Introduction to
Parallel and Distributed Programming

CS 16
CUDA, MPI & OpenMP

Saikishor Jangiti

MPI programs with CUDA

Structure of MPI Programs with CUDA

1. MPI Initialization
    - Initialize the MPI environment and get the rank and size of the communicator.
2. CUDA Initialization
    - Initialize the CUDA environment and allocate device memory.
3. Data Distribution: Distribute the data among the MPI processes.
4. CUDA Kernel Launch: Launch the CUDA kernel on each MPI process.
5. Data Collection: Collect the results from each MPI process.
6. MPI Finalization: Finalize the MPI environment.

Matrix multiplication MPI program with CUDA

```
:#include <mpi.h>
#include <cuda_runtime.h>
// CUDA kernel function
__global__ void matMulKernel(float *A, float *B, float *C, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
        if (idx < N) {
                                        float sum = 0.0f;
                                      for (int i = 0; i < N; i++) {
                                      sum += A[idx * N + i] * B[i * N + idx];        }
      C[idx] = sum;    }
}
```

Matrix multiplication MPI program with CUDA

```
int main(int argc, char **argv) {
  // Initialize MPI environment
  MPI_Init(&argc, &argv);
 int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
 // Initialize CUDA environment
cudaDeviceProp prop;
 cudaGetDeviceProperties(&prop, 0);
 // Define matrix dimensions
  int N = 1024;
    // Allocate host memory
   float *A, *B, *C;
A = (float *)malloc(N * N * sizeof(float));
   B = (float *)malloc(N * N * sizeof(float));
  C = (float *)malloc(N * N * sizeof(float));
 // Allocate device memory
float *d_A, *d_B, *d_C;
 cudaMalloc((void **)&d_A, N * N * sizeof(float));
cudaMalloc((void **)&d_B, N * N * sizeof(float));
cudaMalloc((void **)&d_C, N * N * sizeof(float));
```

Matrix multiplication MPI program with CUDA

```
// Initialize matrices
 for (int i = 0; i < N; i++) {
     for (int j = 0; j < N; j++) {
            A[i * N + j] = i + j;
            B[i * N + j] = i * j;        }
    }
 // Distribute matrices among MPI processes
 int chunkSize = N / size;
  float *A_chunk, *B_chunk;
A_chunk = (float *)malloc(chunkSize * N * sizeof(float));
B_chunk = (float *)malloc(chunkSize * N * sizeof(float));
MPI_Scatter(A, chunkSize * N, MPI_FLOAT, A_chunk, chunkSize * N, MPI_FLOAT, 0,
MPI_COMM_WORLD);
 MPI_Scatter(B, chunkSize * N, MPI_FLOAT, B_chunk, chunkSize * N, MPI_FLOAT, 0,
MPI_COMM_WORLD);
```

Matrix multiplication MPI program with CUDA

```
// Launch CUDA kernel
  int blockSize = 256;
 int numBlocks = (chunkSize * N + blockSize - 1) / blockSize;
  matMulKernel<<<numBlocks, blockSize>>>(d_A, d_B, d_C, chunkSize * N);
 // Collect results
  float *C_chunk;
 C_chunk = (float *)malloc(chunkSize * N * sizeof(float));
   MPI_Gather(d_C, chunkSize * N, MPI_FLOAT, C_chunk, chunkSize * N, MPI_FLOAT, 0, MPI_COMM_WORLD);
  // Finalize MPI environment
  MPI_Finalize();
  return 0;}
```

OpenMP with CUDA

Combining OpenMP and CUDA allows developers to leverage the strengths of both platforms:

Benefits of Using OpenMP with CUDA

1. Hybrid Parallelism: Combine the parallelism of OpenMP on the host (CPU) with the parallelism of CUDA on the device (GPU).
2. Improved Performance: By utilizing both the CPU and GPU, developers can achieve significant performance improvements.
3. Simplified Programming: OpenMP and CUDA provide high-level programming models, making it easier to develop parallel applications.

OpenMP with CUDA

```
#include <omp.h>
#include <cuda_runtime.h>
// Matrix multiplication kernel
__global__ void matMulKernel(float *A, float *B, float *C, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
      if (idx < N) {        float sum = 0.0f;
      for (int i = 0; i < N; i++) {
         sum += A[idx * N + i] * B[i * N + idx];
      }       C[idx] = sum;
   }
}
```

OpenMP with CUDA

```
int main() {
   int N = 1024;
 float *A, *B, *C;
 float *d_A, *d_B, *d_C;
   // Allocate memory on host
 A = (float *)malloc(N * N * sizeof(float));
  B = (float *)malloc(N * N * sizeof(float));
C = (float *)malloc(N * N * sizeof(float));
// Allocate memory on device
   cudaMalloc((void **)&d_A, N * N * sizeof(float));
  cudaMalloc((void **)&d_B, N * N * sizeof(float));
   cudaMalloc((void **)&d_C, N * N * sizeof(float));
```

OpenMP with CUDA

```
// Initialize data on host
 for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
         A[i * N + j] = i + j;
     B[i * N + j] = i * j;        }    }
 // Transfer data from host to device
   cudaMemcpy(d_A, A, N * N * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, B, N * N * sizeof(float), cudaMemcpyHostToDevice);
```

OpenMP with CUDA

```
// Launch kernel with OpenMP parallelism
  #pragma omp parallel
  {       int numBlocks = (N + 255) / 256;
        int blockSize = 256;
  matMulKernel<<<numBlocks, blockSize>>>(d_A, d_B, d_C, N);    }
 // Transfer data from device to host
  cudaMemcpy(C, d_C, N * N * sizeof(float), cudaMemcpyDeviceToHost);
```

OpenMP with CUDA

```
// Print result
for (int i = 0; i < N; i++) {
for (int j = 0; j < N; j++) {
      printf("%f ", C[i * N + j]);       }
      printf("\n");    }
  // Deallocate memory
 free(A);    free(B);    free(C);
  cudaFree(d_A);    cudaFree(d_B);    cudaFree(d_C);    return 0;}
```