Danmarks Tekniske Universitet

DTU

Advanced Parallel Computing with CUDA - Project

The Poisson Problem

Luke Labrie-Cleary – s213878

July 25, 2023

Contents

1 Introduction

1.1 Background

The field of high performance computing (HPC) has seen significant growth in the past decades with the evolution of computational power. In recent years, the transition from sequential processing to concurrent processing paradigms, involving multiple cores or multiple processors to perform computational tasks simultaneously has been of particular interest in the field. One such paradigm that has gained substantial traction is computing with graphical processing units. Traditionally, graphics processing units (GPUs) were designed for rendering graphics for gaming and 3D applications. However, with the advent of GPU computing, GPUs have found application in scientific computing, machine learning, and other data-intensive tasks, owing to their parallel processing capabilities.

GPUs, compared to Central Processing Units (CPUs), offer an advantage in parallel processing due to their higher number of cores, and thus higher throughput. This feature makes GPUs exceptionally suitable for tasks that can be broken down into smaller, independent units of work, like the finite difference methods typically used for solving differential equations. CPUs, while not as adept at parallel processing as GPUs – i.e. lower throughput – possess a higher clock speed – i.e. low latency – and are thus more suited for sequential tasks. However, modern CPUs also offer parallel processing capabilities to a certain extent, thanks to technologies such as multithreading and multiprocessing.

The project herein will investigate the performance of different tools for parallel computing on GPUs, using parallel CPU programs a baseline for comparison. The tools examined include the accelerator support now available with OpenMP, NVIDIA's CUDA API and collective communications library (NCCL), OpenMPI, as well as combinations of each.

1.2 Method

1.2.1 Tools

OpenMP (Open Multi-Processing) is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran. It offers a simple and flexible interface for developing parallel applications. OpenMP is mostly concerned with parallel computing on CPUs, but with the advent of OpenMP 4.0 and later versions, the support for accelerator devices such as GPUs has been introduced.

CUDA (Compute Unified Device Architecture) is a parallel computing platform and API model created by NVIDIA. It uses a variant of C++ programming language, and it enables the use of NVIDIA GPUs for general purpose processing. CUDA allows for fine-grained control over the GPU hardware thereby enabling the design of highly optimized programs making full use of the massive parallel capacity of a GPU.

A common concern of parallel computing is that the communication overhead between different computational units often forms a bottleneck for performance. Efficiently managing this inter-unit communication is crucial for achieving optimal performance, and represents the central investigation of the project herein. The CUDA toolkit offers several tools to

handle this, such as peer-to-peer memory access between GPUs, as well as NCCL (NVIDIA Collective Communications Library), which provides primitives for multi-GPU and multi-node collective communication.

MPI (Message Passing Interface) is a standard for distributed memory parallel programming, widely used in large-scale simulations run across networked machines. The project herein will use MPI to facilitate communication across nodes, enabling us to access more GPUs.

NVIDIA profilers will assist with performance analysis. Profiling of programs in the project herein uses NVIDIA Nsight Compute (NCU) and NVIDIA Nsight Systems (NSYS).

1.2.2 The Poisson Problem

As a basis for the implementation and comparison of these tools, we start with the Poisson problem in three dimensions, as posed in Assignments 2 and 3 of course 02614 High-Performance Computing. The Poisson "problem" consists of solving the Poisson equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = -f(x, y, z), \quad (x, y, x) \in \Omega \tag{1}$$

on a given domain Ω, where $u(x, y, z)$ is the solution and $f(x, y, z)$ is a (constant) source term. The Poisson equation is an elliptic partial differential equation with many applications in physics. Its solution gives the steady-state potential field on the domain Ω for a given distribution (boundary conditions) of mass, charge, heat, etc. The goal of the different solvers implemented in this project will be to solve the Poisson equation for the steady-state heat distribution on a cubical domain

$$\Omega = \{(x, y, z) : |x| \leq 1, |y| \leq 1, |z| \leq 1\} \tag{2}$$

with boundary conditions defined as

$$\begin{aligned} u(x, 1, z) = 20, u(x, -1, z) = 0, -1 \leq x, z \leq 1 \\ u(1, y, z) = u(-1, y, z) = 20, -1 \leq y, z \leq 1 \\ u(x, y, -1) = u(x, y, 1) = 20, -1 \leq x, y \leq 1. \end{aligned}$$

and the source term $f(x, y, z)$ defined as

$$f(x, y, z) = \begin{cases} 200, & -1 \leq x \leq -\frac{3}{8}, -1 \leq y \leq -\frac{1}{2}, -\frac{2}{3} \leq z \leq 0 \\ 0, & \text{elsewhere} \end{cases} \tag{3}$$

The domain is plotted graphically in Figure 1

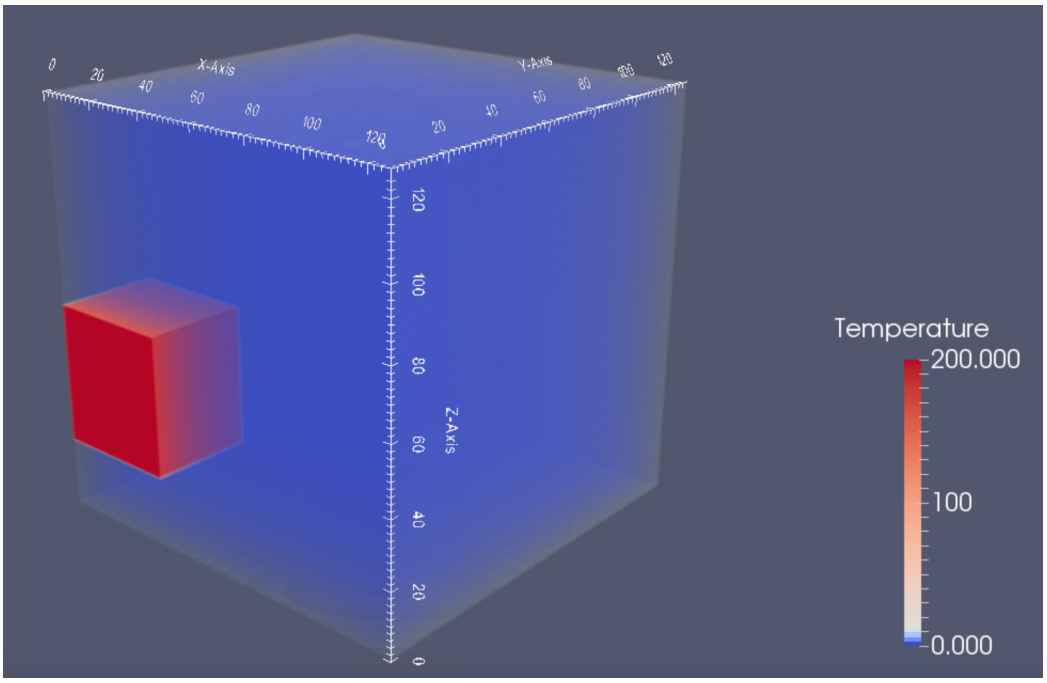


Figure 1: Poisson Problem Domain

Note, what is less visible than the radiator in Figure 1 is that the XZ plane at $Y = 0$ is held at 0°C (excluding the radiator), while all other surfaces of the domain are held at 20°C, as described by the equations above.

To solve the given problem numerically, the domain Ω is dicretized on an NxNxN grid of double-precision floating point numbers. The solution can than be computed iteratively by means of the seven-point stencil formula

$$u_{i,j,k} \leftarrow \frac{1}{6}\left(u_{i-1,j,k} + u_{i+1,j,k} + u_{i,j-1,k} + u_{i,j+1,k} + u_{i,j,k-1} + u_{i,j,k+1} + \Delta^2 f_{i,j,k}\right) \tag{4}$$

where Δ is the grid spacing, $2/N$, and i, j, k are indicies on the domain with $0 \leq i, j, k \leq N - 1$, i.e.

$$u_{i,j,k} = u(i * \Delta, j * \Delta, k * \Delta) \tag{5}$$

To clarify, one iteration consists of updating each point on the domain using the seven-point stencil formula above. There are several ways to do this in practice, but for simplicity, the solvers herein will use exclusively the Jacobi iterative method, in which each update to $u_{i,j,k}$ is performed simultaneously on each iteration. The simultaneous updates require three matrices on the domain; one for the previous iteration, one for the updated domain, and one containing the domain of the source term. There are of course more sophisticated, memory-friendly iterative methods, such as the Gauss-Seidel iterations implemented in 02614, but this is outside of the scope of the project herein. The solution for 1000 iterations on the domain in Figure 1 ($N = 128$) is shown in Figure 2

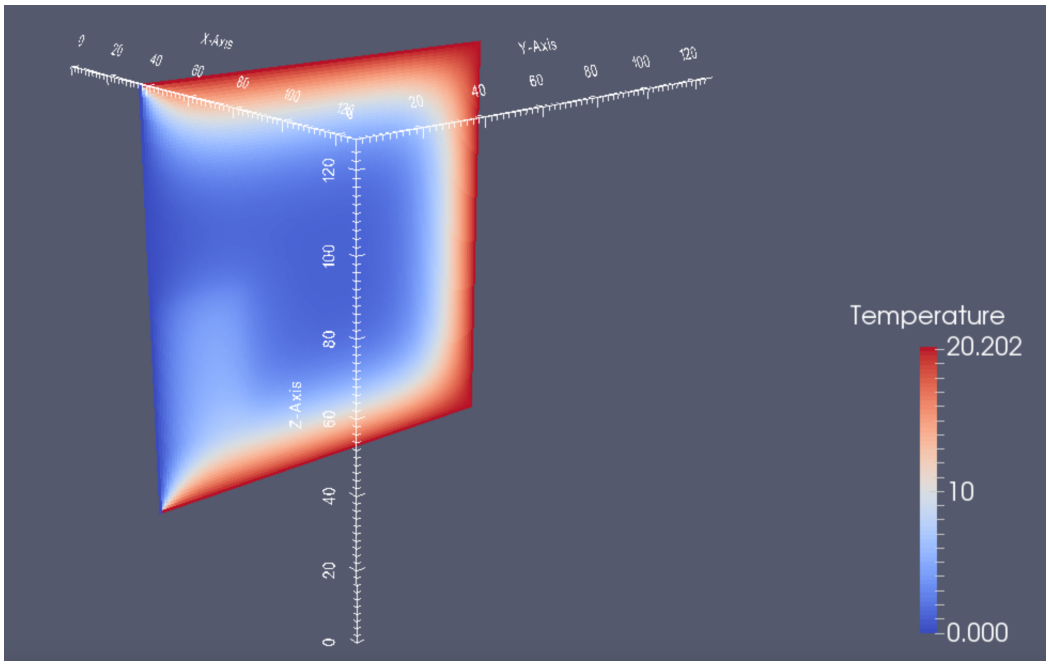


Figure 2: Poisson Problem - Sample Solution for $N = 128$, 1000 Iterations, on the YZ Plane at $X = 36$

In 02614, we compared different iterative methods for the parallel CPU solver using OpenMP, and later we implemented OpenMP's accelerator support to add single and dual-GPU implementations to the solver. The OpenMP GPU implementations will serve as the baseline solver from which we can compare more sophisticated GPU implementations using the tools discussed above. Note, for simplicity, all implementations of the project herein will use only 1000 Jacobi iterations.

To begin, both the single GPU and dual GPU OpenMP poisson solver are ported to CUDA. Then the solver is further expanded to use four GPUs across two nodes, using MPI to initialize processes on both nodes and NCCL directives to communicate on the domain boundary between nodes.

1.2.3 Performance Measurement

The fundamental performance metric will be wall-time measurements for different problem sizes. This can be used to examine how well the problem scales on different solvers, and can also be used to calculate the bandwidth. Both of which can be used to evaluate the relative efficacy of each method. More fine-grained time measurements may be employed to analyze performance. Namely, wall-time measurements including and excluding data-transfer will be employed to gain further insight into performance. For the purposes of the project herein, "data-transfer" time includes memory allocation and freeing, data transfer to and from devices, as well as time to initialize processes like CUDA, NCCL and MPI communicators. Data-transfer time here therefore serves as a broad, catch-all metric to enable more accurate estimates of true realized bandwidth.

The bandwidth B is assumed to be reasonably approximated by the relation

$$B = \frac{3 * N^3 * 8 * K}{t} = \frac{24KN^3}{t} \tag{6}$$

where B is the bandwidth in bytes, K is the number of iterations, N is the grid dimension, and t is the runtime in seconds. The factor of three comes from the three grids required for the Jacobi iterative method; one to store the values the values of the previous iteration, one to populate with updated values, and one to store the source term. The factor of eight is the number of bytes in a double. Note, runtime here is deliberately left ambiguous. For smaller problem sizes, a wall-time estimate can suffice. For larger problems, we might need to exclude the data-transfer times in order to properly estimate realized bandwidth for a given problem. Additionally, the NVIDIA profilers like NCU and NSYS will assist in performance measurement and diagnostics.

In 02614, we were also concerned with convergence, i.e. time and/or number of iterations before some stopping criterion was reached, e.g. the difference in the Frobenius norm of the matrices representing the domain between one iteration and the next. Here however, we are using the same iterative technique, Jacobi iterations, and therefore the result of each method, given the same number of iterations as input, is equal and deterministic. That is, regardless of the method, we should in theory have the exact same number of iterations before reaching some specified tolerance, and we thus decide *a priori* not to use convergence as a basis for comparison.

1.2.4 Driver

The driver executable, `poisson_solver`, can be called as follows

```
./poisson_solver N K T_0 output_type [file_suffix] [threads]
```

in a directory containing the executable, and where `N` is the dimension of the cubical grid, `K` is the number of iterations (defined above), `T_0` is the starting temperature for inner grid points, `output_type` is the result format, `file_suffix` is the file suffix of the grid data (if relevant), and `threads` is the number of threads to use if calling a CPU implementation. The arguments are described in more detail in the `README` file of the project directory.

1.2.5 Hardware

CPU implementations are run on an Intel(R) Xeon(R) Gold 6226 CPU @ 2.70GHz. Runtime and bandwidth tests for GPU implementations are run using NVIDIA Tesla V100s, with 32 GB of device memory, a nameplate memory bandwidth of 900 GB/s and connected by a 32 GB/s PCIe bus. Some profiling of specific implementations and associated kernels is done on NVIDIA Tesla A100s, with 40 GB of device memory, a memory bandwidth rating of 2 TB/s, and connected by an NVLink communicator rated at 25 GB/s. The hardware used will be provided with all results shown. For more detailed hardware information, see Appendix.

2 OpenMP

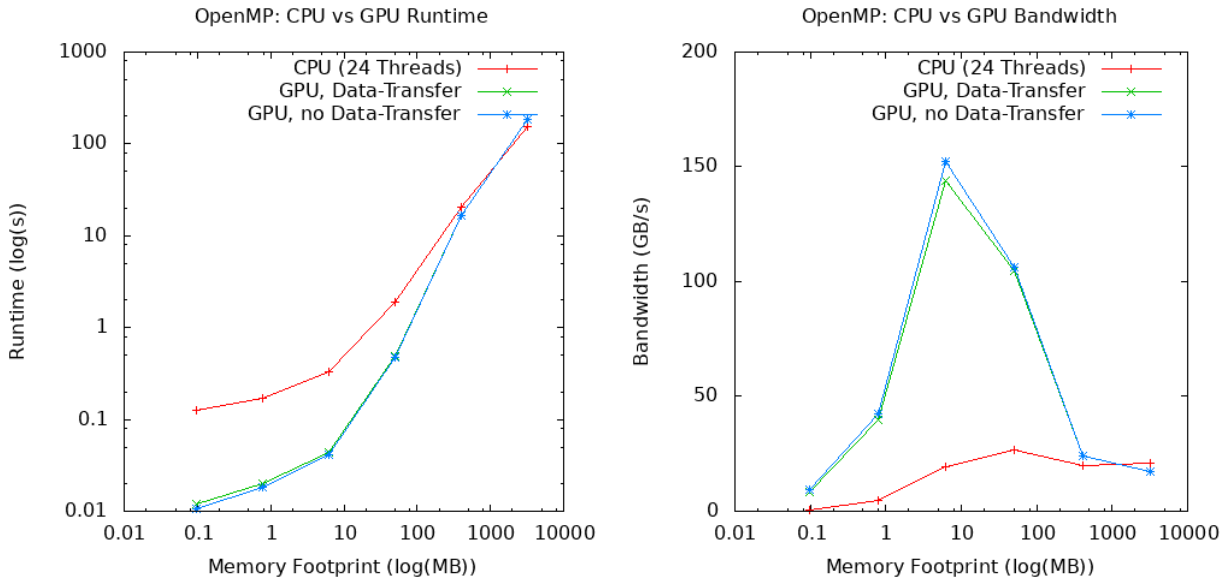
OpenMP was created primarily for shared-memory parallel programming on CPUs. As mentioned, OpenMP now includes accelerator support, i.e. functionality to leverage GPU devices for massive parallelism. With OpenMP, instructions for GPU implementations are given similarly to the parallel CPU implementations; with compiler directives. The `target` directive is used to offload computation to an accelerator device (like a GPU). For instance, the `teams` directive then creates a league of thread teams where the master thread of each team executes the structured block. The `distribute` directive divides the iterations of the enclosed loop among the thread teams. The `parallel` directive followed by a for-loop can be used in the same way they are for CPU implementations, and divide the iterations of the enclosed loop among the threads in a team.

2.1 Single GPU

In the 02614 High-Performance Computing course, we implemented a single-GPU solver using the Jacobi iterative method with OpenMP. Full implementation is in the `jacobi_offload_dynamic` function of `jacobi_offload.c` (see appendix). To demonstrate the use of the aforementioned directives, the loop structure for one iteration is shown below

```
#pragma omp target teams loop is_device_ptr(U_d,V_d,F_d)
                for (int j = 1; j<N-1; j++) {
                    #pragma omp loop bind(parallel)
                    for (int i = 1; i<N-1; i++) {
                        for (int k = 1; k<N-1; k++){
                        V_d[i][j][k] = dev*(U_d[i-1][j][k]+U_d[i+1][j][k]+
   U_d[i][j-1][k]+U_d[i][j+1][k]+U_d[i][j][k-1]+U_d[i][j][k+1]+del2*F_d[i][
   j][k]);
                    }
                }
            }
```

Figure 3 compares the performance of the fastest CPU parallel implementation from 02614 (using 24 threads), and the implementation shown above.



(a) OpenMP CPU vs GPU: Runtime **(b)** OpenMP CPU vs GPU: Memory Bandwidth

Figure 3: OpenMP CPU vs GPU Performance with Intel(R) Xeon(R) Gold 6226 CPU @ 2.70GHz and Tesla V100-PCIE-32GB GPU

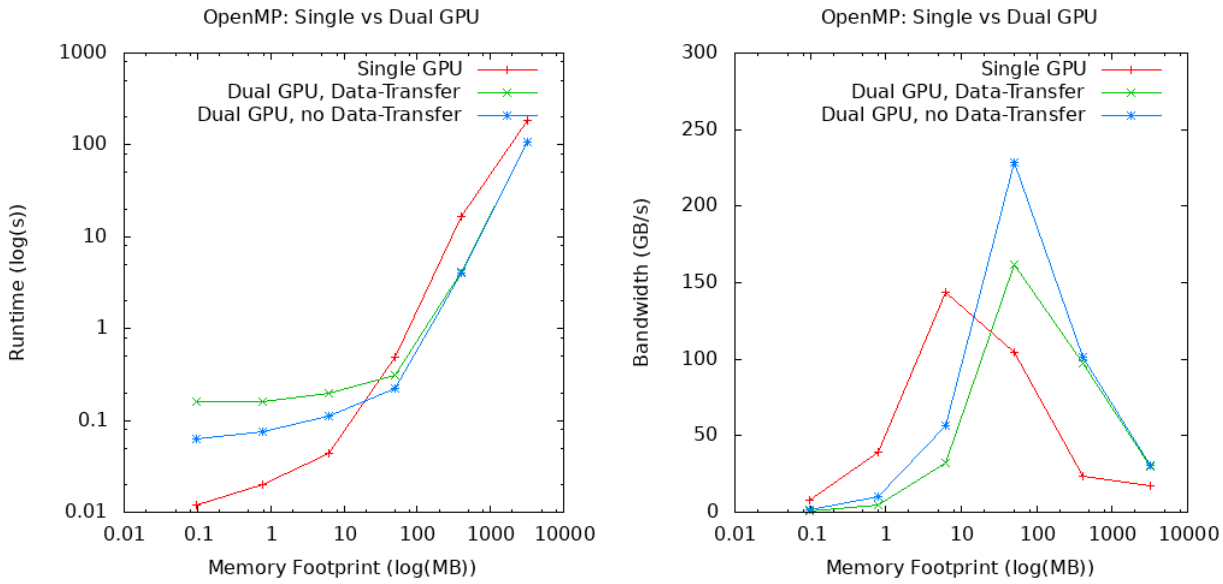
GPU implementation for problem sizes less than 1 GB, corresponding to approximately $N < 512$, outperforms the CPU implementation by approximately an order of magnitude. Performance converges with problem sizes on the order of 1 GB. This likely indicates a memory bottleneck in the OpenMP GPU implementation, as it should in theory continue to outperform the CPU implementation. Maximum realized bandwidth on the GPU implementation was $\sim$ 143 GB/s is well below the nameplate maximum of 900 GB/s. As this serves only as a starting point for porting to more sophisticated implementations, we will not further diagnose the bottleneck.

2.2 Dual GPU

Also implemented in 02614 was a solver using two GPUs. Each solves half of the domain independently, except on the boundary, where each GPU can access data from the other via the PCI Express Bus, by calling the CUDA API function `cudaDeviceEnablePeerAccess()` as follows

```
// Enable peer-to-peer access and offload
cudaSetDevice(0);
cudaDeviceEnablePeerAccess(1, 0); // (dev 1, future flag)
cudaSetDevice(1);
cudaDeviceEnablePeerAccess(0, 0); // (dev 0, future flag)
```

Figure 4 compares the performance of the dual GPU OpenMP implementation against the single GPU OpenMP implementation



(a) Runtime **(b)** Memory Bandwidth

Figure 4: OpenMP Single vs Dual GPU Implementation Tesla V100-PCIE-32GB GPU

Improvement in performance is only realized for larger resolutions; at memory footprints greater than $\sim$ 50 MB, which corresponds to a grid with $N \approx 128$, indicating that the additional overhead required in the initialization and memory management of using two GPUs, and communicating via `cudaDeviceEnablePeerAccess()` will dominate until the problem is sufficiently large to leverage the extra throughput provided by an additional GPU. Maximum realized bandwidth, $\sim$ 161 GB/s, is a modest, if not negligible improvement against the bandwidth of the single GPU OpenMP implementation, and still well below the 900 GB/s rating.

Notable here however, is the significant increase in data-transfer time relative to runtime. In the single-GPU OpenMP implementation, maximum realized bandwidth using full wall-time was 143 GB/s, and 152 GB/s excluding data-transfer. For the dual-GPU implementation, this was 161 GB/s vs 228 GB/s respectively. The computational overhead incurred by leveraging more devices can thus, in this case, almost entirely negate the added performance benefit. Close attention must therefore be paid to the marginal overhead of added parallelism.

3 CUDA

CUDA is the NVIDIA API made specifically for NVIDIA GPUs. Specific to CUDA in accelerated (GPU) computing is the use of kernels, which are units of code executed on

the GPU. They are essentially C/C++ functions that are defined to run in parallel on the device. Each instance of a kernel executes on a separate GPU thread. In CUDA, the thread is the basic unit of execution. Each thread runs an instance of a CUDA kernel and has its own local memory. Threads are grouped into warps, which are groups of threads (typically 32 threads for most NVIDIA GPUs) that are scheduled together for execution. These warps are further organized into blocks, which are groups of threads that can cooperate with each other by sharing data through shared memory and synchronizing their execution. All threads within a block run on the same streaming multiprocessor. Blocks are structured as a grid, which is composed of multiple thread blocks that can be executed independently and can be one, two or three-dimensional. The blocks within a grid can be scheduled in any order across all available streaming multiprocessors. Figure 5 diagrams the described hierarchy.

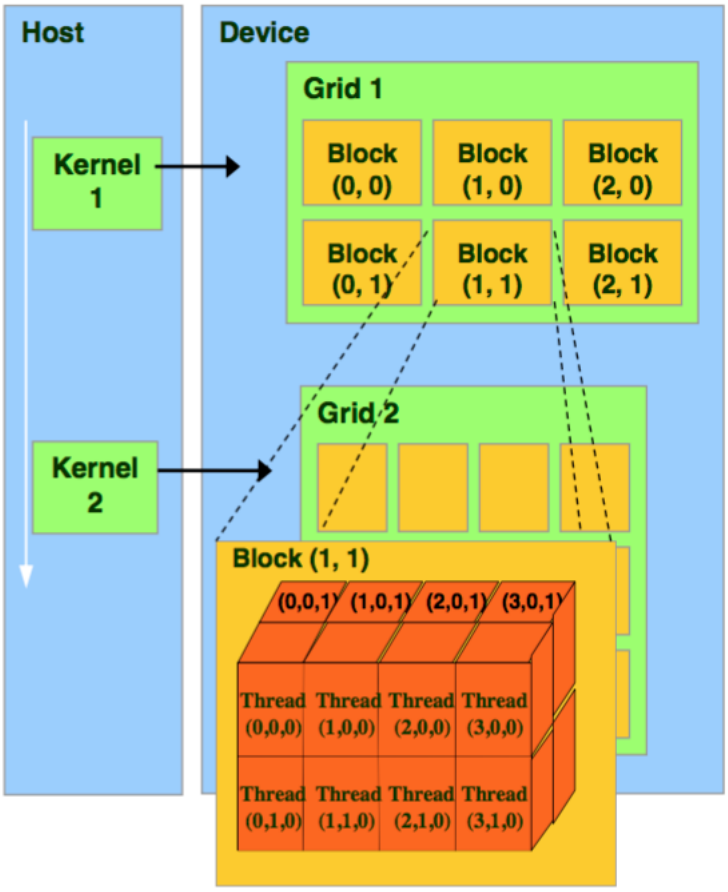


Figure 5: CUDA Architecture, from https://nyu-cds.github.io/python-gpu/02-cuda [?]

3.1 Single GPU

3.1.1 Initial Implementation

The single-GPU implementation from 02614 is ported to CUDA in the function `jacobi_cuda_single()` (see Appendix), using methodology analagous to that of the OpenMP solver; with three

arrays allocated to the device to store updates for each iteration. The function handles memory allocation on the device, defines the parameters of the thread hierarchy described above, and calls the kernel for each iteration. The kernel then handles the updates on the domain. The relevant parameter definitions and associated kernel call are shown below

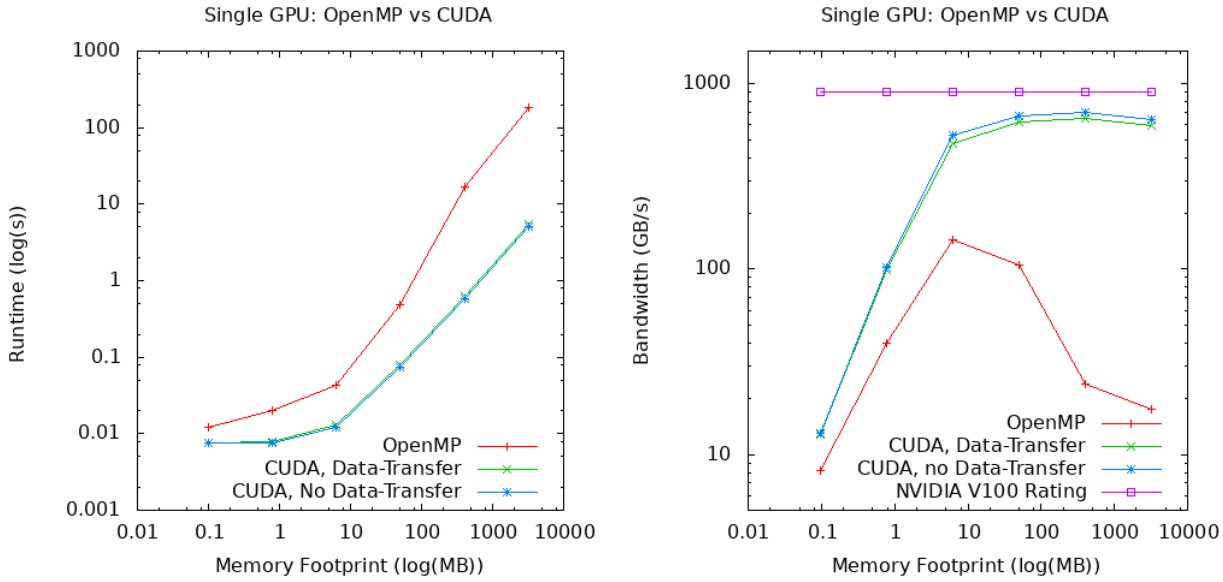
```
// define domain for kernel
dim3 blocks(8, 8, 16); // a 8x8x8 block of threads
dim3 grid((N) / blocks.x, (N) / blocks.y, (N) / blocks.z); // enough blocks
   to cover the whole 3D domain

int m = 0;
// call kernel
// kernel<<<NUM_BLOCKS, THREADS_PER_BLOCK>>>(...);
while (m<K) {
    poisson_single<<<grid, blocks>>>(U_d,V_d,F_d,div,del2,N);
    cudaDeviceSynchronize();
    m += 1;
    double *old = U_d;
    U_d = V_d;
    V_d = old;
}
```

Here, thread blocks are 8 x 8 x 16 = 1024 threads. This was found to be optimal for the signle GPU implementation. To ensure maximum occupancy and utilization, the grid is defined such that the blocks exactly cover the whole domain. The `poisson_single` kernel is show below

```
__global__
void poisson_single(double *u_d, double *v_d, double *f_d, double div,
   double del2, int n) {

    /*
    Single GPU kernel
    */

    // get global indices
    int k = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int i = blockIdx.z * blockDim.z + threadIdx.z;
    bool boundary = (i == (n-1) || j == (n-1) || k == (n-1) || i == 0 || j
   == 0 || k == 0);
    if (!boundary) {
        int global = IDX3D(i,j,k,n);
```

```
        // naming convention from perspective of xz plane
        double down = i > 1 ? u_d[IDX3D(i-1,j,k,n)] : 20.0;
        double up = i < n-2 ? u_d[IDX3D(i+1,j,k,n)] : 20.0;
        double back = j > 1 ? u_d[IDX3D(i,j-1,k,n)] : 0.0;
        double forward = j < n-2 ? u_d[IDX3D(i,j+1,k,n)] : 20.0;
        double right = k < n-2 ? u_d[IDX3D(i,j,k+1,n)] : 20.0;
        double left = k > 1 ? u_d[IDX3D(i,j,k-1,n)] : 20.0;

        // make update
        v_d[global] = div*(down+up+left+right+forward+back+del2*f_d[global])
    ;
     }
}
```

The CUDA objects `blockIdx`, `blockDim` and `threadIdx` allows each thread to place itself within the domain. Figure 6 shows the performance of the single GPU CUDA implementation along with that of OpenMP



(a) Runtime **(b)** Memory Bandwidth

Figure 6: CUDA vs OpenMP: Single GPU Implementation Tesla V100-PCIE-32GB GPU

Here we see dramatic improvement against the OpenMP implementation, with improved runtime at all problem sizes, and a maximum realized bandwidth of $\sim$ 650 GB/s, $\sim$ 72% of the NVIDIA V100 peak rating of 900 GB/s. Excluding data-transfer time, peak realized bandwidth was $\sim$ 700 GB/s $\sim$ 78% of the peak rating. Additionally, we do not see a drop-off at the larger problem sizes considered, i.e. the computation becomes purely compute-bound

without encountering a detriment to performance of the kind seen in the OpenMP implementation. The simplistic implementation of GPU parallelism afforded by OpenMP's compiler directives, designed with CPU parallelism in mind, evidently has significant performance drawbacks, while the fine-grained control afforded by the CUDA API allows us to achieve actual bandwidth closer to nameplate ratings.

3.1.2 Improved Implementation

Figure 7 charts the memory workload for the initial implementation of the single-GPU CUDA solver.

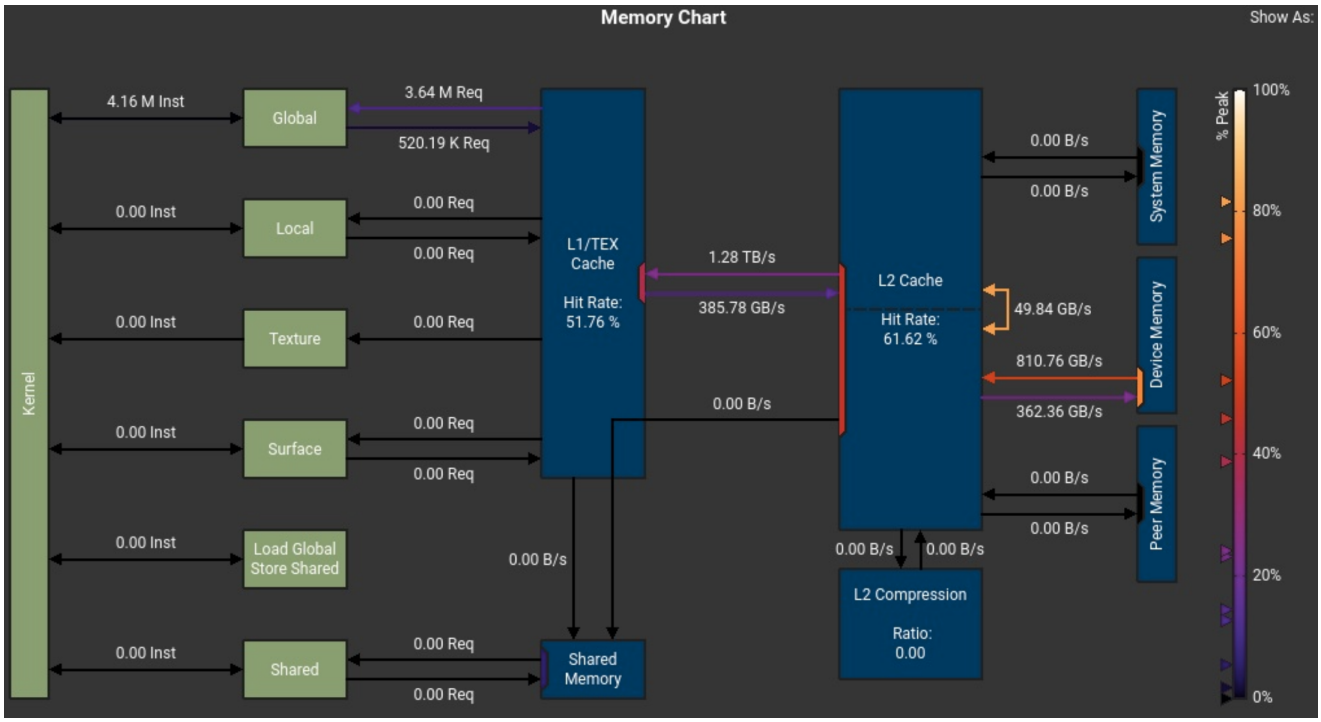


Figure 7: Memory Workload Chart: Single GPU, Initial Implementation, Tesla A100, N=256

Immediately apparent is the potential for improvement in the L1 and L2 cache hit rates. An intuitive way to improve the hit rates is to reduce the read/write calls to global memory locations on different arrays, which are highly unlikely to lie in the same cache line. Below is the line within the `poisson_single` kernel which performs the update to the point on the domain:

```
// make update
v_d[global] = div*(down+up+left+right+forward+back+del2*f_d[global]);
```

The values for the neighboring points – `up`, `down`, `left`, `right`, etc. – are all read from the device array `u_d`. Additionally, the source array `f_d`, of the same size as `u_d`, is read using the global index for the single-pointer indexing used on the device arrays. The same

index is used to write to `v_d`, which stores the updated values. There are thus seven reads from `u_d`, one read from `f_d`, and a write to `f_d`. The size of the L1 cache on an NVIDIA A100 is around 20 MB. These global read and write operations to all three arrays is thus likely to increase the miss rate significantly, impacting overall performance.

Examination of the source function, defined in eq. (3) and rewritten below

$$f(x,y,z) = \begin{cases} 200, & -1 \leq x \leq -\frac{3}{8}, -1 \leq y \leq -\frac{1}{2}, -\frac{2}{3} \leq z \leq 0 \\ 0, & \text{elsewhere} \end{cases} \tag{7}$$

reveals that it is only relevant in the region for which the value of $f(x,y,z) = 200$. Otherwise its contribution in the seven-point stencil formula, eq. (4), is zero. A simple way to reduce the global read operations in the kernel is to elimate reads to `f_d`, because at the points where it contributes to the result, its value is always the same. We can thus rewrite the update as

```
// check if in region of source
bool rad = (-1.0 <= x) && (x <= -3.0/8.0) && (-1.0 <= y) && (y <= -1.0/2.0)
   && (-2.0/3.0 <= z) && (z <= 0.0);

// make update
if (rad) v_d[global] = div*(down+up+left+right+forward+back+del2*200.00);
else v_d[global] = div*(down+up+left+right+forward+back);
```

which introduces a minor branching, but eliminates all read operations to `f_d`.

Reads to `u_d` compose the majority of the read and write operations of the kernel. Therefore, if we can make these memory accesses more coalesced, this should also improve performance. Figure 8 diagrams how the domain is allocated on the device

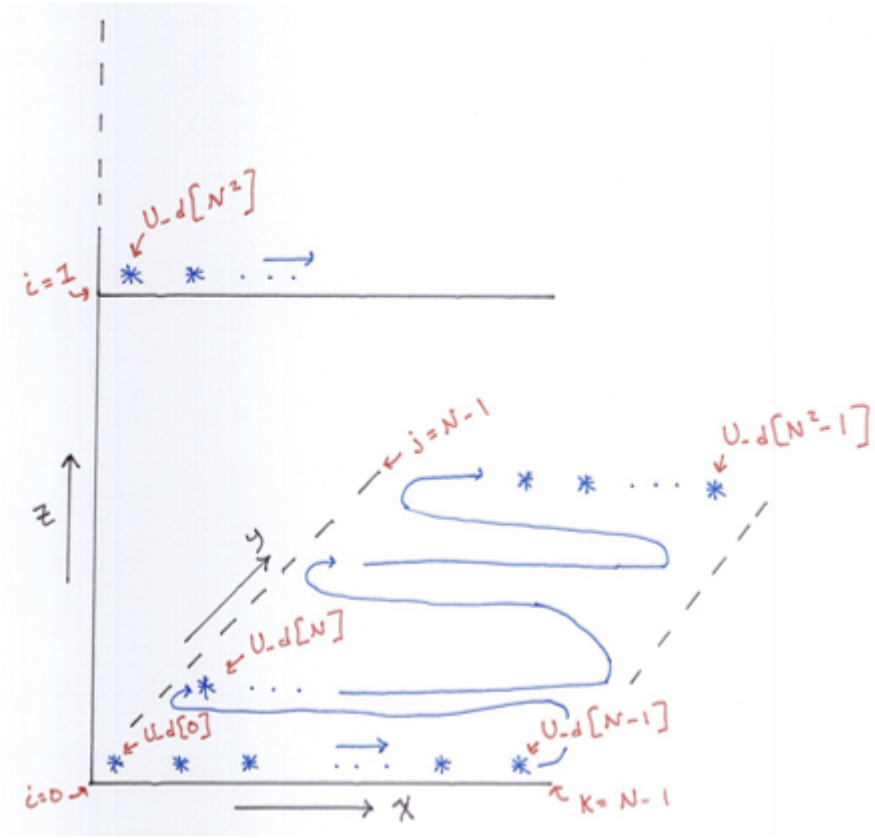


Figure 8: Device Memory Allocation Diagram

As indicated above, the domain is stored contiguously with single-pointers, traversing the x, y, then z axes as it reaches the domain boundaries. Access calls within the kernel can therefore be coalesced by prioritizing accesses in the `right`, `forward` and `up` directions over the `back`, `left` and `down` directions, as these latter calls are more likely to result in a cache miss within the row-major memory paradigm of C++. Original updates were called as follows

```
// naming convention from perspective of xz plane
double down = i > 1 ? u_d[IDX3D(i-1,j,k,n)] : 20.0;
double up = i < n-2 ? u_d[IDX3D(i+1,j,k,n)] : 20.0;
double back = j > 1 ? u_d[IDX3D(i,j-1,k,n)] : 0.0;
double forward = j < n-2 ? u_d[IDX3D(i,j+1,k,n)] : 20.0;
double right = k < n-2 ? u_d[IDX3D(i,j,k+1,n)] : 20.0;
double left = k > 1 ? u_d[IDX3D(i,j,k-1,n)] : 20.0;
```

which is updated to

```
// naming convention from perspective of xz plane
double right = k < n-2 ? u_d[IDX3D(i,j,k+1,n)] : 20.0;
double forward = j < n-2 ? u_d[IDX3D(i,j+1,k,n)] : 20.0;
```

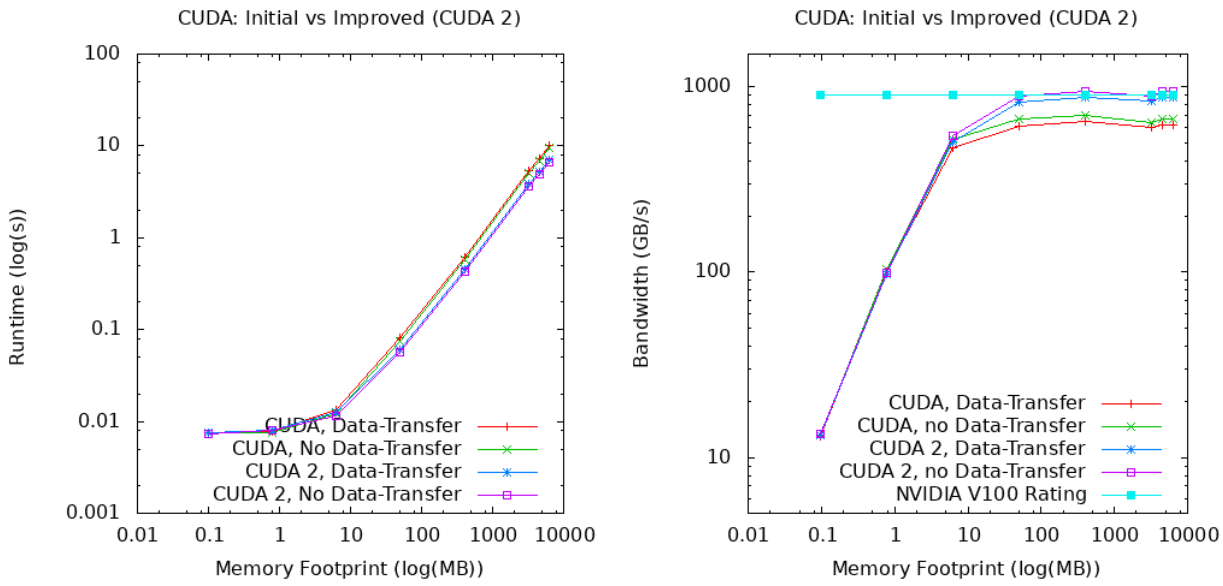
```
double up = i < n-2 ? u_d[IDX3D(i+1,j,k,n)] : 20.0;
double back = j > 1 ? u_d[IDX3D(i,j-1,k,n)] : 0.0;
double left = k > 1 ? u_d[IDX3D(i,j,k-1,n)] : 20.0;
double down = i > 1 ? u_d[IDX3D(i-1,j,k,n)] : 20.0;
```

to coalesce the memory access to `u_d`. Implementation on an NVIDIA A100 shows the following results

Solver	Bandwidth,N=512	L1 Cache Hit Rate	L2 Cache Hit Rate
Initial	1177 GB/s	52 %	62 %
Remove Source Access	1308 GB/s	58.03 %	71 %
Coalesced	1344 GB/s	58.16 %	72 %

Table 1: Performance Improvements on Single-GPU CUDA Solver, NVIDIA A100 (2 TB/s)

Each improvement cumulatively provides significant performance benefit. The minor changes in cache hit rates of the latter two, relative to the performance benefit, indicate the sensitivity of performance to the cache hit rates. The performance tests from the initial implementation are run again for the improved implementation on the NVIDIA Tesla V100s. Results are shown in Figure 9



(a) Runtime **(b)** Memory Bandwidth

Figure 9: CUDA 1 vs CUDA 2: Single GPU Implementation Tesla V100-PCIE-32GB GPU

For the improved implementation, maximum realized bandwidth including data-transfer time was 880 GB/s, and 947 GB/s excluding data-transfer time; 98 % and 105 % respectively

of the NVIDIA V100 rating of 900 GB/s.The latter exceeding the rating is presumed to be the result of efficient memory access in the caches allowing extra computations with little to no runtime cost.

3.2 Dual GPU

The single-GPU implementation is expanded to utilize two GPUs in the function `jacobi_cuda_split()` (see Appendix). Memory access on the domain boundary is handled with `cudaDeviceEnablePeerAccess()`. Relevant parameter definitions and kernel call are shown below

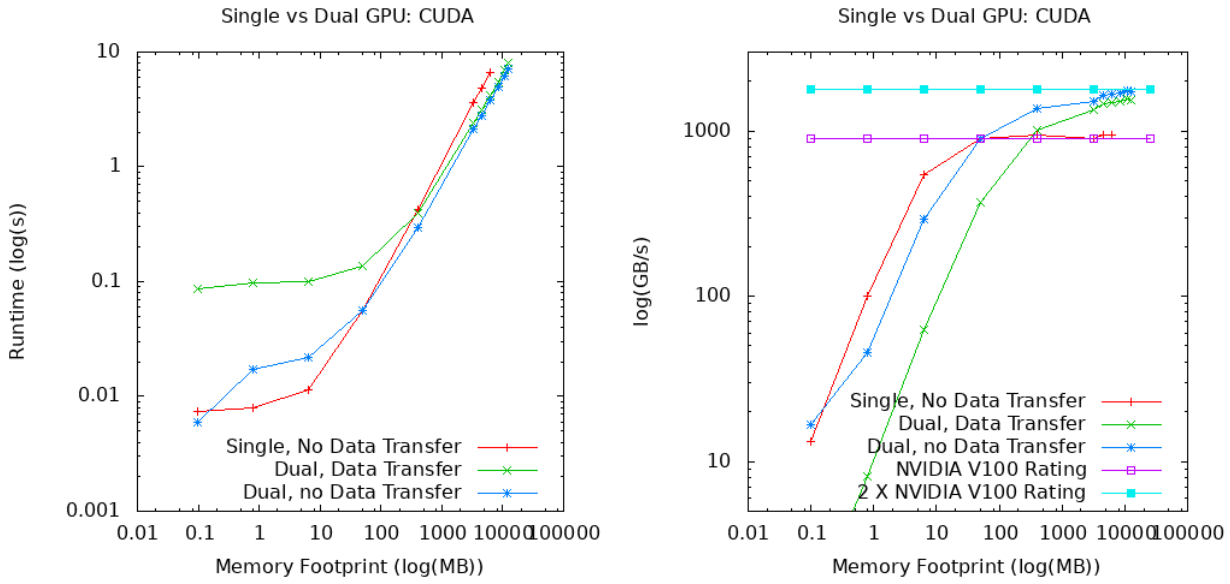
```
// define domain for GPU
dim3 blocks(8, 8, 16); // a 8x8x8 block of threads
dim3 grid((N) / blocks.x, (N) / blocks.y, (N/2) / blocks.z); // enough
   blocks to cover the whole 3D domain
int m = 0;

// kernel<<<NUM_BLOCKS, THREADS_PER_BLOCK>>>(...);
*kt = 0.0;
double sk;
while (m<K) {
    // call kernel
    sk = omp_get_wtime();
    for (int i=0; i<2; i++) {
        cudaSetDevice(i);
        poisson_split<<<grid, blocks>>>(U_d[i],V_d[i],U_d[1-i],div,del2,N,i)
   ;
    }
    // synchronize
    for (int i=0; i<2; i++) {
        cudaSetDevice(i);
        CUDACHECK(cudaDeviceSynchronize());
    }

    // update total kernel runtime
    *kt += omp_get_wtime()-sk;

    // update (pointer-swap)
    for (int i = 0; i < 2; i++) {
        double *old = U_d[i];
        U_d[i] = V_d[i];
        V_d[i] = old;
    }
    m += 1;
    }
```

The dimensions of the `blocks` object, 8 x 8 x 16, were found to be optimal for this implementation. Note, the kernel now needs pointers to the memory on both devices in order to update elements on the boundary. The `poisson_split` kernel is shown below

```
__global__
void poisson_split(double *u_d1, // matricies for 1st GPU
                   double *v_d,
                   double *u_d2, // matricies for 2nd GPU
                   double div,   // coefficients for stencil formula
                   double del2,
                   int n,        // dimensions of XY region
                   int dev      // device number
                   ) {

    /*
    Double GPU kernel
    */

    // get global indices
    int k = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int i = blockIdx.z * blockDim.z + threadIdx.z;

    // variables
    int half = n/2;
    int global = IDX3D(i,j,k,n);
    double x, y, z;
    bool rad, boundary, i_boundary;

    // location
    x = -1.0 + (2.0/(double)n)*(double)k;
    y = -1.0 + (2.0/(double)n)*(double)j;
    z = -1.0 + (2.0/(double)n)*(double)i + 1.0*dev;
    rad = (-1.0 <= x) && (x <= -3.0/8.0) && (-1.0 <= y) && (y <= -1.0/2.0)
   && (-2.0/3.0 <= z) && (z <= 0.0);
    i_boundary = ((dev == 0) && (i == 0)) || ((dev == 1) && (i == (half-1)))
   ;
    boundary = (i_boundary || j == (n-1) || k == (n-1) || j == 0 || k == 0);

    // naming convention from perspective of xz plane
    if (!boundary) {
        double down, up, back, forward, left, right;
```

```
        if (dev == 0) {
            right = k < n-2 ? u_d1[IDX3D(i,j,k+1,n)] : 20.0;
            forward = j < n-2 ? u_d1[IDX3D(i,j+1,k,n)] : 20.0;
            up = i < (half-1) ? u_d1[IDX3D(i+1,j,k,n)] : u_d2[IDX3D(0,j,k,n)
   ];
            back = j > 1 ? u_d1[IDX3D(i,j-1,k,n)] : 0.0;
            left = k > 1 ? u_d1[IDX3D(i,j,k-1,n)] : 20.0;
            down = i > 1 ? u_d1[IDX3D(i-1,j,k,n)] : 20.0;
        }
        else if (dev == 1) {
            right = k < n-2 ? u_d1[IDX3D(i,j,k+1,n)] : 20.0;
            forward = j < n-2 ? u_d1[IDX3D(i,j+1,k,n)] : 20.0;
            up = i < (half-2) ? u_d1[IDX3D(i+1,j,k,n)] : 20.0;
            back = j > 1 ? u_d1[IDX3D(i,j-1,k,n)] : 0.0;
            left = k > 1 ? u_d1[IDX3D(i,j,k-1,n)] : 20.0;
            down = i > 0 ? u_d1[IDX3D(i-1,j,k,n)] : u_d2[IDX3D(half-1,j,k,n)
   ];
        }
        // make update
        if (rad) v_d[global] = div*(down+up+left+right+forward+back+del2
   *200.00);
        else v_d[global] = div*(down+up+left+right+forward+back);
    }
}
```

Now that the problem is being split between two devices, the kernel requires more logic in order to make updates to the solution properly. First, it checks that the thread is associated with an interior point in the domain, and then makes the appropriate updates depending on which half of the domain, and thus on which device, the solution is being updated. Figure 10 shows the performance of the single and dual GPU implementations in CUDA



(a) Runtime **(b)** Memory Bandwidth

Figure 10: Single vs Dual GPU Solver: CUDA, Tesla V100-PCIE-32GB GPUs

Performance of the dual GPU implementation only exceeds that of the single GPU implementation for larger problem sizes; around when $N \geq 512$ corresponding to a memory footprint of ~ 3.2 GB. This behavior can be expected due to the additional system overhead, memory management and logic required to manage the algorithm between two GPUs, which diminishes the benefit of the increased throughput afforded by the extra GPU.

Note, memory limitations on the GPUs restricted previous tests to problem sizes of $N <\sim 650$ elements, corresponding to a memory footprint of ~ 6 GB. With two GPUs, and thus double the memory, the solver could manage problem sizes of $N <\sim 800$, corresponding to a memory footprint of ~ 10 GB, hence the extra measurements in Figure 10b for the purposes of achieving higher bandwidths. Peak realized bandwidth for the dual GPU implementation was ~ 1568 GB/s including data transfer time, and 1744 GB/s excluding it; 87 % and 97 % respectively of the theoretical peak rating of 2 x 900 GB/s = 1800 GB/s. The primary performance advantage of the dual-GPU implementation is seen at high memory footprints.

4 NCCL & MPI

As seen in previous sections, CUDA and OpenMP provide robust solutions for parallel computations, but their implementations alone are confined to a single node environment. To leverage multiple GPUs across nodes, we need to take advantage of specialized libraries such as the NVIDIA Collective Communications Library (NCCL) and the Message Passing Interface (MPI).

NCCL, developed by NVIDIA, provides routines that are capable of achieving high

bandwidth over inter-GPU communication. It is particularly effective when dealing with GPU-centric, high-performance computing, as it facilitates efficient communication between different GPUs, even across different machines. MPI is a message-passing system designed to function in distributed memory environments. It offers efficient, standardized, and portable message-passing programming. Together, MPI and NCCL enable robust programs which can GPU resources across nodes.

4.1 Four-GPU

All previous tests were run on a node with two NVIDIA Tesla V100 GPUs available. On this system, there is another node, with two more Tesla V100s available. As such, NCCL and MPI can be used to distribute the requisite computations of the Poisson problem to up to four GPUs. Doing so introduces additional overhead required by the initialization of both NCCL and MPI, as well as the additional memory management for sharing data between nodes during computation on the boundary data, and after computation for gathering results.

In the four-GPU solver, half of the domain is solved on each node, which is further split between the two GPUs on each node. Solving on each half of the domain is much the same as the dual GPU implementation in Section 3.2; using `cudaDeviceEnablePeerAccess()` on the domian boundary between GPUs. Additional logic is then required at the domain boundary between the nodes. Here, CUDA alone cannot access the data from the other node, and the NCCL point-to-point communication directives `ncclSend()` and `ncclRecv()` are invoked to exchange data between nodes.

The four-GPU implementaiton of the Poisson solver can be found in `jacobi_cuda_split_MPI()` (see Appendix). The iteration loop with some relevant parameter definitions and kernel calls are shown below

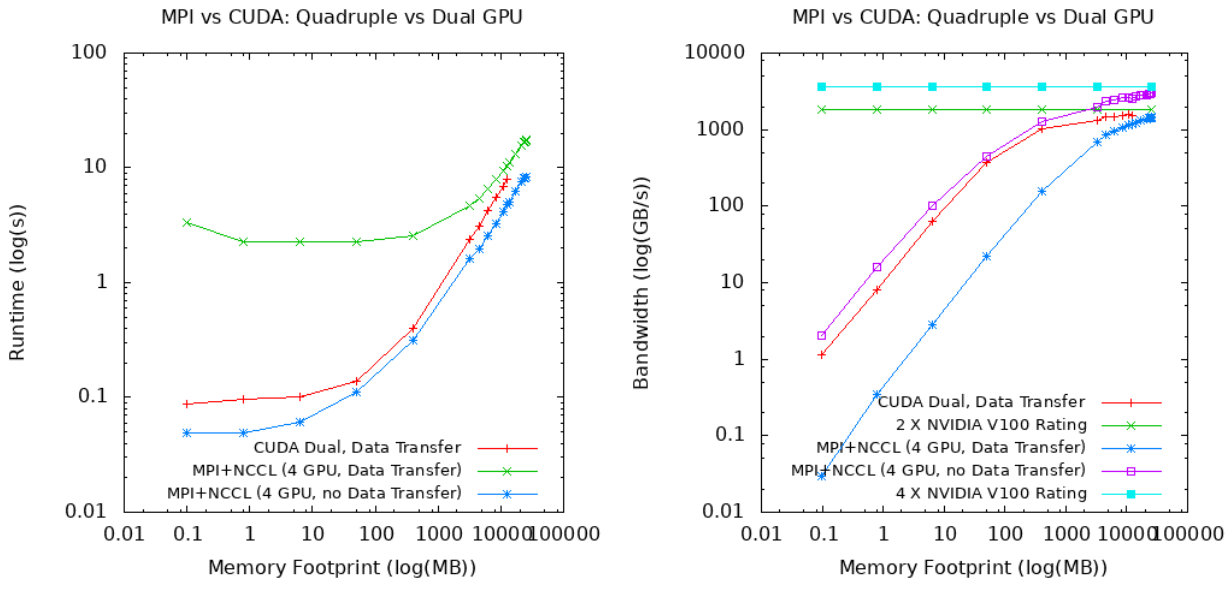
```
// define domain for internal points
dim3 blocks(8, 8, 16); // a 8x8x8 block of threads
dim3 grid((N) / blocks.x, (N) / blocks.y, (N/4) / blocks.z);

// for boundary kernel
dim3 blocks2D(8, 8); // a 8x8x8 block of threads
dim3 grid2D((N) / blocks2D.x, (N) / blocks2D.y);

double node_transfer;
int m = 0;
while (m<K) {
    // call kernel
    for (int i=0; i<2; i++) {
        cudaSetDevice(i);
        poisson_split_MPI<<<grid, blocks>>>(U_d[i],V_d[i],U_d[1-i],div,del2,
   N,i,myRank);
    }
    // synchronize
```

```
    for (int i=0; i<2; i++) {
        cudaSetDevice(i);
        CUDACHECK(cudaDeviceSynchronize());
    }

    // exchange boundary data between nodes with NCCL

    NCCLCHECK(ncclGroupStart());
    MPICHECK(MPI_Barrier(MPI_COMM_WORLD));
    node_transfer = omp_get_wtime();
    if (myRank == 0) {
        NCCLCHECK(ncclSend(U_d[1]+((N/4-1)*N*N), size, ncclDouble, 2, comms
   [1], s[1]));
        NCCLCHECK(ncclRecv(recvbuff[1], size, ncclDouble, 2, comms[1], s[1])
   );
    }
    else if (myRank == 1) {
        NCCLCHECK(ncclSend(U_d[0], size, ncclDouble, 1, comms[0], s[0]));
        NCCLCHECK(ncclRecv(recvbuff[0], size, ncclDouble, 1, comms[0], s[0])
   );
    }
    NCCLCHECK(ncclGroupEnd());
    *dt += omp_get_wtime()-node_transfer;

    // synchronize
    for (int g = 0; g < nDev; g++) {
        cudaSetDevice(g);
        cudaStreamSynchronize(s[g]);
    }

    // call boundary kernels
    if (myRank == 0) {
        cudaSetDevice(1);
        poisson_boundary<<<grid2D, blocks2D>>>(U_d[1],V_d[1],recvbuff[1],div
   ,del2,N,myRank);
    }
    else if (myRank == 1) {
        cudaSetDevice(0);
        poisson_boundary<<<grid2D, blocks2D>>>(U_d[0],V_d[0],recvbuff[0],div
   ,del2,N,myRank);
    }

    // update (pointer-swap)
    for (int i = 0; i < 2; i++) {
```

```
        double *old = U_d[i];
        U_d[i] = V_d[i];
        V_d[i] = old;
    }
    m += 1;
    }
```

One iteration consists firstly of an iteration on each half of the domain using the kernel `poisson_split_MPI`, which is analogous to `poisson_split`, with some extra logic for correct handling of boundaries based on which rank is calling the kernel. Next, `ncclSend()` and `ncclRecv()` are used to send boundary data between the nodes. Lastly, an additional kernel `poisson_boundary` is invoked to update the boundary points within each node using the data received from the other node. In the interest of brevity, these additional kernels are not shown here, but can be found in the Appendix.

Performance of the four-GPU implementation along with that of the dual GPU implementation is shown in Figure 11.



(a) Runtime **(b)** Memory Bandwidth

Figure 11: Quadruple vs Dual GPU Solver: Tesla V100-PCIE-32GB GPUs

Here, CUDA Dual shows the data for the dual-GPU implementation including data-transfer time. Realized bandwidth of the four-GPU implementation including and excluding data-transfer time peaked at 1438 GB/s and 3029 GB/s, or 40 % and 84 % of the peak theoretical bandwidth based on the device rating of 4 x 900 = 3600 GB/s. Notably, in the memory domains available for testing the respective solvers, performance of the four-GPU solver never exceeds that of the dual-GPU solver when including data-transfer, initialization

and allocation time. The additional overhead of NCCL and MPI initialization and communication evidently nullify most, if not all of the additional throughput afforded by more hardware, in the memory regime up to $N =\sim 800$ or 12 GB. The sole advantage of the four-GPU implementation here is therefore to allow for larger problem sizes. As with the dual vs single GPU case, the additional hardware (and thus additional memory) allows for larger problem sizes. The solver could manage problem sizes of $N \leq\sim 1020$, corresponding to a memory footprint of ~ 25 GB, hence the extra measurements in Figure 11b for the purposes of achieving higher bandwidths. The region of higher memory footprints from Figure 11b is shown in more detail in Figure 12

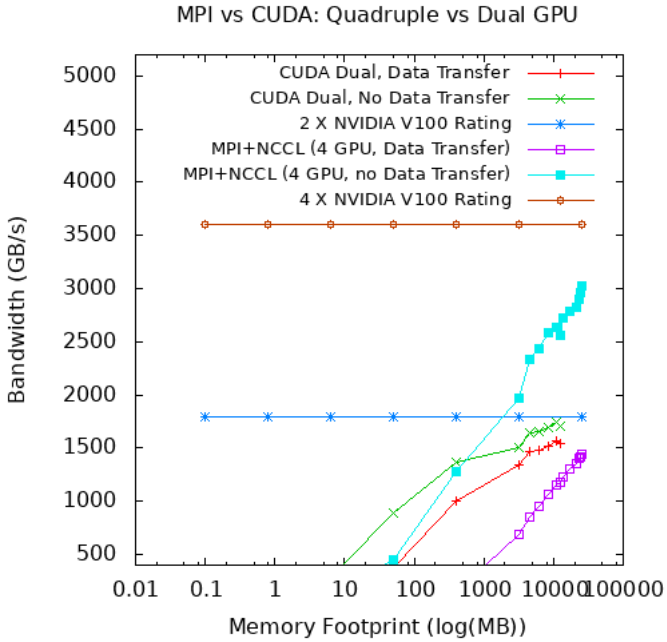


Figure 12: Quadruple vs Dual GPU Solver: Tesla V100-PCIE-32GB GPUs, High Memory Footprint Region

To give some insight into the behavior in this region, Figure 12 uses absolute values of bandwidth, as opposed to the log scale in the previous figure. Additionally, the dual-GPU data excluding data-transfer times is given. Note, it appears that the four-GPU implementation was bound by its global memory limitation ($N <\sim 1020$), before it had attained its maximum bandwidth, which was not the case for the dual-GPU implementation, i.e. we were not able to reach a compute-bound region of the four-GPU solver before running out of memory on the device. Last four readings of the dual-GPU performance were 1657 GB/s, 1693 GB/s, 1744 GB/s and 1713 GB/s for $N = [640, 704, 768, 800]$ respectively, while the last for readings of the four GPU solver were 2896 GB/s, 2963 GB/s, 3026 GB/s and 3029 GB/s for $N = [992, 1000, 1016, 1020]$ respectively. The data thus suggests that we were not able to quite realize the maximum attainable bandwidth of the given algorithm due to memory restrictions on the device. It is therefore likely that the evaluated performance of the algorithm is underestimated.

5 Conclusion

5.1 Results

Table 2 gives an overview of the performance of each solver, using full wall-time, i.e. including initialization, allocation and data-transfer, while Table 3 shows the same using estimates of purely execution time, i.e. time spent computing the solution for 1000 iterations and excluding the aforementioned overhead.

Solver	OMP Single	CUDA 1	CUDA 2	OMP Dual	CUDA Dual	4 GPU
Max BW (GB/s)	144	652	880	161	1568	1438
Rating (GB/s)	900	900	900	1800	1800	3600
Pct. vs Rating	16%	72%	98%	8.9%	87%	40%
N=128, K=1000	0.481 s	0.0816 s	0.0611 s	0.312 s	0.137 s	2.29 s
N=512, K=1000	184 s	5.36 s	3.86 s	107 s	2.40 s	4.70 s
N=640, K=1000	415 s	10.1 s	7.15 s	163 s	4.24 s	6.64 s
N=800, K=1000	out of memory	out of memory	out of memory	330 s	7.97 s	10.5 s
N=992, K=1000	out of memory	out of memory	out of memory	out of memory	out of memory	16.7s

Table 2: Performance Summary of Poisson solvers, including initialization, allocation and data-transfer

Solver	OMP Single	CUDA 1	CUDA 2	OMP Dual	CUDA Dual	4 GPU
Max BW (GB/s)	152	703	947	228	1744	3029
Rating (GB/s)	900	900	900	1800	1800	3600
Pct. vs Rating	17%	78%	105%	13%	97%	84%
N=128, K=1000	0.474 s	0.075 s	0.0564 s	0.222 s	0.0537 s	0.112 s
N=512, K=1000	184 s	5.01 s	3.60 s	106 s	2.13 s	1.61 s
N=640, K=1000	414 s	9.40 s	6.64 s	162 s	3.80 s	2.58 s
N=800, K=1000	out of memory	out of memory	out of memory	327 s	7.17 s	4.89 s
N=992, K=1000	out of memory	out of memory	out of memory	out of memory	out of memory	8.19 s

Table 3: Performance Summary of Poisson solvers, excluding initialization, allocation and data-transfer

To reiterate, CUDA 1 refers to the initial single-GPU implementation, and CUDA 2 refers to the improved version of the single-GPU implementation. The performance benefit of CUDA implementations over OpenMP implementations is unequivocal. Relative performance of CUDA implementations depends on the problem size in a predictable manner. In the lower memory regimes, e.g. $N = 128$, a single-GPU solver is best from a wall-time perspective. In medium-memory regimes, e.g. $N = 640$, the dual-GPU implementation significantly outperforms the single-GPU implementation, even when including the data-transfer overhead. Interestingly, from a total wall-time perspective, the four-GPU implementation

never outperforms the dual-GPU implementation in the allowable problem sizes for the dual-GPU solver. The only benefit of the four-GPU solver, given the hardware limitations of the system, is therefore only to allow computations with higher resolutions than supported by the other solvers. Presumably, if device storage were higher, one would see the four-GPU solver begin to outperform the dual-GPU solver for large problem sizes. The data which excludes the data-transfer, memory allocation and process initialization times, of course, tells a different story; with the four-GPU solver outperforming the dual-GPU solver, even in the medium memory regimes, but this is only to indicate proper utilization of the hardware. The data-transfer times are obviously crucial in making results available and thus decisions on the relative efficacy of program designs should thus be made using data which includes the full program wall-time.

5.2 Possible Extensions

For ease of indexing, the domain was split along the z axis, when split between either two or four GPUs. This means that there are N^2 boundary elements when split between two GPUs, and $3N^2$ boundary elements when split between four GPUs. There will always be N^2 boundary elements in the two GPU case. If however, one were to instead split by making two bisecting planes through the cubical domain, rather than three parallel planes, there would still be four regions, but there would be only $2N^2$ boundary element, as opposed to $3N^2$ when using parallel planes. This could offer significant performance benefit by reducing the data-transfer bottleneck.

As written, there is branching within the CUDA kernels to account for the given boundary conditions. These kernels could be tuned such that no thread needs to check the boundary condition because each thread would occupy its own point on the domain. This would of course require very specific choices of problem sizes, but could result in added performance.

6 Appendix

6.1 Project Directory and Environment

Provided in the project directory submitted with the project are all test scripts and plotting scripts. The README file describes how to use the executable.

Below are the relevant modules and environment variables for compiling and running the executable.

```
export CUDA_VISIBLE_DEVICES=0,1
export OMPI_CXX=nvc++
module purge
module load nvhpc/23.1-nompi
module load cuda/12.1
module load gcc/12.2.0-binutils-2.39
module load mpi/4.1.5-nvhpc-23.1-avx512
```

6.2 Hardware

6.2.1 CPU

Used for initial OpenMP CPU implementation.

```
CPU Info:
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    1
Core(s) per socket:    12
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Gold 6226 CPU @ 2.70GHz
Stepping:              7
CPU MHz:               3500.000
CPU max MHz:           3700.0000
CPU min MHz:           1200.0000
BogoMIPS:              5400.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              19712K
NUMA node0 CPU(s):     0-11
NUMA node1 CPU(s):     12-23
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
   mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe
   syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts
   rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq
   dtes64 ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca
   sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c
   rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb cat_l3 cdp_l3
   invpcid_single intel_ppin ssbd mba ibrs ibpb stibp ibrs_enhanced
   tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1
```

```
   avx2 smep bmi2 erms invpcid cqm mpx rdt_a avx512f avx512dq rdseed adx
   smap clflushopt clwb intel_pt avx512cd avx512bw avx512vl xsaveopt xsavec
    xgetbv1 xsaves cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local dtherm
    ida arat pln pts pku ospke avx512_vnni md_clear flush_l1d
   arch_capabilities
```

6.2.2 GPU

Used for profiling. For MPI implementations, each node contains two of the Tesla V100s shown below.

```
+-----------------------------------------------------------------------------
| NVIDIA-SMI 535.54.03              Driver Version: 535.54.03    CUDA
   Version: 12.2     |
|-----------------------------------------+----------------------+----------
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile
   Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util
    Compute M. |
|                                         |                      |
        MIG M. |
|=========================================+======================+=========
|   0  NVIDIA A100-PCIE-40GB          On  | 00000000:37:00.0 Off |
             0 |
| N/A   28C    P0              35W / 250W |      4MiB / 40960MiB |      0%
       Default |
|                                         |                      |
      Disabled |
+-----------------------------------------+----------------------+----------
|   1  NVIDIA A100-PCIE-40GB          On  | 00000000:86:00.0 Off |
             0 |
| N/A   26C    P0              31W / 250W |      4MiB / 40960MiB |      0%
       Default |
|                                         |                      |
      Disabled |
+-----------------------------------------+----------------------+----------
```

Used for all other results

```
+-----------------------------------------------------------------------------
| NVIDIA-SMI 530.41.03              Driver Version: 530.41.03    CUDA
   Version: 12.1     |
|-----------------------------------------+----------------------+-----------
| GPU  Name                  Persistence-M| Bus-Id        Disp.A | Volatile
   Uncorr. ECC |
| Fan  Temp  Perf            Pwr:Usage/Cap|         Memory-Usage | GPU-Util
    Compute M. |
|                                         |                      |
        MIG M. |
|=========================================+======================+==========
|   0  Tesla V100-PCIE-32GB           On | 00000000:58:00.0 Off |
             0 |
| N/A   31C    P0              27W / 250W|      0MiB / 32768MiB |      0%
    E. Process |
|                                         |                      |
           N/A |
+-----------------------------------------+----------------------+-----------
|   1  Tesla V100-PCIE-32GB           On | 00000000:D8:00.0 Off |
             0 |
| N/A   32C    P0              27W / 250W|      0MiB / 32768MiB |      0%
    E. Process |
|                                         |                      |
           N/A |
+-----------------------------------------+----------------------+-----------
```

6.3 Code

All solver functions and kernels are provided below. Relevant source files and makefile is provided with the project

```
/* jacobi.c - Poisson problem in 3d
 *
 */
#include <math.h>
#include <stdio.h>
```

```
#include <time.h>
#include <omp.h>
#include <cuda.h>
#include <stdbool.h>
#include "/appl/nccl/2.17.1-1-cuda-12.1/include/nccl.h"
#include <mpi.h>
#include <stdlib.h>
#include <iostream>
#include <cuda_runtime.h>
#include <stdint.h>
#include <unistd.h>
#include "alloc3d_cuda.h"
#include "print.h"

/*

MACROS

*/
#define IDX3D(i, j, k, n) ((k) + (n)*(j) + (i)*(n)*(n))
#define IDX2D(j, k, n) ((k) + (n)*(j))

#define MPICHECK(cmd) do {                          \
  int e = cmd;                                      \
  if( e != MPI_SUCCESS ) {                          \
    printf("Failed: MPI error %s:%d '%d'\n",        \
        __FILE__,__LINE__, e);   \
    exit(EXIT_FAILURE);                             \
  }                                                 \
} while(0)

#define CUDACHECK(cmd) do {                         \
  cudaError_t e = cmd;                              \
  if( e != cudaSuccess ) {                          \
    printf("Failed: Cuda error %s:%d '%s'\n",             \
        __FILE__,__LINE__,cudaGetErrorString(e));   \
    exit(EXIT_FAILURE);                             \
  }                                                 \
} while(0)


#define NCCLCHECK(cmd) do {                         \
  ncclResult_t r = cmd;                             \
  if (r!= ncclSuccess) {                            \
```

```
    printf("Failed, NCCL error %s:%d '%s'\n",             \
        __FILE__,__LINE__,ncclGetErrorString(r));   \
    exit(EXIT_FAILURE);                             \
  }                                                 \
} while(0)


static uint64_t getHostHash(const char* string) {
  // Based on DJB2a, result = result * 33 ^ char
  uint64_t result = 5381;
  for (int c = 0; string[c] != '\0'; c++){
    result = ((result << 5) + result) ^ string[c];
  }
  return result;
}


static void getHostName(char* hostname, int maxlen) {
  gethostname(hostname, maxlen);
  for (int i=0; i< maxlen; i++) {
    if (hostname[i] == '.') {
        hostname[i] = '\0';
        return;
    }
  }
}

/*

KERNELS

*/
__global__
void poisson_single(double *u_d, double *v_d, double *f_d, double div,
   double del2, int n) {

    /*
    Single GPU kernel
    */

    // get global indices
    int k = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int i = blockIdx.z * blockDim.z + threadIdx.z;
```

```
    bool boundary = (i == (n-1) || j == (n-1) || k == (n-1) || i == 0 || j
   == 0 || k == 0);
    if (!boundary) {
        int global = IDX3D(i,j,k,n);

        // naming convention from perspective of xz plane
        double down = i > 1 ? u_d[IDX3D(i-1,j,k,n)] : 20.0;
        double up = i < n-2 ? u_d[IDX3D(i+1,j,k,n)] : 20.0;
        double back = j > 1 ? u_d[IDX3D(i,j-1,k,n)] : 0.0;
        double forward = j < n-2 ? u_d[IDX3D(i,j+1,k,n)] : 20.0;
        double right = k < n-2 ? u_d[IDX3D(i,j,k+1,n)] : 20.0;
        double left = k > 1 ? u_d[IDX3D(i,j,k-1,n)] : 20.0;

        // make update
        v_d[global] = div*(down+up+left+right+forward+back+del2*f_d[global])
   ;
    }
}

__global__
void poisson_single_2(double *u_d, double *v_d, double *f_d, double div,
   double del2, int n) {

    /*
    Single GPU kernel, no branching
    */

    // get global indices
    int k = blockIdx.x * blockDim.x + threadIdx.x+1;
    int j = blockIdx.y * blockDim.y + threadIdx.y+1;
    int i = blockIdx.z * blockDim.z + threadIdx.z+1;

    int global = IDX3D(i,j,k,n);

    // naming convention from perspective of xz plane
    double right = u_d[IDX3D(i,j+1,k,n)];
    double forward = u_d[IDX3D(i,j,k+1,n)];
    double up = u_d[IDX3D(i+1,j,k,n)];
    double back = u_d[IDX3D(i,j,k-1,n)];
    double left = u_d[IDX3D(i,j-1,k,n)];
    double down = u_d[IDX3D(i-1,j,k,n)];

    // make update
    v_d[global] = div*(down+up+left+right+forward+back+del2*f_d[global]);
```

```
}

__global__
void poisson_single_3(double *u_down, double *u, double *u_up, double *v,
    double *f, double div, double del2, int n) {

    /*
    Single GPU kernel, layer-by-layer
    */

    // get global indices
    int k = blockIdx.x * blockDim.x + threadIdx.x+1;
    int j = blockIdx.y * blockDim.y + threadIdx.y+1;
    int global = IDX2D(j,k,n);

    // naming convention from perspective of xz plane
    double down = u_down[global];
    double up = u_up[global];
    double left = u[IDX2D(j,k-1,n)];
    double right = u[IDX2D(j,k+1,n)];
    double forward = u[IDX2D(j+1,k,n)];
    double back = u[IDX2D(j-1,k,n)];

    // make update
    v[global] = div*(down+up+left+right+forward+back+del2*f[global]);
}

#define BLOCK_SIZE 8
#define SHARED_SIZE (BLOCK_SIZE + 2)  // adding halo cells

__global__
void poisson_single_4(double *u_d, double *v_d, double *f_d, double div,
    double del2, int n) {

    /*
    Single GPU kernel, shared memory implementation
    */

    int bx = blockIdx.x;
    int by = blockIdx.y;
    int bz = blockIdx.z;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
```

```
    int tz = threadIdx.z;
    int x = bx * blockDim.x + tx;
    int y = by * blockDim.y + ty;
    int z = bz * blockDim.z + tz;
    bool boundary = (x == (n-1) || y == (n-1) || z == (n-1) || x == 0 || y
   == 0 || z == 0);

    __shared__ double u_shared[SHARED_SIZE][SHARED_SIZE][SHARED_SIZE];
    __shared__ double f_shared[SHARED_SIZE][SHARED_SIZE][SHARED_SIZE];

    // Load input elements into shared memory
    if (x < n && y < n && z < n) {
        u_shared[tx+1][ty+1][tz+1] = u_d[IDX3D(x,y,z,n)];
        f_shared[tx+1][ty+1][tz+1] = f_d[IDX3D(x,y,z,n)];
    }

    // Load halo elements
    if (tx == 0 && x > 0) u_shared[0][ty+1][tz+1] = u_d[IDX3D(x-1,y,z,n)];
    if (ty == 0 && y > 0) u_shared[tx+1][0][tz+1] = u_d[IDX3D(x,y-1,z,n)];
    if (tz == 0 && z > 0) u_shared[tx+1][ty+1][0] = u_d[IDX3D(x,y,z-1,n)];
    if (tx == BLOCK_SIZE-1 && x < n-1) u_shared[BLOCK_SIZE+1][ty+1][tz+1] =
   u_d[IDX3D(x+1,y,z,n)];
    if (ty == BLOCK_SIZE-1 && y < n-1) u_shared[tx+1][BLOCK_SIZE+1][tz+1] =
   u_d[IDX3D(x,y+1,z,n)];
    if (tz == BLOCK_SIZE-1 && z < n-1) u_shared[tx+1][ty+1][BLOCK_SIZE+1] =
   u_d[IDX3D(x,y,z+1,n)];

    __syncthreads();

    if (!boundary) {
        double down = u_shared[tx][ty+1][tz+1];
        double up = u_shared[tx+2][ty+1][tz+1];
        double left = u_shared[tx+1][ty][tz+1];
        double right = u_shared[tx+1][ty+2][tz+1];
        double forward = u_shared[tx+1][ty+1][tz+2];
        double back = u_shared[tx+1][ty+1][tz];
        v_d[IDX3D(x,y,z,n)] = div*(down+up+left+right+forward+back+del2*
   f_shared[tx+1][ty+1][tz+1]);
    }
}

__global__
void poisson_single_5(double *u_d, double *v_d, double div, double del2, int
    n) {
```

```
    /*
    Single GPU kernel
    */

    // get global indices
    int k = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int i = blockIdx.z * blockDim.z + threadIdx.z;

    // variables
    double x, y, z;
    bool rad, boundary;

    // location
    x = -1.0 + (2.0/(double)n)*(double)k;
    y = -1.0 + (2.0/(double)n)*(double)j;
    z = -1.0 + (2.0/(double)n)*(double)i;
    rad = (-1.0 <= x) && (x <= -3.0/8.0) && (-1.0 <= y) && (y <= -1.0/2.0)
   && (-2.0/3.0 <= z) && (z <= 0.0);
    boundary = (i == (n-1) || j == (n-1) || k == (n-1) || i == 0 || j == 0
   || k == 0);
    if (!boundary) {
        int global = IDX3D(i,j,k,n);

        // naming convention from perspective of xz plane
        double right = k < n-2 ? u_d[IDX3D(i,j,k+1,n)] : 20.0;
        double forward = j < n-2 ? u_d[IDX3D(i,j+1,k,n)] : 20.0;
        double up = i < n-2 ? u_d[IDX3D(i+1,j,k,n)] : 20.0;
        double back = j > 1 ? u_d[IDX3D(i,j-1,k,n)] : 0.0;
        double left = k > 1 ? u_d[IDX3D(i,j,k-1,n)] : 20.0;
        double down = i > 1 ? u_d[IDX3D(i-1,j,k,n)] : 20.0;

        // make update
        if (rad) v_d[global] = div*(down+up+left+right+forward+back+del2
   *200.00);
        else v_d[global] = div*(down+up+left+right+forward+back);
    }
}

__global__
void poisson_split(double *u_d1, // matricies for 1st GPU
                   double *v_d,
                   double *u_d2, // matricies for 2nd GPU
```

```
                    double div,   // coefficients for stencil formula
                    double del2,
                    int n,        // dimensions of XY region
                    int dev      // device number
                    ) {

    /*
    Double GPU kernel, remove F accesses,
    can also reduce number of arguments, just change update
    based on device number
    */

    // get global indices
    int k = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int i = blockIdx.z * blockDim.z + threadIdx.z;

    // variables
    int half = n/2;
    int global = IDX3D(i,j,k,n);
    double x, y, z;
    bool rad, boundary, i_boundary;

    // location
    x = -1.0 + (2.0/(double)n)*(double)k;
    y = -1.0 + (2.0/(double)n)*(double)j;
    z = -1.0 + (2.0/(double)n)*(double)i + 1.0*dev;
    rad = (-1.0 <= x) && (x <= -3.0/8.0) && (-1.0 <= y) && (y <= -1.0/2.0)
&& (-2.0/3.0 <= z) && (z <= 0.0);
    i_boundary = ((dev == 0) && (i == 0)) || ((dev == 1) && (i == (half-1)))
;
    boundary = (i_boundary || j == (n-1) || k == (n-1) || j == 0 || k == 0);

    // naming convention from perspective of xz plane
    if (!boundary) {
        double down, up, back, forward, left, right;
        if (dev == 0) {
            right = k < n-2 ? u_d1[IDX3D(i,j,k+1,n)] : 20.0;
            forward = j < n-2 ? u_d1[IDX3D(i,j+1,k,n)] : 20.0;
            up = i < (half-1) ? u_d1[IDX3D(i+1,j,k,n)] : u_d2[IDX3D(0,j,k,n)
];
            back = j > 1 ? u_d1[IDX3D(i,j-1,k,n)] : 0.0;
            left = k > 1 ? u_d1[IDX3D(i,j,k-1,n)] : 20.0;
            down = i > 1 ? u_d1[IDX3D(i-1,j,k,n)] : 20.0;
```

```
        }
        else if (dev == 1) {
            right = k < n-2 ? u_d1[IDX3D(i,j,k+1,n)] : 20.0;
            forward = j < n-2 ? u_d1[IDX3D(i,j+1,k,n)] : 20.0;
            up = i < (half-2) ? u_d1[IDX3D(i+1,j,k,n)] : 20.0;
            back = j > 1 ? u_d1[IDX3D(i,j-1,k,n)] : 0.0;
            left = k > 1 ? u_d1[IDX3D(i,j,k-1,n)] : 20.0;
            down = i > 0 ? u_d1[IDX3D(i-1,j,k,n)] : u_d2[IDX3D(half-1,j,k,n)
   ];
        }
        // make update
        if (rad) v_d[global] = div*(down+up+left+right+forward+back+del2
   *200.00);
        else v_d[global] = div*(down+up+left+right+forward+back);
    }
}

__global__
void poisson_split_MPI(double *u_d1, // matricies for 1st GPU
                       double *v_d,
                       double *u_d2, // matricies for 2nd GPU
                       double div,   // coefficients for stencil formula
                       double del2,
                       int n,        // dimensions of XY region
                       int dev,      // device number
                       int rank      // MPI rank
                   ) {

    /*
    Double GPU kernel for MPI implementation (will solve 1/2 of domain)
    */

    // get global indices
    int k = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int i = blockIdx.z * blockDim.z + threadIdx.z;
    int global = IDX3D(i,j,k,n);
    int half = n/4; // notice factor of 4 here, as opposed to 2 for
   poisson_split kernel

    // domain variables
    bool z_boundaries, boundary, rad, out;
    double x, y, z;
    out = (i > half-1) || (i < 0);
```

```
    z_boundaries = ((dev == 0) && (i == 0)) || ((dev == 1) && (i == (half-1)
    ));
    boundary = (z_boundaries || j >= (n-1) || k >= (n-1) || j <= 0 || k <= 0
    || out);

    // location
    x = -1.0 + (2.0/(double)n)*(double)k;
    y = -1.0 + (2.0/(double)n)*(double)j;
    z = -1.0 + (2.0/(double)n)*(double)i + 0.5*dev + 1.0*rank;
    rad = (-1.0 <= x) && (x <= -3.0/8.0) && (-1.0 <= y) && (y <= -1.0/2.0)
    && (-2.0/3.0 <= z) && (z <= 0.0);

    // naming convention from perspective of xz plane
    // if statements go from bottom to top of domain
    if (!boundary) {
    double up, down, left, right, forward, back;
    // bottom quarter
    if (dev == 0 && rank == 0) {
        right = k < n-2 ? u_d1[IDX3D(i,j,k+1,n)] : 20.0;
        forward = j < n-2 ? u_d1[IDX3D(i,j+1,k,n)] : 20.0;
        up = i < (half-1) ? u_d1[IDX3D(i+1,j,k,n)] : u_d2[IDX3D(0,j,k,n)];
        back = j > 1 ? u_d1[IDX3D(i,j-1,k,n)] : 0.0;
        left = k > 1 ? u_d1[IDX3D(i,j,k-1,n)] : 20.0;
        down = i > 1 ? u_d1[IDX3D(i-1,j,k,n)] : 20.0;
    }
    // 2nd quarter
    else if (dev == 1 && rank == 0) {
        right = k < n-2 ? u_d1[IDX3D(i,j,k+1,n)] : 20.0;
        forward = j < n-2 ? u_d1[IDX3D(i,j+1,k,n)] : 20.0;
        up = u_d1[IDX3D(i+1,j,k,n)];  // potential node boundary
        back = j > 1 ? u_d1[IDX3D(i,j-1,k,n)] : 0.0;
        left = k > 1 ? u_d1[IDX3D(i,j,k-1,n)] : 20.0;
        down = i > 0 ? u_d1[IDX3D(i-1,j,k,n)] : u_d2[IDX3D(half-1,j,k,n)];
    }
    // 3rd quarter
    else if (dev == 0 && rank == 1) {
        right = k < n-2 ? u_d1[IDX3D(i,j,k+1,n)] : 20.0;
        forward = j < n-2 ? u_d1[IDX3D(i,j+1,k,n)] : 20.0;
        up = i < (half-1) ? u_d1[IDX3D(i+1,j,k,n)] : u_d2[IDX3D(0,j,k,n)];
        back = j > 1 ? u_d1[IDX3D(i,j-1,k,n)] : 0.0;
        left = k > 1 ? u_d1[IDX3D(i,j,k-1,n)] : 20.0;
        down = u_d1[IDX3D(i-1,j,k,n)]; // potential node boundary
    }
    // top quarter
```

```
    else if (dev == 1 && rank == 1) {
        right = k < n-2 ? u_d1[IDX3D(i,j,k+1,n)] : 20.0;
        forward = j < n-2 ? u_d1[IDX3D(i,j+1,k,n)] : 20.0;
        up = i < (half-2) ? u_d1[IDX3D(i+1,j,k,n)] : 20.0;
        back = j > 1 ? u_d1[IDX3D(i,j-1,k,n)] : 0.0;
        left = k > 1 ? u_d1[IDX3D(i,j,k-1,n)] : 20.0;
        down = i > 0 ? u_d1[IDX3D(i-1,j,k,n)] : u_d2[IDX3D(half-1,j,k,n)];
    }
    // make update
    if (rad) v_d[global] = div*(down+up+left+right+forward+back+del2*200.00)
   ;
    else v_d[global] = div*(down+up+left+right+forward+back);
    }
}

__global__
void poisson_boundary(double *u_d, // local matricies
                      double *v_d,
                      double *u_d2, // boundary data (recvbuff)
                      double div,   // coefficients for stencil formula
                      double del2,
                      int n,        // dimensions of XY region
                      int rank      // MPI rank
                        ) {
    /*
    Boundary Kernel
    */

    // get global indices
    int k = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int global = 0;
    int half = n/4;
    if (rank == 0) global = IDX3D((half-1),j,k,n);
    else if (rank == 1) global = IDX3D(0,j,k,n);
    bool boundary, rad;
    boundary = (j == (n-1) || k == (n-1) || j == 0 || k == 0);

    // location
    double x, y, z;
    x = -1.0 + (2.0/(double)n)*(double)k;
    y = -1.0 + (2.0/(double)n)*(double)j;
    z = 0.0;
    rad = (-1.0 <= x) && (x <= -3.0/8.0) && (-1.0 <= y) && (y <= -1.0/2.0)
```

```
   && (-2.0/3.0 <= z) && (z <= 0.0);

    if (!boundary){
    double up, down, left, right, forward, back;
    if (rank == 0) {
        // naming convention from perspective of xz plane
        right = j < n-2 ? u_d[IDX3D(half-1,j+1,k,n)] : 20.0;
        forward = k < n-2 ? u_d[IDX3D(half-1,j,k+1,n)] : 20.0;
        back = k > 1 ? u_d[IDX3D(half-1,j,k-1,n)] : 20.0;
        left = j > 1 ? u_d[IDX3D(half-1,j-1,k,n)] : 0.0;
        down = u_d[IDX3D(half-2,j,k,n)];
        up = u_d2[IDX3D(0,j,k,n)];
    }
    else if (rank == 1) {
        // naming convention from perspective of xz plane
        right = j < n-2 ? u_d[IDX3D(0,j+1,k,n)] : 20.0;
        forward = k < n-2 ? u_d[IDX3D(0,j,k+1,n)] : 20.0;
        back = k > 1 ? u_d[IDX3D(0,j,k-1,n)] : 20.0;
        left = j > 1 ? u_d[IDX3D(0,j-1,k,n)] : 0.0;
        down = u_d2[IDX3D(0,j,k,n)];
        up = u_d[IDX3D(1,j,k,n)];

    }
    // make update
    if (rad) v_d[global] = div*(down+up+left+right+forward+back+del2*200.00)
   ;
    else v_d[global] = div*(down+up+left+right+forward+back);
    }
}

/*

SOLVERS

*/
void jacobi_cuda_single(double ***U, double ***V, double ***F, int K, int N,
     double *wt, double *dt, double *kt) {
    /*
    Single-GPU cuda implementation
    */

    // device pointers for matiriies
    double *U_d;
    double *V_d;
```

```
    double *F_d;

    // measure execution time
    double start_run = omp_get_wtime();

    // allocate
    const int sizeBytes = N*N*N*sizeof(double);
    CUDACHECK(cudaMalloc((void **)&U_d, sizeBytes));
    CUDACHECK(cudaMalloc((void **)&V_d, sizeBytes));
    CUDACHECK(cudaMalloc((void **)&F_d, sizeBytes));

    // populate
    cudaMemcpy(U_d, U[0][0], N*N*N*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(V_d, V[0][0], N*N*N*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(F_d, F[0][0], N*N*N*sizeof(double), cudaMemcpyHostToDevice);

    // stop data-transfer clock
    double end_t = omp_get_wtime();
    double duration_dt = end_t - start_run;
    *dt = duration_dt;

    // constants
    double del = 2.0/(double)(N);  // grid size
    double div = (1.0/6.0);          // factor fo 1/6 from seven-point
   stencil formula
    double del2=(del*del);           // del squared

    // define domain for kernel
    dim3 blocks(8, 8, 16); // a 8x8x8 block of threads
    dim3 grid((N) / blocks.x, (N) / blocks.y, (N) / blocks.z); // enough
   blocks to cover the whole 3D domain
    int m = 0;

    // call kernel
    // kernel<<<NUM_BLOCKS, THREADS_PER_BLOCK>>>(...);
    *kt = 0.0;
    double sk;
    while (m<K) {
        sk = omp_get_wtime();
        poisson_single<<<grid, blocks>>>(U_d,V_d,F_d,div,del2,N);
        cudaDeviceSynchronize();
        *kt += omp_get_wtime()-sk;
        m += 1;
        double *old = U_d;
```

```
        U_d = V_d;
        V_d = old;
    }

    // transfer data back to host and add to data transfer measurement
    double out = omp_get_wtime();
    cudaMemcpy(U[0][0], U_d, N*N*N*sizeof(double), cudaMemcpyDeviceToHost);

    // free device memory
    free_3d_cuda(U_d);
    free_3d_cuda(V_d);
    free_3d_cuda(F_d);

    // end execution time measurement
    *dt += omp_get_wtime()-out;
    *wt = omp_get_wtime()-start_run;
}

void jacobi_cuda_single_2(double ***U, double ***V, double ***F, int K, int
   N, double *wt, double *dt, double *kt) {

    /*
    Single-GPU cuda implementation, without any kernel branching
    */

    // device pointers for matiriies
    double *U_d;
    double *V_d;
    double *F_d;

    // measure execution time
    double start_run = omp_get_wtime();

    // allocate
    const int sizeBytes = N*N*N*sizeof(double);
    CUDACHECK(cudaMalloc((void **)&U_d, sizeBytes));
    CUDACHECK(cudaMalloc((void **)&V_d, sizeBytes));
    CUDACHECK(cudaMalloc((void **)&F_d, sizeBytes));

    // populate
    cudaMemcpy(U_d, U[0][0], N*N*N*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(V_d, V[0][0], N*N*N*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(F_d, F[0][0], N*N*N*sizeof(double), cudaMemcpyHostToDevice);
```

```
    // stop data-transfer clock
    double end_t = omp_get_wtime();
    double duration_dt = end_t - start_run;
    *dt = duration_dt;

    // constants
    double del = 2.0/(double)(N);  // grid size
    double div = (1.0/6.0);          // factor fo 1/6 from seven-point
   stencil formula
    double del2=(del*del);           // del squared

    // define domain for kernel
    dim3 blocks(8, 8, 16); // a 8x8x8 block of threads
    dim3 grid((N-2) / blocks.x, (N-2) / blocks.y, (N-2) / blocks.z); //
   enough blocks to cover the whole 3D domain
    int m = 0;

    // call kernel
    // kernel<<<NUM_BLOCKS, THREADS_PER_BLOCK>>>(...);
    *kt = 0.0;
    double sk;
    while (m<K) {
        sk = omp_get_wtime();
        poisson_single_2<<<grid, blocks>>>(U_d,V_d,F_d,div,del2,N);
        cudaDeviceSynchronize();
        *kt += omp_get_wtime()-sk;
        m += 1;
        double *old = U_d;
        U_d = V_d;
        V_d = old;
    }

    // transfer data back to host and add to data transfer measurement
    double out = omp_get_wtime();
    cudaMemcpy(U[0][0], U_d, N*N*N*sizeof(double), cudaMemcpyDeviceToHost);

    // free device memory
    free_3d_cuda(U_d);
    free_3d_cuda(V_d);
    free_3d_cuda(F_d);

    // end execution time measurement
    *dt += omp_get_wtime()-out;
    *wt = omp_get_wtime()-start_run;
```

```
}

void jacobi_cuda_single_3(double ***U, double ***V, double ***F, int K, int
    N, double *wt, double *dt, double *kt) {

    /*
    Single-GPU cuda implementation: allocate each layer separately to try to
    reduce memory load on kernels
    */

    // measure execution time
    double start_run = omp_get_wtime();

    // device pointers for matiriies
    double *U_d[N];
    double *V_d[N];
    double *F_d[N];


    // allocate & populate for each layer
    const int sizeBytes = N*N*sizeof(double);
    for (int l=0;l<N;l++) {
        // allocate
        CUDACHECK(cudaMalloc(U_d+l, sizeBytes));
        CUDACHECK(cudaMalloc(V_d+l, sizeBytes));
        CUDACHECK(cudaMalloc(F_d+l, sizeBytes));

        // populate
        CUDACHECK(cudaMemcpyAsync(U_d[l], U[l][0], sizeBytes,
    cudaMemcpyHostToDevice));
        CUDACHECK(cudaMemcpyAsync(V_d[l], V[l][0], sizeBytes,
    cudaMemcpyHostToDevice));
        CUDACHECK(cudaMemcpyAsync(F_d[l], F[l][0], sizeBytes,
    cudaMemcpyHostToDevice));
    }

    // synchronize
    CUDACHECK(cudaDeviceSynchronize());

    // stop data-transfer clock
    double end_t = omp_get_wtime();
    double duration_dt = end_t - start_run;
    *dt = duration_dt;
```

```
 // constants
 double del = 2.0/(double)(N);  // grid size
 double div = (1.0/6.0);          // factor fo 1/6 from seven-point
stencil formula
 double del2=(del*del);           // del squared

 // define domain for kernel
 dim3 blocks(16, 8, 4); // a 8x8x8 block of threads
 dim3 grid((N-2) / blocks.x, (N-2) / blocks.y, (N-2) / blocks.z); //
enough blocks to cover the whole 3D domain
 int m = 0;

 // call kernel
 // kernel<<<NUM_BLOCKS, THREADS_PER_BLOCK>>>(...);

 /*
 TRY A LOOP SENDING 3-LAYERED KERNELS
 */

*kt = 0.00;
 while (m<K) {
     double start_k = omp_get_wtime();
     for (int j = 1; j<(N-1); j++) poisson_single_3<<<grid, blocks>>>(U_d
[j-1],U_d[j],U_d[j+1],V_d[j],F_d[j],div,del2,N);
     CUDACHECK(cudaDeviceSynchronize());
     *kt += omp_get_wtime()-start_k;
     for (int j = 1; j<(N-1); j++) {
         double *old = U_d[j];
         U_d[j] = V_d[j];
         V_d[j] = old;
     }
     m += 1;
 }

 // transfer data back to host and add to data transfer measurement
 double out = omp_get_wtime();
 for (int l=1;l<N-1;l++) {

     // populate
     CUDACHECK(cudaMemcpyAsync(U[l][0], U_d[l], sizeBytes,
cudaMemcpyDeviceToHost));
     CUDACHECK(cudaMemcpyAsync(V[l][0], U_d[l], sizeBytes,
cudaMemcpyDeviceToHost));
```

```
        CUDACHECK(cudaMemcpyAsync(F[l][0], U_d[l], sizeBytes,
    cudaMemcpyDeviceToHost));
    }
    CUDACHECK(cudaDeviceSynchronize());

    // free device memory
    for (int i = 0; i < N; i++) {
        cudaSetDevice(i);
        free_3d_cuda(U_d[i]);
        free_3d_cuda(V_d[i]);
        free_3d_cuda(F_d[i]);
    }
    // end execution time measurement
    *dt += omp_get_wtime()-out;
    *wt = omp_get_wtime()-start_run;
}

void jacobi_cuda_single_4(double ***U, double ***V, double ***F, int K, int
    N, double *wt, double *dt, double *kt) {

    /*
    Single-GPU cuda implementation, shared memory
    */

    // device pointers for matiriies
    double *U_d;
    double *V_d;
    double *F_d;

    // measure execution time
    double start_run = omp_get_wtime();

    // allocate
    const int sizeBytes = N*N*N*sizeof(double);
    CUDACHECK(cudaMalloc((void **)&U_d, sizeBytes));
    CUDACHECK(cudaMalloc((void **)&V_d, sizeBytes));
    CUDACHECK(cudaMalloc((void **)&F_d, sizeBytes));

    // populate
    cudaMemcpy(U_d, U[0][0], N*N*N*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(V_d, V[0][0], N*N*N*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(F_d, F[0][0], N*N*N*sizeof(double), cudaMemcpyHostToDevice);

    // stop data-transfer clock
```

```
    double end_t = omp_get_wtime();
    double duration_dt = end_t - start_run;
    *dt = duration_dt;

    // constants
    double del = 2.0/(double)(N);  // grid size
    double div = (1.0/6.0);          // factor fo 1/6 from seven-point
   stencil formula
    double del2=(del*del);           // del squared

    // define domain for kernel
    dim3 blocks(BLOCK_SIZE, BLOCK_SIZE, BLOCK_SIZE);
    dim3 grid((N + blocks.x - 1) / blocks.x, (N + blocks.y - 1) / blocks.y,
   (N + blocks.z - 1) / blocks.z);
    int m = 0;

    // call kernel
    // kernel<<<NUM_BLOCKS, THREADS_PER_BLOCK>>>(...);
    *kt = 0.0;
    double sk;
    while (m<K) {
        sk = omp_get_wtime();
        poisson_single_4<<<grid, blocks>>>(U_d,V_d,F_d,div,del2,N);
        CUDACHECK(cudaDeviceSynchronize());
        *kt += omp_get_wtime()-sk;
        m += 1;
        double *old = U_d;
        U_d = V_d;
        V_d = old;
    }

    // transfer data back to host and add to data transfer measurement
    double out = omp_get_wtime();
    cudaMemcpy(U[0][0], U_d, N*N*N*sizeof(double), cudaMemcpyDeviceToHost);

    // free device memory
    free_3d_cuda(U_d);
    free_3d_cuda(V_d);
    free_3d_cuda(F_d);

    *dt += omp_get_wtime()-out;
    *wt = omp_get_wtime()-start_run;

}
```

```
void jacobi_cuda_single_5(double ***U, double ***V, double ***F, int K, int
    N, double *wt, double *dt, double *kt) {
    /*
    Single-GPU cuda implementation, don't use a full array for F
    */

    // device pointers for matiriies
    double *U_d;
    double *V_d;

    // measure execution time
    double start_run = omp_get_wtime();

    // allocate
    const int sizeBytes = N*N*N*sizeof(double);
    CUDACHECK(cudaMalloc((void **)&U_d, sizeBytes));
    CUDACHECK(cudaMalloc((void **)&V_d, sizeBytes));

    // populate
    cudaMemcpy(U_d, U[0][0], N*N*N*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(V_d, V[0][0], N*N*N*sizeof(double), cudaMemcpyHostToDevice);

    // stop data-transfer clock
    double end_t = omp_get_wtime();
    double duration_dt = end_t - start_run;
    *dt = duration_dt;

    // constants
    double del = 2.0/(double)(N);  // grid size
    double div = (1.0/6.0);          // factor fo 1/6 from seven-point
    stencil formula
    double del2=(del*del);           // del squared

    // define domain for kernel
    dim3 blocks(8, 8, 16); // a 8x8x8 block of threads
    dim3 grid((N) / blocks.x, (N) / blocks.y, (N) / blocks.z); // enough
    blocks to cover the whole 3D domain
    int m = 0;

    // call kernel
    // kernel<<<NUM_BLOCKS, THREADS_PER_BLOCK>>>(...);
    *kt = 0.0;
    double sk;
```

```
    while (m<K) {
        sk = omp_get_wtime();
        poisson_single_5<<<grid, blocks>>>(U_d,V_d,div,del2,N);
        cudaDeviceSynchronize();
        *kt += omp_get_wtime()-sk;
        m += 1;
        double *old = U_d;
        U_d = V_d;
        V_d = old;
    }

    // transfer data back to host and add to data transfer measurement
    double out = omp_get_wtime();
    cudaMemcpy(U[0][0], U_d, N*N*N*sizeof(double), cudaMemcpyDeviceToHost);

    // free device memory
    free_3d_cuda(U_d);
    free_3d_cuda(V_d);

    // end execution time measurement
    *dt += omp_get_wtime()-out;
    *wt = omp_get_wtime()-start_run;
}

void jacobi_cuda_split(double ***U, double ***V, double ***F, int K, int N,
   double *wt, double *dt, double *kt) {

    // device pointers for matiriies
    double *U_d[2];
    double *V_d[2];

    // start clock
    double start = omp_get_wtime();

    const int sizeBytes = (N/2)*N*N*sizeof(double);

    // allocate & populate device memory
    for (int i=0; i<2; i++) {
        cudaSetDevice(i);
        // allocate

        // CHANGE TO DIRECT MALLOCS in CUDA
        CUDACHECK(cudaMalloc(U_d+i, sizeBytes));
        CUDACHECK(cudaMalloc(V_d+i, sizeBytes));
```

```
        //populate
        int start = i * (N/2);
        CUDACHECK(cudaMemcpyAsync(U_d[i], U[start][0], (N/2)*N*N*sizeof(
    double), cudaMemcpyHostToDevice));
        CUDACHECK(cudaMemcpyAsync(V_d[i], V[start][0], (N/2)*N*N*sizeof(
    double), cudaMemcpyHostToDevice));
    }

    // synchronize
    for (int i=0; i<2; i++) {
        cudaSetDevice(i);
        CUDACHECK(cudaDeviceSynchronize());
    }

    // stop data-transfer clock after update
    *dt = omp_get_wtime()-start;

    // Enable peer-to-peer access and offload
    cudaSetDevice(0);
    cudaDeviceEnablePeerAccess(1, 0); // (dev 1, future flag)
    cudaSetDevice(1);
    cudaDeviceEnablePeerAccess(0, 0); // (dev 0, future flag)

    // constants
    double del = 2.0/(double)(N);  // grid size
    double div = (1.0/6.0);          // factor fo 1/6 from seven-point
    stencil formula
    double del2=(del*del);           // del squared

    // define domain for GPU
    dim3 blocks(8, 8, 16); // a 8x8x8 block of threads
    dim3 grid((N) / blocks.x, (N) / blocks.y, (N/2) / blocks.z); // enough
    blocks to cover the whole 3D domain
    int m = 0;

    // kernel<<<NUM_BLOCKS, THREADS_PER_BLOCK>>>(...);
    *kt = 0.0;
    double sk;
    while (m<K) {
        // call kernel
        sk = omp_get_wtime();
        for (int i=0; i<2; i++) {
            cudaSetDevice(i);
```

```
            poisson_split<<<grid, blocks>>>(U_d[i],V_d[i],U_d[1-i],div,del2,
N,i);
        }
        // synchronize
        for (int i=0; i<2; i++) {
            cudaSetDevice(i);
            CUDACHECK(cudaDeviceSynchronize());
        }

        // update total kernel runtime
        *kt += omp_get_wtime()-sk;

        // update (pointer-swap)
        for (int i = 0; i < 2; i++) {
            double *old = U_d[i];
            U_d[i] = V_d[i];
            V_d[i] = old;
        }
        m += 1;
        }

    // copy back to host and add to data transfer-time
    double out = omp_get_wtime();
    for (int i=0; i<2; i++) {
        cudaSetDevice(i);
        int start = (N/2)*(i);
        cudaMemcpyAsync(U[start][0], U_d[i], (N/2)*N*N*sizeof(double),
cudaMemcpyDeviceToHost);
    }

    // synchronize
    for (int i=0; i<2; i++) {
        cudaSetDevice(i);
        CUDACHECK(cudaDeviceSynchronize());
    }

    // free device memory
    for (int i = 0; i < 2; i++) {
        cudaSetDevice(i);
        free_3d_cuda(U_d[i]);
        free_3d_cuda(V_d[i]);
    }

    // update clocks
```

```
    *dt += omp_get_wtime()-out;
    *wt = omp_get_wtime()-start;

}

void jacobi_cuda_split_MPI(double ***U,
                           double ***V,
                           double ***F,
                           int K,
                           int N,
                           double *wt,
                           double *dt,
                           int argc,
                           char *argv[],
                           char *output_filename,
                           char *output_prefix,
                           char *output_ext,
                           char *output_suffix,
                           int output_type) {

    double start = omp_get_wtime();

    // process info
    int myRank, nRanks, nDev = 0;
    int size = N*N;

    cudaGetDeviceCount(&nDev);

    //initializing MPI
    MPICHECK(MPI_Init(&argc, &argv));
    MPICHECK(MPI_Comm_rank(MPI_COMM_WORLD, &myRank));
    MPICHECK(MPI_Comm_size(MPI_COMM_WORLD, &nRanks));

    //get NCCL unique ID at rank 0 and broadcast it to all others
    ncclUniqueId id;
    if (myRank == 0) ncclGetUniqueId(&id);
    MPICHECK(MPI_Bcast((void *)&id, sizeof(id), MPI_BYTE, 0, MPI_COMM_WORLD)
   );

    //each process is using two GPUs
    double** sendbuff = (double**)malloc(nDev * sizeof(double*));
    double** recvbuff = (double**)malloc(nDev * sizeof(double*));
    ncclComm_t* comms = (ncclComm_t*)  malloc(sizeof(ncclComm_t)  * nDev);
    cudaStream_t* s = (cudaStream_t*)malloc(sizeof(cudaStream_t)*nDev);
```

```
 // initializing NCCL, group API is required around ncclCommInitRank as
it is
 // called across multiple GPUs in each thread/process
 // nccl ranks use a global numbering
 NCCLCHECK(ncclGroupStart());
 for (int i=0; i<nDev; i++) {
     CUDACHECK(cudaSetDevice(i));
     NCCLCHECK(ncclCommInitRank(comms+i, nRanks*nDev, id, myRank*nDev + i
));
 }
 NCCLCHECK(ncclGroupEnd());


 // device pointers for matiriies
 double *U_d[2];
 double *V_d[2];

 const int sizeBytes = (N/4)*N*N*sizeof(double);
 // allocate & populate device memory
 for (int i=0; i<nDev; i++) {
     // set device
     cudaSetDevice(i);

     // allocate
     CUDACHECK(cudaMalloc(U_d+i, sizeBytes));
     CUDACHECK(cudaMalloc(V_d+i, sizeBytes));

     //populate
     int start = ((2*myRank)+i)*(N/4);
     CUDACHECK(cudaMemcpy(U_d[i], U[start][0], (N/4)*N*N*sizeof(double),
cudaMemcpyHostToDevice));
     CUDACHECK(cudaMemcpy(V_d[i], V[start][0], (N/4)*N*N*sizeof(double),
cudaMemcpyHostToDevice));

     // create stream for each GPU
     CUDACHECK(cudaStreamCreate(s+i));
 }

 // populate buffers
 if (myRank == 0) {
     cudaSetDevice(1);
     CUDACHECK(cudaMalloc(recvbuff+1, N*N*sizeof(double)));
 }
```

```
    else if (myRank == 1) {
        cudaSetDevice(0);
        CUDACHECK(cudaMalloc(recvbuff, N*N*sizeof(double)));
    }

    // stop data-transfer clock after update
    MPICHECK(MPI_Barrier(MPI_COMM_WORLD));
    double end_t = omp_get_wtime();
    double duration_dt = end_t - start;
    *dt = duration_dt;

    // Enable peer-to-peer access and offload
    cudaSetDevice(0);
    cudaDeviceEnablePeerAccess(1, 0); // (dev 1, future flag)
    cudaSetDevice(1);
    cudaDeviceEnablePeerAccess(0, 0); // (dev 0, future flag)

    // constants
    double del = 2.0/(double)(N);  // grid size
    double div = (1.0/6.0);          // factor fo 1/6 from seven-point
   stencil formula
    double del2=(del*del);           // del squared

    // kernel<<<NUM_BLOCKS, THREADS_PER_BLOCK>>>(...);
    // define domain for internal points
    dim3 blocks(8, 4, 32); // a 8x8x8 block of threads
    dim3 grid((N) / blocks.x, (N) / blocks.y, ((N/4) / blocks.z)+1);

    // for boundary kernel
    dim3 blocks2D(16, 16); // a 8x8x8 block of threads
    dim3 grid2D((N) / blocks2D.x, (N) / blocks2D.y);

    double node_transfer;
    int m = 0;
    while (m<K) {
        // call kernel
        for (int i=0; i<2; i++) {
            cudaSetDevice(i);
            poisson_split_MPI<<<grid, blocks>>>(U_d[i],V_d[i],U_d[1-i],div,
   del2,N,i,myRank);
        }
        // synchronize
        for (int i=0; i<2; i++) {
            cudaSetDevice(i);
```

```
            CUDACHECK(cudaDeviceSynchronize());
        }

        // exchange boundary data between nodes with NCCL

        NCCLCHECK(ncclGroupStart());
        MPICHECK(MPI_Barrier(MPI_COMM_WORLD));
        node_transfer = omp_get_wtime();
        if (myRank == 0) {
            NCCLCHECK(ncclSend(U_d[1]+((N/4-1)*N*N), size, ncclDouble, 2,
    comms[1], s[1]));
            NCCLCHECK(ncclRecv(recvbuff[1], size, ncclDouble, 2, comms[1], s
    [1]));
        }
        else if (myRank == 1) {
            NCCLCHECK(ncclSend(U_d[0], size, ncclDouble, 1, comms[0], s[0]))
    ;
            NCCLCHECK(ncclRecv(recvbuff[0], size, ncclDouble, 1, comms[0], s
    [0]));
        }
        NCCLCHECK(ncclGroupEnd());
        *dt += omp_get_wtime()-node_transfer;

        // synchronize
        for (int g = 0; g < nDev; g++) {
            cudaSetDevice(g);
            cudaStreamSynchronize(s[g]);
        }

        // call boundary kernels
        if (myRank == 0) {
            cudaSetDevice(1);
            poisson_boundary<<<grid2D, blocks2D>>>(U_d[1],V_d[1],recvbuff
    [1],div,del2,N,myRank);
        }
        else if (myRank == 1) {
            cudaSetDevice(0);
            poisson_boundary<<<grid2D, blocks2D>>>(U_d[0],V_d[0],recvbuff
    [0],div,del2,N,myRank);
        }

        // update (pointer-swap)
        for (int i = 0; i < 2; i++) {
            double *old = U_d[i];
```

```
            U_d[i] = V_d[i];
            V_d[i] = old;
        }
        m += 1;
        }

    MPICHECK(MPI_Barrier(MPI_COMM_WORLD));
    double dt2 = omp_get_wtime();
    int numElements = (N/2)*N*N;
    double *recvbuffMPI = (double *) malloc(2*numElements*sizeof(double));
   // must have room for both processes in recv buffer

    // get all data onto one rank
    for (int i=0; i<2; i++) {
        cudaSetDevice(i);
        int start = ((2*myRank)+i)*(N/4);
        cudaMemcpy(U[start][0], U_d[i], (N/4)*N*N*sizeof(double),
   cudaMemcpyDeviceToHost);
    }

    // if elements exceed max, split in half and use two gather commands
    if (N < 704) {
        MPICHECK(MPI_Gather(U[N/2][0], numElements, MPI_DOUBLE, recvbuffMPI,
     numElements, MPI_DOUBLE, 0, MPI_COMM_WORLD));

        // extract data from recvbuff
        if (myRank == 0) memcpy(U[N/2][0],recvbuffMPI+numElements,
   numElements*sizeof(double));
    }
    else if (N < 832) {
        int halfNumElements = numElements/2;
        MPICHECK(MPI_Gather(U[N/2][0], halfNumElements, MPI_DOUBLE,
   recvbuffMPI, halfNumElements, MPI_DOUBLE, 0, MPI_COMM_WORLD));
        MPICHECK(MPI_Gather(U[3*N/4][0], halfNumElements, MPI_DOUBLE,
   recvbuffMPI + numElements, halfNumElements, MPI_DOUBLE, 0,
   MPI_COMM_WORLD));

        // extract data from recvbuff
        if (myRank == 0) {
            memcpy(U[N/2][0],recvbuffMPI+halfNumElements,halfNumElements*
   sizeof(double));
            memcpy(U[3*N/4][0],recvbuffMPI+3*halfNumElements,halfNumElements
   *sizeof(double));
        }
```

```
    }
    else {
        int quartNumElements = numElements/4;
        MPICHECK(MPI_Gather(U[N/2][0], quartNumElements, MPI_DOUBLE,
    recvbuffMPI, quartNumElements, MPI_DOUBLE, 0, MPI_COMM_WORLD));
        MPICHECK(MPI_Gather(U[5*N/8][0], quartNumElements, MPI_DOUBLE,
    recvbuffMPI + 2*quartNumElements, quartNumElements, MPI_DOUBLE, 0,
    MPI_COMM_WORLD));
        MPICHECK(MPI_Gather(U[6*N/8][0], quartNumElements, MPI_DOUBLE,
    recvbuffMPI + 4*quartNumElements, quartNumElements, MPI_DOUBLE, 0,
    MPI_COMM_WORLD));
        MPICHECK(MPI_Gather(U[7*N/8][0], quartNumElements, MPI_DOUBLE,
    recvbuffMPI + 6*quartNumElements, quartNumElements, MPI_DOUBLE, 0,
    MPI_COMM_WORLD));

        // extract data from recvbuff
        if (myRank == 0) {
            memcpy(U[N/2][0],recvbuffMPI+quartNumElements,quartNumElements*
    sizeof(double));
            memcpy(U[5*N/8][0],recvbuffMPI+3*quartNumElements,
    quartNumElements*sizeof(double));
            memcpy(U[6*N/8][0],recvbuffMPI+5*quartNumElements,
    quartNumElements*sizeof(double));
            memcpy(U[7*N/8][0],recvbuffMPI+7*quartNumElements,
    quartNumElements*sizeof(double));
        }
    }

    MPICHECK(MPI_Barrier(MPI_COMM_WORLD));
    *dt += omp_get_wtime()-dt2;


    MPICHECK(MPI_Barrier(MPI_COMM_WORLD));
    double dt_last = omp_get_wtime();

    // free device memory
    for (int i = 0; i < 2; i++) {
        cudaSetDevice(i);
        free_3d_cuda(U_d[i]);
        free_3d_cuda(V_d[i]);
    }

    //freeing device memory
    for (int i=0; i<nDev; i++) {
```

```
        CUDACHECK(cudaFree(sendbuff[i]));
        CUDACHECK(cudaFree(recvbuff[i]));
    }

    //finalizing NCCL
    for (int i=0; i<nDev; i++) {
        ncclCommDestroy(comms[i]);
    }

    MPICHECK(MPI_Barrier(MPI_COMM_WORLD));
    *dt += omp_get_wtime()-dt_last;
    *wt = omp_get_wtime()-start;

    //finalizing MPI
    MPICHECK(MPI_Finalize());


    // result
    if (output_type == 4) {
    if (myRank == 0) {
        output_ext = ".vtk";
        sprintf(output_filename, "%s_%d_%s%s", output_prefix, N,
   output_suffix, output_ext);
        fprintf(stderr, "Write␣VTK␣file␣to␣%s:␣", output_filename);
        print_vtk(output_filename, N, U);
    }
    }
    else if (output_type == 1) {
    if (myRank == 0) {
        // print data
        double memory = (3.0*8.0*N*N*N)/pow(10.0,6);
        double bandwidth = (1.0/(*wt))*((24.0*N*N*N)/pow(10.0,6));
        double bandwidth_ndt = (1.0/(*wt-*dt))*((24.0*N*N*N)/pow(10.0,6));
        printf("%d␣%f␣%f␣%f␣%f␣%f\n",N,*wt,*dt,memory,bandwidth,
   bandwidth_ndt);
    }
    }
}
```