Lecture 2: Hello World in C
(Piazza is a happening place)
Two Great Questions From After the First Lecture

• 1) cheating: can I look at general code on the internet?

• 2) AP students: do you have enough background?
What should you be doing?

• Setting up VM with ArchLinux
• Playing with Unix
• Editor: Vimtutor, etc.
• Project 1A
• New item, starting today: 2A
Questions on Unix stuff?
Directories are hierarchical

• Directories can be placed within other directories

• “/” -- The root directory
  – Note “/”, where Windows uses “\”

• “/dir1/dir2/file1”
  – What does this mean?

File file1 is contained in directory dir2, which is contained in directory dir1, which is in the root directory
Home directory

• Unix supports multiple users
• Each user has their own directory that they control
• Location varies over Unix implementation, but typically something like “/home/username”
• Stored in environment variables

```bash
fawcett:~ childs$ echo $HOME
/Users/childs
```
cd: change directory

- The shell always has a “present working directory”
  - directory that commands are relative to
- The command “cd” changes the present working directory
- When you start a shell, the shell is in your “home” directory
Unix commands: ls

- **ls**: list the contents of a directory
  - Note this is “LS”, not “is” with a capital ‘i’
- Many flags, which we will discuss later
  - A flag is a mechanism for modifying a Unix programs behavior.
  - Convention of using hyphens to signify special status
- “ls” is also useful with “wild cards”, which we will also discuss later
Conventions for a shell

• *: wildcard ... matches anything
  – h*k: matches hank, hulk, hack, etc
• “.”: the current directory
  – “ls .” would list the current directory
• “..”: the parent directory
  – “ls ..” would list the parent directory
File manipulation

New commands: mkdir, cd, touch, ls, rmdir, rm
Unix commands: mkdir

• mkdir: makes a directory
  – Two flavors
    • Relative to current directory
      – mkdir dirNew
    • Relative to absolute path
      – mkdir /dir1/dir2/dirNew
        » (dir1 and dir2 already exist)
Unix commands: rmdir

• rmdir: removes a directory
  – Two flavors
    • Relative to current directory
      – rmdir badDir
    • Relative to absolute path
      – rmdir /dir1/dir2/badDir
        » Removes badDir, leaves dir1, dir2 in place

• Only works on empty directories!
  – “Empty” directories are directories with no files

Most Unix commands can distinguish between absolute and relative path, via the “/” at beginning of filename.
(I’m not going to point this feature out for subsequent commands.)
Unix commands: touch

• touch: “touch” a file

• Behavior:
  – If the file doesn’t exist
    • → create it
  – If the file does exist
    • → update time stamp

Time stamps record the last modification to a file or directory

Will talk more about this command with build systems
Important: “man”

• Get a man page:

• → “man rmdir” gives:

RMDIR(1) BSD General Commands Manual RMDIR(1)

NAME
rmdir -- remove directories

SYNOPSIS
rmdir [-p] directory ...

DESCRIPTION
The rmdir utility removes the directory entry specified by each directory argument, provided it is empty.

Arguments are processed in the order given. In order to remove both a parent directory and a subdirectory of that parent, the subdirectory must be specified first so the parent directory is empty when rmdir tries to remove it.

The following option is available:

-p Each directory argument is treated as a pathname of which all components will be removed, if they are empty, starting with the last most component. (See rm(1) for fully non-discriminant
Project 1A
Project 1A

• Practice using an editor
• Must be written using editor on Unix platform
  – I realize this is unenforceable.
  – If you want to do it with another mechanism, I can’t stop you
    • But realize this project is simply to prepare you for later projects
Project 1A

- Write \( \geq 300 \) words using editor (vi, emacs, other)
- Topic: what you know about C programming language
- Can’t write 300 words?
  - Bonus topic: what you want from this course
- How will you know if it is 300 words?
  - Unix command: “wc” (word count)
Unix command: wc (word count)

```
fawcett:~ childs$ vi hanks_essay
fawcett:~ childs$ wc -w hanks_essay
    252 hanks_essay
fawcett:~ childs$ wc hanks_essay
   63    252    1071 hanks_essay
fawcett:~ childs$
```

(63 = lines, 252 = words, 1071 = character)
Project 1A

CIS 212: Project #1A
Assigned: October 1st, 2019
Due: October 4th, 2019
(which means submitted by 6am on October 5th, 2019)
Worth 2 points

Assignment:
1) On your virtual box using the class Arch Linux image, use an editor (vi, emacs, sublime, other) to write a 300 word “essay”
   a. The purpose of the essay is to practice using an editor.
      i. Grammar will not be graded
   b. I would like to learn more about what you know about C and want from this class ... I recommend you each write about that.
   c. If you run out of things to say, you don’t have to write original words (copy/paste using vi commands: yyp)

Do not write this in another editor and copy into vi.

Also, do not put more than 100 characters onto any given line. (I want you to practice having multiple lines and navigating.) If you have more than 100 characters per line, you will receive half credit.

Grading: I prefer correct work late over incorrect work on time.
Examples:
- If you have more than 100 characters in any line, then you will get 0.9 points
- If you have less than 300 words, then you will get 0.9 points
- If there is evidence that your text was copy-pasted from a Windows machine (or elsewhere), then you will get 0 points

Turn this in using Canvas.
How to submit

• Canvas
Today’s Lecture:
How Do Computers Work?

• Especially as it relates to Python and C
What is Data?
To a Computer, Data Means Lots of Bits. So What is a Bit?

• Short for “binary digit”
• 2 states:
  – on (1)
  – off (0)
• Like a light switch
• Computers can turn their bits (light switches) on and off
• Computers implement this using electricity in a “capacitor”:
  – has electrical charge: on / 1
  – no electrical charge: off / 0
What is a Byte?

• A group of 8 bits

• We could write the values of the 8 bits in a row.
• Example: 01000001
  – This is a binary number
Binary Numbers

• Normal system (called decimal):
  – Values for any digit are: 0,1,2,3,4,5,6,7,8,9
  – What we use every day

• Binary:
  – Values for any digit are: 0,1
  – What computers use
Conventions Surrounding Binary Numbers

• 01000001 is 65 in decimal
• By convention, 65 represents ‘A’ — Called “ASCII”

Also conventions for supporting decimal numbers (3.456)...

Source: www.LookupTables.com
There are 10 types of people in the world – those that know binary and those that don’t.
How do Bytes relate to computers?

• Computers have “memory”
  – Places to store data and then retrieve it later

• The memory is made up of many, many Bytes
How a Computer is Like Monopoly

- Monopoly is a “turn-based” game
  - Player 1 takes a turn
  - Player 2 takes a turn
  - Etc.
- Various things happen during a turn
- Then it is the next turn
- And so on...
Terminology for a computer: cycle

• A computer operates in “cycles”

• The goal during a cycle is to compute one thing (*)
  – Various things may need to happen for the computer to carry out the cycle

• At the end of a cycle, the computer starts a new cycle, to carry out a new computation
  – (At the end of a monopoly turn, the game moves on to the next turn...)

How a Computer is Not Like Monopoly

- Actions during cycle/turn:
  - Monopoly: roll, buy, build, trade
  - Computer: other

- Who’s turn?
  - Monopoly: passes between players
  - Computer: always the computer’s turn

- Time spent per cycle/turn:
  - Monopoly: variable
  - Computer: fixed

- Duration of turn:
  - Monopoly: sometimes over a minute
  - Computer: much faster!!!
How a Computer is **Not** Like Monopoly

- **Actions during cycle/turn:**
  - Monopoly: roll, buy, build, trade
  - Computer: other
- **Who’s turn?**
  - Monopoly: passes between players
  - Computer: always the computer’s turn
- **Time spent per cycle/turn:**
  - Monopoly: variable
  - Computer: fixed
- **Duration of turn:**
  - Monopoly: sometimes over a minute
  - Computer: much faster!!!
Actions taken by a computer

• A computer operates by loading a special kind of file called a “program”
  – This file has a sequence of instructions in it
• Instructions are very primitive:
  – Add, subtract, <lots of math stuff>
  – Load from memory, store to memory
  – “Jump”
  – Some other things...
Instructions

• Every cycle get an instruction -- “bytes that explain what to do”
• If byte == 000, then add two numbers
• If byte == 001, then subtract two numbers
• If byte == 002, then multiply two numbers
• ...

• (and of course the real conventions are very different)
Computer programs can do iteration

- Line 100: X=0
- Line 101: A=location in memory
- Line 102: if X=10, jump to line 106
- Line 103: A = 2*A  (for programmers: *A=2**A)
- Line 104: X = X+1
- Line 105: jump to line 102
- Line 106: .... (other stuff)

With iteration, we can write finite length programs that can run forever...
Billions and Billions of Instructions Can Add Up to Do Great Things
Von Neumann Architecture

Control unit gets a new instruction every cycle

Instructions come from memory unit

So how are these instructions generated?
How computers do work?

• Instructions come from programs, which are loaded into memory.
• Programs contain a sequence of instructions that CPU can understand (“opcodes”) and those instructions are fetched one by one.
  – The correct term is technically machine code and not opcodes. Opcodes refer to the instruction to perform (plus, minus, times, etc), while machine code augment opcodes with other info.
So where do programs come from?

- (Answer coming in a few slides)
Deep thought: Python is a program

- Python gets input from the keyboard
- It “interprets” the input
- It translates the input to something it understands
- It converts that input to instructions for the CPU
- It sends the instructions to the CPU
Python example

• >>> 4+6
• Python recognizes this as two numbers (4 and 6) and a known instruction (+)
• It sends the numbers and the opcode for + to the CPU as an instruction
• The CPU does the math
• Python takes the result (10) and prints it
Python example

• How did Python know that “4+6” was two numbers and a known instruction?
• Answer: the + instruction and “4” and “6” arguments were the LAST things it did.
• Before that, it used other instructions to understand the string and determine what it was supposed to do.
• Typing “4+6” into the terminal actually requires thousands of instructions (or more) to carry out the 1 instruction you wanted!
Python is an interpreter

• It accepts strings that you type
• It determines your intended actions
• It converts those intended actions into instructions for CPU
• It sends the instructions to the CPU
• It displays the results
• ... and it does this until you quit.
Deep thought: where did Python come from?

- Python is a program.
- Actually a very complex program.
- Something created that program.
- The thing that creates that program is called a compiler.
The Workflow With Compilers

1) start with source code
2) invoke compiler, which takes source code and generates executable
   - The executable is made up of instructions for the CPU (opcode)
3) invoke executable
   - The instructions for the CPU are fed to the CPU one at a time
Hello World Example

```c
#include <stdio.h>
int main()
{
    printf("Hello world\n");
}
```

Hanks-iMac:Downloads hank$ gcc hello.c
.Hanks-iMac:Downloads hank$ ./a.out
Hello world
Notes on previous example

• Ignore for now: `#include <stdio.h>`
• “main” is the first function called in C.
  – (Sort of. There are other things called before main to set up the program. But you won’t be mucking with those.)
• `gcc` is a C compiler. It is short for GNU Compiler Collection. GNU is an open source effort.
• The output is called “a.out”
• We have to add “./”, as it “./a.out”
  – “.” is the directory the shell is in. Therefore “./a.out” says run the program called a.out in the current directory.
More Deep Thoughts

• The compiler is a program. Its job is to make other programs.

• The operating system is a program. Its job is to run other programs (and provide an environment for them).

• Where did the first compiler come from?
<table>
<thead>
<tr>
<th>If your job is to...</th>
<th>How do you feel about it? (*)</th>
<th>Is it fast?</th>
<th>How does it become a program?</th>
</tr>
</thead>
<tbody>
<tr>
<td>By hand, write binary files full of opcodes</td>
<td><img src="image" alt="I Hate My Life." /></td>
<td>Yes</td>
<td>It already is a program</td>
</tr>
<tr>
<td>Write “assembly code” (the instructions in English, not opcodes)</td>
<td><img src="image" alt="Not Great" /></td>
<td>Yes</td>
<td>Assembler</td>
</tr>
<tr>
<td>Write C programs</td>
<td><img src="image" alt="prettygood" /></td>
<td>Yes, although assembly can be faster</td>
<td>Compiler</td>
</tr>
<tr>
<td>Write Python programs</td>
<td><img src="image" alt="Amazing!" /></td>
<td>Almost certainly not, unless it is calling subroutines in C</td>
<td>Python is already a program</td>
</tr>
</tbody>
</table>

* = I feel AMAZING about C, and only pretty good about Python
```assembly
#include <stdio.h>

int main()
{
    int X = 3*3+3;
    printf("X is %d\n", X);
}
```

```
Hanks-iMac:Downloads hank$ cat simple_math.s

.section __TEXT,__text,regular,pure_instructions
.macosx_version_min 10, 12
.globl _main
.p2align 4, 0x90

_main:
    .cfi_startproc
    ## @main
    pushq  %rbp
    Ltmp0:
        .cfi_def_cfa_offset 16
    Ltmp1:
        .cfi_offset %rbp, -16
        movq   %rsp, %rbp
    Ltmp2:
        .cfi_def_cfa_register %rbp
        subq   $16, %rsp
        leaq   L_.str(%rip), %rdi
        movl   $12, -4(%rbp)
        movl   -4(%rbp), %esi
        movb   $0, %al
        callq  _printf
        xorl   %esi, %esi
        movl   %eax, -8(%rbp)  ## 4-byte Spill
        movl   %esi, %eax
        addq   $16, %rsp
        popq   %rbp
        retq
    .cfi_endproc

.section __TEXT,__cstring,cstring_literals
L_.str:
    .asciz "X is %d\n"

 subsections_via_symbols
```
Notes

• Assembly code varies from architecture to architectures

• C does not
  – You can code in C, and the compiler will make your code work anywhere

• Just about every command in C corresponds to a small handful of assembly instructions
  – This means C will be fast

• This is not true in Python
<table>
<thead>
<tr>
<th>If your job is to...</th>
<th>How do you feel about it? (*)</th>
<th>Is it fast?</th>
<th>How does it become a program?</th>
</tr>
</thead>
<tbody>
<tr>
<td>By hand, write binary files full of opcodes</td>
<td><img src="image" alt="I Hate My Life." /></td>
<td>Yes</td>
<td>It already is a program</td>
</tr>
<tr>
<td>Write “assembly code” (the instructions in English, not opcodes)</td>
<td><img src="image" alt="Not Great" /></td>
<td>Yes</td>
<td>Assembler</td>
</tr>
<tr>
<td>Write C programs</td>
<td><img src="image" alt="prettygood" /></td>
<td>Yes, although assembly can be faster</td>
<td>Compiler</td>
</tr>
<tr>
<td>Write Python programs</td>
<td><img src="image" alt="Amazing!" /></td>
<td>Almost certainly not, unless it is calling subroutines in C</td>
<td>Python is already a program</td>
</tr>
</tbody>
</table>

The reason I like C is the compiler. It looks at all of my code, and tells me right away if there is a syntax problem.

* = I feel AMAZING about C, and only pretty good about Python
Aside: diagramming sentences

• English language is made up of sentences.
• There are different formulas for a sentence.
• Sentences always end with punctuation.
  – This punctuation reduces ambiguity and helps the reader.

We don’t try learn languages (spoken or programming) by looking at the grammar that defines them. We do it by example....

Source: wikihow.com
In C, statements are concluded with semicolons. And statements can contain many expressions.

Source: reddit.com
More C: variable types

• When you declare variables, you have to declare their type.
  – We learned one type: int (integer)
  – There are more types (floating point, etc). We will discuss them later.
More C: functions

• As you would expect, you can define your own functions and call them.

```c
#include <stdio.h>
int foo1() { return 3; }
int foo2() { return 4; }
int main()
{
    printf("%d\n", foo1()*foo2());
}
```

```
Hanks-iMac:Downloads hank$ gcc function.c
Hanks-iMac:Downloads hank$ ./a.out
12
```
More C: every function has its own “scope” and its variables live within that scope

```c
#include <stdio.h>

int foo1()
{
    int X = 3;
    return X;
}

int main()
{
    int Y = foo1();
    printf("X is %d\n", X);
}
```

```
Hanks-iMac:Downloads hank$ gcc scope.c
scope.c:12:25: error: use of undeclared identifier 'X'
    printf("X is %d\n", X);
  ^
1 error generated.
```
You can also use more `{ and }` within a function. This affects variable scope.

```c
#include <stdio.h>

int main()
{
    int Y = 3;
    {
        int X = 2;
    }
    printf("X is %d\n", X);
}
```

Hanks-iMac:Downloads hank$ gcc scope2.c
```
scope2.c:9:25: error: use of undeclared identifier 'X'
    printf("X is %d\n", X);
```

1 error generated.
Iteration is done with constructs “for” and “while”

```c
#include <stdio.h>

int main()
{
    int X = 0;
    while (X < 83)
        X += 5;
    printf("X is %d\n", X);
}
```

Hanks-iMac:Downloads hank$ gcc while.c
Hanks-iMac:Downloads hank$ ./a.out
X is 85
What’s the answer?

```c
#include <stdio.h>

int main()
{
    int X = 0;
    while (X < 8)
    {
        X += 5;
        X += 2;
        printf("X is %d\n", X);
    }
}
```

```
Hanks-iMac:Downloads hank$ gcc while.c
./Hanks-iMac:Downloads hank$ ./a.out
```
```c
#include <stdio.h>

int main()
{
    int X = 0;
    while (X < 8)
    {
        X += 5;
        X += 2;
    }
    printf("X is %d\n", X);
}
```

```
Hanks-iMac:Downloads hank$ gcc while.c
Hanks-iMac:Downloads hank$ ./a.out
X is 14
X is 12
```
The for loop

• Three main components:
  – Initialization
  – Termination
  – What to do each step

• Often used with a loop variable (example: i)
  – Initialization: i = 0
  – Termination: i < 10
  – What to do each step: i = i+1
For loop in practice

Hanks-iMac:Downloads hank$ cat sum.c
#include <stdio.h>
int main()
{
    int sum = 0;
    int i;
    for (i = 0 ; i < 10 ; i = i+1)
    {
        sum = sum+i;
    }
    printf("Sum is %d\n", sum);
}

Hanks-iMac:Downloads hank$ gcc sum.c
Hanks-iMac:Downloads hank$ ./a.out
Sum is 45
Handy: increment operator / decrement operator

```c
#include <stdio.h>
int main()
{
    int X = 0;
    X++;   /* Now X is 1 */
    X++;   /* Now X is 2 */
    X--;   /* Now X is 1 again */
    printf("X is %d\n", X);
}
```

Hanks-iMac:Downloads hank$ gcc increment.c
Hanks-iMac:Downloads hank$ ./a.out
X is 1
For loop in practice
(with increment operator)

Hanks-iMac:Downloads hank$ cat sum.c
#include <stdio.h>
int main()
{
    int sum = 0;
    int i;
    for (i = 0 ; i < 10 ; i++)
    {
        sum = sum+i;
    }
    printf("Sum is %d\n", sum);
}
Hanks-iMac:Downloads hank$ gcc sum.c
Hanks-iMac:Downloads hank$ ./a.out
Sum is 45
Homework #2A: find numbers between 10 and 500 that are “odd” and “composite” (not-prime)

My program is creating this output these via printf() functions
CIS 212: Project #2A  
Assigned: October 4th, 2019  
Due: October 8th, 2019  
(which means submitted by 6am on October 9th, 2019)  
Worth 6% of your grade

Assignment:
1) Write a C program that finds the odd, composite numbers between 10 and 500. The name of the C program should be “project2A.c”
   a. Composite means it is not prime, i.e., “C” if composite if there exists an A and B such that A>1, B>1, and A*B = C.
   b. Odd means that it is not divisible by 2.
   c. Therefore “odd, composite”, means both odd and composite
2) Your program should have the exact same output as mine.
3) You can confirm this:
   a. Download “proj2A_checker”
   b. Run your program as “sh ./proj2A_checker”
   c. You can also see the correct output as “proj2A_correct_output”

This project will be graded by:
1) Running the proj1A_checker program
2) Inspection of your code

If the diff program shows any differences whatsoever, you will get less than half credit. This includes formatting differences, not having all the numbers, etc.
Further, this will be checked on the CIS 212 Virtual Box image.

If you write your code on your own machine (not on the VirtualBox): if your program produces the wrong output on that machine, then it will get less than half credit. **Explicitly:** it may run fine on your computer, but not on the VirtualBox. If this happens, you will receive less than half credit. Therefore, when you believe your program works, you should transfer it to the VirtualBox and test it there. This will be the case for all programs for the rest of this term, whether or not I include this information in the prompt.

**What should you upload?:** Just a single file, which is your C source code.

**Note:** I will not award much credit to programs that “cheat.” *I.e.*, if you work out all of the odd, composite numbers ahead of time and put them into a list, and then use that list of numbers, then you will not receive credit.
C constructs I needed

- For loops
- printf
- Modulo: the remainder from a division
  - (5%2) would return 1
- Assignment equal (=) and comparison equal (==)
How a Computer is **Not** Like Monopoly

- **Actions during cycle/turn:**
  - Monopoly: roll, buy, build, trade
  - Computer: other

- **Who’s turn?**
  - Monopoly: passes between players
  - Computer: always the computer’s turn

- **Time spent per cycle/turn:**
  - Monopoly: variable
  - Computer: fixed

- **Duration of turn:**
  - Monopoly: sometimes over a minute
  - Computer: much faster!!!
“First” computer: ENIAC

- Year: 1946
- Location: Pennsylvania
- Purpose: military
- Cost: $487K
  - ($6.9M today)
- Technology:
  - very different than today
  - ... but still the same
Vacuum Tubes

• Vacuum tubes:
  – Glass tubes with no gas
  – Used to control electron flow in early computers

• Occasionally, a bug would get stuck in the tube and cause the program to malfunction

• We no longer have vacuum tubes, but the term bug has remained with us...
An ENIAC Computation

• Used for military calculations:
  – A-bomb design
  – Missile delivery

• ENIAC could do ~5000 calculations in one minute

• In one case:
  – ENIAC did a calculation in 30 seconds
  – Human being took 20 hours
  – 2400x increase in speed

source: wikipedia
Hertz (Hz) = unit of measurement for how fast you do something

- 1 Hertz = do something once per second
- KHz = 1024 Hz
- MHz = 1024 KHz
- GHz = 1024 MHz

- The ENIAC machine ran at 5000 Hertz, or about 5 KHz.
  - Vocab term: “clock speed” → the number of cycles per second
    - the clock speed of the ENIAC was 5 KHz
Today’s Desktop Computers Are Fast!

- Most computers run at ~1-3 GHz
- i.e., operates billions of instructions each second
- This is about one million times faster than the ENIAC
  - ... and the ENIAC was 2400X faster than humans
  - (at least at tasks computers are good at)
What does a million-fold increase mean?

Distance: a 2” map of Oregon is 1:1,000,000 scale

Time: 1 second to 277 hours is 1:1,000,000 scale

Time: 1 minute to 694 days is 1:1,000,000 scale

Time: 1 hour to 114 years is 1:1,000,000 scale

Time: 1 day to 2738 years is 1:1,000,000 scale
1 million-fold increase! How does this happen?

- Moore’s Law (old timer’s version)
  - Clock speed doubles every 18 months
- Moore’s Law (newer version but still for old timers)
  - Clock speed doubles every 24 months
Moore’s Law

• Moore’s Law (actual version)
  – Number of transistors doubles every 24 months
  – And clock speed is a reflection of number of transistors

• So what is a transistor?
  – Semiconductor device for amplifying or switching electronic signals/power
  – Fundamental building block of modern electronics
  – Replacement for vacuum tube
Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.


The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic. Licensed under CC-BY-SA by the author Max Roser.
But actually…

The part about clock speed increasing with the number of transistors stopped about fifteen years ago.

Source: maximumpc.com
The reason is power

- Desktop computer takes ~200W
  - There are multiple components that consume the power:
    - CPU
    - Monitor
    - Disk
    - Memory
- 200W * 1 year → ~$70
Relationship Between Power and Clock Speed

• Clock goes twice as fast $\rightarrow$ Power goes up by factor of 8
  – (Increase of $X$ in clock speed $\rightarrow$ Increase of $X^3$ in power)

• Clock speeds haven’t changed in 12 years

• What if they had doubled every 2 years?
• Then $64X$ faster
  – $\rightarrow$ $262144X$ more power (for the CPU)
  – $\rightarrow$ power bill now $\$18M$
New vocab term: "core"

What Changed?
- We are getting double the transistors every two years
- ... but clock speed is the same
- ... so what is changing?

<table>
<thead>
<tr>
<th>CHOOSE YOUR OPTIMIZATION POINT</th>
<th>INTEGRATED</th>
<th>RECOMMENDED CUSTOMER PRICING</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CORES</td>
<td>GHZ</td>
</tr>
<tr>
<td>7290</td>
<td>72</td>
<td>1.5</td>
</tr>
<tr>
<td>Best Performance/Node</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7250</td>
<td>68</td>
<td>1.4</td>
</tr>
<tr>
<td>Best Performance/Watt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7230</td>
<td>64</td>
<td>1.3</td>
</tr>
<tr>
<td>Best Memory Bandwidth/Core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7210</td>
<td>64</td>
<td>1.3</td>
</tr>
<tr>
<td>Best Value</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Available beginning in September
2. Plus 15W for integrated fabric
3. Pricing shown is for parts without integrated fabric. Add additional $278 for integrated fabric versions of these parts. Integrated fabric parts available in October.

- Intel Xeon Silver
- Intel Xeon Gold
- Intel Xeon Platinum

M - an optional SKU is available with support for up to 1.5TB memory per CPU socket
F - an optional SKU is available with integrated 100Gbps Intel Omni-Path fabric
How To Use Multiple Cores?

• Answer: parallel programming
  – Write computer programs that use all the cores
  – Ideally the coordination between the cores is minimal
Parallel Programming Concepts

• Usual goal:
  – if program takes N seconds to run with one core
  – then take N/2 seconds to run with two cores
  – and N/M seconds to run with M cores

Let’s consider an example outside of computers
Example: paint a house

- One person: 6 days (1 day = 10 hours)
- Two people: 3 days
- Three people: 2 days
- Six people: 1 day

- Sixty people: 1 hour?
- Six hundred people: 6 minutes?
Example: paint a house, plan #3

- One person: paint one house in 6 days
- Two people: paint two houses in 6 days
- Three people: paint three houses in 6 days
- One thousand people: paint 1000 houses in 6 days?

Parallel programming is hard, and smart people spend their whole careers figuring out how to make parallel programs be efficient
GPUs: Graphical Processing Units (graphics cards)

- Historical:
  - Introduced to accelerate graphics (gaming!)
  - Boom with desktop PCs late 90s onward
  - Mid-2000’s: people start "hacking" interface to program a GPU to make it do things besides graphics
  - Late 2000’s: GPU makers jump on board and start encouraging folks to program GPUs directly

- GPGPU: General-purpose GPU programming
  - Mid 2010’s: GPUs used for *lots* of computing problems.
  - Machine learning workhorse!

---

**Emergent Tech**

**Bitcoin heist with a twist: This time it's servers that were stolen**

Icelandic cops cuff 11 on suspicion of data centre robberies

By Simon Sharwood, APAC Editor 5 Mar 2018 at 04:57

Icelandic police have cuffed 11 people in connection with four raids on data centres that targeted cryptocurrency mining equipment.

The raids started in December 2017 when three data centres were cracked in December. Another raid took place in January. 600 servers went missing in the heists.

Icelandic police kept the raids secret while they pursued their investigations. Those efforts culminated in 11 arrests and an appearance before the Reykjanes District Court last Friday. Two of the 11 were detained and the matter held over for another day.

The 600 servers are all still missing, the Associated Press reports. Which is no surprise: x86 kit is pretty generic. The real prize inside a bitcoin-mining rig is either GPUs, RAM or nicely fast solid-state disks. Those components are all tiny compared to servers and could probably have been posted out of Iceland piecemeal without much hassle.

Iceland has become something of a hub for demanding workloads like cryptocurrency mining because cheap energy and low ambient temperatures make it a low-cost destination to run data centres and the kit inside them. The nation also has a low crime rate. 🌍
Why Are GPUs So Good?

Market summary > NVIDIA Corporation
NASDAQ: NVDA - Mar 5, 7:59 PM EST

235.65 USD ↓0.89 (0.38%)
After-hours: 236.50 ↑0.36%

1 day | 5 day | 1 month | 3 month | 1 year | 5 year | max

12.82 Mar 8, 2013

Volta facts:
- 5120 cores
- 1200MHz clock speed
- Can do 2000X more operations than my laptop
- Suggested MSRP: $2,999.00

This level of increase in computation is not just a quantitative change, it is a qualitative one too.