CMPT 880/479
Multicore Programming and Architecture

CMPT 880/479
Multicore Programming and Architecture

Unit 1: Performance Tuning

CMPT 880/479
Multicore Programming and Architecture

Unit 1: Performance Tuning

Today's Agenda

Today's Agenda

- Why all this talk about sequential performance?
  - Seldom taught
  - Even more rarely practiced in coursework
  - Key foundation for efficient parallel code

Today's Agenda

- Why all this talk about sequential performance?
  - Seldom taught
  - Even more rarely practiced in coursework
  - Key foundation for efficient parallel code

Today's Agenda

- Why all this talk about sequential performance?
  - Seldom taught
  - Even more rarely practiced in coursework
  - Key foundation for efficient parallel code

- Lecture on sequential performance tuning
  - Overview, much of it should be review
  - Help you tune code
  - Help you understand odd performance effects

The Process of Performance Tuning

Goals of Performance Tuning

Goals of Performance Tuning

- Reduce resource usage to "go faster"
  - **Runtime**: wall clock time for a single task
  - **Throughput**: items of work per unit time

Goals of Performance Tuning

- Reduce resource usage to "go faster"
  - **Runtime**: wall clock time for a single task
  - **Throughput**: items of work per unit time
- **Memory usage**

Goals of Performance Tuning

- Reduce resource usage to "go faster"
  - **Runtime**: wall clock time for a single task
  - **Throughput**: items of work per unit time
- **Memory usage**
- Or, **energy consumption**
  - Same computation, less energy
  - Why? **battery life, power & cooling costs, fan noise**
  - Run fast and sleep **--or--** run slowly just to meet deadlines

Goals of Performance Tuning

- Reduce resource usage to "go faster"
  - **Runtime**: wall clock time for a single task
  - **Throughput**: items of work per unit time
- **Memory usage**
- Or, **energy consumption**
  - Same computation, less energy
  - Why? **battery life, power & cooling costs, fan noise**
  - Run fast and sleep **--or--** run slowly just to meet deadlines
- Either way: **software tuning basically the same**
  - In fact, processors dynamically trade energy and performance
    - Dynamic voltage/frequency scaling, Core i7 "turbo mode"

Three Hats of Software Development

Three Hats of Software Development

- Starting point: always some pile of existing code

Three Hats of Software Development

- Starting point: always some pile of existing code
- **Refactoring:** No change to visible external behavior

Three Hats of Software Development

- Starting point: always some pile of existing code
- **Refactoring:** No change to visible external behavior
- **Add new functionality**
  - Change behavior

Three Hats of Software Development

- Starting point: always some pile of existing code
- **Refactoring:** No change to visible external behavior
- **Add new functionality**
  - Change behavior
- **Performance optimization**: empirical & experimental
  - 1. wait
  - 2. benchmark and profile, identify bottleneck
  - 3. Modify code, test functionality, measure perf., revert if not beneficial
  - Ongoing performance regression testing

Three Hats of Software Development

- Starting point: always some pile of existing code
- **Refactoring:** No change to visible external behavior
- **Add new functionality**
  - Change behavior
- **Performance optimization**: empirical & experimental
  - 1. wait
  - 2. benchmark and profile, identify bottleneck
  - 3. Modify code, test functionality, measure perf., revert if not beneficial
  - Ongoing performance regression testing
- "We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**"
  - Donald Knuth

Identifying Performance Bottlenecks

Identifying Performance Bottlenecks

- **Where is the code spending its time?**
  - Need representative input (full sized, long running)
  - Various profiling tools (gprof, Valgrind's cachegrind, Intel's VTune)
  - Look beyond code profile (collect by object, algorithm change, etc.)

Identifying Performance Bottlenecks

- **Where is the code spending its time?**
  - Need representative input (full sized, long running)
  - Various profiling tools (gprof, Valgrind's cachegrind, Intel's VTune)
  - Look beyond code profile (collect by object, algorithm change, etc.)
- **What is the cause of the slow performance?**
  - Might not always be what you think

Identifying Performance Bottlenecks

- **Where is the code spending its time?**
  - Need representative input (full sized, long running)
  - Various profiling tools (gprof, Valgrind's cachegrind, Intel's VTune)
  - Look beyond code profile (collect by object, algorithm change, etc.)
- **What is the cause of the slow performance?**
  - Might not always be what you think
- **Hardware performance counters**
  - Various profiling tools collect statistics from the hardware
  - Examples: stall cycles, cache miss rates, branch mispredictions, etc.
  - Can even associate them with specific instructions
  - Using sampling, can be done with low overhead (a few percent)

Identifying Performance Bottlenecks

- **Where is the code spending its time?**
  - Need representative input (full sized, long running)
  - Various profiling tools (gprof, Valgrind's cachegrind, Intel's VTune)
  - Look beyond code profile (collect by object, algorithm change, etc.)
- **What is the cause of the slow performance?**
  - Might not always be what you think
- **Hardware performance counters**
  - Various profiling tools collect statistics from the hardware
  - Examples: stall cycles, cache miss rates, branch mispredictions, etc.
  - Can even associate them with specific instructions
  - Using sampling, can be done with low overhead (a few percent)
- **Focused experiments** (example: large v. small data set)

Performance Tuning of Sequential Code

Overview: Sequential Code Performance

Overview: Sequential Code Performance

- Fast parallel code is fast sequential code running in parallel

Overview: Sequential Code Performance

- Fast parallel code is fast sequential code running in parallel
- Runtime = # of inst. * cycles per inst. * clock period
  - Want to minimize all three

Overview: Sequential Code Performance

- Fast parallel code is fast sequential code running in parallel
- Runtime = # of inst. * cycles per inst. * clock period
  - Want to minimize all three
- When tuning code, consider:
  - "Dynamic" instruction count
  - Micro-architectural impacts
    - Instruction level parallelism of code, branch mispredictions
  - Memory hierarchy
    - Cache misses, etc.

Reducing Instruction Count

Reducing Instruction Count

- Dynamic instruction count is first-order performance metric
  - "Dynamic" means at runtime

Reducing Instruction Count

- Dynamic instruction count is first-order performance metric
  - "Dynamic" means at runtime
- **First order model: runtime is ~ to instruction count**
  - Assumes all instruction take a single cycle
  - Con: ignores all micro-architectural and memory effects
  - Pro: architecture independent, easier to reason about, captures most important metric

Reducing Instruction Count

- Dynamic instruction count is first-order performance metric
  - "Dynamic" means at runtime
- **First order model: runtime is ~ to instruction count**
  - Assumes all instruction take a single cycle
  - Con: ignores all micro-architectural and memory effects
  - Pro: architecture independent, easier to reason about, captures most important metric
- **Reducing instruction count**
  - Better algorithm or approach to problem
  - Avoid software bloat in terms of too much layering, too general
  - Write code the compiler can easily optimize (and turn on -O3)
    - Compilers are really smart, but lack higher level context

Cost of Function Calls

Cost of Function Calls

- **Function calls add overhead**
  - Obvious overhead: “Call” and “Return” instruction overheads

Cost of Function Calls

- **Function calls add overhead**
  - Obvious overhead: "Call" and "Return" instruction overheads
- **Less obvious**
  - Putting parameters/arguments into right registers
    - Or on in-memory stack
  - Calling into a dynamically linked library can be a bit more expensive
  - Virtual functions (C++ and Java): adds a table lookup on call path

Cost of Function Calls

- **Function calls add overhead**
  - Obvious overhead: "Call" and "Return" instruction overheads
- **Less obvious**
  - Putting parameters/arguments into right registers
    - Or on in-memory stack
  - Calling into a dynamically linked library can be a bit more expensive
  - Virtual functions (C++ and Java): adds a table lookup on call path
- **Worse yet, breaks compiler optimization boundaries**
  - Compilers today typically don't look at code in other C files
  - After a call, compiler doesn't know what has changed or not
  - Compilers can inline functions, especially in C header files

Cost of Function Calls

- **Function calls add overhead**
  - Obvious overhead: "Call" and "Return" instruction overheads
- **Less obvious**
  - Putting parameters/arguments into right registers
    - Or on in-memory stack
  - Calling into a dynamically linked library can be a bit more expensive
  - Virtual functions (C++ and Java): adds a table lookup on call path
- **Worse yet, breaks compiler optimization boundaries**
  - Compilers today typically don't look at code in other C files
  - After a call, compiler doesn't know what has changed or not
  - Compilers can inline functions, especially in C header files
- Aside: "extern inline" function in .h file is a fast as a macro

Operation & Instruction Latencies

Operation & Instruction Latencies

- Instruction latencies
  - One cycle: integer adds, subtract, compare, shift, etc.
  - Few cycles: loads/stores (best case), int multiply, floating point
  - Dozens of cycles: integer divide (89 cycles for 64-bit on Core i7)

Operation & Instruction Latencies

- Instruction latencies
  - One cycle: integer adds, subtract, compare, shift, etc.
  - Few cycles: loads/stores (best case), int multiply, floating point
  - Dozens of cycles: integer divide (89 cycles for 64-bit on Core i7)
- Non-instructions (on most machines)
  - Sqrt, exponentiation, sine/cosine, modulo (%), etc.
  - Trend toward more hardware support for these operations

Operation & Instruction Latencies

- Instruction latencies
  - One cycle: integer adds, subtract, compare, shift, etc.
  - Few cycles: loads/stores (best case), int multiply, floating point
  - Dozens of cycles: integer divide (89 cycles for 64-bit on Core i7)
- Non-instructions (on most machines)
  - Sqrt, exponentiation, sine/cosine, modulo (%), etc.
  - Trend toward more hardware support for these operations
- Compilers can by help
  - Selects cheapest instructions (strength reduction)
    - Replace multiply by constant power-of-two with shift: "x * 2" with "x << 1"
  - Tries to eliminate redundant computation

Tuning for Micro-Architectural Impacts

Tuning for Micro-Architectural Impacts

- Superscalar execution: three or four insn per cycle max
  - Restrictions on instruction mix (different function unit types)
  - Loads/stores are most limited (just one or maybe two per cycle)
  - Compiler reorganizes and schedules code with this in mind

Tuning for Micro-Architectural Impacts

- Superscalar execution: three or four insn per cycle max
  - Restrictions on instruction mix (different function unit types)
  - Loads/stores are most limited (just one or maybe two per cycle)
  - Compiler reorganizes and schedules code with this in mind
- Out-of-order (dynamic scheduling)
  - Code runs faster, but makes performance difficult to understand

Tuning for Micro-Architectural Impacts

- Superscalar execution: three or four insn per cycle max
  - Restrictions on instruction mix (different function unit types)
  - Loads/stores are most limited (just one or maybe two per cycle)
  - Compiler reorganizes and schedules code with this in mind
- Out-of-order (dynamic scheduling)
  - Code runs faster, but makes performance difficult to understand
- Branching
  - "Taken" branches disrupt fetch
  - Branch mis-predictions flush pipeline, throws away work
    - Cost hard to quantify, but often dozens of cycles
  - Try to avoid unpredictable branches

Tuning for Micro-Architectural Impacts

- Superscalar execution: three or four insn per cycle max
  - Restrictions on instruction mix (different function unit types)
  - Loads/stores are most limited (just one or maybe two per cycle)
  - Compiler reorganizes and schedules code with this in mind
- Out-of-order (dynamic scheduling)
  - Code runs faster, but makes performance difficult to understand
- Branching
  - "Taken" branches disrupt fetch
  - Branch mis-predictions flush pipeline, throws away work
    - Cost hard to quantify, but often dozens of cycles
  - Try to avoid unpredictable branches
- Loop accelerators (if loop body is <N insn, goes faster)

Code Example I

```
int distance(int a, int b)
{
  return (a > b) ? a-b : b-a;
}
```

Code Example I

```
int distance(int a, int b)
{
  return (a > b) ? a-b : b-a;
}
```

gcc -m32 -O3

```
distance:
   pushl %ebp
   movl  %esp, %ebp
   movl  8(%ebp), %edx
   movl  12(%ebp), %eax
   cmpl  %eax, %edx
   jg    .L6
   subl  %edx, %eax
   popl  %ebp
   ret
.L6:
   subl  %eax, %edx
   movl  %edx, %eax
   popl  %ebp
   ret
```

Code Example I

```
int distance(int a, int b)
{
  return (a > b) ? a-b : b-a;
}
```

gcc -m32 -O3

```
distance:
   pushl %ebp
   movl  %esp, %ebp
   movl  8(%ebp), %edx
   movl  12(%ebp), %eax
   cmpl  %eax, %edx
   jg    .L6
   subl  %edx, %eax
   popl  %ebp
   ret
.L6:
   subl  %eax, %edx
   movl  %edx, %eax
   popl  %ebp
   ret
```

gcc -m32 -march=core2 -O3

```
distance:
   pushl  %ebp
   movl   %esp, %ebp
   pushl  %ebx
   movl   8(%ebp), %ecx
   movl   12(%ebp), %edx
   movl   %ecx, %eax
   movl   %edx, %ebx
   subl   %edx, %eax
   subl   %ecx, %ebx
   cmpl   %edx, %ecx
   cmovle %ebx, %eax
   popl   %ebx
   leave
   ret
```

Code Example I

```
int distance(int a, int b)
{
  return (a > b) ? a-b : b-a;
}
```

gcc -m32 -O3

```
distance:
   pushl %ebp
   movl  %esp, %ebp
   movl  8(%ebp), %edx
   movl  12(%ebp), %eax
   cmpl  %eax, %edx
   jg    .L6
   subl  %edx, %eax
   popl  %ebp
   ret
.L6:
   subl  %eax, %edx
   movl  %edx, %eax
   popl  %ebp
   ret
```

gcc -m32 -march=core2 -O3

```
distance:
   pushl  %ebp
   movl   %esp, %ebp
   pushl  %ebx
   movl   8(%ebp), %ecx
   movl   12(%ebp), %edx
   movl   %ecx, %eax
   movl   %edx, %ebx
   subl   %edx, %eax
   subl   %ecx, %ebx
   cmpl   %edx, %ecx
   cmovle %ebx, %eax
   popl   %ebx
   leave
   ret
```

gcc -m64 -O3

```
distance:
   movl   %edi, %eax
   movl   %esi, %edx
   subl   %esi, %eax
   subl   %edi, %edx
   cmpl   %esi, %edi
   cmovle %edx, %eax
   ret
```

Code Example I

```
int distance(int a, int b)
{
  return (a > b) ? a-b : b-a;
}
```

gcc -m32 -O3

```
distance:
   pushl %ebp
   movl  %esp, %ebp
   movl  8(%ebp), %edx
   movl  12(%ebp), %eax
   cmpl  %eax, %edx
   jg    .L6
   subl  %edx, %eax
   popl  %ebp
   ret
.L6:
   subl  %eax, %edx
   movl  %edx, %eax
   popl  %ebp
   ret
```

gcc -m32 -march=core2 -O3

```
distance:
   pushl  %ebp
   movl   %esp, %ebp
   pushl  %ebx
   movl   8(%ebp), %ecx
   movl   12(%ebp), %edx
   movl   %ecx, %eax
   movl   %edx, %ebx
   subl   %edx, %eax
   subl   %ecx, %ebx
   cmpl   %edx, %ecx
   cmovle %ebx, %eax
   popl   %ebx
   leave
   ret
```

gcc -m64 -O3

```
distance:
   movl   %edi, %eax
   movl   %esi, %edx
   subl   %esi, %eax
   subl   %edi, %edx
   cmpl   %esi, %edi
   cmovle %edx, %eax
   ret
```

- Note: only works (on x86) for non-memory operations
  - Why? Must ensure the memory operation won't seg fault

Code Example II

```
int func(int a, int b, int* array)
{
  return (a > 0) ? b : array[b];
}
```

Code Example II

```
int func(int a, int b, int* array)
{
  return (a > 0) ? b : array[b];
}
```

```
  func:
        testl   %edi, %edi
        jg      .L2
        movslq  %esi,%rax
        movl    (%rdx,%rax,4), %esi
  .L2:
        movl    %esi, %eax
        ret
```

- Comments on x86 assembly
  - Right-most register is usually the output register
  - "mov" instructions are loads, stores, or register copies
    - Memory access is indicated by "(...)"
  - "%exx" are 32-bit registers, "%rxx" are 64-bit registers
  - movslq: "sign extending" a "long" (32-bit) to a quad (64-bit)

Code Example II

```
int func(int a, int b, int* array)
{
  return (a > 0) ? b : array[b];
}
```

```
int func2(int a, int b, int* array)
{
  int temp = array[b];
  return (a > 0) ? b : temp;
}
```

```
 func:
        testl   %edi, %edi
        jg      .L2
        movslq  %esi,%rax
        movl    (%rdx,%rax,4), %esi
 .L2:
        movl    %esi, %eax
        ret
```

- Comments on x86 assembly
  - Right-most register is usually the output register
  - "mov" instructions are loads, stores, or register copies
    - Memory access is indicated by "(...)"
  - "%exx" are 32-bit registers, "%rxx" are 64-bit registers
  - movslq: "sign extending" a "long" (32-bit) to a quad (64-bit)

Code Example II

```
int func(int a, int b, int* array)
{
  return (a > 0) ? b : array[b];
}
```

```
func:
        testl   %edi, %edi
        jg      .L2
        movslq  %esi,%rax
        movl    (%rdx,%rax,4), %esi
.L2:
        movl    %esi, %eax
        ret
```

```
int func2(int a, int b, int* array)
{
  int temp = array[b];
  return (a > 0) ? b : temp;
}
```

```
func2:
        movslq  %esi,%rax
        testl   %edi, %edi
        cmovle  (%rdx,%rax,4), %esi
        movl    %esi, %eax
        ret
```

- Comments on x86 assembly
  - Right-most register is usually the output register
  - "mov" instructions are loads, stores, or register copies
    - Memory access is indicated by "(...)"
  - "%exx" are 32-bit registers, "%rxx" are 64-bit registers
  - movslq: "sign extending" a "long" (32-bit) to a quad (64-bit)

What Can the Compiler Do or Not Do?

```
void loop(int64 n, int64* array1, int64* array2, int64* ptr)
{
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (*ptr > array2[j]) {
        array1[i] += array2[j];
      }
    }
  }
}
```

What Can the Compiler Do or Not Do?

```
void loop(int64 n, int64* array1, int64* array2, int64* ptr)
{
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (*ptr > array2[j]) {
        array1[i] += array2[j];
      }
    }
  }
}
```

- How might you rewrite the code to help?

Code Example III

```
void loop(int64 n, int64* array1,
          int64* array2, int64* ptr)
{
  for (int64 i = 0; i < n; i++) {
    for (int64 j = 0; j < n; j++) {
      if (*ptr > array2[j]) {
        array1[i] += array2[j];
      }
    }
  }
}
```

Code Example III

```
void loop(int64 n, int64* array1,
          int64* array2, int64* ptr)
{
  for (int64 i = 0; i < n; i++) {
    for (int64 j = 0; j < n; j++) {
      if (*ptr > array2[j]) {
        array1[i] += array2[j];
      }
    }
  }
}
```

Inner loop of loop()

```
.L5:  movq  (%rdx,%rax,8), %r8
      cmpq  %r8, (%rcx)
      jle   .L4
      addq  %r8, (%rsi,%r9,8)
.L4:  addq  $1, %rax
      cmpq  %rax, %rdi
      jg    .L5
```

load, add, store

Code Example III

```
void loop(int64 n, int64* array1,
          int64* array2, int64* ptr)
{
  for (int64 i = 0; i < n; i++) {
    for (int64 j = 0; j < n; j++) {
      if (*ptr > array2[j]) {
        array1[i] += array2[j];
      }
    }
  }
}
```

```
void loop2(int64 n, int64* array1,
           int64* array2, int64* ptr)
{
  for (int64 i = 0; i < n; i++) {
    int64 accumulator = 0;
    for (int64 j = 0; j < n; j++) {
      if (*ptr > array2[j]) {
        accumulator += array2[j];
      }
    }
    array1[i] = accumulator;
  }
}
```

Inner loop of loop()

```
.L5:  movq  (%rdx,%rax,8), %r8
      cmpq  %r8, (%rcx)
      jle   .L4
      addq  %r8, (%rsi,%r9,8)
.L4:  addq  $1, %rax
      cmpq  %rax, %rdi
      jg    .L5
```

load, add, store

Code Example III

```
void loop(int64 n, int64* array1,
          int64* array2, int64* ptr)
{
  for (int64 i = 0; i < n; i++) {
    for (int64 j = 0; j < n; j++) {
      if (*ptr > array2[j]) {
        array1[i] += array2[j];
      }
    }
  }
}

void loop2(int64 n, int64* array1,
           int64* array2, int64* ptr)
{
  for (int64 i = 0; i < n; i++) {
    int64 accumulator = 0;
    for (int64 j = 0; j < n; j++) {
      if (*ptr > array2[j]) {
        accumulator += array2[j];
      }
    }
    array1[i] = accumulator;
  }
}
```

Inner loop of loop()

```
.L5:  movq  (%rdx,%rax,8), %r8
      cmpq  %r8, (%rcx)
      jle   .L4
      addq  %r8, (%rsi,%r9,8)
.L4:  addq  $1, %rax
      cmpq  %rax, %rdi
      jg    .L5
```

load, add, store

Inner loop of loop2()

```
.L13: movq  (%rdx,%rax,8), %r8
      cmpq  %r10, %r8
      jge   .L12
      addq  %r8, %r9
.L12: addq  $1, %rax
      cmpq  %rax, %rdi
      jg    .L13
```

Code Example III

```
void loop(int64 n, int64* array1,
          int64* array2, int64* ptr)
{
  for (int64 i = 0; i < n; i++) {
    for (int64 j = 0; j < n; j++) {
      if (*ptr > array2[j]) {
        array1[i] += array2[j];
      }
    }
  }
}

void loop2(int64 n, int64* array1,
           int64* array2, int64* ptr)
{
  for (int64 i = 0; i < n; i++) {
    int64 accumulator = 0;
    for (int64 j = 0; j < n; j++) {
      if (*ptr > array2[j]) {
        accumulator += array2[j];
      }
    }
    array1[i] = accumulator;
  }
}
```

Inner loop of loop()

```
.L5:  movq  (%rdx,%rax,8), %r8
      cmpq  %r8, (%rcx)
      jle   .L4
      addq  %r8, (%rsi,%r9,8)
.L4:  addq  $1, %rax
      cmpq  %rax, %rdi
      jg    .L5
```

load, add, store

Inner loop of loop2()

```
.L13: movq  (%rdx,%rax,8), %r8
      cmpq  %r10, %r8
      jge   .L12
      addq  %r8, %r9
.L12: addq  $1, %rax
      cmpq  %rax, %rdi
      jg    .L13
```

Inner loop of loop2() + CMOV

```
.L14: movq  (%r11,%r8,8), %rax
      leaq  (%r9,%rax), %rdx
      cmpq  %r10, %rax
      cmovl %rdx, %r9
      addq  $1, %r8
      cmpq  %r8, %rdi
      jg    .L14
```

not a load!

Aside: Profile-Based Compilation

Aside: Profile-Based Compilation

- To help the compiler, you can give it "profile" feedback
  - gcc -fprofile-generate ...
  - <run program>
  - gcc -fprofile-use ...

Aside: Profile-Based Compilation

- To help the compiler, you can give it "profile" feedback
  - gcc -fprofile-generate ...
  - <run program>
  - gcc -fprofile-use ...
- Helps compiler:
  - Know which way branches commonly go
  - Frequency of loops (loop unrolling, loop peeling, etc.)

Aside: Profile-Based Compilation

- To help the compiler, you can give it "profile" feedback
  - gcc -fprofile-generate ...
  - <run program>
  - gcc -fprofile-use ...
- Helps compiler:
  - Know which way branches commonly go
  - Frequency of loops (loop unrolling, loop peeling, etc.)
- Good just-in-time (JIT) compilers do this automatically

Memory Hierarchy Impacts

Memory Hierarchy Impacts

- Caches exploit locality
  - **Temporal**: same memory accessed again soon
  - **Spatial**: in same region of memory (64 byte blocks)

Memory Hierarchy Impacts

- Caches exploit locality
  - **Temporal**: same memory accessed again soon
  - **Spatial**: in same region of memory (64 byte blocks)
- Hierarchy of caches
  - First-level **instruction and data caches** (~32KB): a few cycles
  - Second/Third-level cache (~256KB to ~*MB): a dozen cycles or so
  - **Main memory: ~150 cycles**, highly variable
  - **Limited bandwidth** as well (only so many bits to/from memory)

Memory Hierarchy Impacts

- Caches exploit locality
  - **Temporal**: same memory accessed again soon
  - **Spatial**: in same region of memory (64 byte blocks)
- Hierarchy of caches
  - First-level **instruction and data caches** (~32KB): a few cycles
  - Second/Third-level cache (~256KB to ~*MB): a dozen cycles or so
  - **Main memory: ~150 cycles**, highly variable
  - **Limited bandwidth** as well (only so many bits to/from memory)
- Translation look-aside buffers (TLBs)
  - Map pages (4KB), miss can be expensive (at least dozens of cycles)

Memory Hierarchy Impacts

- Caches exploit locality
  - **Temporal**: same memory accessed again soon
  - **Spatial**: in same region of memory (64 byte blocks)
- Hierarchy of caches
  - First-level **instruction and data caches** (~32KB): a few cycles
  - Second/Third-level cache (~256KB to ~*MB): a dozen cycles or so
  - **Main memory: ~150 cycles**, highly variable
  - **Limited bandwidth** as well (only so many bits to/from memory)
- Translation look-aside buffers (TLBs)
  - Map pages (4KB), miss can be expensive (at least dozens of cycles)
- Hardware prefetching
  - Good, but makes it difficult to reason about

Tuning for the Memory Hierarchy

Tuning for the Memory Hierarchy

- **Increase locality** (temporal or spatial)
  - Change access order to re-use data
  - Pack data for spatial locality
  - Reduce code footprint (instruction cache behavior)

Tuning for the Memory Hierarchy

- **Increase locality** (temporal or spatial)
  - Change access order to re-use data
  - Pack data for spatial locality
  - Reduce code footprint (instruction cache behavior)
- Memory allocation location issues
  - Example: reduce TLB misses by packing objects onto same page

Tuning for the Memory Hierarchy

- **Increase locality** (temporal or spatial)
  - Change access order to re-use data
  - Pack data for spatial locality
  - Reduce code footprint (instruction cache behavior)
- Memory allocation location issues
  - Example: reduce TLB misses by packing objects onto same page
- Software prefetching?
  - **__builtin_prefetch**(*const void *addr*)

Tuning for the Memory Hierarchy

- **Increase locality** (temporal or spatial)
  - Change access order to re-use data
  - Pack data for spatial locality
  - Reduce code footprint (instruction cache behavior)
- Memory allocation location issues
  - Example: reduce TLB misses by packing objects onto same page
- Software prefetching?
  - **__builtin_prefetch**(*const void *addr*)
- **Random versus sequential access patterns**
  - Pointer chasing: hard to prefetch, chain of dependencies
  - Array walking: easy to prefetch, few dependencies

Tuning for the Memory Hierarchy

- **Increase locality** (temporal or spatial)
  - Change access order to re-use data
  - Pack data for spatial locality
  - Reduce code footprint (instruction cache behavior)
- Memory allocation location issues
  - Example: reduce TLB misses by packing objects onto same page
- Software prefetching?
  - **__builtin_prefetch**(*const void *addr*)
- **Random versus sequential access patterns**
  - Pointer chasing: hard to prefetch, chain of dependencies
  - Array walking: easy to prefetch, few dependencies
- Example: is a binary tree or an N-array tree more efficient?
  - Binary tree nodes are small (low spatial locality)
  - Binary trees do lots of pointer indirections

More Information

- "Intel 64 and IA-32 Architectures Optimization Reference Manual"
  - http://www.intel.com/Assets/PDF/manual/248966.pdf
- "Software Optimization Guide for AMD Family 10h Processors"
  - http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/40546.pdf
- "man gcc" for compiler flags
- "What Every Programmer Should Know About Memory"
  - Ulrich Drepper, http://people.redhat.com/drepper/cpumemory.pdf
- "Instruction latencies and throughput for x86 processors"
  - Torbjorn Granlund, http://gmplib.org/~tege/x86-timing.pdf
- "x86-64 Machine-Level Programming"
  - Bryant & O'Hallaron,
  - http://www.cs.cmu.edu/~fp/courses/15213-s06/misc/asm64-handout.pdf

Low-Level Tuning of Parallel Code

Multicore Organization

Multicore Organization

- Multiple threads per "core"
  - Share functional units, data cache, instruction cache

Multicore Organization

- Multiple threads per "core"
  - Share functional units, data cache, instruction cache
- Multiple core per chip (or "socket")
  - Private first-level caches (8KB to 64KB)
  - Maybe a second-level cache (256KB or 512KB)
  - Last-level on-chip cache (4MB to 8MB)
  - Memory controllers for directly connect memory chips (DRAM)

Multicore Organization

- Multiple threads per "core"
  - Share functional units, data cache, instruction cache
- Multiple core per chip (or "socket")
  - Private first-level caches (8KB to 64KB)
  - Maybe a second-level cache (256KB or 512KB)
  - Last-level on-chip cache (4MB to 8MB)
  - Memory controllers for directly connect memory chips (DRAM)
- Multiple sockets per system
  - Dual-socket servers standard for years
  - Quad-socket becoming common (from AMD & Intel)
  - Large systems with hundreds of sockets have been built
  - Today, usually no caching shared among sockets

Multi-Socket Memory System

Multi-Socket Memory System

- Chips connected point-to-point by high-speed links

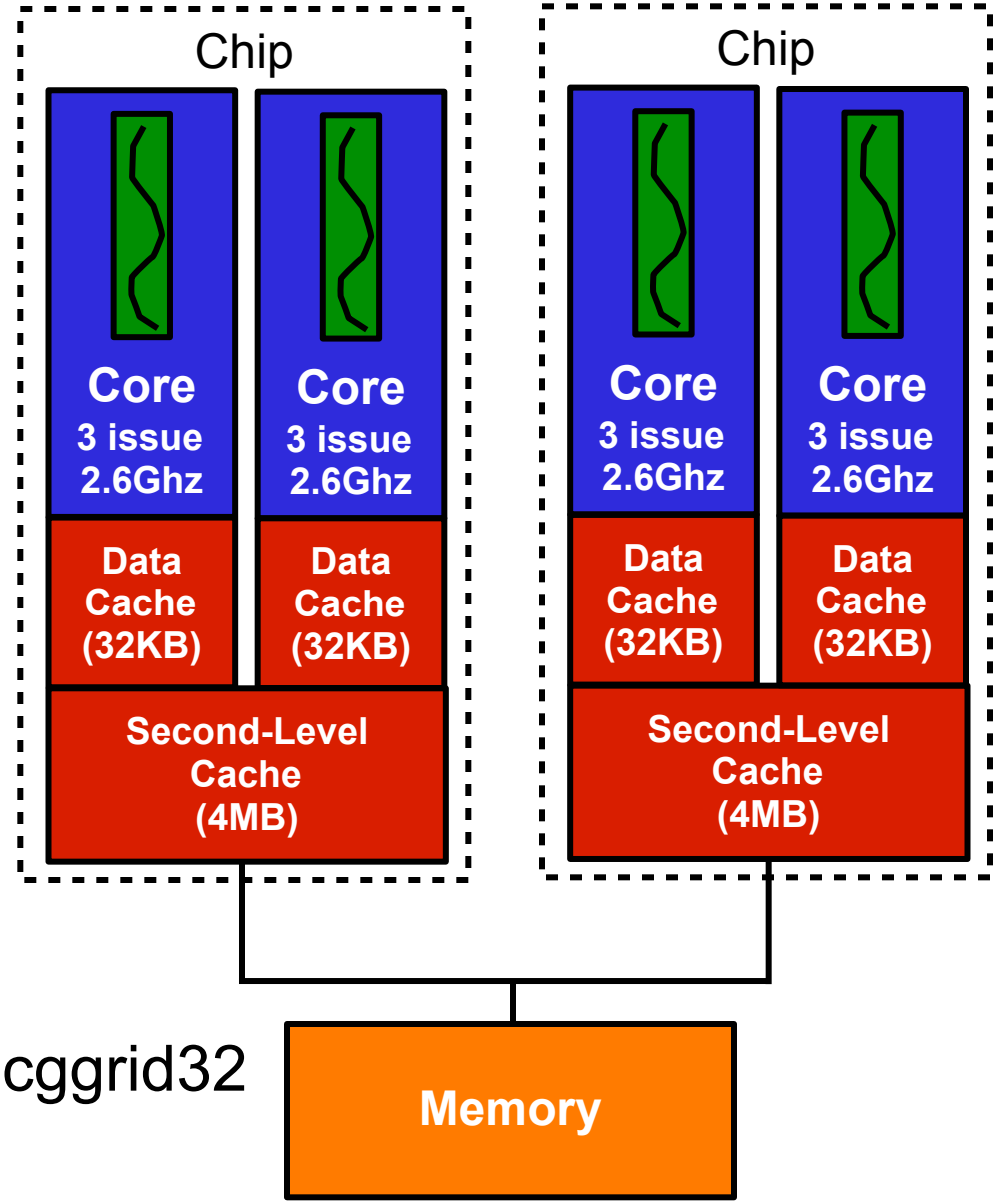
Multi-Socket Memory System

- Chips connected point-to-point by high-speed links
- Each chip connected to 1/Nth the off-chip memory
  - Accessing a "local" page is lower latency than a "remote" page
    - Today, maybe 2x difference at most
  - Old term: Non-Uniform Memory Architecture (NUMA)
    - Was more important when latency difference was 10x or 100x

Multi-Socket Memory System

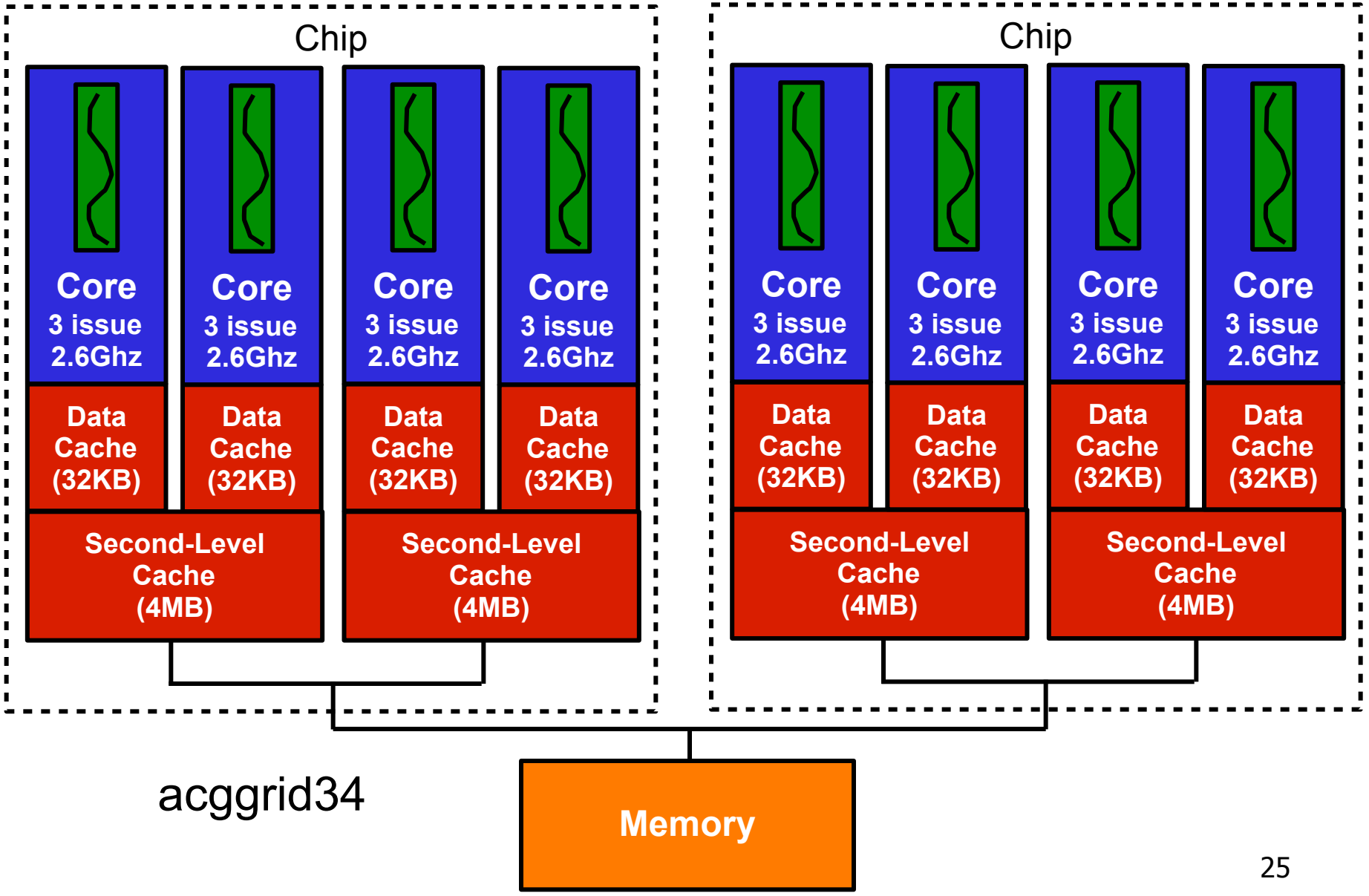
- Chips connected point-to-point by high-speed links
- Each chip connected to 1/Nth the off-chip memory
  - Accessing a "local" page is lower latency than a "remote" page
    - Today, maybe 2x difference at most
  - Old term: Non-Uniform Memory Architecture (NUMA)
    - Was more important when latency difference was 10x or 100x
- Page mapping controlled by OS
  - To optimize for lowest latency:
    - All local pages
  - To optimize for highest bandwidth:
    - Distribute pages to use all memory controllers
  - What happens in practice? oblivious allocation probably does okay

Dual-Socket Dual-Core "Core 2"



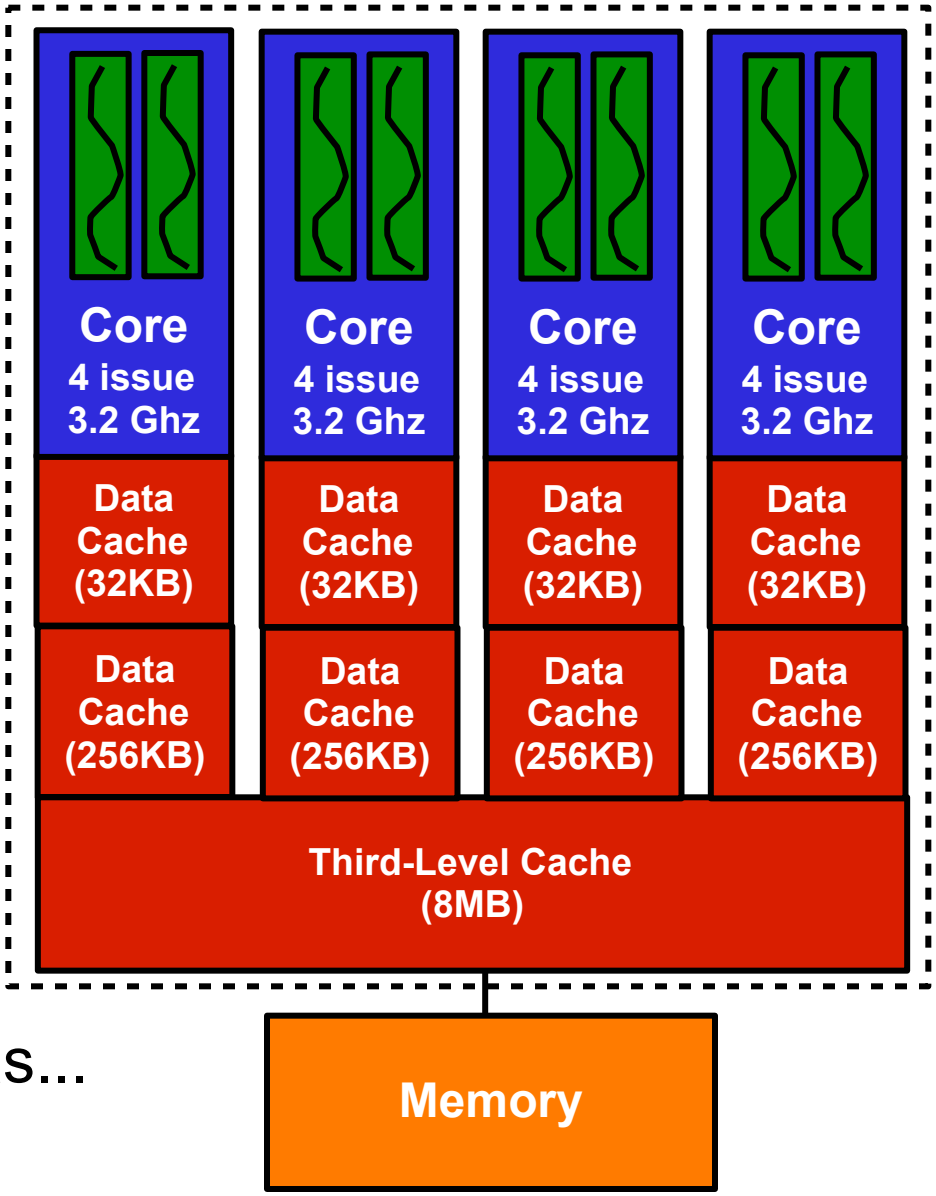
acggrid01 - acggrid32

Dual-Socket Quad-Core "Core 2"



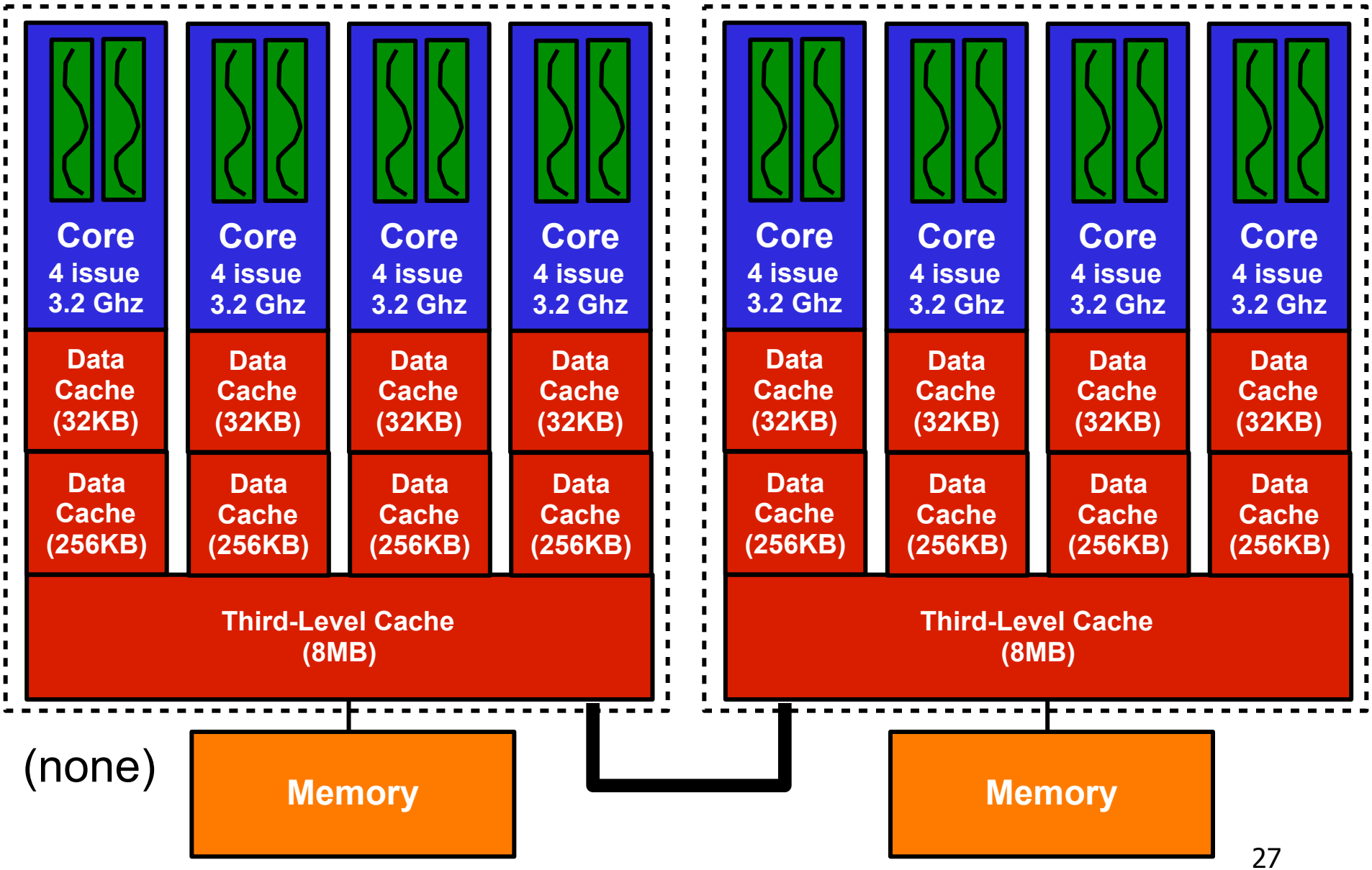
acggrid34

Single-Socket Quad-Core "Core i7"

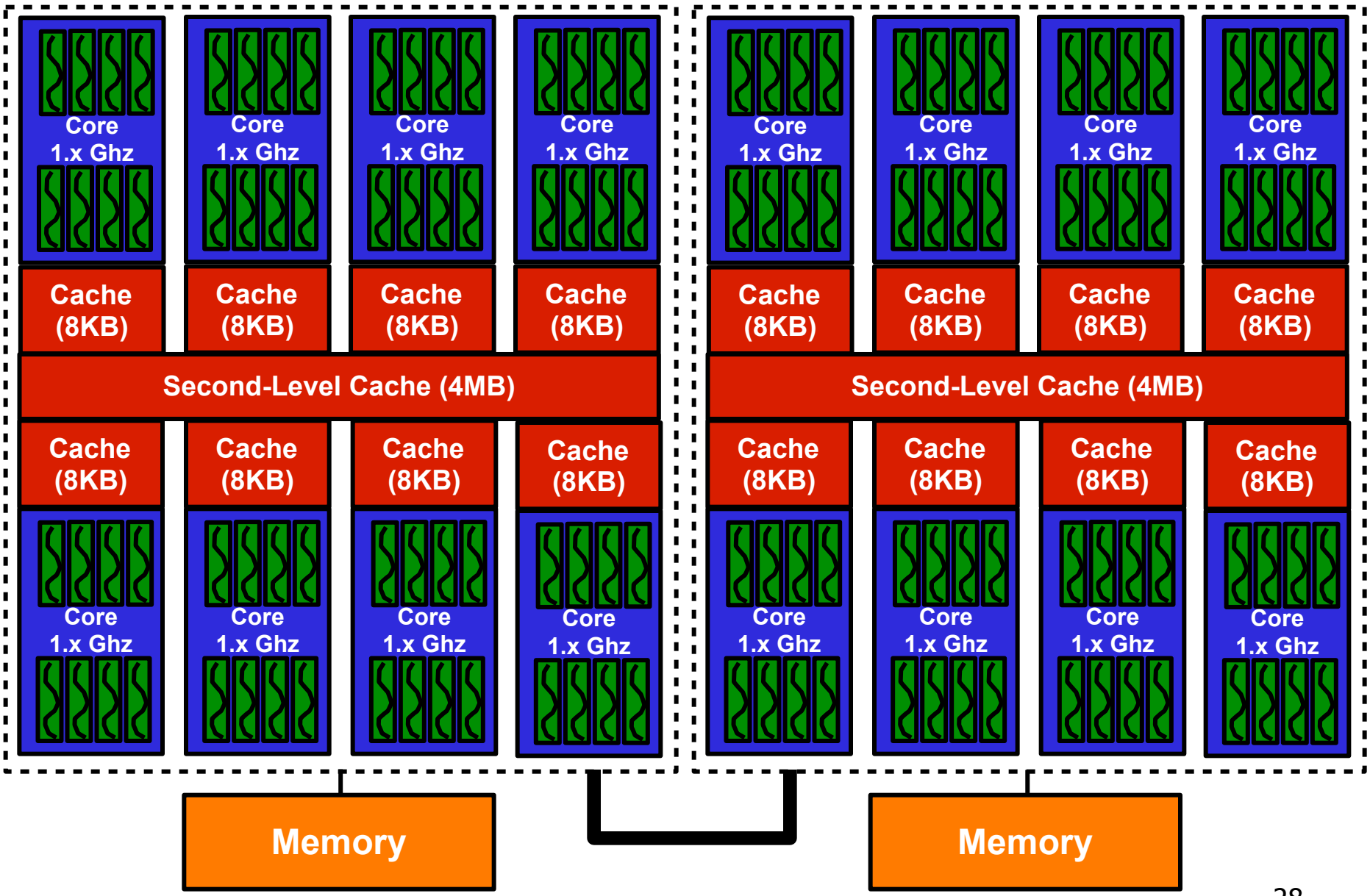


acggrid35.seas...

Dual-Socket Quad-Core "Core i7"



Dual-Socket Oct-Core "Niagara T2"



Cache Coherence

Cache Coherence

- Goal: whenever reading a location in memory...
  - ...receive the value of the last write to that location

Cache Coherence

- Goal: whenever reading a location in memory...
  - ...receive the value of the last write to that location
- Invariant: Single writer --or-- one or more readers

Cache Coherence

- Goal: whenever reading a location in memory...
  - ...receive the value of the last write to that location
- Invariant: Single writer --or-- one or more readers
- Implementation
  - Each cache tracks the "state" of each cache block
    - Modified: read/write state Shared: read-only
    - Invalid: no reads or writes
  - Must transition to "modified" to be able to write block
    - Must "invalidate" all Shared copies before writing block

Cache Coherence

- Goal: whenever reading a location in memory...
  - ...receive the value of the last write to that location
- Invariant: Single writer --or-- one or more readers
- Implementation
  - Each cache tracks the "state" of each cache block
    - Modified: read/write state Shared: read-only
    - Invalid: no reads or writes
  - Must transition to "modified" to be able to write block
    - Must "invalidate" all Shared copies before writing block
- Performance implication: sharing is slow
  - Slower off-chip sharing than on-chip sharing

Data Sharing Penalties

Data Sharing Penalties

- Read-only sharing
  - Fine!

Data Sharing Penalties

- Read-only sharing
  - Fine!
- True (read/write) sharing
  - Write a block invalidates all readers... so reader miss on next read
  - Next writer may stall waiting for invalidations to complete
  - **Take away: avoid read/write sharing of data**

Data Sharing Penalties

- Read-only sharing
  - Fine!
- True (read/write) sharing
  - Write a block invalidates all readers... so reader miss on next read
  - Next writer may stall waiting for invalidations to complete
  - **Take away: avoid read/write sharing of data**
- False sharing
  - If two variables fall in the same cache block, block may ping-pong
  - Example: Each core increments "own" counter in an array
    - Repeatedly fetch and invalidate block -> **extremely slow**
  - Possible solutions:
    - Manually pad data to ensure they are in different cache blocks
    - malloc() could align each object to 64B boundaries

Multicore Memory System Implications

Multicore Memory System Implications

- Positive
  - Multithreading hides lots of miss latency!

Multicore Memory System Implications

- Positive
  - Multithreading hides lots of miss latency!
- Negative
  - **Higher miss rates**
    - More threads all trying to use limited shared cache capacity
      - Impact depends on data locality among threads
    - Sharing misses
  - **High miss latencies**
    - Sharing misses are often slower
    - Perhaps missing to other socket's memory
  - **More memory bandwidth bottlenecks**
    - Many parallel misses trying to use limited memory bandwidth
    - Adding memory bandwidth is expensive, so big concern

Next Time

Next Time

- Review of processes and threads.

Next Time

- Review of processes and threads.
- Review of synchronization, locks and barriers
  - Shared memory/multithreaded coding
  - Synchronization primitives