Energy Auto-tuning using Polyhedral Approach

Wei Wang, John Cavazos
University of Delaware
101 Smith Hall
Newark, DE 19702
weiwang,cavazos@udel.edu

Allan Porterfield
University of North Carolina - RENCI
100 Europa Dr
Chapel Hill, NC 27517
akp@renci.org

ABSTRACT

As the HPC community moves into the exascale computing era, application energy has become a big concern. Tuning for energy will be essential in the effort to overcome the limited power envelope. How is tuning for lower energy related to tuning for faster execution? Understanding that relationship can guide both performance and energy tuning for exascale. In this paper, a strong correlation is presented between the two that allows tuning for execution to be used as a proxy for energy tuning. We also show that polyhedral compilers can effectively tune a realistic application for both time and energy.

For a large number of variants of the Polybench programs and LULESH energy consumption is strongly correlated with total execution time. Optimizations can increase the power and energy required between variants, but variant with minimum execution time also has the lowest energy usage. The polyhedral framework was also used to optimize a 2D cardiac wave propagation simulation application. Various loop optimizations including fusion, tiling, vectorization, and auto-parallelization, achieved a 20% speedup over the baseline OpenMP implementation, with an equivalent reduction in energy on an Intel Sandy Bridge system. On an Intel Xeon Phi system, improvements as high as 21% in execution time and 19% reduction in energy are obtained.

1. INTRODUCTION

As the HPC community approaches the exascale computing era, reducing application energy and power consumption is important. The cost of supplying the peak power and operational energy for exascale systems will be substantial. Controlling the application energy use of the increasingly powerful compute nodes will be required. Before tuning an application for power efficiency, it is necessary to understand how energy consumption is related to the application execution time. Knowledge of the relationship between performance and energy can guide the tuning effort.

Previous work which performed coarse-grain measurement have provided evidence for the existence of opportunities to auto-tune for energy in parallel applications[20]. Using the Resource Centric Reflection daemon tool (RCRtool) developed at RENCI[13], energy consumption can be measured at a fine granularity for any OpenMP program. Fine-grain measurements enable attribution of energy consumption to particular application regions and even to the individual lines of codes. This allows accurate study of the correlation between execution time and energy consumption of an application.

For most scientific applications, nested loops consume the significant portion of the total running time. When tuning the application for better performance and energy usage, some combination of loop optimizations, including loop tiling, loop unrolling, and loop fusion, are usually performed on the program along with the auto-parallelization. Determining which set of optimizations produces the best results is hard. Polyhedral auto-tuning frameworks have shown promising results at simplifying that effort[12, 10, 11] for small computation kernels like the Polybench programs[15]. In addition to the polybench kernels, we also examine two small applications with the polyhedral framework to determine the frameworks' effectiveness at reducing overall energy consumption.

This paper has 3 main contributions: 1) Fine-grained measurement of execution time and energy consumption with RCRtool and documenting the correlation between the two. 2) Additional speedups up to 20% over the already efficient OpenMP baseline implementation of a small realistic application using polyhedral optimizations on Intel Sandy Bridge and Xeon Phi processors. 3) Evaluation of how different architecture effects the utility of the polyhedral optimizations techniques for execution time and energy.

The rest of the paper is organized as follows. In Section 2, we describe the tools used to measure energy consumptions. Section 3 describes the benchmarks used for measuring the energy consumptions and evaluating the effectiveness of polyhedral compilers in optimizing a realistic application. Experimental setup, results and analysis are presented in Section 4 and Section 5. Section 6 explains and compares with related work. Section 7 has our conclusions.

2. ENERGY MEASUREMENT-AND-TUNING TOOLS

To understand energy consumption, execution time and various optimizations, a light-weight fine-grained measurement tool is required. RCRtool provides user-level fast access to hardware counters. Finding the optimal combination of compiler optimizations requires a compilation framework, like the Polyhedral Compiler Collection, that easily produces a large number of program variants with specific optimization parameters.

2.1 RCRtool on Sandy Bridge System

The Intel Sandy Bridge architecture allows users to track energy usage through the exposed Running Average Power Limit (RAPL) interface. A model-specific register (MSR) was added with the Sandy Bridge to track energy consumed by the chip – MSR_PKG_ENERGY_STATUS. The counter is frequently updated and counts the energy in 15.3 micro-Joule units. Experiments have shown[13] and the documentation[6] states that the counter can be accessed as often as every microsecond. The counter is only 32 bits and can wrap in as little as a couple of minutes. The RCRtool detects the wraps and supplies a 64 bit value with the upper 32 bits being the number of wraps since RCRtool instantiation. The RCRtool must run at supervisor level to access the counter. It writes the current value of the counter at least 1000 times a second into a shared-memory data structure. This "blackboard" structure provides a hierarchical view of system where various current performance information is stored. The information is available to any OpenMP applications through a simple API that delineates a code region for measurement with a start and end call. Each region is identified by its file name and line number. If a region is executed multiple times the energy is summed across all executions. All energy information is available during application shutdown.

The ROSE source-to-source compiler[17] finds OpenMP parallel regions and adds RCR API calls around the region automatically. ROSE tracks original file name and source line number allowing simple parallel region identification. When the program finishes execution, the elapsed time, the amount of energy used (in Joules), and the average computed power (in Watts) of the parallel regions and the whole application are output. Additional information such as processor temperature is also available during application shut-down. RCRtool runs on Intel architecture with the RAPL interface, and has been tested on Sandy Bridge and Ivy Bridge implementations.

The overhead of the RCRdaemon is negligible on both architectures. It enables us to measure the energy consumption of the application with a granularity of about a millisecond.

2.2 RCRtool on Xeon Phi System

The Intel MIC architecture in the Intel Xeon Phi chips is a recent addition to the Intel processor offerings. Our Phi accelerator cards contain 61 cores, each core supports 4 hardware threads. One notable feature is the 512-bit wide SIMD vectors providing fine-grain vectorization and high floating-point performance for each thread. With the wide vector registers, a single instruction can operate on 8 adjacent double-precision floating point data or 16 single-precision floating point data. The cores, threads and vector unit combine to achieve well over a Teraflop from a single socket.

RCRtool collects power information of Intel Phi natively or on the host. Natively, users can track power usage in microWatts through a file (/sys/class/micras/power) updated every 50 millisecond. RCRtool monitors the power at user level and computes the energy consumption over time. The information is available to the applications through the same simple API as on Sandy Bridge.

RCRtool can also run on the host. On the host, it collects power information using the MICAccessSDK API provided by Intel at the same granularity as the native version. Measurements in this paper were collected with a native Phi RCRdaemon.

2.3 Polyhedral Optimizations Tools

The Polyhedral Compiler Collection (PoCC)[14] was used to generate program variants with different optimizations. The PoCC requires that programs contain static control parts (SCoP)[2, 4] so that valid transformations can be applied. Polybench is a collection of programs that contain SCoPs and can be polyhedral optimized.

PolyOpt (a Polyhedral Optimizer for the ROSE compiler)[16] was also used to automatically detect SCoPs in applications. PolyOpt is integrated into the ROSE compiler. Aside from its capability of extracting SCoP regions in an automatic way, it fully supports PoCC analysis and optimizations. PolyOpt supports loops fusion, loop tiling, thread-level parallelization and vectorization. PolyOpt has better support for side-effect free program features like math functions[1], allowing some function calls within a SCoP.

PoCC generates hundreds and even thousands of program variants for simple programs, like Polybench. PolyOpt, although more powerful than PoCC, still may not be able to extract any SCoPs because of structural impediments. Changes to the program may be required to "manually" expose the SCoPs for PolyOpt, allowing loop transformations, parallelization, and vectorization to occur.

3. BENCHMARKS FOR ENERGY MEASUREMENT AND TUNING

In this work, we evaluate three kinds of programs for energy auto-tuning with polyhedral framework: Polybench programs, publicly-accessible LULESH program[7], and a realistic application developed and frequently used by our collaborators.

3.1 Polybench

Previous work has obtained significant speedups with the polyhedral framework for the Polybench programs[12, 10, 11]. Extending that work to examine whether the best tuned variants are also the most energy efficient is the focus of this work. Using PoCC, program variants were generated using a different set of the optimizations from the following five groups:

- Loop fusion: smartfuse, maxfuse, nofuse
- Loop unrolling factor: 1, 2, 4, 8
- Loop tiling: 1, 16, 32, 64. Note that the number of different flags depends on the level of nested loops
- Loop vectorization: on, off
- Loop parallelization: on, off

If the maximum nested loop level is 3, applying all possible combinations of the above flags generates 5135 program variants. The ROSE source-to-source compiler was used to add

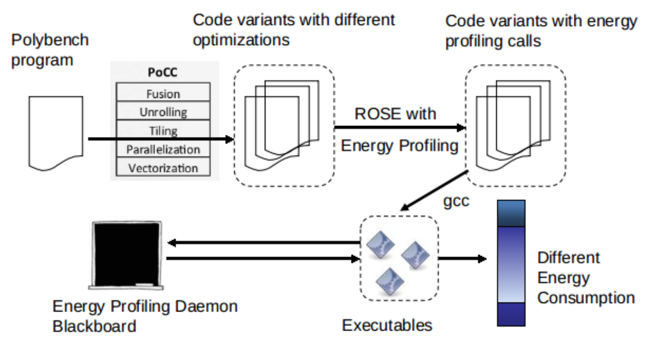

Polybench program
Code variants with different optimizations
Code variants with energy profiling calls
PoCC
Fusion
Unrolling
Tiling
Parallelization
Vectorization
ROSE with Energy Profiling
gcc
Different Energy Consumption
Energy Profiling Daemon Blackboard
Executables

Figure 1: Graph showing the workflow of obtaining energy consumption of polyhedral optimized (Polybench) programs.

energy profiling calls to each variant. GCC(4.4.6) generated the final executable. During execution, periodic queries to the RCRtool blackboard provide the energy consumption information. Figure 1 gives the workflow for measuring energy consumption of Polybench programs using the energy-aware polyhedral compiler framework.

3.2 LULESH

LULESH[7] is a shock hydrodynamic simulation application. It mimics a larger realistic application called ALE3D. We used the OpenMP implementation v1.0 for evaluation. The original LULESH uses a block structured mesh accessed via an indirect reference pattern[7]. In order to make LULESH go through the polyhedral compilation procedure, we modified LULESH by resolving all indirect array accesses. Although doing this oversimplified LULESH, it allows us to study the energy and time relationship of polyhedral compilation techniques with LULESH.

LULESH OpenMP implementation contains 30 parallel regions, 6 of which take up more than 60% of the total application time[13]. We manually converted two most significant parallel regions to two SCoPs so that they can be passed to polyhedral framework. The resulting largest SCoP contained too many dependences and we found it was hard for the polyhedral compiler to finish transformation and parallelization. When all temporary variables were eliminated from the most computationally intensive loop to create an SCoP, greatly expanded statements required hours of compilation to finish generating even one variant. In this work, we focus on optimizing the 2nd (largest) SCoP of LULESH. 200 program variants were produced by applying loop fusion (maxfuse and smartfuse), loop tiling, vectorization, and parallelization. The execution time and energy of each was measured.

3.3 A Realistic Application

In addition to the Polybench programs, a 2D monodomain cardiac wave propagation simulation (named *brdr2d*) was used as a test case. Its model involves solving a set of ODEs and PDEs and is well-known in the computational cardiac modeling field[22]. Equation (1) is the PDE that needs to be solved. The ODEs are used to represent the I_{ion} variable.

$$C_m \frac{\partial V_m}{\partial t} = \nabla \cdot D \nabla V_m - I_{ion} \tag{1}$$

The sequential C implementation is more than 1K lines.

One loop nest takes up more than 90% of the total application execution time. The dominate loop nest is an ideal situation for the polyhedral compiler. The loop nest is inside a while loop and is executed many times. This code structure is not unique to cardiac wave propagation simulation. Computationally dominate loops inside either while loops looking for some termination condition or inside a simulation time-step loop are common in scientific codes. LULESH falls into this category with multiple loop nests within a time-step loop.

While PolyOpt originally cannot extract any SCoP from *brdr2d*, it does output information useful to the user to manually transform the application to contain at least one SCoP. To expose the SCoPs the following changes were required. The computation part of *brdr2d* was fully inlined removing all function calls. Then, all array indexes were changed to be affine functions of the loop iterators. This involved loop unswitching to specialize modular operations like *step* % 2. Finally, the number of dependencies was reduced by forward substitution of temporary variables. After these changes, PolyOpt automatically detected the code region and applied various transformations to the SCoPs.

Different program variants were generated to explore data locality and parallelism using loop fusion (smartfuse/maxfuse), different tiling sizes, vectorization and auto-parallelization. OpenMP pragmas were automatically generated for each variant. The original sequential C implementation had OpenMP pragmas manually added to serve as a baseline. Four different input files for *brdr2d* were used to study how the performance of the program variants is impacted by different input sizes.

4. EXPERIMENTAL SETUP

The tests ran on a 2-socket 8-core Intel Xeon E5-2680 processor with 20MB (40MB total) L3 cache. PoCC v1.2 was used to generate program variants from Polybench v3.2. The extra large data set (specified in Polybench) was used. A few modifications were made to ROSE (version timestamped 1370387370) to insert the energy API calls. GCC v4.4.6 was the backend compiler. Every executable was compiled with -O3 optimization flag. To protect against low start-up energy/power measurements, the system was warmed up with a computational intensive script before any test was executed.

Experiments were also run on a Xeon Phi coprocessor. The Phi architecture accelerator card contained 61 cores clocked at 1.09GHz. Each core had 512KB of L2 cache. The generated program variants used ICC v14.0.0 compiler as their backend, producing OpenMP programs that ran natively on the Phi.

5. EXPERIMENTAL RESULTS

The polyhedral framework was first used to examine the energy usage of (and the execution time) of the Polybench programs and LULESH on the Intel Sandy Bridge architecture. These programs are written to allow easy framework manipulation of the program. A more realistic application *brdr2d* on both the Intel Sandy Bridge and the Intel Xeon Phi architectures is then studied.

5.1 Execution Time and Energy Consumption Correlation

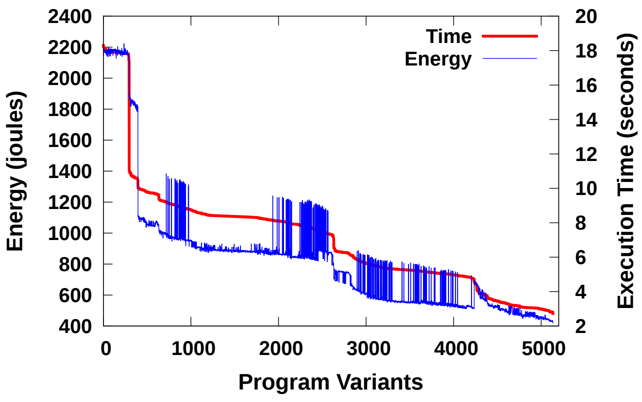


Figure 2: Graph showing the relationship between the execution time and the energy consumption of all *covariance* Polybench program variants on Sandy Bridge Processor (sorted by execution time). The spikes are caused by "maxfuse" loop transformation.

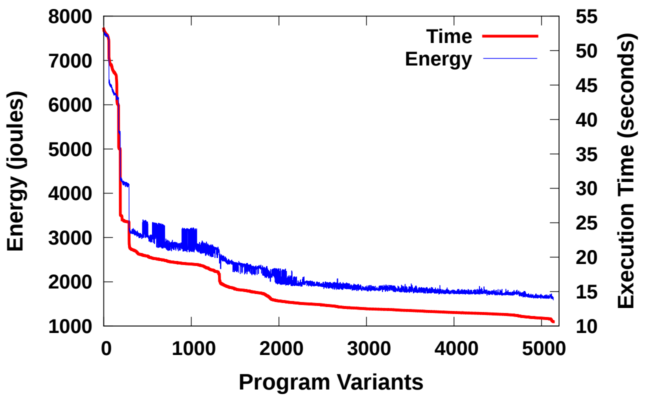


Figure 3: Graph showing the correlation between the execution time and the energy consumption of *2mm* Polybench on Sandy Bridge Processor (sorted by execution time). The spikes are caused by bad tiling configuration.

The first experiments verify the relationship between execution time and energy consumption.

5.1.1 *Polybench*

The Polybench v3.2 contains 30 programs. Because of the large number of variants created, the energy consumption of 2 (*covariance* benchmark and *2mm* benchmark) were chosen for closer examination. Figure 2 shows the relationship between the execution time and the energy usage for the 5135 variants of the *covariance* benchmark, sorted by execution time. The left y-axis shows the energy consumption (in joules) and the right y-axis shows the execution time (in seconds).

There is clear correlation between the time and the energy. The energy line (blue, mostly the bottom) generally follows the time line. The best optimized program variant for time (bottom right in the figure) consumed the least amount of energy. The energy line has many places where 2 runs that take the same amount of time consume significantly different energy. These appear as spikes in the graph. Examining the data, we noticed that the higher energy usage value always had the "maxfuse" flag set. The last jump is at variant 4236, above which all executions have "maxfuse" set. The executable with "maxfuse" requires significantly more power than with either "smartfuse" or "nofuse". For the executables where no performance improvement is gained, this has noticeable energy costs. However, when the polyhedral framework finds the correct tiling size, the "maxfuse" flag produces a significantly fast executable (note the change in the execution curve that occurs at variant 4236). The "smartfuse" and "no fuse" options produce executables that run at lower power and may be beneficial if peak power usage is a constraint, but longer execution times (up to 2×) result in significantly higher overall energy costs. The interaction between optimizations can have significant impact on their effectiveness for both time, energy and power.

A similar correlation between execution time and energy occurs for the *2mm* benchmark (as shown in Figure 3). No single optimization has as great an effect on power as "maxfuse" did in the previous example. The spikes that do occur (especially the left side) are from poor tiling configurations. Power is approximately constant for all the runs, so energy consumption is a function of execution time.

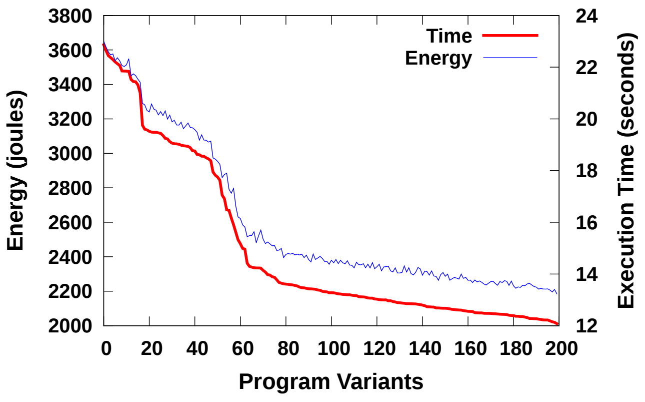


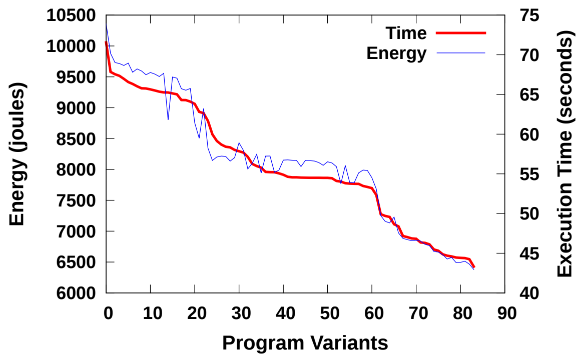
Figure 4: Graph showing the correlation between the execution time and the energy consumption of LULESH program on Sandy Bridge Processor (sorted by execution time).

5.1.2 *Modified LULESH*

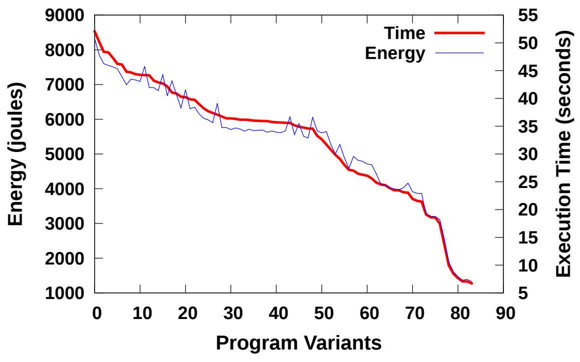
For 200 variants of LULESH, Figure 4 shows the energy used and execution time. The energy curve mirrors the execution time. A slight ($< 2\%$) run-to-run variation in the energy, presents a minor opportunity for energy tuning beyond execution time. LULESH optimizations overall provide almost a 2× reduction in execution time (22.9 vs 12.1 seconds - 47% reduction) and a significant decrease in energy (3650 vs 2185 Joules - 40% reduction). No optimizations resulted in a significant increase in power, although the power required did rise slightly (from 160 Watts to 180 Watts - 12% increase).

5.1.3 *Realistic Application*

brdr2d contains two symmetric SCoPs (because of loop unswitching). Each SCoP contained 42 statements. The



(a) Time and energy correlation on Sandy Bridge



(b) Time and energy correlation on MIC

Figure 5: Graph showing the correlation between the execution time and the energy consumption of *brdr2d* on Sandy Bridge processor and on MIC architecture (both sorted by execution time).

number of dependencies between these statements was 638 (there were no loop carried dependencies). PolyOpt detected and applied loop fusion (maxfuse or smartfuse) and loop tiling transformations (various tile sizes) as well as vectorization and auto-parallelization to the SCoPs. The fastest 84 program variants were chosen for study on both the Sandy Bridge processor and Xeon Phi coprocessor. 49 of the 84 programs had "maxfuse" flag turned on.

Figure 5 compares execution time and energy consumption (for 2048 input size) for the cardiac simulation application on the Sandy Bridge processor and on Xeon Phi card. Both Figure 5(a) and Figure 5(b) show that the energy tracks the time. Saving energy consumptions is consistent with improving performance on both processors. The energy line is above the time line for about half of Figure 5(a). Those variants have "smartfuse" set, which in this case increases the power required by the application (and therefore energy). Figure 5(b) has the time line above the energy line for the left half but below for the right half. For the Phi, the "smartfuse" option used lower power than "maxfuse". The performance of "maxfuse" was much better than "smartfuse" and the overall execution time and energy use for "maxfuse" was lower (up to 5×).

5.2 Polyhedral Optimization Results on the Realistic Application

Optimizing *brdr2d* on the Sandy Bridge Processor and on Phi coprocessor show the advantage using the polyhedral framework to optimize for both execution time and energy.

5.2.1 Results on Sandy Bridge Processor

To better understand the optimization variants of *brdr2d* they were executed with four different input sizes. Figure 6 compares the best variant for each input size with the baseline OpenMP version. The optimal tiling was different as the input grew. For the 256 case 1×128 resulted in the fastest execution, For the larger cases, the variant with tile size 1×256 was fastest. As the problem size grew the optimized variants' relative performance and energy consumption improved (256 - 2.5% to 2048 - 21%). As the loop size increases and the loop nest becomes a more dominant portion of the execution, the relative performance from optimization improves. For the smaller sizes the data fit into various cache

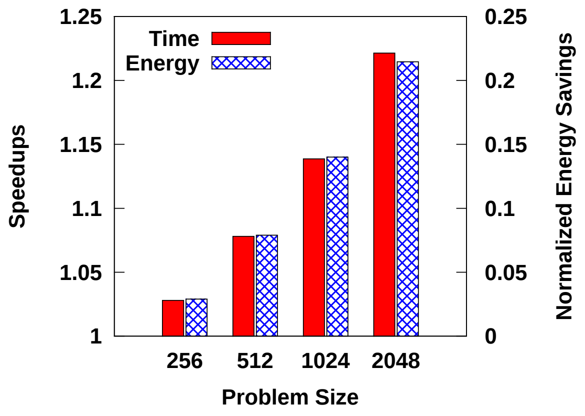


Figure 6: Graph showing the performance improvement and energy savings of the optimal program variant over the baseline OpenMP implementation for different problem size on Sandy Bridge Processor.

levels and the benefits of loop optimizations for data locality are ineffective.

5.2.2 Results on Intel Xeon Phi

To show the benefits of using polyhedral optimization techniques on the Phi accelerator card, the performance of a manual OpenMP implementation can be compared with the best Polyopt/PoCC generated OpenMP program variant (shown in Figure 7). The speedups were calculated against a sequential Sandy Bridge execution.

The best PoCC variant of *brdr2d*, is over 20% faster than the baseline Phi version for small sizes. For the largest size, 2048, the best Polyopt/PoCC variant is still slightly better than the baseline and has an absolute speed up of over 150×. The optimal tiling size changes as the input grows. In each case, $1 \times size$ is preferred for maximum vectorization. As the problem size grows, non-tiled vectorization improves, reducing the effectiveness of tiling. As expected the two main performance drivers for the Phi are parallelization, for threads, and vectorization, within threads.

The polyhedral optimizations also improves energy. Fig-

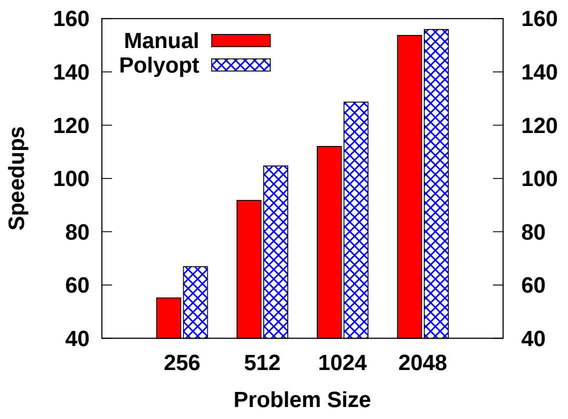


Figure 7: Graph showing the comparison between speedups of manual OpenMP implementation and the best Polyopt/PoCC generated OpenMP program variant over the sequential implementation on MIC architecture.

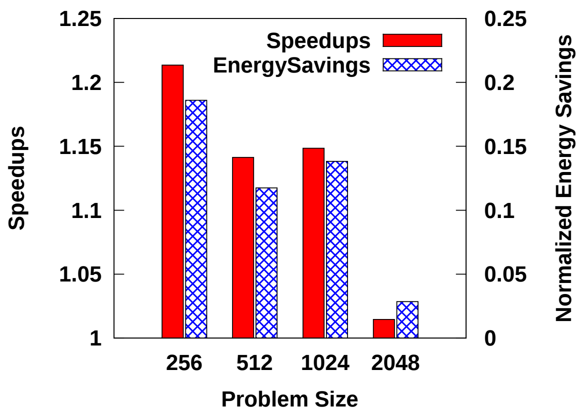


Figure 8: Graph showing the performance improvement and energy savings of the optimal Polyopt/PoCC generated program variant over the baseline OpenMP implementation for different problem size on MIC architecture.

ure 8 shows the *relative* speedups and the normalized energy savings offered by the polyhedral transformations and auto-parallelization. The energy savings approximately match the relative speedups, ranging from 20% down to 3% as size increases (and baseline vectorization improves).

6. RELATED WORK

6.1 Benchmarks for Polyhedral Framework and Auto-parallelization

There are a few benchmarks available to evaluate polyhedral transformations as an approach to improving application performance. Our work adds two non-trivial application (LULESH and *brdr2d*) to the family of benchmarks that are polyhedral optimizable. Polybench[15] and SWIM[21] benchmark are two benchmark suites that are often used. The Polybench programs evaluate polyhedral transformations and are used to construct predictive models by Park *et al.*[12, 10, 11]. They are also used to evaluate auto-parallelization techniques targeting different architectures, using different tools. Grauer-Gray *et al.*[5] utilized high level languages to target GPU architecture by annotating Polybench programs. Konstantinidis *et al.*[9] studied GPU code generation given the Polybench programs that contain affine loops.

6.2 Tuning for Performance and Energy

Rahman *et al.*[18] studied the impact of application level optimizations from both the performance and power efficiency perspective of various applications. They found that optimizing for performance did not guarantee better power consumption. We observed similar results in Figure 2 and Figure 3 for non-optimal program variants but the graphs showed that for the optimal case, tuning for performance and power were effectively equivalent. To improve performance and energy efficiency for a Many-Core architecture, Garcia *et al.*[3] studied the energy consumptions of applications and proposed models characterizing application energy consumption footprints. We did not develop energy models but took advantage of the exposed hardware interfaces to obtain accurate energy consumption information from modern commodity processor architectures like Intel Sandy Bridge and Xeon Phi.

To improve performance, people have developed techniques from distinctive ways. Tavarageri *et al.*[19] adopted compiler analysis approach to configure the cache size to reduce energy consumption without performance loss. New programming languages[8] and models like Chapel, Liszt and others were introduced to facilitate program optimizations on parallel architectures.

7. CONCLUSION

As expected, using an auto-tuning framework on a variety of small benchmarks and small applications has shown the high degree of correlation between execution time and energy consumption. Individual optimization can however have significant impact on the Power required by an application. In the Polybench program, *covariance*, using the "maxfuse" option resulted in a 20+% percent power increase. With the correct tile size "maxfuse" also resulted in the 50+% time decrease. Understanding how optimizations effect power and execution time will be important for future exascale systems.

Polyhedral optimization techniques can provide significant increases in performance but currently require significant user modifications to any real application to generate SCoPs with reasonable compilation times. On small real applications, like LULESH and *brdr2d*, polyhedral transformations allow the discovery of effective tiling sizes for SCoPs within the applications.

8. ACKNOWLEDGMENTS

This work is supported by the DOE XPress (DE-SC0008704) and the DoD ATPAR. The authors would like to thank EunJung Park and Matthew Kay for sharing the polyhedral optimized Polybench programs and the cardiac wave propagation simulation application. The authors would also like to thank Louis-Noël Pouchet, Albert Cohen, and Riyadh Baghdadi for their help with the polyhedral compilation of benchmarks.

9. REFERENCES

[1] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction*, CC'10/ETAPS'10, pages 283–303, 2010.

[2] P. Feautrier. Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[3] E. Garcia and G. Gao. Strategies for improving performance and energy efficiency on a many-core. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 9:1–9:4, 2013.

[4] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. Parallel Program.*, 34(3):261–317, June 2006.

[5] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10, 2012.

[6] Intel. Intel 64 and IA-32 architecures software developer's manual, volume 3. `http://download.intel.com/products/processor/manual/253669.pdf`, 2013.

[7] I. Karlin, A. Bhatele, B. L. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, E. Luke, S. Lloyd, J. McGraw, R. Neely, D. Richards, M. Schulz, C. H. Still, F. Wang, and D. Wong. Lulesh programming model and performance ports overview. Technical Report LLNL-TR-608824, December 2012.

[8] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still. Exploring traditional and emerging parallel programming models using a proxy application. In *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA, May 2013.

[9] A. Konstantinidis, P. H. Kelly, J. Ramanujam, and P. Sadayappan. Parametric gpu code generation for affine loop programs. 2013.

[10] E. Park, J. Cavazos, and M. A. Alvarez. Using graph-based program characterization for predictive modeling. In *CGO*, pages 196–206, 2012.

[11] E. Park, J. Cavazos, L.-N. Pouchet, C. Bastoul, A. Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. *International Journal of Parallel Programming*, 41(5):704–750, 2013.

[12] E. Park, L.-N. Pouche, J. Cavazos, A. Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 119–129, Washington, DC, USA, 2011. IEEE Computer Society.

[13] A. Porterfield, R. Fowler, S. Bhalachandra, and W. Wang. OpenMP and MPI application energy measurement variation. In *Proceedings of the 1st Workshop on Energy Efficient Super Computing*, E2SC '13, 2013.

[14] L.-N. Pouchet. PoCC: the Polyhedral Compiler Collection. `http://www.cs.ucla.edu/~pouchet/software/pocc/`, 2013.

[15] L.-N. Pouchet. Polybench. `http://www.cse.ohio-state.edu/~pouchet/software/polybench/`, 2013.

[16] L.-N. Pouchet. PolyOpt/C: a Polyhedral Optimizer for the ROSE compiler. `http://www.cs.ucla.edu/~pouchet/software/polyopt/`, 2013.

[17] D. J. Quinlan and C. L. Liao. ROSE Compiler. `http://rosecompiler.org/`, 2013.

[18] S. M. F. Rahman, J. Guo, A. Bhat, C. Garcia, M. H. Sujon, Q. Yi, C. Liao, and D. Quinlan. Studying the impact of application-level optimizations on the power consumption of multi-core architectures. In *Proceedings of the 9th conference on Computing Frontiers*, CF '12, pages 123–132, 2012.

[19] S. Tavarageri and P. Sadayappan. A compiler analysis to determine useful cache size for energy efficiency. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 923–930, 2013.

[20] A. Tiwari, M. A. Laurenzano, L. Carrington, and A. Snavely. Auto-tuning for energy usage in scientific applications. In *Proceedings of the 2011 international conference on Parallel Processing - Volume 2*, Euro-Par'11, pages 178–187, Berlin, Heidelberg, 2012. Springer-Verlag.

[21] N. Vasilache, C. Bastoul, and A. Cohen. Polyhedral code generation in the real world. In *In Proceedings of the International Conference on Compiler Construction (ETAPS CC'06), LNCS*, pages 185–201. Springer-Verlag, 2006.

[22] W. Wang, H. Huang, M. Kay, and J. Cavazos. GPGPU accelerated cardiac arrhythmia simulations. In *Engineering in Medicine and Biology Society,EMBC, 2011 Annual International Conference of the IEEE*, pages 724–727, 2011.