MODELING OF HEAT DIFFUSION THROUGH ISOTROPIC MEDIA USING GRAPHICAL PROCESSING UNITS

C. A. Nahas, C.V. Krishnamurthy, Prabhu Rajagopal and Krishnan Balasubramaniam
Centre for NDE, Indian Institute of Technology - Madras, Chennai 600036, T.N., India

ABSTRACT. With accurate simulation of ever-more complex phenomena desired, numerical models are getting increasingly sophisticated and thus take considerable amount of time to run on normal computers. Parallel computing has emerged as an important paradigm in response, allowing engineers to run programs faster. In recent years graphics processing units (GPU) are among the massively parallel devices widely available to the commodity market. General purpose computing on these devices became mainstream after the introduction of NVIDIA ‘compute unified device architecture’ (CUDA). In this paper, we develop CUDA implementations of the finite difference time domain (FDTD) scheme for two- and three-dimensional heat diffusion through homogenous media, a phenomenon of much interest to the NDE community. Simulations are run on the commodity GPU GeForce 9800GT which has 112 CUDA cores. Results show a strong improvement in the speed of computation as compared to that of a serial implementation on a CPU. We also discuss factors that can improve accuracy and speed of GPU based computation.

Keywords: GPU Computing, CUDA, Heat diffusion

INTRODUCTION

From the introduction of microprocessors to early 2000’s programmers relied on hardware vendors in order to increase the speed of their programs. In order to speed up a program they relied upon newer and faster processors. This was because software was written serially and computers were working serially. To solve a problem, an algorithm was constructed and implemented as serial instructions. These instructions were executed on a central processing unit on one computer. Hardware manufactures increased the frequency of the processors that decreased the average time taken to execute an instruction to build powerful computers. As the frequency of each processor increased, the power consumed by these chips increased dramatically. Further, cooling the computer also became a major issue. The microprocessor giant Intel cancelled its ‘Tejas’ and ‘Jayhawk’ processors due to concerns with excessive heat generation in 2004, highlighting its transition from focusing on single- to multi-core processors.
Manufactures realized that computers cannot become faster any more: instead they have to grow wider. Instead of a very fast single core chip two or more cores of moderate speed started to appear in computers. This system has its problems too. Serially coded software does not run faster in wider computers: programs have to be ‘parallelized’. In parallel computing a problem is solved using multiple processing units simultaneously. The concept of parallel programming however, is not new to the computer world. From 1970 onward vector processors were running parallel programs in high end systems like supercomputers and high performance computing systems. With the introduction of multi-core processors in desktop PCs, parallel programs became the mainstream paradigm in day-to-day information processing.

Graphical processing units (GPU) were introduced to personal computers much before multi core CPUs. GPUs are processors dedicated to accelerate building images in the frame buffer intended for output display. These processors are highly parallel and multithreaded, and are much more efficient at handling processes that include large amount of data and single instructions. Their high capacity for single instruction multiple data algorithms made them suitable for video games and high quality video playback and processing systems. In a GPU more numbers of transistors are devoted to data processing rather than data caching and flow control. Thus GPUs are well suited for data parallel algorithms with high floating point operations. Single precision floating point arithmetic was added to mainstream consumer end graphics cards. Soon programmers started to realize that multi core GPUs can be used even for non-graphical algorithms that are of the nature single instruction multiple data. Using shading language application programming interfaces (APIs), programmers attempted to harvest the massively parallel GPU for general purpose scientific and engineering computations.

In late 2006 NVIDIA introduced its new parallel computing architecture called ‘compute unified device architecture’, or CUDA [1]. With the introduction of CUDA programming GPUs for general purpose computing become much more affordable and accessible to the average programmer. As a cross platform programming interface, OpenCL [2] was added to the scenario in 2008. But CUDA already took the lead in scientific and high performance computing world by providing a mature compiler, a better documentation, performance libraries and high end graphics cards like Tesla that fully dedicated for computational purpose only.

THE CUDA PROGRAMMING MODEL

CUDA extends the standard C/C++ languages to make an interface to the GPUs. Normal programs are divided into kernels and host code where kernels are compiled and executed on the GPUs and host code is executed on the CPU. Kernels are the ‘compute intense’ portion of the program whereas host code sets an environment for the program. Kernels are C functions that are executed N times in parallel by N different CUDA ‘threads’. Each process in a kernel is treated as a thread. These threads are grouped into ‘blocks’ which are further grouped into ‘grids’. Each thread has its own private memory and a shared memory visible to all threads in the same block. Apart from these two types there exists a third memory known as global memory, which can be accessed by any thread in the kernel and is the main memory for computations.

As the number of cores in CPUs and GPUs grows the challenge is to create a program that scales automatically when we update the hardware. Since CUDA groups threads into blocks, these blocks can be executed on any processing core without disturbing any other block in the grid. Thus this programming model is extremely scalable.
CUDA programs are executed similarly to other programs. Host code start execution from CPU and kernel executes on GPU. At the end of GPU execution control is given back to CPU. The NVIDIA CUDA Compiler is used to compile the kernel and a normal C compiler such as Microsoft Visual C++ is used to compile the host code.

In this paper, using CUDA, we develop parallel finite difference scheme as applicable to one of the areas of interest to NDE community, namely heat diffusion through isotropic media.

**CUDA IMPLEMENTATION OF 2D HEAT DIFFUSION**

The heat equation is a second order partial differential equation describing the distribution of heat over time over a given region. For a temperature function \( Q(x,y,t) \) of two space variables and time variable \( t \), the heat equation is

\[
\rho c \frac{\partial Q}{\partial t} = k_x \frac{\partial^2 Q}{\partial x^2} + k_y \frac{\partial^2 Q}{\partial y^2}
\]

(1)

Where \( \rho \) is the density \( c \) is the specific heat and \( k_x \ k_y \) are the thermal diffusion coefficients in \( x \) and \( y \) directions. Discretizing the equation with forward time centered space scheme gives the following form:

\[
Q_{ij}^{n+1} = (1 - 2\alpha_x - 2\alpha_y)Q_{ij}^n + \alpha_x(Q_{i+1,j}^n + Q_{i-1,j}^n) + \alpha_y(Q_{ij+1}^n + Q_{ij-1}^n)
\]

(2)

Where \( \alpha_x = \frac{k_x \Delta t}{\rho c \Delta x^2} \) and \( \alpha_y = \frac{k_y \Delta t}{\rho c \Delta y^2} \)

In the above equation \( Q_{ij}^n \) represent temperature at point \((i, j)\) of \(n^{th}\) time step. The whole space at \(n^{th}\) time step is represented using a two dimensional matrix. To increase the efficiency the two-dimensional matrix is then decomposed into a one dimensional array and stored in memory. In sequential computing each array element is updated one after other using a space loop. The space loop is then enclosed in a time loop. In parallel computing the space loop can be decomposed into CUDA kernels which then update the whole space in one go. Since each time step depends on the previous step, the time loop cannot be parallelized. The block size of the kernels is calculated using CUDA occupancy calculator. For maximum efficiency block size is fixed as 16 X 16. Hence 256 threads are available in one block. Grid size is calculated according to the size of the problem domain. If you have \( M \) rows and \( N \) columns then grid width will \( N/\text{block width} \) and grid height will be \( M/\text{block height} \)

**SYSTEM SPECIFICATION**

A normal desktop PC was used. The Intel processor core i3-530 clocked at 2.93 GHz with 3.24GB RAM was used as the host platform. The NVIDIA GeForce 9800GT with 112 CUDA cores clocked at 1.37 GHz and 512 MB ram was used as the device. Windows 7 32 bit professional edition was the operating system used.
PROBLEM SPECIFICATION

Heat diffusion from the center of an aluminium plate of size 15mm x 15mm with specific heat 0.9J/K, thermal conductivity 237 W/m-K, and density of 2700 kg/m3 was chosen as a sample problem. The square plate was divided into a grid of 300 X 300 and the time step chosen was 0.005 second. For the first 200 time steps the temperature at the center of the plate was kept constant at 370K while the ambient temperature was 70K. The heat source was a line of 10 horizontal nodes. After 200 cycles the heat source was removed from the plate and the finite difference time loop was iterated 3000 times.

RESULTS

The CPU implementation took about 2180 milliseconds to complete while the GPU run took just 16 milliseconds. Results obtained from CPU and GPU were analyzed and the A-scan was plotted for randomly selected points in the grid, for comparison. Figure 1 shows the A-scan plot at the grid point (130, 130). We observe a very good agreement between the GPU and the CPU run results. The authors have a concern that the small difference at the peak value could be due to the fact the GeForce 9800GT used in this study is not IEEE754 compliant processor [3], and currently working on this issue.

To understand the speed up factor, the above problem was repeated with different grid sizes and the ratio of the time taken for a CPU run to that of a GPU run (time ratio) is...
plotted against the grid size as shown in Figure 2. We observe that as the grid size increases, the GPU begins to provide a very strong improvement in computation time.

**CUDA IMPLEMENTATION OF 3D HEAT DIFFUSION**

The heat equation in three-dimensional form is given as follows:

$$\rho c \frac{\partial Q}{\partial t} = k_x \frac{\partial^2 Q}{\partial x^2} + k_y \frac{\partial^2 Q}{\partial y^2} + k_z \frac{\partial^2 Q}{\partial z^2}$$  \hspace{1cm} (1)

where again, $\rho$ is the density, $c$ is the specific heat, and $k_x$, $k_y$, $k_z$ are the thermal diffusion coefficients in $x$, $y$ and $z$ directions. Discretizing the equation with the forward time centered space scheme gives the following form:

$$Q_{ijk}^{n+1} = (1 - 2\alpha_x - 2\alpha_y - 2\alpha_z)Q_{ijk}^n + \alpha_x(Q_{i+1jk}^n + Q_{i-1jk}^n) + \alpha_y(Q_{ij+1k}^n + Q_{ij-1k}^n) + \alpha_z(Q_{ijk+1}^n + Q_{ijk-1}^n)$$  \hspace{1cm} (2)

Where $\alpha_x = \frac{k_x \Delta t}{\rho c \Delta x^2}$, $\alpha_y = \frac{k_y \Delta t}{\rho c \Delta y^2}$ and $\alpha_z = \frac{k_z \Delta t}{\rho c \Delta z^2}$.

GPU implementation of the three-dimensional finite difference scheme needs a different strategy from that taken for the two-dimensional case. Here, the extra dimension of space is treated as slices of planes. The Block size is fixed as 16 X 16 for maximum efficiency. As for the two-dimensional case, 256 threads are available in one block. The grid size is calculated according to the size of the problem domain: for $M$ rows, $N$ columns.
and Z number of slices, the grid width will be \((N/\text{block})XZ\) and grid height will be \((M/\text{block height})\). The computation process is similar to that for the two-dimensional case.

**SAMPLE PROBLEM AND RESULTS**

The same problem chosen for the two-dimensional case was taken here with some modification. Here the aluminium plate has a thickness of 2 mm, divided into 40 grids. Hence the dimension of grid became 300 X 300 X 40. Instead of a line source an entire area was heated up. All the grid points in one side, which is 15mmX2mm area was heated up to 200 cycles. Rest of the parameters of the problem was kept the same. The serial C-based CPU run took about 35000 milliseconds while GPU run clocked 150 milliseconds. Figure 3 again shows a comparison of the A-Scans obtained from the CPU and the GPU runs at a point (50,150) on the surface of the plate. We again observe a very good agreement in the result.

![Figure 3](image)

**CONCLUSIONS**

Two-dimensional and three-dimensional heat diffusion through isotropic media was successfully implemented in a GPU using CUDA. GPU computing is very effective to increase the computational speed of data parallel finite difference schemes. With the present hardware, accuracy of the computation is up to single precision only. Currently we are studying the three-dimensional implementation of the heat diffusion equation more extensively. Future work is to extend the CUDA implementation to other areas of interest of NDE community such as elastic wave propagation.
REFERENCES

1. NVIDIA CUDA C Programming Guide Version 4.0 (5/6/2011)  
   http://developer.nvidia.com/category/zone/cuda-zone
2. The OpenCL specification, Version 1.1, revision 44(6/1/2011)  
   http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf
3. N. Whitehead and A. Fit-Florea, Precision & Performance : Floating Point and IEEE 754  
4. B. Deschizeaux and Jean-Yves Blanc, GPU Gems 3, edited by Hubert Nguyen, chapter 38,  
   Addison-Wesley, Boston, 2008, pp. 831-850.
5. Lubomir Riha and Radislav Smid, Computer Standards & Interfaces, Article in press  
   available at sciencedirect as Acceleration of acoustic emission signal processing algorithms  
   using CUDA standard.