STUNIR ROCm Multi-GPU Programming Guide

Guide for developing multi-GPU applications with ROCm/HIP.

Overview

Multi-GPU programming enables:
- **Scaling**: Process larger datasets
- **Performance**: Parallel computation across GPUs
- **Memory**: Combined GPU memory pool

Architecture Patterns

1. Data Parallelism

Split data across GPUs, each runs same kernel:

```
Data: [D0 | D1 | D2 | D3]
        |    |    |    |
      GPU0 GPU1 GPU2 GPU3
        |    |    |    |
Result: [R0 | R1 | R2 | R3]
```

2. Model Parallelism

Split model across GPUs:

```
Input → [GPU0: Layer1] → [GPU1: Layer2] → [GPU2: Layer3] → Output
```

3. Pipeline Parallelism

Overlap computation stages:

```
Time:    T0     T1     T2     T3
GPU0:   [Batch1][Batch2][Batch3]
GPU1:          [Batch1][Batch2][Batch3]
GPU2:                 [Batch1][Batch2][Batch3]
```

Device Management

Enumerate Devices

```
#include "multi_gpu_utils.hip"
using namespace stunir::rocm::multigpu;

DeviceManager mgr;
printf("Found %d GPUs\n", mgr.device_count());

for (int i = 0; i < mgr.device_count(); i++) {
    mgr.device(i).print();
}
```

Switch Devices

```
// Current thread's device
hipSetDevice(0);

// Allocate on GPU 0
float* d_data0;
hipMalloc(&d_data0, size);

// Switch and allocate on GPU 1
hipSetDevice(1);
float* d_data1;
hipMalloc(&d_data1, size);
```

Peer-to-Peer (P2P) Memory Access

Enable P2P

```
// Check P2P capability
int can_access;
hipDeviceCanAccessPeer(&can_access, 0, 1);  // Can GPU0 access GPU1?

if (can_access) {
    hipSetDevice(0);
    hipDeviceEnablePeerAccess(1, 0);  // GPU0 can now access GPU1 memory

    hipSetDevice(1);
    hipDeviceEnablePeerAccess(0, 0);  // GPU1 can now access GPU0 memory
}
```

Direct P2P Copy

```
// Copy between GPUs without going through host
hipMemcpyPeer(d_data1, 1,    // Destination: GPU1
              d_data0, 0,    // Source: GPU0
              size);
```

Direct Kernel Access

```
// Kernel on GPU0 can directly access GPU1 memory (if P2P enabled)
__global__ void kernel_with_p2p(float* local, float* remote) {
    int idx = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x;
    // Read from remote GPU
    local[idx] = remote[idx] * 2.0f;
}
```

Stream-Based Concurrency

Async Operations

```
hipStream_t stream0, stream1;
hipSetDevice(0); hipStreamCreate(&stream0);
hipSetDevice(1); hipStreamCreate(&stream1);

// Launch async operations on both GPUs
hipSetDevice(0);
hipMemcpyAsync(d_data0, h_data, size/2, hipMemcpyHostToDevice, stream0);
kernel<<<grid, block, 0, stream0>>>(d_data0);

hipSetDevice(1);
hipMemcpyAsync(d_data1, h_data + size/2, size/2, hipMemcpyHostToDevice, stream1);
kernel<<<grid, block, 0, stream1>>>(d_data1);

// Synchronize both
hipStreamSynchronize(stream0);
hipStreamSynchronize(stream1);
```

Events for Synchronization

```
hipEvent_t event0;
hipSetDevice(0);
hipEventCreate(&event0);

// GPU0 signals completion
kernel<<<grid, block, 0, stream0>>>(...);
hipEventRecord(event0, stream0);

// GPU1 waits for GPU0
hipSetDevice(1);
hipStreamWaitEvent(stream1, event0, 0);
kernel<<<grid, block, 0, stream1>>>(...);  // Starts after event0
```

Data Distribution

Block Distribution

```
DataDistributor<float> dist(num_gpus, DistributionStrategy::BLOCK);
auto sizes = dist.partition(total_elements);

// sizes[0] = total/4, sizes[1] = total/4, ...
```

Custom Distribution

```cpp
// Weight-based distribution
std::vector<size_t> sizes(num_gpus);
size_t total_weight = 0;
for (int i = 0; i < num_gpus; i++) {
    total_weight += gpu_capabilities[i];
}

for (int i = 0; i < num_gpus; i++) {
    sizes[i] = total_elements * gpu_capabilities[i] / total_weight;
}
```

Example: Multi-GPU Matrix Multiplication

```cpp
void multi_gpu_matmul(int M, int N, int K,
                      const float* A, const float* B, float* C,
                      DeviceManager& mgr) {
    int num_gpus = mgr.device_count();
    int rows_per_gpu = M / num_gpus;

    std::vector<float*> d_A(num_gpus), d_B(num_gpus), d_C(num_gpus);
    std::vector<hipStream_t> streams(num_gpus);

    // Allocate and copy per GPU
    for (int g = 0; g < num_gpus; g++) {
        hipSetDevice(g);
        hipStreamCreate(&streams[g]);

        int rows = (g == num_gpus - 1) ? M - g * rows_per_gpu : rows_per_gpu;
        hipMalloc(&d_A[g], rows * K * sizeof(float));
        hipMalloc(&d_B[g], K * N * sizeof(float));
        hipMalloc(&d_C[g], rows * N * sizeof(float));

        // Copy A partition and full B
        hipMemcpyAsync(d_A[g], A + g * rows_per_gpu * K,
                       rows * K * sizeof(float), hipMemcpyHostToDevice, streams[g]);
        hipMemcpyAsync(d_B[g], B, K * N * sizeof(float),
                       hipMemcpyHostToDevice, streams[g]);
    }

    // Launch kernels
    for (int g = 0; g < num_gpus; g++) {
        hipSetDevice(g);
        int rows = (g == num_gpus - 1) ? M - g * rows_per_gpu : rows_per_gpu;

        dim3 block(16, 16);
        dim3 grid((N + 15) / 16, (rows + 15) / 16);

        matmul_kernel<<<grid, block, 0, streams[g]>>>(
            d_A[g], d_B[g], d_C[g], rows, N, K);
    }

    // Copy results back
    for (int g = 0; g < num_gpus; g++) {
        hipSetDevice(g);
        int rows = (g == num_gpus - 1) ? M - g * rows_per_gpu : rows_per_gpu;

        hipMemcpyAsync(C + g * rows_per_gpu * N, d_C[g],
                       rows * N * sizeof(float), hipMemcpyDeviceToHost, streams[g]);
    }

    // Synchronize and cleanup
    for (int g = 0; g < num_gpus; g++) {
        hipSetDevice(g);
        hipStreamSynchronize(streams[g]);
        hipFree(d_A[g]); hipFree(d_B[g]); hipFree(d_C[g]);
        hipStreamDestroy(streams[g]);
    }
}
```

Performance Considerations

1. PCIe Bandwidth

```
PCIe 4.0 x16: ~32 GB/s bidirectional
PCIe 5.0 x16: ~64 GB/s bidirectional
Infinity Fabric: 200+ GB/s (MI250X)
```

2. Overlap Strategy

```
// Overlap H2D, compute, D2H
for (int batch = 0; batch < num_batches; batch++) {
    // Async H2D for next batch
    if (batch + 1 < num_batches) {
        hipMemcpyAsync(d_input_next, h_input[batch+1], ...);
    }

    // Compute current batch
    kernel<<<grid, block, 0, compute_stream>>>(d_input, d_output);

    // Async D2H for previous batch
    if (batch > 0) {
        hipMemcpyAsync(h_output[batch-1], d_output_prev, ...);
    }

    // Swap buffers
    std::swap(d_input, d_input_next);
    std::swap(d_output, d_output_prev);
}
```

3. Load Balancing

```
// Dynamic load balancing with work stealing
std::atomic<int> work_index(0);

void gpu_worker(int gpu_id, ...) {
    while (true) {
        int work = work_index.fetch_add(1);
        if (work >= total_work) break;

        process_work_item(gpu_id, work);
    }
}
```

Debugging Multi-GPU

Environment Variables

```
# Limit visible GPUs
export HIP_VISIBLE_DEVICES=0,1

# Enable debug output
export AMD_LOG_LEVEL=4

# Trace API calls
export HIP_TRACE_API=1
```

Common Issues

1. **Wrong device context**: Always set device before operations
2. **Missing synchronization**: Use streams/events properly
3. **P2P not enabled**: Check and enable before direct access
4. **Memory on wrong GPU**: Verify allocation device matches kernel device

Scaling Guidelines

GPUs	Use Case	Expected Scaling
2	Small to medium	1.8-1.95x
4	Medium workloads	3.5-3.8x
8	Large workloads	7-7.5x
16+	HPC clusters	Sublinear (communication bound)

Future: NCCL Integration

For collective operations (AllReduce, Broadcast):

```
// Coming in future STUNIR releases
#include <rccl/rccl.h>

ncclComm_t comms[num_gpus];
ncclCommInitAll(comms, num_gpus, ...);

ncclAllReduce(sendbuff, recvbuff, count, ncclFloat, ncclSum, comm, stream);
```

References

- HIP Programming Guide (https://rocm.docs.amd.com/projects/HIP/)
- RCCL Documentation (https://rocm.docs.amd.com/projects/rccl/)
- AMD GPU Architecture (https://rocm.docs.amd.com/en/latest/conceptual/gpu-arch.html)