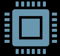
MARKUS GRUBER

High Performance Computing

Content of this Presentation

Problems in High Performance Computing (HPC)

Rules in High Performance Computing (**Moore's Law, Amdahl's Law, Gustafson's Law)**

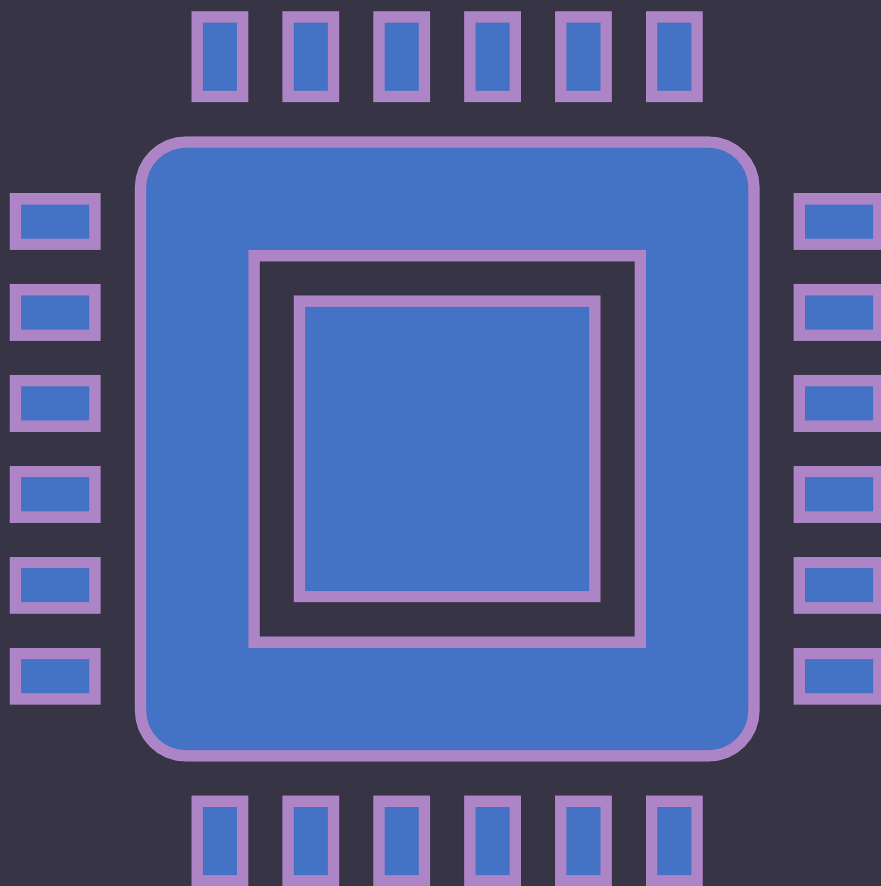

Computer Architectures (CPUs, GPUs, FPGAs)

Dependency Checks in Programms

Multiprocessor Computer Systems (Shared vs. Distributed Systems)

Programming Paradigms

Parallel Programming on CPUs, GPUs



What is high performance computing?

"High Performance Computing most generally refers to the practice of aggregating computing power in a way that delivers much higher performance than one could get out of a typical desktop computer or workstation in order to solve large problems in science, engineering, or business."

https://insidehpc.com/hpc-basic-training/what-is-hpc/

Applications of High Performance Computing

HEALTHCARE

ENGINEERING (Aircraft,

SPACE RESEARCH

URBAN PLANNING

FINANCE & BUSINESS

WEATHER RESEARCH

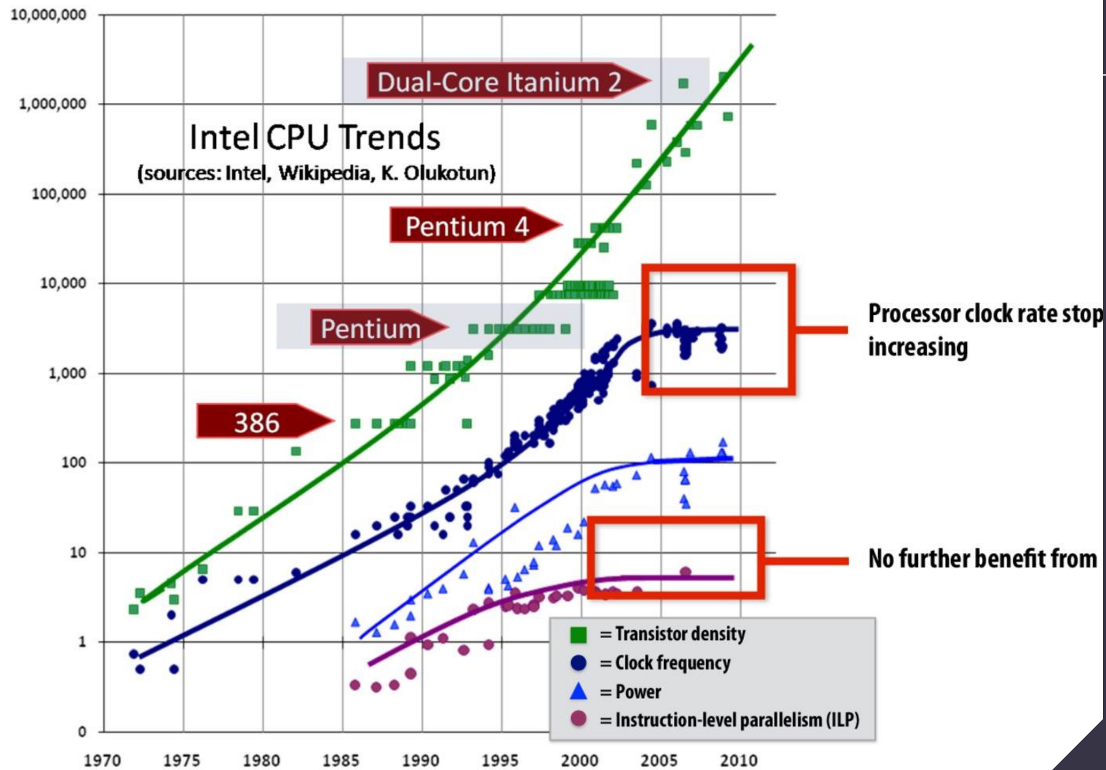
MACHINE LEARNING

https://builtin.com/hardware/high-performance-computing-applications

Problems in High Performance Computing

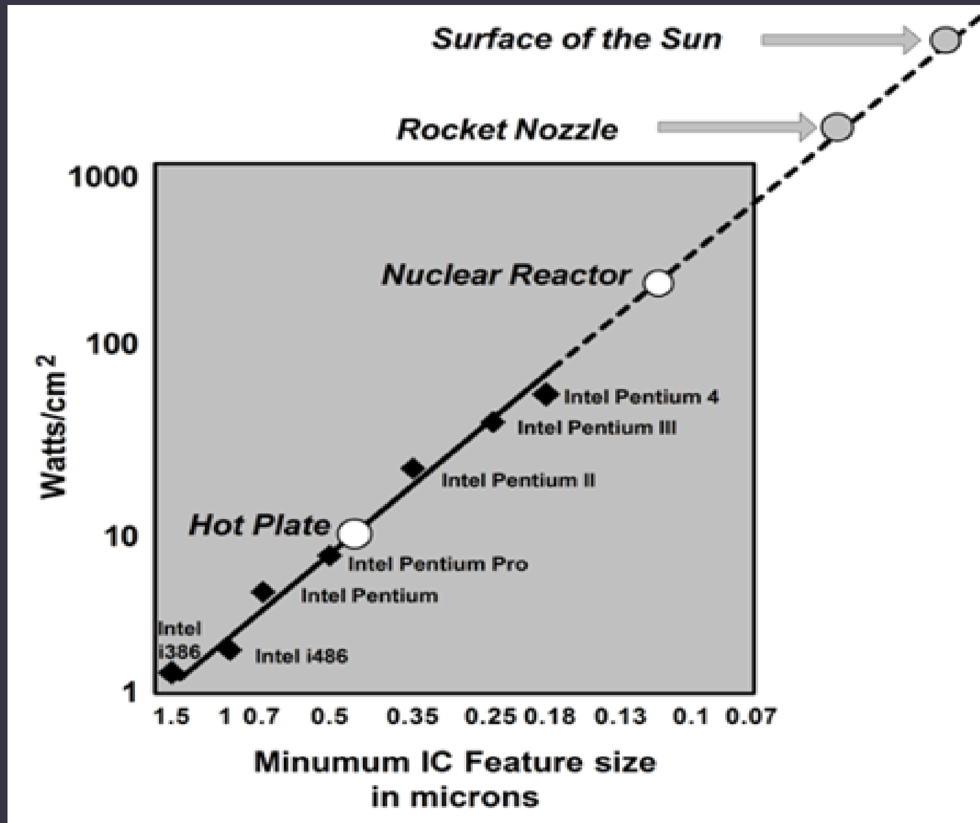
Energy Wall Problem

Bandwidth Problem



Scaling Bound in HPC

- http://15418.courses.cs.cmu.edu/spring2016/lecture/whyparallelism/slide_033

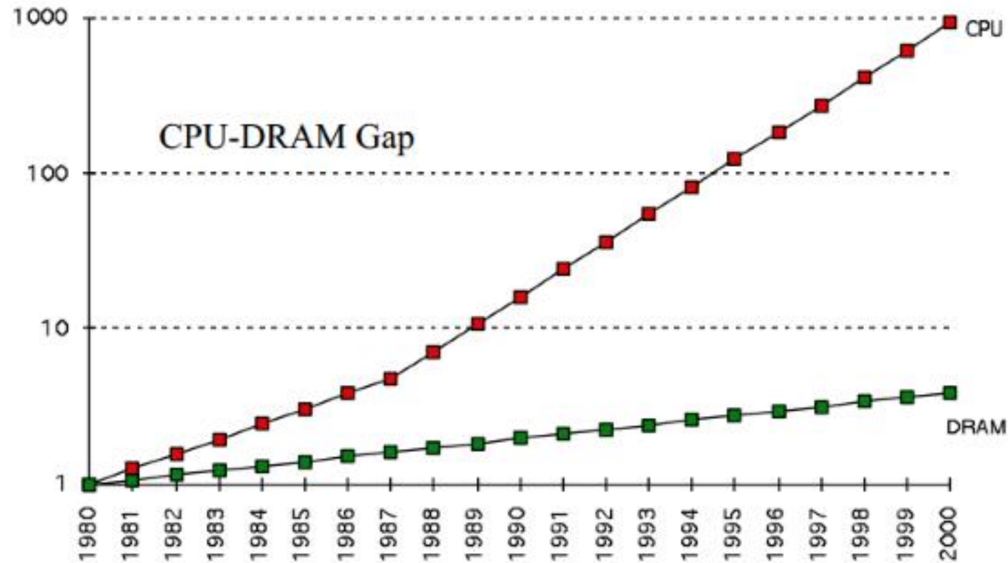


Energy Wall Problem in HPC

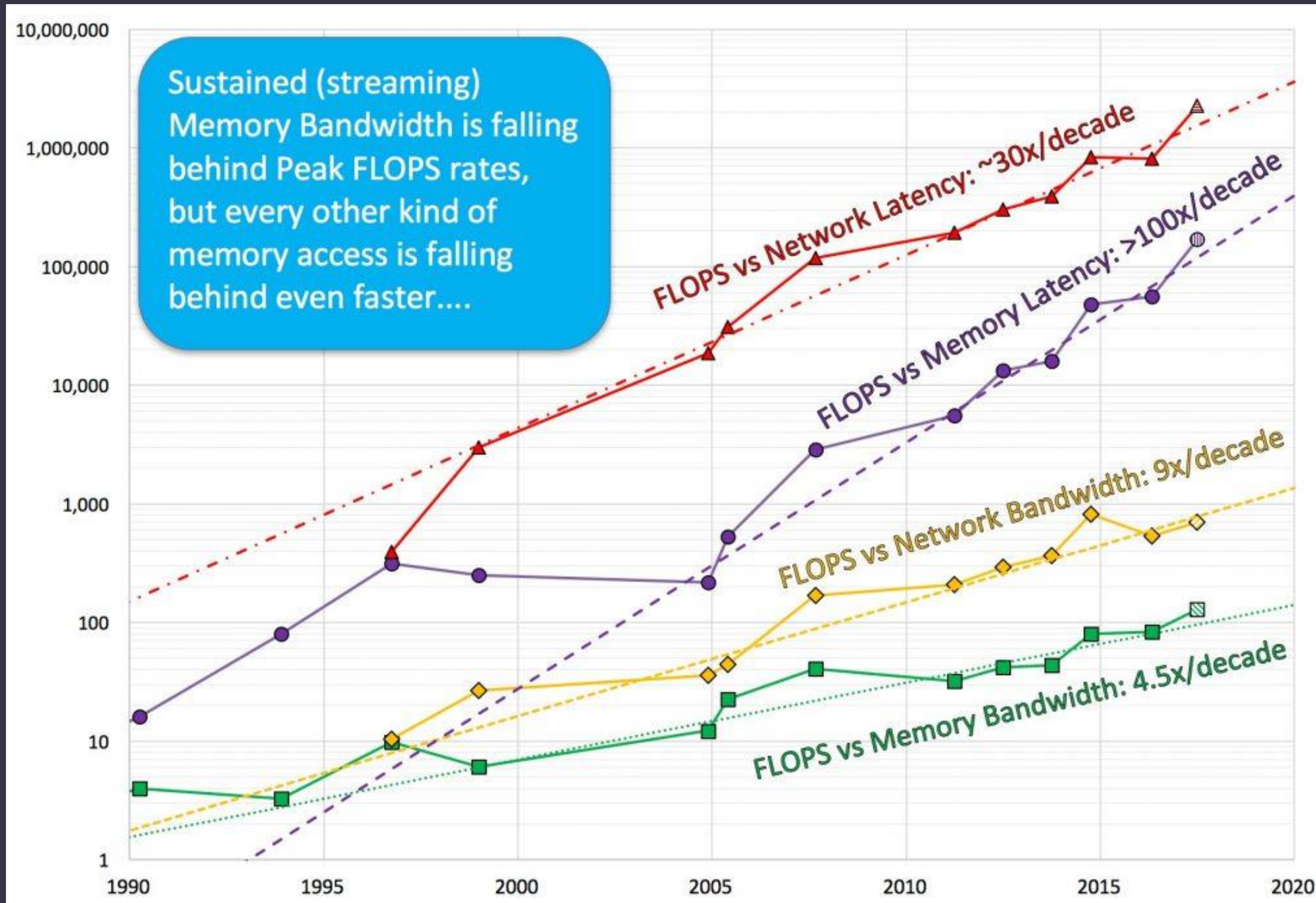
$$P = N * alpha * C * V^2 * f$$

$$Power = Number\ of\ CPUs * Active\ rate\ of\ Processors * Capacity * Voltagte^2 * Clock\ Frequency$$

Processor vs Memory Speed



https://www.extremetech.com/extreme/188776-how-l1-and-l2-cpu-caches-work-and-why-theyre-an-essential-part-of-modern-chips



Bandwidth Bound in HPC

- http://sc16.supercomputing.org/2016/10/07/sc16-invited-talk-spotlight-dr-john-d-mccalpin-presents-memory-bandwidth-system-balance-hpc-systems/index.html

Different Laws in High Performance Computing

Moore's Law: **Moore's law** is the observation that the number of transistors in a dense integrated circuit (IC) doubles about every two years. Moore's law is an observation and projection of a historical trend. Rather than a law of physics, it is an empirical relationship linked to gains from experience in production.

Amdahl's Law: In computer architecture, **Amdahl's law** (or **Amdahl's argument**[1]) is a formula which gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved

Gustafson's Law: In computer architecture, **Gustafson's law** (or **Gustafson–Barsis's law**[1]) gives the theoretical speedup in latency of the execution of a task *at fixed execution time* that can be expected of a system whose resources are improved.

Moore’s Law

Moore's law is the observation that the number of transistors in a dense integrated circuit (IC) doubles about every two years. Moore's law is an observation and projection of a historical trend. Rather than a law of physics, it is an empirical relationship linked to gains from experience in production.

Moore's law

Moore’s Law – The number of transistors on integrated circuit chips (1971-2018)

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 linked to Moore's law.

Transistor count

Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
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.

Speed Up and Efficency

The speed up is defined as the ratio of the serial runtime of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on N processors. :

Speed Up (S) whereas $T_{Serial} \ldots Serial\ Time,\ T_{Parallel} \ldots Parallel\ Time$

$$S = \frac{T_{Serial}}{T_{Parallel}}$$

The efficiency (E) is defined as the ratio of speed up to the number of processors (N). Efficiency measures the fraction of time for which aprocessor is usefully utilized.

Efficiency (E):

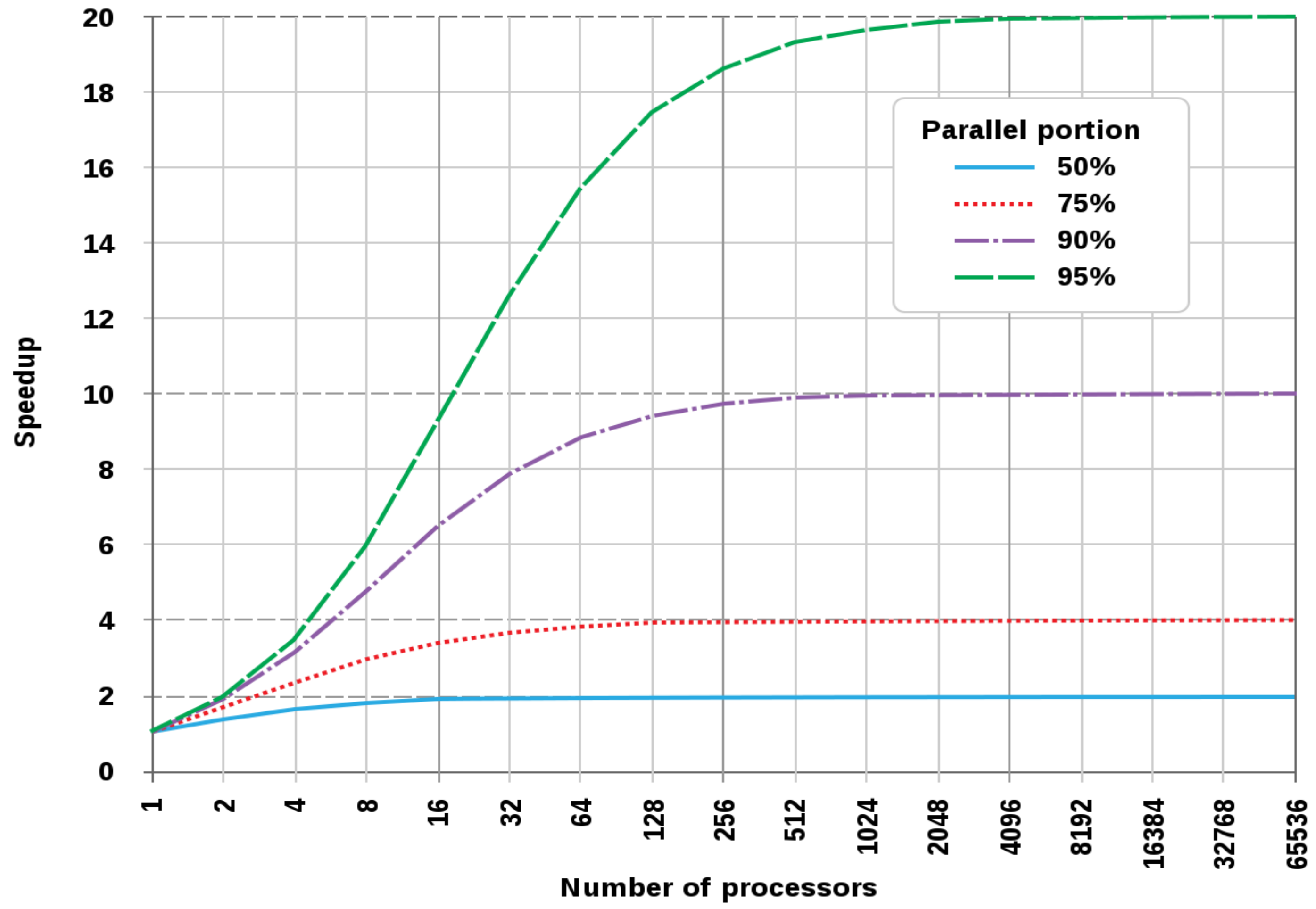
$$E = \frac{S}{N}$$

Amdahl's law

Suppose you have a sequential code and that a fraction f of its computation is parallelized and run on N processing units working in parallel, while the remaining fraction 1-f cannot be improved, i.e., it cannot be parallelized. Amdahl’s law states that the speedup achieved by parallelization is:

$$T = t_s + t_p = (1 - f) + f$$

$$S = \frac{T}{T_s + T_p/N} = \frac{T}{(1-f) + f/N} = \frac{1}{1-f}$$



Amdahl's law

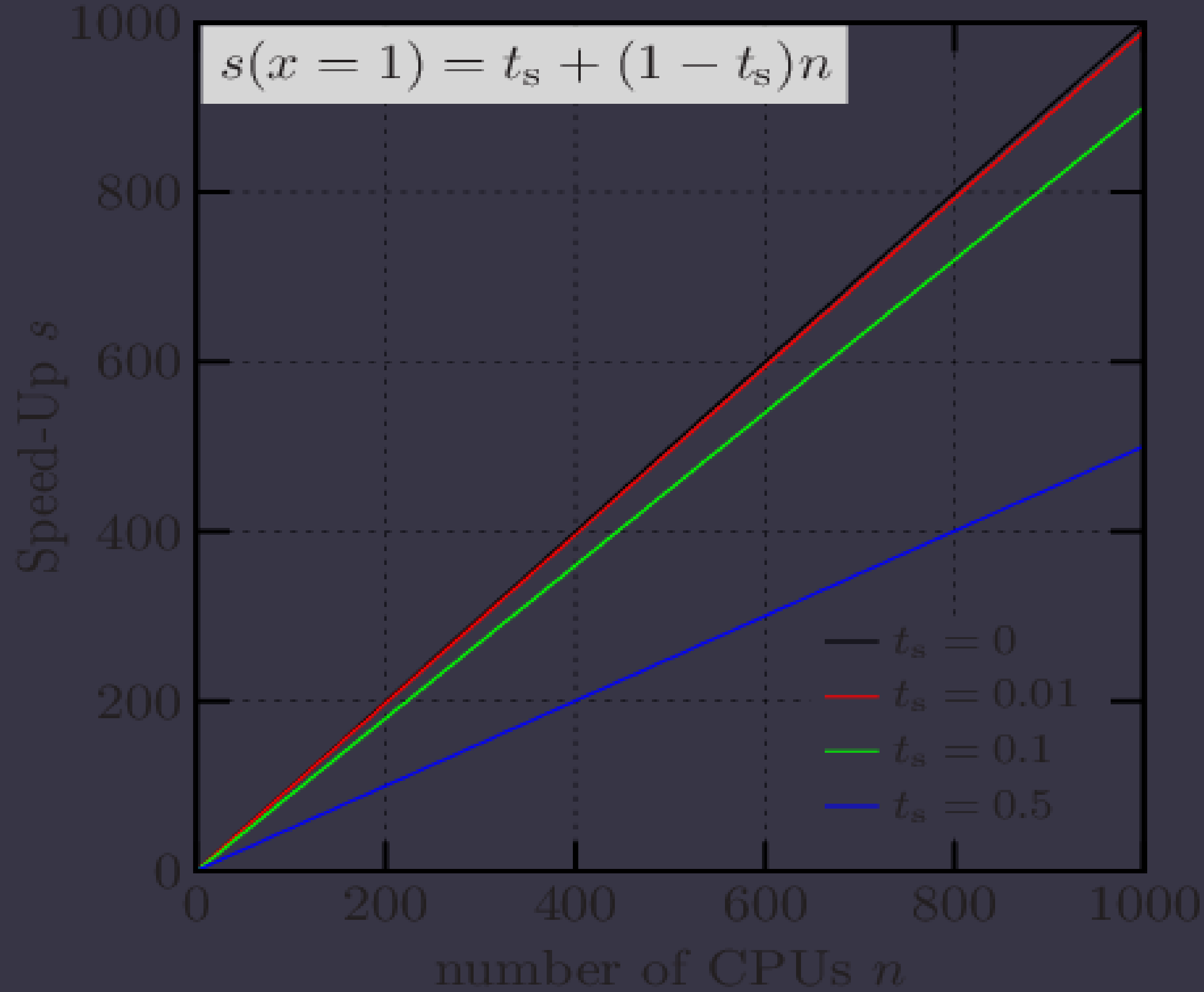
Gustafson's Law

Suppose you have an application taking a time ts to be executed on N processing units.
Of that computing time, a fraction (1-f) must be run sequentially. Accordingly,
this application would run on a fully sequential machine in a time t equal to:

$$T = (1-f)T_s + N * f * T_s$$

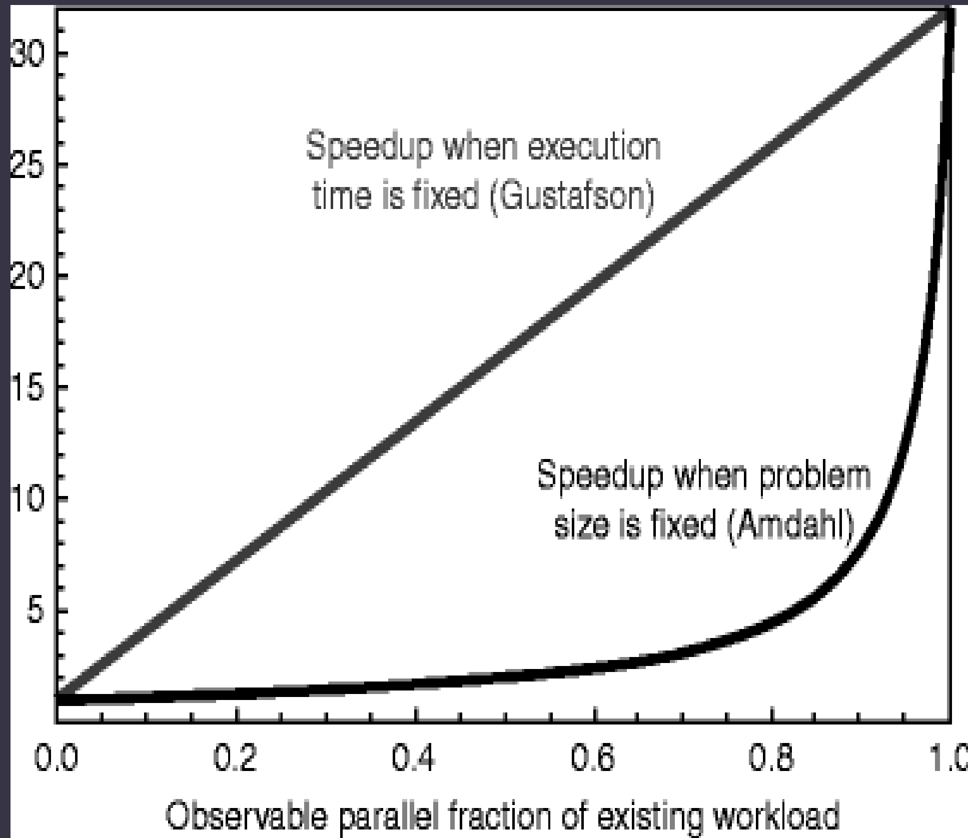
If we increase the problem size, we can increase the number of processing units to keep the fraction of time the code is executed in parallel equal to f·ts. In this case, the sequential execution time increases with N which now becomes a measure of the problem size. The speedup then becomes:

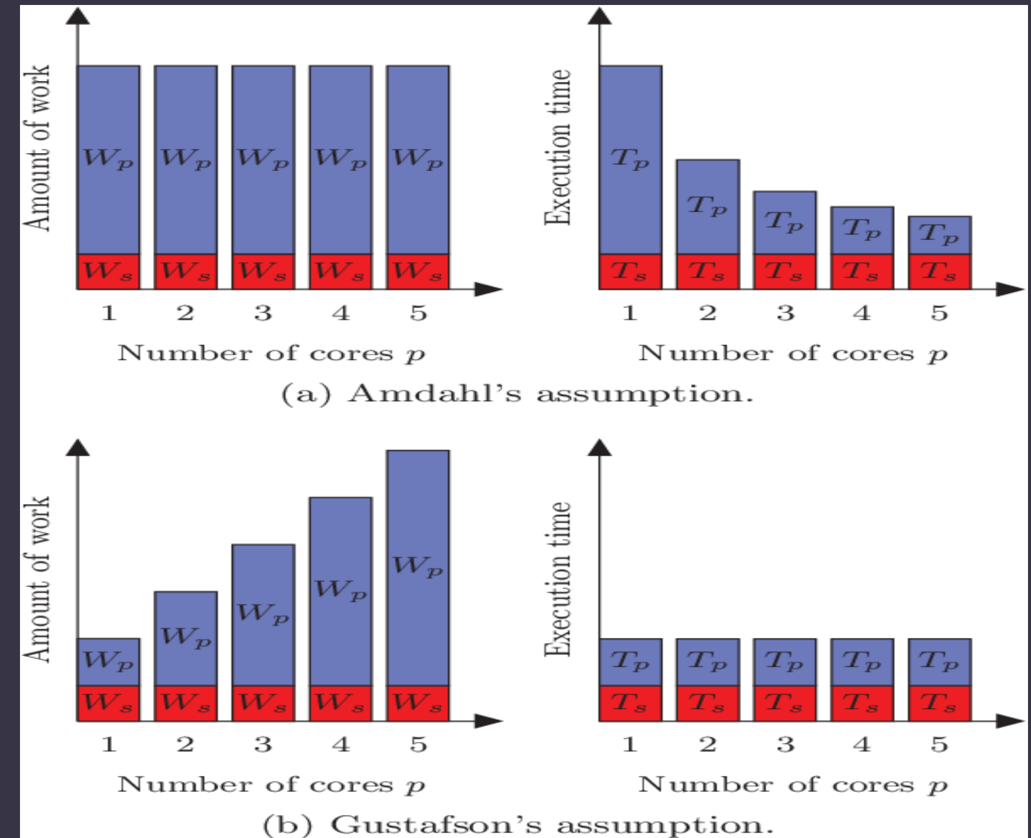
$$S = \frac{(1-f) * T_S + N * f * T_S}{T_S} = (1-f) + N * f$$



Gustafson's Law

Amdahl vs. Gustafson's Law

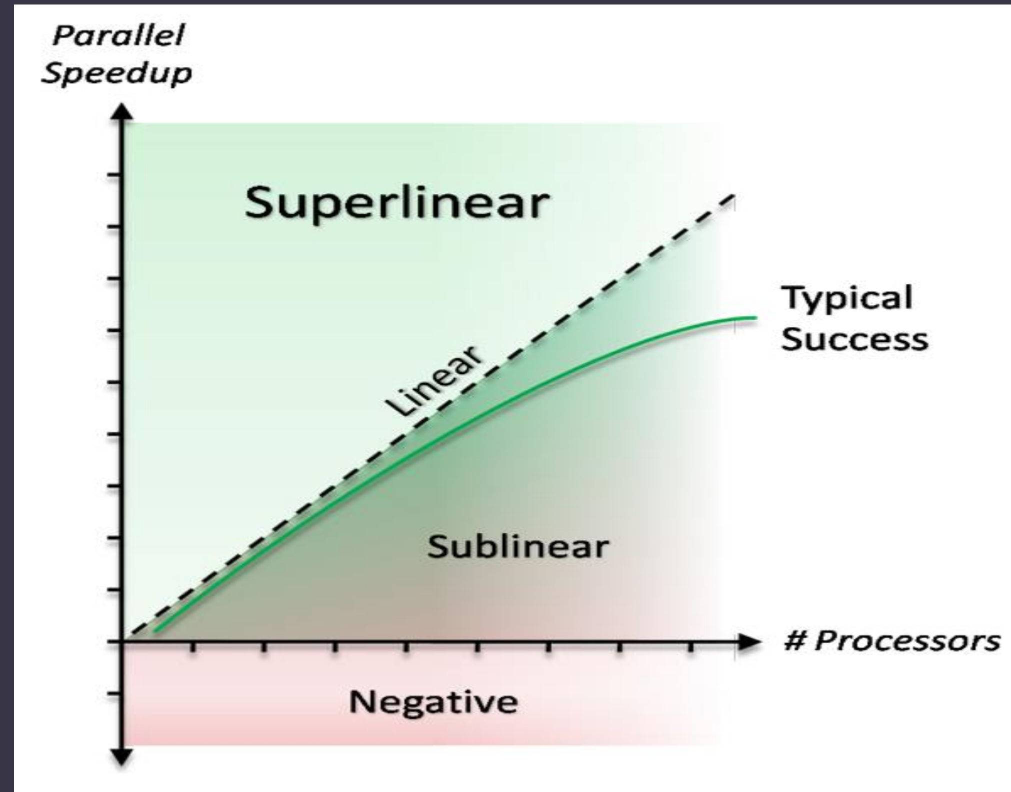




https://link.springer.com/referenceworkentry/10.1007%2F978-0-387-09766-4_78

https://www.semanticscholar.org/paper/Amdahl's-law-for-predicting-the-future-of-harmful-Juurlink-Meenderinck/d9bf0d5b321ee9bb1a0f118037a83e9026ba466e

Different Kind of Speed Ups



Some Metrics of HPC

IPC: In computer architecture, **instructions per cycle** (**IPC**), commonly called **instructions per clock** is one aspect of a processor's performance: the average number of instructions executed for each clock cycle. It is the multiplicative inverse of cycles per instruction

FLOPS: In computing, **floating point operations per second** (**FLOPS**, **flops** or **flop/s**) is a measure of computer performance, useful in fields of scientific computations that require floating-point calculations. For such cases it is a more accurate measure than measuring instructions per second.

$$FLOPS_{CORE} = \frac{FLOPS}{CYCLE} * \frac{CYLES}{SECOND}$$

$$FLOPS_{CORE} = \frac{INSTRUCTIONS}{CYCLE} * \frac{OPERATIONS}{INSTUCTION} * \frac{FLOPS}{OPERATION} * \frac{CYCLES}{SECOND}$$

$$Or\ Simple:\ FLOPS_{CORE} = AVERAGE\ \ FREQUENCY(f) * INSTRUCTIONS\ PER\ CYCLE\ (IPC)$$

Some Examples for Floating Point Operations, c=a*b, c=a+b = y=Ax+y (axpy Operation)

Simple Examples for Computing FLOPS

Node performance in GFlops = (CPU speed in GHz) x (number of CPU cores) x (CPU instruction per cycle) x (number of CPUs per node)

Example 1: Dual-CPU server based on Intel X5675 (3.06GHz 6-cores) CPUs:
3.06 x 6 x 4 x 2 = 144.88 GFLOPS

Example 2: Dual-CPU server based on Intel E5-2670 (2.6GHz 8-cores) CPUs:
2.6 x 8 x 8 x 2 = 332.8 GFLOPS
(Note that the number of instructions per cycle for E5-2600v1 and E5-2600v2 series CPUs is equal to 8)

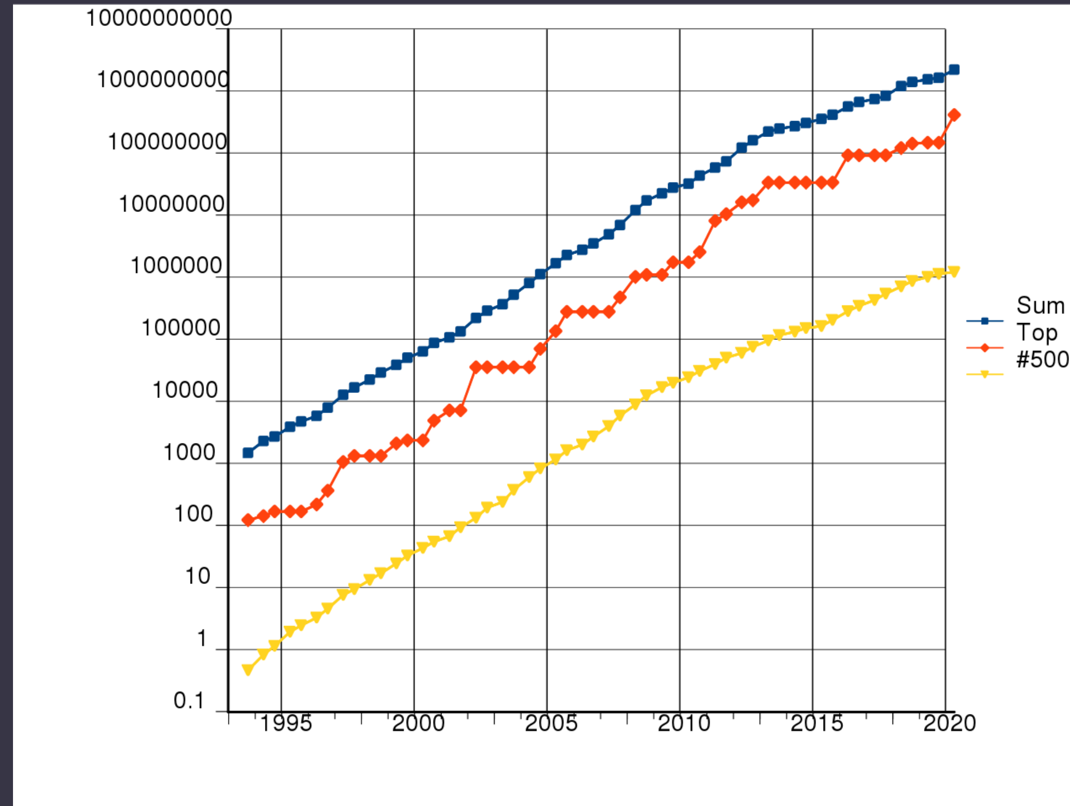
Example 3: Dual-CPU server based on Intel E5-2690v3 (2.6GHz 12-cores) CPUs:
2.6 x 12 x 16 x 2 = 998.4 GFLOPS
(Note that the number of instructions per cycle for E5-2600v3 series CPUs is equal to 16)

Example 4: Dual-CPU server based on AMD 6176 (2.3GHz 12-cores) CPUs:
2.3 x 12 x 4 x 2 = 220.8 GFLOPS

Example 5: Dual-CPU server based on AMD 6274 (2.2GHz 16-cores) CPUs:
2.2 x 16 x 4 x 2 = 281.6 GFLOPS

http://www.novatte.com/our-blog/197-how-to-calculate-peak-theoretical-performance-of-a-cpu-based-hpc-system

TOP 500



https://www.top500.org/statistics/perfdevel/

CPUs, GPUs, FPGAs and ASICS

CPU: A **central processing unit** (**CPU**), also called a **central processor**, **main processor** or just **processor**, is the electronic circuitry within a computer that executes instructions that make up a computer program. The CPU performs basic arithmetic, logic, controlling, and input/output (I/O) operations specified by the instructions in the program.

GPU: A **graphics processing unit** (**GPU**) is a specialized, electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device.

FPGA: A **field-programmable gate array** (**FPGA**) is an integrated circuit designed to be configured by a customer or a designer after manufacturing – hence the term "field-programmable".

ASIC: An **application-specific integrated circuit** (**ASIC** /ˈeɪsɪk/) is an integrated circuit (IC) chip customized for a particular use, rather than intended for general-purpose use.

Quelle: Wiki

Important Components

ALU: In computing, an **arithmetic logic unit** (**ALU**) is a combinational digital circuit that performs arithmetic and bitwise operations on integer binary numbers.[1][2][3] This is in contrast to a floating-point unit (FPU), which operates on floating point numbers.

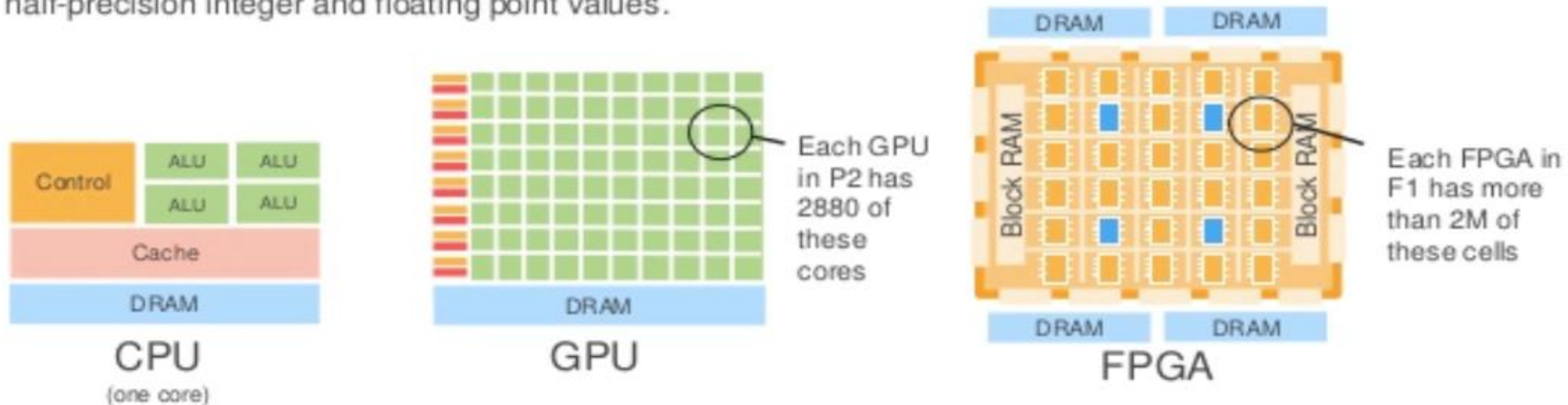
Cache: In computing, a **cache** is a hardware or software component that stores data so that future requests for that data can be served faster; the data stored in a cache might be the result of an earlier computation or a copy of data stored elsewhere.

DRAM: **Dynamic random-access memory** (**DRAM**) is a type of random access semiconductor memory that stores each bit of data in a memory cell consisting of a tiny capacitor and a transistor, both typically based on metal-oxide-semiconductor (MOS) technology.

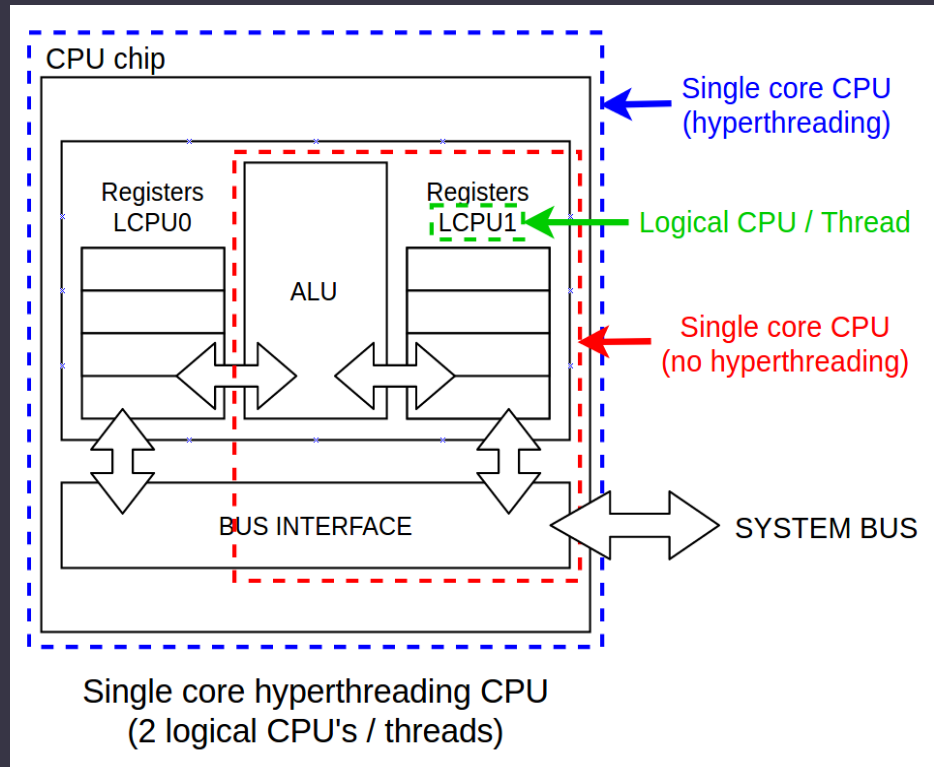
Control: The **control unit** (CU) is a component of a computer's central processing unit (CPU) that directs the operation of the processor. It tells the computer's memory, arithmetic and logic unit and input and output devices how to respond to the instructions that have been sent to the processor.

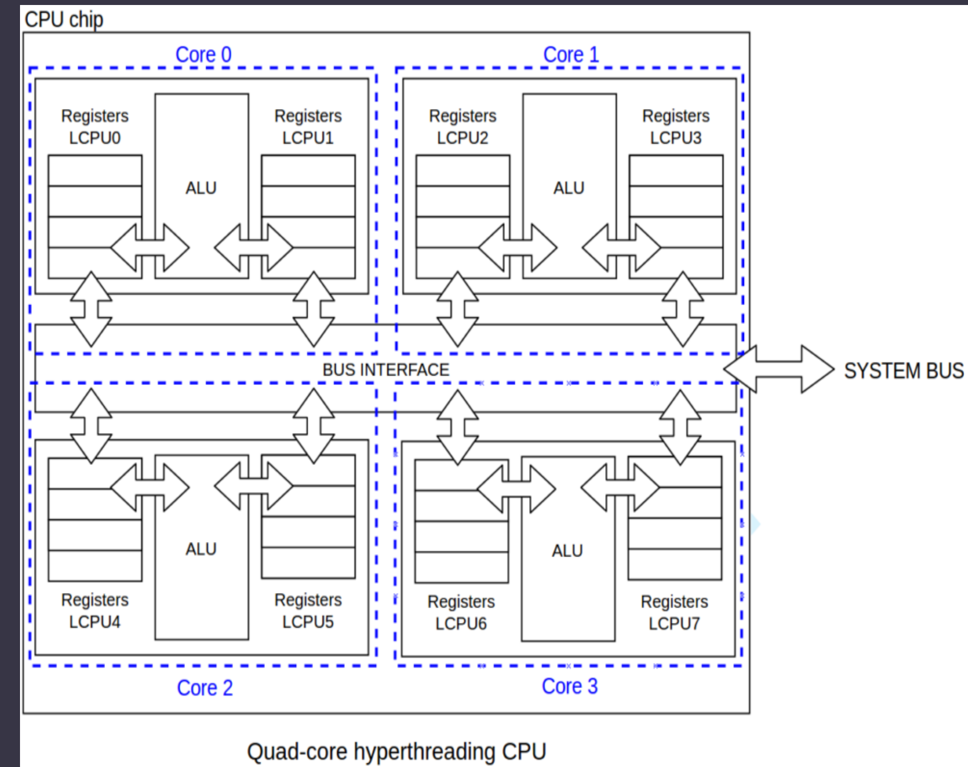
FPU: A **floating-point unit** (**FPU**, colloquially a **math coprocessor**) is a part of a computer system specially designed to carry out operations on floating-point numbers.

CPU vs. GPU vs FPGA



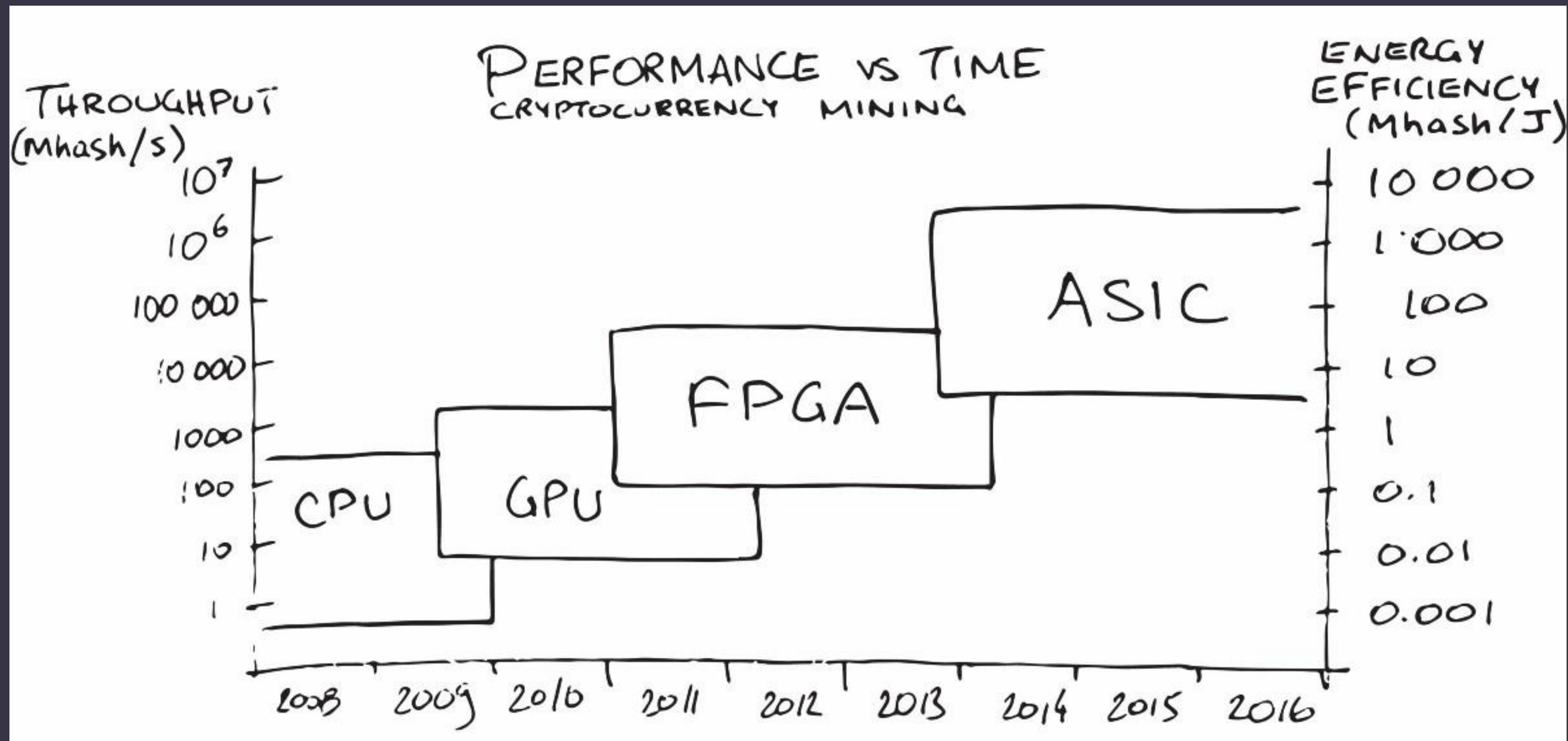
CPU





https://www.daniloaz.com/en/differences-between-physical-cpu-vs-logical-cpu-vs-core-vs-thread-vs-socket/

Performance of CPUs, GPUs, FPGAs and ASICs



https://en.bitcoinwiki.org/wiki/Why_a_GPU_mines_faster_than_a_CPU

Flynn's Taxonomy

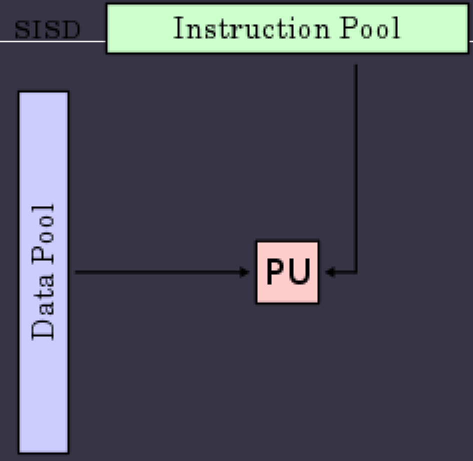
Single instruction stream, single data stream (SISD): A sequential computer which exploits no parallelism in either the instruction or data streams.

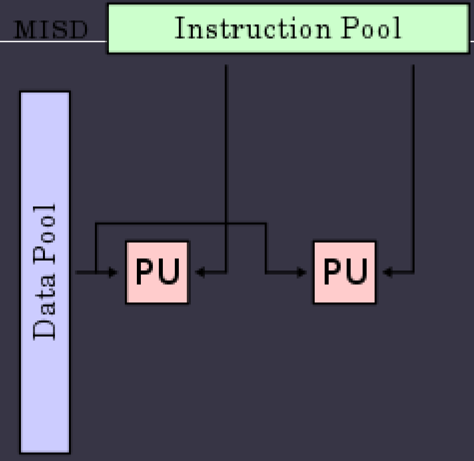
Single instruction stream, multiple data streams (SIMD): A single instruction operates on multiple different data streams. Instructions can be executed sequentially, such as by pipelining, or in parallel by multiple functional units.

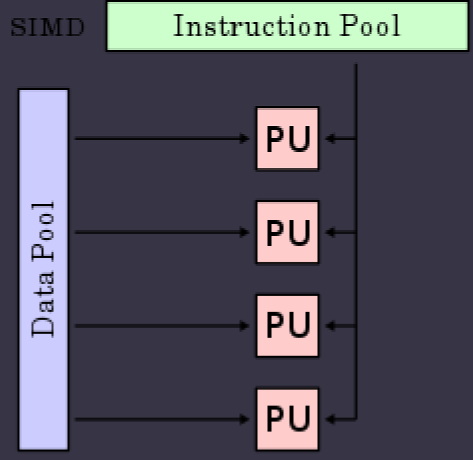
Multiple instruction streams, single data stream (MISD): Multiple instructions operate on one data stream. This is an uncommon architecture which is generally used for fault tolerance.

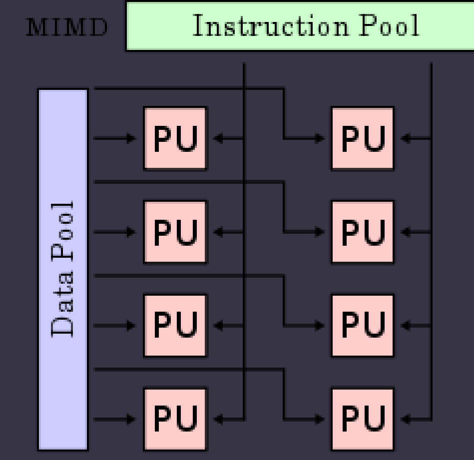
Multiple instruction streams, multiple data streams (MIMD): Multiple autonomous processors simultaneously executing different instructions on different data. MIMD architectures include multi-core superscalar processors, and distributed systems, using either one shared memory space or a distributed memory space

Flynn‘s Taxonomy









Loop Dependence Analysis

for i_1 until U_1 do ...
    for i_2 until U_2 do...
        for in until Un do...
            body
            done
        done
    done

Body:

Statement 1:a[$f_1(i_1, ..., i_n)$, ..., $f_m(i_1, ..., i_n)$] := ...

Statement 2: ... := a[$h_1(i_1, ..., i_n)$, ..., $h_m(i_1, ..., i_n)$]

Loop Dependence analysis

```
for (i = 0; i < U1; i++)
    for (j = 0; j < U2; j++)
        a[i+4−j] = b[2∗i−j] + i∗j;
    end
end
```

f1 would be i+4-j, controlling the write on the first dimension of a and h2 would be 2*i-j, controlling the read on the first dimension of b.

Flow Dependency (True Dependency)

A Flow dependency, also known as a data dependency or true dependency or read-after-write (RAW), occurs when an instruction depends on the result of a previous instruction:

Example 1:

1. A = 3

2. B = A

3. C = B

Example 2:

```
for(j = 1; j < n; j++)
    a[j] = a[j−1];
```

Anti-Dependency (1)

An anti-dependency, also known as write-after-read (WAR), occurs when an instruction requires a value that is later updated.

1. B = 3

2. A = B + 1

3. B = 7

An anti-dependency is an example of a *name dependency*. That is, renaming of variables could remove the dependency, as in the next example:

1. B = 3

N. B2 = B

2. A = B2 + 1

3. B = 7

Anti-Dependency (2)

Example 2:

```
for(j = 0; j < n; j++)
    b[j] = b[j+1];
```

Output Dependency (1)

An output dependency, also known as write-after-write (WAW), occurs when the ordering of instructions will affect the final output value of a variable.

1. B = 3

2. A = B + 1

3. B = 7

As with anti-dependencies, output dependencies are *name dependencies*. That is, they may be removed through renaming of variables, as in the below modification of the above example:

1. B2 = 3

2. A = B2 + 1

3. B = 7

Output Dependency (2)

Example 2:

```
for(j = 0; j < n; j++)
	c[j] = j;
	c[j+1] = 5;
```

Checking Dependencies:

Greatest Common Divisior:

A linear Diophantine equation:

a1*x1 + a2*x2 +... + an*xn =c

has an integer solution x1, x2,..., xn iff GCD (a1,a2,.., an) divides c.

E.g.

2*x1 -2*x2 =1

GCD(2,-2) =2, 2 cannot divide 1. So, there is no integer solution for the equation above.

GCD - Test

It is based on the observation that if a loop carried dependency exists between X[a*i + b] and X[c*i + d] (where X is the array; a, b, c and d are integers, and i is the loop variable), then GCD (c, a) must divide (d – b). For example, in the following loop, a=2, b=3, c=2, d=0 and GCD(a,c)=2 and (d-b) is -3. Since 2 does not divide -3, no dependence is possible.

```
for (i=1; i<=100; i++)
{
  X[2*i+3] = X[2*i] + 50;
}
```

GCD - Test

Loop code in general:

```
for (int i=0; i<n; i++)
{
  s1   a[x*i+k] = ...;
  s2   ... = a[y*i+m];
}
```

To decide if there is loop carried dependence (two array references access the same memory location and one of them is a write operation) between a[x*i+k] and a[y*i+m].

x*i1 +k = y*i2+m (Or x*i1 -y*i2 = m -k)

If GCD(x,y) divides (m-k), then there may exist some dependency in the loop statement s1 and s2. If GCD(x,y) does not divide (m-k) then both statements are independent and can be executed at parallel. Similarly this test is conducted for all statements present in a given loop.

Simple Example

```
for i = 1 to n1:
  a[2*i] = b[i] + c[i];
  d[i] = a[2*i-1];
```

Are there i1 and i2 such that 1 <= i1 <= i2 <= 10 an

2*i1 = 2*i2-1

or, equivalently

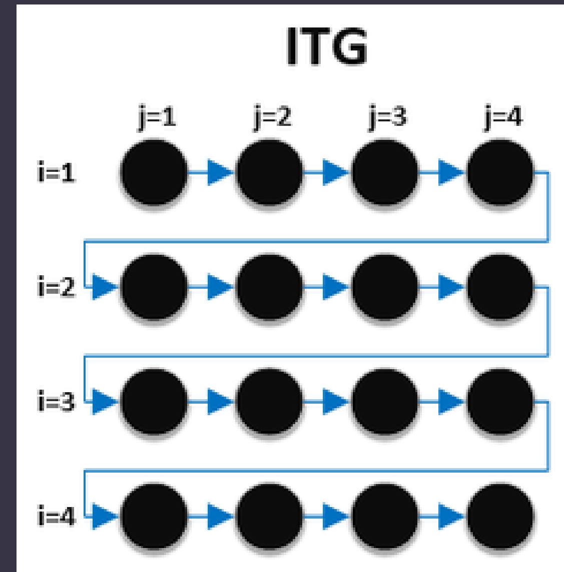
2*i2 - 2*i1 = 1

There is an integer solution if and only if gcd(2,-2) divides 1. This is not the case, so no dependence.

Checking via Graphs

Example:

```
for (i = 0; i < 4; i++)
    for (j = 0; j < 4; j++)
        a[i][j] = a[i][j−1] ∗ x;
    end
end
```



Iteration-space Traversal Graph (ITG)

Checking Dependencies:

OMEGA Test:

http://www.cs.cmu.edu/~emc/spring06/home1_files/p4-pugh.pdf

http://www.cs.umd.edu/~pugh/papers/omega.pdf

Mainly based on Integer Programming.

Further Tests:

ftp://ftp.keldysh.ru/K_student/AUTO_PARALLELIZATION/DATA_DEPENDENCE/Pact99.pdf

Optimization Methods

Loop Unrolling (Compiler, Programmer) = Instruction Level Parallism (ILP)

Vectorization (Compiler, Programmer)

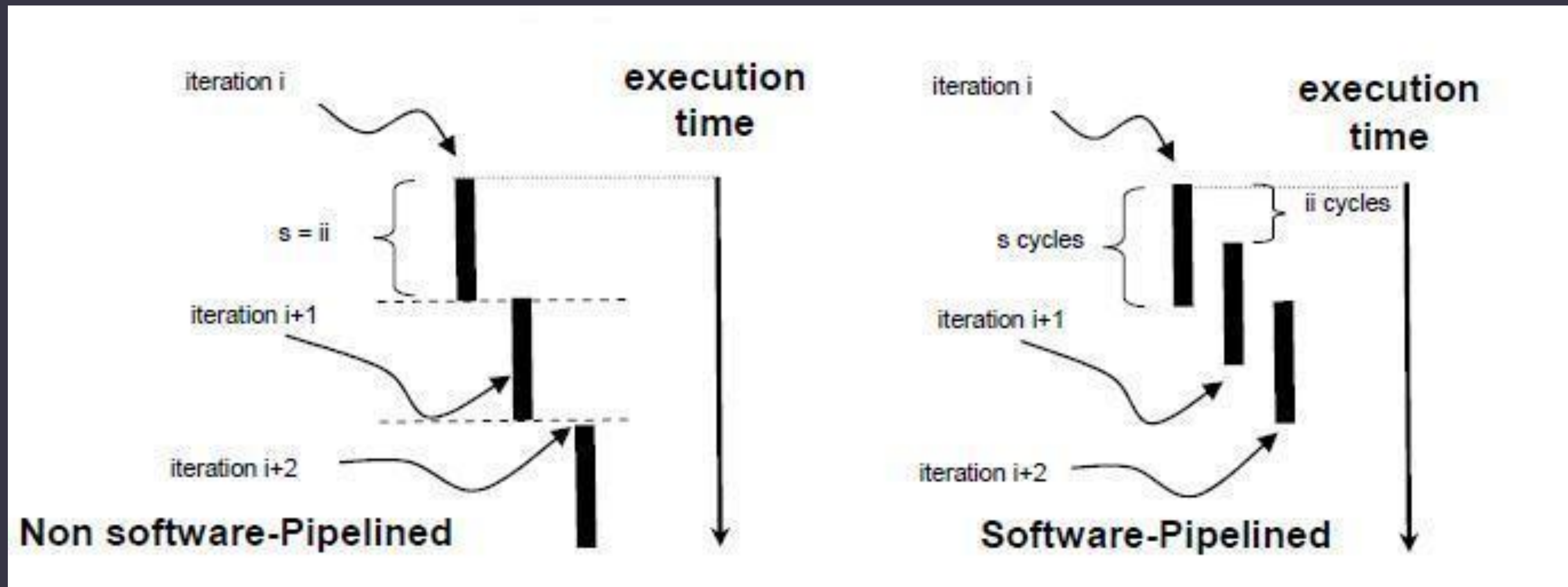
Threading (Mostly Programmer)

Memory Layout (Mostly programmer)

Cache-Oblivious Programming (Mostly programmer)

Branch Prediction (Compiler but can be improved through the programmer)

Loop Unrolling



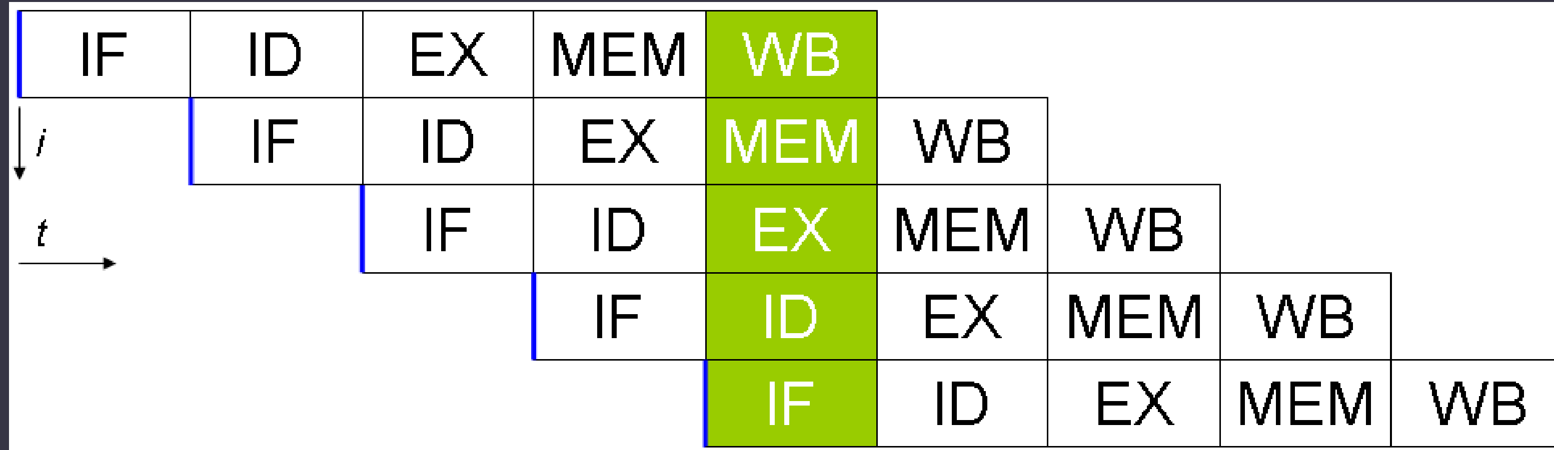
Loop Unrolling

Normal Loop

```
int x;
 for (x = 0; x < 100; x++)
 {
     delete(x);
 }
```

After Loop Unrolling:

```
int x;

 for (x = 0; x < 100; x += 5 )

 {

     delete(x);

     delete(x + 1);

     delete(x + 2);

     delete(x + 3);

     delete(x + 4);

 }
```

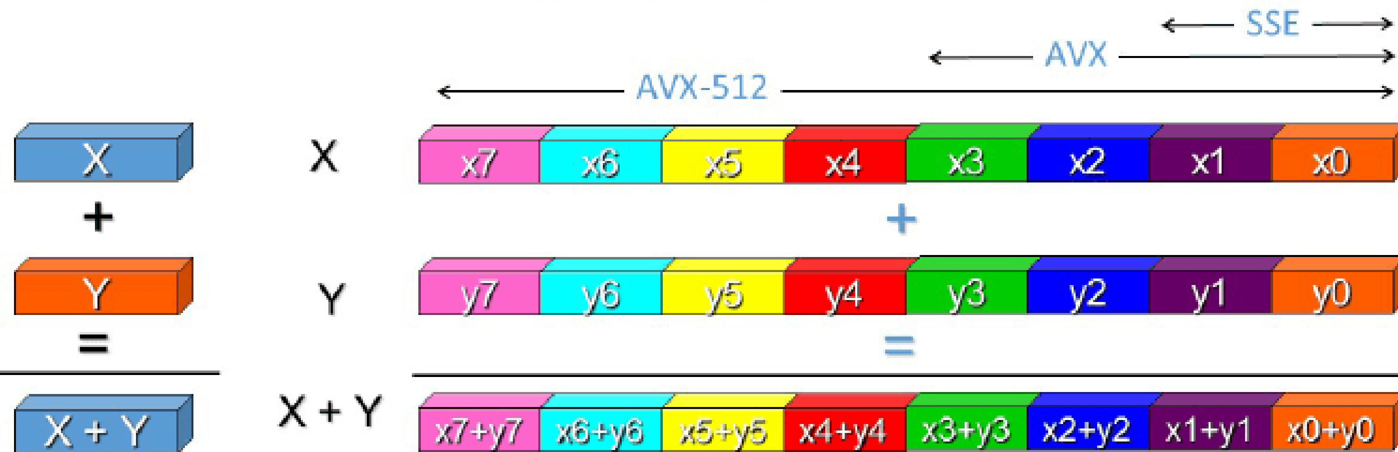


IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back

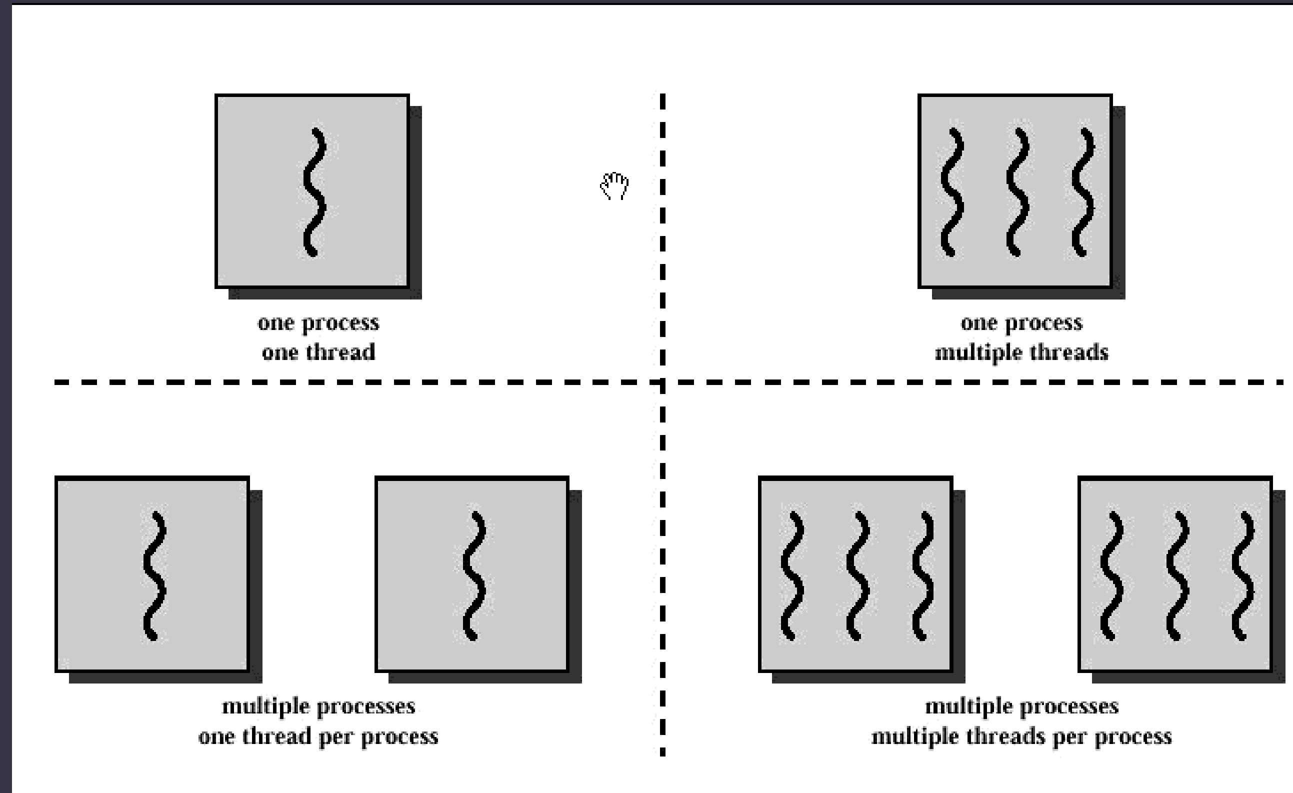
Pseudo Code:

```
int pipelines = 5;
for(int i = 0; i < length; i += pipelines){
    s += (x + y);
    s += (x + y);
    s += (x + y);
    s += (x + y);
    s += (x + y);
}
```

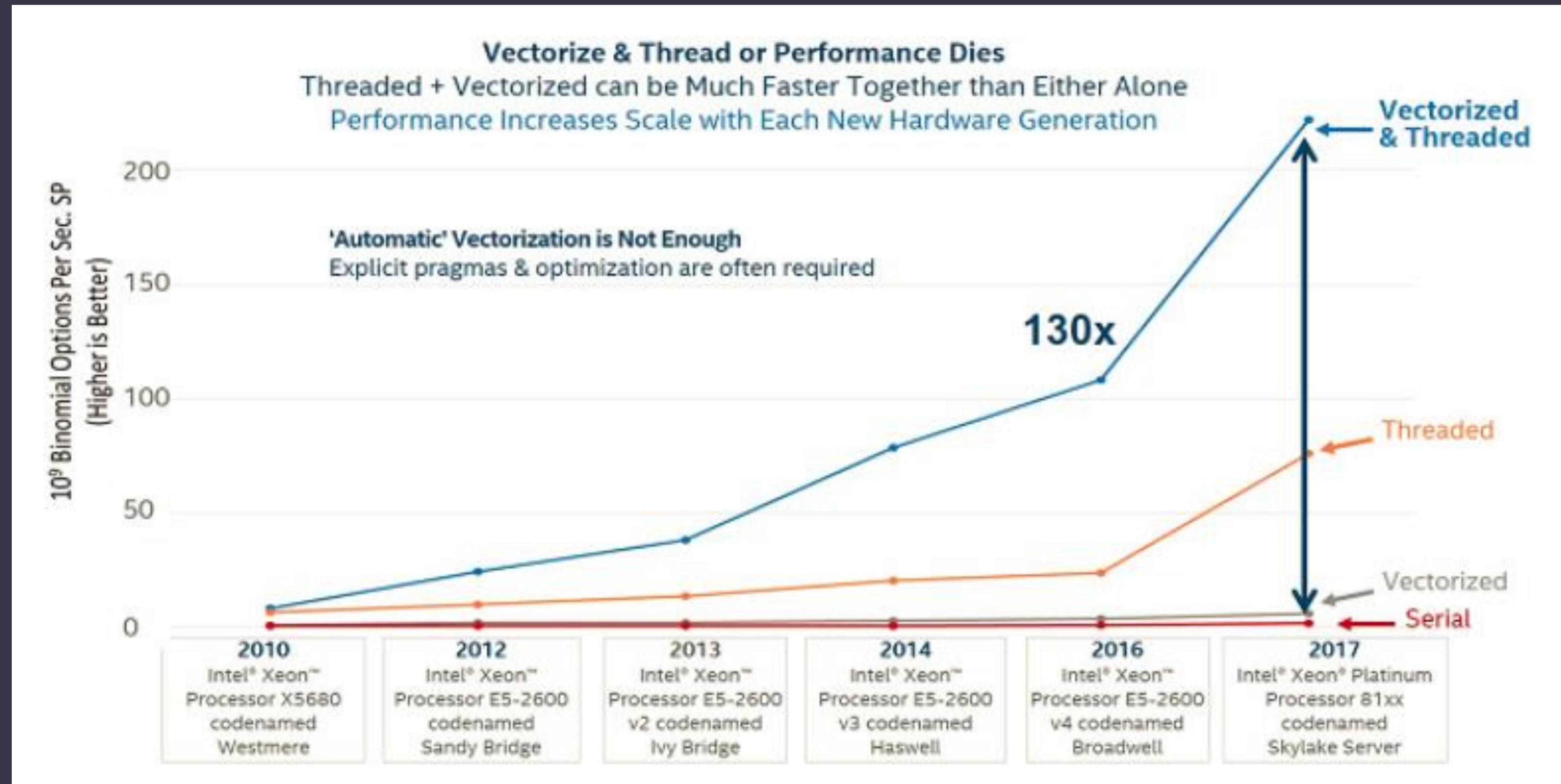
Vectorization



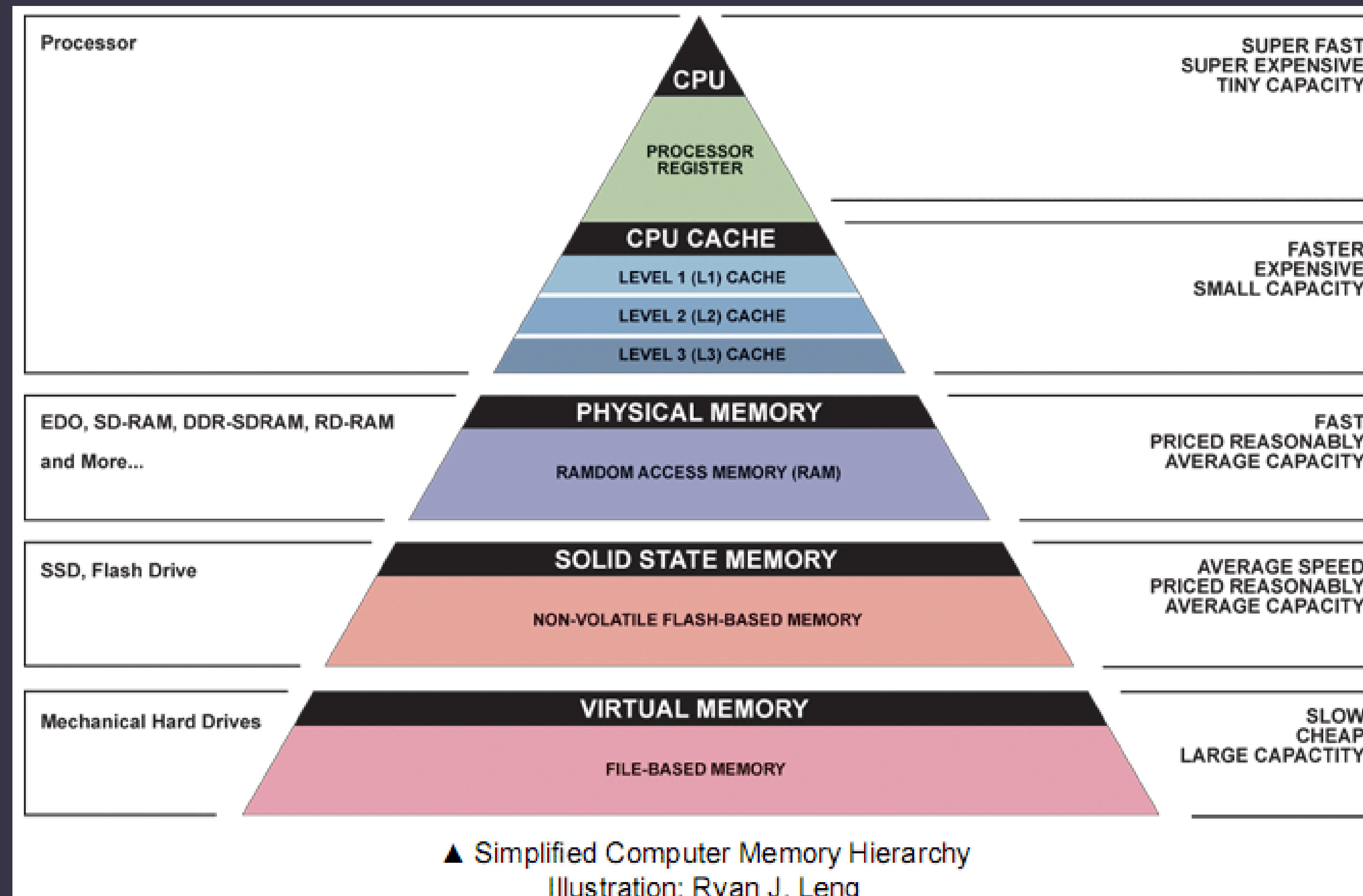
Threading



Threading + Vectorization



Cache-oblivious Programming



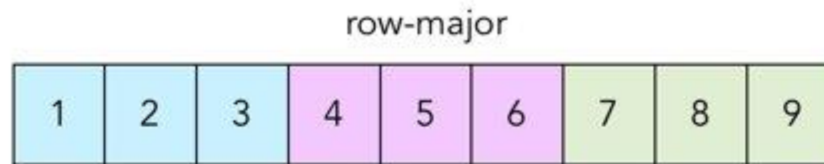
▲ Simplified Computer Memory Hierarchy
Illustration: Ryan J. Leng

https://sites.google.com/site/cachememory2011/memory-hierarchy

Memory Access

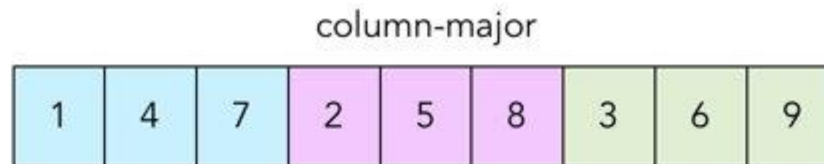
Row Access:

1	2	3
4	5	6
7	8	9



Column Access:

1	2	3
4	5	6
7	8	9



https://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays

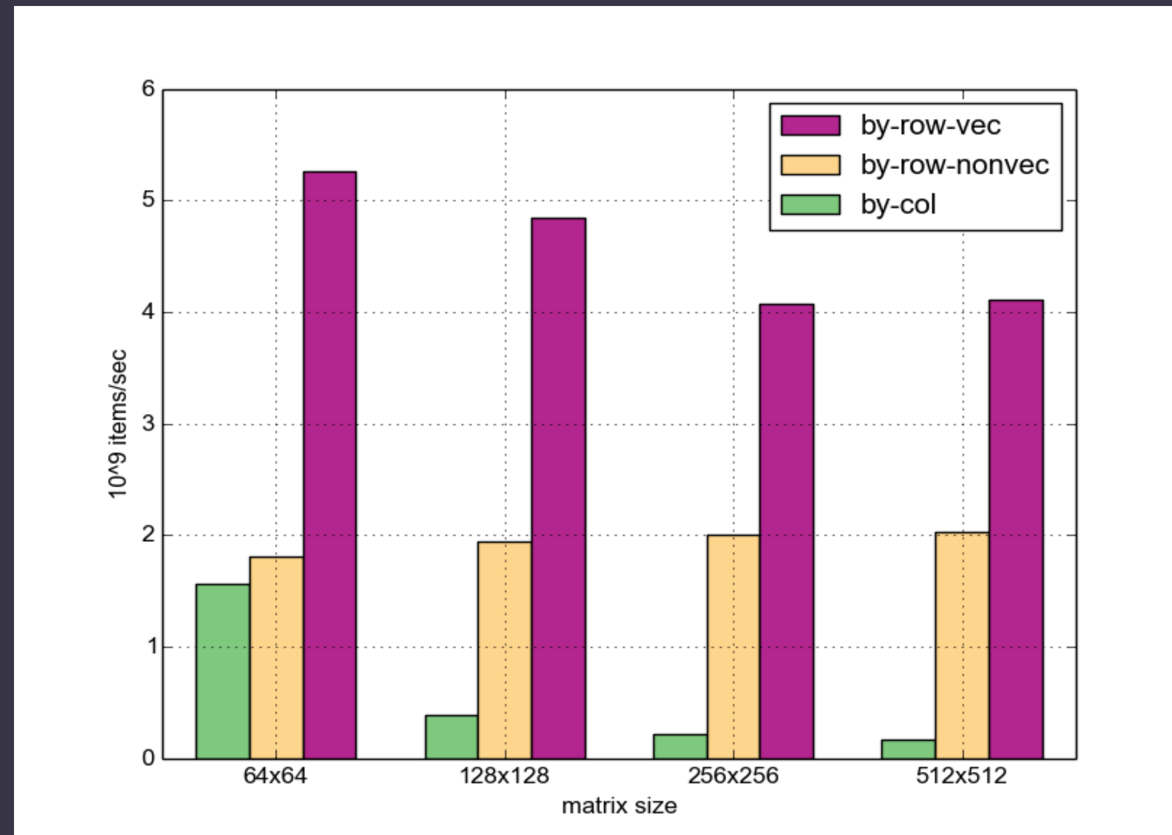
Simple Example (Column and Row Access):

```
void AddMatrixByCol(Matrix& y, const Matrix& x) {
  assert(y.M == x.M);
  assert(y.N == x.N);
  for (size_t col = 0; col < y.N; ++col) {
    for (size_t row = 0; row < y.M; ++row) {
     y.data[row * y.N + col] += x.data[row * x.N + col];
      }
    }
}
```

```
void AddMatrixByRow(Matrix& y, const Matrix& x) {
assert(y.M == x.M);
assert(y.N == x.N);
for (size_t row = 0; row < y.M; ++row) {
  for (size_t col = 0; col < y.N; ++col) {
    y.data[row * y.N + col] += x.data[row * x.N + col];
  }
 }
}
```

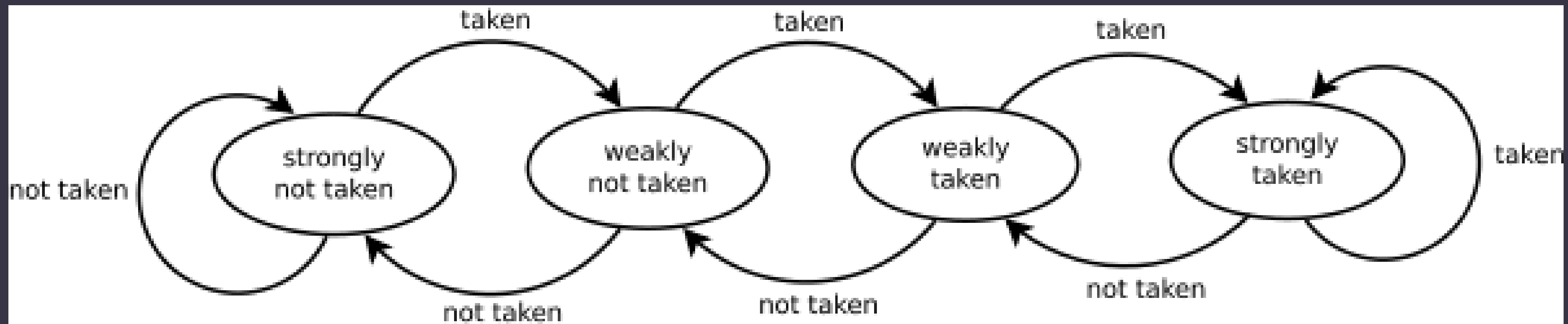
https://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays

Memory Access



https://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays

Branch Prediction



Parallel Programming on CPUs and GPUs

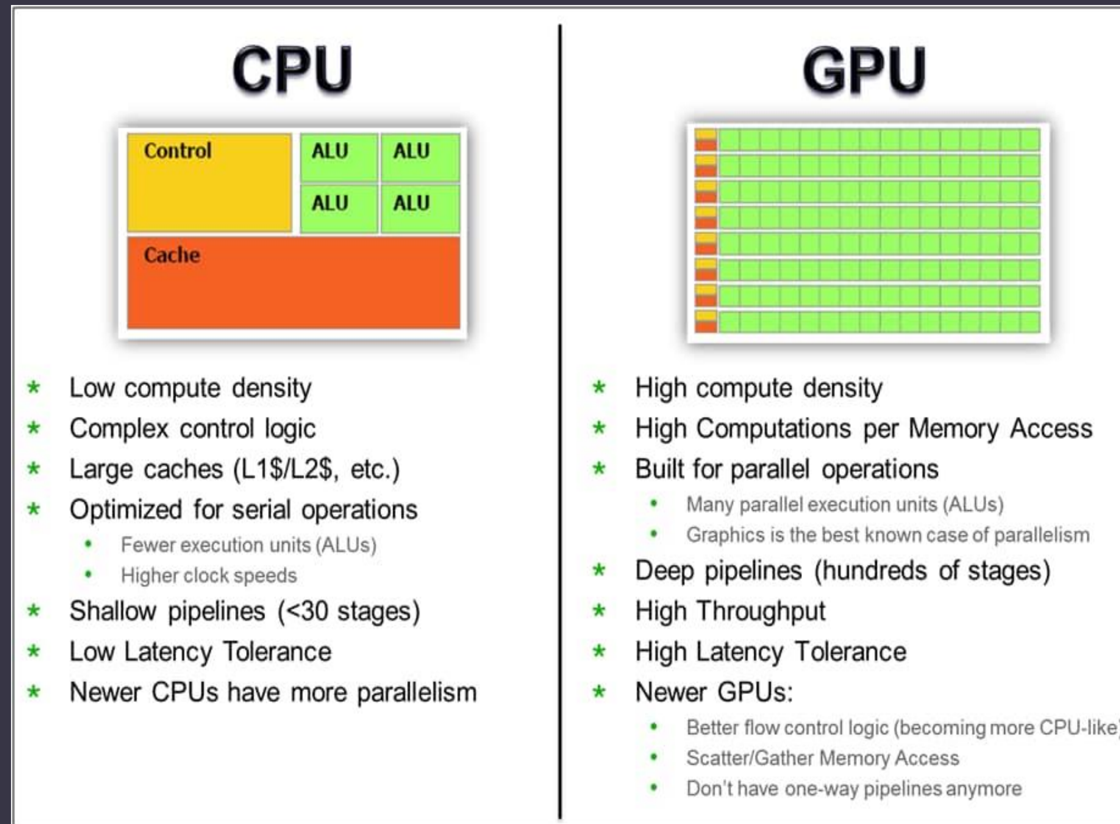
Programming on CPUs:

- Shared vs Distributed Memory Systems
- MPI (Multiprocessor Interface) vs OpenMP, TBB (Thread Building Blocks), Cilk

Programming on GPUs:

- OpenCL vs CUDA:
- Generic Programming (AMD, ..) vs Proprietary Programming (Nvidia)
- OpenCL: Programming on CPUs, GPUs, FPGAs
- CUDA (Nvidia): Programming on GPUs

CPU vs. GPU in Detail



https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Different Kinds of Parallelism

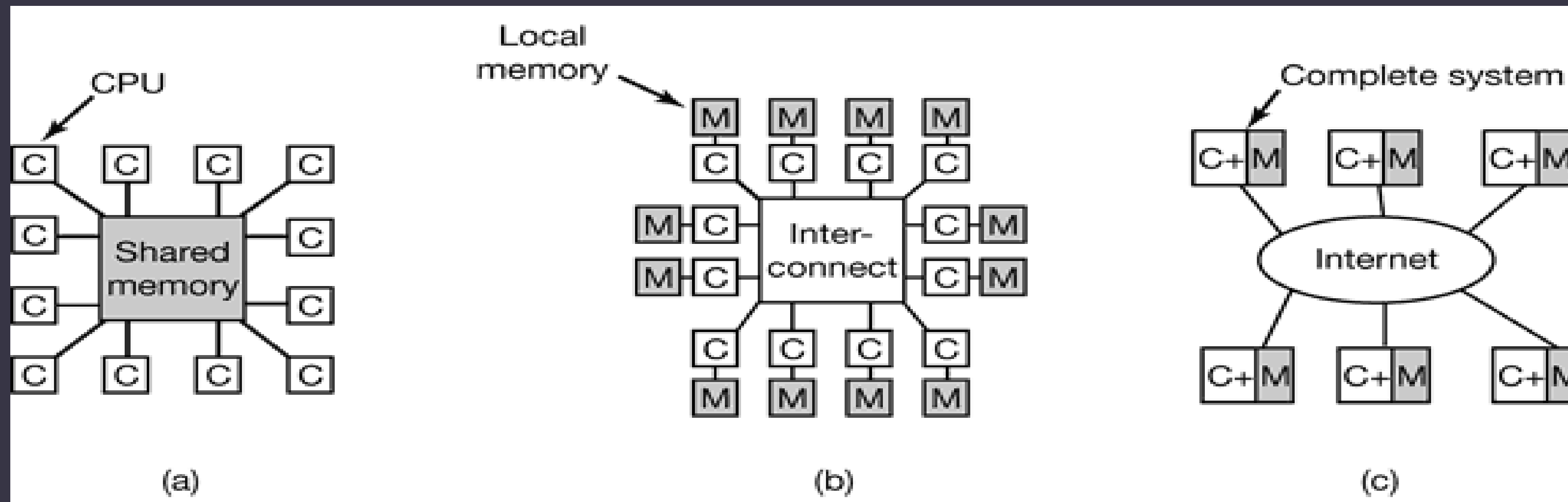
Instruction Level Parallelism (ILP)

Vectorization

Multiprocesser System:

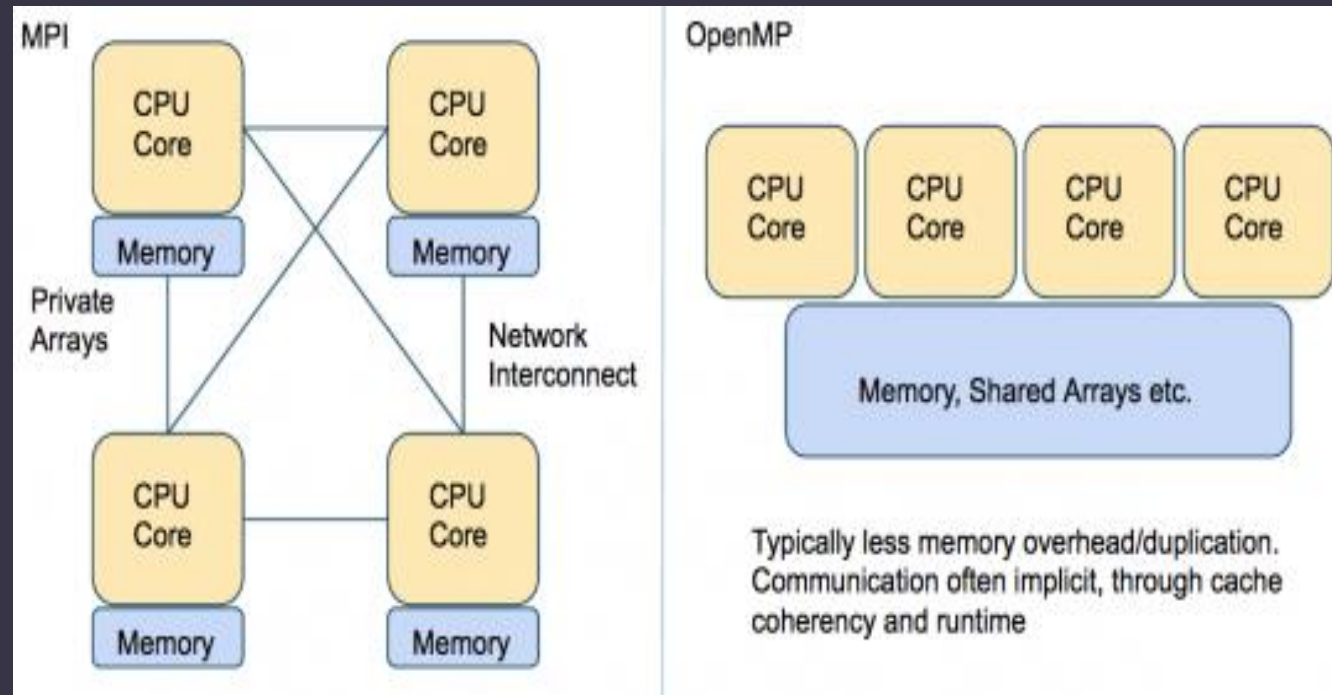
- Shared Memory (Uniform Memory Access)
- Distributed Memory (None Uniform Memory Access)

MULTIPLE PROCESSOR SYSTEMS

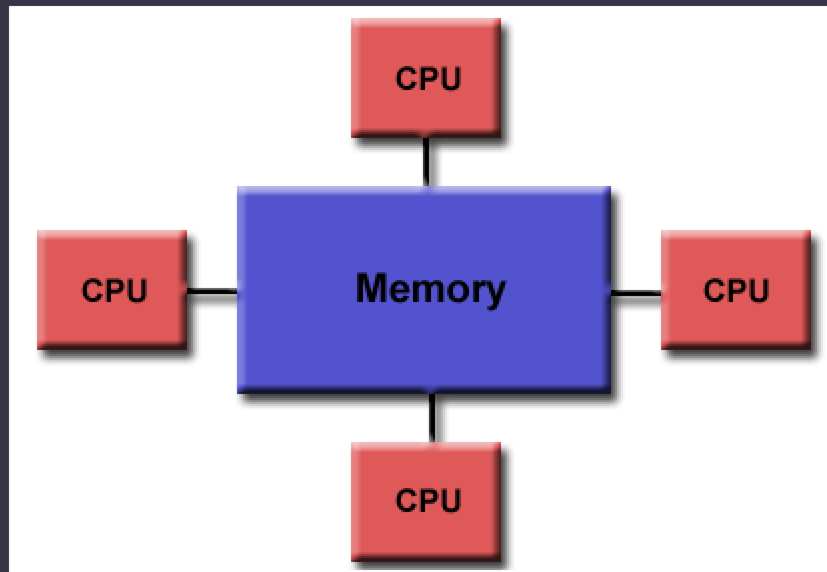


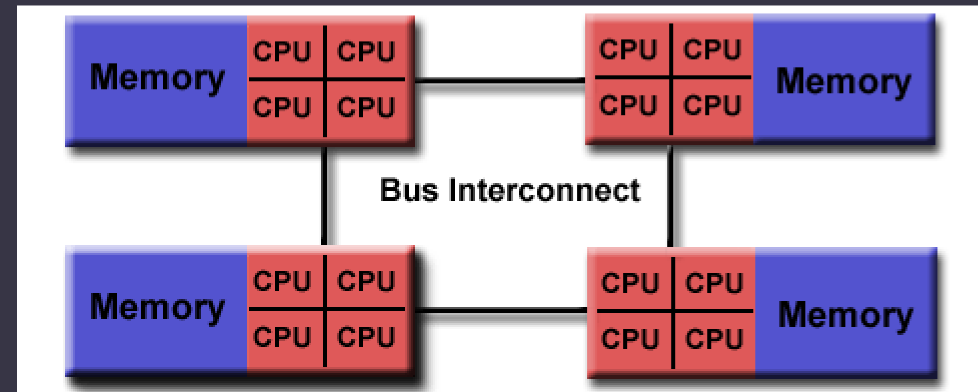
(a) A shared-memory multiprocessor. (b) A message-passing multicomputer. (c) A -wide area distributed system.

Shared Memory (OpenMP) vs Distributed Memory System (MPI)

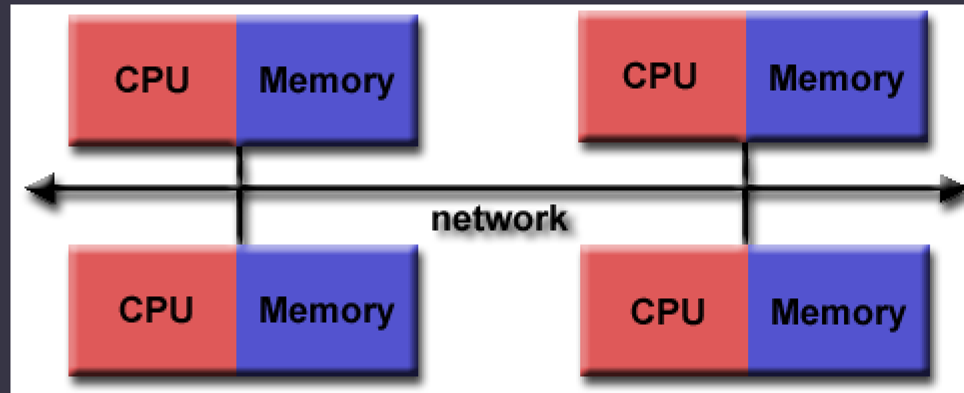


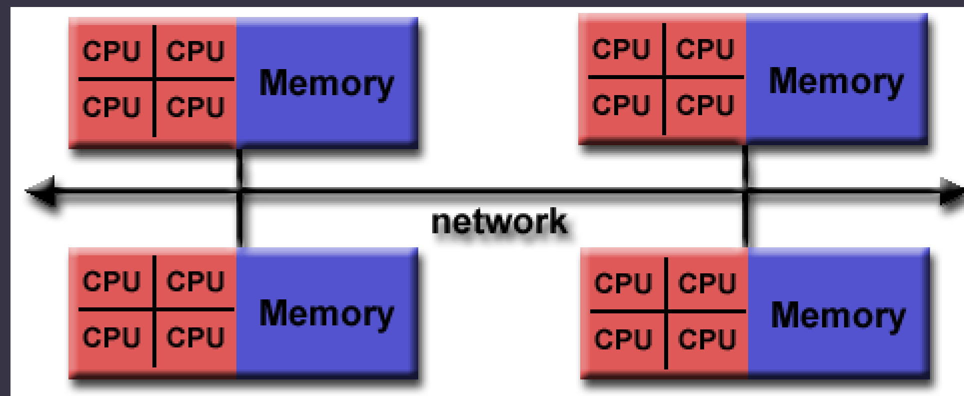
Uniform vs. None Uniform Memory Acces



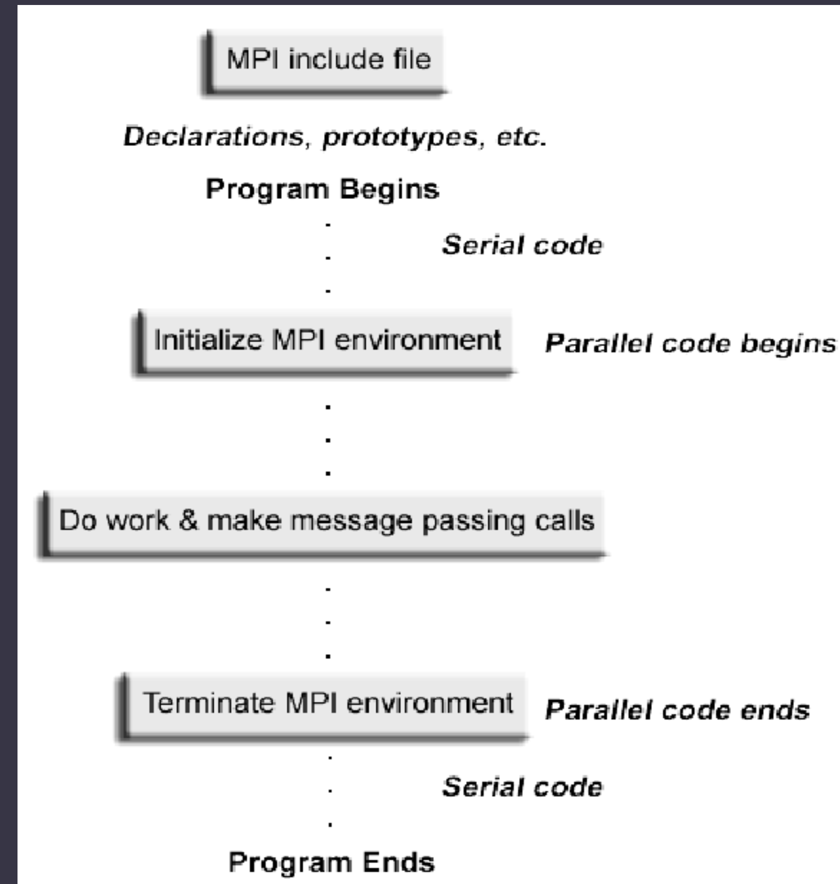


MPI (Message Parsing Interface):

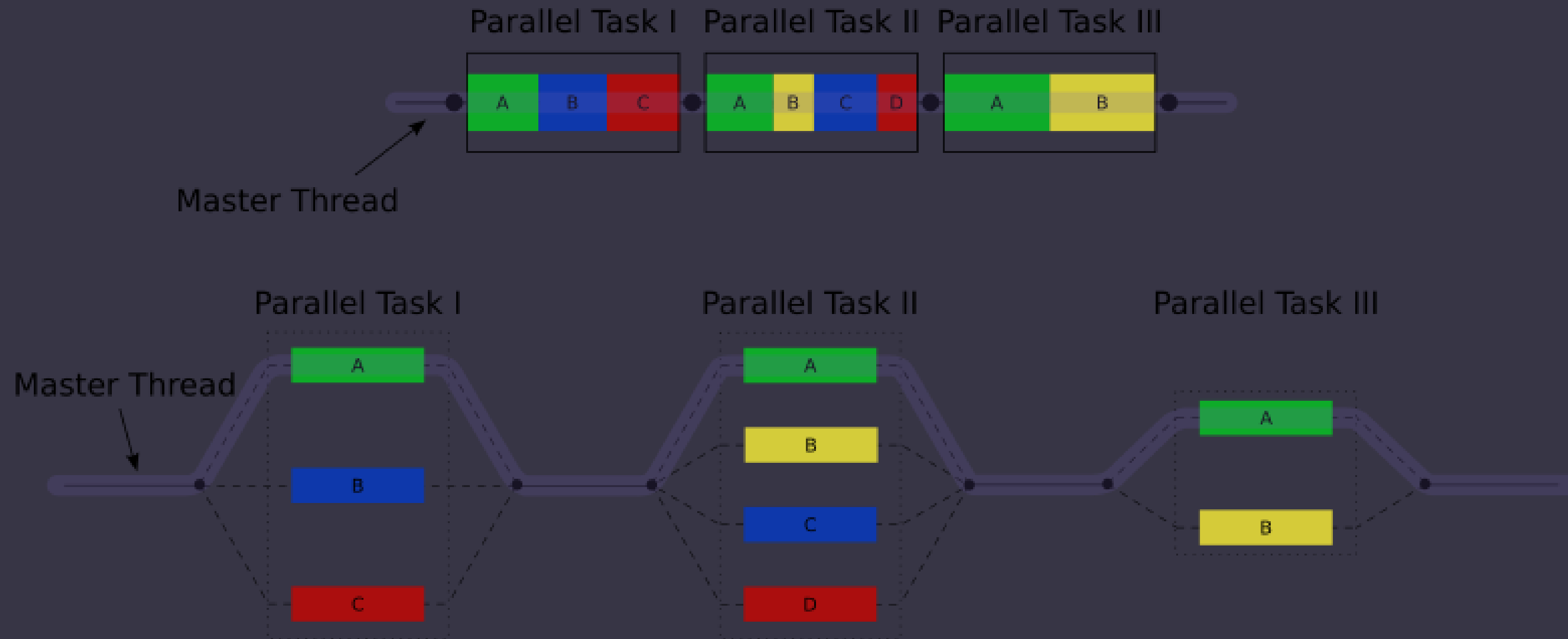




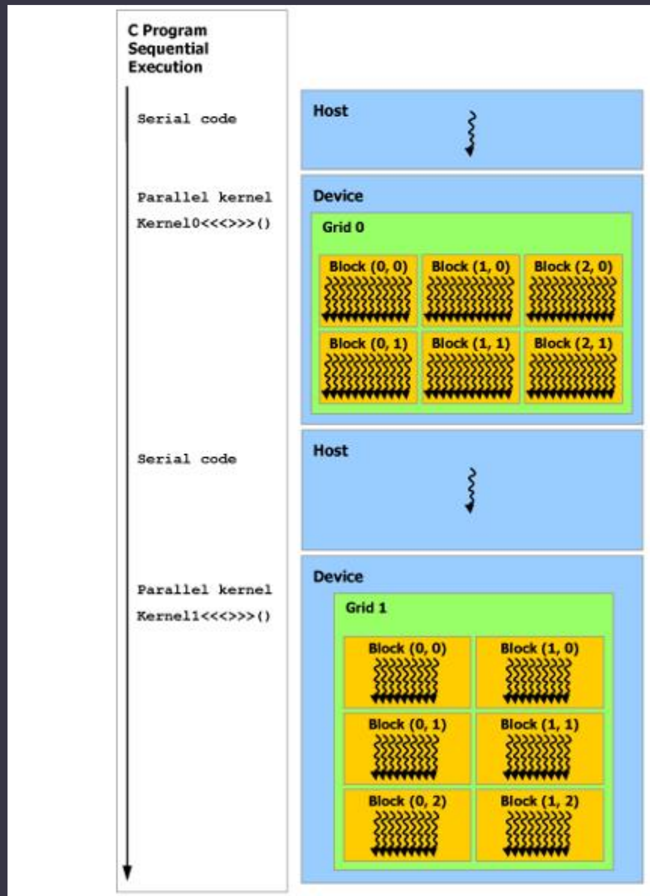
Structure of Communication (MPI)



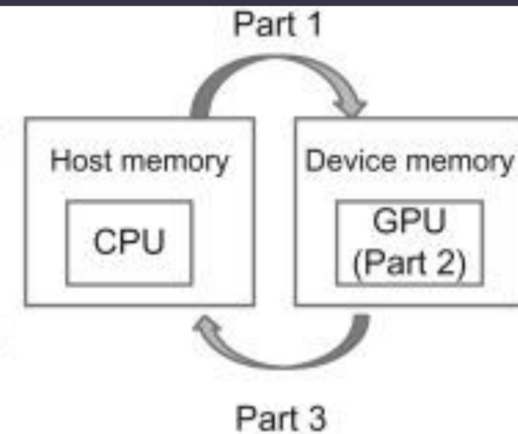
Structure of Communication (OpenMP)



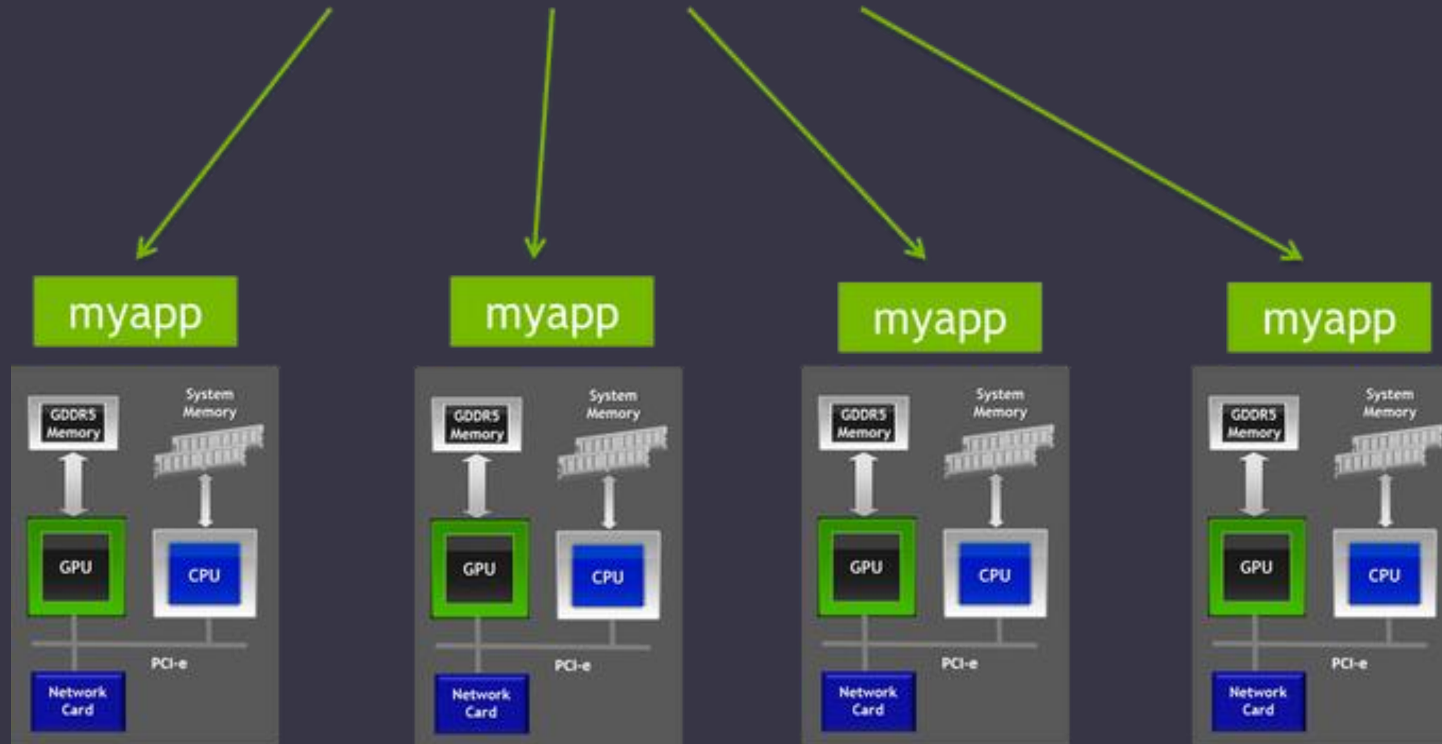
Structure of Communication (CUDA)



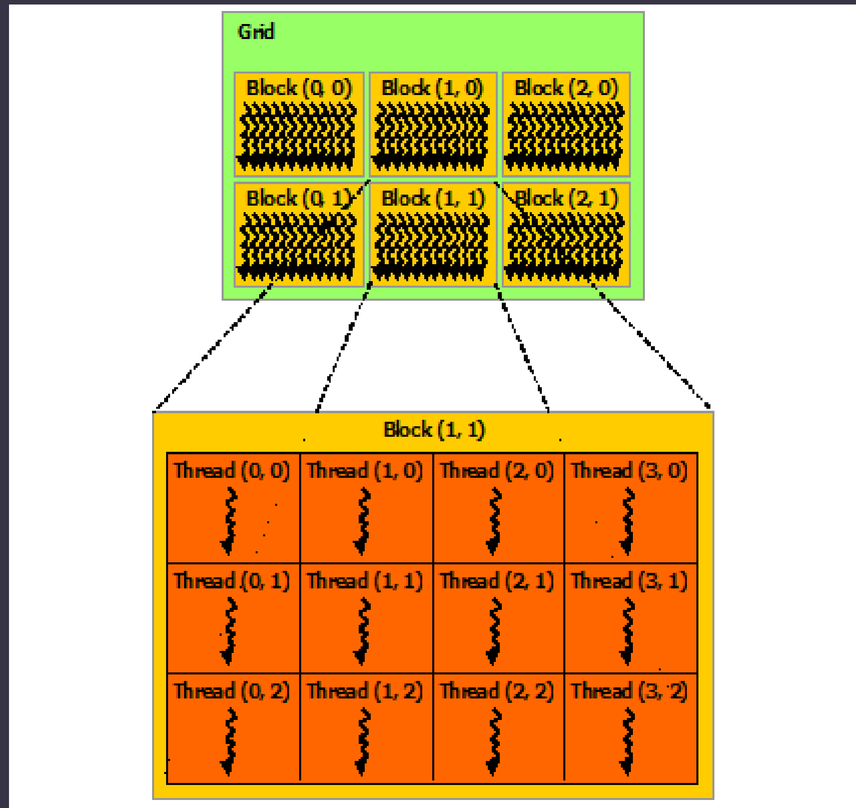

```
#include <cuda.h>
...
void vecAdd(float* A, float* B, float* C, int n)
{
  int  size = n* sizeof(float);
  float  *d_A *d_B, *d_C;
  ...
1. // Allocate device memory for A, B, and C
   // copy A and B to device memory

2. // Kernel launch code – to have the device
   // to perform the actual vector addition

3. // copy C from the device memory
   // Free device vectors
}
```

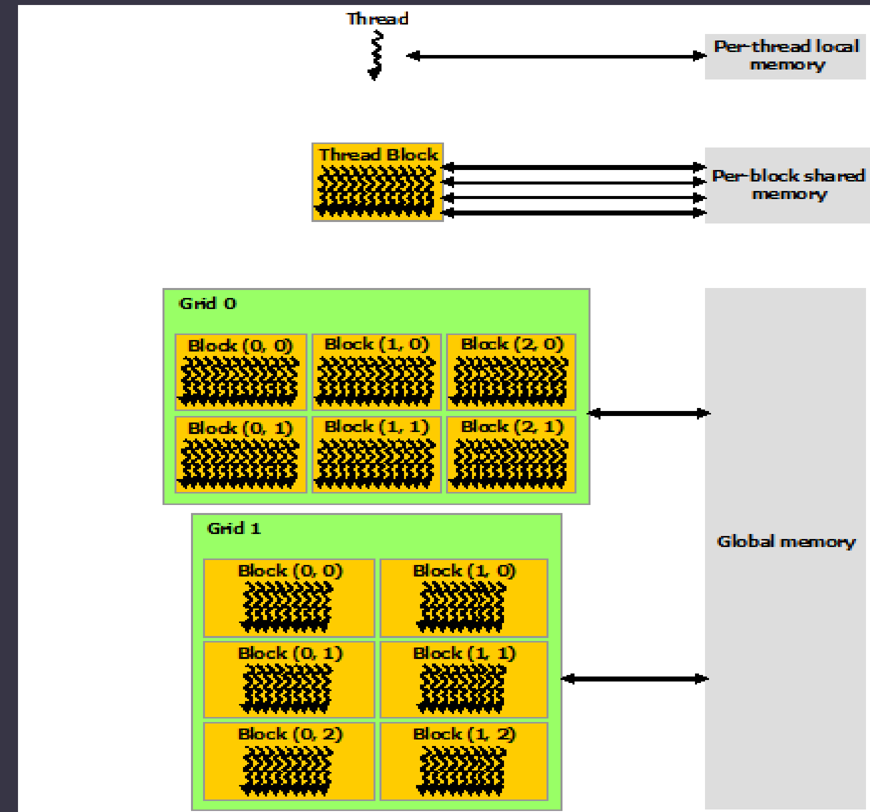


MPI and CUDA



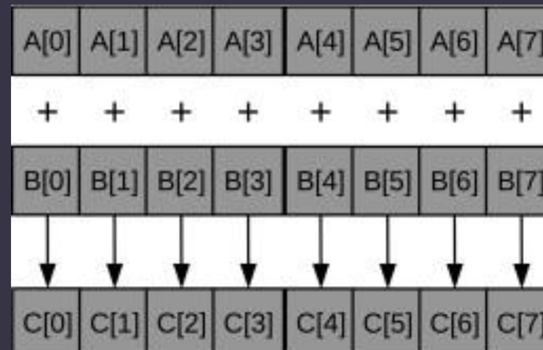
Grid of Thread Blocks / Memory Hierarchy





Simple Vector Operation

```
// Simple Kernel //PSEUDO C Code
void Vector_Addition (int *C , const int *A , int *B, int N)
{
    for(int i=0; i < N; ++i)
        {
            C[i] = A[i] + B[i];
        }
}
```



Kernel for Mulitprocessor Systems

```
// Perform an element-wise addtion of A and B and store in C.
// There are N elements per array and NP CPU cores.
void Vector_Addition (int *C , const int *A , const int *B, int N, int NP, int tid)
{
    int ept = N/NP;
    for(int i=tid*ept; i < (tid+1)*ept; ++i)
        {
            C[i] = A[i] + B[i];
        }
}
```

Kernel for CUDA

```
// CUDA Kernel for Vector Addition
__global__ void Vector_Addition ( const int *dev_a , const int *dev_b , int *dev_c)
{
    //Ge the id of thread within a block
    unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x ;

    while ( tid < N ) // check the boundry condition for the threads
    {
        dev_c [tid] = dev_a[tid] + dev_b[tid] ;
        tid+= blockDim.x * gridDim.x ;
    }
}
```

How to get Thread Position (tid) ?

Block 0	Thread 0	Thread 1	Thread 2	Thread 3
Block 1	Thread 0	Thread 1	Thread 2	Thread 3
Block 2	Thread 0	Thread 1	Thread 2	Thread 3
Block 3	Thread 0	Thread 1	Thread 2	Thread 3

Figure 5.1 A two-dimensional arrangement of a collection of blocks and threads

int **offset = x + y * DIM;**

tid = threadIdx.x + blockIdx.x * blockDim.x.

Getting enough Threads ?

In fact, we will launch too few threads whenever N is not an exact multiple of 128. This is bad. We actually want this division to round up. There is a common trick to accomplish this in integer division without calling ceil(). We actually compute (N+127)/128 instead of N/128. Either you can take our word that this will compute the smallest multiple of 128 greater than or equal to N or you can take a moment now to convince yourself of this fact.

Callin the Kernel:

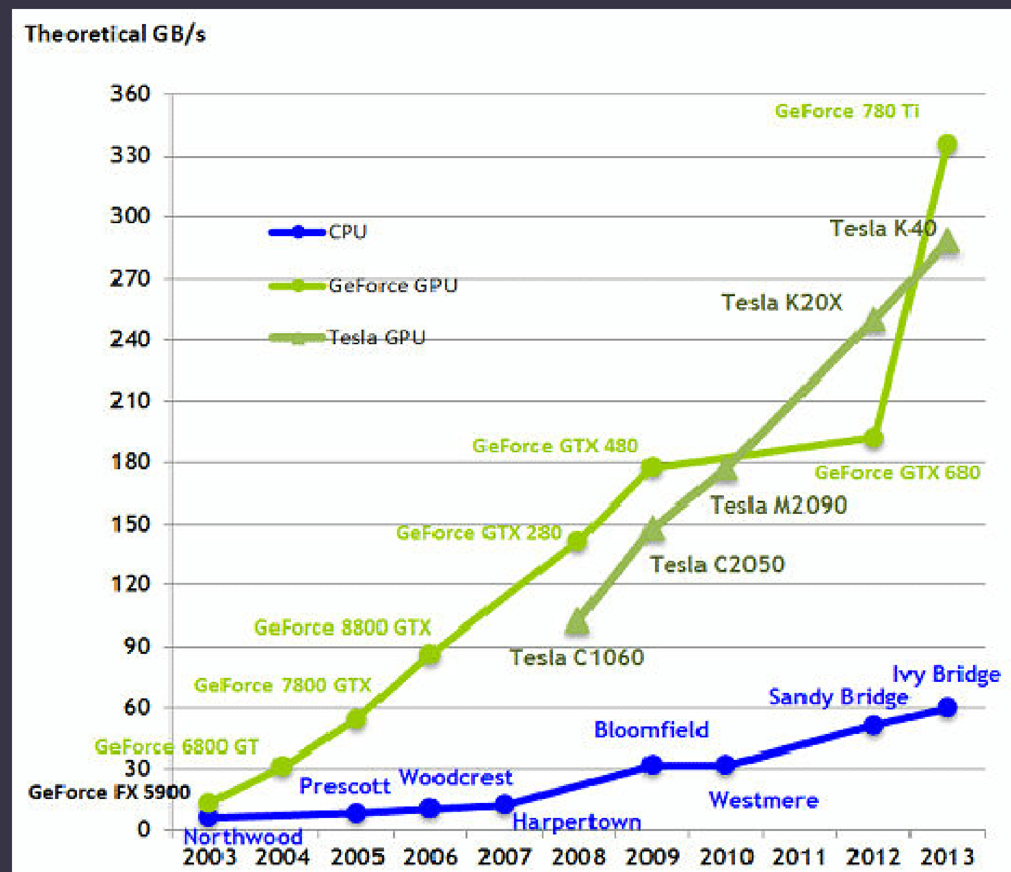
```
Vector_Addition <<< ( N + 127 ) / 128, 128 >>> ( dev _ a, dev _ b, dev _ c );
```

Because of our change to the division that ensures we launch enough threads, we will actually now launch *too many* threads when N is not an exact multiple of 128.

Restrict Memory Access in the Kernel:

```
if (tid < N)
        c[tid] = a[tid] + b[tid];
```

CPU vs GPU Floating Point Operations



The End