Parallel Processing/Programming

with the applications to image processing

Lectures:

1. Parallel Processing & Programming – from high performance mono cores to multi- and many-cores
2. Programming Interfaces (API) for multi-cores, many-cores and heterogeneous programming

Exercises:

1. High performance mono-cores and multi-cores; programming with **`openMP`**
2. Architecture of modern GPUs; many-core processing and programming with **`CUDA`**

SmartComputerLab

Parallel Processing/Programming

with the applications to image processing

Labs:

1. High performance processing with pipelines and cache-memories; simple multi-core programming with **openMP**

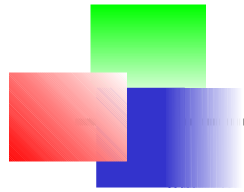
2. **openMP** programming for parallel image processing

3. Simple many-core programming with **CUDA** – vector and matrix processing

4. Many-core parallel image processing with **CUDA** and **openCV**

5. Many-core parallel image generation and animation with **CUDA** and **openGL**

All labs are prepared on embedded boards: Odroid-U3 (**openMP**) and Tegra K1 (**openMP** & **CUDA**)



Performance ?

Performance = 1/Time.to.process.the.given.task= 1/TT

TT= Number.of.Instructions*

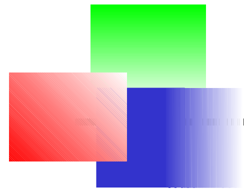
Number.of.Clock.Cycles.per.Instruction.*

Time.of.Clock.Cycle

Number.of.Instructions – task complexity (CISC or RISC)

Number.of.Clock.Cycles.per.Instruction – processor architecture & micro-parallelism

Time.of.Clock.Cycle (or **1/Clock.Frequency**) - technology



Performance – an example

Performance = 1/Time.to.process.the.given.task= 1/TT

TT= Number.of.Instructions*

Number.of.Clock.Cycles.per.Instruction.*

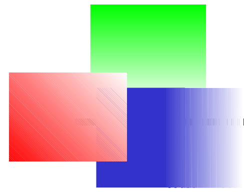
Time.of.Clock.Cycle

Number.of.Instructions = $\mathbf{10^6}$

Number.of.Clock.Cycles.per.Instruction = **2**

Time.of.Clock.Cycle (or 1/Clock.Frequency) = **1 GHz**

What is execution time ?, What is performance ?



Number of Instructions

CISC – Complex Instruction Set Computer

eg. x86 (Intel,..)

+ less instructions (memory) per task

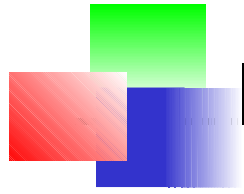
- multiple instruction formats, complex decoding

RISC – Reduced Instruction Set Computer

eg. ARM, MIPS, ..

- more instructions (memory) per task

+ few instruction formats, rapid decoding



Number of Clock Cycles per Instruction

Pipelining:

Elaboration of several instructions at the same time

Multi-scalar processing:

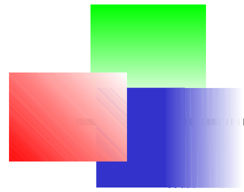
Multiple pipelines with multiple execution units: ALUs – fixed and floating point

Out-of-order processing:

Instruction queues with micro-scheduling, physical and virtual registers

Vector processing units:

Multiple data units processed by the same instruction

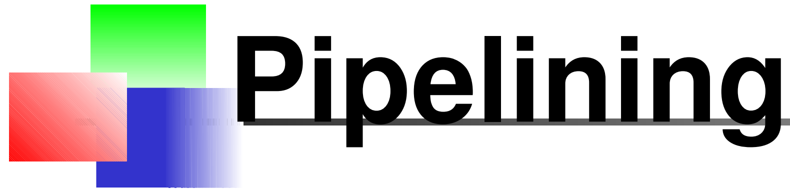


Pipelining

Elaboration of several instructions at the same time

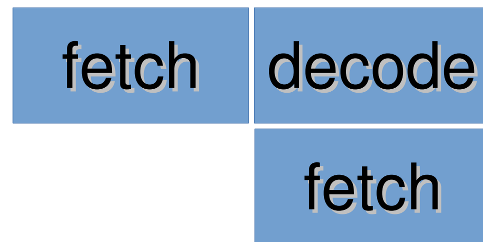
fetch

one clock cycle

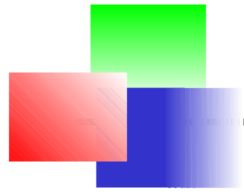


Pipelining

Elaboration of several instructions at the same time

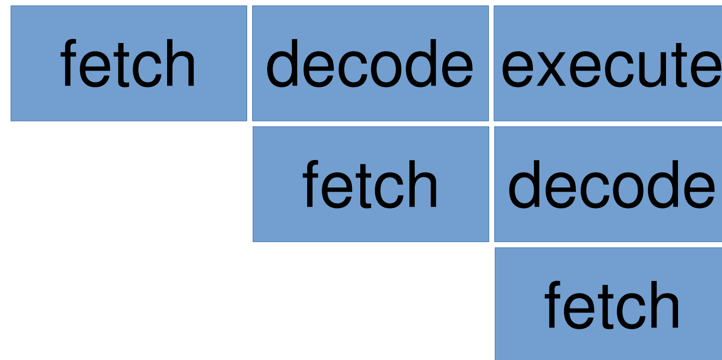


one clock cycle

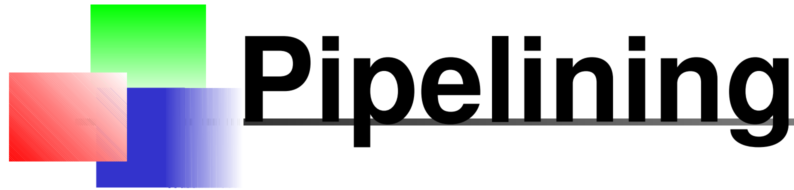


Pipelining

Elaboration of several instructions at the same time



one clock cycle

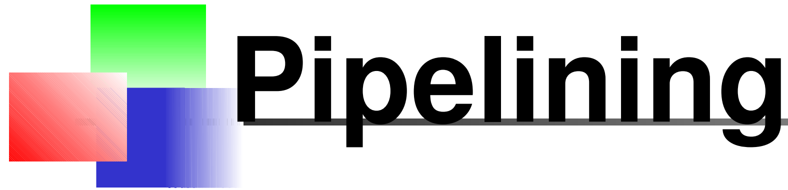


Pipelining

Elaboration of several instructions at the same time

fetch	decode	execute	write
	fetch	decode	execute
		fetch	decode
			fetch

one clock cycle

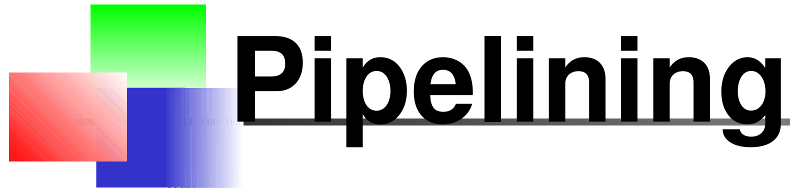


Pipelining

Elaboration of several instructions at the same time

fetch	decode	execute	write		
	fetch	decode	execute	write	
		fetch	decode	execute	
			fetch	decode	

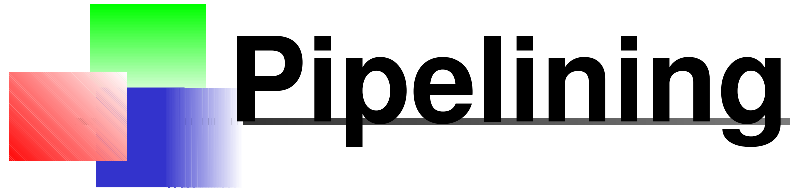
one clock cycle



Pipelining

fetch	decode	execute	write			
	fetch	decode	execute	write		
		fetch	decode	execute	write	
			fetch	decode	execute	

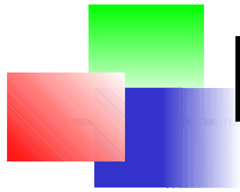
one clock cycle



Pipelining

fetch	decode	execute	write			
	fetch	decode	execute	write		
		fetch	decode	execute	write	
			fetch	decode	execute	write

one clock cycle



Pipelining and caching

L1-D

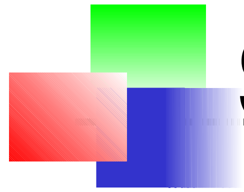
fetch	decode	execute	write			
	fetch	decode	execute	write		
		fetch	decode	execute	write	
			fetch	decode	execute	write

data memory access
Data Cache

instruction memory access
Instruction Cache

L1-I

one clock cycle

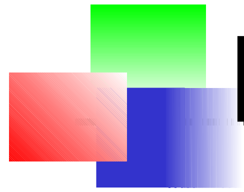


Super-scalar Pipelining

Execution of several instructions (here 2) at the same time

fetch	decode	execute	write			
fetch	decode	execute	write			
	fetch	decode	execute	write		
	fetch	decode	execute	write		
		fetch	decode	execute	write	
		fetch	decode	execute	write	
			fetch	decode	execute	write
			fetch	decode	execute	write

Max number of instructions per clock cycle is 2 ?

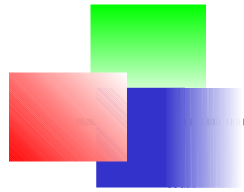


Pipelining & vector processing

Execution of the same instruction on several data units

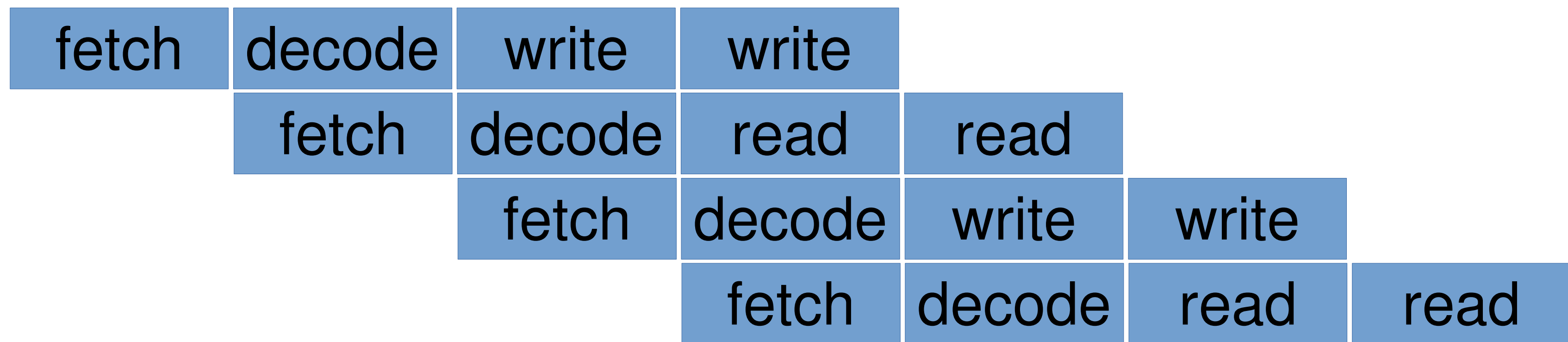
fetch	decode	execute	write			
	fetch	decode	execute	write		
		fetch	decode	execute	write	
			fetch	decode	execute	write

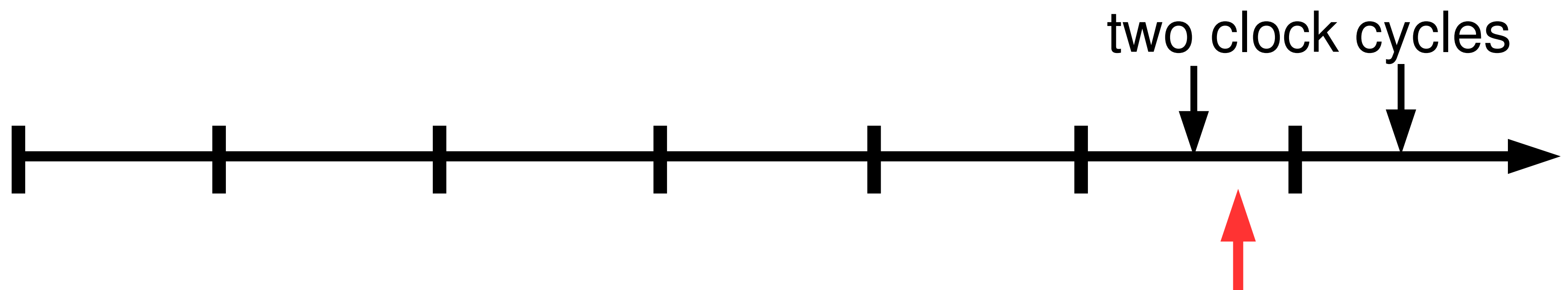
multiple data:
4 int/float



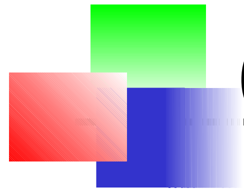
Pipelining and caching

Execution of the memory store/load instructions





L1 – cache hit/miss
if miss ~10 cycles stall for L2 access



Clock frequency

Performance = 1/Time.to.process.the.given.task= 1/TT

TT= Number.of.Instructions*

Number.of.Clock.Cycles.per.Instruction.*

Time.of.Clock.Cycle

or

Performance = Clock.Frequency *

1/(Number.of.Instructions*

Number.of.Clock.Cycles.per.Instruction)

Clock frequency

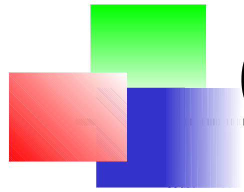
The problem is power consumption:

$$dynamic.power = A*N*C*V^2*f$$

where:

A – activity rate, **N** – number of driven transistors, **C** – input capacity of transistor node, **V** – voltage,

f – frequency (**?**)



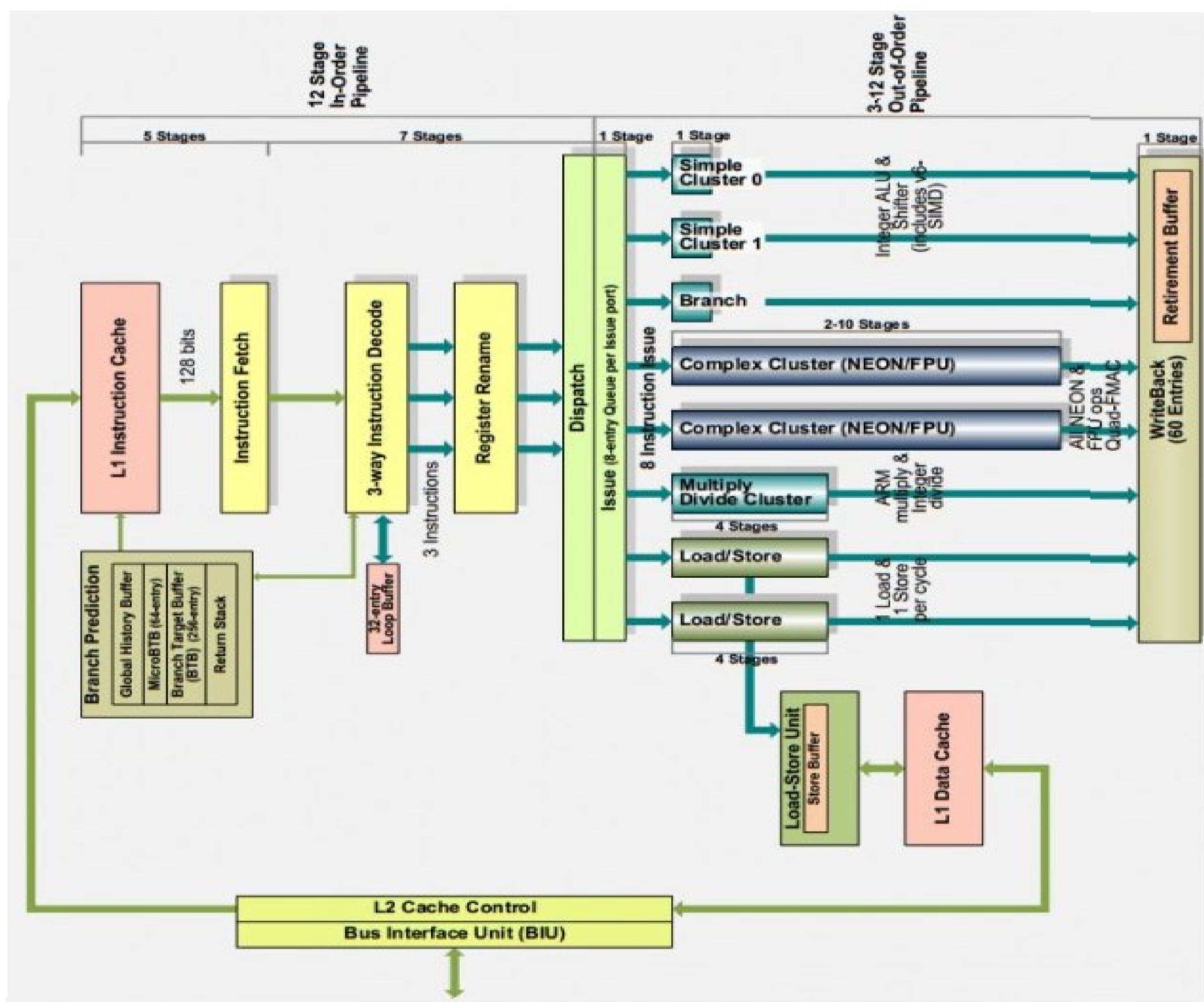
Clock frequency & voltage

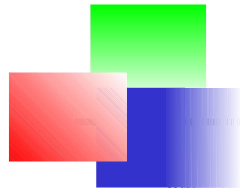
Let us increase the frequency.
What is the impact on the dynamic power consumption ?
Where is the problem ?

dynamic.power = A*N*C*V^2*f

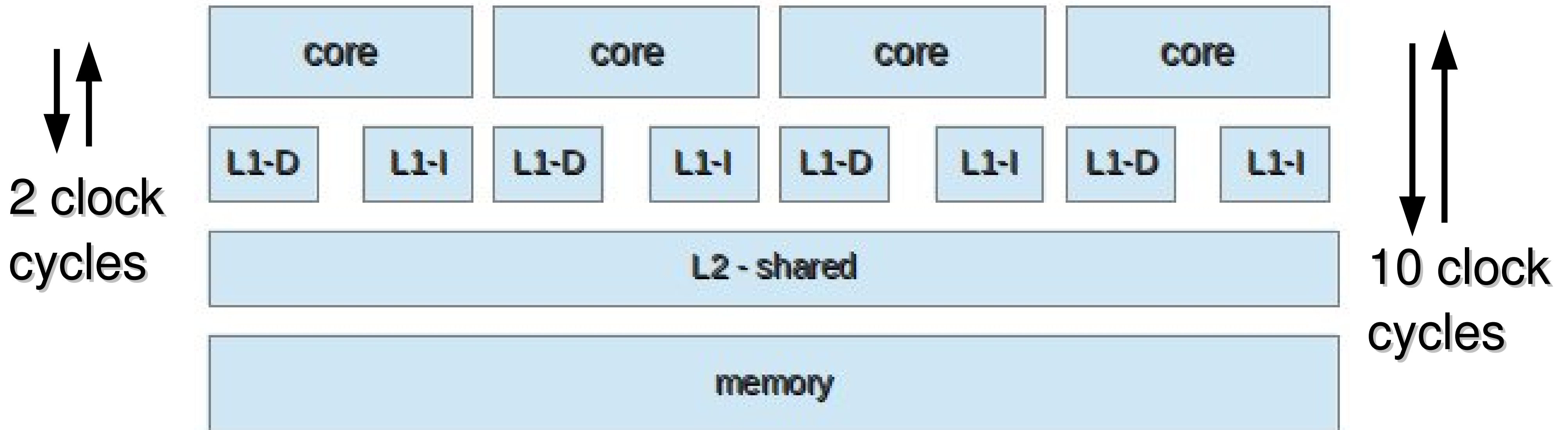
The answer is: the increase of the frequency involves the increase of the voltage.
For example the increase of the frequency by the factor 2 needs the increase of the voltage by the the factor of **1.66**;
so the dynamic power consumption increase is **?**

ARM Cortex-A15 architecture



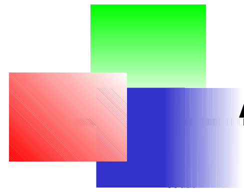


Multi-core architecture

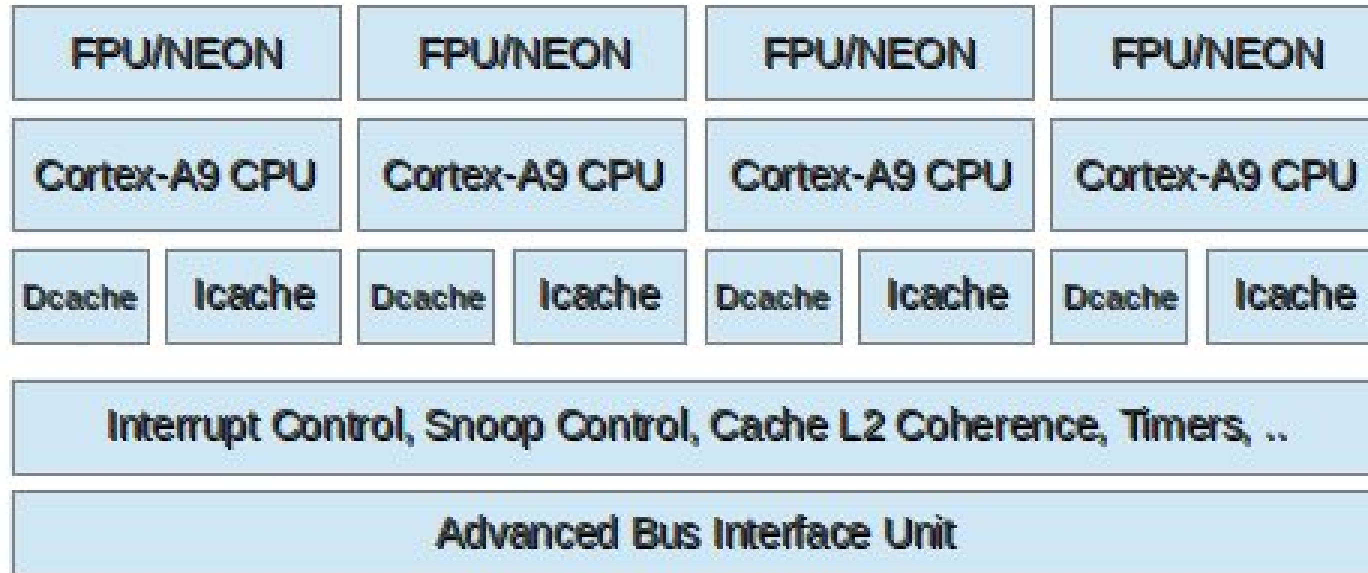


SMP – Symmetric Multi-Processor with shared memory

Independent L1 instruction and data cache (L1-D, L1-I)
Shared L2 cache

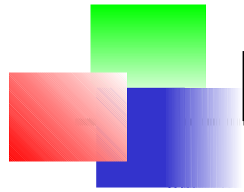


ARM Cortex-15 MPCore



ARM Cortex-15 : the most powerful ARM v7 processor

- 32 KB L1 caches,
- 4 MB L2 cache,
- clock up to 2.5 GHz
- 1TB RAM memory address space



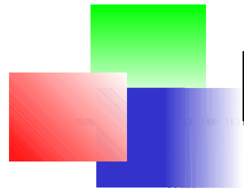
Multi-core programming with openMP

Parallel processing speed-up:

speedup = 1/(S+1/N*(1-S))

where:
S – serial part of the task
N – number of processors

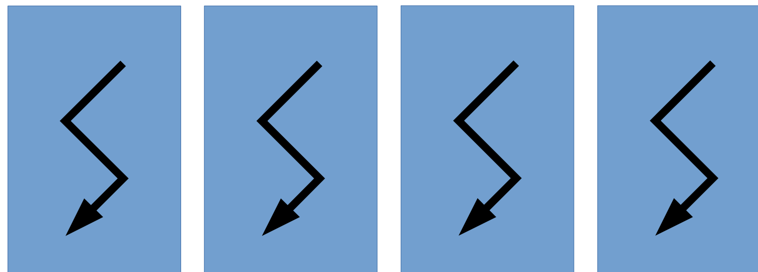
Example:
For S=10% and N=4 the speedup is ?



Multi-core programming with `openMP`

`openMP` operates mostly via compiler **directives**

```
#pragma omp ...
{
...code...
}
```



automatic threads

where the ...**`code`**... is called the parallel region

Multi-core programming with openMP

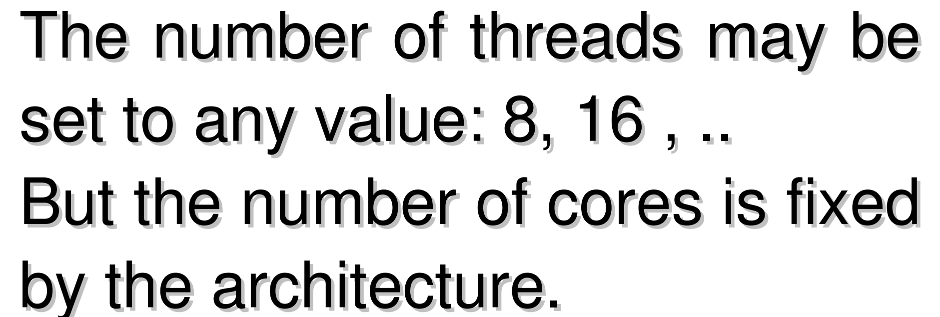
```
#include "stdio.h"  #include <omp.h>

int main(int argc, char *argv[])
{
  #pragma omp parallel
  {
    printf("hello multicore user!\n");
  }
  return(0);
}

%cc  -o Hello.omp Hello.omp.c -fopenmp

%./Hello.omp
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
```

openMP environment

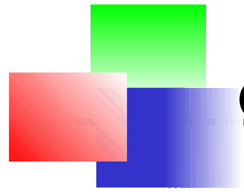
```
#include "stdio.h"  #include <omp.h>
int main(int argc, char *argv[])
{
  #pragma omp parallel
  {
    int NCPU,tid,NPR,NTHR;
    NCPU = omp_get_num_procs(); // get the number of available cores
    tid = omp_get_thread_num(); // get current thread ID
    NPR = omp_get_num_threads(); // get total number of threads
    NTHR = omp_get_max_threads();// get number of threads requested
    if (tid == 0) {              // execute it in master thread
      printf("%i : NCPU\t= %i\n",tid,NCPU);
      printf("%i : NTHR\t= %i\n",tid,NTHR);
      printf("%i : NPR\t= %i\n",tid,NPR);
    }
    printf("%i: I am thread %i out of %i\n",tid,tid,NPR);
  }
return(0);
}
```

openMP environment

```
%cc  -o HelloMulticore HelloMultiCore.c -fopenmp

export OMP_NUM_THREADS=4

%./ HelloMulticore

1 :  I am thread 1 out of 4
2 :  I am thread 2 out of 4
0 : NCPU = 4
0 : NTHR = 1
0 : NPR = 4
0 :  I am thread 0 out of 4
3 :  I am thread 3 out of 4
```

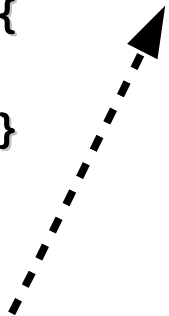


The number of threads may be set to any value: 8, 16 , ..
But the number of cores is fixed by the architecture.



openMP local and global variables

```
int x;
#pragma omp parallel for
for(x=0; x<width; x++)    // shared variable x
{
        for(int y=0; y < height; y++) // local-private variable y
        {
                OutImage[x][y] = RenderPixel(x,y, &InImage );
        }
}
```



implicit local variable

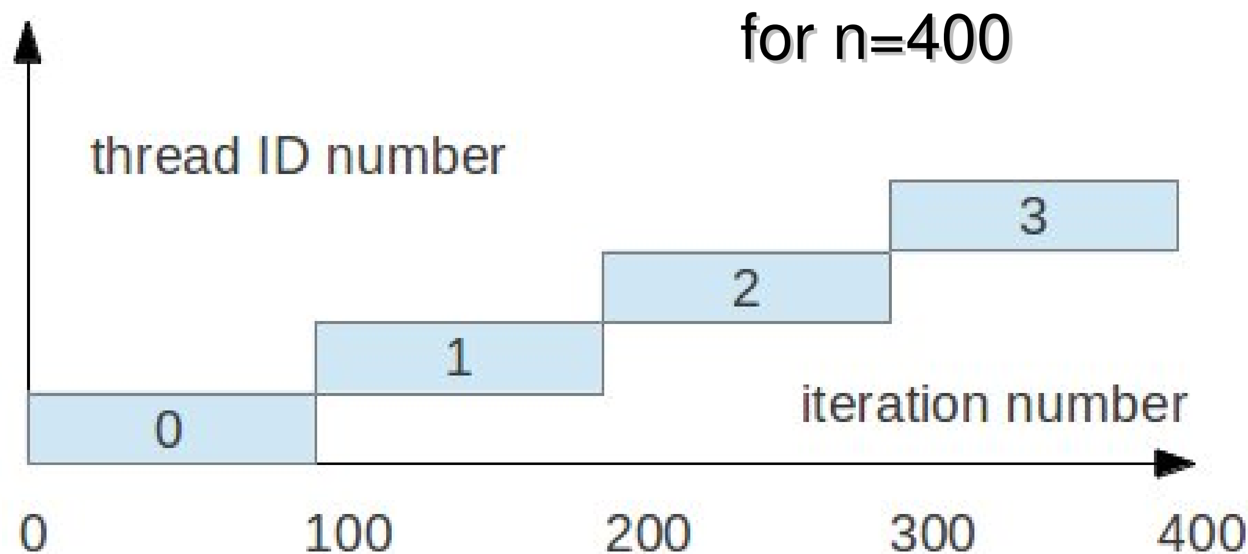
openMP local and global variables

```
int x;
#pragma omp parallel for
for(x=0; x<width; x++)   // shared variable x
{
        for(int y=0; y < height; y++) // local-private variable y
        {
                OutImage[x][y] = RenderPixel(x,y, &InImage );
        }
}

int x,y;
#pragma omp parallel for private(y)
for(x=0; x < width; x++)
{
        for(y=0; y < height; y++)
        {
                OutImage[x][y] = RenderPixel(x,y, &InImage);
        }
}
```

explicit local variable

openMP loop scheduling

```
int i,j,n;
#pragma omp parallel for default(none) schedule(static) private(i,j)
shared(n)
for (i=0; i<n; i++)
{
printf("Iteration %d executed by thread %d\n", i, omp_get_thread_num());
for (j=0; j<i; j++)
system("sleep 1");
} /*-- End of parallel for --*/
```



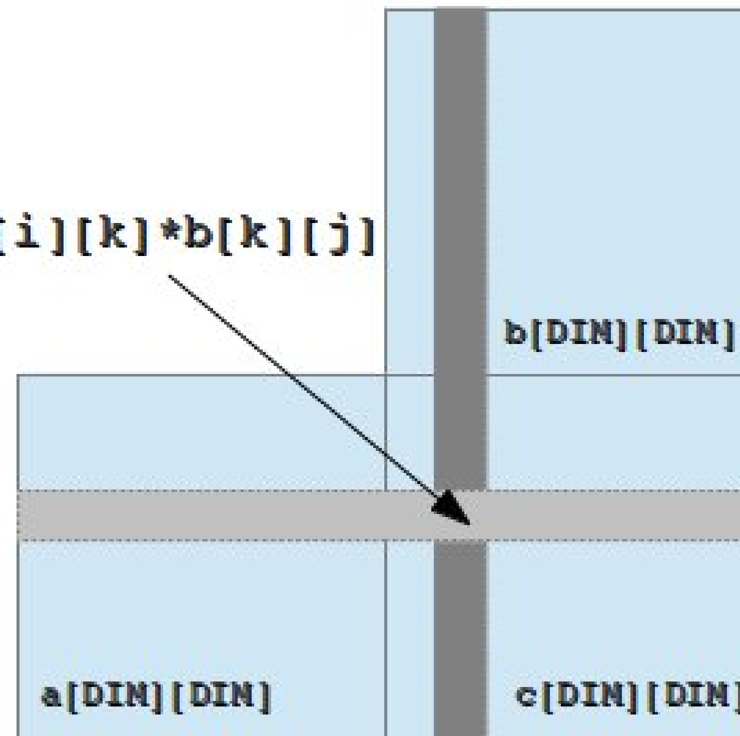
openMP reduction operator

```
#define N 1000
int main (int argc, char *argv[]) {
  double a[N], b[N];
  double sum = 0.0;
  int i, n, tid;
#pragma omp parallel shared(a) shared(b) private(i)
  {
    tid = omp_get_thread_num();
  #pragma omp for
    for (i=0; i < N; i++) {
      a[i] = 1.0;
      b[i] = 1.0;
    }

  #pragma omp for reduction(+:sum)
    for (i=0; i < N; i++) {
      sum += a[i]*b[i];
    }

  } /* End of parallel region */
  printf("Sum = %2.1f\n",sum);
  exit(0);
}
```

The products are calculated in parallel;
the sum is a unique value

openMP matrix multiplication

```
int DIM=512;
#pragma omp parallel for private(i,j,k,dot) shared(a,b,c)
for(i=0;i<DIM;i++) {
  for(j=0;j<DIM;j++) {
    dot=0.0;
    for(k=0;k<DIM;k++)
    dot += a[i][k]*b[k][j];
    c[i][j]=dot; }
}
```



openMP matrix multiplication

```
#pragma omp parallel for private(i,j,k,dot) shared(a,b,c) firstprivate(DIM)
for(i=0;i<DIM;i+=4) {
  for(j=0;j<DIM;j++) {
    dot[0]=dot[1]=dot[2]=dot[3]=0.0;
    for(k=0;k<DIM;k++) {
      dot[0] += a[i+0][k]*b[k][j];
      dot[1] += a[i+1][k]*b[k][j];
      dot[2] += a[i+2][k]*b[k][j];
      dot[3] += a[i+3][k]*b[k][j];   }
    c[i+0][j]=dot[0];
    c[i+1][j]=dot[1];
    c[i+2][j]=dot[2];
    c[i+3][j]=dot[3];}
}
```

- 4 operations are done in parallel in each thread
- `firstprivate(DIM)` specifies that each thread should have its own instance of a variable, and that the variable should be initialized with the value of the variable as it exists before the parallel construct

Summary

- Elements of high performance architectures
- Instruction set architecture (ISA)
- Micro-parallelism
- Clock frequency

- Example of ARM Cortex-15

- Why we need multi-core architectures ?

- Basic multi-core programming with **`openMP`**