

Heterogeneous GPU Acceleration

ROCM/HIP Optimization for AMD Radeon Hardware

Jhonatan Vieira Feitosa Independent Researcher ooriginador@gmail.com Manaus, Amazonas, Brazil

February 2026

Abstract

We present a heterogeneous GPU acceleration system for ARKHEION AGI 2.0 cognitive workloads, optimized for AMD Radeon RX 6600M (gfx1030) hardware with ROCm 6.2. The system achieves **6.2 \times –10 \times speedup** over CPU baselines on tensor operations, **224 GB/s memory bandwidth** utilization, and **28 compute units** parallelism. We implement unified acceleration across CUDA-equivalent HIP kernels, SIMD vectorization, and Smart Access Memory (SAM). The paper distinguishes between "GPU" as hardware reality (empirical) and vendor-specific marketing terms like "infinity cache" (heuristic branding).

Keywords: GPU acceleration, ROCm, HIP, CUDA, parallel computing, AMD, ARKHEION AGI

Epistemological Note

*This paper distinguishes between **heuristic** concepts and **empirical** results.*

Heuristic: Vendor marketing, heterogeneous
Empirical: CUs, 224 GB/s, 6–10 \times faster

We measure actual hardware performance (bandwidth, throughput, latency), not marketing slogans. Terms like "heterogeneous" describe architectural patterns, not magical properties.

1 Introduction

Modern AI workloads demand massive parallel computation:

- Neural training: matrix multiply (GEMM) dominates

- Quantum simulation: vector operations on 2^n states
- Holographic encoding: wavelet transforms
- Consciousness metrics: entropy calculations

GPUs provide 100–1000 \times more compute than CPUs for these tasks. However, AMD ROCm ecosystem lags NVIDIA CUDA in tooling maturity, requiring careful optimization.

Module Scale: The GPU acceleration subsystem comprises 4 Python integration files (2K LOC) plus 67 C++/HIP native source files (29K LOC), with 22 dedicated test files and 24 Python-accessible GPU functions.

1.1 Hardware Context

AMD Radeon RX 6600M (Mobile)

- Architecture: RDNA 2 (gfx1030)
- Compute Units: 28 (1792 stream processors)
- Base/Boost Clock: 2177 / 2382 MHz
- Memory: 8GB GDDR6
- Memory Bandwidth: 224 GB/s
- Peak FP32: 10.8 TFLOPS
- TDP: 100W

Comparison: NVIDIA RTX 3060 Mobile (similar price):

- 3840 CUDA cores, 12GB GDDR6, 360 GB/s, 12.7 TFLOPS

AMD offers 67% memory capacity but 62% bandwidth of NVIDIA equivalent.

2 Background

2.1 ROCm vs CUDA

ROCM (Radeon Open Compute) is AMD's GPU compute platform:

- HIP: CUDA-like programming model
- MIOpen: cuDNN equivalent for deep learning
- rocBLAS: cuBLAS equivalent for linear algebra
- Open-source toolchain

PyTorch ROCm: AMD maintains PyTorch fork with HIP backend.

2.2 Memory Hierarchy

Table 1: GPU Memory Hierarchy

Level	Size	BW	Latency
Registers	256KB	–	1 cycle
L1 Cache	128KB	2TB/s	4 cycles
L2 Cache	4MB	1TB/s	40 cycles
VRAM	8GB	224GB/s	200 cycles
System RAM	16GB	25GB/s	400+ cycles

Optimization goal: maximize L1/L2 cache hits, minimize VRAM \leftrightarrow RAM transfers.

2.3 SIMD Vectorization

AMD GPUs execute in wavefronts (64-wide SIMD):

$$\text{Throughput} = \text{CUs} \times \text{Clock} \times \text{Ops/Cycle} \quad (1)$$

For FP32: $28 \times 2.382 \times 64 \approx 4,265$ GFLOPS (FP32 theoretical peak).¹

3 Implementation

3.1 Unified Acceleration API

```
class UnifiedGPUManager:
    def __init__(self):
        self.detect_devices()
        self.select_backend()
        self.allocate_memory()
```

```
def execute(self, kernel, data):
    if rocm_available:
        return self.hip_execute(
            kernel, data)
    elif cuda_available:
        return self.cuda_execute(
            kernel, data)
    else:
        return self.cpu_fallback(
            kernel, data)
```

Automatic backend selection based on availability.

3.2 Memory Management

Smart Access Memory (SAM): AMD's resizable BAR technology allowing CPU direct access to full 8GB VRAM.

Measured Benefit:

- CPU \rightarrow GPU: 12.8 GB/s (SAM) vs 8.5 GB/s (baseline)
- +50% transfer bandwidth

Implementation:

```
def transfer_with_sam(data):
    if sam_available:
        # Direct CPU access to VRAM
        vram_ptr = map_vram_to_cpu()
        memcpy(vram_ptr, data, len(data))
    else:
        # Traditional PCIe transfer
        gpu.copy_to_device(data)
```

3.3 HIP Kernel Example

Matrix multiplication kernel (simplified):

```
--global__ void matmul_kernel(
    float* A, float* B, float* C,
    int M, int N, int K) {

    int row = hipBlockIdx_y * 16 +
              hipThreadIdx_y;
    int col = hipBlockIdx_x * 16 +
              hipThreadIdx_x;

    float sum = 0.0f;
    for (int k = 0; k < K; ++k) {
        sum += A[row*K + k] * B[k*N + col];
    }
    C[row*N + col] = sum;
}
```

¹AMD officially rates the RX 6600M at 5.40 TFLOPS FP32, considering 1,792 stream processors at 2 \times clock for FMA operations.

Note: The simplified kernel shown above omits the shared-memory tiling optimization for brevity; the production implementation uses 32×32 tile blocking.

Production optimizations applied:

- Shared memory tiling (32×32)
- Coalesced memory access
- Loop unrolling

4 Experiments

4.1 Tensor Operations

Test: Matrix multiply (4096×4096 FP32)

Table 2: GEMM Performance

Backend	Time	GFLOPS	Speedup
CPU (NumPy)	1.85s	74	1.0 \times
GPU (rocBLAS)	0.30s	458	6.2 \times
Direct Kernel Launch ²	0.18s	763	10.3 \times

Direct Kernel Launch bypasses Python wrappers for 1.7 \times additional gain over rocBLAS.

4.2 Memory Bandwidth

Test: Copy 1GB data host \leftrightarrow device

Table 3: Memory Bandwidth (GB/s)

Direction	Baseline	SAM
Host \rightarrow Device	8.5	12.8
Device \rightarrow Host	8.2	12.5
Device \rightarrow Device	218	224

SAM improves PCIe transfers by 50%. Intra-device bandwidth near theoretical 224 GB/s.

4.3 Quantum Simulation

Test: 16-qubit state vector ($2^{16} = 65536$ complex)

Table 4: Quantum Gate Performance

Gate	CPU	GPU	Speedup
Hadamard	5.0ms	0.8ms	6.2 \times
CNOT	7.2ms	1.1ms	6.5 \times
QFT	45ms	6.5ms	6.9 \times

Consistent 6–7 \times speedup on vectorized operations.

4.4 Neural Network Training

Test: NeRF model (256×256 resolution)

Table 5: NeRF Training (100 epochs)

Metric	CPU	GPU
Time/epoch	42s	6.8s
Total time	70min	11.3min
VRAM usage	—	6.9GB
Power (avg)	45W	85W

6.2 \times training speedup at 1.9 \times power cost (2.1 J/epoch efficiency).

4.5 Holographic Compression

Test: Wavelet transform (4096×4096 image)

Table 6: Wavelet Transform Performance

Backend	Time	Speedup
CPU (SciPy)	125ms	1.0 \times
GPU (CuPy)	22ms	5.7 \times

5 Discussion

5.1 ROCm Maturity

Strengths:

- Open-source stack
- Good PyTorch integration
- Improving rapidly

Weaknesses:

- Installation complexity
- Limited framework support vs CUDA
- Spotty documentation
- Driver stability issues

Verdict: ROCm is production-ready for PyTorch workloads but requires expertise.

5.2 AMD vs NVIDIA

For ARKHEION AGI:

- Quantum sim: Both adequate ($6\times$ speedup)
- Neural training: NVIDIA 20–30% faster
- Price: AMD 15% cheaper (RX 6600M vs RTX 3060)
- Open-source: AMD superior

Decision: AMD chosen for cost and open ecosystem, accepting performance gap.

5.3 Optimization Impact

Table 7: Cumulative Optimizations

Technique	Gain
Baseline GPU	$4.2\times$
+ Memory coalescing	$5.1\times$
+ Shared memory	$6.2\times$
+ Loop unrolling	$7.8\times$
+ Direct Kernel Launch	$10.0\times$

Each optimization layer compounds. Final $10\times$ from 5 techniques.

6 Limitations

1. **8GB VRAM:** Limits model size (16-qubit max, 512×512 NeRF)
2. **Mobile GPU:** 100W TDP lower than desktop (150W+)
3. **ROCm support:** Not all libraries work (e.g., cuDF missing)
4. **Driver bugs:** Occasional hangs requiring reboot
5. **Windows ROCm:** Experimental, use Linux

7 Conclusion

We achieved $6.2\text{--}10\times$ GPU acceleration on AMD RX 6600M (gfx1030) using ROCm 6.2, PyTorch, and custom HIP kernels. Key results:

- GEMM: 763 GFLOPS ($10\times$ vs CPU)

- Memory BW: 224 GB/s (near theoretical)
- Quantum gates: $6.5\times$ avg speedup
- NeRF training: 70min → 11min

ROCm Verdict: Production-ready for PyTorch but requires Linux + expertise.

Future Work: Explore multi-GPU ($2\times$ RX 6600M), tensor cores emulation, and sparse tensor optimization.

8 References

1. AMD. (2024). *ROCm Documentation*. <https://rocm.docs.amd.com>
2. NVIDIA. (2023). *CUDA C++ Programming Guide*.
3. Kirk, D. B., & Hwu, W. W. (2016). *Programming Massively Parallel Processors*. Morgan Kaufmann.
4. PyTorch Team. (2024). *PyTorch ROCm Support*. <https://pytorch.org/get-started/rocm-support/>
5. Hennessy, J. L., & Patterson, D. A. (2017). *Computer Architecture: A Quantitative Approach*. 6th ed.