2018 Building Performance Modeling Conference and
SimBuild co-organized by ASHRAE and IBPSA-USA
Chicago, IL
September 26-28, 2018

ENERGYPLUS INTERIOR RADIANT HEAT EXCHANGE RUNTIME PERFORMANCE IMPROVEMENTS

Joshua New, Ph.D. and Mark B. Adams
Oak Ridge National Laboratory, Oak Ridge, TN

ABSTRACT

EnergyPlus is the flagship whole-building energy simulation program developed by the US Department of Energy. This paper describes the various approaches to improve the performance of the interior radiant heat exchange algorithm. Vectorization, optimized Basic Linear Algebra Subprograms (BLAS) library, multithreading, and Graphical Processing Unit (GPU) computation are all investigated. The approach with the best performance, while maintaining drop-in compatibility with existing EnergyPlus code, is the optimized BLAS library. However, GPU computing has the potential to reduce simulation time by orders of magnitude with major refactoring to EnergyPlus.

INTRODUCTION

EnergyPlus began in 1995 to replace DOE-2 and is currently the US Department of Energy's (DOE's) flagship whole-building energy simulation program (Crawley et al. 2001). Since that time, DOE has invested over $65 million in adding new building technologies and modern simulation capabilities. The primary users of EnergyPlus are architects and engineers assessing energy impacts in design, major software vendors offering simulation-based services for buildings, and agencies interested in code and policy impacts on building energy use. EnergyPlus is released under a commercial-friendly, open-source license on GitHub.

EnergyPlus is capable of modeling most building materials, constructions, and equipment with output from annual to 1 minute resolution for energy use, temperature, relative humidity, and other fields of interest. However, this varying level of fidelity can have dramatic effects on the overall runtime of the simulation.

To accurately model one aspect of the building physics, EnergyPlus calculates the heat transfer in the building every timestep. One heat transfer calculation, interior radiant heat exchange, is computationally intensive in its current form, especially for large buildings with many surfaces in each thermal zone. EnergyPlus calculates the net long wave radiation (NLWR) from all surfaces in a thermal zone that have a direct line of sight with another surface. EnergyPlus uses Hottels ScriptF method to approximate the gray body heat exchange of each surface (Hottel 1954). Due to the high computational intensity for calculating ScriptF factors and NLWR for each surface, these were investigated for different methods to speed up the computation. Multiple Central Processing Unit (CPU)-based and Graphical Processing Unit (GPU)-based improvements were studied. To help facilitate easier testing of the various approaches, as well as data structure changes, the original EnergyPlus code for calculating interior radiant heat exchange was ported to a simplified C codebase specific only to this calculation.

CPU-BASED IMPROVEMENTS

This first section looks at various CPU-based performance enhancements to the test code. Since the existing code is serial, CPU-only code, this is a logical starting point for drop-in ready improvements.

Table 1: Comparison of performance improvement techniques using timestep of 15 minutes (35,040 iterations), 256 thermal zones, and 128 surfaces per zone

	Time (minutes)	Speed-up
Naïve	4.48	-
Altered Data Structure	2.04	2.2x
BLAS (single threaded)	2.25	2x
Hand Vectorize	4.09	1.1x

There are numerous approaches to refactor code to reduce CPU usage, however, this paper looks at four specifically: data structure changes, auto-vectorization, optimized BLAS library, and parallelization using OpenMP. All modern CPUs have vector processing units that allow them to perform calculations on multiple pieces of data with a single instruction, allowing for massive performance improvement. When a compiler automatically optimizes a piece of code to use these specific vector instructions, it is called auto-vectorization. Additionally, as modern processors add more cores each year, parallelization becomes more important as a mechanism for performance improvement. OpenMP is a library that enables quicker development of multithreaded code through the use of compiler directives (Dagum and Menon 1998). This provides a mechanism to parallelize existing code with minimal refactoring. Ideally, computation can be distributed to each core on a processor for linear speed-up. Finally, an optimized BLAS library combines vectorization, parallelization, and other techniques to speed-up basic linear algebra functions, such as matrix multiplication (Lawson et al. 1979).

Using a building with 256 thermal zones and 128 surfaces per zone, this provided the basis for testing single threaded performance improvements. The interior radiant heat exchange algorithm iterated 35,040 times corresponding to a 15 minute timestep. As shown in Table 1, auto-vectorization with data structure improvements and BLAS yielded similarly good performance. The auto-vectorized code takes better advantage of the machine used for testing, which is a Haswell-based Intel processor with a 256-bit-wide vector unit (AVX2) and is typical of current Intel-based processors. The 256-bit-wide vector unit means this processor can perform four calculations simultaneously on double precision numbers, or a maximum potential 4x speed-up. However, it is challenging to achieve the ideal maximum speed-up as shown in Table 1, where the maximum speed-up achieved is only 2.2x.

Altering the data structure allows: (1) reduced number of calculations for improved arithmetic intensity, which is the ratio of computations performed versus memory accessed, and (2) reduced number of memory lookups for improved cache coherence of the loop. The hand-vectorized results show the difficulty in writing high performance code that takes full advantage of the hardware and pipeline. A primary difference between hand- vs. auto-vectorized code was that auto-vectorized code performed aggressive loop unrolling while the hand-vectorized code had none. Writing high performance code under all code uses and machines is challenging; therefore, using a highly optimized BLAS library is often the best approach to improve the performance of an application. The single-threaded BLAS implementation was quicker to implement and less intrusive than the data structure changes while completing in nearly the same time.

Table 2: Comparison of performance improvement techniques using timestep of 15 minutes (35,040 iterations), 1 thermal zone, and 1,024 surfaces per zone with 8 threads (4 cores)

	Time (seconds)	**Speed-up**
Naïve	67.2	-
BLAS (multithreaded)	6.67	10.1x
OpenMP	13.59	4.9x
Altered Data Structure	19.93	3.4x

To test multithreading improvements, a building with 1 thermal zone and 1,024 surfaces was used for the simulation. Table 2 quantifies multithreading improvement for a code base that is already vectorized and pipelined. This test stressed the O(n^2) algorithm when calculating the NLWR, where n is the number of surfaces. Comparing the vectorized code improvements in Table 1 and Table 2, the runtime speed-up increases from 2.2x to 3.4x as the number of surfaces per zone increases. This is due to the improvements in loop unrolling, cache coherency, and pipelining. The OpenMP version of the vectorized code saw an improvement, but it was only 1.47 times faster than the altered data structure results when it should have been 8 times faster if there was linear scaling. It is anticipated that this algorithm is memory bound, meaning it is limited by the main memory access speed, and acknowledge that the OpenMP parallelization may not have been at an optimal location. The OpenMP code parallelized the outer loop of surfaces when it is likely better to parallelize the thermal zone loop. The multithreaded BLAS did not have linear scaling, but had the best performance in this test case.

One visualization approach to help guide optimization is a roofline model. This model shows hardware limitations for computation and memory access to provide performance estimates of a given application. Running Lawrence Berkeley National Laboratorys Empirical Roofline Tool on the test machine gives the roofline model shown in Figure 1 (Lo et al. 2014). The interior radiant heat exchange function has an arithmetic intensity of 1 for the auto-vectorized code. This means the code is memory bound and effort should be focused on reducing memory accesses. This analysis also validates the multithreaded findings, which showed minimal benefit beyond the altered data structure. However, the optimized BLAS library is cache aware and cache sensitive, reducing memory accesses, so it still improves the performance of this memory-bound problem.

GPU-BASED IMPROVEMENTS

A GPU accelerator has the potential to significantly improve the performance of some programs. GPUs typically have higher computational power, relative to traditional CPUs, but relatively slow data transfer mechanisms. GPUs require computational kernels with low-data, high-compute algorithms to attain their full potential. NVIDIAs Compute Unified Device Architecture (CUDA) was used on a GeForce GT 750M graphics card with 384 streaming multiprocessors and 2 GB GDDR5 RAM with roofline model, as shown in Figure 2 (Nickolls et al. 2008). Given the same arithmetic intensity of 1 for the GPU accelerated code, the code is compute bound on this GPU device.

In CUDA programming, a GPU kernel is a grid of blocks

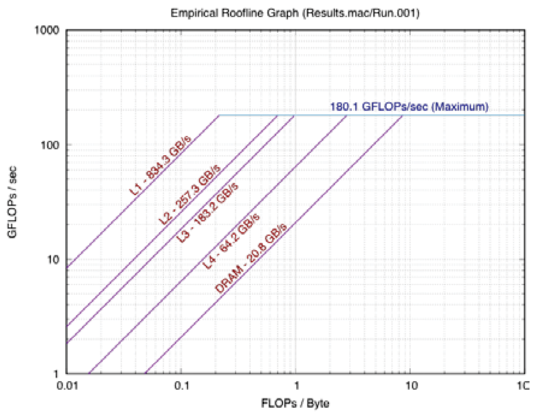

Figure 1: CPU roofline plot for 2013 i7 Haswell MacBook Pro.

and within each block is a grid of threads. This threading hierarchy is how GPUs achieve their massively parallel computation when computational work is divided across all blocks and threads. The direct port of the CPU code to CUDA has one kernel called every timestep (35,040 iterations). This kernel uses one block and assigns the number of threads to the number of zones for simplicity. However, higher parallelization could potentially be achieved with threads assigned to surfaces and blocks assigned to thermal zones. With the single block setup, buildings with larger numbers of thermal zones take better advantage of the 384 GPU cores. However, low numbers of thermal zones can lead to worse performance, or modest improvements for this test compared to the serial CPU code. The overhead of repeated GPU kernel calls can overwhelm parallelization and performance improvements from GPU computation. In order to highlight GPU potential, a test was created using a single GPU kernel for all iterations in all zones. This showed a 26.5 speed-up in simulation time, as shown in Table 3. However, this improvement is not realizable within the current structure of EnergyPlus. This experiment investigates the potential to convert the existing CPU-based code into GPU-based code, however, there are other GPU native approaches that yield greater performance results (Kramer et al. 2015).

FUTURE WORK

The potential of GPU computation within EnergyPlus should be further investigated. Interior radiation heat balance may not be the best place for performance enhancement, other locations in the code are potentially well-suited to GPU computation. Specifically, solar shading calculations seem to lend themselves to significant, GPU-based acceleration (Jones and Greenberg 2012) (Hoover and Dogan 2017).

Table 3: Comparison of GPU performance improvement techniques using timestep of 15 minutes (35,040 iterations), 16 thermal zone, and 16 surfaces per zone

	Time (seconds)	**Speed-up**
CPU-vectorized code	135	-
One kernel total	5.1	26.5x
One kernel per iteration	11.1	7.6x

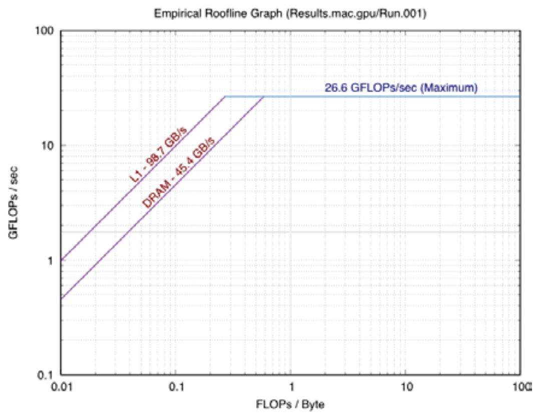

Figure 2: GPU roofline plot for 2013 i7 Haswell MacBook Pro with NVIDIA GT 750M.

Replacing hand-derived linear algebra and matrix inversions within EnergyPluss codebase with calls to an optimized BLAS library would likely yield significant performance increases with relatively minor code changes.

CONCLUSION

The highly optimized BLAS library shows the best performance improvements while maintaining the best drop-in compatibility with existing EnergyPlus code and avoids issues that may arise from hand- vectorizing code or relying on the compiler to auto-vectorize code. GPU computation has the potential to improve performance by orders of magnitude; however, this would require a larger refactoring effort to achieve this performance.

ACKNOWLEDGMENT

REFERENCES

Crawley, Drury B, Linda K Lawrie, Frederick C Winkelmann, Walter F Buhl, Y Joe Huang, Curtis O Pedersen, Richard K Strand, Richard J Liesen, Daniel E Fisher, Michael J Witte, et al. 2001. “EnergyPlus:

creating a new-generation building energy simulation program.” *Energy and buildings* 33 (4): 319–331.

Dagum, Leonardo, and Ramesh Menon. 1998. “OpenMP: an industry standard API for shared-memory programming.” *IEEE computational science and engineering* 5 (1): 46–55.

Hoover, Joel, and Timur Dogan. 2017. “Fast and Robust External Solar Shading Calculations using the Pixel Counting Algorithm with Transparency.” 08.

Hottel, H. C. 1954. *Radiant heat transmission.* 3.

Jones, Nathaniel L, and Donald P Greenberg. 2012. “ARDWARE ACCELERATED COMPUTATION OF DIRECT SOLAR RADIATION THROUGH TRANSPARENT SHADES AND SCREENS.” *Proceedings of SimBuild* 5 (1): 595–602.

Kramer, Stephan C, Ralf Gritzki, Alf Perschk, Markus Rösler, and Clemens Felsmann. 2015. “Fully parallel, OpenGL-based computation of obstructed area-to-area view factors.” *Journal of Building Performance Simulation* 8 (4): 266–281.

Lawson, C. L., R. J. Hanson, D. R. Kincaid, and F. T. Krogh. 1979. “Basic Linear Algebra Subprograms for Fortran Usage.” *ACM Trans. Math. Softw.* 5 (3): 308–323 (September).

Lo, Yu Jung, Samuel Williams, Brian Van Straalen, Terry J Ligocki, Matthew J Cordery, Nicholas J Wright, Mary W Hall, and Leonid Oliker. 2014. “Roofline model toolkit: A practical tool for architectural and program analysis.” *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, 129–148.

Nickolls, John, Ian Buck, Michael Garland, and Kevin Skadron. 2008. “Scalable Parallel Programming with CUDA.” *ACM SIGGRAPH 2008 Classes*, SIGGRAPH ’08. New York, NY, USA: ACM, 16:1–16:14.

NOMENCLATURE

BEM	Building Energy Modeling
BLAS	Basic Linear Algebra Subprograms
CPU	Central Processing Unit
GPU	Graphical Processing Unit
DDR5	Double Data Rate Type 5 Synchronous Graphics Random-Access Memory
RAM	Random Access Memory
CUDA	Compute Unified Device Architecture
NLWR	Net Long Wave Radiation