Lecture 5:
Beginning on Memory
Part 3!!
& character strings
Hank’s OH

- Monday: great OH!
- Thursday: Viet will cover my OH
Still Orienting...

• In particular, I feel my responses to a few of your questions could have been better. Apologies.
Virtual Box

• How is it working?
• I propose avoiding “share”
Grading 2A

• Comments?
This week’s lab...

• If you “do memory wrong,” then your program will crash
• How to fix?
• Special program: gdb (GNU debugger)
  – gdb can give you lots of hints about why your program crashed!
Important note on debugging: buffering and printf

• Important: printf is buffered

• So:
  – printf puts string in buffer
  – other things happen
  – buffer is eventually printed

• But what about a crash?
  – printf puts string in buffer
  – other things happen ... including a crash
  – buffer is never printed!

One solution: fflush(stdout)
Assignment: Create a shell script that will create a directory structure and files within that directory structure, all with the specified file permissions. The script should be named “proj1b.sh”. (A consistent name will help with grading.)

Note: you are only allowed to use the following commands: mkdir, touch, cd, chmod, mv, cp, rm, rmdir. (You do not need to use all of these commands to successfully complete the assignment.)

Posted “how I would check my work if I were you” video last night, extended deadline one day.
Post ideas for YouTube videos

(1) how to validate 1B
(2) arrays -- repeat example from Thursday
(3) how to make a script
--> how is script different from gcc
mv: Unix command for renaming a file

```
C02LN00GFD58:212 hank$ mkdir tmp
C02LN00GFD58:212 hank$ cd tmp
C02LN00GFD58:tmp hank$ ls
C02LN00GFD58:tmp hank$ touch a
C02LN00GFD58:tmp hank$ ls
a
C02LN00GFD58:tmp hank$ mv a b
C02LN00GFD58:tmp hank$ ls
b
C02LN00GFD58:tmp hank$ 
```
Automatic vs Dynamic Memory

• In C, there are two types of variables
  – Actually three, but ignoring “static” for now
• Automatic: taken care of for you automatically
• Dynamic: you manage it

• So, of course, everyone wants automatic, right?
What is happening here?

```
#include <stdio.h>

int main()
{
    int A;
    A=4;
    printf("Value of A is %d\n", A);
    return 0;
}
```

Answer: “automatic” allocation of an integer. This integer has an address in memory. The address is in hexadecimal.
New operator: &

• & == address of
• This will tell you the address of a variable.

```c
#include <stdio.h>

int main()
{
    int A;
    A=4;
    printf("Value of A is %d, address of A is %p\n", A, &A);
    return 0;
}
```

```
Hanks-iMac:Downloads hank$ cat t.c
Hanks-iMac:Downloads hank$ gcc t.c
Hanks-iMac:Downloads hank$ ./a.out
Value of A is 4, address of A is 0x7fff56486bc8
```
Arrays

This is just 84 (4*21) bytes. We interpret these 84 bytes as 21 integers, following known conventions.

```c
#include <stdio.h>

int main()
{
    int A[21];
    int i;
    for (i = 0; i < 21; i++)
        A[i] = 3;
    printf("Value of A[10] is %d\n", A[10]);
}
```
What is an array?

• Block of contiguous memory
• If elements each have size N bytes and there are M elements, then N*M contiguous bytes
• Let A be address of the beginning of the array
• Then A[0] is at “A”
• And A[1] is at “A+N”
• and so on...
C02LN00GFD58:212 hank$ cat A.c
#include <stdio.h>

int main()
{
    int A[5];
    int i;

    printf("A is at \%p\n", A); /* Note I can pass in A as a pointer */
    for (i = 0; i < 5; i++)
    {
        printf("A[%d] is at \%p\n", i, &(A[i]));
    }
}
C02LN00GFD58:212 hank$ gcc A.c
C02LN00GFD58:212 hank$ ./a.out
A is at 0x7fff5e8b5bd0
A[0] is at 0x7fff5e8b5bd0
A[1] is at 0x7fff5e8b5bd4
A[2] is at 0x7fff5e8b5bd8
A[3] is at 0x7fff5e8b5bdc
A[4] is at 0x7fff5e8b5be0
Important Memory Concepts in C: Automatic Vs Dynamic

- You can allocate variables that only live for the invocation of your function
  - Called automatic/stack variables (will talk more about this later)

- You can allocated variables that live for the whole program (or until you delete them)
  - Called dynamic/heap variables (will talk more about this later as well)
#include <stdio.h>

void function1()
{
    int X = 4;
    printf("X is %d\n", X);
    /* X goes out of scope and the end of this function and thus
     * *automatically* disappears and is no longer accessible */
}

int main()
{
    int Z = 6;
    function1();

    /* NEW SCOPE */
    {
        int Y = 5;
    }
    /* Y no longer exists. Automatically allocated and de-allocated */

    /* only valid variable here is Z ... not X, not Y. */
    printf("Z is %d\n", Z);

    /* Now Z goes away. */
}
Dynamic memory works differently

• You allocate it, it stays around until you de-allocate it or the program ends

• Important: you need a way to keep track of memory
  – If not, the memory will be “leaked.”

• So we need a way of managing dynamic memory.

• The concept for doing this in C is **POINTERS**
Pointers

• Pointer: points to memory location
  – Denoted with ‘*’
  – Example: “int *p”
    • pointer to an integer
  – You need pointers to get to dynamic memory

• Address of: gets the address of memory
  – Operator: ‘&’
  – Example:
    ```
    int x;
    int *y = &x;  // this example is pointing to an automatic variable, not a dynamic variable
    ```
int main()
{
    int A[5];
    int i;

    printf("A is at %p\n", A); /* Note I can pass in A as a pointer */
    for (i = 0 ; i < 5 ; i++)
    {
        printf("A[%d] is at %p\n", i, &(A[i]));
    }
}
Dynamic Memory Allocation

• Special built-in function to allocate dynamic memory: `malloc`
  – Interacts with Operating System
  – Argument for malloc is how many bytes you want

• Also built-in function to deallocate memory: `free`
free/malloc example

Enables compiler to see functions that aren’t in this file. More on this next week.

```c
#include <stdlib.h>
int main()
{
    /* allocates memory */
    int *ptr = malloc(2*sizeof(int));

    /* deallocates memory */
    free(ptr);
}
```
Automatic vs Dynamic

• Automatic memory lives only for its current scope
• Dynamic memory lives until you free it, or until the program ends
This is just fine...

Hanks-iMac:Downloads hank$ cat scope.c
#include <stdio.h>
#include <stdlib.h>

int *foo1()
{
    int *X = malloc(sizeof(int)*2);
    X[0] = 1;
    X[1] = 2;
    return X;
}

int main()
{
    int *Y = foo1();
    printf("Y[0] is %d and Y[1] is %d\n", Y[0], Y[1]);
}

Hanks-iMac:Downloads hank$ gcc scope.c
Hanks-iMac:Downloads hank$ ./a.out
Y[0] is 1 and Y[1] is 2
This is not fine...

```c
#include <stdio.h>
#include <stdlib.h>

int *foo1()
{
    int X[2];
    X[0] = 1;
    X[1] = 2;
    return X;
}

int main()
{
    int *Y = foo1();
    printf("Y[0] is %d and Y[1] is %d\n", Y[0], Y[1]);
}
```

```
Hanks-iMac:Downloads hank$ cat bad_scope.c
#include <stdio.h>
#include <stdlib.h>

int *foo1()
{
    int X[2];
    X[0] = 1;
    X[1] = 2;
    return X;
}

int main()
{
    int *Y = foo1();
    printf("Y[0] is %d and Y[1] is %d\n", Y[0], Y[1]);
}
```

```
Hanks-iMac:Downloads hank$ gcc bad_scope.c
bad_scope.c:9:12: warning: address of stack memory associated with local variable 'X' returned [-Wreturn-stack-address]
    return X;
```

1 warning generated.

```
Hanks-iMac:Downloads hank$ ./a.out
Y[0] is 1 and Y[1] is 2
```
And here it goes wrong...
Memory Segments

• Your program is divided into 4 segments:
  – Code segment: where the machine code instructions are
  – Data segment: where the global variables live ... and other things
  – “Stack”: where automatic memory lives
  – “Heap”: where dynamic memory lives
  – Note: stack & heap are data structures and we will learn more about them later.

If you access memory addresses outside your segments, you get a “segmentation fault” ... which causes a crash.
And in some cases it doesn’t have to hit the heap. Just take more memory than intended.
Character Strings
There have been various extensions to ASCII ... now more than 128 characters
Many special characters are handled outside this convention

image source: granneman.com
printf

Hi Everyone,

So it came out in class today that I had been assuming folks knew more about printf than they did. I thought its Python cousin was closer than it is. Sorry for that!!

First, printf is a function. You call printf and it prints stuff; that is probably obvious.

The next thing is the syntax.

printf(STRING_TO_PRINT, arg1, arg2, arg3, ..., argN);

The trick is that STRING_TO_PRINT will contain directions about how to incorporate arg1, arg2, arg3, etc.

For example,
printf("This is 4: %d\n", 4);
will print "this is 4: 4"
The \n is a new line -- it makes the print statement go to the next line.
A % tells printf to bring in the next argument. It would start with arg1, then arg2, etc.
The character that follows the % the type of the argument. So far, we only know 2 types:
%d: the argument is an integer
%p: the argument is a pointer
Some Special Characters

```c
#include <stdio.h>

int main()
{
    char A = 'A';
    char B = 'B';
    char NL = '\n';
    char tab = '\t';
    char backslash = '\\';
    char NULL_character = '\0';
    printf("A = %d, B = %d, NL = %d, tab =%d, backslash = %d, NULL char = %d\n",
            A, B, NL, tab, backslash, NULL_character);
    printf("A = %c, B = %c, NL = %c, tab =%c, backslash = %c, NULL char = %c\n",
            A, B, NL, tab, backslash, NULL_character);
}
```

New: `%c` prints character

C02LN00GFD8:212 hank$ cat char.c
C02LN00GFD8:212 hank$ gcc char.c
C02LN00GFD8:212 hank$ ./a.out
A = 65, B = 66, NL = 10, tab =9, backslash = 92, NULL char = 0
A = A, B = B, NL = 
, tab = , backslash = \, NULL char =
character strings

• A character “string” is:
  – an array of type “char”
  – that is terminated by the NULL character

• The C library has multiple functions for handling strings
building a character string

```c
#include <stdio.h>

int main()
{
    char H[12];
    printf("String is %s\n", H);
}
```

New: %s prints “strings” (arrays of chars terminated by \0)
building a character string

C02LN00GFD58:212 hank$ cat hello_world2.c
#include <stdio.h>

int main()
{
    char H[12] = "hello world";
    printf("H is %s\n", H);
}

C02LN00GFD58:212 hank$ gcc hello_world2.c
C02LN00GFD58:212 hank$ ./a.out
H is hello world
C02LN00GFD58:212 hank$ cat hello_world2.c
#include <stdio.h>

int main()
{
    char H[12] = "hello world";
    printf("H is %s\n", H);
    H[5] = '\0';
    printf("H is %s\n", H);
}
C02LN00GFD58:212 hank$ gcc hello_world2.c
c02ln00gfd58:212 hank$ ./a.out
H is hello world
H is hello
Did you notice

- Characters in C are single quotes: ‘A’
- Strings in C are double quotes: “hello world”
- A string of a single character is still double quotes: “A”
  - (And this is actually “A\0”)
```c
#include <stdio.h>

int main()
{
    char H[12] = "hello world";
    printf("H is %s\n", H);
}
```
Character strings example

```c
#include <stdio.h>

int main()
{
    char str[12] = "hello world";
    char *str2 = str+6;

    printf("str is \"%%s\" and str2 is \"%%s\"\n", str, str2);

    str[5] = '\0';

    printf("Now str is \"%%s\" and str2 is \"%%s\"\n", str, str2);
}
```

128-223-223-72-wireless:330 hank$ cat string.c
128-223-223-72-wireless:330 hank$ gcc string.c
128-223-223-72-wireless:330 hank$ ./a.out
str is "hello world" and str2 is "world"
Now str is "hello" and str2 is "world"
Useful C library string functions

- `strcpy`: string copy
- `strncpy`: string copy, but just first N characters
- `strlen`: length of a string

```c
#include <string.h>
#include <stdio.h>

int main()
{
    char str[12] = "hello world";
    char str2[6], str3[7];
    strcpy(str2, str+strlen("hello "));
    strncpy(str3, str, strlen("hello "));
    printf("%s,%s\n", str2, str3);
}
```
Useful C library string functions

- `strcpy`: string copy
- `strncpy`: string copy, but just first N characters
- `strlen`: length of a string

```c
#include <string.h>
#include <stdio.h>

int main()
{
    char str[12] = "hello world";
    char str2[7], str3[6];
    strcpy(str2, str + strlen("hello "));
    strncpy(str3, str, strlen("hello "));
    printf("%s,%s\n", str2, str3);
}
```

What happened here?

```
128-223-223-72-wireless:330 hank$ cat strcpy.c
#include <string.h>
#include <stdio.h>

int main()
{
    char str[12] = "hello world";
    char str2[7], str3[6];
    strcpy(str2, str + strlen("hello "));
    strncpy(str3, str, strlen("hello "));
    printf("%s,%s\n", str2, str3);
}
128-223-223-72-wireless:330 hank$ gcc strcpy.c
128-223-223-72-wireless:330 hank$ ./a.out
world,hello world
```
# More useful C library string functions

## Functions

### Copying:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>memcpy</code></td>
<td>Copy block of memory (<a href="https://cplusplus.com">function</a>)</td>
</tr>
<tr>
<td><code>memmove</code></td>
<td>Move block of memory (<a href="https://cplusplus.com">function</a>)</td>
</tr>
<tr>
<td><code>strcpy</code></td>
<td>Copy string (<a href="https://cplusplus.com">function</a>)</td>
</tr>
<tr>
<td><code>strncpy</code></td>
<td>Copy characters from string (<a href="https://cplusplus.com">function</a>)</td>
</tr>
</tbody>
</table>

### Concatenation:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strcat</code></td>
<td>Concatenate strings (<a href="https://cplusplus.com">function</a>)</td>
</tr>
<tr>
<td><code>strncat</code></td>
<td>Append characters from string (<a href="https://cplusplus.com">function</a>)</td>
</tr>
</tbody>
</table>

### Comparison:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>memcmp</code></td>
<td>Compare two blocks of memory (<a href="https://cplusplus.com">function</a>)</td>
</tr>
<tr>
<td><code>strcmp</code></td>
<td>Compare two strings (<a href="https://cplusplus.com">function</a>)</td>
</tr>
<tr>
<td><code>strcoll</code></td>
<td>Compare two strings using locale (<a href="https://cplusplus.com">function</a>)</td>
</tr>
<tr>
<td><code>strncmp</code></td>
<td>Compare characters of two strings (<a href="https://cplusplus.com">function</a>)</td>
</tr>
<tr>
<td><code>strxfrm</code></td>
<td>Transform string using locale (<a href="https://cplusplus.com">function</a>)</td>
</tr>
</tbody>
</table>

### Searching:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>memchr</code></td>
<td>Locate character in block of memory (<a href="https://cplusplus.com">function</a>)</td>
</tr>
<tr>
<td><code>strchr</code></td>
<td>Locate first occurrence of character in string (<a href="https://cplusplus.com">function</a>)</td>
</tr>
<tr>
<td><code>strcspn</code></td>
<td>Get span until character in string (<a href="https://cplusplus.com">function</a>)</td>
</tr>
<tr>
<td><code>strpbrk</code></td>
<td>Locate characters in string (<a href="https://cplusplus.com">function</a>)</td>
</tr>
<tr>
<td><code>strrchr</code></td>
<td>Locate last occurrence of character in string (<a href="https://cplusplus.com">function</a>)</td>
</tr>
<tr>
<td><code>strspn</code></td>
<td>Get span of character set in string (<a href="https://cplusplus.com">function</a>)</td>
</tr>
<tr>
<td><code>strstr</code></td>
<td>Locate substring (<a href="https://cplusplus.com">function</a>)</td>
</tr>
<tr>
<td><code>strtok</code></td>
<td>Split string into tokens (<a href="https://cplusplus.com">function</a>)</td>
</tr>
</tbody>
</table>

### Other:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>memset</code></td>
<td>Fill block of memory (<a href="https://cplusplus.com">function</a>)</td>
</tr>
<tr>
<td><code>strerror</code></td>
<td>Get pointer to error message string (<a href="https://cplusplus.com">function</a>)</td>
</tr>
<tr>
<td><code>strlen</code></td>
<td>Get string length (<a href="https://cplusplus.com">function</a>)</td>
</tr>
</tbody>
</table>

## Macros

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>NULL</code></td>
<td>Null pointer (<a href="https://cplusplus.com">macro</a>)</td>
</tr>
</tbody>
</table>

## Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>size_t</code></td>
<td>Unsigned integral type (<a href="https://cplusplus.com">type</a>)</td>
</tr>
</tbody>
</table>
argv & argc

- two arguments to every C program
- argc: how many command line arguments
- argv: an array containing each of the arguments
- "./a.out hank childs"
- \( \rightarrow \) argc == 3
int main(int argc, char *argv[]) {
    int i;
    for (i = 0; i < argc; i++) {
        printf("Argument %d is %s\n", i, argv[i]);
    }
}

C02LN00GFD58:212 hank$ gcc argv.c
C02LN00GFD58:212 hank$ ./a.out hank childs loves C and vi
Argument 0 is ./a.out
Argument 1 is hank
Argument 2 is childs
Argument 3 is loves
Argument 4 is C
Argument 5 is and
Argument 6 is vi
Bonus Slides
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 5KHz.
  - Vocab term: “clock speed” → the number of cycles per second
    - the clock speed of the ENIAC was 5KHz
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...

Source: maximumpc.com
The reason is power

• Desktop computer takes ~200W
  – There are multiple components that consume the power:
    • CPU
    • Monitor
    • Disk
    • Memory

• $200W \times 1\ year \rightarrow \sim$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 64$X$ faster
  – $\rightarrow$ 262144$X$ 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?

CHOOSE YOUR OPTIMIZATION POINT

<table>
<thead>
<tr>
<th></th>
<th>INTEGRATED</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORES</td>
<td>GHZ</td>
</tr>
<tr>
<td>7290¹</td>
<td>72</td>
</tr>
<tr>
<td>Best Performance/Node</td>
<td></td>
</tr>
<tr>
<td>7250</td>
<td>68</td>
</tr>
<tr>
<td>Best Performance/Watt</td>
<td></td>
</tr>
<tr>
<td>7230</td>
<td>64</td>
</tr>
<tr>
<td>Best Memory Bandwidth/Core</td>
<td></td>
</tr>
<tr>
<td>7210</td>
<td>64</td>
</tr>
<tr>
<td>Best Value</td>
<td></td>
</tr>
</tbody>
</table>

¹Available beginning in September ²Plus 15W for integrated fabric
³Pricing 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 #2:

- 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
  – Boom with desktop PCs
  – Mid-2000’s: people start
    program a GPU to make
  – Late 2000’s: GPU makers
    encouraging folks to program
    a GPU to do things besides
    graphics
  – GPGPU: General-purpose
  – Mid 2010’s: GPUs used
    for* lots* of computing
    problems.
  – Machine learning workhorse!
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