Introduction to High Performance Computing
IN4049TU

Assignment Report: Parallel Programming with MPI and CUDA

Arvind Nayak
(5374707)

February 16, 2021

Contents

1 Ex1: Poisson Solver with Finite Differences **2**
1.1 Building a parallel version using `MPI` . . . 2
1.2 Performance Experiments . . . 4

2 Ex2: Poisson Solver with Finite Elements **10**
2.1 Border exchange . . . 10
2.2 Benchmarking . . . 10
2.3 Data transfer . . . 11
2.4 Asymmetry . . . 11
2.5 Data locality ratio . . . 11
2.6 Mesh refinement . . . 11

3 Ex3: Power Method with GPUs **12**

1 Ex1: Poisson Solver with Finite Differences

The first excercise consists of solving the two dimensional Poisson's equation with Finite Differences and utilize the parallel programming paradigm using the Message Passing Interface (`MPI`) library. We omit specifics here. The exact mathematical description along with the solution strategies are given in [1].

1.1 Building a parallel version using `MPI`

With the help of the sequential Poisson solver, `SEQ_Poisson.c`, the build of important tasks of the parallel version is described in this section.

Step 1

To execute clones of the original code in a number of processes, the following additions are made to the main program of the sequential code.

```
MPI_Init(&argc, &argv);
MPI_Finalize();
```

The program was executed twice since two pairs of statements were written to the terminal in comparison to one pair before the modifications. Since we are now running the same program in two different nodes this behavior is expected.

Step 4

Each process wrote to a separate file that does not affect the text displayed. In addition, `diff` showed no differences between the 2 outputs `output0.dat` and `output1.dat`.

Step 5

With the following code incorporated in `Setup_Grid()`, only the process with rank 0 reads data from an input file and subsequently broadcasts this data to all processes. Note that the dots represent code already given in the questions.

```
if(proc_rank==0){
   .
   .
   .
}
MPI_Bcast(&gridsize ,2,MPI_INT ,0,MPI_COMM_WORLD);
MPI_Bcast(&precision_goal ,1,MPI_DOUBLE ,0,MPI_COMM_WORLD);
MPI_Bcast(&max_iter ,1,MPI_INT ,0,MPI_COMM_WORLD);

do {
  if(proc_rank==0)
  ...
  MPI_Bcast(&s,1,MPI_DOUBLE ,0,MPI_COMM_WORLD);
  if(s==3){
    MPI_Bcast(&source_x ,1,MPI_DOUBLE ,0,MPI_COMM_WORLD);
    MPI_Bcast(&source_y ,1,MPI_DOUBLE ,0,MPI_COMM_WORLD);
    MPI_Bcast(&source_val ,1,MPI_DOUBLE ,0,MPI_COMM_WORLD);
    .
    .
    .
  }
}
while (n==3);
if(proc_rank==0) fclose(f);
```

This step demonstrates the use of `MPI_Bcast`.

Step 6

This step helps to assign specific tasks to each process. The following are added to `Setup_Proc_grid()`

```
// Retrieve the no of processes
MPI_Comm_size(MPI_COMM_WORLD ,&P);

//new comm
MPI_Cart_create(MPI_COMM_WORLD ,2,P_grid,periods,reorder ,&grid_comm);

// Retrieve new rank and cartesian coordinates of this process
MPI_Comm_rank(grid_comm ,&proc_rank);
MPI_Cart_coords(grid_comm ,proc_rank ,2,proc_coord);

//calculate ranks of neighbouring processes
MPI_Cart_shift(grid_comm ,1,1,&proc_bottom ,&proc_top);
MPI_Cart_shift(grid_comm ,0,1,&proc_left ,&proc_right);
```

Step 8

This step is to setup `MPI_Datatypes()` which is a datatype that makes communication convenient, and to perform communication. The following code is used

```
void Exchange_Borders()
{

 resume_timer();
 Debug("Exchange_Borders",0);

 MPI_Sendrecv(&phi[1][dim[Y_DIR]-2],1,border_type[Y_DIR], proc_top,1,
          &phi[1][0],1,border_type[Y_DIR], proc_bottom, 1,
          grid_comm, &status);

 MPI_Sendrecv(&phi[1][1],1,border_type[Y_DIR],proc_bottom,2,
              &phi[1][dim[Y_DIR]-1],1,border_type[Y_DIR],proc_top,2,
              grid_comm,&status);

 MPI_Sendrecv(&phi[1][1],1,border_type[X_DIR],proc_left,3,
              &phi[dim[X_DIR]-1][1],1,border_type[X_DIR],proc_right,3,
              grid_comm,&status);

 MPI_Sendrecv(&phi[dim[X_DIR]-2][1],1,border_type[X_DIR],proc_right,4,
              &phi[0][1],1,border_type[X_DIR],proc_left,4,
              grid_comm,&status);
 stop_timer();
 data_communicated += (2 * dim[X_DIR] + 2 * dim[Y_DIR] - 8);

}
```

1.2 Performance Experiments

This section will describe the results of *experiments* performed on the parallel version of the code described in the previous section. We follow the guidelines given in [2].

2.1

Refering to eq.(1) &(2) in [2], the grid point is updated as,

$$c_{i,j}^n = \frac{\phi_{i-1,j}^n + \phi_{i+1,j}^n + \phi_{i,j-1}^n + \phi_{i,j+1}^n - h^2 S_{i,j}}{4} - \phi_{i,j} \tag{1}$$

$$\phi_{i,j}^{n+1} = \phi_{i,j}^n + \omega c_{i,j}^n \tag{2}$$

We return to the discussion of an optimal relaxation ω in the next step. Implementation of the algorithmic improvement can be found in the `Do_Step()` routine in `MPI_Poisson.c`. Here we show the corresponding snippet implementing the above.

```
old_phi = phi[x][y];
c = (phi[x + 1][y] + phi[x - 1][y] +
phi[x][y + 1] + phi[x][y - 1]) * 0.25;
phi[x][y] = (1.0 - w)*old_phi + w*c;
```

2.2

After implementing the improvement, 10 tests were performed to compare different values for the relaxation parameter ω. The results are summarized in table. From this data we concluded that 1.934 is the optimal value for the relaxation parameter, accomplishing almost 19 times less iterations than the original with ω equal to one.

ω	Wtime (s)	n	Reduction
1	0.194709	2355	1
1.9	0.041364	220	10.70
1.91	0.037918	194	12.13
1.92	0.0369	165	14.27
1.93	0.032486	131	17.97
1.934	**0.032996**	**125**	**18.84**
1.936	0.032805	130	18.11
1.94	0.033853	142	16.58
1.96	0.03741	206	11.43
1.98	0.059543	419	5.62

Table 1: Time, number of iterations obtained and respective iteration reduction for different ω values. The topology used was pt:441 with g:100x100

2.3

The goal of this exercise is to investigate the scaling behavior of the code with a fixed relaxation parameter. To accomplish that analysis several runs were measured with various grid sizes. In addition different slices were also tested. The results of the aforementioned experiment are shown in table 2. As one can obverse the different domain partitions have little impact in the overall performance of the program (speedup in the table measures ratio of pt:422 wrt pt:441), even as the grid size increases. As explained in [2] the (average) time t per iteration n can be parametrized as follows,

$$t(n) = \alpha + \beta n$$

where α and β are constants that are determined by running the parallel program for fixed number of iterations. Table 3 describes the constants determined for the same processor topologies and grid sizes as defined in table 3

In order to improve the quality of this an estimation a thorougher study should be performed with more runs. In addition considering other (non- linear) parametrization functions, e.g. exponential, may also yield interesting results.

Wtime(s)				
g	n	pt: 441	pt:422	Speedup
200	50	0.036	0.019	1.89
	100	0.052	0.038	1.37
	200	0.081	0.064	1.27
	300	0.11	0.089	1.24
400	100	0.123	0.108	1.14
	300	0.299	0.286	1.05
	500	0.473	0.462	1.02
	1000	0.862	0.854	1.01
800	100	0.371	0.364	1.02
	300	0.979	0.965	1.01
	500	1.493	1.486	1.00
	1000	2.787	2.777	1.00
	2000	5.332	5.36	0.99
2000	100	1.81	1.786	1.01
	300	4.985	5	1.00
	500	8.36	8.145	1.03
	1000	16.162	16.149	1.00
	2000	31.886	32.142	0.99
3500	100	5.144	5.071	1.01
	300	14.766	14.912	0.99
	500	24.699	24.585	1.00
	1000	48.828	49.286	0.99
	2000	97.911	98.351	1.00

Table 2: The maximum time for different grid sizes and different slicing arrangements. The number of iterations for each run was fixed to enable a more accurate comparison.

	pt:441		pt:422	
g	β	α	β	α
200	0.0002946	0.0219	0.0003	0.0079
400	0.0008	0.0508	0.0008	0.0355
800	0.0026	0.1721	0.0026	0.1548
2000	0.0158	0.3079	0.0160	0.1837
3500	0.0488	0.1847	0.0491	0.1367

Table 3: Using least-squares method we estimated α and β as defined above. The dataset adopted for this computation is present in Table

2.4

Based on the results from the previous question it is expected that the division of the domains does not result is significantly different running times. Thus the choice can be arbitrary.

2.5

For this, we explore the convergence criteria (monitor the no. of iterations) for various problem sizes. The file input.dat is altered so as to perform computations for various grid sizes. The results can be summarized by the following table and the plot below.

g	200	300	400	500
n	382	771	1206	1664

Table 4: No. of iterations for various grid sizes ($\omega = 1.95$).

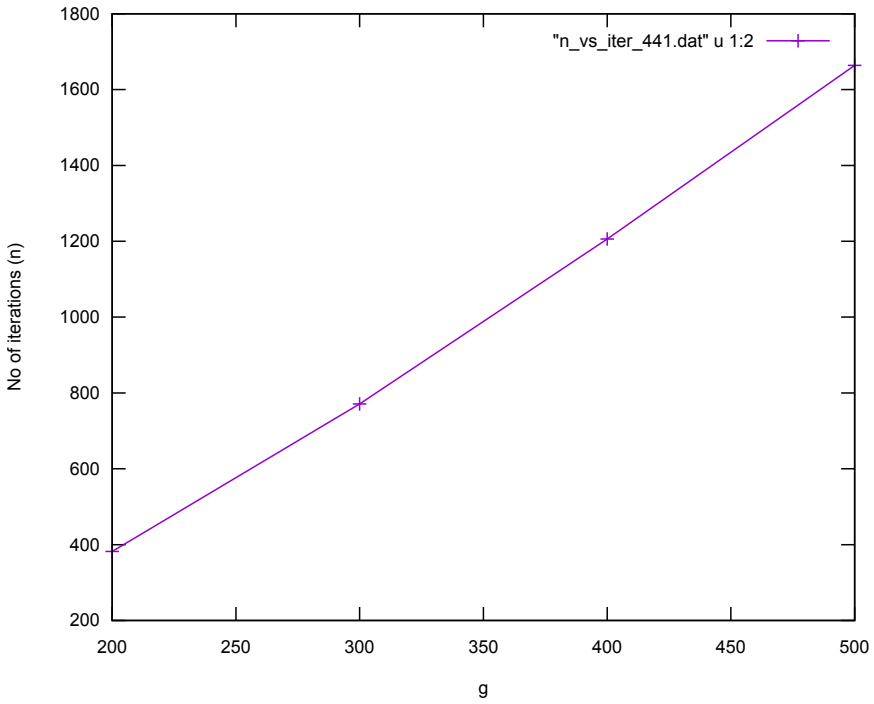

Figure 1: g vs n

Figure 1 illustrates that as the Problem (grid) size increases, the Number of iterations needed to attain convergence also increases. Computations are done for a topology 4×1 (pt:441).

2.6

For this, the behaviour is monitored as a function of iteration number. The number ofiterations considered in the x-axis and the magnitude of the error is taken in the y-axis.An optimal ω of 1.934 was chosen. As expected (also in the plot **??**), the error seems to decrease well enough indicating that, with the implementation of Gauss-Seidel with SOR with Red-Black results in earlier convergence than the normal Gauss-Seidel.

2.7

In an attempt to reduce communication, the global error is calculated and communicated once in 10 iterations. Table 5 shows the maximum execution time for the problem with grid size 100 and a 4×1 topology when the global error is communicated once in every 1, 10 and 100 iterations. Note that k is the frequency of global error communication and t is the maximum execution time.

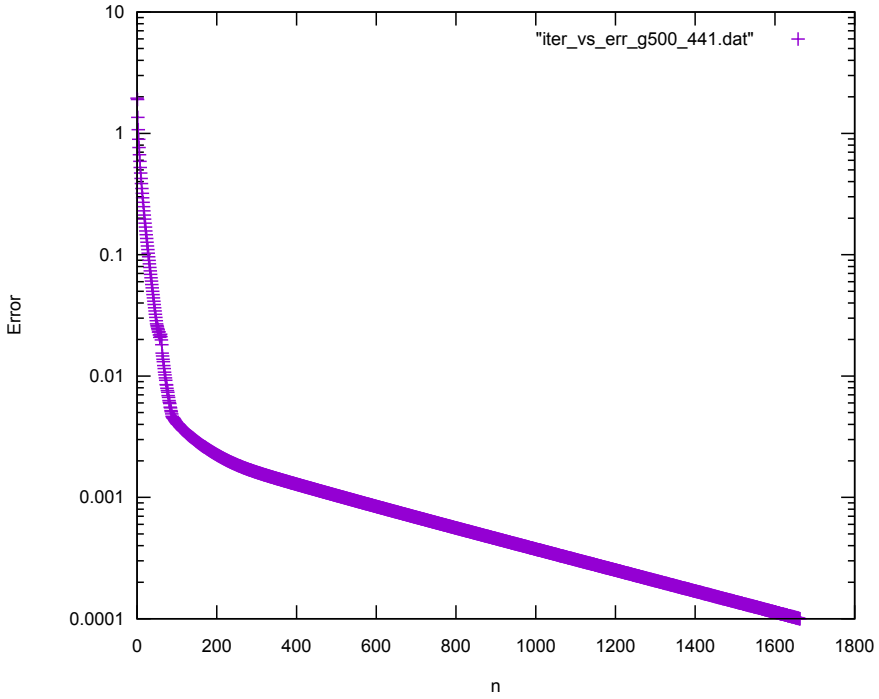

Figure 2: n vs Error for pt:441 and g=500

k	1	10	100
t	0.044293	0.042382	0.045240
n	166	171	201

Table 5: Communication reduction 1

It is found that the communication time and hence the maximum execution time does reduce for a frequency of 10 iterations. However, for a frequency of 100 iterations, the reduction in communication time is superseded by the increase in the number of total iterations needed for convergence. Thus, the total execution time for k = 100 is higher as seen in table 5. The following code is added to the function Solve() for this experiment.

```
if(count%10==0)
MPI_Allreduce(&delta,&global_delta,1,MPI_DOUBLE,MPI_MAX,grid_comm);
```

2.8

In another attempt to reduce communication, the red-black sweep of the Gauss-Seidel method is performed more than once between two border exchanges. One red sweep followed by one black sweep is called one full sweep. We perform more than one full sweeps before exchanging the borders. Note that a half sweep means that the borders are exchanged also between red and black sweeps of a full sweep as done in previous exercises. Table 6 shows the results obtained.

k	1/2	1	2	3
t	0.050375	0.374343	0.679657	0.961812
n	166	5000	4883	5000

Table 6: Communication reduction 2

It is evident from the results that the essence of the red-black SOR method is lost by increasing the number of full sweeps between border exchanges. It is also found that despite the solution converging for k=2, the solution is *nan*. Hence it is clear that border exchanges are necessary after every half sweep for convergence.

For this experiment, the code in the while loop of the Solve() function is modified as follows.

```
int k=0;

while(k<3)
{
  Debug("Do_Step 0", 0);
  delta1 = Do_Step(0);

  Debug("Do_Step 1", 0);
  delta2 = Do_Step(1);

  k++;
}

Exchange_Borders();

delta = max(delta1, delta2);
```

2.9

To sweep only over the points that are to be updated, the inner for loop in the Do_Step() function can be modified as shown in the following code. The logic in the inner for loop statement sweeps only over the necessary points thus eliminating the need for a parity check.

```
 for (x = 1; x < dim[X_DIR] - 1; x++)
  for (y = ((x + offset[X_DIR]) % 2)*(1+parity)
  + !((x + offset[X_DIR]) % 2)*(2-parity); y < dim[Y_DIR] - 1; y+=2)
 if (source[x][y] != 1){
  .
  .
  .
}
```

For the test problem (pt=441, n=100), the maximum execution reduces from 0.068726 sec to 0.055967 sec by doing the above mentioned modification which ensures that the sweep is only over the points that are supposed to be updated in the red and black iterations. For bigger problem sizes the performance significantly goes up. For the problem with 414 topology and n=1600, maximum execution time reduces from 46.16542 sec to 37.72682 sec.

2.10

For this experiment, the timer is resumed before the `Exchange_Borders()` function and stopped right after it as shown in the following code in the `Do_Step()` function.

```
 resume_timer();
 Exchange_Borders();
 stop_timer();
```

The exchange border time is measured as a function of number of processes and a function of problem size and the results are tabulated in tables 7 and 8.

g	200	400	800	1000	2000	3500
t	0.026389	0.031386	0.080393	0.107828	0.355372	2.019741

Table 7: Exchange border time as a function of grid size (pt= 441)

pt	221	441	422	881	842
t	0.002634	0.04723	0.005687	0.045410	0.047286
T	0.028546	0.056971	0.020753	0.061953	0.054358

Table 8: Exchange border time as a function of no. of processes (g=100)

It can be seen in table 7 that the time spent in exchanging borders is close to the time spent on computations for the following problem: g=200, pt=441.

2.11

The overhead of the parallel program is given by,

$$T_o = pT_p - T_s$$

where T_p is the parallel execution time and T_s is the serial execution time. The overhead caused by exchanging borders can be calculated by considering the SOR algorithm with $\omega = 1$ and comparing it with the sequential code. For the configurations and grid sizes from exercise 2.3, the overheads are given in the table 9.

Proc. topology	Grid size	T_p	T_s	T_o
4×1	200	0.195640	0.46	0.3216
	400	0.978629	2.86	1.0535
	800	2.673317	9.65	1.0391
	1000	5.853645	22.54	0.8752
	2000	22.928791	90.24	1.4641
2×2	200	0.172482	0.46	0.2299
	400	0.968356	2.86	1.01
	800	2.87270	9.65	1.8478
	1000	5.924463	22.54	1.1570
	2000	23.025853	90.24	1.8754

Table 9: Overheads

2.12

The impact of the communication overhead is limited. Even for large grid sizes halving the amount of data sent is likely to cause, at best, moderate improvements. In addition the maintainability of the code would decrease rapidly as such modifications would hamper readability and comprehension of the code base. Nonetheless to accommodate this optimization the address of the first point to exchange would have to be calculated differently, taking into account the parity value. Moreover the size of the "jump" between transmittable points would also need to be modified. Such a change would best be done in the `Setup_MPI_Datatypes()` where the custom data types are defined. The greatest advantage would be the 50% reduction number of points to exchange.

2 Ex2: Poisson Solver with Finite Elements

This exercise involves the red-black Gauss-Seidel method applied to finite element problems.

2.1 Border exchange

To perform data communication, the following code is added to the Exchange_Borders() function.

```
void Exchange_Borders(double *vect)
{
 int i;
 for(i = 0; i < N_neighb; i++)
   MPI_Sendrecv(vect,1,send_type[i],proc_neighb[i],0,
                vect,1,recv_type[i],proc_neighb[i],0,
                grid_comm,&status);
       }
```

2.2 Benchmarking

In order to benchmark the MPI_Fempois.c program, we consider the Solve() function which involves point-to-point communication and global communication. Point-to-point communication is done to communicate the function values to neighboring processes whereas global communication is done to communicate the dot product from each process to all processes which is done using the MPI_Allreduce() function. We choose the following for benchmarking.

- Computation time
- Border exchange time
- Global communication time
- Idle time

Computation time is calculated for the function Solve() by masking the global communication function MPI_Allreduce() and the border exchange function, and considering only the computations. Global communication time and border exchange time are calculated by wrapping the global communication function MPI_Allreduce() and the Exchange_Borders() function in a timer respectively. Finally, idle time is calculated using the following relation.

$$\text{Idle time} = \text{total time} - \text{computation time} - \text{communication time}$$

where total time is the time taken by the Solve() function which is calculated by wrapping the Solve() function around a timer and communication time involves both global communication and border exchanges. Note that this benchmarking corresponds only to the Solve() function and not to the entire program. This means that the idle time signifies the time a processor is idle when doing *computations*. The results are tabulated in table 10.

Pt	g	wtime	Computation	Global comm	Exchange borders	Idle
414	100	0.074315	0.040482	0.002495	0.013155	0.000082
	200	0.306800	0.233759	0.004113	0.013163	0.000123
	400	1.795695	1.411101	0.145229	0.057769	0.000256
422	100	0.073564	0.040351	0.002792	0.012588	0.000068
	200	0.308903	0.231843	0.004269	0.012930	0.000128
	400	1.675330	1.418528	0.061721	0.030194	0.000229

Table 10: Benchmarking: task times in seconds

The times calculated will be displayed on the terminal when the program is run. For more details, refer to the MPI_Fempois.c program.

2.3 Data transfer

2.4 Asymmetry

The asymmetry of border exchanges comes from the way of grid partitioning of the finite element method. The N,E,W,S partitioning which applies to FDM, is not suitable for FEM since the domain is triangulated. 2 processes communicate with three neighbors whereas the others communicate with two neighbors.

Similarly, in a 3 × 3 partitioning, the process at the center could possibly communicate with at most 8 neighbors (can be imagined as an octagon at the center) and the processes at the corners could communicate with 3 processes at most.

2.5 Data locality ratio

With four processes, the computation time is almost equal to the communication time for a grid size of 73 × 73. For the problem size 1000 × 1000, computation time far exceeds communication time despite using 32 processes, making it difficult to determine the times for a data locality ratio of 1 by experiments.

2.6 Mesh refinement

Mesh refinement is done by using the command *adapt* and the results are tabulated as follows. tcomp is computation time and adapt refers to the refined mesh.

g	n	n adapt	t	t adapt	tcomp	tcomp adapt
100	141	146	0.080618	0.083603	0.034146	0.046426
200	274	278	0.312281	0.314155	0.228471	0.229128
400	529	532	1.751257	1.766172	1.421017	1.441059

Table 11: Mesh refinement

It is evident from table 11 that the number of iterations, execution time and computation time all increase due to mesh refinement. This is due to the fact that mesh refinement is done to increase the accuracy of the solution by adding more points in the region of interest. More points, more is the computation and more is the number of iterations needed for convergence, which is also intuitive.

3 Ex3: Power Method with GPUs

Using the sequential version of the code `power_cpu.cu`, we build our CUDA implementation of the Power Method problem. We conduct several comparisons of solving eigenvalue problem using power method on CPU and GPU. The essence of power method includes parallelization of matrix-vector multiplication. We skip Step 4 since we explain our conclusions alongside all corresponding results. Unless otherwise mentioned, all times are shown in seconds.

Step 1

We compare the speed differences between **shared** and **global** memory versions, implemented in `power_gpu_shr.cu` and `power_gpu_glb.cu`. The kernels using global memory implemented are as follows:

```
/Kernels
__global__ void Av_Product(float* g_MatA, float* g_VecV, float* g_VecW, int N)
{
  //global thread index
  unsigned int globalid = threadIdx.x + blockIdx.x*blockDim.x;
  float sum = 0;
  if(globalid < N){
    for(int i=0; i<N; i++)
      sum+= g_MatA[i+(globalid*N)]*g_VecV[i];
    g_VecW[globalid]=sum;
  }
}

__global__ void FindNormW(float* g_VecW, float * g_NormW, int N)
{
  unsigned int globalid = threadIdx.x + blockIdx.x*blockDim.x;
  if(globalid < N)
    atomicAdd(g_NormW,g_VecW[globalid]*g_VecW[globalid]);
}

__global__ void NormalizeW(float* g_VecW, float * g_NormW, float* g_VecV, int N)
{
  unsigned int globalid = threadIdx.x + blockIdx.x*blockDim.x;
  if(globalid < N)
    g_VecV[globalid]= g_VecW[globalid]/g_NormW[0];
}

__global__ void ComputeLamda( float* g_VecV, float* g_VecW, float * g_Lamda,int N)
{
  unsigned int globalid = threadIdx.x + blockIdx.x*blockDim.x;
  if(globalid < N)
    atomicAdd(g_Lamda, g_VecV[globalid] * g_VecW[globalid]);
}
```

With the help of shared kernels we list our performance metrics in step 2 for various matrix sizes (N).

Step 2

Table 12 shows the total computation times taken by CPU purely and the GPU in both variants of the memory implementations. Clearly, the shared memory version performs much faster than the global memory version.

N	CPU	GPU_global	GPU_shared
50	0.000171	0.001554	0.001479
500	0.015287	0.002162	0.002243
2000	0.218504	0.008506	0.006295
5000	1.144645	0.036006	0.020887

Table 12: Total compute times (in secs) for CPU and GPU(both memory versions) done for a fixed iteration count of 10 for accurate estimations. ThreadsPerBlock=32

CPU times are comparable when N is small but GPUs outperform as N scales to larger values. Let us now observe the execution times for a matrix A with different sizes and different threads. Let us consider the shared version of the GPU implementation. As we see that for number of threads per block becomes 128, the GPU runs out of shared memory space, and thus we are unable to run the program. In contrast to the global memory

N\ThreadsPerBlock	32	64	128
50	0.001479	0.001482	-
500	0.002243	0.002197	-
2000	0.006295	0.006042	-
5000	0.020887	0.028341	-

Table 13: total compute times(in secs) of shared memory implementation with Different threads and Matrix sizes (N).

implementation, we see that 128 threads can be used to successfully compute the solution. There is no observed

N\ThreadsPerBlock	32	64	128
50	0.001554	0.001529	0.001414
500	0.002162	0.002212	0.002318
2000	0.008506	0.006922	0.006932
5000	0.036006	0.034855	0.037112

Table 14: compute times (in secs) for global memory version

significance in computing times with increase in threads. They roughly compute within same time margins. We can conclude that for increase in performance, shared memory version is better but comes at tradeoff of how much data can be stored in the shared memory space.

Step 3

Let us now analyze the times spent in memory transfers. We compare these results with times elapsed excluding memory transfers.

As we see in 15 the speedup ratios (time taken by global version wrt time taken by shared version) calculated

	T_memory		T_total			
N	**shared**	**global**	**shared**	**global**	**SpeedUp(Inc.Mem)**	**SpeedUp(ExcMem)**
50	0.000397	0.000444	0.001479	0.001554	1.050709939	0.9747747748
500	0.000699	0.000716	0.002243	0.002162	0.9638876505	1.067773167
2000	0.002323	0.00381	0.006295	0.008506	1.351231136	0.8458262351
5000	0.011897	0.014482	0.020887	0.036006	1.723847369	0.4176732949

Table 15: memory times (T_memory) and total compute times (T_total) for different N wrt 32 threads.

in the last 2 columns, show the stark contrast between a) inclusion of memory transfer times and b) exclusion of memory transfer times from the total compute times. The SpeedUp ratios calculated in the latter case show that the shared memory version actually performs worse than the global memory version if we neglect the memory transfers taken place between CPU and GPU. Hence this allows us to conclude that the advantage of the shared memory version is because of utilization of the on-chip memory. The memory access from global memory is slower than the latter on the device, hence the performance difference.

References

dr. A.G.M van Hees. (2002a). *Laboratory excercises (1) introduction hpc.* (TU Delft)

dr. A.G.M van Hees. (2002b). *Laboratory excercises (2) introduction hpc.* (TU Delft)

Kirk, D. B., & Wen-Mei, W. H. (2016). *Programming massively parallel processors: a hands-on approach.* Morgan kaufmann.

Lin, H. X. (2020). *Introduction to high performace computing (in4049).* (Couse Slides)