Hardware – an overview

Simons Programming Preschool

Emanuel Gull, University of Michigan
Why Hardware?

- Know your tools
- Understanding hardware is prerequisite for writing fast / efficient codes
- Leave the details up to the engineers, understand the broad concepts
Why Hardware?

Moore's law: exponential speedup. Algorithms: Steeper exponential

D.P. Landau, UGA

• relative performance
• computer speed

• 1E10
• 1E9
• 1E8
• 1E7
• 1E6
• 1E5
• 1E4
• 1E3
• 1E2
• 1E1
• 1E0
• 1

• 1970
• 1975
• 1980
• 1985
• 1990
• 1995
• 2000
Basic Setup

• We write our code in a high-level language. The **compiler** translates the program to **machine language**. The computer executes the machine language program.

• Need to understand hardware limitations and what the compiler does.
The CPU (central processing unit)

- Memory controller
  - Manages data loading from/to memory
- Registers
  - Store integer or floating point numbers
  - Store constants
  - Values can be loaded from or stored into memory
- Arithmetic and logical units (ALU)
  - Performs arithmetic operations and comparisons
  - Operates on values in the registers (very fast)
- Fetch and decode unit
  - Fetches the next instruction from memory
  - Interprets the numerical value of the instruction and decides what to do
  - Dispatches operations to ALU and memory controller to perform the operation
- Modern CPUs are more complex (see later)
Machine code and assembly language

- The CPU performs instructions read from memory
  - Instructions are given in machine code
  - These are just numbers which are interpreted as instructions
  - Ugly and nearly impossible to interpret
- Assembly language
  - Is a one-to-one translation from machine code to a readable text form
  - Is non-portable: differs depending on CPU-type
- Typical instructions
  - Load values into registers
  - Load data from memory into register or store registers into memory
  - Perform arithmetic and logical instructions on registers
  - Jump (branch) to another instruction

```assembly
subq    $32, %rsp
movq    %rdi, -24(%rbp)
movl    %esi, -28(%rbp)
movl    %edx, -32(%rbp)
movq    -24(%rbp), %rax
leaq    24(%rax), %rcx
movl    -32(%rbp), %edx
movl    -28(%rbp), %eax
movl    %eax, %esi
movq    %rcx, %rdi
call    __ZNK10SpinConfig14energySpinFlipEii
negl    %eax
pxor    %xmm2, %xmm2
cvtsi2sd        %eax, %xmm2
movd    %xmm2, %rax
movq    -24(%rbp), %rdx
movq    8(%rdx), %rdx
movd    %rax, %xmm1
movd    %rdx, %xmm3
divsd   %xmm3, %xmm1
movd    %xmm1, %rax
movd    %rax, %xmm0
call    _exp
```
Types of CPUs

- CISC
  - Complex instruction set: many instructions (assembly language commands).
  - x86
- RISC
  - Reduced instruction set: few instructions (but faster / can be pipelined)
  - PowerPC
- GPU
  - very few instructions, fairly low level, fairly specialized, but very good performance for dedicated tasks
  - Tesla, Fermi, Kepler, Maxwell, Pascal, Volta...
Pipelining

• Typical set of instructions: fetch instruction, decode it, execute it, store results.
• All these commands belong to different physical parts of the chip
• Start loading things before the previous set of instructions is done
• Here: two similar sequences of instruction, done at cycle 5 instead of 8!
• do more of these in parallel?
Pipelining in practice

• Take this loop
  
```c
for (int i=0; i <102400; ++i)
    a[i]=b[i]+c[i];
```

• Consecutive iterations are independent

• unroll operations
  
```c
for (int i=0; i <102400; i+=4){
    a[i]=b[i]+c[i];
    a[i+1]=b[i+1]+c[i+1];
    a[i+2]=b[i+2]+c[i+2];
    a[i+3]=b[i+3]+c[i+3];
}
```

• can now be pipelined

• Unrolling should never be done by hand – leave it up to the compiler
Branch Prediction

- Whenever we have dependence: pipeline stalls
- Branch prediction: continue computing most likely branch, discard results if not needed:

  ```
  for (int i=0; i < 102400; ++i)
    a[i] = b[i] + c[i];
  ```

- works on any modern compiler
Moore’s law

1 exaflop
1 petaflop
1 teraflop
1 gigaflop

Performance Development

1993  2003  2013

linpack performance
top 500 machines
top machine
machine #500
top 500 machines
top machine
machine #500

top500.org
Moore's law
Memory: capacitors vs transistors

- Bits on chips stored in states of transistors: very fast
- Also very expensive: each transistor needs to be etched, takes chip ‘real estate’.
- Most data stored in ‘memory’: capacitor technology, slower, much higher capacity, off-die.
- Two main problems:
  - memory bandwidth (how fast can bits be transported to where they are needed?)
  - memory latency (how long does it take for them to arrive?)
  - How to hide this from user code?
Solution: Caching!

- We can afford to etch some transistors onto the chip…
- Manage data so that we keep any frequently used data close to the chip. The less frequently used the farther away it is stored.
- Different levels of cache depending on how far / speed / latency / core sharing.
Caching has consequences

- If you frequently reuse data (and the compiler knows it) you will be very fast.
- If you randomly access data in large arrays you will run at the speed of the memory (300x slower).
- Use this knowledge to design data structures:
  - use vectors (consecutive data) rather than linked lists
  - use dense matrix linear algebra rather than vector operations
- Let the engineers worry about more advanced techniques: prefetching, etc; use libraries that have been optimized whenever possible.
Virtual Memory

• What if multiple programs run on the same machine, everyone wants memory?
• Programs run in a ‘logical’ address space (value of pointers); hardware/os map logical address space into ‘physical’ address space (on which dimm, on which chip, etc).
• Index of mappings between physical and logical memory is kept in special memory buffer: ‘translation lookaside buffer’ (TLB). Every time you need memory access: TLB lookup
• If you run out of memory, hard drives can be used to provide additional memory.
• Used to be extremely slow (rotating disks with large latencies). Now faster, but still extremely slow, if written to SSD.
• Audiovisual feedback when this happens: machine gets slow, hard drive starts stuttering and disk lights light up!
Virtual Memory Logic

- Virtual memory is organized in ‘pages’. One page is 4K. Always one page at a time is read/written from/to disk.
- Index of pages is kept in page table.
- Locality of data matters: if data is on the same page only one read from disk needed, otherwise more disk access.
- Find algorithms that maximize data locality!
Virtual Memory – The Worst Case

- Request an address
  - Cache miss in L1
  - Cache miss in L2
  - Cache miss in L3
- Lookup physical address
  - Cache miss in TLB
  - Request page table entry
  - Load page table from memory (slow)
- Page fault in page table
  - Store a page to disk (extremely slow)
  - Create and initialize a new page (very slow)
  - Load page from disk (extremely slow)
  - Load value from memory (slow)

- Try to reuse data as much as possible
Where does that leave you?

• **First rule of optimization: do not optimize**

• However, sometimes you have a choice:
  • Algorithm with data locality vs other algorithm: almost always local data wins
  • Nothing of what *you* program runs at peak flop. But some of the dense linear algebra routines almost do. Use them if you have the choice
  • Redundant calculations may not be a bad thing, if they allow you to shift memory operations to FLOPs
  • Memory and flops are diverging.
  • Allow your compiler to optimize for you: minimal, clear dependencies, keep things simple!