

CS475

Cache Locality

Wim Bohm, CS, CSU

sources: Wikipedia,
3C paper: Hill and Smith: "Evaluating Associativity in CPU caches"
Roofline paper: Patterson et.al. Parallel Computing Lab, Berkeley

Cache

- A **cache** is a device that transparently stores data so that future requests for that data can be served faster.
- E.g. an on (CPU) chip cache can be accessed in a few clock cycles, while accessing local memory takes 100s of cycles.

Locality

- Caches exploit locality:
 - **Temporal locality:** if a particular memory location is accessed, it is likely to be accessed in the near future.
 - **Spatial locality:** if a particular memory location is accessed, then it is likely that nearby memory locations will be accessed in the near future.
 - **Equidistant locality:** mixture of the two above: if a particular memory location is accessed, then it is likely that memory locations in an equidistant pattern will be accessed in the near future.

Cache operation

- 🌐 When the CPU accesses memory location x , the cache checks if it has x .
 - 🌐 If so, the memory access is avoided. We call this a **hit**.
 - 🌐 If not, x is fetched from memory and stored in the cache. We call this a **miss**.
- 🌐 Memory is not fetched one word at the time, but in **cache lines or blocks**, for higher memory bandwidth, and good spatial locality.

Cache replacement policy

- 🌐 caches are smaller than local memories, they fill up quickly, and therefore a replacement policy is needed.
- 🌐 The **heuristic** that it uses to choose the entry to **evict** is called the replacement policy. The fundamental problem with any replacement policy is that it must predict which existing cache entry is least likely to be used in the near future.
- 🌐 A popular replacement policy, least-recently used (**LRU**), replaces the least recently accessed entry.

Associativity

- 🌐 The replacement policy decides **where** in the cache a copy of a particular entry of main memory will go.
- 🌐 If the replacement policy is free to choose any entry in the cache to hold the copy, the cache is called **fully associative**.
- 🌐 At the other extreme, if each entry in main memory can go in just one place in the cache, the cache is **direct mapped**.
- 🌐 Many caches implement a compromise in which each entry in main memory can go to any one of N places in the cache, and are described as **N -way set associative**.

Sources of cache misses: the 3C model

- 🌐 **Compulsory:** On the first access to a block; the block must be brought into the cache; also called cold start misses, or first reference misses.
- 🌐 **Capacity:** Occur because blocks are being discarded from cache because cache cannot contain all blocks needed for program execution (program working set is much larger than cache capacity).
- 🌐 **Conflict:** In the case of set associative or direct mapped block placement strategies, conflict misses occur when several blocks are mapped to the same set or block frame; also called collision misses or interference misses.

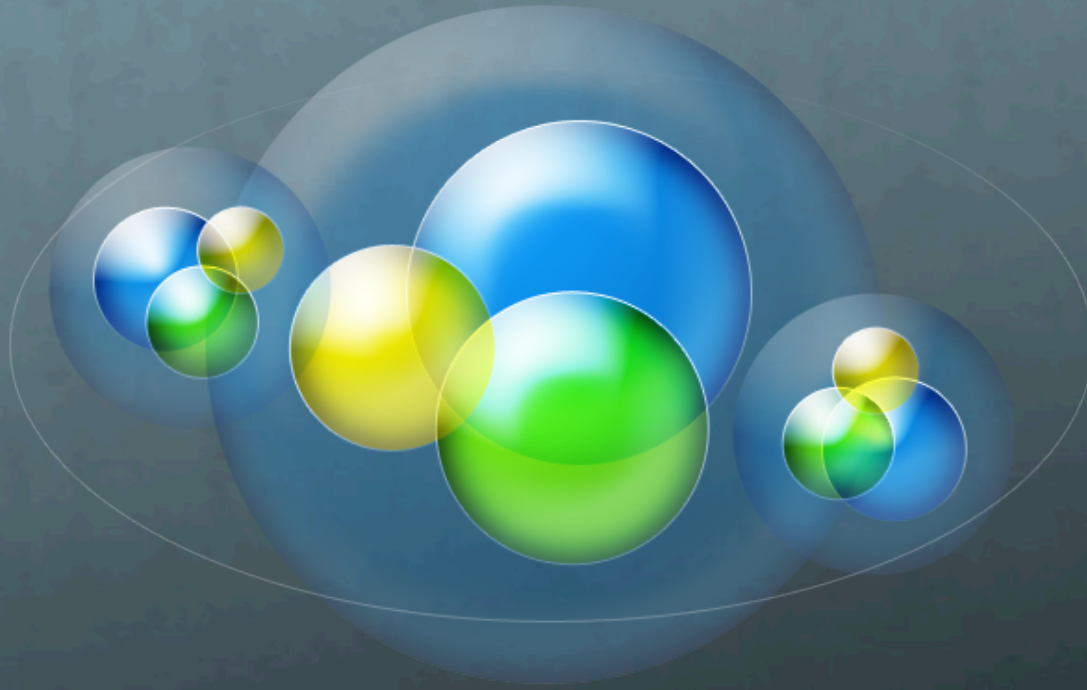
Improving cache performance: hardware

- 🌐 Increased cache capacity
- 🌐 higher associativity
- 🌐 hardware prefetching of instructions and data
 - 🌐 equidistant locality
- 🌐 second-level / third level cache (L2, L3)
 - 🌐 L3 often shared by multiple cores
 - 🌐 there is a difference in access time between L1, L2, L3
- 🌐 out of order instruction execution
- 🌐 branch prediction

All this makes modern CPUs highly complex.

Improving cache performance: software

- 🌐 **Merging Arrays:** Improve spatial locality by single array of structs vs. parallel arrays (Fortran).
- 🌐 **Loop Interchange:** Change nesting of loops to access data in the order stored in memory.
- 🌐 **Loop Fusion:** Combine 2 or more independent loops that have the same looping and some variables overlap.
- 🌐 **Blocking or “tiling”** : Improve temporal locality by accessing “blocks” of data repeatedly vs. going down whole columns or rows. (prime sieve)



Matrix Multiply

Data or loop reordering for improve cache performance

Matrix multiply:

for i = 1 to n

for j = 1 to n

$C[i,j]=0$

for k = 1 to n

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

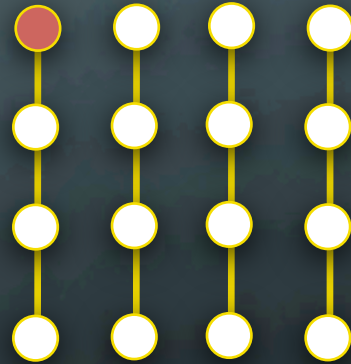
B is accessed in column order. If arrays are (as in C) stored in row major order, cache lines are not helping, which can cause cache misses for all Bs.

Solution: transpose B

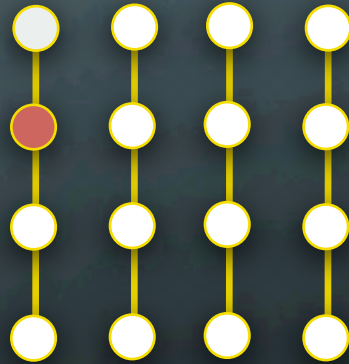
Tiling

- Instead of reading a whole row of A and doing n whole row A column B inner products we can read a block of A and compute smaller inner products with sub columns of B .
- These partial products are then added up.

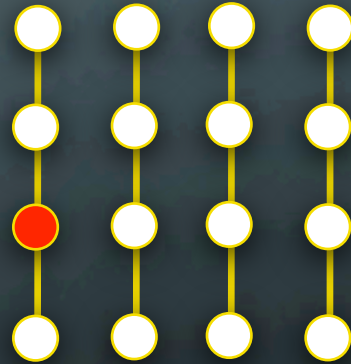
Conventional matrix multiply



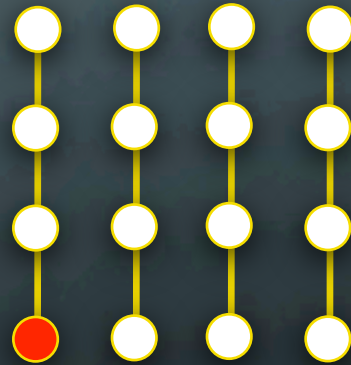
Conventional matrix multiply



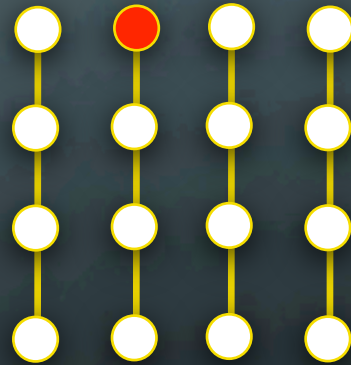
Conventional matrix multiply



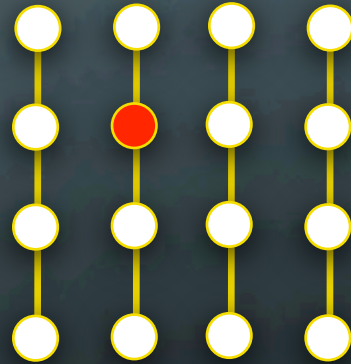
Conventional matrix multiply



Conventional matrix multiply



Conventional matrix multiply



etc.

Conventional matrix multiply



All elements of B are used once, while all of row $A[i]$ are used n times.

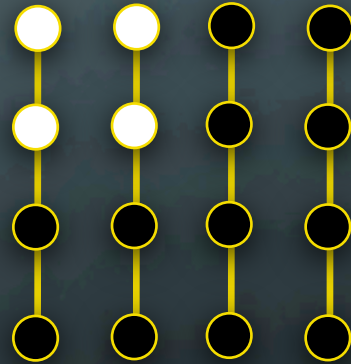
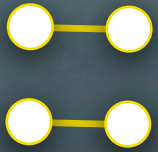
$A[i,*]$ may fit in the cache, B will probably not!

Tiling A and B

- A $k \times k$ tile of A (which can fit in the cache) block multiplies with a $k \times k$ tile of B (which can fit in the cache) and thus reuses the B tile **k times**, better cache use
- Loops become nested loops
 - outer loop visits **tile origins**
 - inner loops visit the **tile points**
- We can parameterize our program with k and experiment

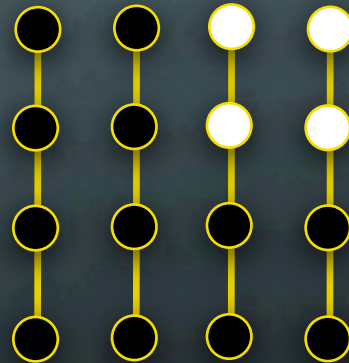
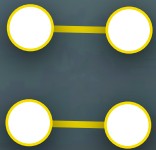
Tiled matrix multiply

Do the whole block $A_{11} \times B_{11}$ multiply

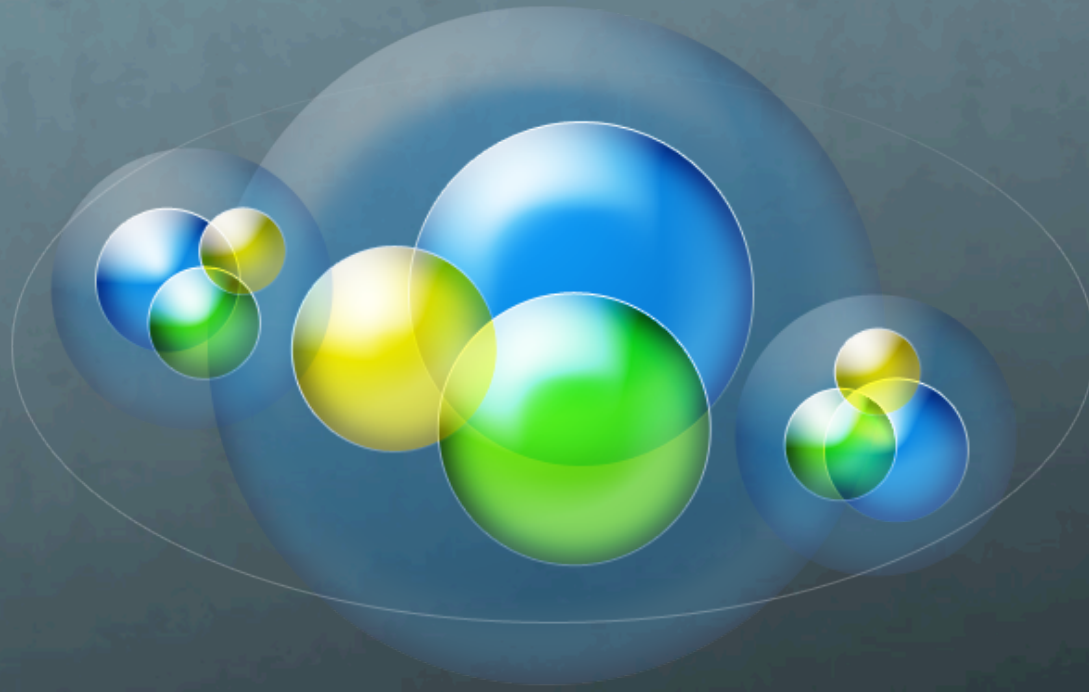


Tiled matrix multiply

The do block $A_{11} \times B_{12}$ multiply
How many times are A and B elements used now?



etc.



Knapsack

Knapsack Problem: Bottom-Up Dynamic Programming

- 🌐 Knapsack. Fill an n -by- W array.

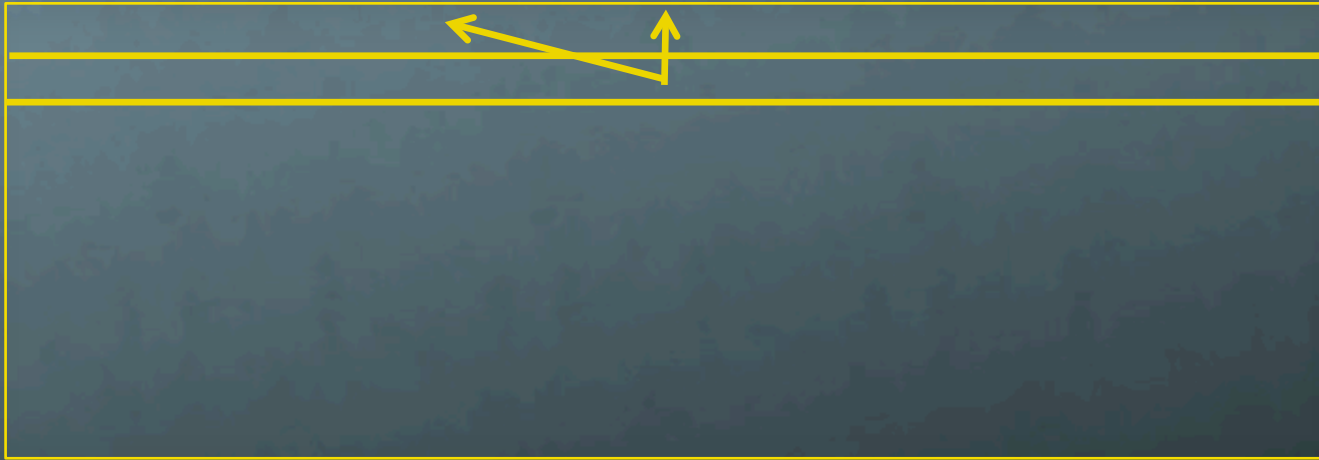
```
Input:  $n, W, w_1, \dots, w_N, v_1, \dots, v_N$ 

for  $w = 0$  to  $W$ 
     $M[0, w] = 0$ 

for  $i = 1$  to  $n$ 
    for  $w = 0$  to  $W$ 
        if  $w_i > w$  :
             $M[i, w] = M[i-1, w]$ 
        else :
             $M[i, w] = \max (M[i-1, w], v_i + M[i-1, w-w_i ])$ 

return  $M[n, W]$ 
```


Knapsack data dependence

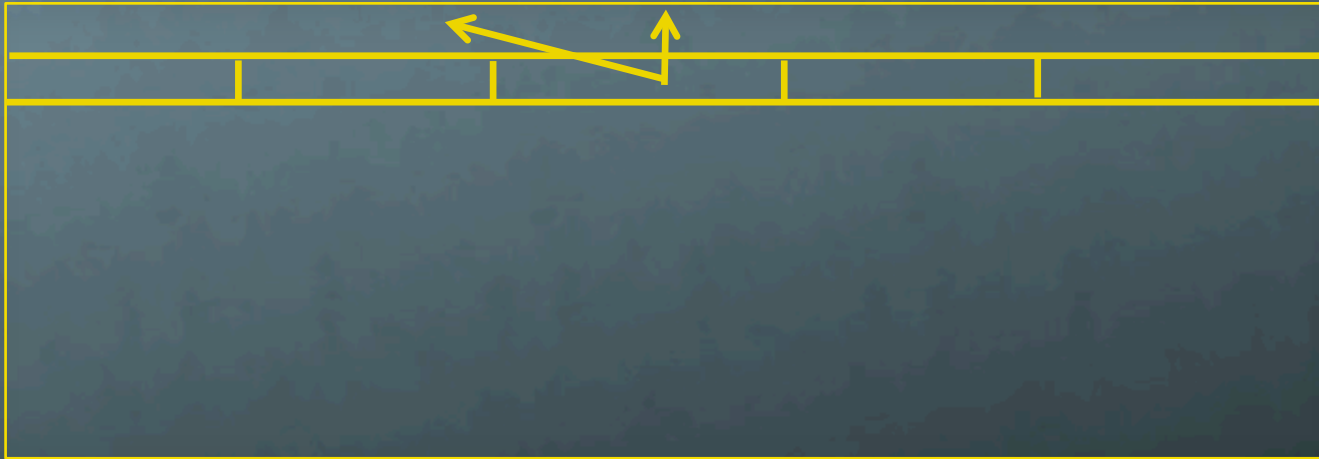


$M[i, w]$ depends on $M[i-1, w]$ and $M[i-1, w-w_i]$

How can we parallelize knapsack?

there are no in row dependences

Knapsack parallelization



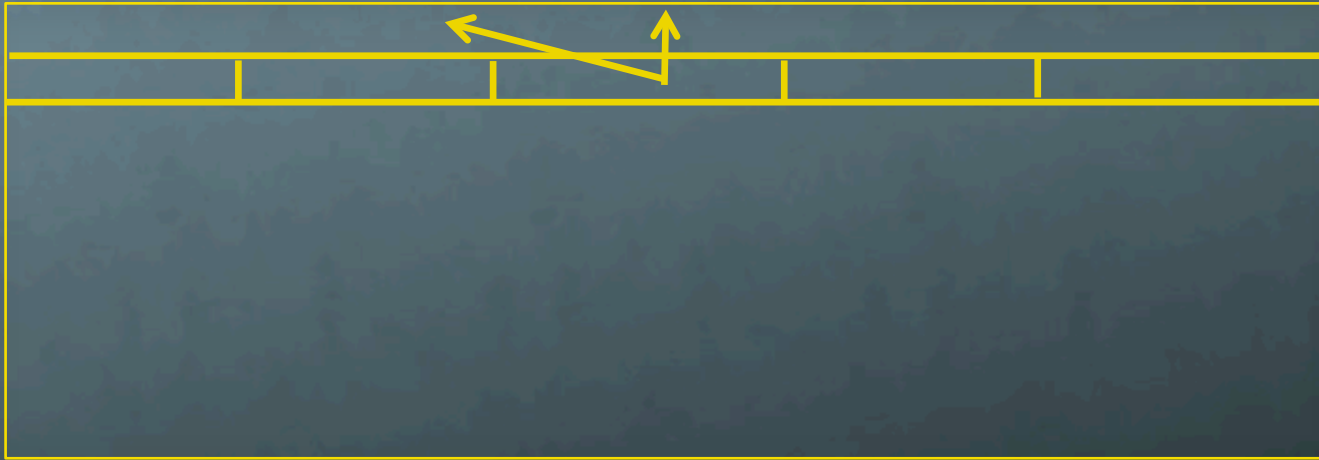
$M[i, w]$ depends on $M[i-1, w]$ and $M[i-1, w-w_i]$

We can parallelize the inner loop

What would be the problems?

cache, many spawns

Tiling Knapsack parallelization



$M[i, w]$ depends on $M[i-1, w]$ and $M[i-1, w-w_i]$

We can tile (as in prime sieve)

What would be the dependences between tiles?

$\text{tile}_{i,j}$ depends on tiles left of it and above it

Knapsack wavefront block parallelization



Knapsack wavefront block parallelization



Knapsack wavefront block parallelization



Knapsack wavefront block parallelization



Knapsack wavefront block parallelization



Knapsack wavefront block parallelization



Knapsack wavefront block parallelization



Knapsack wavefront block parallelization



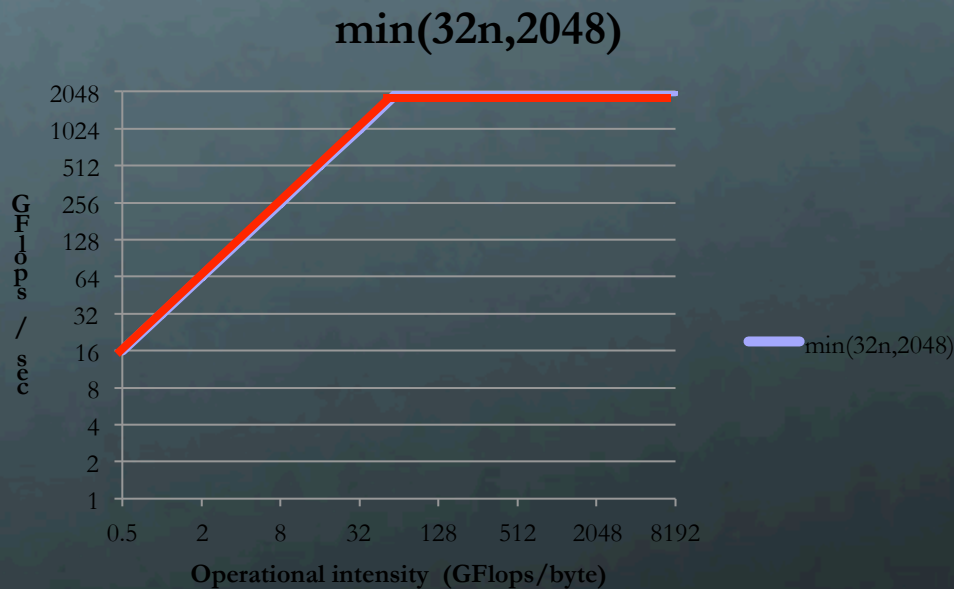
Roofline Model

- Architectural model*, based on intuition that **off-chip memory bandwidth** is the constraining resource.

*:David Patterson et.al. Parallel Computing Lab, Berkeley

- Operational Intensity**: flops per byte of memory traffic, i.e. bytes exchanged between cache(s) and memory.
- Roofline plots Gflops/sec as a function of Gflops/byte on a log log scale
 - Polynomialia become straight lines
 - y intersect: multiplicative factor
 - slope: exponent \rightarrow linear: 45^0 slope

Typical Roofline Plot



Low Operational Intensity:

- very few Flops per byte
- memory bandwidth is limiting factor
- linear slope behavior

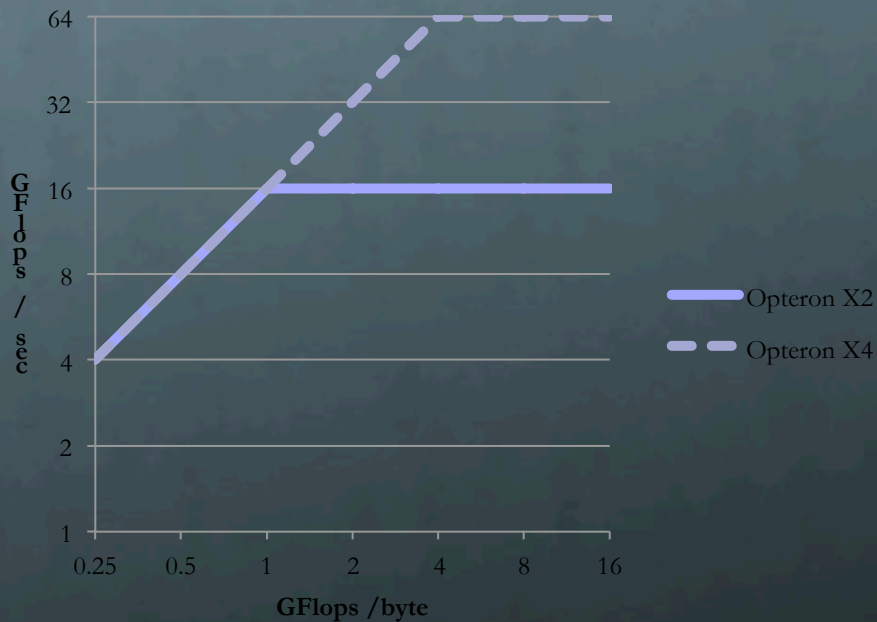
High Operational Intensity:

- many Flops per byte
- machine peak Flop rate is limiting factor
- constant performance

ridge point, where slope meets horizontal:

minimum operational intensity to get maximal performance

Example: Opteron X2 vs. Opteron X4



🌐 Both in same socket, so same memory behavior

🌐 X4: 4x higher Gflops rate

double # cores

double peak performance / core



🌐 4X higher roofline, but only advantageous when there is enough work per byte accessed. Low operational intensity programs do not benefit.

Adding ceilings to roofline



- 🌐 Roofline gives upper bound on performance, achieved only if the program you run can exploit all architectural phenomena.
- 🌐 Without some optimizations, only a lower ceiling can be reached

Reducing computational bottlenecks

Improve ILP

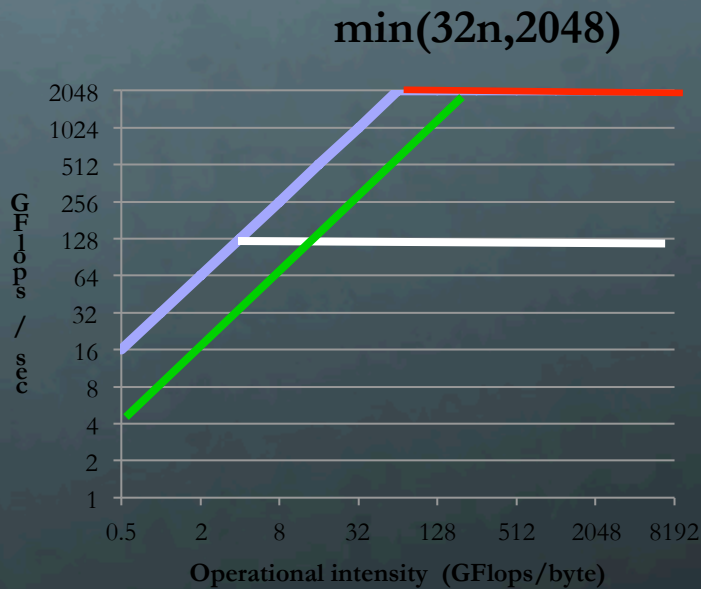
-  Better ILP covers the functional units of the machine better.
-  Can e.g. be achieved by loop unrolling, or applying SIMD (e.g. SSE instructions on Intel machines)

Balance Flop mix (add, multiply)

-  many machine have multiply-add units (inner product)
-  or equal number of add and multiply units

Reducing memory bottlenecks

- 🌐 Restructure loops for unit stride access (cache, hardware prefetching)
- 🌐 Ensure **memory affinity**
 - 🌐 some memory banks are closer to one core, some are closer to another core, so allocate threads and their data to a core / memory pair
- 🌐 Software prefetching can outperform hardware prefetching, e.g., in case of irregular memory access patterns



balance

With perfect flop balance you can reach this line

ILP

Without good ILP, you cannot get above this line

$\text{min}(32n, 2048)$

Similar lower slope ceilings for memory
e.g. unit stride

Roofline and cache

- 🌐 Operational intensity can vary with problem size (e.g. matrix multiply, FFT) because of data reuse and hence better cache behavior, providing a shift right on the roofline. *By doing flops better you go faster*
- 🌐 Also, we can exchange computation, and thus operational intensity, for memory access (table lookup) and shift left on the roofline.

By doing fewer flops you can go faster