Case Study: Matrix Multiplication

6.S898: Advanced Performance Engineering for Multicore Applications
February 22, 2017

MIT COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE LABORATORY

4k-by-4k Matrix Multiplication

Version	*Implementation*	*Running time (s)*	*GFLOPS*	*Absolute speedup*	*Relative speedup*	*Fraction of peak*
1	Python	25,552.48	0.005	1	—	0.00%
2	Java	2,372.68	0.058	11	10.8	0.01%
3	C	542.67	0.253	47	4.4	0.03%
4	Parallel loops	69.80	1.969	366	7.8	0.24%
5	Parallel divide-and-conquer	3.80	36.180	[illegible]	[illegible]	[illegible]
6	+ vectorization	1.10	124.914	23,224	3.5	14.96%
7	+ AVX intrinsics	0.41	337.812	62,806	2.7	40.45%
8	Strassen	0.38	361.177	67,150	1.1	43.24%

Today, we'll look into the performance engineering behind versions 3–7.

Outline

- The matrix multiplication problem
- Serial and parallel looping codes
- Cache-efficient matrix multiplication
- Hands-on: Vectorization using the compiler
- Vectorization by hand

Matrix Multiplication

Problem: Compute the product $\mathsf{C} = (c_{ij})$ of two $n \times n$ matrices $\mathsf{A} = (a_{ij})$ and $\mathsf{B} = (b_{ij})$.

The matrix product obeys the following formula:

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

For simplicity, we shall assume that n is a power of 2.

Three Nested Loops in C

```
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < n; ++j) {
    for (int k = 0; k < n; ++k) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Work of computation:

- n^3 iterations
- Each iteration performs constant work.

$\Theta(n^3)$ total work.

GCC version 5.2.1 with -O3 optimization.

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak
3	C	542.67	0.253	47	4.4	0.03%

Parallel Loops

```
cilk_for (int i = 0; i < n; ++i) {
  cilk_for (int j = 0; j < n; ++j) {
    for (int k = 0; k < n; ++k) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Compute each element of C in parallel.

Version	*Implementation*	*Running time (s)*	*GFLOPS*	*Absolute speedup*	*Relative speedup*	*Fraction of peak*
3	C	542.67	0.253	47	4.4	0.03%
4	Parallel loops	69.80	1.969	366	7.8	0.24%

But the machine has 18 cores!
Where's my 18x speedup!?

Work/Span Analysis of Parallel Loops

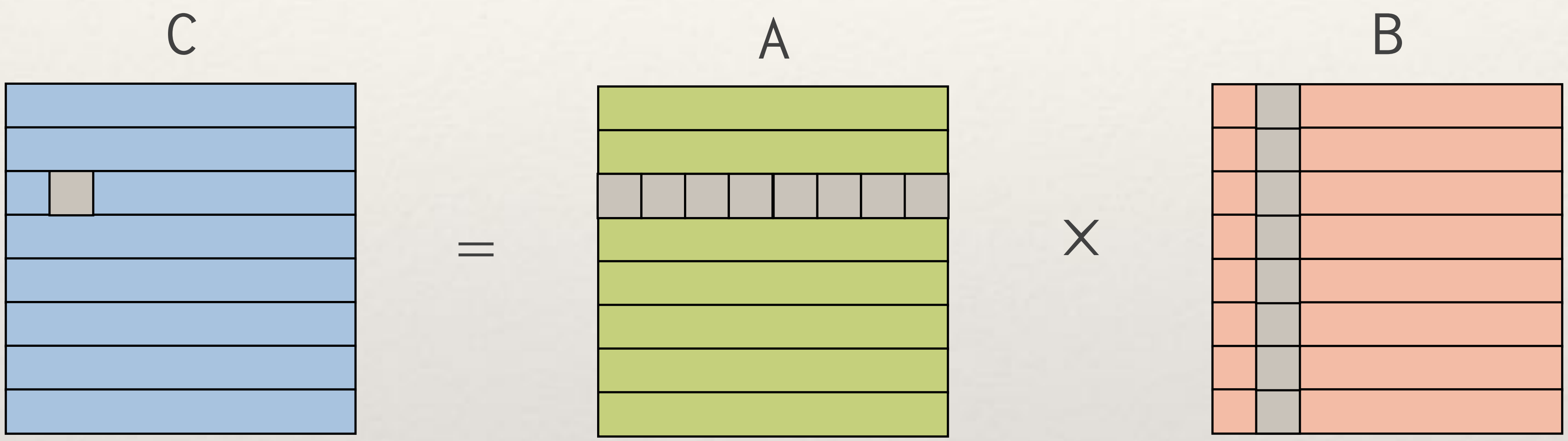
```
cilk_for (int i = 0; i < n; ++i) {
  cilk_for (int j = 0; j < n; ++j) {
    for (int k = 0; k < n; ++k) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

- **Work:** $T_1(\text{n}) = \Theta(n^3)$
- **Span:** $T_\infty(n)$
  $= \Theta(\log n + \log n + n)$
  $= \Theta(n)$
- **Parallelism:**
  $T_1(n) / T_\infty(n) = \Theta(n^2)$

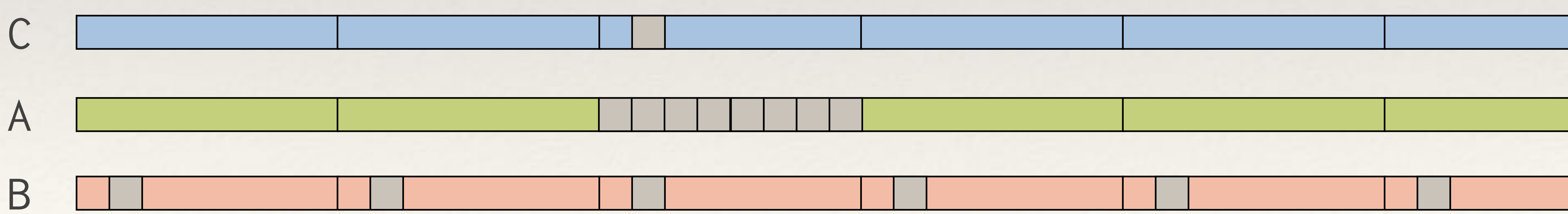
This code has **ample** parallelism,
but still gets **poor** parallel speedup!

Memory Access Pattern for Looping Code

Matrices are stored in **row-major order.**

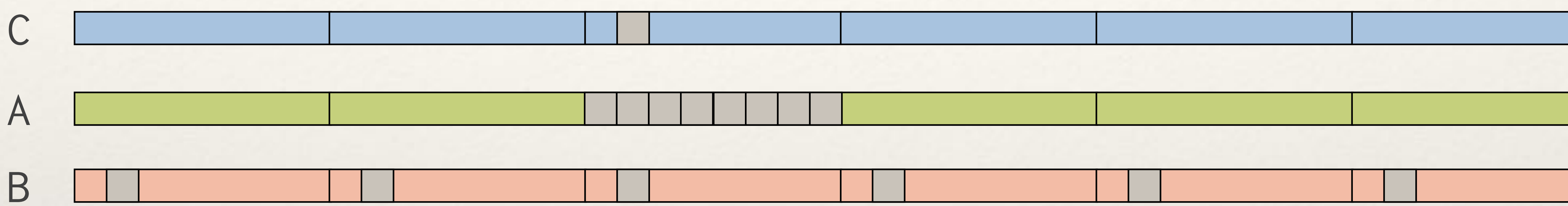


Layout of matrices in memory:



Cache Analysis of Looping Code

Layout of matrices in memory:



Suppose that n is sufficiently large. Let B be the size of a cache line.

- Computing an element of matrix C involves $\Theta(n/B)$ cache misses for matrix A and $\Theta(n)$ cache misses for matrix B.
- **No temporal locality** on matrix B. Cache can't store all of the cache lines for one column of matrix B.
- Computing **each** element of matrix C incurs $\Theta(n)$ cache misses.
- In total, $\Theta(n^3)$ cache lines are read to compute all of matrix C.

Improving Cache Efficiency

We can improve cache efficiency using a **recursive divide-and-conquer** algorithm.

- Imagine each matrix is subdivided into four quadrants.

$$C = \begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} \quad A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \quad B = \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

- The matrix product can be expressed recursively in terms of 8 products of submatrices:

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{pmatrix}$$

Recursive Divide-And-Conquer

Submatrices

Promise to compiler that matrices don't alias

Computation of submatrices

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

Dimension of submatrices

Dimension of original matrices

Coarsened base case

Recursive calls

Analysis of Recursive Divide-And-Conquer

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

Work of computation:

- Recurrence:
  $T(n) = 8T(n/2) + \Theta(1)$
- Solve the recurrence via the Master Method:
  $T(n) = \Theta(n^3)$

Analysis of Recursive Divide-And-Conquer

```c
void mmdac(double *restrict C,
           double *restrict A,
           double *restrict B,
           int size, int n) {
  if (size <= THRESHOLD) {
    mmbase(C, A, B, size);
  } else {
    int s00 = 0;
    int s01 = size/2;
    int s10 = (size/2)*n;
    int s11 = (size/2)*(n+1);
    mmdac(C+s00, A+s00, B+s00, size/2, n);
    mmdac(C+s01, A+s00, B+s01, size/2, n);
    mmdac(C+s10, A+s10, B+s00, size/2, n);
    mmdac(C+s11, A+s10, B+s01, size/2, n);
    mmdac(C+s00, A+s01, B+s10, size/2, n);
    mmdac(C+s01, A+s01, B+s11, size/2, n);
    mmdac(C+s10, A+s11, B+s10, size/2, n);
    mmdac(C+s11, A+s11, B+s11, size/2, n);
 }
}
```

Cache complexity: Let M be the cache size and B the size of a cache line. Assume the base case size fits in cache.

- Base case incurs $\Theta(n^2/B)$ cache misses.
- Recursiv… $Q(n) = 8$… cache m…
- Solution: $Q(n) = \Theta(n^3/M^{1/2}B)$

Significant improvement over $\Theta(n^3)$ misses from looping code.

Parallel Divide-And-Conquer

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

This code has **ample** parallelism.

Work:

$T_1(n) = \Theta(n^3)$

Span:

Recurrence:

$T_\infty(n)$

$= 2T_\infty(n/2) + \Theta(1)$

Solution:

$T_\infty(n) = \Theta(\mathrm{n})$

Performance of Parallel Divide-And-Conquer

Version	*Implementation*	*Running time (s)*	*GFLOPS*	*Absolute speedup*	*Relative speedup*	*Fraction of peak*
3	C	542.67	0.253	47	4.4	0.03%
4	Parallel loops	69.80	1.969	366	7.8	0.24%
5	Parallel divide-and-conquer	3.80	36.180	6,727	18.4	4.33%

Where To Optimize Next?

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

Work of computation:

- Write the recurrence:
  $T(n) = 8T(n/2) + \Theta(1)$
- Solve the recurrence via Master Method:
  $T(n) = \Theta(n^3)$

Practically all of the work is in the **base case!**

Hands-On: Implement the Base Case

- Download mm_dac.c: http://pastebin.com/dl/MSmqi5Bq
- Implement a simple base case.

```
void mmbase(double *restrict C,
            double *restrict A,
            double *restrict B,
            int size) {
  for (int i = 0; i < size; ++i) {
    for (int j = 0; j < size; ++j) {
      for (int k = 0; k < size; ++k) {
        C[i*n+j] += A[i*n+k] * B[k*n+j];
      }
    }
  }
}
```

- Compile the code:

```
$ clang -O3 -g -fcilkplus -o mm_dac mm_dac.c
```

- Run it!

Hands-On: Vectorization Report

Is this compiler vectorizing your code?

- Add the flags `-Rpass=vector` and `-Rpass-analysis=vector` to your `clang` arguments to get a **vectorization report**.
- What does the report say?

For more on LLVM's `-Rpass` flag, see http://blog.llvm.org/2014/11/loop-vectorization-diagnostics-and.html.

IEEE Floating-Point Arithmetic

IEEE floating-point arithmetic is **not associative.**

- The statement `printf(“%.17f”, (0.1+0.2)+0.3);` produces 0.60000000000000009.
- The statement `printf(“%.17f”, 0.1+(0.2+0.3));` produces 0.59999999999999998.

The compiler must assume that you care about this imprecision and therefore cannot reorder the floating-point operations in order to vectorize.

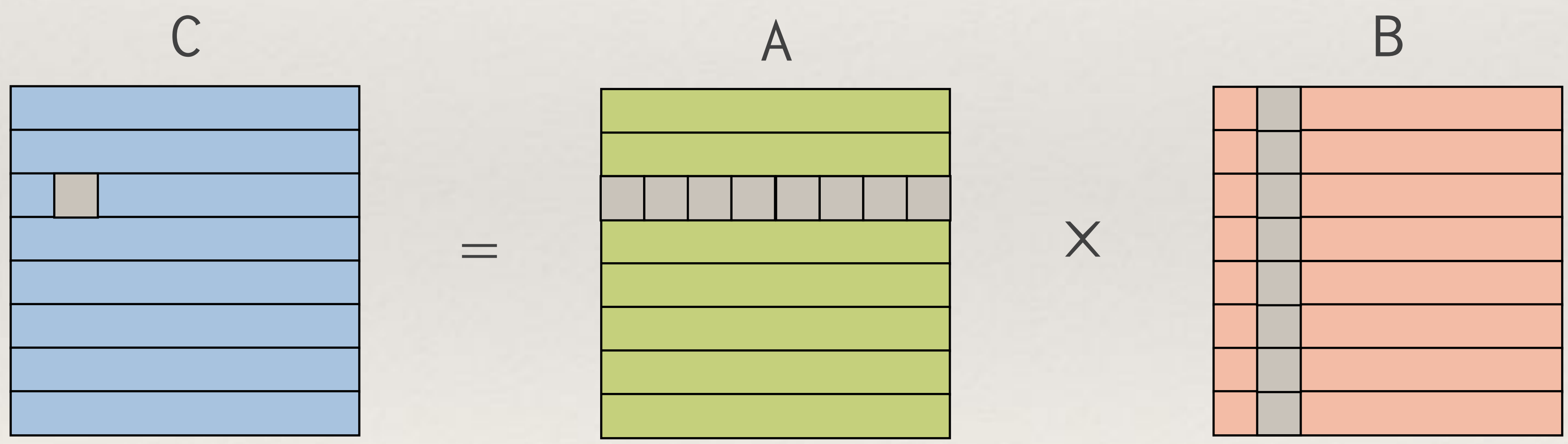
Hands-On: Vectorization, Attempt 1

We don’t care about this level of precision in the code’s floating-point arithmetic, so let’s add the `-ffast-math` flag to `clang` command.

- Is the performance any better?
- What does the vectorization report say now?

Why Didn't It Vectorize?

LLVM does not deem it efficient to vectorize the innermost loop, which reads a column of matrix B.



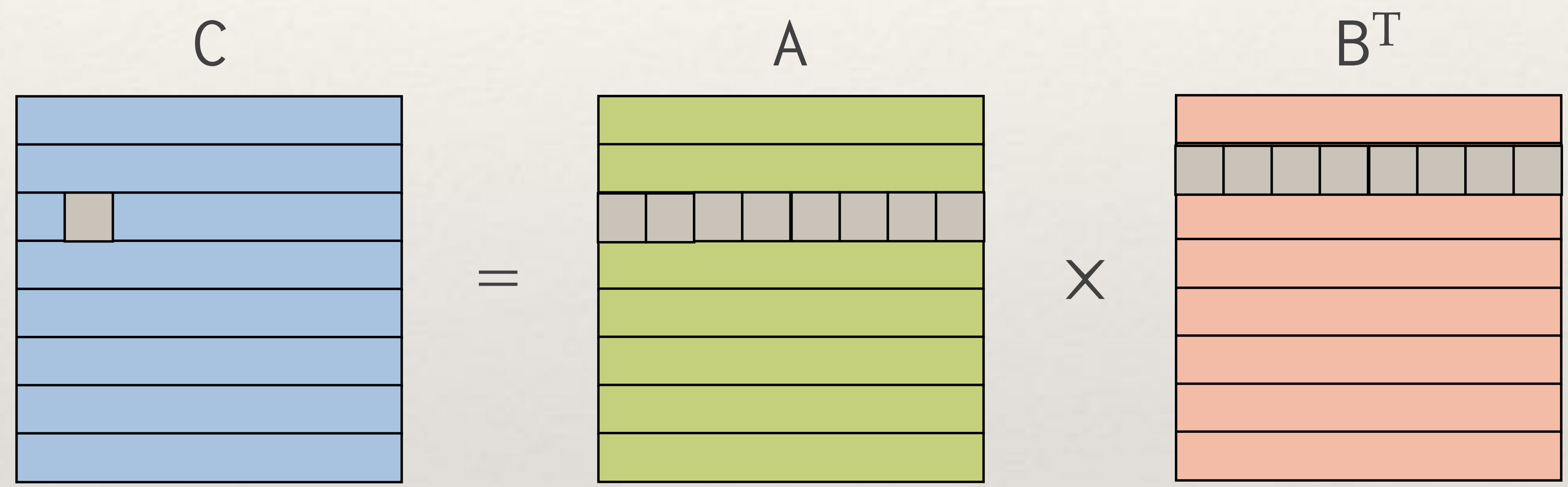
Hands-On: Vectorization, Attempt 2

Here are two strategies you can try for fixing this problem:

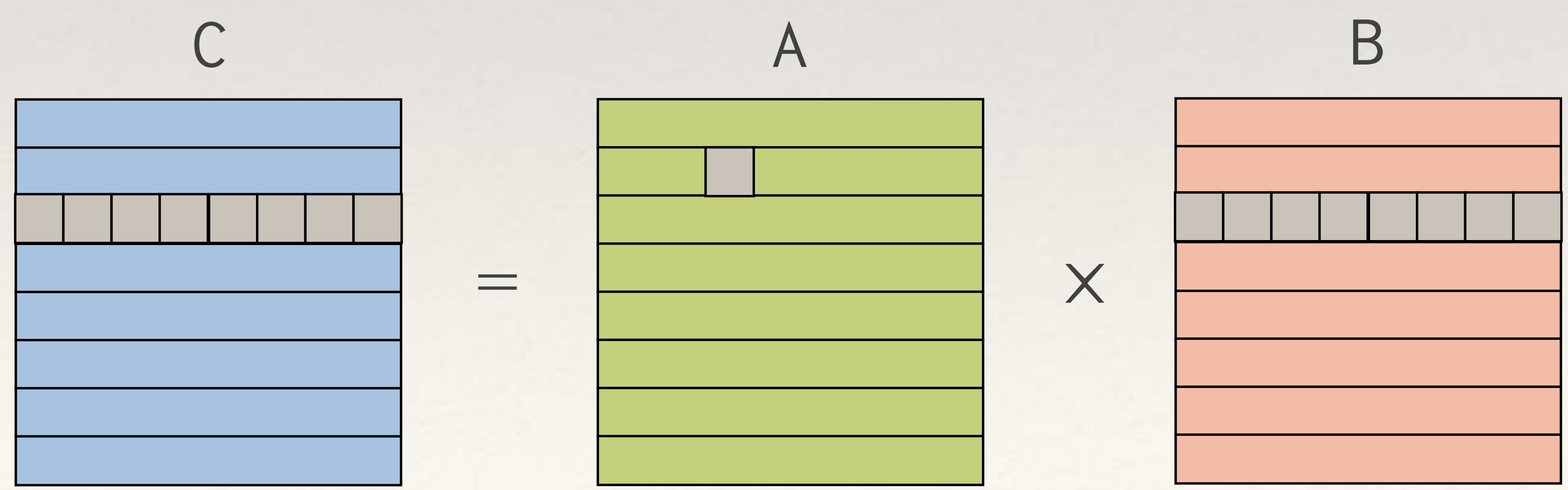
Strategy

Resulting vectorizable access pattern

1. Transpose matrix B.



2. Interchange the loops.



AVX Vector Instructions

Modern Intel processors support the AVX vector instruction set.

- AVX supports **256-bit vector registers**, whereas the older SSE instruction set supports 128-bit vector registers.
- Many common AVX instructions operate on 3 operands, rather than 2, making them **easier to use**.

Hands-On: Vectorization, Attempt 3

Once you have code that vectorizes, try using the AVX instructions, which can operate on 4 elements each.

- Add the `-mavx` flag to your `clang` command.
- What does the vectorizer report say now?
- Did you get a performance increase?

Performance With Vectorization

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak
3	C	542.67	0.253	47	4.4	0.03%
4	Parallel loops	69.80	1.969	366	7.8	0.24%
5	Parallel divide-and-conquer	3.80	36.180	6,727	18.4	4.33%
6	+ vectorization	1.10	124.914	23,224	3.5	14.96%

How do we go even faster?

Vector Intrinsics

Intel provides a library of intrinsic instructions for accessing their various vector instruction sets.

- C/C++ header: `immintrin.h`
- Database of vector intrinsic instructions: https://software.intel.com/sites/landingpage/IntrinsicsGuide/

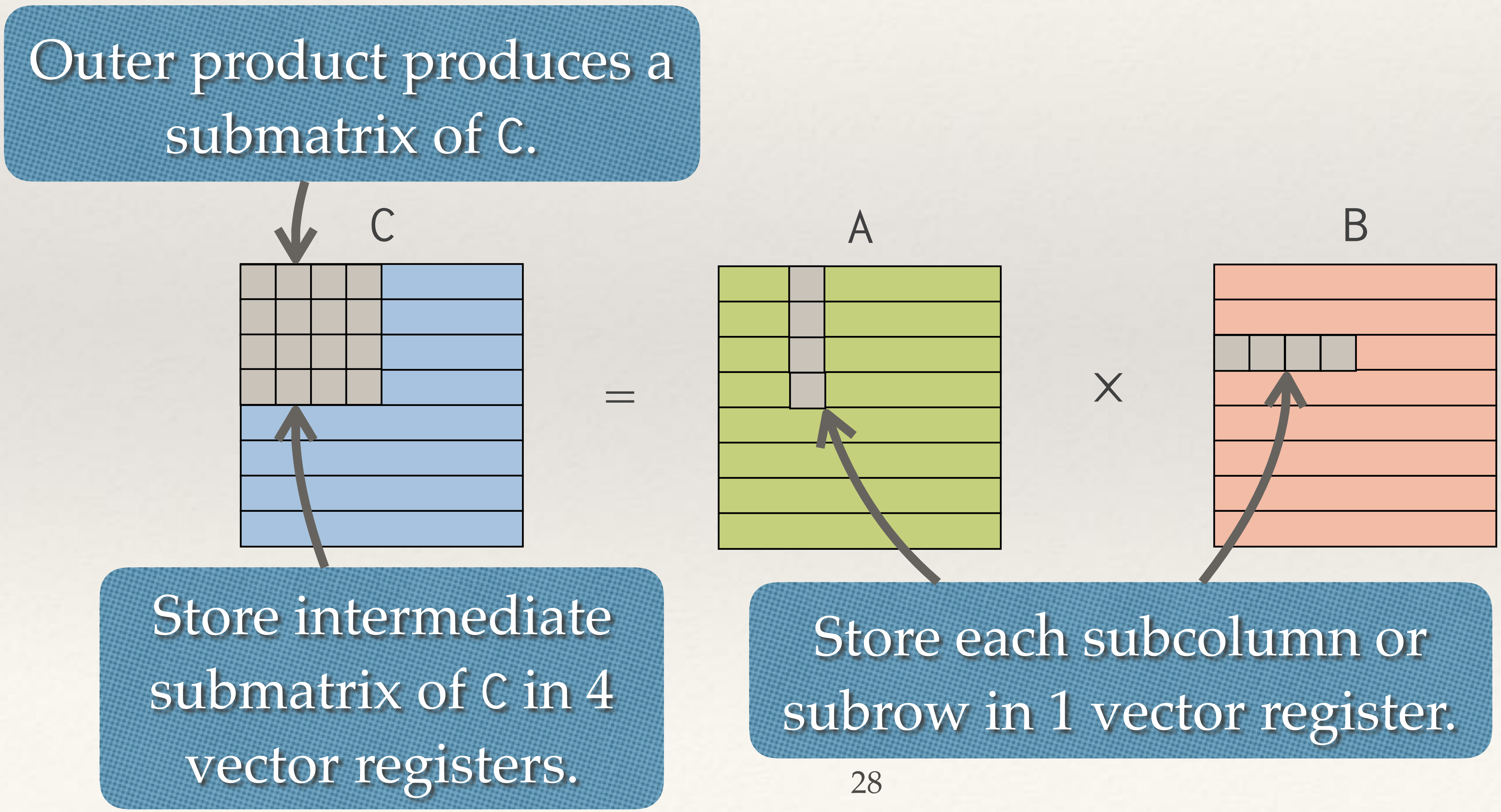
Some Useful AVX/AVX2 Instructions

If we stare at this database and think creatively, we come up with an alternative base case for matrix multiplication!

- The `__m256d` type stores a vector of 4 doubles.
- The AVX intrinsics `_mm256_add_pd()` and `_mm256_mul_pd()` perform addition and multiplication.
- The AVX2 intrinsic `_mm256_fmadd_pd()` performs a **fused multiply-add**.
- The AVX intrinsics `_mm256_permute_pd()` and `_mm256_permute2f128_pd()` permute AVX registers.

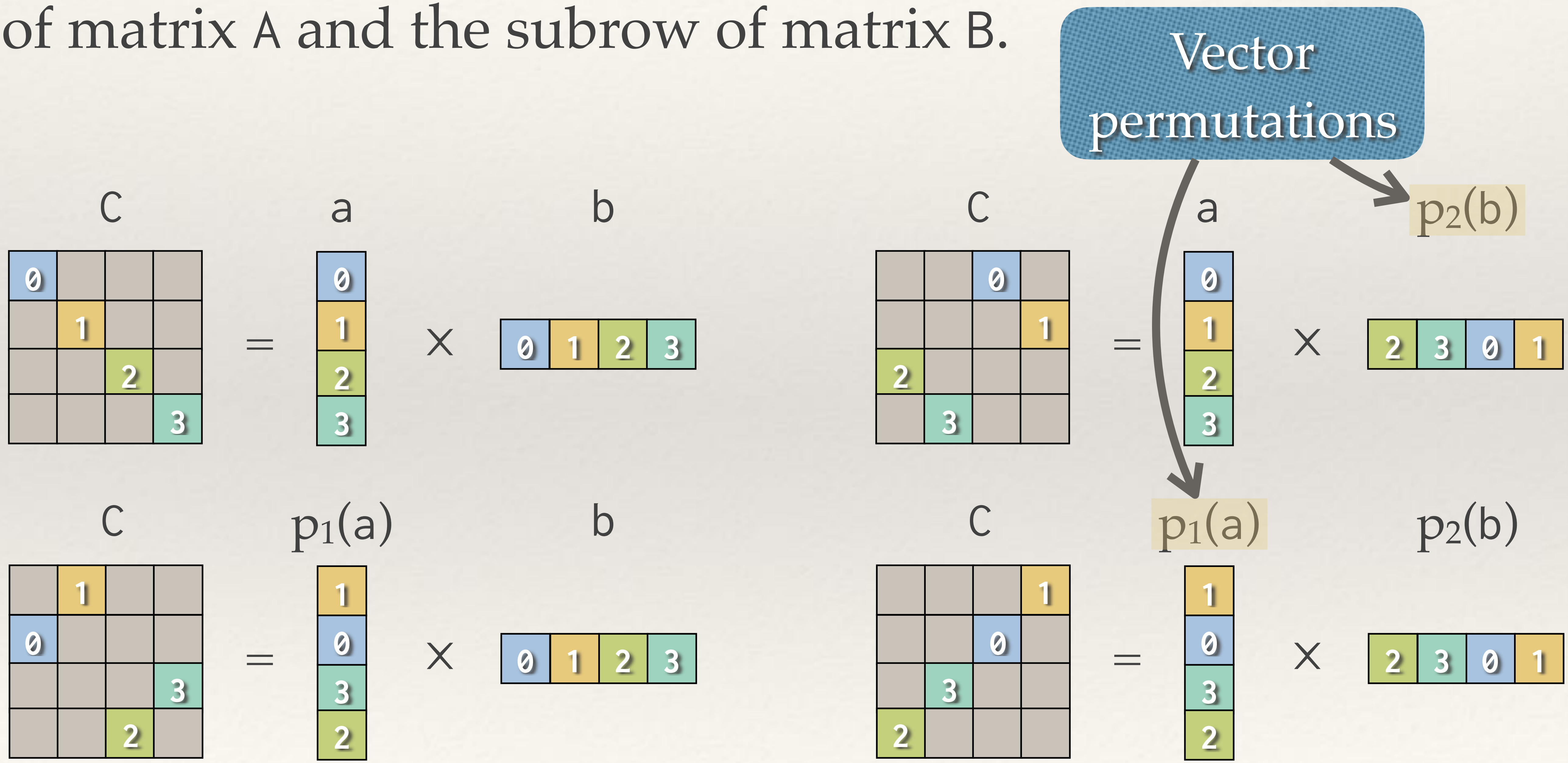
Outer Product Base Case

Idea: Compute outer products between subcolumns of matrix A by subrows of matrix B.



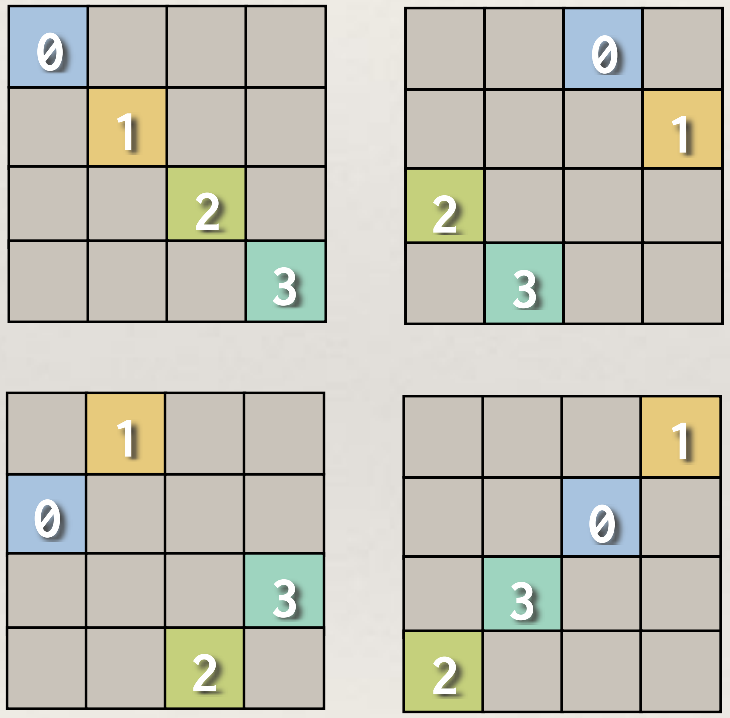
Computing One Outer Product

Compute 4 vector multiplications between the subcolumn of matrix A and the subrow of matrix B.



Computing a Whole Submatrix

Computed products



- Iterate through subcolumns of `A` and subrows of `B` to compute a submatrix of `C`.
- Accumulate elements of `C` submatrix in separate vector registers.
- Once done, write `C` submatrix back to memory.
- All operations are element-wise!

Why Is This Base Case Fast?

The whole base case can be implemented within vector registers using a few vector operations.

- 2 AVX registers to store a subcolumn of `A` and its permutation.
- 2 AVX registers to store a subrow of `B` and its permutation.
- 4 AVX registers to store a submatrix of `C`.
- 2 vector permutation operations.
- 4 vector multiplication and addition operations per subrow-subcolumn pair.

4k-by-4k Matrix Multiplication

Version	*Implementation*	*Running time (s)*	*GFLOPS*	*Absolute speedup*	*Relative speedup*	*Fraction of peak*
1	Python	25,552.48	0.005	1	—	0.00%
2	Java	2,372.68	0.058	11	10.8	0.01%
3	C	542.67	0.253	47	4.4	0.03%
4	Parallel loops	69.80	1.969	366	7.8	0.24%
5	Parallel divide-and-conquer	3.80	36.180	6,727	18.4	4.33%
6	+ vectorization	1.10	124.914	23,224	3.5	14.96%
7	+ AVX intrinsics	0.41	337.812	62,806	2.7	40.45%
8	Strassen	0.38	361.177	67,150	1.1	43.24%