GPU Accelerated Hierarchical Semiseparable (HSS) Matrix Multiplication in Julia

Axel Feldmann

1 Introduction

Operations on large dense matrices are a performance bottleneck in many scientific computing applications. Dense matrices take up $O(n^2)$ space in memory and most computations on them, like multiplication and factorization, are $O(n^2)$ or $O(n^3)$.

Hierarchical semiseparable (HSS) matrices [1] are a *compressed* and *approximate* representation based on the observation that many matrices used in numeric algorithms are dominated by values *near the diagonal*. An HSS matrix H compresses a dense matrix A by storing it as a tree of low-rank approximations for off-diagonal blocks and small full matrices for on-diagonal blocks. For matrices amenable to this form of compression, HSS matrices require only $O(n)$ memory and provide $O(n)$ multiplication and solution. As a result, HSS matrices can be very useful in solving many discretized partial differential equations.

However, despite their asymptotic efficiency, computations on large HSS matrices can still be time-consuming. To address this problem, we turn to hardware-acceleration using GPUs. HSS algorithms are not trivially parallelizable on GPUs: they involve traversing H's tree structure and performing many *small* matrix multiplications, while GPUs are most efficient on *large* parallel kernels.

In this work, we present an implementation of GPU-accelerated HSS matrix-dense matrix multiplication in Julia. Our implementation achieves an up-to 80× speedup over the existing CPU-only implementation in `HssMatrices.jl`. As part of this implementation, we also developed a standalone *batched* GPU matrix multiplication library in Julia that can be used outside the scope of HSS matrices.

2 Background

In this work, we will not provide a rigorous explanation explanation HSS matrix multiplication. Instead, we will focus on providing some intuition and introducing the key components. To see a complete explanation, see [5]. For concreteness, this work will focus on the same example as `HssMatrices.jl`:

$$K(x,y) = \begin{cases} \frac{1}{x-y} & \text{if } x - y \neq 0 \\ 1 & \text{if } x - y = 0 \end{cases} \qquad (1)$$
$$-1 \leq x \leq 1, \quad -1 \leq y \leq 1$$

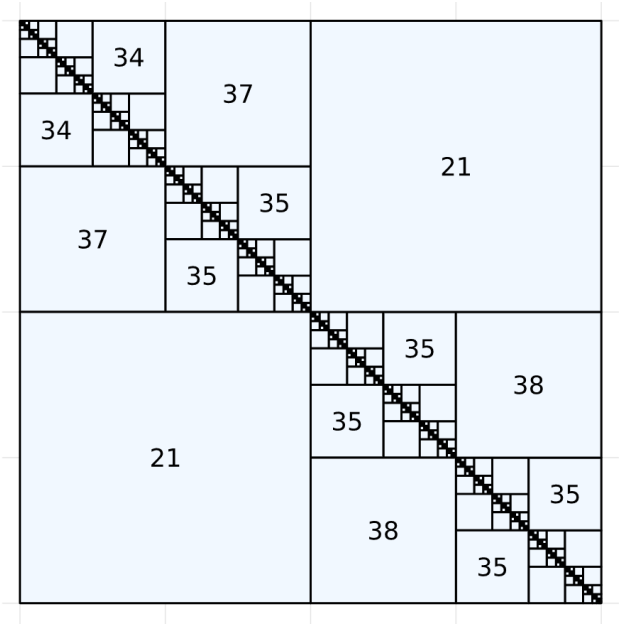

Figure 1: Structure of a HSS matrix representing a $20,000 \times 20,000$ instance of K. The numbers in each block represent the rank of its approximation.

We can adjust the granularity of discretized x and y values to obtain matrices of different sizes.

Figure 1 shows the structure of K when using a granularity of 0.0001, resulting in a $20,000 \times 20,000$ matrix. Blocks closer to the diagonal are represented by higher-rank approximations. Note that the bottom left block, which is of size $10,000 \times 10,000$ is represented by a rank-21 approximation requiring just 0.21% of the memory required to store it densely! Despite this compression, the relative error from multiplying $HSS(K)$ by a random vector b is on the order of 10^{-7}.

2.1 HSS Structure

To compress matrices, we use `HssMatrices.jl`. Running `hss(A)` on a `Matrix{T}` results in an `HssMatrix{T}`. Fundamentally, `HssMatrix` is a tree structure– an `HssMatrix` has 4 main fields: $A_{11}, B_{12}, B_{21}, A_{22}$ each corresponding to a different region of the matrix it represents. A_{11} and A_{22} are themselves `HssMatrix` structs while B_{12} and B_{21} are `Matrix` objects storing low-rank approximations.

An example tree structure is shown in Figure 2. Because only A_{11} and A_{22} are recursive $HssMatrix$ objects, the tree is a full *binary* tree spanning the ma-

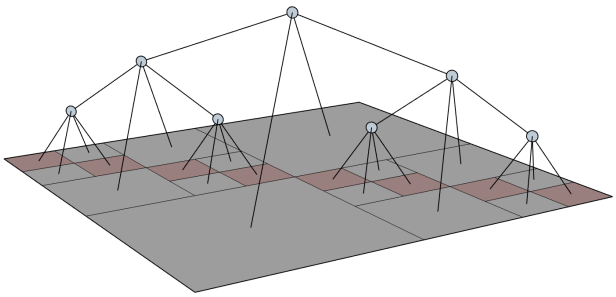

Figure 2: HSS tree structure. Pink tiles are stored full-rank, gray tiles are stored as low-rank approximations.

trix's diagonal. Leaf nodes contain dense blocks of the original matrix A. Each non-leaf node also points to two off-diagonal blocks. As these are ordinary `Matrix` objects, they do not have children.

To recover approximations of A_{12} and A_{21} from B_{12} and B_{21}, each non-leaf node stores a set of 4 small matrices R_1, R_2, W_1, W_2 such that

$$A_{12} = R_1 B_{21} W_1$$
$$A_{21} = R_2 B_{12} W_2$$

In addition to their dense block D, leaf nodes also store small dense matrices U and V that are needed during multiplication.

2.2 Multiplication

Suppose we want to multiply an HSS matrix H and a dense matrix B. Multiplication happens in two passes, a *down pass* and an *up pass*.

Up pass: During the up pass, we propagate values from the leaves up to the root as follows:

```
up(N, B) =
  if isleaf(N)
    return N.V' * B
  else
    n1 = size(N.A11, 2)
    return N.W1' * up(N.A11, B[1:n1,:])
    + N.W2' * up(N.A22, B[n1+1:end,:])
  end
```

We run $up(H, B)$, and cache the result of $up(N, B)$ at each node N.

This computation has a two aspects worth nothing: first, B is recursively split into chunks split across the HSS matrix's leaf nodes. Second, this algorithm is composed of multiplying and adding many small matrices together. After executing this algorithm starting with $N = H$, each node N will store a matrix $N.up$

Down pass: The up pass sends values from the root back down to the leaves. These quantities involve computations with the $N.up$ matrices computed during the up-pass. The up-pass is defined as follows:

```
down(N, B, F) =
  if isleaf(N)
    return N.D * B + N.U * F
  else
    n1 = size(N.A11, 2)
    F1 = N.B12 * N.A22.up + N.R1 * F
    C1 = down(N.A11, B[1:n1,:], F1)
    F2 = N.B21 * N.A11.up + N.R2 * F
    C2 = down(N.A11, B[n1+1:end,:], F2)
    return vcat(C1, C2)
  end
```

We can run $C = down(H, B, nothing)$ starting at the root of H. C will be the approximate result of AB. Again, B and C are recursively split into chunks split across the HSS matrix leaf nodes, and again, the algorithm consists of many small matrix multiplications.

One important observation is that HSS matrix-dense matrix multiplication implements a prefix-sum algorithm. The up-pass and down-pass correspond exactly to the up-pass and down-pass in prefix sum algorithms!

3 Implementation

When implementing this algorithm on a GPU, we must consider the strengths and weaknesses of GPUs as a platform. Both the up and down passes of HSS matrix multiplication involve a lot of multiplications with small matrices. Running each of these multiplications in a separate kernel would result in losing all performance to kernel launch overheads. Furthermore, because the matrices are mostly very small (< 100 rows and columns), a single matrix multiplication will be unable to saturate the GPU's compute resources.

Therefore, we need to *batch* multiple multiplications into the same kernel launch. These multiplications must be *independent*, meaning that we can execute them in parallel without any data races. Fortunately, the tree structure provides us with ample opportunities for inter-node parallelism. Consider the up-pass dataflow on a small HSS tree shown in Figure 3. We are able to multiply all the leaf $N.V$ matrices by their respective chunks of B in parallel. Looking up the tree, we are also able to execute `N5.W1 * N1.up` and `N6.W1 * N3.up` in parallel. We cannot, however, execute `N5.W2 * N2.up` in parallel with `N5.W1 * N1.up` as they both write to the same destination: `N5.up`. Note matrices on the same level of the tree are *not* guaranteed to be the same size.

3.1 Batched Matrix Multiplication

CuBLAS does support batched matrix multiplication with the `cublas<T>gemmBatched()` function, however it requires that all the matrices be the same size. Furthermore, Julia wrappers of this function in `CUDA.jl` appear to be somewhat flaky. As a result, we must implement our own batched matrix multiply from scratch.

Multiplying many matrices of different sizes on a GPU requires some non-trivial engineering. GPU kernels are organized in *grids* of *thread blocks*. Grids, as

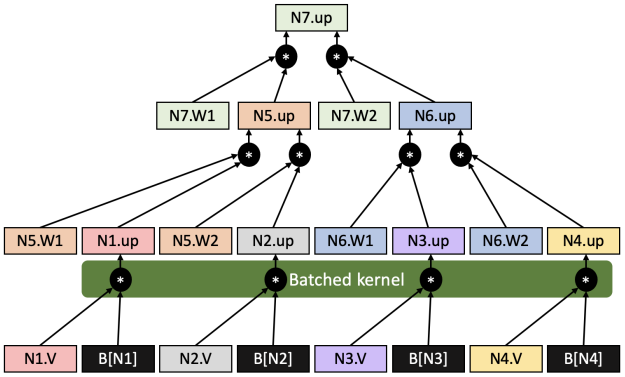

Figure 3: HSS matrix multiplication up-pass dataflow.

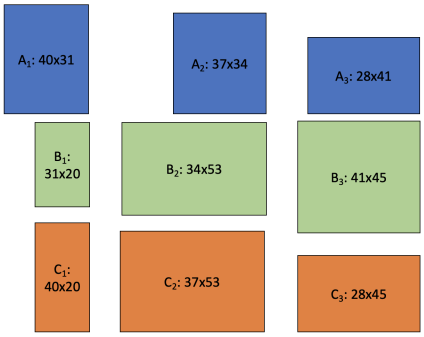

Figure 4: Example matrices for a batched matrix-matrix multiplication. For $i = 1, 2, 3$, $C_i = A_i B_i$

well as thread blocks, must have uniform dimensions. Consider the example shown in Figure 4. Assuming that we assign a single thread to each location in the output matrix C, $C_1 = A_1 B_1$ will require 620 threads, $C_2 = A_2 B_2$ will require 1961 threads, and $C_3 = A_3 B_3$ will require 1260 threads. To further complicate things, 1961 and 1845 are both greater than the 1260 limit of threads/block imposed by most GPUs! As a result of this imbalance, we *cannot* do a simple mapping of k blocks per pair of matrices.

Concretely: if A matrices had uniform size $n \times k$ and B matrices had uniform size $k \times m$, then each matrix multiplication would require $n \times m$ threads. Multiplying two groups of N such matrices could be accomplished with a grid size of $N \times \lceil \frac{n \times m}{1024} \rceil$. Each thread would be able to use `blockIdx.x` to determine the pair of matrices it is multiplying and use `blockIdx.y` to determine its position in the output matrix.

Given non-uniformly sized matrices, this approach is not possible– threads in a block cannot simply read their `blockIdx` to know their position in the multiplication.

Instead, we must implement two new data structures on the GPU. The first is called `BatchedMats` which represents a batch of matrices. Example `BatchedMats` representing the matrices in Figure 4 are shown in Figure 5. A storing N matrices will contain 3 arrays of size N, `rows`, `cols`, and `starts` in addition to a

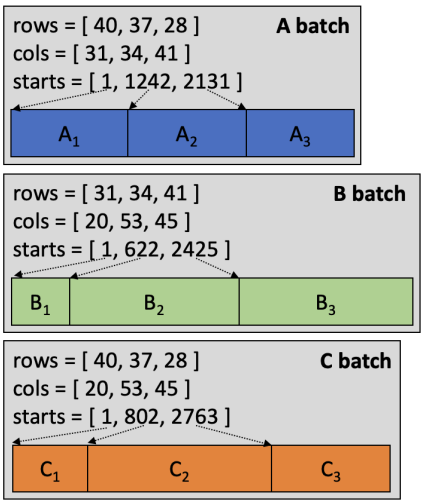

Figure 5: `BatchedMats` structures representing the matrices in Figure 4.

`data` array that holds the actual matrix values. The structure is built such that for all $1 \leq 1 \leq N$, matrix i is of size `rows[i]` $\times$ `cols[i]`. Its values are laid out in column-major order from `data[starts[i]]` to `data[starts[i] + rows[i] * cols[i]]`. This structure is a compact representation of a group of matrices stored in GPU memory.

The second new data structure is called a `Task` and is designed to address the non-uniformity problems discussed earlier. An example `Task` corresponding to the pairwise multiplication of $[A_1, A_2, A_3]$ and $[B_1, B_2, B_3]$ is shown in Figure 6. A task contains 4 arrays: `dest`, `src1`, `src2`, and `chunk`. This is an auxiliary data structure designed to help threads in the matrix multiplication kernel know where to find their inputs and where to store their outputs.

Specifically, each block j in the matrix multiplication kernel `batchedmul!(Cs, As, Bs, task)` is able to find out its work by looking at `task.dest[j]`, `task.src1[j]`, `task.src2[j]`, and `task.block[j]`. These values are used as follows:

1. `task.dest[j]` is used to find a thread's output matrix in `Cs`. If a thread reads `d = task.dest[j]`, then it knows that it is writing to matrix `Cs[d]`, of size `Cs.rows[d]` $\times$ `Cs.cols[d]`, whose data is stored starting at `Cs.data[Cs.starts[d]]`.

2. `task.src1[j]` and `task.src2[j]` are analogous to `task.dest[j]`, except instead of telling the thread where to write its output data, they tell it where to *read* its input data. A thread in a block j will be repsonsible for reading values from `As[task.src1[j]]` and `Bs[task.src2[j]]`. In this basic example, for all j, `dest[j]` is the same as `task.src1[j]` and `task.src2[j]` but this is not always the case. In HSS matrix multiplication, indices often differ.

3. `task.chunk[j]` is used to find which chunk of the destination matrix a particular thread is responsi-

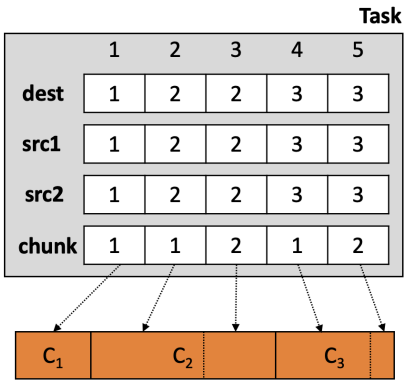

Figure 6: `BatchedMats` structures representing the matrices in Figure 4.

ble for. Consider the example shown in Figure 6: because of the 1024 thread limit, both blocks 2 and 3 need to work on $C_2 = A_2B_2$. To disambiguate which block of the result they are working on, the threads use the `task.chunks` array to calculate their output index.

Using these structures, the kernel is able to locate its inputs and outputs roughly as follows:

```
function kern(...)
    b = blockIdx().x
    t = threadIdx().x

    C_idx = task.dest[b]
    A_idx = task.src1[b]
    B_idx = task.src2[b]
    C_chunk = task.chunk[b]

    C_idx = (C_chunk - 1) * 1024 + t
    ...
end
```

This implementation allows efficient variably-sized batched matrix multiplication and forms the basis of HSS matrix multiplication.

3.2 HSS Multiplication

Using the batched matrix multiplication primitive described in subsection 3.1, we are able to implement an efficient HSS matrix multiplication algorithm. The algorithm is composed of two parts: setup and execution.

Setup: Before beginning the actual HSS matrix multiplication `HSS(A) * B`, data must first be transferred to the GPU and set up appropriately. To leverage our efficient batched matrix multiplication primitives, we must structure `H = HSS(A)` as a set of `BatchedMats`, each representing matrices at all *levels* of the tree. Similarly, we must divide B into a `BatchedMats`, with each of the component matrices mapping to a different leaf of H. This computation is done primarily on the CPU. The expensive part of this computation is structuring H as a set of `BatchedMats`. However, this must only be done once over A's lifetime, and therefore amortized extensively over the course of multiple computations.

As part of setup, storage for intermediates, (such as `up` and `F` from subsection 2.2) must also be allocated. Unfortunately, the size of these intermediates is a function of the number of columns in B, so once intermediates have been allocated for a particular B, their space can only be reused in multiplications with matrices B' that have at most as many columns as the original B. During this stage, we also allocate all GPU `Task` structures.

Execution: Once setup is complete, executing the algorithm is relatively straightforward. We simply iterate through the prepared `Tasks`, performing batched matrix multiplications on the appropriate input and output `BatchedMats` for the given task. Each matrix multiplication takes `4L-1` calls to `batchedmul!`, where L is the number of levels in the HSS tree.

Upon completion, the result can be obtained by unbatching the result C and concatenating its chunks together as follows: `vcat(unpack(C)...)`.

4 Results

All results are obtained on the JuliaHub machines using Nvidia V100 GPUs and 8-core Intel Xeon CPU E5-2686 v4 CPUs.

4.1 Batched Matrix Multiplication

To test the effectiveness of the batched matrix multiplication primitive, we run experiments on pairwise multiplying two N-vectors of $m \times m$ matrices. First, we compare batched kernel performance to CPU performance in Figure 7. We note a few effects. When we have very small values of m, the CPU's BLAS is unable to effectively parallelize the individual matrix multiplications. At large values of m, the CPU is able to parallelize individual matrix multiplications, and therefore our speedups become smaller. Our largest speedups come on problems where the matrices are too small for the CPU to extract intra-multiplication parallelism, but not so small that the GPU is unable to saturate all of its compute resources. It is unclear why we observe slowdowns for large values of N at $m = 64$– perhaps cache effects or benchmarking artifacts.

However, the true baseline for this kernel is not a CPU, but rather an unbatched GPU implementation. Results are shown in Figure 8. Here, we compare a single batched matrix multiplication kernel against sequentially launching `CUBLAS.gemm` kernels with a single synchronization point at the end. Here, we observe that for larger values of m, the highly-optimized CUBLAS kernel begins to significantly outperform our less tuned kernel. However, for smaller values of m, we obtain very large speedups, as our implementation is able to more effectively leverage inter-matrix parallelism. Furthermore, as we have more matrices in the

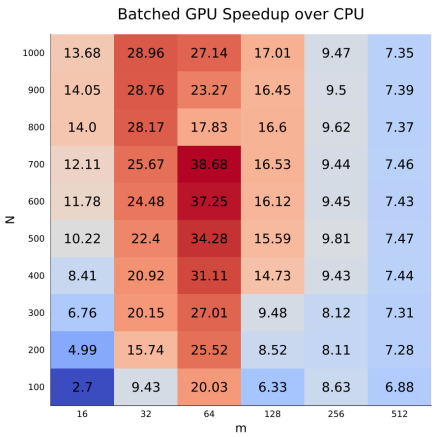

Figure 7: Batched GPU kernel speedups over CPU on pairwise multiplication of two N-vectors of $m \times m$ matrices.

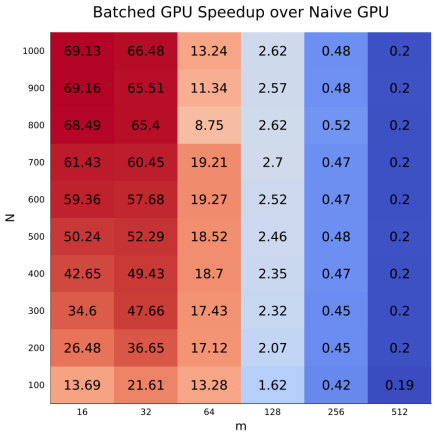

Figure 8: Batched GPU kernel speedups over naive GPU implementation on pairwise multiplication of two N-vectors of $m \times m$ matrices.

batch, the speedup gets larger as the naive GPU implementation incurs more kernel launch overheads.

These results show that the new batched matrix multiplication kernel can obtain large speedups in cases where we have many small, independent matrix multiplications– exactly what is needed for HSS matrices.

4.2 HSS Matrix Multiplication

We evaluate HSS multiplication AB where A is an instance of the K matrix described in section 2 and B is a random dense matrix.

Our experiments sweep different sizes of A from 500 to 35,000 as well as different numbers of columns in B. The CPU baseline is `mul!` from the `HssMatrices.jl` package. Results are shown in Figure 9.

Unsurprisingly, we observe the largest speedups when both A and B are large– this results in large kernel launches which are able to saturate all of the GPU's compute resources. On very small instances of A and B, we do observe slowdowns– even though we batch matrix multiplications, there is simply not enough work

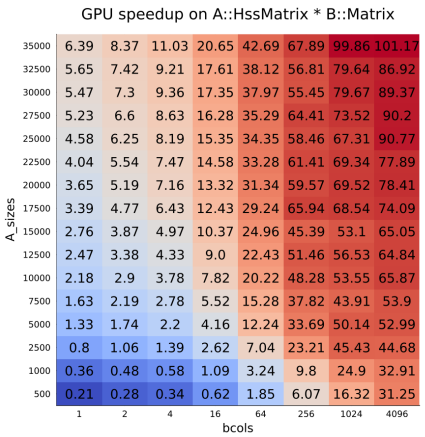

Figure 9: GPU speedups on various HSS matrix multiplication sizes.

to amortize kernel launch overheads. However, for the largest tested values of A and B, we obtain a $> 100\times$ speedup over the baseline.

Importantly, we obtain some speedups even on matrix-vector multiplications (left column of Figure 9). These are particularly challenging because they have very little arithmetic intensity. However, by leveraging batching, we are able to still obtain performance improvements.

4.3 Code

Code is available at `https://github.com/axelfeldmann/hss-gpu`.

5 Future Work

This GPU-accelerated HSS matrix library is by no means complete. The most important improvement is to leverage GPU shared memory to achieve larger speedups for batched matrix multiplications. This should allow batched matrix multiplication kernels to perform closer to unbatched CuBLAS baselines for the larger values of m in Figure 8.

Further work includes accelerating all other HSS matrix primitives. This includes compression (actually creating HSS matrices from dense matrices), factoring, and solution. More performance engineering can also be done to the CPU-based setup phase of HSS matrix multiplication. Finally, it would be interesting to integrate this library into existing applications that use HSS matrices and obtain real end-to-end speedups.

References

[1] Chandrasekaran S., Dewilde P., Gu M., Pals T., van der Veen A.J. (2002). *Fast stable solver for sequentially semi-separable linear systems of equations.* High Performance Computing—HiPC (2002).

[2] Boukram W., Turkiyyah G., and Keyes D. (2019). *Hierarchical Matrix Operations on GPUs: Matrix-*

Vector Multiplication and Compression. ACM Transactions on Mathematical Software, https://dl.acm.org/doi/pdf/10.1145/3232850.

[3] Zhu Z., and Soricut R. (2021). *H-Transformer-1D: Fast One-Dimensional Hierarchical Attention for Sequences*. Arxiv, https://arxiv.org/pdf/2107.11906.pdf.

[4] Vater K., Betcke T. and Dilba B. (2017). *Simple and efficient GPU parallelization of existing H-matrix accelerated BEM code*. Arxiv, https://arxiv.org/pdf/1711.01897.pdf.

[5] Xia J., Chandrasekaran S., Gu M., Li X.S. (2010). *Fast algorithms for hierarchically semiseparable matrices*. Numerical Linear Algebra with Applications (2010).