Auto-tuning the Matrix Powers Kernel with SEJITS

Jeffrey Morlan, Shoaib Kamil, and Armando Fox

Computer Science Division, University of California at Berkeley
Berkeley, CA 94720, USA
{jmorlan,skamil,fox}@cs.berkeley.edu

Abstract. The matrix powers kernel, used in communication-avoiding Krylov subspace methods, requires runtime auto-tuning for best performance. We demonstrate how the SEJITS (Selective Embedded Just-In-Time Specialization) approach can be used to deliver a high-performance and performance-portable implementation of the matrix powers kernel to application authors, while separating their high-level concerns from those of auto-tuner implementers involving low-level optimizations. The benefits of delivering this kernel in the form of a specializer, rather than a traditional library, are discussed. Performance of the matrix powers kernel specializer is evaluated in the context of a communication-avoiding conjugate gradient (CA-CG) solver, which compares favorably to traditional CG.

1 Introduction

Krylov subspace methods (KSMs) are iterative algorithms in linear algebra used to solve linear systems (given matrix A and vector b, solve $Ax = b$ for x) or to find eigenvalues and eigenvectors (given A, solve $Ax = \lambda x$ for λ and x) when the matrix is large and sparse, making direct solvers impractical. The solution vectors these methods produce in the first i iterations lie in the vector space spanned by the vectors $\{x_0, Ax_0, \ldots, A^i x_0\}$ for some starting vector x_0; this kind of space is called a Krylov subspace.

Conventionally, KSMs 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 two useful floating-point operations, making it a highly memory-bound operation, Demmel et al. have proposed *communication-avoiding* algorithms that improve performance by trading redundant computation for memory traffic [1]. In communication-avoiding KSMs, SpMV is replaced by the *matrix powers kernel*, which computes $Ax, A^2x, \ldots, A^kx$ (or some equivalent basis that spans the same vector space) for matrix A, vector x, and a small constant k. 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. Thus, memory traffic can be reduced – by up to a factor of k in the best case – but obtaining the best performance requires substantial tuning.

A difficulty in auto-tuning the matrix powers kernel is that optimal code depends on not only the machine architecture but also on the specific problem instance (namely, the placement of nonzeros in the matrix A), so runtime auto-tuning is necessary to get high performance and performance portability. Moreover, it is desirable to separate concerns of the application writers using KSM solvers and programmers implementing auto-tuning. To do this, we take advantage of SEJITS (Selective Embedded Just-In-Time Specialization) [2], a programming methodology for maximizing separation of concerns between programmers working in specific problem domains and programmers who know how to write efficient low-level code for the kinds of computation used in these domains. The idea is to enable domain experts to express their applications in code without needing to deal with low-level optimizations. This is accomplished by defining a domain-specific language (DSL) or API, embedded in a high-level general purpose language such as Python. The efficiency expert, with the help of a SEJITS framework such as Asp (Asp is SEJITS for Python) [3], writes a *specializer* to compile efficient implementations for the domain-specific code.

This can be seen as a generalization of the common practice of writing a library in a low-level language with bindings allowing it to be called from a high-level language. The ability to generate code at runtime makes specializers applicable in cases where a library would not be able to provide sufficient flexibility and performance. This could be because the computation itself is too general for a library: an example of this is the domain of stencil computations [4]. It would not be possible to compile implementations of all possible stencils up front, but a specializer can lower a stencil function, given as code in its Python-embedded DSL, down to C++, making it capable of generating optimized code for arbitrary stencils.

Although the matrix powers kernel's computation is not application specific, since SEJITS can generate and compile code at runtime, this approach to delivering auto-tuning avoids the combinatorial explosion of code variants implied by the large space of possible optimizations. Application writers can get both high performance and performance portability without having to be concerned with the low-level optimizations making them possible. SEJITS also allows the tuning logic to be written in the high-level language, making it easier to write and maintain while still keeping it well separated from applications. These benefits would be difficult to obtain if the kernel were delivered as a conventional library.

This paper describes a specializer for the matrix powers kernel, which was built on the Asp framework. Section 2 describes the specializer and the various optimizations implemented in it, and Section 3 contains performance results. Finally, discussion of the benefits that SEJITS brings to this kernel is in Section 4.

2 Implementation

The overall structure of the specializer and code using it is shown in Figure 1. The application first calls the tuner, passing in the sparse matrix A and the

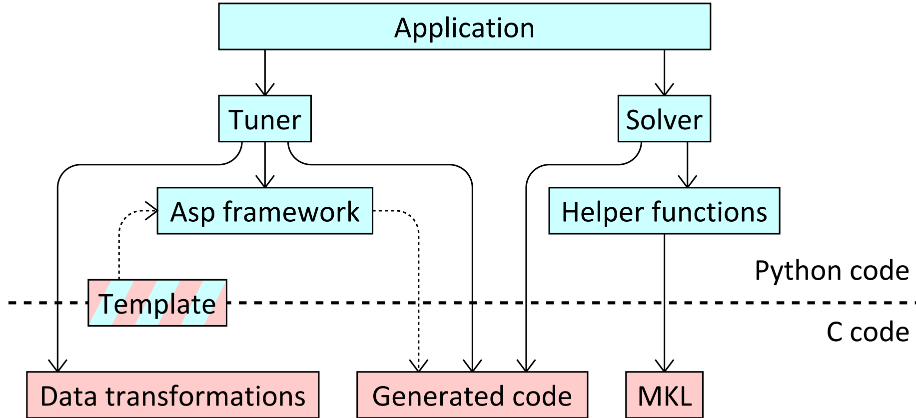

Fig. 1. Overview of specializer and related code. Solid arrows indicate calls, dotted arrows indicate processing of code.

constant k to be used. The tuner attempts to produce an optimized plan for computing matrix powers. To do this, it iterates over feasible ranges of the optimization parameters described in Section 2. It calls into C code to do the necessary transformations on the matrix data, uses the Asp infrastructure to generate from a template any specialized code necessary to execute the plans, and benchmarks the plans to find the fastest. Each candidate plan is benchmarked by running it in a loop until more than a half second has elapsed to get an accurate measurement of its execution time.

An object representing the fastest plan found is returned back to the application, which can then use it in a KSM solver. The solver invokes a method on the object to execute the matrix powers kernel; for other linear algebra operations that KSMs need, the specializer module also provides helper functions which are simply wrappers around Intel Math Kernel Library (MKL) [5] BLAS operations.

2.1 Optimizations

The optimizations of the matrix powers kernel, summarized in Table 1, fall into two major categories: those that reduce memory traffic by storing data more efficiently, and those that re-order computation to parallelize it or make better use of cache. The latter must obey the constraint that if matrix entry A_{ij} is nonzero, then for each level $e : 0 \leq e < k$, component j of $A^e x$ must be computed before it can be used in computing component i of $A^{e+1}x$. Because of this, their effectiveness is highly dependent upon the structure of the matrix's nonzero entries, making runtime auto-tuning necessary for best performance.

The first optimization is to allow for parallelism by *thread blocking*. The matrix rows are partitioned among a number of threads, with each thread being

Optimization	**Type**	**Restrictions**
Thread blocking	Re-ordering	
Explicit cache blocking	Re-ordering	Useful only when $k > 1$
Tiling	Size reduction	
Symmetric representation	Size reduction	$A = A^T$; square tiles only
Implicit cache blocking	Re-ordering	Useful only when $k > 1$; square tiles only
Index array compression	Size reduction	Block must be sufficiently small

Table 1. Summary of optimizations.

responsible for computing the vector components whose indices are that of rows in its partition. In general, however, one thread's set of $A^{e+1}x$ components will depend on a few components of $A^e x$ belonging to other threads. To avoid having dependencies across threads which would force synchronization after each output vector and preclude the cache blocking described later, a thread block contains not only the rows in its partition, but also any additional rows needed for redundantly computing other threads' components as shown in Figure 2.

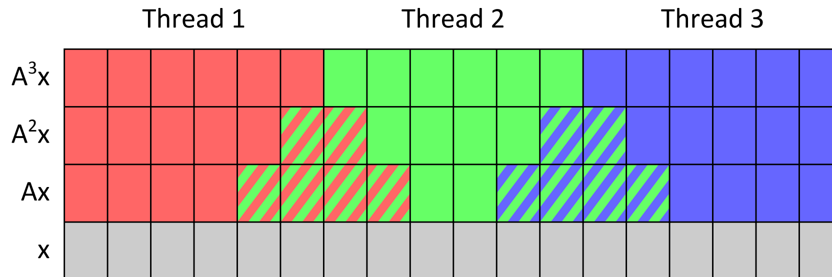

Fig. 2. Thread blocking of an 18x18 tridiagonal matrix. Colors indicate which thread computes each component; the striped components are redundantly computed by two different threads.

Minimizing the redundant computation means choosing a good partitioning, i.e. one with few dependencies between thread blocks. One way would be to make a graph with one vertex per matrix row, add an edge between vertex i and vertex j whenever $A_{ij} \neq 0$, and partition this graph. For a more accurate model of communication volume, what is implemented is to build a hypergraph in which each vertex has a net containing all vertices that have a dependency on it in k steps [6] (that is, net i contains vertex j if $(A^T + I)^k_{ij} \neq 0$, ignoring cancellation), and partition the hypergraph with the PaToH [7] library. Because this can be very time-consuming for larger k, the current tuner only partitions a $k = 1$ hypergraph regardless of the actual value of k.

If a thread block is too large to fit in the processor's cache, then without further division it would be read from RAM k times. We can improve this with *cache blocking*: divide each thread block into sufficiently small cache blocks, and compute the entries for all k vectors in one cache block before moving on to the next. This way, the contents of the thread block are only read once for all k vectors. *Explicit cache blocking* is done in an identical manner to thread blocking; each thread block is simply subdivided recursively until each piece is small enough. An alternative is implicit cache blocking, which is done later on.

Nonzero entries in a sparse matrix are often close together, and this can be taken advantage of by *tiling*. By default, blocks are stored in compressed sparse row (CSR) format, which consists of three arrays: an array of nonzero values (one floating point number for each nonzero), an array of column indices corresponding to those values (one index for each nonzero), and an array indicating where each row begins and ends (one index for each row plus one more, since the end of one row is the start of the next). Tiling a block modifies this to store some fixed-size tile instead of individual values; a tile is stored if any of its individual values are nonzero (Figure 3). This results in a larger values array to hold the extra zeros, but smaller column index and row pointer arrays, which can often make for a net decrease in size. When either dimension of the tile size is even, it also becomes possible to use SIMD instructions to do two multiply-adds at a time, which is implemented via compiler intrinsics.

In many applications, the matrix is *symmetric*, meaning that $A_{ij} = A_{ji}$ for all i, j. When this is the case, the leading square of every block is symmetric as well, since the columns are permuted into the same order as the rows. All entries below the main diagonal can be omitted without losing any information, as they are merely reflections of entries above the diagonal; this optimization can reduce the memory size of a block by almost half. However, it alters the structure of the computation: computing component i now requires going through not only the entries in row i, but also any entry with a column index of i. This adds additional dependencies between components for the purposes of implicit cache blocking, possibly reducing its efficacy.

Unlike the partitioning of the matrix A into thread blocks or of a thread block into explicit cache blocks, where each block is internally stored as a separate

$$\begin{bmatrix} 4 & 5 & & \\ 6 & 7 & & \\ & & & 8 \\ & & 9 & \end{bmatrix}$$

`values:` 4 5 6 7 8 9
`colidx:` 0 1 0 1 3 2
`rowptr:` 0 2 4 5 6

$$\begin{bmatrix} 4 & 5 & & \\ 6 & 7 & & \\ & & 0 & 8 \\ & & 9 & 0 \end{bmatrix}$$

`values:` 4 5 6 7 0 8 9 0
`colidx:` 0 1
`rowptr:` 0 1 2

Fig. 3. A 4x4 block and its representation in compressed sparse row format, before and after 2x2 tiling.

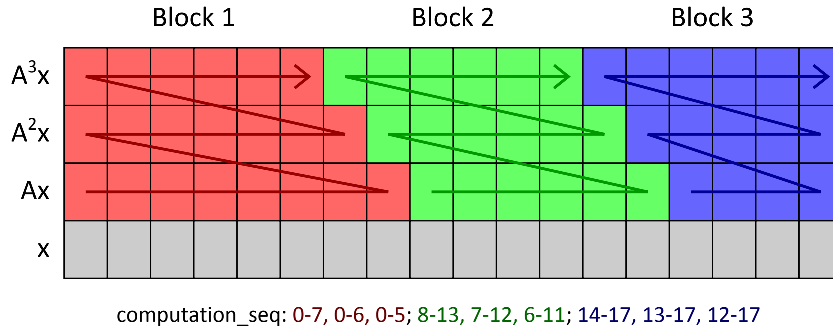

Fig. 4. Implicit cache blocking. Arrows represent the order of computation within each block; the left block is done first, then the middle block, then the right.

matrix, in *implicit cache blocking* (Figure 4) the partitioning into cache blocks is not reflected in the internal data structures in this way. Instead, an array is created that lists the indices of components that need to be computed at each level of each cache block; this array determines the sequence to perform the computation in, which would otherwise simply be one level after another. Since this array will often contain long sequences of increasing integers, it may be stanza-encoded. There is no need for any redundant computation: while creating the array, keep track of what level each entry will have been computed up to at the current point, and just omit any redundancy.

Finally, arrays of indices in each block can be compressed from 32-bit to 16-bit if the block is sufficiently small. If a block is implicitly cache blocked and has fewer than 2^{16} rows, the computation sequence can be compressed. If any block has no more than 2^{16} columns, its `colidx` array can be compressed. With fewer than 2^{16} nonzero tiles, the `rowptr` array can be compressed as well.

The tuner logic currently works as follows: for each possible number of threads (specified in the configuration), both explicit and implicit cache blocking are attempted. For explicit blocking, the tuner iterates over a range of maximum block sizes (5M bytes down to 250K, dividing by 2 each time) recursively bisecting each thread block until all cache blocks are below the maximum size. For implicit blocking, it iterates over a range of the number of implicit blocks per thread (1 to 256, multiplying by 2 each time). With both types of cache blocking, the tile size of each explicit cache block or thread block is chosen to give the smallest memory footprint. Symmetric representation and index compression are used if possible, but implicit blocking is tried both with and without symmetric representation.

3 Results

To test the specializer in a realistic context, we have implemented in Python a communication-avoiding variant of the conjugate gradient (CG) method, a Krylov method for solving symmetric positive definite linear systems. The basic structure of communication-avoiding CG is shown in Algorithm 1. It produces a sequence of solution approximation vectors x_i and residual vectors $r_i = b - Ax_i$, using a three-term recurrence which relates x_{i-1}, x_i, x_{i+1}, and r_i (details of this are described in [8]). On each iteration, it applies the matrix powers kernel to the current residual vector r_{ki+1}; the power vectors, along with vectors from the previous iteration, form a basis B from which all vectors produced in this iteration will be a linear combination. The recurrence relation is used to compute a matrix D giving the iteration's output vectors in terms of B columns, with dot products computed using the Gram matrix $G = B^T B$. Finally, the output vectors are made explicit by multiplying B and D. Constructions of the Gram matrix and the final output vectors are done by calling the specializer's BLAS wrappers.

Algorithm 1 CA-CG algorithm outline

1: $x_0 \leftarrow 0$
2: $x_1 \leftarrow$ initial guess
3: $r_0 \leftarrow 0$
4: $r_1 \leftarrow b - Ax_1$
5: **for** $i = 0, 1, \ldots$ **do**
6: Use matrix powers kernel to compute $[Ar_{ki+1}, \ldots, A^k r_{ki+1}]$
7: $B \leftarrow [x_{ki}, x_{ki+1}, r_{ki-i+2}, \ldots, r_{ki+1}, Ar_{ki+1}, \ldots, A^k r_{ki+1}]$
8: $G \leftarrow B^T B$
9: Compute matrix D of output vectors in terms of B
10: $[x_{ki+i}, x_{ki+i+1}, r_{ki+2}, \ldots, r_{ki+i+1}] \leftarrow BD$
11: **end for**

To demonstrate performance portability, the CA-CG solver was tested on three different multi-core machines: an Intel Xeon (Figure 5), another Intel Xeon with a large number of cores (Figure 6), and an AMD Opteron (Figure 7). The five test matrices are from the University of Florida Sparse Matrix Collection [9] and were chosen for being positive definite and so compatible with CG, being reasonably well-conditioned, and having the kind of locality in their structures that makes it possible to avoid communication. A matrix labeled 149K/10.6M has 149 thousand rows and 10.6 million nonzero elements. The solver is generally several times faster than SciPy's serial implementation of conventional CG, the baseline performance a high-level language application writer could obtain without using any additional libraries. When k is allowed to be greater than 1, it often beats MKL's parallel CG implementation as well (geometric mean is 159% faster for matrix powers alone and 35% faster altogether).

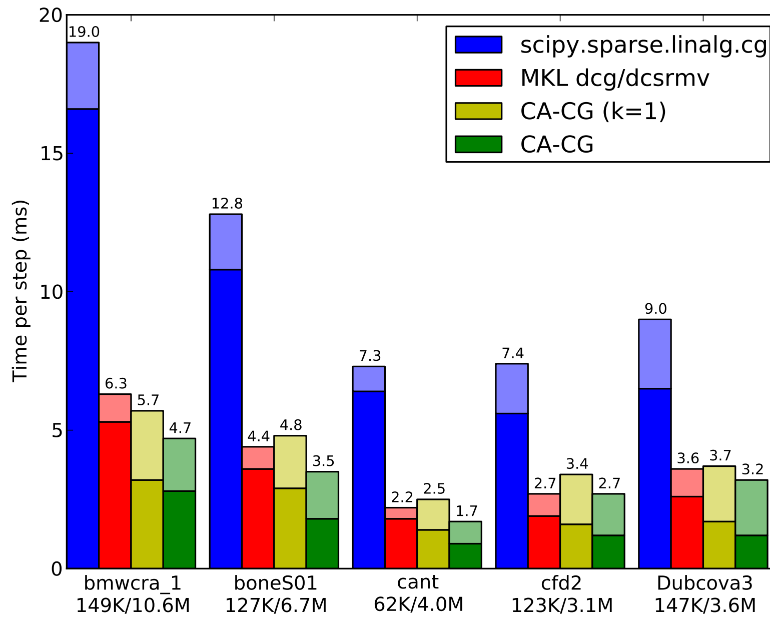

Fig. 5. CG solver performance on 2-socket Intel Xeon X5550 (8 cores, 2.67GHz)

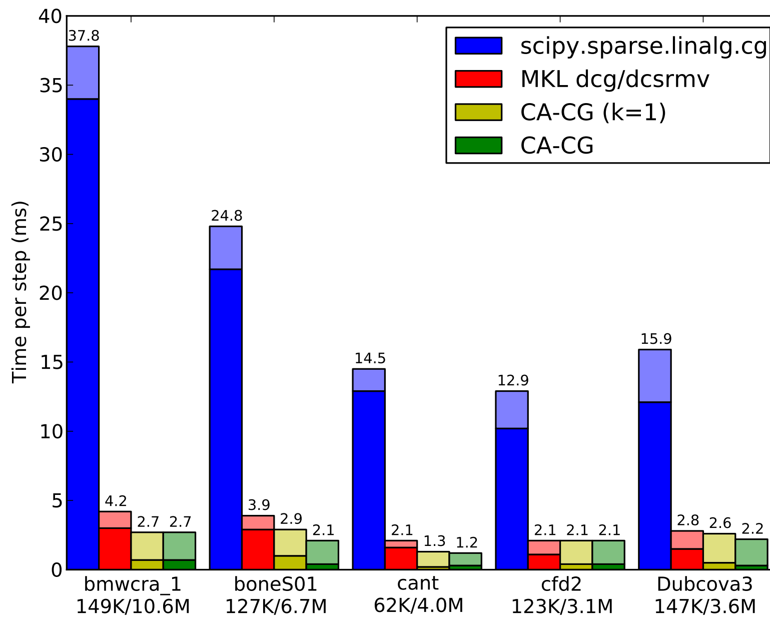

Fig. 6. CG solver performance on 4-socket Intel Xeon X7560 (32 cores, 2.27GHz)

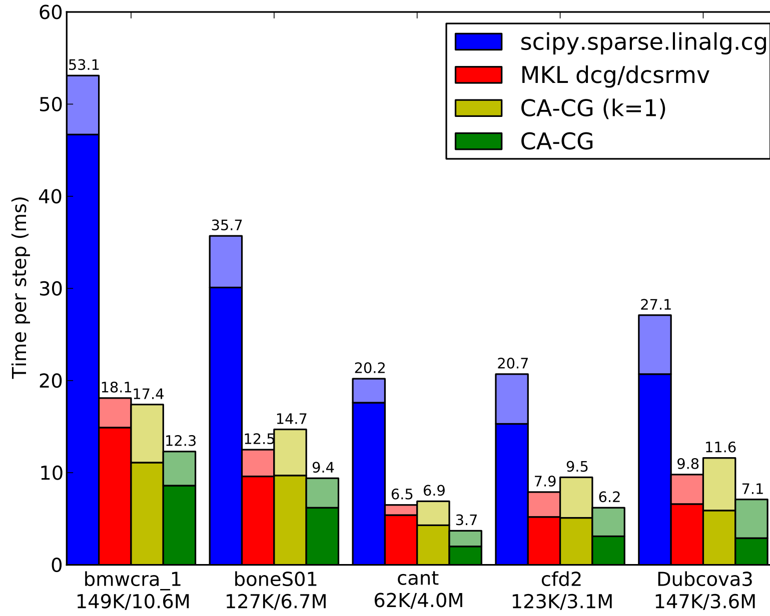

Fig. 7. CG solver performance on 2-socket AMD Opteron 2356 (8 cores, 2.3GHz)

Platform	Solver	Time per step (ms; sparse/dense)				
		`bmwcra_1`	`boneS01`	`cant`	`cfd2`	`Dubcova3`
2-socket Intel Xeon X5550 (8 cores, 2.67GHz)	SciPy	16.6/2.4	10.8/2.0	6.4/0.9	5.6/1.8	6.5/2.5
	MKL	5.3/1.0	3.6/0.8	1.8/0.4	1.9/0.8	2.6/1.0
	CA-CG (k=1)	3.2/2.5	2.9/1.9	1.4/1.1	1.6/1.8	1.7/2.0
	CA-CG (best)	2.8/1.9	1.8/1.7	0.9/0.8	1.2/1.5	1.2/2.0
		k=2	k=2	k=3	k=2	k=2
4-socket Intel Xeon X7560 (32 cores, 2.27GHz)	SciPy	34.0/3.8	21.7/3.1	12.9/1.6	10.2/2.7	12.1/3.8
	MKL	3.0/1.2	2.9/1.0	1.6/0.5	1.1/1.0	1.5/1.3
	CA-CG (k=1)	0.7/2.0	1.0/1.9	0.2/1.1	0.4/1.7	0.5/2.1
	CA-CG (best)	0.7/2.0	0.4/1.7	0.3/0.9	0.4/1.7	0.3/1.9
		k=1	k=2	k=2	k=1	k=2
2-socket AMD Opteron 2356 (8 cores, 2.3GHz)	SciPy	46.7/6.4	30.1/5.6	17.6/2.6	15.3/5.4	20.7/6.4
	MKL	14.9/3.2	9.6/2.9	5.4/1.1	5.2/2.7	6.6/3.2
	CA-CG (k=1)	11.1/6.3	9.7/5.0	4.3/2.6	5.1/4.4	5.9/5.7
	CA-CG (best)	8.6/3.7	6.2/3.2	2.0/1.7	3.1/3.1	2.9/4.2
		k=2	k=2	k=3	k=3	k=2

Table 2. CG solver timing data for Figures 5–7.

For the $k > 1$ case, the time given is the time per iteration divided by k, since one iteration is mathematically equivalent to k iterations of conventional CG. Although CA-CG is more susceptible to accumulating error in the x vectors, for every matrix tested, if CA-CG did converge to a given tolerance then it did so in nearly the expected number of iterations. The dark part of each bar shows time spent on matrix powers while the light part shows time in the remainder of the solver. This does not include time spent in tuning before calling the solver, which is on the order of a few minutes for each matrix and value of k, or typically about 4000–10000 SciPy SpMV calls; however, this cost can be amortized across multiple solves using the same matrix, as tuning need not be repeated. There is also plenty of room for improvement regarding reducing the tuning time, as discussed in section 6.

4 Discussion

From the experience of developing this specializer, several benefits of writing a specializer rather than a traditional library are observable.

One benefit is that the SEJITS framework provides a ready-made templating system for generating code. SEJITS templates are less work to create, and often cleaner, than the ad-hoc code generation scripts typically written in developing auto-tuned libraries. An example of template use is in Figure 8, where normal and unrolled loops integrate nearly seamlessly, in contrast to the more confusing code that would exist to do the same code generation using direct string concatenation.

```
    for (jb = A->browptr[ib]; jb < A->browptr[ib+1]; ++jb) {
%     for i in xrange(b_m):
%       for j in xrange(b_n):
          y[ib*${b_m} + ${i}] += A->bvalues[jb*${b_m*b_n} + ${i*b_n + j}]
                               * x[A->bcolidx[jb]*${b_n} + ${j}];
%       endfor
%     endfor
    }
```

Fig. 8. Template code for computing one row (having index `ib`) of the matrix-vector multiplication $y = Ax$. `b_m` and `b_n` are the tile height and width, respectively.

Another benefit of writing a specializer is that it allows the auto-tuning logic to be written in the high-level language. Not only does this make it easier to write but it also makes it more extensible; if someone wishes to plug in a more advanced auto-tuner, this can be done without having to modify and re-install the specializer.

Finally, being able to generate and compile code at runtime means the combinatorial explosion of all possible code variants does not cause exponential growth

in the size of the specializer. Each combination of parameters for basis, tile size, symmetric representation, implicit cache blocking and index compression requires its own compiled code variant to work efficiently. The set of all possible combinations already numbers in the hundreds, which would make for a large library; adding more features and optimizations could render the library approach unworkable.

5 Related Work

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) [10]. 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 linear algebra) [11], SPIRAL (Fast Fourier Transforms) [12], and stencils [13, 14], 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 the template and tree manipulation systems provided by SEJITS.

The OSKI (Optimized Sparse Kernel Interface) library [11] precompiles 144 variants of each supported operation based on install-time hardware benchmarks 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.

6 Future Work

There are several ways this specializer might be improved or extended. Variations on the matrix powers kernel required by more sophisticated solvers could be added, such as preconditioning, or simultaneous computation of powers of A and A^T as in BiCG. More optimizations could be added based on the extensive existing knowledge of optimizing sparse matrix-vector multiplication. The tuner could be made more advanced, by using a performance model or machine learning, in order to effectively cover a larger search space of possible optimizations without taking excessively long as the current brute-force approach would; note that this would not require changes to the underlying C code.

Currently, the hypergraph partitioning used is the most time-consuming part of the tuning. However, such partitioning is most beneficial when the matrix is highly non-symmetric. One simple optimization would be to use non-hypergraph partitioning for symmetric matrices, such as the matrices used with the CA-CG solver; this could also be extended to other matrices that are not highly non-symmetric. In addition, the search could be implemented using more intelligent

mechanisms such as hill climbing or gradient ascent. Such search strategies would also be amenable to cases when the user wants a tuning decision within a specified time bound, in which case hill climbing or gradient ascent could be used for a few iterations until the maximum bound is reached.

Tuning decisions could potentially be reused for matrices with the same structure; such matrices are commonly used in finite element computations, where the actual values in the matrix may change, but the elements appear in the same locations across modeling problems.

7 Conclusion

Though originally motivated by domains where a library is unsuitable due to the generality of the desired computational kernel, the SEJITS methodology also proves useful for domains where generality comes not from the kernel itself but from the need to tune it for performance. Although the matrix powers kernel could plausibly be written as a library, as a specializer it demonstrates how writing auto-tuners as specializers has benefits for both efficiency-level programmers and for productivity-level programmers who wish to extend the tuning logic.

Acknowledgements

Thanks to Erin Carson and Nicholas Knight for providing the initial version of the code for the matrix powers kernel. Thanks also to Mark Hoemmen, Marghoob Mohiyuddin, and James Demmel for feedback and suggestions on this paper.

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. Mohiyuddin, M., Hoemmen, M., Demmel, J., Yelick, K.: Minimizing communication in sparse matrix solvers. In: Supercomputing 2009, Portland, OR (November 2009)
2. Catanzaro, B., Kamil, S., Lee, Y., Asanović, K., Demmel, J., Keutzer, K., Shalf, J., Yelick, K., Fox, A.: SEJITS: Getting productivity and performance with selective embedded JIT specialization. In: Workshop on Programming Models for Emerging Architectures. PMEA '09, Raleigh, NC (October 2009)
3. Kamil, S.: Asp: A SEJITS implementation for Python. https://github.com/shoaibkamil/asp/wiki
4. Kamil, S., Coetzee, D., Fox, A.: Bringing parallel performance to Python with domain-specific selective embedded just-in-time specialization. In: Proceedings of the 10th Python in Science Conference. SciPy 2011, Austin, TX (2011)

5. Intel: Math Kernel Library. http://software.intel.com/en-us/articles/intel-mkl/
6. Carson, E., Demmel, J., Knight, N.: Hypergraph partitioning for computing matrix powers. http://www.cs.berkeley.edu/~knight/cdk_CSC11_abstract.pdf (October 2010)
7. Catalyürek, Ü.V.: Partitioning Tools for Hypergraph. http://bmi.osu.edu/~umit/software.html
8. Hoemmen, M.: Communication-avoiding Krylov subspace methods. PhD thesis, EECS Department, University of California, Berkeley (Apr 2010)
9. Davis, T., Hu, Y.: The University of Florida sparse matrix collection. http://www.cise.ufl.edu/research/sparse/matrices
10. Bilmes, J., Asanović, K., Chin, C.W., Demmel, J.: Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In: Proceedings of International Conference on Supercomputing, Vienna, Austria (July 1997)
11. Vuduc, R., Demmel, J.W., Yelick, K.A.: OSKI: A library of automatically tuned sparse matrix kernels. Journal of Physics Conference Series **16**(i) (2005) 521–530
12. Püschel, M., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: Code generation for DSP transforms. Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" **93**(2) (2005) 232–275
13. Kamil, S., Chan, C., Oliker, L., Shalf, J., Williams, S.: An auto-tuning framework for parallel multicore stencil computations. In: IPDPS'10. (2010) 1–12
14. Tang, Y., Chowdhury, R.A., Kuszmaul, B.C., Luk, C.K., Leiserson, C.E.: The Pochoir stencil compiler. In: Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures. SPAA '11, New York, NY, USA, ACM (2011) 117–128