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 efficiency 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. Our approach is unique in two ways. First, 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. Second, the mini-compilers emit source code in an efficiency language such as C++/OpenMP or CUDA that allows targeting low-level hardware optimizations using downstream compilers combined with auto-tuning, where many candidate implementations are generated and run to discover the best. The result is source- and performance-portability for productivity programmers, and performance that rivals that of hand-coded efficiency-language implementations.We describe a framework that supports our methodology and five implemented DSELs supporting common computation kernels. The non-trivial applications that use these kernels achieve performance portability across platforms and measured performance comparable to hand-coded efficiency implementations, despite being written entirely in the productivity language Python. 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.

1. Introduction

Our goal is best illustrated by a short scenario that is common among our scientific colleagues. Paul, a productivity programmer whose main research area is biology, has prototyped a new graph algorithm in Python, a language that supports his domain well because of its library support for reading molecule files, graphing results, and so on. Since Python is too slow to run his algorithm on non-toy problems, Paul retains Elena, an efficiency programmer, to create a high-performance version of his algorithm that can exploit the parallelism of Paul's multicore servers with GPUs.

Since programmers like Elena are rare, we would like to reuse their work as widely as possible. Rather than rewriting Paul's application entirely in C++ or CUDA, Elena might "package" the kernel of his algorithm as a library that can be called from Python, though the library would have to include enough generality to allow other scientists to tweak the algorithm without being efficiency programmers themselves. Furthermore, the library would have to adapt to hardware differences such as different numbers of cores or different models of GPU. Even on instruction-set-compatible hardware such as different Intel processors, differences in cache geometry or memory architecture might require quite different decisions to achieve the highest performance.

In our approach, Elena instead designs an embedded domain-specific language (DSEL[1]) for Paul's problem domain. DSELs provide a concise, semantically well-founded way for a programmer to express a computation in a natural notation. The abstractions provided by a DSEL can be chosen to make compilation effective, e.g. through domain-specific parallelism constructs or by guiding the programmer to express intent rather than process. Elena uses our framework to embed the DSEL in Python and create a lightweight compiler that converts her DSEL into source code in an efficiency

[1] Following Hudak's [18] terminology, we use the acronym DSEL for Domain-Specific Embedded Languages to distinguish them from standalone or "external" DSLs.

language such as C++ or CUDA, which exposes enough hardware properties to effect "last-mile" optimizations such as cache blocking and which is well supported by a rich ecosystem of optimizing compilers. Because our framework makes the DSEL compiler much more compact and easier to write than a full compiler (indeed, the DSEL compiler is written in Python and is typically hundreds of lines rather than thousands), most of Elena's effort is in expressing her strategy for mapping the computation to a specific hardware platform (or family of platforms) in the most efficient way. Her strategy may include producing a number of possible variants, each potentially optimal. The combination of procedural code and efficiency-language source code snippets that implement a DSEL compiler is called a *specializer* in our approach. The result is that although Paul's application is source- and performance-portable and expressed in a productivity language (Python in our case, though our approach generalizes to other languages), its performance is comparable to that of an equivalent application coded entirely in an efficiency-level language.

This paper makes the following contributions:

1. The SEJITS methodology (Selective Embedded Just-in-Time Specialization) for embedding DSELs into a common productivity language and creating specializers that generate efficient code from them.
2. A framework called Asp that supports the embedding of DSELs in Python and implementation of lightweight DSEL compilers (specializers) in Python. Python is popular among productivity programmers and simplifies specializer creation for efficiency programmers: typical specializers are hundreds of lines long rather than thousands. Asp is implemented as a Python package that does not alter the interpreter in any way, ensuring existing Python code still runs.
3. A demonstration of the approach via five implemented specializers (stencils[2], Gaussian mixture model training, two styles of graph algorithms, and communication-avoiding sparse linear algebra [22]) that use our framework. Both the specializers and the nontrivial Python applications that use them perform comparably to manually-tuned efficiency-language code. One of the specializers (Gaussian mixture model) can target either a multicore CPU or different models of GPU transparently to the application programmer, and its performance is close enough to hand-tuned code that the research group whose speech-processing application relies on it has replaced their C++ implementation with a Python/specializer implementation to accelerate research.

[2] A stencil is a data-parallel structured grid computation in which each point in an N-dimensional grid is updated according to a function of its neighbor points. The function, boundary calculations, and the number of neighbors considered are application-specific.

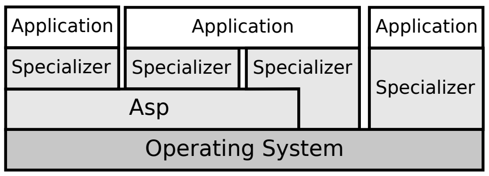


Figure 1. Applications interact with one or more specializers, which may use all, some, or none of the features provided by the Asp framework.

We note that we do *not* claim that our approach improves on the performance of highly-tuned libraries (although in some cases it does), but rather that it allows the reuse of *strategies* for computing different kernels by enabling a degree of runtime tuning and auto-tuning that is often difficult to achieve gracefully with libraries, even while allowing application writers to express key parts of their algorithms in a high-level language while enjoying the performance of an efficiency language.

Section 2 describes the SEJITS methodology and what makes it different from other DSEL-based approaches. Section 3 describes the organization of typical specializers and the support provided by the Asp framework to simplify their creation. Section 4 describes five implemented specializers that use our framework and the performance of nontrivial applications that use them; all are either real customer applications maintained by researchers outside our group or standard domain benchmarks. Section 5 reflects on the results and discusses pros and cons of our approach. Section 6 compares our work to relevant prior work. Section 7 describes our ongoing efforts. Finally, Section 8 concludes with our thoughts on the role of the SEJITS approach for both productivity and efficiency programmers.

2. The SEJITS Methodology

Selective Embedded Just-in-Time Specialization (SEJITS) describes a methodology for building DSEL compilers in high-level languages that bridge the gap between productive programming and high performance. In our approach the DSEL compiler includes domain-specific, imperative procedural code that captures domain-specific and hardware-specific knowledge, such as what code variants make sense to generate, how to tile or block memory, which partitioning strategy to use in subdividing the parallelism in the problem, and so on.

DSELs in other approaches usually transform constructs into lower-level constructs within the same language. This approach is common in languages that support metaprogramming, such as Scheme, Gambit, and Haskell. In contrast, our approach allows programmers to write specializers for languages without first-class metaprogramming, increasing the range of languages in which DSELs can be embedded. Furthermore, SEJITS concentrates on utilizing the language's FFI (foreign function interface) for these specializers, allowing them to take advantage of downstream optimizing compilers by linking object code resulting from DSEL compilation with non-specialized productivity code in Python. This support for external compilers, combined with our framework's ability to easily "templatize" existing source code in efficiency languages such as C or CUDA, means that a specializer can start out as a simple "wrapping" of existing efficiency-level code (e.g. an efficiency programmer's specific solution to one problem instance) and gradually generalize the strategy to handle a wider range of problems, further simplifying the learning curve required to build a specializer.

A key advantage of our approach for specializer writers is that this procedural code can be written in a high-level language (in our case Python) and can leverage the extensive support we provide for "common" tasks such as abstract syntax tree manipulation, low-level code generation and caching, and so on. As a result, the typical specializer comprises a modest number of lines of Python code and does not require a deep understanding of compilers to create, because the source language is highly constrained and we provide a variety of building blocks for implementing specializers in a very high-level language.

In fact, the SEJITS approach can be thought of as providing two major capabilities: embedded DSLs that use the foreign function interface and run-time code generation with auto-tuning. These two pieces work together to enable all the benefits outlined above, but can in fact be used without each other. For example, even a library such as for sparse matrix multiplication, which can be thought of as "trivial" DSEL that expresses a single construct, can benefit from run-time code generation since the code can be tailored to the particular matrix and particular machine the computation is running on, yielding high performance. Such trivial DSELs are an important aspect of obtaining fast parallel performance from a productivity language. Thus, although the combination of FFI-enabled embedded DSELs and run-time code generation enables some of the most interesting uses of the approach, even high performance libraries can use the SEJITS approach.

3. Specializers in the Asp Framework

Figure 1 summarizes how Asp[3], specializers, and applications interact in a system. Because specializers can be built even without the facilities provided by the Asp framework, only some of the specializers in the figure utilize it. Below we discuss features provided by the Asp framework and a prototypical architecture for specializer implementations.

3.1 Framework features

The framework provides two main mechanisms for specializers to generate code: *templates* and *tree transformations*. Templates are efficiency-level code interspersed with productivity-code fragments and control code to fill in substrings based on properties determined at specialization time. These can embed values ranging in complexity from a single scalar value to complex efficiency-language statements, such as loop nests dependent on input properties for depth, and are rich enough that some specializers rely solely on templates.

For domains in which the efficiency-level code depends in a complex way on the DSEL source, we also provide support for tree transformations. It is common for specializers to use templates and tree transformations together to handle scenarios in which part of the output source code depends in a complex way on the input, while other parts such as helper functions are relatively static.

Since compilers are generally written as a multi-phase pipeline processing an intermediate tree data structure, Asp is built to facilitate rapid construction of specializers of this architecture in few lines of code.

The front-end parser is largely eliminated by the use of an embedded DSL, and back-end runtime code generation and caching from a target-language abstract syntax tree is supplied by Asp. Backends can target a variety of high-performance languages and compiler tools, leveraging the existing investment in these tools for target-specific optimization. We leverage CodePy (`mathema.tician.de/software/codepy`) for C/C++-based backends, and a Scala backend is in development.

For intermediate phases, our framework provides a visitor-pattern abstraction for tree traversal and a concise language for specifying strongly-typed intermediate representations. This intermediate representation reflects the semantics of the computation at hand, using the (imperative) Python source code as a declarative description of the computation. In other words, though computations in the DSELs are expressed as imperative code, the translation to the intermediate representation only translates *what* to compute, not how. The DSEL compiler is free to compute the declarative specification in whatever manner suits the domain and underlying hardware.

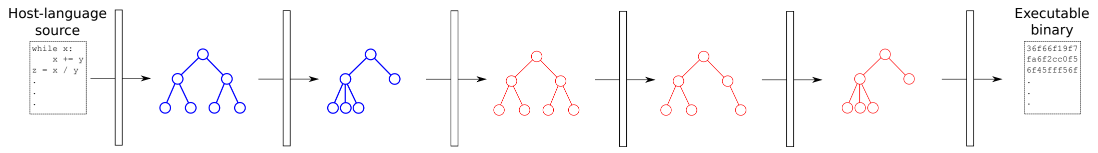
Other Asp features are targeted at specific phases. Each target language backend provides *default translations* for common DSEL constructs such as arithmetic and conditional expressions—in most cases, DSEL implementers only need to specify transformations for custom, domain-specific nodes.

During target-specific optimization, specializers rely on Asp to interrogate available hardware, to perform common generic optimizations such as loop unrolling and cache blocking, and to select the best among several generated code variants.

The combined effect of these features on code size reduction in specializers is dramatic, with complete production specializers such as the stencil and KDT specializers requiring only a few hundred lines of Python code, as shown in Figure 2.

Because specializers rely on generating source code and compiling it into dynamic shared libraries, caching is an important part of our approach as it amortizes the compilation

[3] Asp is SEJITS for Python, `http://github.com/shoaibkamil/asp`



	Parser	Target-independent optimizations		Target-specific optimizations	Target code generation
Stencil specializer lines of code (§4.1)	101	67	159	← 84 →	0
KDT specializer lines of code (§4.3)	36	0	70	← 0 →	61

Figure 2. Lines of code in each phase of two specializers described in Section 4; each has some additional lines of utility code.

time over many runs if subsequent calls can use the cached version. We leverage CodePy's integrated caching (which caches compiled code by comparing MD5 hashes of the source) but also include a higher-level caching method that allows specializers to dictate whether, based on runtime parameters, an existing version is runnable.

3.2 Flexibility

Because they are invoked at runtime, specializers can generate different output code depending on the problem inputs (i.e. arguments to DSEL methods). A simple example is fixing the upper bound of a loop over an input matrix based on its runtime dimensions. A more sophisticated example occurs in the matrix powers specializer (Section 4.4), which samples statistics over matrix elements to help select the best blocking format for a particular problem.

Specialization may fail for a number of reasons: The code written by the application programmer may be valid in the host language, but not represent a valid DSEL program; the specializer may only be able to emit code for a compiler or hardware target that is unavailable at runtime; or downstream errors may be encountered in the code generation phase (e.g. invoking the compiler), due to errors in the specializer or misconfiguration. To this end, we recommend that each specializer include an implementation of its DSEL written purely in the embedding language, in our case Python. Asp automatically falls back to running this alternate version (albeit with poor performance) if specialization fails, improving source portability. For most specializers, a warning is issued in this case, to encourage programmers to express their code in the supported subset and explain the poor performance.

4. Applications and Results

We next describe five implemented specializers and some applications that use them: stencils, Gaussian mixture model training, two approaches to graph algorithms, and matrix powers. In our framework, each specializer is used by subclassing from a specific Python class and implementing specializer-specific virtual methods. Runtime translation is performed when these methods are subsequently called.

Python code outside of any DSEL is executed normally by the Python interpreter; our approach does not modify the standard Python distribution in any way.

Figure 3 lists the specializers, emphasizing the benefit of SEJITS to efficiency programmers by reporting on the approximate size of each specializer, a proxy for the efficiency programmers' effort to create it. The subsequent sections describe specific aspects of interest in each specializer's construction and show performance results of the specializer in a nontrivial application, thereby demonstrating the benefit to productivity programmers. Note that for each specializer, multiple applications have been developed using it; we outline only a single application due to space constraints.

For many of these domains, there is no universally acknowledged "gold standard" for comparing performance. We therefore compare against publicly-available, widely-used libraries and compilers for each domain that are generally accepted as providing good efficiency-level performance. Where possible, we also characterize performance as a portion of achievable peak based on hardware characteristics. For reported performance numbers, we elide JIT compilation time (which is on the order of seconds, mostly due to running an external compiler), since caching means that code generation and compilation occur only the first time a particular problem is run— subsequent executions occur at full speed.

4.1 Stencils

Structured grid computations, also called stencil computations, consist of updating each grid point in an n-dimensional grid with some function of a subset of its neighbors. Stencils occur in image processing, solving linear equations, simulations of physical phenomena, and many other domains.

Why a Specializer: Depending on the specific application, the update function and the definition of a neighbor are highly problem-dependent. A general library would need to perform at least one function call per point. With more advanced techniques such as expression templates in C++, the operator can be inlined but optimizations cannot take advantage of many properties of the stencil. Although a stencil computation is conceptually straightforward, achieving good performance requires many optimizations, including blocking the computation in space and/or time explicitly [12, 20]

§	Specializer	Application	Logic	Tmpl.	Targets	Performance Remarks
4.1	Stencil (structured grid)	Bilateral image filtering	656	0	C++/OpenMP, Cilk+	91% of achievable peak based on roofline model [32]
4.2	Gaussian mixture model (GMM) training	Speech diarization	800	3600	CUDA, Cilk+	CPU & GPU versions fast enough to replace original C++/pthreads code
4.3	Graph algorithms with KDT	Graph500 benchmark	325	0	C++/MPI	99% of performance of handcoding in C++
4.4	Matrix powers ($\mathbf{A}^k\mathbf{x}$)	Conjugate gradient solver	200	2000	C/pthreads	2-4 times faster than SciPy

Figure 3. 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. Recall that 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 providing common functionality such as tree manipulation, code generation, compiler toolchain control, code caching, runtime hardware detection, and transforming common Python constructs such as simple arithmetic expressions into SM nodes.

```
class Div3D(StencilKernel):
  def kernel(self, in_grid1, in_grid2, in_grid3, out_grid
      ):
    for x in out_grid.interior_points():
      for y in in_grid1.neighbors(x, 1):
        out_grid[x] = out_grid[x] + C1*in_grid1[y]
      for y in in_grid2.neighbors(x, 1):
        out_grid[x] = out_grid[x] + C2*in_grid2[y]
      for y in in_grid3.neighbors(x, 1):
        out_grid[x] = out_grid[x] + C3*in_grid3[y]
```

Figure 4. Python source code for 3D divergence kernel using the stencil DSEL. The user may specify grid connectivity or use defaults provided by the specializer. Note the inclusion of DSEL abstractions for `interior_points` and `neighbors`, which avoids the analysis that optimizing compilers must perform when analyzing the ordered loops typical of an efficiency-language stencil implementation.

```
for (int x1x1=1; (x1x1<=256); x1x1=(x1x1+(1*256))) {
  for (int x2x2=1; (x2x2<=256); x2x2=(x2x2+(1*32))) {
#pragma omp parallel for
    for (int x1=x1x1; (x1<=min((x1x1+255),256)); x1=(x1
        +1)) {
      for (int x2=x2x2; (x2<=min((x2x2+31),256)); x2=(x2
          +1)) {
#pragma ivdep
        for (int x3=1; (x3<=(256-0)); x3=(x3+(4*1))) {
          //fully-unrolled neighbor loop,
          //unrolled further by 4
} } } } }
```

Figure 5. The presence of optimizations makes the optimized C++ source code the simple 3D divergence kernel harder to read and maintain. In Python, this is essentially two nested loops, while the optimized C++ is a 5-deep nest due to cache blocking, with loop bounds that are closely tied to the memory architecture of the target machine.

or using cache-oblivious algorithms [16], vectorizing [23], or other techniques combining these optimizations such as polyhedral analysis [33].

The result is that the straightforward computation shown in Figure 4 becomes the complex and difficult-to-read code shown in Figure 5.

Details of Specializer: Our stencil specializer uses a cache-aware approach plus auto-tuning to generate fast, parallel efficiency code in either Intel Cilk+ or C++/OpenMP from the stencil DSEL shown in Figure 4. We have not yet implemented a GPU code generator, but nothing in the specializer structure precludes doing so.

The specializer implements two optimizations described in [20]: thread/cache blocking in phase 4 (target-class optimization) and register blocking in phase 5 (target-specific optimization). The class of stencils it can specialize is nontrivial but incomplete: only a single output grid is allowed, which precludes specializing many important kernels. Future work will expand this class, and any stencil outside the class is executed in pure Python with no changes to source code. All results in this paper use double-precision data, but the specializer also supports single-precision and will soon support integers.

Kernel Results: We show stencil specializer results for two kernels (a 3D 7-point Laplacian kernel and a 3D 7-point divergence kernel) which are discussed in detail in previous work on auto-tuning and occur in a variety of applications including climate simulation [20]. These computations are memory bandwidth-bound, and the roofline performance model [32] tells us the strategy to obtain the best performance is to reduce capacity cache misses.

Figure 6 compares the performance of the two kernels against that of the Pochoir stencil compiler [28], which offline-compiles a DSEL embedded in C++ into output targeted for the Intel C++ compiler. Unlike our cache-aware algorithm, Pochoir implements a cache-oblivious algorithm and runs sub-problems using Intel Cilk+. Because Pochoir's cache-oblivious algorithm benefits from blocking across timesteps, we show results for both a single timestep and for the average over 5 timesteps of a stencil. In all cases, the pure-Python code took three to four orders of magnitude longer to run, and is therefore not shown.

For both the divergence and Laplacian kernels, the specializer slightly outperforms Pochoir for a single timestep. Since Pochoir can take advantage of temporal locality be-

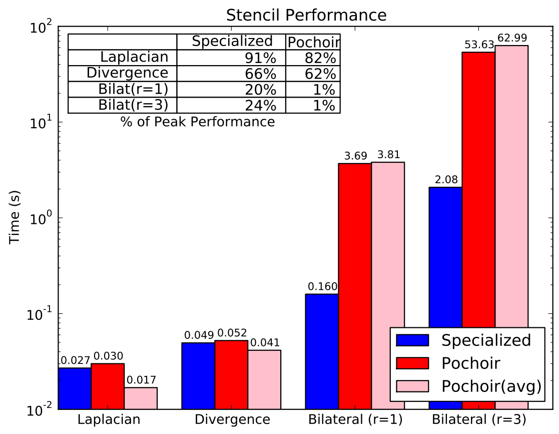


Figure 6. Summary of stencil performance (log-scale) on an Intel Core i7 870 (2.93 GHz) for 3D grid sizes of 258^3. Bilateral filter radius is shown as r. Since Pochoir can benefit from temporal locality across timesteps, we also show the average per-timestep time for Pochoir over 5 iterations. In all cases, the pure Python performance (not shown) was at least 3 orders of magnitude slower than specialized performance.

tween timesteps (theoretically reducing the cache traffic to minimum [16]), we expect it to outperform our tuner for multiple timesteps; our results show that it is faster when taking advantage of temporal locality. We have very recently implemented DSEL extensions in the stencil specializer to express multiple timesteps, and modified the code generator to output explicit cache-aware code that reduces memory traffic to the theoretical minimum; this work is currently being prepared for publication. Therefore, we report only results from our non-timestep aware DSEL.

Performance as a percentage of single-timestep roofline peak is also in Figure 6. For the Laplacian kernel, our specializer obtains 91% of peak memory bandwidth, and 66% for the divergence kernel. Note that the peak memory bandwidth on the machine is dependent on the number of memory streams; thus the peak for the divergence kernel is higher than for the Laplacian kernel.

Application Results: The application is a 3D bilateral filter for reducing noise and enhancing important features in MRI (magnetic resonance imaging) images of the brain [21]. It combines spatial and signal weights to create an edge-preserving smoothing filter. Such an application requires applying the filter with varying radii to highlight features of different sizes. The application loads MRI data using the NiBabel neuro-imaging library (`nipy.sourceforge.net`) and displays output data using matplotlib (`matplotlib.sourceforge.net`), both conveniently available to Python programmers; this illustrates the benefit of selecting a popular common embedding language with broad library support.

Unlike the previous two kernels, the bilateral filter uses all neighboring points within the set radii; for example, this means it is a 27-point stencil with $r = 1$ and a 343-point stencil with $r = 3$. In addition, the weight of each neighbor point is determined through an indirect table lookup into a small array, indexed by the intensity difference and multiplied by a function of the distance; this lookup can potentially result in much slower performance. For this application, the roofline model shows it is bound by in-core computation (because the lookup table is in cache, the computation required to compute the index is more costly), and from the mix of floating point and non-floating point operations, we can expect to obtain at most 62% of peak floating point performance on our test machine.

Figure 6 shows the performance of the computationally-intense portion of the bilateral filter application. Remarkably, the performance is even better than what is obtained by Pochoir, perhaps due to assumptions the Pochoir compiler makes about the stencil function (particularly related to indirect accesses). As shown in Figure 6, the specialized bilateral filter kernel obtains up to 24% of maximum floating point performance, which could be increased were our code better vectorizable. This performance is over three orders of magnitude faster than pure Python and is excellent for a computation-bound problem.

It is important to contextualize these performance gains by also examining the productivity improvement realized by using our stencil DSEL. As a gross metric, the lines of code for the optimized generated stencils are at least an order of magnitude larger; in addition, they encapsulate a large amount of domain-specific tuning knowledge, such as which order to traverse the grids, blocking for cache and registers/vectorization, etc. Furthermore, the optimal parameters change depending on the particular stencil problem. Pochoir helps reduce the necessity of programmers needing to know these low-level details to some extent, but is hampered by the need to write code in C++ as well as requiring an offline-compilation process. Overall, the stencil specializer produces very fast code with high productivity, enabling domain programmers to write in a high-level language and gain the performance of hand-tuned parallel low-level programming.

4.2 Gaussian Mixture Model Training

Gaussian Mixture Models (GMMs) are a class of statistical models used in speech recognition, image segmentation, document classification and numerous other areas. To apply GMM-based techniques to a particular problem, GMMs must be "trained" to match a set of observed data. The most common training algorithm, Expectation Maximization (EM), is computationally intensive, iterative, and highly data-parallel, making it particularly amenable to specialization for multicore hardware with wide SIMD vector support.

Why a Specializer: Current parallel implementations of the EM algorithm such as [25] employ a fixed strategy for mapping the algorithm's data-parallelism onto the parallel hardware. However, the best-performing mapping for various

EM algorithm substeps depends on both the GMM problem size and hardware platform parameters [11].

Details of Specializer: The specializer can emit either CUDA or Cilk+ code using two included sets of templated implementations. The specializer selects the best algorithm variant at runtime based on the size of the training problem and the available hardware, relying primarily on the SEJITS framework's templating mechanisms to instantiate the variant.

Kernel Results: Figure 8 shows that the GMM training performance of our specializer can beat even the handcoded CUDA [25] implementation by selecting the best-performing algorithmic variant at runtime based on training problem size [11]. The specializer can emit CUDA and Cilk+ code, making it performance-portable both within and across architecture families with no changes to client Python applications like the one we describe below.

Application Results: Our target application, a meeting diarizer [1] that identifies the number of speakers in a recorded meeting and determines who spoke when, originally consisted of about 3000 lines of C++ with pthreads. Repeated GMM training was the performance bottleneck. The new implementation consists of about 100 lines of Python implementing the body of the application, plus a specializer consisting of about 800 lines of Python and 3600 lines of CUDA and Cilk+ templates. The reason the specializer is so large is because it integrates a large number of tuned implementations that were not present in the original application's GMM implementation, including support for multiple backends. Note also that the specializer is not specific to this application; that is, a number of other applications have been developed that utilize the specializer, in a variety of domains [11].

Speech recognition domain experts evaluate performance according to the real-time factor ($\times$RT) metric. For example, $100\times$RT means that 1 second of audio can be processed in $1/100$ second. An important domain target is $\geq 200\times$RT, at which point online approaches to speaker diarization become fast enough to obviate further research in offline approaches.

Figure 7 shows that our Python implementation with specialized EM training achieves 50% of that goal, using the CUDA and Cilk+ specializers, while the original C++/pthreads application only gets to 10%. We are confident that future work on specializers for other components of the application will allow us to meet the domain target of $200\times$RT.

4.3 Graph Algorithms (Linear Algebra)

Applications based on graph algorithms include network analysis, bioinformatics, search, recommendations, directions, and recognition. Graph algorithms can be difficult to implement efficiently due to irregular memory access patterns that destroy locality, making the algorithms memory-bound.

Many graph frameworks and libraries have arisen to support graph algorithm research. These frameworks typically provide an infrastructure that schedules and runs a programmer-specified operation on the graph. Two widely-used programming models supported by such frameworks are *linear algebra* and *bulk synchronous parallel (BSP)*. In the linear algebra approach, a graph algorithm is transformed into an equivalent problem involving matrix algebra, whose solution corresponds to the desired graph algorithm result, such as breadth-first search [8]. In BSP, the developer specifies a function to apply to each vertex at each time step, and an execution engine schedules the visiting of vertices. The step function can typically send messages to other nodes, and read messages sent to it in previous time steps. We have created specializers for both approaches. We first describe the linear-algebra graph specializer, which extends the functionality of the Knowledge Discovery Toolbox (KDT, `kdt.sourceforge.net`).

Mic Array	**Orig. C++/pthreads**	**Py+Cilk+**	**Py+CUDA**
	Westmere	Westmere	GTX285/GTX480
Near field	$20\times$	$56\times$	$101\times$ / $117\times$
Far field	$11\times$	$32\times$	$68\times$ / $71\times$

Figure 7. Diarizer application performance as a multiple of real time; "$100\times$" means that 1 second of audio can be processed in $1/100$ second. The Python application using CUDA and Cilk+ outperforms the native C++/pthreads implementation by a factor of 3-6.

Why a Specializer: KDT provides an execution engine and set of C++ operators based on the Combinatorial BLAS [7] that form the core of many graph computations. Python bindings to the C++ operators allow domain experts to express their algorithms in Python and get good performance, but when domain experts implement new graph operators as Python methods, the overhead of upcalling into Python to execute them makes the code $80\times$ slower.

Specializer Details: Our KDT specializer allows the user to provide a standard Python method to be used in a graph algorithm. The specializer uses our framework's facilities to lower CombBLAS operators expressed in Python (see Figure 10). Unlike our other specializers that do all their work at runtime, KDT's usage model involves running a compile-time script that uses the Simplified Wrapper and Interface Generator package (`swig.org`) to generate Python bindings for such extension methods. Supporting this model was simplified by the modular structure of specializers: we replaced the default framework-provided backend compilation, which calls CodePy, with a new one that generates C++ code and the required SWIG files and then calls SWIG just as KDT would. Thus, the modularity of our approach simplifies interoperability with existing frameworks.

Kernel Results: We measure the performance of KDT+SEJITS on Graph 500 (`graph500.org`), a set of benchmarks intended to test performance on data-intensive graph algorithms. Benchmark 1 of Graph 500, "Search," consists of generating a graph and finding breadth-first search trees from

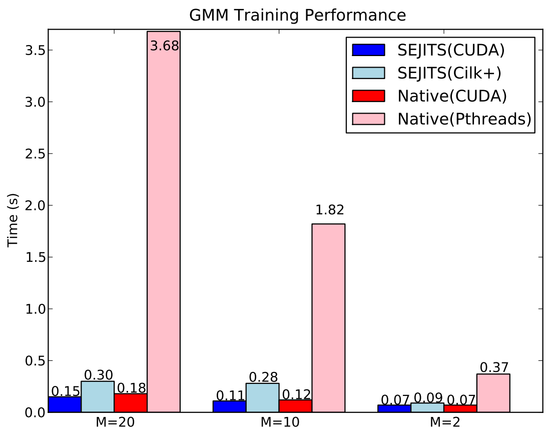


Figure 8. GMM training performance given number of mixture-model components M, which varies as algorithms converge, using the CUDA backend and a native CUDA version [25] (both on NVIDIA GTX480), and the Cilk+ backend and a C++/pthreads version (both on dual-socket Intel X5680 Westmere 3.33GHz).

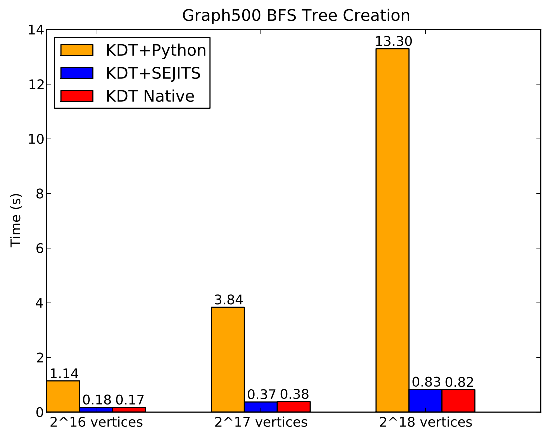


Figure 9. Graph 500 tree generation performance for R-MAT scale factors 16, 17, and 18 on an Intel Core i7-870 processor (2.93 GHz). SEJITS allows coding new KDT functions in Python while matching the performance of KDT's native C++ functions.

```
class BenchmarkOperator (KdtOperator):
    def g500_op(x, y):
        if y == -1:
            return x
        return -1

BenchmarkOperator([
        Operator("g500_op", assoc=True, comm=True)
        ])
```

Figure 10. Python source code for specializing an element-wise binary operator with the KDT specializer.

randomly selected parents. The traditional method for creating an implementation of the benchmark application would be to either write optimized C++ parallel code, or write Python methods for use with KDT and incur the 80× slowdown of upcalling to Python. However, our specializer enables us to write this code in Python, as seen in Figure 10, and get performance equivalent to that obtained when using only the built-in C++ operators of KDT. The results are shown in Figure 9. The prototype specializer omits some optimizations that hand-written C++ operations might use, but because so much speedup occurs with even just plain translation, even this non-optimized approach is very useful. Indeed, we are working with KDT developers to integrate specialization into the next version of KDT for enabling graph algorithms on semantic graphs.

4.4 Communication-Avoiding Sparse Linear Algebra

Many algorithms for solving sparse linear systems ($Ax = b$), or for finding eigenvalues of a sparse matrix, are iterative processes that access the matrix A with one or more sparse matrix-vector multiplications (SpMVs) per iteration. Since an SpMV must read a matrix entry from memory for every 2 useful floating-point operations, Demmel et al. have proposed *communication-avoiding* algorithms that improve performance by trading redundant computation for memory traffic [22]. Both serial and parallel implementations can benefit from these algorithms, as communication in the serial case refers to memory-to-cache traffic.

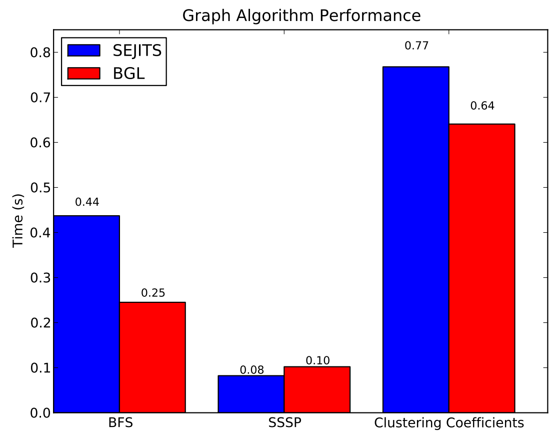


Figure 11. Summary of graph algorithm performance on an Intel Xeon X5680 3.33 GHz for graph libraries that do not use the linear algebra representation. BFS and Clustering use Graph500 graphs of SCALE=20 and SCALE=14 respectively. SSSP is run on webbase-1M from (`www.cise.ufl.edu/research/sparse/matrices`). Results for NetworkX and graph-tool are not shown because they are more than two orders of magnitude slower.

We have implemented a specializer for a communication-avoiding *matrix powers kernel* (so-called $\mathbf{A}^k\mathbf{x}$). Matrix powers computes $Ax, A^2x, \ldots, A^kx$ (or some equivalent

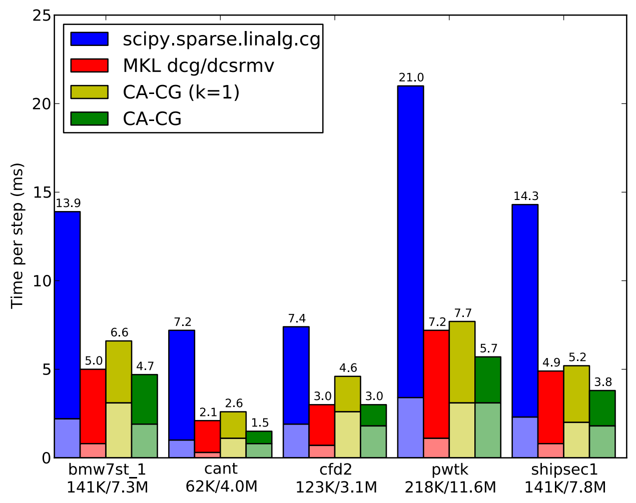


Figure 12. Conjugate Gradient solver performance using communication-avoiding matrix powers kernel on a dual-socket Intel Xeon X5550 (2.67GHz) on test matrices from finite-element and fluid dynamics applications (`www.cise.ufl.edu/research/sparse/matrices`). A matrix labeled 141K/7.3M has 141K rows and 7.3M nonzero elements. The dark part of each bar shows time spent on matrix powers while the light part shows time in the remainder of the solver. Note that the convergence properties are at most 4.5% worse for 10^{-6} reduction in $|r|^2$; however, we report time per step since the decision to use CA-CG is independent of our tuner.

basis that spans the same vector space) for matrix A, vector x, and a small constant k. $\mathbf{A}^k\mathbf{x}$ is an important ingredient in Krylov-subspace solvers such as Conjugate Gradient (CG), because once the computation has been performed, the next k steps of the solver can proceed without further memory accesses to A by combining vectors from this set.

Why a Specializer: Although this specializer does not lower any user-provided code, the tuning logic associated with blocking and tiling (which requires inspecting input values, as described below) was easy to write in Python. To the best of our knowledge, our $\mathbf{A}^k\mathbf{x}$ specializer is the first publicly available productivity-friendly implementation of communication-avoiding $\mathbf{A}^k\mathbf{x}$.

Details of Specializer: The specializer partitions the set of matrix rows into cache blocks, and computes a cache block's entries in all k output vectors before moving on to the next cache block. This will only work if the matrix structure is such that the dependencies between cache blocks do not get too large. In addition to cache-blocking the matrix, the usual SpMV optimization of register tiling reduces the memory size of the matrix and makes it possible to use SIMD instructions. The specializer generates code for different blocking and tiling formats using templates, and chooses among them by auto-tuning. (The choice of k must be made outside the specializer, as it affects the rest of a Krylov-subspace method too.)

Kernel and Application Results: We have implemented a Communication-Avoiding Conjugate Gradient (CG) solver, the simplest useful Krylov-subspace method solver, in Python. The CG solver uses the $\mathbf{A}^k\mathbf{x}$ specializer and calls the Intel Math Kernel for the operations other than matrix powers. Although MKL is well-optimized, the ability to compose the $\mathbf{A}^k\mathbf{x}$ computation with the solver's subsequent dot product operations would avoid having to read the vectors from memory an extra time. (As described later, specializer composition is an important avenue for future work.) Nonetheless, our CG solver runs several times faster than SciPy's serial and non-communication-avoiding efficiency language implementation, and in fact is even faster than MKL's optimized parallel CG implementation. The convergence properties for our test matrices differ slightly between the two CG algorithms on a few of the matrices (at worse requiring 4.5% more multiplies), but the decision as to whether the difference is meaningful depends on the application.

Figure 12 shows the results, demonstrating both the algorithmic and auto-tuning benefits of our CG implementation.

5. Discussion

5.1 DSELs and Runtime Code Generation

Structurally, the Asp framework provides two distinct sets of capabilities—DSL embedding and runtime code generation—that work particularly well when combined: DSELs with well-chosen abstractions allow capturing programmer intent rather than implementation, simplifying the task of generating good code, while runtime code generation allows the target code to make late decisions based on details of the target platform and even the input data for each problem instance. Indeed, the matrix powers ($\mathbf{A}^k\mathbf{x}$) specializer of Section 4.4 is not really a DSEL at all but a single method call, yet the characteristics of the problem (dependence on structure of input matrices, knowledge of cache geometry for tiling and blocking, etc.) made it appealing to use Asp's runtime code generation as the delivery mechanism for the specializer, and as an added benefit allowed the optimized code to be called from Python. Conversely, a separate specializer on which we have not reported in this paper generates Java code for Hadoop (an open source implementation of Map/Reduce), allowing Python programs to embed other specializers in the body of a Map/Reduce computation. In that case, the DSEL embedding was more useful than the ability to generate optimized code at runtime.

We conclude that the ability to embed lightweight DSELs and construct their compilers is separate from the ability to execute those compilers at runtime, but combining the two mechanisms opens new opportunities for bridging the productivity/performance gap.

5.2 Disadvantages of DSLs

There are two main disadvantages to a DSL. The first is that programmers must learn a new language with a new syntax. We address this problem by *embedding* the domain-specific languages in a common host language. DSELs ease integration with existing code and allow developers to use a familiar syntax, making them, to a developer, appear as simple as using libraries.

The second disadvantage is that the DSL implementor must invest substantial effort in tasks tangential to the problem domain, such as parsing the source language, performing generic optimizations, emitting the target language, and providing facilities for general computation that are outside the problem domain (file manipulation, e.g.). To this end, Asp is essentially a framework for doing JIT code generation of kernel-specific embedded DSLs, specifically designed to make it easy to capture a human-expert-created computation strategy.

5.3 Reusability and Extensibility

Beyond simply moving the complexity of getting good performance from one site in the application to another, specializers are easily usable in new Python applications, and some such as the GMM specializer can even transparently target either multicore CPU or GPU at runtime depending on hardware availability. Reuse can be increased by providing existing specializers with additional code-generation back ends (for new hardware or compilers) or additional DSEL front-ends (for exposing the specializer to other productivity languages).

While the most visible customers of Asp specializers are productivity programmers, Asp allows efficiency programmers who devise and optimize highly-performant code to encapsulate their strategy in a specializer, greatly increasing the potential for flexible reuse of that strategy. Although the initial effort for writing a specializer is more work than a one-off optimization of a particular application, we argue that the benefit is far greater— many more people can take advantage of the knowledge of an efficiency-level programmer through a specializer, and the overall effort is less than if the efficiency-level programmer were to hand-optimize several instances of computations in the same domain.

Asp also provides an incremental adoption path for efficiency-level programmers: they can take an existing prototype written in an efficiency-level language, rapidly encapsulate it in a Asp specializer, and then gradually refine and generalize it over time. A common path is to embed an existing prototype in a template (see Section 3) and gradually add more "holes" into the template, into which runtime-computed values are substituted. Eventually an entire function or functions may be generated at runtime using tree transformation techniques. Automated tests help to ensure behavior is preserved during this refactoring. This incremental adoption strategy proved valuable in practice for our early adopters: the GMM and $\mathbf{A}^k\mathbf{x}$ specializer authors both began with a specific efficiency-level implementation that had limited support for sophisticated optimizations or multiple targets, and then added these features as the specializers were generalized.

5.4 Performance Portability

The SEJITS approach provides performance portability: if the specializer can generate code for the target platform (e.g. x86 multicore) then the same Python application will get high performance across many machines; if the specializer can target multiple platforms (e.g. GPUs as well as multicore) then the same application will get high performance even across platforms.

If no specializer exists, the code is still source-portable since it can run in unmodified Python. The portability aspect of productivity languages remains; with the SEJITS approach, that portability also extends across architectures and platforms.

6. Related Work

Going beyond libraries for domain experts. A popular way to provide good performance to productivity-language programmers has been to provide native libraries with high-level-language bindings such as SciPy (`scipy.org`), Biopython (`biopython.org`), BLAS [6], ScaLAPACK [5], and FFTW [15].

However, some DSEL benefits are difficult to achieve with libraries. One difficulty lies in conditioning code generation on the input problem parameters, as our GMM and $\mathbf{A}^k\mathbf{x}$ specializers do. The OSKI (Optimized Sparse Kernel Interface) sparse linear algebra library [31] precompiles 144 variants of each supported operation based on install-time hardware benchmarks that take hours to run, and includes logic to select the best variant at runtime, but applications using OSKI must still intermingle tuning code (hinting, data structure preparation, etc.) with the code that performs the calls to do the actual computations

Another difficulty is the frequent mismatch between the library's API and the preferred domain abstractions. For example, whereas solving a matrix in MATLAB is as simple as writing $X = A\backslash B$, the same operation in ScaLAPACK requires the application programmer to determine the processor layout, initialize array descriptors for each input and output matrix, load the data so that each processor has the correct portions of the input data, and finally call the solve function. A higher-level language could be layered over ScaLAPACK to reduce this impedance mismatch; indeed, our approach does just this, but takes the additional step of embedding the higher-level language into a common host language and allowing greater flexibility in how the language is JIT-compiled.

Finally, most widely-used libraries written in efficiency languages do not gracefully handle higher-order functions as we needed for the stencil and graph specializers—even if the productivity language from which they're called does support them. This is usually because the efficiency languages do not have well-integrated support for higher order functions themselves. Even libraries such as Intel's Array Building Blocks (`http://software.intel.com/en-us/articles/intel-array-building-blocks/`) or others using

C++ expression templates cannot benefit from all the runtime knowledge available to DSEL compilers. Because SEJITS allows JIT compilation *and* allows DSELs to leverage the host language's support of higher-order functions, it supports the tuning of such computations by lowering and inlining the interior functions. Thus, from the productivity programmer's point of view, the experience of using SEJITS DSELs is not only similar to a library, but idiomatic in the productivity language.

DSLs for bridging performance and productivity. Like the Delite project [10] (the most similar recent work to our own), we exploit domain-specific information available in a DSEL to improve compilation effectiveness. In contrast to that work, we allow compilation to external code directly from the structural and computational patterns expressed by our DSELs; in the Delite approach, DSELs are expressed in terms of lower-level patterns and it is those patterns that are then composed and code-generated for. Put another way, by eschewing the need for intermediate constructs, SEJITS "stovepipes" each DSEL all the way down to the hardware without sacrificing either domain knowledge or potential optimizations. The most prominent example of this is that SEJITS allows using efficiency-language templates to implement runtime-generated libraries (or "trivial" DSLs). Furthermore, auto-tuning is a central aspect of our approach, enabling high performance without needing complex machine and computation models.

The Weave subpackage of SciPy allows users to embed C++ code in strings inside Python code; the C++ code is then compiled and run, and uses the Python C API to access Python data. Cython (`cython.org`) is an effort to write a compiler for a subset of Python, while also allowing users to write extension code in C. Both of these expose low-level interfaces for efficiency programmers. Closer to our own approach is Copperhead [9], which provides a Python-embedded DSEL that translates data-parallel operations into CUDA GPU code.

Auto-tuning. The idea of using multiple variants with different optimizations is a cornerstone of auto-tuning. Auto-tuning was first applied to dense matrix computations in the PHiPAC library (Portable High Performance ANSI C) [4]. Using parameterized code generation scripts written in C, PHiPAC generated variants of generalized matrix multiply (GEMM) with a number of optimizations plus a search engine, to, at install time, determine the best GEMM routine for the particular machine. The technology has since been broadly disseminated in the ATLAS package (`math-atlas.sourceforge.net`). Auto-tuning libraries include OSKI (sparse matrix-vector multiplication) [31], SPIRAL (Fast Fourier Transforms) [27], and stencils [20, 28], in each case showing large performance improvements over non-autotuned implementations. With the exception of SPIRAL and Pochoir, all of these code generators use ad-hoc Perl or C with simple string replacement, unlike our template and tree manipulation systems.

Specialization. Early work on specialization appeared in the Synthesis Kernel, in which a code synthesizer specialized kernel routines on-the-fly when possible [26]. Engler and Proebsting [14] illustrated the benefits of dynamically generating small amounts of performance-critical code at runtime. Jones [17, 19] and Thibault and Consel [30] proposed a number of runtime specialization transformations to increase performance of programs, including partial evaluation or interpreters customized for specific programs. Despite their different contexts, these strategies, like SEJITS, rely on selectively changing execution of programs using runtime as well as compile-time knowledge.

Just-in-time code generation and compilation. Sun's HotSpot JVM [24] performs 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 productivity-language code. Our approach is more in the spirit of Accelerator [29], 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. Asp is more general in allowing specializer writers to use the full set of Python features to implement specializers that capture their problem-solving strategy.

7. Future Work

To support auto-tuning, we are building a global database that will aggregate runtime information (performance, hardware characteristics, and problem parameters) from specializers running at different sites. These data will enable future specializer invocations to base tuning parameters on information from similar problems on similar machines. As with specializers that take advantage of the input problem data, there is a tradeoff between the efficiency gained by using such information and the cost of repeating one or more specializer phases when the inputs are altered in a way that affects the result. We intend to investigate this tradeoff using modeling techniques based on machine learning.

We are investigating how best to compose specializers. We believe many combinations of Semantic Models can be composed while preserving both their productivity-level abstractions and the efficiency-level performance of the composition. Like database query optimization [2], we have a tree of operators communicating over edges, each of which may afford multiple implementations. Concepts from parallel/distributed query optimization, including independent parallelism, pipelined parallelism, partitioned parallelism [13] and adaptive query processing [3] could be applied to help allocate parallel resources to specializers and select strategies that allow specializers to cooperate effectively.

Our prototype SEJITS framework uses Python as both the embedding language and the specializer implementation language, but these tasks are logically separate. Any modern scripting language that supports introspection, dynamic linking at runtime, and has a reasonable foreign function interface will serve as an embedding language. Ruby, Scala and Lua, for example, all have these properties.

Errors in a complex specializer, manifesting as incorrect behavior of generated code or cryptic feedback to the application programmer, can be difficult to diagnose due to its dynamic and language-crossing nature. Tool support for reproducing and isolating errors, visualizing and verifying code transformations, producing clear error messages, and modular testing of specializers is essential.

Finally, we have only begun building specializers for a few domains. Other specializers, especially those expressing higher-level patterns such as Map, are in development as a response to application needs. We are also investigating other target platforms for specializers such as public cloud computing.

8. Conclusions

We have attempted to support two claims applying to two different types of SEJITS stakeholders. For *efficiency programmers* accustomed to writing high-performance code, SEJITS and the Asp library make their work more easily available and widely reusable by productivity programmers than would a standalone compiled library. For *productivity programmers*, SEJITS provides both parallel efficiency-level performance and performance portability across hardware, all with the same effort as sequential productivity programming.

Because each specializer is essentially a self-contained mini-compiler for a particular DSEL, new specializers can be continuously added to the ecosystem (similar to the CPAN, RubyForge, or other library repositories). We hope others will contribute to this ecosystem and help raise the level of abstraction for productivity programmers while taking advantage of state-of-the-art efficiency-language parallel code.

References

[1] X. Anguera, S. Bozonnet, N. W. D. Evans, C. Fredouille, G. Friedland, and O. Vinyals. Speaker diarization : A review of recent research. *Accepted for publication in "IEEE Transactions On Acoustics Speech and Language Processing" (TASLP), special issue on "New Frontiers in Rich Transcription", 2011*, 2011.

[2] P. Apers, A. Hevner, and S. Yao. Optimization algorithms for distributed queries. *Software Engineering, IEEE Transactions on*, SE-9(1):57 – 68, jan. 1983. ISSN 0098-5589.

[3] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, pages 261–272, New York, NY, USA, 2000. ACM. ISBN 1-58113-217-4.

[4] 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.

[5] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, 1997. www.netlib.org/scalapack.

[6] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of Basic Linear Algebra Subroutines (BLAS). *ACM Trans. Math. Soft.*, 28(2), June 2002. www.netlib.org/blas.

[7] 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.

[8] A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. *CoRR*, abs/1104.4518, 2011.

[9] 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.

[10] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In C. Cascaval and P.-C. Yew, editors, *PPOPP*, pages 35–46. ACM, 2011. ISBN 978-1-4503-0119-0.

[11] 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.

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

[13] A. Deshpande and L. Hellerstein. Flow algorithms for parallel query optimization. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 754–763, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-1-4244-1836-7.

[14] 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. ISBN 0-89791-660-3.

[15] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[16] M. Frigo and V. Strumpen. Cache oblivious stencil computations. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 361–366, New York, NY, USA, 2005. ACM. ISBN 1-59593-167-8.

[17] R. Glück and N. D. Jones. Automatic program specialization by partial evaluation: an introduction. In *Software Engineering*

in Scientific Computing, pages 70–77, 1996.

[18] 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.

[19] N. D. Jones. Transformation by interpreter specialisation. *Sci. Comput. Program.*, 52:307–339, August 2004. ISSN 0167-6423.

[20] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *IPDPS'10*, pages 1–12, 2010.

[21] K. McPhee, C. Denk, Z. Al Rekabi, and A. Rauscher. Bilateral filtering of magnetic resonance phase images. *Magn Reson Imaging*, 2011.

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

[23] F. Ortigosa, M. Araya-Polo, F. Rubio, M. Hanzich, R. de la Cruz, and J. M. Cela. Evaluation of 3d rtm on hpc platforms. *SEG Technical Program Expanded Abstracts*, 27(1):2879–2883, 2008.

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

[25] A. D. Pangborn. Scalable data clustering using gpus. Master's thesis, Rochester Institute of Technology, 2010.

[26] C. Pu, H. Massalin, and J. Ioannidis. The synthesis kernel. *Computing Systems*, 1:11–32, 1988.

[27] 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, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[28] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The Pochoir stencil compiler. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0743-7.

[29] 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.

[30] S. Thibault, C. Consel, S. T. C. Consel, J. L. Lawall, and R. M. G. Muller. Static and dynamic program compilation by interpreter specialization. In *Higher-Order and Symbolic Computation*, pages 161–178, 2000.

[31] R. Vuduc, J. W. Demmel, and K. A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics Conference Series*, 16(i):521–530, 2005.

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

[33] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. *Parallel and Distributed Processing Symposium, International*, 0:171, 2000. ISSN 1530-2075. doi: http://doi.ieeecomputersociety.org/10.1109/IPDPS.2000.845979.