• More about machine-dependent optimizations
• cache memory, data locality, pipelines
• How to make your program run faster on modern hardware

Elias Rudberg, room P2440
elias.rudberg@it.uu.se
Quick summary – important hardware-related concepts

- Pipelining --> independent instructions are good
- Cache memory --> data locality is good
Caches

- Several levels of cache with varying latency
- L1 (approx. 10-100 kB) 1-5 cycles
- L2 (256 kB-) 10-20 cycles
- L3 (> 2 MB), slower
- Caches organized into "cache lines", approx. 128 bytes each
Cache optimization – how?

Try to increase the chances that the needed memory locations are already in cache:

- Nearby memory accesses in time should be to nearby locations in memory
Cache

CPU

address

address

Cache

hit

data (a word)

Memory
Example of Memory Hierarchy

**L0:** registers
- CPU registers hold words retrieved from L1 cache.

**L1:** on-chip L1 cache (SRAM)
- L1 cache holds cache lines retrieved from the L2 cache memory.

**L2:** off-chip L2 cache (SRAM)
- L2 cache holds cache lines retrieved from main memory.

**L3:** main memory (DRAM)
- Main memory holds disk blocks retrieved from local disks.

**L4:** local secondary storage (local disks)
- Local disks hold files retrieved from disks on remote network servers.

**L5:** remote secondary storage (distributed file systems, Web servers)
- Smaller, faster, and costlier (per byte) storage devices
- Larger, slower, and cheaper (per byte) storage devices
General Caching Concepts

- “Cache hit”
- “Cache miss”
- Cache Performance Metrics: "hit rate" or "miss rate"
Caches and performance

- Caches are extremely important for performance
- Level 1 latency is usually 1 or 2 cycles
- Work well for problems with nice locality properties
- Caching can be used in other areas as well, example: web-caching (proxies)
- Modern CPUs have two or three levels of cache
- Most of the chip area is used for caches
How caches work -- cache lines

- Cache divided into **cache lines**
- When memory is read into cache, a whole cache line is always read at the same time
- Good if we have data locality: nearby memory accesses will be fast
- Typical cache line size: 64-128 bytes
How do caches work?

- **"Direct-mapped cache"**: Hash function maps to a single place for every unique address.
- **"Fully associative cache"**: Cache space is divided into n sets. Address is distributed modulo n. k-way set associative (k = 2...24).
- **"Set associative cache"**: Maps to any slot. Hard to build in practice.
Caches in hierarchies

To synchronize data in hierarchies caches can either be:

- **Write-through**
  - Reflect change immediately
  - L1 is often write-through

- **Write-back**
  - Synchronize all data at a given signal
  - Less traffic, but other drawbacks instead
SSE SIMD instructions

• Single instruction, multiple data (SIMD)
• Streaming SIMD Extensions (SSE)
  --> “SSE SIMD” instructions
• Allows increased performance by doing the same operation for several data entries at once.
• As with pipelining, requires independent instructions.
• Tricky to use in your own code. Intrinsics. The compiler can (hopefully) do this for us
• Heavily used in optimized linear algebra libraries
How to get more independent instructions?

• Loop unrolling – rewrite loop to do more work in each loop iteration

• Loop fusion – turn two (or more) loops into one
Loop unrolling

- Loop unrolling – rewrite loop to do more work in each loop iteration

```c
for(i = 0; i < N; i++) {
    // do something related to i
}

-->

for(i = 0; i < N; i+=3) {
    // do something related to i
    // do something related to i+1
    // do something related to i+2
}
// Afterwards: take care of remaining part, if any
// (needed in case N does not divide evenly by 3)
```
Loop fusion – turn two (or more) loops into one

```c
for(i = 0; i < N; i++) {
    // do “work A” related to i
}
for(i = 0; i < N; i++) {
    // do “work B” related to i
}
```

-->

```c
for(i = 0; i < N; i++) {
    // do “work A” related to i
    // do “work B” related to i
}
```
Theoretical Peak Performance

- Assumptions:
  1. Functional units produce a result every clock cycle
  2. No cache miss penalties
  3. Infinite memory bandwidth
  4. Enough independent instructions to use all units

- Theoretical Peak Performance
  Peak = Clock_freq*No_functional_units
  Ex, 2 FP Units, 3GHz --> Peak = 3*10^9 * 2 = 6 GFlops


Modern computers, summary

- Practically all modern computers are LOAD/STORE architectures
- Pipelining heavily used
- They have a two, or three level memory hierarchy with large on-chip caches
- Clock frequencies are measured in GHz
- Most CPU:s are at least 4-way superscalar which, as an example, could mean that it can execute one LOAD/STORE, one integer ALU, and two FP instructions per clock cycle
Problems in modern computers

- Hard to find independent instructions
- Caches only work if programs exhibit good spatial or temporal locality
- Superscalar pipelined microprocessors require very high memory bandwidth
- The CPU-Memory gap is increasing
- Caches get more important
- Compilers get better and better but cannot keep up with the innovations in hardware
Compilers are improving

- Example: “restrict test” code performs differently depending on gcc version
- Tested at the Kalkyl computer at UPPMAX
- Version 4.4.6 of gcc (from 2011) gives no speedup from using restrict in this test case
- Version 4.5.3 of gcc gives nice speedup
Recap of optimization stuff

- Remember: care about correctness and flexibility before performance
Priorities:

- 1: Correctness
- 2: Flexibility
- 3: Performance

Priorities: correctness comes first!
Do profiling first!

• Always measure performance of different parts of the code before doing optimizations

Remember:

*Premature optimization is the root of all evil*
Always test if attempted optimizations give improvements

- Is there any real benefit? TEST!
  - If results almost unchanged, go for the simpler version
That's all!

- Questions?