A

REPORT

ON

RENDERING OF 2D HEAT EQUATION USING CUDA

ECE 5950-PARALLEL COMPUTING

By
Asha Ganesan (ag2248)
Deepa.G.M (dm722)
Sravya Chinthalapati (sc2655)

Project Advisor: Adam Bojanczyk

May, 2015

Table of Contents

1. Introduction

Parallel computing has developed a lot during the last years. In particular, computer scientists have tried to first increase the CPU clock rate of our computers, and then to increase the amount of actual CPUs. With this latter increase, they are faced with the problem of making them smaller and smaller, until their size becomes such that quantum effects get involved in the process and make their production impossible. In the mean time, people have begun to use GPUs for their calculations, which are known to contain thousands of processing units. For the scientific community, this way of proceeding was not optimal, because one had to translate the problems into graphical problems and then solve them by applying graphics functions (with OpenGL for example). As time went on, Nvidia has made available a new library named CUDA allowing for an easier way to work with GPUs. In this project, we have exploited the features of parallel computing in solving and rendering a 2-D heat equation using CUDA.

1.1 Motivation

Partial differential equations are commonly used to describe physical phenomena that continuously change in space and time. One of the most studied and well known of such equations is the *Heat Equation*, which mathematically models the steady-state heat flow in a region that exposes certain dimensionality, with certain fixed temperatures on its boundaries. The heat equation is a parabolic partial differential equation that describes the distribution of heat (or variation in temperature) in a given region over time .

Heat equation is one of the most important mathematical equations, which is widely applied in engineering applications such as groundwater simulation and reservoir modelling. Even with the significant advances made in computational algorithms for solving partial differential equations in scientific computation, to solve large-scale heat equation remains a challenge. The problem could arise from intensive computational efforts and large memory capacity required for solving large sparse matrix systems of discrete equations. Meanwhile, parallel computing has been proved to be an efficient way to address the primary limitations of single-processor computers and could significantly advance the modelling capability of researchers. Thus, research on parallel computing for heat equation has received considerable attention in recent years. Most of these works deal with improving stability and convergence rate of algorithms [5].

The heat equation is of fundamental importance in diverse scientific fields:

1. **Mathematics:** it is the prototypical parabolic partial differential equation.
2. **Probability theory:** it is connected with the study of Brownian motion via the Fokker–Planck equation.
3. **Financial mathematics:** it is used to solve the Black–Scholes partial differential equation.
4. The diffusion equation, a more general version of the heat equation, arises in connection with the study of chemical diffusion and other related processes [6].

1.2 Difference between CUDA and other parallel programming models

Traditionally, a CPU contains a few cores (arithmetic logical units ALU) that share their memory in a common cache and a common virtual memory (DRAM). On one single computer, it is possible to use shared or distributed memory models (with OpenMP or MPI). In the former case, each core of a same computer can communicate without any problems, while a latency occurs in the latter. Usually, we have a few cores per computer and, for large parallel applications, we link together a certain amount of computers to build a cluster, so that workload and memory are distributed over different computers. GPUs work a bit differently in the sense that a GPU consists of thousands of arithmetic logical units. These ALUs are grouped in blocks sharing the same memory. Memory cannot be accessed directly from one block to the other. Additionally, in the same way CPUs work with DRAM memory, GPUs work with a reasonably large amount of global memory. Depending on the user needs, the ALUs can be ordered 1, 2 or

3-dimensionally. Apart from global memory, a GPU has constant and texture memory that may affect performance if used appropriately.

2. Heat transfer model

2.1 Example for heat propagation

Assuming bar starts at x=0 and extends till x = L (length of the rod).
With an assumption that at any location *x,* the temperature will be constant at every point in the cross section at that *x*. In other words, temperature will only vary in *x* and we can hence consider the bar to be a 1-D bar. Note that with this assumption the actual shape of the cross section (*i.e.* circular, rectangular, *etc.*) doesn't matter. This is because the lateral surface can be assumed to be perfectly insulated, hence the heat has to travel in only one direction i.e., from left to right or right to left creating a 1D temperature distribution. Figure 1 below shows the temperature distribution along the rod with respect to time.

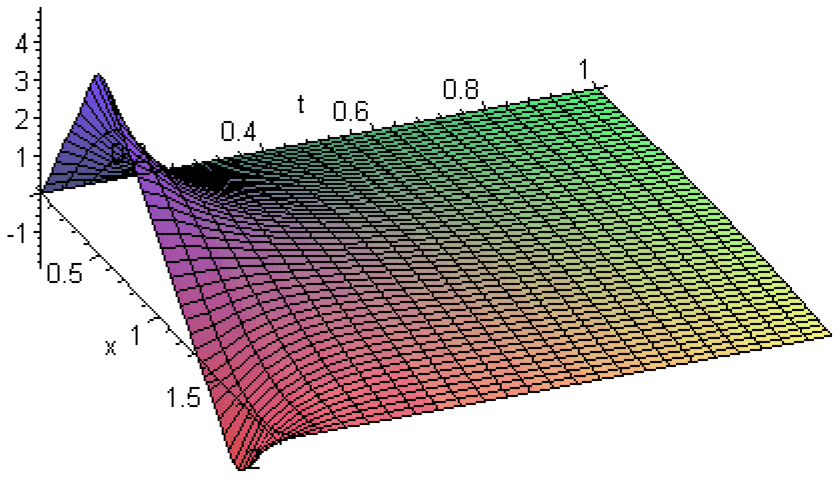


Figure 1: heat distribution across a rod with respect to time.

The assumption of the lateral surfaces being perfectly insulated though is unrealistic it is possible to put enough insulation on the lateral surfaces that there will be very little heat flow through them for a small amount of time.

2.2 2-D heat equation

Consider a thin rectangular plate made of some thermally conductive material. Suppose the dimensions of the plate are a × b as shown in figure 2.

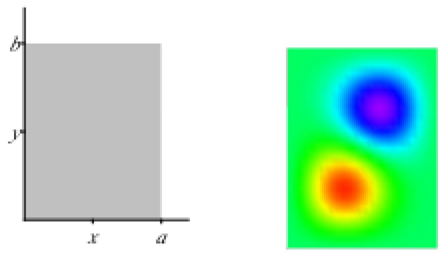


Figure 2: Rectangular plate and a sample heat simulation

We place heat sources at random locations in the rectangular space and the goal is to know the simulated result at different times in different locations within this space.
We let u(x, y,t) = temperature of plate at position (x, y) and time t. The heat conduction through the space follows the partial differential equation given by (1)

$$\frac{\partial u}{\partial t} = \tilde{k}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \tag{1}$$

for $0 < x < a$, $0 < y < b$.
Where k is the thermal conductivity

So in order to solve the above differential equation, many solutions are prescribed in the prior art, out of which we have selected the finite difference method of solving the partial differential equation. According to this solution, we divide the rectangular space into a 2-D grid and discretize the temperatures at different points as shown in figure() and update the temperatures for different time instances. The number of points in the grid determine the resolution of the grid that will be considered in our problem. We have tried the resolution of 256x256, 512x512 and 1024x1024.
There are different types of approximations that are taken to solve the partial differential equation using finite difference method and for our implementation we have used the Central difference approximation. Finally, the discretized solution to a 2-D heat equation as per [1],[2] is as shown in equation 2 and is illustrated using figure 3.

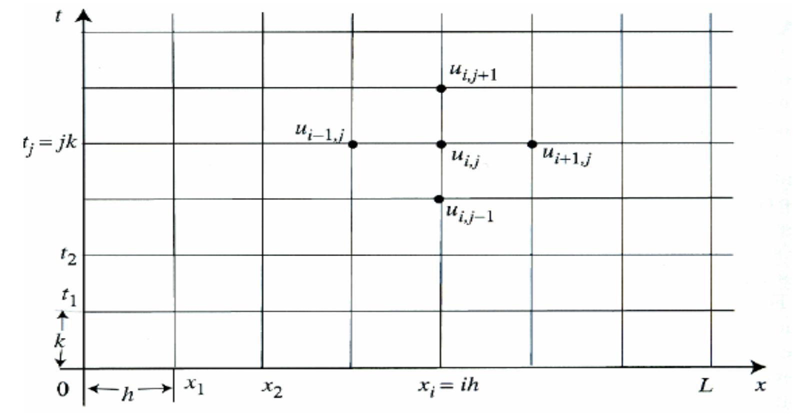


Figure 3: Discretized Grid showing temperatures at different points at time t

$$u_{i,j}(t+1) = u_{i,j}(t) + k*(u_{i,j+1}(\ t) + u_{i,j+1}(t) + u_{i-1}(t) + u_{i+1,j}(t) - 4*\ u_{i,j}(t)) \tag{2}$$

The equation illustrates that the new temperature of every node in the grid depends on its temperature in the previous time step and on the differences between its temperature with respect to four of its neighboring nodes. The four neighbors whose temperatures affect the new temperature of a node are highlighted in figure 4 .

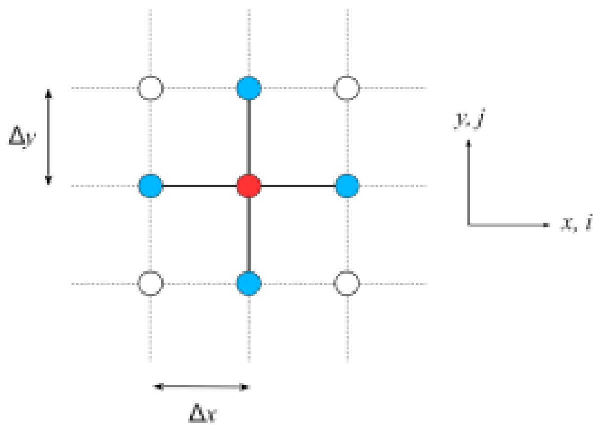


Figure 4: Current node under consideration (in red) and the four neighboring nodes (in blue) that affect the temperature of current node

As seen above, at each time step, the temperature of every node is updated our goal is to do the simulation as fast as possible using Parallelization. There are two ways in which we can stop the simulation to obtain results: (1)We can fix the number of iterations for which the updation of temperature for every node in the grid happens; (2) Calculate the residual temperature of every node at the end of every time step and when to stop the simulation when the residual temperature reaches ~0.0005. The residual temperature of a node is the difference between the new and old temperatures of the node and temperature gradient as low as 0.0005 indicates steady-state condition.

The approach we have taken to parallelize the solving of heat equation and the results obtained are presented in the further sections.

3. Approach

3.1 Serial Execution of Heat equation

We start by assuming that we have some rectangular space that we divide into a grid. Inside the grid, we will randomly scatter a handful of "heaters" with various fixed temperatures. Figure 5 shows an example of what this room might look like.

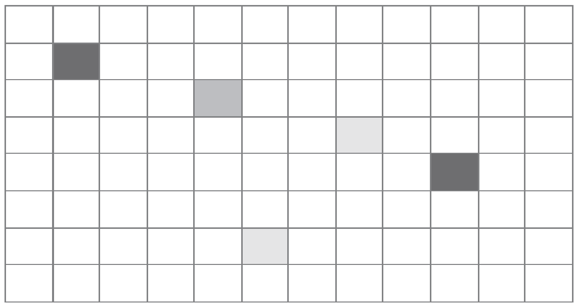


Figure 5: Rectangular space with heaters at highlighted locations

Given a rectangular grid and configuration of heaters, we are looking to simulate what happens to the temperature in every grid cell as time progresses. For simplicity, cells with heaters in them always remain a constant temperature. At every step in time, we will assume that heat “flows” between a cell and its neighbors. If a cell’s neighbor is warmer than it is, the warmer neighbor will tend to warm it up. Conversely, if a cell has a neighbor cooler than it is, it will cool off. Qualitatively, figure 6 represents this flow of heat.

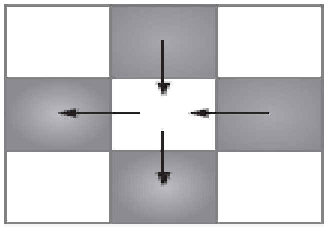


Figure 6: Showing the transfer of heat from heat sources to all the nodes

In our heat transfer model, we will compute the new temperature in a grid cell as a sum of the differences between its temperature and the temperatures of its neighbor, or, essentially, an update equation as (3):

$$T_{NEW} = T_{OLD} + k * (T_{TOP} + T_{BOTTOM} + T_{LEFT} + T_{RIGHT} - 4 * T_{OLD}) \quad (3)$$

In equation (3), for updating a cell’s temperature, the constant k simply represents the rate at which heat flows through the simulation. A large value of k will drive the system to a constant temperature quickly, while a small value will allow the solution to retain large temperature gradients longer. In our implementation, we have fixed the value of k to be 0.25.

Before beginning the simulation, we assume we have generated a grid of constants. Most of the entries in this grid are zero, but some entries contain nonzero temperatures that represent heaters at fixed temperatures. This buffer of constants will not change over the course of the simulation and gets read at each time step. This basic setup remains the same for both serial and parallel executions. However, for serial implementation, we have used the Jacobian method of iterating the temperature computation of each node.

We implemented the sequential version using a simple C code a portion of which has been shown in figure 7:

```
void Jacobi(float* T, float* T_new)
{
  const int MAX_ITER = 1000;
  int iter = 0;

  while(iter &lt; MAX_ITER)
  {
    for(int i=1; i&lt;NX-1; i++)
    {
      for(int j=1; j&lt;NY-1; j++)
      {
        float T_E = T[(i+1) + NX*j];
        float T_W = T[(i-1) + NX*j];
        float T_N = T[i + NX*(j+1)];
        float T_S = T[i + NX*(j-1)];
        T_new[i+NX*j] = 0.25*(T_E + T_W + T_N + T_S);
      }
    }
```

```
        for(int i=1; i&lt;NX-1; i++)
        {
           for(int j=1; j&lt;NY-1; j++)
           {
              float T_E = T_new[(i+1) + NX*j];
              float T_W = T_new[(i-1) + NX*j];
              float T_N = T_new[i + NX*(j+1)];
              float T_S = T_new[i + NX*(j-1)];
              T[i+NX*j] = 0.25*(T_E + T_W + T_N + T_S);
           }
        }
        iter+=2;
    }
}
```

Figure 7: Snippet of the serial execution

We use the jacobi iterative method here for updating the node temperatures. After the function is run, we swap the input and output temperature arrays. This continues for the initialized number of time steps.
The host device was used to execute the above portion of the code 'm' number of times where 'm' is the number of iterations for which the heat equation has to be solved.

3.2. Parallel Implementation

A key to take advantage of GPUs is related to the memory management. There are several kinds of memory available on GPUs with different access times and sizes that constitute a memory hierarchy, as illustrated in figure (). The effective bandwidth of each type of memory can vary by an order of magnitude depending on the access pattern for each type of memory. There is a parallel memory interface between the global memory and every SM of the GPU. From the programmer's point of view, the GPU is considered as a set of SIMD (Single Instruction stream, Multiple Data streams) multiprocessors with shared memory. Therefore, the SPMD (Single Program Multiple Data) programming model is offered by CUDA. Moreover, in order to optimise the GPU performance, we have to consider two main goals: (1) to balance the computation of the sets of threads, and (2) to optimise the data access through the memory hierarchy. A CUDA program is composed of a host program run on the CPU (host) and a set of kernels launched from the host program. A kernel is executed on a grid, which is composed of one or more blocks. And each block has the same number of threads. When all threads of a kernel complete their execution, the corresponding grid terminates and the execution continues on the host until another kernel is invoked. Therefore, in our application which has highly intensive computations, the computation tasks can be offloaded to the GPU, and thus be performed faster in parallel. Separated from the host memory, GPU has its own memories. Among them, global memory is large and has a very high latency. Shared memory is smaller but has a lower latency compared to the global memory. Threads in one block can access the shared memory of their own block and all threads can access the global memory. Global and shared memories are the most widely used memory units of a GPU. However, for our implementation we also introduce another type of GPU memory that has been designed by NVIDIA specially for graphical applications like rendering simulated data in real-time. It has been described in further sections.

3.2.1 Simulating Heat Equation using CUDA

In CUDA, we could think of the entire grid made of sub-grids (blocks). These blocks are two dimensional blocks, square blocks in our case, but need not be square necessarily. For implementation, we made sure that each thread works on updating the temperature of one node in each time step. Therefore, every node in our grid is represented by a unique thread, and a number of threads make up a block of threads (depending on the block dimension). We have fixed the

number of blocks per grid same for all resolutions and varied the number of thread per block depending on the grid size(resolution).
We have fixed the number of iterations for our implementation and for each time step, the host calls the kernel function called *anim_gpu* the snippet of which is shown in figure 8. The kernel function essentially performs 3 steps in updating the new temperature of each cell.
Step 1: We start with the output grid from the previous time step. Given some grid of input temperatures, copy the temperature of cells with heaters to this grid. This will overwrite any previously computed temperatures in these cells, thereby enforcing our restriction that "heating
cells" remain at a constant temperature.
Step 2: To perform the updates, we can have each thread take responsibility for a single cell in our simulation. Each thread will read its cell's temperature and the temperatures of its neighboring cells, perform the previous update computation, and then update its temperature with the new value.
Step 3: Swap the input and output buffers in preparation of the next time step. The output temperature grid computed in step 2 will become the input temperature grid that we start with in step 1 when simulating the next time step.
There is a considerable amount of spatial locality in the memory access pattern required to perform the temperature update in each step. This fact can be used to improve the performance of our simulation by using a special kind of memory called the Texture Memory.

```
void anim_gpu( DataBlock *d, int ticks )
{
HANDLE_ERROR( cudaEventRecord( d->start, 0 ) );
dim3 blocks(DIM/16,DIM/16);
dim3 threads(16,16);
CPUAnimBitmap *bitmap = d->bitmap;

for (int i=0; i<90; i++)
{
copy_const_kernel<<<blocks,threads>>>( d->dev_inSrc, d->dev_constSrc );
blend_kernel<<<blocks,threads>>>( d->dev_outSrc, d->dev_inSrc );
swap( d->dev_inSrc, d->dev_outSrc );
}

float_to_color<<<blocks,threads>>>( d->output_bitmap, d->dev_inSrc );
HANDLE_ERROR( cudaMemcpy( bitmap->get_ptr(), d->output_bitmap,
bitmap->image_size(), cudaMemcpyDeviceToHost ) );
HANDLE_ERROR( cudaEventRecord( d->stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( d->stop ) );
float elapsedTime;
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime, d->start, d->stop ) );
d->totalTime += elapsedTime;
++d->frames;
printf( "Average Time per frame: %3.1f ms\n", d->totalTime/d->frames );
}
```

Figure 8: Snippet of the kernel function executed on the GPU

3.2.2 Texture memory

Like constant memory, texture memory [3] is another variety of read-only memory that can improve performance and reduce memory traffic when reads have certain access patterns. Although texture memory was originally designed for traditional graphics applications, it can also be used quite effectively in some GPU computing applications.Although NVIDIA designed the texture units for the classical OpenGL and DirectX rendering pipelines, texture memory has some properties that make it extremely useful for computing. Like constant memory, texture memory is cached on chip, so in some situations it will provide higher effective bandwidth by reducing memory requests to off-chip DRAM. Specifically, texture caches are designed for graphics applications where memory access patterns exhibit a great deal of spatial locality. In a computing application, this roughly implies that a thread is likely to read from an address "near" the address that nearby threads read, as shown in figure()

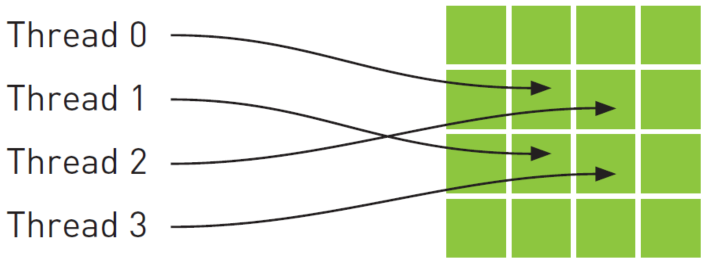


Figure 9: Thread access pattern exploiting spatial locality

Note:Arithmetically, the four addresses shown are not consecutive, so they would not be cached together in a typical CPU caching scheme. But since GPU texture caches are designed to accelerate access patterns such as this one, you will see an increase in performance in this case when using texture memory instead of global memory.

The read-only texture memory space is cached as shown in figure(). Therefore, a texture fetch costs one device memory read only on a cache miss; otherwise, it just costs one read from the texture cache. The texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture addresses that are close together will achieve best performance. Texture memory is also designed for streaming fetches with a constant latency; that is, a cache hit reduces DRAM bandwidth demand, but not fetch latency.

In certain addressing situations, reading device memory through texture fetching can be an advantageous alternative to reading device memory from global or constant memory.

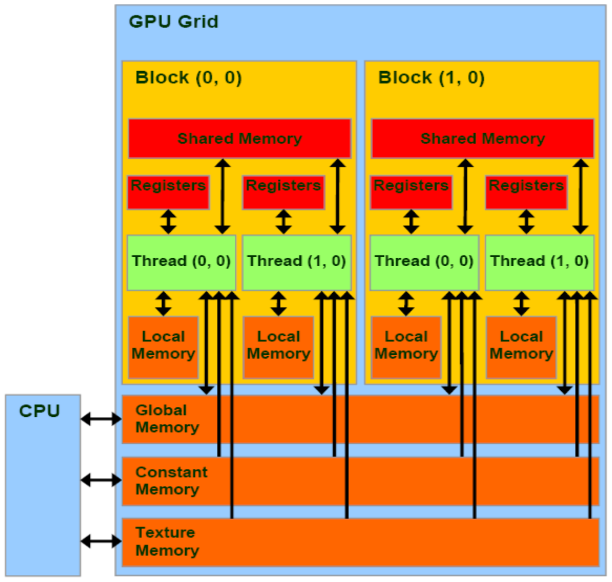


Figure 10:Memory Hierarchy in CUDA indicating Texture Memory as an on-chip cache memory unit

Within a kernel call, the texture cache is not kept coherent with respect to global memory writes, so texture fetches from addresses that have been written via global stores in the same kernel call return undefined data. That is, a thread can safely read a memory location via texture if the location has been updated by a previous kernel call or memory copy, but not if it has been previously updated by the same thread or another thread within the same kernel call. This is relevant only when fetching from linear or pitch-linear memory because a kernel cannot write to CUDA arrays.
Texture memory is to be used when there tends to be some kind of spatial locality to read access pattern and also when in order to visualize data using the graphics pipeline. Since, our project also entails rendering of the heat simulation over time, using texture memory seemed to be appropriate. We have used a 1-D Texture memory for our implementation and at the beginning of each time step, when the kernel is called, each thread reads the temperatures from the input buffer which is in the global memory and writes the new temperature value to the texture memory. When all the threads finish updating their respective node temperatures, the texture memory buffer will copied into the global memory as the input buffer for the next iteration. Also, after every 90 frames the updated temperatures are written into the OpenGL pipeline to visualize the simulation using different colors.

The specifications of the texture memory present in the GPU that we used for our implementation are mentioned below.
Size of texture memory for Tesla K40 GPU:

L2 Cache Size: 1572864 bytes	
Maximum Texture Dimension Size (x,y,z):	1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
Maximum Layered 1D Texture Size, (num) layers:	1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers:	2D=(16384, 16384), 2048 layers

3.2.3 Consequences of changing the shape of the heat body

We have considered a rectangular space for our implementation and this has simplified the access pattern for the threads in parallel execution. However, if we consider a more unstructured space, the memory access patterns would be much more complicated. In that case, we are assuming that by dividing the irregular space into minimum number of bigger rectangles and assigning grid blocks to them would be an ideal approach. Suitable modifications have to be made to the kernel functions.

4 Results

The results obtained from the GPU kernel exactly matched those obtained from the CPU kernel.

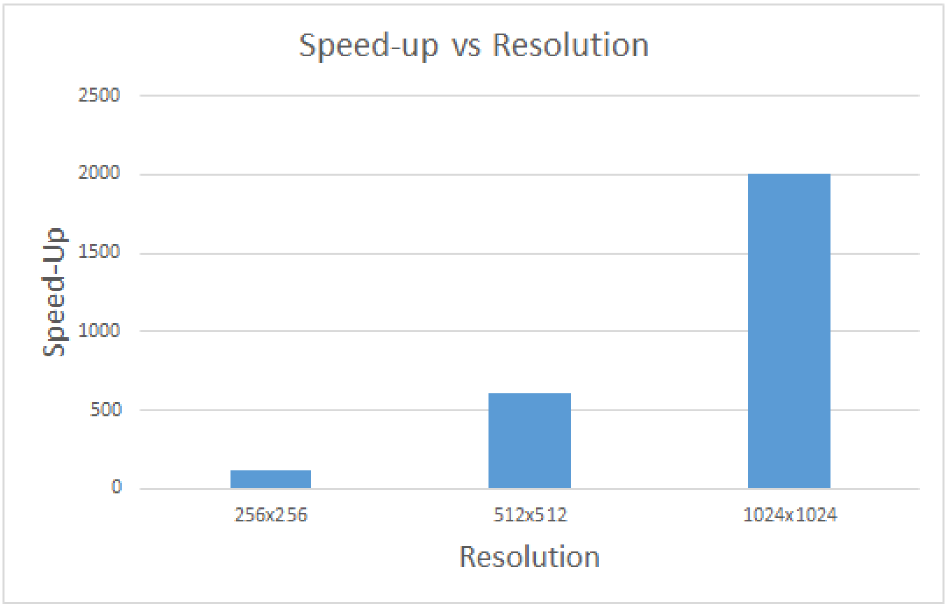
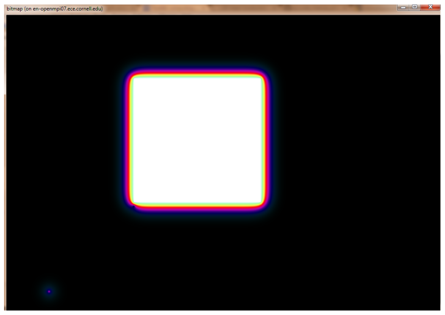


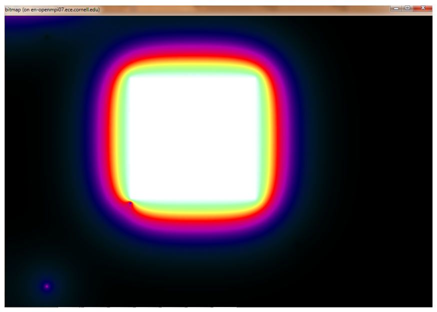
Figure 11: Speed-up obtained for different grid dimensions(resolution)

The Speed-up achieved for different grid dimensions are shown in the figure 11. For 256x256, we considered 16x16 threads per block and were able to get a speed-up of 100x using our implementation of solving the heat equation. Similarly, for 512x512 resolution, we changed the number of threads per block to 32x32 and obtained a speed-up of 600x and finally for a resolution of 1024x1024 which was our target grid size, the number of threads per block were made as 64x64 and this resulted in a speed-up of 2000x. For all the three resolutions, we made sure that each thread operates on updating the temperature of one node in any given time step.

From Figure 11, we see that as we increase the resolution the speed-up incurred also increases and also this increase is seen to be almost a quadratic. This is because as we increase the resolution, we are increasing the number of nodes in both x and y dimensions and hence the total number of nodes quadruples resulting in such high speed-ups. We can clearly see from this that using a GPU to offload compute intensive tasks especially for large number of datapoints helps in increasing the performance to a large extent.

Demo images





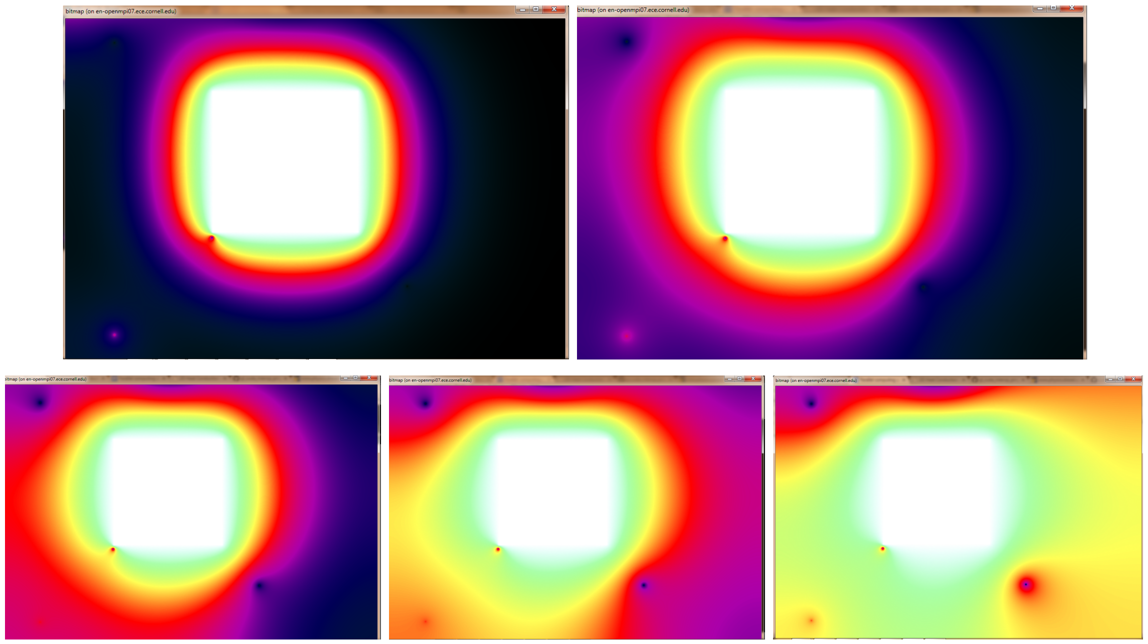


Figure 12: Heat distribution in the object for 7 progressive timestamps

5. Rendering

Rendering is the process of generating an image/video to be output for visualization from a 2D or 3D model (or models in what collectively could be called a *scene* file), by means of computer programs.

Why rendering with GPU is gaining popularity when CPU can already do it ?
Memory load is the bottleneck in today's computing even though on-chip memories have increased considerably. GPUs have a dedicated graphic memory(ram), which are quicker, less fragmented and don't need memory-access-protection, and don't spam the CPU cache providing many ways to optimize time consuming memory access operations. Hence in space of rendering, which is memory intensive and has only particular access pattern GPUs can perform much faster than traditional CPUs

5.1 Rendering using CUDA and OpenGL

Following are the two steps involved in rendering the heat equation:

1. Cuda C kernel is used to generate the image data
2. This data is passed onto openGL driver to render

GLUT calls need to be made before the other GL calls :

```
glutInit( &argc, argv );
glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGBA );
glutInitWindowSize( DIM, DIM );
glutCreateWindow( "bitmap" );
```

```
glGenBuffers( 1, &bufferObj );
glBindBuffer( GL_PIXEL_UNPACK_BUFFER_ARB, bufferObj );
glBufferData( GL_PIXEL_UNPACK_BUFFER_ARB, DIM * DIM * 4,
NULL, GL_DYNAMIC_DRAW_ARB );

HANDLE_ERROR( cudaGraphicsGLRegisterBuffer( &resource, bufferObj,
cudaGraphicsMapFlagsNone ));

static void draw_func( void ) {
glDrawPixels( DIM, DIM, GL_RGBA, GL_UNSIGNED_BYTE, 0 );
glutSwapBuffers();
}
CPUAnimBitmap - structure used from CPU to render using OpenGL
GPUAnimBitmap - structure used from GPU to render using OpenGL
```

Figure 13: Snippet of the OpenGL functions called for Rendering

The OpenGL driver is initialized, display mode and window size are set for drawing the image frame (bitmap) for viewing. Shared data buffers are the key component to interoperation between CUDA C kernels and OpenGL rendering. To pass data between OpenGL and CUDA a buffer is created that can be used with both A pixel buffer object is created in OpenGL and the handle is stored in our global variable GLuint bufferObj, this is the buffer handle which is then bound to pixel buffer. Then OpenGL is requested to allocate buffer of DIM x DIM , which will be dynamically updated and is indicated by the field value - GL_DYNAMIC_DRAW_ARB. This OpenGL buffer, bufferObj is notified to CUDA runtime as a shared object by registering it as CUDA graphics resource. Once the image data to be rendered is generated the shared resource has to be unmapped by calling cudaGraphicsUnmapResources(), this call provides synchronization ensuring that before calling any graphics application all the operations are synchronized. The function glDrawPixels() is uses the shared buffer registered previously to draw the frame.

The only difference in terms of code in GPU rendering is usage of GPUAnimBitmap as opposed to CPUAnimBitmap, which is the greatest flexibility allowed by CUDA. If CPU is used to render the frame generated then once the image is generated from GPU it is copied to the CPU which calls OpenGL driver to render the image. With GPU rendering this overhead of copying from GPU to CPU is removed.
Time taken per frame is 22.31 ns on NVIDIA GeForce GT 420M (with 4GB dedicated graphics memory) with CPU rendering and with GPU rendering it the time taken was 19.18ns, which is approximately 15% speedup.

6 . Conclusions

Heat equation being one of the computationally intensive systems, parallel computing benefits this problem by speeding up this computation and also enabling us to visualize the distribution of heat within the object under consideration. From the discussions in the results section it can be seen that a huge increase in speed, upto 2000x, is achieved by offloading compute intensive part of solving Heat equation to GPU. Especially the use of texture memory for has improved the performance of our solution in terms of both execution time as well graphical rendering. Hence, we can say that parallelization using GPUs can solve problems involving intense computation and also high memory bandwidth demanding applications that are difficult to be implemented on CPUs.

7. References

[1] 2-D Heat equation solving using finite difference method

http://geodynamics.usc.edu/~becker/teaching/557/problem_sets/problem_set_fd_2dheat.pdf
[2]Finite Difference Solution to 2-D Heat Equation
http://www.u.arizona.edu/~erdmann/mse350/_downloads/2D_heat_equation.pdf
[3] Texture Memory
http://cuda-programming.blogspot.com/2013/02/texture-memory-in-cuda-what-is-texture.html
[4] CUDA by Example- An introduction to GPU programming, Jason Sanders & Edward Kandrot
[5] Applications of Heat Equation
http://www.google.com/url?q=http%3A%2F%2Flink.springer.com%2Fchapter%2F10.1007%2F978-3-642-13136-3_42&sa=D&sntz=1&usg=AFQjCNHTuXIXBZMCC0b8nnupyV9J2fg9sg
[6] Heat Equation
http://www.google.com/url?q=http%3A%2F%2Fen.wikipedia.org%2Fwiki%2FHeat_equation%23Applications&sa=D&sntz=1&usg=AFQjCNHosozCGqxreDaYm94mQYCo_cElaw