Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Energy-efficient Sparse Matrix Auto-tuning with CSX

Jan Christian Meyer[a,*] , Lasse Natvig[b], Vasileios Karakasis, Dimitris Siakavaras, and Konstantinos Nikas[c]

[a]*High Performance Computing Section, IT Dept., NTNU, Norway*
[b]*Dept. of Computer and Information Science (IDI), NTNU, Norway*
[c] *School of ECE, NTUA, Greece*

Abstract

This whitepaper describes the programming techniques used to develop an auto-tuning compression scheme for sparse matrices with respect to accelerating matrix-vector multiplication and minimizing its energy footprint, as well as a method for extracting a power profile from a corresponding implementation of the conjugate gradient method. Using two example systems, we show how these techniques can be leveraged to automatically detect a non-trivial local optimum in the execution parameter space, suggesting that it is feasible to integrate the energy efficiency evaluation of the automatic adaptation with the automatic tuning process.

1. Introduction

This whitepaper describes the programming techniques used to develop an auto-tuning compression scheme for sparse matrices with respect to accelerating matrix-vector multiplication and minimizing its energy footprint, as well as a method for extracting a power profile from a corresponding implementation of the conjugate gradient method. Using two example systems, we show how these techniques can be leveraged to automatically detect a non-trivial local optimum in the execution parameter space, suggesting that energy efficiency evaluation of the automatic adaptation is feasible to integrate with the automatic tuning process.

2. Methods

2.1. Auto-tuned matrix compression

The most widely used method for storing sparse matrices is the Compressed Sparse Row (CSR) format. This format stores a sparse matrix using three arrays: (a) an array storing the non-zero values of the matrix, (b) an array of equal size storing the corresponding column indices, and (c) an array of row pointers, pointing to the start of each row. While being relatively compact, CSR storage contains a lot of redundant information in the column index storage. The contention for memory bandwidth resources is the key performance problem of the sparse matrix-vector kernel in modern multicore architectures, making minimization of the matrix representation size most important for the optimization of this kernel.

The most successful format toward the direct compression of the matrix representation is the Compressed Sparse eXtended format [5]. CSX employs explicit compression techniques and exploits non-zero elements substructures inside the matrix, in order to minimize the column index information of the original CSR format. CSX is able to detect a variety of non-zero elements substructures, including horizontal, vertical, diagonal, anti-diagonal and two-

* Corresponding author. *E-mail address*: Jan.Christian.Meyer@ntnu.no.

dimensional (blocks) ones. Thanks to its advanced detection mechanism and the compact representation of the encoded substructures, CSX is able to compress the matrix memory footprint significantly – reaching the theoretical maximum in many cases. This leads to important performance improvements in both SMP and NUMA multicore architectures.

In CSX, the sparse matrix is organized in *units* of encoded non-zero elements. A unit can either be a *substructure unit*, i.e., a sequence of non-zero elements forming a substructure, or a *delta unit*, i.e., a unit of stray elements, not in a substructure, but encoded with a delta indexing scheme. For each unit, CSX keeps only a two-byte descriptor and its initial column index, encoded as a delta distance from the previous one and stored in a variable size integer. If the unit is a substructure, no further information is stored, otherwise, if it is a delta unit, the delta encoded values of the unit's non-zero elements are stored immediately afterwards. Since the exact substructure instantiations (e.g., blocks 2x3, 5x7, etc.) are a priori unknown, CSX generates substructure-specific code at the runtime using the LLVM [6] framework. This technique not only allows high-performance matrix-specific matrix-vector implementations, but also, in conjunction with the powerful CSX's substructure detection mechanism, offers a high degree of flexibility, since CSX is able to combine high performance and high compression ratios of the matrix.

Substructure detection in CSX employs an auto-tuning logic with respect to the size of the final matrix representation and the overall decompression cost at the runtime. The detection process proceeds in a greedy fashion in multiple steps. Starting from the original unencoded matrix, CSX tries successive encodings for all the supported substructure types and collects statistics, reflecting the matrix compression and the decompression overhead. At this step, CSX selects the best substructure type, encodes matching sub-matrices, and repeats the detection process for the rest of the matrix. The detection phase finishes when no more substructures can be encoded. The fitness metric for selecting a substructure type for encoding depends on the underlying architecture; for example, in SMP systems, the metric depends solely on a prediction of the matrix size reduction, while in NUMA systems, where the computational part of the kernel is more exposed, the fitness metric tries also to minimize the total number of encoded substructure instantiations, as a matter of decreasing the branch instructions in the critical path. Towards the same direction, CSX relaxes its compression scheme depending on the underlying architecture, in order to minimize the decompression cost.

In this whitepaper, we employ the CSX format in the execution of the CG iterative solution algorithm (we are using the implementation supplied with the CSX software). CSX offers a mechanism for reducing considerably the matrix preprocessing cost, by using statistical sampling of the input matrix; however, we do not employ this mechanism in this preliminary examination. The CG execution using the CSX format consists of the following five phases:

1. Loading of the matrix from the disk (single-threaded)
2. Substructure detection (multithreaded)
3. Matrix encoding (multithreaded)
4. Code generation (single-threaded)
5. Sparse matrix-vector kernel (multithreaded)

Phases 1-4 occur once at the initialization of the algorithm, while the matrix-vector kernel persists in every iteration of the algorithm. When using CSR, only phases 1 and 5 are present.

2.2. Hardware platform

All experiments are carried out on a quad-core desktop computer with Hyperthreading capabilities. It contains an Intel® Core i7-2600 Sandy Bridge multiprocessor, clocked at its maximum frequency of 3.4GHz. It has 16GB of main memory and a shared level3 cache of 8MB. This configuration means that tested matrices fit in main memory, but not in the last-level cache, thus emphasizing the effect of compression schemes on the resulting memory traffic.

2.3. Power instrumentation

Power instrumentation is based on Model Specific Registers (MSRs) first featured in the Intel Sandy Bridge architecture and expands upon the technique described in [1]. The programming interface to this approach takes the form of a library featuring a data structure that tracks the difference of processor package energy consumption, using privileged access to device-files which reflect MSR register state, allowing the difference of energy consumed to be computed between two calls from the application. While this is sufficient to support the instrumentation of CPU energy use for short benchmark sections [2], the requirements of distinguishing between stages of execution with CSX creates three technical challenges:

- Capturing a running estimate of application power use
- Minimizing interference from the instrumentation code, and unrelated program execution

- Identifying program points on the time scale of the resulting traces of time and power

Capturing a running estimate of application energy use is subject to a 1 ms update frequency in the energy performance counters in the target architecture, constraining both the obtainable accuracy and the maximal interval of a single sample, as the energy counter registers wrap around [3]. A further constraint is posed by accessing them through the use of an operating system device file, since it introduces the running kernel as a potential source of inaccuracy. An important consideration to our approach is that while the processor tracing energy use is also executing the application program, the accuracy lost in lowering the sampling frequency is a trade-off against the introduced interference from the sampling itself.

Single sample capture is implemented as a stateful C function, containing two statically allocated structures for use by the energy counter library and two timer variables. All timings are captured using the *gettimeofday* system call, which reports a wall clock measured to microsecond resolution. Instrumentation begins with including the initialization of one of the structures in the initialization of the measured program, initializing a log file, and recording time. The capture itself consists of calling the power library to finalize the measurement started in initialization, copying out the values of the structure at the time it stopped, recording time, and starting measurement again. Having obtained an energy difference and a time difference, power for the interval can then be estimated discretely as (ΔE / Δt), which is logged along with the end time of the sample interval. The total data volume involved in this operation is below 0.5 kB and its execution time negligible in comparison with the finest attainable energy estimate resolution. The logic for finalizing the structures and closing the log file is contained in a function which is registered with the *atexit* function at initialization time, making it unnecessary to modify the application beyond adding to its initialization.

Continuous sampling is implemented by registering the sample capture function as a signal handler for the Unix SIGPROF signal, which an application can request to have periodically issued from the O/S kernel, calling the POSIX *setitimer* function at initialization. Initial experiments with the granularity suggested that stable and reproducible results are attainable using a 100 ms interval, making each sample account for approximately 100 updates of the energy registers. As there is an element of non-determinism in both signal delivery and energy capture mechanisms, trial runs comparing a running ideal sum of timestamps to the wall clock estimate. The aggregate drift of these for runs of up to 105 seconds proved to be on the 10 ms scale, or 10% of a single sample interval, suggesting that the deviation due to overhead lies well below 1ms per sample. Unfortunate timing can at most misrepresent energy by 1 sample point for a given interval, bounding its energy estimate deviation to 1% of the state reflected in the hardware register. Sampling further intervals transfers the recording of the outstanding amount to the next sample, so differences between measured and actual energy consumption is restricted to first and last samples in a sequence. As the measured stages of execution encompass sums of tens to hundreds of samples, the error in the total energy estimate becomes negligible. Refining the sampling interval to a length where a single sample point accounts for a larger amount of the recorded energy displays the effect quite clearly: unfortunately timed samples appear as sharp under-estimates, immediately followed by an equal over-estimate in the following sample. Because this effect is an artifact of the interaction between sample sizes and minute drift from the wall clock, we note that at the 100 ms sample scale, the magnitude of the noise is small enough to provide detailed views of how power develops with program execution, and successfully admits runs hundreds of seconds without overflowing the energy counters. The impact of software overhead is not investigated further.

Interference from unrelated system activity is minimized internally by allocating all required data structures statically, producing a constant memory requirement except for the log file, which has a natural linear dependence on time. In order to eliminate uncertainty caused by flushes of the file buffer, this buffer was replaced with a static, fixed-size allocation of 5MB committed using the POSIX *setbuffer* call, which sufficed for our experiments. Elimination of interference from unrelated programs was attempted using the Linux real-time priority scheduling mechanism to remove everything aside from unmaskable interrupts. This led to an effect where two consecutive samples would make underestimates from one sample, and proportional over-estimates on the next. Attributing this effect to the update of the MSR device file being postponed from one sample interval to the next, we opt instead for detecting significant interference by large anomalies in the power/time graph; as results make it markedly visible when several execution threads have been active over an interval, it is reasonable to expect that external program interference would cause a similar effect, visible when the measured program is known to run at steady states for extended periods.

With the establishment of IPC signal handling as a trigger for log file events, timing of events in the program logic is a straightforward extension, accomplished by registering signal handlers for SIGUSR1 and SIGUSR2 which only record wall time according to *gettimeofday* in a separate log file. Instrumentation of the remainder of the code is thus decoupled from the state of the handler functions, and can be easily instrumented using calls to the *getpid* and *kill* functions. Program event and power logs are thus recorded on the same time scale, permitting them to be combined in a post-processing step which shows how they coincided after the run is complete.

3. Results

In combination, the described techniques provide power/time data which relates the known stages of CSX execution with a trace of its power consumption, permitting the balance between the energy cost of matrix preprocessing to be evaluated with respect to its impact on matrix-vector multiplication.

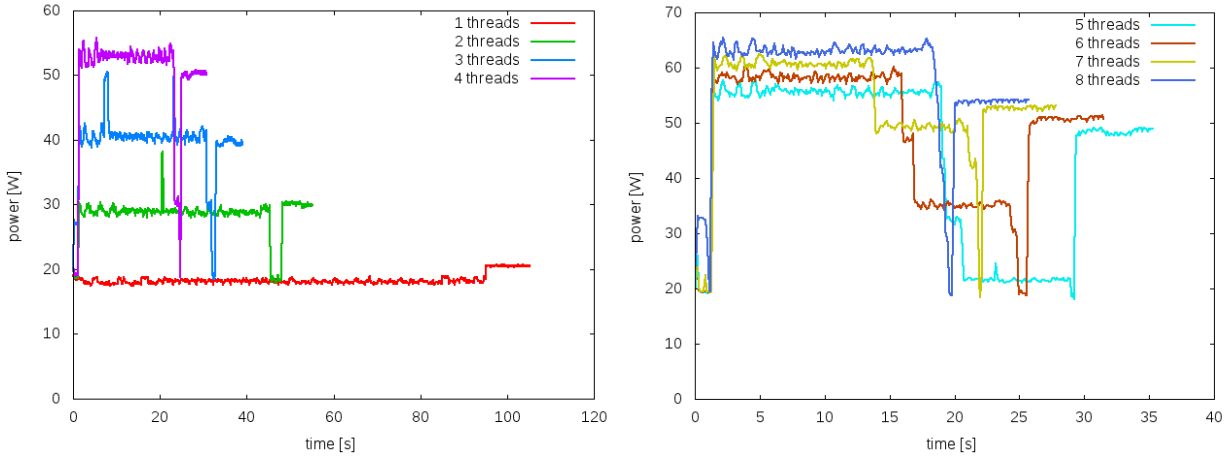

Fig. 1. (a) CSX matrix compression, with 1-4 threads; (b) CSX matrix compression with 5-8 threads and Hyperthreading

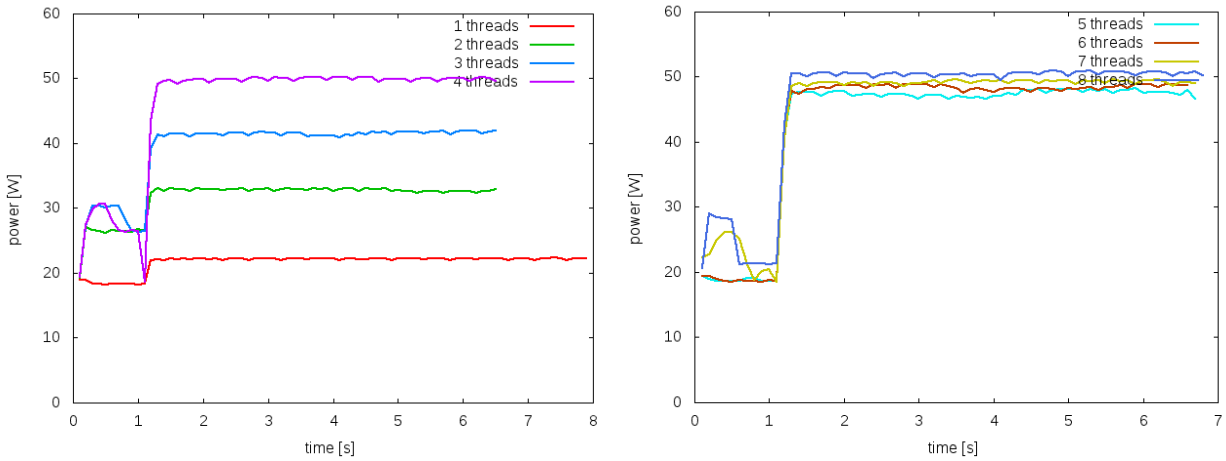

Fig.2. (a) CSR matrix compression, with 1-4 threads; (b) CSR matrix compression with 5-8 threads and Hyperthreading

As shown in Figs. 1 and 2, the initialization, preprocessing and iteration stages each display their own steady and characteristic power requirements.

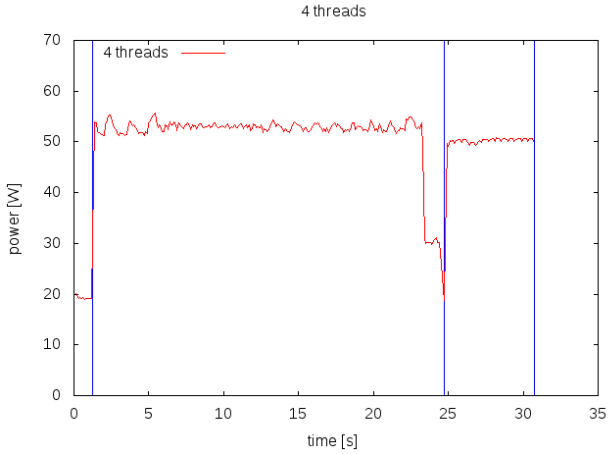

Fig.3. Detailed view of CSX matrix compression with 4 threads, including program events/stages

Merging these with the recorded transitions between stages yields a detailed view as the 4-thread CSX case displayed in Fig. 3. The vertical, blue lines denote the transitions from stage 1 to 2, stage 4 to 5, and program termination. The figure does not discriminate between the intermediate preprocessing stages 2, 3, and 4, as their aggregate cost is of primary interest.

Recognizing the structure of the stages from Fig. 3 in Figs. 1 and 2, characteristics of program behavior can already be extracted by visual inspection. It is clear that the preprocessing stage of CSX compression is quite amenable to parallelization, providing time improvements up to 7 threads, utilizing Hyperthreads at minor additional power, while CSR shows no significant benefit from Hyperthreading. The figures clearly display that a constant iteration count of 1024 for stage 5 is sufficient to bring power to a steady state, making it feasible to project that energy effects observed at this scale are representative of the sustained power consumption if iterations were continued until convergence. This can enable automatic selection of the energy-optimal configuration from collecting small sample run data prior to execution, as the iteration counts for very large systems can require them to run for significantly longer than these tests.

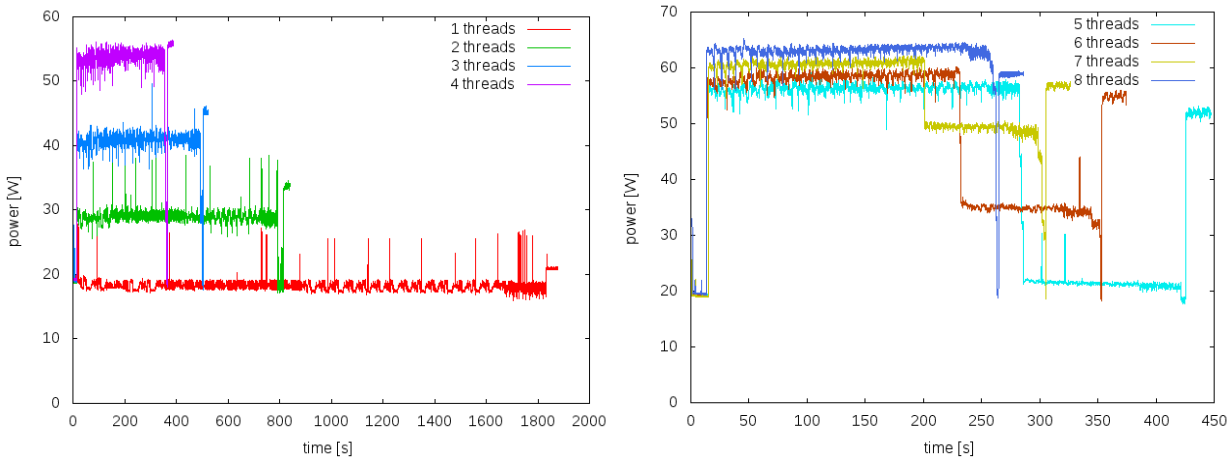

Fig. 4. (a) CSX matrix compression, with 1-4 threads; (b) CSX matrix compression with 5-8 threads and Hyperthreading

We may note that the *parabolic_fem* [4] matrix which gave the results displayed in Figs. 1 and 2 has an irregular structure which makes it poorly suited to provide improved benefits to iteration speed for either format: beyond two threads, the length of the iteration stage does not decrease significantly in spite of the increases in power. The *boneS10* [4] matrix has a structure which is more amenable to compression; the size of this system makes the distinctions between stages less visible from power/time graphs. Fig. 4 shows measured CSX results in full detail, while further tabulated results tabulate values obtained from integrating these graphs using the trapezoid method. As

the collected power estimates are piecewise linear, integrals are exact to the resolution of the samples displayed in the graphs. Adjusting the interval of integration by using the event timings, execution characteristics are tabulated by stages in Tab. 1 for *parabolic_fem*, and Tab.2 for *boneS10*.

Tab. 1 shows that the net cost of the preprocessing stage reduces with parallelism, while time and energy per iteration increases. The characteristics of the *parabolic FEM* system show that CPU energy consumption will be unilaterally lowest for single-threaded CSR, but also reveal that there are non-linearities in the relationship between the increases in per-iteration energy vs. the cost of the CSX auto-tuning: the first two Hyperthreads mark an area where the per-iteration cost is slightly lowered, at an increased preprocessing cost.

Tab. 2 shows the same tendency of the preprocessing stage, and growing efficiency for CSX with increasing thread count. Single-threaded execution favors CSR, while multithreaded execution amortizes the additional preprocessing cost in $15759.618/(1.434 - 1.225) \approx 75404$ iterations for two threads, or less. Considering that the *boneS10* system is expressed in 914898 dimensions gives that as an upper bound on the number of conjugate gradient iterations to solution, suggesting that the overhead of preprocessing is worthwhile in every other case for this system. However, in practice, where a pre-conditioner might be used for the CG method, the 75K iterations is a quite large number. Enabling the fast preprocessing mode of CSX (sampling of the matrix and/or targeted encoding of specific substructures) will allow an order of magnitude faster amortization [5].

As witnessed by these results, our described instrumentation is capable of capturing and estimating the energy requirements for a given input set in a fraction of the system's time to solution, suggesting that it can form the basis of an automatic tuning mechanism to select optimal configurations at solver startup time by sampling available alternatives.

Table 1. Energy consumption and time by execution stages, *parabolic_fem*

Thread #	CSX init [J]	CSX preproc [J]	CSX E/iter [J]	CSX t/iter [s]	CSR init [J]	CSR E/iter [J]	CSR t/iter [s]
1	20.396	1699.677	0.201	0.00991	16.566	0.145	0.00671
2	20.771	1327.642	0.206	0.00701	23.522	0.170	0.00532
3	29.543	1253.750	0.243	0.00622	25.644	0.215	0.00530
4	21.202	1205.862	0.289	0.00586	24.909	0.258	0.00531
5	22.013	1224.329	0.283	0.00601	16.951	0.250	0.00544
6	22.033	1187.789	0.287	0.00578	16.929	0.255	0.00543
7	24.379	1147.162	0.292	0.00565	20.653	0.264	0.00550
8	32.886	1134.643	0.304	0.00577	22.007	0.276	0.00559

Table 2. Energy consumption and time by execution stages, *boneS10*

Thread #	CSX init [J]	CSX preproc [J]	CSX E/iter [J]	CSX t/iter [s]	CSR init [J]	CSR E/iter [J]	CSR t/iter [s]
1	266.460	33080.091	0.861	0.0411	220.760	0.831	0.0365
2	250.736	22993.266	0.794	0.0236	227.315	0.880	0.0254
3	260.820	19897.076	0.916	0.0204	216.183	1.109	0.0249
4	263.964	18693.004	1.129	0.0203	218.891	1.346	0.0251
5	293.853	18143.053	1.114	0.0216	221.781	1.322	0.0260
6	262.536	16937.524	1.127	0.0207	227.038	1.329	0.0257
7	295.292	16335.871	1.172	0.0208	223.161	1.376	0.0259
8	279.543	15759.618	1.225	0.0209	232.604	1.434	0.0262

4. Conclusions and future work

We have described the stages of execution for a program which compresses sparse matrices to automatically optimize the memory access patterns resulting from sparse matrix-vector multiplication, and a method to instrument it which captures the energy consumption related to its main phases of execution. As examples of how the parameters of the compression interact with matrix properties, we have shown a difference in the characteristics of a conjugate gradient solver using the compression method, suggesting that the energy efficiency of applying compression methods differ between input data sets, making the identification of the most suitable parameter sets candidate for further run-time automation. This technique has been applied to investigate the impact of variable clock frequency, degree of parallelism, compression methods, and a greater range of matrices, to understand the energy implications of software/hardware interactions in this parameter space. The difference in energy/iteration cost favors compression for large, regular matrices. For a description of the full study, the interested reader can refer to our corresponding conference paper [7].

The sampling preprocessing feature of CSX has not been applied in the presented results, but is expected to significantly reduce the substructure detection stage of the preprocessing phase, for a reduction in its energy cost. Validating this expectation makes an interesting direction for future work.

Acknowledgements

This work was financially supported by the PRACE-2IP project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-283 493.

References

1. J. C. Meyer, L. Natvig, Power instrumentation of task-based applications using model-specific registers on the Sandy Bridge architecture [PRACE White paper]

2. H. Lien, L. Natvig, A. Al Hasib, J. C. Meyer, Case Studies of Multi-core Energy Efficiency in Task Based Programs, LNCS Vol 7453 pp. 44-54, Springer 2012

3. Intel Corp. Intel® 64 and IA-32 Architectures Software Developer Manual., online documentation (accessed 18.01.2013), available at http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

4. The University of Florida Sparse Matrix Collection, online data set collection (accessed 18.01.2013) available at http://www.cise.ufl.edu/research/sparse/matrices

5. V. Karakasis, T. Gkountouvas, K. Kourtis, G. Goumas, N. Koziris, An Extended Compression Format for the Optimization of Sparse Matrix-Vector Multiplication, IEEE Transactions on Parallel and Distributed Systems, To Appear.

6. C. Lattner, V. Adve, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, 2004 Intl. Symposium on Code Generation and Optimization (CGO'04)

7. J. C. Meyer, V. Karakasis, J. Cebrian, L. Natvig, D. Siakavaras, K. Nikas, Energy-efficient Sparse Matrix Autotuning with CSX – A Trade-off Study, The Ninth Workshop on High-Performance, Power-Aware Computing 2013, at The 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS'13). (to appear)