

Diffusion Equation Solver with CUDA Implementation

Problem description:

Partial differential equations are commonly used to model the flow of material in and out of a given region. Continuity states that matter cannot be created, or that matter entering a region must equal that leaving. In other words,

$$\frac{\partial \varphi}{\partial t} + \nabla \cdot j = 0$$

where $\varphi(r, t)$ is the material density at location r , time t , and j is the flux of the diffusing material.

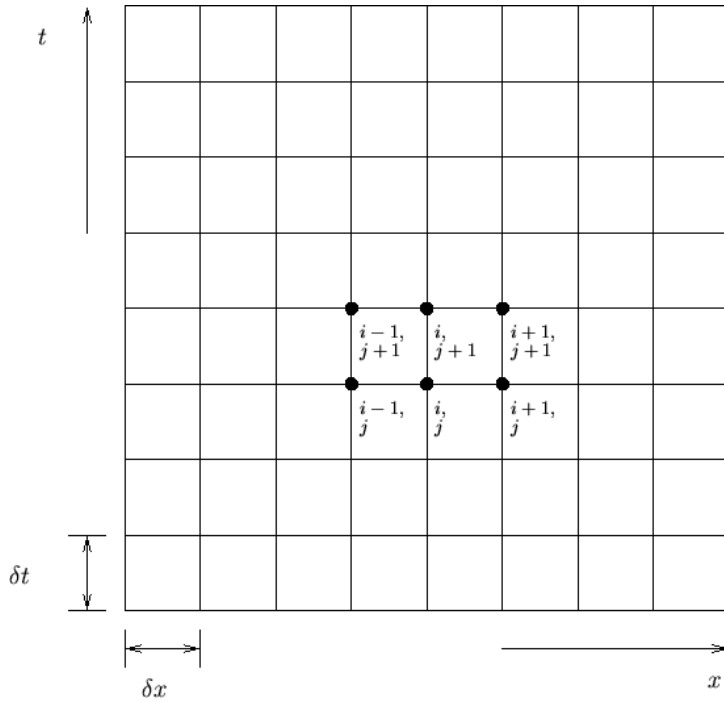
Assuming that the flux is proportional to the local density gradient gives what is commonly referred to as the **diffusion equation**

$$\frac{\partial \varphi(r, t)}{\partial t} = \nabla \cdot [D(\varphi, r) \nabla \varphi(r, t)]$$

where $D(\varphi, r)$ is the diffusion coefficient for density φ at location r . When $D(\varphi, r)$ is constant, this gives the **heat equation**.

Upon applying boundary conditions, such as fixed temperatures along edges, no flow through walls, etc, real life problems are often impossible to solve analytically, which leads to importance of finding a numerical approximation for these equations. Two approaches are commonly used; finite differences and finite elements. This report will adopt the finite difference approximation approach, which performs the same calculations multiple times, over a region that has been partitioned into a regular grid. The fixed spacing of the grid, and repetition of the calculations, lends itself well to parallelization.

The diffusion equation is a partial differential equation, and the method of finite differences approximates these values averaging values of immediate neighbors on the grid. This applies to moving through time, as well as spatial directions. For example, consider the grid below



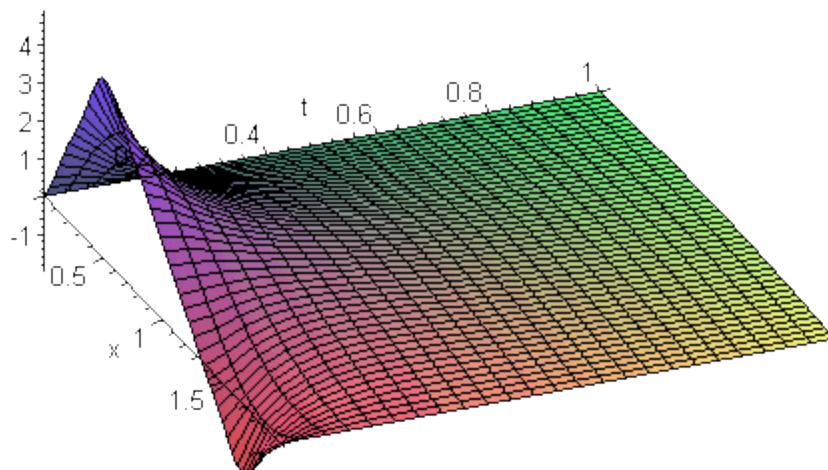
Partial derivatives can be approximated by averaging values at neighboring points, divided by the step size in time or space:

$$\frac{\partial \varphi}{\partial x} \approx \frac{\varphi_{i+1,j} - \varphi_{i-1,j}}{\delta x}$$

$$\frac{\partial \varphi}{\partial t} \approx \frac{\varphi_{i,j} - \varphi_{i,j-1}}{\delta t}$$

$$\frac{\partial^2 \varphi}{\partial x^2} \approx \frac{\varphi_{i+1,j} - 2\varphi_{i,j} + \varphi_{i-1,j}}{(\delta x)^2}$$

Below is a diagram of a typical solution for the 1-D Heat Equation. For this example, a thin rod (ie. Only considering one dimension) has an initial even temperature distribution, when both ends are placed in ice water. Assume the rod is insulated, so no heat is escaping into the air, just being transferred along the rod. As time marches on, the heat gradually dissipates to an even temperature. This is the problem that will be solved using a serial approach, then with parallelization.



Basic serial approach:

The problem solution involves constant recalculation of all points until the desired time is reached. A serial Fortran77 code was used to calculate values for the 1D diffusion equation using various grid spacing to test for scalability. Below is the portion on the code which recalculates the point values at each time step:

```
do i = 1, n
  h_new(i) = h(i) &
    + ( time_delt * k / x_delx / x_delx ) &
    * ( h(i-1) - 2.0 * h(i) + h(i+1) ) &
    + time_delt * BC ( x(i), time )
end do
```

Initial boundary conditions are applied to represent the initial temperature of the rod, with the two temperatures at both ends starting at freezing.

Parallel approach and hardware considerations:

To solve any kind of realistic problem, to the level of detail desired, requires a huge amount of repetitive calculations. The method becomes unstable if you attempt to step through time too quickly. Also, when considering the design of an airplane wing, designers are interested in data over as many points as possible along the surface of the wing, creating a very fine spatial grid of points. Since the values at each point only depend upon the values of their immediate neighbors, this problem lends itself well to parallelization. Simply breaking up the region into equal portions allows for calculations to be done concurrently. Potential difficulties arise with points on the boundaries, since to calculate point value on one thread, you need knowledge of point values on another thread. However, unlike MPI or hyperthreading, GPU threads access a shared memory.

One of the largest manufacturers of graphics cards, NVIDIA, has developed a library of commands to enable a programmer to access the parallel capability of their second generation

graphics cards for mathematical calculations. Operations still need to go through the computer's CPU, and only those by Intel currently support this technology (AMD chips do not.) These graphics cards also use more power, so a minimum 600W power supply is needed. Since this problem is commonly solved, a very complete public domain solution already exists. Even just implementing this program took a considerable amount of work, but it was used to demonstrate the potential of CUDA. For this report, a GeForce GT 220 NVIDIA graphics card was used, on a computer with a dual core Intel Celeron processor 450.



Code:

NVIDIA researchers developed the open-source product OpenCurrent that was used in this project. Given my unfamiliarity with C++, my code didn't show nearly as much performance improvement, and would not justly show what CUDA is capable of. There are a few things which need to be loaded in advance before this code can be run.

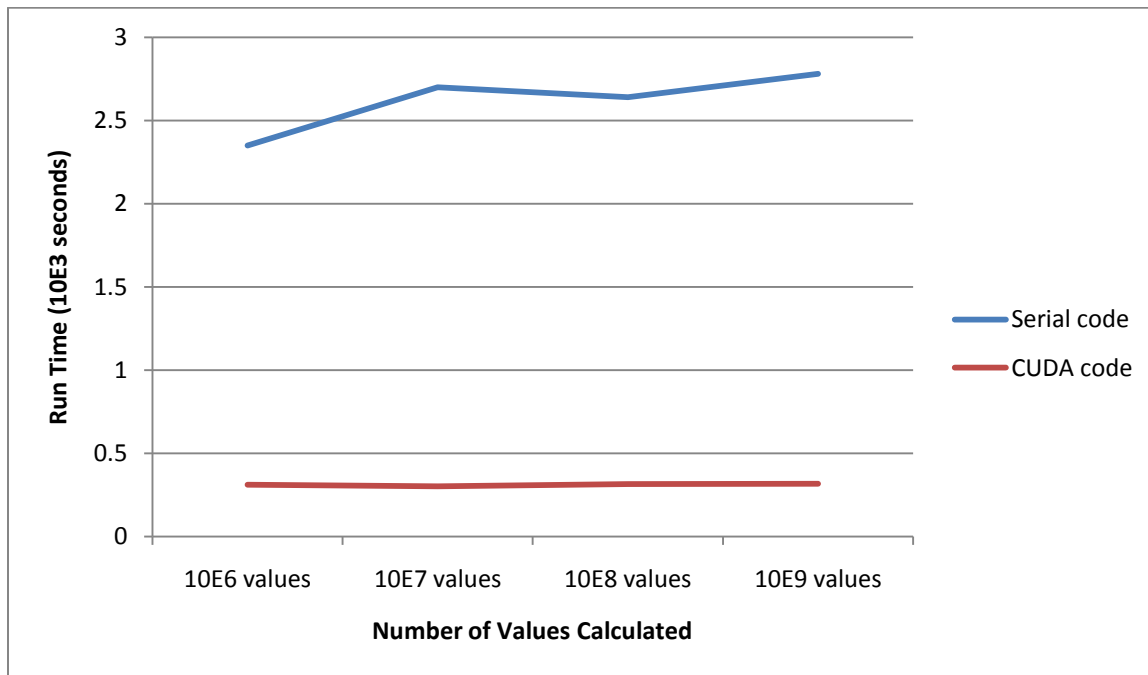
- 1) CUDA 2.3 must be loaded (available from the NVIDIA website, http://www.nvidia.com/object/cuda_get.html) as well as an NVIDIA graphics card driver for any of their second generation graphics cards. From the NVIDIA website, you simply download the CUDA cgtoolkit.
- 2) CMake 2.6.4 or later <http://www.cmake.org/>
- 3) NetCDF can be downloaded from <http://www.unidata.ucar.edu/software/netcdf/>

There are a variety of test cases included within the OpenCurrent programs, including one for the one dimensional heat equation. This was run for the as described in the next section.

Performance:

Calculations were done at 10^4 points along a rod which is 1 meter long. To create a matrix large enough to show any real difference between solution speeds, both the serial and parallel codes were over 10^2 , 10^3 , 10^4 , and 10^5 points.

	Serial Code run time (sec)	CUDA code run time (sec)
10^6 values being calculated	2.35×10^{-3}	3.10×10^{-4}
10^7 values being calculated	2.70×10^{-3}	3.02×10^{-4}
10^8 values being calculated	2.64×10^{-3}	3.15×10^{-4}
10^9 values being calculated	2.78×10^{-3}	3.16×10^{-4}



In general, using the CUDA code gave almost a factor of 10 increase in speed. However, this was obtained using optimized CUDA code, which is difficult to create, and for a problem that lends itself very well to parallelization.

Scalability:

It is definitely worth pointing out that increasing the size of a problem has virtually no effect on the amount of time it takes the CUDA code to run. Graphics cards are designed with an almost unlimited number of cores. Additionally, since graphics cards work off a shared memory, there are no latency issues from having to switch between different levels of cached memory. The run time for the GPU code was almost unchanged when the problem size increased, whereas the CPU code runtime adjusted sporadically.

Lessons learned, level of difficulty:

CUDA programming is definitely not for beginners. Even the experts agree that the appropriate choice of kernel size, number of blocs, number of threads per block, etc, are mainly done through trial and error or prior experience. The commands being used are not intuitive, and currently the CUDA library is only for use with C programs. As a novice C programmer, I was unable to see any performance difference, and had to instead really on code provided by NVIDIA. Naturally they chose a problem that would highlight the advantages of the product the most, and a quasi factor of ten performance increase is noteworthy.

The hardware requirements for this are not trivial, but thanks to the gaming industry, are on their way to becoming mainstream. When I attempted to replace my original power supply with a 600W power supply that would support the GT 220 graphics card, I found my (8 year old) motherboard would not support it. When replacing a motherboard, it's time to consider replacing the computer, and the latest eMachine being sold at MicroCenter for \$319 already has an NVIDIA GeForce graphics card as standard.

Future of CUDA:

In June 2009, the Portland Group and NVIDIA announced they were working together to create CUDA Fortran. It has since been released, and is now for purchase. This was created as a variation of the Fortran 2003 compiler.

PGI CUDA Fortran Compiler

Graphic processing units or GPUs have evolved into programmable, highly parallel computational units with very high memory bandwidth. GPU designs are optimized for the computations found in graphics rendering, but are general enough to be useful in many data-parallel, compute-intensive programs common in high-performance computing (HPC).

CUDA™ is the architecture of the NVIDIA line of GPUs. Currently, the CUDA programming environment is comprised of an extended C compiler and tool chain, known as CUDA C. CUDA C allows direct programming of the GPU from a high level language.



PGI and NVIDIA announced in June 2009 that they are working in cooperation to develop CUDA Fortran. CUDA Fortran includes a Fortran 2003 compiler and tool chain for programming NVIDIA GPUs using Fortran. Available now, PGI 2010 includes support for CUDA Fortran on Linux, Mac OS X and Windows.

A free 15 day trial of CUDA Fortran is available now as part of the standard [PGI 2010 download packages](#). Installation and configuration information, along with the [CUDA Fortran Programming Guide and Reference](#), are included in the download packages. Either a current PGI license or a [PGI trial license](#) is required.

As a Fortran programmer, I am very interested in any CUDA libraries where I can try to understand the process a little better. I feel CUDA programming has a definite future, since NVIDIA is such a large graphics card manufacturer, and their product is so popular with the gaming community. While I was researching and buying my graphics card and power supply at Microcenter, four guys bought NVIDIA graphics cards to upgrade their machines and there were 3 full racks of cards. While the math coprocessor function will most likely remain secondary, we can take advantage of the increasing card performance. NVIDIA is working to make CUDA coding more user friendly, such as providing open source codes like I used on their website and developing CUDA Fortran. Still, it will need to become quite a bit more logical to program before it is commonly adopted.