Parallelism

Introduction to Computer Architecture

- Amdahl’s law
- Flynn’s taxonomy
  - SIMD, MIMD, SISD, MISD
- Explicit vs. Implicit parallelism
  - ILP, DLP, TLP
  - Shared Memory
A fundamental law: Amdahl's law

The problem with serial code

- How much does that little tiny almost-insignificant non-parallel part hurt us?

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.
The problem with serial code

- How much does that little tiny almost-insignificant non-parallel part hurt us?
- Example:
  - I have a program that is 75% parallel, 25% serial
  - I have 100,000 processors
  - How much faster is my program?
  - Well, 75% goes 100,000x faster (e.g., nearly infinitely faster)
  - So 75% of the time → 0% of the time
  - But I still have that other 25% that doesn’t go any faster
  - Final speed: 75% → 0%, 25% → 25%
    overall: it takes 25% of the time, or 4x faster

Q: If 90% of a program can run in parallel and I have 100,000 processors, how much faster can my program go?
  - 10x
  - 100x
  - 100,000x

A: 10x
The 90% will go 100,000x faster (or virtually infinitely fast) so you will be left with the remaining 10%. That’s 1/10th leftover, so your program is 10x faster. Kind of disappointing, really.

Amdahl's law

- Parallel performance is limited by the serial part of your code

\[
\text{Speedup} = \frac{1}{(1 - P) + \frac{P}{S}}
\]

- Example:
  - P=75%
  - S=100,000
  - \(\frac{1}{(25\% + 75\%)/100,000}\)
  - = \left(\frac{1}{25\%}\right) = 4x faster

90% parallel at 8 cores = 4.7x

99% Parallel (91x)
90% Parallel (10x)
75% Parallel (4x)

Limits of Amdahl’s Law
Parallel Architectures & Models

Parallel Architectures: Flynn’s Taxonomy

<table>
<thead>
<tr>
<th>Data stream</th>
<th>Instruction stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>Single</td>
</tr>
<tr>
<td></td>
<td>SISD</td>
</tr>
<tr>
<td>Multiple</td>
<td>Multiple</td>
</tr>
<tr>
<td></td>
<td>SIMD</td>
</tr>
<tr>
<td></td>
<td>MISD</td>
</tr>
<tr>
<td></td>
<td>MIMD</td>
</tr>
</tbody>
</table>
Parallel Architectures: Flynn’s Taxonomy

- **SISD**
  - Single Instruction, Single Data
  - Data and program flow in a sequential manner

- **MISD**
  - Multiple Instruction, Single Data
  - Different instructions can be executed simultaneously

- **SIMD**
  - Single Instruction, Multiple Data
  - Same instruction applied to multiple data elements

- **MIMD**
  - Multiple Instruction, Multiple Data
  - Independent execution of multiple instructions on multiple data elements

- **SPMD**
  - Single Program, Multiple Data
  - Same program executed on multiple data elements

- **MPMD**
  - Multiple Programs, Multiple Data
  - Execution of multiple programs on multiple data elements

A newer categorization more software-oriented

- **ILP**: Instruction-Level Parallelism
  - Serial Program
  - Implicit by the HW or the compiler, but NOT EXPLICIT by the programmer

- **DLP**: Data-Level Parallelism
  - Parallel operations on data
  - Can be explicit or implicit (compiler --- vectorization)

- **TLP**: Task-level Parallelism
  - Parallel Program
  - Explicit by the programmer (tasks, threads, etc.)
  - Implicit: automatic (& speculative) parallelization of serial programs
Instruction-level Parallelism

Example: static dual-issue

• Clever insight: 1/3 of all instructions are ld/st. Can we do them in parallel?
• What do we need to do ld/st in parallel with other operations?
  – Add a second ALU
  – More register file ports
  – Fetch two instructions per cycle
• The compiler puts instructions together in pairs, and we execute them
  – Dual-issue pipeline
• Static multiple-issue: instruction pairs are static (fixed) and don’t change

1: add r1, r2, r3
2: ld r4, r5
3: sub r7, r1, r4
4: or r5, r8, r9
5: st r8, r9

1: add r1, r2, r3
1: ld r4, r5
2: sub r7, r1, r4
2: st r8, r9
3: or r5, r8, r9
3: nop
Dual-issue pipeline

- Regular Path
- Ld/St Path

- Added:
  - More RF ports
  - 2nd instruction fetch
  - 2nd sign-extension
  - 2nd ALU
  - More forwarding logic and paths

- Now we can issue both a ld/st and any other instruction at the same time!

1:  
   add r1, r2, r3
1:  
   ld r4, r5
2:  
   sub r7, r1, r4
2:  
   st r8, r9
3:  
   or r5, r8, r9
3:  
   nop

Parallelism in the pipeline

In the best case we now have a IPC of 2.0 (or CPI of 0.5). With a dual-issue design we can be up to 2x faster.
Static Instruction level parallelism (ILP): VLIW

- Find independent instructions and execute them in parallel in long instruction “bundles”
  
  1: add r1, r2, r3
  2: ld r4, r5
  3: sub r7, r1, r4
  4: or r5, r8, r9
  5: st r8, r9

  1: add r1, r2, r3
  2: sub r7, r1, r4
  3: or r5, r8, r9
  4: st r8, r9

- Create VERY LONG INSTRUCTION WORDS (e.g., 3-16 instructions)
- Need more hardware
  - Extra ALUs (add, ld, st / sub, or at same time)
  - Extra register file ports (read r2, r3, r5, r9 and r1, r4, r8, r9, write r1, r4, r8 and r7, r5)
  - Extra control logic (control ALUs, memory, detect hazards, forward)
  - Read multiple instructions each cycle
- But, we now finish in 2 cycles instead of 5!

Superscalar: dynamic ILP

- What if we make things WAY more complicated:
  - Have the processor look ahead at the next 100 instructions and figure out which ones are independent
  - Don’t even have to do them in order! (out-of-order execution)
- What do we need?
  - Somewhere to store instructions that are waiting (reservation stations)
  - Something to schedule them
  - A buffer at the end to finish them in-order (remember we still have to do what the ISA promises even if we execute out-of-order)
Intel Haswell (2013)

- **Out-of-order superscalar**
- This is incredibly complex
- Take the advanced computer architecture class to learn more!

ILP summary

- Instruction Level Parallelism (ILP)
  - Find independent instructions that we can execute in parallel
  - Need extra hardware to do them in parallel
  - Compiler or hardware needs to determine dependencies

- Complex, messy, and very important!
  - ~2-3x performance gain from superscalar and out-of-order
  - >3-4x power cost
  - Power efficient processors are usually in-order
Data-level Parallelism

DLP = SIMD

• SIMD architectures can exploit significant data-level parallelism for:
  – matrix-oriented scientific computing
  – media-oriented image and sound processors

• SIMD is more energy efficient than MIMD
  – Only needs to fetch one instruction per data operation
  – Makes SIMD attractive for personal mobile devices

• SIMD allows programmer to continue to think sequentially
DLP/SIMD

- Vector architectures
- SIMD extensions
  - MMX: Multimedia Extensions (1996)
  - SSE: Streaming SIMD Extensions
- Graphics Processor Units (GPUs)

Vector

- Basic idea:
  - Read sets of data elements into “vector registers”
  - Operate on those registers (ADD, SUB, MUL, DIV)
  - Disperse the results back into memory
- Registers are controlled by compiler
  - Register files act as compiler controlled buffers
  - Used to hide memory latency
  - Leverage memory bandwidth
- Vector loads/stores deeply pipelined
  - pay for memory latency once per vector ld/st!
- Regular loads/stores
  - pay for memory latency for each vector element
### Vector Example: VMIPS

- **Vector registers**
  - Each register holds a 64-element, 64 bits/element vector
  - Register file has 16 read ports and 8 write ports
- **Vector functional units**
  - Fully pipelined
  - Data and control hazards are detected
- **Vector load-store unit**
  - Fully pipelined
  - Words move between registers
  - One word per clock cycle after initial latency
- **Scalar registers**
  - 32 general-purpose registers
  - 32 floating-point registers

### Vectorizing compilers

<table>
<thead>
<tr>
<th># C code</th>
<th># Scalar Code</th>
<th># Vector Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>for (i=0; i&lt;64; i++)</code>&lt;br&gt;<code>C[i] = A[i] + B[i];</code></td>
<td><code>LI R4, 64</code>&lt;br&gt;<code>loop:</code>&lt;br&gt;<code>L.D F0, 0(R1)</code>&lt;br&gt;<code>L.D F2, 0(R2)</code>&lt;br&gt;<code>ADD.D F4, F2, F0</code>&lt;br&gt;<code>S.D F4, 0(R3)</code>&lt;br&gt;<code>DADDIU R1, 8</code>&lt;br&gt;<code>DADDIU R2, 8</code>&lt;br&gt;<code>DADDIU R3, 8</code>&lt;br&gt;<code>DSUBIU R4, 1</code>&lt;br&gt;<code>BNEZ R4, loop</code></td>
<td><code>LI VLR, 64</code>&lt;br&gt;<code>LV V1, R1</code>&lt;br&gt;<code>LV V2, R2</code>&lt;br&gt;<code>ADDV.D V3, V1, V2</code>&lt;br&gt;<code>SV V3, R3</code></td>
</tr>
</tbody>
</table>
### SIMD vs. Vector

- Multimedia SIMD extensions **fix the number of operands** in the opcode (e.g., 8, 16)
  - Vector architectures have a VLR to specify the number of operands (e.g., \(N=1..128\))
- Multimedia SIMD extensions: **No sophisticated addressing modes** (strided, scatter-gather)
- **No mask registers**
- These features enable vector compiler to vectorize a larger set of applications
- Make it harder for compiler to generate SIMD code and make programming in SIMD assembly harder

### SIMD Example

- **Implementations:**
  - Intel MMX (1996)
    - Repurpose 64-bit floating point registers
    - Eight 8-bit integer ops or four 16-bit integer ops
  - Streaming SIMD Extensions (SSE) (1999)
    - Separate 128-bit registers
    - Eight 16-bit ops, Four 32-bit ops or two 64-bit ops
    - Single precision floating point arithmetic
  - Double-precision floating point in
  - Advanced Vector Extensions (2010)
    - Four 64-bit integer/fp ops
- **Operands must be consecutive and aligned memory locations**
GPUs vs. Vector

- Multiprocessor
- Multithreaded SIMD units
- Instructions are SIMD
- Threads execute the same instructions (only one fetch unit for all threads) but not in lockstep → SPMD Single Program Multiple Data
- Multiple functional units as opposed to deeply pipelined fewer functional units of Vector processor!

Figure 4.22 A vector processor with four lanes on the left and a multithreaded SIMD Processor of a GPU with four SIMD Lanes on the right. (GPUs typically have 8 to 16 SIMD Lanes.) The control processor supplies scalar operands for scalar-vector operations, increments addressing for unit and non-unit stride accesses to memory, and performs other accounting-type operations. Peak memory performance only occurs in a GPU when the Address Coalescing unit can discover localized addressing. Similarly, peak computational performance occurs when all internal mask bits are set identically. Note that the SIMD Processor has one PC per SIMD thread to help with multithreading.
Task-Level Parallelism

Back to Flynn for TLP

- Parallel Computers
  - SIMD
  - MIMD
    - Shared Memory
      - SMP
      - NUMA
      - DSM
    - Distributed Memory
      - Scalable
      - Linear
      - Fixed
Shared Memory

- Multicores are just an SMP Shared Memory MIMD

Distributed Memory

- Typically message-passing machines
- The memories are local to the nodes
- MPI: Message-Passing Interface (send-receive)
- RDMA: Remote DMA
- All Top 500 Supercomputers
Shared Memory Programming

- All Tasks (threads) “see” the same memory
- MIMD
  - Multiple instruction streams (Tasks, Threads), all independent, accessing any memory location
- Large variety or programming models:
  - Pthreads, OpenMP, Parallel C, Intel TBB, StarPU, ...
- Problems:
  - How to synchronize?
  - How to guarantee atomic updated of shared data structures?
  - How about cached data?

Synchronization example

- Example: your bank account has 510 EUR in it.
  - You have ordered online for 500 EUR
  - You withdraw 200 EUR
  - What happens when they each run on a different processor at the bank?

```plaintext
Processor 1

Online order
load bal
if (bal>=500)
  store bal-500;
else
  fail

Bank
bal
510

Processor 2

Withdrawal
load bal
if (bal>=200)
  store bal-200;
else
  fail
```
If they arrive one after another:

- You have **ordered online** for 500 EUR
- You withdraw 200 EUR

If they arrive at the same time:

- You have **ordered online** for 500 EUR
- You withdraw 200 EUR

This is NOT what you expect.
Protecting data with locks

- A lock is a variable that protects data
  - You have to acquire (get) the lock before you access the data
  - While you have the lock, no one else is allowed to access it
  - When you are done you release (give up) the lock
  - Now others can get it

```c
Online order
if (lock(bal_lk)) {
    load bal
    if (bal>=500)
        store bal-500;
    else
        fail
} unlock(bal_lk)
```

```c
Withdrawal
if (lock(bal_lk)) {
    load bal
    if (bal>=200)
        store bal-200;
    else
        fail
} unlock(bal_lk)
```

Cache coherency problem

- Multiple processors each have their own (private) cache
  - When they bring data into the cache they get the latest value from DRAM
  - How do they know other processors don’t have newer versions in their caches?

- Coherency: need to keep track of who has the latest data so we always get the most recent version

- One solution: snooping
  - Each cache listens in on what the others are doing (connect address wires)
  - If it sees another processor write to a line it has, then it invalidates that line
  - If the line is dirty it needs to write it back so the other cache loads it
Summary

• Amdahl’s law
  – You cannot escape Amdahl’s law: serial sections will kill the speedup

• Flynn’s taxonomy
  – SIMD → Vector, SIMD extensions, GPUs
  – MIMD → General purpose parallel architectures: multicores, SMP, ...

• Explicit vs. Implicit parallelism
  – ILP: implicit HW or compiler
  – DLP: explicit by the programmer or implicit by vectorizing compiler
  – TLP: explicit parallel programming
  – Shared Memory
    • Synchronization and Coherence are important problems