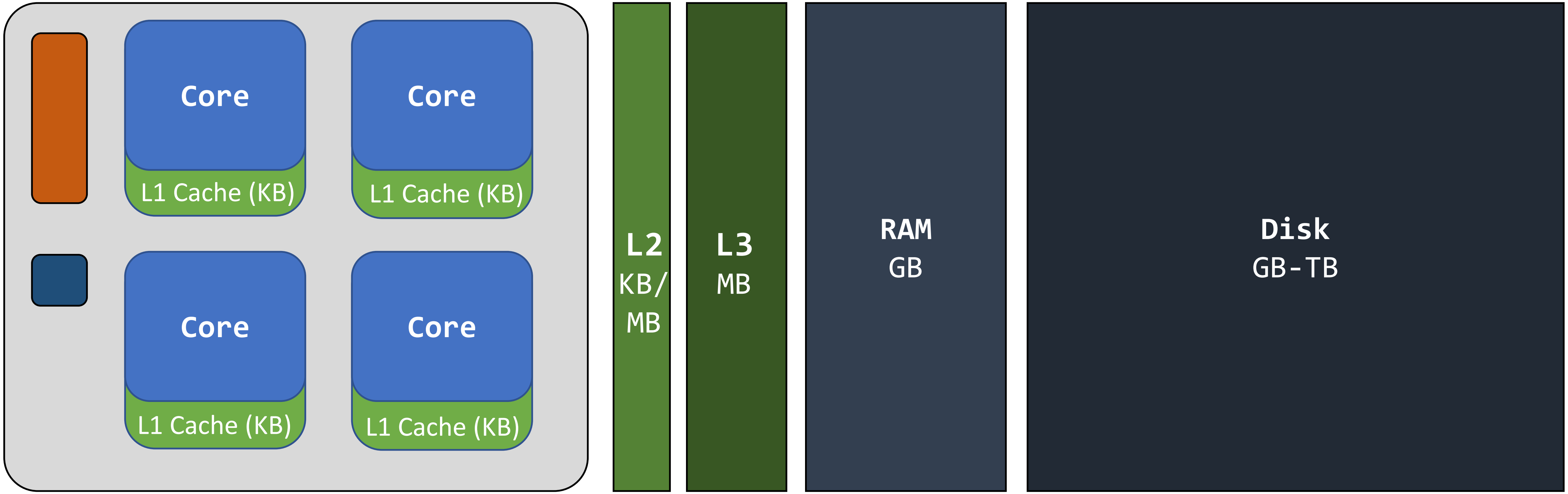
CMOR 421/521:

Case study: linear algebra

M	W	F	
			1
			2
			3
			4
			5
			6
			7
			8
			9
			10
			11
			12
			13
			14
			15

Computer Architecture: Caches

Example Scheme



How do we take advantage of cache in practice?

- Consider summing entries of a matrix
- Should your inner loop be over rows or columns?

```
double val = 0.0
for (int i=0; i<n; ++i){
    for (int j=0; j<n; ++j){
      val += A[i][j]
    }
}
// inner loop over columns
```

```
double val = 0.0
for (int j=0; j<n; ++j){
    for (int i=0; i<n; ++i){
            val += A[i][j]
    }
}
// inner loop over rows
```

How do we take advantage of cache in practice?

- Consider summing entries of a (row major) matrix
- Should your inner loop be over rows or columns?

```
double val = 0.0
for (int i=0; i<n; ++i){
    for (int j=0; j<n; ++j){
      val += A[j + i*n]
    }
}
// inner loop over columns
```

```
double val = 0.0
for (int j=0; j<n; ++j){
    for (int i=0; i<n; ++i){
      val += A[j + i*n]
    }
}
// inner loop over rows
```

How do we take advantage of cache in practice?

- Consider summing entries of a matrix
- Do you loop through rows or columns first?
  - Correct answer: whichever is stored contiguously. We used row major storage.

How do we take advantage of cache in practice?

- Consider summing entries of a matrix
- Do you loop through rows or columns first?
    - Correct answer: whichever is stored contiguously.

How much faster do you expect it to be, and why?

Demo

Matrix summation demo

- L1 cache: 100x faster than RAM
  - Typically 16KB - 128 KB per core
- L2 cache: 25x faster than RAM
  - typically 128KB – 1MB per core

Observations:

- L1 cache is typically 4x faster than L2 cache
- An nxn matrix fits into L1 cache until n ~ 128

https://www.makeuseof.com/tag/what-is-cpu-cache/

How do we take advantage of cache in practice?

- Consider matrix-vector multiplication
- What loop ordering/matrix format should you use?

```
double val = 0.0
for (int i=0; i<n; ++i){
  for (int j=0; j<n; ++j){
    val += A[i][j] * u[j]
  }
}
```

```
double val = 0.0
for (int j=0; j<n; ++j){
  for (int i=0; i<n; ++i){
     val += A[i][j] * u[j]
  }
}
```

Theoretical performance models

- Assume just 2 levels in memory hierarchy: fast and slow
- All data initially in slow memory
  - m = number of “words” of memory moved between fast and slow memory
  - t_m = time per slow memory operation (inverse bandwidth in best case)
  - f = number of arithmetic operations
  - t_f = time per arithmetic operation (which is << t_m)

Theoretical performance models

- Assume just 2 levels in memory hierarchy: fast and slow
- All data initially in slow memory
  - m = number of "words" of memory moved between fast and slow memory
  - t_m = time per slow memory operation (inverse bandwidth in best case)
  - f = number of arithmetic operations
  - t_f = time per arithmetic operation (which is << t_m)
  - CI (Computational intensity) = f / m, the average number of flops performed per slow memory access.

Theoretical performance models

- Assume just 2 levels in memory hierarchy: fast and slow
- All data initially in slow memory
    - m = number of “words” of memory moved between fast and slow memory
    - t_m = time per slow memory operation (inverse bandwidth in best case)
    - f = number of arithmetic operations
    - t_f = time per arithmetic operation (which is << t_m)
    - CI (Computational intensity) = f / m, the average number of flops performed per slow memory access.
- Model of runtime
    - $(f * t_f) + (m * t_m) = f * t_f * (1 + t_m/t_f \ \ * 1/CI)$
- Larger CI means time closer to the minimum time: $f * t_f$

Matrix-vector multiplication

```
for i = 1:n
    for j = 1:n
      y(i) = y(i) + A(i,j)*x(j)
```

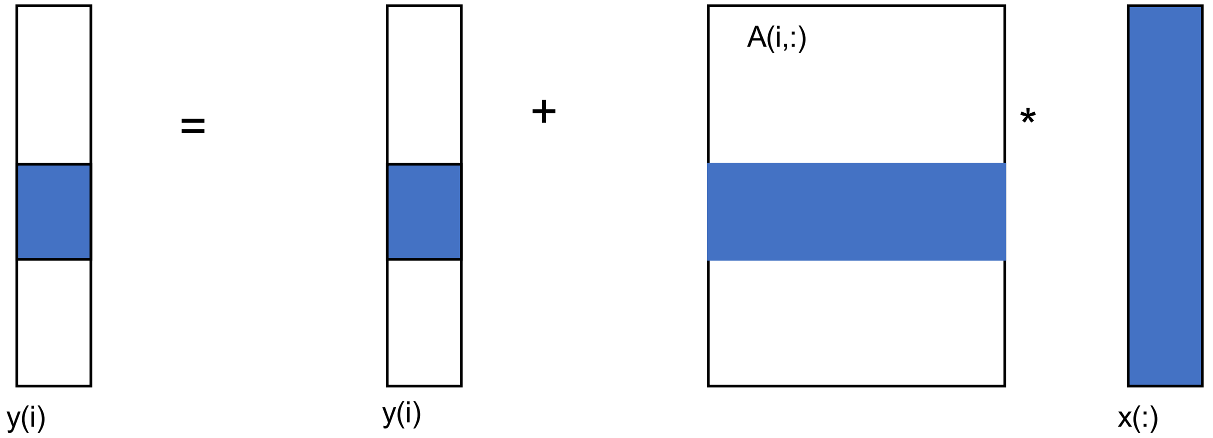


Image from Demmel

Matrix-vector multiplication

```
{read x(1:n) into fast memory}
{read y(1:n) into fast memory}
for i = 1:n
     {read row i of A into fast memory}
     for j = 1:n
        y(i) = y(i) + A(i,j)*x(j)
{write y(1:n) back to slow memory}
```

What is the CI (Computational Intensity)?

Matrix-vector multiplication

```
{read x(1:n) into fast memory} = n
{read y(1:n) into fast memory} = n
for i = 1:n
     {read row i of A into fast memory} = n^2
     for j = 1:n
        y(i) = y(i) + A(i,j)*x(j) 2 flops n^2 times
{write y(1:n) back to slow memory} = n
```

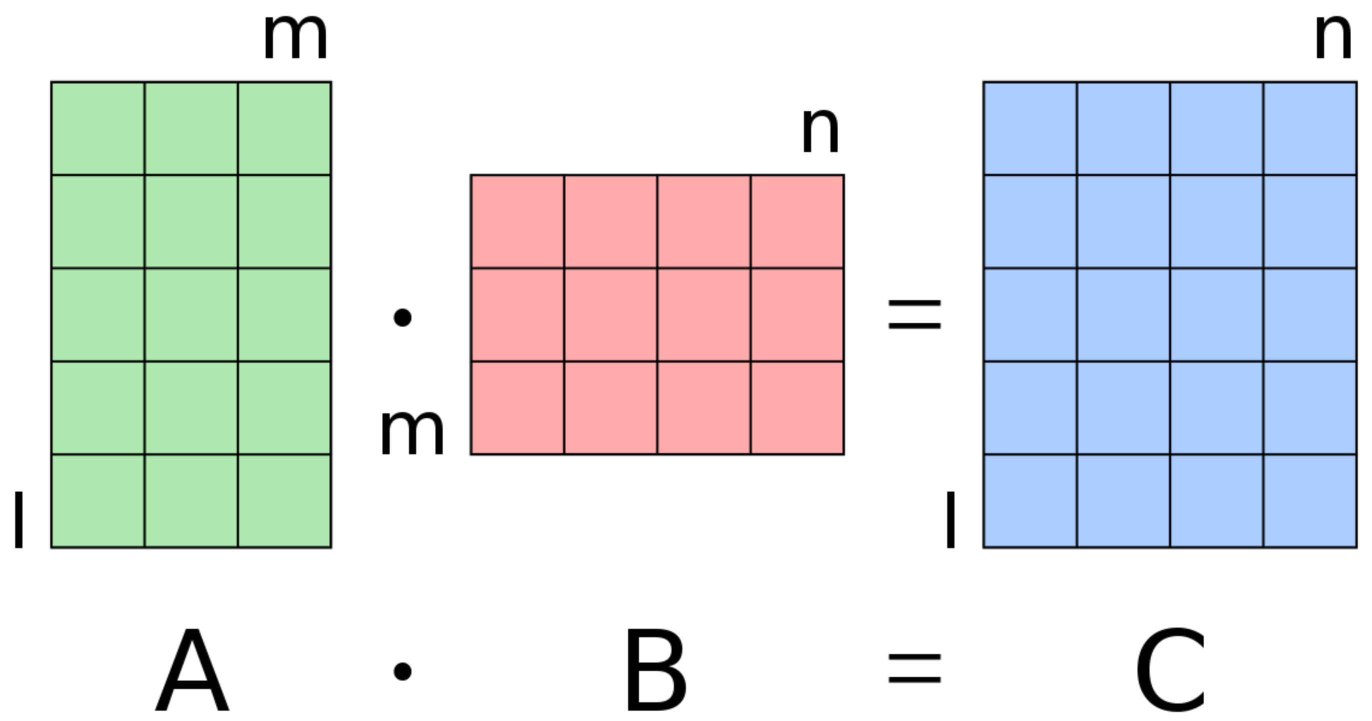
3*n + n^2 slow memory accesses, 2n^2 flops. CI ~ 2

Simplifying assumptions

- Constant “peak” computation rate
- Fast memory is large enough to hold three vectors
- The cost of a fast memory access is ~0
  - True for register memory, but not for cache (even L1)
- Ignores memory latency completely

Could simplify more by ignoring memory load/stores on x and y; reading the larger matrix dominates.

Matrix-matrix multiplication



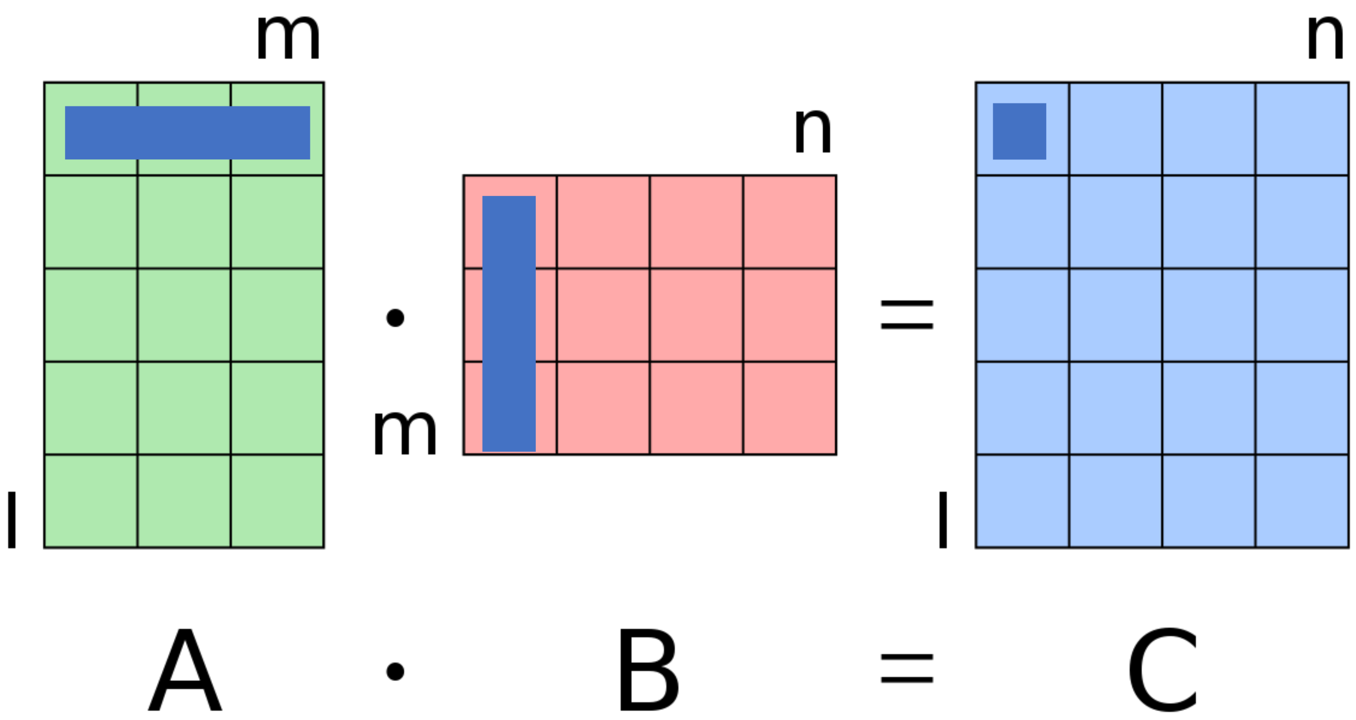
From Wikipedia

Matrix-matrix multiplication

- Why do we expect this to do better?
- Matrix-vector multiplication: O(n^2) memory accesses, O(n^2) flops = constant CI.
- Matrix-matrix multiplication: O(n^2) memory accesses, O(n^3) flops
  - O(n^3) = O(n^2) dot products of O(n) each
  - Potentially O(n) CI?

Naïve implementation

$$C_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}$$



Matrix-matrix multiplication

C = C + A * B, assume matrices are nxn (square).

```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

What is the CI of this implementation?

Matrix-matrix multiplication

Assume matrix is nxn (square).

```
for i = 1 to n
  {read row i of A into fast memory}
   for j = 1 to n
       {read C(i,j) into fast memory}
       {read column j of B into fast memory}
       for k = 1 to n
           C(i,j) = C(i,j) + A(i,k) * B(k,j)
       {write C(i,j) back to slow memory}
```

Matrix-matrix multiplication

Assume matrix is nxn (square).

```
for i = 1 to n
  {read row i of A into fast memory} (n^2)
   for j = 1 to n
       {read C(i,j) into fast memory} (n^2)
       {read column j of B into fast memory} (n^3)
       for k = 1 to n
           C(i,j) = C(i,j) + A(i,k) * B(k,j) (2*n^3)
       {write C(i,j) back to slow memory} (n^2)
```

Matrix-matrix multiplication

Assume matrix is nxn (square).

3n^2 + n^3 memory accesses vs 2n^3 flops

→ CI ~ 2 again!

This is just as bad as matrix-vector multiplication.

Matrix-matrix multiplication

Bottleneck is the repeated memory loads of B(:,j).
How to fix? Use that fast memory is "free" (in theory).

```
for i = 1 to n
  {read row i of A into fast memory} (n^2)
   for j = 1 to n
       {read C(i,j) into fast memory} (n^2)
       {read column j of B into fast memory} (n^3)
       for k = 1 to n
           C(i,j) = C(i,j) + A(i,k) * B(k,j) (2*n^3)
       {write C(i,j) back to slow memory} (n^2)
```

Blocking/Tiling

- Divide each nxn matrix into N bxb sub-blocks
- Assume for simplicity n = N * b
- Assume bxb sub-blocks fit into fast memory

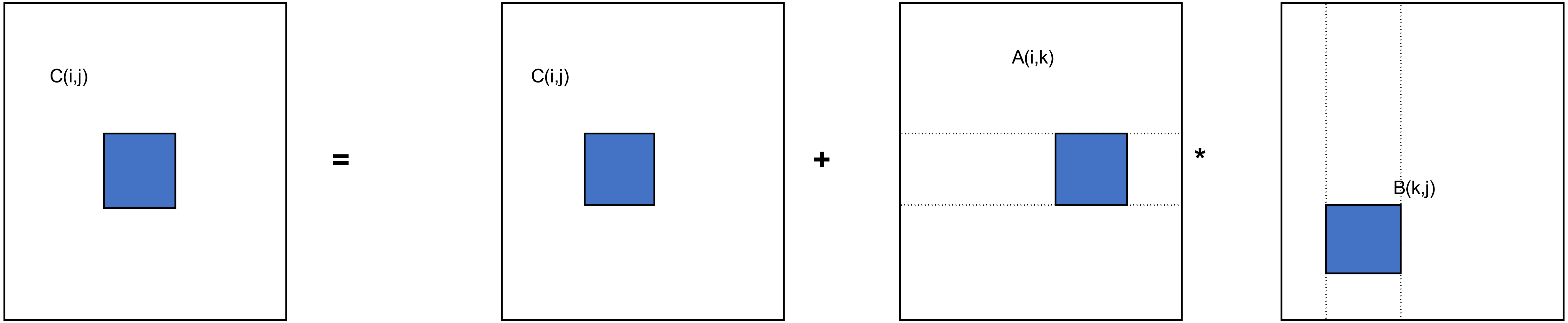


Image from Demmel

Blocking/Tiling

```
for i = 1 to N
      for j = 1 to N
            {read block C(i,j) into fast memory}
            for k = 1 to N
                  {read block A(i,k) into fast memory}
                  {read block B(k,j) into fast memory}
                  // do a matrix multiply on blocks
                  // this is really 3 nested loops!
                  C(i,j) = C(i,j) + A(i,k) * B(k,j)

{write block C(i,j) back to slow memory}
```


Image from Demmel

Blocking/Tiling

```
for i = 1 to N
      for j = 1 to N
            {read block C(i,j) into fast memory} = n^2
            for k = 1 to N
                  {read block A(i,k) into fast memory} = N^3 * b^2
                  {read block B(k,j) into fast memory} = N^3 * b^2
                  // do a matrix multiply on blocks
                  // Note: this is really 3 nested loops
                  C(i,j) = C(i,j) + A(i,k) * B(k,j) = n^3

Use that b = n / N, so N^3 * b^2 = N * n^2
CI = O(n^3) flops / O(N * n^2) memory = O(n / N) = O(b)
```


Image from Demmel

Blocking/Tiling

- Key assumption: b is chosen so that bxb blocks of A, B, C all fit into fast memory!
  - b = sqrt((fast memory size) / 3)
  - Slow memory costs are O(n^2 / b) = O(n^2 / sqrt(fast memory size))
- Need CI >= (t_m/t_f) to get half of peak performance
  - Implies fast memory size should be 3(t_m/t_f)^2, which is close
  - In practice, blocking/tiling doesn’t achieve O(b) CI without additional optimizations
- Code gets uglier if
  - dimensions of A, B, C are not perfectly divisible by block size b
  - you optimize by exploiting multiple levels of cache

Alternatives to blocking/tiling

- Blocking/tiling is known as a “cache-aware” algorithm (you have to know what size your cache is to implement this)
- Cache-oblivious “recursive matrix-multiplication” algorithms

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

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

Image from Demmel

Pseudocode for recursive matmul

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

```
function RMM(C, A, B, n)
if (n == 1){
    C = A * B;
} else {
    C00 = RMM(A00, B00, n / 2) + RMM(A01, B10, n / 2);
    C01 = RMM(A00, B01, n / 2) + RMM(A01, B11, n / 2);
    C10 = RMM(A10, B00, n / 2) + RMM(A11, B10, n / 2);
    C11 = RMM(A10, B01, n / 2) + RMM(A11, B11, n / 2);
}
```

Performance estimate for recursive matmul

- Floating point operations for n > 1
  - 8 * (cost of n/2 matmul) + 4 * (n/2) adds per level
  - Can show this gives you 2n^3 – n^2 + … operations
- Memory movement: cache is not explicitly managed, assume arrays move into fast memory until it's full
  - 8 * (memory movement for n/2 matmul) + 4 * 3 * (n/2)^2 (subblocks) per level if A, B, C don't fit in fast memory
  - Can show that the memory cost over all recursive levels is O(n^3 / sqrt(size of fast memory) + …), similar to blocking/tiling.

More related to blocking/tiling

- In general algorithms require tuning block sizes, parameters.
  - Even cache-oblivious algorithms like RMM don’t recurse to n=1, they go until matrices are small enough then execute a “microkernel”.
- Alternative data layouts
  - Copy data into a different format prior to operating on it
  - Block versions of row major, column major
  - Recursive block ordering of matrix entries (often uses space-filling curves like Morton ordering or Z-ordering)
- Strassen’s algorithm: recursive algorithm for matrix-matrix multiplication, which uses a clever rewriting of 2x2 matrix multiplication to achieve O(n^2.8074) vs O(n^3) complexity

Strassen's algorithm

- Recursive matrix multiplication with a twist

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix}$$

$$\begin{aligned}
M_1 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}); \\
M_2 &= (A_{21} + A_{22}) \times B_{11}; \\
M_3 &= A_{11} \times (B_{12} - B_{22}); \\
M_4 &= A_{22} \times (B_{21} - B_{11}); \\
M_5 &= (A_{11} + A_{12}) \times B_{22}; \\
M_6 &= (A_{21} - A_{11}) \times (B_{11} + B_{12}); \\
M_7 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}),
\end{aligned}$$

Image from Wikipedia: Strassen's algorithm

Strassen's algorithm

- Recursive matrix multiplication with a twist
- Uses that matrix-matrix multiplication dominates the memory access costs. Strassen requires only 7 multiplications

$$
\begin{aligned}
M_1 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}); \\
M_2 &= (A_{21} + A_{22}) \times B_{11}; \\
M_3 &= A_{11} \times (B_{12} - B_{22}); \\
M_4 &= A_{22} \times (B_{21} - B_{11}); \\
M_5 &= (A_{11} + A_{12}) \times B_{22}; \\
M_6 &= (A_{21} - A_{11}) \times (B_{11} + B_{12}); \\
M_7 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}),
\end{aligned}
$$

Image from Wikipedia: Strassen's algorithm

Do I have to code this myself?

- Linear algebra libraries are probably the most successful example of HPC software on CPU architectures.
  - LINPACK benchmarks used to create the Top500 list

- BLAS (Basic Linear Algebra Subprograms) specifies interfaces and operations. A vendor then provides optimized implementations of BLAS for ***their specific architecture***.
  - Level 1 BLAS: vector operations (dot, scalar*x + y). O(n), low CI
  - Level 2 BLAS: matrix-vector operations. O(n^2), still low-ish CI
  - Level 3 BLAS: matrix-matrix operations. O(n^3), high CI

- Example of recursive algorithms in practice: LibFLAME

BLAS 1 and 2 vs BLAS 3 performance

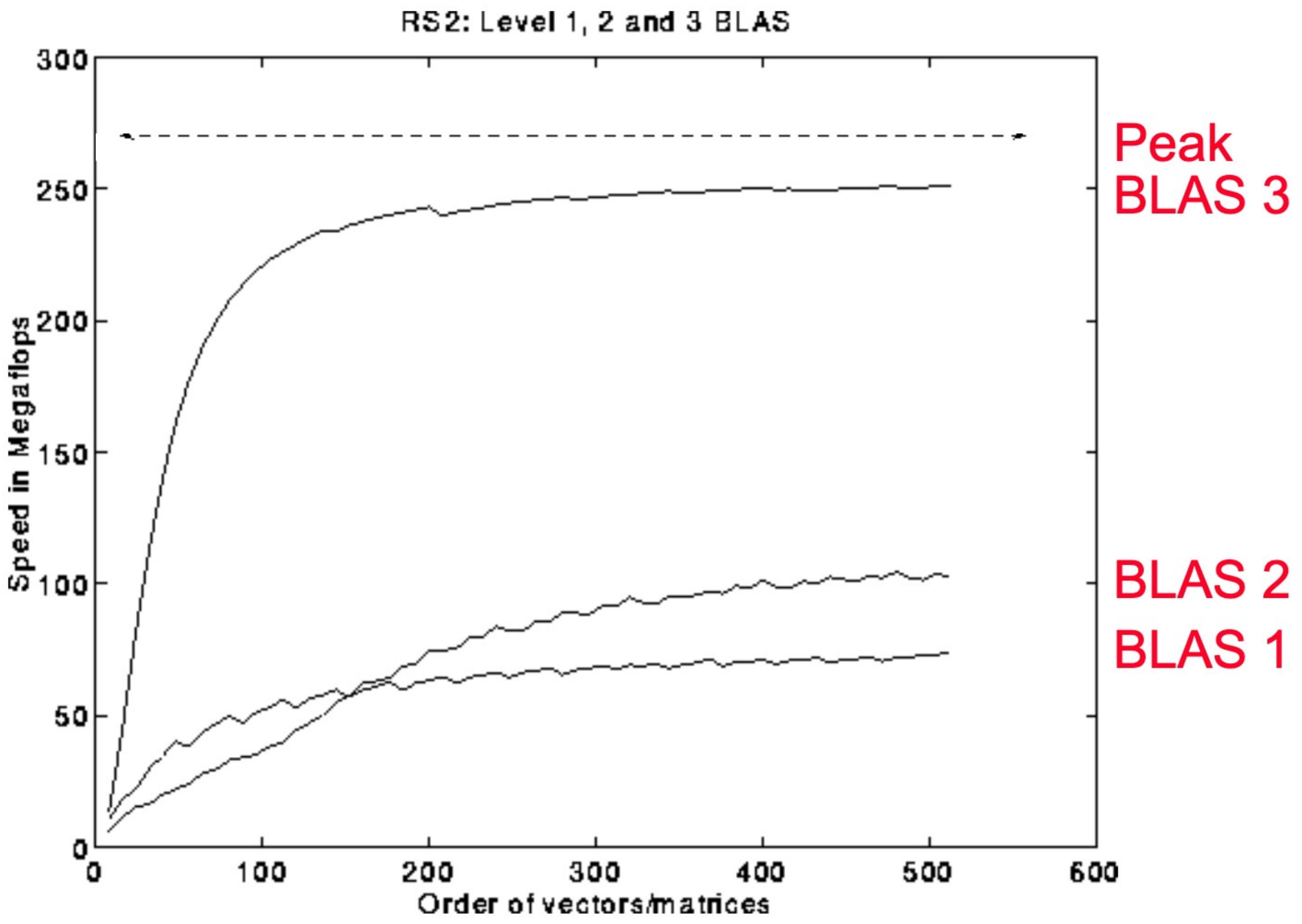


Image from Tao Yang