Modern C++ for High-Performance Computing
(the slides are available under “Presentation Materials” in the above URL)
Date: June 12, 2019
Presented by: Andrew Lumsdaine (Pacific Northwest National Laboratory & University of Washington)

Q. Why have num_rows_, when the vector knows its own length?

A. Yes, in principle this member is not necessary. I think that it just makes it more clear in the slides.
B. Std::vector probably computes its size. If an instance of this Vector will always have the same size, it’s better to store the size than have std::vector compute it when necessary
    a. I don’t think that there is any computation involved. Std::vector stores the size in the same way that Andrew’s vector does.
    b. All implementations of std::vector of which I am aware store three pointers. The size() member function subtracts pointers to get a length. This is fast. The cost of storing the extra size_t value makes each vector larger, and thus comes as some cost. I question whether storing the (redundant) size is really an optimization, and wanted to know of the speaker has data saying that it is.

Q. is there any performance issue using move over copy constructor?

A. There is no performance issue I am aware of. There is a semantic issue where move leaves the variable moved from in uninitialized state.

Q. When using a move constructor such as X(X&&) does this mean that you move the object into the new object? Or does it still copy?

A. The move constructor *may* actually move, but the language does not enforce that the move actually happens. However, X(X&&) really should actually move the X into a new object. (e.g., a move constructor could have empty body that does not really move anything, but that would be unusual)

Q. In reference to the comment, “A pointer should almost never be necessary”, what is the recommendation for multi-lingual programming?

A. When memory is returned from outside of C++, it can (should) be encapsulated in std::span (C++20 or GSL https://github.com/microsoft/GSL) or similar (there are quite a few core guidelines on how to deal with raw pointers, but if you can, you should avoid them). When memory needs to be allocated in C++ to pass somewhere else, it can be obtained in a multitude of ways (e.g., by having an std::vector variable, then std::vector::data()).
    a. BTW, sometimes we just get a void*. Then we need to cast it back and forth as we get it to our C++ code and then out. This may happen in HPC with a legacy library (e.g., Global Arrays at PNNL return void*). We can use reinterpret_cast in such cases.

Q. What was the “big six” he was referring to earlier?

A. The automatically generated special members (constructors, assignment, destructor).

Q. Some questions about the use of std::vector for the Vector and Matrix classes and the performance test run:

1. Were the tests compiled with alignment or vectorization options in addition to general optimization flags?
2. Why not use alignas(64) double storage_[num_rows_]; instead of std::vector?
3. Isn’t the strength of std::vector the ability to resize the container? Is resizing an intended capability for the Vector and Matrix classes?

A.

a. I think the tests were compiled with -Ofast and -march=native.
b. Std::vector is clear, not clever, and it should only not be used if there is a good reason for it (don’t be clever and premature optimization is the source of all evil).
    i. Also, num_rows_ is a dynamic argument to the constructor.
        1. Good point, for this case, why use std::vector rather than,
            double *storage_ = nullptr;
            posix_memalign(storage, 64, num_rows_); ,
        with free(storage_); in the destructor?

        A: This could be done, but why? Is there a case for complicating the code?
        Vectorization of operations using the Vector class.

OK, this could probably use an additional abstraction such as a vector class that has some guarantees for vectorization. So if one could show that this helps, then yes, but would be nice to wrap up in some reusable abstraction.

c. In the simplified version showed in the slides, there is no interface for resizing. Std::vector can be used for dynamically resized storage, but it is also good for just getting some storage without the need for resizing. In general, there could be a well thought out interface for resizing vectors and matrices, but it is not developed in the slides.

Q. how it use a memory? virtual memory alloc?

A. Could you clarify? I have thought vector(size) how its allocate memory?
B. I don't know that std::vector has any guarantee on how it obtains memory. If it does, I am not aware of this of the top of my head.

Q. If you were to disallow moving by the = operator with X& operator=(X&&)=delete and you were to use a function that specifically used moving like std::swap(..), would that be an error or would it default to a copy?

A. I think that in C++17, it would be an error. In earlier versions there would be a copy. (https://en.cppreference.com/w/cpp/algorithm/swap)

Q. What high-performance tensor operations (multidimensional arrays- permutations, transposes etc) libraries can you recommend?
Could you repeat please?

A. We mentioned the NWChem-Ex project which is still in development and TAL_SH for GPU tensor operations. NWChem-Ex will at some point develop its own GPU implementation.