**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);

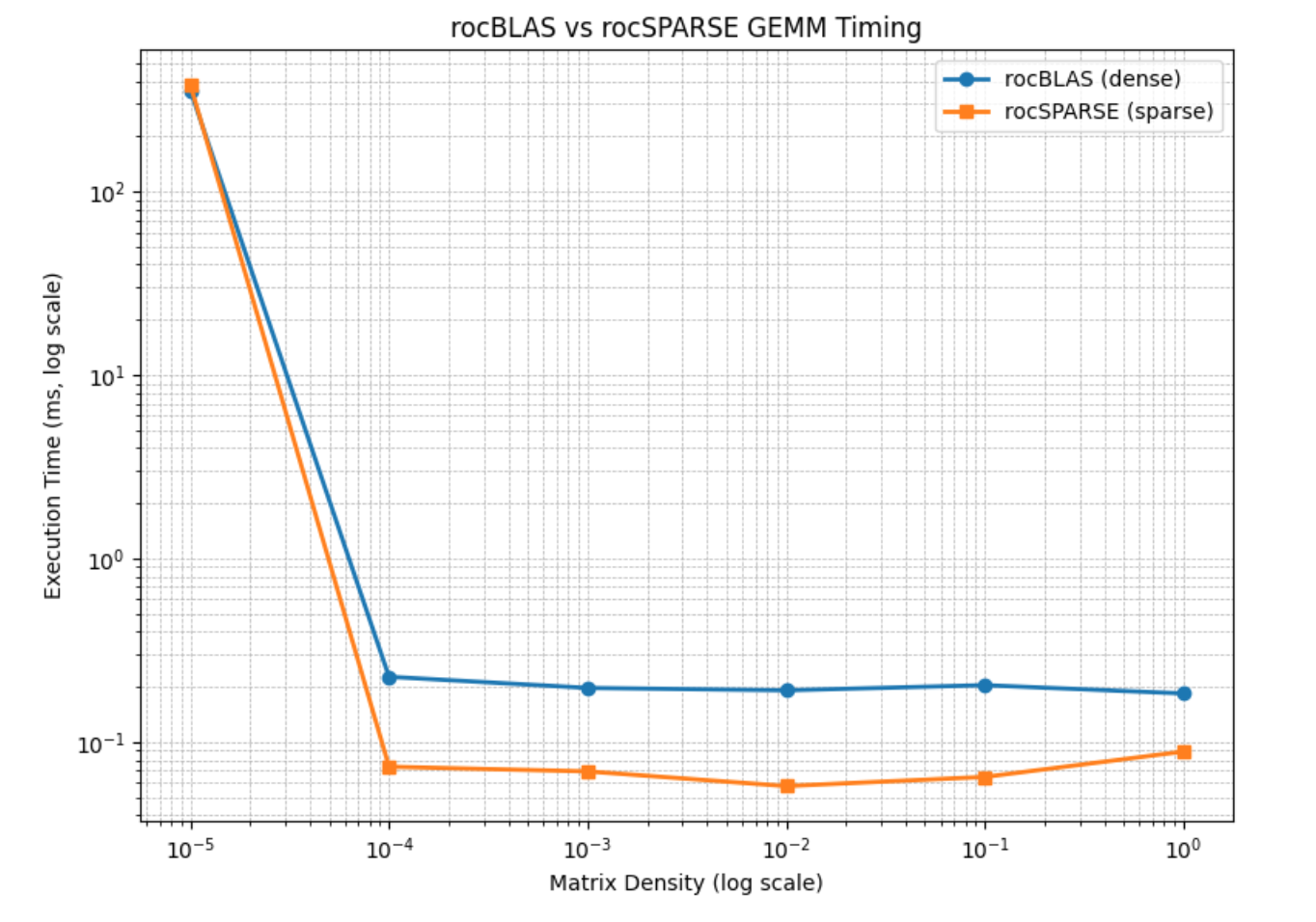
rocsparse\_destroy\_mat\_descr(descrB);

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.