Improving Sparse Matrix-vector Product Kernel Performance and Accessibility

James M. Willenbring and Andrew A. Anda
Computer Science Department
St. Cloud State University
St. Cloud, MN 56301
wija0304@stcloudstate.edu
aanda@stcloudstate.edu

Michael A. Heroux
Computational Mathematics and Algorithms
Sandia National Laboratories
Albuquerque, NM 87185
maherou@sandia.gov

Abstract

Tuning dense Linear Algebra topics has been a topic of intense research. Less research has been dedicated to sparse kernel tuning. Many of the issues associated with tuning sparse kernels are different than dense kernel tuning issues.

The Optimized Sparse Kernel Interface (OSKI) from the Berkeley Benchmarking and Optimization Group (BeBOP) is a recently released software package providing automatically tuned sparse computational kernels. The Trilinos Project, developed primarily at Sandia National Laboratories, targets the development of robust numerical algorithms. Trilinos utilizes existing libraries for improving performance including the various implementations of the BLAS, but does not currently have access to any automatically tuned sparse numerical kernels.

Our current efforts focus on making OSKI functionality available to Trilinos via a templated Trilinos package called Kokkos. We describe an interface to an optimized sparse matrix-vector product kernel which makes the kernel accessible to the many scientific applications that rely on Trilinos for solver capabilities.

1 Background

For a wide array of scientific problems, the cost of computing a solution is dominated by a very small number of lines of code, termed *kernels*. These computationally expensive kernels often perform a matrix-vector or matrix-matrix operation, such as a matrix-vector multiply. It is valuable to distinguish between operations involving dense matrices and operations involving sparse matrices. A *dense* matrix has enough nonzero values that it makes sense to store and operate on every entry in the matrix. Conversely, a *sparse* matrix contains mostly zero values, enough that it is worthwhile to operate on and store only the nonzero values. In general, it is not possible to determine if a matrix should be handled as sparse or dense. The answer may depend on many factors including the operation to be performed, and the machine on which the operation will take place.

Tuning dense linear algebra kernels has been a topic of intense research. Tuning a kernel refers to modifying the kernel in some way to improve performance. The modifications could be based on the properties of the matrix, platform, or both. The Basic Linear Algebra Subprograms (BLAS) [8] [2] are a set of Fortran routines that perform many basic vector and dense matrix operations. Several vendors supply machine specific BLAS implementations. These hand tuned libraries exploit specific architectural characteristics to improve performance. There are also many projects that provide BLAS implementations that are or can be automatically tuned for specific architectures, e.g. ATLAS [10] and PHiPAC [1]. When installing a self-tuning BLAS implementation such as ATLAS, a suite of benchmarks are run to determine the highest performance way to run certain kernels. Instruction cache size, level one data cache size, and the number of registers, for example, can dictate how many times to unroll a loop within a particular routine. Performance will be reduced if the loop is unrolled too few times because there will not be enough instructions available to keep the pipeline full. Unrolling the loop too many times will cause unnecessary cache misses, which lead to significant performance degradation.

Less research has been dedicated to sparse kernel tuning than has been dedicated to dense kernel tuning. In addition, the properties of a sparse matrix make it generally more difficult to write high performance kernels for sparse matrix operations than dense matrix operations. We will examine this issue further in Section 1.3. First we will discuss an automatically tuned sparse kernel library called OSKI [9] as well as a software project that targets the development of a wide array of robust numerical algorithms called Trilinos [4].

1.1 OSKI

The Optimized Sparse Kernel Interface (OSKI) from the Berkeley Benchmarking and Optimization Group (BeBOP) is a recently released software package providing automatically tuned sparse computational kernels. OSKI has a BLAS-style interface, meaning that individual routines are called based on a number of arguments that are passed including pointers to the matrix and vector data, and dimensions of the matrices. OSKI also provides a parameter list input (a way to specify a variable number of inputs) that can be used to indicate the properties of a given problem that OSKI can exploit during the tuning process. The more information a user can provide, the less work it is for OSKI to automatically tune because it narrows down the number of tuning techniques that have to be tried.

1.2 Trilinos

As mentioned above, the Trilinos Project targets the development of robust numerical algorithms. Roughly thirty packages comprise Trilinos. Each package is developed by a small, largely autonomous development team and contains implementations of a number of related algorithms, a basic set of container classes, such as those for matrices or vectors, or generally useful utility classes.
Trilinos utilizes existing libraries for improving performance including the various implementations of the BLAS, but does not currently have access to any automatically tuned sparse numerical kernels. This project has focused on making OSKI functionality available to Trilinos via a *templated*[1] Trilinos package called Kokkos [7]. Kokkos defines interfaces for software developers to implement for the purpose of providing tuned routines. Kokkos also provides default implementations of these interfaces and customized container classes for sparse linear algebra libraries and applications.

1.3 Sparse Matrix-Vector Multiplication

It is much more difficult to obtain high performance when performing sparse matrix-vector multiplication than when performing dense matrix-vector multiplication. Dense matrix-vector multiplication can achieve good temporal locality and excellent spacial locality, which makes it relatively easy to write high performance dense matrix-vector multiplication kernels.
Temporal locality is a measure of the degree to which values are repeatedly referenced in a short period. Referencing a value twice within a short period of time increases the probability that the value is still near the ALU (in a register or cache). After a value is retrieved from memory and while it is still close to the processor, subsequent references to that value are very inexpensive. A problem has good spacial locality if referencing a value in memory implies that nearby values are likely to be referenced in the near future.
The matrix in a row-wise dense matrix-vector multiplication operation exhibits good spacial locality because the values are accessed across the rows. The vector exhibits good spacial and temporal locality because the vector is used repeatedly and is accessed in order. For relatively large problems, blocking can be employed to achieve temporal locality on the vector. Blocking refers to the idea of breaking the matrix into rectangular blocks and the vector into segments corresponding to the number of columns in each block of the matrix and then treating each block as its own independent matrix-vector multiplication problem. This is necessary when the problem is big enough that the beginning of the vector is no longer in cache when multiplication by the next row of the matrix begins. Blocking is also very effective for matrix-multivector multiplication and matrix-matrix multiplication, but those topics are beyond the scope of this paper.
Achieving high performance for sparse matrix-vector multiplication is challenging. When most of the values in the matrix are zero, the temporal and spacial locality on the vector are easily lost because of the irregular data access patterns. In addition, more data per nonzero in the matrix, specifically the value and the index of the value, needs to be stored

[1] Templates can be used in C++ to specify generic data types.

which effectively reduces the size of the cache and increases the number of compulsory load operations per floating-point operation.

We will consider two types of sparse matrices for this project, those generated from the discretization of partial differential equations (PDEs), which are the most commonly encountered type of sparse matrix, and those generated from circuit problems. We will simply refer to these matrices as PDE matrices and circuit matrices.

The structure of PDE matrices tends to contain small dense blocks (or the matrices can be permuted to extract such blocks). Dense operations can then be performed on these dense blocks to improve performance. Circuit matrices are more sparse than PDE matrices and do not typically exhibit this block structure. An example of a PDE matrix can be seen in Figure 1. The black portions of the matrix denote the locations of nonzero values. This matrix is a finite element model of a heart, and is 3,557 by 3,557 with approximately 390 nonzero values per row. An example of a circuit matrix can be seen in Figure 2. This circuit matrix is 430 x 430 with only about 3.6 nonzero values per row. Both matrices were obtained from the University of Florida Sparse Matrix Collection [3], which has been compiled by Tim Davis. Numerous people have submitted matrices to the collection. The names of the contributors of the matrices in the collection are available on the website.

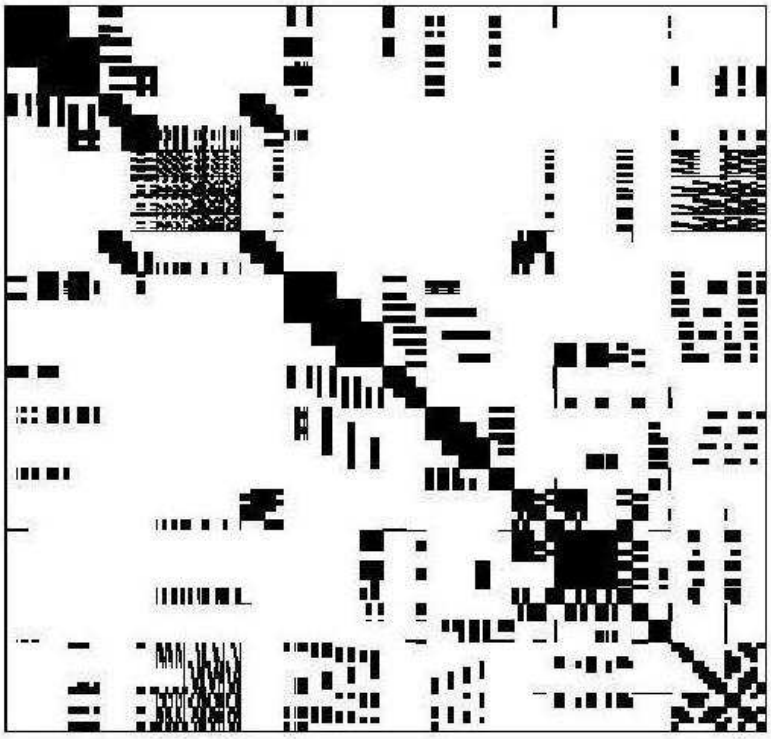

Figure 1: An Example of a PDE Matrix

Looking at the two figures it is easy to see the block structure of the PDE matrix and the lack of block structure in the circuit matrix.

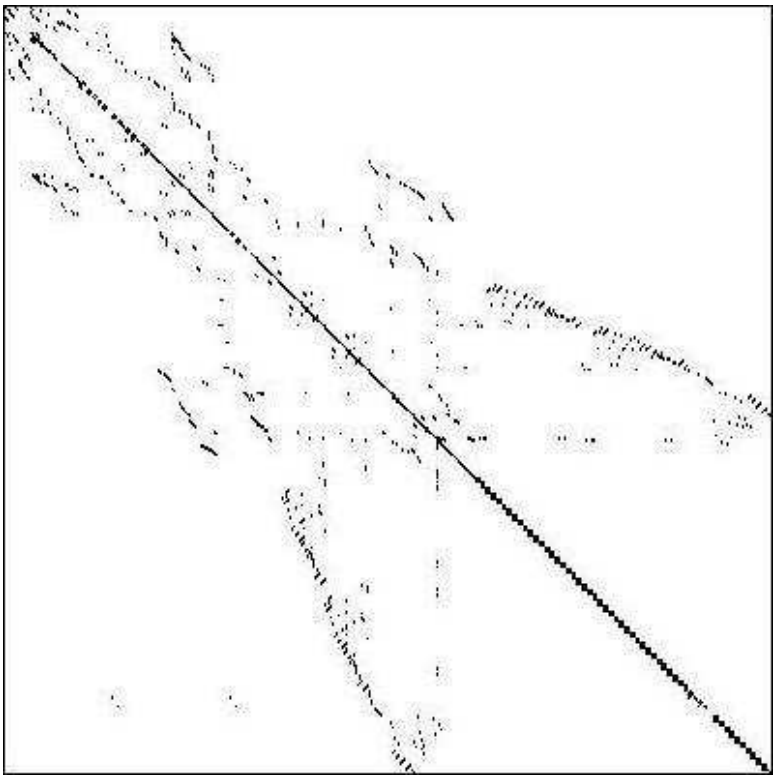

Figure 2: An Example of a Circuit Matrix

2 Objective and Motivation

Developing an interface from Kokkos to OSKI to provide Trilinos with access to OSKI's automatically tuned sparse matrix-vector multiplication kernel is the primary objective of this phase of the project.
Kokkos has always been able to take advantage of optimized dense BLAS kernels, but the available sparse kernels were not highly optimized. Significant performance improvements provided by this interface may lead to additional interfaces to OSKI functionality in the future. Improving the performance of the sparse kernels available to Kokkos would significantly contribute to the overall performance of Trilinos and therefore the many applications that use Trilinos libraries because of the high percentage of time that is spent in important sparse kernels.

3 Implementation

To integrate OSKI functionality into the Trilinos set of tools, we must make use of OSKI C-style objects[2]. Conversion from the default implementation of the Kokkos `CisMatrix` class, `HbMatrix`, to an OSKI compatible matrix was one option, but one that had potential

[2]The C language does not really have objects but, when coding in C with an object-oriented style, structs can be used in ways that make them look somewhat like classes.

limitations including some unnecessary overhead. It would have also been awkward to collect any additional information that OSKI might be able to use of after an `HbMatrix` had already been constructed.

The approach that we took was to design a new `OskiMatrix` class in Kokkos that provides an interface to the native OSKI matrix and still implements the `CisMatrix` class to insure compatibility with packages that assume the `CisMatrix` abstract data type. The methods of the `OskiMatrix` class are virtually identical to the methods of the `HbMatrix` class except that when initializing the structure of the `OskiMatrix`, it is possible to include some optional parameters that provide more information about the matrix, although it is not yet possible to pass all of the hints that OSKI can recognize. The `OskiMatrix` class also has the restriction that it cannot accept a matrix that is stored in generalized Harwell-Boeing (HB) matrix format, only classic HB matrix format.

We also created an `OskiMultiVector` class that implements the Kokkos `MultiVector` class. The methods of the `OskiMultiVector` class are virtually identical to the methods of the Kokkos `DenseMultiVector` class. One additional restraint on the `OskiMultiVector` class is that currently an `OskiMultiVector` must be strided in memory. The `DenseMultiVector` class allows for more general storage. In practice, this is not a severe restriction and it could be lifted if necessary.

In addition, we added a simple `OskiVector` class that inherits from the `OskiMultiVector` class. An `OskiVector` is simply an `OskiMultiVector` with exactly one vector. This is the same approach that Epetra [5] uses, but is different than the approach that Kokkos uses. Kokkos has a `DenseVector` class that implements the `Vector` class. We decided to break away from what Kokkos does in this case because there is not a clear advantage to the approach that Kokkos adopted, and having `OskiVector` inherit from `OskiMultiVector` prevents a lot of code duplication because instead of having two versions of a method that takes an `OskiMultiVector` or an `OskiVector` as a parameter, only one is necessary.

The final class we added is the `OskiSparseMultiply` class that is used to perform the matrix-vector multiplication. `OskiSparseMultiply` implements the Kokkos `SparseOperation` base class and is analogous to `BaseSparseMultiply`, which is the Kokkos class that can be used to multiply an `HbMatrix` by a `DenseMultiVector` or `DenseVector`. A performance comparison between the `OskiSparseMultiply` and `BaseSparseMultiply` classes is the focus of our results in Section 4.4.

Basic tests for each new class described above were added to the suite of Kokkos tests. The classes and associated tests are not yet in a releasable state, and are not included in the current Trilinos release.

4 Performance Tests

4.1 Test Problems

We tested the performance of the `OskiSparseMultiply` and the `BaseSparseMultiply` kernels on nine different sparse matrices of varying size, including six PDE matrices and three circuit matrices. All nine matrices are stored in classic HB

format and were downloaded from the University of Florida Sparse Matrix Collection [3]. The size (number of rows) and type of the matrices, along with a brief description of each matrix (provided in the matrix file) can be found in Table 1. All of the matrices are square.

Matrix Name	Size	Type	Description
crystk03.rsa	24696	PDE	FEM crystal free vibration stiffness matrix
msc10848.rsa	10848	PDE	Symmetric test matrix
pwtk.rsa	217918	PDE	Pressurized wind tunnel stiffness matrix
psmigr_3.rua	3140	PDE	Intercountry migration (sorted columns)
gemat12.rua	4929	PDE	Unsymmetric matrix
bcsstk28.rsa	4410	PDE	Solid element model
scircuit.rua	170998	Circuit	Many parasitics
hcircuit.rua	105676	Circuit	No parasitics
memplus.rua	17758	Circuit	Memory circuit

Table 1: Summary of Test Matrices

4.2 Test Platforms

The nine matrices were tested on three machines with a total of five different sets of configuration options. All of the machines were either SMP or dual core machines, but the kernels we ran are not able to take advantage of this potential parallelism. One of the machines was rebooted with a non-SMP Linux kernel to verify that there were no performance differences stemming from the kernel. Although the tests were unable to take advantage of the parallelism, it may have been useful when there was an extra load on the third machine (which we used for three of the five configurations) as we were unable to secure dedicated time on that machine.

The first machine, which is named HerouxSMP, has four Intel Pentium III processors running at 500 MHz, a 512 KB L2 cache and 1 GB total RAM. The machine was used for one test configuration using GCC 3.2.2 compilers and the following set of compiler flags: `CXXFLAGS=-O3 CFLAGS=-O3 FFLAGS=''-O3 -funroll-all-loops''`.

These flags are suggested for high performance in the Epetra Performance Optimization Guide [6]. Epetra is a Trilinos package that has a superset of the capabilities of Kokkos, but can use only real, double precision values.

The second machine, which we shall call Dimension, has a dual core 64-bit Intel Pentium D 820 processor running at 2.8 GHZ, a 1 MB L2 cache and 1 GB DDR2 SDRAM running at 533 MHz. The machine was used for one test configuration using GCC 4.0.2 compilers and the same compiler flags that were used on HerouxSMP.

The third machine, which we shall call Intel, has dual 32-bit 3.06 GHz Xeon processors, a 512 KB L2 cache and 4 GB total RAM. The machine was used for three test configurations. "Intel GCC" used GCC 3.2.3 compilers and the same compiler flags that were used on HerouxSMP. The other two configurations used Intel 8.1 compilers. "Intel" used the following set of compiler flags: `CFLAGS=-O3 CXXFLAGS=-O3 FFLAGS=-O3`, "Intel

Opt" used this set:
`CFLAGS=''-O3 -tpp7 -mcpu=pentium4 -march=pentium4 -std=c99''`
`CXXFLAGS=''-O3 -tpp7 -mcpu=pentium4 -march=pentium4 -std=c99''`
`FFLAGS=''-O3 -tpp7 -mcpu=pentium4 -march=pentium4 -std=c99''`,
which is the set of flags that OSKI selected during its configuration. We chose to use the exact flags that OSKI used for at least one test run to guarantee that compiler flags could not be credited with any speedup that was realized.
The configuration flags noted above were used to configure Trilinos only. For every machine, OSKI was configured using its automatic configuration capability. We did not explicitly specify any optimization flags.

4.3 Test Setup

Each result that is reported is an average of three test runs. Each matrix-vector multiplication was performed sixty times per test run to ensure that the smaller tests would run for enough time to obtain an accurate performance measurement.

4.4 Results

The most obvious and important result that we found was that the new OSKI-based Kokkos kernel showed an average speedup of between 1.73 and 1.98 on the five platforms. We calculated speedup by dividing the performance (in MFLOPS) of the OSKI-based kernel by the performance of the existing Kokkos kernel. The average speedup on all problems for each test platform is shown in Figure 3.

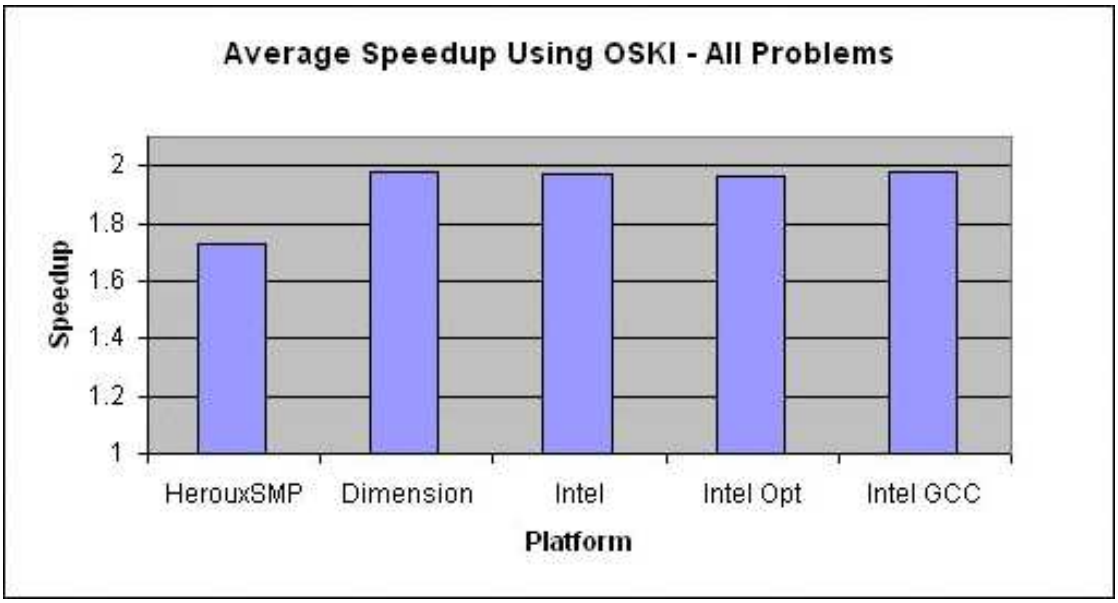

Figure 3: Average Speedup

We predicted that the matrix-vector multiplication kernel performance would be lower for circuit matrices than for PDE matrices, and that did turn out to be the case. Since OSKI

uses a wide array of optimization techniques, we were curious to see if the speedup would be higher for PDE matrices or circuit matrices. Recall that our implementation does not currently allow users to take advantage of all of the hints about the problem that can be passed to OSKI improve performance. These results are only meant to be a baseline for default behavior. It turns out that the average speedups for PDE matrices and circuit matrices were very close, ranging from 1.67 to 2.01 for PDE matrices and 1.85 to 1.98 for circuit matrices. The average speedup for each problem type is shown in Figure 4.

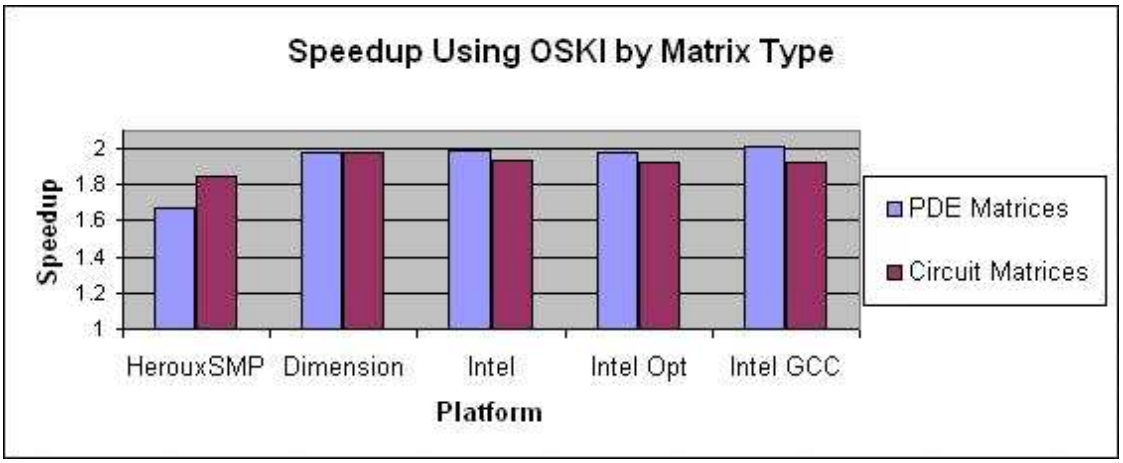

Figure 4: Average Speedup by Matrix Type

Another interesting result is the performance in MFLOPS considering both problem type and whether or not an OSKI-based kernel was used to perform the matrix-vector multiplication. This information is provided in Figure 5. In this figure, we see that the performance of the OSKI enhanced kernels for circuit problems turned out to be higher than the performance of the existing Kokkos kernels for PDE problems! However, we cannot make a general conclusion based on this data because we are using small sampling of PDE and circuit matrices and there are too many variables involved including matrix size, number of nonzero values, etc. The matrix size and the number of nonzero values are significant because in all of the tests that we ran, including some very small problems not reported here, there was a definite correlation between problem size and performance, which is not surprising.

Despite our inability to determine if the performance of the OSKI-based kernels for circuit problems is in fact generally higher than the performance of the existing Kokkos kernels for PDE problems, by looking at the specific performance results for every problem on every platform, and considering the fact that we are using a variety of both PDE and circuit matrix problem sizes, we can see that the performance of the OSKI-based kernels for circuit problems is at least comparable to the performance of the existing Kokkos kernels for PDE problems. On HerouxSMP and Dimension, the performance of every circuit problem using OSKI-based kernels is higher than the performance of every PDE problem when not using the OSKI-based kernels. On Intel, there are a handful of exceptions to this observation across the three configurations. This is an exciting observation because using the OSKI enhanced kernels, it is now possible for Kokkos users to get performance on circuit problems

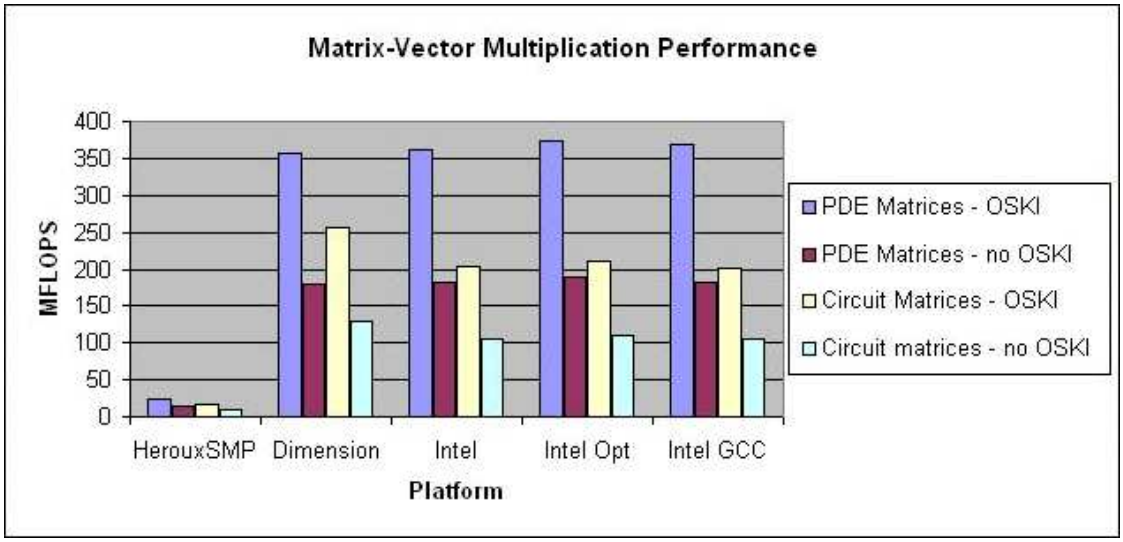

Figure 5: How OSKI Affects Performance

that is comparable to what was previously possible only on PDE problems. These detailed results can be found in Figure 6, Figure 7, and Figure 8. Note that along the x-axis of the graphs in these figures that the name of the matrix followed by an "O" indicates that the performance listed is for the OSKI-based kernel.

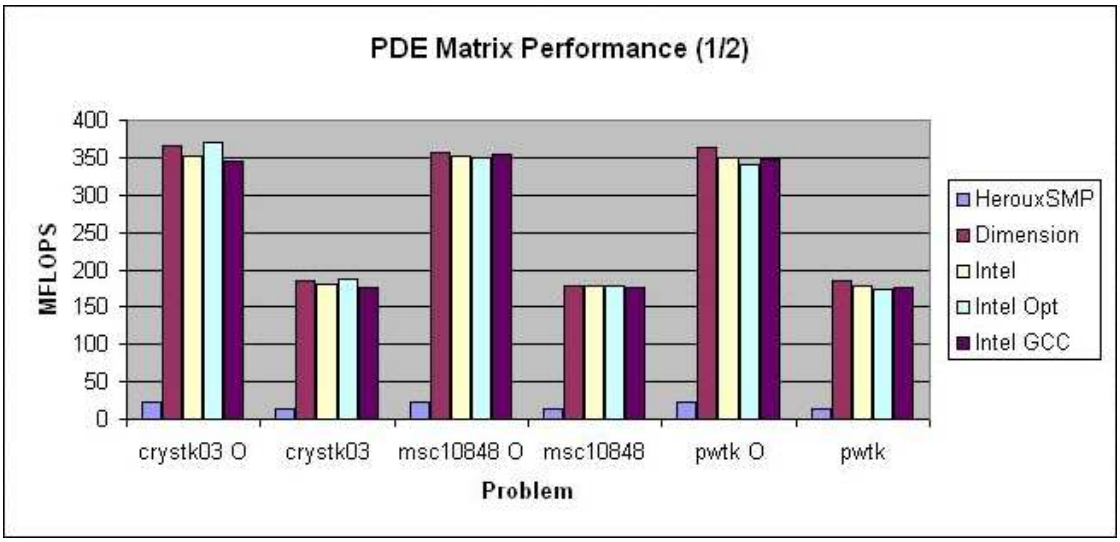

Figure 6: Performance Details by Problem

It is interesting to note that other sets of compiler flags were tested, including not unrolling loops and lower levels of optimizations, and for our particular kernels, the results were very similar. More importantly, the performance and speedups were very consistent across test runs, which makes it easier to draw conclusions from the test results.

For example, it was simple to conclude that the OSKI interface improved performance because every individual test run for every problem that we ran (including some problems

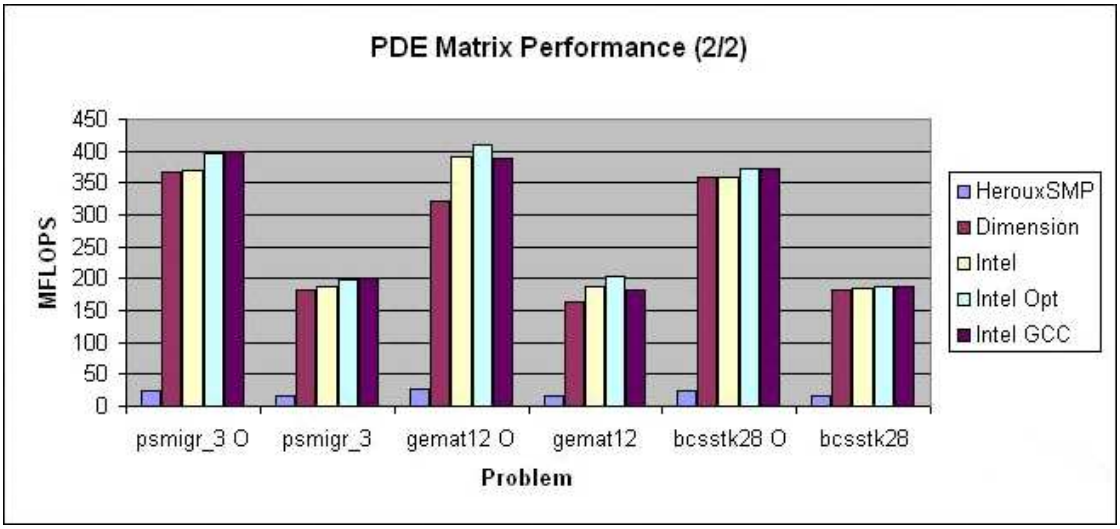

Figure 7: Performance Details by Problem

not reported here that were as small as 4 by 4) on every platform, showed a performance improvement using the OSKI-based kernel. The performance of both kernels decreased as the problem size decreased, but the minimum speedup that was seen for any problem of any size was 1.06 for the 4 by 4 problem. We did not include this in our primary results because of the size of the problem, but we did want to run the test to verify that there is not significant overhead associated with calling our interface.

5 Future Work

Given the performance improvements that OSKI has provided, we plan to write additional, and more flexible Kokkos interfaces to OSKI. For example, interfaces to OSKI's triangular solve and matrix power times a vector ($A^n x$) capabilities could be added. We will also consider adding the ability to pass additional information about the problem to OSKI.
Also, we will research tuning techniques for improving the performance of circuit matrix problems, which despite the fact that circuit matrix problems are less common than PDE matrix problems, is an important research area.

6 Conclusion

We have introduced four new Kokkos classes that provide an interface to OSKI's sparse matrix-vector multiplication kernel. We then presented performance results comparing the performance of this new interface to the existing Kokkos sparse matrix-vector multiplication kernel. Using the new interface, we achieved a speedup of more than 1.9, on average, for both PDE matrix and circuit matrix problems.
In addition, despite the fact that the performance for circuit problems is naturally lower than the performance for PDE problems, without any hand-tuning, the interface provides

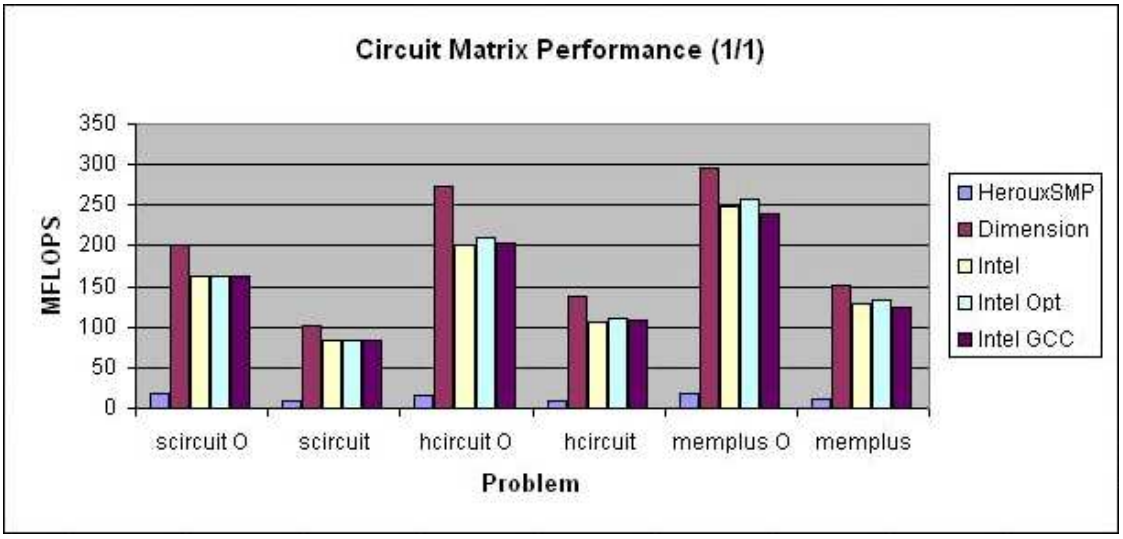

Figure 8: Performance Details by Problem

Kokkos users with a level of performance for circuit problems that is comparable to what was formerly only available for PDE problems.

References

[1] BILMES, J., AND ET. AL. Phipac fast matrix multiply home page. http://www.icsi.berkeley.edu/ bilmes/phipac/, 2005.

[2] BLACKFORD, L. S., DEMMEL, J., DONGARRA, J., DUFF, I., HAMMARLING, S., HENRY, G., HEROUX, M., KAUFMAN, L., LUMSDAINE, A., PETITET, A., POZO, R., REMINGTON, K., AND WHALEY, R. C. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software 28*, 2 (2002), 135–151.

[3] DAVIS, T. University of florida sparse matrix collection. http://www.cise.ufl.edu/research/sparse/matrices, NA Digest, vol. 92, no. 42, October 16, 1994, NA Digest, vol. 96, no. 28, July 23, 1996, and NA Digest, vol. 97, no. 23, June 7, 1997.

[4] HEROUX, M., BARTLETT, R., HOEKSTRA, V. H. R., HU, J., KOLDA, T., LEHOUCQ, R., LONG, K., PAWLOWSKI, R., PHIPPS, E., SALINGER, A., THORNQUIST, H., TUMINARO, R., WILLENBRING, J., AND WILLIAMS, A. An overview of the trilinos project. *ACM Transactions on Mathematical Software 31*, 3 (September 2005), 397–423.

[5] HEROUX, M. A. Epetra home page. http://software.sandia.gov/Trilinos/packages/epetra, July 2004.

[6] HEROUX, M. A. Epetra Performance Optimization Guide. Tech. Rep. SAND2005-1668, Sandia National Laboratories, 2005.

[7] HEROUX, M. A. Kokkos home page. http://software.sandia.gov/trilinos/packages/kokkos, 2005.

[8] LAWSON, C., HANSON, R., KINCAID, D., AND KROGH, F. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software 5* (1979).

[9] VUDUC, R., DEMMEL, J. W., AND YELICK, K. A. The optimized sparse kernel interface (oski) library user's guide for version 1.0. Tech. rep., University of California Berkeley, 2005.

[10] WHALEY, R. C., AND PETITET, A. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience 35*, 2 (February 2005), 101–121. `http://www.cs.utsa.edu/~whaley/papers/spercw04.ps`.