CPU

- Optimized to process a single sequence of instructions
- Extremely fast – clock speed reflects speed of operations
  - Not getting a lot faster
    - memory latency
    - power
    - sequential processing
- We’re still very good at adding transistors
- Current trend: instead of doing something faster, do more things

Limitations on CPU computing

• Memory wall
  • CPU clock is much faster than memory latency
  • Main memory is stored outside the CPU
  • Bottleneck is bandwidth between the two

• Power wall
  • Moore’s law: transistor density increases over time
  • Dennard scaling: as transistors get smaller, so do power requirements
  • Current leakage: tradeoffs aren’t equal – power per unit area is increasing

• Instruction Level Parallelism wall
Instruction Level Parallelism

• Instruction level parallelism (ILP) wall
  • Speed increases in CPUs rely on finding parallelism within a single thread

---

**pipelining**

execution over time

program queue

fetch decode execute write

pipeline

completed operations

**speculative execution**

1) int sign;
2) if(x == NaN)
   3) sign = x;
4) else if(x == 0)
   5) sign = 0;
6) else
   7) sign = x/abs(x);

• (7) requires an arithmetic operation and is most likely to require an arithmetic operation and is most likely
• Execute (7) while (2) and (4) are tested

**out-of-order execution**

1) e = a + b;
2) f = c + d;
3) m = e * f;
   • (3) is the only line with a dependence
   • (1) and (2) can be executed in parallel
Speedup

• Dependencies limit the amount of ILP in any program
• Ultimate goal is to achieve *speedup*
  • reduce the processing time for the same problem

• Given a process that requires time $T_1$
  • calculate the job in a shorter time $T_2 < T_1$
  • linear speedup (generally ideal) would allow $T_2 \approx \frac{T}{P}$
    • where $P$ is the number of processors
  • superlinear speedup \( T_2 < \frac{T_1}{P} \) is possible
    • usually memory fetch gains: increased cache hits, cache coherence
Parallelism in High Performance Computing

• Two ways to achieve true parallelism to solve a problem

Given $P$ processors:

• *task parallelism*: break the problem up into $T \geq P$ tasks and pass them each off to a processor

• *data parallelism*: break your input/output data into $D \geq P$ subsets and launch one thread $T = D$ for each piece of data
Task Parallelism

• Divide your problem up into $T$ tasks
  • assign the first $P$ tasks to a processor
  • when any processor finishes a task $T_n$, move on to task $T_{P+1}$
  • repeat until all tasks are completed

• Tasks are ideally independent – the result of one does not change based on the result of another
  • when one task finished, does it matter which one you start? (ideally NO)
    • if so, can you arrange for them to run in the correct order (ideally YES)
    • if not, task $T_n$ can either
      • wait for it’s dependent task to complete (you lose the processor during this time)
      • copy the current state of the task into main memory and start another process
      • how would you make this decision?
      • whenever it takes more time to store (+ later load) the task state to continue processing
Task Parallelism – Current Use

• this is the multi-core CPU model: you can play Hearthstone and watch Netflix while your homework runs in the background

• this has generally been the primary model for cluster computing and supercomputing
Data Parallelism

• Partition your data
  • send the first $P$ threads on different processors
  • once any thread $T_n$ completes, launch another thread
  • repeat until all threads have completed

• single instruction multiple data (SIMD)
  • all cores execute the same instruction
  • different data can be used (register, memory address, etc.)

• Advantages
  • your processors can be slower, simpler, smaller – you can have more of them
  • data can often be stored close to the processor
SIMD principles

Assume $N = 5000$ array elements

- **Program A**
  1) for($i = 0; i < N; i$){
  2)    $C[i] = A[i] + B[i];$
  3) }

  - Assume 1 copy/add takes 1 clock cycle. How much time is required for N processors?
    $\approx 1$ clock cycle (plus any overhead launching the threads, writing results)

- **Program B**
  1) for($i = 0; i < N; i$){
  2)    $c = A[i] + B[i];$
  3)    if( $c > 100$ )
  4)        $A[i] = c;$
  5)    else
  7) }

  - Assume a comparison also takes 1 cycle. How much time is required for N SIMD processors?
    $\approx 4$ clock cycles – all threads have to execute the same instruction
  - If $c > 100$ for half of the threads, the other half do nothing while line (4) executes
Practical considerations

- Not all algorithms work well on GPUs
  - Processors are slower
  - No instruction level parallelism
  - Best with a lot of data parallelism

- Algorithms that work well on GPUs:
  - Large loops that do similar things
  - Regular memory access patterns
  - Examples: linear algebra, convolution and stencil operations

- Algorithms that don’t work well on GPUs:
  - Irregular memory accesses, loops where each iteration is dependent on previous iterations
  - Searching and sorting – particularly of small data sets
  - Graph algorithms
Practical considerations

- **Heterogeneous computing**: Using different types of processors to optimize complex algorithms
- Most modern supercomputers have GPUs connected to clusters of CPUs
  - Blue Waters (NCSA/Illinois supercomputer)
    - 22,640 CPU (XE) nodes with 362,240 cores
    - 4,228 CPU+GPU (XK) nodes, each containing:
      - 8 core CPU
      - 1 nVidia Tesla K20X (Kepler) GPUs
- The goal of this class is to teach you heterogeneous computing
  - Design an algorithm with heterogeneous computing in mind
  - Prototype using a CPU-based implementation
  - Move select pieces to the GPU
  - Optimize both GPU and CPU portions
Heterogeneous Computing Examples

• Calculate the $\ell_1$-norm of the sum of two large vectors:

$$c = \|a + b\|_1$$

the $\ell_1$-norm is defined as:

$$\|x\|_1 = \sum_{i=1}^{N} |x_i|$$

data parallelism:

$$a[i] + b[i] = c[i]$$

sequential operation

$$c_1 + c_2 + \cdots + c_N$$

any options?

CPU serial

binary addition → CPU
Heterogeneous Computing Examples

- Calculate the inverse of the outer product of two vectors:

$$(xx^T)^{-1}$$

$$\begin{pmatrix} x_1 & \cdots & x_N \\ \vdots & \ddots & \vdots \\ x_N & & x_N \end{pmatrix}^{-1}$$

Data parallelism:
- $x_{ij} = x_i x_j$
- Also symmetric
- $x_{ij} = x_{ji} = x_i x_j$

Several dependencies
- Common CPU and supercomputing benchmark
Guiding principles

• For a serial process, the CPU will almost always be faster
  • almost always split these off to a CPU
• For small data, the CPU will almost always be faster
  • reduce data through parallel processing with the GPU
  • hand off final calculations to the GPU
• The CPU compiler is usually smarter
  • good at optimizing code, reducing instruction count
  • cache management
• On the GPU, every instruction is important
  • thread divergence – stalls introduced because some threads require execution of an instruction that others do not
    • problem areas include:
      • if statements
      • loops with variable numbers of iterations
Physical Comparisons

**CPU – i7 6970HQ**
- 14nm lithography
- 4 cores
- $\approx 82\text{mm}^2$
- 2.8Ghz clock
- **Memory:**
  - 64GB main memory
  - 8MB L3
  - 256KB / core L2

**GPU – Tesla P100**
- 16nm lithography
- 56 multiprocessors
  - FP32 cores/SM = 64
  - FP64 cores/SM = 32
  - Theoretical load (cores)
    - 3584 FP32 cores
    - 1792 FP64 cores
- $\approx 610\text{mm}^2$
- 1.3Ghz clock
- **Memory:**
  - 16GB device memory
  - 4MB L2
  - 256KB / SM register file
Computational power is measured in Floating Point Operations per Second (FLOPS)

- Common units are giga-FLOP (GFLOPS) and tera-FLOPS (TFLOPS)
  - Intel Core i7-6700K ≈ 113 GFLOPS (10^9)
  - nVidia GeForce GTX 980 ≈ 5000 TFLOPS (10^{12})
  - nVidia GeForce GTX 1080 ≈ 9000 TFLOPS (10^{12})
  - Blue Waters (NCSA, Illinois) ≈ 13.3 petaFLOPS (10^{15})

Theoretical limitations, assuming full utilization

Practical limits:

- Data transfers from CPU (main memory) to the GPU
- Memory latency – time required to read data for processing
  - Access patterns can take affect caching
    - Cache hit – memory fetch is stored close to the processor in a cache
    - Cache miss – have to wait for a value to be retrieved from main memory
  - GPUs have different types of memory (low and high latency)
- Occupancy – how many processors are being used
  - Thread divergence – some threads are waiting for others
  - Register limits – not enough registers to maximize # of threads
Memory Latency and Bandwidth

• *fetch* – request for a piece of data from memory
• *latency* – length of time between the fetch and receiving the first element of data
• *bandwidth* – how much data can be transferred in a given time

• Intel Skylake
  • memory latency $\approx 42$ clock cycles
  • memory bandwidth $\approx 34$ GB/s

• L2 cache latency $\approx 12$ clock cycles
• L1 cache latency $\approx 4$ clock cycles
Intel i7 CPU

- Core 1
- Core 4
- L3 Cache (shared)
- L1
- L2
- to RAM (off chip)
Memory Latency and Bandwidth

• Trade-offs
  • latency is correlated to distance from the processor
    • shorter distance provides lower latency
  • latency is inversely correlated to physical size
    • larger memory provides higher latency
  • these are related: as memory banks increase in size, individual banks have to be placed further away from the processor

• Shared access
  • All cores have access to the same off-chip RAM and L3 cache
  • Each core has its own L1 and L2 cache
  • So, if each core wanted fast access to the same value, a separate copy would have to be stored in each L1 cache

• These complexities result from optimizing distance + latency
• GPUs choose a different optimization technique
nVidia GPU model

Each GPU Contains
• Global memory bank
• Several Streaming Multiprocessors
  • a bank of shared memory
  • shared memory to all cores on the chip
  • a register file
  • used to allocate registers across cores
  • multiple cores
  • each core can run at least one thread
  • 63 32-bit registers for calculations
  • note: cores cannot access registers allocated to other cores
nVidia GPU model

nVidia GeForce GTX 970
- 4 GB Global Memory
- 13 SMs
  - 128 cores
  - 48 Kb shared memory
  - 256 Kb register file
  - 1.3 GHz clock

nVidia Tesla P100
- 16 GB Global Memory
- 56 SMs
  - 64 cores
  - 48 Kb shared memory
  - 256 Kb register file
  - 1.3 GHz clock

Global Memory
16 GB