Portable Parallel Performance from Sequential, Productive, Embedded Domain-Specific Languages

Shoaib Kamil Derrick Coetzee Scott Beamer Henry Cook Ekaterina Gonina
Jonathan Harper† Jeffrey Morlan Armando Fox
UC Berkeley and †Mississippi State University
{skamil,dcoetzee,sbeamer,hcook,egonina,jmorlan,fox}@cs.berkeley.edu, jwh376@msstate.edu

Abstract

Domain-expert *productivity programmers* desire scalable application performance, but usually must rely on *efficiency programmers* who are experts in explicit parallel programming to achieve it. Since such programmers are rare, to maximize reuse of their work we propose encapsulating their strategies in mini-compilers for domain-specific embedded languages (DSELs) glued together by a common high-level host language familiar to productivity programmers. The nontrivial applications that use these DSELs perform up to 98% of peak attainable performance, and comparable to or better than existing hand-coded implementations. Our approach is unique in that each mini-compiler not only performs conventional compiler transformations and optimizations, but includes imperative procedural code that captures an efficiency expert's strategy for mapping a narrow domain onto a specific type of hardware. The result is source- and performance-portability for productivity programmers and parallel performance that rivals that of hand-coded efficiency-language implementations of the same applications. We describe a framework that supports our methodology and five implemented DSELs supporting common computation kernels.

Our results demonstrate that for several interesting classes of problems, efficiency-level parallel performance can be achieved by packaging efficiency programmers' expertise in a reusable framework that is easy to use for both productivity programmers and efficiency programmers.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Frameworks; D.1.3 [*Programming Techniques*]: Parallel programming

General Terms Design, Languages, Performance

Keywords Asp, SEJITS, Python, Domain-Specific Languages

1. Introduction

Domain-expert *productivity programmers* must choose between writing high-level code or working with low-level *efficiency programmers* who understand details of hardware in order to obtain good parallel performance. Instead, we propose that these expert efficiency programmers encapsulate their knowledge of how to make computations in a particular domain fast and parallel into compilers for domain-specific embedded languages (DSELs[1]). These DSELs are coordinated and embedded into a high-level programming language such as Python that can be used by productivity programmers to write their own programmers, but obtain the performance benefits of low-level machine-aware code.

```
class Laplacian3D(StencilKernel):
  def kernel(self, in_grid, out_grid):
    # the following lines are translated into
    # parallel C++ loops by the compiler & run
    for x in out_grid.interior_points():
      for y in in_grid.neighbors(x, 1):
        out_grid[x] = out_grid[x] + (1/6)*in_grid[y]
```

Figure 1. Python source code for 3D divergence kernel using the stencil DSEL. The user may specify grid connectivity or use defaults provided by the specializer (not shown).

We have created a framework called Asp that helps efficiency programmers write DSEL compilers (which we call *specializers*) by abstracting away many common tasks, including code generation, code caching, and just-in-time compilation. Using this framework, we have built five DSELs in disparate areas such as stencil computations, statistical machine learning, and linear algebra; each of these DSEL compilers are being used in nontrivial applications that achieve performance portability across platforms and obtain peak performance that rivals low-level hand-coded performance for the domain.

2. Asp Infrastructure & Walkthrough

The Asp infrastructure provides a number of capabilities for building a DSEL compiler, which can be used as building blocks by the DSEL developer. Specializers are typically used by subclassing a particular class and providing a few functions, which follow documented restrictions based on the specializer's capabilities. On instantiation, the function definitions are introspected and a Python parse tree is generated from them by the Asp framework. At execution time, this tree is then transformed into a DSEL-specific intermediate form, which encapsulates the semantics of the expressed computation— we call this the semantic model (SM). Further tree transformation phases occur, optionally depending on aspects of the input to the specialized function. At the end of these phases, the domain-specific code has been turned into low-level optimized code that is automatically compiled, linked, and run, using

[1] Following Hudak's [3] terminology, we use the acronym DSEL for Domain-Specific Embedded Languages to distinguish them from standalone or "external" DSLs.

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.
ACM 978-1-4503-1160-1/12/02.

Framework Feature	Used by Specializers	Provided by
Parse Python source to AST	Stencil, BSP, KDT	Asp/Python
Generic lowering translations (e.g. arithmetic expressions)	Stencil, BSP, KDT	Asp
Interrogate available hardware/software	Stencil, GMM	Asp
Generic optimizations	Stencil, BSP	Asp
C++ AST	Stencil, BSP, KDT	Asp
Instantiate templates	GMM, KDT, Akx	Asp
Compile/Invoke C++ with Caching	All except KDT	Asp/CodePy
Tree visitor and translation	Stencil, BSP, KDT	Asp
Tree grammar definition and checking	Stencil	Asp
Fallback to Python version	All	Asp
Auto-tuning & Timing Support	Stencil, GMM, Akx	Asp

Specializer	Application	Logic	Tmpl.	Targets	Performance Remarks
Stencil (structured grid)	Bilateral image filtering	656	0	C++/OpenMP, Cilk+	91% of achievable peak based on roofline model [6]
Gaussian mixture model (GMM) training	Speech diarization	800	3600	CUDA, Cilk+	CPU & GPU versions fast enough to replace original C++/pthreads code
Graph algorithms with KDT/CombBLAS	Graph500 benchmark	325	0	C++/MPI	99% of performance of handcoding in C++
Graph algorithms in BSP style	Social Network Analytics	250	280	C++	56–120% of performance of native C++ Boost version
Matrix powers ($\mathbf{A}^k\mathbf{x}$)	Conjugate gradient solver	200	2000	C/pthreads	2-4 times faster than SciPy

Figure 2. *Top:* Features of the Asp framework and which specializers use them. *Bottom:* For each specializer we report the LOC of logic, LOC of templates, target languages, and a summary of the performance of the Python+SEJITS application compared to the original efficiency-language implementations. Specializer logic is Python code that manipulates intermediate representations in preparation for code generation and templates are static efficiency-language "boilerplate" files into which generated code is interpolated. Our framework itself comprises 2094 LOC.

the CodePy (`http://mathema.tician.de/software/codepy`) library. Optionally, many versions can be generated in order to enable auto-tuning.

An overview of the different features of the Asp framework is shown in Figure 2 (top), as well as which specializers use which capabilities. The next section gives more details about the different DSELs we have implemented.

3. Implemented DSELs & Performance Results

We have implemented five specializers using the Asp framework. The first is a DSEL for stencil computations, which operate on a multidimensional grid and update each point with a function of a subset of its neighbors. We have also implemented two DSELs for graph computations: one using the Knowledge Discovery Toolbox framework (KDT, `kdt.sourceforge.net`) which casts graph algorithms as linear algebra[1]; and one for bulk-synchronous-style graph algorithms similar to Pregel [4]. We have also built auto-tuned libraries as specializers for training Gaussian Mixture Models [2] and for communication-avoiding $A^k x$, a building block in communication-avoiding Krylov subspace methods for solvers [5].

The five specializers, applications, and performance results are summarized in Figure 2 (bottom). Overall, the combination of auto-tuning and just-in-time compilation allows the creation of DSELs that enable non-expert programmers to write Python code that runs as fast or faster than existing low-level libraries or hand-tuned code in the domain in question.

4. Conclusion

With our Asp infrastructure for building DSEL compilers, DSEL developers can leverage the library to perform many common tasks. Our infrastructure is publicly available (`http://github.com/shoaibkamil/asp`), and a number of DSELs are under development. Ultimately, as the number of DSELs increases, parallelism and high performance will be even more accessible to domain scientists for use in their computations while still programming in high-level languages.

Acknowledgments

This work was performed at the UC Berkeley Parallel Computing Laboratory (Par Lab), supported by DARPA (contract #FA8750-10-1-0191) and by the Universal Parallel Computing Research Centers (UPCRC) awards from Microsoft Corp. (Award #024263) and Intel Corp. (Award #024894), with matching funds from the UC Discovery Grant (#DIG07-10227) and additional support from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Oracle, and Samsung.

References

[1] A. Buluç and J. R. Gilbert. The combinatorial BLAS: Design, implementation, and applications. Technical Report UCSB-CS-2010-18, University of California, Santa Barbara, 2010.

[2] H. Cook, E. Gonina, S. Kamil, G. Friedland, and D. P. A. Fox. Cuda-level performance with python-level productivity for gaussian mixture model applications. In *3rd USENIX conference on Hot topics in parallelism (HotPar'11)*, Berkeley, CA, USA, 2011.

[3] P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28:196, December 1996. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/242224.242477.

[4] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. *SIGMOD*, Jun 2010.

[5] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick. Minimizing communication in sparse matrix solvers. In *Supercomputing 2009*, Portland, OR, Nov 2009.

[6] S. Williams, A. Waterman, and D. A. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, pages 65–76, 2009.