ACCELERATION OF RADIANCE FOR LIGHTING SIMULATION BY USING PARALLEL COMPUTING WITH OPENCL

Wangda Zuo, Andrew McNeil, Michael Wetter, Eleanor Lee
Building Technologies Department,
Environmental Energy Technologies Division,
Lawrence Berkeley National Laboratory,
Berkeley, CA 94720, USA

ABSTRACT

We report on the acceleration of annual daylighting simulations for fenestration systems in the Radiance ray-tracing program. The algorithm was optimized to reduce both the redundant data input/output operations and the floating-point operations. To further accelerate the simulation speed, the calculation for matrix multiplications was implemented using parallel computing on a graphics processing unit. We used OpenCL, which is a cross-platform parallel programming language. Numerical experiments show that the combination of the above measures can speed up the annual daylighting simulations 101.7 times or 28.6 times when the sky vector has 146 or 2306 elements, respectively.

INTRODUCTION

Due to the increasing demands in accuracies and resolutions, building simulations requires more and more computing power. Since increasing the clock rate alone cannot meet the rapidly growing demands on computing power, it is more feasible to compute in parallel on multiple processors. Parallel computing on supercomputers is already widely used in other industries and there are also a few applications in building industry (Wenisch et al. 2007, Hasama et al. 2008, Mazumdar and Chen 2008). However, purchasing and maintaining supercomputers is usually too expensive for small businesses that make up the majority of the building industry. A low-cost and high-performance parallel computing is necessary to meet the increasing computational needs of building simulations. Besides cloud computing (Armbrust et al. 2009), there are two other promising options for parallel computing. One is to use single/multiple CPUs with multi-cores, which are widely adopted by personal computers. The other is computing on graphics processing units (GPUs). The GPU is the core of a computer graphics card and has hundreds of low-frequency processors. Both options cost only a few hundred US dollars and can be realized on a desktop computer or a laptop computer. For example, Zuo and Chen (2010) accelerated an indoor flow simulation up to 30 × using a GPU on a desktop computer.

Radiance is a highly accurate ray-tracing program that is widely regarded as best in class for lighting simulation (Larson and Shakespeare 1998). A recent addition to Radiance, known as three-phase simulation method, enables users to perform annual daylight simulations for complex and/or dynamic fenestration systems (Ward 2010, Ward 2011).

The three-phase method breaks luminous energy traversal of the model into three phases: from sky to exterior of the fenestration, through the fenestration and from interior of the fenestration to the sensor points. Luminous energy transfer for each phase is described by a matrix of coefficients. The daylight (exterior) matrix characterizes how energy from each of 145 Tregenza sky patches arrives into 145 directional Klems patches at the window. The daylight matrix characterizes the external environment including obstructions. The fenestration transmission matrix characterizes how light incident on the fenestration in each of 145 incident patches leaves through 145 exiting patches. The transmission matrix characterizes transmission properties of a fenestration system, including diffusion and redirection of daylight. And finally the interior, or view, matrix characterizes how lighting leaving the fenestration in each of the 145 directional patches arrives at each of the illuminance sensor points. The view matrix characterizes flux transversal through the interior space model. Each matrix is independent of the others so, for example, the daylight matrix can be changed in order to simulate a different orientation or additional external obstructions without changing the other two matrices.

The three-phase method uses Radiance's *rtcontrib* program to produce the daylight and view matrices. The transmission matrix can either be produced using Radiance's *genBSD*, Window 6 or a combination of the two.

To generate an illuminance result we first create a sky vector using Radiance's *genskyvec*. The sky vector is 145 values, the luminance of the 145 Tregenza patches for a given time, location and sky type. The sky vector is multiplied by the other three coefficient matrices to generate a result for all of the sensor points. The three-phase method allows users to generate annual results for many different

fenestration systems including those with dynamic components.

The Radiance program that performs the matrix multiplication, called *dctimestep*, is a sequential code written in C language. The main calculation for this feature is to multiply matrices with large dimensions, which may take hours for an annual simulation depending on the number of illuminance sensor points. Accelerating *dctimestep* will enable users to quickly evaluate different fenestration systems and optimize the design of fenestration systems through parametric study. This could accelerate the adoption of emerging daylighting technologies by reducing the technical and market barriers.

To accelerate *dctimestep*, the key is computing the matrices multiplication more efficiently, such as computing them in parallel using multi-core CPUs or GPUs. There are various parallel programming languages, such as OpenMP (Chapman et al. 2007), MPI (Gropp et al. 1999), CUDA (NVIDIA 2007) and OpenCL (Munshi 2010). Since Radiance is a publicly released code running on various types of computing hardware, it is important that the programming language is supported by various platforms so that we can save a lot of repeated efforts on code development. Thus, we selected OpenCL because it is a cross-platform language and supported by major CPU and GPU vendors.

OPENCL

OpenCL is the first open standard for parallel programming on heterogeneous platforms, including CPUs, GPUs, embedded processors and other processors. The development of OpenCL was initiated by Apple Inc in 2008 and is currently led by Khronos Group. The most recent release of OpenCL was version 1.1 in 2010. The official website of OpenCL is http://www.khronos.org/opencl/.

OpenCL adopts a host-device platform model (Figure 1). A *host* is the commander that connects to one or more devices. A *device* contains one or more compute units. A *compute unit* can be further divided into one or more processing elements. The *processing element* is the basic unit for computing on a device. For instance, a computer may have two CPUs and one GPU. Each CPU has 2 processors and 4 cores per processor. The GPU has 42 processors and 8 cores on each processor. In OpenCL platform model, one CPU can be the host, the other CPU and GPU can be two devices. The device 1 (CPU) has 2 compute units (processors) and 4 processing units (cores) on each compute unit. The device 2 (GPU) has 42 compute units (processors) and 8 processing elements (cores) on each compute unit.

To execute an OpenCL program, we need a *host program* running on the host and one or more *kernels* running on devices. The host program identifies and initializes the OpenCL hardware, creates the OpenCL environment, defines and manages the kernel. The parallel computing is conducted through kernels on devices. Users can specify different kernels on various devices depending on the parallel modeling they use. Currently, OpenCL supports both data parallel programming and task parallel programming. For more details of OpenCL, one can refer the OpenCL specification (Munshi 2010).

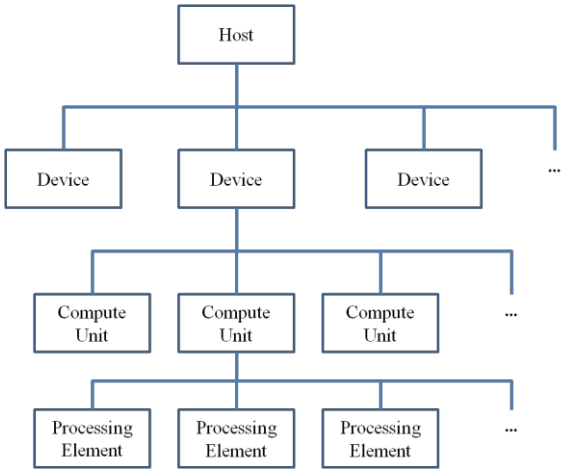


Figure 1 Platform model in OpenCL

IMPLEMENTATION

The *dctimestep* code was written for sequential computing. Additionally *dctimestep* was written to produce a result for one time step. To perform an annual simulation, *dctimestep* would be called roughly 4380 times by a script. To speed it up, it is critical to profile and identify potential parts for parallelization, modify algorithms for parallel computing, and optimize the implementation for better performance. This section will first introduce the analysis and optimization to reduce redundant computation and input/output (I/O) operation of the *dctimestep* algorithm for annual simulation. After that, we will discuss the further speedup by implementing matrices multiplication in OpenCL on a GPU.

Analyses and Optimization of Daylighting Simulation for a Period

The *dctimestep* is to calculate an illuminace vector, $V_I(t)$, at each simulation sensor point for one time step:

$$V_I(t) = M_V M_T M_D V_S(t), \quad (1)$$

where M_V is a view matrix defining lighting connection from the exiting directions of the windows to the sensors. M_T is a matrix converted from bidirectional transmittance distribution function (BTDF), which describes transmission of the light passes through the surface of studied windows. M_D is a daylighting matrix defining coefficients between incoming directions for the windows and sky patches. $V_S(t)$ is the sky vector defining sky patch radiance for a specific time step, t (Ward 2010). The implemented calculating sequence in *dctimestep* is

$$V_R(t) = M_D V_S(t), \quad (2)$$

$$V_C(t) = M_T V_R(t), \quad (3)$$

$$V_I(t) = M_V V_C(t), \quad (4)$$

where $V_R(t)$ and $V_C(t)$ are temporary vectors.

dctimestep calculates $V_I(t)$ for three basic colors (red, green and blue). For each color, the dimensions of M_V, M_T, M_D and $V_S(t)$ are the same although the value of entries may be different. Suppose $V_S(t)$ is a column vector with N entries and M_D, M_T and M_V are the $N \times N$ matrices, to calculate an entry $v_{Ri,j}(t)$ in $V_R(t)$ for one color, we need $2N - 1$ floating-point operations, including N multiplications and N-1 additions. $V_R(t)$ has N entries so that we need $N(2N - 1)$ floating-point operations in total for calculating $V_R(t)$. We also need the same number of floating-point operations for equations (3) and (4). Thus, the total number of floating-point operations for these matrices multiplications is $3N(2N - 1) \approx 6N^2$ (when $N \gg 1$). To calculate three colors at one time step, *dctimestep* needs $6N^2 \times 3 = 18N^2$ floating-point operations. For n time steps, *dctimestep* has to be invoked n times, so the total number of floating-point operations is about $18nN^2$.

Considering M_V, M_T and M_D are constant during the simulation, the other approach is to calculate $M_V M_T M_D$ once and use the result for the rest of simulation:

$$M_{VSD} = M_V M_T M_D, \quad (5)$$

$$V_I(t_k) = M_{VSD}\, V_S(t_k), \ \{t_k\}_{k=1,\ldots n}. \quad (6)$$

Assuming $N \gg 1$, the floating-point operations in equations (5) and (6) are $2N^2(2N - 1) \approx 4N^3$ and $nN(2N - 1) \approx 2nN^2$, respectively. For three colors, the total number of floating-point operations will be around $12N^3 + 6nN^2$.

Compared to the current approach in equation (2) to (4), the new approach in equations (5) and (6) can reduce the floating-point operations by $12nN^2 - 12N^3$. Assume there are 12 hours daylighting per day, we need to calculate $n = 4380$ vectors for an hourly simulation over a year. Suppose $N = 145$, then the reduction in floating-point operations is 1.07×10^9, which is 66% of total computing efforts.

Meanwhile, we found that there were significantly redundant data I/O operations when repeatedly calling dctimestep for annual simulation. For a simulation with n time steps, the current approach repeatedly reads the matrices M_V, M_T and M_D at each time step although they were the same. On the contrary, the new approach reads them only once.

Furthermore, it takes time to identify GPU platform, setup OpenCL programming environment and initialize the GPU memory when the GPU program is invoked. To utilize the high computing capacity of GPU, we can reduce the number of program invocations by merging n sky vectors $V_S(t_k)$ $\{t_k\}_{k=1,\ldots n}$ into a single matrix, $M_{VS} = [V_S(t_1), \ldots, V_S(t_n)]$.

Combining the optimizations mentioned above, we implemented a new algorithm for daylighting calculation over a period:

$$M_{VI} = M_V M_T M_D M_{VS}, \quad (7)$$

where $M_{VI} = [V_I(t_1), \ldots, V_I(t_n)]$.

In addition, the entries of the daylighting vector may be zero when there is no daylighting, such as at night. Thus, when $V_S(t_k)$ has only zero entries, we can set the corresponding entries in $V_I(t_k)$ to be zeros without calculation. We used three steps to implement the filtering-inserting procedure. First, we identify and remove the zero $V_S(t_k)$ from M_{VS}, which can reduce the $N \times n$ matrix M_{VS} to a $N \times n_1$ matrix M'_{VS1} (where n_1 is the number of non-zero sky vectors). Then, we calculate equation (7) and get a shrunk M'_{VI1} by using M_{VS1}. Finally, we get M_{VI} by inserting the zero vectors at corresponding columns in M'_{VI1}.

Using the new approach in equation (7) and the filtering-inserting approach, we implemented two programs. One was called *dctimestep_gpu* that was a hybrid code of C and OpenCL and ran on both CPU and GPU. The other was *dctimestep_cpu* that was a C code and ran only on CPU. The *dctimestep_cpu* was used to compare with *dctimestep_gpu* to quantify the performance enhancement by using GPU. Meanwhile, *dctimestep_cpu* can be useful for users who do not have OpenCL supported hardware. Since the implementation of *dctimestep_cpu* is straight forward, this paper will only discuss the implementation of GPU program *dctimestep_gpu* in detail.

Implementation of GPU Program Using OpenCL

We used a computer with an Intel Xeon CPU and a NVIDIA GeForce GTX 460. The configurations are given in Table 1. In our implementation, the host was CPU and the GPU was the device. Although the Xeon CPU has 4 processors and 24 cores, only one core is used by the host program. The GTX 460 GPU has 42 multi-streaming processors and 8 streaming processors on each multi-streaming processor, which is corresponding to 42 compute units and 336 processing elements in OpenCL.

Table 1 Configurations of computer hardware

CATEGORY	COMPUTER 1
GPU Type	NVIDIA GeForce GTX 460
GPU Cores	336
GPU Processor Clock	1350 MHz
GPU Memory Bandwidth	115.2 Gb/s
GPU Memory	1 GB GDDR5
Host-Dev Bandwidth	2767.6 MB/s
Dev-Host Bandwidth	2910.1 MB/s
CPU Type	Intel Xeon
CPU Processor Clock	2.67 GHz
CPU Cores	24
CPU Cache	48 MB
CPU Memory Clock	1333 MHZ
Operating System	Ubuntu 10.04

NVIDIA GPUs supports the OpenCL by running it on the CUDA architecture that is designed for NVIDIA GPUs. The basic computing unit in CUDA is *thread* and one multi-streaming processor can support 756 threads. Thus, the GeForce GTX 460

GPU can support up to 1,354,752 CUDA threads. For details about CUDA, one can refer the CUDA programming guide (NVIDIA 2007).

Figure 2 shows the schematic of implementation. The host programs are "dctimestep_gpu.c" and "matrixmul.c". The "dctimestep_gpu.c" is C code modified from "dctimestep.c". It reads M_V, M_T, M_D, and M_{VS} which contain data for three colors. Thus, we split them so that each matrix only contain data for one color. It also writes the M_{VI} when the simualtion is done.

The "matrixmul.c" prepares for matrix multiplication on GPU. It was modified from a matrix multiplication sample code in NIVIDA OpenCL SDK (NVIDIA 2011). It identifies and initializes the OpenCL device, creates OpenCL programming environment, allocates device memory and copies the data from host memory to device memory, and defines input parameters needed by the kernel functions. The original SDK code was written in C++ and we changed it to C code to be compatible with "dctimestep_gpu.c". In addition, the SDK code was desigend for $16a \times 16b$ matrices, where a and b are positive integers. Since the dimensions of matrices in our application can be arbitrary numbers, we modfied the code to eliminate this limitation.

After the initialization, a kernel "matrixmul.cl" is launched for matrix multiplication in parallel on device. In our implementation, one thread computes one entry by using the following code:

```
__kernel void
matrixMul ( __global float* C, __global float* A,
__global float* B, int uiWA, int uiWB){
 float Cele = 0;
 int i;

 //Identify the coordinate of thread
 int col = get_global_id(0);
 int row = get_global_id(1);

 //Matrix multiplication for element C(row, col)
 for(i = 0; i < uiWA; i++)
     Cele += A[uiWA * row + i]*B[i*uiWB +col];

 //Copy data from temporary variable to matrix
 C[row*uiWB+col] = Cele;
}
```

__kernel is an OpenCL key word which defines kernel functions. Function *get_global_id* ( ) returns the indices of a thread. All the data are stored in GPU global memory, which is the main GPU memory.

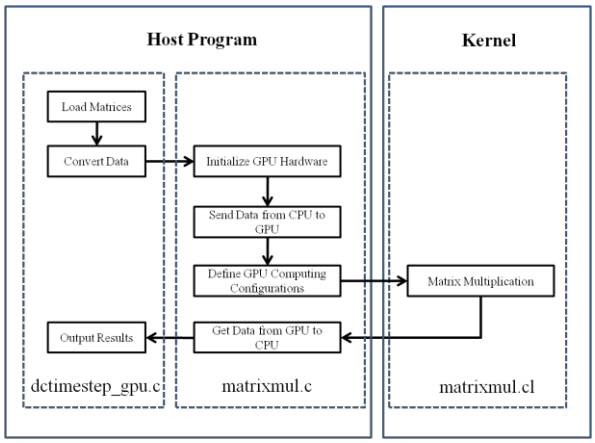


Figure 2 Schematic of implementation

NUMERICAL EXPERIMENTS

Settings

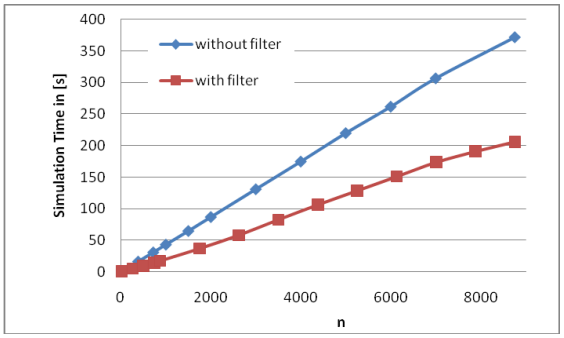
We evaluated three programs, including current *dctimestep* in Radiance, an optimized CPU code *dctimestep_cpu* and a GPU code *dctimestep_gpu*. They were compared by conducting daylighting simulation using two kinds of sky vectors. The first type of sky vectors used Tregenza sky discretization (Tregenz 1987) and had 146 elements. The second used Reinhart sky discretization (Reinhart 2001) and had 2306 elements. A finer sky subdivision provides more accurate results, but is more time consuming during matrix multiplication stage. Table 2 lists the matrices used in our study. We used hourly data for sky vectors and the n in M_{VS} defines the simulated period, which varies from a day (n = 24) to a year (n = 8760). The approach of filtering zero sky vectors was applied to all programs. It was performed in a bash script for *dctimestep* and in C code for *dctimestep_cpu* and *dctimestep_gpu*.

Table 2 Dimensions of matrices used in numerical experiments

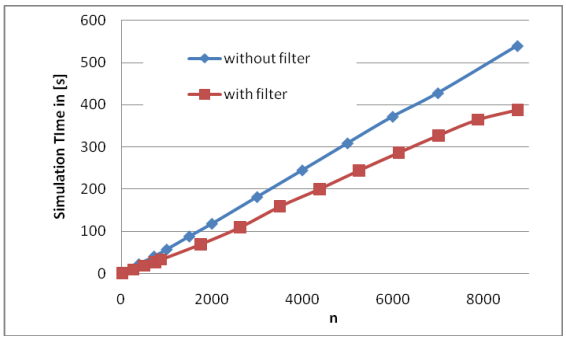
MATRIX	SETTING 1	SETTING 2
M_V	64 × 145	64 × 145
M_T	145 × 145	145 × 145
M_D	145 × **146**	145 × **2306**
M_{VS}	**146** × n	**2036** × n
M_{VI}	64 × n	64 × n

Computing Time

Figure 3 compares the computing time of *dctimestep* with and without filtering zero sky vectors. The time is elapsed time measured by *time* command in Linux. It includes the time to run *dctimestep* as well as the time to filter the zero vectors from M_{VS} and insert zero vectors into M_{VI} that are performed by a bash script in Linux. When n = 8760, filtering zero vectors can reduce the total simulation time by 45% for setting 1 and 28% for setting 2.



(a) Setting 1



(b) Setting 2

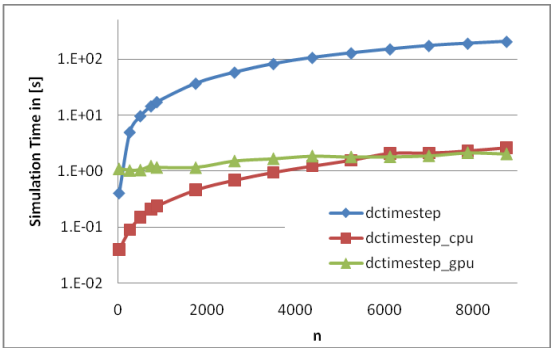
Figure 3 Comparison of simulation time used by dctimestep with and without filtering zero daylighting vectors

Because *time* command was used to measure the time for *dctimestep*, we also used it to measure *dctimestep_cpu* and *dctimestep_gpu* for consistency. As we will see later, the elapsed time includes both time spending on overhead in Linux and computing time used by the program.

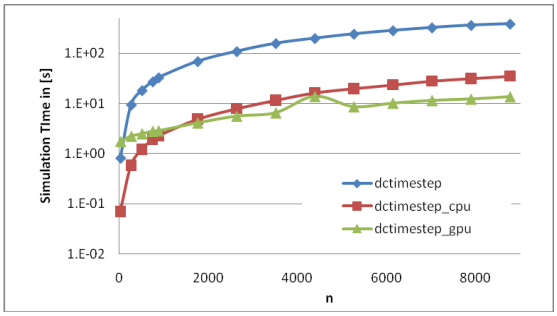
Figure 4 compares the simulation time used by different programs for various n in two settings. *dctimestep_cpu* is always faster than *dctimestep*. This indicates that the optimization for sequential code can reduce the computing time. On the other side, using GPU does not necessarily speed up the simulation. *dctimestep_gpu* is slower than *dctimestep_cpu* when $n < 5000$ in setting 1 and $n < 1000$ in setting 2. Because the GPU needs more time for initialization than CPU does, its advantage in computing speed can only be seen when computing demand is large enough.

Figure 5 compares the speed ratio of different programs. The speed ratio is defined as the ratio of computing speed of two programs, which is a multiplicative inverse of ratio of computing time. For setting 1, *dctiemstep_cpu* is from 10 × (n = 24) to 86.9 × (n = 4380) faster than *dctimestep* (plotted as "S1 cpu"). The ratio is 78.8 for annual simulation (n = 8760). On the other hand, the ratio of *dctimestep_gpu* to *dctimestep* (plotted as "S1 gpu") increases from 0.36 (n = 24) to 101.7 (n = 8760).

For setting 2, the speed enhancement by *dctimestep_cpu* is from 11.0 × (n = 8760) to 16.0 × (n = 264). The *dctimestep_gpu* has better performance than *dctimestep_cpu* with a speedup of 28.6 × at n = 8760.



(a) Setting 1



(b) Setting 2

Figure 4 Comparison of computing time used by different programs

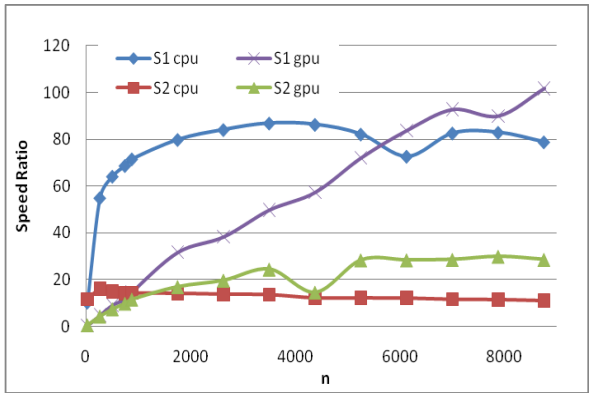


Figure 5 Comparison of speed ratios between the optimized codes and original code. S1: Setting 1, S2 Setting 2; cpu: dctimestep_cpu/dctimestep; gpu: dctimestep_gpu/dctimestep

To see the performance change by switching from CPU to GPU, Figure 6 compares the speed ratio of *dctimestep_cpu* and *dctimestep_gpu*. For setting 1, the GPU code is slower than CPU code when n < 5000 and there is only a speedup less than 1.5 × when n > 6000. For setting 2, GPU code is faster than CPU code when n > 1000. The speedup reaches 2.5 × when n > 6000. When n = 8760, the ratios are 1.29 for setting 1 and 2.60 for setting 2.

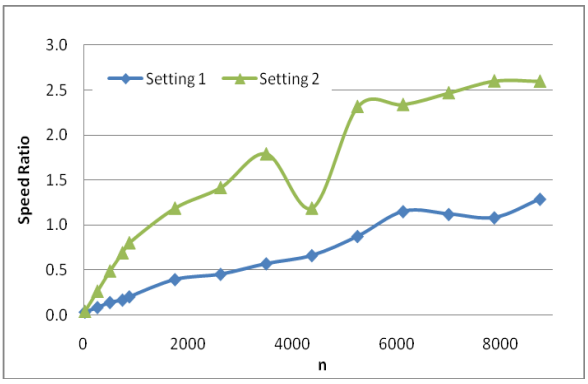


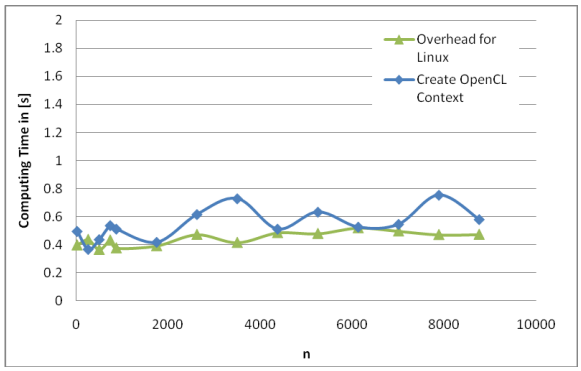
Figure 6 *Comparison of speed between the dctimestep_cpu and dctimestep_gpu. The ratio is speed of GPU code divided that of CPU*

Figures 5 and 6 also indicate some oscillations in the performance of GPU code. To find the course of oscillations, we measured the detailed usage of computing time by *dctimestep_gpu*:

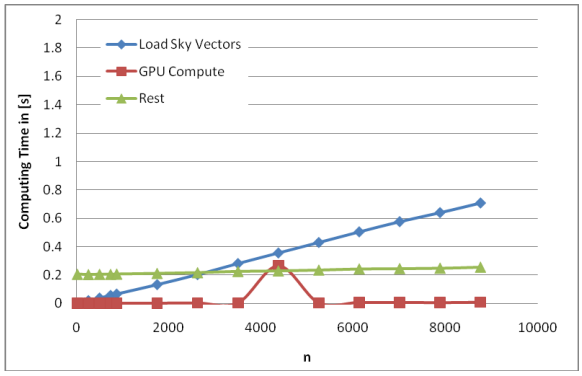
$$\begin{aligned} \textit{Elapsed Time} = {} & \textit{Overhead for Linux} \\ & + \textit{Load Sky Vectors} \\ & + \textit{Create OpenCL Context} \\ & + \textit{GPU Compute} \\ & + \textit{Rest}, \end{aligned} \tag{8}$$

where *Overhead for Linux* is the time for Linux system to invoke the *dctimestep_gpu* and release it after the execution is over. It is the difference between elapsed time measured by Linux *time* command and the program running time measured by a C function gettimeofday( ) in *dctimestep_gpu*. *Load Sky Vectors* is the time used to read the matrix M_{VS} from hard disk drive to CPU memory. *Create OpenCL Context* is the time to create OpenCL context by running a OpenCL function clcreateContext( ), which creates a OpenCL context for a device. clcreateContext( ) is a part of the initialization process for computing on device using OpenCL. The OpenCL runtime uses context for managing objects such as command-queues, memory, program and kernel objects and for executing kernels on devices specified in the context. *GPU Compute* is the time to conduct computing on GPU. *Rest* is the time used by other parts of the program, including reading and writing other matrices, filtering the zero vectors, transferring data between CPU and GPU, initializing the GPU hardware and OpenCL programming environment except clcreateContext( ).

Figure 7 shows the details in computing time by *dctimestep_gpu* for setting 1. The *Overhead for Linux* and *Create OpenCL Context* (Figure 7a) use significant amount of time. For instance, they count for 90% of total time when M_{VS} is a 146 × 24 matrix. We also found that the time spent on *Overhead for Linux* and *Create OpenCL Context* was random. They are not related to size of matrices and not repeatable for different program runs with the same matrices. They contribute the most to the oscillations of simulation time. The computing time reported in this paper was averaged over 20 program runs.



(a)

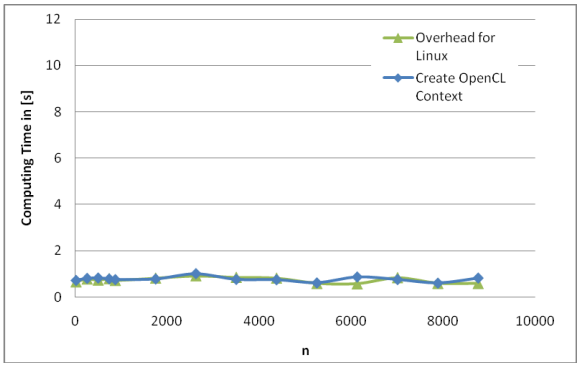


(b)

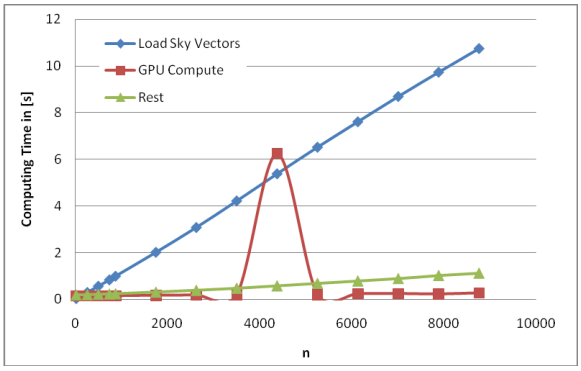
Figure 7 A detailed analysis of computing time used by GPU for setting 1 with the 146 × n matrix M_{VS}

The time for *Load Sky Vectors* continuously increases when the size of M_{VS} increases and it becomes significant when M_{VS} is large. For instance, when M_{VS} is a 146 × 8760 matrix, it needs 0.696 s to load M_{VS}, which is the biggest part in simulation time. The GPU computing time (*GPU Compute* in Figure 7b) is less than 0.01s except n = 4380, where the matrix actually computed is 146 × 2347 after filtering the zero columns. The sudden increase in computing time is repeatable and they are caused by difficulty in transferring data between GPU memory and processors for certain dimensions of array. Similar phenomenon was also observed in other work (Zuo and Chen, 2010). The time used by other features of the program is about 0.2 s (*Rest* in Figure 7b). It is increasing with the size of M_{VS} because the program needs more time to pre-process and transfer data when M_{VS} is getting bigger.

With a larger matrix (2306 × n), *Load Sky Vectors* becomes the dominant part of GPU computing time in setting 2 (Figure 8b). For instance, it counts for 79% time when M_{VS} is a 2306 × 8760 matrix. Time for Linux overhead and creating OpenCL context still changes randomly. However, they are less than 2s and are less important in total simulation time. The GPU computing time is less than 0.3 s except n = 4380.



(a)



(b)

Figure 8 A detailed analysis of computing time used by GPU for setting 2 with the 2306 × n matrix M_{VS}

Accuracy

To compare the accuracy of the new codes, we calculated the relative error $e_{i,j}$ for each entry in M_{VI}:

$$e_{i,j} = (y_{i,j} - y_{refi,j}) / y_{ref,i,j}, i = 1,.., 64, j = 1, .., n \quad (9)$$

where $y_{i,j}$ is an entry in M_{VI} calculated by *dctimestep_cpu* or *dctimestep_gpu* and $y_{ref,ij,}$ is the reference value computed by *dctimestep*. If $y_{ref,ij} = 0$, $e_{i,j}$ will be computed by the following equation:

$$e_{i,j} = 10^7 y_{i,j}, i = 1,.., 64, j = 1, .., n \quad (10)$$

We compared the distribution of $e_{i,j}$ when n = 8760. The zero columns in M_{VI} are not counted in comparison since they are not computed. M_{VI} has 4590 non-zero columns so the number of total entries in comparison is 64 × 4590 × 3 = 881,280. The comparison shows that the *dctimestep_cpu* code could produce identical results as that by the *dctimestep*. The results by *dctimestep_gpu* were slightly different from the one by CPU codes. As shown in Figure 9, the relative errors for both settings are in a range of -1×10^{-5} to 1×10^{-5}. With more computing efforts, more data in setting 2 has larger relative errors than setting 1.

DISCUSSION

To further speed up *dctimestep_gpu*, the key is to load M_{VS} more efficiently. For instance, it is possible to reduce the I/O time if one uses solid state drive, which is significantly faster than traditional hard disk drive used in the current study.

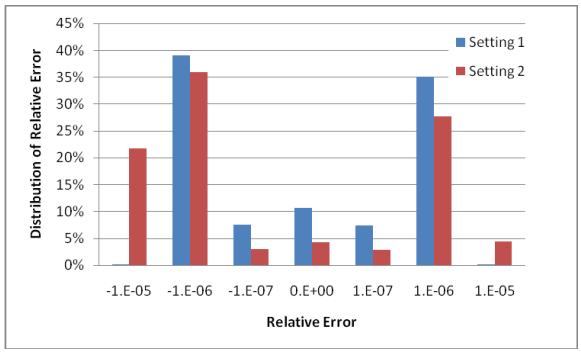


Figure 9 Distributions of relative errors of GPU results compared to CPU results when n = 8760

Using GPU does not have much advantage when computing demand is small. As we have seen, the initialization of GPU and overhead for Linux system can count for a large portion of total GPU simulation time for setting 1. To reduce the influence of initialization, we should run a larger simulation job which requires longer simulation time, such as we did for setting 2.

To avoid the sudden increase in GPU computing time, such as when M_{VS} is a 146 × 2347 matrix, we should optimize the data flow in GPU. Furthermore, we can achieve higher performance by optimizing the implementation of GPU code for specific GPU hardware to fully utilize the capacity of GPU hardware (Volkov and Demmel 2008). Since Radiance is widely used software and may run on many hardware configurations, it will need too many resource to optimize Radiance for different hardware platform. Thus, one must balance one's choice between performance and compatibility.

CONCLUSION

By optimizing the *dctimestep* code and running it in parallel on a GPU, we have accelerated the annual daylighting simulation of Radiance by two orders of magnitudes. The optimization of the algorithm can speed up an annual daylighting simulation on a CPU by a factor of 86.9 and 11 using sky vectors with 146 and 2306 elements, respectively. Running in parallel on a GPU using OpenCL can further accelerate the simulation by a factor of 1.29 and 2.60. This leads to total speed-ups of factor 101.7 or 28.6.

As a pilot study, we only tested the OpenCL code on one GPU hardware. Since OpenCL is a cross-platform language, it would be interesting to evaluate the performance of the same code on different hardware platforms.

ACKNOWLEDGEMENT

This work was supported by the Assistant Secretary for Energy Efficiency and Renewable Energy, Building Technologies Program of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 and by the California Energy Commission through its Public Interest Energy Research (PIER) Program on behalf of the citizens of California.

The authors would also like to thank Amir Roth from the U.S. Department of Energy for his constructive comments on this paper.

REFERENCES

Armbrust, M., A. Fox, et al. (2009). "Above the Clouds: A Berkeley View of Cloud Computing," Electrical Engineering and Computer Sciences, Technical Report No. UCB/EECS-2009-28, University of California at Berkeley.

Chapman, B., G. Jost, et al. (2007). Using OpenMP: Portable Shared Memory Parallel Programming, The MIT Press.

Gregg, C., and K. Hazelwood (2011). "Where is the Data? Why You Cannot Debate GPU vs. CPU Performance Without the Answer," International Symposium on Performance Analysis of Systems and Software (ISPASS), Austin, TX.

Gropp, W., E. Lusk, et al. (1999). Using MPI: Portable Parallel Programming with the Message-Passing Interface, The MIT Press.

Hasama, T., S. Kato, et al. (2008). "Analysis of Wind-induced Inflow and Outflow through a Single Opening using LES & DES." Journal of Wind Engineering and Industrial Aerodynamics **96**(10-11): 1678-1691.

Larson, G. W. and R. A. Shakespeare (1998). Rendering With Radiance: The Art And Science Of Lighting Visualization, Morgan Kaufmann Publishers.

Mazumdar, S. and Q. Chen (2008). "Influence of Cabin Conditions on Placement and Response of Contaminant Detection Sensors in A Commercial Aircraft." Journal of Environmental Monitoring **10**(1): 71-81.

Munshi, A. (2010). The OpenCL Specification Version 1.1, Khronos OpenCL Working Group.

NVIDIA (2007). NVIDIA CUDA Compute Unified Device Architecture-- Programming Guide (Version 1.1). Santa Clara, California, NVIDIA Corporation.

NVIDIA (2011). "NVIDIA OpenCL SDK Code Samples." Retrieved on 01/26/2011, from http://developer.download.nvidia.com/compute/opencl/sdk/website/samples.html.

Reinhart, C.F. (2001). "Daylight Availability And Manual Lighting Control in Office Buildings: Simulation Studies and Analysis of Measurement," Ph.D. thesis, Department of Architecture, Technical University of Karlsruhe.

Tregenza, P.R. (1987). "Subdivision of The Sky Hemisphere For Luminance Measurements." Lighting Research & Technology, 19, 13-14.

Volkov, V. and J.W. Demmel, (2008). "Benchmarking GPUs to Tune Dense Linear Algebra." 2008 ACM/IEEE Conference on Supercomputing (SC08).

Ward Larson, G. and Shakespeare, R. (1998). "Rendering with Radiance: The Art and Science of Lighting Visualization." San Francisco, Morgan Kaufmann.

Ward, G., Mistrick, R., et al. (2011). "Simulating the Daylight Performance of Complex Fenestration Systems Using Bidirectional Scattering Distribution Functions Within Radiance." Technical report LBNL-4414E, Lawrence Berkeley National Laboratory.

Wenisch, P., C. v. Treeck, et al (2007). "Computational Steering on Distributed Systems: Indoor Comfort Simulations as A Case Study of Interactive CFD on Supercomputers." International Journal of Parallel, Emergent and Distributed Systems, **22**(4): 275-291.

Zuo, W. and Q. Chen (2010). "Fast and Informative Flow Simulations in A Building By Using Fast Fluid Dynamics Model on Graphics Processing Unit." Building and Environment, **45**(3): 747-757.