Revisiting Matrix Multiplication

6.506 Algorithm Engineering
May 11, 2023

Tao B. Schardl
MIT CSAIL

Recap: 6.106 lecture 1 matrix-multiplication case study

$$C \qquad A \qquad B$$

$$\begin{pmatrix} c_{00} & c_{01} & \cdots & c_{0(n\text{-}1)} \\ c_{10} & c_{11} & & \vdots \\ \vdots & & \ddots & \\ c_{(n\text{-}1)0} & \cdots & & c_{(n\text{-}1)(n\text{-}1)} \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} & \cdots & a_{0(n\text{-}1)} \\ a_{10} & a_{11} & & \vdots \\ \vdots & & \ddots & \\ a_{(n\text{-}1)0} & \cdots & & a_{(n\text{-}1)(n\text{-}1)} \end{pmatrix} * \begin{pmatrix} b_{00} & b_{01} & \cdots & b_{0(n\text{-}1)} \\ b_{10} & b_{11} & & \vdots \\ \vdots & & \ddots & \\ b_{(n\text{-}1)0} & \cdots & & b_{(n\text{-}1)(n\text{-}1)} \end{pmatrix}$$

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$

As in 6.106, we will use $n = 4096$.

Final verdict from 6.106 lecture 1

Version	Implementation	Running time (s)	Relative speedup	Absolute speedup	GFLOPS	Fraction of peak
1	Python	21,041.67	1	1	0.006	0%
2	Java	2,387.32	8.8	9	0.058	0.007%
3	C	1,155.77	2.1	18	0.118	0.014%
4	+ interchange loops	177.68	6.5	118	0.774	0.093%
5	+ optimization flags	54.63	3.3	385	2.516	0.301%
6	Parallel loops	3.04	18	6,921	45.211	5.408%
7	Parallel divide-and-conquer	1.30	1.4	16,197	105.722	12.646%
8	+ compiler vectorization	0.70		30,272	196.341	23.486%
9	+ AVX intrinsics	0.39		53,292	352.408	41.677%
10	Intel MKL	0.41		51,497	335.217	40.098%

Today's focus

Problem: Performance-measurement methodology

Version	Implementation	Running time (s)	Relative speedup	Absolute speedup	GFLOPS	Fraction of peak
1	Python	21,041.67	1	1	0.006	0%
2	Java	2,387.32	8.8			
3	C	1,155.77	2.1			
4	+ interchange loops	177.68	6.5			
5	+ optimization flags	54.63	3.3	385	2.516	0.301%
6	Parallel loops	3.04	18			
7	Parallel divide-and-conquer	1.30	1.4			
8	+ compiler vectorization	0.70	1.9	30,272	196.341	23.486%
9	+ AVX intrinsics	0.39	1.8	53,292	352.408	41.677%
10	Intel MKL	0.41	1	51,497	335.217	40.098%

Each running time is the minimum of 5 runs of a binary.
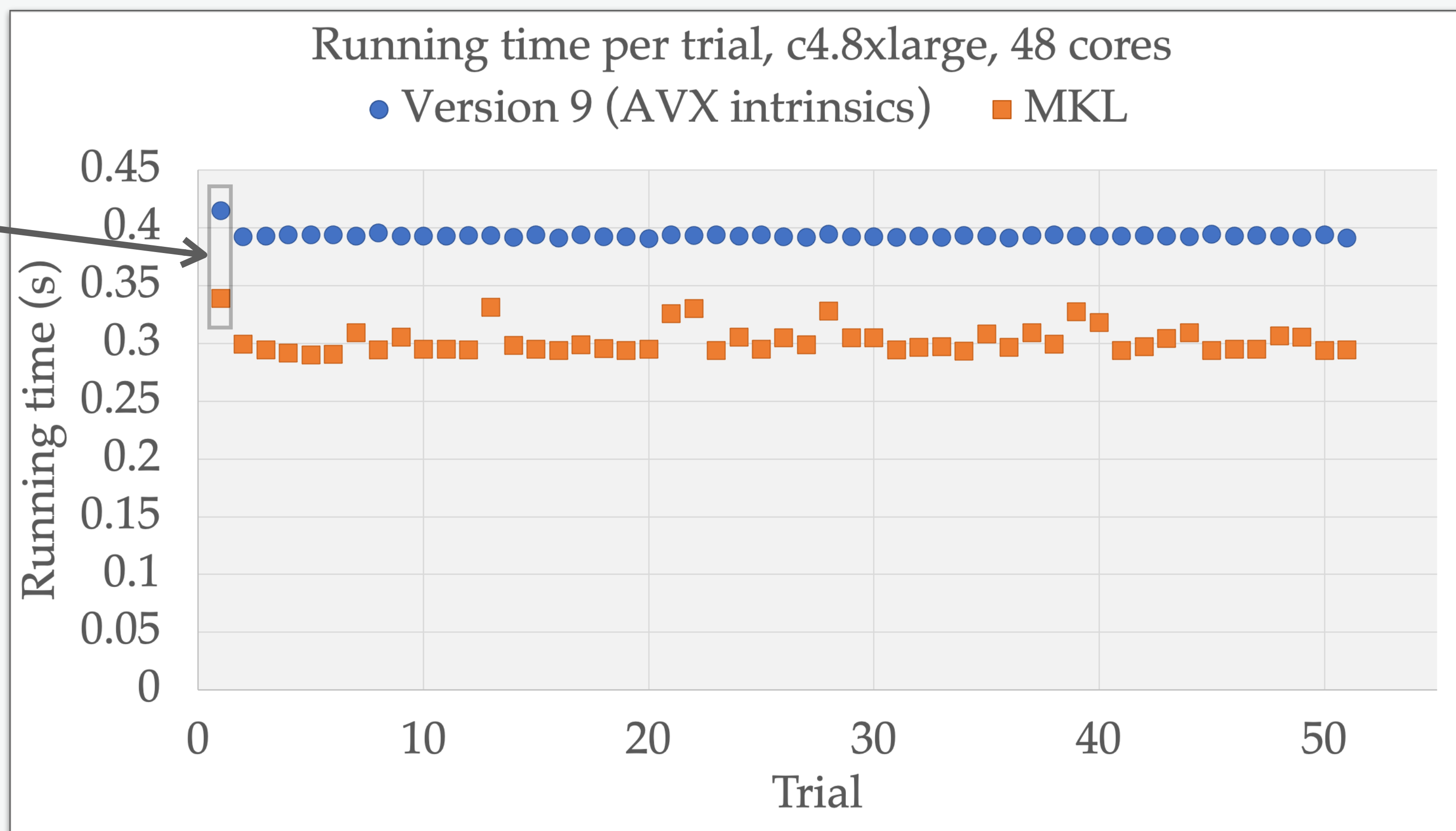
Each binary runs matrix-multiplication **once**.

An overlooked performance gap

What happens if the binary runs matrix-multiplication many times?

The first matrix-multiply call is **slower** than subsequent calls.

Version 9 (AVX intrinsics) takes ~0.39 seconds.

Intel MKL takes ~0.30 seconds!

Running time per trial, c4.8xlarge, 48 cores
● Version 9 (AVX intrinsics)   ■ MKL

The case study's conclusions are fine, but MKL is faster than we thought!

New hardware: AWS c5.metal machine specs

Feature	Specification
Microarchitecture	Cascade Lake (Intel Xeon Platinum 8275CL)
Clock frequency	3.0 GHz
Processor chips	2
Processor cores	24 per processor chip
Floating-point unit	32 double-precision operations, including fused-multiply-add, per core per cycle
Cache-line size	64 B
L1 data cache	32 KB private, 8-way set associa
L2 cache	1 MB private, 16-way set associa
L3 cache	35.75 MB shared, 11-way set assoc
DRAM	189 GB

Theoretical peak performance:
3.0 GHz * 2 * 24 * 32
= 4608 GFLOPS

Alternative oneMKL threading options

Intel oneMKL offers different threading options that give different performance on new hardware.

Version 9 (AVX intrinsics) takes ~0.12 seconds.

oneMKL with TBB takes ~0.11 seconds.

oneMKL with OpenMP takes ~0.06 seconds!

Running time per trial, c5metal, 48 cores

● Version 9 (AVX intrinsics)   ■ oneMKL, TBB threading
◆ oneMKL, OMP threading
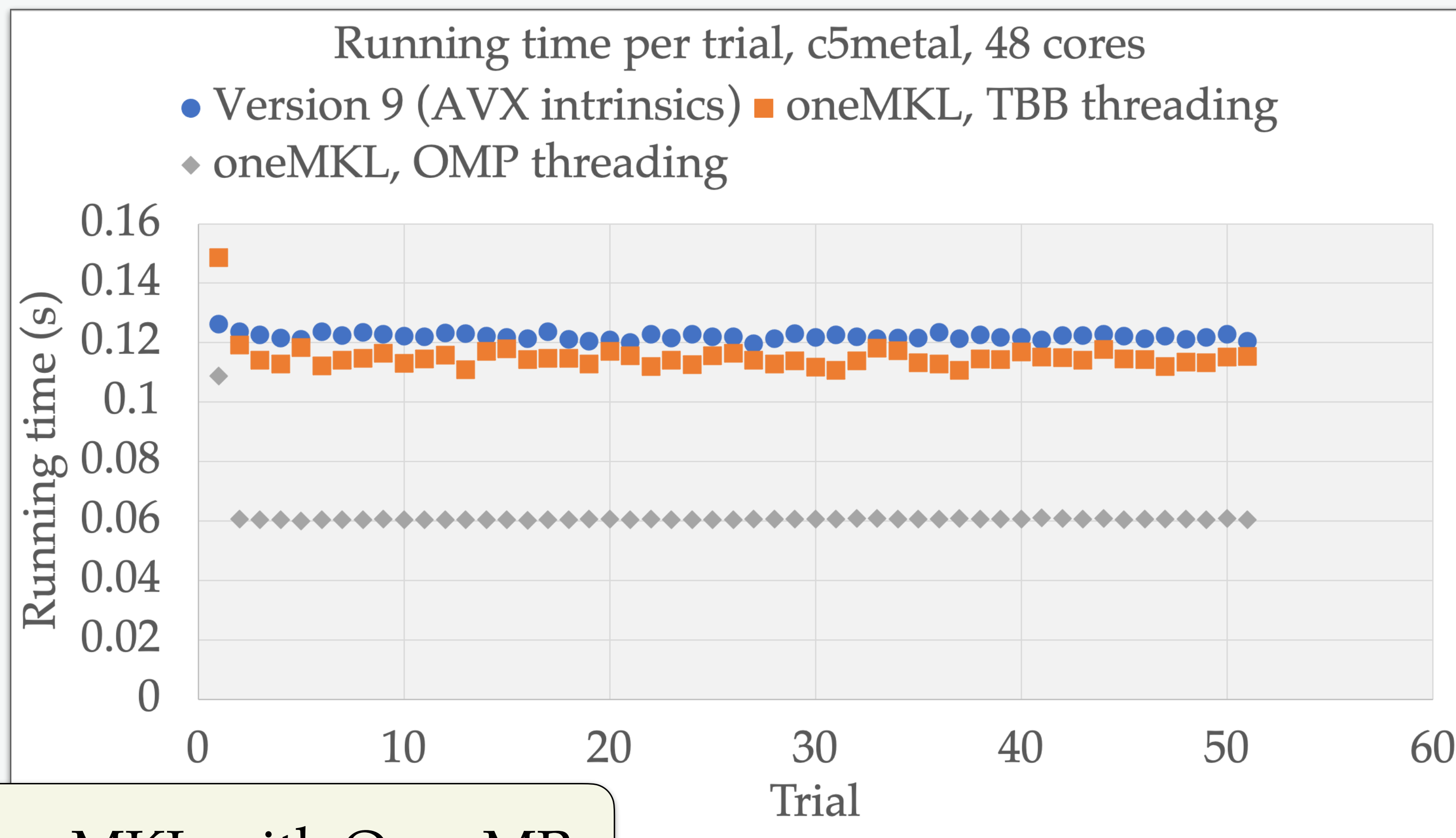
New performance results on a c5.metal instance

Implementation	Running time (s)	Relative speedup	Absolute speedup	GFLOPS	Fraction of peak
Parallel divide-and-conquer	1.091	1	1	125.998	2.734%
+ AVX2 compiler vectorization	0.878	1.2	1.242	156.511	3.397%
+ AVX512 compiler vectorization	0.824	1.1	1.324	166.817	3.620%
+ hand vectorization	0.052	15.9	21.087	2656.893	57.658%
oneMKL with OpenMP	0.061	0.9	18.002	2268.232	49.224%

Today, we will look at the algorithms and engineering behind these vectorized matrix-multiplication codes.

Outline

- Compiler vectorization

- Vectorization by hand

- Vectorization by hand, another approach

- Performance-engineering the hand-vectorized version

- Intel oneMKL

Outline

- **Compiler vectorization**

- Vectorization by hand

- Vectorization by hand, another approach

- Performance-engineering the hand-vectorized version

- Intel oneMKL

Recap: Parallel divide-and-conquer matrix multiplication (Version 7)

Matrix-multiply routine

```
void mmdac(double *restrict C, double *restrict A,          C
          double *restrict B, size_t size) {
 if (size == S) {
   mmbase(C, A, B);
 } else {
  size_t s00 = 0;
  size_t s01 = size/2;
  size_t s10 = (size/2)*n;
  size_t s11 = (size/2)*(n+1);
  cilk_scope {
   cilk_spawn mmdac(C+s00, A+s00, B+s00, size/2);
   cilk_spawn mmdac(C+s01, A+s00, B+s01, size/2);
   cilk_spawn mmdac(C+s10, A+s10, B+s00, size/2);
         mmdac(C+s11, A+s10, B+s01, size/2);
  }
  cilk_scope {
   cilk_spawn mmdac(C+s00, A+s01, B+s10, size/2);
   cilk_spawn mmdac(C+s01, A+s01, B+s11, size/2);
   cilk_spawn mmdac(C+s10, A+s11, B+s10, size/2);
         mmdac(C+s11, A+s11, B+s11, size/2);
```

mmbase() snippet

```
for (size_t i = 0; i < S; ++i)          C
  for (size_t k = 0; k < S; ++k)
    for (size_t j = 0; j < S; ++j)
```
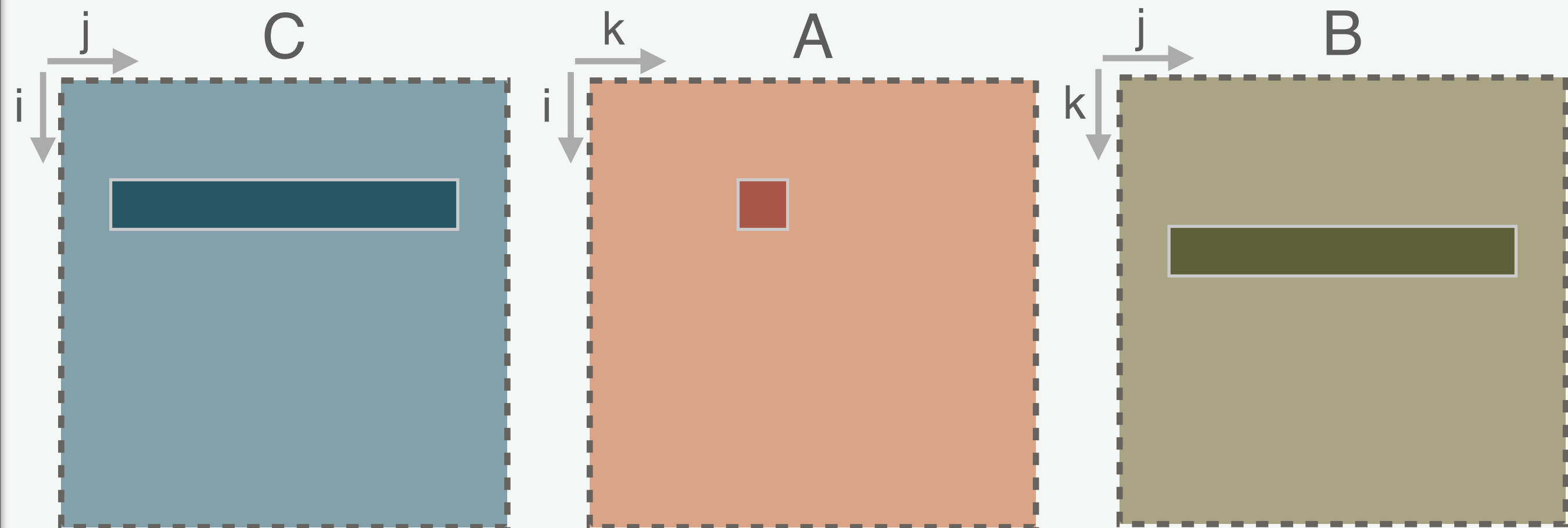

For an S×S submatrix of C, the base case repeatedly multiplies a row of B by a value in A and adds the result to a row of C.

AVX2 and AVX512 vectors

We will use AVX2 and AVX512 vector instructions to speed up this code.
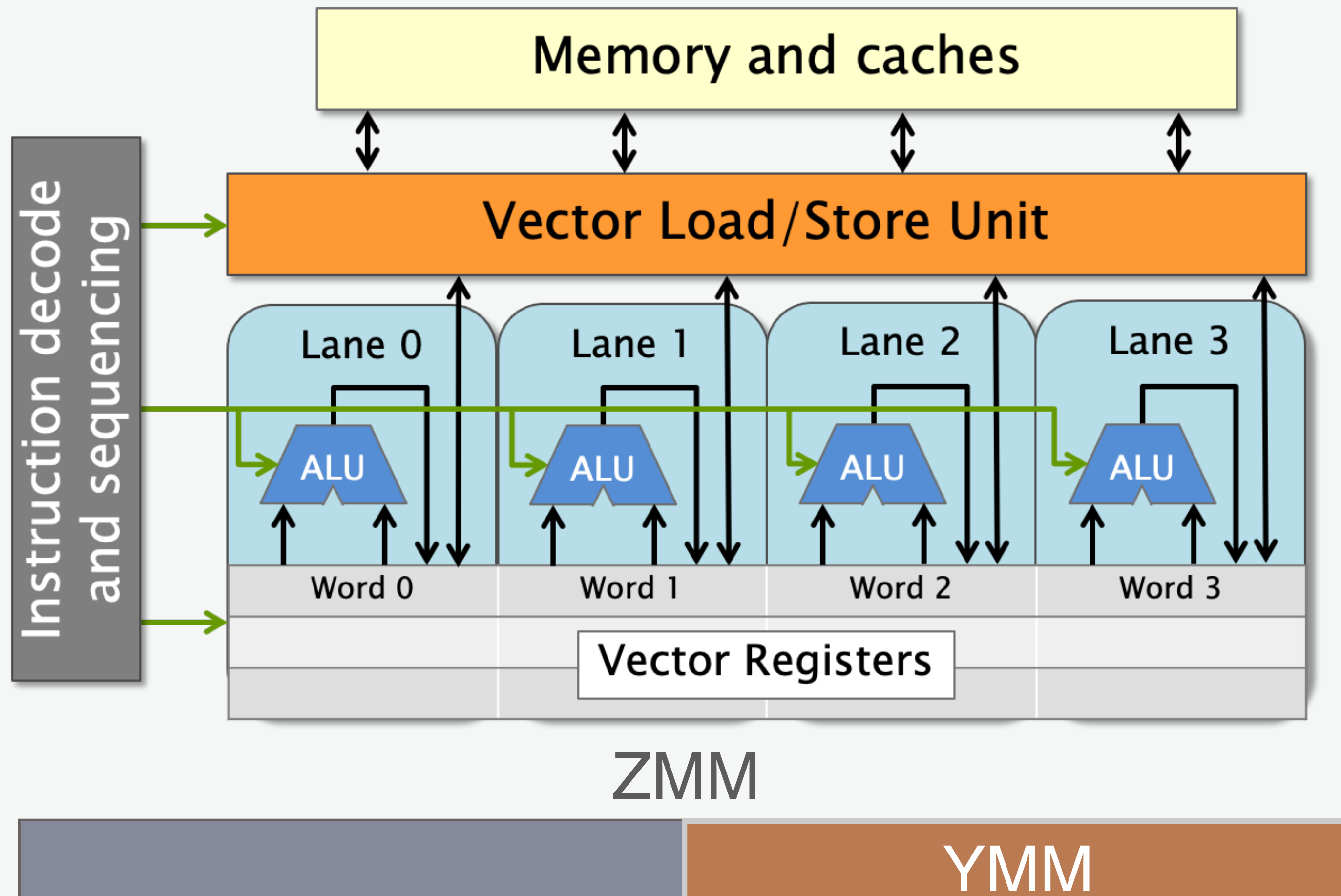
- AVX2 supports 256-bit ymm vector registers (4 doubles).

- AVX512 supports 512-bit zmm vector registers (8 doubles).

- The machine supports 32 ymm and zmm registers.

- All ymm and zmm registers are **aliased**.

AVX2 and AVX512 vector instructions

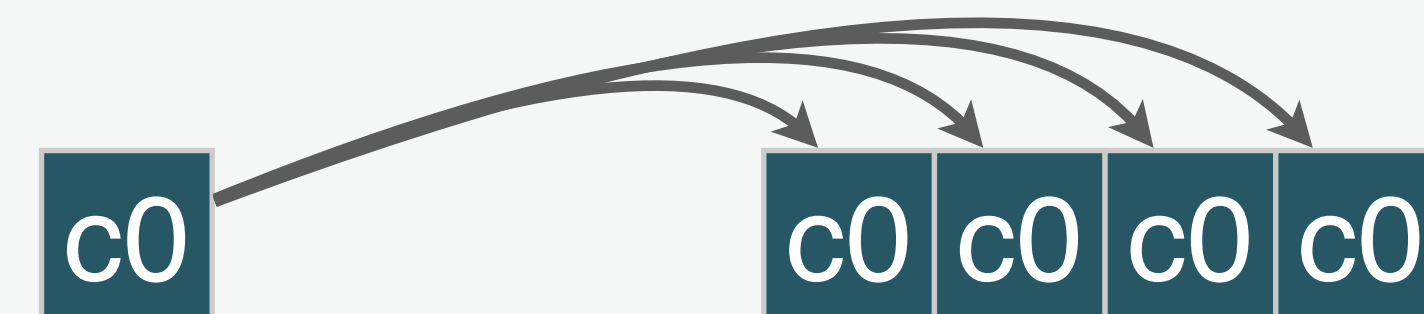Today, we can focus on just a subset of the available vector instructions.

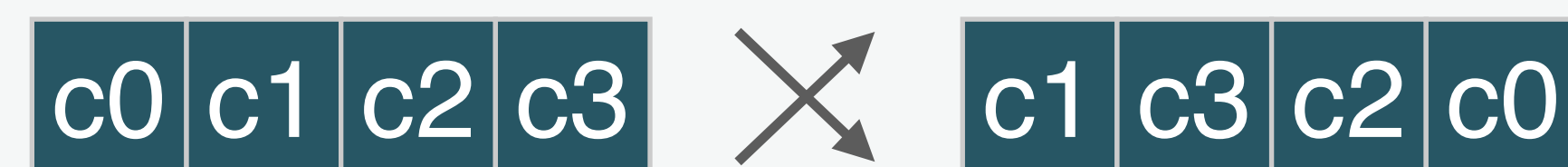- Vector load and store (aligned and unaligned).
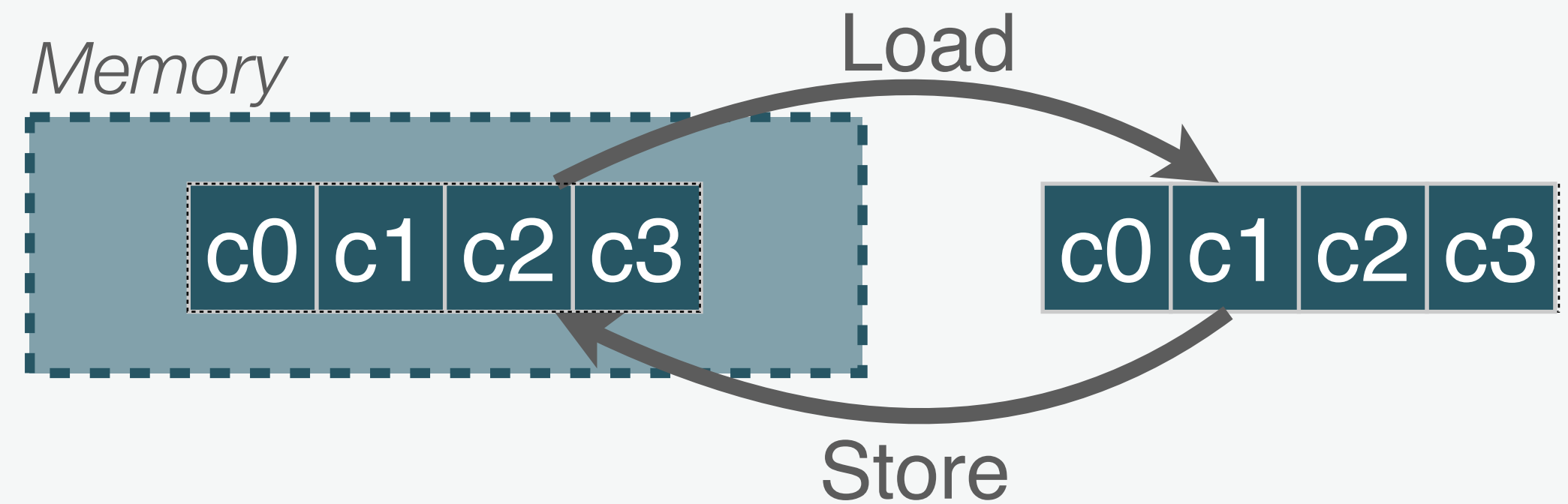
- Element-wise arithmetic, including **fused-multiply-add** (**FMA**).

- **Broadcast:** Fill all entries in a vector register with the same value.

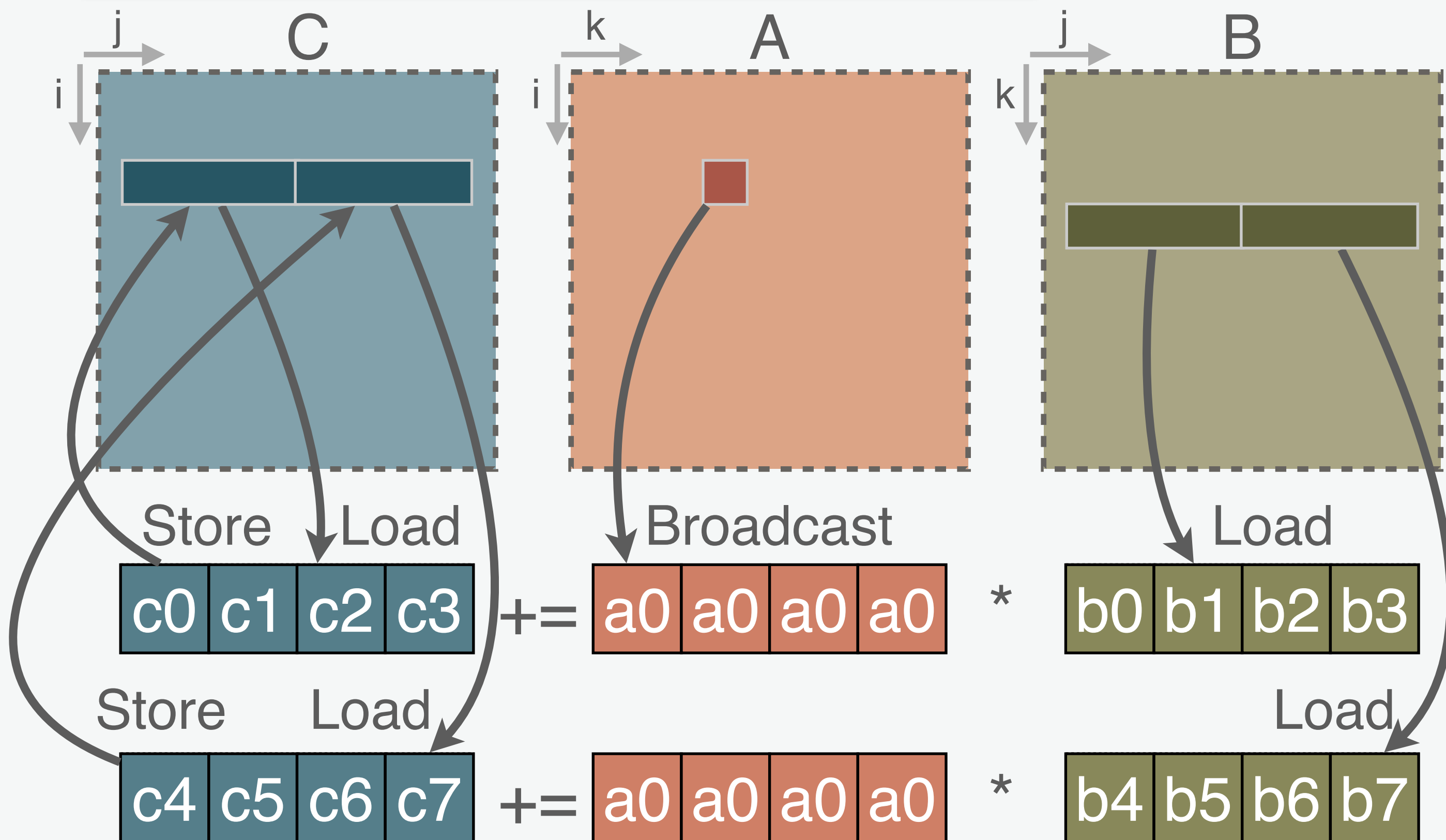- **Shuffle:** Permute the entries in a vector register. (More on this operation later.)

Memory

Load

| c0 | c1 | c2 | c3 |

| c0 | c1 | c2 | c3 |

Store

| c0 | c1 | c2 | c3 | += | a0 | a1 | a2 | a3 | * | b0 | b1 | b2 | b3 |

| c0 | | c0 | c0 | c0 | c0 |

| c0 | c1 | c2 | c3 | | c1 | c3 | c2 | c0 |

Disclaimer: I will typically illustrate operations using 4-element vectors, but 8-element vectors are used in practice.

Compiler vectorization

Matrix-multiply base case

```
for (size_t i = 0; i < S; ++i)
  for (size_t k = 0; k < S; ++k)
    for (size_t j = 0; j < S; ++j)
```

C

j   C

i

k   A

i

j   B

k

Store   Load   Broadcast   Load

| c0 | c1 | c2 | c3 | += | a0 | a0 | a0 | a0 | * | b0 | b1 | b2 | b3 |

Store   Load   Load

| c4 | c5 | c6 | c7 | += | a0 | a0 | a0 | a0 | * | b4 | b5 | b6 | b7 |

For each k:

1. $av = Broadcast(A[i,k])$

2. For $j \in [0, S)$ by vector width:

   a. $bv = Vector\text{-}load(B[k,j])$

   b. $cv = Vector\text{-}load(C[i,j])$

   c. $cv \mathrel{+}= av * bv$

   d. $Vector\text{-}store(cv, C[i,j])$

14

Compiler vectorization

> $ clang -o mm mm.c -fopencilk -O3 -march=native

Snippet of compiler-vectorized base case

```
                                                                    LLVM IR
%78 = load double, ptr %59, align 8
%79 = insertelement <4 x double> poison, double %78, i64 0
%80 = shufflevector <4 x double> %79,
   <4 x double> poison, <4 x i32> zeroinitializer
%88 = getelementptr inbounds double, ptr %2, i64 %60
%89 = load <4 x double>, ptr %88, align 8
%96 = load <4 x double>, ptr %16, align 8
%100 = tail call <4 x double> @llvm.fmuladd.v4f64(<4 x double> %80,
```

Load and broadcast a value from A

Vector-load from B

Vector-load from C

FMA

Vector-store into C

Compiler vectorization

```
$ clang -o mm mm.c -fopencilk -O3 -march=native
```

Snippet of compiler-vectorized base case

LLVM IR

```
%78 = load double, ptr %59, align 8
%79 = insertelement <4 x double> poison, double %78, i64 0
%80 = shufflevector <4 x double> %79,
   <4 x double> poison, <4 x i32> zeroinitializer
%88 = getelementptr inbounds double, ptr %2, i64 %60
%89 = load <4 x double>, ptr %88, align 8
%96 = load <4 x double>, ptr %16, align 8
%100 = tail call <4 x double> @llvm.fmuladd.v4f64(<4 x double> %80,
```

The compiler is using **32-bit** vector registers, but AVX512 offers **64-bit** vector registers!

Why isn't the compiler using AVX512?

CPU frequency scaling with AVX instructions

It is often hard to get performance out of AVX512 instructions, due to **downclocking**.

- Modern Intel CPUs reduce their clock frequency when they execute AVX512 instructions.

- This downclocking slows down non-AVX instructions running on the core as well.

- Modern compilers are reluctant to use AVX512, because it's often not worth it.

- This issue has improved on newer CPUs.
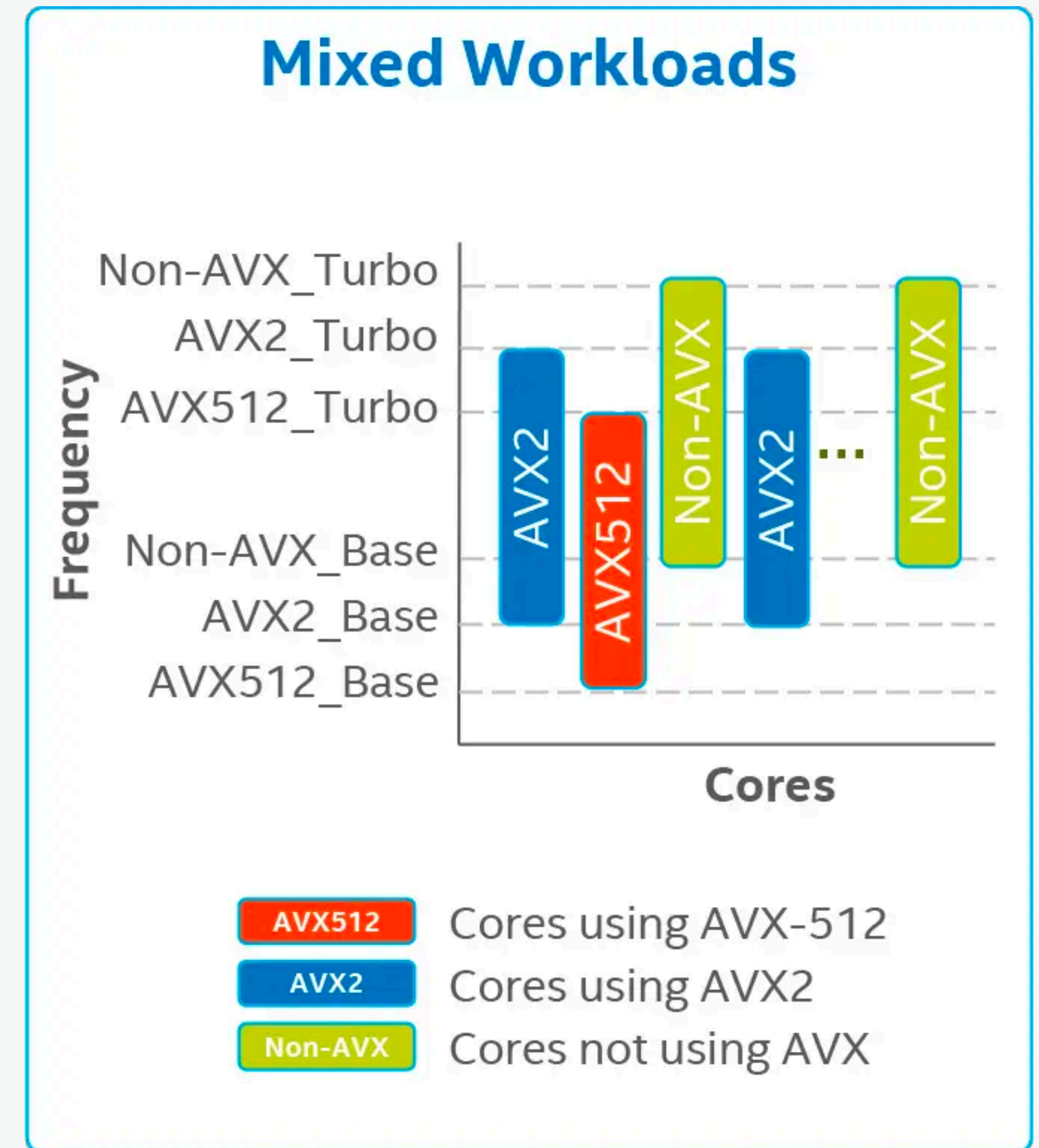
Compiler vectorization with AVX512

We can make the compiler to use AVX512 with different compiler flags. How much performance do we get?

Implementation	Running time (s)	Relative speedup	Absolute speedup	GFLOPS	Fraction of peak
Parallel divide-and-conquer	1.091	1	1	125.998	2.734%
+ AVX2 compiler vectorization	0.878	1.2	1.242	156.511	3.397%
+ AVX512 compiler vectorization	0.824	1.1	1.324	166.817	3.620%

We get a small boost from AVX512, but there's a still a lot of performance available.

Outline

- Compiler vectorization

- **Vectorization by hand**

- Vectorization by hand, another approach

- Performance-engineering the hand-vectorized version

- Intel oneMKL

How many loads and stores?
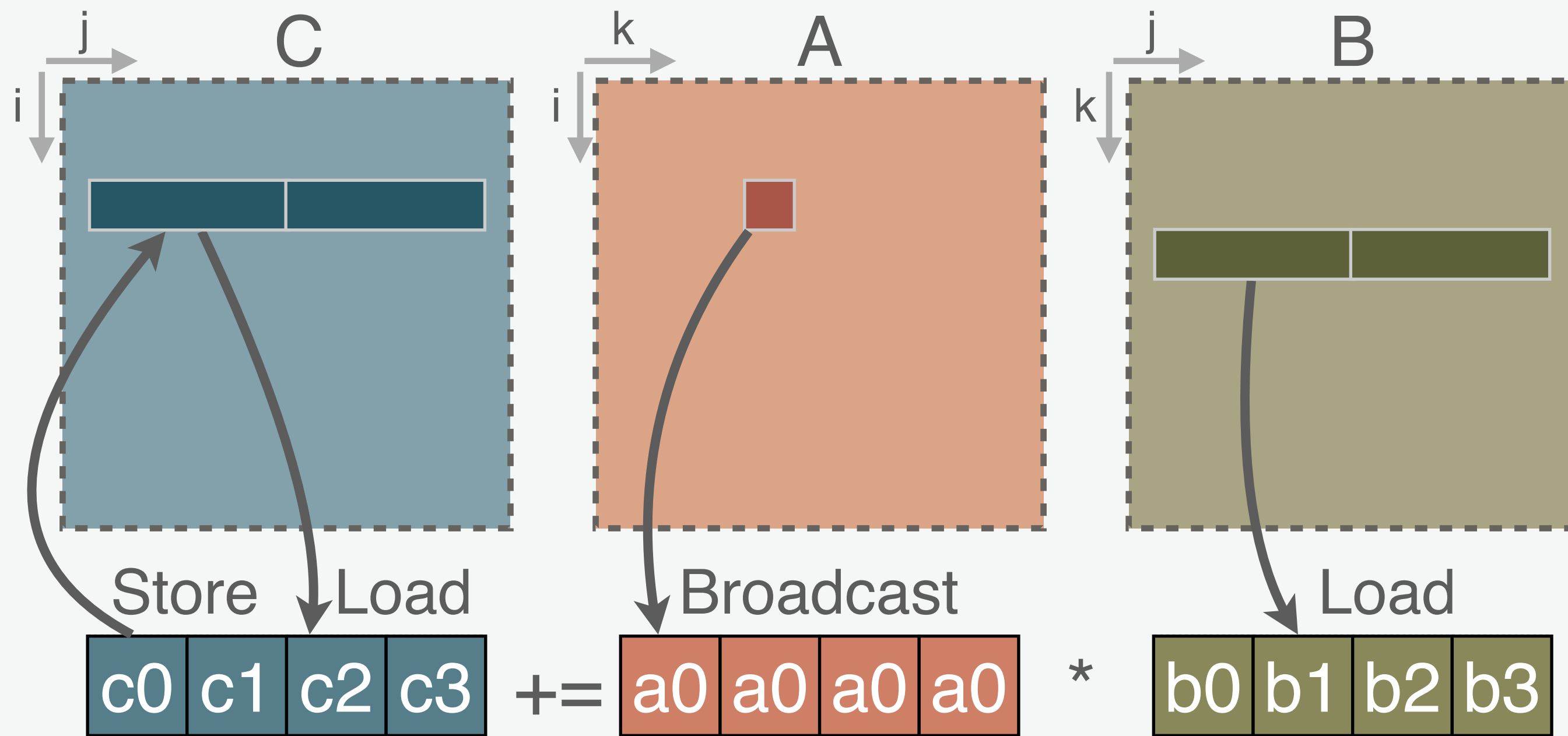
Matrix-multiply base case

```
for (size_t i = 0; i < S; ++i)
  for (size_t k = 0; k < S; ++k)
    for (size_t j = 0; j < S; ++j)
```

Store Load   Broadcast   Load

| c0 | c1 | c2 | c3 | += | a0 | a0 | a0 | a0 | * | b0 | b1 | b2 | b3 |

Suppose the base-case size S is 16 and uses 4-element vectors.

The inner loop performs 4 loads from C, 4 loads from B, and 4 stores into C.

Computing a row of 16 elements in C requires:
$16 * (1 + 4 + 4 + 4)$
$= 208$ loads and stores.

Alternative: Compute outer products

Alternatively, the base case can compute an **outer product** between a **subcolumn** of A and a **subrow** of B to update a submatrix in C.

	b_{k0}	b_{k1}	b_{k2}	b_{k3}
a_{0k}	$c_{00} \mathrel{+}= a_{0k}\, b_{k0}$	$c_{01} \mathrel{+}= a_{0k}\, b_{k1}$	$c_{02} \mathrel{+}= a_{0k}\, b_{k2}$	$c_{03} \mathrel{+}= a_{0k}\, b_{k3}$
a_{1k}	$c_{10} \mathrel{+}= a_{1k}\, b_{k0}$	$c_{11} \mathrel{+}= a_{1k}\, b_{k1}$	$c_{12} \mathrel{+}= a_{1k}\, b_{k2}$	$c_{13} \mathrel{+}= a_{1k}\, b_{k3}$
a_{2k}	$c_{20} \mathrel{+}= a_{2k}\, b_{k0}$	$c_{21} \mathrel{+}= a_{2k}\, b_{k1}$	$c_{22} \mathrel{+}= a_{2k}\, b_{k2}$	$c_{23} \mathrel{+}= a_{2k}\, b_{k3}$
a_{3k}	$c_{30} \mathrel{+}= a_{3k}\, b_{k0}$	$c_{31} \mathrel{+}= a_{3k}\, b_{k1}$	$c_{32} \mathrel{+}= a_{3k}\, b_{k2}$	$c_{33} \mathrel{+}= a_{3k}\, b_{k3}$
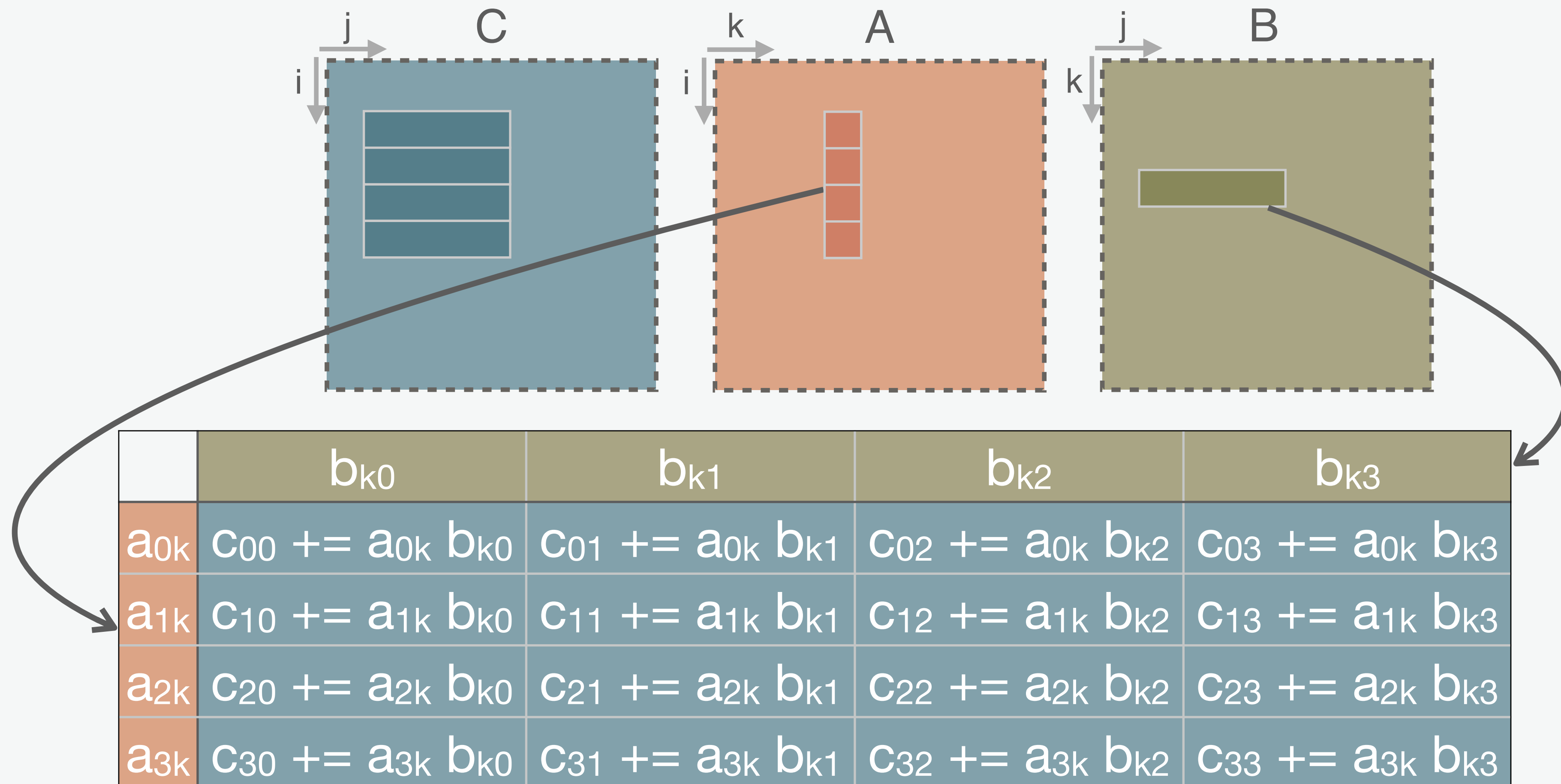
Vectorizing outer products
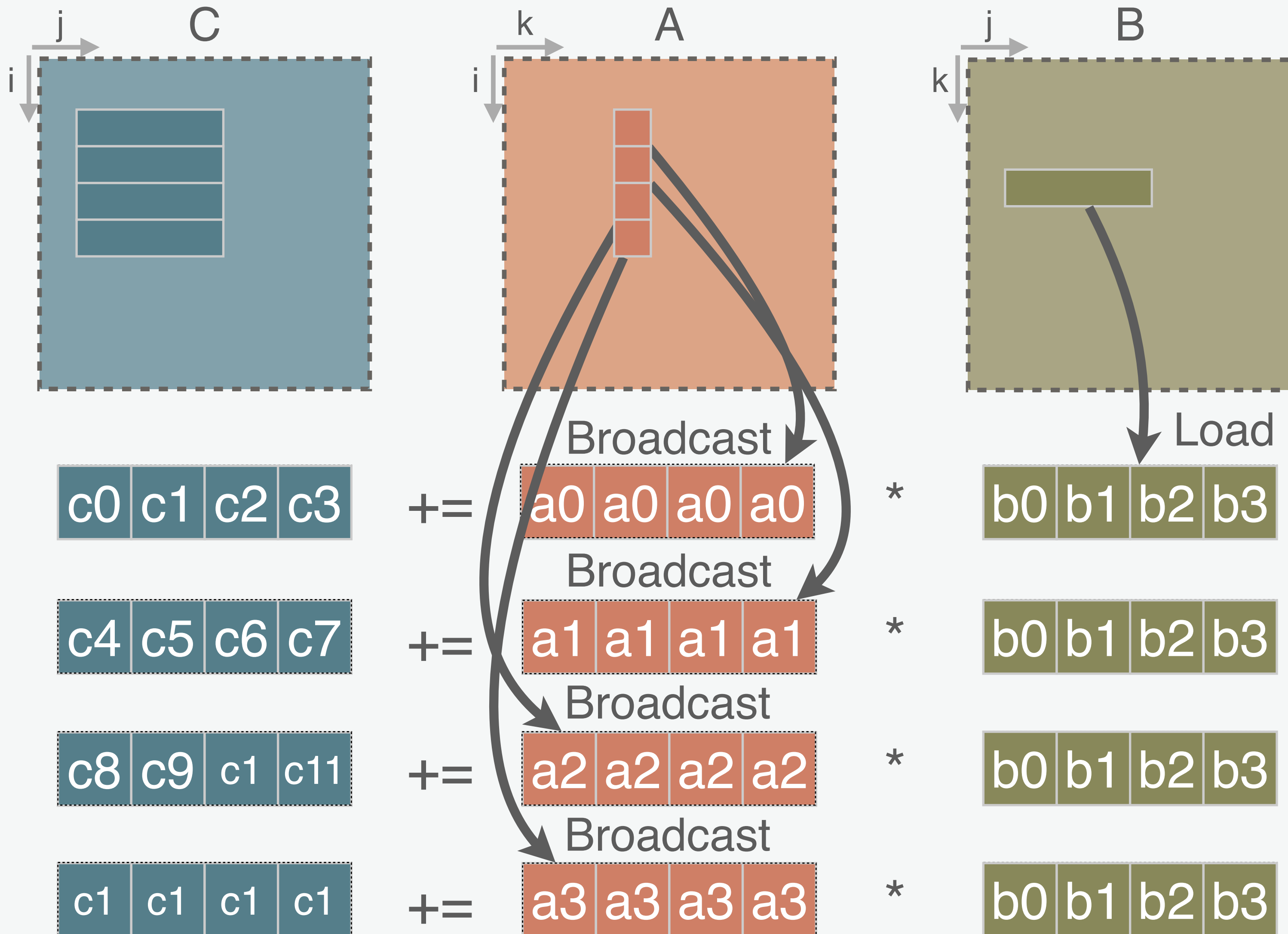
For each k:

1. bv = Vector-load(B[k,j])

2. For x in #C-vectors:

   a. av = Broadcast(A[i+x,k])

   b. cv[x] += av * bv

Analysis of outer-product base case

Suppose instead that the base case computes 16 elements in a 4×4 block in C using 4-element vectors.

If $S = 16$, then computing a elements 4×4 block of elements requires:
$4 + 4 + 16 * (1 + 4)$
$= 88$ loads and stores.
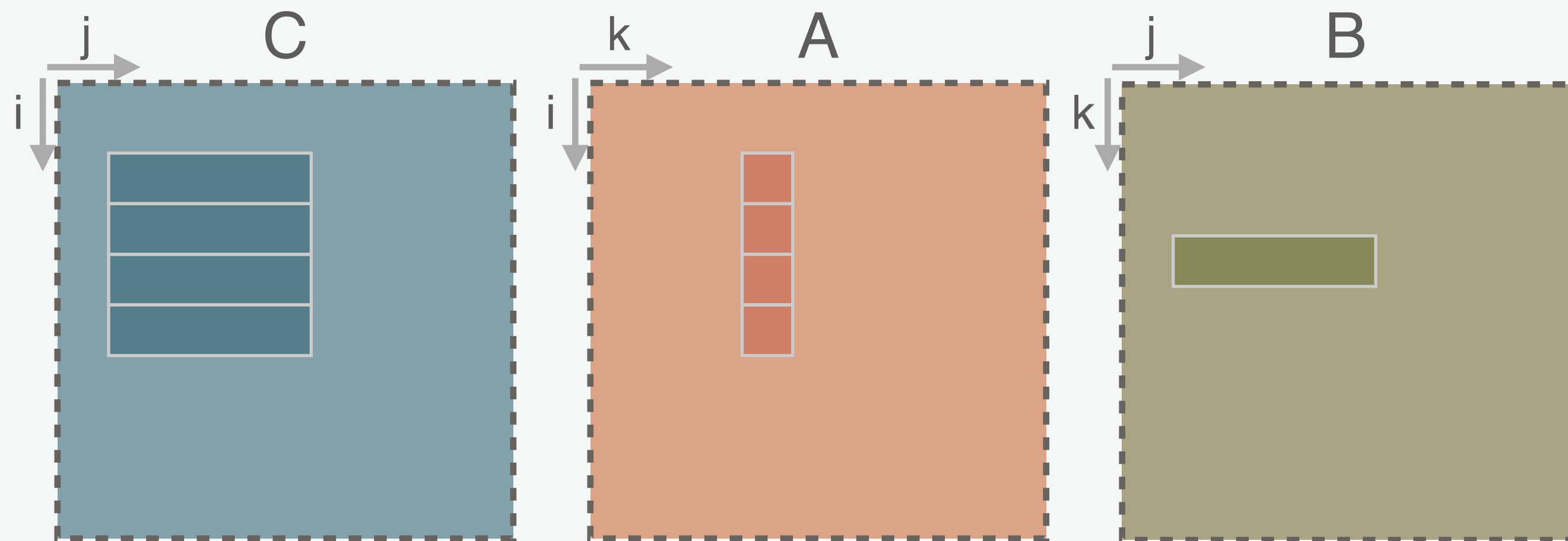
Computing a 4×4 block requires 4 loads from C, 4 stores into C, and, for each k, 4 loads from A and 1 load from B.

Less than half the loads and stores!

Advantages over compiler vectorization

This base case performs **fewer loads and stores** than the compiler-vectorized base case.
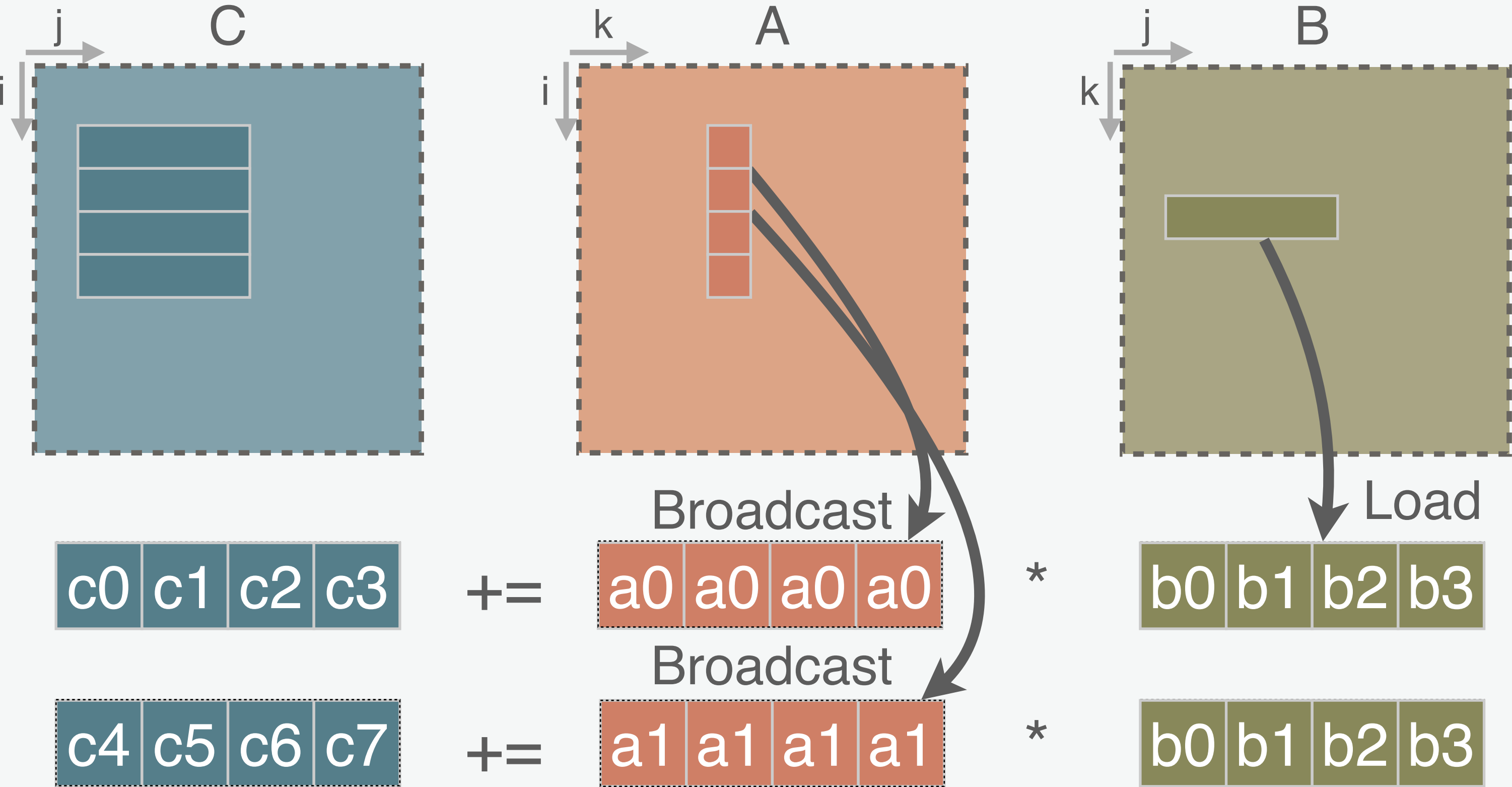
- Vector registers storing the C submatrix act like an **additional, faster cache**.

- Processing a C submatrix results in **better data locality**.

Implementation using the GCC vector extension

```c
// Vector type
typedef double vdouble __attribute__((vector_size(sizeof(double) * 8)));

// Zero-initialize the C-submatrix vectors.
vdouble cv[8];
for (int ivec = 0; ivec < 8; ++ivec)
  cv[ivec] = (vdouble){0.0};

// Loop over k.
for (int k = 0; k < BC; ++k) {
  // Load a vector from B.
  vdouble bv = *(const vdouble *)(&B[B_index(k, j, BC)]);
  for (int ivec = 0; ivec < 8; ++ivec)
    // Load a value from A, broadcast that value, and perform FMA.
    cv[ivec] += bv * A[A_index(i + ivec, k, BC)];
}

// Add the C-submatrix vectors to the C.
```

C Snippet of broadcast-based matrix-multiply base case (simplified)

Type definition for a vector of 8 doubles.

Vector-load from B.

Broadcast from A and perform FMA.

Store into C.

Budgeting the AVX512 vector registers

Each zmm register stores 8 doubles.

To compute an 8×8 C submatrix requires 10 registers:

- 8 registers to store the C submatrix;

- 1 register for a value from A; and

- 1 register for a vector from B.

There are 32 registers available, so compute a larger C submatrix!

- An 8×24 C submatrix requires 24 registers for C, 3 registers for B, and 1 register for A.
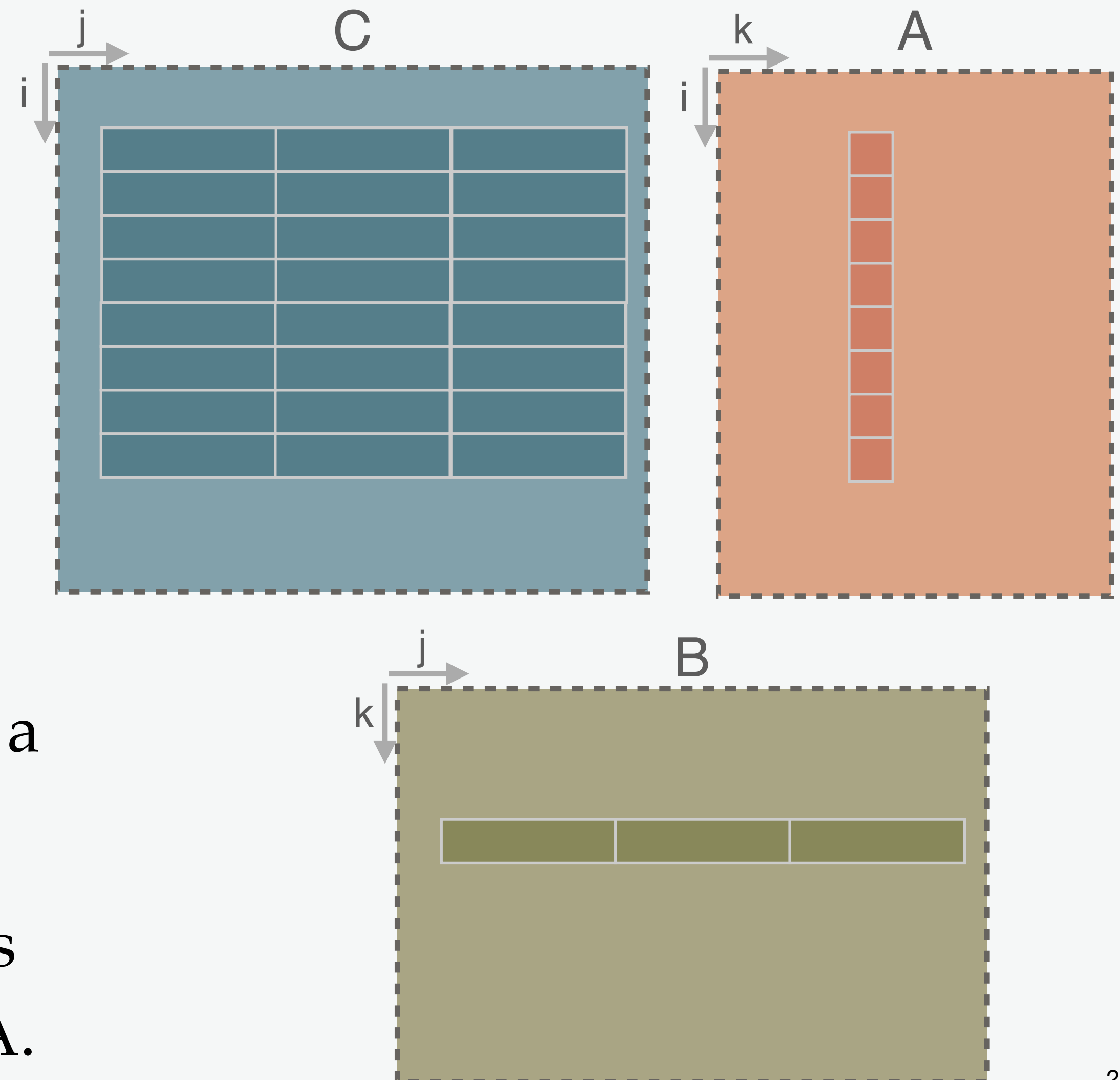
Instruction comparison in practice

We can use **performance counters** on the machine to compare the operations of these different implementations in practice.

Implementation	Instructions	L1 loads
AVX512 compiler vectorization	1.40E+10	1.00E+10
Broadcast outer product	1.68E+10	5.3E+09

Broadcast outer-product base case performs approximately half the L1 loads in practice!

Outline

- Compiler vectorization

- Vectorization by hand

- **Vectorization by hand, another approach**

- Performance-engineering the hand-vectorized version

- Intel oneMKL

Can we do even better?

j C   k A   j B

i   i   k

Broadcast   Load

| c0 | c1 | c2 | c3 | += | a0 | a0 | a0 | a0 | * | b0 | b1 | b2 | b3 |

Broadcast

| c4 | c5 | c6 | c7 | += | a1 | a1 | a1 | a1 | * | b0 | b1 | b2 | b3 |

Broadcast

| c8 | c9 | c1 | c11 | += | a2 | a2 | a2 | a2 | * | b0 | b1 | b2 | b3 |

Broadcast

| c1 | c1 | c1 | c1 | += | a3 | a3 | a3 | a3 | * | b0 | b1 | b2 | b3 |
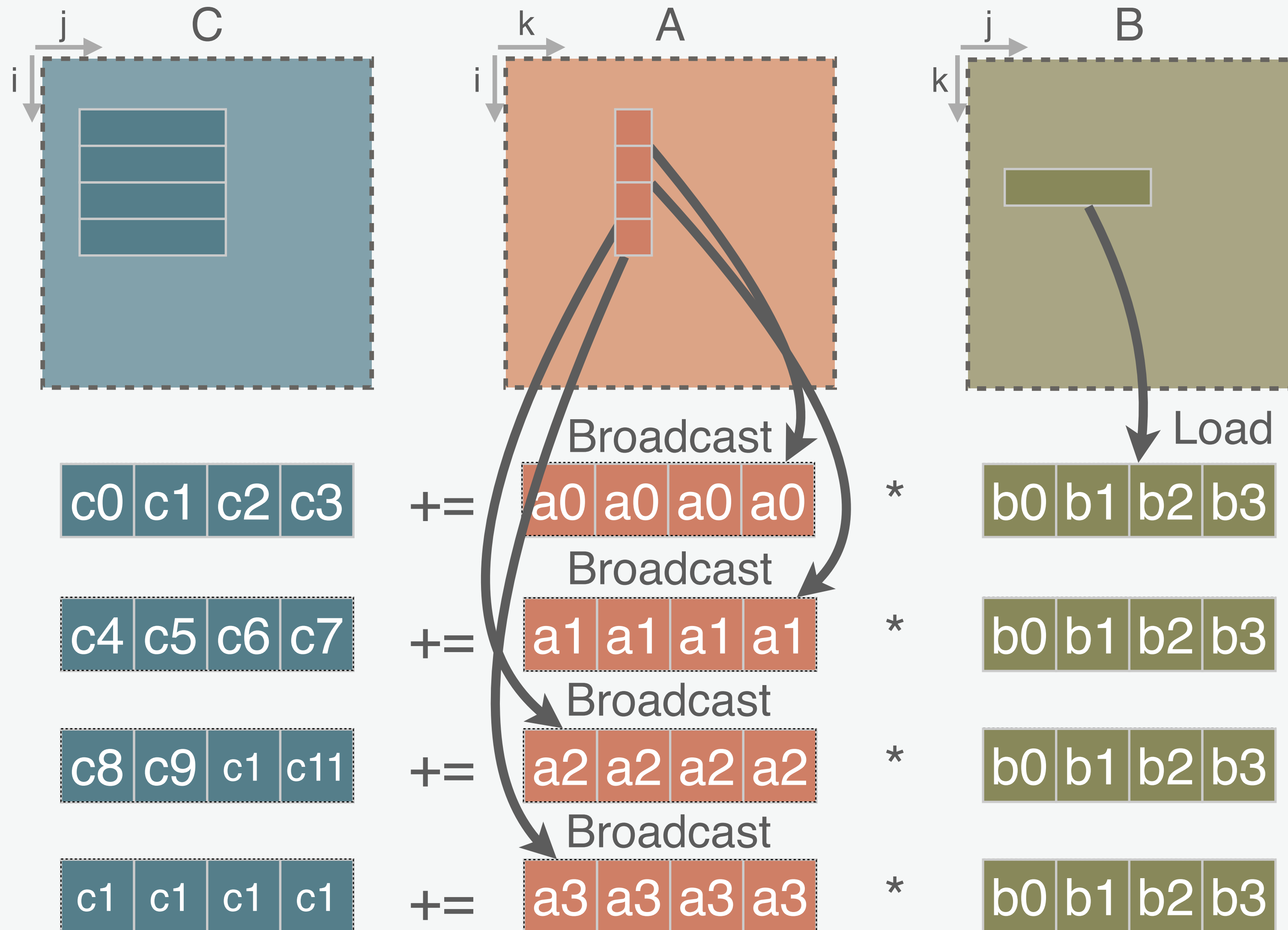
Currently, the code performs many scalar loads from A for each vector-load from B.

Scalar and vector loads have the **same cost**, but vector loads handle **more data**.

Can we vectorize the loads from A?

Using a vector from A

Supposing we can efficiently vector-load a subcolumn of A, there are other concerns.

The element-wise vector product produces a **diagonal** of the C submatrix!
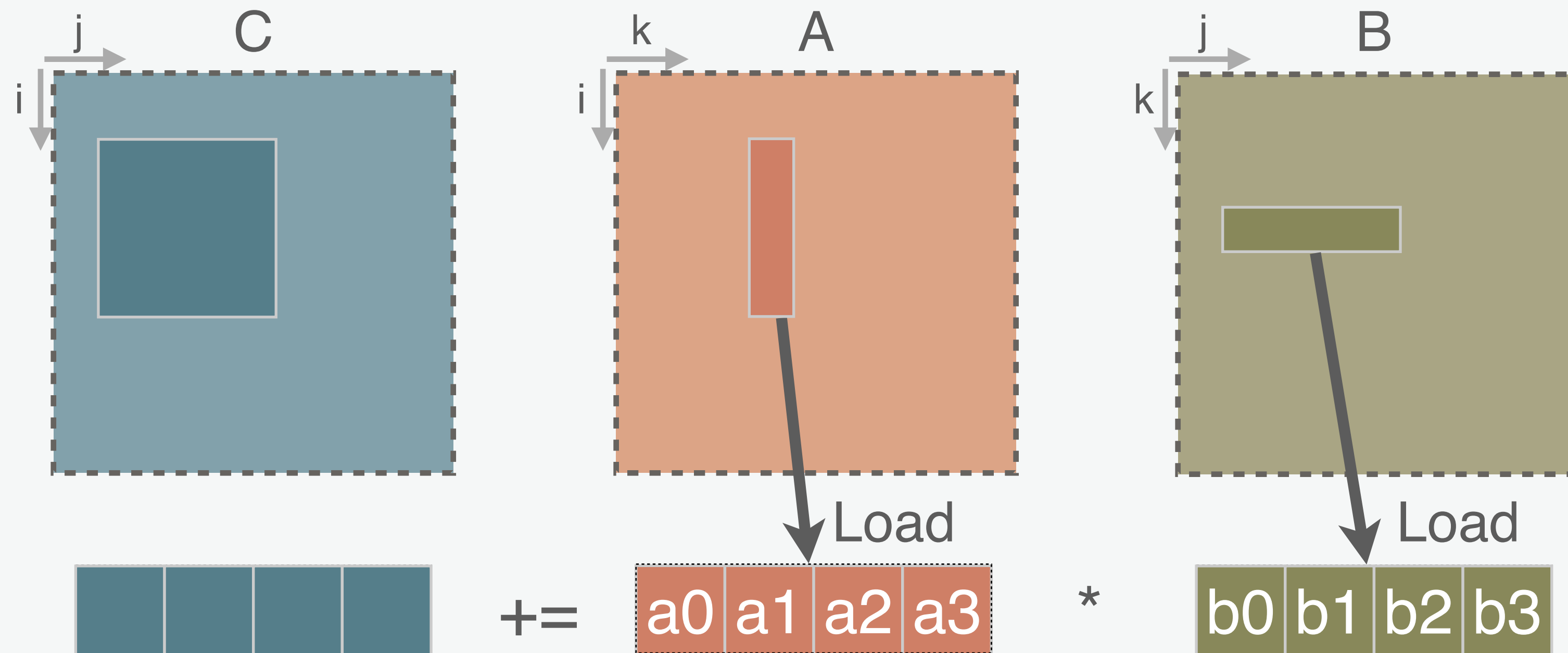
How do we get the other values in the C submatrix?

Answer: Vector shuffle instructions

Vector shuffle instructions

AVX2 and AVX512 offer **vector-shuffle** instructions for shuffling elements in arbitrary ways.

- Some shuffles operate on 1 vector register, while others operate on 2.

- Some shuffles have specialized instructions, while others use a register to describe the shuffle.

- Some shuffles are more expensive than others.

What shuffles do we need?

Outer products with shuffles

C

j
i

A

k
i

B

j
k

α += | a0 | a1 | a2 | a3 | * | b0 | b1 | b2 | b3 |

Load

Shuffle

β += | a1 | a0 | a3 | a2 | * | b0 | b1 | b2 | b3 |

γ += | a0 | a1 | a2 | a3 | * | b2 | b3 | b0 | b1 |

Shuffle

δ += | a1 | a0 | a3 | a2 | * | b2 | b3 | b0 | b1 |
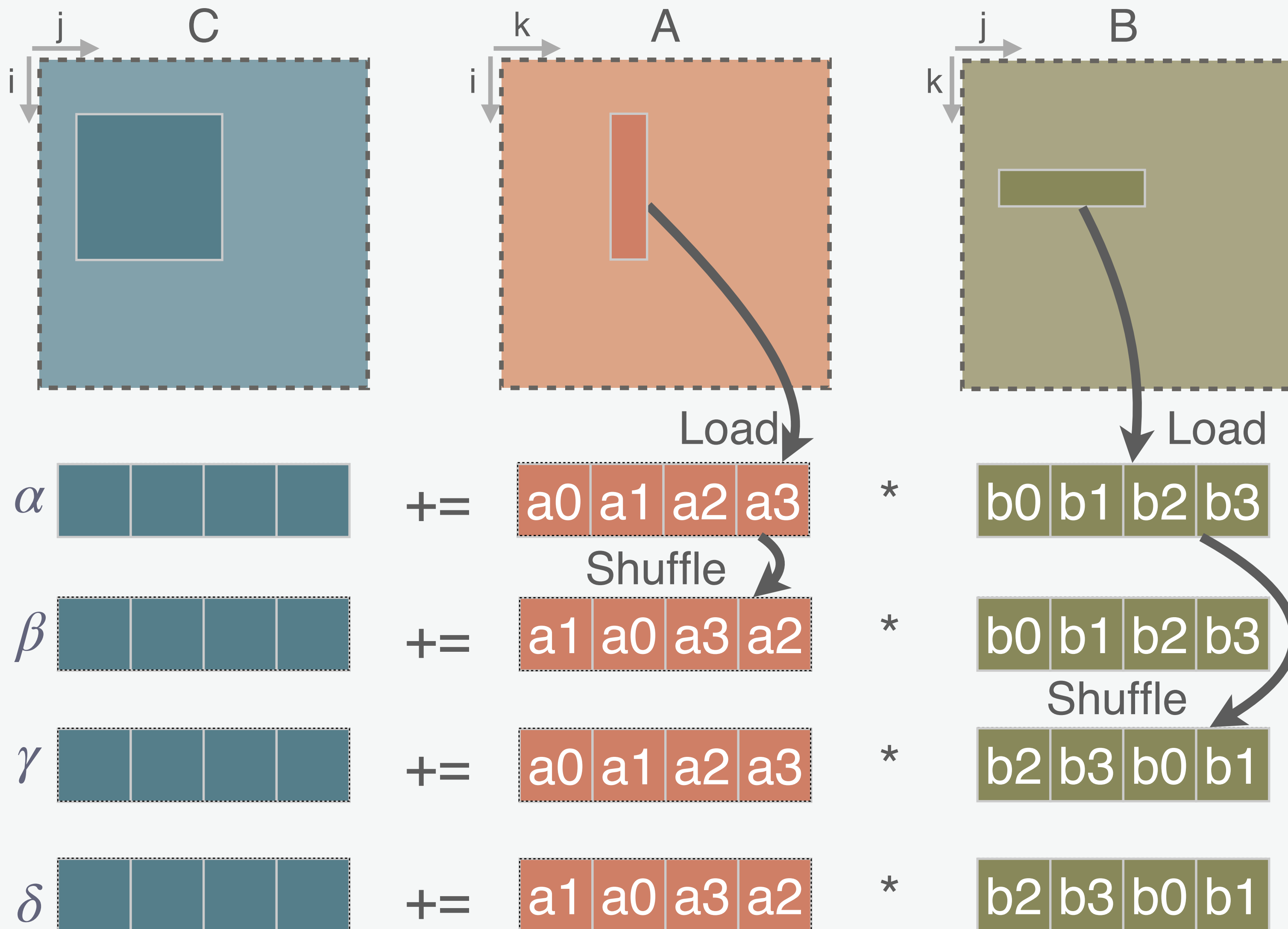
For each k:

1. bv = Vector-load(B[k,j])

2. av = Vector-load(A[i,k])

3. cv[0] += av * bv

4. av' = Shuffle(av, 1,0,3,2)

5. cv[1] += av' * bv

6. bv' = Shuffle(bv, 2,3,0,1)

7. cv[2] += av * bv'

8. cv[3] += av' * bv'

Comparison against broadcast base case

Broadcast base case accesses to A.

k A
i

Broadcast
| a0 | a0 | a0 | a0 |

Broadcast
| a1 | a1 | a1 | a1 |

Broadcast
| a2 | a2 | a2 | a2 |

Broadcast
| a3 | a3 | a3 | a3 |

Shuffle base case accesses to A.

k A
i

Load
| a0 | a1 | a2 | a3 |

Shuffle
| a1 | a0 | a3 | a2 |

| a0 | a1 | a2 | a3 |

| a1 | a0 | a3 | a2 |
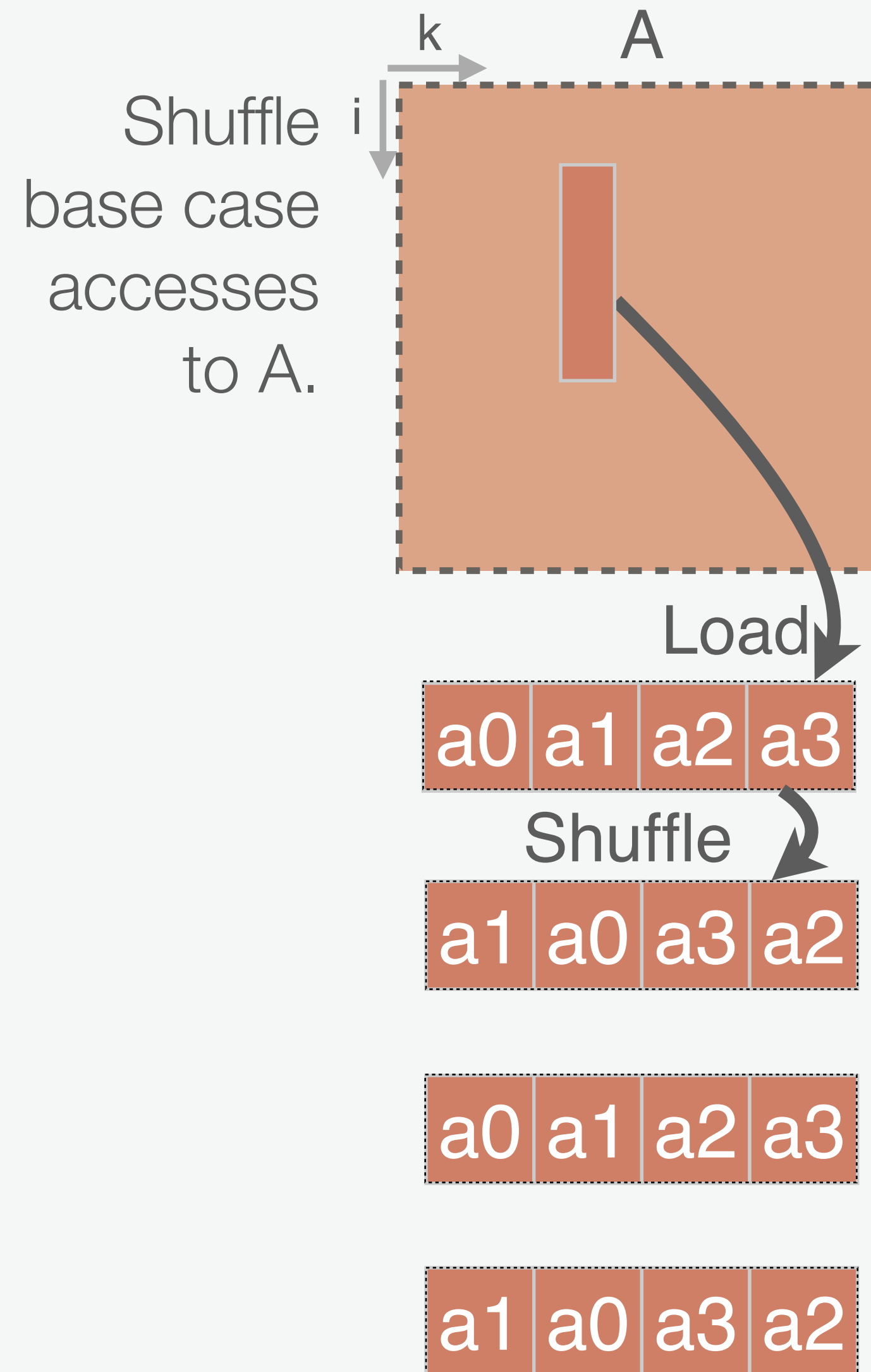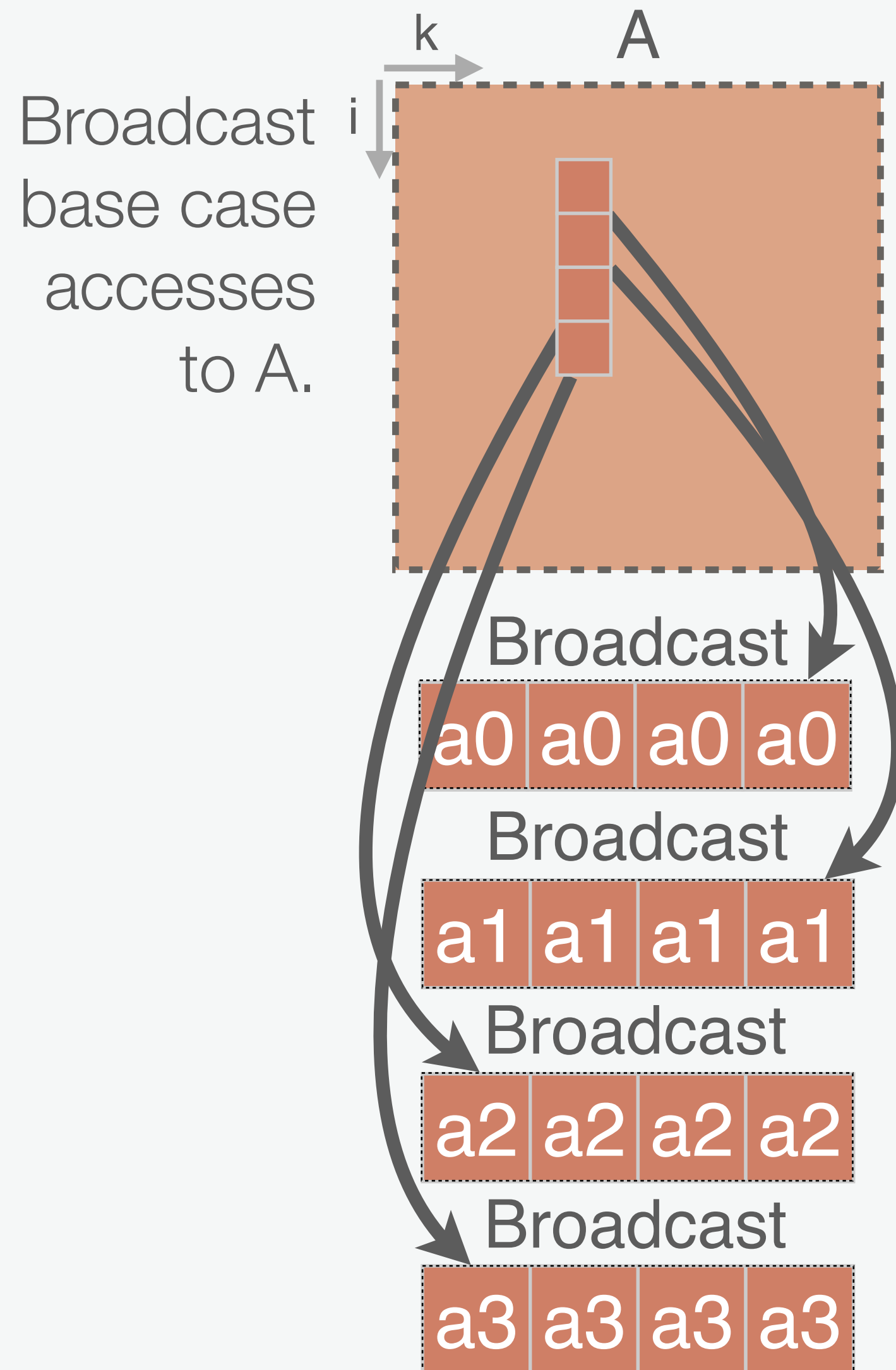
- The shuffle base case performs **fewer loads** than the broadcast base case.

- Shuffles operate entirely on **registers**.

- More shuffles are needed to write the results to C, but that happens **rarely**.

Generalizing to 8-element vectors

We can generalize this idea to use 4 permutations of 8-element vectors to compute an 8x8 C submatrix, such as by using the following 4 permutations:
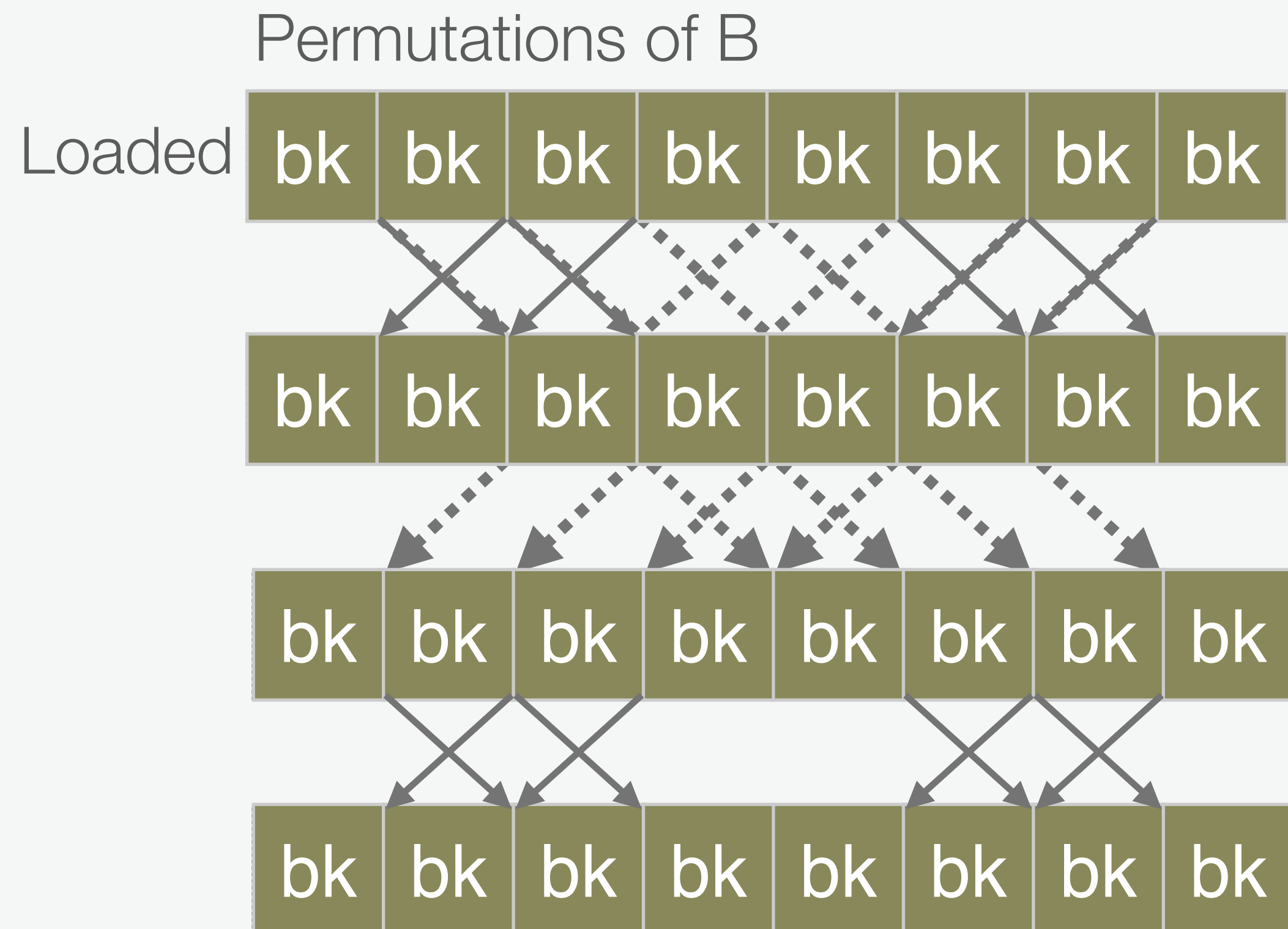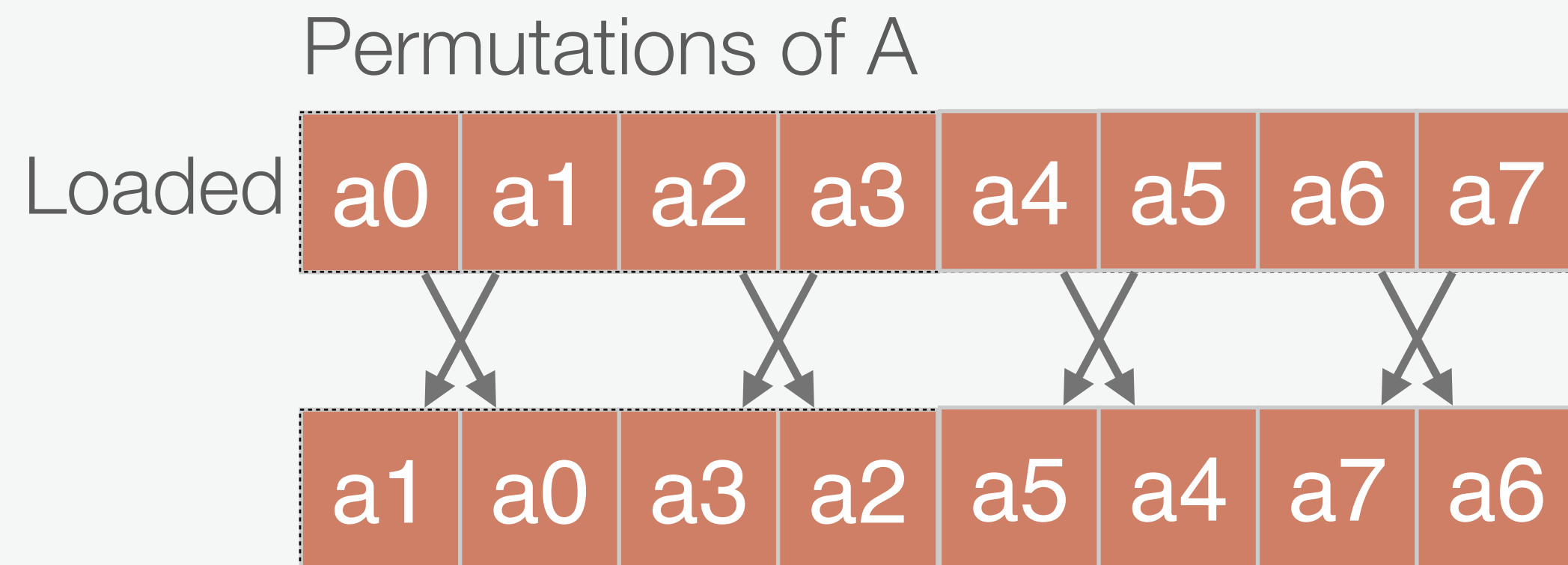
Implementating the shuffle base case using the GCC vector extension

```
// Vector type
typedef double vdouble __attribute__((vector_size(sizeof(double) * 8)));
// Zero-initialize the C-submatrix vectors…
// Loop over k.
for (int k = 0; k < BC; ++k) {
  // Load vectors from A and B.
  vdouble bv = *(const vdouble *)(&B[B_index(k, j, BC)]);
  vdouble av = *(const vdouble *)(&A[A_index(i, k, BC)]);
  // av_p = A1 A0 A3 A2 A5 A4 A7 A6
  vF a_p = __builtin_shufflevector(av, av, 1, 0, 3, 2, 5, 4, 7, 6);
  // bv_p0 = B2 B3 B0 B1 B6 B7 B4 B5
  vF bv_p0 = __builtin_shufflevector(bv, bv, 2, 3, 0, 1, 6, 7, 4, 5);
  cv[0] += av * bv;
  cv[1] += av_p * bv;
  // bv_p1 = B4 B5 B6 B7 B0 B1 B2 B3
  vF bv_p1 = __builtin_shufflevector(bv, bv, 4, 5, 6, 7, 0, 1, 2, 3);
  cv[2] += av * bv_p0;
  cv[3] += av_p * bv_p0;
  // bv_p2 = B6 B7 B4 B5 B2 B3 B0 B1
  vF bv_p2 = __builtin_shufflevector(bv_p0, bv_p0, 4, 5, 6, 7, 0, 1, 2, 3);
  cv[4] += av * bv_p1;
  cv[5] += av_p * bv_p1;
```

C Snippet of shuffle-based matrix-multiply base case (simplified)

Type definition for a vector of 8 doubles.

Vector-loads from A and B.

Shuffle A.

Shuffle B.

Shuffle B again.

Shuffle B yet again.

36

Budgeting the AVX512 vector registers for the shuffle base case

Each of the 32 zmm registers stores 8 doubles.

To compute an 8×8 C submatrix requires 14 registers:

- 8 registers to store the C submatrix;

- 2 registers for A and its permutation; and

- 4 registers for B and its permutations.

Can we compute an 8×24 C submatrix?

- An 8×24 C submatrix seems to require 34 registers: 24 for C, 6 for A and its permutations, and 4 for B and its permutations.

- If we order the operations carefully, we can reuse registers for different permutations and require just 2 registers for B.

Comparison in practice

How do the broadcast and shuffle outer-product base cases compare in practice?

Outer-product base case	Instructions	L1 loads	Running time (s)
Broadcast	1.68E+10	5.30E+09	0.052
Shuffle	1.64E+10	2.53E+09	0.063

The shuffle base case performs **fewer** instructions and **half** the L1 loads, but runs **slower**!

Why wasn't it faster?

Problem: We're running out of **functional units** on the core to perform arithmetic and shuffles.

Recall complex pipelining:
A processor core in a modern multicore chip has **many** functional units on different paths through the execution pipeline.
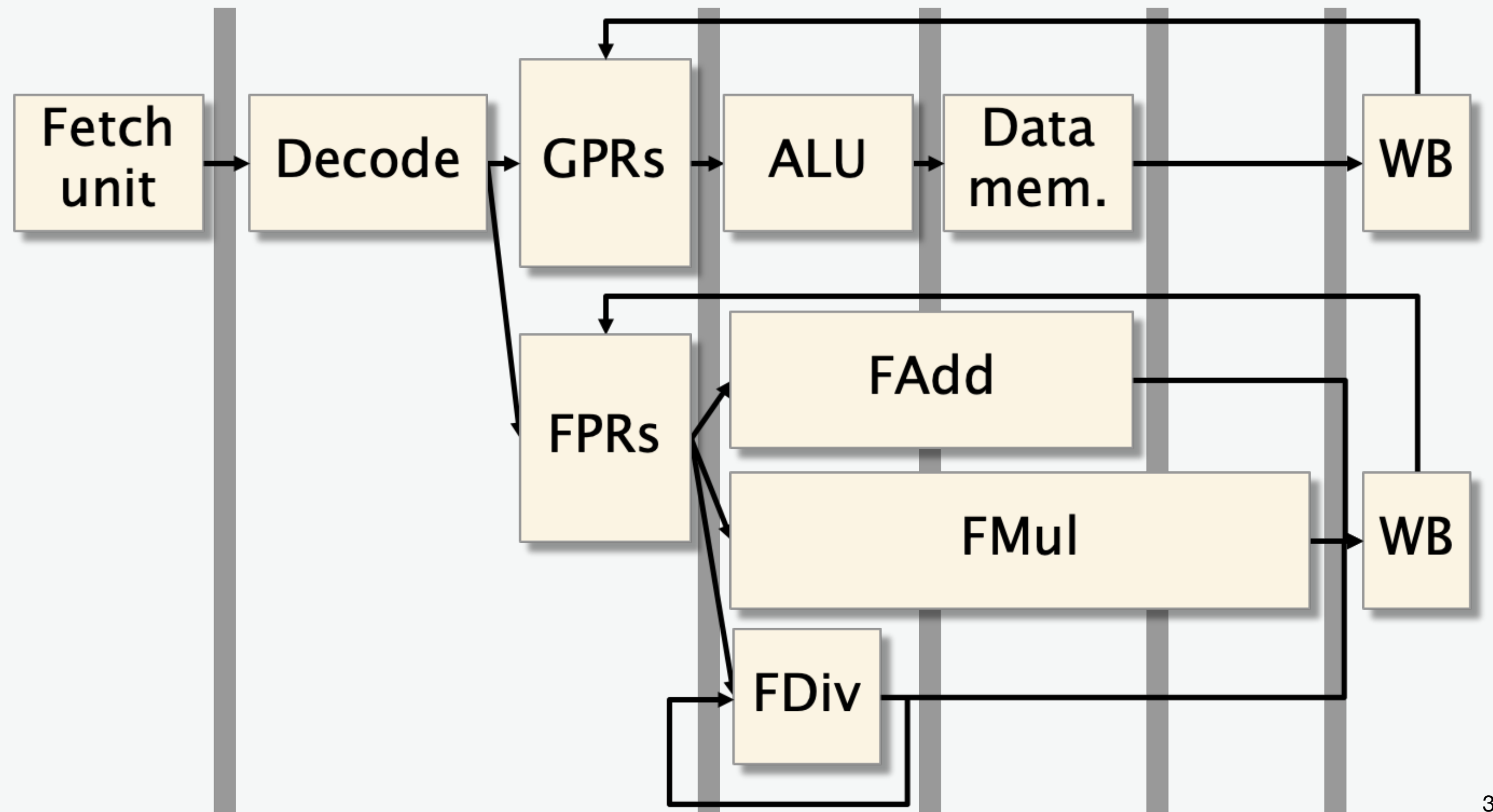
Not enough ports

Cascade Lake execution engine schematic

Technology Guide

intel.

Intel® Advanced Vector Extensions 512 (Intel® AVX-512) - Permuting Data Within and Between AVX Registers
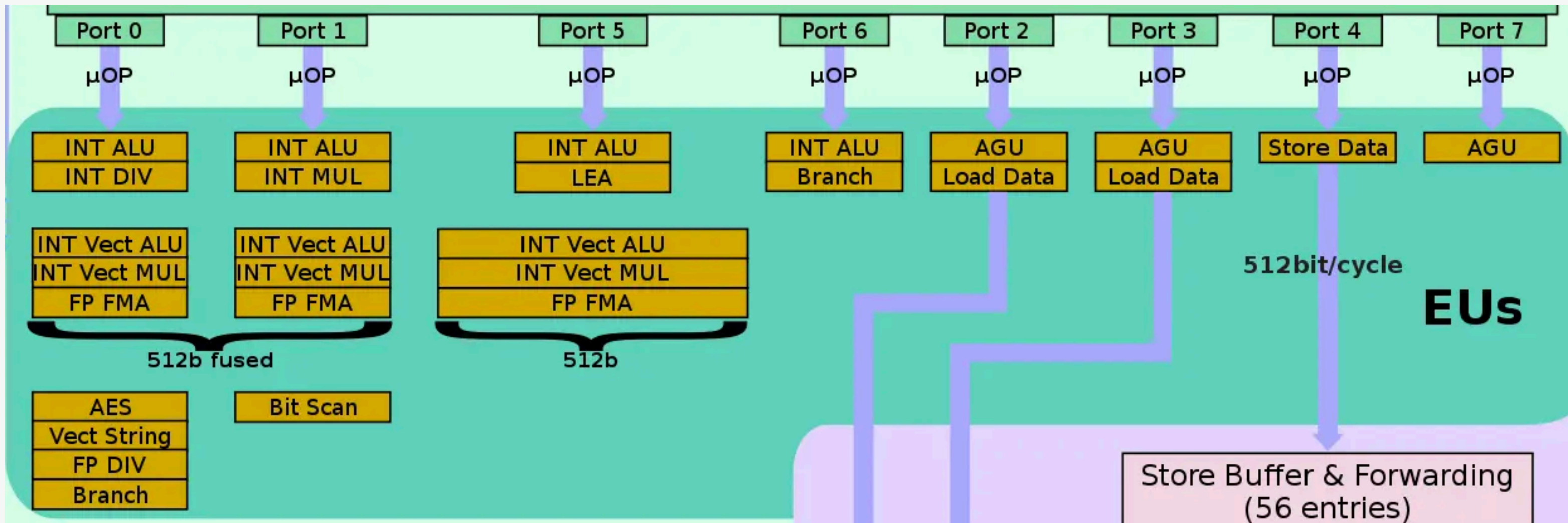
On Intel CPUs, different **ports** support different functional units.

- Ports 0/1 and 5 handle FMAs.

- Ports 2 and 3 load from memory.

- Port 5 handles **all shuffle instructions**.

Source: https://en.wikichip.org/wiki/intel/microarchitectures/cascade_lake

Hope remains!

Technology Guide

intel.

Intel® Advanced Vector Extensions 512 (Intel® AVX-512) - Permuting Data Within and Between AVX Registers

Author

Daniel Towner

1   Introduction

The Intel® Advanced Vector Extensions (Intel® AVX) family of instruction sets on Intel processors provides a rich variety of capabilities for supporting many different single instruction, multiple data (SIMD) instructions and data types. Like many other SIMD instruction sets, Intel AVX instructions are predominantly vertical, or map, instructions, where one or more SIMD values are converted to another SIMD value element-by-element. However, it can be desirable to reorder elements within or across SIMD values as well, and the Intel AVX families of instructions have many clever ways of achieving this. This document discusses the many different ways to perform permutations, describes the trade-offs, and shows how the techniques described can be used to implement versions of a few selected algorithms.

This document is part of the Network Transformation Experience Kits.

"Note that on 3rd Gen…and 4th Gen Intel Xeon Scalable processors…, port 1 also can execute some shuffle instructions…"

The shuffle outer-product base case might be faster on the next generation of CPU hardware.

Outline

- Compiler vectorization

- Vectorization by hand

- Vectorization by hand, another approach

- **Performance-engineering the hand-vectorized version**
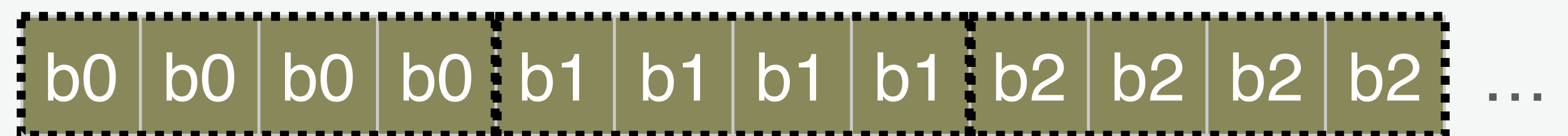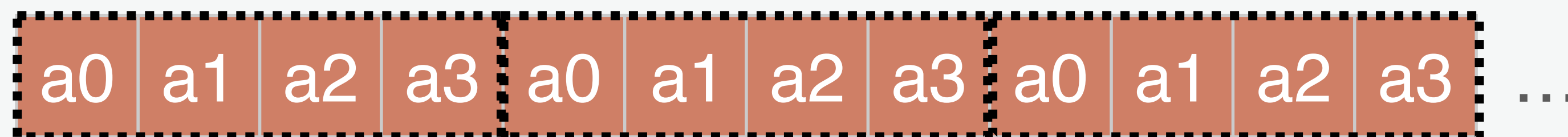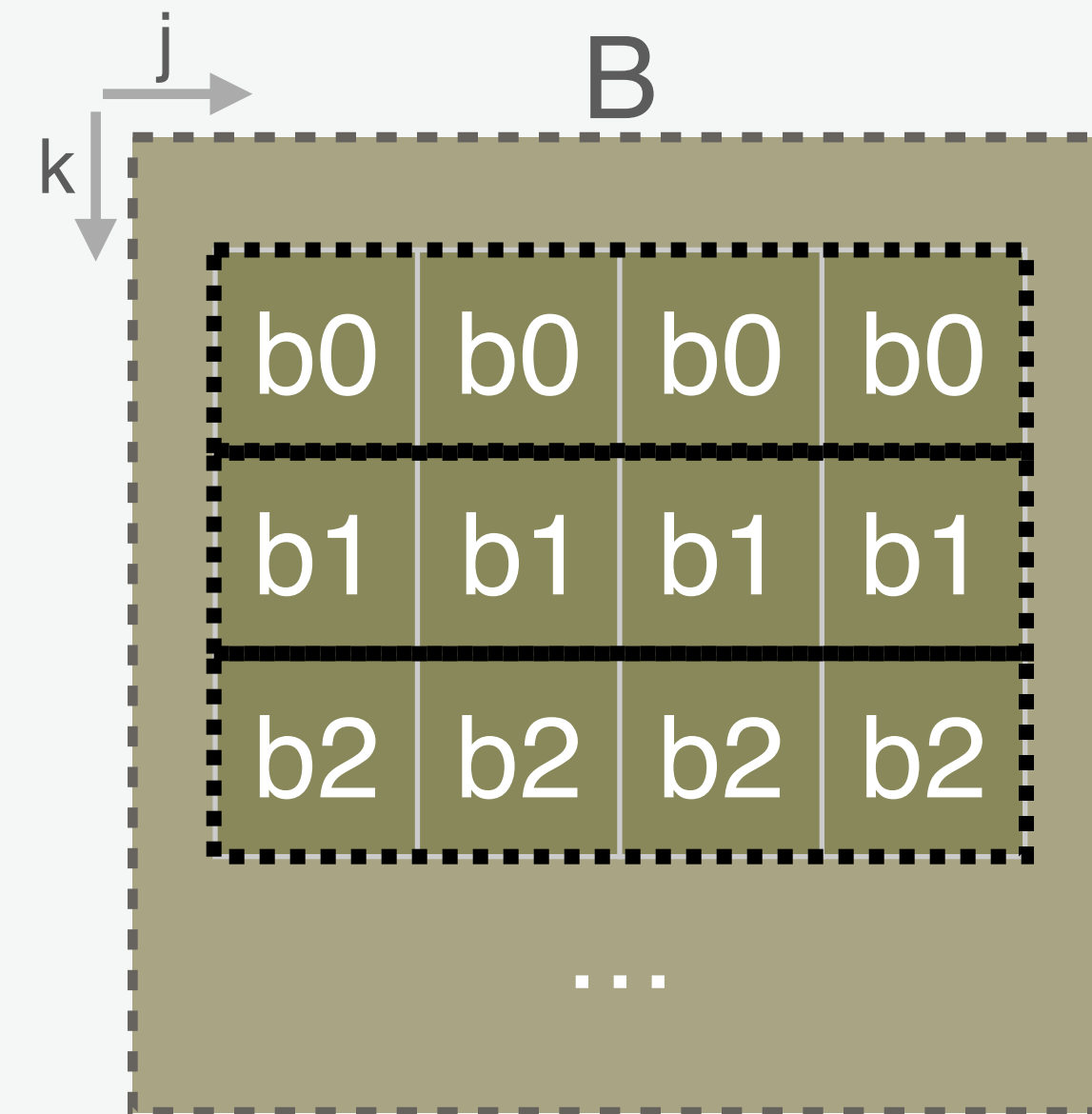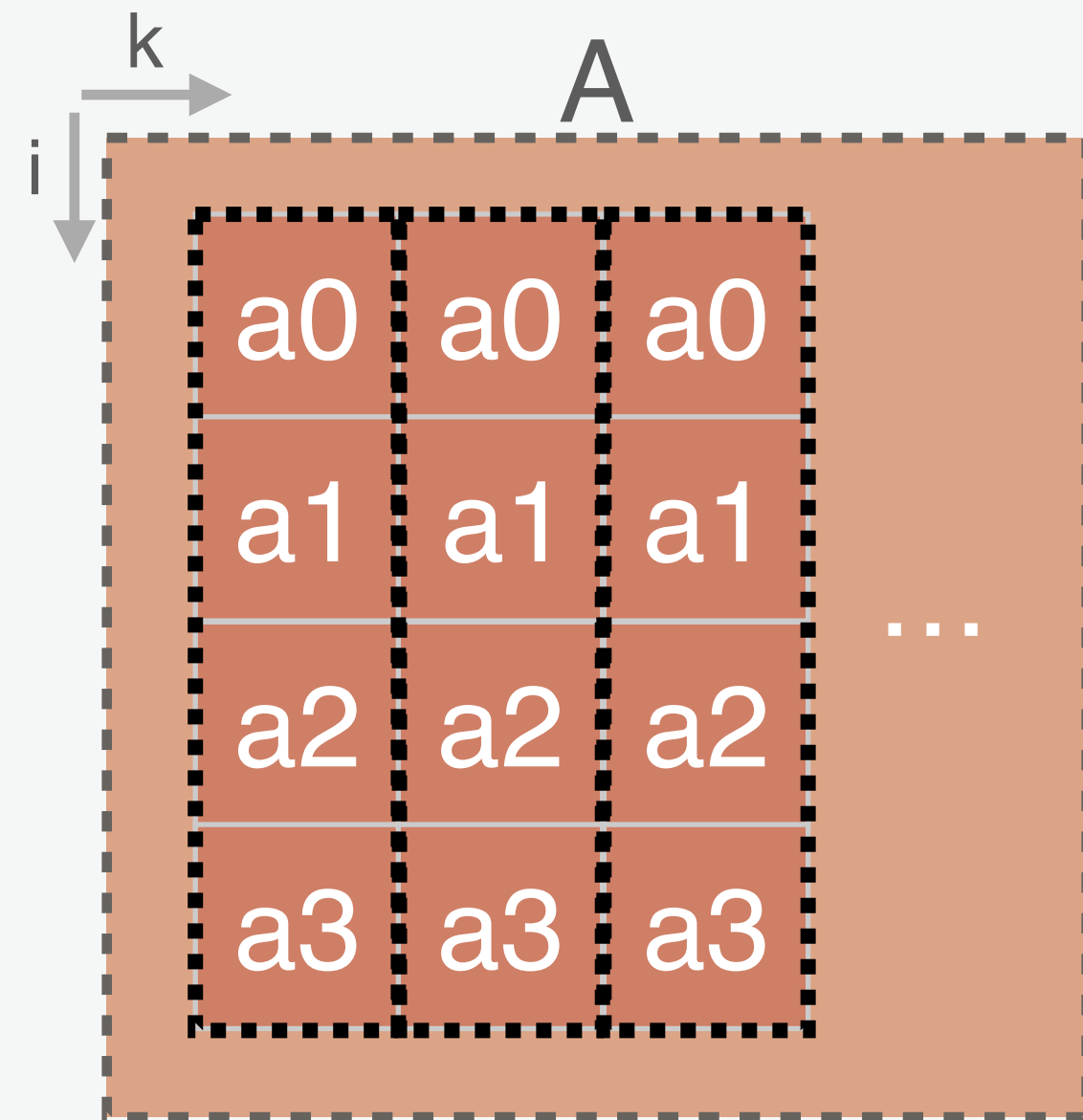
- Intel oneMKL

Order of memory accesses
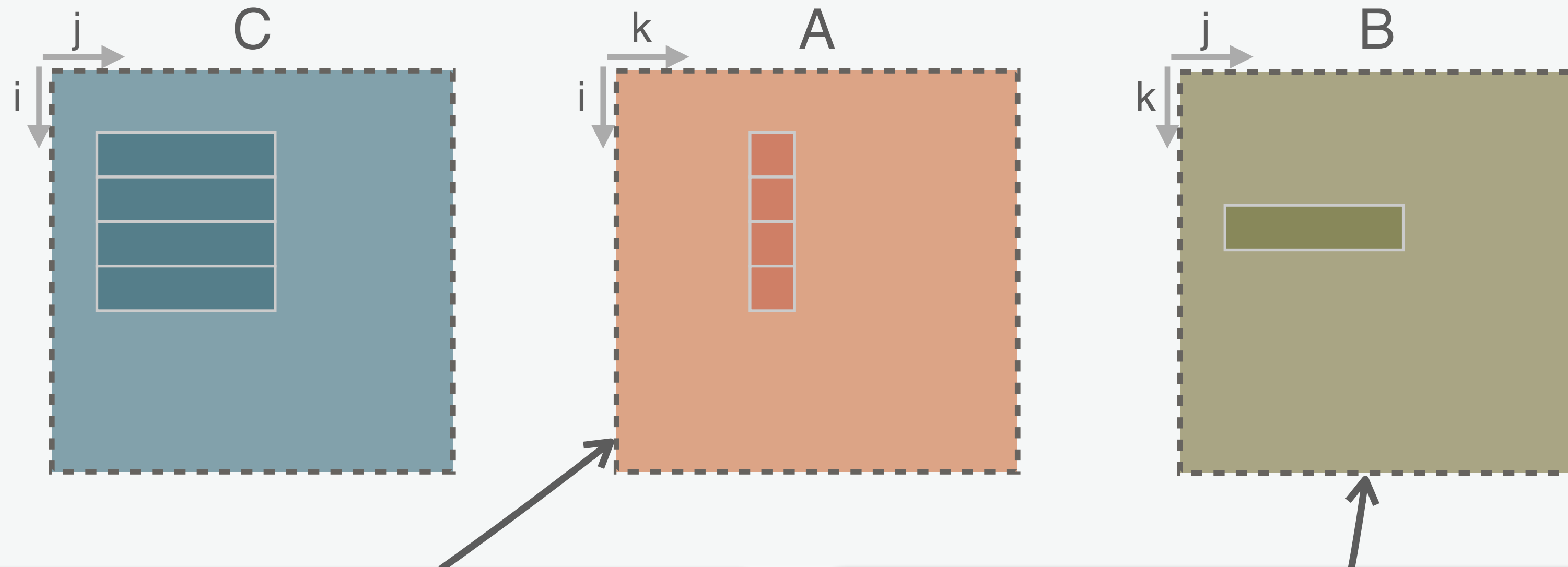
For each k, the base case accesses a **subcolumn** of elements of A.

For each k, the base case accesses a **subrow** of elements of B.

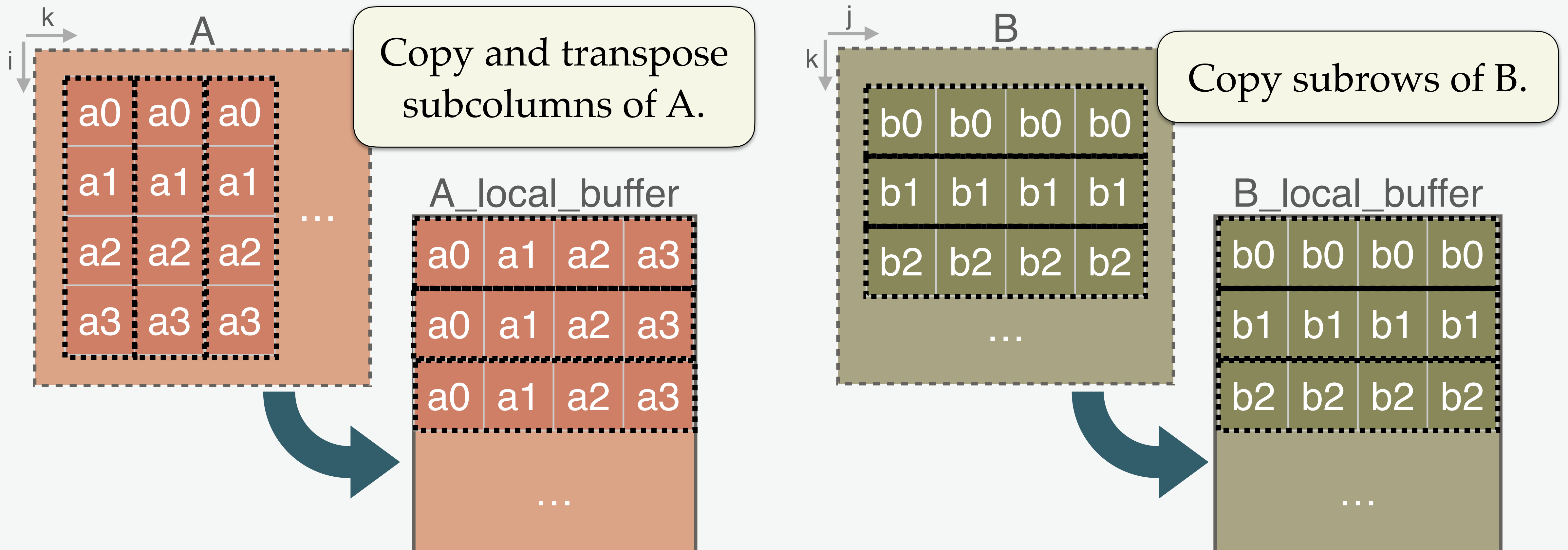Ideally, the base case should access all memory **in order**.

Ideal data layouts for A and B

Local buffers

Idea: At the start of the base case, move the relevant entries in A and B into **small, local buffers**.

Indexing buffers for A and B

To index matrix elements in these local buffers, **interleave the bits** of the matrix index.

Index calculation for A buffer

```
int64_t A_index(int64_t i, int64_t k, int64_t BC) {
  return ((i / 8) * BC * 8) + (k * 8) + (i % 8);
```

The compiler can optimize the divisions by constants in these index calculations.

Index calculation for B buffer

```
int64_t B_index(int64_t j, int64_t k, int64_t BC) {
  return ((j / 24) * BC * 24) + (k * 24) + (j % 24);
```
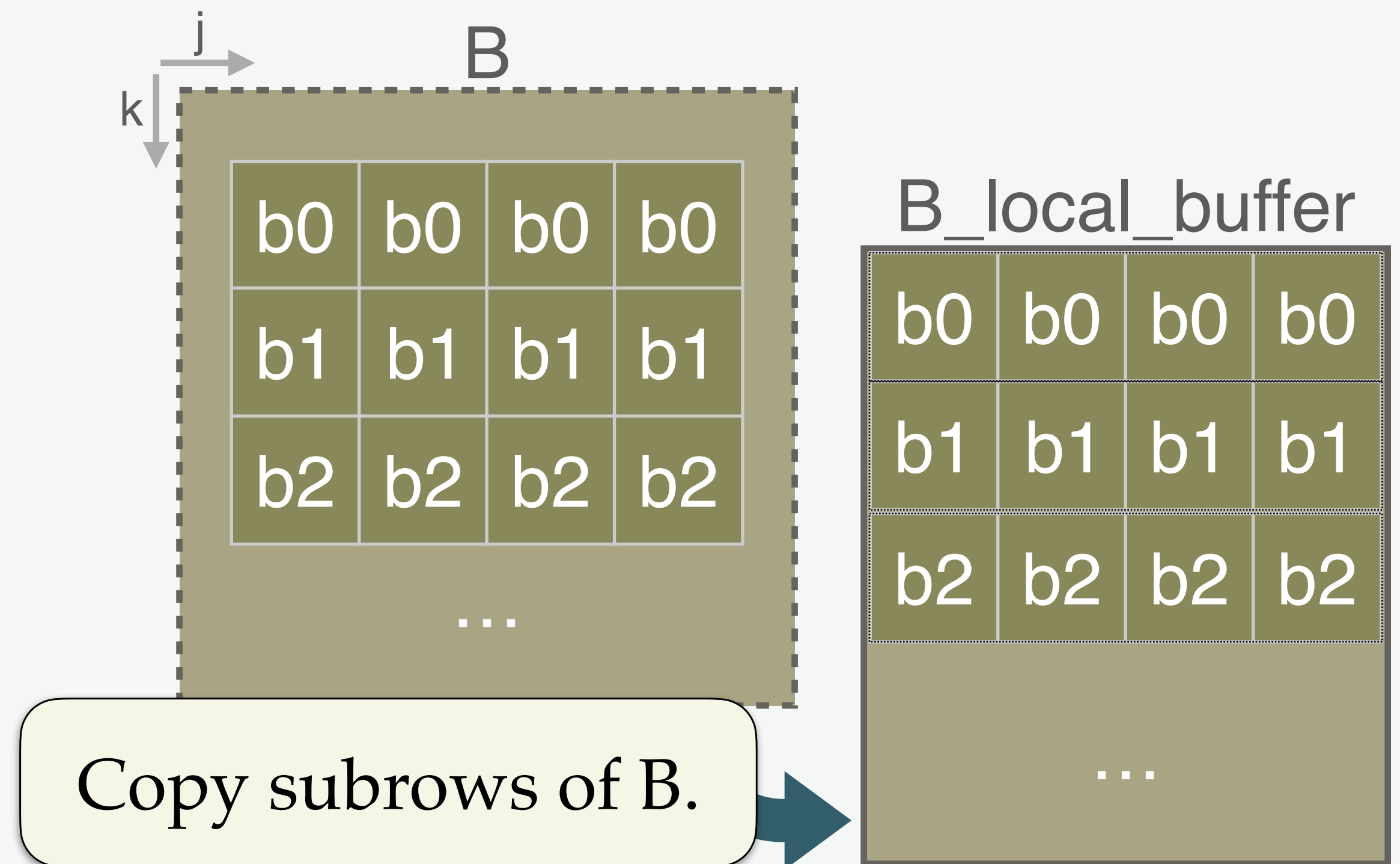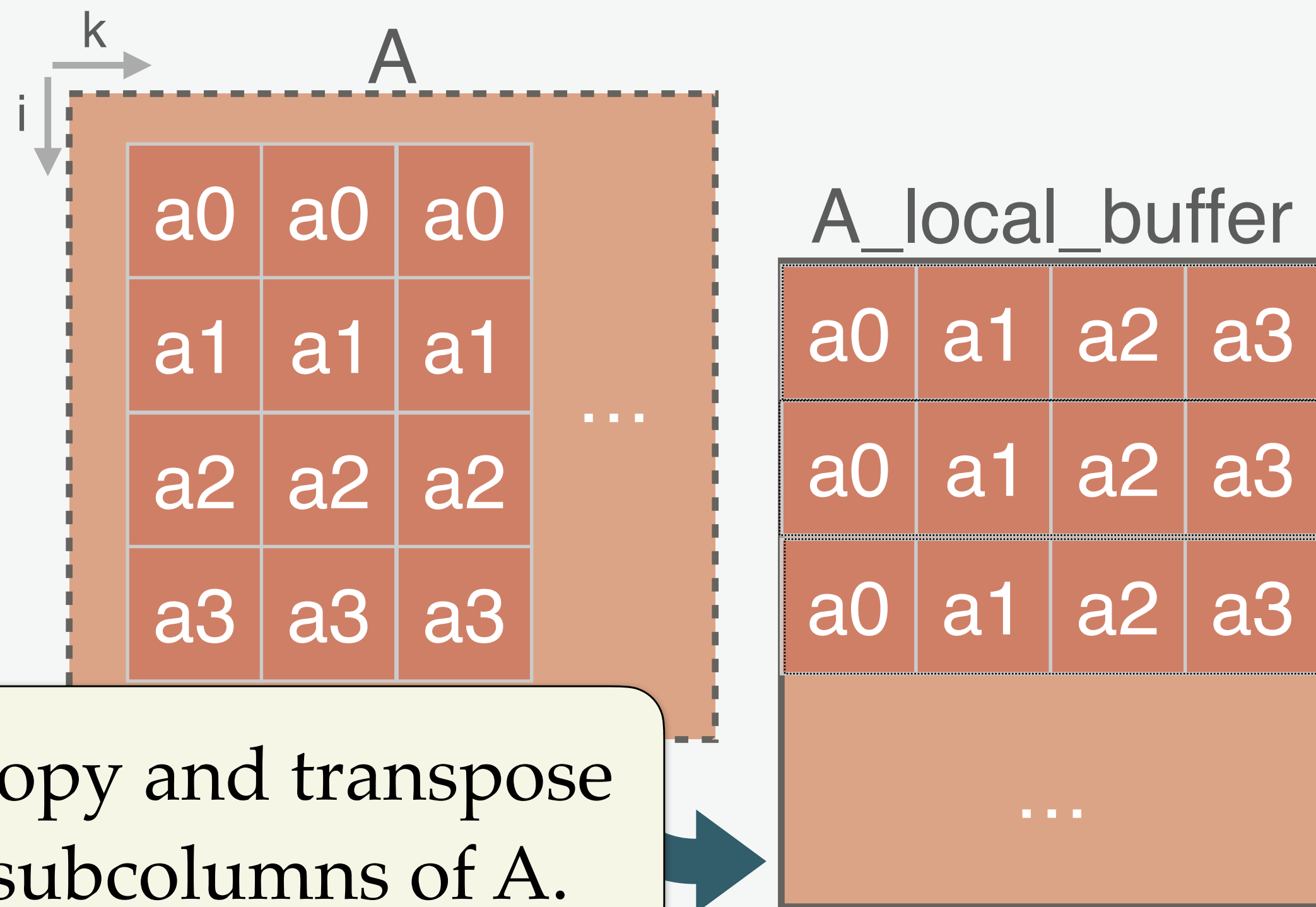
In the loop over k in the base case, these addresses simply increase.

What do these buffers cost?

How much work does this filling a local buffer add to the base case of size S?

- In theory, $\Theta(S^2)$, which the $\Theta(S^3)$ work of the base case dominates.

- In practice, after optimizing the $\Theta(S^3)$ base-case work, this cost is noticeable.

Copy and transpose subcolumns of A.
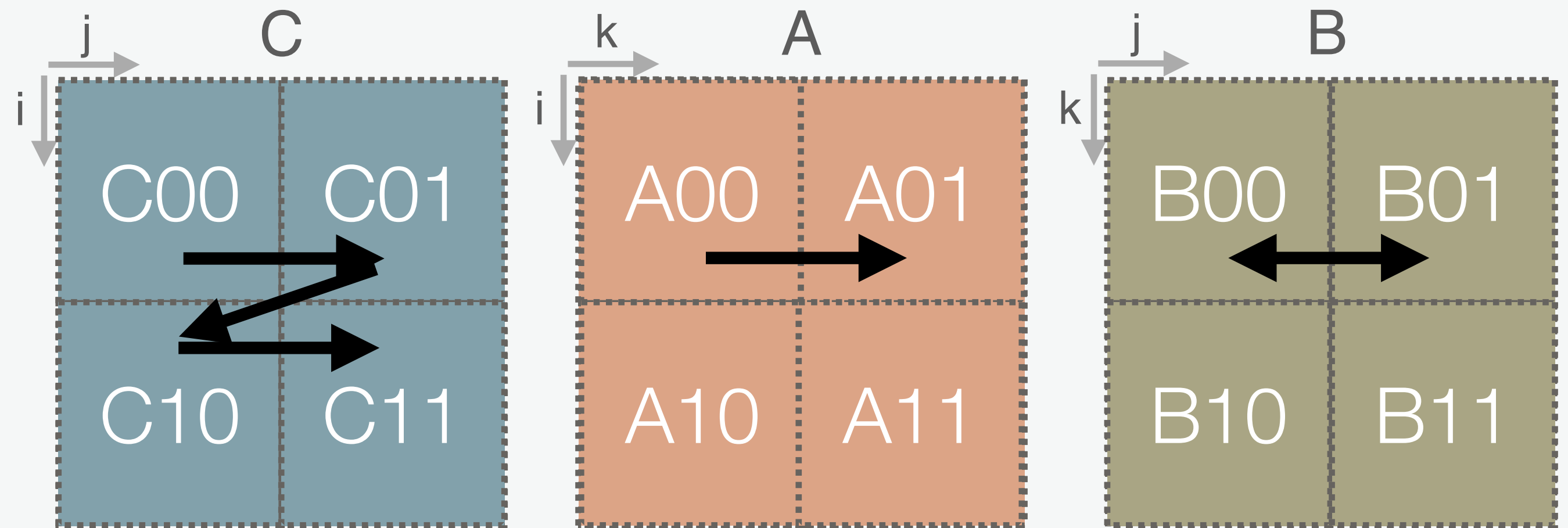
Copy subrows of B.

Avoiding unnecessary work

How can we reduce the extra work of filling local buffers?

Matrix-multiply routine

```
void mmdac(double *restrict C, double *restrict A,
           double *restrict B, size_t size) {
  if (size == S) {
    mmbase(C, A, B);
  } else {
    size_t s00 = 0;
    size_t s01 = size/2;
    size_t s10 = (size/2)*n;
    size_t s11 = (size/2)*(n+1);
    cilk_scope {
     cilk_spawn mmdac(C+s00, A+s00, B+s00, size/2);
     cilk_spawn mmdac(C+s01, A+s00, B+s01, size/2);
     cilk_spawn mmdac(C+s10, A+s10, B+s00, size/2);
                mmdac(C+s11, A+s10, B+s01, size/2);
    }
    cilk_scope {
     cilk_spawn mmdac(C+s00, A+s01, B+s10, size/2);
     cilk_spawn mmdac(C+s01, A+s01, B+s11, size/2);
```

j   C   k   A   j   B

i | C00 | C01 | i | A00 | A01 | k | B00 | B01 |
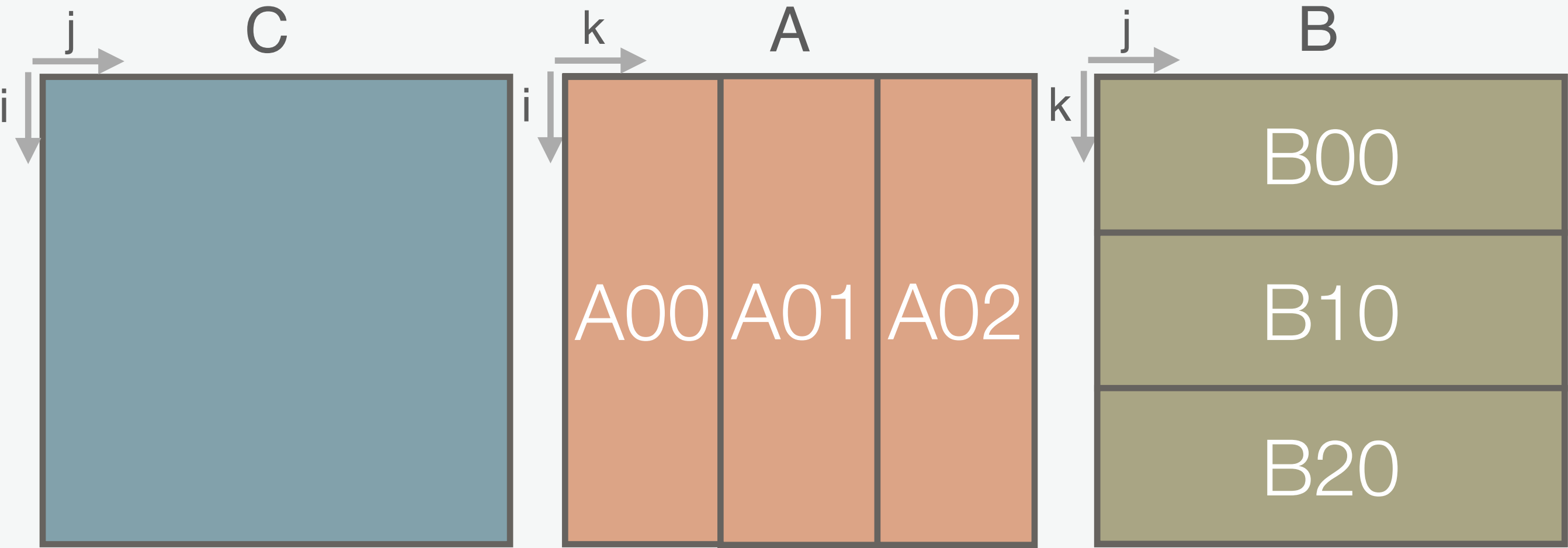 | C10 | C11 | | A10 | A11 | | B10 | B11 |

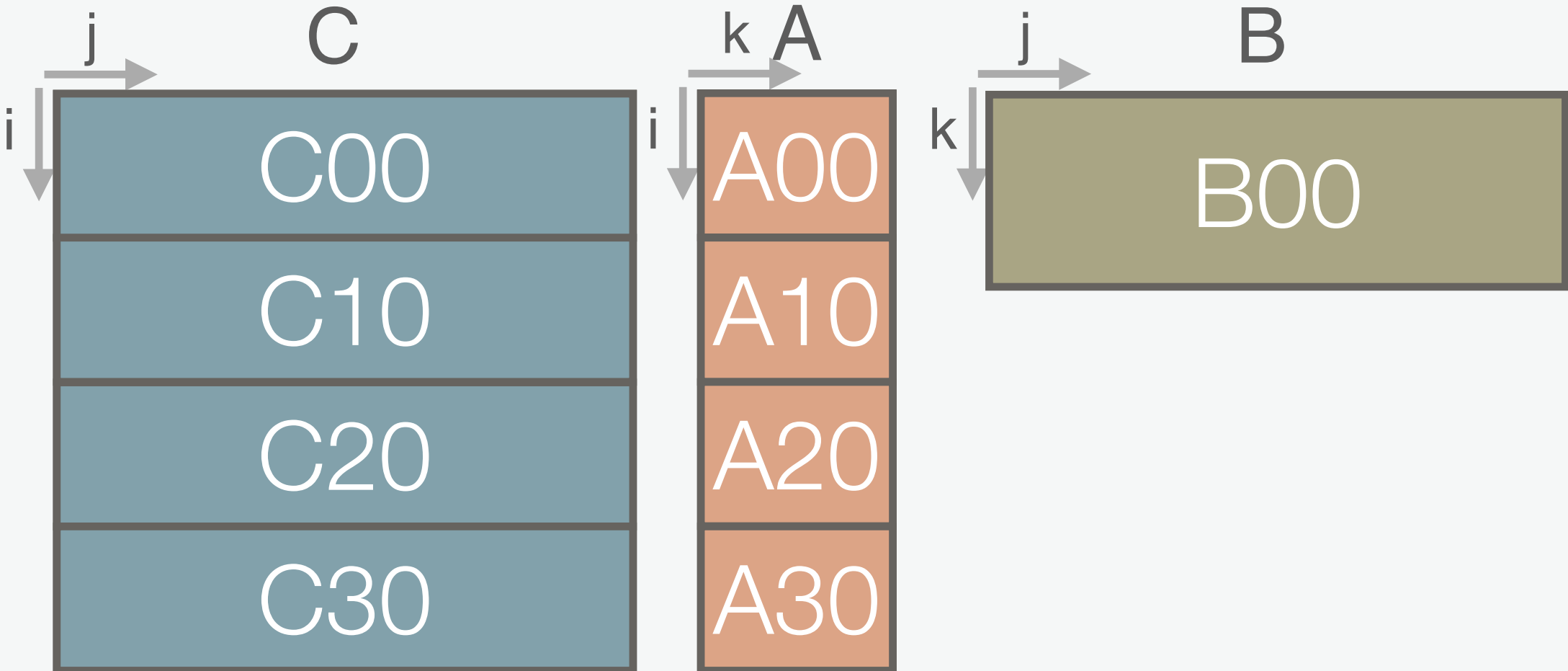If we use the same input submatrix between calls to mmbase(), we don't need to refill the corresponding buffer.

Reorder the calls to the base case to avoid refilling buffers!
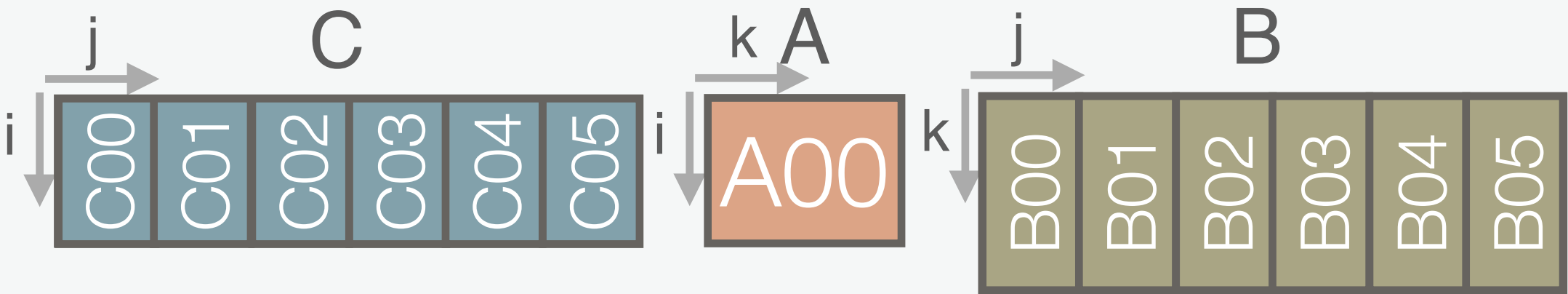
A new subdivision strategy

1. Divide the k dimension into large pieces.

C
j → i ↓

A
k → i ↓
A00 A01 A02

B
j → k ↓
B00
B10
B20

3. Divide the j dimension into small pieces.

2. Divide the i dimension into medium pieces.

C
j → i ↓
C00
C10
C20
C30

A
k → i ↓
A00
A10
A20
A30

B
j → k ↓
B00

C
j → i ↓
C00 C01 C02 C03 C04 C05

A
k → i ↓
A00

B
j → k ↓
B00 B01 B02 B03 B04 B05

Why does this subdivision strategy work?

1. Divide the k dimension into large pieces.

C   A   B

A00 A01 A02

B00

B10

B20
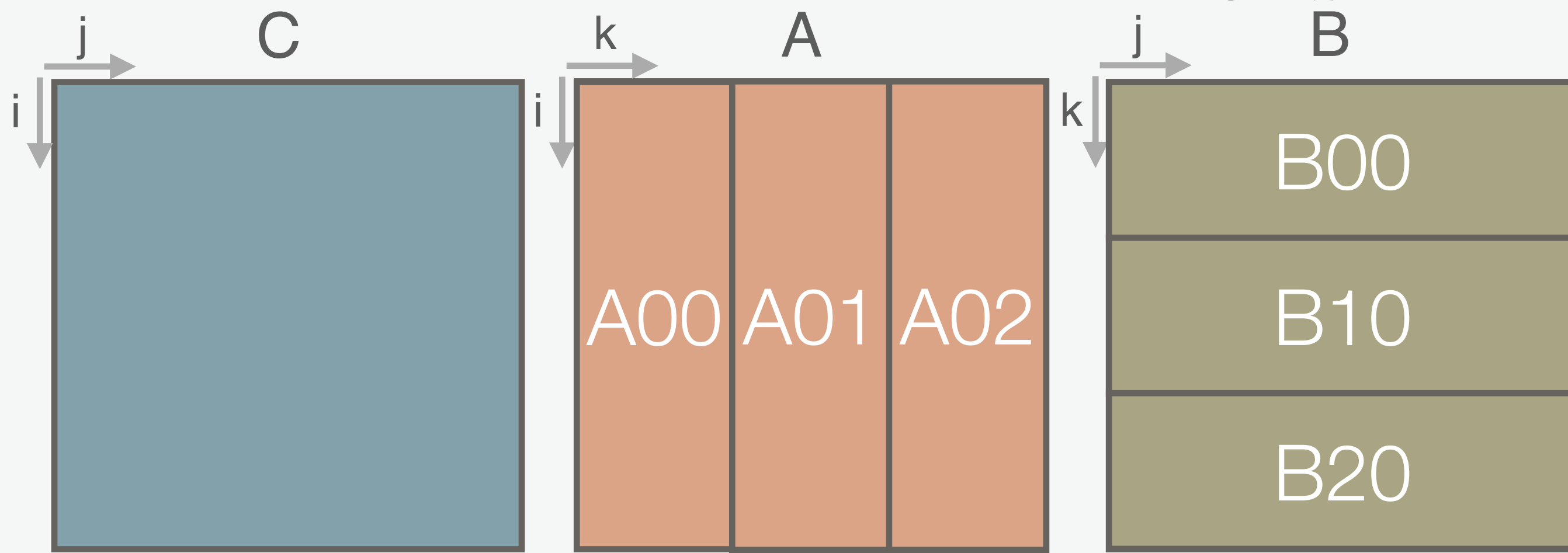
Larger k for the base case means fewer loads and stores from C.

It's cheaper to copy from B than to copy and transpose from A.

3. Divide the j dimension into small pieces.

2. Divide the i dimension into medium pieces.

C   A   B

C00

C10

C20

C30

A00

A10

A20

A30

B00

C   A   B

C00 C01 C02 C03 C04 C05

A00

B00 B01 B02 B03 B04 B05
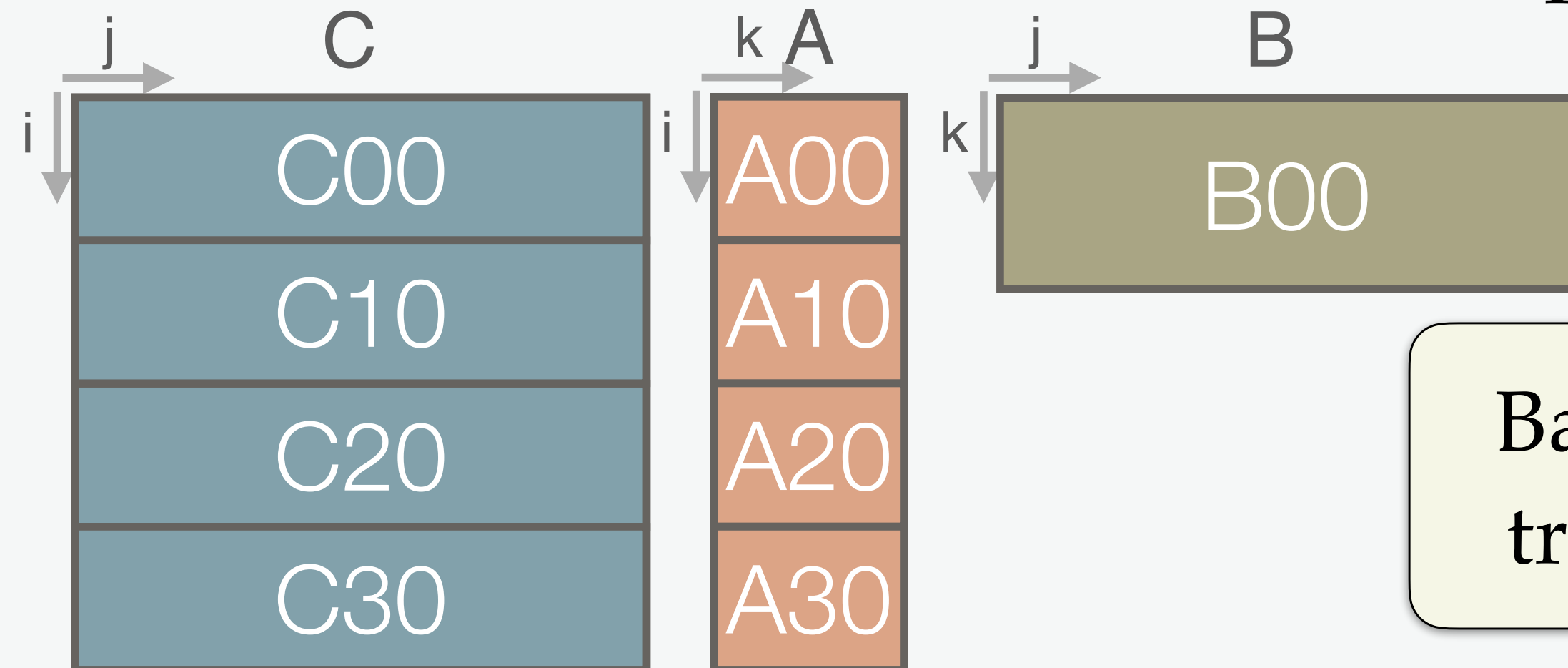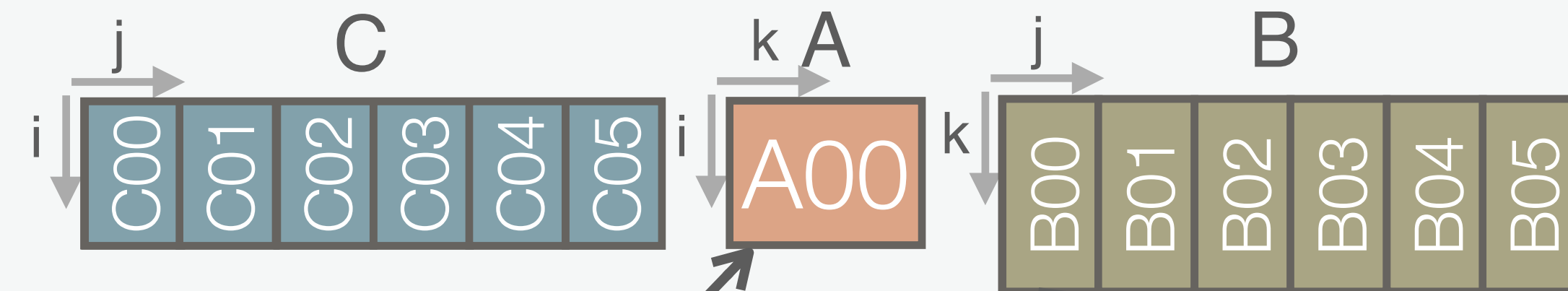
Base-cases fill local buffers with 1 copy and transpose from A and many copies from B.

Outline

- Compiler vectorization

- Vectorization by hand

- Vectorization by hand, another approach

- Performance-engineering the hand-vectorized version

- **Intel oneMKL**

Performance results versus oneMKL

On 48 cores, this hand-vectorized implementation outperforms the latest version of Intel oneMKL with OpenMP threads.

Implementation	Running time (s)	GFLOPS	Fraction of peak
Cilk, hand-vectorization	0.052	2656.893	57.658%
oneMKL with OpenMP	0.061	2268.232	49.224%

But on 24 cores on 1 chip, the performance difference is reversed!
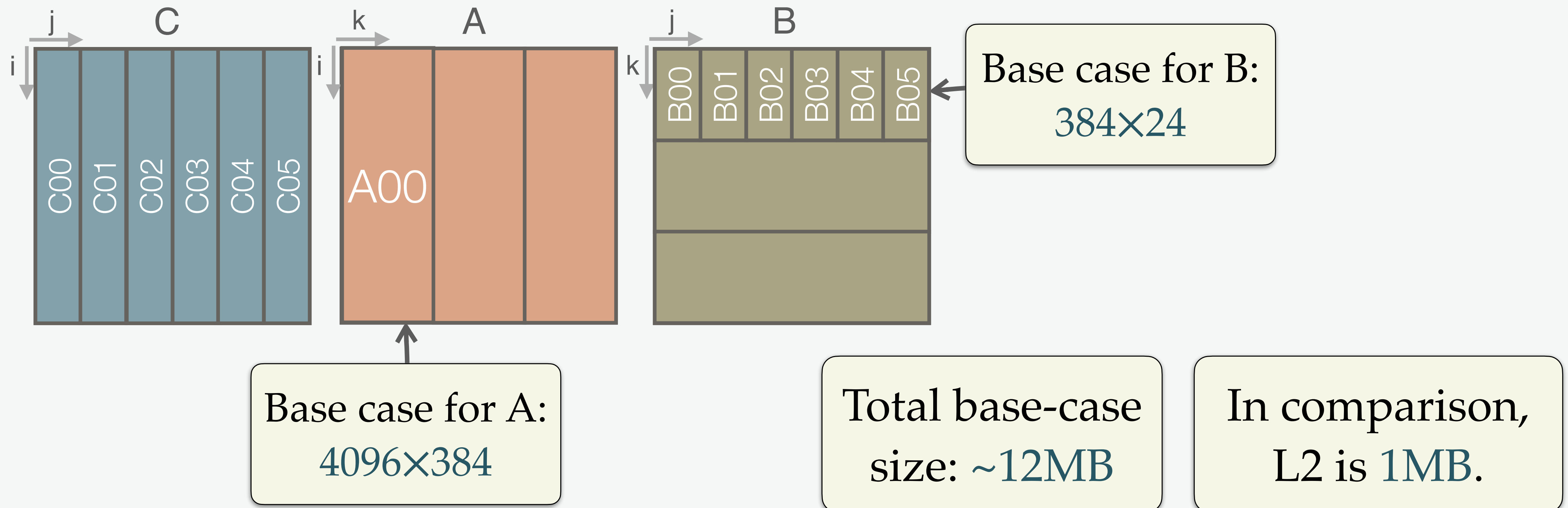
What's causing this performance difference?

Implementation	Running time (s)	GFLOPS	Fraction of peak
Cilk, hand-vectorization	0.092	1494.807	32.439%
oneMKL with OpenMP	0.082	1670.604	36.254%

Intel oneMKL

Intel oneMKL uses the same broadcast outer-product base case and local buffers, but it subdivides the problem differently.

Intel oneMKL matrix subdivision on 1 thread

Base case for B: 384×24

Base case for A: 4096×384

Total base-case size: ~12MB

In comparison, L2 is 1MB.

Software prefetching

Intel oneMKL's base case uses **software prefetching** to speed up memory accesses.

- The base case mixes a few software-prefetch operations to load locations in A and B local buffers to be accessed in the future.

- During the last few iterations before storing into C, the base case software-prefetches locations in C.

Snippet of broadcast-based matrix-multiply base case (simplified)

```
// Loop over k.
for (int k = 0; k < BC; ++k) {
  // Prefetch from A.
  __builtin_prefetch(&A[A_index(i, k + 4, BC)]);

  // Load a vector from B.
  vdouble bv = *(const vdouble *)(&B[B_index(k, j, BC)]);
  for (int ivec = 0; ivec < 8; ++ivec)
    // Load a value from A, broadcast that value,
    // and perform FMA.
    cv[ivec] += bv * A[A_index(i + ivec, k, BC)];
```

C

Load a value from A that will be used in a future loop iteration.

My experience with software prefetching

Software prefetch instructions are tricky to use.

- Software prefetches increase instruction counts, memory traffic, and data in residing in cache.

- Software prefetches can only improve performance if their use hides memory latency more effectively than the hardware prefetchers.

Careful uses of software prefetch instructions can improve performance by a few percent, but poor uses can hurt performance.

Takeaways

- One can use OpenCilk to implement parallel programs with state-of-the-art performance that competes with professionally engineered high-performance software.

- Many theoretically good algorithms can have the performance impact that theory predicts, but many systems issues need to be handled first.

- Even when most of a program's running time is spent in a small amount of the code, the keys to optimizing that hot spot might lie elsewhere in the code.

- Check your performance-testing methodology and your raw performance data!

Questions?