Compiling Matmul to Blocks and Tiles, Version 2

Technical Report, Preliminary Draft, GSI Technology

Brian Beckman
Technology Fellow
November, 2023

Abstract

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 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 to, multiplying in, and storing from chunks of A and B in the APU's L1 cache and main memory. The solution to the layout problem includes finding optimal sizes of chunks and optimal sequences of operations for moving and multiplying data. *Optimal* means *minimum running time*. Compile time is not considered. Running time includes the time for I/O between L1 and main memory.

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*. presents an approach wherein the compiler breaks up input domain matrices into ***blocks*** and ***tiles***. Blocks are optimized to fit L1, tiles are optimized to fit main memory, where multiplication occurs. We investigate and mechanize 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=

1	$5.\times10^{-6}$ s	0.000017 s	0.000012 s
26	0.000018 s	0.001684 s	0.000081 s
51	0.000013 s	0.007965 s	0.000225 s
76	0.000241 s	0.026589 s	0.000637 s
101	0.00006 s	0.063752 s	0.001698 s
126	0.000071 s	0.113922 s	0.003041 s
151	0.000087 s	0.173122 s	0.003815 s
176	0.000348 s	0.305297 s	0.004737 s

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]=

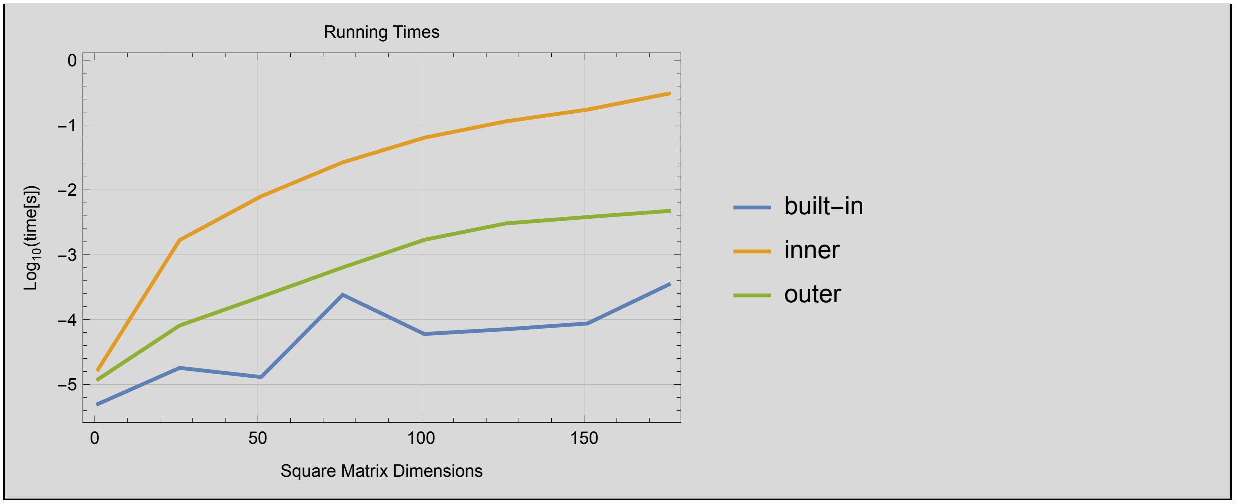


Table 1 — VSR and ACC

Kuzma presents a worked-out example for his IBM POWER10 ***MMA*** chip, which has ***VSR***s of 128 bits and ***ACC***s of 512 bits. The bits in these registers can handle seven different types of elements.

In[∘]:=

TABLE 1 MMA instruction summary.

Input type	Computation size $m \times k \cdot k \times n$	Result shape and type
4-bit integer (*i*4)	$4 \times 8 \cdot 8 \times 4$	4×4 i32
8-bit integer (*i*8)	$4 \times 4 \cdot 4 \times 4$	
16-bit integer (*i*16)	$4 \times 2 \cdot 2 \times 4$	
brain-float (*bf*16)	$4 \times 2 \cdot 2 \times 4$	4×4 f32
IEEE half-precision (*f*16)	$4 \times 2 \cdot 2 \times 4$	
IEEE single-precision (*f*32)	$4 \times 1 \cdot 1 \times 4$	
IEEE double-precision (*f*64)	$4 \times 1 \cdot 1 \times 2$	4×2 *f*64

In[14]:=

```
ClearAll[mmaI, mmaI4, mmaI8, mmaI16, mmaBf16, mmaF16, mmaF32, mmaF64];
```

The integer MMA instructions for the POWER10 consume four 128-bit VSRs — an ACC, for a total of 512 bits. Up to 32 VSRs can be used, 4 at a time, in this way.

In[15]:=
```
mmaI[bitCount_?(MemberQ[{4, 8, 16}, #] &)] :=
 With[{m = 4, k = 32 / bitCount, n = 4},
  With[{A = RandomInteger[{0, 2^bitCount - 1}, {m, k}],
    B = RandomInteger[{0, 2^bitCount - 1}, {k, n}]},
   accumulatedOuterProduct[m, k, n, A, B]]]
```

In[16]:=
```
mmaI[8]
```

Out[16]=
```
<|m → 4, k → 4, n → 4,
 result → {{66708, 81090, 53226, 66894}, {93391, 87949, 77167, 66791}, {58967,
    73132, 52862, 59159}, {77830, 82236, 51364, 60610}}, time → 0.000022 s|>
```

accumulatedOuterProduct also works as a rank-1 outer product.

In[17]:=
```
accumulatedOuterProduct[5, 1, 4, (1; 2; 3; 4; 5), (7 11 13 19)]["result"] // MatrixForm
```

Out[17]//MatrixForm=

$$\begin{pmatrix} 7 & 11 & 13 & 19 \\ 14 & 22 & 26 & 38 \\ 21 & 33 & 39 & 57 \\ 28 & 44 & 52 & 76 \\ 35 & 55 & 65 & 95 \end{pmatrix}$$

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. α and β are free scalar parameters required by GEMM.

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};
```

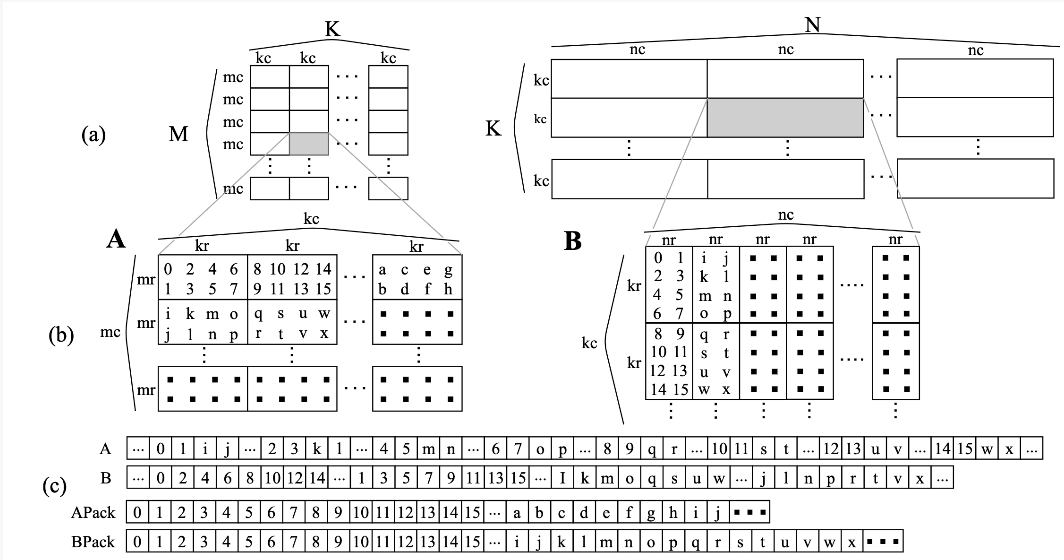
Algorithm 1. Algorithm overview for GEMM

```
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.

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**.

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 that are necessary 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 tiles inside, then triples of integers for *inner* and *outer dimensions*. The three outer dimensions, **mc**, **kc**, and **nc**, correspond to the dimensions of block multiplicands, $\mathrm{mc} \times \mathrm{kc}$ and $\mathrm{kc} \times \mathrm{nc}$. The three inner dimensions, **mr**, **kr**, and **nr**, correspond to dimensions of tile multiplicands, namely $\mathrm{mr} \times \mathrm{kr}$ and $\mathrm{kr} \times \mathrm{nr}$. Each outer dimension must be evenly divisible by the corresponding inner dimension, meaning that tiles must fit 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 $\frac{\mathrm{mc}}{\mathrm{mr}}$, $\frac{\mathrm{kc}}{\mathrm{kr}}$, and $\frac{\mathrm{nc}}{\mathrm{nr}}$ 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 a tile product. The API has separate parameters for them for the sake of symmetry in the API, making it easier to remember.

Let’s 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, from general matrices, to produce matrices full of snug blocks, in-turn full of 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 fully blocked and tiled matrix. Below is an example. Notice we build the dimensions bottom-up to ensure integer divisibility and 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]=

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

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

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;
        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;
        (*griddit[A,mc,M,kc,K],*)
        (*griddit[B,kc,K,nc,N]*)}]]]]]]]
```

```
<|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]=

```
1867 1960  2027 2007  2386 1714  1480 2287  2018 2486    1815 152
1660 1517  2015 1777  2269 1659  1402 2207  2215 2249    1733 131
1860 1801  2081 2092  2362 1846  1581 2523  1911 2645    1955 179
1970 1706  1898 1948  2315 1687  1737 2255  2163 2665    1820 164
1806 1861  2127 2208  2577 1866  1837 2403  2453 2629    2031 206
1780 1674  2003 1969  1753 1618  1520 1986  1948 2201    1433 155
2156 1907  1782 1760  2362 1784  1690 2129  2313 2385    1775 169
1815 1656  1922 1744  2020 1712  1554 1904  2096 1936    1716 145
2413 2230  2249 2564  2688 2289  2120 2880  2643 2895    2187 204
1924 1733  1901 1968  2119 1785  1507 2288  1981 2481    1795 151
2460 2041  2227 2278  2797 2237  2016 2608  2728 2878    2109 195
1729 1756  1666 1734  1851 1612  1593 2055  1764 2153    1715 147
1875 1816  2070 1911  2037 1812  1619 2469  2344 2174    1747 186
1953 1897  2207 2330  2228 1821  1718 2536  2485 2668    2026 175
1772 1679  1942 1730  2156 1709  1491 2160  2038 2038    1529 164
1755 1706  1997 2085  2219 1833  1834 2175  2178 2664    1723 194
2107 1931  2138 2194  2470 1912  1712 2592  2289 2787    2017 189
1826 1698  1905 1718  2055 1710  1681 2023  2104 1987    1557 155
2090 2021  2299 2130  2526 2019  2027 2562  2569 2485    2102 182
1817 1629  1925 1765  1889 1791  1605 2180  2161 2000    1506 134
1885 1808  1719 1853  2098 1627  1370 1985  1936 2116    1792 138
1877 2001  1811 1964  2537 1954  1834 2488  2177 2314    2078 175
1660 1834  1767 2173  2098 1791  1781 2364  2212 2316    1982 167
1547 1413  1621 1785  1649 1541  1448 1719  1940 2022    1436 132
2207 2201  2119 1987  2448 1919  1987 2423  2515 2431    1812 167
1928 1798  1905 1900  2116 1510  1366 1828  2269 2534    1487 147
2031 1689  2044 1926  2435 1730  1744 2459  2339 2524    1817 172
1827 1511  1890 1774  2075 1908  1725 2036  2160 2145    1657 166
1540 1548  1762 1871  2069 1762  1640 2012  1893 2274    1619 138
1692 1734  2136 2030  2191 1759  1716 2150  2178 2153    1855 167
```

```
1867 1960 2027 2007 2386 1714 1480 2287 2018 2486 1815 1522 1974 1828 2027 2
1660 1517 2015 1777 2269 1659 1402 2207 2215 2249 1733 1316 2028 1994 1865 2
1860 1801 2081 2092 2362 1846 1581 2523 1911 2645 1955 1799 2116 2002 2190 2
1970 1706 1898 1948 2315 1687 1737 2255 2163 2665 1820 1648 1972 1983 2111 2
1806 1861 2127 2208 2577 1866 1837 2403 2453 2629 2031 2065 2080 2087 2355 2
1780 1674 2003 1969 1753 1618 1520 1986 1948 2201 1433 1555 1555 1592 1766 2
2156 1907 1782 1760 2362 1784 1690 2129 2313 2385 1775 1698 2087 1790 2257 2
1815 1656 1922 1744 2020 1712 1554 1904 2096 1936 1716 1457 1824 1816 1966 1
2413 2230 2249 2564 2688 2289 2120 2880 2643 2895 2187 2042 2558 2246 2472 3
1924 1733 1901 1968 2119 1785 1507 2288 1981 2481 1795 1519 2142 1984 2032 2
2460 2041 2227 2278 2797 2237 2016 2608 2728 2878 2109 1955 2407 2129 2516 2
1729 1756 1666 1734 1851 1612 1593 2055 1764 2153 1715 1470 1845 1521 1873 1
1875 1816 2070 1911 2037 1812 1619 2469 2344 2174 1747 1866 1993 2101 2236 2
1953 1897 2207 2330 2228 1821 1718 2536 2485 2668 2026 1750 1925 2013 2435 2
1772 1679 1942 1730 2156 1709 1491 2160 2038 2038 1529 1642 1807 1648 1993 2
1755 1706 1997 2085 2219 1833 1834 2175 2178 2664 1723 1948 1897 2070 2152 2
2107 1931 2138 2194 2470 1912 1712 2592 2289 2787 2017 1890 2214 1865 2302 2
1826 1698 1905 1718 2055 1710 1681 2023 2104 1987 1557 1556 1799 1734 2170 1
```

```
2090 2021 2299 2130 2526 2019 2027 2562 2569 2485 2102 1821 2209 2085 2449 2
1817 1629 1925 1765 1889 1791 1605 2180 2161 2000 1506 1345 1798 1785 1972 2
1885 1808 1719 1853 2098 1627 1370 1985 1936 2116 1792 1383 2009 1695 1772 1
1877 2001 1811 1964 2537 1954 1834 2488 2177 2314 2078 1758 2311 1868 2082 2
1660 1834 1767 2173 2098 1791 1781 2364 2212 2316 1982 1674 1961 1892 2022 2
1547 1413 1621 1785 1649 1541 1448 1719 1940 2022 1436 1320 1640 1594 1840 1
2207 2201 2119 1987 2448 1919 1987 2423 2515 2431 1812 1670 2087 1910 1975 2
1928 1798 1905 1900 2116 1510 1366 1828 2269 2534 1487 1478 1971 1725 1913 2
2031 1689 2044 1926 2435 1730 1744 2459 2339 2524 1817 1720 1893 1722 2187 2
1827 1511 1890 1774 2075 1908 1725 2036 2160 2145 1657 1662 1866 1808 1852 2
1540 1548 1762 1871 2069 1762 1640 2012 1893 2274 1619 1384 2054 1836 2007 2
1692 1734 2136 2030 2191 1759 1716 2150 2178 2153 1855 1672 1864 2012 2168 2
```

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 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 in L1. *B* is partitioned into blocks of dimension 6144×5, requiring 15 VRs 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 harvesting by the host computer. The prototype **blockTileMul** in this paper does no data movement, but it does perform blocked and tiled multiplication.

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 *)
  A // 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;
  (*griddit[A,mc,M,kc,K],*)
  (*griddit[B,kc,K,nc,N]*)}]]]]]]]
```

```
<|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]=

(19 778 916 013 054)	(19 853 380 813 110)	(19 699 723 520 757)	(20 089 802 879 770)	(
(19 645 620 781 660)	(19 739 415 086 534)	(19 603 776 803 489)	(19 859 247 074 723)	(
(19 792 629 049 349)	(19 710 578 316 741)	(19 540 408 055 908)	(19 882 192 333 930)	(
(19 670 882 432 586)	(19 764 565 480 243)	(19 629 901 547 102)	(19 968 169 994 600)	(
(19 716 860 364 385)	(19 719 207 418 628)	(19 703 572 976 377)	(20 027 430 017 537)	(
(19 678 719 735 447)	(19 753 353 564 545)	(19 542 816 455 189)	(19 965 521 275 835)	(
(19 860 648 515 157)	(19 929 538 947 846)	(19 768 746 183 613)	(20 088 615 550 375)	(
(19 757 946 930 592)	(19 787 614 392 197)	(19 695 345 622 499)	(19 947 071 924 457)	(
(19 821 994 187 268)	(19 838 774 614 781)	(19 669 361 760 040)	(19 964 265 697 244)	(
(19 710 232 231 497)	(19 710 694 169 764)	(19 593 035 388 587)	(19 959 828 085 613)	(
(19 625 411 885 830)	(19 799 104 854 588)	(19 699 729 944 087)	(19 923 431 955 371)	(
(19 932 426 131 330)	(19 874 570 579 461)	(19 808 537 670 866)	(20 193 007 285 170)	(
(19 601 484 108 285)	(19 811 095 874 249)	(19 531 140 468 808)	(19 927 097 803 468)	(
(19 739 854 781 601)	(19 756 184 698 826)	(19 615 485 537 683)	(19 933 379 959 308)	(
(19 818 265 211 124)	(19 831 232 101 619)	(19 663 798 623 225)	(20 008 426 425 976)	(

19 778 916 013 054	19 853 380 813 110	19 699 723 520 757	20 089 802 879 770	20 106 557 852 2
19 645 620 781 660	19 739 415 086 534	19 603 776 803 489	19 859 247 074 723	19 913 253 262 7
19 792 629 049 349	19 710 578 316 741	19 540 408 055 908	19 882 192 333 930	19 855 100 747 1
19 670 882 432 586	19 764 565 480 243	19 629 901 547 102	19 968 169 994 600	19 944 256 789 9
19 716 860 364 385	19 719 207 418 628	19 703 572 976 377	20 027 430 017 537	19 888 711 657 7
19 678 719 735 447	19 753 353 564 545	19 542 816 455 189	19 965 521 275 835	19 843 412 058 3
19 860 648 515 157	19 929 538 947 846	19 768 746 183 613	20 088 615 550 375	20 112 245 738 1
19 757 946 930 592	19 787 614 392 197	19 695 345 622 499	19 947 071 924 457	20 013 296 678 3
19 821 994 187 268	19 838 774 614 781	19 669 361 760 040	19 964 265 697 244	20 055 766 024 2
19 710 232 231 497	19 710 694 169 764	19 593 035 388 587	19 959 828 085 613	19 865 078 813 4
19 625 411 885 830	19 799 104 854 588	19 699 729 944 087	19 923 431 955 371	19 890 746 084 8
19 932 426 131 330	19 874 570 579 461	19 808 537 670 866	20 193 007 285 170	20 049 516 487 6
19 601 484 108 285	19 811 095 874 249	19 531 140 468 808	19 927 097 803 468	19 882 568 167 0
19 739 854 781 601	19 756 184 698 826	19 615 485 537 683	19 933 379 959 308	19 991 027 049 3
19 818 265 211 124	19 831 232 101 619	19 663 798 623 225	20 008 426 425 976	19 949 082 972 7