CCF Transactions on High Performance Computing
https://doi.org/10.1007/s42514-021-00075-8

REGULAR PAPER

Implementing LU and Cholesky factorizations on artificial intelligence accelerators

Yuechen Lu[1] · Yuchen Luo[1] · Haocheng Lian[1] · Zhou Jin[1] · Weifeng Liu[1]

Received: 14 April 2021 / Accepted: 3 August 2021

Abstract
LU and Cholesky factorizations for dense matrices are one of the most fundamental building blocks in a number of numerical applications. Because of the $O(n^3)$ complexity, they may be the most time consuming basic kernels in numerical linear algebra. For this reason, accelerating them on a variety of modern parallel processors received much attention. We in this paper implement LU and Cholesky factorizations on novel massively parallel artificial intelligence (AI) accelerators originally developed for deep neural network applications. We explore data parallelism of the matrix factorizations, and exploit neural compute units and on-chip scratchpad memories of modern AI chips for accelerating them. The experimental results show that our various optimization methods bring performance improvements and can provide up to 41.54 and 19.77 GFlop/s performance using single precision data type and 78.37 and 33.85 GFlop/s performance using half precision data type for LU and Cholesky factorizations on a Cambricon AI accelerator, respectively.

Keywords LU factorization · Cholesky factorization · AI accelerator

1 Introduction

LU and Cholesky factorizations are one of the most commonly used matrix operations in solving systems of linear equations (Golub and van Loan 2013). LU factorization decomposes a square matrix A into the multiplication of two matrices L and U, where L is a lower triangular matrix, and U is an upper triangular matrix. Cholesky factorization can be seen as a special form of LU factorization. It decomposes a symmetric positive definite matrix A into LL^T, where L is a lower triangular matrix, and L^T is its transpose.

✉ Weifeng Liu
weifeng.liu@cup.edu.cn

Yuechen Lu
2020211270@student.cup.edu.cn

Yuchen Luo
2020211257@student.cup.edu.cn

Haocheng Lian
2017011344@student.cup.edu.cn

Zhou Jin
jinzhou@cup.edu.cn

[1] Super Scientific Software Laboratory, Department of Computer Science and Technology, China University of Petroleum-Beijing, Beijing, China

Over the past few decades, designing acceleration algorithms and optimization techniques for LU and Cholesky factorizations has received extensive attention (Yamazaki et al. 2015; Haidar et al. 2017; Kurzak et al. 2016; Dorris et al. 2016). There have been a few major approaches to parallelize LU and Cholesky on a variety of parallel hardware architectures. On CPUs and GPUs, panel (Rothberg 1996) and tiling (Dongarra et al. 2014) methods are most used for exploiting parallelism. Besides, batched factorization methods for very small matrices (Haidar et al. 2018; Abdelfattah et al. 2016; Dong et al. 2014) have been developed as well. In addition, mixed precision solvers can use low precision LU or Cholesky for generating initial solution, and iterative refinement for giving high precision solution vector (Yamazaki et al. 2015). Many packages, such as LAPACK (Anderson et al. 1990), ScaLAPACK (Choi et al. 1996a, b), PLASMA (Agullo et al. 2009) and MAGMA (Agullo et al. 2009; Abdelfattah et al. 2017), can provide optimized parallel implementation of LU and Cholesky factorizations.

As the importance of artificial intelligence (AI) increases, building special purpose architectures for AI computations has became a hot topic in recent years (Chen et al. 2020; Reuther et al. 2019, 2020). Representative work such as Diannao family (Chen et al. 2014, 2016) and tensor processing units (Jouppi et al. 2017, 2018) already showed higher

Published online: 24 August 2021

performance and lower energy consumption than general purpose processors, e.g., CPUs and GPUs. Because one of the most important functions of the AI chips is to accelerate matrix-vector and matrix-matrix computations in deep neural networks, they have great potential to accelerate more numerical linear algebra operations. However, to the best of our knowledge, such research opportunity has been largely ignored.

In our work, we utilize BANG C programming language for implementing the two factorization methods and optimize them on a Cambricon AI accelerator. Specifically, we first realize serial algorithm of LU and Cholesky using the Global-DRAM (GDRAM) of the Cambricon AI accelerator. Since GDRAM is an off-chip memory, we use its performance as a baseline of the subsequent optimizations. Then, we migrate the serial code to the on-chip Neural-RAM (NRAM) which has higher read and write throughput, and carry out tensor quantification. After this, we start to use multiple cores of the AI chip to scale out the algorithms. Algorithm-wise, we first treat each row as an independent task and hand it over to a core for execution, but this method in general causes lots of repeated calls to the core function, and degrades performance. So we further improve the parallelism to row block level, which reduces the number of calls of kernel function to a much lower degree. Finally, we carefully tune the size of the row blocks and the number of cores used through a large amount of experimental results.

Our experiments are carried out on an MLU270-S4 AI card. The performance of LU and Cholesky factorizations under different optimization algorithms is tested, and the matrix orders are from 128 to 8192. Compared with the basic serial LU factorization, the performance of the row level parallel algorithm is improved by up to 359.33% than that of the serial algorithm, and the row level parallel algorithm of Cholesky factorization is up to 277.44% better than the serial algorithm. After analyzing the architecture of the AI card, we further optimized the algorithm to a row-block level version scheduling strategy. Compared with the row level versions, the best performance of LU and Cholesky factorizations is further increased up to 235% and 229%, respectively. Finally, our row-block level parallel LU and Cholesky using single precision data type reach 41.54 and 19.77 GFlop/s, and half precision data type reach 78.37 and 33.85 GFlop/s, respectively.

This work makes the following contributions:

- To our knowledge, this a very early work that implements and optimizes LU and Cholesky factorizations on modern AI accelerators.
- The factorization algorithms proposed are optimized according to AI architectures.
- The optimized factorization methods achieve good performance on matrices of various sizes.

2 Background

In this section, we give a background overview of the research. We first introduce the basic LU and Cholesky factorizations and their serial implementation, and then introduce the Cambricon AI architecture and its BANG C development language.

2.1 LU Factorization

LU factorization is used for square matrix decomposition in linear systems. It is a variant of the Gaussian elimination method and belongs to the direct method for linear solvers. LU factorization decomposes a matrix A into the product of a lower triangular matrix L and an upper triangular matrix U by executing the following equations:

$$U_{ij} = A_{ij} - \sum_{k=0}^{j-1} L_{ik} U_{kj} \begin{cases} i = 1, 2, ..., N \\ j = i, i+1, ..., N \end{cases} \tag{1}$$

$$L_{ij} = \frac{1}{U_{jj}} \left(A_{ij} - \sum_{k=1}^{j-1} L_{ik} U_{kj} \right) \begin{cases} i = 1, 2, ..., N \\ j = 1, 2, ..., i-1 \end{cases} \tag{2}$$

For a dense matrix, the time complexity of LU decomposition is $O(n^3)$, where n is the order of the input matrix. The pseudocode in Algorithm 1 shows a basic implementation of LU factorization, and Figure 1 plots the procedure of decomposing a matrix of size 5-by-5.

Fig. 1 Steps of serial LU factorization

Algorithm 1 A pseudocode of LU factorization.

```
for i = 0 : n − 1 do
    for j = i + 1 : n − 1 do
        A_ji = A_ji / A_ii
        for x = i + 1 : n − 1 do
            A_jx = A_jx − A_ji × A_ix
        end for
    end for
end for
```

2.2 Cholesky Factorization

Cholesky factorization can be seen as a special form of the LU factorization. It decomposes a symmetric positive definite matrix A into the product of a lower triangular matrix L and its transpose matrix L^T. Because of using the symmetry of the input matrix, the Cholesky factorization is in general more efficient than LU factorization. It can be computed through

$$L_{ii} = \sqrt{A_{ii} - \sum_{p=1}^{i-1} L_{ip}^2}, i \in [2, n], and \tag{3}$$

$$L_{ji} = \frac{A_{ji} - \sum_{p=1}^{i-1} L_{ip} L_{jp}}{L_{ii}}, i \in [2, n-1], \quad j \in [i+1, n] \tag{4}$$

A serial pseudocode for Cholesky is shown in Algorithm 2.

Algorithm 2 A pseudocode of Cholesky factorization.

```
for i = 0 : n − 1 do
    A_ii = √A_ii
    for j = i + 1 : n − 1 do
        A_ij = A_ij / A_ii
    end for
    for k = i + 1 : n − 1 do
        for j = k : n − 1 do
            A_kj = A_kj − A_ij × A_ik
        end for
    end for
end for
```

2.3 AI Accelerator and BANG C Programming Language

As the growth of AI applications, designing domain specific architectures for accelerating key AI kernels such as matrix multiplication and convolution computations received much attention. Besides running multiple parallel threads, which is very similar to the multi- and many-core processors, the AI chips can also take some specific hardware-level optimizations for neural network compute units and memory systems. Also, the cores with fixed function units can be simpler than the regular cores in general purpose processors.

Cambricon has developed a series of machine learning processors called MLU (Machine Learning Unit) to achieve good trade-off between flexibility and efficiency. A card named MLU270 is a representative accelerator of this series. Figure 2 shows a block diagram of the MLU270 AI card. As can be seen, each MLU270 has four compute clusters, and every cluster has four physical cores. Each core is mainly composed of a functional unit (FU), a general register group (GPR), a neuron storage unit NRAM, and a weight storage unit (Weight–RAM, WRAM). The Shared-RAM (SRAM) on the chip is shared by the four cores in the same cluster. In addition, all cores can access global shared memory called GDRAM using DDR technology, and each core has a separate piece of memory called Local-DRAMs (LDRAMs).

BANG C language is a new language proposed by Cambricon for programming MLU hardware. It brings general purpose programming capabilities to Cambricon chips and increases the freedom of user programming. Bang C language provides a wealth of call APIs for vector function units (VFU) and matrix function units (MFU), through which computing performance can be greatly improved. For example, using the vector function unit API (__bang_add) instead of the vector addition implemented using the ordinary for loop, the performance may have a gap of hundreds of times.

BANG C can support half precision data type of two bytes. In the experiments of this paper, we improve the performance of the program by using half and float type, and the loss of accuracy may be endurable in mixed precision linear algebra routines (Yamazaki et al. 2015).

3 LU and Cholesky Factorizations on AI Chips

3.1 Baseline Implementation

We first implement the serial LU and Cholesky factorization algorithms based on Gaussian elimination to the Cambricon chip. At this time, the input matrix is loaded without any preprocessing, and the data is stored in the global memory GDRAM. The entire calculation process does not take advantage of the superiority of the MLU hardware architecture at all. So the execution time can be pretty long for large matrices. Because all operations at this time are executed by one core, its load is too heavy to finish in a reasonably time. In order to improve performance, and also to handle larger-scale matrices, it is very necessary to optimize the algorithm based the architecture advantages of the MLU270 device.

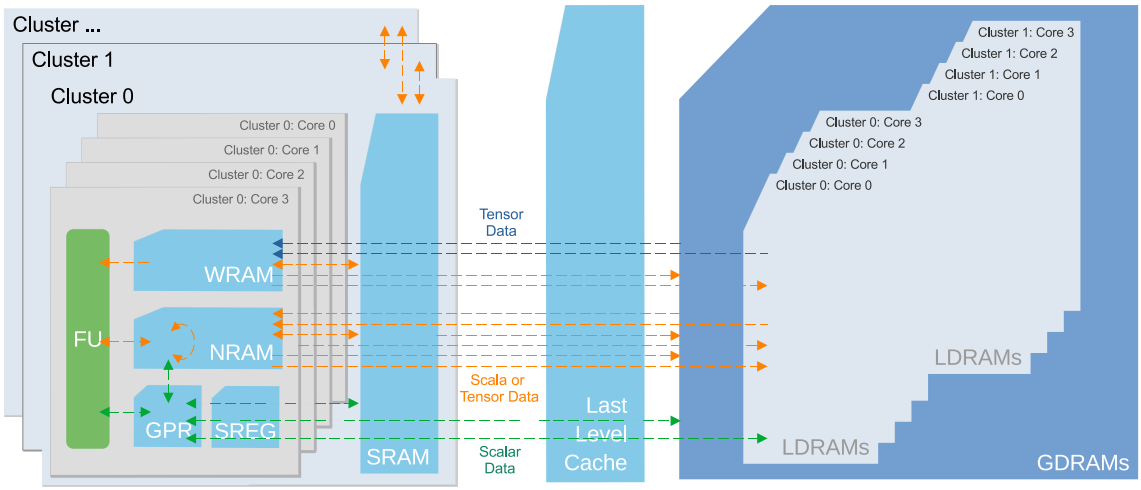


Fig. 2 Block diagram of Cambricon MLU270 AI accelerator

3.2 Optimizations Using On-Chip Memories

On the Cambricon chip, on-chip memory is the storage unit closer to the compute unit, and it also has better efficiency in reading and writing. Therefore, we used NRAM instead of GDRAM when performing matrix factorization operations to increase the speed of the program. NRAM is a shared memory on each core. Although it has a smaller space than GDRAM, it can achieve higher read and write bandwidth and lower access latency. In actual operation, we first apply for a piece of NRAM space on the MLU terminal, then move the GDRAM data to the NRAM, and move the NRAM data back to the GDRAM after the calculation is completed. Because the space on NRAM is relatively small, when we are dealing with larger-scale matrices, we need to copy the data in batches.

Algorithm 3 Cholesky Factorization Algorithm Using Tensor Quantization

Input: A : unfactorized matrix, n : integer
Output: A : factorized matrix

function CNRTINVOKEKERNEL(A, n)
  $nAii$[32] : NRAM float
  $nAij$[32] : NRAM float
  $nAik$[8192] : NRAM float
  $nAjk$[8192] : NRAM float
  $nAjsum$[8192] : NRAM float
  for $i = 0 : n - 1$ **do**
    $A_{ii} = \text{sqrtf}(A_{ii})$
    if $i = n - 1$ **then**
      return
    end if
    memcpy($nAik$, A_{ii} , $n - i - 1$, GDRAM2NRAM) // $nAik \leftarrow A_{ii} : A_{in}$
    memset($nAii$, $1/A_{ii}$) // $nAii \leftarrow 1/A_{ii}$
    __bang_cycle_mul($nAik$, $nAik$, $nAii$, 8192, 32)
    memcpy(A_{ii} , $nAik$, $n - i - 1$, NRAM2GDRAM) // $A_{ii} : A_{in} \leftarrow nAik$
    for $j = i + 1 : n - 1$ **do**
      memset($nAij$, A_{ij}) // $nAij \leftarrow A_{ij}$
      memcpy($nAjk$, A_{jj} , $n - j$, GDRAM2NRAM) // $nAjk \leftarrow A_{jj} : A_{jn}$
      __bang_cycle_mul($nAjsum$, $nAik + j - i - 1$, $nAij$, 8192, 32)
      __bang_sub($nAjk$, $nAjk$, $nAjsum$, 8192)
      memcpy(A_{jj} , $nAjk$, $n - j$, NRAM2GDRAM) // $A_{jj} : A_{jn} \leftarrow nAjk$
    end for
  end for
end function

3.3 Optimizations Using Tensor Quantization

In the LU and Cholesky factorization algorithms, there are many scalar multiplication of vectors structures (Jia et al. 2012; Kurzak et al. 2012). When processing this part, we need to perform the scalar multiplication operations repeatedly, which takes a long time. In the MLU architecture, there are arithmetic modules VFU (Vector Function Unit) and MFU (Matrix Function Unit) dedicated to tensor calculation on each core, which are used to complete vector operations and matrix operations, respectively. Bang C language provides developers with interface for tensor quantization calculation. Using this interface, a large number of scalar multiplication calculations are combined into tensor calculations, and the hardware tensor calculation unit can be better utilized. In general, it also improves the execution time of the program. In BANG C language, the data calculated using tensor quantization should be stored in NRAM.

Algorithm 3 is the Cholesky factorization algorithm of tensor quantization. We directly call the kernel function and transfer the entire matrix and matrix size to the MLU chip. Here we introduce in detail the operation steps of using on-chip storage NRAM and tensor quantization: first, applying for NRAM space (lines 2-5), and then entering the for loop (line 7) to start the update of each layer. In the update of each layer, the first step is to calculate the value A_{ii} on the diagonal, and then copy this value and the value of the entire row to NRAM; the second step is to use the tensor quantization interface provided by BANG C __bang_cycle_mul to perform vector multiplication calculation and update the value of the current row, and then return it to the matrix A; the third step is to enter the for loop (line 16), using the tensor quantization calculation interface and the addition and subtraction function to update the remaining rows, and finally returns the result to the matrix A. Until each layer is updated in turn, the factorization of the entire matrix is completed.

3.4 Optimizations Using Row Level Parallelism

In the previous subsection, the factorization task of a matrix is performed by an MLU core, and its efficiency was relatively low. MLU270 is a multi-core heterogeneous platform for acceleration. Using this feature, the performance of the factorization algorithm can be greatly improved. Taking the LU factorization algorithm as an example, because the current updating layer are dependent on the results of the previous layer, this determines that only the parallelism of the current panel update can be improved, and the update between panels needs to be executed sequentially. We use the "row parallel" approach to update each row as a task. When the current panel is updating all rows, all tasks are executed in parallel to improve the efficiency of the algorithm.

In row parallel LU factorization, we have implemented two algorithms: (1) updating the column vector and the panel separately (NRS) and (2) updating the column vector and panel combined (NRC). The former assigns column vector update and panel update to two kernel functions to execute respectively, while the latter combines the two steps into one kernel function. Note that the Cholesky factorization algorithm has one more dependency than the LU factorization algorithm. Thus, there is no such way of merge execution in Cholesky's row parallel algorithm.

3.5 Optimizations Using Row-Block Level Parallelism

To further improve the performance, we propose a new parallel pattern to complete LU and Cholesky factorizations. The main idea is to divide the currently updated panel equally by rows to obtain row blocks, and each row block is handed over to a core for calculation. All row blocks are calculated in parallel. After the calculation is completed, the results are combined and returned to the matrix to achieve the current panel update work. Compared with the algorithms proposed in this paper, although row-block level parallelism reduces the degree of parallelism, it actually reduces the overhead of repeatedly calling kernel functions.

Algorithm 4 LU Factorization Algorithm of Row-Block-Level Parallelism

Input: A : unfactorized matrix, n : integer, workingKernelNum : integer
Output: A : factorized matrix

```
1: for i = 0 : n − 1 do
2:     var rowBlockSz : integer
3:     rowBlockSz = (n − i − 1)/workingKernelNum
4:     function CNRTINVOKEKERNEL(id = 0 : workingKernelNum)
5:         nAji[32] : NRAM float
6:         tmp[32] : NRAM float
7:         nAjx[8192] : NRAM float
8:         nAix[8192] : NRAM float
9:         memcpy(nAix,A_{i(i+1)},n-i-1,GDRAM2NRAM)// nAix ← A_{i(i+1)} : A_in
10:        for k = i : i + rowBlockSz do
11:            var j : integer
12:            j = id × rowBlockSz + k + 1
13:            if j >= n then
14:                return
15:            end if
16:            A_ji = A_ji / A_ii
17:            memset(nAji , A_ji)        // nAji ← A_ji
18:            memcpy(nAjx,A_{j(i+1)},n-i-1,GDRAM2NRAM)// nAjx ← A_{j(i+1)} : A_jn
19:            __bang_cycle_mul(tmp, nAix, nAji, levelLen(n − i − 1), 32)
20:            __bang_sub(nAjx, nAjx, tmp, levelLen(n − i − 1))
21:            memcpy(A_{j(i+1)},nAjx,n-i-1,NRAM2GDRAM)// A_{j(i+1)} : A_jn ← nAjx
22:        end for
23:    end function
24: end for
```

Algorithm 4 is a row-block parallel algorithm of LU factorization. The for loop (line 1) is running in the host, and the variable *rowBlockSz* represents the number of rows contained in each block of the current layer, and is also the number of rows that each core needs to calculate. Line 4 calls the kernel function to start multi-core parallel computing, and the variable *workingKernelNum* is the number of cores participating in the calculation. The specific operation steps of each core are as follows: First, applying for NRAM spaces to temporarily store the variables used in the current core calculation (lines 5-8), and copying the value of the first row of the current layer to *nAix* (line 9). Then, starting to update the rows of the current core. The first value of the current row is calculated and copied to *nAji*, and then the remaining value of the current row is copied to *nAjx*. Next, using the tensor quantization interface to multiply *nAix* and *nAji*, and temporarily storing the calculated result in *tmp* (line 19). At last, using *nAjx* to subtract *tmp* to update the value of *nAjx*, and finally returning the value of *nAjx* to matrix *A*, which completes the update of the current rows. Until all cores have finished updating all rows, the update operation of the current layer is completed. Through the for loop on the host side, the factorization of the entire LU matrix is completed after all the layers are updated in sequence.

As can be seen in Figure 2, there are four clusters on the MLU270 chip, and four cores on each cluster. So we need to use the best combination of the clusters and cores (e.g., the performance of using 2 clusters × 4 cores and 4 clusters × 2 cores may be different). Because when multiple cores on the same cluster are accessing global memory at the same time, there may be the contention of data transmission channel, which will affect performance to a certain extent. Figure 3 shows the scheduling of computing cores when 8 cores are used for row-block parallel computing. The number of rows to be updated in the current layer is 16. So each core updates two rows. Specially, the red borders represent the part that a core needs to calculate. The color of the row block in the matrix is the same as the color of the core doing the calculation. In this case, eight cores come from four clusters.

In the row-block level parallel algorithm, we have also made some detailed optimizations. When using the tensor quantization interface, one of the parameters needs to be obtained by repeatedly calling a simple function. We definite a macro function which can make the calling part expanded by the macro expander to avoid calling the simple function repeatedly. At the same time, in the compilation process, the use of loop unrolling can also lift the speed.

4 Experimental Results

4.1 Experimental Setup

We used an MLU270-S4 AI card as the experimental platform for testing the performance of LU and Cholesky factorizations with various algorithm implementations. The test matrix sizes are from 128 × 128 to 8192 × 8192. We store the matrices data with float type and half type separately. The

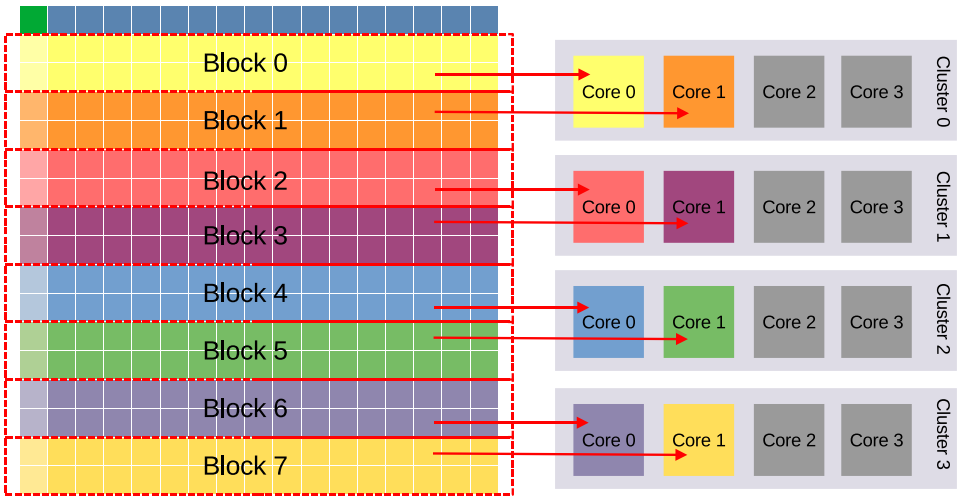
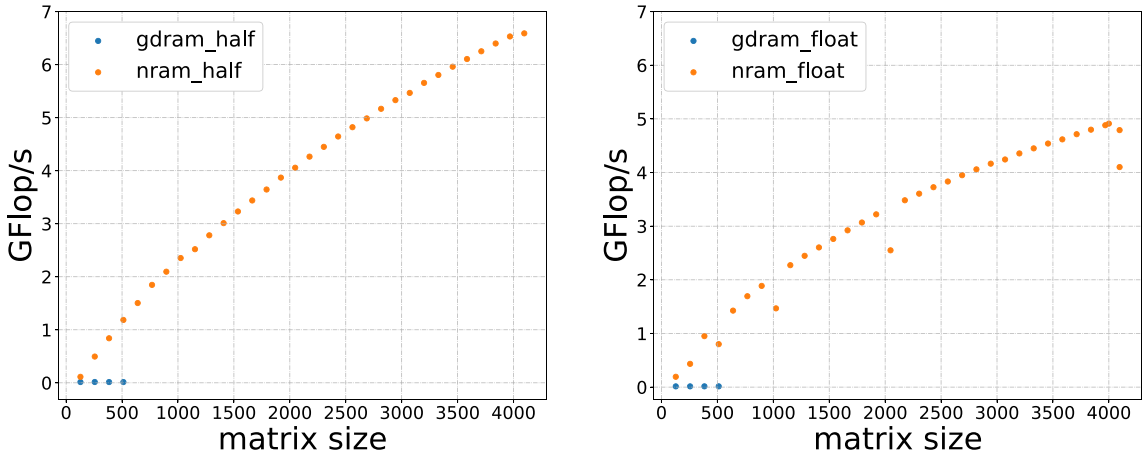
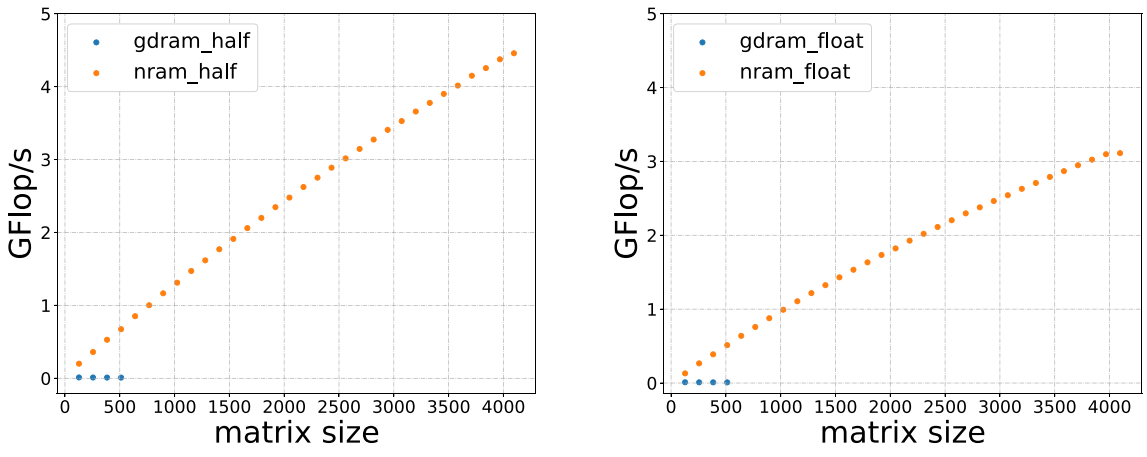


Fig. 3 An example of core allocation in the LU factorization algorithm of row-block level parallelism. The left part indicates that at the current layer, there are 16 rows of data to be updated, and we assume that we use eight cores for calculation. The right part of the figure is the architectural abstraction diagram of MLU270: the eight cores involved in the calculation are evenly distributed among four clusters



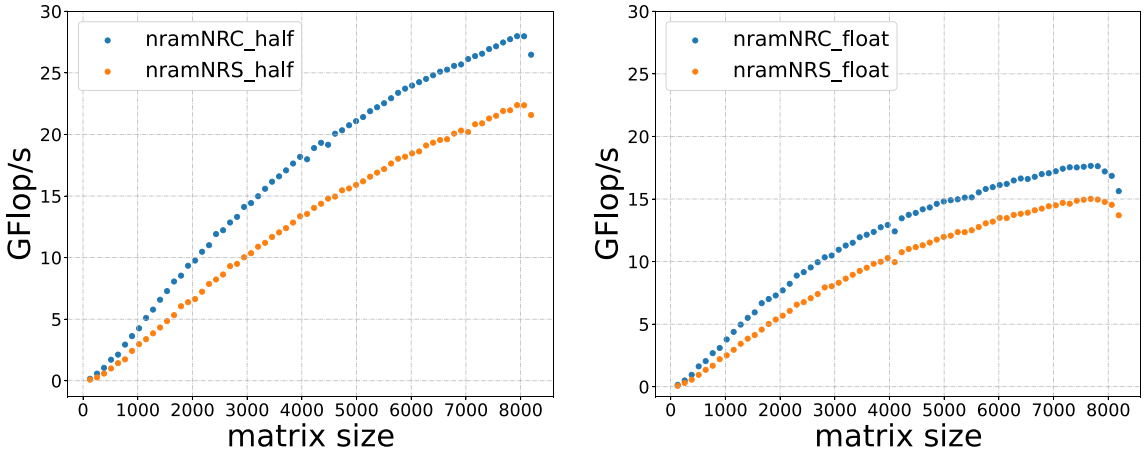
(a) LU in half precision **(b)** LU in single precision

Fig. 4 The performance of LU factorization using GDRAM only and NRAM with tensor quantization, respectively



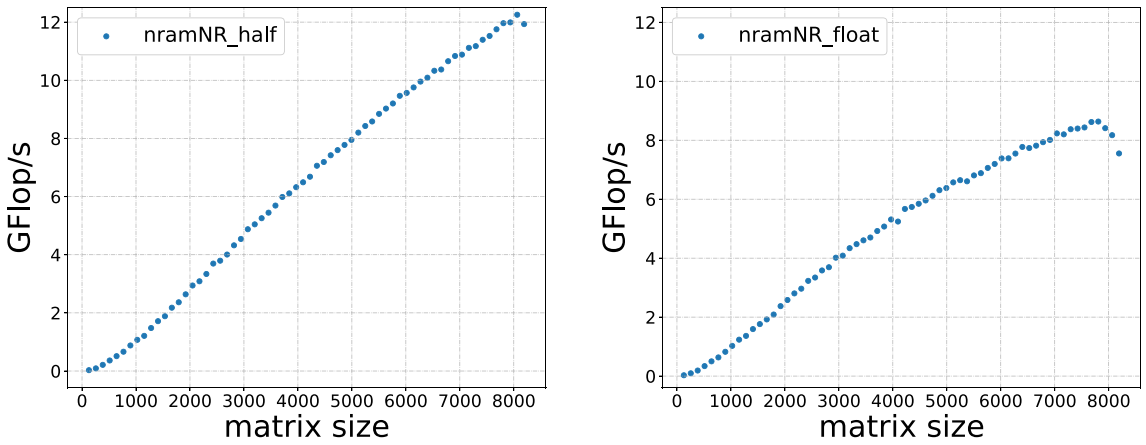
(a) Cholesky in half precision **(b)** Cholesky in single precision

Fig. 5 The performance of Cholesky factorization using GDRAM only and NRAM with tensor quantization, respectively



(a) LU in half precision **(b)** LU in single precision

Fig. 6 The performance of NRC LU factorization and NRS LU factorization with row level parallelism, which (**a**) uses half data type and (**b**) uses float data type



(a) Cholesky in half precision **(b)** Cholesky in single precision

Fig. 7 The performance of Cholesky factorization with row level parallelism, which (**a**) uses half data type and (**b**) uses float data type

following parts will show the performance of our algorithms and analyze their performance.

When using GDRAM, because its MLU270 single-core scalar computing power is very weak, it can only support calculations of up to 600×600 matrix in our test. Also, when using NRAM with tensor quantization, we use the VFU inside the core. It can be calculated for larger scale matrices of size up to 4096×4096.

4.2 Performance of Tensor Quantization

We firstly implement the serial Cholesky and LU factorization algorithms on the off-chip memory GDRAM. We migrate the serial code to the on-chip storage NRAM with higher read and write throughput, and carry out tensor quantization. We test the two algorithms using the two data types separately, the performance comparison charts are shown in Figures 4 and 5.

As can be seen, the performance of the tensor quantization code on NRAM is much better than that of serial code on GDRAM. The method on the GDRAM in the figures only shows the performance of the matrix size of up to 600. This is because when the matrix size is larger than 600, the program will end with time out. As for the code performance on NRAM, the maximum display matrix size is 4096×4096. When the matrix size gets larger, the situation of time out will also occur. Both of these cases are because the serial algorithms take too long time on one core. From these figures, we can see that for the code on NRAM, as the matrix size increases, performance shows an scalable trend.

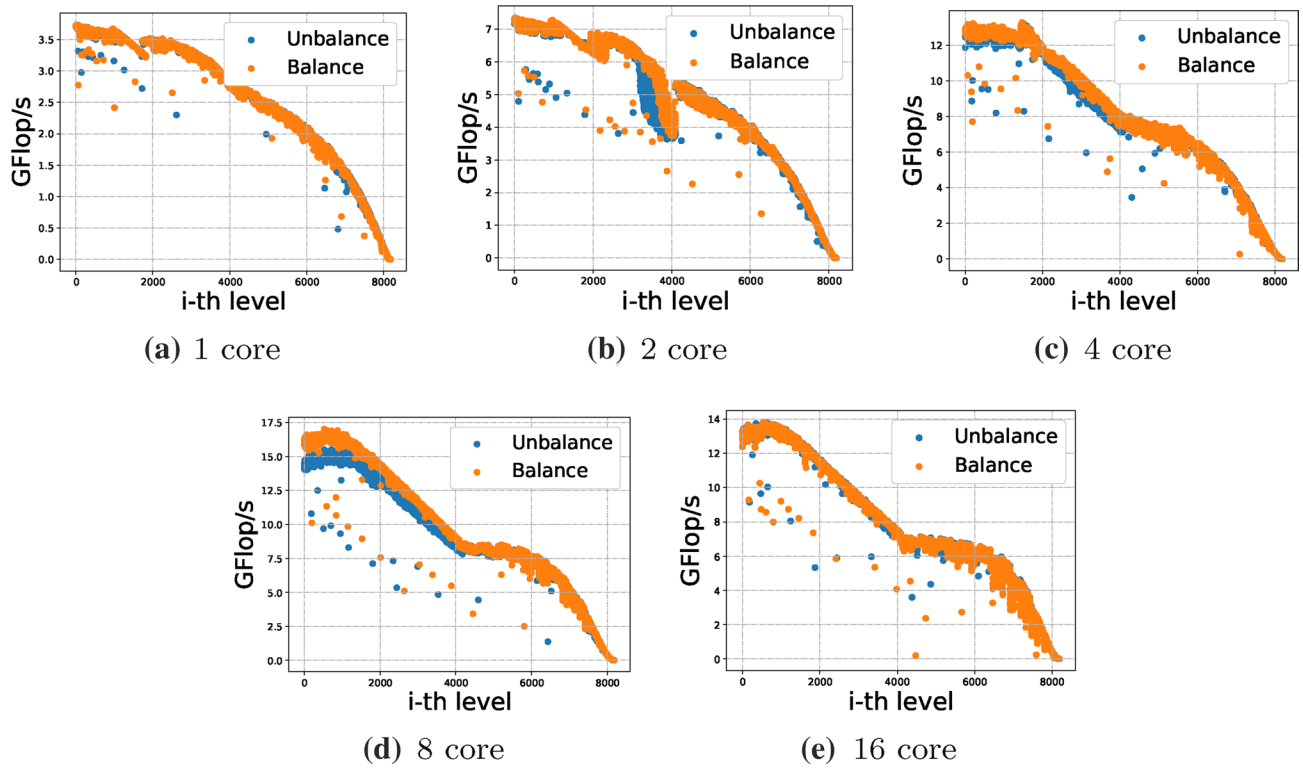


Fig. 8 The performance of calling clusters with unbalanced strategy and balanced strategy

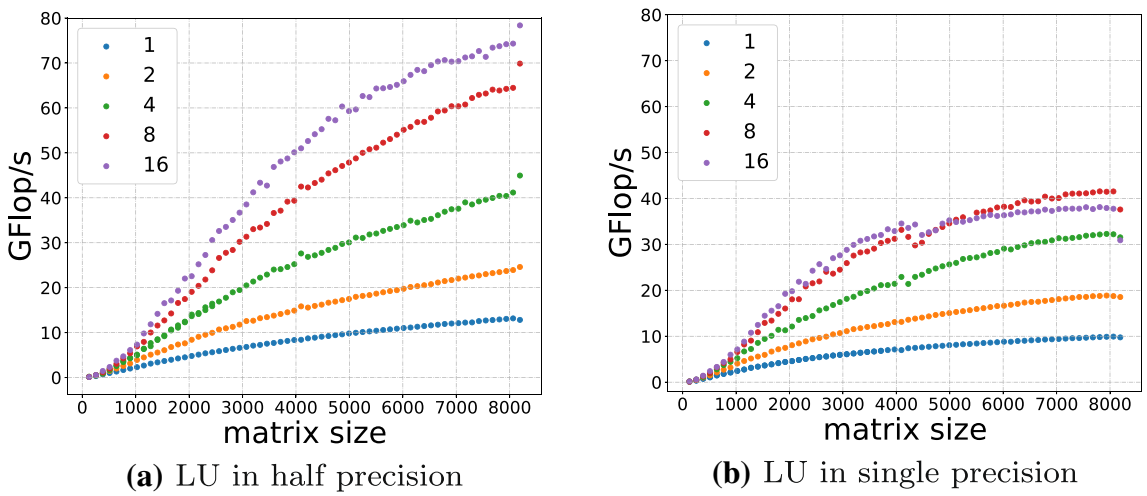
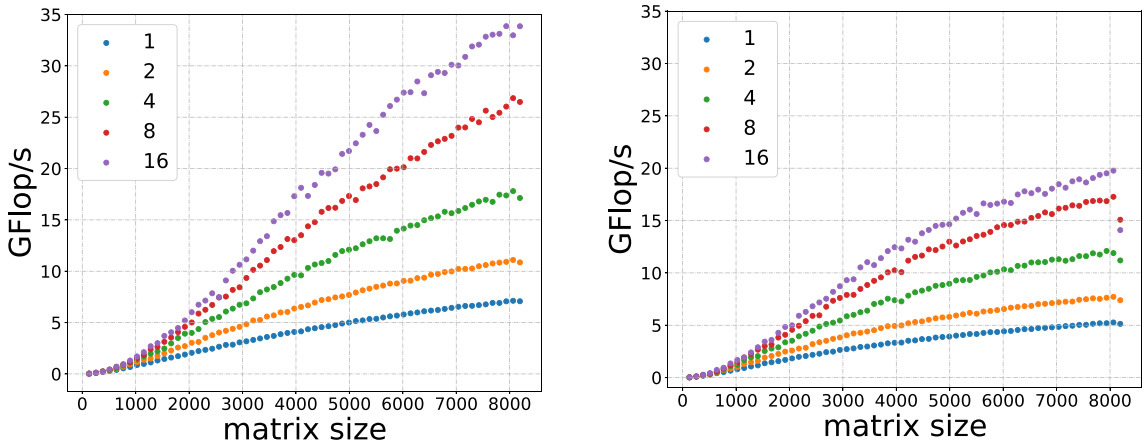


Fig. 9 The best performance of LU factorization with row-block level parallelism

In these figures, the peak performance of LU factorization on GDRAM reaches up to 0.02 GFlop/s, the peak performance of Cholesky factorization reaches 0.01 GFlop/s. The peak performance of LU facorization on NRAM after tensor quantization reaches 4.91 GFlop/s using float type and 6.59 GFlop/s using half type, and Cholesky peak performance reaches 3.11 GFlop/s using float type and 4.46 GFlop/s usign half type. From these data, we can get the code with tensor quantization achieved a preliminary performance improvement. Also, the performance of LU and Cholesky factorization using half data type is 134.15% and 143.22% higher than float data type, respectively.



(a) Cholesky in half precision

(b) Cholesky in single precision

Fig. 10 The best performance of Cholesky factorization with row-block level parallelism

4.3 Performance of Row Level Parallelism

In section 3.4, we proposed two LU factorization algorithms named NRS and NRC. In the experiment, we tested the two algorithms, and the results showed in Figure 6 match our prediction. The performance of the two LU algorithms reaches the best performance at a matrix size of 7936×7936 when using half type and 7800×7800 when using float type. The maximum performances of NRC with the two data type reach 27.98 and 17.65 GFlop/s, the maximum performance of NRS with the two data type reach 2.38 and 15.01 GFlop/s. The former avoids repeated copying of data from GDRAM to NRAM and reduces the number of calls to MLU functions, so the performance of NRC is better than that of NRS. Also, changing from float type to half type can improve performance by about 1.5 times.

Similarly, we also test the row level parallel algorithm of Cholesky factorization, and get performances as shown in Figure 7. Comparing the two parallel factorization algorithms with the two serial factorization algorithms, it can be obtained that the performance of LU row level parallel algorithm is up to 359.33% higher than the serial algorithm, and the performance of Cholesky factorization row level parallel algorithm is up to 277.44% higher than the serial algorithm.

4.4 Performance of Row-Block Level Parallelism

We compare the two scheduling strategies of calling cluster normally (clusters load unbalanced) and calling more clusters preferentially (clusters load balanced) for LU factorization on a matrix with a size of 8192×8192. In Figure 8, the blue dots are the performance when the unbalanced scheduling strategy updating each layer of panel, and the yellow dots are the performance of the balanced scheduling strategy. When the numbers of cores are 1 and 16, there is almost no difference in performance between the two; when the numbers of cores are 2, 4, and 8, different numbers of clusters bring significant different performance. In particular, when using eight cores, balanced strategy can be 2 GFlop/s higher than unbalanced strategy.

As for the scaling test, we measure LU and Cholesky with matrices of size $(128 \times 128 - 8192 \times 8192)$ using two data types, and obtained the performance in Figures 9 and 10. Except the LU factorization algorithm with float data type, the performances of other algorithms are completely positively correlated with the number of cores. In the Figure 9(b), when the matrix size exceeds 5000×5000, the performance of eight cores exceeds the performance of 16 cores. This is because that when the matrix size becomes larger, the cluster's internal cores copy more data from GDRAM at the same time, causing data transmission congestion. So reducing the number of active cores in the cluster at this time can further improve performance. Like Figure 9, the performance of row-block level parallel LU algorithm with half data type is 2.54 times higher than float data type. Using the half type row block parallel LU algorithm to factorize the 8192×8192 matrix on 16 cores can achieve the highest performance 78.37 GFlop/s and using float type gets the best performance 41.53 GFlop/s when using eight cores. As for Cholesky, the performance is always positively correlated with the number of cores. This is because that of the memory copy required for Cholesky factorization is much less than LU. Using the half type and float type row block parallel Cholesky algorithm to factorize the 8192×8192 matrix on 16 cores can achieve up to 33.85 and 19.77 GFlop/s, respectively.

Moreover, we especially try to decompose a matrix of 16384×16384 with Cholesky. The 8-core performance with

float type is 13.89 GFlop/s, and the 16-core performance is 12.32 GFlop/s, which are lower than that of 8192×8192 size.

5 Conclusion

To the best of our knowledge, this is the first work that has implemented LU and Cholesky factorizations on modern AI chips, and developed a series of optimization techniques for utilizing the specific architectures originally designed for deep neural network computations. The experimental results show that LU and Cholesky factorizations can obtain 78.37 and 33.85 GFlop/s with half data type, 41.53 and 19.77 GFlop/s with float data type on a Cambrican AI chip, respectively, and a variety of optimizations demonstrated their effectiveness.

Acknowledgements We would like to thank the invaluable comments from all the reviewers. Weifeng Liu is the corresponding author of this paper. This research was supported by the National Natural Science Foundation of China under Grant No. 61972415, and the Science Foundation of China University of Petroleum, Beijing under Grant Nos. 2462019YJRC004, 2462020XKJS03.

References

Abdelfattah, A., Haidar, A., Tomov, S., Dongarra, J. J.: "Performance tuning and optimization techniques of fixed and variable size batched cholesky factorization on gpus," In *International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA*, ser. Procedia Computer Science, M. Connolly, Ed., vol. 80. Elsevier, (2016), pp. 119–130

Abdelfattah, A., Haidar, A., Tomov, S., Dongarra, J.J.: Fast Cholesky factorization on GPUs for batch and native modes in MAGMA. J. Comput. Sci. **20**, 85–93 (2017)

Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S.: Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects. J. Phys. Conf. Ser. **180**, 012037 (2009)

Anderson, E., Bai, Z., Dongarra, J., Greenbaum, A., McKenney, A., Du Croz, J., Hammarling, S., Demmel, J., Bischof, C., Sorensen, D.: In: Lapack: a portable linear algebra library for high-performance computers, pp. 2–11. IEEE Computer Society Press, Washington, DC, USA (1990)

Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., Temam, O.: Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. Presented at the (2014)

Chen, Y., Chen, T., Xu, Z., Sun, N., Temam, O.: Diannao family: energy-efficient hardware accelerators for machine learning. Commun. ACM **59**(11), 105–112 (2016)

Chen, Y., Xie, Y., Song, L., Chen, F., Tang, T.: A survey of accelerator architectures for deep neural networks. Engineering **6**(3), 264–274 (2020)

Choi, J., Demmel, J., Dhillon, I., Dongarra, J., Ostrouchov, S., Petitet, A., Stanley, K., Walker, D., Whaley, R.: "Scalapack: a portable linear algebra library for distributed memory computers – design issues and performance," *Computer Physics Communications*, vol. 97, no. 1, pp. 1–15, (1996), high-Performance Computing in Science

Choi, J., Dongarra, J.J., Ostrouchov, S., Petitet, A., Walker, D.W., Whaley, R.C.: Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. Sci. Program. **5**(3), 173–184 (1996b)

Dong, T., Haidar, A., Luszczek, P., Harris, J. A., Tomov, S., Dongarra, J. J.: "LU factorization of small matrices: Accelerating batched DGETRF on the GPU," In *2014 IEEE International Conference on High Performance Computing and Communications, 6th IEEE International Symposium on Cyberspace Safety and Security, 11th IEEE International Conference on Embedded Software and Systems, HPCC/CSS/ICESS 2014, Paris, France, August 20-22, 2014*. IEEE, (2014), pp. 157–160

Dongarra, J.J., Faverge, M., Ltaief, H., Luszczek, P.: Achieving numerical accuracy and high performance using recursive tile LU factorization with partial pivoting. Concurr. Comput. Pract. Exp. **26**(7), 1408–1431 (2014)

Dorris, J., Kurzak, J., Luszczek, P., YarKhan, A., Dongarra, J. J.: "Task-based cholesky decomposition on knights corner using openmp," In *High Performance Computing - ISC High Performance 2016 International Workshops, ExaComm, E-MuCoCoS, HPC-IODC, IXPUG, IWOPH, P 3A, VHPC, WOPSSS, Frankfurt, Germany, June 19-23, 2016, Revised Selected Papers*, ser. Lecture Notes in Computer Science, M. Taufer, B. Mohr, and J. M. Kunkel, Eds., vol. 9945, (2016), pp. 544–562

Golub, G.H., van Loan, C.F.: Matrix computations, 4th edn. JHU Press, USA (2013)

Haidar, A., Abdelfattah, A., Tomov, S., Dongarra, J. J.: "High-performance cholesky factorization for gpu-only execution," In *Proceedings of the General Purpose GPUs, GPGPU@PPoPP, Austin, TX, USA, February 4-8, 2017*. ACM, (2017), pp. 42–52

Haidar, A., Abdelfattah, A., Zounon, M., Tomov, S., Dongarra, J.J.: A guide for achieving high performance with very small matrices on GPU: a case study of batched LU and Cholesky factorizations. IEEE Trans. Parallel Distrib. Syst. **29**(5), 973–984 (2018)

Jia, Y., Luszczek, P., Dongarra, J. J.: "Multi-gpu implementation of LU factorization," In *Proceedings of the International Conference on Computational Science, ICCS 2012, Omaha, Nebraska, USA, 4-6 June, 2012*, ser. Procedia Computer Science, H. H. Ali, Y. Shi, D. Khazanchi, M. Lees, G. D. van Albada, J. J. Dongarra, and P. M. A. Sloot, Eds., vol. 9. Elsevier, (2012), pp. 106–115

Jouppi, N.P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-L., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T.V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C.R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., Yoon, D.H.: In-datacenter performance analysis of a tensor processing unit. Presented at the (2017)

Jouppi, N.P., Young, C., Patil, N., Patterson, D.: A domain-specific architecture for deep neural networks. Commun. ACM **61**(9), 50–59 (2018)

Kurzak, J., Luszczek, P., Faverge, M., Dongarra, J. J.: "Programming the LU factorization for a multicore system with accelerators," In *High Performance Computing for Computational Science - VECPAR 2012, 10th International Conference, Kobe, Japan, July 17-20, 2012, Revised Selected Papers*, ser. Lecture Notes in

Computer Science, M. J. Daydé, O. Marques, and K. Nakajima, Eds., vol. 7851. Springer, (2012), pp. 28–35

Kurzak, J., Anzt, H., Gates, M., Dongarra, J.J.: Implementation and tuning of batched Cholesky factorization and solve for NVIDIA GPUs. IEEE Trans. Parallel Distrib. Syst. **27**(7), 2036–2048 (2016)

Reuther, A., Michaleas, P., Jones, M., Gadepally, V., Samsi, S., Kepner, J.: "Survey and benchmarking of machine learning accelerators," In. IEEE High Performance Extreme Computing Conference (HPEC) **2019**, 1–9 (2019)

Reuther, A., Michaleas, P., Jones, M., Gadepally, V., Samsi, S., Kepner, J.: "Survey of machine learning accelerators," In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, (2020), pp. 1–12

Rothberg, E.: Performance of panel and block approaches to sparse Cholesky factorization on the ipsc/860 and paragon multicomputers. SIAM J. Sci. Comput. **17**(3), 699–713 (1996)

Yamazaki, I., Tomov, S., Dongarra, J.: Mixed-precision Cholesky QR factorization and its case studies on multicore CPU with multiple GPUs. SIAM J. Sci. Comput. **37**(3), C307–C330 (2015)