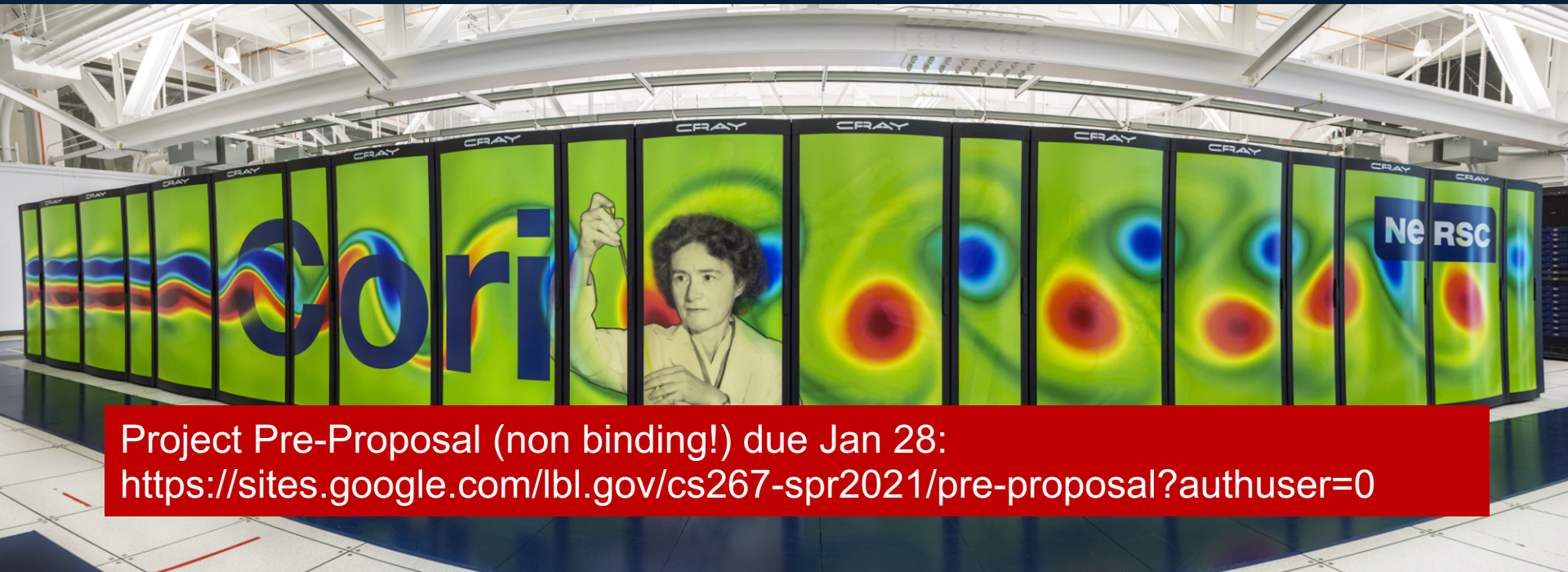
Applications of Parallel Computers

Memory Hierarchies and Matrix Multiplication

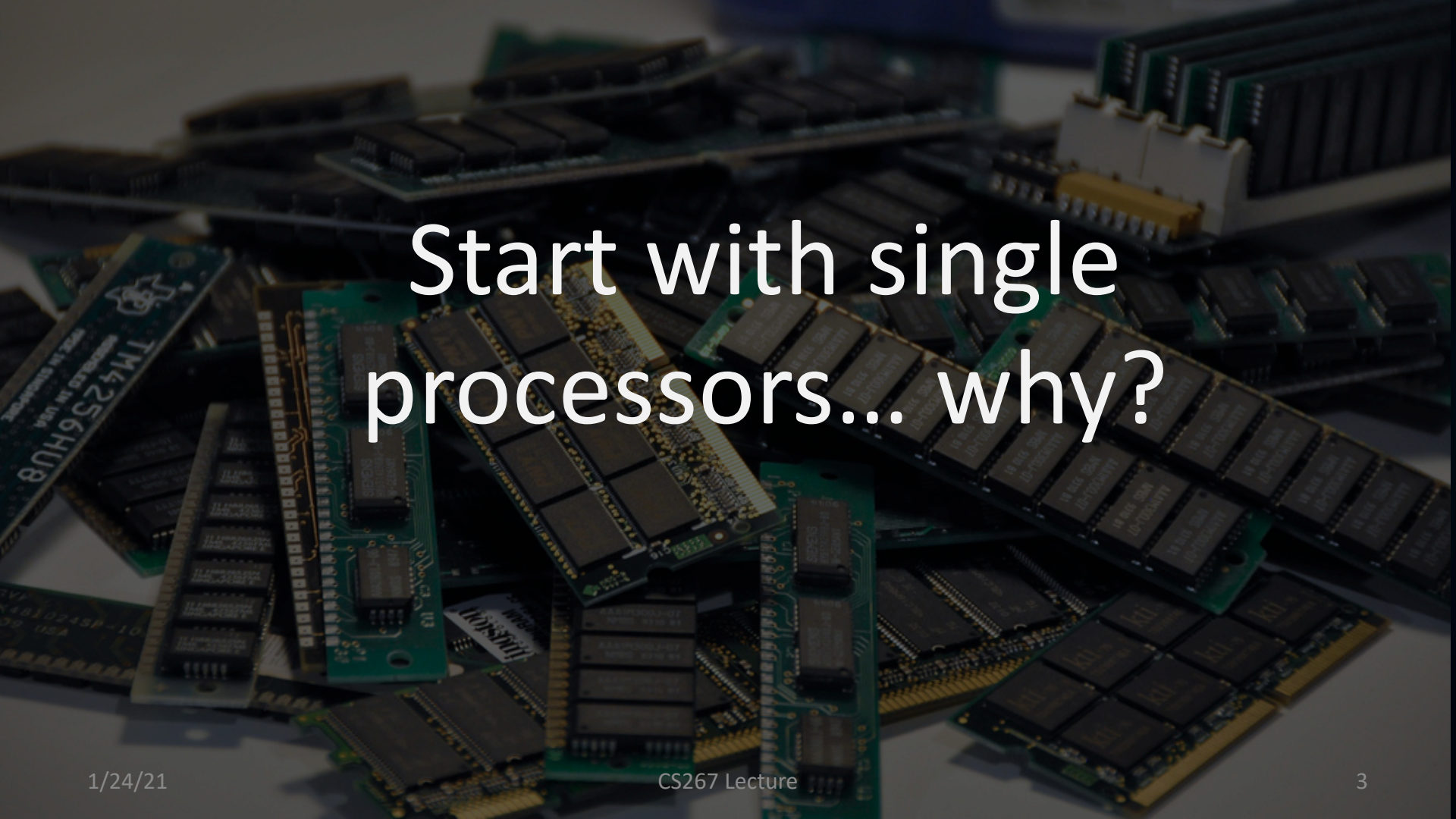
https://sites.google.com/lbl.gov/cs267-spr2021



Project Pre-Proposal (non binding!) due Jan 28:
https://sites.google.com/lbl.gov/cs267-spr2021/pre-proposal?authuser=0

Possible conclusions from today's lecture

- "Computer architectures are fascinating!"
- "These optimized algorithms are fascinating!"
- "I hope that most of the time I can call libraries so I don't have to worry about this!"
- "I wish the compiler would handle everything for me."
- "I would like to write a compiler that would handle all of these details."
- "I want to understand how Meltdown/Spectre work"

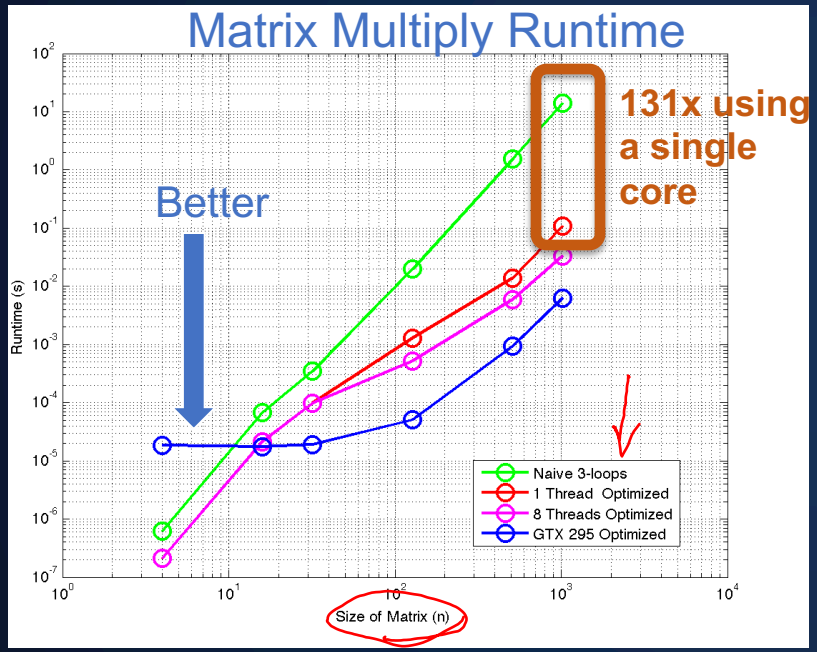


Start with single processors… why?

Looking Under the Hood

- Most applications run at < 10% of the "peak" performance
- Much of the performance is lost on a single processor, moving data

In a course on parallel programming why care about serial performance?



Parallelizing slow serial code only produces slow parallel code.

Linear speedup is not necessarily fast code

Matrix multiply is extreme, but 10x from sequential optimizations is common.

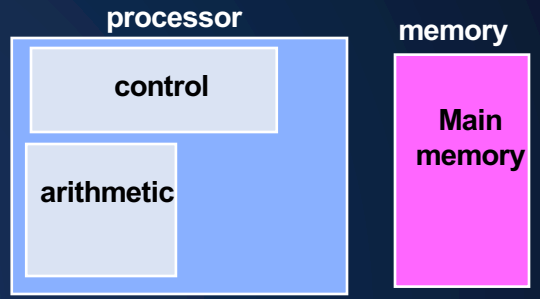
Note: Both x and y axes are log-scale

Outline

- Costs in modern processors
  - Idealized models and actual costs
- Memory hierarchies
- Parallelism in a single processor
- Case study: Matrix multiplication
- Optimizations in Practice

Idealized Uniprocessor Model

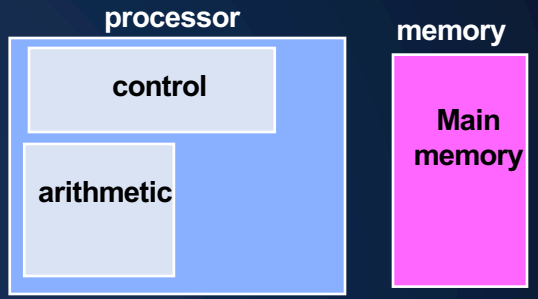
- **Processor names variables:**
  - Integers, floats, doubles, pointers, arrays, structures, etc
- **Processor performs operations on those variables:**
  - Arithmetic, logical operations, etc.
- **Processor controls the order, as specified by program**
  - Branches (if), loops, function calls, etc.



- ***Idealized Cost***
  - Each operation (+,*, etc.) has roughly the same cost
  - And reading/writing variables is "free"

Idealized Uniprocessor Model

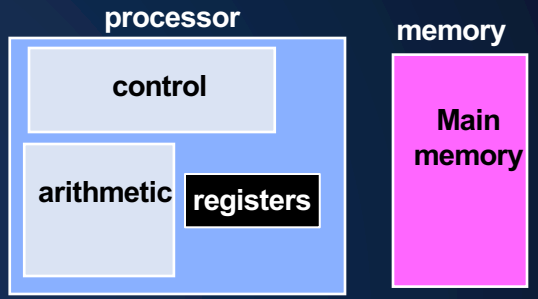
- **Processor names variables:**
  - Integers, floats, doubles, pointers, arrays, structures, etc
  - *really bytes, words, etc. in its address space*
- **Processor performs operations on those variables:**
  - Arithmetic, logical operations, etc.



- **Processor controls the order, as specified by program**
  - Branches (if), loops, function calls, etc.
  - *Compiler translates into "obvious" lower level instructions*
  - *Hardware executes instructions in order specified by compiler*
  - *Read returns the most recently written data*
- ***Idealized Cost***
  - Each operation (+,*, etc.) has roughly the same cost
  - And reading/writing variables is "free"

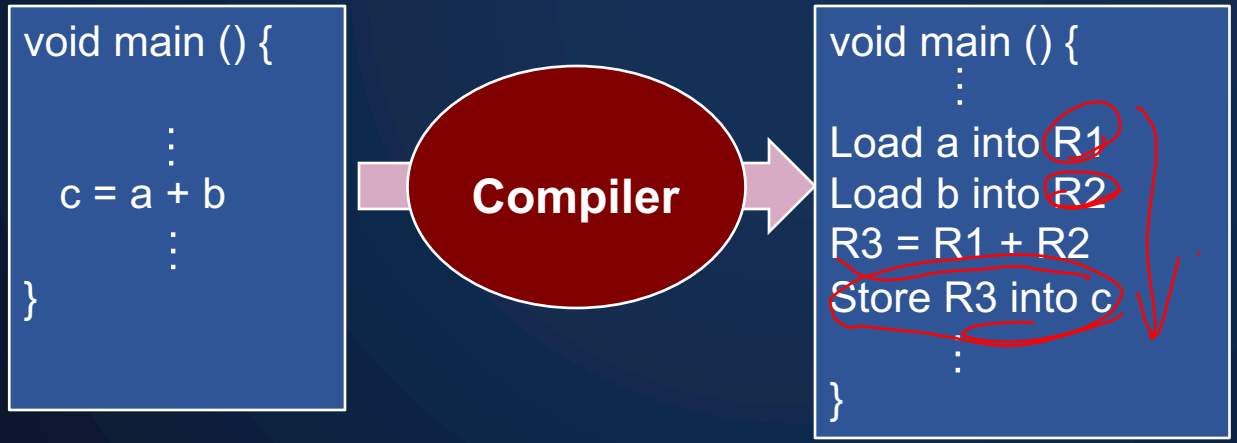
Idealized Uniprocessor Model

- **Processor names variables:**
  - Integers, floats, doubles, pointers, arrays, structures, etc
  - *really bytes, words, etc. in its address space*
- **Processor performs operations on those variables:**
  - Arithmetic operations, etc. only on values in registers
  - Load/Store variables between memory and registers
- **Processor controls the order, as specified by program**
  - Branches (if), loops, function calls, etc.
  - *Compiler translates into "obvious" lower level instructions*
  - *Hardware executes instructions in order specified by compiler*
  - *Read returns the most recently written data*
- ***Idealized Cost***
  - Each operation (+,*, &, etc.) has roughly the same cost (on "free" registers)
  - Load/store is 100x the cost of +,*,&, etc.



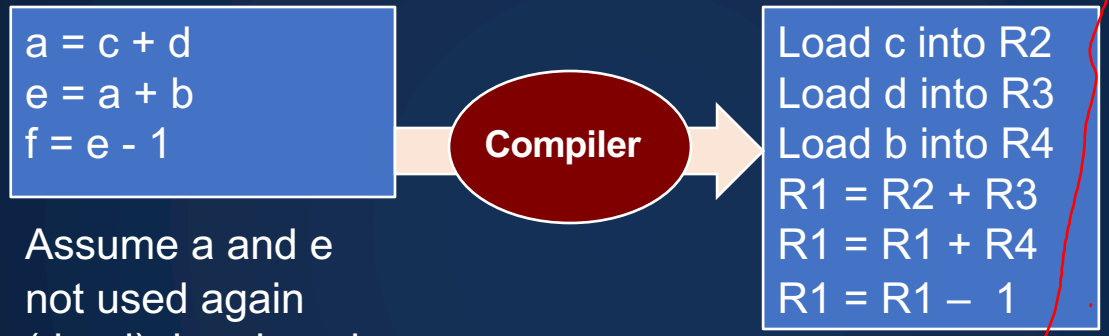
Compilers and assembly code

- Compilers for languages like C/C++ and Fortran:
  - Check that the program is legal
  - Translate into assembly code
  - Optimizes the generated code



Compilers Manage Memory and Registers

- Compiler performs "register allocation" to decide when to load/store vs reuse



Register allocation in first Fortran compiler in 1950s, graph coloring in 1980. JITs may use something cheaper

Compiler Optimizes Code

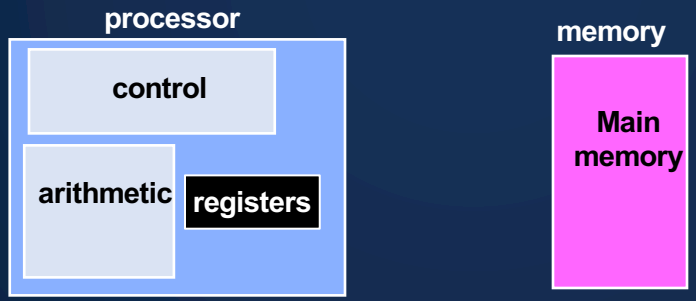
- Besides register allocation, the compiler performs optimizations:
  - Unrolls loops (because control isn't free)
  - Fuses loops (merge two together)
  - Interchanges loops (reorder)
  - Eliminates dead code (the branch never taken)
  - Reorders instructions to improve register reuse and more
  - Strength reduction (e.g., shift left rather than multiply by 2)
- Why is this your problem?
  - Because sometimes it does the best thing possible
  - But other times it does not…

Outline

- Processors and registers
- Memory hierarchies
  - Temporal and spatial locality
  - Basics of caches
  - Use of microbenchmarks to characterize performance
- Parallelism in a single processor
- Case study: Matrix multiplication
- Optimizations in Practice

More Realistic Uniprocessor Model

- **Memory accesses (load / store) have two costs**
  - **Latency:** cost to load or store 1 word (α)
  - **Bandwidth:** Average rate (bytes/sec) to load/store a large chunk
    or inverse time/byte, β



Latency vs Bandwidth

- Memory systems and networks

Bandwidth

≈ data throughput (bits/s(bytes/second)

Low Bandwidth

High Bandwidth

Latency

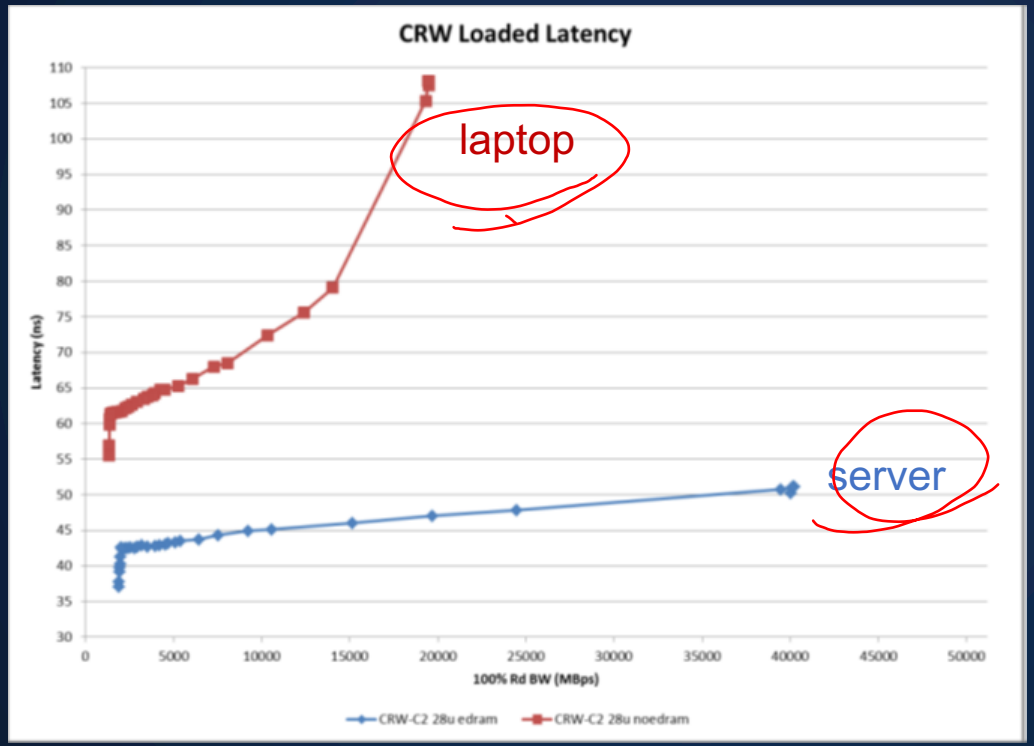
≈ delay due data travel time (m (secs)

Low Latency

High Latency

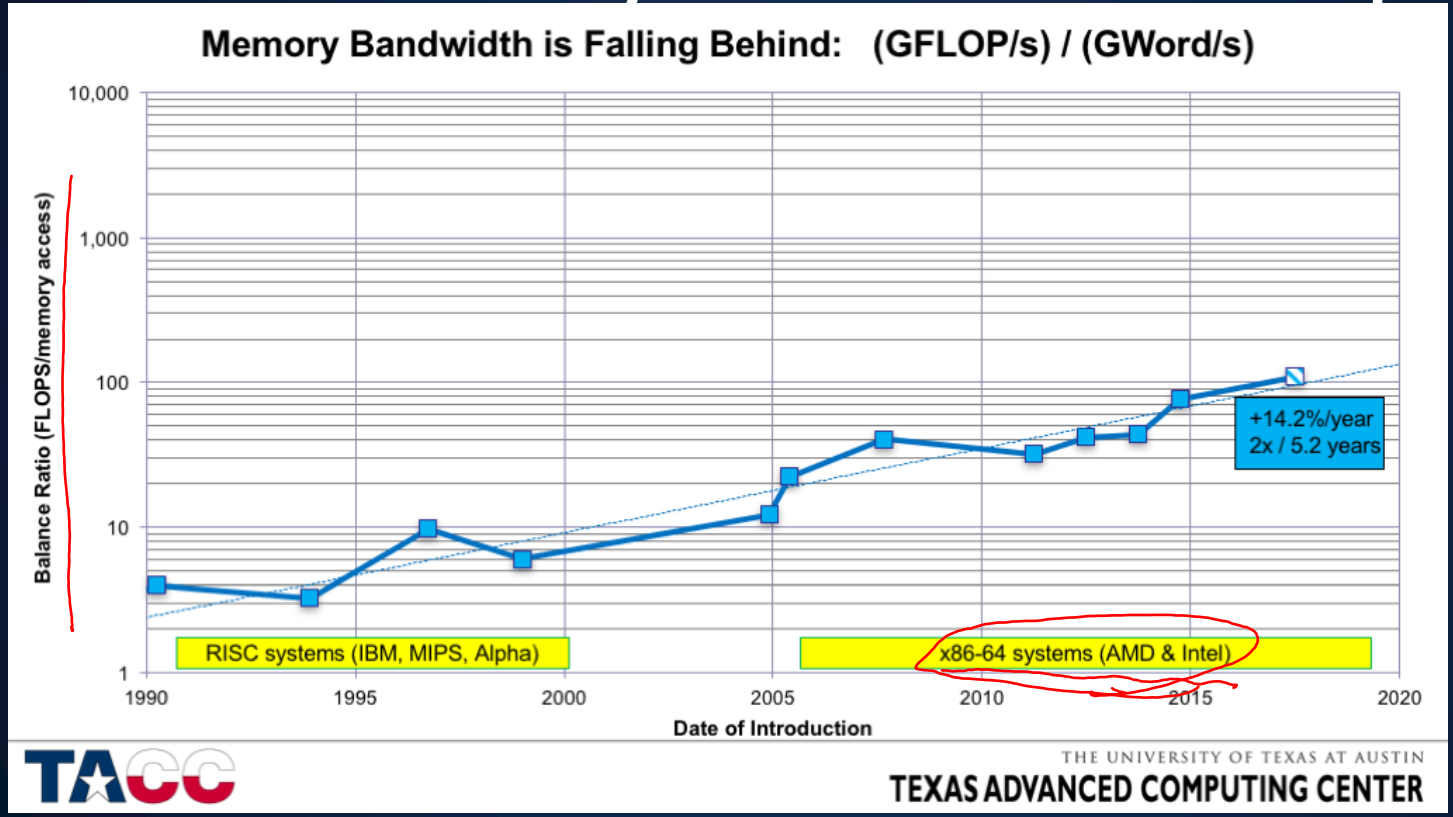
Image: Katie Hempenius

Latency and Bandwidth on Intel Haswell



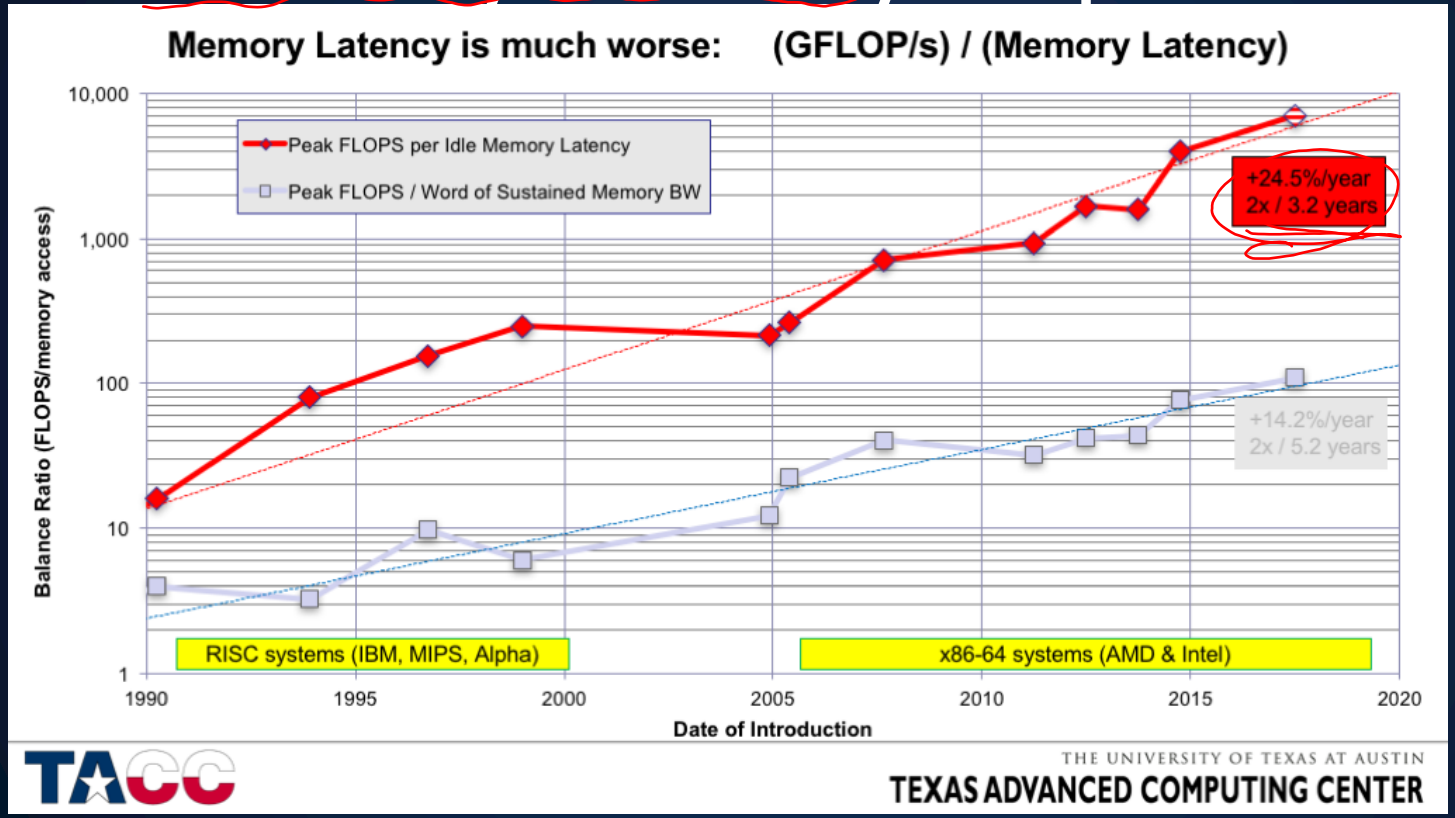
Latency (α) 10s-100 ns
Bandwidth 2-40 GB/s
(β) .025-.5 ns/B

Memory Bandwidth Gap



John McCalpin, SC16 invited talk

Memory Latency Gap is Worse

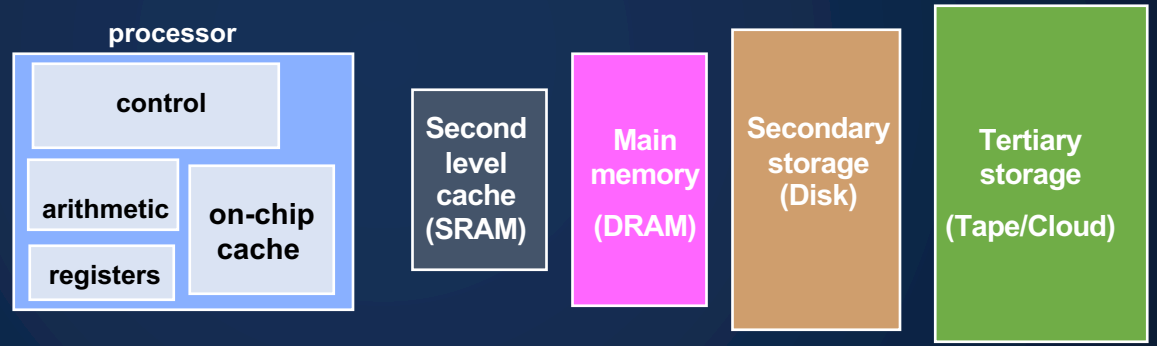


John McCalpin, SC16 invited talk

Memory Hierarchy

- Most programs have a high degree of locality
  - **spatial locality:** accessing things nearby previous accesses
  - **temporal locality:** reusing an item that was previously accessed

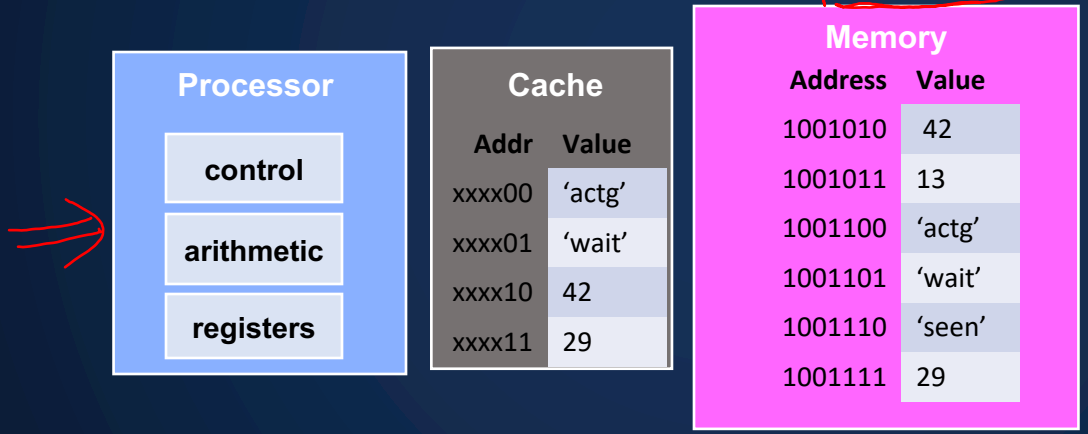
Hierarchy speeds up average case



Speed	1ns	10ns	100ns	10ms	10sec
Size	KB	MB	GB	TB	PB

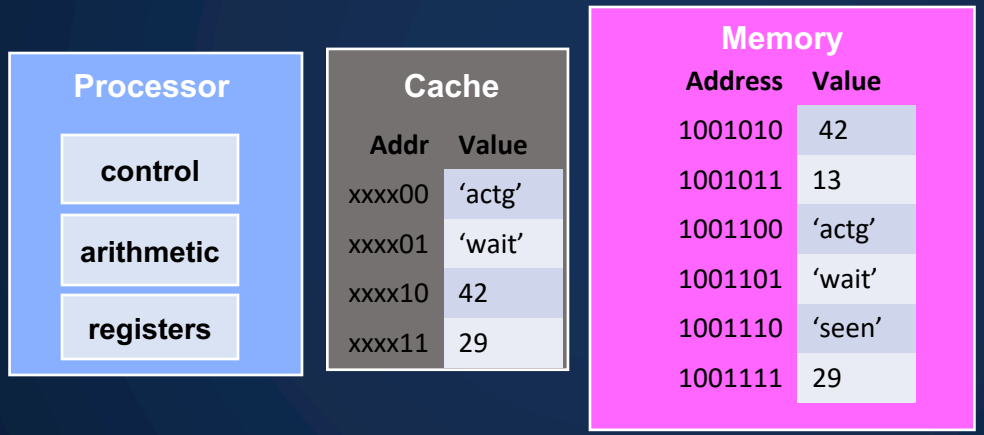
Cache Basics

- **Cache is fast (expensive) memory which keeps copy of data; it is hidden from software**
  - **Simplest example: data at memory address xxxxxxx10 is stored at cache location 10**



Cache Basics

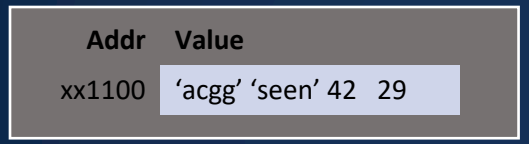
- **Cache is fast (expensive) memory which keeps copy of data; it is hidden from software**
  - **Simplest example: data at memory address xxxxxxx10 is stored at cache location 10**



- **Cache hit:** in-cache memory access—cheap
- **Cache miss:** non-cached memory access—expensive
  - **Need to access next, slower level of memory**

Cache Basics

- **Cache line length: # of bytes loaded together in one entry**
  - Ex: If either xxxxx1100 or xxxxx1101 is loaded, both are



- **Associativity**
  - direct-mapped: only 1 address (line) in a given range in cache
    - Data at address xxxxx1101 stored at cache location 1101
    - (Only 1 such value from memory)

Why Have Multiple Levels of Cache?

- **On-chip vs. off-chip**
  - On-chip caches are faster, but smaller
- **A large cache has delays**
  - Hardware to check longer addresses in cache takes more time
  - Associativity, which gives a more general set of data in cache, also takes more time

- **Some examples:**
  - Cray T3E eliminated one cache to speed up misses
  - Intel Haswell (Cori Ph1) uses a Level 4 cache as "victim cache"
- **There are other levels of the memory hierarchy**
  - Register, pages (TLB, virtual memory), …
  - And it isn't always a hierarchy

Approaches to Handling Memory Latency

- **Reuse values in fast memory (bandwidth filtering)**
  - need temporal locality in program
- **Move larger chunks (achieve higher bandwidth)**
  - need spatial locality in program
- **Issue multiple reads/writes in single instruction (higher bw)**
  - vector operations require access set of locations (typically neighboring)
- **Issue multiple reads/writes in parallel (hide latency)**
  - prefetching issues read hint
  - delayed writes (write buffering) stages writes for later operation
  - both require that nothing dependent is happening (parallelism)

concurrency

How much concurrency do you need?

To run at bandwidth speeds rather than latency

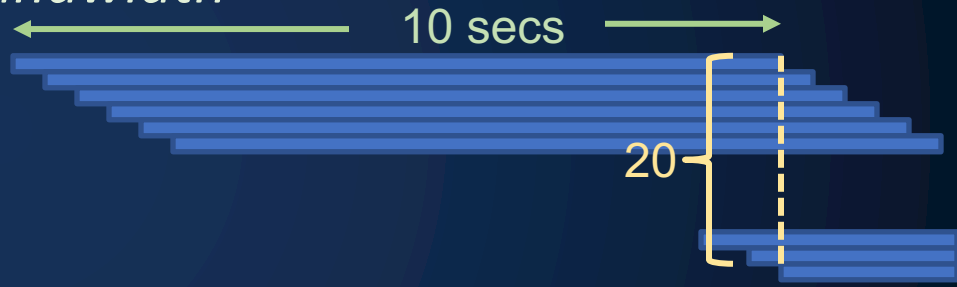
- Little's Law from queuing theory says:

  *concurrency = latency * bandwidth*

- For example:

  latency = 10 sec

  bandwidth = 2 Bytes/sec

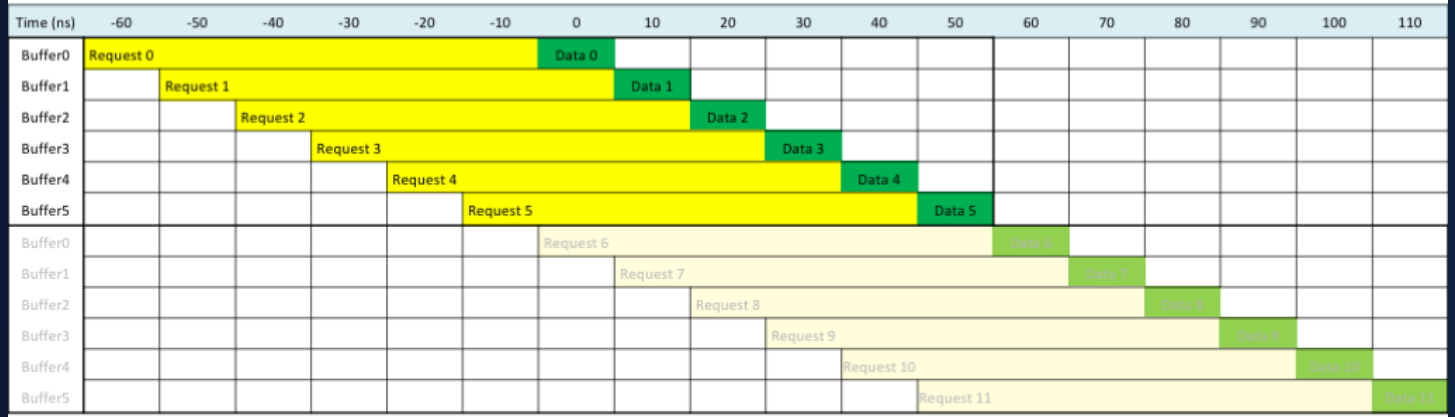


- Requires 20 bytes in flight to hit bandwidth speeds
- That means finding 20 independent things to issue

More realistic example

Little's Law: illustration for 2005-era Opteron processor

60 ns latency, 6.4 GB/s (=10ns per 64B cache line)



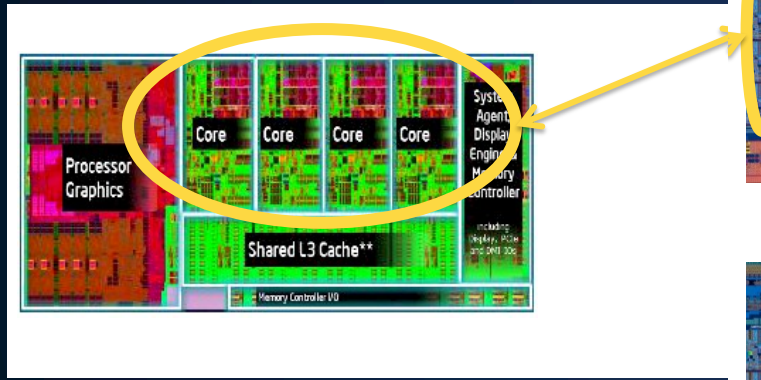
- 60 ns * 6.4 GB/s = 384 Bytes = 6 cache lines
- To keep the pipeline full, there must always be 6 cache lines "in flight"
- Each request must be launched at least 60 ns before the data is needed

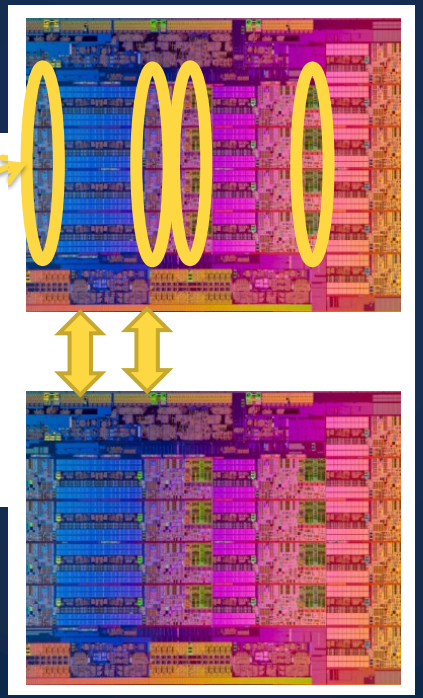
TACC

THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER

Cori Haswell Node

Quad Core Laptop





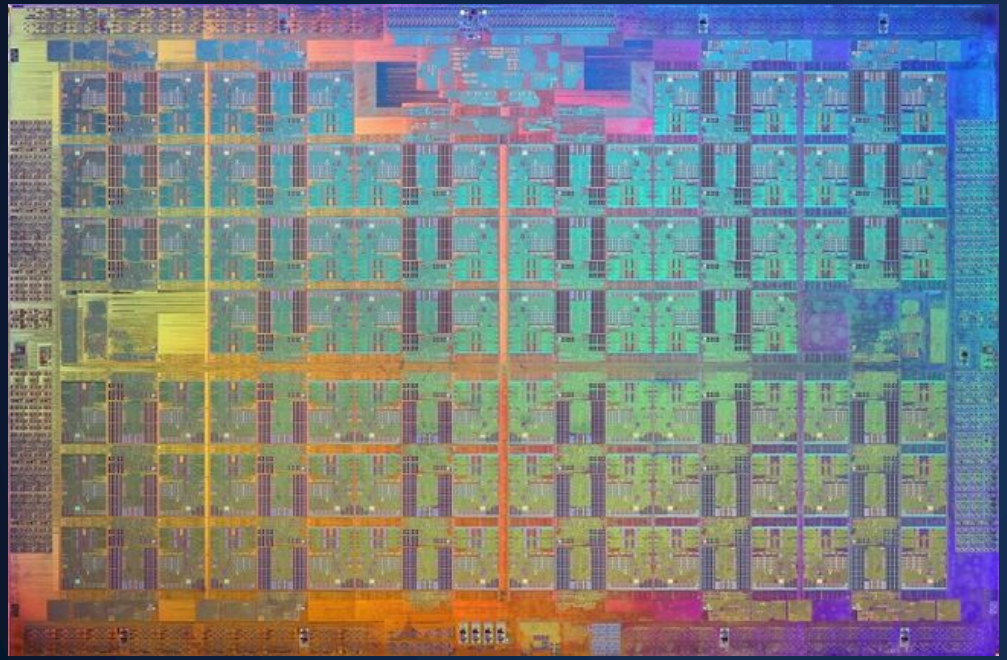
Haswell node

- 2 sockets
- 16 cores each
- 2.3 GHz
- 2 x 256-bit-wide vector
- 64-bit FMA multiply-add

Peak (in theory):

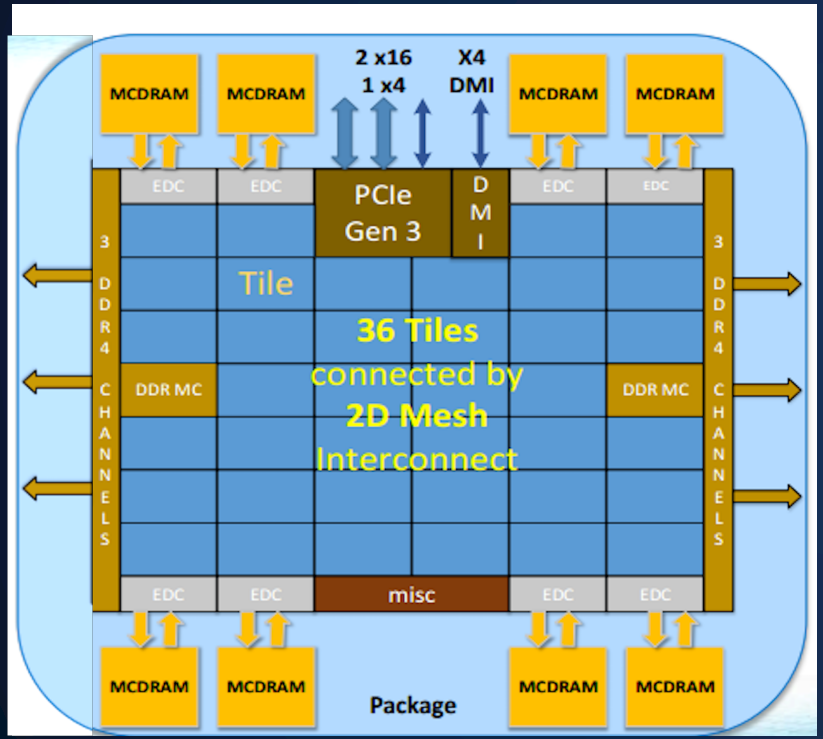
- 2*256/64*2*2.3 = 36.8 Gflops/core
- 2*36.8*16 = 1.2 TFlops/node
- Memory: 128 GB RAM

Cori KNL Node



- 68 cores
- Each core is slower than a Haswell core
- Smaller area
- Less power

KNL Nodes



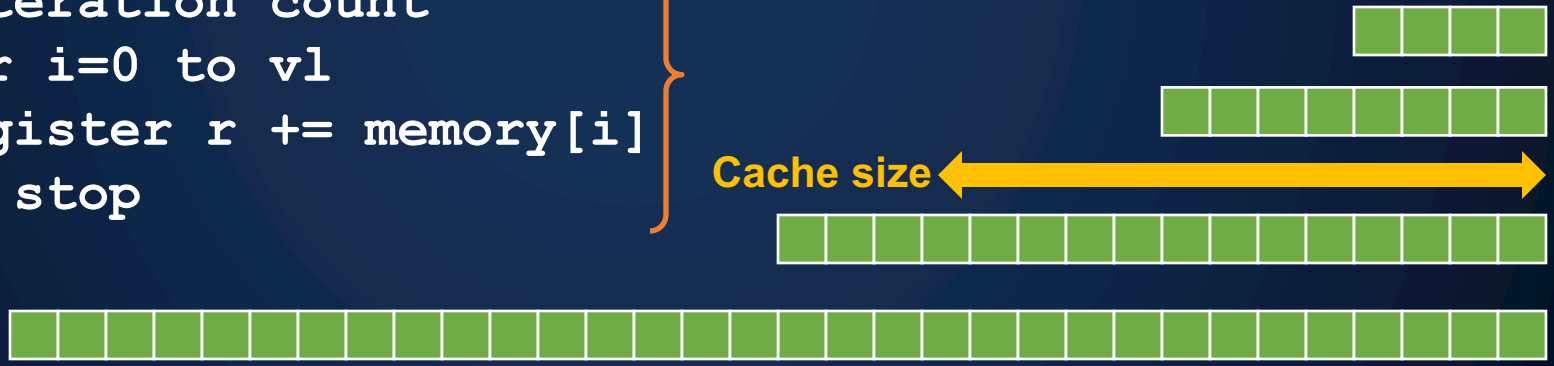
- **1 KNL socket Intel Xeon Phi Processor 7250 ("Knights Landing")**
- **68 cores @ 1.4 GHz**
  - 2x 512-bit vector units (AVX-512)
  - 64 KB L1 cache (32 i-cache, 32 data)
  - (Each core has 4 hardware threads)
  - 44.8 GFlops/core (theoretical peak)
- **Each tile (2 cores) shares a 1MB L2 cache**
- **Per node**
  - 3 TFlops/node (theoretical peak)
  - 96 GB DDR4 2400 MHz memory, six 16 GB DIMMs (102 GiB/s peak bandwidth)
  - 16 GB MCDRAM (multi-channel DRAM), >460 GB/s peak bandwidth

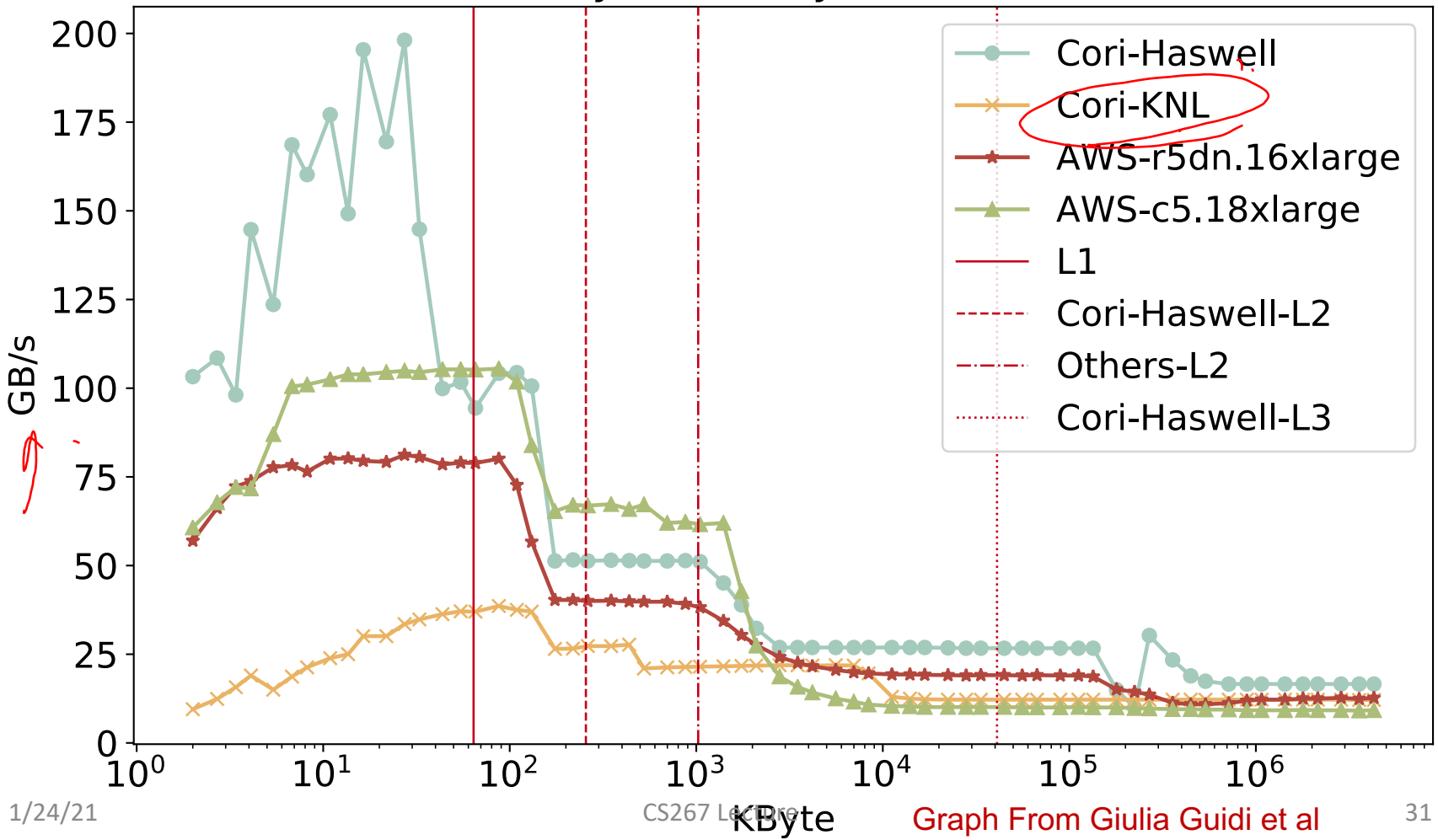
Memory Benchmark (CacheBench)

- Microbenchmark for memory system performance

```
for vl = all vector lengths
   memory[vl] //alloc+init
   timer start
   for iteration count
      for i=0 to vl
      register r += memory[i]
   timer stop
```

1 data point:
BW = bytes/word x vl / average time





Graph From Giulia Guidi et al

Take-Aways

- Performance is complicated
- Memory is hierarchical
  - Registers under compiler control
  - Caches (multiple) under hardware control
  - Order of magnitude between layers (speed and size)
- Trends: growing gap
- Little's Law: concurrency to overlap latency

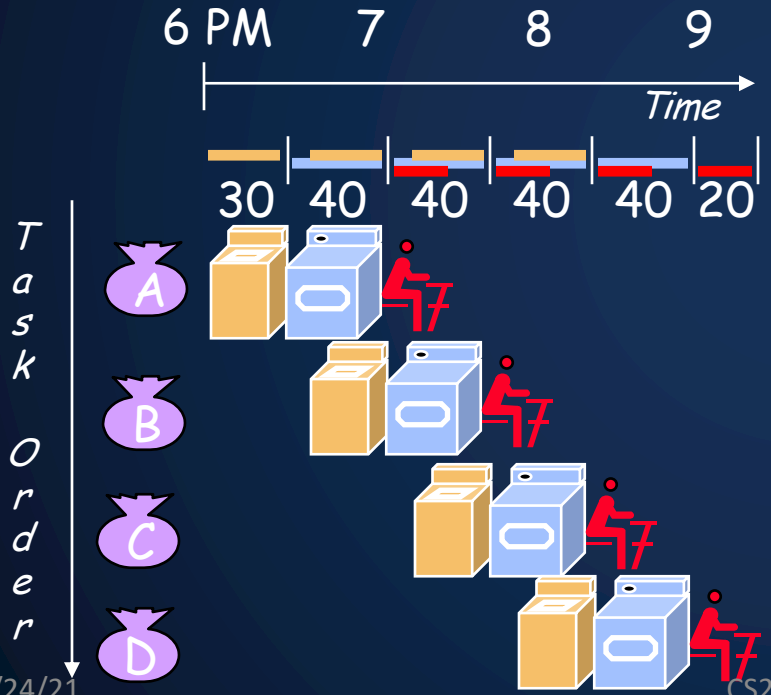
Outline

- Processors and registers
- Memory hierarchies
- Parallelism within single processors
  - Instruction Level Parallelism (ILP) and Pipelining
  - SIMD units
  - Special Instructions (FMA)
- Case study: Matrix Multiplication
- Optimization in practice

What is Pipelining?

Dave Patterson's Laundry example

Latency: wash (30 min) + dry (40 min) + fold (20 min) = 90 min

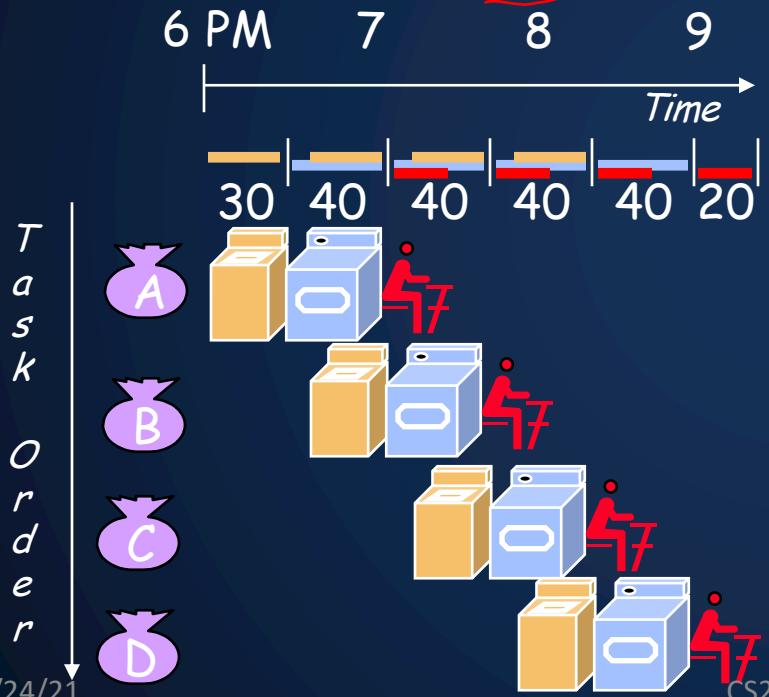


- In this example (4 loads):
  - Sequential execution takes 4 * 90min = 6 hours

What is Pipelining?

Dave Patterson's Laundry example

Latency: wash (30 min) + dry (40 min) + fold (20 min) = 90 min



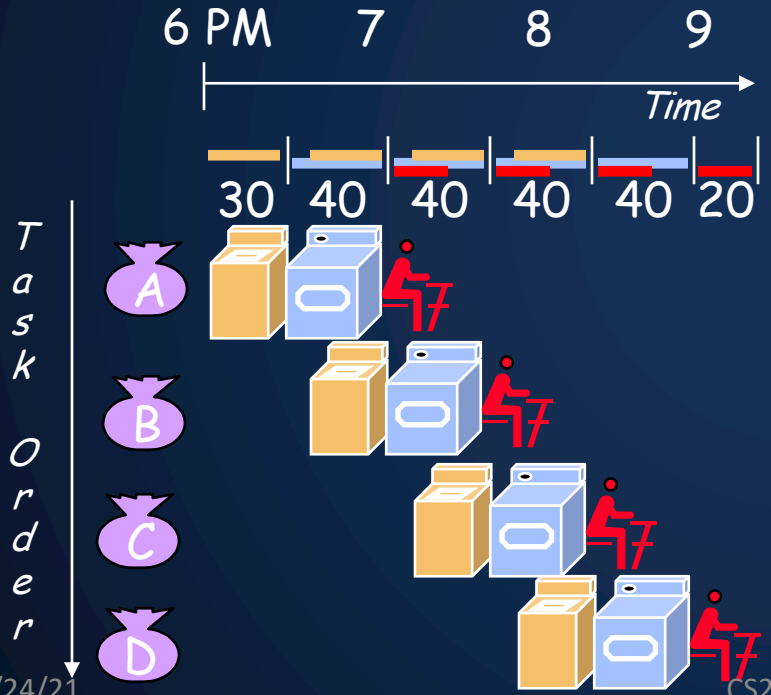
- In this example (4 loads of laundry):
  - Sequential execution takes 4 * 90min = 6 hours
  - Pipelined execution takes 30+4*40+20 = 3.5 hours
- Bandwidth = loads/hour
  = 4/6 l/h w/o pipelining
  = 4/3.5 l/h w pipelining
  <= 1.5 l/h w pipelining

通过流水线的到

What is Pipelining?

Dave Patterson's Laundry example

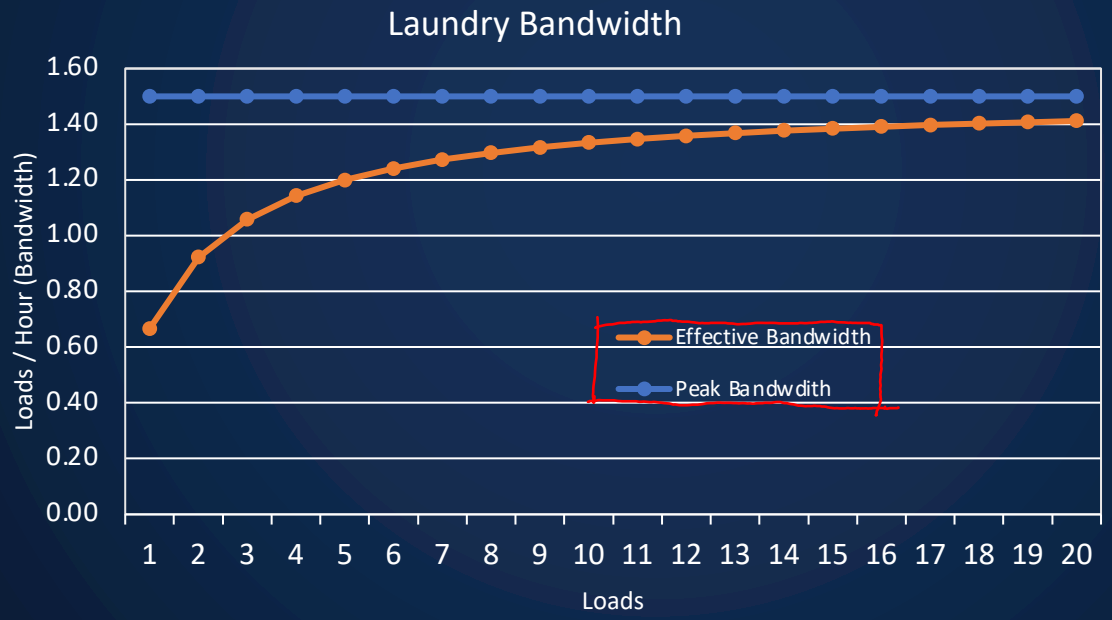
Latency: wash (30 min) + dry (40 min) + fold (20 min) = 90 min



- Pipelining doesn't change latency (90 min)
- Bandwidth limited by slowest pipeline stage
- Peak bandwidth is 1/slowest
- Speedup <= # of stages

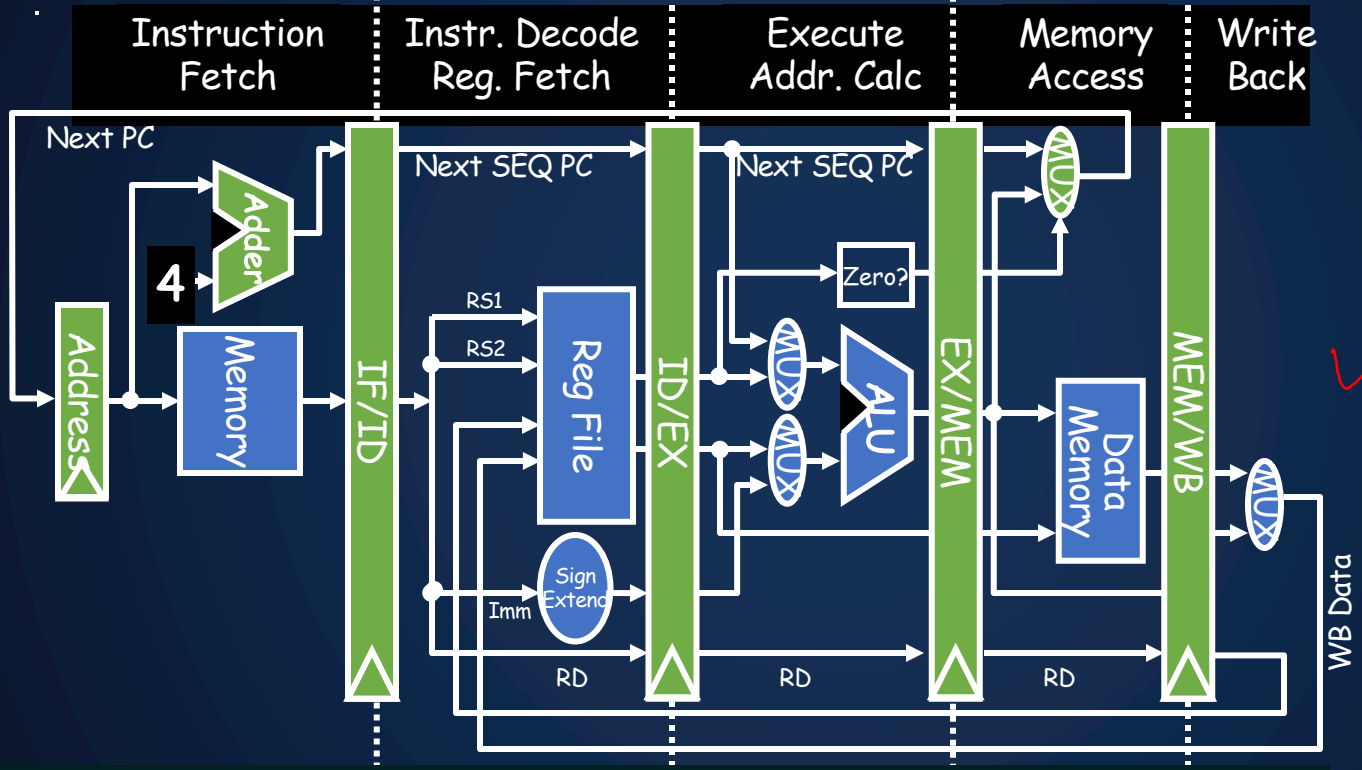
Laundry Bandwidth

- The system bandwidth is 1 load per 40 minutes (1.5 Loads / hour)
- How much of that you see depends on available loads that can be done in parallel



Example: 5 Steps of MIPS Datapath

Figure 3.4, Page 134 , CA:AQA 2e by Patterson and Hennessy



- Pipelining is also used within arithmetic units
  - a fp multiply may have latency 10 cycles, but throughput of 1/cycle

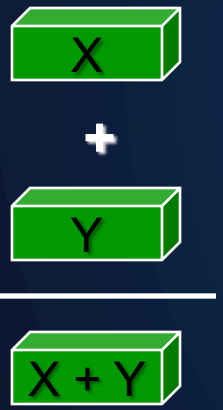
SIMD: Single Instruction, Multiple Data

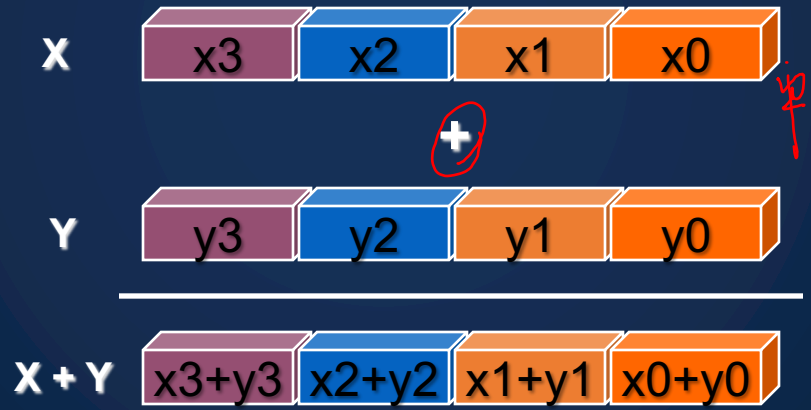
Scalar operation:

1 input → 1 result

SIMD (Vector) operation:

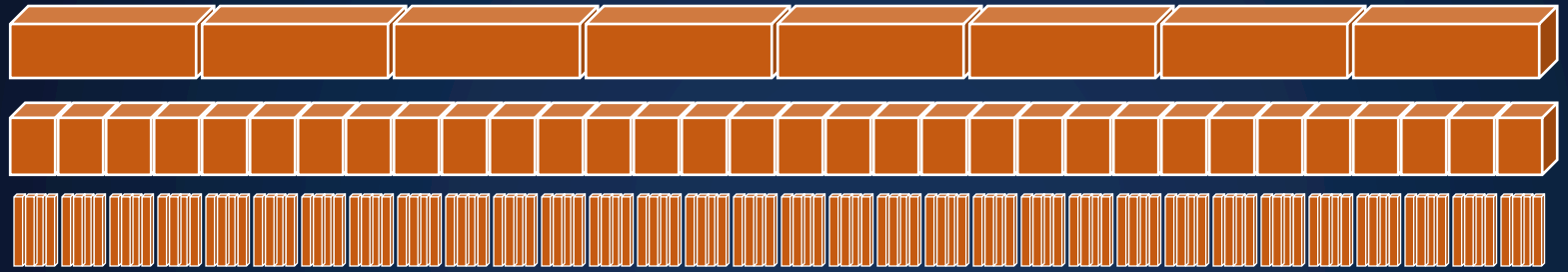
1 operation → n inputs, n results





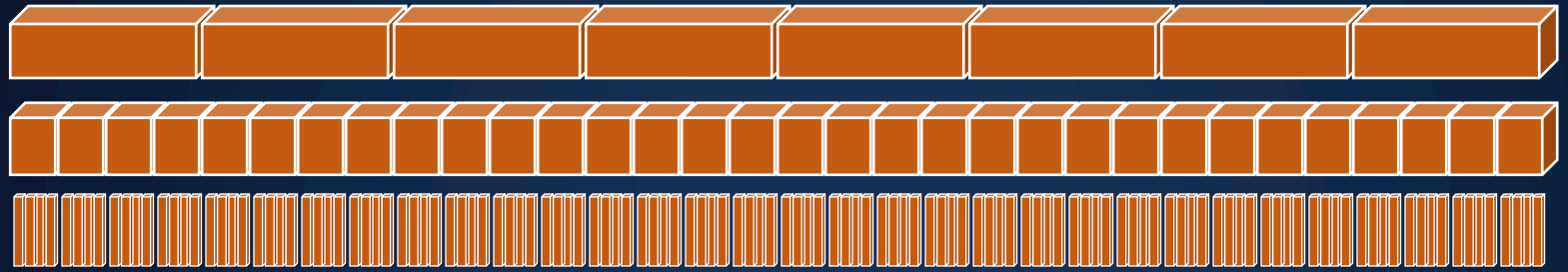
AVX-512 SIMD on KNL (Cori)

- AVX-512 data types: anything that fits into 64 bytes, e.g.



AVX-512 SIMD on KNL (Cori)

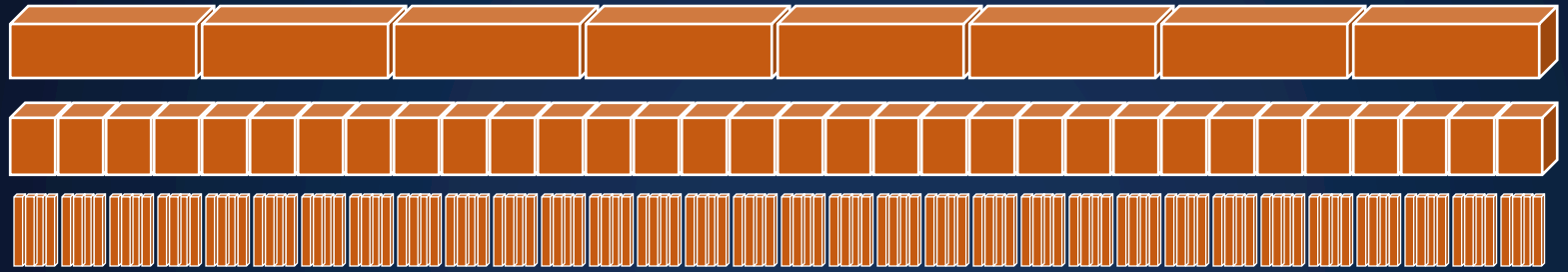
- AVX-512 data types: anything that fits into 64 bytes, e.g.



- Instructions perform add, multiply etc. on all values in parallel

AVX-512 SIMD on KNL (Cori)

- AVX-512 data types: anything that fits into 64 bytes, e.g.



- Instructions perform add, multiply etc. on all values in parallel

- Need to:
  - Expose parallelism to compiler (or write manually)
  - Be contiguous in memory and cache aligned (in most cases)

Data Dependencies Limit Parallelism

Parallelism can get the wrong answer if instructions execute out of order

Types of dependencies

- RAW: Read-After-Write

  X = A ; B = X;

- WAR: Write-After-Read

  A = X ; X = B;

- WAW: Write-After-Write

  X = A ; X = B;

- No problem / dependence for RAR: Read-After-Read

Data Dependencies Limit Parallelism

Parallelism can get the wrong answer if instructions execute out of order

Types of dependencies

- RAW: Read-After-Write

  X = A ; B = X;

- WAR: Write-After-Read

  A = X ; X = B;

- WAW: Write-After-Write

  X = A ; X = B;

- No problem / dependence for RAR: Read-After-Read

Hardware can't break these

Data Dependencies Limit Parallelism

Parallelism can get the wrong answer if instructions execute out of order

Types of dependencies

- RAW: Read-After-Write

  X = A ; B = X;

- WAR: Write-After-Read

  A = X ; X = B;

- WAW: Write-After-Write

  X = A ; X = B;

- No problem / dependence for RAR: Read-After-Read

Hardware can't break these

Compiler can't either

Memory Alignment and Strides

- Non-contiguous memory access hampers efficiency
- Can be done if write conflicts are avoided

Strided load	… = a[i*4]
Strided store	a[i*4] = …
Indexed (gather)	… = a[b[i]]
Indexed (scatter)	a[b[i]] = …

- Aligning on cache line boundaries key

FMA: Fused Multiply Add

- Multiply followed by add is very common on programs

  x = y + c * z

- Useful in matrix multiplication
- Fused Multiply-Add (FMA) instructions:
  - Performs multiply/add
  - Same rate as + or * alone
  - And does so with a single rounding step

  x = round(c * z+ y)

What does this mean to you?

- In theory, the compiler understands all of this
  - It will rearrange instructions to maximizes parallelism, uses FMAs and SIMD
  - While preserving dependencies

- But in practice the compiler may need your help
  - Choose a different compiler, optimization flags, etc.
  - Rearrange your code to make things more obvious
  - Use special functions (“intrinsics”) or write assembly ☹

Compilers Optimizations

- Unrolls loops (because control isn't free)
- Fuses loops (merge two together)
- Interchanges loops (reorder)
- Eliminates dead code (the branch never taken)
- Reorders instructions to improve register reuse and more
- Strength reduction (multiply by 2, into shift left)
- Strip-mine: turn one loop into nested one
- Strip-mine + interchange = tiling

Compilers Optimizations

- Unrolls loops (because control isn't free)
- Fuses loops (merge two together)
- Interchanges loops (reorder)
- Eliminates dead code (the branch never taken)
- Reorders instructions to improve register reuse and more
- Strength reduction (multiply by 2, into shift left)
- Strip-mine: turn one loop into nested one
- Strip-mine + interchange = tiling

Survey by Prof. Susan Graham and students

https://dl.acm.org/doi/pdf/10.1145/197405.197406

Take-Aways

Even “serial” processors use parallelism

- Instruction level parallelism (ILP): need to consider instruction mix
- SIMD instructions
- Special instructions like fused multiply add (FMA)

Compilers help with this, but the programmer can make their job easy or hard

- Dependencies are a big source of problems for the compiler
- They need to work conservatively

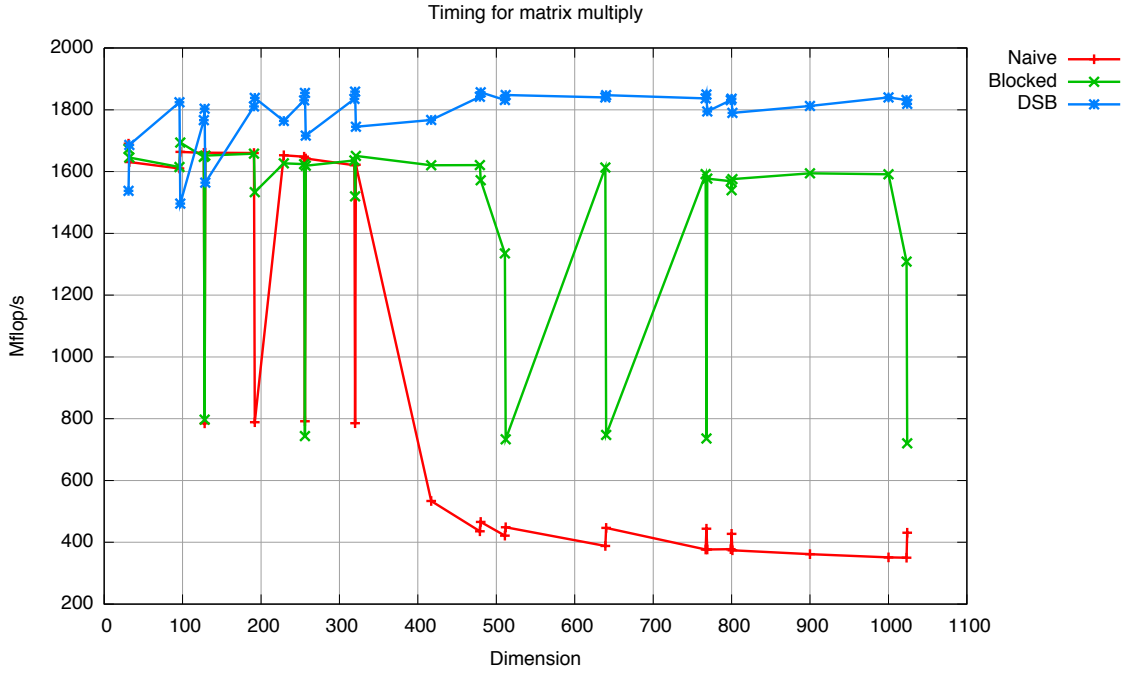
Outline

- Idealized and actual costs in modern processors
- Memory hierarchies
- Parallelism within single processors
- Case study: Matrix Multiplication
  - Use of performance models to understand performance
  - Warm-up: Matrix-vector multiplication
  - Tiled (cache-aware) matrix multiplication
  - Cache oblivious matrix multiplication
- Optimizations in practice

Why Matrix Multiplication?

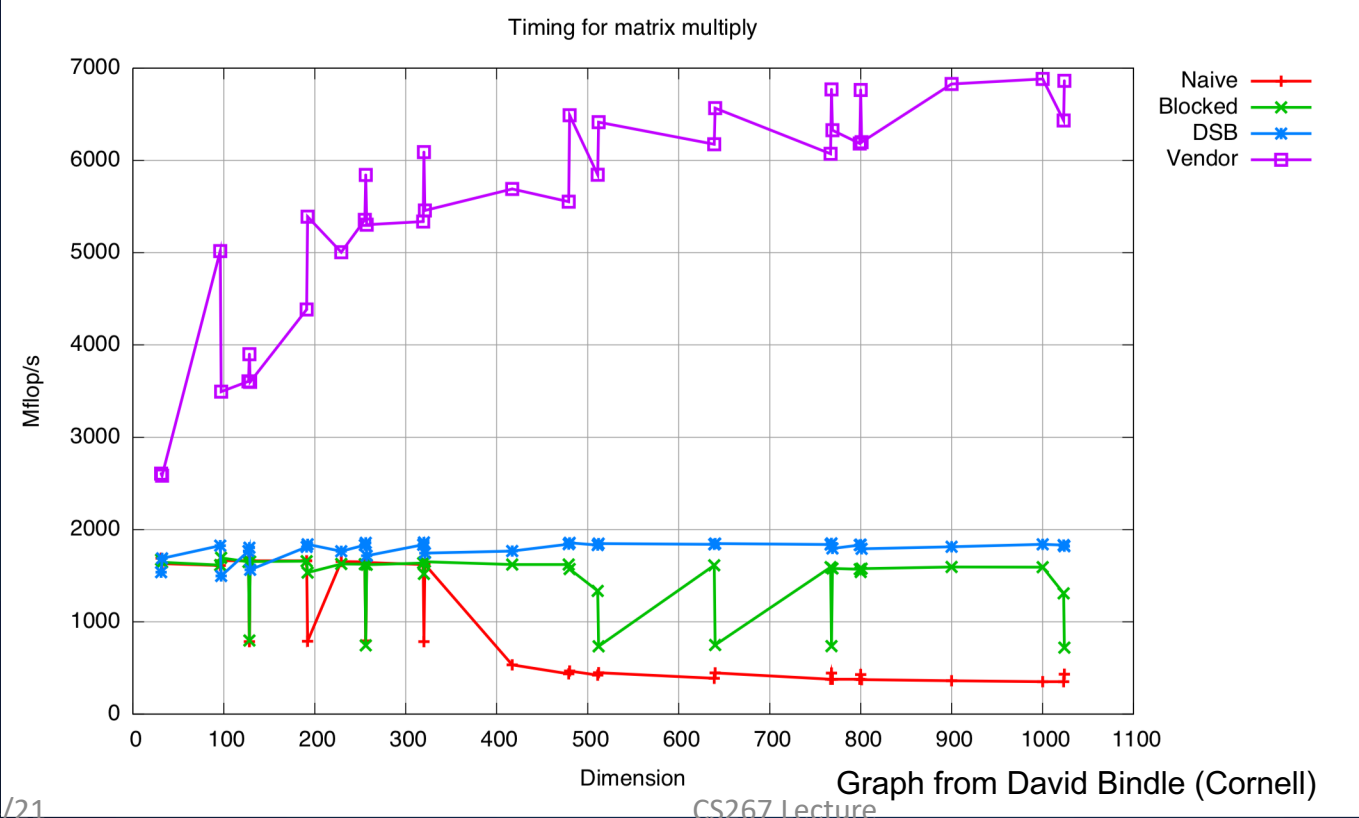
- An important kernel in many problems
  - Dense linear algebra is a motif in every list
  - Closely related to other algorithms, e.g., transitive closure on a graph
  - **And dominates training time in deep learning (CNNs)**
- Optimization ideas can be used in other problems
- The best case for optimization payoffs
- The most-studied algorithm in high performance computing

Performance from Optimization



Graph from David Bindle (Cornell)

Performance from Optimization



Graph from David Bindle (Cornell)

A Simple Model of Memory

- Assume just 2 levels in the hierarchy, fast and slow
- All data initially in slow memory
  - m = number of memory elements (words) moved between fast and slow memory
  - t_m = time per slow memory operation (inverse bandwidth in best case)
  - f = number of arithmetic operations
  - t_f = time per arithmetic operation << t_m

A Simple Model of Memory

- Assume just 2 levels in the hierarchy, fast and slow
- All data initially in slow memory
  - m = number of memory elements (words) moved between fast and slow memory
  - t_m = time per slow memory operation (inverse bandwidth in best case)
  - f = number of arithmetic operations
  - t_f = time per arithmetic operation << t_m
  - $CI = f / m$ average number of flops per slow memory access

A Simple Model of Memory

- Assume just 2 levels in the hierarchy, fast and slow
- All data initially in slow memory
  - m = number of memory elements (words) moved between fast and slow memory
  - t_m = time per slow memory operation (inverse bandwidth in best case)
  - f = number of arithmetic operations
  - t_f = time per arithmetic operation << t_m
  - $CI = f / m$ average number of flops per slow memory access
- Minimum possible time = $f * t_f$ when all data in fast memory

A Simple Model of Memory

- Assume just 2 levels in the hierarchy, fast and slow
- All data initially in slow memory
  - m = number of memory elements (words) moved between fast and slow memory
  - t_m = time per slow memory operation (inverse bandwidth in best case)
  - f = number of arithmetic operations
  - t_f = time per arithmetic operation << t_m
  - $CI = f / m$ 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 $f * t_f$

A Simple Model of Memory

- Assume just 2 levels in the hierarchy, fast and slow
- All data initially in slow memory
  - m = number of memory elements (words) moved between fast and slow memory
  - t_m = time per slow memory operation (inverse bandwidth in best case)
  - f = number of arithmetic operations
  - t_f = time per arithmetic operation << t_m
  - $CI = f / m$ 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 $f * t_f$

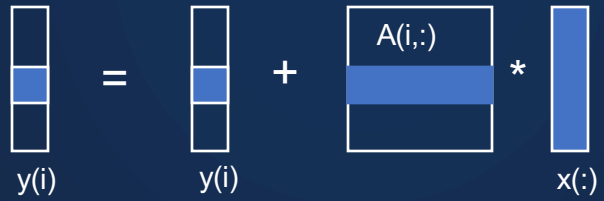
Computational Intensity (CI): **Key to algorithm efficiency**

Machine Balance: **Key to machine efficiency**

Warm up: Matrix-vector multiplication

{implements y = y + A*x}

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



Warm up: Matrix-vector multiplication

```
{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 slow memory refs = $3n + n^2$
- f = number of arithmetic operations = $2n^2$
- q = $f / m \approx 2$ (Low Computational Intensity)

- Matrix-vector multiplication limited by slow memory speed

Note: Shown with 1-based indexing for simplicity

Simplifying Assumptions

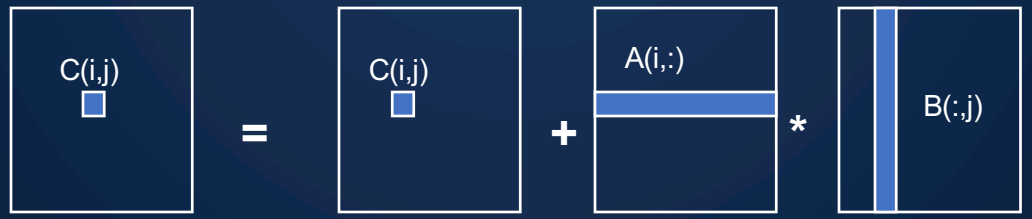
- What simplifying assumptions did we make in this analysis?
  - Constant “peak” computation rate
  - Fast memory was large enough to hold three vectors
  - The cost of a fast memory access is 0
    - OK for registers
    - Not for cache (even L1)
  - Memory latency is constant
- Could simplify further by ignoring memory operations in X and Y
  - Not bad: time to read matrix (bandwidth) may dominate

Naïve Matrix Multiply

```
{implements C = C + A*B}
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)
```

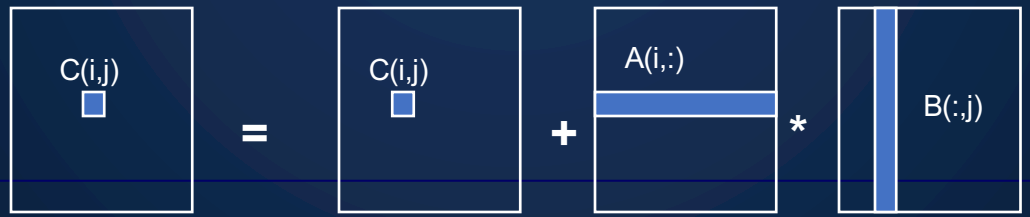
Algorithm has $2*n^3 = O(n^3)$ Flops and operates on $3*n^2$ words of memory

Computational intensity (q) *potentially* as large as $2*n^3 / 3*n^2 = O(n)$



Naïve Matrix Multiply

```
{implements C = C + A*B}
for i = 1 to n
  {read row i of A into fast memory}
   for j = 1 to n
       {read C(i,j) into fast memory}
       {read column j of B into fast memory}
       for k = 1 to n
           C(i,j) = C(i,j) + A(i,k) * B(k,j)
       {write C(i,j) back to slow memory}
```



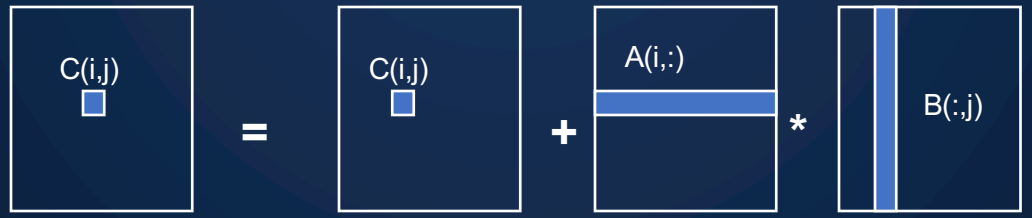
Naïve Matrix Multiply

```
{implements C = C + A*B}
for i = 1 to n
  {read row i of A into fast memory}
   for j = 1 to n
      {read C(i,j) into fast memory}
      {read column j of B into fast memory}
      for k = 1 to n
          C(i,j) = C(i,j) + A(i,k) * B(k,j)
      {write C(i,j) back to slow memory}
```

# of slow memory ops:

$m = n^3$ to read each column of B n times
$+ n^2$ to read each row of A once
$+ 2n^2$ to read and write each element of C once
$= n^3 + 3n^2$

So $q = f / m = 2n^3 / (n^3 + 3n^2)$ computational intensity
≈ 2 for large n, no improvement over matrix-vector multiply



Naïve Matrix Multiply

{implements C = C + A*B}

for i = 1 to n

f = $2n^3$ arithmetic ops. How many slow memory?

for j = 1 to n

for k = 1 to n

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

Naïve Matrix Multiply

{implements C = C + A*B}

for i = 1 to n

  {read row i of A into fast memory}

   for j = 1 to n

f = $2n^3$ arithmetic ops. How many slow memory?

n^2 to read each row of A once

```
for k = 1 to n
    C[i,j] = C[i,j] + A[i,k] * B[k,j]
```

Naïve Matrix Multiply

{implements C = C + A*B}

for i = 1 to n

  {read row i of A into fast memory}

  for j = 1 to n

f = $2n^3$ arithmetic ops. How many slow memory?

n^2 to read each row of A once

```
for k = 1 to n
    C[i,j] = C[i,j] + A[i,k] * B[k,j]
```

Naïve Matrix Multiply

{implements C = C + A*B}

for i = 1 to n

  {read row i of A into fast memory}

   for j = 1 to n

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

      for k = 1 to n

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

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

f = $2n^3$ arithmetic ops. How many slow memory?

n^2 to read each row of A once

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

Naïve Matrix Multiply

```
{implements C = C + A*B}
for i = 1 to n
  {read row i of A into fast memory}
   for j = 1 to n
      {read C[i,j] into fast memory}
      {read column j of B into fast memory}
      for k = 1 to n
          C[i,j] = C[i,j] + A[i,k] * B[k,j]
      {write C[i,j] back to slow memory}
```

$f = 2n^3$ arithmetic ops. How many slow memory?

n^2 to read each row of A once

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

n^3 to read each column of B n times

Naïve Matrix Multiply

{implements C = C + A*B}

for i = 1 to n

  {read row i of A into fast memory}

   for j = 1 to n

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

      {read column j of B into fast memory}

      for k = 1 to n

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

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

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

n^2 to read each row of A once

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

n^3 to read each column of B n times

Naïve Matrix Multiply

```
{implements C = C + A*B}
for i = 1 to n
  {read row i of A into fast memory}
   for j = 1 to n
      {read C[i,j] into fast memory}
      {read column j of B into fast memory}
      for k = 1 to n
          C[i,j] = C[i,j] + A[i,k] * B[k,j]
      {write C[i,j] back to slow memory}
```

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

n^2 to read each row of A once

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

n^3 to read each column of B n times

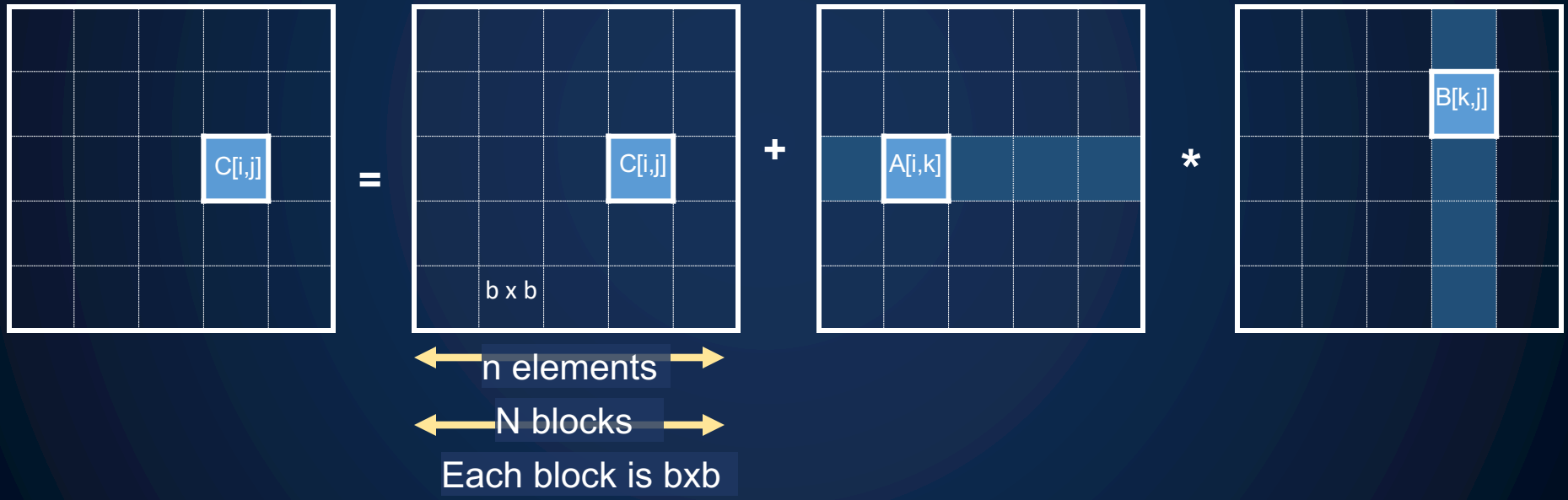
So the computational intensity is:

CI = f / m = $2n^3 / [n^3 + 3n^2]$ ~= 2

No better than matrix-vector!

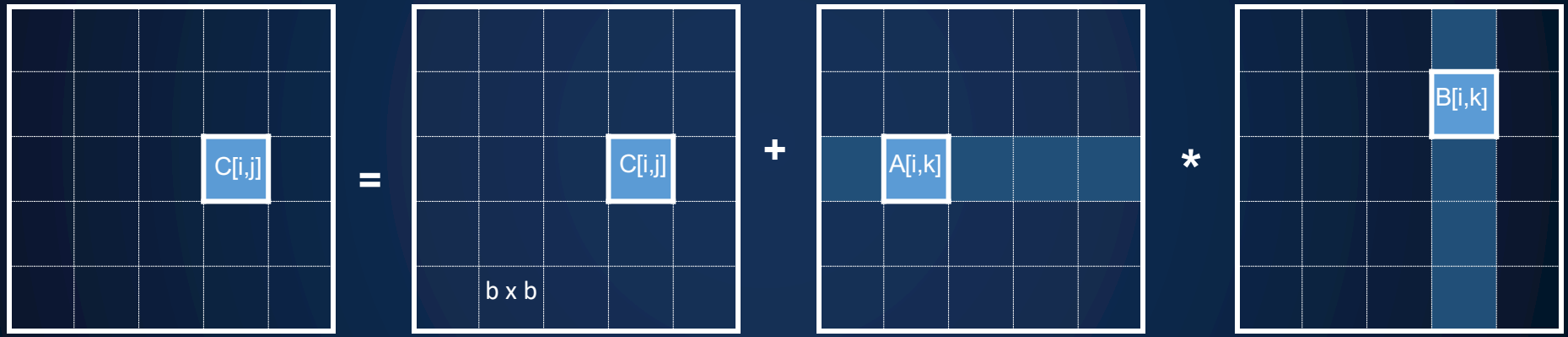
Blocked [Tiled] Matrix Multiply

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



Blocked [Tiled] Matrix Multiply

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



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

n elements

N blocks

Each block is bxb

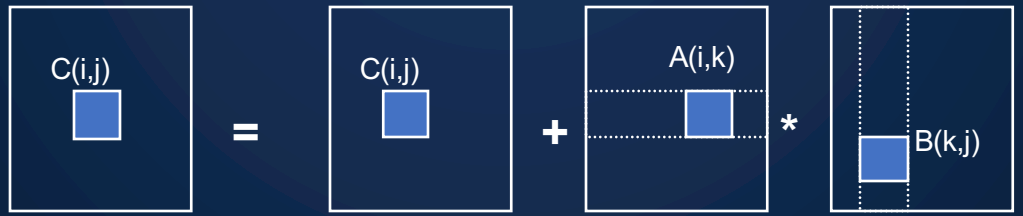
Blocked (Tiled) Matrix Multiply

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

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

3 nested loops inside

block size = loop bounds



Tiling for registers or caches

Blocked [Tiled] Matrix Multiply

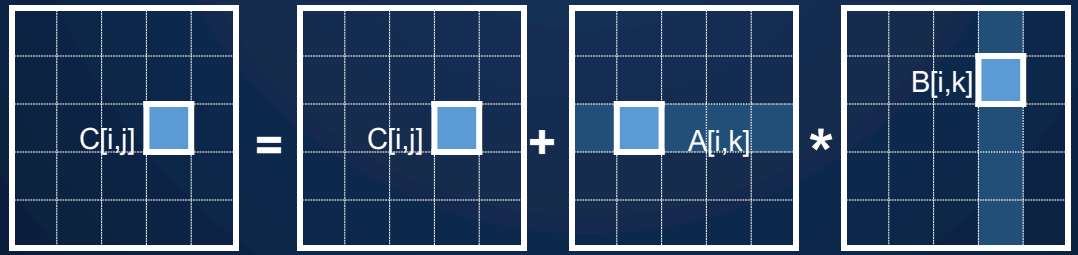
Consider A,B,C to be N-by-N matrices of b-by-b subblocks where b=n / N is called the block size

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

    for k = 1 to N
```

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

nxn elements
NxN blocks
Each block is bxb



Blocked [Tiled] Matrix Multiply

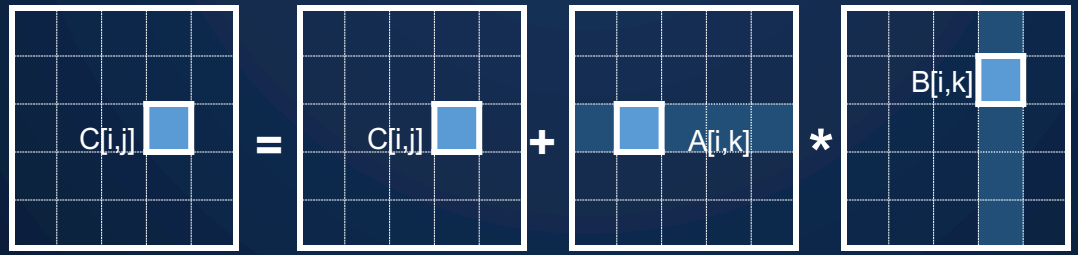
Consider A,B,C to be N-by-N matrices of b-by-b subblocks where b=n / N is called the block size

```
for i = 1 to N
  for j = 1 to N
    {read block C[i,j] into fast memory}
    for k = 1 to N
```

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

```
      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



Blocked [Tiled] Matrix Multiply

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

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

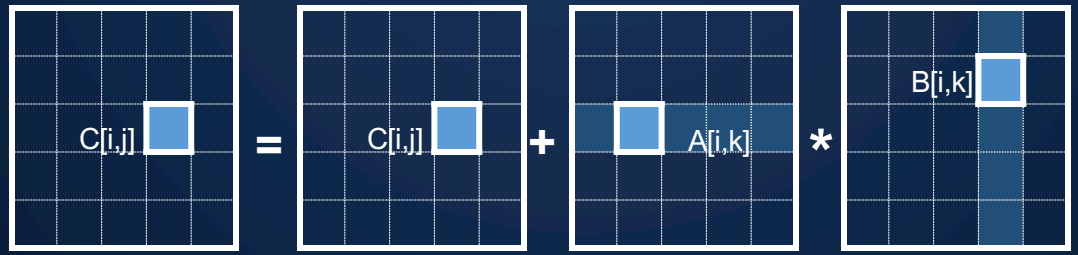
$2n^2$ to read and write each block of C once

$N*n^2$ to read each block of A N^3 times
($N^3 * b^2 = N^3 * (n/N)^2$)

nxn elements

NxN blocks

Each block is bxb



Blocked [Tiled] Matrix Multiply

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

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

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

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

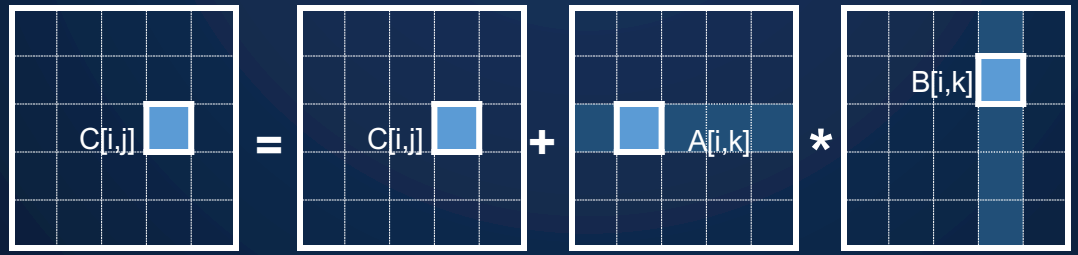
$N*n^2$ to read each block of B N^3 times

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

nxn elements

NxN blocks

Each block is bxb



Blocked [Tiled] Matrix Multiply

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

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

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

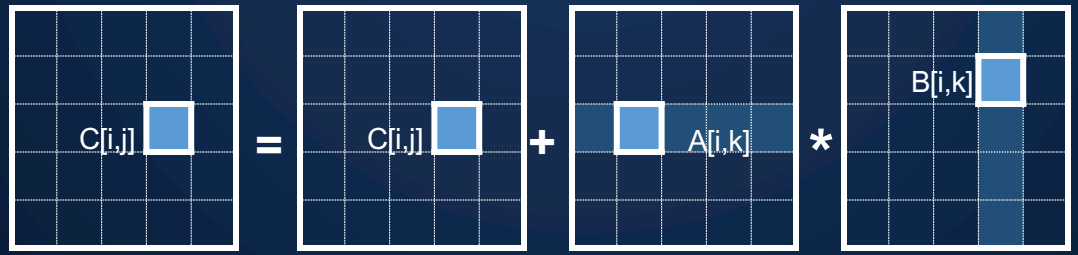
$N*n^2$ to read each block of A N^3 times

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

nxn elements

NxN blocks

Each block is bxb



Blocked [Tiled] Matrix Multiply

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

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

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

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

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

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

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

Tiling for Registers

If the block dimension b is very small (e.g., 2) then:

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

Tiling for Registers

If the block dimension b is very small (e.g., 2) then:

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

1) we can write the inner matmul without loops (unrolled)

Tiling for Registers

If the block dimension b is very small (e.g., 2) then:

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

1) we can write the inner matmul without loops (unrolled)

C[0,0] += A[0,0]·B[0,0] + A[0,1]·B[1,0]

C[0,1] += A[0,0]·B[0,1] + A[0,1]·B[1,1]

C[1,0] += A[1,0]·B[0,0] + A[1,1]·B[1,0]

C[1,1] += A[1,0]·B[0,1] + A[1,1]·B[1,1]

Tiling for Registers

If the block dimension b is very small (e.g., 2) then:

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

1) we can write the inner matmul without loops (unrolled)

2) The values may all fit in register

C[0,0] += A[0,0]·B[0,0] + A[0,1]·B[1,0]

C[0,1] += A[0,0]·B[0,1] + A[0,1]·B[1,1]

C[1,0] += A[1,0]·B[0,0] + A[1,1]·B[1,0]

C[1,1] += A[1,0]·B[0,1] + A[1,1]·B[1,1]

Tiling for Registers

If the block dimension b is very small (e.g., 2) then:

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

1) we can write the inner matmul without loops (unrolled)

2) The values may all fit in register

C[0,0] += A[0,0]·B[0,0] + A[0,1]·B[1,0]

C[0,1] += A[0,0]·B[0,1] + A[0,1]·B[1,1]

C[1,0] += A[1,0]·B[0,0] + A[1,1]·B[1,0]

C[1,1] += A[1,0]·B[0,1] + A[1,1]·B[1,1]

Register blocks need not be square

Are we getting the same answer?

Rather than something like

C[i,j] += AB + AB + AB … AB

Each AB some product of an element of A and B, indices omitted here

Are we getting the same answer?

Rather than something like

C[i,j] += AB + AB + AB … AB

Which hardware executes as

C[i,j] += (((AB + AB) + AB) … AB)

Each AB some product of an element of A and B, indices omitted here

Are we getting the same answer?

Rather than something like

C[i,j] += AB + AB + AB … AB

Which hardware executes as

C[i,j] += (((AB + AB) + AB) … AB)

The tiled version does something like

C[i,j] += AB + AB; … C[i,j] += AB + AB

Each AB some product of an element of A and B, indices omitted here

Are we getting the same answer?

Rather than something like

C[i,j] += AB + AB + AB … AB

Which hardware executes as

C[i,j] += (((AB + AB) + AB) … AB)

Each AB some product of an element of A and B, indices omitted here

The tiled version does something like

C[i,j] += AB + AB; … C[i,j] += AB + AB

Which has more stores to memory and is more like

- C[i,j] += ((AB + AB) + (AB + AB))

Are we getting the same answer?

Rather than something like

C[i,j] += AB + AB + AB … AB

Which hardware executes as

C[i,j] += (((AB + AB) + AB) … AB)

Assumes + is associative:
(x + y) + z = x + (y + z)
And extra stores don't matter

Isn't strictly true for floating point, but close enough for matrix multiply!

The tiled version does something like

C[i,j] += AB + AB; … C[i,j] += AB + AB

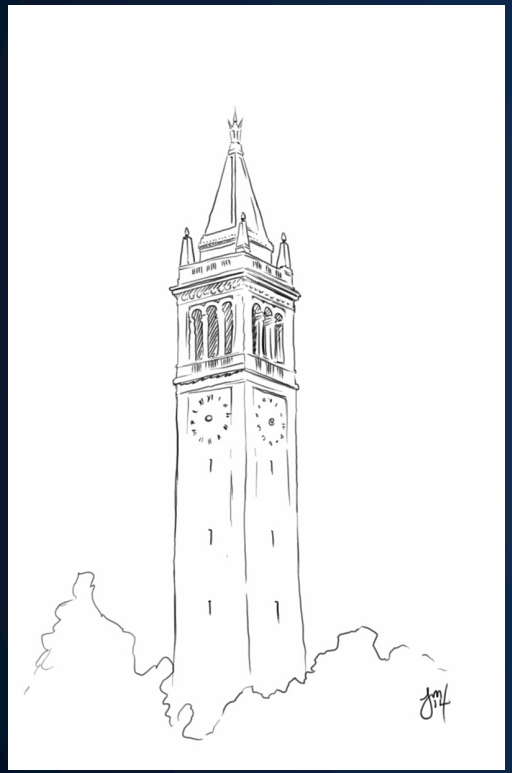
Which has more stores to memory and is more like

- C[i,j] += ((AB + AB) + (AB + AB))

Tuning Code in Practice

- Tuning code can be tedious
  - Many optimizations besides blocking
  - Behavior of machine and compiler hard to predict
- Approach #1: Analytical performance models
  - Use model (of cache, memory costs, etc.) to select best code
  - But model needs to be both simple and accurate ☹
- Approach #2: "Autotuning"
  - Let computer generate large set of possible code variations
  - Search for the fastest ones (may be matrix size dependent too!)
  - Sometimes all done "off-line," sometimes at run-time

The Birth of Autotuning at Cal



- ‘90-92: 4 new professors arrive
- Buy a mini supercomputer
- Taught parallel computing

The PHiPAC (Portable High Performance ANSI C) Page
for
BLAS3 Compatible Fast Matrix Matrix Multiply

Jeff A Bilmes

Krste Asanovic

Rich Vuduc

Sriram Iyer

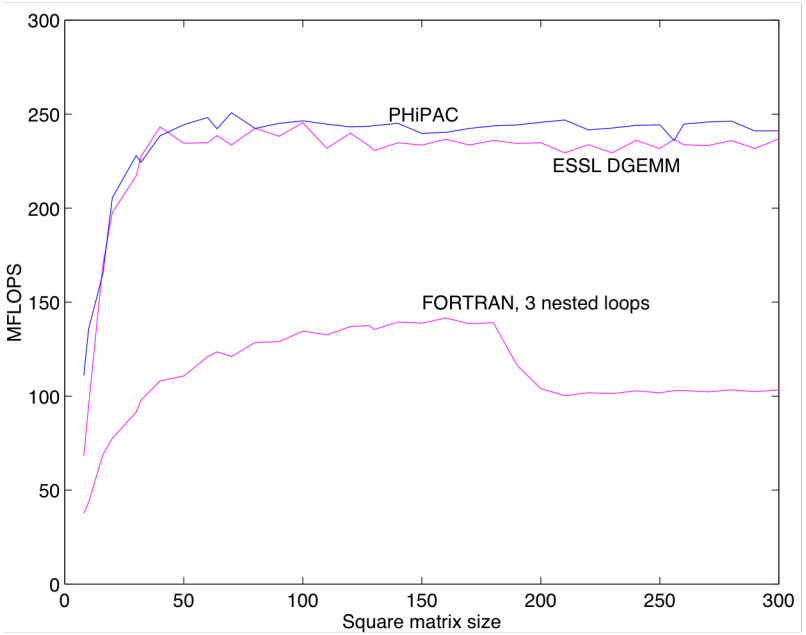
Jim Demmel

CheeWhye Chin

Dominic Lam

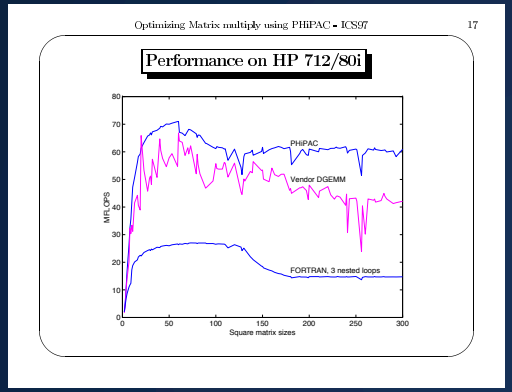
Portable automatic generation of fast BLAS-GEMM compatible matrix-matrix multiply using PHiPAC techniques.

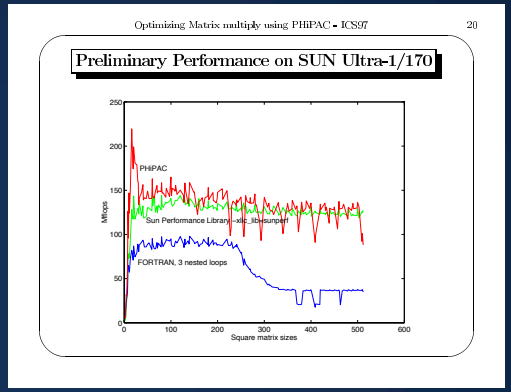
Beat a vendor implementation

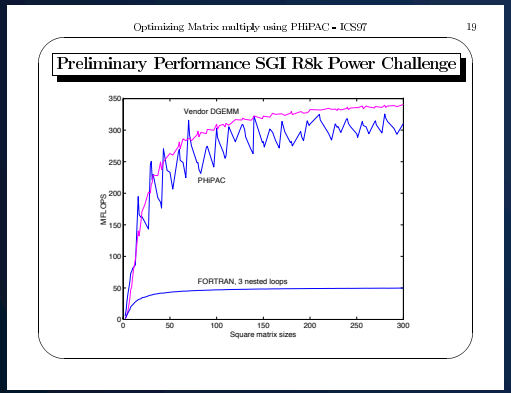


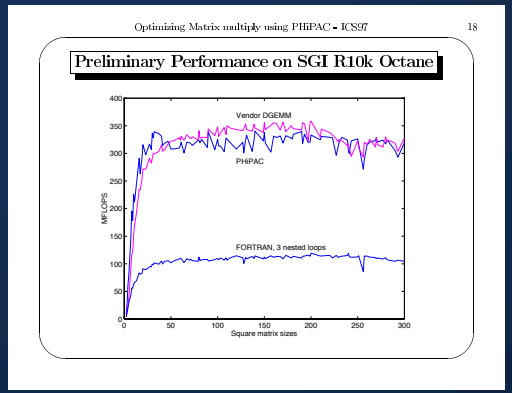
Vendor code likely hand-tuned assembly

Beat vendors…on multiple machines

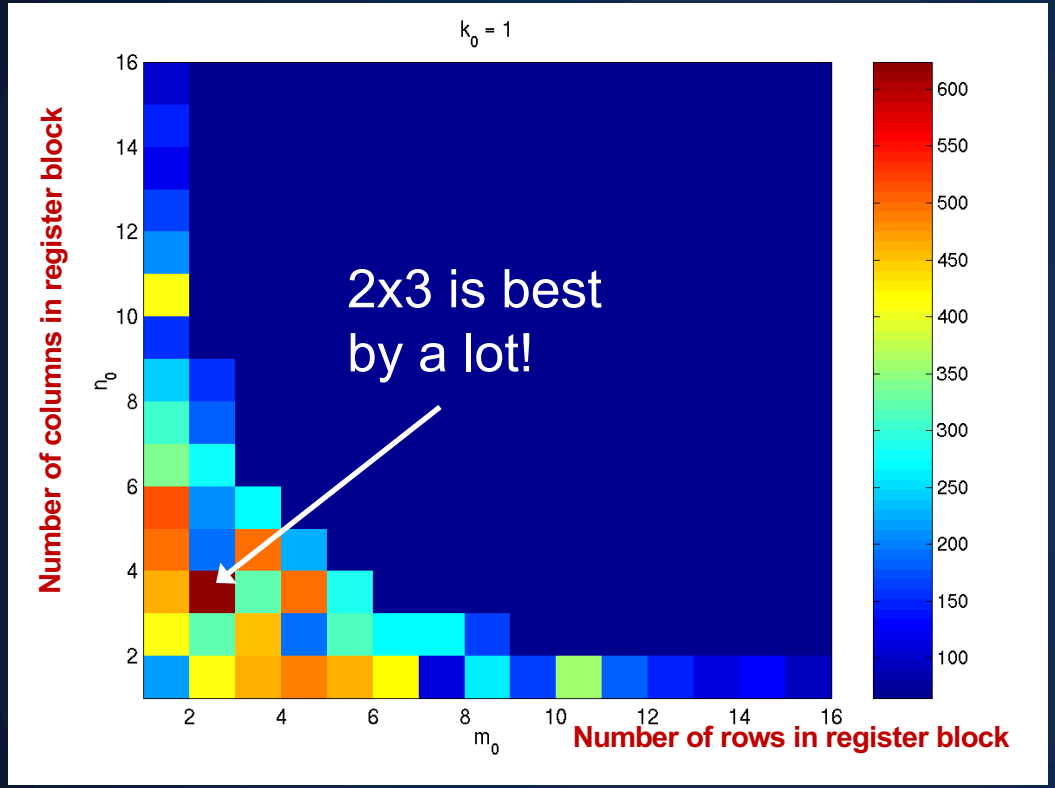








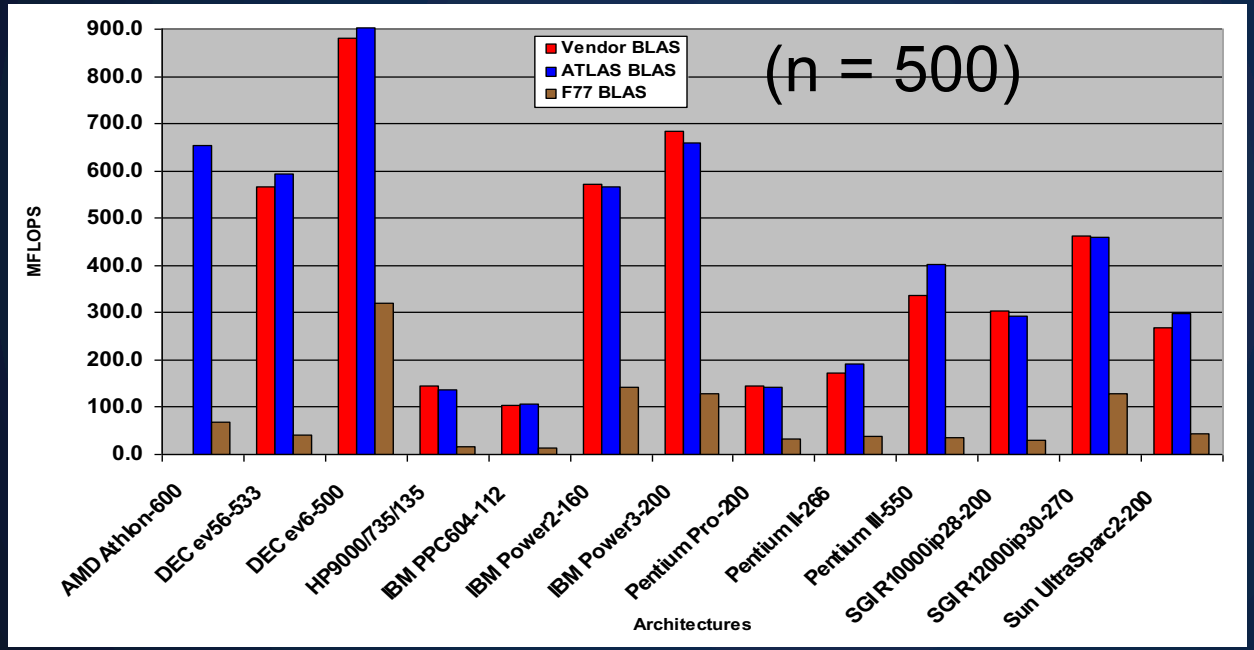
What the Search Space Looks



A 2-D slice of a 3-D register-tile search space. The dark blue region was pruned.
(Platform: Sun Ultra-IIi, 333 MHz, 667 Mflop/s peak, Sun cc v5.0 compiler)

ATLAS from UTK Became Standard

- ATLAS is faster than all other portable BLAS implementations and it is comparable with machine-specific libraries provided by the vendor.



Source: Jack Dongarra

Take-Aways

- Matrix vector and matrix matrix multiplication key (dense LA motif)
- Matrix vector multiplication
  - Opportunities for better parallelism and use FMA, etc.
  - Limited by bandwidth ($O(2n^2)$ flops on $O(n^2)$ data)
- Matrix matrix multiplication
  - Can improve computational intensity $O(2n^3)$ flops on $O(3n^2)$ data
  - Tiling (aka blocking)
- Optimizations can be tricky
  - Use of performance models to predict the best tile size
  - Or autotuning search to find the best (in a range)
- Autotuning is especially useful in dense matrix operations
  - Performance depends on the size of the matrices, not the values in them
  - But it will come up elsewhere too!

Optimizing in Practice

- Tiling for registers
  - loop unrolling, use of named "register" variables
- Tiling for multiple levels of cache and TLB
- Exploiting fine-grained parallelism in processor
  - superscalar; pipelining
- Complicated compiler interactions (flags)
- Hard to do by hand (but you'll try)
- Automatic optimization an active research area
  - ASPIRE: `aspire.eecs.berkeley.edu`
  - BeBOP: `bebop.cs.berkeley.edu`
    - Weekly group meeting Mondays 1pm
  - PHiPAC: `www.icsi.berkeley.edu/~bilmes/phipac`

    in particular `tr-98-035.ps.gz`
  - ATLAS: `www.netlib.org/atlas`

Note on Matrix Storage

- A matrix is a 2-D array of elements, but memory addresses are "1-D"
- Conventions for matrix layout
  - by column, or "column major" (Fortran default); A(i,j) at A+i+j*n
  - by row, or "row major" (C default) A(i,j) at A+i*n+j
  - recursive (later)

Column major

0	5	10	15
1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19

Row major

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Green row of matrix is stored in red cache lines

Column major matrix in memory

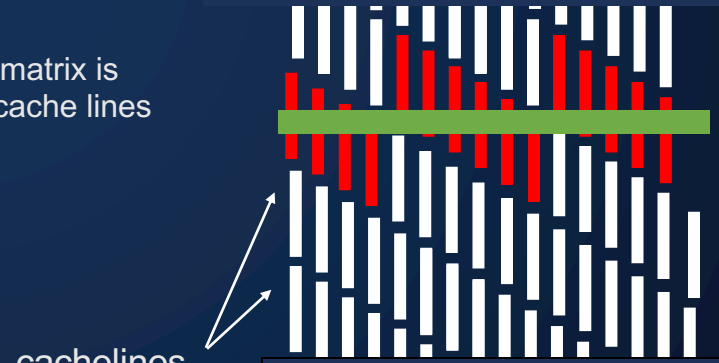


Figure source: Larry Carter, UCSD

Copy optimization

- Copy input operands or blocks
  - Pad a matrix to reduce cache conflicts
  - Constant array offsets for fixed size blocks (A[i*n + j] make n constant)
  - Align blocks on cache lines for vectorization

Original matrix row-major
(numbers are addresses)

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Reorganized into
2x2 blocks

0	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

Loop Unrolling

- Expose instruction-level parallelism and reduce control overhead

```
float f0 = filter[0], f1 = filter[1], f2 = filter[2];
float s0 = signal[0], s1 = signal[1], s2 = signal[2];
*res++ = f0*s0 + f1*s1 + f2*s2;
do {
   signal += 3;
   s0 = signal[0];
   res[0] = f0*s1 + f1*s2 + f2*s0;

   s1 = signal[1];
   res[1] = f0*s2 + f1*s0 + f2*s1;

   s2 = signal[2];
   res[2] = f0*s0 + f1*s1 + f2*s2;

   res += 3;
} while( … );
```

Removing False Dependencies

- Using local variables, reorder operations to remove false dependencies

```
a[i] = b[i] + c;
a[i+1] = b[i+1] * d;
```

false read-after-write hazard between a[i] and b[i+1]

```
float f1 = b[i];
float f2 = b[i+1];

a[i] = f1 + c;
a[i+1] = f2 * d;
```

With some compilers, you can declare a and b unaliased.

- Done via "restrict pointers," compiler flag, or pragma
- In Fortran, can use function calls (arguments assumed unaliased, maybe).

Exploit Multiple Registers

- Reduce memory bandwidth by pre-loading into local variables

```
while( … ) {
   *res++ = filter[0]*signal[0]
            + filter[1]*signal[1]
            + filter[2]*signal[2];
   signal++;
}
```

Example is a convolution

Exploit Multiple Registers

- Reduce memory bandwidth by pre-loading into local variables

```
while( … ) {
   *res++ = filter[0]*signal[0]
            + filter[1]*signal[1]
            + filter[2]*signal[2];
   signal++;
}
```

Example is a convolution

Exploit Multiple Registers

- Reduce memory bandwidth by pre-loading into local variables

```
while( … ) {
   *res++ = filter[0]*signal[0]
          + filter[1]*signal[1]
          + filter[2]*signal[2];
   signal++;
}
```

Example is a convolution

also: `register float f0 = …;`

```
float f0 = filter[0];
float f1 = filter[1];
float f2 = filter[2];
while( … ) {
   *res++ = f0*signal[0]
          + f1*signal[1]
          + f2*signal[2];
   signal++;
}
```

Expose Independent Operations

- Hide instruction latency
  - Use local variables to expose independent operations that can execute in parallel or in a pipelined fashion
  - Balance the instruction mix (what functional units are available?)

```
f1 = f5 * f9;
f2 = f6 + f10;
f3 = f7 * f11;
f4 = f8 + f12;
```

Minimize Pointer Updates

- Replace pointer updates for strided memory addressing with constant array offsets

```
f0 = *r8; r8 += 4;
f1 = *r8; r8 += 4;
f2 = *r8; r8 += 4;
```

```
f0 = r8[0];
f1 = r8[4];
f2 = r8[8];
r8 += 12;
```

- Pointer vs. array expression costs may differ.
  Some compilers do a better job at analyzing one than the other

Summary

- Performance programming on uniprocessors requires
  - understanding of memory system
  - understanding of fine-grained parallelism in processor
- Simple performance models can aid in understanding
  - Two ratios are key to efficiency (relative to peak)
  1. computational intensity of the algorithm:
     - q = f/m = # floating point operations / # slow memory references
  2. machine balance in the memory system:
     - t_m/t_f = time for slow memory reference / time for floating point operation
- Want q > t_m/t_f to get half machine peak
- Blocking (tiling) is a basic approach to increase q
  - Techniques apply generally, but the details (e.g., block size) are architecture dependent
  - Similar techniques are possible on other data structures and algorithms
- Now it's your turn: Homework 1