Efficient Off-Diagonal Low-Rank Matrix-Vector Multiplication

Jonathan Wang
youchun2@illinois.edu

1 Introduction

Hierarchically Off-Diagonal Low-Rank (HODLR) matrices [5] are dense matrices structured in a recursive 2×2 block partitioning with the off-diagonal blocks being low-rank. In each level, the matrix is split into four blocks, with the two off-diagonal blocks have rank-k, and the two diagonal blocks are divided recursively in the same method. In particular, matrix-vector multiplication (MVM) of HODLR matrices can be performed in $O(nk \log n)$ time [4]. One can accelerate linear algebra operations with reduced runtime and memory usage by exploiting its structure.

2 Algorithm and Implementation

Recursive HODLR MVM Algorithm

For the HODLR matrix of size n and rank-k, I create a structure storing each block partitioning as a component. For off-diagonal blocks, the matrix is stored as UV^T with U $(n/2 \times k)$ and V $(n/2 \times k)$. The recursive algorithm is implemented as follows: Let A be an $n \times n$ HODLR matrix represented at the top level as $A = (A_{11}A_{12}A_{21}A_{22})$, where A_{11} and A_{22} are the dense diagonal blocks, and A_{12}, A_{21} are the off-diagonal blocks.

Given an input vector x of length n, the input is first split into $x = \begin{pmatrix} x_1 & x_2 \end{pmatrix}$. To compute $y = Ax$, the algorithm recursively compute $y_1' = A_{11}x_1$, $y_2' = A_{22}x_2$, and then add the results from the off-diagonal blocks $y_1 = y_1' + A_{12}x_2$ and $y_2 = y_2' + A_{21}x_1$ [1–3]. For the off-diagonals, the computation is done through $U(V^T x)$, with an intermediate $\alpha = V^T x$. The same process repeats in each level until the matrix reaches the maximum specified level (*max_level*). At the base level, where the diagonal blocks are stored as dense matrices, the algorithm simply performs a standard dense MVM. The result y is obtained when all the sub-operations are computed and added together.

C++/CUDA Implementation

The algorithm above is implemented in C++ with CUDA. The matrix components are defined as a struct, which is stored in a vector. To generate such matrices, I simply generate random dense matrices and compress them into the HODLR structure. The user gets to specify the rank-k and the maximum level of recursion (*max_level*).

With the HODLR structure, GPU memory was first allocated. Then, for all dense diagonal blocks, a single batched cuBLAS call (cublasSgemvBatched) is made to perform all the MVMs of the same size together. The reason it works is that each multiplication doesn't interfere with the others, as the output indices don't overlap. The same action is performed on off-diagonal blocks of the same size, but in a two-step fashion. The first cuBLAS call is used to compute $\alpha = V^T x$, with the latter for $U\alpha$. The design focuses on the parallel execution of independent operations and minimizes data movement if possible. After the computation is completed, the result is copied back to the host. To verify the correctness of the computation, the result is compared to a direct CPU MVM of the original dense matrix (for testing purposes).

3 Optimization Techniques

The following list the optimizations performed to achieve high performance of HODLR MVM:

Use of cuBLAS library: For all components that involve dense MVM, NVIDIA's cuBLAS library is used to achieve high performance. This also allows cuBLAS to handle some of the optimizations, such as memory coalescing and concurrency. The library provides a significant speedup by efficiently leveraging GPU acceleration.

Batched Matrix-Vector Multiplications: One of the most significant optimizations is to batch together as many independent MVMs of the same size and execute them within a single kernel launch. Initially, one might loop over hundreds of matrix blocks and call MVM for each of them. Instead, cublasSgemvBatched was used to process multiple multiplications in one kernel call. This also allows all dense blocks to be processed by one GEMV call and dramatically reduces overhead. Without it, the implementation will be latency-bound by all the small kernel launches.

Efficient Memory Management and Computation: The memory transfers and allocations are also carefully handled. By transferring the required parts and the U, V matrices of the off-diagonal blocks, the memory copy was minimized as much as possible. In addition, each off-diagonal block multiplication is broken into two steps, and the intermediate result is used immediately after being computed, removing the need to store the result back to the host.

4 Evaluation

Experiment Setup

The results were obtained from the machine in the SciComp lab (porter.cs.illinois.edu) equipped with an NVIDIA TITAN V GPU. The matrices were generated with size ranging from $n = 2^7$ up to $n = 2^{15}$, with a combination of rank-k from 6 to 18 and max_level from 2 to 12. One of the experiments included 5 runs of a warm-up run for the kernel, labeled *warmup*, the other didn't. For all tests, we recorded the average from 8 runs to ensure the accuracy of the results.

We compare three methods: (1) a single-threaded CPU dense MVM (for baseline and accuracy check), (2) a GPU dense MVM using cuBLAS, and (3) the GPU HODLR MVM implementation. For the two GPU methods, both the pure kernel execution time and the total time including data transfer are recorded. Timing on the GPU was done with CUDA events for accuracy. We also measured the operation counts and computed the throughput in GFLOPs/s. The relative error of the results was compared to the CPU dense MVM to ensure correctness.

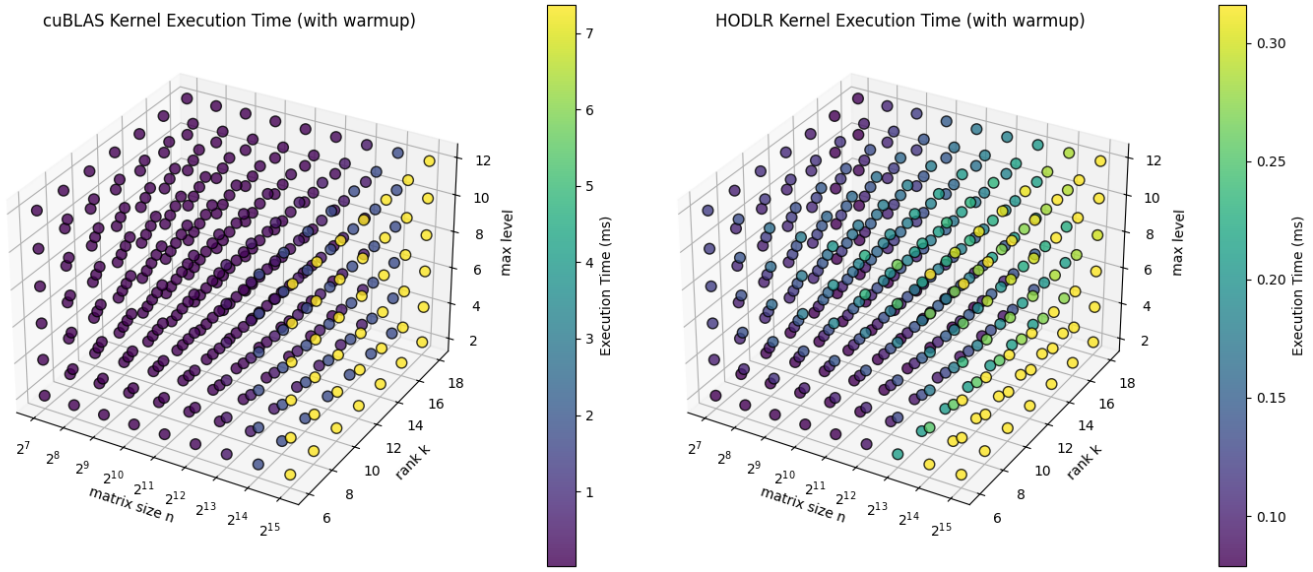

Figure 1: kernel execution time (with warm-up) for n, k, and max level

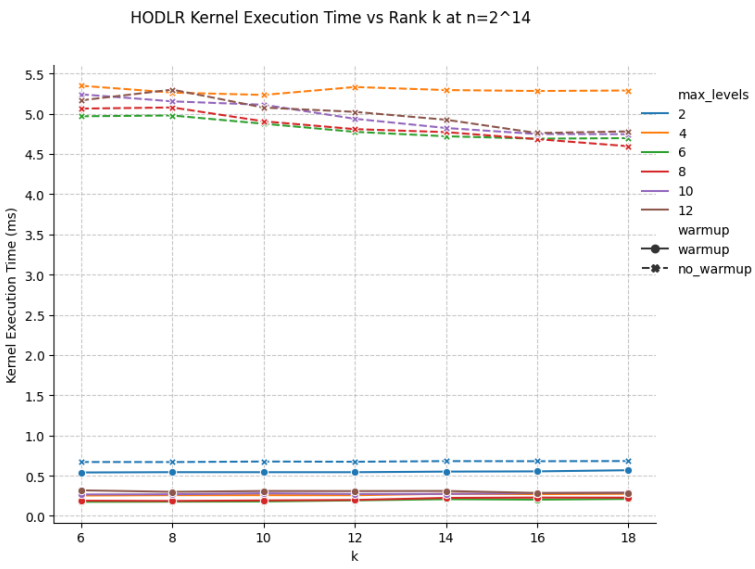

Figure 2

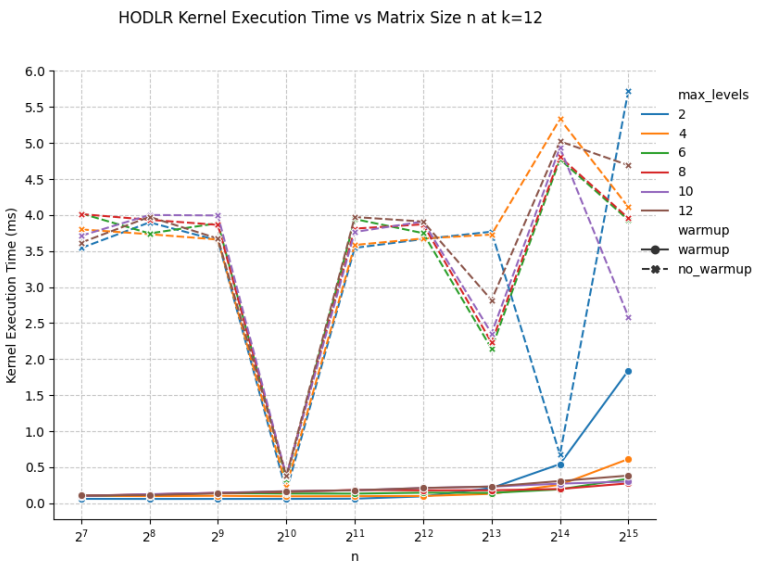

Figure 3

Results

Figure 1 shows the kernel execution time of the dense cuBLAS MVM and the HODLR MVM, as a result of matrix size n, rank-k, and max_level. We can observe that in an identical configuration, the execution time of the HODLR algorithm outperformed dense cuBLAS by roughly an order of magnitude. The execution time also spikes when the matrix size approaches 2^{15}.

Next, to better visualize the relationship of independent variables with respect to the kernel execution time of HODLR MVM, we fix the matrix size n at 2^{14}, rank-k at 12, and max_level at 8 for Figure 2 3 4, respectively.

In Figure 2, we observe that the rank-k of the matrix has little effect on the execution time of the kernel when the matrix size is fixed, possibly because $k << n$ for all these k and how optimize cuBLAS library is optimized for small dense MVM. The performance peaked when the max level is set to 6, although the difference is not that significant. This suggests the overhead from many tiny blocks overtook the benefit of reducing flop count per block. In Figure 3, the plot shows that for small matrix sizes, the execution times are

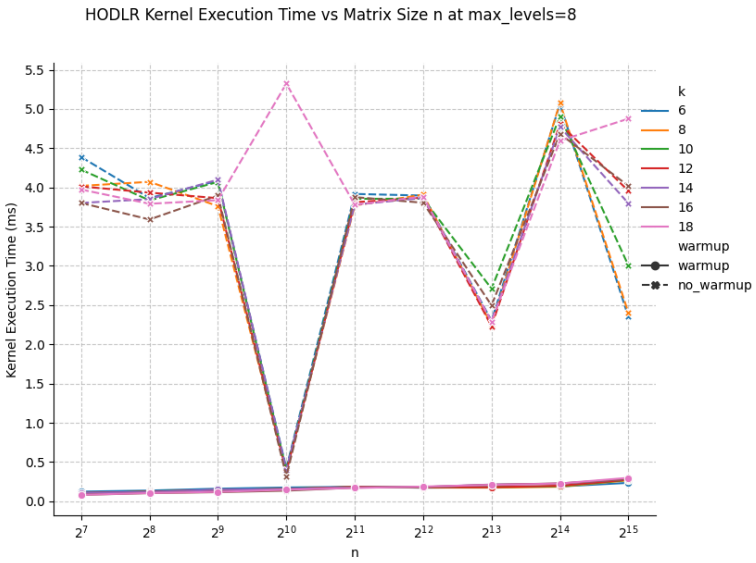

Figure 4

similar and have little relation to the choices of max level. However, for $n = 2^{14}$ and 2^{15}, the execution time for max level = 2 took off, likely due to having to process a large MVM, and the cache was unable to store all the results at once. It is also interesting to see the time for no warm-up when $n = 2^{10}$ being an outlier. The reason could possibly be due to either cache alignment in vector size or the randomness from not performing a warm-up for cuBLAS. Similarly, the outlier at $n = 2^{10}$ can also be seen at Figure 4.

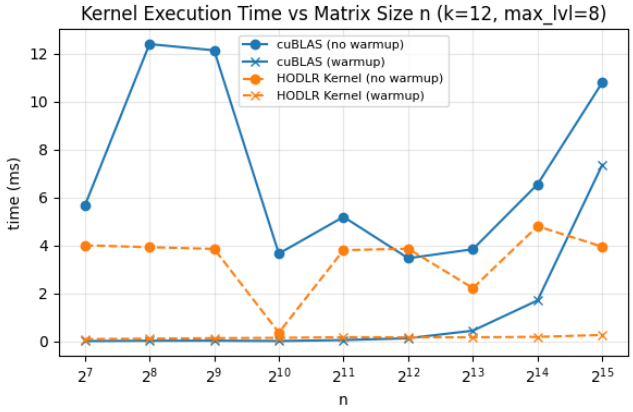

Figure 5

To see the one-on-one relation between matrix size n with respect to kernel execution time, GFLOPs/s, and total execution time for dense cuBLAS MVM and HODLR MVM, we plot Figure 5 6 7, fixing the parameters with rank-k at 12, and max level at 8.

From Figure 5, we can observe the uncertainty involved in execution time when there are no warm-up runs. These scenarios also performed worse than those with warm-ups. The kernel execution

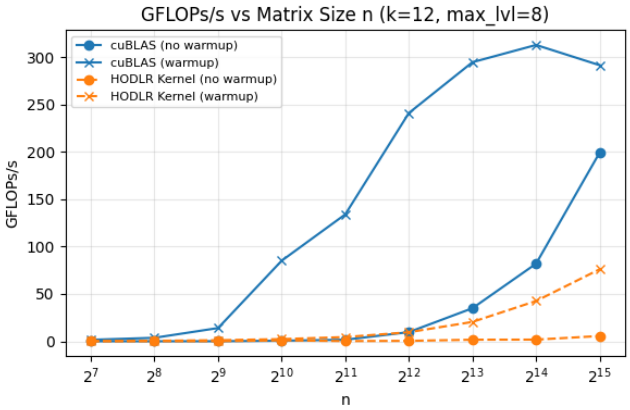

Figure 6

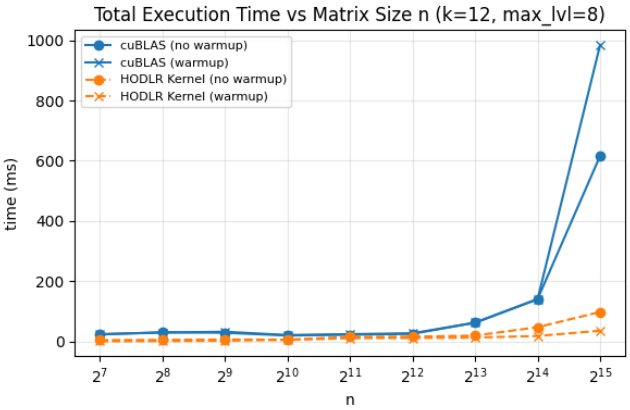

Figure 7

time for HODLR MVM with warmup was stable and maintained low throughout the x-axis. On the other hand, when memory transfer is taken into account, the results become clear. In Figure 7, the total execution time has a positive relationship with large matrix sizes. The differences between dense cuBLAS MVM and HODLR MVM total time can reach beyond one order of magnitude when $n = 15$.

However, in Figure 6, we see that the GFLOPs/s is lower for the HODLR implementation. This is due to the total amount of operations saved from storing the off-rank matrix in matrices U, V. One might also observe that the GFLOPs/s for both methods peak at around 10 and 200, respectively, far below the theoretical value for the TITAN V GPU. The reason is that the operation is memory-bound, becoming bottlenecked by PCIe transfer. For example, sending a 16k×16k matrix (approx. 1GB) can take hundreds of milliseconds alone, which is on par with the compute time. Using Nsight Compute Profiling Tool, we see that the memory throughput reached over 98%, while the compute throughput peaked at 46% when performing batch MVM for the dense leaf blocks.

Lastly, to ensure the consistency of the HODLR MVM implementation, we recorded the maximum total execution time that occurred at each matrix size n, and compared the results to the

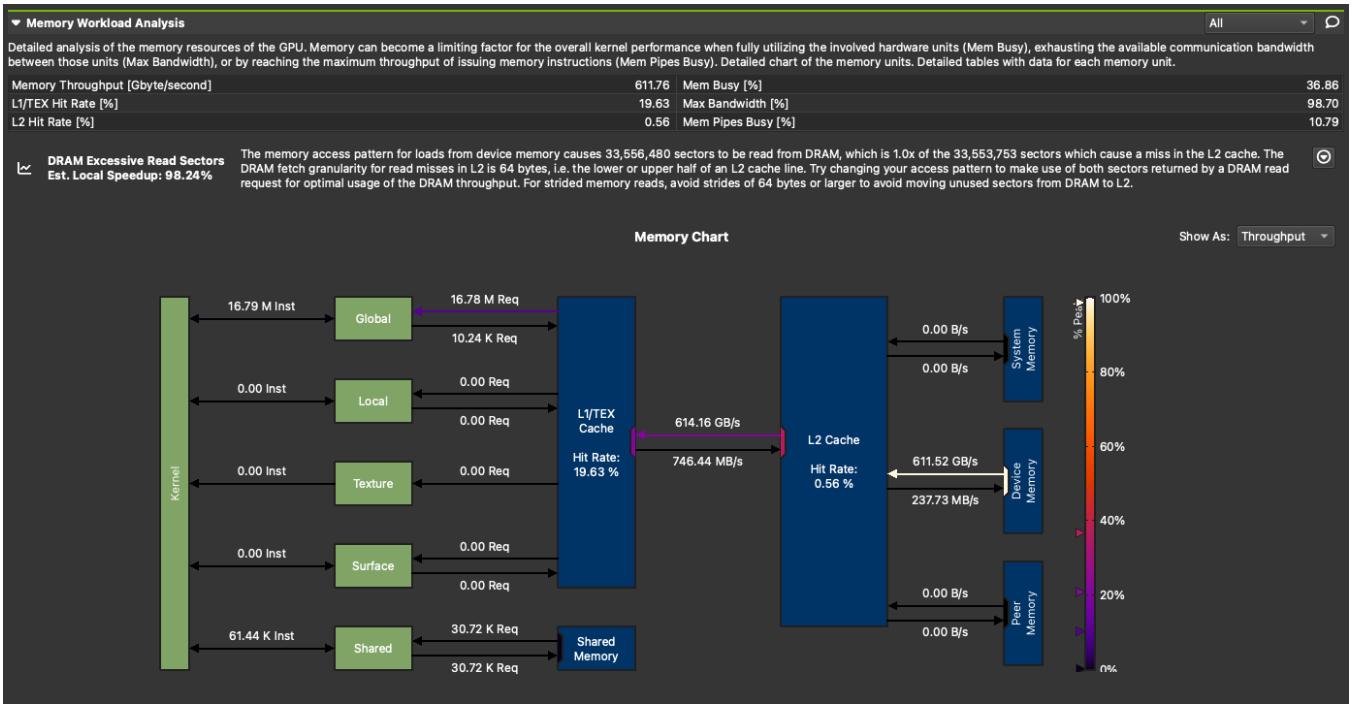

Figure 8

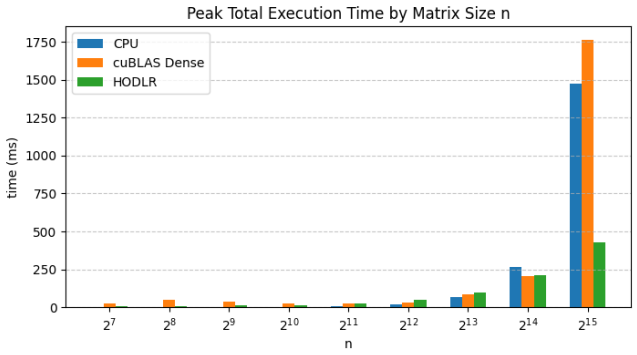

Figure 9

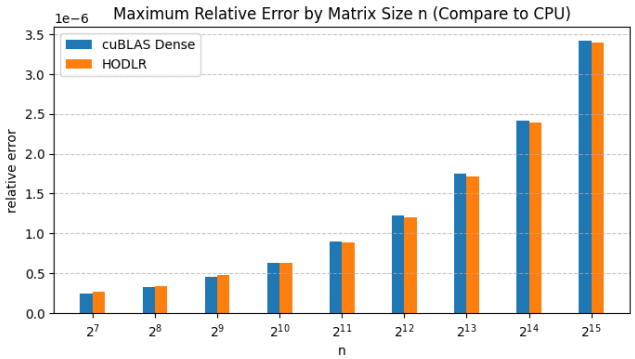

Figure 10

CPU and dense cuBLAS methods. As Figure 9 showed, the gap grew bigger as the matrix size increased. Figure 10 shows the correctness of the GPU methods by comparing the relative error with the single-thread CPU version. The error in the magnitude of 10^{-6} concluded the claim.

5 Conclusion

Overall, through the exploitation of HODLR structure on GPUs, the GPU-accelerated HODLR matrix-vector multiplication achieved significant performance gains over CPU and dense cuBLAS MVM implementations, with execution time speed up at around one order of magnitude. Thanks to the optimization techniques implemented (batching, memory management, etc.), the gap further widens for large matrix sizes. We also note that the current implementation still performs below the theoretical peak of the hardware, as the kernel is both memory-bound and launch-bound, but the reduction in complexity makes up for it. Future work can try to address this by fusing kernels, using multiple streams, or other techniques. Nonetheless, we yield a solution that is both fast and scalable, providing a stepping stone for fast solvers for linear system problems involving HODLR matrices.

References

[1] Michele Benzi, Daniele Bini, Jonas Ballani, and Daniel Kressner. 2014. *Matrices with Hierarchical Low-Rank Structures.* Technical Report SMA/2014/016. École Polytechnique Fédérale de Lausanne. https://sma.epfl.ch/~anchpcommon/publications/cime.pdf

[2] Chao Chen and Per-Gunnar Martinsson. 2022. Solving Linear Systems on a GPU with Hierarchically Off-Diagonal Low-Rank Approximations. *arXiv preprint arXiv:2208.06290* (2022). https://doi.org/10.48550/arXiv.2208.06290 arXiv:2208.06290 [cs.NA]

[3] Greg O'Neil. 2020. *Fast Algorithms (Chapter 5).* Lecture Notes. Courant Institute of Mathematical Sciences, New York University. https://cims.nyu.edu/~oneil/courses/fa20-math2011/Fast_Algorithms_Chapter_5.pdf

[4] user17762. 2013. Special matrices for which the cost of matrix–vector multiplication is less than $O(n^2)$. Mathematics Stack Exchange, https://math.stackexchange.com/q/281723. Accessed: 2025-05-02.

[5] Wikipedia contributors. 2025. Hierarchical matrix. https://en.wikipedia.org/wiki/Hierarchical_matrix. Accessed: 2025-05-02.