```cpp
#include "bench_funcs_blas.h"
#include <cblas.h>

double dot_cblas(const std::vector<double>& x, const std::vector<double>& y) {
    return cblas_ddot((int)x.size(), x.data(), 1, y.data(), 1);
}

void matvec_cblas(const std::vector<double>& A, std::size_t M, std::size_t N,
                  const std::vector<double>& x, std::vector<double>& b) {
    b.resize(M);
    cblas_dgemv(CblasRowMajor, CblasNoTrans,
                (int)M, (int)N,
                1.0, A.data(), (int)N,
                x.data(), 1,
                0.0, b.data(), 1);
}

void matmul_cblas(const std::vector<double>& A, std::size_t M, std::size_t L,
                  const std::vector<double>& B, std::size_t N,
                  std::vector<double>& C) {
    C.resize(M * N);
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                (int)M, (int)N, (int)L,
                1.0, A.data(), (int)L,
                B.data(), (int)N,
                0.0, C.data(), (int)N);
}
```

```
 1: #ifndef BENCH_FUNCS_BLAS_H
 2: #define BENCH_FUNCS_BLAS_H
 3:
 4: #include <vector>
 5:
 6: // ===== BLAS-based benchmark functions =====
 7:
 8: // (A2) cBLAS dot product
 9: double dot_cblas(const std::vector<double>& x, const std::vector<double>& y);
10:
11: // (B2) cBLAS matrixâ\200\223vector product
12: void matvec_cblas(const std::vector<double>& A, std::size_t M, std::size_t N,
13:                   const std::vector<double>& x, std::vector<double>& b);
14:
15: // (C2) cBLAS matrixâ\200\223matrix product
16: void matmul_cblas(const std::vector<double>& A, std::size_t M, std::size_t L,
17:                   const std::vector<double>& B, std::size_t N,
18:                   std::vector<double>& C);
19:
20: #endif
```

```cpp
#include "bench_funcs.h"
#include <cmath>
#include <cassert>
#include <limits>

double dot_basic(const std::vector<double>& x, const std::vector<double>& y) {
    assert(x.size() == y.size());
    double s = 0.0;
    std::size_t N = x.size();
    for (std::size_t i = 0; i < N; ++i) {
        s += x[i] * y[i];
    }
    return s;
}

double dot_kahan(const std::vector<double>& x, const std::vector<double>& y) {
    assert(x.size() == y.size());
    double sum = 0.0;
    double c = 0.0;
    std::size_t N = x.size();
    for (std::size_t i = 0; i < N; ++i) {
        double prod = x[i] * y[i];
        double yk = prod - c;
        double t = sum + yk;
        c = (t - sum) - yk;
        sum = t;
    }
    return sum;
}

double norm_basic(const std::vector<double>& x) {
    double s = 0.0;
    for (std::size_t i = 0; i < x.size(); ++i) {
    double v = x[i];
    s += v * v;
    }
    return std::sqrt(s);
}

void matvec_rowmajor(const std::vector<double>& A, std::size_t M, std::size_t N,
                     const std::vector<double>& x, std::vector<double>& b) {
    assert(A.size() == M * N);
    assert(x.size() == N);
    b.assign(M, 0.0);
    for (std::size_t i = 0; i < M; ++i) { //over rows
        double sum = 0.0;
        std::size_t base = i * N; //the start index of row 1
        for (std::size_t j = 0; j < N; ++j) { //dot product of that row with x
            sum += A[base + j] * x[j];
        }
        b[i] = sum;
    }
}

void matmul_rowmajor(const std::vector<double>& A, std::size_t M, std::size_t L,
                     const std::vector<double>& B, std::size_t N,
                     std::vector<double>& C) {
    assert(A.size() == M * L);
    assert(B.size() == L * N);
    C.assign(M * N, 0.0);
    for (std::size_t i = 0; i < M; ++i) {
        for (std::size_t k = 0; k < L; ++k) {
            double a = A[i * L + k]; //a=A[i,k]
            std::size_t baseB = k * N;
            std::size_t baseC = i * N;
            for (std::size_t j = 0; j < N; ++j) {
                C[baseC + j] += a * B[baseB + j];
            }
```

```
    69:         }
    70:     }
    71: }
    72:
    73: void polyp_horner(const std::vector<double>& a, const std::vector<double>& x,
    74:                    std::vector<double>& y) {
    75:     std::size_t p = (a.size() == 0) ? 0 : a.size() - 1; //if a.size=0, then p=0, oth
erwise p=a.soze-1
    76:     std::size_t N = x.size();
    77:     y.assign(N, 0.0);
    78:     for (std::size_t i = 0; i < N; ++i) { //loops over all evaluation points x[i]
    79:         double xi = x[i];
    80:         double acc = a[p];
    81:         for (std::size_t k = p; k-- > 0;) { // from p-1 down to 0
    82:             acc = acc * xi + a[k];
    83:         }
    84:         y[i] = acc;
    85:     }
    86: }
    87:
    88: void jacobi_csr(const CSR& K, const std::vector<double>& f, std::vector<double>& u,
    89:                 std::size_t max_iter, double omega, double tol) {
    90:     std::size_t n = K.n;
    91:     assert(f.size() == n);
    92:     u.assign(n, 0.0);
    93:     std::vector<double> u_new(n, 0.0);
    94:     for (std::size_t iter = 0; iter < max_iter; ++iter) { //Jacobi iteration
    95:         double max_err = 0.0;
    96:         for (std::size_t i = 0; i < n; ++i) {
    97:             double diag = 0.0;
    98:             double sum = 0.0;
    99:             for (std::size_t idx = K.row_ptr[i]; idx < K.row_ptr[i+1]; ++idx) { //it
erates over the non zero entries of row i
   100:                 std::size_t j = K.col[idx];
   101:                 double v = K.val[idx];
   102:                 if (j == i) diag = v;
   103:                 else sum += v * u[j]; //accomulates sum = Î£_{jâ\211 i} K_ij * u[j]
   104:             }
   105:             double uk1 = u[i] + omega * (f[i] - sum - diag * u[i]) / diag; //stoers
the diagonal separately
   106:             u_new[i] = uk1;
   107:             double err = std::fabs(uk1 - u[i]);
   108:             if (err > max_err) max_err = err;
   109:         }
   110:         u.swap(u_new);
   111:         if (max_err < tol) break;
   112:     }
   113: }
```

oh! do {

} while

```
 1: #pragma once
 2: #include <vector>
 3: #include <cstddef>
 4:
 5: // now we declare functions that we define in benh_funcs.cpp
 6: double dot_basic(const std::vector<double>& x, const std::vector<double>& y);
 7: double dot_kahan(const std::vector<double>& x, const std::vector<double>& y);
 8: double norm_basic(const std::vector<double>& x);
 9:
10: void matvec_rowmajor(const std::vector<double>& A, std::size_t M, std::size_t N,
11:                      const std::vector<double>& x, std::vector<double>& b);
12:
13: void matmul_rowmajor(const std::vector<double>& A, std::size_t M, std::size_t L,
14:                      const std::vector<double>& B, std::size_t N,
15:                      std::vector<double>& C);
16:
17: void polyp_horner(const std::vector<double>& a, const std::vector<double>& x,
18:                   std::vector<double>& y);
19:
20: struct CSR { //compressed Sparse Row
21:     std::size_t n;                  // number of rows
22:     std::vector<double> val;        // non-zero values
23:     std::vector<std::size_t> col;   // column indices
24:     std::vector<std::size_t> row_ptr; // row pointers (size n+1)
25: };
26: void jacobi_csr(const CSR& K, const std::vector<double>& f, std::vector<double>& u,
27:                 std::size_t max_iter, double omega, double tol);
```

```cpp
#include <iostream>
#include <vector>
#include <iomanip>
#include <chrono>
#include <cmath>
#include "bench_funcs_blas.h"

using namespace std;
using namespace std::chrono;

void gen_vector_x_y(std::size_t N, std::vector<double>& x, std::vector<double>& y) {
    x.resize(N);
    y.resize(N);
    for (std::size_t i = 0; i < N; ++i) {
        x[i] = static_cast<double>((i % 219) + 1); // xi := (i mod 219) + 1
        y[i] = 1.0 / x[i]; // yi := 1/xi
    }
}

void gen_matrix_A(std::size_t M, std::size_t N, std::vector<double>& A) {
    A.resize(M * N);
    for (std::size_t i = 0; i < M; ++i) {
        for (std::size_t j = 0; j < N; ++j) {
            A[i * N + j] = static_cast<double>(((i + j) % 219) + 1);
        }
    }
}

high_resolution_clock::time_point tic_timer;
void tic() { tic_timer = high_resolution_clock::now(); }
double toc() {
    auto t1 = high_resolution_clock::now();
    duration<double> elapsed = t1 - tic_timer;
    return elapsed.count();
}

//CHANGE FLAG BASED ON WHAT YOU WANT TO DO
int main() {
    cout << fixed << setprecision(6);
    int flag = 2;  // 1=A2 (dot), 2=B2 (matvec), 3=C2 (matmul)

    if (flag == 1) {// A2) DOT via cBLAS
        size_t N = 5000000;
        vector<double> x, y;
        gen_vector_x_y(N, x, y);
        cout << "Running cBLAS dot (A2)\n";
        tic();
        double s = dot_cblas(x, y);
        double dt = toc();

        double flops = 2.0 * N;
        double gflops = (flops / dt) / 1e9;
        double traffic_bytes = 2.0 * N * sizeof(double);
        double gib_s = (traffic_bytes / dt) / (1024.0 * 1024.0 * 1024.0);

        cout << "A2 (BLAS): N=" << N << " time=" << dt
             << " s  GFLOPS=" << gflops << "  GiB/s=" << gib_s << "\n";
    }

    else if (flag == 2) { // B2) GEMV via cBLAS
        size_t M = 10000, N = 10000;
        vector<double> A, x, b;
        gen_matrix_A(M, N, A);
        x.resize(N);
        for (size_t j = 0; j < N; ++j)
            x[j] = 1.0 / (((17 + j) % 219) + 1);

        cout << "Running cBLAS matvec (B2)\n";
```

more flexible via command line argument

```
   69:         tic();
   70:         matvec_cblas(A, M, N, x, b);
   71:         double dt = toc();
   72:
   73:         double flops = 2.0 * M * N;
   74:         double gflops = (flops / dt) / 1e9;
   75:         double traffic_bytes = (M * N + N + M) * sizeof(double);
   76:         double gib_s = (traffic_bytes / dt) / (1024.0 * 1024.0 * 1024.0);
   77:
   78:         cout << "B2 (BLAS): M=" << M << " N=" << N << " time=" << dt
   79:              << " s  GFLOPS=" << gflops << "  GiB/s=" << gib_s << "\n";
   80:     }
   81:
   82:     else if (flag == 3) { // C2) GEMM via cBLAS
   83:         size_t M = 500, L = 500, N = 500;
   84:         vector<double> A, B, C;
   85:         gen_matrix_A(M, L, A);
   86:         gen_matrix_A(L, N, B);
   87:
   88:         cout << "Running cBLAS matmul (C2)\n";
   89:         tic();
   90:         matmul_cblas(A, M, L, B, N, C);
   91:         double dt = toc();
   92:
   93:         double flops = 2.0 * M * L * N;
   94:         double gflops = (flops / dt) / 1e9;
   95:         double traffic_bytes = (M * L + L * N + M * N) * sizeof(double);
   96:         double gib_s = (traffic_bytes / dt) / (1024.0 * 1024.0 * 1024.0);
   97:
   98:         cout << "C2 (BLAS): M=" << M << " L=" << L << " N=" << N << " time=" << dt
   99:              << " s  GFLOPS=" << gflops << "  GiB/s=" << gib_s << "\n";
  100:     }
  101:
  102:     else {
  103:         cout << "Invalid flag. Choose 1â\200\2233.\n";
  104:     }
  105:
  106:     cout << "\nDone\n";
  107:     return 0;
  108: }
```

```
    1: #include <iostream>
    2: #include <vector>
    3: #include <cmath>
    4: #include <iomanip>
    5: #include <chrono>
    6: #include "bench_funcs.h"
    7: #include "bench_funcs.cpp"
    8:
    9: using namespace std;
   10: using namespace std::chrono;
   11:
   12: void gen_vector_x_y(std::size_t N, std::vector<double>& x, std::vector<double>& y) {
   13:     x.resize(N);
   14:     y.resize(N);
   15:     for (std::size_t i = 0; i < N; ++i) {
   16:         x[i] = static_cast<double>((i % 219) + 1); // xi := (i mod 219) + 1
   17:         y[i] = 1.0 / x[i]; // yi := 1/xi
   18:     }
   19: }
   20:
   21: void gen_matrix_A(std::size_t M, std::size_t N, std::vector<double>& A) {
   22:     A.resize(M * N);
   23:     for (std::size_t i = 0; i < M; ++i) {
   24:         for (std::size_t j = 0; j < N; ++j) {
   25:             A[i * N + j] = static_cast<double>(((i + j) % 219) + 1);
   26:         }
   27:     }
   28: }
   29:
   30: high_resolution_clock::time_point tic_timer;
   31: void tic() { tic_timer = high_resolution_clock::now(); }
   32: double toc() {
   33:     auto t1 = high_resolution_clock::now();
   34:     duration<double> elapsed = t1 - tic_timer;
   35:     return elapsed.count();
   36: }
   37:
   38: //CHANGE FLAG BASED ON WHAT YOU WANT TO DO
   39: int main() {
   40:     int flag = 8;          // 1=A, 2=B, 3=C, 4=D, 5=Jacobi 6=Norm in 5a, 7= Kahan i
n 5b and 8=colums access in 5c
   41:     size_t N = 5000000;       // default vector length
   42:     size_t M = 3000, L = 3000;
   43:     if (flag == 1) {
   44:         // A) Inner product
   45:         vector<double> x, y;
   46:         gen_vector_x_y(N, x, y);
   47:         cout << "Running dot_basic" << endl;
   48:         tic();
   49:         volatile double s = dot_basic(x, y); (void)s;
   50:         double dt = toc();
   51:
   52:         double flops = 2.0 * N; //x_i*y_i=1 FLOPS, the sum = 1 FLOPS, for N times =
2N
   53:         double gflops = (flops / dt) / 1e9; // computes how many flops per second do
es my computer and then converts to giga FLOPS
   54:         double traffic_bytes = 2.0 * N * sizeof(double); //Memory usage in bytes: 2
vectors of length N times the size of double
   55:         double gib_s = (traffic_bytes / dt) / (1024.0*1024.0*1024.0); //computes how
 many bytes are moved by my computer per second and converts to Gibibytes per second
   56:
   57:         cout << "A: N=" << N << " time=" << dt << " s  GFLOPS=" << gflops
   58:              << "  GiB/s=" << gib_s << "\n";
   59:     }
   60:
   61:     else if (flag == 2) {
   62:         //B) Matrix*vector
   63:         size_t m = M, n = 5000;
```

```
   64:         vector<double> A, x, b;
   65:         gen_matrix_A(m, n, A);
   66:         x.resize(n);
   67:         for (size_t j = 0; j < n; ++j)
   68:             x[j] = 1.0 / (((17 + j) % 219) + 1);
   69:
   70:         cout << "Running matvec\n";
   71:         tic();
   72:         matvec_rowmajor(A, m, n, x, b);
   73:         double dt = toc();
   74:
   75:         double flops = 2.0 * m * n; //each y_i does N multiplications and N addition
s = Mx2N
   76:         double gflops = (flops / dt) / 1e9;
   77:         double traffic_bytes = (m*n + n + m) * sizeof(double); //Memory usage in byt
es: Matrix size mxn, 2 vectors n and m
   78:         double gib_s = (traffic_bytes / dt) / (1024.0*1024.0*1024.0);
   79:
   80:         cout << "B: M=" << m << " N=" << n << " time=" << dt
   81:              << " s  GFLOPS=" << gflops << "  GiB/s=" << gib_s << "\n";
   82:     }
   83:
   84:     else if (flag == 3) {
   85:         // C) Matrix*matrix
   86:         size_t m = M, l = L, n = 500;
   87:         vector<double> A, B, C;
   88:         gen_matrix_A(m, l, A);
   89:         gen_matrix_A(l, n, B);
   90:
   91:         cout << "Running matmul\n";
   92:         tic();
   93:         matmul_rowmajor(A, m, l, B, n, C);
   94:         double dt = toc();
   95:
   96:         double flops = 2.0 * m * l * n; //each element of C does N moltiplications a
nd N additions: dim(C)=MxL hence MxLx2N
   97:         double gflops = (flops / dt) / 1e9;
   98:         double traffic_bytes = (m*l + l*n + m*n) * sizeof(double); //Memory usage in
 bytes: 3 matices MxN, NxL, MxL
   99:         double gib_s = (traffic_bytes / dt) / (1024.0*1024.0*1024.0);
  100:
  101:         cout << "C: M=" << m << " L=" << l << " N=" << n << " time=" << dt
  102:              << " s  GFLOPS=" << gflops << "  GiB/s=" << gib_s << "\n";
  103:     }
  104:
  105:     else if (flag == 4) {
  106:         // D) Polynomial
  107:         size_t p = 100; // degree
  108:         vector<double> a(p+1), x(N), y;
  109:         for (size_t k = 0; k <= p; ++k)
  110:             a[k] = 1.0 / (k+1);
  111:         for (size_t i = 0; i < N; ++i)
  112:             x[i] = (i % 219) * 0.001 + 1.0;
  113:
  114:         cout << "Running POLYNOMIAL test (D)\n";
  115:         tic();
  116:         polyp_horner(a, x, y);
  117:         double dt = toc();
  118:
  119:         double flops = 2.0 * p * N; //each evaluation p moltiplications + p addition
s = 2pxN
  120:         double gflops = (flops / dt) / 1e9;
  121:         double traffic_bytes = (p+1 + N + N) * sizeof(double); //Memory usage in byt
es: p+1 coefficients and N evaluation points
  122:         double gib_s = (traffic_bytes / dt) / (1024.0*1024.0*1024.0);
  123:
  124:         cout << "D: p=" << p << " N=" << N << " time=" << dt
  125:              << " s  GFLOPS=" << gflops << "  GiB/s=" << gib_s << "\n";
```

```
126:     }
127:
128:     else if (flag == 5) {
129:         // E) Jacobi
130:         size_t n = 10000;
131:         CSR K; K.n = n;
132:         vector<double> f(n, 1.0), u;
133:         K.row_ptr.resize(n+1);
134:         K.val.reserve(3*n);
135:         K.col.reserve(3*n);
136:
137:         for (size_t i = 0; i < n; ++i) {
138:             K.row_ptr[i] = K.val.size();
139:             if (i > 0) { K.val.push_back(-1.0); K.col.push_back(i-1); }
140:             K.val.push_back(2.0); K.col.push_back(i);
141:             if (i+1 < n) { K.val.push_back(-1.0); K.col.push_back(i+1); }
142:         }
143:         K.row_ptr[n] = K.val.size();
144:
145:         size_t maxit = 5000;
146:         double omega = 1.0, tol = 1e-8;
147:
148:         cout << "Running JACOBI solver test...\n";
149:         tic();
150:         jacobi_csr(K, f, u, maxit, omega, tol);
151:         double dt = toc();
152:         cout << "Jacobi: n=" << n << " time=" << dt << " s\n";
153:     }
154:
155:     else if (flag == 6) {//5(a)
156:     size_t N = 5000000; // large enough for ~10 s runtime
157:     vector<double> x(N);
158:     for (size_t i = 0; i < N; ++i)
159:         x[i] = static_cast<double>((i % 219) + 1);
160:
161:     cout << "Running norm test (A2)\n";
162:     tic();
163:     volatile double s = 0.0;
164:     for (size_t i = 0; i < N; ++i)
165:         s += x[i] * x[i];
166:     double dt = toc();
167:
168:     double flops = 2.0 * N; // 1 mult + 1 add per element
169:     double gflops = (flops / dt) / 1e9;
170:     double traffic_bytes = N * sizeof(double); // only one vector is read, memory is
halved
171:     double gib_s = (traffic_bytes / dt) / (1024.0*1024.0*1024.0);
172:
173:     cout << "A2 (norm): N=" << N << " time=" << dt
174:          << " s  GFLOPS=" << gflops << "  GiB/s=" << gib_s << "\n";
175:     }
176:
177:     else if (flag == 7) { //5(b) Kahan
178:     size_t N = 50'000'000;
179:     vector<double> x(N), y(N);
180:     for (size_t i = 0; i < N; ++i) {
181:         x[i] = static_cast<double>((i % 219) + 1);
182:         y[i] = 1.0 / x[i];
183:     }
184:
185:     cout << "Running Kahan dot product (A3)...\n";
186:     tic();
187:     volatile double s = dot_kahan(x, y); (void)s;
188:     double dt = toc();
189:
190:     double flops = 6.0 * N; // more operations per element than standard dot
191:     double gflops = (flops / dt) / 1e9;
192:     double traffic_bytes = 2.0 * N * sizeof(double);
```

```
193:     double gib_s = (traffic_bytes / dt) / (1024.0*1024.0*1024.0);
194:
195:     cout << "A3 (Kahan): N=" << N << " time=" << dt
196:          << " s  GFLOPS=" << gflops << "  GiB/s=" << gib_s << "\n";
197:     }
198:
199:     else if (flag == 8) { //5(c): compare row-wise vs column-wise matrix access
200:     size_t M = 1500, L = 1500, N = 1500;
201:     vector<double> A, B, C;
202:     gen_matrix_A(M, L, A);
203:     gen_matrix_A(L, N, B);
204:
205:     cout << "Running matrix-matrix (row-wise)\n";
206:     tic();
207:     matmul_rowmajor(A, M, L, B, N, C);
208:     double dt_row = toc();
209:
210:     cout << "Row-wise time: " << dt_row << " s\n";
211:
212:     // Column-wise version
213:     C.assign(M*N, 0.0);
214:     cout << "Running column-wise version\n";
215:     tic();
216:     for (size_t j = 0; j < N; ++j)
217:         for (size_t k = 0; k < L; ++k)
218:             for (size_t i = 0; i < M; ++i)
219:                 C[i*N + j] += A[i*L + k] * B[k*N + j];
220:     double dt_col = toc();
221:     cout << "Column-wise time: " << dt_col << " s\n"; //usually slower because it ca
uses cache misses and lower memory bandwidth
222:     }
223:
224:     else {
225:         cout << "Invalid flag. Choose 1â\200\2238.\n";
226:     }
227:
228:     cout << "\nDone\n";
229:     return 0;
230: }
```