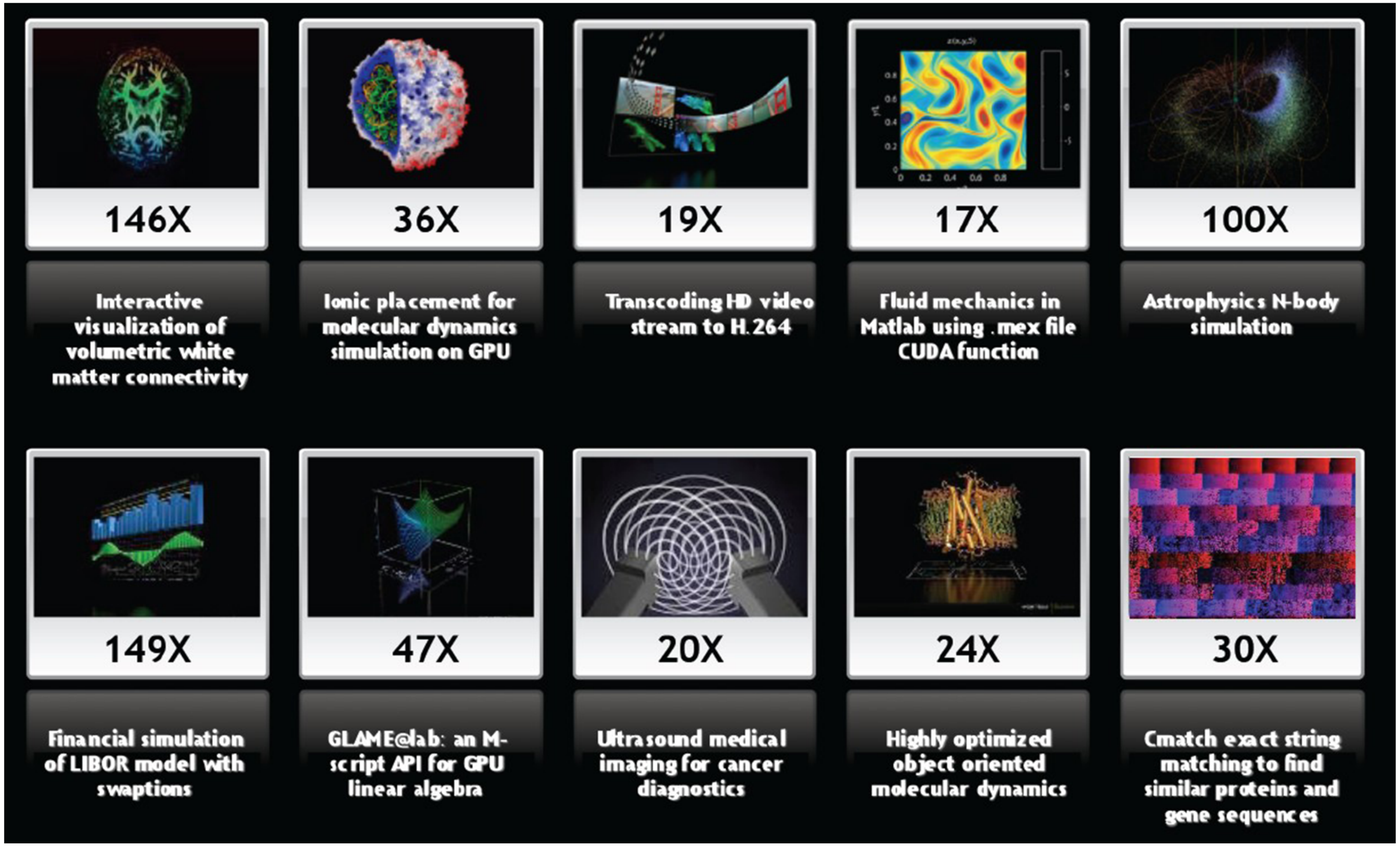
Modern GPUs (Graphics Processing Units)

- Powerful data-parallel computation platform.
- High computation density, high memory bandwidth.
- Relatively low-cost.

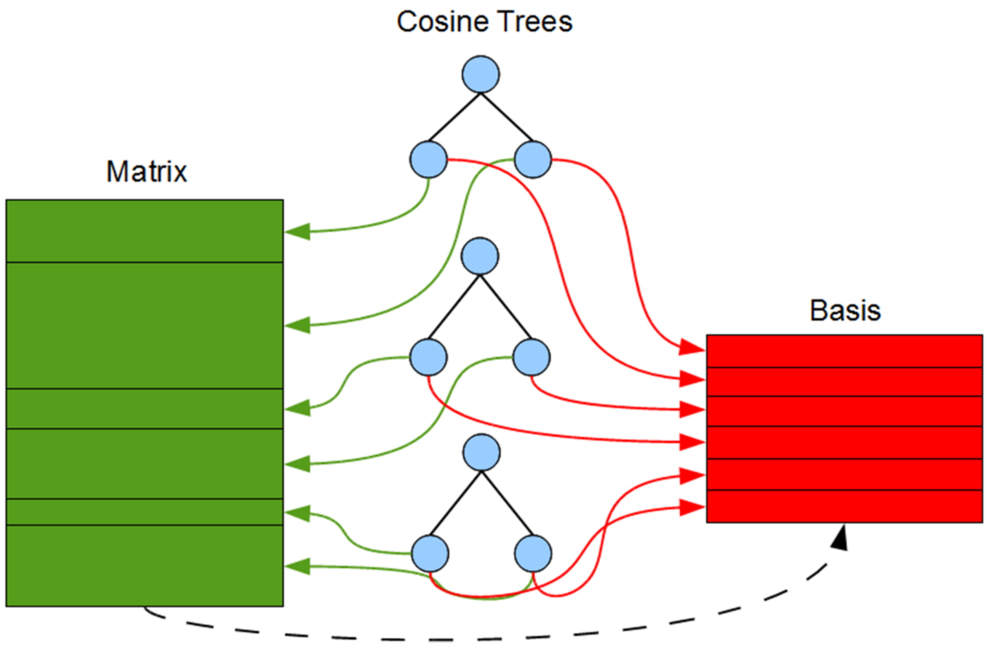
NVIDIA GTX 580
512 cores
1.6 Tera FLOPs
1.5 GB memory
200GB/s bandwidth
$499

GPU for Scientific Computing



[NVIDIA]

GPU-based QUIC-SVD Algorithm

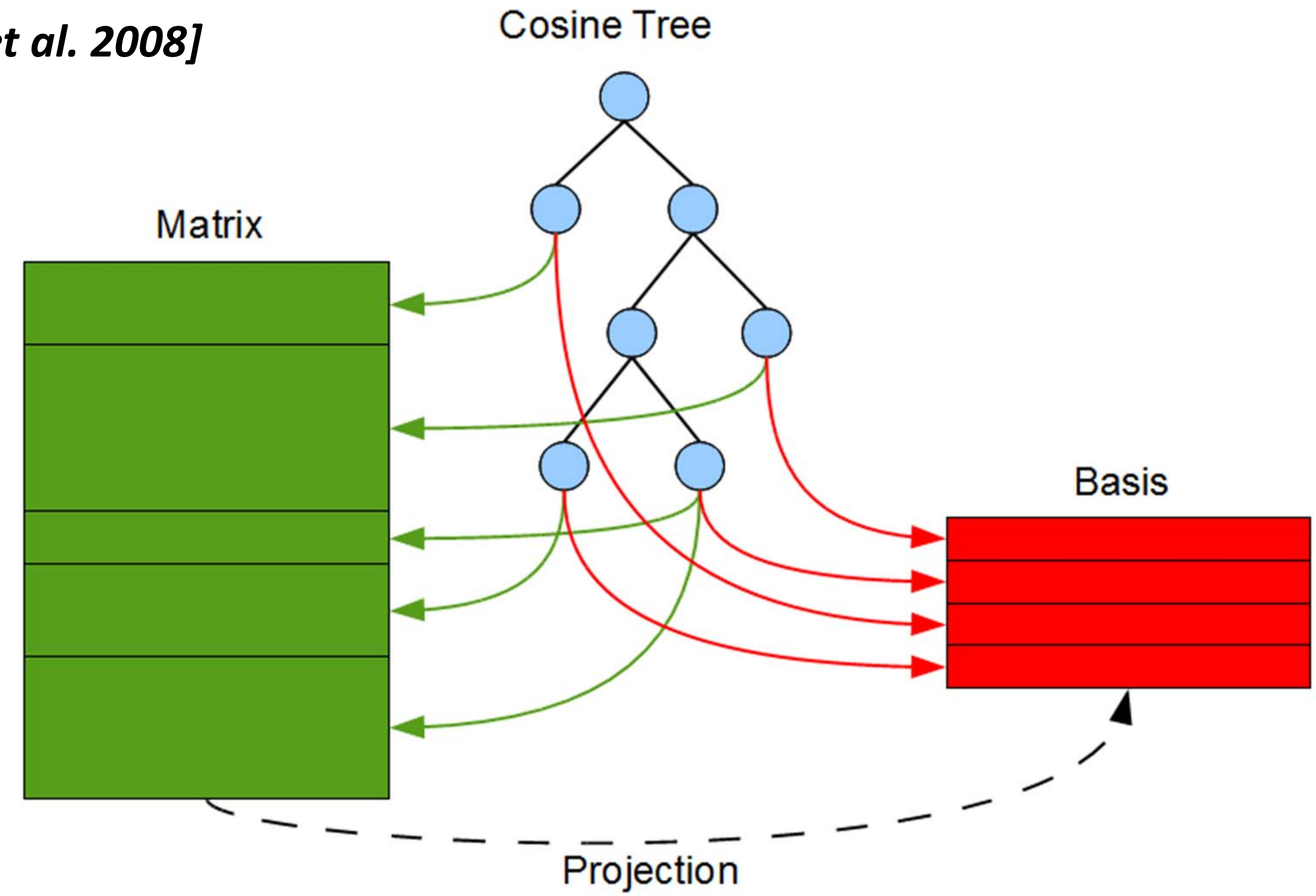


Introduction

- Singular Value Decomposition (SVD) of large matrices.
- Low-Rank Approximation.
- ***QUIC-SVD: Fast SVD using cosine trees****, by Michael Holmes, Alexander Gray and Charles Isbell, NIPS 2008.*

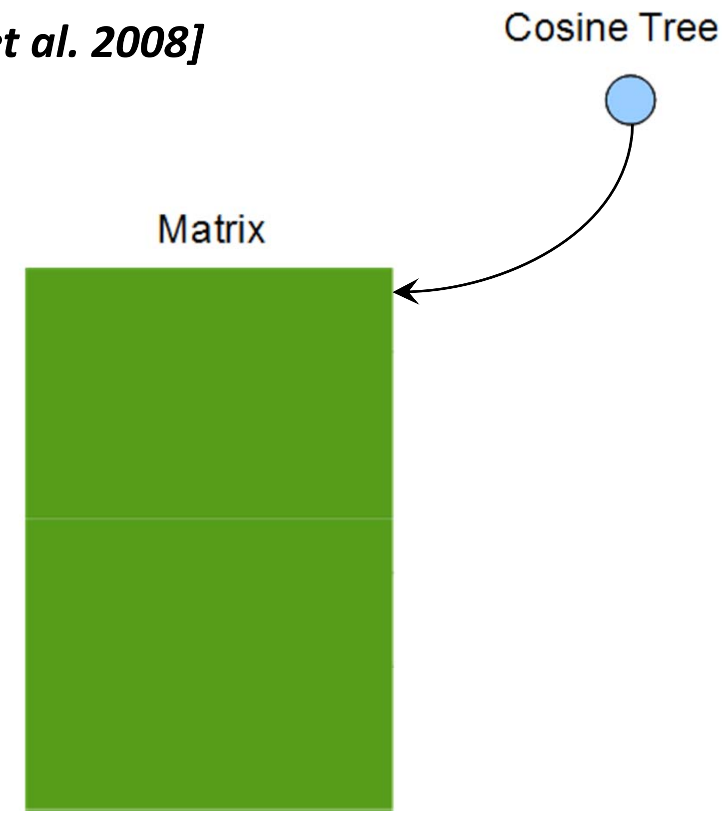
QUIC-SVD: Fast SVD using Cosine Trees

[Holmes et al. 2008]



QUIC-SVD: Fast SVD using Cosine Trees

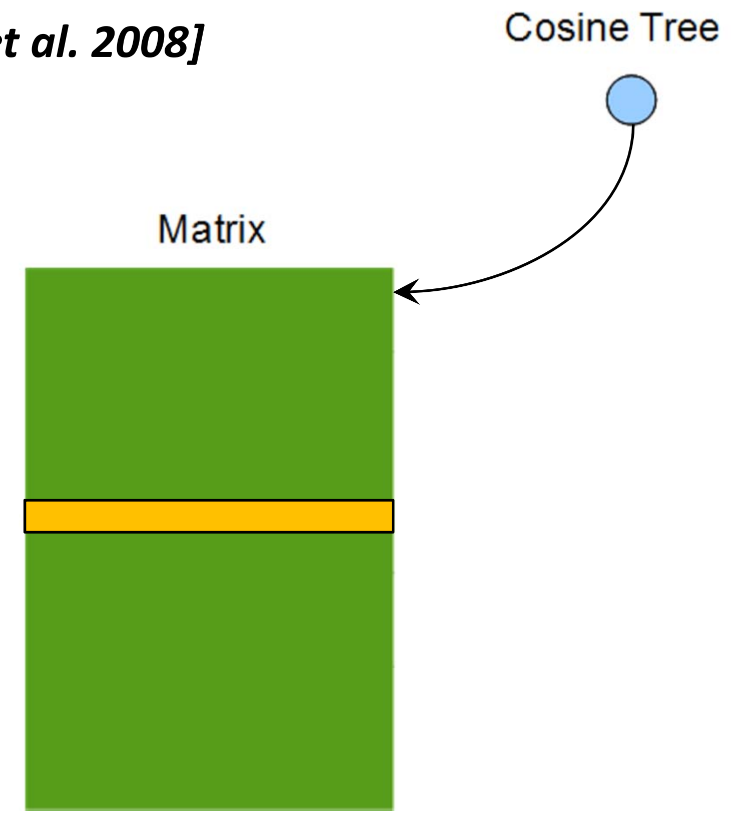
[Holmes et al. 2008]



Start from a root node that owns the entire matrix.

QUIC-SVD: Fast SVD using Cosine Trees

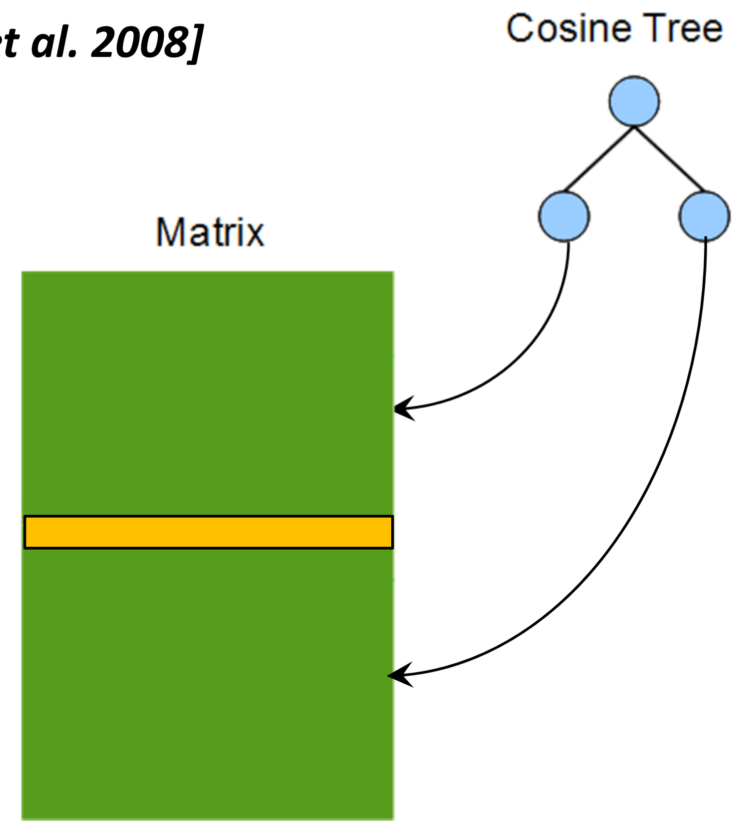
[Holmes et al. 2008]



Select pivot row based on length square distribution;
Compute inner product of every row with the pivot row;

QUIC-SVD: Fast SVD using Cosine Trees

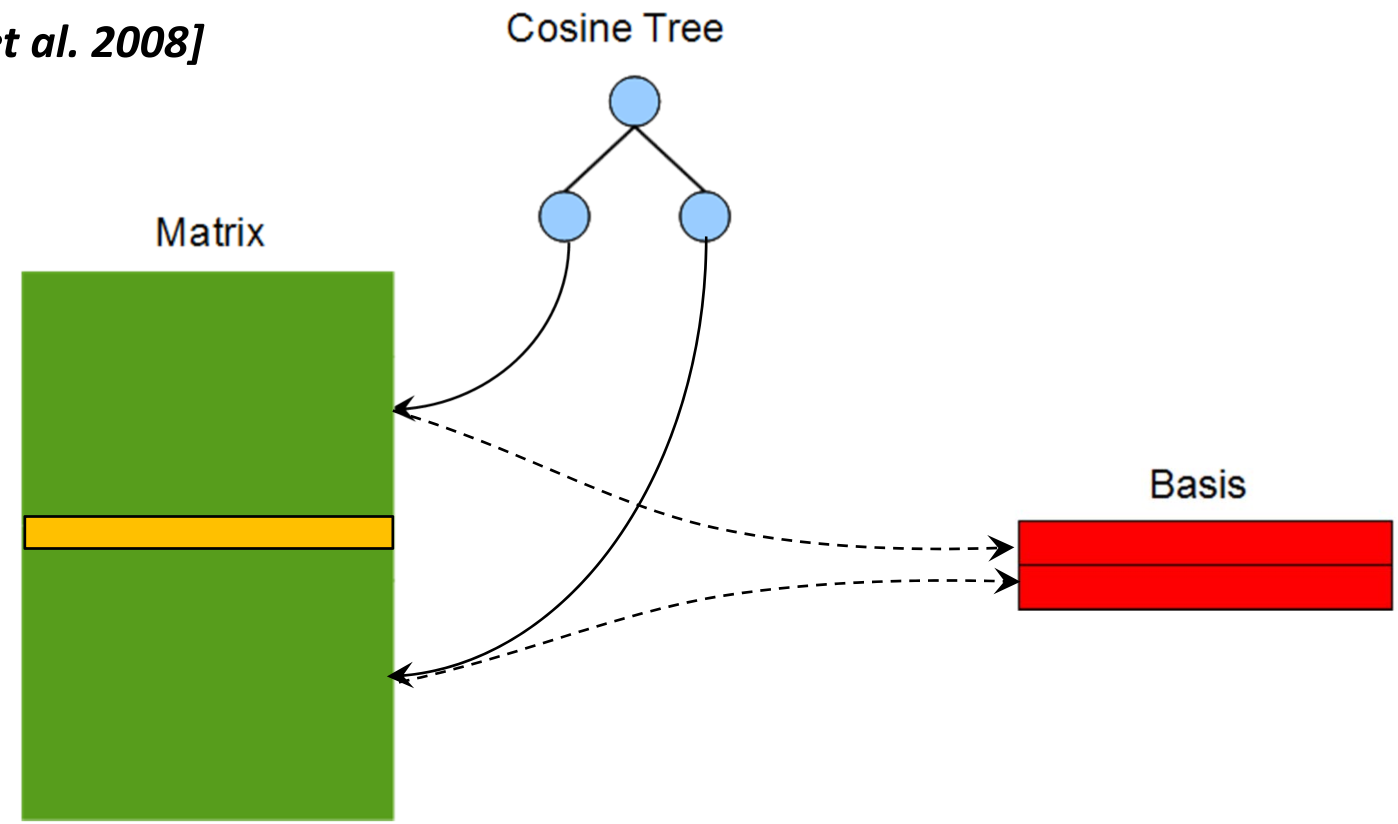
[Holmes et al. 2008]



The inner product results determine how the rows are partitioned to 2 subsets, each represented by a child node.

QUIC-SVD: Fast SVD using Cosine Trees

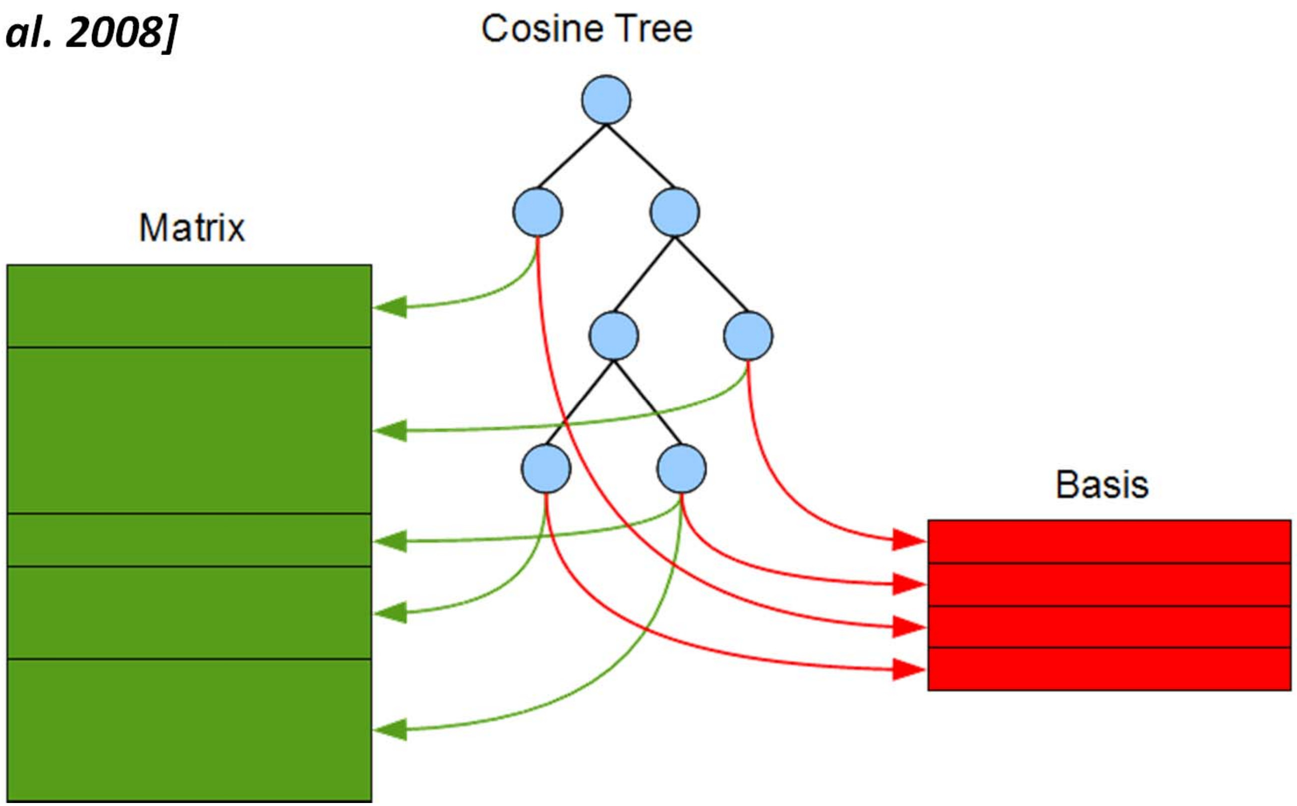
[Holmes et al. 2008]



Compute the row mean (centroid) of each subset;
add it to the current basis set (**keep orthogonalized**).

QUIC-SVD: Fast SVD using Cosine Trees

[Holmes et al. 2008]



Repeat, until the estimated error is below a threshold. The final basis set then provides a good low-rank approximation.

GPU-based Implementation of QUIC-SVD

- Most computation time is spent on splitting nodes.
- Computationally expensive steps:
  - **Computing vector inner products**
  - **Computing row means (centroids)**
  - **Basis orthogonalization**
- **Key**:
  find enough computation to keep the GPU busy!

Parallelize Vector Inner Products and Row Means

- Compute all inner products and row means in parallel.
- There are enough row vectors to keep the GPU busy.

Parallelize Gram Schmidt Orthogonalization

- **Classical Gram Schmidt:**
  Assume vectors $u_1, u_2, u_3 \ldots u_n$ are already orthgonalized, to orthogonalize a new vector v with respect to them:

$$u_{k+1} = v - \frac{\langle v, u_1 \rangle}{\langle u_1, u_1 \rangle} u_1 - \frac{\langle v, u_2 \rangle}{\langle u_2, u_2 \rangle} u_2 - \ldots - \frac{\langle v, u_n \rangle}{\langle u_n, u_n \rangle} u_n$$

- This is easy to parallelize (as each projection is independently calculated), but it has **<u>poor numerical stability</u>** due to rounding errors.

Parallelize Gram Schmidt Orthogonalization

- **Modified Gram Schmidt:**

$$v^{(1)} = v - \frac{\langle v, u_1 \rangle}{\langle u_1, u_1 \rangle} u_1, \quad v^{(2)} = v^{(1)} - \frac{\langle v^{(1)}, u_2 \rangle}{\langle u_2, u_2 \rangle} u_2, \quad v^{(3)} = v^{(2)} - \frac{\langle v^{(2)}, u_3 \rangle}{\langle u_3, u_3 \rangle} u_3, \quad ....$$

$$\ldots \quad u_{k+1} = v^{(k-1)} - \frac{\langle v^{(k-1)}, u_k \rangle}{\langle u_k, u_k \rangle} u_k$$

- This has good numerical stability, but is hard to parallelize, as the **computation is sequential**!

Parallelize Gram Schmidt Orthogonalization

- **Our approach:** ***Blocked Gram Schmidt***

$$v^{(1)} = v - \frac{\langle v, u_1 \rangle}{\langle u_1, u_1 \rangle} u_1 - \frac{\langle v, u_2 \rangle}{\langle u_2, u_2 \rangle} u_2 - \ldots - \frac{\langle v, u_m \rangle}{\langle u_m, u_m \rangle} u_m$$

$$v^{(2)} = v^{(1)} - \frac{\langle v^{(1)}, u_{m+1} \rangle}{\langle u_{m+1}, u_{m+1} \rangle} u_{m+1} - \frac{\langle v^{(1)}, u_{m+2} \rangle}{\langle u_{m+2}, u_{m+2} \rangle} u_{m+2} - \ldots - \frac{\langle v^{(1)}, u_{2m} \rangle}{\langle u_{2m}, u_{2m} \rangle} u_{2m}$$

$$u_{k+1} = v^{(2)} - \frac{\langle v^{(2)}, u_{2m+1} \rangle}{\langle u_{2m+1}, u_{2m+1} \rangle} u_{2m+1} - \frac{\langle v^{(2)}, u_{2m+2} \rangle}{\langle u_{2m+2}, u_{2m+2} \rangle} u_{m+2} - \ldots - \frac{\langle v^{(2)}, u_k \rangle}{\langle u_k, u_k \rangle} u_k$$

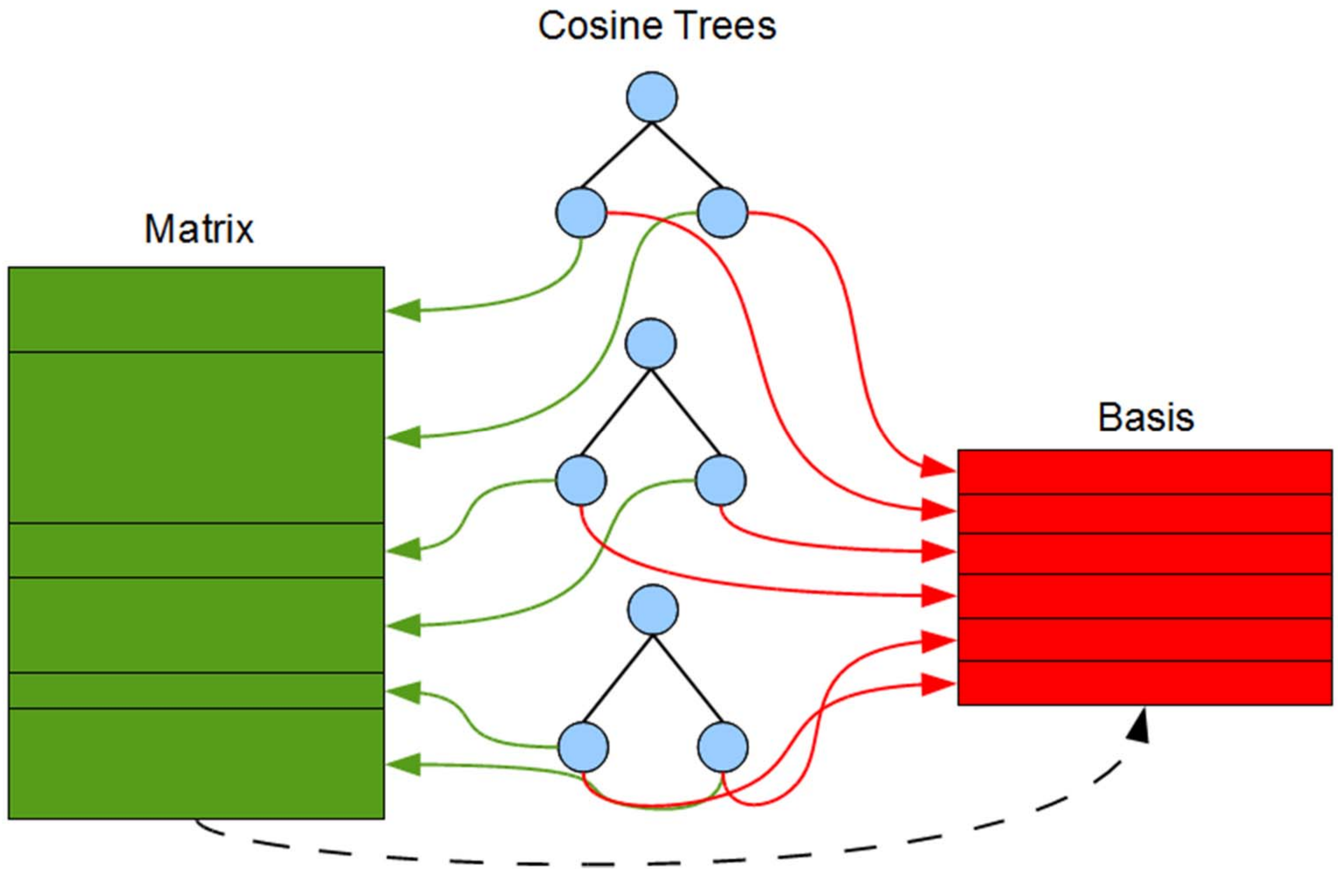
- This hybrid approach proves to be both **numerically stable**, and **GPU-friendly**.

Partitioned QUIC-SVD

- As the advantage of exploiting the GPU is only obvious for large-scale problems, we need to test our algorithm on large matrices (>10,000x10,000).
- For dense matrices, this will soon become an out-of-core problem.
- We modified our algorithm to use a matrix partitioning scheme, so that each partition can fit in GPU memory.

UMassAmherst

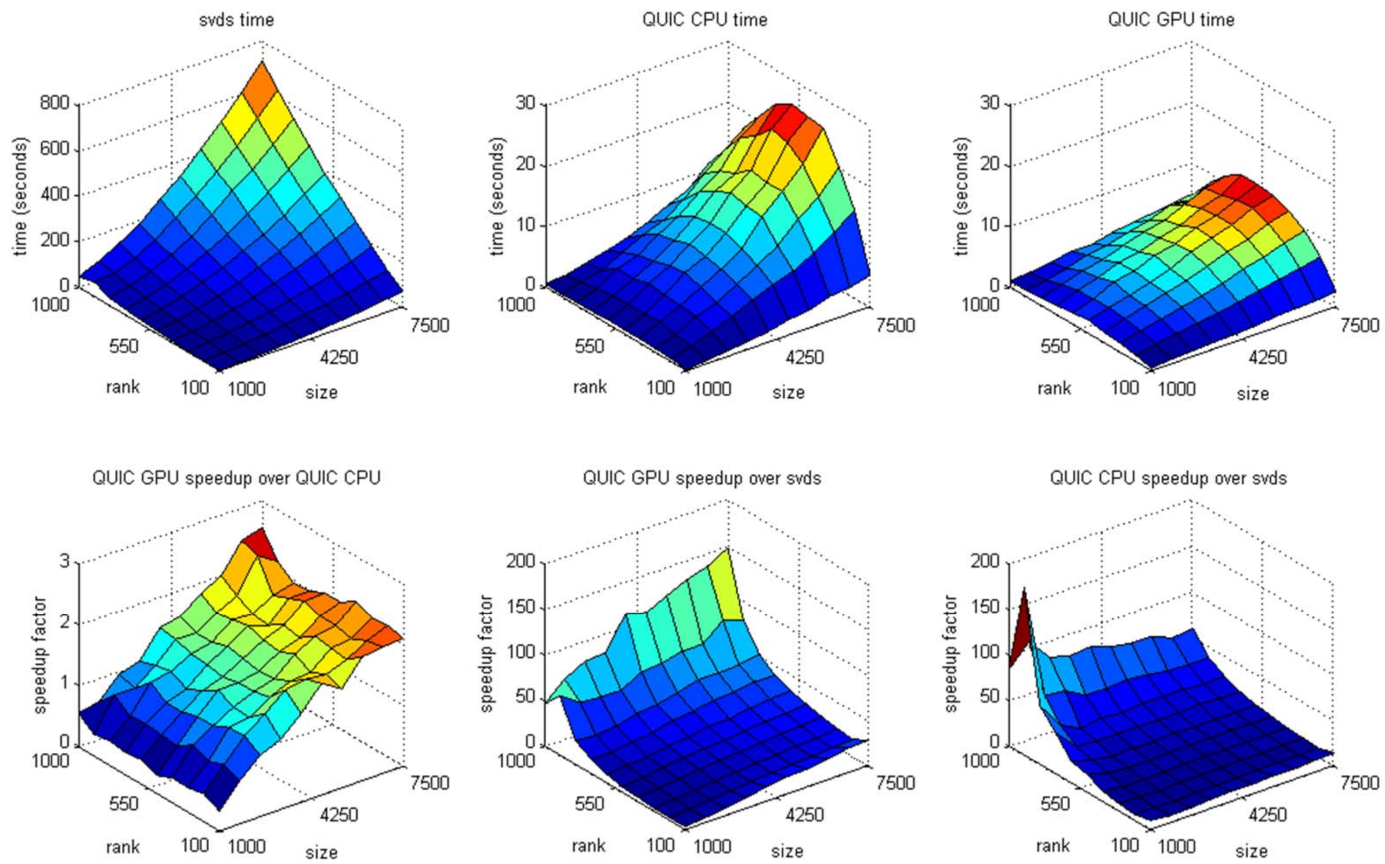
Partitioned QUIC-SVD



Performance

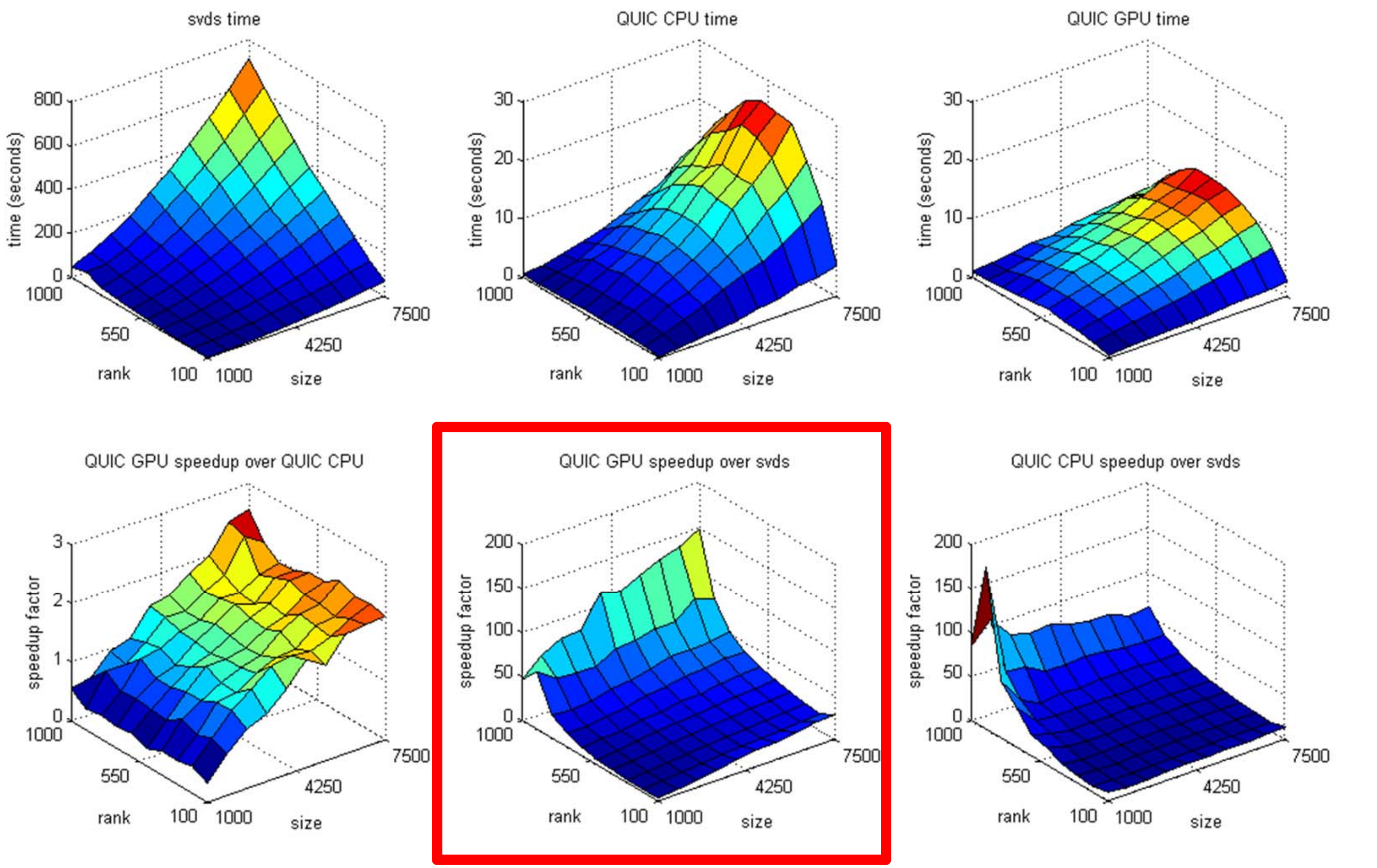
- We generate test data by multiplying two randomly generated left and right matrices, resulting in a low-rank matrix.
- We set the termination error in QUIC-SVD to 10^{-6}. This forces the algorithm to produce accurate results.
- We compare 1) Matlab's **svds**; 2) CPU-implementation of QUIC-SVD; 3) GPU-implementation of QUIC-SVD.
  - CPU version considerably optimized using Intel MKL and tested on an Intel Core i7
  - GPU version is tested on an NVIDIA 480 GTX

Performance

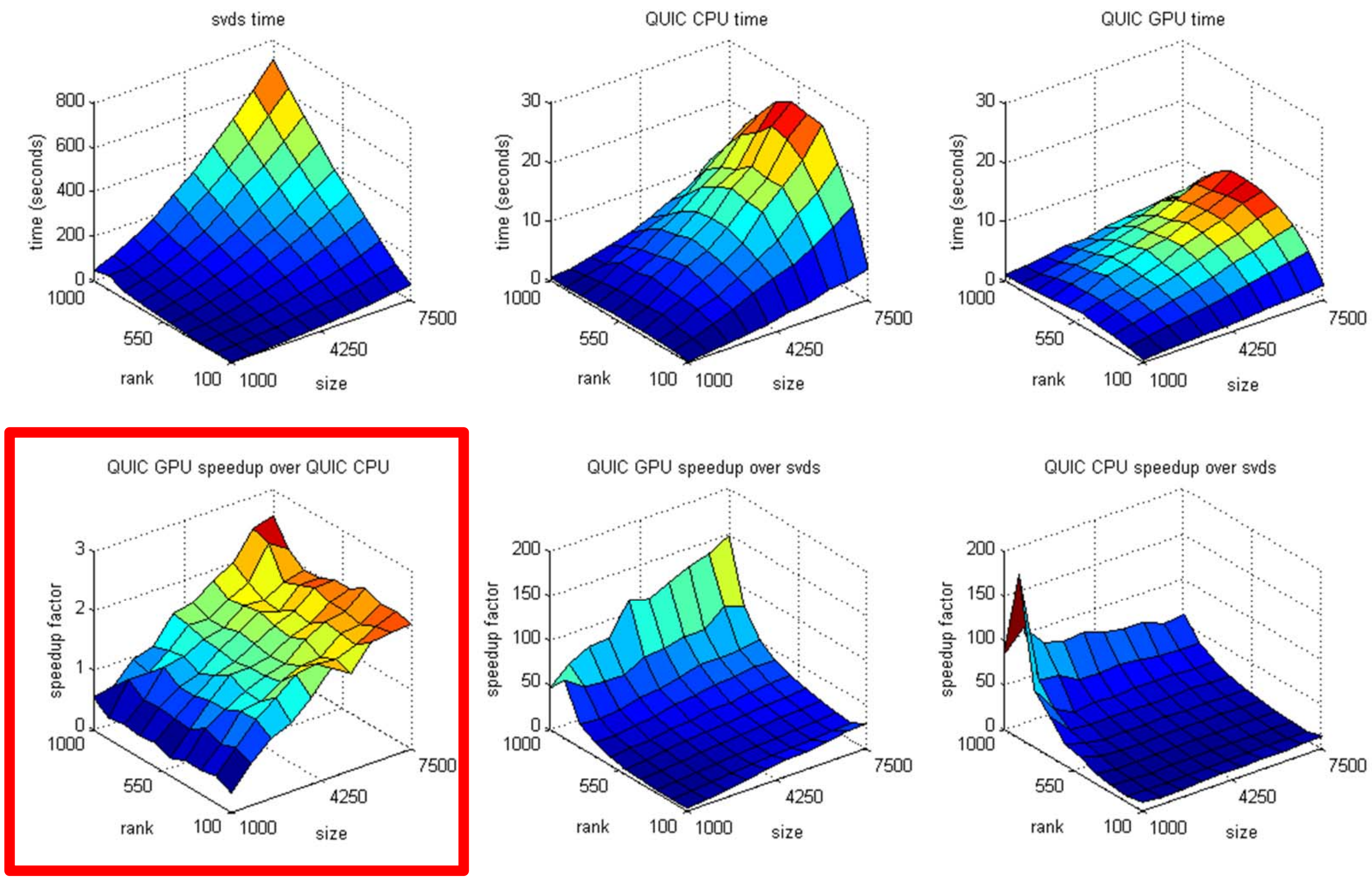


UMassAmherst

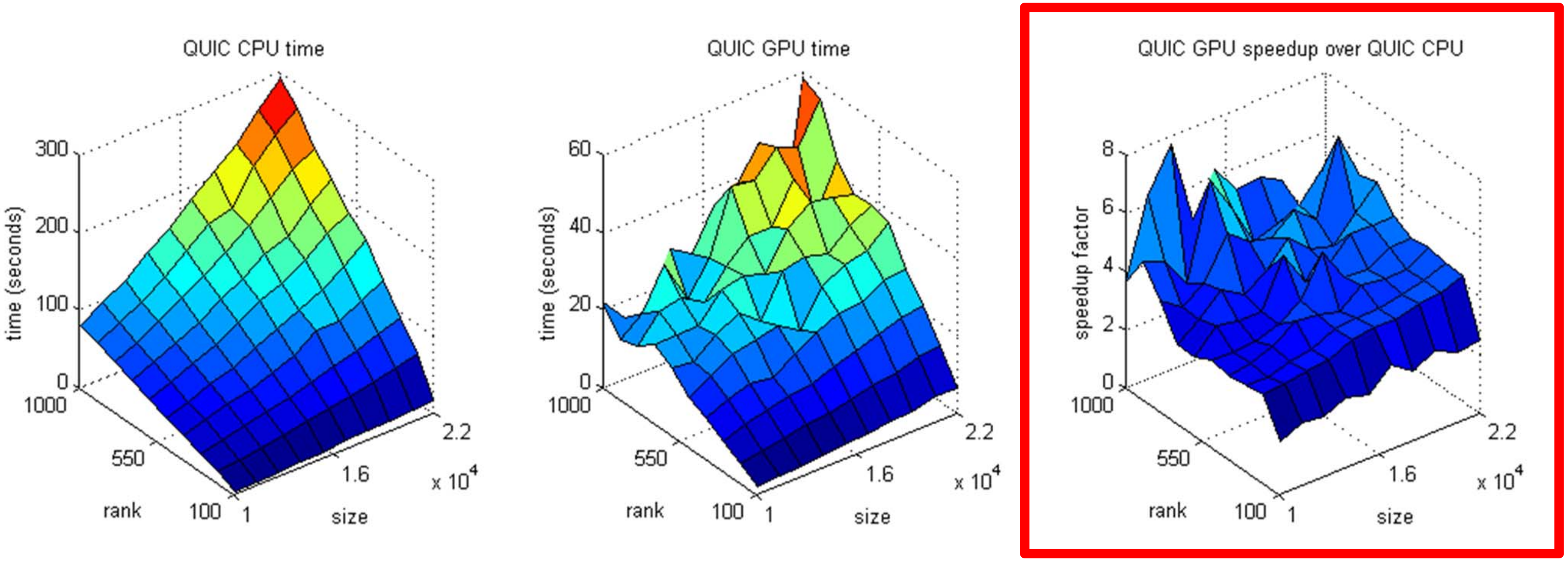
Performance



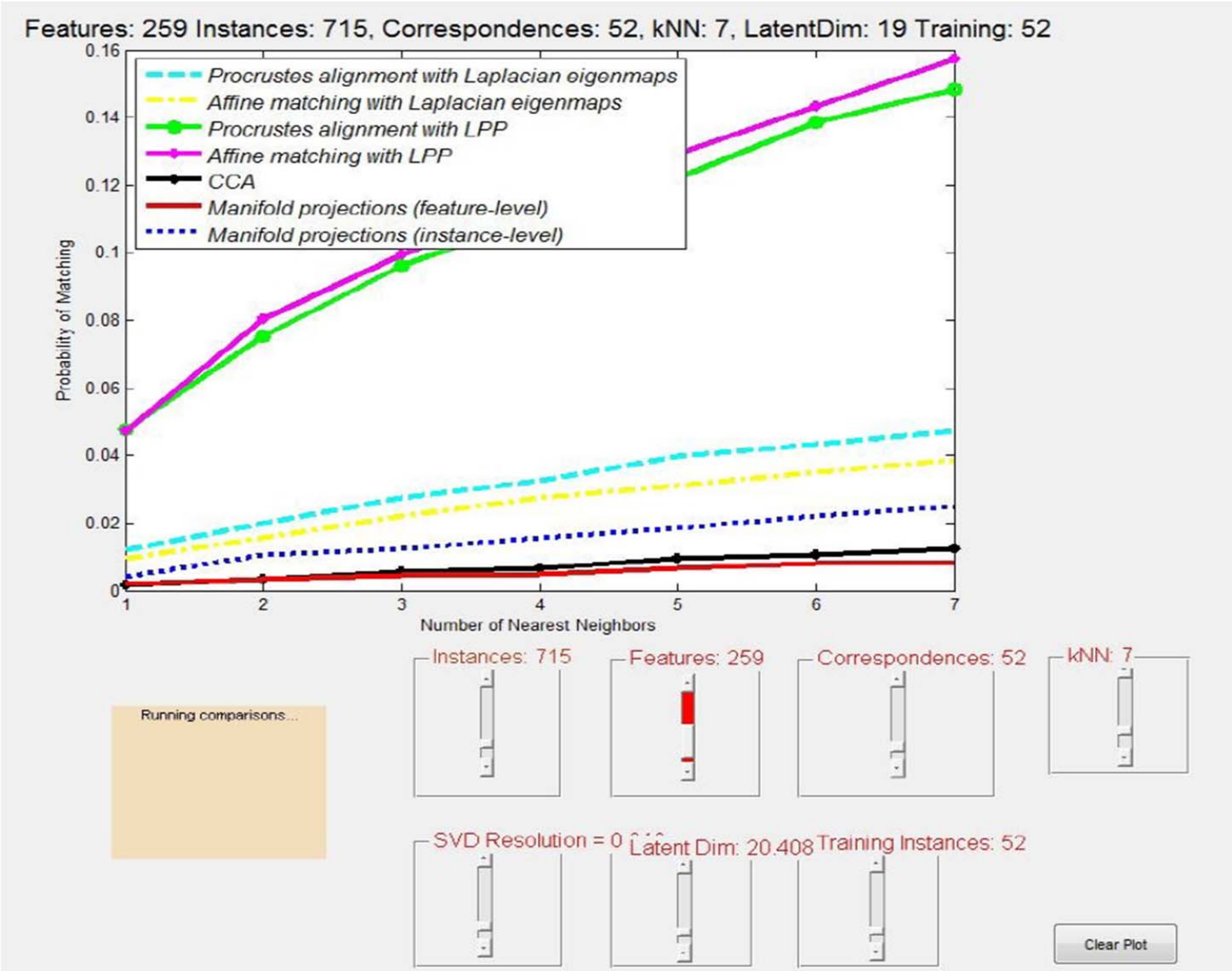
Performance



Performance



Integration with Manifold Alignment Framework



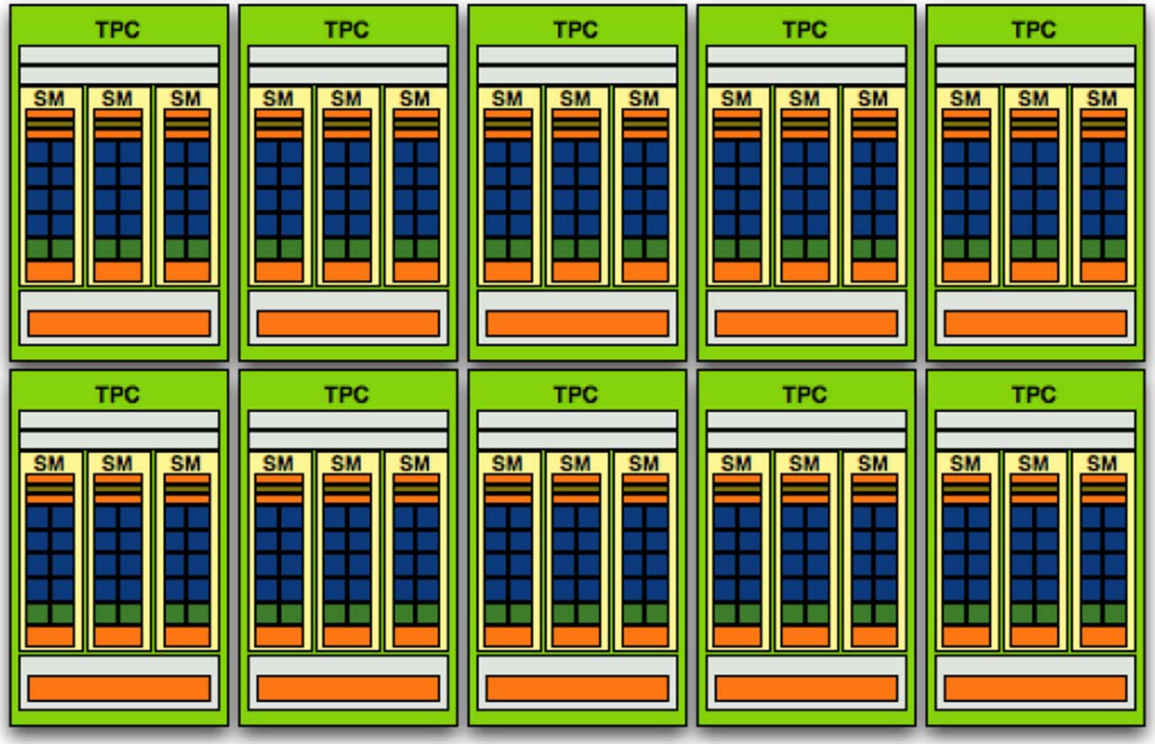
Conclusion and Future Work

- Our initial effort has shown that a GPU-based implementation of QUIC-SVD can achieve reasonable speedup.
- With additional optimizations, 10X speedup foreseeable.

- Handle sparse matrices.
- Components from this project can become fundamental building blocks for other algorithms: random projection trees, diffusion wavelets etc.
- Design new algorithms that are suitable for data-parallel computation.

Programming on the GPU

- **General-Purpose GPU Programming Language**
  - CUDA, OpenCL, DirectCompute, BrookGPU …



[NVIDIA]