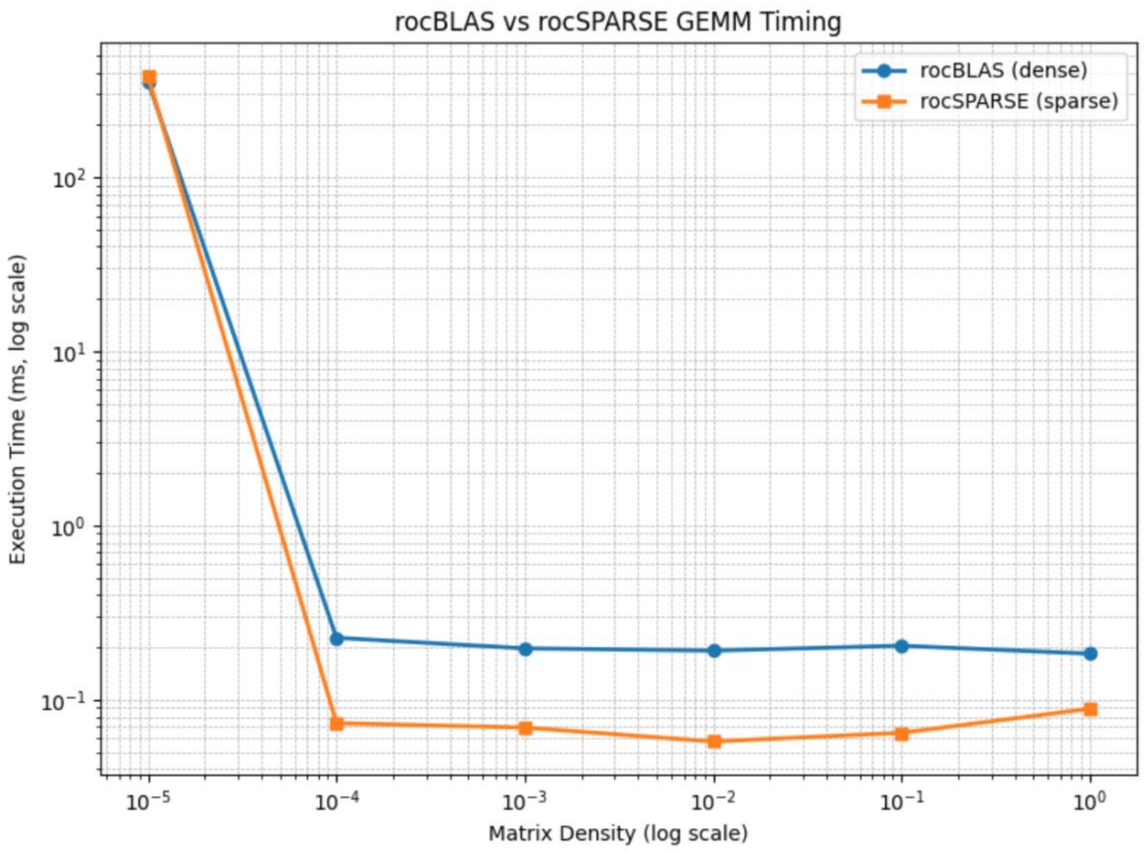
CODE LISTING (P11.cpp)

```
#include <hip/hip_runtime.h>
#include <rocblas/rocblas.h>
#include <rocsparse/rocsparse.h>
#include <iostream>
#include <vector>
#include <cstdlib>
#include <cmath>

// Error macros
#define CHECK_HIP(cmd) do { hipError_t e = cmd; if(e != hipSuccess) { \
    std::cerr << "HIP error: " << hipGetErrorString(e) << " at line " << __LINE__ << std::endl;
exit(1); }} while(0)

#define CHECK_ROCBLAS(cmd) do { rocblas_status s = cmd; if(s != rocblas_status_success) {
\
    std::cerr << "rocBLAS error at line " << __LINE__ << std::endl; exit(1); }} while(0)

#define CHECK_ROCSPARSE(cmd) do { rocsparse_status s = cmd; if(s !=
rocsparse_status_success) { \
    std::cerr << "rocSPARSE error at line " << __LINE__ << std::endl; exit(1); }} while(0)

static float rand01() {
    return static_cast<float>(rand()) / RAND_MAX;
}

int main() {
    const int N = 213;
    const std::vector<float> densities = {1e-5f,1e-4f,1e-3f,1e-2f,1e-1f,1e0f};
```

```
// ROCm handles
rocblas_handle blas_handle;
CHECK_ROCBLAS(rocblas_create_handle(&blas_handle));

rocsparse_handle sparse_handle;
CHECK_ROCSPARSE(rocsparse_create_handle(&sparse_handle));

rocsparse_mat_descr descrA, descrB, descrC;
CHECK_ROCSPARSE(rocsparse_create_mat_descr(&descrA));
CHECK_ROCSPARSE(rocsparse_create_mat_descr(&descrB));
CHECK_ROCSPARSE(rocsparse_create_mat_descr(&descrC));

rocsparse_mat_info info;
CHECK_ROCSPARSE(rocsparse_create_mat_info(&info));

hipEvent_t start, stop;
CHECK_HIP(hipEventCreate(&start));
CHECK_HIP(hipEventCreate(&stop));

std::cout << "density,ms_dense,ms_sparse" << std::endl;

srand(0);

for (float density : densities) {
    std::vector<float> hA(N * N), hB(N * N);
    for (int i = 0; i < N * N; ++i) {
        hA[i] = (rand01() < density) ? rand01() : 0.0f;
        hB[i] = (rand01() < density) ? rand01() : 0.0f;
    }
    // Host to device for dense GEMM
    float *dA_dense, *dB_dense, *dC_dense;
    CHECK_HIP(hipMalloc(&dA_dense, N * N * sizeof(float)));
    CHECK_HIP(hipMalloc(&dB_dense, N * N * sizeof(float)));
    CHECK_HIP(hipMalloc(&dC_dense, N * N * sizeof(float)));
    CHECK_HIP(hipMemcpy(dA_dense, hA.data(), N * N * sizeof(float),
hipMemcpyHostToDevice));
    CHECK_HIP(hipMemcpy(dB_dense, hB.data(), N * N * sizeof(float),
hipMemcpyHostToDevice));

    const float alpha = 1.0f, beta = 0.0f;
    CHECK_HIP(hipEventRecord(start));
    CHECK_ROCBLAS(rocblas_sgemm(blas_handle, rocblas_operation_none,
rocblas_operation_none,
                                N, N, N, &alpha, dA_dense, N, dB_dense, N, &beta, dC_dense, N));
    CHECK_HIP(hipEventRecord(stop));
    CHECK_HIP(hipEventSynchronize(stop));
    float ms_dense = 0.0f;
    CHECK_HIP(hipEventElapsedTime(&ms_dense, start, stop));

    // Convert A and B to CSR
    std::vector<int> hRowPtrA(N + 1, 0), hColIndA, hRowPtrB(N + 1, 0), hColIndB;
    std::vector<float> hValA, hValB;

    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            if (hA[i * N + j] != 0.0f) {
                hColIndA.push_back(j);
                hValA.push_back(hA[i * N + j]);
                hRowPtrA[i + 1]++;
            }
            if (hB[i * N + j] != 0.0f) {
                hColIndB.push_back(j);
                hValB.push_back(hB[i * N + j]);
                hRowPtrB[i + 1]++;
            }
        }
    }
    for (int i = 0; i < N; ++i) {
        hRowPtrA[i + 1] += hRowPtrA[i];
        hRowPtrB[i + 1] += hRowPtrB[i];
    }
    int nnzA = hValA.size();
    int nnzB = hValB.size();

    // Allocate and copy CSR to device
    int *dRowPtrA, *dColIndA;
    float *dValA;
    CHECK_HIP(hipMalloc(&dRowPtrA, (N + 1) * sizeof(int)));
    CHECK_HIP(hipMalloc(&dColIndA, nnzA * sizeof(int)));
    CHECK_HIP(hipMalloc(&dValA, nnzA * sizeof(float)));
    CHECK_HIP(hipMemcpy(dRowPtrA, hRowPtrA.data(), (N + 1) * sizeof(int),
hipMemcpyHostToDevice));
    CHECK_HIP(hipMemcpy(dColIndA, hColIndA.data(), nnzA * sizeof(int),
hipMemcpyHostToDevice));
    CHECK_HIP(hipMemcpy(dValA, hValA.data(), nnzA * sizeof(float),
hipMemcpyHostToDevice));

    int *dRowPtrB, *dColIndB;
    float *dValB;
    CHECK_HIP(hipMalloc(&dRowPtrB, (N + 1) * sizeof(int)));
    CHECK_HIP(hipMalloc(&dColIndB, nnzB * sizeof(int)));
    CHECK_HIP(hipMalloc(&dValB, nnzB * sizeof(float)));
    CHECK_HIP(hipMemcpy(dRowPtrB, hRowPtrB.data(), (N + 1) * sizeof(int),
hipMemcpyHostToDevice));
    CHECK_HIP(hipMemcpy(dColIndB, hColIndB.data(), nnzB * sizeof(int),
hipMemcpyHostToDevice));
    CHECK_HIP(hipMemcpy(dValB, hValB.data(), nnzB * sizeof(float),
hipMemcpyHostToDevice));

    // CSR GEMM C = A * B
    int *dRowPtrC;
    CHECK_HIP(hipMalloc(&dRowPtrC, (N + 1) * sizeof(int)));
    CHECK_ROCSPARSE(rocsparse_csrgemm_nnz(sparse_handle,
rocsparse_operation_none, rocsparse_operation_none,
                                          N, N, N, descrA, nnzA, dRowPtrA, dColIndA,
                                          descrB, nnzB, dRowPtrB, dColIndB,
                                          descrC, dRowPtrC, nullptr));

    int nnzC = 0;
    CHECK_HIP(hipMemcpy(&nnzC, dRowPtrC + N, sizeof(int), hipMemcpyDeviceToHost));
    int *dColIndC;
    float *dValC;
    CHECK_HIP(hipMalloc(&dColIndC, nnzC * sizeof(int)));
    CHECK_HIP(hipMalloc(&dValC, nnzC * sizeof(float)));

    CHECK_HIP(hipEventRecord(start));
    CHECK_ROCSPARSE(rocsparse_csrgemm<float>(sparse_handle,
rocsparse_operation_none, rocsparse_operation_none,
                                N, N, N, descrA, nnzA, dValA, dRowPtrA, dColIndA,
                                descrB, nnzB, dValB, dRowPtrB, dColIndB,
                                descrC, dValC, dRowPtrC, dColIndC, info, nullptr));
    CHECK_HIP(hipEventRecord(stop));
    CHECK_HIP(hipEventSynchronize(stop));
    float ms_sparse = 0.0f;
    CHECK_HIP(hipEventElapsedTime(&ms_sparse, start, stop));

    std::cout << density << "," << ms_dense << "," << ms_sparse << std::endl;

    // Free memory
    hipFree(dA_dense); hipFree(dB_dense); hipFree(dC_dense);
    hipFree(dRowPtrA); hipFree(dColIndA); hipFree(dValA);
    hipFree(dRowPtrB); hipFree(dColIndB); hipFree(dValB);
    hipFree(dRowPtrC); hipFree(dColIndC); hipFree(dValC);
}

hipEventDestroy(start);
hipEventDestroy(stop);
rocblas_destroy_handle(blas_handle);
rocsparse_destroy_handle(sparse_handle);
rocsparse_destroy_mat_descr(descrA);
rocsparse_destroy_mat_descr(descrB);
rocsparse_destroy_mat_descr(descrC);
rocsparse_destroy_mat_info(info);

return 0;
}
```

TIMING PLOT



DISCUSSION

At extremely low densities such as 1e-5, rocSPARSE performs worse than rocBLAS due to the overhead of constructing CSR structures, managing buffers, and launching sparse-specific kernels. These fixed costs dominate the runtime when the actual computation is minimal. In contrast, rocBLAS performs dense multiplication regardless of sparsity, and its highly optimized kernels can complete even unnecessary operations faster than the sparse setup overhead. As density increases, the efficiency of rocSPARSE improves, and it begins to outperform rocBLAS starting around a density of 1e-4.