Compiling Matmul to Blocks and Tiles, Version 2

Technical Report, GSI Technology, Preliminary Draft

Brian Beckman
Technology Fellow, GSI Technology
November, 2023

Abstract

This is an executable design specification. It is straightforward to transcribe its proofs and prototypes from Wolfram language to any ordinary programming language of your choice.

The ***layout problem*** answers “how to rearrange matrices to fit the APU?” Consider domain matrices, $A[m, k]$ and $B[k, n]$, of arbitrary but *compatible* dimensions, meaning that the column count, k, of A equals the row count, k, of B. Due to compatibility, the non-commutative matrix product $A.B$ is sensible. Now consider the Gemini-I APU, which has a main memory (MMB) of $24 \times$VR bits, where a VR is 64 HBs and an HB (half-bank) is 2048×16 bits. The Gemini-I APU also has 53 VRs worth of space in L1 cache (parity off). The layout problem for matrix multiplication is finding an optimal procedure for dynamically loading, multiplying, and storing non-overlapping partitions of A and B in the APU’s L1 cache and main memory. The solution to the layout problem includes finding optimal sizes of ***blocks*** and ***tiles*** and optimal sequences of operations for moving and multiplying blocks and tiles. Blocks are optimized to fit L1. tiles are optimized to fit main memory, where multiplication occurs. *Optimal* means *with minimum running time*. Compile time is not considered. Running time includes the time for I/O between L1 and main memory but not I/O to a host computer.

At first glance, the layout problem seems like a constrained combinatorial optimization problem, thus difficult to pose well and expensive to solve. This paper by Kuzma *et al*. (https://arxiv.org/pdf/2305.18236.pdf) presents optimal partitioning of matrices into blocks and blocks into tiles at compile time. We investigate Kuzma’s algorithm in this paper, first by reproducing Kuzma’s original example, then by adapting that example to the APU.

Accumulated Outer Product

First, we note that accumulated outer product is preferable to iterated inner product for all dimensions > 1. This fact justifies the inner-most routine shown below, **tileMul**.

In[1]:=
```
ClearAll[row, col];
row[M_, i_] := M⟦i⟧;
col[M_, i_] := Mᵀ⟦i⟧ᵀ;
```

In[4]:=
```
ClearAll[iteratedInnerProduct, accumulatedOuterProduct, builtInProduct];
iteratedInnerProduct[m_, k_, n_, A_, B_] :=
  Module[{i, j, ab = ConstantArray[0, {m, n}], result, time},
   {time, result} = AbsoluteTiming[
     For[i = 1, i ≤ m, i++,
      For[j = 1, j ≤ n, j++,
       ab⟦i, j⟧ = row[A, i].col[B, j]]]; ab];
   <|"m" → m, "k" → k, "n" → n, "result" → result,
    "time" → Quantity[time, "Seconds"]|>];
accumulatedOuterProduct[m_, k_, n_, A_, B_] :=
  Module[{kk, ab = ConstantArray[0, {m, n}], result, time},
   {time, result} = AbsoluteTiming[
     For[kk = 1, kk ≤ k, kk++,
      ab += Outer[Times, col[A, kk], row[B, kk]]]; ab];
   <|"m" → m, "k" → k, "n" → n, "result" → result,
    "time" → Quantity[time, "Seconds"]|>];
builtInProduct[m_, k_, n_, A_, B_] :=
  Module[{kk, ab, result, time},
   {time, result} = AbsoluteTiming[ab = A.B];
   <|"m" → m, "k" → k, "n" → n, "result" → result,
    "time" → Quantity[time, "Seconds"]|>];
```

Large Matrices

In[8]:=
```
On[Assert];
```

In[9]:=

```
ClearAll[timings];
(timings = With[{precision = 1.*^-5},
    Module[{timings =
       Table[With[{m = d, k = d, n = d},
         With[{A = RandomReal[{0., 1.}, {m, k}],
           B = RandomReal[{0., 1.}, {k, n}]},
          <|"dim" → d,
           "built-in" → builtInProduct[m, k, n, A, B],
           "inner" → iteratedInnerProduct[m, k, n, A, B],
           "outer" → accumulatedOuterProduct[m, k, n, A, B]|>
         ]], {d, 1, 200, 25}]},
     Map[Assert[
        Round[#["built-in"]["result"], precision] ===
         Round[#["inner"]["result"], precision] ===
         Round[#["outer"]["result"], precision]
       ] &, timings];
     Map[{#["dim"], #["built-in"]["time"],
        #["inner"]["time"], #["outer"]["time"]} &, timings]
    ]]) // MatrixForm
```

Out[10]//MatrixForm=

$$\begin{pmatrix} 1 & 7.\times10^{-6}\text{ s} & 0.000021\text{ s} & 0.000015\text{ s} \\ 26 & 0.000015\text{ s} & 0.001694\text{ s} & 0.000082\text{ s} \\ 51 & 0.000014\text{ s} & 0.008331\text{ s} & 0.000244\text{ s} \\ 76 & 0.000246\text{ s} & 0.027375\text{ s} & 0.000636\text{ s} \\ 101 & 0.000911\text{ s} & 0.059154\text{ s} & 0.001963\text{ s} \\ 126 & 0.000083\text{ s} & 0.120059\text{ s} & 0.003073\text{ s} \\ 151 & 0.000099\text{ s} & 0.186912\text{ s} & 0.004844\text{ s} \\ 176 & 0.000166\text{ s} & 0.20466\text{ s} & 0.005295\text{ s} \end{pmatrix}$$

In[11]:=

```
ClearAll[plottableTimings];
plottableTimings[j_] :=
 {col[timings, 1], (Log10@*QuantityMagnitude)[col[timings, j]]}ᵀ
```

In[13]:=
```
ListLinePlot[{plottableTimings[2],
  plottableTimings[3], plottableTimings[4]},
 (*ImageSize→Large,*)GridLines → Automatic,
 Frame → True, PlotLegends → {"built-in", "inner", "outer"},
 FrameLabel →
  {{"Log10(time[s])", ""}, {"Square Matrix Dimensions", "Running Times"}}]
```

Out[13]=

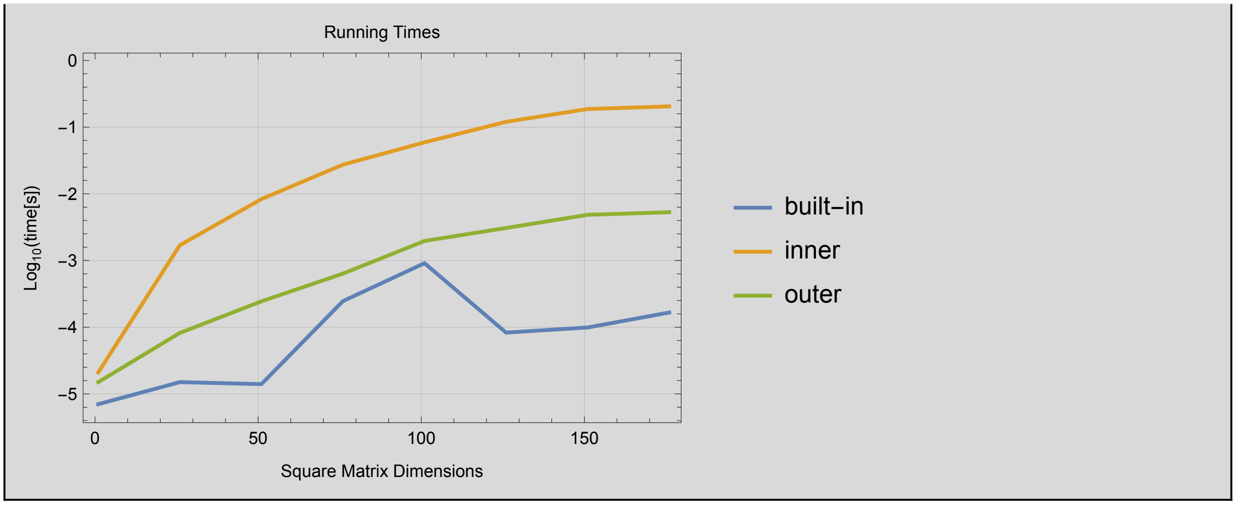


Table 1 — VSR and ACC

Codegen for GEMM

GEMM is a standard operation in LAPACK.

Algorithm 1

A, **B**, **APack**, **BPack**, **AccTile**, **ATile**, **BTile**, **ABTile**, **CTile**, and **CNewTile** are free-variable pointers to memory. **nr** , **kr**, **mr** are free ***packing parameters***. In my opinion, they would be better called *tiling parameters* because they're tuned to the intrinsic LLVM on line 12, but I'll follow the paper's nomenclature for now. **nc**, **kc**, **mc** are free ***blocking parameters*** that divide matrices into blocks appropriately sized and ordered (row-major versus column-order) for cache. **lda**, **ldb**, **ldc** are free ***leading dimensions***, thus strides, and pertain to either row-major or column-major storage conventions. The ***pack*** function reorders blocks into row-major or column-major order as needed for optimal tile-multiplication speed. $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ are free scalar parameters required by GEMM. My implementation refactors Algorithm 1 for greater clarity. Later, we show a direct transcription of Algorithm 1 and compare it to our refactored form.

In[18]:=
```
ClearAll[packingParameters, mr, kr, nr, blockingParameters,
  mc, kc, nc, A, APack, B, BPack, leadingDimensions, lda, ldb,
  ldc, ATile, BTile, AccTile, ABTile, CTile, CNewTile, β, α];
packingParameters = {mr, kr, nr};
blockingParameters = {mc, kc, nc};
leadingDimensions = {lda, ldb, ldc};
```

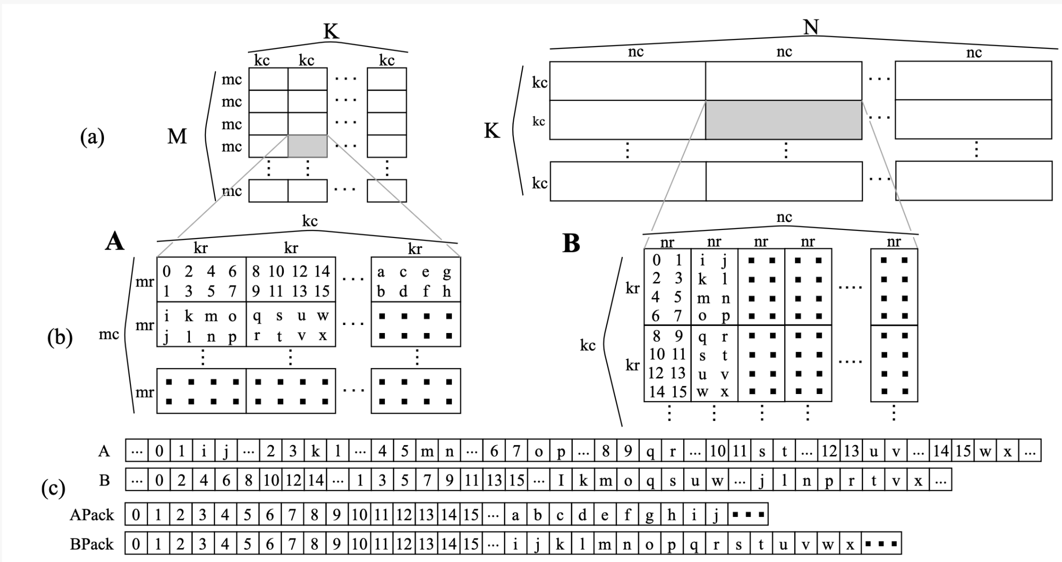
Algorithm 1. Algorithm overview for GEMM

```
for j ← 0, N, step nc do
    for k ← 0, K, step kc do
        pack(B, BPack, k, j, kc, nc, kr, nr, "B," "Row")
        for i ← 0, M, step mc do
            pack(A, APack, i, k, mc, kc, mr, kr, "A," "Col")
            for jj ← 0, nc step nr do
                for ii ← 0, mc, step mr do
                    AccTile ← 0
                    for kk ← 0, kc, step kr do
                        BTile ← loadTile(BPack, kk, jj, kr, nr, ldb)
                        ATile ← loadTile(APack, ii, kk, mr, kr, lda)
                        ABTile ← llvm.matrix.multiply(ATile, BTile, mr, kr, nr)
                        AccTile ← ABTile + AccTile
                    end for
                    CTile ← loadTile(C, i + ii, j + jj, mr, nr, ldc)
                    if k == 0 then
                        CTile ← β×CTile
                    end if
                    CNewTile ← α×AccTile
                    CTile ← CTile + NewCTile
                    storeTile(CTile, C, i + ii, j + jj, mr, nr, ldc)
                end for
            end for
        end for
    end for
end for
```

M, **K**, **N** are original dimensions: $M \times K$ for **AOriginal**, $K \times N$ for **BOriginal**. **kc** (block size) must divide **K**; **mc** (block size) must divide **M**, **nc** (block size) must divide **N**. If not, the original matrices, **AOriginal** and **BOriginal**, must be padded out with zeros to integer multiples of **mc**, **kc**, **nc**. Such is preprocessing, not described here. Likewise, **mr** (tile size) must divide **mc**, **kr** (tile size) must divide **kc**, and **nr** (tile size) must divide **nc**.

In the following illustration, **AOriginal** and **BOriginal** are stored in column-major order.

In[ ]:=



Let us mechanize a concrete version of this illustration by ignoring most ellipses (triple dots). An exception is the picture of **B**, for which we increase **kc** from 2 kr to 3 kr for consistency with the picture of **A**. The two pictures for **A** and for **B** represent the (4, 2) and (2, 2) 1-indexed blocks, respectively, of the original matrices, **AOriginal** and **BOriginal**.

Compiling MatMul to Blocks and Tiles

tileMul

Everything gets compiled to calls of **tileMul**. This is our refactored stand-in for the loop over *llvm.matrix.multiply* on lines 9 through 13 of Algorithm 1. Our stand-in for *llvm.matrix.multiply* itself is an invocation of Mathematica's built-in **Dot** operator.

tileMul multiplies blocks that contain small *tiles*, multiplying each tile at maximum speed in the machine. A ***tile*** is a sub-matrix that snugly fits in the particular machine registers used for multiplication. **tileMul** is here parameterized to the dimensions of blocks and tiles so that we can compile to various devices, such as the Gemini-I APU and the Gemini-II APU, which differ in dimensions.

tileMul takes a pair of blocks with snug tiles inside, then triples of integers for *inner* and *outer dimensions*. The three outer dimensions, **mc**, **kc**, and **nc**, are dimensions of block multiplicands, **mc** × **kc** and **kc** × **nc**. The three inner dimensions, **mr**, **kr**, and **nr**, are dimensions of tile multiplicands, **mr** × **kr** and **kr** × **nr**. Each inner dimension must evenly divide the corresponding outer dimension, meaning that tiles must partition blocks, that is, fit in blocks with no gaps or overlaps. The number of block rows must be an integer multiple of the number of tile rows, and likewise for columns. Each of $\frac{\text{mc}}{\text{mr}}$, $\frac{\text{kc}}{\text{kr}}$, and

$\frac{\text{nc}}{\text{nr}}$ must be integers.

As an illustration, consider the following two tiled blocks.

In[22]:=

$$\texttt{aBlockTiled\$} = \begin{pmatrix} \begin{pmatrix} 15 & 14 & 10 & 15 \\ 3 & 9 & 8 & 5 \end{pmatrix} & \begin{pmatrix} 6 & 0 & 3 & 8 \\ 5 & 10 & 9 & 6 \end{pmatrix} & \begin{pmatrix} 3 & 14 & 1 & 1 \\ 6 & 1 & 14 & 4 \end{pmatrix} \\ \begin{pmatrix} 11 & 14 & 9 & 10 \\ 11 & 13 & 3 & 3 \end{pmatrix} & \begin{pmatrix} 12 & 2 & 0 & 15 \\ 5 & 3 & 15 & 13 \end{pmatrix} & \begin{pmatrix} 5 & 4 & 14 & 1 \\ 13 & 4 & 7 & 3 \end{pmatrix} \\ \begin{pmatrix} 12 & 7 & 14 & 14 \\ 15 & 6 & 14 & 13 \end{pmatrix} & \begin{pmatrix} 2 & 7 & 15 & 15 \\ 1 & 13 & 0 & 8 \end{pmatrix} & \begin{pmatrix} 5 & 0 & 9 & 1 \\ 14 & 5 & 2 & 10 \end{pmatrix} \end{pmatrix};$$

$$\texttt{bBlockTiled\$} = \begin{pmatrix} \begin{pmatrix} 9 & 4 \\ 4 & 1 \\ 11 & 9 \\ 15 & 3 \end{pmatrix} & \begin{pmatrix} 9 & 13 \\ 14 & 12 \\ 9 & 13 \\ 12 & 13 \end{pmatrix} & \begin{pmatrix} 6 & 7 \\ 7 & 11 \\ 10 & 3 \\ 7 & 15 \end{pmatrix} & \begin{pmatrix} 12 & 3 \\ 8 & 4 \\ 0 & 13 \\ 3 & 12 \end{pmatrix} & \begin{pmatrix} 4 & 11 \\ 8 & 11 \\ 5 & 4 \\ 5 & 5 \end{pmatrix} \\ \begin{pmatrix} 13 & 15 \\ 14 & 2 \\ 0 & 13 \\ 10 & 0 \end{pmatrix} & \begin{pmatrix} 7 & 0 \\ 10 & 15 \\ 3 & 15 \\ 8 & 7 \end{pmatrix} & \begin{pmatrix} 1 & 4 \\ 6 & 6 \\ 15 & 6 \\ 11 & 7 \end{pmatrix} & \begin{pmatrix} 15 & 5 \\ 11 & 6 \\ 15 & 6 \\ 11 & 11 \end{pmatrix} & \begin{pmatrix} 5 & 8 \\ 14 & 10 \\ 8 & 15 \\ 5 & 15 \end{pmatrix} \\ \begin{pmatrix} 3 & 14 \\ 10 & 10 \\ 7 & 5 \\ 10 & 8 \end{pmatrix} & \begin{pmatrix} 0 & 3 \\ 14 & 5 \\ 6 & 1 \\ 0 & 3 \end{pmatrix} & \begin{pmatrix} 5 & 3 \\ 9 & 7 \\ 13 & 2 \\ 14 & 12 \end{pmatrix} & \begin{pmatrix} 5 & 2 \\ 11 & 15 \\ 1 & 10 \\ 7 & 5 \end{pmatrix} & \begin{pmatrix} 9 & 12 \\ 11 & 0 \\ 2 & 7 \\ 8 & 6 \end{pmatrix} \end{pmatrix};$$

The outer dimensions of the pair (**aBlockTiled$**, **bBlockTiled$**) are **mc** = (3 × (**mr** = 2)) = 6 (three rows of 2-row tiles in **aBlockTiled$**), **kc** = (3 × (**kr** = 4)) = 12 (three columns of 4-column tiles in **aBlockTiled$**, and three rows of 4-row tiles in **bBlockTiled$**), and **nc** = (5 × (**nr** = 2)) = 10 (five columns of 2-column tiles in **bBlockTiled$**). The inner dimensions are **mr** = 2, **kr** = 4, **nr** = 2, corresponding respectively to the row dimension, **mr**, of a left-multiplicand tile; to the column dimension, **kr**, of a left-multiplicand tile, equal to the row dimension of a right-multiplicand tile; and to the column dimension, **nr**, of a right-multiplicand tile.

Notice that **nr** must equal **mr** because tiles are of transposed shapes on the left and the right of our tile product. This restriction does not pertain to Kuzma's original Algorithm 1, only to our refactoring of it. The API has separate parameters for them for the sake of symmetry in the API, making it easier to remember.

Define **tileMul** as an iterated inner product of tiles and accumulated outer product within the tiles, then apply it to these examples:

In[24]:=

```
ClearAll[tileMul];
tileMul[ATiles_, BTiles_, mc_, kc_, nc_, mr_, kr_, nr_] :=
  Module[{tm, tk, tn, CTile, McByMr = mc/mr, KcByKr = kc/kr, NcByNr = nc/nr},
   CTile = ConstantArray[ConstantArray[0, {mr, nr}], {McByMr, NcByNr}];
   For[tm = 1, tm ≤ McByMr, tm++, (* for each row of A's tiles *)
    For[tn = 1, tn ≤ NcByNr, tn++, (* for each column in B's tiles *)
     For[tk = 1, tk ≤ KcByKr, tk++, (* iterated inner products of tiles *)
      (* ATiles⟦tm,tk⟧.BTiles⟦tk,tn⟧
       implicitly by accumulated outer product *)
      CTile⟦tm, tn⟧ += ATiles⟦tm, tk⟧.BTiles⟦tk, tn⟧]]];
   CTile];
```

In[26]:=

```
tileMul[aBlockTiled$, bBlockTiled$, 6, 12, 10, 2, 4, 2] // MatrixForm
```

Out[26]//MatrixForm=

$$\begin{pmatrix} \begin{pmatrix} 850 & 533 \\ 657 & 516 \end{pmatrix} & \begin{pmatrix} 918 & 872 \\ 593 & 692 \end{pmatrix} & \begin{pmatrix} 700 & 733 \\ 739 & 496 \end{pmatrix} & \begin{pmatrix} 737 & 778 \\ 592 & 601 \end{pmatrix} & \begin{pmatrix} 582 & 696 \\ 541 & 748 \end{pmatrix} \\ \begin{pmatrix} 901 & 541 \\ 624 & 650 \end{pmatrix} & \begin{pmatrix} 860 & 745 \\ 656 & 813 \end{pmatrix} & \begin{pmatrix} 770 & 656 \\ 833 & 610 \end{pmatrix} & \begin{pmatrix} 731 & 778 \\ 858 & 607 \end{pmatrix} & \begin{pmatrix} 539 & 866 \\ 629 & 1004 \end{pmatrix} \\ \begin{pmatrix} 862 & 585 \\ 989 & 608 \end{pmatrix} & \begin{pmatrix} 803 & 1066 \\ 784 & 968 \end{pmatrix} & \begin{pmatrix} 949 & 703 \\ 811 & 747 \end{pmatrix} & \begin{pmatrix} 780 & 826 \\ 710 & 751 \end{pmatrix} & \begin{pmatrix} 618 & 1000 \\ 735 & 852 \end{pmatrix} \end{pmatrix}$$

To check this result against a straightforward matrix product, we must flatten the tile level.

untileBlock

Is the result above equivalent to the matrix product **aBlockTiled$**.**bBlockTiled$**? First define **untileBlock**, which does exactly what its name says.

In[27]:=

```
ClearAll[untileBlock];
untileBlock[ATiledBlock_, mr_, mc_, kr_, kc_] :=
  (* Produce 1 mc×kc block from its tiles, each mr×kr. *)
  Module[{ABlock = ConstantArray[0, {mc, kc}], tileI, tileJ, inI, inJ, bm, bk},
   For[bm = 1, bm ≤ mc, bm++,
    For[bk = 1, bk ≤ kc, bk++,
     tileI = 1 + Quotient[(bm - 1), mr];
     tileJ = 1 + Quotient[(bk - 1), kr];
     inI = 1 + Mod[(bm - 1), mr];
     inJ = 1 + Mod[(bk - 1), kr];
     ABlock⟦bm, bk⟧ = ATiledBlock⟦tileI, tileJ, inI, inJ⟧]];
   ABlock];
```

Apply **untileBlock** to **aBlockTiled$** and to **bBlockTiled$**., compute the matrix product via Wolfram’s built-in, then visually check that the untiled matrices match their tiled brethren above.

In[29]:=
```
(aBlock$ = untileBlock[aBlockTiled$, 2, 6, 4, 12]) // MatrixForm
(bBlock$ = untileBlock[bBlockTiled$, 4, 12, 2, 10]) // MatrixForm
(cBlock$ = aBlock$.bBlock$) // MatrixForm
```

Out[29]//MatrixForm=

$$\begin{pmatrix} 15 & 14 & 10 & 15 & 6 & 0 & 3 & 8 & 3 & 14 & 1 & 1 \\ 3 & 9 & 8 & 5 & 5 & 10 & 9 & 6 & 6 & 1 & 14 & 4 \\ 11 & 14 & 9 & 10 & 12 & 2 & 0 & 15 & 5 & 4 & 14 & 1 \\ 11 & 13 & 3 & 3 & 5 & 3 & 15 & 13 & 13 & 4 & 7 & 3 \\ 12 & 7 & 14 & 14 & 2 & 7 & 15 & 15 & 5 & 0 & 9 & 1 \\ 15 & 6 & 14 & 13 & 1 & 13 & 0 & 8 & 14 & 5 & 2 & 10 \end{pmatrix}$$

Out[30]//MatrixForm=

$$\begin{pmatrix} 9 & 4 & 9 & 13 & 6 & 7 & 12 & 3 & 4 & 11 \\ 4 & 1 & 14 & 12 & 7 & 11 & 8 & 4 & 8 & 11 \\ 11 & 9 & 9 & 13 & 10 & 3 & 0 & 13 & 5 & 4 \\ 15 & 3 & 12 & 13 & 7 & 15 & 3 & 12 & 5 & 5 \\ 13 & 15 & 7 & 0 & 1 & 4 & 15 & 5 & 5 & 8 \\ 14 & 2 & 10 & 15 & 6 & 6 & 11 & 6 & 14 & 10 \\ 0 & 13 & 3 & 15 & 15 & 6 & 15 & 6 & 8 & 15 \\ 10 & 0 & 8 & 7 & 11 & 7 & 11 & 11 & 5 & 15 \\ 3 & 14 & 0 & 3 & 5 & 3 & 5 & 2 & 9 & 12 \\ 10 & 10 & 14 & 5 & 9 & 7 & 11 & 15 & 11 & 0 \\ 7 & 5 & 6 & 1 & 13 & 2 & 1 & 10 & 2 & 7 \\ 10 & 8 & 0 & 3 & 14 & 12 & 7 & 5 & 8 & 6 \end{pmatrix}$$

Out[31]//MatrixForm=

$$\begin{pmatrix} 850 & 533 & 918 & 872 & 700 & 733 & 737 & 778 & 582 & 696 \\ 657 & 516 & 593 & 692 & 739 & 496 & 592 & 601 & 541 & 748 \\ 901 & 541 & 860 & 745 & 770 & 656 & 731 & 778 & 539 & 866 \\ 624 & 650 & 656 & 813 & 833 & 610 & 858 & 607 & 629 & 1004 \\ 862 & 585 & 803 & 1066 & 949 & 703 & 780 & 826 & 618 & 1000 \\ 989 & 608 & 784 & 968 & 811 & 747 & 710 & 751 & 735 & 852 \end{pmatrix}$$

blockIt, tileIt

We now know how to multiply blocks full of snug tiles. We need to partition general matrices into snug blocks, in-turn partitioned into snug tiles. The dimensions of the snug blocks must divide the dimensions of the matrices, but that is the only restriction. If the matrices don’t snugly contain blocks, pad out the matrices in a pre-processing step. We do not consider that padding step in this paper.

Define a pair of functions, **blockIt** and **tileIt**, that, respectively, produce a blocked matrix and a tiled block.

In[32]:=
```
ClearAll[blockIt, tileIt];

blockIt[A_, mc_, M_, kc_, K_] := Table[
   A⟦m ;; m + mc - 1, k ;; k + kc - 1⟧, {m, 1, M, mc}, {k, 1, K, kc}];

tileIt[ABlock_, mr_, mc_, kr_, kc_] := Table[
   ABlock⟦m ;; m + mr - 1, k ;; k + kr - 1⟧, {m, 1, mc, mr}, {k, 1, kc, kr}];
```

Iterate **tileIt** over the result of **blockIt** on a matrix to get a doubly partitioned blocked and tiled matrix. Below is an example. Notice we build the dimensions bottom-up to ensure integer divisibility and to avoid padding. The regular structure in the displays is evident and instructive. Strive to see how 2D iterations of **tileMul** produces desired results.

In[35]:=

```
With[{bitCount = 4},
 With[{mr = 2, kr = 4, nr = 2}, (* -- tiles *)
  With[{mc = 2 mr, kc = 2 kr, nc = 2 nr}, (* mc=4, kc=8, nc=4 -- blocks *)
   With[{M = 2 mc, K = 2 kc, N = 2 nc}, (* M=8, K=16, N=8, -- original dims *)
    With[{A = RandomInteger[{0, 2^bitCount - 1}, {M, K}],
      B = RandomInteger[{0, 2^bitCount - 1}, {K, N}]},
     Module[{
       ABlocked = blockIt[A, mc, M, kc, K],
       BBlocked = blockIt[B, kc, K, nc, N],
       ATiled, BTiled},
      ATiled =
       Table[tileIt[ABlocked[[bm, bk]], mr, mc, kr, kc], {bm, 1, M/mc}, {bk, 1, K/kc}];
      BTiled =
       Table[tileIt[BBlocked[[bk, bn]], kr, kc, nr, nc], {bk, 1, K/kc}, {bn, 1, N/nc}];
      Column[{(* displays *)
        ATiled // MatrixForm,
        BTiled // MatrixForm,
        <|"dim[A]" → Dimensions[A],
           "dim[B]" → Dimensions[B],
           "dim[A_blocked]" → Dimensions[ABlocked],
           "dim[B_blocked]" → Dimensions[BBlocked],
           "A_tiled" → Dimensions[ATiled],
           "B_tiled" → Dimensions[BTiled],
           "bits" → bitCount,
           "mr" → mr, "kr" → kr, "nr" → nr,
           "mc" → mc, "kc" → kc, "nc" → nc,
           "M" → M, "K" → K, "N" → N|> // Print;}]]]]]]
```

```
<|dim[A] → {8, 16}, dim[B] → {16, 8}, dim[A_blocked] → {2, 2, 4, 8},
 dim[B_blocked] → {2, 2, 8, 4}, A_tiled → {2, 2, 2, 2, 2, 4}, B_tiled → {2, 2, 2, 2, 4, 2},
 bits → 4, mr → 2, kr → 4, nr → 2, mc → 4, kc → 8, nc → 4, M → 8, K → 16, N → 8|>
```

Out[35]=

$$\begin{pmatrix} \begin{pmatrix} \begin{pmatrix} 15 & 1 & 15 & 9 \\ 1 & 12 & 11 & 14 \end{pmatrix} & \begin{pmatrix} 0 & 10 & 1 & 7 \\ 0 & 0 & 5 & 9 \end{pmatrix} \\ \begin{pmatrix} 5 & 4 & 1 & 7 \\ 13 & 2 & 14 & 9 \end{pmatrix} & \begin{pmatrix} 4 & 7 & 10 & 15 \\ 10 & 7 & 6 & 4 \end{pmatrix} \end{pmatrix} & \begin{pmatrix} \begin{pmatrix} 2 & 0 & 7 & 1 \\ 5 & 15 & 12 & 14 \end{pmatrix} & \begin{pmatrix} 1 & 14 & 9 & 2 \\ 13 & 14 & 13 & 12 \end{pmatrix} \\ \begin{pmatrix} 1 & 12 & 10 & 8 \\ 7 & 12 & 5 & 10 \end{pmatrix} & \begin{pmatrix} 15 & 11 & 12 & 12 \\ 6 & 1 & 0 & 3 \end{pmatrix} \end{pmatrix} \\ \begin{pmatrix} \begin{pmatrix} 10 & 12 & 11 & 7 \\ 1 & 14 & 5 & 11 \end{pmatrix} & \begin{pmatrix} 2 & 8 & 0 & 1 \\ 9 & 2 & 10 & 0 \end{pmatrix} \\ \begin{pmatrix} 2 & 11 & 2 & 8 \\ 0 & 13 & 1 & 14 \end{pmatrix} & \begin{pmatrix} 6 & 15 & 11 & 14 \\ 12 & 8 & 13 & 12 \end{pmatrix} \end{pmatrix} & \begin{pmatrix} \begin{pmatrix} 8 & 5 & 15 & 15 \\ 11 & 15 & 0 & 8 \end{pmatrix} & \begin{pmatrix} 13 & 11 & 9 & 14 \\ 11 & 15 & 2 & 9 \end{pmatrix} \\ \begin{pmatrix} 11 & 5 & 7 & 9 \\ 10 & 13 & 9 & 11 \end{pmatrix} & \begin{pmatrix} 9 & 15 & 13 & 8 \\ 8 & 10 & 9 & 14 \end{pmatrix} \end{pmatrix} \end{pmatrix}$$

$$\begin{pmatrix} \begin{pmatrix} \begin{pmatrix} 9 & 8 \\ 7 & 0 \\ 7 & 13 \\ 7 & 9 \end{pmatrix} & \begin{pmatrix} 7 & 10 \\ 15 & 9 \\ 6 & 14 \\ 0 & 6 \end{pmatrix} \\ \begin{pmatrix} 5 & 6 \\ 6 & 1 \\ 1 & 10 \\ 9 & 1 \end{pmatrix} & \begin{pmatrix} 14 & 1 \\ 5 & 3 \\ 12 & 11 \\ 9 & 5 \end{pmatrix} \end{pmatrix} & \begin{pmatrix} \begin{pmatrix} 10 & 1 \\ 1 & 4 \\ 13 & 9 \\ 13 & 0 \end{pmatrix} & \begin{pmatrix} 6 & 6 \\ 10 & 1 \\ 4 & 11 \\ 10 & 12 \end{pmatrix} \\ \begin{pmatrix} 6 & 10 \\ 13 & 15 \\ 8 & 9 \\ 14 & 3 \end{pmatrix} & \begin{pmatrix} 1 & 14 \\ 5 & 3 \\ 0 & 9 \\ 4 & 7 \end{pmatrix} \end{pmatrix} \\ \begin{pmatrix} \begin{pmatrix} 0 & 11 \\ 15 & 14 \\ 9 & 4 \\ 14 & 3 \end{pmatrix} & \begin{pmatrix} 9 & 9 \\ 1 & 8 \\ 7 & 12 \\ 4 & 11 \end{pmatrix} \\ \begin{pmatrix} 4 & 5 \\ 3 & 1 \\ 2 & 11 \\ 8 & 5 \end{pmatrix} & \begin{pmatrix} 0 & 8 \\ 8 & 2 \\ 15 & 12 \\ 11 & 5 \end{pmatrix} \end{pmatrix} & \begin{pmatrix} \begin{pmatrix} 0 & 10 \\ 7 & 5 \\ 7 & 6 \\ 9 & 8 \end{pmatrix} & \begin{pmatrix} 11 & 9 \\ 2 & 10 \\ 3 & 13 \\ 15 & 13 \end{pmatrix} \\ \begin{pmatrix} 6 & 1 \\ 10 & 12 \\ 7 & 13 \\ 15 & 12 \end{pmatrix} & \begin{pmatrix} 6 & 15 \\ 14 & 15 \\ 6 & 12 \\ 9 & 15 \end{pmatrix} \end{pmatrix} \end{pmatrix}$$

unBlock

unBlock is exactly parallel to **untileBlock**. It does not need a unit test or an illustrative example.

In[36]:=

```
ClearAll[unblock];

unblock[ABlocked_, mc_, M_, kc_, K_] :=
  Module[{A = ConstantArray[0, {M, K}], blockI, blockJ, inI, inJ, m, k},
   For[m = 1, m ≤ M, m++,
    For[k = 1, k ≤ K, k++,
     blockI = 1 + Quotient[(m - 1), mc];
     blockJ = 1 + Quotient[(k - 1), kc];
     inI = 1 + Mod[(m - 1), mc];
     inJ = 1 + Mod[(k - 1), kc];
     A〚m, k〛 = ABlocked〚blockI, blockJ, inI, inJ〛]];
   A];
```

blockTileMul, blockMul

blockTileMul is the intermediate target of compilation, after matrices have been blocked and tiled as described above. **blockTileMul** calls **tileMul** at bottom.

For testing, we include a **blockMul** routine for blocked-but-not-tiled matrices: the untiled results of **blockTileMul** must match the results of **blockMul**, and the unblocked results must match the results of Mathematica's built-in matrix multiplication. The following defines **blockTileMul** and **blockMul**, then **Asserts** the requirements on an example.

In[38]:=
```
On[Assert];
ClearAll[blockMul, blockTileMul];

blockTileMul[ABlocks_, BBlocks_, M_, K_, N_, mc_, kc_, nc_, mr_, kr_, nr_] :=
  (* ABlocks is an array of mc×kc blocks, BBlock of kc×nc blocks. *)
  Module[
   {bm, bk, bn, MByMc = M/mc, KByKc = K/kc, NByNc = N/nc, McByMr = mc/nr, NcByNr = nc/nr,
    CTiled, ATiles, BTiles},
   CTiled = ConstantArray[ConstantArray[ConstantArray[0, {mr, nr}],
      {McByMr, NcByNr}], {MByMc, NByNc}];
   (* for each input block *)
   For[bm = 1, bm ≤ MByMc, bm++,
    For[bn = 1, bn ≤ NByNc, bn++,
     (* iterated inner product *)
     For[bk = 1, bk ≤ KByKc, bk++,
      ATiles = tileIt[ABlocks⟦bm, bk⟧, mr, mc, kr, kc];
      BTiles = tileIt[BBlocks⟦bk, bn⟧, kr, kc, nr, nc];
      CTiled⟦bm, bn⟧ += tileMul[ATiles, BTiles, mc, kc, nc, mr, kr, nr]]]];
   CTiled];

blockMul[ABlocks_, BBlocks_, M_, K_, N_, mc_, kc_, nc_, mr_, kr_, nr_] :=
  Module[
   {bm, bk, bn, MByMc = M/mc, KByKc = K/kc, NByNc = N/nc, McByMr = mc/nr, NcByNr = nc/nr,
    CBlocked},
   CBlocked = ConstantArray[ConstantArray[0, {mc, nc}], {MByMc, NByNc}];
   (* for each input block *)
   For[bm = 1, bm ≤ MByMc, bm++,
    For[bn = 1, bn ≤ NByNc, bn++,
     (* iterated inner product *)
     For[bk = 1, bk ≤ KByKc, bk++,
      CBlocked⟦bm, bn⟧ += ABlocks⟦bm, bk⟧.BBlocks⟦bk, bn⟧]]];
   CBlocked];
```

```
With[{bitCount = 4},
 With[{mr = 2, kr = 4, nr = 2}, (* -- tiles *)
  With[{mc = 3 mr, kc = 3 kr, nc = 5 nr}, (* mc=6, kc=12, nc=10 -- blocks *)
   With[{M = 5 mc, K = 3 kc, N = 3 nc}, (* M=30, K=36, N=30, -- original dims *)
    With[{A = RandomInteger[{0, 2^bitCount - 1}, {M, K}],
      B = RandomInteger[{0, 2^bitCount - 1}, {K, N}]},
     Module[{
       ABlocks = blockIt[A, mc, M, kc, K],
       BBlocks = blockIt[B, kc, K, nc, N],
       CTiled, CBlocked, CBlockedCheck, C, CCheck, bm, bk, bn, tm, tk, tn},
      CTiled = blockTileMul[ABlocks, BBlocks, M, K, N, mc, kc, nc, mr, kr, nr];
      (* Check intermediate forms. *)
      CBlocked =
       Table[untileBlock[CTiled[[m, n]], mr, mc, nr, nc], {m, 1, M/mc}, {n, 1, N/nc}];
      CBlockedCheck = blockMul[ABlocks, BBlocks, M, K, N, mc, kc, nc, mr, kr, nr];
      Assert[CBlockedCheck === CBlocked];
      C = unblock[CBlocked, mc, M, nc, N];
      CCheck = A.B;
      Assert[CCheck === C];
      Column[{(* displays *)
        A // MatrixForm;
        B // MatrixForm;
        ABlocks // MatrixForm;
        BBlocks // MatrixForm;
        CTiled // MatrixForm,
        CBlockedCheck // MatrixForm;
        CBlocked // MatrixForm;
        C // MatrixForm,
        <|"dim[A]" → Dimensions[A],
           "dim[B]" → Dimensions[B],
           "dim[C_tiled]" → Dimensions[CTiled],
           "dim[A_blocks]" → Dimensions[ABlocks],
           "dim[B_blocks]" → Dimensions[BBlocks],
           "dim[C]" → Dimensions[C],
           "bits" → bitCount,
           "mr" → mr, "kr" → kr, "nr" → nr,
           "mc" → mc, "kc" → kc, "nc" → nc,
           "M" → M, "K" → K, "N" → N|> // Print;}]]]]]]
```

<|dim[A] → {30, 36}, dim[B] → {36, 30}, dim[C_{tiled}] → {5, 3, 3, 5, 2, 2}, dim[A_{blocks}] → {5, 3, 6, 12}, dim[B_{blocks}] → {3, 3, 12, 10}, dim[C] → {30, 30}, bits → 4, mr → 2, kr → 4, nr → 2, mc → 6, kc → 12, nc → 10, M → 30, K → 36, N → 30|>

Out[42]=

```
2076 2080   1886 2374   2152 2348   2190 2689   2464 2099    2245 241
1312 1509   1355 1710   1478 1593   1472 1570   1470 1469    1313 166
1486 1851   1635 1946   1741 1642   1732 2340   1924 1854    2060 199
1831 2331   2188 2529   2143 2209   2305 2765   2674 1995    2178 255
1806 1697   1440 2225   1848 1797   1845 2281   1886 1699    1810 199
1806 1954   1764 2152   2021 2020   2169 2431   2195 1997    2146 229

1817 1720   1622 2044   1921 1705   2028 2370   2214 1784    1997 231
1821 2183   1751 2283   2276 2181   2162 2374   2384 1793    2012 220
2033 2120   2104 2462   1875 1993   2110 2345   2322 2076    2084 222
1877 2275   2131 2457   1956 2347   2247 2462   2416 2099    2272 247
2073 2311   1775 2257   1923 2371   2153 2116   2141 1938    2059 230
1801 2147   1927 2562   2326 2121   2244 2498   2209 2267    2143 233

1672 1802   1791 1891   1840 1794   1796 1756   1915 1778    1934 208
1477 2049   1701 1940   1968 1946   1715 1983   2148 1525    2058 201
1707 1815   1667 2060   1791 2002   1887 2274   2051 1889    1848 220
1576 1638   1660 1822   1866 1693   1982 2090   1929 1838    2060 185
1204 1643   1416 1737   1390 1698   1491 1882   1790 1420    1160 172
1680 1867   1815 2132   1727 1933   2004 2308   1941 1998    1784 210

1642 1910   1500 1952   1460 1908   1813 2143   1846 1561    1648 197
1721 2191   2097 2493   2012 1969   2192 2569   2346 2212    2137 235
1880 1807   1721 2063   1831 1892   1832 2242   1823 1862    1942 206
1759 1511   1595 1931   1744 1588   1771 2104   1971 1409    1766 177
1847 2111   1874 2254   2041 2102   2071 2204   2052 1777    2206 240
1942 1904   2034 2248   1946 1871   2085 2303   2379 1720    1880 212

1836 1823   1773 2065   1913 1794   1710 2077   2067 1989    1938 206
1571 1880   2008 2027   2121 1774   2041 2085   2109 2027    1925 234
1663 2223   2082 2328   1741 1838   2155 2433   2087 2007    2038 213
1845 2210   2068 2402   2110 2188   2109 2593   2300 2243    2212 252
1921 2162   2168 2196   2199 2163   2253 2522   2485 2107    2267 240
1808 2321   2105 2120   1823 2299   2057 1982   2410 1600    1938 214

2076 2080 1886 2374 2152 2348 2190 2689 2464 2099 2245 2412 2328 2091 2323 1
1312 1509 1355 1710 1478 1593 1472 1570 1470 1469 1313 1669 1299 1317 1433 1
1486 1851 1635 1946 1741 1642 1732 2340 1924 1854 2060 1990 1903 1598 1991 1
1831 2331 2188 2529 2143 2209 2305 2765 2674 1995 2178 2552 2023 1927 2137 1
1806 1697 1440 2225 1848 1797 1845 2281 1886 1699 1810 1995 1830 1734 1821 1
1806 1954 1764 2152 2021 2020 2169 2431 2195 1997 2146 2293 1955 1793 2414 2
1817 1720 1622 2044 1921 1705 2028 2370 2214 1784 1997 2315 1692 1799 2010 1
1821 2183 1751 2283 2276 2181 2162 2374 2384 1793 2012 2209 1985 1796 2308 1
2033 2120 2104 2462 1875 1993 2110 2345 2322 2076 2084 2229 1959 1950 1962 1
1877 2275 2131 2457 1956 2347 2247 2462 2416 2099 2272 2474 2268 1742 2085 2
2073 2311 1775 2257 1923 2371 2153 2116 2141 1938 2059 2304 1909 2099 1999 1
1801 2147 1927 2562 2326 2121 2244 2498 2209 2267 2143 2336 2215 1738 2306 2
1672 1802 1791 1891 1840 1794 1796 1756 1915 1778 1934 2081 1508 1512 1887 1
1477 2049 1701 1940 1968 1946 1715 1983 2148 1525 2058 2019 1619 1547 1916 1
1707 1815 1667 2060 1791 2002 1887 2274 2051 1889 1848 2204 1802 1588 1840 1
1576 1638 1660 1822 1866 1693 1982 2090 1929 1838 2060 1855 1636 1656 2088 1
1204 1643 1416 1737 1390 1698 1491 1882 1790 1420 1160 1722 1737 1339 1766 1
1680 1867 1815 2132 1727 1933 2004 2308 1941 1998 1784 2107 1910 1568 1972 1
```

1642	1910	1500	1952	1460	1908	1813	2143	1846	1561	1648	1976	1732	1866	1965	1
1721	2191	2097	2493	2012	1969	2192	2569	2346	2212	2137	2356	2163	2069	2085	2
1880	1807	1721	2063	1831	1892	1832	2242	1823	1862	1942	2064	1808	1966	2125	1
1759	1511	1595	1931	1744	1588	1771	2104	1971	1409	1766	1773	1460	1790	1900	1
1847	2111	1874	2254	2041	2102	2071	2204	2052	1777	2206	2409	1668	1960	2228	1
1942	1904	2034	2248	1946	1871	2085	2303	2379	1720	1880	2125	1786	1845	2027	1
1836	1823	1773	2065	1913	1794	1710	2077	2067	1989	1938	2061	1836	1585	1881	1
1571	1880	2008	2027	2121	1774	2041	2085	2109	2027	1925	2340	1718	1479	2238	1
1663	2223	2082	2328	1741	1838	2155	2433	2087	2007	2038	2135	2051	1767	2085	2
1845	2210	2068	2402	2110	2188	2109	2593	2300	2243	2212	2529	2054	1757	2596	1
1921	2162	2168	2196	2199	2163	2253	2522	2485	2107	2267	2409	2172	1876	2509	2
1808	2321	2105	2120	1823	2299	2057	1982	2410	1600	1938	2142	1949	1644	1946	1

Fitting the APU

For the Gemini-I APU, a common tile size is 1×2048, a half-bank's worth of 16-bit data. **tileMul** can perform 16-bit by 16-bit multiplication from two input half-banks and leave the 32-bit result in another pair of half banks. Other plausible choices for the width of a tile are 32 768, corresponding to 16 half banks in a single APUC core, or 128 Kib, corresponding to 64 half banks in four cores of an entire APU.

Notice that 16-bit integer matrix multiplication will not overflow 32 bits.

The following example shows multiplication of 16-bit matrices in tiles of dimension 1×2048. The left-hand multiplicand, A, has dimensions $15 \times 18\,432$ and the right-hand multiplicand, B, has dimensions $18\,432 \times 15$. The output is 15×15 by 32 bits and fits at the left of two VRs in main memory, or in subsequent columns (*plats*) of one VR in main memory.

A is partitioned into blocks of dimension 3×6144, requiring 9 VRs out of 53 available in L1. B is partitioned into blocks of dimension 6144×5, requiring 15 VRs out of 53 available in L1. Blocks of A and B must be moved into VRs in main memory prior to multiplication. **blockTileMul** is responsible for data movement, for accumulating results in one or two VRs of main memory, and for moving final results back into L1 for later harvesting by the host computer. The prototype **blockTileMul** in this paper does no data movement, but it does perform blocked and tiled multiplication. The 15×15 display after the example can be visually checked.

In[43]:=

```
With[{bitCount = 16},
 With[{mr = 1, kr = 2048, nr = 1}, (* -- mr must equal nr -- tiles *)
  With[{mc = 3 mr, kc = 3 kr, nc = 5 nr}, (* mc=3, kc=6144, nc=5 -- blocks *)
   With[{M = 5 mc, K = 3 kc, N = 3 nc},
    (* M=15, K=18432, N=15, -- original dims *)
    With[{A = RandomInteger[{0, 2^bitCount - 1}, {M, K}],
      B = RandomInteger[{0, 2^bitCount - 1}, {K, N}]},
     Module[{
       ABlocks = blockIt[A, mc, M, kc, K],
       BBlocks = blockIt[B, kc, K, nc, N],
       CTiled, CBlocked, CBlockedCheck, C, CCheck, bm, bk, bn, tm, tk, tn},
      CTiled = blockTileMul[ABlocks, BBlocks, M, K, N, mc, kc, nc, mr, kr, nr];
      (* Check intermediate forms. *)
      CBlocked =
       Table[untileBlock[CTiled[[m, n]], mr, mc, nr, nc], {m, 1, M/mc}, {n, 1, N/nc}];
      CBlockedCheck = blockMul[ABlocks, BBlocks, M, K, N, mc, kc, nc, mr, kr, nr];
      Assert[CBlockedCheck === CBlocked];
      C = unblock[CBlocked, mc, M, nc, N];
      CCheck = A.B;
      Assert[CCheck === C];
      Column[{(* displays *)
        CTiled // MatrixForm,
        C // MatrixForm,
        <|"dim[A]" → Dimensions[A],
           "dim[B]" → Dimensions[B],
           "dim[C_tiled]" → Dimensions[CTiled],
           "dim[A_blocks]" → Dimensions[ABlocks],
           "dim[B_blocks]" → Dimensions[BBlocks],
           "dim[C]" → Dimensions[C],
           "bits" → bitCount,
           "mr" → mr, "kr" → kr, "nr" → nr,
           "mc" → mc, "kc" → kc, "nc" → nc,
           "M" → M, "K" → K, "N" → N|> // Print;}]]]]]]]
```

```
<|dim[A] → {15, 18432}, dim[B] → {18432, 15}, dim[C_tiled] → {5, 3, 3, 5, 1, 1},
 dim[A_blocks] → {5, 3, 3, 6144}, dim[B_blocks] → {3, 3, 6144, 5}, dim[C] → {15, 15},
 bits → 16, mr → 1, kr → 2048, nr → 1, mc → 3, kc → 6144, nc → 5, M → 15, K → 18432, N → 15|>
```

Out[43]=

```
( ( ( 19 579 727 656 849 )  ( 19 545 781 952 476 )  ( 19 696 525 885 901 )  ( 19 870 041 337 062 )  (
    ( 19 685 315 881 210 )  ( 19 585 699 218 462 )  ( 19 634 615 148 962 )  ( 19 828 249 931 462 )  (
    ( 19 639 492 727 845 )  ( 19 616 191 821 426 )  ( 19 732 365 640 531 )  ( 19 858 988 701 565 )  (
  ( ( 19 627 808 195 669 )  ( 19 577 678 529 046 )  ( 19 667 009 810 401 )  ( 19 862 906 029 110 )  (
    ( 19 795 558 192 753 )  ( 19 643 693 177 318 )  ( 19 878 936 707 583 )  ( 19 945 797 391 805 )  (
    ( 19 719 867 655 207 )  ( 19 688 913 334 546 )  ( 19 809 308 771 006 )  ( 19 875 093 166 170 )  (
  ( ( 19 667 848 242 494 )  ( 19 595 842 228 725 )  ( 19 678 188 468 505 )  ( 19 809 687 726 739 )  (
    ( 19 939 517 308 967 )  ( 19 899 872 931 821 )  ( 20 001 360 131 779 )  ( 20 174 616 943 741 )  (
    ( 19 575 248 771 893 )  ( 19 569 437 868 364 )  ( 19 652 507 144 948 )  ( 19 712 898 468 158 )  (
  ( ( 19 832 892 323 076 )  ( 19 717 191 461 477 )  ( 19 942 261 363 792 )  ( 19 907 321 625 161 )  (
    ( 19 536 292 490 495 )  ( 19 640 474 920 486 )  ( 19 750 452 943 709 )  ( 19 770 361 192 788 )  (
    ( 19 729 784 758 928 )  ( 19 698 097 785 074 )  ( 19 799 240 630 071 )  ( 19 912 109 389 987 )  (
  ( ( 19 709 376 613 297 )  ( 19 659 137 860 693 )  ( 19 837 162 844 497 )  ( 19 891 076 021 674 )  (
    ( 19 754 763 994 172 )  ( 19 810 404 493 241 )  ( 19 804 758 601 127 )  ( 19 997 640 027 625 )  (
    ( 19 763 351 052 364 )  ( 19 765 315 947 707 )  ( 19 876 690 329 469 )  ( 19 946 041 096 201 )  (
( 19 579 727 656 849  19 545 781 952 476  19 696 525 885 901  19 870 041 337 062  19 514 566 361 4
  19 685 315 881 210  19 585 699 218 462  19 634 615 148 962  19 828 249 931 462  19 672 563 344 0
  19 639 492 727 845  19 616 191 821 426  19 732 365 640 531  19 858 988 701 565  19 585 607 465 6
  19 627 808 195 669  19 577 678 529 046  19 667 009 810 401  19 862 906 029 110  19 581 631 669 5
  19 795 558 192 753  19 643 693 177 318  19 878 936 707 583  19 945 797 391 805  19 677 543 432 9
  19 719 867 655 207  19 688 913 334 546  19 809 308 771 006  19 875 093 166 170  19 719 999 840 8
  19 667 848 242 494  19 595 842 228 725  19 678 188 468 505  19 809 687 726 739  19 548 911 490 1
  19 939 517 308 967  19 899 872 931 821  20 001 360 131 779  20 174 616 943 741  19 860 608 565 0
  19 575 248 771 893  19 569 437 868 364  19 652 507 144 948  19 712 898 468 158  19 491 531 672 2
  19 832 892 323 076  19 717 191 461 477  19 942 261 363 792  19 907 321 625 161  19 790 852 187 7
  19 536 292 490 495  19 640 474 920 486  19 750 452 943 709  19 770 361 192 788  19 565 951 228 1
  19 729 784 758 928  19 698 097 785 074  19 799 240 630 071  19 912 109 389 987  19 589 391 680 0
  19 709 376 613 297  19 659 137 860 693  19 837 162 844 497  19 891 076 021 674  19 612 719 425 6
  19 754 763 994 172  19 810 404 493 241  19 804 758 601 127  19 997 640 027 625  19 727 693 531 5
( 19 763 351 052 364  19 765 315 947 707  19 876 690 329 469  19 946 041 096 201  19 653 923 835 8
```

Direct Implementation of Algorithm 1

Our refactoring above is arguably easier to understand than the monolithic code of Algorithm 1. We now show, in steps, that they are equivalent. Along the way, we produce a new refactoring, closer to Algorithm 1, which includes correct-by-construction packing and tiling ratios.

Definitions

A block is a unit of storage optimized for caching. A tile is a unit of storage optimized for matrix multiplication. **mc** is the number of rows in a block. **kc** is the number of columns in a block. **mr** is the number of rows in a tile. **kr** is the number of columns in a tile. **mr** must divide **mc**, and **kr** must divide **kc**. Finally, M and K are the dimensions of a matrix that contains one or more blocks. **mc** must divide **M** and **kc** must divide **K**.

Indices

Mathematica indices are 1-based. We compute indices 0-based, then index arrays 1-based by incrementing 0-based indices *in situ*, that is, by adding 1 inside Mathematica’s *Part* expressions — doubled square brackets. All indices outside *Part* notations are 0-based.

Matrix Metadata

A major activity of software engineering is representing data economically, minimizing duplication and satisfying constraints by construction.

For block and tile operations, matrix metadata are (1) block and tile dimensions and (2) identifiers (names and UUIDs). Ratios represent block and matrix dimensions so that they automatically satisfy partitioning constraints (no overlapping, overhangs, or under-hangs).

UUIDs should be managed in a global registry. Here, to reduce the complexity of this specification, we do not manage UUIDs. User code is responsible for avoiding duplication of UUIDs.

In most programming languages, we’d represent annotated matrices as *structs* or *classes*. Mathematica does not have structs or classes, but has several equivalent mechanisms. We’ll employ the most elementary mechanism, *Associations*, similar to dictionaries in Python or hashmaps in Clojure.

Helper Function

Check that some datum is a UUID, for satisfying constraints.

In[44]:=
```
ClearAll[uuidQ];
uuidQ[candidate_] :=
 Module[{chars = Characters[candidate]},
   (chars⟦9⟧ === "-" === chars⟦14⟧ === chars⟦19⟧ === chars⟦24⟧) &&
    Module[{hexes = Select[chars,
        ToCharacterCode["0"]⟦1⟧ ≤ ToCharacterCode[#]⟦1⟧ ≤ ToCharacterCode["9"]⟦1⟧ |
          ToCharacterCode["a"]⟦1⟧ ≤
           ToCharacterCode[#]⟦1⟧ ≤ ToCharacterCode["f"]⟦1⟧ &]},
     Length[hexes] === 32 && StringQ[candidate]]]
uuidQ[CreateUUID[]]
```

Out[46]=
```
True
```

Factory Functions

A **blockTileMatrix** is a 2D array with suitable metadata. A **packedMatrix** is a 1D array with suitable metadata.

Running Example

Here is an example adapted from the Kuzma paper. It has 12 2×4 column-major tiles stored row-major in the block.

In[47]:=

$$\left(\text{utA\$} = \left(\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|}\hline 0 & 2 & 4 & 6 & 8 & 10 & 12 & 14 & a & c & e & g \\ \hline 1 & 3 & 5 & 7 & 9 & 11 & 13 & 15 & b & d & f & h \\ \hline i & k & m & o & q & s & u & w & \alpha & \gamma & \epsilon & \eta \\ \hline j & l & n & p & r & t & v & x & \beta & \delta & \zeta & \vartheta \\ \hline \iota & \lambda & \nu & o & \rho & \tau & \phi & \psi & \Delta & \Lambda & \Pi & \Phi \\ \hline \kappa & \mu & \xi & \pi & \sigma & \upsilon & \xi & \omega & \Theta & \Xi & \Sigma & \Omega \\ \hline \end{array}\right)\right) \text{ // MatrixForm}$$

Out[47]//MatrixForm=

$$\begin{pmatrix} 0 & 2 & 4 & 6 & 8 & 10 & 12 & 14 & a & c & e & g \\ 1 & 3 & 5 & 7 & 9 & 11 & 13 & 15 & b & d & f & h \\ i & k & m & o & q & s & u & w & \alpha & \gamma & \epsilon & \eta \\ j & l & n & p & r & t & v & x & \beta & \delta & \zeta & \vartheta \\ \iota & \lambda & \nu & o & \rho & \tau & \phi & \psi & \Delta & \Lambda & \Pi & \Phi \\ \kappa & \mu & \xi & \pi & \sigma & \upsilon & \xi & \omega & \Theta & \Xi & \Sigma & \Omega \end{pmatrix}$$

makeBlockTileMatrix annotates 2D matrix content. The storage order of elements in tiles must be the same for all tiles in the matrix. The storage order of tiles in blocks must be the same for all blocks in the matrix. All arguments are constrained, that is, type-checked.

In[48]:=

```
ClearAll[makeBlockTileMatrix];
makeBlockTileMatrix[
   mr_Integer, kr_Integer,
   mcByMr_Integer, kcByKr_Integer, (*dimensional ratios*)
   MByMc_Integer, KByKc_Integer, (*dimensional ratios*)
   ElementInTileStorageOrder_String /; (ElementInTileStorageOrder === "row" ||
      ElementInTileStorageOrder === "column"),
   TileInBlockStorageOrder_String /;
    (TileInBlockStorageOrder === "row" || TileInBlockStorageOrder === "column"),
   matrixContent_List, matrixName_String,
   uuid_ /; (uuid === Null || uuidQ[uuid])] :=
  <|"mr" → mr, "kr" → kr,
   "mc/mr" → mcByMr, "mc" → mr mcByMr,
   "kc/kr" → kcByKr, "kc" → kr kcByKr,
   "M/mc" → MByMc, "M" → mr mcByMr MByMc,
   "K/kc" → KByKc, "K" → kr kcByKr KByKc,
   "element-in-tile storage order" → ElementInTileStorageOrder,
   "tile-in-block storage order" → TileInBlockStorageOrder,
   "content" → matrixContent, "name" → matrixName,
   "uuid" → If[uuid === Null, CreateUUID[], uuid],
   "type" → "blockTileMatrix"|>;
btm$ = makeBlockTileMatrix[
  2(*mr*), 4(*kr*), 3(*mc/mr*), 3(*kc/kr*), 1(*M/mc*), 1(*K/kc*),
  "column"(*element-in-tile storage order*),
  "row"(*tile-in-block storage order*),
  utA$(*content*),
  "unit-test matrix"(*name -- could be empty*),
  "e5bdae0c-0ba8-4ad7-95ef-d3fa13ad8920"(*UUID -- could be Null*)]
```

Out[50]=

```
<|mr → 2, kr → 4, mc/mr → 3, mc → 6, kc/kr → 3, kc → 12, M/mc → 1,
 M → 6, K/kc → 1, K → 12, element-in-tile storage order → column,
 tile-in-block storage order → row, content →
  {{0, 2, 4, 6, 8, 10, 12, 14, a, c, e, g}, {1, 3, 5, 7, 9, 11, 13, 15, b, d, f, h},
   {i, k, m, o, q, s, u, w, α, γ, ϵ, η}, {j, l, n, p, r, t, v, x, β, δ, ζ, θ},
   {ι, λ, ν, ο, ρ, τ, ϕ, ψ, Δ, Λ, Π, Φ}, {κ, μ, ξ, π, σ, υ, ξ, ω, Θ, Ξ, Σ, Ω}},
 name → unit-test matrix, uuid → e5bdae0c-0ba8-4ad7-95ef-d3fa13ad8920,
 type → blockTileMatrix|>
```

makePackedMatrix annotates 1D packed matrix content. The storage-order metadata drives packing and unpacking processes. There are four possibilities: the element-in-tile storage order can be *column* or *row*, and, independently, the tile-in-block storage order can be *column* or *row*.

In[51]:=
```
ClearAll[makePackedMatrix];
makePackedMatrix[len_Integer /; (len ≥ 0),
  ElementInTileStorageOrder_String /; (ElementInTileStorageOrder === "row" ||
     ElementInTileStorageOrder === "column"),
  TileInBlockStorageOrder_String /;
   (TileInBlockStorageOrder === "row" || TileInBlockStorageOrder === "column"),
  content_List, name_String, uuid_ /; (uuid === Null || uuidQ[uuid])] :=
 <|"len" → len,
  "element-in-tile storage order" → ElementInTileStorageOrder,
  "tile-in-block storage order" → TileInBlockStorageOrder,
  "content" → content, "name" → name,
  "uuid" → If[uuid === Null, CreateUUID[], uuid],
  "type" → "packedMatrix"|>
```

Type Checkers

In[53]:=
```
ClearAll[blockTileMatrixQ, packedMatrixQ];
blockTileMatrixQ[it_Association] := it["type"] === "blockTileMatrix";
blockTileMatrixQ[btm$]
packedMatrixQ[it_Association] := it["type"] === "packedMatrix"
```

Out[55]=
```
True
```

packBlock

Packs one block from a block matrix into a 1D array, following the storage orders specified in the source block matrix. The unit test has column order for elements in tiles and row order for tiles in blocks.

In[57]:=

```
ClearAll[packBlock];
packBlock[m_?blockTileMatrixQ,
   (*Pick a single block via 0-
    based block indices from the middle of a blockTileMatrix*)
   fromRowBlock_Integer, fromColumnBlock_Integer,
   optionalName_String, uuid_ /; (uuid === Null || uuidQ[uuid])] :=
  With[{mr = m["mr"], kr = m["kr"]},
   With[{mc = mr * m["mc/mr"], kc = kr * m["kc/kr"]},
    With[{i = fromRowBlock * mc, k = fromColumnBlock * mc, A = m["content"]},
     Module[{ii, kk, tk, tm, Ai = 0, len = mc kc, APack},
      APack = ConstantArray[0, len];
      (* tile iteration -- for each tile in the block *)
      If[m["tile-in-block storage order"] === "row",
       For[ii = i, ii < i + mc, ii += mr,
        For[kk = k, kk < k + kc, kk += kr,
         If[m["element-in-tile storage order"] === "column",
          For[tk = kk , tk < (kk + kr), tk++,
           For[tm = ii, tm < (ii + mr), tm++,
            APack[[1 + Ai]] = A[[1 + tm, 1 + tk]]; Ai++]],
          If[m["element-in-tile storage order"] === "row",
           For[tm = ii, tm < (ii + mr), tm++,
            For[tk = kk , tk < (kk + kr), tk++,
             APack[[1 + Ai]] = A[[1 + tm, 1 + tk]]; Ai++]],
           Throw[752]]]]],
       If[m["tile-in-block storage order"] === "column",
        For[kk = k, kk < k + kc, kk += kr,
         For[ii = i, ii < i + mc, ii += mr,
          If[m["element-in-tile storage order"] === "column",
           For[tk = kk , tk < (kk + kr), tk++,
            For[tm = ii, tm < (ii + mr), tm++,
             APack[[1 + Ai]] = A[[1 + tm, 1 + tk]]; Ai++]],
           If[m["element-in-tile storage order"] === "row",
            For[tm = ii, tm < (ii + mr), tm++,
             For[tk = kk , tk < (kk + kr), tk++,
              APack[[1 + Ai]] = A[[1 + tm, 1 + tk]]; Ai++]],
            Throw[752]]]]],
        Throw[753]]];
      makePackedMatrix[len,
       m["element-in-tile storage order"], m["tile-in-block storage order"],
       APack(*content*), optionalName, uuid]]]]];
btp$ =
 packBlock[btm$, 0, 0, "packed btm", "d7a1c986-e579-4a01-9713-a1475c791587"]
```

Out[59]=

```
<|len → 72, element-in-tile storage order → column,
 tile-in-block storage order → row,
 content → {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, a, b, c, d, e, f, g, h,
   i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, α, β, γ, δ, ϵ, ζ, η, θ, ι, κ, λ, μ,
   ν, ξ, ο, π, ρ, σ, τ, υ, ϕ, ξ, ψ, ω, Δ, Θ, Λ, Ξ, Π, Σ, Φ, Ω}, name → packed btm,
 uuid → d7a1c986-e579-4a01-9713-a1475c791587, type → packedMatrix|>
```

unpackBlock

Unpack a 1D packedMatrix array into a given block-location inside a block matrix, mutating that content. The unit test shows round-tripping column order for elements in tiles and row order for tiles in blocks. The visualization also shows blocks divided by solid lines and tiles divided by dashed lines along with a UUID check.

In[60]:=

```
ClearAll[griddit];
griddit[m_List, mr_, kr_, mcByMr_, kcByKr_] :=
  With[{mc = mr mcByMr, kc = kr kcByKr},
   Module[{kd = ConstantArray[False, kc], md = ConstantArray[False, mc], mm, kk},
    kd[[kc]] = True; md[[mc]] = True;
    For[mm = mr, mm < mc, mm += mr, md[[mm]] = Dashed];
    For[kk = kr, kk < kc, kk += kr, kd[[kk]] = Dashed];
    Grid[m, Dividers → {{True, kd, True}, {True, md, True}}]]];
ClearAll[unpackBlock];
unpackBlock[
   dest_Symbol /; blockTileMatrixQ[dest], (*pattern for mutability*)
   m_?packedMatrixQ,
   toRowBlock_Integer, toColumnBlock_Integer,
   optionalName_String, uuid_ /; (uuid === Null || uuidQ[uuid])] :=
  (On[Assert];
   Assert[m["element-in-tile storage order"] ===
     dest["element-in-tile storage order"]];
   Assert[
    m["tile-in-block storage order"] === dest["tile-in-block storage order"]];
   With[{mr = dest["mr"], kr = dest["kr"]},
    With[{mc = mr dest["mc/mr"], kc = kr dest["kc/kr"]},
     With[{i = toRowBlock mc, k = toRowBlock kc, APack = m["content"]},
      Module[{ii, kk, tk, tm, Ai = 0, len = mc kc, A = dest["content"]},
       (* tile iteration -- for each tile in the block *)
       If[m["tile-in-block storage order"] === "row",
        For[ii = i, ii < i + mc, ii += mr,
         For[kk = k, kk < k + kc, kk += kr, (* element iteration *)
          If[m["element-in-tile storage order"] === "column",
```

```
            For[tk = kk , tk < (kk + kr) , tk++,
             For[tm = ii, tm < (ii + mr) , tm++,
              A⟦1 + tm, 1 + tk⟧ = APack⟦1 + Ai⟧; Ai++]],
            If[m["element-in-tile storage order"] === "row",
             For[tm = ii, tm < (ii + mr) , tm++, (* element iteration *)
              For[tk = kk , tk < (kk + kr) , tk++,
               A⟦1 + tm, 1 + tk⟧ = APack⟦1 + Ai⟧; Ai++]],
             Throw[752]]]]],
         If[m["tile-in-block storage order"] === "column",
          For[kk = k, kk < k + kc, kk += kr,
           For[ii = i, ii < i + mc, ii += mr,
            If[m["element-in-tile storage order"] === "column",
             For[tk = kk , tk < (kk + kr) , tk++, (* element iteration *)
              For[tm = ii, tm < (ii + mr) , tm++,
               A⟦1 + tm, 1 + tk⟧ = APack⟦1 + Ai⟧; Ai++]],
             If[m["element-in-tile storage order"] === "row",
              For[tm = ii, tm < (ii + mr) , tm++, (* element iteration *)
               For[tk = kk , tk < (kk + kr) , tk++,
                A⟦1 + tm, 1 + tk⟧ = APack⟦1 + Ai⟧; Ai++]],
              Throw[752]]]]]],
          Throw[753]]];
        dest["content"] = A;
        dest]]]]);
(* unit test *)
With[{mr = 2, kr = 4, ignored = 0,
   mcByMr = 3, kcByKr = 3,
   MByMc = 5, KByKc = 3},
  With[{mc = mr mcByMr, kc = kr kcByKr},
   With[{M = mc MByMc, K = kc KByKc},
    Module[{house =
        makeBlockTileMatrix[mr, kr, mcByMr, kcByKr, MByMc, KByKc,
         "column"(*element-in-tile storage order*),
         "row"(*tile-in-block storage order*),
         ConstantArray[0, {M, K}], "house", Null]},
      Print[house["uuid"]];
      (*Unevaluated is a pattern for mutability in Mathematica*)
      left$ = Module[{result = unpackBlock[
            Unevaluated[house], btp$, 1, 1, "unpacked", house["uuid"]]},
        Print[result["uuid"]]; result]]];
   Print[left$["content"] // griddit[#, mr, kr, mcByMr, kcByKr] &]]];
```

712cd444-8bab-4f0f-a415-ccaa0d7a55b8

712cd444-8bab-4f0f-a415-ccaa0d7a55b8

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	2	4	6	8	10	12	14	a	c	e	g	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	3	5	7	9	11	13	15	b	d	f	h	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	i	k	m	o	q	s	u	w	α	γ	ϵ	η	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	j	l	n	p	r	t	v	x	β	δ	ζ	θ	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	ι	λ	ν	ο	ρ	τ	ϕ	ψ	Δ	Λ	Π	Φ	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	κ	μ	ξ	π	σ	υ	ξ	ω	Θ	Ξ	Σ	Ω	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Create a numerical example for a left operand because the symbolic result of a multiplication, while instructive, is too lengthy for inspection.

In[65]:=
```
With[{bitcount = 2},
  With[{mr = 2, kr = 4, ignored = 0,
    mcByMr = 3, kcByKr = 3,
    MByMc = 5, KByKc = 3},
   With[{mc = mr mcByMr, kc = kr kcByKr},
    With[{M = mc MByMc, K = kc KByKc},
     left$ = makeBlockTileMatrix[mr, kr, mcByMr, kcByKr, MByMc, KByKc,
       "column"(*element-in-tile storage order*),
       "row"(*tile-in-block storage order*),
       RandomInteger[2^bitcount - 1, {M, K}], "left$", Null];
     Print[left$["uuid"]];
     Print[left$["content"] // griddit[#, mr, kr, mcByMr, kcByKr] &]]]]];
```

c2fa1931-b6bc-44f0-8332-2b2829f01aa0

2 0 0 0	1 2 3 3	2 3 0 3	2 0 0 0	3 2 0 0	1 3 0 0	0 1 1 1	3 3 1 3	0 0 1 0
1 1 1 2	0 1 1 1	1 2 2 1	1 2 3 1	3 0 1 3	0 1 0 3	0 2 3 3	2 1 1 0	3 2 1 0
3 0 2 3	3 1 3 2	3 0 2 3	3 2 0 2	1 1 3 2	2 2 0 1	0 1 0 1	2 0 0 2	0 1 2 0
1 0 3 2	1 3 0 3	0 1 2 0	3 1 2 0	0 1 3 1	2 3 1 0	0 1 1 1	2 0 1 3	0 0 2 0
1 0 1 1	1 0 0 1	3 3 2 1	0 2 0 3	1 2 2 0	0 1 1 1	1 0 0 2	3 2 2 3	3 0 0 0
0 3 1 3	1 1 3 2	1 0 0 1	3 1 3 0	3 1 0 1	0 3 2 0	0 2 3 1	2 1 1 1	3 1 1 1
3 3 0 1	2 0 2 1	3 3 0 2	0 3 3 0	3 3 0 1	1 1 3 1	0 1 2 2	2 3 0 0	3 1 3 3
2 2 3 1	3 2 1 3	0 0 1 0	1 1 2 2	2 2 3 3	3 2 2 1	1 0 1 3	2 0 0 3	2 3 1 0
3 3 0 3	1 1 3 3	3 2 1 2	0 3 0 0	3 3 2 2	3 3 1 2	0 3 0 0	0 0 0 3	2 2 2 1
1 0 3 1	2 3 1 0	3 2 1 0	2 1 1 1	0 0 0 0	0 3 2 2	3 2 3 0	0 0 2 0	2 0 2 0
2 2 3 1	0 0 3 3	0 1 2 3	3 3 1 3	1 2 3 0	1 2 1 2	3 3 1 3	0 3 2 1	3 2 0 3
0 0 2 0	0 2 3 0	1 2 1 3	0 2 3 3	2 0 0 1	1 3 2 1	0 3 0 2	3 1 2 3	0 0 3 0
2 0 3 0	0 3 1 0	2 2 2 2	1 0 3 0	3 0 1 0	3 0 2 3	3 3 3 2	0 2 2 3	0 1 1 1
0 2 3 1	2 2 3 1	1 3 2 0	1 1 3 3	2 0 3 0	2 0 2 1	1 3 3 0	0 0 0 1	2 3 2 2
1 2 2 1	2 3 2 1	0 2 0 2	1 2 0 3	1 1 0 0	1 2 2 2	0 2 3 1	2 0 1 0	1 2 2 0
3 1 1 2	0 2 2 0	0 2 2 1	0 2 2 1	1 3 3 2	2 1 3 0	3 3 3 1	2 3 3 1	0 2 0 3
1 3 2 0	3 1 3 2	1 3 0 0	1 2 1 1	1 3 2 1	0 1 3 3	2 0 3 0	1 0 2 1	0 3 3 3
2 2 0 3	1 3 1 0	3 3 1 0	0 3 0 3	3 1 2 2	2 3 0 0	1 2 2 0	1 2 2 0	3 2 0 2
1 2 0 0	1 0 2 1	3 2 2 0	0 2 1 0	1 1 3 2	3 3 0 1	0 0 0 0	3 1 3 3	3 3 1 2
1 0 1 3	1 2 3 0	0 3 1 3	1 0 1 1	1 0 0 0	3 0 2 2	2 1 1 1	2 1 1 1	3 0 2 1
3 0 3 3	3 3 0 2	1 3 2 2	0 1 1 1	3 0 1 2	2 1 3 1	1 0 0 0	2 1 0 0	3 2 0 2
0 3 1 1	1 1 3 0	1 1 1 2	2 2 1 3	2 1 3 1	1 3 2 2	3 3 2 3	2 0 3 2	2 0 1 1
2 1 2 3	1 0 2 2	0 3 1 1	2 3 3 1	1 3 1 3	3 2 3 1	1 3 2 1	2 1 0 2	2 1 2 3
3 3 0 3	2 1 0 3	0 3 1 1	3 1 3 0	2 1 1 2	1 2 3 0	2 2 3 2	1 2 0 0	2 2 3 0
0 2 0 1	1 2 0 1	0 0 0 2	2 3 1 2	3 3 0 1	1 3 1 0	2 3 2 0	0 3 3 1	3 0 2 2
2 1 0 0	1 1 2 2	2 1 0 1	1 3 1 0	3 3 3 2	2 1 1 1	2 2 1 1	3 1 1 1	2 2 3 2
3 2 3 1	2 2 0 0	0 2 1 1	3 1 1 1	1 0 0 0	2 1 2 0	2 2 3 0	1 2 1 1	2 0 3 3
3 3 2 3	2 1 1 1	2 0 2 1	3 2 1 0	2 1 3 1	0 1 1 0	3 3 1 3	2 1 0 0	2 0 1 0
2 2 2 2	2 2 2 3	0 0 3 2	1 3 3 0	0 0 0 0	2 1 3 1	2 3 1 0	2 3 0 2	3 1 0 0
3 0 2 2	2 0 3 3	0 2 3 2	2 0 3 0	2 2 1 0	3 3 1 1	1 1 1 1	0 3 2 3	2 1 3 2

Give an example of a right-hand multiplicand, again reproduced from Kuzma’s Figure 2.

In[66]:=

```
With[{bitcount = 2},
 With[{kr = 4, nr = 2, ignored = 0,
   kcByKr = 3, ncByNr = 5,
   KByKc = 3, NByNc = 3},
  With[{kc = kr kcByKr, nc = nr ncByNr},
   With[{K = kc KByKc, N = nc NByNc},
    right$ =
     makeBlockTileMatrix[kr, nr, kcByKr, ncByNr, KByKc, NByNc,
      "row"(*element-in-tile storage order*),
      "row"(*tile-in-block storage order*),
      RandomInteger[2^bitcount - 1, {K, N}], "right$", Null];
    Print[right$["uuid"]];
    Print[right$["content"] // griddit[#, kr, nr, kcByKr, ncByNr] &]]]]]
```

08104e70-2a1a-4df9-b2d7-7190a026d715

1	1	1	1	1	3	0	2	3	2	1	0	1	3	2	0	3	0	0	2	2	2	0	2	1	3	2	1	2	0
0	0	2	2	2	1	0	1	0	3	2	1	3	3	2	1	3	3	1	0	2	1	1	0	2	0	2	3	1	3
2	3	0	1	3	0	1	1	1	0	1	3	1	2	1	1	0	2	3	3	1	2	2	0	0	2	0	1	3	2
2	1	1	1	1	0	3	1	0	2	0	1	0	0	0	1	2	3	3	0	3	2	3	1	3	1	1	0	2	2
1	0	3	0	0	3	1	3	3	2	0	3	1	2	2	0	2	3	3	2	3	2	1	3	1	3	2	1	1	0
3	0	2	1	2	0	1	0	0	1	0	1	3	1	0	0	1	1	0	1	3	1	3	0	1	2	2	1	2	0
2	2	0	0	3	3	2	0	2	0	0	1	2	3	1	2	2	2	1	1	3	3	3	3	2	1	2	1	2	1
0	3	3	2	1	3	0	3	1	1	2	2	0	1	1	3	0	0	0	2	2	1	0	0	2	1	2	2	1	2
2	3	0	3	1	1	2	1	1	3	0	1	3	1	0	2	2	2	1	1	1	3	3	0	0	2	2	0	3	3
2	1	1	1	2	2	1	2	1	2	3	1	3	2	0	0	1	0	2	2	0	0	1	0	3	2	3	1	2	3
3	2	3	0	2	0	2	2	1	1	0	2	3	1	0	2	3	3	3	3	1	2	2	3	2	3	1	2	2	3
1	3	3	2	3	3	0	2	3	2	0	2	1	0	1	2	3	2	0	2	2	2	2	3	3	0	0	2	2	0
1	1	0	3	1	3	1	3	1	0	0	1	2	3	3	0	3	0	0	1	3	3	2	2	2	0	1	1	3	3
3	0	2	3	3	3	1	1	0	0	3	3	3	1	2	0	2	3	3	3	2	1	2	1	0	1	1	0	1	1
1	3	3	2	3	2	2	2	1	2	0	3	0	2	1	3	1	0	1	3	3	3	3	3	2	2	3	3	3	3
3	2	0	2	1	3	3	0	2	3	1	1	1	2	1	1	3	1	1	1	2	0	2	1	3	3	2	1	1	0
3	1	1	2	2	1	3	3	1	0	1	0	3	3	0	1	0	2	0	3	1	0	1	1	2	1	2	0	3	0
1	0	0	3	2	3	3	2	0	0	3	1	0	1	2	1	1	1	1	1	1	2	2	0	1	1	3	1	1	0
0	3	2	0	0	2	1	1	3	1	3	3	0	2	3	1	3	1	0	1	2	3	3	1	0	3	0	1	2	1
0	1	2	0	2	0	0	3	2	3	3	3	0	1	2	0	3	1	3	0	3	3	2	3	1	3	3	3	1	2
0	1	0	3	1	1	0	0	0	3	1	3	2	0	3	3	0	3	1	3	2	1	2	2	0	2	3	0	2	1
3	2	1	2	0	3	1	2	0	1	3	0	1	1	2	2	3	2	3	0	0	3	3	3	1	1	1	1	3	0
0	2	3	0	0	0	0	1	0	3	1	1	3	1	1	1	1	3	0	0	1	2	2	0	3	0	2	0	1	3
0	2	3	1	0	0	2	2	2	2	0	2	0	3	1	1	1	1	0	2	1	1	1	1	3	1	3	2	0	2
3	2	3	2	0	0	0	0	3	0	1	3	2	2	3	0	3	3	2	2	1	3	0	0	0	0	2	0	2	0
2	3	1	3	0	0	2	0	1	0	3	3	3	0	3	3	3	1	1	2	1	3	0	2	1	3	0	1	1	1
0	2	1	0	2	1	2	3	2	3	1	3	3	3	2	3	1	1	1	1	2	1	2	2	3	0	2	1	1	1
3	3	0	0	0	0	2	2	1	3	3	1	0	3	1	2	2	0	0	0	0	0	1	0	2	3	3	2	0	2
3	2	2	0	0	1	2	3	2	0	1	0	1	1	2	1	0	3	2	3	0	3	1	3	0	2	2	3	2	2
3	1	3	0	0	0	2	2	3	0	0	0	1	1	2	1	1	2	0	2	0	1	0	1	1	3	1	1	3	3
1	0	1	0	2	2	3	1	3	3	3	0	2	2	2	2	3	0	1	2	1	3	3	1	1	2	0	3	3	2
2	0	2	3	0	1	3	2	0	0	0	2	2	3	3	2	2	2	1	0	1	1	2	3	0	2	3	0	0	0
1	3	2	0	2	1	1	2	1	2	3	0	3	1	0	0	0	3	1	0	1	0	0	0	2	3	2	0	2	1
1	2	3	3	3	0	1	3	2	3	2	3	1	2	3	1	3	1	1	0	1	2	2	2	3	0	2	1	3	1
3	3	0	2	1	1	2	2	2	2	0	0	3	2	2	1	2	2	2	1	3	0	0	0	3	0	3	2	0	3
1	0	2	0	0	2	1	3	3	3	2	2	2	2	2	2	1	2	3	3	3	1	3	2	2	3	3	1	2	0

In[67]:=

```
Short[right$, 2]
```

Out[67]//Short=

```
⟨| mr → 4, kr → 2, mc/mr → 3, mc → 12, ≪9≫, name → right$,
 uuid → 08104e70-2a1a-4df9-b2d7-7190a026d715, type → blockTileMatrix |⟩
```

Ground truth for the matrix product, via Mathematica's built-in.

In[68]:=

```
With[{mr = 2, mcByMr = 3},
 With[{nr = 2, ncByNr = 5},
  Print[
   left$["content"].right$["content"] // griddit[#, mr, nr, mcByMr, ncByNr] &]]]
```

83	67	60	66	53	79	68	84	61	46	49	45	74	75	65	62	69	63	40	68	59	74	68	69	63	6
81	94	80	58	76	55	79	96	69	83	73	78	79	89	64	65	84	72	61	73	76	78	79	71	91	8
87	91	70	79	67	90	74	89	77	73	54	85	74	85	84	65	107	90	71	77	99	103	97	89	73	9
71	73	63	63	51	63	61	76	46	53	55	73	64	70	74	61	74	65	60	65	76	83	83	65	55	7
83	71	65	53	47	63	75	72	60	61	64	52	70	70	56	48	72	74	54	63	46	65	68	49	57	8
76	84	74	67	72	72	76	93	56	72	64	67	88	87	72	72	81	86	61	62	87	90	88	74	88	6
91	93	92	76	81	89	81	112	82	101	81	76	107	101	81	73	89	103	71	90	93	86	87	72	102	9
82	95	92	78	73	77	74	104	72	92	82	101	77	104	97	71	96	95	71	75	96	95	96	81	81	1
87	91	87	103	81	92	81	97	66	89	88	91	103	93	92	80	109	106	75	79	100	99	99	82	90	9
76	77	59	56	54	53	60	61	58	66	49	66	89	74	59	49	78	71	62	61	68	80	74	47	66	6
105	116	103	90	86	101	89	104	102	90	101	107	101	115	110	88	126	102	74	101	100	111	104	84	104	1
102	90	67	71	67	69	83	71	62	66	58	67	87	79	70	74	87	79	62	78	71	83	88	79	76	8
88	94	85	76	69	51	80	81	78	80	53	95	101	99	83	82	88	80	51	95	79	90	89	72	80	8
82	99	83	74	82	70	79	83	75	90	68	104	106	100	82	76	92	91	71	85	99	86	97	72	97	9
77	76	66	64	68	69	66	74	61	77	61	70	86	81	69	57	81	77	57	65	77	70	77	60	87	6
93	86	98	69	76	75	85	92	93	91	91	100	99	96	105	80	112	96	76	95	93	114	107	86	86	1
73	80	90	71	76	86	75	101	86	94	78	94	96	113	94	65	97	89	75	80	97	93	96	66	97	7
97	75	77	74	75	76	82	84	73	93	89	76	105	84	77	61	101	96	78	77	86	87	99	69	81	1
74	74	80	67	65	74	71	90	70	83	81	73	86	83	85	67	89	90	72	72	71	91	93	80	61	9
75	77	68	50	59	55	63	66	65	71	43	62	82	70	58	55	68	84	54	70	75	68	71	61	82	7
81	82	95	56	71	64	59	94	73	86	63	80	85	76	61	51	75	96	73	86	85	81	84	68	84	9
100	101	85	78	70	85	91	87	84	90	90	91	105	109	99	80	120	102	72	80	91	107	105	82	91	9
93	101	95	90	83	92	86	112	78	96	94	108	102	100	105	85	104	107	93	97	109	109	106	94	102	1
84	98	97	78	73	77	69	111	76	96	79	86	97	99	91	69	104	89	68	75	98	94	79	73	107	8
87	67	76	78	65	77	77	82	66	68	78	67	96	76	84	63	90	87	62	73	81	78	78	62	77	7
86	89	83	81	69	82	75	98	82	76	85	84	93	92	96	67	92	92	65	86	89	95	84	72	74	8
79	72	71	62	59	65	60	83	73	76	55	73	101	87	82	56	82	86	67	81	86	75	71	63	78	7
89	92	77	68	60	66	68	84	72	67	70	80	86	89	82	57	103	92	65	74	85	99	76	64	70	8
84	90	106	71	71	66	63	82	65	66	56	91	93	76	81	72	86	104	66	87	86	95	77	80	77	8
95	99	92	80	77	94	88	109	87	83	67	86	93	103	94	89	100	95	75	97	98	100	96	94	97	1

loadTile (UNDONE)

Just as packBlock and unpackBlock use 0-based block indices in the matrix, loadTile and saveTile use 0-based tile indices in a block. This may differ from Kuzma.

In[69]:=

```
ClearAll[loadTile];
```

saveTile (UNDONE)

In[70]:=

```
ClearAll[saveTile];
```

Algorithm 1, Robustly

```
packBlock[m_?blockTileMatrixQ,
 (*Pick a single block via 0-
  based block indices from the middle of a blockTileMatrix*)
 fromRowBlock_Integer, fromColumnBlock_Integer,
 optionalName_String, uuid_ /; (uuid === Null || uuidQ[uuid])]
```

This next test is UNDONE.

In[ ]:=
```
With[{A = left$, B = right$},
 With[{mr = 4, kr = 2, nr = 2},
  With[{mcByMr = 3, kcByKr = 3, ncByNr = 5},
   With[{MByMc = 5, KByKc = 3, NByNc = 3},
    With[{mc = mr mcByMr, kc = kr kcByKr, nc = nr ncByNr},
     With[{M = mc MByMc, K = kc KByKc, N = nc NByNc},
      Module[{j, k, i, jj, ii, kk, APack, BPack, C = ConstantArray[0, {M, N}]},
       (*for each block*)
       For[j = 0, j < N, j += nc,
        For[k = 0, k < K, k += kc,
         BPack = Echo@packBlock[B, k / kc, j / nc, "BPack", Null];
         For[jj = 0, jj < nc, jj += nr,
          For[ii = 0, ii < mc, ii += mr,
           For[kk = 0, kk < kc, kk += kr,
            Null]]]
        ]]]]]]]]
```

In[71]:=
```
ClearAll[blockTileOp];
blockTileOp[mr_Integer, kr_Integer, nr_Integer,
   mcByMr_Integer, kcByKr_Integer, ncByNr_Integer,
   MByMc_Integer, KByKc_Integer, NByNc_Integer,
   unaryOrBinaryOp_,
   leftOperandMatrix_Symbol, leftIndices_List,
   rightOperandMatrix_List, rightIndices_List] :=
  With[{mc = mr mcByMr, kc = kr kcByKr, nc = nr ncByNr},
   With[{M = mc MByMc, K = kc KByKc, N = nc NByNc},
    unaryOrBinaryOp[Unevaluated[leftOperandMatrix], leftIndices,
     rightOperandMatrix, rightIndices,
     mr, kr, nr, mc, kc, nc, M, K, N]]];
```

copyBlockCanon

In[73]:=
```
ClearAll[copyBlockCanon];
copyBlockCanon[
   left_Symbol, leftIndices_List,
   right_List, rightIndices_List,
   mr_Integer, kr_Integer, nr_Integer,
   mc_Integer, kc_Integer, nc_Integer,
   M_Integer, K_Integer, N_Integer] :=
  Module[{ii, kk, destI, destJ, srcI, srcJ},
   (* unpack indices *)
   {destI, destJ} = leftIndices;
   {srcI, srcJ} = rightIndices;
   For[ii = 0, ii < mc, ii++,
    For[kk = 0, kk < kc, kk++,
     left⟦1 + destI + ii, 1 + destJ + kk⟧ = right⟦1 + srcI + ii, 1 + srcJ + kk⟧]];
   left];
```

Unit Test

For this test, we didn’t purchase fewer *With* expressions, but we got expressions of greater reliability and clarity.

In[75]:=
```
With[{mr = 2, kr = 4, ignored = 0,
   mcByMr = 3, kcByKr = 3,
   MByMc = 5, KByKc = 3},
  With[{mc = mr mcByMr, kc = kr kcByKr},
   With[{M = mc MByMc, K = kc KByKc},
    Module[{housingMatrix = ConstantArray[0, {M, K}]},
     blockTileOp[mr, kr, ignored,
      mcByMr, kcByKr, ignored,
      MByMc, KByKc, ignored,
      copyBlockCanon,
      Unevaluated[housingMatrix], {1 mc, 1 kc},
      utA$, {0, 0}]]]]] // MatrixForm
```

Out[75]//MatrixForm=

$$
\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 4 & 6 & 8 & 10 & 12 & 14 & a & c & e & g & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 3 & 5 & 7 & 9 & 11 & 13 & 15 & b & d & f & h & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & i & k & m & o & q & s & u & w & \alpha & \gamma & \epsilon & \eta & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & j & l & n & p & r & t & v & x & \beta & \delta & \zeta & \theta & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \iota & \lambda & \nu & o & \rho & \tau & \phi & \psi & \Delta & \Lambda & \Pi & \Phi & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \kappa & \mu & \xi & \pi & \sigma & \upsilon & \xi & \omega & \Theta & \Xi & \Sigma & \Omega & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
$$

In[76]:=
```
With[{bitCount = 16},
 With[{mr = 1, kr = 2048, nr = 1}, (* -- mr must equal nr -- tiles *)
  With[{mc = 3 mr, kc = 3 kr, nc = 5 nr}, (* mc=3, kc=6144, nc=5 -- blocks *)
   With[{M = 5 mc, K = 3 kc, N = 3 nc},
    (* M=15, K=18432, N=15, -- original dims *)
    With[{A = RandomInteger[{0, 2^bitCount - 1}, {M, K}],
      B = RandomInteger[{0, 2^bitCount - 1}, {K, N}]},
     Module[{
       ABlocks = blockIt[A, mc, M, kc, K],
       BBlocks = blockIt[B, kc, K, nc, N],
       CTiled, CBlocked, CBlockedCheck, C, CCheck, bm, bk, bn, tm, tk, tn},
      CTiled = blockTileMul[ABlocks, BBlocks, M, K, N, mc, kc, nc, mr, kr, nr];
      (* Check intermediate forms. *)
      CBlocked =
       Table[untileBlock[CTiled[[m, n]], mr, mc, nr, nc], {m, 1, M/mc}, {n, 1, N/nc}];
      CBlockedCheck = blockMul[ABlocks, BBlocks, M, K, N, mc, kc, nc, mr, kr, nr];
      Assert[CBlockedCheck === CBlocked];
      C = unblock[CBlocked, mc, M, nc, N];
      CCheck = A.B;
      Assert[CCheck === C];
      Column[{(* displays *)
        CTiled // MatrixForm,
        C // MatrixForm,
        <|"dim[A]" → Dimensions[A],
           "dim[B]" → Dimensions[B],
           "dim[C_tiled]" → Dimensions[CTiled],
           "dim[A_blocks]" → Dimensions[ABlocks],
           "dim[B_blocks]" → Dimensions[BBlocks],
           "dim[C]" → Dimensions[C],
           "bits" → bitCount,
           "mr" → mr, "kr" → kr, "nr" → nr,
           "mc" → mc, "kc" → kc, "nc" → nc,
           "M" → M, "K" → K, "N" → N|> // Print;}]]]]]]
```

```
<|dim[A] → {15, 18432}, dim[B] → {18432, 15}, dim[C_tiled] → {5, 3, 3, 5, 1, 1},
 dim[A_blocks] → {5, 3, 3, 6144}, dim[B_blocks] → {3, 3, 6144, 5}, dim[C] → {15, 15},
 bits → 16, mr → 1, kr → 2048, nr → 1, mc → 3, kc → 6144, nc → 5, M → 15, K → 18432, N → 15|>
```

Out[76]=

(19 740 733 816 606)	(19 798 662 265 966)	(19 771 155 932 121)	(19 750 697 475 859)	(
(19 654 933 054 807)	(19 732 288 959 513)	(19 759 787 698 380)	(19 780 689 976 289)	(
(19 622 825 369 964)	(19 651 495 933 610)	(19 762 402 306 994)	(19 640 466 034 417)	(
(19 671 022 518 357)	(19 769 463 228 663)	(19 802 912 494 522)	(19 726 554 773 721)	(
(19 764 155 915 905)	(19 728 509 432 458)	(19 852 414 148 243)	(19 797 787 001 695)	(
(19 831 597 709 968)	(19 922 283 748 601)	(19 959 702 817 686)	(19 797 327 018 049)	(
(19 853 410 851 232)	(19 830 216 731 774)	(19 897 863 615 941)	(19 743 484 470 147)	(
(19 667 376 550 637)	(19 661 031 743 980)	(19 843 196 120 081)	(19 747 844 575 264)	(
(19 772 518 004 474)	(19 824 280 568 839)	(20 009 371 412 787)	(19 854 832 130 478)	(
(19 633 974 245 733)	(19 784 574 800 389)	(19 784 420 268 288)	(19 717 696 973 265)	(
(19 689 417 653 191)	(19 767 844 784 855)	(19 861 997 973 462)	(19 776 767 969 297)	(
(19 684 301 406 680)	(19 778 579 034 376)	(19 711 644 251 290)	(19 705 059 065 189)	(
(19 806 390 841 202)	(19 848 561 952 141)	(19 869 991 033 034)	(19 791 177 377 227)	(
(19 717 908 175 763)	(19 773 768 793 630)	(19 889 098 706 618)	(19 739 484 396 405)	(
(19 762 003 560 659)	(19 921 310 094 886)	(19 897 483 318 549)	(19 776 385 082 489)	(

19 740 733 816 606	19 798 662 265 966	19 771 155 932 121	19 750 697 475 859	19 829 122 217 2
19 654 933 054 807	19 732 288 959 513	19 759 787 698 380	19 780 689 976 289	19 812 200 158 6
19 622 825 369 964	19 651 495 933 610	19 762 402 306 994	19 640 466 034 417	19 803 587 432 4
19 671 022 518 357	19 769 463 228 663	19 802 912 494 522	19 726 554 773 721	19 761 741 167 9
19 764 155 915 905	19 728 509 432 458	19 852 414 148 243	19 797 787 001 695	19 769 902 449 5
19 831 597 709 968	19 922 283 748 601	19 959 702 817 686	19 797 327 018 049	19 972 754 361 8
19 853 410 851 232	19 830 216 731 774	19 897 863 615 941	19 743 484 470 147	19 894 482 340 1
19 667 376 550 637	19 661 031 743 980	19 843 196 120 081	19 747 844 575 264	19 893 073 753 5
19 772 518 004 474	19 824 280 568 839	20 009 371 412 787	19 854 832 130 478	19 952 607 755 4
19 633 974 245 733	19 784 574 800 389	19 784 420 268 288	19 717 696 973 265	19 908 698 763 3
19 689 417 653 191	19 767 844 784 855	19 861 997 973 462	19 776 767 969 297	19 943 648 046 6
19 684 301 406 680	19 778 579 034 376	19 711 644 251 290	19 705 059 065 189	19 802 527 469 8
19 806 390 841 202	19 848 561 952 141	19 869 991 033 034	19 791 177 377 227	19 934 037 551 6
19 717 908 175 763	19 773 768 793 630	19 889 098 706 618	19 739 484 396 405	19 886 222 241 1
19 762 003 560 659	19 921 310 094 886	19 897 483 318 549	19 776 385 082 489	19 899 772 962 9