An Anatomy of Optimized Matrix Multiplication in AArch64

www.huawei.com

Haochen Wang, Tomasz Czajkowski, Ehsan Amiri
Huawei’s Bisheng compiler team, developed from the llvm-project

HUAWEI TECHNOLOGIES CO., LTD.

Motivation

- There are many mature techniques for matrix multiplication:
  - Tiling and Packing
    - Steven Muchnick; Muchnick and Associates (15 August 1997). Advanced Compiler Design Implementation. Morgan Kaufmann. ISBN 978-1-55860-320-2. tiling.
    - MIT 6.172 Performance Engineering of Software Systems, Fall 2018 Instructor: Charles Leiserson. Lecture 1. Introduction and Matrix Multiplication
  - Vectorization and SIMD instructions
    - Optimizing C Code with Neon Intrinsics (arm.com). https://developer.arm.com/documentation/102467/0100/Matrix-multiplication-example
    - https://developer.arm.com/architectures/instruction-sets/intrinsics/
  - Outer product expansion
    - Kurzak, Jakub & Gates, Mark & Yarkhan, Asim & Yamazaki, Ichitaro & Wu, Panruo & Luszczek, Piotr & Finney, Jamie & Dongarra, Jack. (2018). Parallel BLAS Performance Report.
- But their effectiveness doesn't scale well over a wide range of matrix sizes.
- We present our work on choosing the appropriate techniques for different matrix sizes, and how to best combine the techniques and sizes together.

Matrix Multiplication (MM)

- Widely used in many algorithms
  - Solver of linear equation systems
  - Training machine learning models
  - Rendering computer graphics

$$\begin{pmatrix} A1 & B1 \\ C1 & D1 \end{pmatrix} \times \begin{pmatrix} A2 & B2 \\ C2 & D2 \end{pmatrix} = \begin{pmatrix} A1A2 + B1C2 & A1B2+B1D2 \\ C1A2 + D1C2 & C1B2+D1D2 \end{pmatrix}$$

- C+=A*B
- Double precision floating point
- Single-thread

Performance Results

- Theoretical maximum for the testing machine is 10.4 GFLOPs (double precision)
- Single-thread

Matrix size	Performance (GFLOPs)
128	9.92
512	9.50
1024	9.12
1536	9.88
2048	9.90
2560	9.93
3072	9.94
3584	9.95
4096	9.97
32768 (= 2^15)	9.89

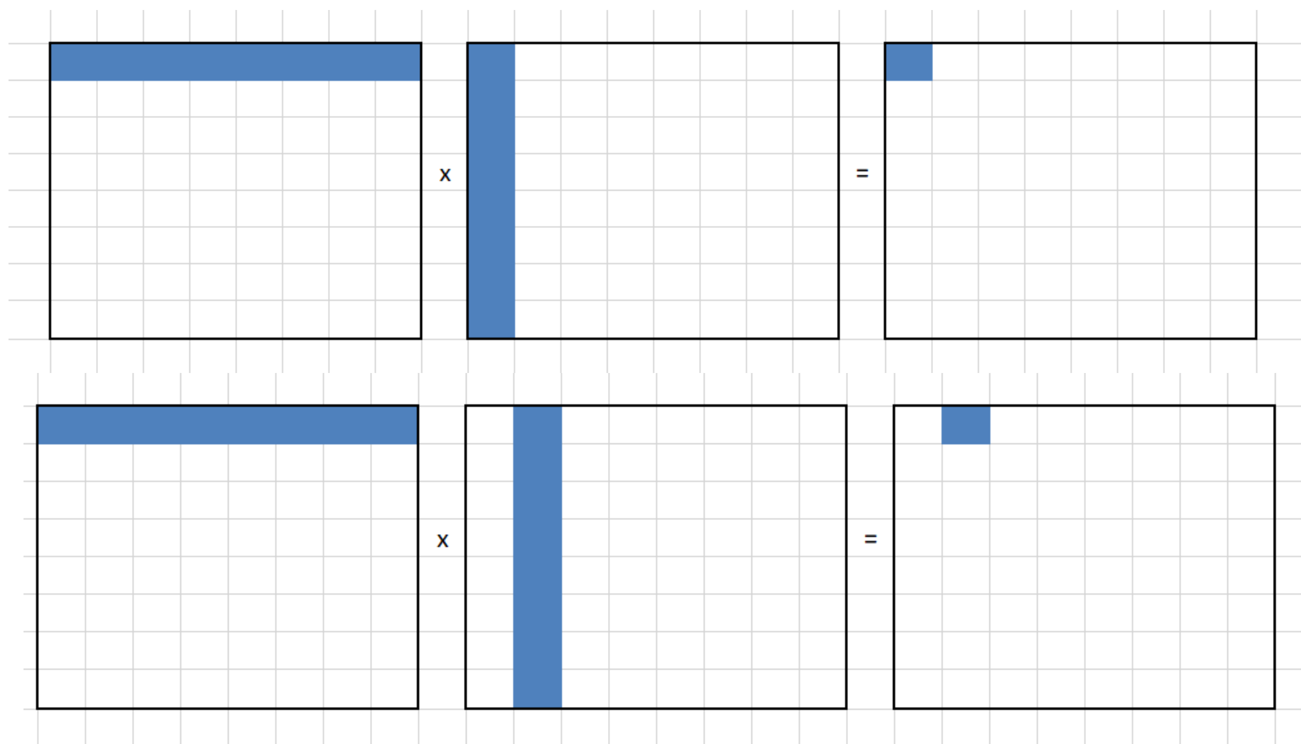
* For the ease of tabulating performance results, only square MM performance measurements are shown. Rectangular MM of similar sizes have similar performance.

* GFLOPs = giga (10^9) floating-point operations per second

Why is MM so slow?

- Unnecessary reloads: the same source data undergoing multiple load instructions. $O(n^3)$ loads from each matrix.

- At least one of the source matrices breaks cache locality



mapping between array `double *a` and the matrix A is

$$\begin{pmatrix} A(0,0) & A(0,1) & A(0,2) \\ A(1,0) & A(1,1) & A(1,2) \\ A(2,0) & A(2,1) & A(2,2) \end{pmatrix} = \begin{pmatrix} \mathrm{a}[0] & \mathrm{a}[3] & \mathrm{a}[6] \\ \mathrm{a}[1] & \mathrm{a}[4] & \mathrm{a}[7] \\ \mathrm{a}[2] & \mathrm{a}[5] & \mathrm{a}[8] \end{pmatrix}$$

Countering reloads

- The more registers you have, the fewer reloads you have to do.
  - Want to exploit all the available registers

- We talk about AArch64 in this talk. It has 32 vector registers available, each can hold 2 doubles (128 bits)
  - NEON
  - https://developer.arm.com/architectures/instruction-sets/intrinsics/

- If matrix is small enough, then all matrix elements can be held inside these registers.

The MM Hierarchy

4x4

- All matrix elements can fit in vector registers entirely
- No need for reloads at all

128x128

- Need reloads into vector registers as we have more matrix elements
- Any way to avoid some of these reloads?
- Cache locality isn't that bad yet

1024x1024 (and higher)

- Cache locality degrades drastically
- Tiling/packing are needed

The small: 4x4 MM

Fits entirely in the registers

The 4x4 microcore

- 32 vector registers on AArch64, 2 doubles each
- 8 regs for A, 8 regs for C, 16 regs for B; each element loaded just once
- Reminder: column major order

a00	a01	a02	a03		b00	b01	b02	b03		c00	c01	c02	c03
a10	a11	a12	a13	x	b10	b11	b12	b13	=	c10	c11	c12	c13
a20	a21	a22	a23		b20	b21	b22	b23		c20	c21	c22	c23
a30	a31	a32	a33		b30	b31	b32	b33		c30	c31	c32	c33

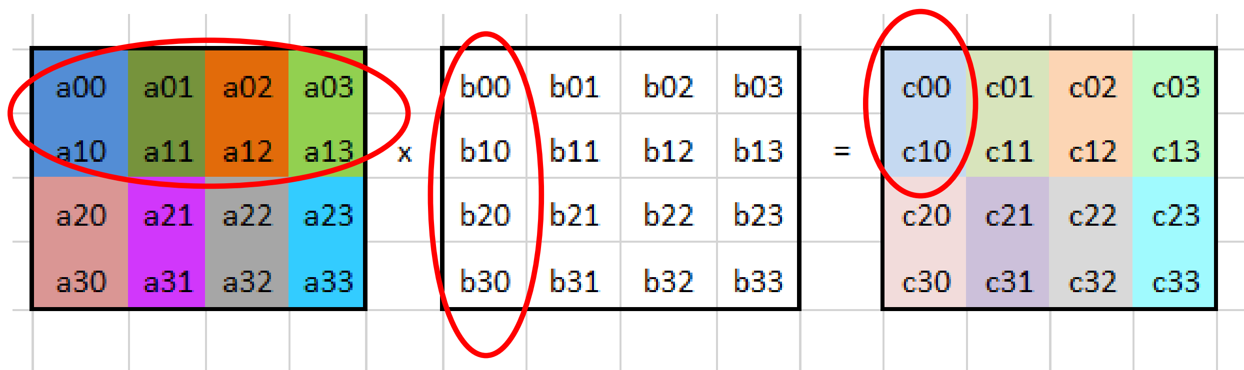
The 4x4 microcore

```
dup v30.2d, b00
dup v31.2d, b10
…

load v0.2d, (a00, a10)
load v2.2d, (a01, a11)
load v1.2d, (c00, c10)
…

// every matrix element fits in the vec regs, no reloads whatsoever
fmla v1.2d, v0.2d, v30.2d
fmla v1.2d, v2.2d, v31.2d
……

store v1.2d, (c00, c10)
….
```




```
%1 = load <2 x double>, …
%2 = call <2 x double> @llvm.fma.v2f64(…, …)
store <2 x double>, …
```

* To accomplish dup, might need help from extractelement, insertelement, shufflevector

The medium: 128x128 MM

Any way to avoid some of these reloads?

Outer Product Expansion (OPE)

- Matrix multiplication can be done in arbitrary blocks, as long as the blocks are of legal dimensions. In the example A, B, C, D can be plain numbers or matrices.

$$\begin{pmatrix} A1 & B1 \\ C1 & D1 \end{pmatrix} X \begin{pmatrix} A2 & B2 \\ C2 & D2 \end{pmatrix} =$$

$$\begin{pmatrix} A1A2 + B1C2 & A1B2+B1D2 \\ C1A2 + D1C2 & C1B2+D1D2 \end{pmatrix}$$

Outer Product Expansion (OPE)

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix} \begin{bmatrix} 4 & 5 & 6 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 4 & 5 & 6 \\ -4 & -5 & -6 \\ 4 & 5 & 6 \end{bmatrix}$$

p=0 p=1

Outer Product Expansion (OPE)

- The (m,n)-th element in the product

= inner product between m-th row of A

and n-th col of B

But this inner product only has one item, so it’s just a single multiplication.

$$\begin{bmatrix}1\\1\\1\end{bmatrix}\begin{bmatrix}1 & 2 & 3\end{bmatrix} = \begin{bmatrix}1 & 2 & 3\\1 & 2 & 3\\1 & 2 & 3\end{bmatrix}$$

Outer Product Expansion (OPE)

- The (m,n)-th element in the product

= inner product between m-th row of A

and n-th col of B

But this inner product only has one item, so it’s just a single multiplication.

$$\begin{bmatrix}1\\1\\1\end{bmatrix}\begin{bmatrix}1 & 2 & 3\end{bmatrix} = \begin{bmatrix}1 & 2 & 3\\1 & 2 & 3\\1 & 2 & 3\end{bmatrix}$$

Outer Product Expansion (OPE)

- The (m,n)-th element in the product

= inner product between m-th row of A

and n-th col of B

But this inner product only has one item, so it's just a single multiplication.

$$\begin{bmatrix}1\\1\\1\end{bmatrix}\begin{bmatrix}1 & 2 & 3\end{bmatrix} = \begin{bmatrix}1 & 2 & 3\\1 & 2 & 3\\1 & 2 & 3\end{bmatrix}$$

Outer Product Expansion (OPE)

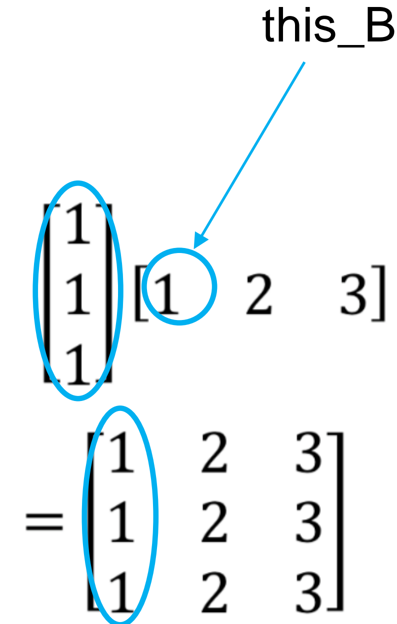
- The (m,n)-th element in the product
= inner product between m-th row of A
  and n-th col of B

$$\begin{bmatrix}1\\1\\1\end{bmatrix}\begin{bmatrix}1 & 2 & 3\end{bmatrix}$$

$$=\begin{bmatrix}1 & 2 & 3\\1 & 2 & 3\\1 & 2 & 3\end{bmatrix}$$

But this inner product only has one item, so it’s just a single multiplication.

OPE inherently supports loop invariant code motion

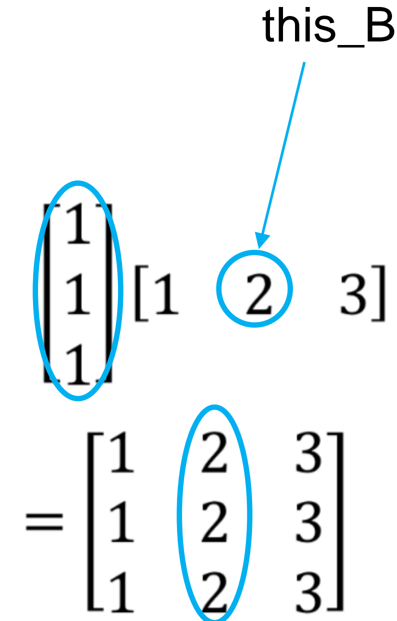


Inspect the p=0 outer product

```
for (i in the current B row):
   this_B = B(i,p=0)
   for (j in the current A col):
      C(i,j) += A(i,j)*this_B
```

- The load of B(i,0) is lifted from the innermost loop
- Each B(i,p) is loaded only once! $O(n^2)$ loads from B.
- A single outer product has $O(n^2)$ loads from A, so a total of $O(n^3)$ loads from A for n outer products in the whole MM

OPE inherently supports loop invariant code motion

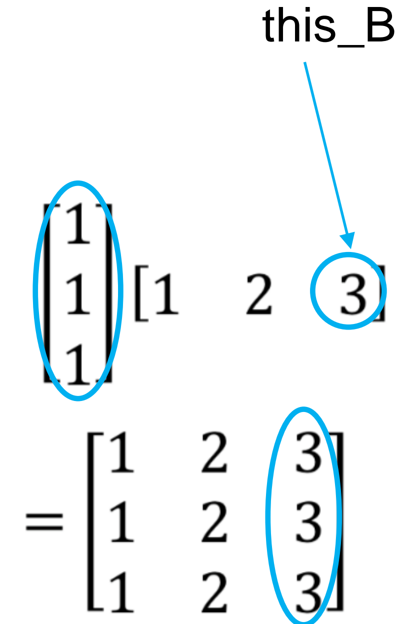


Inspect the p=0 outer product

```
for (i in the current B row):
   this_B = B(i,p=0)
   for (j in the current A col):
      C(i,j) += A(i,j)*this_B
```

- The load of B(i,0) is lifted from the innermost loop
- Each B(i,p) is loaded only once! $O(n^2)$ loads from B.
- A single outer product has $O(n^2)$ loads from A, so a total of $O(n^3)$ loads from A for n outer products in the whole MM

OPE inherently supports loop invariant code motion



$$\begin{bmatrix}1\\1\\1\end{bmatrix}\begin{bmatrix}1 & 2 & 3\end{bmatrix} = \begin{bmatrix}1 & 2 & 3\\1 & 2 & 3\\1 & 2 & 3\end{bmatrix}$$

<u>Inspect the p=0 outer product</u>

```
for (i in the current B row):
   this_B = B(i,p=0)
   for (j in the current A col):
      C(i,j) += A(i,j)*this_B
```

- The load of B(i,0) is lifted from the innermost loop
- Each B(i,p) is loaded only once! O(n^2) loads from B.
- A single outer product has O(n^2) loads from A, so a total of O(n^3) loads from A for n outer products in the whole MM

How about loads from C?

Outer product procedure:

1. Do the p-th outer product and store into C in main memory.

```
for i:
  for j:
    C(i, j) += …
```

2. Switch to the next outer product (p++).

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix} \begin{bmatrix} 4 & 5 & 6 \end{bmatrix}$$

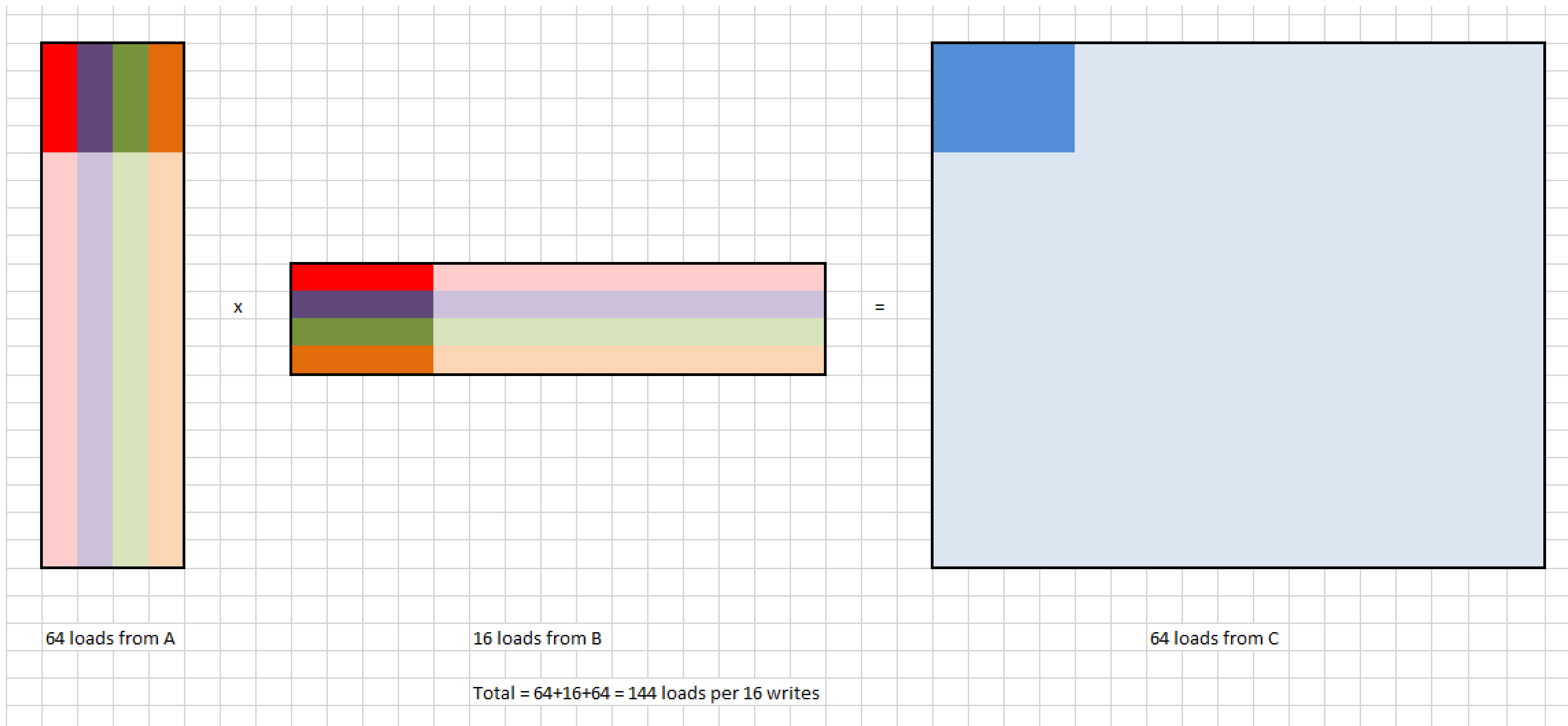
$$= \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 4 & 5 & 6 \\ -4 & -5 & -6 \\ 4 & 5 & 6 \end{bmatrix}$$

A total of $O(n^3)$ loads from C

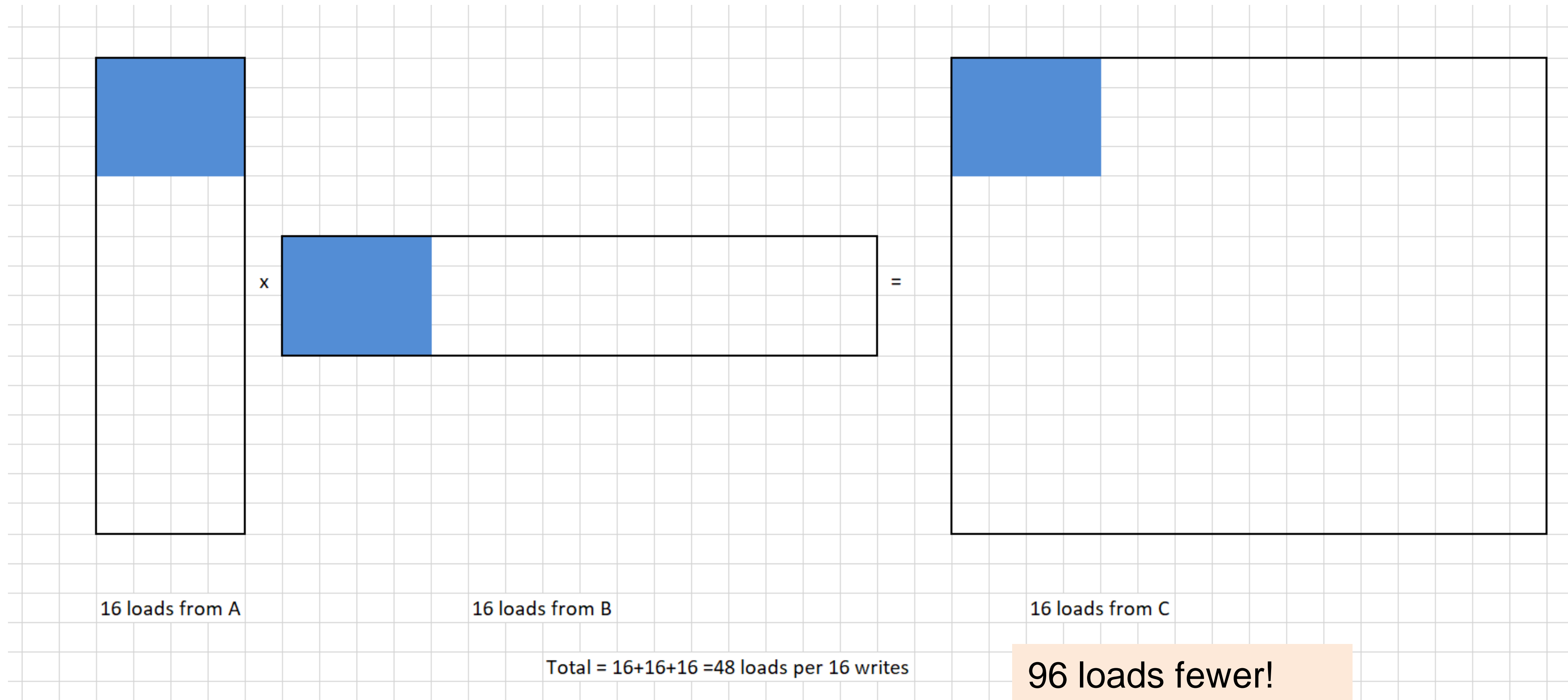
Benefits from OPE

- $O(n^3)$ loads from A and C, $O(n^2)$ loads from B
- Try further reducing A and C loads

A (mx4)x(4xn) outer product



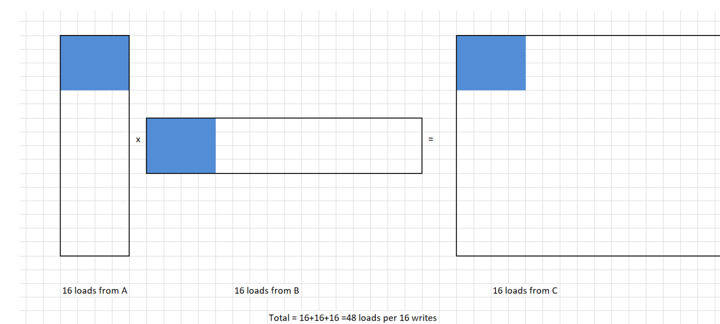
- Do 4x4 matrix fma with all entries from A, B and C loaded only once, with the 4x4 microcore. Then the outer product looks like this.
- Of course, lift the loads from B for the same mx4 column of A



Loop Invariant Code Motion, the 4x4 version

a00	a01	a02	a03		b00	b01	b02	b03		c00	c01	c02	c03
a10	a11	a12	a13	x	b10	b11	b12	b13	=	c10	c11	c12	c13
a20	a21	a22	a23		b20	b21	b22	b23		c20	c21	c22	c23
a30	a31	a32	a33		b30	b31	b32	b33		c30	c31	c32	c33

move the 4x4 from B into vec reg, takes up 16 regs



for (a 4x4 in the mx4 column of A){

load the 4x4 from A and C, takes up 8 regs each

do the 4x4 MM

}

unroll, prefetch, …

- Exploiting fully out of the 32 vec regs!

Results of the first version

- Theoretical maximum for the testing machine is 10.4 GFLOPs (double precision)
- Single-thread

Matrix size	Performance (GFLOPs)
128	10.0
512	8.2
1024	7.8
1536	7.7
2048	6.9

- This version is very fast at small sizes but degrades very quickly.
- Use as a 128x128 macrocore!

The large: 1028x1028 MM

Cache considerations and tiling

The problem: column jumps are too big when loading

- Here the (a00, a10) and (a01, a11) register loads are 128/2048 addresses apart in main memory if the matrix size is 128/2048.
- Bad caching due to poor special locality

a00	a01	a02	a03		b00	b01	b02	b03		c00	c01	c02	c03
a10	a11	a12	a13	x	b10	b11	b12	b13	=	c10	c11	c12	c13
a20	a21	a22	a23		b20	b21	b22	b23		c20	c21	c22	c23
a30	a31	a32	a33		b30	b31	b32	b33		c30	c31	c32	c33

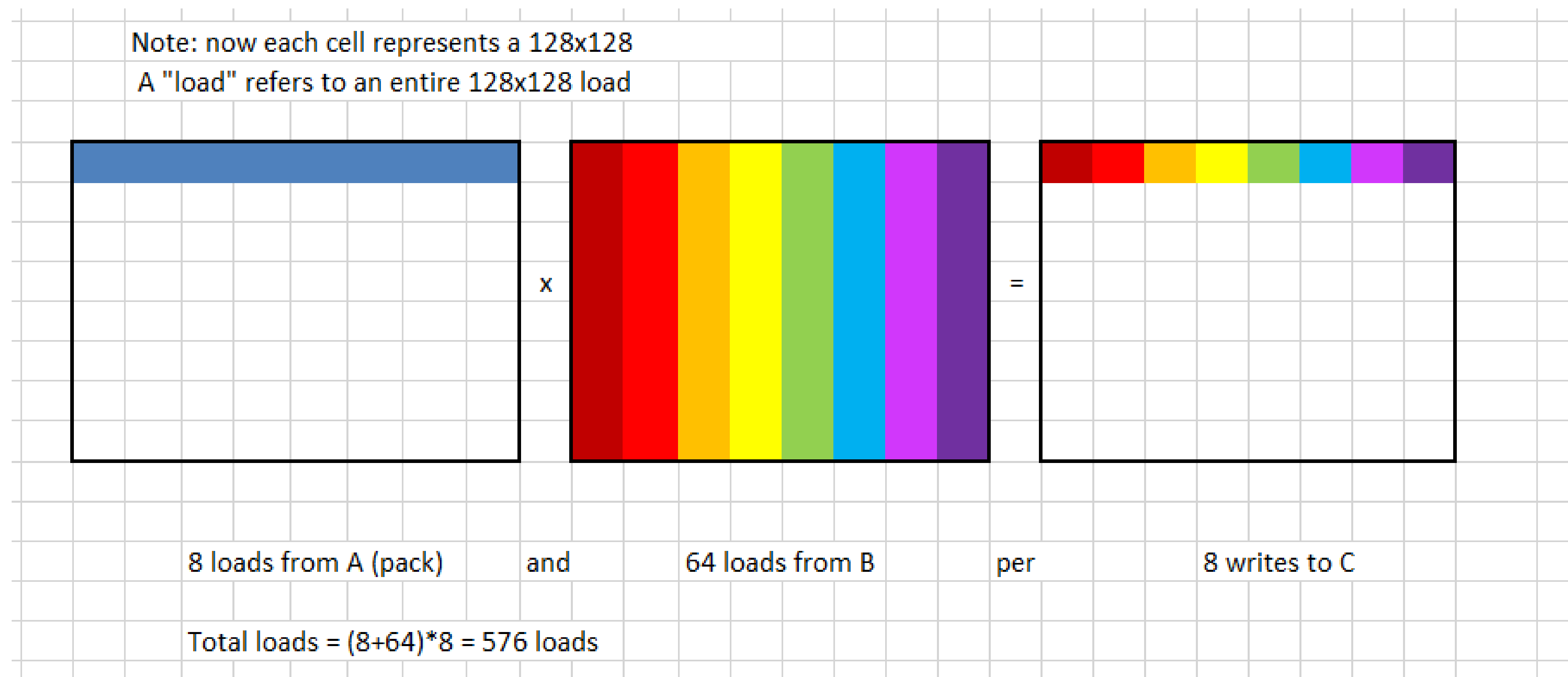
Using temporary arrays for macrocore

- Therefore we want to do the entire 2048x2048 MM in 128x128 blocks.

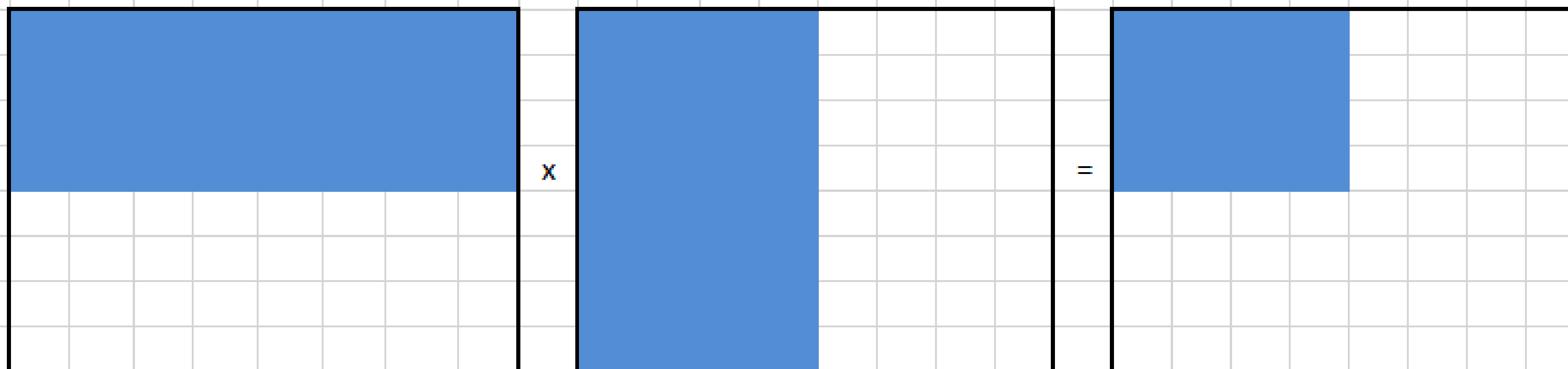
- Specifically, we want three temporary arrays:
  - cur_a = (double *)malloc(sizeof(double)*128*128);
  - cur_b = (double *)malloc(sizeof(double)*128*128);
  - cur_c = (double *)malloc(sizeof(double)*128*128);

- … pack the 128x128 blocks into these temporary arrays, and use the first method on these temporary arrays. We want to do this because we know the first method is fast (10G!) on 128x128 arrays.

How to schedule the 128x128 macrocores?

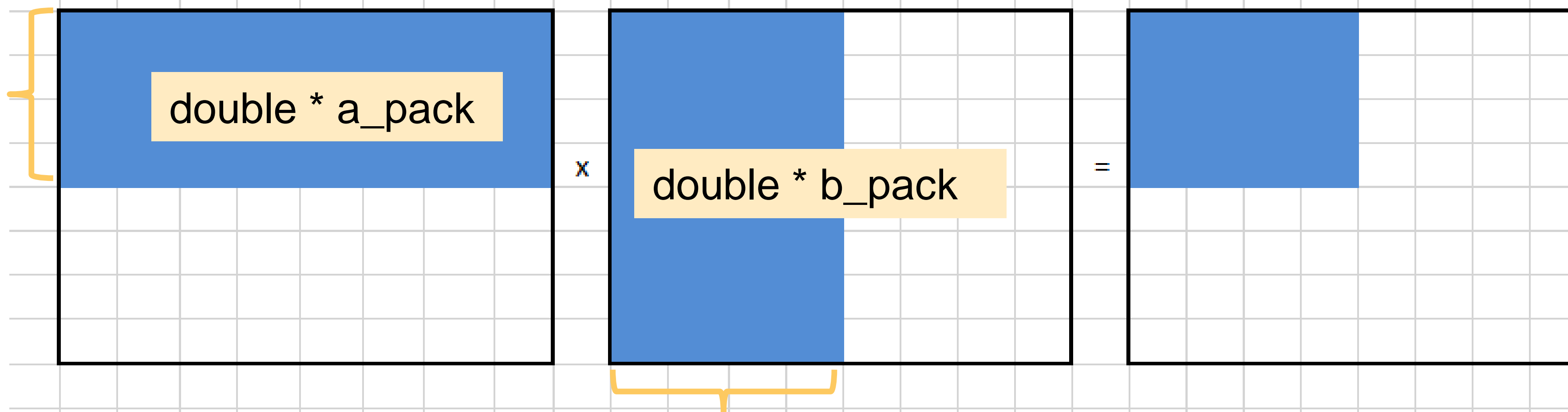


How to schedule the 128x128 macrocores?



HUAWEI

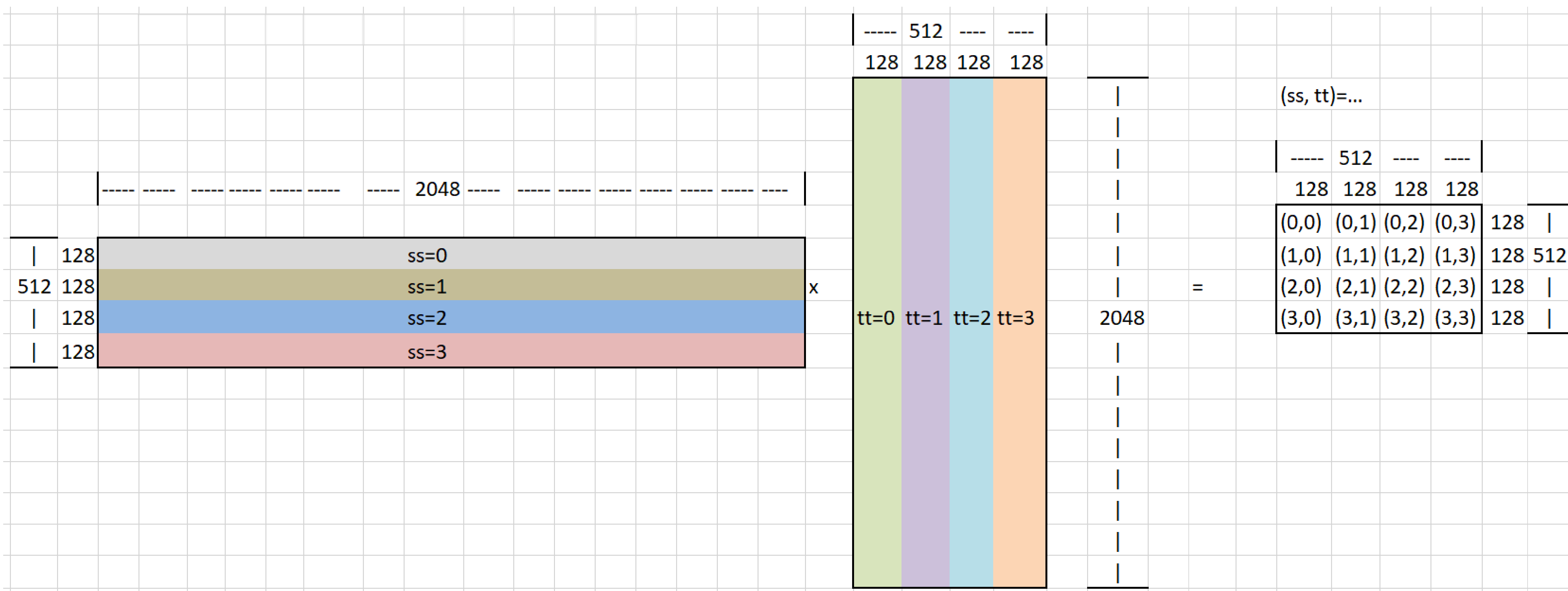
How to schedule the 128x128 macrocores?



Each tile has 4 macrocore-sized row/column panels

HUAWEI

How to schedule the 128x128 macrocores?

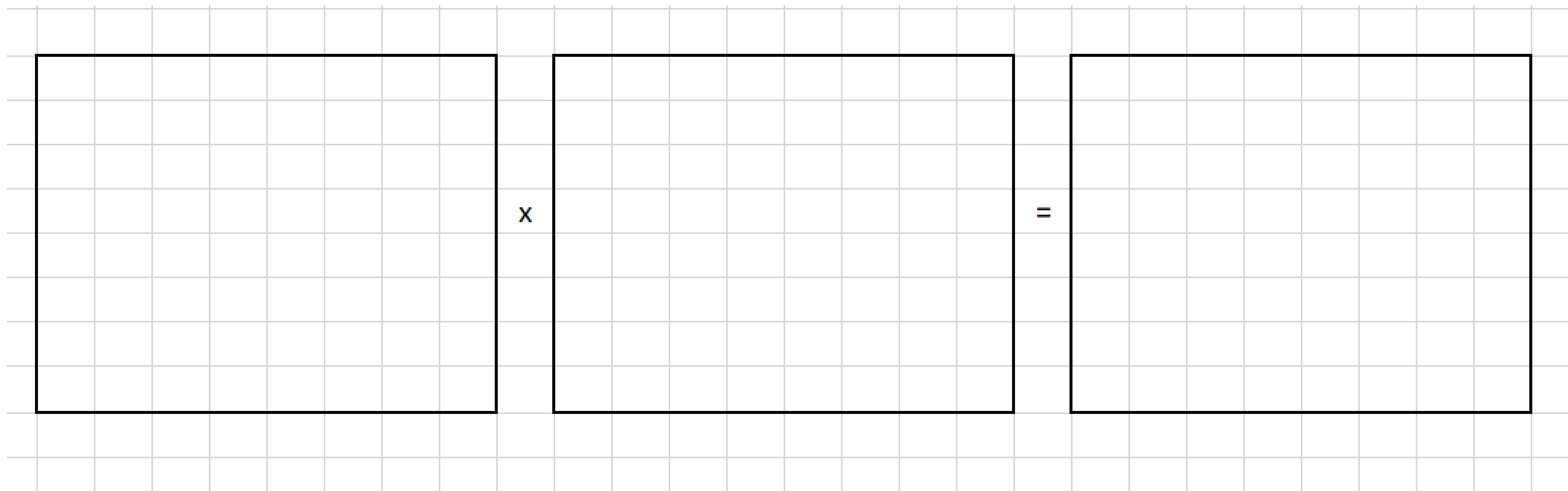


Packing the tiles

- Problem with packing of the tiles: need an extra load from the tile's packing array to temporary 128x128 arrays

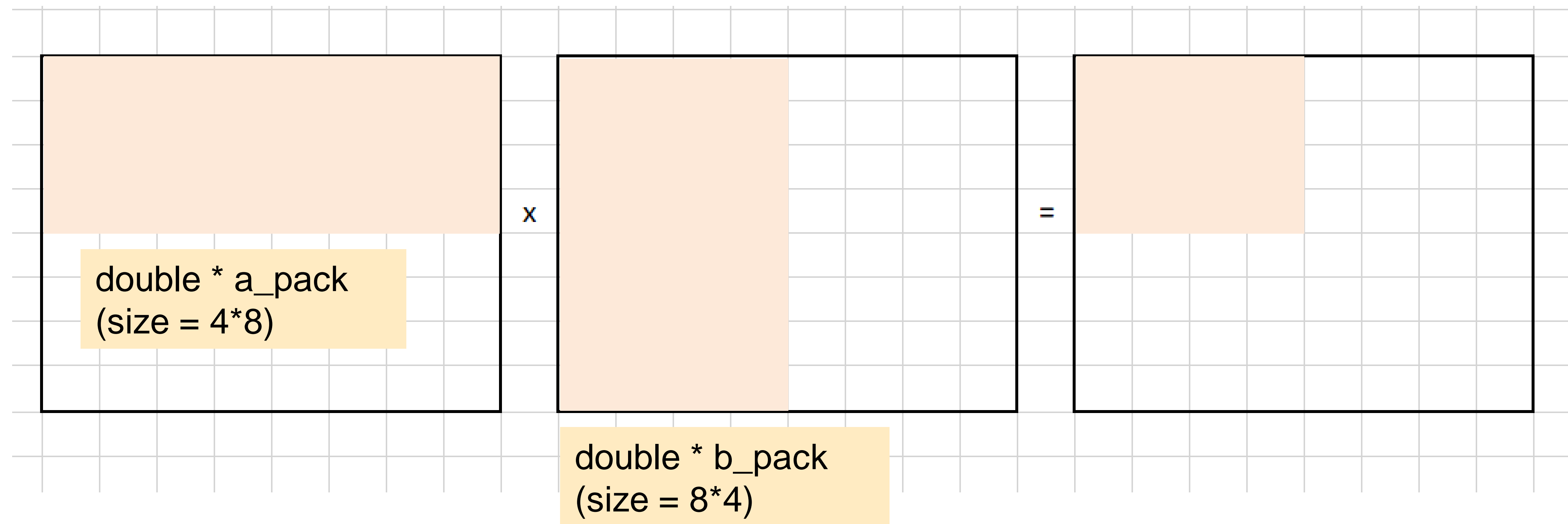
- To illustrate, imagine a macrocore size of 2x2, and a tile size of 4.
  - Each tile has 2 rows/columns of macrocore sized panels.

Packing the tiles

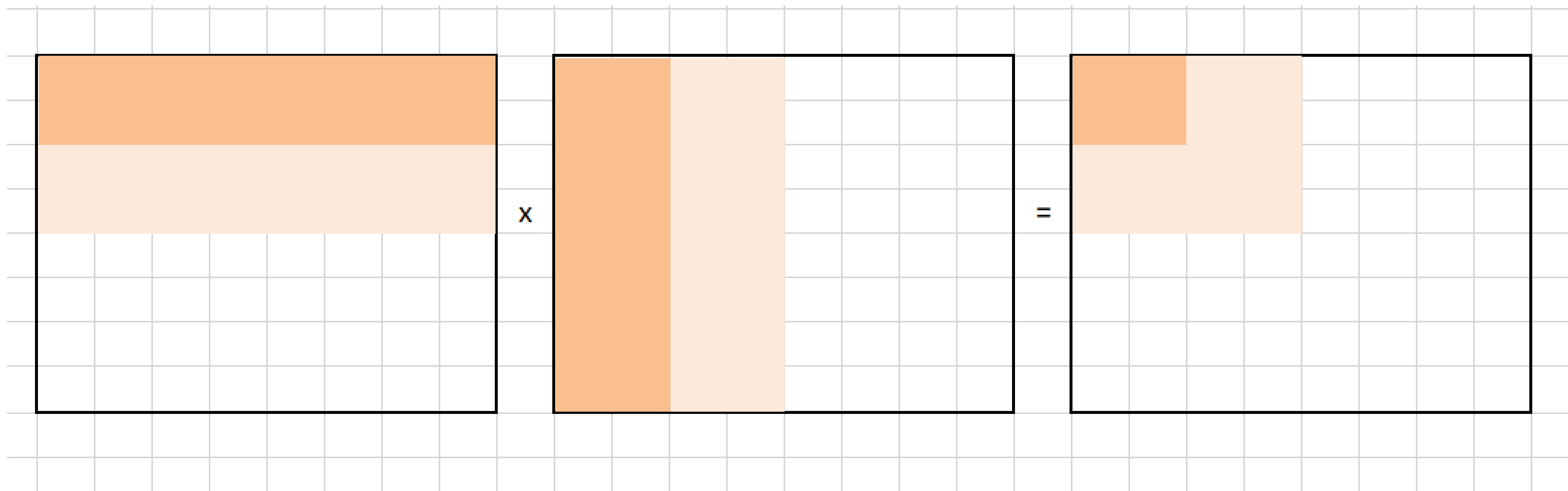


HUAWEI

Packing the tiles

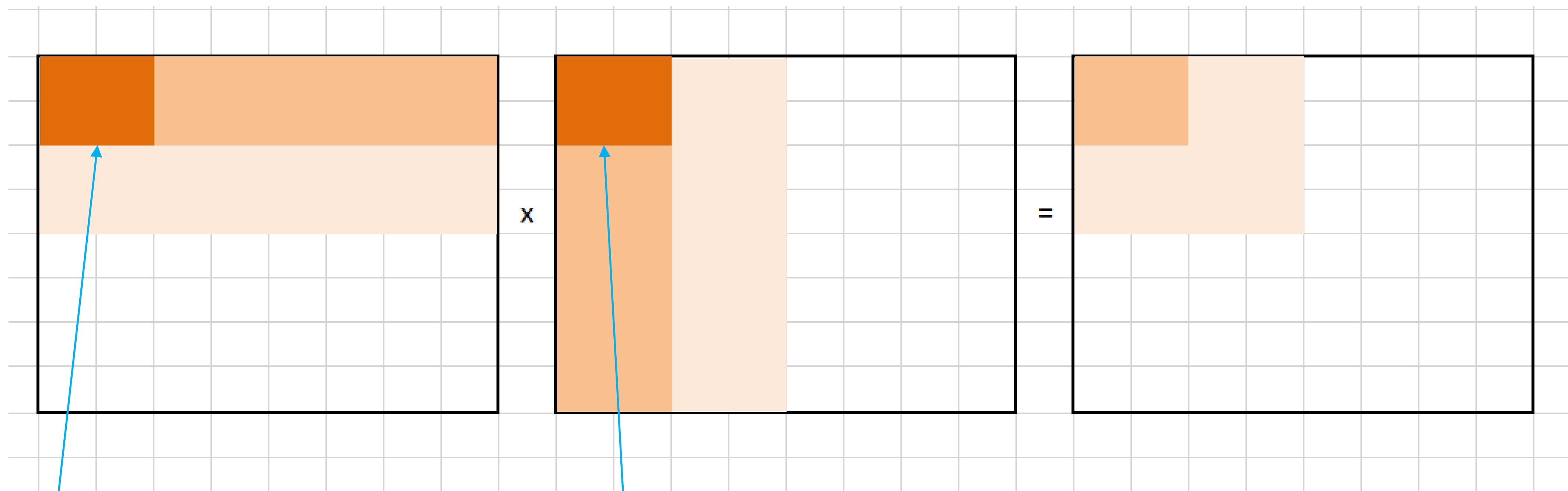


Packing the tiles



HUAWEI

Packing the tiles

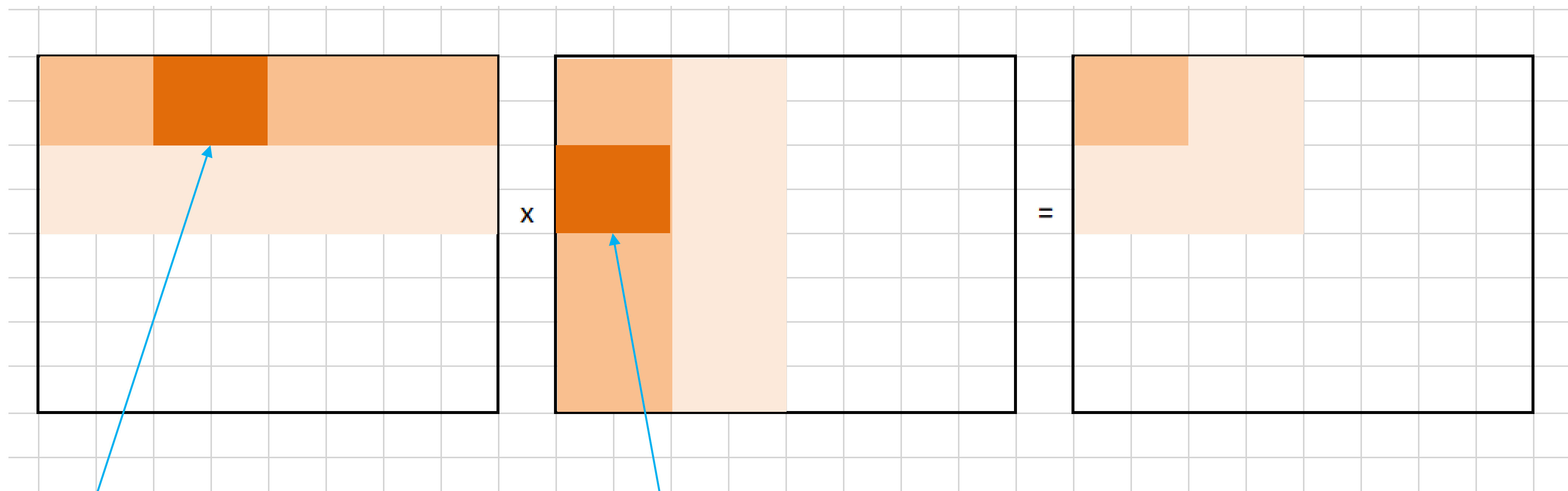


double * cur_a
(size = 2*2)

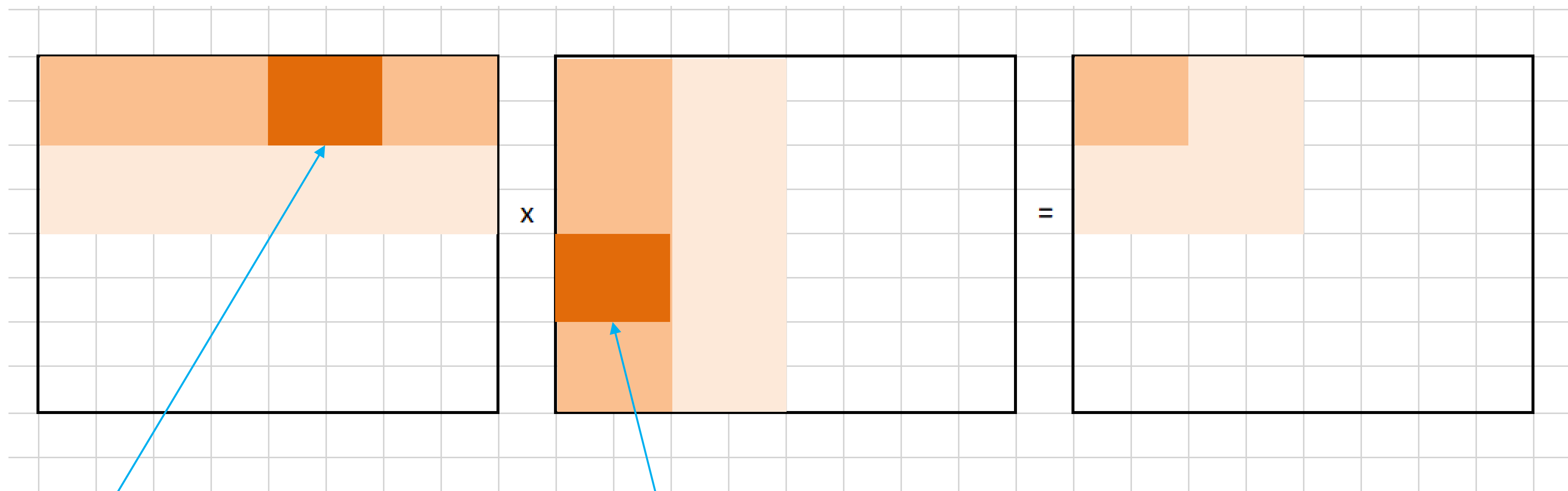
double * cur_b
(size = 2*2)

HUAWEI

Packing the tiles



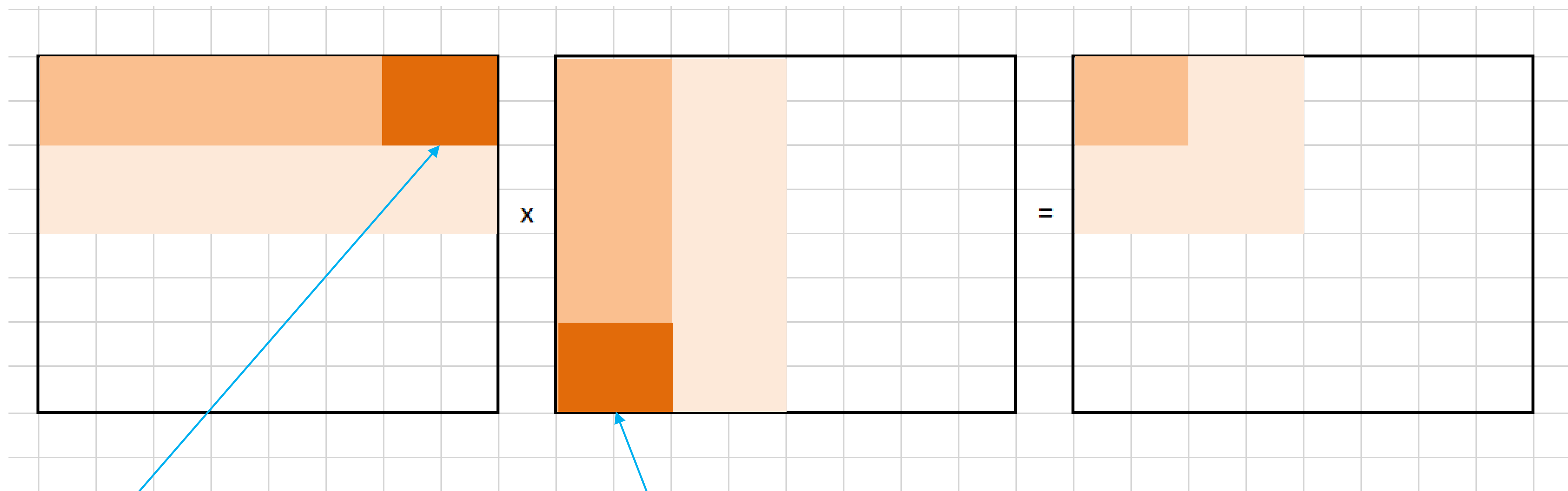
Packing the tiles



double * cur_a
(size = 2*2)

double * cur_b
(size = 2*2)

Packing the tiles

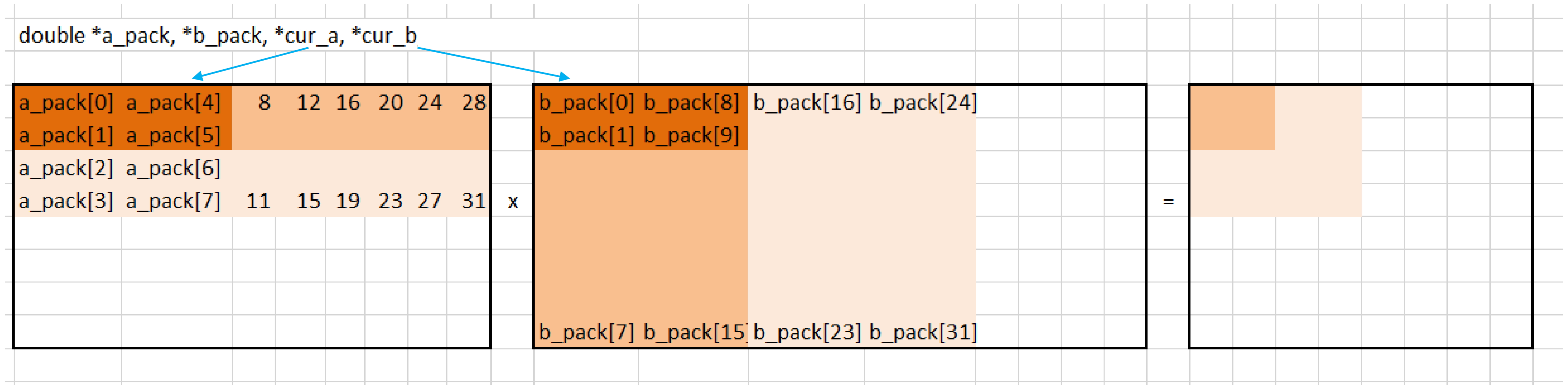


double * cur_a
(size = 2*2)

double * cur_b
(size = 2*2)

Packing the tiles

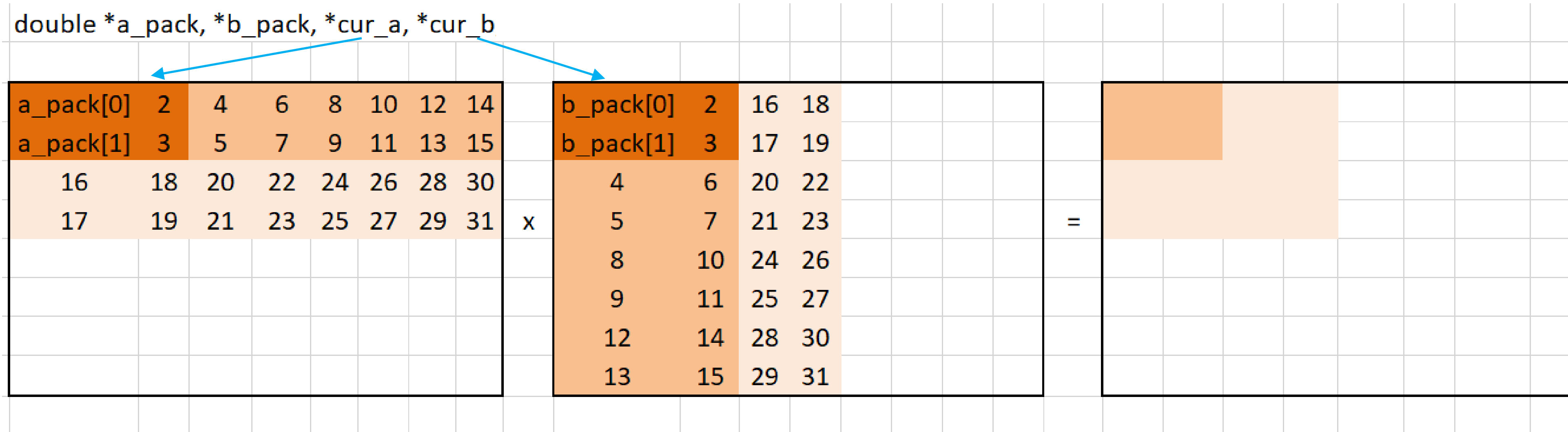
Note: the numbers in this figure are addresses within the packing array, not values of matrix elements.



- Need to load from the pack arrays into cur_a and cur_b arrays, since the 2x2 macrocores are not in contiguous locations in the pack

Packing the tiles

Note: the numbers in this figure are addresses within the packing array, not values of matrix elements.



- No need to load from the packing arrays into cur_a and cur_b arrays, since the 2x2 macrocores are in contiguous locations in the pack.
- To go to the next macrocore, simply increment the pointer!

```
cur_a += 2*2; cur_b += 2*2;
```

Packing the tiles

Note: the numbers in this figure are addresses within the packing array, not values of matrix elements.

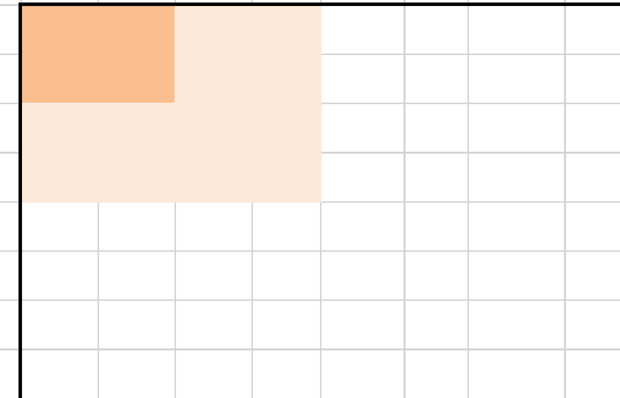
```
double *a_pack, *b_pack, *cur_a, *cur_b
```

a_pack[0]	2	4	6	8	10	12	14
a_pack[1]	3	5	7	9	11	13	15
16	18	20	22	24	26	28	30
17	19	21	23	25	27	29	31

x

b_pack[0]	2	16	18
b_pack[1]	3	17	19
4	6	20	22
5	7	21	23
8	10	24	26
9	11	25	27
12	14	28	30
13	15	29	31

=



- No need to load from the packing arrays into cur_a and cur_b arrays, since the 2x2 macrocores are in contiguous locations in the pack.
- To go to the next macrocore, simply increment the pointer!

```
cur_a += 2*2; cur_b += 2*2;
```

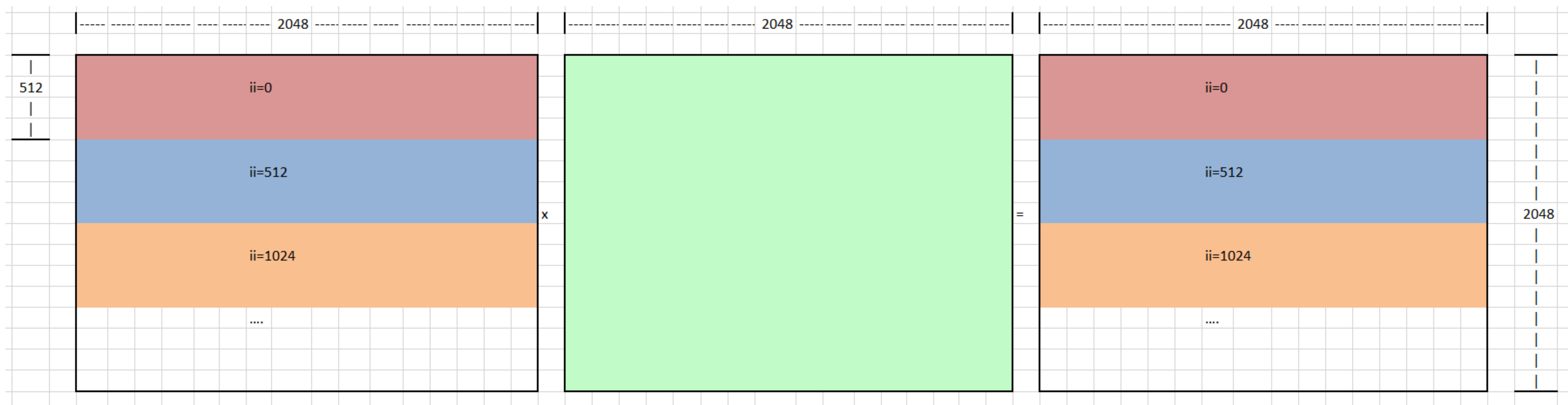

HUAWEI

Complete summary

HUAWEI

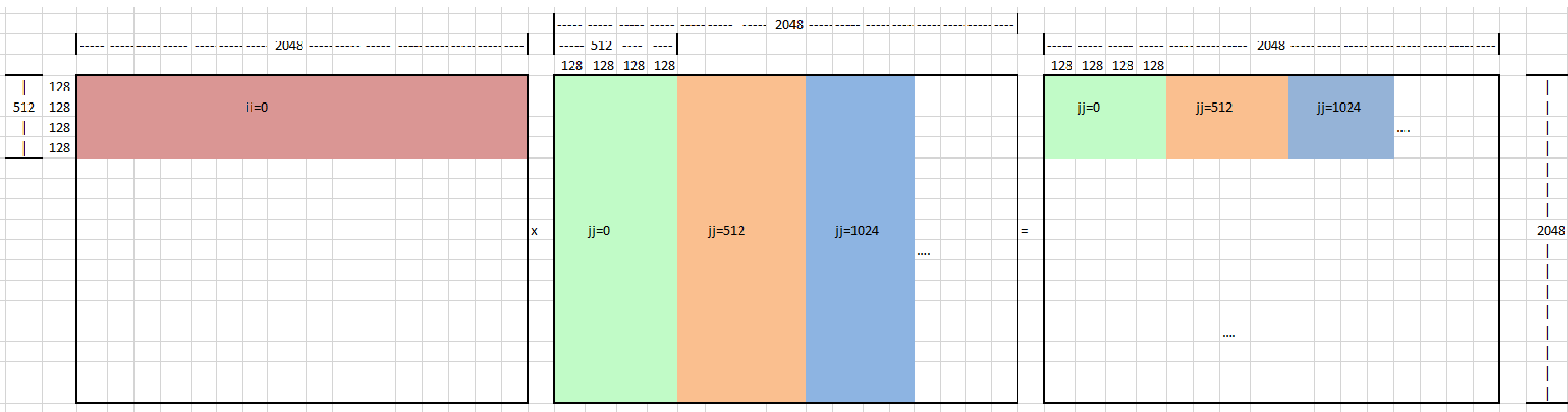
1. Tile A into row tiles. In the demo tile size=512

Pack the A row tiles in the zigzag fashion



2. Tile B into column tiles. In the demo tile size = 512.

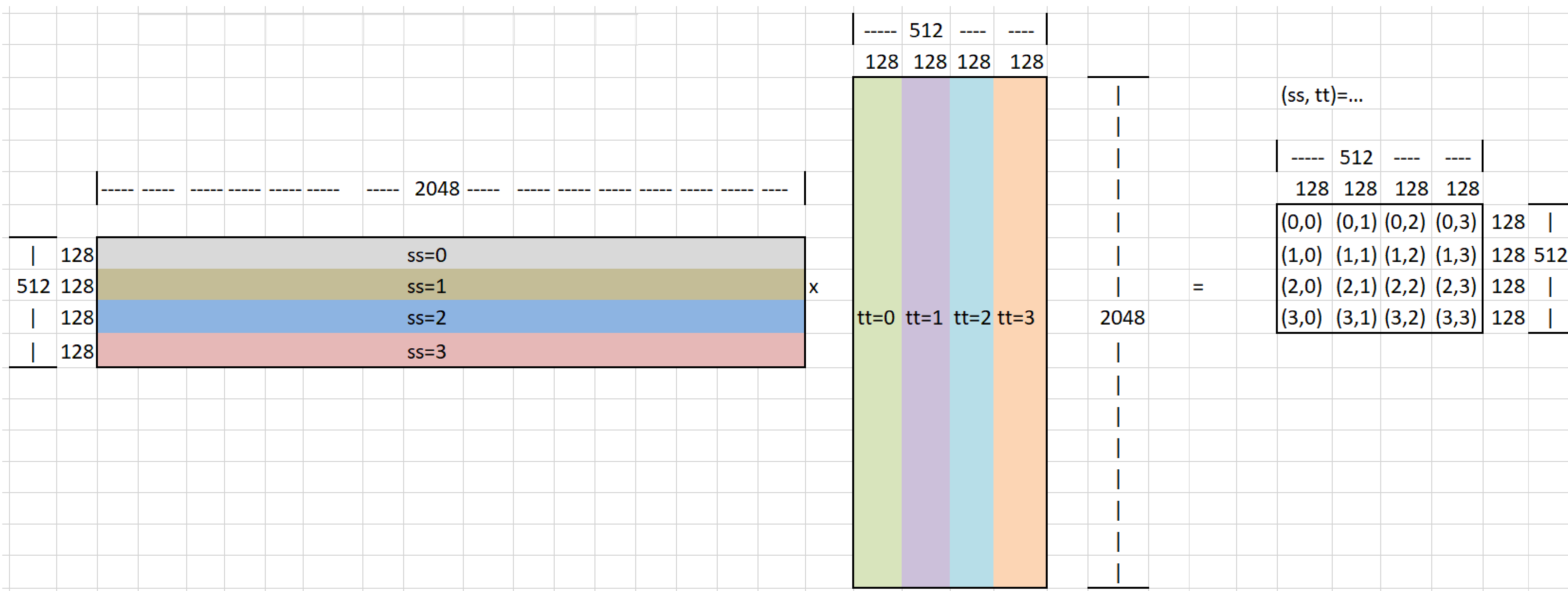
Pack the B column tiles in the zigzag fashion



HUAWEI

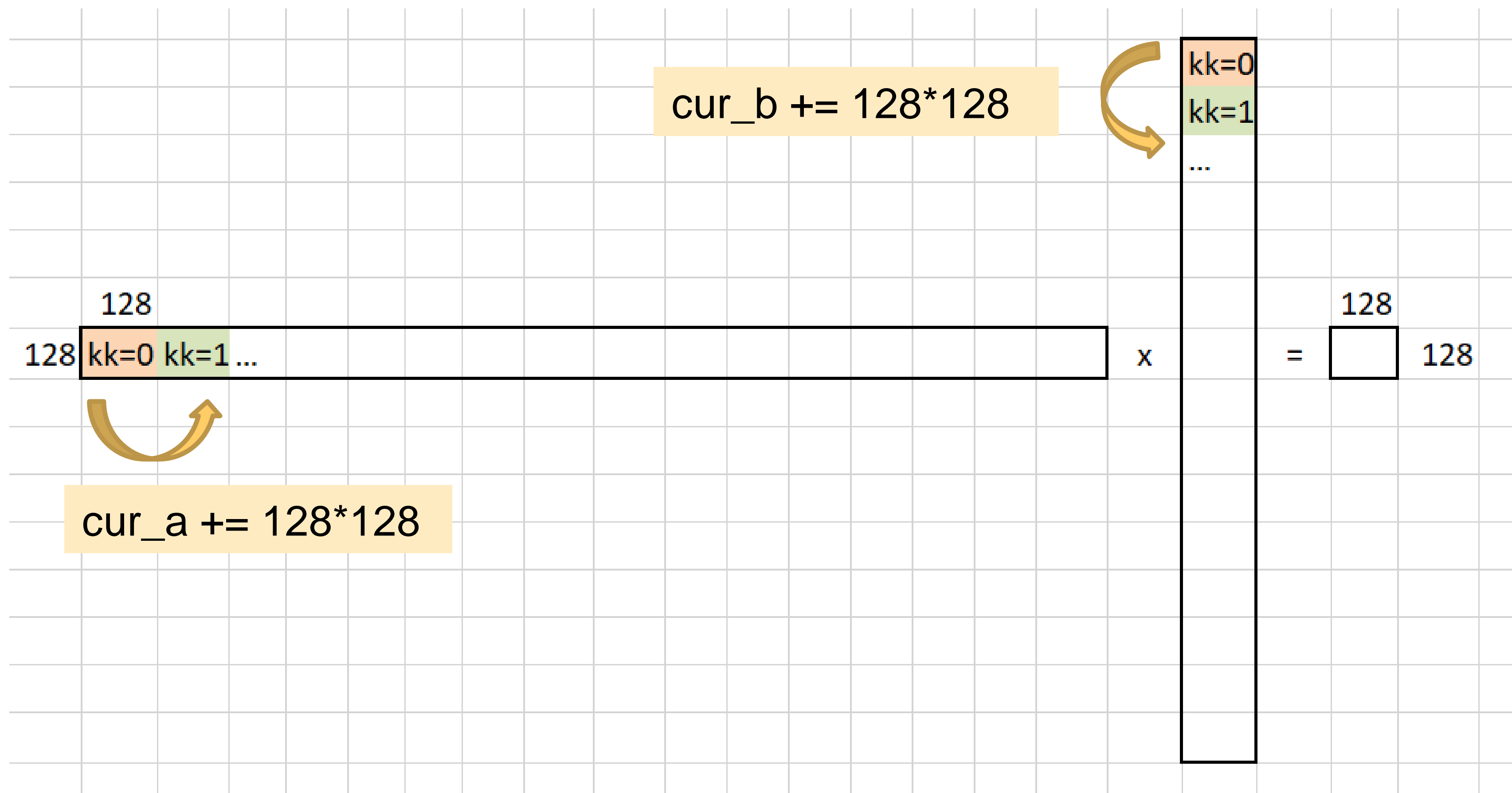
3. Within the inner product for a tile (i.e. within the same jj),

do (tile size/macrocore size)^2 macrocore-sized inner products.



HUAWEI

4. Within one macrocore-sized inner product, use zigzag packing and iterate by simply incrementing pointers



HUAWEI

Within a 128x128 macrocore:

- Use the first version, i.e. outer product + 4x4_microcore

Thank you

…and thanks to excel that made these otherwise really annoying graphs possible