KOÇ UNIVERSITY

Performance Modeling

Didem Unat

COMP 429/529 Parallel Programming

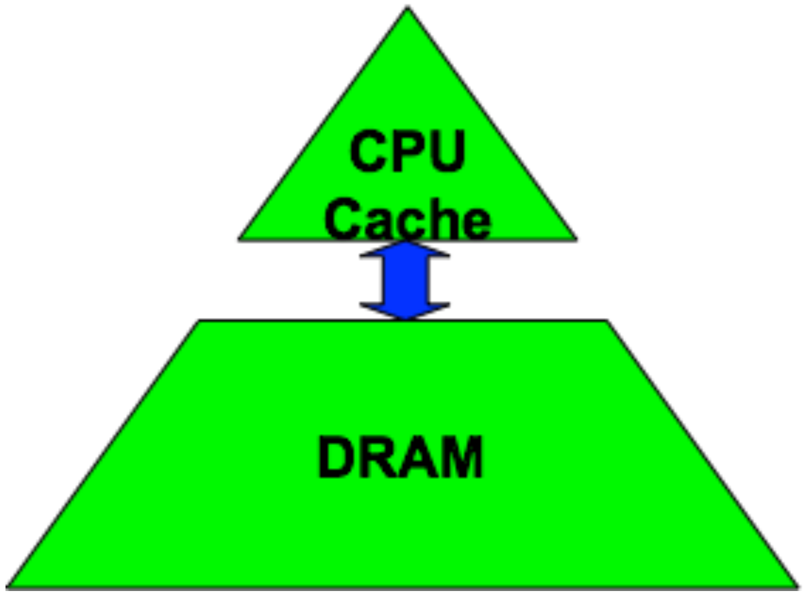
Why model performance?

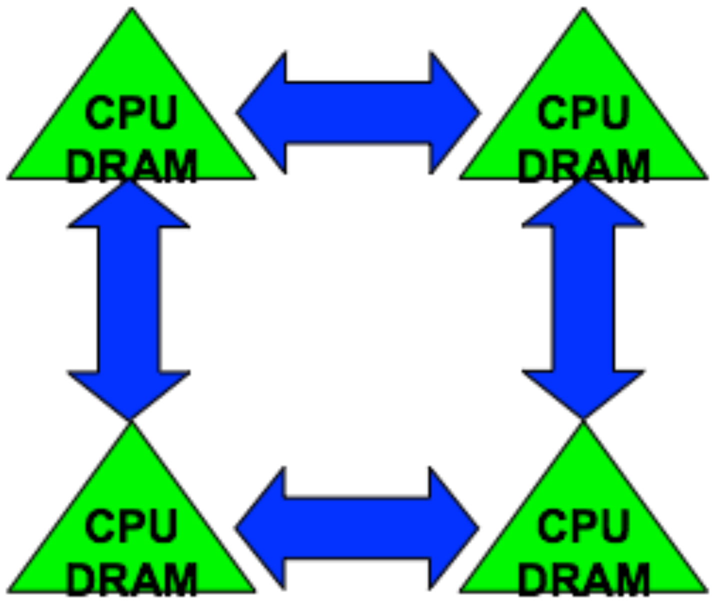
- Identify bottlenecks in an application
- Focus on the most time-consuming regions of an application
- Reason about the capabilities of underlying architecture

- Remember, we are interested in performance (speedup) at the and of the day

Data Movement

- Algorithms have three costs (two of them can be classified under one)
  - Arithmetic (Flops)
  - Communication: moving data between levels of memory hierarchy and processors over a network





Modeling Performance

- At any given time, a processor can be busy with one of the following:
  - Computation
  - Communication
    - Data access (to memory or cache)
    - Communication over network
  - Disk I/O

- A simple model for total execution time (excluding IO)
  - T_{total} = Compute time + Memory time + Communication time

    $= T_F + T_M + T_C$

Performance Model

- $T_{total} = T_F + T_M + T_C$
- Definitions:
  - f = total # of floating point arithmetic (flop) operations
  - t_f = time per flop << t_m
  - m = total # of memory elements (words) moved between fast and slow memory
  - t_m = time per slow memory access
  - c = total # of bytes communicated between nodes
  - t_c = average time for communicating one byte
- $T_{total} = f \cdot t_f + m \cdot \mathbf{t_m} + \mathbf{c} \cdot \mathbf{t_c}$

Overlaps

- Different operations can be overlapped with each other
- Prefetching:
  - overlapped computation and memory access
  - hardware/software prefetching
- Non-blocking communication primitives:
  - overlapped computation and communication
- Algorithmically overlapping computation/communication:
  - designate computation & communication threads
  - can still use blocking communication primitives

*-bound

- $T_{total} = f * t_f + m * t_m + c * t_c$
- If $T_{total} \approx c * t_c$ => performance is **communication-bound**
  - Performance improvements may be possible
  - Ex: graph traversals, sparse matrix computations
- If $T_{total} \approx m * t_m$ => performance is **memory-bound**
  - Data locality/access patterns may be improved
  - Ex: stencil applications
- If $T_{total} \approx f * t_f$ => performance is **CPU-bound**
  - CPU performing close to its peak – as good as one can get
  - Ex: dense matrix computations

Arithmetic Intensity

- How many flops performed per memory access in an application?
- Assumptions:
  - Only 2 levels in the memory hierarchy

    fast (cache) and **slow** (memory)
  - All data are initially in the slow memory
- **Arithmetic Intensity: q = f / m**
  - average number of flops per slow memory access
  - also known as flop:byte ratio (flop to byte ratio)

FLOPS-to-Byte Ratio of a Machine

Also called **Machine Balance**=

Flops:Bytes = Peak Flop Rate / Peak Memory Bandwidth

- Here peak can be replaced with sustained (or measured values).
- Vendors like Intel or AMD usually announce the peak flop rate of a chip
  - e.g. 1 Teraflops etc.
- Memory bandwidth
  - e.g 200 GB/sec etc
- The machine is considered to be *imbalance* if the flops:bytes is too high or too low
  - Usually high these days because it is harder to provide more memory bandwidth ($)
  - Performing computation is cheap

Memory Intensive or Compute Intensive?

- If Machine Balance > Arithmetic Intensity of an Application
  - Then the application is memory bandwidth limited
  - Otherwise the application is compute-bound

- Memory intensive
  - means that the application needs faster data transfer rate than the machine can provide thus memory hierarchy creates a performance bottleneck
    - If we had more bandwidth, our performance would increase
- Compute-intensive
  - means that the application needs faster compute rate than the machine can provide thus the CPU speed creates a performance bottleneck
    - If we had a higher clock rate, our performance would increase

GPU Tesla K40

- Peak double precision floating point performance
  - 1.43 Teraflops (base clock)
  - 1.66 Teraflops (GPU boost)
- Memory Bandwidth (Sustained)
  - ˜184 GB/sec
  - Can achieve higher with tuning (or ECC off)
- Machine Balance
  - 184/1043 = 0.18 (byte:flop ratio)
  - or the machine can perform ˜5.6 flops per byte

Example Application

- A simulation performs 14 flops per data point
  - Note that some operations (e.g. division) are more expensive than other flops but let's ignore that for now
- Per data point, it performs 3 memory accesses at best

  3 * 8 bytes = 24 bytes
- q = f/m = 14/24 = 0.583 (flop to byte ratio)

Limitations to Performance

- Compare flops:byte
  - Tesla K40 can perform 5.6 flops per byte.
  - The application only needs 14/24 = 0.583 flops per byte
- or bytes:flop
  - The application needs 24/14 = 1.71 bytes per flop
  - Tesla K40 can only bring 0.18 bytes per flop

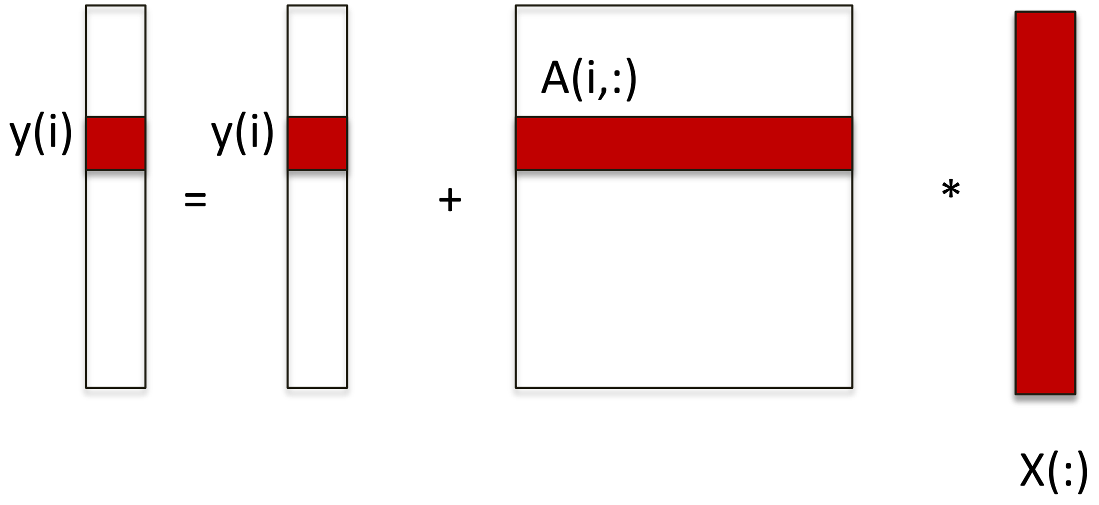
- This application is clearly memory bandwidth-limited. If we had more bandwidth, the app will perform better!

Case Study with Matrix-Vector Multiply

Case Study: Matrix-Vector Multiply

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

- how many bytes accessed at best?
- how many flops performed?



n	n	n^2	n

Case Study: Matrix-Vector Multiply

```
{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}
```

- m = number of bytes accessed on slow memory = $3n + n^2$ doubles
- f = number of arithmetic operations = $2n^2$
- q = f / m ≈ 2 / 8 (in double precision)

=> Matrix-vector multiplication limited by the slow memory speed, because in general $t_f << t_m$

Case Study: Matrix-Vector Multiply

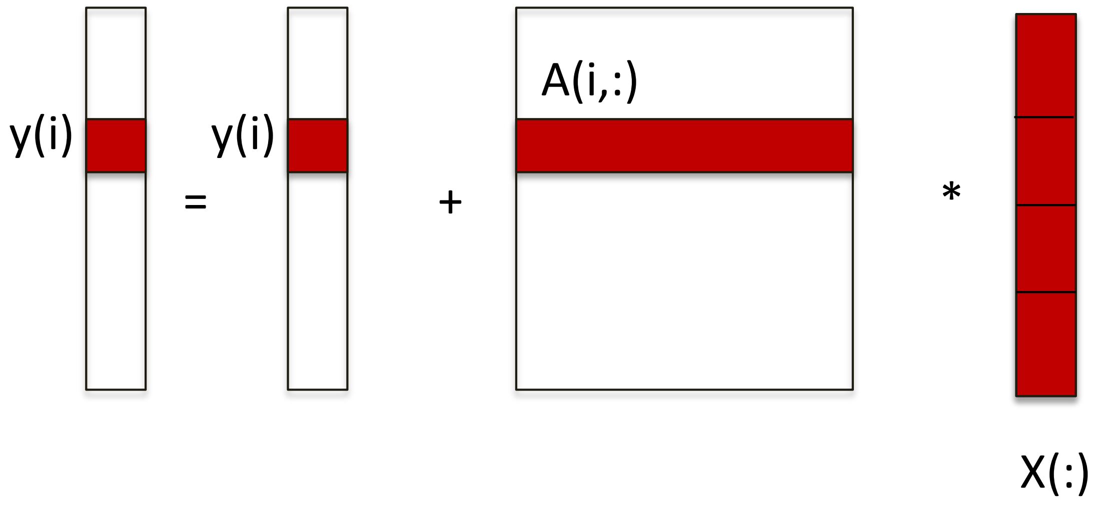
```
{read x(1:n) into fast memory}
{read y(1:n) into fast memory}
for i = 1:n
    {read row i of A into fast memory}
    for j = 1:n
        y(i) = y(i) + A(i,j)*x(j)
{write y(1:n) back to slow memory}
```

- **Important assumptions:**
  - Fast memory is large enough to hold 3n doubles
  - Time to access data in fast memory costs nothing!

Case Study: Matrix-Vector Multiply

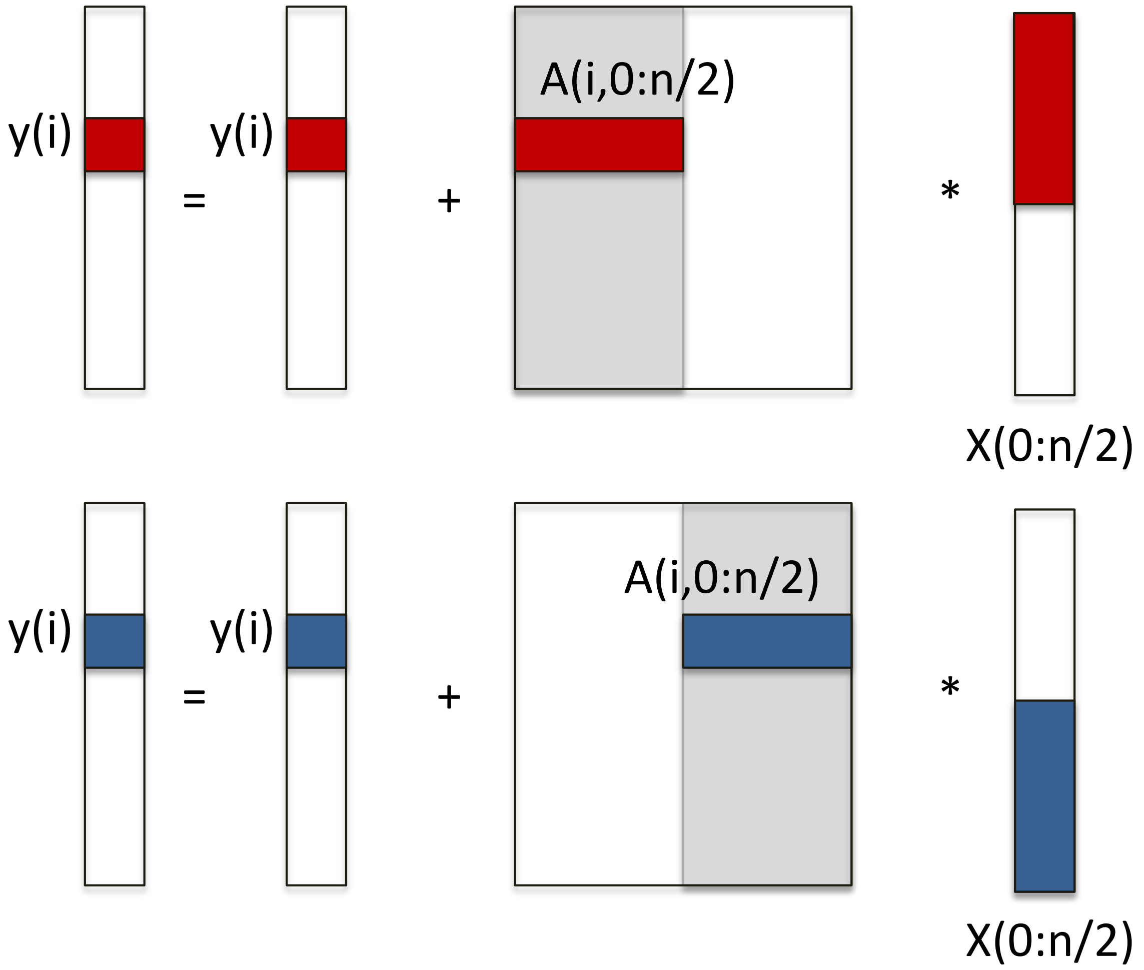
- How many cache misses would such an implementation incur on x, if the **cache size was < n** and Least Recently Used replacement policy is used?

n^2/w **w**:cache-line width
For every row of A, n/w misses occur
Is it possible to do better?



Case Study: Matrix-Vector Multiply

- A blocked algorithm should do better



Case study: Matrix-Vector multiply

Tiling (Cache Blocking)

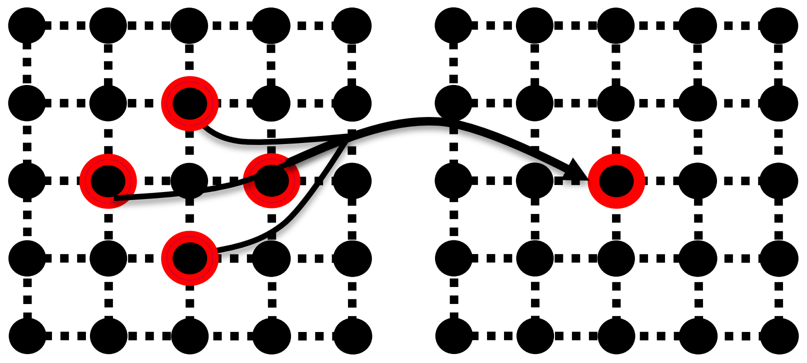
- Tiling or cache blocking is a well-known cache optimization to reduce cache misses
- Block the data into tiles and process each tile before computing the next tiles

With cache-blocking (tiling):

- read each element of **A** once from slow memory
- read each element of **x** once from slow memory
- read & write each element of y **twice** from/to memory
  - Twice because they are two blocks (in this example)
- total cache misses on x: n/w
- total cache misses on y: 2n/w

Case Study: 5-point Stencil

```
//image smoothing example using 5-point stencil
for iter = 1 : nSmooth
  for (i,j) in (0:N-1, 0:N-1)
       Imgnew [i,j] = (Img[i-1,j]+Img[i+1,j]+Img[i,j-1]+Img[i, j+1])*.25
 Swap(Imgnew,Img)
```

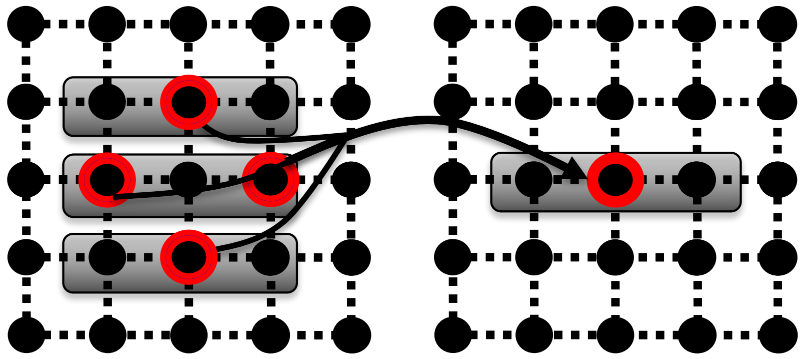


Img ImgNew

- Four loads and one store from/to memory
  - 5 accesses per element
- 3 Adds + 1 MUL
  - 4 flops per element

5-point Stencil

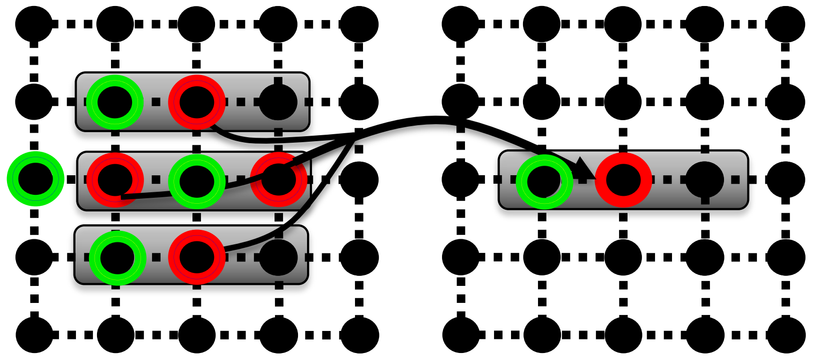
```
//image smoothing example using 5-point stencil
for iter = 1 : nSmooth
  for (i,j) in (0:N-1, 0:N-1)
       Imgnew [i,j] = (Img[i-1,j]+Img[i+1,j]+Img[i,j-1]+Img[i, j+1])*.25
 Swap(Imgnew,Img)
```



- So only three of the four loads count for the bytes move
  - In the middle row, two accesses are in the same cache-line
- Bytes:flops ratio then
  - 4 words / 4 flops = 1 word

5-point Stencil

```
//image smoothing example using 5-point stencil
for iter = 1 : nSmooth
  for (i,j) in (0:N-1, 0:N-1)
       Imgnew [i,j] = (Img[i-1,j]+Img[i+1,j]+Img[i,j-1]+Img[i, j+1])*.25
 Swap(Imgnew,Img)
```

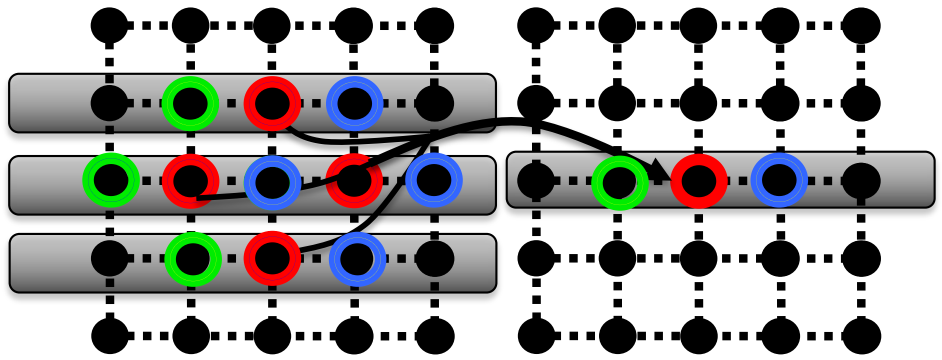


Img   ImgNew

- Row can stay in the cache for three successive row traversals if the cache is large enough to hold more than two rows

5-point Stencil

```
//image smoothing example using 5-point stencil
for iter = 1 : nSmooth
  for (i,j) in (0:N-1, 0:N-1)
       Imgnew [i,j] = (Img[i-1,j]+Img[i+1,j]+Img[i,j-1]+Img[i, j+1])*.25
 Swap(Imgnew,Img)
```

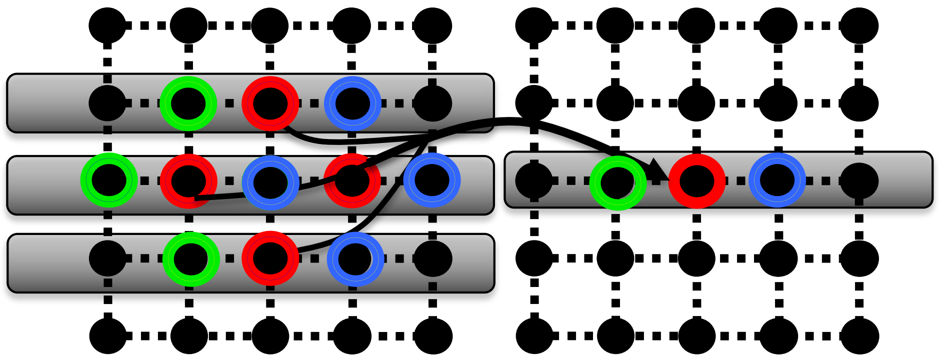


Img ImgNew

- Row can stay in the cache for three successive row traversals if the cache is large enough to hold more than two rows

5-point Stencil

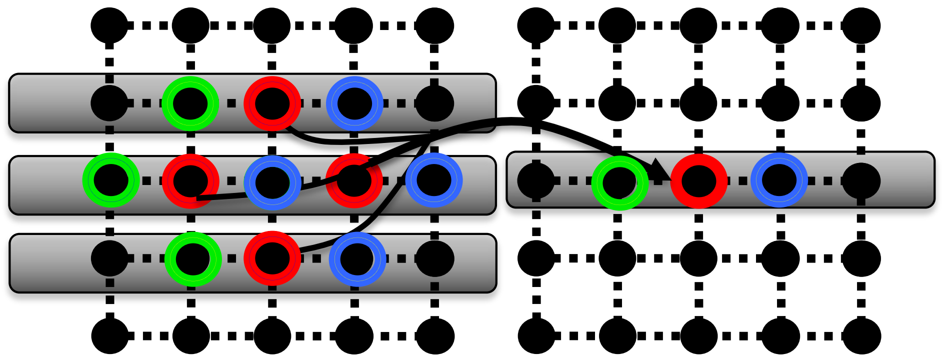
```
//image smoothing example using 5-point stencil
for iter = 1 : nSmooth
  for (i,j) in (0:N-1, 0:N-1)
       Imgnew [i,j] = (Img[i-1,j]+Img[i+1,j]+Img[i,j-1]+Img[i, j+1])*.25
 Swap(Imgnew,Img)
```



- Under this condition, we can assume that loading the neighbors at row i-1 and i+1 comes at no cost
- We have already loaded i-1 while computing previous row and i+1 will be reused when we compute the next row.

5-point Stencil

```
//image smoothing example using 5-point stencil
for iter = 1 : nSmooth
  for (i,j) in (0:N-1, 0:N-1)
       Imgnew [i,j] = (Img[i-1,j]+Img[i+1,j]+Img[i,j-1]+Img[i, j+1])*.25
 Swap(Imgnew,Img)
```



Img   ImgNew

We require only one load and one store

Bytes:Flops = 2 words/4 flops= 0.5 words

However, when the image is large and three successive rows cannot fit into cache, then we are back to 1 words/flop

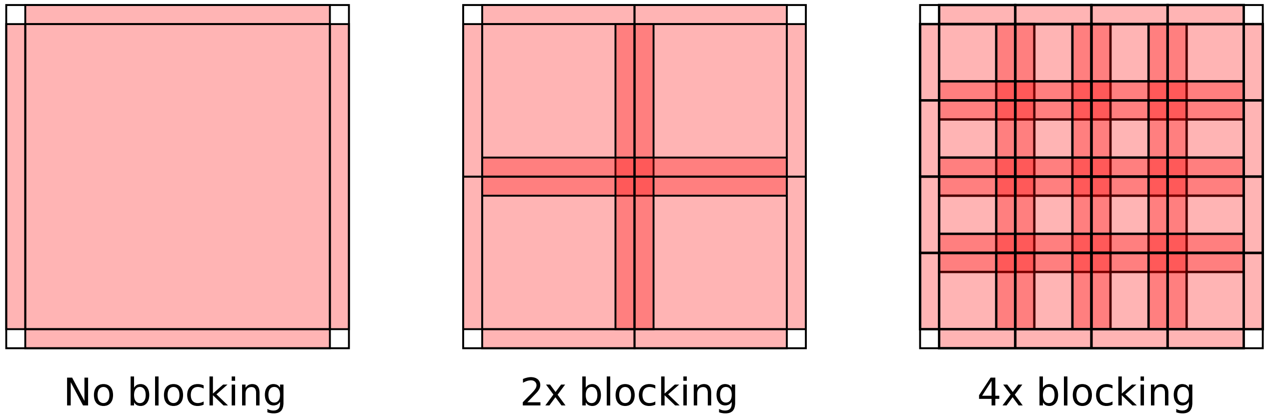
5-point Stencil

```
//image smoothing example using 5-point stencil
for iter = 1 : nSmooth
  for (i,j) in (0:N-1, 0:N-1)
       Imgnew [i,j] = (Img[i-1,j]+Img[i+1,j]+Img[i,j-1]+Img[i, j+1])*.25
 Swap(Imgnew,Img)
```

- If image size is N, each element is 4 bytes, then **at best**, we move (in reality, it might be worse)
  - Bytes_Moved = N*N*2*4 bytes
  - Sustained Memory Bandwidth = Bytes_Moved/Running Time
  - Compare this with the machine bandwidth rate (Stream benchmark) to see how good or bad you are doing
- If the data doesn’t fit into cache, then apply tiling

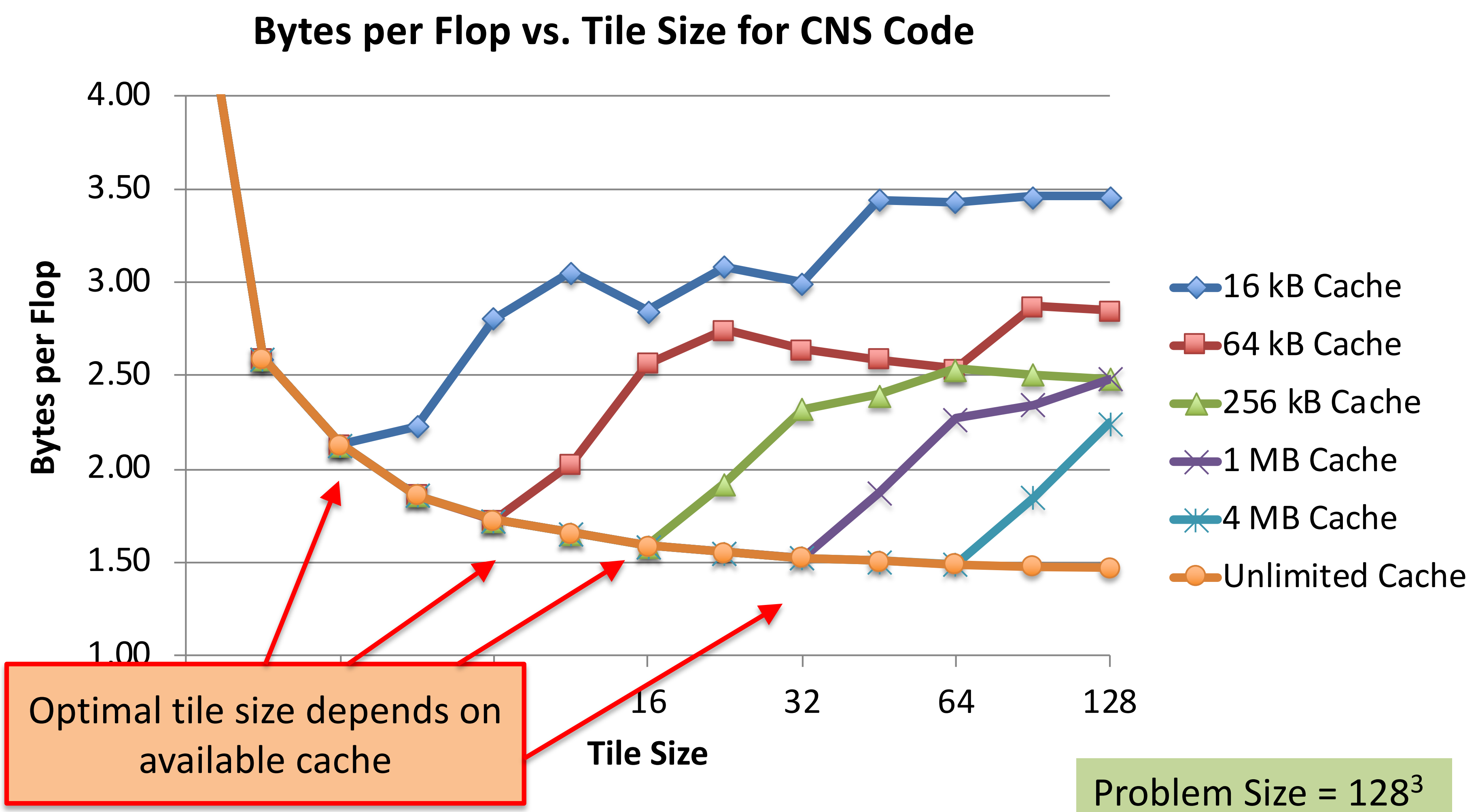
Cache Blocking (Tiling)

- Cache blocking exposes trade-off between cache size and memory bandwidth:
  - PRO: Smaller working set –allows working set to fit into cache, enabling reuse
  - CON: Redundant memory traffic –pulls overlapping ghost zones in from memory
  - Important parameter to tune: tile size (cache block size)



Cache and Tile Size Effects on Memory Traffic

An Example application: Choosing block size based on your cache size and working set



Problem Size = 128^3

Didem Unat, Cy Chan, Weiqun Zhang, Samuel Williams, John Bachan, John Bell, John Shalf, "ExaSAT: An Exascale Co-Design Tool for Performance Modeling", International Journal of High Performance Computing Applications (IJHPCA), February, 2015

Acknowledgments

- These slides are inspired and partly adapted from
  - Metin Aktulga (Michigan State Univ.)
  - Scott Baden (UCSD)
  - Jim Demmel from UC Berkeley
  - Cy Chan, Didem Unat (ExaSAT journal)
  - The course book (Pacheco)
  - Didem Unat, Cy Chan, Weiqun Zhang, Samuel Williams, John Bachan, John Bell, John Shalf, "ExaSAT: An Exascale Co-Design Tool for Performance Modeling", International Journal of High Performance Computing Applications (IJHPCA), February, 2015