Recursion Brings Speedup to Out-of-Core TensorCore-based Linear Algebra Algorithms: A Case Study of Classic Gram-Schmidt QR Factorization

Shaoshuai Zhang
szhang36@uh.edu
Department of Computer Science,University of Houston
Houston, Texas, USA

Panruo Wu
pwu7@uh.edu
Department of Computer Science,University of Houston
Houston, Texas, USA

Abstract

Out-of-core processing aims to handle large amount of data when the memory is limited. There exists several out-of-core applications including disk-memory and CPU-GPU processing. Ideally, these out-of-core applications can be expected to be close to the peak performance of the in-core computations, if the data movement between different memory hierarchies can be overlapped by the in-core computations effectively. However, with the emergence of matrix accelerators such as TensorCore GPU, the imbalance between the speed of computations and data movement is further exacerbated, such that even high computation intensity kernels can be dominated by data movement cost. In such cases, the algorithms need to be redesigned to reduce communication volume and overlap the data movement by pipelines. In this paper, we select classic Gram-Schmidt QR factorization as an example to illustrate our recursive strategy, which shows smaller amount of data movement and higher overlapping ratio than the conventional blocking QR factorization algorithm. The results suggest this technique can potentially be applied to broader matrix computations kernels.

ACM Reference Format:
Shaoshuai Zhang and Panruo Wu. 2021. Recursion Brings Speedup to Out-of-Core TensorCore-based Linear Algebra Algorithms: A Case Study of Classic Gram-Schmidt QR Factorization. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3472456.3473522

1 Introduction

Out-of-core (OOC) means out of memory (when main memory was called core memory a long time ago). However, OOC has received less attention than distributed parallel computing but we think OOC has its use cases in the era of data-centric computing. Both can enable solving bigger problems that cannot fit in the memory of a single node, but the latter also entails using multiple processing units. In the era of vector and matrix accelerators, we think OOC is becoming more important as it enables an increasingly common use case because of the convenience of using a single node and the ubiquitous accessibility of such platform. Furthermore, for some problems, a single accelerator can rival the speed of a medium sized cluster of CPUs. For example, a single A100 GPU with TensorCore has a peak performance at around 300 TFLOPS, which is equal to about thousands of modern high-end CPU cores with AVX512 vector units. Yet, the memory capacity of the accelerator is often very limited. The limited problem size due to insufficient memory capacity greatly limits the potential of the extremely fast accelerator.

A particular class of problems that potentially benefits from OOC with accelerators is dense linear algebra. In this paper, we explore techniques to enable efficient OOC computing on extremely fast accelerators such that the computing is no longer limited by scarce GPU memory. The key challenge is the management of data movement. It has been shown that the optimal data movement volume is asymptotically $O(\frac{\#\text{flops}}{\sqrt{M}})$ [3] where #flops is the amount of floating-point operations needed and M is the number of words in fast memory. Assuming the computation speed is R_2 (#flops/s) and communication speed is R_1 (#words/s), the time spent moving data is

$$O(\frac{\#\text{flops}}{\sqrt{M}R_1})$$

while the time spent doing computation is

$$O(\frac{\#\text{flops}}{R_2})$$

Now with the TensorCore unit (a.k.a. neural engines, neural processors, or matrix accelerators) the ratio of matrix computation speed and data movement speed is getting extremely high. On a modern Volta or Ampere NVIDIA GPU cards with TensorCore as matrix computing units, the R_2 can be 10^{14}, while $\sqrt{M}R_1$ is $10^5 \times 10^{9.5} = 10^{14.5}$. Assuming an algorithm achieves optimal data movement, the cost of data movement is comparable to computation. Such trend is likely going to continue. With a suboptimal data movement algorithm the communication may dominate computation. In this situation, it's critically important to minimize data movement and increase overlapping between computation and communication, while at the same time without increasing #flops or decreasing the execution rate R_2 significantly.

There has been a long history and interest in OOC computing for linear algebra algorithms. And several similar OOC applications exist that work for CPU-GPU hierarchy. For instance, BLASX [23] and cuBLASXt provides OOC high-performance BLAS3 operations and some other specific applications such as OOC SVD [15] and [14]. However, over a long period of accumulating faster computation speed growth over data movement speed and with the advent of

ICPP '21, August 9–12, 2021, Lemont, IL, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9068-2/21/08…$15.00
https://doi.org/10.1145/3472456.3473522

specialized matrix accelerator in addition to vector accelerator, the balance between computing and communication is further skewed. As a result, the conventional design pattern of OOC algorithms becomes inadequate to hide the cost of data movement, even with the abundance of computation intensity present in dense linear algebra operations.

To make the discussion concrete, we use NVIDIA TensorCore as an example of a matrix accelerator. On NVIDIA Volta and Ampere architectures, matrix computations can not only be carried out by CUDA cores (SGEMM), but also by TensorCore units (TC-GEMM). On Nvidia V100 GPU, CUDA core SGEMM usually performs at 14 TFLOPS, whereas TensorCore TC-GEMM can reach 112 TFLOPS, representing an 8x speedup by using the matrix accelerator. On the newer A100 GPU, the ratio is even higher at 16x. But the time cost of data movement between host and device and memory capacity do not improve commensurately. Consequently, the algorithms used by state-of-the-art works such as BLASX [23] is no longer adequate to hide all the communication costs. For instance, in terms of the V100 PCIe graphic card, the peak rate of data movement is around 13GB/s, which means moving two matrices with a size $16384 * 16384$ from host to device will take around 150ms. To put it into context, the computation using SGEMM will take 630ms while using TC-GEMM will take 97ms. It is evident that data movement will very likely become the bottleneck.

In view of the new situation, the TensorCore-based OOC application optimization needs to be data movement centric. Without careful arrangement of data access patterns, the overall performance will be dominated by the data movement rather than the computations. So, there are two main challenges to be tackled to design high-performance OOC linear algebra algorithms. 1)How to reduce the data movement between different memory levels; 2)How to overlap the data movement with computations.

To illustrate how to address these challenges, we select QR factorization as an example, as it has a well-implemented in-core TensorCore-based algorithm [24]. And in the rest of this paper, we will try to explain why the conventional blocking algorithms cannot work efficiently and why the recursive algorithms have better performance. Besides, we will also discuss the details of how to implement the OOC QR factorization with data movement perfectly overlapped. We consider our contributions to be:

- Our algorithmic analysis, performance modeling, and empirical demonstration suggest that conventional blocking linear algebra algorithm is bottlenecked by either the data movement for fundamental matrix computation such as matrix-multiplication and QR factorization on modern matrix accelerator architectures, or the inefficient GEMMs due to the special shapes restricted by the fixed blocksize.
- We design a novel CPU-GPU hybrid computation module, data movement management, and pipeline based on recursive formulation of matrix-multiplication and QR factorization, which nearly perfectly overlap bidirectional data movement and highly efficient in-core computation, thereby the inside OOC TC-GEMMs can be said to be optimal.
- We demonstrate an end-to-end out-of-core QR factorization algorithm whose performance is up to 2x faster than conventional blocking QR factorization algorithm by comparison, and achieve around 45% of TensorCore peak performance.

The rest of the paper is organized as follows: Section II talks about the related work; section III analyzes the performance behaviors of the blocking and recursive algorithms; section IV explains the implementation and optimization details and section V provides experimental evaluations. And section VI draws a conclusion and depicts the future work.

2 Related Work

The out-of-core processing has a long history, but there's an approximate chronological boundary that before the year 2000, the out-of-core processing is usually deployed on a single disk-CPU hybrid system, while after the year 2000, it's often used on CPU-GPU hybrid system. In the late 2000s, the development of GPGPU and distributed systems attracts many researchers to explore how to use GPU to deal with a massive scale of data and thereby leads to the usage of out-of-core processing on CPU-GPU hybrid system and distributed systems. Since 2017 various devices such as NVIDIA TensorCore, Google Tensor Processing Unit (TPU), and so on are quickly emerging in response to the high demand for neural network training and inference tasks. The speed of the matrix multiplication becomes so fast, that the biggest problem fit into the device memory would only take a second to finish. However without out-of-core capability, such impressive speed cannot be used to solve a larger problem that does not fit into the very limited device memory space. Highly efficient out-of-core matrix computations are highly desirable due to the accessibility and simplicity of a single machine.

2.1 Disk-CPU Out-of-Core

In 1996, the SOLAR [21] was released and it became the first well-implemented out-of-core package. It combines LAPACK [2] and ScaLAPACK [4] based in-core subroutines, out-of-core subroutines and input-output subroutines. Later in 2002, Toledo and Rabani [22] proposed an out-of-core filter-diagonalization method to solve the very large electronic structure calculations. Note that they also proposed an out-of-core recursive QR factorization, which is not included in the SOLAR package, in this work.

With the development of distributed systems, researchers are more interested in solving large problems on distributed machines or multiple threads architecture rather than on a single machine. As a result, out-of-core processing is investigated on distributed systems. D'Azvedo and Dongarra [6] migrated LU, QR, and Cholesky subroutines in ScaLAPACK to out-of-core relative subroutines. And in 2012, Gregorio et al. [19] designed some out-of-core subroutines that don't lose any performance on multithreaded architectures.

2.2 CPU-GPU Out-of-Core

cuBLASXt [1] library offered by Nvidia provides hybrid CPU-GPU implementations of BLAS3 routines. In 2016, BLASX [23] used different out-of-core strategy and the performance is slightly better

[1]https://docs.nvidia.com/cuda/cublas/index.html#using-the-cublasXt-api

than cuBLASXt. In addition, both cuBLASXt and BLASX aims at the multiple-GPUs platform.

There're also some other CPU-GPU OOC applications that solve specific problems. For example, both[14] and[15] discusses the OOC SVD implementation on CPU-GPU hybrid system.

2.3 TensorCore Technology

In 2017, TensorCore technology was introduced by Nvidia on its Tesla architecture[17]. There're some investigations [16], micro-architecture analysis and benchmarking [13]. Except half precision GEMMs, researchers also indicate that TensorCore can also be used for reduction and scan [5]. In [10–12], TensorCore was used for accelerating linear system solvers in the framework of hybrid CPU/GPU linear algebra package MAGMA[7]; and TensorCore based QR factorization [24] mentioned before.

2.4 Recursive Linear Algebra Algorithms

The recursive linear algebra algorithms have long been known to the numerical algorithm communities and have been shown to be advantageous over blocking algorithms, however such performance gains are typically rather small, as blocking alone was able to achieve near peak performance on previous architectures. Examples include recursive Cholesky and LU proposed in Lawra [1] and ReLAPACK [18], recursive partial pivoted LU by Toledo [20]. Recursive QR factorization has been studied by [9] as panel factorization solver and extended to full matrix factorization to TensorCore [24]. Recently, Zhang has proposed some TensorCore recursive BLAS3 algorithms [25] that can be used in matrix factorization. A recursive version of the Linpack benchmark (LU factorization) is studied on iPad 2 in [8].

3 Performance analysis

In this section, we'll analyze the performance of the recursive algorithm and compare it with the blocking algorithm. We'll discuss the amount of data movement and the overlapping behaviors of OOC TCGEMMs. We'll also give a brief introduction to QR factorization to make the analysis more understandable.

3.1 QR factorization

The QR factorization, also known as QR decomposition, aims to factorize a matrix A into a product of an orthogonal matrix Q and an upper triangular matrix R. It's widely used in the scientific and engineering area to solve orthogonalization, linear least square problems, eigenvalue decomposition, and singular value decomposition problems. In general, there are three common-used algorithms, including Gram-Schmidt, Householder, and Givens rotation. In this paper, we use the classic Gram-Schmidt to illustrate our methodology, thus we'll only give the background knowledge of the classic Gram-Schmidt QR factorization algorithm.

3.1.1 Gram-Schmidt process The Gram-Schmidt process aims to find a set of orthonormal vectors in an inner product space. Given an array of linearly independent vectors $[a_1|a_2|a_3|...|a_n]$, the Gram-Schmidt will find the orthonormal basis one by one. Then the set of orthonormal vectors $[q_1|q_2|q_3|...|q_n]$ is given by Equation (1).

$$\begin{aligned}
&u_1 = a_1, && q_1 = u_1/||u_1|| \\
&u_2 = a_2 - \mathrm{Proj}_{u_1}(a_2), && q_2 = u_2/||u_2|| \\
&u_3 = a_3 - \mathrm{Proj}_{u_1}(a_3) - \mathrm{Proj}_{u_2}(a_3), && q_3 = u_3/||u_3|| \\
&\vdots && \vdots \\
&u_n = a_n - \sum_{j=1}^{n-1} \mathrm{Proj}_{u_j}(a_n), && q_n = u_n/||u_n||
\end{aligned} \tag{1}$$

in which $\mathrm{Proj}_u(a) = uu^T a = u(u^T a)$ is the orthogonal projection of the vector a onto an unit vector u. We also obtain R during the Gram-Schmidt process: $r_{ij} = q_i^T a_i$. Depending on the evaluation order of the procedure described in (1), there are two mathematically equivalent but numerically different variants called classic Gram-Schmidt (CGS) and modified Gram-Schmidt (MGS). CGS executes row by row in (1), whereas MGS subtracts $\mathrm{Proj}_{u_i}(a_j)$ from a_j for all $j > i$ as soon as u_i is computed. MGS can be visualized as evaluating (1) with row and column interleaved. This subtle difference has important implications in numerical stability and parallelism exposed. MGS is more stable but less parallel.

As described above, the Gram-Schmidt process is inefficient on hierarchical memory systems due to the low data locality of the vector-matrix operations. To improve locality, blocking must be applied such that the orthogonal projection of multiple vectors onto multiple orthonormal vectors is performed in one shot. CGS can be trivially blocked since it can be directly "upgraded" into blocking algorithm by considering the vectors a_i, u_i not as vectors but as a block matrix (a group of column vectors). MGS, on the other hand, is not obvious to block.

3.1.2 Blocking Strategy The blocking QR factorization's workflow is shown in Fig 1. At first, we select a blocksize b and factorize the first b columns (termed as panel). Then we have the factorized Q_1 and let it time the rest of the matrix A_2 to get R_{12}. The next step is updating the rest of the matrix A_2. After that, we start a new panel factorization and repeat the steps until the last panel is factorized. See Fig 1 for the intuitive steps.

3.1.3 Recursive Strategy Compared to the blocking strategy, the recursive strategy is not that popular. The reasons might be that blocking algorithms give enough data locality and parallelism on previous architectures; but now on the new matrix accelerators, it no longer can. The recursive algorithm is assembled like this equation (2) and the workflow is: at first, we divide evenly its columns into two halves, denoted by $A = [A_1|A_2]$. At first, we factorize the first half $A_1 = Q_1R_{11}$, and then compute north-east quarter of $R_{12} = Q_1^T A_2$. Next we update the second half $A_2 = A_2 - Q_1R_{12}$. Finally, QR factorize the updated second half $A_2 = Q_2R_{22}$. Note that the QR of the two halves can be recursed using this algorithm itself. Fig 2 shows the general steps of factorizing a matrix with 2-level recursion.

$$[A_1|A_2] = [Q_1|Q_2]\left[\begin{array}{c|c} R_{11} & R_{12} \\ \hline & R_{22} \end{array}\right] \tag{2}$$

Zhang [24] has explained and shown the recursive QR performs better than the conventional QR on TensorCore. The reason behind the speedup is that recursive algorithms can provide larger GEMMs

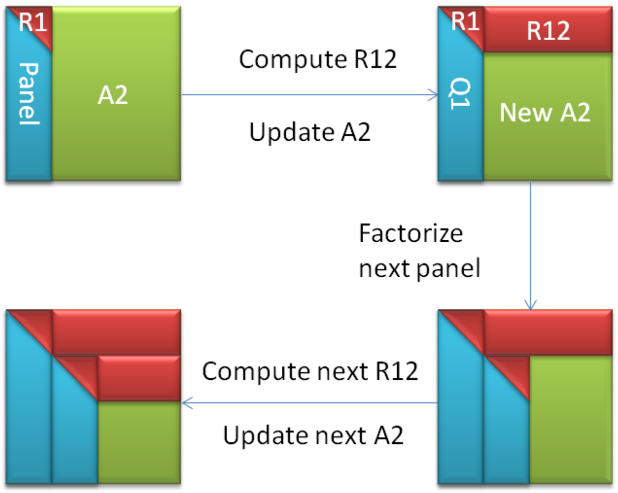

Figure 1: Blocking QR factorization steps

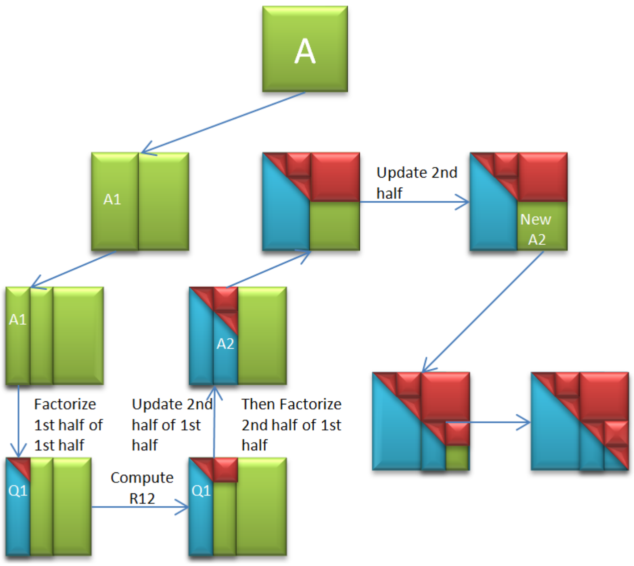
A
A1
Update 2nd half
New A2
A1
A2
Factorize 1st half of 1st half
Update 2nd half of 1st half
Then Factorize 2nd half of 1st half
Q1
Compute R12
Q1

Figure 2: Blocking QR factorization steps

which can be executed more quickly on TensorCore, while the conventional algorithms cannot provide such GEMMs. Different from the in-core algorithms, the OOC algorithms focus more on the data movement and elaborate pipelining issues. So, the following subsections will discuss the effects on the data movement of such algorithms, by using OOC QR factorization as an example.

3.2 The Amount of Data Movement

In the out-of-core scenario, the panel factorization will be performed on GPU step by step, while the GEMMs will be out-of-core and follow some specific computing pattern. But in this section, to simplify the computations, we'll only discuss the data movement of the naive implementation, that the elements will not be reused and it can give us an overview of the data movement overhead in the worst case.

3.2.1 Blocking Strategy Given a matrix with size $m \times n$, where m is the number of rows and n is the number of columns. The panel size (blocksize) is b and the matrix will need k iterations to be fully factorized. Then we have $n = kb$. In the i-th iteration, the overall host to device data movement is $mb + mb + m(n - ib) + mb + b(n - ib) + m(n - ib)$, and the overall device to host data movement is $mb + b^2 + b(n - ib) + m(n - ib)$. As a result, during the entire factorization, the amount of data movement from host to device is:

$$\sum_{i=1}^{k}[3mb + (2m + b)(n - ib))] = (k + 2)mn + \frac{n^2}{2} - \frac{nb}{2}$$

And the amount of data movement from device back to host is:

$$\sum_{i=1}^{k}[mb + b^2 + (m + b)(n - ib)] = \frac{1}{2}[(k + 1)mn + n^2 + nb]$$

3.2.2 Recursive Strategy The amount of data movement in the recursive algorithm is a bit more complicated. First of all, the recursion depth is $\log_2^k$. Different from the blocking algorithm, only the deepest recursion does the factorization, while other levels of recursion perform GEMMs. So, the overall data movement of the deepest recursion is mn. For the recursions of GEMMs, the amount of data movement is $\sum_i^{\log_2^k - 1}[2mn + 2^{i-1}b^2]$. Hence, the total amount of data movement from host to device is:

$$2(\log_2 k + 1)mn + \frac{mn}{2} - \frac{nb}{2}$$

Similarly, the amount of data movement from device to host is:

$$(\frac{1}{2}log_2^k + 1)mn + \frac{n^2}{2}$$

According to the quantitative analysis, it can be observed that both the amount of data movement of blocking and recursive OOC QR is highly related to the number of blocks k. Obviously, the data movement of the recursive QR is related to $\log k$, while the blocking QR is linearly related to k. This means the gap will become even larger as the k increases. Note that this is only a rough estimation of the data movement because it's based on the assumption that all of the data won't be reused. In practice, some of the matrices can definitely be reused in the computations. But the results can still be a good reference to show the advantages of recursive algorithms, and we will also give the real quantitative data movement results in the evaluation section.

3.3 Overlap Ratio in GEMMs

In the above subsection, we've discussed the data movement behaviors and it has been shown that the recursive algorithm has less data movement. In this subsection, we'll try to explain that the recursive algorithm has advantages over the blocking algorithm with regard to GEMMs overlap ratio (here the overlap ratio means how much the data movement can be overlapped by computation). Typically, there are two types GEMMs, one is an **inner product** (to simplify the name, we'll call the GEMM types in Fig 3, 4 inner product) to generate $R_{12} = Q_1^T A_2$ and another is an **outer product** (Fig 5, 6) to update $A_2 = A_2 - Q_1 R_{12}$. The overlap patterns of these two types are different, thus we'll discuss them separately.

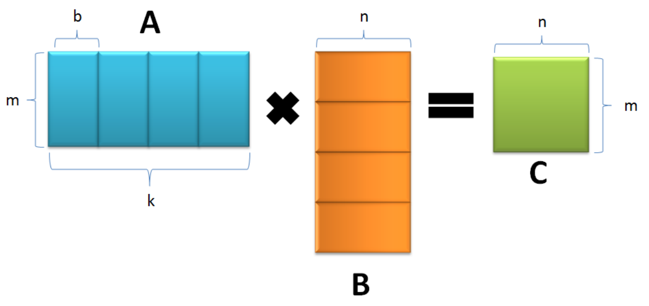

Figure 3: Out-of-core inner product tiling strategy in recursive QR factorization

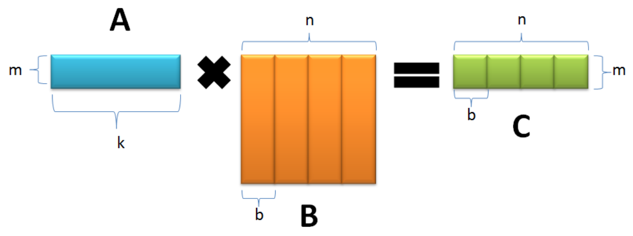

Figure 4: Out-of-core inner product tiling strategy in blocking QR factorization

3.3.1 Inner Product Suppose we have two matrices A and B with size $m \times k$ and $k \times n$. In the recursive QR factorization, m equals to n. As the matrix is typically too large to store on the device, we may want to divide the matrices into several tiles. For each tile, we will move in and move out GPU memory to perform computations. To obtain the best performance, the time cost of data movement should be overlapped by the time cost of in-core TCGEMMs.

To minimize the data movement in recursive QR factorization, we follow Fig 3 to divide the two matrices. In this case, the matrix A and B will be accessed only once. Then the data movement from device to host will cost $\frac{4mk+4nk}{R_m}$ and the TCGEMM will cost $\frac{2mnk}{R_g}$, where R_m and R_g denote the CPU-GPU data transferring rate and TCGEMM computation rate respectively. To hide the data movement cost by overlapping, $\frac{4mk+4nk}{R_m}$ must less than $\frac{2mnk}{R_g}$, which means m should larger than $\frac{4R_g}{R_m}$. On V100 GPU, the R_g is around 90TFLOPs and R_m is around 12GB/s if using pinned memory. Then the final m should be larger than 30,000. And this is usually the case for problems that require out-of-core computation.

When it comes to the blocking algorithm, we can let the matrix A to be stored on GPU and divide B into blocks with blocksize b, see 4. Therefore, the inequality becomes $\frac{4kb}{R_m} < \frac{2mkb}{R_g} \Longrightarrow m > \frac{2R_g}{R_m}$ and the final $m = 15,000$.

It seems that the m in blocking algorithm is easier to achieve, but one thing should be mentioned that the m in blocking algorithm is typically relatively small and fixed to allow the panel to be stored on the device, especially when the GPU memory is limited and the number of columns of the original matrix is large. For example, if the GPU memory is 16GB and we want to factorize a matrix with size 131072×131072, then the m is typically less than 10,000 to avoid running out of GPU memory. In contrast, the m in the recursive algorithm is more flexible and not related to the blocksize. Still, with a GPU has 16GB memory, factorize a 131072×131072 matrix will lead to four inner products if the blocksize is 8192, including $8192\times131072\times8192$, $16384\times131072\times16384$, $32768\times131072\times32768$ and $65536\times131072\times65536$. At least the data movement in the largest two GEMMs can be perfectly overlapped (computing bound), while all of the GEMMs in blocking QR are memory bound. Regarding the smaller GEMMs in the recursive QR, we'll discuss how to optimize them in the next section.

3.3.2 Outer Product Similarly,suppose the GEMM size is $m \times n \times k$ (matrices sizes are $m \times k$ and $k \times n$). For the outer product, we'll use a different strategy to minimize the data movement.

For the recursive algorithm, we can let the matrix B to be stored on GPU, and we move the blocks with size $b \times k$ of matrix A and C in and out (Fig 5). Then the data movement will cost $\frac{4bk+4bn}{R_m}$ and the TCGEMM will cost $\frac{2bkn}{R_g}$, thus we have $n > \frac{4R_g}{R_m} = 3e4$, which is the same as previous result.

For the blocking algorithm, as the matrix A and B is tall and skinny, we can let both A and B to be stored on GPU to avoid unnecessary data movement. In this case, we can only care about the data movement of C, and the strategy is shown in Fig 6. The time cost of moving C from CPU to GPU will take $\frac{4b_1b_2}{R_m}$ and we'll spend $\frac{2b_1kb_2}{R_g}$ on TCGEMM. Hence, the k should be larger than 1.5e4 to overlap the data movement.

Although the strategy is different in the two GEMM types, the results are the same. Based on our analysis, we cannot expect the data movement of the blocking algorithm to be overlapped if we desire minimum data movement. Note that we didn't compute the time cost of moving data from GPU to CPU, as this kind of data

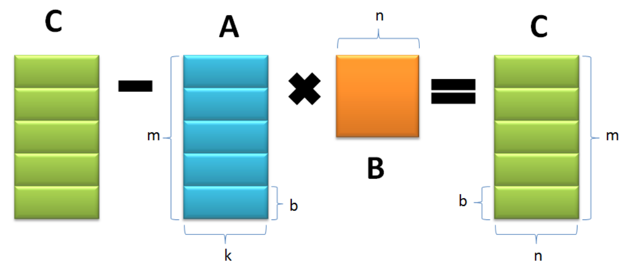

Figure 5: Out-of-core outer product tiling strategy in recursive QR factorization

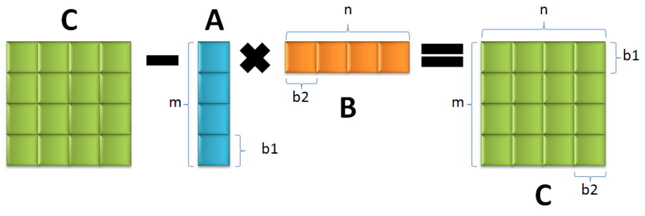

Figure 6: Out-of-core outer product tiling strategy in blocking QR factorization

movement can always be overlapped by moving data from CPU to GPU.

3.4 Summary

We've mentioned in the first section that in designing a high-performance OOC algorithm, the data movement should be reduced and the overlap ratio should be higher. From our above analysis, we could find that the recursive algorithm has advantages over the blocking algorithm because it can reduce data movement and increase data overlap ratio. Specifically, the performance of GEMMs in the recursive algorithm is less relevant to the blocksize, while the performance of GEMMs in the blocking algorithm is restricted by the blocksize. This means the recursive algorithm is more likely to perform even better on GPUs with smaller memory or larger scale matrices. In addition, we can also have more flexibility to optimize the large GEMMs in the recursive algorithm, because we don't really care about the blocksize. For the small GEMMs, we'll introduce the optimization strategy in the next section.

4 Implementation and Optimization

We've already shown the recursive QR factorization has better data movement behaviors and overlapping ratio, the implementation and optimization are very important as well. So, in this section, we'll illustrate our implementation and optimization strategy. Note that we divide the optimization into two parts, the first part is the optimization inside GEMMs, which is called GEMM-level optimization; the second part is optimizing the data transferring between panel factorization and GEMMs, and it's termed as QR-level optimization.

4.1 GEMM-Level Implementation and Optimization

Similar to the performance analysis, the two types of GEMMs have two different implementations.

4.1.1 Inner Product Typically, to implement a high concurrent CPU-GPU program, we should be clear about how to use the streams. Cuda streams[2] are introduced by Nvidia to solve the concurrency problems of executing programs on the device. Generally speaking, with streams and proper devices (most modern Nvidia GPUs support CUDA streams), moving data from device to host, from host to device, and in-core computations can be executed at the same time. In our problem scenario, moving data from host to device, in-core computations, and moving data from device to host can work concurrently, because computations can be overlapped by communications, and the host-device communication links are bidirectional. Therefore, we need at least three streams to make these three assignments run in parallel. But because it's inner product, it's unnecessary to transfer the matrix C from device to host before the entire computations is ended. Thus, we can only let the move-in operations and the computations be asynchronous, and move the data out in the end. In this case, we create several streams and each stream takes charge of one block. The final pipeline is shown in Fig 8.

4.1.2 Outer Product In terms of this type of GEMM, as we discussed before, the B is already stored on GPU (the B is the result of the inner product, there's no need to copy it from CPU to GPU again). So, we only need to read tiles in A and C and write to the tiles in C. But because of the limited GPU memory, we use the same GPU memory space to store the matrix C, which means the move-in operation should be performed after finishing the entire move-out operation to avoid conflict. This waiting impedes the performance because the move-out operation cannot be overlapped. To solve this problem, we use extra memory space to temporally store the C and once the move-out operation is done, we perform a fast in-core data transferring, and at the same time, we start another move-in operation, see Fig 10.

4.1.3 Blocksize Matters The blocksize of GEMMs cannot be selected casually. Smaller blocksize leads to slower in-core computation, and larger blocksize will increase the overhead of the first move-in operation and the last move-out operation. So we can follow this strategy: start with a relatively small blocksize and gradually increase it to the max blocksize. Consider this situation, when we're computing the largest inner product in recursive QR with GEMM size $65536 * 131072 * 65536$, and we set the blocksize to be 8192. Then the first step is to compute a GEMM with a size of $65536 * 8192 * 65536$. Obviously, if we start with blocksize 8192, then the first move-in operation cannot be overlapped by computations. But if we start with blocksize 2048 and gradually increase the blocksize to 8192, then the same part of the move-in operation of the first block ($65536 * 8192$) can be overlapped. Actually, based on our experimental results, this trick increases the inner product performance from around 85TFLOPs to 87TFLOPs.

4.2 QR-Level Implementation and Optimization

So far, the optimization of OOC GEMMs is done, but there are still some problems left. For example, how to deal with small GEMMs? The m or n in small GEMMs is not large enough so that the data movement cannot be overlapped perfectly. Indeed, we cannot optimize the small GEMMs anymore because of the insufficient data intensity, but we can tackle this problem in the QR-level optimization that can helps us avoid unnecessary data movement. The optimization doesn't happen inside GEMMs but works between QR panel factorization and the GEMMs. Besides, using the results from the inner product in the outer product to reduce unnecessary data movement can also be regarded as QR-level optimization to some extent.

So, after answering what's QR-level optimization and implementation, the question becomes how to implement and optimize it. Actually, the implementation is quite easy, following the design pattern of the in-core recursive QR [24] is the solution. But the difficulty is how to optimize it, including cutting off the unnecessary data movement and enabling cross BLAS operation overlapping.

The first optimizing strategy is cutting off some move-in operations of the panel. Because when the GEMM size is small, the entire results can be stored on GPU and they can be used directly in the next panel factorization. Similarly, when the GEMM size is small, it doesn't really need to read the matrix A from CPU, it can directly use the panel factorization results and only read B from CPU.

The second optimizing strategy is trying to hide the move-out operations between panel factorization and GEMMs. For example,

[2] https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/

when the panel factorization is finished and the factorized matrix is ready to be transferred to host, we can perform a move-in operation for inner product at the same time. This kind of overlapping can also happen between inner product and outer product. When we're moving R_{12} out, we can move in the first blocks of A and C. Also, after the outer product, the last move-out operation can be overlapped by moving in the first few columns of the panel. See Fig 13 for the intuitive optimization results.

5 Experimental evaluation

In this section, we're going to show our experimental results of the OOC QR factorization. We will evaluate the performance of the GEMMs and the entire QR factorization, by providing the overall execution time and the timelines.

For all experiments we use a3.10.0-1160.11.1.el7.x86_64 Linux operating system with 128GB memory. The GPU is NVIDIA V100 GPU (32GB) PCIe version. The CUDA version is 10.1, which contains a C++ compiler and cublas library. The in-core QR factorization is based on the LATER project.[3]

5.1 The Performance Behaviours of GEMMs

As we discussed before, the shapes and sizes of GEMMs in the recursive QR and blocking QR are different, therefore, in this subsection, we'll study the performance behaviors of the largest GEMMs (because they cost a large portion of the entire QR factorization time) in recursion and blocking. Table 1 and Table 2 show the quantitative results of the GEMMs, including the time cost of data movement, GEMMs, overall time cost, and relative flops. Meanwhile, Fig 7, 8, 9 and 10 give the timelines of the different OOC GEMMs.

5.1.1 Inner product performance In terms of inner product, in the previous analysis, we assume that all in-core computations run at 90TFLOPs. But the results suggest that the largest GEMM in blocking QR doesn't obey the rule, as the in-core GEMM only runs at 52.6TFLOPs, while the in-core recursive GEMM runs at 99.9TFLOPs. This is probably because of the special shape of the blocking in-core GEMM. Actually, in the TensorCore-based QR paper [24], the authors also mention that tall and skinny GEMMs are very hard to run at peak speed on TensorCore. Fortunately, the in-core recursive GEMMs don't have this weak point when the GEMMs are large enough. No doubt that the small GEMMs will also have some problem, but after all, they're only a small part of the entire computations, while the blocking GEMMs need to face this problem in the complete period because it consists of fixed small-sized and shaped GEMMs which are inefficient on TensorCore.

Both timelines of the inner product (Fig 7,8) show a good overlap rate. In other words, the time cost of close to peak GEMMs dominate the entire computations.

5.1.2 Outer product performance When it comes to the outer product performance, things become slightly different. When the QR blocksize is 16384, there's no big difference between the two types of GEMMs. Both the in-core GEMM and the overlap rate are desirable. But because the experiments are performed on V100 GPU which has 32GB device memory, the QR blocksize can be as large as

Single Block Time Cost	Recursive	Blocking
Host to device	693ms	728ms
GEMM	1408ms	1337ms
Device to Host	1306ms	81ms
In-core flops	99.9TFLOPs	52.6TFLOPs

Overall Time cost	Recursive	Blocking
Synchronous	18183ms	14920ms
Synchronous flops	62.0TFLOPs	33.0TFLOPs
Asynchronous	12932ms	11286ms
Asynchronous flops	87.1TFLOPs	43.6TFLOPs

Table 1: Inner product behaviors, recursive matrix size is 65536*131072*65536 with blocksize 16384, blocking matrix size is 16384*131072*114688 with blocksize 16384

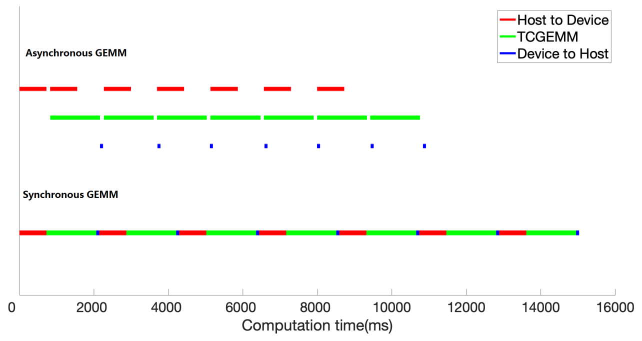

Figure 7: The timeline of computing max inner product GEMM in 0.13M*0.13M in blocking QR factorization, the matrix size is 16384*131072*114688, the blocksize is 16384.

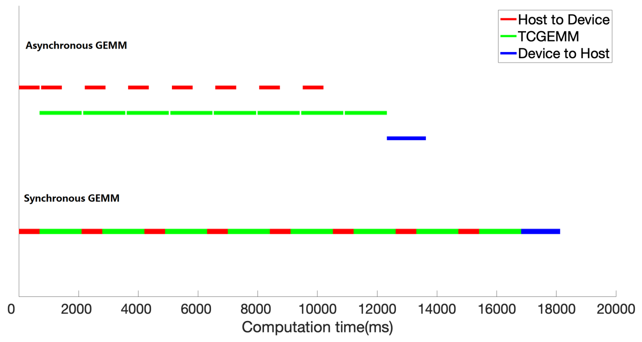

Figure 8: The timeline of computing max inner product GEMM in 0.13M*0.13M in recursive QR factorization, the matrix size is 65536*131072*65536, the blocksize is 16384.

16384 so that the data movement is able to be overlapped perfectly based on our analysis.

However, most of the widely-used GPUs, such as RTX 20,30 series, don't have such a big device memory. On these GPUs, the QR blocksize need to be reduced as a compromise. For example, Fig 11 shows the timeline of the blocking GEMMs with QR blocksize 8192, and we can see that the GEMMs cannot be overlapped anymore.

[3]https://github.com/Orgline/LATER

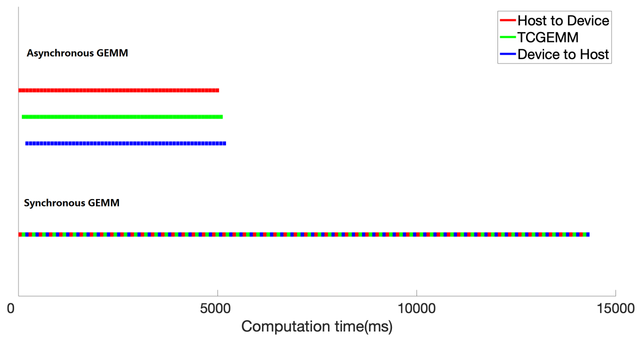

Figure 9: The timeline of computing max outer product GEMM in 0.13M*0.13M in blocking QR factorization, the matrix size is 131072*16384*114688, the blocksize b_1, b_2 is 16384 and 16384.

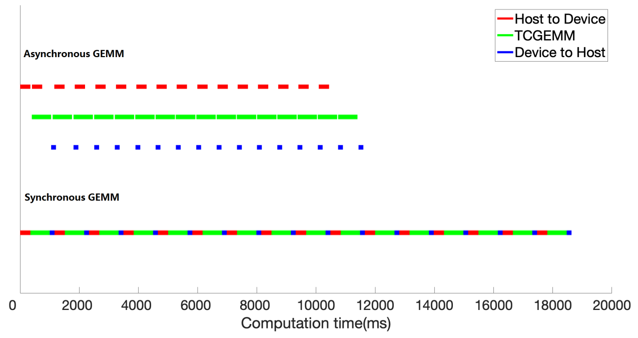

Figure 10: The timeline of computing max outer product GEMM in 0.13M*0.13M in recursive QR factorization, the matrix size is 131072*65536*65536, the blocksize is 8192

Quantitatively, the time cost of host to device, GEMM, and device to host is 347ms, 170ms, 326ms. The computation time is dominated by the data movement, thus even if the total number of the flop of blocking GEMMs with QR blocksize 8192 is around 2x less than the GEMMs with QR blocksize 16384, the computations still spend more time.

Single Block Time Cost	Recursive	Blocking
Host to device	347ms	86ms
GEMM	654ms	89ms
Device to Host	163ms	81ms
In-core flops	107.6TFLOPs	98.8TFLOPs

Overall Time cost	Recursive	Blocking
Synchronous	14129ms	5119ms
Synchronous flops	60.3TFLOPs	34.7TFLOPs
Asynchronous	11517ms	11286ms
Asynchronous flops	97.7TFLOPs	96.2TFLOPs

Table 2: Outer product behaviours, recursive matrix size is 131072*65536*65536 with blocksize 8192, blocking matrix size is 131072*16384*114688 with blocksize 16384 and 16384

In fact, the implementations of the GEMMs can be said to be optimal. For example, in terms of the outer product, our OOC GEMM will cost 11,517ms in total and the in-core computation costs 654*16=10,464ms. Add the first move-in and the last move-out time consumption (because they cannot be overlapped at all), we get the ideal time cost is 10,974ms. This is a fixed upper bound of the given GEMM with blocksize 8192. But there's no doubt that in real computations, some time will be wasted due to the delays and synchronizations, so the gap (543ms) is reasonable. In other words, the GEMMs are optimal.

5.2 The Performance Behaviour of QR

The performance of the two types of QR factorization is highly affected by the GEMMs because the in-core panel factorization is exactly the same. This means the recursive algorithm has big advantages over blocking algorithms because it has faster GEMMs. With

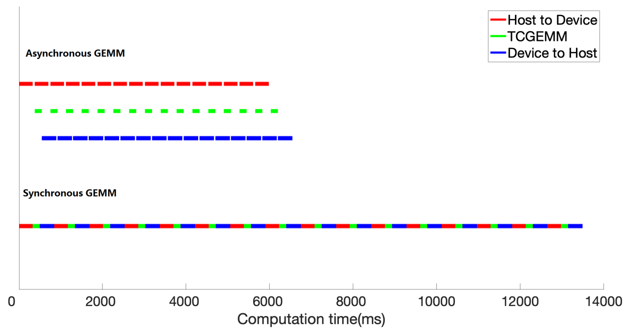

Figure 11: The timeline of computing outer product GEMM with QR blocksize 8192, the matrix size is 131072*16384*131072, the inside GEMM blocksize b_1, b_2 is 32768 and 32768.

regard to the QR-level optimization and overlapping, according to our analysis, the acceleration from reducing data movement and overlapping is very similar.

Fig 12 and Fig 13 demonstrate the timeline of the entire QR factorization. It's obvious that with QR blocksize 16384, the recursive QR takes advantage of the higher rate of inner product thereby beats the blocking QR factorization. We can also find the QR-level optimization helps the two factorization gain around 15% speedup.

However, when the blocksize is 8192, the performance of blocking QR goes down significantly due to the increasing time cost of GEMMs, as the data movement cannot be overlapped by the in-core GEMMs anymore (the time cost is 347ms, 170ms, 326ms for move-in, GEMM, move-out respectively). In contrast, the performance of recursive QR doesn't change much. It's not surprising to see this result, because the panel factorization is recursive, therefore, the only difference is the amount of data movement (can be overlapped mostly). We simulate the factorization by limiting the memory usage to be less than 16GB on V100 GPU and the results are shown in Fig 14, 15. The speedup of recursive QR factorization is nearly 2x compared to the blocking QR factorization. The results also suggest that the higher ratio computation speed/memory capacity is, the more advantageous recursive vs. blocking is.

Data movement time	Recursive	Blocking
Host to device	37.9s	47.2s
Device to Host	19.3s	22.3s

Table 3: The time cost of data movement of two different types of QR factorization with QR blocksize 16384

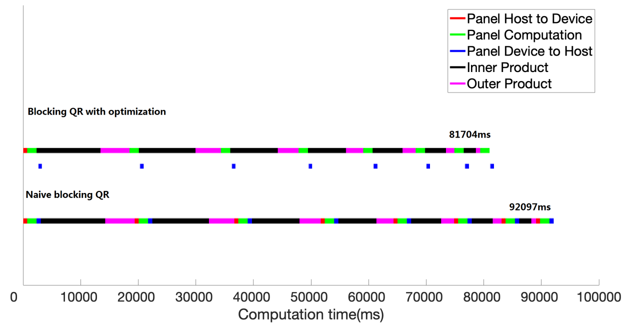

Figure 12: The timeline of computing blocking out-of-core QR, the blocksize is 16384.

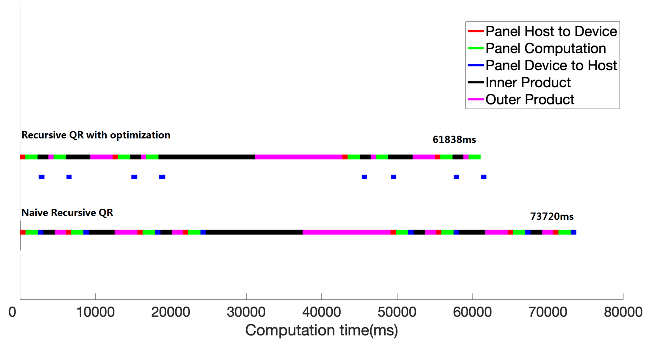

Figure 13: The timeline of computing recursive out-of-core QR, the blocksize is 16384.

Another obvious difference between the two factorization is the amount of the overall data movement. See Table 3. The results can verify our claim that the recursive algorithm has less data movement than the blocking algorithm. Actually, the data movement is not very important here, because most data movement can be overlapped. But it's still valuable to see such difference because in some specific situations where moving data can not be hidden, the recursive algorithm still can help.

To make our experiments more complete, we also experiment with different matrix shapes and sizes. However, limited by our main memory capacity, we only tested the matrices with sizes 65536*65536 and 262144*65536. Table 4 demonstrates the time cost of the GEMMs and panel when we're using the blocking and the recursive strategy with blocksize 8192, respectively. In terms of the overall performance (Panel+GEMMs+data movement), we get 1.5x and 1.7x speedup respectively. As we can see in the Table 4, the time cost of the panel is the same among the two strategies, because

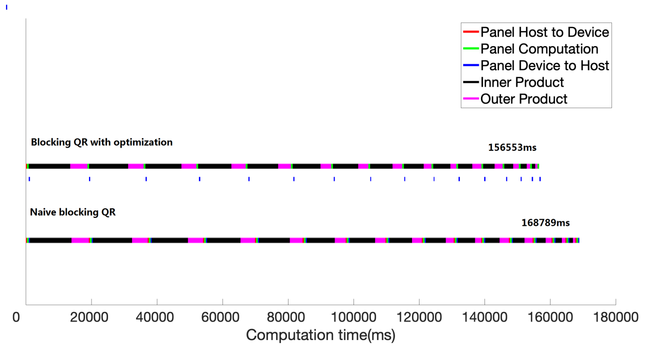

Figure 14: The timeline of computing blocking out-of-core QR, the blocksize is 8192.

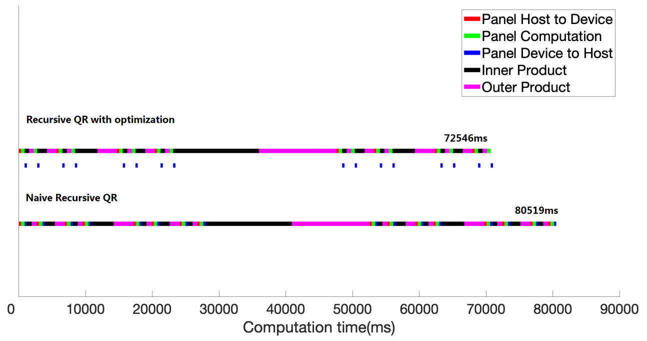

Figure 15: The timeline of computing recursive out-of-core QR, the blocksize is 8192.

we're using the same in-core panel factorization. The time cost of the GEMMs show significant differences, for the reason that the recursive strategy can provide larger data overlap ratio and better matrices' shapes to be performed on TensorCore. However, we can observe that the speedup is not as high as the size 131072*131072 (2.0x) because of two reasons: 1) the size 131072*131072 has larger proportion of GEMMs, which means it has more opportunities to enjoy the faster OOC GEMMs; 2) the matrix shapes are easier to be accelerated on TensorCore (the GEMMs in the size 262144*65536 are taller and thinner). Based on the these experiments, we can conclude that the recursive QR factorization usually show a great advantage over the blocking algorithm on different sizes, and the larger and the more square matrices the matrices are the better.

5.3 Summary

In this section, we've shown the experimental results that can support our previous claim: 1) The data movement in recursive QR is less than the data movement in blocking QR; 2) The blocksize of QR factorization needs to be larger than 15k to overlap data movement perfectly; 3)the recursive QR has great advantages over blocking QR in terms of speed.

Generally speaking, the recursive OOC QR is around 1.25x faster than the conventional blocking QR on GPUs with larger device memory, and around 2x faster than blocking QR when the memory is small.

Partition	Recursive	Blocking
Matrix Size	65536*65536	
GEMMs	10.5s	18.9s
Panel	2.7s	2.7s
Matrix Size	262144*65536	
GEMMs	38.5s	77.0s
Panel	9.0s	9.0s

Table 4: The total time cost of GEMMs and panel of two different sizes of QR factorization (65536*65536 and 262144*65536) with blocksize 8192.

6 Conclusion and Future work

In this paper, we've discussed the recursive out-of-core QR factorization that shows better performance over blocking out-of-core QR factorization. We've implemented the recursive QR factorization and the experiments imply that such acceleration is generally from the inside GEMMs. We've also analyzed the reasons behind such speedup. In short, the GEMMs in conventional blocking QR factorization cannot run at peak performance on TensorCore, neither overlap the data movement by in-core computations due to the fixed blocksize, while the GEMMs in recursive QR factorization is insensitive to the blocksize, in fact, it provides dynamically adjusted blocksizes. Hence, they're more flexible and easier to be optimized thereby can perform faster.

It's interesting to think about if this kind of strategy can be applied to other applications such as LU and Cholesky factorization. As there's no in-core TensorCore based partial pivoted LU and Cholesky factorization, we can only analyze it theoretically. The pattern of out-of-core LU and Cholesky factorization is very similar to the out-of-core QR factorization, which is interleaving panel factorization and trailing matrix update. For example, the trailing matrix update in LU factorization is also of outer product form, and the recursive algorithm can definitely help this kind of GEMMs. Anyway, the other out-of-core linear algebra algorithms are included in our future work.

Another attempt will be considered in the future is trying to deploy the recursive algorithms on A100 GPU, whose peak performance of TensorCore is over 300TFLOPS. In this situation, the blocksize must be larger than 60k if we want the inside GEMMs to be computation-dominated. So, it becomes impossible for blocking algorithms to have such a big blocksize due to the limited memory space. But for recursive algorithms, the dominant large GEMMs can run at full speed, which indicates the recursive algorithm might be even more advantageous than on V100. More accessible architectures such as RTX20,30 series with slightly reduced computation speed and much reduced memory capacity will see a bigger advantage of recursion as well for the same reason. Going forward, the gap between computation speed and data movement speed/memory capacity is likely going to continue to increase, in which cases conventional blocking will be increasingly data movement constrained and has ever lower efficiency.

Acknowledgments

We are very grateful for the valuable suggestions from the anonymous reviewers and their time and efforts in reviewing the manuscript. We would like to thank the University of Houston and the state of Texas for providing the startup funding in supporting this work. We acknowledge the use of the Carya Cluster and the advanced support from the Research Computing Data Core at the University of Houston to carry out the research presented here. We are also grateful for the generous of hardware donation of NVIDIA Corporation. We gratefully acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper.

References

[1] Bjarne S Andersen, Fred Gustavson, Alexander Karaivanov, Minka Marinova, Jerzy Waśniewski, and Plamen Yalamov. 2000. LAWRA linear algebra with recursive algorithms. In *International Workshop on Applied Parallel Computing*. Springer, 38–51.

[2] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. 1999. *LAPACK Users' guide*. SIAM.

[3] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2011. Minimizing communication in numerical linear algebra. *SIAM J. Matrix Anal. Appl.* 32, 3 (2011), 866–901.

[4] L Susan Blackford, Jaeyoung Choi, Andy Cleary, Eduardo D'Azevedo, James Demmel, Inderjit Dhillon, Jack Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, et al. 1997. *ScaLAPACK users' guide*. SIAM.

[5] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. 2019. Accelerating reduction and scan using tensor core units. In *Proceedings of the ACM International Conference on Supercomputing*. 46–57.

[6] Eduardo D'Azevedo and Jack Dongarra. 2000. The design and implementation of the parallel out-of-core ScaLAPACK LU, QR, and Cholesky factorization routines. *Concurrency: Practice and Experience* 12, 15 (2000), 1481–1493.

[7] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Ichitaro Yamazaki. 2014. Accelerating numerical dense linear algebra calculations with GPUs. In *Numerical computations with GPUs*. Springer, 3–28.

[8] Jack Dongarra and Piotr Luszczek. 2012. Anatomy of a globally recursive embedded LINPACK benchmark. In *2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 1–6.

[9] Erik Elmroth and Fred G Gustavson. 2000. Applying recursion to serial and parallel QR factorization leads to better performance. *IBM Journal of Research and Development* 44, 4 (2000), 605–624.

[10] Azzam Haidar, Ahmad Abdelfattah, Mawussi Zounon, Panruo Wu, Srikara Pranesh, Stanimire Tomov, and Jack Dongarra. 2018. The design of fast and energy-efficient linear solvers: On the potential of half-precision arithmetic and iterative refinement techniques. In *International Conference on Computational Science*. Springer, 586–600.

[11] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J Higham. 2018. Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 603–613.

[12] Azzam Haidar, Panruo Wu, Stanimire Tomov, and Jack Dongarra. 2017. Investigating half precision arithmetic to accelerate dense linear system solvers. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. 1–8.

[13] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. 2018. Dissecting the nvidia volta gpu architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826* (2018).

[14] Khairul Kabir, Azzam Haidar, Stanimire Tomov, Aurelien Bouteiller, and Jack Dongarra. 2017. A framework for out of memory svd algorithms. In *International Supercomputing Conference*. Springer, 158–178.

[15] Yuechao Lu, Ichitaro Yamazaki, Fumihiko Ino, Yasuyuki Matsushita, Stanimire Tomov, and Jack Dongarra. 2020. Reducing the amount of out-of-core data access for GPU-accelerated randomized SVD. *Concurrency and Computation: Practice and Experience* (2020), e5754.

[16] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. 2018. Nvidia tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 522–531.

[17] Tesla NVIDIA. 2017. NVIDIA Tesla V100 GPU Architecture.

[18] Elmar Peise and Paolo Bientinesi. 2016. Recursive algorithms for dense linear algebra: The relapack collection. *arXiv preprint arXiv:1602.06763* (2016).

[19] Gregorio Quintana-Ortí, Francisco D Igual, Mercedes Marqués, Enrique S Quintana-Ortí, and Robert A Van de Geijn. 2012. A runtime system for programming out-of-core matrix algorithms-by-tiles on multithreaded architectures. *ACM Transactions on Mathematical Software (TOMS)* 38, 4 (2012), 1–25.

[20] Sivan Toledo. 1997. Locality of reference in LU decomposition with partial pivoting. *SIAM J. Matrix Anal. Appl.* 18, 4 (1997), 1065–1081.

[21] Sivan Toledo and Fred G Gustavson. 1996. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations. In *Proceedings of the fourth workshop on I/O in parallel and distributed systems: part of the federated computing research conference.* 28–40.

[22] Sivan Toledo and Eran Rabani. 2002. Very large electronic structure calculations using an out-of-core filter-diagonalization method. *J. Comput. Phys.* 180, 1 (2002), 256–269.

[23] Linnan Wang, Wei Wu, Zenglin Xu, Jianxiong Xiao, and Yi Yang. 2016. Blasx: A high performance level-3 blas library for heterogeneous multi-gpu computing. In *Proceedings of the 2016 International Conference on Supercomputing.* 1–11.

[24] Shaoshuai Zhang, Elaheh Baharlouei, and Panruo Wu. 2020. High accuracy matrix computations on neural engines: A study of qr factorization and its applications. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing.* 17–28.

[25] Shaoshuai Zhang, Vivek Karihaloo, and Panruo Wu. 2020. Basic Linear Algebra Operations on TensorCore GPU. In *2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA).* IEEE, 44–52.