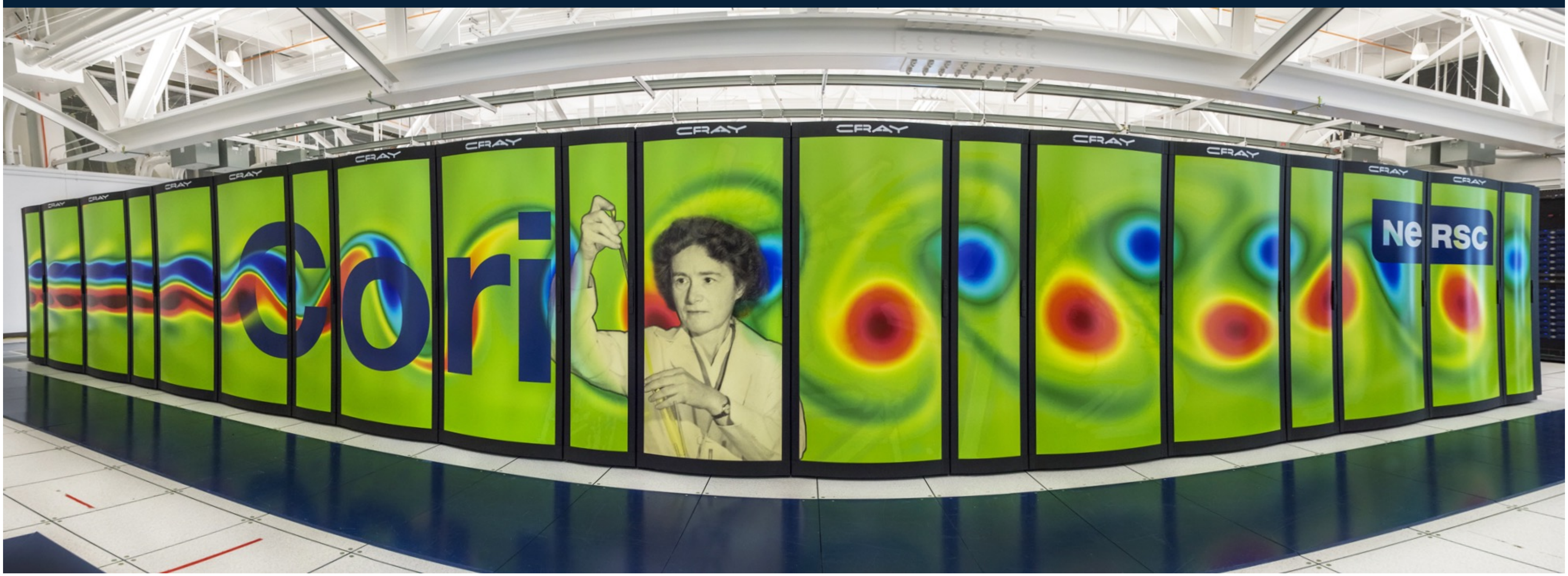
Applications of Parallel Computers

Matrix Multiplication and the Roofline Model

https://sites.google.com/lbl.gov/cs267-spr2022



Review

A Simple Model of Memory

- Assume just 2 levels in the hierarchy, fast and slow
- All data initially in slow memory
  - m = number of memory elements (words) 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 << t_m
  - $\boxed{CI = f / m}$ average number of flops per slow memory access
- Minimum possible time = $f * t_f$ when all data in fast memory
- Actual time
  - $f * t_f + m * t_m = f * t_f * (1 \boxed{+ t_m/t_f} * 1/CI)$
- Larger CI means time closer to minimum $f * t_f$

Computational Intensity (CI): **Key to algorithm efficiency**

Machine Balance: **Key to machine efficiency**

Naïve Matrix Multiply

{implements C = C + A*B}

```
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}
```

$f = 2n^3$ arithmetic ops. $m = n^3 + 3n^2$ slow memory

n^2 to read each row of A once

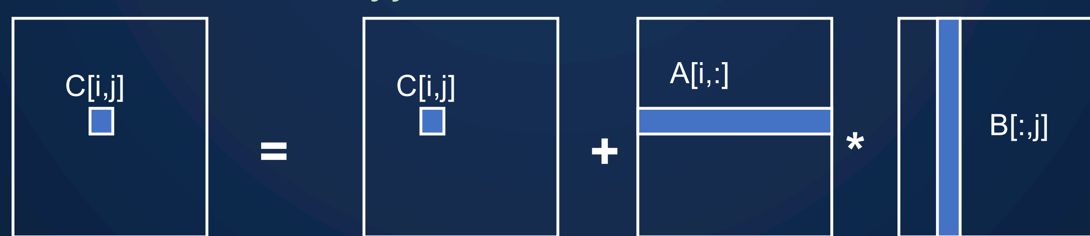
$2n^2$ to read and write each element of C once

n^3 to read each column of B n times

So the computational intensity is:

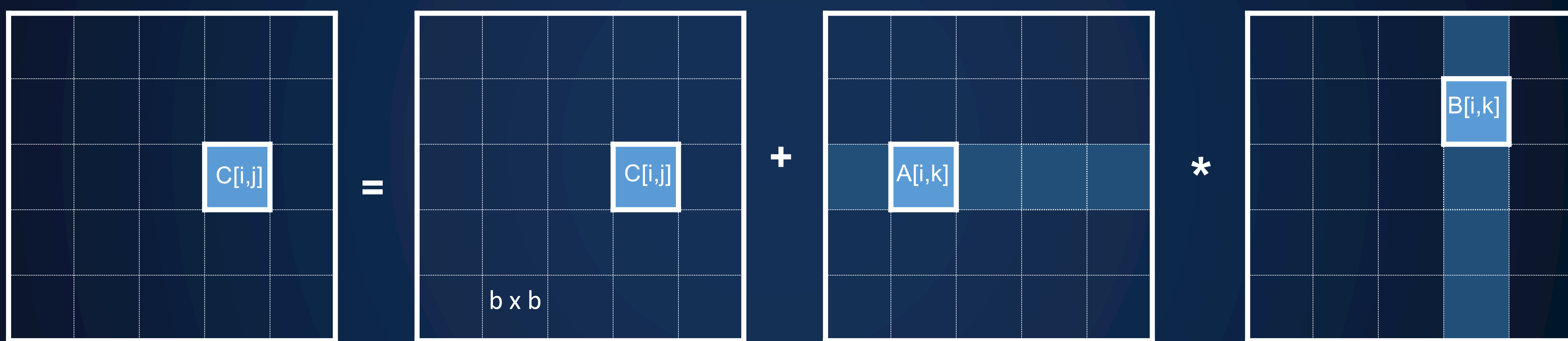
$CI = f / m = 2n^3 / [n^3 + 3n^2] \;\sim= 2$

No better than matrix-vector!



Blocked (Tiled) Matrix Multiply

Consider A,B,C to be N-by-N matrices of b-by-b subblocks where b=n / N is called the block size



All of this works if the blocks or matrices are not square

n elements

N blocks

Each block is bxb
Assume 3 bxb blocks fit in cache

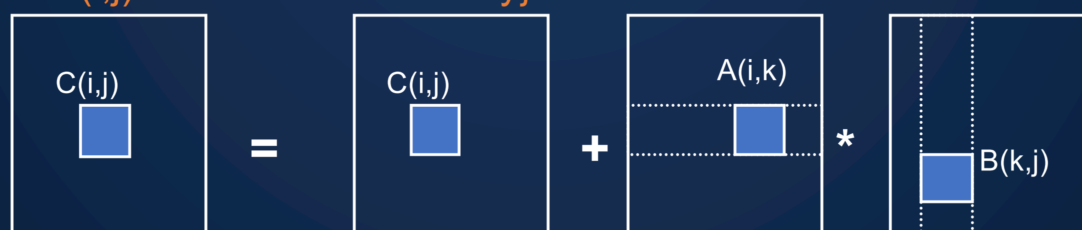
Blocked (Tiled) Matrix Multiply

Consider A,B,C to be N-by-N matrices of b-by-b subblocks where b=n / N is called the block size

```
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}
          C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
      {write block C(i,j) back to slow memory}
```

3 nested loops inside

block size = loop bounds



Tiling for registers or caches

Blocked [Tiled] Matrix Multiply

Consider A,B,C to be N-by-N matrices of b-by-b subblocks where b=n / N is called the **block size**

    for i = 1 to N
      for j = 1 to N

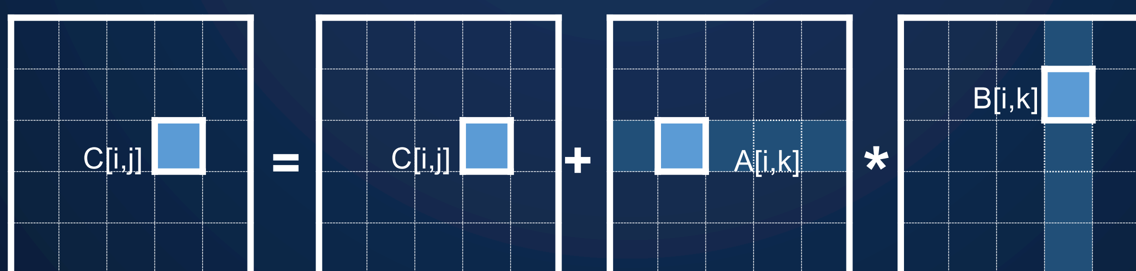
          for k = 1 to N

              C[i,j] = C[i,j] + A[i,k] * B[k,j] {do a matrix multiply on blocks}

nxn elements

NxN blocks

Each block is bxb



Blocked [Tiled] Matrix Multiply

Consider A,B,C to be N-by-N matrices of b-by-b subblocks where b=n / N is called the block size

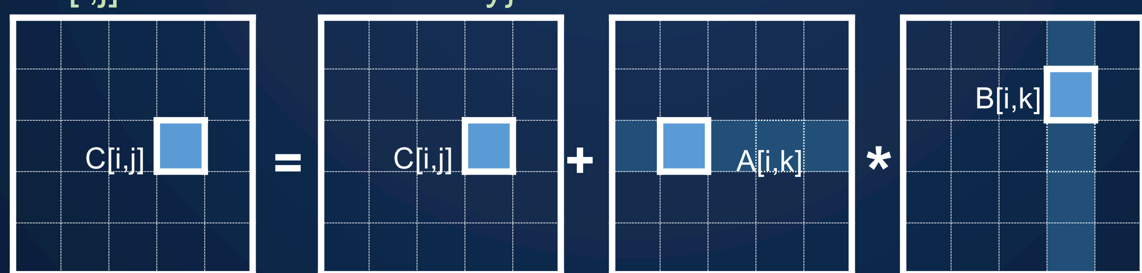
```
for i = 1 to N
   for j = 1 to N
      {read block C[i,j] into fast memory}
      for k = 1 to N
         C[i,j] = C[i,j] + A[i,k] * B[k,j] {do a matrix multiply on blocks}
      {write C[i,j] back to slow memory}
```

$2n^2$ to read and write each block of C once ($2N^2 * b^2 = 2n^2$)

nxn elements

NxN blocks

Each block is bxb



Blocked [Tiled] Matrix Multiply

Consider A,B,C to be N-by-N matrices of b-by-b subblocks where b=n / N is called the block size

```
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}

         C[i,j] = C[i,j] + A[i,k] * B[k,j] {do a matrix multiply on blocks}
     {write C[i,j] back to slow memory}
```

$2n^2$ to read and write each block of C once

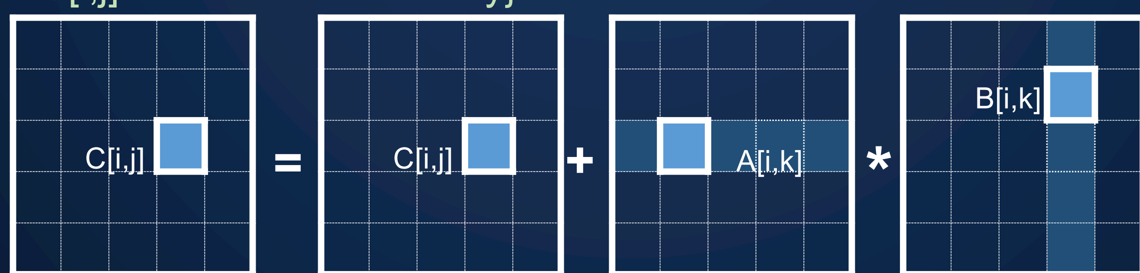
$N*n^2$ to read each block of A N^3 times

$(N^3 * b^2 = N^3 * (n/N)^2)$

nxn elements

NxN blocks

Each block is bxb



Blocked [Tiled] Matrix Multiply

Consider A,B,C to be N-by-N matrices of b-by-b subblocks where b=n / N is called the block size

```
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}
         C[i,j] = C[i,j] + A[i,k] * B[k,j] {do a matrix multiply on blocks}
      {write C[i,j] back to slow memory}
```

$2n^2$ to read and write each block of C once

$N*n^2$ to read each block of A N^3 times

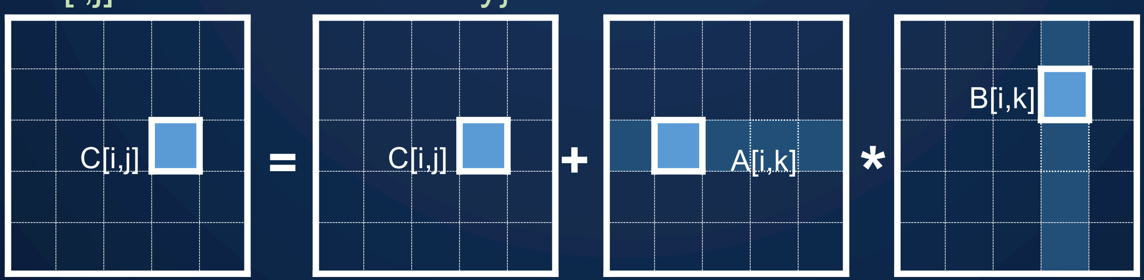
$N*n^2$ to read each block of B N^3 times

$(N^3 * b^2 = N^3 * (n/N)^2)$

nxn elements

NxN blocks

Each block is bxb



Blocked [Tiled] Matrix Multiply

Consider A,B,C to be N-by-N matrices of b-by-b subblocks where b=n / N is called the block size

```
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}
         C[i,j] = C[i,j] + A[i,k] * B[k,j] {do a matrix multiply on blocks}
      {write C[i,j] back to slow memory}
```

$2n^2$ to read and write each block of C once

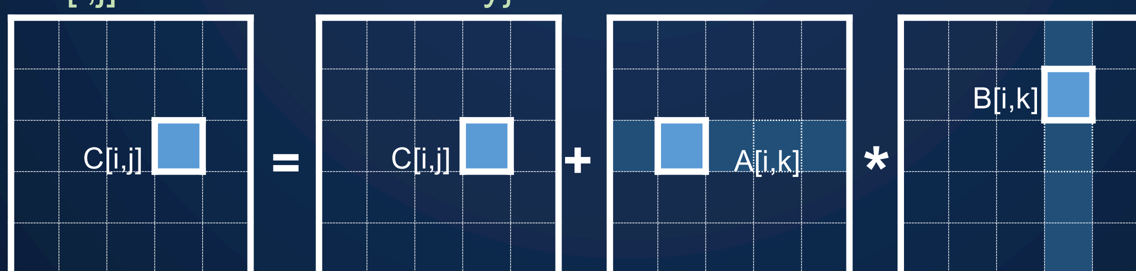
$N*n^2$ to read each block of A N^3 times

$N*n^2$ to read each block of B N^3 times

nxn elements

NxN blocks

Each block is bxb



Blocked [Tiled] Matrix Multiply

Consider A,B,C to be N-by-N matrices of b-by-b subblocks where $b=n / N$ is called the block size

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}

      C[i,j] = C[i,j] + A[i,k] * B[k,j] {do a matrix multiply on blocks}

    {write C[i,j] back to slow memory}

$2n^2$ to read and write each block of C once

$N*n^2$ to read each block of A N^3 times

$N*n^2$ to read each block of B N^3 times

#words moved = $m = 2n^2 + N^* n^2 + N^* n^2 = (2N + 2) * n^2$

Computational Intensity = $CI = f / m = 2n^3 / ((2N + 2) * n^2)$

$\approx n / N = b$ for large n

Blocked [Tiled] Matrix Multiply

Computational Intensity (CI) = b for large n

How large can we make b?   Assume our fast memory has size M:

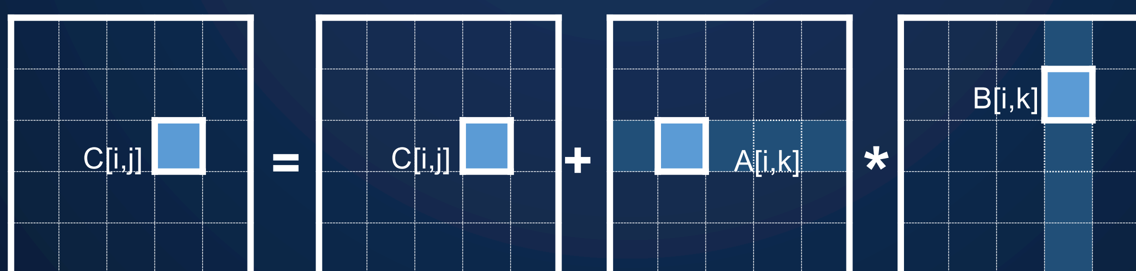
$$b \le \sqrt{M/3}$$

Since M must hold 3 bxb blocks

nxn elements

NxN blocks

Each block is bxb



Is there a more elegant approach?

Is there a more elegant approach?

Yes, but not quite as fast

Recursive Matrix Multiplication

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

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

- True when each bock is a 1x1 or n/2 x n/2
- For simplicity: square matrices with $n = 2^m$
  - Extends to general rectangular case

Recursive Matrix Multiplication

```
Define C = RMM (A, B, n)
    if (n=1) {
       C = A * B;
    }




    return C
```

Recursive Matrix Multiplication

```
Define C = RMM (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 (A11 , B01 , n/2) + RMM (A11 , B11 , n/2)
   }
   return C
```

Recursive Matrix Multiplication

Define C = RMM (A, B, n)
  if (n==1) { C = A * B ; } else
  { C_{00} = RMM (A_{00} , B_{00} , n/2) + RMM (A_{01} , B_{10} , n/2)
  C_{01} = RMM (A_{00} , B_{01} , n/2) + RMM (A_{01} , B_{11} , n/2)
  C_{10} = RMM (A_{10} , B_{00} , n/2) + RMM (A_{11} , B_{10} , n/2)
  C_{11} = RMM (A_{11} , B_{01} , n/2) + RMM (A_{11} , B_{11} , n/2) }
  return C

How many flops (f) and memory moves (m)?

Recursive Matrix Multiplication

Define C = RMM (A, B, n)
  if (n==1) { C = A * B_0; } else
    { C_{00} = RMM (A_{00} , B_{00} , n/2) + RMM (A_{01} , B_{10} , n/2)
      C_{01} = RMM (A_{00} , B_{01} , n/2) + RMM (A_{01} , B_{11} , n/2)
      C_{10} = RMM (A_{10} , B_{00} , n/2) + RMM (A_{11} , B_{10} , n/2)
      C_{11} = RMM (A_{11} , B_{01} , n/2) + RMM (A_{11} , B_{11} , n/2) }
  return C

How many flops (f) and memory moves (m)?

Arith(n) = # arithmetic operations in RMM(. , . , n)

Recursive Matrix Multiplication

```
Define C = RMM (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 (A11 , B01 , n/2) + RMM (A11 , B11 , n/2)  }
   return C
```

Arith(n) = # arithmetic operations in RMM(. , . , n)

$= 8 \cdot \text{Arith}(n/2) + 4(n/2)^2$ if $n > 1$, else 1

Recursive Matrix Multiplication

Define C = RMM (A, B, n)
  if (n==1) { C = A * B ; } else
   { C_{00} = RMM (A_{00} , B_{00} , n/2) + RMM (A_{01} , B_{10} , n/2)
    C_{01} = RMM (A_{00} , B_{01} , n/2) + RMM (A_{01} , B_{11} , n/2)
    C_{10} = RMM (A_{10} , B_{00} , n/2) + RMM (A_{11} , B_{10} , n/2)
    C_{11} = RMM (A_{11} , B_{01} , n/2) + RMM (A_{11} , B_{11} , n/2) }
  return C

Arith(n) = # arithmetic operations in RMM(. , . , n)
  $= 8 \cdot \text{Arith}(n/2) + 4(n/2)^2$ if $n > 1$, else 1
  $= 2n^3 - n^2$ … ~same operations as usual, in different order

Recursive Matrix Multiplication

```
Define C = RMM (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 (A11 , B01 , n/2) + RMM (A11 , B11 , n/2)  }
   return C
```

Arith(n) = # arithmetic operations in RMM(. , . , n)

$= 8 \cdot \text{Arith}(n/2) + 4(n/2)^2$ if $n > 1$, else 1

$\sim 2n^3$ this is our f = # flops

Recursive Matrix Multiplication

Define C = RMM (A, B, n)
  if (n==1) { C = A * B ; } else
    { C_{00} = RMM (A_{00} , B_{00} , n/2) + RMM (A_{01} , B_{10} , n/2)
      C_{01} = RMM (A_{00} , B_{01} , n/2) + RMM (A_{01} , B_{11} , n/2)
      C_{10} = RMM (A_{10} , B_{00} , n/2) + RMM (A_{11} , B_{10} , n/2)
      C_{11} = RMM (A_{11} , B_{01} , n/2) + RMM (A_{11} , B_{11} , n/2) }
  return C

f = { Arith(n) = # arithmetic operations in RMM(. , . , n)
$= 8 \cdot \text{Arith}(n/2) + 4(n/2)^2$ if $n > 1$, else 1
$\sim 2n^3$ }

What is m, data moved?

Recursive Matrix Multiplication

```
Define C = RMM (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 (A11 , B01 , n/2) + RMM (A11 , B11 , n/2)  }
   return C
```

f = Arith(n) = # arithmetic operations in RMM(. , . , n)

$= 8 \cdot \text{Arith}(n/2) + 4(n/2)^2$ if $n > 1$, else 1

$\sim 2n^3$

m = W(n) = # words moved between fast, slow memory by RMM(. , . , n)

Recursive Matrix Multiplication

```
Define C = RMM (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 (A11 , B01 , n/2) + RMM (A11 , B11 , n/2)  }
   return C
```

f = Arith(n) = # arithmetic operations in RMM(. , . , n)

$= 8 \cdot \text{Arith}(n/2) + 4(n/2)^2$ if $n > 1$, else 1

$\sim 2n^3$ this is our f = # flops

m = W(n) = # words moved between fast, slow memory by RMM(. , . , n)

$= 8 \cdot W(n/2) + 4 \cdot 3(n/2)^2$ if $3n^2 > M_{fast}$, else $3n^2$

4 lines of code
3 matrices per line

Recursive Matrix Multiplication

Define C = RMM (A, B, n)
    if (n==1) { C = A * B ; } else
      { C_{00} = RMM (A_{00} , B_{00} , n/2) + RMM (A_{01} , B_{10} , n/2)
        C_{01} = RMM (A_{00} , B_{01} , n/2) + RMM (A_{01} , B_{11} , n/2)
        C_{10} = RMM (A_{10} , B_{00} , n/2) + RMM (A_{11} , B_{10} , n/2)
        C_{11} = RMM (A_{11} , B_{01} , n/2) + RMM (A_{11} , B_{11} , n/2) }
    return C

f = { Arith(n) = # arithmetic operations in RMM(. , . , n)
$= 8 \cdot \text{Arith}(n/2) + 4(n/2)^2$ if $n > 1$, else 1
$\sim 2n^3$ this is our f = # flops

m = { W(n) = # words moved between fast, slow memory by RMM(. , . , n)
$= 8 \cdot W(n/2) + 4 \cdot 3(n/2)^2$ if $3n^2 > M_{fast}$, else $3n^2$
$= O(n^3 / (M_{fast})^{1/2} + n^2)$ … same as blocked matmul

Recursive Matrix Multiplication

Define C = RMM (A, B, n)
    if (n==1) { C = A * B ; } else
      { C_{00} = RMM (A_{00} , B_{00} , n/2) + RMM (A_{01} , B_{10} , n/2)
        C_{01} = RMM (A_{00} , B_{01} , n/2) + RMM (A_{01} , B_{11} , n/2)
        C_{10} = RMM (A_{10} , B_{00} , n/2) + RMM (A_{11} , B_{10} , n/2)
        C_{11} = RMM (A_{11} , B_{01} , n/2) + RMM (A_{11} , B_{11} , n/2) }
    return C

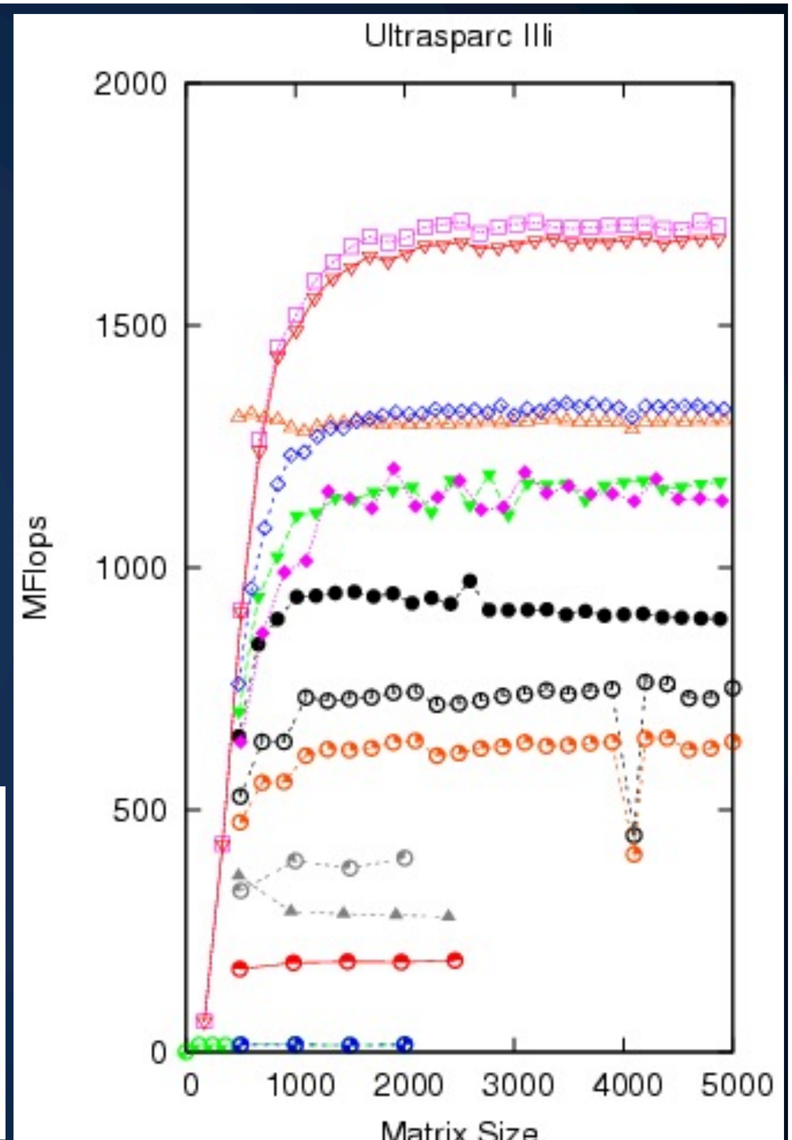
f = { Arith(n) = # arithmetic operations in RMM(. , . , n)
$= 8 \cdot \text{Arith}(n/2) + 4(n/2)^2$ if $n > 1$, else 1
$\sim 2n^3$ this is our f = # flops }

m = { W(n) = # words moved between fast, slow memory by RMM(. , . , n)
$= 8 \cdot W(n/2) + 4 \cdot 3(n/2)^2$ if $3n^2 > M_{fast}$, else $3n^2$
$= O(n^3 / (M_{fast})^{1/2} + n^2)$ … same as blocked matmul }

Don't need to know M_{fast} for this to work!

Cache Oblivious

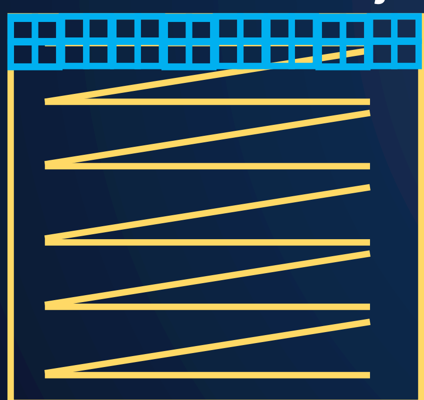
- In practice, cut off recursion well before 1x1
  - Call “micro-kernel” on small blocks
- Pingali et al report about 2/3 of peak
  - Recursive + optimized micro-kernel
  - See: https://www.slideserve.com/lazar/a-comparison-of-cache-conscious-and-cache-oblivious-programs
  - Atlas with ‘unleashed’ autotuning close to vendor



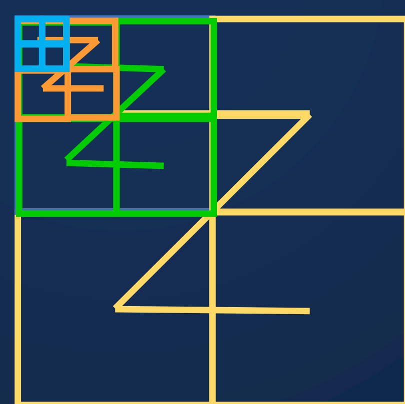
Alternate Data Layouts

- May also use blocked or recursive layouts
- Several possible recursive layouts, depending on the order of the sub-blocks
- Copy optimization may be used to move

Blocked-Row Major



Z-Morton order (recursive)



- works well for any cache size
- but index calculations to find A[i,j] are expensive
- May switch to col/row major for small sizes

Theory: Communication lower bound

Theorem (Hong & Kung, 1981):

Any reorganization of matmul (using only commutativity and associativity) has computational intensity $q = O((M_{fast})^{1/2})$, so

$\#$words moved between fast/slow memory = $\Omega\ (n^3 / (M_{fast})^{1/2})$

- Extensions, both lower bounds and (some) optimal algorithms (later in course)
  - Parallel matrix multiply, optimize latency as well as bandwidth
  - Rest of linear algebra (Gaussian elimination, least squares, tensors …)
  - Nested loops accessing arrays (eg All-Pairs-Shortest-Paths, N-body, …)
  - Open problems:
    - Small loop bounds (eg matrix-vector vs matrix-matrix multiply)
    - Dependencies, i.e. when only some reorganizations are correct

Strassen's Matrix Multiply

- The traditional algorithm (with or without tiling) has $O(n^3)$ flops
- Strassen discovered an algorithm with asymptotically lower flops
  - $O(n^{2.81})$
- Consider a 2x2 matrix multiply, normally takes 8 multiplies, 4 adds
  - Strassen does it with 7 multiplies and 18 adds

Let $M = \begin{pmatrix} m11 & m12 \\ m21 & m22 \end{pmatrix} = \begin{pmatrix} a11 & a12 \\ a21 & a22 \end{pmatrix} \begin{pmatrix} b11 & b12 \\ b21 & b22 \end{pmatrix}$

Let p1 = (a12 - a22) * (b21 + b22)

p2 = (a11 + a22) * (b11 + b22)

p3 = (a11 - a21) * (b11 + b12)

p4 = (a11 + a12) * b22

p5 = a11 * (b12 - b22)

p6 = a22 * (b21 - b11)

p7 = (a21 + a22) * b11

Then m11 = p1 + p2 - p4 + p6

m12 = p4 + p5

m21 = p6 + p7

m22 = p2 - p3 + p5 - p7

Extends to nxn by divide&conquer

Strassen (continued)

T(n) = Cost of multiplying nxn matrices
= $7*T(n/2) + 18*(n/2)^2$
= $O(n^{\log_2 7})$
= $O(n^{2.81})$

- Asymptotically faster
  - Several times faster for large n in practice
  - Cross-over depends on machine
  - "Tuning Strassen's Matrix Multiplication for Memory Efficiency", M. S. Thottethodi, S. Chatterjee, and A. Lebeck, in Proceedings of Supercomputing '98

- Possible to extend communication lower bound to Strassen
  - #words moved between fast and slow memory
    $= \Omega(n^{\log_2 7} / M^{(\log_2 7)/2 - 1}) \sim \Omega(n^{2.81} / M^{0.4})$
    (Ballard, D., Holtz, Schwartz, 2011, **SPAA Best Paper Prize**)
  - Attainable too, more on parallel version later

Other Fast Matrix Multiplication Algorithms

- World's record was $O(n^{2.375477...})$
  - Coppersmith & Winograd, 1987
- New Record! 2.37*5477* reduced to 2.37*29269*
  - Virginia Vassilevska Williams, UC Berkeley & Stanford, 2011
- Newer Record! 2.372*9269* reduced to 2.372*8639*
  - Francois Le Gall, 2014

Latest Record! 2.3728*639* reduced to 2.3728*596*

  - Virginia Vassilevska Williams and Josh Alman, 2020
- Lower bound on #words moved can be extended to (some) of these algorithms (2015 thesis of Jacob Scott): $\Omega(n^w / M^{(w/2-1)})$
- Can show they all can be made numerically stable
  - Demmel, Dumitriu, Holtz, Kleinberg, 2007
- Can do rest of linear algebra (solve Ax=b, Ax=λx, etc) as fast , and stably
  - Demmel, Dumitriu, Holtz, 2008
- Fast methods (besides Strassen) may need unrealistically large n

Basic Linear Algebra Subroutines (BLAS)

- **Industry standard interface: www.netlib.org/blas, www.netlib.org/blas/blast--forum**
- **Vendors, others supply optimized implementations**

BLAS1: 1970s
- 15 operations
- vector ops: dot product, saxpy ($y=\alpha*x+y$), root-sum-squared, etc.
- **Computational Intensity**:
- m=2n, f=2n
- CI = ~1 or less

Slow

OK on vector machines

BLAS2: Mid-1980s
- 25 operations
- matrix-vector ops: matvec, etc.
- **Computational Intensity**:
- $m=n^2$, $f=2*n^2$
- CI~2 and less overhead

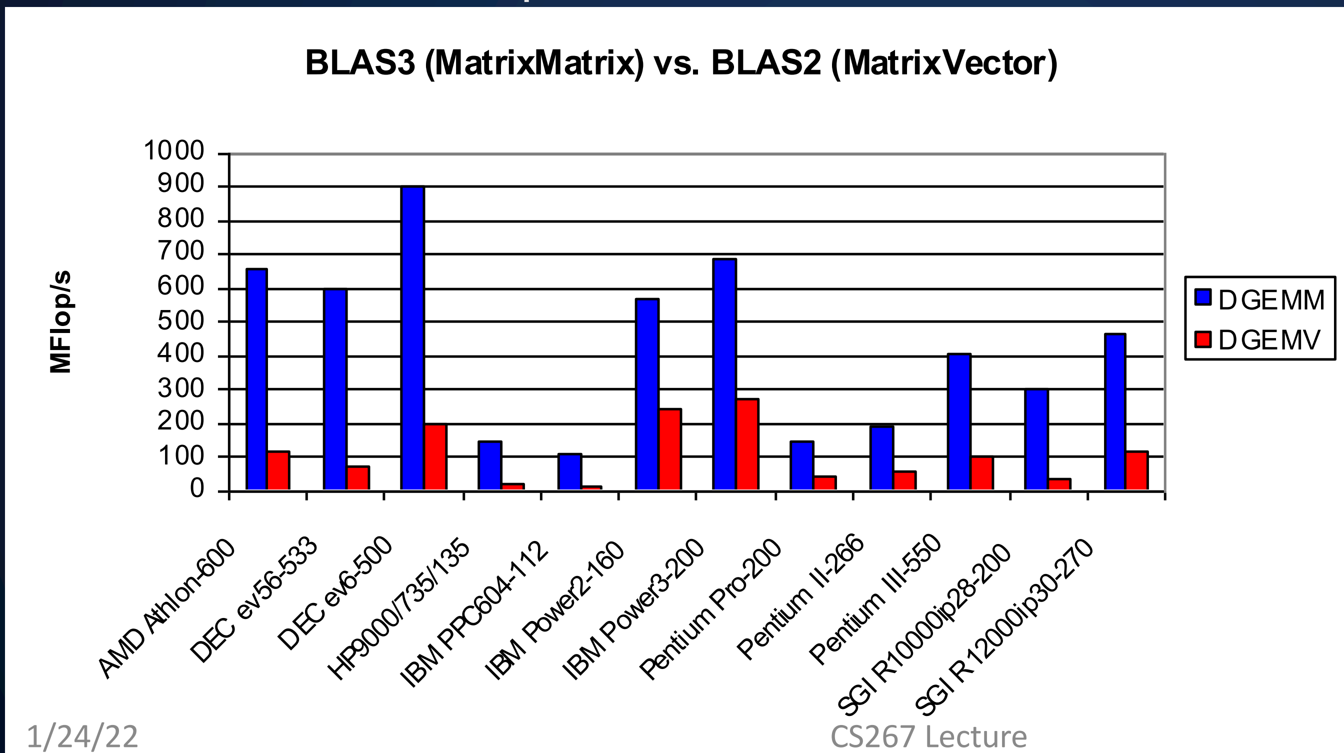
Better

BLAS3: Late-1980s
- 9 operations
- matrix-matrix ops: matmul, etc.
- **Computational Intensity**:
- $m <= 3n^2$, $f=O(n^3)$,
- CI as large as n

Much better

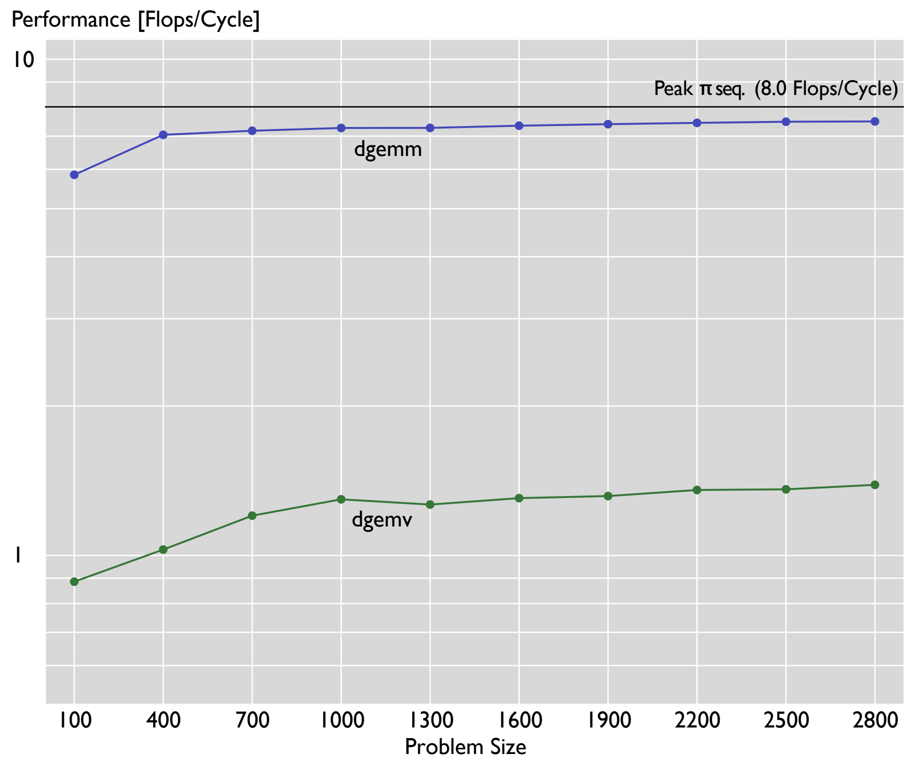
Dense Linear Algebra: BLAS2 vs. BLAS3

- BLAS2 and BLAS3 have very different computational intensity, and therefore different performance



ta source: Jack Dongarra

Measuring Performance — Flops/Cycle



Performance gap (flop/sec)

ETH

Image and paper by G. Ofenbeck, R. Steinman, V. Caparrós Cabezas, D. Spampinato, M. Püschel

Some reading on MatMul

- Sourcebook Chapter 3, (note that chapters 2 and 3 cover the material of lecture 2 and lecture 3, but not in the same order).
- "Performance Optimization of Numerically Intensive Codes", by Stefan Goedecker and Adolfy Hoisie, SIAM 2001.
- Web pages for reference:
  - BeBOP Homepage
  - ATLAS Homepage
  - BLAS (Basic Linear Algebra Subroutines), Reference for (unoptimized) implementations of the BLAS, with documentation.
  - LAPACK (Linear Algebra PACKage), a standard linear algebra library optimized to use the BLAS effectively on uniprocessors and shared memory machines (software, documentation and reports)
  - ScaLAPACK (Scalable LAPACK), a parallel version of LAPACK for distributed memory machines (software, documentation and reports)
- Tuning Strassen's Matrix Multiplication for Memory Efficiency Mithuna S. Thottethodi, Siddhartha Chatterjee, and Alvin R. Lebeck in Proceedings of Supercomputing '98, November 1998 postscript
- Recursive Array Layouts and Fast Parallel Matrix Multiplication" by Chatterjee et al. IEEE TPDS November 2002.
- Many related papers at bebop.cs.berkeley.edu

Take-Aways

- Matrix matrix multiplication
  - $2n^3$ flops on $3n^2$ data, so Computational Intensity up to O(n)
- Tiling matrix multiplication (cache aware)
  - Can increase block size to b if bxb blocks fit in fast memory
  - b = sqrt(M/3), the fast memory size M
  - Tiling (aka blocking) “cache-aware”
  - Cache-oblivious alternative
    - Works for any depth of (nested) memory hierarchy
    - In contrast, tiling uses 3 more nested loops for each level
- Optimized libraries (BLAS) exist

Roofline Model

How fast can an algorithm go in practice?

What is a Performance Model?

A formula to estimate performance

Running time

Bandwidth

Memory footprint

Energy Use

Percent of Peak

What is a Performance Model?

A formula to estimate performance

$O(n)$

$f * t_f + m * t_m$

$Lat + X / BW$

Examples we've seen for time

Why Use a Performance Model?

Understand performance behavior

- Differences between Architectures, Programming Models, implementations, etc.

?

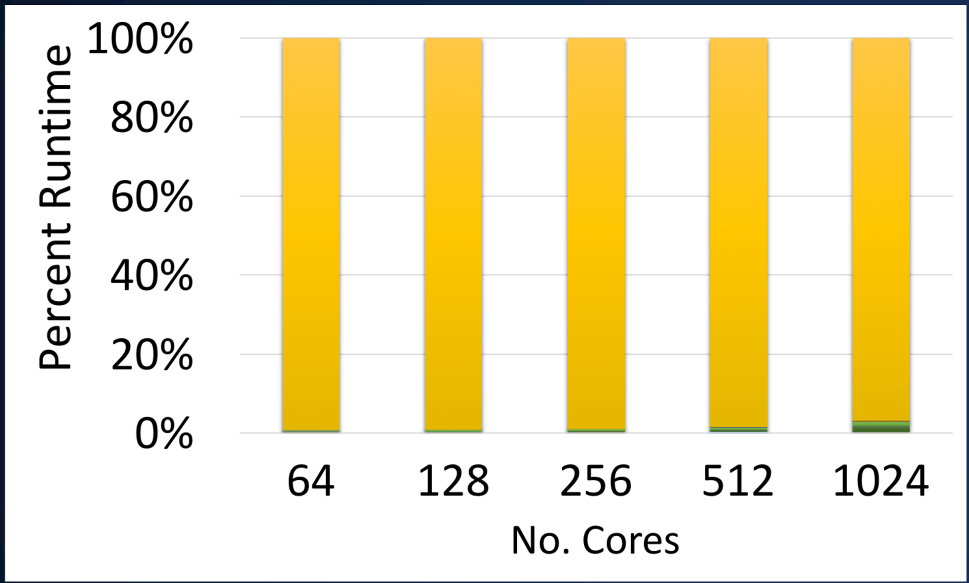
?

GCC

?

Intel
C++
Compiler

Why Use a Performance Model?



- Identify performance bottlenecks

Why Use a Performance Model?

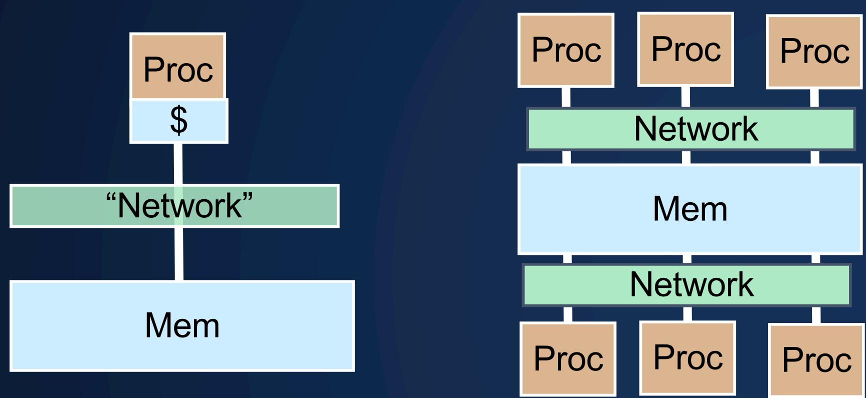
- Do you need
  - better software,
  - better hardware,
  - or a better algorithm

Why Use a Performance Model?

- **Determine when we're done optimizing**

Serial and Shared Memory Machines

Two types of machines so far in this class



Sequential Machine	Shared Memory

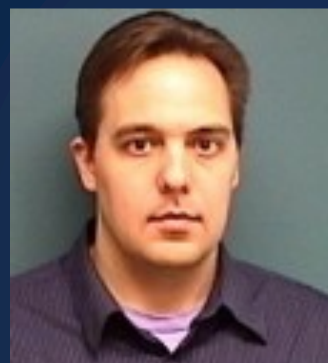
Critical performance issues

- Clock Speed and Parallelism (ILP, SIMD, Multicore)
- Memory latency and bandwidth

History of the Roofline Model

Sam Williams, PhD 2008

History of the Roofline Model



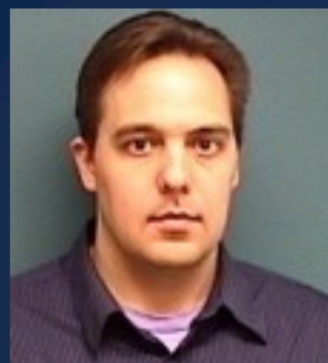
Samuel Williams, Andrew Waterman, David Patterson. "Roofline: an insightful visual performance model for multicore architectures." *Communications of the ACM* 52.4 (2009): 65-76.

2263 citations!

Sam Williams, PhD 2008

History of the Roofline Model

Roofline as a verb!



Sam Williams, PhD 2008

Samuel Williams, Andrew Waterman, David Patterson. "Roofline: an insightful visual performance model for multicore architectures." *Communications of the ACM* 52.4 (2009): 65-76.

1774 citations!

Roofline

Idea: applications are limited by either compute peak or memory bandwidth:

- Bandwidth bound (matvec)
- Compute bound (matmul)

What's in the Roofline Model?

Three pieces: 2 for machine and 1 for application

What's in the Roofline Model?

Three pieces: 2 for machine and 1 for application

- Arithmetic performance (flops/sec)
  - Clock Speed and Parallelism (ILP, SIMD, Multicore)

What’s in the Roofline Model?

Machine

Three pieces: 2 for machine and 1 for application

- Arithmetic performance (flops/sec)
  - Clock Speed and Parallelism (ILP, SIMD, Multicore)
- Memory bandwidth (bytes /sec)
  - Latency not included (looking at best case)

What's in the Roofline Model?

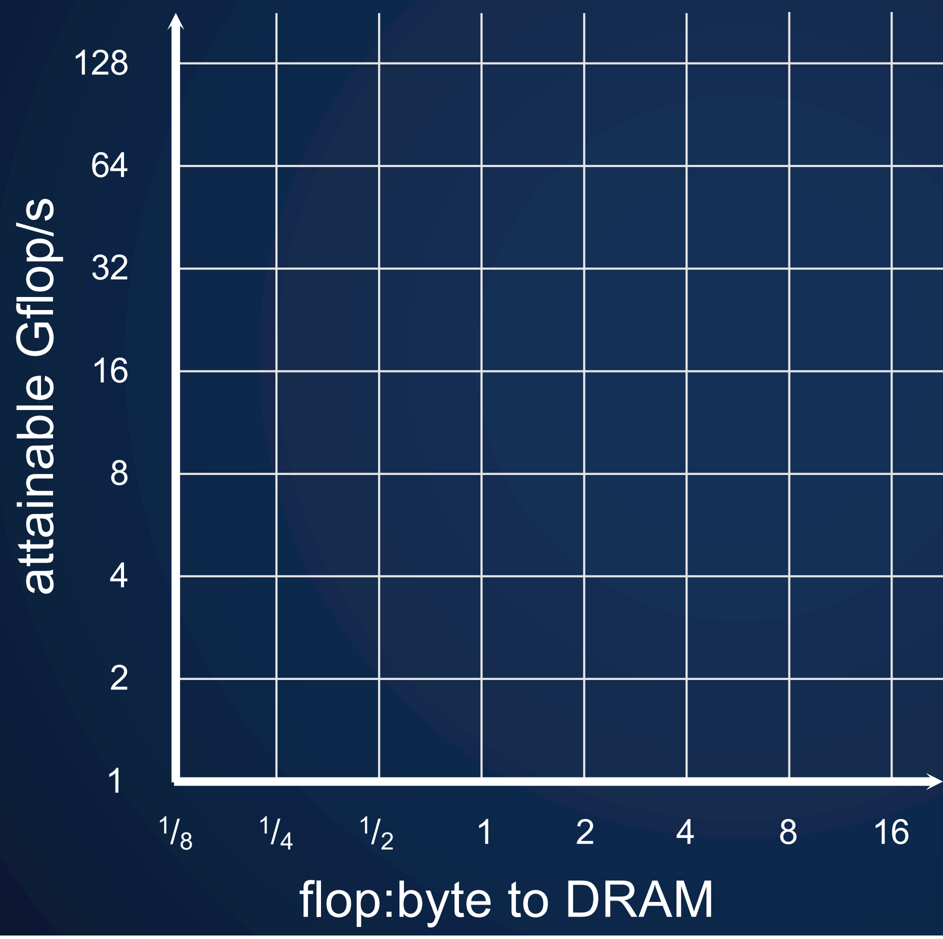
Machine

Application

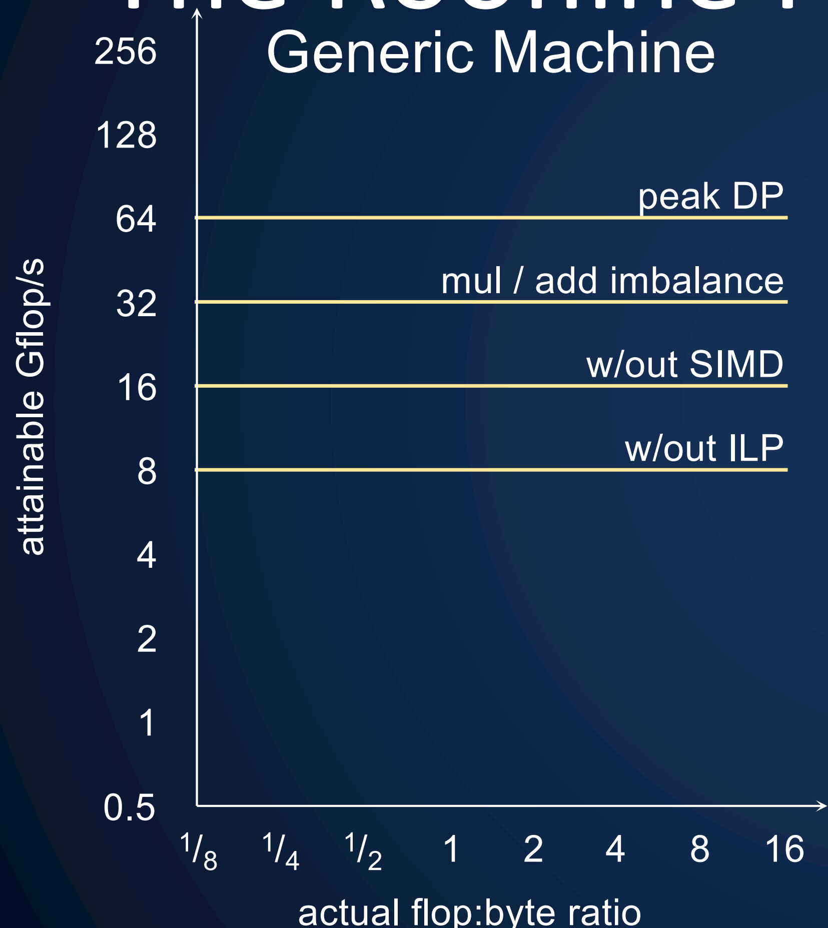
Three pieces: 2 for machine and 1 for application

- Arithmetic performance (flops/sec)
  - Clock Speed and Parallelism (ILP, SIMD, Multicore)
- Memory bandwidth (bytes /sec)
  - Latency not included (looking at best case)
- Computational (Arithmetic) Intensity
  - Application balances (flops/word or flops/byte)

The Roofline Performance Model

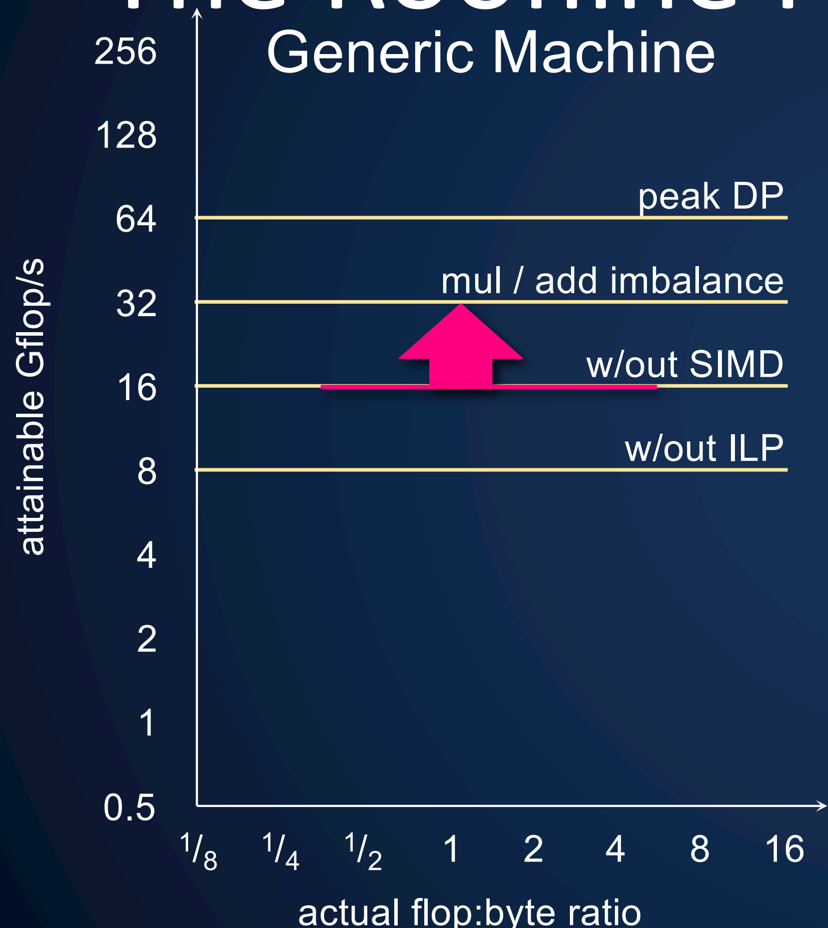


The Roofline Performance Model



- Top of the roof is the peak compute rate
- No FMA, no SIMD, no ILP will lower what is attainable

The Roofline Performance Model



- Top of the roof is the peak compute rate
- No FMA, no SIMD, no ILP will lower what is attainable

How good is flop/s as a model?

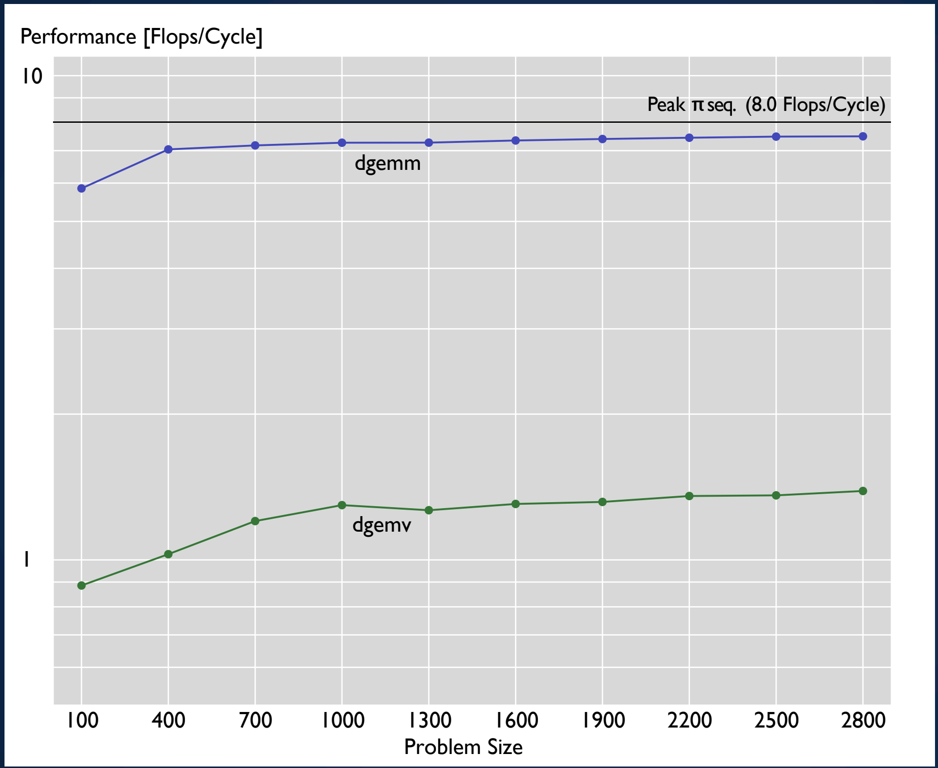
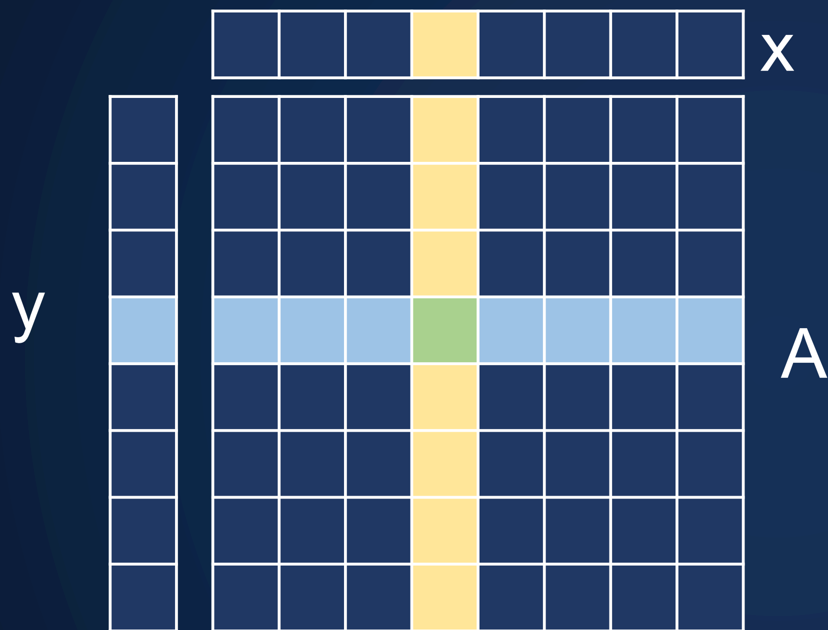


Image and paper by G. Ofenbeck, R. Steinman, V. Caparrós Cabezas, D. Spampinato, M. Püschel

What's a better model for DGEMV (Matrix-vector multiply)?

What's a better model for MatVec?



y += A * x

Best case is ~2 flops / word
(1/2 per byte for single, ¼ for double)

Data Movement Complexity

- Assume run time ~= data moved to/from DRAM
- Hard to estimate without cache details

Data Movement Complexity

- Assume run time ~= data moved to/from DRAM
- Hard to estimate without cache details
- Compulsory data movement (data structure sizes) are good first guess
- Performance upper bound: guaranteed not to exceed

Operation	FLOPs	Data
Dot Prod	O(n)	O(n)
Mat Vec	$O(n^2)$	$O(n^2)$
MatMul	$O(n^3)$	$O(n^2)$
N-Body	$O(n^2)$	O(n)
FFT	O(n log n)	O(n)

Machine Balance and Computational Intensity

- Machine balance is:

$$\text{Balance} = \frac{\text{Peak DP FLOP/s}}{\text{Peak Bandwidth}}$$

Machine Balance and Computational Intensity

- Machine balance is:

$$\text{Balance} = \frac{\text{Peak DP FLOP/s}}{\text{Peak Bandwidth}}$$

What is typical? 5-10 Flops/Byte
And not getting better (lower) over time

Machine Balance and Computational Intensity

- Machine balance is:

$$\text{Balance} = \frac{\text{Peak DP FLOP/s}}{\text{Peak Bandwidth}}$$

What is typical? 5-10 Flops/Byte
And not getting better (lower) over time

Haswell is 10 Flops/Byte
KNL is 34 Flops/Byte to DRAM
7 Flops/Byte to HBM

Machine Balance and Computational Intensity

- Machine balance is:

$$\text{Balance} = \frac{\text{Peak DP FLOP/s}}{\text{Peak Bandwidth}}$$

- Computational / arithmetic intensity (CI/AI) is:

$$\text{CI} = \frac{\text{FLOPs Performed}}{\text{Data Moved}}$$

Machine Balance and Computational Intensity

- Machine balance is:

$$\text{Balance} = \frac{\text{Peak DP FLOP/s}}{\text{Peak Bandwidth}}$$

- Computational / arithmetic intensity (CI/AI) is:

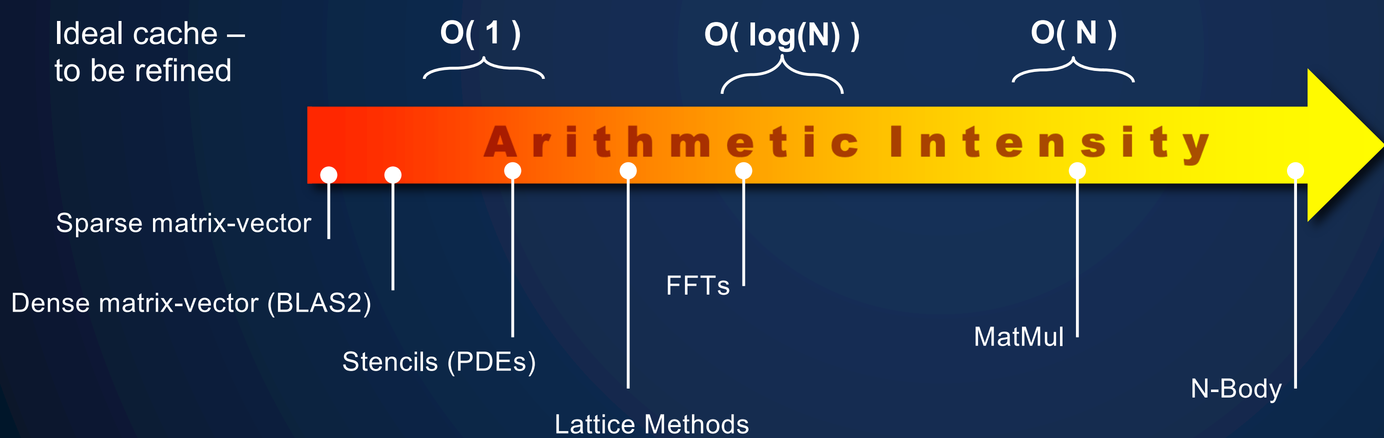
$$\text{CI} = \frac{\text{FLOPs Performed}}{\text{Data Moved}}$$

Operation	FLOPs	Data	CI
Dot Prod	$O(n)$	$O(n)$	$O(1)$
Mat Vec	$O(n^2)$	$O(n^2)$	$O(1)$
MatMul	$O(n^3)$	$O(n^2)$	$O(n)$
N-Body	$O(n^2)$	$O(n)$	$O(n)$
FFT	$O(n \log n)$	$O(n)$	$O(\log n)$

Ideal (infinite cache)

Computational Intensity

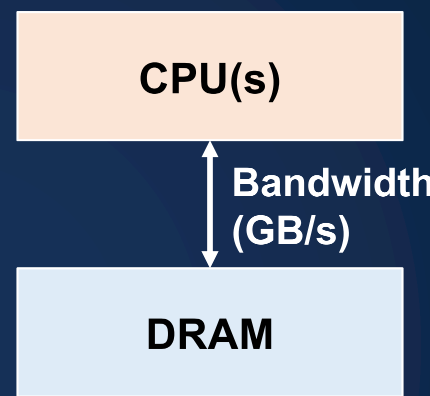
- Can look at computational intensity as a spectrum
- Constants (at least leading constants) will matter



(DRAM) Roofline

Assume

- Idealized processor/caches
- Cold start (data in DRAM)

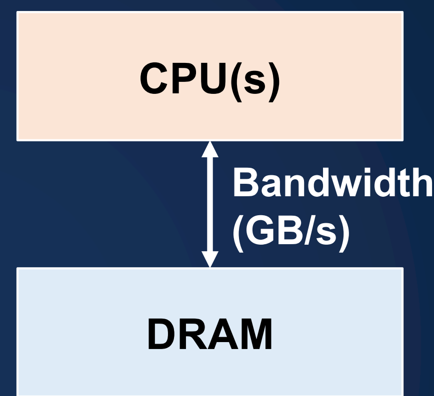


$$\textbf{Time} = \max \begin{cases} \text{\#FP ops / Peak GFLOP/s} \\ \text{\#Bytes / Peak GB/s} \end{cases}$$

(DRAM) Roofline

Assume

- Idealized processor/caches
- Cold start (data in DRAM)



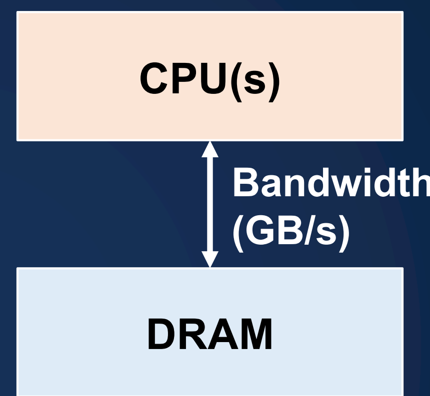
Time = max $\begin{cases} \text{\#FP ops / Peak GFLOP/s} \\ \text{\#Bytes / Peak GB/s} \end{cases}$

Why max rather than sum?

(DRAM) Roofline

Assume

- Idealized processor/caches
- Cold start (data in DRAM)

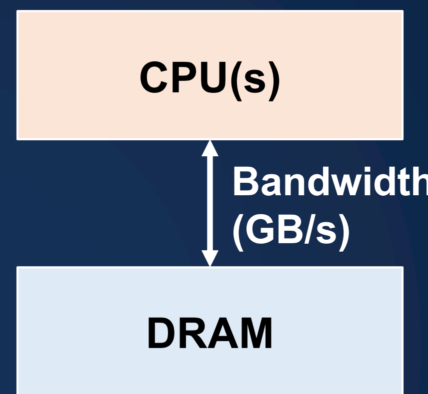


$$\frac{\text{\#FP ops}}{\text{Time}} = \min \begin{cases} \text{Peak GFLOP/s} \\ \text{(\#FP ops / \#Bytes) * Peak GB/s} \end{cases}$$

(DRAM) Roofline

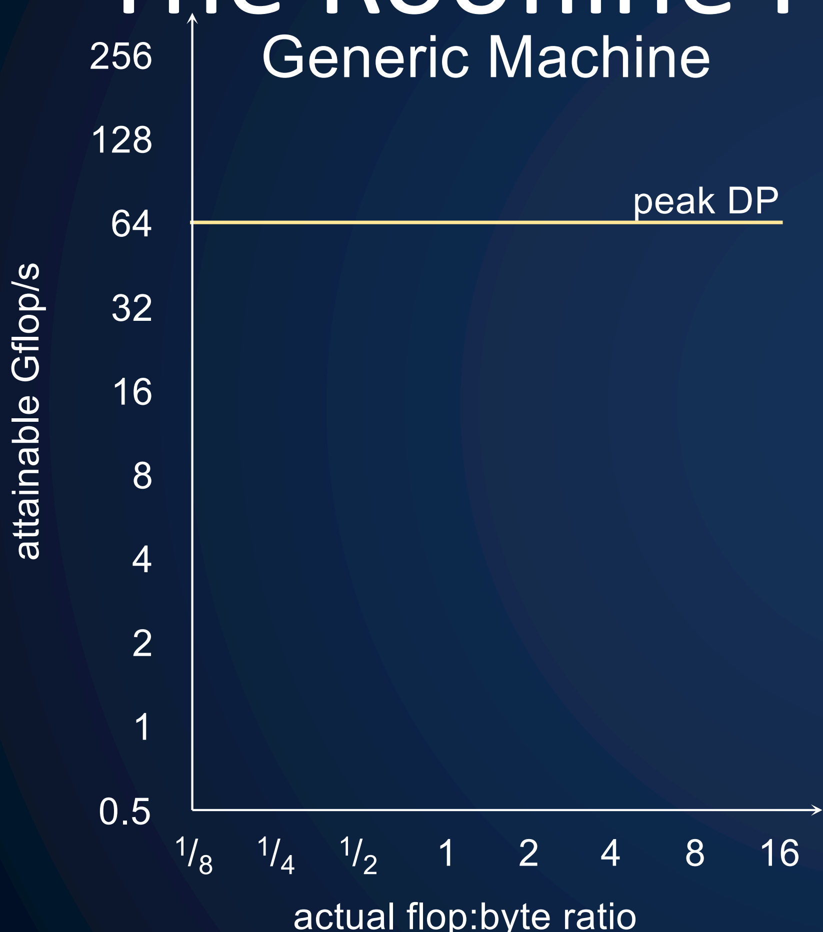
Assume

- Idealized processor/caches
- Cold start (data in DRAM)



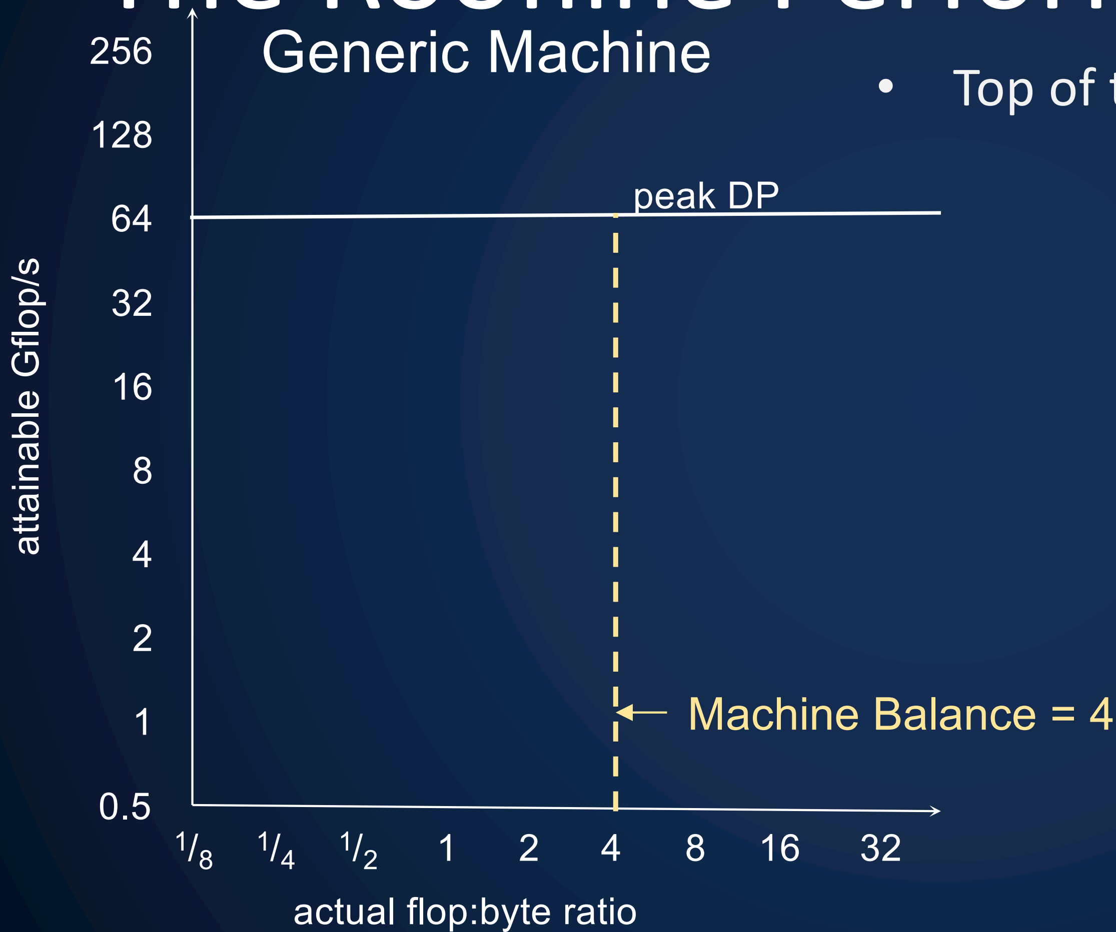
$$\text{GFlop/sec} = \min \begin{cases} \text{Peak GFLOP/s} \\ (\text{\#FP ops / \#Bytes}) * \text{Peak GB/s} \end{cases}$$

The Roofline Performance Model



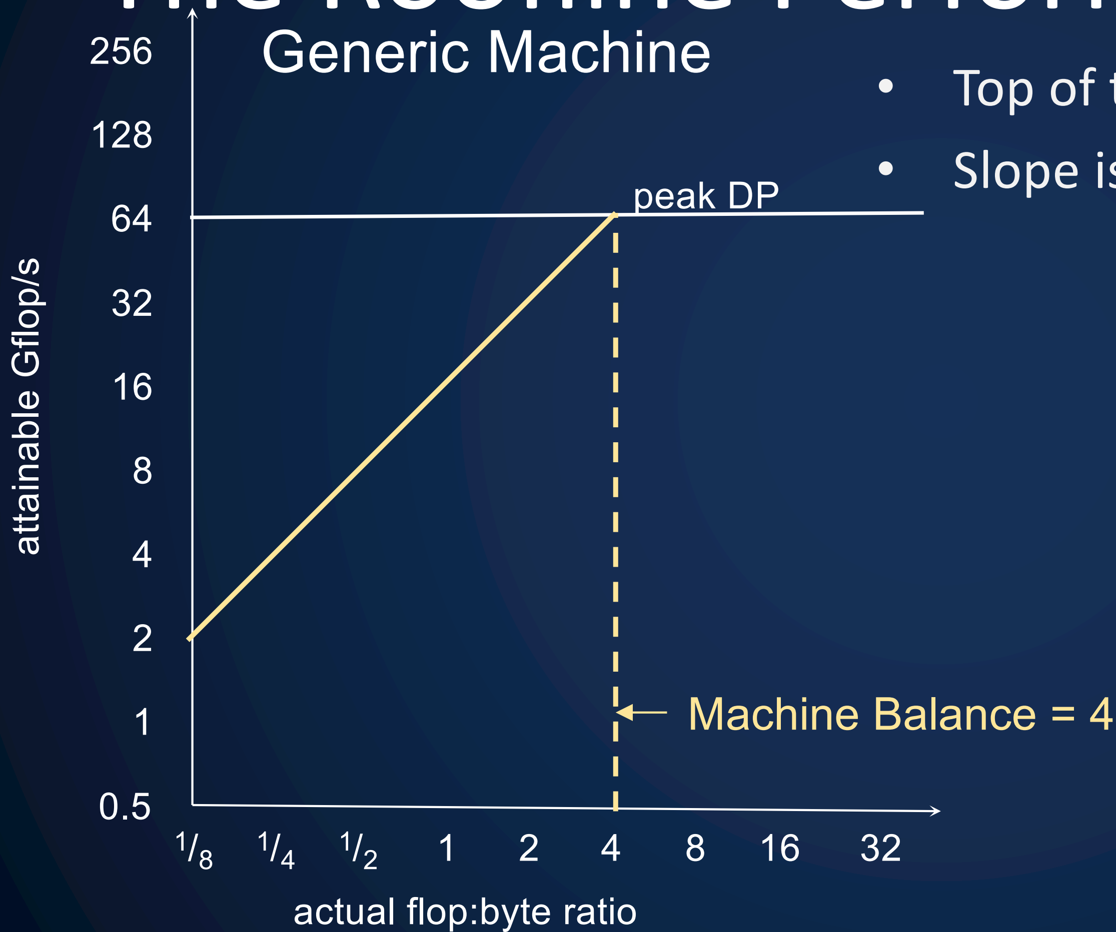
- Top of the roof is the peak compute rate

The Roofline Performance Model



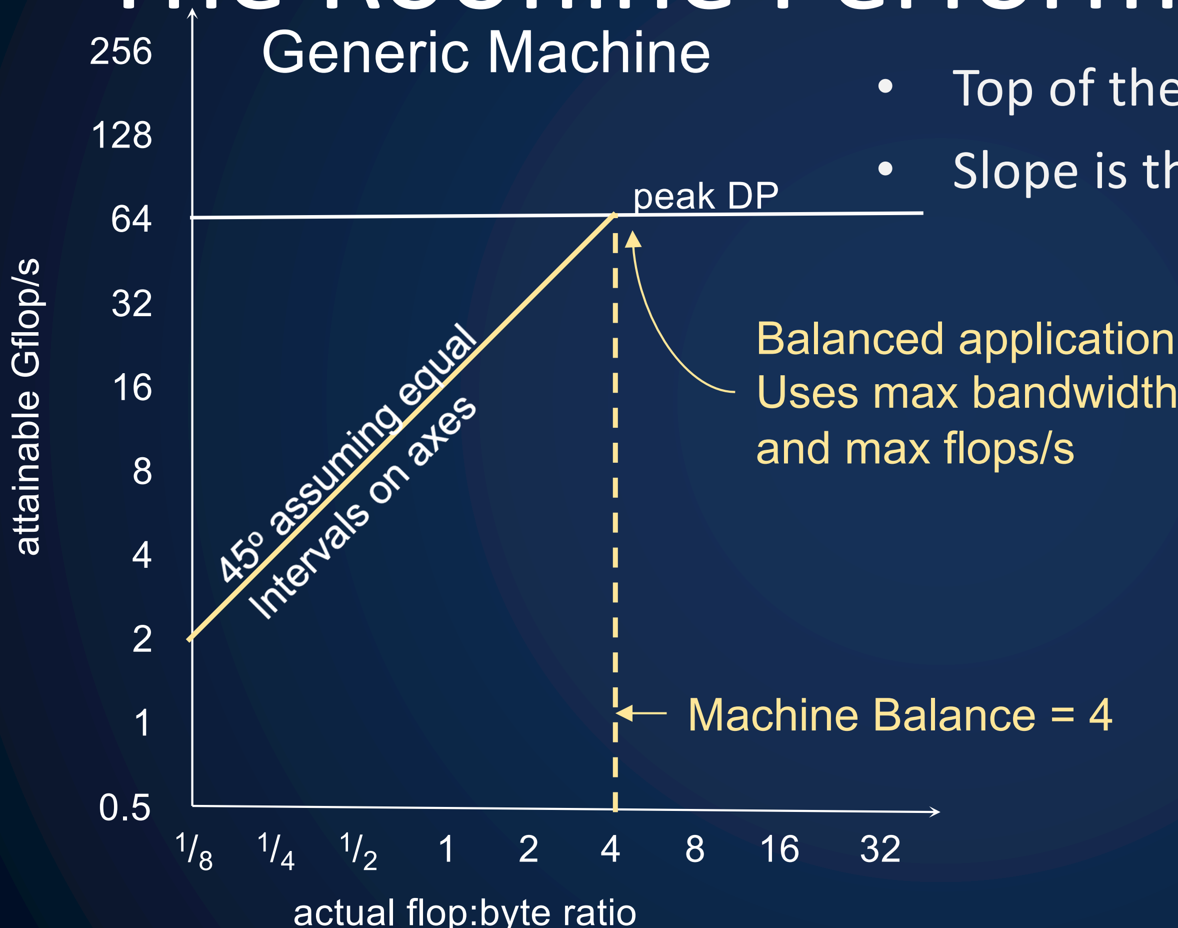
- Top of the roof is the peak compute rate

The Roofline Performance Model



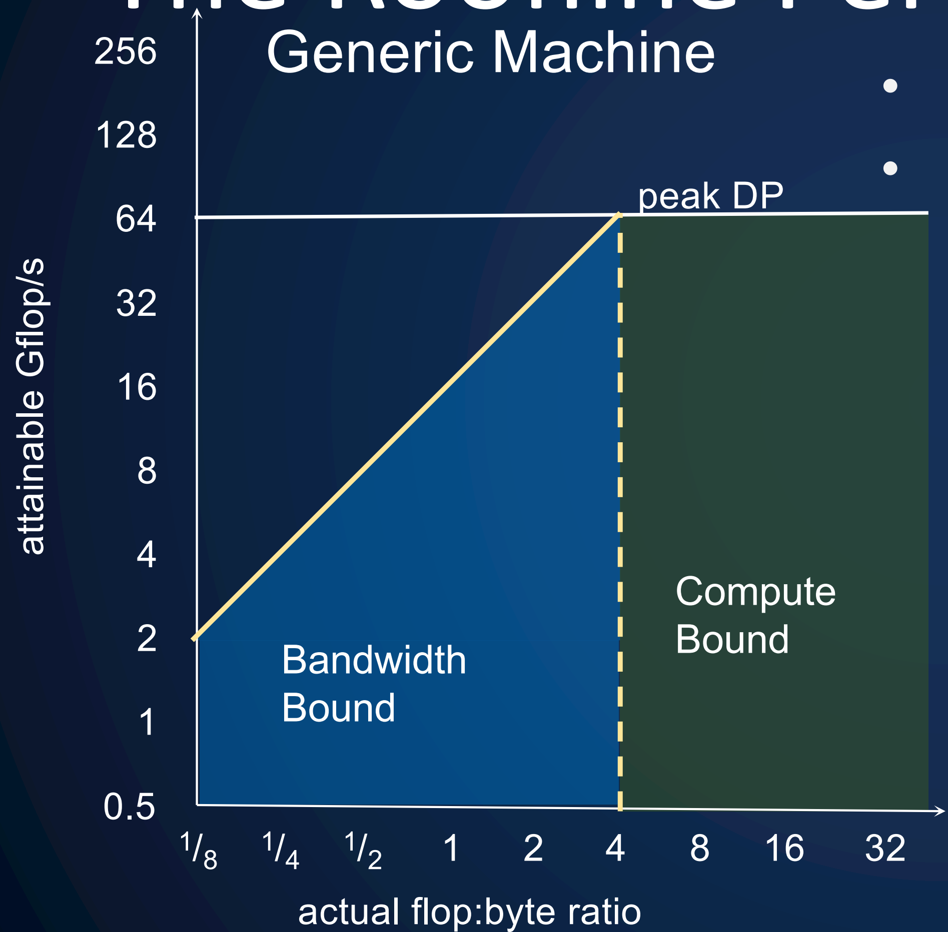
- Top of the roof is the peak compute rate
- Slope is the memory bandwidth

The Roofline Performance Model



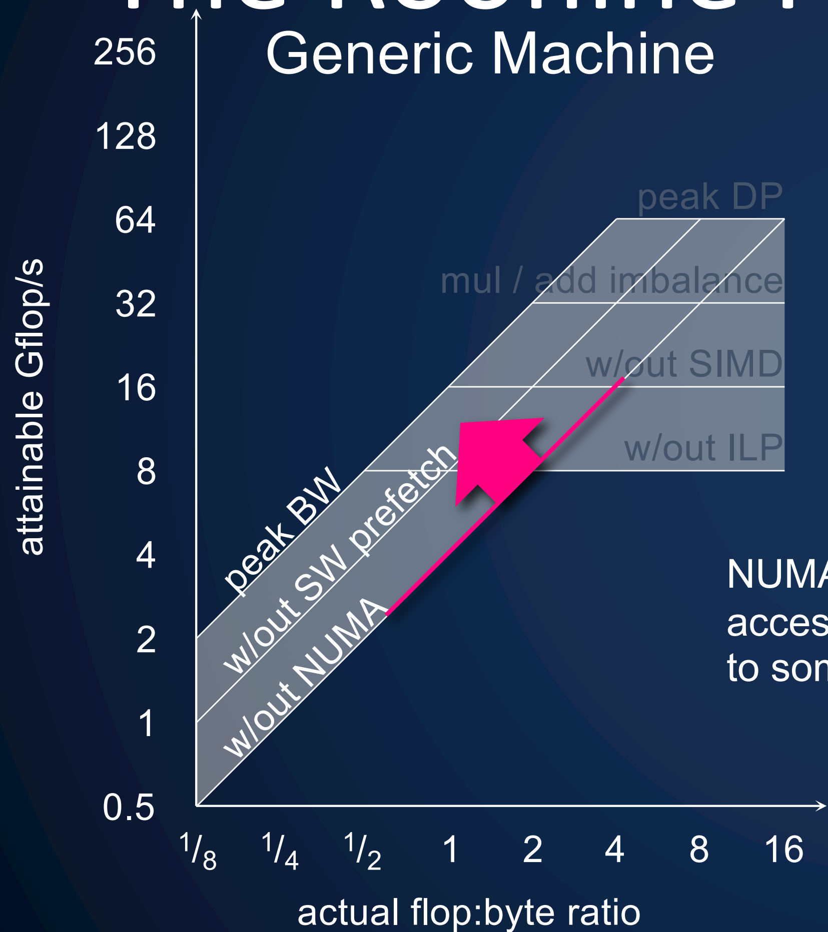
- Top of the roof is the peak compute rate
- Slope is the memory bandwidth

The Roofline Performance Model



- Top of the roof is the peak compute rate
- Slope is the memory bandwidth

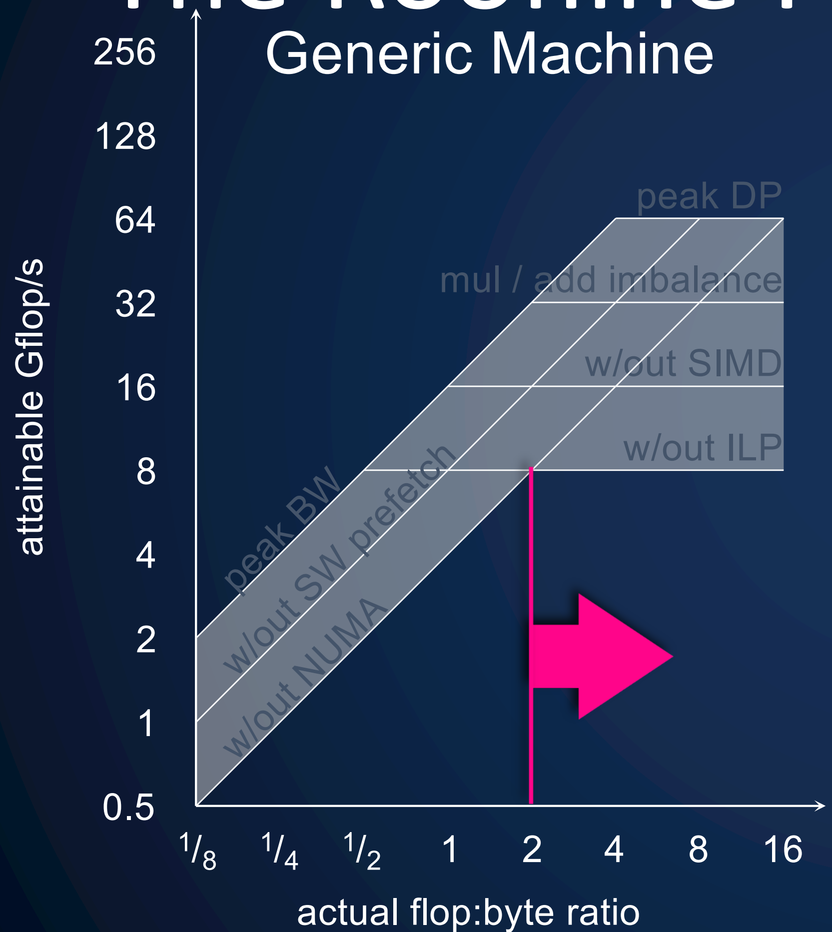
The Roofline Performance Model



- Lack prefetch, ignoring NUMA, etc. will reduce attainable bandwidth
- Different bandwidths for multiple levels (e.g., HBM vs DRAM)

NUMA = non-uniform memory access (some memory closer to some cores, e.g. 2 sockets)

The Roofline Performance Model

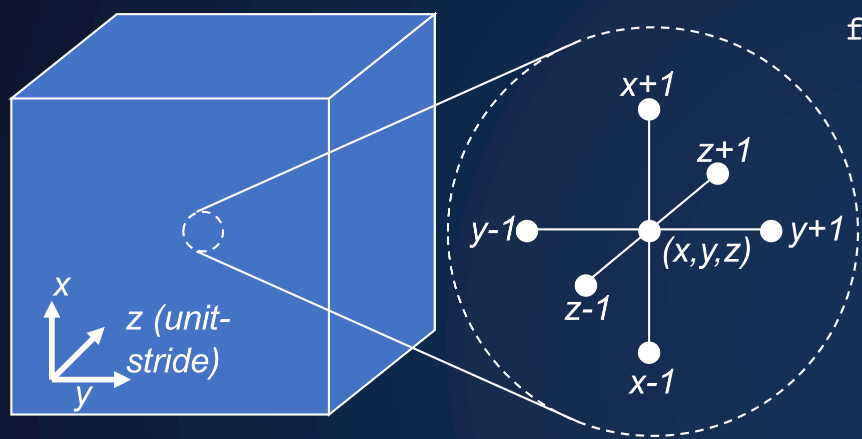


- Locations of posts in the building are determined by algorithmic intensity
- Will vary across algorithms and with bandwidth-reducing optimizations, such as better cache re-use (tiling), compression techniques

Roofline Example #1


```
#pragma omp parallel for
for(i=0;i<N;i++){
  Z[i] = X[i] + alpha*Y[i];
}
```

Roofline Example #2



3D 7-point stencil

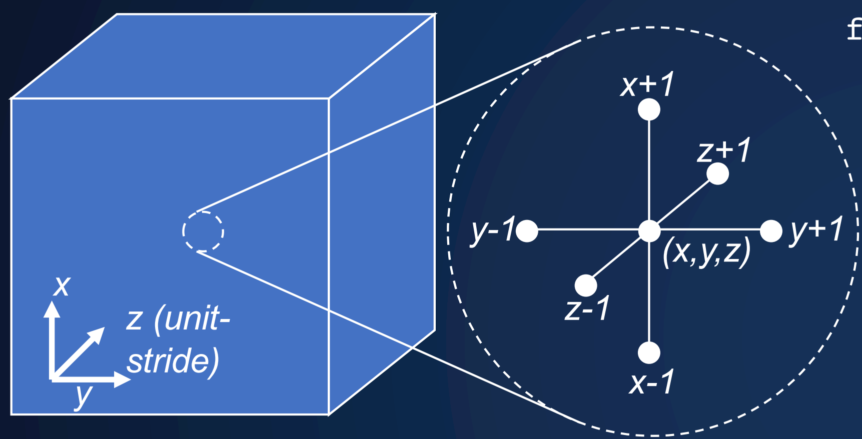
```
for x,y,z in 0 to n-1
 Next[x,y,z]=
   C0 * Current[x,y,z]+
   C1 *(Current[x-1,y,z]+
        Current[x+1,y,z]+
        Current[x,y-1,z]+
        Current[x,y+1,z]+
        Current[x,y,z-1]+
        Current[x,y,z+1]);
```

Inner Loop Pseudocode

A 7-point constant coefficient stencil…

- 7 flops, 8 memory references (7 reads, 1 store) per point
- **CI = 0.11 flops per byte**

Roofline Example #2



3D 7-point stencil

```
for x,y,z in 0 to n-1
 Next[x,y,z]=
   C0 * Current[x,y,z]+
   C1 *(Current[x-1,y,z]+
        Current[x+1,y,z]+
        Current[x,y-1,z]+
        Current[x,y+1,z]+
        Current[x,y,z-1]+
        Current[x,y,z+1]);
```

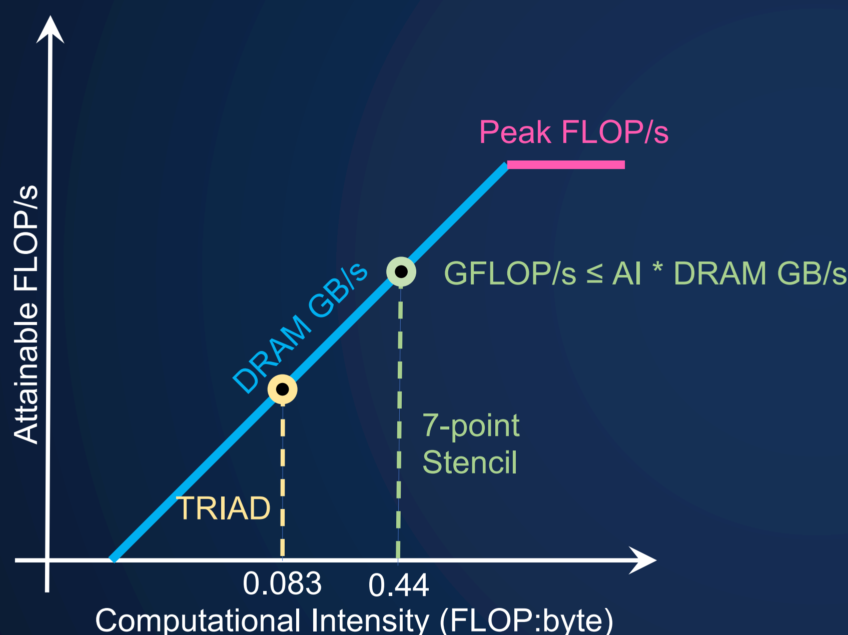
Inner Loop Pseudocode

A 7-point constant coefficient stencil…

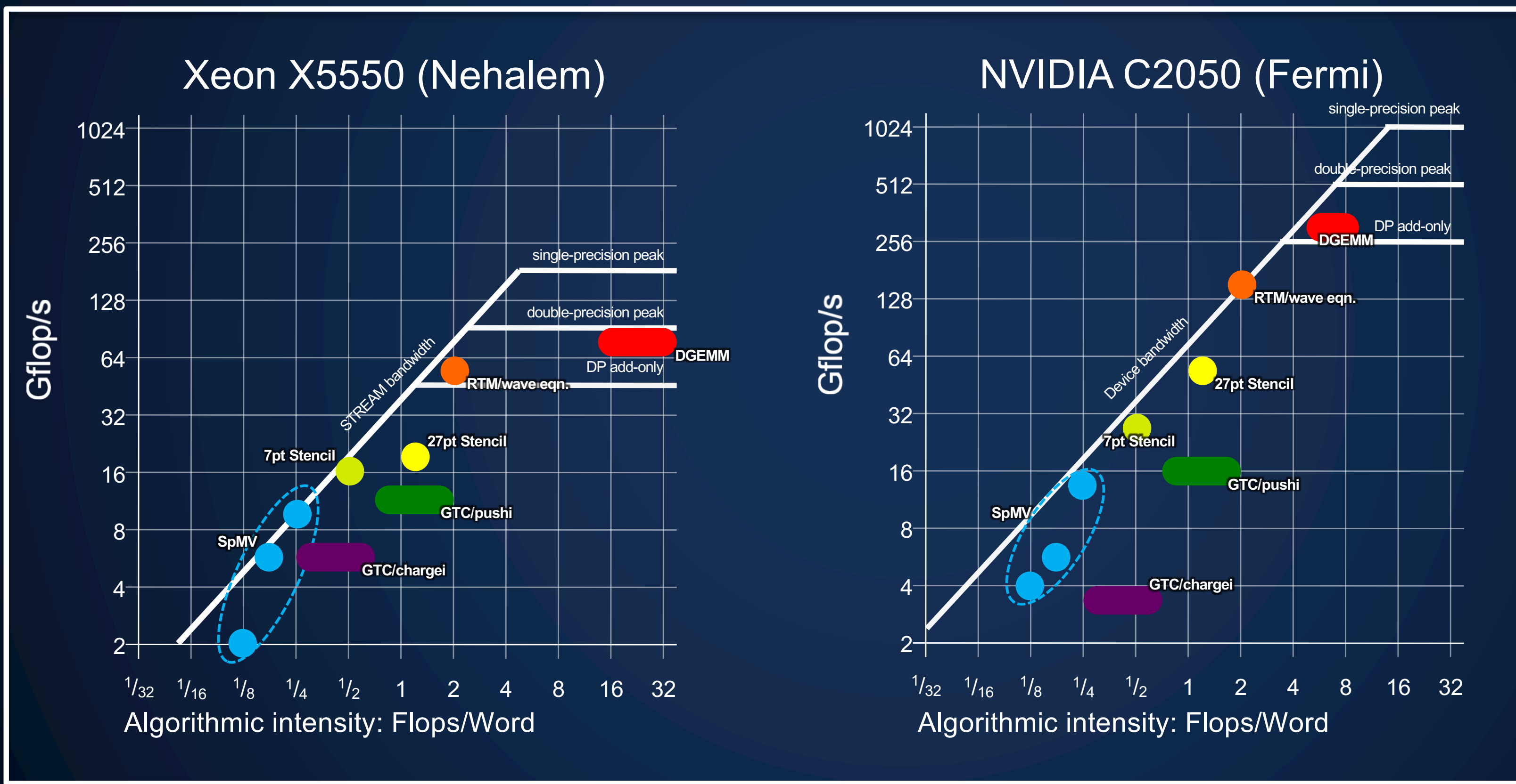
- 7 flops, 8 memory references (7 reads, 1 store) per point
- Cache can filter all but 1 read and 1 write per point
- **CI = 0.44 flops per byte**

Roofline Example #2

- Still O(1) flops / byte
- **But (leading) constants matter**



Roofline Across Algorithms



Work by Williams, Oliker, Shalf, Madduri, Kamil, Im, Ethier,…

Takeaways

- Roofline captures upper bound performance
- The min of 2 upper bounds for a machine
  - Peak flops (or other arith ops)
  - Memory bandwidth max
- Algorithm computational intensity
  - Usually defined as best case, infinite cache
- Originally for single processors and SMPs
- Widely used in practice and adapted to any bandwidth/compute limit situation