Successive Over Relaxation Method using CUDA

MAE609 - High Performance Computing I - Course Project
Instructor - Prof. Matthew Jones

University at Buffalo

Pranav Ladkat (5009-4671)

1 Introduction

Successive Over Relaxation (SOR) Method is an iterative scheme to solve the linear system of equations. As discretization of partial differential equations often results in the system of linear equations, SOR method is generally used to solve PDEs. In the present work, the SOR scheme is implemented to run on GPUs using CUDA.

2 Problem Definition

A two dimensional Laplace equation of the form

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0 \tag{1}$$

in the domain $(0 \leq x, y \leq 1)$ with Dirichlet boundary conditions

$$\phi(x,0) = sin(\pi x) \tag{2}$$
$$\phi(x,1) = sin(\pi x)e^{-\pi} \tag{3}$$
$$\phi(0,y) = 0 \tag{4}$$
$$\phi(1,y) = 0 \tag{5}$$

is solved using SOR scheme.

3 Formulation of SOR

Discretizing equation (1) using finite difference:

$$\frac{\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}}{\Delta x^2} + \frac{\phi_{i,j+1} - 2\phi_{i,j} + \phi_{i,j-1}}{\Delta y^2} = 0$$
$$(\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}) + \frac{\Delta x^2}{\Delta y^2}(\phi_{i,j+1} - 2\phi_{i,j} + \phi_{i,j-1}) = 0$$

Let $\beta = \dfrac{\Delta x^2}{\Delta y^2}$

$$(\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}) + \beta(\phi_{i,j+1} - 2\phi_{i,j} + \phi_{i,j-1}) = 0$$
$$(\phi_{i+1,j} + \phi_{i-1,j}) + \beta(\phi_{i,j+1} + \phi_{i,j-1}) - 2(1+\beta)\phi_{i,j} = 0$$

Rearranging

$$\phi_{i,j} = \frac{1}{2(1+\beta)} \left(\phi_{i+1,j} + \phi_{i-1,j} + \beta\left(\phi_{i,j+1} + \phi_{i,j-1}\right)\right)$$

Adding and subtracting $\phi_{i,j}$

$$\phi_{i,j} = \phi_{i,j} + \frac{1}{2(1+\beta)} \left(\phi_{i+1,j} + \phi_{i-1,j} + \beta\left(\phi_{i,j+1} + \phi_{i,j-1}\right) - 2(1+\beta)\phi_{i,j}\right)$$

Multiplying the second term on RHS by factor ω, which is a relaxation factor

$$\phi_{i,j} = \phi_{i,j} + \frac{\omega}{2(1+\beta)} \left(\phi_{i+1,j} + \phi_{i-1,j} + \beta\left(\phi_{i,j+1} + \phi_{i,j-1}\right) - 2(1+\beta)\phi_{i,j}\right)$$

Rearranging,

$$\phi_{i,j}^{n+1} = (1-\omega)\phi_{i,j}^{n} + \frac{\omega}{2(1+\beta)} \left(\phi_{i+1,j}^{n} + \phi_{i-1,j}^{n+1} + \beta\left(\phi_{i,j+1}^{n} + \phi_{i,j-1}^{n+1}\right)\right) \tag{6}$$

The value of $\phi_{i,j}$ at iteration $n+1$ is calculated using equation (6).

4 Red/Black SOR Method

In traditional SOR method, values of $\phi_{i-1,j}^{n+1}$ and $\phi_{i,j-1}^{n+1}$ are accessed to compute the value of $\phi_{i,j}^{n+1}$. In parallel, this access of variables at time level $(n+1)$ creates high data dependency and will not be able to leverage its full performance. To avoid this high data dependency, Red/Black (or Odd/Even) SOR method is implemented.

In the Red/Black SOR, equation (6) is implemented using red-black ordering technique. The node (i,j) is denoted as red or black according to whether $(i+j)$ is odd or even. if $(i+j)$ is odd, the node is marked red and if $(i+j)$ is even, it is marked as black. The red/black ordering is illustrated in figure (4).

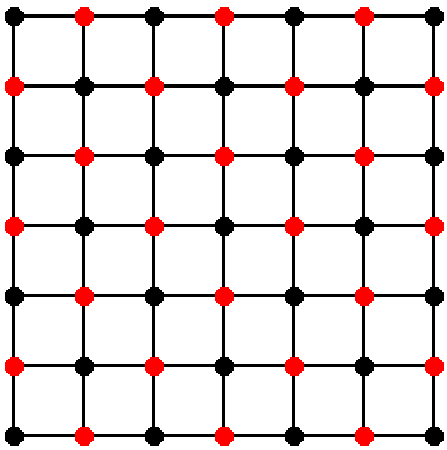

Figure 1: Red/Black ordering

In the Red/Black SOR Method, solution is updated in two stages:

1) Values of all Red/Odd nodes are updated first using equation, if $(i+j)$ is odd:

$$\phi_{i,j}^{n+1} = (1-\omega)\phi_{i,j}^{n} + \frac{\omega}{2(1+\beta)} \left(\phi_{i+1,j}^{n} + \phi_{i-1,j}^{n} + \beta\left(\phi_{i,j+1}^{n} + \phi_{i,j-1}^{n}\right)\right) \tag{7}$$

2) After updating the red/odd values, black/even values are updated using equation, if $(i+j)$ is even:

$$\phi_{i,j}^{n+1} = (1-\omega)\phi_{i,j}^{n+1} + \frac{\omega}{2(1+\beta)} \left(\phi_{i+1,j}^{n+1} + \phi_{i-1,j}^{n+1} + \beta\left(\phi_{i,j+1}^{n+1} + \phi_{i,j-1}^{n+1}\right)\right) \tag{8}$$

5 GPU Implementation

The architecture of GPU is inherently different than a CPU. Architecturally, the CPU is composed of a only few cores with lots of cache memory that can handle a few software threads at a time. In contrast, a GPU is composed of hundreds of cores that can handle thousands of threads simultaneously, hence exploiting large data parallelism. This is illustrated in the figure (5). The number of processor count is not the only difference between the two but that is not the current topic, hence I will limit the discussion.

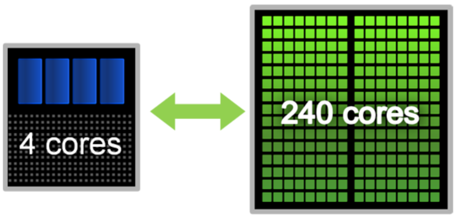

Figure 2: Typical processor count in a CPU and GPU, src: Internet

Basic steps to offload the work to GPUs:

1. create device variables, similar to host variables
2. allocate memory for device variables using ***cudaMalloc()*** in the device's global memory
3. copy contents from host variable to device variable using ***cudaMemcpy()***
4. call cuda kernel to perform the task
5. copy the results from device variables back to host bariables using ***cudaMemcpy()***

Launching Cuda Kernel

The kernel functions (or, simply, kernels) typically generate a large number of threads to exploit data parallelism. The threads in cuda are arranged in two layers. The first layer or a top layer is called the grid and consists of 1 or 2 dimensional array of Blocks (or thread Block). The Blocks themselves consists of 1 or 2 or 3 dimensional array of threads. The brief illustration of cuda grid with 2 dimensional Block is shown in figure (5).

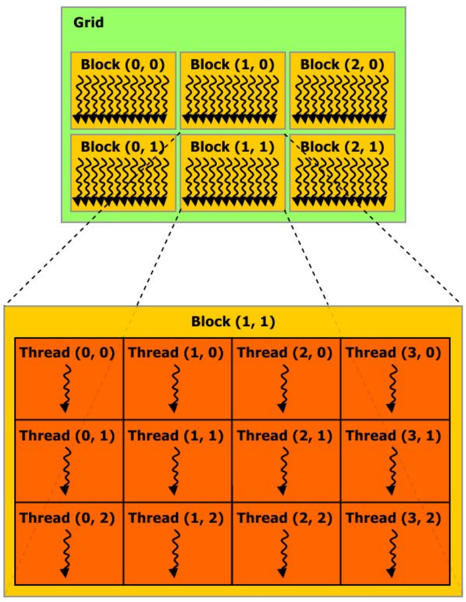

Figure 3: Example of cuda grid, src: Internet

Launching CUDA kernels (host code):

```
int BLOCK_SIZE=16                         // number of threads in each block direction
int gridx = (width-1)/BLOCK_SIZE + 1;     // nblocks in x
int gridy = (height-1)/BLOCK_SIZE + 1;    // nblocks in y
dim3 dimBlock(BLOCK_SIZE,BLOCK_SIZE,1);   // threads in each block in x,y,z
dim3 dimGrid(gridx,gridy,1);              // blocks in x,y

double stime = clock();
for(size_t it = 0; it < itmax; it++){
  solve_odd <<<dimGrid,dimBlock>>> (odd,even); // update red/odd values
  solve_even <<<dimGrid,dimBlock>>> (odd,even);// update black/even values
}
merge_oddeven <<<dimGrid,dimBlock>>> (odd,even);
double etime = clock();
cout << "GPU time : " << (etime-stime)/CLOCKS_PER_SEC << endl;
```

CUDA kernels (device code):

```
__global__ void solve_odd(double* odd,double* even){

  size_t tx = blockIdx.x*blockDim.x + threadIdx.x;
  size_t ty = blockIdx.y*blockDim.y + threadIdx.y;
  size_t index = tx*height+ty;

  if((tx + ty)%2 != 0){
    if(tx > 0 && ty > 0 && tx < width-1 && ty < height-1){
      odd[index] = (1.0-omega)*odd[index] + omega/(2*(1+beta))
          *(even[index+1] + even[index-1] + beta*(even[index+height] + even[index-height]));
    }
  }
}

__global__ void solve_even(double* odd,double* even){

  size_t tx = blockIdx.x*blockDim.x + threadIdx.x;
    size_t ty = blockIdx.y*blockDim.y + threadIdx.y;
    size_t index = tx*height+ty;

    if((tx + ty)%2 == 0){
      if(tx > 0 && ty > 0 && tx < width-1 && ty < height-1){
      even[index] = (1.0-omega)*even[index] + omega/(2*(1+beta))
           *(odd[index+1] + odd[index-1] + beta*(odd[index+height] + odd[index-height]));
      }
    }
}

__global__ void merge_oddeven(double* odd,double* even){

  size_t tx = blockIdx.x*blockDim.x + threadIdx.x;
  size_t ty = blockIdx.y*blockDim.y + threadIdx.y;
  size_t index = tx*height+ty;

  if((tx + ty)%2 == 0 && tx < width-1 && ty < height-1){
      odd[index] = even[index];
  }

}
```

The device memory allocation and data copy functions are shown, they can be found in the code in appendix.

Note: Norm of the error is not calculated in the current work as parallel reduction needs to be implemented which is not implemented.

6 OpenMP Implementation

To compare the performance of the GPU with that of the CPU, OpenMP is used to utilize all cores available in the CPU.

OpenMP code (Uses Red/Black Ordering):

```
  double stime = omp_get_wtime();
  for(size_t it = 0; it < itmax; it++){

#pragma omp parallel
{
#pragma omp for private(i,j,index)
  // update odd cells
  for(i = 0; i < width; i++){
    for(j = 0; j < height; j++){
      index = i*height+j;
      if((i + j)%2 != 0){
        if(i > 0 && j > 0 && i < width-1 && j < height-1){
          odd[index] = (1.0-omega)*odd[index] + omega/(2*(1+beta))
                       *(even[index+1] + even[index-1] + beta*(even[index+height]
                       + even[index-height]));
        }
      }
    }
  }

#pragma omp for private(i,j,index)
  // update even cells
  for(i = 0; i < width; i++){
    for(j = 0; j < height; j++){
      index = i*height+j;
      if((i + j)%2 == 0){
        if(i > 0 && j > 0 && i < width-1 && j < height-1){
          even[index] = (1.0-omega)*even[index] + omega/(2*(1+beta))
                        *(odd[index+1] + odd[index-1] + beta*(odd[index+height]
                        + odd[index-height]));
        }
      }
    }
  }
} // end omp parallel

  } // end iteration loop

#pragma omp parallel for private(i,j,index)
  // merge odd-even solution
  for(i = 0; i < width; i++){
      for(j = 0; j < height; j++){
          index = i*height+j;
          if((i + j)%2 == 0){
              odd[index] = even[index];
          }
      }
  }
  double etime = omp_get_wtime();
  cout << "CPU time : " << (etime-stime) << endl;
```

7 Program Output

Below is the solution obtained using CUDA and OpenMP.

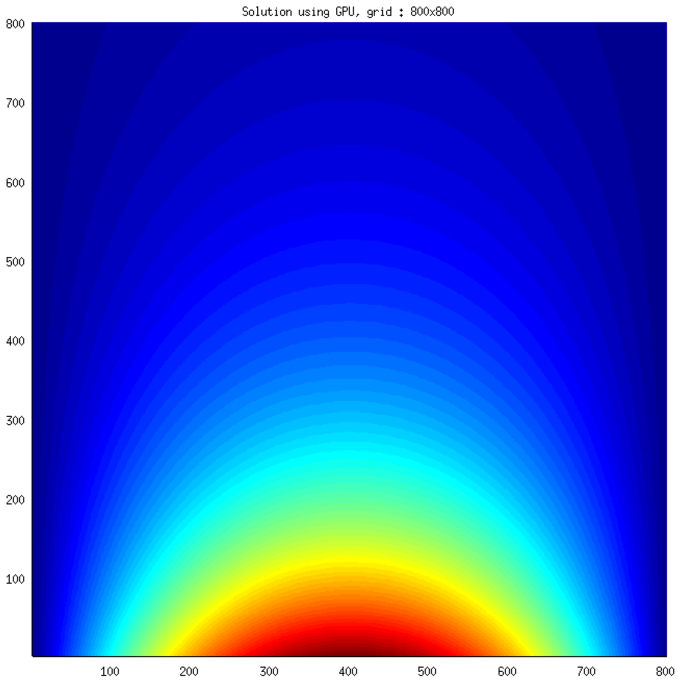

Figure 4: Solution using CUDA

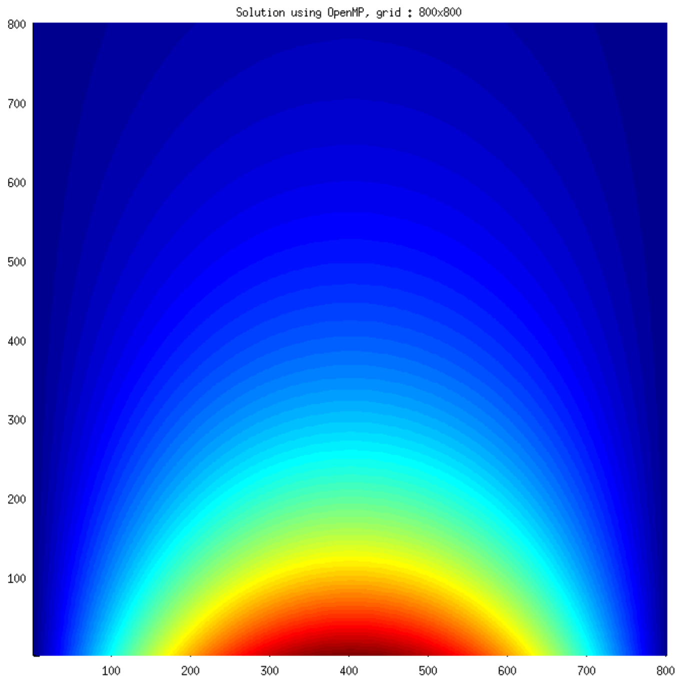

Figure 5: Solution using OpenMP

8 Performance Benchmark

The Performance is checked for three different grids of sizes 800×800, 2048×2048 and 4096×4096. As the norm is not being computed to stop the iterations at convergence, a fixed number of iterations of 1000 are performed and time is measured to iterate over 1000 iterations.

Grid : 800×800

Table 1: Execution Time and parallel speed up for grid size 800×800

N_p	Without Optimization		With Optimization (-O3)	
	Time in Sec	Speed-up	Time in Sec	Speed-up
1	14.1581	1	4.25875	1
2	7.47365	1.8935	2.17003	1.9625
3	4.79196	2.9532	1.39608	3.0505
4	3.60345	3.9272	1.08763	3.9156
6	2.52589	5.6026	0.716876	5.9407
8	1.8319	7.7252	0.545337	7.8093
12	1.24755	11.34	0.377673	11.2762
16	0.961283	14.7230	0.303587	14.02801
GPU	0.29	48.8210	0.29	14.6853

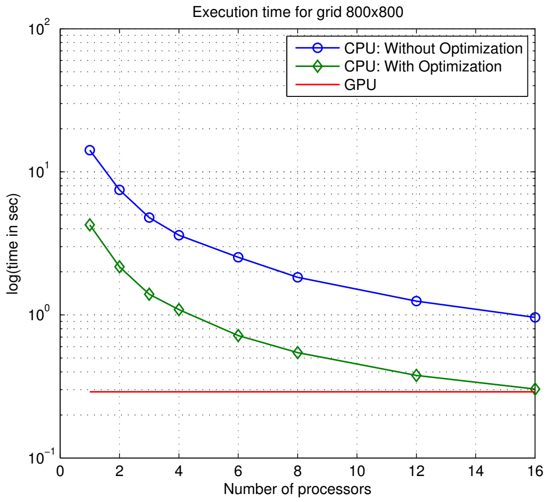

Figure 6: Number of processors vs execution time, with and without optimization flags

The execution timings and parallel speed up is tabulated in Table (1) and are plotted in the figures (6) and (7). In case of non-optimized code (i.e. compiled without optimization flags), the parallel speed up is steady when number of processors are increased from 1 to 16. It can be seen that the GPU attains a speed up of 48.82 when compared with 1 CPU time which is significantly higher. When the Optimization flag (-O3) is used while compiling, the execution time reduced greatly. This can be seen from the above figure. GPU attained a speedup of 14.68 when compared with the execution time of 1 CPU with optimized code; But so did the 16 processors which took approximately same time to execute.

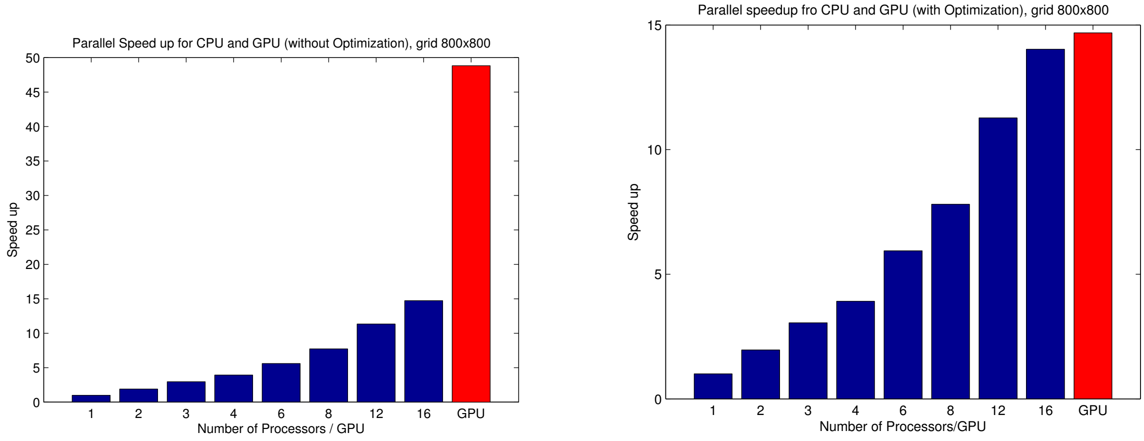

Figure 7: Parallel speed up for grid 800×800

Grid : 2048×2048

Table 2: Execution Time and parallel speed up for grid size 2048×2048

N_p	Without Optimization		With Optimization (-O3)	
	Time in Sec	Speed-up	Time in Sec	Speed-up
1	91.7262	1	56.6644	1
2	47.0344	1.9502	28.4204	1.9937
3	31.105	2.9489	9.48793	5.9722
4	23.34	3.93	6.28152	9.0208
6	15.8192	5.7984	4.35299	13.01735
8	12.0994	7.581	3.65927	15.4851
12	8.1356	11.27	2.18099	25.9810
16	6.358	14.42	1.68599	33.6089
GPU	1.92	47.774	1.92	29.5127

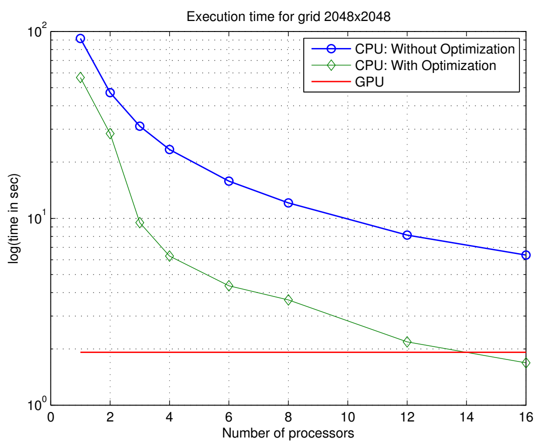

Figure 8: Number of processors vs execution time, with and without optimization flags

The GPU continued to show the speed up of 47 over non-optimized code and significantly better performance than 16 processors. And it showed speed up of 29.5 over optimized version of the code with 1 CPU. Interestingly, it can be seen from the figure (9), optimized version of the code executed in less time that GPU. One reason for this might be optimization flag has enabled better cache memory usage or unrolling of loops.

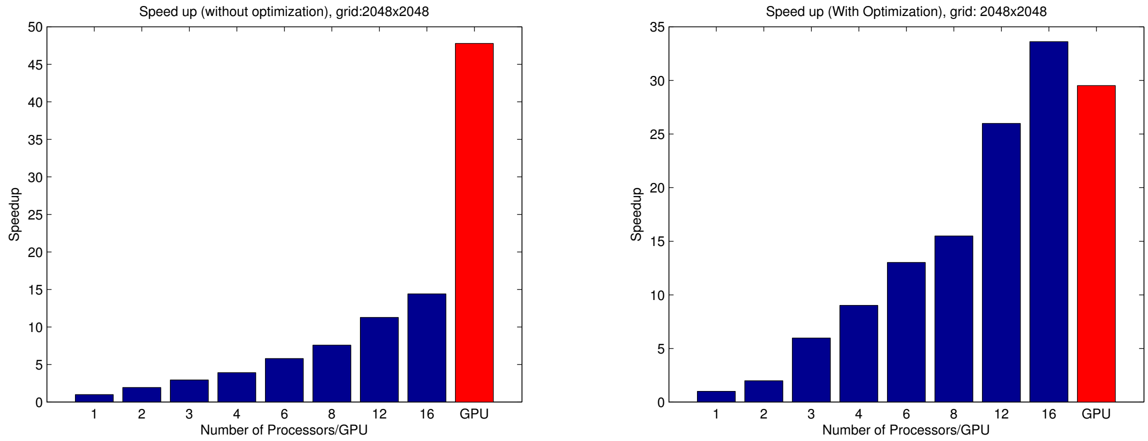

Figure 9: Parallel speed up for grid 2048×2048

Grid : 4096×4096

Table 3: Execution Time and parallel speed up for grid size 4096×4096

N_p	Without Optimization		With Optimization (-O3)	
	Time in Sec	Speed-up	Time in Sec	Speed-up
1	404.833	1	401.614	1
2	210.276	1.9252	220.839	1.8185
3	143.255	2.8259	145.298	2.7640
4	108.336	3.7368	113.796	3.5292
6	74.0907	5.4640	75.0378	5.3521
8	57.9233	6.9891	59.9978	6.6938
12	42.6892	9.4832	41.9063	9.5836
16	38.9089	10.40	30.1972	13.2997
GPU	7.79	51.9682	7.79	51.555

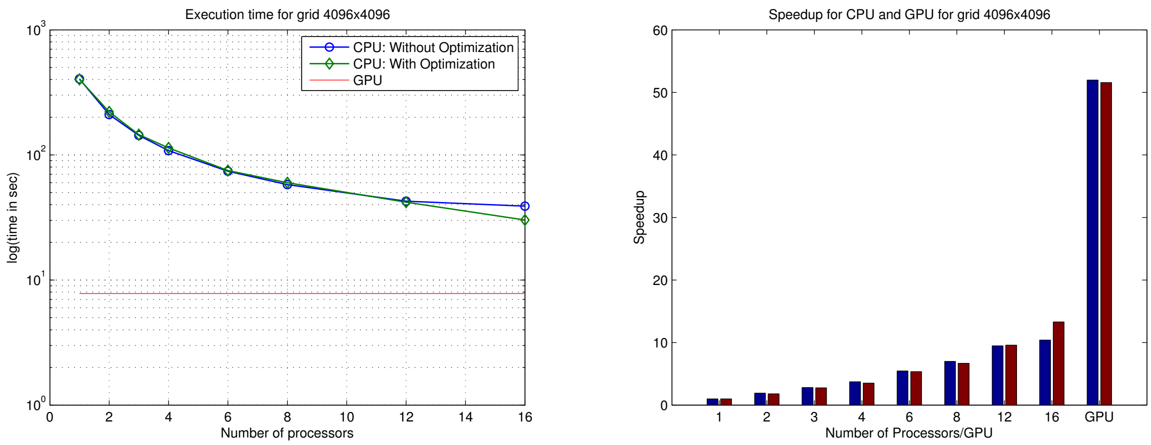

Figure 10: Parallel speed up for grid 4096×4096

For a larger grid size, optimization flags do not provide significant speedup which can be seen from the figure (10). However the GPU continues to achieve the speed up of 51 when compared with 1 CPU time.

9 Compiler

A NVIDIA compiler (nvcc) is used to compile the source code. Cuda version 5.5.22 is used for the present work. The code is compiled with command:

Without Optimization: nvcc -arch=sm_20 -Xcompiler -fopenmp cudaSOR.cu -o sor
With Optimization : nvcc -O3 -arch=sm_20 -Xcompiler -fopenmp cudaSOR.cu -o sor

10 Conclusion

A GPU accelerated Successive Over Relaxation Method is implemented and compared with multiple processors making use of OpenMP. A significant speed up over CPU implementation can be seen from the benchmark results.

11 Appendix

Source code: (cudaSOR.cu)

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>
#include <ctime>
#include <omp.h>


using namespace std;

// global variables
size_t const BLOCK_SIZE = 16;
size_t const width = 800;
size_t const height = 800;
size_t itmax = 5000;
double const omega = 1.97;
double const beta = (((1.0/width) / (1.0/height))*((1.0/width) / (1.0/height)));


// functions
void generategrid(double*,double*,const double,const double,const double,const double);
void setBC(double*, const double*, const double*);
void solve_sor_host(double*);
void solve_sor_cuda(double*);
__global__ void solve_odd(double*,double*);
__global__ void solve_even(double*,double*);
__global__ void merge_oddeven(double*,double*);
void write_output(double*);


// boundary conditions
double leftBC(const double&, const double&);
double rightBC(const double&, const double&);
double topBC(const double&, const double&);
double bottomBC(const double&, const double&);

int main(){

  //host variables
  double *x, *y;        // grid x and y
  double Xmin = 0.0, Xmax = 1.0,
         Ymin = 0.0, Ymax = 1.0;     // grid coordinates bounds
  double *sol;

  // allocate memory for grid
  size_t memsize = width*height;
  x = new double [memsize];
  y = new double [memsize];

  // generate grid
  generategrid(x,y,Xmin,Xmax,Ymin,Ymax);

  // allocate sol memory + set it to zero
  sol = new double [memsize];
  memset(sol,0,memsize*sizeof(double));

  // set boundary conditions
  setBC(sol,x,y);

  // call solvers
  //solve_sor_cuda(sol);
  solve_sor_host(sol);
```

```
  // write output
  write_output(sol);

  delete [] x;
  delete [] y;
  delete [] sol;

  cout << "End!" << endl;
  return 0;
}



void generategrid(double* x,double* y,const double Xmin,const double Xmax,const double Ymin,
    const double Ymax){

  double dx = fabs(Xmax-Xmin)/(width-1);
  double dy = fabs(Ymax-Ymin)/(height-1);

  for(size_t i = 0; i < width; i++){
      for(size_t j = 0; j < height; j++){
          x[i*height + j] = Xmin + i*dx;
          y[i*height + j] = Ymin + j*dy;
          //cout << setw(12) << y[i*height + j];
      }
      //cout << endl;
  }
}


void setBC(double* sol,const double* x, const double* y){

  for(size_t i = 0; i < width; i++){
      for(size_t j = 0; j < height; j++){

          size_t index = i*height + j ;
          if(i == 0){
              sol[index] = leftBC(x[index],y[index]);
          }
          if(i == width-1){
              sol[index] = rightBC(x[index],y[index]);
          }
          if(j == 0){
              sol[index] = bottomBC(x[index],y[index]);
          }
          if(j == height-1){
              sol[index] = topBC(x[index],y[index]);
          }
      }
  }
}


// boundary conditions
double leftBC(const double& x, const double& y){
  return 0;
}

double rightBC(const double& x, const double& y){
  return 0;
}

double topBC(const double& x, const double& y){
  return sin(M_PI*x)*exp(-M_PI);
}
```

```
double bottomBC(const double& x, const double& y){
  return sin(M_PI*x);
}



void solve_sor_cuda(double* sol_host){


  const int memsize = width*height;

  // device variables -> odd and even
  double *odd, *even;

  cudaMalloc(&odd,memsize*sizeof(double));
  cudaMalloc(&even,memsize*sizeof(double));
  cudaMemset(&odd,0,memsize);
  cudaMemset(&even,0,memsize);

  // copy initial guess from host to device memory
  cudaMemcpy(odd,sol_host,memsize*sizeof(double),cudaMemcpyHostToDevice);
  cudaMemcpy(even,sol_host,memsize*sizeof(double),cudaMemcpyHostToDevice);

  int gridx = (width-1)/BLOCK_SIZE + 1;
  int gridy = (height-1)/BLOCK_SIZE + 1;
  dim3 dimBlock(BLOCK_SIZE,BLOCK_SIZE,1);
  dim3 dimGrid(gridx,gridy,1);

  cout << "gridx = " << gridx << "\t" << "gridy = " << gridy << endl;

  double stime = clock();
  for(size_t it = 0; it < itmax; it++){
      solve_odd <<<dimGrid,dimBlock>>> (odd,even);
      solve_even <<<dimGrid,dimBlock>>> (odd,even);
  }
  merge_oddeven <<<dimGrid,dimBlock>>> (odd,even);
  double etime = clock();
  cout << "GPU time : " << (etime-stime)/CLOCKS_PER_SEC << endl;

  // copy solution from device to host memory
  cudaMemcpy(sol_host,odd,memsize*sizeof(double),cudaMemcpyDeviceToHost);

  cudaFree(odd);
  cudaFree(even);

}



__global__ void solve_odd(double* odd,double* even){

  size_t tx = blockIdx.x*blockDim.x + threadIdx.x;
  size_t ty = blockIdx.y*blockDim.y + threadIdx.y;
  size_t index = tx*height+ty;

  if((tx + ty)%2 != 0){
      if(tx > 0 && ty > 0 && tx < width-1 && ty < height-1){
          odd[index] = (1.0-omega)*odd[index] + omega/(2*(1+beta))
                       *(even[index+1] + even[index-1] + beta*(even[index+height] + even[index
    -height]));
      }
  }
}

```

```

__global__ void solve_even(double* odd,double* even){

  size_t tx = blockIdx.x*blockDim.x + threadIdx.x;
    size_t ty = blockIdx.y*blockDim.y + threadIdx.y;
    size_t index = tx*height+ty;
    //double beta = pow((1.0/width) / (1.0/height),2);

    //even
    if((tx + ty)%2 == 0){
        if(tx > 0 && ty > 0 && tx < width-1 && ty < height-1){
            even[index] = (1.0-omega)*even[index] + omega/(2*(1+beta))
                        *(odd[index+1] + odd[index-1] + beta*(odd[index+height] + odd[index-
    height]));
        }
    }


}



__global__ void merge_oddeven(double* odd,double* even){

  size_t tx = blockIdx.x*blockDim.x + threadIdx.x;
  size_t ty = blockIdx.y*blockDim.y + threadIdx.y;
  size_t index = tx*height+ty;

  if((tx + ty)%2 == 0 && tx < width-1 && ty < height-1){
      odd[index] = even[index];
  }

}



void solve_sor_host(double* sol){

  const int memsize = width*height;
  double *odd, *even;
  size_t i,j, index;

  odd = new double [memsize];
  even = new double [memsize];

  memcpy(odd,sol,memsize*sizeof(double));
  memcpy(even,sol,memsize*sizeof(double));


  double stime = omp_get_wtime();
  for(size_t it = 0; it < itmax; it++){

#pragma omp parallel
{
#pragma omp for private(i,j,index)
  // update odd cells
  for(i = 0; i < width; i++){
    for(j = 0; j < height; j++){
      index = i*height+j;
      if((i + j)%2 != 0){
        if(i > 0 && j > 0 && i < width-1 && j < height-1){
          odd[index] = (1.0-omega)*odd[index] + omega/(2*(1+beta))
                      *(even[index+1] + even[index-1] + beta*(even[index+height]
                      + even[index-height]));
        }
      }
```

```
    }
  }

#pragma omp for private(i,j,index)
  // update even cells
  for(i = 0; i < width; i++){
    for(j = 0; j < height; j++){
      index = i*height+j;
      if((i + j)%2 == 0){
        if(i > 0 && j > 0 && i < width-1 && j < height-1){
          even[index] = (1.0-omega)*even[index] + omega/(2*(1+beta))
                        *(odd[index+1] + odd[index-1] + beta*(odd[index+height]
                        + odd[index-height]));
        }
      }
    }
  }
} // end omp parallel

  } // end iteration loop

#pragma omp parallel for private(i,j,index)
  // merge odd-even solution
  for(i = 0; i < width; i++){
      for(j = 0; j < height; j++){
          index = i*height+j;
          if((i + j)%2 == 0){
              odd[index] = even[index];
          }
      }
  }
  double etime = omp_get_wtime();
  cout << "CPU time : " << (etime-stime) << endl;


  // copy solution from odd to sol
  memcpy(sol,odd,memsize*sizeof(double));

  delete [] odd;
  delete [] even;

}


void write_output(double* sol){

  ofstream file("cusol.dat");
    for(int i = 0; i < width; i++){
      for(int j = 0; j < height; j++){
        file << setw(12) << sol[i*height + j];
      }
      file << endl;
    }
    file.close();
}
```