Announcements / Reminders

- Pre-proposal due this Thursday (Jan 18) - use your GT login to access papers, if necessary
- HW1 due next Monday (Jan 22)
- PACE ICE down Jan 23-25
- Originally the plan was to release HW2 on Jan 25th due to PACE maintenance - I will release it earlier (on Jan 22nd) to allow extra time to read the handout. **The deadline will stay the same** (Feb 6)
- Please search for project partners via Ed discussion thread

Previously on CSE 6230

Memory hierarchy

Caches (and associativity) -> also, a short sidebar to answer a question from last lecture

Memory latency and bandwidth

Ideal-Cache model

Matrix multiplication

- Cache-aware algorithms
- **TO FINISH** -> Cache-oblivious algorithms

Set associativity and resource augmentation

Given an α-way set-associative cache of total size k, assuming sets are chosen with a fully-random hash function,

Recall: **tradeoff** in α between hit rate and search latency

- For $\alpha = \omega(\log k)$, the paging cost of an α-way set-associative LRU cache is within *additive* $O(1)$ of that a fully-associative LRU cache of size $(1-o(1))k$, with probability $1-1/\operatorname{poly}(k)$, for all request sequences of length $\operatorname{poly}(k)$.
- For $\alpha = o(\log k)$, and for all $c = O(1)$ and $r = O(1)$, the paging cost of an α-way set-associative LRU cache is *not* within a *factor* c of that a fully-associative LRU cache of size k/r, for some request sequence of length $O(k^{1.01})$.
- For $\alpha = \omega(\log k)$, if the hash function can be occasionally changed, the paging cost of an α-way set-associative LRU cache is within a *factor* $1+o(1)$ of that a fully-associative LRU cache of size $(1-o(1))k$, with probability $1-1/\operatorname{poly}(k)$, for request sequences of arbitrary (e.g., super-polynomial) length.

From "An Associativity Threshold Phenomenon in Set-Associative Caches," Bender et al., SPAA '23.

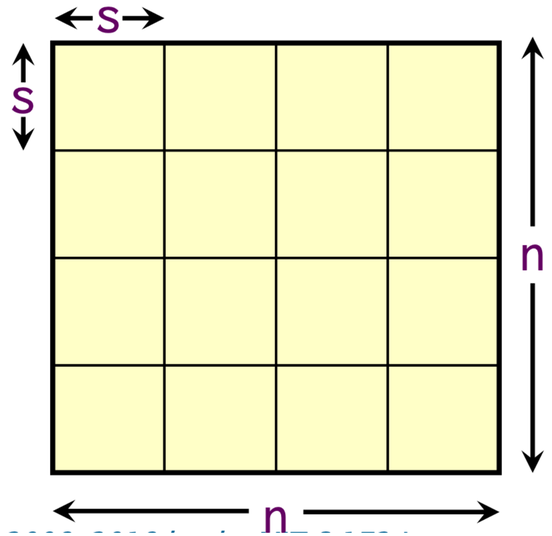
Recall: Tiled (Blocked) matrix multiply

```
void Tiled_Mult(double *C, double *A, double
  for (int64_t i1=0; i1<n/s; i1+=s)
    for (int64_t j1=0; j1<n; j1+=s)
      for (int64_t k1=0; k1<n; k1+=s)
        for (int64_t i=i1;                    i+
          for (int64_t j=j                    n; j++)
            for (int64_t k                    k<n; k++)
              C[i*n+j] +=                     n+j];
}
```

Challenging in **multilevel caches** and **multiprogrammed environments**

How?

Can we achieve cache optimality with none (or at least fewer) tuning parameters?



Analysis of cache

Tune s so that the che:

$s = \Theta(M^{1/2})$.

Submatrix Caching Lemma i plies $\Theta(s^2/B)$ misses per submatrix.

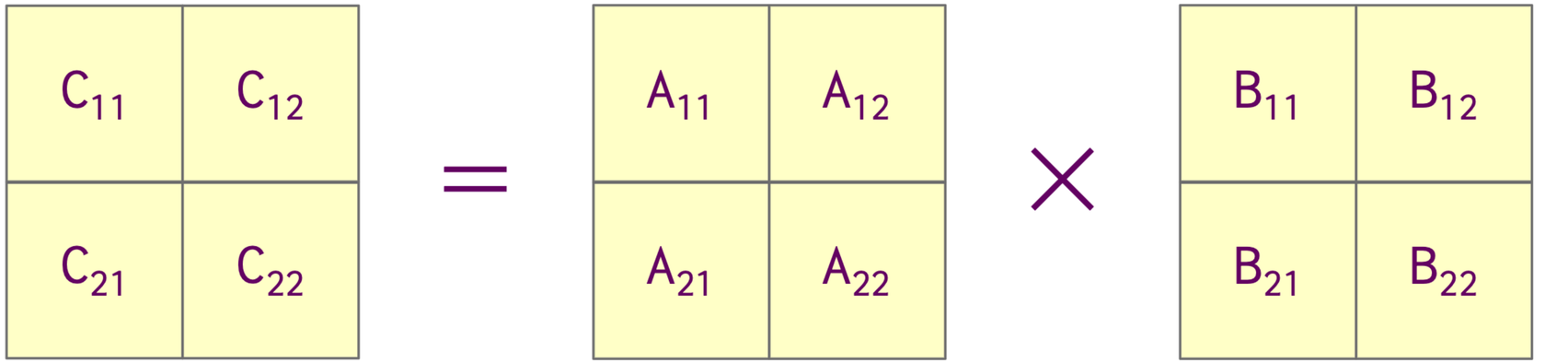
$Q(n) = \Theta((n/s)^3(s^2/B) = \Theta(n^3/(BM^{1/2}))$.

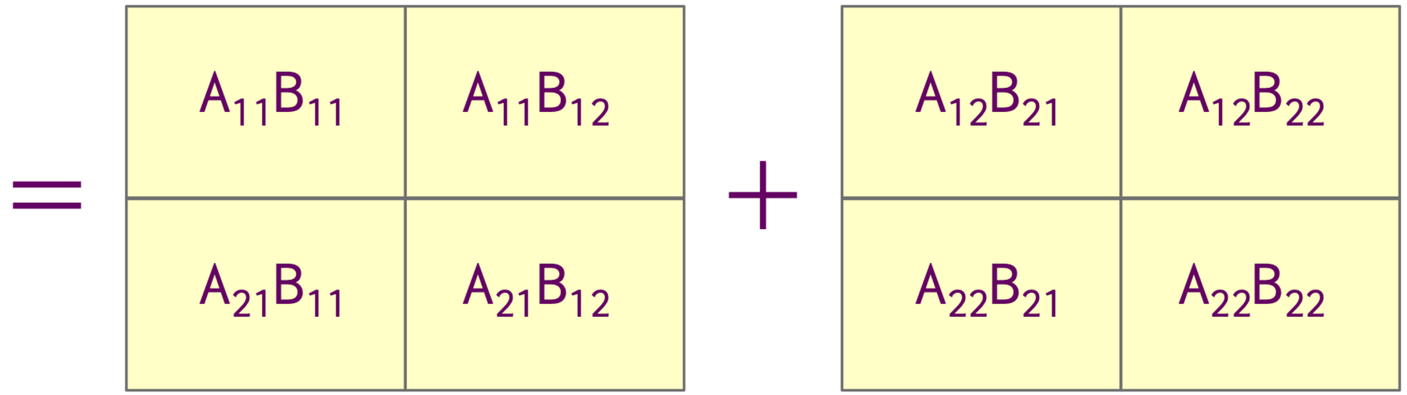
Optimal [HK81]

Divide and Conquer

Recursive matrix multiplication

Divide-and-conquer on nxn matrices:





8 multiply-adds of (n/2)x(n/2) matrices.

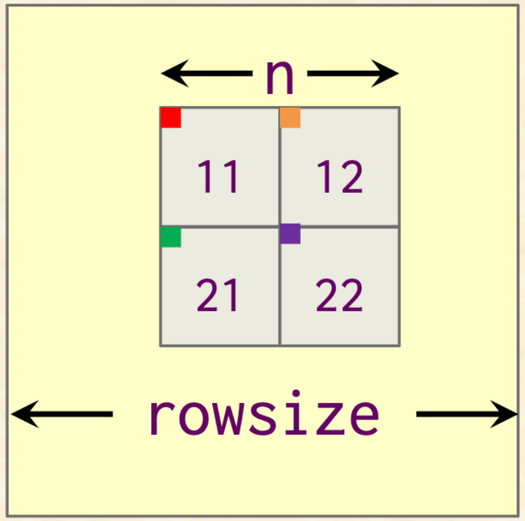
Recursive code

```
// Assume that n is an exact power of 2.
void Rec_Mult(double *C, double *A, double *B,
              int64_t n, int64_t rowsize) {
  if (n == 1)
    C[0] += A[0] * B[0];
  else {
    int64_t d11 = 0;
    int64_t d12 = n/2;
    int64_t d21 = (n/2) * rowsize;
    int64_t d22 = (n/2) * (rowsize+1);

    Rec_Mult(C+d11, A+d11, B+d11, n/2, rowsize);
    Rec_Mult(C+d11, A+d12, B+d21, n/2, rowsize);
    Rec_Mult(C+d12, A+d11, B+d12, n/2, rowsize);
    Rec_Mult(C+d12, A+d12, B+d22, n/2, rowsize);
    Rec_Mult(C+d21, A+d21, B+d11, n/2, rowsize);
    Rec_Mult(C+d21, A+d22, B+d21, n/2, rowsize);
    Rec_Mult(C+d22, A+d21, B+d12, n/2, rowsize);
    Rec_Mult(C+d22, A+d22, B+d22, n/2, rowsize);
} }
```

Coarsen base case to overcome function-call overheads.

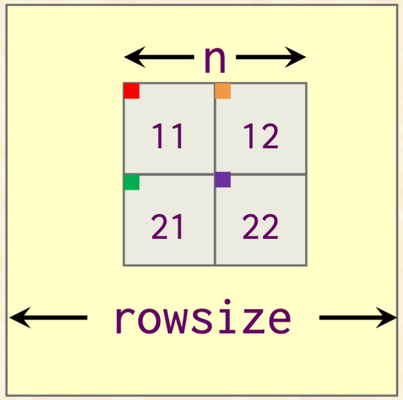
Recursive code

```
// Assume that n is an exact power of 2.
void Rec_Mult(double *C, double *A, double *B,
              int64_t n, int64_t rowsize) {
  if (n == 1)
    C[0] += A[0] * B[0];
  else {
    int64_t d11 = 0;
    int64_t d12 = n/2;
    int64_t d21 = (n/2) * rowsize;
    int64_t d22 = (n/2) * (rowsize+1);

    Rec_Mult(C+d11, A+d11, B+d11, n/2, rowsize);
    Rec_Mult(C+d11, A+d12, B+d21, n/2, rowsize);
    Rec_Mult(C+d12, A+d11, B+d12, n/2, rowsize);
    Rec_Mult(C+d12, A+d12, B+d22, n/2, rowsize);
    Rec_Mult(C+d21, A+d21, B+d11, n/2, rowsize);
    Rec_Mult(C+d21, A+d22, B+d21, n/2, rowsize);
    Rec_Mult(C+d22, A+d21, B+d12, n/2, rowsize);
    Rec_Mult(C+d22, A+d22, B+d22, n/2, rowsize);
} }
```



Analysis of work

```
// Assume that n is an exact power of 2.
void Rec_Mult(double *C, double *A, double *B,
              int64_t n, int64_t rowsize) {
  if (n == 1)
    C[0] += A[0] * B[0];
  else {
    int64_t d11 = 0;
    int64_t d12 = n/2;
    int64_t d21 = (n/2) * rowsize;
    int64_t d22 = (n/2) * (rowsize+1);

    Rec_Mult(C+d11, A+d11, B+d11, n/2, rowsize);
    Rec_Mult(C+d11, A+d12, B+d21, n/2, rowsize);
    Rec_Mult(C+d12, A+d11, B+d12, n/2, rowsize);
    Rec_Mult(C+d12, A+d12, B+d22, n/2, rowsize);
    Rec_Mult(C+d21, A+d21, B+d11, n/2, rowsize);
    Rec_Mult(C+d21, A+d22, B+d21, n/2, rowsize);
    Rec_Mult(C+d22, A+d21, B+d12, n/2, rowsize);
    Rec_Mult(C+d22, A+d22, B+d22, n/2, rowsize);
} }
```



$$W(n) = 8W(n/2) + \Theta(1)$$
$$= \Theta(n^3)$$

Analysis of work

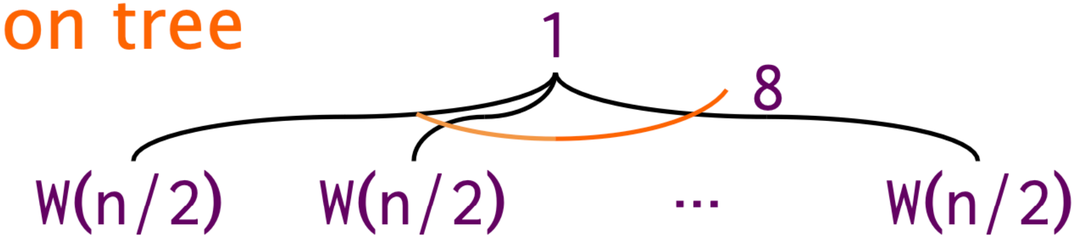
$$W(n) = 8W(n/2) + \Theta(1)$$

recursion tree $W(n)$

Analysis of work

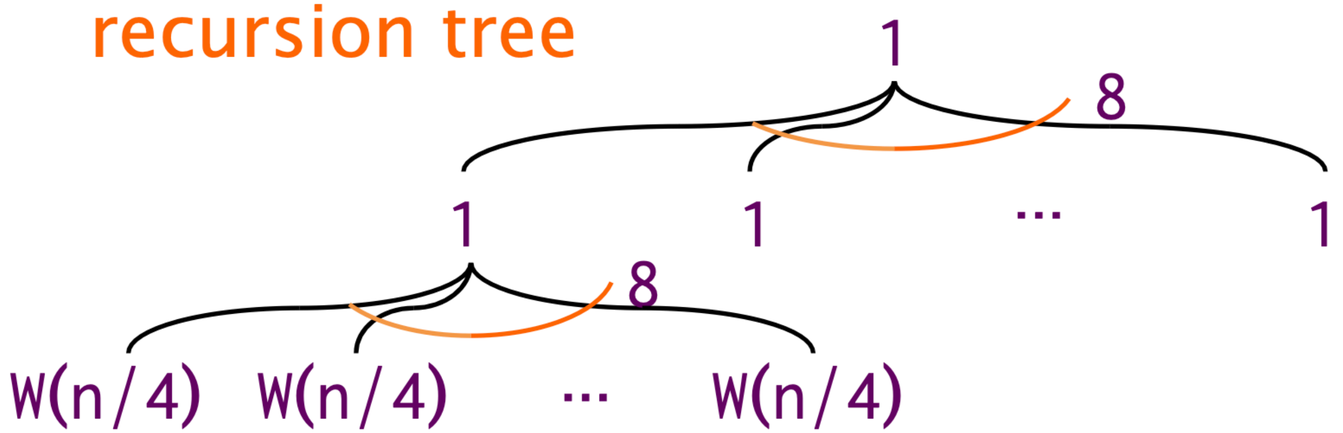
$$W(n) = 8W(n/2) + \Theta(1)$$

recursion tree



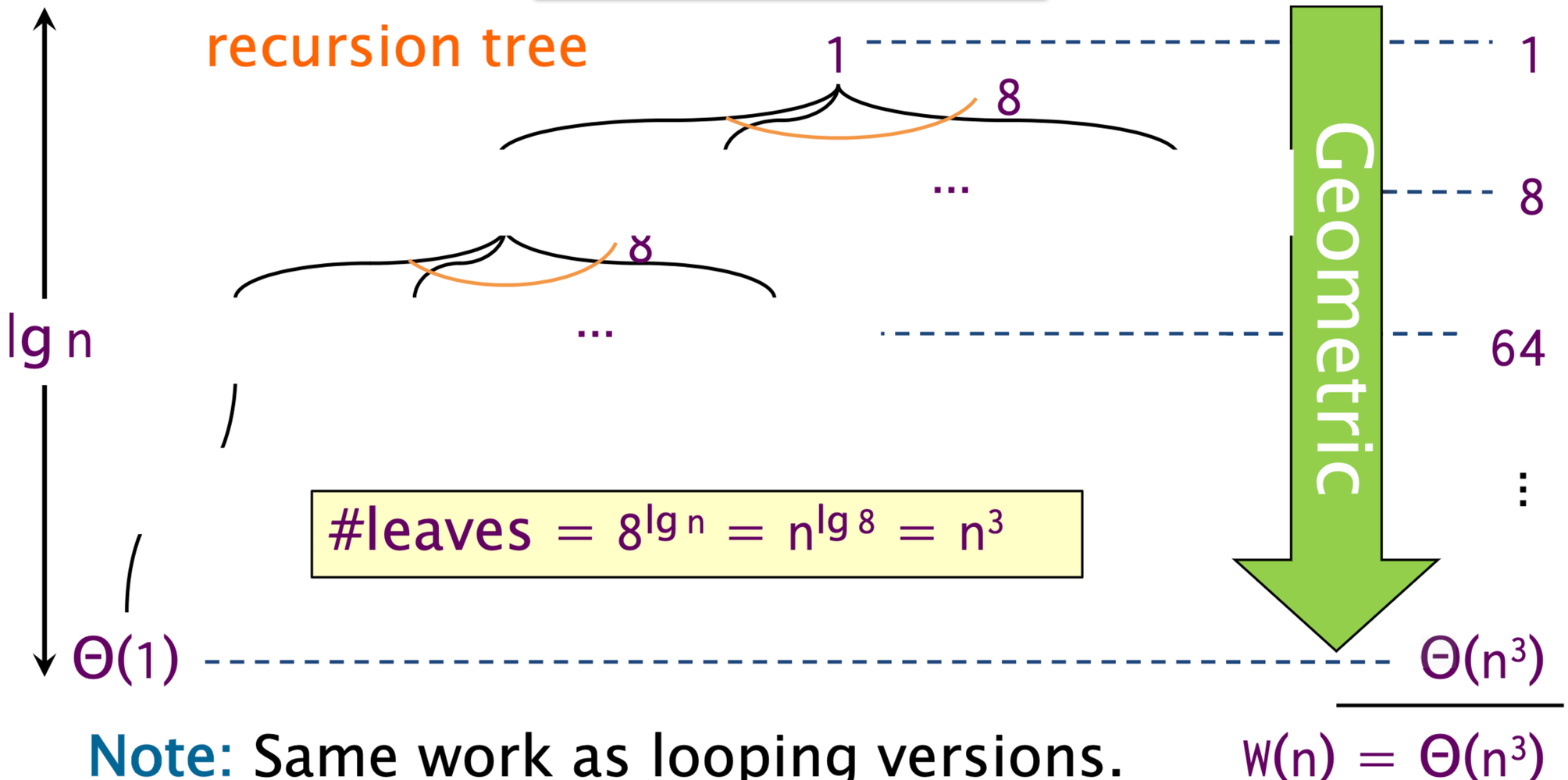
Analysis of work

$$W(n) = 8W(n/2) + \Theta(1)$$



Analysis of work

$$W(n) = 8W(n/2) + \Theta(1)$$



$W(n) = \Theta(n^3)$

Note: Same work as looping versions.

Analysis of cache misses

```
// Assume that n is an exact power of 2.
void Rec_Mult(double *C, double *A, double *B,
              int64_t n, int64_t rowsize) {
  if (n == 1)
    C[0] += A[0] * B[0];
  else {
    int64_t d11 = 0;
    int64_t d12 = n/2;
    int64_t d21 = (n/2) * rowsize;
    int64_t d22 = (n/2) * (rowsize+1);

    Rec_Mult(C+d11, A+d11, B+d11, n/2, rowsize);
    Rec_Mult(C+d11, A+d12, B+d21, n/2, rowsize);
    Rec_Mult(C+d12, A+d11, B+d12, n/2, rowsize);
    Rec_Mult(C+d12, A+d12, B+d22, n/2, rowsize);
    Rec_Mult(C+d21, A+d21, B+d11, n/2, rowsize);
    Rec_Mult(C+d21, A+d22, B+d21, n/2, rowsize);
    Rec_Mult(C+d22, A+d21, B+d12, n/2, rowsize);
    Rec_Mult(C+d22, A+d22, B+d22, n/2, rowsize);
} }
```

Submatrix Caching Lemma

$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) \text{ if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8Q(n/2) + \Theta(1) \text{ otherwise.} \end{cases}$$

Analysis of cache misses

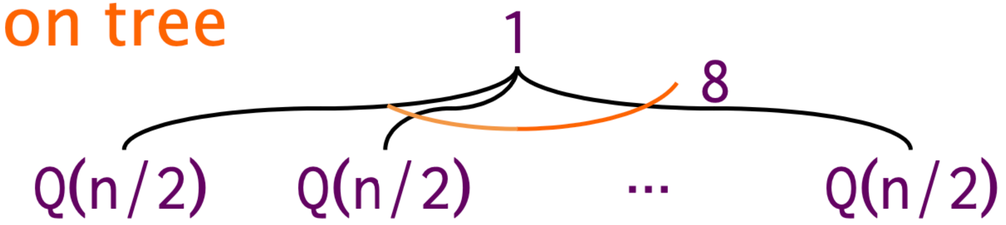
$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) \text{ if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8Q(n/2) + \Theta(1) \text{ otherwise.} \end{cases}$$

recursion tree $Q(n)$

Analysis of cache misses

$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) \text{ if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8Q(n/2) + \Theta(1) \text{ otherwise.} \end{cases}$$

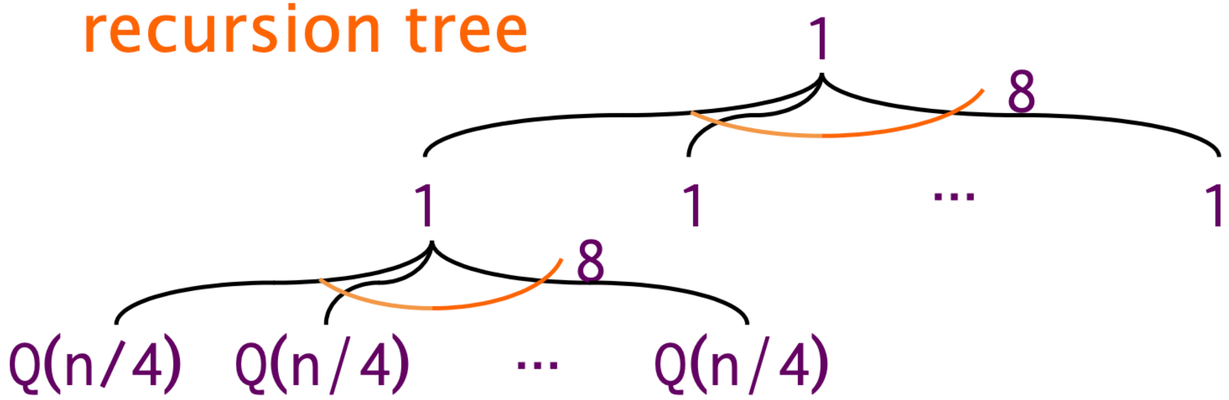
recursion tree



Analysis of cache misses

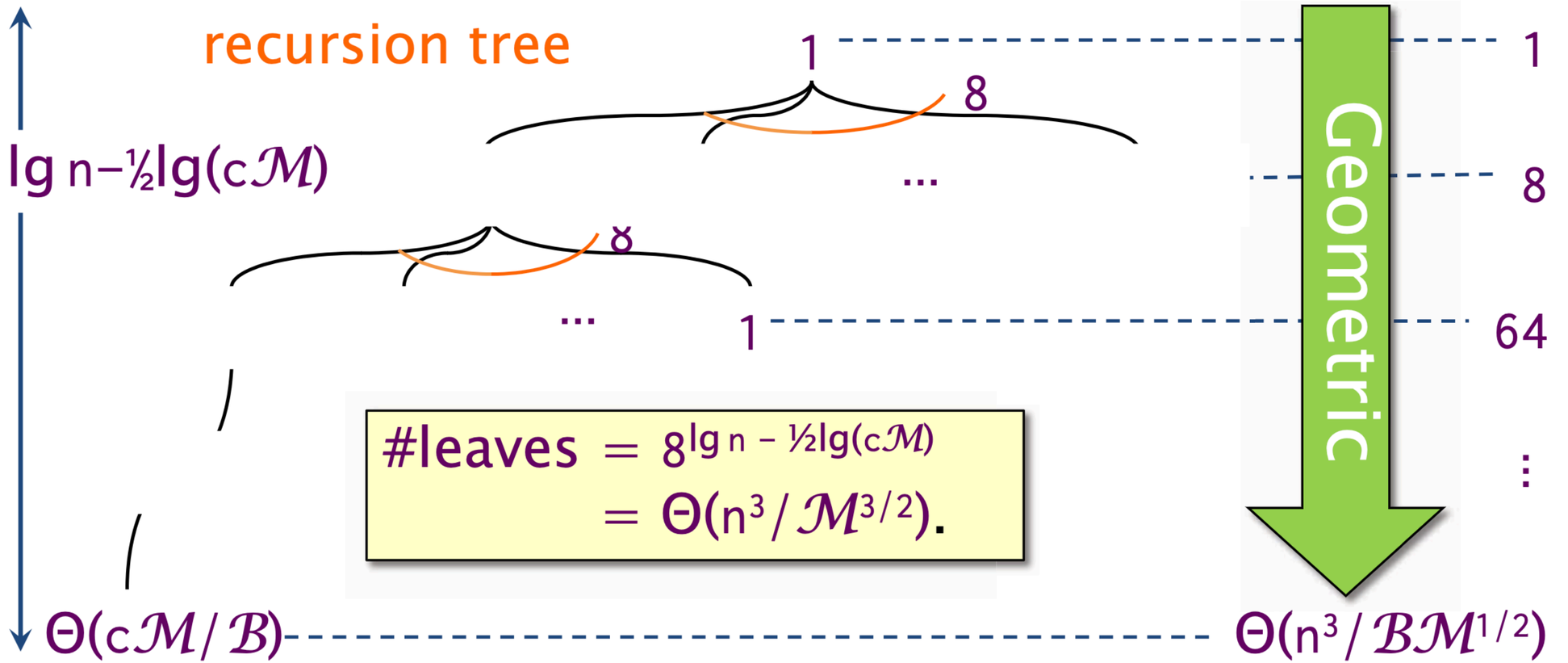
$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) \text{ if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8Q(n/2) + \Theta(1) \text{ otherwise.} \end{cases}$$

recursion tree



Analysis of cache misses

$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) \text{ if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8Q(n/2) + \Theta(1) \text{ otherwise.} \end{cases}$$



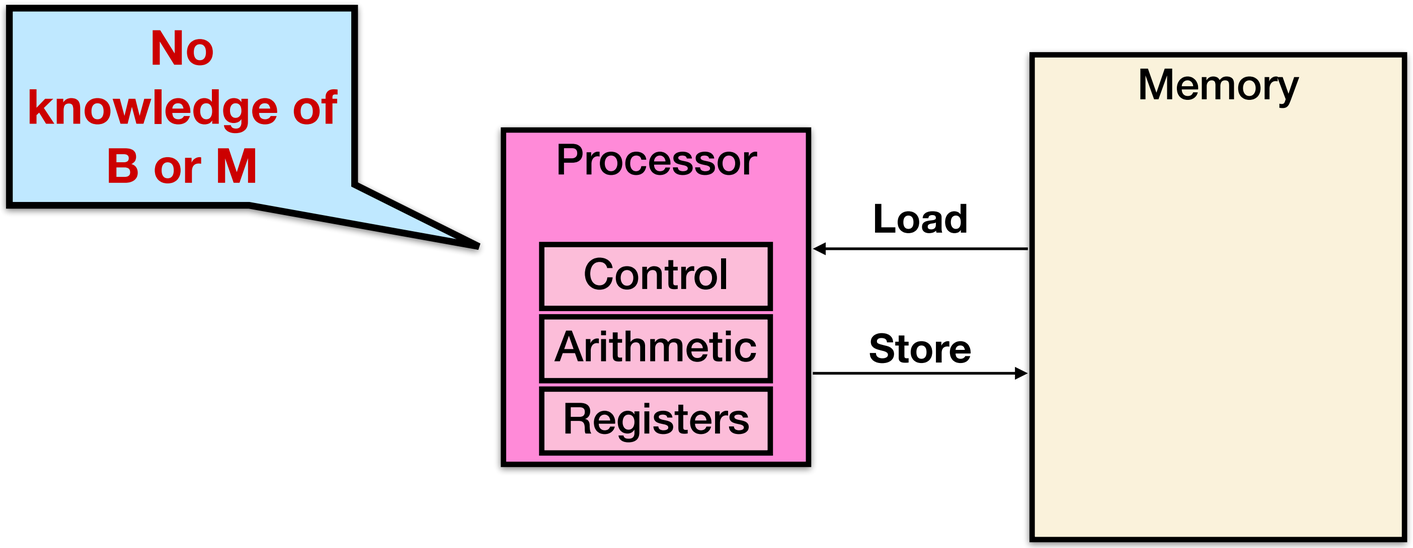
#leaves $= 8^{\lg n - \frac{1}{2}\lg(c\mathcal{M})}$
$= \Theta(n^3/\mathcal{M}^{3/2})$.

$$Q(n) = \Theta(n^3/\mathcal{B}\mathcal{M}^{1/2})$$

Same cache misses as with tiling!

Efficient cache-oblivious algorithms

- No tuning parameters.
- No explicit knowledge of cache sizes.
- Handle multilevel caches automatically (asymptotically optimally).
- Good in multiprogrammed environments (see: cache-adaptive algorithms).

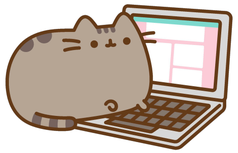


Next on CSE 6230

Today: How do the theoretical models and analysis we learned last class pan out in practice?

How can we further model specific machines and applications?

CSE 6230:
HPC Tools and Applications


+


Lecture 3: Practical Matrix Multiplication and the Roofline Model

Helen Xu
hxu615@gatech.edu

Georgia Tech College of Computing
School of Computational Science and Engineering

(Some slides from MIT's OCW 6.172, UC Berkeley CS267, Sam Williams' roofline talk)

Case study: Matrix multiplication
(From MIT 6.172 OCW)

(Square) Matrix Multiplication

```
void Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i=0; i < n; i++)
    for (int64_t j=0; j < n; j++)
      for (int64_t k=0; k < n; k++)
        C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```

$$\begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

C A B

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \, b_{kj}$$

From MIT 6.172

For simplicity, assume that n is a power of 2.

AWS c4.8xlarge Machine Specs

Feature	Specification
Microarchitecture	Haswell (Intel Xeon E5-2666 v3)
Clock frequency	2.9 GHz
Processor chips	2
Processing cores	9 per processor chip
Hyperthreading	2 way
Floating-point unit	8 double-precision operations, including fused-multiply-add, per core per cycle
Cache-line size	64 B
L1-icache	32 KB private 8-way set associative
L1-dcache	32 KB private 8-way set associative
L2-cache	256 KB private 8-way set associative
L3-cache (LLC)	25 MB shared 20-way set associative
DRAM	60 GB

Peak = $(2.9 \times 10^9) \times 2 \times 9 \times 2 \times 8 = 839$ GFLOPS

Version 1: Nested loops in Python

```
import sys, random
from time import *

n = 4096

A = [[random.random()
      for row in xrange(n)]
     for col in xrange(n)]
B = [[random.random()
      for row in xrange(n)]
     for col in xrange(n)]
C = [[0 for row in xrange(n)]
     for col in xrange(n)]

start = time()
for i in xrange(n):
    for j in xrange(n):
        for k in xrange(n):
            C[i][j] += A[i][k] * B[k][j]
end = time()

print '%0.6f' % (end - start)
```

Running time
= 21042 seconds
≈ 6 hours

Is this fast?

Should we expect more from our machine?

Version 1: Nested loops in Python

```
import sys, random
from time import *

n = 4096

A = [[random.random()
      for row in xrange(n)]
```

Running time
= 21042 seconds
≈ 6 hours

Is this fast?

Back-of-the-envelope calculation

$2n^3 = 2(2^{12})^3 = 2^{37}$ floating-point operations
Running time = 21042 seconds
∴ Python gets $2^{37}/21042 \approx 6.25$ MFLOPS
Peak ≈ 836 GFLOPS
Python gets ≈ 0.00075% of peak

```
print '%0.6f' % (end - start)
```

Version 2: Java

```
import java.util.Random;

public class mm_java {
  static int n = 4096;
  static double[][] A = new double[n][n];
  static double[][] B = new double[n][n];
  static double[][] C = new double[n][n];

  public static void main(String[] args) {
    Random r = new Random();

    for (int i=0; i<n; i++) {
      for (int j=0; j<n; j++) {
        A[i][j] = r.nextDouble();
        B[i][j] = r.nextDouble();
        C[i][j] = 0;
      }
    }

    long start = System.nanoTime();

    for (int i=0; i<n; i++) {
      for (int j=0; j<n; j++) {
        for (int k=0; k<n; k++) {
          C[i][j] += A[i][k] * B[k][j];
        }
      }
    }

    long stop = System.nanoTime();

    double tdiff = (stop - start) * 1e-9;
    System.out.println(tdiff);
  }
}
```

Running time = 2,738 seconds
≈ 46 minutes
… about 8.8× faster than Python.

```
for (int i=0; i<n; i++) {
  for (int j=0; j<n; j++) {
    for (int k=0; k<n; k++) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Version 3: C

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>

#define n 4096
double A[n][n];
double B[n][n];
double C[n][n];

float tdiff(struct timeval *start,
            struct timeval *end) {
  return (end->tv_sec-start->tv_sec) +
    1e-6*(end->tv_usec-start->tv_usec);
}

int main(int argc, const char *argv[]) {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      A[i][j] = (double)rand() / (double)RAND_MAX;
      B[i][j] = (double)rand() / (double)RAND_MAX;
      C[i][j] = 0;
    }
  }

  struct timeval start, end;
  gettimeofday(&start, NULL);

  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      for (int k = 0; k < n; ++k) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }

  gettimeofday(&end, NULL);
  printf("%0.6f\n", tdiff(&start, &end));
  return 0;
}
```

Using the Clang/LLVM 5.0 compiler

Running time = 1,156 seconds
≈ 19 minutes,
or about 2× faster than Java and about 18× faster than Python.

```c
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < n; ++j) {
    for (int k = 0; k < n; ++k) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Where we stand so far

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.007	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.119	0.014

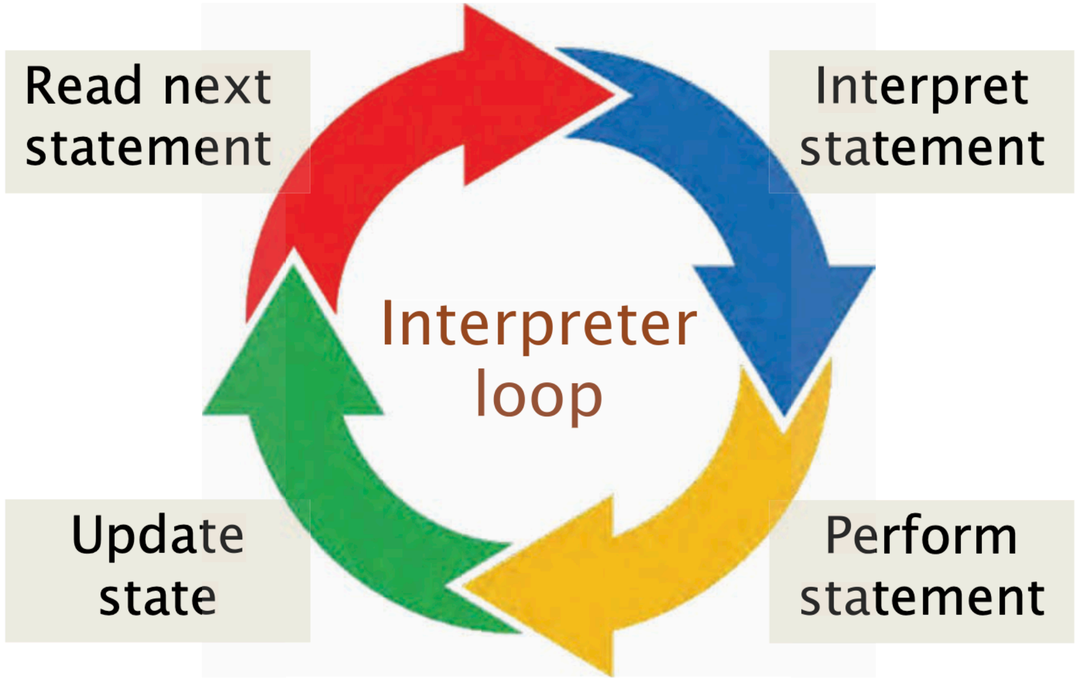
Why is Python so slow and C so fast?

- Python is interpreted.
- C is compiled directly to machine code.
- Java is compiled to byte-code, which is then interpreted and just-in-time (JIT) compiled to machine code.

From MIT 6.172

Interpreters are versatile but slow

- The interpreter reads, interprets, and performs each program statement and updates the machine state.
- Interpreters can easily support high-level programming features — such as dynamic code alteration — at the cost of performance.

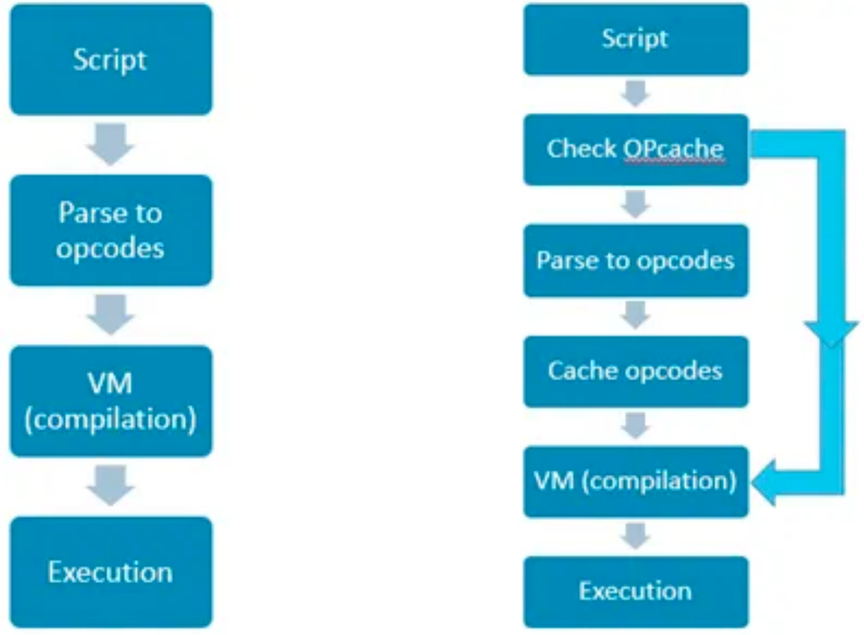


From MIT 6.172

JIT Compilation

JIT compilers can **recover some of the performance** lost by interpretation.

- When code is **first executed**, it is interpreted.
- The runtime system **keeps track** of how often the various pieces of code are executed.
- Whenever some piece of code executes sufficiently frequently, it gets compiled to **machine code** in real time.
- **Future executions** of the code use the more-efficient compiled version.



https://keeplearning.dev/jit-compiler-in-php-655d690f976c

From MIT 6.172

Recall: Loop order

We can change the **order of the loops** in this program without affecting correctness.

```
void Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i=0; i < n; i++)
    for (int64_t j=0; j < n; j++)
      for (int64_t k=0; k < n; k++)
        C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```

$Q(n) = \Theta(n^3)$
(Worst case when B doesn't fit into cache)

```
void Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i=0; i < n; i++)
    for (int64_t k=0; k < n; k++)
      for (int64_t j=0; j < n; j++)
        C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```

$Q(n) = n \cdot \Theta(n^2/B) = \Theta(n^3/B)$

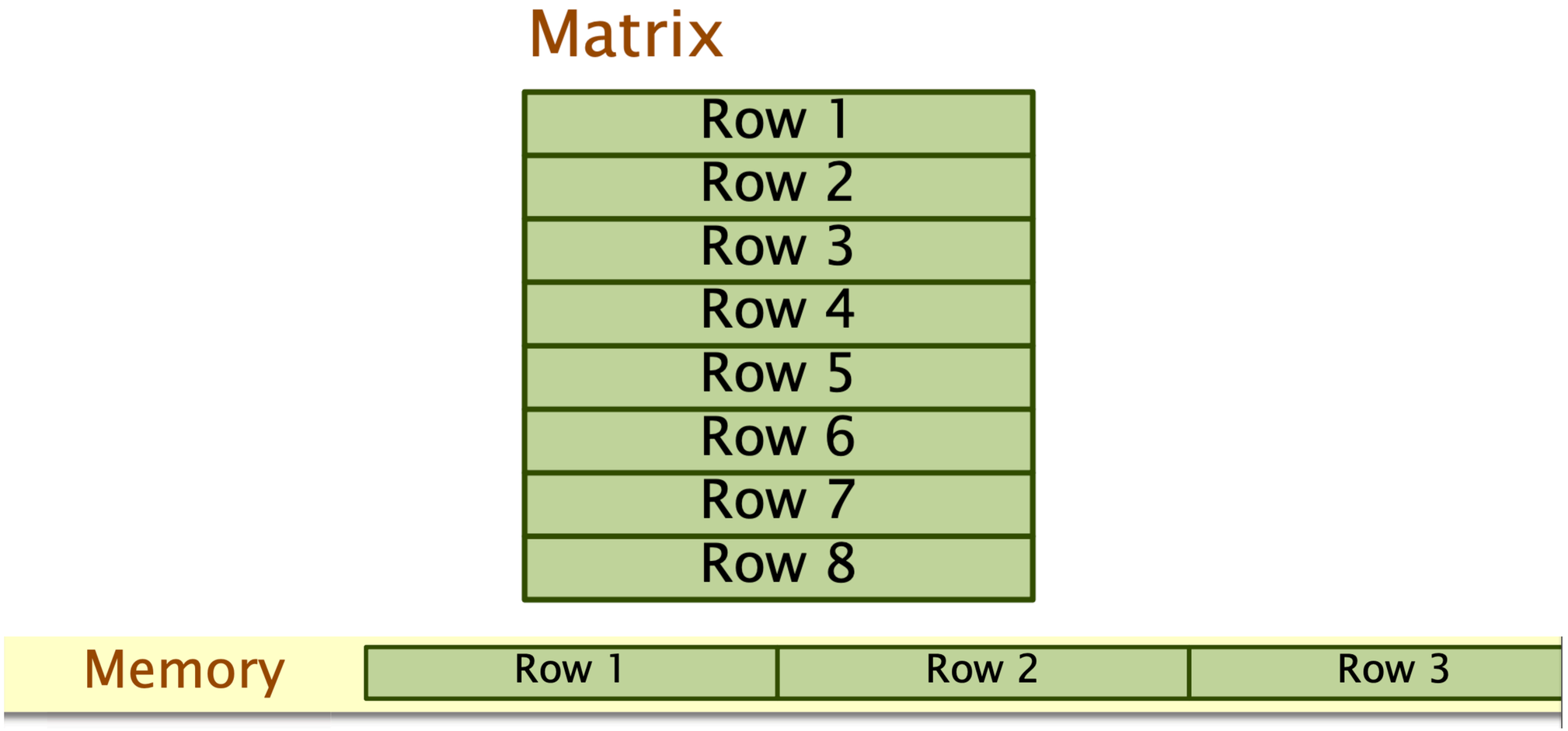
Performance of different loop orders

Loop order (outer to inner)	Running time (s)
i, j, k	1155.77
i, k, j	177.68
j, i, k	1080.61
j, k, i	3056.63
k, i, j	179.21
k, j, i	3032.82

Loop order affects running time by a factor of 18!

Recall: Row-major order

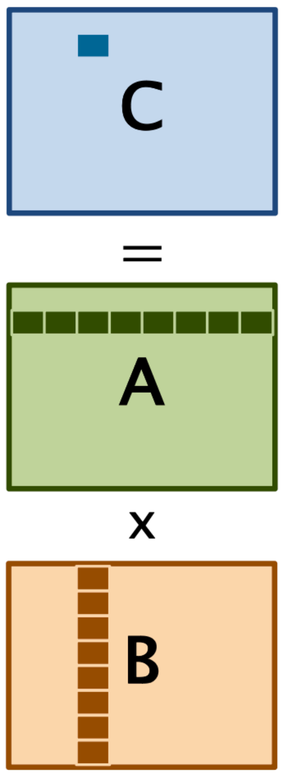
In this matrix-multiplication code, matrices are laid out in memory in **row-major order**.



Access pattern for order i, j, k

```
for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j)
    for (int k = 0; k < n; ++k)
      C[i][j] += A[i][k] * B[k][j];
```

Running time:
1155.77s



In-memory layout

Excellent spatial locality

Good spatial locality

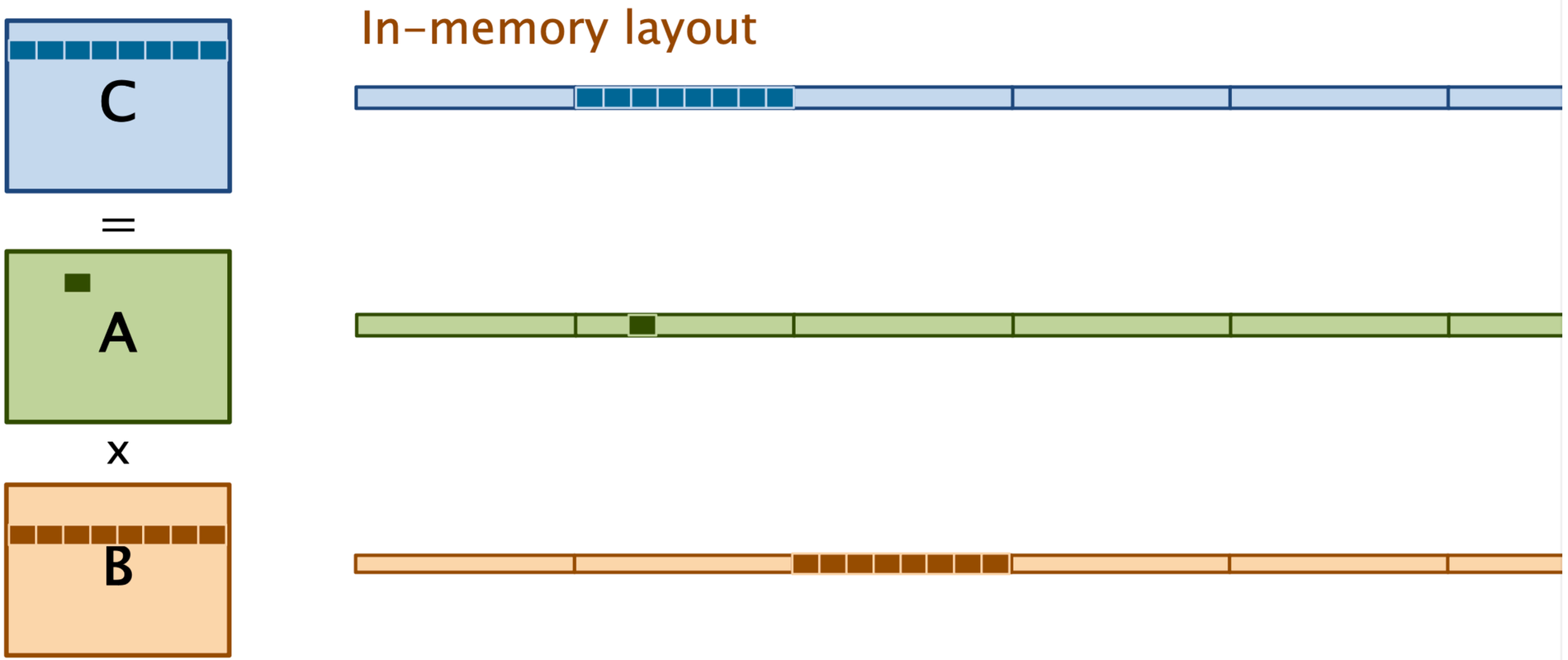
Poor spatial locality

4096 elements apart

Access pattern for order i, k, j

```
for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

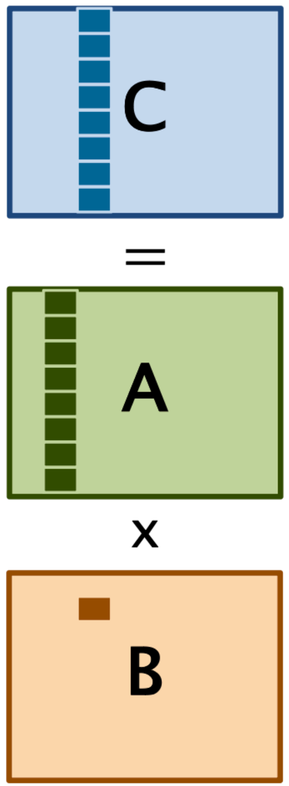
Running time:
177.68s



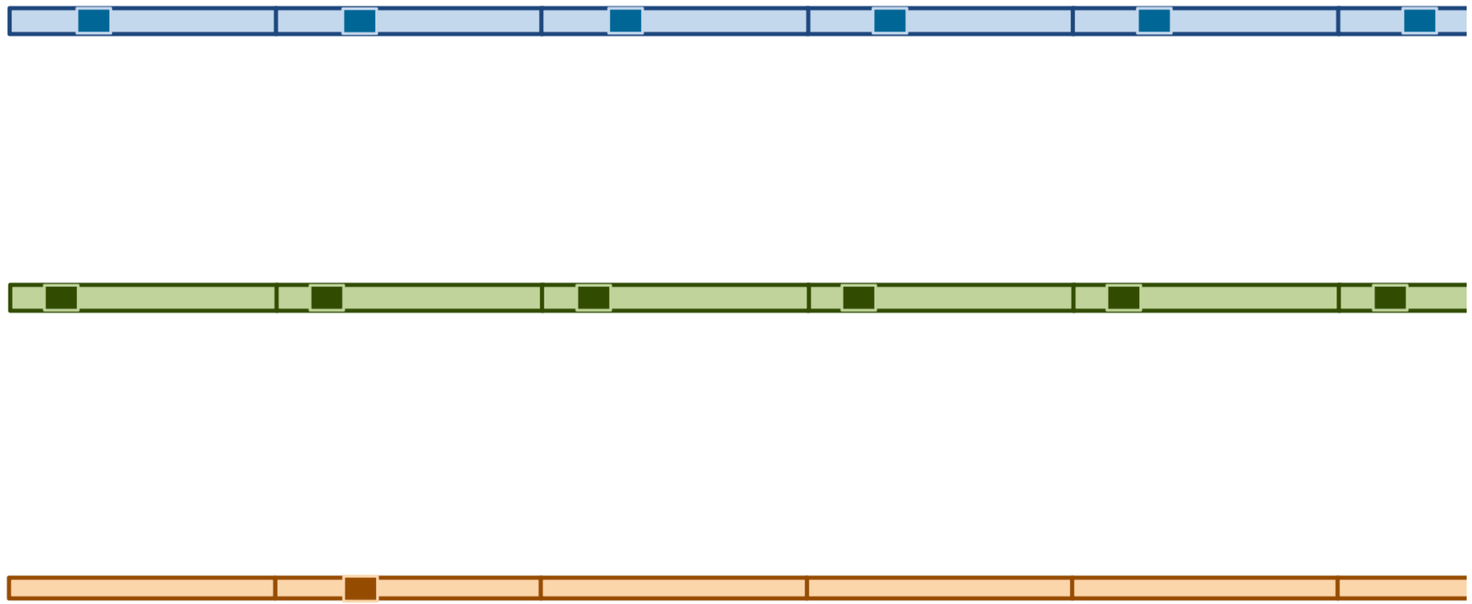
Access pattern for order j, k, i

```
for (int j = 0; j < n; ++j)
  for (int k = 0; k < n; ++k)
    for (int i = 0; i < n; ++i)
      C[i][j] += A[i][k] * B[k][j];
```

Running time:
3056.63s



In-memory layout



Cache performance with different orders

We can measure the effect of different access patterns using the Cachegrind cache simulator:

```
$ valgrind --tool=cachegrind ./mm
```

Loop order (outer to inner)	Running time (s)	Last-level-cache miss rate
i, j, k	1155.77	7.7%
i, k, j	177.68	1.0%
j, i, k	1080.61	8.6%
j, k, i	3056.63	15.4%
k, i, j	179.21	1.0%
k, j, i	3032.82	15.4%

Version 4: Interchange loops

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093

What other simple changes can we try?

Compiler optimization

Clang provides a collection of optimization switches. You can specify a switch to the compiler to ask it to optimize.

Opt. level	Meaning	Time (s)
`-O0`	Do not optimize	177.54
`-O1`	Optimize	66.24
`-O2`	Optimize even more	54.63
`-O3`	Optimize yet more	55.58

Clang also supports optimization levels for special purposes, such as `-Os`, which aims to limit code size, and `-Og`, for debugging purposes.

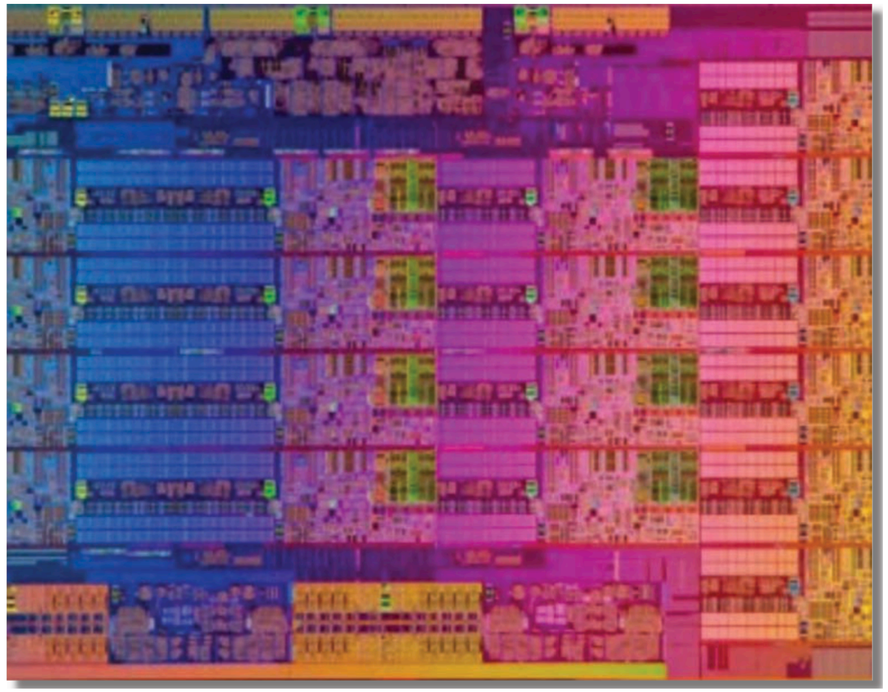
Version 5: Optimization switches

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301

With simple code and compiler technology, we can achieve 0.3% of the peak performance of the machine.

What's causing the low performance?

Multicore parallelism



Intel Haswell E5: 9 cores per chip

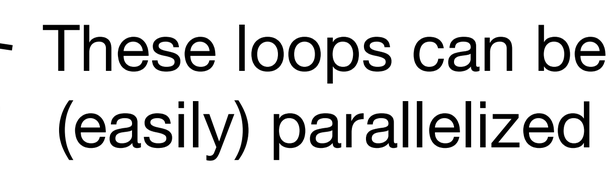
The test machine has 2 of these chips.

We’re running on just 1 of the 18 parallel processing cores on the system.
What happens if we use them all?

Parallel loops

Many parallel programming methods (e.g., OpenMP, Cilk, TBB, etc.) provide **parallel for loops** that allow all iterations of the loop to execute in parallel.

```
parallel_for (int64_t i=0; i < n; i++)
  for (int64_t k=0; k < n; k++)
    parallel_for (int64_t j=0; j < n; j++)
      C[i*n+j] += A[i*n+k] * B[k*n+j];
```



Which loops should we parallelize?

Experimenting with parallel loops

Parallel i loop

```
parallel_for (int64_t i=0; i < n; i++)
  for (int64_t k=0; k < n; k++)
    for (int64_t j=0; j < n; j++)
       C[i*n+j] += A[i*n+k] * B[k*n+j];
```

Runtime: 3.04s

Rule of thumb: Parallelize **outer loops** rather than inner loops.

Parallel j loop

```
for (int64_t i=0; i < n; i++)
   for (int64_t k=0; k < n; k++)
     parallel_for (int64_t j=0; j < n; j++)
        C[i*n+j] += A[i*n+k] * B[k*n+j];
```

Runtime: 531.71s

Parallel i and j loop

```
parallel_for (int64_t i=0; i < n; i++)
   for (int64_t k=0; k < n; k++)
     parallel_for (int64_t j=0; j < n; j++)
       C[i*n+j] += A[i*n+k] * B[k*n+j];
```

Runtime: 10.64s

(parallel loops here implemented in Cilk)

Version 6: Parallel loops

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408

Using parallel loops gets us almost 18× speedup on 18 cores! (Disclaimer: Not all code is so easy to parallelize effectively.)

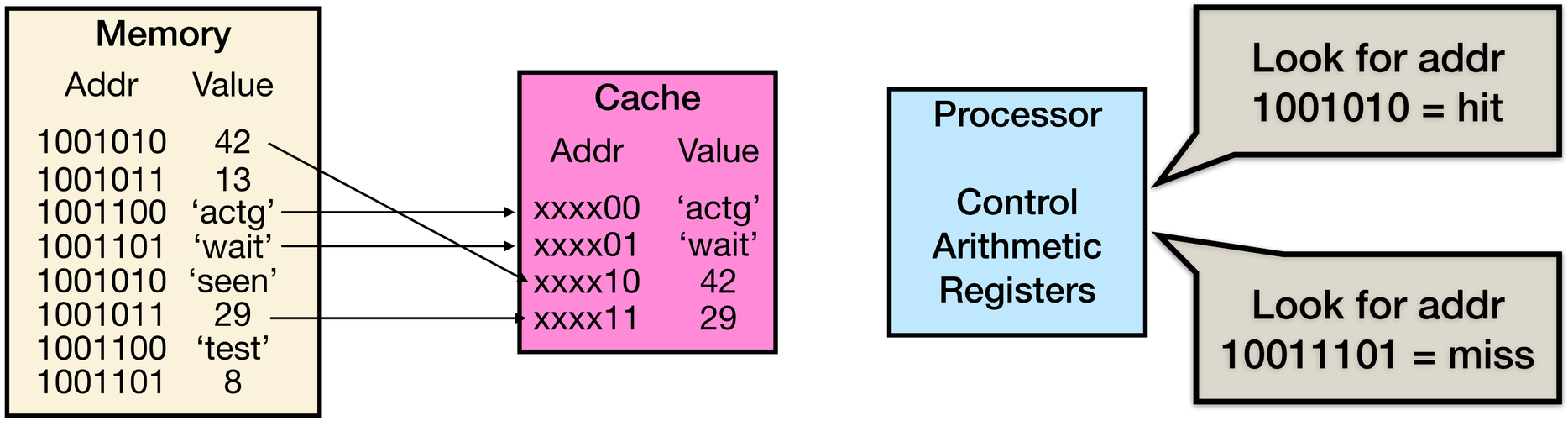
Why are we still getting just 5% of peak?

Recall: Cache basics

Cache is fast (expensive) memory which keeps a copy of the data; it is hidden from software.

Cache-line length: number of bytes loaded together in one entry (often 64 bytes).

Simple example: data at address xxxxx10 is stored at cache location 10.



Cache **hit**: access to a memory address in cache - cheap

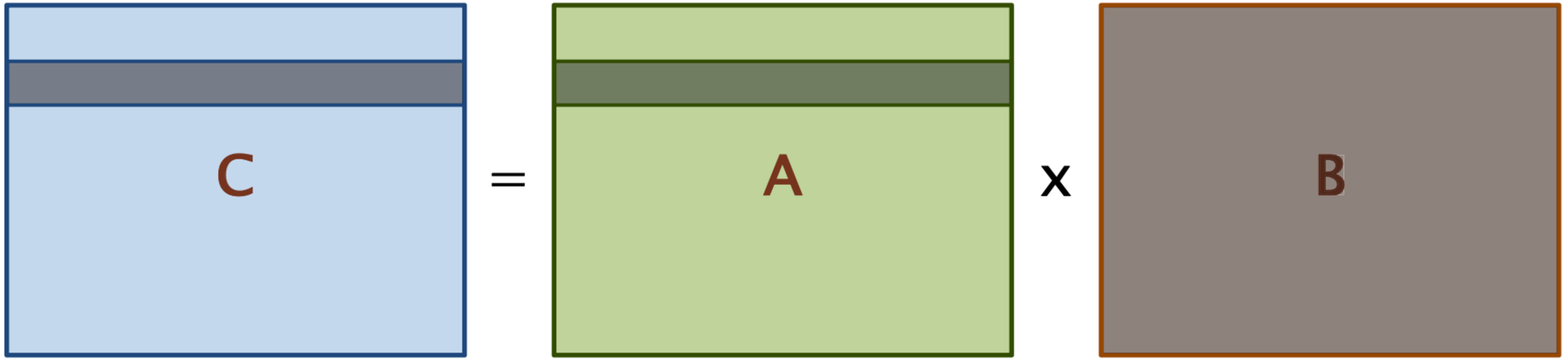
Cache **miss**: non-cached memory access - expensive

Need to look in next, slower level of memory.

Data reuse: Loops

How many memory accesses must the looping code perform to fully compute 1 row of C?

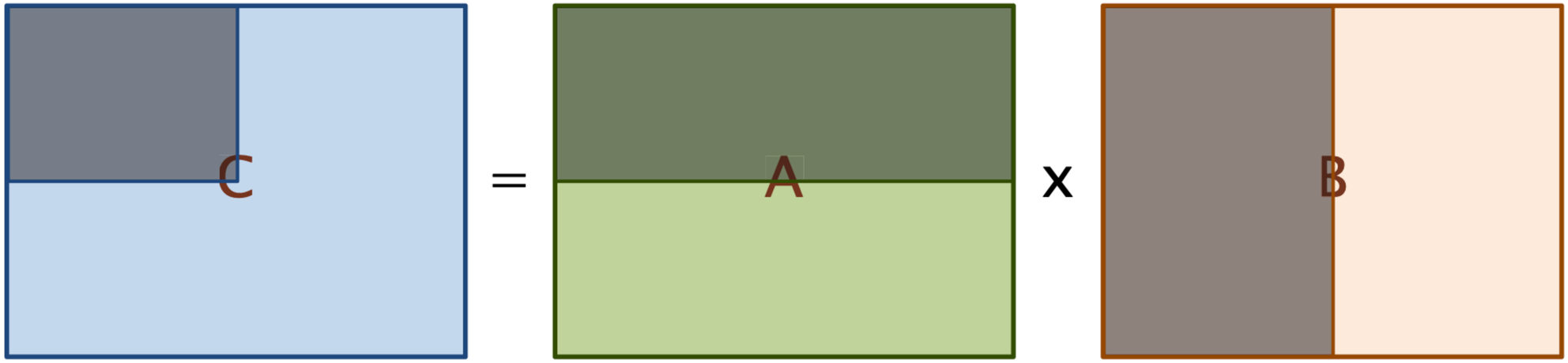
- 4096 * 1 = 4096 writes to C,
- 4096 * 1 = 4096 reads from A, and
- 4096 * 4096 = 16,777,216 reads from B, which is
- 16,785,408 memory accesses total.



Data reuse: Blocks

How about to compute a 64×64 block of C?

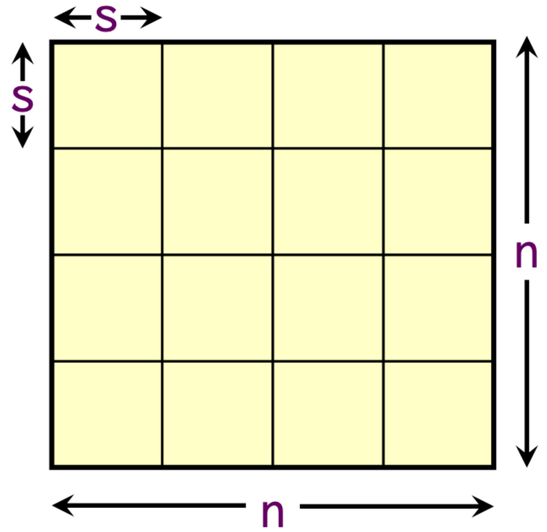
- $64 \cdot 64 = 4096$ writes to C,
- $64 \cdot 4096 = 262{,}144$ reads from A, and
- $4096 \cdot 64 = 262{,}144$ reads from B, or
- 528,384 memory accesses total.



Tiled matrix multiplication

```
void Tiled_Mult(double *C, double *A, double *B, int64_t n) {
  parallel_for (int64_t i1=0; i1<n; i1+=s)
    parallel_for (int64_t j1=0; j1<n; j1+=s)
      for (int64_t k1=0; k1<n; k1+=s)
        for (int64_t i=i1; i<i1+s && i<n; i++)
          for (int64_t k=k1; k<k1+s && k<n; k++)
            for (int64_t j=j1; j<j1+s && j<n; j++)
              C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```

Tuning parameter: How do we find the right value of s? Experiment!



Tile size	Running time (s)
4	6.74
8	2.76
16	2.49
32	1.74
64	2.33
128	2.13

Version 7: Tiling

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408
7	+ tiling	1.79	1.70	11,772	76.782	9.184

Implementation	Cache references (millions)	L1-d cache misses (millions)	Last-level cache misses (millions)
Parallel loops	104,090	17,220	8,600
+ tiling	64,690	11,777	416

The tiled implementation performs about 62% fewer cache references and incurs 68% fewer cache misses.

From MIT 6.172

Recall: Divide-and-conquer matrix multiplication

IDEA: Tile for every power of 2 simultaneously.

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

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

8 multiplications of $n/2 \times n/2$ matrices.
1 addition of $n \times n$ matrices.

Serial cache-oblivious matrix multiplication

```
// C += A * B
void mm_dac(double *C, int n_C, double *A, int n_A, double *B, int n_B, int n) {
  assert((n & (-n)) == n);
  if (n <= 1) {
    *C += *A * *B;
  } else {
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);

    mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
  }
}
```

Parallelizing divide-and-conquer matrix multiply

```
// C += A * B
void mm_dac(double *C,                                  n_A, double *B, int n_B, int n) {
  assert((n & (-n)) ==
  if (n <= 1) {
    *C += *A * *B;
  } else {
#define X(M,r,c) (M                 (n_ ## M) + c)*(n/2))
    parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    parallel_sync;
    parallel_spawn mm_dac                                n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac                                n_A, X(B,1,1), n_B, n/2);
    parallel_spawn mm_dac                                n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac                                n_A, X(B,1,1), n_B, n/2);
    parallel_sync;
  }
}
```

The child function call is **spawned**, meaning it may execute in parallel with the parent caller.

Sync means that control may not pass this point until **all spawned children have returned**.

Parallelizing divide-and-conquer matrix multiply

```
// C += A * B
void mm_dac(double *C, int n_C,                          e *B, int n_B, int n) {
  assert((n & (-n)) == n);
  if (n <= 1) {
    *C += *A * *B;
  } else {
#define X(M,r,c)                                  (n*(n_##M) + c)*(n/2))
    parallel_spawn                                0,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn                                0,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn                                1,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn                                1,0), n_A, X(B,0,1), n_B, n/2);
    parallel_sync;
    parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    parallel_sync;
  }
}
```

Initial runtime: 93.93s
…about **50x** slower than the last version!

The base case is too small. We must **coarsen** to overcome function-call overheads.

Coarsening the recursion

Just one tuning parameter, for the size of the **base case**

```
// C += A * B
void mm_dac(double *C, int                A, double *B, int n_B, int n) {
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    parallel_sync;
    parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    parallel_sync;
  }
}
```

Coarsening the recursion

Just one tuning parameter, for the size of the **base case**

```
// C += A * B
void mm_dac(double *C, int
                                                    A, double *B, int n_B, int n) {
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
```

```
void mm_base(double *C, int n_C, double *A, int n_A,
double *B, int n_B, int64_t n) {
  for (int64_t i=0; i < n; i++)
    for (int64_t k=0; k < n; k++)
      for (int64_t j=0; j < n; j++)
        C[i*n_C+j] += A[i*n_A+k] * B[k*n_B+j];
```

```
                                                    X(B,0,0), n_B, n/2);
                                                    X(B,0,1), n_B, n/2);
                                                    X(B,0,0), n_B, n/2);
                                                    X(B,0,1), n_B, n/2);

                                                    X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    parallel_sync;
  }
}
```

Coarsening the recursion

Just one tuning parameter, for the size of the **base case**

```
// C += A * B
void mm_dac(double *C, int n_C, double *A, int n_A, double *B, int n_B, int n) {
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
    parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    parallel_sync;
  }
}
```

```
void mm_base(double *C, int n_C, double *A, int n_A,
double *B, int n_B, int64_t n) {
  for (int64_t i=0; i < n; i++)
    for (int64_t k=0; k < n; k++)
      for (int64_t j=0; j < n; j++)
        C[i*n_C+j] += A[i*n_A+k] * B[k*n_B+j];
```

Base-case size	Running time (s)
4	3.00
8	1.34
16	1.34
32	1.30
64	1.95
128	2.08

Version 8: Divide and Conquer

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408
7	+ tiling	1.79	1.70	11,772	76.782	9.184
8	Parallel divide-and-conquer	1.30	1.38	16,197	105.722	12.646

Implementation	Cache references (millions)	L1-d cache misses (millions)	Last-level cache misses (millions)
Parallel loops	104,090	17,220	8,600
+ tiling	64,690	11,777	416
Parallel divide-and-conquer	58,230	9,407	64

Caveat: Some work has found that carefully-tuned cache-aware programs are faster than carefully-tuned cache-oblivious programs.

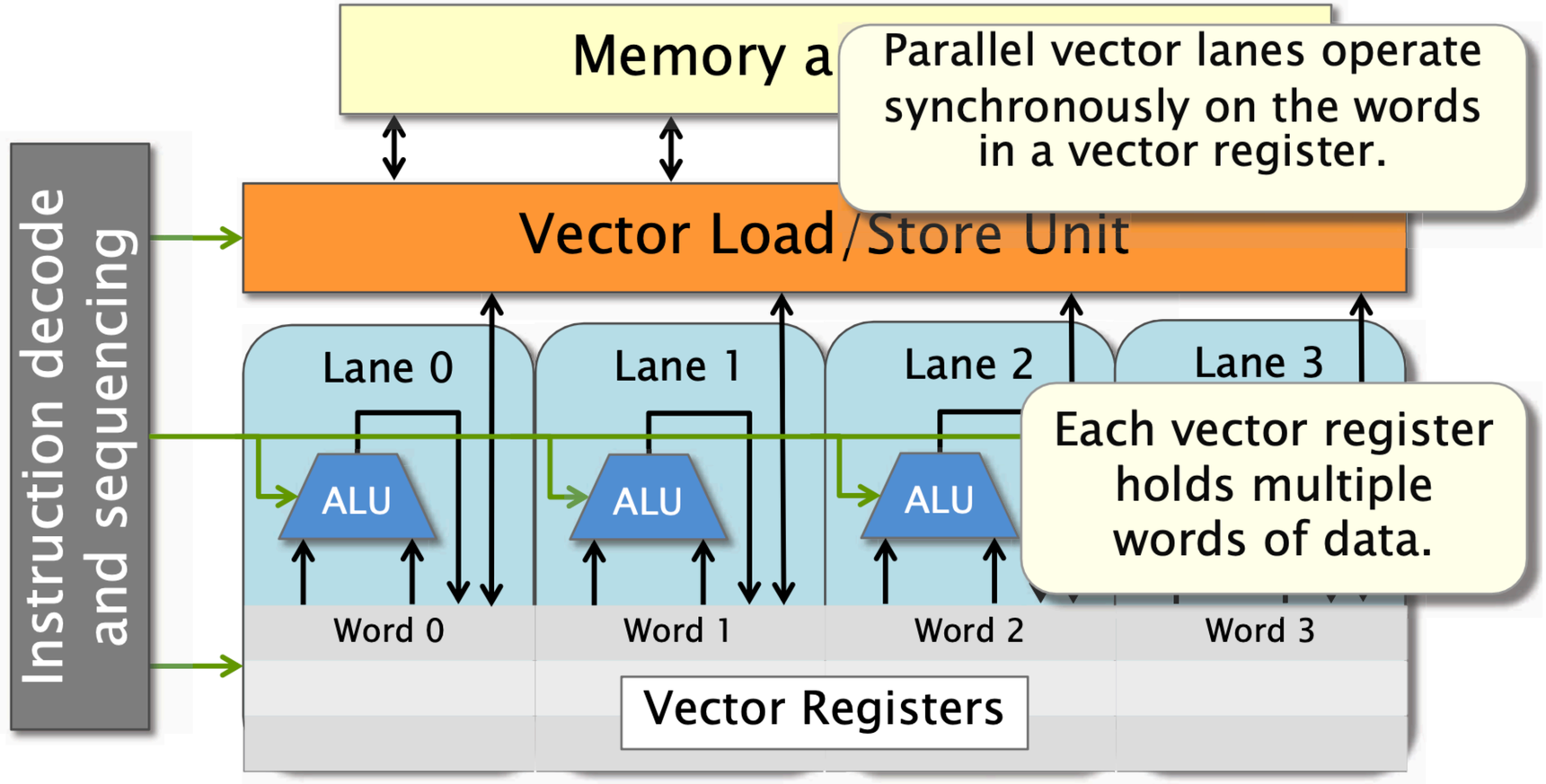
See: “An experimental comparison of cache-oblivious and cache-conscious programs" by Yotov et al. ‘07

From MIT 6.172

Vector Hardware

Modern microprocessors incorporate vector hardware to process data in single-instruction stream, multiple-data stream (SIMD) fashion.

We will have more in-depth lectures later on SIMD in theory and in practice



Vectorization flags

Programmers can direct the compiler to use modern vector instructions using compiler flags such as the following:

- `-mavx`: Use Intel AVX vector instructions.
- `-mavx2`: Use Intel AVX2 vector instructions.
- `-mfma`: Use fused multiply-add vector instructions.
- `-march=<string>`: Use whatever instructions are available on the specified architecture.
- `-march=native`: Use whatever instructions are available on the architecture of the machine doing compilation.

Due to restrictions on floating-point arithmetic, additional flags, such as `-ffast-math`, might be needed for these vectorization flags to have an effect.

Version 9: Compiler vectorization

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408
7	+ tiling	1.79	1.70	11,772	76.782	9.184
8	Parallel divide-and-conquer	1.30	1.38	16,197	105.722	12.646
9	+ compiler vectorization	0.70	1.87	30,272	196.341	23.486

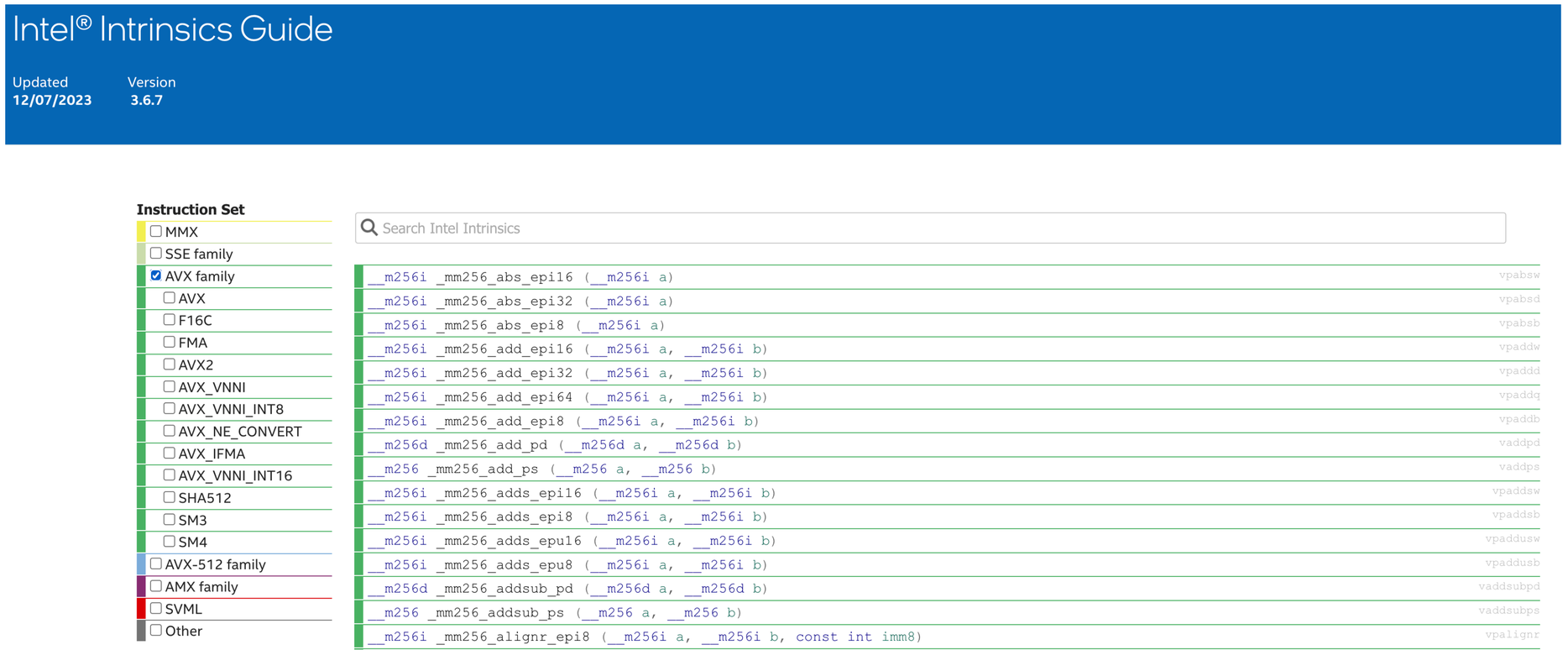
Using the flags `-march=native -ffast-math` nearly doubles the program's performance!

Can we be smarter than the compiler?

Intel vector intrinsics

Intel provides C-style functions, called **intrinsic instructions**, that provide direct access to hardware vector operations:

`https://software.intel.com/sites/landingpage/IntrinsicsGuide`



Version 10: AVX Intrinsics

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408
7	+ tiling	1.79	1.70	11,772	76.782	9.184
8	Parallel divide-and-conquer	1.30	1.38	16,197	105.722	12.646
9	+ compiler vectorization	0.70	1.87	30,272	196.341	23.486
10	+ AVX intrinsics	0.39	1.76	53,292	352.408	41.677

From MIT 6.172

Comparison with Intel MKL

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408
7	+ tiling	1.79	1.70	11,772	76.782	9.184
8	Parallel divide-and-conquer	1.30	1.38	16,197	105.722	12.646
9	+ compiler vectorization	0.70	1.87	30,272	196.341	23.486
10	+ AVX intrinsics	0.39	1.76	53,292	352.408	41.677
11	Intel MKL	0.41	0.97	51,497	335.217	40.098

Version 10 is competitive with Intel's professionally engineered Math Kernel Library!

From MIT 6.172

Comparison with Intel MKL

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
9	+ compiler vectorization	0.70	1.87	30,272	196.341	23.486
10	+ AVX intrinsics	0.39	1.76	53,292	352.408	41.677
11	Intel MKL	0.41	0.97	51,497	335.217	40.098

You generally won't see such extreme performance improvements (~50,000x) as we did for matrix multiplication.

But in CSE 6230, we will learn general techniques for performance improvement of a wide variety of applications.

Version 10 is competitive with Intel's professionally engineered Math Kernel Library!

From MIT 6.172

The roofline model for performance
(Slides inspired by UC Berkeley CS267)

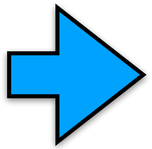
What is a performance model?

A performance model is a mathematical description based on a simplified machine model (ignoring many details) that aims to come up with a **quantitative estimate for expected performance**.

Bandwidth

Energy use

Clock speed



Performance (Often runtime)

Memory footprint

Total work (in flops)

Fraction of peak

…and other factors

Why use a performance model?

Performance models help us to **understand and predict performance behavior** without confounding factors from architectures, programming models, implementations, etc.

?

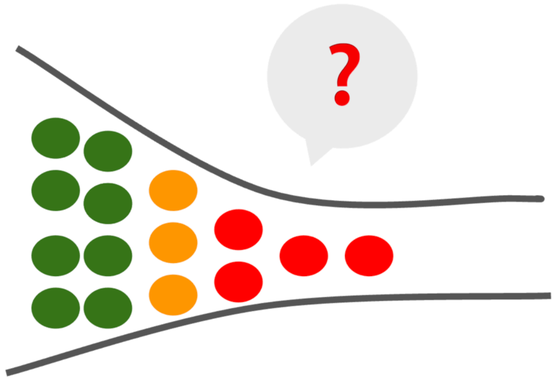
?

OpenMP

?

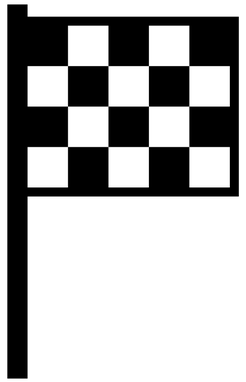
OpenCilk

Why use a performance model?



Help identify performance bottlenecks

Find where the bottleneck is (software? hardware? algorithm?)



Determine when we're done optimizing

Roofline model

Main idea: applications are limited by either compute peak or memory bandwidth.

Bandwidth bound - e.g., matrix-vector multiply

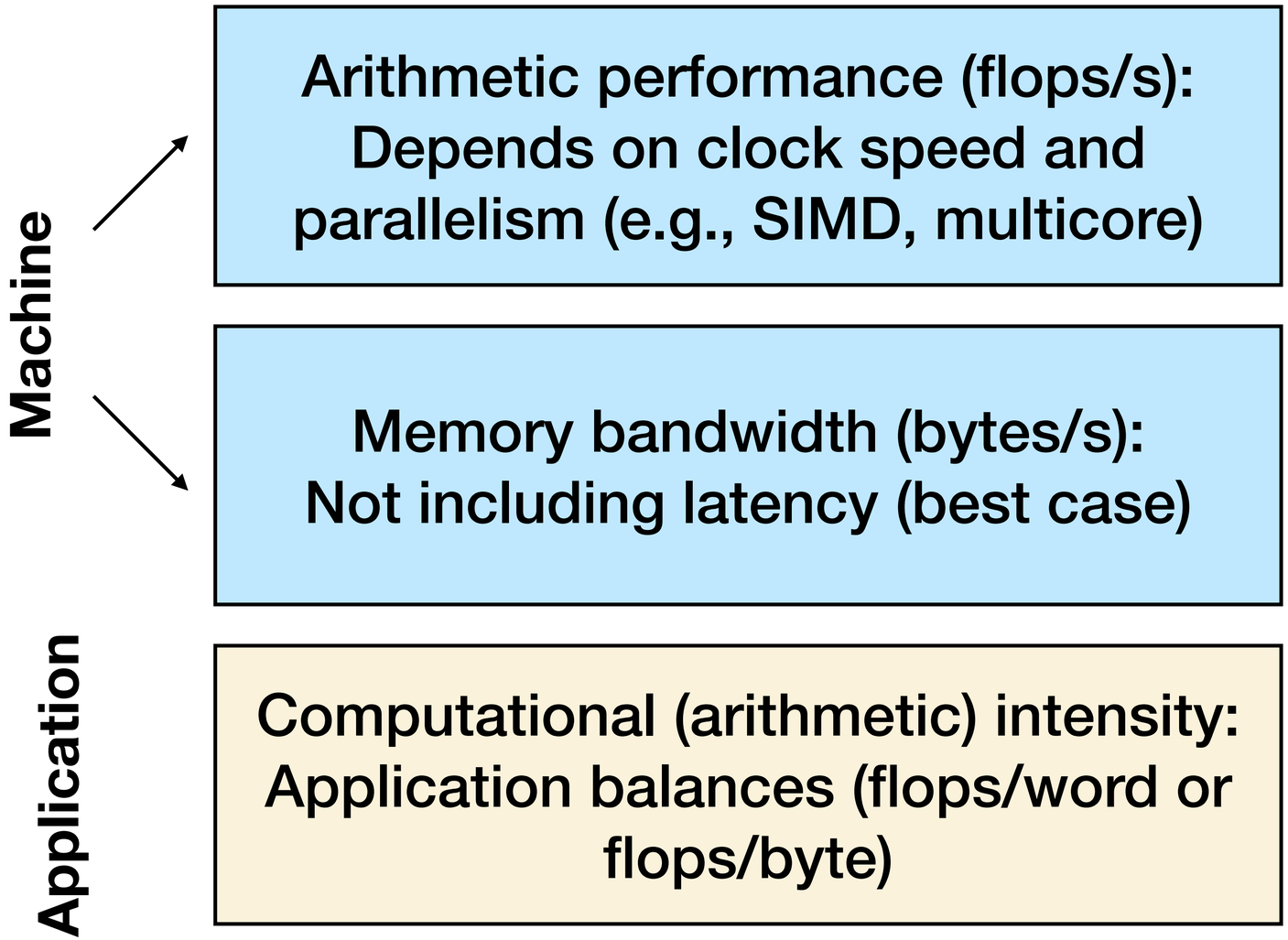
Compute bound - e.g., matrix-matrix multiply

Sam Williams

2847 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.

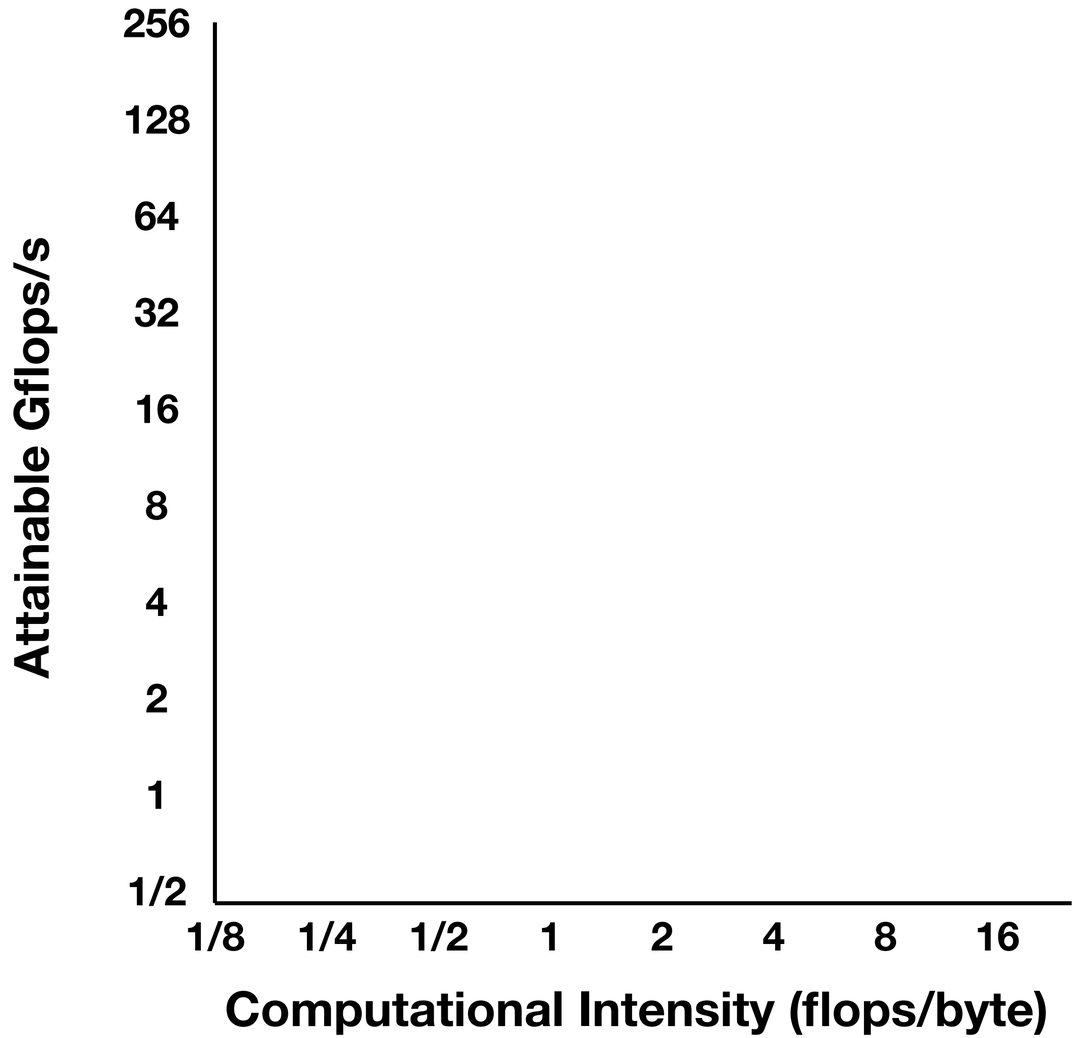
Components of roofline model



Maximum
achievable
performance
for your application

Roofline performance model

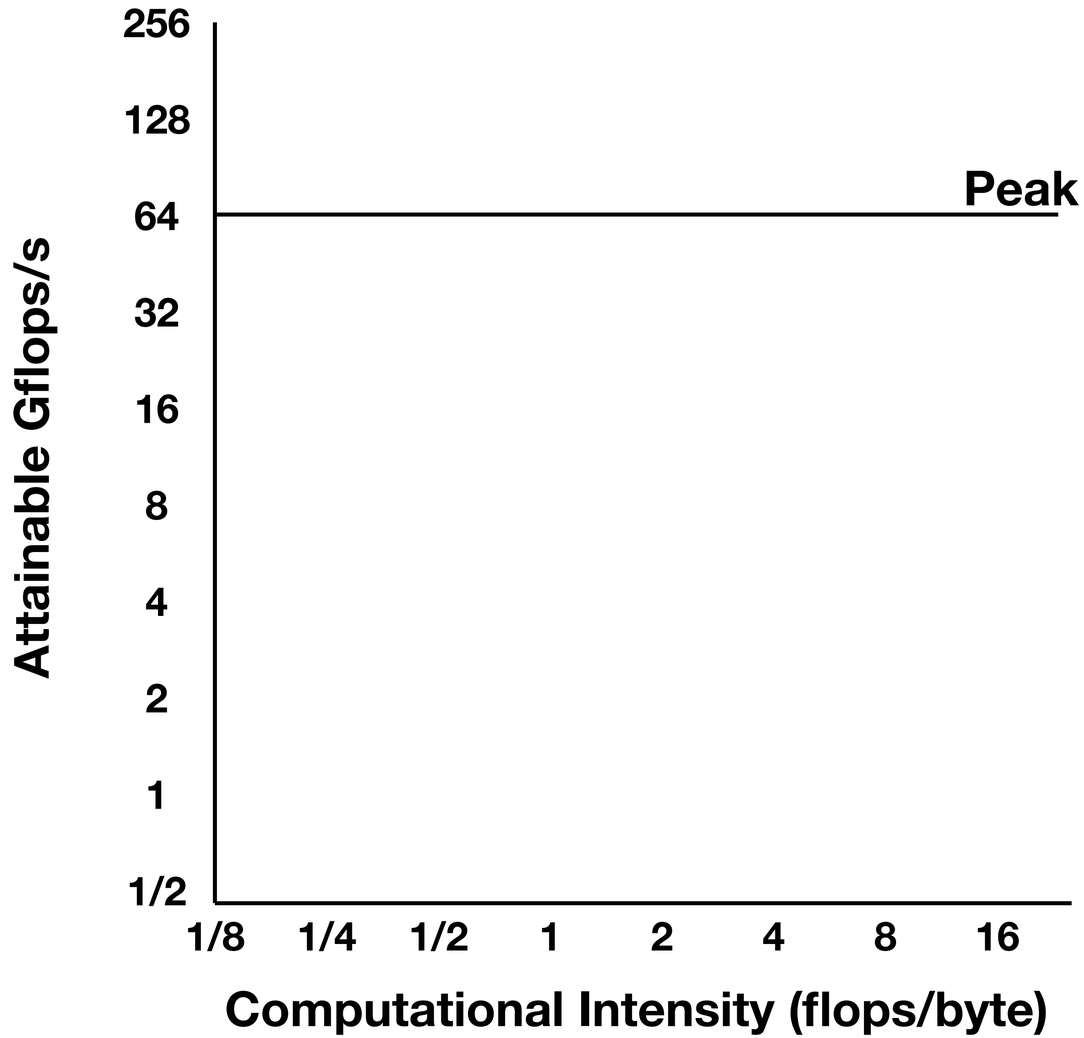
Example machine:



From UC Berkeley CS267

Roofline performance model

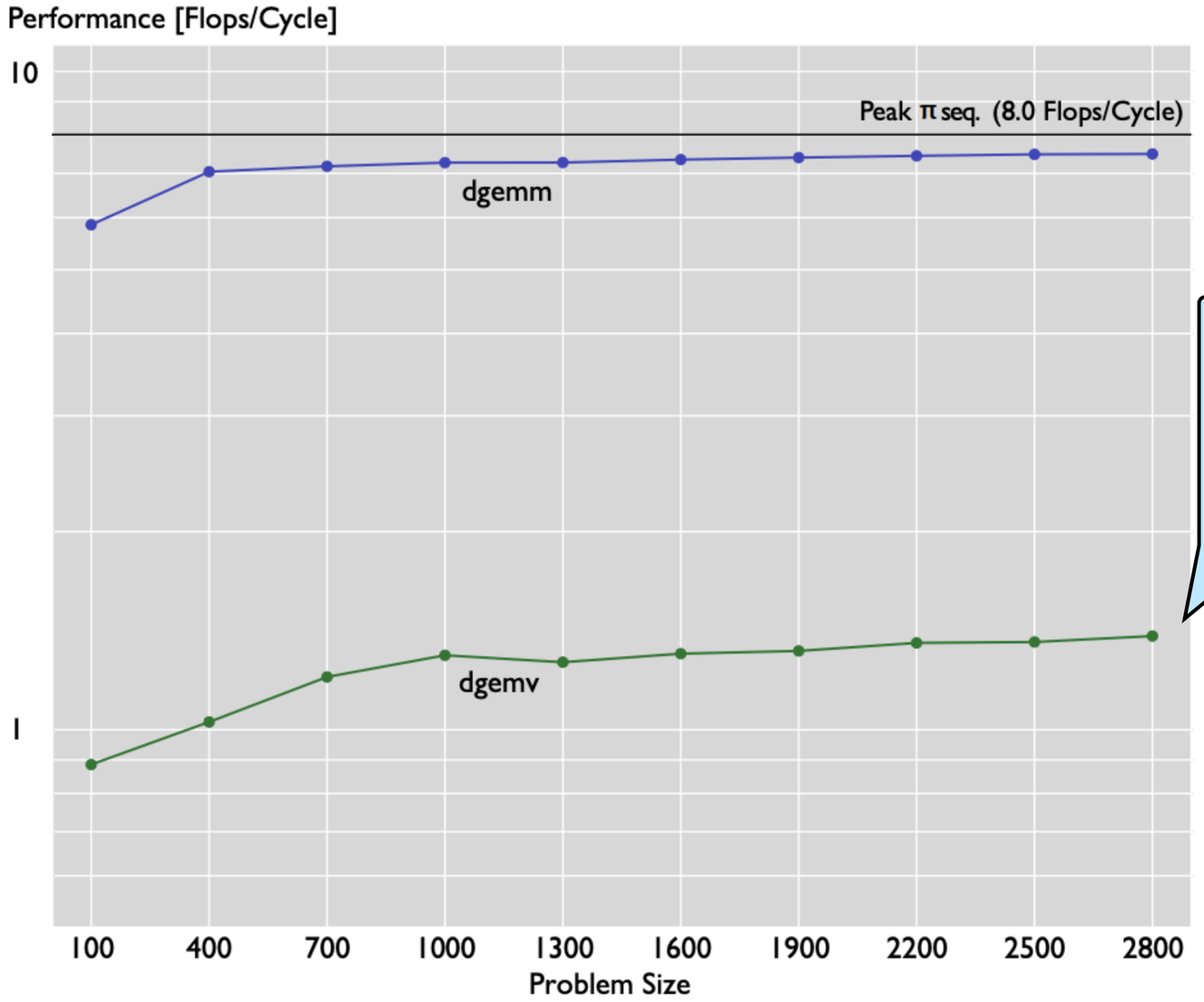
Example machine:



Top of the **roof is the peak compute rate**.

From UC Berkeley CS267

How good is flops/s as a model?



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

From UC Berkeley CS267 Image and related paper "Applying the Roofline Model" by G. Ofenbeck et al.

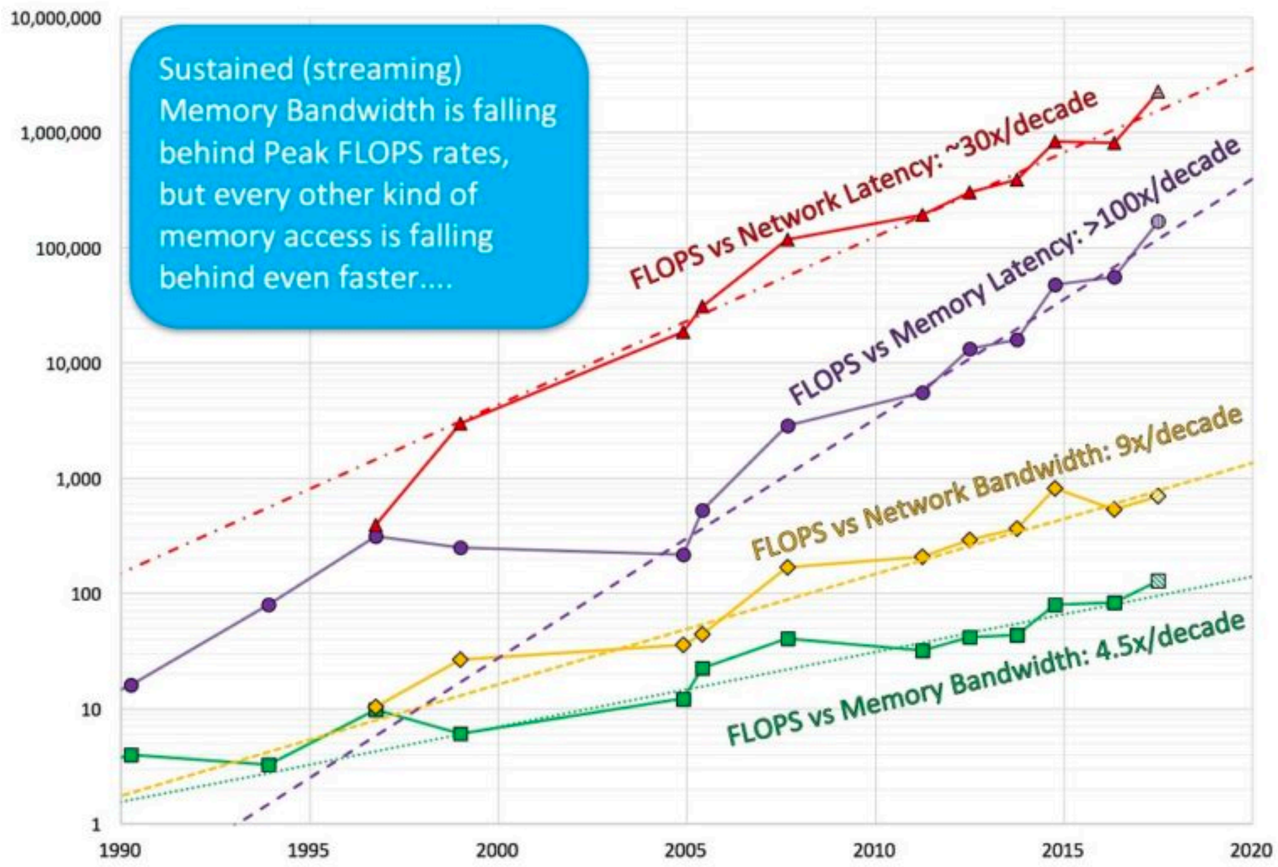
Machine balance

Machine balance =

$$\frac{\text{Peak Performance (flop/s)}}{\text{Peak Bandwidth (bytes/s)}}$$

Ideally, balance ~ 1, but usually, it is much **greater than 1** (and increasing).

Typical is 5-10 flops/byte.



https://www.hpcwire.com/2016/11/07/mccalpin-traces-hpc-system-balance-trends/

From UC Berkeley CS267

Computational (Arithmetic) Intensity

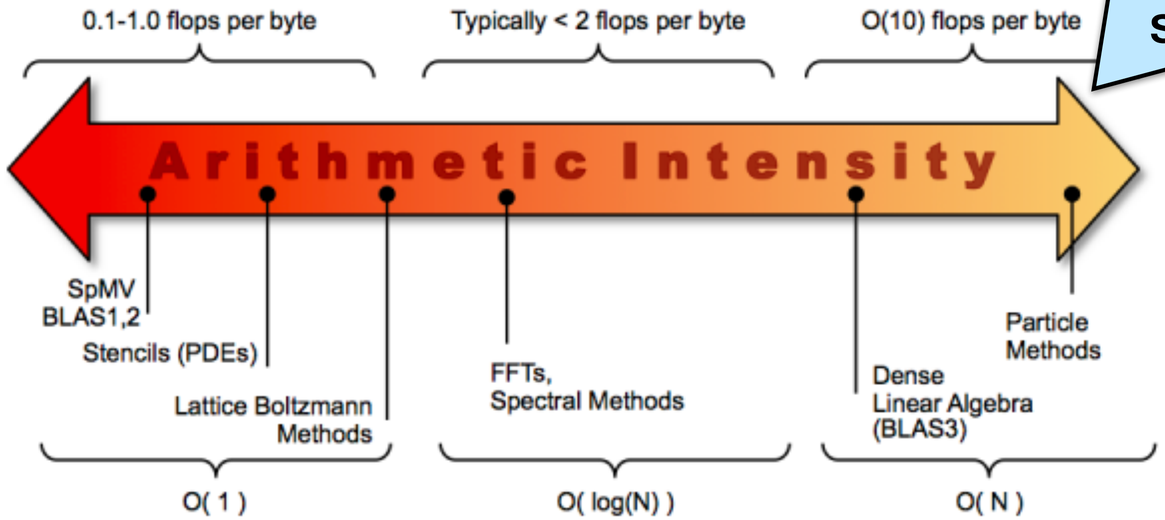
Depends on application

$$\text{Computational intensity (CI)} = \frac{\text{Flops performed}}{\text{Data moved (in bytes)}}$$

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(logn)

Ideal (infinite cache)

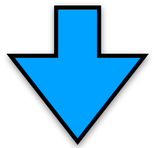
CI is a spectrum



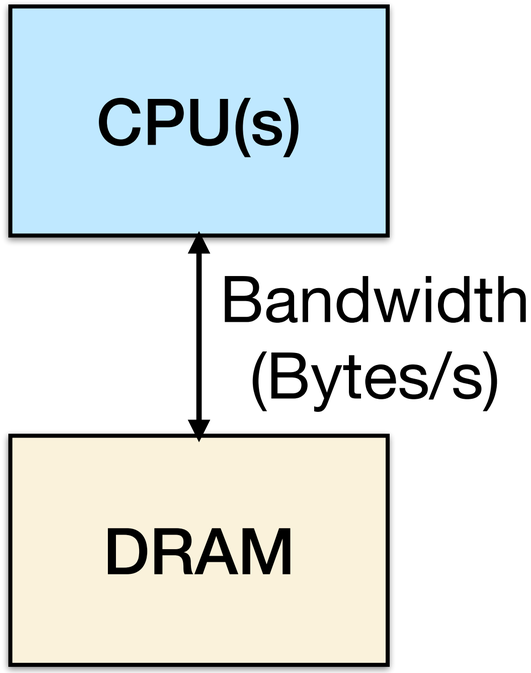
(DRAM) Roofline

Assuming idealized processors/caches and cold start (data in DRAM):

$$\text{Time} = \textbf{max} \begin{cases} \text{\#FP ops / Peak flop/s} \\ \text{\#bytes moved / peak bytes/s} \end{cases}$$

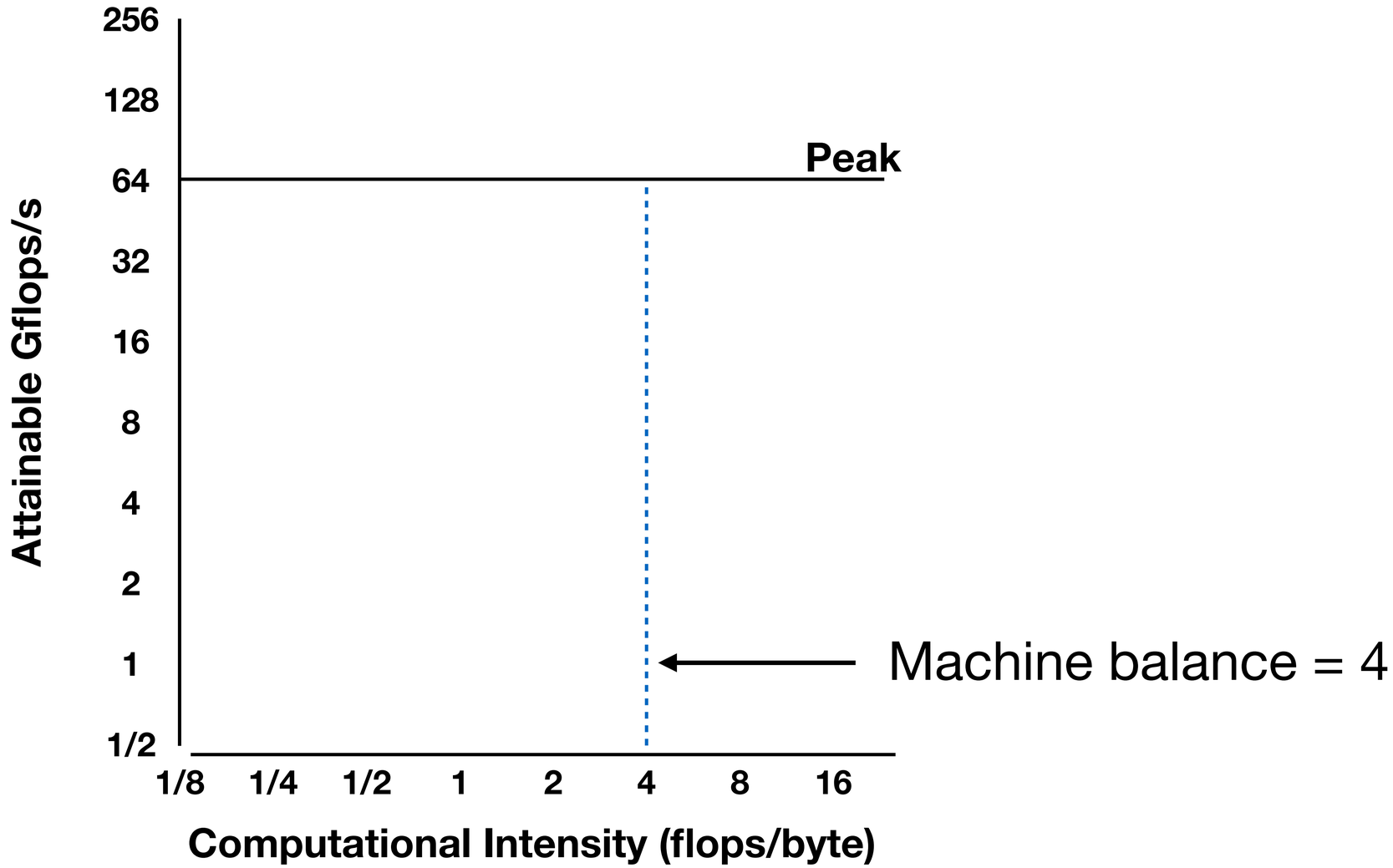


$$\frac{\text{\#FP ops}}{\text{Time}} = \textbf{min} \begin{cases} \text{Peak flops/s} \\ \text{(\#FP ops / \#bytes moved) * peak bytes/s} \end{cases}$$



Roofline performance model

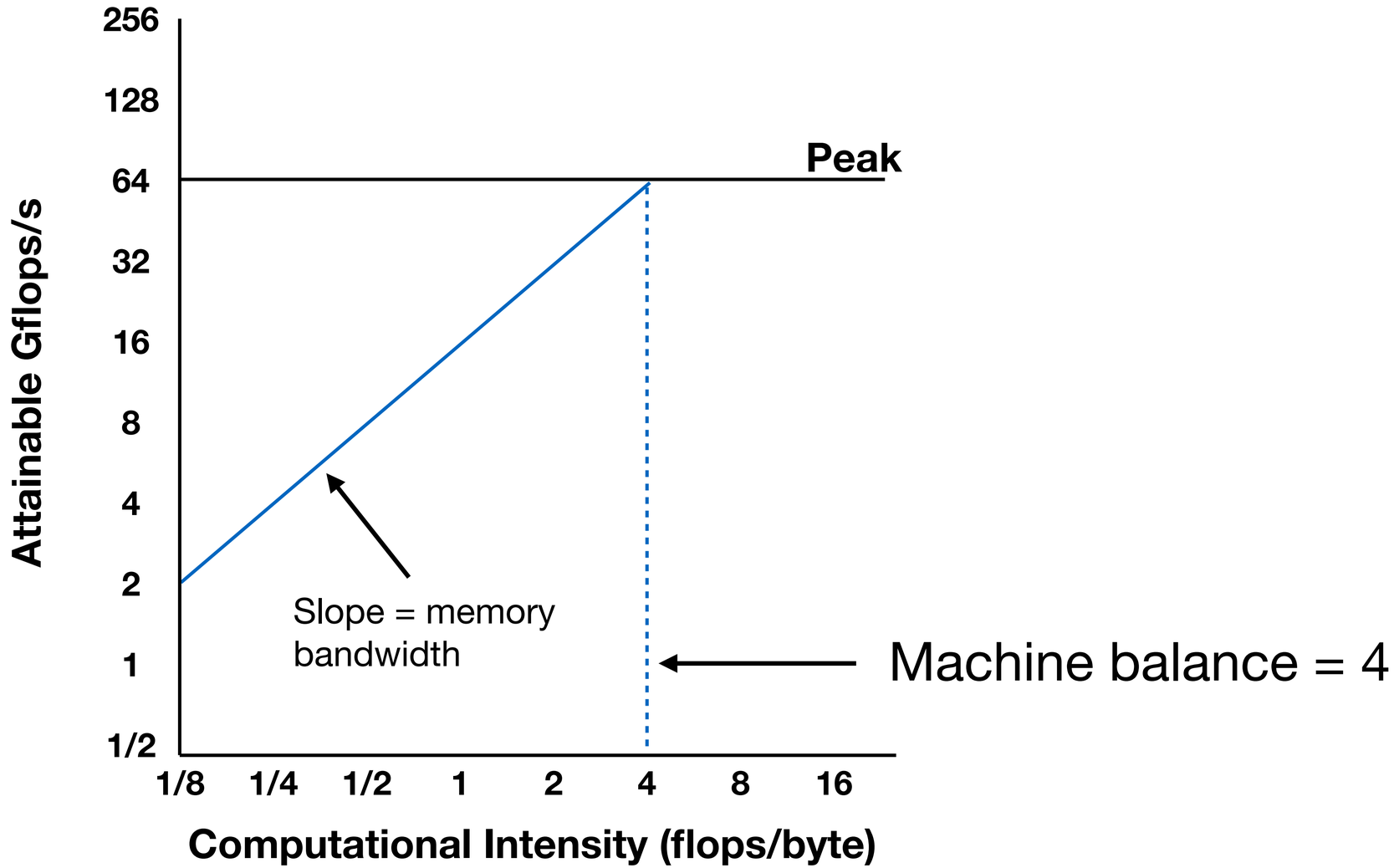
Example machine:



From UC Berkeley CS267

Roofline performance model

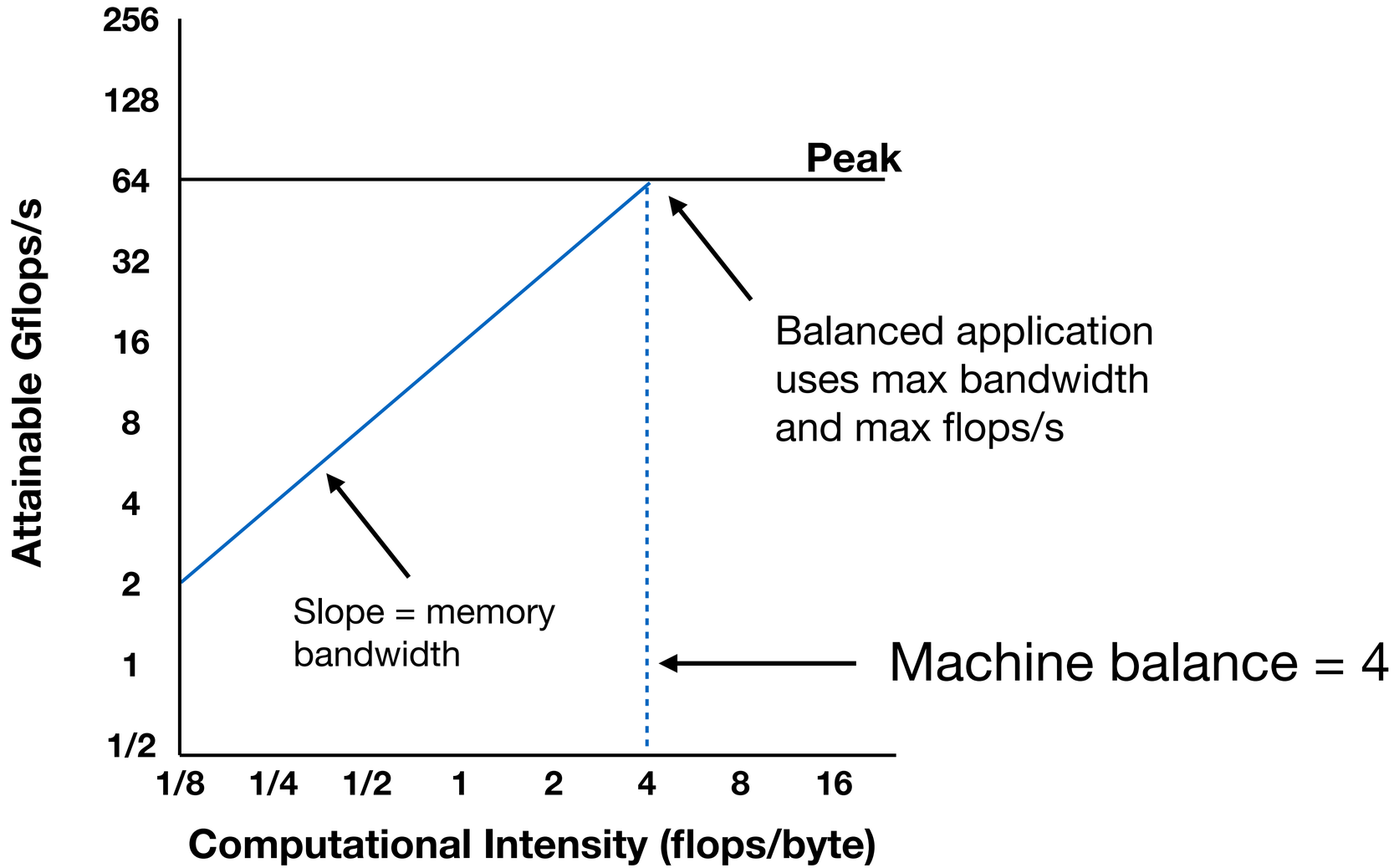
Example machine:



From UC Berkeley CS267

Roofline performance model

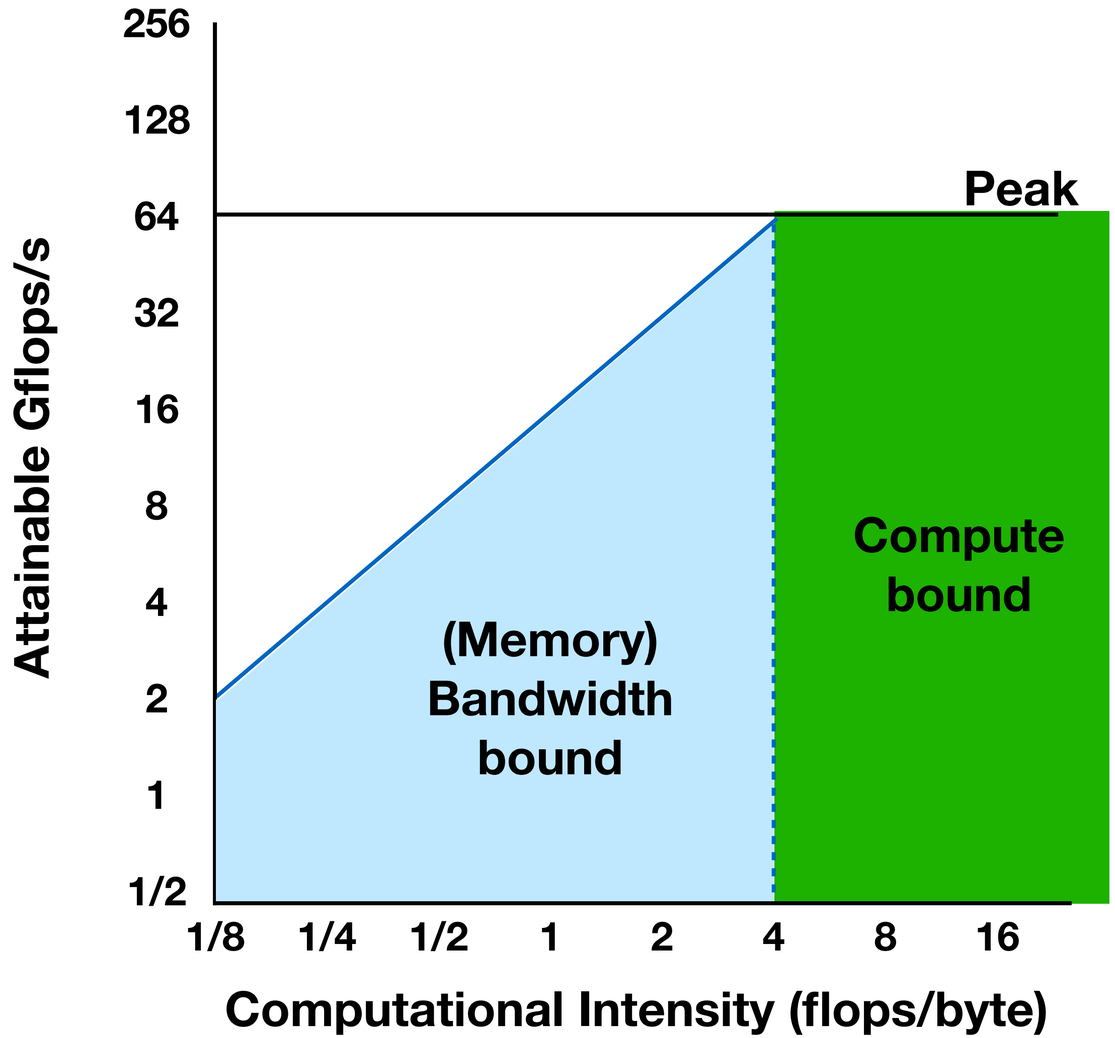
Example machine:



From UC Berkeley CS267

Roofline performance model

Example machine:

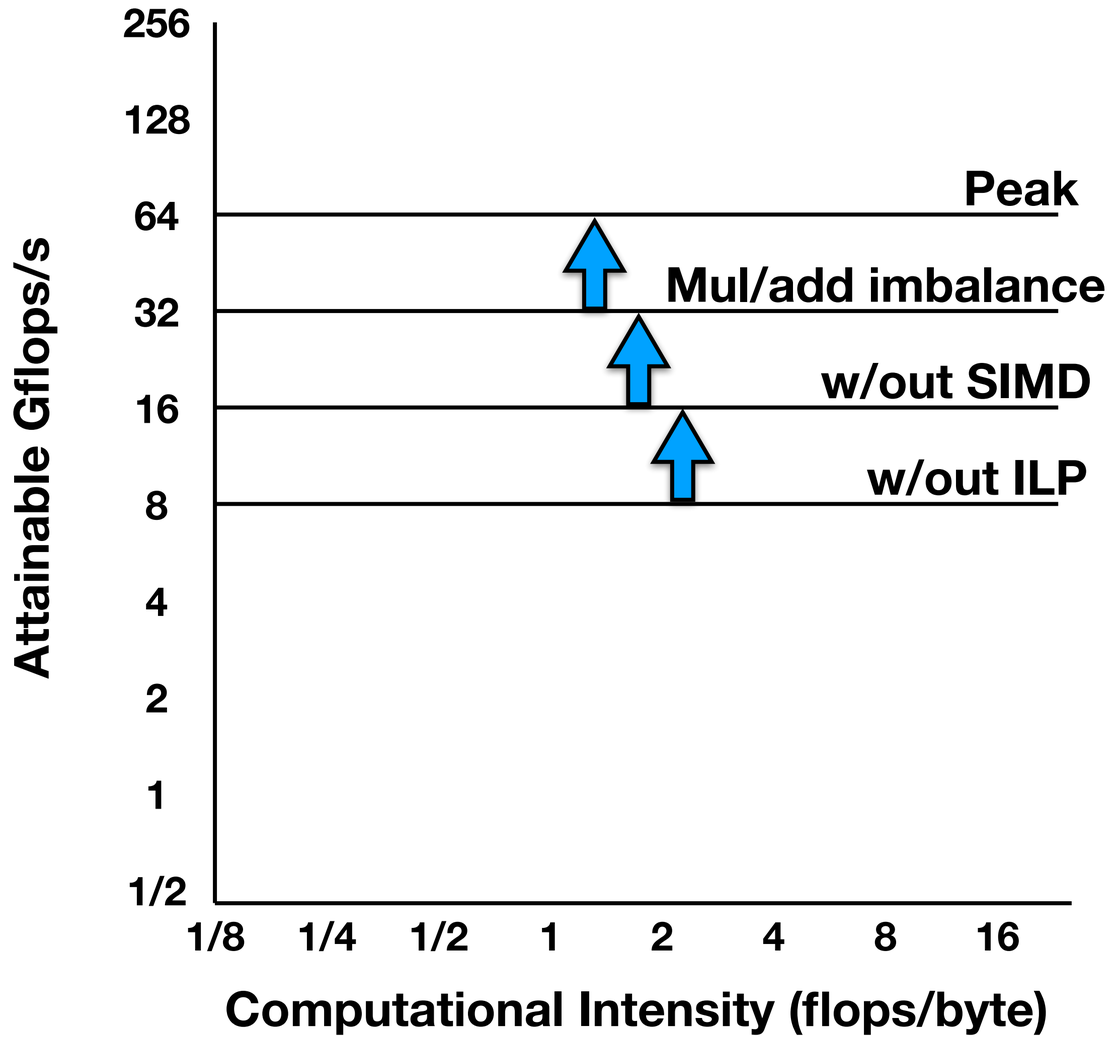


From UC Berkeley CS267

Three categories of software optimization

Maximizing attained in-core performance

Example machine:



No fused multiply-add (FMA), SIMD, instruction-level parallelism (ILP) will lower **what is attainable**.

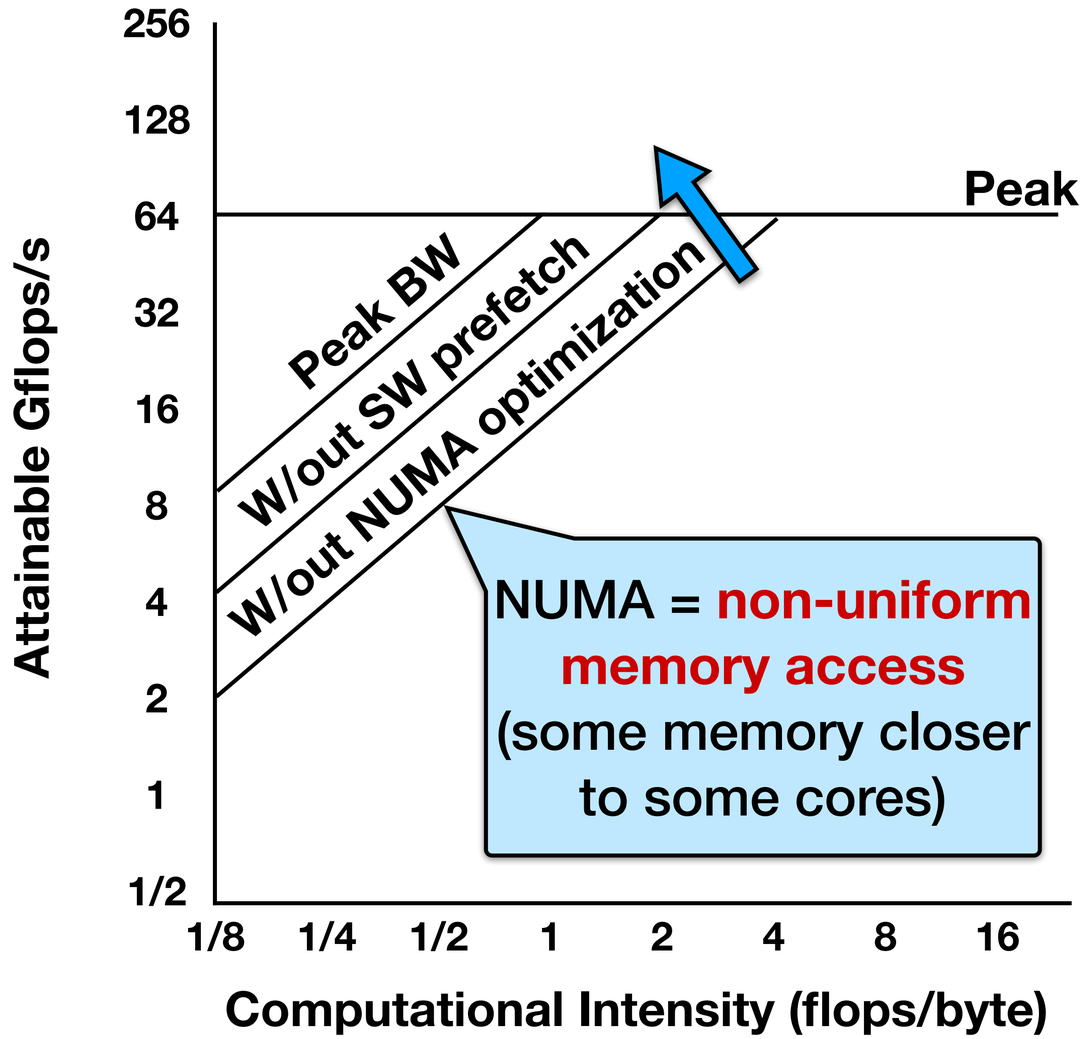
Software optimizations such as explicit SIMD can **increase the horizontal ceiling** (what can be expected from the compiler).

Other examples include loop unrolling, reordering, etc.

From UC Berkeley CS267

Maximizing attained memory bandwidth

Example machine:



Compilers won't give great out-of-the-box bandwidth.

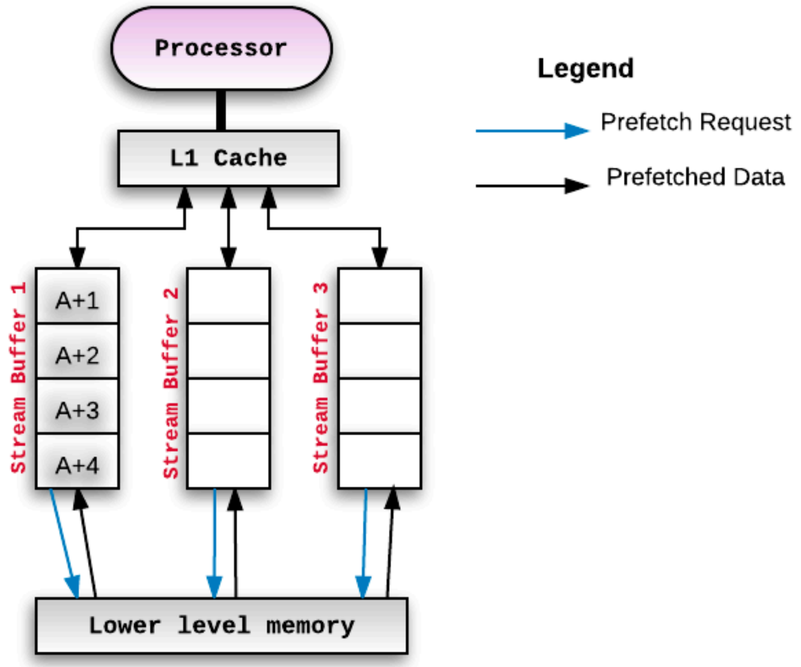
Increase bandwidth ceilings:

- NUMA aware allocation and parallelization
- Software prefetching
- Maximize memory-level parallelism (MLP)

Cache prefetching

Prefetching is a technique that fetches instructions / data from their original storage **before they are needed**.

Needed to achieve **full memory bandwidth** capabilities

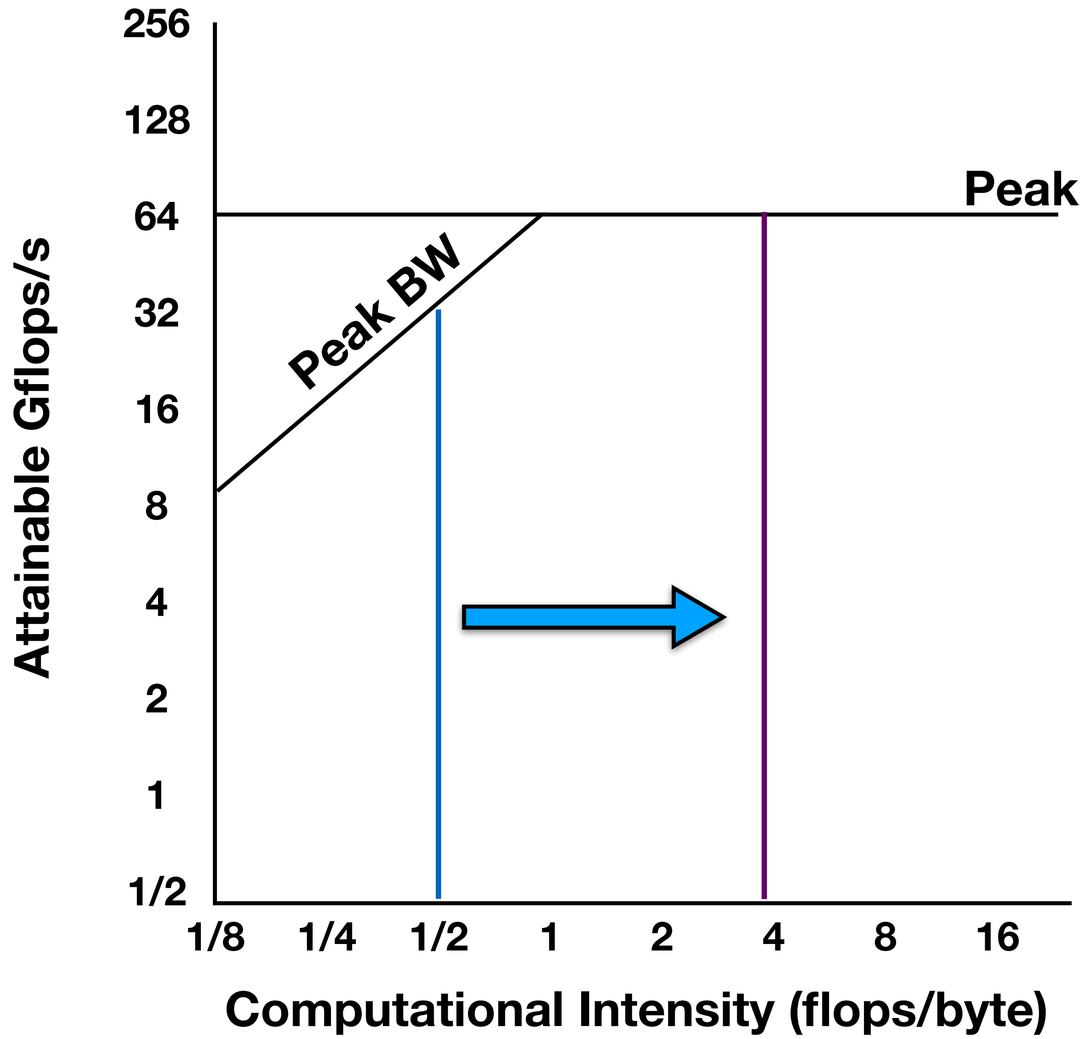


Hardware prefetching - triggered by two or more consecutive cache misses in succeeding or preceding cache lines (not necessarily unit stride).

Software prefetching - the compiler can insert instructions (or the programmer can manually add them) to prefetch specific addresses.

Minimizing memory traffic

Example machine:



Use performance counters to measure empirical arithmetic intensity (flops/byte).

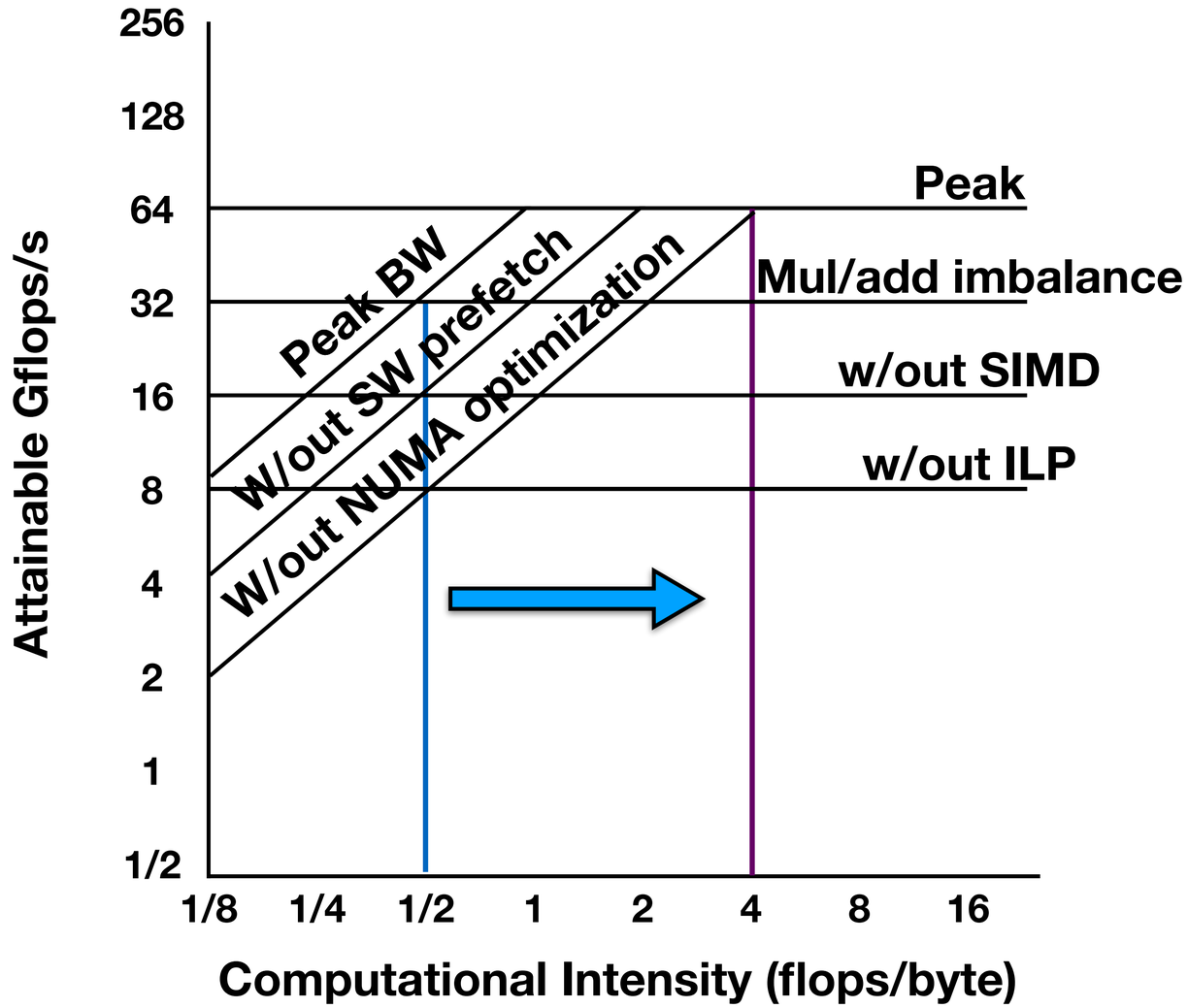
Might be much **lower than the best case**:

- Cache capacity / associativity
- Pad structures to avoid conflict misses
- Use cache blocking to avoid capacity misses

From UC Berkeley CS267

Effective Roofline (before and after)

Example machine:



Before optimization, traffic, and limited bandwidth - **performance is limited** to a very narrow window.

After optimization, ideally, performance is significantly better.

From UC Berkeley CS267