OSKI's kernels can be tuned according to matrix/vector structure. The new Epetra/OSKI interface enables Trilinos and application developers to leverage the highly tuned kernels provided by OSKI in a standardized manner.

In this paper, we discuss our implementation of an interface to OSKI within Epetra and assess its performance. In Section 2, we give an overview of the design and features of the OSKI package itself. In Section 3, we discuss the design of the Epetra interface to OSKI. In Section 4, we discuss the results of performance tests run on the OSKI kernels within Epetra. Tests were run on individual OSKI kernels, and include small scaling studies. In Section 5, conclusions of the work and results described in this paper are presented. In Section 6, ways to add more functionality to our implementation, and suggestions of things to test in new OSKI releases are presented.

2. OSKI High Level Overview. OSKI is a package used to perform optimized sparse matrix-vector operations. It provides both a statically tuned library created upon installation and dynamically tuned routines created at runtime. OSKI provides support for single and double precision values of both real and complex types, along with indexing using both integer and long types. When possible it follows the sparse BLAS standard [5] as closely as possible in defining operations and functions.

Before a matrix can use OSKI functionality, it first must be converted to the matrix type `oski_matrix_t`. To store a matrix as an `oski_matrix_t` object, a create function must be called on a CSR or CSC matrix. An `oski_matrix_t` object can either be created using a deep or shallow copy of the matrix. When a shallow copy is created, the user must only make changes to the matrix's structure through the OSKI interface. When a deep copy is created, the matrix that was passed in can be edited by the user as desired. OSKI automatically makes a deep copy when any matrix is tuned in a manner that changes its structure.

Routine	Calculation
Matrix-Vector Multiply	$y = \alpha Ax + \beta y$ or $y = \alpha A^T x + \beta y$
Triangular Solve	$x = \alpha A^{-1} x$ or $x = \alpha A^{T^{-1}} x$
Matrix Transpose Matrix-Vector Multiply	$y = \alpha A^T Ax + \beta y$ or $y = \alpha AA^T x + \beta y$
Matrix Power Vector Multiply	$y = \alpha A^p x + \beta y$ or $y = \alpha A^{T^p} x + \beta y$
Matrix-Vector Multiply and Matrix Transpose Vector Multiply	$y = \alpha Ax + \beta y$ and $z = \omega Aw + \zeta z$ or $z = \omega A^T w + \zeta z$

TABLE 2.1
Computational kernels from OSKI available in Epetra.

OSKI provides five matrix-vector operations to the user. The operations are shown in Table 2.1. Hermitian operations are available in OSKI, but are not shown in the table since Epetra does not include Hermitian functionality. The last three kernels are composed operations using loop fusion [6] to increase data reuse. To further improve performance, OSKI can link to a highly tuned BLAS library.

OSKI creates optimized routines for the target machine's hardware based on empirical search, in the same manner as ATLAS [15] and PHiPAC [3]. The goal of

the search is create efficient static kernels to perform the operations listed in Table 2.1. The static kernels then become the defaults that are called by OSKI when runtime tuning is not used. Static tuning can create efficient kernels for a given data structure. To use the most efficient kernel, the matrix data structure may need to be reorganized.

When an operation is called enough times to amortize the cost of rearranging the data structure, runtime tuning can be more profitable than using statically tuned functions. OSKI provides multiple ways to invoke runtime tuning, along with multiple levels of tuning. A user can explicitly ask for a matrix to always be tuned for a specific kernel by selecting either the moderate or aggressive tuning option. If the user wishes for OSKI to decide whether enough calls to a function occur to justify tuning, hints can be used. Possible hints include telling OSKI the number of calls expected to the routine and information about the matrix, such as block structure or symmetry. In either case, OSKI tunes the matrix either according to the user's requested tuning level, or whether it expects to be able to amortize the cost of tuning if hints are provided. Instead of providing hints the user may, periodically call the tune function. In this case, the tune function predicts the number of future kernel calls based on past history, and tunes the routine only if it expects the tuning cost to be recovered via future routine calls.

OSKI can also save tuning transformations for later reuse. Thus, the cost of tuning searches can be amortized over future runs. Specifically, a search for the best tuning options does not need to be run again, and only the prescribed transformations need to be applied.

OSKI is under active development. As of this writing, the current version is 1.0.1h, with a multi-core version under development [12]. While OSKI provides many optimized sparse matrix kernels, some features have yet to be implemented, and certain optimizations are missing. OSKI is lacking multi-vector kernels and stock versions of the composed kernels. These would greatly add to both OSKI's usability and performance. The Matrix Power Vector Multiply is not functional. Finally, OSKI cannot transform (nearly) symmetric matrices to reduce storage or convert from a CSR to a CSC matrix (or vice versa). Both could provide significant memory savings. Thus, performance gains from runtime tuning should not be expected for point matrices. An exception is pseudo-random matrices, which may benefit from cache blocking.

3. Design and Implementation. In the design and implementation of the Epetra OSKI interface the Epetra coding guidelines [8] were followed as closely as possible. In doing so, we ensured the consistency of our code with the existing Epetra code base, as well as its readability and maintainability. Finally, the Epetra interface to OSKI will likely be ported to Kokkos [10], and the interface's design will make this process easier.

In the design phase we focused on allowing the greatest amount of flexibility to the user, and exposing as much of the functionality of OSKI as possible. In some places, however, OSKI functionality is not exposed because there is not a corresponding Epetra function. For example, OSKI has a function that allows changing a single value in a matrix, but Epetra does not. When two copies of a matrix exist, as when the OSKI constructor makes a deep copy of the underlying data, the corresponding Epetra copy is guaranteed to contain the same data. Since Epetra can only change data values one row at a time, a point set function is not included in the OSKI interface. Instead, we include a function to change a row of data within OSKI by overloading the Epetra function to change row data. When a single copy of the data

exists, the Epetra function is called on the matrix. When both an OSKI and Epetra matrix exist, both the matrix copies are modified to keep the data consistent. The Epetra function is called once for the Epetra version of the matrix, and the OSKI matrix has its point function called once for each entry in the row.

When there are clear equivalent functions in OSKI and Epetra, the OSKI function is designed to overload the Epetra function. In the cases where OSKI provides more functionality than Epetra, the interface is designed with two functions to perform the operation. The first function mimics Epetra's functionality and passes values that eliminate the extra functionality from OSKI. The second function exposes the full functionality OSKI provides. Also, as appropriate new functions are added that are specific to OSKI, such as the tuning functions. Conversely, Epetra functions without any analogue in the OSKI context are not overloaded in the `Epetra_Oski` namespace.

The interface is also designed to maintain robustness and ease of use. All `Epetra_OskiMatrix` functions that take in vectors or multi-vectors allow for the input of both `Epetra_Vector` or `Epetra_MultiVector` objects, and `Epetra_OskiVector` or `Epetra_OskiMultiVector` objects. The objects are converted to the proper types as necessary through the use of the lightest weight wrapper or converter possible.

The implementation follows the idea of wrapping and converting data structures in as lightweight a fashion as possible, to maximize speed and minimize space used. In addition, the implementation provides the user with as much flexibility as possible. For example, the user can specify as many tuning hints as they like. Alternatively, the user can ask Epetra to figure out as much as it can about the matrix and pass along those hints to OSKI. Both options can be combined, with user-specified hints taking precedence over automatically generated hints. Options are passed by the user via Teuchos parameter lists [11].

Class	Function
`Epetra_OskiMatrix`	Derived from `Epetra_CrsMatrix`. Provides all OSKI matrix operations.
`Epetra_OskiMultiVector`	Derived from `Epetra_MultiVector`. Provides all OSKI multi-vector operations.
`Epetra_OskiVector`	Derived from `Epetra_OskiMultiVector`. Provides all OSKI vector operations.
`Epetra_OskiPermutation`	Stores permutations and provides Permutation functions not performed on a `Epetra_OskiMatrix`.
`Epetra_OskiError`	Provides access to OSKI error handling functions and the ability to change the default OSKI error handler.
`Epetra_OskiUtils`	Provides the initialize and finalize routines for OSKI.

TABLE 3.1
OSKI classes within Epetra.

Finally, the design is broken into six separate classes. Table 3.1 shows the classes and provides information about which classes each derives from, and what functions each contains. The design is as modular as possible to allow for the easy addition of new functions, and to logically group related functions together.

4. Results. To assess the potential benefit of using OSKI in Sandia applications, we ran tests on representative data and a variety of advanced architectures. For these

tests OSKI version 1.0.1h was used. OSKI runtimes were compared to the runtimes of the currently used Epetra algorithms, in both serial and parallel. In this section, we first present our test environment and methodology, and then present the results of performance tests run comparing Epetra to OSKI.

4.1. Test Environment and Methodology. Performance tests were run on two different machine architectures in serial and parallel. The first test machine has two Intel Clovertown processors. The second test machine has one Sun Niagara-2 processor. Machine specifications and compilers are shown in Table 4.1. On each machine, Trilinos was compiled with widely used optimizations levels, and OSKI was allowed to pick the best optimization flags itself.

processor	#chips	cores	threads	frequency	L2 cache	compiler
Clovertown	2	8	8	1.87 Ghz	4 M per 2 cores	Intel
Niagara-2	1	8	64	1.4 Ghz	4 M per core	Sun

TABLE 4.1
Test machines used for performance testing.

These machines were chosen for their diversity and potential for use at Sandia. The Clovertown is one of Intel's latest processors, and the Niagara is an example of an extremely parallel chip.

On each machine, tests were run on three matrices arising from Sandia applications. The first matrix is from a finite element discretization within a magnetics simulation. The second is a block-structured Poisson matrix. The third matrix is unstructured and represents term-document connectivity. The data is from the Citeseer application. Table 4.2 gives some matrix properties. Each matrix was able to fit within the main memory of each test machine. These matrices were also used in a

matrix	rows	columns	nnz	structure
point	556356	556356	17185984	nearly symmetric point
block	174246	174246	13300445	symmetric 3 by 3 blocks
Citeseer	607159	716770	57260599	unstructured point

TABLE 4.2
Test machines for Epetra OSKI performance testing.

scaling study. Tests were run up to the total number of available threads that can be executed simultaneously, on each machine.

4.2. Performance Test Results. The serial results for each machine are shown in Figures 4.1 and 4.2 for four OSKI kernels: Ax, A^Tx, A^TAx, and the two-vector multiplication $y = Ax$; $z = Aw$. The last operation is henceforth referred to as "2Mult". In addition, Table 4.3 shows the speeds of Epetra calculations as a baseline. Since OSKI has no atomic versions of the composed kernels, the OSKI stock numbers represent two separate matrix-vector multiply calls to OSKI. There is potential that the tuned composed kernels are not performing optimally due to tuning to a non-ideal data structure, as is seen in the tuning cost data later. Results for the matrix power kernel are unavailable due to a bug in the kernel. Also results for the AA^T kernel were excluded because Epetra only stores matrices in CSR. OSKI cannot convert CSR to CSC, which is needed to take advantage of these kernels in serial. Finally, the direct solve kernel was not profiled, as it is not critical to many Sandia applications.

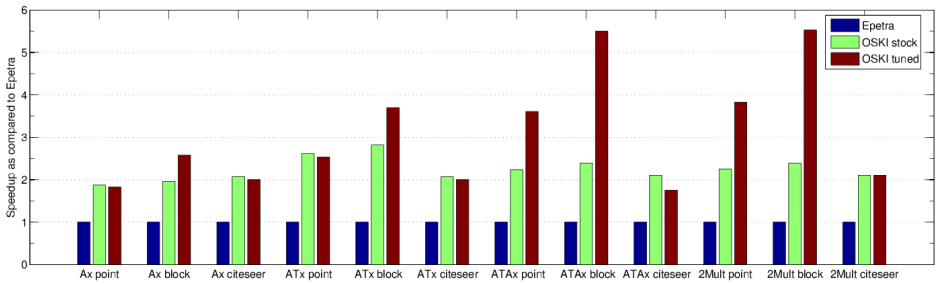


FIG. 4.1. *Relative performance of Epetra and OSKI in serial on Clovertown.*

Machine	Ax	A^Tx	A^TA	2Mult
Clovertown	220/227/55	150/154/43	178/183/48	178/184/48
Niagara	58.3/69.9/20.7	56/66.4/20.3	57.1/68.1/20.5	57.1/68.1/20.5

TABLE 4.3
Epetra serial routine speeds in Mflops. Results are in the form point/block/Citeseer.

On the Clovertown, OSKI produced large speedups over Epetra for all matrices in serial, as shown in Figure 4.1. The stock kernels demonstrated speedups of 1.8 to 2.8. Tuning improved the block matrices by about one third when compared to the stock kernels. The composed algorithms demonstrated even more significant speedups of up to 5.5, when composing and blocking were combined. Tuning did not improve the runtime of point matrices, except when a composed kernel was used. In the case of the Citeseer matrix, a composed kernel resulted in either no performance gain or performance degradation.

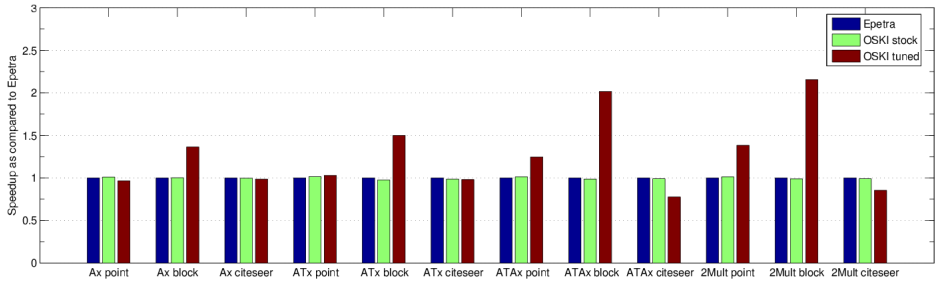


FIG. 4.2. *Relative performance of Epetra and OSKI in serial on Niagara.*

Figure 4.2 shows that on the Niagara, the stock OSKI and Epetra kernels had roughly the same performance Tuning for point matrices once again resulted in either no gains or slight losses. Tuning for block matrices resulted in a one third to one half gain in speed. Again, composing increased the speed of all kernels significantly, except for the Citeseer matrix, for which the OSKI kernels where actually slower.

As expected, the serial tests show that the tuning of point matrices is counter-productive, except when needed to use composed kernels. However, tuning of block matrices results in significant speedups through the reduction of indirect addressing. For the pseudo random Citeseer matrix, tuning is never beneficial. This is probably due to either lack of cache-blocking in the composed kernels and/or more random

access, which create a greater number of cache misses. For structured matrices, composing results in a 25% to 60% gain over the faster of the stock and tuned kernels.

Even if the tuning gains shown above are large, the amount of time it takes to tune a matrix at runtime is important in determining whether tuning will result in performance gains. Tables 4.4, 4.5 and 4.6 show the cost of tuning and the number of matrix-vector calls needed to amortize that cost for the point, block, and Citeseer matrices, respectively. The tuning and retuning costs are expressed in terms of the number of matrix-vector multiplies that could be performed in the time it takes to tune. *Tuning cost* is the amount of time it takes to tune a matrix the first time, and includes time to analyze the matrix to determine what optimizations are beneficial. *Retuning cost* is the amount of time it takes to tune the matrix if the optimizations to be performed are already known. All comparisons are to the faster of the Epetra and OSKI matrix-vector multiplies. The amortize columns show the number of calls to the tuned kernel needed to realize tuning gains. When N/A is listed in an amortize column, it is never better to tune because the tuned kernels are no faster than the untuned kernels. We note that the tuning cost depends only on the matrix structure, not on the matrix kernel to be performed.

Machine	Tune/Retune	Amortize Ax/Retune	Amortize A^TA/Retune	Amortize 2Mult/Retune
Clovertown	37.6 / 20.1	N/A	48 / 26	45 / 24
Niagara	22.1 / 12.7	N/A	56 / 33	40 / 24

TABLE 4.4
OSKI tuning costs for point matrix. Cost is equivalent number of matrix-vector multiplications.

Machine	Tune/Retune	Amortize Ax/Retune	Amortize A^TA/Retune	Amortize 2Mult/Retune
Clovertown	31.1 / 17.7	131 / 75	27 / 16	28 / 16
Niagara	22.5 / 14.1	86 / 54	22 / 14	21 / 13

TABLE 4.5
OSKI tuning costs for block matrix. Cost is equivalent number of matrix-vector multiplications.

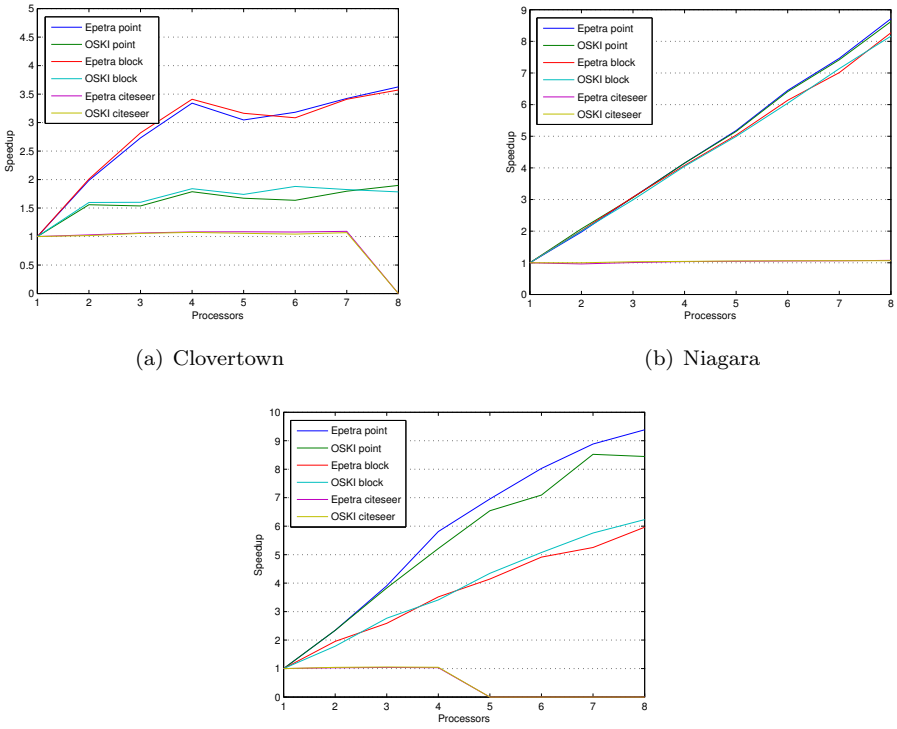
Machine	Tune/Retune	Amortize Ax/Retune	Amortize A^TA/Retune	Amortize 2Mult/Retune
Clovertown	14.5 / 6.7	N/A	N/A	N/A
Niagara	11.5 / 5.2	N/A	N/A	N/A

TABLE 4.6
OSKI tuning costs for Citeseer matrix. Cost is equivalent number of matrix-vector multiplications.

In many cases, the tuned OSKI kernels are much more efficient than the Epetra and OSKI stock kernels. However, the data structure rearrangement required to create an OSKI kernel is non-trivial. The cost of tunings ranges from 11.5 to 37.6 equivalent matrix-vector multiplies. It can require as many as 131 subsequent kernel applications to recoup the cost of initial tuning. However, re-tuning costs are usually slightly over half the cost of the initial tuning, so saving transformations for later use could be profitable. Block matrices require the smallest number of calls to recover tuning costs, and when combined with composed kernels, this number drops even

more. For point matrices tuning the matrix-vector multiply is never profitable, but the tuning of composed kernels can be profitable for structured matrices.

While serial performance is important to application performance, most scientific simulations are run on parallel machines. The first level of parallelism is within a single node, which typically contains one or two multicore processors. To test the scalability of our implementation of OSKI, within Epetra, we ran tests on each matrix on 1 to 8 cores of each machine and also on 1 to 8 threads per core on the Niagara.



(a) Clovertown

(b) Niagara

(c) Niagara multi-threaded

FIG. 4.3. *OSKI matrix-vector multiply strong scaling results.*

Figures 4.3(a)-4.3(c) show the strong scaling of the matrix-vector kernel for each matrix. Figure 4.3(a) shows that on the Clovertown that Epetra has better scaling than OSKI. Table 4.7 shows, however, that the overall performance of OSKI is either comparable or better to that of Epetra. The better scaling for Epetra comes from its slower performance in the single processor case, which allows for more improvement within a limited memory bandwidth situation. For the point matrix, both Epetra and OSKI improve significantly until each is running at about 735 Mflops on 4 cores. At this point, the calculations likely become memory bandwidth limited. With added processing power, the speeds then improve to slightly under 800 Mflops. The block matrix results show a similar pattern, with the OSKI block matrix remaining more efficient throughout. The Citeseer matrix does not scale most likely due to the large amounts of data it needs to exchange, because its unstructured. Also it could not be run on 8 processors due to an increasing memory footprint, perhaps due to exchanged data.

machine	point Epetra/OSKI	block Epetra/OSKI	Citeseer Epetra/OSKI
Clovertown	798/782	810/1099	59.6/122
Niagara 1 thread/core	508/507	578/778	22.3/22.0
Niagara multiple threads/core	4767/4321	3447/4847	23.2/23.2

TABLE 4.7
Epetra and OSKI maximum parallel matrix vector multiply speeds in Mflops.

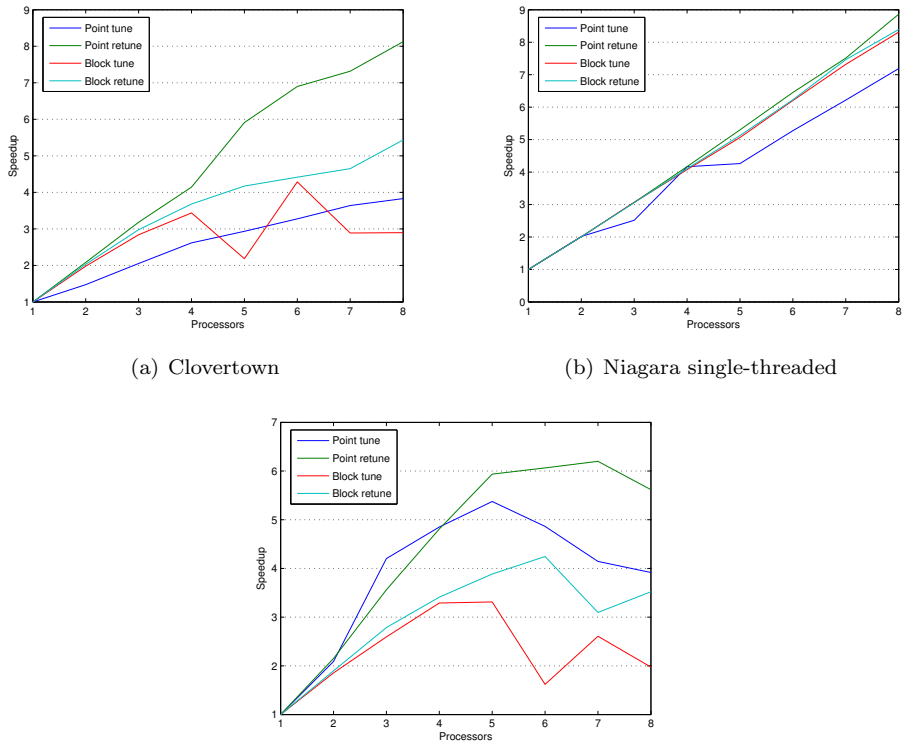
Figure 4.3(b) shows that on the Niagara both the point and block matrix algorithms scale linearly with the number of cores. Essentially, there is enough memory bandwidth to feed each core. As seen in Figure 4.3(c), adding more threads per core to the calculating power leads to approximately linear speedup for all matrices. This begins to tail off at 5 threads for block matrices, and 7 threads for point matrices. The Citeseer matrix once again does not scale and becomes too large to run above 32 threads.

Scalability also matters when a matrix is being tuned. Figures 4.4(a)-4.4(c) show how well each matrix scales on each machine in terms of tuning cost. Scaling is usually linear or slightly better with the number of processors. This result is expected as tuning is a local computation with no communication between processors. As seen in Figure 4.4(c), increasing the number of threads per Niagara processor initially leads to improved performance, before dropping off at 6 or more threads per processor. The dropoff is most likely due to threads competing for processor resources. Results for the Citeseer matrix were not shown, as OSKI does not tune its matrix-vector multiply kernel for the Citeseer matrix. Finally, note that the retune function demonstrates better scaling than the same tune function in all cases.

In addition to strong scaling tests, we also ran a weak scaling test on the Niagara. We used the block matrix from the 8 thread test case in Table 4.2. Tests were run on 1, 8, 27 and 64 threads. Results are shown in Figures 4.5(a)-4.5(c). As seen in Figure 4.5(a), the OSKI tuned and untuned matrix-vector multiplies both scale similarly to Epetra's matrix-vector multiply. Figure 4.5(b), shows that the tuned composed kernels do not scale well. The same result was seen for the untuned composed kernels. For these operations to be possible there is extra data copying in the wrapping of the serial kernels, which could be the problem. There could also be inefficiencies in the code in other places or resource contention on the processor. Figure 4.5(c) shows that re-tuning scales better than tuning as the problem size grows.

5. Conclusions. Overall, OSKI can produce large speedups in sparse matrix computational kernels. This is especially true when the matrix is block structured or multiple multiplications are performed using the same matrix. In some cases it can also produce large gains for matrix-vector multiplies involving only a single matrix. However, OSKI is still missing some features, such as a multi-vector kernel and the ability to tune matrices to make them symmetric. Both could produce large runtime gains. Our Epetra/OSKI interface has stubs to allow the use of these missing features as soon as they become available in OSKI. Our experiments show that Sandia applications that make heavy use certain sparse matrix kernels can benefit from the current version of OSKI. As new OSKI features become available, its potential impact on other Sandia applications should increase.

6. Future Work. For the current (1.0.1h) version of OSKI, a developer may want to implement the solve function and run more weak scalability or other parallel



(a) Clovertown

(b) Niagara single-threaded

(c) Niagara multi-threaded

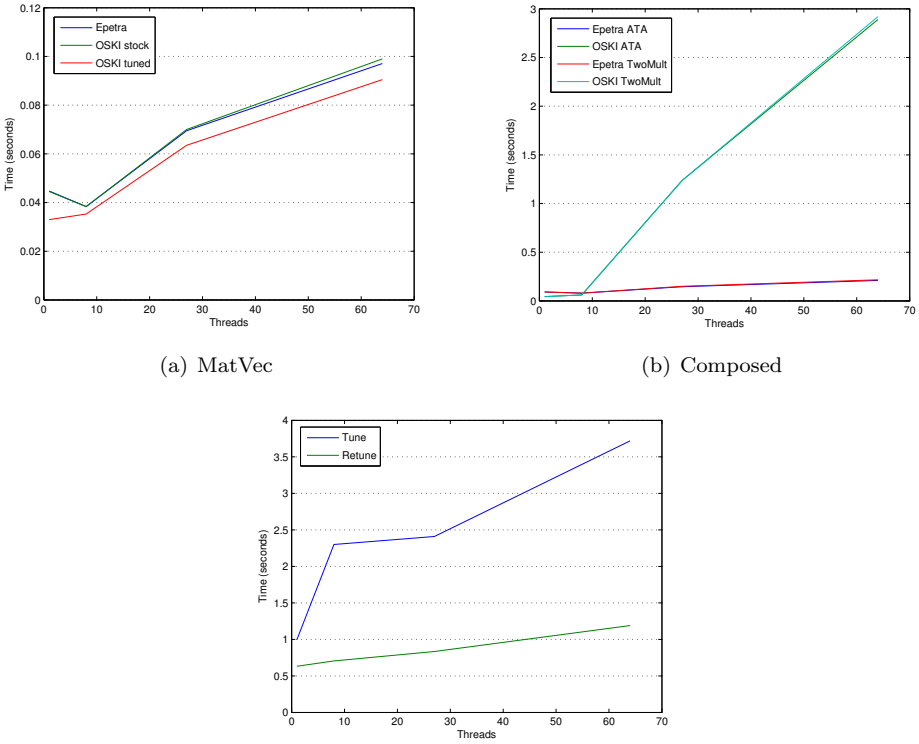
FIG. 4.4. *Scalability of OSKI tuning.*

tests to determine why the composed kernels do not scale well. For a newer version of OSKI, a developer may want to test any new tuning features, the matrix power kernel, as well as any other new functions. Finally, we recommend any new version of OSKI be tested on the Barcelona and Xeon chips, as we were never able to successfully install OSKI on these architectures. The Barcelona is of particular interest, as it is the processor found in the center section of Red Storm.

7. Acknowledgments. We would like to thank Brian Barrett and Doug Doerfler for access to the Niagara and Clovertown architectures, respectively. We also would like to thank Danny Dunlavy and Chris Siefert for providing us with the test matrices. In addition, we would like to thank Mike Heroux, Chris Siefert and Jim Willenbring for reviewing the interface design and answering questions along the way. Jim's partial OSKI implimentation of an interface within Kokkos helped serve as a model for our development. Finally, we would also like to thank Rich Vuduc for his help with Oski-related questions.

REFERENCES

[1] E. ANGERSON, Z. BAI, J. DONGARRA, A. GREENBAUM, A. MCKENNEY, J. DU CROZ, S. HAMMARLING, J. DEMMEL, C. BISCHOF, AND D. SORENSEN, *Lapack: A portable linear algebra library for high-performance computers*, Nov 1990, pp. 2–11.



(a) MatVec

(b) Composed

(c) Tuning

FIG. 4.5. *Weak scalability of OSKI on Niagara*

[2] BERKELEY BENCHMARKING AND OPTIMIZATION GROUP, *OSKI: Optimized Sparse Kernel Interface*. http://bebop.cs.berkeley.edu/oski/about.html, May 2008.

[3] J. BILMES, K. ASANOVIC, C.-W. CHIN, AND J. DEMMEL, *Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology*, in ICS '97: Proceedings of the 11th international conference on Supercomputing, New York, NY, USA, 1997, ACM, pp. 340–347.

[4] 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 Subprograms (BLAS)*, ACM Transactions on Mathematical Software, 28 (2002), pp. 135–151.

[5] I. S. DUFF, M. A. HEROUX, AND R. POZO, *An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum*, ACM Transactions on Mathematical Software (TOMS), 28 (2002).

[6] G. R. GAO, R. OLSEN, V. SARKAR, AND R. THEKKATH, *Collective loop fusion for array contraction*, in 1992 Workshop on Languages and Compilers for Parallel Computing, no. 757, New Haven, Conn., 1992, Berlin: Springer Verlag, pp. 281–295.

[7] M. A. HEROUX, R. A. BARTLETT, V. E. HOWLE, R. J. HOEKSTRA, J. J. HU, T. G. KOLDA, R. B. LEHOUCQ, K. R. LONG, R. P. PAWLOWSKI, E. T. PHIPPS, A. G. SALINGER, H. K. THORNQUIST, R. S. TUMINARO, J. M. WILLENBRING, A. WILLIAMS, AND K. S. STANLEY, *An overview of the Trilinos project*, ACM Trans. Math. Softw., 31 (2005), pp. 397–423.

[8] M. A. HEROUX AND P. M. SEXTON, *Epetra developers coding guidelines*, Tech. Rep. SAND2003-4169, Sandia National Laboratories, Albuquerque, NM, December 2003.

[9] SANDIA NATIONAL LABORATORIES, *Epetra - Home*. http://trilinos.sandia.gov/packages/epetra/index.html, May 2008.

[10] ———, *Kokkos - Home*. http://trilinos.sandia.gov/packages/kokkos/index.html, May 2008.

[11] ———, *Teuchos - Home*. http://trilinos.sandia.gov/packages/teuchos, May 2008.

[12] R. Vuduc, *Personal Communication*, July 2008.
[13] R. Vuduc, J. W. Demmel, and K. A. Yelick, *Oski: A library of automatically tuned sparse matrix kernels*, Journal of Physics Conference Series, 16 (2005), pp. 521–530.
[14] ———, *The Optimized Sparse Kernel Interface (OSKI) library user's guide for version 1.0.1h*, tech. rep., University of California at Berkeley, Berkeley, CA, June 2007.
[15] R. C. Whaley and J. J. Dongarra, *Automatically tuned linear algebra software*, Procceding of the 1998 ACM/IEEE conference on Supercomputing (CDROM), (1998), pp. 1–27.