International Journal of
HIGH PERFORMANCE
COMPUTING APPLICATIONS

Special Issue Paper

The International Journal of High Performance Computing Applications
2024, Vol. 38(6) 671–691
© The Author(s) 2024
Article reuse guidelines:
sagepub.com/journals-permissions
DOI: 10.1177/10943420241281050
journals.sagepub.com/home/hpc
Sage

Numerical eigen-spectrum slicing, accurate orthogonal eigen-basis, and mixed-precision eigenvalue refinement using OpenMP data-dependent tasks and accelerator offload

Piotr Luszczek[1,2], Anthony Castaldo[3], Yaohung M Tsai[4], Daniel Mishler[2] and Jack Dongarra[2,5,6]

Abstract

Performing a variety of numerical computations efficiently and, at the same time, in a portable fashion requires both an overarching design followed by a number of implementation strategies. All of these are exemplified below as we present transitioning the PLASMA numerical library from relying on dependence-driven large tasks to achieving utilization of fine grain tasking and offload to hardware accelerators while keeping its core dependence sets: OpenMP source code pragmas and runtime for most system-level functionality and basic low-level numerical kernels provided directly by hardware vendors or open source projects with vendor contributions. We also present new algorithmic methods and their efficient parallel implementations including fine grained tasking for eigen-spectrum slicing and offload for mixed-precision eigenvalue refinement. We provide performance, scaling, and numerical results showing sizable gains over the available solutions from either the open source and vendor-provided packages.

Keywords

Hardware accelerator offloading, numerical linear algebra, mixed-precision algorithms, OpenMP pragma directives, sturm sequence eigenvalue finding

1. Introduction

For a known numerical solver, using a specialized software library is one of the most productive ways to achieve high levels of efficiency on modern hardware platforms featuring specialized accelerators that require ever increasing levels of parallelism exposed by the user code. The upside of abstracting away the solver details comes with high expectations from scientific simulation community in terms of delivering state-of-the-art performance that rivals that of specialized techniques matched only by the most skilled programmers. But access to such a talent is problematic for even a single hardware type, let alone an entire set of platforms that now form the varied landscape of the national computing cyberinfrastructure. To both maintain the functionality surface and provide competitive performance, we use the standards-based portable design of a numerical library to show how these goals can be achieved in practice with long term sustainability in mind. In particular, we present the overarching design featuring descriptive, prescriptive, and even meta directives with judicious use of vendor-provided primitives, all of which serve our stated vision of both portable and performant numerical linear algebra with the machine precision accuracy. We back up our claims using a set of numerical experiments that are verified to deliver on all three fronts. We present the process of transitioning the PLASMA numerical library (Dongarra

[1]MIT Lincoln Lab, LLS, CMIT Lincoln Laboratory, Lexington, MA, USA
[2]Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA
[3]Technical Development, Synopsys, Inc., Sunnyvale, CA, USA
[4]ML Compilers and AI Accelerators, Meta, Inc., Menlo Park, CA, USA
[5]Computer Science and Mathematics, Oak Ridge National Laboratory, Oak Ridge, TN, USA
[6]Applied Mathematics, University of Manchester, Manchester, UK

Corresponding author:
Piotr Luszczek, University of Tennessee Knoxville College of Engineering, 1122 Volunteer Blvd #203, Knoxville, TN 37996, USA.
Email: piotr.luszczek@ll.mit.edu

et al., 2019) from its sole reliance on dependence-driven large tasks towards more comprehensive use of much more fine-grained facilities on the CPU as well as offload to hardware accelerators, primarily GPUs. Keeping the main goal of relying on OpenMP in the source code and at runtime for the vast majority of system-level interactions such as thread management, asynchronous task scheduling, and interacting with the OS kernel drivers to GPU software toolchains. The responsibility of providing the common high-performance low-level numerical kernels was relegated to hardware vendors or specialized open source projects that often receive much needed input from vendor-sponsored teams.

Our new contribution is a parallel algorithm implemented portably on top of the modern OpenMP constructs. Additionally, we contribute the efficient management of two main problems that often plague the instances with large n: occurrences of underflow/overflow and loss of orthogonality in the inverse iteration. The former necessitates autoscaling of the intermediate results in order to avoid underflow and overflow in eigenvalue computation, based on the Sturm sequences. The latter problem, originating in the inverse iteration used to find eigenvectors, is implemented in LAPACK's `DSTEIN()` in double precision, and in our tests could produce non-orthogonal eigenvectors for a symmetric matrix. We also address a known issue in LAPACK's `DSTEVX()` that always orthogonalizes the resultant eigenvalues using the Modified Gram-Schmidt (MGS) algorithm. This step dominates the execution time of `DSTEVX()` (and some of our algorithms), often accounting for over 95% of the runtime in our experiments. To orthogonalize the eigenvectors, we introduce the use of the specialized QR factorization for tall-and-narrow matrices with excess number of rows compared to columns. This also allows us to achieve higher speedup with much easier parallelization than is possible with MGS.

We also present a new implementation of mixed-precision eigen-pair refinement with explicit offload to GPU accelerators and show its performance across the hardware spectrum: CPU-only, low-end GPU, and high-end GPU. Our results indicate there is a need for heterogeneous parallelism to benefit from mixed precision and GPU acceleration.

2. Related work

The parallel algorithms for reduction of square symmetric or Hermitian matrices to condensed forms with similarity transformations while maintaining high utilization of multicore processors were pioneered in the PLASMA library (Haidar et al., 2011, 2014; Luszczek et al., 2011) and similar techniques are applicable to singular value calculations (Haidar et al., 2012, 2013; Ltaief et al., 2012). However, the tridiagonal eigen-solver remained elusive for these approaches with the LAPACK library (Anderson et al., 1999) and its sequential implementation providing the most common workaround justified by the low computational intensity. We address this missing piece of numerical diagonalization with a parallel algorithm and careful numerical considerations for tridiagonal matrices.

There are many numerical and middleware libraries that provide eigenvalue algorithms with DPLASMA (Bosilca et al., 2011) and Chameleon (Agullo et al., 2012) being prominent out of a few recent efforts. They approach the computational problem of eigenvalue solvers from a large scale and multi-GPU perspective while we aim at the small scale of a single node and single accelerator to deal with common problem sizes. A similar scale difference exists with the SLATE library (Gates et al., 2019), that in addition to targeting large distributed memory systems with many accelerators, also relies on industry standards including OpenMP for portability of performance critical aspects of its design.

One way to recursively compute the selected eigenvalues of a matrix is with the *bisection eigenvalue algorithm* (Demmel et al., 1995) that also applies for bidiagonal matrices resulting from SVD (Gu et al., 1994). With the mild assumption that the floating-point arithmetic is monotonic, these methods are accurate and numerically stable (Demmel et al., 1995). For example, monotonic summation satisfies the property that for any $x < \bar{x}$ and $y < \bar{y}$ implies $\mathrm{FP}(x + y) \le \mathrm{FP}(\bar{x} + \bar{y})$, which is requested by the IEEE 754 standard but may be violated in hardware (Fasi et al., 2021). The natural splitting of work in the bisection leads to parallel approaches (Volkov and Demmel, 2007) based on a divide-and-conquer scheme (Gu and Eisenstat, 1995). We extend this work to allow for the range-based parallelization and dynamic work balancing that is data-driven based on the eigenvalue-count within the divided spectrum “slices”.

There are multiple orthogonalization schemes and, in the context of the Krylov subspace iteration, and more specifically in the Arnoldi procedure of the GMRES method, two methods stand out: CGS (Classical Gram-Schmidt) and MGS (Modified Gram-Schmidt). The former is often used due to its practical advantages such as a lower computational cost and data-access overhead (Paige and Strakos, 2001). On the other hand, the latter scheme retains orthogonality at a better rate and may be simply implemented using matrix-vector products (Giraud et al., 2005a). For CGS, reorthogonalizing only once is sufficient in practice to provide numerical stability (Giraud et al., 2005b). However, both of these methods suffer from exposure to the bandwidth bottleneck because they are memory-bound. An orthogonalization scheme may present issues in situations, for example, with rank-deficient matrices, for which the orthogonal basis vectors, contained in the matrix customarily named Q, could be arbitrarily far from orthonormality when the Gram-Schmidt process is used (Daniel et al., 1976; Kiełbasiński, 1974).

3. Portability strategies

We use a multi-pronged approach to achieve portable performance across a wide range of hardware targets, some of which were used to obtain the results presented below. First off, we opted for a judicious choice of only a subset of features used from the supported programming languages: C (ISO/IEC9899, 2023), C++ (ISO/IEC14882, 2023), and Fortran (ISO/IEC1539-1, 2018). Another aspect of our approach is to delegate the majority of the system-level programming to OpenMP (Architecture Review Board, 2021), which includes thread management, asynchronous task scheduling, offload to hardware accelerators, and management of disjoint memory spaces. Finally, we use portability layers for low-level numerical computations such as CBLAS and LAPACKE in C or BLAS++ and LAPACK++ in C++. We discuss these further in this section.

It is a generally safe assumption that both C and C++ share the same backend compiler passes while the Fortran toolchain is included in the system's linker infrastructure and dynamic loading of binaries. With this in mind, it is a safe portable method of sharing symbols and functions using the C linkage available in C++ through the extern "C" statement and with ISO C bindings introduced in Fortran 2003 and further extended in Fortran 2008. Limited attention has to be paid to the corner cases of the base languages interacting with OpenMP compiler and runtime. Notable examples include avoiding the use of virtual methods relying on compiler-generated v-tables in C++ or using assumed-type dummy arguments in Fortran. While the former is explicitly prohibited by the OpenMP standard and may easily be flagged by the compiler, the latter may result in undefined behavior, which is altogether more subtle and thus much harder to detect, especially at user's site if reported as a bug. At the same time avoiding some features such as C++ modules is easy due to their limited implementation availability across the vendor and compilation platforms. This also includes complex C++ concept constructs but we still intend to use exposition-only concepts as very productive documentation aids.

The portability contribution of OpenMP is hard to overstate both in its span across the hardware platforms and durability throughout decades spanning vector, RISC, multicore, and now GPUs. However, the fast-paced growth of the standard contributed to its somewhat uneven state of its implementations. This creates a different kind of challenge to probe, detect, and guard against specific OpenMP features either not implemented, partially available, or simply incorrect. A primary example is often used clause default(none) that requires scope sharing specification to be provided for every variable in OpenMP region. However, different versions of GNU C Compiler would either require for-loop iteration variable to be included in private clause or be left out completely as per the standard's default designation as private for all participating threads. This leads to a portability strategy shown in Figure 1(a) that shows incremental building of the pragma string during compilation by using the standard _Pragma() macro. Figure 1(b) shows a way for the aforementioned dispatch to vendor-specific numerical kernels with meta-directives. Figure 1(c) is a trivial example that replaces memory allocation and transfer routines with OpenMP equivalence. Finally, Figure 1(d) shows how accelerator-specific features can be leveraged for optimization with asynchronous numerical kernel processing.

4. Motivation for eigen-spectrum slicing

In scientific applications that use $n \times n$ matrices to represent equations of a quantum state and, for some of them, n can grow extremely large for many-particle systems (Löwdin, 1955): as large as tens of thousands or millions of rows and columns. While the roots of the characteristic equation $\det(A - I\lambda_i) = 0$ are the eigenvalues λ_i of matrix A, in practice, however, generic root-finding is neither as numerically stable nor as performance-efficient as the more matrix-based specific approaches. The resulting eigenvalue-eigenvector pairs, or eigen-pairs for short, represent salient information from the field where the matrix originated. More concretely, in quantum chemistry or condensed matter physics (Ando, 1963; Carlson and Keller, 1961) the eigenvalues represent energy levels while the eigenvectors represent the coefficients of the corresponding wave function in a suitable basis. Diagonalization of the specific and large Hamiltonian matrices could also be used to determine entanglement of spinless Fermions (Barghathi et al., 2017).

We assume the standard formulation of the symmetric or Hermitian eigenvalue problem for the ith eigenpair: $Av_i = \lambda_i v_i$, where $A \in \mathbb{R}^{n\times n}$ and $\lambda_i \in \mathbb{R}$ for $i \in \{1, \ldots, n\}$. When considering complex Hermitian matrices, the eigenvalues are still real and our results carry-over naturally. In block form, this is represented by grouping the eigenvalues on the diagonal and gathering the eigenvectors to form the similarity transformation into the following diagonal form: $AV = \Lambda V$, where V is unitary: $V^* V = I$.

When n is large, the cubic complexity of $\mathcal{O}(n^3)$ makes the compute time and memory cost of finding all n eigenpairs prohibitive, or just not possible. Fortunately, the domain scientists are often interested in just a thin "slice" of the full range where the eigenvalues reside. This naturally motivates the *spectrum slicing* approach: it finds eigenvalues from just a limited range, with their corresponding eigenvectors computed subsequently with other methods. The slice contains k eigenpairs, with $k \ll n$. The serial implementation is available from LAPACK as subroutine

a) Conditional use of clauses.

```
#ifdef USE_DEFAULT_NONE
#define OCLAUSE "default(none)"
#else
#define OCLAUSE ""
#endif
_Pragma("#pragma omp parallel for" OCLAUSE)
```

b) Dispatch to device specific numerical kernels.

```
int dev = omp_get_default_device();
double *a = mkl::malloc(a_ld * n * 8, 64);
#pragma omp target data map(to:a,b) map(tofrom:c)
{
#pragma omp target variant dispatch \
  use_device_ptr(a,b,c) device(dev) nowait
mkl::dgemm(tA, tB, m, n, k, alpha, a, a_ld, b, b_ld, beta, c,
c_ld);
#pragma omp taskwait
}
```

c) Hiding accelerator-specific memory allocations and transfers.

```
double *a = omp_target_alloc(device, a_ld * n * 8, 64);
omp_target_memcpy(..., a);
omp_target_free(a);
```

d) Leveraging accelerator-specific command queues.

```
#pragma omp target data map(a[0:n*n],b[0:n*n]) \
  map(alloc:c[0:n*n])
#pragma omp target data use_device_ptr(a,b,c)
{
cudaStream_t omp_stream = \
  (cudaStream_t) omp_get_cuda_stream(dev);
cublasSetStream(handle, stream);
cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, \
  m, n, k, &alpha, a, a_ld, b, b_ld, &beta, c, c_ld);
}
```

Figure 1. OpenMP strategies for portable code across various toolchain implementation. (a) Conditional use of clauses. (b) Dispatch to device specific numerical kernels. (c) Hiding accelerator-specific memory allocations and transfers. (d) Leveraging accelerator-specific command queues.

`DSTEVX()`, which implements, among its other available functional options, the spectrum slicing procedure.

5. Methodology for eigen-spectrum slicing

Like LAPACK's[1] `DSTEVX()`, we use the Sturm sequence and bisection to first discover all of the eigenvalues in the given range. The Sturm sequence $\mathbb{S}(u)$ is a function with complexity $\mathcal{O}(n)$ for an n-by-n matrix. It processes the values on the diagonal and off-diagonals of the tridiagonal matrix $T \in \mathbb{R}^{n \times n}$ to compute the number of the k possible eigenvalues that are less than u, and returns the integer k to the user.

We use $\mathbb{S}(u)$ in the *bisection* method. We are given a half-open interval as $[l_g, u_g)$ to find eigenvalues in the eigen-spectrum that the user is interested in. $\mathbb{S}(l_g)$ yields the number of eigenvalues less than l_g, and $k = \mathbb{S}(u_g) - \mathbb{S}(l_g)$ is the number of eigenvalues k that are in the eigen-slice. Our data structure called *bracket* consists of a lower bound l and an upper bound u, within which lie a count of k eigenvalues. This is sufficient information to prepare the output: a vector of k eigenvalues, a vector of k corresponding multiplicity counts, and an $n \times k$ matrix of orthogonal eigenvectors.

We subdivide the range by computing a mid-point $p_m = (l + u)/2$ and then $t_p = \mathbb{S}(p_m)$. If $0 < t_p < k$, we can replace the bracket with two brackets, a left and a right one. On the left, setting $u = p_m$ gives us a bracket with t_p eigenvalues in it; and on the right setting $l = p_m$ gives us a bracket with $(k - t_p)$ eigenvalues in it. If $t_p = 0$, we do not require a split: we can just replace l with p_m. If $t_p = k$, we do not require a split either: we can just replace u with p_m.

We process these replacements in a similar pattern that naturally lends itself to computing with recursive calls. Ultimately, since we know there are k eigenvalues in the user's range, we will end up with at most k brackets (or less if any eigenvalues have larger than 1 duplicity)[2]. If we continue to subdivide, then, eventually, in finite precision arithmetic, we will reach the limit of the machine precision for that exponent, or ULP (the unit at last place): $t_p = (l + u)/2$ will result in either $t_p = l$ or $t_p = u$.

At that point, we have found an eigenvalue to machine precision: $u = l + ULP$. This means that in $[l, l + ULP)$ there are k eigenvalues. But the upper bound of this range is open, so l is the most precise number we can get for an eigenvalue of multiplicity k. If $k = 1$ then we only indicate finding of a single eigenvalue. If $k > 1$ then we indicate a single eigenvalue with multiplicity k or multiple eigenvalues within ULP of each other that cannot be distinguished within the limits of the working precision.

Finding a single eigenvalue by *bisection* should be an $\mathcal{O}(n \cdot \log(range))$ operation; with $range = u_g - l_g$. We narrow *range* by a factor of two with each bisection. Finding all k eigenvalues within *range* has complexity $\mathcal{O}(k \cdot n \cdot \log(range))$. Having found an eigenvalue, we then invoke LAPACK's `DSTEIN()` to compute, by inverse iteration, the unit eigenvector corresponding to l.

Then we can store this eigenpair in the return vectors and matrices directly in its proper sort order. Recall there are globally $\mathbb{S}(l_g)$ eigenvalues less than l_g, which we have already calculated, and we also already calculated $\mathbb{S}(l)$ for the current bracket, the difference between them, $\mathbb{S}(l) - \mathbb{S}(l_g)$ is the zero-relative index into the result vectors and matrices. This lets us directly store the eigenvalue, its multiplicity and eigenvector into the result arrays to be returned to the user.

5.1. Sturm sequence modification

For some matrices, the intermediate values produced in the Sturm function can grow or shrink monotonically as it processes the diagonal and off-diagonal elements; and for very large n, this can result in an underflow or overflow. Using a method from Zhang, (2003), we implement a Scaled Sturm sequence, a backward stable algorithm that automatically rescales the computation in progress while keeping it an $\mathcal{O}(n)$ computation.

```
Input: LaunchTask $(l_g, u_g)$
$l_0 \leftarrow l_g \mid u_0 \leftarrow u_g \mid \Delta_g \leftarrow$ TotThreads$^{-1}$
for $i \leftarrow 1 \ldots (\mathbb{S}(u_g) - \mathbb{S}(l_g))$ do
    $k_i \leftarrow \mathbb{S}(l_{i-1}) \mid n_i \leftarrow \mathbb{S}(u_{i-1})$
    $\Delta_i \leftarrow \max(\Delta_g, \text{ULP})$
    if $n_i - k_i \geq 1$ then // got at least one $\lambda$
        for $j \leftarrow 1 \ldots$ TotThreads do
            LaunchTask $(l_g + (j-1)\Delta_i, l_g + j\Delta_i)$
        end
    end
end
```

Algorithm 1: Recursive tasking code for finding eigenvalues in range (l_g, u_g) based on the number of threads.

```
Input: LaunchTask $(l_g, u_g)$
$p_m \leftarrow \frac{1}{2}(l_g + u_g)$ // middle point
if $p_m - l_g > ULP$ then // left half-slice
    if $\mathbb{S}(l_g + p_m) - \mathbb{S}(l_g)$ then // $\lambda$'s available
        LaunchTask $(l_g, l_g + p_m)$
    end
end
if $u_g - p_m > ULP$ then // right slice
    if $\mathbb{S}(u_g) - \mathbb{S}(u_g - p_m)$ then // $\lambda$'s available
        LaunchTask $(u_g - p_m, u_g)$
    end
end
```

Algorithm 2: Recursive tasking code for finding eigenvalues in range (l_g, u_g) based on bisection with stopping criterion based on Sturm sequence.

5.2. Parallelization of finding eigenvalues in a range

We parallelize using fine grain tasks using the standard OpenMP tasking constructs. In Alg. 1, we initially divide the user's *Range* of eigenvalues into TotThreads equal parts. As we will show in the testing section, this does not ensure an equal amount of work for each thread, because eigenvalues are not necessarily distributed uniformly. In Alg. 2, we use the mid-point technique described earlier to launch tasks for each half of the range as long as it is large enough.

5.3. Eigenpair search performance

Our testing was done on a 36-core machine: Intel Xeon Gold 6140 2.30 GHz CPUs with 25344 KB Level 2 cache. The configuration had hyper-threading enabled for the total of 72 OpenMPthreads. Some scaling experiments limit the number of threads but our final performance experiments use all 72 threads. This is counter-intuitive for purely compute-bound runs but we still obtain better performance with hardware threads (or hyper-threading) due to our mixed-bottleneck workloads.

Figure 2 shows performance for a single thread completing eigenpair discovery. The performance is scaled to $\mathcal{O}(n \cdot k)$. The slight decline indicates some non-linear advantage for size. Once we get to larger matrices, we require roughly 290 ns per $(n \cdot k)$ steps, with k being the number of eigenpairs in the user specified range.

Figure 3 shows parallel performance of eigenpair discovery. The performance is also scaled to $\mathcal{O}(n \cdot k)$. The massive decline indicates poor parallelism for 72 threads on relatively small matrices. At $n = 64\,000$ the number is 6.5 ns per $\mathcal{O}(n \cdot k)$ pairs.

Figure 4 shows parallel speedups in eigenpair discovery. They range from 2.3× up to 44×, for 72 threads. We show in the left bar the speedup averaged over 7 matrices, and in the right the speedup in the Glued Wilkinson matrix – a "difficult" matrix with large eigenvalue clusters. As expected, little parallelism is available for smaller matrices, while 44× is a reasonable speedup on our particular test machine. In other words, we managed to extract parallelism from a code with limited computational intensity with only $\mathcal{O}(n \cdot\ k \cdot \log(range))$ operations, which permits little data reuse in order to alleviate the memory system overheads.

We should note that, for large matrices, eigenpair discovery runtime is typically dwarfed by the time taken by the orthogonalization step that follows. This is because given k eigenvalues, eigenpair discovery is effectively an $\mathcal{O}(n \cdot k)$ algorithm, while the orthogonalization step is an $\mathcal{O}(n \cdot k^2)$ algorithm. For $n > 8000$ and relatively small k, say $k < 2500$ in our tests, eigenpair discovery is 60% of the runtime on the small end but declines to less than 10% around $k = 2500$. At $k = 7630$, eigenpair discovery is only 5.6% of the runtime.

6. Numerical testing of challenging eigenvalue problems

We use matrices proposed by Demmel et al. (2006) and Marques et al. (2008), including "121", Wilkinson, Clement, Legendre, Laguerre, Hermite, and GluedWilkinson. We also test with a Kahan matrix, similar to the "121" with a tiny value for the diagonal 10^{-5} instead of 2. Clement matrix is useful for verifying that any algorithm finds eigenvalues correctly at all; analytically it produces eigenvalues of $\pm\ (n)$, $\pm(n\ -\ 2)$, … . For closely spaced eigenvalues, we test with the Kahan matrix which has analytically computable eigenvalues that are also tightly packed.

More specifically, the eigenvalues of the Kahan matrix are given by the following algorithm, with x being the diagonalp

$$\lambda_k = \sqrt{x^2 + 4\cos\left(\frac{k}{(N+1)^2}\right)} \qquad k = 1..n/2 \qquad (1)$$

$$\lambda_{N+1-k} = -\lambda_k \qquad k = 1..n/2 \qquad (2)$$

$$\lambda_{(n+1)/2} = 0 \qquad \text{if } N \text{ is odd.} \qquad (3)$$

Other matrices are interesting for the distribution of eigenvalues in their range.

Figure 5 shows eigenvalues of the "121" matrix: all diagonals are set to 2, super- and sub-diagonals are set to 1. The eigenvalue distribution for the Kahan matrix is visually identical to this and not shown. Note the near exponential increase at the far ends. There are 2500 eigenvalues tightly

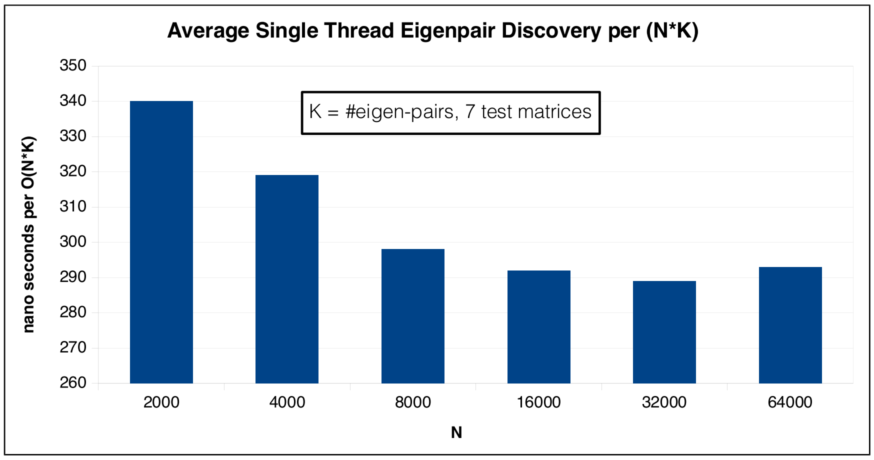

Figure 2. Single thread performance of eigenpair discovery with K on the order of $\mathcal{O}(N/10)$ depending on the user-provided value range for the 7 matrices we used throughout this study.

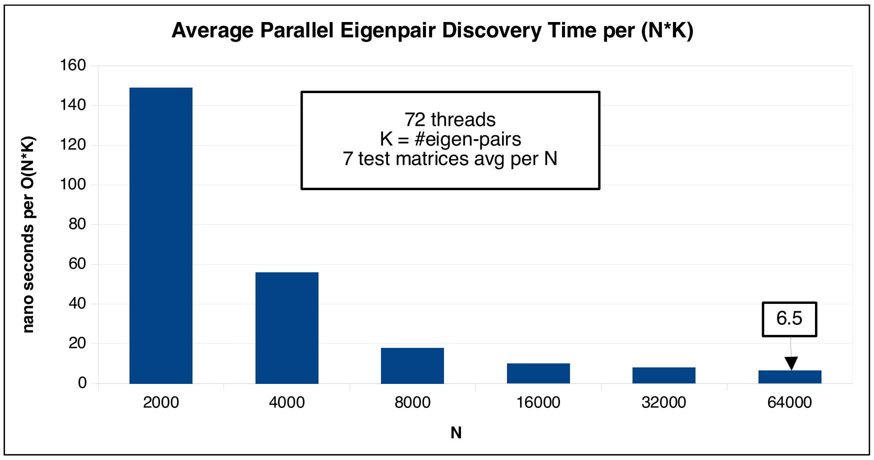

Figure 3. Parallel performance of eigenpair discovery.

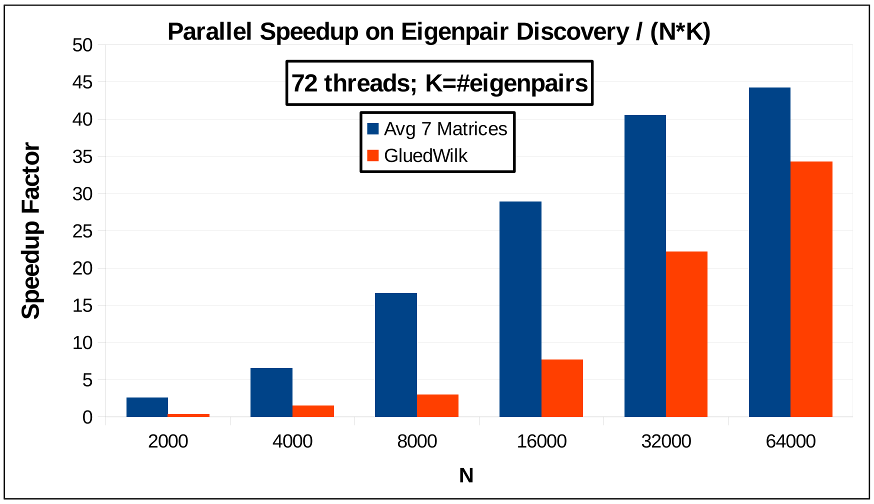

Figure 4. Parallel speedup in eigenpair discovery.

packed in (0, 0.04). Even if we zoom in on just that leftmost bar (not shown) it also looks like an exponentially declining term. This dramatically increases the density of eigenvalues. `DSTEIN()` finds nearly identical (but not quite) eigenvectors. As n increases, the non-orthogonality of the `DSTEIN()`'s vectors increases, meaning the dot-product of tightly packed eigenvectors is increasingly greater than zero.

Figure 6 shows the spectrum of the Hermite matrix with non-uniform density, but not dramatically so. The eigenvalues are most closely packed in the center, where our slice is. We use this matrix extensively for performance testing.

The spectrum of Laguerre matrix in Figure 7 shows a wide range of eigenvalues. The range increases proportionally to n, but the density is increasing very rapidly from right-to-left as we approach zero: 12.5% of eigenvalues are in 1% of the range. We zoom in on the near-zero part of this range in the second graph of Figure 7: it is just the first 10% of the first bar in the upper graph. It shows the same high density near zero. There are 50 eigenvalues in a millionth of the entire range of eigenvalues. With Laguerre matrix, we get a tight cluster of eigenvalues very close to zero. This is what makes a slice of Laguerre near zero ideal for testing orthogonality of eigenvectors and the relative eigenvector error (discussed later), this cluster in the neighborhood of zero is where these measures look the worst.

The Wilkinson matrix shown in Figure 8 looks nearly perfectly uniform, but in this matrix eigenvalues occur in extremely close pairs (clusters). The *clusters* are well separated by ≈ 1.0. The magnitude of dot-product of the eigenvectors for these close pairs approaches 1.0, but not quite, so they need to be orthogonalized. However, at large n, the Wilkinson pairs on the far right end becoming closer than ULP, and to our algorithm appear as a single eigenvalue with multiplicity 2; even though analytically (in infinite precision) we know they are distinct. The Glued Wilkinson matrix has a similarly uniformly

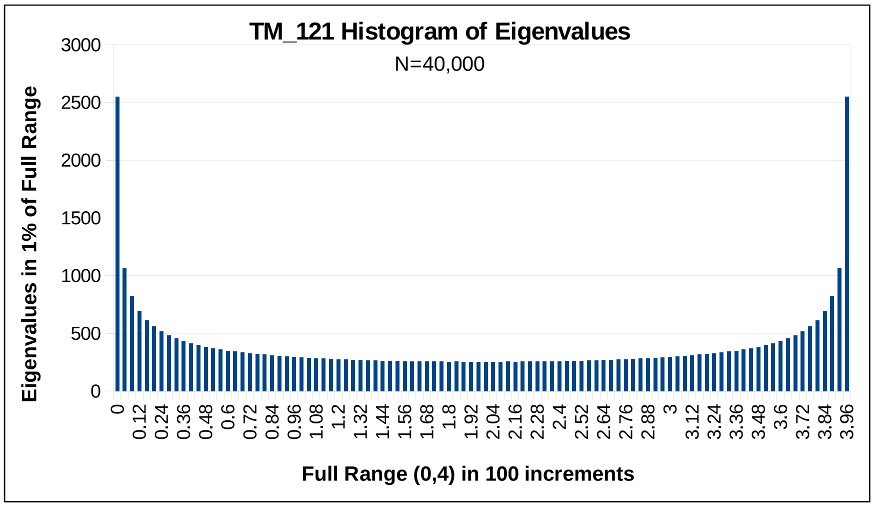

Figure 5. 100 step eigenvalue distribution over full range for matrix 121.

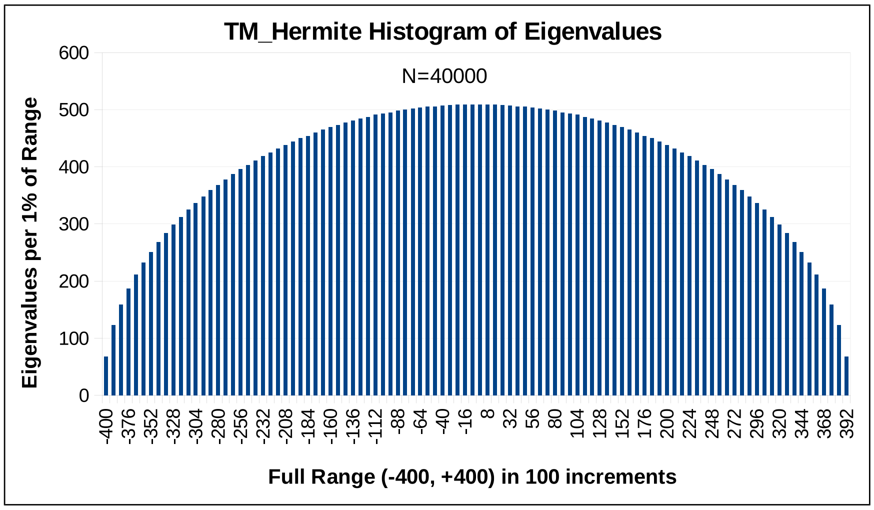

Figure 6. 100 step eigenvalue distribution over full range for Hermite matrix.

distributed spectral histogram, but it is designed to exacerbate the clustering property with groups of ten closely spaced eigenvalues.

The Legendre matrix spectrum from Figure 9 shows a tight packing of eigenvalues, in a relatively small range of $(-1, +1)$. In our experiments, we take the final slice on the right where the eigenvalues are most densely packed.

7. QR-based orthogonalization of eigenvectors

For our selected symmetric matrices, all eigenvectors are theoretically orthogonal in infinite precision arithmetic. Yet, `DSTEIN()` generates increasingly non-orthogonal eigenvectors as eigenvalues occur closer together. In Wilkinson matrices, eigenvalues occur in close pairs that grow increasingly closer as n increases. In Figure 10, we show the largest absolute dot product of any two eigenvectors as produced by `DSTEIN()`.

We see that the eigenvectors produced by `DSTEIN()` are nearly unit vectors with each other. They are slightly different, and can be orthogonalized, but `DSTEIN()` does not "naturally" produce nearly orthogonal vectors. Even with well separated eigenvalues, `DSTEIN()` can produce two eigenvectors with a dot-product of 10^{-10}, which can be corrected by orthogonalization to 10^{-16} or full double-precision accuracy ϵ_{64}.

The severity of this problem is unique to Wilkinson in our test suite, but the trend of losing orthogonality with increasing matrix size n occurs in all the matrices as we see increasingly tight packing in the eigenvalue distribution. Besides Wilkinson, our test matrices show this phenomenon for large n with "121", Kahan, Laguerre, and Legendre matrices.

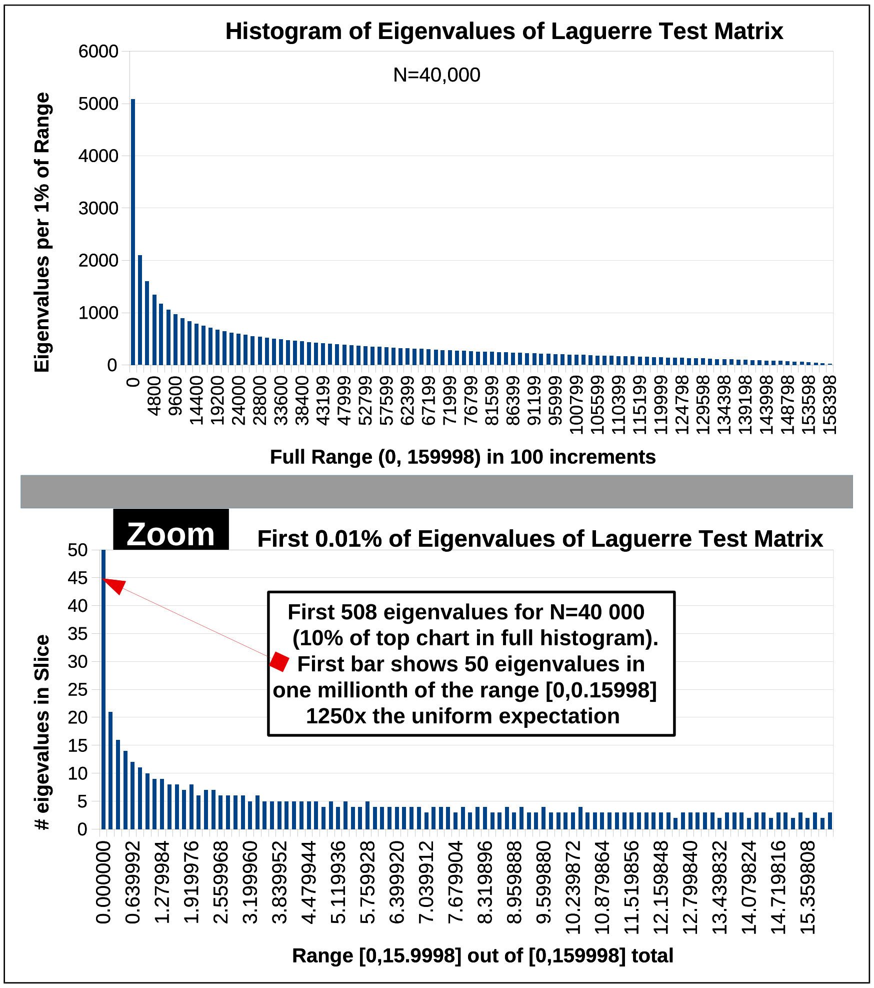

Figure 7. 100 step eigenvalue distribution over full range for Laguerre matrix.

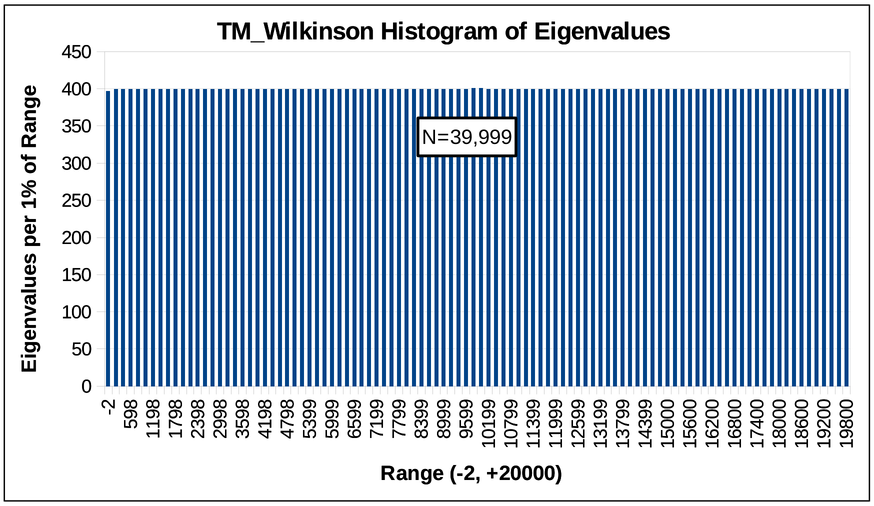

Figure 8. 100 step eigenvalue distribution over full range for Wilkinson matrix.

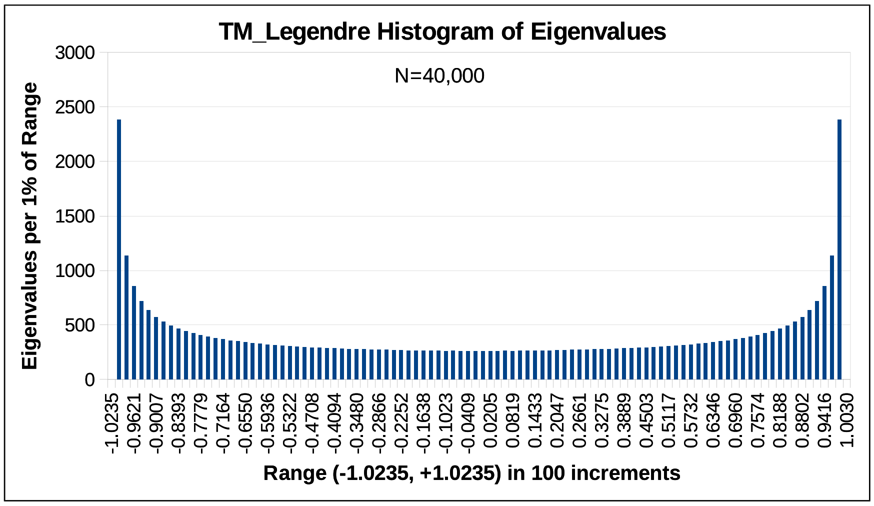

Figure 9. 100 step eigenvalue distribution over full range for Legendre matrix.

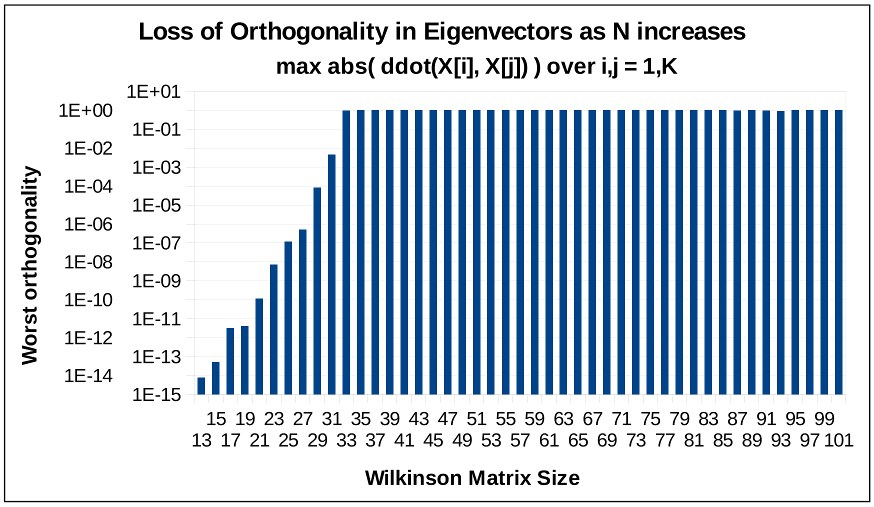

Figure 10. Orthogonality issues for Wilkinson matrix.

Thus, orthogonalization is a necessary post-processing operation. In LAPACK's `DSTEVX()`, this orthogonalization is accomplished with the Modified Gram-Schmidt (MGS) algorithm: each column of the matrix (one eigenvector) is orthogonalized against all prior columns and then normalized. For our largest test matrices, it may be 95% of the total runtime. In the MGS approach, a given eigenvector requires all previous orthogonalized eigenvectors and this data dependence produces a memory-bound sequential runs.

A second approach often used, and empirically better at orthogonalization, is the Classical Gram-Schmidt algorithm, applied twice (CGS-2). This requires approximately 1.5× as many flops, and suffers the same data dependency problem making parallelization difficult.

We implemented an alternative approach, using QR factorization, which is already available in PLASMA (Buttari et al., 2008). Our matrix of eigenvectors is $n \times k$, with n being the vector length, and k the number of eigenvectors found in the user's specified range. Since we expect $n \gg k$, our eigenvector matrix is very "tall" lending itself to specialized methods.

The classic QR factorization, as implemented in LAPACK's `DGEQRF()`, performs best for square matrices and produces Q and R as $n \times n$ square matrices for square inputs, with Q constituting an orthonormal basis of the columns of A, and R being an upper triangular matrix, such that $A = Q \times R$. But since in spectrum slicing n is very large, factoring an $n \times n$ matrix would require prohibitive memory and time. However, there exist specialized variants of the QR algorithms such as CAQR (Demmel et al., 2012, 2015), TSQR (Terao et al., 2020), or PAQR (Sid-Lakhdar et al., 2023) for this very purpose. They accept a matrix $A \in \mathbb{R}^{n\times k}$, and produce $Q \in \mathbb{R}^{n\times k}$ and $R \in \mathbb{R}^{k\times k}$. The workspaces required are affordably small. We regard the upper triangular R as a "scaling" matrix. We know $A = Q \times R$ (barring the roundoff error), and A is the original set of our eigenvectors. In particular, the ith column of A will be produced by $Q \times R_{1..n,i}$ (in 1-based indices, $R_{1..n,i}$ is the ith column of R). Only the first i elements matter, the rest are zero and each $R_{j,i}$ ($j = 1..i$) is the "scaling" of column $Q_{1..n,j}$, the jth eigenvector. These scaled eigenvectors are summed to produce the original eigenvector in column i of A, i.e. $A_{1..n,i}$.

In this application, the only purpose of R is to reconstruct the original A from the orthonormal basis delivered in Q. But we have no need of A, which are the original eigenvectors produced by `DSTEIN()`, so while it is necessary for the algorithm to form R, we can discard it once the specialized rectangular QR factorization finished.

The columns of Q are the orthogonalized normal vectors as we would get[3] using either MGS or CGS-2. Each column of Q is the orthogonalized version of the same column in A; and thus associated with the same eigenvalue.

There is a crucial performance difference: the specialized rectangular QR method is optimized and parallelized by several numerical libraries and reaps the benefits of using high performance implementations of Level 3 BLAS, which, generally speaking, are various matrix-matrix operations). We leverage that effort in our approach. In fact, using the high performance Level 3 BLAS alone is sufficient to warrant the use of any of the specialized QR factorizations for rectangular matrices.

In Figure 11, we show the performance of LAPACK's `DSTEVX()`. It is a sequential code executing on a single thread and thus it cannot take advantage of multicore hardware. Both `DSTEVX()` and the algorithm described in this paper are completely dominated by their orthogonalization step, using MGS and specialized rectangular QR, respectively. We can see that the former method tops out at 759 Mflop/s, for $m = 32,000$ with $n = 3392$ (#eigenvalues), and begins declining (likely due to Level 3 cache effects).

In Figure 12, we show the performance of the Spectrum Slicer with specialized rectangular QR orthogonalization available from Intel's MKL library, also in sequential mode for a fair comparison. As this is not using any parallelization at all, the matrix sizes and #eigenvalues remain identical. The Y-axis scale is 22× larger, this is due to using the BLAS Level 3 and cache-friendly blocking factors.

In Figure 13, we show the performance ratio at each problem size. We see a linearly improving speedup, from 5× to nearly 22× faster at $m = 60000$ (m is the number of rows in this graph; n is the number of columns being orthogonalized). On top of this serial speedup, we also show parallel speedup.

Intel's MKL library uses system threads and has `DLATSQR()` which produces the Q and R of the economic factorization in a packed form and `DORGTSQR()`, which then explicitly produces the $Q \in \mathbb{R}^{n\times k}$ matrix of orthonormal eigenvectors.

Figure 14, shows the average speedup we achieved using the fully parallelized version of our Spectrum Slicer implementation that has all the steps running in parallel. We use `DLATSQR()` for orthogonalization and compare against the vendor's LAPACK routine `DSTEVX()` which was the fastest available on the platform and included multithreading. The maximum speedup achieved was 156× for the matrix of size $m = 64000$. This high speedup number may be decomposed into multiple improvements over the reference sequential code. In the following analysis, we use the timing results for a matrix with $m = 64,000$ rows and only 10% columns or $n = 6400$. The first improvement comes from using efficient orthogonalization. For example, using specialized rectangular QR, a variant of QR well suited for our use case, instead of the MGS scheme as implemented by the LAPACK's `DSTEVX()` routine achieves about 20× speedup based on our estimates of the *single thread* runs of the tested machine. Further improvement in speedup comes from using the full parallelism of the hardware threads that compounds the speedup by 8× which is enabled by the purely algorithmic change. This gives us the total speedup of about $160 = 20 \times 8$. We hope to achieve even higher speedup for specialized rectangular QR by focusing on the potential bottlenecks of this QR variant that does not share the strong-scaling properties of the classic QR for square matrices. With the 1-to-10 ratio of matrix columns to matrix rows, the compute intensity per byte transferred shifts the code towards memory-bound category of codes, and thus the independent work diminishes limiting the available parallelism.

Another issue originates in the implicit barrier resulting in normalization of each orthogonal basis column which creates synchronization overhead that can only partially be mitigated with the look-ahead technique inside and across the matrix panels. Finally, a full exploration of the data and algorithmic blocking factors would be in order to improve performance tuning close to the optimal setting. Nevertheless, the final 156× speedup figure we observed in our initial runs is impressive nonetheless and at the same time offers potential new performance research directions for possible improvements in our future work.

7.1. Errors in orthogonalization of eigenvectors

As mentioned earlier, we replaced the MGS orthogonalization algorithm with specialized rectangular QR. We explore the two algorithms in producing comparable levels of

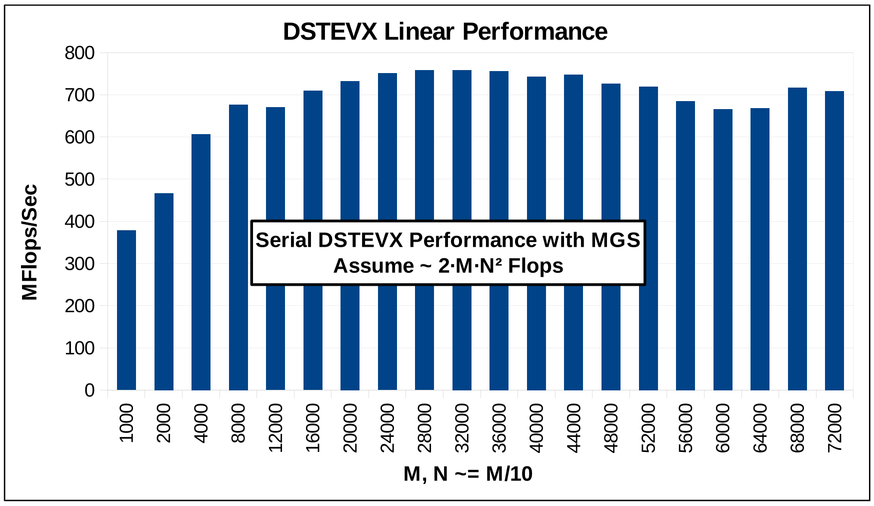

Figure 11. Serial DSTEVX () performance with MGS orthogonalization.

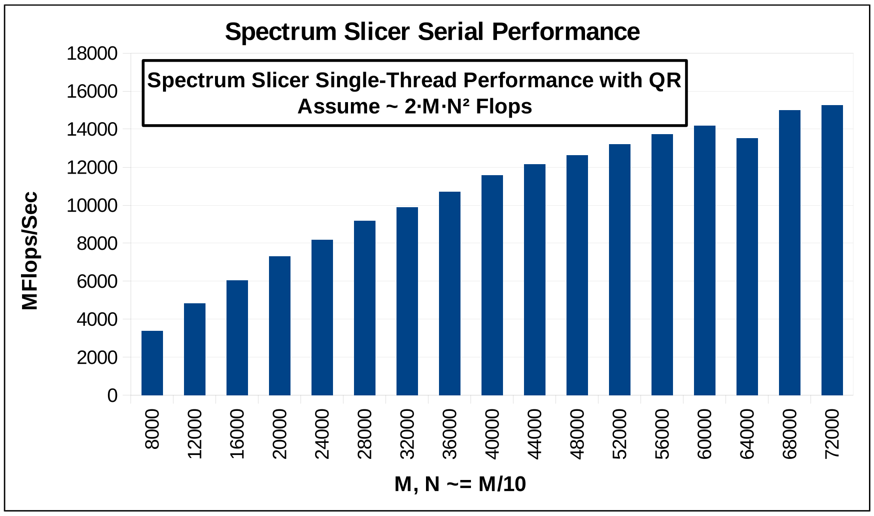

Figure 12. Serial Spectrum Slicer performance with specialized rectangular QR orthogonalization.

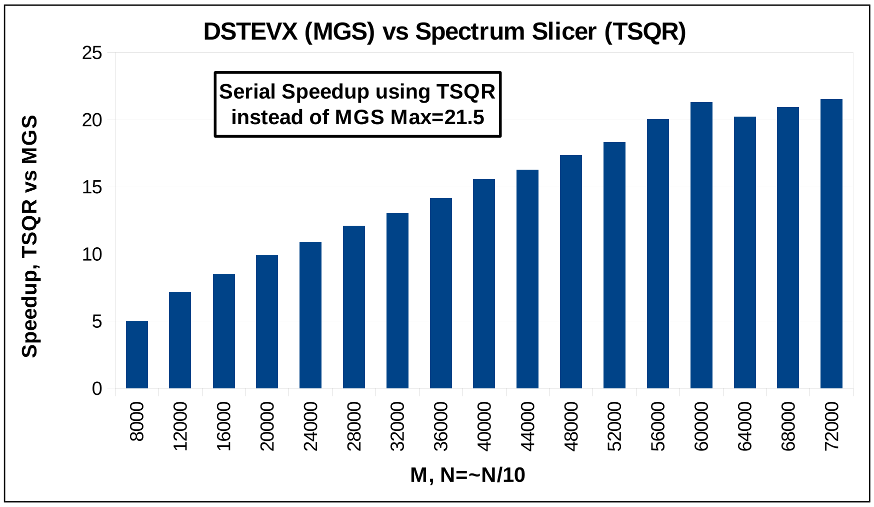

Figure 13. Comparison of speedup of serial Spectrum-Slicer with specialized rectangular QR (marked TSQR) over serial DSTEVX with MGS orthogonalization.

numerical errors. The short answer here is in affirmative: the numerical errors are comparable, and for many cases our proposed method slightly improves upon the numerical accuracy of MGS. The detailed analysis follows.

In Figure 15, we compute the orthogonality error by taking the absolute value of the dot-product of every pair of eigenvectors. This is for testing purposes only, we can do this efficiently with a matrix multiply; given our eigenvector matrix $Q_{n\times k}$ with n the rows and k the number of eigenpairs, we produce $\tilde{I}_{k\times k} = Q^T_{n\times k} \cdot Q_{n\times k}$, which should be symmetric. Q is supposed to be an orthonormal basis, thus the result should be an identity matrix. We take as the orthogonality the maximum of the absolute values of these off-diagonal matrix elements; if found on row/column i, j, then the eigenvectors in columns i and j of Q are the least orthogonal (values furthest from 0).

Figure 15 compares the orthogonality errors produced by specialized rectangular QR and implemented in the vendor's LAPACK routine `DSTEVX()`. The two began with the identical source matrix, Laguerre, with an eigenvalue distribution shown in Figure 7, and the slice taken in the far left: near zero. We chose this matrix because it is the most troublesome, producing tightly packed eigenvalues near zero.

Figure 15 uses logarithmic scale to emphasize the errors between the two algorithms. The specialized rectangular QR clearly produces more orthogonal vectors than MGS, by an average factor of 7.28×. This may be due to the blocked variant of QR, and/or other changes that reduce round-off error.

Figure 16 shows the worst absolute eigenvector errors. We compute this for an eigenpair (λ_v, v) as

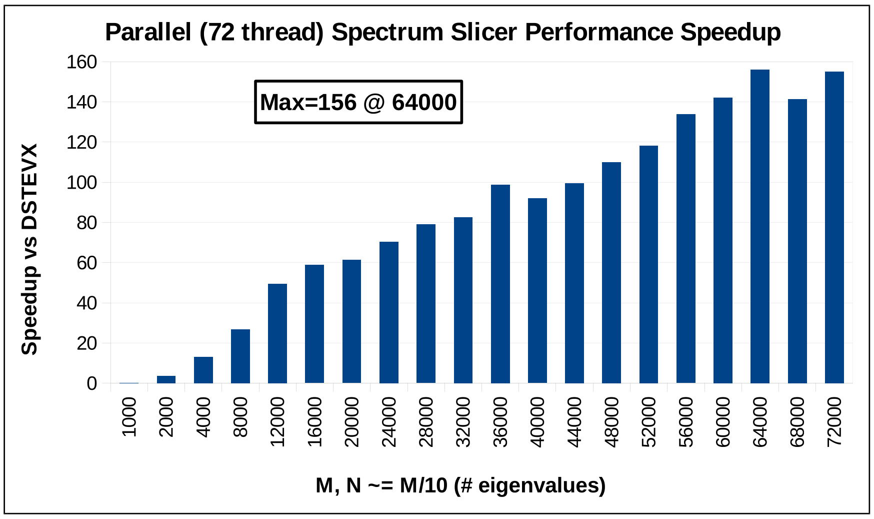

Figure 14. Speedup for QR-based parallel orthogonalization.

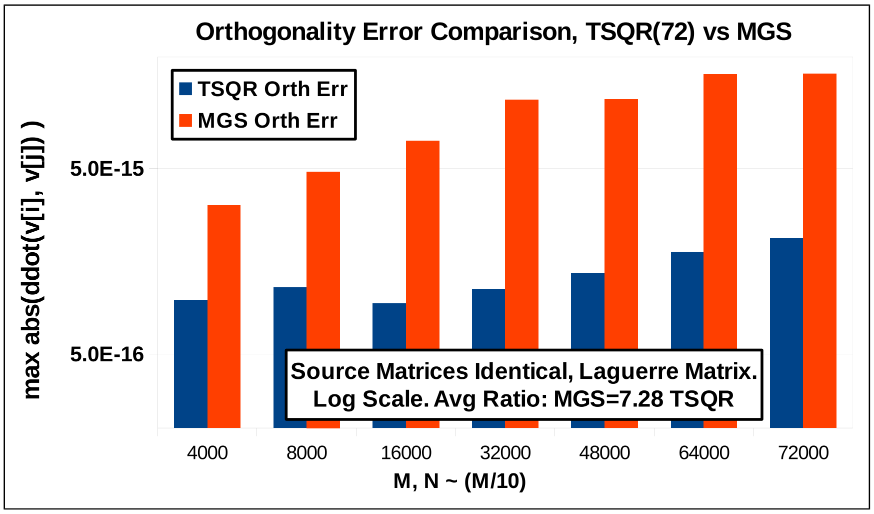

Figure 15. Comparison of numerical orthogonalization errors.

$e_r = \|(A \cdot v - \lambda_v \cdot v)\|_\infty$. The result being computed *should* be 0 in infinite precision. The absolute error is the largest magnitude value in the vector. What we report in Figure 16 is the largest magnitude error over all k eigenvectors.

As we can see in the graph, which uses logarithmic scale, neither algorithm is superior across all tested matrix sizes. The average proportion of the two errors is 0.995, favoring MGS slightly on these 7 Laguerre matrices, but this may be due to noise that would vanish with a more comprehensive testing set with a variety of application matrices, or with randomly generated matrices.

Figure 17 shows the worst *relative error*. For an eigenpair, we compute (λ_v, v) as $e_r = \|A \cdot v - \lambda_v \cdot v\|_\infty / |\lambda_v|$. We find the worst error in an eigenvector relative to its eigenvalue. This is not the same as the vector with the worst absolute error. Vectors with very large eigenvalues may produce large absolute errors that are tiny relative to their eigenvalue. In Figure 17, we report the largest relative error over all k eigenvectors. The intuition is that $e = (A \cdot v - \lambda_v \cdot v)/\lambda_v$ should be a vector of elements that are relatively small compared to λ_v. Specialized rectangular QR is overall slightly superior to MGS, showing an average of half the relative error of MGS. Again, this may be due to differences in the round-off error of the two algorithms.

Overall, our conclusion is that specialized rectangular QR is the more numerically accurate algorithm, on the particularly problematic smallest magnitude eigenvalues of the Laguerre matrix.

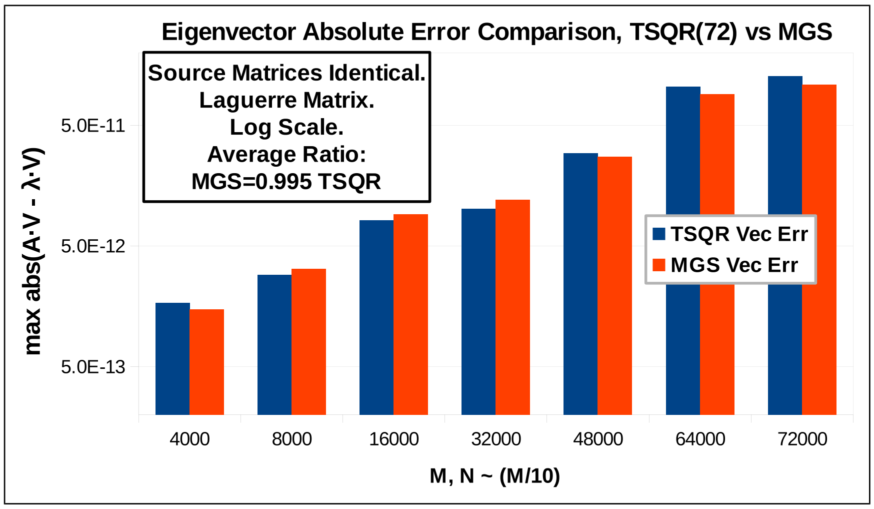

Figure 16. Comparison of numerical eigenvector errors.

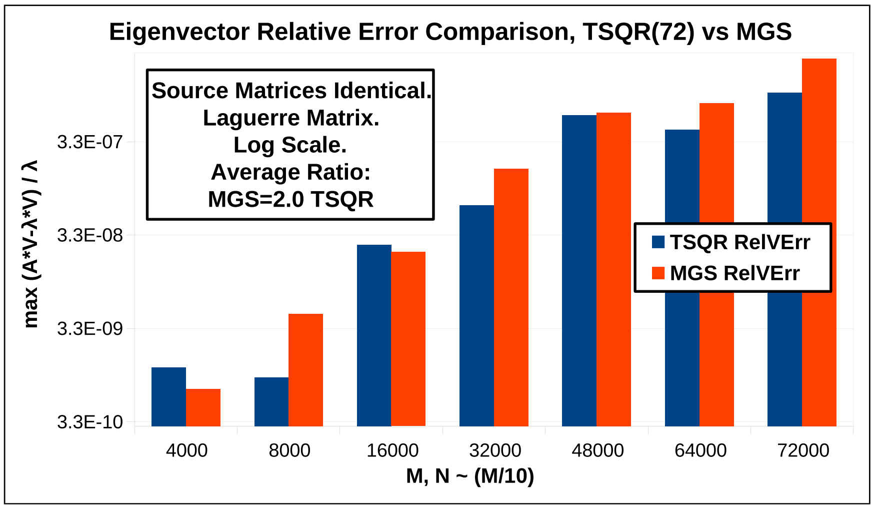

Figure 17. Comparison of relative eigenvector error.

8. Eigenvector swapping

It is possible that when eigenvalues are extremely close, the eigenvectors, after orthogonalization, may produce less error if they are swapped.

For this operation, we measure eigenvector error (given the symmetric tridiagonal matrix $A \in \mathbb{R}^{n \times n}$, eigenvector v, and eigenvalue λ) as $e = \| A \cdot v - \lambda \cdot v \|_\infty$, an $\mathcal{O}(n)$ operation. If λ and v were exact, this would produce zero.

For a swap to occur, we require a close pair of eigenvalues λ_i, λ_{i+1} s.t. $|\lambda_{i+1} - \lambda_i| < 10^{-11}$.

If that occurs, we compute the following:

1. $e_{i,i} = \|A \cdot v_i - \lambda_i \cdot v_i\|_\infty$,
2. $e_{i+1,i+1} = \|A \cdot v_{i+1} - \lambda_{i+1} \cdot v_{i+1}\|_\infty$,
3. $e_{i,i+1} = \|A \cdot v_{i+1} - \lambda_i \cdot v_{i+1}\|_\infty$ (swapped),
4. $e_{i+1,i} = \|A \cdot v_i - \lambda_{i+1} \cdot v_i\|_\infty$ (swapped).

Then, if $e_{i,i+1} < e_{i,i}$ **and** $e_{i+1,i} < e_{i+1,i+1}$, both eigenvectors work better with the opposite eigenvalue, so we swap the vectors.

In practice, the number of eigenvalues closer than 10^{-11} is infrequent; except in the Wilkinson matrix. The fact that it occurs there, and swaps occur regularly in our testing, suggests it is a beneficial check to make in practice.

Because it occurs so infrequently, we do not time it as comprehensively as the other test cases with greater relevance for practical settings. It is an $\mathcal{O}(k)$ cost to make the comparisons (all the eigenvalues are sorted); an $\mathcal{O}(n)$ cost to compute the effects of a swap, and $\mathcal{O}(n)$ to make the swap and thus it constitutes a relatively small cost of the overal computational and data transfer cost.

9. Performance of eigen-spectrum slicing

The primary function here is the *bisection* method to find eigenvalues and then the formation of at least an initial eigenvector for later refinement as necessary.

We will turn to the performance of the orthogonalization step that follows. We will begin with the Clement matrix, because we compute all $(k - 1)$ dot-products. However, there are never any orthogonalizations to perform for this case as all the eigenvalues are integers separated by 2. Thus the graph in Figure 18 consists of just the overhead of finding non-orthogonal eigenvectors when none exist in practice.The graph bars in the figure are scaled to $O(n \cdot k)$, but do not form a flat line. The difference can easily be attributed to cache coherency effects such as a transition from Level 1 to Level 2 and its associated NUMA data traffic. There is a 10% rise in per $O(n \cdot k)$ runtime from $n = 2000$ to $n = 64{,}000$.The graph bars in Figure 19 are scaled to $O(n \cdot k)$, but clearly there is another factor involved. We excluded Wilkinson matrices from this graph, and show it as a separate graph in Figure 20. The runtime growth here, as per the $O(n \cdot k)$ scaling, between $n = 2000$ and $n = 64, 000$, is about 30%. Cache effects may account for 10%, but the other 20% is something else, because approximately twice the slowdown occurs for matrices not fitting in cache. To our best estimates, the number of non-orthogonal eigenvectors is growing non-linearly with n, especially for the test matrices that suffer eigenvalue crowding that increases with n.

The graph in Figure 20 is scaled to $O(n \cdot k)$ as before to showcase the Wilkinson matrices and thus it is another relevant case of a numerical test. We see a predictable number of $k/2$ orthogonalizations performed by our method. Note that the y-axis here is about 10× the scale of the graph in Figure 19. Observe the runtime growth per $O(n \cdot k)$, from $n = 2000$ to $n = 64, 000$ is about 19%. The cache effects in our estimates account for about 10%, but the other 9% seems to be an unidentified additional factors worth further investigation.

10. Parallel scaling of eigen-spectrum slicing

In Figure 21, we show the behavior of our algorithm given a single large problem size, executed with various numbers of threads. We see the performance rising up to its peak at 20 threads, then leveling off at about 100 Tflop/s.

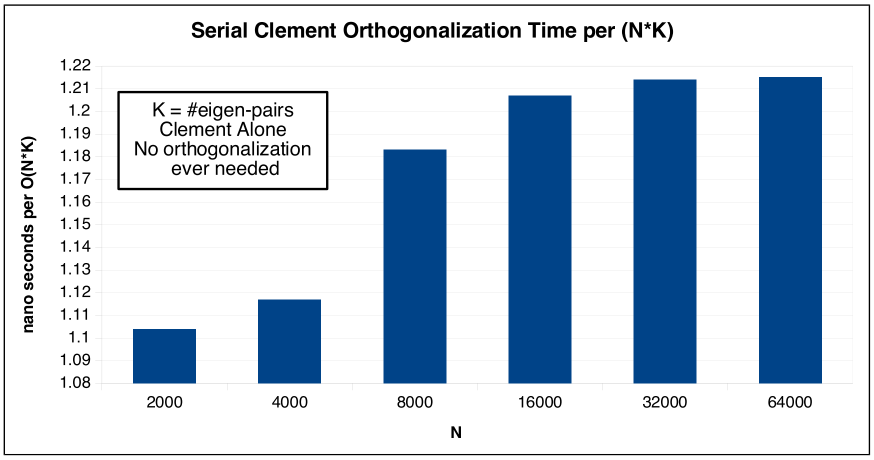

Figure 18. Single thread orthogonalization performance for the Clement matrix.

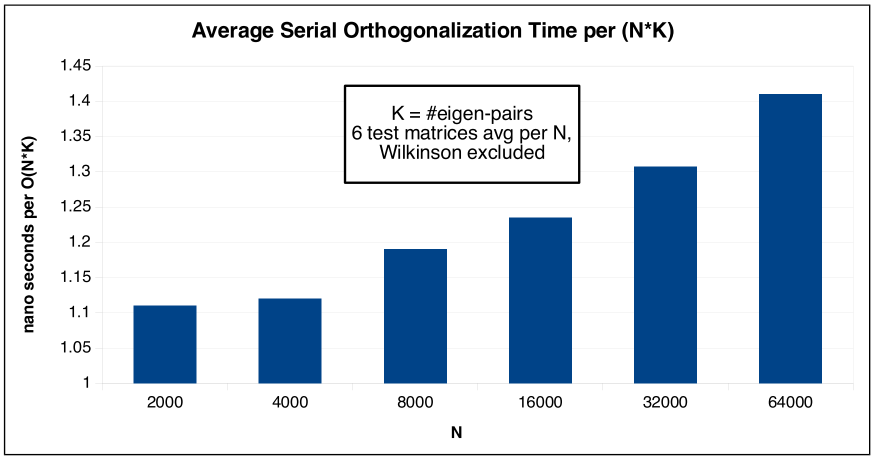

Figure 19. Single thread orthogonalization performance.

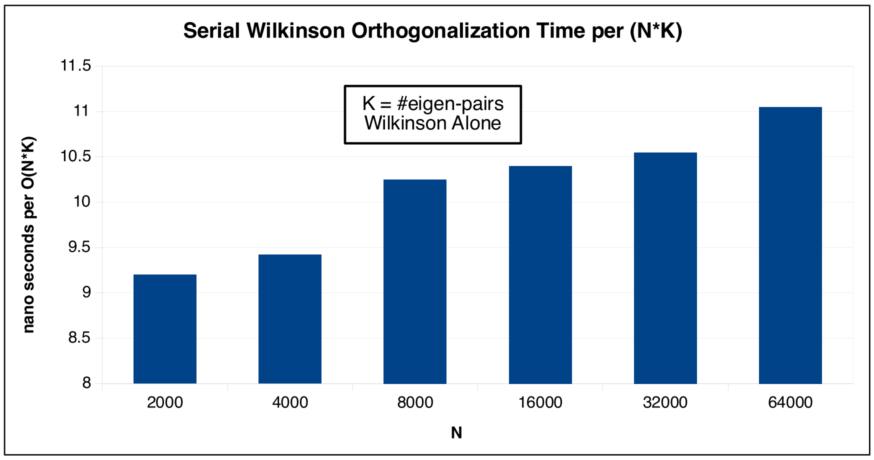

Figure 20. Wilkinson matrix results with a single thread showing orthogonalization performance.

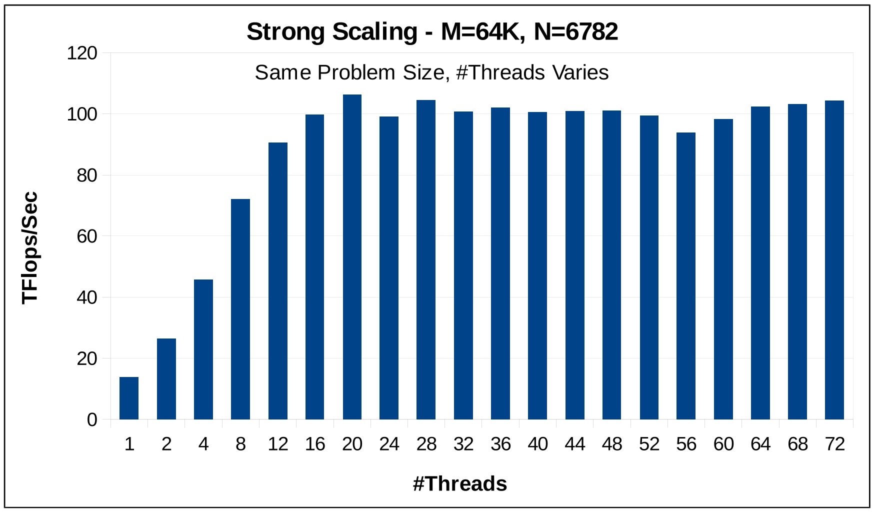

Figure 21. Strong scaling of our Spectrum Slicer performance across the available hardware threads.

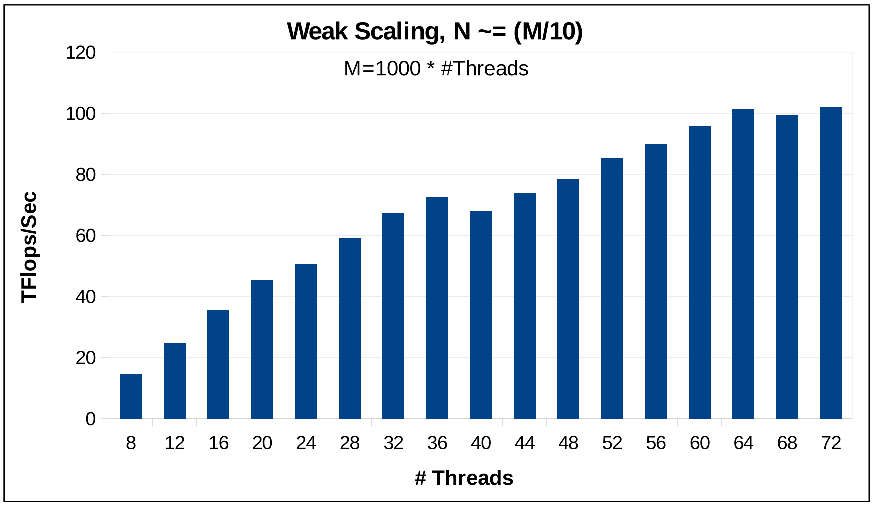

Figure 22. Weak Scaling of our Spectrum Slicer performance across the available hardware threads.

In Figure 22, we see an improvement in weak scaling regime, but again leveling off occurs at the very end, with $n \geq 64000$, at about 100 Tflop/s. Some of these limitations might be improved by better tuning of the specialized rectangular QR algorithm, which is dominating the runtime at these sizes where the leveling-off threshold is reached.

11. Performance of mixed-precision eigenvalue refinement

Our SICE-SM algorithm was originally introduced for mixed-precision eigenvalue solvers (Tsai et al., 2022). Here, we present the results when combining it with the portable offloading techniques and spectrum slicing methods that fit perfectly in one of the phases of the full and robust eigenvalue solver and eigenvector computation. We break down this contribution into refinement merged with reorthogonalization, which are separate from the actual speedups across a range of CPUs and GPUs, that we discuss last.

The eigenvalue refinement algorithm was originally proposed by Ogita and Aishima (2018) and we present in Table 1 the results from this type of refinement based on our portable implementation outlined earlier. We also note that in addition to the techniques based on special-purpose QR factorization of tall-and-narrow matrices, which we presented earlier, we also used Newton-Schulz iteration (Chen and Chow, 2014) for re-orthogonalization of the eigenvectors during the refinement. This was performed according to the formula

$$X_{i+1} \leftarrow X_i + \frac{1}{2} X_i (I - X_i^* X) \tag{4}$$

The main advantage of using portable approaches to implement complex numerical schemes is the ability to test

Table 1. Timing and performance of refining eigenvalues and eigenvectors simultaneously with respect to the number of refined eigen-pairs on NVIDIA Volta V100.

Matrix	Eigenvector	Time	Performance
size	count	(ms)	(Tflop/s)
20000	1	3.76	0.212
20000	8	3.79	1.688
20000	32	6.48	3.949
20000	128	13.57	7.544

their performance across the variety of hardware targets. This is done in Figure 23 that shows advantages and disadvantages of using a mixed-precision eigensolver, which depend on the specific hardware structure and the balance between performance levels of the floating-point units for different data formats. On common x86 and other RISC CPUs, the 32-bit format units are twice as fast as their 64-bit counterparts. Thus, we observe a slight slowdown when employing the mixed-precision approach. On NVIDIA Volta V100 GPUs, the performance difference is the same as the CPUs, however, our approach derived from the SICE-SM algorithm (Tsai et al., 2022), takes advantage not only of the GPU, but also includes the CPU in the computation thus showing speedup for real domain problems. That advantage increases for the complex domain to about 50% speedup over the uniform precision implementation. Finally, NVIDIA GTX1060 card is an example of a gaming GPU card without physical FP64 units that are instead emulated by the FP32 hardware at the warp level and thus the imbalance of the two data formats is 1 to 32, a staggering difference from the previous two targets. The figure clearly indicates how this can translate into much higher speedups of 2× and 3.6× for real and complex domains, respectively. We expect the

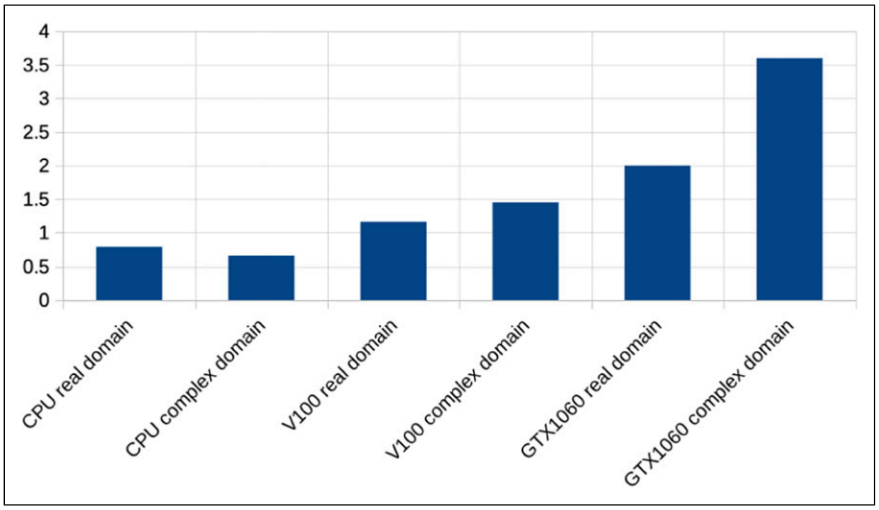

Figure 23. Speedup of our portable mixed-precision eigenvalue solver across CPUs and GPUs that feature different balance between performance of different floating-precition formats.

gains to be even larger for newer hardware accelerators that feature much higher imbalance of different floating-point formats albeit at the cost of much lower bit counts in the fast formats relegated to the "tensor core" units.

Speedup of mixed-precision eigensolver over its uniform precision counterpart

12. Conclusions and future work

We presented a number of contributions to the parallel and portable PLASMA library (Dongarra et al., 2019) with a combined focus on both hardware efficiency and portable performance while maintaining asymptotic scaling up to and beyond the sizes of modern platforms and input problem dimensions. The numerical contributions include parallel spectrum slicing and mixed-precision iterative refinement results for the SICE-SM algorithm (Tsai et al., 2022). The former is akin to LAPACK's `DSTEVX()` routine, to extract the eigenvalues of a matrix in a given range and return an orthonormal matrix of the corresponding eigenvectors. We demonstrated the speedup of 156× using 72 threads and OpenMP tasks which compares favorably against the available sequential implementations that do not fully exploit the algorithmic space to enable high levels of parallelism. Our analysis showed the improvement may have been attributed to the replacement of the original MGS orthogonalization with the specialized rectangular QR algorithm. Our proposed replacement achieves superior performance thanks to primarily a data blocking strategy and the use of Level 3 BLAS computational kernels as well as exposing far greater levels of parallelism to the hardware's compute units.

Potential extensions left for future work include investigation, better understanding, and subsequent tuning of the cause of the observed leveling off we reported for the performance metric in our strong and weak scaling results. Using and extending our GPU portability strategies to further the functionality available on the hardware accelerator is currently under development to deliver even greater efficiency and accuracy in a combined processor-accelerator systems. Finally, moving beyond a single accelerator or even beyond a single node is also under consideration to either address even larger problem sizes or to accommodate existing application contexts in which the matrix data is distributed in some fashion.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative. This research uses resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. This research also uses resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. This work was partially funded by an NSF Collaborative Research Framework: Basic ALgebra LIbraries for Sustainable Technology with Interdisciplinary Collaboration

(BALLISTIC), a project of the University of Tennessee 2004541. Research was sponsored by the Department of the Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Department of the Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

ORCID iD

Piotr Luszczek https://orcid.org/0000-0002-0089-6965

Notes

1. We used the latest (as of this writing) LAPACK version 3.12.0 released on November 24, 2023.
2. For a symmetric matrix with eigenvalue duplicities of 1 the eigenvalues should be distinct; but in certain problematic matrices like the Wilkinson matrices, a cluster of arithmetically distinct eigenvalues may be so close they are all within machine-precision of each other; thus the software using finite floating-point precision arithmetic cannot distinguish or represent them and thus produces a single eigenpair with a multiplicity factor larger then 1.
3. With minor differences due to algorithmic differences such as data/algorithmic blocking and changes in the operations' order producing round-off error similar in magnitude but not its value.

References

Agullo E, Augonnet C, Dongarra J, et al. (2012) Faster, cheaper, better – a hybridization methodology to develop linear algebra software for GPUs. *GPU Computing Gems Jade*. Edition. Elsevier Inc, 473–484. DOI: 10.1016/B978-0-12-385963-1.00034-4.

Anderson E, Bai Z, Bischof C, et al. (1999) *LAPACK User's Guide*. 3rd edition. Philadelphia: Society for Industrial and Applied Mathematics.

Ando T (1963) Properties of Fermion density matrices. *Reviews of Modern Physics* 35(3): 690–702.

Architecture Review Board (2021) Openmp application programming interface. Version 5.2, November.

Barghathi H, Casiano-Diaz E and Del Maestro A (2017) Particle partition entanglement of one dimensional spinless fermions. *Journal of Statistical Mechanics* 2017: 083108. DOI: 10.1088/1742-5468/aa819a.

Bosilca G, Bouteiller A, Danalis A, et al. (2011) Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In: 12th IEEE international workshop on parallel and distributed scientific and engineering computing (PDSEC'11), Anchorage, Alaska, 20 May 2011.

Buttari A, Langou J, Kurzak J, et al. (2008) Parallel tiled QR factorization for multicore architectures. *Concurrency and Computation: Practice and Experience* 20(13): 1573–1590. DOI: 10.1002/cpe.1301.

Carlson BC and Keller JM (1961) Eigenvalues of density matrices. *Physical Review* 121(3): 659–661.

Chen J and Chow E (2014) *A Newton-Schulz Variant for Improving the Initial Convergence in Matrix Sign Computation*. Technical Report ANL/MCS-P5059-0114, Mathematics and Computer Science Division. Argonne, IL, USA: Argonne National Laboratory.

Daniel JW, Gragg WB, Kaufman L, et al. (1976) Reorthogonalization and stable algorithms for updating the Gram-Schmidt QR factorization. *Mathematics of Computation* 30(136): 772–795.

Demmel JW, Dhillon I and Ren H (1995) On the correctness of some bisection-like parallel eigenvalue algorithms in floating point arithmetic. *Electronic Transactions on Numerical Analysis* 3: 116–149.

Demmel JW, Marques OA, Parlett B, et al. (2006) *A Testing Infrastructure for LAPACK's Symmetric Eigensolvers*. Technical Report 182. LAPACK Working Notes.

Demmel J, Grigori L, Hoemmen M, et al. (2012) Communication-optimal parallel and sequential QR and LU factorizations. *SIAM Journal on Scientific Computing* 34(1): A206–A239.

Demmel J, Grigori L, Gu M, et al. (2015) Communication avoiding rank revealing QR factorization with column pivoting. *SIAM Journal on Matrix Analysis and its Applications* 36(1): 55–89.

Dongarra J, Gates M, Haidar A, et al. (2019) PLASMA: parallel linear algebra software for multicore using OpenMP. *ACM Transactions on Mathematical Software* 45(2): 16:1–16:35. DOI: 10.1145/3264491.

Fasi M, Higham NJ, Mikaitis M, et al. (2021) Numerical behavior of nvidia tensor cores. *PeerJ Computer Science* 7: e330. DOI: 10.7717/peerj-cs.330.

Gates M, Kurzak J, Charara A, et al. (2019) SLATE: design of a modern distributed and accelerated linear algebra library. In: Proceedings of the international conference for high performance computing, networking, storage and analysis, Denver, CO, 12–17 November 2023, pp. 1–18.

Giraud L, Langou J and Rozloznik M (2005a) The loss of orthogonality in the Gram-Schmidt orthogonalization process. *Computers & Mathematics with Applications* 50(7): 1069–1075. DOI: 10.1016/j.camwa.2005.08.009.

Giraud L, Langou J, Rozložník M, et al. (2005b) Rounding error analysis of the classical Gram-Schmidt orthogonalization process. *Numerische Mathematik* 101(1): 87–100. DOI: 10.1007/s00211-005-0615-4.

Gu M and Eisenstat SC (1995) A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem. *SIAM Journal on Matrix Analysis and Applications* 16(1): 172–191.

Gu M, Demmel J and Dhillon I (1994) *Efficient Computation of the Singular Value Decomposition with Applications to Least*

Squares Problems. Technical Report CS-94-257. Knoxville, TN: University of Tennessee.

Haidar A, Ltaief H and Dongarra J (2011) Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels. In: Proceedings of 2011 international conference for high performance computing, networking, storage and analysis, SC '11, Seattle, Washington, 12–18 November 2011. New York, NY: ACM, pp. 8:1–8:11.

Haidar A, Ltaief H, Luszczek P, et al. (2012) A comprehensive study of task coalescing for selecting parallelism granularity in a two-stage bidiagonal reduction. In: Proceedings of the IEEE international parallel and distributed processing symposium, Shanghai, China, 21–25 May 2012.

Haidar A, Kurzak J and Luszczek P (2013) An improved parallel singular value algorithm and its implementation for multicore hardware. In: SC13, the international conference for high performance computing, networking, storage and analysis. Denver, Colorado, 17–21 November 2013.

Haidar A, Luszczek P and Dongarra J (2014) New algorithm for computing eigenvectors of the symmetric eigenvalue problem. In: The 15th IEEE international workshop on parallel and distributed scientific and engineering computing (PDSEC 2014). Phoenix, AZ, 23 May 2014.

ISO/IEC14882 (2023) Information technology – programming languages – C++.

ISO/IEC1539-1 (2018) Information technology – programming languages – Fortran.

ISO/IEC9899 (2023) Information technology – programming languages – C.

Kiełbasiński A (1974) Analiza numeryczna algorytmu ortogonalizacji Grama-Schmidta. In: *Roczniki Polskiego Towarzystwa Matematycznego, Seria III: Mathematyka Stosowana II*, volume 2, number 2. Polskie Towarzystwo Matematyczne, pp. 15–35. DOI: 10.14708/ma.v2i2.1048.

Löwdin PO (1955) Quantum theory of many-particle systems I. Physical interpretations by means of density matrices, natural spin-orbitals, and convergence problems in the method of configurational interaction. *Physical Review* 97(6): 1474–1489.

Ltaief H, Luszczek P, Haidar A, et al. (2012) Enhancing parallelism of tile bidiagonal transformation on multicore architectures using tree reduction. In: Proceedings of 9th international conference, PPAM 2011 (eds Wyrzykowski R, Dongarra J, Karczewski K, et al.), Toruń, Poland, 11–14 September 2011, pp. 661–670.

Luszczek P, Ltaief H and Dongarra J (2011) Two-stage tridiagonal reduction for dense symmetric matrices using tile algorithms on multicore architectures. In: IPDPS 2011: IEEE international parallel and distributed processing symposium, Anchorage, Alaska, 16–20 May 2011.

Marques OA, Vömel C, Demmel JW, et al. (2008) Algorithm 880: a testing infrastructure for symmetric tridiagonal eigensolvers. *ACM Transactions on Mathematical Software* 35(1): 1–13.

Ogita T and Aishima K (2018) Iterative refinement for symmetric eigenvalue decomposition. *Japan Journal of Industrial and Applied Mathematics* 35(3): 1007–1035.

Paige CC and Strakos Z (2001) Residual and backward error bounds in minimum residual Krylov subspace methods. *SIAM Journal on Scientific Computing* 23(6): 1898–1923. DOI: 10.1137/S1064827500381239.

Sid-Lakhdar W, Cayrols S, Bielich D, et al. (2023) PAQR: pivoting avoiding QR factorization. In: Proceedings of 36th IEEE international parallel & distributed processing symposium (IPDPS), Lyon, France, 30 May–3 June 2022. Best paper nominee.

Terao T, Ozaki K and Ogita T (2020) LU-Cholesky QR algorithms for thin Q R decomposition. *Parallel Computing* 92: 102571. DOI: 10.1016/j.parco.2019.102571.

Tsai Y, Luszczek P and Dongarra J (2022) Mixed-precision algorithm for finding selected eigenvalues and eigenvectors of symmetric and Hermitian matrices. In: ScalAH22: 13th workshop on latest advances in scalable algorithms for large-scale heterogeneous systems, Dallas, Texas, 13 November 2022, pp. 1–10.

Volkov V and Demmel J (2007) *Using GPUs to Accelerate the Bisection Algorithm for Finding Eigenvalues of Symmetric Tridiagonal Matrices*. Technical Report UCB/EECS-2007-179, EECS Department. Berkeley: University of California. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-179.html.

Zhang J (2003) The scaled Sturm sequence computation. In: Proceedings of the Eighth SIAM Conference on Applied Linear Algebra (LA03). *Society of Industrial and Applied Mathematics*.

Piotr Luszczek received the BS and MSc degrees in computer science from the AGH University of Science and Technology in Kraków, Poland, and the Ph.D. degree in computer science from the University of Tennessee Knoxville. He is currently a Staff Scientist at MIT Lincoln Laboratory and a Research Assistant Professor with Innovative Computing Laboratory, University of Tennessee, Knoxville's Tickle College of Engineering. His research interests include benchmarking, numerical linear algebra for high-performance computing, automated performance tuning for modern hardware, and stochastic models for performance. He also holds the position of the Editors-in-Chief of ACM Transactions on Mathematical Software (TOMS).

Anthony Castaldo an associates degree in computer programming and has about twenty-five years of boot-level assembly for embedded devices. He later received Bachelor's and Master's in Mathematics and later MSc and PhD in computer science to become a full-time research assistant professor at the University of Texas in San Antonio. His prior industrial career spanned banking, finance, health systems, and HPC consultancy services.

Yaohung Tsai earned both my Bachelor's and Master's degrees in mathematics from National Taiwan University. Later, he earned his PhD degree in Computer Science from the University of Tennessee Knoxville. Afterwards, he joined the advanced hardware architecture group at Facebook (now Meta).

Daniel Mishlerreceived the Bachelor's of Science degree at the University of Indiana at Bloomington. He currently pursues a PhD degree at the University of Tennessee Knoxville. He conducts research on distributed memory HPC systems and alternative messaging systems both of his home institution and during internships at US national laboratories.

Jack Dongarra holds an appointment at the University of Tennessee, Oak Ridge National Laboratory, and the University of Manchester. He specializes in numerical algorithms in linear algebra, parallel computing, advanced computer architectures, programming methodology, and tools for parallel computers. He was chosen for the IEEE Sid Fernbach Award in 2004; in 2008, he was the recipient of the first IEEE Medal of Excellence in Scalable Computing; in 2010, he was the first recipient of the SIAM Special Interest Group on Supercomputing's award for Career Achievement; in 2011 he was the recipient of the IEEE Charles Babbage Award; in 2013 he received the ACM/IEEE Ken Kennedy Award; in 2019 he received the ACM/SIAM Computational Science and Engineering Prize, in 2020 he received the IEEE-CS Computer Pioneer Award, and in 2022 he received the ACM A.M. Turing Award for pioneering contributions to numerical algorithms and software that have driven decades of extraordinary progress in computing performance and applications. He is a Fellow of the AAAS, ACM, IEEE, and SIAM and a foreign member of the British Royal Society and a US National Academy of Engineering member.