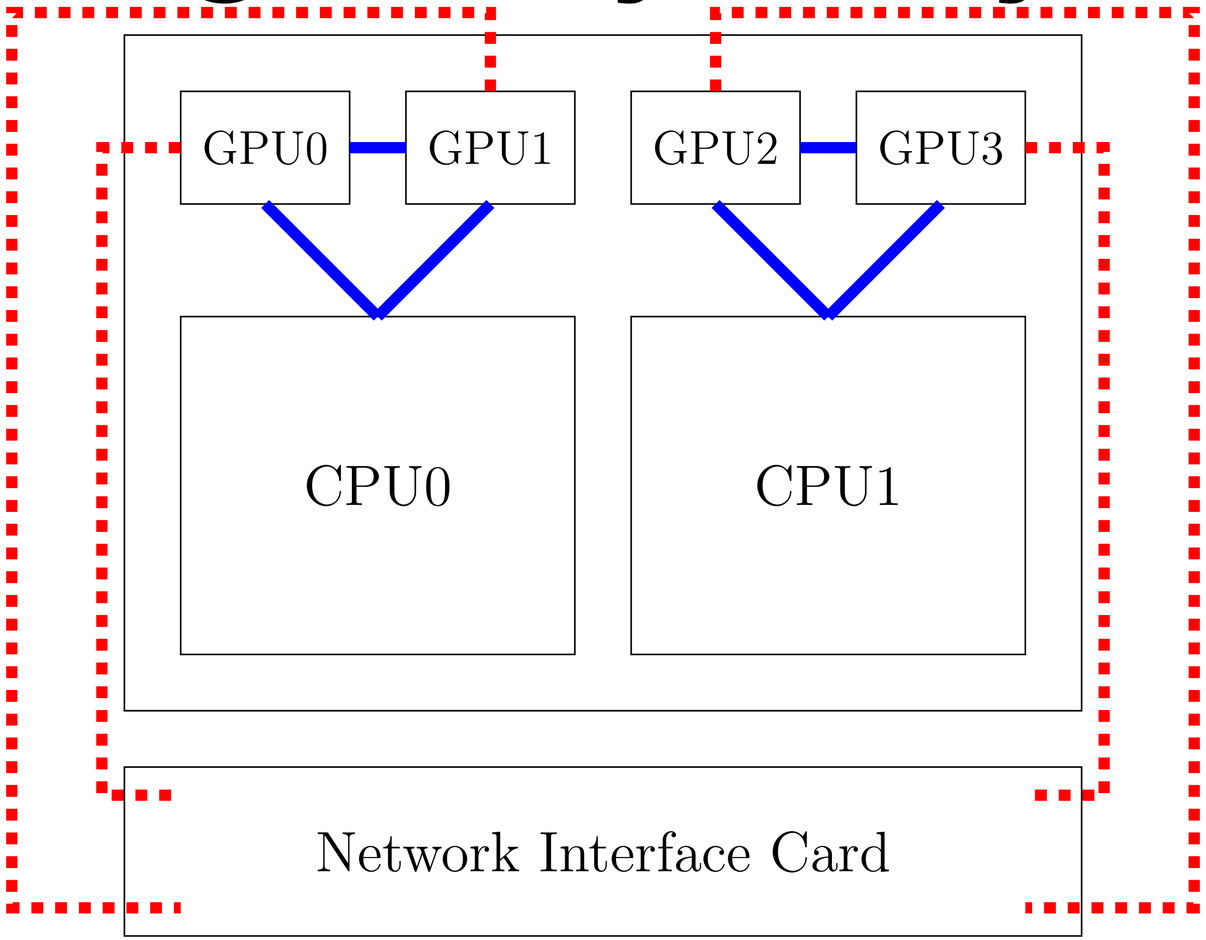
Introduction to Parallel Processing

Lecture 28 : OpenMP Offloading

Professor Amanda Bienz

Heterogeneity in Systems



Want to use MPI between nodes, but can further optimize among processes within a node!

OpenMP Offloading

- OpenMP parallelizes for loops or blocks of code, by splitting across multiple threads
- Similar to accelerators (CUDA) where tasks are split across many many threads on the GPU
- Recent OpenMP versions support GPUs:
  - Write OpenMP code
  - Behind the scenes, data is moved to the GPU, threads are executed on the GPU, and data is copied back to the CPU

Matrix-Matrix Multiplication

- Let's look back at our example of matrix-matrix multiplication with OpenMP
- Each of our outermost loops (both setting values and multiplying) has `#pragma omp parallel`
- Remember, we need to declare variables as private

Moving to the GPU

- `#pragma omp target` : instructs compiler to map variables to device data and execute enclosed block of code on GPU
- Map : which variables need to be copied between CPU and GPU (each of the matrices)
  - Takes a direction
    - to : copy original values from CPU to GPU
    - from : copy final values from GPU to CPU
    - tofrom : copy both directions
    - alloc : data is allocated on device
  - Must use array notation for pointers : `map(to:A[0:N])`

Matrix-Matrix Multiplication

```
void test_omp_gpu(int n, double* A, double* B, double* C, int n_iter)
{
    double val;
    int size = n*n;
    for (int iter = 0; iter < n_iter; iter++)
    {
        #pragma omp target map(tofrom:A[0:size], B[0:size], C[0:size])
        {
            #pragma omp parallel for private(val)
            for (int i = 0; i < n; i++)
            {
                for (int j = 0; j < n; j++)
                {
                    val = 0;
                    for (int k = 0; k < n; k++)
                    {
                        val += A[i*n+k] * B[k*n+j];
                    }
                    C[i*n+j] = val;
                }
            }
        }
    }
}
```

Matrix-Matrix Multiplication

```
void test_omp_gpu_tofrom(int n, double* A, double* B, double* C, int n_iter
{
    double val;
    int size = n*n;
    for (int iter = 0; iter < n_iter; iter++)
    {
        #pragma omp target map(to:A[0:size], B[0:size]) map(from:C[0:size])
        {
            #pragma omp parallel for private(val)
            for (int i = 0; i < n; i++)
            {
                for (int j = 0; j < n; j++)
                {
                    val = 0;
                    for (int k = 0; k < n; k++)
                    {
                        val += A[i*n+k] * B[k*n+j];
                    }
                    C[i*n+j] = val;
                }
            }
        }
    }
}
```

Matrix-Matrix Multiplication Times

- 1000x1000 Matrix Timings :
- Serial : 1.74e00 seconds
- OpenMP Threads (CPU) : 1.12e-01 seconds
- OpenMP Offloading : 1.12e01 seconds

OpenMP Target Data Only

- `#pragma omp target data`
  - Offloads data from CPU to GPU, but does not execute
  - Manually specify which executions should happen on the GPU and everything else will happen on the CPU

OpenMP Teams

- `#pragma omp target teams num_teams(n)`: Create n teams (or groups of threads)
- `#pragma omp distribute parallel for :` Distribute iterations to teams of threads
- Ideally want to parallelize within the teams : call parallel for loop within each team (inner loop)

Matrix-Matrix Multiplication

```
void test_omp_gpu_teams(int n, double* A, double* B, double* C, int n_iter)
{
    double val;
    int size = n*n;
    for (int iter = 0; iter < n_iter; iter++)
    {
        #pragma omp target data map(to:A[0:size], B[0:size]) map(from:C[0:si
        {
            #pragma omp target teams num_teams(n)
            {
                #pragma omp distribute parallel for private(val)
                for (int i = 0; i < n; i++)
                {
                    for (int j = 0; j < n; j++)
                    {
                        val = 0;
                        for (int k = 0; k < n; k++)
                        {
                            val += A[i*n+k] * B[k*n+j];
                        }
                        C[i*n+j] = val;
                    }
                }
            }
        }
    }
```

Matrix-Matrix Multiplication

```
}

void test_omp_gpu_split_rr(int n, double* A, double* B, double* C, int n_ite
{
    double val;
    int size = n*n;
    for (int iter = 0; iter < n_iter; iter++)
    {
        #pragma omp target data map(to:A[0:size], B[0:size]) map(from:C[0:si
        {
            #pragma omp target teams num_teams(n)
            {
                #pragma omp distribute
                for (int i = 0; i < n; i++)
                {
                    #pragma omp parallel for private(val)
                    for (int j = 0; j < n; j++)
                    {
                        val = 0;
                        for (int k = 0; k < n; k++)
                        {
                            val += A[i*n+k] * B[k*n+j];
                        }
                        C[i*n+j] = val;
                    }
                }
            }
        }
```

Collapse

- Add “collapse(n)” to your pragma statement to combine n of the loops into one
- Distribute then splits up all iterations between teams

Matrix-Matrix Multiplication

```
void test_omp_gpu_collapse(int n, double* A, double* B, double* C, int n_iter)
{
    double val;
    int size = n*n;
    for (int iter = 0; iter < n_iter; iter++)
    {
        #pragma omp target data map(to:A[0:size]) map(to:B[0:size]) map(from:C[0:size])
        {
            #pragma omp target
            #pragma omp teams distribute parallel for collapse(2) num_teams(n) thread_limit(n) private(val)
            for (int i = 0; i < n; i++)
            {
                for (int j = 0; j < n; j++)
                {
                    val = 0;
                    for (int k = 0; k < n; k++)
                    {
                        val += A[i*n+k] * B[k*n+j];
                    }
                    C[i*n+j] = val;
                }
            }
        }
    }
}
```

Schedule

- GPUs only support static scheduling
- On CPUs, schedule(static) works best, splitting n/p continuous iterations to each process
- On GPUs, this is probably not what we want (think memory coalescing)
  - Would probably rather have schedule(static,1) to partition iterations in a round-robin style

Matrix-Matrix Multiplication

```
void test_omp_gpu_collapse_rr(int n, double* A, double* B, double* C, int n_iter)
{
    double val;
    int size = n*n;
    for (int iter = 0; iter < n_iter; iter++)
    {
        #pragma omp target data map(to:A[0:size]) map(to:B[0:size]) map(from:C[0:size])
        {
            #pragma omp target
            #pragma omp teams distribute parallel for collapse(2) num_teams(n) thread_limit(n) private(val) schedule(static,1)
            for (int i = 0; i < n; i++)
            {
                for (int j = 0; j < n; j++)
                {
                    val = 0;
                    for (int k = 0; k < n; k++)
                    {
                        val += A[i*n+k] * B[k*n+j];
                    }
                    C[i*n+j] = val;
                }
            }
        }
    }
}
```

Optimized Matrix-Matrix Multiplication

Results for 1000x1000 matrices :

- Serial: Sum 1.439273e+04, Time Per MatMat 3.082108e+00
- CPU OMP: Sum 1.439273e+04, Time Per MatMat 4.663191e-01
- GPU OMP Teams: Sum 1.439273e+04, Time Per MatMat 6.970195e+00
- GPU OMP Split: Sum 1.439273e+04, Time Per MatMat 4.011209e+00
- GPU OMP Collapse: Sum 1.439273e+04, Time Per MatMat 2.851319e+00
- GPU OMP Split RR: Sum 1.439273e+04, Time Per MatMat 3.737256e+00
- GPU OMP Collapse RR: Sum 1.439273e+04, Time Per MatMat 2.936583e+00

Optimized Matrix-Matrix Multiplication

Results for 1000x1000 matrices :

- Serial: Sum 4.056327e+04, Time Per MatMat 9.425226e+01
- CPU OMP: Sum 4.056327e+04, Time Per MatMat 1.295492e+01
- GPU OMP Teams: Sum 4.056327e+04, Time Per MatMat 6.763105e+01
- GPU OMP Split: Sum 4.056327e+04, Time Per MatMat 5.731349e+01
- GPU OMP Collapse: Sum 4.056327e+04, Time Per MatMat 4.445983e+01
- GPU OMP Split RR: Sum 4.056327e+04, Time Per MatMat 5.629621e+01
- GPU OMP Collapse RR: Sum 4.056327e+04, Time Per MatMat 4.592058e+01

Reduction

- Reduction does what you would expect (same as OpenMP conceptually)
- However, the variable that is being reduced must appear in a map(to from) clause to get the value back at the end

If you want to learn more

- There are a LOT of available methods, can manually allocate data on device, specify to use shared memory, etc

- https://www.nersc.gov/assets/Uploads/OLCF-OMP-Day2.pdf