CONCURRENCE, PARALLELISM AND DISTRIBUTED SYSTEMS

Assignment 2
Solving the Heat Equation Using Several Parallel Programming Models

Mehdi Hassanpour - cpds1124
mehdi.hassanpour@upc.edu

Fall 2024-25
Polytechnic University of Catalonia

22-10-2024

Student's Note

Due to being in a different academic year from most of my peers, combined with challenging experiences in past group projects, I completed this assignment independently rather than as part of a typical two-person team. Although this added to the workload, it provided a unique opportunity to immerse myself in each detail of the project, enhancing my comprehension of the material. I apologize if this decision has caused any inconveniences.

Contents

1 Shared–Memory Parallelization with OpenMP

1.1 Jacobi

The goal is to assign different blocks within the grid to different threads, ensuring that computations in each iteration only depend on values from the previous iteration. All updates during the current iteration can proceed in parallel, as they do not rely on the newly computed values from the same iteration. This independence eliminates the need for synchronization within the current iteration, as each update only requires ”old” values from the previous iteration. Below is the code before applying OpenMP.

```c
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey)
{
    double diff, sum=0.0;
    int nbx, bx, nby, by;

    nbx = NB;
    bx = sizex/nbx;
    nby = NB;
    by = sizey/nby;
    for (int ii=0; ii<nbx; ii++)
        for (int jj=0; jj<nby; jj++)
            for (int i=1+ii*bx; i<=min((ii+1)*bx, sizex-2); i++)
                for (int j=1+jj*by; j<=min((jj+1)*by, sizey-2); j++) {
                    utmp[i*sizey+j]= 0.25 * (u[i*sizey + (j-1)]+  // left
                                             u[i*sizey + (j+1)]+  // right
                                             u[(i-1)*sizey + j]+  // top
                                             u[(i+1)*sizey + j]); // bottom
                    diff = utmp[i*sizey+j] - u[i*sizey + j];
                    sum += diff * diff;
                }

    return sum;
}
```

In this approach, threads can operate on separate sections of the grid simultaneously without causing conflicts. OpenMP is used to parallelize the block assignments, which involves removing one of the four nested loops by distributing iterations over different threads. Specifically, the double loop over `ii` and `jj` (responsible for block assignments) is collapsed using the OpenMP collapse clause, allowing each block to be assigned to a different thread for concurrent execution.

Since this code is executed in a shared-memory environment, it is essential to handle potential race conditions when updating values. The variable `diff` is private to each thread and is recalculated independently within each thread’s execution. In contrast, the arrays `u` and `utmp` must be shared across all threads, as they store the grid values. However, since each thread writes to distinct elements of `utmp`, there is no risk of race conditions during these updates.

Finally, the variable `sum`, which accumulates the total difference across all threads, must be handled carefully to avoid conflicts. Using OpenMP’s reduction clause, we ensure that the sum of the `diff` values from all threads is computed safely and efficiently, preventing any race conditions during accumulation.
We change the code to the following:

```c
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey)
{
    double diff, sum=0.0;
    int nbx, bx, nby, by;
    nbx = NB;
    bx = sizex/nbx;
    nby = NB;
    by = sizey/nby;
    #pragma omp parallel for collapse(2) private(diff) reduction(+:sum)
        shared(u, utmp)
    for (int ii=0; ii<nbx; ii++)
        for (int jj=0; jj<nby; jj++)
            for (int i=1+ii*bx; i<=min((ii+1)*bx, sizex-2); i++)
                for (int j=1+jj*by; j<=min((jj+1)*by, sizey-2); j++) {
                    utmp[i*sizey+j]= 0.25 * (u[i*sizey + (j-1)]+  // left
                                             u[i*sizey + (j+1)]+  // right
                                             u[(i-1)*sizey + j]+  // top
                                             u[(i+1)*sizey + j]); // bottom
                    diff = utmp[i*sizey+j] - u[i*sizey + j];
                    sum += diff * diff;
                }

    return sum;
}
```

1.2 Gauss-Seidel

1.2.1 Explicit Tasks with Dependencies

To parallelize the Gauss-Seidel algorithm using OpenMP's task dependency model, the grid is divided into blocks, and each block is assigned to a separate task. The grid is subdivided along the x and y dimensions, with the number of blocks determined by `nbx` and `nby`, respectively. Block sizes bx and by are calculated to fit the grid as evenly as possible, allowing adjustments for uneven grid sizes.

The outer loops over the block indices (`ii` and `jj`) are enclosed within an OpenMP single directive to ensure only one thread handles task creation. Within each block iteration, an OpenMP task is created, allowing concurrent execution of tasks. To ensure correctness in the Gauss-Seidel method, which relies on data from neighboring cells, task dependencies are implemented using the depend clause. Each task has an input dependency (`depend(in:)`) on the blocks directly above (`aux_depend[ii-1][jj]`) and to the left (`aux_depend[ii][jj-1]`) to enforce a sequence. This ensures that tasks do not start until the neighboring data is available. Additionally, each task has an output dependency (`depend(out:)`) on its own `aux_depend[ii][jj]` to prevent any other tasks from modifying the same block concurrently.

The use of `aux_depend` as a dependency array provides an effective way to track the dependencies between blocks. Each block waits for the completion of its preceding tasks based on this dependency matrix, which corresponds to the required sequential order in the Gauss-Seidel update pattern.

Inside each task, the primary grid array `u` and the global `sum` are shared to enable consistent updates across tasks. Intermediate variables `unew` and `diff` are private to each task, preventing conflicts during calculations. Each task calculates a local `omp_sum` for the block's squared differences, which is added atomically to the global sum to prevent race conditions.

This approach enables each grid block to be processed in parallel while respecting dependencies between tasks. The OpenMP runtime manages task scheduling based on the dependency clauses, allowing the Gauss-Seidel algorithm to execute correctly with task-based parallelism. The combination of task-based parallelization and dependency control achieves efficient execution with necessary synchronization between neighboring grid points.

```
bx = sizex / nbx + (sizex % nbx != 0);
by = sizey / nby + (sizey % nby != 0);
int aux_depend[nbx][nby];
#pragma omp parallel
{
    #pragma omp single
    {
        for (int ii=0; ii<nbx; ii++) {
            for (int jj=0; jj<nby; jj++) {
            #pragma omp task depend(in: aux_depend[ii-1][jj], aux_depend[ii][
                jj-1]) depend(out: aux_depend[ii][jj]) private(diff, unew)
                {
                    double omp_sum = 0.0;
                    for (int i = 1 + ii * bx; i <= min((ii + 1) * bx, sizex -
                         2); i++) {
                         for (int j = 1 + jj * by; j <= min((jj + 1) * by,
                             sizey - 2); j++) {
                            unew= 0.25 * (u[i*sizey + (j-1)]+  // left
                                          u[i*sizey + (j+1)]+  // right
                                          u[(i-1)*sizey + j]+  // top
                                          u[(i+1)*sizey + j]); // bottom
                            diff = unew - u[i*sizey+ j];
                            omp_sum += diff * diff;
                            u[i*sizey+j]=unew;
                         }
                    }
                    #pragma omp atomic
                    sum += omp_sum;
                }
            }
        }
    }
}
```

1.2.2 Do-Across

The `#pragma omp parallel for collapse(2)` directive allows the outer loops over block indices to be executed in parallel, effectively merging two nested loops to increase the granularity of work distribution. The `reduction(+:sum)` clause ensures safe accumulation of the total difference across threads, preventing race conditions. The `ordered(2)` clause maintains the execution order, particularly for the second loop, enabling correct updates based on dependencies.

Task dependencies are established with `#pragma omp ordered depend(sink: ii - 1, jj)` and `#pragma omp ordered depend(source)`, ensuring that each block waits for the completion of its neighboring blocks before proceeding. This is critical for the correctness of the Gauss-Seidel algorithm, which relies on up-to-date neighboring values.

```c
double relax_gauss(double *u, unsigned sizex, unsigned sizey) {
    double unew, diff, sum = 0.0;
    int nbx, bx, nby, by;
    nbx = NB;
    bx = sizex / nbx + (sizex % nbx != 0);
    nby = NB;
    by = sizey / nby + (sizey % nby != 0);
    #pragma omp parallel for collapse(2) reduction(+:sum) ordered(2)
    for (int ii = 0; ii < nbx; ii++) {
        for (int jj = 0; jj < nby; jj++) {
            #pragma omp ordered depend(sink: ii-1, jj) depend(sink: ii, jj-1)
            {
                for (int i = 1 + ii * bx; i <= min((ii + 1) * bx, sizex - 2);
                     i++) {
                    for (int j = 1 + jj * by; j <= min((jj + 1) * by, sizey -
                         2); j++) {
                        unew = 0.25 * (u[i * sizey + (j - 1)] +  // left
                                       u[i * sizey + (j + 1)] +  // right
                                       u[(i - 1) * sizey + j] +  // top
                                       u[(i + 1) * sizey + j]);  // bottom
                        diff = unew - u[i * sizey + j];
                        sum += diff * diff;
                        u[i * sizey + j] = unew;
                    }
                }
            }
            #pragma omp ordered depend(source)
        }
    }
    return sum;
}
```

1.3 Results and Analysis

Table 1 and figures 1 present the performance results of applying OpenMP to various algorithms used in this assignment. The data shown represents the speedup achieved by parallelization, using the baseline as the elapsed time for each algorithm running on a single thread without parallelization.

Across both grid resolutions (256x256 and 512x512), the speedup trends are similar, though differences in performance become more pronounced with the larger problem size.

The Jacobi algorithm generally achieves higher speedups (compared to Gauss-Seidel), particularly for smaller thread counts. However, the benefits of parallelism diminish beyond two threads, suggesting that Jacobi's structure limits scalability at higher thread counts.

In contrast, the Gauss-Seidel algorithm demonstrates limited scalability across both grid sizes when parallelized using the explicit task-based method. This approach introduces dependencies that limit parallel efficiency, particularly as the grid resolution increases, making the synchronization overhead more pronounced.

However, the Do-Across method with Gauss-Seidel achieves more promising results, reaching peak speedup with four threads. This method minimizes synchronization overhead compared to explicit tasks by taking advantage of loop independence, making it a more effective parallelization strategy for Gauss-Seidel in this context.

			# Threads				
Resolution	Algorithm		1	2	4	8	16
256	Jacobi		2.065	2.915	1.894	2.047	1.811
	Gauss-Seidel	Explicit	1.652	1.501	1.421	1.355	1.318
		DoAcross	1.567	1.629	1.817	1.646	1.633
512	Jacobi		1.927	2.808	1.772	2.173	2.343
	Gauss-Seidel	Explicit	1.714	1.598	1.494	1.464	1.454
		DoAcross	1.619	1.694	2.021	1.782	1.738

Table 1: Speedup of different algorithms using OpenMP over different number of threads and problem resolutions.

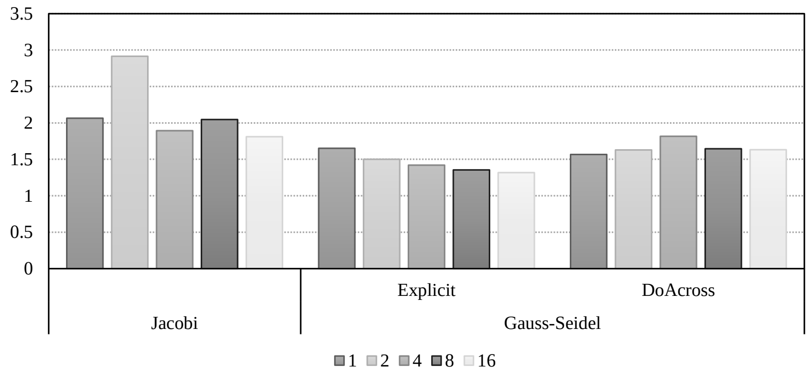

Figure 1: Comparison of speedup using OpenMP for an image of size 256x256

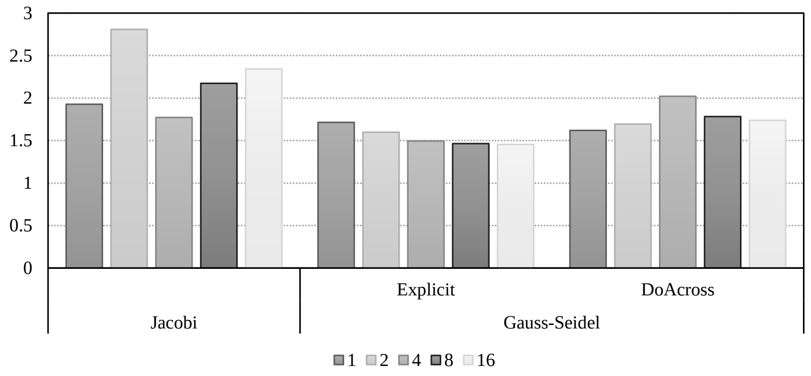

Figure 2: Comparison of speedup using OpenMP for an image of size 512x512

2 Message–Passing Parallelization with MPI

2.1 Jacobi

The first step is to distribute the work between the master and worker processes by creating and sending relevant information to the workers. The workload for each worker is specified by the parameter `rows_per_proc`. In the code snippet below, this parameter is adjusted, and each worker receives the appropriate subset of the matrix along with its corresponding data sizes. Only the modified or newly added sections of code are shown here, while the complete code is available in the attached files.

```c
if (myid == 0) { // master node
    rows_per_proc = param.resolution / numprocs;
    ...
    // send to workers the necessary data to perform computation
    for (int i=0; i<numprocs; i++) {
        if (i>0) {
                MPI_Send(&param.maxiter, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
                MPI_Send(&param.resolution, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
                MPI_Send(&param.algorithm, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
                MPI_Send(&param.u[i * rows_per_proc * np], (np)*(rows_per_proc + 2), MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
                MPI_Send(&param.uhelp[i * rows_per_proc * np], (np)*(rows_per_proc + 2), MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
        }
    }
}
```

From the slides, we apply the following parallelization strategy for the Jacobi method:

1. Exchange boundary values with the two adjacent processors.
2. Each processor computes the elements within its assigned `utmp` segment.

Adjacent cells have unique corner cases based on the specific row being processed. For instance, in the first row (managed by the master node), data is only sent to the next processor. Conversely, in the last row, data is only received from the previous processor. These cases are managed by two conditions: `(myid == 0)` for the master node and `(myid != numprocs-1)` for the last processor.

```c
// myid = 0, I'm master
case 0: // JACOBI
    if (numprocs > 1) {
        MPI_Send(&param.u[np * rows_per_proc], np, MPI_DOUBLE, myid+1, 0, MPI_COMM_WORLD);
        MPI_Recv(&param.u[np * (rows_per_proc+1)], np, MPI_DOUBLE, myid+1, 0, MPI_COMM_WORLD, &status);
    }
    residual = relax_jacobi(param.u, param.uhelp, rows_per_proc+2, np);
    // Copy uhelp into u
    double* temp_u = param.u;
    param.u = param.uhelp;
    param.uhelp = temp_u;
```

Additionally, a pointer swap is applied instead of copying `uhelp` directly into `u`, which is intended to bypass the nested loop and enhance processing speed. Convergence is determined by computing the residual here, using `Allreduce` (as in assignment 1).

```
MPI_Allreduce(MPI_IN_PLACE, &residual, 1,MPI_DOUBLE, MPI_SUM,MPI_COMM_WORLD);
...
// receive the partial results form workers
for (int i=1; i < numprocs; i++) {
    MPI_Recv(&param.u[np * (rows_per_proc * i + 1)], rows_per_proc * np,
        MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &status);
}
```

Based on the given description, the snippet below includes only the modifications for non-master nodes, presenting only the key lines for the reader's convenience.

```
// myid != 0, I'm a worker
MPI_Recv(&u[0], (rows_per_proc+2)*(np), MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &
    status);
MPI_Recv(&uhelp[0], (rows_per_proc+2)*(np), MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
     &status)
...
case 0: // JACOBI
    MPI_Send(&u[np], np, MPI_DOUBLE, myid-1, 0, MPI_COMM_WORLD);
    MPI_Recv(&u[0], np, MPI_DOUBLE, myid-1, 0, MPI_COMM_WORLD, &status);
    if (myid != (numprocs - 1)) {
        MPI_Send(&u[rows_per_proc * np], np, MPI_DOUBLE, myid+1, 0,
            MPI_COMM_WORLD);
        MPI_Recv(&u[(rows_per_proc+1) * np], np, MPI_DOUBLE, myid+1, 0,
            MPI_COMM_WORLD, &status);
    }
    residual = relax_jacobi(u, uhelp, rows_per_proc+2, np);
...
MPI_Send(&u[np], np*rows_per_proc, MPI_DOUBLE, 0, myid, MPI_COMM_WORLD);
```

As for the code in solver, the outer loops have been removed; Since the work is being distributed in the main function across different MPI processes. Checking the output (using diff) with part 1 results, show that the function is working properly.

2.2 Gauss-Seidel

In the Gauss-Seidel method, no temporary matrix holds the data, so dependencies require careful handling. At each iteration, the previous row must be fully calculated before it can be used in the current row. The file `heat-mpi.c` remains unchanged for most of the part; However, additional lines are required to support the Gauss-Seidel algorithm in both master and worker processes. These adjustments enable each process to execute the function with the appropriate workload and necessary information.

```
...
// master
case 2:
    residual = relax_gauss(param.u, rows_per_proc + 2, np);
    if (numprocs > 1)
        MPI_Recv(&param.u[(rows_per_proc + 1)*np], np, MPI_DOUBLE, myid + 1,
            0, MPI_COMM_WORLD, &status);
    ...
// worker
```

```c
case 2:
    residual = relax_gauss(u, rows_per_proc + 2, np);
    MPI_Send(&u[np], np, MPI_DOUBLE, myid-1, 0, MPI_COMM_WORLD);
    if(myid != numprocs - 1)
        MPI_Recv(&u[(rows_per_proc+1) * np], np, MPI_DOUBLE, myid + 1, 0,
            MPI_COMM_WORLD, &status);
```

Dependencies in the solver are managed in the `solver-mpi.c` file. Each process, except for process 0 (`myid > 0`), must first receive boundary updates from the preceding process before starting computations. After performing its calculations, each process then passes updates to the next one, if there is a subsequent process (`myid < numprocs-1`).
An additional condition has been included in the boundary exchanges to ensure that these updates only occur when dealing with the first/last row of the matrix accordingly. This adjustment ensures that boundary exchanges happen only where necessary, optimizing communication by focusing on the relevant rows.

```c
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
for (int ii=0; ii<nbx; ii++)
    for (int jj=0; jj<nby; jj++) {
        if(ii == 0 && myid > 0) {
            MPI_Recv(&u[jj * by], by, MPI_DOUBLE, myid-1, 0, MPI_COMM_WORLD,
                &status);
        }
        for (int i=1+ii*bx; i<=min((ii+1)*bx, sizex-2); i++)
            for (int j=1+jj*by; j<=min((jj+1)*by, sizey-2); j++) {
               unew= 0.25 * (    u[ i*sizey + (j-1) ]+  // left
                  u[ i*sizey   + (j+1) ]+  // right
                  u[ (i-1)*sizey   + j     ]+  // top
                  u[ (i+1)*sizey   + j     ]); // bottom
               diff = unew - u[i*sizey+ j];
               sum += diff * diff;
               u[i*sizey+j]=unew;
            }
        if (ii == (nbx - 1) && myid < (numprocs - 1)) {
            MPI_Send(&u[((sizex-2) * sizey) + jj * by], by, MPI_DOUBLE, myid
                + 1, 0, MPI_COMM_WORLD);
        }
    }
```

2.3 Results and Analysis

The table below shows the execution time of using MPI with varying numbers of processes and problem sizes. Unfortunately, when running with two or more MPI processes, the application consistently fails to complete within the allocated time, exceeding the time limit in each case. This outcome suggests that the overhead associated with MPI communication, particularly the data copying required between iterations, negates any potential performance gains from parallelism. Consequently, the MPI implementation achieves a speedup of less than 1 across all cases, indicating that the communication costs outweigh the computational benefits, resulting in slower performance than the sequential version.

		# MPI Process		
Resolution	Algorithm	1	2	4
256	Jacobi	1.846	>Time Limit	>Time Limit
	Gauss-Seidel	4.677	>Time Limit	>Time Limit
512	Jacobi	11.166	>Time Limit	>Time Limit
	Gauss-Seidel	40.01	>Time Limit	>Time Limit

Table 2: Execution time of Jacobi and Gauss-Seidel algorithms using MPI

3 Parallelization with CUDA

3.1 Initial Kernel Implementation

This part only focuses on implementing an initial kernel- computing the matrix values in parallel and leaving the residuals to CPU. First, the `gpu_Heat` kernel is completed by using GPU thread blocks to locate the row and column of a particular element and perform computations based on neighboring elements. Each kernel invocation processes a single element, with special attention given to corner cases.

```
__global__ void gpu_Heat (float *h, float *g, int N) {
  int j = threadIdx.x + blockIdx.x * blockDim.x; // column index
  int i = blockIdx.y * blockDim.y + threadIdx.y; // row index
  if (i > 0 && i < (N-1) && j > 0 && j < (N-1)) {
    g[i * N + j] = 0.25 * (h[i * N + (j-1)] + // left
                               h[i * N + (j+1)] + // right
                               h[(i-1) * N + j] + // top
                               h[(i+1) * N + j]); //bottom
  }
}
```

This code segment handles essential memory allocations and data exchanges across CPU and GPU; Ensuring results are stored accordingly. Followed by memory deallocation. (This part is filling the *TODO* tasks left in the file `heat-CUDA.cu`.

```
// 1. Allocation on GPU for matrices u and uhelp
cudaMalloc(&dev_u, param.u, np * np * sizeof(float));
cudaMalloc(&dev_uhelp, param.uhelp, np * np * sizeof(float));
...
// 2. Copy initial values in u and uhelp from host to GPU
cudaMemcpy(dev_u, param.u, np * np * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(dev_uhelp, param.uhelp,  np * np * sizeof(float),
    cudaMemcpyHostToDevice);
...
// 3. Residual is computed on host, we need to get from GPU values computed
    in u and uhelp
cudaMemcpy(param.uhelp, dev_uhelp, np * np * sizeof(float),
    cudaMemcpyDeviceToHost);
cudaMemcpy(param.u, dev_u, np * np * sizeof(float), cudaMemcpyDeviceToHost);
...
// 4. Get result matrix from GPU
cudaMemcpy(param.u, dev_u, np * np * sizeof(float), cudaMemcpyDeviceToHost);
...
// 5. Free memory used in GPU
cudaFree(dev_u);
cudaFree(dev_uhelp);
```

Following is a result of submitting the job after these changes. GPU version execution time is almost half of the CPU version.

```
  Running heat-CUDA
Iterations :  25000
Resolution :  256
Num.  Heat sources :  2
1:  (0.10, 0.10) 1.00 2.50
2:  (0.70, 1.00) 1.00 2.50
Execution on CPU (sequential)
-----------------------------
Time on CPU in ms.= 8964.936523 (12.019 GFlop => 1340.72
MFlop/s)
Execution on GPU
----------------
Time on GPU in ms.  = 4543.772461 (12.019 GFlop => 2645.27
MFlop/s)
```

Commenting out the residual computation, which is sequentially performed on the CPU after each thread completes its assigned element, significantly reduces GPU execution time, as the communication required for this step introduces a noticeable amount of overhead.

```
  Execution on GPU
----------------
Time on GPU in ms.  = 0.553056 (12.019 GFlop => 21732878.00
MFlop/s)
Convergence to residual=0.000050:  1 iterations
```

3.2 Reduction Kernel Implementation

To compute the residual entirely on the GPU, a parallel reduction approach is applied, where each thread calculates the local residual for its assigned matrix element, storing this intermediate result in shared memory for efficient access during reduction. The reduction kernel then sums these partial residuals using an optimized tree-like structure, halving the number of active threads at each step to accelerate convergence.

Once the reduction is complete, the residuals from each block are organized as an array and copied back to the CPU for final summation. This method minimizes data transfer overhead, improving performance by retaining calculations on the GPU. To implement this, both the kernel and main function require modifications. An additional parameter is introduced to store residual values on the device, with each thread computing its assigned element and contributing to the final residual calculation, using a unique identifier to track values within its block. Partial residuals are stored in shared memory, and thread 0 in each block holds the cumulative result, facilitating efficient final residual calculations on the CPU. A copy of the files for the initial implementation (without GPU-based reduction) also exists, identified by the suffix ”noreduction” for convenience. ’

```
 __global__ void gpu_Heat (float *h, float *g, int N, float* residual) {
  extern __shared__ float sdata[]; // shared memory for partial residuals
  unsigned int tid = threadIdx.x * blockDim.x + threadIdx.y; // unique id for
      each thread within a block
    int j = threadIdx.x + blockIdx.x * blockDim.x; // column index
    int i = blockIdx.y * blockDim.y + threadIdx.y; // row index
    int index = i * N + j;
    if (i > 0 && i < (N-1) && j > 0 && j < (N-1)) {
    g[index] = 0.25 * (h[i * N + (j-1)] + // left
                       h[i * N + (j+1)] + // right
                       h[(i-1) * N + j] + // top
                       h[(i+1) * N + j]); // bottom
    // calculate of diff value
    float diff = g[index] - h[index];
    sdata[tid] = diff * diff;
    }
    __syncthreads();
    // reduction
    for (unsigned int s = blockDim.x * blockDim.y / 2; s > 32; s >> 1) {
    if (tid < s) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }
     if (tid < 32) {
        sdata[tid] += sdata[tid + 32];
        sdata[tid] += sdata[tid + 16];
        sdata[tid] += sdata[tid + 8];
        sdata[tid] += sdata[tid + 4];
        sdata[tid] += sdata[tid + 2];
        sdata[tid] += sdata[tid + 1];
    }
    if (tid == 0) { // Thread 0 will hold the residual for this block
        residual[blockIdx.x * gridDim.y + blockIdx.y] = sdata[0];
    }
}
```

The results provided for resolutions of 256 and 512 show a significant improvement in GPU performance after including the residual's computation in the kernel.

```
   Iterations :  25000
 Resolution :  256
 Num.  Heat sources :  2
 1:  (0.10, 0.10) 1.00 2.50
 2:  (0.70, 1.00) 1.00 2.50
 Execution on CPU (sequential)
 -----------------------------
 Time on CPU in ms.= 8802.777344 (12.019 GFlop => 1365.42
 MFlop/s)
 Convergence to residual=0.000050:  16673 iterations
 Execution on GPU
 ----------------
```

```
Time on GPU in ms.  = 587.640808 (12.019 GFlop => 20453.82
MFlop/s)
Convergence to residual=240307856571356758089744252928.000000:
25000
=========================================================
iterations
Iterations :  25000
Resolution :  512
Num.  Heat sources :  2
1:  (0.10, 0.10) 1.00 2.50
2:  (0.70, 1.00) 1.00 2.50
Execution on CPU (sequential)
-----------------------------
Time on CPU in ms.= 50196.839844 (72.090 GFlop => 1436.14
MFlop/s)
Convergence to residual=0.000069:  25000 iterations
Execution on GPU
----------------
Time on GPU in ms.  = 506.018555 (72.090 GFlop => 142464.34
MFlop/s)
Convergence to residual=0.000050:  8307 iterations
```

3.3 Results and Analysis

To assess the impact of thread block sizes on performance, we varied the values of threads per block among 4, 8, and 16 (the default being 8). We recorded the execution times for both CPU (sequential) and GPU (CUDA) executions. For the GPU, there are two separate rows: one for the basic kernel execution and another for the enhanced kernel with reduction, where both the residual computation and the heat equation matrix are calculated within the same kernel.

In nearly all configurations, the kernel with reduction outperformed the basic kernel significantly, reducing data movement between the CPU and GPU and performing calculations in parallel instead of sequentially. However, for a problem size of 512x512 and 4 threads per block, this configuration led to a segmentation fault error. Table 4 shows the speedup achieved for each configuration, with the CPU execution time for each specific setup used as the baseline.

In conclusion, the kernel with reduction, leveraging shared memory, avoids a substantial amount of data transfer and maximizes parallelism, achieving substantial performance gains over the CPU baseline.

			# Threads per Block		
Resolution	Device	Reduction	4	8	16
256	CPU	No	8911.676	8964.937	8653.719
	GPU	No	4565.624	4543.772	4546.52
		Yes	1509.724	587.6408	358.4315
512	CPU	No	49979.13	50196.84	49728.83
	GPU	No	25251.7	27289.16	24993.49
		Yes	Seg fault	506.0186	118.7824

Table 3: Execution time of Jacobi on CPU and GPU using CUDA

			# Threads per Block		
Resolution	Device	Reduction	4	8	16
256	CPU	No	1.000	1.000	1.000
	GPU	No	1.952	1.973	1.903
		Yes	5.903	15.256	24.143
512	CPU	No	1.000	1.000	1.000
	GPU	No	1.979	1.839	1.990
		Yes	-	99.200	418.655

Table 4: Speedup of CUDA kernels on Jacobi over different number of threads per block