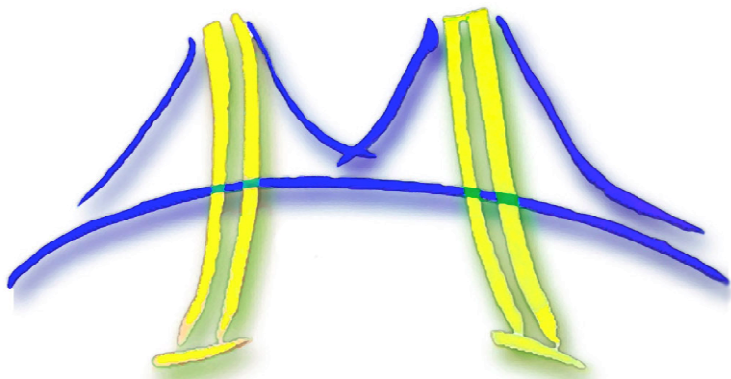


# CPU-GPU Hybrid Eigensolvers for Symmetric Eigenproblems

Bryan Catanzaro, Vasily Volkov,  
Bor-Yiing Su, Narayanan Sundaram,  
Jim Demmel, Kurt Keutzer



# Overview

- We're presenting work on CPU-GPU hybrid eigensolvers
  - Tridiagonal eigenvalue solver (SSTEBZ equivalent)
  - Inverse iterations eigenvector solver (SSTEIN equivalent)
  - Eigenvector solver for multiply-banded matrices (Laplacian matrix from stencil operation)
    - Arising from image segmentation problem

# Introduction

- Flops are getting cheaper, e.g. NVIDIA GTX280:

Peak arithmetic throughput	624 Gflop/s	(single precision $a*b+c$ )
Peak memory throughput	141 GB/s	18 flops per (32-bit) word
Kernel launch overhead	$\sim 5\mu s$	$\sim 3,000,000$ flops

- If bound by overhead, doing extra work may be free
  - If doing 1 flop is  $5\mu s$ , doing 300,000 flops is  $5.1\mu s$
- Reduce memory traffic at the cost of extra flops
  - Recompute matrix factors instead of retrieving them
- Try fast algorithms that might fail (but rarely)
  - Check for failure, recompute if needed

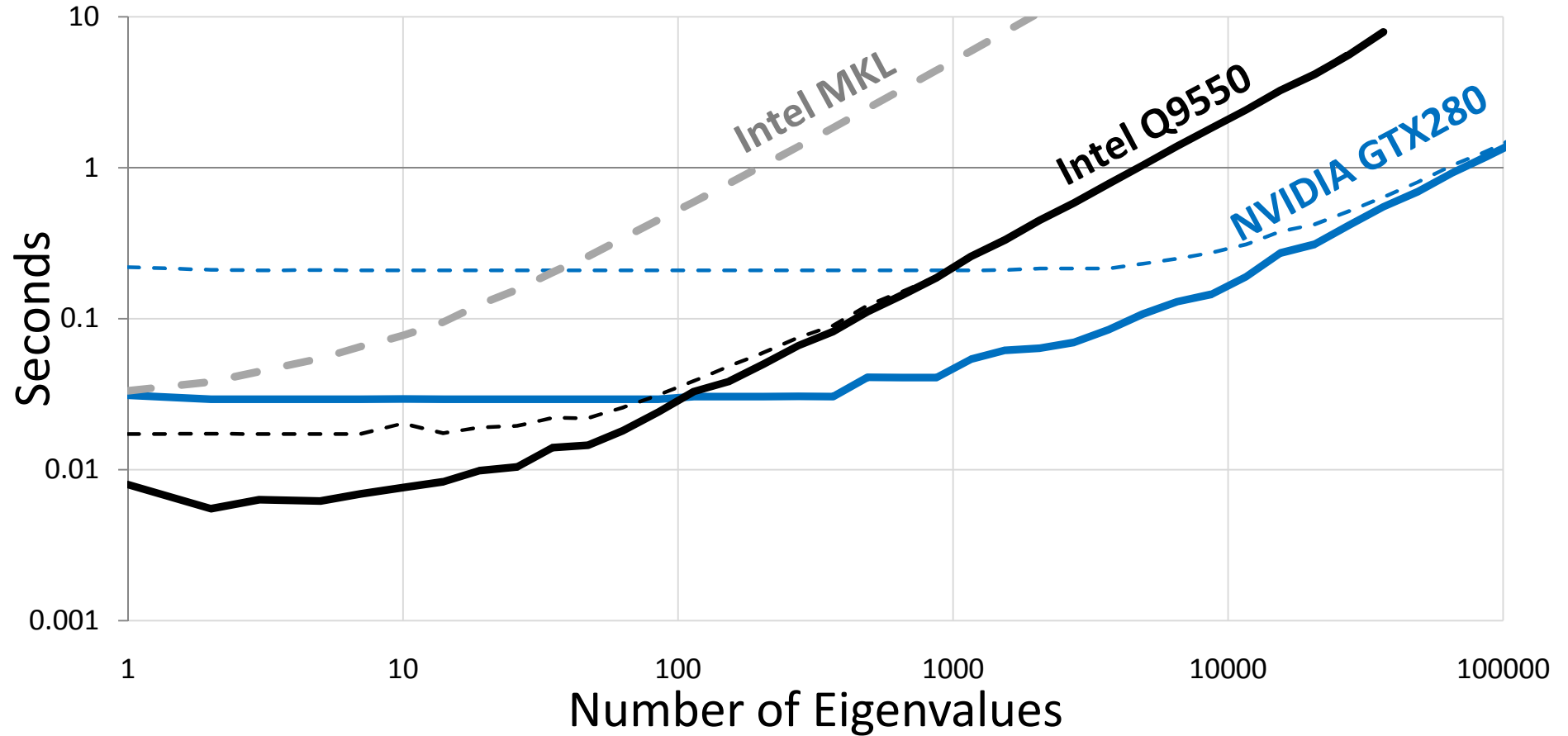
# Bisection Eigensolver (LAPACK's SSTEBSZ)

- Finding eigenvalues of symmetric tridiagonal  $T$  in  $[lb,ub]$ 
  - Let  $a(1...N)$  hold diagonal of  $T$ ,  $b(0...N-1)$  offdiagonal ( $b(0)=0$ )

```
Function Count(x,T) ... # eigenvalues of T < x
  Count = 0
  d = 1
  for j = 1 to N
    d = a(j) - x - b(j-1)2/d
  (*) if (d<0) then Count = Count + 1
```

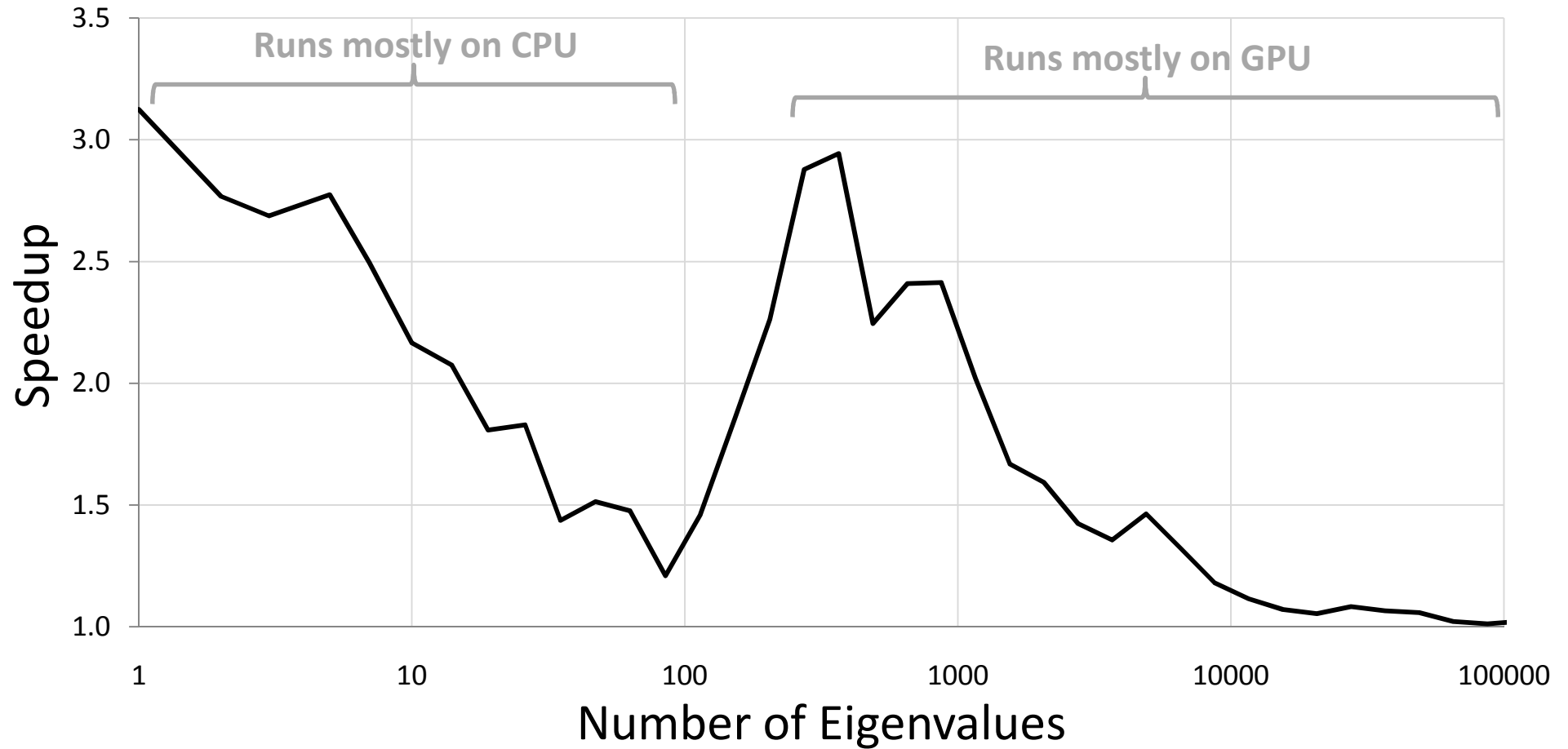
- If many values of  $x$  available, parallelize over them
  - Usual approach: repeated bisection of  $[lb,ub]$ 
    - Minimizes flops needed to find narrow intervals containing each eigenvalue
    - Not much parallelism at start (when there are only 1,2,4,... intervals)
  - Faster approach: Multisection of intervals into  $M \geq 2$  subintervals
    - Choose  $M$  on-the-fly by maximizing Efficiency =  $\log(M)/\text{time}(M)$
    - Use formula for  $\text{time}(M)$  using off-line benchmarking and data fitting

# Find a Subset of Eigenvalues, $N \approx 100,000$



- (dashed line – bisection, solid line – multisection)
- CPU is still faster at small problems!

# Speedup for Heterogeneous Algorithm



- Runs on CPU or GPU depending on N and current #intervals

# Inverse Iterations (LAPACK's SSTEIN)

- Inverse iteration for eigenvectors
  - Solve  $(T - \lambda I) z = x(i)$ ,  $x(i+1) = z / \|z\|$
- LAPACK's implementation: BLAS1 parallelism only
  - Computes one eigenvector at a time
  - Not vectorized, doesn't scale with #cores
- We compute all eigenvectors in parallel
  - Challenge: how to keep them orthogonal?
    - This is future work. (Post-process with QR?)

# Inverse iteration compute bound on GPU?

- Estimate required parallelism using Little's law
  - ALUs: 24 cycle latency, 240 ops/cycle throughput
  - 5760 independent operations must be in the flight
    - Otherwise – poor utilization of hardware
- Little parallelism in finding one eigenvector
  - Must solve for thousands eigenvectors in parallel
- How much space we have per eigenvector solve?
  - ~3MB on GPU in total
    - Registers, shared memory, texture caches
  - $3\text{MB}/5760 \approx 140$  single precision numbers
    - If  $N > 140$ , data doesn't fit on chip => **bandwidth bound**

# Reducing Memory Traffic in SSTEIN

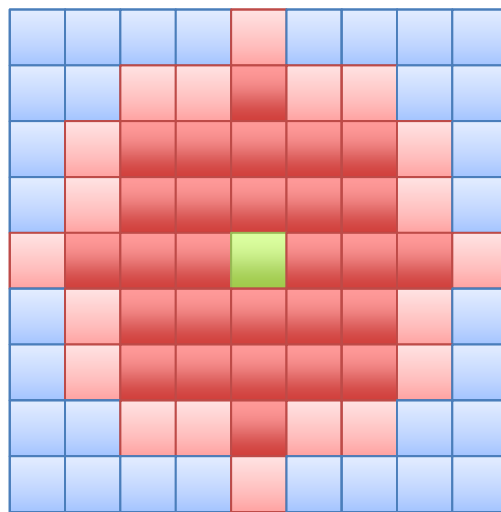
- Memory traffic in LAPACK's implementation:
  - Factorizing  $T - \lambda I$  produces  $4N$  words
  - Factorization must be read once per iteration ( $+4N$ )
  - Each iteration updates right hand side twice ( $+4N$ )
  - Total cost is  $4N + 8Nn_{\text{iter}}$  per one eigenvector
- First, abandon pivoting
  - I.e.  $T - \lambda I = LDL^T$  for diagonal  $D$ , unit bidiagonal  $L$
  - New total cost is  $2N + 6Nn_{\text{iter}}$
- Second, store only  $D^{-1}$ , recompute  $L$  using  $T$ 
  - New total cost is  $N + 5Nn_{\text{iter}}$ , 1.75x speedup for  $n_{\text{iter}} = 3$

# Conclusion

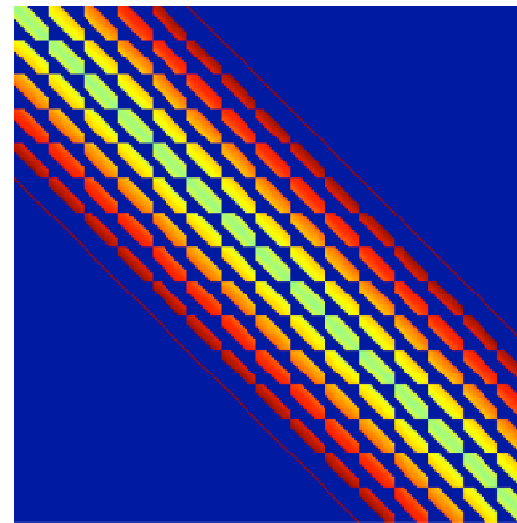
- Total memory traffic is  $16N$  per eigenvector
- Transferring result to CPU costs another  $N$ 
  - $16x$  smaller, but runs at  $20x$  lower throughput!
    - **Consumes half of the runtime**
    - But may still outperform CPU
- Practice:  $120x$  faster than LAPACK on CPU
  - However, computation failed in  $0.06\%$  cases
    - Other time  $8x$  lower accuracy was detected
    - This is the outcome of using no pivoting
    - Future work: recompute using safe algorithm

# From Image Segmentation to Eigenproblem

- Using spectral partitioning via Approximate Normalized Cuts
- Relaxing the problem from a binary decision to a real-valued indicator turns it into a generalized eigenproblem
- Find smallest  $k$  eigenvectors of affinity matrix measuring similarity between each pixel and its neighbors (Here,  $k=8$ )



Local pixel affinities



Affinity Matrix

nPixels

# Lanczos Algorithm and Computational Bottlenecks

**Algorithm:** Lanczos

**Input:**  $A$  (Symmetric Matrix)  
 $v$  (Initial Vector)

**Output:**  $\Theta$  (Ritz Values)  
 $X$  (Ritz Vectors)

1 Start with  $r \leftarrow v$  ;  
2  $\beta_0 \leftarrow \|r\|_2$  ;  
3 **for**  $j \leftarrow 1, 2, \dots$ , **until** convergence

4  $v_j \leftarrow r / \beta_{j-1}$  ;

5  $r \leftarrow Av_j$  ;

6  $r \leftarrow r - v_{j-1}\beta_{j-1}$  ;

7  $\alpha_j \leftarrow v_j^* r$  ;

8  $r \leftarrow r - v_j\alpha_j$  ;

9 Reorthogonalize if necessary ;

10  $\beta_j \leftarrow \|r\|_2$  ;

11 Compute Ritz values  $T_j = S\Theta S$  ;

12 Test bounds for convergence ;

13 **end for**

14 Compute Ritz vectors  $X \leftarrow V_j S$  ;

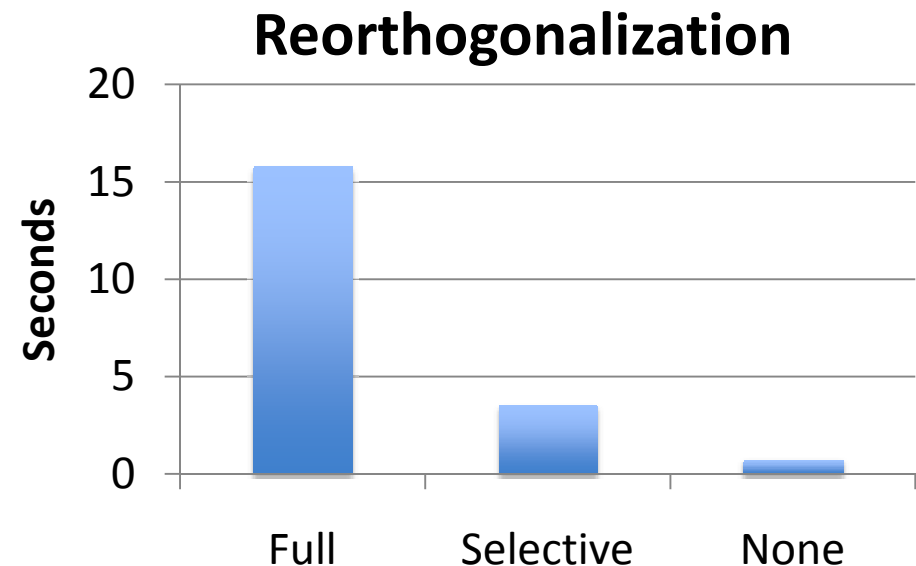
GPU

CPU

- Sparse matrix vector multiply
  - Use GPU with appropriate data structure
- Reorthogonalization
  - Choose method carefully

# Reorthogonalization

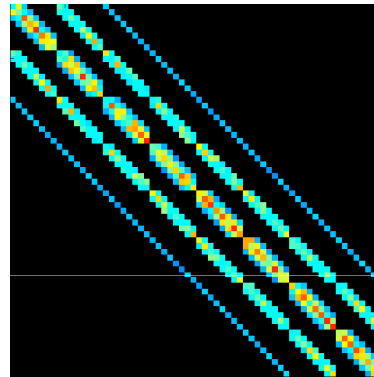
- We use a standard Lanczos algorithm, but without reorthogonalization
  - Cullum-Willoughby test removes spurious eigenvalues
- Convergence testing and tridiagonal eigensolve done on CPU (too small for GPU)
- Our eigensolver is 20x faster solely due to reorthogonalization technique
- Justified as multiple eigen values do not occur (would mean multiple equivalent segmentations)



# Sparse Vector Matrix Multiply

- The matrix is stored using a variant of Compressed Sparse Diagonal Format

- Similar approach to [1]

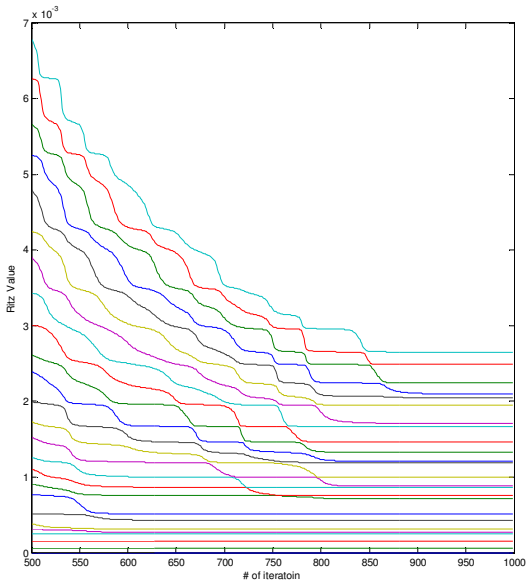


- For a matrix where  $n=154401$ , with 12.4M nonzeros, we achieve 39.5 SP GFLOP/s on Nvidia GTX280
  - Estimate: dual socket Nehalem, Single Precision, using the same matrix structure optimizations, should achieve  $\sim 24$  GFLOP/s

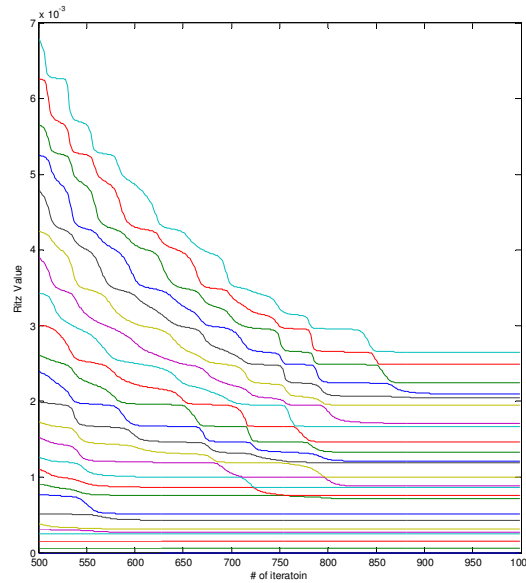
[1] Bell, Garland. "Implementing sparse matrix-vector multiplication on throughput-oriented processors." Supercomputing 2009

# Convergence Behavior

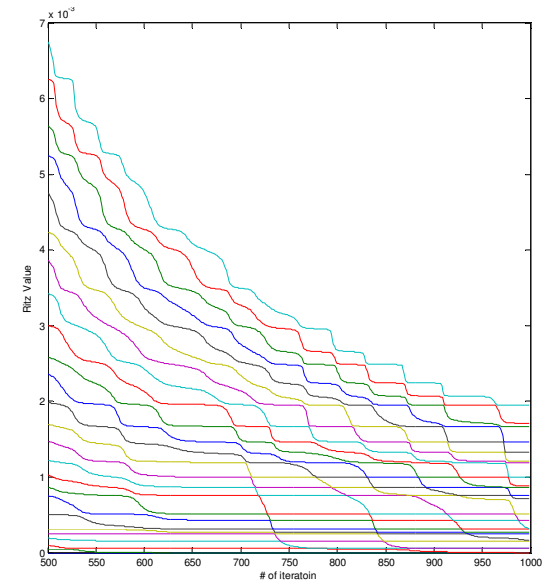
- The convergence behaviors of full reorthogonalization and selective reorthogonalization are similar
- Without reorthogonalization, the eigenvalues converge more slowly (more iterations), but overall is faster



Full Reorthogonalization



Selective Reorthogonalization

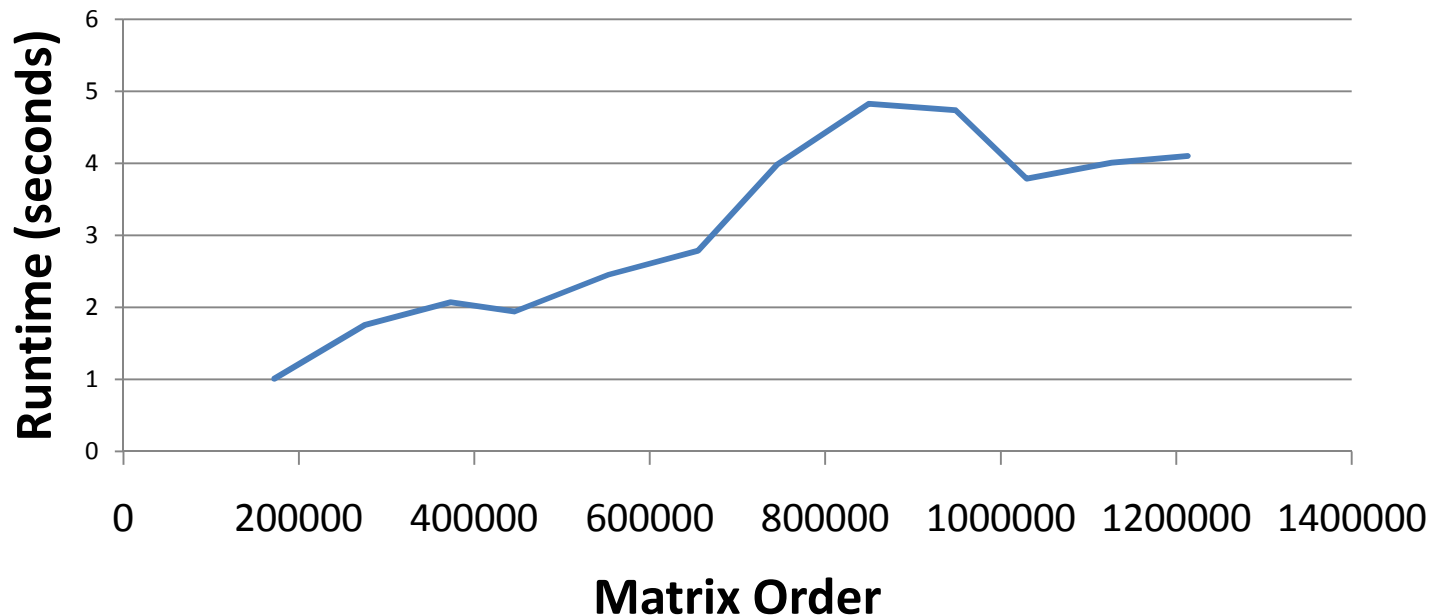


No Reorthogonalization

# Scalability

- Experiments run on Tesla C1060 (memory limited)
- If GPU memory runs out storing Ritz vectors, we revert to a 2-pass algorithm
  - First pass computes the Ritz values and the convergence limit
  - Second pass computes the eigenvectors from the Ritz values and Ritz vectors.

## Average Runtime versus Matrix Order

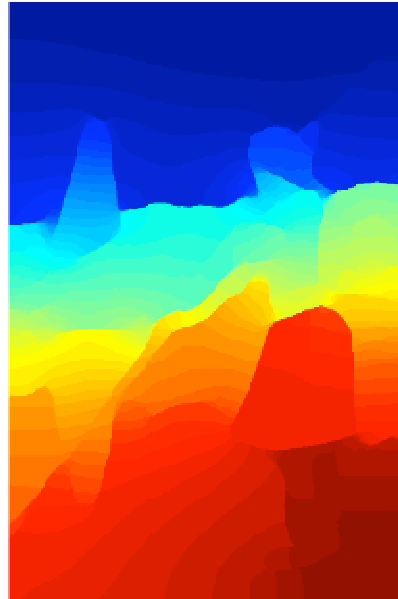


# Results

- Runtime is reduced 200x compared to MATLAB eigs running on all 8 cores of an Intel Core i7 @ 2.66 GHz
- Image contour quality using these eigenvectors is unchanged



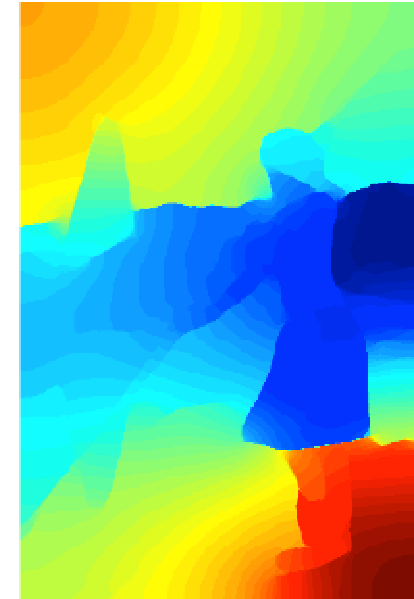
Original Image



Second Eigenvector  
from Hybrid Solver



Third Eigenvector  
from Hybrid Solver



Fourth Eigenvector  
from Hybrid Solver

# Conclusions

- CPU-GPU Hybrid eigensolvers can change the scope of practically solvable eigenproblems
- Algorithm and data structure selection is key
  - Favor simpler algorithms, even if they might fail
- Division of labor between CPU and GPU is problem specific
  - Using both together can provide speedup