Runtime System for GPU-based Hierarchical LU factorization

Qianxiang Ma
Tokyo Institute of Technology
Tokyo, Japan
ma@rio.gsic.titech.ac.jp

Rio Yokota
Global Scientific Information and Computing Center
Tokyo, Japan
rioyokota@gsic.titech.ac.jp

ACM Reference Format:
Qianxiang Ma and Rio Yokota. 2019. Runtime System for GPU-based Hierarchical LU factorization. In *Proceedings of SC'19*. ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Hierarchical low-rank matrix is a useful representation of compressed large matrices, that it uses a combination of dense and low-rank blocks to store matrices in accordance to their ranks, with controllable numerical accuracy.[4]

Modern linear algebra libraries that utilize GPUs, such as cuBLAS and MAGMA[3], already have very efficient methods doing many dense matrix calculations, but they have very limited support on H-matrix formats. H-matrix libraries on the other hand, utilizes CPUs primarily, such as H2LIB and HLIBpro[2, 7]. GPU-based H-matrix libary, hmglib[5], used batched operations at different levels solving BEM or general matrix multiplications, but it provided little support solving matrix factorization problems.

Highly paralleled hierarchical matrix factorization is more challenging than hierarchical matrix multiplications, due to the data dependencies existing between the operations among the blocks. Ignoring data dependencies causes incorrect factorization results. Another challenge exists in the levels of the hierarchy, that different level tasks have different problem sizes. Launching many routines from the dense libraries solving small-sized tasks has large overheads, that could build up into a performance-costly move eventually.

2 DESIGN

Runtime systems widely exist among linear algebra libraries, such as PaRSEC used by PLASMA, and StarPU used by MAGMA.[6, 8] Other stand-alone runtime systems, such as the Stanford Legion[1], are also popular choices. However, although many of them provide support for CUDA kernel launches and stream management, none could manage the tasks happening inside kernels once launched, not even the cuGraph stream management tool provided in the CUDA programming model. In order to avoid both the kernel setup and the host-device communication overheads from launching many small kernels, we have also ruled out the option of using dynamic parallelism that launches kernels recursively, and decided to take a different approach that can batch the tasks into a higher level, by having a single kernel launched.

Therefore, in our research, we are developing a new runtime system, "Pastel-Palettes", for GPUs, that specializes in solving a hierarchical matrix's LU factorization. Our approach keeps in mind of the different strong points of a heterogeneous computing platform: The CPU has less cores, and less calculation potentials, but it has better branch predictors and more efficient in recursive structures (such as trees). For this reason, we have the CPU reads the hierarchical structure of the matrix, and builds up the necessary task information, batches them together, and then passes to the GPU to do the actual matrix factorization.

2.1 Host

The host section consists of 4 major components:

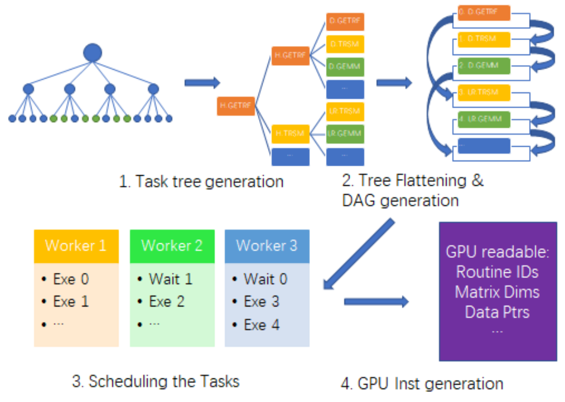

Figure 1: An Illustration of the Host.

- `1. Task Tree Generation`:
  CPU traverses the hierarchical matrix structure and determines the operations needed to be performed on each block to complete the LU factorization, without modifications on the actual data.
- `2. Tree Flattening & DAG Generation`:
  Tree flattening takes the tree of tasks generated, and gets rid of intermediate hierarchical nodes which are fully represented by their dense and low-rank children. Data dependency checks are then performed between each task, and the result is stored as a DAG (Directed Acyclic Graph).
- `3. Scheduling the Tasks`:
  With the tasks and their data dependency information, the scheduler uses 2 different heuristics to create a static scheduling: which task to fetch & which worker to schedule to.

SC'19, November 17 - 22, 2019, Denver, CO, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Additional trimming and the estimated FLOPS are also used to assist the scheduler to further improve device utilization.

- `4. GPU Inst Generation`:
  The last task of the CPU is translating the scheduling and the task information to the GPU correctly. Data pointers are batched together into an array, and BLAS-like and low-rank routines are enumerated. Along with the matrix dimensions and pointer offsets, each worker gets an integer array to represent the tasks being assigned to them.

2.2 Device

As all tasks are properly batched, there is no need for GPU to know the hierarchical structure, which implies no more recursion inside the kernel. In general, the kernel is made of 2 components: kernel level controlling functions and thread-block level routines. Additionally, warp-level intrinsics are used to optimize the thread-block level routines.

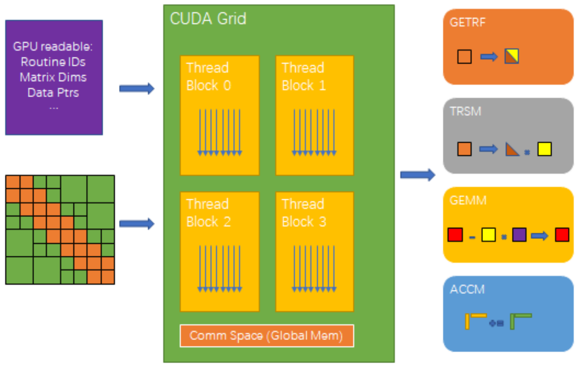

Figure 2: An Illustration of the Device.

On the kernel level, it is sometimes necessary to deliberately halt one or several SMs in order to correctly meet data dependency requirements until another SM finishes its work. However, as NVIDIA does not provide such functions in the CUDA programming model, we allocated space in the global memory as communication space for the SMs, and adopted a spin-lock mechanism. Instead of halting, SMs each assigns 1 thread that actively checks the state of the lock that it is waiting. In order to save the memory bandwidth, and possess minimal impact to other busy SMs, a waiting time hint is also provided by the CPU, correlating to the estimated FLOPS that needs to be completed.

On the thread-block level, several BLAS-like routines and Low-rank routines are implemented as device functions in the kernel. Each device function takes all threads assigned to 1 SM to execute, and utilizes the full amount of shared memory and L1 cache. Inside each device function, vectorized memory I/O and warp shuffling techniques are used to further optimize each routines.

List of routines:

- `GETRF`: LU decomposition.
- `TRSM`: Triangular solve of linear systems.
- `GEMM`: General Matrix-matrix multiplication.
- `ACCM`: Accumulation of low-rank blocks.

3 RESULT

When testing our code, we used double precision (FP64), constant rank 16, and a minimal block size of 256 x 256 for compression. Large low-rank blocks are partitioned deeper to at most N/8 x N/8 to increase task level parallelism. In general, without extreme tuning, we achieved satisfactory performance on a single GPU (NVIDIA Tesla V100 PCI-E 16GB), that can rival some well-optimized CPU-based hierarchical matrix libraries.

Table 1: Pastel-Palettes, H-LU Performance

N	Flops	Host (ms)	Kernel (ms)	Kernel (GFlops)
1024	3.027 E8	1.8	35.0	8.7
2048	8.149 E8	3.3	71.0	11.5
4096	2.440 E9	11.3	148	16.5
8192	6.132 E9	53.9	302	20.3
16384	1.519 E10	150	627	24.3
32768	3.716 E10	472	1305	28.5

Hierarchical low-rank matrix compresses the total FLOPS for LU factorization needed to 0.158% of the dense LU (2.346 E13 Flops, N=32768), which is equivalent to more than 15TFlops/s (from 28.5GFlops/s) if performed directly on a dense matrix.

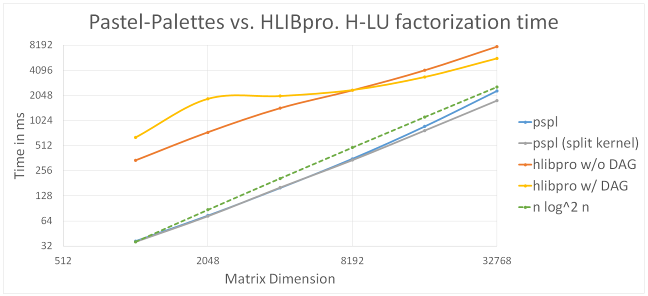

Figure 3: Pastel-Palettes vs. HLIBpro. H-LU Performance.

When comparing to existing H-matrix libraries, HLIBpro[7] also implemented task-based H-LU using only CPU. HLIBpro uses a well-tuned linear algebra library, LAPACK, for dense and low-rank routines. As the result shows, our implementation shows generally faster performance than HLIBpro, and the DAG overheads are less exhibited in the overall performance.

4 CONCLUSION

Conclusively, we discovered that although GPUs are typically considered ineffective handling tree structures and recursions, with enough preprocessing from the CPU, trees can be transformed into batched tasks effectively. Additionally, Hierarchical Low-rank Approximations of matrices compresses the FLOPS required for matrix calculations very significantly, which is not only LU factorization. We believe that our approach could be developed further to accommodate even more kinds of H-matrix calculations.

ACKNOWLEDGMENTS

This work was supported by JST CREST Grant Number JPMJCR19F5.

REFERENCES

[1] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 66, 11 pages. http://dl.acm.org/citation.cfm?id=2388996.2389086

[2] Steffen Börm. 2019. Hierarchical matrix arithmetic with accumulated updates. *Computing and Visualization in Science* (14 Jun 2019). https://doi.org/10.1007/s00791-019-00311-3

[3] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Ichitaro Yamazaki. 2014. Accelerating Numerical Dense Linear Algebra Calculations with GPUs. *Numerical Computations with GPUs* (2014), 1–26.

[4] Wolfgang Hackbusch. 2015. *Hierarchical Matrices: Algorithms and Analysis*. Vol. 49. https://doi.org/10.1007/978-3-662-47324-5

[5] Helmut Harbrecht and Peter Zaspel. 2018. A scalable H-matrix approach for the solution of boundary integral equations on multi-GPU clusters. *arXiv e-prints*, Article arXiv:1806.11558 (Jun 2018), arXiv:1806.11558 pages. arXiv:cs.MS/1806.11558

[6] Reazul Hoque, Thomas Herault, George Bosilca, and Jack Dongarra. 2017. Dynamic Task Discovery in PaRSEC: A Data-flow Task-based Runtime. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA '17)*. ACM, New York, NY, USA, Article 6, 8 pages. https://doi.org/10.1145/3148226.3148233

[7] Ronald Kriemann. 2013. H-LU factorization on many-core systems. *Computing and Visualization in Science* 16 (06 2013), 105–117. https://doi.org/10.1007/s00791-014-0226-7

[8] Samuel Thibault. 2018. *On Runtime Systems for Task-based Programming on Heterogeneous Platforms*. Habilitation à diriger des recherches. Université de Bordeaux. https://hal.inria.fr/tel-01959127