GPU TECHNOLOGY CONFERENCE

Josh Romero, Stanford University/NVIDIA

Developing an Improved Generalized Eigensolver with Limited CPU Offloading

Outline

- Motivation
- Problem Description
- Discussion of Solver Routines
  - Implementation
  - Performance Results (K80, P100, Xeon and Xeon Phi with MKL)
- Overall Performance and Conclusions

Motivation

- Generalized eigenvalue problems arise in a number of scientific computing applications
  - Example: quantum chemistry simulations with plane-wave solvers
  - Solving this type of problem is a major component of existing computational methods

- Several existing libraries which provide this capability
  - cuSOLVER
    - Developed by NVIDIA for GPUs
    - Contains implementation of select LAPACK routines for solving sparse and dense linear systems
  - MAGMA
    - Developed by the Innovative Computing Laboratory (ICL) at the University of Tennessee
    - Emphasis on heterogeneous ***"multicore + GPU"*** operation

Motivation

- MAGMA implements hybrid CPU/GPU subblocking strategy:
  - Key computations on subblock of matrix offloaded to CPU
  - Updates to bulk completed on GPU during subblock computation
  - Strategy mirrors that used in CPU LAPACK implementation

- MAGMA performs well but heterogeneous operation has drawbacks:
  - Operating on a system where the ratio of available CPU to GPU resources is small can introduce performance bottlenecks
  - Requires active communication of subblock data between CPU and GPU
    - Can generate lots of PCIe traffic which can impact scalability
  - Requires access to high-performing CPU libraries

Motivation

- For these reasons, a generalized eigensolver with reduced CPU offloading is desirable.

- General strategy:
  - Use subblocking algorithms in LAPACK and MAGMA as baseline
  - Replace offloaded CPU subblock computations with on device GPU computations
  - Preserve concurrency using CUDA streams

Performance Testing Details

- The solver has been developed for implementation in the open-source QuantumESPRESSO (QE) code
  - We focus only on required configuration path
  - Tested routines for system sizes ranging from N = 512 to N = 8192
  - Source written in CUDA Fortran
- We report results from the following systems
  - System 1:
    - Intel Xeon E5-2698 V3 (16 core Haswell, 2.30 GHz)
      - NVIDIA K80 GPU (875 MHz boosted)
      - NVIDIA P100-PCIe GPU (1328 MHz boosted)
  - System 2:
    - Intel Xeon Phi 7250 (KNL, 68 core, 1.4 GHz)
    - N <= 2048 (34 threads) and N >2048 (68 threads), MCDRAM only

Problem Description

Begin with a generalized Hermitian-definite Eigenproblem: $\mathbf{A}x = \lambda\mathbf{B}x$

1. Reduce to standard Eigenproblem: $\mathbf{C}x = \lambda x$
   a. Compute Cholesky factorization: $\mathbf{B} = \mathbf{U}^H\mathbf{U}$ (ZPOTRF)
   b. Form matrix C: $\mathbf{C} = \mathbf{U}^{-H}\mathbf{A}\mathbf{U}^{-1}$ (ZHEGST)

2. Compute eigenvalues/vectors of standard problem using QR: (ZHEEVD)
   a. Tridiagonalize C: $\tilde{\mathbf{C}} = \mathbf{Q}^H\mathbf{C}\mathbf{Q}$ (ZHETRD)
   b. Compute eigenvalues/vectors of tridiagonal system ← Remains on CPU (ZSTEDC)
   c. Apply householder reflectors to eigenvectors (ZUNMTR)

3. Triangular solve to obtain eigenvectors for original problem (ZTRSM)

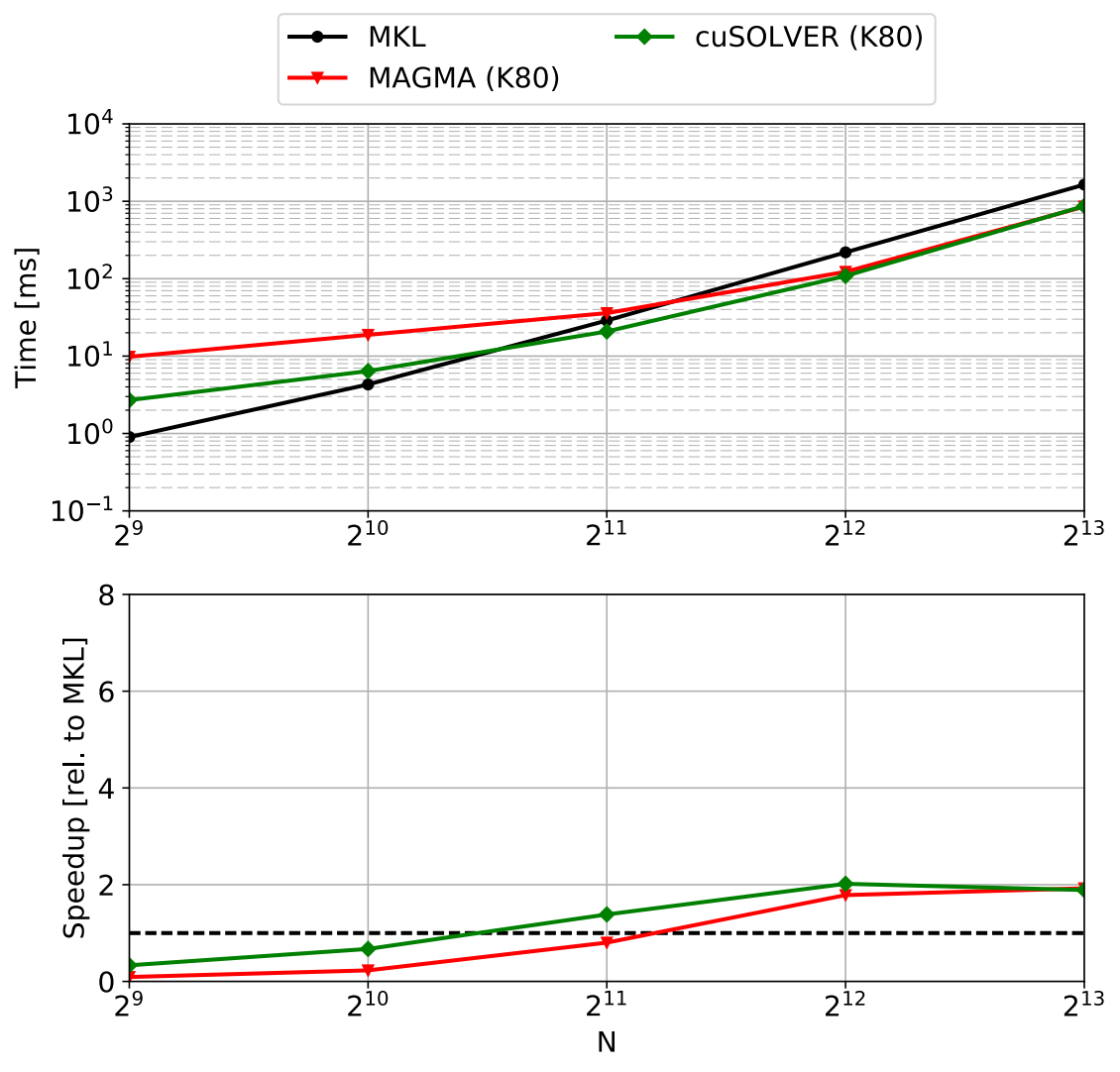
Step 1a: Cholesky factorization

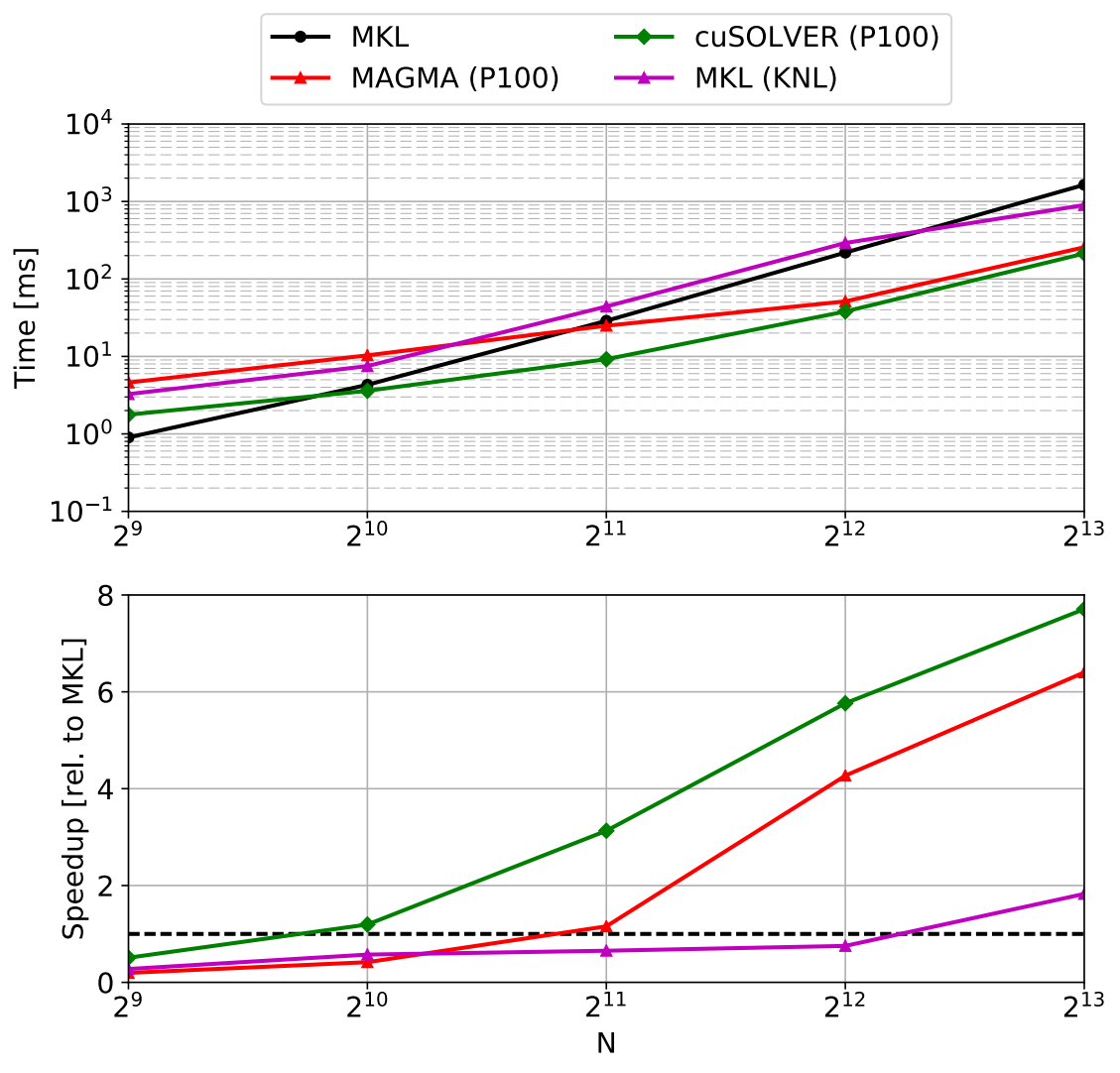
- Use the `cusolverDnZpotrf` function available within cuSOLVER. Requires definition of types/interfaces to be called from CUDA Fortran:

```
 type cusolverDnHandle
    type(c_ptr) :: handle
  end type
cusolverDnHandle
```

```
interface
  integer(c_int) function cusolverDnZpotrf( &
    handle, uplo, n, A, lda, Workspace, Lwork, devInfo) &
    bind(C,name='cusolverDnZpotrf')
      use iso_c_binding
      import cusolverDnHandle
      type(cusolverDnHandle), value :: handle
      integer(c_int), value :: uplo, n, lda
      complex(8), device :: A(*)
      !pgi$ ignore_tkr (r) Workspace
      complex(8), device :: Workspace(*)
      integer(c_int), value :: Lwork
      integer(c_int), device :: devInfo
    end function cusolverDnZpotrf
 end interface
```

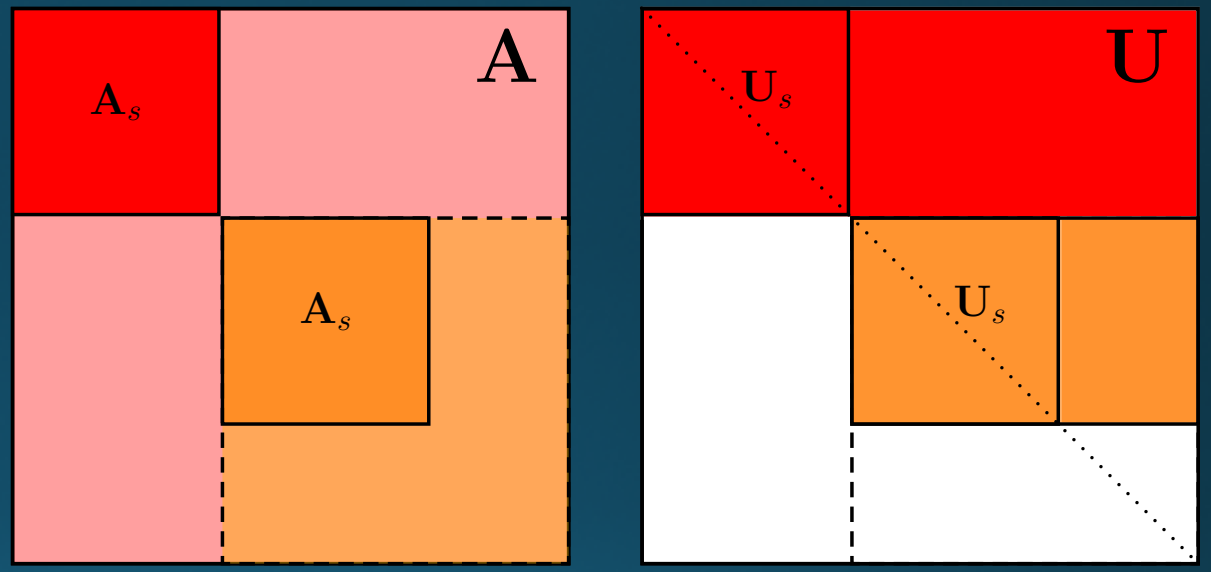
Step 1a: ZPOTRF





Step 1b: Form Matrix C

- Corresponds to `ZHEGST` routine in standard LAPACK library
  - Matrix updates: `ZTRSM`, `ZHEMM`, `ZHER2K`
    - BLAS Level 3
    - Use CUBLAS functions
  - Subblock computation: `ZHEGS2`
    - BLAS Level 1 and 2



Step 1b: Form Matrix C

- `ZHEGS2` routine is responsible for reducing the subblock:

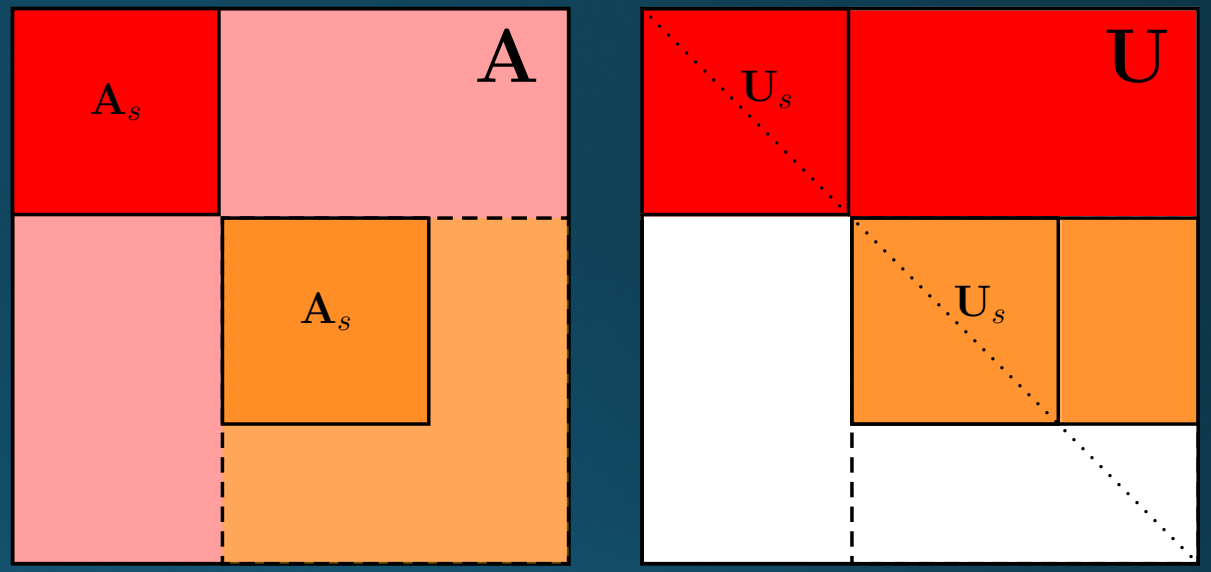
$$\mathbf{C}_s = \mathbf{U}_s^{-H} \mathbf{A}_s \mathbf{U}_s^{-1}$$

- Can be completed using two `ZTRSM` calls on the subblock:

$$\mathbf{U}_s^H \mathbf{X}_1 = \mathbf{A}_s \rightarrow \mathbf{X}_1 = \mathbf{U}_s^{-H} \mathbf{A}_s$$

$$\mathbf{X}_2 \mathbf{U}_s = \mathbf{X}_1 \rightarrow \mathbf{X}_2 = \mathbf{X}_1 \mathbf{U}_s^{-1} = \mathbf{U}_s^{-H} \mathbf{A}_s \mathbf{U}_s^{-1}$$

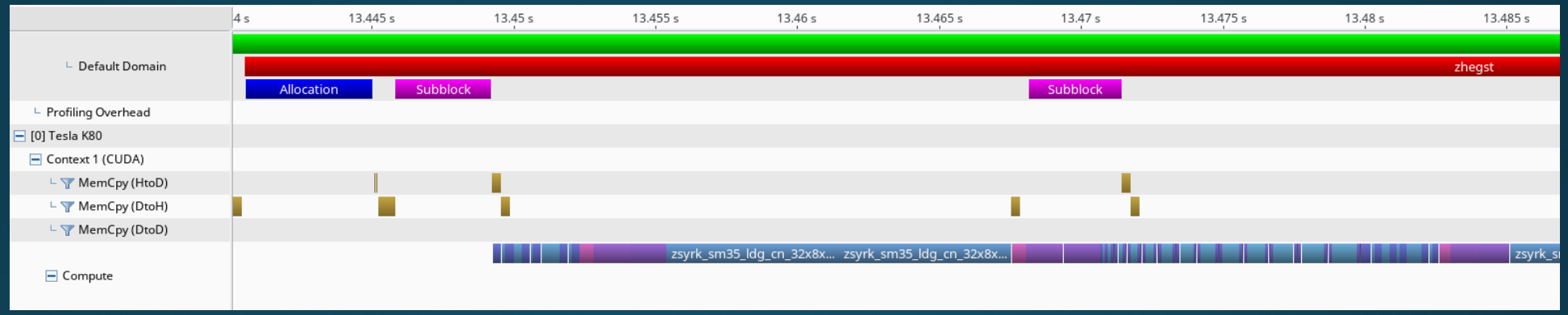
- Use `ZTRSM` from CUBLAS to complete subblock computation!



Step 1b: Form Matrix C

- To preserve concurrency, subblock computation placed in separate stream

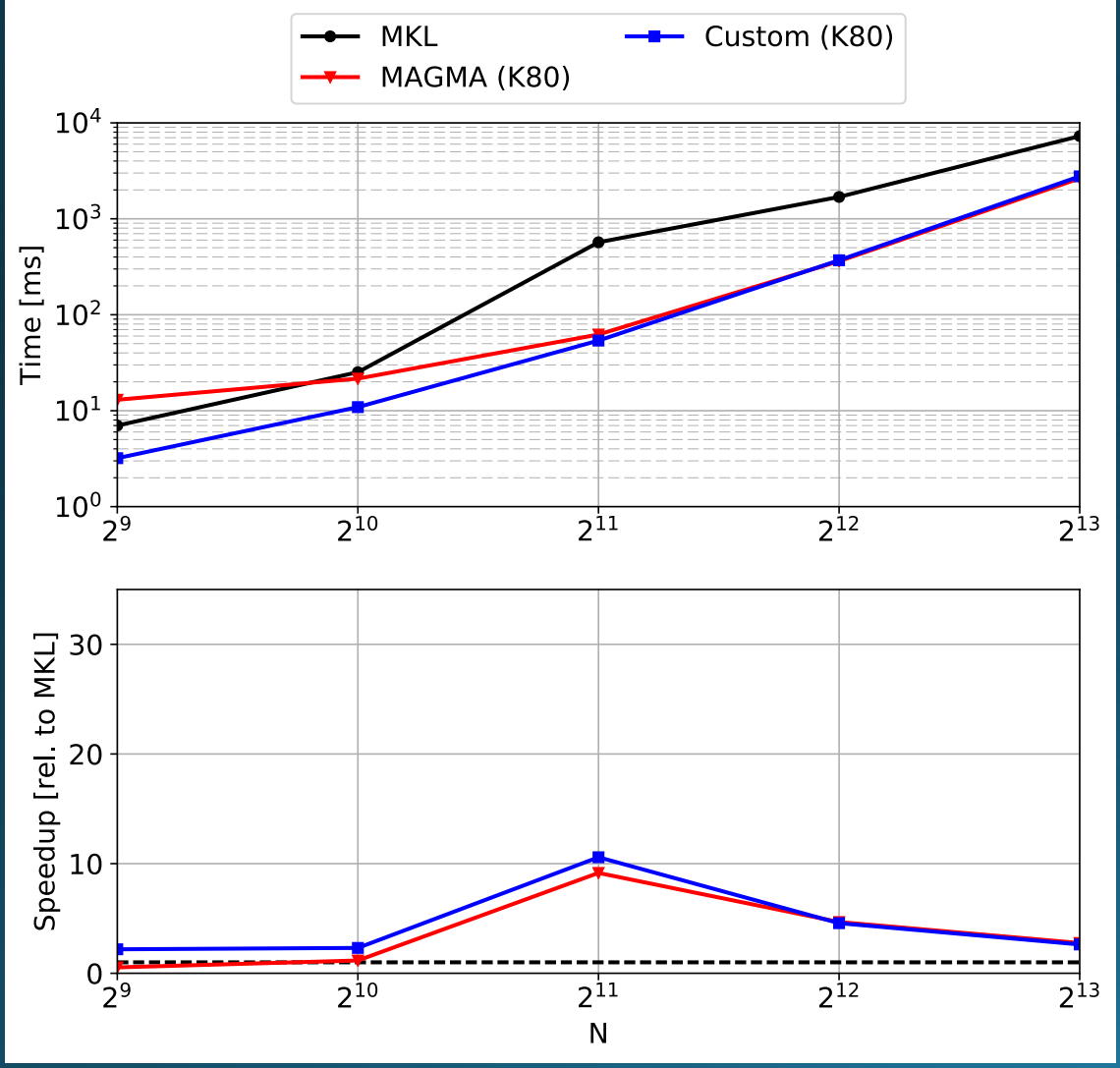
MAGMA

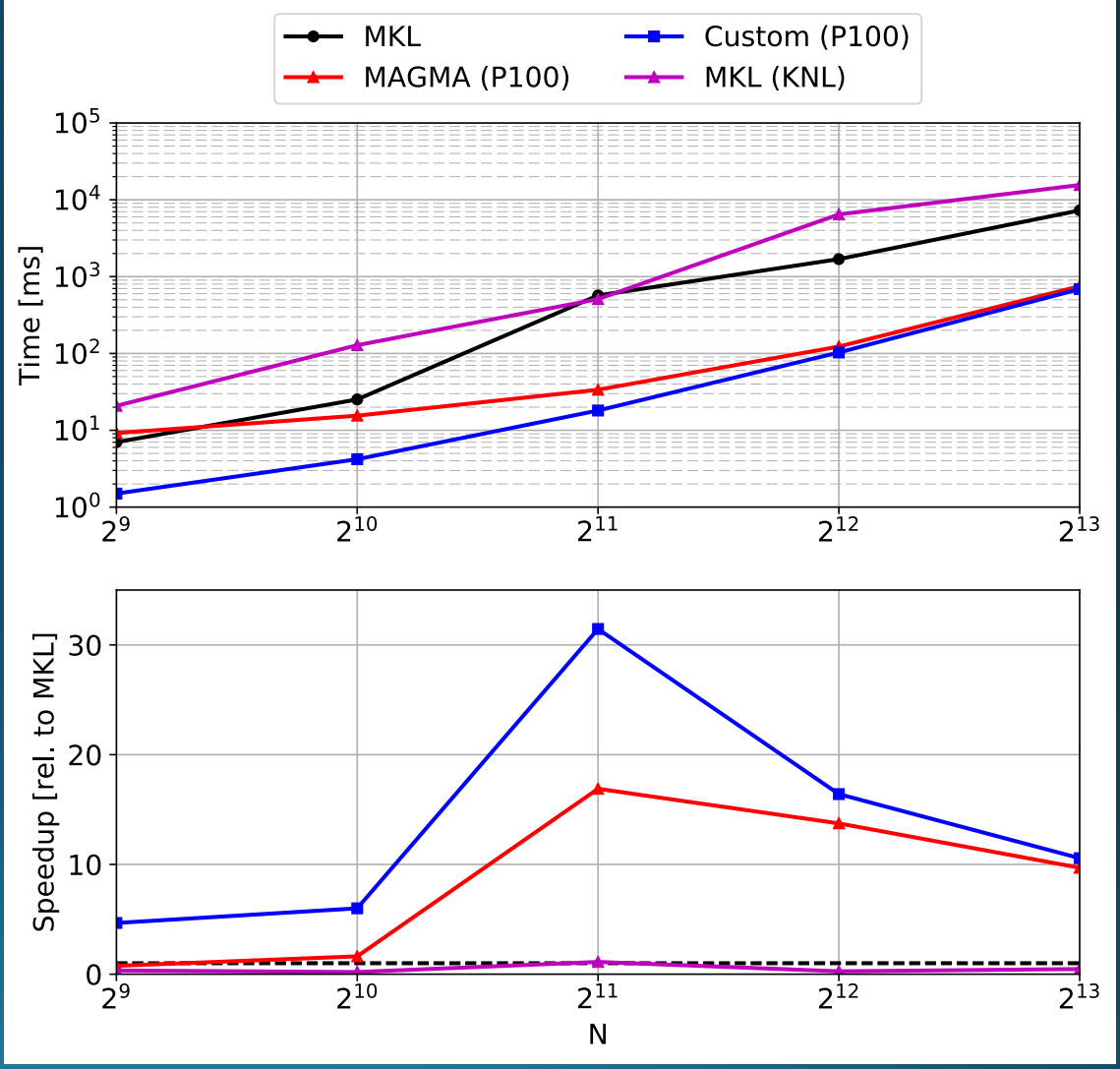


Custom

(Timelines not to scale)

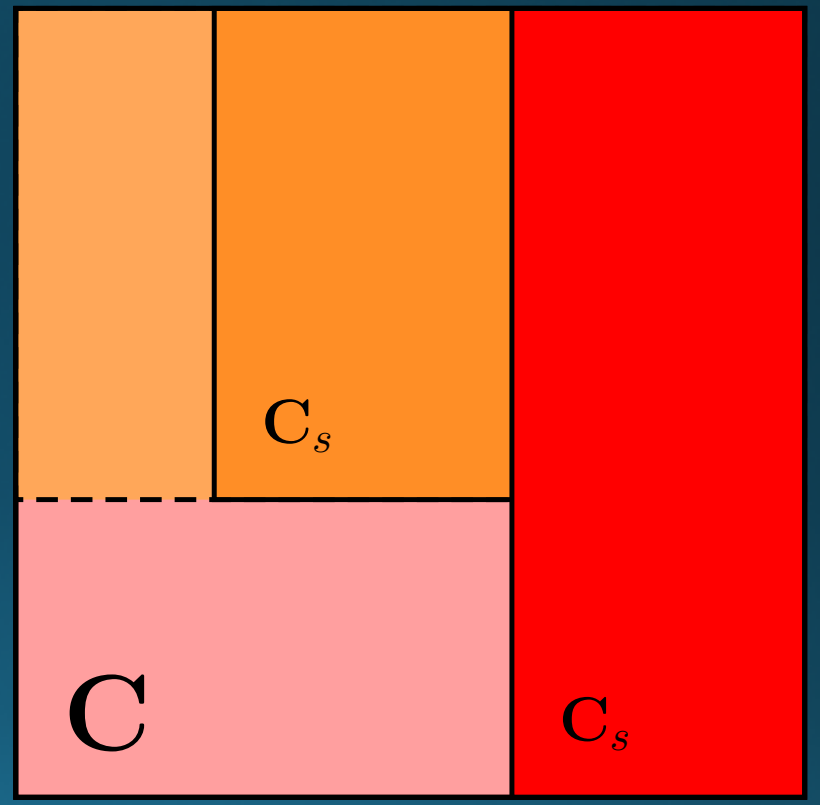
Step 1b: ZHEGST





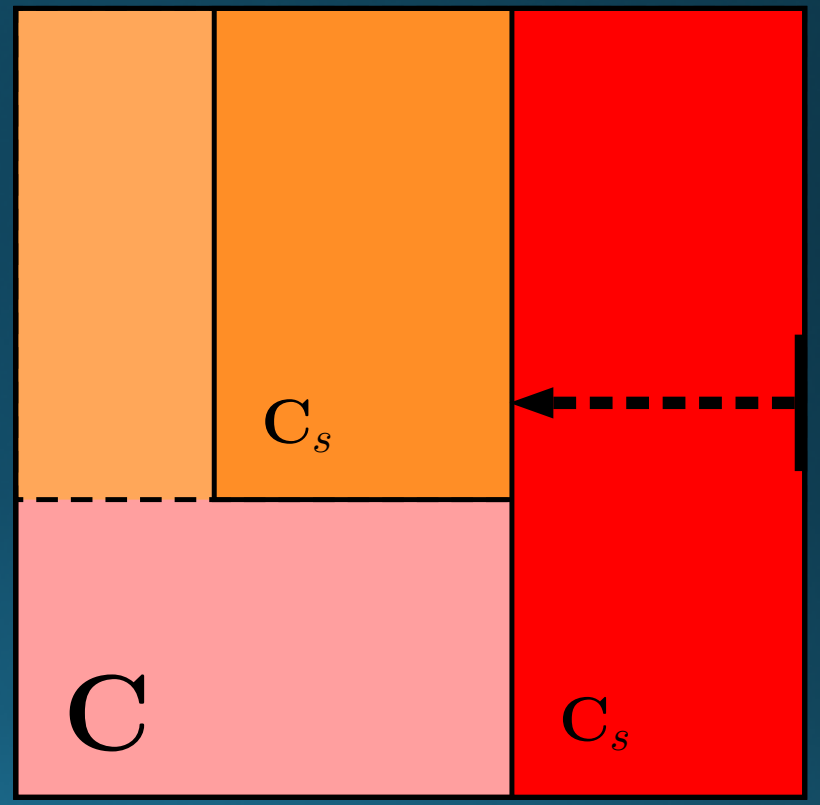
Step 2a: Tridiagonalize C

- Corresponds to `ZHETRD` routine in standard LAPACK library
  - Matrix updates: `ZHER2K`
    - BLAS Level 3
    - Use CUBLAS function
  - Subblock computation: `ZLATRD, ZHETD2`
    - BLAS Level 1 and 2
    - Fairly difficult function for GPU: MAGMA uses CPU implementation for N < 3000



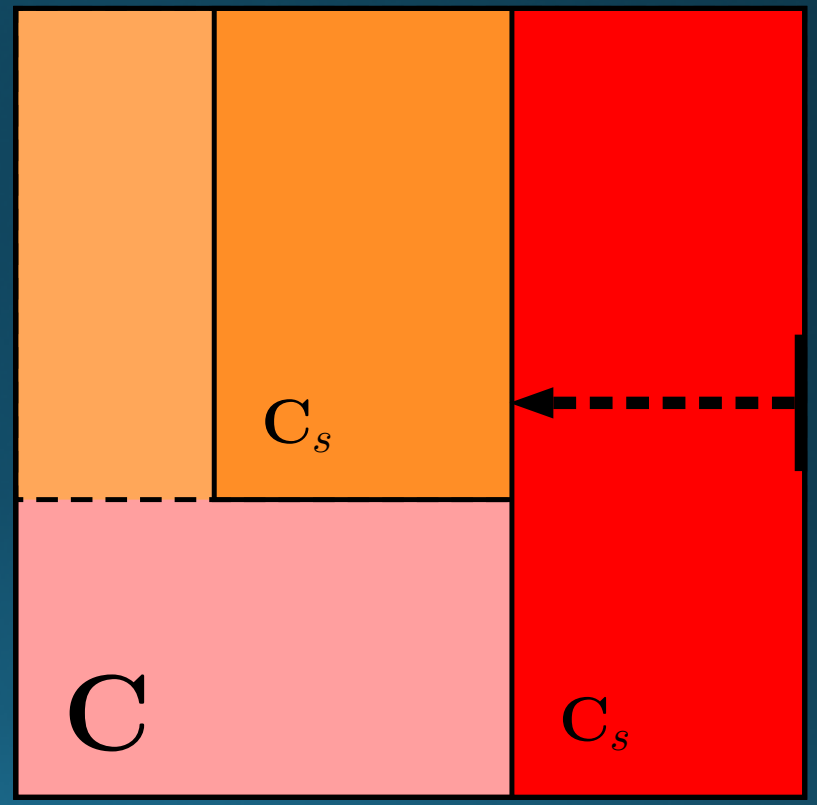
Step 2a: Tridiagonalize C

- Subblock tridiagonalization proceeds column-by-column from right to left
  - Many individual (hermetian) matrix-vector products, some involving full trailing submatrix
  - Computing householder reflectors involves inner products and scaling of vectors (`ZLARFG`)
  - Many performance hurdles to overcome on GPU



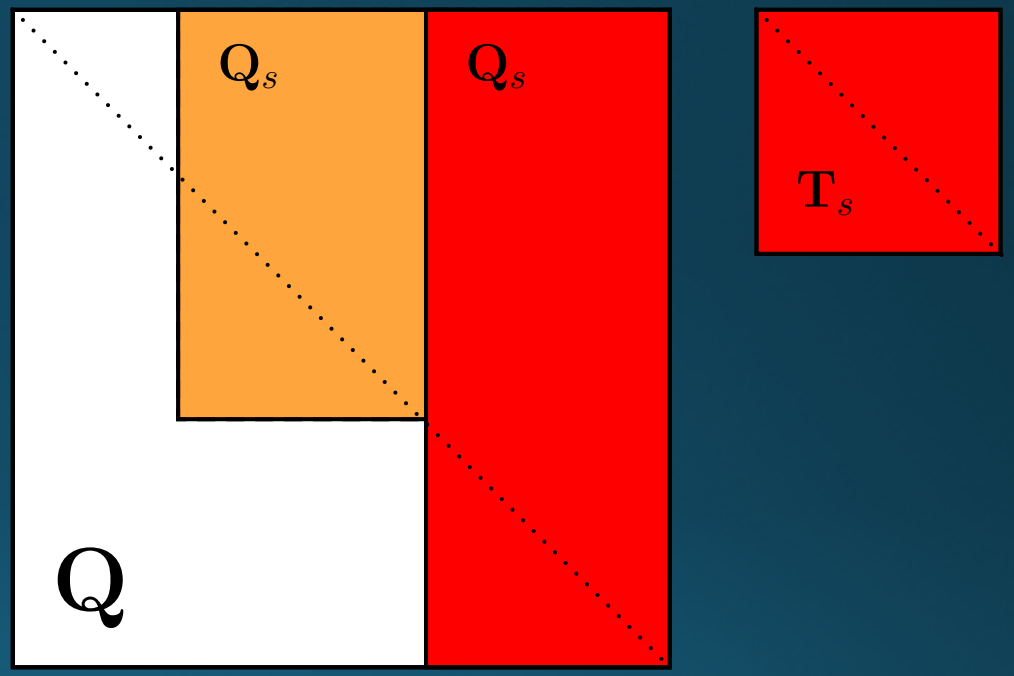
Step 2a: Tridiagonalize C

- For this routine, did not make any major modifications to the existing algorithm.
  - BLAS2 functions from CUBLAS
  - ZHEMV from MAGMA and also custom
- Strategies to achieve higher performance:
  - Fusing operations into single custom kernels, e.g. `ZLARFG`
  - Exposing and assigning independent operations to multiple streams
  - Goal: Preserve bandwidth and mask kernel launch latencies



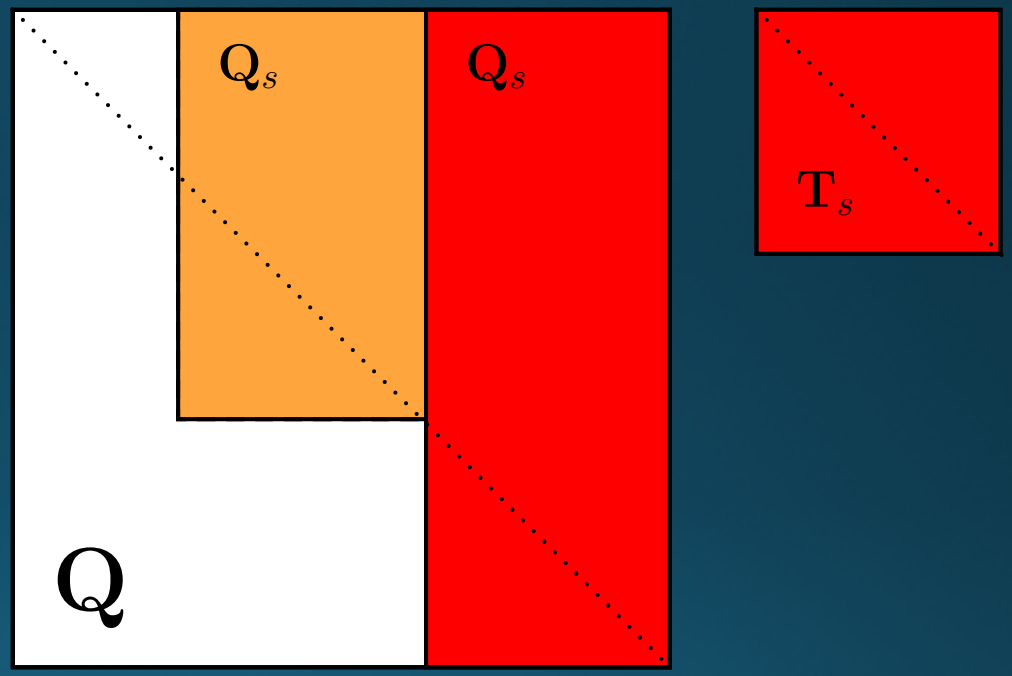
Step 2c: Reflect eigenvectors

- Corresponds to `ZUNMTR` routine in standard LAPACK library
  - Matrix updates: `ZLARFB(ZGEMM, ZTRMM)`
    - BLAS Level 3
    - Use CUBLAS functions
  - Subblock computation: `ZLARFT`
    - Mostly BLAS Level 2 (`ZGEMV, ZTRMV`)

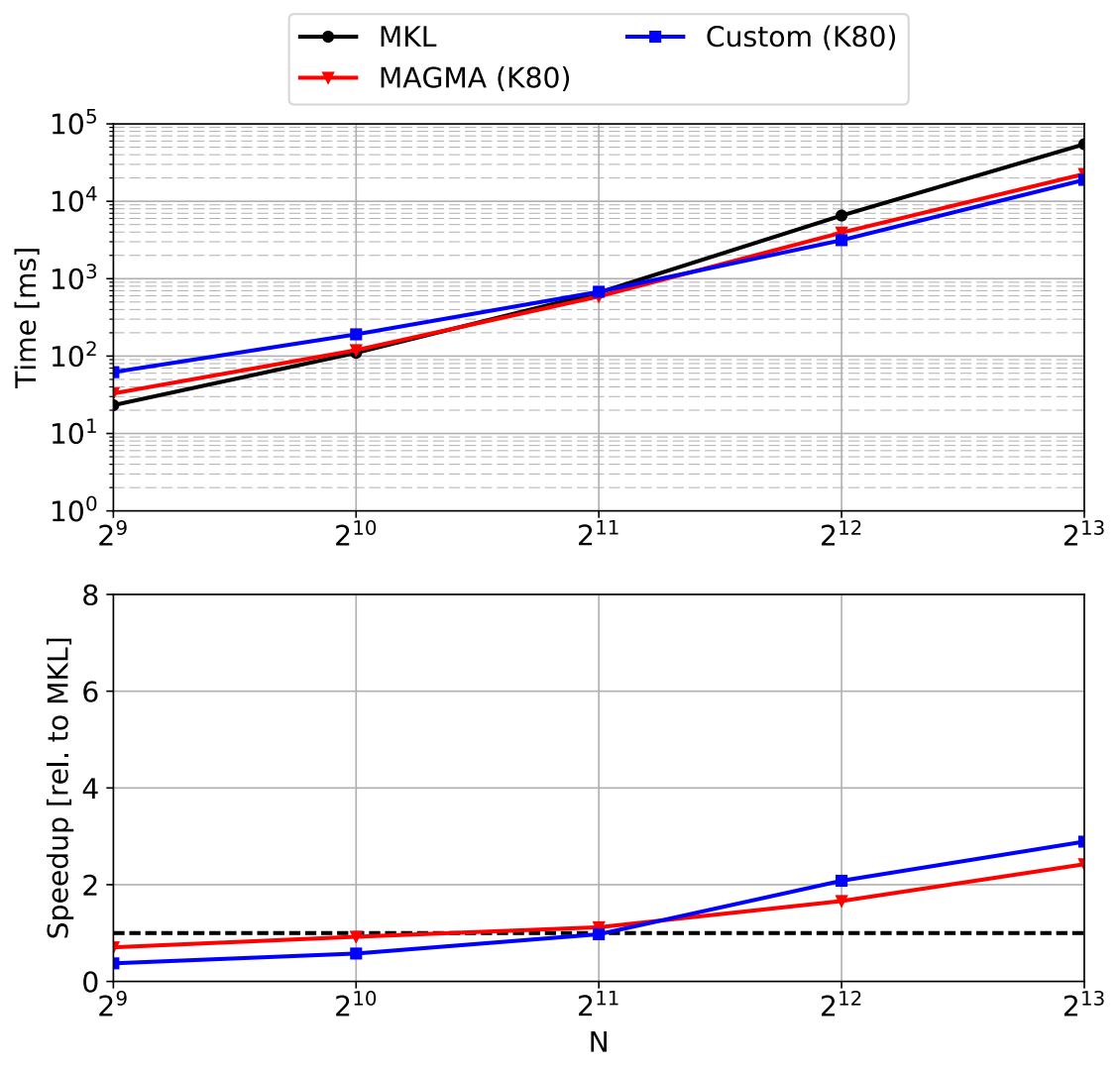


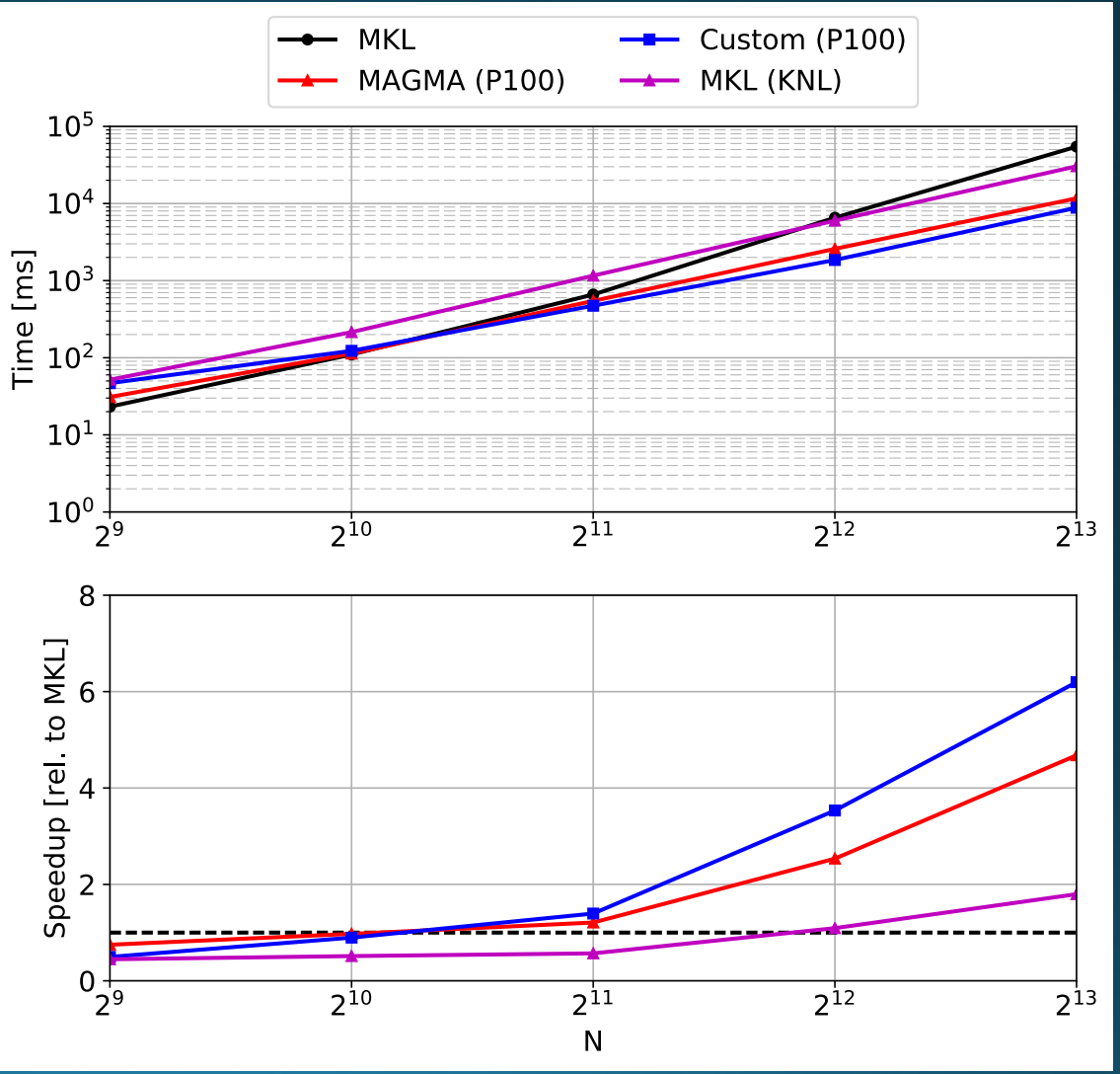
Step 2c: Reflect eigenvectors

- `ZLARFT` routine forms triangular reflector T to update eigenvectors
- Using modified procedure to enable use of BLAS 3 routines:
  1. Form preliminary T matrix using `ZHERK`:
     $$\mathbf{T}_s = \mathbf{Q}_s^H \mathbf{Q}_s$$
  2. Complete generation of T in custom kernel using shared memory
- Like ZHEGST, subblock computation placed in separate stream to preserve concurrency.



Step 2: ZHEEVD

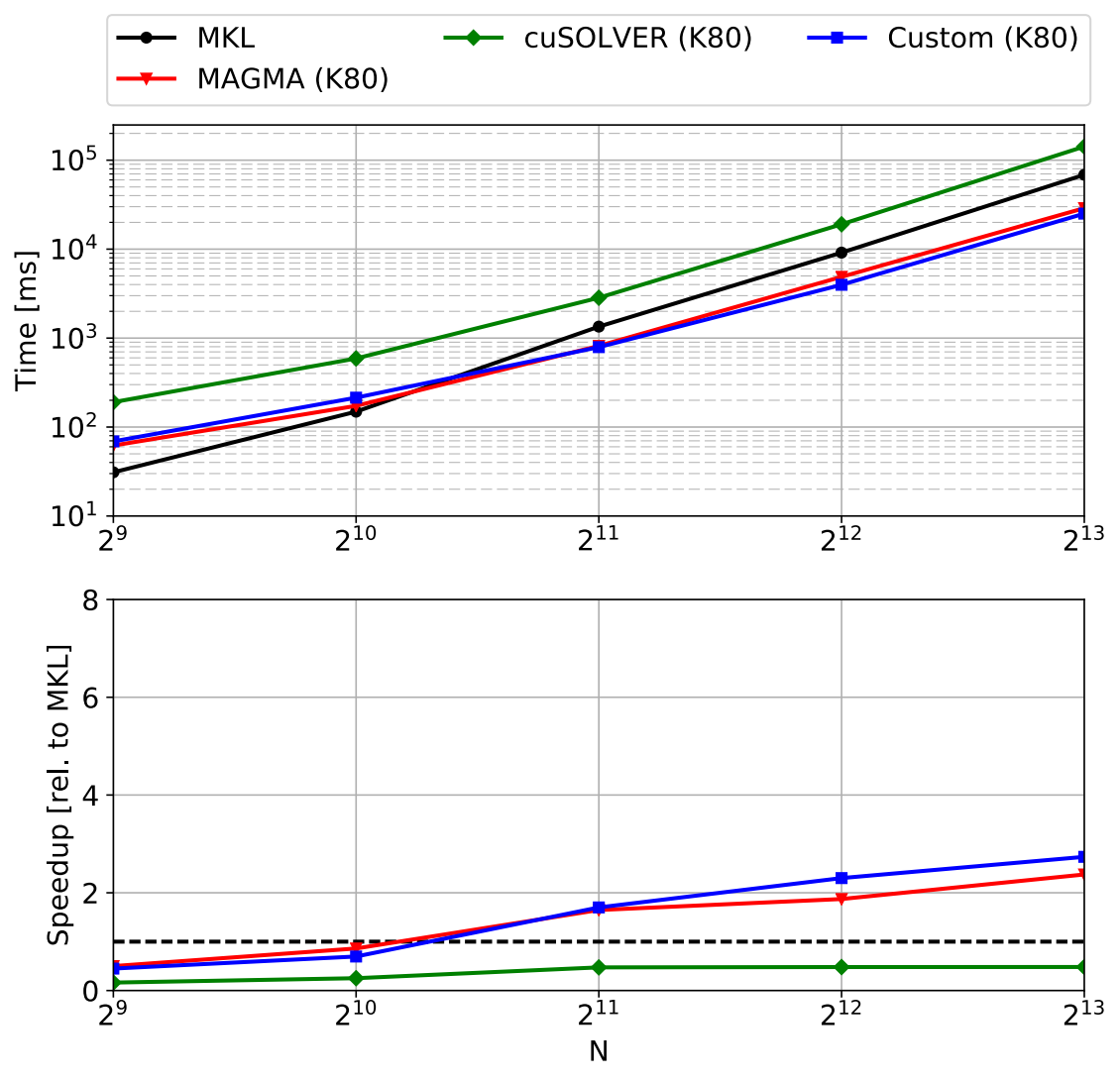


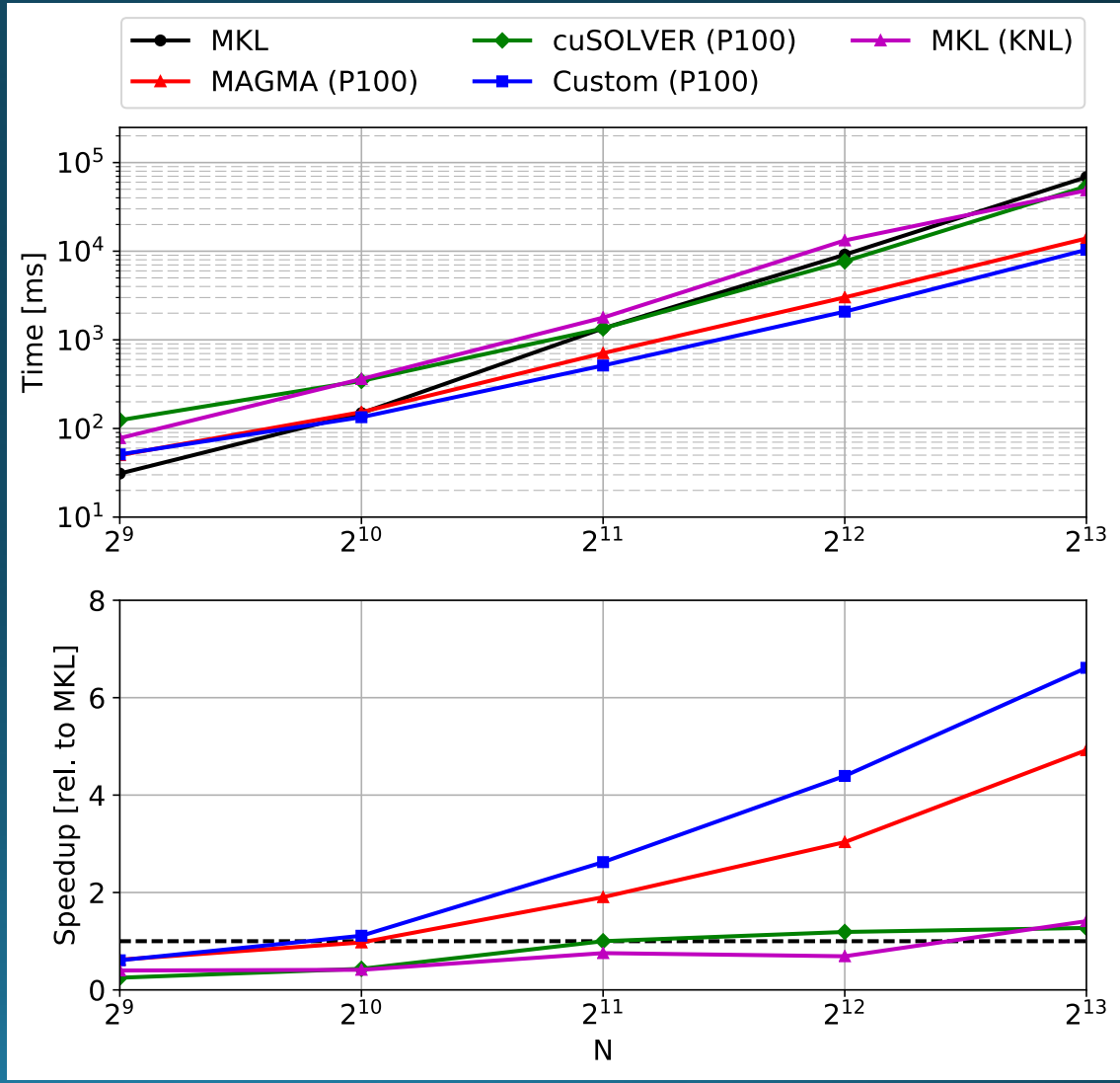


Step 3: Triangular solve

- Last step of computation to get final eigenvectors.
- Corresponds to `ZTRSM` BLAS routine. Can use implementation from CUBLAS library directly.

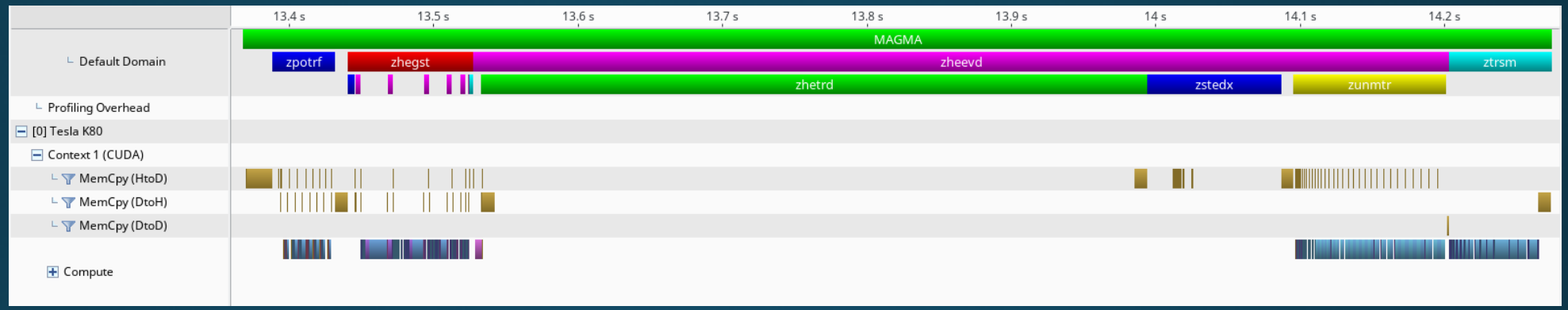
Full solve: ZHEGVD



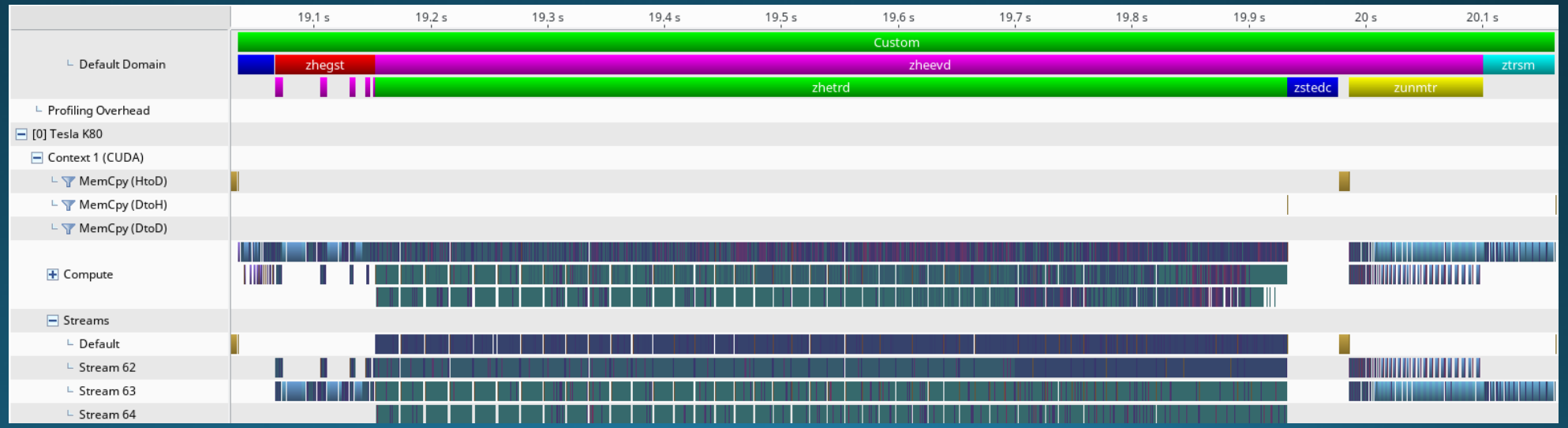


NVVP Timelines (N = 2048)

MAGMA



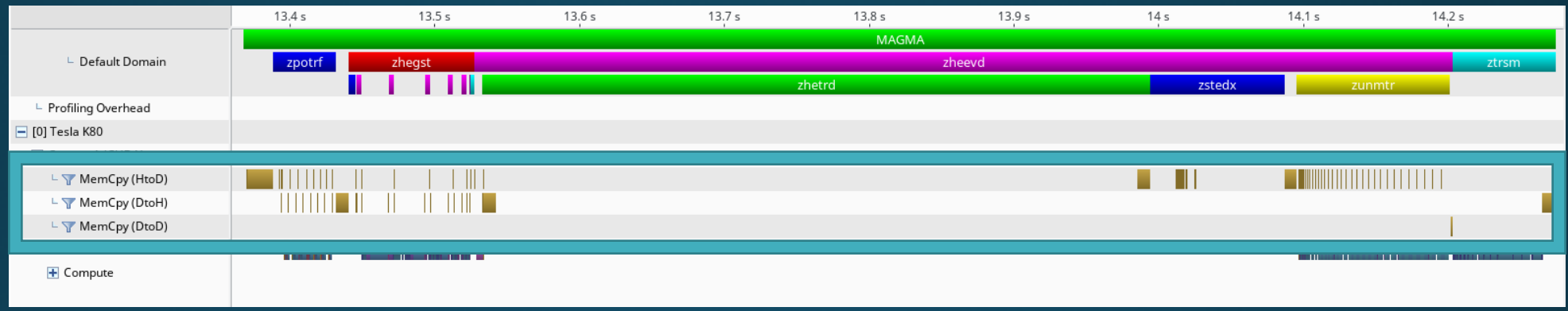
Custom



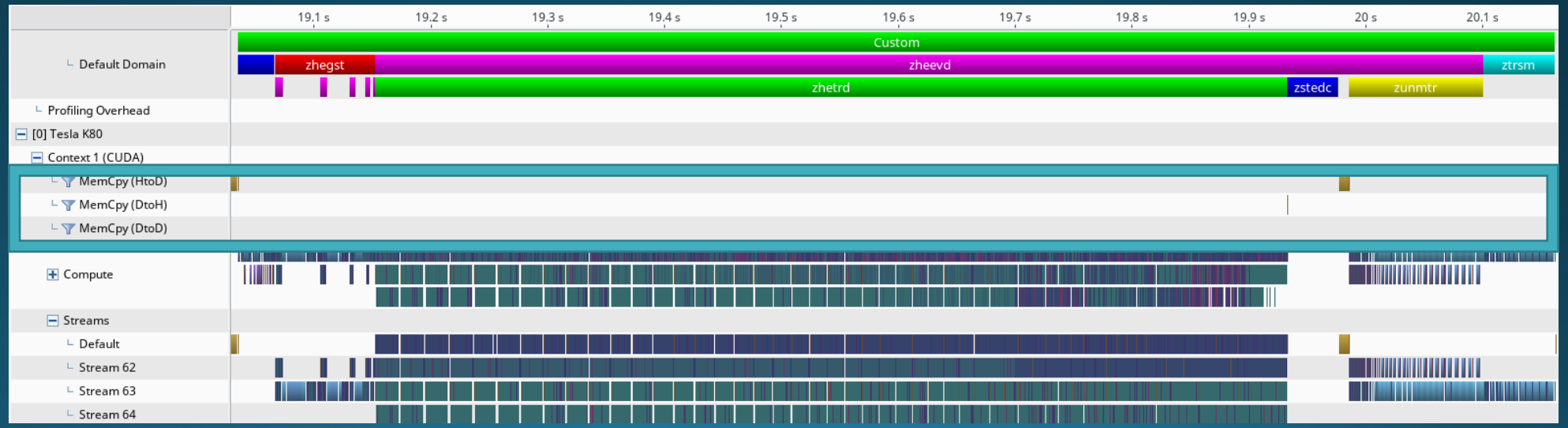
(Timelines not to scale)

NVVP Timelines (N = 2048)

MAGMA



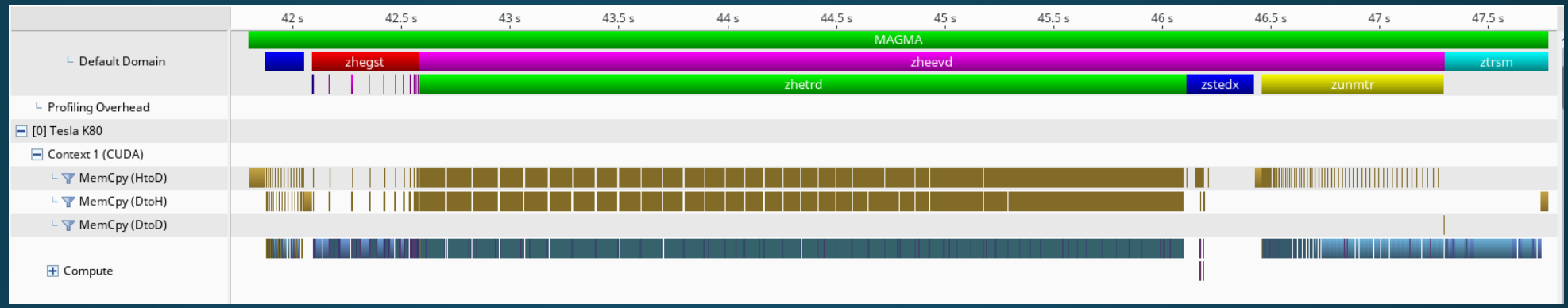
Custom



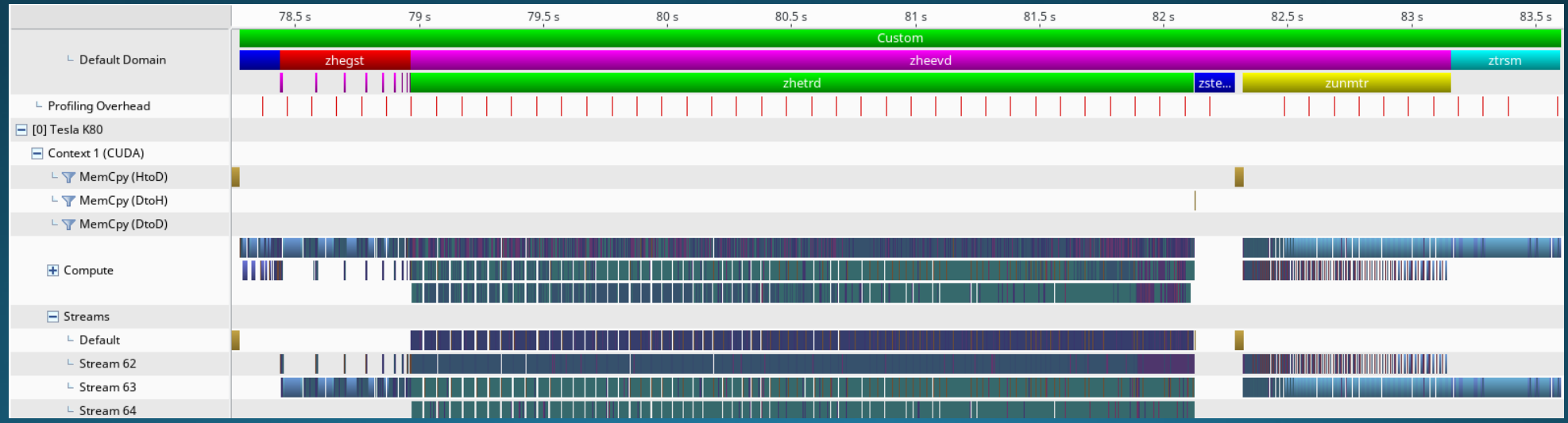
(Timelines not to scale)

NVVP Timelines (N = 4096)

MAGMA



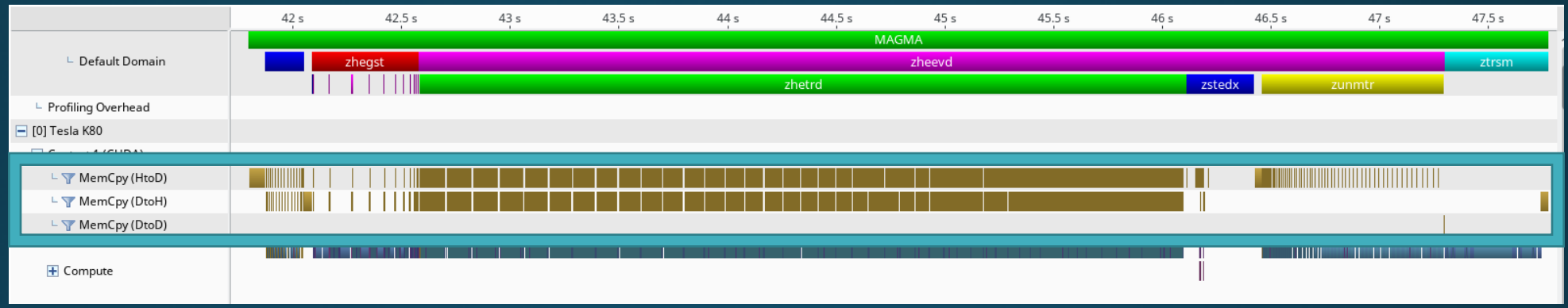
Custom



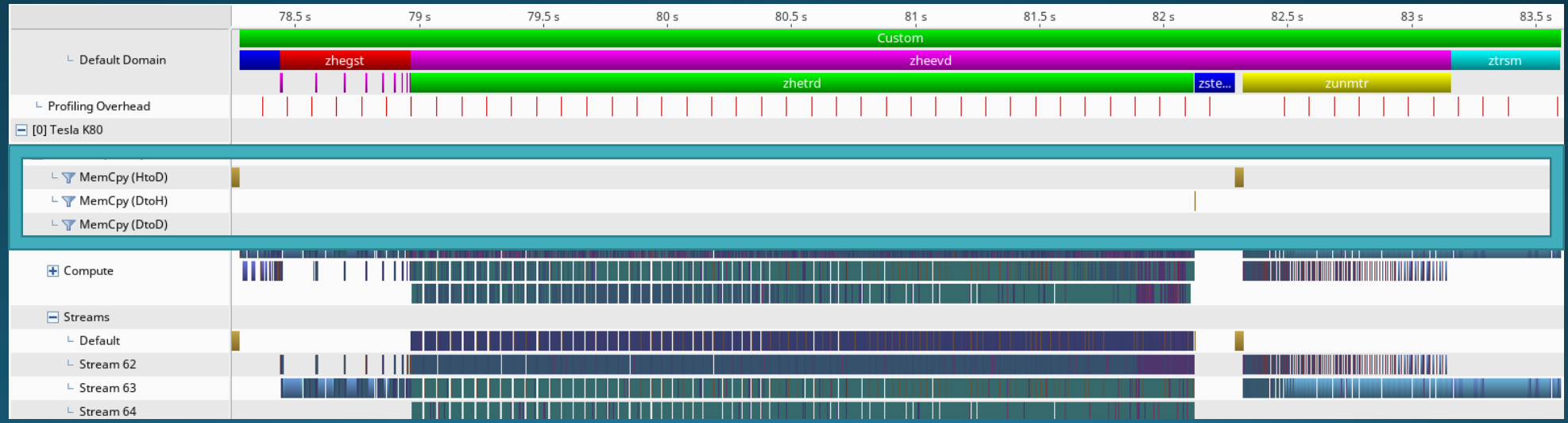
(Timelines not to scale)

NVVP Timelines (N = 4096)

MAGMA



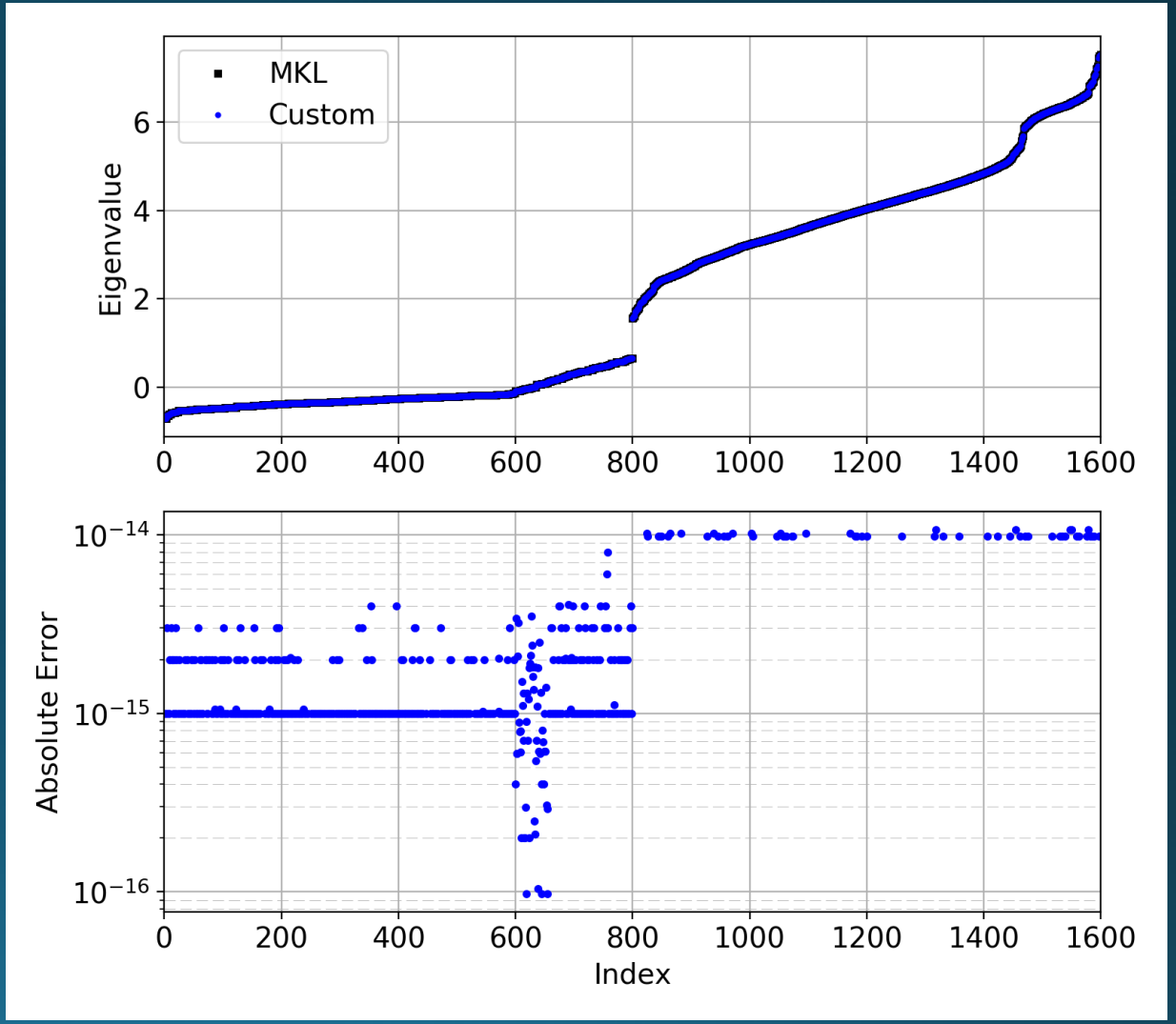
Custom



(Timelines not to scale)

Verification

- Eigenvalues computed using reference matrices (N = 1600) from QE.
- Results from custom solver show excellent agreement with MKL results.
- Full QE simulation results converge to correct solution



Conclusions and Future Work

- We have developed a new generalized eigensolver with limited CPU offloading:
  - Competitive performance for smaller N
  - Improved performance at larger N, especially when coupled with a faster GPU
  - Significant reduction in host-to-device data movement
  - Only uses CPU to compute eigenvalues/vectors of tridiagonal system

- Future work
  - Consider port of tridiagonal eigensolver to GPU to remove all dependencies on CPU resources for compute
  - Generalize code for alternative problem configurations

Thanks for listening!

- Questions?