SANDIA REPORT

SAND2017-5953
Unlimited Release
Printed June, 2017

Optimizing the Performance of Sparse-Matrix Vector Products on Next-Generation Processors

S.D. Hammond and C.R. Trott
Center for Computing Research
Sandia National Laboratories
Albuquerque, NM, 87185

{sdhammo, crtrott}@sandia.gov

Sandia National Laboratories

SAND2017-5953
Unlimited Release
Printed June, 2017

Optimizing the Performance of Sparse-Matrix Vector Products on Next-Generation Processors

S.D. Hammond and C.R. Trott
MS 1318
Center for Computing Research
Sandia National Laboratories
Albuquerque, NM, 87185-1318

{sdhammo, crtrott}@sandia.gov

Matrix-vector products are ubiquitous in high-performance scientific applications and have a growing set of occurances in advanced data analysis activities. Achieving high performance for these kernels is therefore paramount, in part, because these operations can consume vast amounts of application execution time.

In this report we document the development of several sparse-matrix vector product kernel implementations using a variety of programming models and approaches. Each kernel is run on a broad set of matrices selected to demonstrate the wide variety of matrix structure and sparsity that is possible with a single, generic kernel. For benchmarking and performance analysis, we utilize leading computing architectures for the NNSA/ASC program including Intel's Knights Landing processor and IBM's POWER8.

Acknowledgment

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

The machines used in this report for benchmarking and performance analysis are provided for research evaluation by Sandia's NNSA/ASC Advanced Architecture Test Bed project. We are very grateful to the system admins and support team for their help in making this hardware available for benchmarking.

Contents

Chapter 1

Optimizing the Performance of Sparse-Matrix Vector Products on Next-Generation Processors

1.1 Background

Matrix-vector products are ubiquitous in high-performance scientific applications. In fact, many of the earliest use of proto-computers were dedicated to the solution of systems of equations using these primitive computational kernels. Most recently, the use of matrix-vector linear algebra primitives has been prevelent in the domain of data analytics and machine learning – perhaps now the fastest growing sub-domain of high performance computing.

The representation of matrices, and indeed vectors, can take many forms [4, 7, 8, 9]. The most natural, although expensive form, is a dense representation in which every matrix element is represented by a stored value. For applications that are relevant to Sandia, for instance, structural analysis, systems modeling, radiation solvers, computation-fluid dynamics (CFD) *etc.*, a high to very high proportion of these values are represented by zeros. Encoding them into a dense formulation is wasteful and can consume vast amounts of memory. It is not uncommon for non-zero terms to be present in much less than 1% of the matrix for problems of interest. With this in mind, considerable attention has been paid to the design of efficient representation of *sparse* matrices in which only non-zero values are stored, reducing memory consumption and improving computation performance by focusing mathematical operations on only non-zero terms. Alternatives to the use of fully sparse matrices, can also include block-based definitions which optimize element access and operations for well defined block structures in the matrix, or, vector-based sparse matrices which optimize for the storage of non-zero values commensurate with the use of structures to efficiently load and compute over the elements in vector-processors.

Application codes and mathematics libraries, in particular, the Trilinos solver project [6], have settled on the use of Compressed-Row Storage (CRS) format matrices over time, although, specialized use cases for other formats do appear in many routines. In this format a sparse matrix is defined by an array of coefficients, most commonly stored in IEEE 64-bit floating point format, an array of integer values which represent column indices and an array of row offsets which define the start and end of a row. Sparse-matrix vector products are executed by iterating over these arrays one row at a time, then performing iteration through each of the elements found in a row. Parallelism can be added by breaking the execution over rows into a discrete set of rows per executing thread.

Our benchmarking of SpMV routines within Sandia's application portfolio has shown extensive use of SpMV operations over CSR matrices with as much as 90% of the execution time being spent in this class of routine. Common cases show that between 40-60% of complete application execution time is spent in SpMV depending on domain area. With such a significant amount of time being spent in this class of operation, it will come as no surprise to the reader that considerable effort has therefore been spent on analyzing the behavior of these kernels and optimizing them for each computing platform of interest.

During 2016, anecdotal reports appeared in Sandia's Center for Computing Research that the SpMV operations provided by the Kokkos [5] and TPetra kernel [1] packages were operating slower than expected. The kernels in use were written to utilize dynamic execution over rows such that threads would execute their local allocation of rows and then attempt to steal any additional work from other threads when complete. Dynamic execution was used because of a belief that the potentially random structure which any call to sparse-matrix vector productions could present would cause load imbalance over the executing threads. For instance, a single very dense row would require much longer to compute that a particularly sparse row. Since the sparsity of the rows is potentially random and diverse, a dynamic execution in theory provides an attempt to prevent load imbalance from underutilizing some threads.

In this Sandia technical report we show that, for next-generation processors, this design choice does in fact make little sense since the use of dynamic execution can provide considerable overhead. The static structure of the CRS matrix for the duration of the SpMV call presents, instead, an opportunity to utilize a simple static load balance and have the threads use this work schedule to execute efficiently. This can significantly reduce the overheads associated with SpMV execution while continuing to insulate the caller from experiencing underutilization of the compute resources. We compare this approach to the prior dynamic execution kernel from Kokkos (which was in turn used by TPetra), to several other implementations, including, the use of a purely static, naive partition of rows over threads, the use of OpenMP [2] dynamic execution and the use of a simple structure analyzing dispatch written in both Kokkos and direct OpenMP. The structure analyzing kernel written directly in OpenMP provides the best possible performance, up to 9X over that of the original kernel – a significant improvement on a kernel which is traditionally regarded as memory bandwidth bound. This implementation with small modifications has subsequently been accepted for deployment in the new KokkosKernels subpackage of Trilinos that will provide mathematical primitives for next-generation computing platforms.

The remainder of this report is laid out as follows: the rest of Chapter 1 is dedicated to discussion of SpMV implementations using a variety of programming models and libraries of interest to Sandia. In Chapter 2, we present benchmarking results from next-generation Intel Knights Landing (KNL) and IBM POWER8 processors. We conclude in Chapter 3 with a short review of the report. Additional data chapters/appendices are provided for the reader to record the performance which has been achieved.

1.2 Implementation of CSR-SpMV Operations

1.2.1 Base Implementation

Listing 1.1. Base Implementation of a Sparse Matrix Vector Product

```
for(int row = 0; row < n; ++row) {
  double sum = beta*ycoefs[row];

  for(int i = matrixRowOffsets[row]; i < matrixRowOffsets[row+1]; ++i) {
    sum += matrixCoefs[i]*xCoefs[matrixCols[i]];
  }

  yCoefs[row] = sum;
}
```

The base implementation of a sparse-matrix vector product using a CRS matrix is shown above. In this implementation the code iterates over the rows the matrix in turn, processing the matrix-elements of each row in the inner loop. The sparsity of the CRS definition can be seen in the use of the `matrixCols` array to provide indirect access to the coefficients of the `x` vector. No parallelization is provided in this example but we use this basic kernel defintion as the foundation for the implementations that follow.

The performance expectation of this implementation is that for each matrix coefficient that we plan to process (each non-zero of the CRS matrix), we must load a minimum of 4-bytes for each column entry and 8-bytes for each matrix coefficient. There is no reuse of the matrix elements ensuring we must load at least 12-bytes for every non-zero entry in the matrix from memory. We may also need to load a further 8-bytes for each coefficient of the `x` vector however, such values may also be cached because in the case where subsequent rows hit the same locations in the vector. It is not uncommon to find such behavior in matrices which are diagonally dominant meaning we may load only a small number of new values for each row and reuse them in subsequent rows. This observation provides a rough guidance that at a minimum we will load 12-bytes from memory in order to calculate two floating-point operations (a multiply and an addition which may be implemented in a single fused-multiply-add (FMA) operation). In the case where we load the `x` values from memory this adds an additional 8-bytes, giving a total of 20-bytes per 2 floating-point operations. This gives a bound of between 6 and 10-bytes per mathematical operation which far exceeds the capability of modern memory systems. These are typically designed to provide in the order of 0.1 to 0.5 bytes of memory bandwidth per floating-point operation of the processor. For this reason, we typically thing of SpMV product performance as being heavily correlated with memory subsystem performance. It is important for us to ensure strong streaming capability to maximize memory bandwidth (allowing prefetchers to stream data from memory) which is achieved by having each processor core executing the operations above perform the longest sequence which is possible – *i.e.* that when a processor executes the row j, it also performs the operations for $j + 1$ subject to all processor cores executing roughly equivalent work. This observation will become important later in our discussion.

1.2.2 Direct OpenMP with Static Thread Scheduling

In Section 1.2.1 we implemented a basic serial SpMV multiplication kernel. In this kernel updates to the `y` vector are all independent by virtue of the fact the kernel processes each row independently. Since there is no dependency between each row in this kernel, a naive parallel implementation can be formed by applying an OpenMP `parallel-for` clause ahead of the outer for-loop (than iterates over rows). This creates a parallel kernel which utilizes vendor-defined scheduling of iterations to threads. In all cases we have used, this equates to a static decomposition of rows to threads in which the runtime takes the maximum number of iterations (in this case rows of the matrix) and as close to evenly as possible divides these over executing threads. The theory is that this creates approximately equal amounts of work which would be the case if every row in this SpMV kernel had the same number of non-zero entries, however, due to the unknown sparsity pattern of the CRS matrix being used, this cannot be guaranteed.

1.2.3 Direct OpenMP with Dynamic Thread Scheduling

An alternative to static scheduling (evenly dividing work over executing threads) is to form a series of work chunks – in this case chunks of rows to be calculated over – and allow the threads to dynamically grab a chunk to work on. This policy is implemented in our study by using the same OpenMP `parallel-for` construct as with static scheduling but, instead, forcing the use of a dynamic work schedule. OpenMP-compliant compilers provide this as part of their standard implementation. This may have the effect of causing contention around the work queue and breaking the strong linear pattern for prefetchers which we would like to see to improve memory bandwidth.

1.2.4 Base Kokkos Implementation (Dynamic Execution Policy)

The base Kokkos implementation used for this work provides a similar strategy – using dynamic work allocation but instead perform a rough static scheduling of work chunks to executing threads so that they can benefit from the near linear ordering of work items for as long as possible. Once each threads local work is complete, they are free to steal work from neighboring threads. The Kokkos dynamic execution policy is a custom implementation and does not call into the standard OpenMP dynamic schedule.

1.2.5 Intel Math-Kernel Library (MKL)

Intel provide the Math-Kernel Library (MKL) for use on X86-based processors. The MKL was initially started as a collection of heavily optimized BLAS and LAPACK routines which provided superior math primitives on Intel's hardware. Since the initial released, the MKL has team has gradually added sparse matrix support along since the traditional dense matrix/vector support found in BLAS and LAPACK. In this report we utilize the Intel CRS sparse-matrix support and benchmark its performance.

1.2.6 Direct OpenMP with Non-Zero Load Balancing Structure Detection

In this implementation, we take a more aggressive approach to optimizing SpMV execution. Because the structure of the matrix is easily discoverable and static for the duration of each SpMV call – and is very often static for a large number of calls to SpMV in iterative solvers – we can perform a static load balancing of work to threads using a custom function. Our custom balancer scans each row in the matrix and then decides to allocate an approximately even number of non-zeros to each thread subject to the use of complete rows, *i.e.* the work is allocated to threads in linear groups of rows so that each is given roughly the equivalent number of non-zeros to process. This may mean parts of the matrix with denser rows will result in an individual thread receiving fewer rows to execute but still roughly the same number of non-zeros. Since it is the presence of non-zeros in the matrix that cause load imbalance, our approach should reduce thread idling while ensuring that each thread maximizes the prefetching and linear-flow of the rows it is processing. Since the work is statically mapped to threads there is no work stealing performed and no centralized work queue at which contention can form relieving one of the main bottlenecks associated with dynamic execution. We choose to make this implementation directly in OpenMP bypassing the use of Kokkos execution dispatch.

1.2.7 Kokkos with Non-Zero Load Balancing Structure Detection

In this version we take the OpenMP non-zero balanced version described above and perform a re-implementation using Kokkos to control execution dispatch.

Chapter 2

Case Study: Benchmarking SpMV Implementation Performance on Next-Generation Processors

2.1 Hardware Resources

2.1.1 Intel Xeon Phi 7250 (Knights Landing) Processor

We utilize the Intel Xeon Phi 7250 (Knights Landing) many-core processor which is currently being installed on the NNSA/ASC Trinity Advanced Technology System (ATS). This processor is the first in Intel's Xeon Phi to be self-hosted (*i.e.* it does not require an accompanying Xeon processor to load or manage the operating system). It comprises of 34 dual-core 'tiles', each of which holds two, quad-hyperthreaded processor cores connected to the interprocessor mesh network-on-chip via a 1MB shared L2 cache. The processor cores operate at 1.4GHz in standard operation but drop to 1.2GHz when frequent dense vectorized mathematics operations are used to avoid thermal overload. The on-chip fabric operates at 1.7GHz to ensure low-latency/high bandwidth inter-core communication. Each Knights Landing in use on Trinity provides 96GB of DDR4 main system memory arranged as 6 channels of 16GB. An additional 16GB of on-package high-bandwidth MC-DRAM memory is provided that can yield over 5X more bandwidth but at a smaller capacity. We typically use the term high-bandwidth memory (HBM) and MC-DRAM interchangeably to refer to this memory store. The HBM can be used for direct memory allocation via specialized memory allocation functions or, alternatively, as a cache for the DDR4 when booted into this special mode during BIOS initialization.

For compilation of our codes we use the Intel 17 Update 1 compiler with `-O3 -xMIC-AVX512 -fopenmp` optimization/code generation flags. Results reported with MKL are using the mathematics library that ship with this compiler from Intel.

2.1.2 IBM POWER8 Processor

Sandia was the first DOE laboratory to deploy IBM's POWER8 processor. Since its initial release several variants of the design have been made available with tuning to suit slightly different markets and design points. For this study, we make use of the latest IBM design for High Performance Computing – the IBM S822LC. Each node comprises of two 8-core POWER8 designs running at a maximum frequency of 3.25GHz. The cores of the POWER8 are 8-way SMT giving the appearance of 64 hardware threads per socket. There is a 512kB and 8MB L2 and L3 cache respectively per core and 64MB of shared L4 cache per socket. Considerable off chip bandwidth is available to on-board Centar memory buffer chips that orchestrate memory transactions to improve bandwidths and latencies. While the latest S822LC nodes at Sandia feature 2 NVIDIA P100 GPUs per socket, connected via NVIDIA NVLINK high performance GPU connection fabric, this study focuses solely on the performance offered by the IBM POWER8 processor. To mitigate the effect of high NUMA effects between sockets, our benchmarking is limited to a single socket as would be the case when running a single MPI rank per socket and utilizing explicit messaging passing between sockets on the node and to further ranks off-node. We argue this is the fairest benchmarking of the processor and the most representative of how our codes would use this hardware in a production setting.

For compilation of the benchmarks on POWER8 we use the GCC 5.4.0 compiler with `-O3 -mcpu=power8 mtune=power8 -fopenmp` optimization and code generation flags. MKL is not available on this platform so results are not provided.

2.2 Performance Metrics

Historically we have compared the performance of processor designs using a floating-point operations per-second metric (commonly referenced as GFLOP/s, TFLOP/s *etc.*). For each matrix element we count two floating-point operations, the basic multiply of the element against the vector and then a floating-point addition to add the contribution to the running sum.

2.3 Performance Comparison

Node	**Kokkos**	**OMP-Stat**	**OMP-Dyn**	**OMP-NNZ**	**Kok-NNZ**	**MKL**	**TPetra**
KNL-DDR4	9.83	12.18	0.80	12.70	10.60	5.43	5.52
KNL-Cache	17.17	19.17	0.85	19.87	17.54	6.33	12.53
KNL-HBM	17.00	19.13	0.86	20.13	17.66	6.28	12.34
POWER8	7.29	8.67	0.79	10.17	7.97	N/A	N/A

Table 2.1. Benchmarked SpMV Performance in GFLOP/s averaged over the selected Sparse-Matrices from the University of Florida collection [3] and Sandia codes

Node	**Kokkos**	**OMP-Stat**	**OMP-Dyn**	**OMP-NNZ**	**Kok-NNZ**	**MKL**	**TPetra**
KNL-DDR4	1.00	1.24	0.08	1.29	1.08	0.55	0.56
KNL-Cache	1.00	1.12	0.05	1.16	1.02	0.37	0.73
KNL-HBM	1.00	1.13	0.05	1.18	1.04	0.37	0.73
POWER8	1.00	1.19	0.11	1.40	1.09	N/A	N/A

Table 2.2. Averaged Benchmarked SpMV Performance Speedup for the selected Sparse-Matrices from the University of Florida [3] collection and Sandia codes

Tables 2.1 and 2.2 present the benchmarked SpMV kernel performance from each of our implementations. In the former we present the mean GFLOP/s achieved by matrix. For each matrix we benchmark the run 10 times and take the harmonic mean as the benchmark reports the computational rate. In the second table we present an alternative view of the data as the speedup achieved. Note that the baseline for this is the optimized Kokkos implementation which utilizes dynamic execution. For the KNL we show the performance/speedup relative to the pre-existing TPetra implementation, noting that the inspection-based implementation achieve in excess of 2X correlating with the reports of slow computational performance of the existing kernels. We note that the fatest implementation is the kernel written directly in OpenMP (not using Kokkos) which uses the static non-zero load balancing method. We attribute the use of OpenMP to enabling additional optimization for the source code which is obscured by the use of the Kokkos abstraction layers.

Chapter 3

Conclusion

Sparse-Matrix vector products occupy a significant number of the computing cycles at leading HPC centers such as Sandia National Laboratories. They are a vital component in solving complex scientific and engineering problems, and have a growing place in large-scale data analytics calculations.

In this report we have provided benchmarked performance results for several implementations of the basic SpMV algorithm. The motivation for this study was anecdotal reports of slower than expected kernel performance which, this report shows, were indeed slower than some alternatives.

The conclusion of our report is that higher performance alternatives are available. As a result of this work the optimized OpenMP non-zero balancing kernel has been adopted in the forthcoming Kokkos-Kernels package being developed for mathematic primitives in the Trilinos framework. This kernel will provide high performance for future releases of Trilinos that are being integrated into our production application portfolio. We look forward to the reevaluation of solver performance once these newer kernels are successfully integrated.

Chapter 4

Results from Intel Xeon Phi 7250 (Knights Landing) Processor (DDR4 Only)

	KK	TPETRA	MKL	OMP-DYN	OMP-STAT	OMP-INSP	KK-INSP
Andrews_Andrews	9.91	5.65	3.6	0.37	12.39	12.45	11.02
ANSYS_Delor64K	18.42	6.88	4.79	0.28	24.24	16.96	13.02
Bai_olm2000	0.49	0.1	0.56	0.08	0.97	1	0.69
Bai_rbsb480	1.47	0.27	1.19	0.42	1.73	1.9	1.34
Boeing_bcsstk39	41.19	14.55	10.49	1.32	54.89	50.74	45.96
Boeing_bcsstm34	2.14	0.38	1.53	0.54	3.01	3.07	2.18
Boeing_crystm01	7.48	1.57	5.29	0.56	10.94	10.96	7.68
Boeing_crystm02	17.61	4.45	10.54	0.64	24.78	24	17.53
Boeing_crystm03	23.71	7.33	11.57	0.68	34.06	32.15	24.83
Boeing_ct20stif	21.71	11.72	10.02	1.51	22.43	27.65	24.87
Boeing_pwtk	11.11	11.16	11.52	1.38	11.9	12.1	11.07
Bourchtein_atmosmodl	8.15	8.06	8.68	0.18	8.65	8.77	8.25
Bova_rma10	19.05	12.4	8.58	1.5	24.33	41.69	27.69
Brethour_coater1	1.6	0.29	1.2	0.27	2.4	2.31	1.66
Brethour_coater2	13.38	2.98	8.38	0.6	18.56	18.35	13.41
DIMACS10_id2010	11.61	5.27	4.13	0.12	16.21	15.28	12.07
DIMACS10_italy_osm	4.65	4.75	2.32	0.05	4.3	4.3	4.72
DIMACS10_ne2010	12.54	6.82	4.45	0.12	16.87	16.71	12.92
DIMACS10_road_central	2.73	2.77	1.36	0.06	2.61	2.6	2.63
DIMACS10_vsp_vibrobox_scagr7-2c_rlfddd	2.7	3.61	1.05	0.33	3.07	9.97	9.36
DNVS_m_t1	12.32	11.39	11.33	2.75	12.61	12.72	12.54
DNVS_shipsec1	12.09	10.99	11.3	1.5	12.28	12.52	12.29
Dziekonski_dielFilterV2real	10.58	10.67	6.17	1.10	10.13	10.72	10.62
FIDAP_ex1	0.39	0.06	0.32	0.13	0.52	0.5	0.39
FIDAP_ex10	4.55	0.82	3.38	0.52	6.38	6.41	4.61
FIDAP_ex11	40.4	11.47	9.85	1.86	56.86	55.95	44.16
FIDAP_ex15	6.81	1.42	4.49	0.39	9.99	10.13	6.99
FIDAP_ex18	4.78	1.03	3.71	0.32	7.6	7.18	5.4
FIDAP_ex25	2.12	0.39	1.7	0.46	3.03	3.01	2.18
Fluorem_HV15R	11.81	11.76	11.19	3.99	11.66	11.87	11.87
Freescale_memchip	6.88	7.01	3.5	0.14	5.93	5.52	7.31
GHS_indef_blockqp1	1.61	0.93	1.43	0.31	1.66	6.79	7.53
GHS_indef_bmw3_2	11.19	11.09	11.37	1.19	12	12.02	10.97
GHS_indef_brainpc2	2.43	1.03	1.9	0.18	2.76	6.73	6.22
GHS_indef_ncvxqp7	12.14	5.12	4.19	0.19	15	11.5	8.82
GHS_psdef_audikw_1	11.31	11.26	5.49	1.99	10.20	11.37	11.29
GHS_psdef_bmwcra_1	11.52	11.33	11.75	1.93	12.44	12.56	11.33
GHS_psdef_inline_1	10.77	10.66	5.93	1.94	9.62	10.94	10.78
GHS_psdef_ldoor	11.58	11.44	10.51	1.21	11.56	11.76	11.57
GHS_psdef_s3dkq4m2	12.98	10.98	11.46	1.41	13.85	13.94	13.46
GHS_psdef_wathen100	19.79	5.69	10.13	0.45	28.22	26.44	20.04
GHS_psdef_wathen120	20.28	6.9	10.21	0.45	30.53	28.49	21.59
Grund_bayer04	7.82	2.16	5.22	0.22	12.12	5.78	4.34

Hamm_bcircuit	9.6	3.91	3.84	0.16	14.17	14.04	10.69
Hamm_hcircuit	8.92	4.94	2.78	0.14	11.39	11.97	8.80
HB_bcsstk02	0.4	0.06	0.22	0.16	0.55	0.56	0.4
HB_bcsstk13	6.38	1.25	4.53	0.91	9.59	9.53	6.75
HB_lnsp_131	0.03	0	0.03	0.01	0.06	0.07	0.05
HB_lnsp_511	0.2	0.04	0.2	0.06	0.35	0.36	0.25
HB_mbeacxc	3.42	0.76	2.53	1.23	4.37	5.3	3.61
HB_mbeaflw	3.41	0.75	2.54	1.23	4.3	5.3	3.62
HB_orani678	4.03	1.14	2.56	0.82	5.56	5.46	3.84
Hollinger_mark3jac020	3.89	0.75	2.57	0.16	5.56	5.66	4.10
IBM_dc1	1.81	1.22	1.38	0.16	1.67	3.43	3.65
IBM_EDA_ckt11752_dc_1	8.48	3.39	4.11	0.19	12.34	10	7.75
IBM_EDA_trans4	1.81	1.22	1.39	0.15	1.67	3.44	3.6
IPSO_TSC_OPF_300	6.05	3.43	4.56	2.27	6.48	26.55	20.82
Janna_Fault_639	10.27	10.27	10.69	1.19	10.52	10.54	10.24
Janna_Hook_1498	11.27	11.21	11.21	1.02	11.37	11.39	11.23
Janna_StocF-1465	9.80	9.74	9.44	0.39	10.03	7.64	9.45
JGD_Homology_m133-b3	8.76	5.92	3.78	0.1	13.75	13.33	10.19
JGD_Homology_n3c6-b7	3.92	0.71	2.92	0.21	5.62	5.29	3.98
JGD_Homology_n4c6-b6	16.49	7.28	7.23	0.2	23.63	23.14	17.71
JGD_Homology_shar_te2-b2	8.30	5.15	3.65	0.07	11.67	11.69	8.93
JGD_Kocay_Trec13	30.46	7.83	11.25	9.16	39.49	43.52	35.98
JGD_Relat_relat9	8.29	8.19	1.71	0.08	7.5	1.47	0.87
JGD_Trefethen_Trefethen_20000	21.88	6.92	9.87	0.8	31.26	30.99	24.12
Koutsovasilis_F1	11.41	11.14	4.32	1.73	8.07	11.87	11.46
LPnetlib_lp_osa_30	0.45	0.68	0.35	1.93	0.48	2.76	2.87
Meszaros_bas1lp	17.86	6.92	6.4	2.81	24.1	31.99	24.75
Meszaros_mod2	7.11	2.37	4.36	0.16	9.05	8.95	6.64
Meszaros_model4	3.7	0.69	2.66	0.65	5.13	4.77	3.64
Meszaros_stat96v1	34.04	7.46	10.17	2.51	44.49	44.32	33.82
Mittelmann_pds-80	6.39	4.22	2.51	0.19	6.99	7.01	6.78
Norris_torso2	18.18	9.87	8.17	0.19	26.52	25.83	20.07
Oberwolfach_bone010	12.02	12.02	12.03	1.88	12.15	12.22	12.05
PARSEC_SiNa	12.19	2.71	7.41	0.91	16.76	18.25	13.19
Rajat_rajat16	5.00	3.39	2.95	0.2	5.82	9.02	7.45
Rajat_rajat25	5.57	2.83	3.64	0.2	5.9	8.26	6.07
Rucci_Rucci1	8.20	7.97	5.11	0.1	8.26	8.59	8.29
Sandia_adder_dcop_05	0.81	0.16	0.66	0.13	1.16	1.28	0.95
Sandia_adder_dcop_09	0.82	0.16	0.67	0.13	1.3	1.28	0.97
Schenk_IBMNA_c-26	2.41	0.41	1.87	0.2	3.67	3.1	2.25
Schenk_nlpkkt240	10.38	10.53	10.42	0.8	10.39	10.42	10.44
Schmid_thermal2	6.66	7.29	3.92	0.18	6.74	6.68	6.45
SNAP_cit-Patents	3.36	3.09	0.68	0.12	1.99	0.92	0.64
SNL_ASIC_320ks	6.65	5.83	4.26	0.13	8.36	15.13	13.64
SNL_ASIC_680k	2.51	2.43	2.24	0.14	3.53	3.94	4.11
SNL_MiniFE150	10.27	10.3	10.89	0.73	10.38	10.37	10.28
SNL_NALU_HeatCondEQS	9.93	9.86	5.48	0.71	8.37	8.29	9.91
TKK_tube2	36.49	9.92	10.31	1.2	49.36	46.4	36.71
vanHeukelum_cage10	8.84	2.17	6.1	0.37	13.46	11.72	8.18
vanHeukelum_cage13	9.69	9.23	8.04	0.37	10.15	8.72	8.16
vanHeukelum_cage14	8.43	8.5	6.72	0.48	8.32	7.55	8.48

Table 4.1: Benchmarked Sparse-Matrix Vector Product Calculation Speed in GFLOP/s from a Single Socket POWER8 Processor (KK=Kokkos, TPETRA=Original SpMV Kernel used in TPetra Solvers, MKL=Intel Math Kernel Library, OMP-STAT=OpenMP Static Parallelism, OMP-DYN=OpenMP Dynamic Parallelism, OMP-INSP=OpenMP with Non-Zero Load Balancing across Threads, KK-INSP=Kokkos with Non-Zero Load Balancing across Threads

Chapter 5

Results from Intel Xeon Phi 7250 (Knights Landing) Processor (Cache Memory Mode)

	KK	TPETRA	MKL	OMP-DYN	OMP-STAT	OMP-INS	KK-INS
Andrews_Andrews	9.9	5.84	3.56	0.37	12.34	12.51	10.7
ANSYS_Delor64K	18.29	8.43	5.05	0.29	24.77	17.080	13.01
Bai_olm2000	0.49	0.09	0.55	0.08	0.97	1	0.69
Bai_rbsb480	1.48	0.31	1.19	0.42	1.82	1.84	1.41
Boeing_bcsstk39	44.57	20.15	14.08	1.31	57.75	53.47	46.87
Boeing_bcsstm34	2.15	0.43	1.52	0.54	3	3.05	2.16
Boeing_crystm01	7.36	1.8	5.48	0.56	10.67	10.91	7.59
Boeing_crystm02	17.39	5.48	10.37	0.64	24.77	23.92	17.74
Boeing_crystm03	23.67	8.54	11.35	0.68	33.76	31.8	24.62
Boeing_ct20stif	41.28	20.39	12.77	1.51	46.05	46.75	36.09
Boeing_pwtk	38.6	37.82	16.52	1.6	45.89	50.35	39.32
Bourchtein_atmosmodl	19.26	18.52	10.03	0.2	27.73	24.97	19.01
Bova_rma10	31.03	18.11	9.97	1.48	43.43	46.76	27.89
Brethour_coater1	1.6	0.31	1.19	0.27	2.39	2.33	1.64
Brethour_coater2	13.29	3.51	8.31	0.6	18.63	18.26	13.2
DIMACS10_id2010	11.49	5.15	4.11	0.14	16.10	15.19	11.93
DIMACS10_italy_osm	6.7	8.47	2.34	0.06	7.46	7.47	6.71
DIMACS10_ne2010	12.55	6.78	4.39	0.14	16.72	16.67	12.75
DIMACS10_road_central	4.60	4.75	1.36	0.07	4.61	4.66	4.60
DIMACS10_vsp_vibrobox_scagr7-2c_rlfddd	2.64	3.55	1.05	0.33	2.97	9.95	9.22
DNVS_m_t1	43.56	36.32	15.23	2.97	46.4	51.56	48.41
DNVS_shipsec1	41.16	33.16	15.59	1.65	44.64	48.6	45.51
Dziekonski_dielFilterV2real	39.69	33.34	7.12	1.26	26.18	38.28	40.95
FIDAP_ex1	0.39	0.08	0.32	0.13	0.52	0.53	0.39
FIDAP_ex10	4.59	0.93	3.34	0.51	6.39	6.41	4.63
FIDAP_ex11	41	14.35	12.73	1.86	56.93	56.05	44.78
FIDAP_ex15	6.63	1.56	4.44	0.38	9.80	10.05	6.98
FIDAP_ex18	4.8	1.16	3.7	0.33	7.63	7.2	5.25
FIDAP_ex25	2.13	0.43	1.7	0.46	3.04	3.08	2.17
Fluorem_HV15R	52.58	51.4	15.14	4.24	48.73	51.65	52.29
Freescale_memchip	14.94	14.6	3.62	0.16	10.45	9.55	15.01
GHS_indef_blockqp1	1.59	0.92	1.4	0.31	1.63	6.72	7.44
GHS_indef_bmw3_2	37.18	36.80	14.93	1.49	45.35	46.02	37.26
GHS_indef_brainpc2	2.44	1.03	1.88	0.18	2.75	6.68	6.3
GHS_indef_ncvxqp7	11.93	5.07	4.10	0.19	15.18	11.46	8.70
GHS_psdef_audikw_1	43.85	47.66	6.27	2.47	23.53	45.98	49.31
GHS_psdef_bmwcra_1	37.63	38.00	16.33	2.14	48.96	51.04	38.61
GHS_psdef_inline_1	43.61	43.19	6.59	2.19	23.03	44.44	46.93
GHS_psdef_ldoor	46.83	45.28	14.21	1.47	42.82	48.98	47.07
GHS_psdef_s3dkq4m2	43.23	27.37	16.36	1.58	47.5	47.7	42.02
GHS_psdef_wathen100	19.91	7.22	9.9	0.45	28.06	26.3	19.99
GHS_psdef_wathen120	20.2	7.08	9.97	0.45	30.34	28.31	21.35
Grund_bayer04	7.75	2.25	5.18	0.22	11.98	5.79	4.34

Hamm_bcircuit	9.66	3.9	3.7	0.16	14.11	14.04	10.52
Hamm_hcircuit	8.94	4.98	2.73	0.14	11.39	11.91	8.69
HB_bcsstk02	0.4	0.08	0.22	0.16	0.47	0.57	0.41
HB_bcsstk13	6.36	1.45	4.46	0.91	9.54	9.54	6.88
HB_lnsp_131	0.03	0.01	0.03	0.01	0.06	0.06	0.05
HB_lnsp_511	0.2	0.04	0.2	0.06	0.35	0.36	0.25
HB_mbeacxc	3.4	0.8	2.54	1.21	4.34	5.33	3.59
HB_mbeaflw	3.39	0.8	2.53	1.22	4.34	5.35	3.61
HB_orani678	3.99	1.16	2.46	0.82	5.51	5.44	3.82
Hollinger_mark3jac020	3.9	0.81	2.54	0.17	5.57	5.72	4.05
IBM_dc1	1.79	1.21	1.38	0.19	1.65	3.36	3.56
IBM_EDA_ckt11752_dc_1	8.48	3.44	4.06	0.19	12.34	10	7.64
IBM_EDA_trans4	1.79	1.21	1.38	0.19	1.65	3.34	3.54
IPSO_TSC_OPF_300	5.98	3.46	4.76	2.26	6.39	26.43	20.6
Janna_Fault_639	45.22	44.41	16.03	1.35	50.17	49.98	45.53
Janna_Hook_1498	48.25	46.43	15.65	1.23	49.07	49.19	48.11
Janna_StocF-1465	30.12	33.6	11.82	0.43	34.80	11.12	25.88
JGD_Homology_m133-b3	8.71	6.04	3.73	0.12	13.74	13.4	10.22
JGD_Homology_n3c6-b7	3.95	0.75	2.88	0.21	5.6	5.41	4.03
JGD_Homology_n4c6-b6	16.42	7.64	7.1	0.21	23.29	23.08	17.44
JGD_Homology_shar_te2-b2	8.25	5.3	3.61	0.09	11.76	11.54	8.85
JGD_Kocay_Trec13	30.47	8.9	11.29	9.17	40.05	43.26	36.22
JGD_Relat_relat9	9.81	10.6	2.24	0.09	8.29	1.53	0.89
JGD_Trefethen_Trefethen_20000	21.54	7.34	9.81	0.79	30.03	29.98	24.02
Koutsovasilis_F1	32.85	32.62	4.39	2.36	15.32	33.27	37.20
LPnetlib_lp_osa_30	0.45	0.65	0.34	1.93	0.45	2.76	2.85
Meszaros_bas1lp	17.68	7.54	6.33	2.78	24.6	31.86	24.68
Meszaros_mod2	7.07	2.41	4.35	0.16	9.01	8.9	6.61
Meszaros_model4	3.69	0.76	2.64	0.65	5.2	4.77	3.64
Meszaros_stat96v1	33.58	8.99	10.97	2.53	44.58	44.34	34
Mittelmann_pds-80	6.24	4.32	2.51	0.21	6.87	7.01	6.61
Norris_torso2	18.13	11.08	8.13	0.26	27	25.65	19.95
Oberwolfach_bone010	52.17	51.89	16.82	2.19	52.2	53	52.19
PARSEC_SiNa	12.24	3.4	7.4	0.91	16.83	18.3	13.43
Rajat_rajat16	4.86	3.37	2.95	0.2	5.69	8.89	7.42
Rajat_rajat25	5.44	2.78	3.56	0.2	5.87	8.09	5.94
Rucci_Rucci1	11.38	12.67	5.08	0.11	14.64	14.35	11.47
Sandia_adder_dcop_05	0.81	0.17	0.66	0.13	1.27	1.29	0.95
Sandia_adder_dcop_09	0.73	0.16	0.67	0.13	1.3	1.3	0.96
Schenk_IBMNA_c-26	2.39	0.43	1.87	0.2	3.65	3.14	2.24
Schenk_nlpkkt240	26.89	27.04	13.98	0.83	26.65	26	26.94
Schmid_thermal2	10.8	14.92	4.06	0.21	12.85	12.45	11.53
SNAP_cit-Patents	5.11	4.86	0.64	0.13	2.27	1	0.65
SNL_ASIC_320ks	7.3	6.74	4.3	0.17	8.84	18.11	13.77
SNL_ASIC_680k	3.22	3.22	2.22	0.17	3.91	5.01	4.88
SNL_MiniFE150	47.72	45.05	14.65	0.8	49.05	49.37	47.04
SNL_NALU_HeatCondEQS	36.91	36.85	6.58	0.79	20.58	20.38	36.47
TKK_tube2	35.9	12.48	12.58	1.19	49.09	46.63	36.26
vanHeukelum_cage10	8.79	2.38	6.07	0.37	13.35	11.69	8.18
vanHeukelum_cage13	24.4	23.85	9.72	0.5	28.23	22.18	17.93
vanHeukelum_cage14	31.89	30	8.18	0.54	27	19.54	28.42
vanHeukelum_cage15	33.32	30.9	8.23	0.58	25.01	19.06	32.76
vanHeukelum_cage8	0.89	0.17	0.68	0.19	1.24	1.39	0.97
VanVelzen_Zd_Jac6	25.43	14.26	8.15	2.16	29.69	11.7	6.96
VDOL_kineticBatchReactor_7	3.89	1.21	3.19	0.28	5.31	5.28	3.8
Williams_cant	43.89	25.75	16.33	1.88	49.63	49.83	37.45
Williams_mc2depi	9.35	8.57	5.48	0.12	15.58	15.48	11.91
Zaoul_kkt_power	11.48	11.68	2.24	0.21	4.58	13.17	13.7

Table 5.1: Benchmarked Sparse-Matrix Vector Product Calculation Speed in GFLOP/s from a Single Socket POWER8 Processor (KK=Kokkos, TPETRA=Original SpMV Kernel used in TPetra Solvers, MKL=Intel Math Kernel Library, OMP-STAT=OpenMP Static Parallelism, OMP-DYN=OpenMP Dynamic Parallelism, OMP-INSP=OpenMP with Non-Zero Load Balancing across Threads, KK-INSP=Kokkos with Non-Zero Load Balancing across Threads

Chapter 6

Results from Intel Xeon Phi 7250 (Knights Landing) Processor (HBM Only)

	KK	TPETRA	MKL	OMP-DYN	OMP-STAT	OMP-INSP	KK-INSP
Andrews_Andrews	10.08	5.78	3.64	0.37	12.44	12.71	10.96
ANSYS_Delor64K	18.40	8.45	5.10	0.3	24.98	17.10	13.17
Bai_olm2000	0.49	0.09	0.55	0.08	0.95	0.99	0.7
Bai_rbsb480	1.48	0.3	1.18	0.42	1.82	1.84	1.41
Boeing_bcsstk39	44.39	20.28	14.26	1.32	57.2	55.24	47.39
Boeing_bcsstm34	2.16	0.43	1.53	0.55	3.01	3.05	2.17
Boeing_crystm01	7.57	1.81	5.4	0.56	10.92	10.86	7.58
Boeing_crystm02	17.53	5.44	10.4	0.64	24.78	23.92	17.69
Boeing_crystm03	23.61	8.5	11.41	0.68	34.01	32.18	24.35
Boeing_ct20stif	41.76	20.49	12.84	1.52	46.24	46.66	36.85
Boeing_pwtk	38.35	38.17	16.62	1.6	45.41	50.39	39.31
Bourchtein_atmosmodl	19.19	18.52	10.09	0.2	27.82	24.83	18.94
Bova_rma10	31.64	18.39	10.03	1.49	43.63	46.68	28.46
Brethour_coater1	1.59	0.31	1.2	0.27	2.36	2.1	1.64
Brethour_coater2	13.39	3.51	8.30	0.6	18.52	18.26	13.16
DIMACS10_id2010	11.57	5.27	4.12	0.14	16.06	15.3	12.1
DIMACS10_italy_osm	6.74	8.52	2.34	0.06	7.45	7.43	6.73
DIMACS10_ne2010	12.59	6.88	4.41	0.14	16.73	16.75	12.96
DIMACS10_road_central	4.58	4.75	1.36	0.07	4.60	4.63	4.60
DIMACS10_vsp_vibrobox_scagr7-2c_rlfddd	2.64	3.58	1.05	0.33	3.01	9.91	9.36
DNVS_m_t1	43.7	36.57	15.34	2.95	46.14	51.94	49.14
DNVS_shipsec1	42.03	33.24	15.67	1.66	45.17	48.98	46.3
Dziekonski_dielFilterV2real	39.45	33.23	7.04	1.26	26.13	38.27	41.04
FIDAP_ex1	0.39	0.08	0.32	0.13	0.53	0.53	0.38
FIDAP_ex10	4.52	0.93	3.39	0.52	6.28	6.28	4.58
FIDAP_ex11	40.46	14.35	12.83	1.84	55.86	56.21	45.05
FIDAP_ex15	6.6	1.57	4.57	0.39	9.97	10.18	7.11
FIDAP_ex18	4.78	1.15	3.7	0.33	7.68	7.16	5.26
FIDAP_ex25	1.88	0.43	1.71	0.46	2.87	3.11	2.17
Fluorem_HV15R	52.38	52.25	15.15	4.2	48.05	52.91	52.64
Freescale_memchip	14.89	14.55	3.56	0.16	10.44	9.61	14.95
GHS_indef_blockqp1	1.6	0.93	1.42	0.31	1.65	6.73	7.54
GHS_indef_bmw3_2	37.20	36.94	15.01	1.49	45.21	45.86	37.03
GHS_indef_brainpc2	2.43	1.02	1.87	0.18	2.71	6.53	6.28
GHS_indef_ncvxqp7	11.88	5.12	4.15	0.19	14.9	11.55	8.85
GHS_psdef_audikw_1	43.54	47.25	6.27	2.46	23.47	45.63	49
GHS_psdef_bmwcra_1	37.62	38.11	16.59	2.13	49.15	50.91	38.51
GHS_psdef_inline_1	43.5	43.2	6.62	2.20	23.04	44.28	46.92
GHS_psdef_ldoor	46.63	45.33	14.23	1.47	42.49	48.8	46.96
GHS_psdef_s3dkq4m2	43.76	26.98	16.54	1.58	47.86	47.34	42.45
GHS_psdef_wathen100	19.98	7.16	9.88	0.45	27.58	26.38	19.97
GHS_psdef_wathen120	20.27	7.08	10	0.45	30.49	28.55	21.62
Grund_bayer04	7.79	2.26	5.08	0.22	11.98	5.8	4.36

Hamm_bcircuit	9.55	3.91	3.81	0.16	14.17	14.08	10.6
Hamm_hcircuit	8.96	5.05	2.73	0.14	11.08	12	8.80
HB_bcsstk02	0.4	0.08	0.22	0.16	0.54	0.56	0.4
HB_bcsstk13	6.35	1.44	4.56	0.91	9.39	9.55	6.85
HB_lnsp_131	0.03	0.01	0.03	0.01	0.06	0.07	0.04
HB_lnsp_511	0.2	0.04	0.2	0.06	0.35	0.35	0.25
HB_mbeacxc	3.42	0.79	2.55	1.23	4.33	5.28	3.6
HB_mbeaflw	3.41	0.8	2.55	1.22	4.36	5.24	3.58
HB_orani678	4.06	1.19	2.49	0.82	5.52	5.43	3.84
Hollinger_mark3jac020	3.92	0.8	2.67	0.17	5.62	5.66	4.05
IBM_dc1	1.78	1.21	1.37	0.19	1.64	3.35	3.55
IBM_EDA_ckt11752_dc_1	8.43	3.36	4.02	0.19	12.2	10.06	7.73
IBM_EDA_trans4	1.79	1.21	1.37	0.19	1.63	3.39	3.57
IPSO_TSC_OPF_300	6.09	3.49	4.8	2.29	6.45	26.51	20.83
Janna_Fault_639	44.91	44.57	16.06	1.35	49.84	49.82	45.33
Janna_Hook_1498	47.71	46.46	15.68	1.22	49.6	50.21	47.54
Janna_StocF-1465	29.97	33.73	11.92	0.43	34.9	11.06	25.9
JGD_Homology_m133-b3	8.64	6.06	3.75	0.12	13.69	13.46	10.20
JGD_Homology_n3c6-b7	3.96	0.76	2.9	0.21	5.63	5.38	4.02
JGD_Homology_n4c6-b6	16.48	7.69	7.16	0.2	23.49	23.22	17.58
JGD_Homology_shar_te2-b2	8.28	5.28	3.66	0.08	11.9	11.71	9.01
JGD_Kocay_Trec13	30.74	8.92	11.38	9.13	40.35	43.63	36.12
JGD_Relat_relat9	9.84	10.63	2.25	0.09	8.28	1.53	0.88
JGD_Trefethen_Trefethen_20000	22.15	7.37	9.91	0.79	31.79	31.33	24.28
Koutsovasilis_F1	32.86	32.57	4.42	2.35	15.55	33.55	37.25
LPnetlib_lp_osa_30	0.44	0.66	0.34	1.93	0.46	2.77	2.86
Meszaros_bas1lp	17.61	7.58	6.36	2.8	24.39	32.07	24.72
Meszaros_mod2	7.11	2.37	4.38	0.16	9	8.9	6.62
Meszaros_model4	3.63	0.76	2.67	0.65	5.17	4.79	3.63
Meszaros_stat96v1	34.07	8.99	11.03	2.53	42.92	44.42	34
Mittelmann_pds-80	6.47	4.37	2.48	0.21	6.93	7.01	6.76
Norris_torso2	18.3	11.14	8.21	0.26	27.21	25.74	20.16
Oberwolfach_bone010	51.42	51.26	16.95	2.18	51.84	53.85	51.47
PARSEC_SiNa	12.25	3.38	7.32	0.91	16.89	18.22	13.13
Rajat_rajat16	4.96	3.39	2.95	0.2	5.74	8.80	7.48
Rajat_rajat25	5.45	2.79	3.6	0.2	5.86	8.14	5.94
Rucci_Rucci1	11.49	12.7	5.09	0.11	14.68	14.44	11.59
Sandia_adder_dcop_05	0.81	0.17	0.67	0.13	1.29	1.27	0.95
Sandia_adder_dcop_09	0.74	0.17	0.67	0.13	1.31	1.3	0.98
Schenk_IBMNA_c-26	2.4	0.43	1.85	0.2	3.69	3.14	2.24
Schmid_thermal2	10.89	14.77	4.09	0.21	12.99	12.71	11.42
SNAP_cit-Patents	5.18	4.88	0.64	0.13	2.32	1	0.65
SNL_ASIC_320ks	7.36	6.74	4.34	0.17	8.86	18.11	13.99
SNL_ASIC_680k	3.23	3.21	2.25	0.17	4	5.01	4.85
SNL_MiniFE150	47.53	45.1	14.6	0.8	50.76	49.23	47.2
SNL_NALU_HeatCondEQS	36.78	36.83	6.53	0.8	20.18	20.12	36.16
TKK_tube2	36.72	12.48	12.8	1.19	49.72	46.4	36.91
vanHeukelum_cage10	8.74	2.37	6.09	0.37	13.42	11.75	8.21
vanHeukelum_cage13	24.47	23.87	9.78	0.5	28.11	22.01	18.01
vanHeukelum_cage14	31.8	30.04	8.22	0.54	27.27	19.55	28.19

Table 6.1: Benchmarked Sparse-Matrix Vector Product Calculation Speed in GFLOP/s from a Single Socket POWER8 Processor (KK=Kokkos, TPETRA=Original SpMV Kernel used in TPetra Solvers, MKL=Intel Math Kernel Library, OMP-STAT=OpenMP Static Parallelism, OMP-DYN=OpenMP Dynamic Parallelism, OMP-INSP=OpenMP with Non-Zero Load Balancing across Threads, KK-INSP=Kokkos with Non-Zero Load Balancing across Threads

Chapter 7

Results from IBM POWER8

	KK	OMP-STAT	OMP-DYN	OMP-INSP	KK-INSP
Andrews_Andrews	7.98	12.83	0.27	13.12	8.97
ANSYS_Delor64K	10.99	13.68	0.21	12.39	7.04
Bai_olm2000	1.18	2.37	0.08	2.52	1.44
Bai_rbsb480	3.25	4.81	0.64	4.92	3.12
Boeing_bcsstk39	17.94	18.02	0.97	20.3	17.14
Boeing_bcsstm34	3.91	6.33	0.77	6.33	4.14
Boeing_crystm01	9.84	12.89	0.44	13.53	9.41
Boeing_crystm02	13.28	16.08	0.49	17.32	12.39
Boeing_crystm03	13.67	17.11	0.5	18.10	12.53
Boeing_ct20stif	10.1	15.95	1.11	20.09	17.63
Boeing_pwtk	11.74	11.54	1.21	11.86	11.87
Bourchtein_atmosmodl	8.18	9.27	0.141	9.24	7.95
Bova_rma10	11.35	11.67	1.11	19.26	16.08
Brethour_coater1	2.92	5.12	0.28	5.11	3.17
Brethour_coater2	12.72	15.33	0.45	15.77	11.29
DIMACS10_id2010	5.29	11.68	0.1	12.05	6.69
DIMACS10_italy_osm	4.44	4.83	0.04	4.94	4.36
DIMACS10_ne2010	7.8	12.03	0.1	12.01	6.6
DIMACS10_road_central	2.42	2.4	0.05	2.69	2.54
DIMACS10_vsp_vibrobox_scagr7-2c_rlfddd	1.73	1.94	0.24	9.30	5.6
DNVS_m_t1	11.13	11.43	2.33	12.31	11.66
DNVS_shipsec1	12.18	12.09	1.26	13	12.69
Dziekonski_dielFilterV2real	11.12	8.35	0.95	10.82	11.34
FIDAP_ex1	0.83	1.49	0.3	1.43	0.91
FIDAP_ex10	6.35	9.24	0.47	9.62	6.74
FIDAP_ex11	14.09	18.15	1.46	20.01	17.58
FIDAP_ex15	8.92	12.69	0.3	12.8	8.1
FIDAP_ex18	7.1	11.08	0.26	11.21	6.66
FIDAP_ex25	3.8	5.85	0.56	6.23	3.81
Fluorem_HV15R	11.41	10.68	3.39	11.29	11.48
Freescale_memchip	6.78	6.43	0.12	5.92	6.85
GHS_indef_blockqp1	0.64	0.63	0.23	4.60	3.28
GHS_indef_bmw3_2	11.75	11.42	1.12	11.78	11.87
GHS_indef_brainpc2	0.94	1.01	0.14	3.92	3.54
GHS_indef_ncvxqp7	7.34	9.61	0.14	8.72	6.23
GHS_psdef_audikw_1	11.43	6.58	1.91	11.09	11.64
GHS_psdef_bmwcra_1	11.98	11.64	1.63	12.12	12.06
GHS_psdef_inline_1	11.83	6.82	1.69	11.34	11.91
GHS_psdef_ldoor	11.28	10.47	1.11	11.36	11.4
GHS_psdef_s3dkq4m2	15.65	20.27	1.19	20.54	17.54
GHS_psdef_wathen100	13.38	16.76	0.33	17.03	11.81
GHS_psdef_wathen120	10.47	17.08	0.33	17.31	12.17
Grund_bayer04	8.16	11.45	0.16	10.59	5.53
Hamm_bcircuit	6.3	10.35	0.11	11.92	6.19
Hamm_hcircuit	4.90	6.87	0.1	12.04	6.24
HB_bcsstk02	0.85	1.35	0.57	1.45	0.9
HB_bcsstk13	7.99	10.27	0.87	12.4	8.87
HB_lnsp_131	0.09	0.19	0.04	0.2	0.11
HB_lnsp_511	0.45	0.96	0.09	0.92	0.57
HB_mbeacxc	3.6	5.14	1.91	8.97	6.63

HB_mbeaflw	3.61	5.12	1.9	8.98	6.59
HB_orani678	1.81	2.52	0.75	6.28	3.69
Hollinger_mark3jac020	5.36	8.88	0.13	8.89	5.12
IBM_dc1	0.98	1.07	0.14	1.79	1.39
IBM_EDA_ckt11752_dc_1	3.1	3.52	0.14	10.41	5.49
IBM_EDA_trans4	0.98	1.07	0.14	1.83	1.39
IPSO_TSC_OPF_300	2.30	2.35	1.85	16.23	13.12
Janna_Fault_639	11.65	11.64	1.02	11.58	11.71
Janna_Hook_1498	11.48	11.22	0.92	11.24	11.47
Janna_StocF-1465	10.17	10.38	0.31	9.42	9.28
JGD_Homology_m133-b3	6.87	10.83	0.08	10.76	5.76
JGD_Homology_n3c6-b7	5.24	9.11	0.17	9.17	5.27
JGD_Homology_n4c6-b6	7.1	15.11	0.15	14.99	8.41
JGD_Homology_shar_te2-b2	4.85	9.29	0.06	10.05	6.31
JGD_Kocay_Trec13	10.7	11.61	14.38	19.07	18.65
JGD_Relat_relat9	4.66	5.54	0.06	1.42	0.65
JGD_Trefethen_Trefethen_20000	11.76	16.87	0.61	17.49	13.56
Koutsovasilis_F1	11.39	5.79	1.85	11.2	11.88
LPnetlib_lp_osa_30	0.31	0.33	1.25	2.33	2
Meszaros_bas1lp	4.81	5.02	2.53	17.23	12.53
Meszaros_mod2	5.33	6.6	0.12	8.78	5.9
Meszaros_model4	4.08	4.98	0.7	8.28	5.16
Meszaros_stat96v1	12.34	18.83	2.28	19.23	17.26
Mittelmann_pds-80	8.04	9.81	0.15	10.17	6.89
Norris_torso2	11.09	16.54	0.19	16.45	10.19
Oberwolfach_bone010	11.77	11.64	1.67	11.64	11.79
PARSEC_SiNa	6.43	8.13	0.75	15.96	11.9
Rajat_rajat16	3.52	3.25	0.14	4.29	4.19
Rajat_rajat25	3.65	3.92	0.15	5.42	3.37
Rucci_Rucci1	7.16	9.61	0.08	9.62	5.79
Sandia_adder_dcop_05	1.66	1.61	0.12	1.81	1.45
Sandia_adder_dcop_09	1.69	1.56	0.12	1.56	1.39
Schenk_IBMNA_c-26	2.44	4.17	0.17	6.23	3.27
Schenk_nlpkkt240	10.59	10.42	0.62	10.39	10.56
Schmid_thermal2	7.55	7.52	0.15	7.26	6.52
SNAP_cit-Patents	1.95	1.28	0.09	1.43	0.58
SNL_ASIC_320ks	8.16	11.36	0.12	13.79	8.27
SNL_ASIC_680k	1.65	1.67	0.12	2.47	1.81
SNL_MiniFE150	10.94	10.98	0.59	10.8	11.06
SNL_NALU_HeatCondEQS	10.4	7.56	0.59	7.21	10.49
TKK_tube2	15.87	18.02	0.92	19.63	16.60
vanHeukelum_cage10	7.68	11.06	0.28	12.52	6.6
vanHeukelum_cage13	9.26	9.75	0.37	9.95	8.44
vanHeukelum_cage14	10.05	8.89	0.4	9.15	10.07

Table 7.1: Benchmarked Sparse-Matrix Vector Product Calculation Speed in GFLOP/s from a Single Socket POWER8 Processor (KK=Kokkos, OMP-STAT=OpenMP Static Parallelism, OMP-DYN=OpenMP Dynamic Parallelism, OMP-INSP=OpenMP with Non-Zero Load Balancing across Threads, KK-INSP=Kokkos with Non-Zero Load Balancing across Threads

References

[1] Christopher G Baker and Michael A Heroux. Tpetra, and the Use of Generic Programming in Scientific Computing. *Scientific Programming*, 20(2):115–128, 2012.

[2] Leonardo Dagum and Ramesh Menon. OpenMP: an Industry Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[3] Timothy A Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[4] Eduardo F DAzevedo, Mark R Fahey, and Richard T Mills. Vectorized Sparse Matrix Multiply for Compressed Row Storage Format. In *International Conference on Computational Science*, pages 99–106. Springer, 2005.

[5] H Carter Edwards, Daniel Sunderland, Vicki Porter, Chris Amsler, and Sam Mish. Manycore Performance-Portability: Kokkos Multidimensional Array Library. *Scientific Programming*, 20(2):89–114, 2012.

[6] Michael A Heroux, Roscoe A Bartlett, Vicki E Howle, Robert J Hoekstra, Jonathan J Hu, Tamara G Kolda, Richard B Lehoucq, Kevin R Long, Roger P Pawlowski, Eric T Phipps, et al. An Overview of the Trilinos Project. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):397–423, 2005.

[7] Tomáš Oberhuber, Atsushi Suzuki, and Jan Vacata. New Row-grouped CSR Format for Storing the Sparse Matrices on GPU with Implementation in CUDA. *arXiv preprint arXiv:1012.2270*, 2010.

[8] Sivan Toledo. Improving the Memory-system Performance of Sparse-Matrix Vector Multiplication. *IBM Journal of research and development*, 41(6):711–725, 1997.

[9] Richard W Vuduc and Hyun-Jin Moon. Fast Sparse Matrix-Vector Multiplication by Exploiting Variable Block Structure. In *International Conference on High Performance Computing and Communications*, pages 807–816. Springer, 2005.

DISTRIBUTION:

1	MS 1319	Michael A. Heroux, 01400
1	MS 1319	Simon D. Hammond, 01422
1	MS 1319	Robert J. Hoekstra, 01422
1	MS 1319	Erik O. Strack, 01426
1	MS 1319	Christian R. Trott, 01426
1	MS 0845	Kendall H. Pierson, 01542
1	MS 0845	Michael W. Glass, 01545
1	MS 1079	Michael Holmes, 05210
1	MS 1071	Reno Sanchez, 05250
1	MS 0845	Technical Library, 09536
1	MS 0899	Technical Library, 9536 (electronic copy)

Sandia National Laboratories