Accelerating the Higher Order Singular Value Decomposition Algorithm for Big Data with GPUs

Yuhsiang M. Tsai, Jeewhan Choi (advisor), Xing Liu (advisor), Weichung Wang (advisor)

Institute of Applied Mathematical Sciences, National Taiwan University / IBM Research

Motivation

With the explosion of big data, finding ways of compressing and analyzing large data sets with multi-way relationship - i.e. tensors - quickly and efficiently have become critical in HPC. While Higher-Order Singular Value Decomposition (HOSVD) and Sequential-Truncated Higher-Order Singular Value Decomposition (STHOSVD) provide us with the means to attain both extremely high compression ratio and low error rate though low-rank approximation, optimizing them on accelerators with limited memory is difficult. We share our experience and findings on optimizing these algorithms on a node with multiple GPUs, and demonstrate up to 163.21× speedup over a CUDA library-based solution.

Background

If we wish to find a low-rank approximation to compress and/or extract significant features from a large data set with multi-way relationship, Higher-Order Singular Value decomposition allows us to represent the original data with a *core* tensor, G, and a set of factor matrices along each mode $\{A_i\}_{i=1}^N$.

Figure 1: $X \approx G \times_{i=1}^{N} A_i$

For example, if we want to store a tensor with dimension $1024 \times 1024 \times 1024$, we need $1024^3 \times 8$ bytes which is approximate 8 GBs of memory. If we apply HOSVD on this data with rank of $256, 256, 256$, we can reduce its size to a mere ≈ 0.13 GB, while retaining most of its properties.

For a general tensor X with dimensions $d_1 \times d_2 \times \cdots \times d_n$, we can calculate a low-rank approximation using the following algorithm:

Algorithm 1 Higher Order Singular Value Decomposition (HOSVD)
Require: $X, \{r_i\}$
Ensure: G, A_i
  for $i = 1$ to n **do**
    $A_i \leftarrow r_i$ leading left singular vectors of $X_{(i)}$ ▷ $X_{(i)}$ is mode-i of X
  end for
  $G = X \times_1 A_1^T \times_2 A_2^T \times_3 \ldots \times_n A_n^T$
  return G, A_i ▷ tensor G, matrices $A_1, ..., A_N$

CuSolver SVD

One of the key bottlenecks is calculating the r_n left singular vectors of the matricized tensor. For dense tensors with large mode *lengths* or large *number* of modes, these are extremely large matrices that will not fit on the GPU memory. Unfortunately, SVD library provided by NVIDIA's cuSolver library requires that the entire data set be in memory before it can be factorized, limiting the range of tensors that can be decomposed, and a more scalable solution is required.

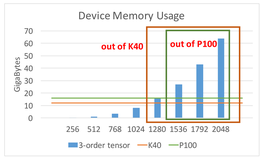
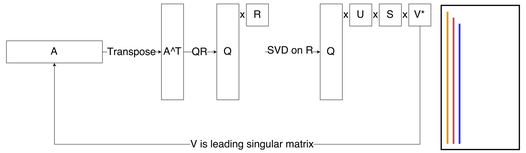


Figure 2: CUDA SVD is failed when tensor is large

Image Notice

We use x->r to represent that we transform the original $x \times x \times x$ tensor into $r \times r \times r$ core tensor. The entries of the tensor are produced randomly.
Testing environment:
CPU: full 24 threads of 2x Intel(R) Xeon(R) CPU E5-2670 v3
GPU: 1x or 2x NVIDIA Tesla K40c

QR method



(a) QR-based method (b) HouseholderQR

One first solution is to use QR decomposition to reduce the matrix size before finding its singular vectors. We implemented the commonly used Householder QR method which allows us to stream the data vector-by-vector, thereby requiring very little device memory.
Limitations:
1) Transferring the data vector-by-vector reduces bandwidth utilization, and
2) The calculation is composed mostly of BLAS-2 operations, leading to reduced compute utilization.

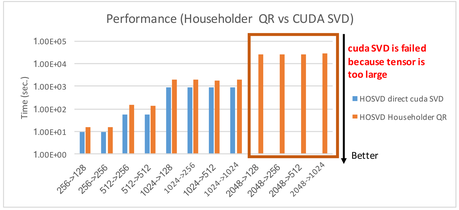


Figure 3: Householder QR method can solve large tensor

Householder QR is good at calculating HOSVD for large tensors, but is slower than cuSolver SVD due to lower bandwidth and compute utilization.

Modified Block QR and Tall Skinny QR

We explored using modified Block QR and Tall Skinny QR (TSQR) to overcome the throughput issues. By using extra device memory, modified QR combines several Householder factors into two matrices to increase the memory throughput. However, when the matrix is tall, the corresponding block used in modified Block QR is too skinny to fully utilize the compute units, and reduces the overall performance (much like the Household QR case).
TSQR is more suitable for this problem than Block QR because it splits the rows to avoid the above situation. It divides the matrix into several square blocks in a column. To solve the QR problem, we solve the QR factorization for two adjacent blocks independently, and combine them. Inductively, it forms the whole process. They use BLAS-3 operations and interleaves matrix transfer to improve performance. Modified Block QR and TSQR significantly improves the performance of the HOSVD algorithm.

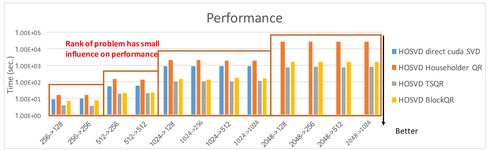
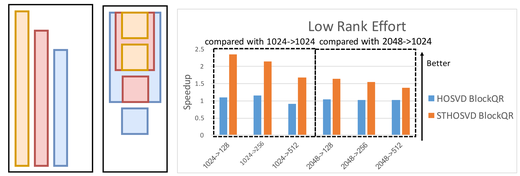


Figure 4: Performance of HOSVD with QR methods

Limitation:
Modified Block QR and TSQR improves performance over the Householder QR and cuSolver SVD methods. However, it still does not fully utilize the compute capability of the GPUs. Therefore, we introduce the STHOSVD method that can further improve the compute unit utilization, and reduce the overall work when the rank is smaller by calculating the Tensor-Times Matrix (TTM) step within the inner loop.

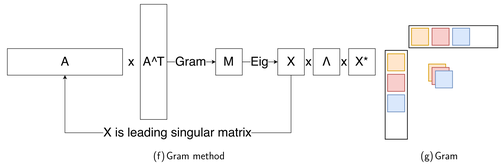
Algorithm 2 Sequential Truncated Higher Order Singular Value Decomposition (STHOSVD)
Require: $X, \{r_i\}$
Ensure: $G, \{A_i\}$
  G = X
  for $i = 1$ to n **do**
    $A_i \leftarrow r_i$ leading left singular vectors of $G_{(i)}$ ▷ $G_{(i)}$ is mode-i of G
    $G \leftarrow G \times_i A_i^T$
  end for
  return G, A_i ▷ tensor G, matrices $A_1, ..., A_N$



(c) BlockQR (d) TSQR (e) STHOSVD vs HOSVD

Gram method

Besides QR methods we also use Gram method followed by eigenvalue decomposition to solve the problem. In this process, we multiply the matricized tensor by its transpose and regard the initial SVD problem as an eigenvalue problem of Gram matrix with smaller problem size.



(f) Gram method (g) Gram

The block multiplications used in Gram method are independent, so we are able to interleave the matrix transfer with computation to increase efficiency, and then combine it with the STHOSVD method to increase performance.

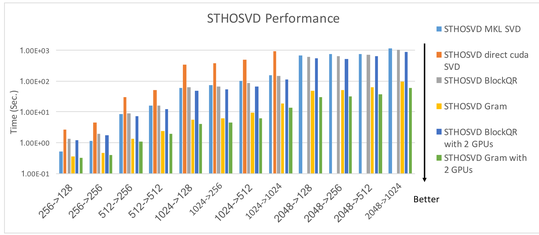


Figure 5: STHOSVD: Gram method is the fastest algorithm

Accuracy

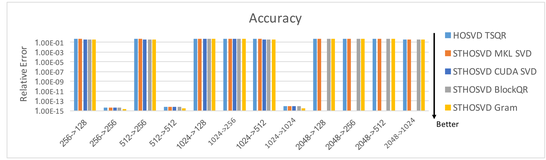


Figure 6: Gram and QR methods have similar accuracy

Multiple GPUs

- Householder QR and Block QR:
  We implement Householder QR and Block QR block-wise. The information also updates block-wise. These methods update simultaneously on multiple GPUs. We make one GPU update fewer blocks than the other GPUs and solve QR factorization of the next block when other GPUs still update the remaining blocks. With this schedule, the next round updating step does not need to wait for the QR factorization.
- TSQR:
  We split a matrix into several tall and skinny blocks by the number of GPUs. We use TSQR to solve each block, so it is an independent process. And then we combine the results together in a single matrix and apply TSQR method on this matrix.
- Gram method:
  We split the matrix into several small blocks. For each block, we only need to calculate the product of its transpose and itself. The operations in each block are independent, so we can assign those works to multiple GPUs equally.

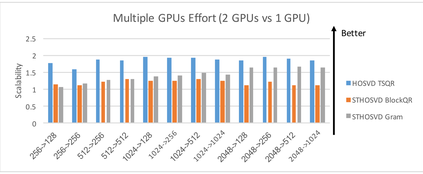


Figure 7: Scalability of multiple GPUs

Conclusion

We study and optimize four methods - Householder QR, Modified Block QR, TSQR, and Gram method - to solve the SVD step in HOSVD and STHOSVD. Among them, Gram method is the fastest algorithm and the Simplest to implement, and provides comparable accuracy in less time. QR methods requires the same amount of storage as that of the original tensor, and spend additional time to form the matrix explicitly. However, in the case that the condition number of the original tensor is large, it may provide higher accuracy.
For the overall performance, view for `https://goo.gl/QsovD1` and the QR code.

Figure 8: All Performance