STUNIR ROCm Performance Tuning Guide

Comprehensive guide for optimizing GPU performance on AMD ROCm platform.

Architecture Overview

AMD GPU Architectures

Architecture	Products	Wavefront	L2 Cache	Features
RDNA3	RX 7000	32/64	96MB	Infinity Cache, AI accelerators
RDNA2	RX 6000	32	128MB	Ray tracing, VRS
CDNA3	MI300	64	256MB	Matrix cores, HBM3 (5.3 TB/s)
CDNA2	MI250X	64	8MB	HBM2e (3.2 TB/s)

Key Metrics

- **Wavefront Size**: 64 threads on compute GPUs (CDNA), 32 on gaming (RDNA)
- **Shared Memory**: 64 KB per compute unit
- **Registers**: 256 VGPRs per work-item
- **Memory Bandwidth**: Up to 5.3 TB/s (MI300X)

Optimization Strategies

1. Memory Optimization

Coalesced Memory Access

```
// Good: Coalesced access (consecutive threads access consecutive memory)
__global__ void coalesced(float* data) {
    int idx = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x;
    data[idx] = data[idx] * 2.0f;  // Threads 0,1,2... access data[0,1,2...]
}

// Bad: Strided access
__global__ void strided(float* data, int stride) {
    int idx = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x;
    data[idx * stride] = ...;  // Memory access pattern scattered
}
```

Shared Memory

```
// Use shared memory to reduce global memory traffic
__global__ void matmul_tiled(const float* A, const float* B, float* C, int N) {
    __shared__ float As[TILE][TILE];
    __shared__ float Bs[TILE][TILE];

    // Load tiles collaboratively
    As[ty][tx] = A[row * N + t * TILE + tx];
    Bs[ty][tx] = B[(t * TILE + ty) * N + col];
    __syncthreads();

    // Compute from shared memory (100x faster than global)
    for (int k = 0; k < TILE; k++) {
        sum += As[ty][k] * Bs[k][tx];
    }
}
```

Memory Pool Usage

```
#include "rocm/memory/memory_pool.hip"
using namespace stunir::rocm::memory;

// Pool allocation is 10-100x faster than hipMalloc
MemoryPool pool;
float* data = pool.allocate_typed<float>(1024);
// ... use data ...
pool.deallocate(data);  // Returns to pool, not freed
```

2. Occupancy Optimization

Register Pressure

```
// Reduce register usage with __launch_bounds__
__global__ __launch_bounds__(256, 4)  // 256 threads, 4 blocks/SM
void optimized_kernel(...) {
    // Compiler optimizes for specified occupancy
}
```

Shared Memory vs Registers

```
// Trade shared memory for registers in tight loops
__global__ void kernel(...) {
    float local_val = 0.0f;  // Register

    #pragma unroll 8
    for (int i = 0; i < 8; i++) {
        local_val += ...;  // Stays in register
    }
}
```

3. Instruction Optimization

Use Fast Math

```
# Compile with fast math for ~20% speedup on FP operations
hipcc -ffast-math -o kernel kernel.hip
```

Intrinsics

```
// Use intrinsics for common operations
float fast_rsqrt = __frsqrt_rn(x);  // Fast reciprocal sqrt
float fast_exp = __expf(x);          // Fast exponential
float fma_result = __fmaf_rn(a, b, c);  // Fused multiply-add
```

Loop Unrolling

```
#pragma unroll 4
for (int i = 0; i < 16; i += 4) {
    // Compiler unrolls 4 iterations
}
```

4. Warp-Level Optimization

Warp Shuffle

```
// Fast warp-level reduction
__device__ float warp_reduce(float val) {
    for (int offset = 32; offset > 0; offset /= 2) {
        val += __shfl_down(val, offset);
    }
    return val;
}
```

Avoid Warp Divergence

```
// Bad: Divergent branches
if (tid % 2 == 0) {
    do_work_a();
} else {
    do_work_b();
}

// Good: Predication or separate kernels
float result = (tid % 2 == 0) ? compute_a() : compute_b();
```

5. hipBLAS/hipSPARSE Tuning

Tensor Cores (MI100+)

```
// Use tensor core GEMM for matrix operations
hipblasGemmEx(handle, HIPBLAS_OP_N, HIPBLAS_OP_N,
              m, n, k, &alpha,
              A, HIP_R_16F, lda,  // FP16 input
              B, HIP_R_16F, ldb,
              &beta, C, HIP_R_32F, ldc,  // FP32 output
              HIPBLAS_COMPUTE_32F,  // Compute in FP32
              HIPBLAS_GEMM_DEFAULT_TENSOR_OP);
```

Batched Operations

```
// Batch small operations for better utilization
hipblasSgemmBatched(handle, ..., batch_count);
// Or strided batched for contiguous data
hipblasSgemmStridedBatched(handle, ..., strideA, strideB, strideC, batch_count);
```

Profiling Tools

rocprof

```
# Basic profiling
rocprof --stats ./my_app

# Detailed metrics
rocprof -i metrics.txt ./my_app

# Timeline trace
rocprof --hip-trace ./my_app
```

Omniperf (ROCm 5.3+)

```
# Comprehensive analysis
omniperf profile -n my_run -- ./my_app
omniperf analyze -p workloads/my_run/
```

rocm-smi

```
# Monitor GPU utilization
watch -n 0.5 rocm-smi

# Power and temperature
rocm-smi --showpower --showtemp
```

Performance Checklist

- [ ] Memory access is coalesced
- [ ] Shared memory used for data reuse
- [ ] Occupancy is sufficient (> 50%)

- [ ] No excessive register spilling
- [ ] Warp divergence minimized
- [ ] Memory pool for frequent allocations
- [ ] hipBLAS for dense linear algebra
- [ ] hipSPARSE for sparse operations
- [ ] Async operations overlap compute/transfer
- [ ] Multi-GPU for large workloads

Expected Performance

Operation	MI250X	MI300X	Notes
SGEMM (peak)	47.9 TFLOPS	122 TFLOPS	FP32
HGEMM (peak)	383 TFLOPS	980 TFLOPS	FP16
Memory BW	3.2 TB/s	5.3 TB/s	HBM
SpMV	~50 GB/s	~80 GB/s	Depends on sparsity

References

- AMD ROCm Documentation (https://rocm.docs.amd.com/)
- HIP Programming Guide (https://rocm.docs.amd.com/projects/HIP/)
- Omniperf User Guide (https://rocm.docs.amd.com/projects/omniperf/)
- rocprof User Guide (https://rocm.docs.amd.com/projects/rocprofiler/)