SpMM vs GEMM Implementation and Performance Analysis

1 Question 1: Dense Matrix-Matrix Multiplication (GEMM)

My GEMM implementation employs several optimizations for high-performance matrix multiplication:

1.1 1. Tiling with Triple Blocking

- **Implementation**: Used three tiling parameters:
  - TILE_ROW = 32 : Block size for rows of matrix A
  - TILE_COL = 64 : Block size for columns of matrix B
  - TILE_INNER = 128 : Block size for inner dimension
- **Purpose**: Optimizes cache utilization by ensuring working sets fit in L1/L2 cache

1.2 2. Memory Management

- **Aligned Memory Allocation**: aligned_alloc(32, bytes) for all matrices
- **Buffer Copying**: Uses temporary buffers buffer_A and buffer_B to improve locality
- **Pre-zeroing**: `std::fill_n(matrix_C, rows * cols, 0.0f)` for result matrix

1.3 3. AVX2 SIMD Vectorization

- **Implementation**:
  - 256-bit registers processing 8 floats per instruction
  - Key AVX2 instructions:

```
__m256 sum = _mm256_loadu_ps(&matrix_C[i_idx * cols + j_idx]);
__m256 a_val = _mm256_set1_ps(row_A[k_idx]);
__m256 b_vec = _mm256_loadu_ps(&buffer_B[k_idx * TILE_COL + (j_idx -
    j_start)]);
sum = _mm256_fmadd_ps(a_val, b_vec, sum);
```

- **Purpose**: 8x throughput for floating-point operations

1.4 4. Thread-Level Parallelism

- **Implementation**:

```
int num_threads = omp_get_num_procs();
omp_set_num_threads(num_threads);
#pragma omp parallel
```

- **Dynamic Scheduling**: `#pragma omp for schedule(dynamic, 1)` for load balancing

1.5 5. Scalar Fallback

- **Implementation**: Scalar code paths for edge cases (tiles not multiple of 8):

```
for (int j_offset = 0; j_idx + j_offset < j_end; ++j_offset) {
    float* result_cell = &matrix_C[i_idx * cols + j_idx + j_offset];
    float cell_sum = *result_cell;

    for (int k_idx = 0; k_idx < k_end - k_start; ++k_idx) {
        cell_sum += row_A[k_idx] * buffer_B[k_idx * TILE_COL + (j_idx -
    j_start) + j_offset];
    }

    *result_cell = cell_sum;
}
```

2 Question 2: Sparse Matrix-Matrix Multiplication (SpMM)

My SpMM implementation efficiently handles sparse matrices using several advanced techniques:

2.1 1. Sparse Accumulator (SPA) Data Structure

- **Implementation**: Hash-based accumulator with optimized collision handling:

```
class SparseAccumulator {
    int* indices;
    double* values;
    bool* occupied;
    size_t capacity, size, table_size, mask;
    inline void add(int col, double val) { ... }
    std::vector<std::pair<int, double>> get_sorted_entries() { ... }
};
```

- **Purpose**: Efficiently accumulates sparse products without expensive searches

2.2 2. Sparsity-Aware Memory Management

- **Implementation**:
  - Output arrays dynamically allocated based on estimated sparsity:

```
int estimate_row_nnz(const int* A_row_ptr, int row,
                     const double* A_values, const int* A_col_ind,
                     const int* B_row_ptr, int k, int n) {
    double A_density = static_cast<double>(A_nnz) / k;
    double B_avg_density = B_total_nnz / (k * n);
    double expected_density = A_density * B_avg_density * 1.5 * k;
}
```

  - Working only with non-zero elements

2.3 3. Prefetching for Irregular Memory Access

- **Implementation**: Strategic prefetching to hide memory latency:

```
if (jA + 1 < A_row_ptr[i+1]) {
    __builtin_prefetch(&A_col_ind[jA + 1], 0, 3);
    __builtin_prefetch(&A_values[jA + 1], 0, 3);
}

__builtin_prefetch(&B_row_ptr[col_A], 0, 3);
```

- **Purpose**: Mitigates cache misses from irregular memory access patterns

2.4 4. Thresholding for Numerical Stability

- **Implementation**: Filters out tiny values to prevent excessive fill-in:

```
if (std::abs(product) >= 1e-14) {
    spa.add(col_B, product);
}
```

  and

```
C_temp[i] = spa.get_sorted_entries(1e-14);
```

- **Purpose**: Maintains sparsity by dropping near-zero results

2.5 5. Adaptive Parallelization

- **Implementation**:

```
int chunk_size = std::max(1, std::min(64, m / (omp_get_max_threads() * 2)));
#pragma omp parallel for schedule(dynamic, chunk_size)
```

- **Purpose**: Balances load while minimizing thread management overhead

3 Question 3: Performance Comparison

3.1 Theoretical Analysis

	GEMM	**SpMM**
Storage	$O(n^2)$ for $n \times n$ matrices	$O(nnz)$ where nnz = non-zeros
Memory Access	Regular, predictable pattern	Irregular, pointer-chasing pattern
Operations	Always $O(n^3)$ for $n \times n$ matrices	$O(nnz(A) \times avg\_nnz\_per\_row(B))$
When to Use	Dense matrices (¡10% zeros)	Sparse matrices (¿90% zeros)

Table 1: Theoretical comparison between GEMM and SpMM

3.2 Benchmarking Results from Our Tests

Matrix used	**Matrix size**	**Non zero entries**	**spmm**	**gemm**
dwt_512 (suit_sparse)	512×512	3,502	10ms	21ms
delaunay_n10 (suit_sparse)	1024×1024	6,112	36ms	53ms
rdb2048 (suit_sparse) (graph)	2048×2048	12,032	74ms	195ms
Facebook (SNAP dataset)	4039×4039	0.54% sparsity	341ms	1047ms

Table 2: Performance comparison between SpMM and GEMM implementations

4 Conclusion

My implementation of SpMM significantly outperforms GEMM for sparse matrices, with performance gains directly proportional to matrix sparsity. The key techniques contributing to this efficiency are:

1. The SparseAccumulator hash table for efficient accumulation
2. Strategic memory prefetching
3. Adaptive thread parallelism with intelligent chunking
4. Sparsity-preserving thresholds for numerical values

These optimizations make SpMM the clear choice for sparse matrices, especially in domains like network analysis, scientific computing, and machine learning with sparse features.