# Course outline

Sunday, January 10, 2021 9:27 AM

Instructors:

- Naga Kandasamy
- Shihao Song
- Nosheen Afroz

Description:

- See syllabus document on BBLearn

Pre-requisite knowledge:

- Programming in C
- Basic computer architecture

Programming assignments:

- Approximately 12 assignments
- Can work in team of up to two (not three)

Grade breakdown:

- Programming assignments (100%)

Computing resources:

- xunil-03.coe.drexel.edu
- xunil-05.coe.drexel.edu (has Nvidia 1080 GTX GPU)
- Recommend installing virtualbox on your machine (if you don't have a Linux box)

Piazza site:

Search for teammates

Post coding-related questions to other students

# Limits of single core machines

Sunday, January 10, 2021 7:57 PM

Thermal issues

Wire delays

DRAM access latency

ILP limit

# Thermal issues

Sunday, March 29, 2020 8:08 PM

The main factors contributing to the CPU power consumption are dynamic power consumption, short-circuit power consumption, and power loss due to transistor leakage currents.

$P_{cpu} = P_{dyn} + P_{SC} + P_{leakage}$

$P_{dyn} = \alpha C V^2 f$

- $\alpha$: switching factor ($0 \to 1$, $1 \to 0$)
- $C$: Capacitance
- $V$: Operating voltage
- $f$: Operating frequency

Operate the CPU at lower $f$. Can lower $V$ as well to safely operate at low $V$.

Parallelism is a method of reducing power consumption while improving performance. Example: NVidia 640 GT has 384 cores, each clocked at about 800 MHz, whereas the GTX Titan Z has 5760 cores, each clocked at 700 MHz. The 1080 GTX that we will use for programming assignments has 2560 cores.

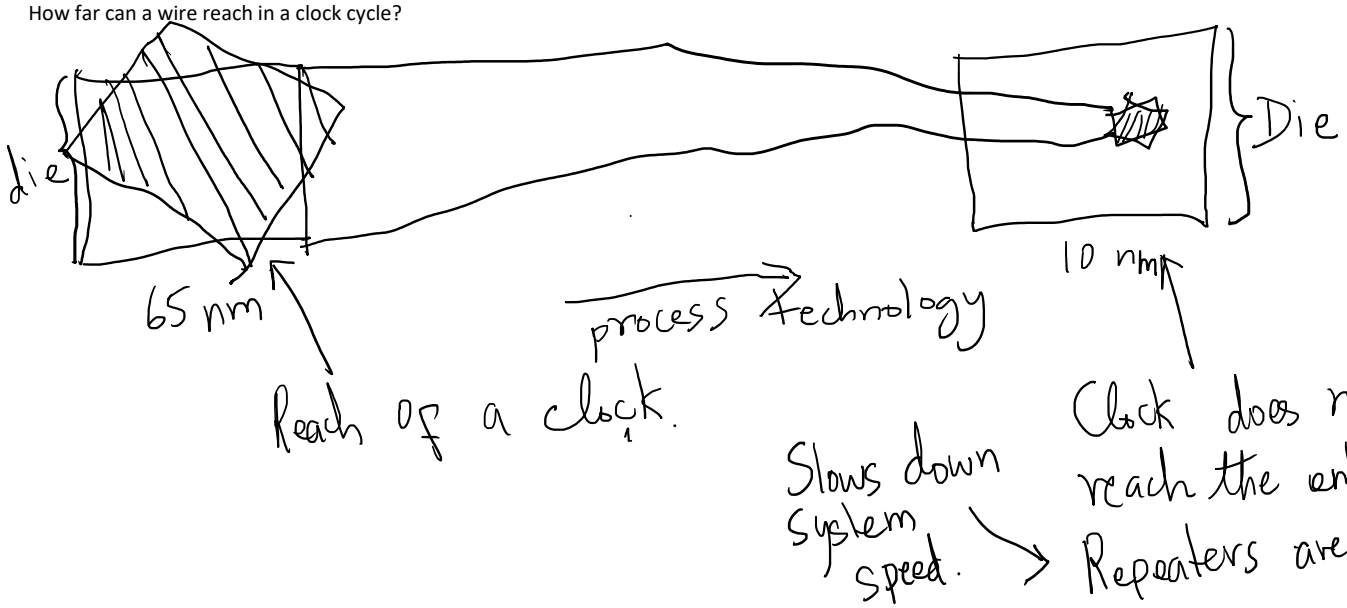
1080 GTX core operating frequency is ~1.6 GHz

The Titan Z core operates at much lower frequency (700 MHz)

# Wire delays

Sunday, March 29, 2020 8:12 PM

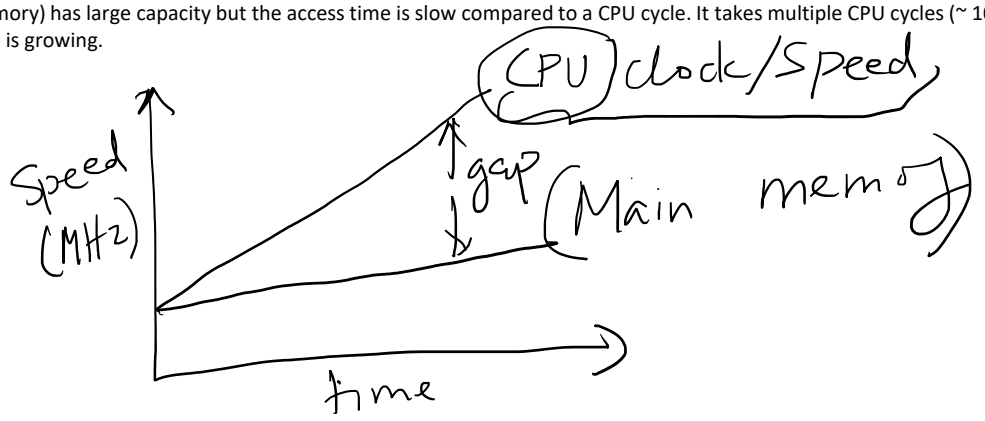
The critical path is affected due to circuit delays, called the propagation delay, as well as delays due to long wires on chips.
How far can a wire reach in a clock cycle?



# DRAM access latency

Sunday, March 29, 2020 8:14 PM

DRAM (main memory) has large capacity but the access time is slow compared to a CPU cycle. It takes multiple CPU cycles (~ 100s cycles) to access an item in main memory. The gap is growing.




A cache hierarchy comprising of more expensive SRAM memory (flip flops) closer to the processor (L1 and L2 caches) is used to speed things up. However, SRAM is getting harder to scale.

# ILP limit

Sunday, March 29, 2020 8:15 PM

Compilers are smart enough to extract ``implicit'' parallelism from serial code. Pipelined hardware aims to provide an ideal CPI of 1. Multi-scalar processors (n-way issue pipelines) aim to get CPI < 1, or IPC > 1. We have squeaked out the last implicit parallelism using 2 -way to 6-way issue, out-of-order issue, branch prediction, speculative execution, etc. We have hit a limit at around 0.5 CPI in terms of extracting implicit parallelism fro m serial code.

MIPS pipeline:

IF ID EX MEM WB

IF: Instruction fetch
ID: Decode
EX: Execute
MEM: Memory
WB: Write back

Goal of pipelining:
Minimize CPI (cycles per instruction)
Ideal CPI = 1

Superscalar pipelines:
Multiple EX units ⇒ are able to issue and execute multiple instructions per cycle

Goal: maximize IPC
IPC: Instructions per cycle

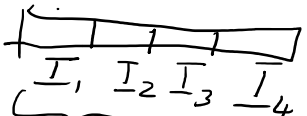
# Implicit parallelism

Monday, March 30, 2020 12:29 PM

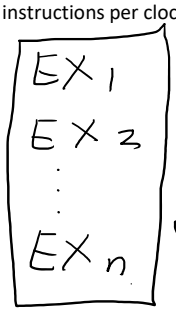
Superscalar processors issue varying numbers of instructions per clock cycle.
Example: superscalar MIPS/RISC V

VLIW (Very large instruction width)

Example:




four instructions




IF ID MEM WB

Includes integer/floating point units
Load/store units

A VLIW packet is fetched each clock cycle.

Statically generated by compiler.

Instructions within a VLIW packet must be independent

↳ How many can compiler identify?

↳ Limit is ~4 instructions

Instructions can be statically scheduled by the compiler and issued in-order by the hardware, or the hardware can dynamically schedule instructions in parallel as data becomes available in an out-of-order fashion (scoreboarding and Tomasulo's method are covered in ECEC 412/621).

Modern compilers and hardware aim to extract instruction-level parallelism (ILP) from serial code.
Goal: reduce cycles per instruction (CPI) on single pipeline or increase instructions per cycle (IPC) on multi-scalar pipelines.

Some examples:

Instruction scheduling
Loop unrolling

Limits of compiler extracted parallelism. The so called "ILP limit."

4 instructions/clock cycle can be issued.

# Instruction scheduling — Assume MIPS/RISC V pipeline

Monday, March 30, 2020 3:08 PM

Consider the MIPS/RISC V pipeline executing the code snippet:

C = A + B
D = E - F

Without instruction scheduling:

```
lw $r1, <addr of A>
lw $r2, <addr of B>
add $r3, $r1, $r2
sw $r3, <addr of C>
```

| Clock cycle: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | IF | ID | EX | MEM | WB | | | | |
| | | IF | ID | EX | MEM | WB | | | |
| | | | IF | ID | stall | EX | MEM | WB | |
| | | | | IF | stall | ID | EX | MEM | WB |

One stall cycle between the lw and dependent add inst.
Even with forwarding enabled.
Similarly, one stall cycle between the lw and dependent sub instruction.

Modern compilers will reorder instruction sequencing in an attempt to eliminate/reduce stalls in pipeline.

Load promotion:

```
a = b + c;
d = e - f;
```

```
lw $r1, <addr of B>
lw $r2, <addr of C>
lw $r3, <addr of e>
lw $r4, <addr of f>
add $r5, $r1, $r2 # Calculate a
sub $r6, $r3, $r4 # Calculate d
sw $r5, <addr of a>
sw $r6, <addr of d>
```

no stalls

forwarding is enabled in the pipeline.

Trade off:
Increased register usage in this example.

Further reading: see my notes on computer architecture on BBLearn

# Loop unrolling

Monday, March 30, 2020 3:17 PM

SAXPY: Single precision A X plus Y.

The SAXPY loop: Y = aX + Y where X and Y are vectors of length N and a is a constant.

```
for (int i = 0; i < N; i++)
    y[i] = a*x[i] + y[i]
```

Tight loop
Not enough instruction within the loop body for reordering

Compiler will unroll the loop to expose more instructions within loop body
→ More flexibility/opportunity for instruction reordering.

Unroll loop twice

```
for (i=0; i<N; i=i+2){
    y[i] = a X[i] + y[i];
    y[i+1] = aX[i+1] + y[i+1];
}
```

Unrolling four times

```
for (i=0; i<N; i+=4){
    y[i] = aX[i] + y[i];
    y[i+1] = aX[i+1] + y[i+1];
    y[i+2] = aX[i+2] + y[i+2];
    y[i+3] = aX[i+3] + y[i+3];
}
```

Tradeoff: loop unrolling results in more pressure on the register file due to increased register usage

# Data and control hazards

Monday, April 6, 2020 12:13 PM

Data hazards:

```
a = b + c
d = a + e
```

Read after write (RAW) dependency

Mitigated using a combination of forwarding/bypassing within the pipeline. Instruction scheduling (by the compiler or by pipeline)

Control hazards:

```
If (cond) {
/* Taken path */
}
else {
/* Not taken path */
}

for (cond) {
/* Code */
}

While (cond) {
/* Code */
}
```

Mitigated by branch prediction

Read my notes on BBLearn.

# Explicit parallelism

Monday, March 30, 2020 12:28 PM

Parallel execution units are exposed to the programmer or to the compiler.

Flynn's classification of parallel machines

SISD : Single instruction single data
SIMD : Single instruction multiple data
MIMD : Multiple instruction multiple data
MISD : Multiple instruction single data

We can also use:
Program (P)
Thread or task (T)
Example: SPMD (or) STMD

SISD: serial code
SIMD: We will operate the GPU in SIMD fashion
↳ Program running on GPU is called [Kernel]
MIMD: Most general form of parallelism
MISD: Not useful for performance (perhaps fault tolerance)

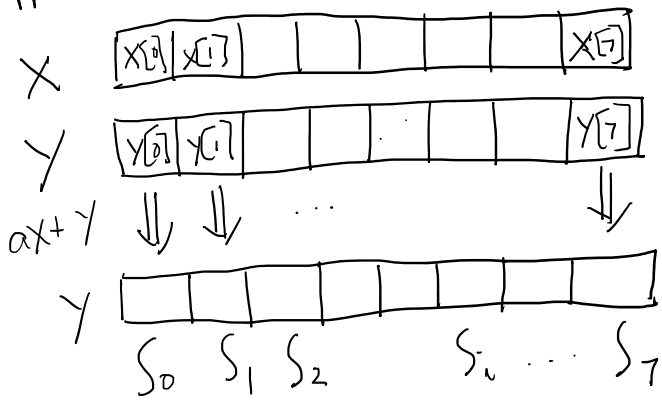
---

## Example of SIMD or SPMD code:

Consider the following SAXPY loop:

```
for(i=0; i<8; i++)
    y[i] = a*x[i] + y[i];
```

Suppose we wish to parallelize loop over 8 threads.




$S_i$ : thread assigned to calculate the $i^{th}$ output element.

Each thread executes the same program, or set of instructions:

$y[\,] = a\,x[\,] + y[\,]$

Only thing that differs for each thread is the data

SIMD program:

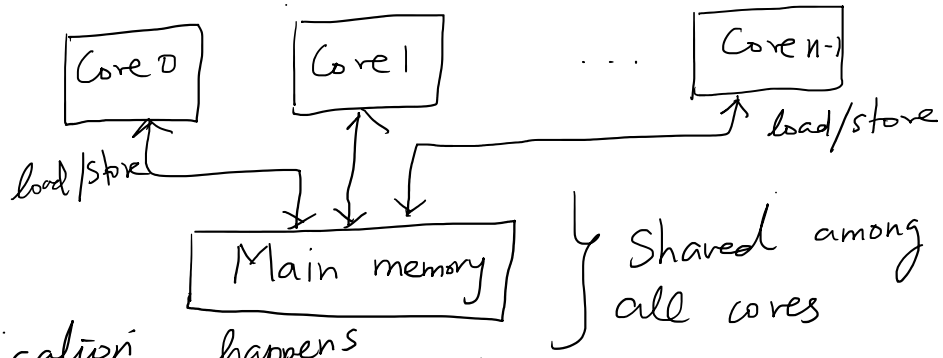
```
tid = obtain thread ID from system;
y[tid] = a*x[tid] + y[tid];
```

Note: tid (thread ID) will be different for each thread.

# Shared memory versus distributed memory machines

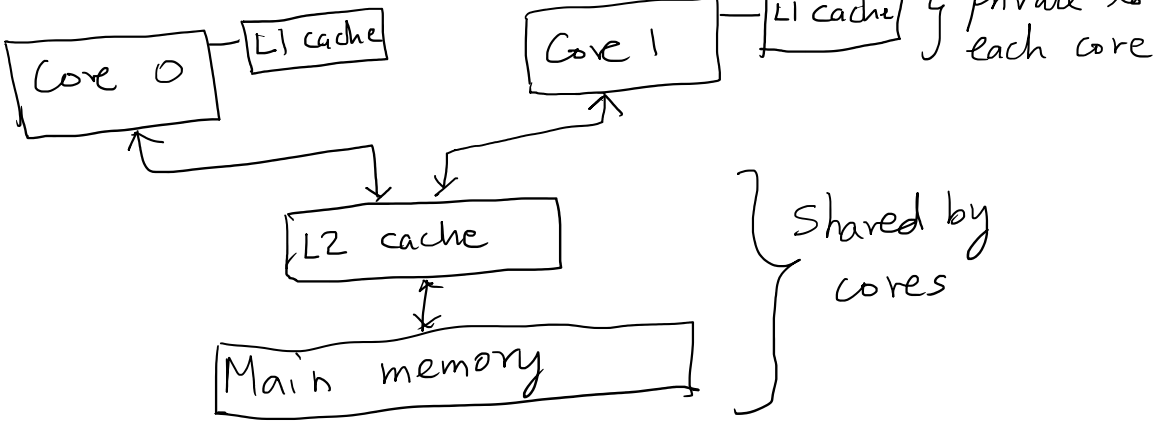
Monday, March 30, 2020 12:29 PM

Shared memory:

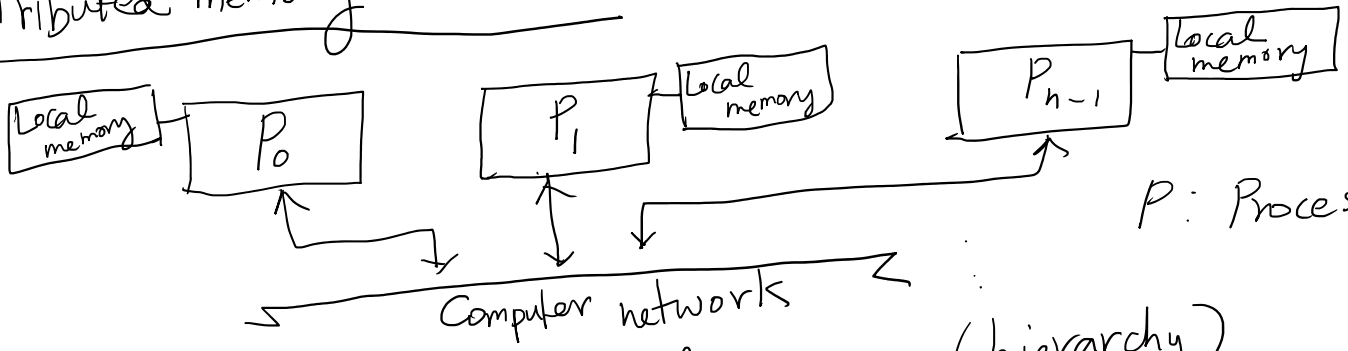



Inter-core Communication happens using load/store instructions.

Example of 2-core machine with cache hierarchy:




Distributed memory machines:




Each processor has its own local memory (hierarchy)
Processors communicate using messages over the network.

Example:

$P_i$

```
while(1){
    msg = recv(Pj);
    /* calculate */
    msg = create new message;
    send(Pj, msg);
}
```

$P_j$

```
while(1){
    msg = recv(Pi);
    /* calculate */
    msg = create new message;
    send(Pi, msg);
}
```

Note: the send() function is non-blocking.
recv() function is blocking.

MPI (Message passing interface) is a popular programming interface for distributed machines.
↳ Not covered in this class.

# Performance models

Monday, April 6, 2020 9:44 AM

Provide insights to programmers and computer architects on improving the performance of parallel software and hardware.

Viability of parallelization.

"Back of the envelop" estimates of performance gains due to parallelization.

Amdahl's law.

Gustafson's law.

Roofline model.

# Amdahl's law

Monday, April 6, 2020 9:35 AM

Is a principle applicable to general computer architecture:

Make the common case fast; that is, in making a design tradeoff, favor the frequent case over the infrequent case.

Principle also applies when determining how to spend resources since the impact on making some occurrence faster is higher if that occurrence is frequent. So, we have to decide what the frequent case is and how much performance can be gained by making that case faster.

Amdahl's law is used to quantify the above principle.

Amdahl's law can also be used to ascertain the viability of implementing large-scale parallelism.

Consider a program that we wish to parallelize.
Normalize its execution time to 1.
We profile the code thoroughly and identify...
s: fraction of code that must be run in serial fashion; $0 \leq s \leq 1$
p: fraction of code that can be parallelized. $0 \leq p \leq 1$

So, $s + p = 1$ (since overall execution time is normalized to 1)

Also, assume (to simplify the analysis) that the parallel portion of the code is "embarrassingly parallel"

- Trivial to parallelize and map to cores.
- Given N cores, the Speedup is P/N

So,

$$\text{Speedup} = \frac{\text{Time taken by serial code}}{\text{Time taken by parallelized code}}$$

$$= \frac{s+p}{s + p/N} = \frac{1}{s + p/N}, \text{ since } s+p=1$$

$$\text{Speedup} = \frac{1}{S + P/N}$$

Unfortunately speedup is very pessimistic.

Suppose 5% of code must be run in serial; $S = 5/100$

95% of the code is parallelizable; $P = 95/100$

We are given $N = \infty$ processors.

$$\text{Speedup} = \frac{1}{\frac{5}{100} + \frac{95/100}{\infty}} = \frac{1}{\frac{5}{100}} = 20$$

↳ Only 20× speedup even with $\infty$ processors ☹

Consider a practical case where $N = 1024$ cores.

$$\text{Speedup} = \frac{1}{\frac{5}{100} + \frac{95/100}{1024}} = 19.6$$

↳ Again very pessimistic given the number of available cores

Why so ~~serious~~ pessimistic?

Key assumption made by Amdahl:

↳ P (the parallelizable version of the code) is independent of N.

# Gustafson's law

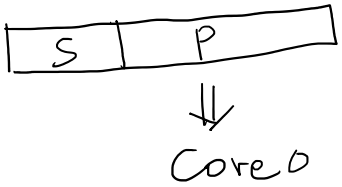
Monday, April 6, 2020 9:43 AM

Key assumption: problem size scales with N

↳ That is, developers will find a way to scale the problem, given more cores to use.

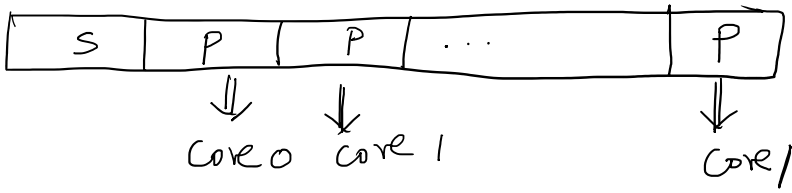
Example:

Assume $s+p=1$, as before




Running time = $s+p$

Given more cores:




Running time = $s+p$ (since each P runs on its own core in parallel)

So, running time of parallel code = $s+p=1$

Running time of equivalent serial code = $s+Np$

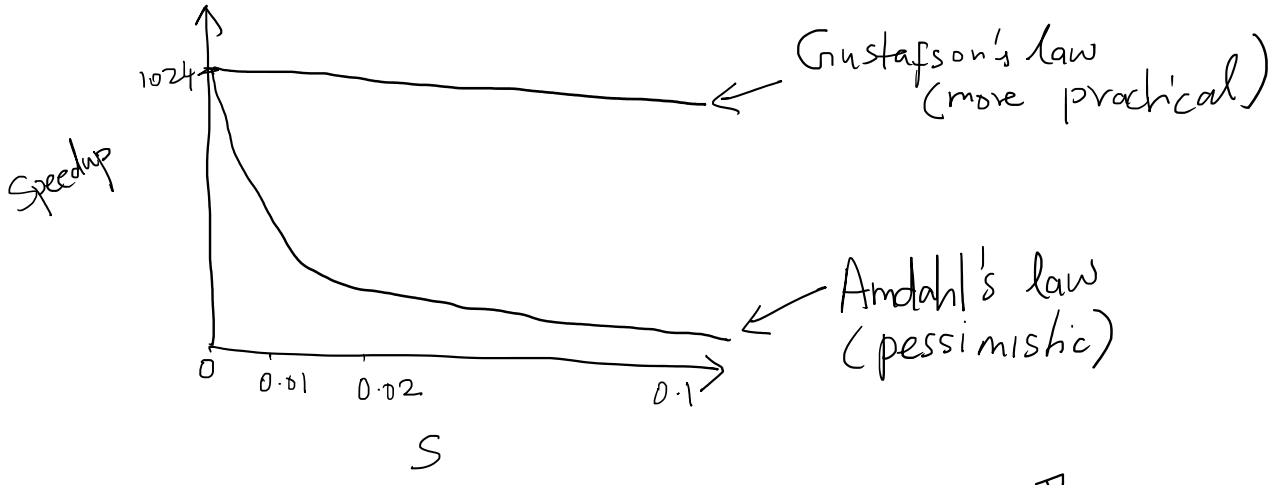
since each p must be run sequentially.

$$\text{Scaled speedup} = \frac{s+Np}{s+p} = \frac{s+Np}{1} = s+Np.$$

Since $p = 1 - s$,

$$\text{Scaled speedup} = s + N(1-s) = N + (1-N)s$$

Speedup plot for $N = 1024$ cores as a function of $s$:




Read more in my notes on BBLearn!

# Roofline model

Monday, April 6, 2020 9:47 AM

Giga Floating point Operations

Answers two questions related to performanc
1) What is the achievable GFLOPS/s of your code?
2) Is your code compute bound or memory bound?

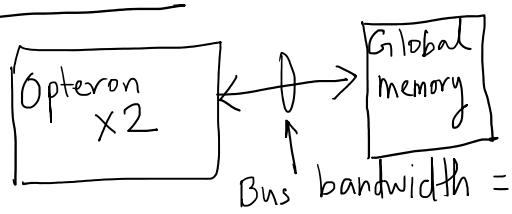
Off chip memory (global memory) bandwidth will be the constraining factor for performance. Remember the "memory gap."

Let's examine the performance of the SAXPY loop on AMD's Opteron X2 processor.

From the manufacturer's spess, we know:

Peak FP performance = 16 GFLOPS/sec
Peak memory bandwidth = 16 GBytes/sec

System schematic:




The SAXPY loop body:

$y[i] = ax[i] + y[i]$

X, Y: Live in global memory
a: scalar value lives in register

Let's now define the notion of "arithmetic intensity":

$$\frac{\text{Total FP operations}}{\text{Total data movement (in bytes)}}$$

Each time through the SAXPY loop, we perform:

1 multiplication $(a.x[i])$,
1 addition $(a.x[i] + y[i])$

Single precision FP = 4 bytes each

We load 8 bytes ($x[i]$ and $y[i]$) from memory.
We store 4 bytes (updated value of $y[i]$) to memory
= 12 bytes in total.

So, arithmetic intensity = $\frac{2}{12} = 0.167 \frac{FLOPS}{Bytes}$

Once we know the arithmetic intensity, we can calculate the GFLOPS/sec achieved by the SAXPY loop as follows:

Attainable GFLOPS/sec = min (Peak FP performance, Peak memory bandwidth × Arithmetic intensity)

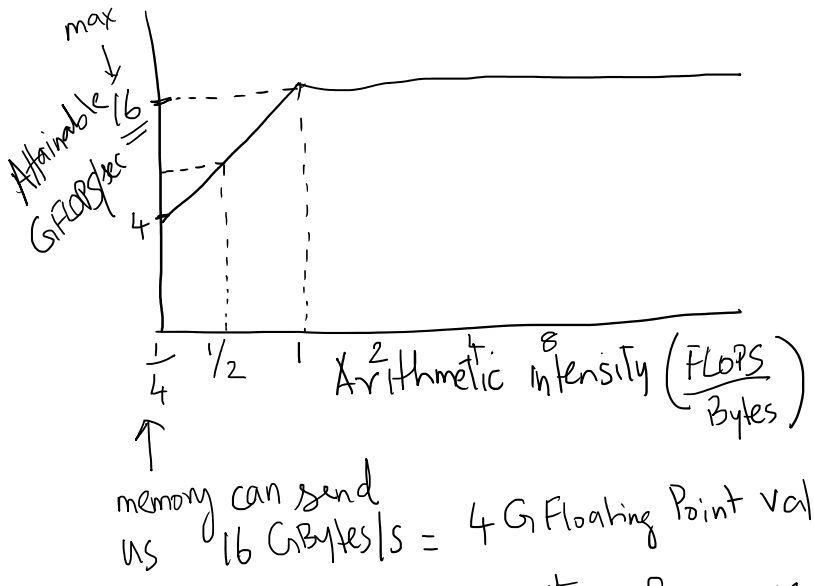
$16 \frac{GBytes}{sec} \times 0.167 \frac{FLOPS}{Bytes}$

$= 2.67 \frac{GFLOPS}{sec}$.

Attainable GFOPS/sec = min (16, 2.67)
= 2.67
→ Not even close!

So, SAXPY is a memory bound program. That is, the performance bottleneck is global memory.

In general, for the opteron X2, the "roofline" performance model looks like this:




Calculate the arithmetic intensity for your program, and place it on the X-axis. Project to the roofline model and obtain Y-axis value.

If you end up on the slope, your program is memory bound (not using all compute power available to you).

If you end up on the roof, your program is compute bound.

Note: For the opteron X2, to become compute bound, arithmetic intensity should be 1.
Because,
Attainable GFLOPS/s = min (16, 16×1)

To achieve an arithmetic intensity of 1, each floating point value loaded from memory [illegible] be reused 4 times by

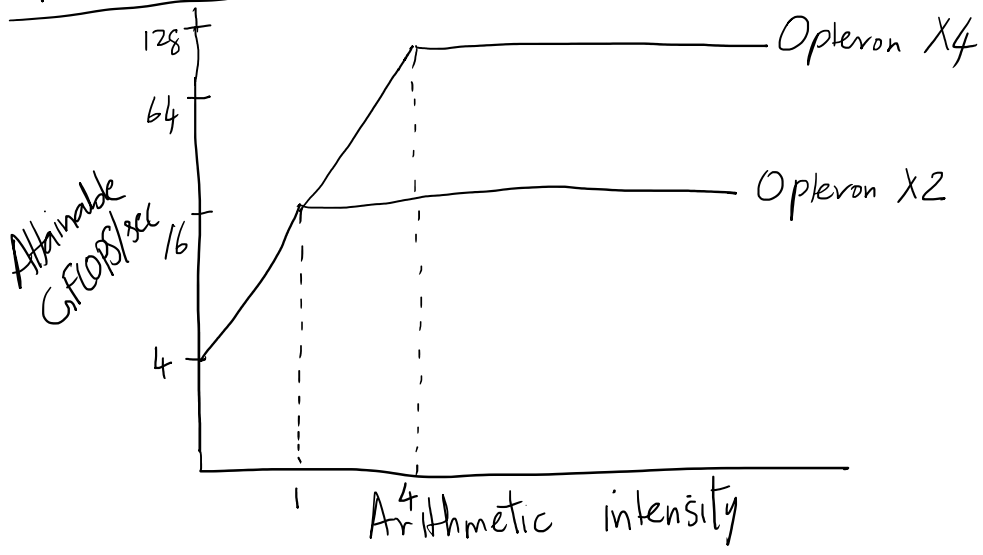
floating point value loaded from memory must be reused 4 times by the CPU (since arithmetic intensity = $\frac{4}{4}$)

This is a lot of reuse!
↳ Not many problems exhibit this behavior.

Problem gets even worse for more powerful processors.

Example:

Opteron X4




To fully utilize the compute capacity on the X4, arithmetic intensity = 4 (or ~ $\frac{16}{4}$)

That is each FP value must be used 16 times by the core!

# Writing parallel programs

Monday, April 6, 2020 9:48 AM

## Key steps:

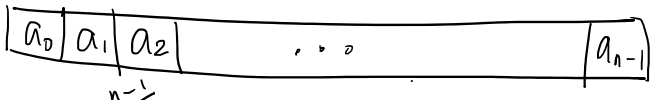
* · Decomposition: Problem must be decomposed between available threads/tasks.
  · Mapping: Assign threads/tasks to processors. Thread to data mapping.
* · Orchestration: data access, communication, synchronization between threads

*: topics this class will cover.

# Reduction

Monday, April 6, 2020 9:49 AM

Given array A with n elements.




calculate

$$sum = \sum_{i=0}^{n-1} a_i$$

## Implementation on multi-core CPU

**Decomposition:** Chunk the input array between k threads.

**chunk size:**

$$chunk = \left\lfloor \frac{n}{k} \right\rfloor$$

$\lfloor \rfloor$ is the floor operator

**Orchestration:**

Each thread gets:

- Thread id (tid)
- Pointer to array A
- chunk size (chunk)

Each thread:

- Computes offset within A as:

$$offset = tid \times chunk$$

- Calculates partial sum: $PS_{tid} = \sum_{i=offset}^{offset+chunk} a_i$

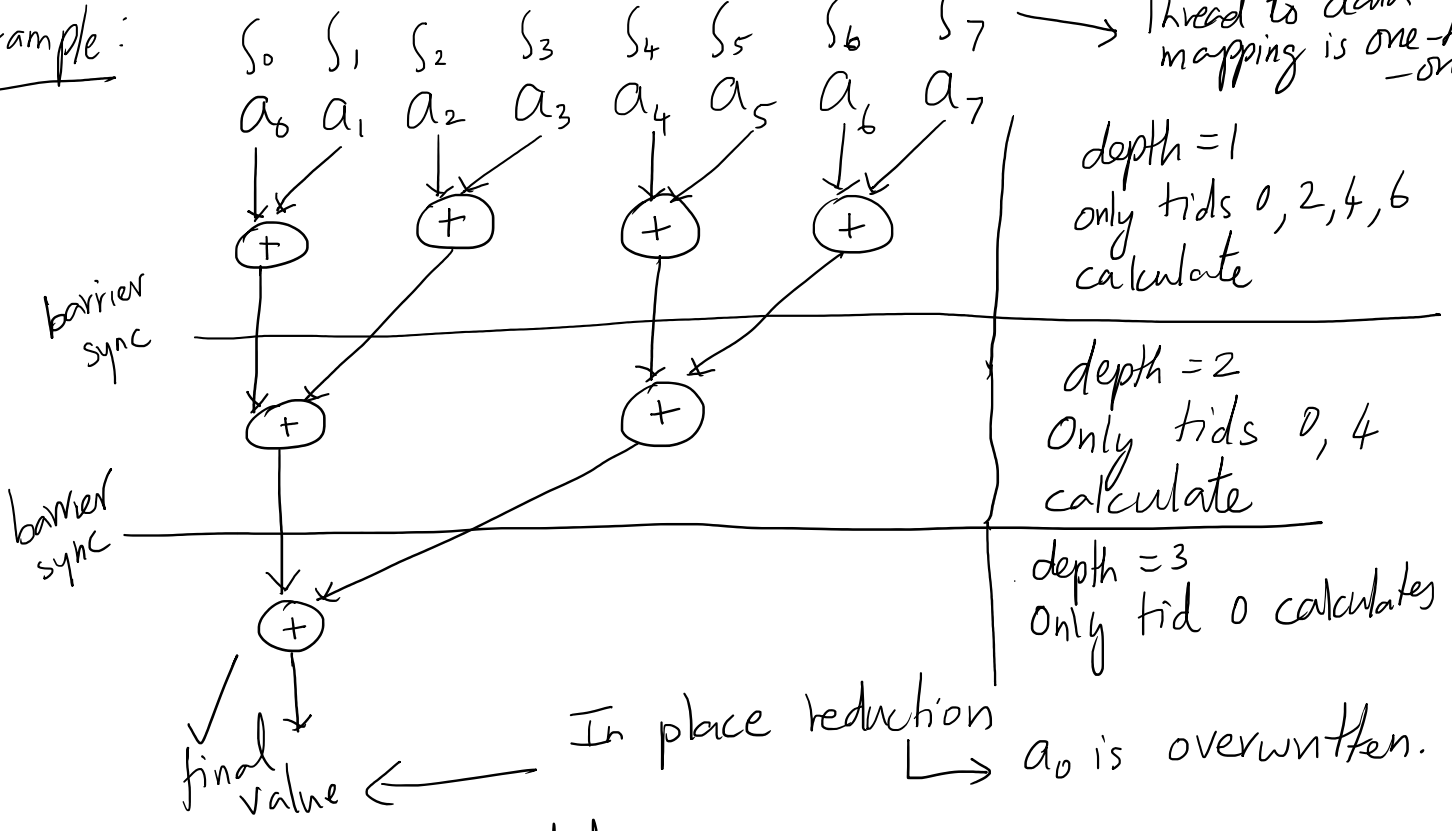
- The last thread with $tid = k-1$ may need to process more elements $PS_{tid=k-1} = \sum_{i=offset}^{n-1} a_i$

- Once each thread calculates $PS_{tid}$, the partial values must be accumulated into the global variable sum.
  - Have one thread do this.
  - Use a lock to protect the shared variable sum, and have each worker thread accumulate into sum.

We will look at code examples

Tree-style reduction on massively multi-core processors → GPUs

Example:




→ Thread to data mapping is one-to-one

depth = 1
only tids 0, 2, 4, 6 calculate

depth = 2
Only tids 0, 4 calculate

depth = 3
Only tid 0 calculates

In place reduction → $a_0$ is overwritten.

Three steps are needed.
For n elements, log(n) steps are needed.

# Grid-based operations (Gauss Seidel and Jacobi)

Monday, April 6, 2020 9:50 AM

Read notes on the parallelization process.
We will revisit this problem under OpenMP.