

Applications of Parallel Computers

Cache Oblivious MatMul and the Roofline Model

<https://sites.google.com/lbl.gov/cs267-spr2021>



Review

A Simple Model of Memory

- Assume just 2 levels in the hierarchy, fast and slow

- All data initially in slow memory

- $m = \text{number of memory elements (words) moved between fast and slow memory}$
- $t_m = \text{time per slow memory operation (inverse bandwidth in best case)}$
- $f = \text{number of arithmetic operations}$
- $t_f = \text{time per arithmetic operation} < t_m$
- $\boxed{CI = f / m}$ average number of flops per slow memory access

- Minimum possible time = $f * t_f$ when all data in fast memory

- Actual time

$$f * t_f + m * t_m = f * t_f * (1 + t_m/t_f * 1/CI)$$

- Larger CI means time closer to minimum $f * t_f$

Computational Intensity (CI): Key to algorithm efficiency

Machine Balance: Key to machine efficiency

1/25/21

CS267 Lecture

2

Naïve Matrix Multiply

{implements $C = C + A * B$ }

for $i = 1$ to n

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

{read row i of A into fast memory}

n^2 to read each row of A once

for $j = 1$ to n

{read $C[i,j]$ into fast memory}

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

{read column j of B into fast memory}

n^3 to read each column of B n times

for $k = 1$ to n

$C[i,j] = C[i,j] + A[i,k] * B[k,j]$

So the computational intensity is:
 $CI = f / m = 2n^3 / [n^3 + 3n^2] \approx 2$

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

No better than matrix-vector!

$$\begin{matrix} C[i,j] \\ \quad \quad \quad \end{matrix} = \begin{matrix} C[i,j] \\ \quad \quad \quad \end{matrix} + \begin{matrix} A[i,:] \\ \quad \quad \quad \end{matrix} * \begin{matrix} B[:,j] \\ \quad \quad \quad \end{matrix}$$

1/25/21

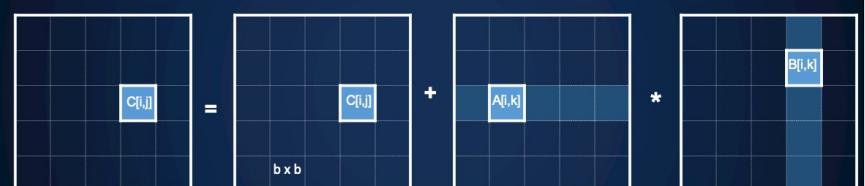
CS267 Lecture

3

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
 $\leftarrow n \text{ elements} \rightarrow$
 $\leftarrow N \text{ blocks} \rightarrow$
Each block is $b \times b$

1/25/21

CS267 Lecture

4

Blocked (Tiled) Matrix Multiply

Consider A,B,C to be N-by-N matrices of b-by-b subblocks where

```
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}
```

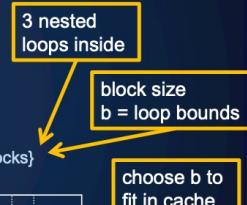
Tiling for registers or caches

1/26/21

CS267 Lecture

5

b=n / N is called the block size



choose b to fit in cache

Blocked [Tiled] Matrix Multiply

Consider A,B,C to be N-by-N matrices of b-by-b subblocks where

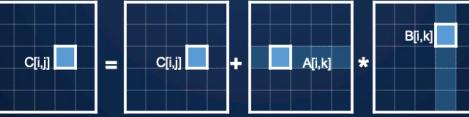
```
for i = 1 to N
  for j = 1 to N
```

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

1/25/21



b=n / N is called the block size

6

Blocked [Tiled] Matrix Multiply

Consider A,B,C to be N-by-N matrices of b-by-b subblocks where

```
for i = 1 to N
  for j = 1 to N
    {read block C[i,j] into fast memory} 2n2 to read and write each block of C once
    for k = 1 to N
      {read block A[i,k] into fast memory} (2N2 * b2 = 2n2)
```

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}

nxn elements
NxN blocks
Each block is bxb

1/25/21

CS267 Lecture

7

Blocked [Tiled] Matrix Multiply

Consider A,B,C to be N-by-N matrices of b-by-b subblocks where

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}

2n² to read and write each block of C once

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

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

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

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

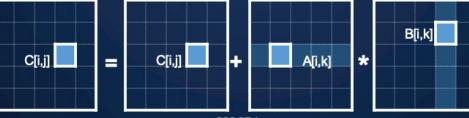
nxn elements
NxN blocks
Each block is bxb

1/25/21

CS267 Lecture

b=n / N is called the block size

8



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} 2n2 to read and write each block of C once  
    {read block B[k,j] into fast memory} N*n2 to read each block of A N3 times  
    for k = 1 to N  
      {read block A[i,k] into fast memory} N*n2 to read each block of B N3 times  
      {read block B[k,j] into fast memory} (N3*b2 =N3*(n/N)2)  
      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}
```



1/25/21

9

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} 2n2 to read and write each block of C once  
    {read block B[k,j] into fast memory} N*n2 to read each block of A N3 times  
    for k = 1 to N  
      {read block A[i,k] into fast memory} N*n2 to read each block of B N3 times  
      {read block B[k,j] into fast memory} N*n2 to read each block of B N3 times  
      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}
```



1/25/21

10

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} 2n2 to read and write each block of C once  
    {read block B[k,j] into fast memory} N*n2 to read each block of A N3 times  
    for k = 1 to N  
      {read block A[i,k] into fast memory} N*n2 to read each block of B N3 times  
      {read block B[k,j] into fast memory} N*n2 to read each block of B N3 times  
      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}
```

Memory words moved: $m = 2n^2 + N*n^2 + N*n^2 = 2n^2(1+N)$

1/25/21

CS267 Lecture

11

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} 2n2 to read and write each block of C once  
    {read block B[k,j] into fast memory} N*n2 to read each block of A N3 times  
    for k = 1 to N  
      {read block A[i,k] into fast memory} N*n2 to read each block of B N3 times  
      {read block B[k,j] into fast memory} N*n2 to read each block of B N3 times  
      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}
```

Memory words moved: $m = 2n^2 + N*n^2 + N*n^2 = 2n^2(1+N)$

Computational Intensity: $C_I = f / m = 2n^3 / ((2N + 2) * n^2)$
 $\approx n / N = b$ for large n

1/25/21

CS267 Lecture

12

Blocked [Tiled] Matrix Multiply

Computational Intensity (CI) = b for large n

How large can we make b ? Assume our fast memory has size M_{fast} :

$$b \leq \sqrt{M_{fast}/3} \quad \text{To hold 3 bxb blocks (may use less in practice)}$$

nxn elements
NxN blocks
Each block is bxb

$$\begin{matrix} C[i,j] \\ \boxed{\square} \end{matrix} = \begin{matrix} C[i,j] \\ \boxed{\square} \end{matrix} + \begin{matrix} \boxed{\square} \\ A[i,k] \end{matrix} * \begin{matrix} B[j,k] \\ \boxed{\square} \end{matrix}$$

1/25/21

CS267 Lecture

13

Blocked [Tiled] Matrix Multiply

Computational Intensity (CI) = b for large n

How large can we make b ? Assume our fast memory has size M_{fast} :

$$b \leq \sqrt{M_{fast}/3} \quad \text{To hold 3 bxb blocks (may use less in practice)}$$

So $m = 2n^2(1+N) = 2n^2(1+n/b) = O(n^3/\sqrt{M})$
 $CI = O(\sqrt{M})$ Bigger cache, larger blocks, better performance

nxn elements
NxN blocks
Each block is bxb

$$\begin{matrix} C[i,j] \\ \boxed{\square} \end{matrix} = \begin{matrix} C[i,j] \\ \boxed{\square} \end{matrix} + \begin{matrix} \boxed{\square} \\ A[i,k] \end{matrix} * \begin{matrix} B[j,k] \\ \boxed{\square} \end{matrix}$$

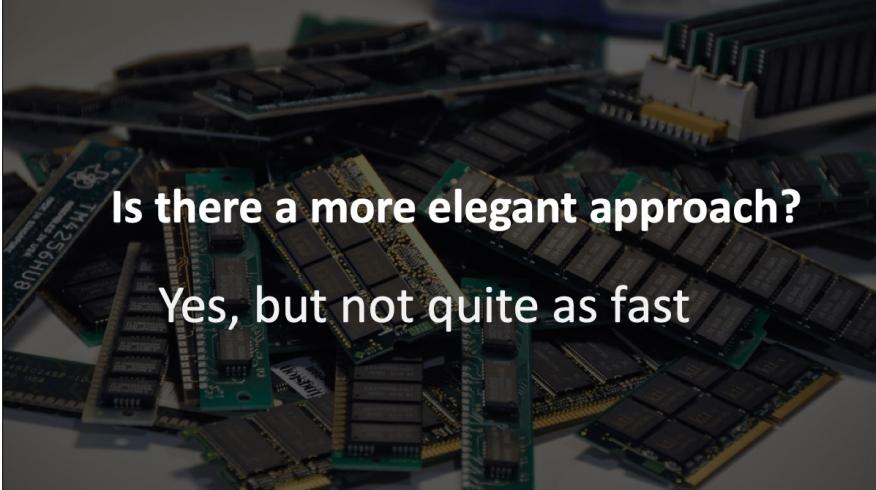
1/26/21

CS267 Lecture

14



Is there a more elegant approach?



Is there a more elegant approach?

Yes, but not quite as fast

Recursive Matrix Multiplication

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

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

- True when each block is a 1×1 or $n/2 \times n/2$
- For simplicity: square matrices with $n = 2^m$
 - Extends to general rectangular case

1/25/21

CS267 Lecture

17

Recursive Matrix Multiplication

```
Define C = RMM (A, B, n)
if (n==1) {
    C00 = A00 * B00;
}
```

return C

1/25/21

CS267 Lecture

18

Recursive Matrix Multiplication

```
Define C = RMM (A, B, n)
if (n==1) {
    C00 = A00 * B00;
} 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
```

1/25/21

CS267 Lecture

19

Recursive Matrix Multiplication

```
Define C = RMM (A, B, n)
if (n==1) { C00 = A00 * B00; } 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
```

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

1/25/21

CS267 Lecture

20

Recursive Matrix Multiplication

```
Define C = RMM (A, B, n)
if (n==1) { C00 = A00 * B00; } 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
```

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

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

1/25/21

CS267 Lecture

21

Recursive Matrix Multiplication

```
Define C = RMM (A, B, n)
if (n==1) { C00 = A00 * B00; } 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 · Arith(n/2) + 4(n/2)² if n > 1, else 1

22

1/25/21

CS267 Lecture

Recursive Matrix Multiplication

```
Define C = RMM (A, B, n)
if (n==1) { C00 = A00 * B00; } 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 · Arith(n/2) + 4(n/2)² if n > 1, else 1
= 2n³ ... same operations as usual, in different order

1/25/21

CS267 Lecture

23

Recursive Matrix Multiplication

```
Define C = RMM (A, B, n)
if (n==1) { C00 = A00 * B00; } 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 · Arith(n/2) + 4(n/2)² if n > 1, else 1
= O(n³) this is our f = # flops

24

CS267 Lecture

23

Recursive Matrix Multiplication

```
Define C = RMM (A, B, n)
if (n==1) { C00 = A00 * B00; } 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
```

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

What is
m, data
moved?

1/25/21

CS267 Lecture

25

Recursive Matrix Multiplication

```
Define C = RMM (A, B, n)
if (n==1) { C00 = A00 * B00; } 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
```

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

$$m = \begin{cases} W(n) = \# \text{ words moved between fast, slow memory by RMM(. . . , n)} \end{cases}$$

1/25/21

CS267 Lecture

26

Recursive Matrix Multiplication

```
Define C = RMM (A, B, n)
if (n==1) { C00 = A00 * B00; } 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
```

4 lines of code
3 matrices per line
stops if 3 matrices fit

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

$$m = \begin{cases} W(n) = \# \text{ words moved between fast, slow memory by RMM(. . . , n)} \\ = 8 \cdot W(n/2) + 4 \cdot 3(n/2)^2 \text{ if } 3n^2 > M_{\text{fast}}, \text{ else } 3n^2 \end{cases}$$

1/25/21

CS267 Lecture

27

Recursive Matrix Multiplication

```
Define C = RMM (A, B, n)
if (n==1) { C00 = A00 * B00; } 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
```

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

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

1/25/21

CS267 Lecture

28

Recursive Matrix Multiplication

```
Define C = RMM (A, B, n)
if (n==1) { C00 = A00 * B00; } 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
```

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

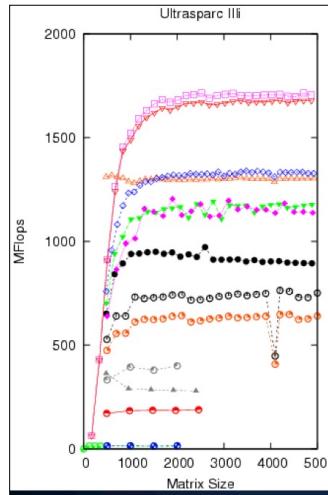
$$m = \begin{cases} W(n) = \# \text{ words moved between fast, slow memory by RMM(. . . , n)} \\ \quad = 8 \cdot W(n/2) + 4 \cdot 3(n/2)^2 \text{ if } 3n^2 > M_{\text{fast}}, \text{ else } 3n^2 \\ \quad = O(n^3 / \sqrt{M_{\text{fast}}}) \dots \text{ same as blocked matmul} \end{cases}$$

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

1/25/21

CS267 Lecture

29



Cache Oblivious Practice

- In practice, cut off recursion well before 1×1
 - 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

30

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

1/25/21 Lecture

31

$O(n^3 / \sqrt{M_{\text{fast}}})$ the minimum data movement possible?

1/26/21

CS267 Lecture

32

Theory: Communication lower bounds

How much data must be transferred?



1/26/21

CS267 Lecture

33

Theory: Communication lower bounds

How much data must be transferred?



1/26/21

CS267 Lecture

34

Theory: Communication lower bounds

How much data must be transferred?



1/26/21

CS267 Lecture

35

Theory: Communication lower bounds

Theorem (Hong & Kung, 1981):

Any reorganization of matmul (using only associativity) has computational intensity $CI = O(\sqrt{M_{fast}})$, so

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



1/26/21

CS267 Lecture

36

Theory: Communication lower bounds

Theorem (Hong & Kung, 1981):

Any reorganization of matmul (using only associativity) has computational intensity $CI = O(\sqrt{M_{fast}})$, so

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



- Cost also depends on the number of “messages” (e.g., cache lines)
 - #messages = $\Omega(n^3 / M_{fast}^{3/2})$
- Tiled matrix multiply (with tile size = $\sqrt{M_{fast}/3}$) achieves this lower bound
- Lower bounds extend to similar programs nested loops accessing arrays

1/26/21

CS267 Lecture

37

Since matmul is flop-limited, can we do better than $O(n^3)$?

1/26/21

CS267 Lecture

38

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 2×2 matrix multiply, normally takes 8 multiplies, 4 adds
 - Strassen does it with 7 multiplies and 18 adds

$$\text{Let } M = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

Let $p_1 = (a_{12} - a_{22}) * (b_{21} + b_{22})$ $p_5 = a_{11} * (b_{12} - b_{22})$
 $p_2 = (a_{11} + a_{22}) * (b_{11} + b_{22})$ $p_6 = a_{22} * (b_{21} - b_{11})$
 $p_3 = (a_{11} - a_{21}) * (b_{11} + b_{12})$ $p_7 = (a_{21} + a_{22}) * b_{11}$
 $p_4 = (a_{11} + a_{12}) * b_{22}$

Then $m_{11} = p_1 + p_2 - p_4 + p_6$ Extends to $n \times n$ by divide&conquer
 $m_{12} = p_4 + p_5$
 $m_{21} = p_6 + p_7$
 $m_{22} = p_2 - p_3 + p_5 - p_7$

1/25/21

CS267 Lecture

39

Strassen (continued)

$T(n)$	= Cost of multiplying $n \times n$ 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

1/25/21

CS267 Lecture

40

Other Fast Matrix Multiplication Algorithms

- World's record was $O(n^{2.37548\dots})$
 - Coppersmith & Winograd, 1987

1/25/21

CS267 Lecture

41

Other Fast Matrix Multiplication Algorithms

- World's record was $O(n^{2.37548\dots})$
 - Coppersmith & Winograd, 1987
- **New Record! 2.37548 reduced to 2.37293**
 - Virginia Vassilevska Williams, UC Berkeley & Stanford, 2011

1/25/21

CS267 Lecture

42

Other Fast Matrix Multiplication Algorithms

- World's record was $O(n^{2.37548\dots})$
 - Coppersmith & Winograd, 1987
- **New Record! 2.37548 reduced to 2.37293**
 - Virginia Vassilevska Williams, UC Berkeley & Stanford, 2011
- **Newer Record! 2.37293 reduced to 2.37287**
 - Francois Le Gall, 2014
- **Latest Record! 2.37287 reduced to 2.37286**
 - Virginia Vassilevska Williams and Josh Alman, 2020

1/25/21

CS267 Lecture

43

Other Fast Matrix Multiplication Algorithms

- World's record was $O(n^{2.37548\dots})$
 - Coppersmith & Winograd, 1987
- **New Record! 2.37548 reduced to 2.37293**
 - Virginia Vassilevska Williams, UC Berkeley & Stanford, 2011
- **Newer Record! 2.37293 reduced to 2.37287**
 - Francois Le Gall, 2014
- **Latest Record! 2.37287 reduced to 2.37286**
 - 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): $\Omega(n^w / M^{(w/2-1)})$

1/25/21

CS267 Lecture

44

Other Fast Matrix Multiplication Algorithms

- World's record was $O(n^{2.37548\dots})$
 - Coppersmith & Winograd, 1987
- New Record! 2.37548 reduced to 2.37293
 - Virginia Vassilevska Williams, UC Berkeley & Stanford, 2011
- Newer Record! 2.37293 reduced to 2.37287
 - Francois Le Gall, 2014
- Latest Record! 2.37287 reduced to 2.37286
 - 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 $Ax=b$, $Ax=\lambda x$, etc) as fast , and stably
 - Demmel, Dumitriu, Holtz, 2008

1/25/21

CS267 Lecture

45

Other Fast Matrix Multiplication Algorithms

- World's record was $O(n^{2.37548\dots})$
 - Coppersmith & Winograd, 1987
- New Record! 2.37548 reduced to 2.37293
 - Virginia Vassilevska Williams, UC Berkeley & Stanford, 2011
- Newer Record! 2.37293 reduced to 2.37287
 - Francois Le Gall, 2014
- Latest Record! 2.37287 reduced to 2.37286
 - 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 $Ax=b$, $Ax=\lambda x$, etc) as fast , and stably
 - Demmel, Dumitriu, Holtz, 2008
- Fast methods (besides Strassen) may need unrealistically large n

1/25/21

CS267 Lecture

46

Basic Linear Algebra Subroutines (BLAS)

- Industry standard interface (evolving)
 - www.netlib.org/blas, www.netlib.org/blas/blast-forum
- Vendors, others supply optimized implementations
- History
 - BLAS1 (1970s): 15 different operations
 - vector operations: dot product, saxpy ($y=Cl*x+y$), root-sum-squared, etc
 - $m=2^n$, $f=2^n$, $q=f/m$ = computational intensity ~1 or less
 - BLAS2 (mid 1980s): 25 different operations
 - matrix-vector operations: matrix vector multiply, etc
 - $m=n^2$, $f=2^2n^2$, $q=2$, less overhead
 - somewhat faster than BLAS1
 - BLAS3 (late 1980s): 9 different operations
 - matrix-matrix operations: matrix matrix multiply, etc
 - $m \leq 3n^2$, $f=O(n^3)$, so $q=f/m$ can possibly be as large as n , so BLAS3 is potentially much faster than BLAS2
- Good algorithms use BLAS3 when possible (LAPACK & ScaLAPACK)
 - See www.netlib.org/{lapack,scalapack}
- More later in the course

1/25/21

CS267 Lecture

47

Basic Linear Algebra Subroutines (BLAS)

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



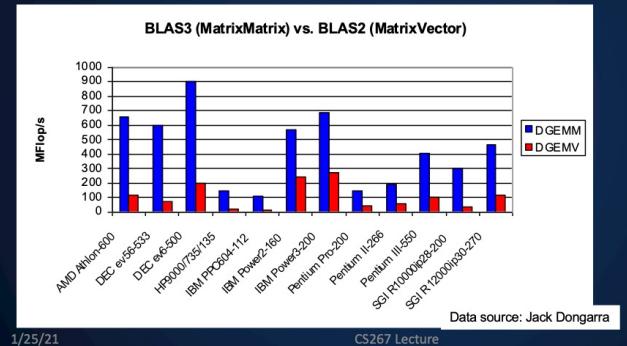
1/26/21

CS267 Lecture

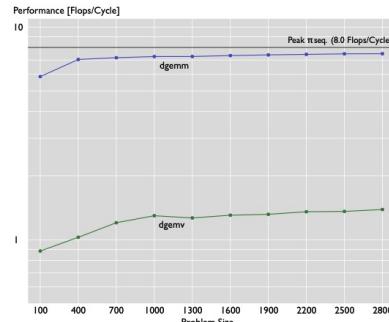
48

Dense Linear Algebra: BLAS2 vs. BLAS3

- Different computational intensity, different performance



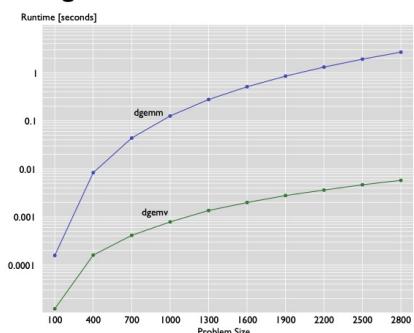
Measuring Performance — Flops/Cycle



Performance gap (flop/sec)

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

Measuring Performance — Runtime



Performance gap (time)

Don't confuse time and rate!

Some reading on MatMul

- Sourcebook on Parallel Computing Chapter 3.
- 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)
- "[Performance Optimization of Numerically Intensive Codes](#)", by Stefan Goedecker and Adolfy Hoisie, SIAM 2001.
- "[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

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

Take-Aways

- Matrix matrix multiplication
 - Computational intensity $O(2n^3)$ flops on $O(3n^2)$ data
- Tiling matrix multiplication (cache aware)
 - Can increase to b if $b \times b$ blocks fit in fast memory
 - $b = \sqrt{M/3}$, the fast memory size M
 - Tiling (aka blocking) "cache-aware"
 - Cache-oblivious
- Optimized libraries (BLAS) exist

1/25/21

CS267 Lecture

53

Roofline Model

How fast can an algorithm go in practice?

1/25/21

CS267 Lecture

54

What is a Performance Model?

A formula to estimate performance

Running time

Bandwidth

Memory footprint

Energy Use

Percent of Peak

55

What is a Performance Model?

A formula to estimate performance

$O(n)$

$f * t_f + m * t_m$

$\text{Lat} + X / \text{BW}$

Examples we've seen for time

56

Why Use a Performance Model?

Understand performance behavior

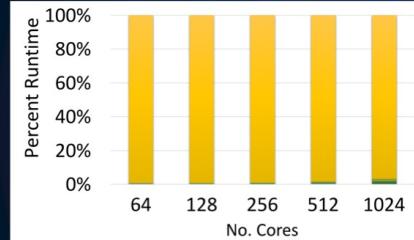
- Differences between Architectures, Programming Models, implementations, etc.



57

Why Use a Performance Model?

- Identify performance bottlenecks



1/25/21

CS194 Lecture

58

Why Use a Performance Model?

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

1/25/21

CS194 Lecture

59

Why Use a Performance Model?

- Determine when we're done optimizing

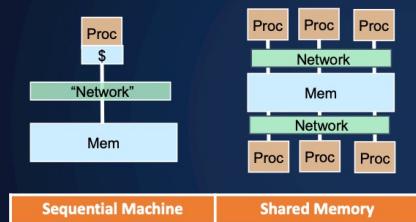


1/25/21

CS194 Lecture

60

Serial and Shared Memory Machines



Critical performance issues

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

History of the Roofline Model



Sam Williams, PhD 2008



1/25/21

CS194 Lecture

62

History of the Roofline Model



Sam Williams, PhD 2008

1774 citations!

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



1/25/21

CS194 Lecture

63

History of the Roofline Model



Sam Williams, PhD 2008

**Roofline
as a verb!**

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!



1/25/21

CS194 Lecture

64

Roofline

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

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

1/25/21

CS267 Lecture

65

What's in the Roofline Model?

Three pieces: 2 for machine and 1 for application

1/25/21

CS194 Lecture

66

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)

1/25/21

CS194 Lecture

67

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)
- Memory bandwidth (bytes /sec)
 - Latency not included (looking at best case)

1/25/21

CS194 Lecture

68

What's in the Roofline Model?

Three pieces: 2 for machine and 1 for application

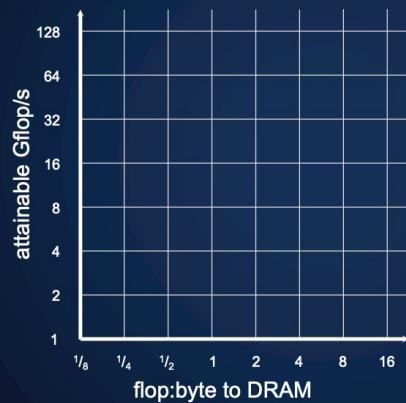
- Arithmetic performance (flops/sec)
 - Clock Speed and Parallelism (ILP, SIMD, Multicore)
- Memory bandwidth (bytes /sec)
 - Latency not included (looking at best case)
- Computational (Arithmetic) Intensity
 - Application balances (flops/word)

1/25/21

CS194 Lecture

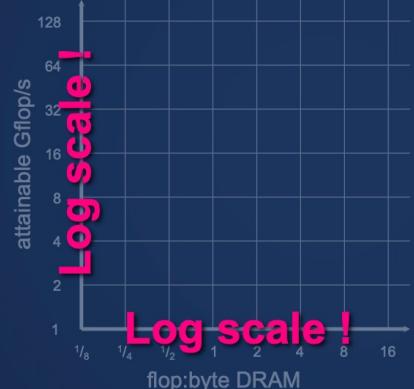
69

The Roofline Performance Model



70

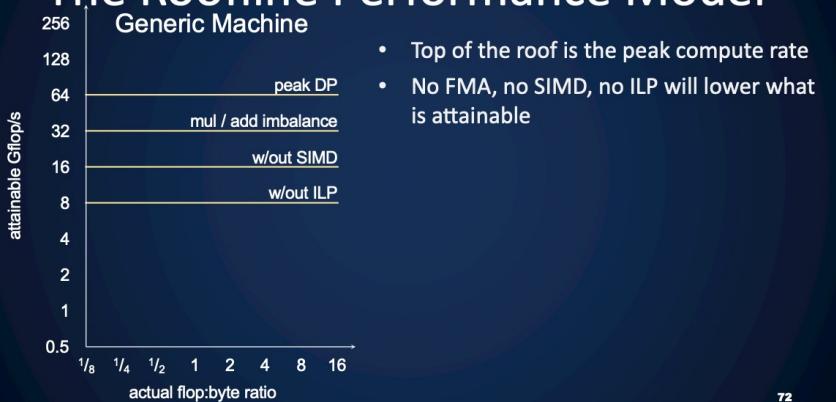
The Roofline Performance Model



71

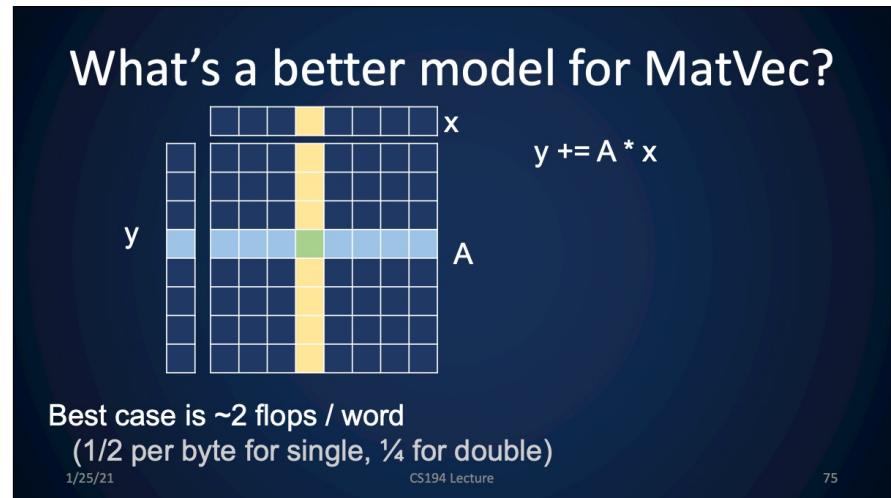
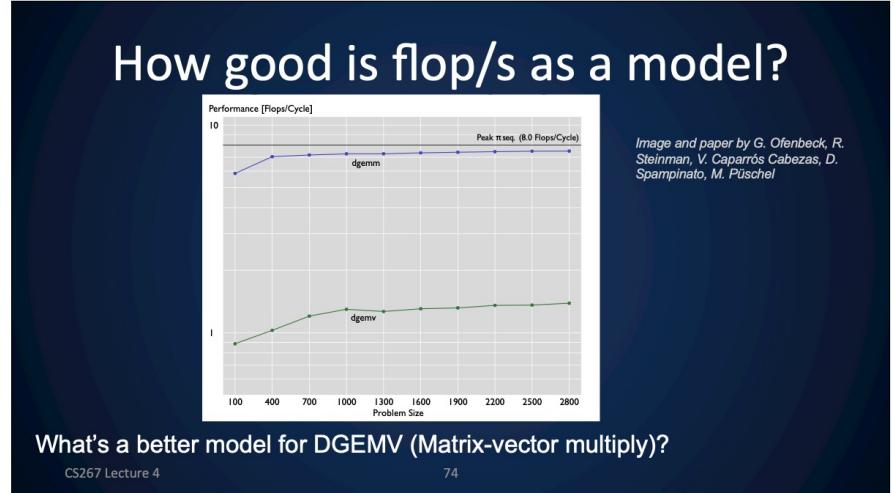
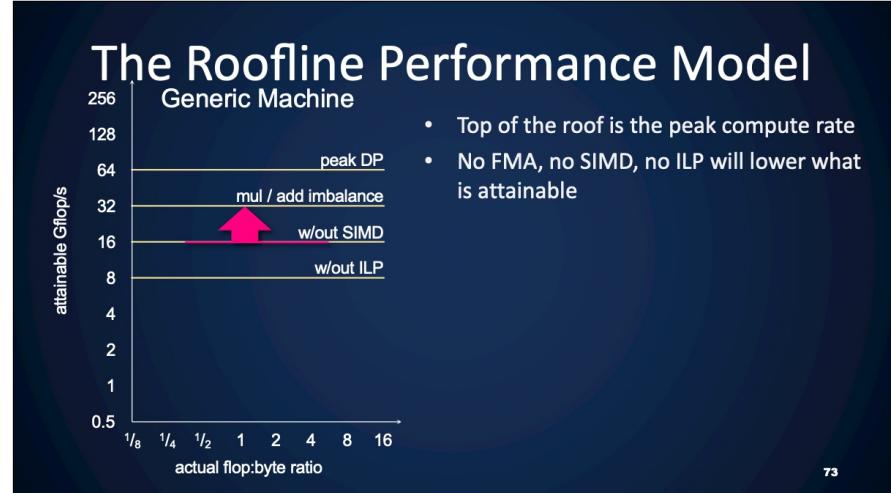
The Roofline Performance Model

Generic Machine



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

72



- ## Data Movement Complexity
- Assume run time \approx data moved to/from DRAM
 - Hard to estimate without cache details
- CS267 Lecture 4
- 76

Data Movement Complexity

- Assume run time \approx 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	$O(n \log n)$	$O(n)$

CS267 Lecture 4

77

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/q) is:

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

Machine Balance and Computational Intensity

- Machine balance is:

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

Ideal
(infinite cache)

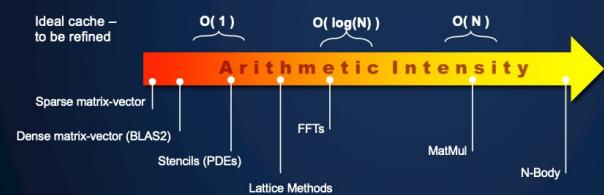
Operation	FLOPs	Data	CI
Dot Prod	$O(n)$	$O(n)$	$O(1)$
Mat Vec	$O(n^2)$	$O(n^2)$	$O(1)$
MatMul	$O(n^3)$	$O(n^2)$	$O(n)$
N-Body	$O(n^2)$	$O(n)$	$O(n)$
FFT	$O(n \log n)$	$O(n)$	$O(\log n)$

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

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

Computational Intensity

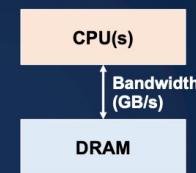
- Can look at computational intensity as a spectrum
- Constants (at least leading constants) will matter



(DRAM) Roofline

Assume

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



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

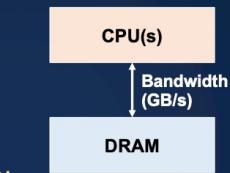
(DRAM) Roofline

Assume

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

$$\text{Time} = \max \left\{ \frac{\#FP \text{ ops}}{\text{Peak GFLOP/s}}, \frac{\#Bytes}{\text{Peak GB/s}} \right\}$$

Why max rather than sum?



(DRAM) Roofline

Assume

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

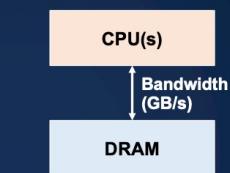
$$\frac{\#FP \text{ ops}}{\text{Time}} = \min \left\{ \frac{\text{Peak GFLOP/s}}{\#Bytes}, (\#FP \text{ ops} / \#Bytes) * \text{Peak GB/s} \right\}$$

(DRAM) Roofline

Assume

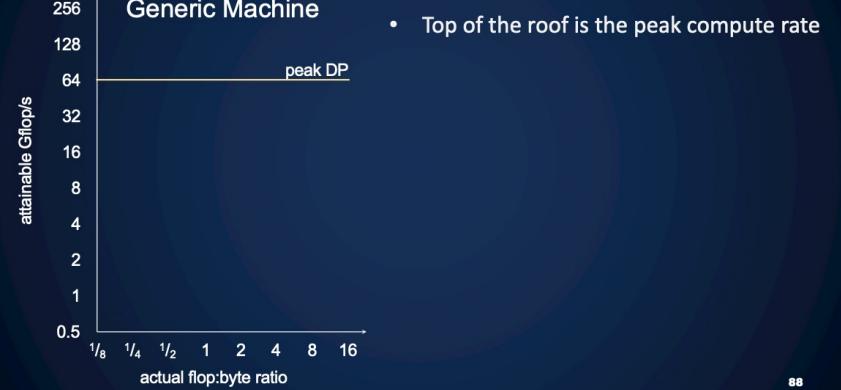
- Idealized processor/caches
- Cold start (data in DRAM)

$$\text{GFlop/sec} = \min \left\{ \frac{\text{Peak GFLOP/s}}{\#Bytes}, (\#FP \text{ ops} / \#Bytes) * \text{Peak GB/s} \right\}$$

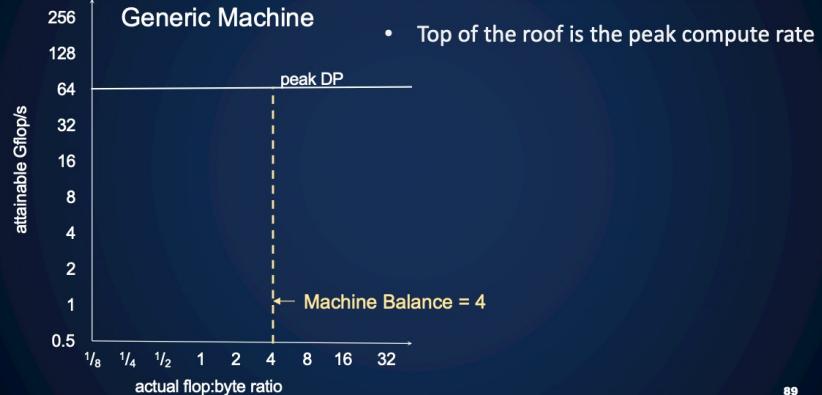


The Roofline Performance Model

Generic Machine

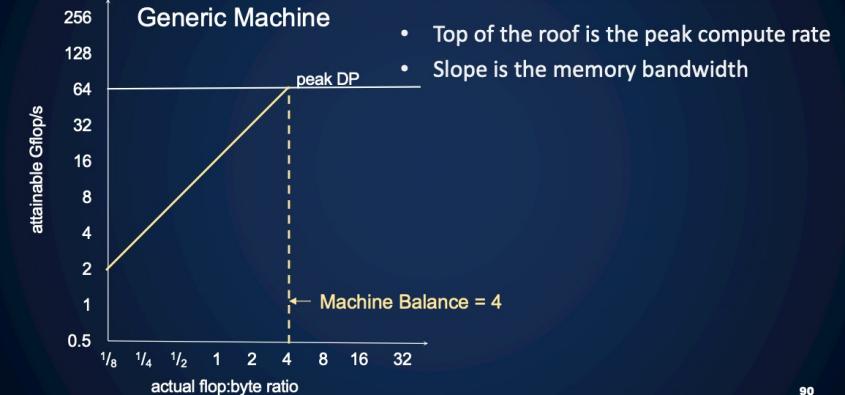


The Roofline Performance Model



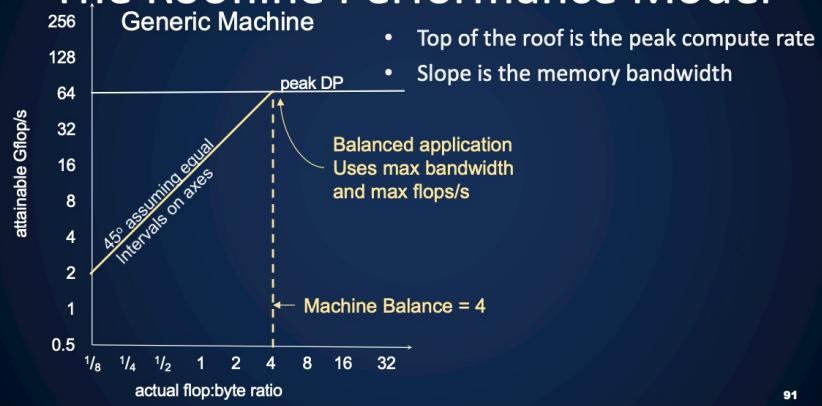
89

The Roofline Performance Model



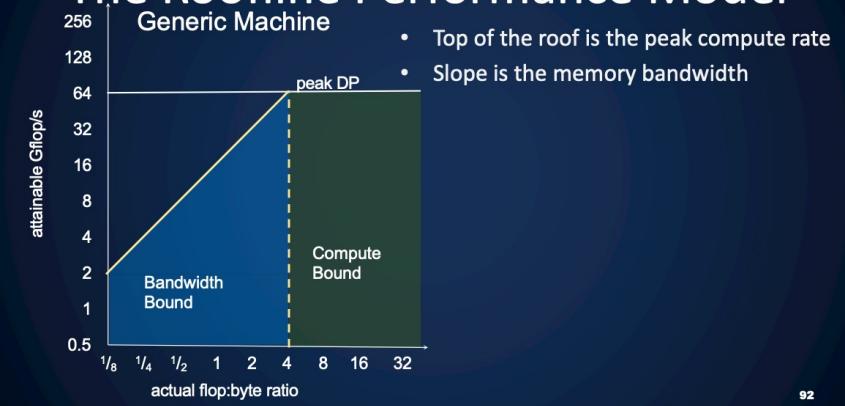
90

The Roofline Performance Model



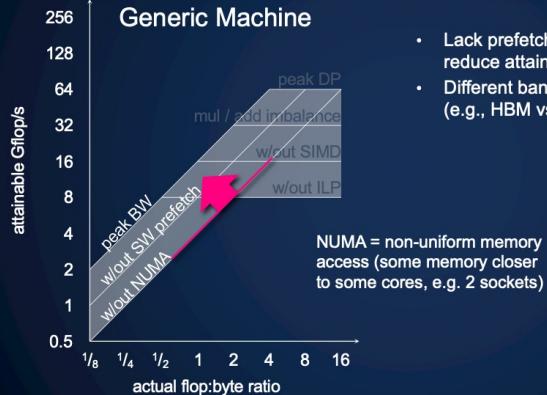
91

The Roofline Performance Model



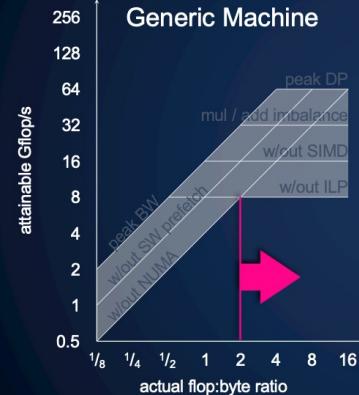
92

The Roofline Performance Model



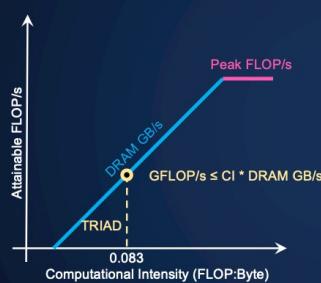
93

The Roofline Performance Model



94

Roofline Example #1

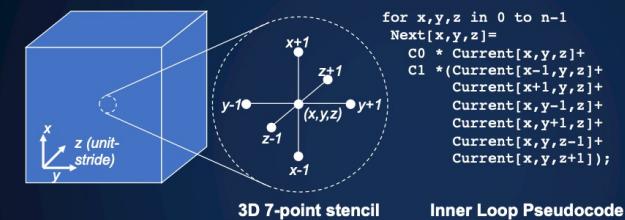


```
#pragma omp parallel for
for(i=0;i<n;i++){
    z[i] = x[i] + alpha*y[i];
}
```

CS267 Lecture 4

95

Roofline Example #2



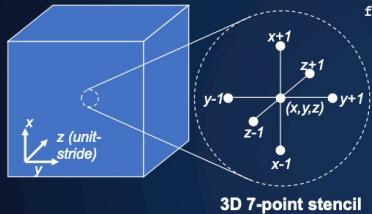
A 7-point constant coefficient stencil...

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

CS267 Lecture 4

96

Roofline Example #2



```
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...

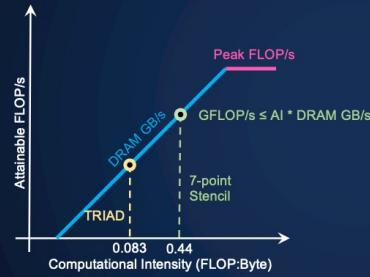
- 7 flops, 8 memory references (7 reads, 1 store) per point
- Cache can filter all but 1 read and 1 write per point
- **CI = 0.44 flops per byte**

CS267 Lecture 4

97

Roofline Example #2

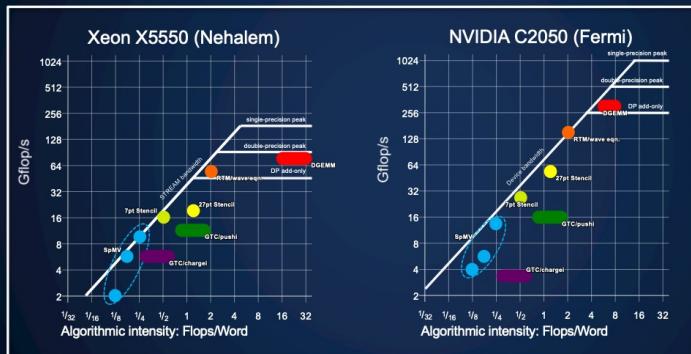
- Still O(1) flops / byte
- **But (leading) constants matter**



CS267 Lecture 4

98

Roofline Across Algorithms



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

99

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 SPMs
- Widely used in practice and adapted to any bandwidth/compute limit situation

1/26/21

CS267 Lecture

100