Memory Hierarchies and Optimizations

Kathy Yelick
yelick@cs.berkeley.edu
www.cs.berkeley.edu/~yelick/cs194f07
Motivation

- Multiple cores or processors on a single system are there for performance
- Many applications run well below the “peak” of the systems, often under 10% of arithmetic performance
- Perhaps optimizing the code on a single core will give as much benefit as writing in parallel
  - Most of the single processor performance loss is in the memory system
  - To understand this, we need to look under the hood of modern processors

- A parallel programmer is also a *performance* programmer
  - Understand your hardware
Outline

• Idealized and actual costs in modern processors
• Parallelism within single processors
• Memory hierarchies
  • Use of microbenchmarks to characterized performance
• Case study: Matrix Multiplication
  • Use of performance models to understand performance
Outline

• Idealized and actual costs in modern processors
• Parallelism within single processors
• Memory hierarchies
  • Use of microbenchmarks to characterized performance
• Case study: Matrix Multiplication
  • Use of performance models to understand performance
Idealized Uniprocessor Model

- Processor names bytes, words, etc. in its address space
  - These represent integers, floats, pointers, arrays, etc.
- Operations include
  - Read and write into very fast memory called registers
  - Arithmetic and other logical operations on registers
- Order specified by program
  - Read returns the most recently written data
  - Compiler and architecture translate high level expressions into “obvious” lower level instructions

\[
A = B + C 
\Rightarrow
\]

  - Read address(B) to R1
  - Read address(C) to R2
  - \( R3 = R1 + R2 \)
  - Write R3 to Address(A)

- Hardware executes instructions in order specified by compiler
- **Idealized Cost**
  - Each operation has roughly the same cost
    - (read, write, add, multiply, etc.)
Uniprocessors in the Real World

• Real processors have
  • registers and caches
    • small amounts of fast memory
    • store values of recently used or nearby data
    • different memory ops can have very different costs
  • parallelism
    • multiple “functional units” that can run in parallel
    • different orders, instruction mixes have different costs
  • pipelining
    • a form of parallelism, like an assembly line in a factory

• Why is this your problem?
  • In theory, compilers understand all of this and can optimize your program; in practice they don’t.
  • Even if they could optimize one algorithm, they won’t know about a different algorithm that might be a much better “match” to the processor
Outline

• Idealized and actual costs in modern processors

• Parallelism within single processors
  • Hidden from software (sort of)
  • Pipelining
  • SIMD units

• Memory hierarchies
  • Use of microbenchmarks to characterized performance

• Case study: Matrix Multiplication
  • Use of performance models to understand performance
What is Pipelining?

Dave Patterson’s Laundry example: 4 people doing laundry

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

• In this example:
  • Sequential execution takes 4 * 90 min = 6 hours
  • Pipelined execution takes 30 + 4 * 40 + 20 = 3.5 hours
• Bandwidth = loads/hour
  • BW = 4/6 l/h w/o pipelining
  • BW = 4/3.5 l/h w pipelining
• BW <= 1.5 l/h w pipelining, more total loads
• Pipelining helps bandwidth but not latency (90 min)
• Bandwidth limited by slowest pipeline stage
• Potential speedup = Number pipe stages
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 processing
  • traditional mode
  • one operation produces one result

• SIMD processing
  • with SSE / SSE2
  • one operation produces multiple results

Slide Source: Alex Klimovitski & Dean Macri, Intel Corporation
SSE / SSE2 SIMD on Intel

- SSE2 data types: anything that fits into 16 bytes, e.g.,
  - Instructions perform add, multiply etc. on all the data in this 16-byte register in parallel
- Challenges:
  - Need to be contiguous in memory and aligned
  - Some instructions to move data around from one part of register to another
What does this mean to you?

• In addition to SIMD extensions, the processor may have other special instructions
  • Fused Multiply-Add (FMA) instructions:
    \[ x = y + c \times z \]
    is so common some processor execute the multiply/add as a single instruction, at the same rate (bandwidth) as + or * alone

• In theory, the compiler understands all of this
  • When compiling, it will rearrange instructions to get a good “schedule” that maximizes pipelining, uses FMAs and SIMD
  • It works with the mix of instructions inside an inner loop or other block of code

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

- Whenever you look at performance, it’s useful to understand what you’re expecting.
- Machine peak is the maximum number of arithmetic operations a processor (or set of them) can perform:
  - Often referred to as “speed of light” or “guaranteed not to exceed” number.
  - Treat integer and floating point separately and be clear on how wide (# bits) the operands have.
- If you see a number higher than peak, you have a mistake in your timers, formulas, or …


Peak Performance Examples

• Example 1: Power5 procs in Bassi at NERSC
  • 1.9 GHz (G cycles/sec) with 4 double precision floating point ops/cycle → 7.6 GFlop/s peak per processor
  • Two fp units, each can do a fused MADD (throughput!)
  • Each SMP has 8 processors → 60.8 Gflop/s

• Example 2: 20 of the PSI nodes (in CITRIS)
  • 2.33GHz with SSE2 (2-wide for 64-bit FPUs with FMA) → 9.32 GFlop/s
  • 2 Quad-Cores per node → 74.56 Gflop/s
Outline

• Idealized and actual costs in modern processors
• Parallelism within single processors
• Memory hierarchies
  • Temporal and spatial locality
  • Basics of caches
  • Use of microbenchmarks to characterized performance
• Case study: Matrix Multiplication
  • Use of performance models to understand performance
Memory Hierarchy

- Most programs have a high degree of **locality** in their accesses
  - **spatial locality**: accessing things nearby previous accesses
  - **temporal locality**: reusing an item that was previously accessed
- Memory hierarchy tries to exploit locality
Processor-DRAM Gap (latency)

- Memory hierarchies are getting deeper
  - Processors get faster more quickly than memory

"Moore’s Law"

Processor-Memory Performance Gap:
(grows 50% / year)

- DRAM 7%/yr.
- μProc 60%/yr.
Approaches to Handling Memory Latency

- Bandwidth has improved more than latency
- Approach to address the memory latency problem
  - Eliminate memory operations by saving values in small, fast memory (cache) and reusing them
    - need **temporal locality** in program
  - Take advantage of better bandwidth by getting a chunk of memory and saving it in small fast memory (cache) and using whole chunk
    - need **spatial locality** in program
  - Take advantage of better bandwidth by allowing processor to issue multiple reads to the memory system at once
    - concurrency in the instruction stream, eg load whole array, as in vector processors; or prefetching
Little’s Law

• Latency vs. Bandwidth
  • Latency is physics (wire length)
    • e.g., the network latency on the Earth Simulation is only about 2x times the speed of light across the machine room
  • Bandwidth is cost:
    • add more wires to increase bandwidth (over-simplification)
• Principle (Little's Law): the relationship of a production system in steady state is:
  \[ \text{Inventory} = \text{Throughput} \times \text{Flow Time} \]
• For parallel computing, Little’s Law is about the required concurrency to be limited by bandwidth rather than latency
  • Required concurrency = Bandwidth * Latency
    \[
    \text{bandwidth-delay product}
    \]
• For parallel computing, this means:
  \[ \text{Concurrency} = \text{bandwidth} \times \text{latency} \]
Little’s Law

• Example 1: a single processor:
  • If the latency is to memory is 50ns, and the bandwidth is 5 GB/s (.2ns / Bytes = 12.8 ns / 64-byte cache line)
  • The system must support $50/12.8 \approx 4$ outstanding cache line misses to keep things balanced (run at bandwidth speed)
  • An application must be able to prefetch 4 cache line misses in parallel (without dependencies between them)

• Example 2: 1000 processor system
  • 1 GHz clock, 100 ns memory latency, 100 words of memory in data paths between CPU and memory.
  • Main memory bandwidth is:
    \[ \sim 1000 \times 100 \text{ words} \times 10^9/\text{s} = 10^{14} \text{ words/sec}. \]
  • To achieve full performance, an application needs:
    \[ \sim 10^{-7} \times 10^{14} = 10^7 \text{ way concurrency} \]
    (some of that may be hidden in the instruction stream)
Cache Basics (Review)

- **Cache** is fast (expensive) memory which keeps copy of data in main memory; it is hidden from software
- **Cache hit**: in-cache memory access—cheap
- **Cache miss**: non-cached memory access—expensive
  - Need to access next, slower level of cache
- Consider a tiny cache (for illustration only)

<table>
<thead>
<tr>
<th>Address Pattern</th>
<th>Data (2 Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>X000</td>
<td>101000 through 101001</td>
</tr>
<tr>
<td>X010</td>
<td>001010 through 001011</td>
</tr>
<tr>
<td>X100</td>
<td>111100 through 111101</td>
</tr>
<tr>
<td>X110</td>
<td>110110 through 110111</td>
</tr>
</tbody>
</table>

- **Cache line length**: # of bytes loaded together in one entry
  - 2 in above example
- **Associativity**
  - direct-mapped: only 1 address (line) in a given range in cache
  - n-way: \( n \geq 2 \) lines with different addresses can be stored
Why Have Multiple Levels of Cache?

• On-chip vs. off-chip
  • On-chip caches are faster, but limited in size
• 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
  • IBM uses a level of cache as a “victim cache” which is cheaper
• There are other levels of the memory hierarchy
  • Register, pages (TLB, virtual memory), …
  • And it isn’t always a hierarchy
Experimental Study of Memory (Membench)

- Microbenchmark for memory system performance

- for array A of length L from 4KB to 8MB by 2x
  for stride s from 4 Bytes (1 word) to L/2 by 2x
  time the following loop
    (repeat many times and average)
    for i from 0 to L by s
      load A[i] from memory (4 Bytes)
Membench: What to Expect

- Consider the average cost per load
  - Plot one line for each array length, time vs. stride
  - Small stride is best: if cache line holds 4 words, at most ¼ miss
  - If array is smaller than a given cache, all those accesses will hit (after the first run, which is negligible for large enough runs)
- Picture assumes only one level of cache
- Values have gotten more difficult to measure on modern procs
Memory Hierarchy on a Pentium III

Katmai processor on Millennium, 550 MHz

Array size

- 4KB
- 8KB
- 16KB
- 32KB
- 64KB
- 128KB
- 256KB
- 512KB
- 1MB
- 2MB
- 4MB
- 8MB
- 16MB
- 32MB
- 64MB

L1: 32 byte line?

L2: 512 KB 60 ns

L1: 64K 5 ns, 4-way?
Memory Hierarchy on a Power3 (Seaborq)

Power3, 375 MHz

L1: 32 KB
128B line
.5-2 cycles

L2: 8 MB
128B line
9 cycles

Mem: 396 ns
(132 cycles)
Memory Performance on Itanium 2 (CITRIS)

Itanium2, 900 MHz

L2: 256 KB
128 B line
3-20 cycles

L3: 2 MB
128 B line
3-20 cycles

L1: 32 KB
64B line
.5-4 cycles

Mem:
11-60 cycles

Uses MAPS Benchmark: http://www.sdsc.edu/PMaC/MAPs/maps.html
Stanza Triad

- Even smaller benchmark for prefetching
- Derived from STREAM Triad
- **Stanza (L)** is the length of a unit stride run
  
  ```
  while i < arraylength
    for each L element stanza
      A[i] = scalar * X[i] + Y[i]
    skip k elements
  ```

  ![Diagram](image)

  1) do L triads 2) skip k elements 3) do L triads

Source: Kamil et al, MSP05
Stanza Triad Results

- This graph (x-axis) starts at a cache line size (>=16 Bytes)
- If cache locality was the only thing that mattered, we would expect
  - Flat lines equal to measured memory peak bandwidth (STREAM) as on Pentium3
- Prefetching gets the next cache line (pipelining) while using the current one
  - This does not “kick in” immediately, so performance depends on L
Lessons

• Actual performance of a simple program can be a complicated function of the architecture
  • Slight changes in the architecture or program change the performance significantly
  • To write fast programs, need to consider architecture
    • True on sequential or parallel processor
  • We would like simple models to help us design efficient algorithms

• We will illustrate with a common technique for improving cache performance, called blocking or tiling
  • Idea: used divide-and-conquer to define a problem that fits in register/L1-cache/L2-cache
Outline

• Idealized and actual costs in modern processors
• Parallelism within single processors
• Memory hierarchies
  • Use of microbenchmarks to characterized performance
• Case study: Matrix Multiplication
  • Use of performance models to understand performance
  • Simple cache model
  • Warm-up: Matrix-vector multiplication
  • Case study continued next time
Why Matrix Multiplication?

• An important kernel in scientific problems
  • Appears in many linear algebra algorithms
  • Closely related to other algorithms, e.g., transitive closure on a graph using Floyd-Warshall

• Optimization ideas can be used in other problems

• The best case for optimization payoffs

• The most-studied algorithm in parallel computing
Administrivia

- Homework 1 (with the code!) is online
  - Lot to understand, relatively little code to write
  - Understanding performance results and code important in grade!
- Homework 0 should be done
- Submit should (finally) be working, but perhaps not on CITRIS/PSI machines (use instructional machines to submit)
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 (for now)

![Column major matrix in memory diagram]
Using a Simple Model of Memory to Optimize

- 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
  - \( f \) = number of arithmetic operations
  - \( t_f \) = time per arithmetic operation \(<< t_m \)
  - \( q = f / m \) average number of flops per slow memory access
- Minimum possible time = \( f \cdot t_f \) when all data in fast memory
- Actual time
  - \( f \cdot t_f + m \cdot t_m = f \cdot t_f \cdot (1 + t_m / t_f \cdot 1/q) \)
- Larger \( q \) means time closer to minimum \( f \cdot t_f \)
  - \( q \geq t_m / t_f \) needed to get at least half of peak speed
Warm up: Matrix-vector multiplication

\{\text{implements } y = y + A^*x\}\n
\text{for } i = 1:n \\
\hspace{1cm} \text{for } j = 1:n \\
\hspace{2cm} y(i) = y(i) + A(i,j)\cdot 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 = \( 2n^2 \)
• \( q \) = \( f / m \) \( \approx 2 \)

• Matrix-vector multiplication limited by slow memory speed
Modeling Matrix-Vector Multiplication

- Compute time for nxn = 1000x1000 matrix
- Time
  - \( f \times t_f + m \times t_m = f \times t_f \times (1 + t_m/t_f \times 1/q) \)
  - \( = 2n^2 \times t_f \times (1 + t_m/t_f \times 1/2) \)
- For \( t_f \) and \( t_m \), using data from R. Vuduc’s PhD (pp 351-3)
  - For \( t_m \) use minimum-memory-latency / words-per-cache-line

<table>
<thead>
<tr>
<th>Machine</th>
<th>Clock MHz</th>
<th>Peak Mflop/s</th>
<th>Mem Lat (Min,Max) cycles</th>
<th>Linesize Bytes</th>
<th>( t_m/t_f )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ultra 2i</td>
<td>333</td>
<td>667</td>
<td>38</td>
<td>66</td>
<td>16</td>
</tr>
<tr>
<td>Ultra 3</td>
<td>900</td>
<td>1800</td>
<td>28</td>
<td>200</td>
<td>32</td>
</tr>
<tr>
<td>Pentium 3</td>
<td>500</td>
<td>500</td>
<td>25</td>
<td>60</td>
<td>32</td>
</tr>
<tr>
<td>Pentium3M</td>
<td>800</td>
<td>800</td>
<td>40</td>
<td>60</td>
<td>32</td>
</tr>
<tr>
<td>Power3</td>
<td>375</td>
<td>1500</td>
<td>35</td>
<td>139</td>
<td>128</td>
</tr>
<tr>
<td>Power4</td>
<td>1300</td>
<td>5200</td>
<td>60</td>
<td>10000</td>
<td>128</td>
</tr>
<tr>
<td>Itanium1</td>
<td>800</td>
<td>3200</td>
<td>36</td>
<td>85</td>
<td>32</td>
</tr>
<tr>
<td>Itanium2</td>
<td>900</td>
<td>3600</td>
<td>11</td>
<td>60</td>
<td>64</td>
</tr>
</tbody>
</table>
Simplifying Assumptions

- What simplifying assumptions did we make in this analysis?
  - Ignored parallelism in processor between memory and arithmetic within the processor
    - Sometimes drop arithmetic term in this type of analysis
  - Assumed fast memory was large enough to hold three vectors
    - Reasonable if we are talking about any level of cache
    - Not if we are talking about registers (~32 words)
  - Assumed the cost of a fast memory access is 0
    - Reasonable if we are talking about registers
    - Not necessarily if we are talking about cache (1-2 cycles for L1)
  - Memory latency is constant
- Could simplify even further by ignoring memory operations in X and Y vectors
  - Mflop rate/element = \( \frac{2}{2 \times t_f + t_m} \)
Validating the Model

- How well does the model predict actual performance?
  - Actual DGEMV: Most highly optimized code for the platform
- Model sufficient to compare across machines
- But under-predicting on most recent ones due to latency estimate
Summary

- Details of machine are important for performance
  - Processor and memory system (not just parallelism)
  - Before you parallelize, make sure you’re getting good serial performance
  - What to expect? Use understanding of hardware limits

- There is parallelism hidden within processors
  - Pipelining, SIMD, etc

- Locality is at least as important as computation
  - Temporal: re-use of data recently used
  - Spatial: using data nearby that recently used

- Machines have memory hierarchies
  - 100s of cycles to read from DRAM (main memory)
  - Caches are fast (small) memory that optimize average case

- Can rearrange code/data to improve locality