GPU Architecture for Machine Learning

Workload is the king, as always

Li Shang
lishang@slai.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.

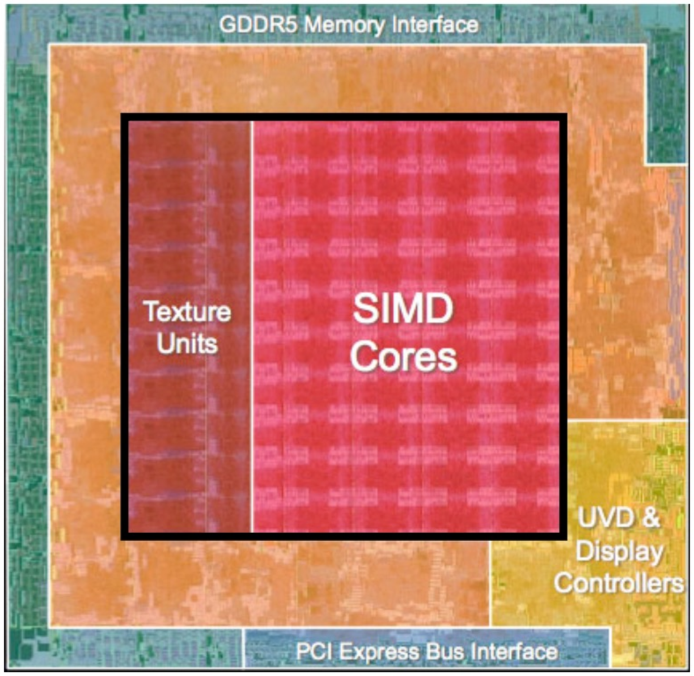
Part of the course material was created by LLM itself.

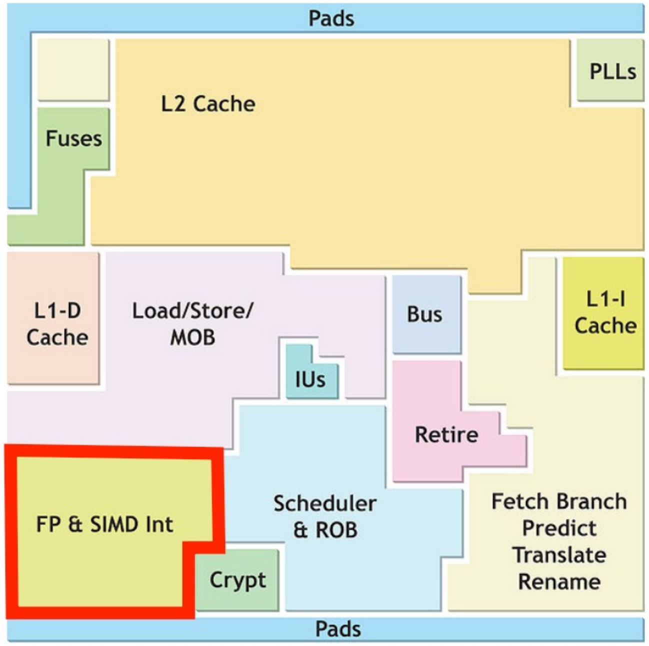
More importantly, for each and every of us, LLM shall be heavily involved in daily learning.

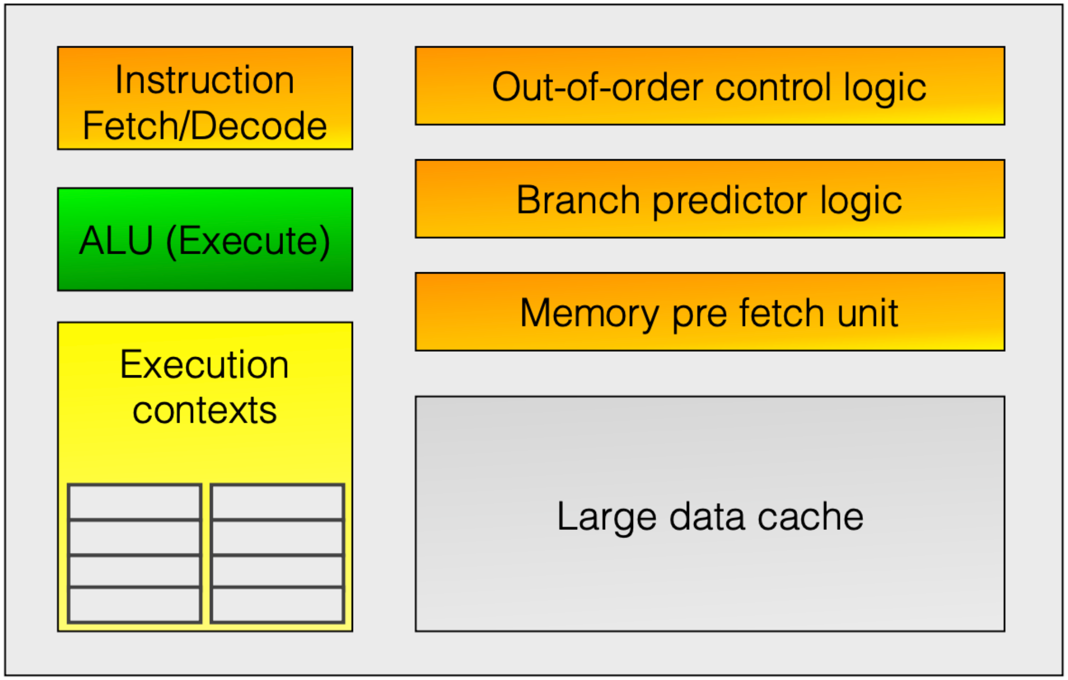
CPU vs. GPU

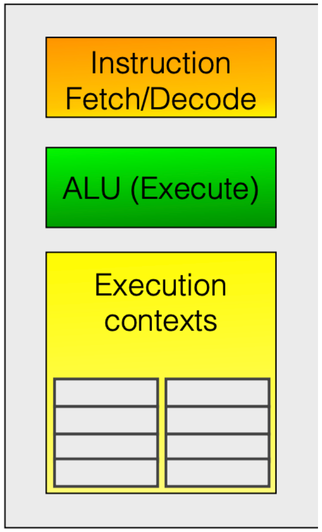
Feature	ILP	Stream Processing
Parallelism source	independent instructions	independent data elements
Typical hardware	CPU	GPU
Parallel width	~4–8 ops	thousands of threads
Programming model	sequential	data-parallel
Example	superscalar execution	kernels / streams

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

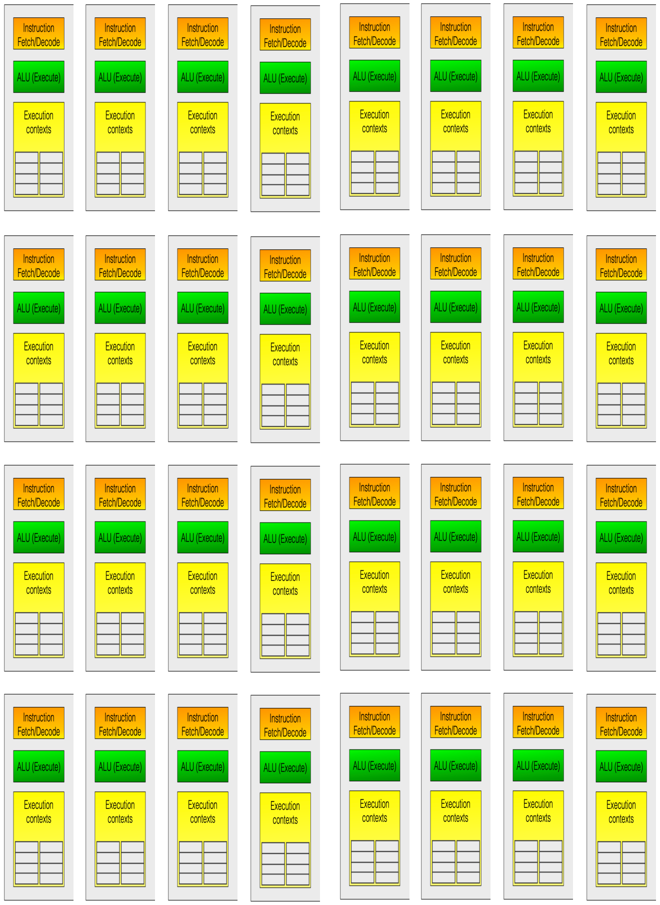

Instruction Fetch/Decode
ALU (Execute)
Execution contexts
Out-of-order control logic
Branch predictor logic
Memory pre fetch unit
Large data cache

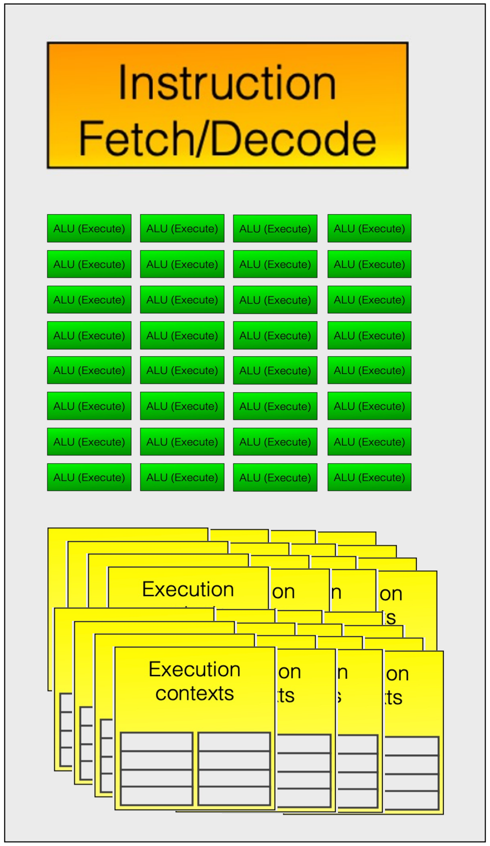


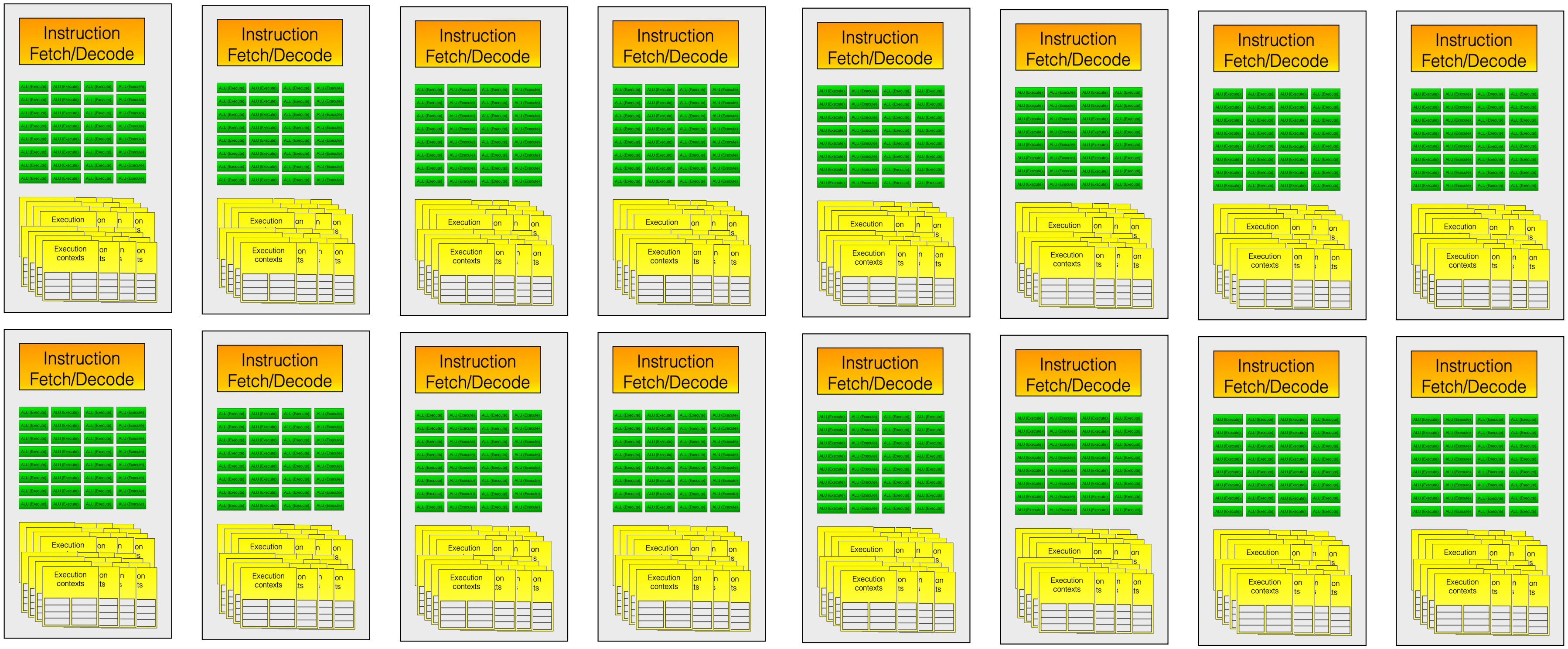
Instruction Fetch/Decode

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

GPGPU was born differently

- Throughput matters and single threads do not.
- Hide memory latency through parallelism.
- Avoid high frequency clock speed, due to power bound.

Modern GPGPU

Objective： Flexible, programming graphics and high-performance computing

Architecture： Unified graphics and parallel computing architecture

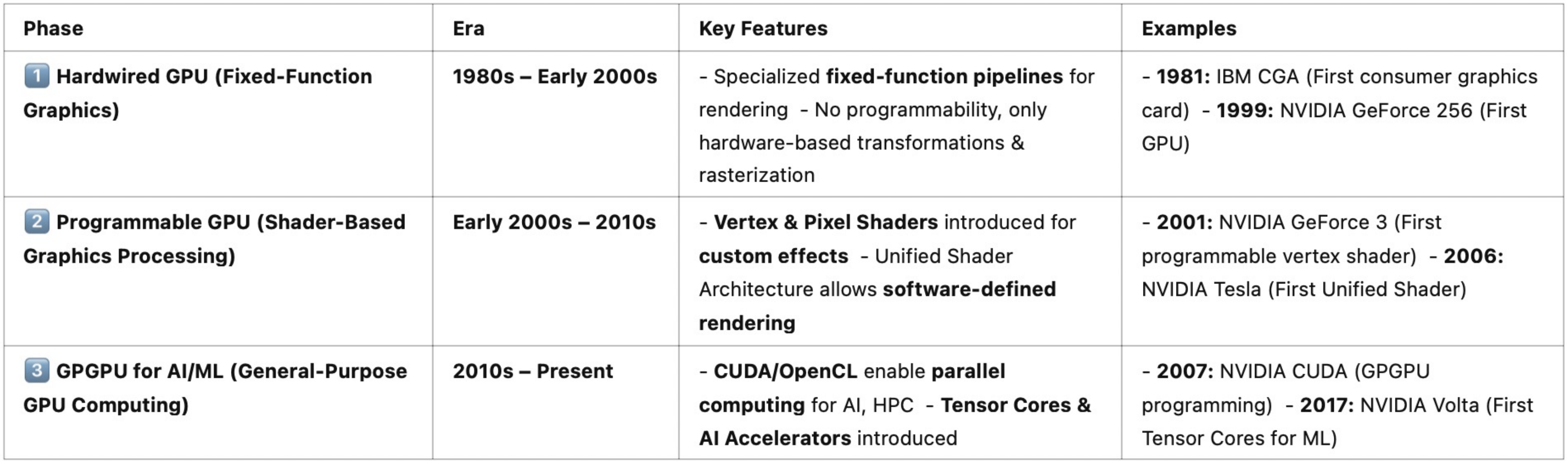
Scalability： Parallel array of processors are massively multithreaded

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

GPU: Three Generations

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

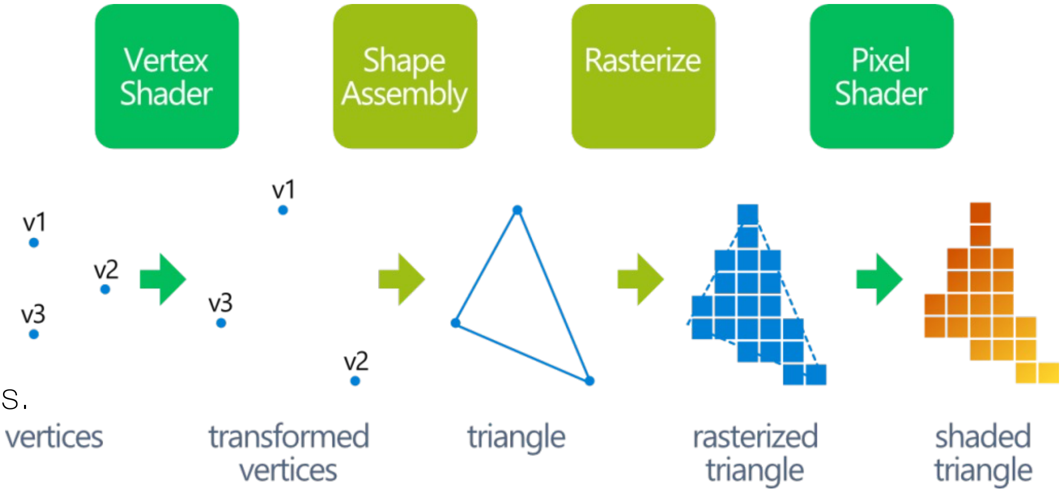


Phase	Era	Key Features	Examples
1️⃣ **Hardwired GPU (Fixed-Function Graphics)**	**1980s – Early 2000s**	- Specialized **fixed-function pipelines** for rendering - No programmability, only hardware-based transformations & rasterization	- **1981:** IBM CGA (First consumer graphics card) - **1999:** NVIDIA 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)

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 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:** Fixed-function stage that converts assembled triangles into pixels or fragments.
- **Pixel Shader Stage:** Determines the color of each pixel. Programmed using a pixel shader (also known as a fragment shader) executed once per pixel.



Programmable GPU (GeForce 7800@2005)

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

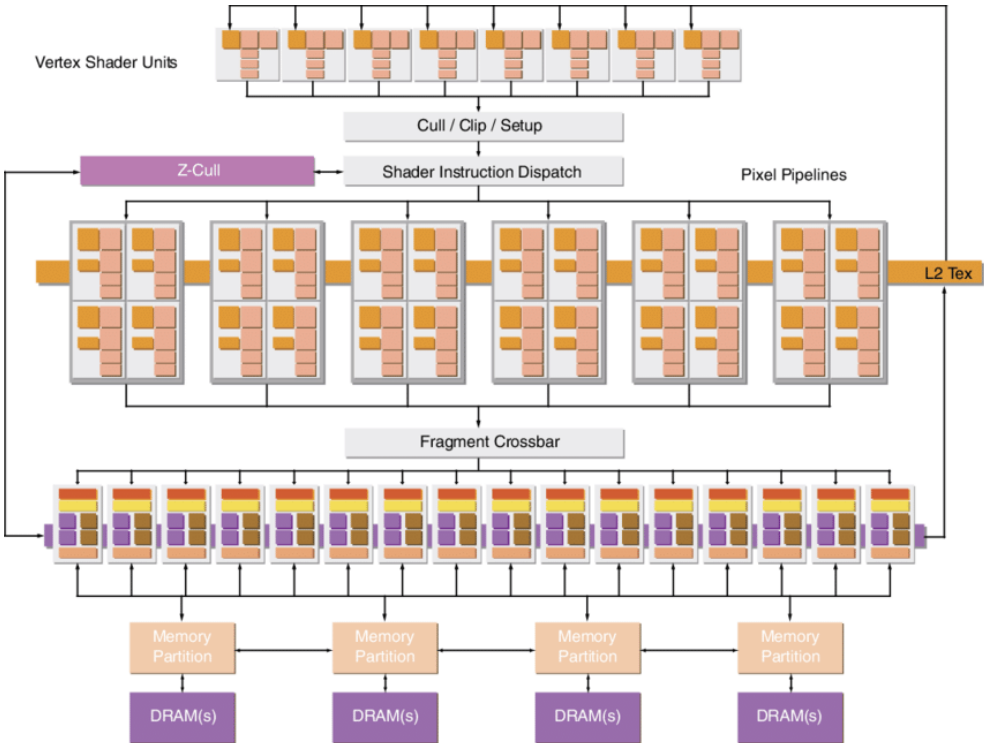
- Coordinate transformations (model → world → screen space)
- Per-vertex lighting calculations
- Vertex morphing and animation

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



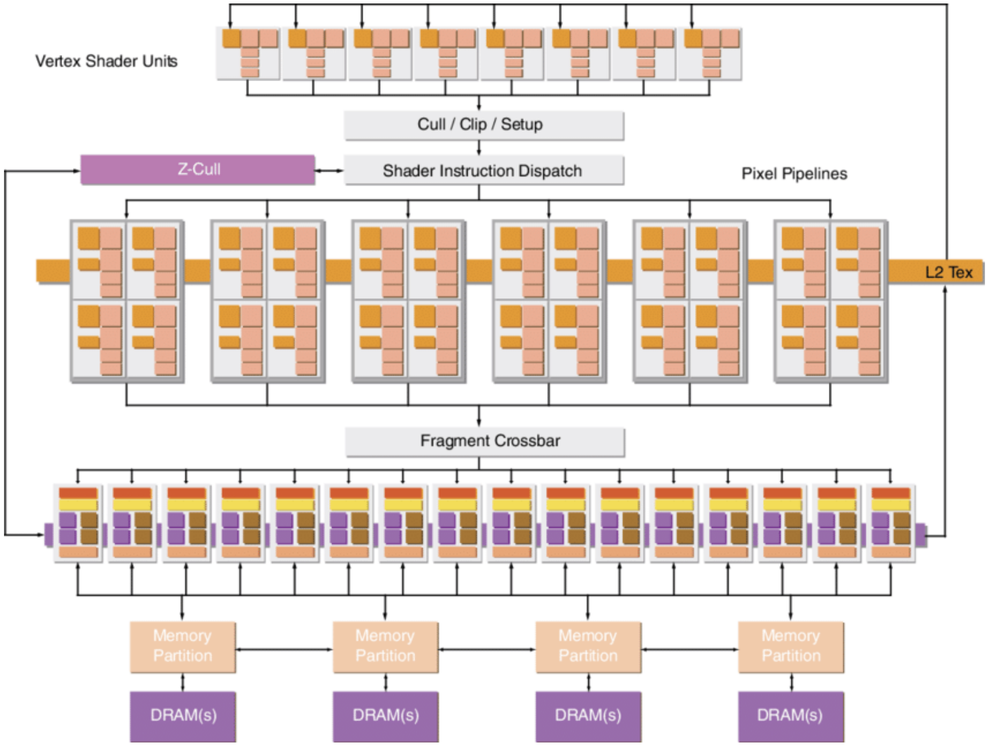
Programmable GPU (GeForce 7800@2005)

Is it really programable?

sure, it is

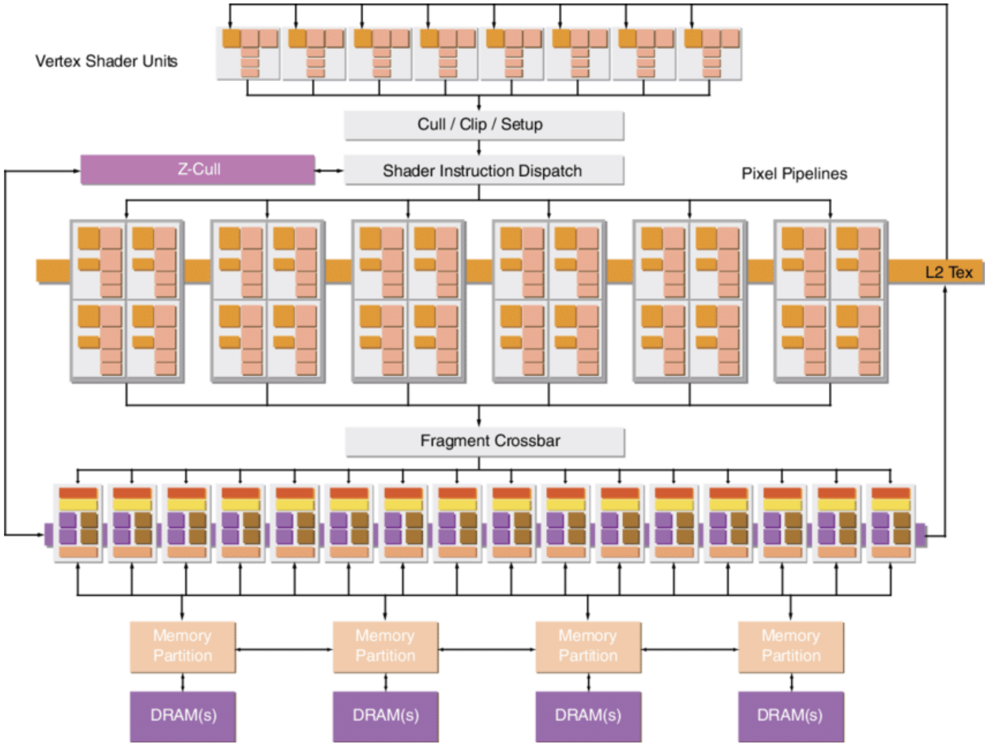
but…

Is it programmer friendly?



The Fixed Graphics Pipeline

```
Vertex shader (programmable)
↓
Primitive assembly
↓
Rasterizer
↓
Pixel shader (programmable)
↓
Framebuffer
```



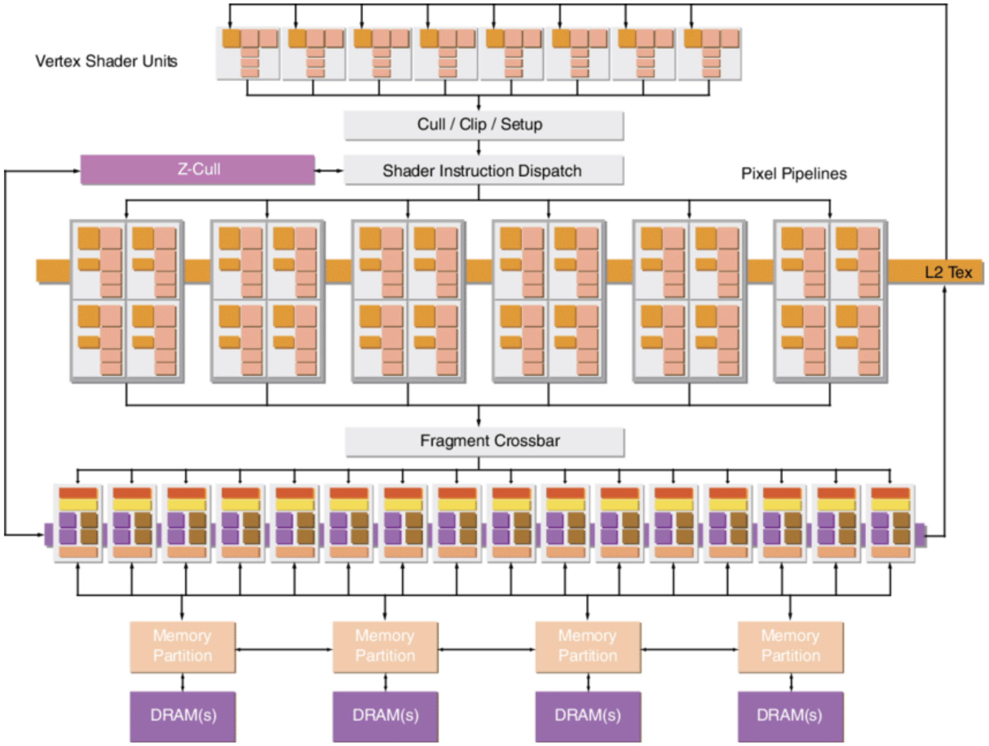
The Fixed Graphics Pipeline

This pipeline was designed for **drawing images**, not for computing numbers.

To use the GPU for computation, researchers realized:

> pixels can represent data
> fragment shaders can represent functions applied to that data

So the entire compute workflow became a **fake rendering process**.



Data Stored as Textures

Data type	Representation
Vector	1D texture
Matrix	2D texture
Field / grid	2D or 3D texture

Each pixel stored several floating-point values (RGBA channels).

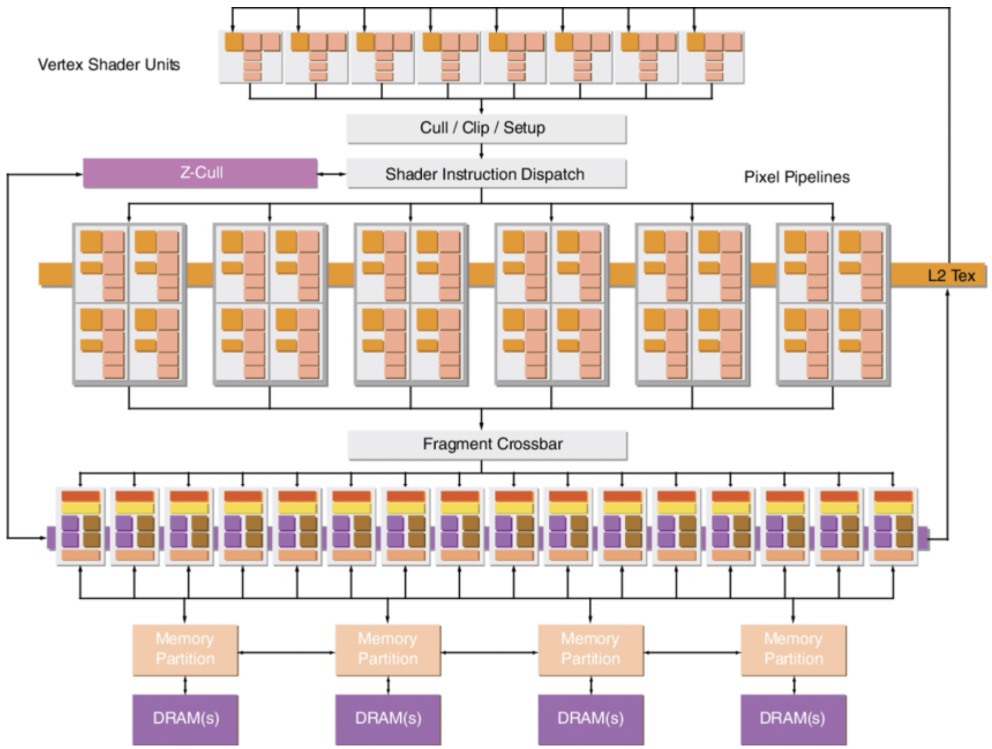
Example:

code

```
Texture pixel = (x, y, z, w)
```

So a matrix multiplication input might be stored as two textures.

This is why **texture memory was the GPU's "global memory."**



Rasterization as Parallel Iteration

Next, you needed to **apply a function to every data element**.

But GPUs didn’t have loops over arrays.

Instead you used **rasterization**.

You would render a **screen-aligned rectangle (quad)** whose size matched the texture size

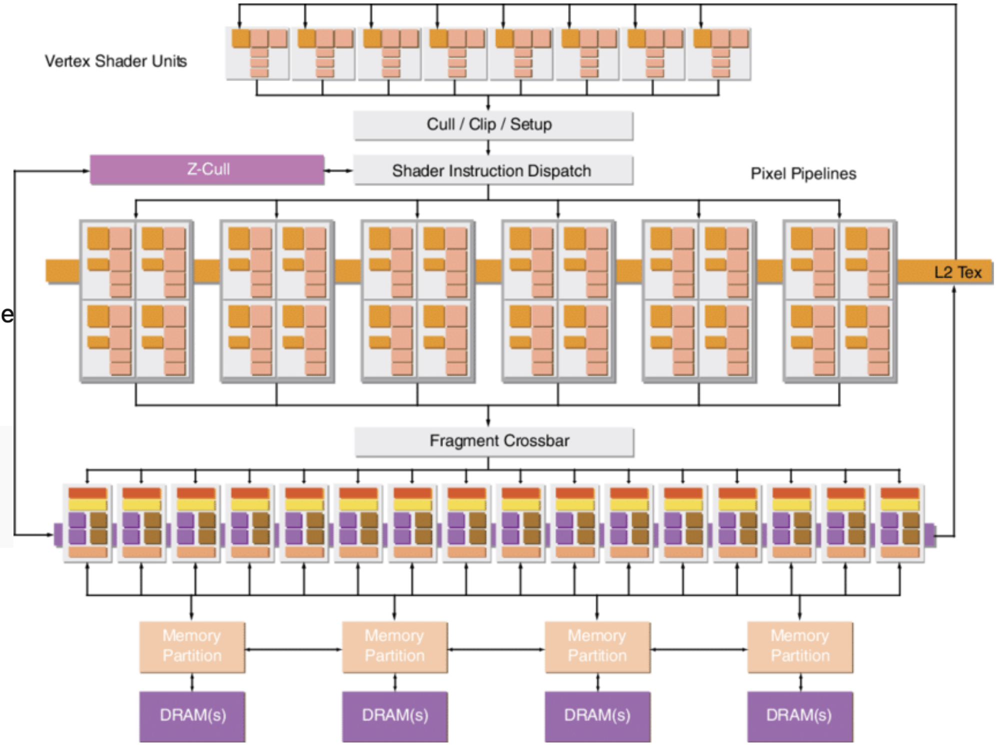
Example:

```code
Render a quad of size 1024 x 1024
```

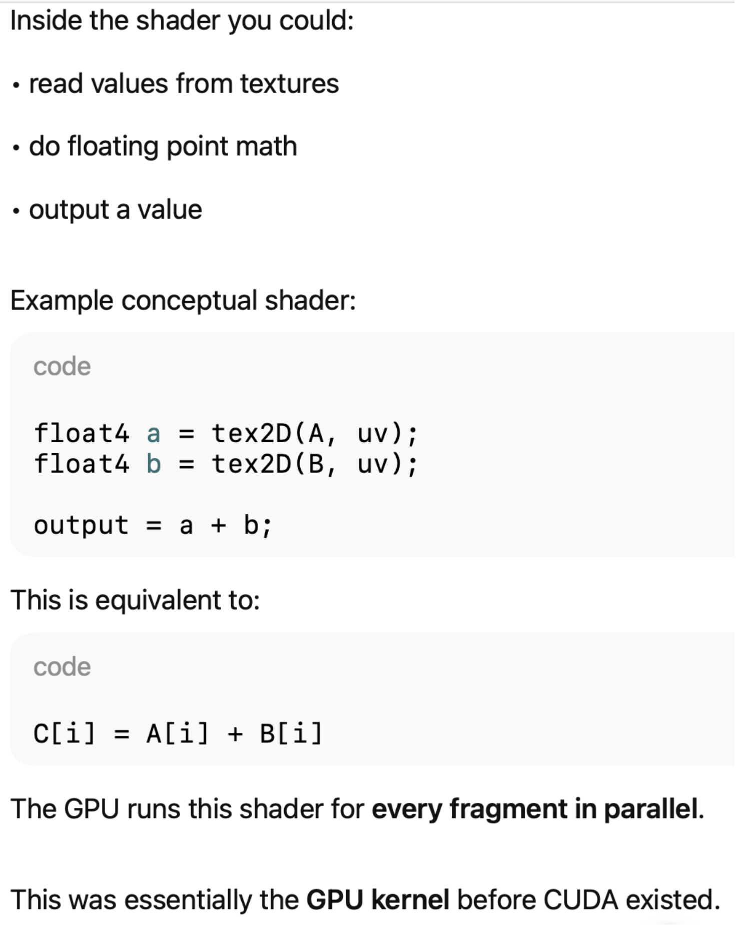
The GPU would generate **one fragment per pixel**.

So you suddenly had **1 million parallel executions**.

Each fragment corresponded to **one element of your dataset**.



Fragment Shader == Compute Kernel



Inside the shader you could:

- read values from textures
- do floating point math
- output a value

Example conceptual shader:

code

```
float4 a = tex2D(A, uv);
float4 b = tex2D(B, uv);

output = a + b;
```

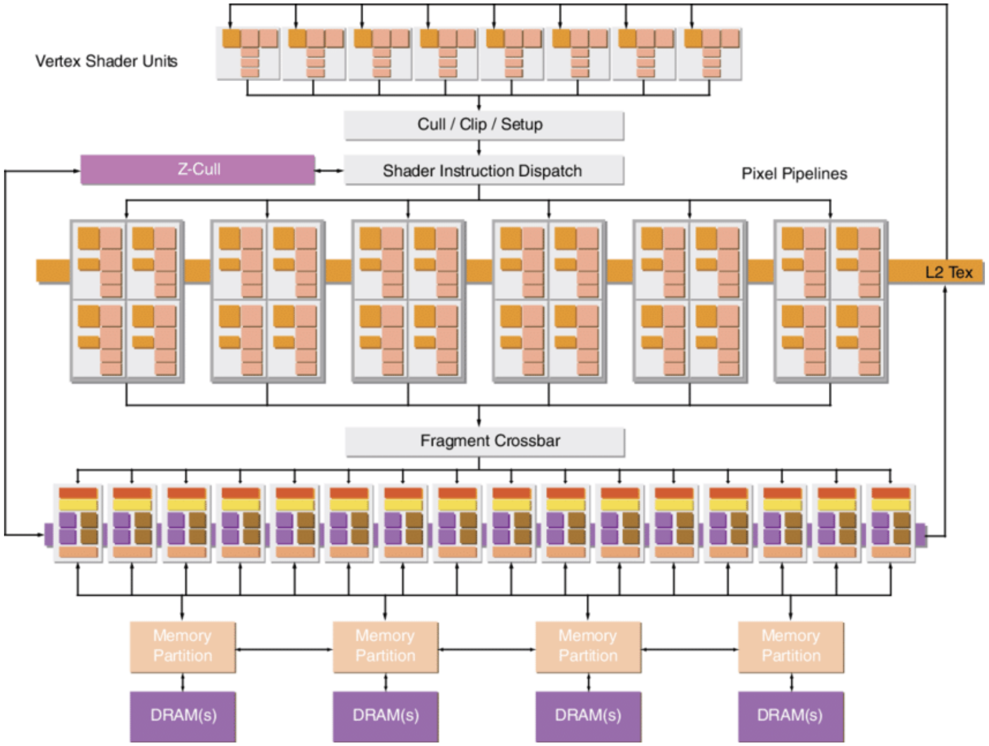
This is equivalent to:

code

```
C[i] = A[i] + B[i]
```

The GPU runs this shader for **every fragment in parallel**.

This was essentially the **GPU kernel** before CUDA existed.



Fragment Shader == Compute Kernel

```
Fragment shader
 ├─ ALU ops
 ├─ SFU ops
 └─ Texture fetch (tex2D)
```

```
Fragment shader
    ↓
Instruction issued
    ↓
Either:
   → CUDA core (ALU)
   → SFU
   → Texture unit
```

Category	Instruction Type	Examples	Functionality
ALU (scalar/vector)	Arithmetic	`add`, `mul`, `mad` (FMA), `dp3`, `dp4`	Basic math operations
Vector ops	Vector arithmetic	`dp3`, `dp4`, `normalize`	Dot product, vector math
Special function (SFU)	Transcendental	`sin`, `cos`, `exp`, `log`, `rsqrt`	Complex math functions
Texture (memory)	Texture sampling	`tex2D`, `texRECT`, `texFETCH`	Read from texture memory
Control flow	Branching	`if`, `loop`, `rep`	Conditional execution
Move / register	Data movement	`mov`, `swizzle`	Register manipulation
Output	Write result	`outColor` / `gl_FragColor`	Write to framebuffer

Render Targets == Output Arrays

Where did results go?

Into the **framebuffer**.

But instead of writing to the screen, you wrote to an **off-screen texture**.

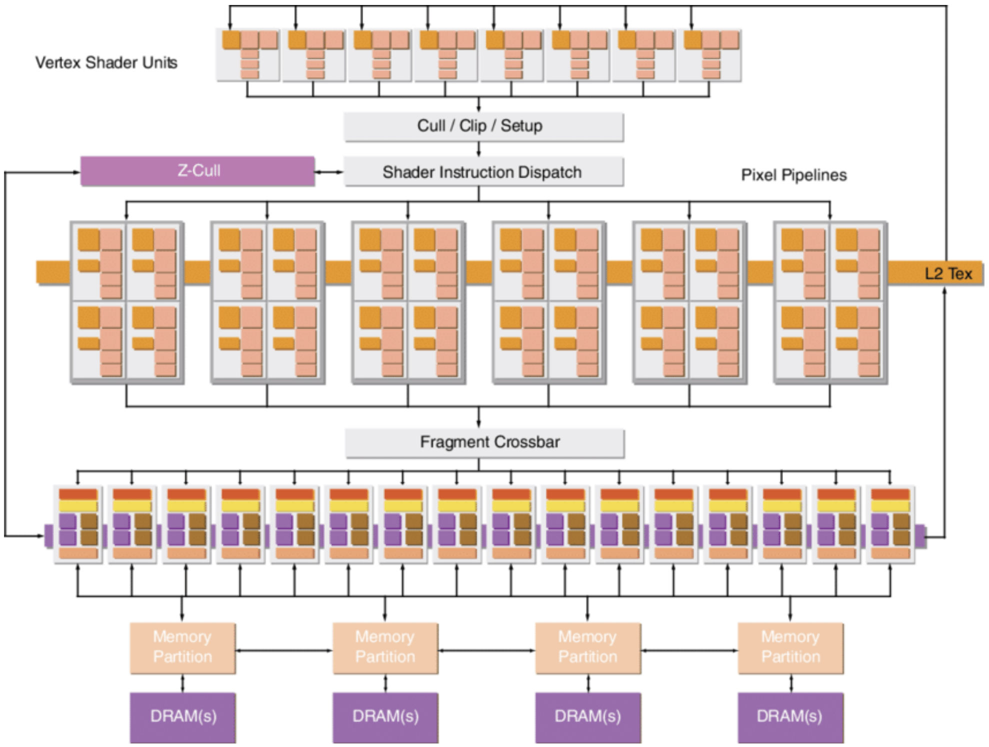
This texture was called a **render target**.

So the compute step looked like:

code

```
Input textures
    |
fragment shader
    |
output render target texture
```

That texture could then be used as input to the next pass.



CG-Specific Memory Hierarchy

Because of this restriction:

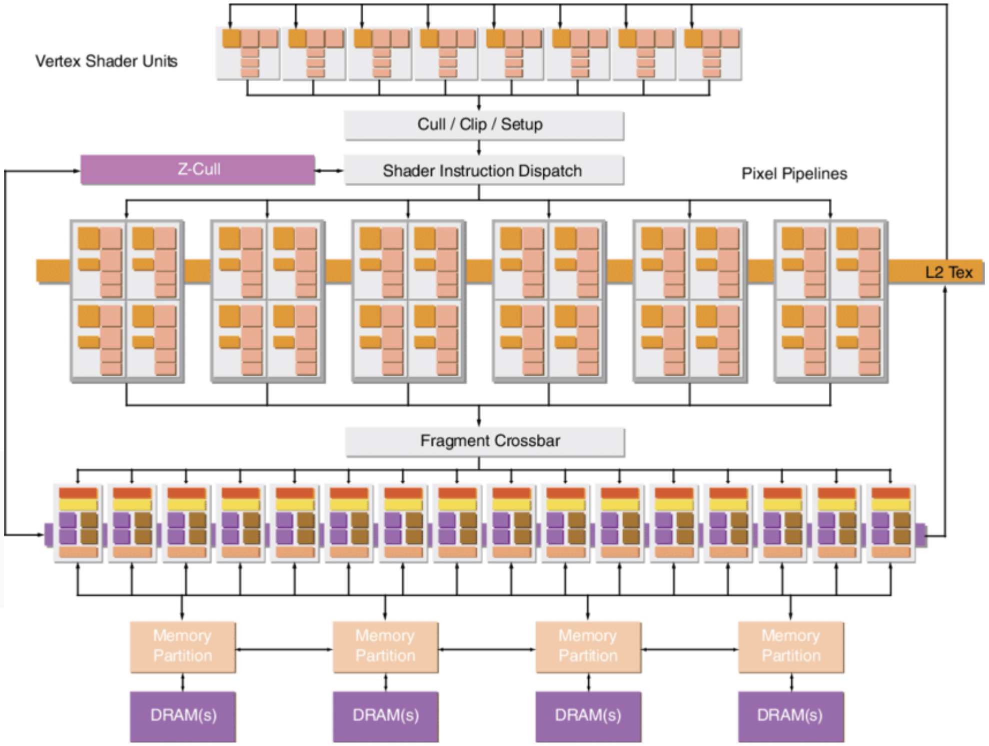
- no scatter
- no shared memory
- no atomics

Everything had to be expressed as:

code

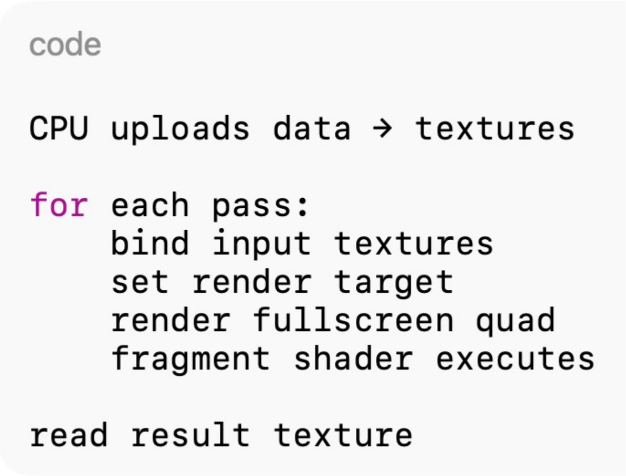
```
output[i] = f(input[i], neighbors...)
```

This is why many GPU algorithms looked like **image processing filters**.



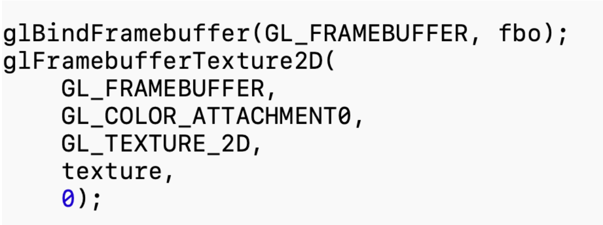
CG-Specific Compute Pipeline

A full compute pipeline looked like this:



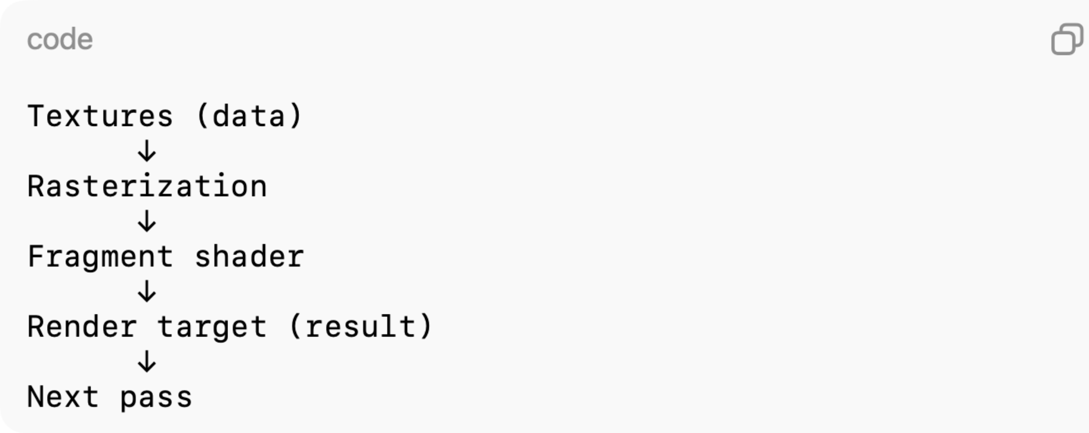
code

```
CPU uploads data → textures

for each pass:
    bind input textures
    set render target
    render fullscreen quad
    fragment shader executes

read result texture
```



```
glBindFramebuffer(GL_FRAMEBUFFER, fbo);
glFramebufferTexture2D(
    GL_FRAMEBUFFER,
    GL_COLOR_ATTACHMENT0,
    GL_TEXTURE_2D,
    texture,
    0);
```

Graphically:



code

```
Textures (data)
        ↓
Rasterization
        ↓
Fragment shader
        ↓
Render target (result)
        ↓
Next pass
```

This is exactly the workflow Brook was designed to **hide from programmers**.

Why it was Painful

Programmers had to:

- think in graphics terms
- manage render passes manually
- encode arrays as textures
- avoid illegal memory patterns
- debug using visualization

So writing GPU compute code meant thinking like this:

code

```
How do I draw a picture whose pixels represent my answer?
```

instead of

code

```
How do I compute a result?
```

Brook replaced this:

code

```
render quad
texture lookup
framebuffer write
```

with this:

code

```
kernel(streamA, streamB) -> streamC
```

Conceptually the same hardware pipeline, but with a **stream computing abstraction**.

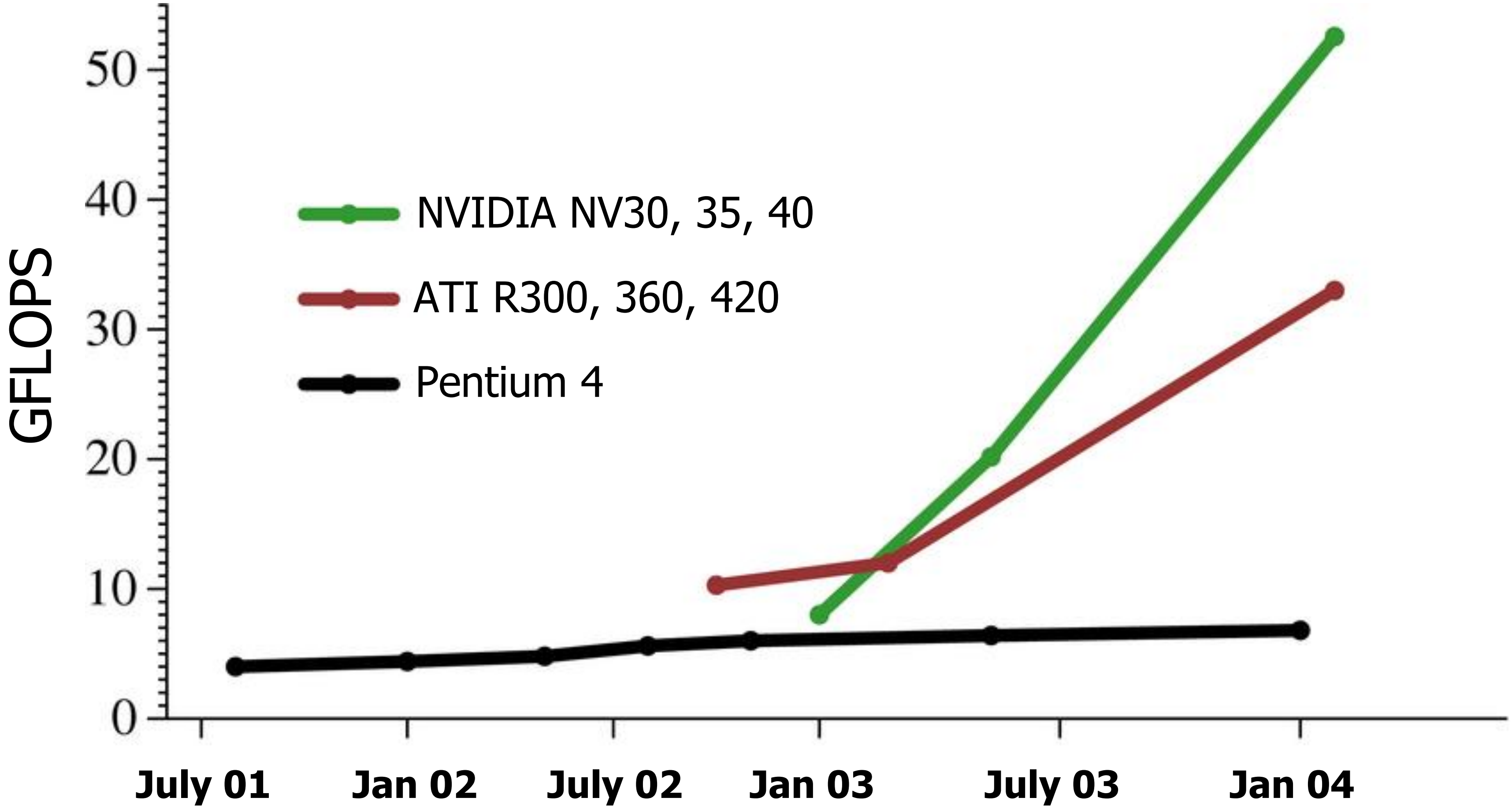
Brook for GPUs: Stream Computing on Graphics Hardware

Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan

Computer Science Department

Stanford University

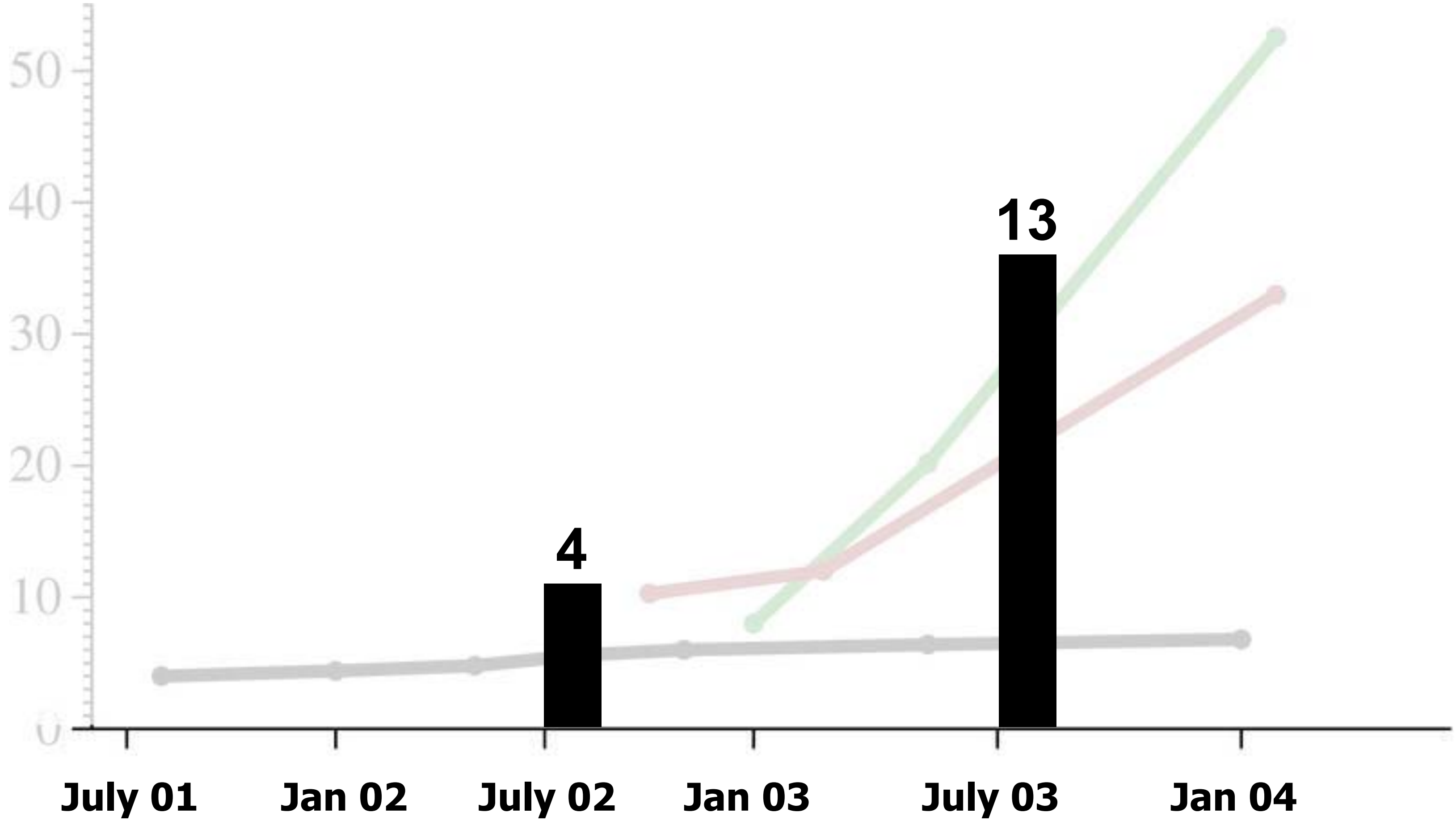
recent trends



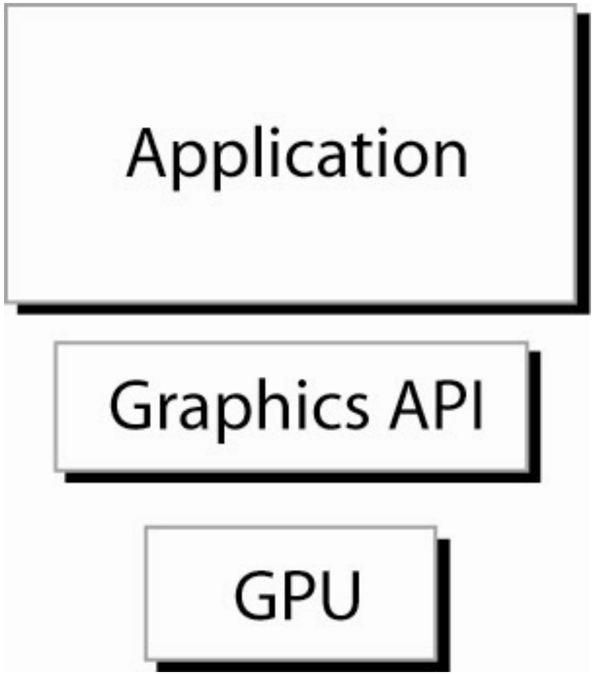
multiplies per second

recent trends

GPU-based SIGGRAPH/Graphics Hardware papers



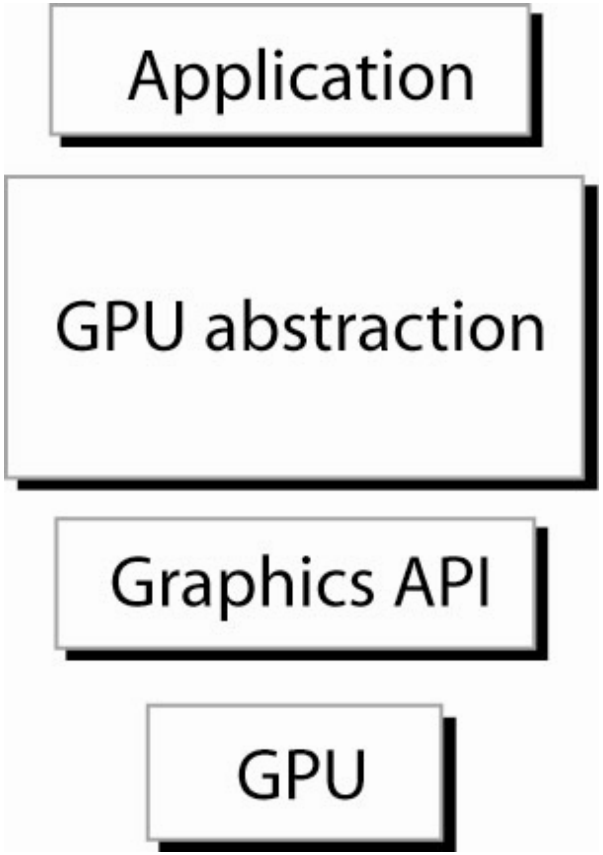
domain specific solutions



map directly to graphics primitives

requires extensive knowledge of GPU programming

building an abstraction



Application

GPU abstraction

Graphics API

GPU

general GPU computing question

- can we simplify GPU programming?
- what is the correct abstraction for GPU-based computing?
- what is the scope of problems that can be implemented efficiently on the GPU?

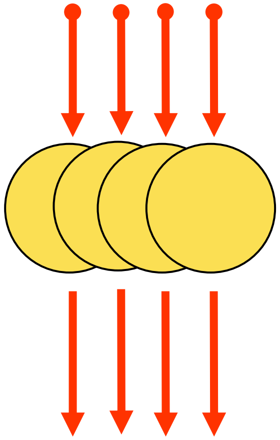
contributions

- Brook stream programming environment for GPU-based computing
  - language, compiler, and runtime system
- virtualizing or extending GPU resources
- analysis of when GPUs outperform CPUs

GPU programming model

each fragment shaded independently

- no dependencies between fragments
  - temporary registers are zeroed
  - no static variables
  - no read-modify-write textures
- multiple “pixel pipes”



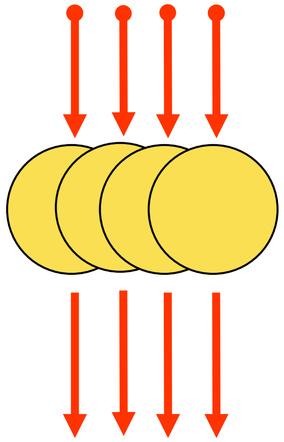
GPU = data parallel

each fragment shaded independently

- no dependencies between fragments
    - temporary registers are zeroed
    - no static variables
    - no read-modify-write textures
- multiple “pixel pipes”

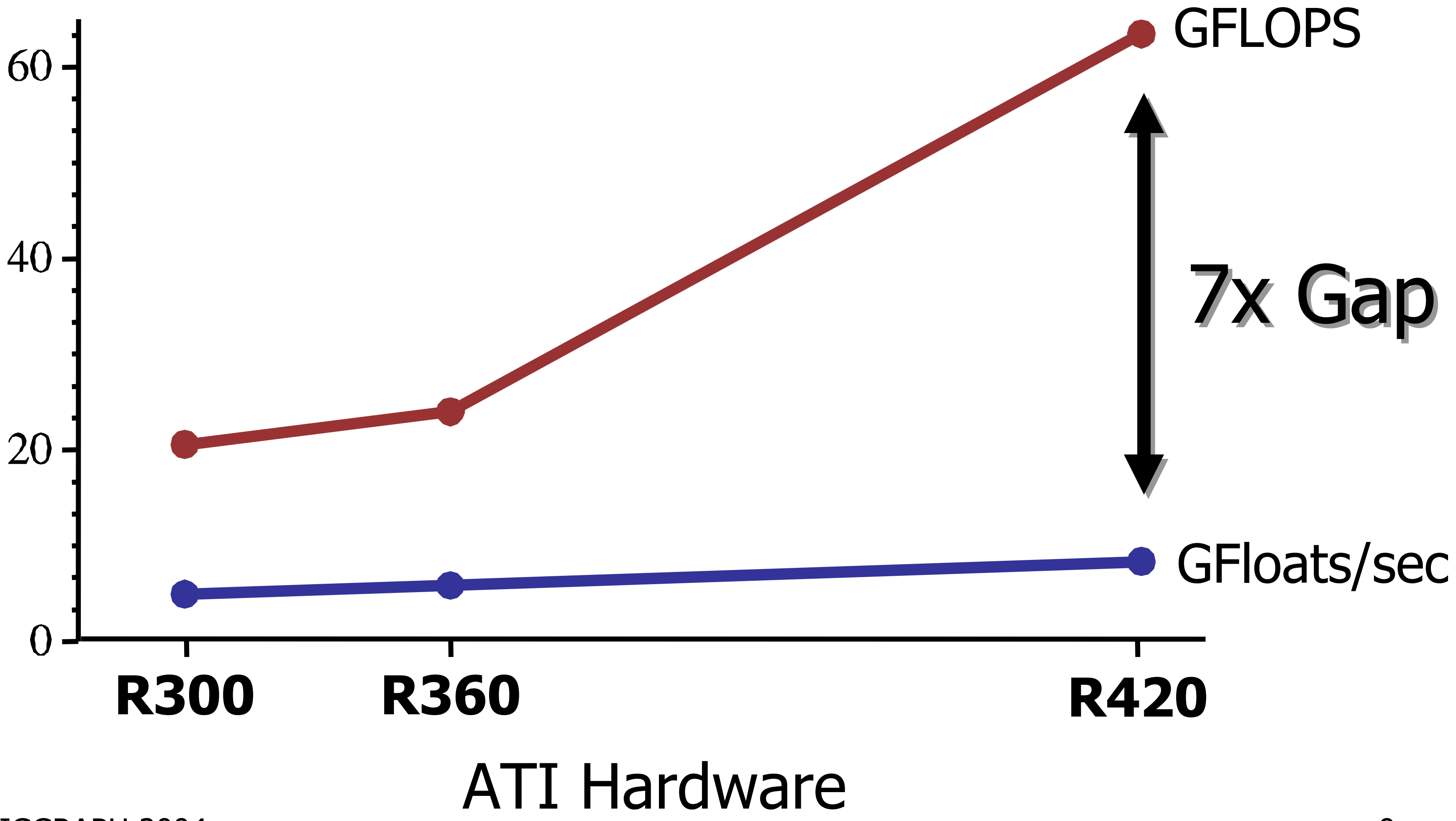
data parallelism

- support ALU heavy architectures
- hide memory latency



[Torborg and Kajiya 96, Anderson et al. 97, Igehy et al. 98]

compute vs. bandwidth



compute vs. bandwidth

arithmetic intensity =

compute-to-bandwidth ratio

graphics pipeline

– vextex
  - BW: 1 vertex = 32 bytes;
  - OP: 100-500 f32-ops / vertex
– fragment
  - BW: 1 fragment = 10 bytes
  - OP: 300-1000 i8-ops/fragment

Brook language

stream programming model

- enforce data parallel computing
  - streams
- encourage arithmetic intensity
  - kernels

design goals

- general purpose computing
  - GPU = general streaming-coprocessor
- GPU-based computing for the masses
  - no graphics experience required
  - eliminating annoying GPU limitations
- performance
- platform independent
  - ATI & NVIDIA
  - DirectX & OpenGL
  - Windows & Linux

Other languages

- Cg / HLSL / OpenGL Shading Language
  - + C-like language for expressing shader computation
  - – graphics execution model
  - – requires graphics API for data management and shader execution
- Sh [McCool et al. '04]
  - + functional approach for specifying shaders
  - evolved from a shading language
- Connection Machine C*
- StreamIt, StreamC & KernelC, Ptolemy

Brook language

C with streams

- streams
  - collection of records requiring similar computation
    - particle positions, voxels, FEM cell, …

```
Ray r<200>;
float3 velocityfield<100,100,100>;
```

  - data parallelism
    - provides data to operate on in parallel

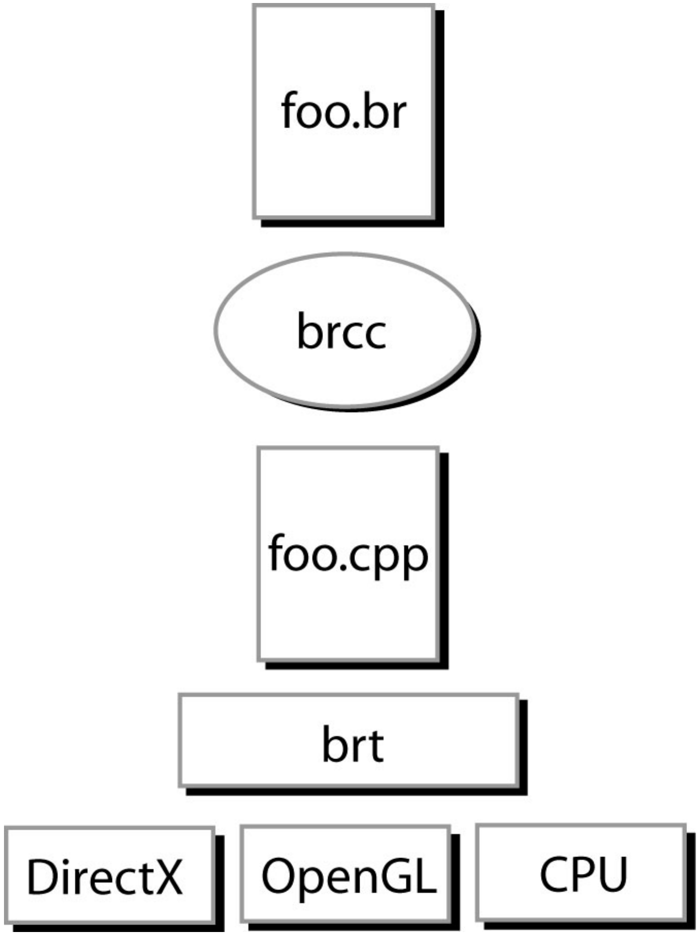
kernels

- kernels
  - functions applied to streams
    - similar to for_all construct
    - no dependencies between stream elements

```
kernel void foo (float a<>, float b<>,
                 out float result<>) {
  result = a + b;
}

float a<100>;
float b<100>;
float c<100>;

foo(a,b,c);
```

```
for (i=0; i<100; i++)
      c[i] = a[i]+b[i];
```

system outline



brcc

source to source compiler

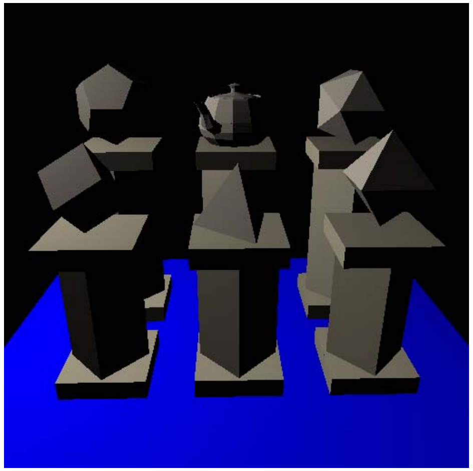
- generate CG & HLSL code
- CGC and FXC for shader assembly
- virtualization

brt

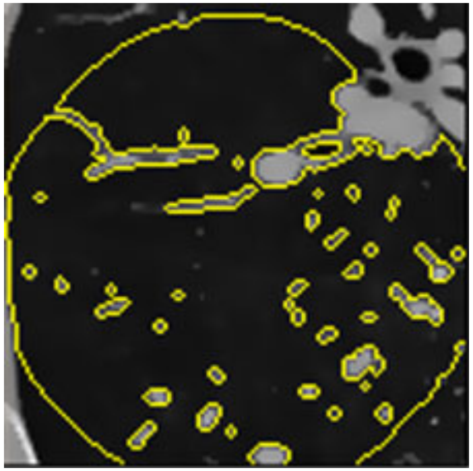
Brook run-time library

- stream texture management
- kernel shader execution

applications

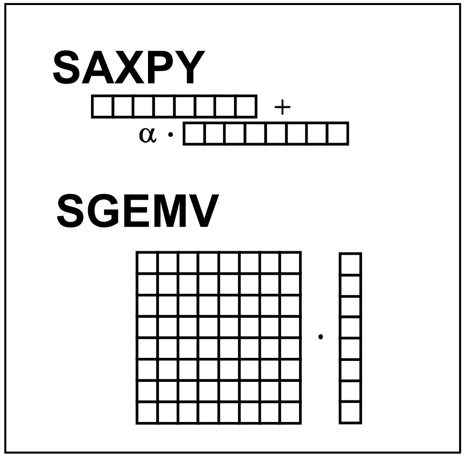


ray-tracer



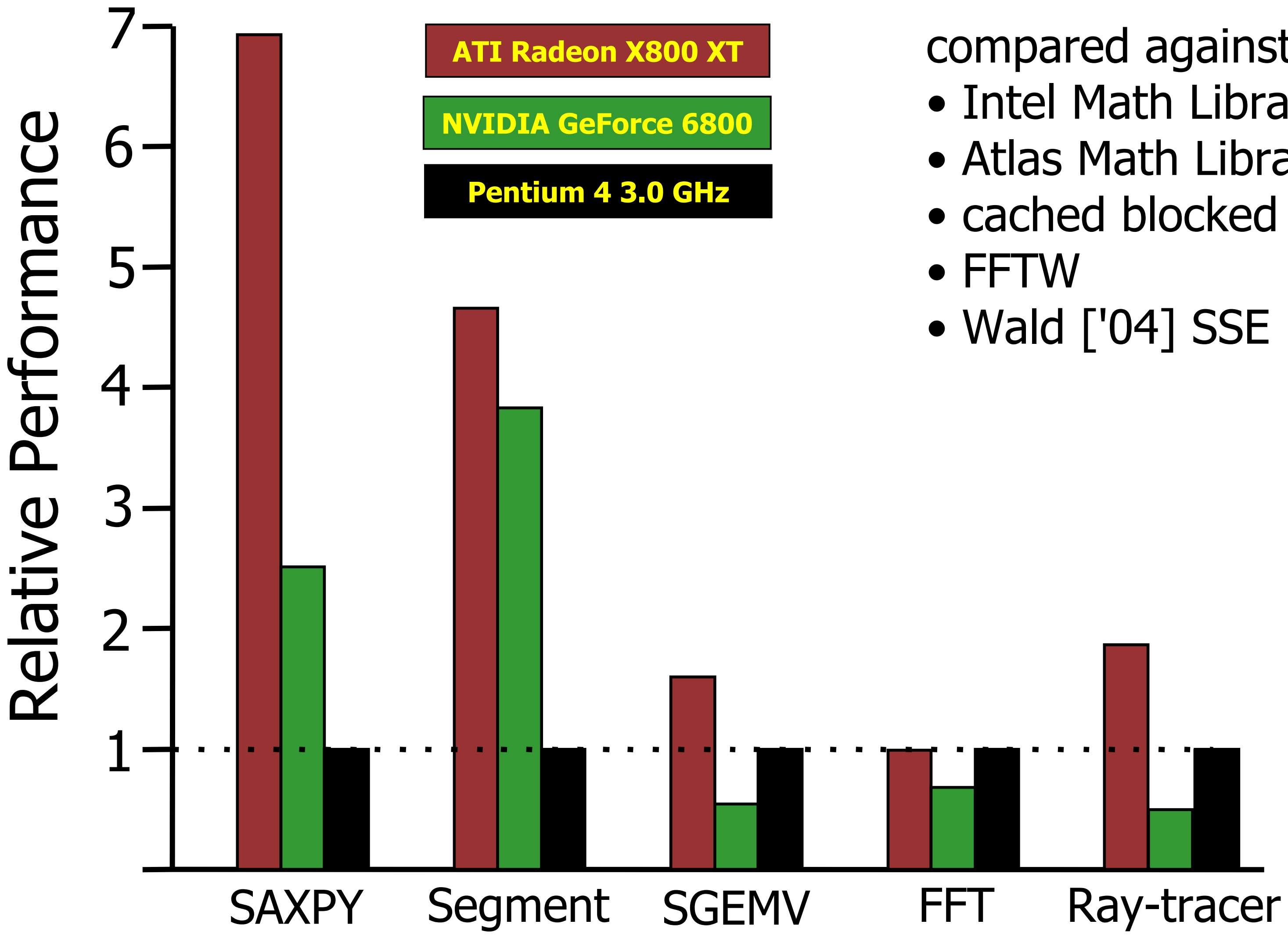
segmentation

fft edge detect



linear algebra

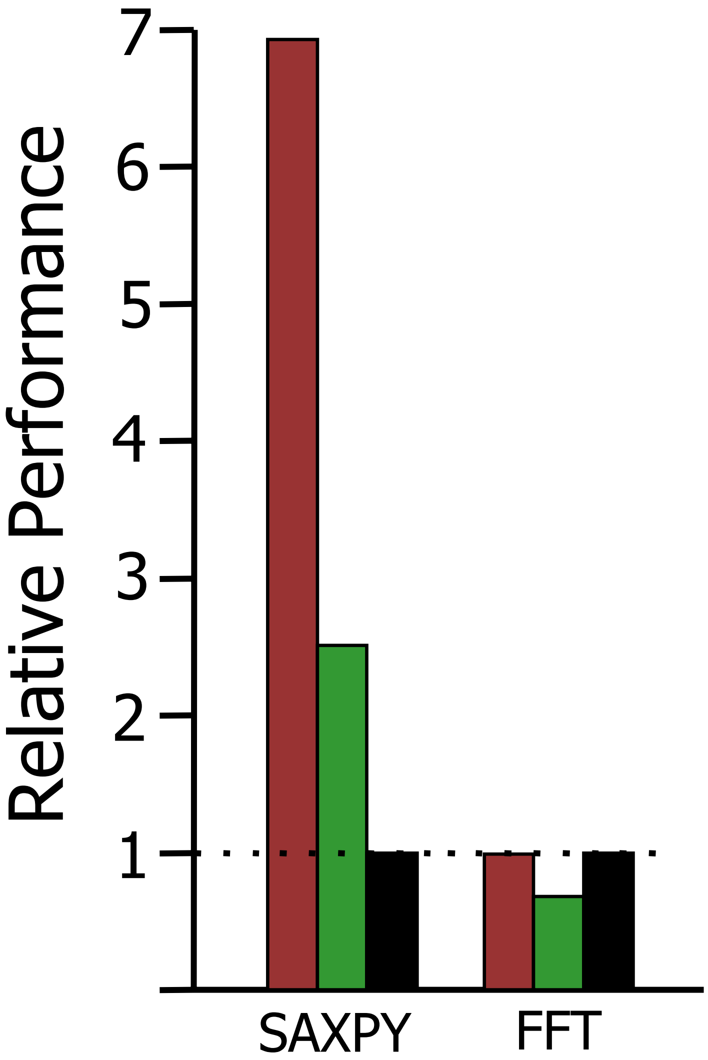
evaluation



compared against:

- Intel Math Library
- Atlas Math Library
- cached blocked segmentation
- FFTW
- Wald ['04] SSE Ray-Triangle

evaluation



GPU wins when…

- limited data reuse
  - ✓ SAXPY
  - ✗ FFT

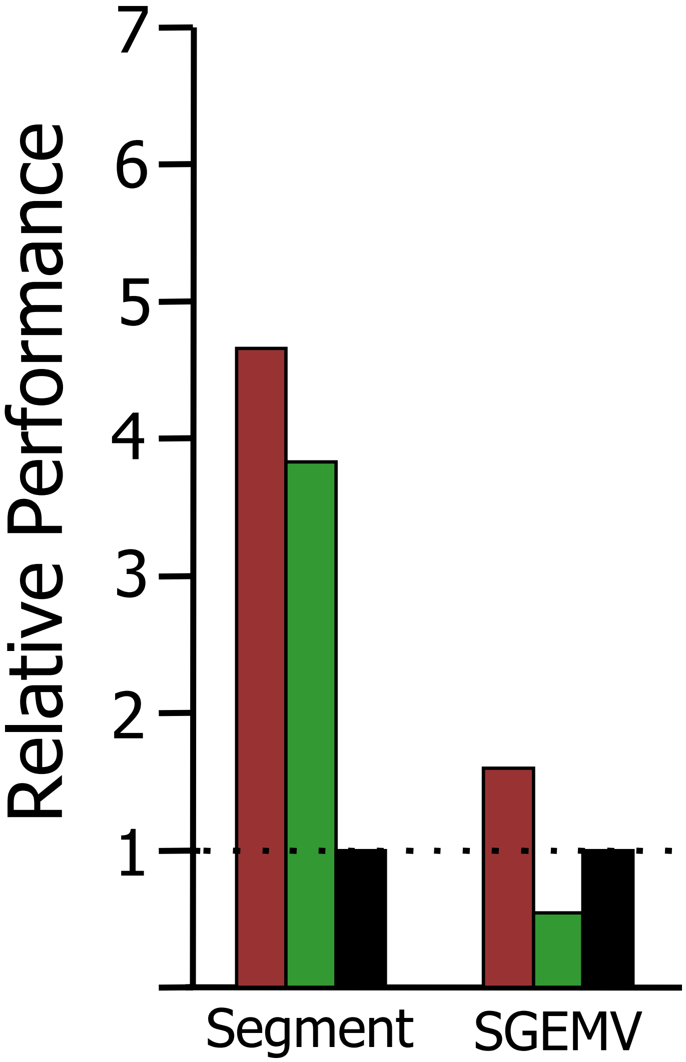
Pentium 4 3.0 GHz

44 GB/sec peak cache bandwidth

NVIDIA GeForce 6800 Ultra

36 GB/sec peak memory bandwidth

evaluation



GPU wins when…

- arithmetic intensity
  - ✓ Segment

    3.7 ops per word
  - ✗ SGEMV

    1/3 ops per word

outperforming the CPU

considering GPU transfer costs: T_r

- computational intensity: γ

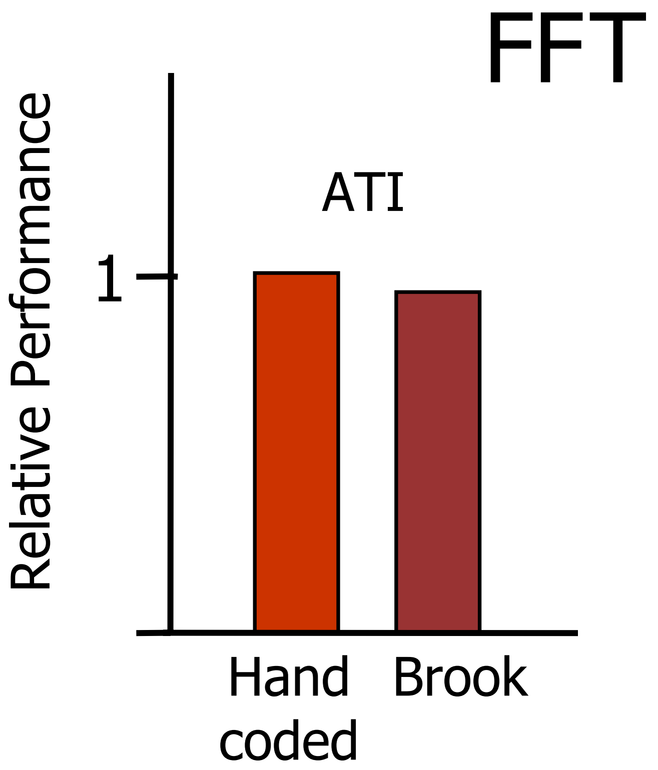
$$\gamma \equiv K_{gpu} / T_r$$

work per word transferred

considering CPU cost to issuing a kernel

efficiency

Brook version within 80% of hand-coded GPU version

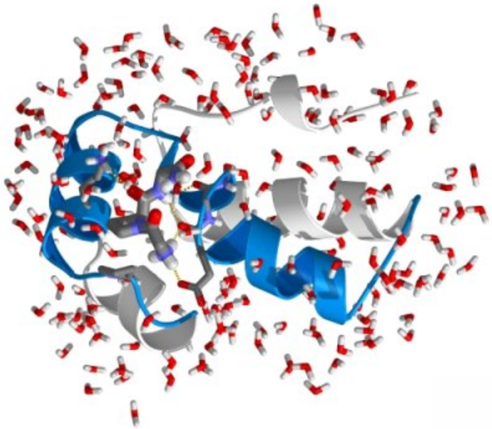


summary

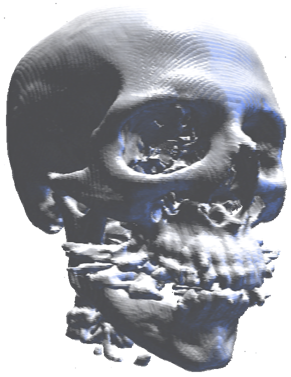
- GPUs are faster than CPUs
  - and getting faster
- why?
  - data parallelism
  - arithmetic intensity
- what is the right programming model?
  - Brook
  - stream computing

summary

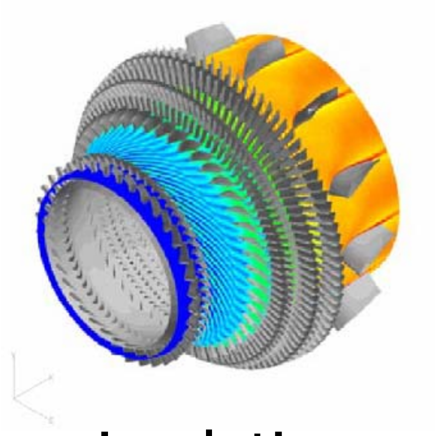
GPU-based computing for the masses



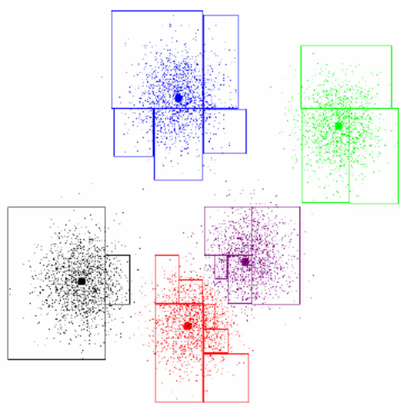
bioinfomatics



rendering



simulation



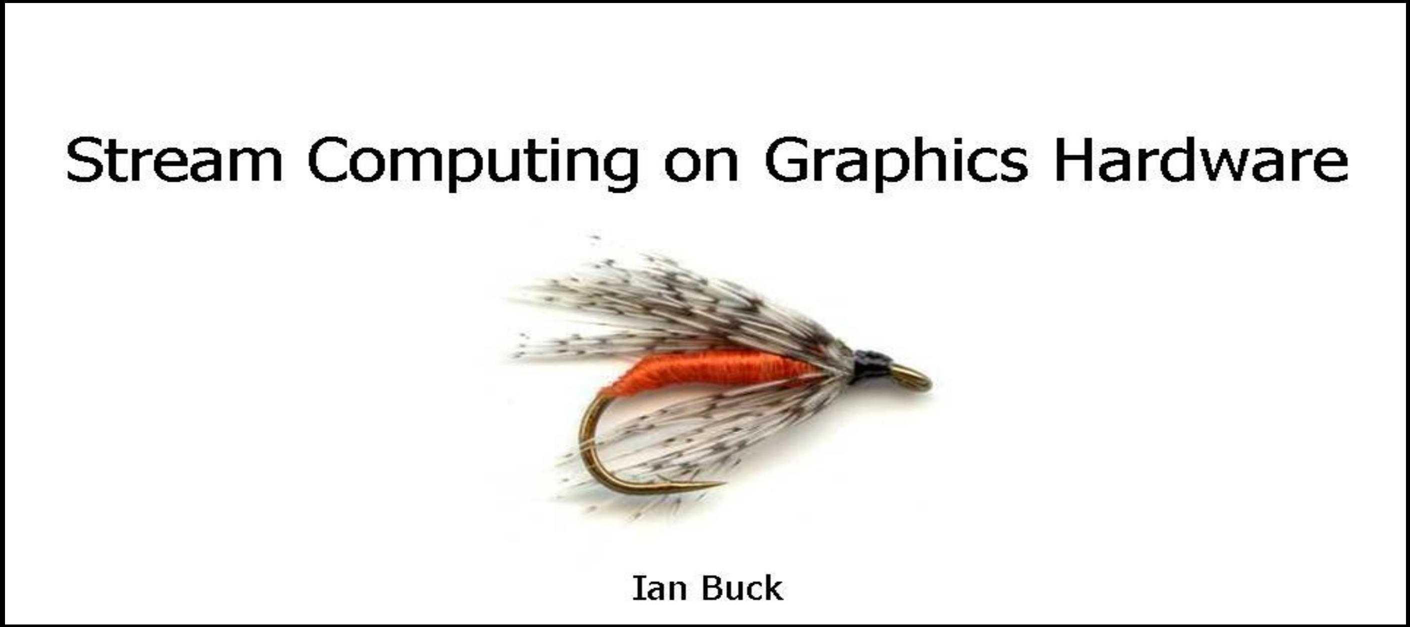
statistics

GPU TECHNOLOGY CONFERENCE

GPU Computing Past, Present, Future

Ian Buck, GM GPU Computing Sw

History….

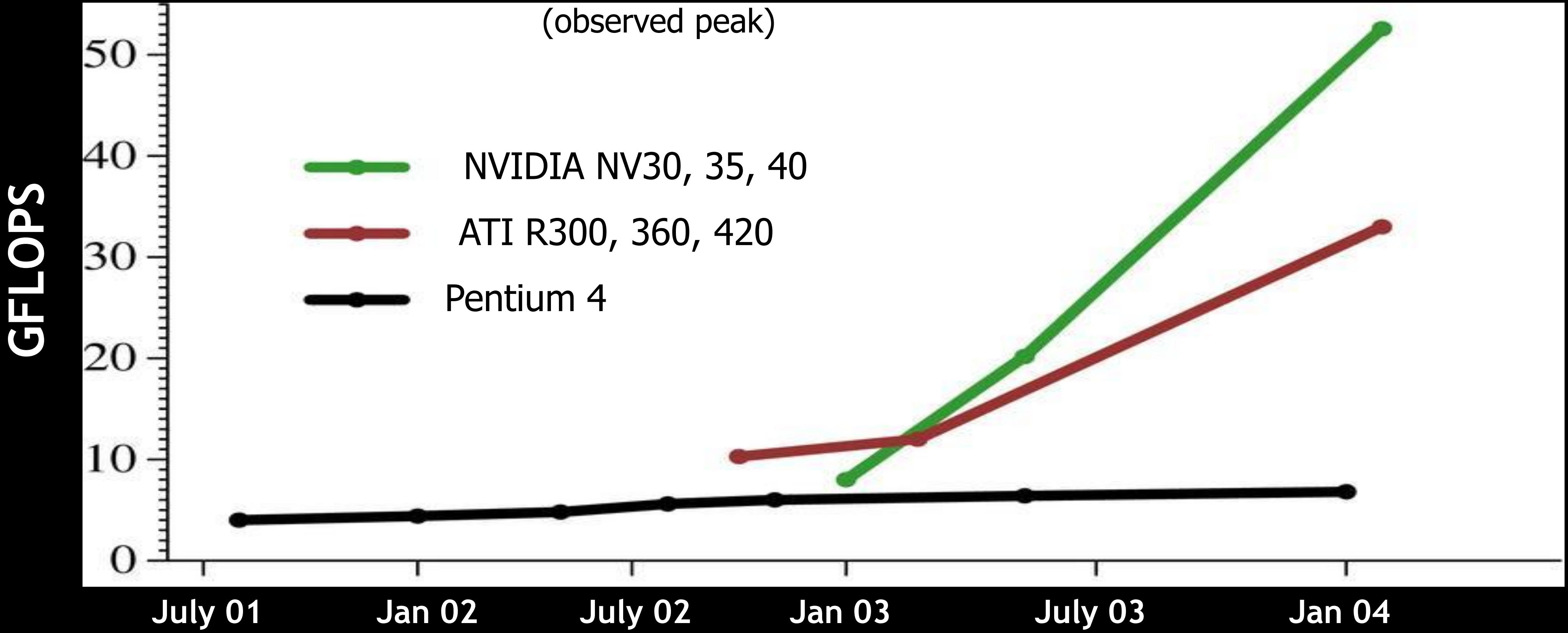


Stream Computing on Graphics Hardware

Ian Buck

GPGPU in 2004

recent trends



GPU history

	Product	Process	Trans	MHz	GFLOPS (MUL)
Aug-02	GeForce FX5800	0.13	121M	500	8
Jan-03	GeForce FX5900	0.13	130M	475	20
Dec-03	GeForce 6800	0.13	222M	400	53

translating transistors into performance

- 1.8x increase of transistors
- 20% *decrease* in clock rate
- 6.6x GFLOP speedup

Stunning Graphics Realism

Lush, Rich Worlds

Crysis © 2006 Crytek / Electronic Arts

POWERED BY NVIDIA

Incredible Physics Effects

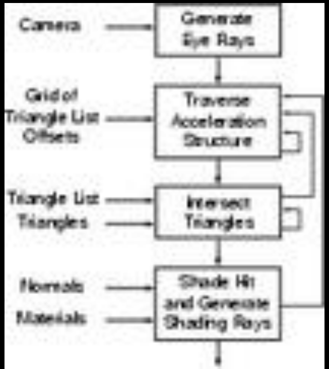
Hellgate: London © 2005-2006 Flagship Studios, Inc. Licensed by NAMCO BANDAI Games America, Inc.

Core of the Definitive Gaming Platform

Full Spectrum Warrior: Ten Hammers © 2006 Pandemic Studios, LLC. All rights reserved. © 2006 THQ Inc. All rights reserved.

Early GPGPU (2002)

www.gpgpu.org


Camera
Generate Eye Rays
Grid of Triangle List Offsets
Traverse Acceleration Structure
Triangle List
Triangles
Intersect Triangles
Normals
Materials
Shade Hit and Generate Shading Rays

Early Raytracing

- **Ray Tracing on Programmable Graphics Hardware**
  Purcell *et al.*
- **PDEs in Graphics Hardware**
  Strzodka,,Rumpf
- **Fast Matrix Multiplies using Graphics Hardware**
  Larsen, McAllister
- **Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis.**
  Thompson *et al.*

Programming model challenge

- Demonstrate GPU performace
- PHD computer graphics to do this
- Financial companies hiring game programmers

- “GPU as a processor”

Brook (2003)

C with streams

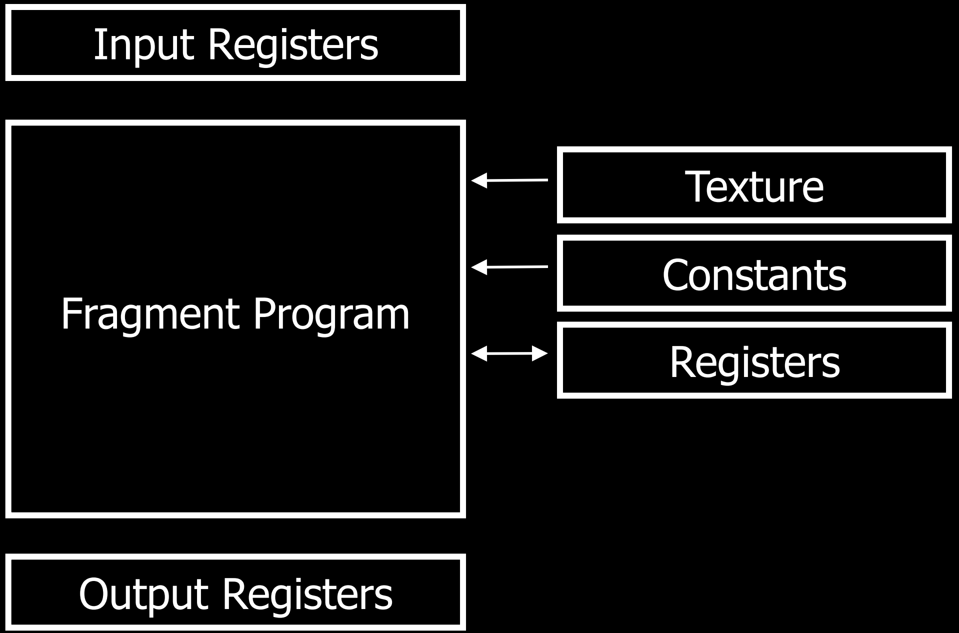
- streams
  - collection of records requiring similar computation
    - particle positions, voxels, FEM cell, …

      ```
      Ray r<200>;
      float3 velocityfield<100,100,100>;
      ```
  - similar to arrays, but…
    - index operations disallowed: `position[i]`
    - read/write stream operators:

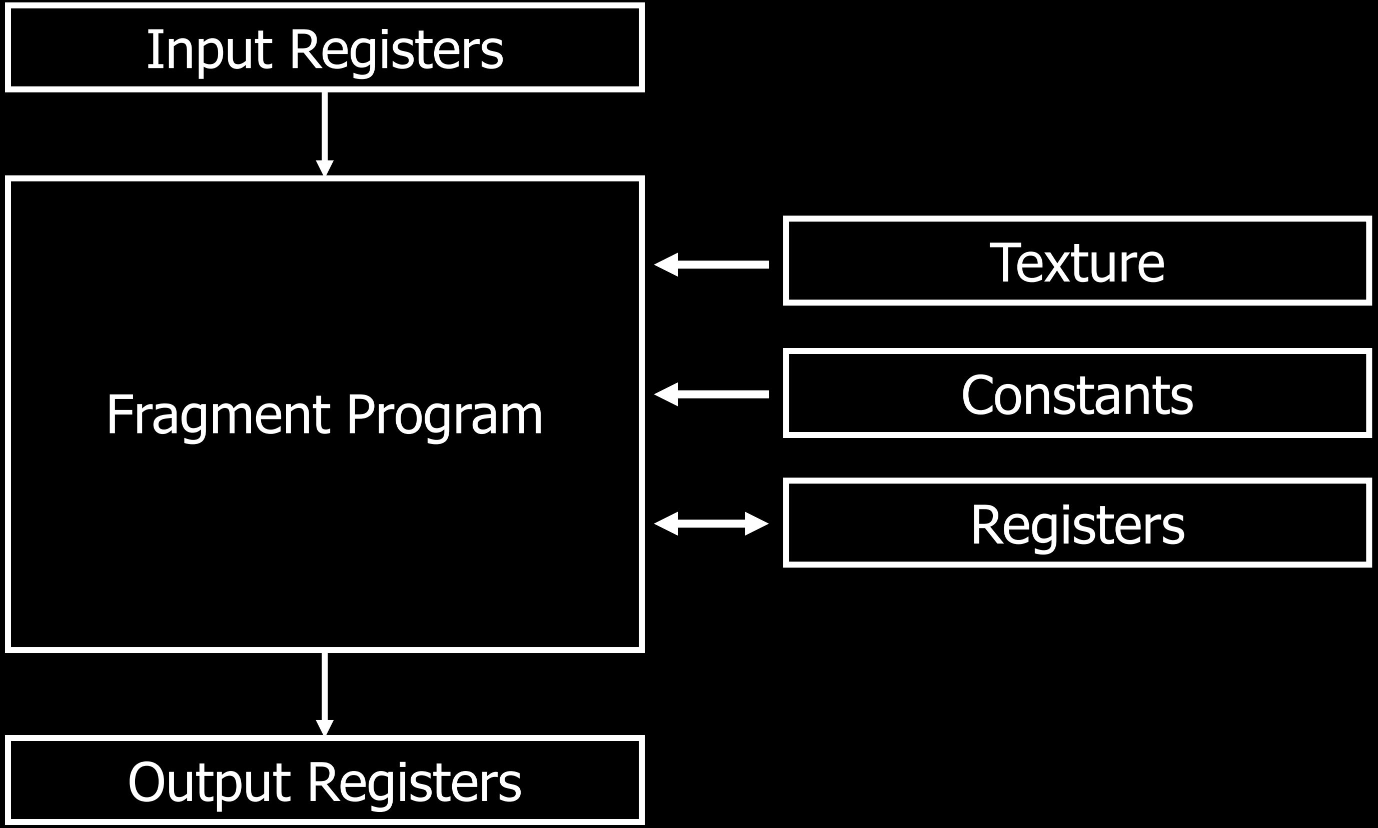
      ```
      streamRead (positions, p_ptr);
      streamWrite (velocityfield, v_ptr);
      ```

Challenges

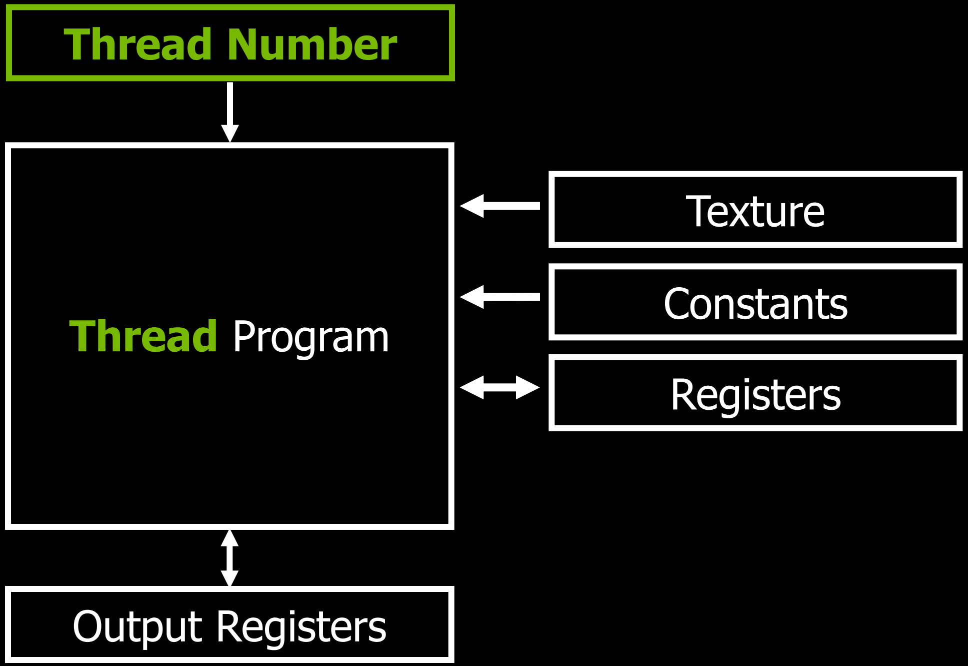
- Graphics API
- Addressing modes
  - Limited texture size/dimension
- Shader capabilities
  - Limited outputs
- Instruction sets
  - Integer & bit ops
- Communication limited
  - Between pixels
  - Scatter a[i] = p



GeForce 7800 Pixel



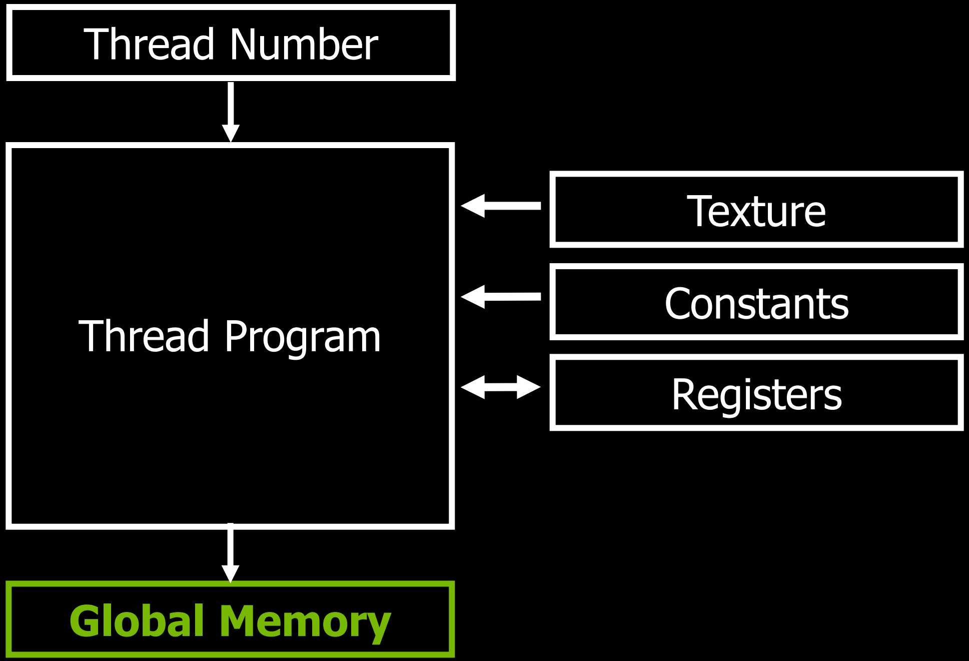
Thread Programs



Features

- Millions of instructions
- Full Integer and Bit instructions
- No limits on branching, looping
- 1D, 2D, or 3D thread ID allocation

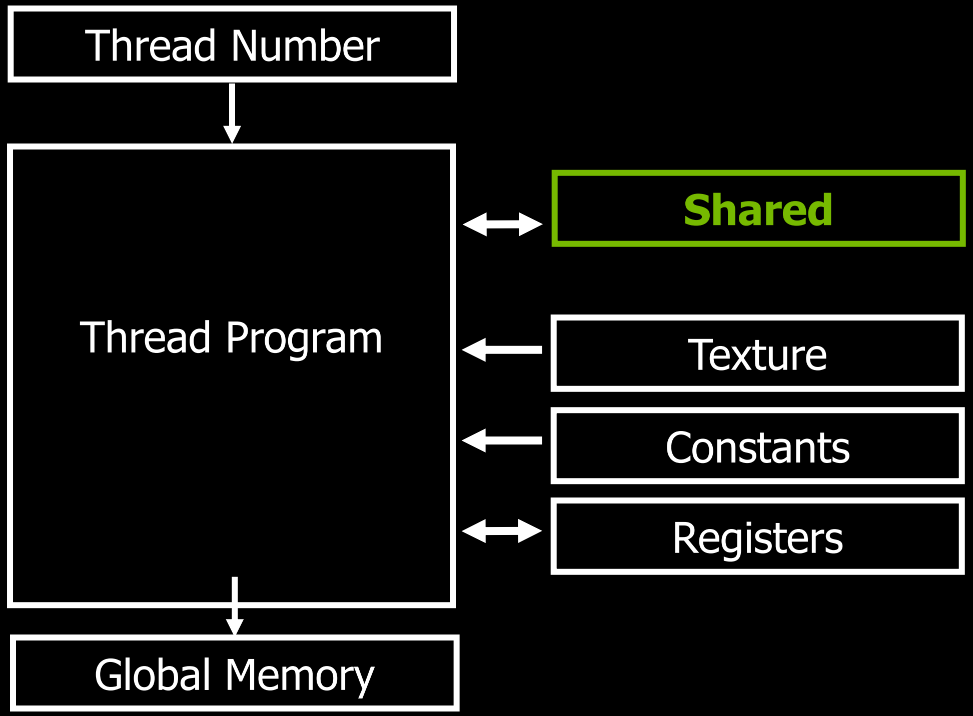
Global Memory



Features

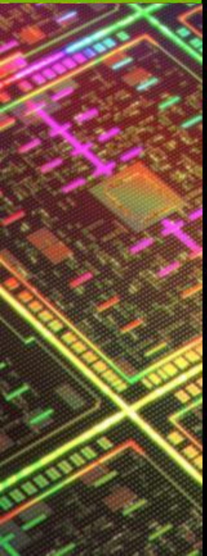
- Fully general load/store to GPU memory: Scatter/Gather
- Programmer flexibility on how memory is accessed
- Untyped, not limited to fixed texture types
- Pointer support

Shared Memory



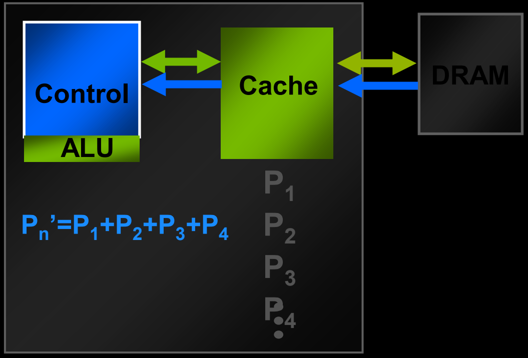
Features

- Dedicated on-chip memory
- Shared between threads for inter-thread communication
- Explicitly managed
- As fast as registers



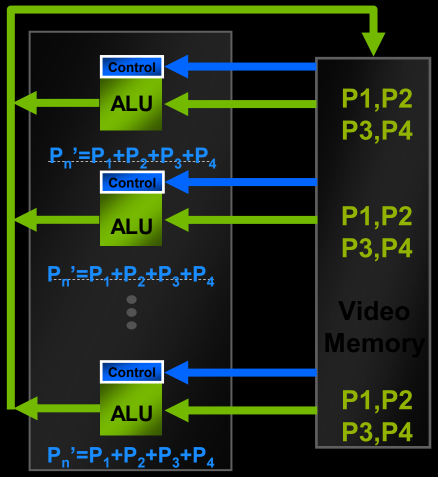
Shared Memory

CPU



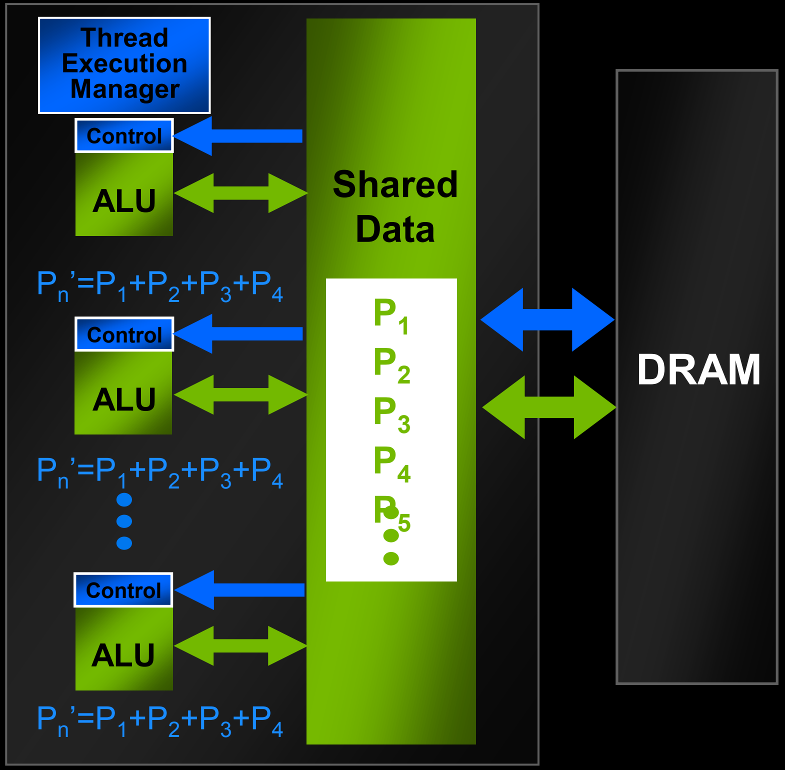
Single thread out of cache

GPGPU



Multiple passes through video memory

CUDA GPU Computing



🟩 Data/Computation

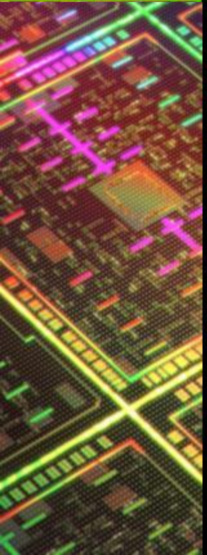
🟦 Program/Control

CUDA: C on the GPU

- A simple, explicit programming language solution
- Extend only where necessary

```
__global__ void KernelFunc(...);

__shared__ int SharedVar;

KernelFunc<<< 500, 128 >>>(...);
```

- Explicit GPU memory allocation
  - `cudaMalloc(), cudaFree()`
- Memory copy from host to device, etc.
  - `cudaMemcpy(), cudaMemcpy2D(), ...`



CUDA: Threading in Data Parallel

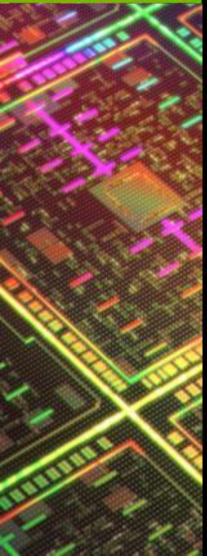
- Threading in a data parallel world
  - Operations drive execution, not data
- Users simply given thread id
  - They decide what thread access which data element
  - One thread = single data element or block or variable or nothing….
  - No need for accessors, views, or built-ins
- Flexibility
  - Not requiring the data layout to force the algorithm
  - Blocking computation for the memory hierarchy (shared)
  - Think about the algorithm, not the data

Divergence in Parallel Computing

- Removing divergence pain from parallel programming

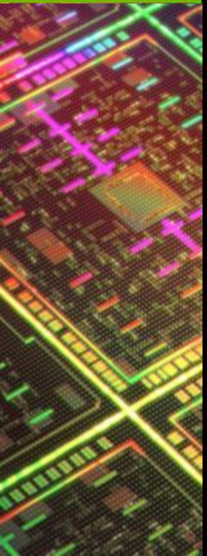
- SIMD Pain
  - User required to SIMD-ify
  - User suffers when computation goes divergent

- GPUs: Decouple execution width from programming model
  - Threads can diverge freely
  - Inefficiency only when granularity exceeds native machine width
  - Hardware managed
  - Managing divergence becomes performance optimization
  - Scalable



Building GPU Computing Ecosystem

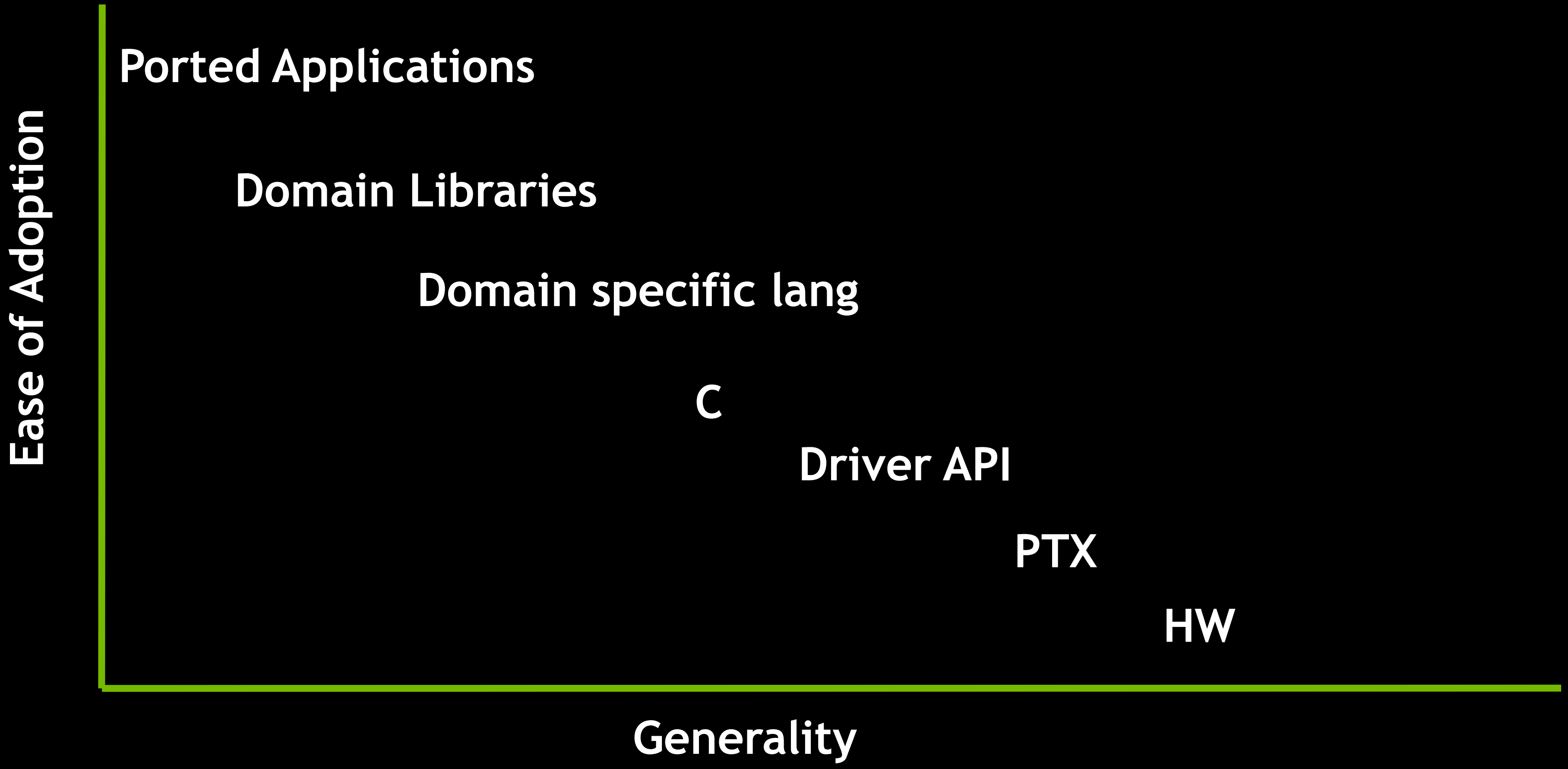
- Convince the world to program an entirely new kind of processor
- Tradeoffs between functional vs. performance requirements
- Deliver HPC feature parity
- Seed larger ecosystem with foundational components



GPU Computing By the Numbers:

>350,000,000	Compute Capable GPUs
>1,000,000	Toolkit Downloads
>120,000	Active CUDA Developers
>450	Universities Teaching CUDA
100%	OEMs offer CUDA GPU PCs

Customizing Solutions



CUDA Math Libraries

High performance math routines for your applications:

- cuFFT – Fast Fourier Transforms Library
- cuBLAS – Complete BLAS Library
- cuSPARSE – Sparse Matrix Library
- cuRAND – Random Number Generation (RNG) Library
- NPP – Performance Primitives for Image & Video Processing
- Thrust – Templated Parallel Algorithms & Data Structures
- math.h - C99 floating-point Library

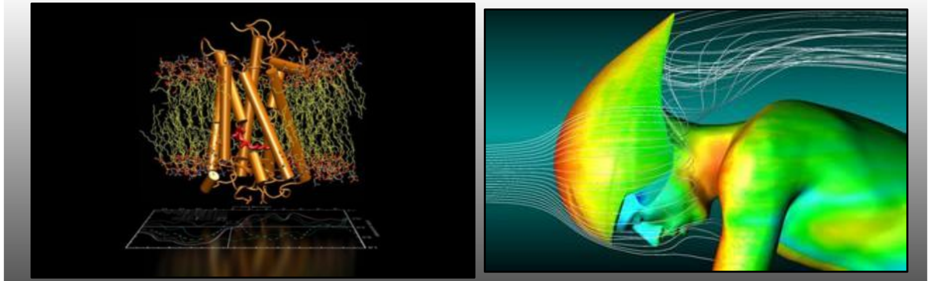
Included in the CUDA Toolkit Free download @ www.nvidia.com/getcuda

More information on CUDA libraries:

http://www.nvidia.com/object/gtc2010-presentation-archive.html#session2216

Education & Research

Domain Science

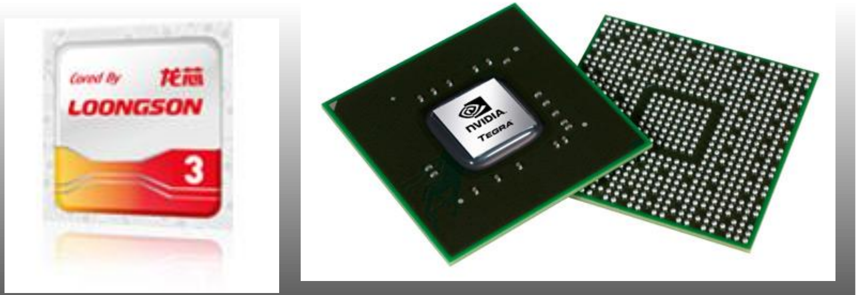


Language and Compilers

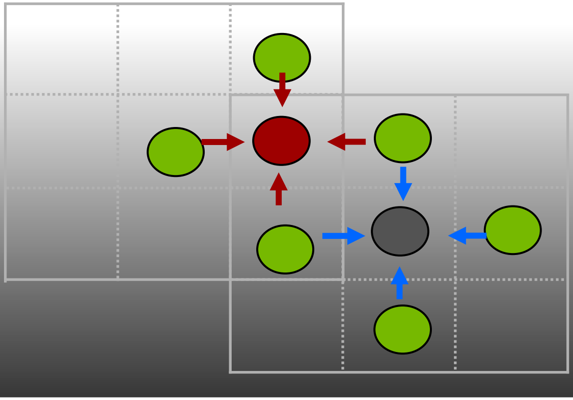
LLVM COMPILER INFRASTRUCTURE

Thrust

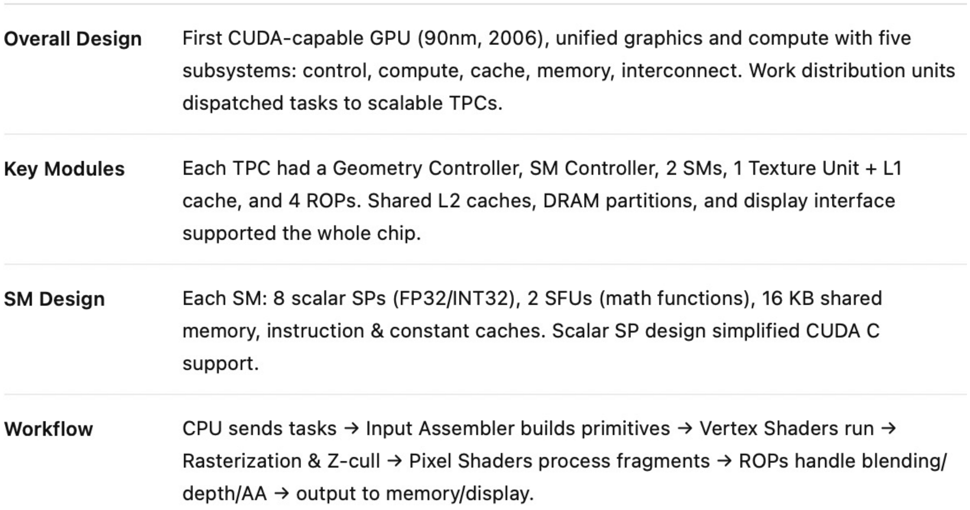
Heterogeneous Architectures



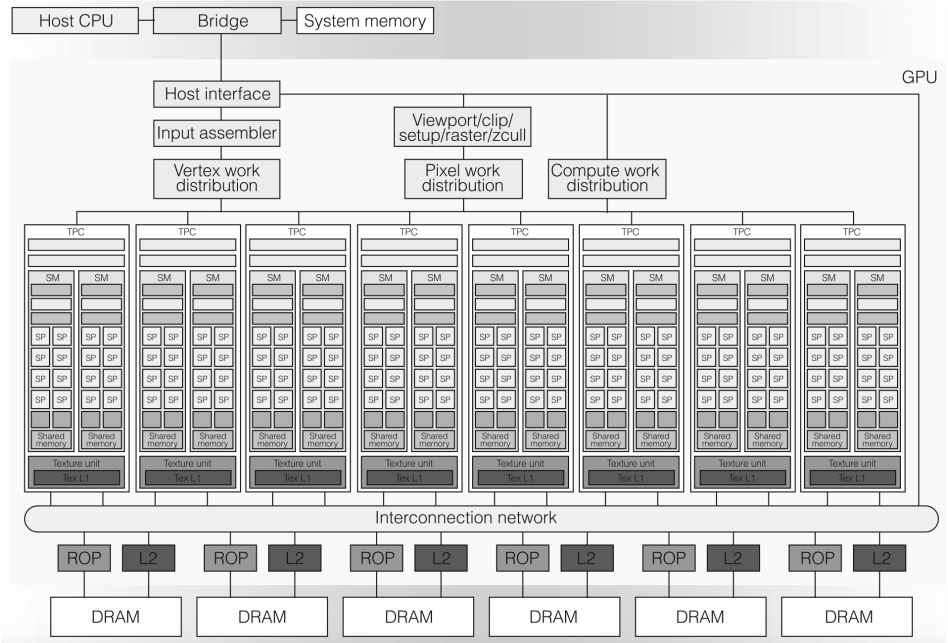
Computer Science



Tesla 2006: Unified shader architecture

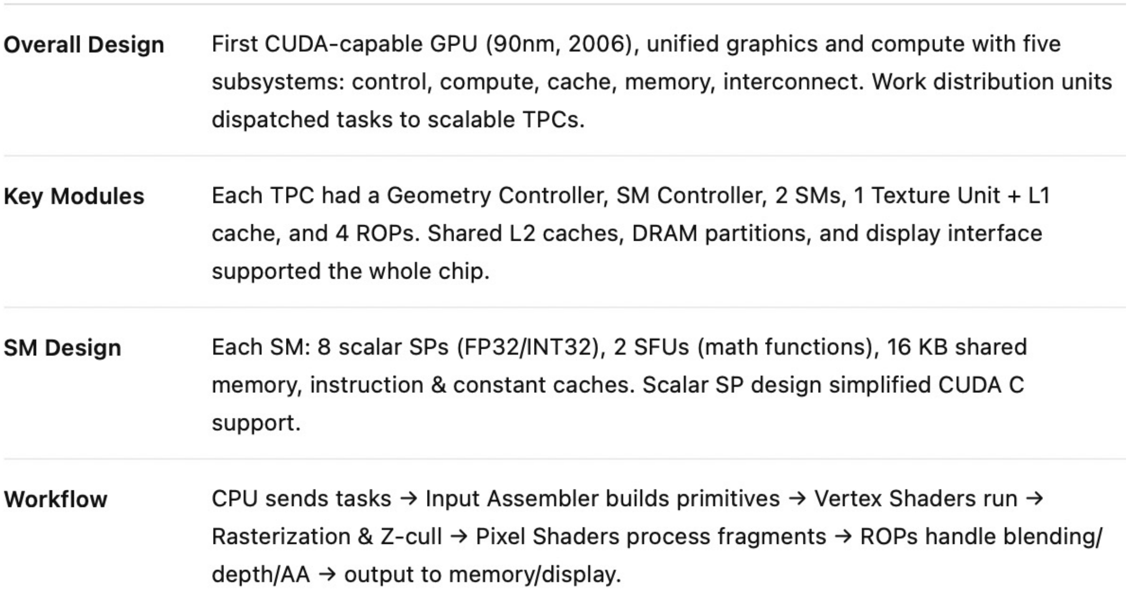


Overall Design	First CUDA-capable GPU (90nm, 2006), unified graphics and compute with five subsystems: control, compute, cache, memory, interconnect. Work distribution units dispatched tasks to scalable TPCs.
Key Modules	Each TPC had a Geometry Controller, SM Controller, 2 SMs, 1 Texture Unit + L1 cache,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

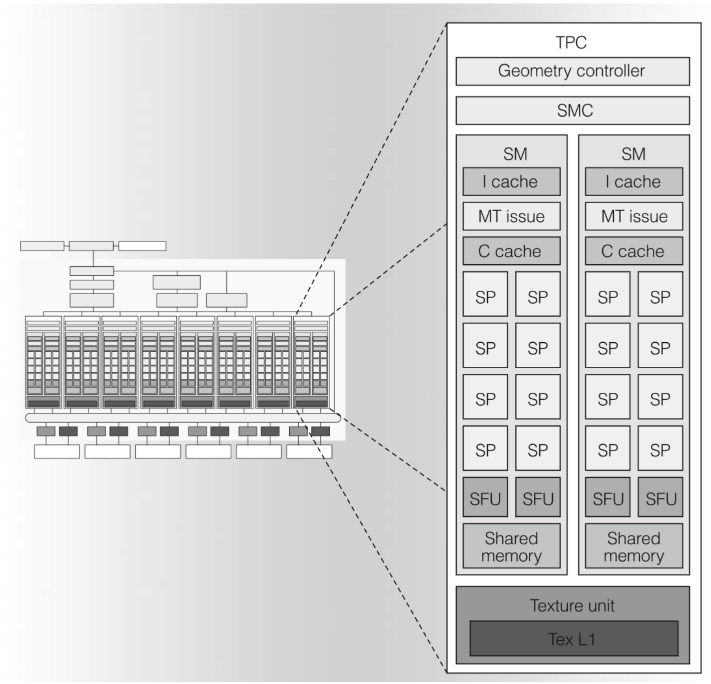


In 2006, NVIDIA introduced the GeForce 8800. This design featured a “unified shader architecture” with 128 processing elements distributed among eight shader cores. Each shader core could be assigned to any shader task, eliminating the need for stage-by-stage balancing and greatly improving overall performance.

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.



Starting from now, software developers can write C code on GPU.

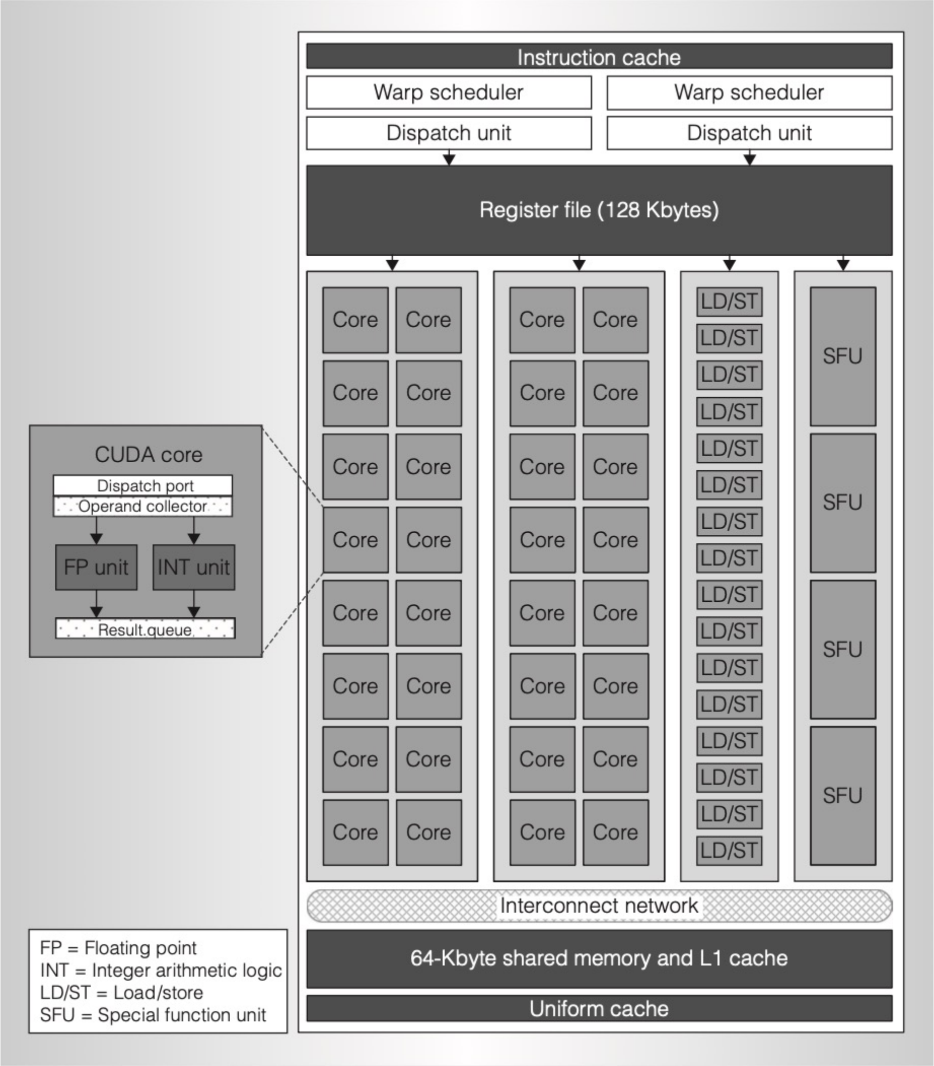
Fermi 2010

Streaming Multiprocessors (SM): Each SM includes 32 CUDA processor cores, 16 load/ store units, and four special function units (SFUs). It also possesses a 64-Kbyte configurable shared memory+L1 cache, 128-Kbyte register file, instructions cache, and two multi-threaded wrap schedulers and two instruction dispatch units.

CUDA Cores: Each pipelined CUDA core executes an instruction per clock for a thread. With 32 cores architecture, an SM can execute up to 32 thread instructions per clock. Executable instructions include scalar floating-point instruction, implemented by floating-point unit (FP unit), and integer instruction, implemented by integer unit (INT unit).

Special functional units (SFU): The SFUs are in charge of executing 32-bit floating-point instructions for fast approximations of reciprocal, reciprocal square root, sin, cos, exp, and log functions. The approximations are precise to better than 22 mantissa bits.

Load/store units: The streaming multiprocessor load/store units execute load, store, and atomic memory access instructions. A warp of 32 active threads presents 32 individual byte addresses, and the instruction accesses each memory address. The load/store units coalesce 32 individual thread accesses into a minimal number of memory block accesses.



CUDA Core

CUDA Cores: General-purpose shader cores in NVIDIA GPUs, designed to handle a wide range of programmable tasks, including graphics rendering, general-purpose computing (GPGPU), and traditional compute workloads.

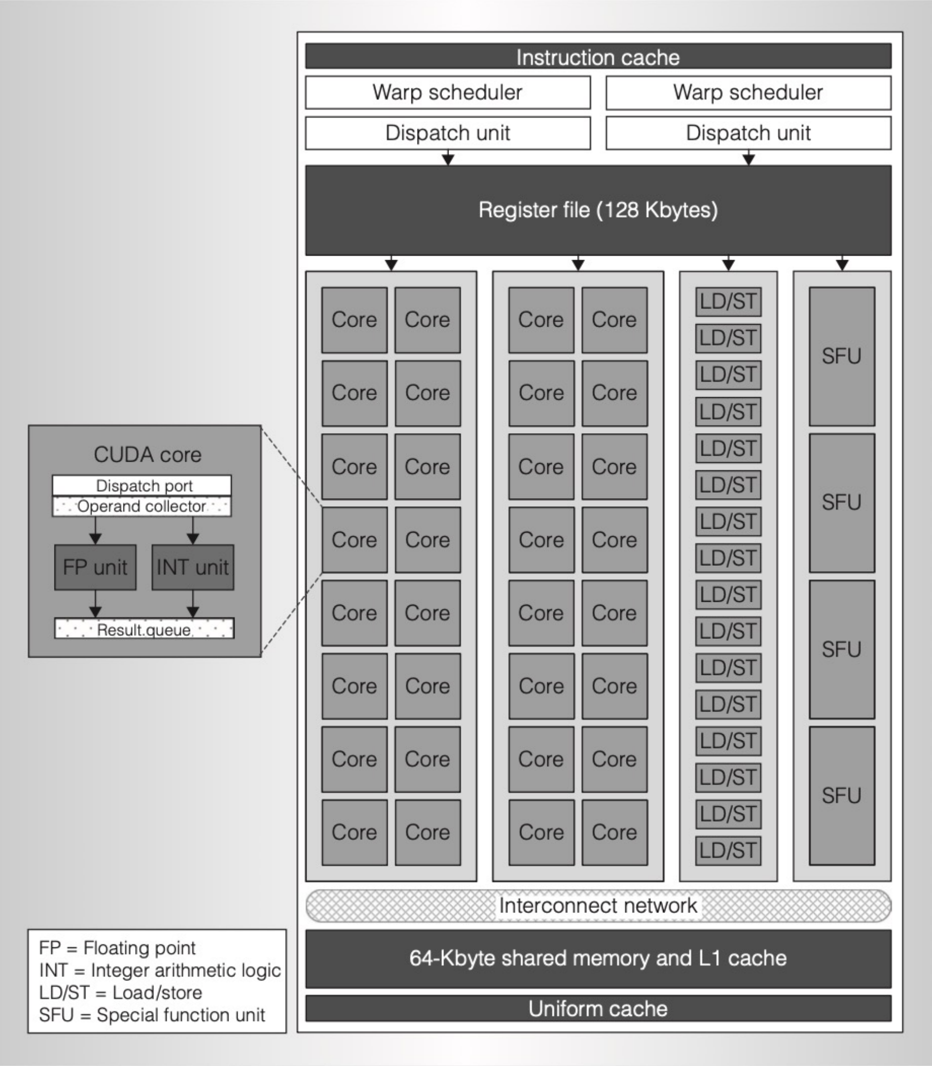
Purpose: The backbone of NVIDIA's unified shader architecture, introduced with the GeForce 8 series in 2006. They execute a variety of instructions, including floating-point (FP32, FP64), integer, and bitwise operations.

Structure: Each CUDA core is a scalar processor capable of performing one operation per clock cycle on a single data element. Multiple CUDA cores are grouped into Streaming Multiprocessors (SMs), which manage thread scheduling and shared memory.

Instruction Set: Supports a broad instruction set, enabling flexibility for graphics (e.g., vertex, geometry, fragment shaders) and compute tasks (e.g., physics simulations, ray tracing).

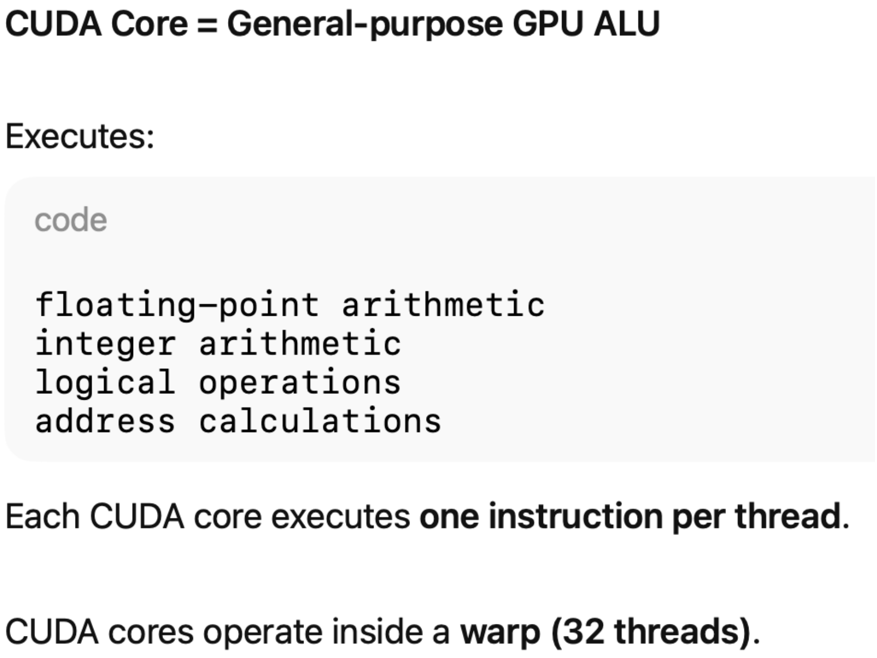
Precision: Primarily FP32 (single-precision floating-point) with limited FP64 (double-precision) support, depending on the GPU generation (e.g., higher-end GPUs like the A100 offer robust FP64).

Limitations: Less efficient for matrix-intensive operations due to lack of specialized hardware, relying on software libraries (e.g., cuBLAS) to optimize such workloads.



CUDA Core

CUDA Cores: General-purpose shader cores in NVIDIA GPUs, designed to handle a wide range of programmable tasks, including graphics rendering, general-purpose computing (GPGPU), and traditional compute workloads.



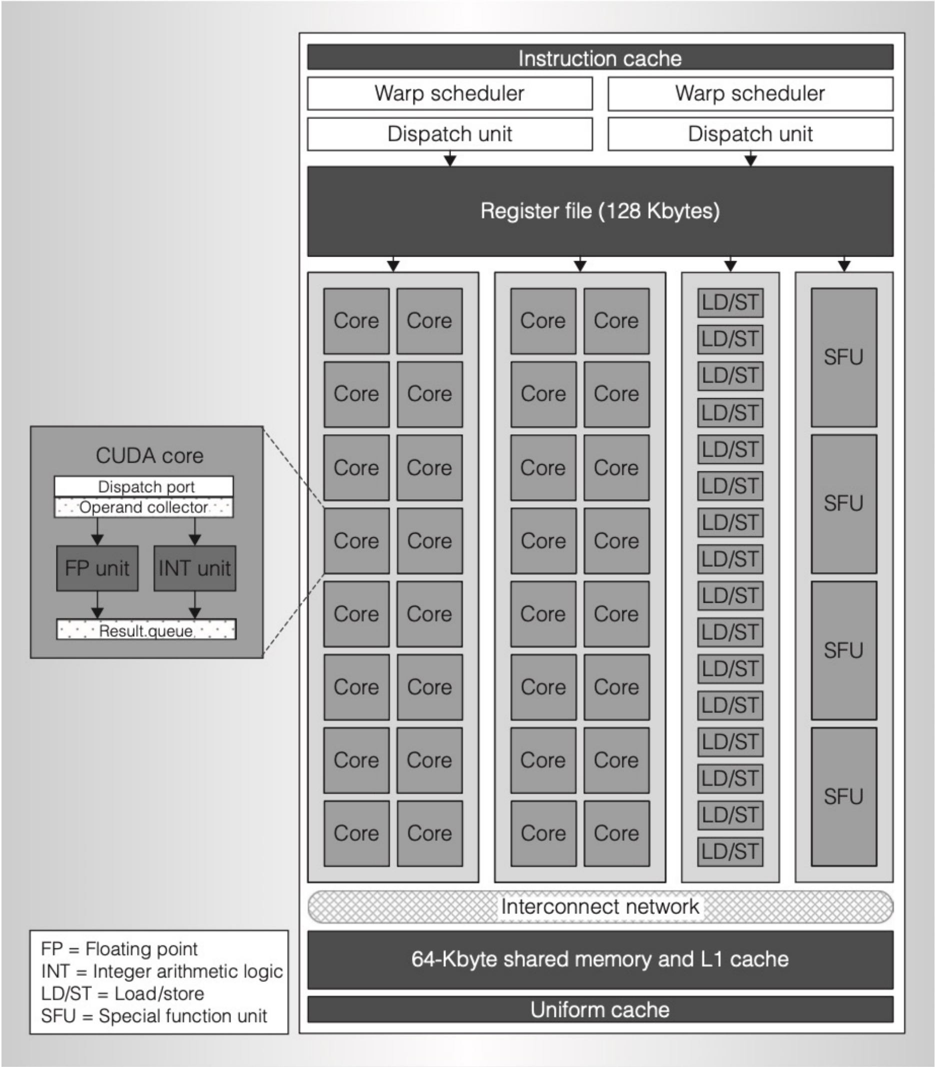
CUDA Core = General-purpose GPU ALU

Executes:

```
floating-point arithmetic
integer arithmetic
logical operations
address calculations
```

Each CUDA core executes **one instruction per thread.**

CUDA cores operate inside a **warp (32 threads)**.



CUDA Core

CUDA Cores: General-purpose shader cores in NVIDIA GPUs, designed to handle a wide range of programmable tasks, including graphics rendering, general-purpose computing (GPGPU), and traditional compute workloads.

CUDA cores execute threads using **SIMT**

code

```
Warp = 32 threads
All threads execute same instruction
```

Example kernel:

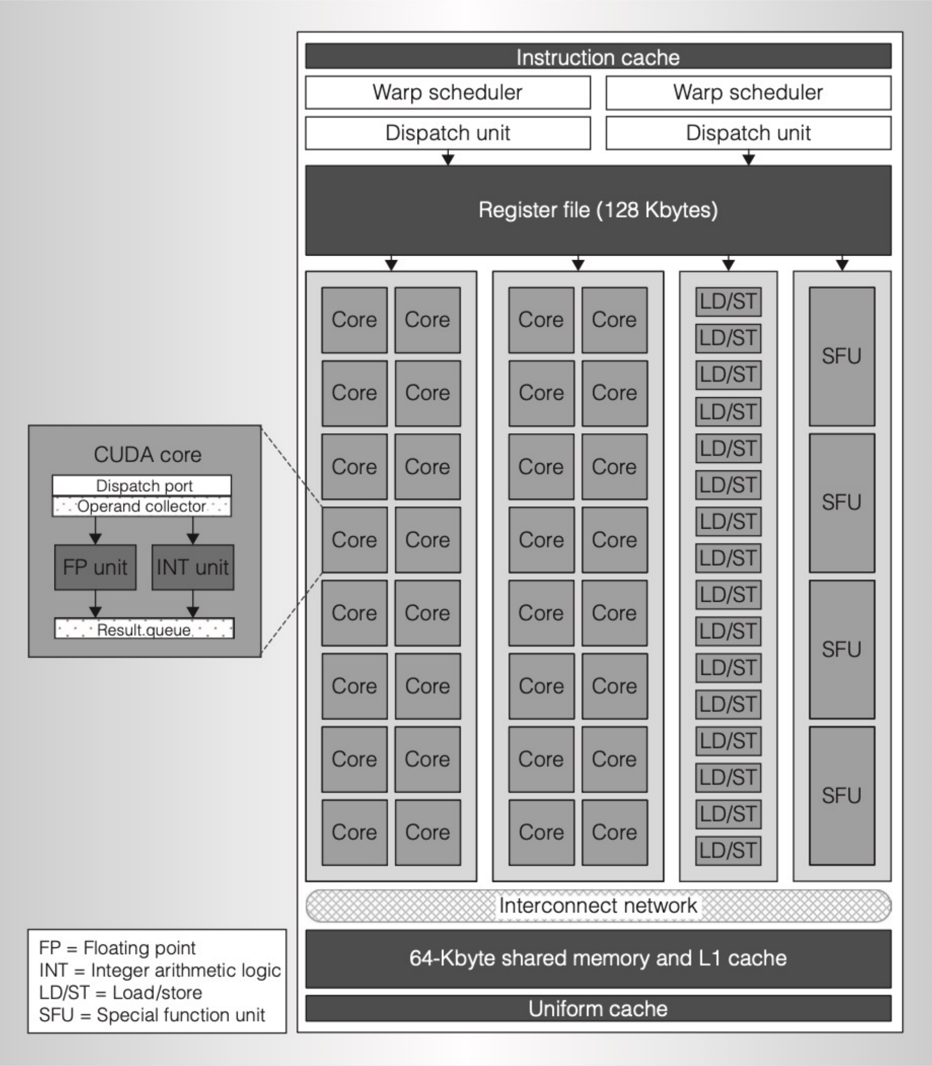
code

```
C[i] = A[i] + B[i]
```

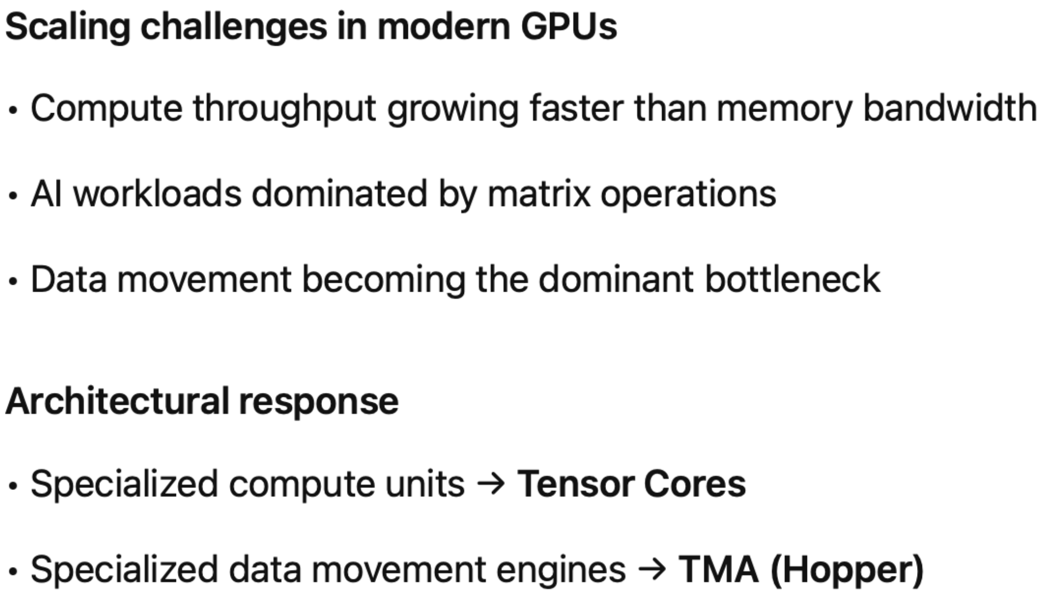
Executed as:

code

```
32 threads
32 elements
32 CUDA cores
```



Motivation: Why New GPU Unites?



Scaling challenges in modern GPUs

- Compute throughput growing faster than memory bandwidth
- AI workloads dominated by matrix operations
- Data movement becoming the dominant bottleneck

Architectural response

- Specialized compute units → **Tensor Cores**
- Specialized data movement engines → **TMA (Hopper)**

Tensor Core

Tensor Core = Matrix Multiply Accelerator

Performs operation:

$D = A \times B + C$

Where A, B, C, D are **small matrices**

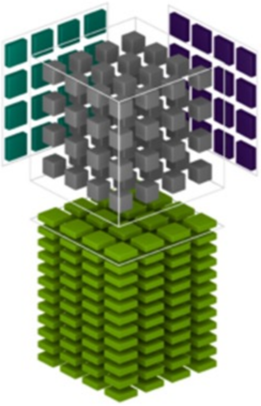
Benefits:

- Much higher FLOPS per area
- Higher energy efficiency
- Optimized for deep learning workloads

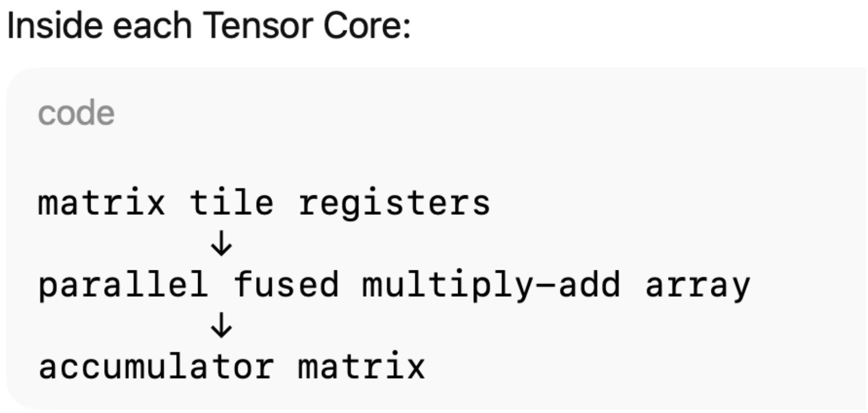
Tensor Core

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

HMMA	FP16 or FP32	FP16	FP16	FP16 or FP32
IMMA	INT32	INT8 or UINT8	INT8 or UINT8	INT32



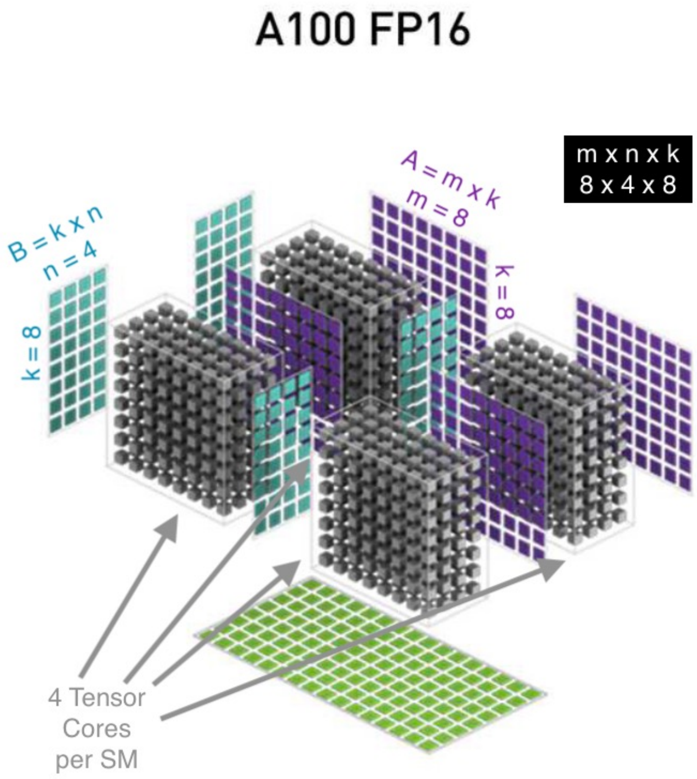
Tensor Core



Inside each Tensor Core:

code

```
matrix tile registers
         ↓
parallel fused multiply-add array
         ↓
accumulator matrix
```



Tensor Core

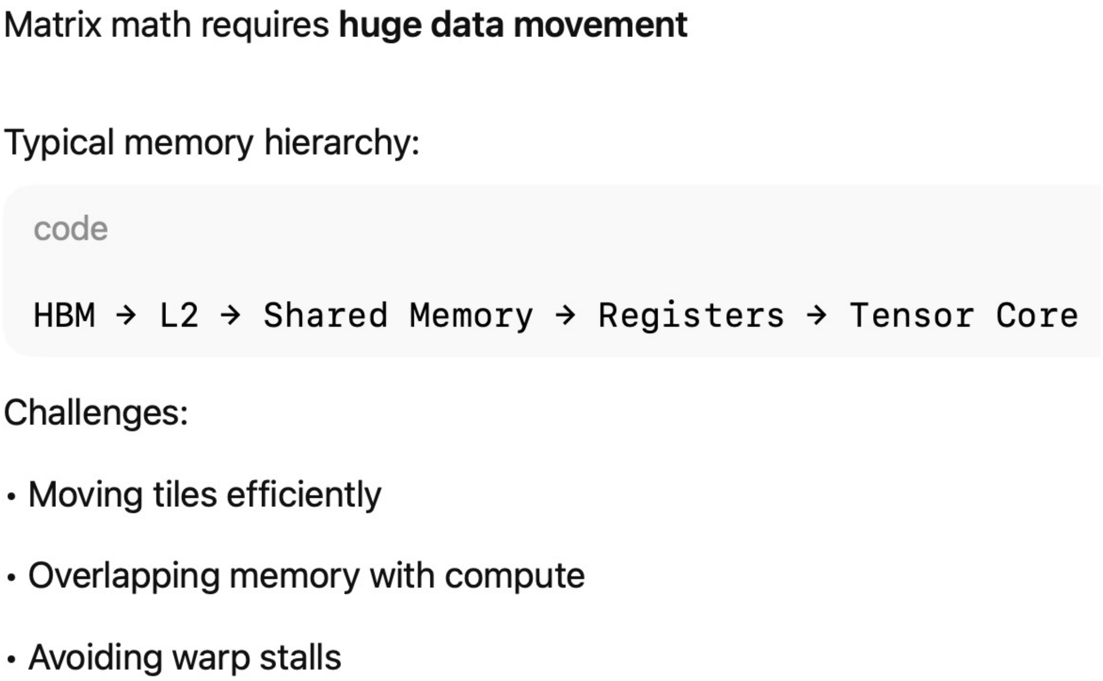
Feature	CUDA Core	Tensor Core
Operation	scalar/vector math	matrix multiply
Parallelism	thread-level	matrix tile
Use cases	general compute	AI / linear algebra

Example peak compute (H100):

Unit	Performance
FP32 CUDA cores	~60 TFLOPS
Tensor cores	**>1000 TFLOPS (FP16/FP8)**

```
CUDA core: a * b + c
Tensor core: A × B + C
```

The New Bottleneck: Data Movement



Matrix math requires **huge data movement**

Typical memory hierarchy:

```code
HBM → L2 → Shared Memory → Registers → Tensor Core
```

Challenges:

- Moving tiles efficiently
- Overlapping memory with compute
- Avoiding warp stalls

Tensor Memory Accelerator (TMA)

TMA introduced in NVIDIA Hopper (H100)

Purpose:

Efficiently move tensor tiles between:

code

```
Global Memory ↔ Shared Memory
```

Key features:

- asynchronous data movement
- multidimensional tensor copies
- hardware address generation
- minimal thread overhead

Tensor Memory Accelerator (TMA)

Before TMA:

Threads copy data manually.

code

```
for each thread:
    load element
    store to shared memory
```

Problems:

- many instructions
- high register pressure
- synchronization overhead

TMA workflow:

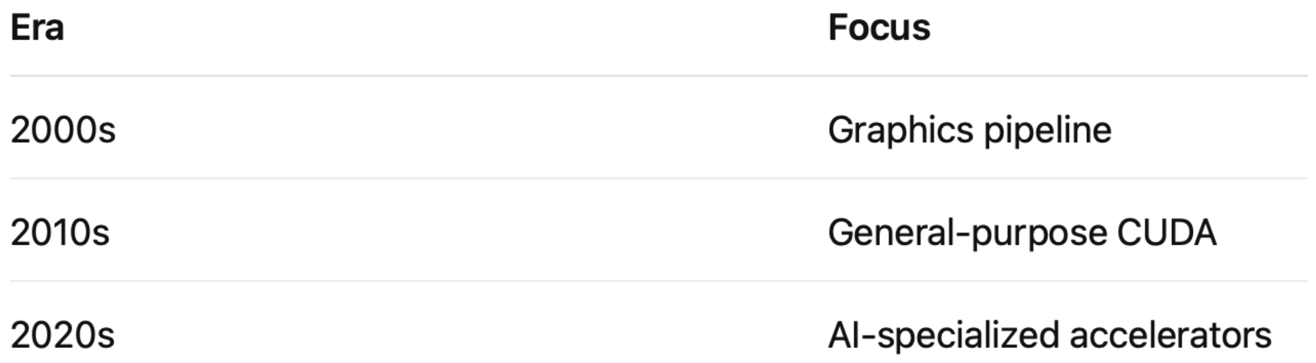
code

```
1. Define tensor layout
2. Launch asynchronous copy
3. Hardware transfers tile
4. Tensor core computation continues
```

Benefits:

- hardware DMA-like transfer
- overlap compute and memory
- reduced instruction count

Tensor Core + TMA and Beyond



Era	Focus
2000s	Graphics pipeline
2010s	General-purpose CUDA
2020s	AI-specialized accelerators

FMA: Fused Multiply-Add

FMA: Fundamental arithmetic instruction

$$d = a \times b + c$$

Key properties:

- Executes **multiply + add in one instruction**
- **Two floating-point operations per instruction**
- Reduces rounding error
- Core operation in GPU ALUs

Hardware:

code

```
CUDA Core
```

Example:

code

```
d = a * b + c
```

HDP4: Half-Precision Dot Product

HDP4: 4-element dot product instruction

$$d = a_0b_0 + a_1b_1 + a_2b_2 + a_3b_3$$

Computes **four multiplications + additions in one instruction**

Purpose:

- vector math acceleration
- neural network inference
- signal processing

Example:

code

```
dp4(a, b)
```

Where:

code

```
a = (a0, a1, a2, a3)
b = (b0, b1, b2, b3)
```

HMMA: Tensor Core Matrix Multiply

HMMA: Half-precision Matrix Multiply-Accumulate

Operation:

$$D = A \times B + C$$

```
for i,j,k:
    C[i][j] += A[i][k] * B[k][j]
```

Benefits:

- massive parallelism
- extremely high FLOPS
- optimized for deep learning training

code

```
A, B, C, D = matrix tiles
```

Typical tile size:

code

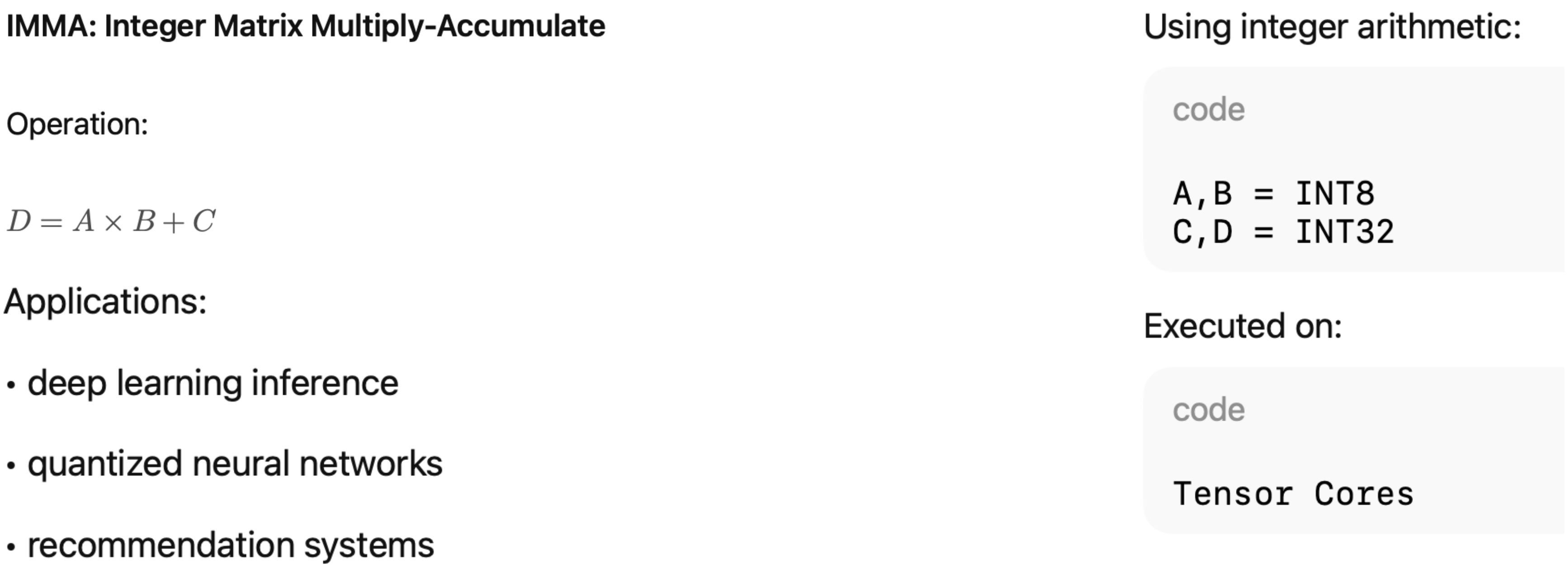
```
16 × 16 × 16
```

Executed on:

code

```
Tensor Cores
```

IMMA: Integer Matrix Multiply



IMMA: Integer Matrix Multiply-Accumulate

Operation:

$$D = A \times B + C$$

Applications:

- deep learning inference
- quantized neural networks
- recommendation systems

Using integer arithmetic:

```
code

A,B = INT8
C,D = INT32
```

Executed on:

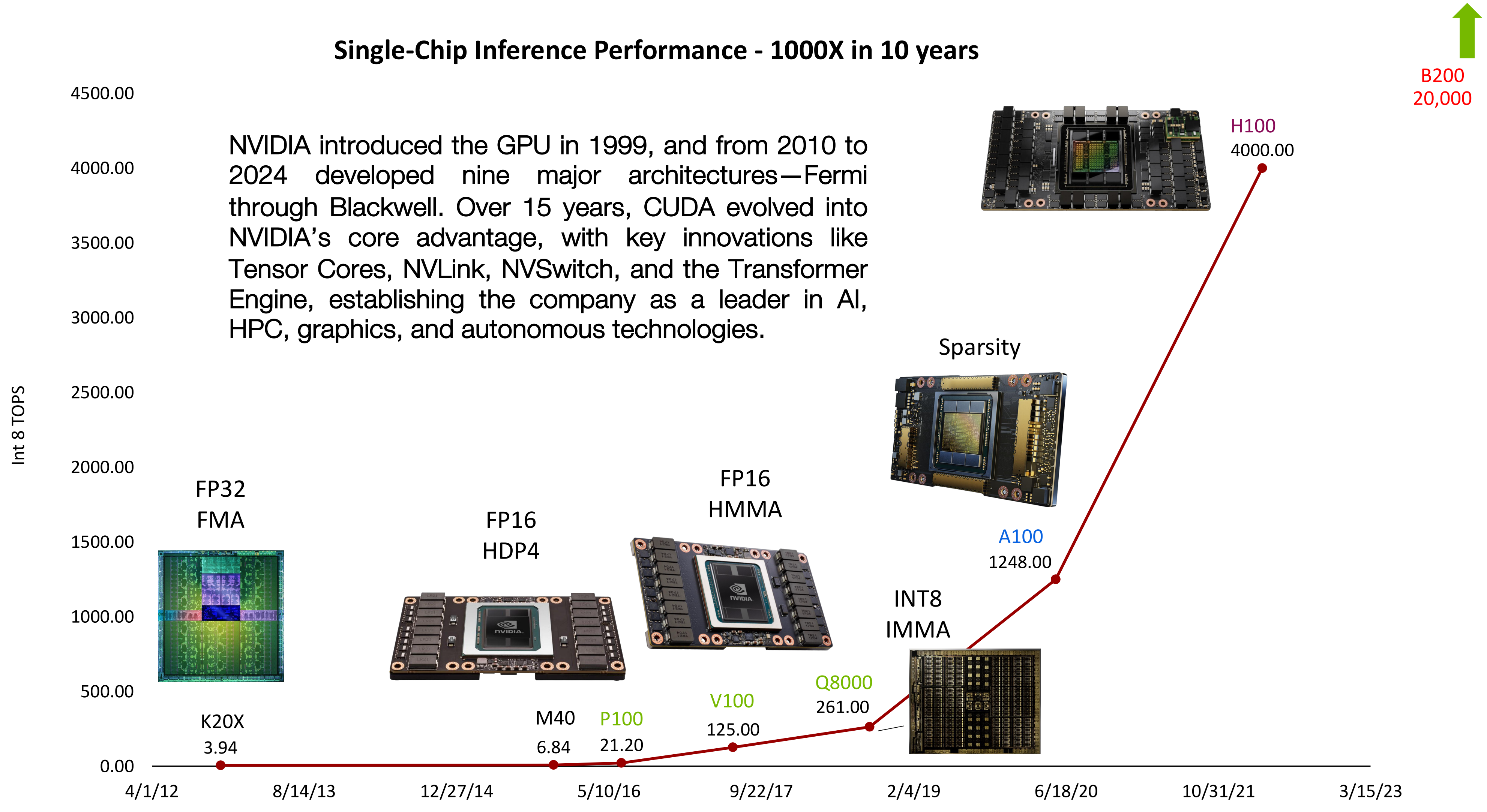
```
code

Tensor Cores
```

Evolution of GPU Compute Primitives

```
FMA   → scalar arithmetic
HDP4  → vector dot product
HMMA  → matrix multiply
IMMA  → quantized matrix multiply
```

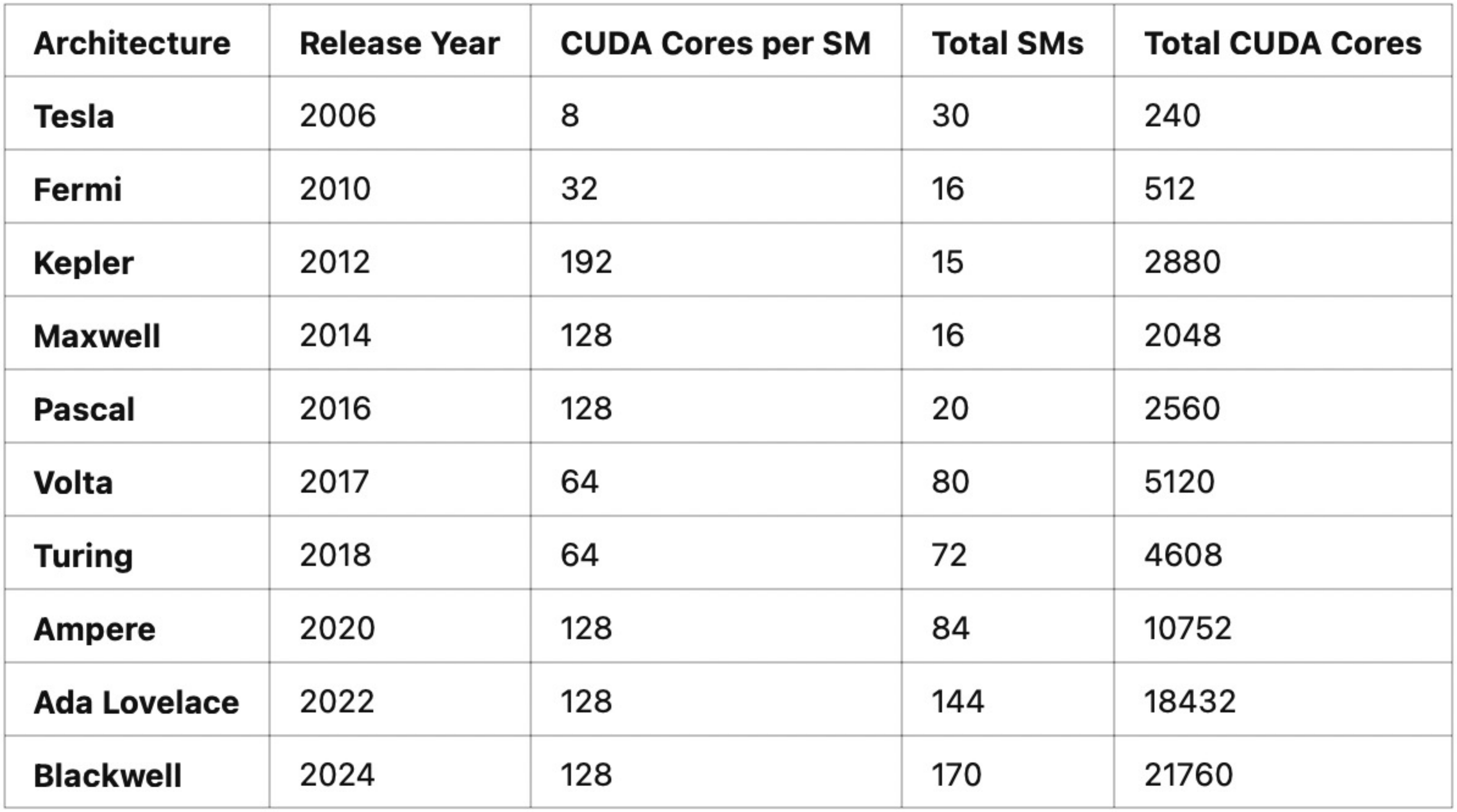
Generation	Primitive	Hardware
2000s	FMA	CUDA cores
2010s	DP4 / HDP4	vector ALUs
2017+	HMMA	Tensor cores
2018+	IMMA	Tensor cores

Single-Chip Inference Performance - 1000X in 10 years



NVIDIA introduced the GPU in 1999, and from 2010 to 2024 developed nine major architectures—Fermi through Blackwell. Over 15 years, CUDA evolved into NVIDIA’s core advantage, with key innovations like Tensor Cores, NVLink, NVSwitch, and the Transformer Engine, establishing the company as a leader in AI, HPC, graphics, and autonomous technologies.

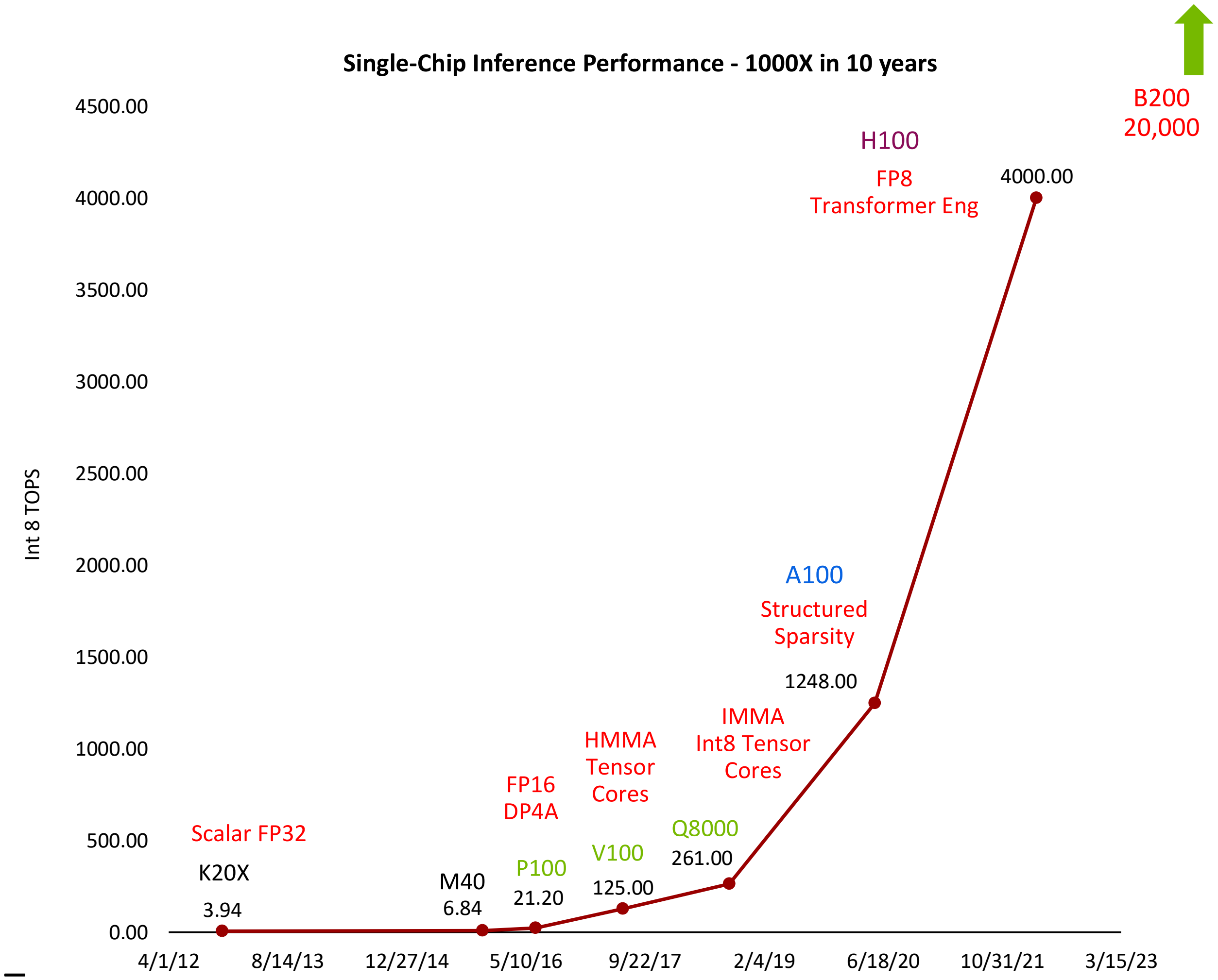
Increasing Massive Parallelism



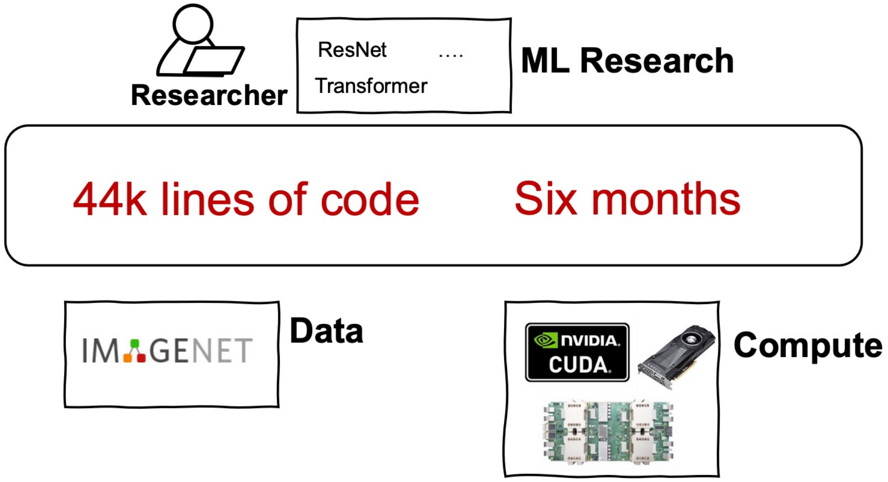
Architecture	Release Year	CUDA Cores per SM	Total SMs	Total CUDA Cores
Tesla	2006	8	30	240
Fermi	2010	32	16	512
Kepler	2012	192	15	2880
Maxwell	2014	128	16	2048
Pascal	2016	128	20	2560
Volta	2017	64	80	5120
Turing	2018	64	72	4608
Ampere	2020	128	84	10752
Ada Lovelace	2022	128	144	18432
Blackwell	2024	128	170	21760

Gains from

- Number Representation
  - FP32, FP16, Int8, FP4
  - (TF32, BF16)
  - ~16x, 32x
- Complex Instructions
  - DP4, HMMA, IMMA
  - ~12.5x
- Process
  - 28nm, 16nm, 7nm, 5nm, 4nm
  - ~2.5x, 3x
- Sparsity ~2x
- Die Size 2x
- Model efficiency has also improved – overall gain > 1000x


Single-Chip Inference Performance - 1000X in 10 years

The Programming Challenge



Researcher

ResNet ....

Transformer

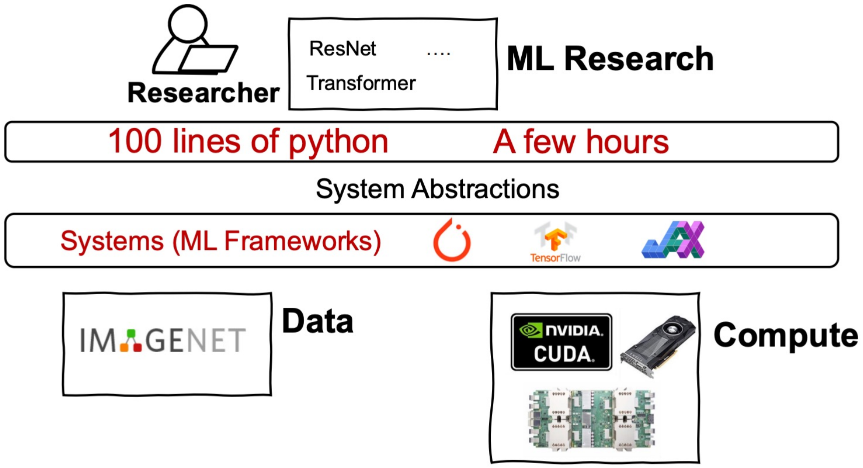
ML Research

44k lines of code

Six months

Data

Compute



Researcher

ResNet ....

Transformer

ML Research

100 lines of python

A few hours

System Abstractions

Systems (ML Frameworks)

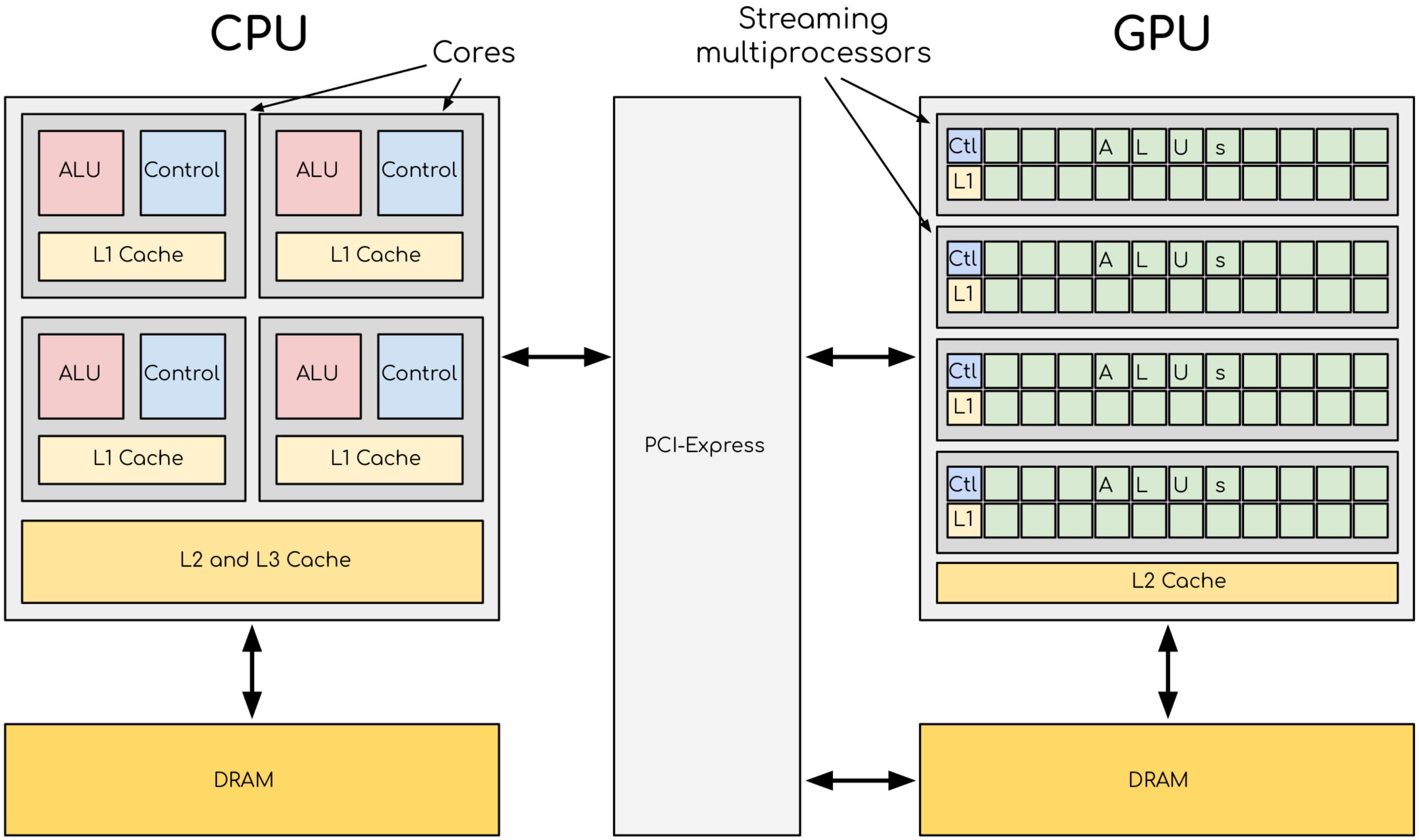
Data

Compute

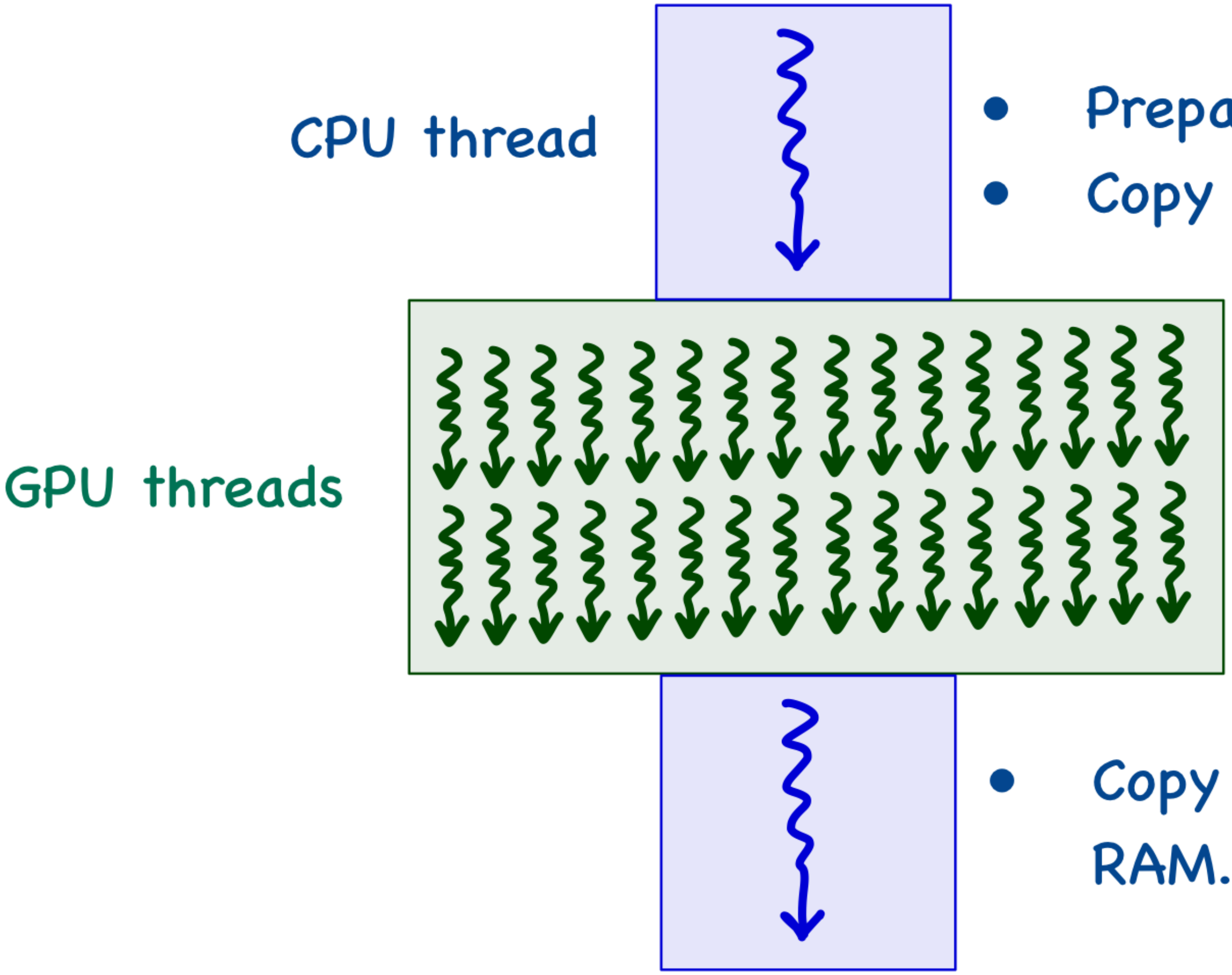
CUDA Compute Unified Device Architecture

- **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.

How to program GPGPU?



GPGPU as a really powerful accelerator



- Prepare data (matrices A, B, C).
- Copy data from RAM to VRAM.

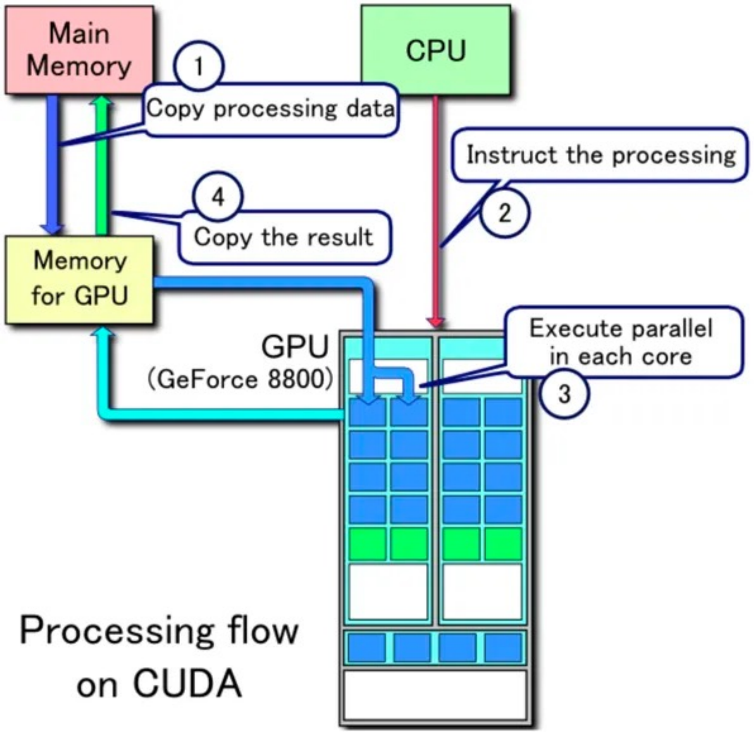
- Parallel Matrix Multiplication.

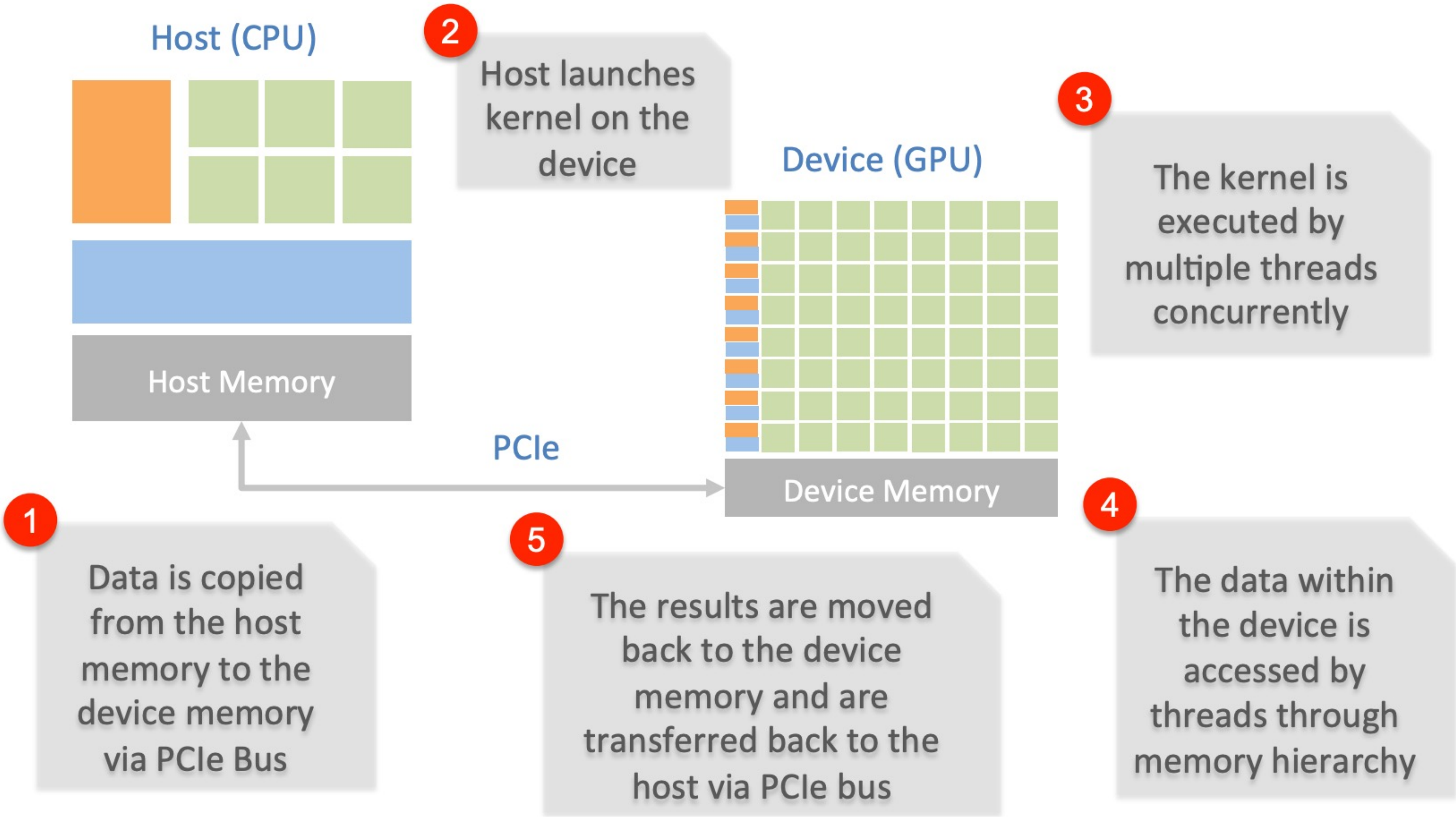
- Copy results from VRAM to RAM.

CUDA

A typical CUDA program processing flow is as follows: Since GPU memory and CPU main memory are physically separate, the host program (Host Code) must first allocate both main memory and GPU memory space. Before the GPU can execute the Device Code, the code and data must be transferred from main memory to GPU memory. Only then can the GPU begin executing the kernel function, as illustrated below:

1. The Host Code copies the Device Code and data from main memory into GPU memory.
2. The Host Code launches the kernel.
3. The GPU executes the kernel function in parallel, accessing data from GPU memory during execution.
4. Once execution is complete, the GPU writes the results from GPU memory back to main memory.





1 Data is copied from the host memory to the device memory via PCIe Bus

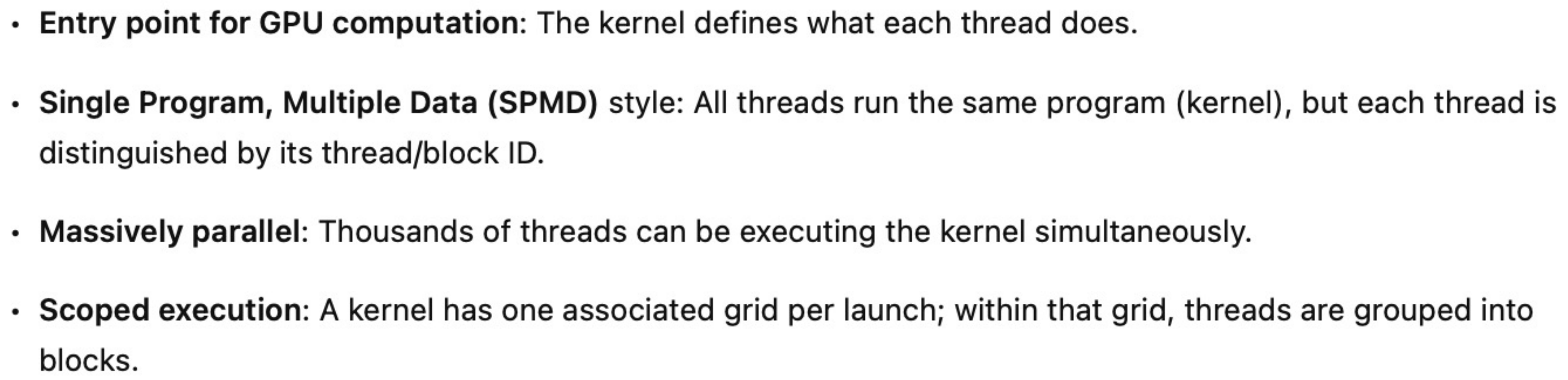
2 Host launches kernel on the device

3 The kernel is executed by multiple threads concurrently

4 The data within the device is accessed by threads through memory hierarchy

5 The results are moved back to the device memory and are transferred back to the host via PCIe bus

Kernel



- **Entry point for GPU computation**: The kernel defines what each thread does.
- **Single Program, Multiple Data (SPMD)** style: All threads run the same program (kernel), but each thread is distinguished by its thread/block ID.
- **Massively parallel**: Thousands of threads can be executing the kernel simultaneously.
- **Scoped execution**: A kernel has one associated grid per launch; within that grid, threads are grouped into blocks.

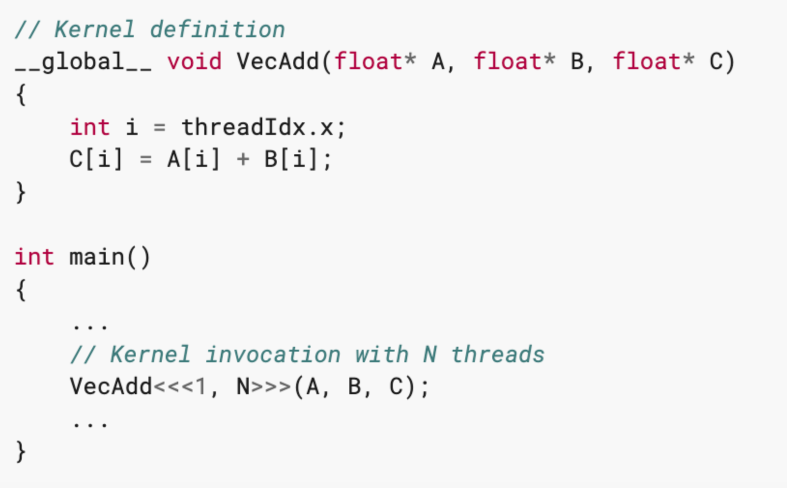
Kernel

A **kernel** is a function written in CUDA C/C++ (or another GPU programming language) that runs on the GPU and is executed **in parallel by many threads**.

- Declared with the `__global__` qualifier.
- Launched from the host (CPU) using the special syntax:

```
myKernel<<<numBlocks, threadsPerBlock>>>(arguments);
```

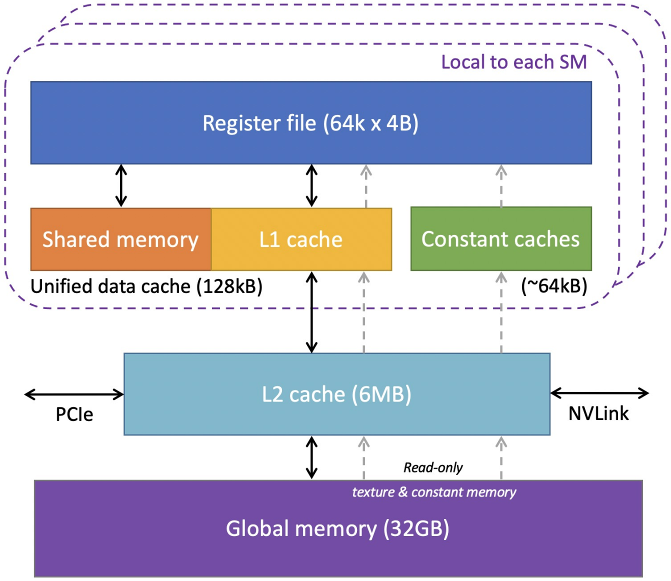
- When launched, CUDA creates a **grid** of threads (organized into blocks), and *each thread executes the same kernel function*, but on different data.



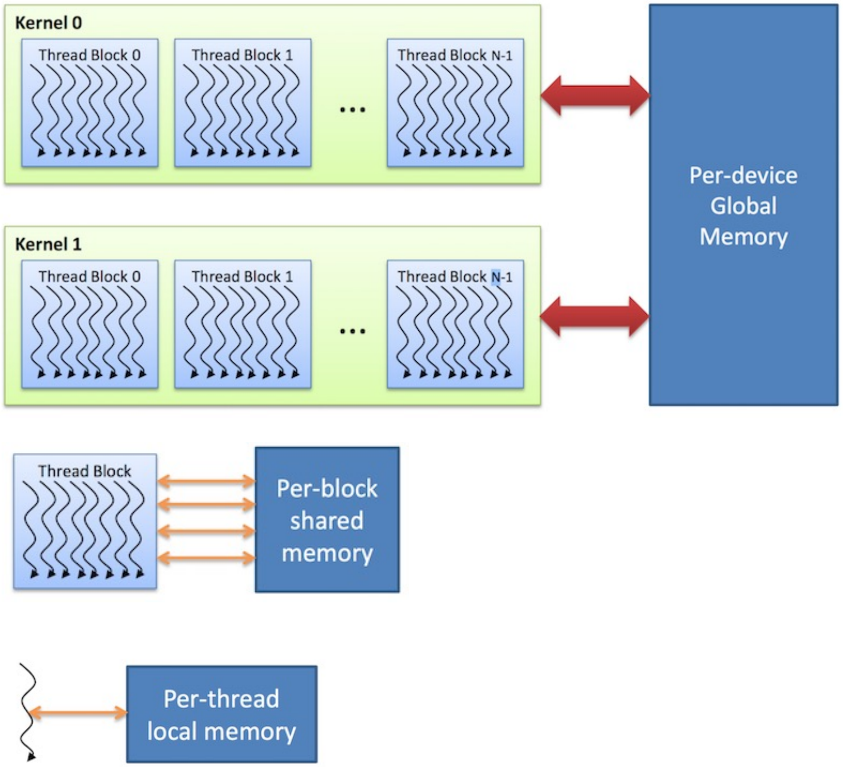
```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

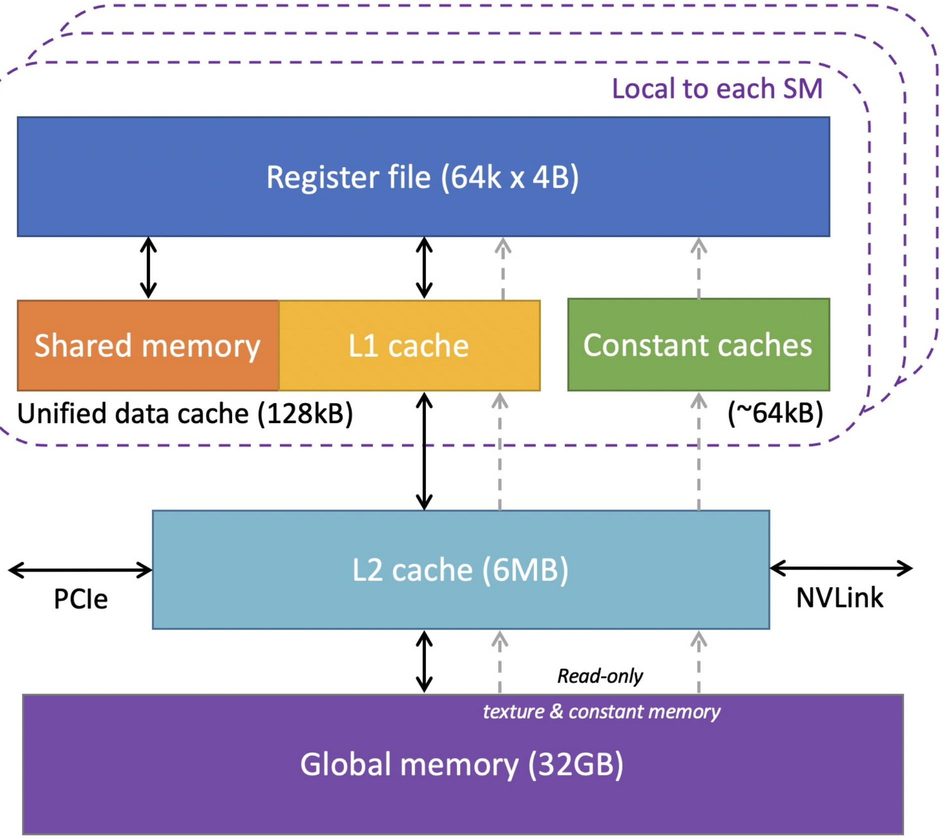
Memory Hierarchy

Level	Location	Latency	Size	Scope
Registers	SM	~1 cycle	very small	per thread
Shared Memory	SM	~20 cycles	~100 KB	per block
L2 Cache	GPU	~200 cycles	tens of MB	whole GPU
Global Memory (HBM)	GPU DRAM	~500–800 cycles	tens of GB	all threads
Host Memory	CPU	very high	large	system



Memory Hierarchy





Kernel

CUDA Concept	Computation Hardware	Memory Hierarchy
Grid	Entire GPU	Global memory (HBM / DRAM)
Block	Streaming Multiprocessor (SM)	Shared memory / L1 cache
Thread	CUDA core (warp lane)	Registers

Key intuition

- **Threads** perform computation using **registers**.
- **Threads in a block** cooperate on the same **SM** and communicate through **shared memory**.
- **Blocks in a grid** execute across the entire **GPU** and communicate via **global memory**.

Kernel

```
// HOST code to queue kernel
simpleKernel <<< dimGrid, dimBlock >>> (N, d_a);
```

__global__ specifies kernel

ThreadIdx & blockIdx determine thread rank that is mapped to array index

```
__global__ void simpleKernel(int N, float *d_a){
    // Convert thread and thread-block indices into array index
    const int n  = threadIdx.x + blockDim.x*blockIdx.x;

    // If index is in [0,N-1] add entries
    if(n<N)
      d_a[n] = n;
}
```

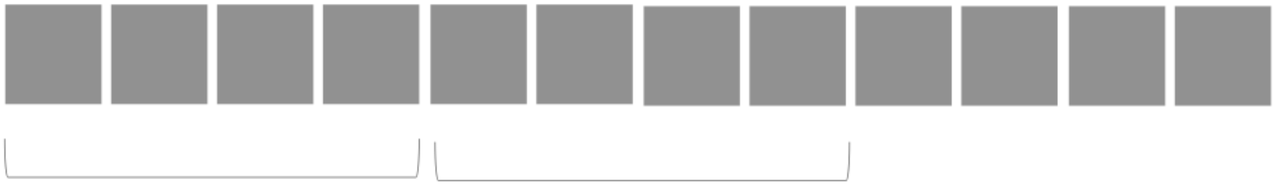
Action performed by each thread

Key observation: the loops are implicitly executed by thread parallelism and *do not* appear in the CUDA kernel code.

Grids, Blocks and Threads

Array traversal

```
int index = threadIdx.x + blockDim.x * blockIdx.x;
```



```
blockDim.x  = 4
blockIdx.x  = 0
threadIdx.x = 0, 1, 2, 3
Index       = 0, 1, 2, 3
```

```
blockDim.x  = 4
blockIdx.x  = 1
threadIdx.x = 0, 1, 2, 3
Index       = 4, 5, 6, 7
```

Grids, Blocks and Threads

A similar approach is used for 3D threads and 2D / 3D grids; can be very useful in 2D / 3D finite difference applications.

How do 2D / 3D threads get divided into warps?

1D thread ID defined by

```
  threadIdx.x +
  threadIdx.y * blockDim.x +
  threadIdx.z * blockDim.x * blockDim.y
```

and this is then broken up into warps of size 32.

Example

```
int main(int argc,char **argv){
    int N = 3789; // size of array for this DEMO

    float *d_a;  // Allocate DEVICE array
    cudaMalloc((void**) &d_a, N*sizeof(float));

    int B = 512;
    dim3 dimBlock(B,1,1);          // 512 threads per thread-block
    dim3 dimGrid((N+B-1)/B, 1, 1); // Enough thread-blocks to cover N

    // Queue kernel on DEVICE
    simpleKernel <<< dimGrid, dimBlock >>> (N, d_a);

    // HOST array
    float *h_a = (float*) calloc(N, sizeof(float));

    // Transfer result from DEVICE array to HOST array
    cudaMemcpy(h_a, d_a, N*sizeof(float), cudaMemcpyDeviceToHost);

    // Print out result from HOST array
    for(int n=0;n<N;++n) printf("h_a[%d] = %f\n", n, h_a[n]);
}
```

1. Allocate array space on DEVICE:

   ```
   float *d_a; // Allocate DEVICE array (pointers used as array handles)
   cudaMalloc((void**) &d_a, N*sizeof(float));
   ```

2. Design thread-array:

   ```
   dim3 dimBlock(512,1,1);          // 512 threads per thread-block
   dim3 dimGrid((N+511)/512, 1, 1); // Enough thread-blocks to cover N
   ```

3. Queue compute task on DEVICE:

   ```
   // specify number of threads with <<< block count, thread count >>>
   simpleKernel <<< dimGrid, dimBlock >>> (N, d_a);
   ```

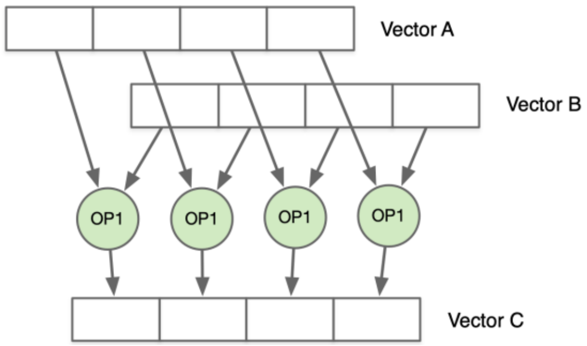
4. Copy results from DEVICE to HOST:

   ```
   float *h_a = (float*) calloc(N, sizeof(float));
   cudaMemcpy(h_a, d_a, N*sizeof(float), cudaMemcpyDeviceToHost)
   ```

SIMT

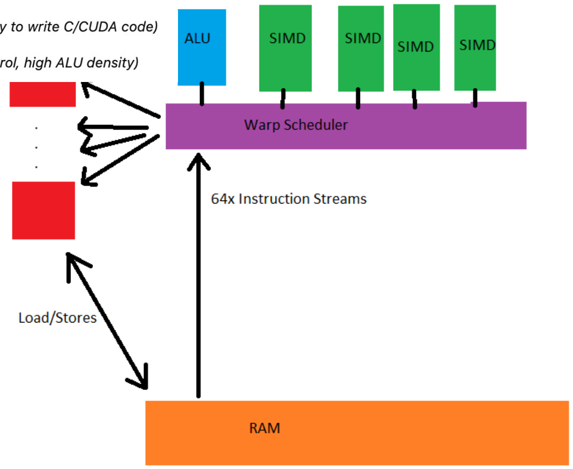
Single instruction, multiple threads (SIMT) is an execution model used in parallel computing where single instruction, multiple data (SIMD) is combined with multithreading. All instructions in all "threads" are executed in lock-step. The SIMT execution model is the primary programming model for GPUs.

$$C[0:3] = A[0:3] \times B[0:3]$$



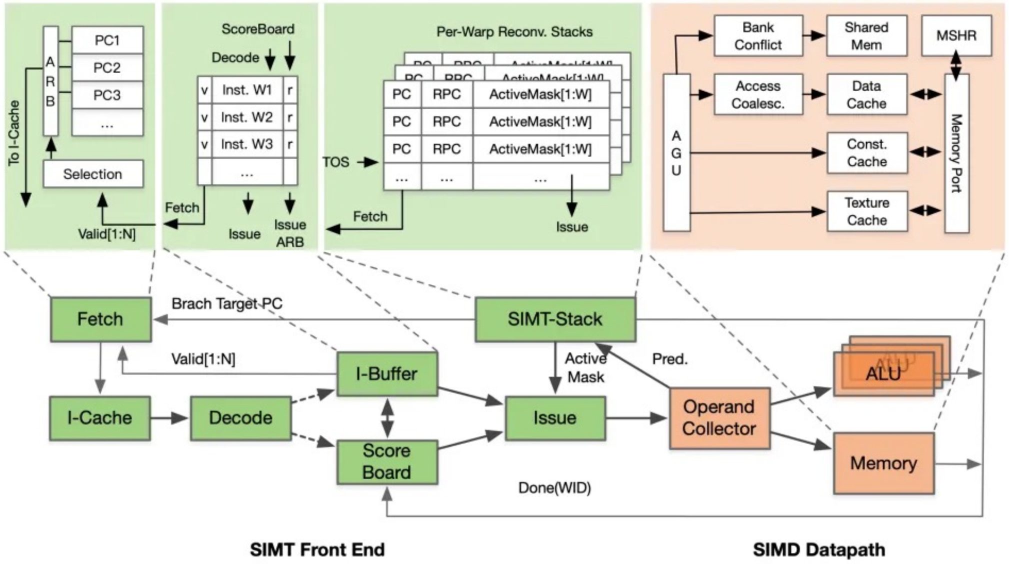
(SIMD)

- Programmer's view: *scalar multithreading (easy to write C/CUDA code)*
- Hardware's view: *SIMD efficiency (shared control, high ALU density)*



(SIMT)

SIMT



- Programmer's view: *scalar multithreading (easy to write C/CUDA code)*
- Hardware's view: *SIMD efficiency (shared control, high ALU density)*

SIMT = Front-End Control (Warp/Thread Scheduler) + Back-End Execution (SIMD)

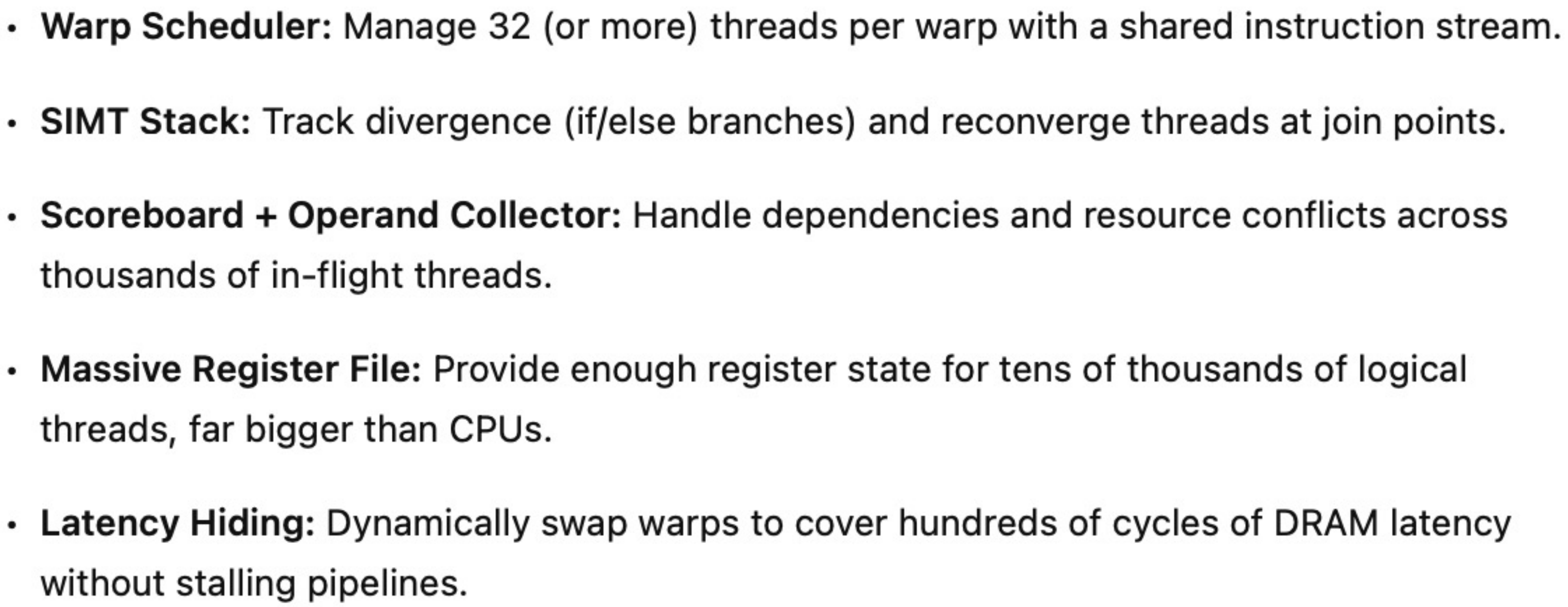
Front-End Control (Warp Scheduler):

- **Fetch stage:** Fetch → I-Cache → Decode → I-Buffer. The Warp Scheduler uses the PC (program counter) to obtain the address of the next instruction from main memory.
- **Instruction issue stage:**
  - **I-Buffer**
  - **Scoreboard:** Tracks resource conflicts and data dependencies.
  - **Issue / MT Issue (Multi-thread issue):** Distributes Warps/Threads to CUDA cores.
  - **SIMT-Stack:** Manages branch divergence caused by if-else statements.

Back-End Execution (SIMD):

- **Execution stage:**
  - **Operand Collector:** Fetches the required operands from the Register Files.
  - **ALU:** Executes arithmetic instructions (e.g., FP32/INT32 operations).
  - **Memory:** Handles LD/ST (load/store) requests.

SIMT



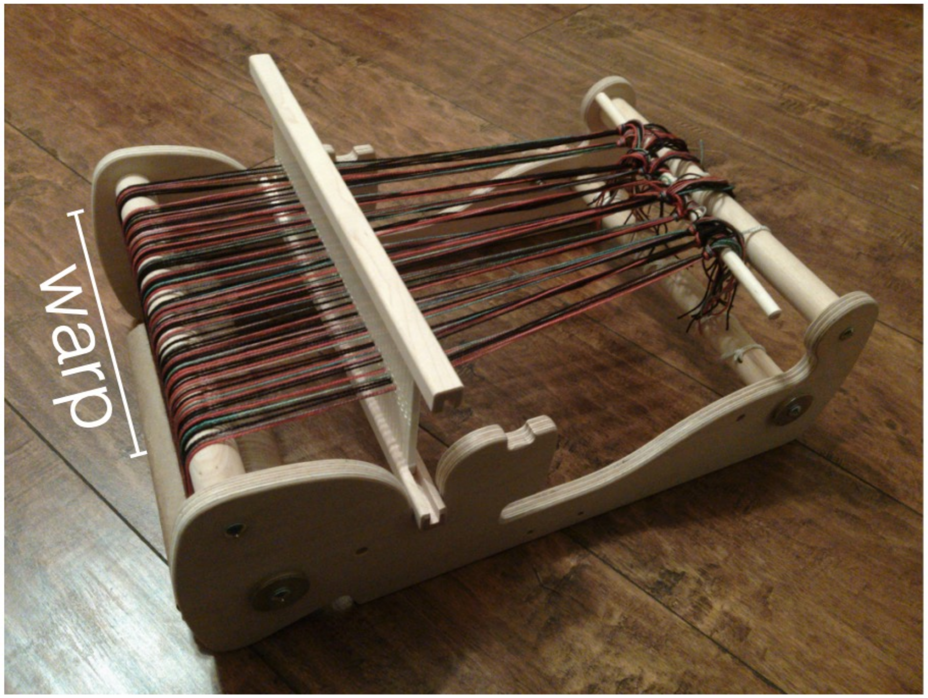
- **Warp Scheduler:** Manage 32 (or more) threads per warp with a shared instruction stream.
- **SIMT Stack:** Track divergence (if/else branches) and reconverge threads at join points.
- **Scoreboard + Operand Collector:** Handle dependencies and resource conflicts across thousands of in-flight threads.
- **Massive Register File:** Provide enough register state for tens of thousands of logical threads, far bigger than CPUs.
- **Latency Hiding:** Dynamically swap warps to cover hundreds of cycles of DRAM latency without stalling pipelines.

Warp

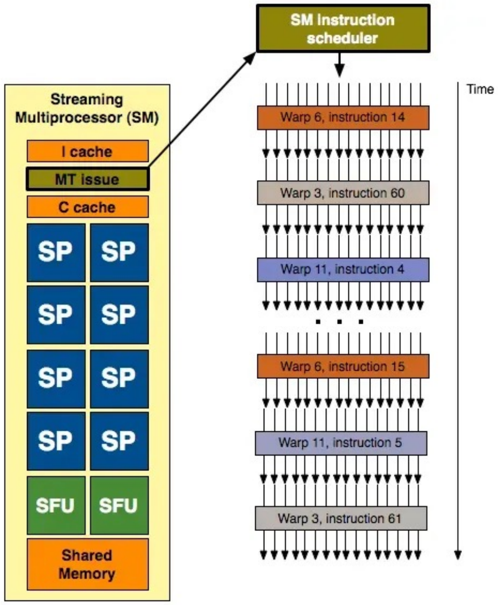
In CUDA programming, a warp is the implementation of the Single Instruction, Multiple Threads (SIMT) execution model. A warp consists of 32 threads that execute the same instruction simultaneously, allowing for efficient parallel processing on NVIDIA GPUs.

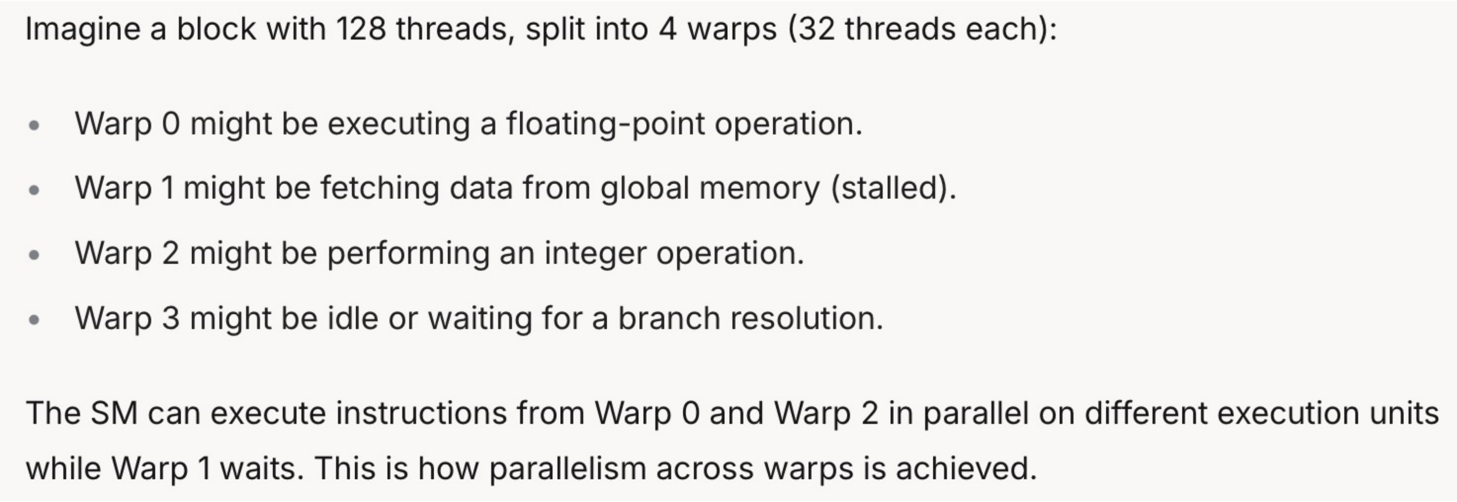
- **Memory Sharing:** Threads within a warp can efficiently share data using shared memory, which is a fast, on-chip memory accessible to all threads in the same block. Efficient use of shared memory within a warp can lead to significant performance improvements.
- **Warp Divergence:** When threads within the same warp follow different execution paths due to conditional statements (e.g., if-else branches). In such cases, the warp must serialize the execution of each divergent path, leading to reduced parallel efficiency and potential performance degradation.
- **Performance Considerations:** Optimal performance is achieved when all threads in a warp follow the same execution path. Divergence, where threads take different paths, can lead to performance degradation.

Warp



Warp



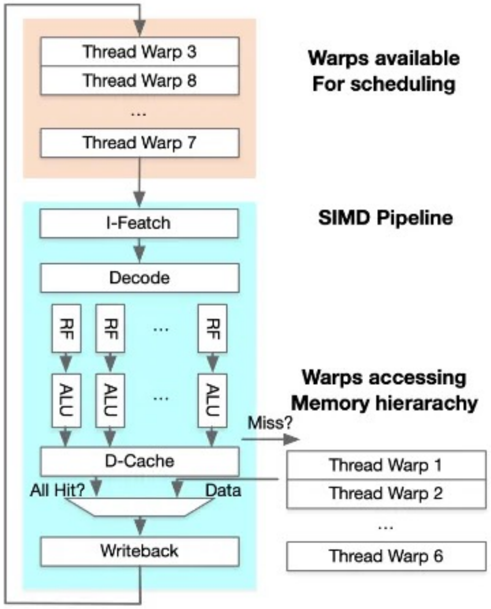


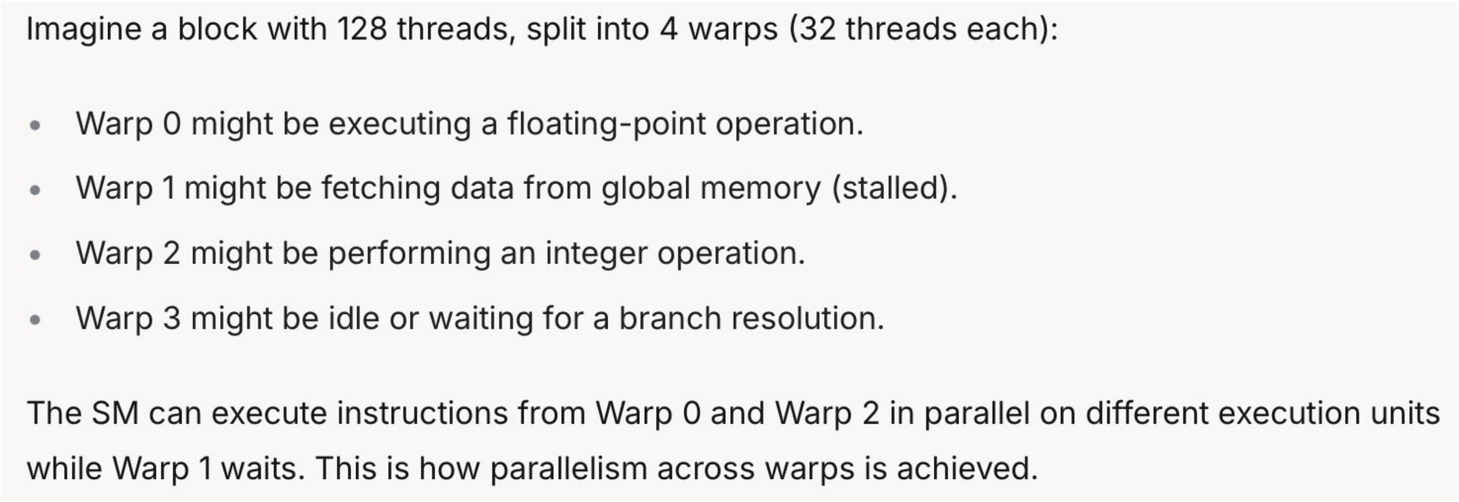
Imagine a block with 128 threads, split into 4 warps (32 threads each):

- Warp 0 might be executing a floating-point operation.
- Warp 1 might be fetching data from global memory (stalled).
- Warp 2 might be performing an integer operation.
- Warp 3 might be idle or waiting for a branch resolution.

The SM can execute instructions from Warp 0 and Warp 2 in parallel on different execution units while Warp 1 waits. This is how parallelism across warps is achieved.

Warp





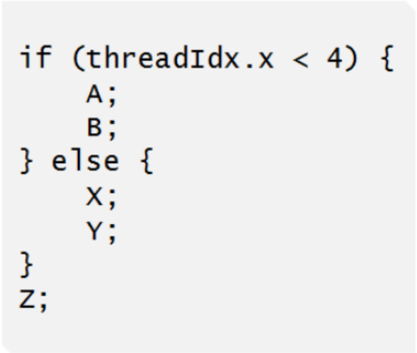
Imagine a block with 128 threads, split into 4 warps (32 threads each):

- Warp 0 might be executing a floating-point operation.
- Warp 1 might be fetching data from global memory (stalled).
- Warp 2 might be performing an integer operation.
- Warp 3 might be idle or waiting for a branch resolution.

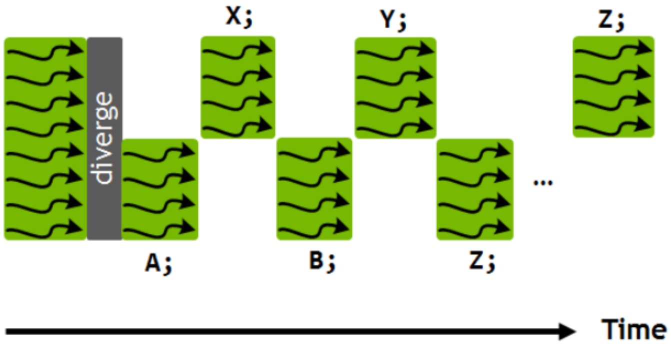
The SM can execute instructions from Warp 0 and Warp 2 in parallel on different execution units while Warp 1 waits. This is how parallelism across warps is achieved.

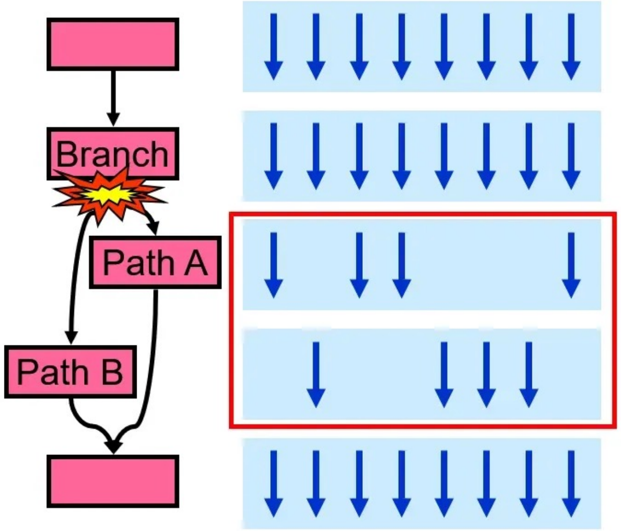
Divergence

Warp divergence happens when threads in the same warp take different control flow paths (e.g., inside an if/else or loop).



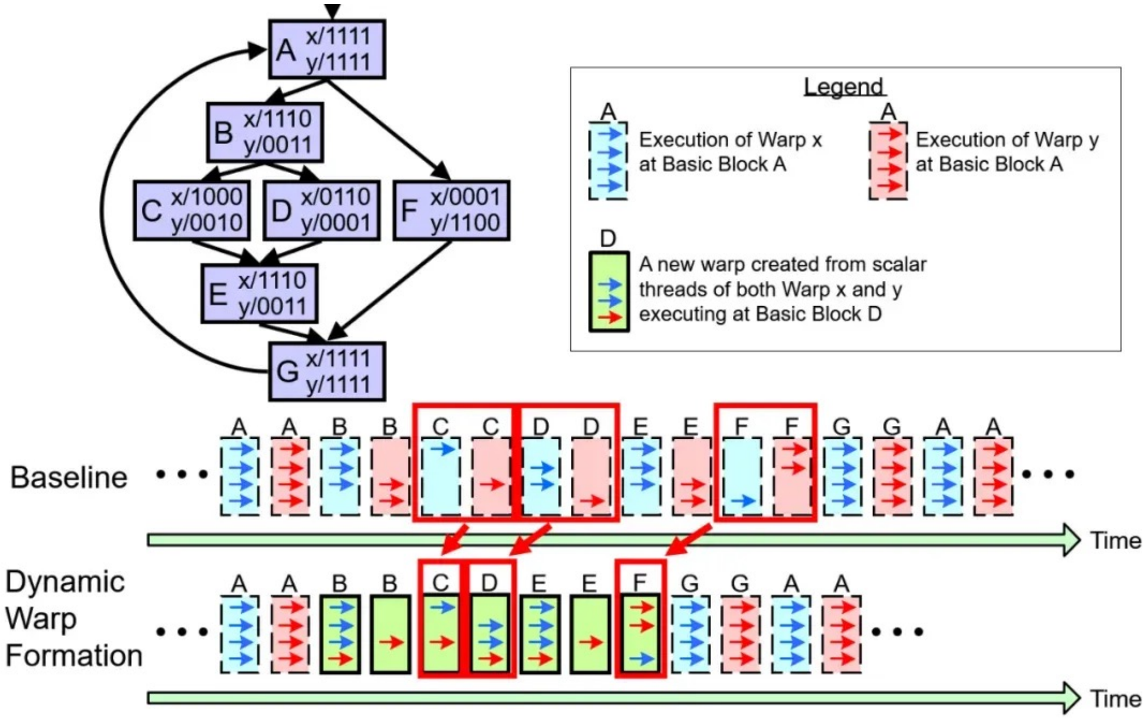
```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```





Dynamic Warp Formation

Dynamic Warp Formation (DWF) is a hardware technique to reduce SIMT branch divergence overhead. DWF solves wrap divergence by dynamically regrouping threads that share the same program counter into new warps at runtime, allowing them to execute in parallel again. This improves ALU utilization and overall throughput while preserving the scalar-thread programming model seen by developers.


A x/1111 y/1111
B x/1110 y/0011
C x/1000 y/0010
D x/0110 y/0001
F x/0001 y/1100
E x/1110 y/0011
G x/1111 y/1111
Legend
A
Execution of Warp x at Basic Block A
A
Execution of Warp y at Basic Block A
D
A new warp created from scalar threads of both Warp x and y executing at Basic Block D
Baseline
A A B B C C D D E E F F G G A A
Time
Dynamic Warp Formation
A A B B C D E E F G G A A
Time

Choosing Grid and Block Sizes in CUDA

- Use multiples of 32 threads per block (match the warp size).
- Typical block size: 128–256 threads for good SM utilization.
- Launch many more blocks than SMs to ensure load balancing.
- Match block dimensions to data layout (e.g., 16×16 for matrices).
- Be mindful of shared memory usage, which limits blocks per SM.
- Watch register usage, since high register pressure reduces occupancy.
- Ensure the grid covers the entire dataset (gridDim ≈ ceil(N / blockDim)).
- Expose enough parallelism so the GPU always has work to schedule.