Matrix Multiplication Optimization on V100 GPU

Execution Time

My report is about the details of optimization of a matrix multiplication implementation for V100 GPUs. Starting from a baseline GPU implementation with execution time of 2,624 ms, we achieved significant speedup to 315ms - an 8.3x improvement over the naive implementation and 46.5x faster than the CPU version (14,639.82ms).

Implementation Strategy

My implementation uses a multi-tiered approach that selects specialized kernels based on matrix dimensions. I combined several well-established GPU optimization techniques with V100-specific optimizations to achieve this performance.

Core Optimizations

1. Shared Memory Tiling

All kernels employ shared memory tiling to reduce global memory accesses. For an N×N matrix multiplication, this reduces memory accesses from $O(N^3)$ to $O(N^3/\text{TILE\_SIZE})$.

```
__shared__ float As[TILE_SIZE][TILE_SIZE+1];  // Note the +1 padding
__shared__ float Bs[TILE_SIZE][TILE_SIZE+1];
```

2. Bank Conflict Avoidance

Added padding to shared memory arrays (+1 or +4) to prevent bank conflicts:

```
__shared__ float As[BLOCK_SIZE][BLOCK_SIZE+4];
__shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE+4];
```

Without padding, consecutive threads accessing consecutive elements in the inner dimension would cause bank conflicts, significantly reducing shared memory throughput.

3. Thread Coarsening / Register Blocking

For matrices 1024×1024, each thread computes multiple output elements (2×2 or 4×4), which:

- Improves arithmetic intensity per thread
- Reduces shared memory accesses
- Increases instruction-level parallelism

```
// Each thread handles a 2 2  or 4 4  block of output values
float accum[ITEMS_PER_THREAD_Y][ITEMS_PER_THREAD_X] = {{0.0f
    }};
```

4. Read-Only Cache Utilization

Leveraged V100's texture cache path using `__ldg()` for global memory loads:

```
As[threadIdx.y][threadIdx.x] = __ldg(&A[row * K + tileCol]);
```

This optimization reduces cache thrashing and improves memory access patterns.

5. Double-Precision Accumulation

For improved numerical stability in `matrixMultiplyMixedPrecision`, we use double-precision accumulation:

```
double sum = 0.0;
// ...
sum = __fma_rd(static_cast<double>(As[ty][k]),
               static_cast<double>(Bs[k][tx]),
               sum);
// ...
C[row * N + col] = static_cast<float>(sum);
```

6. Memory Transfer Optimizations

We utilize several techniques to optimize memory transfers:

```
// Pinned memory for faster transfers
CUDA_CHECK(cudaMallocHost(&h_A, n*k*sizeof(float)));

```

```
// Multiple streams for concurrent operations
constexpr int NUM_STREAMS = 4;
cudaStream_t streams[NUM_STREAMS];

// High-priority compute stream
CUDA_CHECK(cudaStreamCreateWithPriority(&streams[0],
                                        cudaStreamNonBlocking,
                                             greatestPriority))
                                           ;
```

7. Cache Configuration Optimization

Configured L1/shared memory balance for optimal performance:

```
cudaDeviceSetCacheConfig(cudaFuncCachePreferEqual);
```

8. Block Size Optimization

For V100 GPUs, we use 64×64 blocks for large matrices, which better matches the V100's warp scheduling and register capabilities:

```
constexpr int BLOCK_SIZE = 64;
```

9. Loop Unrolling

Aggressive loop unrolling with `#pragma unroll` to reduce branch overhead and improve instruction scheduling:

```
#pragma unroll
for (int k = 0; k < BLOCK_SIZE; ++k) {
    sum += As[threadIdx.y][k] * Bs[k][threadIdx.x];
}
```

Matrix Size-Based Kernel Selection

My implementation dynamically selects between specialized kernels based on matrix dimensions:

1. **Very Large Matrices** (2048×2048):
   - Uses `matrixMultiplyV100Large` with 4×4 thread coarsening
   - 64×64 tile sizes for higher occupancy
   - More aggressive thread coarsening
2. **Medium Matrices** (1024×1024):

- Uses `matrixMultiplyV100` with 2×2 thread coarsening
- Good balance between resource usage and parallelism

3. **Small Matrices**:
   - Uses `matrixMultiplyMixedPrecision` with double-precision accumulation
   - Optimizes for numerical precision while maintaining good performance

Performance Analysis

Implementation	Time (ms)	Speedup
CPU Version	14,639.82	1×
Naive GPU	2,624	5.57×
Optimized GPU	315	46.5×

For a 2048×2048×2048 matrix multiplication, this represents approximately:

- **Theoretical FLOPS**: $2 \times 2048 \times 2048 \times 2048 = 17{,}179{,}869{,}184$ operations
- **Performance**: $17{,}179{,}869{,}184 \,/\, (0.315 \times 10^9) = 54.54$ GFLOPS

Conclusion and Future Improvements

My implementation achieves excellent performance on the V100 GPU, with a speedup of 8.3× over the naive GPU implementation and 46.5× over the CPU version. The combination of shared memory tiling, thread coarsening, and V100-specific optimizations proved highly effective.

Potential future improvements:

1. Implementation of Tensor Core operations for half-precision inputs
2. Memory prefetching for overlapped memory access and computation
3. More sophisticated autotuning to select optimal parameters