Outline

- Processors and registers
- Memory hierarchies
  - Temporal and spatial locality
  - Cache timing
  - Use of microbenchmarks to characterize performance
- Case study: Matrix multiplication
- Optimizations in Practice

Idealized Uniprocessor Model

- Processor names variables:
  - Integers, floats, doubles, pointers, arrays, structures, … are really bytes, words, etc. in 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 Manage Memory and Registers

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

a = c + d

e = a + b

f = e - 1

Assume a and e not used again (dead), b,c,d, and f are live

a R1

b R4

f R1

Register allocation in first Fortran compiler in 1950s, graph coloring in 1980.

Load c into R2

Load d into R3

Load b into R4

R1 = R2 + R3

R1 = R1 + R4

R1 = R1 – 1

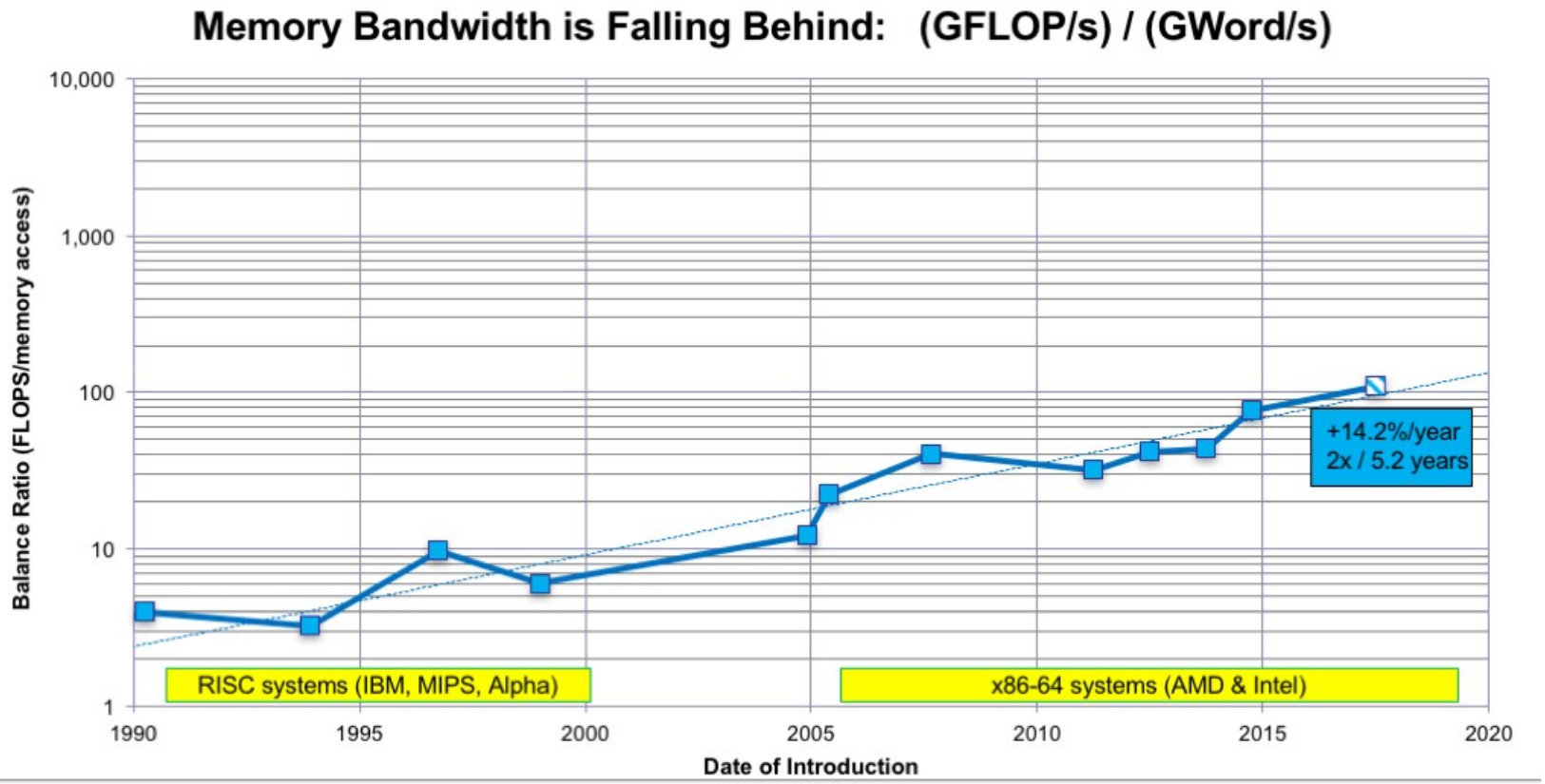
Optimizing Compilers

- Besides register allocation, the compiler performs optimizations:
  - Unrolls loops (because control isn't free)
  - Fuses loops (merge two together)
  - Interchanges iteration variables (reorder, can lead to vectorization)
  - 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 should you care?
  - Because sometimes it does the best thing possible
  - But other times it does not…

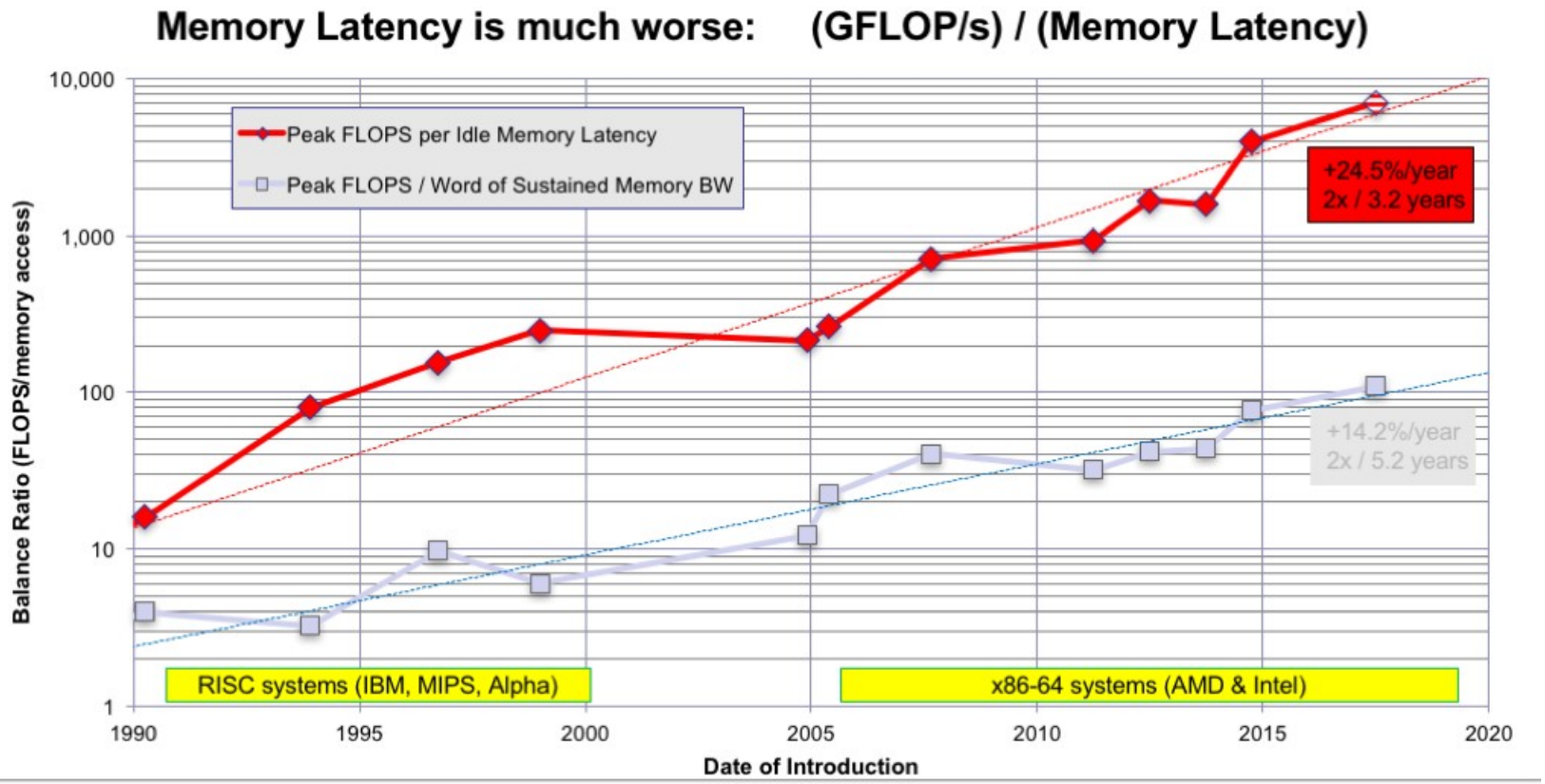
More Realistic Uniprocessor Model

- Memory accesses have *two* costs:
  - Latency (e.g. 10s to 100 ns)
  - Bandwidth: Average rate in bytes/second (or, inverse, seconds/byte) : (2-40 GB/s $\rightarrow$ 0.025 to 0.5 ns/byte)
- Bandwidth can't keep up with processor demands
- Latency is worse: DRAM isn't getting faster

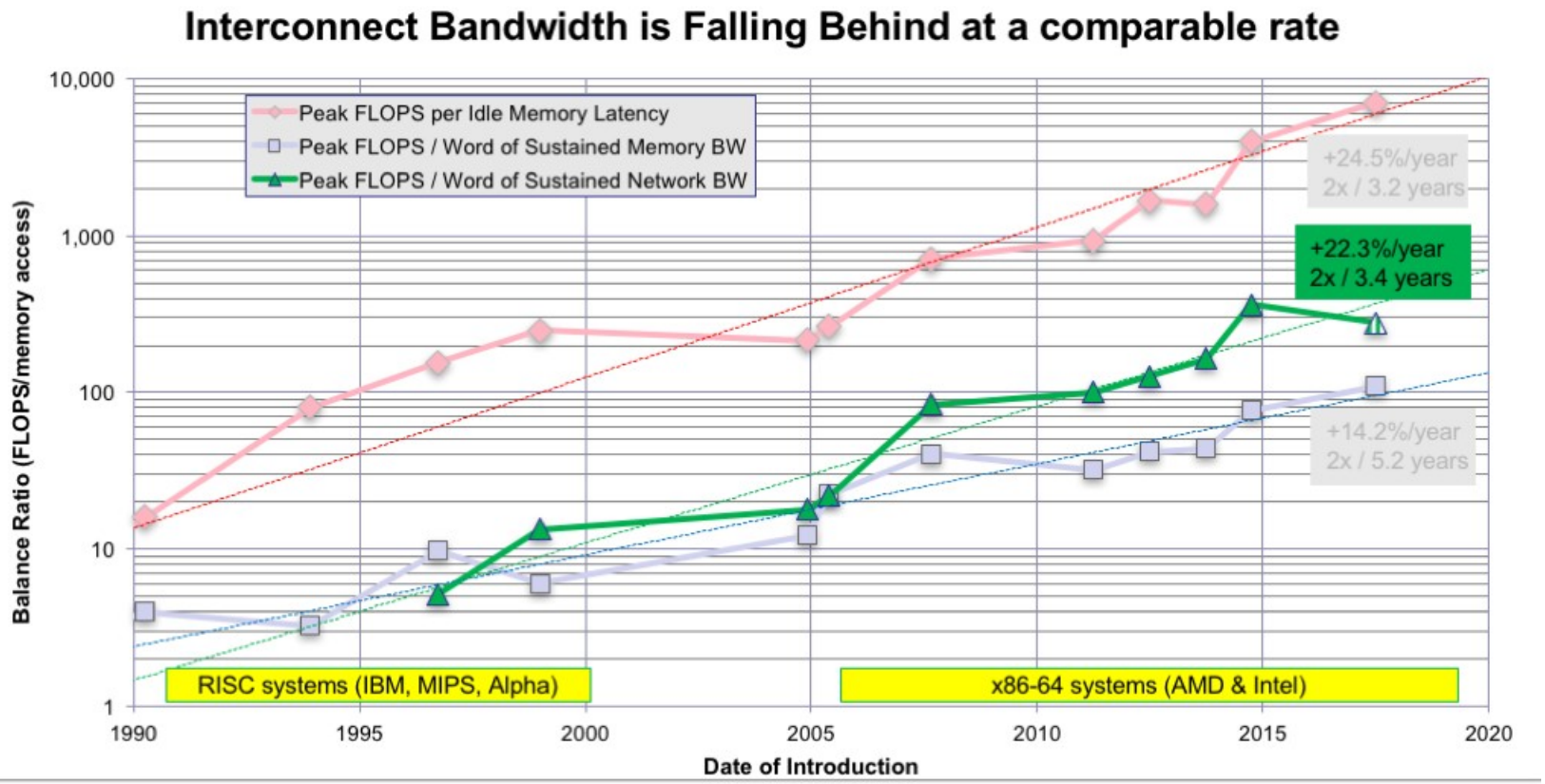
Memory Bandwidth is falling behind



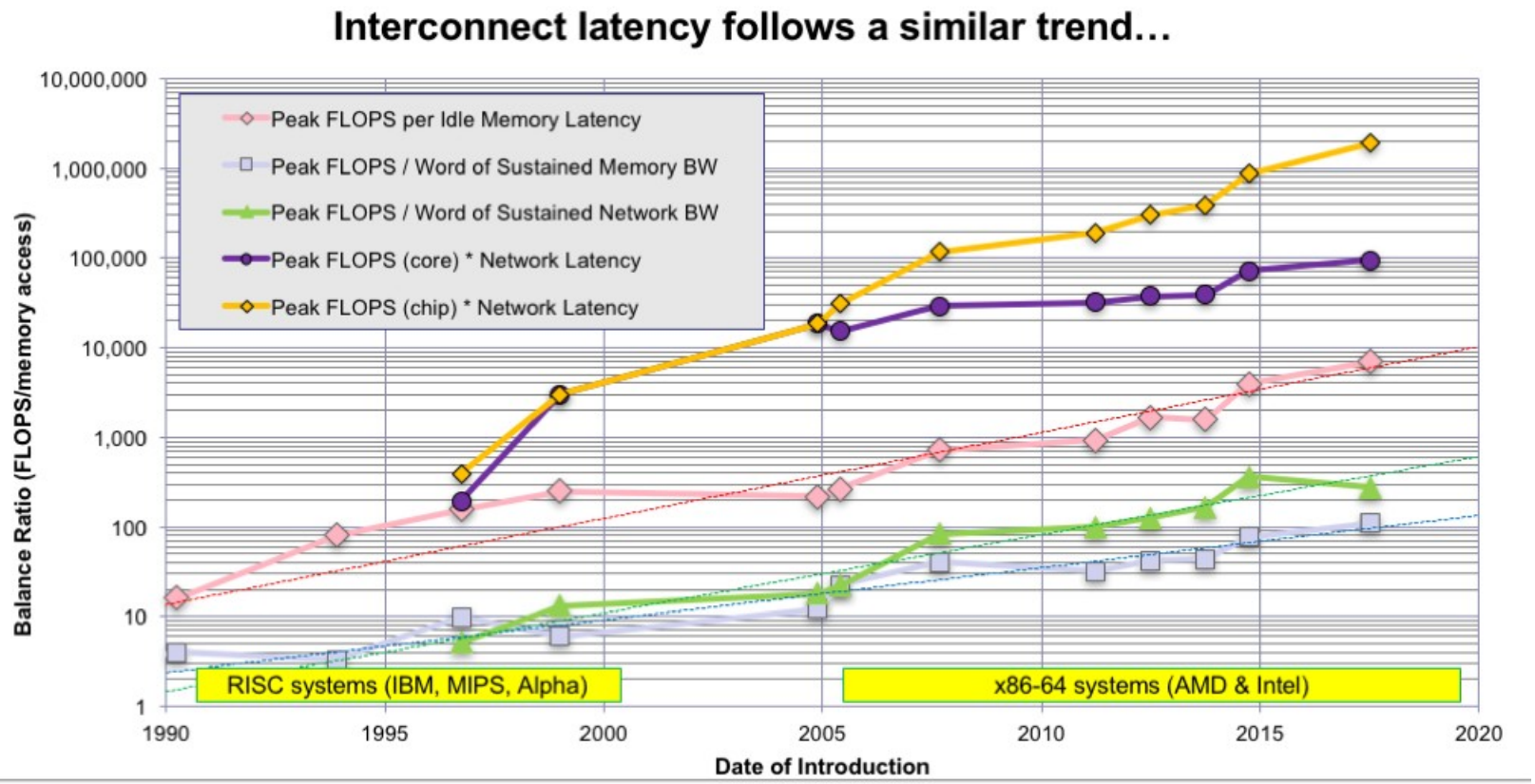
Memory Latency is worse



Interconnect Bandwidth is also falling behind



Interconnect Latency follows...



Why is memory bandwidth slowly increasing?

- Slow rate of pin speed increase
  - RAMBUS™ is an exception
  - Emphasis on capacity, not speed
  - Multiplexed DIMMs
- 6-T SRAM cell time unchanged
- Pins cost money (and manufacturers are cheap)

Why is memory latency slowly increasing?

- More levels in cache hierarchy
- Many different clock domains (Intel example)
  - Local Memory: “Core” → Ring → DDR → Ring → ”Core”
  - Snoopy: (n.b. QPI = QuickPath Interconnect)
    “Core” → Ring → QPI → Ring → QPI → Ring → ”Core”
- More cores to keep coherent (and coherency takes time)

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
- Caches may make latency (worst case) even worse
  - Robot loading tapes at supercomputer center
- Latest change: “nonvolative memory” (NVM), like Flash
  - Flash: Storage without power; Reading and writing need power. But writing can take 2x – 10x longer than reading.

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 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 bandwidth)
  - 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)

Latency, Bandwidth and Concurrency

- *Little's Law* from queuing theory says:

  concurrency = latency * bandwidth

- For example:

  latency = 10 sec

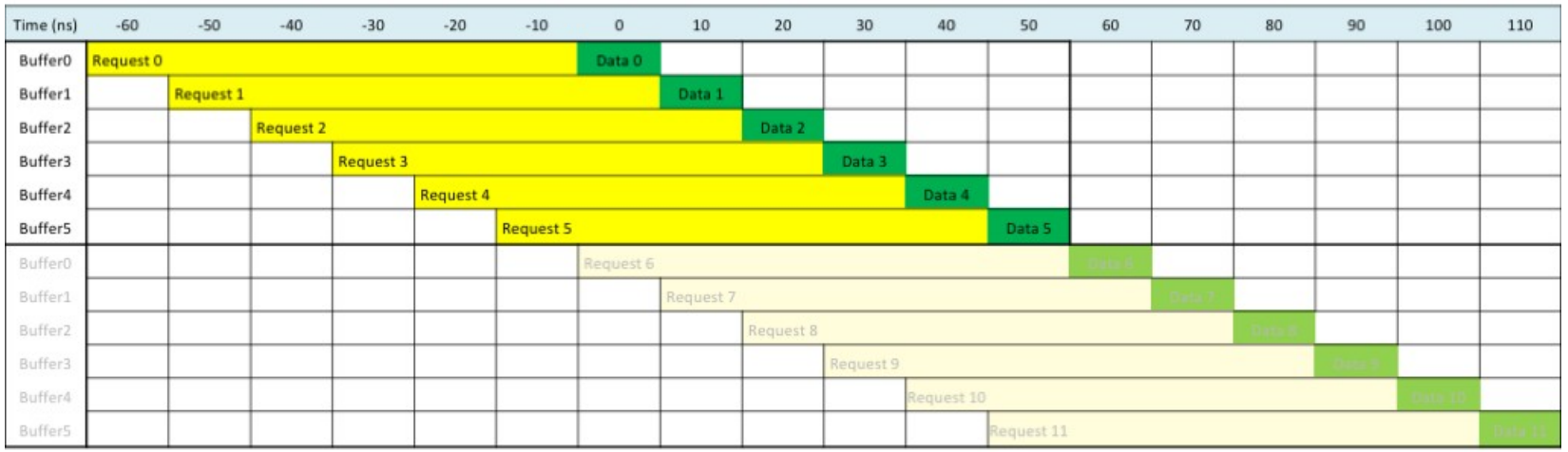
  bandwidth = 2 Bytes/sec

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

Real 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

Memory Benchmark (CacheBench)

```
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
```

Why Matrix Multiplication?

- An important kernel in many problems
  - Dense linear algebra is extremely common
  - 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

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
  - Computational Intensity (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

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)
```

Matrix-Vector Multiplication (with memory)

```
// implements y = y + A*x
// 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
```

Matrix-Vector Multiplication (complexity)

- Assume for simplicity vectors x and y, and one row of A, all fit in fast memory
- m = number of slow memory refs = $3n + n^2$
- f = number of arithmetic operations = $2n^2$
- q = f / m ≈ 2 (Low Computational Intensity)
- Matrix-vector multiplication limited by slow memory speed

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 on x & 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
    for j = 1 to n
        for k = 1 to n
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

# 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

Blocked (Tiled) Matrix Multiply

```
// implements C = C + A*B
// A,B,C to be N-by-N matrices of b-by-b subblocks
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            // Matrix multiply over blocks of size b = n / N
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

if n % b ~= 0 you need to have code for the “edges/fringes”

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

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

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

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

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

2) The values may all fit in registers

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]

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

Which has more stores to memory and is more like

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!

Recursive Matrix Multiplication

Define C = RMM (A, B, n)

if (n=1) {

$C_{00} = A_{00} * B_{00}$;

} else {

$C_{00} = RMM (A_{00} , B_{00} , n/2) + RMM (A_{01} , B_{10} , n/2)$

$C_{01} = RMM (A_{00} , B_{01} , n/2) + RMM (A_{01} , B_{11} , n/2)$

$C_{10} = RMM (A_{10} , B_{00} , n/2) + RMM (A_{11} , B_{10} , n/2)$

$C_{11} = RMM (A_{11} , B_{01} , n/2) + RMM (A_{11} , B_{11} , n/2)$

}

return C

Experience with Cache-Oblivious Algorithms

- Recursive Matrix Multiply is cache oblivious
- In practice, need to cut off recursion well before 1x1 blocks
- Call “micro-kernel” on small blocks
- Careful attention to micro-kernel is needed
- Pingali et al report that they never got more than 2/3 of peak.
- Issues with Cache Oblivious (recursive) approach

Basic Linear Algebra Subroutines (BLAS)

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

Take-Aways

- Matrix vector and matrix matrix multiplication key (linear algebra)
- Matrix vector multiplication
  - Opportunities for better parallelism and use special instructions, Fused Multiply Add, etc.
  - Limited by bandwidth ($O(2n^2)$ flops on $O(n^2)$ data)
- Matrix matrix multiplication
  - Can improve computational intensity O(2n3) flops on O(3n2) data
  - Tiling (aka blocking)
- Optimizations in practice
  - Expose parallelism to compiler (SIMD, ILP, prefetching)
  - Help compiler with register use
  - Help hardware (data layout) with cache locality (temporal and spatial)
- Optimized libraries (BLAS) exist

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

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)
- Intense interaction with cache line size

Tips on Tuning

"We should forget bout small efficiences, say about 97% of the time: premature optimization is the root of all evil."– C.A.R. Hoare (quoted by Donald Knuth)

- Tradeoff: speed vs readability, debuggability, maintainability...
- Only optimize when needful
- Go for low-hanging fruit first: data layouts, libraries, compiler flags
- Concentrate on the bottleneck
- Concentrate on inner loops
- Get correctness (and a test framework) first

Tip #1: Tools & Libraries

- We have gcc. The Intel compilers are better.
- Fortran compilers often do better than C compilers (less pointer aliasing)
- Intel VTune, cachegrind, and Shark can provide useful profiling information (including information about cache misses)
- Libraries build on someone else's hard work

Tip #2: Compiler flags

- -O3: Aggressive optimization
- -march=core2: Tune for specific architecture
- -ftree-vectorize: Automatic use of SSE (supposedly)
- -funroll-loops: Loop unrolling
- -ffast-math: Unsafe floating point optimizations

Tip #3: Memory layout

- Arrange data for unit stride access
- Arrange algorithm for unit stride access
- Tile for multiple levels of cache
- Tile for registers (loop unrolling + “register” variables)

Tip #4: Use small data structures

- Smaller data types are faster
- Bit arrays vs int arrays for flags?
- Minimize indexing data — store data in blocks
- Some advantages to mixed precision calculation (float for large data structure, double for local calculation)
- Sometimes recomputing is faster than saving

Tip #5: Inline judiciously

- Function call overhead is often minor...
- ... but structure matters to optimizer!
- C++ has inline keyword to indicate inlined functions