Multi-core Processors

Processor development till 2004

- Moore's Law: transistor count doubles every 18 months
  - Used to improve processor performance by 2x every 18 months
  - Single core, binary compatible to previous generations

- Contributors to performance improvements
  - More ILP through OOO techniques
    - Wider issue, better prediction, better scheduling, …
    - Better memory hierarchies
  - Clock frequency improvements

Out-of-order
Instruction scheduling

Historical Perspective

- 80’s till 2004 the exponential increase in the number of transistors predicted by Moore’s Law (2x /18 mons.) was used:
  - Implement sophisticated O.O.O & Superscalar designs
  - Design Large Caches
- Increasing operating frequency

Although successful for several years, this approach has eventually hit the **Power, Memory and ILP walls**

Solution: Chip Multiprocessors (CMP)

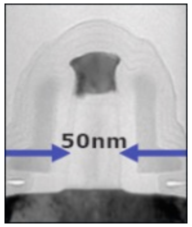
Use the transistors to increase the number of cores on a chip

New challenge: **Concurrency**

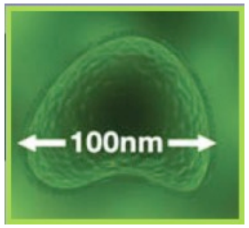
The Switch to Multi-core

The number of transistors is still increasing but more aggressive wider-issue, higher-clocked superscalars are not produced anymore, Why??

Transistor



Influenza Virus



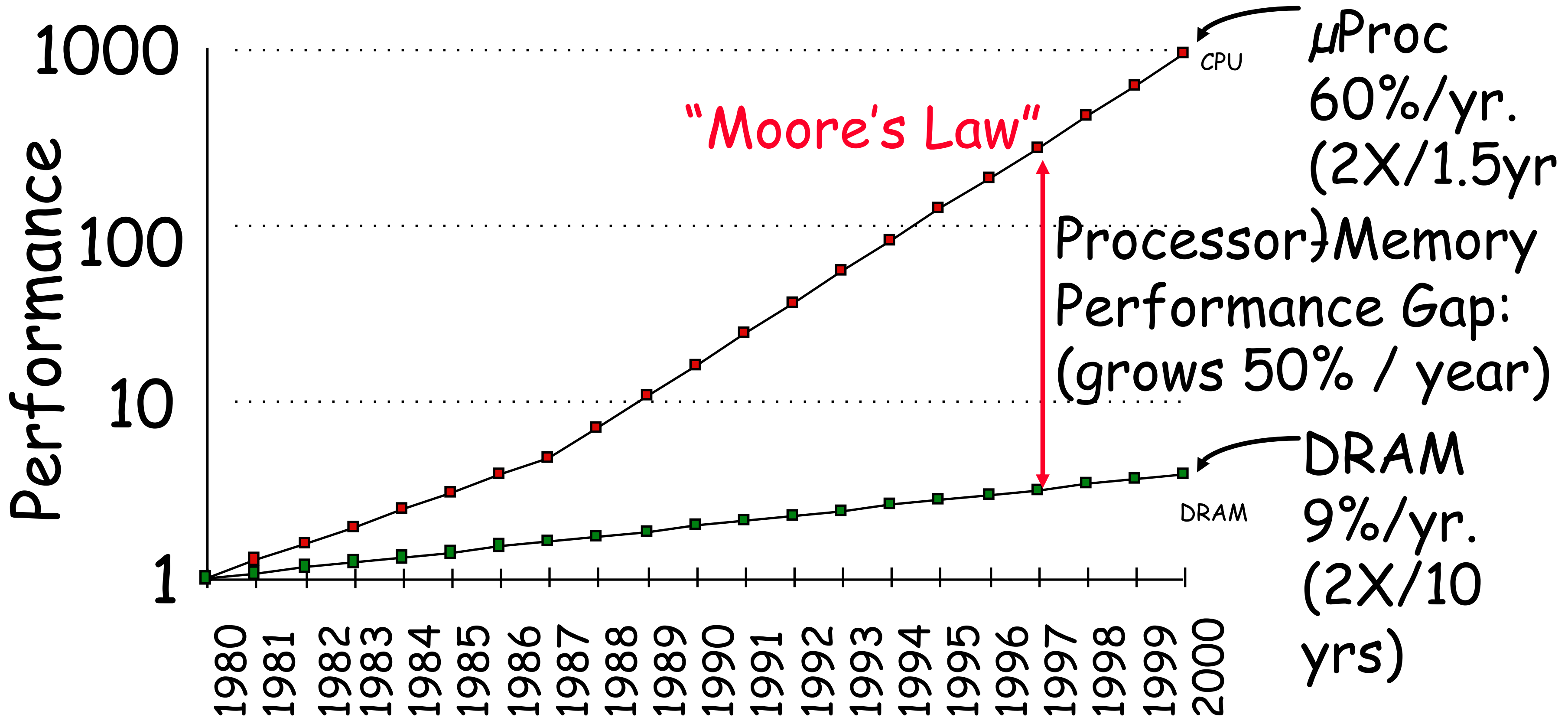
	2004	2006	2008	2010	2012	2014	2016	2018
Feature Size	90nm	65nm	45nm	32nm	22nm	16nm	11nm	8nm
Billions Transis	2	4	8	16	32	64	128	256 ⁎⁎

- Complexity of the designs: power, leakage, heat, design difficulty, deep pipelines
- Power/Heat (Power Wall):heat, bills cooling, packaging, hot-spot, deep pipelines, cannot increase clock frequencies
- Lack of ILP (ILP Wall): ILP rarely exceeds 7, with average 5
- Marginal gain of incremental logic:

N: transistors Perf = O(sqrt(N)) Power O(N)

⁎⁎Source: Intel

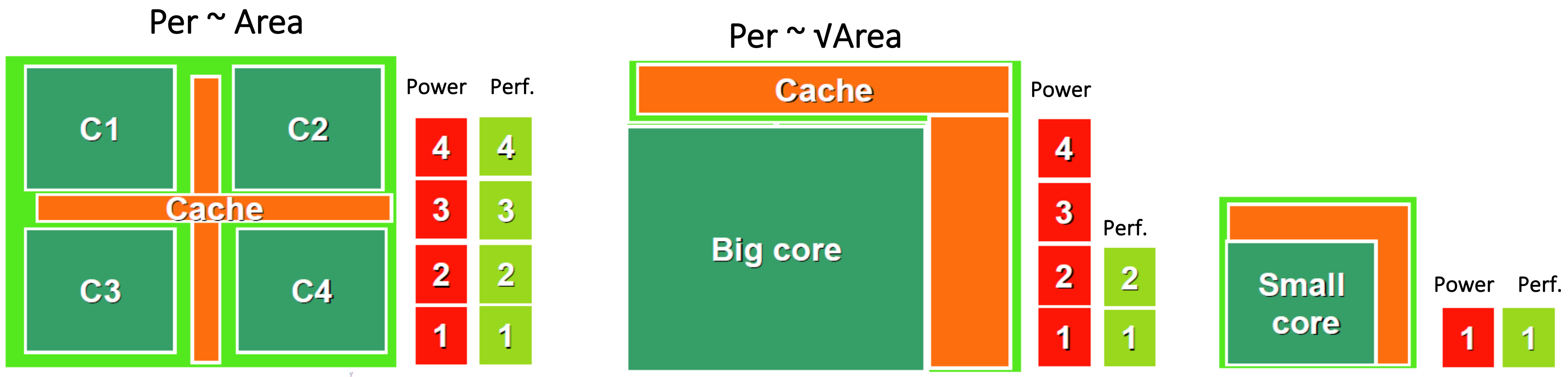
The Memory Wall



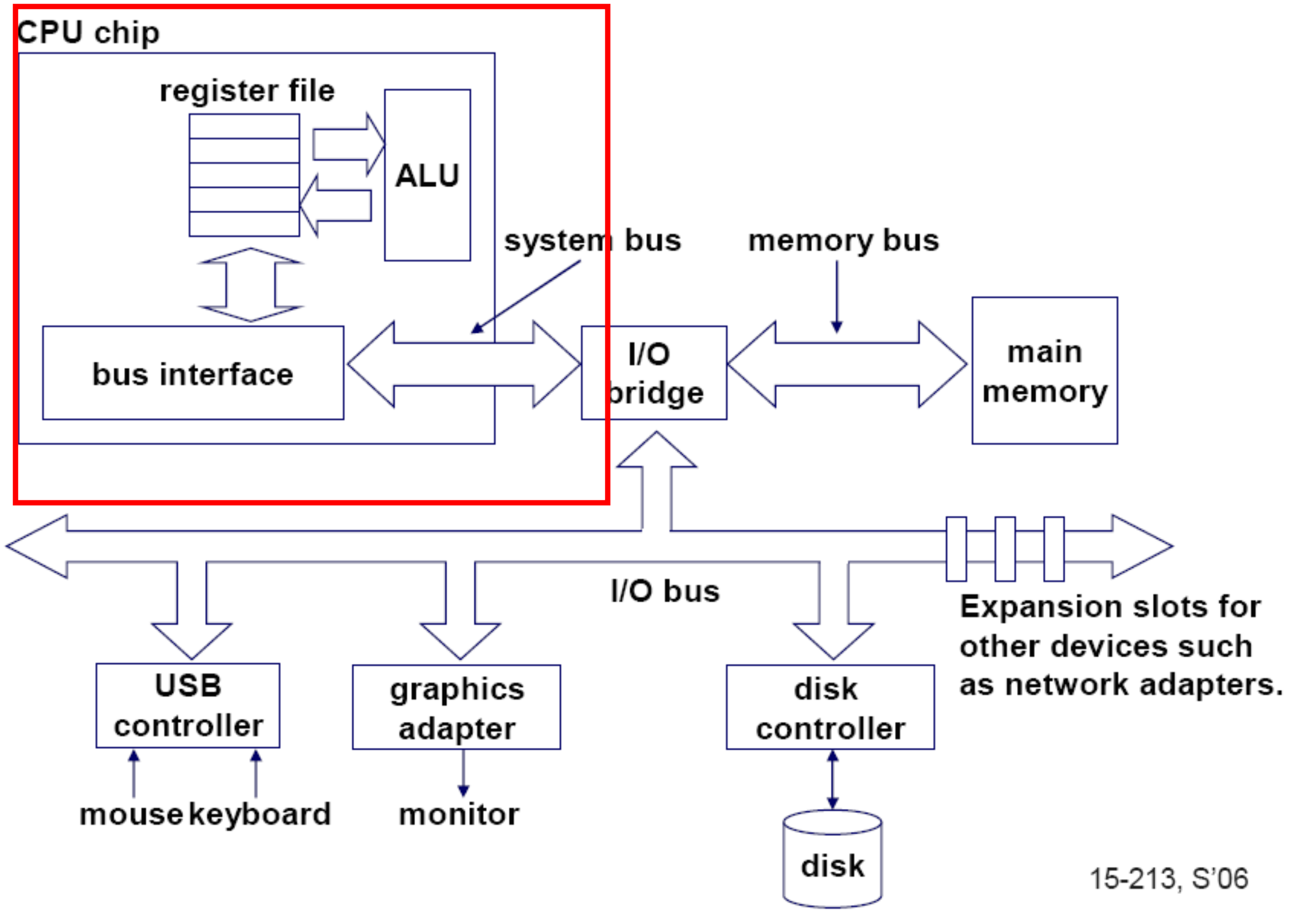
The Multi-core/Concurrency Era (Why Multi-core?)

- The solution for the previous problems is:
  - Use the silicon estate to put more processors on the same die
  - More area/power efficient
  - Scale the processor without complex designs
  - Shift the focus to extracting Thread Level Parallelism (TLP)

    Currently “throughput programming”

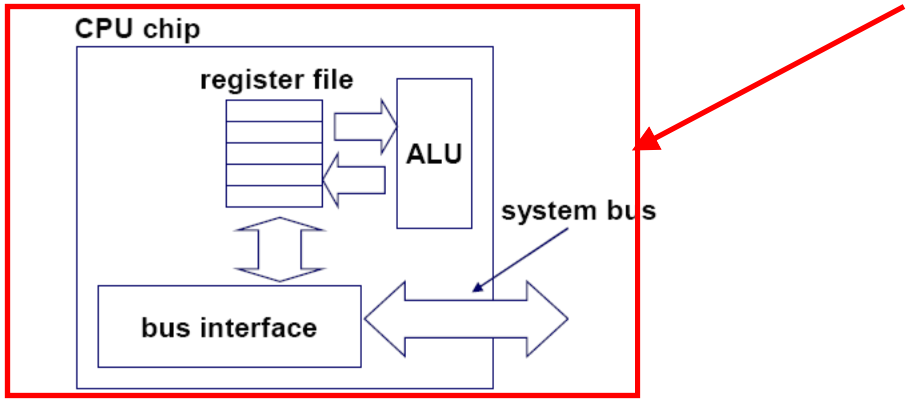


Single-core computer



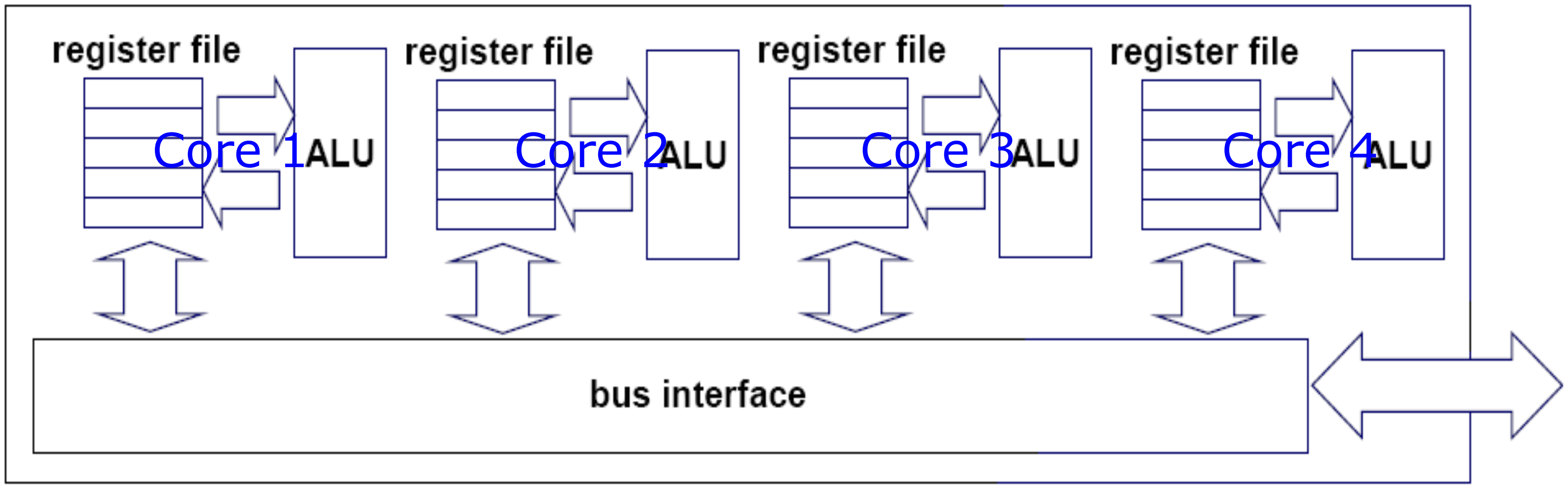
Single-core CPU chip

the single core



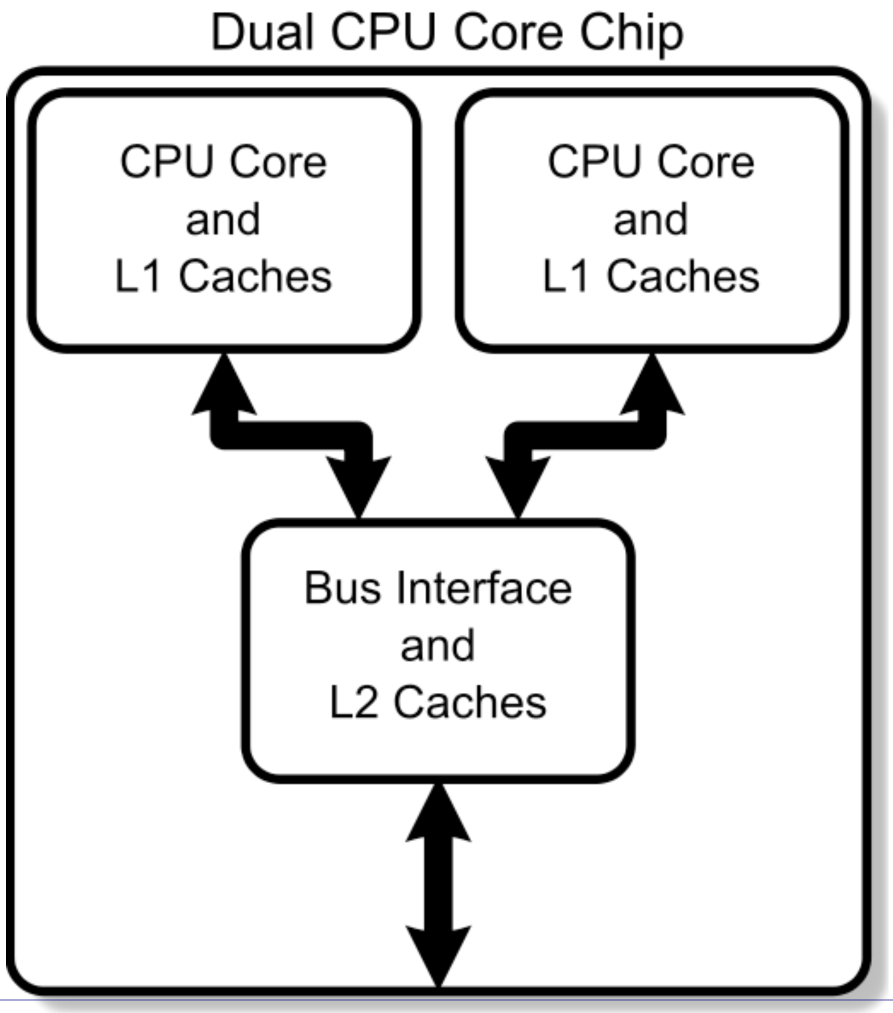
Multi-core architectures

Replicate multiple processor cores on a single die.



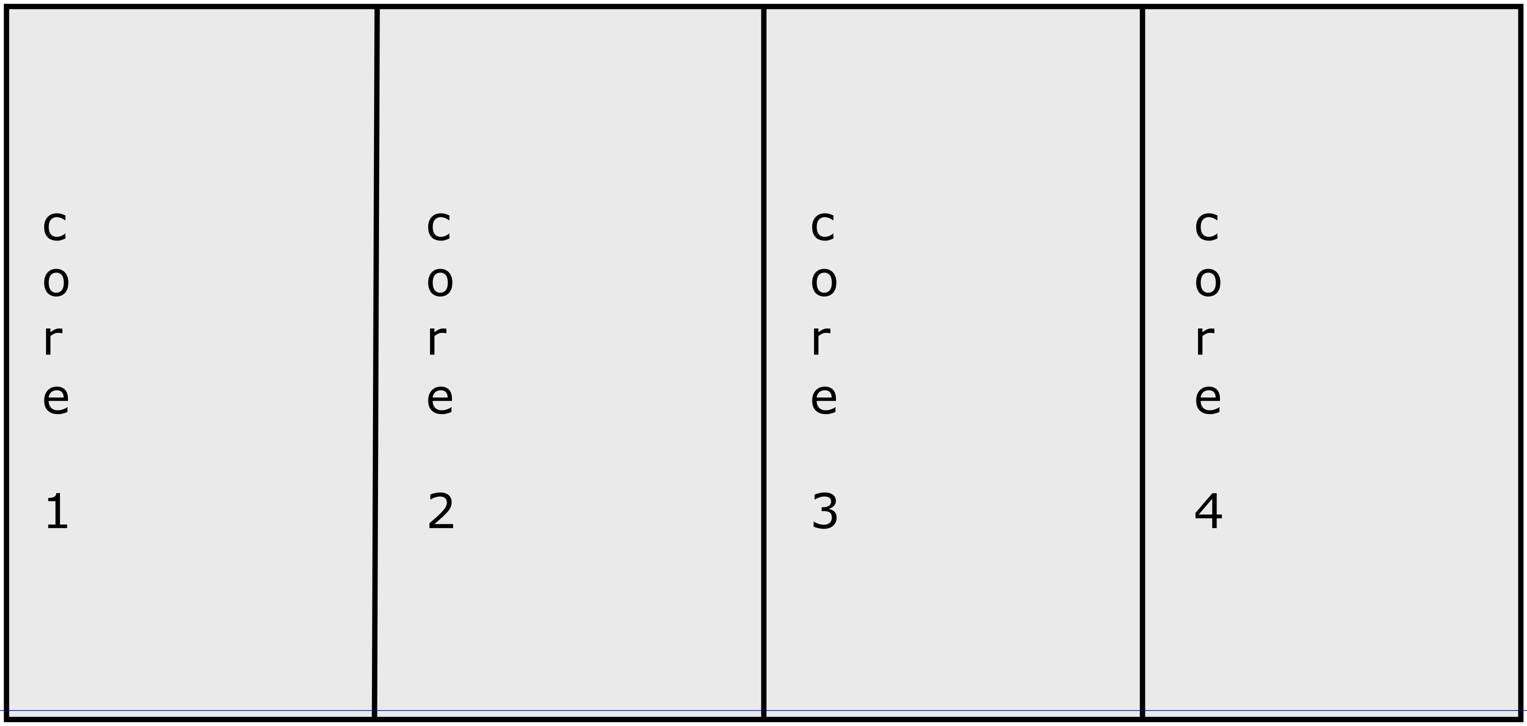
Multi-core CPU chip

Multi-core processor

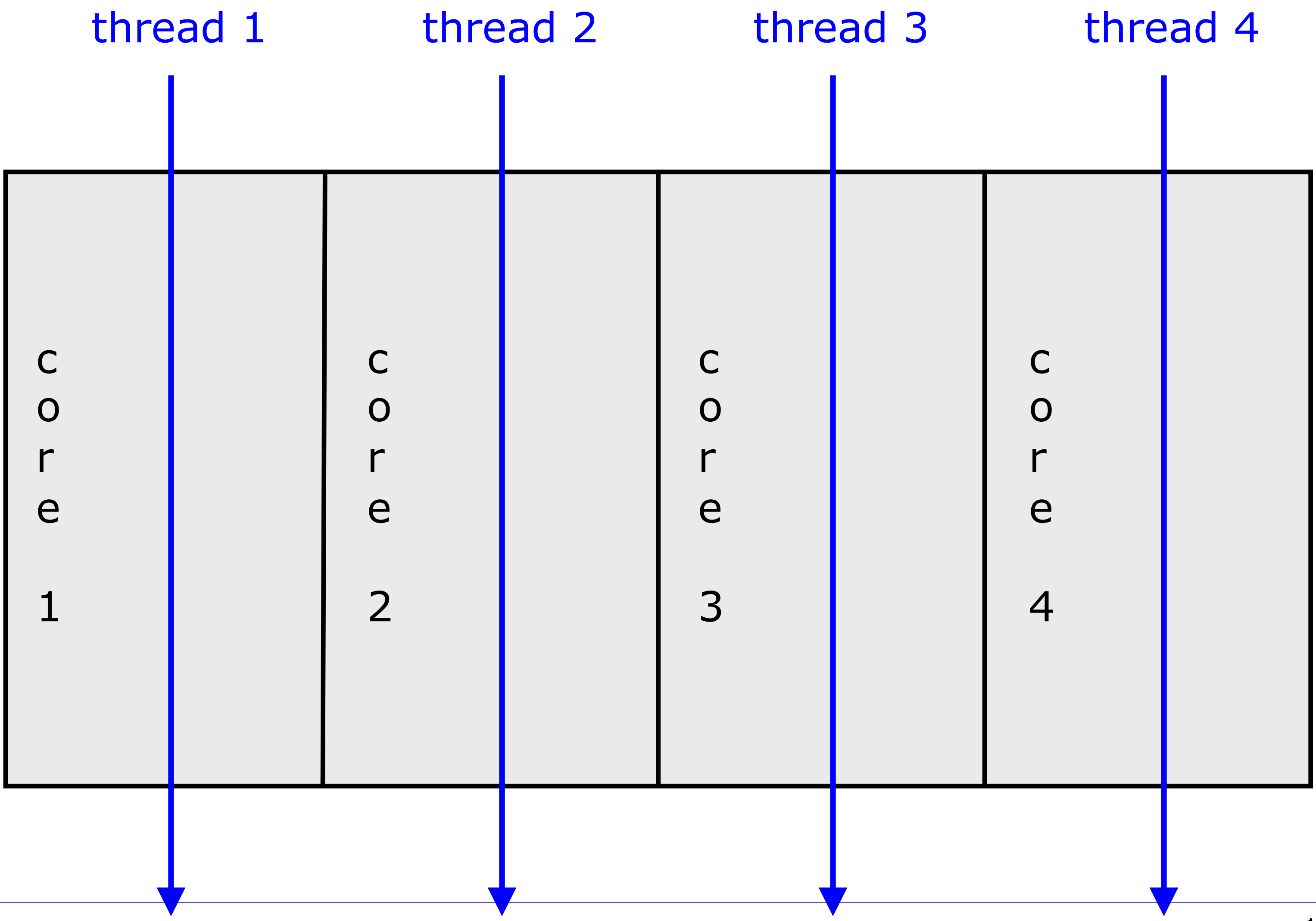


Multi-core CPU chip

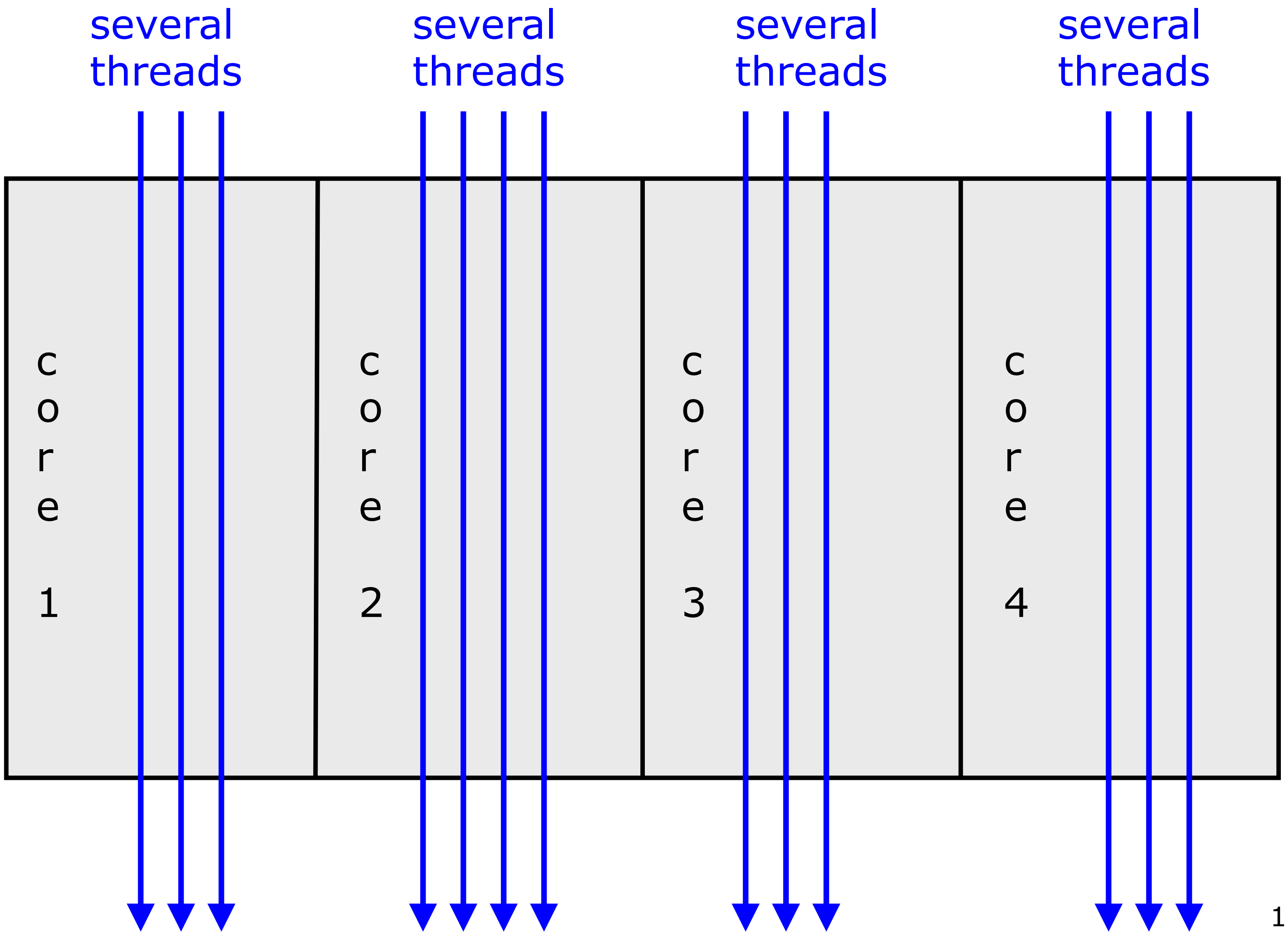
- The cores fit on a single processor socket
- Also called CMP (Chip Multi-Processor)



The cores run in parallel



Within each core, threads are time-sliced (just like on a uniprocessor)



Interaction with the Operating System

- OS perceives each core as a separate processor
- OS scheduler maps threads/processes to different cores
- Most major OS support multi-core today: Windows, Linux, Mac OS X, …

Instruction-level parallelism (ILP)

- Parallelism at the machine-instruction level
- The processor can re-order, pipeline instructions, split them into microinstructions, do aggressive branch prediction, etc.
- Instruction-level parallelism enabled rapid increases in processor speeds over the last 15 years

Thread-level parallelism (TLP)

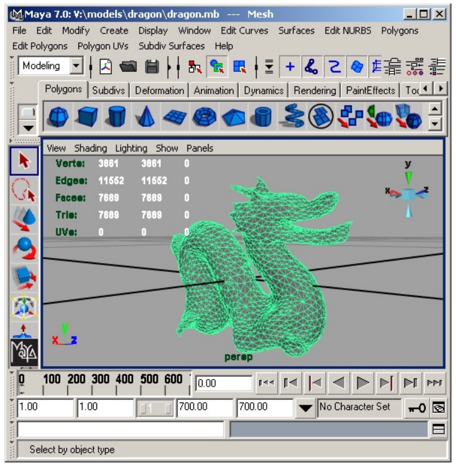
- This is parallelism on a more coarser scale
- Server can serve each client in a separate thread (Web server, database server)
- A computer game can do AI, graphics, and a physics-related computation in three separate threads
- Single-core superscalar processors cannot fully exploit TLP
- Multi-core architectures are the next step in processor evolution: explicitly exploiting TLP

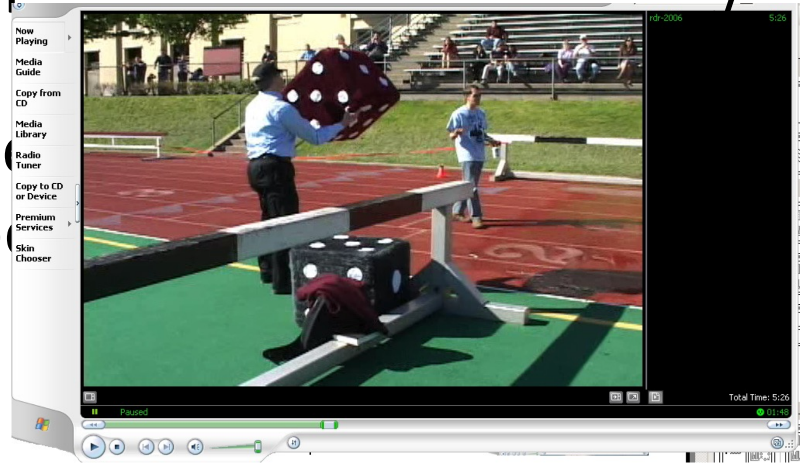
Data Level Parallelism (DLP)

- Also called SIMD or Vectorization
- More on this later

What applications benefit from multi-core?

- Database servers
- Web servers (Web commerce)
- Multimedia applications
- Scientific applications, CAD/CAM
- In general, applications with *Thread-level parallelism* (as opposed to instruction-l parallelism) are better supp


Maya 7.0: V:\models\dragon\dragon.mb --- Mesh
File Edit Modify Create Display Window Edit Curves Surfaces Edit NURBS Polygons
Edit Polygons Polygon UVs Subdiv Surfaces Help
Modeling
Polygons Subdivs Deformation Animation Dynamics Rendering PaintEffects
View Shading Lighting Show Panels
Verts: 3861 3861 0
Edges: 11552 11552 0
Faces: 7689 7689 0
Tris: 7689 7689 0
UVs: 0 0 0
persp
No Character Set
Select by object type



Each can run on its own core

More examples

- Editing a photo while recording a TV show through a digital video recorder
- Downloading software while running an anti-virus program
- "Anything that can be threaded today will map efficiently to multi-core"
- BUT: some applications difficult to parallelize

Multi-core design issues & Challenges

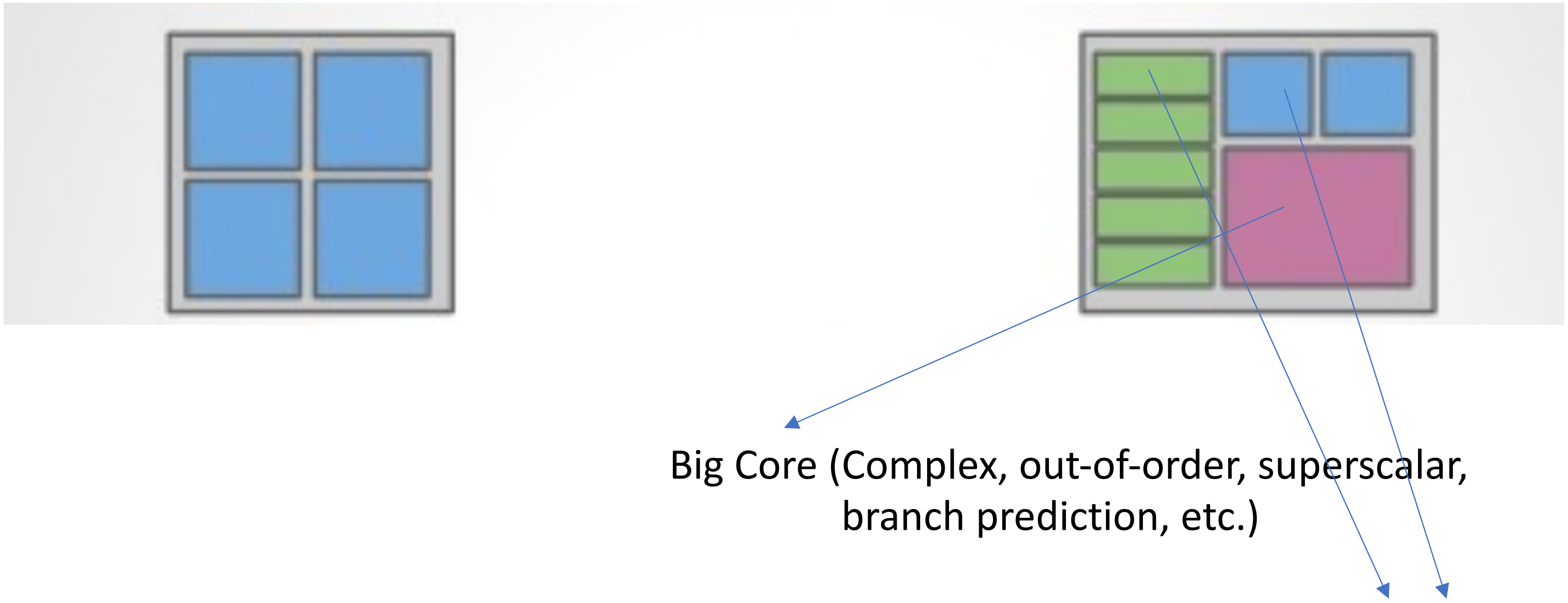
- Programming multi-cores (Concurrency) – is *Difficult and is the biggest challenge now*
  - ILP was implicit v.s. TLP and DLP explicit
  - Parallelizing Compilers
  - Parallel Lang./Extensions
  - H.W support
- On-chip Memory (Memory Wall) & Interconnection Network:
  - How to Scale up to tens/hundreds of cores: need on chip memory
  - Cache design/Shared vs. private Caches/Coherency (space vs latency)
  - Interconnect choices significantly affects overall performance
  - Bus/Point-to-Point/Cross-bar/Ring/Mesh/Omega (cost vs latency)
- What is the best building blocks?
  - Simple/complex/SMT/in-order/out-of-order cores?
  - Homogeneous or heterogeneous?
- Best Balance between Cores/Caches/Interconnect?

Heterogeneous Multicores

Heterogeneous vs. Homogeneous

Homogeneous Multi-core

Heterogeneous Multi-core



Big Core (Complex, out-of-order, superscalar, branch prediction, etc.)

Small Core (Simple, focus on ALU.)

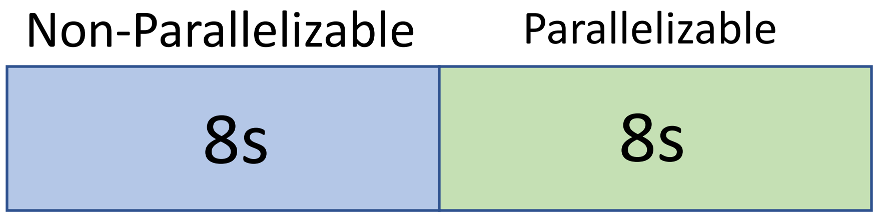
Heterogeneous Multicores

- Offers advantages compared to homogeneous multicores in:
  - **Power/Area:** code requiring complex out-of-order execution with branch-predication/speculation runs on the complex cores (fast power inefficient cores).

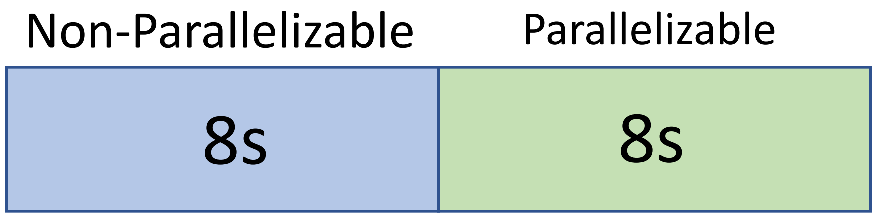
    Less control-flow code requiring ALU and data parallel (ex. SIMD/Vector)   (multiple threads running on many cores)
  - **Mitigating Amdahl's Law:** execute serial portions on the fast and complex core and parallel portions on the smaller cores.
  - **Different types of applications run simultaneously on these cores.**

Amdahl’s Law

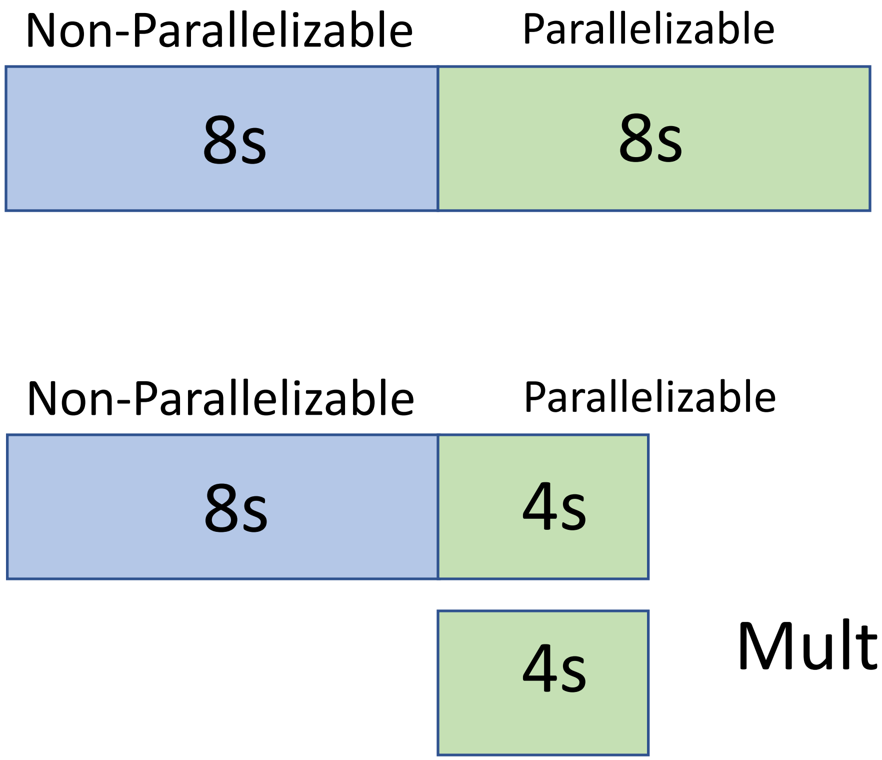
The performance of a parallel program is limited by the serial (non-parallelizable) part of the program



Single Core

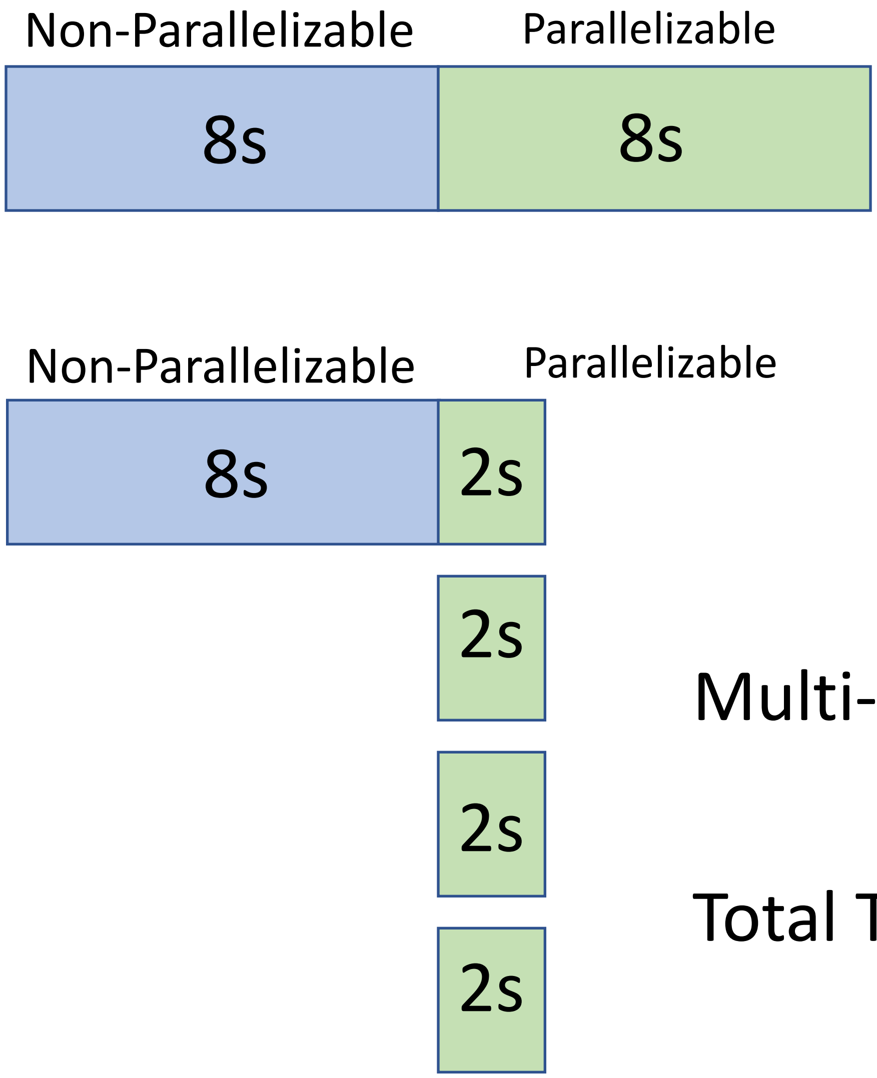


Total Time: 16s



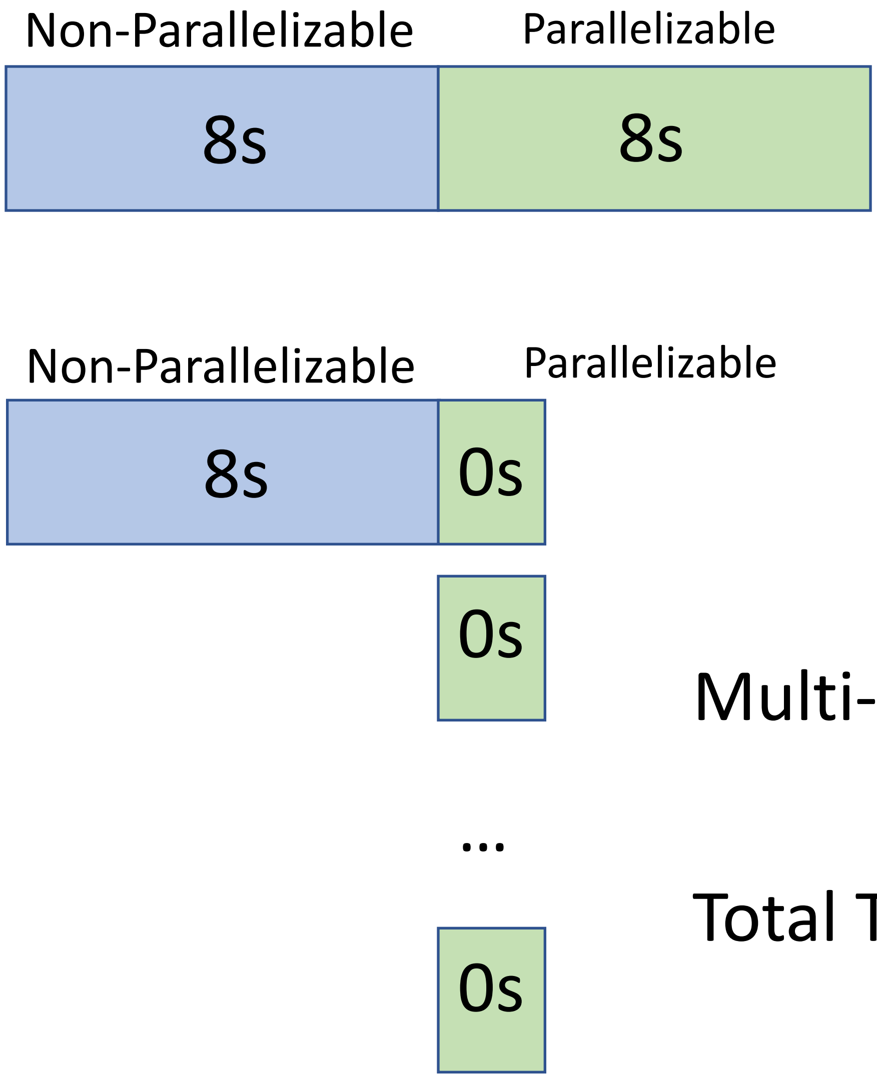
Multi-Core (2 cores)

Total Time: 12s



Multi-Core (4 cores)

Total Time: 10s



Multi-Core (∞ cores)

Total Time: 8s

Thread Level Parallelism (TLP)

SIMD

Vectorization

DLP

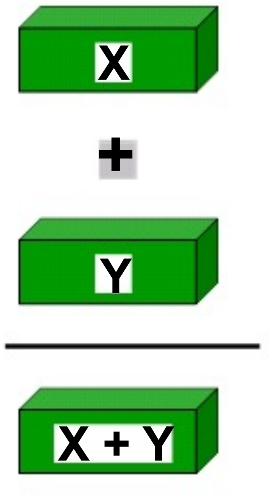
Motivation

- Intel analyzed multimedia applications and found they share the following characteristics:
  - Small native data types (8-bit pixel, 16-bit audio)
  - Recurring operations (same instruction on many pieces of large data sets).
  - Inherent parallelism

- This gave birth to Vector/SIMD support at the level of instructions in mainstream-processors.

Scalar vs. SIMD

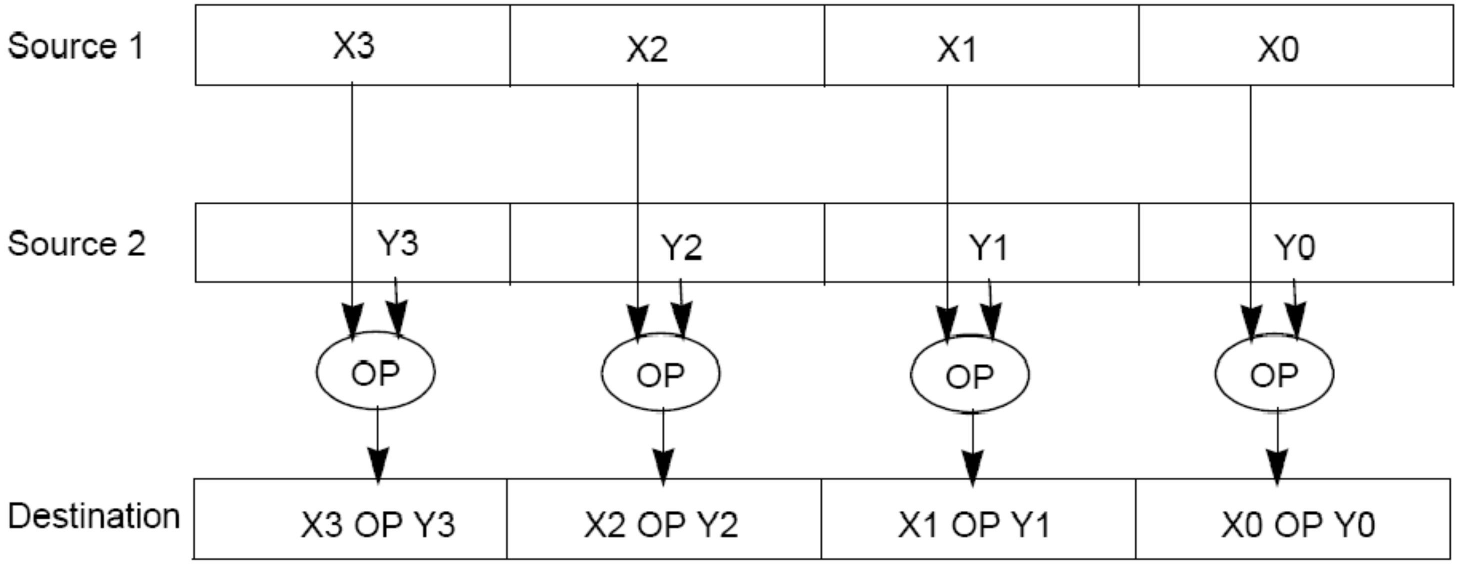
- Scalar processing
  - traditional mode
  - one operation **produces** one result



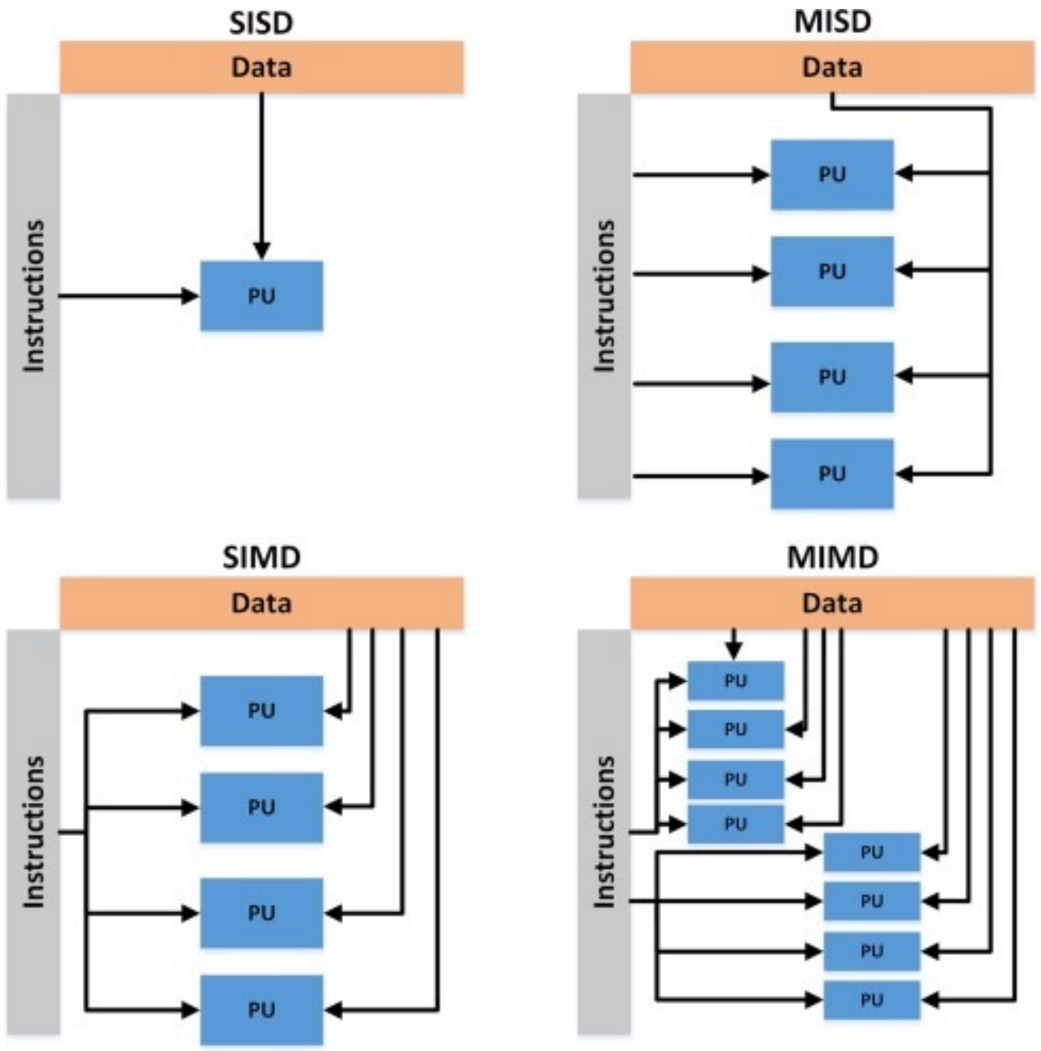
Slide Credit: Alex Klimovitski & Dean Macri, Intel Corporation

SIMD

- SIMD (single instruction multiple data) architecture performs the same operation on multiple data elements in parallel
- **`PADDW MM0, MM1`**



Flynn Taxonomy

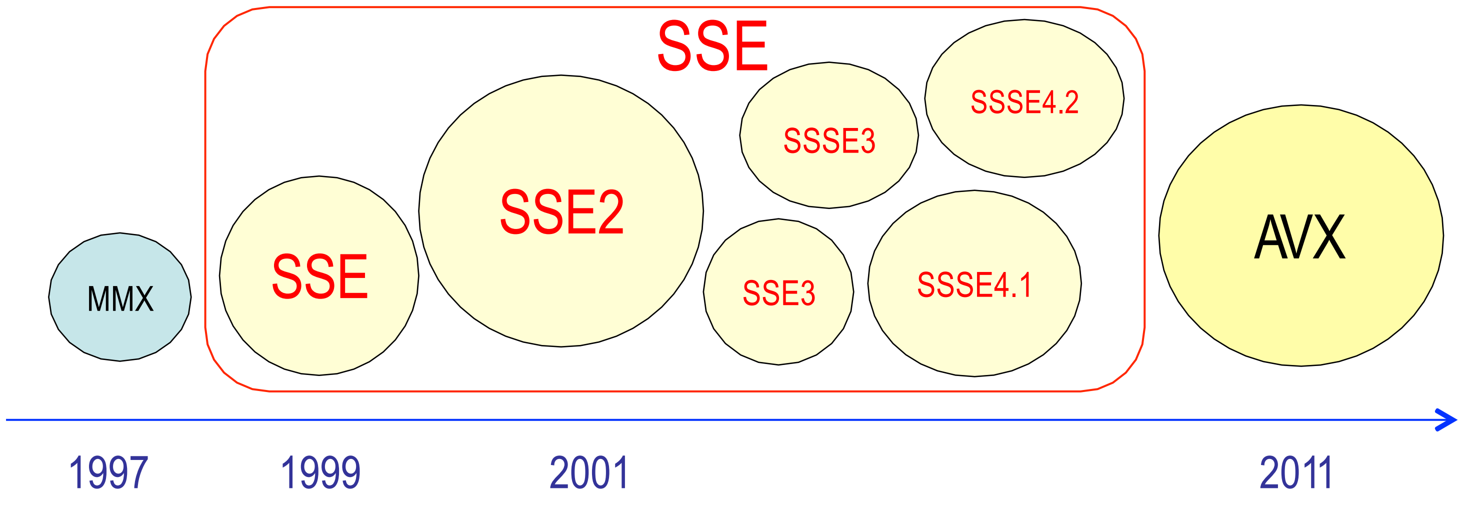


SIMD Architecture Support

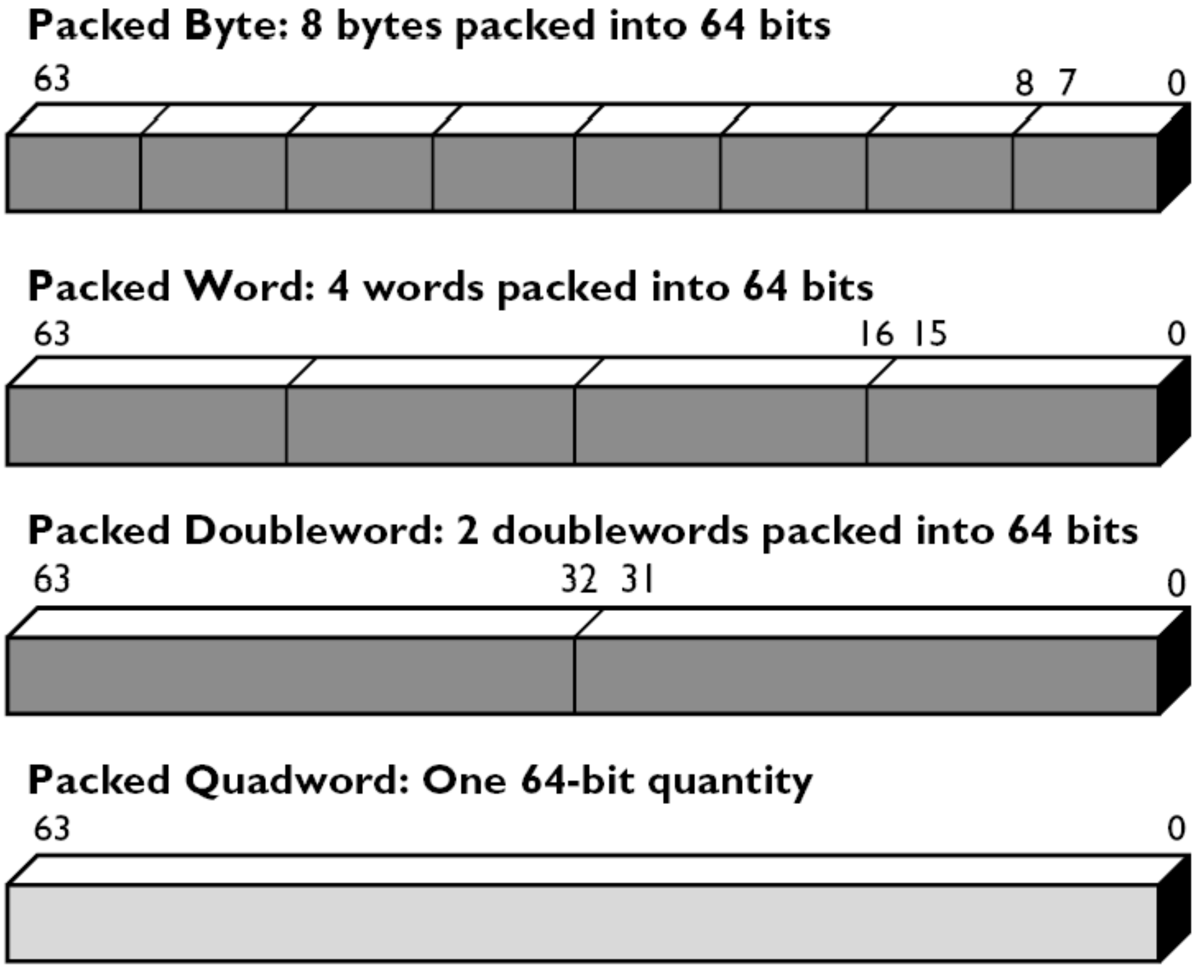
- **MMX** (Multimedia Extension) was introduced in 1996 - PI / PII.
    - 64-bit vectors / Only integer operations
- **SSE** (Streaming SIMD Extension) - Pentium III.
    - 128-bit vectors (only floating-point)
- **SSE2** - P4. (floating-point, double & Integer)
- **SSE3** - P4/HT
    - 13 more instructions.
- **AVX** (Advanced Vector Extensions) – 2011
    - 256-bit vectors
- SIMD extensions for other architectures:
    - 3DNow! / Altivec / VIS / VMX

The SSE family of Vector extensions

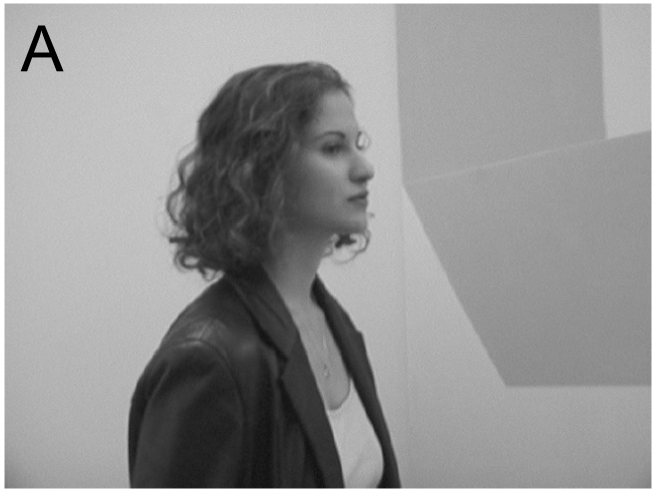
- Include over 400 instructions
- Available in most modern processors
- Uses 16 dedicated registers of length 128 bits
- Programming with AVX is similar, vector length and instruction names differ

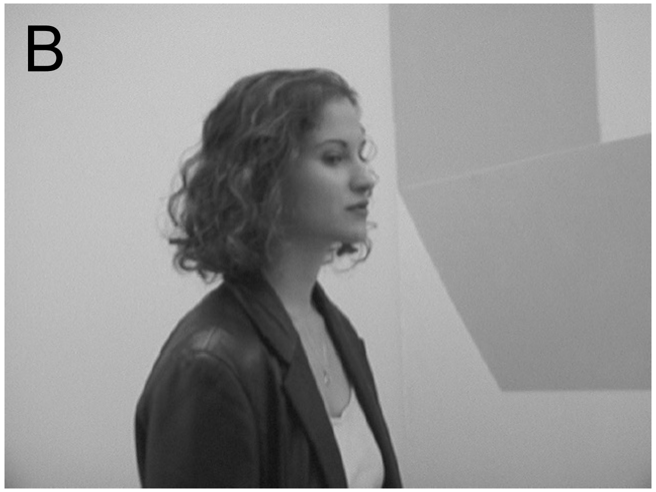


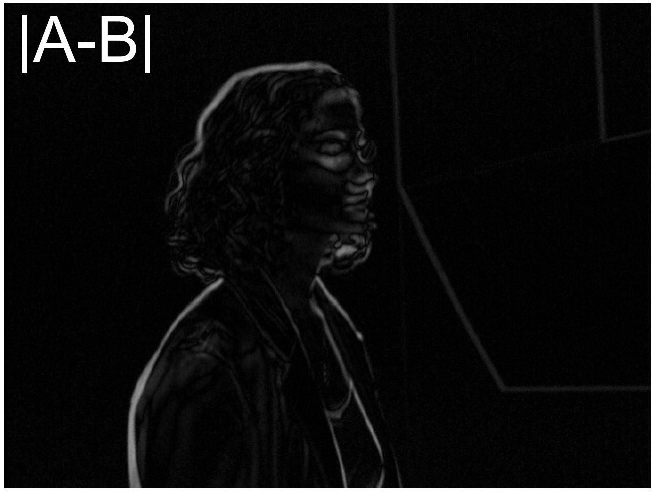
MMX data types


Packed Byte: 8 bytes packed into 64 bits
63
8 7
0
Packed Word: 4 words packed into 64 bits
63
16 15
0
Packed Doubleword: 2 doublewords packed into 64 bits
63
32 31
0
Packed Quadword: One 64-bit quantity
63
0

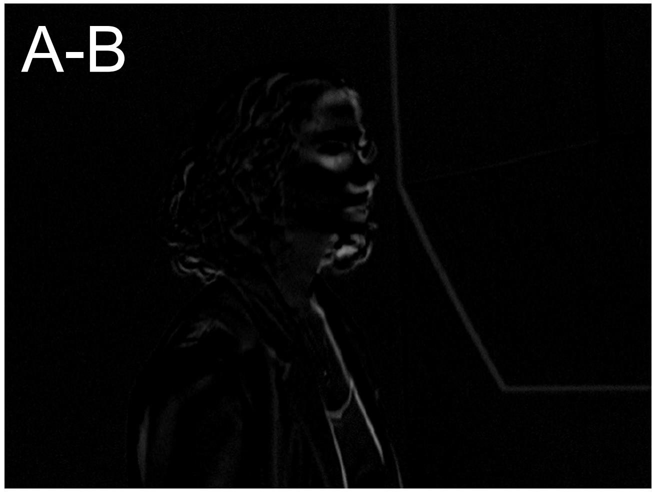
Application: frame difference

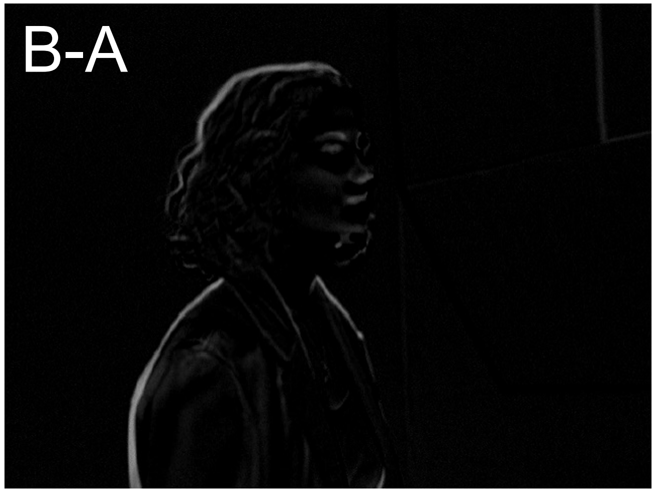




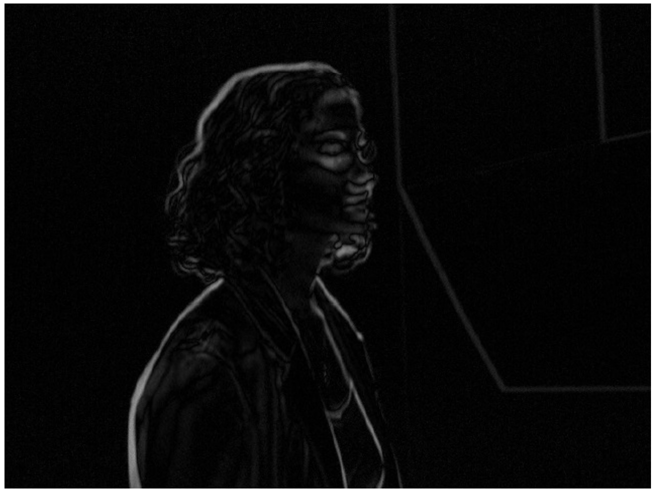


Application: frame difference

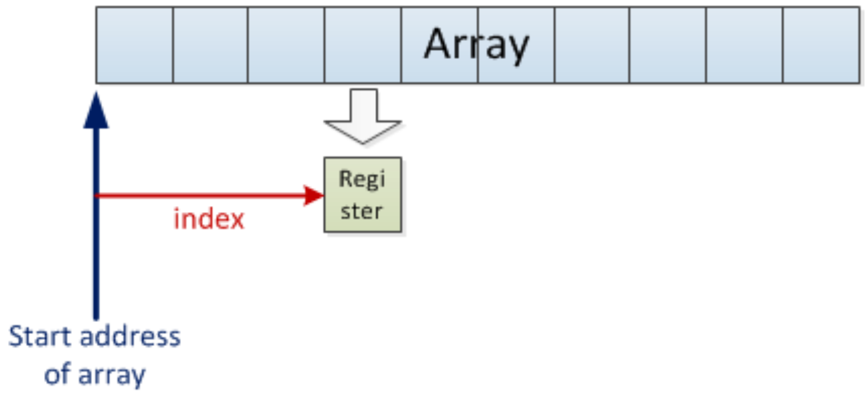




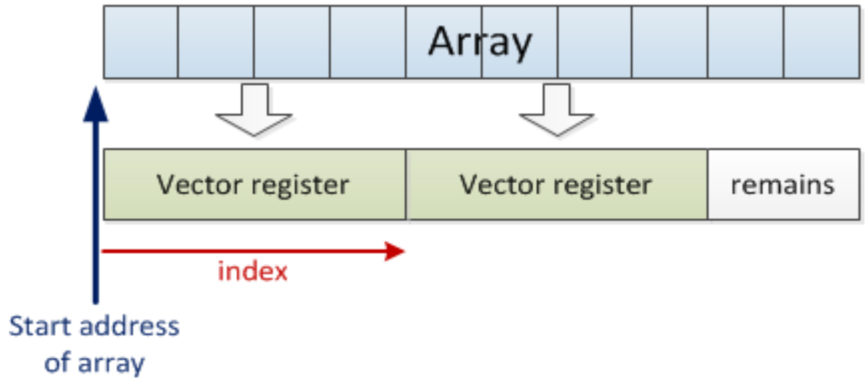
(A-B) or (B-A)



Loading from array to register



Scalar



Vector/SIMD

Application: frame difference

```
MOVQ      mm1, A //move 8 pixels of image A
MOVQ      mm2, B //move 8 pixels of image B
MOVQ      mm3, mm1 // mm3=A
PSUBSB    mm1, mm2 // mm1=A-B
PSUBSB    mm2, mm3 // mm2=B-A
POR       mm1, mm2 // mm1=|A-B|
```

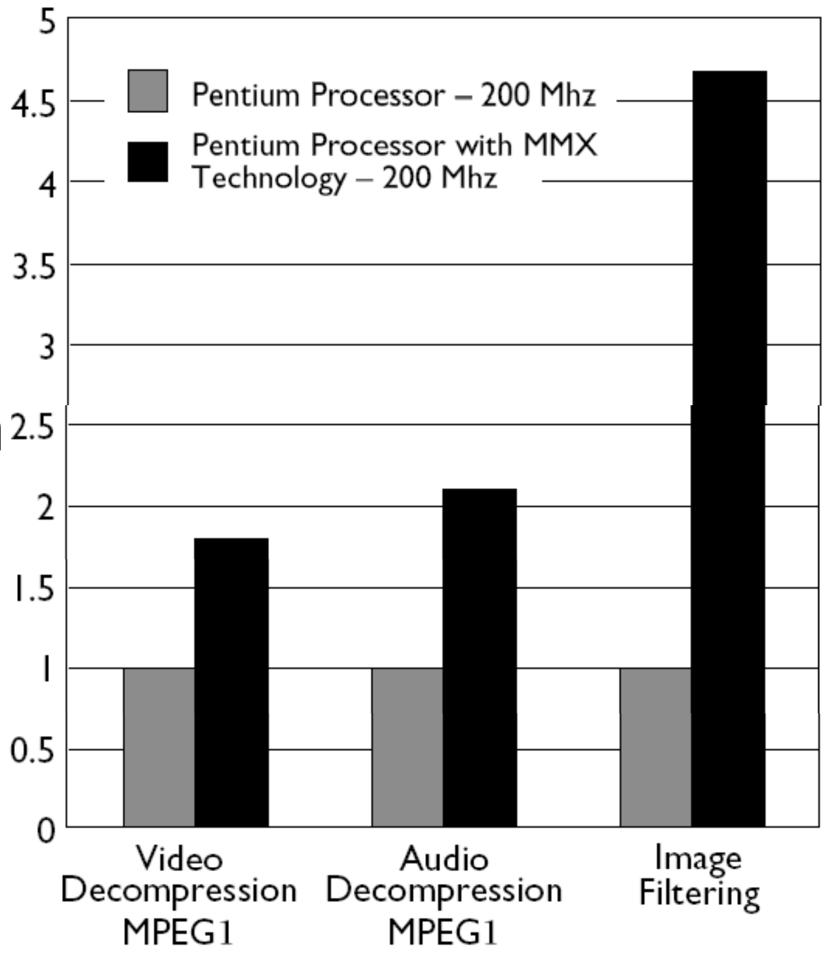
Performance boost (data from 1996)

Benchmark kernels: FFT, FIR, vector dot-product, IDCT, motion compensation.

65% performance gain

Lower the cost of multimedia programs

by removing the need of specialized DSP chips



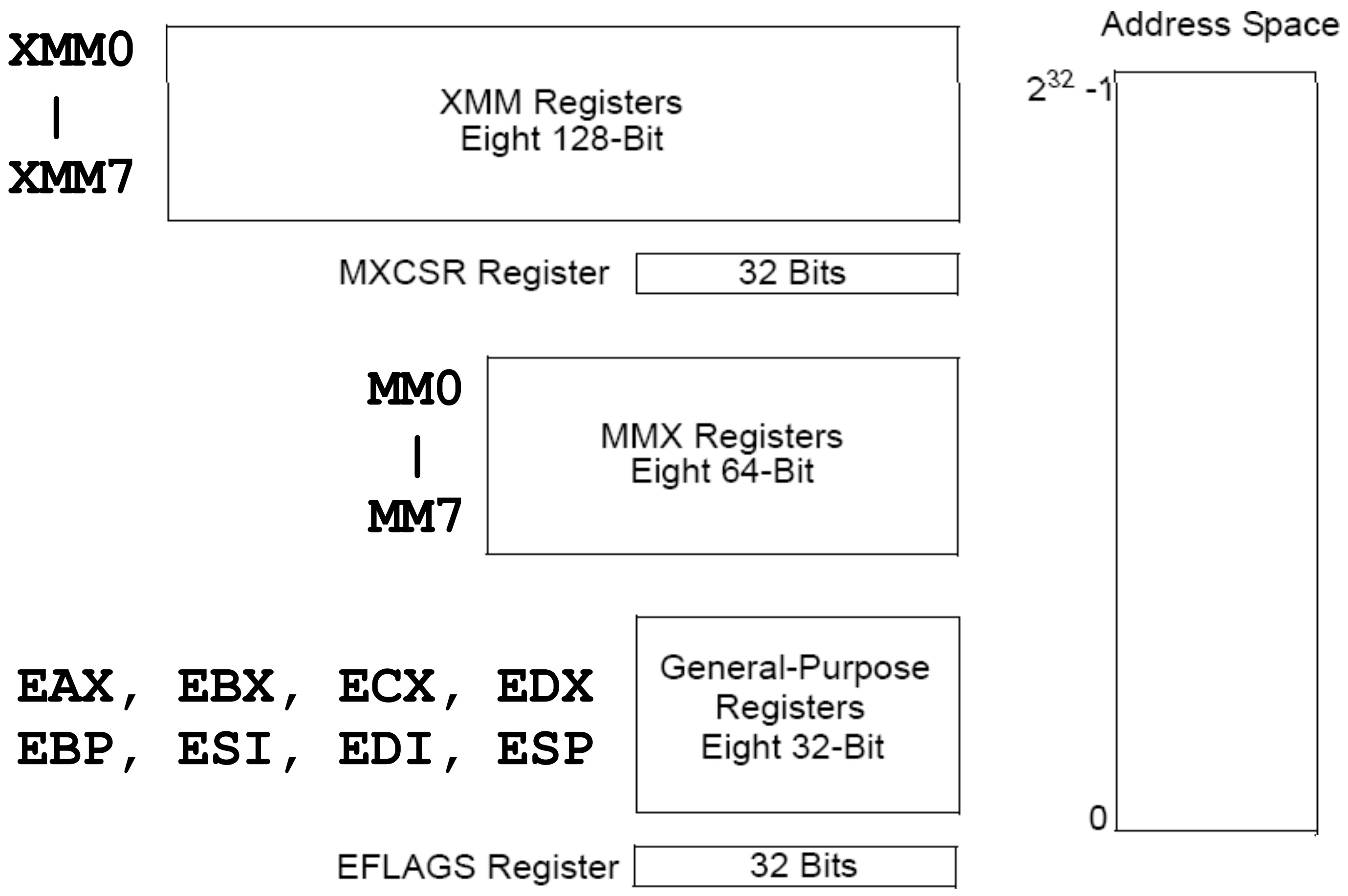
Issues with MMX

- Only supported integer operations
- MMX and FPU can not be used at the same time.
  - Used the same set of registers
  - Big overhead to switch.
- This proved to be a bad decision later.
- It is why Intel introduced SSE later as a separate unit.

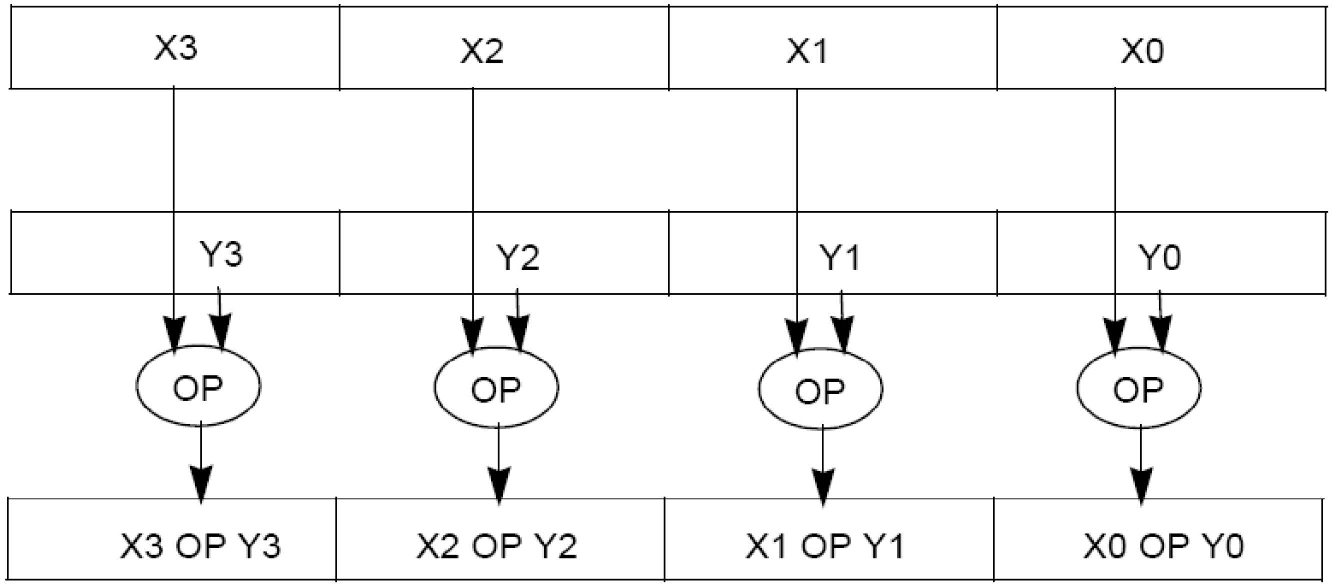
SSE

- Adds eight 128-bit registers
- Allows SIMD operations on packed single-precision floating-point numbers
    - Later added support for double-precision FP
- Most SSE instructions require 16-aligned addresses
- Instructions that explicitly prefetch data, control data cacheability and ordering of store

SSE programming environment

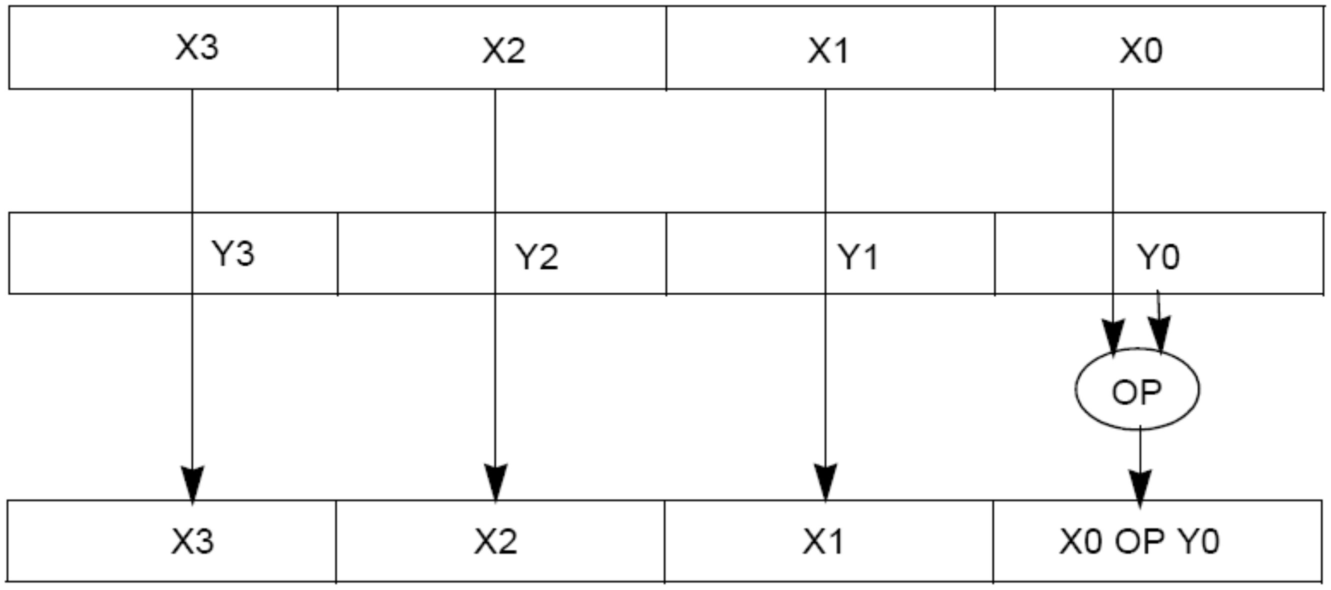


SSE packed FP operation



- **`ADDPS/SUBPS`**: packed single-precision FP

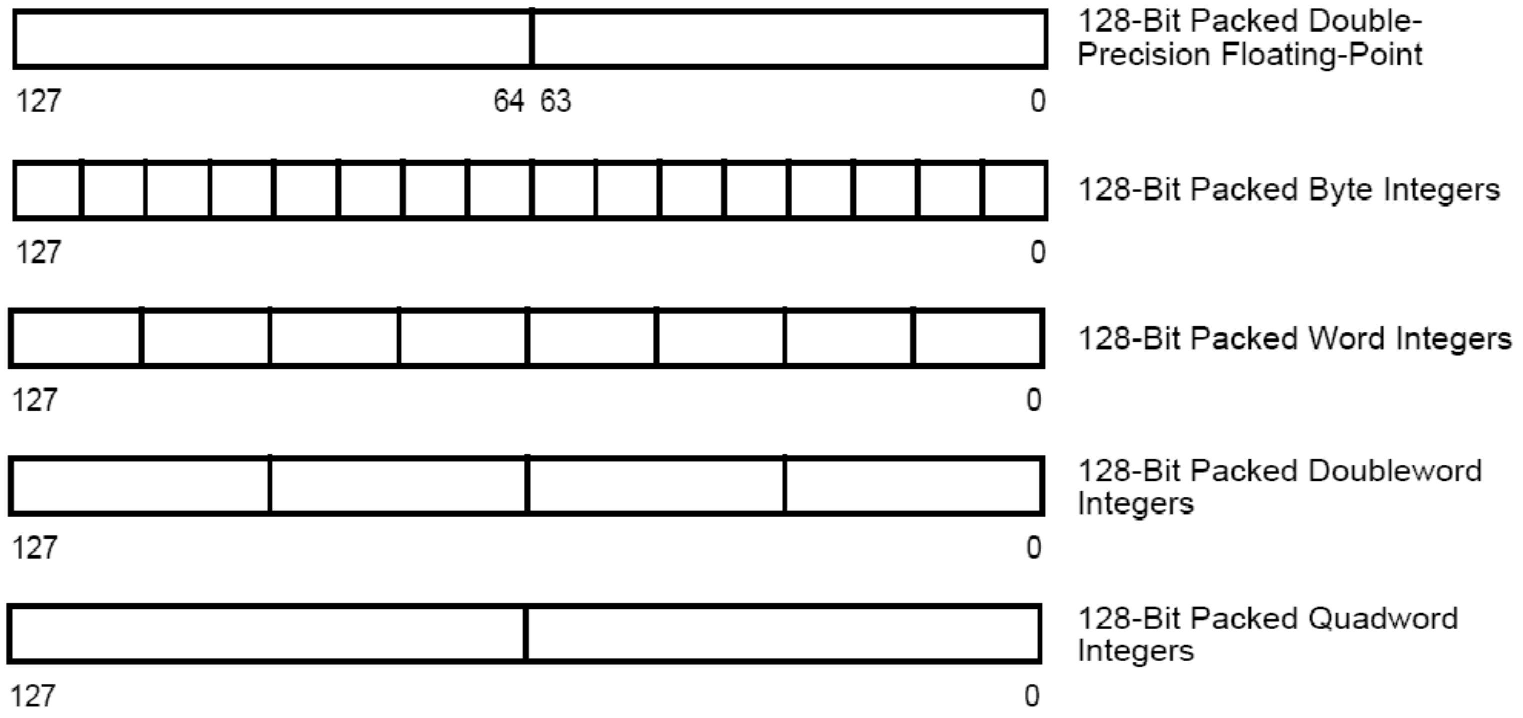
SSE scalar FP operation



- **`ADDSS/SUBSS`**: scalar single-precision FP

SSE features

- Add data types and instructions for them



128-Bit Packed Double-Precision Floating-Point
127
64 63
0
128-Bit Packed Byte Integers
127
0
128-Bit Packed Word Integers
127
0
128-Bit Packed Doubleword Integers
127
0
128-Bit Packed Quadword Integers
127
0

Programming environment unchanged

How to use assembly in projects

- Write the whole project in assembly
  - Difficult/rare
- Link with high-level languages
  - Develop most of the program in HLL
  - Use assembly for performance critical parts
- Inline assembly
  - Inline Assembly code directly into a HLL program.
  - Compilers such as Visual C++ have compiler-specific directives to identify inline ASM code.
- Intrinsics

Intrinsics

- A function known by the compiler that directly maps to a sequence of assembly instructions.

- The compiler manages :
  - Register names
  - Register allocations
  - Memory locations of data

- More efficient than called functions
  - No calling linkage is required.

_mm_<opcode>_<suffix>

ps: packed single-precision

ss: scalar single-precision

Intrinsics

```
#include <xmmintrin.h>

__m128 a , b , c;
c = _mm_add_ps( a , b );

float a[4] , b[4] , c[4];
for( int i = 0 ; i < 4 ; ++ i
    )  c[i] = a[i] + b[i];

// a = b * c + d / e;

__m128 a = _mm_add_ps( _mm_mul_ps( b , c ) ,
                       _mm_div_ps( d , e ) );
```

More Examples

VMX Example (PowerPC)

E.g. 1: Use in variable initialization statement (Vector literal is bolded)

```
 __vector signed int va = (__vector signed int) { -2, -1,  1,  2 };

   va = vec_add(va, ((__vector signed int) { 1, 2, 3, 4 }));
```

E.g. 2: Accessing vector as a scalar - the third element of a vector (Pointer cast is bolded)

```
__vector signed int va = (__vector signed int) { 1, 2, 3, 4 };

int *a = (int *) &va;

printf(“a[2] = %d”, a[2]);
```

E.g. 3:Accessing a scalar array as a vectors (Pointer cast is bolded)

```
int a[8] __attribute__((aligned(16))) = { 1, 2, 3, 4, 5, 6, 7, 8 };
__vector signed int *va = (__vector signed int *) a;

// va[0] = { 1, 2, 3, 4}, va[1] = { 5, 6, 7, 8 }
vb = vec_add(va[0], va[1]);
```

VMX Example (PowerPC)

```
#include <stdio.h>
#include <altivec.h>

 // declares input/output scalar varialbes

int a[4] __attribute__((aligned(16))) = { 1, 3, 5, 7 };
int b[4] __attribute__((aligned(16))) = { 2, 4, 6, 8 };
int c[4] __attribute__((aligned(16)));

int main(int argc, char **argv)
{
     // declares vector variables which points to scalar arrays

    __vector signed int *va = (__vector signed int *) a;
    __vector signed int *vb = (__vector signed int *) b;
    __vector signed int *vc = (__vector signed int *) c;

     // adds four signed integers at once

    *vc = vec_add(*va, *vb);    // 1 + 2, 3 + 4, 5 + 6, 7 + 8

     // output results

    printf("c[0]=%d, c[1]=%d, c[2]=%d, c[3]=%d\n", c[0], c[1], c[2], c[3]);

    return 0;
}
```

SSE Example (intel)

```
float Scalar(float *s1, float *s2) {
    int i;
    float prod;

    for(i=0; i<size; i++) {
        prod += s1[i] * s2[i];
    }
    return prod;
}
```

```
float ScalarSSE(float *s1, float *s2) {

    float prod;
    int i;
    __m128 X, Y, Z;

    for(i=0; i<size; i+=4) {
        X = _mm_load_ps(&s1[i]);
        Y = _mm_load_ps(&s2[i]);
        X = _mm_mul_ps(X, Y);
        Z = _mm_add_ps(X, Z);
    }

    for(i=0; i<4; i++) {
        prod += Z[i];
    }
    return prod;
}
```

Z[0] = s1[0] * s2[0] + s1[4] * s2[4] + s1[8] * s2[8]+…

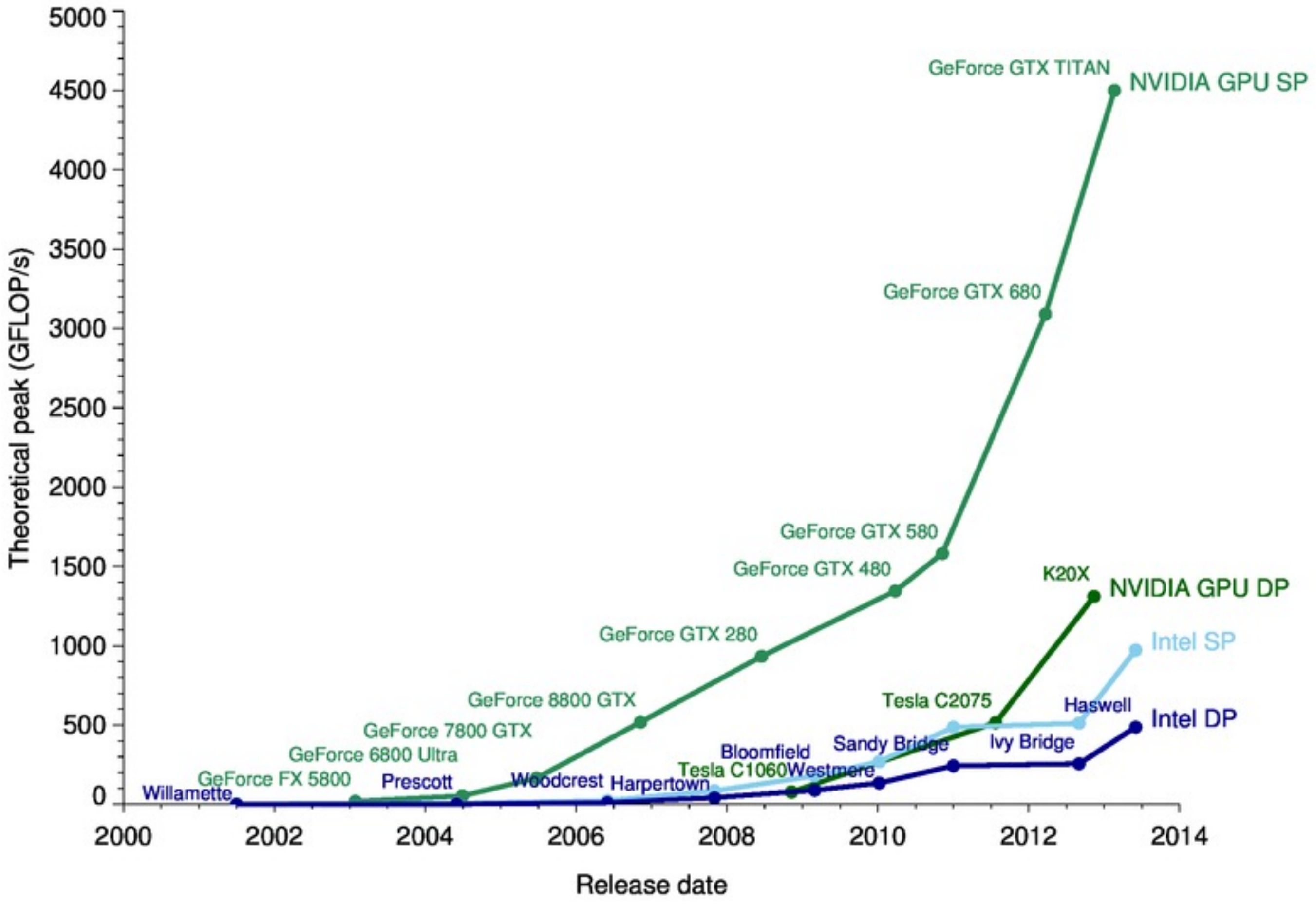
Z[1] = s1[1] * s2[1] + s1[5] * s2[5] + s1[9] * s2[9]+…

Z[2] = s1[2] * s2[2] + s1[6] * s2[6] + s1[10]*s2[10]+…

Z[3] = s1[3] * s2[3] + s1[7] * s2[7] + s1[11]*s2[11]+…

Other SIMD architectures

- Graphics Processing Unit (GPU)



References

- *Intel MMX for Multimedia PCs*, CACM, Jan. 1997
- Chapter 11 *The MMX Instruction Set*, The Art of Assembly
- Chap. 9, 10, 11 of IA-32 Intel Architecture Software Developer’s Manual: Volume 1: Basic Architecture
- http://www.csie.ntu.edu.tw/~r89004/hive/sse/page_1.html