CUDA Implementation of Batched Cholesky Factorization of Small Matrices with Interleaved (Simple and Chunked) Layout

High-Performance Parallel Programming Term Project Spring 2020-21
Indian Institute of Technology Kharagpur

Group VI

18AE30008 Kshitij Anand

18EE10012 Arghyadeep Bandyopadhyay

18ME30058 Suman Dutta

18QE30002 Shubhesh Anand

18QE30008 Divyanshu Sheth

19MA20004 Anil Kumar Yogi

19MA20030 Nishant Goyal

19MA20033 Parv Maheshwari

Motivation

The paradigm of parallel programming promises significantly lesser total processing and computation time when compared to serial programming on large data sets. The difference in processing time diminishes as the size of the data set reduces. The reason for this phenomenon primarily lies in the overheads associated with the steps involved in setting up the parallel processing environment as well as in the manners of memory exchange. For instance, the NVIDIA GTX series graphics card while copying data to the shared memory of a streaming multiprocessor performs I/O access from the GPU Global Memory (GPU DRAM) in sets of 128 bytes at once. Hence, for data smaller than 128 bytes or for data stored non-contiguously at gaps wider than 128 bytes results in similar or degraded performance when compared to serial processing. For the concern of our project, we will be focusing on factorizing smaller matrices of size lesser than 100 x 100 consisting of single-precision floating-point numbers. Much linear algebra software has shown high efficiency for solving large linear systems on GPU-based computers. However, on the contrary, the performance with regard to smaller linear systems is comparatively poor due to the limited use of parallelism. With an aim to solve this challenge, we present a large set of smaller linear systems to be processed via batch implementation and allows

for better usage of the parallel hardware. Numerous applications deal with large sets of small linear solves that call for batch processing on GPUs such as finite element methods, computational lithography, and collaborative filtering, to name a few.

Problem Definition

Batch matrix operations address the case of solving the same linear algebra problem for a very large number of very small matrices. A popular linear algebra routine is the Cholesky factorization which decomposes a matrix into the product of a lower triangular matrix and its conjugate transpose. The linear algebra routine is required typically for solving a system of linear equations. The objective of this project is to implement a batch Cholesky factorization in CUDA using single-precision arithmetic following the optimizations outlined in the given reference paper.

1. Develop an efficient tiled implementation for obtaining the Cholesky factorization of a single matrix.
2. Develop efficient batched implementations of the Cholesky factorization for a set of small matrices following the two data layouts outlined in the paper (simple interleaved and interleaved chunked).
3. The algorithms outlined in the paper present a set of tunable parameters -
    a. Tiling
    b. Order of Looking
    c. Chunk Size (for the interleaved chunked layout)
    d. Unrolling

Conduct extensive experimental results by varying the first three parameters for a set of matrices and determine which set of parameters proves to be ideal for executing the algorithm. Consider the total number of matrices to be a power of 2 starting from 1024 and going up to the maximum number of matrices you can fit in the GPU memory. Consider the size of each matrix in the set to be in the range {20,30,40,...,100}. Present execution time statistics for all experimental observations and discuss your findings in detail.

Solution Method

Generation of Dataset

A matrix A is said to be symmetric positive definite, if and only if $a_{ij} = a_{ji}$ for all $i,j \in [1,n]$ where n denotes the size of the matrix, and all the eigenvalues are positive. In other words, a matrix A is said to be symmetric positive definite if $\boldsymbol{x^T Ax} > 0$ for all vectors $x \neq 0$.

MATLAB was used to generate the dataset matrices. The following steps generate a symmetric positive definite matrix:

```
%Generate a matrix containing zeros
A = zeros(dim);
%generate a symmetric matrix
A = randi(dim,dim);
A = 0.5*(A+A');
%Construction and a symmetric diagonally dominant matrix
%is symmetric positive definite, which can be ensured by adding nI
A = A + dim*eye(dim);
%Matrix Generated
```

The matrices were obtained for various dimensions and the number of matrices obtained was in increasing powers of 2 starting from 32 and going up to 131072 (2^{17}) - the maximum that can fit on a 2 GB GPU Memory. The method used for the generation of matrices is ***O(n^2)*** and consumed more than 3 hours to generate the biggest file to fit the GPU memory. Hence, for smaller files MATLAB can be used, however, we strongly suggest using Python for the generation of files of higher sizes.

Link of Dataset:
https://drive.google.com/file/d/1hCHJI6G5UZLa2_zZOrg9RIs2T7meQH9j/view?usp=sharing

Auxiliary functions for Part A (Left Looking, Right Looking, and Top Looking Tile implementation)

A symmetric positive definite system of linear equations of the form AX=B can be solved by computing the Cholesky Factorization of Matrix A and expressing it as

$$A=LL^T$$

and then applying forward substitution and backward substitution to vector B using factors L and L^T. Here L is a Lower Triangular matrix.

The tiled implementation of Left, Right and Top Looking versions involves the following 4 main auxiliary functions -

1) **POTRF :** Cholesky factorization
   Prototype for the function in our implementation is -
   __device__ void spotrf_tile(float* t_A)

   input - the pointer to shared memory for a tile given by t_A
   Output - performs Cholesky factorization of the tile in the shared memory itself.

Working principle -

i) We are iterating over each column of the tile matrix using a for loop (marked by variable k in the implementation)

ii) First step is taking the square root for the diagonal element of the k^{th} column followed by the division step in which we divide all the elements in that k^{th} column by the updated diagonal element.

iii) This step involves updating the lower half values of the matrix to the right of the k^{th} column using the updated values of the kth column.

2) **TRSM** : Triangular Solve

Prototype for the function in our implementation is -

__device__ void strsm_tile(float *t_A1, float *t_A2)

Input - inputs are pointers to the shared memory for a column of tiles given by t_A1(for reading purpose) and a single tile t_A2(for write purpose)

Output - perform triangular solve for a tile namely implementing triangular solve on tile t_A2 using t_A1

Working Principle -

i) We are iterating over each column of the tile matrix using a for loop (marked by variable i in the implementation)

ii) In i^{th} iteration we divide each element of the i^{th} column of tile t_A2 with the diagonal element in tile t_A1 i.e. corresponding (i,i) element.

iii) In next step we update the elements of each row in t_A2 as follows

t_A2[threadIdx.y*TILE_SIZE+ty]-=t_A2[threadIdx.y*TILE_SIZE + i]*t_A1[threadIdx.x*TILE_SIZE + i]

3) **SYRK**: symmetric rank-k update

Prototype for the function in our implementation is -

__device__ void syrk_tile(float* rA1, float* rA2)

Input - inputs are pointers to the shared memory for a column of tiles given by rA1(for reading purpose) and a single tile rA2(for write purpose)

Output - performs a rank-k update of the tile rA2 using tile rA1

Working Principle -

i) We extract information about the row and column index of the elements of rA2.

ii) We are iterating over each column of the matrix rA2 using a for loop (marked by variable i in the implementation), with the row and column indices of rA1 being used as the two-row indices of two of the respective elements of rA2.

iii) We multiply these two elements and add them up to a sum variable, and repeat this for the whole i^{th} iteration.

iv) In the end, we subtract the value of the sum from the tile rA2.

4) **GEMM:**

Prototype for the function in our implementation is -

__device__ void sgemm_tile(const float* rA1, const float* rA2, float* rA3)

Output - Function to perform general matrix multiplication

Inputs - pointers to the shared memory for three tiles given by rA1, rA2 and rA3

Working principle -

implementing sgemm on tile rA3 using rA1 and rA2. It basically multiples rA1 tile matrix with rA2 tile and update the values of rA3 as follows

updatedValue -= rA1[row * TILE_SIZE + i] * rA2[i * TILE_SIZE + column];

rA3[row * TILE_SIZE + column] = updatedValue;

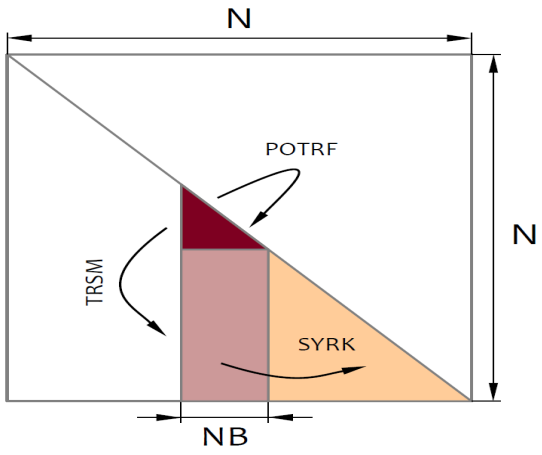
Kernel code for Right Looking tile implementation

Basic Idea: -

```
      do k = 1,N
        // Scale current column
B1(k) : A(k,k) = sqrt(A(k,k))
        do i = k+1,N
          A(i,k) = A(i,k)/A(k,k)

        // Update from current column
        // to columns to right
        do j = k+1,N
B2(k,j) : do i = j,N
            A(i,j) = A(i,j)-A(i,k)*A(j,k)
```

(a) Right-looking Cholesky



Algorithm: -

i. We iterate over all the columns of tiles denoted by the variable i (here i denotes the column number).

ii. We apply PORTF function for that (i^{th}, i^{th}) tile.

iii. For each tile **column** i, we iterate over all the tiles in that column from tile no. j = i+1 upto N/TILE_SIZE - 1 (index being starting from **0**), where j being the row number.

iv. For each j^{th} iteration, we apply TRSM function for that (i^{th}, j^{th}) tile using the complete

data of i^{th} row as the input.

v. For each j^{th} tile, we iterate over all the tiles in that j^{th} row from tile no. k = i+1 upto j, where k being the column number.

vi. For each k^{th} iteration, we apply SYRK function for that (k^{th}, j^{th}) tile using the i^{th} row as the input data.

Algorithm 2 Right looking Cholesky

1: **for** $i = 0$, nb to N **do**
2: $\{$Panel factorize $\mathbf{A_{i:m,i:i+nb}}\}$
3: DPOTF2 $A_{i:i+nb,i:i+nb}$
4: DTRSM $A_{i+nb:m,i:i+nb} = A_{i+nb:m,i:i+nb} \times A^{-1}_{i:i+nb,i:i+nb}$
5: $\{$Update trailing matrix $\mathbf{A_{i+nb:m,i+nb:m}}\}$
6: DSYRK:
$A_{i+nb:m,i+nb:m} = A_{i+nb:m,i+nb:m} - A_{i+nb:m,i:i+nb} \times A^{T}_{i+nb:m,i:i+nb}$
7: **end for**

Kernel Code for Top Looking Tiled Implementation

In top looking Cholesky factorization, instead of factoring the whole panel, only a diagonal triangle of size n_b is factored, and updates to the rows below the triangle as well as updates to the trailing submatrix are deferred. In each step of the algorithm, first, all pending updates from the top of the matrix (which is already factored) are applied to a stripe of the matrix to the left of the diagonal triangle. Then, that stripe is used to update the diagonal triangle. Finally, the diagonal triangle is factored in. This is repeated until the entire matrix is factored in.

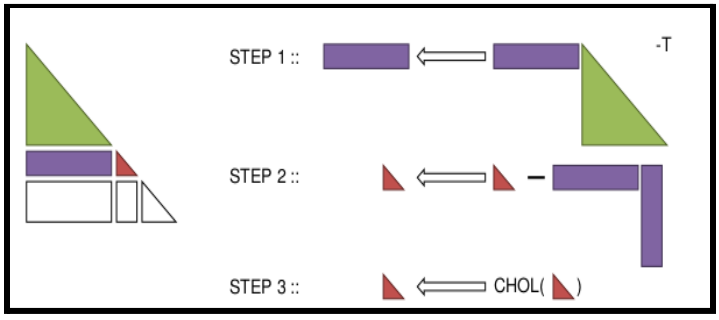


Figure showing the three main operations performed at each step for top looking Cholesky factorization

Algorithm: -

(The original matrix to be factored is stored in global memory)
N = order of the matrix
NB = Tile dimension

1: for k = 0 to ceil(N/NB)-1 do
2: for n = 0 to k-1 do
3: Load tile at kth row and nth column of the tiled matrix in shared memory
4: for m = 0 to n-1 do
5: Load tiles at kth row and mth column, nth row and mth column of the tiled matrix in shared memory
6: Perform a sgemm using these two tiles and update the result in the tile loaded in step 3
7: end for
8: Load the lower triangular half of the tile at nth row and nth column in shared memory
9: Update the tile at kth row and nth column using this by triangular solve
10: Store the updated tile back into the original matrix
11: end for
12: Load the diagonal triangle at kth row and kth column of tiled matrix in shared memory
13: for n = 0 to k-1 do
14: Load tile at kth row and nth column of the tiled matrix in shared memory
15: Update the diagonal triangle using this tile
16: end for
17: Factorize the diagonal triangle
18: Store the diagonal triangle back to the original matrix
19: end for

The operations in the for loop from steps 4 to 7 also the operations in the for loop from step 13 to 16 in the algorithm may be performed in parallel by launching multiple blocks, as these two loops have no data dependency in them. However, this would imply launching kernels from within a kernel. Also, the data which is being used across multiple kernels needed to be stored in global memory, which would have increased the execution time due to global memory accesses. Thus. in order to ensure greater utilization of shared memory, and also to avoid the complexities of kernel launch from within a kernel, all the operations were performed in a single kernel in sequence.

Kernel code for Left Looking tile implementation

The basic left-looking approach involves first updating the current column using the column left to the current column and then compute the current column.

Algorithm: -

1: for K = 0 to ceil(N/NB) - 1 do
2: Read the K'th panel from global memory to local memory.
3: __syncthreads();

4: for L = 0 to K - 1 do
5: Read L'th panel from global memory to shared memory.
6: __syncthreads();
7: Apply update from L'th panel to K'th panel.
8: __syncthreads();
9: end for
10: Factor K'th panel using local and shared memory.
11: __syncthreads();
12: Save K'th panel from shared memory to global memory.
13: __syncthreads();
14: end for

Algorithm 1 LL Cholesky

1: **for** $i = 0, nb$ to N **do**
2: **if** (i > 0) **then**
3: {Update current panel $\mathbf{A_{i:m,i:i+nb}}$}
4: DSYRK:
$A_{i:i+nb,i:i+nb} = A_{i:i+nb,i:i+nb} - A_{i:i+nb,0:i} \times A^{T}_{i:i+nb,0:i}$
5: DGEMM:
$A_{i+nb:m,i:i+nb} = A_{i+nb:m,i:i+nb} - A_{i+nb:m,0:i} \times A^{T}_{i:i+nb,0:i}$
6: **end if**
7: {Panel factorize $\mathbf{A_{i:m,i:i+nb}}$}
8: DPOTF2 $A_{i:i+nb,i:i+nb}$
9: DTRSM $A_{i+nb:m,i:i+nb} = A_{i+nb:m,i:i+nb} \times A^{1}_{i:i+nb,i:i+nb}$
10: **end for**

We observed that during implementation **syrk** and **gemm** had data dependency of the kind that multiple blocks can not be launched parallelly to carry out the parallel execution of **syrk** and **gemm** on the loaded panel. We load the whole panel including the lower half diagonal tile and all full tiles below it (from tile labeled as (i,i) to the last tile in that panel) into the shared memory first, carry out all necessary operations including **syrk, portf, gemm, and trsm** and then store back the complete panel back to global memory so that we have a minimum number of global memory access and all the access of values during execution of auxiliary functions take place from shared memory only. We noticed that we were facing bank conflict in 3 functions primarily **gemm, syrk** and **portf**. We resolved the issue using padding particularly updating the tile dimensions from (TILE_SIZE, TILE_SIZE) to (TILE_SIZE, TILE_SIZE +1) and we found that

this further boosted the performance (faster execution) of the overall Left looking implementation.

Part B: Implementing Efficient Batched Cholesky Factorization

The first important step towards starting the program was reading the input files (filled with matrices in row-major form) into arrays in the single interleaved and interleaved chunked layout.

Pseudocode to read input files into the input host side array **h_A** in a **single interleaved format.**

```
for i = 0 to N - 1 do
        for r = 0 to M - 1 do
                for c = 0 to M -1 do
                        readFromFile float -> temp;
                        global_index = r * M * N + c * N + i;
                        h_A[global_index] = temp;
                end for
        end for
end for
```

The iteration index i denotes the i^{th} matrix we are reading. N denotes the number of matrices in the input file and M denotes the dimension of the matrices present in the input files. The variable *global_index* is computed in a manner such that the array h_A stores all the elements of all the matrices in a simple interleaved format.

Pseudocode to read input files into the input host side array **h_A** in an **interleaved chunked format.** Here we use the variable ***chunk_size*** to denote the chunk size that we will be using in this section.

```
for p = 0 to ceil(N/chunk_size) do
        for i = 0 to chunk_size - 1 do
                for r = 0 to M - 1 do
                        for c = 0 to M -1 do
                                readFromFile float -> temp;
                                x = r*M + c;
                                global_index = (p*chunk_size) * M * M + x*chunk_size + i;
                                h_A[global_index] = temp;
                        end for
                end for
        end for
end for
```

Then we switched the Block Dimensions of the kernel launch from (TILE_SIZE, TILE_SIZE) which were used while testing part A code to (num_matrices_per_block, TILE_SIZE, TILE_SIZE) where num_matrices_per_block denotes the number of matrices given to per block which is calculated by the following in the following input format cases-

I. Simple Interleaved Format:
num_matrices_per_block = min(1024/(TILE_SIZE * TILE_SIZE), num_of_matrices);

II. Chunked Interleaved Format:
Num_matrices_per_block=min(min(1024/(TILE_SIZE*TILE_SIZE), num_of_matrices), chunk_size);

This was done to incorporate multiple matrices being passed to the kernel as all original Part A codes were written for a single input matrix. Here it can be observed that the x dimension is used for the matrix index, y - dimension for the row, and the z dimension for the column. These dimensions.

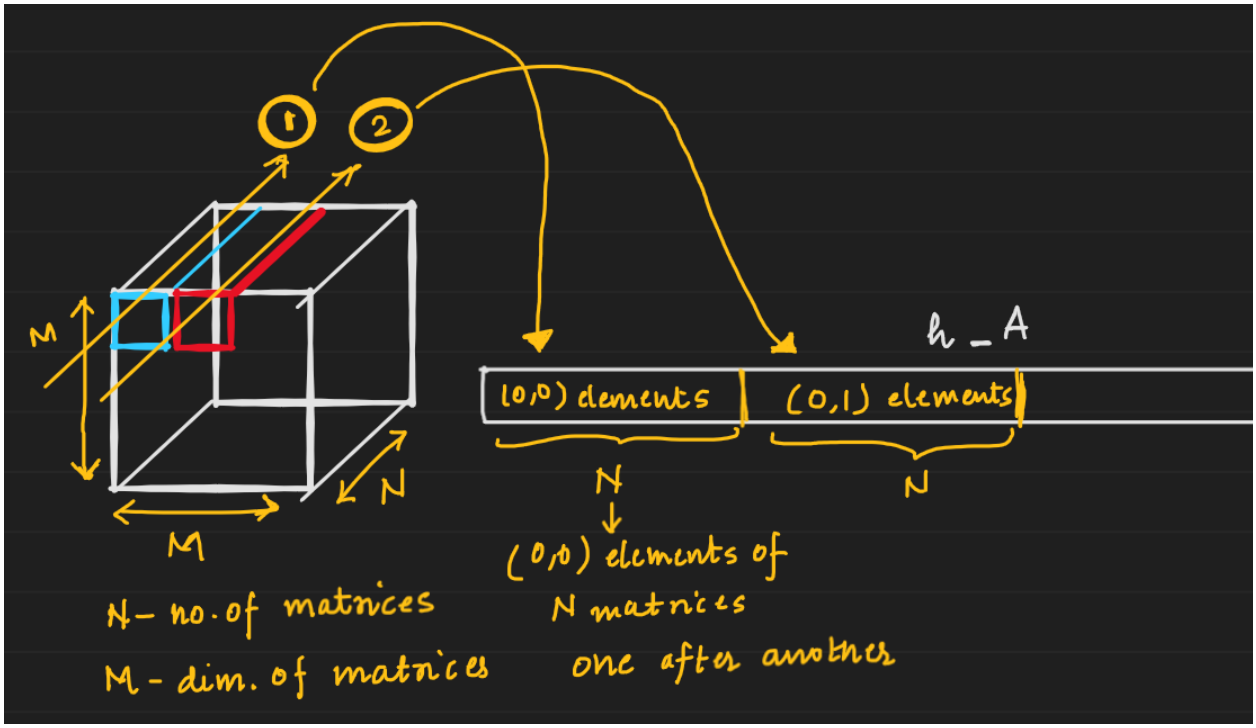
To incorporate these dimension changes, all the dimensions and axes in the auxiliary functions also were accordingly changed.

The next step was to edit the auxiliary functions written in Part A for a single matrix so that data of multiple matrices in the interleaved and interleaved chunked format can be given as input to the kernels.

For this, the access expressions in-store and load functions had to be modified. The modifications made for interleaved and interleaved chunked access expressions were different.

**tile_z and tile_y are the tile indexes along with z and y directions respectively, M is the dimension of the matrices and N is the total number of matrices in all pseudo codes in the report.*

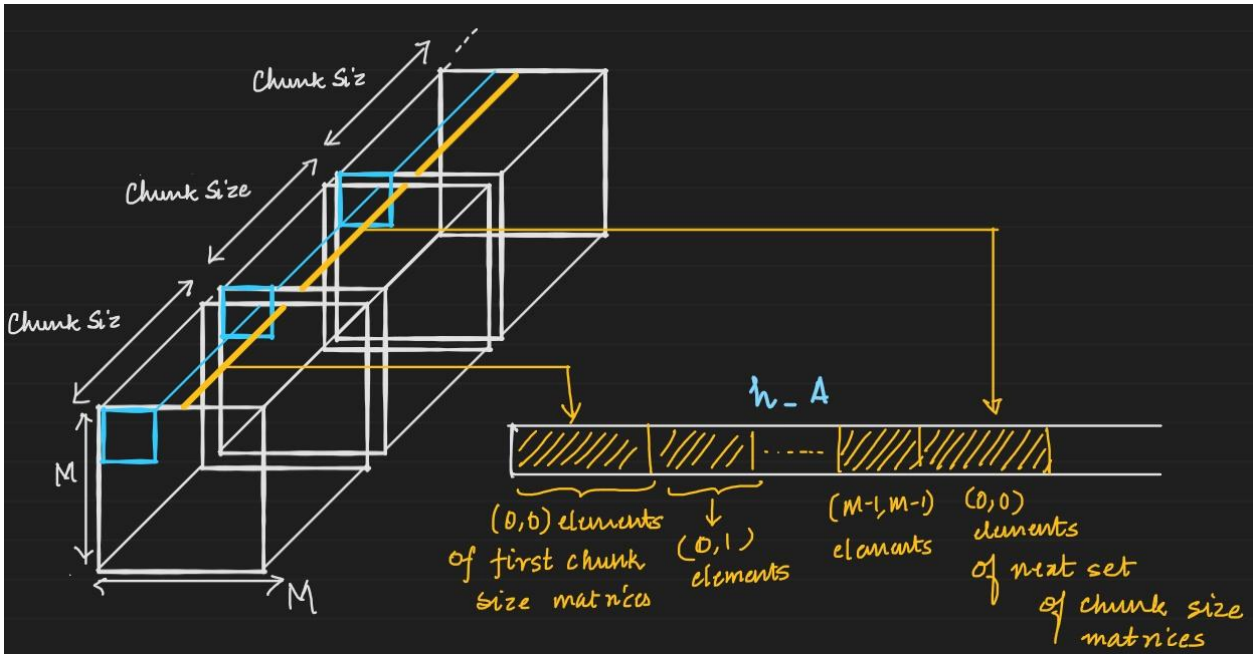
Simple Interleaved Access Expressions for Store and Load functions



*row = tile_z * TILE_SIZE + threadIdx.z;*
*column = tile_y * TILE_SIZE + threadIdx.y;*
*global_memory_id = blockIdx.x * blockDim.x + threadIdx.x + row * N * M + column * M*
*shared_memory_id = threadIdx.z * TILE_SIZE + threadIdx.y + shared_size_single_matrix * threadIdx.x*

Here shared_size_single_matrix is the shared memory used by part A kernel launch for a single matrix.

Interleaved Chunked Access Expressions for Store and Load functions




```
row = tile_z * TILE_SIZE + threadIdx.z;      // access row
column = tile_y * TILE_SIZE + threadIdx.y;   // access col
g_threadX = blockIdx.x * blockDim.x +  threadIdx.x;
global_id = ( ( g_threadX / chunk_size ) * chunk_size ) * N * N + ( row * N + column )
* chunk_size +  ( g_threadX % chunk_size ) ;
```

Now all auxiliary functions apart from the store and load, work on tiles on only a single matrix. Hence from the kernel function, we provided an offset of tx*shared_size_single_matrix to all the shared memory addresses passed to these functions. This offset made sure that tiles of the same position across different matrices are passed to these functions.

The output of this kernel was also according to the input format i.e. if the input format was simple interleaved, then the output of the kernel is also in the simple interleaved format. The same applies to the chunked interleaved format. Now, this output had to be again converted to row-major form before writing it to the output file.

Pseudocode to transform the output host side array **h_A (** in **single interleaved format)** into row-major form and then write it to the output file.

```
for i = 0 to N - 1 do
        for r = 0 to M - 1 do
                for c = 0 to M -1 do
                        global_index = r * M * N + c * N + i;
                        temp = h_A[global_index];
                        writeToFile float <- temp;
                end for
        end for
end for
```

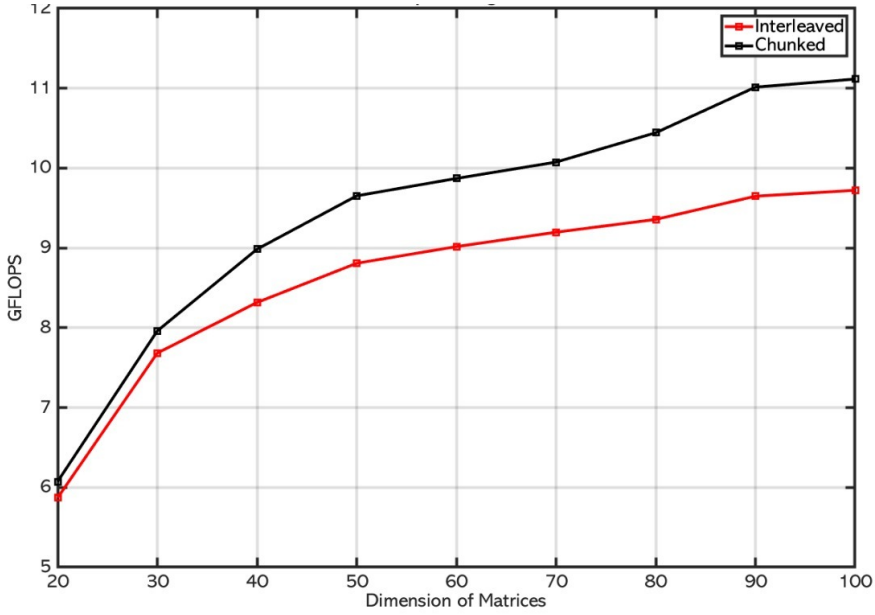
The iteration index i denotes the i^{th} matrix we are reading. N denotes the number of matrices in the input file and M denotes the dimension of the matrices present in the input files. The variable *global_index* is computed in a manner such that we read all the elements of all the matrices from array h_A in a row-major form

Pseudocode to transform the output host side array **h_A (** in **interleaved chunked format)** into row-major form and then write it to the output file. Here we use the variable ***chunk_size*** to denote the chunk size that we will be using in this section.

```
for p = 0 to ceil(N/chunk_size) do
        for i = 0 to chunk_size - 1 do
                for r = 0 to M - 1 do
                        for c = 0 to M -1 do
                                x = r*M + c;
                                global_index = (p*chunk_size) * M * M + x*chunk_size + i;
                                temp = h_A[global_index];
                                writeToFile float <- temp;
                        end for
                end for
        end for
end for
```

Part C: Analysis

1. Variation in Data Layout (Simple Interleaved, and Interleaved Chunked)

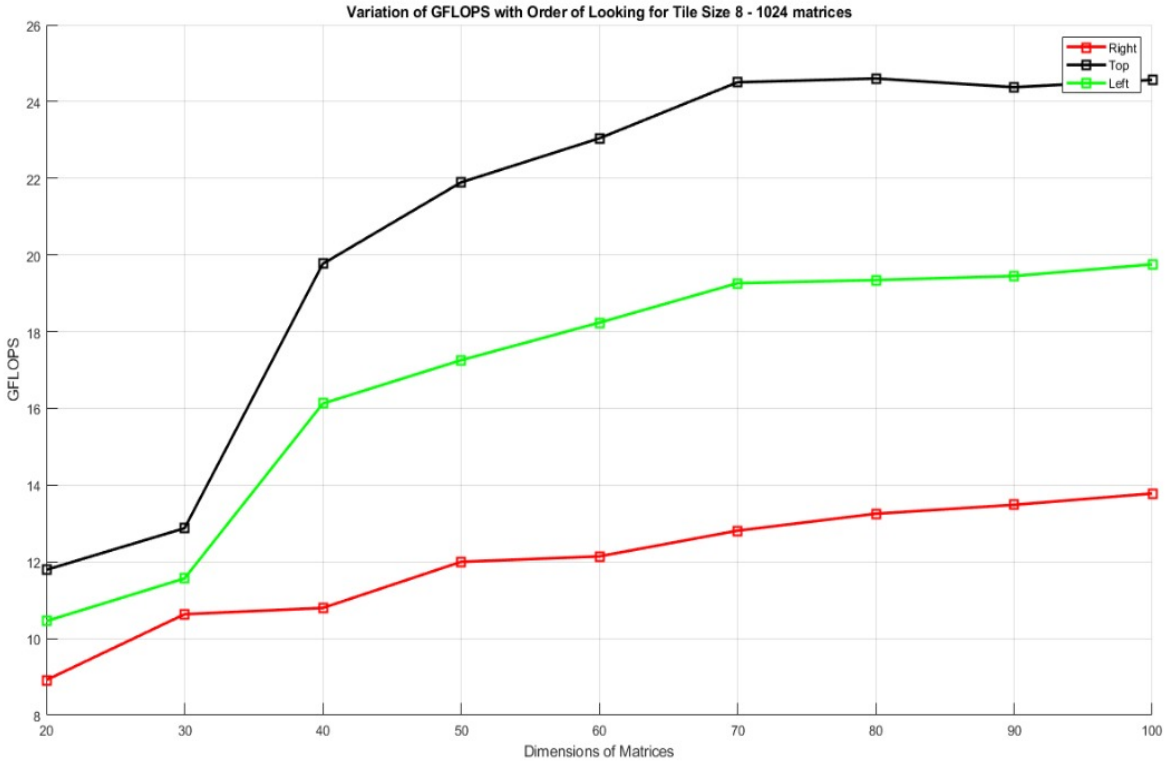


For analyzing the effect of chunked data layout implementation vs. simple interleaved data layout implementation, experiments were conducted on the top-looking version of the Cholesky factorization. The tile size used was 2, and the dataset contained 1024 matrices of 20 x 20 dimensions. The GFlop/s for chunked data layout were found to be higher than the interleaved data layout format, in accordance with the reference paper.

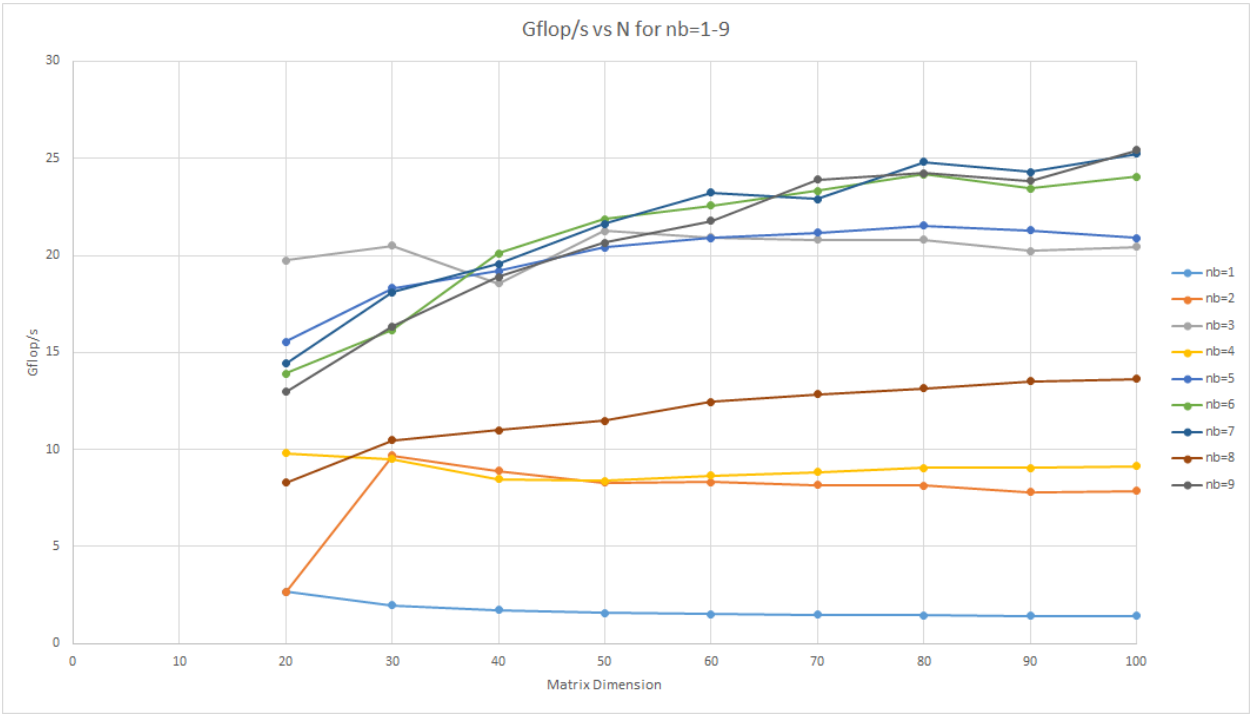
A smaller tile size i.e. tile size = 2 has been used in our experiments involving the interleaved chunked data layout for reasons stated below in sections of Analysis of Chunk Size and Interesting Results.

2. Order of Looking (Left, Right, and Top)

The interleaved implementations for left, right, and top looking were compared, using the same tile size of 8, and 1024 matrices in total. The GFlop/s for the Top-Looking code were found to be highest, followed by Left-Looking and Right-Looking implementations. This is consistent with the results presented in the paper.

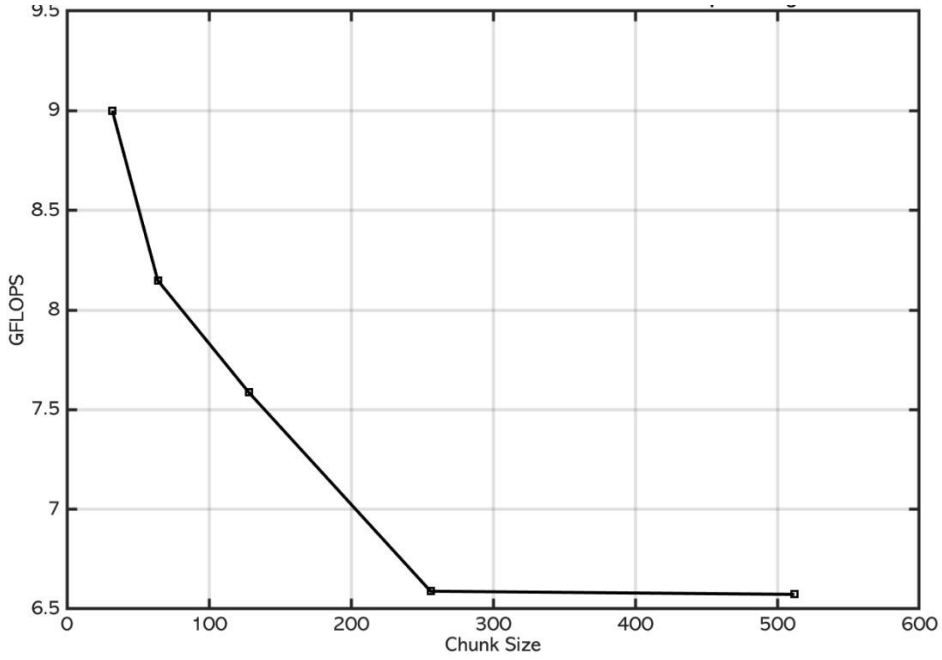


3. Tile Size (variation: 1 to 9)



The simple interleaved implementation of top looking with tile size varied from 1 to 9 were compared, using the same interleaved implementation of top looking tile size of 8, and 1024 matrices in total. 'nb' here, represents the tile size used in the implementation. As tile size increases, Gflop/s were found to increase, although there are some noticeable aberrations in the trend.

4. **Chunk Size (variation: 2^5 to 2^9)**



The above graph shows the performance in terms of GFLOP/s with a varying chunk size. We have used the top looking implementation with 1024 matrices, tile size = 2 and matrices of dimension = 20. Now as it is visible here that as the chunk size increases, the performance of the kernel decreases. This is in accordance with the reference paper.

An interesting observation to notice here is that the condition

$$chunk\ size\ \leq \frac{1024}{(Tile\ size)^{\ 2}} \leftarrow eq\ 1$$

needs to be satisfied in order to reap benefits of using this data layout. This is because in our driver code for chunked interleaved implementation of layout, we have defined a term:

```
int num_of_matrices_per_block = min(min(1024/(TILE_SIZE * TILE_SIZE),
num_of_matrices), chunk_size);
```

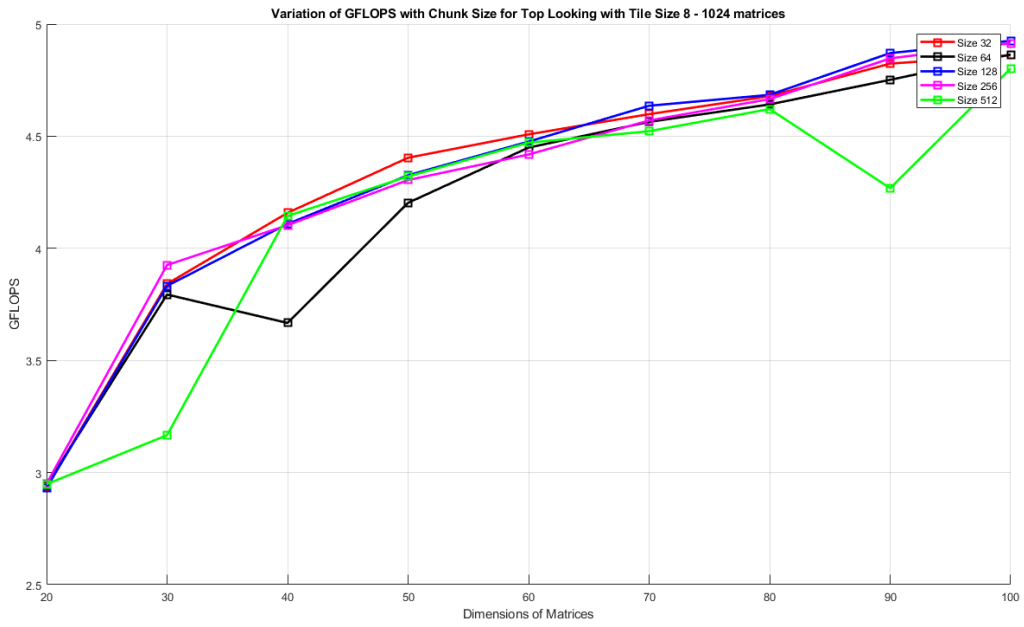
Here we can see that for num_of_matrices >= 1024,

$$num\ of\ matrices\ per\ block\ =\ min(\frac{1024}{(Tile\ size)^{\ 2}},\ chunk\ size)\ \leftarrow eq\ 2$$

Hence when *eq 1* is violated, the parameter num_of_matrices_per_block is the same for all chunk sizes. Hence there are no effects of chunking as the only major difference (apart from small cache differences if there) in terms of performance between performance with different chunk sizes is in terms of num_of_matrices_per_block and thus in terms of dimensions of our grid and blocks. Since smaller chunk sizes ensure that the same computation can be done in multiple smaller blocks instead of one large block and thus leading to easy and more efficient block allocation by the scheduler.

```
int num_of_matrices_per_block = min(min(1024/(TILE_SIZE * TILE_SIZE) , num_of_matrices), chunk_size);
dim3 grid(num_of_matrices / num_of_matrices_per_block , 1, 1);
dim3 block(num_of_matrices_per_block, TILE_SIZE, TILE_SIZE);
// no of tiles in a column
int INPUT_SIZE = dim_of_matrix;
int no_of_tiles = (INPUT_SIZE / TILE_SIZE) + (INPUT_SIZE % TILE_SIZE != 0); // ceil of (INPUT_SIZE / TILE_SIZE)
launch_kernel<<<grid, block, num_of_matrices_per_block * 3 * TILE_SIZE * TILE_SIZE * sizeof(float)>>>(d_A, dim_of_matrix, num_of_matrices ,3 * TILE_SIZE * TILE_SIZE);
```

Since for chunk size = 512 , the condition is not fulfilled we can see that the chunking effect vanishes after chunk size = 256



The above graph is for top looking implementation with 1024 matrices and tile size = 8. As stated earlier *eq 1 n*eeds to be satisfied in order to reap benefits of using this data layout. This implies chunks of chunk size < =16 show visible changes in the performance. Hence for chunk size from 32 to 512, the expected performance are same

We can clearly observe from the graph that for tile size 8 , there is no significant effect of chunking except for a few cases.

Interesting Results

As the paper suggested, chunk interleaved outperformed the simple interleaved data layout in terms of performance. It was however observed that the difference in the average runtimes of interleaved chunked and simple interleaved was not as profound as shown in the paper.

Figure 17 shows the best performance of the interleaved implementation with and without chunking. Clearly, chunking is very beneficial to performance. While we cannot say exactly why this is the case, intuitively, this is the expected outcome. The spatial locality principle takes effect at some level of the memory hierarchy.

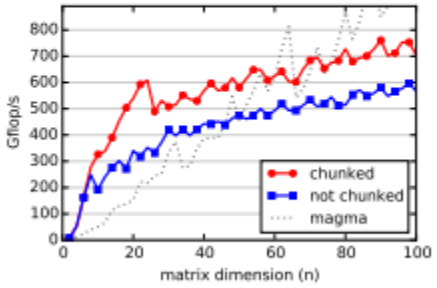


Figure 17. The best performance of the interleaved implementation with and without chunking.

The authors of the paper also have clearly mentioned that they themselves are not aware of the reason for chunked implementation outperforming simple interleaved .

Also it was observed that the condition

$$chunk\ size \leq \frac{1024}{(Tile\ size)^2}$$

needs to be satisfied in order to reap benefits of using this data layout.

Contributions:

- Kshitij Anand - Dataset generation, creation of I/O code, host-side code, plotting profiling data
- Arghyadeep Bandyopadhyay - Top-looking tiled implementation for a single matrix
- Suman Dutta - Auxiliary Functions, Right-looking tiled implementation for a single matrix, plotting profiling data
- Shubhesh Anand - Creation of I/O code, Interleaved and chunked data layouts integration for multiple matrices for left, right, and top looking, host-side code, Part-C analysis
- Divyanshu Sheth - Dataset generation, creation of I/O code, Interleaved and chunked data layouts integration for multiple matrices for left, right, and top looking, host-side code, Part-C analysis
- Anil Kumar Yogi - Auxiliary Functions, Left-looking code for a single matrix
- Nishant Goyal - Left-looking code for a single matrix, Part-C analysis

- Parv Maheshwari - Creation of I/O code, Interleaved and chunked data layouts integration for multiple matrices for left, right, and top looking, host-side code Part-C analysis

The code for all our experiments is available at https://github.com/shubheshanand/HP3_Batched_Cholesky_Decomposition.

Reference Paper

"Autotuning Batch Cholesky Factorization in CUDA with Interleaved Layout of Matrices" by Gates et al. published in IPDPS Workshop 2017.