Handling many Hierarchical Matrix calculations using one CUDA kernel launch

# Background

Hierarchical matrices are a powerful tool for representing large matrices, that it uses a combination of dense and low-rank blocks to store matrices in accordance to their ranks, while still maintaining a high numerical accuracy. There exist many researches on how to perform standard matrix operations directly on the hierarchical structure, both on CPU and heterogeneous systems.

The current day heterogeneous matrix calculation libraries, such as MAGMA and CUBLAS, already have very efficient methods doing dense matrix calculations. Building a new heterogeneous h-matrix based matrix library from these dense libraries has potential performance limitations:

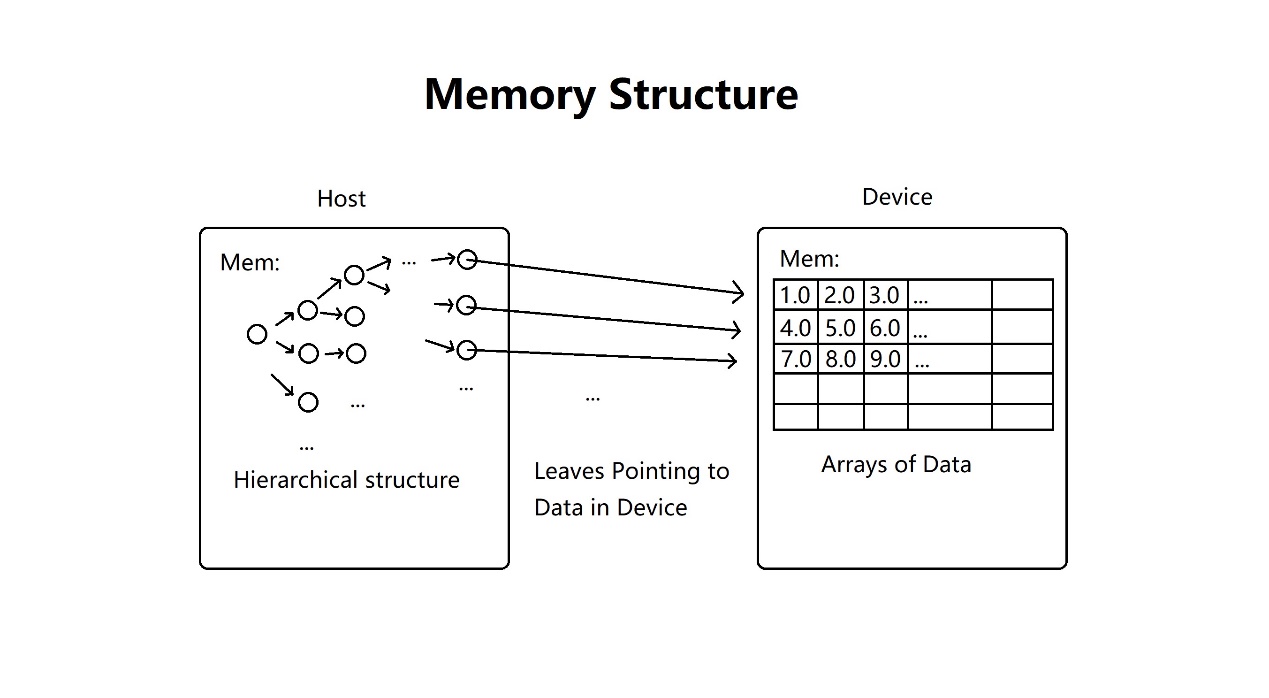
Hierarchical matrix operations all have different problem sizes, which not all blocks in the hierarchical matrix divides into equally sized blocks. Solving a hierarchical problem includes building up the small solutions from dense matrix and low rank matrix solutions. If the problem size is small, launching many routines from the dense libraries could build up into a performance-costly move eventually.

On the other hand, launching many kernels may potentially cause the communication cost between host and device to grow significantly, which there will be many synchronization calls made through the PCI-Express bus. In systems that have multiple GPUs installed, the data needs to move frequently between host and GPU devices, and among different devices.

# Method

In our research, we are proposing to build a brand new kernel that specializes in handling a hierarchical matrix calculation. Using the one single kernel launch to solve an adequately sized hierarchical matrix can potentially help alleviating the communication costs between the host and devices, and resizing the hierarchically divided problems into more consistent problem sizes, due to the fact that a hierarchically partitioned matrix consists of many smaller hierarchical matrices.

Our approach consists of a routine that execute heterogeneously: The CPU only reads the hierarchical structure of the matrix, and build up the necessary information, such as the addresses of the blocks, the block dimensions, and what kind of calculation etc. and pass them to the GPUs. After all information is gathered, CPU initiates the kernel and each GPU executes its corresponding task and starts performing changes to the data.



The hierarchical information gathering done on CPUs consists of majorly 4 parts:

## 1. Generation of a tree of hierarchical operations from the hierarchical structure.

The generated tree consists many hierarchical operations as intermediate nodes too, but eventually they all divide into dense or low-rank operations. The tree flattens and reduces (by eliminating intermediate operations involving hierarchical nodes) into a list of dense and low-rank operations.

## 2. From the list of operations, CPU generates a DAG to find data dependencies that exist between entries of the list.

We determine dependency by the indexes of the nodes (rather than memory location or handles), which are the operands of the hierarchical operations.

## 3. Scheduling the DAG

According to the DAG and the kernel launching configurations, CPU schedules the list of operations into individual instruction memory that dedicates to each worker (in the CUDA context: thread blocks), in a way that both parallelize the list of instructions, while maintaining the data dependencies.

The scheduling algorithm a heuristic algorithm that prioritize first by the FLOPS count of the ready hierarchical operation, and then by the number of outward going dependencies. This algorithm balances the loads well while maintaining a high speed in the progression.

## 4. Generating GPU readable Instructions.

The Instructions for GPU consists of purely integers, which can be either enumerations of opcode or BLAS-like routines, the controlling integers for matrix dimensions, pointer offsets, and transpose indicators, and matrix pointer index in the global memory.

As CPU already collected all information that the GPU needs, it copies the individual instruction memories, bundles the matrix data pointers into CUDA global memory, while at the same time allocates a room for thread blocks to communicate. The kernel is now ready to launch.

Inside GPU Kernel:

## 1. Taking CPU generated instructions, the starting pointers of the data arrays of each node, and a thread-block wide communication space

Instructions has 4 kinds: execute, signal, wait, and finish.

“Execute” takes some data arrays and controlling integers (dimensions, transpose) and do a BLAS-like routine.

“Signal” writes to the communication space, to tell other thread-blocks that some “execute” instruction has finished.

“Wait” reads the communication space and checks for the signal wrote, to postpone the next execution when dependency requirements are not met.

Wait uses spinlock to check for signals wrote, that it reads communication spaces iteratively until some other thread-blocks modify the value.

“Finish” proceeds to the end of the kernel.

## 2. Thread-block level BLAS-like routines.

GETRF: LU decomposition

TRSM: Triangular solve of linear systems.

GEMM: General Matrix-matrix multiplications, subtracting from accumulator

GEMM-plus: General Matrix-matrix multiplications, adding to accumulator.

ACCM: Accumulation of low-rank blocks.

GEMM-3x: General Matrix multiplication with 3 matrices, subtracting from accumulator.

GEMM-4x: General Matrix multiplication with 4 matrices, subtracting from accumulator.