a construct is unavailable in the DSEL and also makes the DSEL more easily evolvable as the domain evolves. Our motivations for embedding the specializer machinery in the PLL are analogous to that for DSELs: non-specializable functions can be executed in the PLL, and extending the system with new specializers is easy since it doesn't require going "outside" the PLL interpreter.

Using productivity languages for high-performance computing. Python in particular is garnering a rapidly-growing scientific computing community. Of the many efforts to improve Python's performance for scientific computing, the most closely related to our work are Cython and Weave. Cython (cython.org) allows annotating Python source with additional keywords giving static type information to a C compiler. In effect, the programmer promises not to use

certain dynamic Python language features on certain objects; Cython compiles the program exploiting this information to speed up inner loops. However, a "Cythonized" Python program can no longer be executed by a standard Python interpreter. Another approach is the Weave subpackage of SciPy/NumPy, which focuses on easy inlining of C/C++ code into Python functions rather than integration of entire C/C++ libraries with Python. In both approaches, the specific optimizations are visible directly in the application logic. In contrast, our goal is to keep the application logic free of such considerations. Furthermore, in general the existing efforts do not attempt a framework for transparent and retargetable specialization, though they do provide some machinery that may facilitate our future efforts extending the SEJITS approach.

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 (PLLs) at which domain experts would like to program and the efficiency-level languages (ELLs) one *must* use to get performance 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.

7 Acknowledgements

In addition to our ParLab colleagues, we thank Jim Larus, Yannis Smaragdakis, and Fernando Perez for valuable feedback on earlier drafts 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).

Bibliography

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

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

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

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

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

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

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

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

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

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

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

[23] 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.

Copperhead: Compiling an Embedded Data Parallel Language

Bryan Catanzaro, Michael Garland, and Kurt Keutzer

© 2011 Association for Computing Machinery, Inc. Reprinted by permission. This work originally appeared as "Copperhead: Compiling an Embedded Data Parallel Language" by Bryan Catanzaro, Michael Garland, and Kurt Keutzer, in the PPoPP '11 *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, Pages 47-56 . Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. DOI: 10.1145/1941553.1941562

Abstract

Modern parallel microprocessors deliver high performance on applications that expose substantial fine-grained data parallelism. Although data parallelism is widely available in many computations, implementing data parallel algorithms in low-level languages is often an unnecessarily difficult task. The characteristics of parallel microprocessors and the limitations of current programming methodologies motivate our design of Copperhead, a high-level data parallel language embedded in Python. The Copperhead programmer describes parallel computations via composition of familiar data parallel primitives supporting both flat and nested data parallel computation on arrays of data. Copperhead programs are expressed in a subset of the widely used Python programming language and interoperate with standard Python modules, including libraries for numeric computation, data visualization, and analysis.

In this paper, we discuss the language, compiler, and runtime features that enable Copperhead to efficiently execute data parallel code. We define the restricted subset of Python which Copperhead supports and introduce the program analysis techniques necessary for compiling Copperhead code into efficient low-level implementations. We also outline the runtime support by which Copperhead programs interoperate with standard Python modules. We demonstrate the effectiveness of our techniques with several examples targeting the CUDA platform for parallel programming on GPUs. Copperhead code is concise, on average requiring 3.6 times fewer lines of code than CUDA, and the compiler generates efficient code, yielding 45-100% of the performance of hand-crafted, well optimized CUDA code.

1 Introduction

As the transition from sequential to parallel processing continues, the need intensifies for high-productivity parallel programming tools and methodologies. Manual implementation of parallel programs using efficiency languages such as C and C++ can yield high performance, but at a large cost in programmer productivity, which some case studies show as being 2 to 5 times worse than productivity-oriented languages such as Ruby and Python [15, 25]. Additionally, parallel programming in efficiency languages is often viewed as an esoteric endeavor, to be undertaken by experts only. Without higher-level abstractions that enable easier parallel programming, parallel processing will not be widely utilized. Conversely, today's productivity programmers do not have the means to capitalize on fine-grained, highly parallel microprocessors, which limits their ability to implement computationally intensive applications. To enable widespread use of parallel processors, we need to provide higher level abstractions which are both productive and efficient.

Although the need for higher-level parallel abstractions seems clear, perhaps less so is the type of abstractions which should be provided, since there are many potential abstractions to choose from. In our view, nested data parallelism, as introduced by languages such as NESL [4], is particularly interesting. Nested data parallelism is

abundant in many computationally intensive algorithms. It can be mapped efficiently to parallel microprocessors, which prominently feature hardware support for data parallelism. For example, mainstream x86 processors from Intel and AMD are adopting 8-wide vector instructions, Intel's Larrabee processor used 16-wide vector instructions, and modern GPUs from NVIDIA and AMD use wider SIMD widths of 32 and 64, respectively. Consequently, programs which don't take advantage of data parallelism relinquish substantial performance, on any modern processor.

Additionally, nested data parallelism as an abstraction clearly exposes parallelism. In contrast to traditional auto-parallelizing compilers, which must analyze and prove which operations can be parallelized and which can not, the compiler of a data-parallel language needs only to decide which parallel operations should be performed sequentially, and which should be performed in parallel. Accordingly, nested data parallel programs have a valid sequential interpretation, and are thus easier to understand and debug than parallel programs that expose race conditions and other complications of parallelism.

Motivated by these observations, Copperhead provides a set of nested data parallel abstractions, expressed in a restricted subset of the widely used Python programming language. Instead of creating an entirely new programming language for nested data parallelism, we repurpose existing constructs in Python, such as `map` and `reduce`. Embedding Copperhead in Python provides several important benefits. For those who are already familiar with Python, learning Copperhead is more similar to learning how to use a Python package than it is to learning a new language. There is no need to learn any new syntax, instead the programmer must learn only what subset of Python is supported by the Copperhead language and runtime. The Copperhead runtime is implemented as a standard Python extension, and Copperhead programs are invoked through the Python interpreter, allowing them to interoperate with the wide variety of existing Python libraries for numeric computation, file manipulation, data visualization, and analysis. This makes Copperhead a productive environment for prototyping, debugging, and implementing entire applications, not just their computationally intense kernels.

Of course, without being able to efficiently compile nested data parallel programs, Copperhead would not be of much use. To this end, the Copperhead compiler attempts to efficiently utilize modern parallel processors. Previous work on compilers for nested data parallel programming languages has focused on flattening transforms to target flat SIMD arrays of processing units. However, today's processors support a hierarchy of parallelism, with independent cores containing tightly coupled processing elements. Accordingly, the Copperhead compiler maps nested data parallel programs to a parallelism hierarchy, forgoing the use of flattening transforms. We find that in many circumstances, this is substantially more efficient than indiscriminate application of the flattening transform.

The current Copperhead compiler targets CUDA C++, running on manycore Graphics Processors from NVIDIA. We generate efficient, scalable parallel programs, performing within 45-100% of well optimized, hand-tuned CUDA code. Our initial implementation focus on CUDA does not limit us to a particular hardware platform, as the techniques we have developed for compiling data-parallel code are widely applicable to other platforms as well. In the future, we anticipate the development of Copperhead compiler backends for other parallel platforms. Additionally, the compiler techniques we propose are suitable for compiling other data parallel languages, such as NESL and Data Parallel Haskell.

2 Related Work

There is an extensive literature investigating many alternative methods for parallel programming. Data parallel approaches [2, 3, 13] have often been used, and have historically been most closely associated with SIMD and vector machines, such as the the CM-2 and Cray C90, respectively. Blelloch *et al.* designed the NESL language [4], demonstrating a whole-program transformation of nested data parallel programs into flat data parallel programs. The flattening transform has been extended to fully higher-order functional languages in Data Parallel Haskell [7]. In contrast to these methods, we attempt to schedule nested procedures directly onto the hierarchical structure of the machines we target.

The CUDA platform [23, 26] defines a blocked SPMD programming model for executing parallel computations on GPUs. Several libraries, such as Thrust [14], provide a collection of flat data parallel primitives for use in CUDA programs. Copperhead uses selected Thrust primitives in the code it generates.

Systems for compiling flat data parallel programs for GPU targets have been built in a number of languages, including C# [28], C++ [22, 21], and Haskell [19]. Such systems typically define special data parallel array types and use operator overloading and metaprogramming techniques to build expression trees describing the computation to be performed on these arrays. The Ct [11] library adopts a similar model for programming multicore processors. How-

ever, these systems have not provided a means to automatically map nested data parallel programs to a hierarchically nested parallel platform.

Rather than providing data parallel libraries, others have explored techniques that mark up sequential loops to be parallelized into CUDA kernels [12, 20, 30]. In these models, the programmer writes an explicitly sequential program consisting of loops that are parallelizable. The loop markups, written in the form of C pragma preprocessor directives, indicate to the compiler that loops can be parallelized into CUDA kernels.

There have also been some attempts to compile various subsets of Python to different platforms. The Cython compiler [9] compiles a largely Python-like language into sequential C code. Clyther [8] takes a similar approach to generating OpenCL kernels. In both cases, the programmer writes a Python program which is transliterated into an isomorphic C program. Rather than transliterating Python programs, PyCUDA [18] provides a metaprogramming facility for textually generating CUDA kernel programs, as well as Python/GPU bindings. Theano [29] provides an expression tree facility for numerical expressions on numerical arrays. Garg and Amaral [10] recently described a technique for compiling Python loop structures and array operations into GPU-targeted code.

These Python/GPU projects are quite useful; in fact Copperhead's runtime relies on PyCUDA. However, to write programs using these tools, programmers must write code equivalent to the *output* of the Copperhead compiler, with all mapping and scheduling decisions made explicit in the program itself. Copperhead aims to solve a fairly different problem, namely compiling a program from a higher level of abstraction into efficiently executable code.

3 Copperhead Language

A Copperhead program is a Python program that imports the Copperhead language environment:

```
from copperhead import *
```

A Copperhead program is executed, like any other Python program, by executing the sequence of its top-level statements. Selected procedure definitions within the body of the program may be marked with the Copperhead decorator, as in the following:

```
@cu
def add_vectors(x, y):
    return map(lambda xi,yi: xi+yi, x, y)
```

This `@cu` decorator declares the associated procedure to be a Copperhead procedure. These procedures must conform to the requirements of the Copperhead language, and they may be compiled for and executed on any of the parallel platforms supported by the Copperhead compiler. Once defined, Copperhead procedures may be called just like any other Python procedure, both within the program body or, as shown below, from an interactive command line.

```
>>> add_vectors(range(10), [2]*10)
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

The `@cu` decorator interposes a wrapper object around the native Python function object that is responsible for compiling and executing procedures on a target parallel platform.

Copperhead is fundamentally a data parallel language. It provides language constructs, specifically `map`, and a library of primitives such as `reduce`, `gather`, and `scatter` that all have intrinsically parallel semantics. They operate on 1-dimensional arrays of data that we refer to as *sequences*.

3.1 Restricted Subset of Python

The Copperhead language is a restricted subset of the Python 2.6 language. Every valid Copperhead procedure must also be a valid Python procedure. Portions of the program outside procedures marked with the `@cu` decorator are normal Python programs that will be executed by the standard Python interpreter, without any restrictions imposed by the Copperhead runtime or compiler. The Copperhead language supports a very restricted subset of Python in order to enable efficient compilation.

Copperhead adopts the lexical and grammatical rules of Python. In summarizing its restricted language, we denote expressions by E and statements by S. We use lower-case letters to indicate identifiers and F and A to indicate function-valued and array-valued expressions, respectively.

3.1.1 Expressions

The most basic expressions are literal values, identifier references, tuple constructions, and accesses to array elements.

$$
\begin{aligned}
E \;:\; & x \;\mid\; (E_1,\ \ldots,\ E_n) \;\mid\; A[E] \\
& \mid\; \textbf{True} \;\mid\; \textbf{False} \;\mid\; \texttt{integer} \;\mid\; \texttt{floatnumber}
\end{aligned}
$$

The basic logical and arithmetic operators are allowed, as are Python's `and`, `or`, and conditional expressions.

$$
\begin{aligned}
& \mid\; E_1 + E_2 \;\mid\; E_1 < E_2 \;\mid\; \textbf{not}\ E \;\mid\; \ldots \\
& \mid\; E_1\ \textbf{and}\ E_2 \;\mid\; E_1\ \textbf{or}\ E_2 \;\mid\; E_1\ \textbf{if}\ E_p\ \textbf{else}\ E_2
\end{aligned}
$$

Expressions that call and define functions must use only positional arguments. Optional and keyword arguments are not allowed.

$$
\mid\; F(E_1,\ \ldots,\ E_n) \;\mid\; \textbf{lambda}\ x_1,\ \ldots,\ x_n\texttt{:}\ E
$$

Copperhead relies heavily on `map` as the fundamental source of parallelism and elevates it from a built-in function (as in Python) to a special form in the grammar. Copperhead also supports a limited form of Python's list comprehension syntax, which is de-sugared into equivalent `map` constructions during parsing.

$$
\begin{aligned}
& \mid\; \textbf{map}(F,\ A_1,\ \ldots,\ A_n) \\
& \mid\; [E\ \textbf{for}\ x\ \textbf{in}\ A] \\
& \mid\; [E\ \textbf{for}\ x_1,\ \ldots,\ x_n\ \textbf{in}\ \textbf{zip}(A_1,\ \ldots,\ A_n)]
\end{aligned}
$$

In addition to these grammatical restrictions, Copperhead expressions must be *statically well-typed*. We employ a standard Hindley-Milner style polymorphic type system. Static typing provides richer information to the compiler that it can leverage during code generation, and also avoids the run-time overhead of dynamic type dispatch.

3.1.2 Statements

The body of a Copperhead procedure consists of a *suite* of statements: one or more statements S which are nested by indentation level. Each statement S of a suite must be of the following form:

$$
\begin{aligned}
S \;:\; & \textbf{return}\ E \\
& \mid\; x_1,\ \ldots,\ x_n\ \texttt{=}\ E \\
& \mid\; \textbf{if}\ E\texttt{:}\ \texttt{suite}\ \textbf{else}\texttt{:}\ \texttt{suite} \\
& \mid\; \textbf{def}\ f(x_1,\ \ldots,\ x_n)\texttt{:}\ \texttt{suite}
\end{aligned}
$$

Copperhead further requires that every execution path within a suite must return a value, and all returned values must be of the same type. This restriction is necessary since every Copperhead procedure must have a well-defined static type.

All data in a Copperhead procedure are considered immutable values. Thus, statements of the form `x = E` bind the value of the expression `E` to the identifier `x`; they *do not* assign a new value to an existing variable. All identifiers are strictly lexically scoped.

Copperhead does not guarantee any particular order of evaluation, other than the partial ordering imposed by data dependencies in the program. Python, in contrast, always evaluates statements from top to bottom and expressions from left to right. By definition, a Copperhead program must be valid regardless of the order of evaluations, and thus Python's mandated ordering is one valid ordering of the program.

Because mutable assignment is forbidden and the order of evaluation is undefined, the Copperhead compiler is free to reorder and transform procedures in a number of ways. As we will see, this flexibility improves the efficiency of generated code.

3.2 Data Parallel Primitives

Copperhead is a data parallel language. Programs manipulate data sequences by applying aggregate operations, such as `map`, or `reduce`. The semantics of these primitives are implicitly parallel: they may always be performed by some parallel computation, but may also be performed sequentially.

Table 1 summarizes the main aggregate operations used by the examples in this paper. They mirror operations found in most other data parallel languages. With two minor exceptions, these functions will produce the same

Table 1: Selected data-parallel operations on sequences.

`y = replicate(a, n)`	Return an `n`-element sequence whose every element has value `a`.
`y = map(f, x1, ..., xn)`	Returns sequence `[f(x1[0], ..., xn[0]), f(x1[1], ..., xn[1]), ...]`.
`y = zip(x1, x2)`	Return sequence of pairs `[(x1[0],x2[0]), (x1[1],x2[1]), ...]`.
`y = gather(x, indices)`	Produce sequence with `y[i] = x[indices[i]]`.
`y = permute(x, indices)`	Produce `y` where `y[indices[i]] = x[i]`.
`s = reduce(⊕, x, prefix)`	Computes `prefix⊕x[0]⊕x[1]⊕...` for a commutative and associative operator ⊕.
`y = scan(f, x)`	Produce `y` such that `y[0]=x[0]` and `y[i] = f(y[i-1], x[i])`. Function `f` must be associative.

```
@cu
def spmv_csr(vals, cols, x):
    def spvv(Ai, j):
        z = gather(x, j)
        return sum(map(lambda Aij, xj: Aij*xj, Ai, z))

    return map(spvv, vals, cols)
```

Figure 1: Procedure for computing Ax for a matrix A in CSR form and a dense vector x. Underlined operations indicate potential sources of parallel execution.

result regardless of whether they are performed in parallel or sequentially. The `permute` primitive is the primary exception. If the `indices` array given to `permute` contains one or more repeated index values, the resulting sequence is non-deterministic since we guarantee no particular order of evaluation. The other exception is `reduce`. If given a truly commutative and associative operation, then its result will always be identical. However, programs often perform reductions using floating point values whose addition, for instance, is not truly associative. These problems are commonly encountered in general parallel programming, and are not unique to Copperhead.

To demonstrate our use of these operators, Figure 1 shows a simple Copperhead procedure for computing the sparse matrix-vector product (SpMV) $y = Ax$. Here we assume that A is stored in Compressed Sparse Row (CSR) format—one of the most frequently used representation for sparse matrices—and that x is a dense vector. The matrix representation simply records each row of the matrix as a sequence containing its non-zero values along with a corresponding sequence recording the column index of each value. A simple example of this representation is:

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

```
vals = [[1,7],[2,8],[5,3,9],[6,4]]
cols = [[0,1],[1,2],[0,2,3],[1,3]]
```

The body of `spmv_csr` applies a sparse dot product procedure, `spvv`, to each row of the matrix using `map`. The sparse dot product itself produces the result y_i for row i by forming the products $A_{ij}x_j$ for each column j containing a non-zero entry, and then summing these products together. It uses `gather` to fetch the necessary values x_j from the dense vector x and uses `sum`, a convenient special case of `reduce` where the operator is addition, to produce the result.

```
# lambda0 :: (a, a) → a
def lambda0(Aij, xj):
    return op_mul(Aij, xj)

# spvv :: ([a], [Int], [a]) → [a]
def spvv(Ai, j, _k0):
    z0 = gather(_k0, j)
    tmp0 = map(lambda0, Ai, z0)
    return sum(tmp0)

# spmv_csr :: ([[a]], [[Int]], [a]) → [a]
def spmv_csr(vals, cols, x):
    return map(closure([x], spvv), vals, cols)
```

Figure 2: SpMV procedure from Figure 1 after transformation by the front end compiler.

This simple example illustrates two important issues that are a central concern for our Copperhead compiler. First, it consists of a number of potentially parallel aggregate operations, which are underlined. Second, these aggregate operators are nested: those within `spvv` are executed within an enclosing `map` operation invoked by `spmv_csr`. One of the principal tasks of our compiler is to decide how to schedule these operations for execution. Each of them may be performed in parallel or by sequential loops, and the nesting of operations must be mapped onto the hardware execution units in an efficient fashion.

4 Compiler

The Copperhead compiler is a source-to-source compiler responsible for converting a suite of one or more Copperhead procedures into a code module that can execute on the target parallel platform. We have designed it to support three basic usage patterns. First is what we refer to as JIT (Just-In-Time) compilation. When the programmer invokes a `@cu`-decorated function either from the command line or from a Python code module, the Copperhead runtime may need to generate code for this procedure if none is already available. Second is batch compilation, where the Copperhead compiler is asked to generate a set of C++ code modules for the specified procedures. This code may be subsequently used either within the Copperhead Python environment or linked directly with external C++ applications. The third common scenario is one where the compiler is asked to generate a collection of variant instantiations of a Copperhead procedure in tandem with an autotuning framework for exploring the performance landscape of a particular architecture.

In the following discussion, we assume that the compiler is given a single top level Copperhead function—referred to as the "entry point"—to compile. It may, for instance, be a function like `spmv_csr` that has been invoked at the Python interpreter prompt. For the CUDA platform, the compiler will take this procedure, along with any procedures it invokes, and produce a single sequential *host* procedure and one or more parallel *kernels* that will be invoked by the host procedure.

4.1 Front End

After parsing the program text using the standard Python `ast` module, the compiler front end converts the program into a simplified abstract syntax tree (AST) form suitable for consumption by the rest of the compiler. It applies a sequence of fairly standard program transformations to: lift all nested `lambda`s and procedures into top level definitions, make closures explicit, convert all statements to single assignment form with no nested expressions, and infer static types for all elements of the program. Programs that contain syntax errors or that do not conform to the restricted language outlined in Section 3 are rejected.

For illustration, Figure 2 shows a program fragment representing the AST for the Copperhead procedure shown in Figure 1. Each procedure is annotated with its most general polymorphic type, which we determine using a standard Hindley-Milner style type inference process. Lexically captured variables within nested procedures are eliminated by converting free variables to explicit arguments—shown as variables `_k0`, `_k1`, etc.—and replacing the original

nested definition point of the function with a special `closure([`k_1`, ..., `k_n`], f)` form where the k_i represent the originally free variables. In our example, the local variable `x` was free in the body of the nested `spvv` procedure, and is thus converted into an explicit closure argument in the transformed code. Single assignment conversion adds a unique subscript to each local identifier (e.g., $\texttt{z}_0$ in `spvv`), and flattening nested expressions introduces temporary identifiers (e.g., $\texttt{tmp}_0$) bound to the value of individual sub-expressions. Note that our single assignment form has no need of the ϕ-functions used in SSA representations of imperative languages since we disallow loops and every branch of a conditional must return a value.

4.2 Scheduling Nested Parallelism

The front end of the compiler carries out platform independent transformations in order to prepare the code for scheduling. The middle section of the compiler is tasked with performing analyses and scheduling the program onto a target platform.

At this point in the compiler, a Copperhead program consists of possibly nested compositions of data parallel primitives. A Copperhead procedure may perform purely sequential computations, in which case our compiler will convert it into sequential C++ code. Copperhead makes *no attempt* to auto-parallelize sequential codes. Instead, it requires the programmer to use primitives that the compiler will *auto-sequentialize* as necessary. Our compilation of sequential code is quite straightforward, since we rely on the host C++ compiler to handle all scalar code optimizations, and our restricted language avoids all the complexities of compiling a broad subset of Python that must be addressed by compilers like Cython [9].

Copperhead supports both flat and nested data parallelism. A flat Copperhead program consists of a sequence of parallel primitives which perform purely sequential operations to each element of a sequence in parallel. A nested program, in contrast, may apply parallel operations to each sequence element. Our `spmv_csr` code provides a simple concrete example. The outer `spmv_csr` procedure applies `spvv` to every row of its input via `map`. The `spvv` procedure itself calls `gather`, `map`, and `sum`, all of which are potentially parallel. The Copperhead compiler must decide how to map these potentially parallel operations onto the target hardware: for example, they may be implemented as parallel function calls or sequential loops.

One approach to scheduling nested parallel programs, adopted by NESL [4] and Data Parallel Haskell [7] among others, is to apply a flattening (or vectorization) transform to the program. This converts a nested structure of vector operations into a sequence of flat operations. In most cases, the process of flattening replaces the nested operations with segmented equivalents. For instance, in our `spmv_csr` example, the nested `sum` would be flattened into a segmented reduction.

The flattening transform is a powerful technique which ensures good load balancing. It also enables data parallel compilation for flat SIMD processors, where all lanes in the SIMD array work in lockstep. However, we take a different approach. Flattening transformations are best suited to machines that are truly flat. Most modern machines, in contrast, are organized hierarchically. The CUDA programming model [24, 23], for instance, provides a hierarchy of execution formed into four levels:

1. A *host thread* running on the CPU, which can invoke
2. Parallel *kernels* running on the GPU, which consist of
3. A grid of *thread blocks* each of which is comprised of
4. Potentially hundreds of *device threads* running on the GPU.

The central goal of the Copperhead compiler is thus to map the nested structure of the program onto the hierarchical structure of the machine.

Recent experience with hand-written CUDA programs suggests that direct mapping of nested constructions onto this physical machine hierarchy often yields better performance. For instance, Bell and Garland [1] explored several strategies for implementing SpMV in CUDA. Their Coordinate (COO) kernel is implemented by manually applying the flattening transformation to the `spmv_csr` algorithm. Their other kernels represent static mappings of nested algorithms onto the hardware. The flattened COO kernel only delivers the highest performance in the exceptional case where the distribution of row lengths is extraordinarily variable, in which case the load balancing provided by the flattening transform is advantageous. However, for 13 out of the 14 unstructured matrices they examine, applying the flattening transform results in performance two to four times slower than the equivalent nested implementation.

Experiences such as this lead us to the conclusion that although the flattening transform can provide high performance in certain cases where the workload is extremely imbalanced, the decision to apply the transform should be under programmer control, given the substantial overhead the flattening transform imposes for most workloads.

Our compiler thus performs a static mapping of nested programs onto a parallelism hierarchy supported by the target parallel platform.

Returning to our `spmv_csr` example being compiled to the CUDA platform, the `map` within its body will become a CUDA kernel call. The compiler can then chose to map the operations within `spvv` to either individual threads or individual blocks. Which mapping is preferred is in general program and data dependent; matrices with very short rows are generally best mapped to threads while those with longer rows are better mapped to blocks.

Because this information is data dependent, we currently present the decision of which execution hierarchy to target as a compiler option. The programmer can arrange the code to specify explicitly which is preferred, an autotuning framework can explore which is better on a particular machine, or a reasonable default can be chosen based on any static knowledge about the problem being solved. In the absence of a stated preference, the default is to map nested operations to sequential implementations. This will give each row of the matrix to a single thread of the kernel created by the call to `map` in `spmv_csr`.

4.3 Synchronization Points and Fusion

The compiler must also discover and schedule synchronization points between aggregate operators. We assume that our target parallel platform provides a SPMD-like execution model, and therefore synchronization between a pair of parallel operations is performed via barrier synchronization across parallel threads. Similarly, a synchronization point between a pair of operators that have been marked as sequentialized implies that they must be implemented with separate loop bodies.

The most conservative policy would be to require a synchronization point following every parallel primitive. However, this can introduce far more synchronization, temporary data storage, and memory bandwidth consumption than necessary to respect data dependencies in the program. Requiring a synchronization point after every parallel primitive can reduce performance by an order of magnitude or more, compared with equivalent code that synchronizes only when necessary. In order to produce efficient code, our compiler must introduce as few synchronization points as possible, by fusing groups of parallel primitives into single kernels or loop bodies.

In order to determine where synchronization points are necessary, we need to characterize the data dependencies between individual elements of the sequences upon which the parallel primitives are operating. Consider an aggregate operator of the form $x = f(y_1, ..., y_n)$ where x and the y_i are all sequences. Since the elements of x may be computed concurrently, we need to know what values of y_i it requires.

We call the value `x[i]` *complete* when its value has been determined and may be used as input to subsequent operations. The array `x` is complete when all its elements are complete.

We categorize the bulk of our operators into one of two classes. The first class are those operators which perform induction over the domain of their inputs. This class is essentially composed of `permute`, `scatter`, and their variants. We cannot in general guarantee that any particular `x[i]` is complete until the entire operation is finished. In other words, either the array `x` is entirely complete or entirely incomplete.

The second class are those operators which perform induction over the domain of their outputs. This includes almost all the rest of our operators, such as `map`, `gather`, and `scan`. Here we want to characterize the portion of each y_j that must be complete in order to complete `x[i]`. We currently distinguish three broad cases:

- *None*: `x[i]` does not depend on any element of y_j
- *Local*: completing `x[i]` requires that y_j`[i]` be complete
- *Global*: completing `x[i]` requires that y_j be entirely complete

With deeper semantic analysis, we could potentially uncover more information in the spectrum between local and global completion requirements. However, at the moment we restrict our attention to this simple classification.

Each primitive provided by Copperhead declares its completion requirements on its arguments (none, local, global) as well as the completion of its output. Completing the output of class 1 operators requires synchronization, while the output of class 2 operators is completed locally. The compiler then propagates this information through the body of user-defined procedures to compute their completion requirements. For example, a synchronization point is required between a class 1 primitive and any point at which its output is used, or before any primitive that requires one of its

arguments to be globally complete. We call a sequence of primitives that are not separated by any synchronization points a *phase* of the program. All operations within a single phase may potentially be fused into a single parallel kernel or sequential loop by the compiler.

4.4 Shape Analysis

Shape analysis determines, where possible, the sizes of all intermediate values. This allows the back end of the compiler to statically allocate and reuse space for temporary values, which is critical to obtaining efficient performance. For the CUDA backend, forgoing shape analysis would require that every parallel primitive be executed individually, since allocating memory from within a CUDA kernel is not feasible. This would lead to orders of magnitude lower performance.

Our internal representation gives a unique name to every temporary value. We want to assign to each a *shape* of the form $\langle$`[`d_1`, ..., `d_n`], s`$\rangle$ where d_i is the array's extent in dimension i and `s` is the shape of each of its elements. Although Copperhead currently operates on one-dimensional sequences, the shape analysis we employ applies to higher dimensional arrays as well. The shape of a 4-element, 1-dimensional sequence `[5,4,8,1]` would, for example, be $\langle$`[4], Unit`$\rangle$ where `Unit=`$\langle$`[], ·`$\rangle$ is the shape reserved for indivisible types such as scalars. Nested sequences are not required to have fully determined shapes: in the case where subsequences have differing lengths, the extent along that dimension will be undefined, for example: $\langle$`[2], `$\langle$`[*], Unit`$\rangle\rangle$. Note that the dimensionality of all values are known as a result of type inference. It remains only to determine extents, where possible.

We approach shape analysis of user-provided code as an abstract interpretation problem. We define a shape language consisting of `Unit`, the shape constructor $\langle$`D, s`$\rangle$ described above, identifiers, and shape functions `extentof(s)`, `elementof(s)` that extract the two respective portions of a shape `s`. We implement a simple environment-based evaluator where every identifier is mapped to either (1) a shape term or (2) itself if its shape is unknown. Every primitive `f` is required to provide a function, that we denote `f.shape`, that returns the shape of its result given the shapes of its inputs. It may also return a set of static constraints on the shapes of its inputs to aid the compiler in its analysis. Examples of such shape-computing functions would be:

```
gather.shape(x, i) = ⟨extentof(i), elementof(x)⟩
zip.shape(x1, x2)  = ⟨extentof(x1),
                       (elementof(x1),elementof(x2))⟩
                         with extentof(x1)==extentof(x2)
```

The `gather` rule states that its result has the same size as the index array `i` while having elements whose shape is given by the elements of `x`. The `zip` augments the shape of its result with a constraint that the extents of its inputs are assumed to be the same. Terms such as `extentof(x)` are left unevaluated if the identifier `x` is not bound to any shape.

To give a sense of the shape analysis process, consider the `spvv` procedure shown in Figure 2. Shape analysis will annotate every binding with a shape like so:

```
def spvv(Ai, j, _k0):
    z0 :: ⟨extentof(j), elementof(_k0)⟩
    tmp0 :: ⟨extentof(Ai), elementof(Ai)⟩
    return sum(tmp0) :: Unit
```

In this case, the shapes for z_0, tmp_0, and the return value are derived directly from the shape rules for `gather`, `map`, and `sum`, respectively.

Shape analysis is not guaranteed to find all shapes in a program. Some identifiers may have data dependent shapes, making the analysis inconclusive. For these cases, the compiler must treat computations without defined shapes as barriers, across which no fusion can be performed, and then generate code which computes the actual data dependent shape before the remainder of the computation is allowed to proceed.

Future work involves allowing shape analysis to influence code scheduling: in an attempt to better match the extents in a particular nested data parallel problem to the dimensions of parallelism supported by the platform being targeted. For example, instead of implementing the outermost level of a Copperhead program in a parallel fashion, if the outer extent is small and the inner extent is large, the compiler may decide to create a code variant which sequentializes the outermost level, and parallelizes an inner level.

```
struct lambda0 {
    template<typename T_Aij, typename T_xj >
    __device__ T_xj operator()(T_Aij Aij, T_xj xj)
        { return Aij * xj; }
};

struct spvv {
    template<typename T_Ai,
             typename T_j,
             typename T_k0 >
    __device__ typename T_Ai::value_type
    operator()(T_Ai Ai,
               T_j j,
               T_k0 _k0) {
        typedef typename T_Ai::value_type _a;
        gathered<...> z0 = gather(_k0, j);
        transformed<...> tmp0 =
            transform<_a>(lambda0(), Ai, z0);
        return sequential::sum(tmp0);
    }
};

template<typename T2>
struct spvv_closure1 {
    T2 k0;
    __device__ spvv_closure1(T2 _k0) : k0(_k0) { }

    template<typename T0, typename T1>
    __device__ typename T0::value_type
    operator()(T0 arg0, T1 arg1) {
        return spvv()(arg0, arg1, k0);
    }
};

template<typename _a > __global__
void spmv_csr_phase0(nested_sequence<_a, 1> vals,
                     nested_sequence<int,1> cols,
                     stored_sequence<_a> x,
                     stored_sequence<_a> _return) {
    int i = threadIdx.x + blockIdx.x*blockDim.x;

    if( i < vals.size() )
        _return[i] = spvv_closure1<stored_sequence<_a> >
                         (x)(vals[i], cols[i]);
}
```

Figure 3: Sample CUDA C++ code generated for `spmv_csr`. Ellipses (...) indicate incidental type and argument information elided for brevity.

4.5 CUDA C++ Back End

The back end of the Copperhead compiler generates platform specific code. As mentioned, we currently have a single back end, which generates code for the CUDA platform. Figure 3 shows an example of such code for our example `spmv_csr` procedure. It consists of a sequence of function objects for the nested `lambdas`, closures, and procedures used within that procedure. The templated function `spmv_csr_phase0` corresponds to the Copperhead entry point.

Not shown here is the Copperhead generated C++ host code that invokes the parallel kernels. It is the responsibility of the host code to marshal data where necessary, allocate any required temporary storage on the GPU, and make the necessary CUDA calls to launch the kernel.

We generate templated CUDA C++ code that makes use of a set of sequence types, such as `stored_sequence` and `nested_sequence` types, which hold sequences in memory. Fusion of sequential loops and block-wise primitives is performed through the construction of compound types. For example, in the `spvv` functor shown above, the calls to `gather`, and `transform` perform no work, instead they construct `gathered_sequence` and `transformed_sequence` structures that lazily perform the appropriate computations upon dereferencing. Work is only performed by the last primitive in a set of fused sequential or block-wise primitives. In this example, the call to `seq::sum` introduces a sequential loop which then dereferences a compound sequence, at each element performing the appropriate computation. When compiling this code, the C++ compiler statically eliminates all the indirection present in this code, yielding machine code which is as efficient as if we had generated the fused loops directly.

We generate fairly high level C++ code, rather than assembly level code, for two reasons. Firstly, existing C++ compilers provide excellent support for translating well structured C++ to efficient machine code. Emitting C++ from our compiler enables our compiler to utilize the vast array of transformations which existing compilers already perform. But more importantly, it means that the code generated by our Copperhead compiler can be reused in external C++ programs. We believe that an important use case for systems like Copperhead is to prototype algorithms in a high-level language and then compile them into template libraries that can be used by a larger C++ application.

5 Runtime

Copperhead code is embedded in standard Python programs. Python function decorators indicate which procedures should be executed by the Copperhead runtime. When a Python program calls a Copperhead procedure, the Copperhead runtime intercepts the call, compiles the procedure, and then executes it on a specified execution place. The Copperhead runtime uses Python's introspection capabilities to gather all the source code pertaining to the procedure being compiled. This model is inspired by the ideas from Selective, Embedded, Just-In-Time Specialization [5].

The Copperhead compiler is fundamentally a static compiler that may be optionally invoked at runtime. Allowing the compiler to be invoked at runtime matches the no-compile mindset of typical Python development. However, the compiler does not perform dynamic compilation optimizations specific to the runtime instantiation of the program, such as treating inputs as constants, tracing execution through conditionals, etc. Forgoing these optimizations enables the results of the Copperhead compiler to be encapsulated as a standard, statically compiled binary, and cached for future reuse or incorporated as libraries into standalone programs which are not invoked through the Python interpreter.

Consequently, the runtime compilation overhead we incur is analogous to the build time of traditional static compilers, and does not present a performance limitation. The Copperhead compiler itself typically takes on the order of 300 milliseconds to compile our example programs, and the host C++ and CUDA compilers typically take on the order of 20 seconds to compile a program. Both of these overheads are not encountered in performance critical situations, since Copperhead's caches obviate recompilation. The actual runtime overhead, compared with calling an equivalent C++ function from within C++ code, is on the order of 100 microseconds per Copperhead procedure invocation.

For each Copperhead procedure, the CUDA runtime generates a host C++ function which coordinates the launch and synchronization between multiple parallel phases, as well as allocates data for temporary results. Copperhead uses PyCUDA [18] and CodePy [17] to provide mechanisms for compiling, persistent caching, linking and executing CUDA and C++ code. The Copperhead runtime uses Thrust [14] to implement fully parallel versions of certain data parallel primitives, such as `reduce` and variations of `scan`.

5.1 Places

In order to manage the location of data and kernel execution across multiple devices, the Copperhead runtime defines a set of *places* that represent these heterogeneous devices. Data objects are created at a specific place. Calling a

Copperhead procedure will execute a computation on the current target place, which is controlled utilizing the Python `with` statement.

Currently we support two kinds of places: CUDA capable GPUs and the native Python interpreter. Copperhead is designed to allow other types of places, with corresponding compiler back ends to be added. For instance, multi-core x86 back end would be associated with a new place type.

To faciliate interoperability between Python and Copperhead, all data is duplicated, with a *local* copy in the Python interpreter, and a *remote* copy which resides at the place of execution. Data is lazily transferred between the local and remote place as needed by the program.

5.2 Data Structures

Copperhead adopts a common technique for representing arbitrarily nested sequences as a flat sequence of data, along with a descriptor sequence for each level of nesting. The descriptor sequences provide the necessary information to build a view of each subsequence, including empty subsequences.

In addition to supporting arbitrarily nested sequences, Copperhead also allows programmers to construct uniformly nested sequences, which support the important special case where the shape is completely defined. For such sequences, a set of strides and lengths are sufficient to describe the nesting structure - descriptor sequences are not required. Uniformly nested sequences also allow the data to be arbitrarily ordered: when the programmer creates a uniformly nested sequence, they either specify the data ordering or provide a tuple of strides directly. This allows the programmer to express data layouts analogous to row- and column-major ordering for doubly nested sequences, as well as their analogues for more deeply nested sequences. The programmer may provide the strides directly, which allows the subsequences to be aligned arbitrarily. This can provide important performance benefits when data access patterns with standard nested sequences are not matched well to the processor's memory hierarchy.

We have seen have seen two to three fold performance improvements by using the correct nested data structure type. Future work may investigate autotuning over alignments and data layouts.

6 Examples

In this section, we investigate the performance of Copperhead code in several example programs. As we compare performance, we compare Copperhead compiled code to published, well-optimized, hand-crafted CUDA implementations of the same computation. Our compiler can't apply all the transformations which a human can, so we don't expect to achieve the same performance as well-optimized code. Still, we aim to show that high-level data parallel computations can perform within striking distance of human optimized efficiency code.

The Copperhead compiler and runtime is freely available at `http://code.google.com/p/copperhead`.

6.1 Sparse Matrix Vector Multiplication

Continuing with the Sparse Matrix Vector Multiplication example, we examine the performance of Copperhead generated code for three different SpMV kernels: compressed sparse row, vector compressed sparse row, and ELL. The CSR kernel is generated by compiling the Copperhead procedure for CSR SpMV onto the standard parallelism hierarchy, which distributes computations along the outermost parallelism dimension to independent threads. Data parallel operations are sequentialized into loops inside each thread. For the CSR kernel, each row of the matrix is then processed by a different thread, and consequently, adjacent threads are processing widely separated data. On the CUDA platform, this yields suboptimal performance for most datasets.

The vector CSR kernel improves on the performance of the CSR kernel by mapping to a different parallelism hierarchy: one where the outermost level distributes computations among independent thread blocks, and subsequent data parallel operations are then sequentialized to block-wise operations. As mentioned earlier, the Copperhead programmer can choose which hierarchy to map to, or can choose to autotune over hierarchies. In this case, mapping to the block-wise hierarchy improves memory performance on the CUDA platform, since adjacent threads inside the same thread block are processing adjacent data, allowing for vectorized memory accesses.

The ELL representation stores the nonzeros of the matrix in a dense M-by-K array, where K bounds the number of nonzeros per row, and rows with fewer than K nonzeros are padded. The array is stored in column-major order, so

that after the computation has been mapped to the platform, adjacent threads will be accessing adjacent elements of the array. For example:

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

```
vals = [[1,2,5,6],[7,8,3,4],[*,*,9,*]]
cols = [[0,1,0,1],[1,2,2,3],[*,*,3,*]]
```

ELL generally performs best on matrices where the variance of the number of non-zeros per row is small. If exceptional rows with large numbers of non-zeros exist, ELL will perform badly due to the overhead resulting from padding the smaller rows.

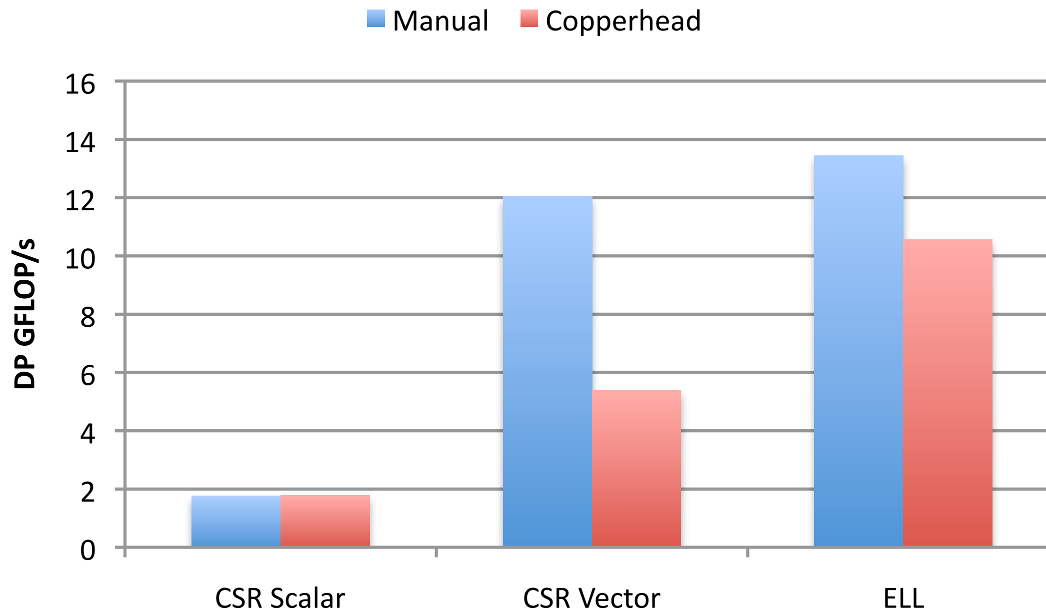

Figure 4: Average Double Precision Sparse Matrix Vector Multiply Performance.

We compare against Cusp [1], a C++ library for Sparse Matrix Vector multiplication, running on an NVIDIA GeForce GTX 480 GPU. We use a suite of 8 unstructured matrices which were used by Bell and Garland [1], and which are amenable to ELL storage. Copperhead generated code achieves identical performance for the scalar CSR kernel, and on average provides 45% and 79% of Cusp performance for the vector CSR and ELL kernels, respectively.

Our relatively low performance on the vector CSR kernel is due to a specialized optimization which the Cusp vector CSR implementation takes advantage of, but the Copperhead compiler does not: namely the ability to perform "warp-wise" reductions without synchronization, packing multiple rows of the SpMV computation into a single thread block. This optimization is an important workaround for the limitations of today's CUDA processors, but we consider it too special purpose to implement in the Copperhead compiler.

6.2 Preconditioned Conjugate Gradient Linear Solver

The Conjugate Gradient method is widely used to solve sparse systems of the form $Ax = b$. We examine performance on a preconditioned conjugate gradient solver written in Copperhead, which forms a part of an fixed-point non-linear solver used in Variational Optical Flow methods [27]. Figure 5 was generated using the `matplotlib` Python library, operating directly from the Copperhead solver, highlighting the ability of Copperhead programs to interoperate with other Python modules, such as those for plotting.

Conjugate gradient performance depends strongly on matrix-vector multiplication performance. Although we could have used a preexisting sparse-matrix library and representation, in this case we know some things about the structure of the matrix, which arises from a coupled 5-point stencil pattern on a vector field. Taking advantage of this structure, we can achieve significantly better performance than any library routine by creating a custom matrix format

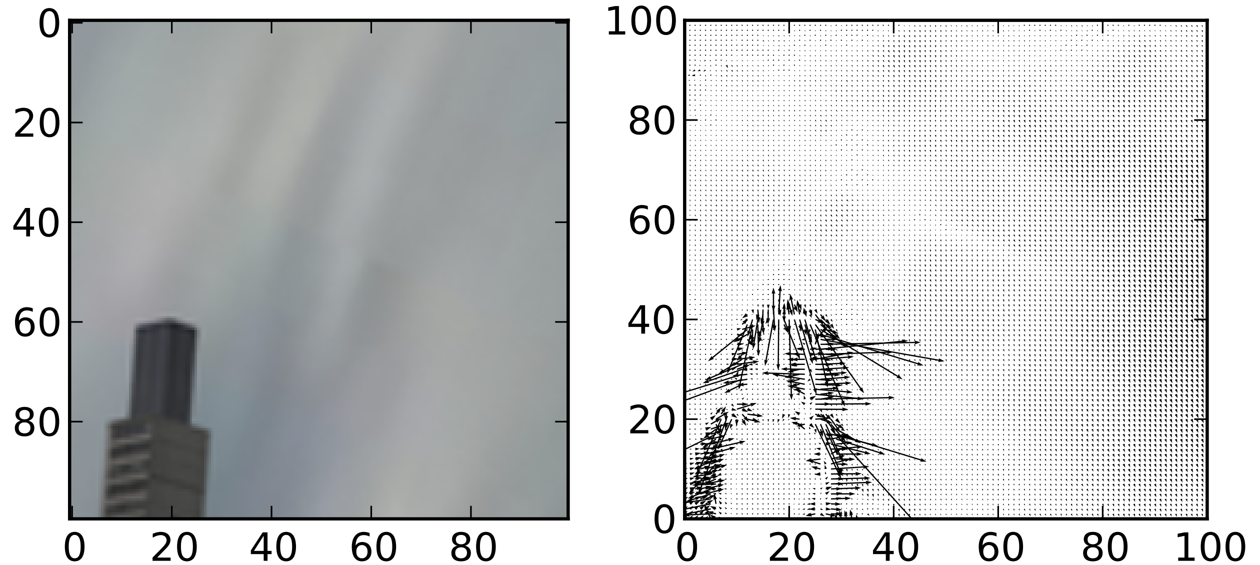

Figure 5: Left: closeup of video frame. Right: gradient vector field for optical flow.

and sparse matrix-vector multiplication routine. This is an ideal scenario for Copperhead, since the productivity gains enable programmers to write custom routines that take advantage of special knowledge about a problem, as opposed to using a prebuilt library.

In addition to writing a custom sparse-matrix vector multiplication routine, practically solving this problem requires the use of a preconditioner, since without preconditioning, convergence is orders of magnitude slower. We utilize a block Jacobi preconditioner. Figure 6 shows the Copperhead code for computing the preconditioner, which involves inverting a set of symmetric 2x2 matrices, with one matrix for each point in the vector field, as well as applying the preconditioner, which involes a large number of symmetric 2x2 matrix multiplicaitons.

We implemented the entire solver in Copperhead. The custom SpMV routine for this matrix runs within 10% of the hand-coded CUDA version, achieving 49 SP GFLOP/s on a GTX 480, whereas a hand-tuned CUDA version achieves 55 SP GFLOP/s on the same hardware. Overall, for the complete preconditioned conjugate gradient solver, the Copperhead generated code yields 71% of the custom CUDA implementation.

6.3 Quadratic Programming: Nonlinear Support Vector Machine Training

Support Vector Machines are a widely used classification technique from machine learning. Support Vector Machines classify multidimensional data by checking where the data lies with respect to a decision surface. Nonlinear decision surfaces are learned via a Quadratic Programming optimization problem, which we implement in Copperhead.

We make use of the Sequential Minimal Optimization algorithm, with first-order variable selection heuristic [16], coupled with the RBF kernel, which is commonly used for nonlinear SVMs. Computationally, this is an iterative algorithm, where at each step the majority of the work is to evaluate distance functions between all data points and two active data points, update an auxiliary vector and then to select the extrema from data-dependent subsets of this vector. More details can be found in the literature [6]. Space does not permit us to include the Copperhead code that implements this solver, but we do wish to point out the RBF kernel evaluation code, shown in Figure 9, is used both on its own, where the compiler creates a version that instantiates the computation across the entire machine, as well as nested as an inner computation within a larger parallel context, where the compiler fuses it into a sequential loop.

We compare performance against GPUSVM, a publically available CUDA library for SVM training. Figure 8 shows the throughput of the Copperhead implementation of SVM training over four datasets. On average, the Copperhead implementation attains 51% of hand-optimized, well tuned CUDA performance, which is quite good, given the Copperhead implementation's simplicity. The main advantage of the hand coded CUDA implementation on this example compared to the Copperhead compiled code is the use of on-chip memories to reduce memory traffic, and we expect that as the Copperhead compiler matures, we will be able to perform this optimization in Copperhead as well, thanks to our existing shape analysis, which can inform data placement.

```
@cu
def vadd(x, y):
    return map(lambda a, b: return a + b, x, y)
@cu
def vmul(x, y):
    return map(lambda a, b: return a * b, x, y)
@cu
def form_preconditioner(a, b, c):
    def det_inverse(ai, bi, ci):
        return 1.0/(ai * ci - bi * bi)
    indets = map(det_inverse, a, b, c)
    p_a = vmul(indets, c)
    p_b = map(lambda a, b: -a * b, indets, b)
    p_c = vmul(indets, a)
    return p_a, p_b, p_c
@cu
def precondition(u, v, p_a, p_b, p_c):
    e = vadd(vmul(p_a, u), vmul(p_b, v))
    f = vadd(vmul(p_b, u), vmul(p_c, v))
    return e, f
```

Figure 6: Forming and applying the block Jacobi preconditioner.

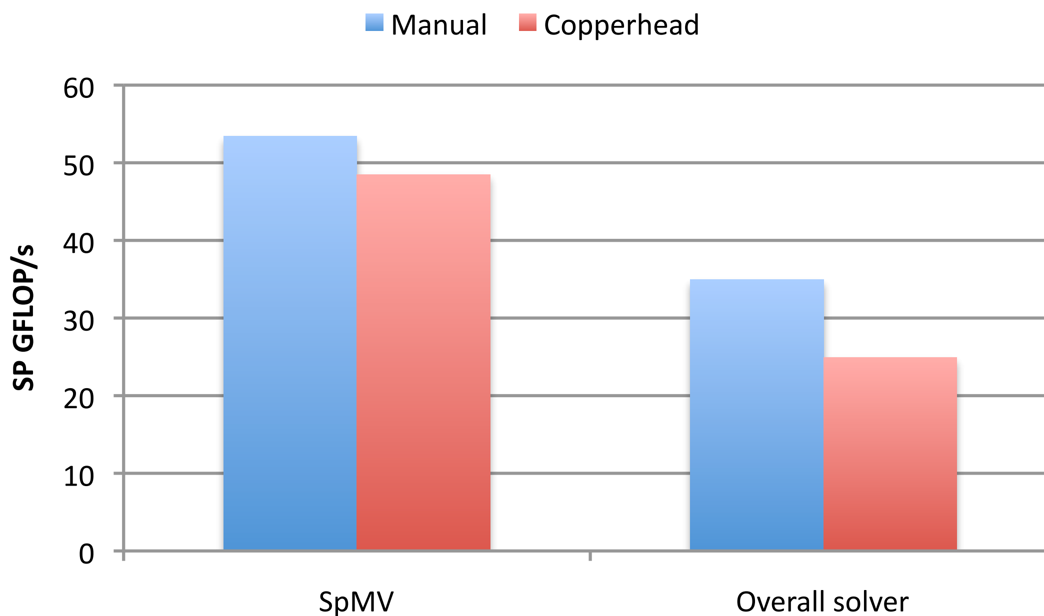

Figure 7: Performance on Preconditioned Conjugate Gradient Solver.

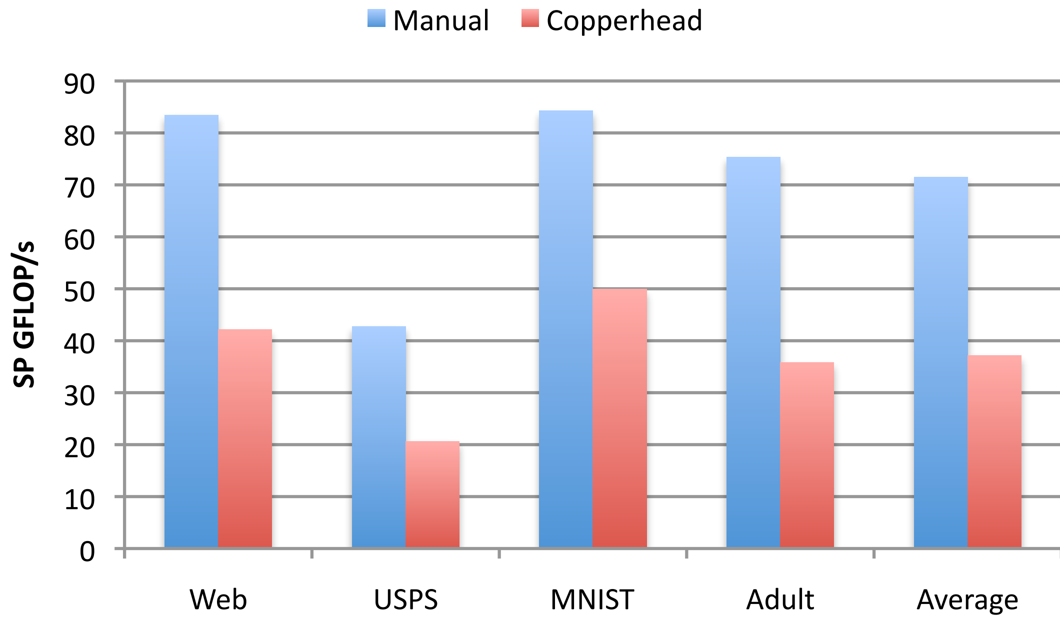

Figure 8: Support Vector Machine Training Performance.

```
@cu
def norm2_diff(x, y):
    def el(xi, yi):
        diff = xi - yi
        return diff * diff
    return sum(map(el, x, y))
@cu
def rbf(gamma, x, y):
    return exp(-gamma * norm2_diff(x,y))
```

Figure 9: RBF Kernel evaluation.

6.4 Productivity

Productivity is difficult to measure, but as a rough approximation, Table 2 provides the number of lines of code needed to implement the core functionality of the examples we have put forth, in C++/CUDA as well as in Python/Copperhead. On average, the Copperhead programs take about 4 times fewer lines of code than their C++ equivalents, which suggests that Copperhead programming is indeed more productive than the manual alternatives.

Table 2: Number of Lines of Code for Example Programs.

Example	CUDA & C++	Copperhead & Python
Scalar CSR	16	6
Vector CSR	39	6
ELL	22	4
PCG Solver	172	79
SVM Training	429	111

7 Future Work

There are many avenues for improving the Copperhead runtime and compiler. The first set involves improving the quality of code which the compiler emits. For example, the compiler currently does not reorder parallel primitives to increase fusion and reduce synchronization. The compiler also does not attempt to analyze data reuse and place small, yet heavily reused sequences in on-chip memories to reduce memory bandwidth, nor does it attempt to take advantage of parallelism in the dataflow graph or parallel primitives. Another avenue for future work involves broadening the scope of programs which can be compiled by the Copperhead compiler, such as supporting multi-dimensional arrays, adding new data parallel primitives, and supporting forms of recursion other than tail recursion.

We also intend to create other back-ends for the Copperhead compiler, in order to support other platforms besides CUDA. Some potential choices include support for multicore x86 processors and OpenCL platforms.

8 Conclusion

This paper has shown how to efficiently compile a nested data parallel language to modern parallel microprocessors. Instead of using flattening transforms by default, we take advantage of nested parallel hardware by mapping data parallel computations directly onto the hardware parallelism hierarchy. This mapping is enabled by synchronization and shape analyses, which we introduce. Instead of creating a new data parallel language, we have embedded Copperhead into a widely used productivity language, which makes Copperhead programs look, feel and operate like Python programs, interoperating with the vast array of Python libraries, and lowering the amount of investment it takes for a programmer to learn data parallel programming.

We then demonstrated that our compiler generates efficient code, yielding from 45 to 100% of the performance of hand crafted, well-optimized CUDA code, but with much higher programmer productivity. On average, Copperhead programs require 3.6 times fewer lines of code than CUDA programs, comparing only the core computational portions.

Large communities of programmers choose to use productivity languages such as Python, because programmer productivity is often more important than attaining absolute maximum performance. We believe Copperhead occupies a worthwhile position in the tradeoff between productivity and performance, providing performance comparable to that of hand-optimized code, but with a fraction of the programmer effort.

9 Acknowledgments

Research partially supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding from U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National

Instruments, NEC, Nokia, NVIDIA, Samsung, and Sun Microsystems. Bryan Catanzaro was supported by an NVIDIA Graduate Fellowship.

Bibliography

[1] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proc. Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11. ACM, 2009.

[2] G. E. Blelloch. *Vector models for data-parallel computing*. MIT Press, Cambridge, MA, USA, 1990.

[3] G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, 1996.

[4] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, Apr. 1994.

[5] B. Catanzaro, S. Kamil, Y. Lee, K. Asanović, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox. SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization. Technical Report UCB/EECS-2010-23, EECS Department, University of California, Berkeley, 2010.

[6] B. Catanzaro, N. Sundaram, and K. Keutzer. Fast Support Vector Machine Training and Classification on Graphics Processors. In *Proceedings of the 25th International Conference on Machine Learning*, pages 104–111, 2008.

[7] M. M. T. Chakravarty, R. Leshchinskiy, S. P. Jones, G. Keller, and S. Marlow. Data parallel Haskell: a status report. In *Proc. 2007 Workshop on Declarative Aspects of Multicore Programming*, pages 10–18. ACM, 2007.

[8] Clyther: Python language extension for opencl. http://clyther.sourceforge.net/, 2010.

[9] Cython: C-extensions for python. http://cython.org/, 2010.

[10] R. Garg and J. N. Amaral. Compiling python to a hybrid execution environment. In *GPGPU '10: Proc. 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 19–30. ACM, 2010.

[11] A. Ghuloum, E. Sprangle, J. Fang, G. Wu, and X. Zhou. Ct: A Flexible Parallel Programming Model for Tera-scale Architectures. Technical Report White Paper, Intel Corporation, 2007.

[12] T. D. Han and T. S. Abdelrahman. hiCUDA: a high-level directive-based language for GPU programming. In *GPGPU-2: Proc. 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 52–61. ACM, 2009.

[13] W. D. Hillis and J. Guy L. Steele. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

[14] J. Hoberock and N. Bell. Thrust: A parallel template library. `http://www.meganewtons.com/`, 2009. Version 1.2.

[15] 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.

[16] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy. Improvements to Platt's SMO Algorithm for SVM Classifier Design. *Neural Comput.*, 13(3):637–649, 2001.

[17] A. Klöckner. Codepy. `http://mathema.tician.de/software/codepy`, 2010.

[18] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. Pycuda: Gpu run-time code generation for high-performance computing. *CoRR*, abs/0911.3456, 2009.

[19] S. Lee, M. M. T. Chakravarty, V. Grover, and G. Keller. GPU kernels as data-parallel array computations in Haskell. In *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods (EPAHM 2009)*, 2009.

[20] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 101–110. ACM, 2009.

[21] M. McCool. Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform. In *Proc. GSPx Multicore Applications Conference*, Nov. 2006.

[22] M. D. McCool, Z. Qin, and T. S. Popa. Shader metaprogramming. In *Proc. Graphics Hardware 2002*, pages 57–68. Eurographics Association, 2002.

[23] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, Mar/Apr 2008.

[24] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, May 2010. Version 3.1.

[25] L. Prechelt. Are Scripting Languages Any Good? A Validation of Perl, Python, Rexx, and Tcl against C, C++, and Java. In *Advances in Computers*, volume 57, pages 205 – 270. Elsevier, 2003.

[26] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, July 2010.

[27] N. Sundaram, T. Brox, and K. Keutzer. Dense Point Trajectories by GPU-accelerated Large Displacement Optical Flow. In *European Conference on Computer Vision*, pages 438–445, September 2010.

[28] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. *SIGARCH Comput. Archit. News*, 34(5):325–335, 2006.

[29] Theano: A python array expression library. http://deeplearning.net/software/theano/, 2010.

[30] M. Wolfe. Implementing the PGI accelerator model. In *Proc. 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 43–50. ACM, 2010.