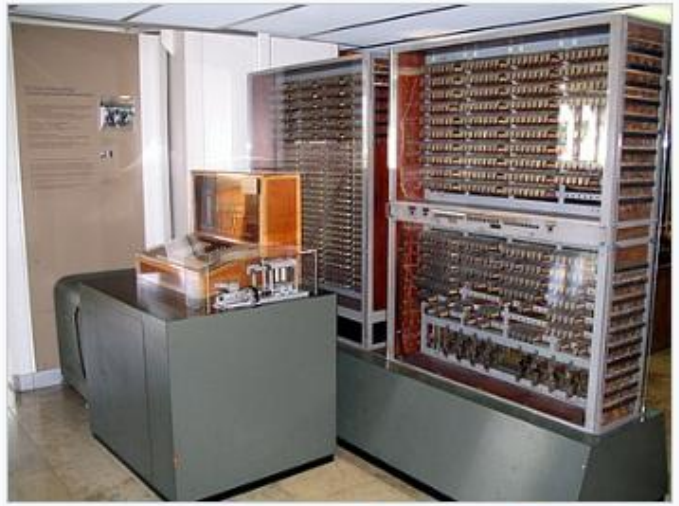
DTS205TC High Performance Computing

Lecture 2 Infrastructure 2

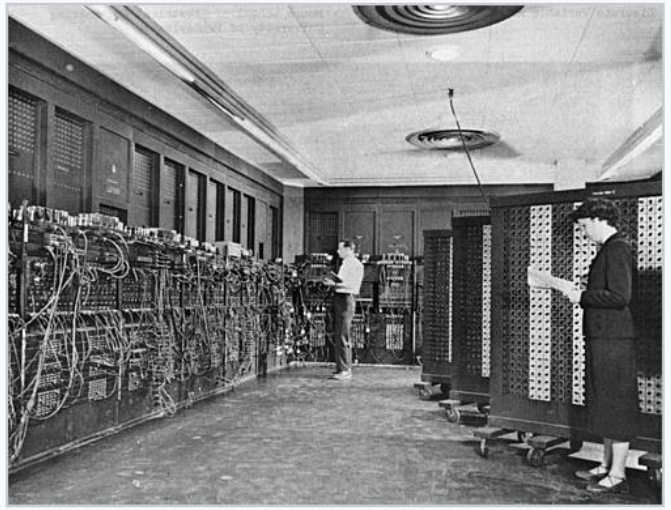
Di Zhang, Spring 2024

History of computers



Replica of Konrad Zuse's Z3, the first fully automatic, digital (electromechanical) computer

1941, Germeny

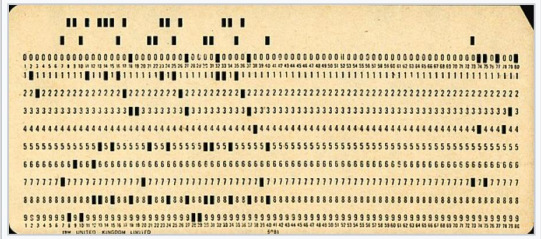


ENIAC was the first electronic, Turing-complete device, and performed ballistics trajectory calculations for the United States Army.

1945, US

A section of the reconstructed Manchester Baby, the first electronic stored-program computer

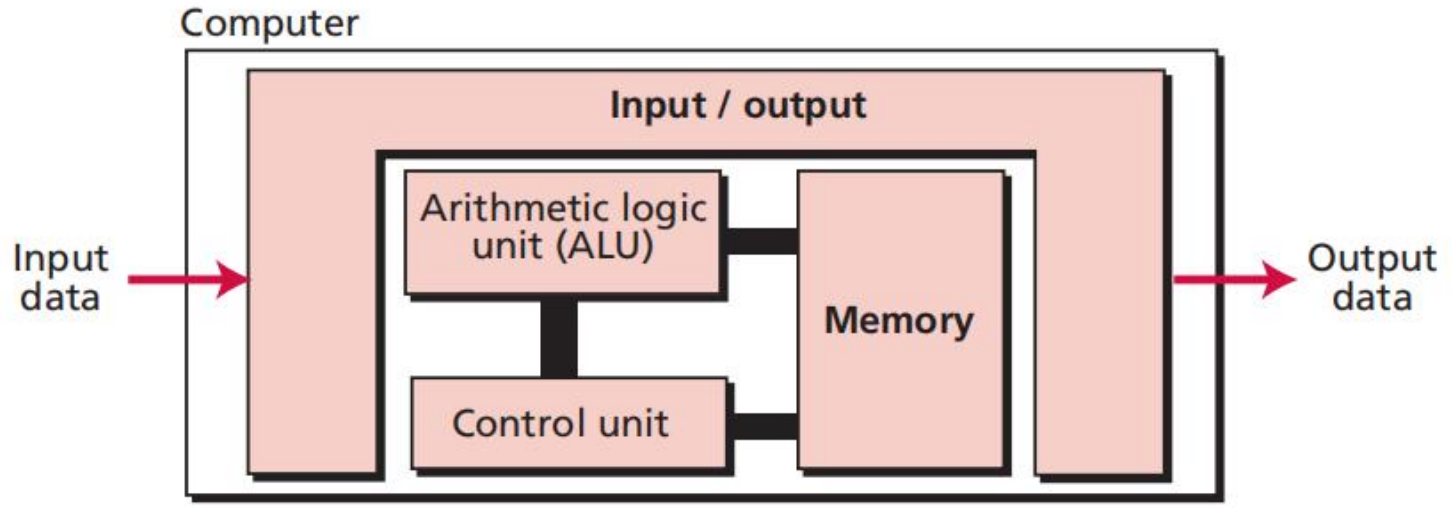
1948, UK



A 12-row/80-column IBM punched card from the mid-twentieth century

Von Neumann Model

Figure 1.5 *The Von Neumann model*



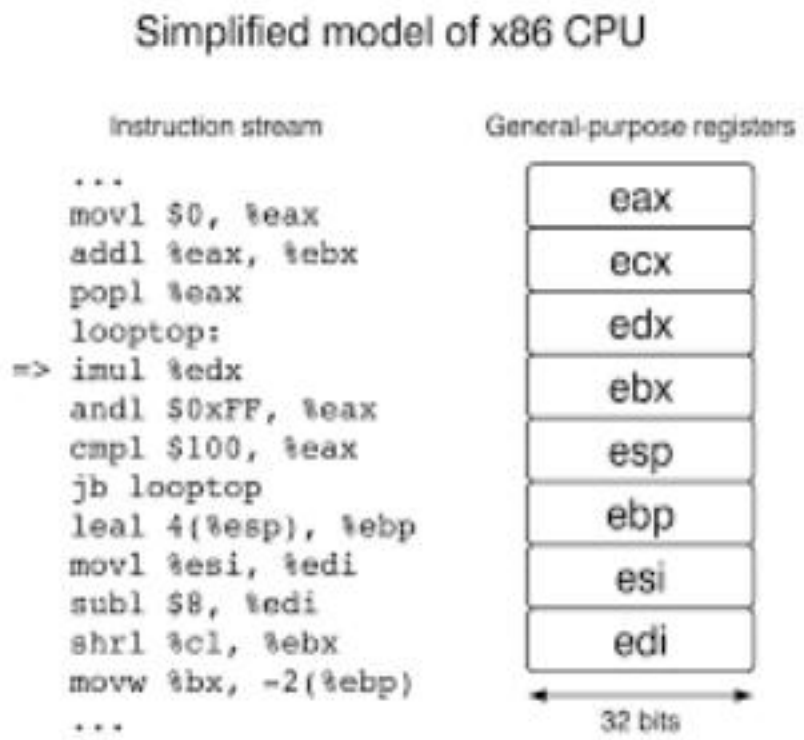
- Programs must be stored
- Sequential execution of instructions

Components

- ALU
- Memory
- Control unit
- Input / output

*Turing and Von Neumann

- A set of fixed instructions or a programming language is enough



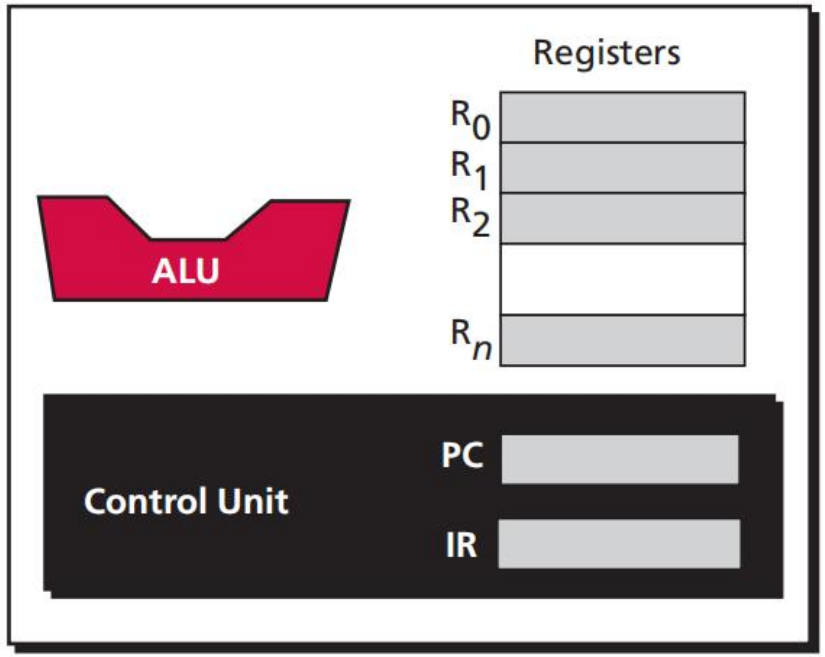
Demo

- https://peterhigginson.co.uk/lmc/?F5=12-Feb-24_14:04:09

- https://justine.lol/ape.html

Central processing unit (CPU)

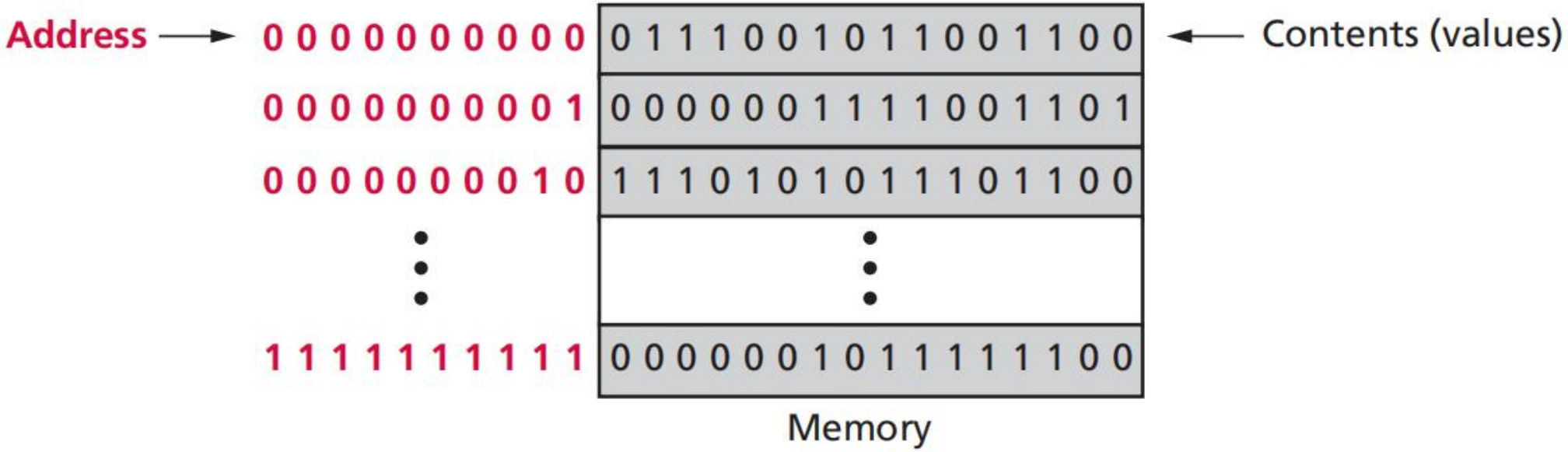
Figure 5.2 *Central processing unit (CPU)*



R: Register
PC: Program Counter
IR: Instruction Register

Main memory

Figure 5.3 *Main memory*

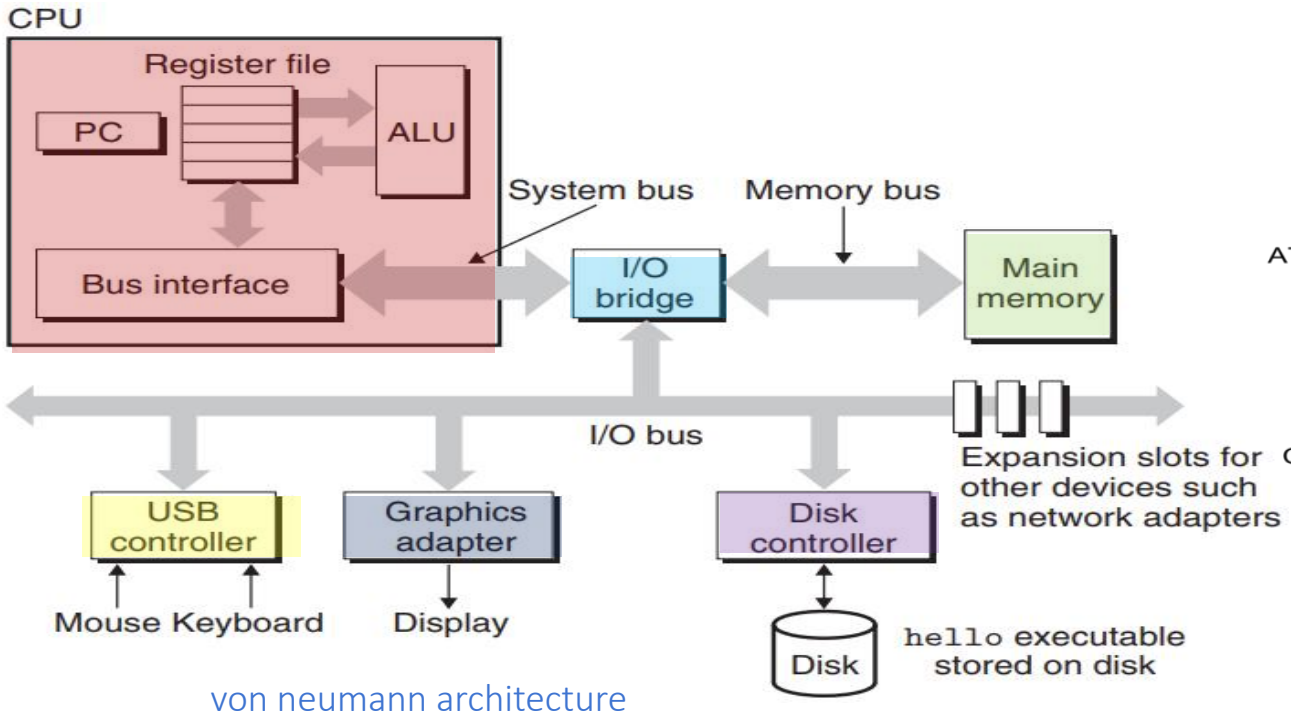


Memory types

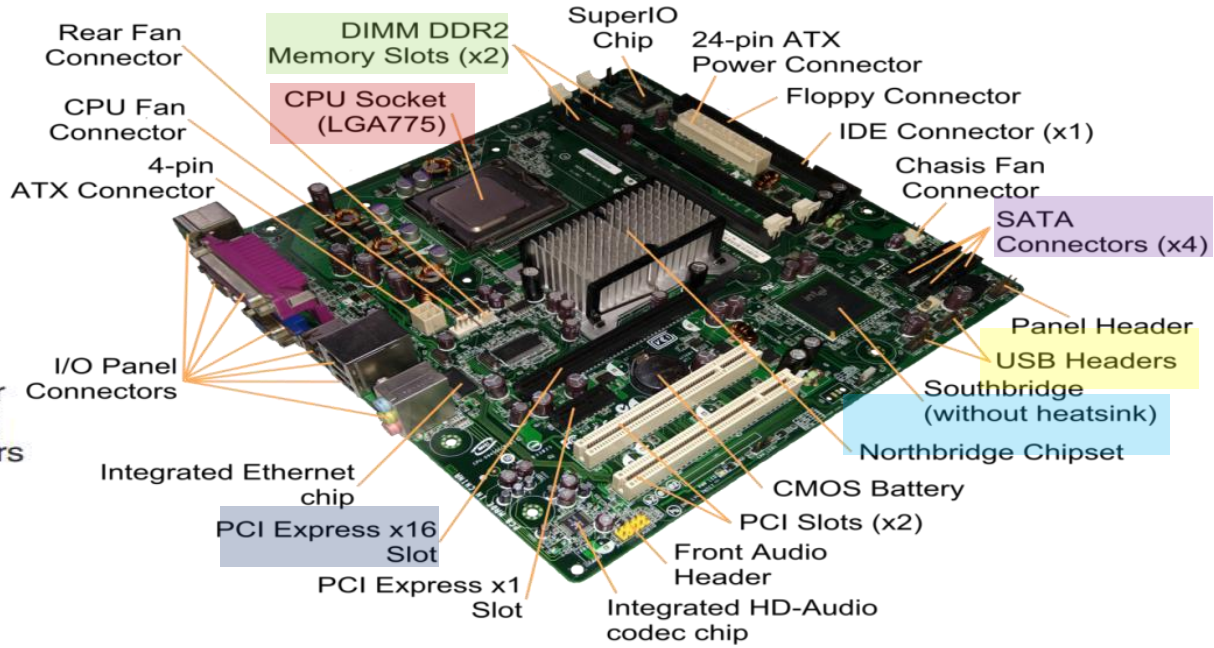
- ROM

- RAM

Structure of computer



von neumann architecture



Intel D945GCCR

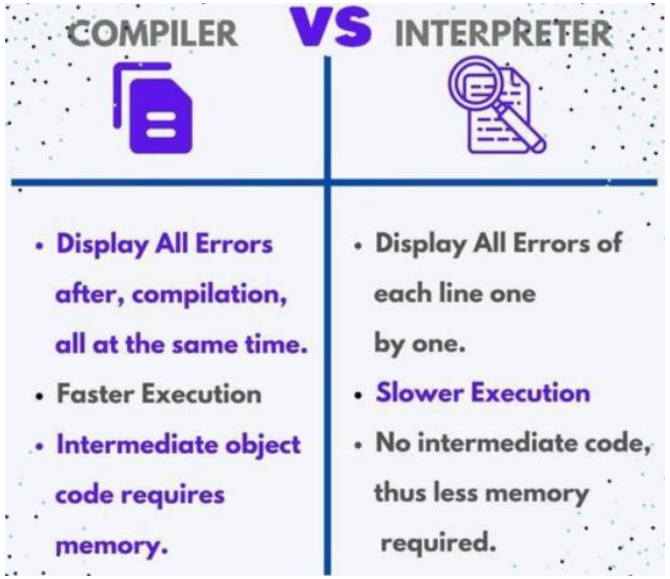
- The main components (review of 103) :
  - CPU
  - I/O
  - Memory
  - Graphic card

Doom: PC Graphics Benchmark

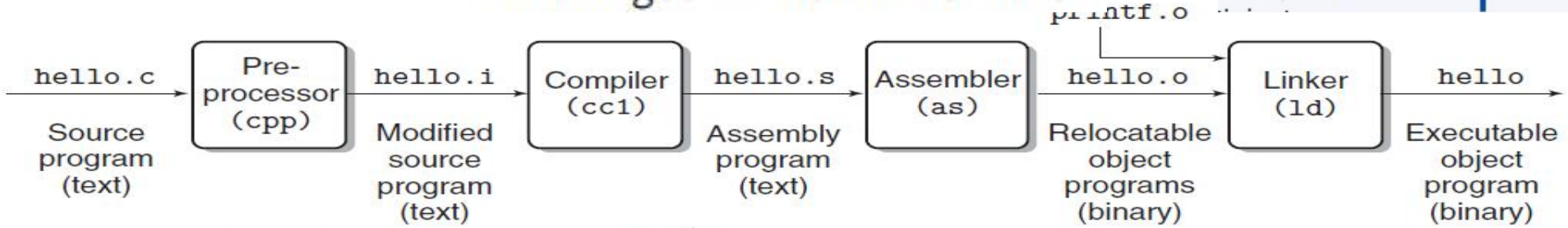
Example: Hello world


```c
#include <stdio.h>

int main()
{
    printf("hello, world\n");
}
```




```
linux> gcc -o hello hello.c
```

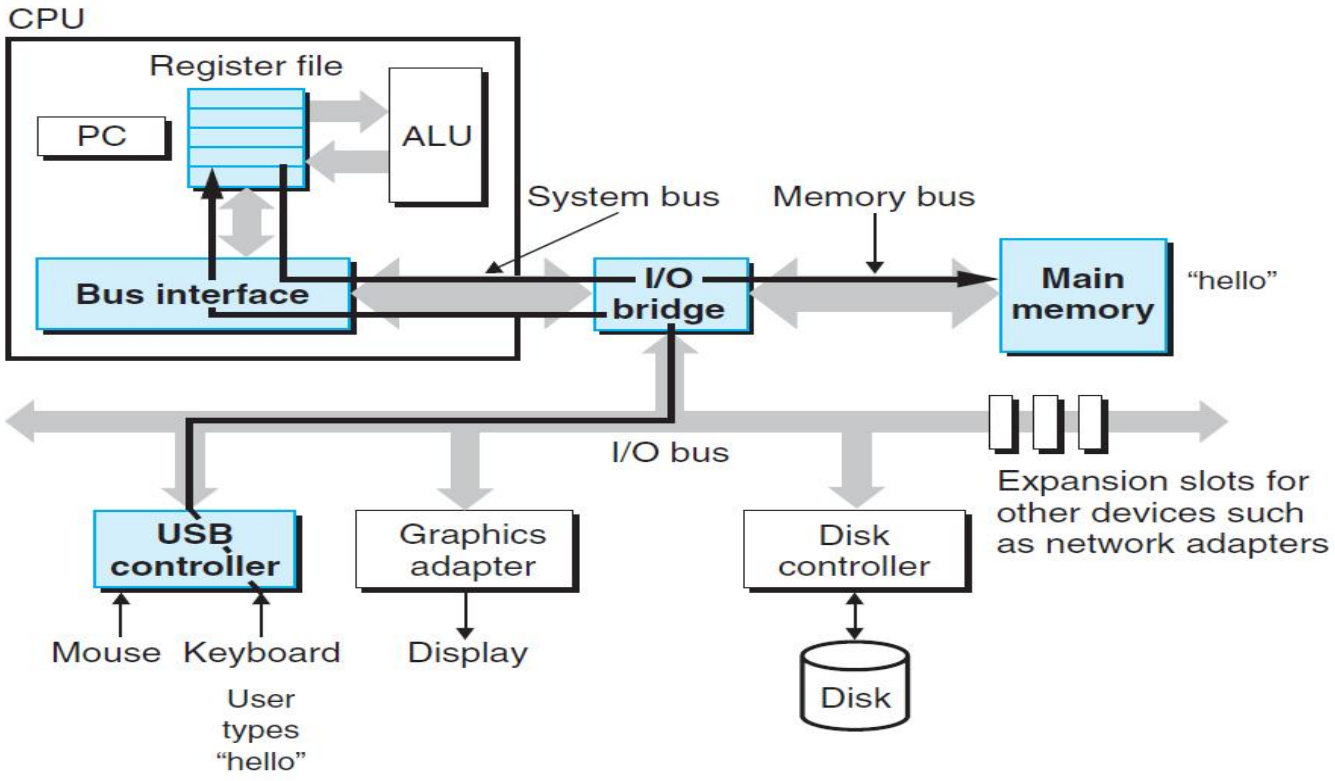


Dealing with #include

```
main:
    subq    $8, %rsp
    movl    $.LC0, %edi
    call    puts
    movl    $0, %eax
    addq    $8, %rsp
    ret
```

Q: Anyone knows differences between C and Python?

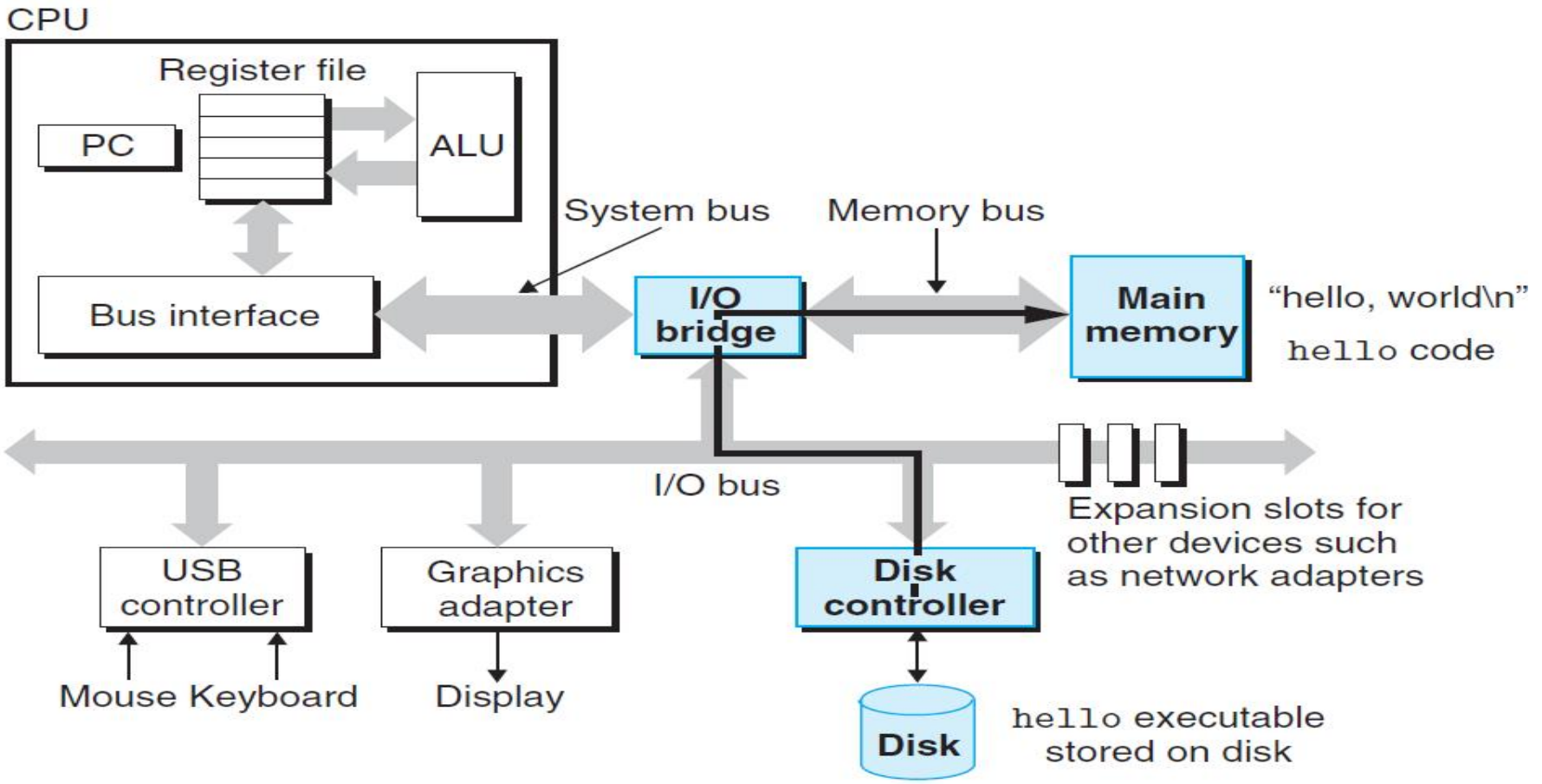
Example: Hello world




```
unix> ./hello
hello, world
unix>
```

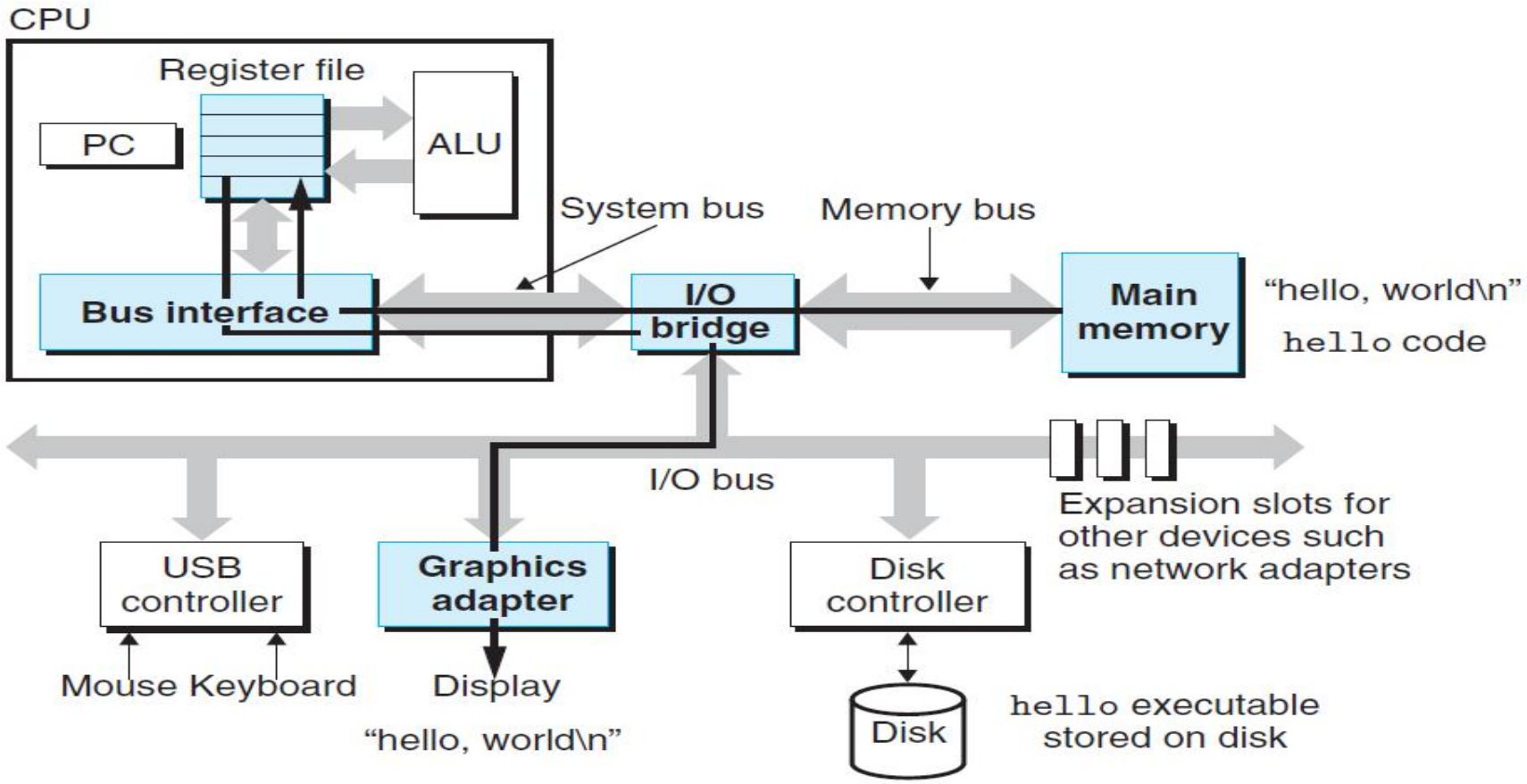
1. Shell reads the hello command into memory from the keyboard.

Example: Hello world



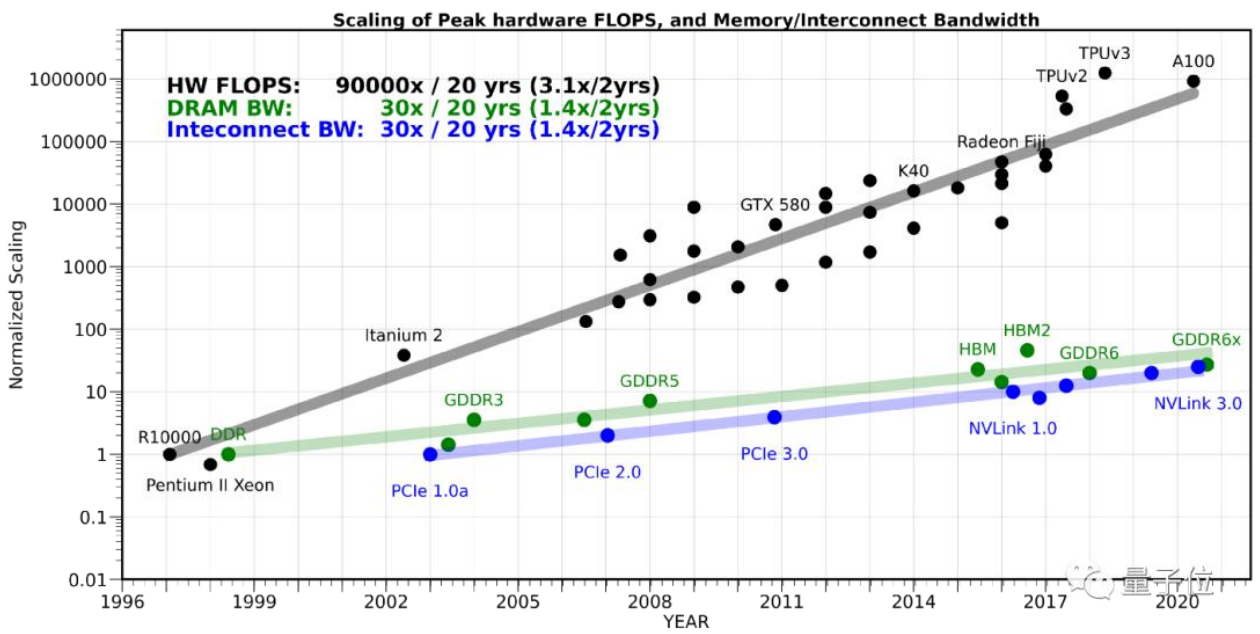
2. OS loads the executable from disk into main memory.

Example: Hello world



3. The OS prepares the program, which then executes itself and writes the output string from memory to the display.

Speed Mismatch: Memory Wall



Latency Comparison Numbers (~2012)					
L1 cache reference	0.5	ns			
Branch mispredict	5	ns			
L2 cache reference	7	ns			14x L1 cache
Mutex lock/unlock	25	ns			
Main memory reference	100	ns			20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy	3,000	ns	3 us		
Send 1K bytes over 1 Gbps network	10,000	ns	10 us		
Read 4K randomly from SSD*	150,000	ns	150 us		~1GB/sec SSD
Read 1 MB sequentially from memory	250,000	ns	250 us		
Round trip within same datacenter	500,000	ns	500 us		
Read 1 MB sequentially from SSD*	1,000,000	ns	1,000 us	1 ms	~1GB/sec SSD, 4X memory
Disk seek	10,000,000	ns	10,000 us	10 ms	20x datacenter roundtrip
Read 1 MB sequentially from disk	20,000,000	ns	20,000 us	20 ms	80x memory, 20X SSD
Send packet CA->Netherlands->CA	150,000,000	ns	150,000 us	150 ms	

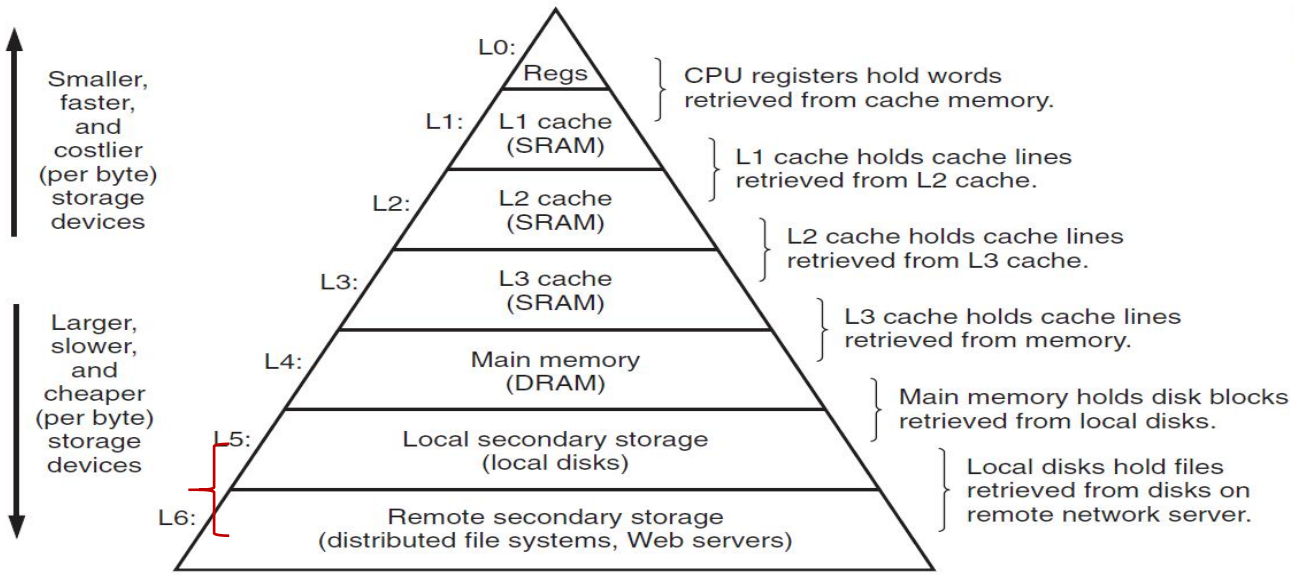
Notes
1 ns = 10^-9 seconds
1 us = 10^-6 seconds = 1,000 ns
1 ms = 10^-3 seconds = 1,000 us = 1,000,000 ns

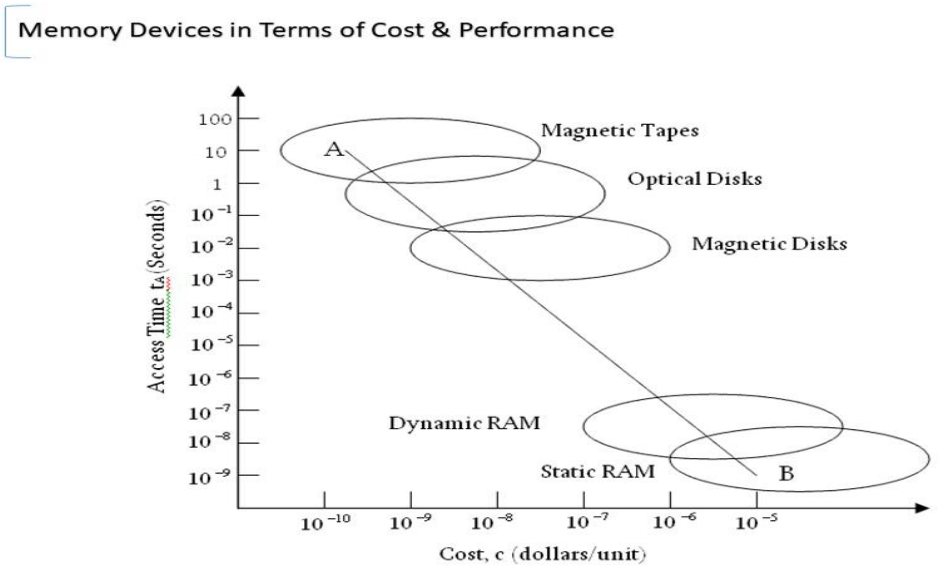
GFLOPS=Giga (10^9) Floating-point Operations Per Second

- Problem: In order to read and write data, the CPU must wait for the memory

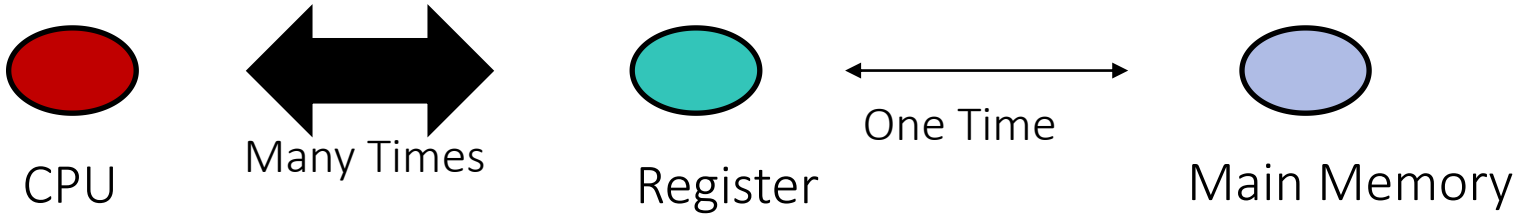
An Example of Speed Mismatching

Caches matter





They can be cache at bottom!



- Register: fast but expensive
- The role of cache: play the role of agent; based on the principle of locality, it can speed up the visiting
- Hierarchical cache system: Multi-Level Acceleration

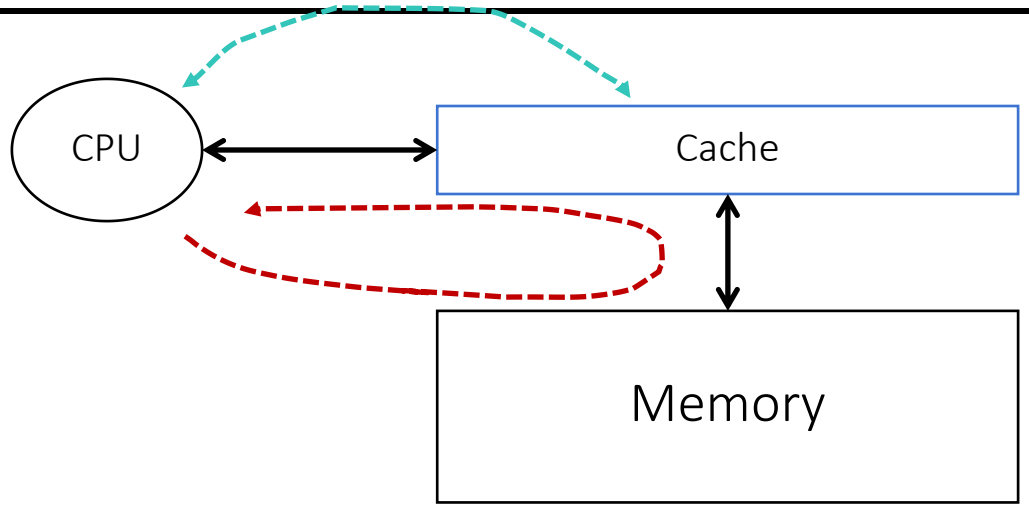
Example: Summing 1-D Arrays

- Consider summing an array 'int a[N]'

```
// Loop Over, and add every item to Addition
for(i = 0; i < Size; i ++)
{
    Addition = Addition + a[i];
}
```

- It requires N-1 *additions*, as well as 2*(N-1) *reads* and N-1 *writes*
- To estimate running time, in classic algorithm theory (ref. 203), we only count the number of operations
- However, if the read and write time is a hundred times the calculation time, then it must be considered
- Without cache, it needs cycle time:
  - N-1 + 100* 3*(N-1)
- With cache and have all data put into it
  - N-1 + 0.5* 3*(N-1)
- When N=10,000, the latter is 0.0083 (<1%) of the former!

Cache Policy



- A **Policy** needs to be defined to keep the access to Memory in the cache as much as possible

$$T = m \times T_m + T_h$$

- m： cache miss rate
- T_m: memory access time
- T_h: cache access time

- → the lower m the better

First In First Out (FIFO)

Data Addr.	A	B	B	E	F	B	C	B	E
	A	A	A	**E**	E	E	E	E	**E**
	B	**B**	**B**	B	**F**	F	F	F	F
	C	C	C	C	C	**B**	B	**B**	B
	D	D	D	D	D	D	**C**	C	C

- Missing rate = 4/9
- Always replace the earliest entry in the cache
- This part will be Lab 1

Least recently used (LRU)

Data Addr.	A	B	B	E	F	B	C	B	E
	A(0)	A	A	A	A	A	C(7)	C	C
	B(-3)	B(2)	B(3)	B	B	B(6)	B	B(8)	B
	C(-2)	C	C	E(4)	E	E	E	E	E(9)
	D(-1)	D	D	D	F(5)	F	F	F	F

- Missing rate = 3/9
- This strategy requires keeping a "date of birth" for cache lines and tracking the "least recently used" (oldest) cache line based on it
- The difference between LRU and FIFO is that Processor's access to a record will update its "date of birth", while FIFO does not

Random Replacement Policy

- Steps:
  - If the cache is not full, save the most recently accessed data in the cache
  - If it is full, replace one of them at random.

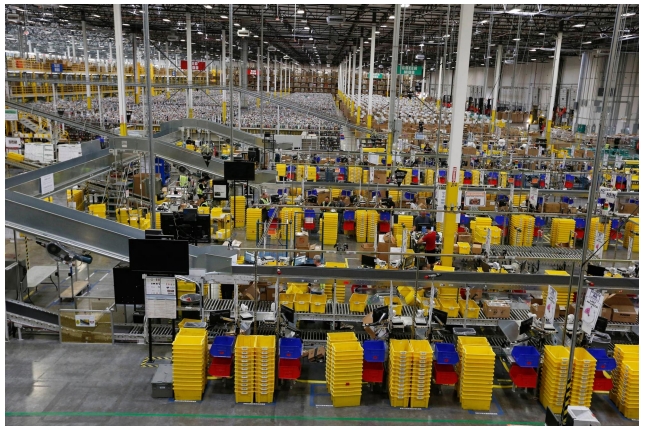
  - Simple and stable, once used in ARM processors

Other Policies

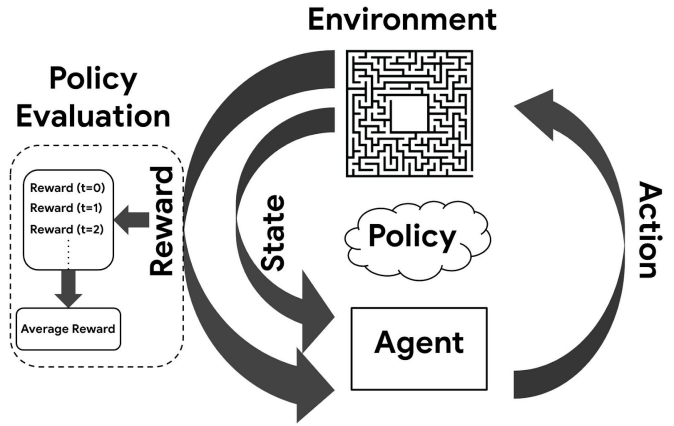
- LIFO
- More policies can be found:
- https://en.wikipedia.org/wiki/Cache_replacement_policies

Further Reading

- Cache is everywhere
  - Web Content
  - Video Broadcast
  - Logistic/warehousing
- Can it be solved by ML? Yes.



Amazon Logistics



Policy in Reinforcement Learning

EXPERIMENT

Cache Hit Ratio Optimization

ST Stefano Tempesta • March 13, 2018

Be the first to like.

Summary

Optimize cache hit ratios and reduce "miss rates" with regression algorithms processed by machine learning.

Performance optimization

- Choose the right algorithm and data structure
  - Go to Course 203
- Choose the right language (C is usually the fastest, but also hard to maintain)
  - Go to Course 102
- Choose compiler
  - icc vs gcc
  - gcc compiler option –O3
  - For python, switch to Cython
- Loop optimization
  - Code movement: extract repeated calculations out of the loop

Performance optimization (serial)


```
1   /* Move call to vec_length out of loop */
2   void combine2(vec_ptr v, data_t *dest)
3   {
4       long int i;
5       long int length = vec_length(v);
6
7       *dest = IDENT;
8       for (i = 0; i < length; i++) {
9           data_t val;
10          get_vec_element(v, i, &val);
11          *dest = *dest OP val;
12      }
13  }
```

- Pay attention to the implicit loop: the linked list must traverse itself when finding the length

Performance optimization

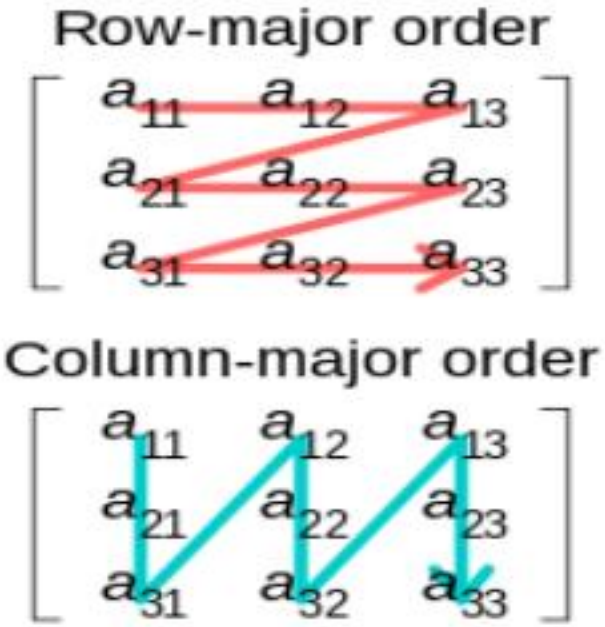
- Write temporary variables inside the loop instead of global variables - compiler can put temporary variables completely in registers without writing back to memory

Row-major order is used in C/C++/Objective-C (for C-style arrays), PL/I,[4] Pascal,[5] Speakeasy,[*citation needed*] and SAS.[6]

Column-major order is used in Fortran, MATLAB,[7] GNU Octave, Julia,[8] S, S-PLUS,[9] R,[10] Scilab,[11] Yorick, and Rasdaman.[12]

```
void suma(int rows, int cols, long data[][cols], long *ptotal) {
  long total = 0;
  for (int i = 0; i < rows; i++)
    for (int j = 0; j < cols; j++)
      total += data[i][j];
  *ptotal = total;
}
```

```
void sumb(int rows, int cols, long data[][cols], long *ptotal) {
  long total = 0;
  for (int j = 0; j < cols; j++)
    for (int i = 0; i < rows; i++)
      total += data[i][j];
  *ptotal = total;
}
```



Performance optimization (serial)


```
a@a: ~

Performance counter stats for './row 10 a' (100 runs):

             26.80 msec task-clock                #    0.988 CPUs utilized            ( +-  0.10% )
                 0      context-switches          #    0.000 /sec
                 0      cpu-migrations            #    0.000 /sec
             7,135      page-faults               #  266.118 K/sec                    ( +-  0.00% )
        73,568,049      cycles                    #    2.744 GHz                      ( +-  0.08% )  (40.30%)
       114,269,204      instructions              #    1.56  insn per cycle           ( +-  0.06% )  (55.22%)
        16,434,432      branches                  #  612.963 M/sec                    ( +-  0.80% )  (55.23%)
            26,064      branch-misses             #    0.16% of all branches          ( +-  0.75% )  (55.30%)
         8,221,496      L1-dcache-loads           #  306.641 M/sec                    ( +-  0.07% )  (66.42%)
           790,271      L1-dcache-load-misses     #    9.59% of all L1-dcache accesses  ( +-  1.48% )  (74.63%)
            39,207      LLC-loads                 #    1.462 M/sec                    ( +-  1.81% )  (59.69%)
            42,949      LLC-load-misses           #  100.21% of all LL-cache accesses  ( +-  0.80% )  (48.44%)

          0.0271274 +- 0.0000281 seconds time elapsed  ( +-  0.10% )

a@a:~$ perf stat -d --repeat 100 ./col 10 a

Performance counter stats for './col 10 a' (100 runs):

             90.95 msec task-clock                #    1.017 CPUs utilized            ( +-  0.20% )
                 2      context-switches          #   22.459 /sec                     ( +-  6.24% )
                 0      cpu-migrations            #    0.000 /sec
             7,134      page-faults               #   80.112 K/sec                    ( +-  0.00% )
       235,965,836      cycles                    #    2.650 GHz                      ( +-  0.07% )  (47.24%)
        86,014,658      instructions              #    0.37  insn per cycle           ( +-  0.11% )  (60.41%)
        13,131,749      branches                  #  147.464 M/sec                    ( +-  0.26% )  (60.42%)
            14,802      branch-misses             #    0.12% of all branches          ( +-  1.13% )  (61.28%)
        14,045,201      L1-dcache-loads           #  157.721 M/sec                    ( +-  0.37% )  (65.69%)
         8,019,803      L1-dcache-load-misses     #   59.79% of all L1-dcache accesses  ( +-  0.14% )  (65.94%)
         3,652,627      LLC-loads                 #   41.017 M/sec                    ( +-  1.08% )  (51.92%)
         1,280,277      LLC-load-misses           #   36.37% of all LL-cache accesses  ( +-  1.53% )  (47.51%)

          0.089411 +- 0.000181 seconds time elapsed  ( +-  0.20% )

a@a:~$
```

CPU has a prefetch mechanism based on locality!

Performance optimization

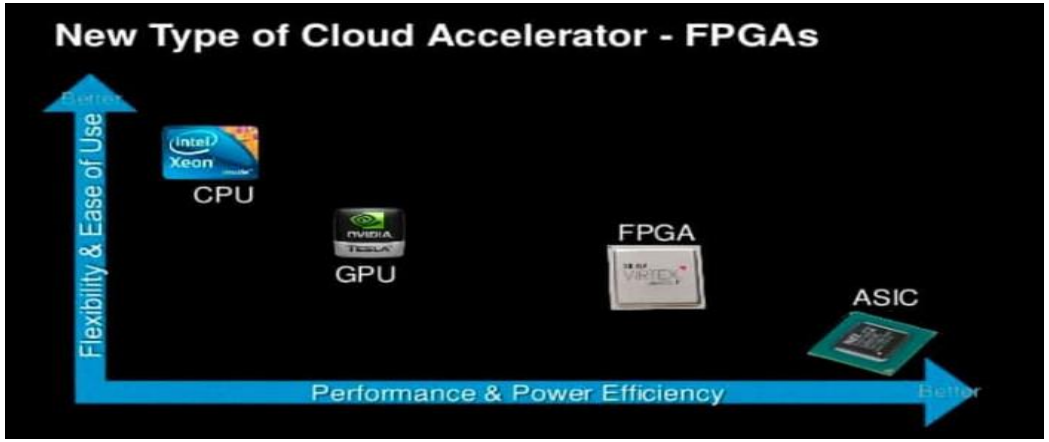
- Function call
  - Minimize the function call in the loop, or replace it with #define in C or C++ template
- Specify register
  - C++ register variables
- Data type
  - Reduce floating point operations, replace with integers

```
C01E 8D F0     INHEX   BSR    INCH      GET A CHAR
C020 81 30             CMP A  #'0       ZERO
C022 2B 11             BMI    HEXERR    NOT HEX
C024 81 39             CMP A  #'9       NINE
C026 2F 0A             BLE    HEXRTS    GOOD HEX
C028 81 41             CMP A  #'A
C02A 2B 09             BMI    HEXERR    NOT HEX
C02C 81 46             CMP A  #'F
C02E 2E 05             BGT    HEXERR
C030 80 07             SUB A  #7        FIX A-F
C032 84 0F     HEXRTS  AND A  #$0F      CONVERT ASCII TO DIGIT
C034 39                RTS

C035 7E C0 AF  HEXERR  JMP    CTRL      RETURN TO CONTROL LOOP
```

The final choice:
Switch to Assembly, if you can!

Further Reading



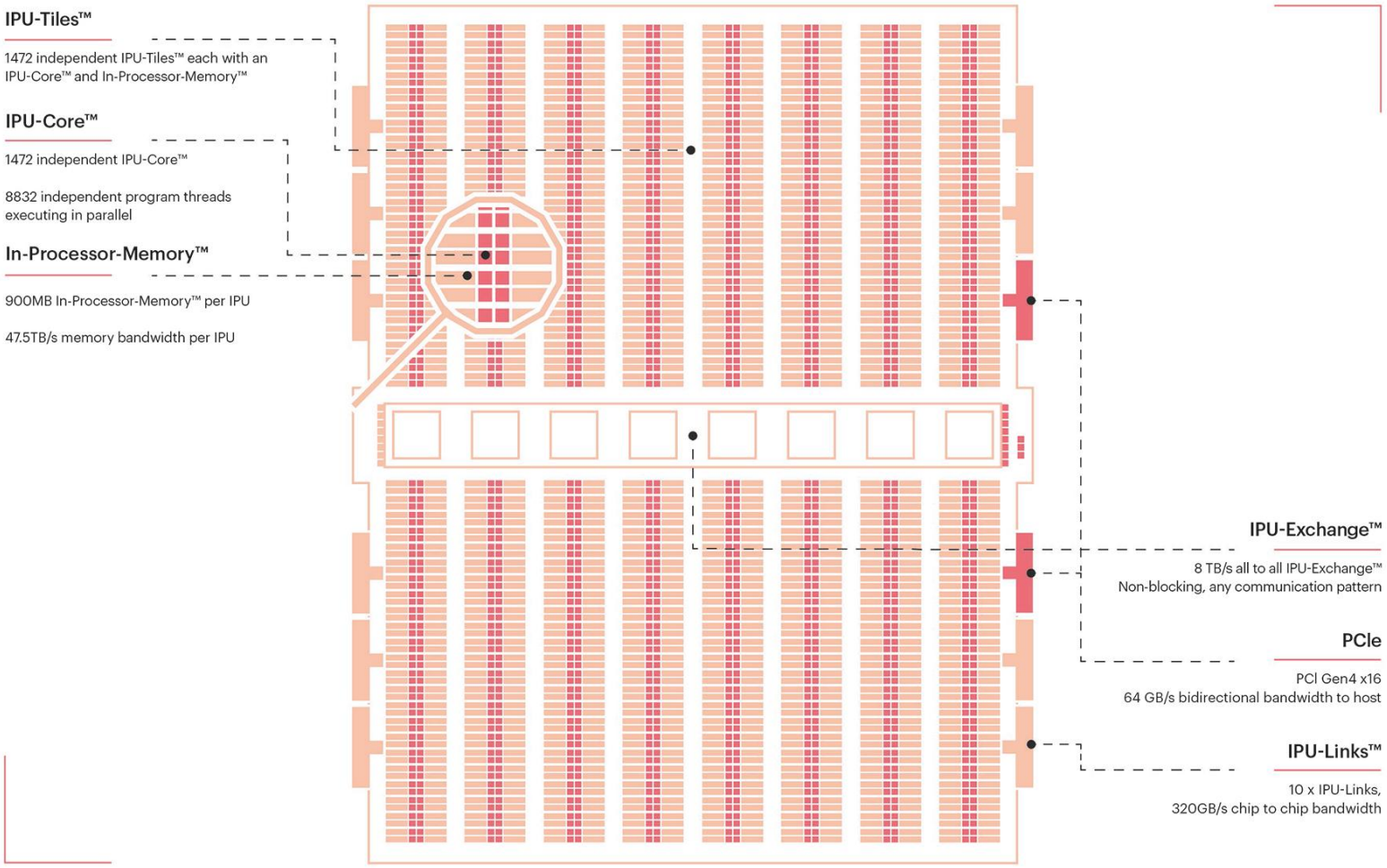
application-specific integrated circuit

- What is the fastest way to speed up an algorithm?
  - Abandoning the von Neumann architecture
  - Sacrifice Generality for Specialized Acceleration: from CPU to ASIC
  - Example: summing 10,000-length array
    - A circuit supporting 10,000 inputs + 1 output

Field-programmable gate array

Frontier

- Intelligence Processing Unit @ Graphcore



Design for Deep Learning:
Non-von Neumann architectur
125 Tflops@120W
45TB/s Memory Bandwidth