AI Infrastructure

Build scalable AI services through model, system software, and hardware co-design

Li Shang
lishang@fudan.edu.cn

Acknowledgement

Machine learning systems is a broad and rapidly evolving field. The course material has been developed using a broad spectrum of resources, including research papers, lecture slides, blog posts, research talks, tutorial videos, and other materials shared by the research community.

Part of the course material was created by LLM itself.

More importantly, for each and every of us, LLM shall be heavily involved in daily learning.

AI Infra Architect

Computer system development is workload driven. Scalability is the first principle of large models, therefore enabling scalable AI services the foremost responsibility of AI infrastructure.

AI infrastructure, in particular the silicon, takes 2-3 years to build, large models evolve in months. Therefore, predicting the future workload is the ultimate challenge of AI infrastructure development.

“the best way to predict the future is to invent it – Alan Kay”: Identify and then overcome the scalability roadblocks through model-software-hardware co-design.

AI Infra architect wears multiple hats: mathematician, software developer, silicon architect & engineer; and of course, agentic AI is our best teammate.

Today’s lecture covers GPU, or GPGPU. What we want to understand is how the workload evolution has been driving the transition from CPU to GPU, as well as the evolution of GPU architecture.

CPU Architecture for Instructure Level Parallelism

Workload is the king, as always

CPU: Instructure level parallelism

How to speed up the code

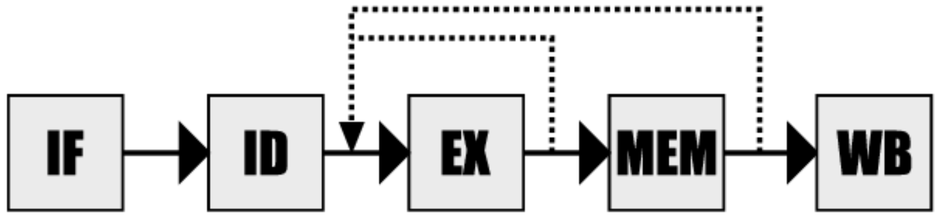
```
int sum_array(int *A, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += A[i];
    }
    return sum;
}
```

CPU: Instructure level parallelism

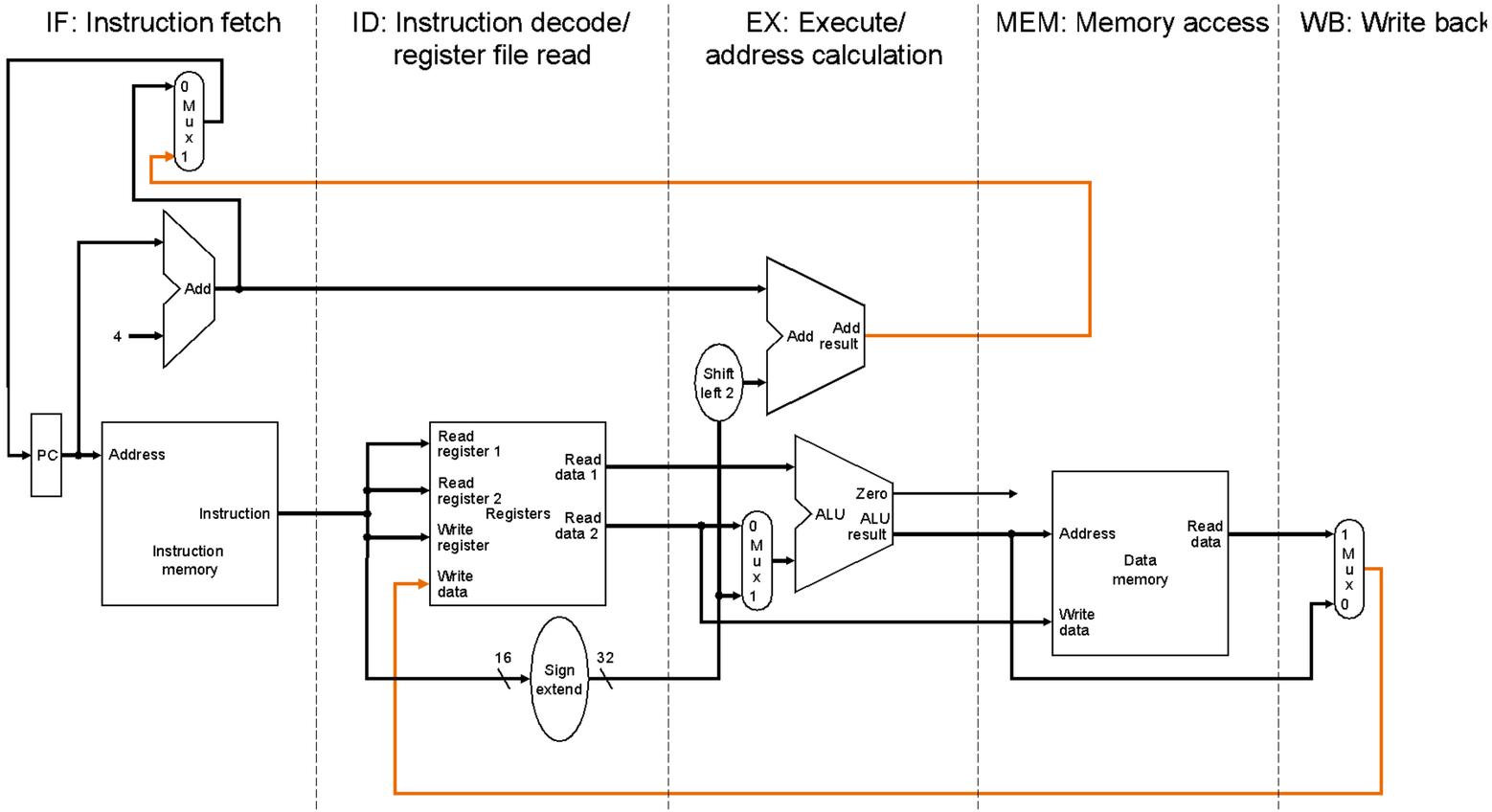
```c
int sum_array(int *A, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += A[i];
    }
    return sum;
}
```

```
L1: cmp   i, n          ; compare i with n
    jge   L2            ; if i >= n, exit loop
    load  r1, [A+i*4]   ; load A[i] from memory
    add   sum, r1       ; sum += A[i]
    inc   i             ; i++
    jmp   L1            ; repeat
L2: ret                 ; return sum
```

How does a CPU look like



IF: Instruction fetch
ID: Instruction decode and register read
EX: Execute, address generation
MEM: Memory access
WB: Write back to registers



Pipelining

1. **IF**: Instruction Fetch
2. **ID**: Decode & register fetch
3. **EX**: Execute (ALU op, branch calc)
4. **MEM**: Memory access
5. **WB**: Write-back (to register file)

What is the catch?

```
L1: cmp   i, n          ; compare i with n
    jge   L2            ; if i >= n, exit loop
    load  r1, [A+i*4]   ; load A[i] from memory
    add   sum, r1       ; sum += A[i]
    inc   i             ; i++
    jmp   L1            ; repeat
L2: ret                 ; return sum
```

Cycle	IF	ID	EX	MEM	WB
1	cmp(k)				
2	jge(k)	cmp(k)			
3	load(k)	jge(k)	cmp(k)		
4	add(k)	load(k)	jge(k)	cmp(k)	
5	inc(k)	add(k)	load(k)	jge(k)	cmp(k)
6	jmp(k)	inc(k)	add(k)	load(k)	jge(k)
7	cmp(k+1)	jmp(k)	inc(k)	add(k)	load(k)
8	jge(k+1)	cmp(k+1)	jmp(k)	inc(k)	add(k)
9	load(k+1)	jge(k+1)	cmp(k+1)	jmp(k)	inc(k)
10	add(k+1)	load(k+1)	jge(k+1)	cmp(k+1)	jmp(k)
11	inc(k+1)	add(k+1)	load(k+1)	jge(k+1)	cmp(k+1)
12	jmp(k+1)	inc(k+1)	add(k+1)	load(k+1)	jge(k+1)

Branch Prediction

Avoid pipeline stalls

```
L1: cmp   i, n
    jge   L2
    load  r1, [A+i*4]
    add   sum, r1
    inc   i
    jmp   L1
L2: ret
```

- The `jge L2` is a **branch**.
- If the CPU waits every time to know whether to continue, the pipeline stalls.
- Modern CPUs **predict**:
  - "Most of the time the loop continues" → fetches next iteration's instructions ahead of time.
- If prediction is correct (99% of iterations), the pipeline stays full.
- If wrong (last iteration), CPU discards speculative work, small penalty.

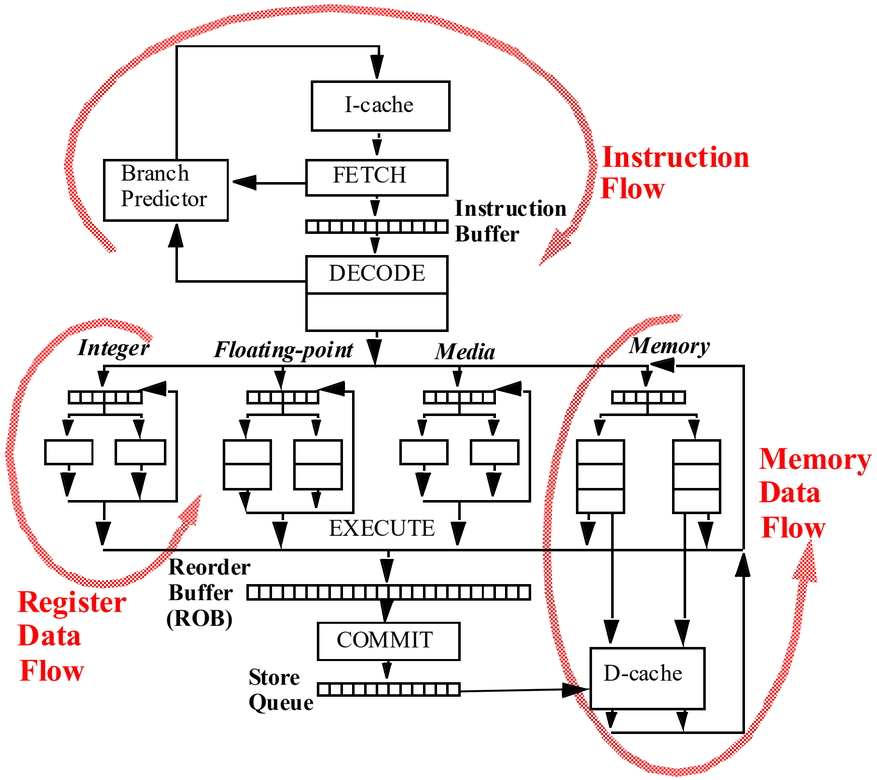
Out-of-Order Execution

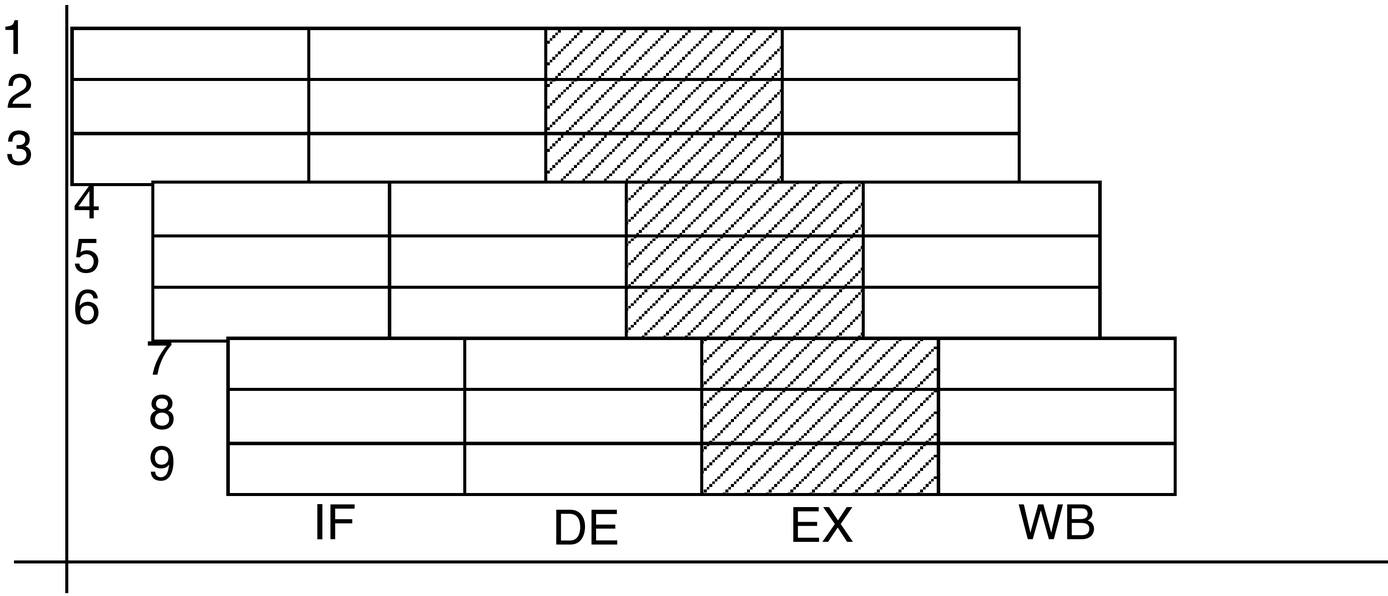
Hide memory latency

```
L1: cmp   i, n
    jge   L2
    load  r1, [A+i*4]
    add   sum, r1
    inc   i
    jmp   L1
L2: ret
```

- In this sequence, the **load from memory** (`load r1, [A+i*4]`) is slow compared to the integer add.
- A modern CPU doesn't just sit idle — it looks ahead:
  - While waiting for `A[i]` to arrive, it issues the `load [A+(i+1)*4]` early.
  - It may also speculatively start decoding the next `cmp` and `add`.
- **Result:** instructions overlap, latency is hidden, and the loop runs much faster.

Superscalar





Loop Unrolling

```c
int sum_array(int *A, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += A[i];
    }
    return sum;
}
```

```c
int sum_array_unroll4(const int *A, int n) {
    int s0 = 0, s1 = 0, s2 = 0, s3 = 0;
    int i = 0;
    int n4 = n & ~3;                    // floor to multiple of 4
    for (; i < n4; i += 4) {
        s0 += A[i+0];
        s1 += A[i+1];
        s2 += A[i+2];
        s3 += A[i+3];
    }
    int sum = (s0 + s1) + (s2 + s3); // combine partials
    for (; i < n; ++i) sum += A[i];  // tail (remainder) loop
    return sum;
}
```

Loop Unrolling

```c
int sum_array_unroll4(const int *A, int n) {
    int s0 = 0, s1 = 0, s2 = 0, s3 = 0;
    int i = 0;
    int n4 = n & ~3;                  // floor to multiple of 4
    for (; i < n4; i += 4) {
        s0 += A[i+0];
        s1 += A[i+1];
        s2 += A[i+2];
        s3 += A[i+3];
    }
    int sum = (s0 + s1) + (s2 + s3); // combine partials
    for (; i < n; ++i) sum += A[i];  // tail (remainder) loop
    return sum;
}
```

```asm
; Assume rA = base of A, ri = i
; s0..s3 kept in registers (e.g., r4..r7)

L1:
    load  r10, [rA + ri*4 +  0]   ; A[i+0]
    load  r11, [rA + ri*4 +  4]   ; A[i+1]
    load  r12, [rA + ri*4 +  8]   ; A[i+2]
    load  r13, [rA + ri*4 + 12]   ; A[i+3]

    add   r4, r4, r10             ; s0 += A[i+0]
    add   r5, r5, r11             ; s1 += A[i+1]
    add   r6, r6, r12             ; s2 += A[i+2]
    add   r7, r7, r13             ; s3 += A[i+3]

    add   ri, ri, 4               ; i += 4
    cmp   ri, n4
    jl    L1

; combine: r4=r4+r5, r6=r6+r7, then r4=r4+r6
```

- **Fewer branches:** 1 branch per 4 elements (vs 1 per element).
- **More ILP:** four independent `load`+`add` pairs. The CPU can issue loads early and overlap them; adds don't wait on a single running dependency (since s0..s3 are independent).
- **Better use of caches/prefetchers:** sequential loads hint the hardware prefetcher; each miss can be overlapped with others.

Vectorization

Vectorization improves parallelism by using **SIMD registers** to perform multiple loads and additions in one instruction. In the summation loop, AVX2 can sum 8 integers at once, cutting loop iterations by 8× and reducing branch/control overhead, while also aligning with memory prefetching for high throughput.

SIMD: Single instruction multiple data

```c
#include <immintrin.h>   // AVX intrinsics

int sum_array_vectorized(const int *A, int n) {
    __m256i acc = _mm256_setzero_si256();  // vector accumulator
    int i = 0;

    // Process 8 ints per loop
    for (; i <= n-8; i += 8) {
        __m256i v = _mm256_loadu_si256((__m256i*)&A[i]); // load 8 ints
        acc = _mm256_add_epi32(acc, v);                  // acc += v
    }

    // Horizontal add (sum across vector lanes)
    int tmp[8];
    _mm256_storeu_si256((__m256i*)tmp, acc);
    int sum = tmp[0]+tmp[1]+tmp[2]+tmp[3]+tmp[4]+tmp[5]+tmp[6]+tmp[7];

    // Handle tail elements
    for (; i < n; i++) sum += A[i];

    return sum;
}
```

Summary

	Processing Side
Compiler (Software)	Loop unrolling, vectorization → expose parallel ops
CPU (Hardware)	Out-of-order execution Multiple-issue (superscalar) Branch prediction

Data Locality

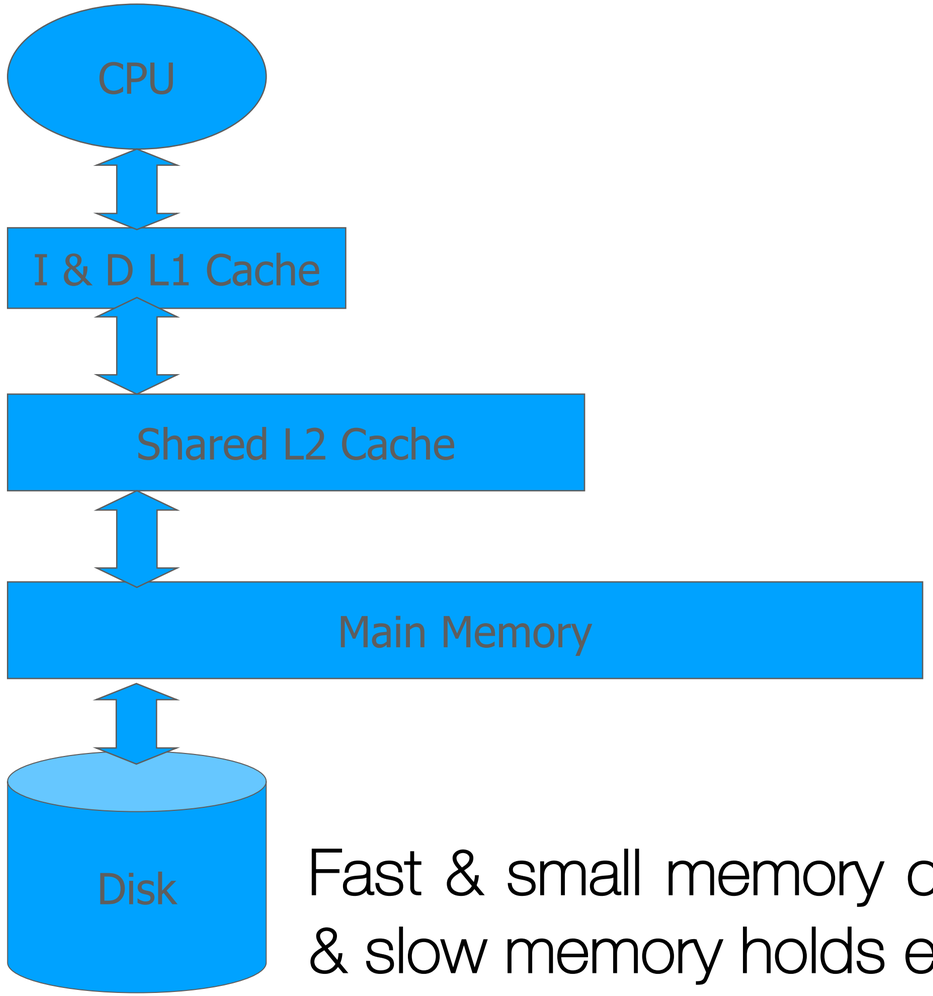
```c
int sum_array(int *A, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += A[i];
    }
    return sum;
}
```

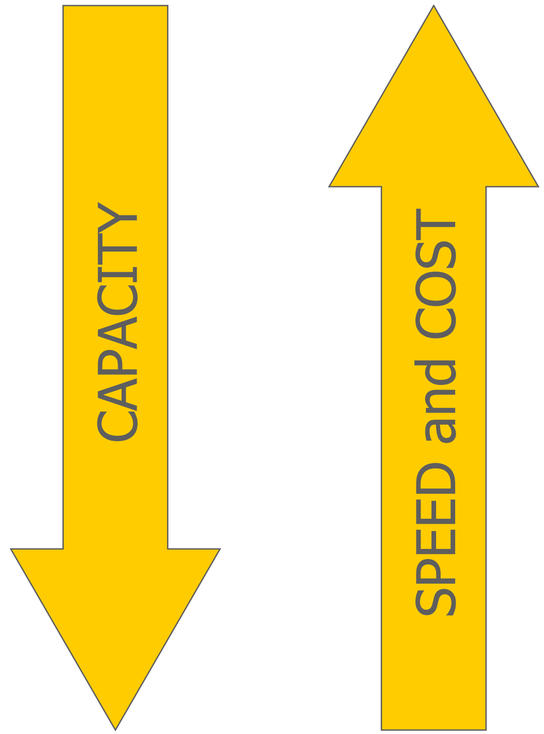
Locality type	Meaning	Example in code
Temporal	Reuse same data soon	`sum += A[i]` → `sum` reused each iteration
Spatial	Reuse nearby data soon	`A[i]`, then `A[i+1]` in a loop

Memory Hierarchy: Why does it work?

Temporal Locality
- Keep recently referenced items at higher levels
- Future references satisfied quickly

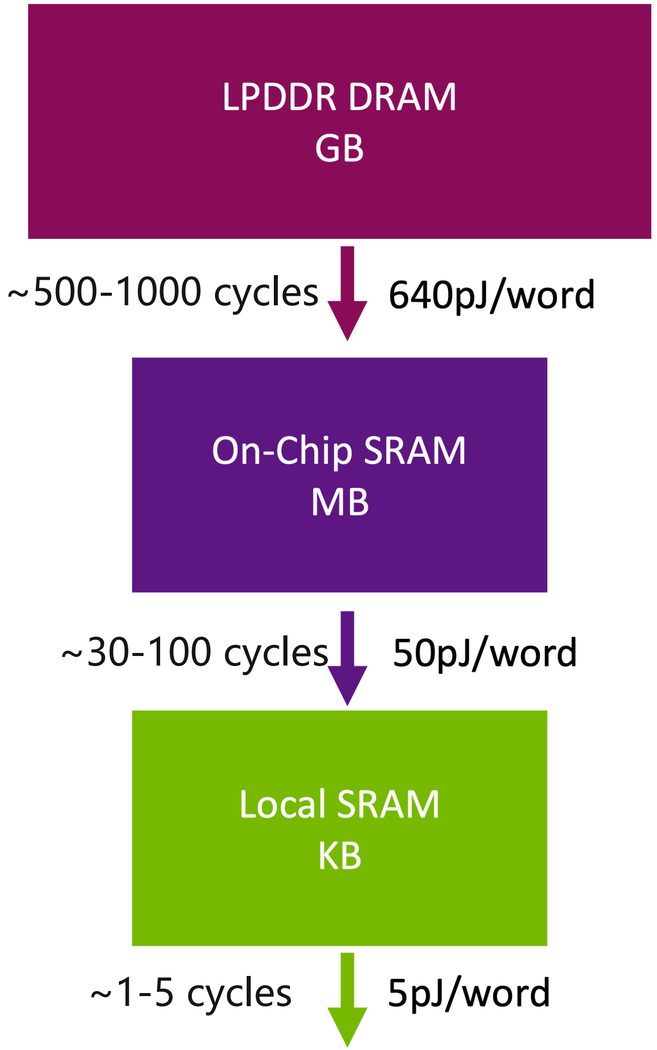
Spatial Locality
- Bring neighbors of recently referenced to higher levels
- Future references satisfied quickly



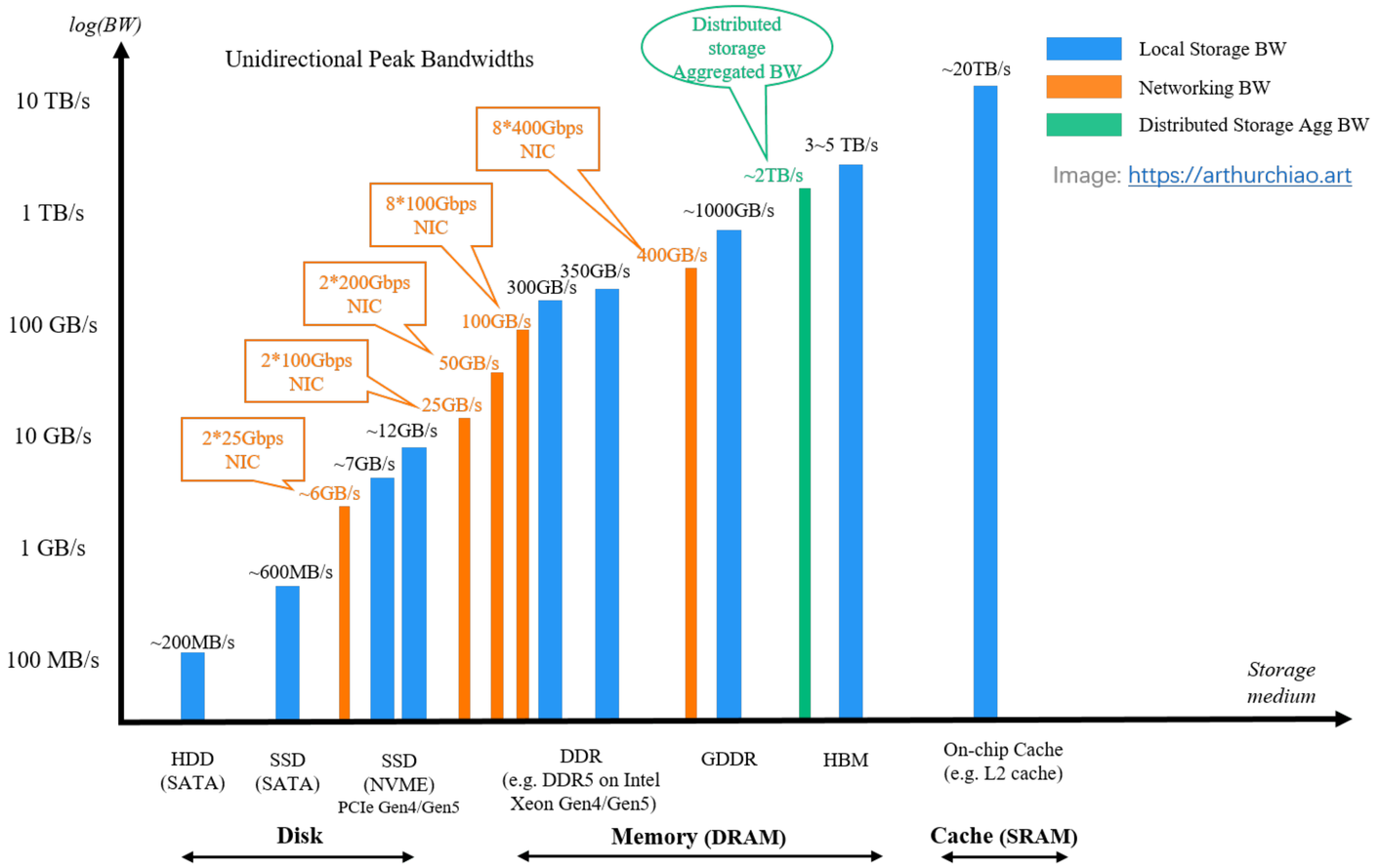


Fast & small memory caches frequent data, large & slow memory holds everything.

The Importance of Staying Local

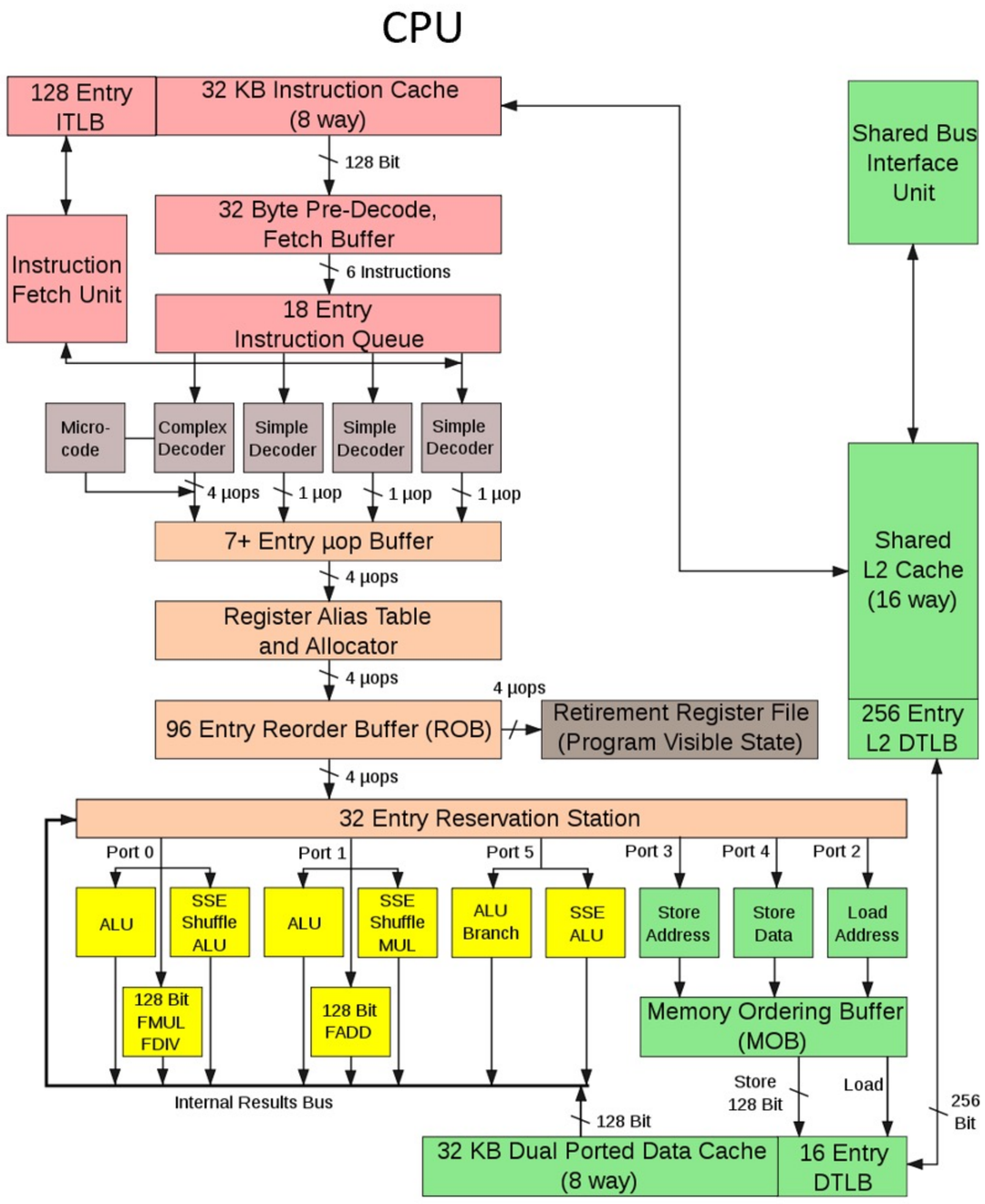


Moving data can cost much more than computing



Summary

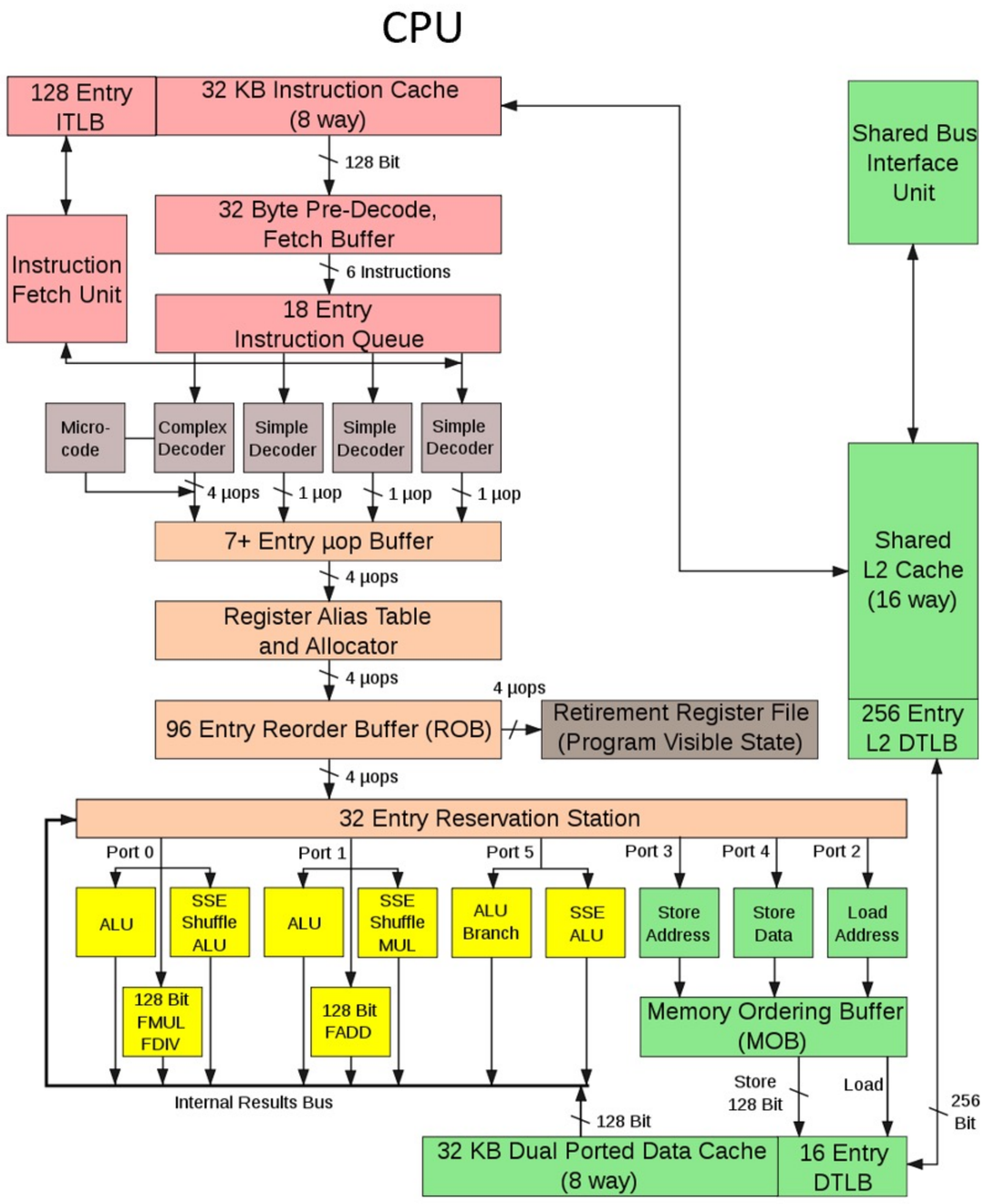
	Processing Side	Data Side
Compiler (Software)	Loop unrolling, vectorization → expose parallel ops	Ensure contiguous arrays, unroll for cache-line use, software prefetch hints
CPU (Hardware)	Out-of-order execution Multiple-issue (superscalar) Branch prediction	Cache hierarchy (L1/L2/L3), hardware prefetchers, spatial + temporal locality



Intel Core 2 Architecture

⚙️ Overall Pipeline Flow Summary

1. **Fetch**: Instructions are fetched from instruction cache (I-Cache) with the help of ITLB.
2. **Pre-decode**: Instructions are pre-decoded and buffered for the decoding stage.
3. **Decode**: Instructions are translated into µops by simple or complex decoders or micro-code ROM.
4. **Rename & Allocate**: Registers are renamed (RAT), µops are allocated into the reorder buffer (ROB).
5. **Issue & Execute**: µops wait in reservation stations until operands are ready, then dispatched to execution units.
6. **Memory Access**: Load/store instructions handled by MOB, data cache, and DTLB.
7. **Write-back & Commit**: Execution results are written back via internal result buses, and instructions commit in order via ROB, updating the Retirement Register File.

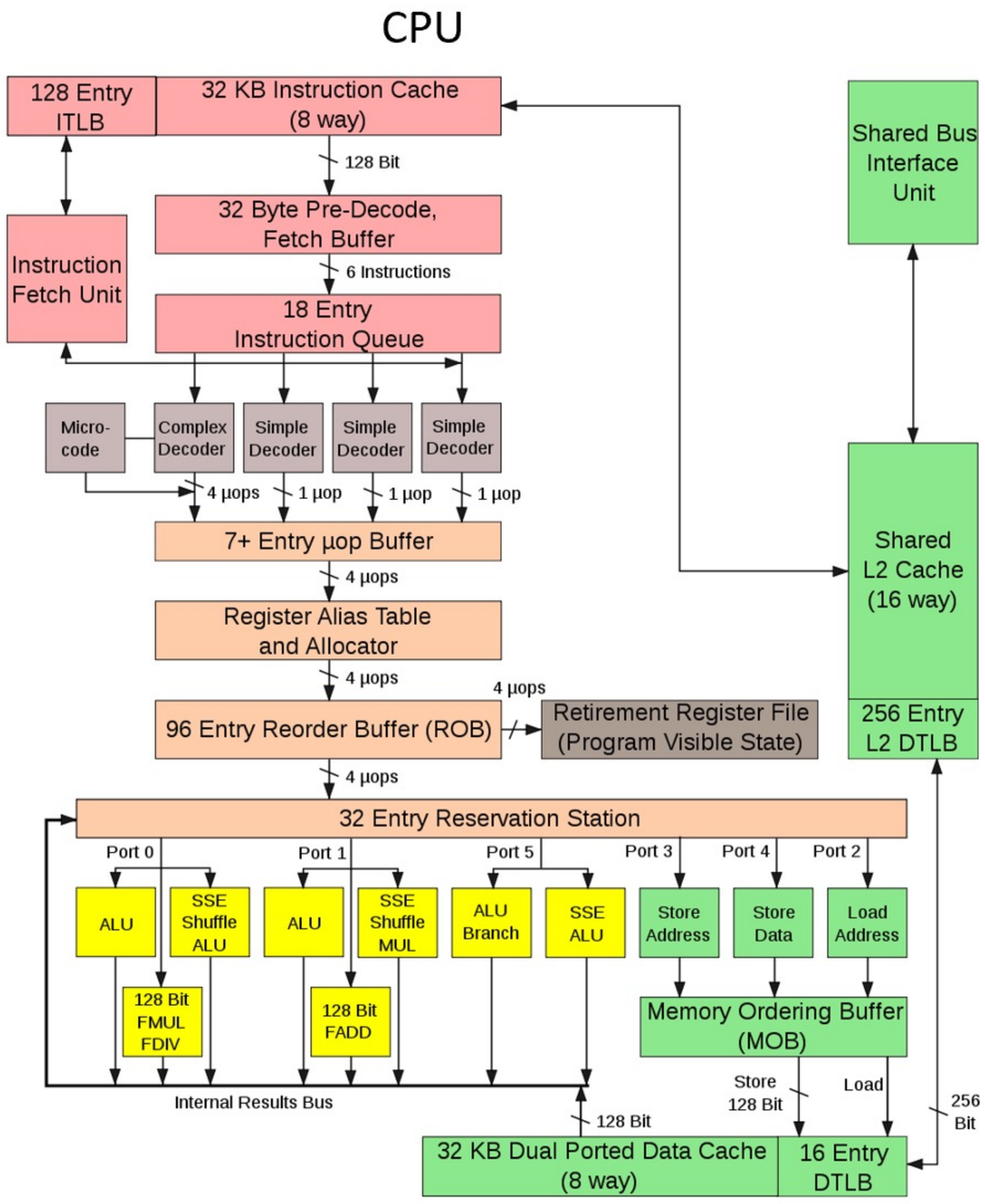


Intel Core 2 Architecture

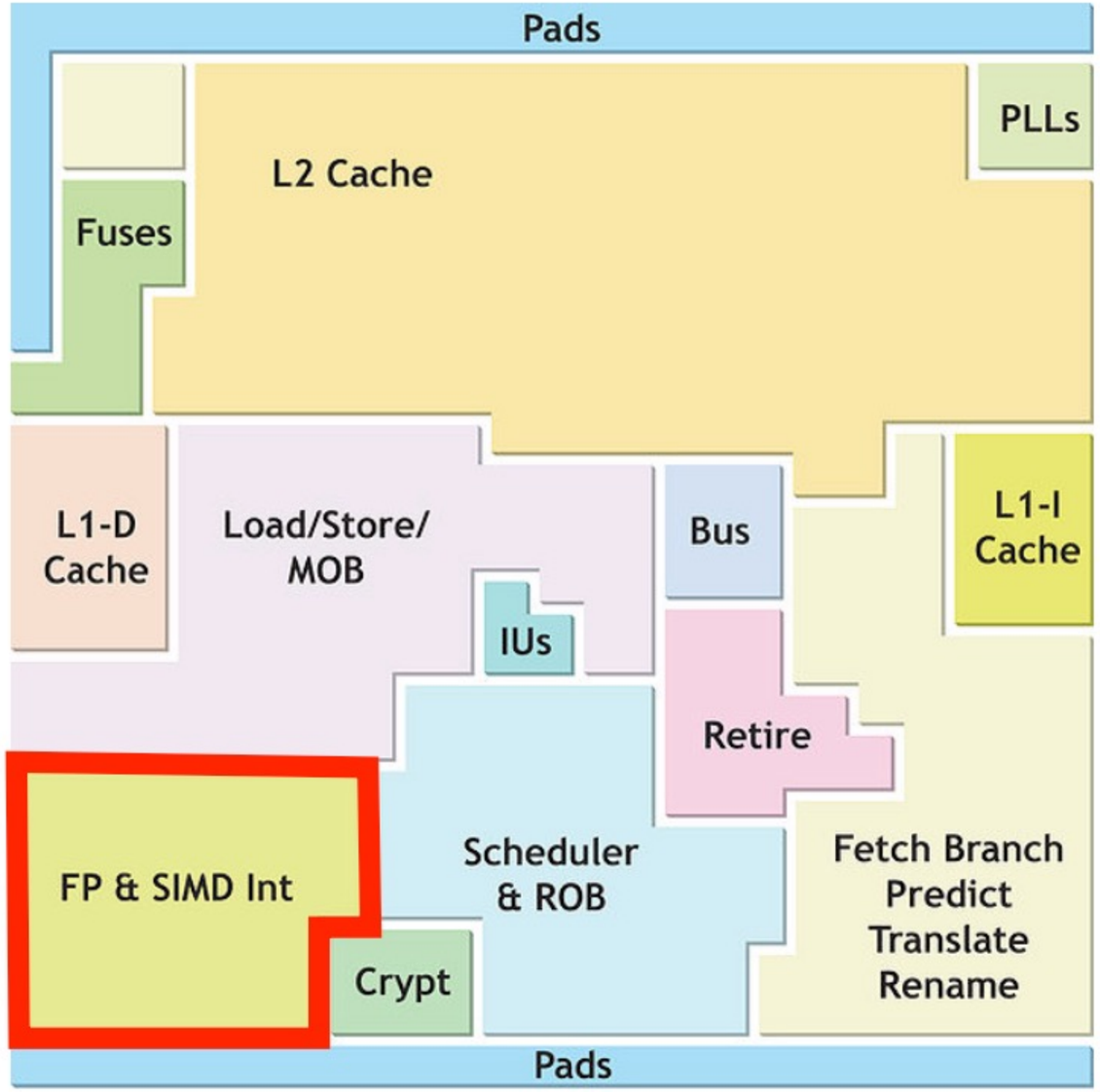
📌 Key Design Features

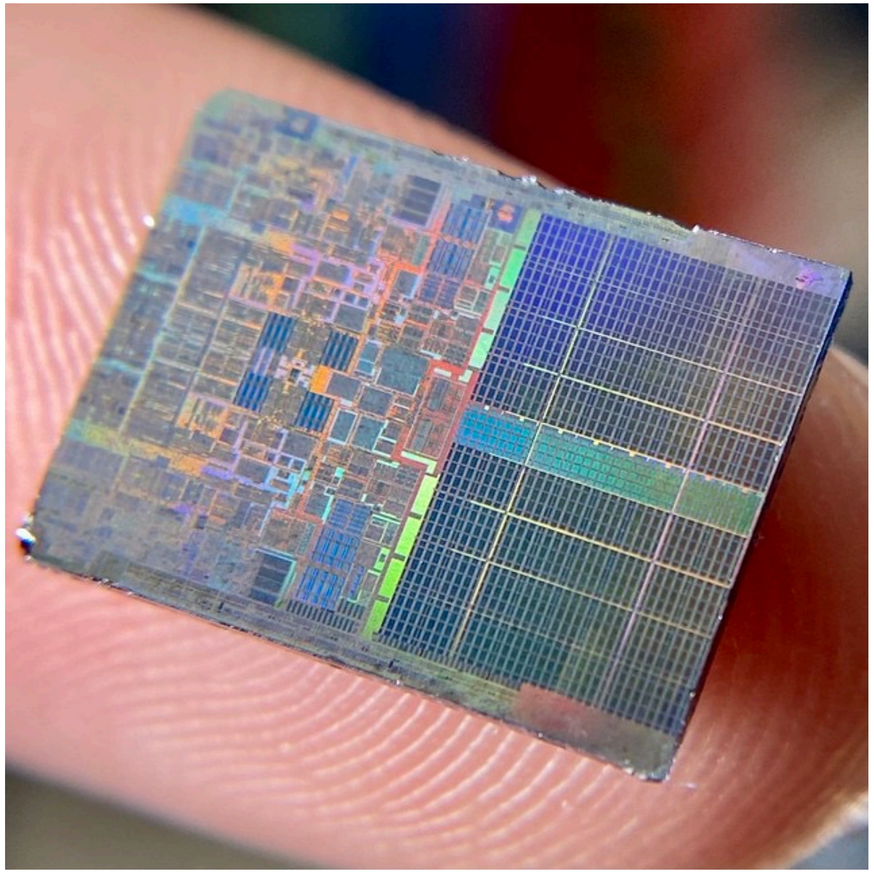
- **Out-of-order execution**: Instructions can be executed out-of-order but commit results strictly in program order.
- **Register renaming**: Prevents pipeline stalls caused by register conflicts.
- **Multiple execution units and ports**: Enables parallel execution of multiple instructions per cycle.
- **Separate instruction and data caches**: Improves cache locality and reduces latency.
- **Multi-level caching (L1, L2)**: Balances latency, throughput, and hit-rate.
- **Translation Lookaside Buffers (ITLB, DTLB)**: Accelerates virtual-to-physical address translations.
- **Sophisticated branch prediction (implied)**: Ensures high instruction throughput and pipeline efficiency.

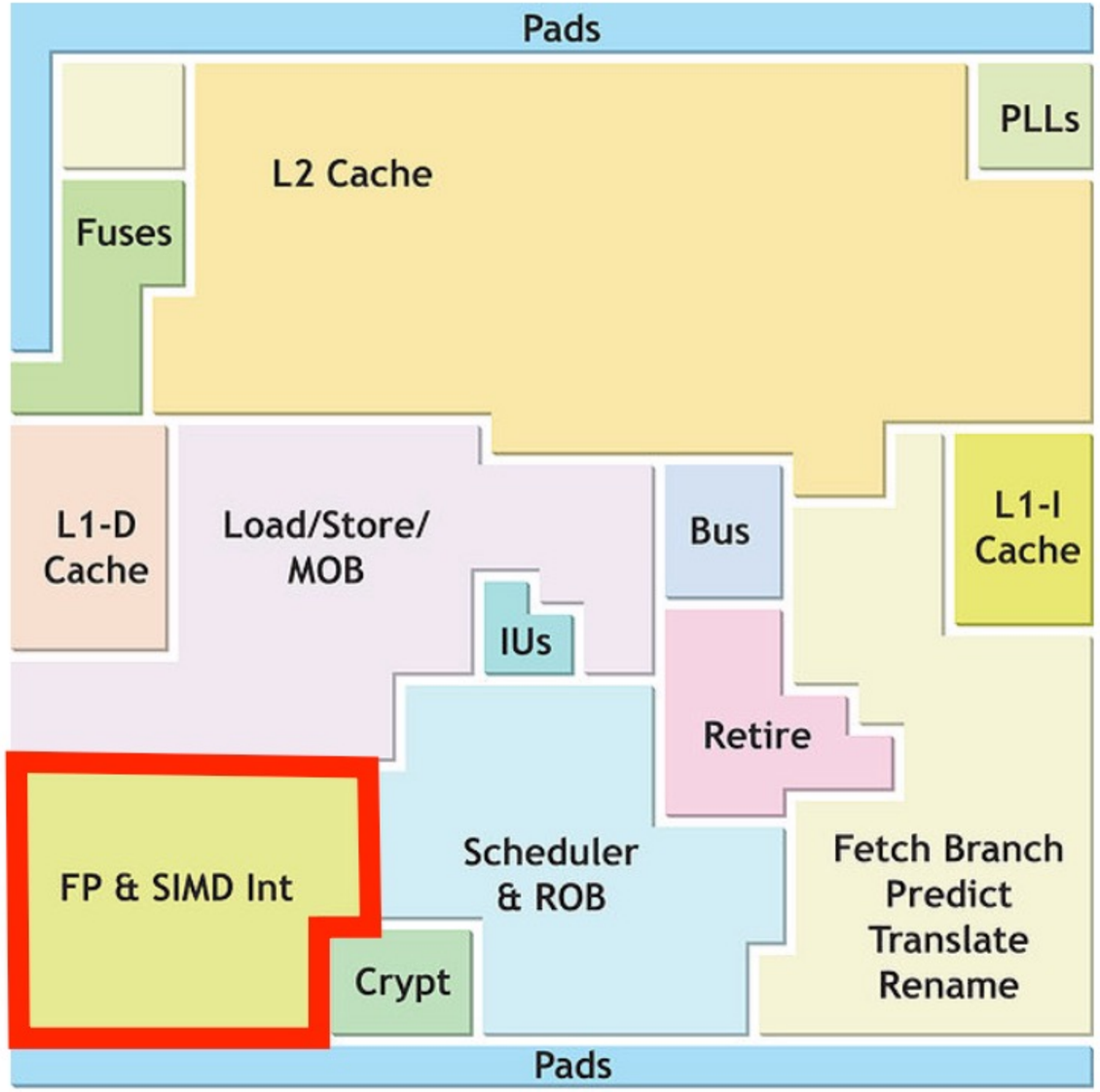
This pipeline structure illustrates how Intel processors achieve high performance by combining deep pipelines, extensive parallel execution, and advanced memory-hierarchy management.



Intel Core 2 Architecture






Pads
L2 Cache
PLLs
Fuses
L1-D Cache
Load/Store/ MOB
Bus
L1-I Cache
IUs
Retire
FP & SIMD Int
Scheduler & ROB
Fetch Branch Predict Translate Rename
Crypt
Pads

What are the lessons learned?

–Tom Jerry

Instruction level parallel is hard…

–Tom Jerry

GPU Architecture for Machine Learning

Workload is the king, as always

GPU was born differently

The computational complexity of matrix multiplication is $O(n^3)$. A typical formulation is:

$$> C = A \times B, \qquad C_{ij} = \sum_k A_{ik} \cdot B_{kj} >$$

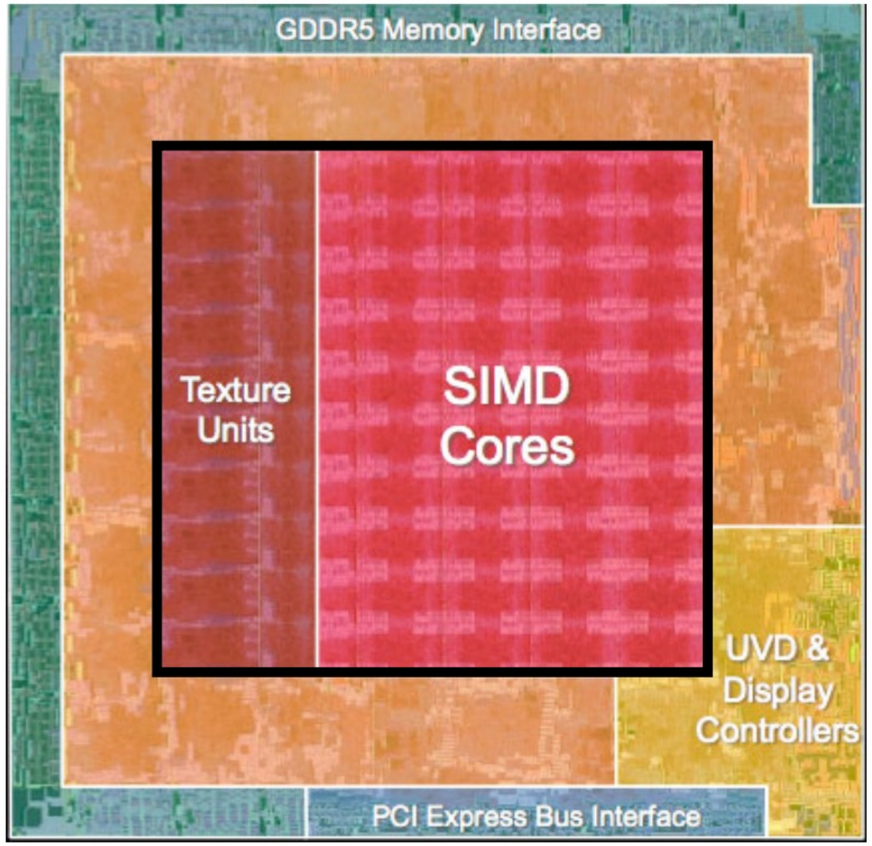
$$\begin{bmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,n-1} \\ b_{1,0} & b_{1,1} & \cdots & b_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n-1,0} & b_{n-1,1} & \cdots & b_{n-1,n-1} \end{bmatrix}$$

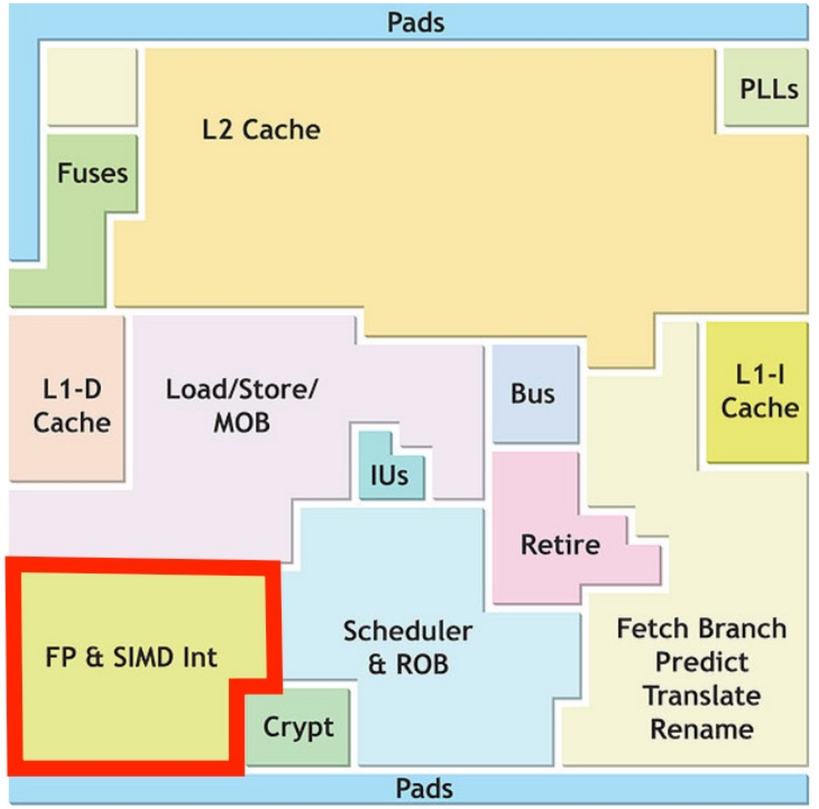
$$\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix} \quad \begin{bmatrix} \\ \\ \\ \\ \end{bmatrix}$$

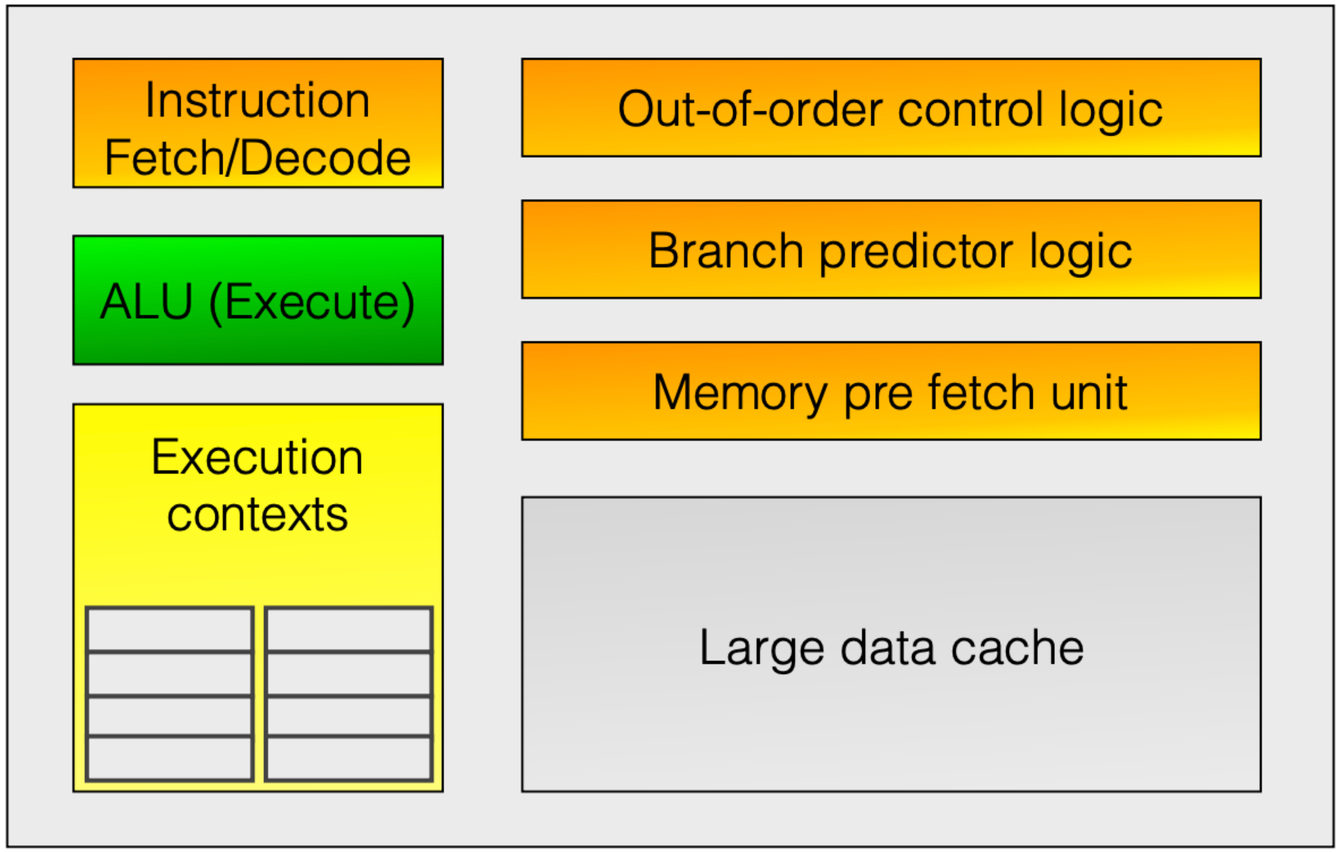
GPU was born differently

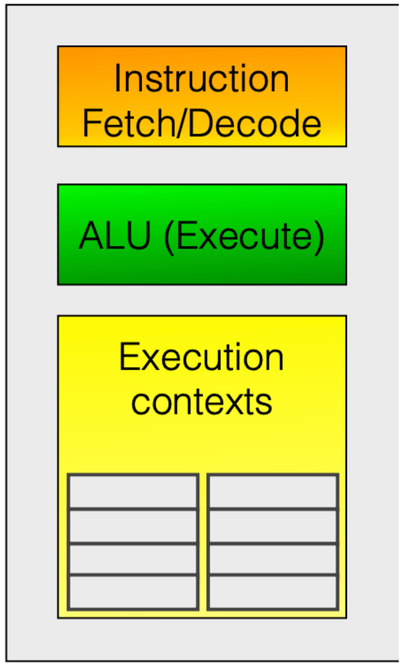
- Throughput matters and single threads do not.
- Hide memory latency through parallelism.
- Let programmer/software deal with “raw” storage hierarchy.
- Avoid high frequency clock speed, desirable for massive parallel computing

CPU vs. GPU

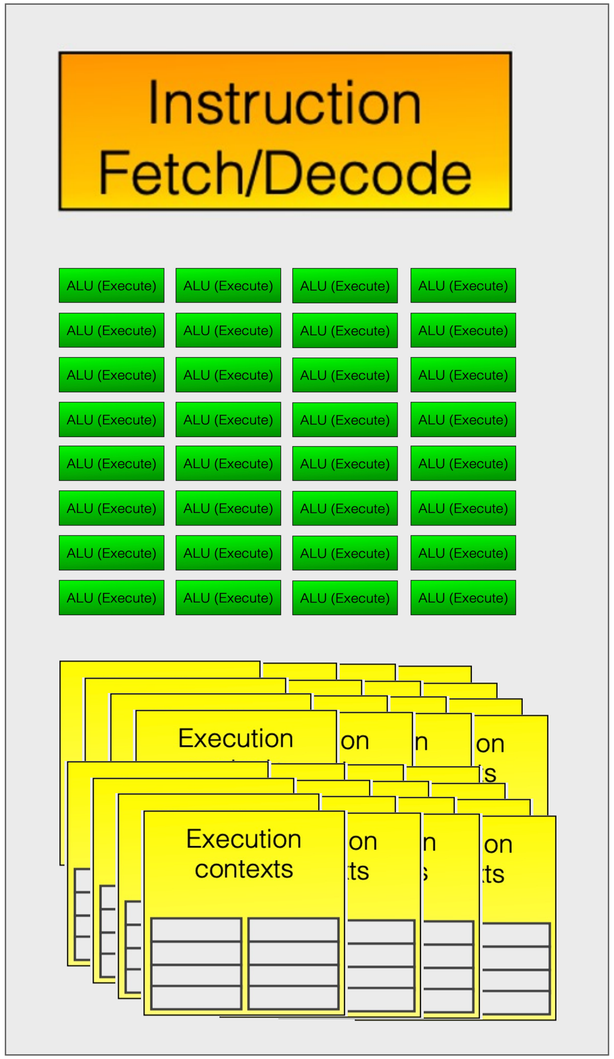

GDDR5 Memory Interface
Texture Units
SIMD Cores
UVD & Display Controllers
PCI Express Bus Interface


Pads
PLLs
L2 Cache
Fuses
L1-D Cache
Load/Store/ MOB
Bus
L1-I Cache
IUs
Retire
FP & SIMD Int
Scheduler & ROB
Fetch Branch Predict Translate Rename
Crypt
Pads

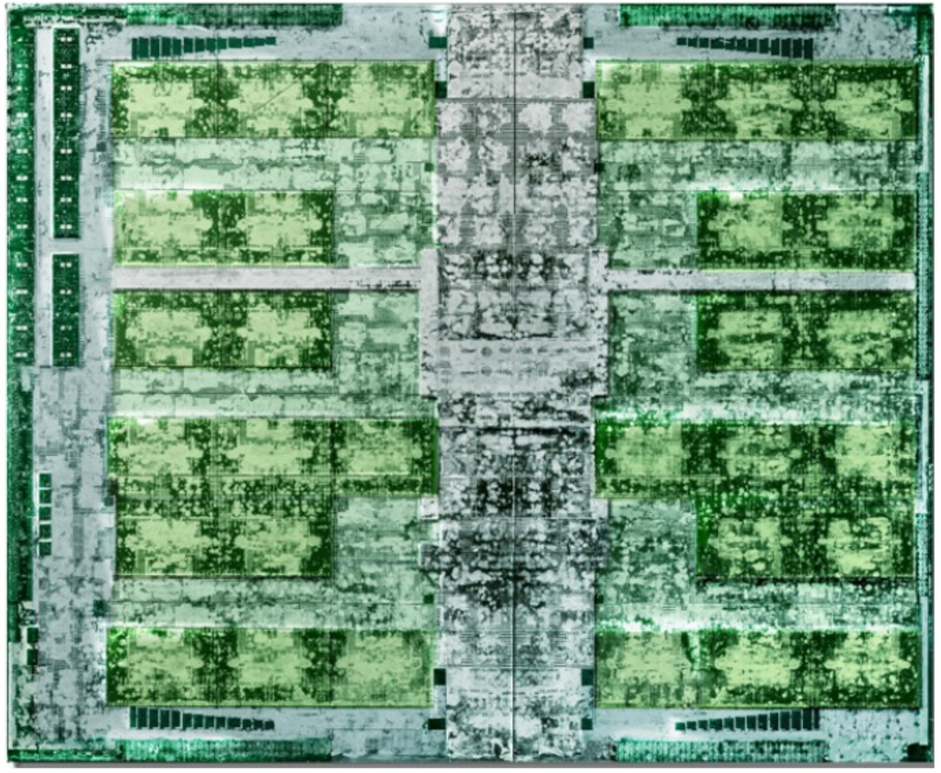

Instruction Fetch/Decode
ALU (Execute)
Execution contexts
Out-of-order control logic
Branch predictor logic
Memory pre fetch unit
Large data cache


Instruction
Fetch/Decode
ALU (Execute)
Execution
contexts

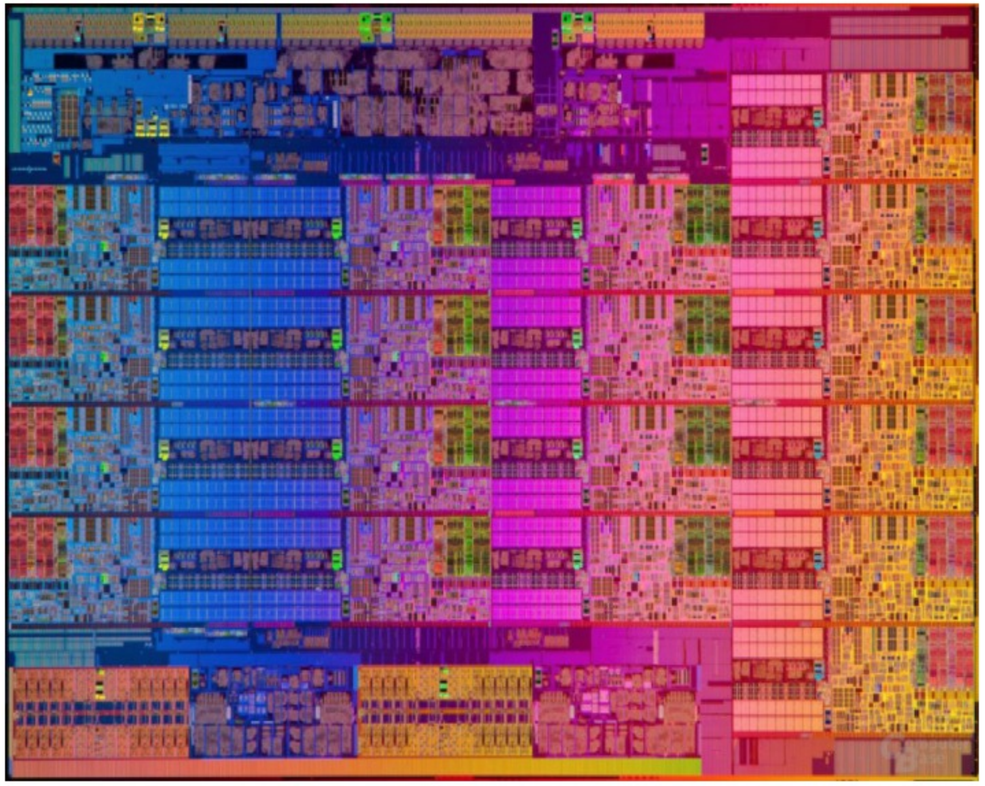

Instruction Fetch/Decode
ALU (Execute)
Execution contexts


Instruction Fetch/Decode
ALU (Execute)
Execution contexts


Instruction Fetch/Decode
ALU (Execute)
Execution contexts



NVIDIA P100: 56 “cores” with
4 32-way SIMT units



Intel E7-8894 v4: 24 hyper- threading
cores with 256 bit AVX2 instructions

A 30-year Journey

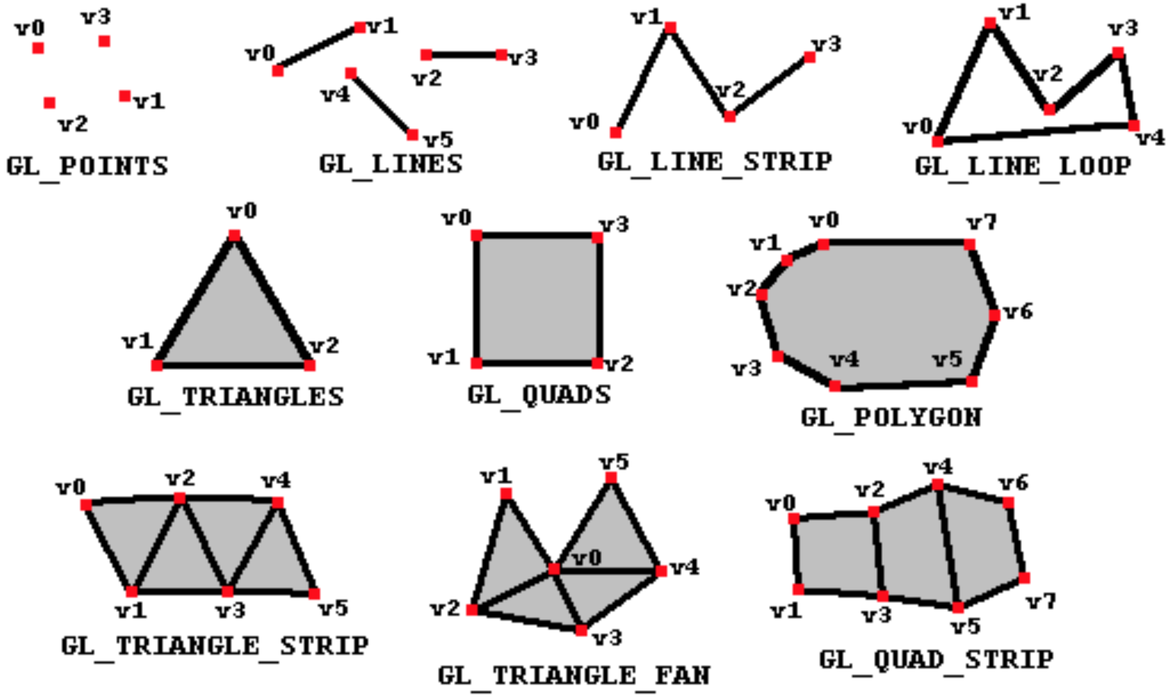
- **1993:** NVIDIA is founded by Jensen Huang, Chris Malachowsky, and Curtis Priem, focusing on graphics processing technology.
- **1997:** The company gains prominence in the gaming industry with the release of the RIVA series of graphics processors.
- **1999:** NVIDIA introduces the term “GPU” (Graphics Processing Unit) with the launch of the GeForce 256, marking a significant advancement in graphics processing.
- **2000:** NVIDIA acquired assets from 3dfx, a former competitor in the graphics card market, enhancing its technological portfolio and market position.
- **2006:** The company launches CUDA, a parallel computing platform and programming model, enabling GPUs to be used for general-purpose processing.
- **2018:** NVIDIA’s GPUs become the powerhouse to AI research and applications, solidifying its position in the AI industry.
- **2022-now:** The company’s market capitalization started to take off at the end of 2022, and surpasses $5 trillion in 2026, reflecting its leadership in AI and high-performance computing.

A Bumpy Journey

Ups and Downs

- **1995:** NVIDIA's first graphics accelerator, the NV1, was designed to process quadrilateral primitives, differing from competitors who preferred triangle primitives. When Microsoft introduced DirectX, it exclusively supported triangles, leading to the NV1's market failure.
- **1996:** NVIDIA partnered with Sega to supply the graphics chip for the Dreamcast console. However, NVIDIA's technology lagged behind competitors, and Sega chose another vendor. Sega's president, Shoichiro Irimajiri, invested $5 million in NVIDIA, which kept the company afloat during this challenging period.
- **2008:** NVIDIA faced a significant setback due to manufacturing defects in certain mobile chipsets and GPUs, leading to a $200 million write-down and a class-action lawsuit.
- **2022:** The company announced the termination of its planned $40 billion acquisition of Arm Holdings, citing regulatory challenges.
- **January 2025:** NVIDIA experienced a substantial market capitalization loss of $589 billion following the emergence of the Chinese startup DeepSeek, which introduced a low-cost AI model. This event marked the largest single-day loss in stock market history.

Historical question: why is quadratic texture mapping bad?



CASPA
since 1991

CASPA 華美半導體協會

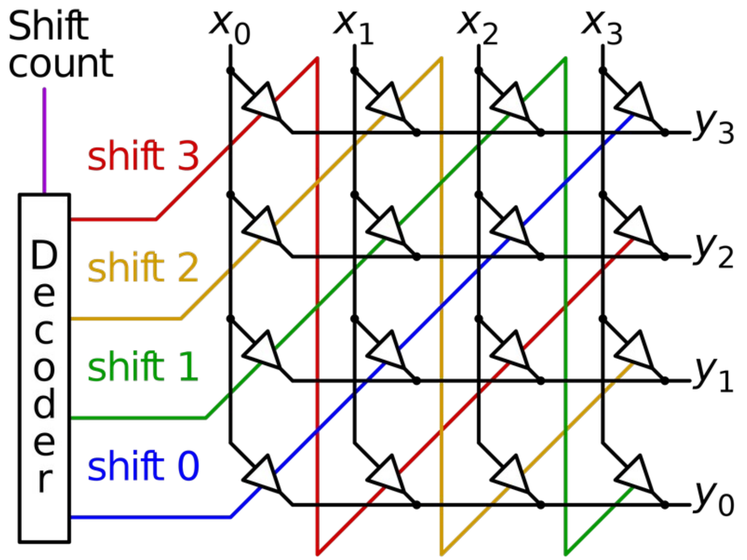
CHINESE AMERICAN SEMICONDUCTOR PROFESSIONAL ASSOCIATION

Every great technology has a humble origin

–Tom Jerry

GPU V0.1: Barrel Shifter

The Memory Constraints of Early Graphics Systems. Early arcade machines could not afford to store a full framebuffer due to cost and space limitations.
The Solution: Instead of pre-storing the entire frame in memory, these systems used real-time compositing where video chips assembled graphics dynamically as the screen was being drawn (a process called raster scan output).



1️⃣ Alien sprite data is stored in memory

- The CPU loads the **bitmap** from memory.

2️⃣ The barrel shifter shifts the sprite left or right instantly

- The CPU **doesn't modify the original data**, just controls how much to shift.

3️⃣ The new shifted data is sent directly to the display

- The arcade hardware **composites the sprite** in real-time onto the video output.

Key Barrel Shifter Operations in Graphics

Operation	Purpose	Example in Early Games
Bit Shifting	Moves sprite data horizontally/vertically	Sliding backgrounds in **Space Invaders (1978)**
Bit Rotation	Wraps bits around during shifts	Looping animations without recomputing data
Masking & Merging	Composites multiple bitmaps together efficiently	Overlaying characters over scrolling backgrounds

GPU: Three Generations

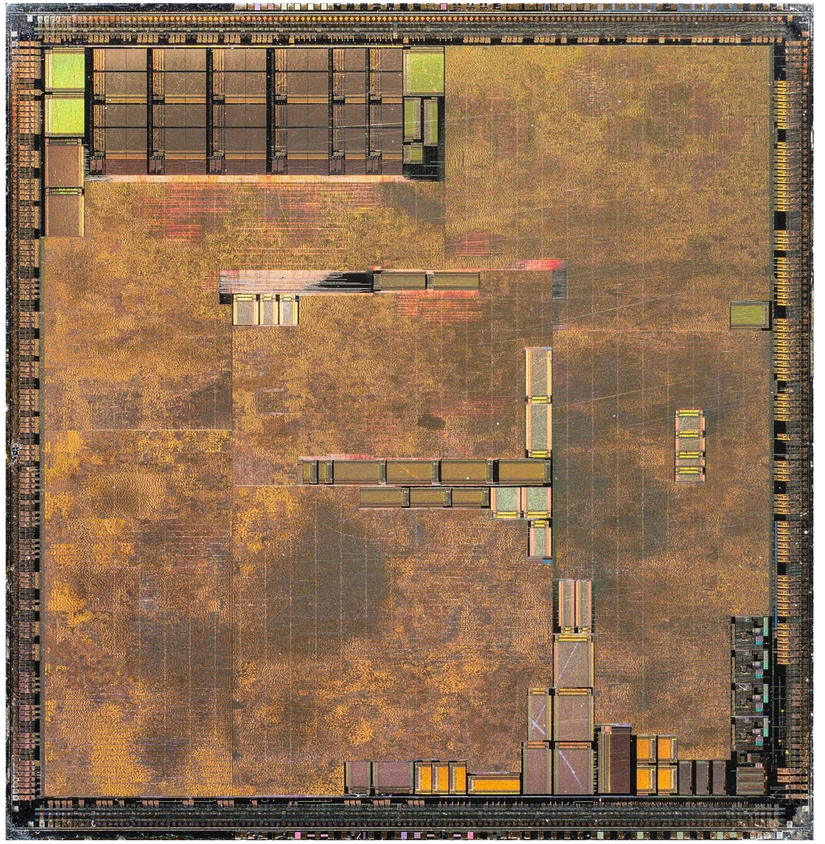
- GPU evolution: From fixed-function accelerator to programmable processor.
- Early adoption in machine learning: The massive parallel nature suitable for linear algebra.
- GPU in the deep learning : Architecture evolution driven by rapid growth of deep learning.

Phase	Era	Key Features	Examples
1️⃣ **Hardwired GPU (Fixed-Function Graphics)**	**1980s – Early 2000s**	- Specialized **fixed-function pipelines** for rendering - No programmability, only hardware-based transformations & rasterization	- **1981:** IBM CGA (First consumer graphics card) - **1999:** NVIDIA GeForce 256 (First GPU)
2️⃣ **Programmable GPU (Shader-Based Graphics Processing)**	**Early 2000s – 2010s**	- **Vertex & Pixel Shaders** introduced for **custom effects** - Unified Shader Architecture allows **software-defined rendering**	- **2001:** NVIDIA GeForce 3 (First programmable vertex shader) - **2006:** NVIDIA Tesla (First Unified Shader)
3️⃣ **GPGPU for AI/ML (General-Purpose GPU Computing)**	**2010s – Present**	- **CUDA/OpenCL** enable **parallel computing** for AI, HPC - **Tensor Cores & AI Accelerators** introduced	- **2007:** NVIDIA CUDA (GPGPU programming) - **2017:** NVIDIA Volta (First Tensor Cores for ML)

Phase 1: Hardwired GPU with fixed function pipeline

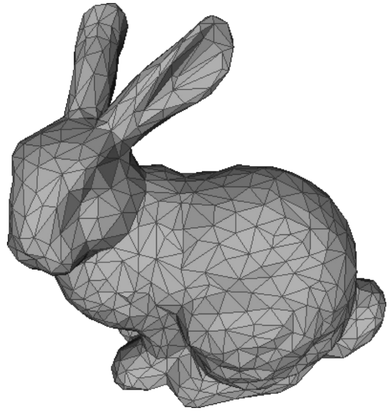
- **1980s-2000s:** GPUs were fixed-function, meaning all graphics tasks were hardcoded in hardware.
- **Key Limitations:** No programmability; only predefined transformations, rasterization, and texture mapping.
- **Examples:** IBM CGA (1981), NVIDIA GeForce 256 (1999) (first consumer GPU).

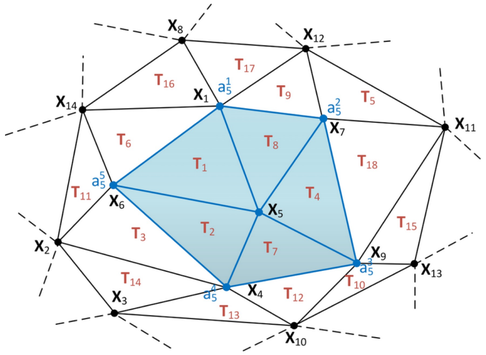
A single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second.

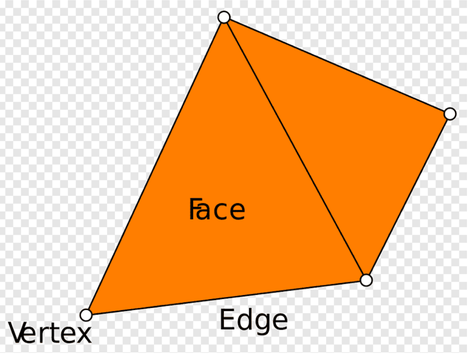


GPU is a Massive Shader

1. **3D Model Representation**:
   - A complex 3D model, such as a human figure, is represented in a digital environment.
2. **Mesh Generation**:
   - The surface of the 3D model is divided into numerous small triangles, forming a mesh.
   - Each triangle consists of three vertices, which are points in 3D space.
3. **Vertex Data Extraction**:
   - The coordinates and other attributes (like color, normal vectors, texture coordinates) of each vertex are extracted.
   - This data is organized into structures known as Vertex Buffer Objects (VBOs).
4. **Storage in Vertex Buffers**:
   - The VBOs are stored in the GPU's memory to facilitate efficient rendering during the graphics pipeline process.

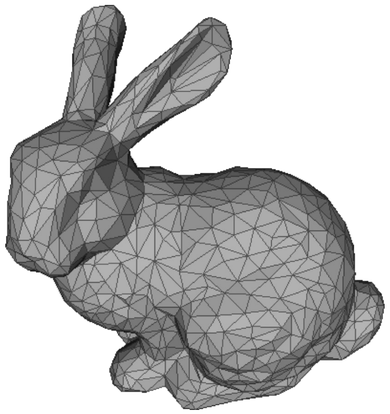


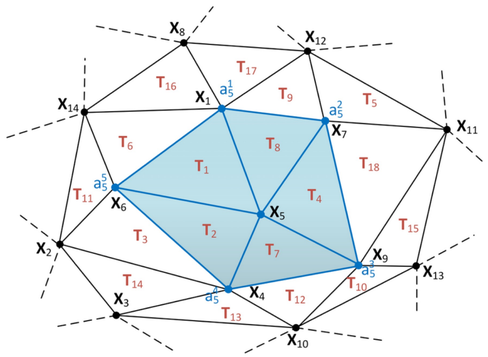


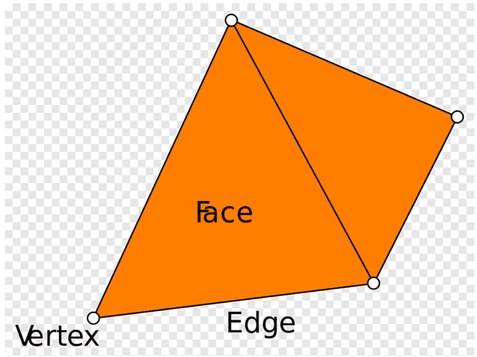


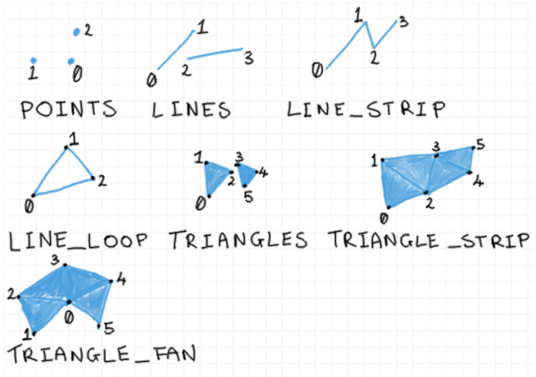
Vertex

1. **Vertex Position**: Specifies the 3D coordinates of the vertex in object space.
2. **Vertex Color**: Defines the color associated with the vertex, allowing for vertex-based color interpolation.
3. **Texture Coordinates**: Indicate how textures are mapped onto the surface of the geometry.
4. **Normal Vectors**: Provide information about the surface orientation at the vertex, essential for lighting calculations.
5. **Tangent and Bitangent Vectors**: Used in advanced shading techniques, such as normal mapping, to handle texture orientation.
6. **Bone Weights and Indices**: Utilized in skeletal animation to determine the influence of bones on the vertex.





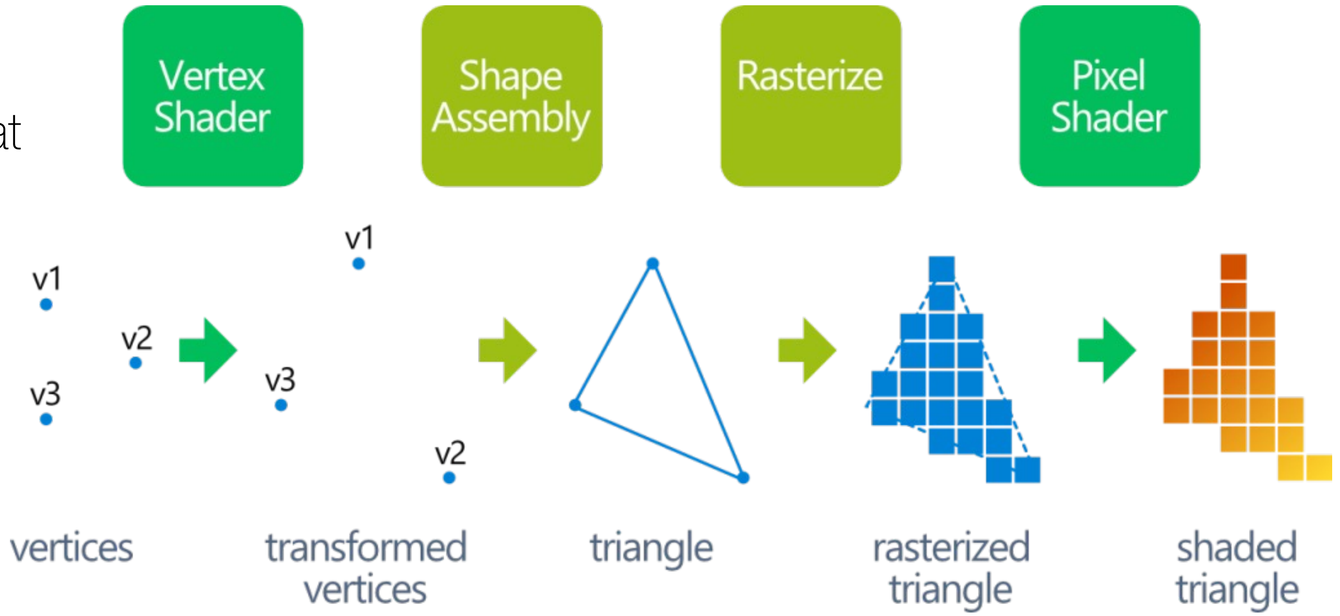




GPU is a Massive Shader

The graphics rendering pipeline consists of several stages, each responsible for specific tasks in transforming 3D models into 2D images.

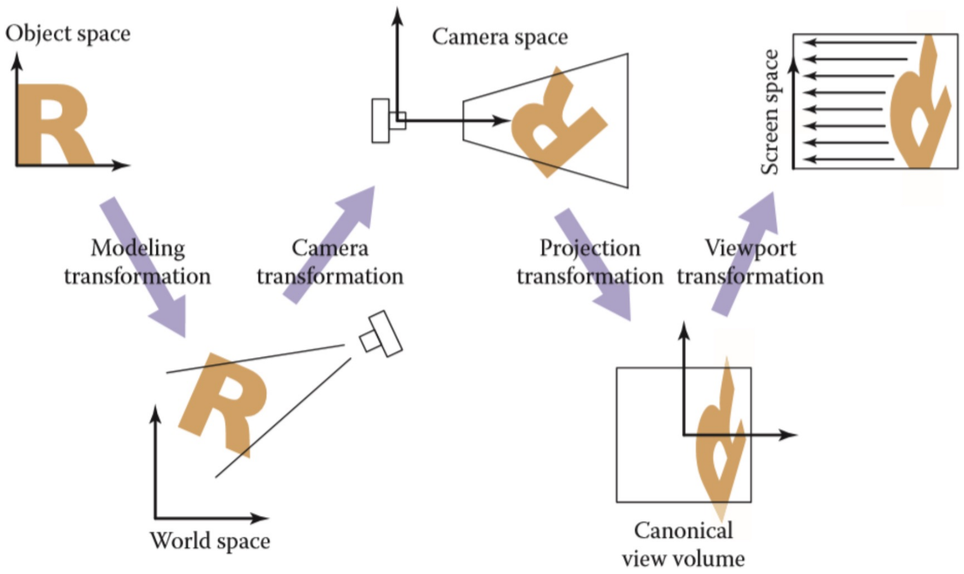
- **Vertex Transformation Stage:** Transforms individual triangle vertices from 3D world space to 2D screen space.
- **Shape Assembly Stage:** A fixed-function stage that assembles transformed vertices into geometric shapes, typically triangles.
- **Rasterization Stage:** Another fixed-function stage that converts assembled triangles into a set of pixels or fragments.
- **Pixel Shader Stage:** Determines the color of each pixel. Programmed using a pixel shader (also known as a fragment shader in OpenGL and Metal), which is executed once per pixel.



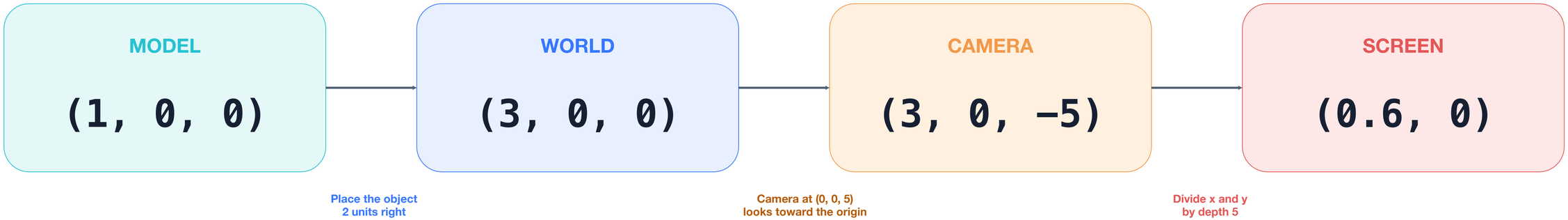
MVP: Modeling-View-Projection translation

To display objects from three-dimensional space on a two-dimensional plane, they must undergo the MVP transformation. The MVP transformation refers to three stages: Model transformation (Model), View transformation (View), and Projection transformation (Projection).

- **Model transformation (M):** moves an object from its local coordinate system into the shared world.
- **View transformation (V):** expresses the world relative to the camera.
- **Projection transformation (P):** maps the camera-space scene into clip space using perspective or orthographic projection.



MVP: Modeling-View-Projection translation



What changes at each step

MODEL `(1, 0, 0) + (2, 0, 0) = (3, 0, 0)`

VIEW The vertex is 3 units right and 5 units in front of the camera.

SIGN With the usual convention, a point in front has negative z.

Perspective calculation

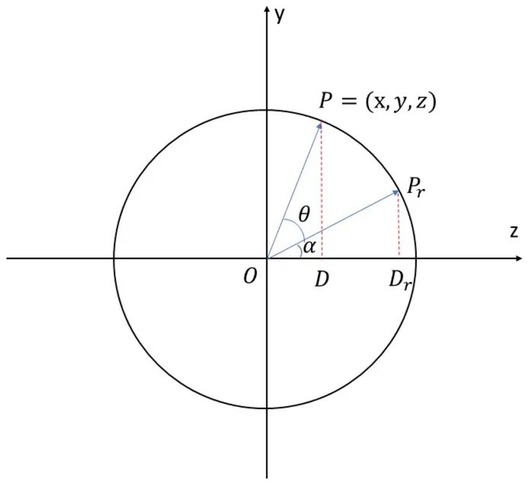
```
x_screen = 3 / 5 = 0.6
y_screen = 0 / 5 = 0
```

Result: the point appears right of center.

Greater depth makes screen coordinates smaller: 3 / 5 = 0.6, while 3 / 10 = 0.3.

Modeling Translation

Model transformation is the matrix that transforms an object from **local space** to **world space**. This matrix **changes as the object moves or transforms**.

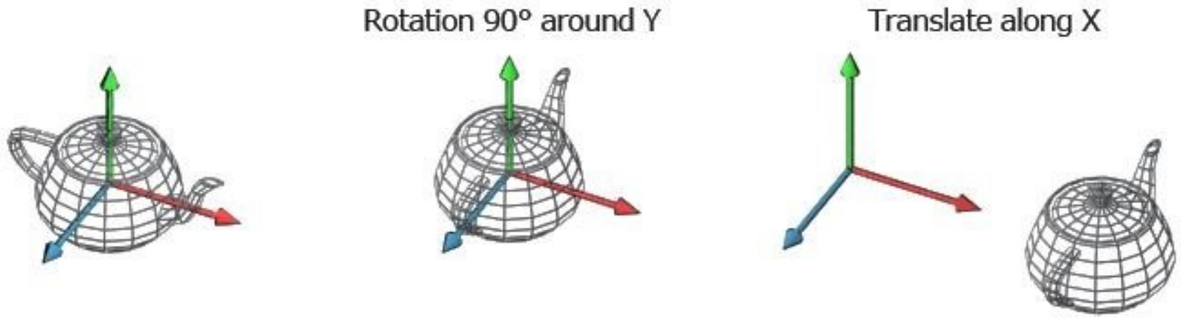


$$Pr = M\theta \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad \begin{bmatrix} x \\ \sin\alpha \\ \cos\alpha \\ 1 \end{bmatrix} = M\theta \cdot \begin{bmatrix} x \\ \sin(\alpha+\theta) \\ \cos(\alpha+\theta) \\ 1 \end{bmatrix}$$

$$y = \sin(\theta+\alpha) = \sin\theta\cos\alpha + \cos\theta\sin\alpha$$
$$z = \cos(\theta+\alpha) = \cos\theta\cos\alpha - \sin\theta\sin\alpha$$

$$y\sin\theta = \sin^2\theta\cos\alpha + \sin\theta\cos\theta\sin\alpha$$
$$z\cos\theta = \cos^2\theta\cos\alpha - \sin\theta\cos\theta\sin\alpha$$

$$\cos\alpha = y\sin\theta + z\cos\theta$$
$$\sin\alpha = y\cos\theta - z\sin\theta$$



$$Translate = \begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$Scale = \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M\theta x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M\theta z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M\theta y = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 0 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$Model = Translate \cdot Rotate \cdot Scale$$

GPU is a Massive Shader

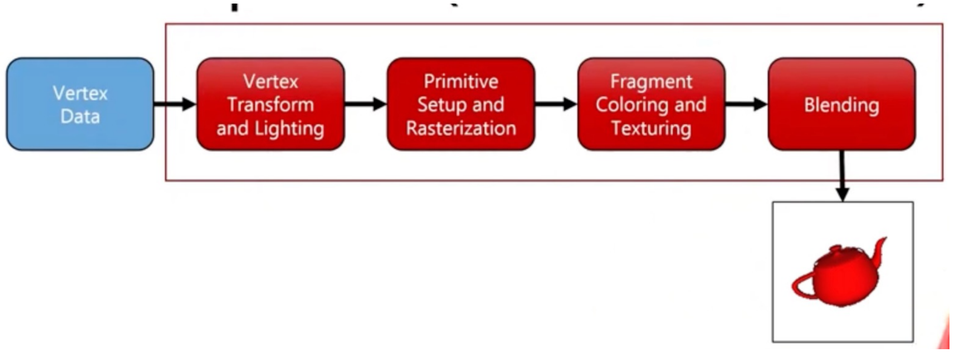
1. **Vertex Shader**: Processes individual vertices, transforming them from object space to clip space (e.g., applying world, view, and projection transformations).
2. **Tessellation Stage** (Optional): Subdivides geometry into smaller primitives (as explained in a previous response).
3. **Geometry Shader**: Takes entire primitives as input (e.g., a triangle with three vertices), processes them, and outputs zero or more primitives.
4. **Rasterization**: Converts the output primitives into fragments (pixels) for the Fragment Shader.
5. **Fragment Shader**: Computes the final color and other attributes of each pixel.

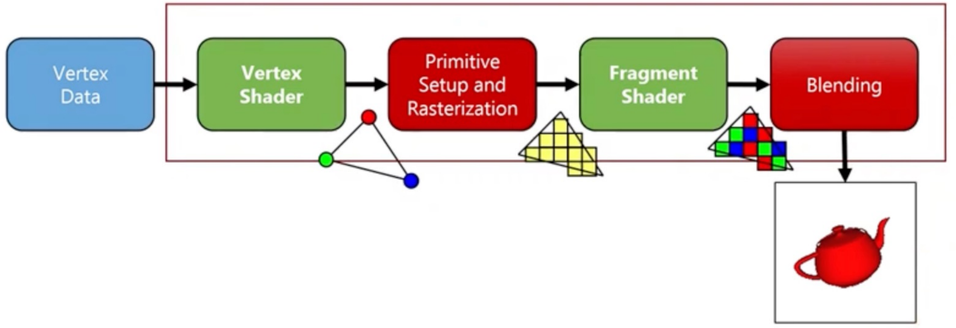
Software-Programmable GPU

- **2001-2010s:** GPUs introduced programmable shaders, allowing developers to write custom graphics effects.
- Key Innovations:
  - Vertex & pixel shaders (NVIDIA GeForce 3, 2001).
  - Unified Shader Architecture (NVIDIA Tesla, 2006).
  - DirectX & OpenGL revolutionized GPU programmability.
- Examples:
  - NVIDIA GeForce series
  - AMD/ATI Radeon series

Software-Programmable GPU

- **2001-2010s:** GPUs introduced programmable shaders, allowing developers to write custom graphics effects.
- Key Innovations:
  - Vertex & pixel shaders (NVIDIA GeForce 3, 2001).
  - Unified Shader Architecture (NVIDIA Tesla, 2006).
  - DirectX & OpenGL revolutionized GPU programmability.
- Examples:
  - NVIDIA GeForce series
  - AMD/ATI Radeon series

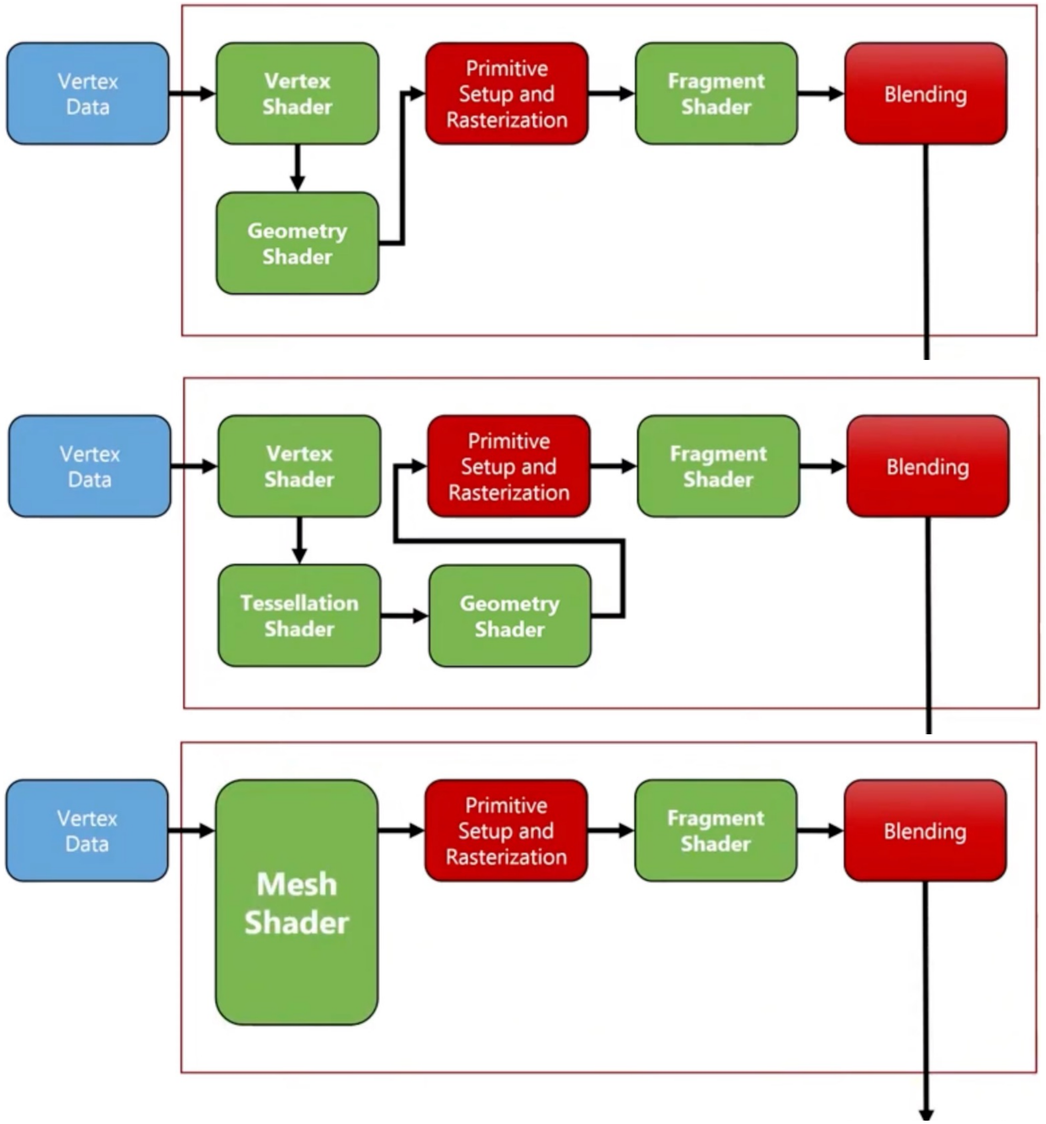




In the **second phase** of GPU evolution, characterized by the introduction of **programmable shaders**, several notable GPUs exemplify this transition:

1. **NVIDIA GeForce 3 Series (2001):**
   - Introduced the first programmable **vertex shaders**, allowing developers to write custom programs to manipulate vertex data, enabling more dynamic and realistic graphics effects.
2. **ATI Radeon 9700 (2002):**
   - Featured both programmable **vertex and pixel shaders**, providing enhanced flexibility in rendering complex visual effects and contributing to more immersive gaming experiences.
3. **NVIDIA GeForce 8 Series (2006):**
   - Implemented the **Unified Shader Architecture**, where previously separate vertex and pixel shaders were combined into a single programmable unit, optimizing resource utilization and performance.
4. **AMD Radeon HD 2000 Series (2007):**
   - Adopted a similar **unified shader model**, enhancing programmability and efficiency in rendering processes.

These advancements marked a significant shift from fixed-function pipelines to more flexible, programmable GPUs, laying the groundwork for subsequent developments in general-purpose computing on GPUs (GPGPU).



Software-Programmable GPU

Shader Stage	Purpose	Executed By
Vertex Shader	Transforms vertices to screen space	**CUDA Cores / Shader Cores**
Tessellation Shader	Dynamically subdivides geometry	**CUDA Cores / Shader Cores**
Geometry Shader	Processes full primitives (triangles, points)	**CUDA Cores / Shader Cores**
Mesh Shader (New in Vulkan & DX12)	Replaces Vertex & Geometry shaders for efficiency	**CUDA Cores / Shader Cores**
Pixel/Fragment Shader	Computes final color of each pixel	**CUDA Cores / Shader Cores**

GPU: Three Generations

- GPU evolution: From fixed-function accelerator to programmable processor.
- Early adoption in machine learning: The massive parallel nature suitable for linear algebra.
- GPU in the deep learning : Architecture evolution driven by rapid growth of deep learning.

Phase	Era	Key Features	Examples
1️⃣ **Hardwired GPU (Fixed-Function Graphics)**	**1980s – Early 2000s**	- Specialized **fixed-function pipelines** for rendering - No programmability, only hardware-based transformations & rasterization	- **1981:** IBM CGA (First consumer graphics card) - **1999:** NVIDIA GeForce 256 (First GPU)
2️⃣ **Programmable GPU (Shader-Based Graphics Processing)**	**Early 2000s – 2010s**	- **Vertex & Pixel Shaders** introduced for **custom effects** - Unified Shader Architecture allows **software-defined rendering**	- **2001:** NVIDIA GeForce 3 (First programmable vertex shader) - **2006:** NVIDIA Tesla (First Unified Shader)
3️⃣ **GPGPU for AI/ML (General-Purpose GPU Computing)**	**2010s – Present**	- **CUDA/OpenCL** enable **parallel computing** for AI, HPC - **Tensor Cores & AI Accelerators** introduced	- **2007:** NVIDIA CUDA (GPGPU programming) - **2017:** NVIDIA Volta (First Tensor Cores for ML)

Programmable GPU (GeForce 7800@2005)

Vertex Shader Engine: Handles per-vertex computations such as geometry transformations, vertex lighting, and vertex displacement.

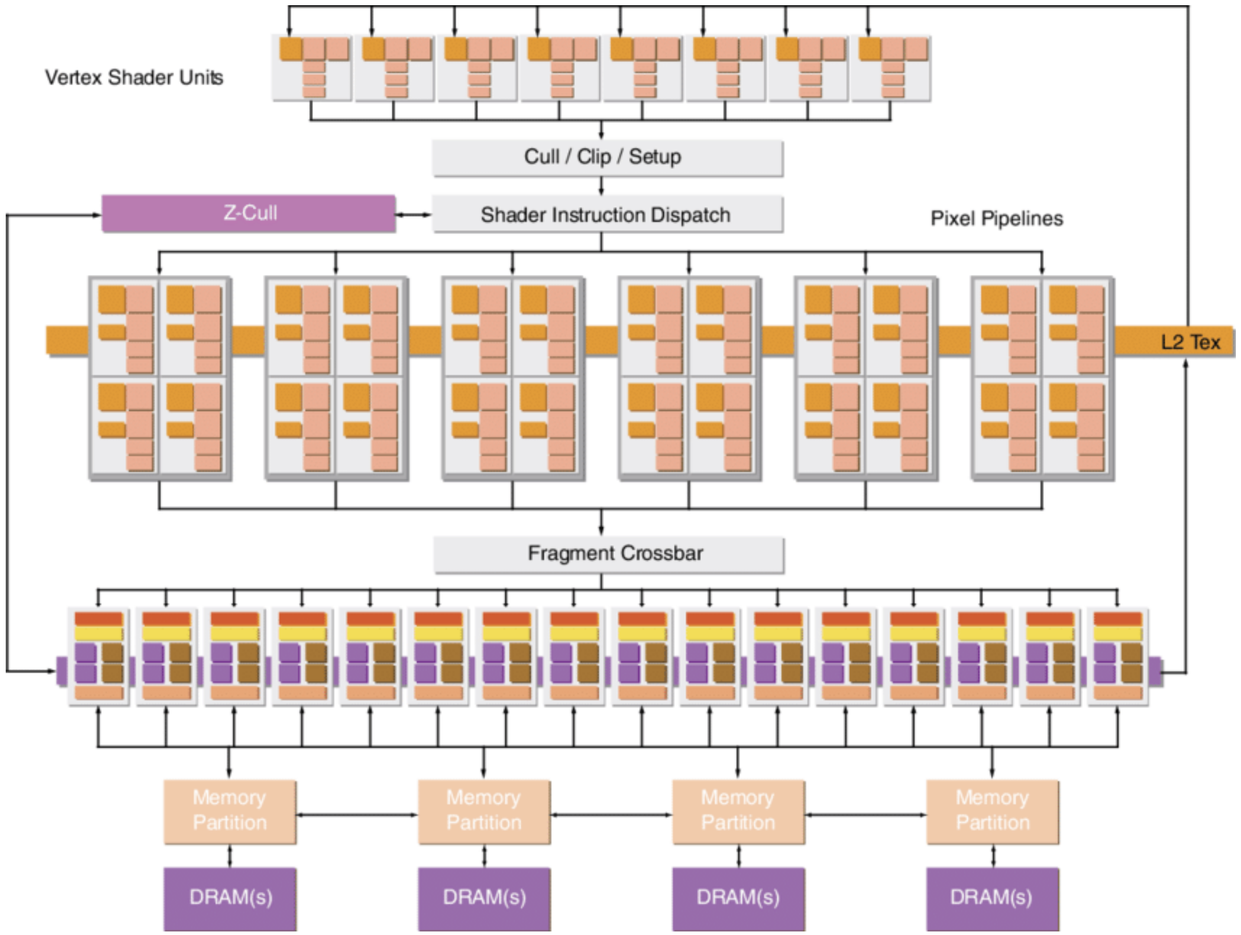
- Coordinate transformations (model → world → screen space)
- Per-vertex lighting calculations
- Vertex morphing and animation

Pixel Shader (Fragment Shader) Engine: Processes individual pixels (fragments), handling color calculations, texture blending, lighting effects, and other pixel-level operations.

- Per-pixel lighting and shading
- Texture mapping, filtering, and blending
- Complex visual effects (reflections, shadows, bump mapping)

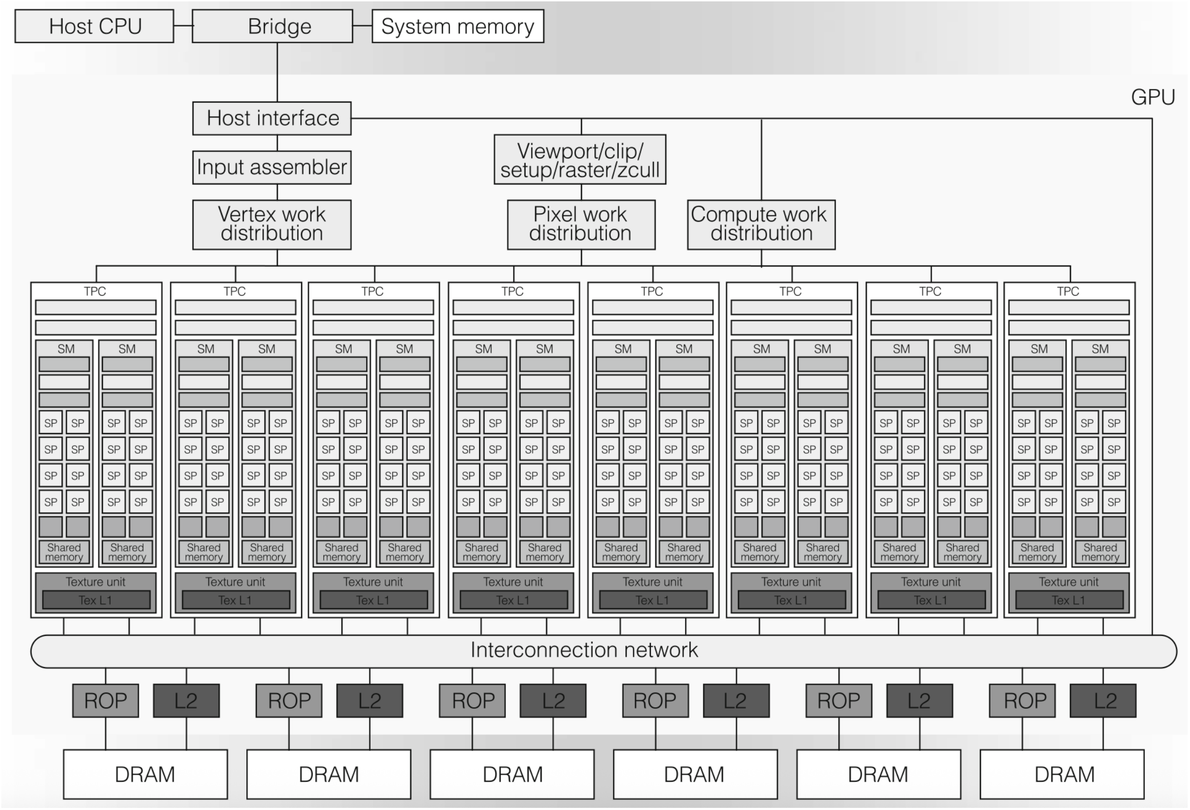
ROP (Raster Operations Pipeline): Final stage of the rendering pipeline, converting fragment outputs into pixels stored in the framebuffer.

- Depth (Z) testing and stencil operations
- Color blending and transparency
- Anti-aliasing (AA)
- Writing final pixel values to memory



Tesla 2006: Unified shader architecture

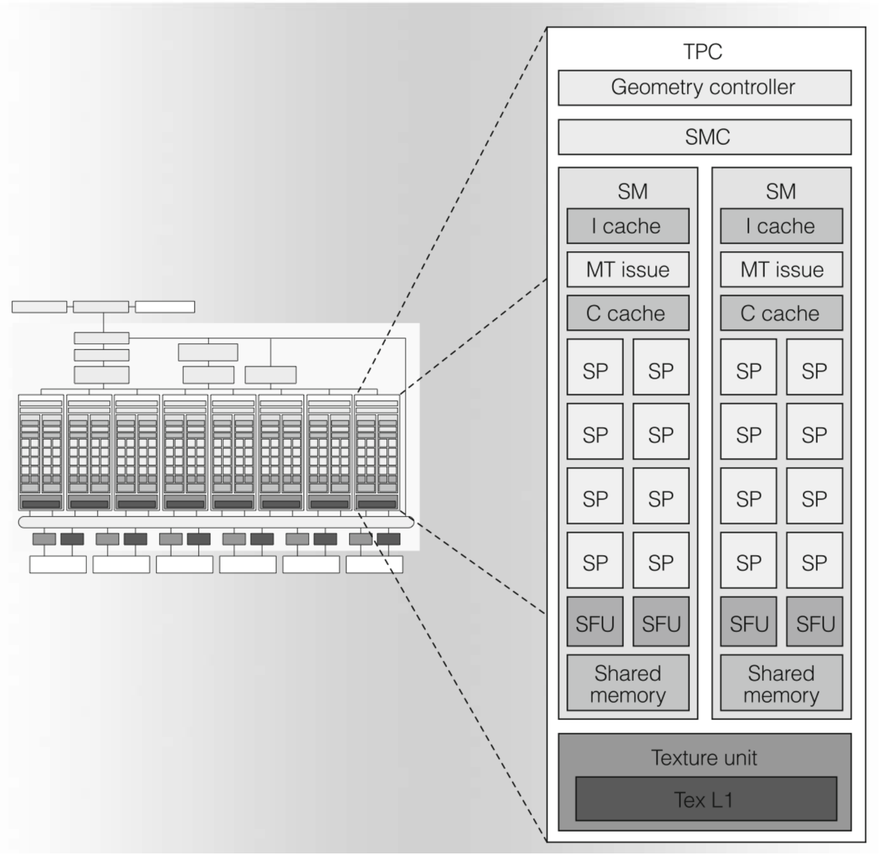
Overall Design	First CUDA-capable GPU (90nm, 2006), unified graphics and compute with five subsystems: control, compute, cache, memory, interconnect. Work distribution units dispatched tasks to scalable TPCs.
Key Modules	Each TPC had a Geometry Controller, SM Controller, 2 SMs, 1 Texture Unit + L1 cache, and 4 ROPs. Shared L2 caches, DRAM partitions, and display interface supported the whole chip.
SM Design	Each SM: 8 scalar SPs (FP32/INT32), 2 SFUs (math functions), 16 KB shared memory, instruction & constant caches. Scalar SP design simplified CUDA C support.
Workflow	CPU sends tasks → Input Assembler builds primitives → Vertex Shaders run → Rasterization & Z-cull → Pixel Shaders process fragments → ROPs handle blending/ depth/AA → output to memory/display.



In 2006, NVIDIA introduced the GeForce 8800. This design featured a “unified shader architecture” with 128 processing elements distributed among eight shader cores. Each shader core could be assigned to any shader task, eliminating the need for stage-by-stage balancing and greatly improving overall performance.

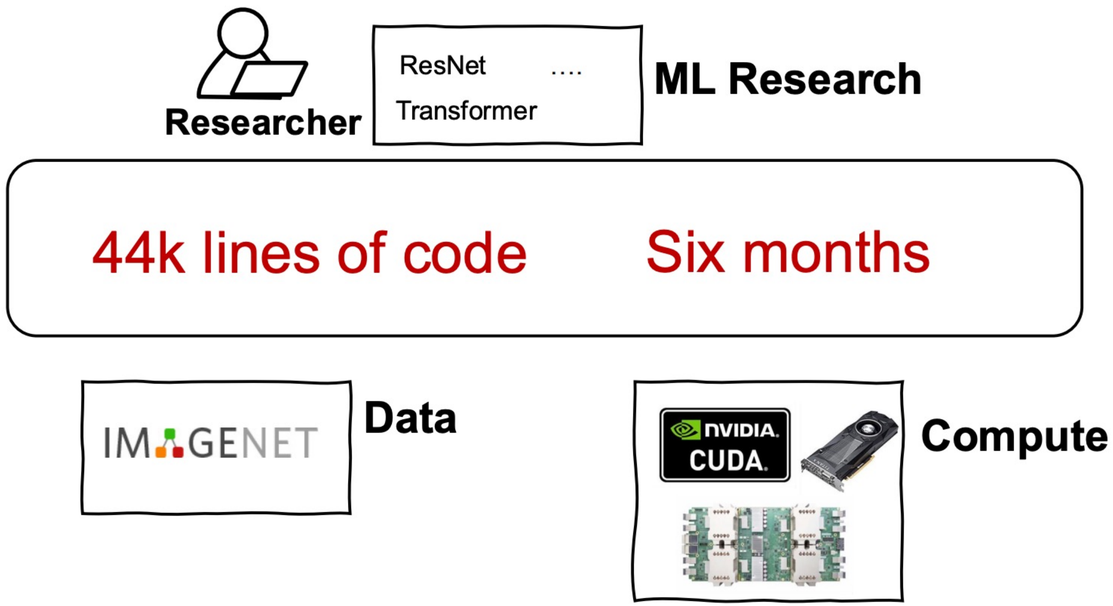
Tesla 2006: Unified shader architecture

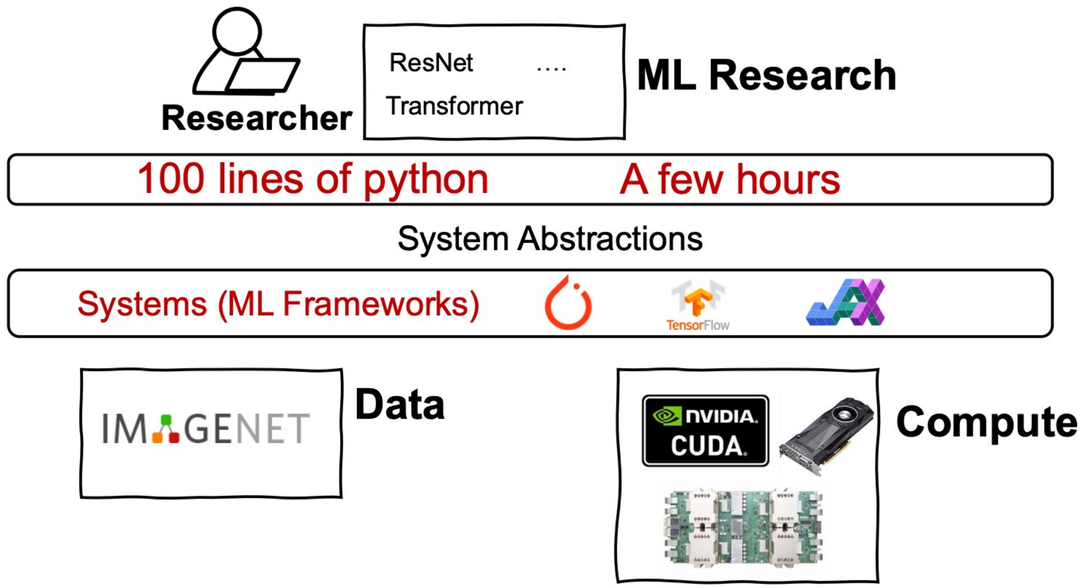
Overall Design	First CUDA-capable GPU (90nm, 2006), unified graphics and compute with five subsystems: control, compute, cache, memory, interconnect. Work distribution units dispatched tasks to scalable TPCs.
Key Modules	Each TPC had a Geometry Controller, SM Controller, 2 SMs, 1 Texture Unit + L1 cache, and 4 ROPs. Shared L2 caches, DRAM partitions, and display interface supported the whole chip.
SM Design	Each SM: 8 scalar SPs (FP32/INT32), 2 SFUs (math functions), 16 KB shared memory, instruction & constant caches. Scalar SP design simplified CUDA C support.
Workflow	CPU sends tasks → Input Assembler builds primitives → Vertex Shaders run → Rasterization & Z-cull → Pixel Shaders process fragments → ROPs handle blending/ depth/AA → output to memory/display.



In 2006, NVIDIA introduced the GeForce 8800. This design featured a “unified shader architecture” with 128 processing elements distributed among eight shader cores. Each shader core could be assigned to any shader task, eliminating the need for stage-by-stage balancing and greatly improving overall performance.

The Programming Challenge





CUDA Compute Unified Device Architecture

- **Programming Model:** It provides a programming model that allows for the development of software that can execute parallel computations efficiently on GPUs.
- **Parallel Computing Platform:** CUDA enables developers to harness the parallel processing power of NVIDIA GPUs for general-purpose computing tasks, facilitating significant performance improvements in various applications.
- **API Support:** CUDA offers an API that allows software developers to utilize NVIDIA GPUs for general-purpose processing, enabling the development of high-performance applications across various domains.
- **Extensive Ecosystem:** Over the years, NVIDIA has built a comprehensive ecosystem around CUDA, including specialized code libraries and AI models, making it a dominant platform in AI and high-performance computing.

The History of CUDA

SC08
Conference for High Performance Computing
Austin Texas