Accelerating Numerical Software Libraries with Multi-Precision Algorithms
(the slides are available under “Presentation Materials” in the above URL)
Date: May 13, 2020
Presented by: Hartwig Anzt (Karlsruhe Institute of Technology) and Piotr Luszczek (University of Tennessee)

Q. Do you block the system? (slide 18)

A. No, we use the matrix in plain CSR format

Q. Why not twice as fast? (slide 18)

A. If all the data included in the sparse algorithm would be of ValueType - and all operations running at the bandwidth, we would see a speedup of two. Unfortunately, the CSR storage introduces some overhead (column indices & row pointer)

Q. Would it help to run most iterations with single precision, then finally converge with higher precision?

A1. You can use that strategy for fixed-point iterations, e.g.: https://dl.acm.org/doi/abs/10.1145/3380934. You start with low precision, then slowly increase the precision.
A2. It is also possible for Krylov-type solvers. However, in this case you need to start with high precision, relax the precision in later iterations.
https://arxiv.org/pdf/1907.10550.pdf

Q. Could you contrast the C++ precision templates with Julia's multiple dispatch model? Julia is JIT compiled so it compiles on the first iteration but I’m not an expert either.

A. C++ precision templates are simply regular C++ templates on primitive floating-point types: solver<float>(A, x) or solver<double>(A, x). Julia has those as well: function solver(A::T, x::T) where {T} … end You can inquire what Julia functions have been defined with methods(solver) but it’s a little harder to find all implementations of “solver” that C++ compiler sees. Multiple dispatch is a feature of Julia that is more dynamic and would resolve at runtime. C++ has limited support for it with virtual methods and class inheritance: only single dispatch on “this” is possible.

Q. I've read that newer CPUs convert single precision to double precision because hardware for double precision is faster than single precision. Can you comment about that?

A. There might be processors like that especially in the embedded space (consider IBM's Blue Gene line of supercomputers). But for mainstream CPUs and accelerators the opposite is true.

Q. Would it not be fairer to compare speeds asking for only a single prec residuum?

A. The requirement for the magnitude of the residual comes from the application domain. If your application requires only single precision accuracy then you should probably switch everything to single precision and reap all the potential benefits. In our case, we give to the user a double precision accuracy while computing in single precision internally at higher rate.

Q. Since once SP stagnates you can get underflows etc etc which can raise traps and things?

A. Yes, if , for example, values exceed the exponent range, the single precision will fail.

Q. Is there a lightweight multi - reduced-precision C++ template library that you recommend? Do they use DL Boost calls on CPUs?

A. Gingko, KokkosKernels, Trilinos, Eigen.

Q. Do you have the ability to accumulate (add) dot products in longer precision? Wilkinson showed this improved accuracy.

A. No, we do not. On CPU, it is sometimes possible to do it.

Q. Do computer architectures like FPGAs open any interesting possibilities for implementing mixed precision algorithms?

A. Yes, there is a lot of research in this direction. Also in the direction of natively supporting non-conventional formats.

Q. Do you literally invert the blocks? Why not solve a system, is there a specific reason for that? (it is common belief to "never invert the matrix")

A. The numerical effects are negligible for this small block size. The runtime difference inversion vs. factorization is negligible (bandwidth-bound). The application is much faster using inverted blocks. See:
https://www.icl.utk.edu/files/publications/2018/icl-utk-1068-2018.pdf

Q. How do you choose the appropriate lower precision that you use? If that comes from experimenting, in reality would anyone run their application multiple times to figure out which precision to use? And what about memory use if more than one precisions are used.

A. It is determined using the condition number of the distinct diagonal blocks and the accuracy we want to preserve.

Q. For the numerical experiments, do you use parallelism? Any experience with CPUs?

A. All experimental results we presented are from running on an NVIDIA V100 GPU with as much thread parallelism as is needed to saturate the available memory bandwidth. On CPUs, our library take advantage of vectorization (both for compute and conversion), and multicore multithreading.

Q. Do any of the software packages involve any shared code?

A. The question is not clear to me. Ginkgo is publicly available:
https://github.com/ginkgo-project/ginkgo
And MAGMA is available at https://icl.utk.edu/magma/software

Q. What is the difference between Gingko and MAGMA?

A. Ginkgo is C++, MAGMA is C and C++. Ginkgo has full support for AMD, NVIDIA, and OpenMP in one library. MAGMA supports AMD, Intel, and NVIDIA accelerators. Ginkgo is focused on sparse matrices. MAGMA is focused on dense matrices, batched, but also supports sparse matrices on a single and multiple accelerators. Ginkgo is for iterative linear system solves. MAGMA has linear system solvers, least-squares solvers, eigenvalue solvers, singular values solvers, and iterative solvers. MAGMA covers a large portion of BLAS and LAPACK and related auxiliary routines.

Q. Are there methods to mixed precision with explicit type ODE solvers? The number of iterations on these problems may be a problem.

A. I am not aware of that. I suggest you contact the team behind SUNDIALS library for more details: https://computing.llnl.gov/projects/sundials

C. There is also libxsnm for multi precision matrices. (I think whoever posted this answer meant to say libxsmm located at github.com/hfp/libxsmm. But there are no solvers in the library, rather, just a matrix-matrix multiplies and DNN kernels that can be plugged in directly to other solvers in fixed- and floating-point precisions.)

Q. Instead of choosing between precision formats, wouldn't it be easier to normalize the data and store one additional scaling factor (even in double precision)?

A. Experiments indicate this is not a more attractive strategy. Especially as scaling all entries does not remove the need to have a significant length allowing for the condition number not to destroy regularity. Also, consider a multitude of available scaling methods: column and scaling. Balancing for eigenvalue problems is also a possible solution.

Q. How expensive is the function call gko::precision_reduction::autodetect()?

A. Moderate overhead - at most 50% overhead to the generation of the preconditioner.

Q. Going back to the roofline model, you showed that the low precision offers speedup but reduces arithmetic intensity? Don't we want high arithmetic intensity on modern HPC systems? So how to maintain a balance?

A. Yes, we would want high arithmetic intensity. But the algorithms often have low arithmetic intensity, in particular sparse linear algebra algorithms. We can't really do anything about this, but have to live with it.
And no, lower precisions do not lower arithmetic intensity in principle. The lower precisions have a different roofline because of the hardware's peak performance and bit-length.

Q. How important is base language support for reduced precision? How much time for example is wasted on conversions on CPUs?

A. Difficult to generalize. But it definitely is an important aspect - we need parallel high performance implementations for these conversions.

Base language support (which I assume stands for native data types for lower precisions) help with portability. But we still need to write code for each precision and templating is unlikely to cover corner cases when the floating-precision format runs out of range or significand's bits.
Conversion is almost always handled transparently by hiding it behind other operations such as transfers to the accelerator or pipelining with the computation.

Q. Would it help to run most iterations with single precision, then finally converge with higher precision?

A. See answer above. It depends on the method.