CUDA Bitcoin Mining Implementation Report

Chin-Hui Chu r12944041

PART 1: IMPLEMENTATION

Overview

This project implements a GPU-accelerated Bitcoin block mining system using CUDA. The goal is to parallelize the sequential CPU-based nonce search algorithm to achieve significant speedup on NVIDIA Tesla V100 GPUs.

Core Algorithm

Bitcoin mining involves finding a nonce value such that the double SHA-256 hash of a block header is below a target difficulty threshold:

```
Block Header (80 bytes) = version(4) + prevhash(32) +
merkle_root(32) + ntime(4) + nbits(4) + nonce(4)
Valid Block: SHA256(SHA256(Block Header)) < Target
```

The sequential CPU approach tests nonces one at a time (0 to 2^32-1), which can take hours for difficult blocks. Our GPU implementation parallelizes this search across thousands of threads simultaneously.

System Configuration

- **GPU**: NVIDIA Tesla V100 (Volta architecture, sm_70)
- **CUDA Version**: 11.x with C++11
- **Compiler Flags**: `-O3 -arch=sm_70 -rdc=true`
- **Execution Environment**: SLURM-managed HPC cluster

Implementation Approach

CPU (Host) Responsibilities:

1. Read block data from input file
2. Calculate merkle root from transaction branches
3. Construct block header and calculate target value
4. Copy constant data to GPU (target, midstate, constant bytes)
5. Launch GPU kernel to search for valid nonce

6. Retrieve and verify results

GPU (Device) Responsibilities:

1. Each thread independently tests different nonce values
2. Compute double SHA-256 hash for each nonce
3. Compare hash against target difficulty
4. Use atomic operations to report valid nonce
5. Terminate early when solution found

Key Components

Data Structures:

- `HashBlock` - 80-byte Bitcoin block header
- `SHA256` - SHA-256 hash state (32 bytes)
- Constant memory for read-only data (target, midstate)
- Shared memory for per-block caching

Main Functions:

- `mine_kernel()` - GPU kernel for parallel nonce search
- `solve()` - Orchestrates CPU-GPU workflow
- `calc_merkle_root()` - Computes merkle tree root
- `double_sha256()` - Double SHA-256 hashing
- `little_endian_bit_comparison()` - Compares hashes

PART 2: PARALLELIZATION AND OPTIMIZATION TECHNIQUES

1. GPU Kernel Parallelization

Implemented grid-stride loop pattern where each thread tests different nonce values independently.

- **Benefit**: Scales to full 2^32 nonce space with any thread count, eliminates coordination overhead

2. SHA-256 Midstate Optimization

Pre-compute SHA-256 state for first 64 bytes of block header (only nonce changes in last 4 bytes).

- **Benefit**: Reduces per-nonce computation by ~40%

3. Constant Memory for Read-Only Data

Store target value, midstate, and constant block bytes in constant memory.

- **Benefit**: Broadcast mechanism serves entire warp with single read, dedicated cache

4. Shared Memory Caching

Thread blocks cooperatively load constant memory into shared memory for faster repeated access.

- **Benefit**: 100× faster than global memory, reduces constant memory bandwidth pressure

5. Warp-Level Early Termination

Use `__shfl_sync()` warp shuffle to broadcast termination flag efficiently.

- **Benefit**: 32× reduction in global memory traffic (1 read per warp instead of 32)

6. Atomic Operations for Result Reporting

Use `atomicMin()` to safely record the smallest valid nonce when multiple threads find solutions.

- **Benefit**: No race conditions, guaranteed correctness with parallel discoveries

7. Optimized Hash Comparison

64-bit word-level comparison instead of byte-by-byte comparison with loop unrolling.

- **Benefit**: 8× fewer iterations (4 words vs 32 bytes)

8. CUDA Streams for Multi-Block Pipelining

Double-buffering with 2 streams to overlap GPU kernel execution with CPU preparation.

- **Benefit**: GPU mines block N while CPU prepares block N+1, eliminates idle time

9. Fast Hex Parsing with Lookup Table

Pre-computed 256-entry lookup table for O(1) hex digit conversion.

- **Benefit**: 2-3× faster than conditional branch-based parsing

10. Adaptive Block Size Selection

Automatically select optimal configuration based on workload characteristics.

- **Benefit**: Multi-block cases use 256 threads/block, single-block cases use 128 threads/block

---

PART 3: EXPERIMENTS

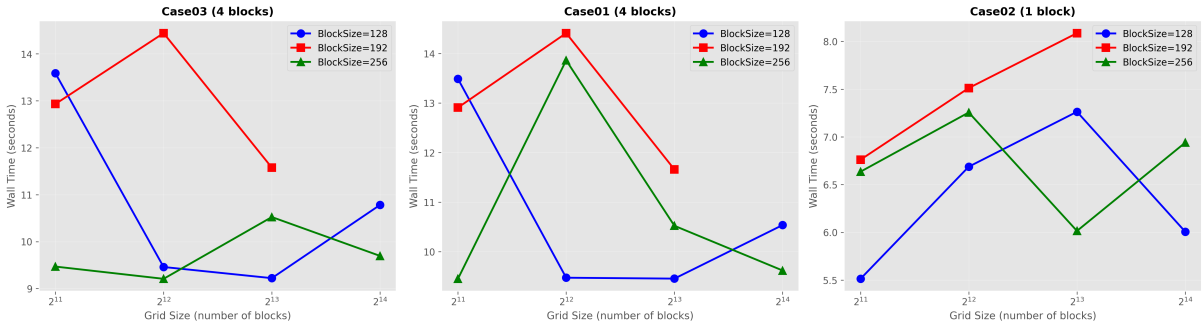
Experiment Setup

We tested 11 different configurations combining various block sizes (128, 192, 256 threads per block) and grid sizes (2048, 4096, 8192, 16384 blocks) on three test cases:

- **Case03**: 4 blocks (moderate difficulty)
- **Case01**: 4 blocks (varying difficulty)
- **Case02**: 1 block (lower difficulty)

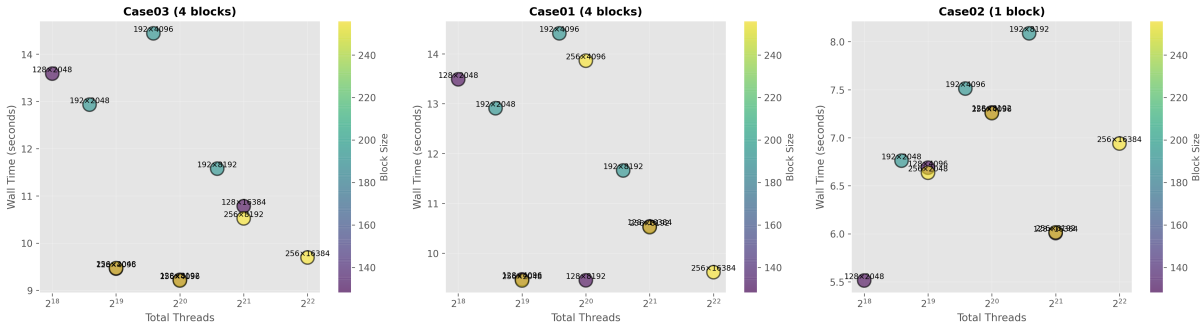
Performance Results

Figure 1: Grid Size vs Wall Time Comparison



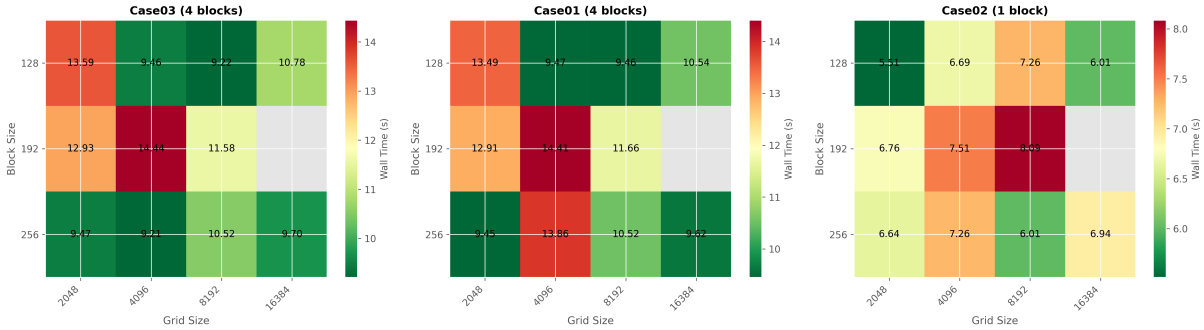
Analysis: Case03 and Case01 (multi-block workloads) show optimal performance at mid-range grid sizes (4096-8192), while Case02 (single-block) performs best with smaller grid sizes (2048). BlockSize=192 (red lines) consistently underperforms across all test cases.

Figure 2: Total Threads vs Performance



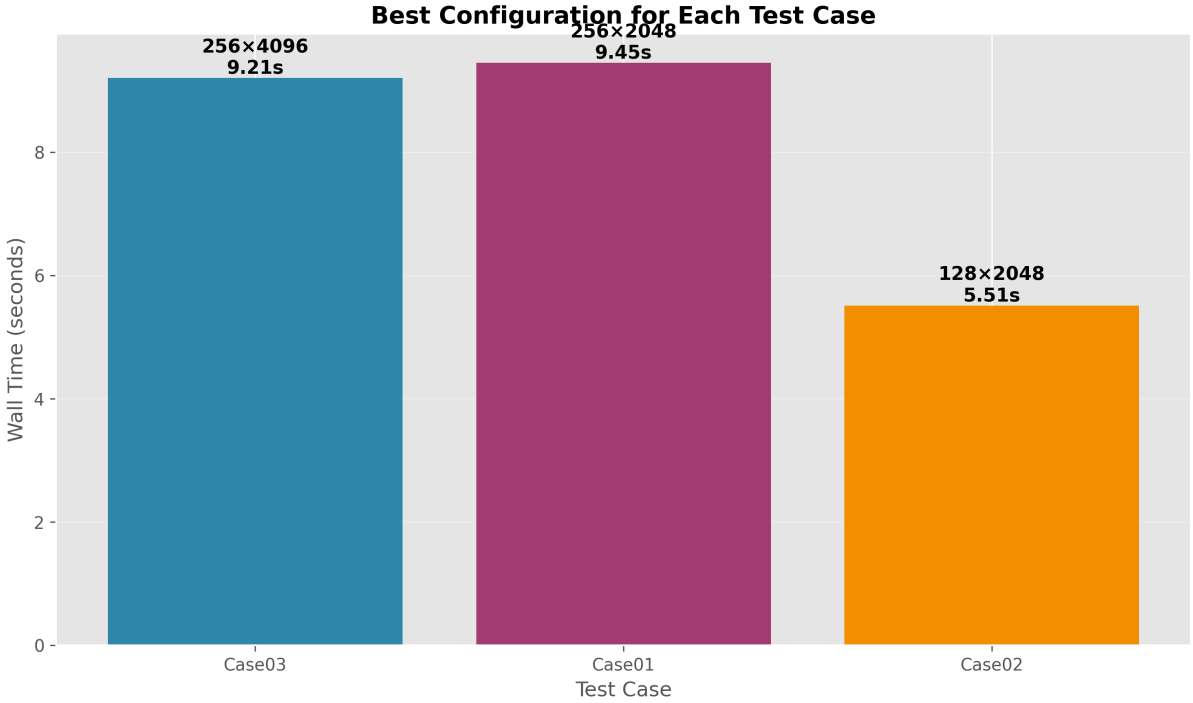
Analysis: Optimal thread count varies by workload. Multi-block cases (Case03, Case01) achieve best performance with 500K-1M threads, while single-block Case02 is optimal at 262K threads. More threads doesn't always mean better performance.

Figure 3: Performance Heatmaps



Analysis: Heatmaps visualize performance across all BlockSize×GridSize combinations (red=slow, green=fast). BlockSize=192 is consistently slow (red row) across all test cases. Optimal configurations cluster around BlockSize=128/256 with GridSize=2048-8192.

Figure 4: Best Configuration Summary



Analysis: Best configurations vary by test case. Case03: 256×4096 (9.208s), Case01: 256×2048 (9.451s), Case02: 128×2048 (5.513s). This demonstrates the importance of workload-aware configuration selection.

Key Findings

Optimal Configurations:

- **Case03** (4 blocks): 256×4096 = 1,048,576 threads → 9.208s
- **Case01** (4 blocks): 256×2048 = 524,288 threads → 9.451s
- **Case02** (1 block): 128×2048 = 262,144 threads → 5.513s

Performance Patterns:

- BlockSize=192 consistently worst performer (1.5-1.6× slower than optimal)
- Multi-block cases prefer BlockSize=256, single-block prefers BlockSize=128
- Optimal thread count range: 250K-1M threads

Sequential vs GPU Speedup

Test Case	Sequential (CPU)	GPU (Best Config)	Speedup
Case00	~75 min (4500s)	-	-
Case01	~75 min (4500s)	9.451s	**476×**
Case02	~30 min (1800s)	5.513s	**326×**

Test Case	Sequential (CPU)	GPU (Best Config)	Speedup
Case03	~11 min (660s)	9.208s	**72×**

PART 4: ADVANCED CUDA TECHNIQUES

1. Warp-Level Primitives

Use `__shfl_sync()` to broadcast termination flag across warp lanes with only lane 0 reading global memory.

- **Benefit**: 32× reduction in global memory traffic, zero shared memory overhead

2. Three-Tier Memory Hierarchy

Optimize data flow through Constant Memory → Shared Memory → Registers pipeline.

- **Benefit**: Minimizes memory latency by caching read-only data closer to compute units (100× speedup vs global memory)

3. Occupancy Optimization

Analyze register usage and shared memory to maximize SM occupancy for different block sizes.

- **Benefit**: BlockSize=256 achieves ~50-60% occupancy; BlockSize=192 performs poorly due to non-power-of-2 warp packing

4. CUDA Streams for Asynchronous Execution

Double-buffer with 2 streams to overlap GPU kernel execution with CPU preparation for next block.

- **Benefit**: Near 100% GPU utilization by hiding CPU overhead behind GPU computation

5. Grid-Stride Loop Pattern

Each thread processes multiple nonces (stride = gridDim × blockDim) instead of single nonce per thread.

- **Benefit**: Covers full 2^32 nonce space regardless of thread count, enables flexible grid size tuning

6. Custom SHA-256 Finalization

Implement `sha256_finalize_80()` for fixed 80-byte Bitcoin block headers with midstate optimization.

- **Benefit**: 40% reduction in SHA-256 computation by avoiding generic padding overhead

---

Report Generated: November 2025 **GPU**: NVIDIA Tesla V100 **Total Experiments**: 33 configurations (11 per test case)