# Intel® hStreams
# Reference Sample Codes

## Purpose and Scope

The purpose of this document is to list and describe the various reference sample codes that Intel® hStreams ships with.

This document will grow over time to provide more detail; this is a rather "bare-bones" edition. It covers all of the reference codes in the Intel® Many-core Platform Software Stack.

## Overview

Intel® hStreams ships with several reference sample codes.

The purposes of these codes include

- Give users a feel for what can be done with hStreams, in terms of functionality and performance
- Illustrate the use of hStreams APIs
- Make building blocks freely available for integration into user code

The current set of reference codes are as follows, with the directory name shown after each

- Mini applications
  - Cholesky Factorization (`cholesky`)
  - LU Factorization (`lu`)
  - Matrix multiply (`matMult`)
- Directed tests
  - Basic performance (`basic_perf`)
  - IO performance (`io_perf`)
  - Test app (`test_app`)

Each of these reference codes is in a different directory. Cholesky and lu have two versions of the code, one which is tiled for a Xeon host, and one that is tiled with hStreams. The per-code directories contain a README.txt file that describes what it is, how to build it, how to run it, and how to interpret the results. That directory also contains the source code files (*.cpp, *.h), a Makefile and a script to run it.

There is also a `common` directory, which has building blocks used for timing, files with functions that are common to multiple reference codes, and a script to set up the environment.

Finally, there is a `windows` directory that has a subdirectory for each of the reference codes, and in those directories and their subdirectories reside Microsoft® Visual Studio solutions and project files, in addition to Microsoft® Windows batch files for running the ref codes.

In the following sections, we provide more detail on each of the ref codes.

# 1. Cholesky Factorization

Cholesky factorization (or Cholesky decomposition) is a decomposition of Hermitian, postive-definite matrix into the product of a lower triangular matrix and its conjugate transpose, useful for efficient numerical solutions and Monte Carlo simulations. When it is applicable, the Cholesky factorization is roughly twice as efficient as LU factorization for solving systems of linear equations [Ref. 1].

To make use of the high-performance BLAS level-3 functions of the Intel® MKL library, we next describe a tiled version of the Cholesky factorization algorithm [Ref. 2]. For this, the matrix is divided into square tiles as shown in Fig. 1.1. The algorithm is given in Algorithm 1.1.

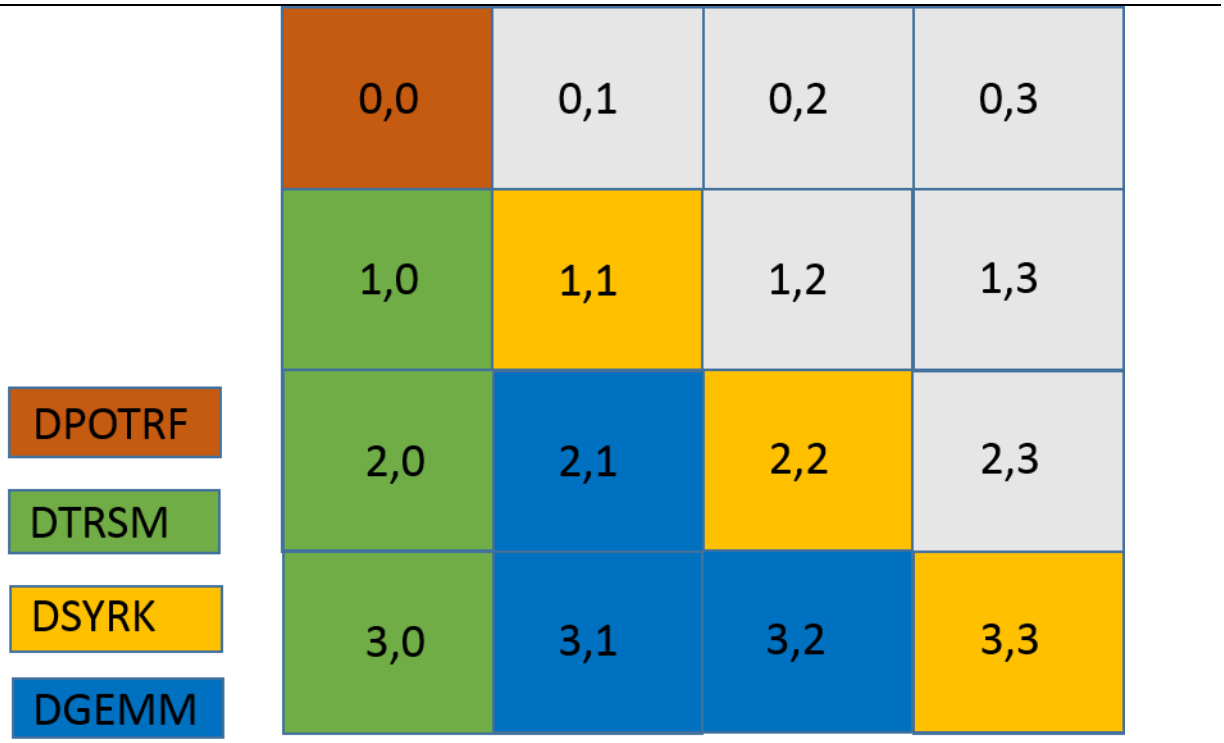


**Figure 1.1**: Tiling of matrix for Cholesky block-level Cholesky factorization. The tiles are colored for the corresponding BLAS operation at the first pass of the outer-loop.

In Fig. 1.1, the color of tiles corresponds to different MKL BLAS operations. For details on these MKL function calls, the reader is referred to the Intel® MKL reference manual. The colored tiles in the picture are those processed in the first pass of the outer loop. The tiles in each column get finalized (contain factored data) as we move through the outer loop. Note that only the bottom half of the matrix (including the diagonal tiles) are operated on. The white tiles are not computed on. We note that the compute is dominated by the `cblas_dgemm` operation at the inner most loop which executes on the matrix tile of size *tile_size*. The cost of this dgemm on the tile is

2*tile_size$^3$. The total computational cost for the tiled algorithm is also $m^3/3$ (not derived), which is the same as the untiled elemental algorithm, where *m* is the matrix size, with *m = T*tile_size.*

Using this tiled algorithm as baseline, we next describe an offload algorithm for tiled Cholesky factorization. The offload algorithm uses both the host and device using the hStreams framework. Before describing the algorithm, it is important to understand the data and compute dependencies in a tiled Cholesky program as described above. For this, we construct a task-graph for the tiled Cholesky program. Fig. 1.2 shows this task-graph for a *4* x *4* block matrix as shown in Fig. 1.1. For each outer-loop pass, the compute dependencies are as follows: POTRF is computed first on the "hot" diagonal tile ("hot" tiles contain the final factored data after this pass of the outer-loop), TRSMs are conducted next on the tiles in the "hot" column, GEMMs and SYRKs are finally conducted on all the remaining tiles on the right of "hot" column, but only for the lower-triangular part of the matrix. Note that POTRF requires only the data for its own tile, TRSM requires two input tiles - the diagonal tile above on which POTRF was executed and its own tile. SYRK also require two input tiles - the "hot" tile on which TRSM was executed in the same block row as itself, and its own tile. Lastly, GEMM requires three input parameters - two "hot" tiles on which TRSM was executed, one corresponding to the row-index and other corresponding to the col-index of the GEMM tile, and its own tile. Also note, that GEMMs and SYRKs do not depend on each other and can be concurrently executed. Thus the concurrency (how many BLAS operations can be executed in parallel) for a *4* x *4* tiled matrix, during the first pass of outer-loop is: 1 DPOTRF, 3 DTRSMs, and 3 DGEMMs + 3 DSYRKs. So the maximum concurrency is 6 and it comes from the DGEMMs and DSYRKs. One can follow the dependency graph as we move to the next iteration of outer loop.

The discussion in the above paragraph provides insight into how to best parallelize the compute on the card, and also how to pipeline data transfer to/from the card with compute on the card. For the offload algorithm, we have found that a *6* x *6* tiling of the input matrix achieves better performance than a *4* x *4* tiling. By using the analogy of the *4* x *4* tiled matrix, we can find that for a *6* x *6* tiled matrix, we have, for the first pass of outer-loop: 1 DPOTRF, 5 DTRSMs, and 10 DGEMMs + 5 DSYRKs. Thus the maximum concurrency increases to 15 if the tile count increases, but of course now the work associated with each tile is smaller. It is observed that the maximum concurrency is always an integer multiple of number of DTRSMs, where the number of DTRSMs are equal to number of tiles minus one (for the first pass). Therefore, for the offload algorithm, we partition the card into *num_tiles* - 1 physical partitions. This amounts to 5 physical partitions on the card for a *6* x *6* tiling. For a C0-7120A Xeon Phi card, which as 61 cores, this means each partition is 12-cores, since the last core is reserved for operating system.

For the offload tiled Cholesky algorithm using the hStreams library, we first define an important concept called “function-specific targeting”, which means that we get most performance if we can execute functions on architectures for which they are best suited. For example, executing a function which does not have much parallelism on the CPU host (which has good serial performance) and a function which as a great deal of intrinsic parallelism on the card. For this algorithm, we execute the POTRF on the diagonal “hot” tile on the CPU host as POTRF does not have a lot of parallelism because of task dependencies in the POTRF function. All the other functions (i.e. TRSM, SYRK, and GEMM) are executed on the card. hStreams cannot yet use the CPU host as a stream for computation, so we execute the POTRF using the host thread and MKL library takes care of spawning multiple host threads to run the POTRF.

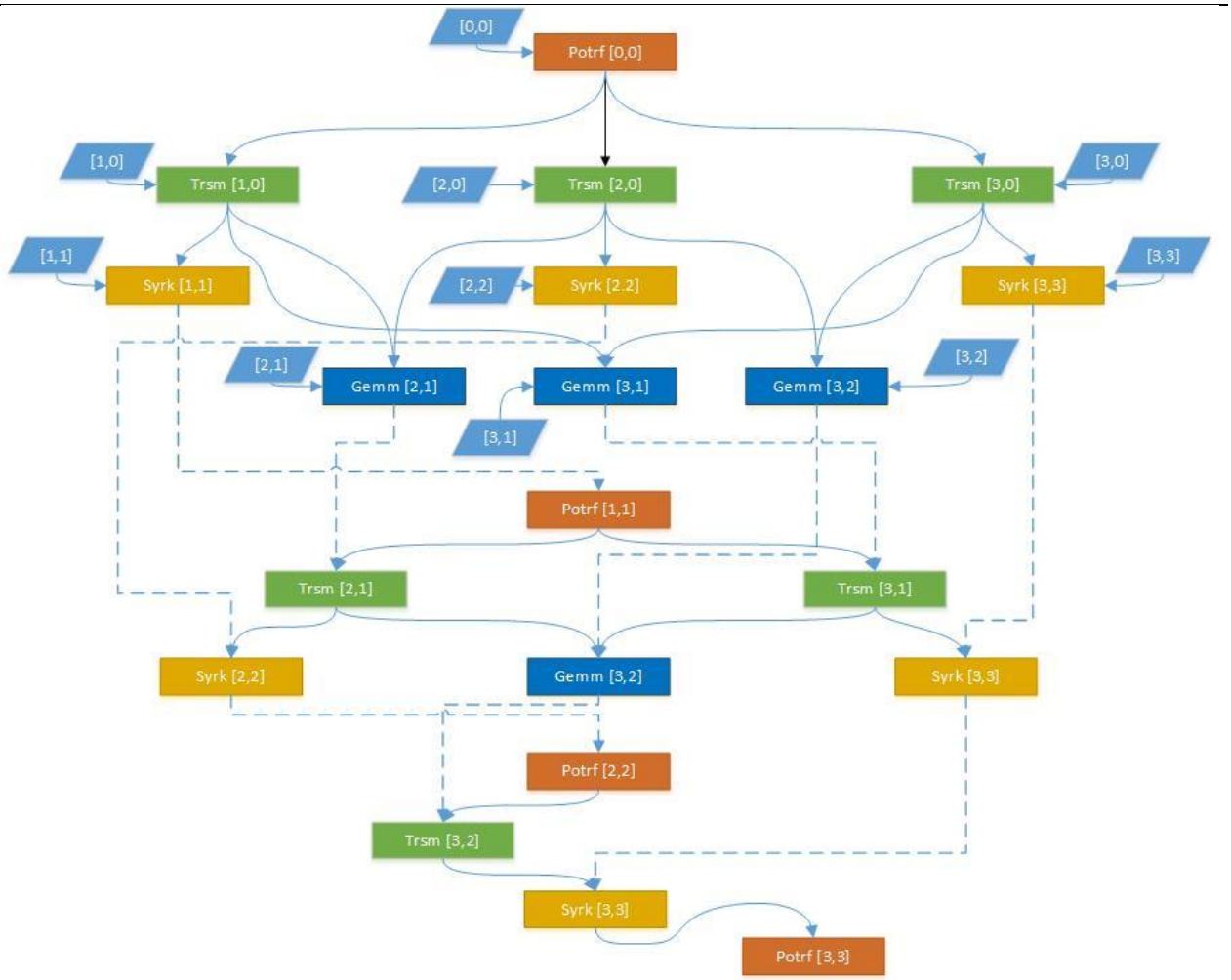


**Figure 1.2**: Tiled Cholesky task graph.

**Algorithm 1.1**: Block-level Cholesky factorization

```
for ( k = 1, T ) do
      A[k,k] = LAPACKE_dpotrf( A[k,k] )
      for ( m = k+1, T ) do
          A[m,k] = cblas_dtrsm( A[k,k], A[m,k] )
      end for

      for ( n = k+1, T ) do
         A[n,n] = cblas_dsyrk( A[n,k], A[n,n] )
         for ( m = n+1, T ) do
            A[m,n] = cblas_dgemm( A[m,k], A[n,k], A[m,n] )
         end for
      end for
end for
```

# 2. LU Factorization

LU decomposition (or LU factorization), where 'LU' stands for 'Lower Upper', is a decomposition of a matrix into the product of a lower triangular matrix and an upper triangular matrix. It is useful for efficient numerical solutions and Monte Carlo simulations. The LU decomposition is just a matrix form of Gaussian elimination [Ref. 1].

Let **A** be a square matrix having M rows by M columns. The algorithm can also be applied to rectangular matrices, but this is rarely done in practice, and the supplied reference code only works for square matrices. The idea is to transform **A** into an M by M upper triangular matrix **U** by introducing zeros below the diagonal, first in column 1, then in column 2, and so on. This is done by subtracting multiples of each row from subsequent rows. This "elimination" process is equivalent to multiplying **A** by a sequence of lower-triangular matrices $\mathbf{L_k}$ on the left:

**Equation 2.1** $\mathbf{L_{m-1}}\ldots\mathbf{L_2}\mathbf{L_1}\mathbf{A} = \mathbf{U}$

and setting $\mathbf{L} = \mathbf{L_1^{-1}}\mathbf{L_2^{-1}}\ldots\mathbf{L_{m-1}^{-1}}$, gives

**Equation 2.2** $\mathbf{A} = \mathbf{LU}$

Thus we obtain LU factorization of **A**, where **U** is upper-triangular and **L** is unit lower-triangular, which means all of its diagonal entries are equal to 1 [Ref. 1].

The computational cost of a LU decomposition is $(2/3)*M^3$ floating point operations. All the three matrices **A**, **L**, and **U** are not needed to be stored; to minimize memory use, **L** and **U** are overwritten on input matrix **A**. Thus the memory footprint is $M^2$*sizeof(double) bytes (for double-precision real-valued matrix). An algorithm which is applied on elements of matrix **A** can only proceed one column at a time as there is a data dependence which limits the overall concurrency. The reference code implements a right-looking LU decomposition algorithm and therefore as we proceed down the columns, the number of entries updated reduces [Ref. 1].

In this program, we do a tiled implementation of LU decomposition without pivoting (for double-precision real-valued matrices) using fast level-3 BLAS components (using

the Intel(R) MKL library). The tiled-algorithm is similar to the one for Cholesky factorization as given in Ref. 2. We use the hStreams framework to partition the card into 4 or 6 physical partitions and schedule BLAS3 operations (DGETRF, DTRSM, DGEMM) into compute-streams associated with these partitions in an as concurrent manner as possible. Note that for DGETRF, we use a custom hand-written kernel instead of the one from MKL. This is because we implement the LU decomposition without pivoting, whereas the function available in MKL does pivoting. The matrix is divided into T x T tiles each containing (M/T * M/T elements). The outer-loop is iterated over 1:T. For the k-th outer-loop iteration, we have 1 DGETRF (of the k x k tile), 2*(T-k) DTRSMs, and (T-k)*(T-k) DGEMMs. The order of operations (and dependence) of BLAS3 operations for a given k is: DGETRF -> DTRSM -> DGEMM (i.e. DGEMM depends on DTRSM, which in turn depends on DGETRF). Thus the max concurrency for a given k is : no. of DGEMMs = (T-k)*(T-k).

The concurrency is highest for k = 1, and is = (T-1)*(T-1). If T = 6, for example, then the max concurrency is 25. It is best if this max concurrency is divisible by number of partitions on the card. As a rule of thumb, we have observed that no. of partitions on card = T - 1 gives the best results, and in particular, we have observed that T = 6, and no. of partitions = 5, give best results for most cases.

There are two versions of the tiled-LU program. One called tiled_host performs the tiled-LU on the host only, without using hStreams or the card. The performance of this program is compared against the MKL DGETRF without automatic offload. The other version is called tiled_hstreams. This uses both the host and the card, by using the hStreams library. The performance of the tiled_hstreams version is compared against the MKL DGETRF with automatic offload. For the tiled_hstreams LU decomposition, data transfer to/from the card is interleaved with compute on the card using the concurrently running asynchronous streams.

To understand the data/compute dependencies, it's recommended that the user first becomes familiar with the algorithm in the tiled_host example and then it will be easier to understand the various synchronizations (in the form of _event_wait) required in the tiled_hstreams example.

References:
1) Trefethen, Lloyd N. and Bau III, David, Numerical Linear Algebra, SIAM (1997).
2) Jeannot, Emmanuel. Performance Analysis and Optimization of the Tiled Cholesky Factorization on NUMA Machines. PAAP 2012-IEEE International Symposium on Parallel Architectures, Algorithms and Programming. 2012.

# 3. Matrix Multiply

Matrix multiplication is an expensive mathematical operation used in many software applications. If we define two input matrices **A** and **B** with **A** having *m* rows by *k* columns and **B** having *k* rows by *n* columns, then the matrix product **C** = (**A** x **B**) is of size *m* rows by *n* columns. Each element $\boldsymbol{C_{ij}}$ of **C** is obtained by taking the inner product (or dot product) of the $i^{th}$ row of **A** with the $j^{th}$ column of **B**. This can be written formally as shown below:

***Equation 3.1:*** $\boldsymbol{C_{ij}} = \sum_{s=1}^{k} \boldsymbol{A_{is}} \times \boldsymbol{B_{sj}}$

Here, the first subscript denotes the row index and second subscript denotes the column index. Please note that in all the algorithms presented in this document, we will be using 1-indexing (start the arrays with an index 1) for matrices and vectors.

A simple algorithm for matrix multiplication is given in Algorithm 3.1. From Algorithm 3.1, to compute one element of **C** requires $k$ multiplications and $k$ additions, or 2 $k$ floating point operations (flops). Therefore, to obtain the complete **C** matrix, the total number of flops is equal to $2mnk$.

**Algorithm 3.1**: Simple elemental matrix multiplication algorithm

```
for i = 1, m do
    for j = 1, n do
        C_ij = 0
        for s = 1, k do
          C_ij = C_ij + A_is * B_sj
        end for
    end for
end for
```

A similar algorithm could be applied if the matrices are subdivided into tiles (or blocks). In the example in Fig. 3.1 the matrices are divided into *3* x *3* blocks. If we use the Intel® MKL call for double-precision (DP) matrix multiplication (cblas_dgemm) the algorithm is given in Algorithm 3.2. Note that not all input parameters to cblas_dgemm are shown. Here, it is assumed that the matrices **A** and **B** are square and the total tiles in each direction is equal for both matrices (= *num_tiles*). Note that the total compute for the block matrix multiplication still remains the same as the elemental matrix multiplication. This blocking concept will be used in the next algorithm where we use an offload programming model with the help of hStreams library.

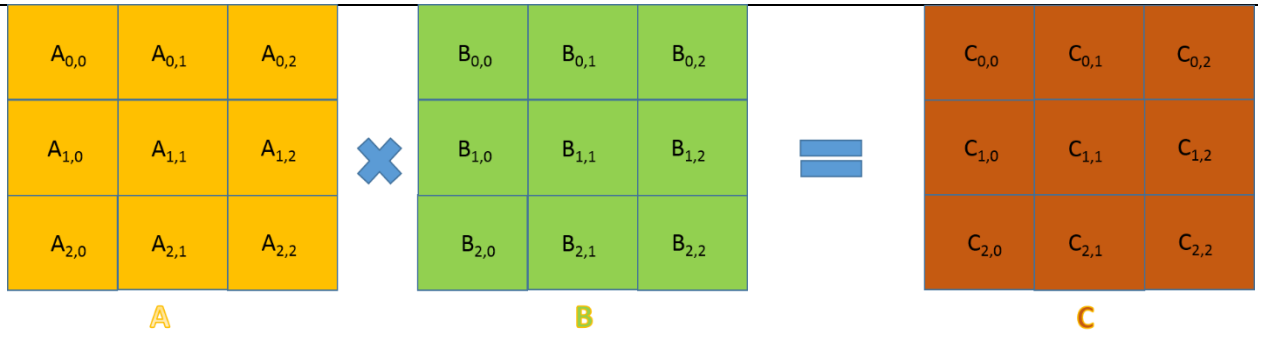


**Figure 3.1**: Decomposition of input matrices **A** and **B** and output matrix **C** into blocks

Next, we discuss an algorithm which uses a high-speed compute device (Xeon Phi™ card in the present case) connected to a host, using an offload programming model. To use a high speed compute device to do matrix multiplication for double-precision matrices, we need to send $mk$*sizeof(double) bytes for matrix **A** and send $kn$*sizeof(double) bytes for matrix **B**. We typically have to allocate enough memory

*mn**sizeof(double)) to hold the output matrix first. In certain common situations we need to initialize that **C** output matrix to zero first. Then we start the matrix multiplication operation on the high speed compute device. And finally, we need to bring the output data (matrix **C**) back from the device. So the total data transfer is quantified as *mk+kn+mn* double
precision numbers where double precision numbers require 8 bytes each.

**Algorithm 3.2**: Block matrix multiplication algorithm

```
for i = 1, num_tiles do
    for j = 1, num_tiles do
        C_ij = 0
        for s = 1, num_tiles do
          cblas_dgemm(A_is, B_sj, C_ij)
        end for
    end for
end for
```

For simplicity of the following analysis, we assume that all the three matrices **A**, **B**, and **C** are square of size *n* by *n*. Then the matrix multiplication requires $2n^3$ flops on $24n^2$ bytes of data. If $R_{to}$ is the data rate from the host to the compute device and $R_{from}$ is the data rate from the device back to the host, then the total data transfer time, $T_{xfer}$ is:

***Equation 3.2:*** $$T_{xfer} = 16n^2 / R_{to} + 8n^2 / R_{from}$$

If $G_H$ is the achievable flops per second on the host, then the time to compute the matrix multiplication on the host only, $T_H$ is:

***Equation 3.3:*** $$T_H = 2n^3 / G_H$$

Similarly, if $G_D$ is the achievable flops per second on the device, then the time to compute the matrix multiplication on the device alone, $T_D$ is:

***Equation 3.4:*** $$T_D = 2n^3 / G_D$$

Using the device to compute the matrix multiplication is beneficial only if the following condition holds:

***Equation 3.5:*** $$T_D + T_{xfer} < T_H$$

Expanding the terms and approximating $R_{to} \sim= R_{from} = R$ yields the following:

***Equation 3.6:***

$$2n^3 / G_D + 24n^2 / R < 2n^3 / G_H$$
$$\rightarrow 1 / G_H + 12 / Rn < 1 / G_H$$
$$\rightarrow n > 12 / R (1 / G_H - 1 / G_D )$$

Eq. 3.6 gives a lower-bound on *n*, below which doing the computation on the device is not beneficial to the performance. To relax the requirement specified in Eq. 3.6, and to

extract more performance out of the device, we next look at a tiled algorithm for conducting the matrix multiplication.

## Tiled Algorithm

For the tiled algorithm, let us divide the matrices **A** and **B** into blocks, such that **A** contains *mblocks x kblocks* and **B** contains *kblocks x nblocks*. Thus each block of **A** contains ( *m* / *mblocks* ) x ( *k* / *kblocks* ) elements and each block of **B** contains *k* / *kblocks* x *n* / *nblocks* elements.

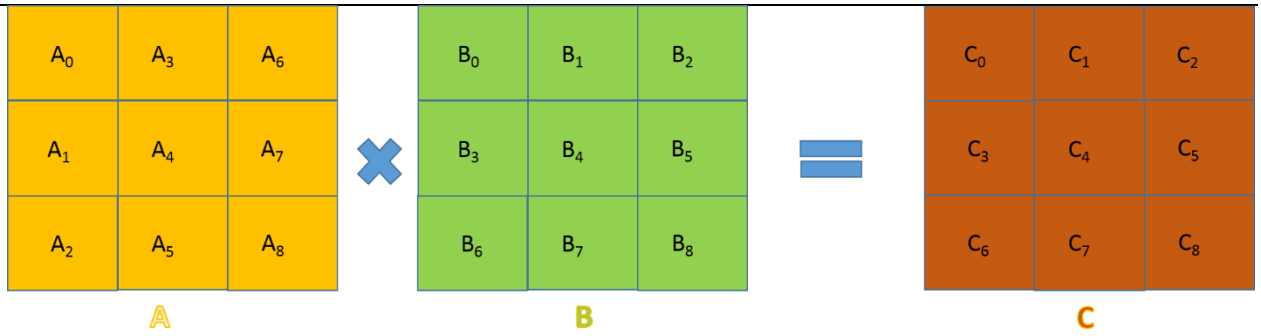


**Figure 3.2**: Decomposition of input matrices **A** and **B** and output matrix **C** into blocks.

An example for square matrices divided into *3* x *3* blocks is shown in Fig. 3.2. Note that in Fig. 3.2 we have ordered the blocks in **A** in a column-major fashion and the blocks in **B** and **C** in a row-major fashion. By this ordering we obtain some useful generalizations as we show next. Expanding the entries for $C_i$'s:

$$C_0 = A_0 \times B_0 + A_3 \times B_3 + A_6 \times B_6$$
$$C_1 = A_0 \times B_1 + A_3 \times B_4 + A_6 \times B_7$$
$$C_2 = A_0 \times B_2 + A_3 \times B_5 + A_6 \times B_8$$
$$C_3 = A_1 \times B_0 + A_4 \times B_3 + A_7 \times B_6$$
$$\ldots$$
$$C_8 = A_2 \times B_2 + A_5 \times B_5 + A_8 \times B_8$$

The first generalization that we obtain is for the validity of the multiplication of the blocks of **A** and **B**. The multiplication between a block $A_i$ and $B_j$ is valid if and only if it satisfies Eq.3.7. Otherwise the multiplication does not contribute to any of the blocks of the result matrix **C**.

***Equation 3.7:***
$$\lfloor \frac{i}{mblocks} \rfloor == \lfloor \frac{j}{nblocks} \rfloor$$

where $\lfloor x \rfloor$ denotes the greatest integer less than or equal to $x$.
The next useful generalization is the index $k$ of the block of **C**, $C_k$, obtained as a result of multiplying a valid $A_i$ and $B_j$ pair. Eq.3.8 gives the result.

***Equation 3.8:***
$$k = (i \,\%\, mblocks) * nblocks + j \,\%\, nblocks$$

Using these two generalizations we can write an algorithm which does the matrix multiplication for any value of the block sizes. The key steps of the algorithm are

**Algorithm 3.3**: Tiled Matrix Multiplication using hStreams

shown in Algorithm 3.3. The various functions used in the algorithm related to partitioning the device, creation of streams, data transfer between host and device, compute on the device, etc. are all available using the hStreams APIs. These are marked by *hS in the algorithm. These steps can be described as follows. First partition the device into a small number (typically 4) of physical partitions and create and associate a logical stream per partition. Start transferring the matrix tiles (or blocks) from matrix **A** and matrix **B** to the device in logical streams. The logical stream used to transfer tiles is given by the tile subscript modulo maximum logical streams available. Next queue up multiplications on the card. For this, when the device receives a new tile for **A** then loop over all **B** tiles residing on the device and if the multiplication of the new **A** tile with the existing **B** tile is valid, then queue up the multiplication. Similarly queue up multiplications when a new **B** tile is received. Finally, transfer over all the computed **C** tiles from device to host. Note that using this algorithm, we are able to queue up compute on the device in a ``greedy'' manner, i.e. as soon as the two input matrices for compute are available on the device, the computation is queued. This “pipelining” effect maximizes the interleaving of compute and data transfer. Moreover, to minimize the need for stream synchronization, each individual block of output matrix **C** is computed in a fixed stream. This way, we can rely on the FIFO characteristics of the stream to give correct answer (without the need for extra synchronizations).

**Partition** the device and **create** one logical stream per partition. Let the total streams created be nstreams (*hS)
*totalblocksA = mblocks*kblocks*
*totalblocksB = kblocks*nblocks*
countA = -1
countB = -1
**while (** countA < *totalblocksA - 1* OR countB < *totalblocksB – 1* **) do**
  **if (** countA ≤ countB **)**
    **increment** countA
    **transfer $\mathbf{A_{countA}}$** from host to device in stream id: countA % nstreams (*hS)
  **else if (** countB < countA **)**
    **increment** countB
    **transfer $\mathbf{B_{countB}}$** from host to device in stream id: countB % nstreams (*hS)
  **end if**
  **if (** countA > countB **)** **//**A new **$\mathbf{A_{countA}}$** is received
    **for (** *j* = 0; *j* ≤ countB; *j*++ **) do //** loop over all **$\mathbf{B_{countB}}$** 's present on device
      // Check validity of multiplication per Eq.1.7
      **if (** ⌞*countA* / *mblocks*⌟ == ⌞*j* /nblocks⌟ **)**

        *countC* = (*countA* % *mblocks*)*nblocks + *j* % *nblocks* //Using Eq.1.8
        **call** cblas_dgemm on the device to compute **$\mathbf{C_{countC}}$** = **$\mathbf{C_{countC}}$** + **$\mathbf{A_i}$** x **$\mathbf{B_j}$**
          in stream id: countC % nstreams (*hS)
      **end if**
    **end for**
  **else** //A new **$\mathbf{B_{countB}}$** is received
    **for (** *i* = 0; *i* ≤ countA; *i*++**) do** //loop over all **$\mathbf{A_i}$**s present on device
      //Check validity of multiplication per Eq.1.7
      **if (** ⌞ *i* / *mblocks*⌟ == ⌞ *countB* / nblocks⌟ **)**
        countC = (*i* % *mblocks*)*nblocks + *countB* % *nblocks* // Using Eq.1.8
        **call** cblas_dgemm on the device to compute **$\mathbf{C_{countC}}$** = **$\mathbf{C_{countC}}$** + **$\mathbf{A_i}$** x **$\mathbf{B_{countB}}$**
          in stream id: countC % nstreams (*hS)
      **end if**
    **end for**
  **end if**
**end while**
Transfer computed blocks of **C** from device to host (*hS)

# Basic performance

The basic performance microbenchmark provides very basic performance testing using the app APIs. It transfers data to a card, calls gemm, and transfers the result back from a card, for each of 1 stream and 4 streams. There are no command line arguments.

The size of the matrix moved, the type, the gemm type, the number of hStreams and the number of iterations may be modified using macro definitions.

The results show that DMAs in different channels don't slow down much, as expected. If it goes faster with more streams, that's likely in the noise.

This also shows that concurrent dgemms in multiple streams are a bit faster than dgemms that are OpenMP-parallelized in a single stream.

## IO performance

The IO performance microbenchmark can be used to measure data transfers between the host and card, while varying the buffer size, number of concurrent streams, iterations, and direction.

## Test app

The test app microbenchmark is unique in its use of core APIs. It can perform various combinations of data transfers and gemm operations, for varying numbers of streams, and can count init and alloc time if desired. The actions to be performed are controlled with a bit vector input parameter. The README.txt describes the input arguments.