Solving Large Quantities of Small Matrix Problems on Cache-Coherent SIMD Architectures

Bryce Lelbach, Hans Johansen, and Samuel Williams
Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720
{*balelbach, hjohansen, swwilliams*}*@lbl.gov*

***Abstract*—A number of computational science algorithms lead to discretizations that require a large number of independent, small matrix solves. Examples include small non-linear coupled chemistry and flow systems, one-dimensional sub-systems in climate and diffusion simulations, alternating direction implicit preconditioners, and semi-implicit time integrators, among others. We present a performant approach for solving large quantities of independent matrix problems on cache-coherent SIMD architectures. Unlike many vectorized or "batched" approaches that rely on reusing the matrix factorization across multiple solves, our algorithm supports the case of sets of matrices that are different, due to spatial variation or non-linear solvers, for example. We demonstrate the approach with a prototypical tridiagonal matrix solver, derived from a 1D solver that is part of an implicit-explicit finite difference discretization for a 3D advection-diffusion problem. Performance is evaluated on several Intel architectures with different cache, vectorization, and threading features, and compared to theoretical roofline models across parameter studies. We conclude that the approach is effective at optimizing both vectorization and memory bandwidth, and improves on existing approaches for efficiently solving large numbers of small matrix problems.**

I. Introduction

One important class of problems in computational science is the solving of smaller-dimensional matrix subproblems that are duplicated across many degrees of freedom in a larger two or more dimension computation. Several examples of this include:

- Pointwise chemistry systems in the context of a larger, flow simulations. Examples include geochemistry [**?**], cloud microphysics [**?**], and combustion [**?**];
- Solving one-dimensional systems that represent a "primary" direction for a physical phenomenon. Examples here include atmospheric radiation [**?**], groundwater penetration [**?**], or models for cloud convection [**?**]; and
- Implicit solvers that need to couple these kinds of subsystems, such as physics-based preconditioners [**?**], alternating direction implicit ADI, [**?**], and operator-split or semi-implicit time integrators [**?**].

In most cases, these matrices are relatively small (ranging from $O(10-100)$ chemistry components or "levels" in a climate application), and may be sparse or dense, but must to be solved repeatedly, but with different entries each time, to advance the overall simulation. Thus, because these are often non-linear matrix systems with space- and time-dependent entries, these applications may not use a "factor once, solve many times" approach, which is often used as a model matrix performance test. This prevents amortizing setup and factorization costs across multiple right-hand side solves as in *dgxxxxx* [**?**], and also challenges SIMD vectorization due to the odd size and dissimilar entries of the matrices, as well as the memory access patterns relative to the bigger simulation data layout. In that case, it is usually sub-optimal on many-core SIMD or SIMT GPU architectures, to just call an optimized linear algebra library, such as Intel's MKL version of LAPACK [**?**] or NVIDIA's cuBLAS [**?**]; these may not achieve peak memory bandwidth and VPU performance across the range of small matrix size. In that sense, it can leads applications to create their own custom implementations, which may not be optimal, and create a (potentially unnecessary) maintainance burden for the applications running across multiple many-core architectures as well.

To this end, we have developed a model matrix kernel that mimics what is encountered in these kinds of large-scale simulations. Key aspects of the code include:

- Matrix systems that must be created and solved at each point in a two-dimensional subdomain (represented by (i, j) indices) of a three-dimensional application (that is, (i, j, k) indices).
- Each matrix is tri-diagonal, and must be solved for all $O(30-100)$ values in the k index.
- The matrix is derived from finite difference discretization for the 1D diffusion equation, which allows it to be solved without pivoting (pivoting will be addressed in future work).

II. Related Work

Approaches to solving large numbers of small matrices have been done in a variety of contexts. Many implementations simply solve each matrix in parallel or for multiple right-hand sides, using platform-specific implementations of LAPACK, such as Intel's Math Kernel Library [**?**] or NVIDIA's cuBLAS [**?**] implementations. In many cases, there is a benefit from vectorization and thread parallelism, but there may be overheads that deteriorate performance, such as data copies into local arrays, dynamic determination of optimal performance parameters, partially vectorized"peel" loops, etc. [**?**]. Some specialized approaches include specialized linear algebra-specific compilers for small problem sizes and target architectures [**?**], [1] **(add build-to-order ref, Siek/Jessup?)**. Other libraries like *Blaze* [2], *PLASMA* [3], *MAGMA* [4], and *libxsmm* [**?**] are intended to support SIMD vectorization

for standard vector sizes as well as batched computation for sparse and dense matrices. Overall, there is a gap in approaches that both have vectorization, regardless of matrix size, with or without dense/sparse/pivoting assumptions, amortizing factorization across multiple right-hand-sides, and other assumptions.

(Hans - fill in more background / details on these.)

III. Implementation

Bryce to write

For a positive-definite tridiagonal system, a simplified form of Gaussian elimination can be used, which does not perform any pivoting. This method is known as the Thomas algorithm [], and it is $O(n)$ in time, a significant improvement over full Gaussian elimination for a completely dense matrix, which is $O(n^3)$ in time.

$$\begin{bmatrix} b_0 & c_0 & & & 0 \\ a_1 & b_1 & c_1 & & \\ & a_2 & b_2 & ... & \\ & & ... & ... & c_{n-2} \\ 0 & & & a_{n-1} & b_{n-1} \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ ... \\ ... \\ u_{n-1} \end{bmatrix} = \begin{bmatrix} u_0 \\ u_1 \\ ... \\ ... \\ u_{n-1} \end{bmatrix}$$

Forward elimination:

```
for (index_type k = 1; k < n; ++k) {
    double const m = a[k] / b[k - 1];
    b[k] = b[k] - m * c[k - 1];
    u[k] = u[k] - m * u[k - 1];
}
```

Backward substitution:

```
u[n - 1] = d[n - 1] / b[n - 1];

for (index_type k = n - 2 ; k >= 0; --k) {
    u[k] = (u[k] - c[k] * u[k + 1]) / b[k];
}
```

TODO: Mention that algorithm is in-place and destroys the input matrix (this is ok, because of reforming the matrix for the non-linear iteration). Describe AI.

In the applications described in Section **??**, we perform such a solve on each vertical (e.g. z) "column" in a 3D Cartesian grid (Figure **??**). The matrix coefficients for each column depend on the problem state, so a unique matrix for each column needs to be constructed before each solve. There are two different approaches to computing these batch solves:

- **Solve Columns Independently**: The most straightforward approach is to deal with each tridiagonal solve separately, independent of the other solves. An nzxnz matrix is constructed for each column, and then the Thomas algorithm is used to solve the system formed by the matrix and the vertical column. Because the column solves are independent, different column solves can be executed concurrently via task-level parallelism. Additionally, vectorization can be attempted in the vertical (z) dimension for each independent solve.
- **Interleave Column Solves**: The alternative approach is to simultaneously solve multiple columns. For each tw_xxtw_y horizontal tile of the grid, a tw_xxtw_y subgrid of nzxnz matrices (e.g. a 4D grid stored in a contiguous memory region) is constructed, and then a modified Thomas algorithm is used to simultaneously solve tw_xxtw_y independent vertical columns. Each tile is solved independently, so the tile solves can be executed concurrently via task-level parallelism. Data-level parallelism can be applied in one of the two horizontal dimensions (x or y).

TABLE I: **Comparison of Mixed-Precision and Double-Precision Algorithms**: To study the trade-off in performance and accuracy between the mixed-precision and double-precision variants of our algorithm, we solved a 1D diffusion problem with both codes on a single Intel Xeon ??? "Ivy Bridge" core. We used two measures to quantify error: the L2 norm of the difference between the analytic solution and the computing solution, and the absolute maximum of the residual vector of the tridiagonal solve. The test problem had storage requirements of approximately 4.3 GB. 10 sample runs were performed with each sample performing 5 time steps. The walltime and uncertainty metrics below are normalized to the double-precision results; the L2 norm and residual metrics specify order of magnitude.

Algorithm	Walltime (Normalized)	L2 Norm	Residual
Double-Precision	1.0 ± 0.007	O(1e-07)	O(1e-16)
Mixed-Precision	0.836 ± 0.006	O(1e-06)	O(1e-08)

The interleaved approach offers a major benefit over the independent approach; it allows vectorization in one of the horizontal dimensions. The independent approach is restricted to vectorizing in the vertical dimension. The extent of the vertical dimension (nz) tends to be very small in the applications we are concerned with (**HOW SMALL**). In this approach, the loops we are trying to vectorize will typically have iteration counts between 10 and 100. If we vectorize in one of the horizontal dimensions, we can control the iteration count of the loops via the tile size. Additionally, we cannot vectorize the forward-elimination loop, because it contains a read-after-write dependency (**ELABORATE**).

Thus, we pick one of the horizontal dimensions as our unit stride dimension. The vertical dimension will have the greatest stride.

IV. Experimental Setup

Bryce to write Edison [5], Cori, etc... Compilers Problem Size

- 3x3 or 30x30 or 400x400
- Vectorized vs. MKL / non-vectorized
- Mixed precision
- Tiled (experiment for non-tiled code blowing out cache)

V. Results and Analysis

Bryce to write

VI. Conclusion

Bryce to write The conclusion goes here.

Acknowledgments

Comment this section out before submitting... Reformat before finalizing... Intel IPCC ack, too

This research used resources in Lawrence Berkeley National Laboratory and the National Energy Research Scientific Computing Center, which are supported by the U.S. Department

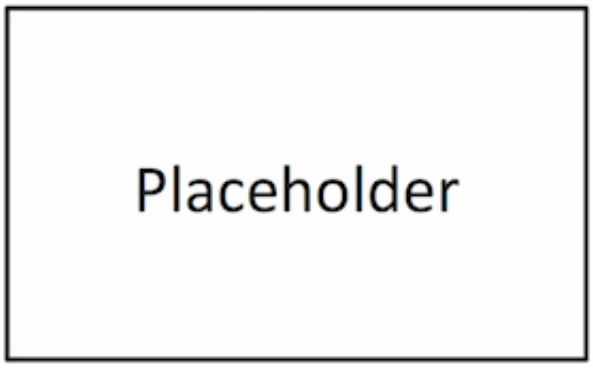

Fig. 1: **(Add the figure)** Baseline performance using MKL (and hand) using i-major data layout (not vectorized), for KNL, HSW.

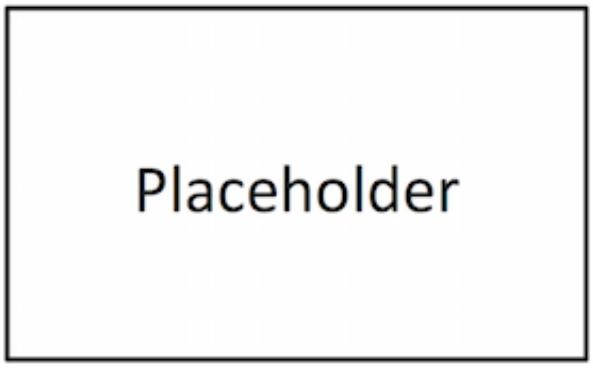

Fig. 2: **(Add the figure)** Performance as a function of 32b RCP NR (not needed on KNL), stored reciprocal (cuts divides in half) for fixed file size (4?)

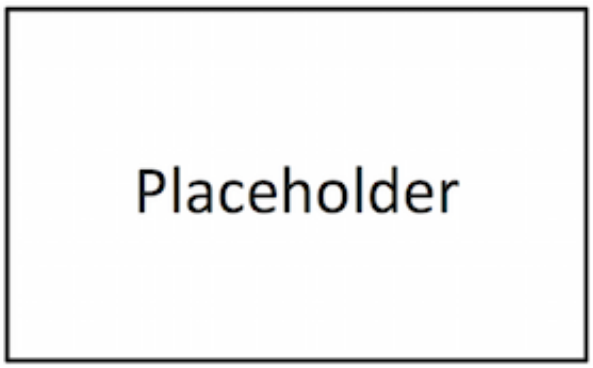

Fig. 3: **(Add the figure)** Performance vs. Tile Size (jtile = 1,2,4,8,16,32)

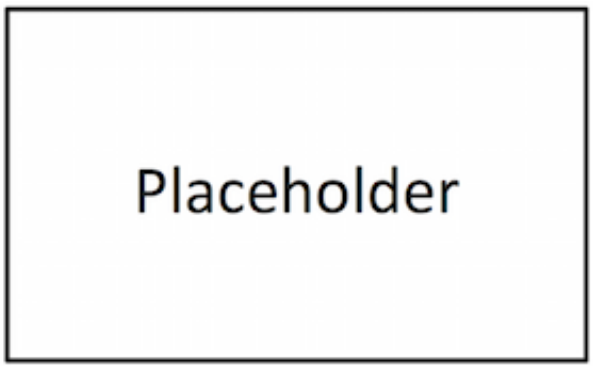

Fig. 4: **(Add the figure)** Best Performance vs. MKL using same data layout vs. MKL using its best (include DRAM BW limit)

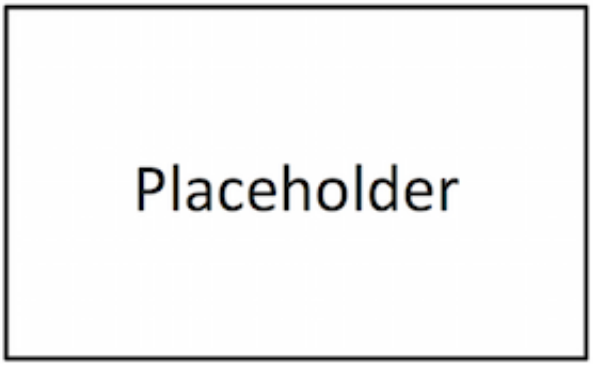

Fig. 5: **(Add the figure)** Performance as a function of total parallelism for constant tile size [optional time permitting]

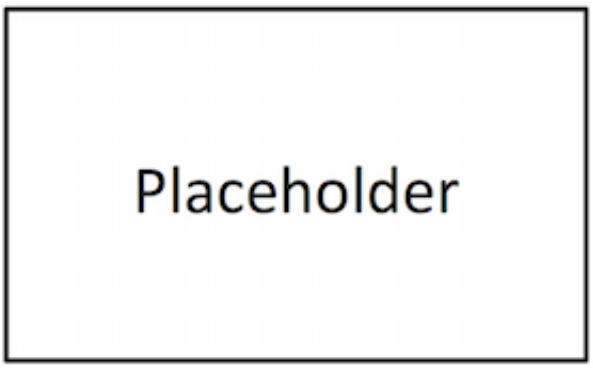

Fig. 6: **(Add the figure)** Performance with 2,3,4 hyperthreads... maybe just prose comments?

of Energy Office of Science's Advanced Scientific Computing Research program under contract number DE-AC02-05CH11231. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program.

REFERENCES

[1] D. G. Spampinato and M. Püschel, "A basic linear algebra compiler," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. New York, NY, USA: ACM, 2014, pp. 23:23–23:32. [Online]. Available: http://doi.acm.org/10.1145/2544137.2544155

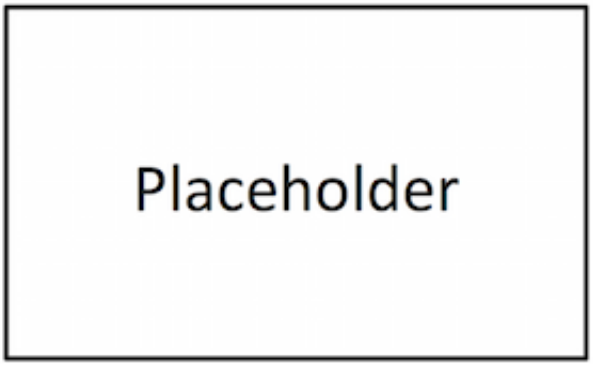

Fig. 7: **(Add the figure)** effect of kdim != pow(2)... maybe just prose comments?

[2] "Blaze:," https://bitbucket.org/blaze-lib/blaze.
[3] "PLASMA:," http://icl.cs.utk.edu/plasma.
[4] A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. Dongarra, "Optimization for performance and energy for batched matrix computations on gpus," in *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, ser. GPGPU-8. New York, NY, USA: ACM, 2015, pp. 59–69. [Online]. Available: http://doi.acm.org/10.1145/2716282.2716288
[5] "Edison:," www.nersc.gov/users/computational-systems/edison.

NOTES/OUTLINE

Submission info: https://easychair.org/conferences/?conf=pmbs16

- Climate apps use HE-VI model 3D+1D. Also a pattern for 2D+1D and 3D+0D chemistry
- Results demonstrate importance of "batching" solves, MLK doesn't do
- Cache coherency is important: IMEX RK accum, tiling
- Explicit part is HO stencil op, memory b/w bound (w/ ghost cells)
- Implicit part is non-linear solver: App requires different matrix at each i,j index
- Results in repeated vertical sparse banded solve
- Solve (tridiag, banded, dense) should vectorize on i (unit stride), tiled in pencils
- Considerations for vector alignment, tiling, and memory
- Performance results, scaling, comparison, by platform / parameter
- Future work: communication hiding, load imbalance, IMEX

(Bryce's email comments) Looking over the outline:
It's crucial that we clearly identify the optimizations that we believe are novel/high-impact, vs the optimizations that are well-known to the community (even if they were not well known to us). For example, on the outline, we have listed "IMEX RK accum, tiling". Do we want to present the accumulation optimizations which removed unnecessary temporaries from the time integrator? Do we feel that optimization is novel, or is that just a common-sense thing that only affected us because of the Chombo programming model? Likewise, do we want to present tiling as a focal point in this paper? Tiling is a well-known technique; we certainly can't claim tiling in general as novel work. Is some part of our tiling approach novel? (how about parallelizing the tile loop - is that fairly novel? I know Sam does this in HPGMG, but is this a widely used technique?)
We would be well served by applying the scientific method at this juncture:

- Question: What concrete research question(s) are we answering?
  - Prior research indicates that HE-VI methods show promise as a scalable approach to solving global climate problems on cubed sphere geometries because ¡explanation¿ [cite prior studies]. How do we implement HE-VI methods which perform well on cache-coherent SIMD architectures?
  - When solving a problem with a high horizontal-vertical aspect ratio (e.g. the horizontal extents are much greater than the vertical extents) with HE-VI methods, it is necessary to perform a large number of small vertical implicit solves (¡give examples of matrix sizes¿ [citation]) which are independent of each other. ¡explain the type of solves in the climate dycore - e.g. non-linear but nearly banded¿ [citation]. How can we efficiently solve large quantities of small non-linear bandedish/banded/tridiagonal matrices on cache-coherent SIMD architectures?
- Hypothesis: What theories did we come up with that would answer our research question(s)?
  - Optimal memory access patterns, management of working set sizes (e.g. staying in cache) and efficient use of vector units are necessary to achieve good performance on cache-coherent SIMD architectures.
    - Optimal memory access patterns: moving through memory in unit stride, controlling the number of streams.
    - Management of working set sizes: tiling, reducing size of temporaries/localizing temporaries (thread-local or otherwise)
    - Efficient use of vector units: moving through memory in unit stride, controlling memory alignment, controlling array strides, annotation-assisted autovectorization
    - TODO: List of individual, concrete optimizations from above.
  - Mainstream linear algebra libraries (¡examples¿ [citation]) are not well-suited for solving large quantities of small non-linear bandedish/banded/tridiagonal matrices on cache-coherent SIMD architectures because ¡expalanation¿ [cite prior studies if possible].
- Prediction: If our hypotheses are true, what results can we expect to see?
  → TODO: What performance characteristics should we see with and without each of the concrete optimizations from our hypothesis above?
- Experiment: Investigate the predictions we've made.
  → TODO: Methodology for benchmarking the optimizations identified above and measuring the performance characteristics we're interested in.
- Analysis: What were the results of our experiments?

List of figures from meeting on 8/11: For KNL, HSW, SNB(?)... List of figures from meeting on 8/11: -For KNL, HSW, SNB(?)...

1) Baseline performance using MKL (and hand) using i-major data layout (not vectorized)
2) Performance as a function of 32b RCP NR (not needed on KNL), stored reciprocal (cuts divides in half) for fixed file size (4?)
3) Performance vs. Tile Size (jtile = 1,2,4,8,16,32)
4) Best Performance vs. MKL using same data layout vs. MKL using its best (include DRAM BW limit)

5) Performance as a function of total parallelism for constant tile size [optional time permitting]
6) Performance with 2,3,4 hyperthreads... maybe just prose comments?
7) effect of kdim != pow(2)... maybe just prose comments?