

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

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0 \quad (1)$$

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

$$\phi(x, 0) = \sin(\pi x) \quad (2)$$

$$\phi(x, 1) = \sin(\pi x)e^{-\pi} \quad (3)$$

$$\phi(0, y) = 0 \quad (4)$$

$$\phi(1, y) = 0 \quad (5)$$

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 $\beta = \frac{\Delta x^2}{\Delta y^2}$

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

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

Rearranging

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

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

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

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

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

Rearranging,

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

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).

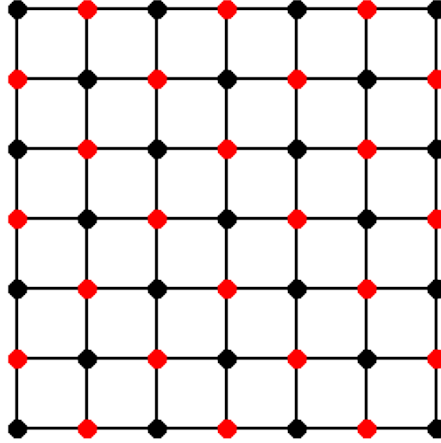


Figure 1: Red/Black ordering

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:

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

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

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

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.

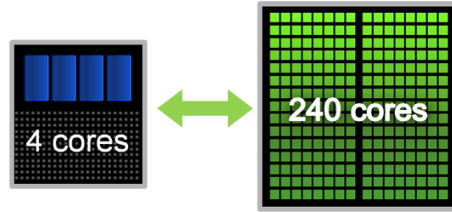


Figure 2: Typical processor count in a CPU and GPU, src: Internet

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 variables 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).

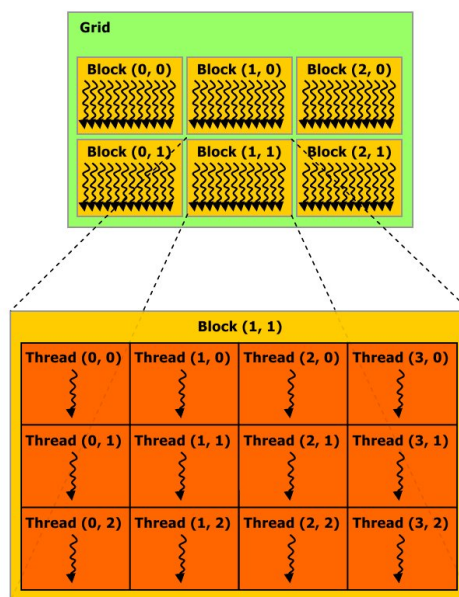


Figure 3: Example of cuda grid, src: Internet

Launching CUDA kernels (host code):

```
1 int BLOCK_SIZE=16 // number of threads in each block direction
2 int gridx = (width-1)/BLOCK_SIZE + 1; // nblocks in x
3 int gridy = (height-1)/BLOCK_SIZE + 1; // nblocks in y
4 dim3 dimBlock(BLOCK_SIZE,BLOCK_SIZE,1); // threads in each block in x,y,z
5 dim3 dimGrid(gridx ,gridy ,1); // blocks in x,y
6
7 double stime = clock();
8 for(size_t it = 0; it < itmax; it++){
9     solve_odd <<<dimGrid,dimBlock>>> (odd,even); // update red/odd values
10    solve_even <<<dimGrid,dimBlock>>> (odd,even); // update black/even values
11 }
12 merge_oddeven <<<dimGrid,dimBlock>>> (odd,even);
13 double etime = clock();
14 cout << "GPU time : " << (etime-stime)/CLOCKS_PER_SEC << endl;
```

CUDA kernels (device code):

```
1 __global__ void solve_odd(double* odd,double* even){
2
3     size_t tx = blockIdx.x*blockDim.x + threadIdx.x;
4     size_t ty = blockIdx.y*blockDim.y + threadIdx.y;
5     size_t index = tx*height+ty;
6
7     if((tx + ty)%2 != 0){
8         if(tx > 0 && ty > 0 && tx < width-1 && ty < height-1){
9             odd[index] = (1.0-omega)*odd[index] + omega/(2*(1+beta))
10                *(even[index+1] + even[index-1] + beta*(even[index+height] + even[index-height]));
11         }
12     }
13 }
14
15 __global__ void solve_even(double* odd,double* even){
16
17     size_t tx = blockIdx.x*blockDim.x + threadIdx.x;
18     size_t ty = blockIdx.y*blockDim.y + threadIdx.y;
19     size_t index = tx*height+ty;
20
21     if((tx + ty)%2 == 0){
22         if(tx > 0 && ty > 0 && tx < width-1 && ty < height-1){
23             even[index] = (1.0-omega)*even[index] + omega/(2*(1+beta))
24                *(odd[index+1] + odd[index-1] + beta*(odd[index+height] + odd[index-height]));
25         }
26     }
27 }
28
29 __global__ void merge_oddeven(double* odd,double* even){
30
31     size_t tx = blockIdx.x*blockDim.x + threadIdx.x;
32     size_t ty = blockIdx.y*blockDim.y + threadIdx.y;
33     size_t index = tx*height+ty;
34
35     if((tx + ty)%2 == 0 && tx < width-1 && ty < height-1){
36         odd[index] = even[index];
37     }
38 }
39 }
```

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):

```
1  double stime = omp_get_wtime();
2  for(size_t it = 0; it < itmax; it++){
3
4  #pragma omp parallel
5  {
6  #pragma omp for private(i,j,index)
7  // update odd cells
8  for(i = 0; i < width; i++){
9      for(j = 0; j < height; j++){
10         index = i*height+j;
11         if((i + j)%2 != 0){
12             if(i > 0 && j > 0 && i < width-1 && j < height-1){
13                 odd[index] = (1.0-omega)*odd[index] + omega/(2*(1+beta))
14                     *(even[index+1] + even[index-1] + beta*(even[index+height]
15                     + even[index-height]));
16             }
17         }
18     }
19 }
20
21 #pragma omp for private(i,j,index)
22 // update even cells
23 for(i = 0; i < width; i++){
24     for(j = 0; j < height; j++){
25         index = i*height+j;
26         if((i + j)%2 == 0){
27             if(i > 0 && j > 0 && i < width-1 && j < height-1){
28                 even[index] = (1.0-omega)*even[index] + omega/(2*(1+beta))
29                     *(odd[index+1] + odd[index-1] + beta*(odd[index+height]
30                     + odd[index-height]));
31             }
32         }
33     }
34 }
35 } // end omp parallel
36
37 } // end iteration loop
38
39 #pragma omp parallel for private(i,j,index)
40 // merge odd-even solution
41 for(i = 0; i < width; i++){
42     for(j = 0; j < height; j++){
43         index = i*height+j;
44         if((i + j)%2 == 0){
45             odd[index] = even[index];
46         }
47     }
48 }
49 double etime = omp_get_wtime();
50 cout << "CPU time : " << (etime-stime) << endl;
```

7 Program Output

Below is the solution obtained using CUDA and OpenMP.

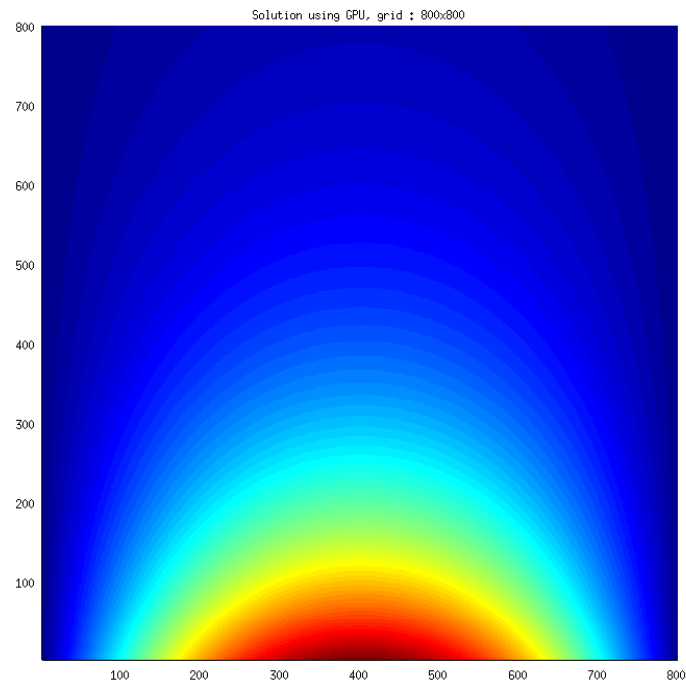


Figure 4: Solution using CUDA

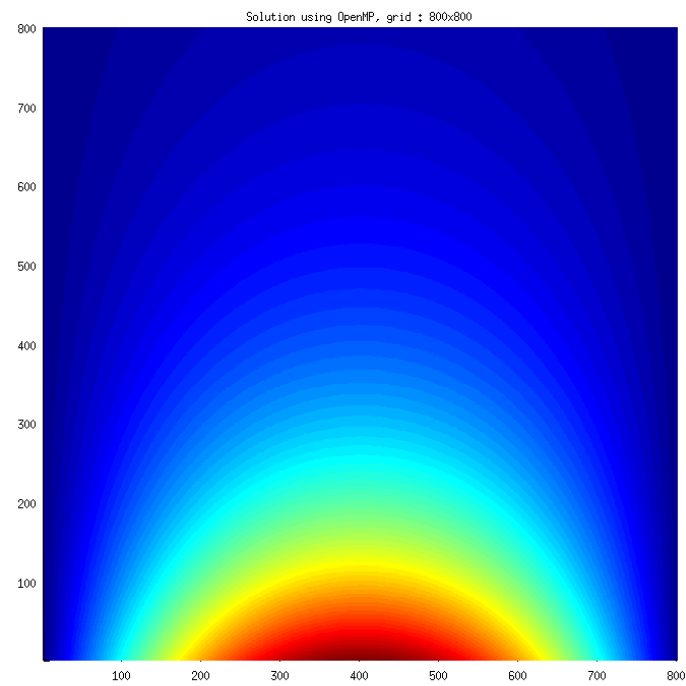


Figure 5: Solution using OpenMP

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

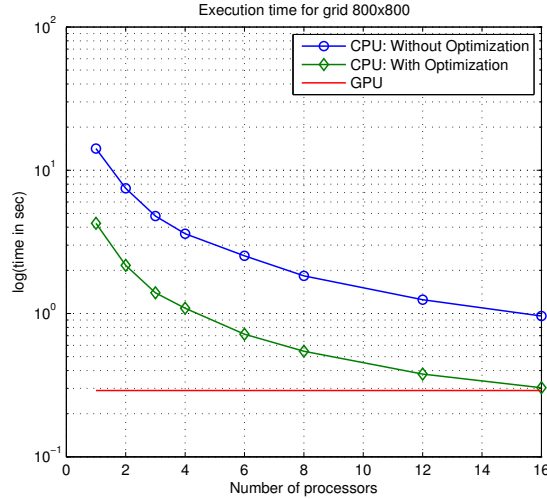


Figure 6: Number of processors vs execution time, with and without optimization flags

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.

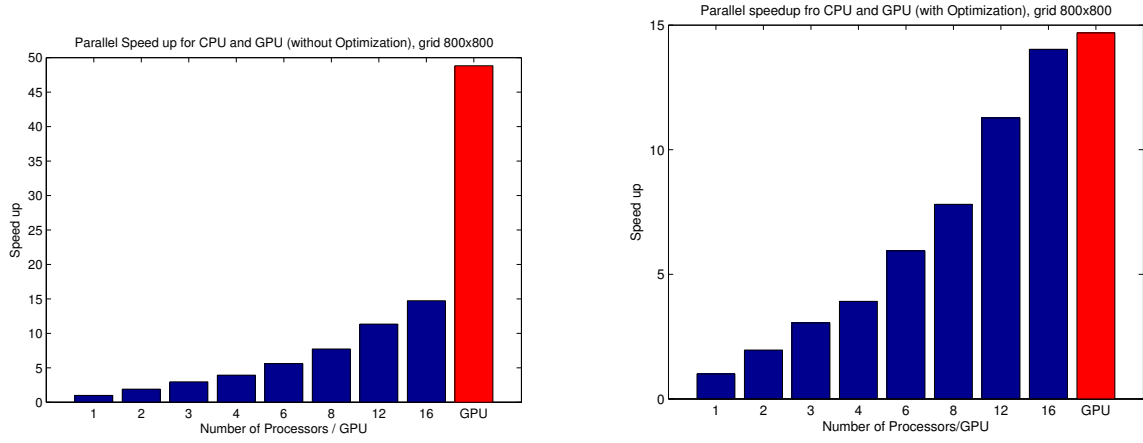


Figure 7: Parallel speed up for grid 800×800

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

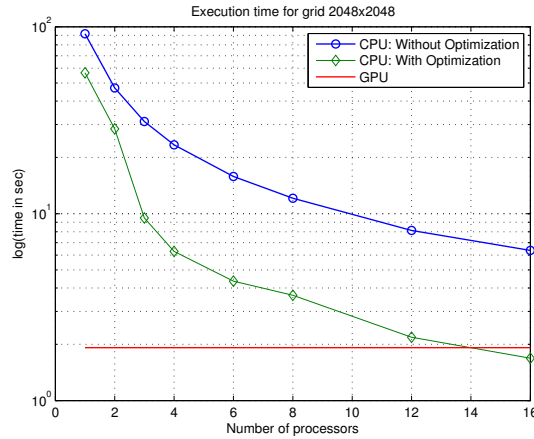


Figure 8: Number of processors vs execution time, with and without optimization flags

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.

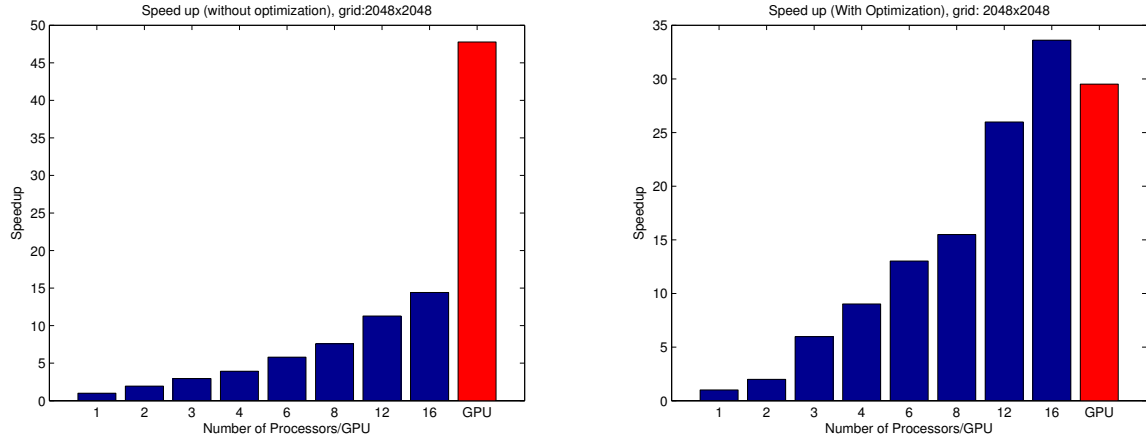


Figure 9: Parallel speed up for grid 2048×2048

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

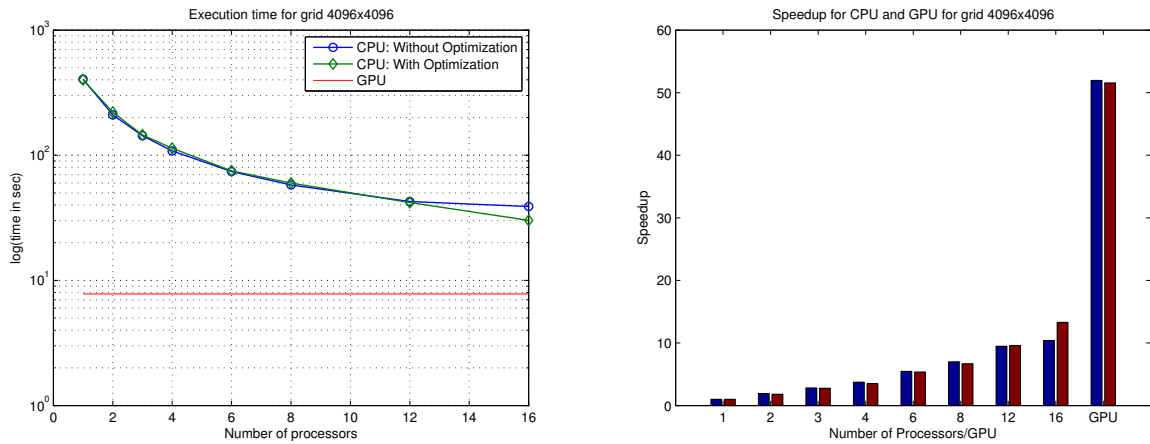


Figure 10: Parallel speed up for grid 4096×4096

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)

```
1 #include <iostream>
2 #include <fstream>
3 #include <iomanip>
4 #include <cmath>
5 #include <ctime>
6 #include <omp.h>
7
8
9 using namespace std;
10
11 // global variables
12 size_t const BLOCK_SIZE = 16;
13 size_t const width = 800;
14 size_t const height = 800;
15 size_t itmax = 5000;
16 double const omega = 1.97;
17 double const beta = (((1.0/width) / (1.0/height))*((1.0/width) / (1.0/height)));
18
19
20 // functions
21 void generategrid(double*,double*,const double,const double,const double,const double);
22 void setBC(double*, const double*, const double*);
23 void solve_sor_host(double*);
24 void solve_sor_cuda(double*);
25 __global__ void solve_odd(double*,double*);
26 __global__ void solve_even(double*,double*);
27 __global__ void merge_oddeven(double*,double*);
28 void write_output(double*);
29
30
31 // boundary conditions
32 double leftBC(const double&, const double&);
33 double rightBC(const double&, const double&);
34 double topBC(const double&, const double&);
35 double bottomBC(const double&, const double&);
36
37 int main(){
38
39     //host variables
40     double *x, *y;           // grid x and y
41     double Xmin = 0.0, Xmax = 1.0,
42            Ymin = 0.0, Ymax = 1.0;    // grid coordinates bounds
43     double *sol;
44
45     // allocate memory for grid
46     size_t memsize = width*height;
47     x = new double [memsize];
48     y = new double [memsize];
49
50     // generate grid
51     generategrid(x,y,Xmin,Xmax,Ymin,Ymax);
52
53     // allocate sol memory + set it to zero
54     sol = new double [memsize];
55     memset(sol,0,memsize*sizeof(double));
56
57     // set boundary conditions
58     setBC(sol,x,y);
59
60     // call solvers
61     //solve_sor_cuda(sol);
62     solve_sor_host(sol);
```

```

63 // write output
64 write_output(sol);
65
66 delete [] x;
67 delete [] y;
68 delete [] sol;
69
70
71 cout << "End!" << endl;
72 return 0;
73 }
74
75
76
77 void generategrid(double* x, double* y, const double Xmin, const double Xmax, const double Ymin,
78                 const double Ymax){
79
80     double dx = fabs(Xmax-Xmin)/(width-1);
81     double dy = fabs(Ymax-Ymin)/(height-1);
82
83     for(size_t i = 0; i < width; i++){
84         for(size_t j = 0; j < height; j++){
85             x[i*height + j] = Xmin + i*dx;
86             y[i*height + j] = Ymin + j*dy;
87             //cout << setw(12) << y[i*height + j];
88         }
89         //cout << endl;
90     }
91 }
92
93 void setBC(double* sol, const double* x, const double* y){
94
95     for(size_t i = 0; i < width; i++){
96         for(size_t j = 0; j < height; j++){
97
98             size_t index = i*height + j ;
99             if(i == 0){
100                 sol[index] = leftBC(x[index], y[index]);
101             }
102             if(i == width-1){
103                 sol[index] = rightBC(x[index], y[index]);
104             }
105             if(j == 0){
106                 sol[index] = bottomBC(x[index], y[index]);
107             }
108             if(j == height-1){
109                 sol[index] = topBC(x[index], y[index]);
110             }
111         }
112     }
113 }
114
115 // boundary conditions
116 double leftBC(const double& x, const double& y){
117     return 0;
118 }
119
120
121 double rightBC(const double& x, const double& y){
122     return 0;
123 }
124
125 double topBC(const double& x, const double& y){
126     return sin(M_PI*x)*exp(-M_PI);
127 }

```

```

128
129 double bottomBC(const double& x, const double& y){
130     return sin(M_PI*x);
131 }
132
133
134
135
136 void solve_sor_cuda(double* sol_host){
137
138     const int memsize = width*height;
139
140     // device variables -> odd and even
141     double *odd, *even;
142
143     cudaMalloc(&odd, memsize*sizeof(double));
144     cudaMalloc(&even, memsize*sizeof(double));
145     cudaMemset(&odd, 0, memsize);
146     cudaMemset(&even, 0, memsize);
147
148     // copy initial guess from host to device memory
149     cudaMemcpy(odd, sol_host, memsize*sizeof(double), cudaMemcpyHostToDevice);
150     cudaMemcpy(even, sol_host, memsize*sizeof(double), cudaMemcpyHostToDevice);
151
152     int gridx = (width-1)/BLOCK_SIZE + 1;
153     int gridy = (height-1)/BLOCK_SIZE + 1;
154     dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE, 1);
155     dim3 dimGrid(gridx, gridy, 1);
156
157     cout << "gridx = " << gridx << "\t" << "gridy = " << gridy << endl;
158
159     double stime = clock();
160     for(size_t it = 0; it < itmax; it++){
161         solve_odd <<<dimGrid, dimBlock>>> (odd, even);
162         solve_even <<<dimGrid, dimBlock>>> (odd, even);
163     }
164     merge_oddeven <<<dimGrid, dimBlock>>> (odd, even);
165     double etime = clock();
166     cout << "GPU time : " << (etime-stime)/CLOCKS_PER_SEC << endl;
167
168     // copy solution from device to host memory
169     cudaMemcpy(sol_host, odd, memsize*sizeof(double), cudaMemcpyDeviceToHost);
170
171     cudaFree(odd);
172     cudaFree(even);
173 }
174
175 }
176
177
178
179 --global-- void solve_odd(double* odd, double* even){
180
181     size_t tx = blockIdx.x*blockDim.x + threadIdx.x;
182     size_t ty = blockIdx.y*blockDim.y + threadIdx.y;
183     size_t index = tx*height+ty;
184
185     if((tx + ty)%2 != 0){
186         if(tx > 0 && ty > 0 && tx < width-1 && ty < height-1){
187             odd[index] = (1.0-omega)*odd[index] + omega/(2*(1+beta))
188                 *(even[index+1] + even[index-1] + beta*(even[index+height] + even[index
189                 -height]));
190         }
191     }
192 }

```

```

193
194 --global-- void solve_even(double* odd, double* even){
195
196     size_t tx = blockIdx.x*blockDim.x + threadIdx.x;
197     size_t ty = blockIdx.y*blockDim.y + threadIdx.y;
198     size_t index = tx*height+ty;
199     //double beta = pow((1.0/width) / (1.0/height), 2);
200
201     //even
202     if((tx + ty)%2 == 0){
203         if(tx > 0 && ty > 0 && tx < width-1 && ty < height-1){
204             even[index] = (1.0-omega)*even[index] + omega/(2*(1+beta))
205                 *(odd[index+1] + odd[index-1] + beta*(odd[index+height] + odd[index-
206             height]));
207         }
208     }
209 }
210
211
212
213
214 --global-- void merge_oddeven(double* odd, double* even){
215
216     size_t tx = blockIdx.x*blockDim.x + threadIdx.x;
217     size_t ty = blockIdx.y*blockDim.y + threadIdx.y;
218     size_t index = tx*height+ty;
219
220     if((tx + ty)%2 == 0 && tx < width-1 && ty < height-1){
221         odd[index] = even[index];
222     }
223 }
224
225
226
227 void solve_sor_host(double* sol){
228
229     const int memsize = width*height;
230     double *odd, *even;
231     size_t i, j, index;
232
233     odd = new double [memsize];
234     even = new double [memsize];
235
236     memcpy(odd, sol, memsize*sizeof(double));
237     memcpy(even, sol, memsize*sizeof(double));
238
239
240     double stime = omp_get_wtime();
241     for(size_t it = 0; it < itmax; it++){
242
243     #pragma omp parallel
244     {
245     #pragma omp for private(i, j, index)
246         // update odd cells
247         for(i = 0; i < width; i++){
248             for(j = 0; j < height; j++){
249                 index = i*height+j;
250                 if((i + j)%2 != 0){
251                     if(i > 0 && j > 0 && i < width-1 && j < height-1){
252                         odd[index] = (1.0-omega)*odd[index] + omega/(2*(1+beta))
253                             *(even[index+1] + even[index-1] + beta*(even[index+height]
254                             + even[index-height]));
255                     }
256                 }
257             }

```

```

258     }
259 }
260
261 #pragma omp for private(i,j,index)
262 // update even cells
263 for(i = 0; i < width; i++){
264     for(j = 0; j < height; j++){
265         index = i*height+j;
266         if((i + j)%2 == 0){
267             if(i > 0 && j > 0 && i < width-1 && j < height-1){
268                 even[index] = (1.0-omega)*even[index] + omega/(2*(1+beta))
269                     *(odd[index+1] + odd[index-1] + beta*(odd[index+height]
270                     + odd[index-height]));
271             }
272         }
273     }
274 }
275 } // end omp parallel
276
277 } // end iteration loop
278
279 #pragma omp parallel for private(i,j,index)
280 // merge odd-even solution
281 for(i = 0; i < width; i++){
282     for(j = 0; j < height; j++){
283         index = i*height+j;
284         if((i + j)%2 == 0){
285             odd[index] = even[index];
286         }
287     }
288 }
289 double etime = omp_get_wtime();
290 cout << "CPU time : " << (etime-stime) << endl;
291
292 // copy solution from odd to sol
293 memcpy(sol, odd, memsize*sizeof(double));
294
295 delete [] odd;
296 delete [] even;
297
298 }
299
300
301
302 void write_output(double* sol){
303
304     ofstream file("cusol.dat");
305     for(int i = 0; i < width; i++){
306         for(int j = 0; j < height; j++){
307             file << setw(12) << sol[i*height + j];
308         }
309         file << endl;
310     }
311     file.close();
312 }

```