Lecture 34: Bonus topics

Philipp Koehn, David Hovemeyer

601.229 Computer Systems Fundamentals

Outline

- Systems courses
- GPU programming
- Virtualization and containers
- Digital circuits
- Compilers

Code examples on web page: `bonus.zip`

Systems courses

Systems courses

- One of the main goals of CSF is to prepare you to take the upper level systems courses
- So, what are these courses?
  - Computer Networks (601.414)
  - Distributed Systems (601.417) (*)
  - Operating Systems (601.318/418) (*)
  - Cloud Computing (601.419)
  - Parallel Computing for Data Science (601.420) (*)

(*) Offered Fall 2023 (?)

601.414 Computer Networks

- Network protocols (e.g., IP) and routing
- Transport protocols (e.g., TCP)
- Link layer protocols
- Application protocols and system-level network APIs
- Learn how networks really work at various scales

- A *distributed system* is a system implemented using cooperating processes which communicate over a network
- Huge advantages of distributed systems: true scalability, fault tolerance
- Huge challenges of distributed systems: state is distributed, network communication is unreliable

601.418 Operating Systems

- Process/thread scheduling, multiprogramming
- Virtual memory
- Filesystems
- "Course work includes the implementation of operating systems techniques and routines, and critical parts of a small but functional operating system."
- Opinion: operating systems are incredibly interesting and fun

601.419 Cloud Computing

- "Cloud" $=$ presenting data storage or computation as a service over the network
- Cloud infrastructure: virtual servers, cloud services
- All "internet-scale" applications are built using cloud technology
- Note: 601.414 Computer Networks is a prerequisite

601.420 Parallel Computing for Data Science

- Use multiple processors to speed up large computations
  - Often, large computations will have large amounts of data: storage and transfer must be considered
- Making parallel computation work:
  - Designing parallel algorithms
  - Using parallel APIs and environments to run parallel programs
- "This course studies parallelism in data science, drawing examples from data analytics, statistical programming, and machine learning. It focuses mostly on the Python programming ecosystem but will use C/C++ to accelerate Python and Java to explore shared-memory threading. It explores parallelism at all levels, including instruction level parallelism (pipelining and vectorization), shared-memory multicore, and distributed computing."

GPU programming

3D graphics

Rendering 3D graphics requires significant computation:

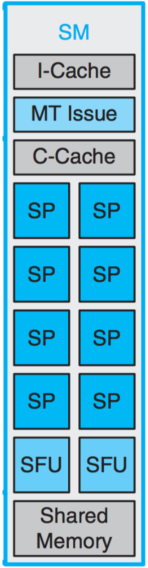
- Geometry: determine visible surfaces based on geometry of 3D shapes and position of camera
- Rasterization: determine pixel colors based on surface, texture, lighting

A *GPU* is a specialized processor for doing these computations fast

GPU computation: use the GPU for general-purpose computation

Streaming multiprocessor

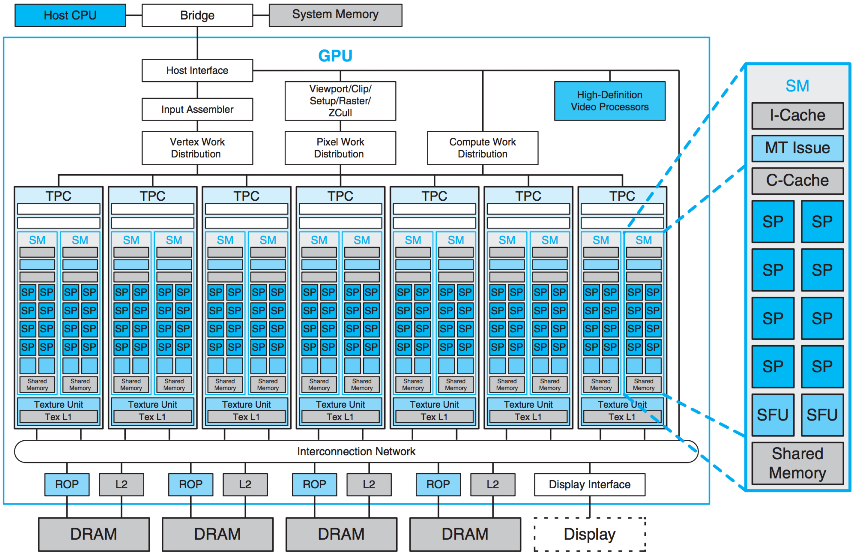
- Fetches instruction (I-Cache)
- Has to apply it over a vector of data
- Each vector element is processed in one thread (MT Issue)
- Thread is handled by scalar processor (SP)
- Special function units (SFU)



Flynn's taxonomy

- SISD (single instruction, single data)
  - uni-processors (most CPUs until 1990s)
- MIMD (multi instruction, multiple data)
  - all modern CPUs
  - multiple cores on a chip
  - each core runs instructions that operate on their own data
- SIMD (single instruction, multiple data)
  - Streaming Multi-Processors (e.g., GPUs)
  - multiple cores on a chip
  - same instruction executed on different data

GPU architecture



GPU programming

- If you have an application where
  - Data is regular (e.g., arrays)
  - Computation is regular (e.g., same computation is performed on many array elements)

  then doing the computation on the GPU is likely to be much faster than doing the computation on the CPU
- Issues:
  - GPU has its own instruction set (need a compiler)
  - GPU has its own memory (need to allocate buffers on GPU, copy data between host memory and GPU memory)

Options for GPU programming

- OpenCL (`https://www.khronos.org/opencl/`)
  - Advantage: device agnostic (supported by multiple vendors)
  - Disadvantage: complicated
- CUDA (`https://developer.nvidia.com/cuda-toolkit`)
  - Advantage: fairly straightforward to use (dialect of C)
  - Disadvantage: only supports NVIDIA hardware

Application: image processing

- Gaussian blur: pixels of result image are weighted average of NxN block of surrounding pixels
- Just a straightforward 2D array problem
- On the GPU, a *kernel function* will compute result pixel values in parallel
- A *kernel function* will compute a result pixel value
- Kernel function invocations for each combination of block coordinate and thread number
  - A "block" typically specifies an array element or range of array elements
  - Each block spawns some number of threads
- We'll implement this using CUDA (see `blur.cu`)

Core (per-pixel) computation

```
// x/y are pixel coordinates, in is original image,
// result is array of computed color component values
unsigned filter_index = 0;

for (int i = y - FILTER_WIDTH/2; i <= y + FILTER_WIDTH/2; i++) {
  for (int j = x - FILTER_WIDTH/2; j <= x + FILTER_WIDTH/2; j++) {
    if (i >= 0 && i < h && j >= 0 && j < w) {
      int index = i*w + j;
      uint32_t orig_pixel = in[index];
      float fac = filter[filter_index];
      result[0] += (RED(orig_pixel) * fac);
      result[1] += (GREEN(orig_pixel) * fac);
      result[2] += (BLUE(orig_pixel) * fac);
      weight_sum += fac;
    }
    filter_index++;
  }
}
```

Normalize, clamp, compute result pixel

```
// out is result image
for (int i = 0; i < 3; i++) {
  // normalize
  result[i] /= weight_sum;
  // clamp to range 0..255
  if (result[i] < 0) {
    result[i] = 0;
  } else if (result[i] > 255) {
    result[i] = 255;
  }
}

uint32_t transformed_pixel = RGBA(
  (unsigned)result[0], (unsigned)result[1], (unsigned)result[2],
  ALPHA(in[index])
);
out[index] = transformed_pixel;
```

Execute using CPU

```
void execute_blur(struct Image *img, struct Image *out, float *filter) {
  for (unsigned i = 0; i < img->height; i++) {
    for (unsigned j = 0; j < img->width; j++) {
      // ...per-pixel computation...
    }
  }
}
```

Execute using GPU

```
void execute_blur_cuda(struct Image *img, struct Image *result_image,
                       float *filter) {
  // ...allocate device buffers, copy host data to device...

  int grid_w = img->width / THREADS_PER_BLOCK;
  if (img->width % THREADS_PER_BLOCK != 0) {
    grid_w++;
  }
  int grid_h = img->height;

  dim3 grid(grid_w, grid_h);

  // invoke the kernel!
  cuBlur<<<grid, THREADS_PER_BLOCK>>>(
    dev_imgdata, dev_filter, dev_imgdata_out, img->width, img->height
  );

  // ...copy device buffers back to host...
}
```

GPU kernel function

```
__global__ void cuBlur(uint32_t *in, float *filter, uint32_t *out,
                       int w, int h)
{
  // pixel to compute
  int x = blockIdx.x * THREADS_PER_BLOCK + threadIdx.x;

  if (x < w) {
    int y = blockIdx.y;

    // ...per-pixel computation...
  }
}
```

Experiment

- `acadia.png` is a 3840x2160 PNG image
- CPU is Core i5-4590, GPU is GeForce GT 1030
- Time to perform image processing measured using `gettimeofday`

```
$ ./blur cpu acadia.png out.png
Computation completed in 9.290 seconds
$ ./blur gpu acadia.png out2.png
3 GPU processors, 1024 max threads per block
Computation completed in 0.263 seconds
```

CPU code could have been optimized better, but still, a very nice performance boost

Virtualization and containers

Running an application program

An application program will run correctly only when:

- It is run on the correct kind of processor
- It is run on the correct operating system
- The correct runtime libraries are available

Let's assume that you have the right processor, but not necessarily the right OS and libraries

What to do?

Virtualization

One solution is *virtualization*

- Create a hard disk image containing the operating system, all required libraries and software components, and the application
- Run this OS image in a *hypervisor*

Hypervisor

- The hypervisor emulates the hardware of a computer, but it's really just a program
  - The hypervisor is the "host"
  - The OS image containing the application is the "guest"
- How it works:
  - Modern CPUs have special instructions and execution modes to make this reasonably efficient
  - The guest's kernel mode is really executing in user mode
  - System instructions executed by the guest OS kernel (e.g., reloading the page directory address register) trap to the hypervisor
  - Hypervisor emulates hardware devices (storage, display, network adapter, etc.)

Disadvantages of virtualization

- Virtualization is somewhat heavyweight
- Significant duplication in code, data structures between hypervisor and guest OS kernel

Containers

- An “operating system” is:
  - A *kernel* (e.g., the Linux kernel) providing a system call API
  - Supporting programs and libraries
- In general, the OS kernel will have a high degree of backwards compatibility
  - So, it is the supporting programs and libraries that are the most important application dependency
- *Container*: an isolated environment in which arbitrary applications and libraries can be run
  - Can be configured to have its own filesystem namespace, process id namespace, network interface, resource limits, etc.
  - But: there is only one kernel serving all processes (including processes running inside a container)

Docker

- Docker (https://www.docker.com/) is a set of tools and an ecosystem for building OS images to run inside a Linux container
- Uses *layered filesystems*
  - Base layers are for the OS executables and libraries (e.g., Ubuntu)
  - You add your application files "on top" of the base OS layer
- A Docker image can be easily deployed to an arbitrary server
  - And you don't need to worry about availability or compatibility of libraries, because they're part of your Docker image

Digital circuits

How do computers actually work?

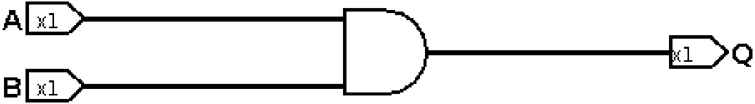
- Computers are *digital circuits*
- Volage levels (high and low) represent true/false
  - or 1/0
- *Logic gates* take 1 or more input voltages, and produce an output voltage that is a boolean function on the input voltages

Learn by doing!

- We will barely scratch the surface of this topic
- But: this is a topic you can explore on your own
- How to do it:
  - Download Logisim evolution (`https://github.com/reds-heig/logisim-evolution`)
  - Build circuits, test their behavior
- Example Logisim files are in the example code

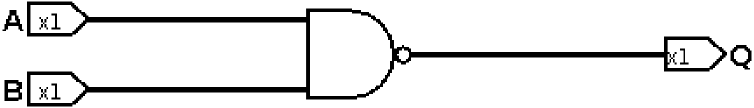
AND gate

Output is 1 IFF inputs are both 1



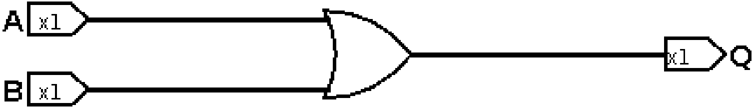
NAND gate

Output is 1 IFF inputs are not both 1 (“not AND”)



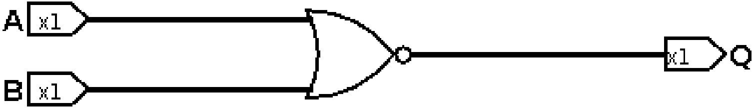
OR gate

Output is 1 IFF either input is 1



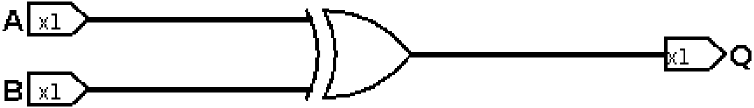
NOR gate

Output is 1 IFF inputs neither input is 1 ("not OR")

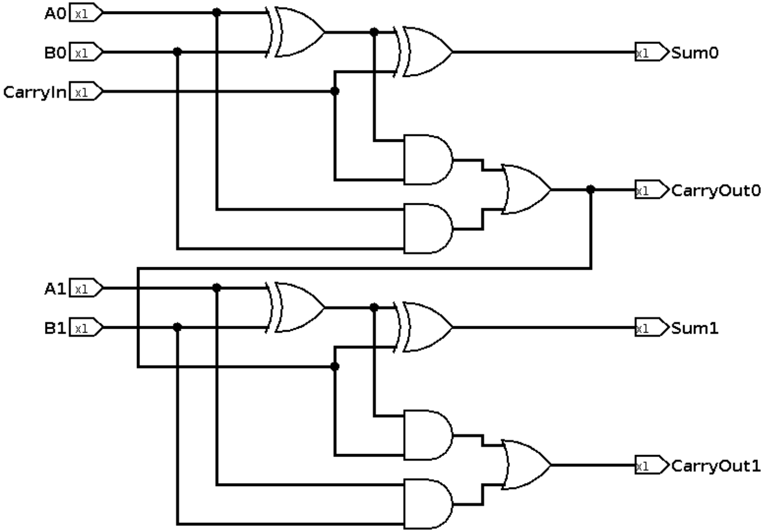


XOR gate

Output is 1 IFF exactly 1 input is 1 (“exclusive OR”)



Two bit adder



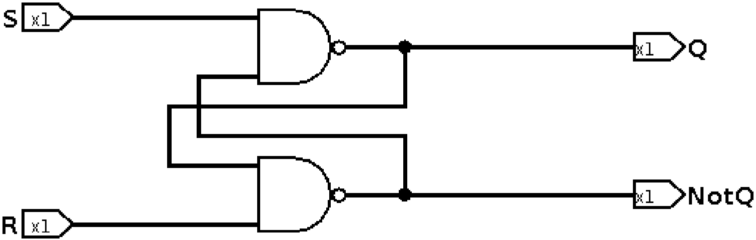
Combinational vs. sequential logic

- Previous examples are *combinational* logic
- Mapping of inputs to outputs is a mathematical function
- Digital circuits that have feedback paths can implement *sequential* logic: there is "state" that can change

SR latch

Normally, S and R inputs should both be set to 1

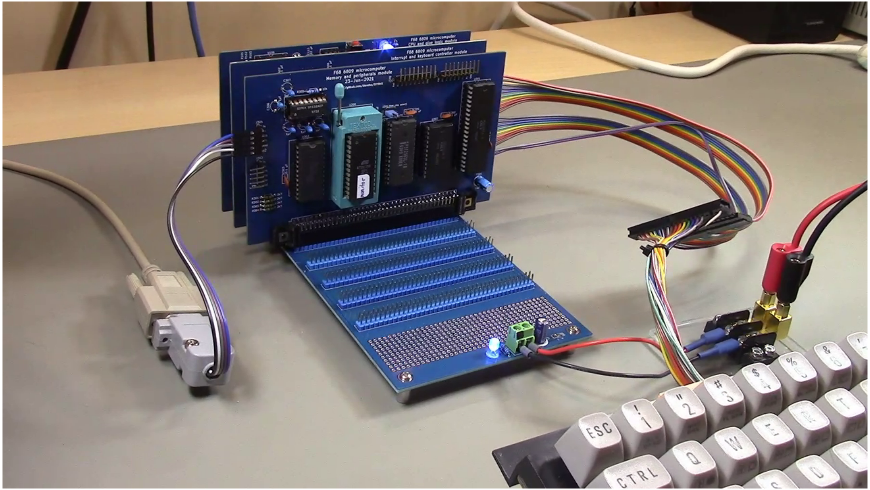
- Pulse S to 0 and back to 1 to change Q output to 1
- Pulse R to 0 and back to 1 to change Q output to 0
- NotQ output is always the inverse of Q



This is a 1-bit memory!

Building a microcomputer

If you know how to design digital circuits, you can build an actual computer



Compilers

Compilers

- We know that writing assembly language is challenging
- A compiler is a program that automates the generation of assembly language
- 601.428 Compilers and Interpreters
  - Lexical analysis and parsing
  - Semantic analysis
  - Code generation
  - Program analysis
  - Code optimization
- Offered in Fall 2023(?)

Compiler bootstrapping

- When a compiler is implemented in the same source language that it accepts as input, it can be *self-hosting*
- E.g., let's say that you write a C compiler in C
- Build it using `gcc`, then use the resulting compiler executable to "compile itself"
- This "bootstrapping" process is typical for new compiler implementations of existing source languages
- When implementing a compiler for a completely new language, you must first implement a compiler or interpreter using a `different` language
  - For example, the first version of the Rust compiler was written in OCaml
  - This allowed the bootstrapping of a later version which was written in Rust