Lab 3

Lab III

Warmup – vector reduction

Create an array of random floating point values. The length of the array should be the benchmark parameter. Compute their sum in the following ways:

1. Use `makeRandomVector` to create the array (in this case, wrapped in a `std::vector`). Implement the reduction naively, i.e., use a single accumulator and loop over the vector.
2. Use `makeRandomVector` to create the array (in this case, wrapped in a `std::vector`). Use a vector register-sized accumulator and compute the reduction in a vectorized fashion. To this end, use the compiler intrinsics appropriate for your CPU. At the end of the loop, manually sum the entries of the accumulator (store them to memory and do a final small loop). Since the underlying array is only guaranteed to be aligned to the size of the floating point value, be sure to use unaligned load/store instrinsics!
3. Repeat 2, this time using aligned storage and memory access. Use the provided convenience function `makeRandomAlignedArray` to ensure alignment (align to 64B to further optimize cache behavior).
4. Compute the reduction using an aligned array and the standard algorithm `std::reduce`.

Compare and discuss your results.

Vectorizing matrix-matrix multiplication

Vectorize the Goto algorithm code written during the previous class. Assume that the underlying arrays are properly aligned (you can check this in the initialization code). Note that only the micro-kernel needs to be modified. Be sure to use the fused multiply-add operations. Discuss your observations with other students. Ask the instructors for help, if needed.

Introduction to Eigen

Installation

Installation of Eigen is actually not required, as it is a header-only library. Simply download the source code from Gitlab and include the requisite headers. If you're using `spack`, `spack install eigen` will do the trick (just remember to use the corresponding `load` command). The functionality discussed herein is contained within the `Eigen/Dense` header. Be sure to have the quick reference page open.

`Eigen::Matrix`

`Matrix` is the foundation of Eigen. It is a template used to represent matrices. The type of the values stored in the matrix, as well as its size are given by the template parameters. For example, `Eigen::Matrix<double, 4, 6>` is a type representing a 4x6 matrix of values of type `double`. If the size of the matrix is unknown at compile time, the special value `Eigen::Dynamic` can be used. The matrix manages its own storage, meaning that no manual memory management is required from the user. Some convenience typedefs are provided by the library, for example:

- `Eigen::Matrix3d` <=> `Eigen::Matrix<double, 3, 3>`
- `Eigen::MatrixXd` <=> `Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic>`
- `Eigen::VectorXd` <=> `Eigen::Matrix<double, Eigen::Dynamic, 1>`

The matrix entries are uninitialized by default. You can access them using `operator()`, leading to Matlab-like code. Note that Eigen uses 0-based indexing. For example:

```
Eigen::Matrix3d mat;
mat(0, 0) = 3.14;
mat(1, 1) = 1.;
mat(2, 2) = 42.;
std::cout << mat; // note the support for iostreams
// diagonal initialized, off-diagonal entries contain garbage from memory
```

Initialization to some common cases can be done as follows

```
using namespace Eigen;
MatrixXd m1 = MatrixXd::Zeros(10, 12); // dynamic size => dimensions nee
Matrix3d m2 = Matrix3d::Ones();        // static size => dimensions infe
Vector4d v1 = Vector4d::Random();
```

Alternatively, you can use the `setX` methods:

```
m1.setZero();
m2.setOnes();
m3.setRandom();
```

Eigen arithmetic

Arithmetic in Eigen is defined in terms of the usual operators: `+`, `-`, `*`, etc. The only catch is that operators don't return matrices, but proxy objects, which can be converted into matrices using the appropriate constructor or assignment operator. What this means in practice is that we can't use `auto` when creating new matrices:

```
auto mat3 = mat1 * mat2; // the deduced type is not a matrix, but a prox
// Instead do
Matrix<...> mat3 = mat1 * mat2;
```

The reason for this behavior is that it allows optimization on chain expressions, e.g.

```
mat4 = mat1 * mat2 + mat3;
```

would normally first evaluate the product of `mat1` and `mat2`, and then sum the temporary result with `mat3`. Instead, some optimizations can be performed, e.g., writing the result of the temporary product to `mat4` and then doing the sum in-place. This does not meaningfully impact usage (other than not being able to use `auto`), but it is worth knowing about.

Transpositions are done as follows

```
c = a.transpose() * b;
d.transposeInPlace();
```

Algebraic solvers

The final feature of Eigen that we will mention is solving systems of algebraic equations. In Matlab, this is simply done using the \ operator, which selects an algorithm based on some heuristic. In Eigen, we need to explicitly specify which solver we'd like to call. The full list is available here. The syntax is as follows

```
// Solve random 5x5 system of equations
constexpr int n = 5;
Matrix<double, n, n> A;
Matrix<double, n, 1> b;
A.setRandom();
b.setRandom();

// Householder rank-revealing QR decomposition of a matrix with column-pivotir
vec_t x = A.colPivHouseholderQr().solve(b);

std::cout << "A:\n" << A;
std::cout << "b:\n" << b;
std::cout << "Ax = b => x:\n" << x;
```

Notes

- Eigen is highly optimized for multiple architectures. The code is vectorized. Feel free to confirm this by comparing its performance with that of your own code.
- Eigen is multi-threaded by default, assuming you link your executable with OpenMP (by passing `-fopenmp` to the compiler). To disable this (e.g. for the purpose of comparison with non-multi-threaded code), define the `EIGEN_DONT_PARALLELIZE` macro before including the library headers (or just don't link with OpenMP).
- Eigen has many interesting features that were not covered here. Examples include eigensolvers and sparse matrix support. Feel free to explore these, especially for the purposes of your projects.
- Eigen is open-source and developed by volunteers. Consider contributing!

Problems for self-study

1. Compare the performance of your matrix-matrix product with that of Eigen
2. Implement the power method
3. Consider how you can use Eigen in your project