Machine Learning Systems

Build scalable and reliable ML services through the vertical integration of algorithms, system software, and hardware

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.

Course Logistics

Time: Tuesday 9:30-12:15

Location: H3409

Instructor: Li Shang, Yuedong Xu

Course components: Lecture & guest lecture & course projects and presentations

Course evaluation: class participation & discussion 30%, homework & paper reviews 20%, course project 50%

What is ML Systems

Our view： The process of Machine learning algorithm design, development and deployment with scalability in mind.

Vertically integrated optimization hierarchy： Algorithms, system software, hardware architecture, with biggest opportunities in the algorithm layer.

Computer system design is driven by the workload, so is ML sys.

- Machine intelligence is all about learning representations.
- Learning representations require tons of matrix multiplications.
- Processing is fast, latency is physics, and large-fast storage is impossible.
- Bending the laws of physics, is that possible?
- Who sells memory better, CPU or GPU?

Machine Learning Systems

ML Sys. is the discipline of designing, implementing, and operating artificially intelligent systems across computing scales—from resource-constrained embedded devices to warehouse-scale computers.

It focuses on algorithm-software-hardware co-design to create systems that are efficient, scalable, and reliable for their deployment context.

It encompasses the complete lifecycle of AI applications: from requirements engineering and data collection through model development, system integration, deployment, monitoring, and maintenance.

Schedule		Topics
1	Introduction	ML Sys Intro. CPU architecture review
2-3	CPU-GPU Architecture	Overview CPU-GPU architecture Advanced features
4-5	CUDA Programming	Fundamentals and architecture Management optimization and debugging Hands-on CUDA programming
6-7	Compiler & Optimization	DL compiler LLM compiler
8-9	Parallel Training	LLM basics Parallelism Computation communication memory optimization
10	Inference	LLM inference infra
11-12	Model Optimization	Stability, Quantization Pruning & Sparsification
13	Beyond Performance	Privacy, Safety, Security, Responsibility, Sustainability
14	Embodied AI	Automated driving systems Humanoid robotic systems
15	AI Accelerators	Customized AI accelerators
16	Project	Course project presentation

CPU Foundations and GPU Emergence

An overview of CPU/GPU from an optimization perspective.

Performance

$$\text{Processor Performance} = \frac{\text{Time}}{\text{Program}}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instructions}} \times \frac{\text{Time}}{\text{Cycles}}$$

(code size) (CPI) (cycle time)

Model Flops Utilization (MFU)

$$\frac{\text{optimal matmul time}}{\text{actual matmul time}} \qquad \frac{\text{matmul time}}{\text{compute time}} \qquad \frac{\text{compute time}}{\text{compute time} + \text{communication time}}$$

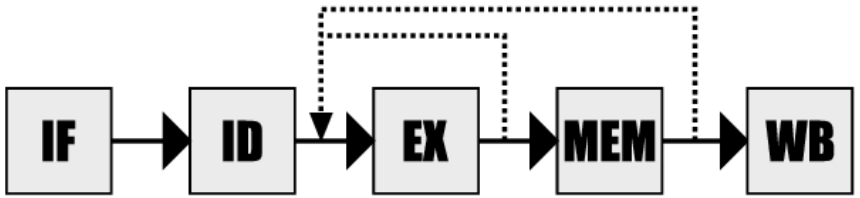
MFU Component	Mathematical Term	What It Measures	Dominant Factors
Kernel / Hardware Efficiency	$\frac{\text{optimal matmul time}}{\text{actual matmul time}}$	How close GEMMs run to peak hardware FLOPs	• matrix shapes (aspect ratios, sizes)• data layout (contiguity, alignment, transposes)• precision (FP16 / BF16 / FP8)• kernel fusion (epilogues, bias+GELU fusion)
Algorithmic Compute Density	$\frac{\text{matmul time}}{\text{compute time}}$	Fraction of computation spent in GEMMs vs non-GEMMs	• GEMMs (QKV, FFN, output projections)• elementwise ops (GELU, layernorm, dropout)• softmax (attention normalization)• reductions (layernorm stats, loss)• indexing / masking (attention masks, routing)
System / Parallel Efficiency	$\frac{\text{compute time}}{\text{compute time+communication time}}$	Fraction of step time spent doing useful compute	• all-reduce (gradient aggregation)• parameter synchronization (optimizer states)• pipeline bubbles (PP imbalance)• CPU–GPU stalls (launch latency, host sync)• memory bandwidth limits (HBM, NVLink)

Assembly Instruction

ADD r1, r3, #5

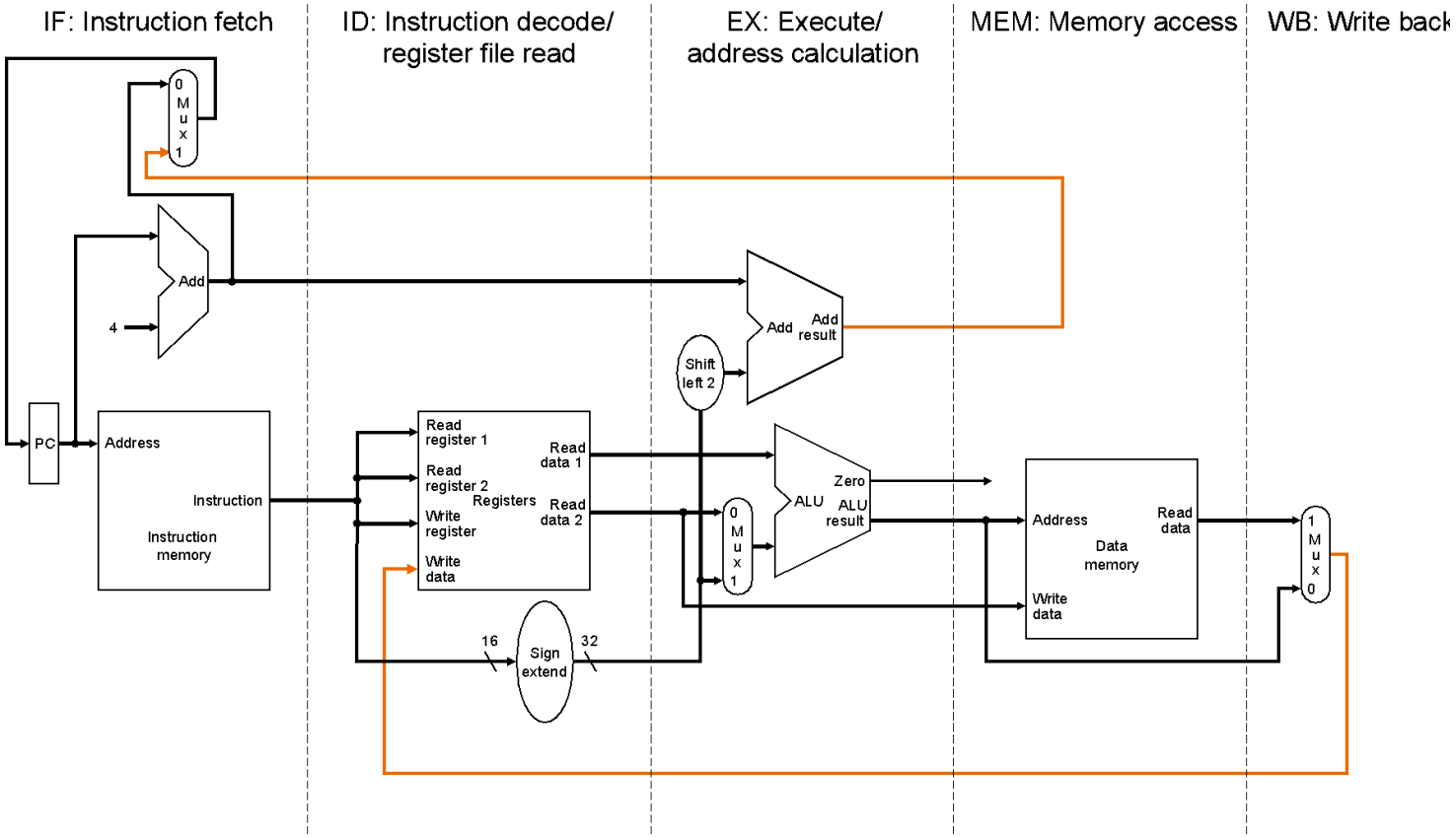
Machine Code (binary)

Instruction Execution



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

Datapath


IF: Instruction fetch
ID: Instruction decode/ register file read
EX: Execute/ address calculation
MEM: Memory access
WB: Write back
0
M
u
x
1
Add
4
PC
Address
Instruction
Instruction memory
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign extend
32
Shift left 2
Add
Add result
0
M
u
x
1
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
1
M
u
x
0

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;
}
```

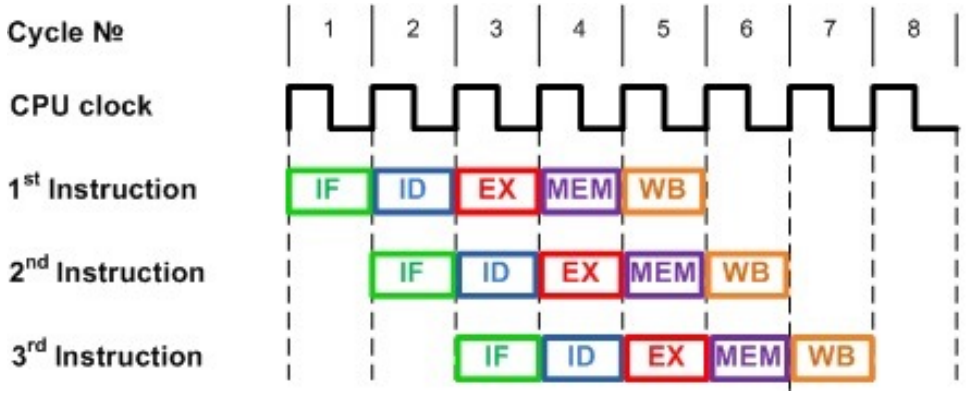
```c
int sum_array(int *A, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += A[i];
    }
    return sum;
}
```

```
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
```

Pipelining

How to improve CPI (cycles per instruction)?

- Instruction Latency = 5 cycles
- Instruction throughput = 1 instr/cycle
- CPI was 5 cycles per instruction
- CPI = 1 cycle per instruction
- CPI = cycle between instruction completion = 1



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)

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)

What is the catch?

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

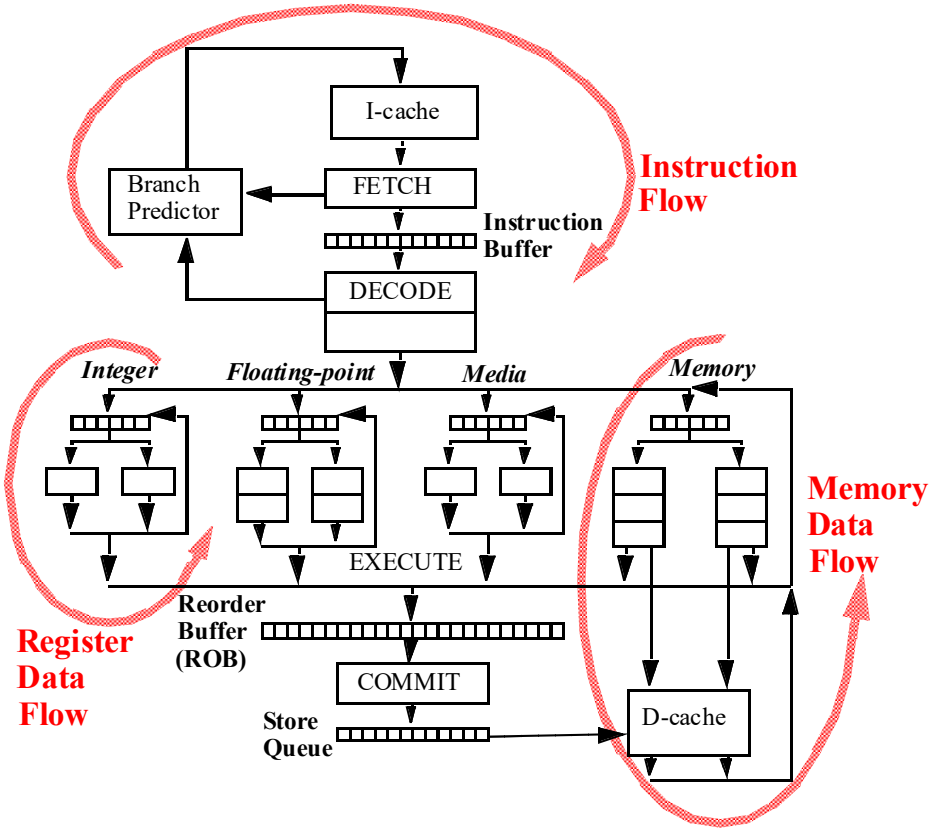
Hide memory latency

```
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.

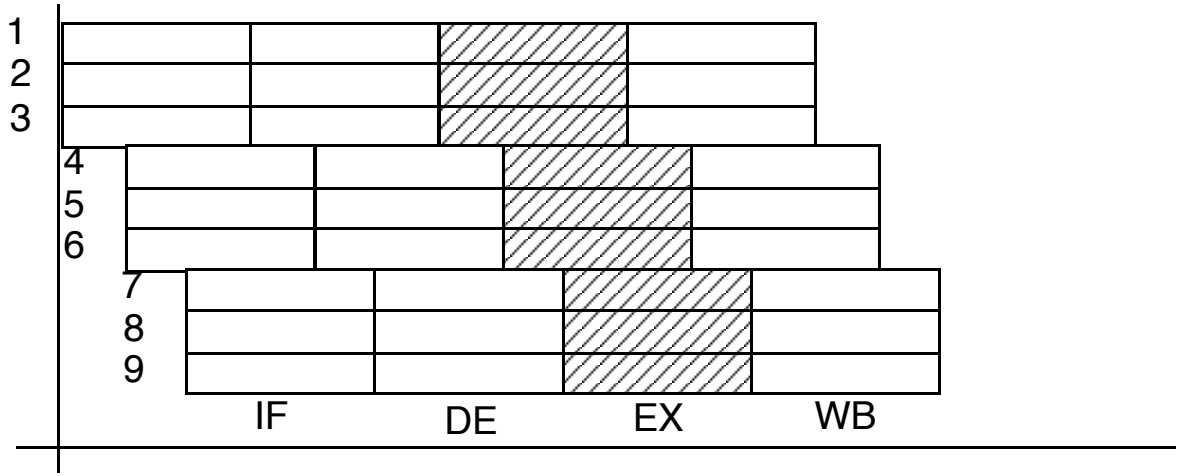
Superscalar

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



Superscalar

- Superpipelined-Superscalar
  - Issue parallelism = IP = n inst / minor cycle
  - Operation latency = OP = m minor cycles
  - Peak IPC = n x m instr / major cycle



🔷 Hardware Optimizations (no code change)

Optimization	What CPU Does	Benefit
Branch Prediction	Predicts loop will continue	Avoids pipeline stalls each iteration
Out-of-Order Execution (OoO)	While waiting for `A[i]` to load, speculatively executes `inc i`, `cmp i, n`, even pre-issues `load A[i+1]`	Hides memory latency
Multiple Issue (Superscalar)	Issues `load` + `add` in same cycle (if ready)	Parallelizes independent instructions
Caches (spatial/temporal)	- Spatial: `A[i..i+15]` come in one cache line- Temporal: `sum` stays hot in registers	Reduce memory stalls

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

Summary

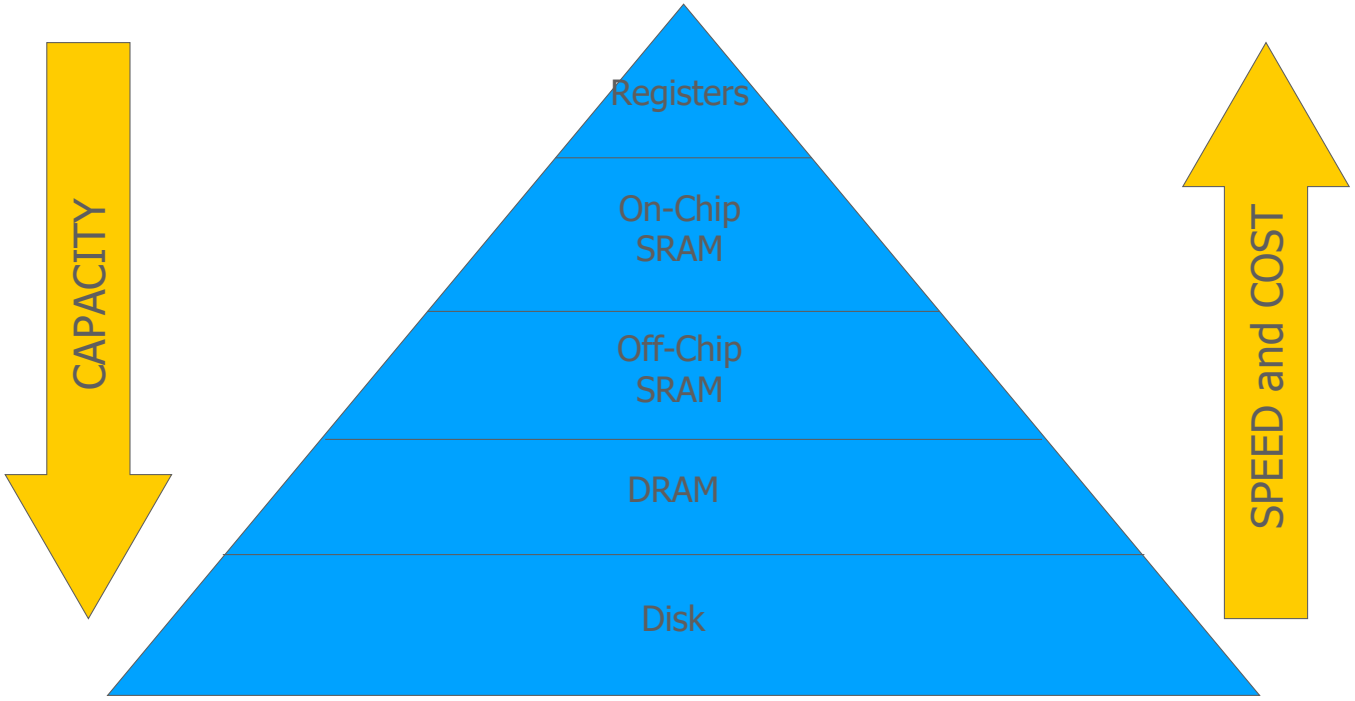
Dimension	Software (Compiler)	Hardware (CPU)	How They Help Each Other
Instruction Parallelism	Loop unrolling, vectorization → expose more independent ops	OoO & multiple-issue → schedule them in parallel	Compiler exposes → CPU exploits
Control Hazards	Loop unrolling reduces branch frequency	Branch predictor speculates next iterations	Compiler reduces branch count, CPU hides branch penalty
Data Locality	Ensure arrays contiguous; unroll to encourage cache line use; software prefetch	Cache hierarchy (L1/L2/L3), hardware prefetchers	Compiler arranges memory, CPU hardware delivers fast
Latency Hiding	Unrolled loops → more outstanding loads	OoO overlaps loads with adds, prefetchers fetch ahead	More outstanding memory ops → more latency hidden

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



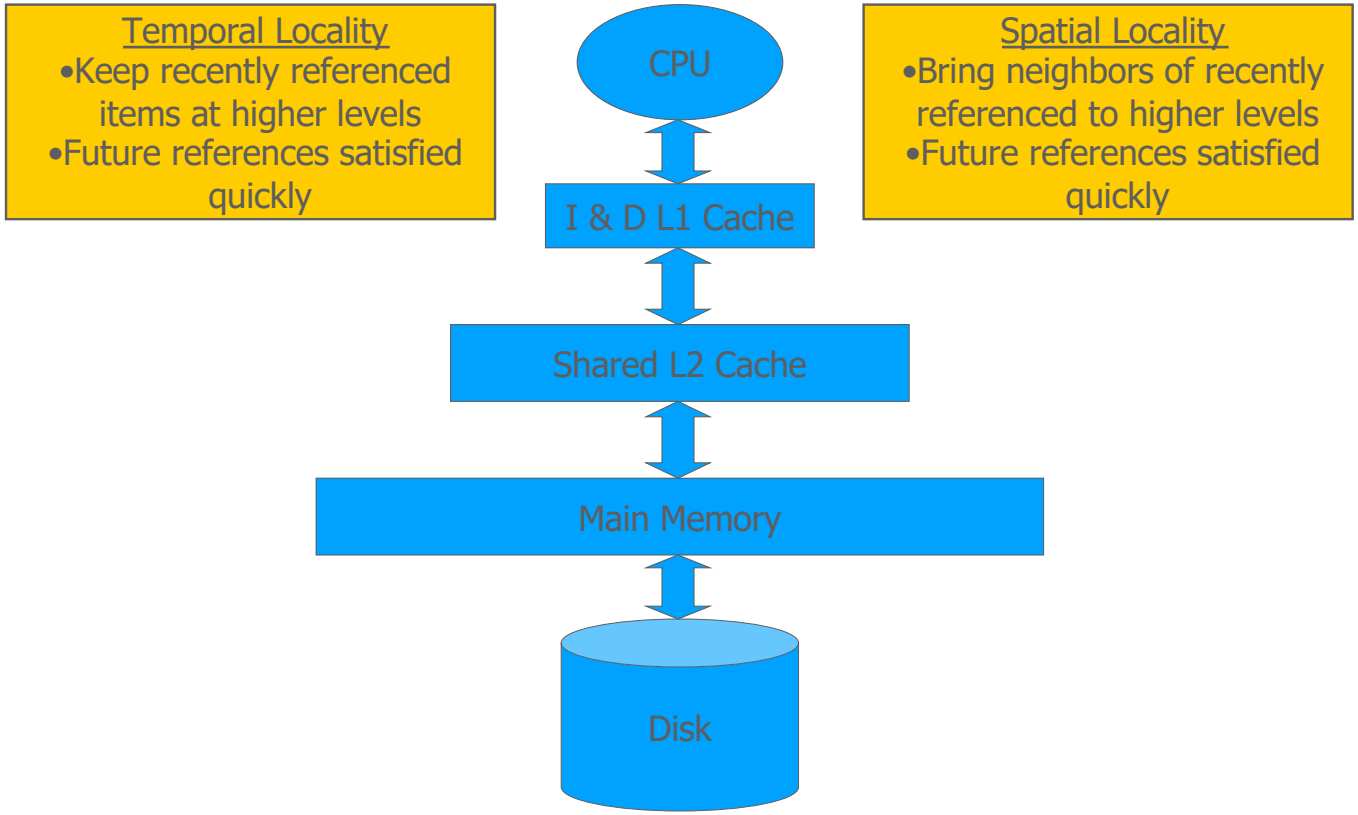
Why Memory Hierarchy?

- Fast and small memories
  - Enable quick access (fast cycle time)
  - Enable lots of bandwidth (1+ L/S/I-fetch/cycle)
- Slower larger memories
  - Capture larger share of memory
  - Still relatively fast
- Slow huge memories
  - Hold rarely-needed state
  - Needed for correctness
- All together: provide appearance of large, fast memory with cost of cheap, slow memory

Why Does a Hierarchy Work?

- Locality of reference
  - Temporal locality
    - Reference same memory location repeatedly
  - Spatial locality
    - Reference near neighbors around the same time
- Empirically observed
  - Significant!
  - Even small local storage (8KB) often satisfies >90% of references to multi-MB data set

Memory Hierarchy

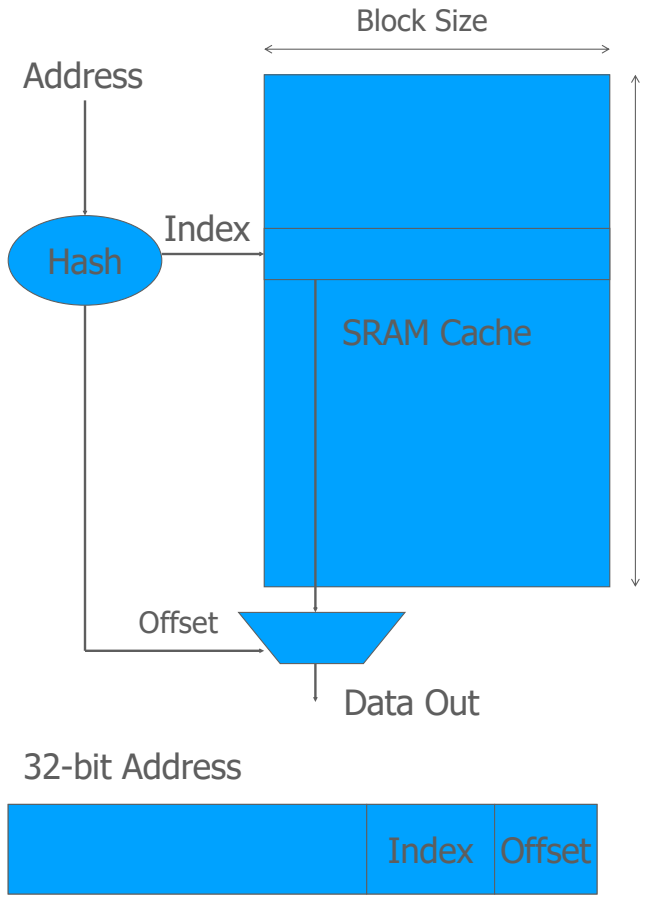


Cache: Towards ideal hardwired storage

Criteria	Reason
Low Hit Time	High performance and speed
Low Miss Rate	Maximizes cache hit rate
Low Miss Penalty	Minimizes memory fetch delay
High Bandwidth	Supports multiple simultaneous requests
Effective Replacement Policy	Improves hit ratio
Efficient Write Handling	Optimizes memory bandwidth usage
Coherence (Multiprocessor)	Ensures data consistency
Energy Efficiency	Reduces power consumption
Scalability	Adapts to diverse workloads
Reliability & Fault Tolerance	Ensures robust operation

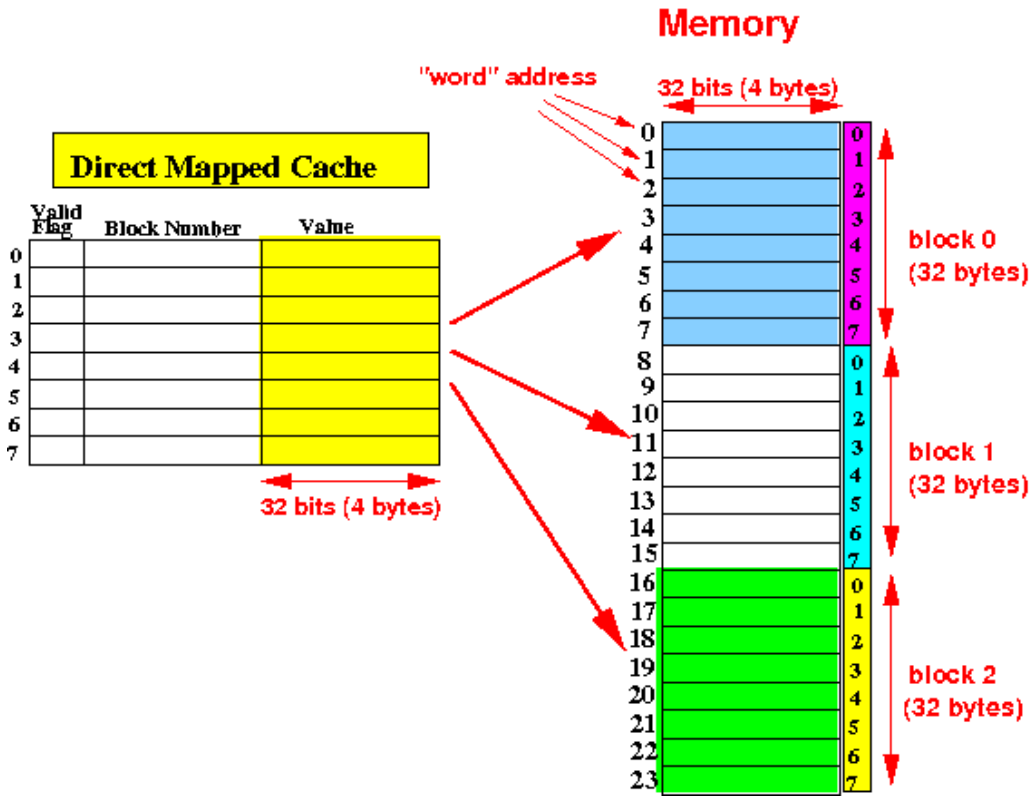
Cache: Placement

- Address Range
  - Exceeds cache capacity
- Map address to finite capacity
  - Called a hash
  - Usually just masks high-order bits
- Direct-mapped
  - Block can only exist in one location
  - Hash collisions cause problems



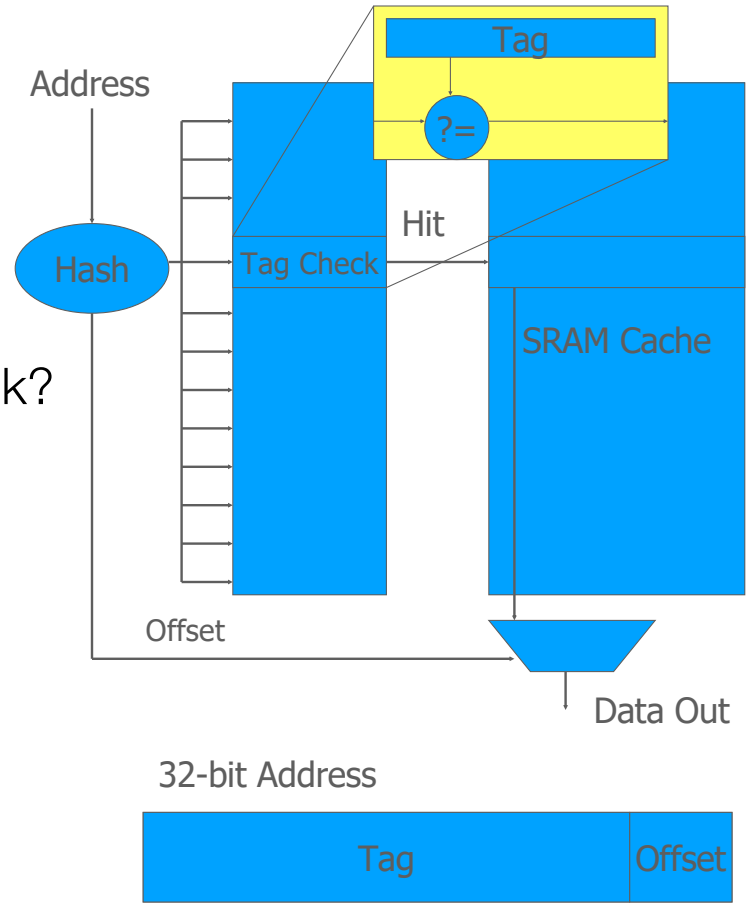
Cache: Placement

- Address Range
  - Exceeds cache capacity
- Map address to finite capacity
  - Called a hash
  - Usually just masks high-order bits
- Direct-mapped
  - Block can only exist in one location
  - Hash collisions cause problems



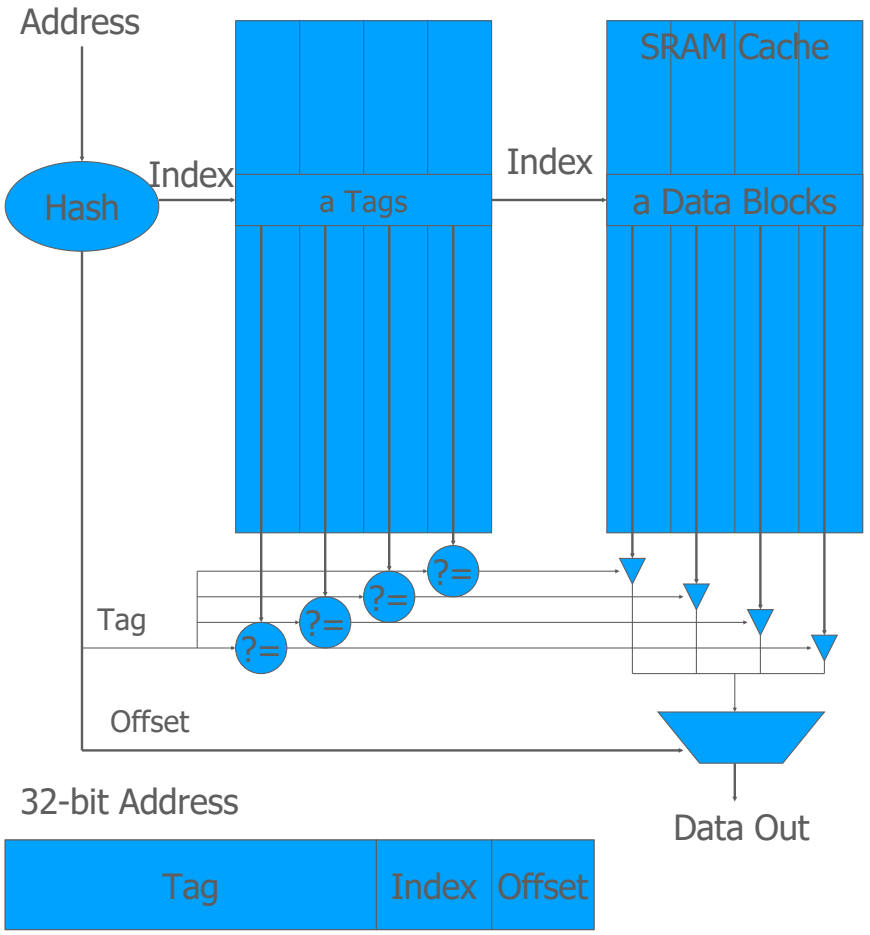
Cache: Placement

- *Fully-associative*
  - Block can exist anywhere
  - No more hash collisions
- *Identification*
  - How do I know I have the right block?
  - Called a *tag check*
    - Must store address tags
    - Compare against address
- Expensive!
  - Tag & comparator per block

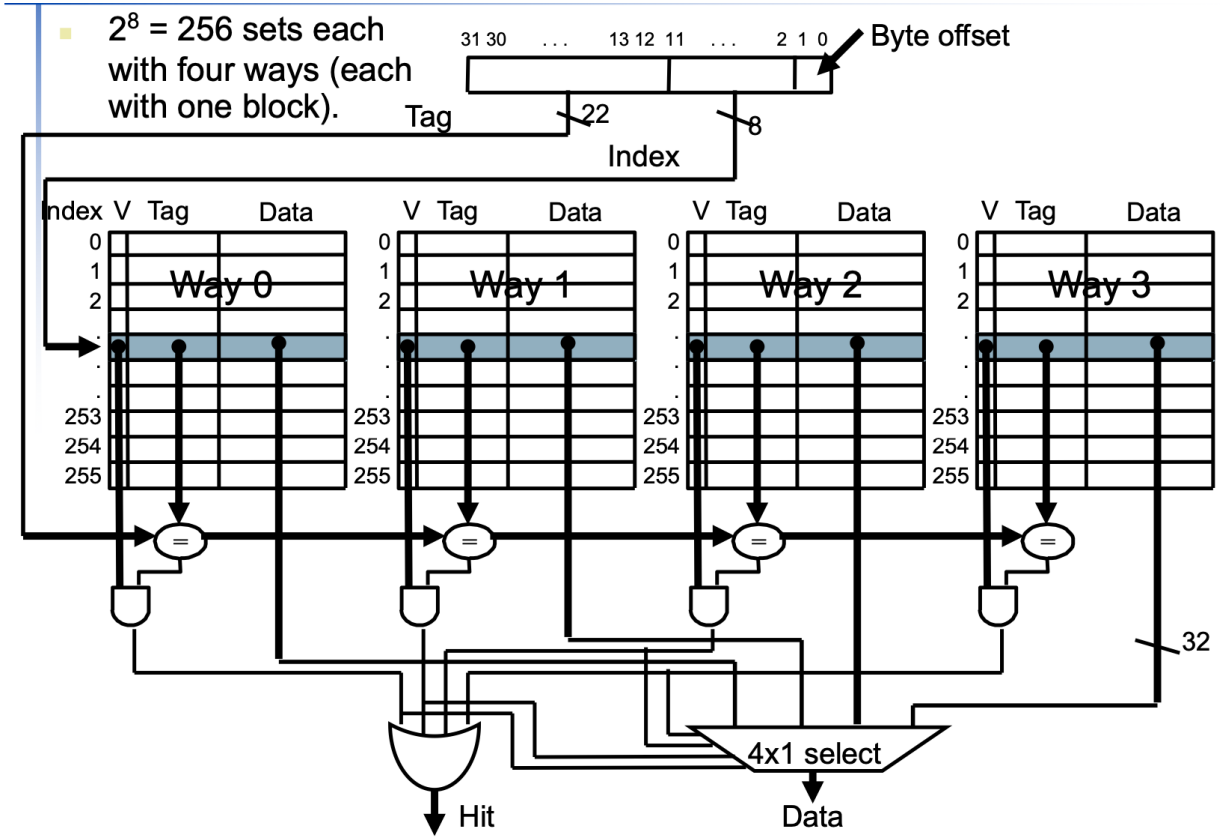


Cache: Placement

- *Set-associative*
  - Block can be in a locations
  - Hash collisions:
    - a still OK
- *Identification*
  - Still perform *tag check*
  - However, only a few in parallel



- 2^8 = 256 sets each with four ways (each with one block).



The formula for the total size of the cache is:

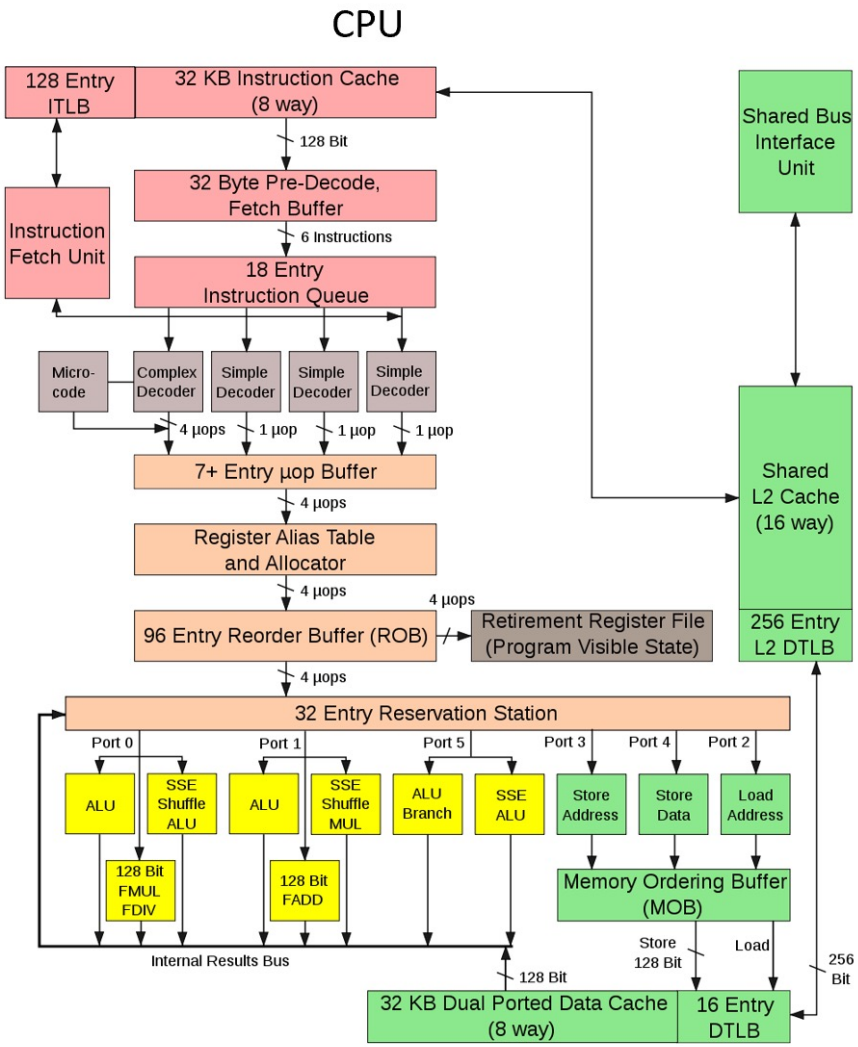
$$\text{Total Size} = BS \times B = BS \times S \times \left(\frac{B}{S}\right)$$

Where:

- **BS** = Block Size (the size of each cache line, usually measured in bytes)
- **B** = Number of blocks (total cache lines in the cache)
- **S** = Number of sets (the number of different locations in the cache where data can be placed)
- **B/S** = The number of blocks per set (this is how many blocks are grouped in each set, e.g., 1 block per set in direct-mapped caches or multiple blocks per set in a set-associative cache)

Summary

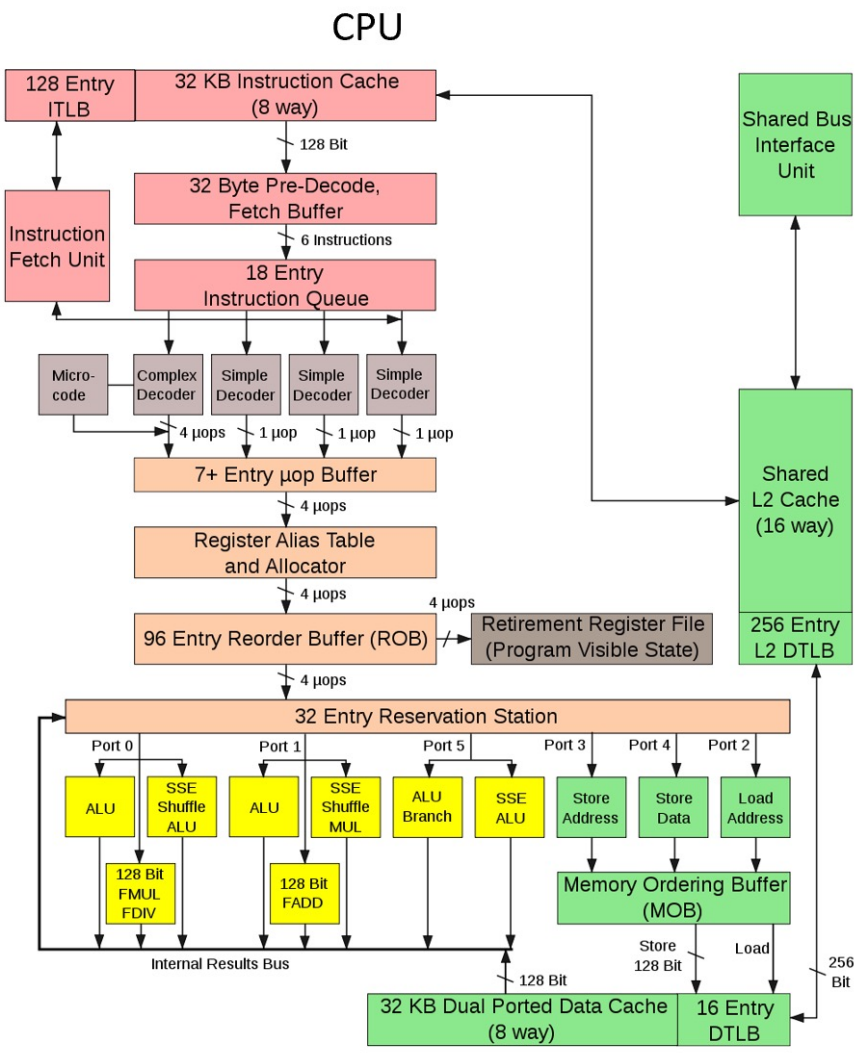
	Processing Side	Data Side
Compiler (Software)	Loop unrolling, vectorization → expose parallel ops	Ensure contiguous arrays, unroll for cache-line use, software prefetch hints
CPU (Hardware)	Out-of-order execution Multiple-issue (superscalar) Branch prediction	Cache hierarchy (L1/L2/L3), hardware prefetchers,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



Intel Core 2 Architecture

⚙️ Overall Pipeline Flow Summary

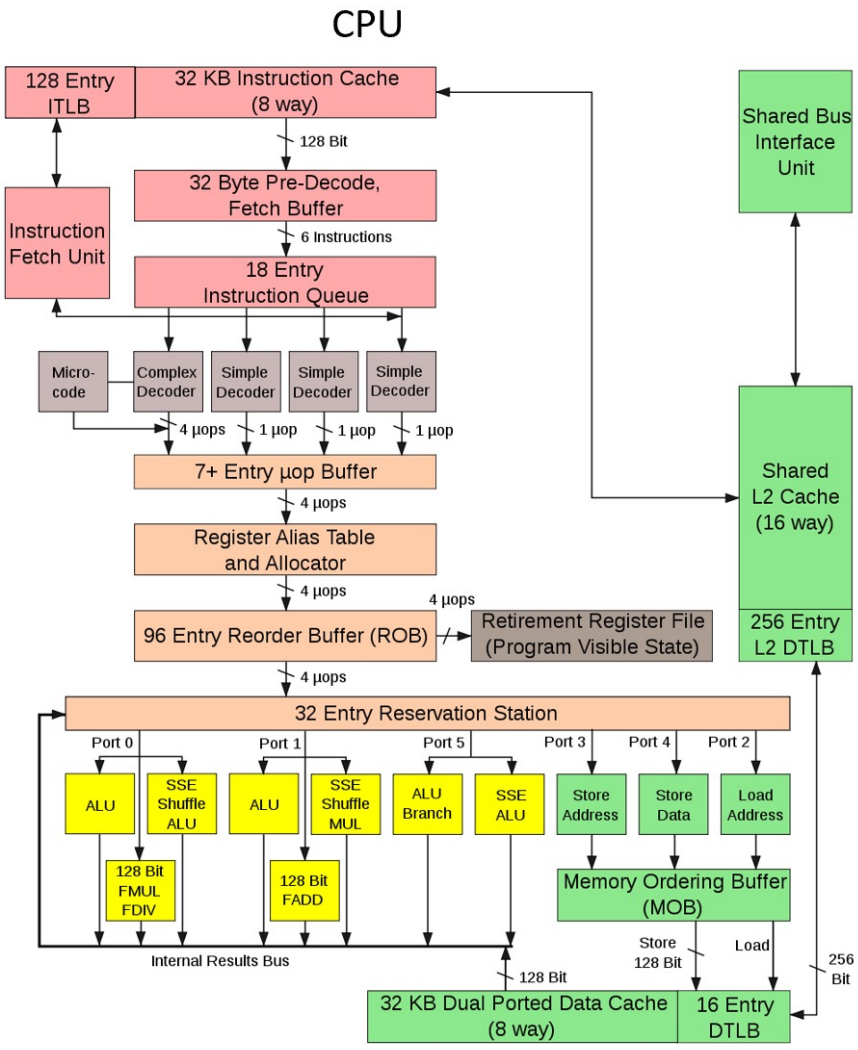
1. **Fetch**: Instructions are fetched from instruction cache (I-Cache) with the help of ITLB.
2. **Pre-decode**: Instructions are pre-decoded and buffered for the decoding stage.
3. **Decode**: Instructions are translated into µops by simple or complex decoders or micro-code ROM.
4. **Rename & Allocate**: 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

🔴 **Instruction Fetch and Decode Stage (Red Boxes)**

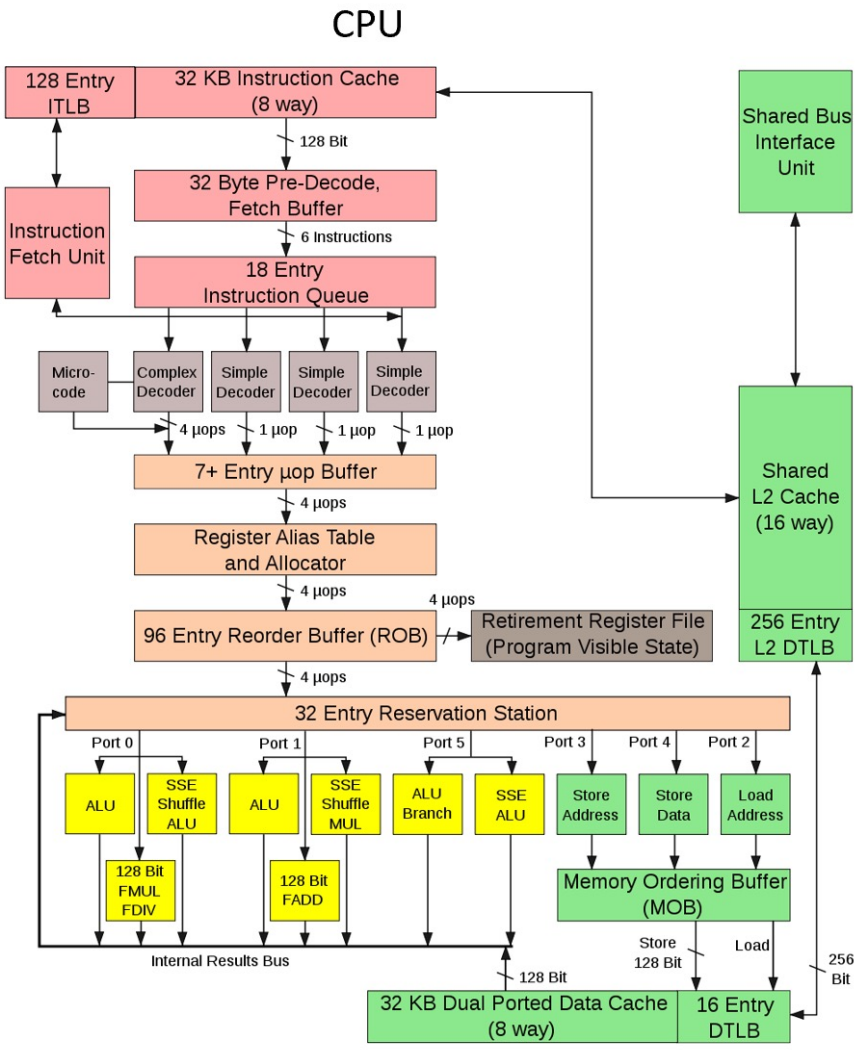
- **Instruction Fetch Unit**:
  - Fetches instructions from memory.
  - Uses the Instruction Translation Lookaside Buffer (**ITLB**, 128 entries) to speed up virtual-to-physical address translation for instructions.
- **32 KB Instruction Cache (I-Cache, 8-way set associative)**:
  - Holds recently accessed instructions to quickly supply the fetch stage, improving performance by reducing memory access latency.
- **32-byte Pre-decode Fetch Buffer**:
  - Stores prefetched instructions (up to 6 instructions).
  - Performs initial decoding/preparation, easing decoding complexity downstream.
- **18-entry Instruction Queue**:
  - Buffers decoded instructions to manage pipeline flow and maintain throughput.
- **Decoders (Simple & Complex decoders + Micro-code ROM)**:
  - **Simple Decoders** handle common, straightforward instructions, translating them into single micro-operations (µops).
  - **Complex Decoder** handles more intricate instructions, decoding them into multiple µops.
  - **Micro-code ROM** handles special or extremely complex instructions by translating them into micro-code sequences.



Intel Core 2 Architecture

Micro-operation Handling Stage (Orange Boxes)

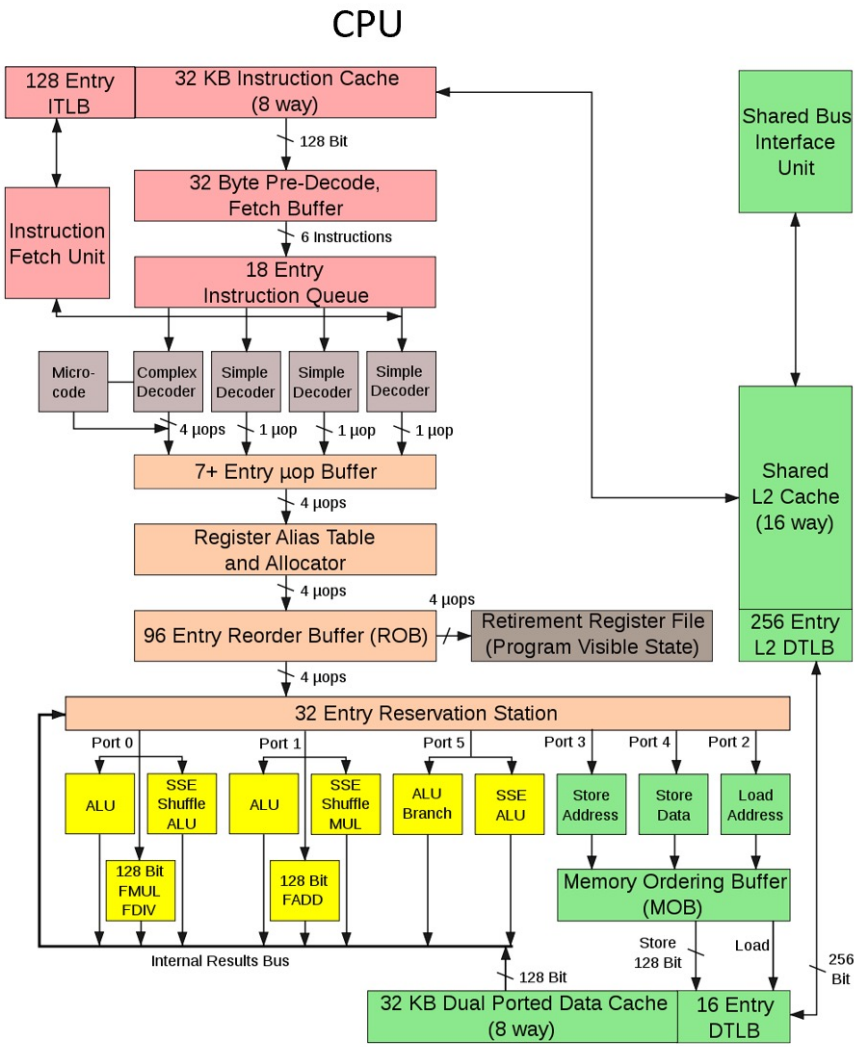
- **Micro-operation (µop) Buffer (7+ entries)**:
  - Temporarily stores decoded micro-operations before they enter the execution pipeline.
- **Register Alias Table (RAT) and Allocator**:
  - Implements Register Renaming, mapping architectural registers to physical registers.
  - Eliminates false dependencies (WAR and WAW hazards) by dynamically renaming registers, allowing parallel execution.
- **96-entry Reorder Buffer (ROB)**:
  - Facilitates out-of-order execution and precise exceptions.
  - Tracks micro-operations, allowing the CPU to commit instructions in program order, ensuring correct execution semantics.
- **Retirement Register File (Program Visible State)**:
  - Maintains the committed state visible to software (architectural state), updated as instructions retire from the ROB.



Intel Core 2 Architecture

🟡 Execution Stage (Yellow Boxes)

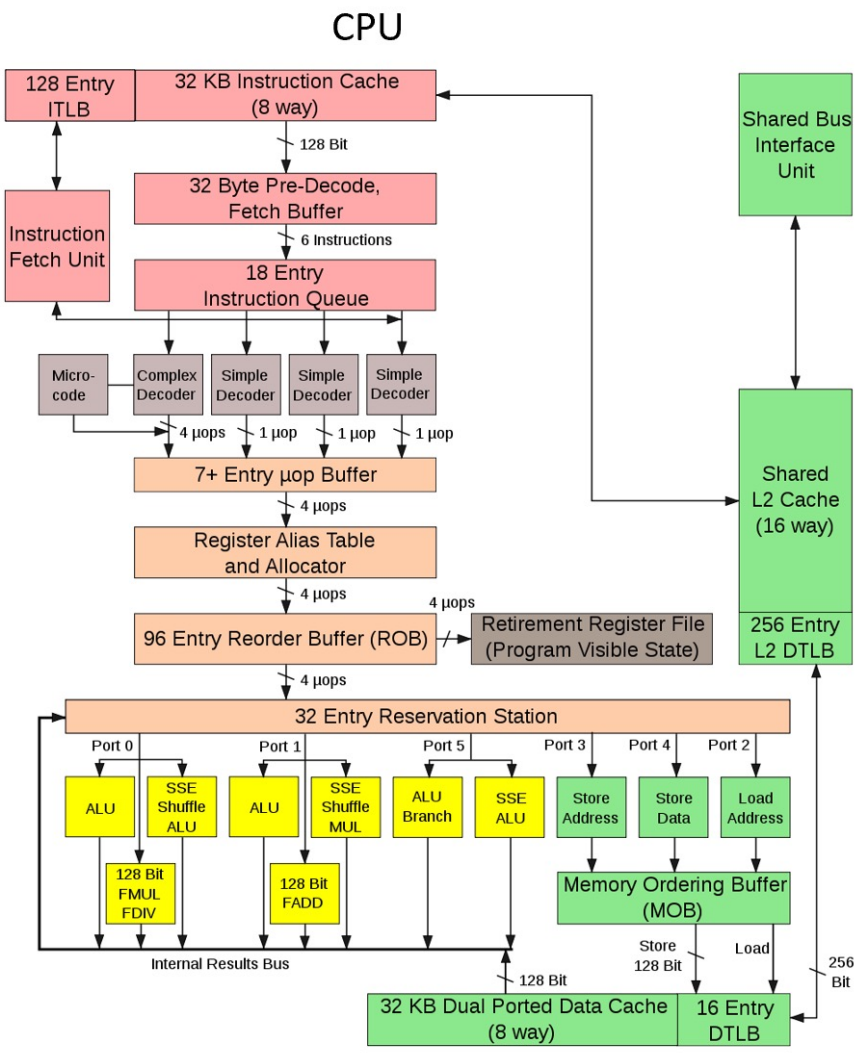
- **32-entry Reservation Station**:
  - Buffers µops until operands and execution resources become available.
  - Dispatches ready µops to available execution units.
- **Execution Units** (multiple functional units):
  - **ALUs (Arithmetic Logic Units)**: Perform integer arithmetic and logic operations.
  - **SSE/Vector Units (Shuffle, MUL, ADD, FMUL, FDIV)**: Handle floating-point and SIMD instructions.
  - **Branch Unit (ALU Branch)**: Evaluates branch conditions and computes target addresses.
- Each execution unit receives µops from specific ports (Port 0, Port 1, Port 5, etc.), enabling parallel execution of independent instructions.
- Results from execution units are communicated via the **Internal Results Bus** back to the reservation station, ROB, and other pipeline elements.



Intel Core 2 Architecture

🟢 Memory Access and Data Handling (Green Boxes)

- **Memory Ordering Buffer (MOB)**:
  - Ensures correct memory ordering semantics for load/store operations, preserving memory consistency.
  - Coordinates execution of load/store operations in the correct order.
- **32 KB Dual-Ported Data Cache (D-Cache, 8-way set associative)**:
  - Holds recently accessed data to reduce latency of data memory accesses.
  - "Dual ported" allows simultaneous load/store operations, improving memory throughput.
- **Data Translation Lookaside Buffer (DTLB, 16 entries)**:
  - Speeds up virtual-to-physical address translation for data accesses.
- **Shared L2 Cache (16-way set associative)**:
  - Provides a unified second-level cache for both instruction and data.
  - Interfaces with the external memory system through the **Shared Bus Interface Unit**.

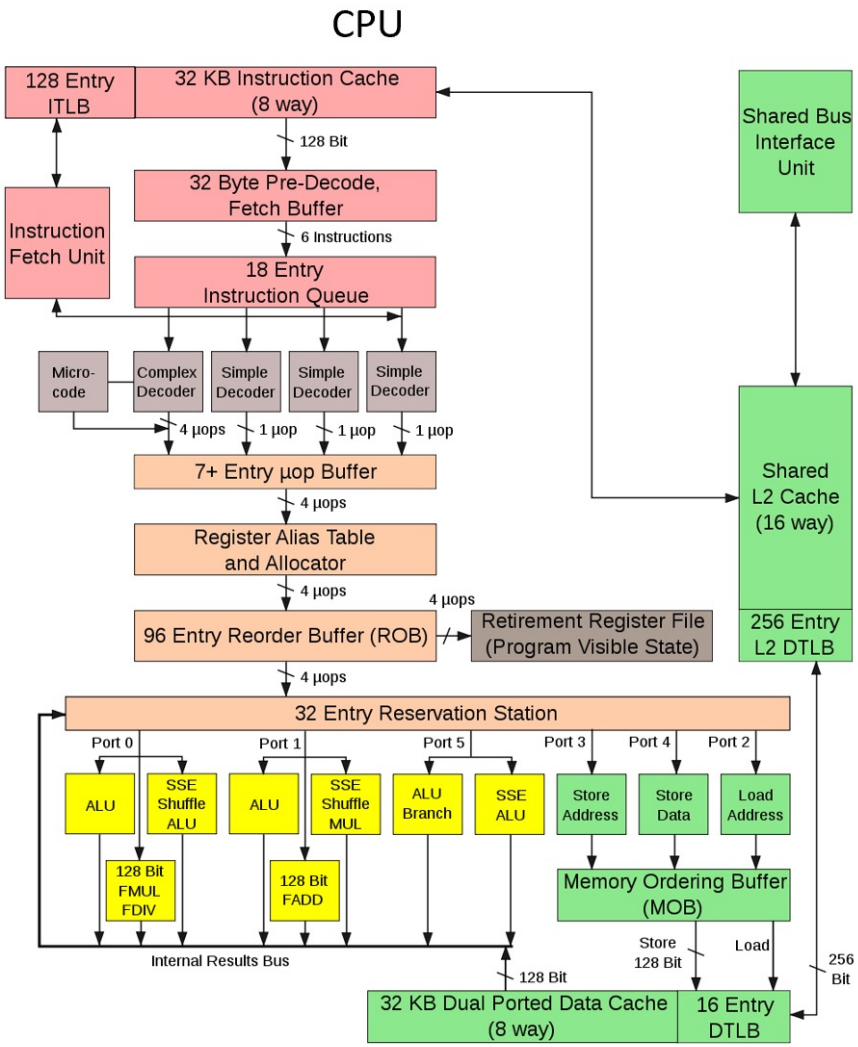


Intel Core 2 Architecture

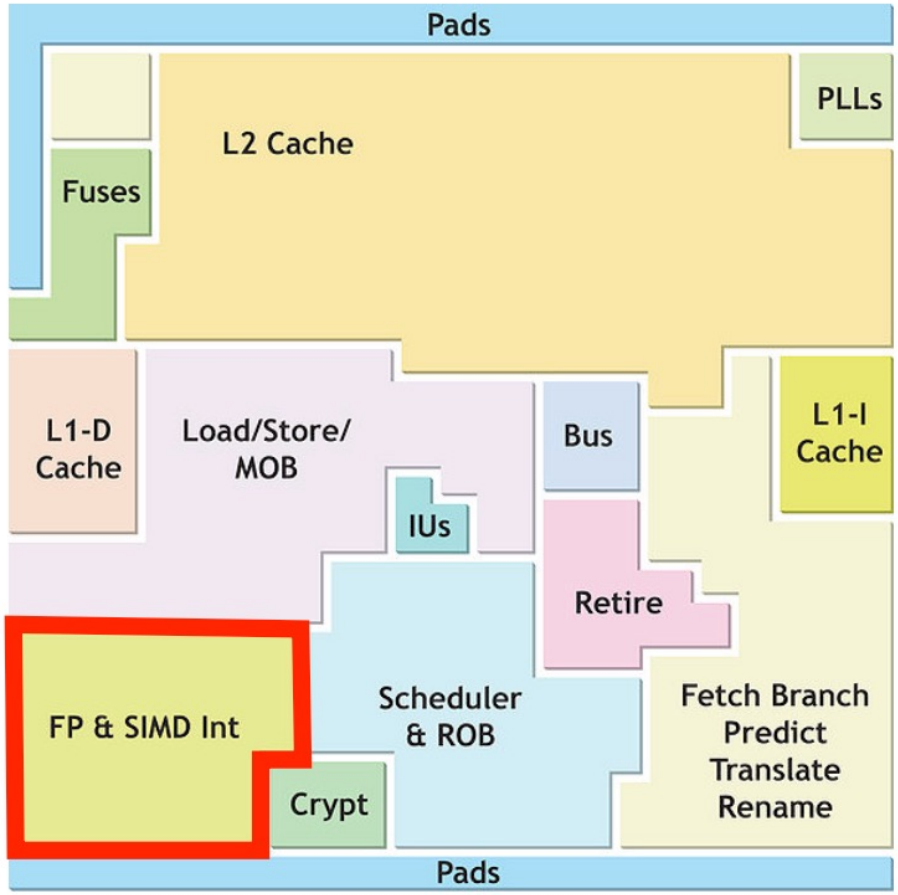
📌 Key Design Features

- **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



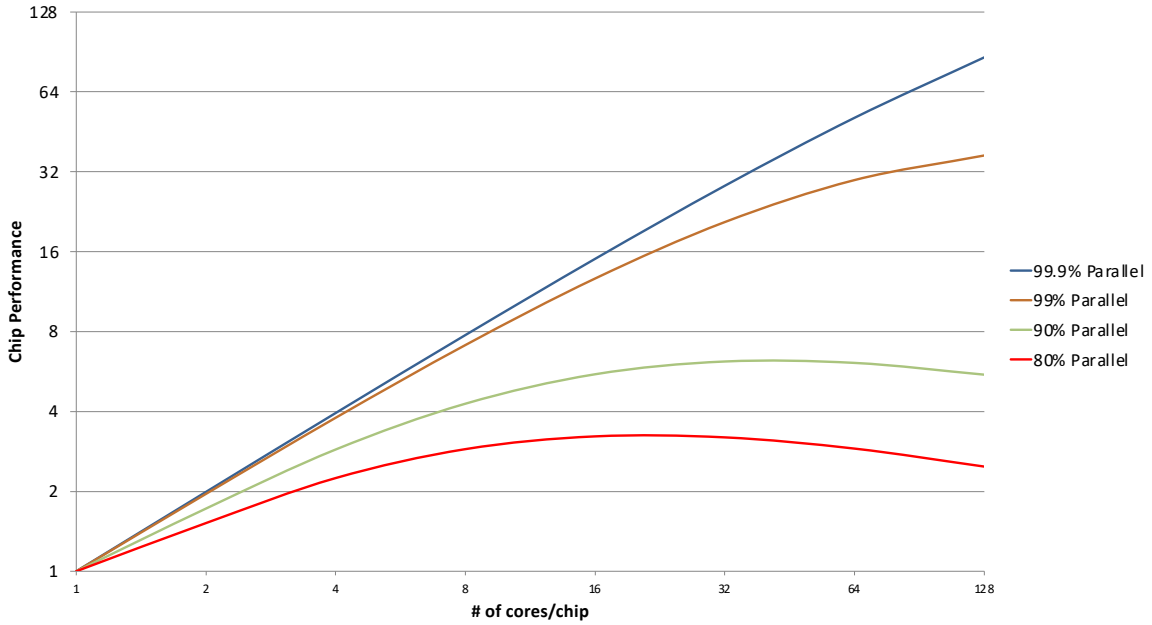
Why Multicore

- Instruction-level parallelism
  - Reaps performance by finding independent work in a single thread
- Thread-level parallelism
  - Reaps performance by finding independent work across multiple threads
- Historically, requires explicitly parallel workloads
  - Originate from mainframe time-sharing workloads
  - Even then, CPU speed >> I/O speed
  - Had to overlap I/O latency with “something else” for the CPU to do
  - Hence, operating system would schedule other tasks/processes/threads that were “time-sharing” the CPU

Why Multicore

$$Speedup = \frac{1}{(1-f) + \frac{f}{n}}$$

$$\lim_{n \to \infty} \frac{1}{1-f+\frac{f}{n}} = \frac{1}{1-f}$$



- Fixed power budget forces slow cores
- Serial code quickly dominates

Matrix Multiplication

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

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

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

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

Matrix Multiplication

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    for (k=0; k<n; k++)
      C[i][j] += A[i][k] * B[k][j];
```

Matrix Multiplication

(1) Cache Blocking / Loop Tiling

- **Divide the matrices into blocks**, so that the computation can be completed as much as possible within the **L1/L2 cache**, reducing repeated loads from main memory.
- **Idea:** Split the large matrix into small blocks of size $B \times B$.
  Convert the original **three nested loops** into **block-nested loops**.
- **Benefit:** Exploits **spatial and temporal locality of data**.

Matrix Multiplication

(2) Loop Unrolling

- **Unroll the loop body**, allowing the CPU to execute **multiple instructions per iteration**, thereby **reducing branch overhead**.
- **Introduce multiple accumulators simultaneously** to eliminate instruction dependencies and **increase instruction-level parallelism (ILP)**.

Matrix Multiplication

(3) SIMD Vectorization (Vectorization)

- **Use the CPU's SIMD instruction set** (such as **AVX / AVX-512**) to perform **multiply-add operations on multiple floating-point numbers simultaneously**.
- **For example**, AVX-512 can execute **16 single-precision floating-point FMA (Fused Multiply-Add) operations in one instruction**.
- **Typical implementation:** use instructions such as **_mm256_fmadd_ps** inside the inner loop.

Matrix Multiplication

(4) Multicore Parallelism (Multithreading)

- **Different threads process different blocks of the matrix.**
- **OpenMP / TBB** can be used to **quickly parallelize the outer loops**.
- **For example**, divide matrix C **by rows or by blocks** and assign them to **different CPU cores**.

Matrix Multiplication

```
// Blocked + vectorized GEMM (伪代码)
for (ii = 0; ii < n; ii += B)             // ← 外层按 B×B 分块：把 A、B、C 划成能装进 L2/L1 的小块
  for (jj = 0; jj < n; jj += B)           //    这样同一块数据被多次复用，减少反复从内存拿数据（时间/空间局部性）
    for (kk = 0; kk < n; kk += B)         //    三重分块(ii,jj,kk) ≈ 经典 cache blocking（也叫 loop tiling）

      for (i = ii; i < ii + B; i++)       // —— 下面在一个 B×B 子块上做“小GEMM”
        for (j = jj; j < jj + B; j++) {
          __m256 c_vec = _mm256_setzero_ps();           // ← SIMD 向量累加器：8×float 并行累加（AVX 256-bit）

          for (k = kk; k < kk + B; k += 8) {            // ← 内层步长=8：一次处理 8 个乘加，匹配 AVX 向量宽度
              __m256 a_vec = _mm256_loadu_ps(&A[i][k]); //    向量加载：取 A[i,k..k+7]
              __m256 b_vec = _mm256_loadu_ps(&B[k][j]); //     （示例直取一列不太贴近真实高效实现，真实代码多用
                                                        //      pack/转置/寄存器重排或 micro-kernel 以提升 B 的访存局部性）
              c_vec = _mm256_fmadd_ps(a_vec, b_vec, c_vec); //  FMA：c_vec += a_vec * b_vec（8 路并行，指令级并行/SIMD）
          }

          C[i][j] += horizontal_add(c_vec);             // ← 把 8 路向量累加成标量写回（水平求和）
        }
```

What are the lessons learned?

–Tom Jerry

GPU Architecture for Machine Learning

Workload is the king, as always

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

Modern GPU

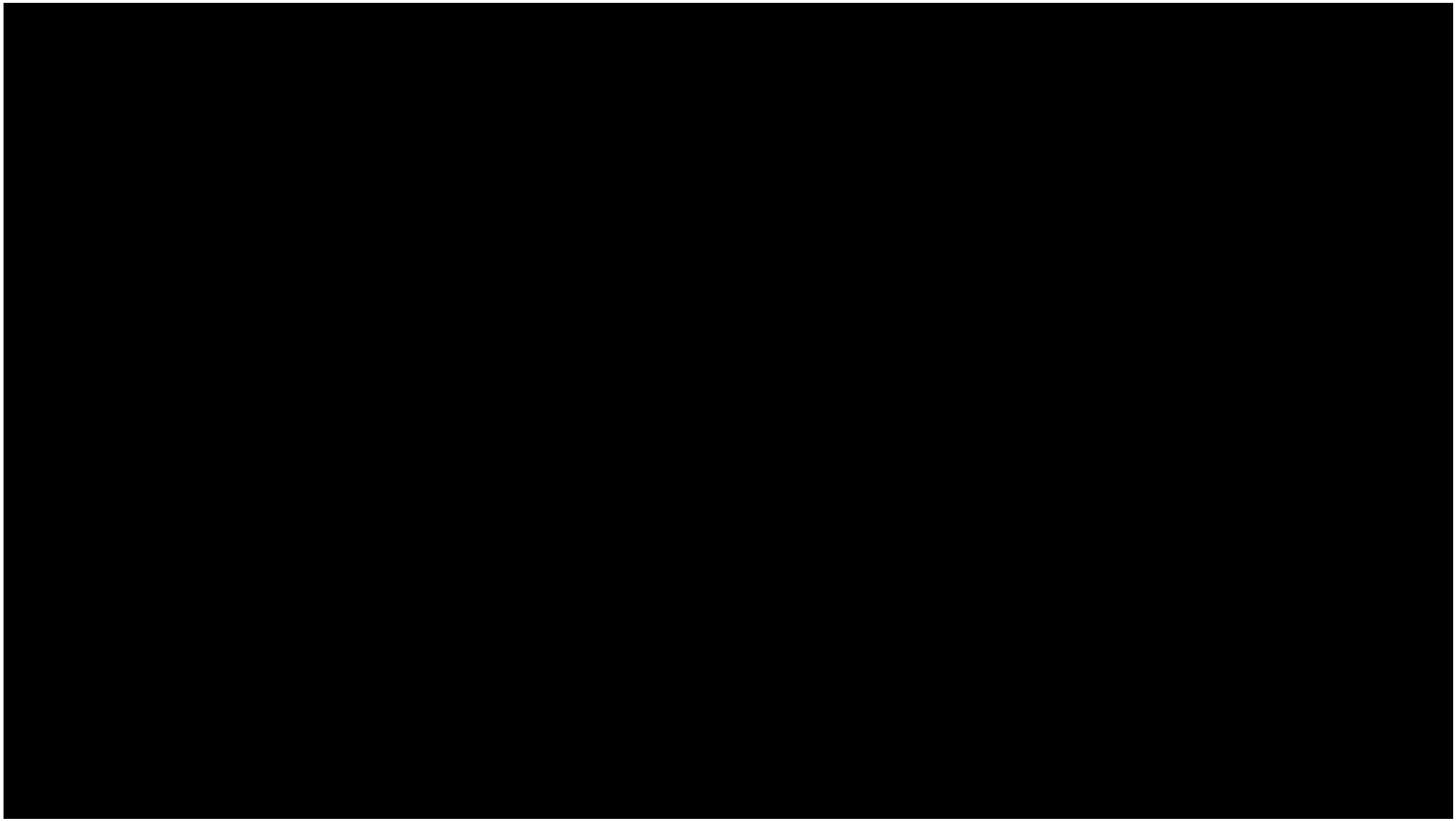
Objective：Flexible, programming graphics and high-performance computing

Architecture：Unified graphics and parallel computing architecture

Scalability：Parallel array of processors are massively multithreaded

Programming Flexibility：CUDA programming model for high-performance GPGPU programming

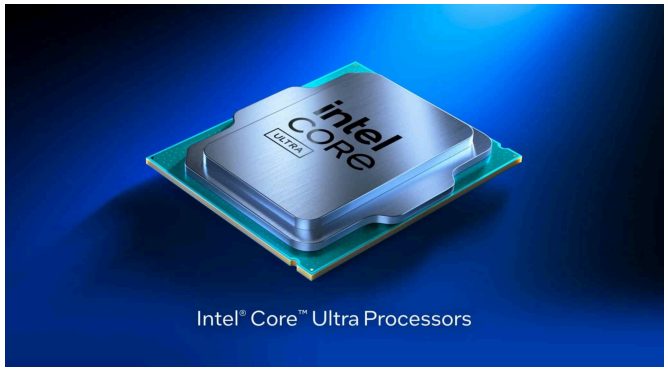
Eric Lindholm, et al., "Nvidia Tesla: A unified graphics and computing architecture, IEEE Micro, 2008



CPU vs. GPU

The GeForce RTX 5090 is powered by the NVIDIA Blackwell architecture, and is estimated to have a memory bandwidth of **1,920 GB/s**.

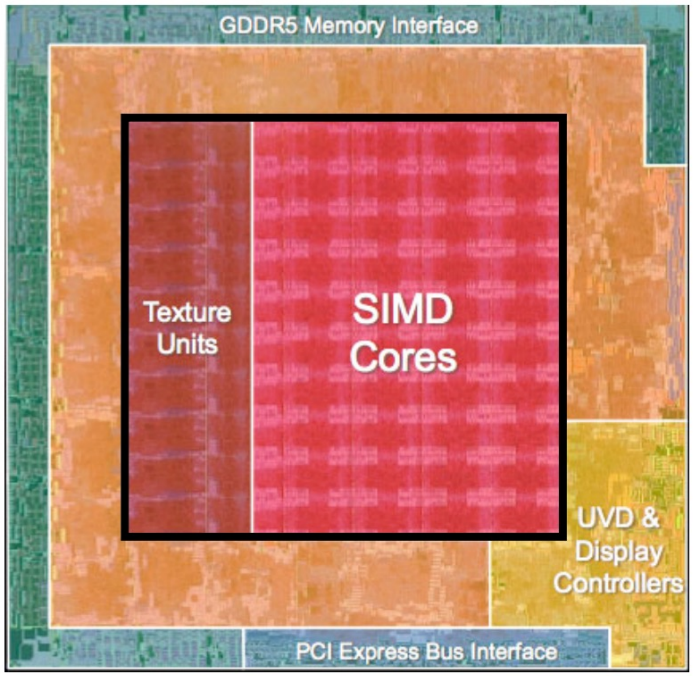
100+ TFLOPs Shader, ray tracing, and tensor operations

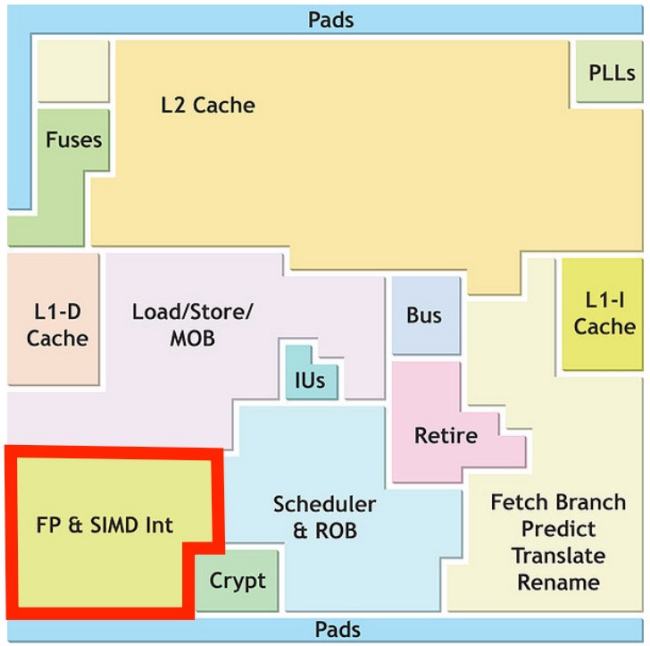


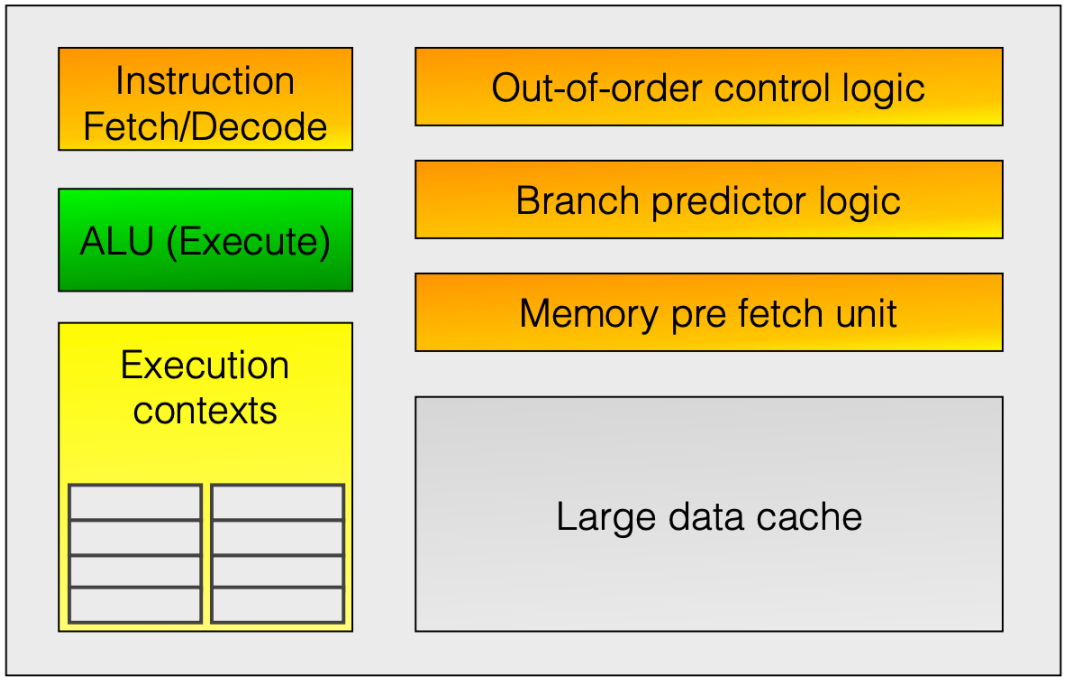
The Intel Core i9-15900K is a 15th generation processor, also known as the Intel Core Ultra 9 285K. It supports DDR5-8400 RAM in a dual-channel configuration, yielding a system memory bandwidth of about **134.4 GB/s**.

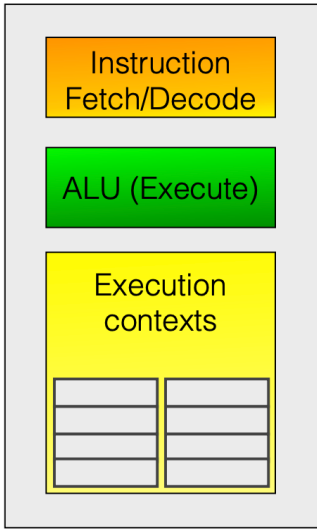
Thread-level and ILP optimizations

CPU vs. GPU

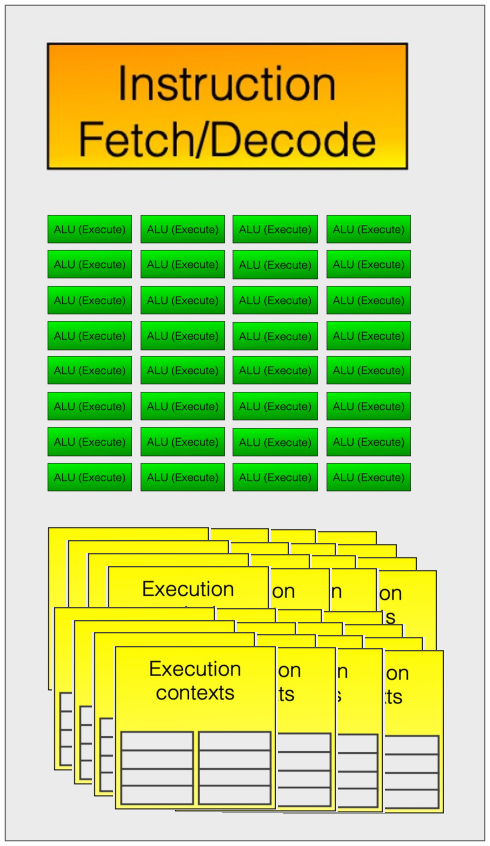

GDDR5 Memory Interface
Texture Units
SIMD Cores
UVD & Display Controllers
PCI Express Bus Interface


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

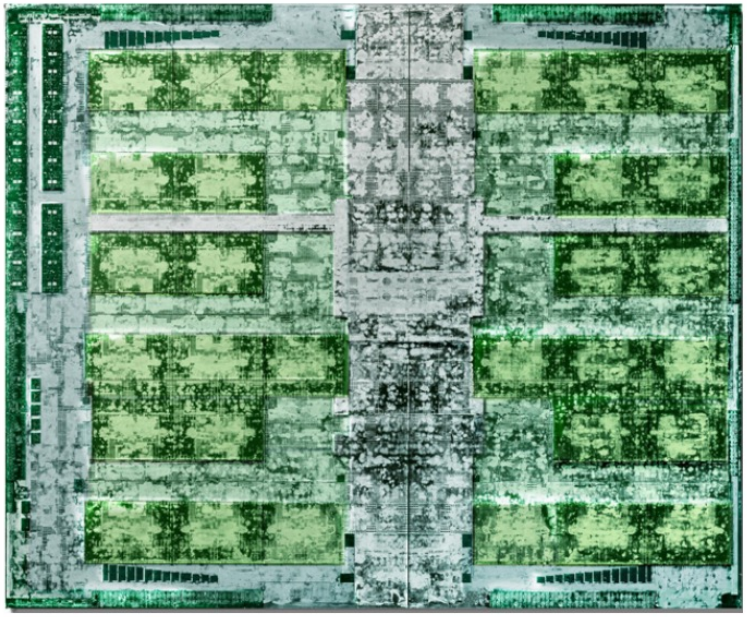

Instruction Fetch/Decode
ALU (Execute)
Execution contexts
Out-of-order control logic
Branch predictor logic
Memory pre fetch unit
Large data cache


Instruction
Fetch/Decode
ALU (Execute)
Execution
contexts

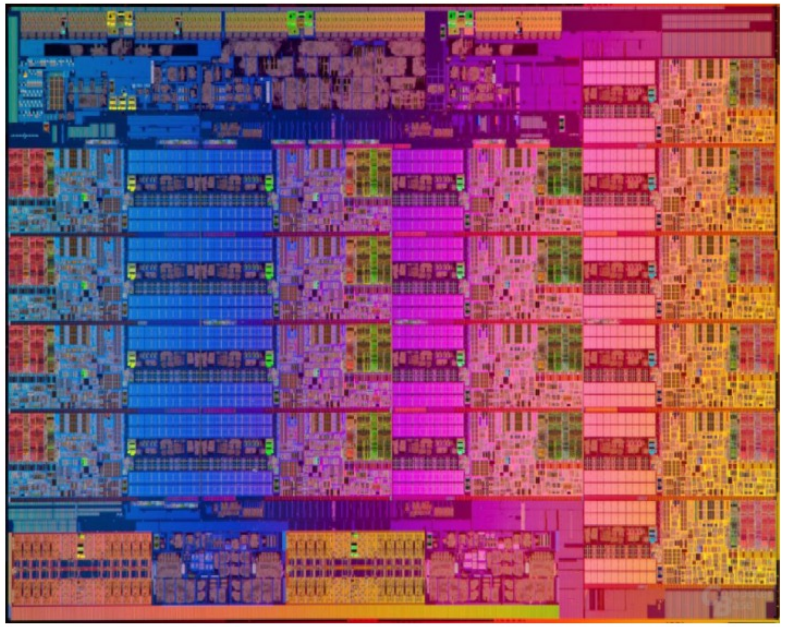

Instruction Fetch/Decode
ALU (Execute)
Execution contexts


Instruction Fetch/Decode
ALU (Execute)
Execution contexts


Instruction Fetch/Decode
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

GPU Architecture

- GPU: From computer graphics to deep learning
- SIMT: single instruction multiple thread computing
- Memory hierarchy: Global/shared/register storage
- Advanced GPU Features and Future Directions

A 30-year Journey

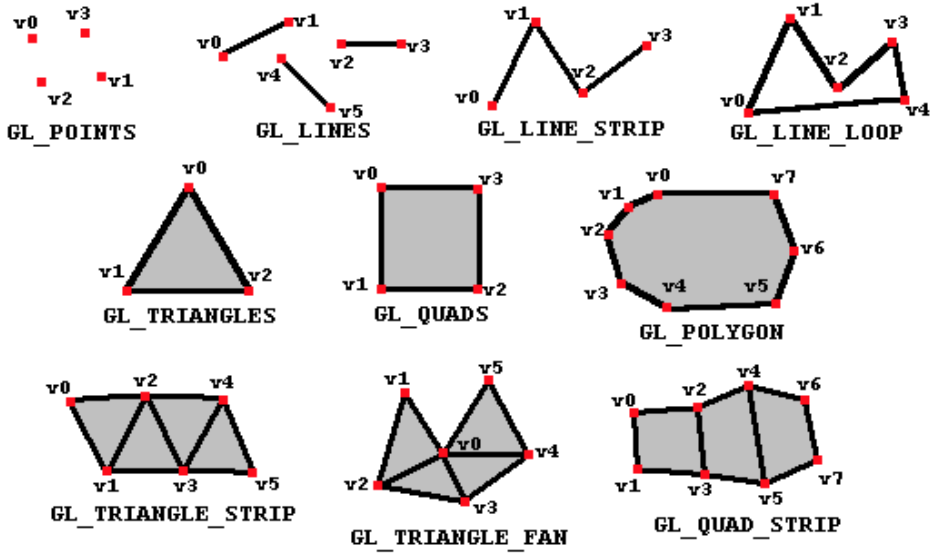
- **1993:** NVIDIA is founded by Jensen Huang, Chris Malachowsky, and Curtis Priem, focusing on graphics processing technology.
- **1997:** The company gains prominence in the gaming industry with the release of the RIVA series of graphics processors.
- **1999:** NVIDIA introduces the term “GPU” (Graphics 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.
- **2024:** The company’s market capitalization surpasses $3 trillion, 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.

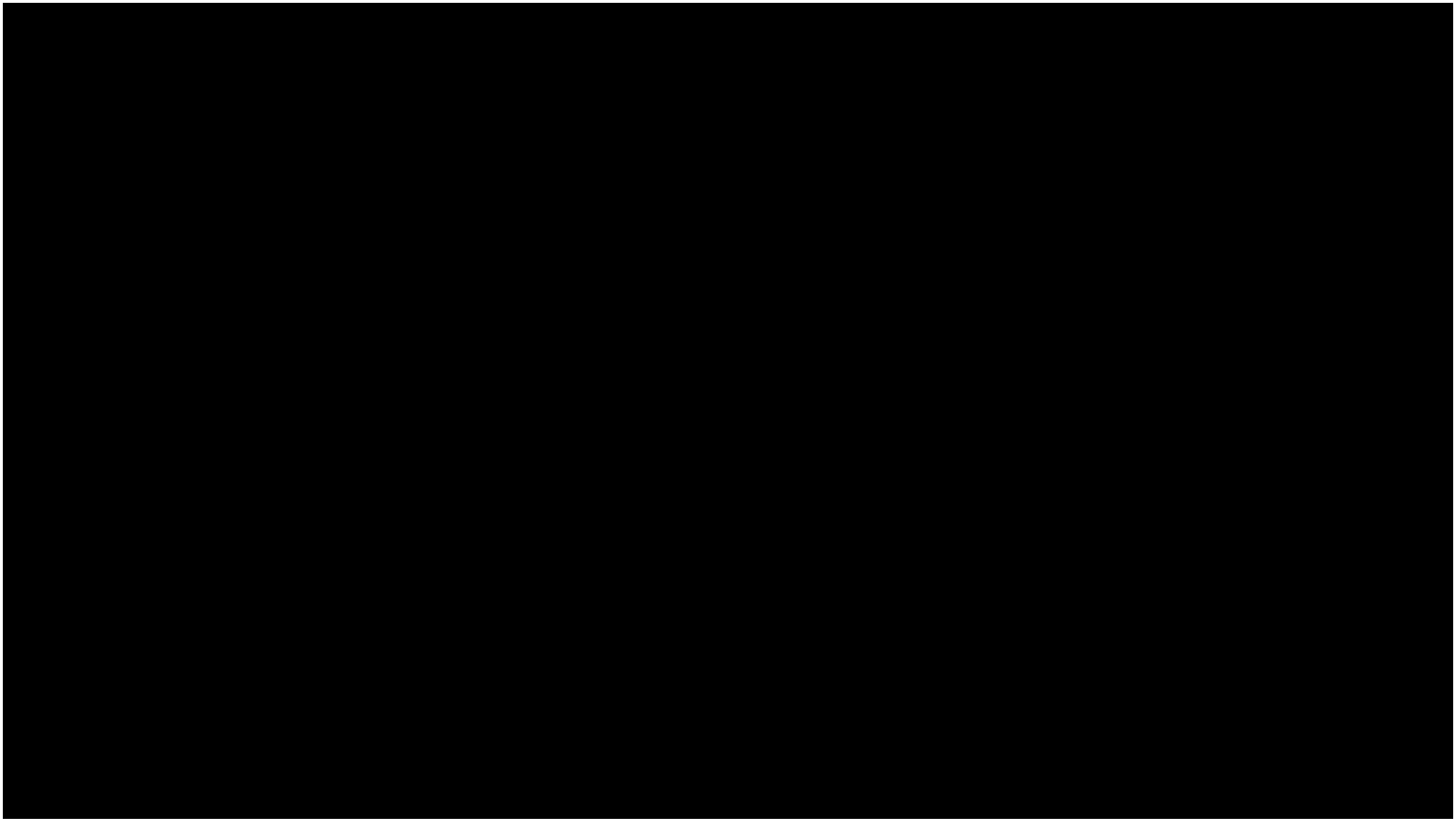
Historical question: why is quadratic texture mapping bad?




CASPA華美半導體協會
CASPA
since 1991
CHINESE AMERICAN SEMICONDUCTOR PROFESSIONAL ASSOCIATION
CASPA
CASPA

Every great technology has a humble origin

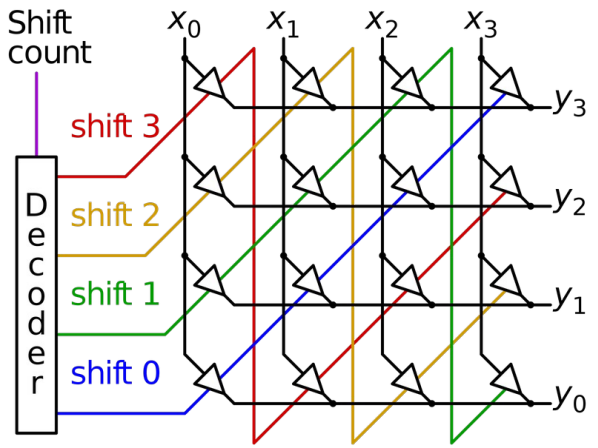
–Tom Jerry



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

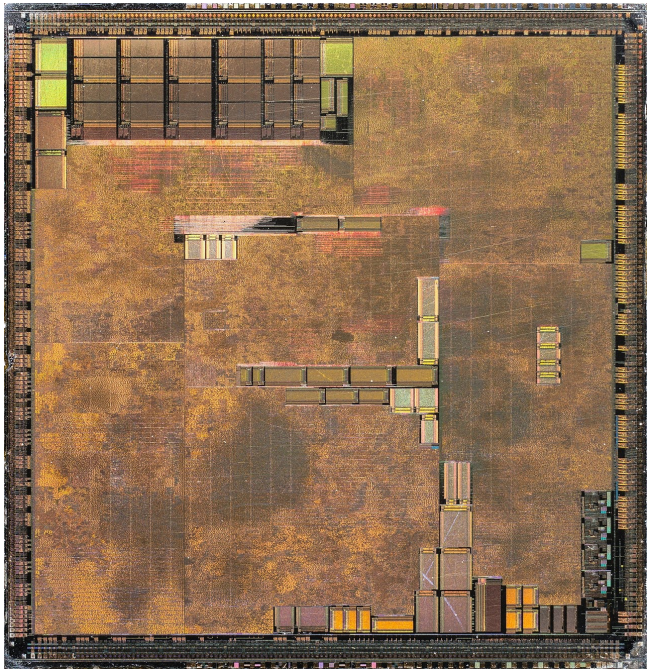
- GPU evolution: From fixed-function accelerator to programmable processor.
- Early adoption in machine learning: The massive parallel nature suitable for linear algebra.
- GPU in the deep learning : Architecture evolution driven by rapid growth of deep learning.

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)

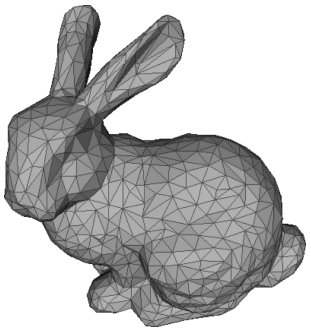
Phase 1: Hardwired GPU with fixed function pipeline

- **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).

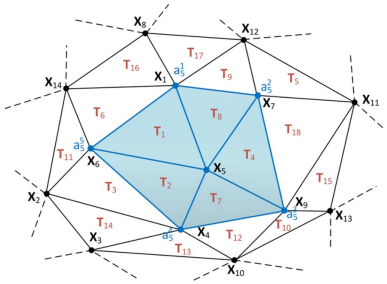
A single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second.

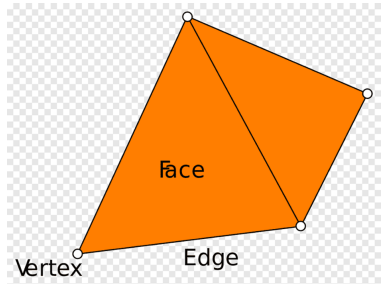


GPU is a Massive Shader



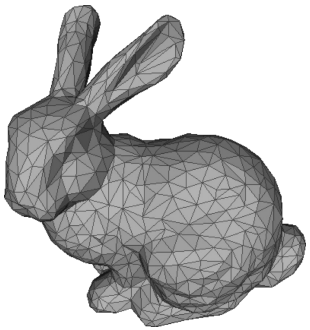
1. **3D Model Representation**:
   - A complex 3D model, such as a human figure, is represented in a digital environment.
2. **Mesh Generation**:
   - The surface of the 3D model is divided into numerous small triangles, forming a mesh.
   - Each triangle consists of three vertices, which are points 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.

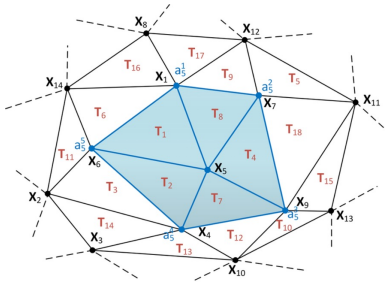


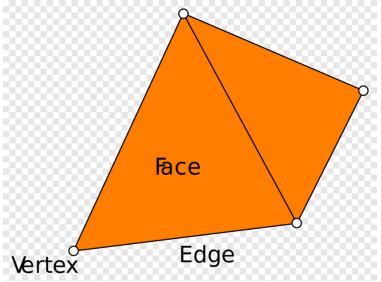


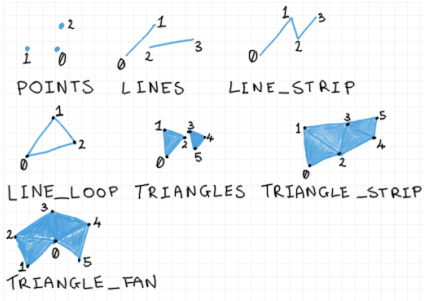
Vertex

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.





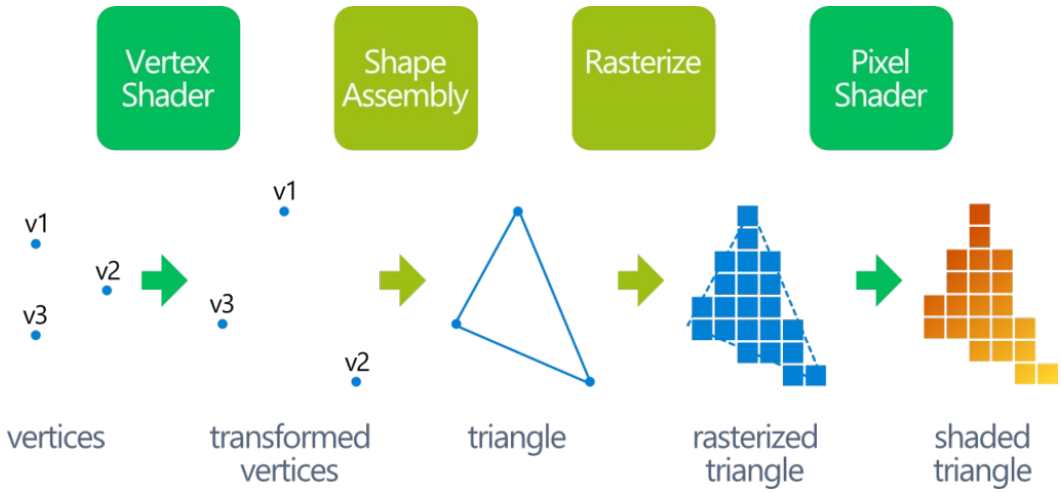




GPU is a Massive Shader

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

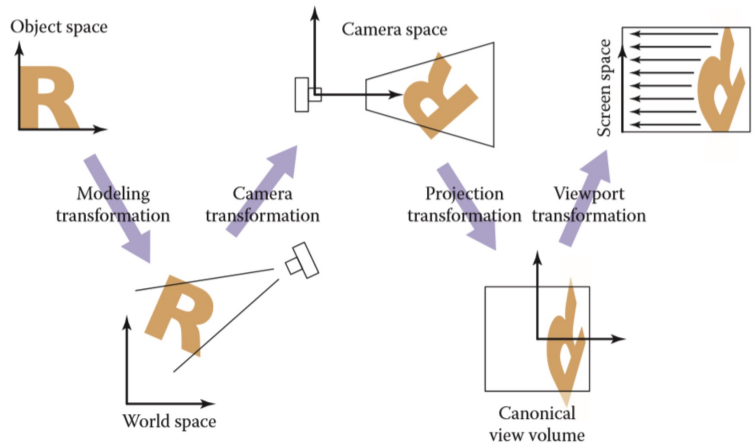
- **Vertex Transformation Stage:** Transforms individual triangle vertices from 3D world space to 2D screen space.
- **Shape Assembly Stage:** A fixed-function stage that assembles transformed vertices into geometric shapes, typically triangles.
- **Rasterization Stage:** Another fixed-function stage 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

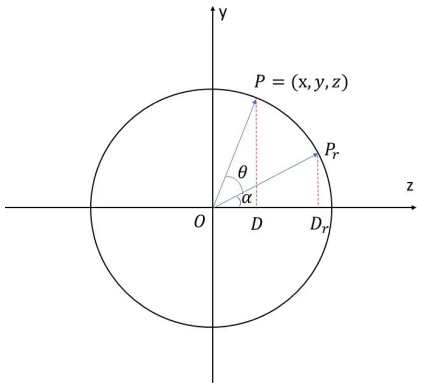
To display objects from three-dimensional space on a two-dimensional plane, they must undergo the MVP transformation. The MVP transformation refers to three stages: Model transformation (Model), View transformation (View), and Projection transformation (Projection).

- Posing → Model Transformation (Model)
- Adjusting the camera position → View Transformation (View)
- Taking the photo → Projection Transformation (Projection)



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.**

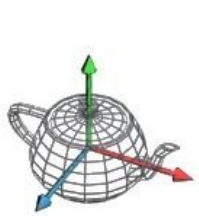


$$Pr = M\theta \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad \begin{bmatrix} x \\ \sin\alpha \\ \cos\alpha \\ 1 \end{bmatrix} = M\theta \cdot \begin{bmatrix} x \\ \sin(\alpha+\theta) \\ \cos(\alpha+\theta) \\ 1 \end{bmatrix}$$

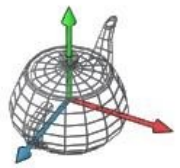
$$y = \sin(\theta+\alpha) = \sin\theta\cos\alpha + \cos\theta\sin\alpha$$
$$z = \cos(\theta+\alpha) = \cos\theta\cos\alpha - \sin\theta\sin\alpha$$

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

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



Rotation 90° around Y



Translate along X

$$Translate = \begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad Scale = \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

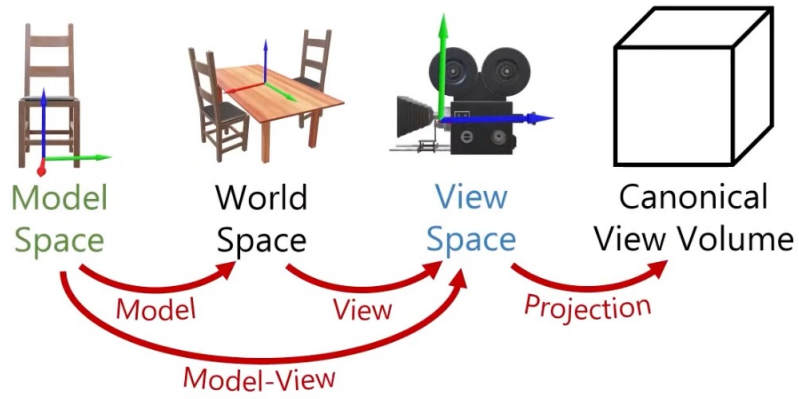
$$M\theta z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

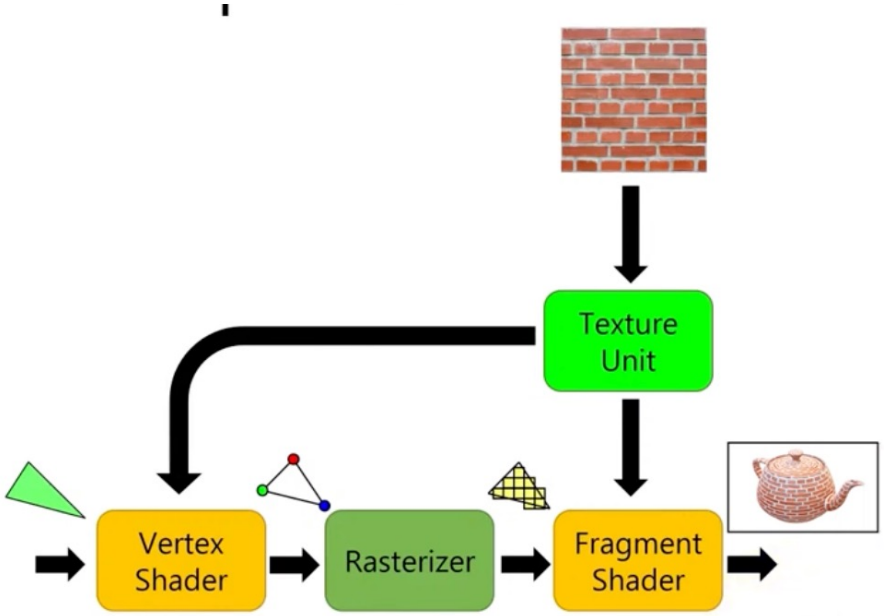
$$M\theta y = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 0 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

GPU is a Massive Shader

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





GPU is a Massive Shader

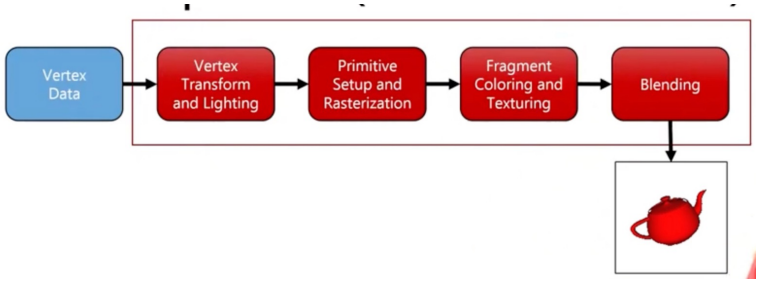
1. **Vertex Shader**: Processes individual vertices, transforming them from object space to clip space (e.g., applying world, view, and projection transformations).
2. **Tessellation Stage** (Optional): Subdivides geometry into smaller primitives (as explained in a previous response).
3. **Geometry Shader**: 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.

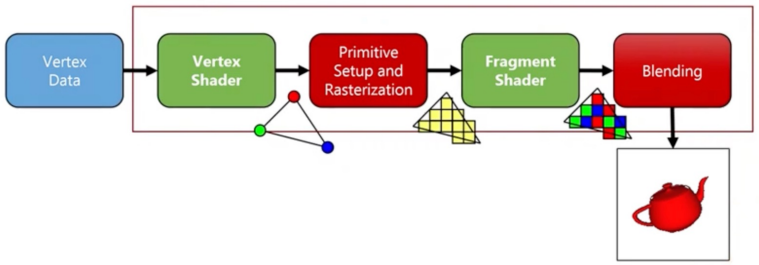
Software-Programmable GPU

- **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

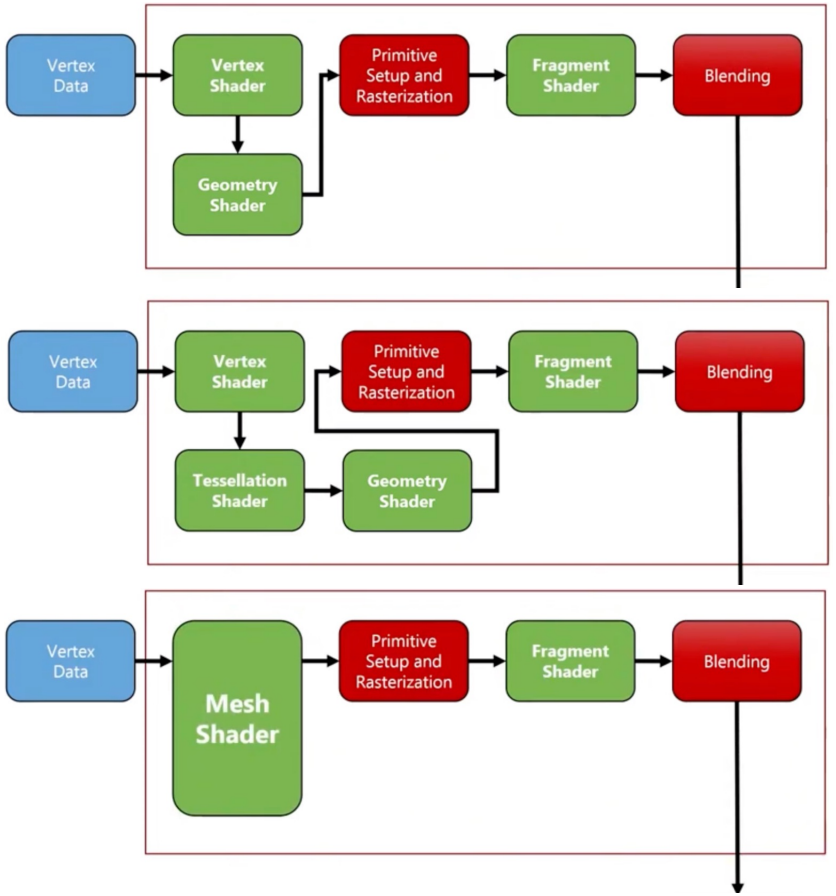




In the **second phase** of GPU evolution, characterized by the introduction of **programmable shaders**, several notable GPUs exemplify this transition:

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.

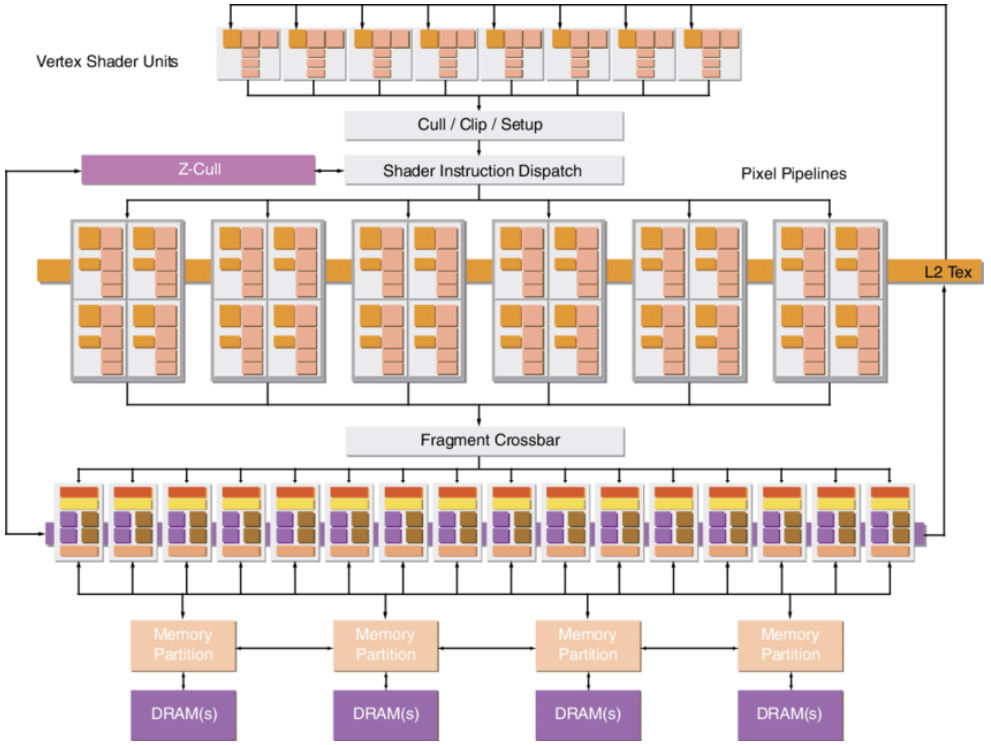
- Coordinate transformations (model → world → screen space)
- Per-vertex lighting calculations
- Vertex morphing and animation

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)

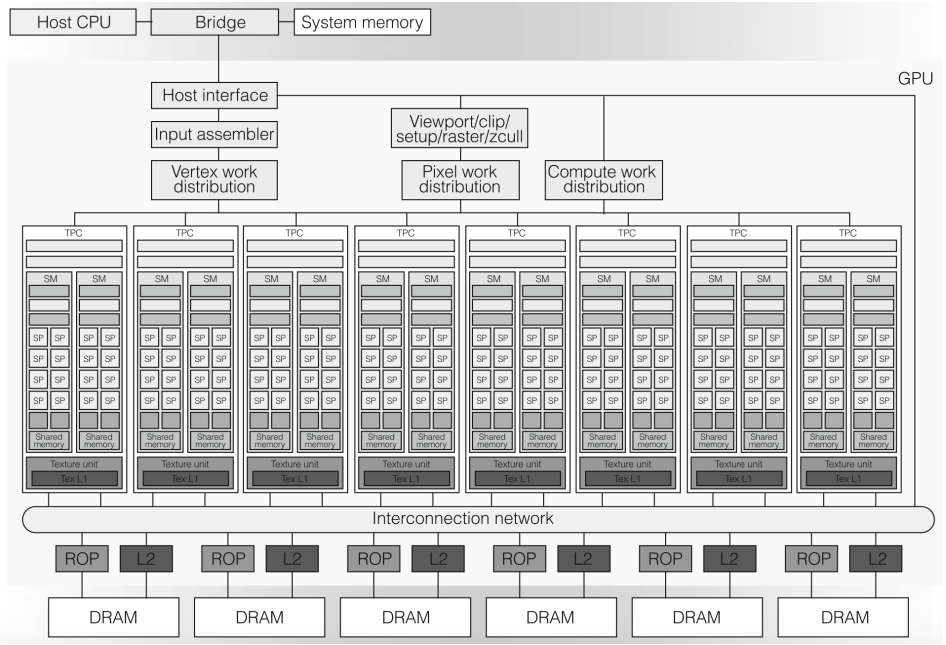
ROP (Raster Operations Pipeline): Final stage of the rendering pipeline, converting fragment outputs into pixels stored in the framebuffer.

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



Tesla 2006: Unified shader architecture

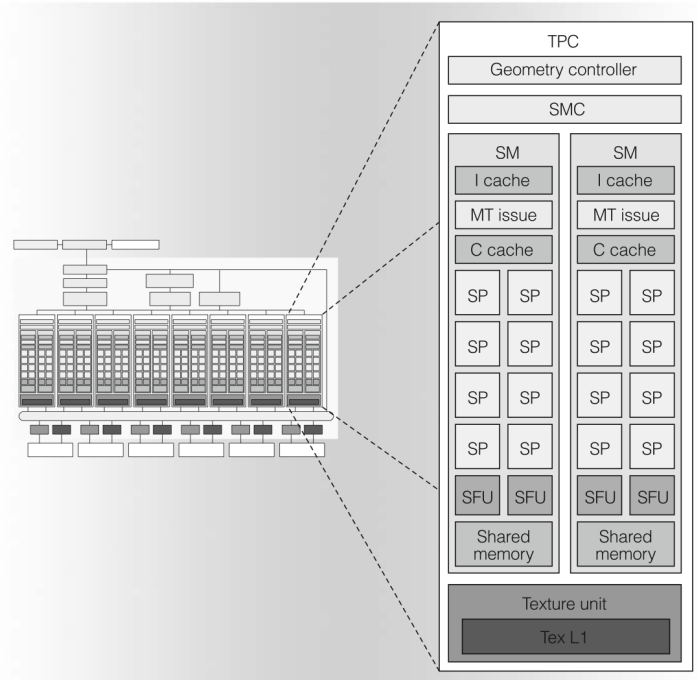
Overall Design	First CUDA-capable GPU (90nm, 2006), unified graphics and compute with five subsystems: control, compute, cache, memory, interconnect. Work distribution units dispatched tasks to scalable TPCs.
Key Modules	Each TPC had a Geometry Controller, SM Controller, 2 SMs, 1 Texture Unit + L1 cache, 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.

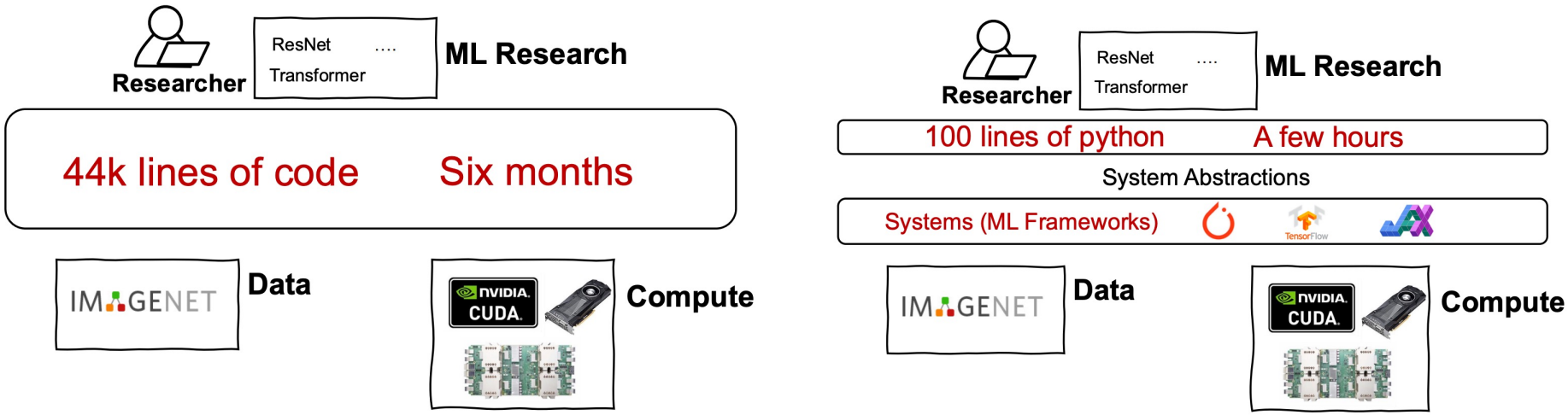
Tesla 2006: Unified shader architecture

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