计算机体系结构

GPGPU

Review

- **单发射Tumasulo + 推断执行，superscalar, VLIW, Fined-grained MT (FGMT), coarse-grained MT, CMP, SMT, vector**
  - 哪个的概念和实现还不清楚、需要重新过一遍？

Review

- **向量机的存储器访问**
  - 存储器组织：独立存储体、多体交叉方式
  - Stride : 固定步长（1 or 常数）, 非固定步长（index）
- **基于向量机模型的优化**
  - 链接技术
  - 有条件执行
  - 稀疏矩阵的操作

- **多媒体扩展指令**
  - 扩展的指令类型较少
  - 向量寄存器长度较短
- **GPU**

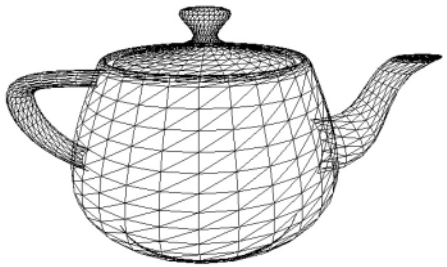
GPU

- GPU的起源
- GPU基本的硬件结构
- GPU编程模型
- GPU的存储层次
- GPU分支处理（发散与汇聚）

Graphics Processing Units (GPUs)

- **早期的GPU是指带有高性能浮点运算部件、可高效生成3D图形的具有固定功能的专用设备 (mid-late 1990s)**
  - 让PC机具有类似工作站的图形功能
  - **用户可以配置图形处理流水线，但不是真正的对其编程**
- **2001-2005，GPU加入了越来越多的可编程性**
  - 例如新的语言 Cg可用来编写一些小的程序处理图形的顶点或像素，是Windows DirectX的变体
  - 大规模并行（针对每帧上百万顶点或像素）但非常受限于编程模型
- **有些用户注意到通过将输入和输出数据映射为图像，并对顶点或像素渲染计算 可进行通用计算**
  - 因为不得不使用图形流水线模型，这对完成通用计算来说是个非常难用的编程模型

What GPUs were originally designed to do: 3D rendering



Input: description of a scene:
3D surface geometry (e.g., triangle mesh)
surface materials, lights, camera, etc.

Image credit: Henrik Wann Jensen

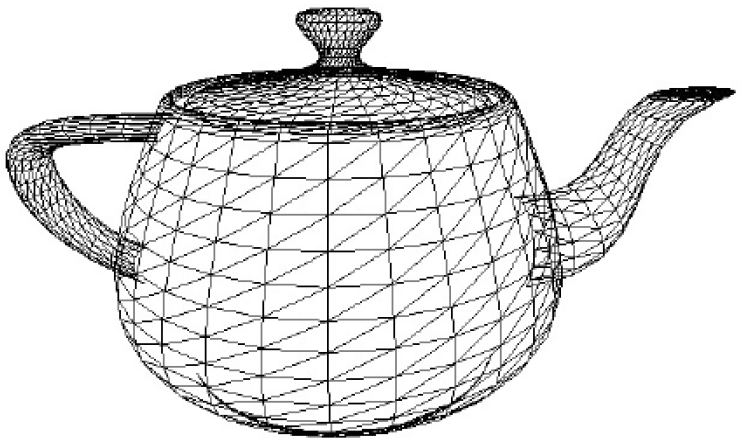
Output: image of the scene

Simple definition of rendering task: Generating a photorealistic image from 2D/3D models. Computing how each triangle in 3D mesh contributes to appearance of each pixel in the image?

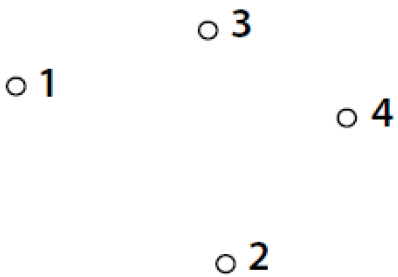
Tip: how to explain a system

- Step 1: describe the things (key entities) that are manipulated
  - The nouns

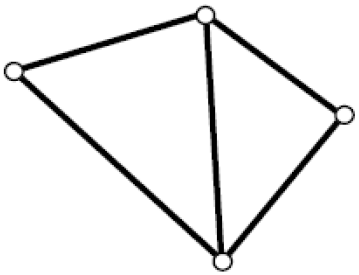
Real-time graphics primitives (entities)



Represent surface as a 3D triangle mesh

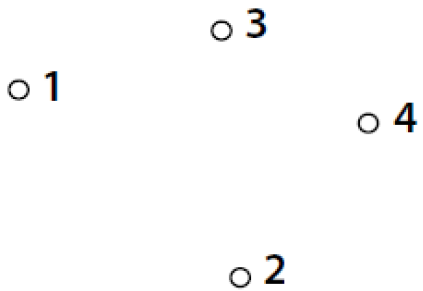


Vertices

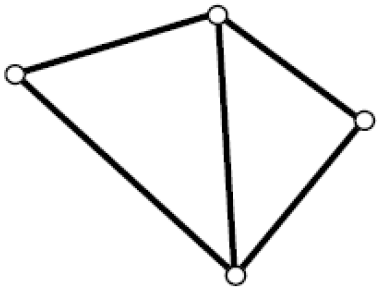


Primitives
(e.g., triangles, points, lines)

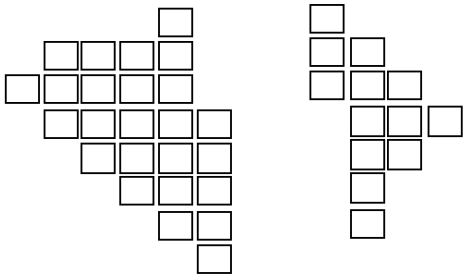
Real-time graphics primitives (entities)



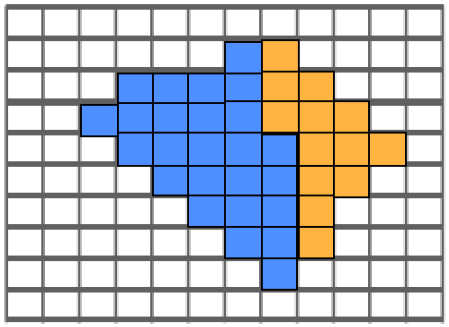
Vertices



Primitives
(e.g., triangles, points, lines)



Fragments



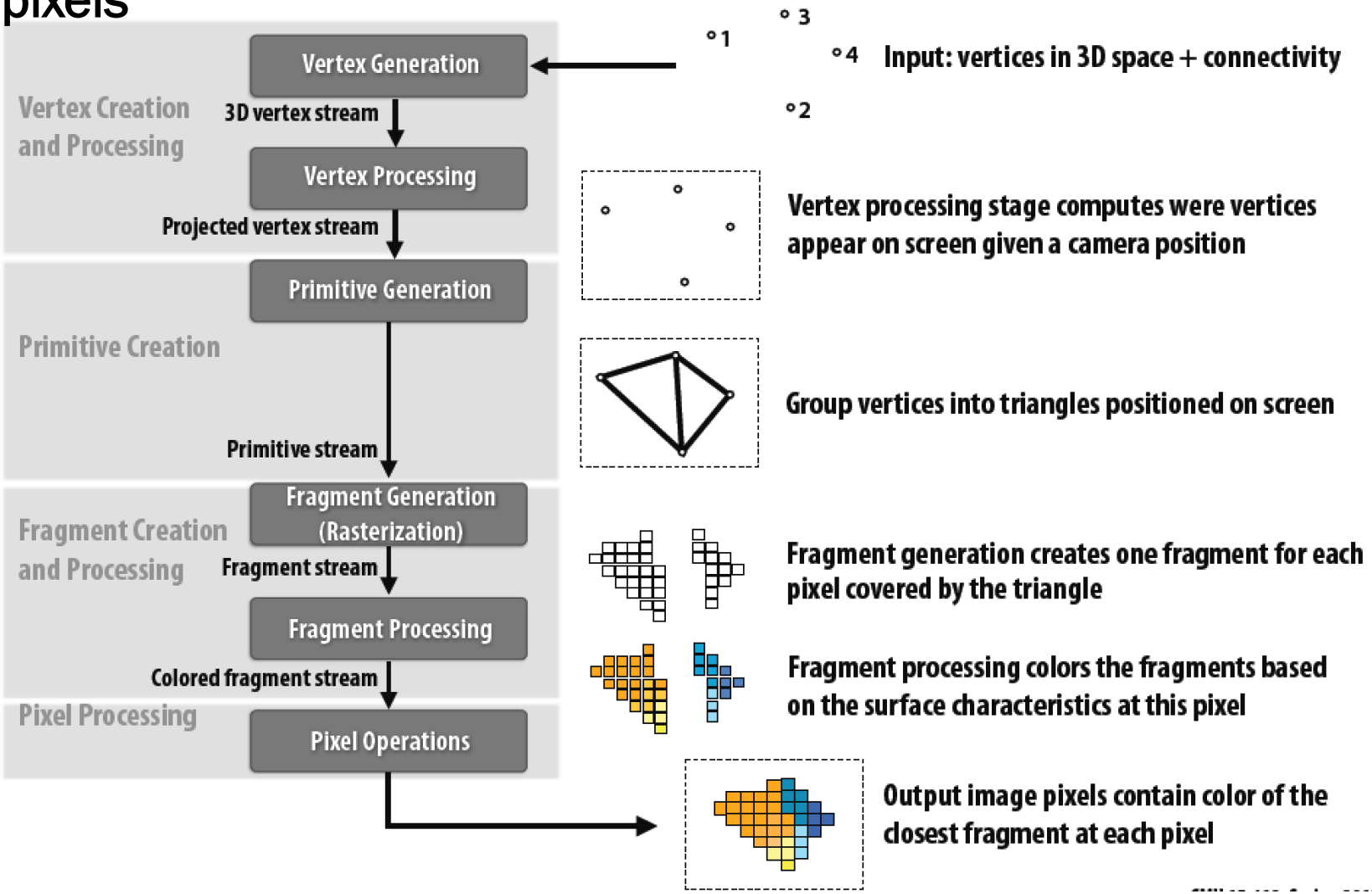
Pixels (in an image)

Tip: how to explain a system

- Step 1: describe the things (key entities) that are manipulated
  - The nouns
- Step 2: describe operations the system performs on the entities
  - The verbs

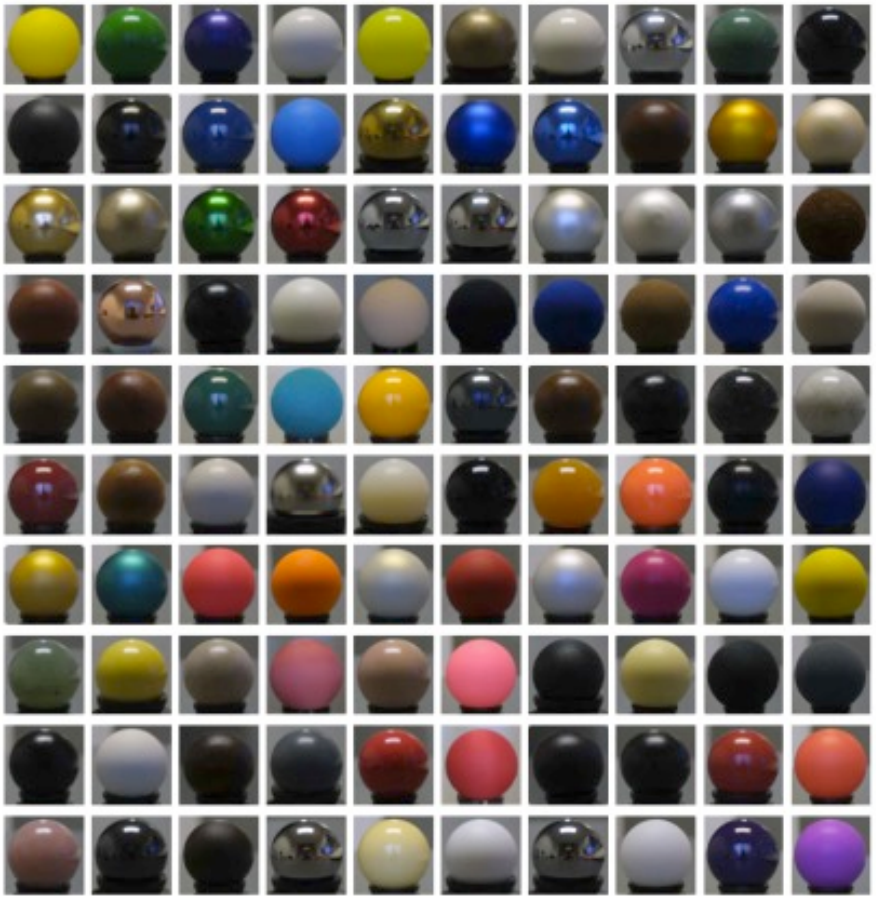
Graphics pipeline architecture

Performs operations on vertices, triangles, fragments, and pixels



Fragment processing simulates reflection of light off of real-world materials

Example materials:

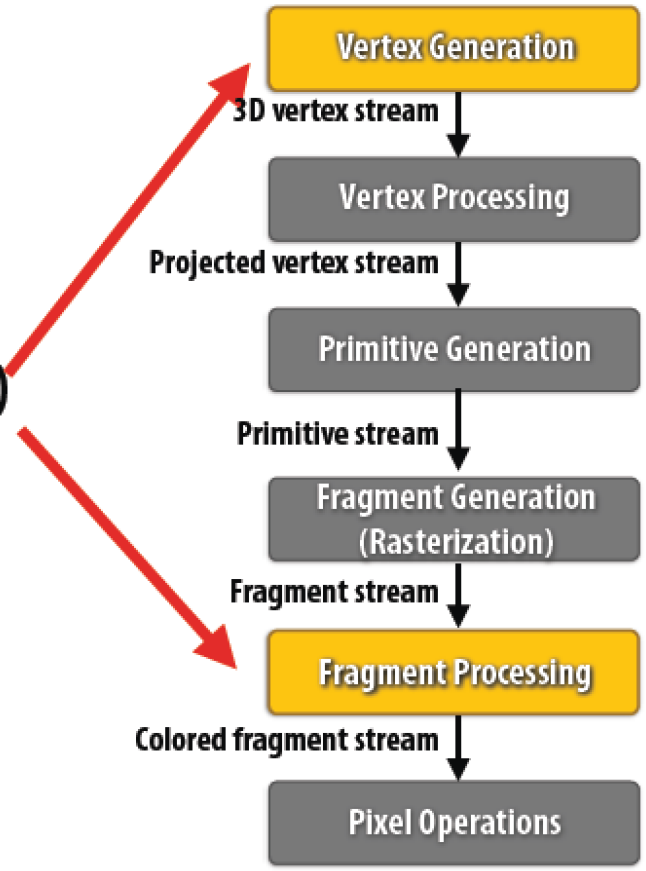


Early Graphics Programming

- **Graphics programming APIs provided application programmer mechanisms to set parameters of lights and materials**
  - `glLight(light_id, parameter_id, parameter_value)`
    - **Examples of light parameters: ambient/diffuse/specular color, position, direction**
  - `glMaterial(face, parameter_id, parameter_value)`
    - **Examples of material parameters: surface color, shininess**

Graphics Shading Language

- Allow application to specify materials and lights programmatically!
  - Support large diversity in materials
  - Support large diversity in lighting conditions
- Programmer provides mini-programs ("shaders") that defines pipeline logic for certain stages
  - Pipeline maps shader function onto all elements of input stream



Example Fragment Shader

Run once per fragment (per pixel covered by a triangle)

HLSL language shader program:
defines behavior of fragment processing stage

```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
  float3 kd;
  kd = myTex.Sample(mySamp, uv);
  kd *= clamp( dot(lightDir, norm), 0.0, 1.0);
  return float4(kd, 1.0);
}
```

read-only global variables

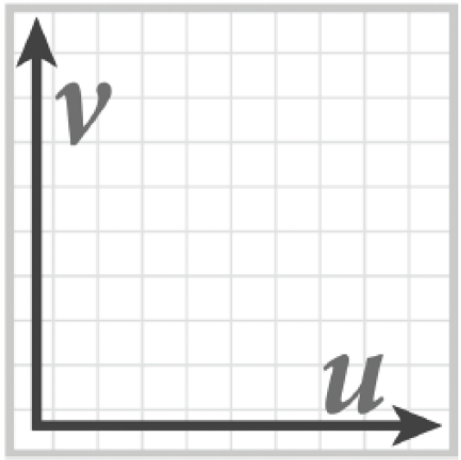
per-fragment inputs

"fragment shader"
(a.k.a kernel function mapped onto input fragment stream)

per-fragment output: surface color at pixel

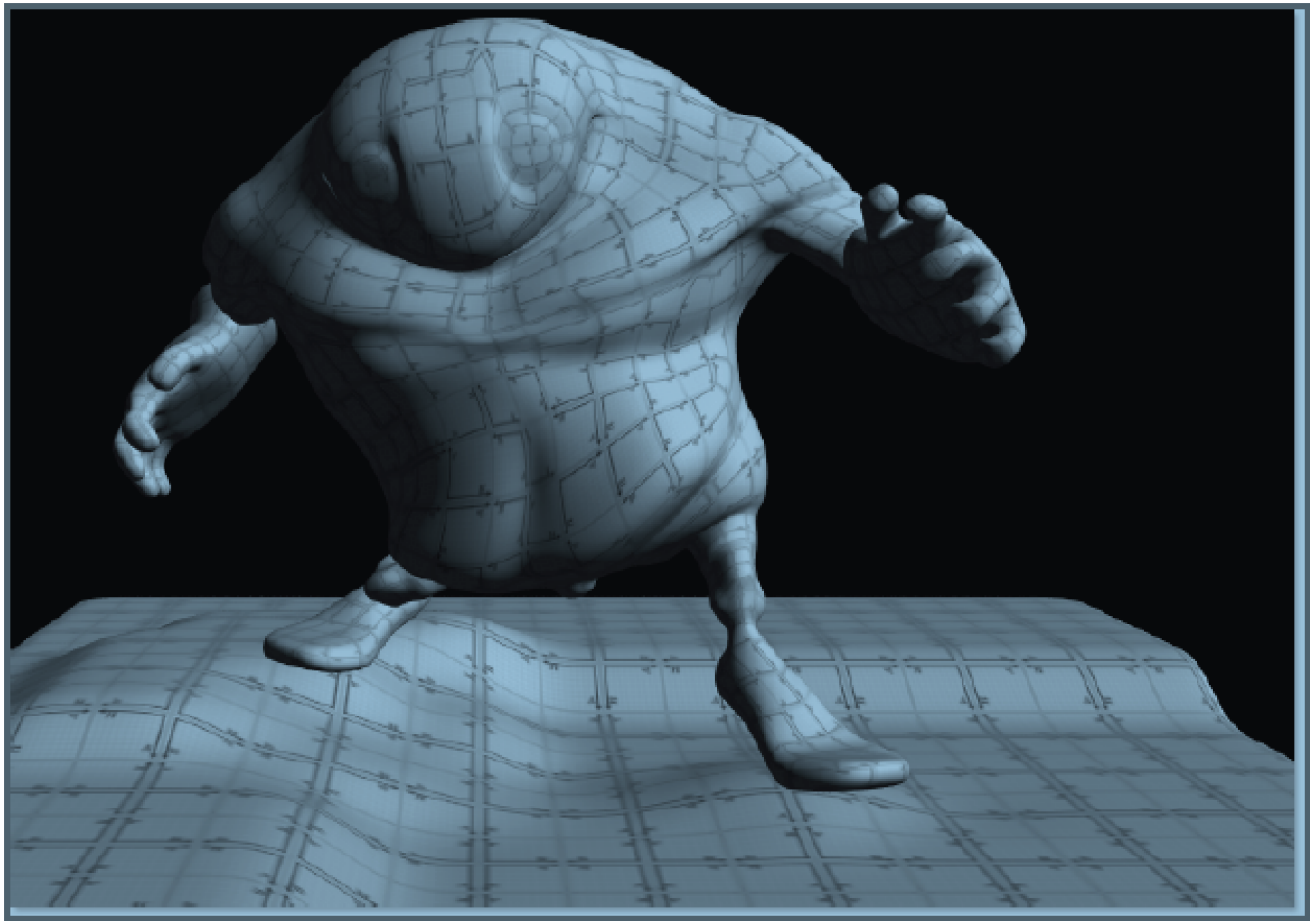
myTex is a texture map

myTex =



Shader Result

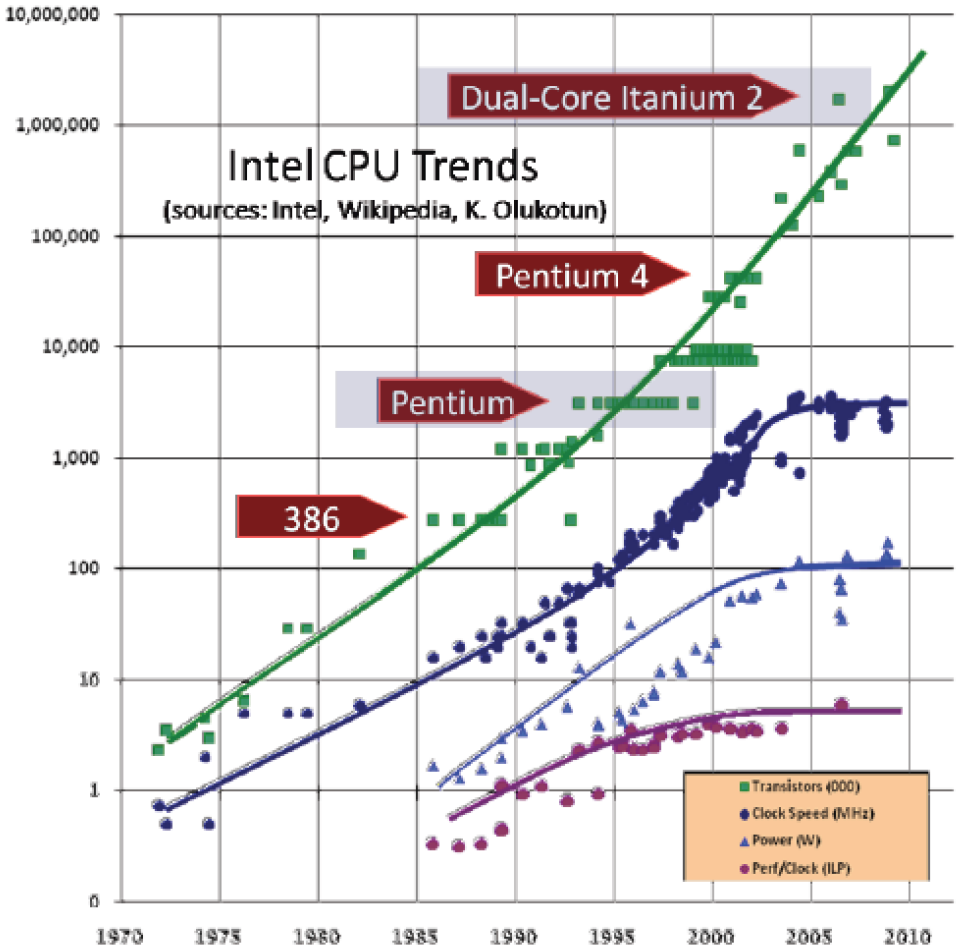
Image contains output of diffuseShader for each pixel covered by surface
(Pixels covered by multiple surfaces contain output from surface closest to camera)



Observation circa 2001-2003

These GPUs are very fast processors for performing the same computation (shaders) on collections of data (streams of vertices, fragments, pixels).

Wait a minute! That sounds a lot like data parallelism to me! I remember data parallelism from exotic supercomputers in the 90s.



Observation circa 2001-2003

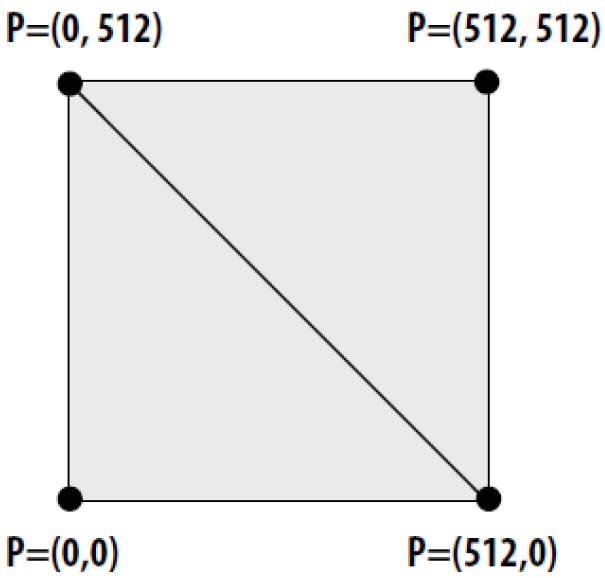
Set OpenGL output image size to be output array size (e.g., 512 x 512)

Render 2 triangles that exactly cover screen

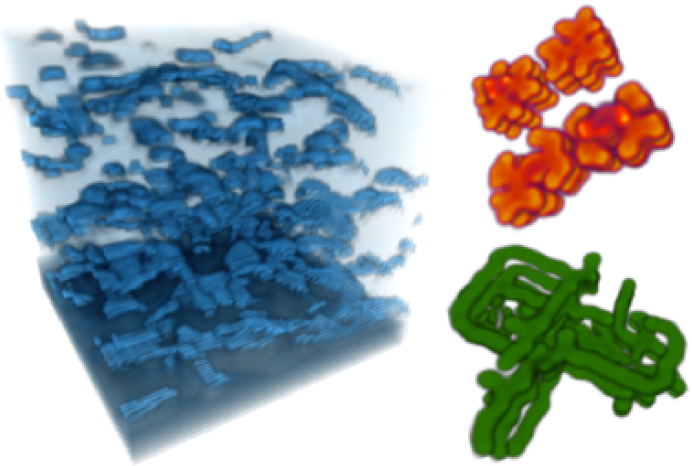
(one shader computation per pixel = one shader computation output image element)

We now can use the GPU like a data-parallel programming system.

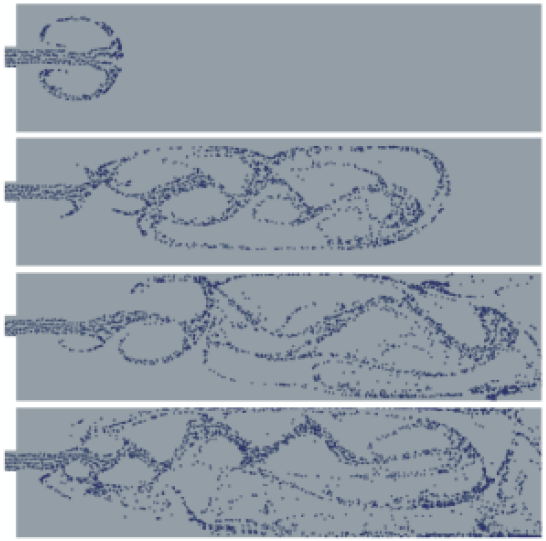
Fragment shader function is mapped over 512 x 512 element collection.



“GPGPU” 2002-2003



Coupled Map Lattice Simulation [Harris 02]



Sparse Matrix Solvers [Bolz 03]

Ray Tracing on Programmable Graphics Hardware [Purcell 02]

Brook stream programming language (2004)

- Stanford research project
- Abstract GPU as data-parallel processor

```
kernel void scale(float amount, float a<>, out float b<>)
{
   b = amount * a;
}

// note: omitting stream element initialization
float scale_amount;
float input_stream<1000>;
float output_stream<1000>;

// map kernel onto streams
scale(scale_amount, input_stream, output_stream);
```

- Brook compiler turned generic stream program into OpenGL commands such as drawTriangles()

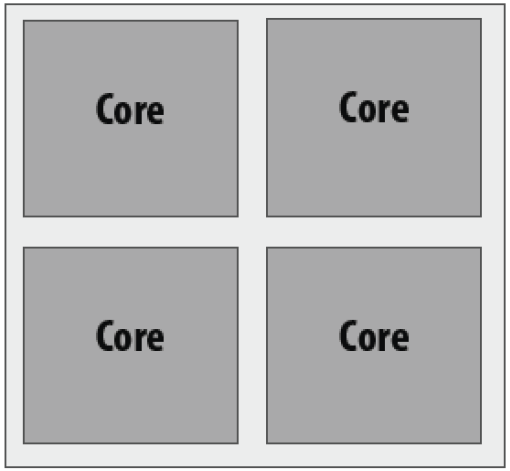
General-Purpose GPUs (GP-GPUs)

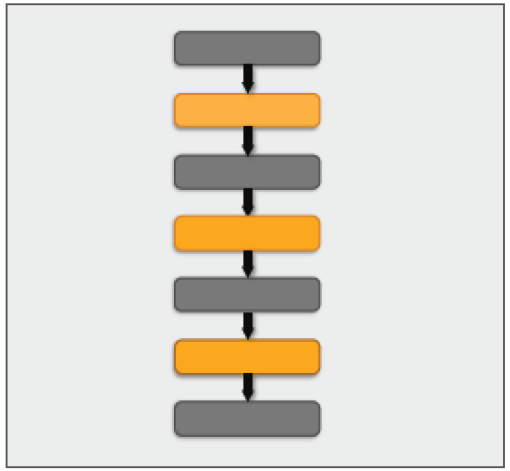
- **2006年, Nvidia 的 GeForce 8800 GPU 支持一种新的编程语言: CUDA**
  - “Compute Unified Device Architecture”
  - 随后工业界推出OpenCL，与CUDA具有相同的ideas, 但独立于供应商
- **Idea: 针对通用计算，发挥GPU计算的高性能和存储器的高带宽来加速一些通用计算中的核心（Kernels）**
- **一种协处理器模型（GPU作为附加设备）：Host CPU发射数据并行的kernels 到GP-GPU上运行**
- **我们仅讨论Nvidia CUDA样式的简化版本，仅考虑GPU的计算核部分，不涉及图形加速部分**

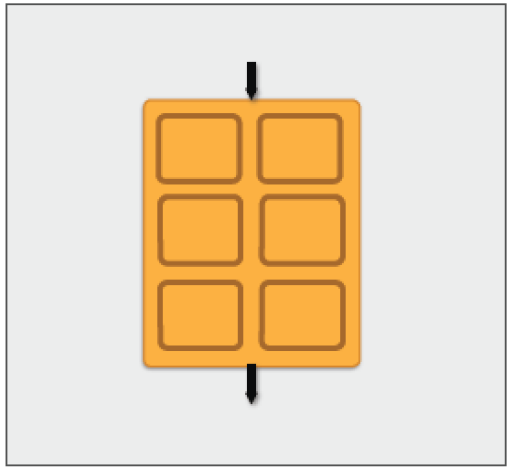
NVIDIA Tesla architecture (2007)

(GeForce 8xxx series)

First alternative, non-graphics-specific ("compute mode") software interface to GPU







Multi-core CPU architecture

CPU presents itself to OS as a multi-processor system

ISA provides instructions for managing execution context (program counter, VM mappings, etc.) on a per core basis

Pre-2007 GPU architecture

GPU presents following interface** to system software (driver):

Set screen size
Set shader program for pipeline
DrawTriangles

(** interface also included many other commands for configuring graphics pipeline)

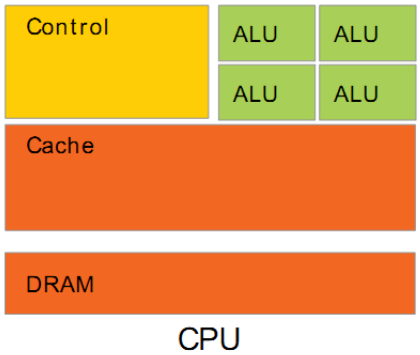
Post-2007 "compute mode" GPU architecture

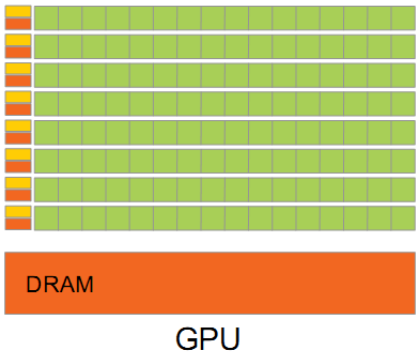
GPU presents a new data-parallel interface to system software (driver):

Set kernel program
Launch(kernel, N)

Using CPU+GPU Architecture

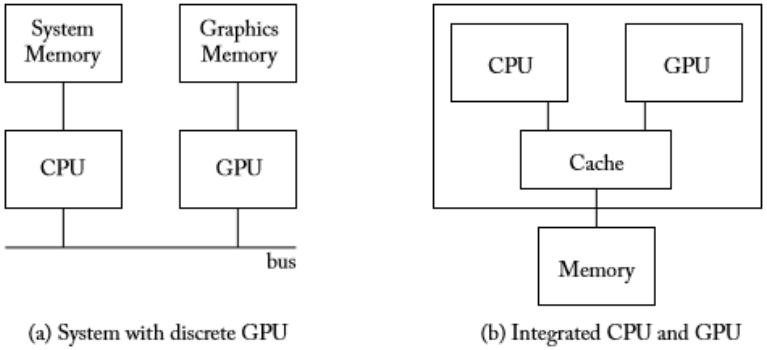
- CPU+GPU异构多核系统
  - 针对每个任务选择合适的处理器和存储器
- 通用CPU 适合执行一些串行的线程
  - 串行执行快
  - 带有cache，访问存储器延时低
- GPU 适合执行大量并行线程
  - 可扩放的并行执行
  - 高带宽的并行存取





强控制、弱计算

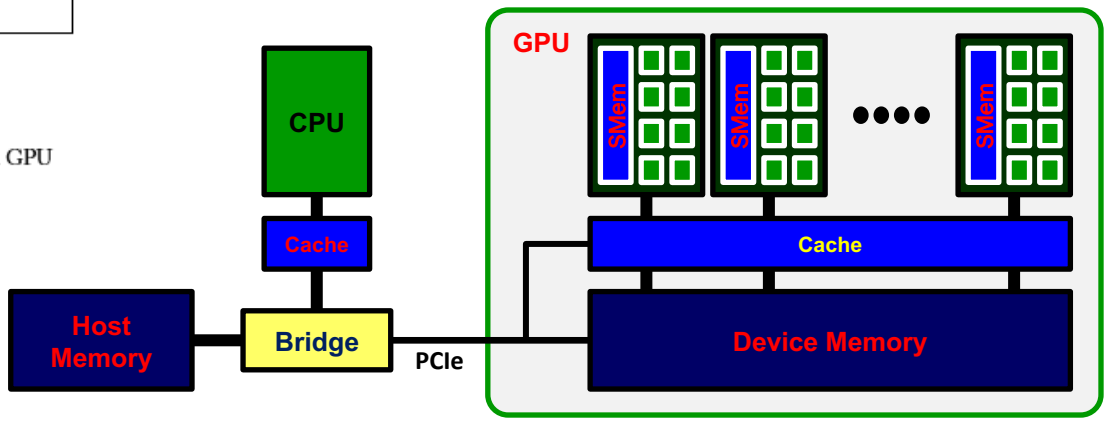
弱控制、强计算



(a) System with discrete GPU

(b) Integrated CPU and GPU

Figure 1.1: GPU computing systems include CPUs.



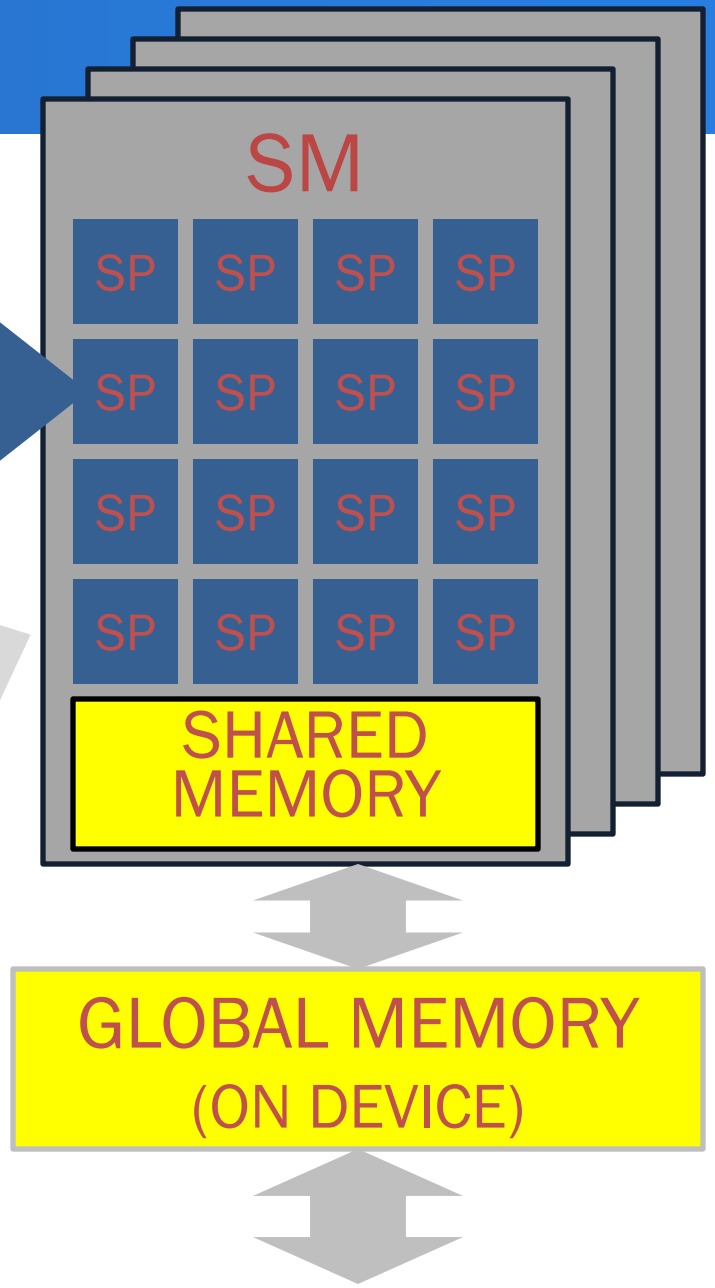
GPU: a multithreaded coprocessor

SP: scalar processor
'CUDA core'

Executes one thread

SM
streaming multiprocessor
32xSP (or 16, 48 or more)

<u>Fast</u> local '**shared memory**'
(shared between SPs)
16 KiB (or 64 KiB)



GPUs are SIMD Engines Underneath

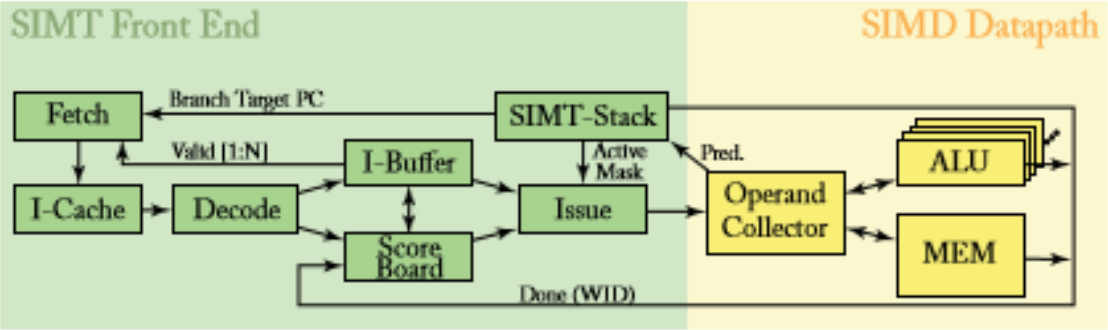


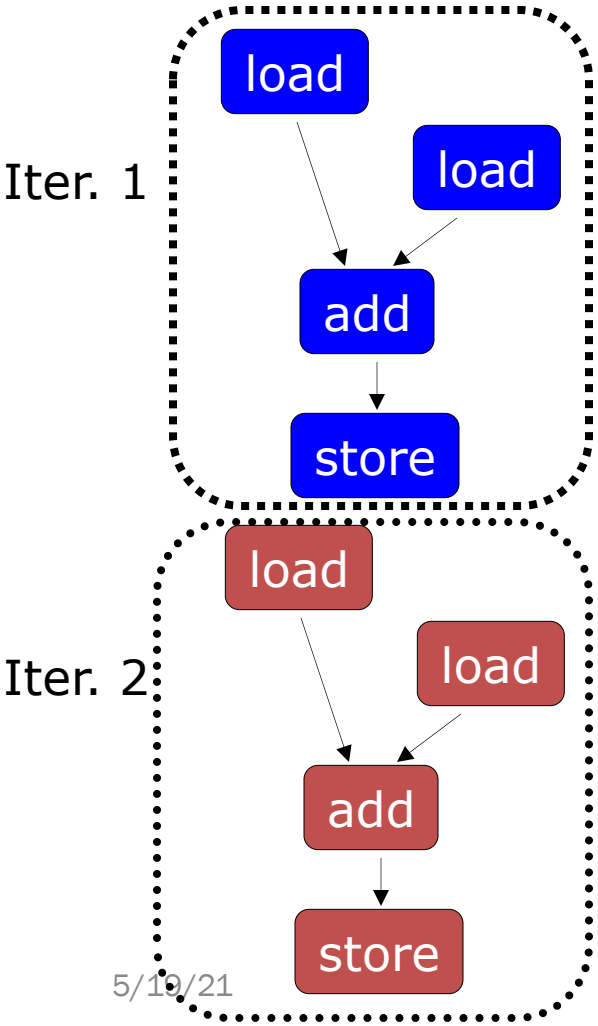
Figure 3.1: Microarchitecture of a generic GPGPU core.

- **指令流水线类似于SIMD的流水线。**
  - 不是用SIMD指令编程
  - 基于一般的指令，应用由一组线程构成。
- **两个概念**
  - Programming Model (Software) vs Execution Model (Hardware)
- **编程模型指程序员如何描述应用（从程序员角度看到的机器模型）**
  - 例如, 顺序模型 (von Neumann), 数据并行(SIMD), 数据流模型、多线程模型 (MIMD, SPMD), …
- **执行模型指硬件底层如何执行代码**
  - 例如, 乱序执行、向量机、数据流处理机、多处理机、多线程处理机等
- **执行模型与编程模型可以差别很大**
  - 例如., 顺序模型可以在乱序执行的处理器上执行。
  - SPMD 模型可以用SIMD处理器实现 (a GPU)

How Can You Exploit Parallelism Here?

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

Scalar Sequential Code

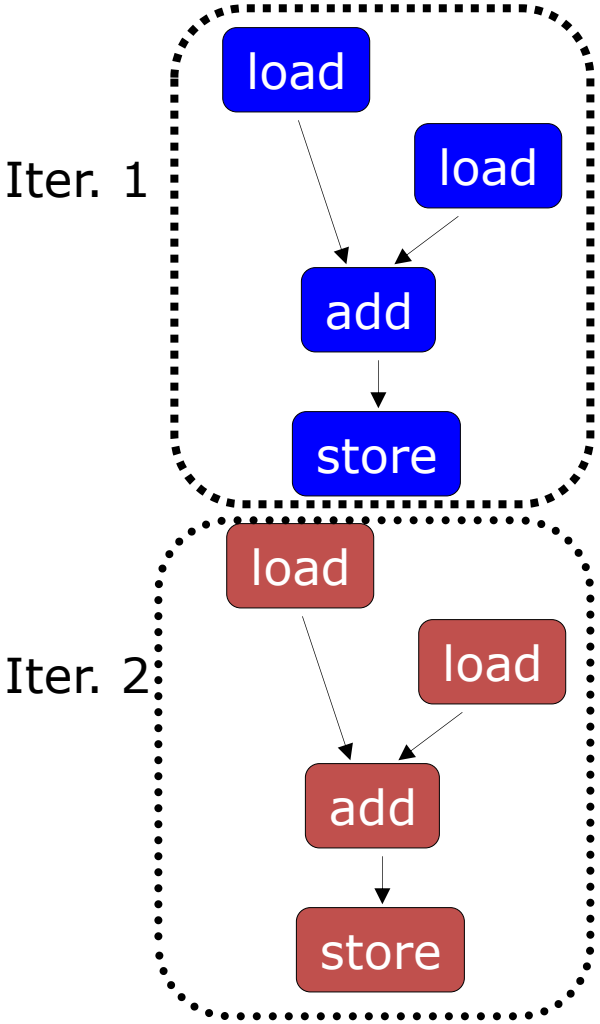


Let's examine three programming options to exploit instruction-level parallelism present in this sequential code:

1. Sequential (SISD)
2. Data-Parallel (SIMD)
3. Multithreaded (MIMD/SPMD)

Prog. Model 1: Sequential (SISD)

Scalar Sequential Code



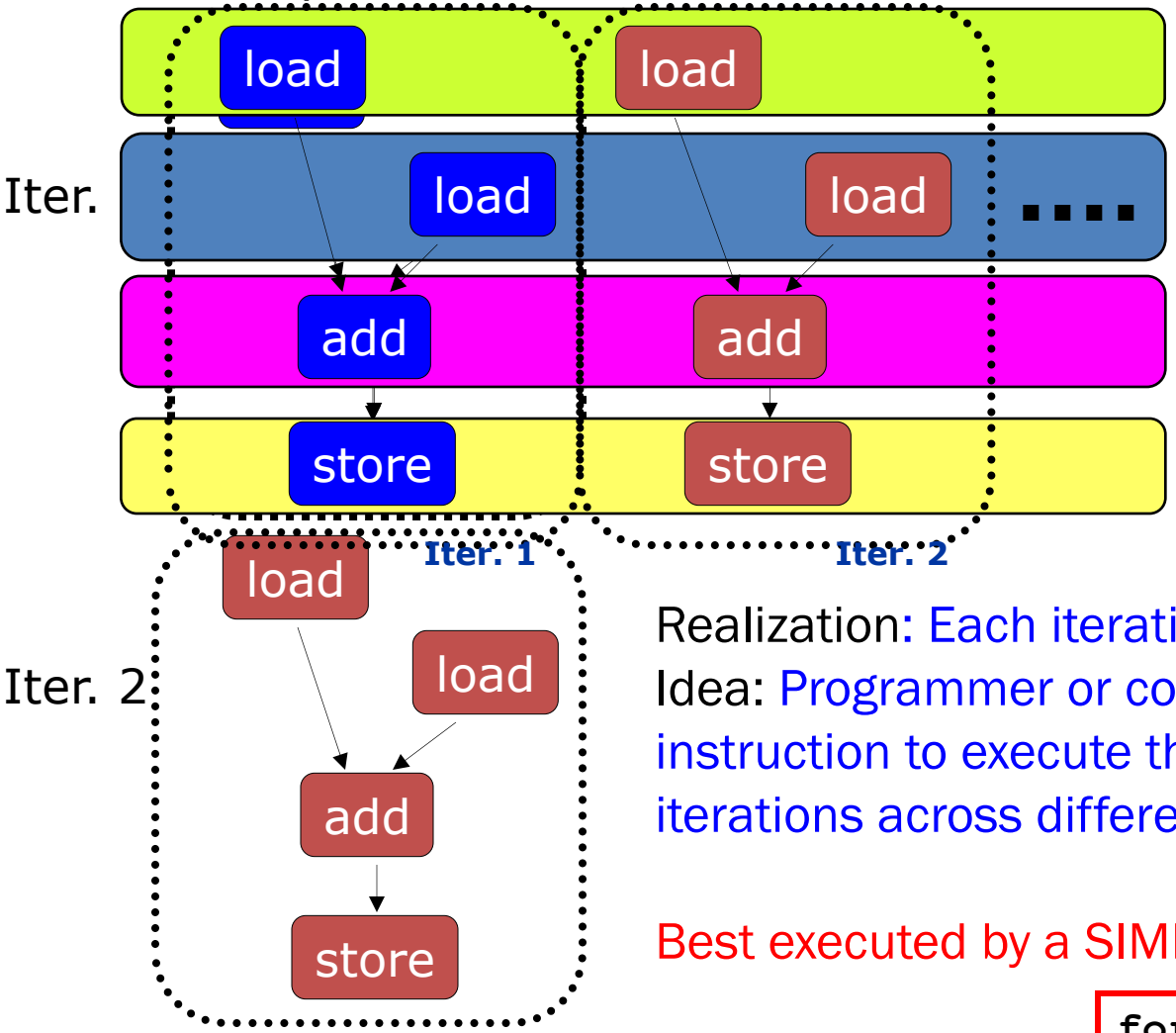
- **Can be executed on a Pipelined processor**
- **Out-of-order execution processor**
  - Independent instructions executed when ready
  - Different iterations are present in the instruction window and can execute in parallel in multiple functional units
  - In other words, the loop is dynamically unrolled by the hardware
- **Superscalar or VLIW processor**
  - Can fetch and execute multiple instructions per cycle

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

Prog. Model 2: Data Parallel (SIMD)

Scalar Sequential Code

Vector Instruction



Vectorized Code

VLD A → V1

VLD B → V2

VADD V1 + V2 → V3

VST V3 → C

Realization: Each iteration is independent

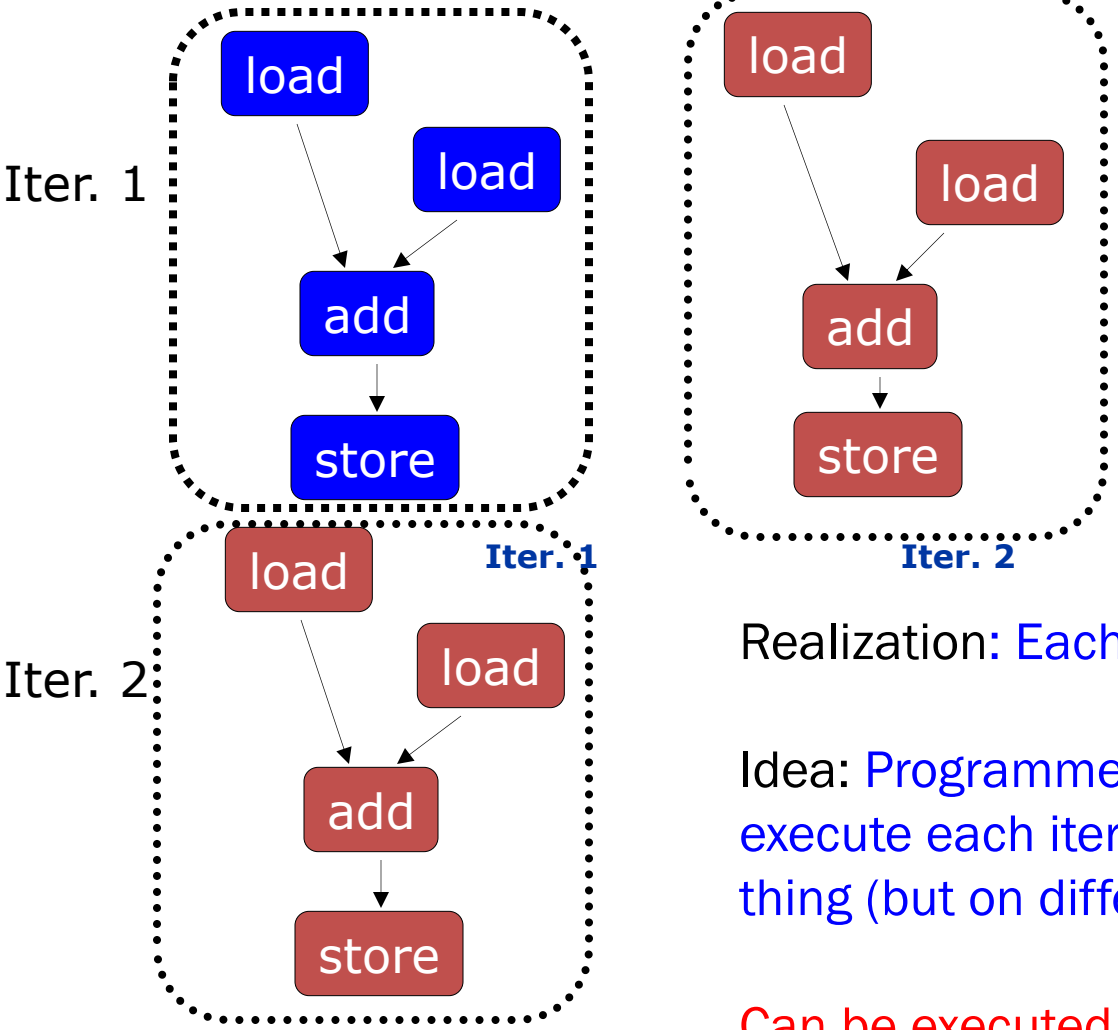
Idea: Programmer or compiler generates a SIMD instruction to execute the same instruction from all iterations across different data

Best executed by a SIMD processor (vector, array)

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

Prog. Model 3: Multithreaded

Scalar Sequential Code




```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

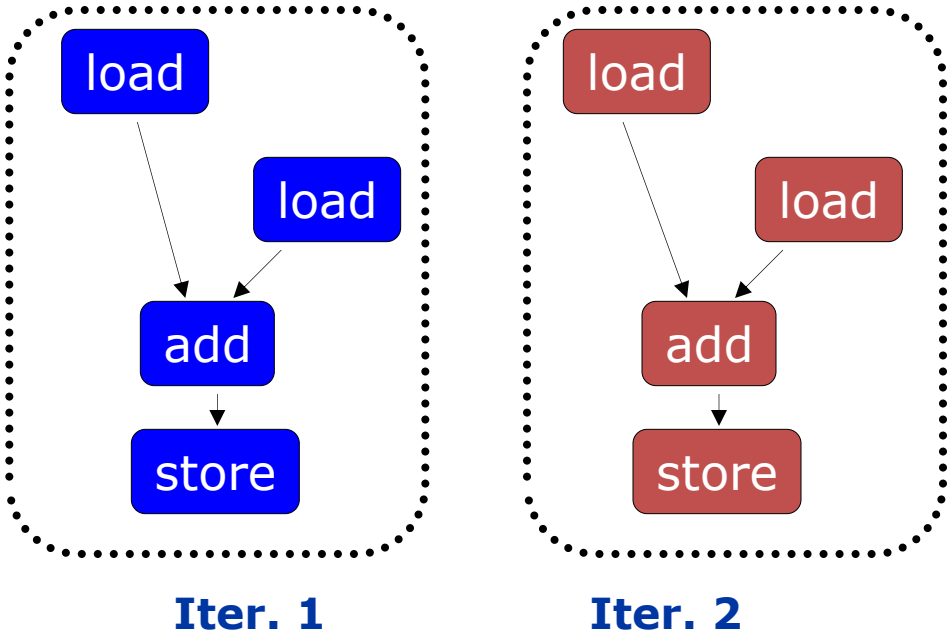
....

Realization: Each iteration is independent

Idea: Programmer or compiler generates a thread to execute each iteration. Each thread does the same thing (but on different data)

Can be executed on a MIMD machine

Prog. Model 3: Multithreaded




```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

....

This particular model is also called:

SPMD: Single Program Multiple Data

Can be executed on a SIMT machine
Single Instruction Multiple Thread

Can be executed on a SIMD machine

SPMD

- **Single procedure/program, multiple data**
  - 它是一种编程模型而不是计算机组织
- **每个处理单元执行同样的过程，处理不同的数据**
  - 这些过程可以在程序中的某个点上同步，例如 barriers

- **多条指令流执行相同的程序**
  - 每个程序/过程
    - 操作不同的数据
    - 运行时可以执行不同的控制流路径
  - 许多科学计算应用以这种方式编程，运行在MIMD硬件结构上 (multiprocessors)
  - 现代 GPUs 以这种类似的方式编程，运行在SIMD硬件上

A GPU is a SIMD (SIMT) Machine

- **GPU不是用SIMD指令编程**
- **使用线程 (SPMD 编程模型)**
  - 每个线程执行同样的代码，但操作不同的数据元素
  - 每个线程有自己的上下文(即可以独立地启动/执行等）
- **一组执行相同指令的线程由硬件动态组织成warp**
  - 一个warp是由硬件形成的SIMD操作

Threads and Blocks

- 一个线程对应一个数据元素
- 大量的线程组织成很多线程块
- 许多线程块组成一个网格

- GPU 由硬件对线程进行管理
  - Thread Block Scheduler
  - SIMD Thread Scheduler

Nvidia CUDA	OpenCL	Hennessy & Patterson[7]
Thread	Work-item	Sequence of SIMD Lane operations
Warp	Wavefront	Thread of SIMD Instructions
Block	Workgroup	Body of vectorized loop
Grid	NDRange	Vectorized loop

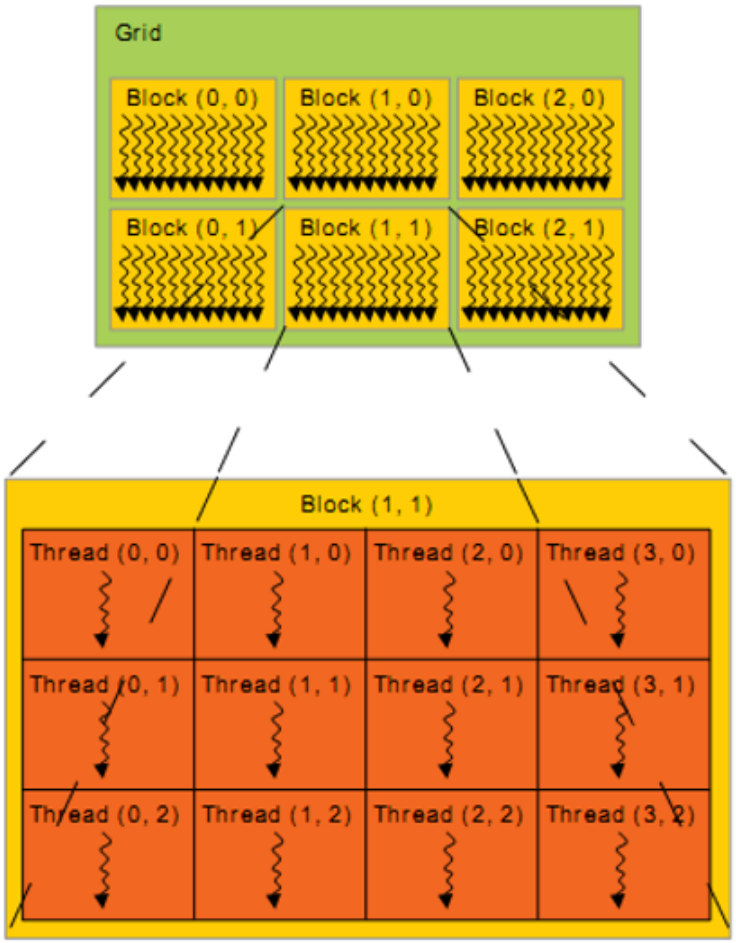
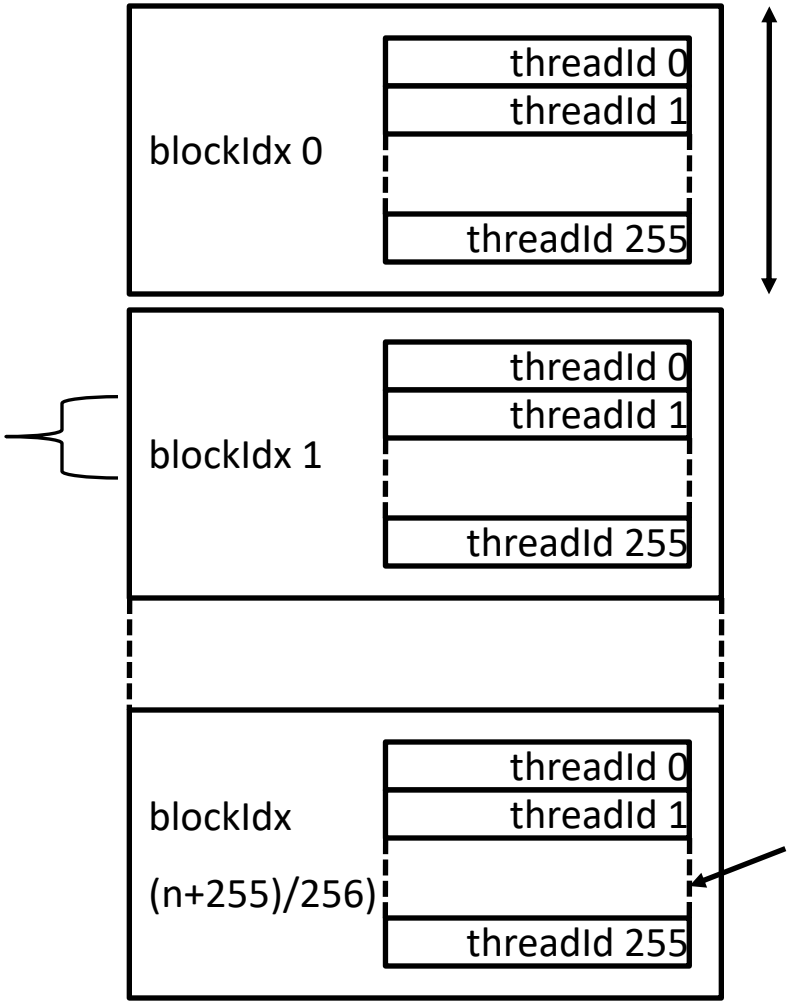


Figure 6 Grid of Thread Blocks

Programmer’s View of Execution

创建足够的线程块以适应输入向量(Nvidia 中将由多个线程块构成的、在GPU上运行的代码称为*Grid*，*Grid可以是2维的*)



blockDim = 256 (programmer can choose)

Conditional **(i<n)** turns off unused threads in last block

处理向量长度为8192的程序组织：

Grid 包含16个线程块（512个元素/Block）

Block包含16个SIMD 线程

每个SIMD线城处理32个向量元素

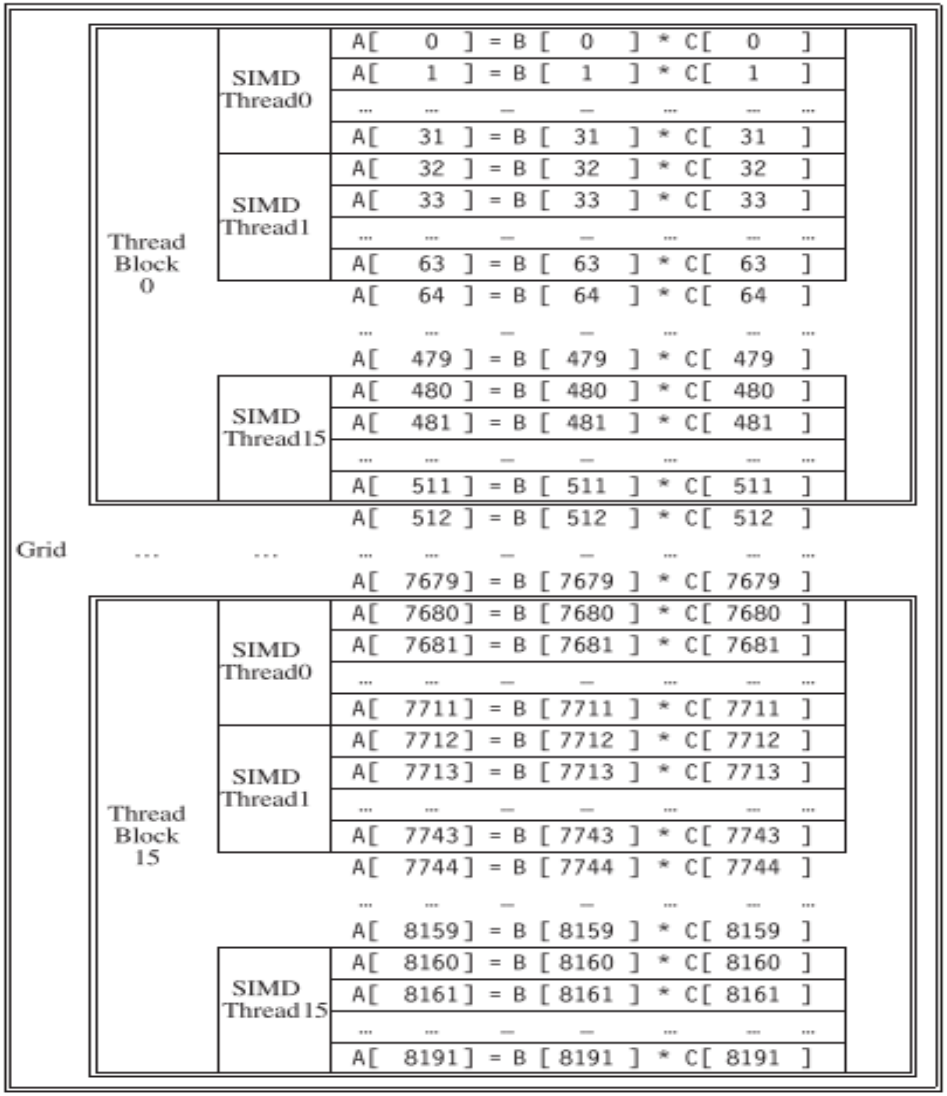
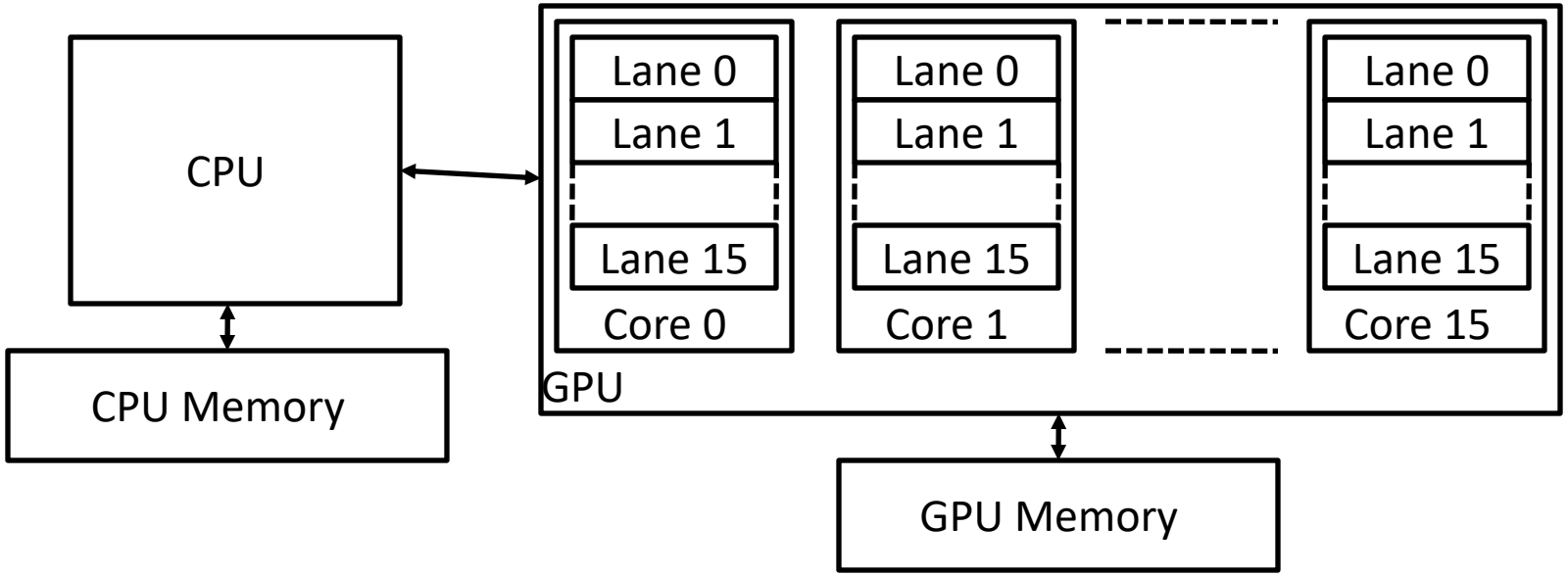


Figure 4.13 The mapping of a Grid (vectorizable loop), Thread Blocks (SIMD basic blocks), and threads of SIMD instructions to a vector-vector multiply, with each vector being 8192 elements long. Each thread of SIMD instructions calculates 32 elements per instruction, and in this example, each Thread Block contains 16 threads of SIMD instructions and the Grid contains 16 Thread Blocks. The hardware Thread Block Scheduler assigns Thread Blocks to multithreaded SIMD Processors, and the hardware Thread Scheduler picks which thread of SIMD instructions to run each clock cycle within a SIMD Processor. Only SIMD Threads in the same Thread Block can communicate via local memory. (The maximum number of SIMD Threads that can execute simultaneously per Thread Block is 32 for Pascal GPUs.)

Hardware Execution Model



- **GPU 由多个并行核构成，每个核是一个多线程SIMD处理器（包含多个车道（Lanes））**
- **CPU 发送整个 “grid” 到GPU，由GPU将这些线程块分发到多个核上（每个线程块在一个核上运行）**
  - GPU上核的数量对程序员而言是透明的

Simplified CUDA Programming Model

- **计算由大量的相互独立的线程(*CUDA threads* or *microthreads*) 完成，这些线程组合成线程块（*thread blocks*）**

```
// C version of DAXPY loop.
void daxpy(int n, double a, double*x, double*y)
{  for (int i=0; i<n; i++)
       y[i] = a*x[i] + y[i]; }

// CUDA version.
__host__   // Piece run on host processor.
int nblocks = (n+255)/256; // 256 CUDA threads/block
daxpy<<<nblocks,256>>>(n,2.0,x,y);

__device__   // Piece run on GP-GPU.
void daxpy(int n, double a, double*x, double*y)
{  int i = blockIdx.x*blockDim.x + threadId.x;
   if (i<n) y[i]=a*x[i]+y[i]; }
```

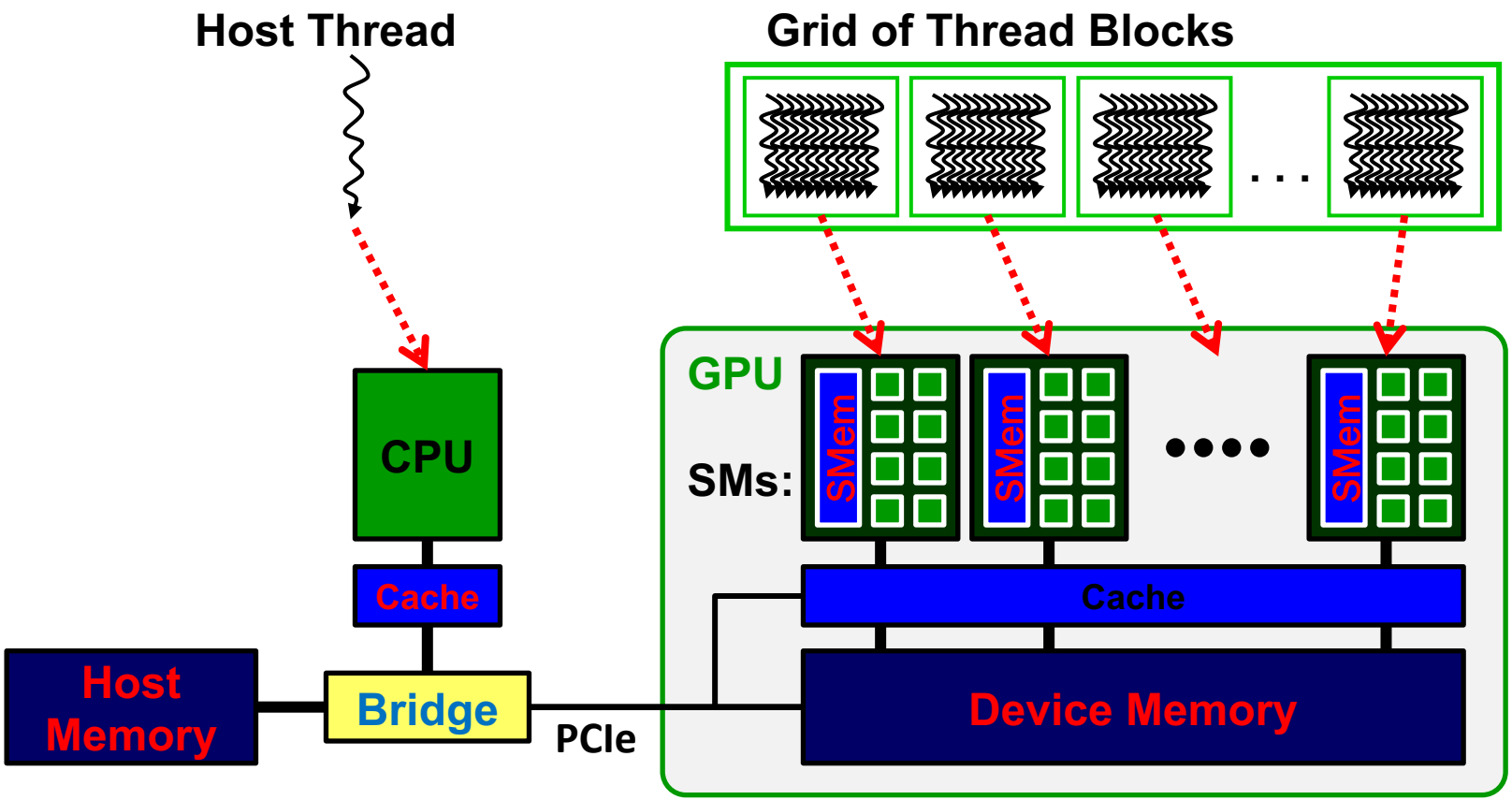
NVIDIA Instruction Set Arch.

- ISA 是硬件指令集的抽象
  - “Parallel Thread Execution (PTX)”
  - 使用虚拟寄存器
  - 用软件将其翻译成机器码
  - Example:

```
shl.s32  R8, blockIdx, 9       ; Thread Block ID * Block size (512 or 2^9)
add.s32  R8, R8, threadIdx  ; R8 = i = my CUDA thread ID
ld.global.f64      RD0, [X+R8]   ; RD0 = X[i]
ld.global.f64      RD2, [Y+R8]   ; RD2 = Y[i]
mul.f64   RD0, RD0, RD4      ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64   RD0, RD0, RD2      ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64    [Y+R8], RD0  ; Y[i] = sum (X[i]*a + Y[i])
```

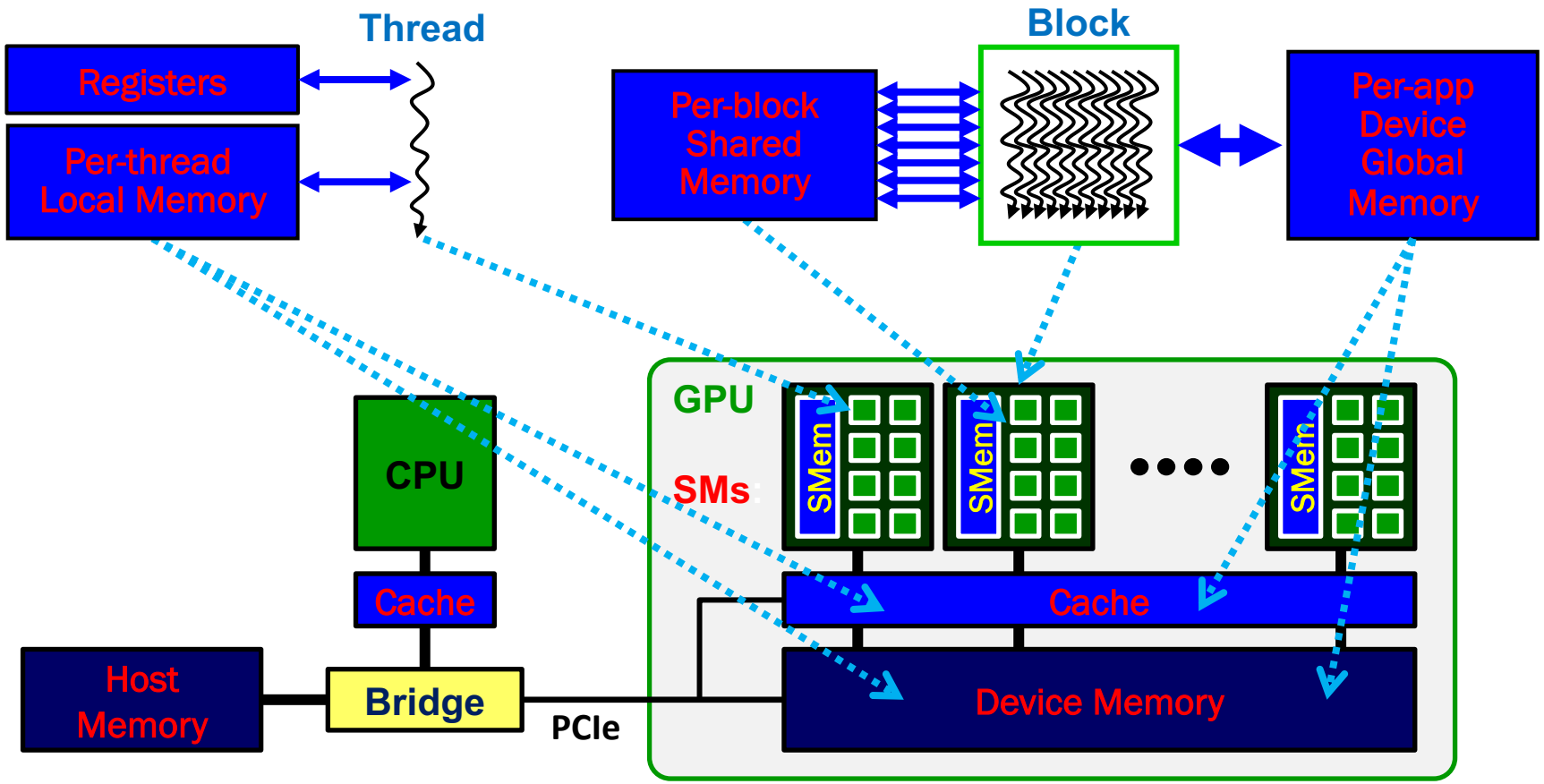
CUDA kernel maps to Grid of Blocks

```
kernel_func<<<nblk, nthread>>>(param, … );
```



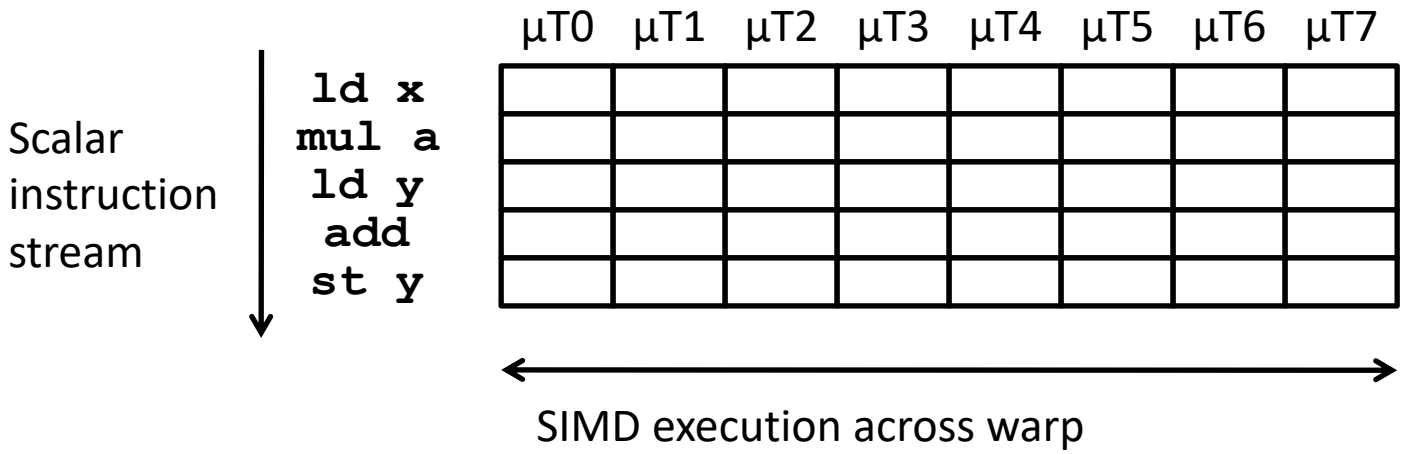
Thread blocks execute on a SM
Thread instructions execute on a SP

float myVar; **__shared__ float shVar;** **__device__ float glVar;**



CUDA： “Single Instruction, Multiple Thread”

- **GPUs 使用 SIMT模型, 每个CUDA线程的标量指令流汇聚在一起在硬件上以SIMD方式执行 (Nvidia groups 32 CUDA threads into a warp)**



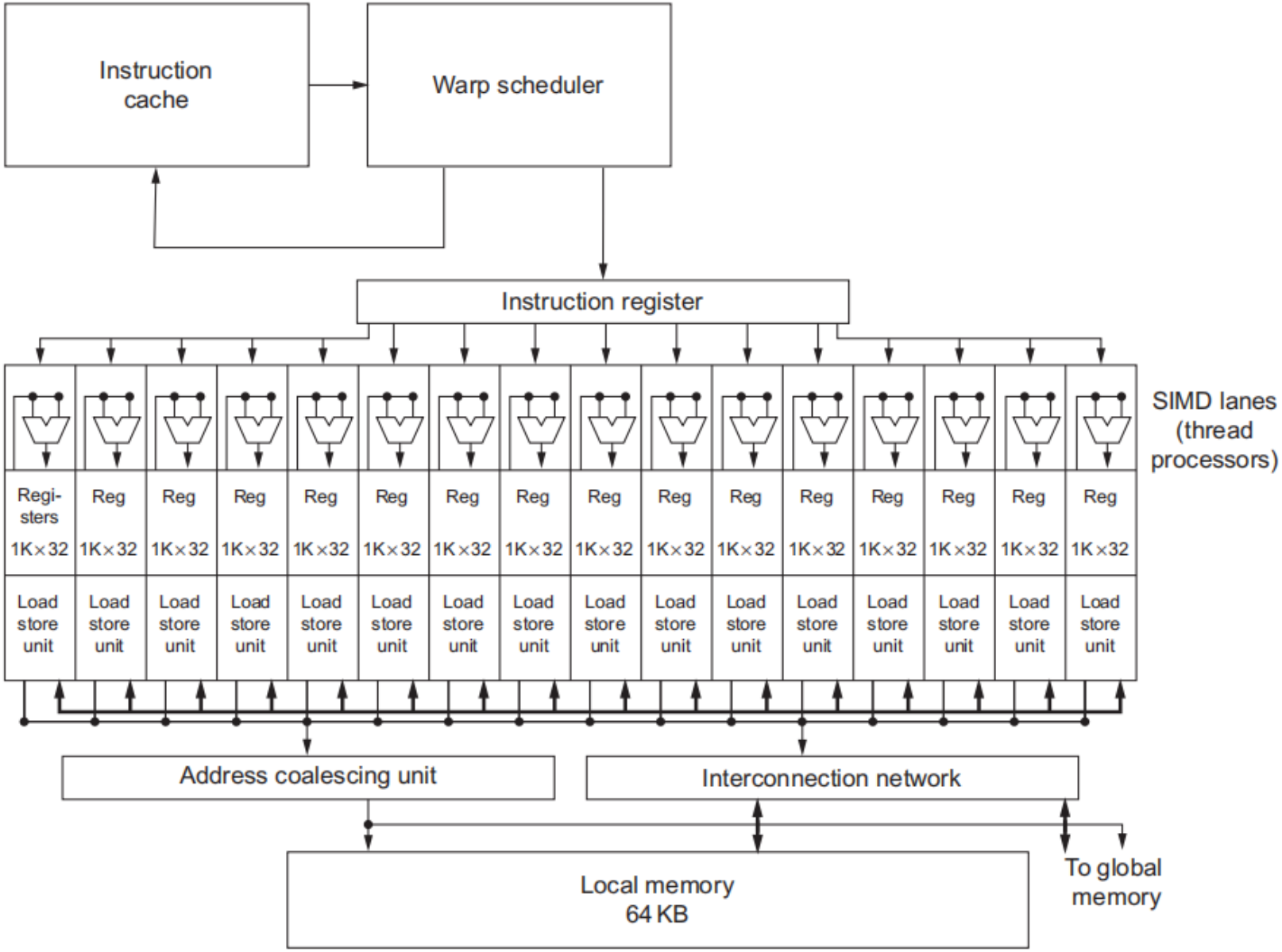
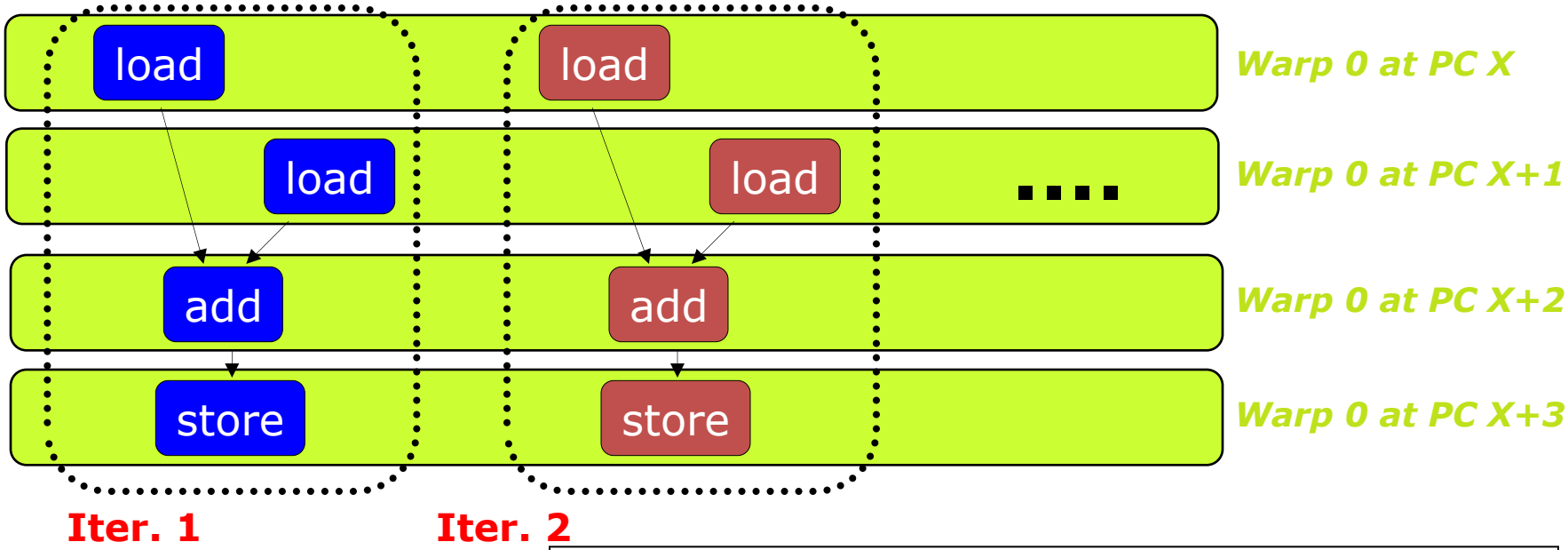


Figure 4.14 Simplified block diagram of a multithreaded SIMD Processor. It has 16 SIMD Lanes. The SIMD Thread Scheduler has, say, 64 independent threads of SIMD instructions that it schedules with a table of 64 program counters (PCs). Note that each lane has 1024 32-bit registers.

SPMD on SIMT Machine



Warp: A set of threads that execute the same instruction (i.e., at the same PC)

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

This particular model is also called:

SPMD: Single Program Multiple Data

A GPU executes it using the SIMT model:
Single Instruction Multiple Thread

SIMD vs. SIMT Execution Model

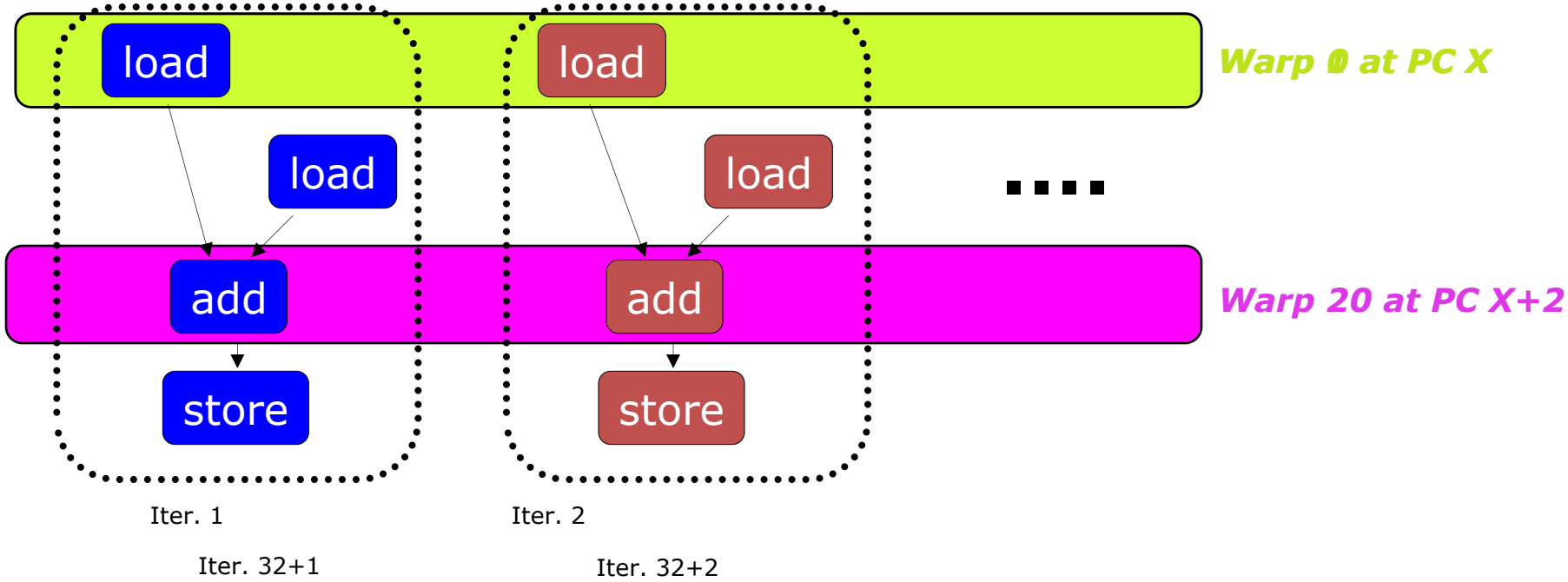
- **SIMD: 一条指令流（一串顺序的SIMD指令），每条指令对应多个数据输入（向量指令）**
- **SIMT: 多个指令流（标量指令）构成线程，这些线程动态构成warp。一个Warp处理多个数据元素**

- **SIMT 主要优点：**
  - 可以独立地处理线程,即每个线程可以在任何标量流水线上单独执行（ MIMD 处理模式）
  - 可以将线程组织成warp，即可以将执行相同指令流的线程构成warp，形成SIMD 处理模式, 以充分发挥SIMD处理的优势

Multithreading of Warps

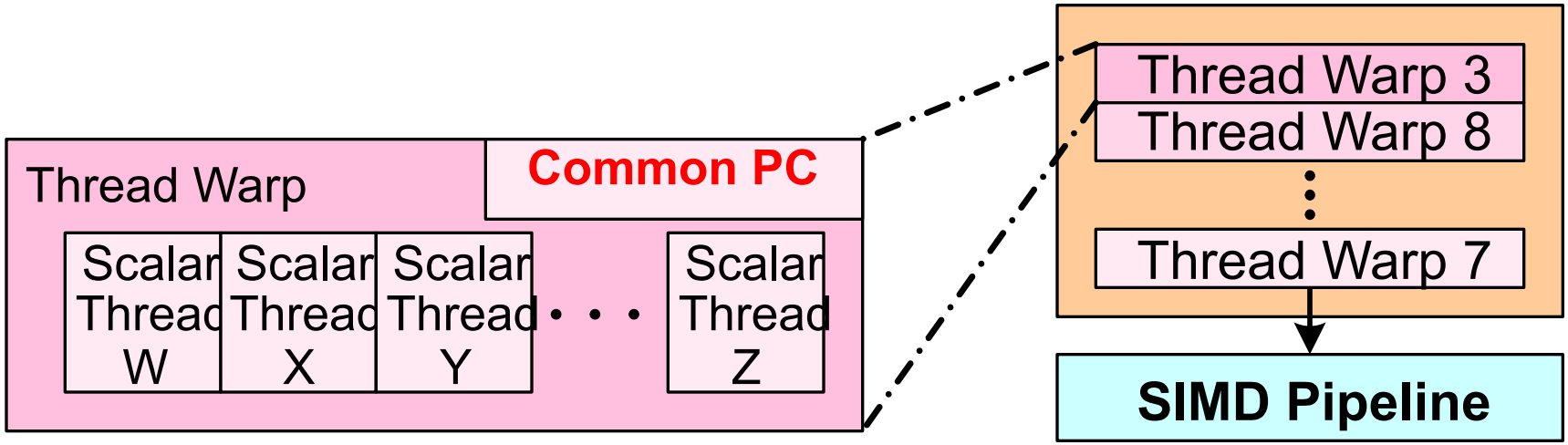
- 设一个warp由 32 threads构成
- 如果有32K次循环 → 1K 个warps
- 这些Warps可以在同一条流水线上交替执行
  - Fine grained multithreading of warps

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



Warps and Warp-Level FGMT

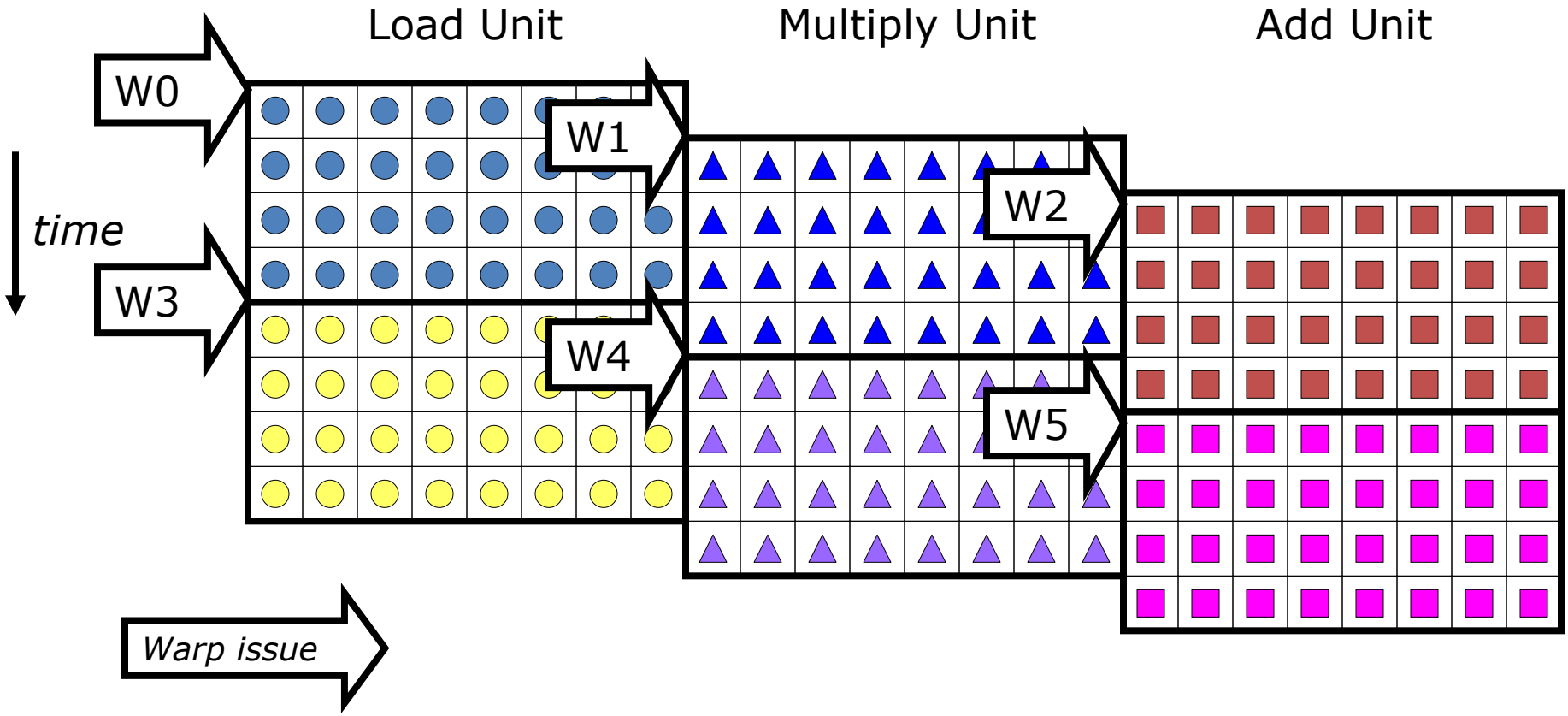
- **Warp: 执行相同指令（操作不同的数据元素）的一组线程**
- **同一Warp中的所有线程执行相同代码**
- **Warp: The threads that run lengthwise in a woven fabric …**



Warp Instruction Level Parallelism

Can overlap execution of multiple instructions

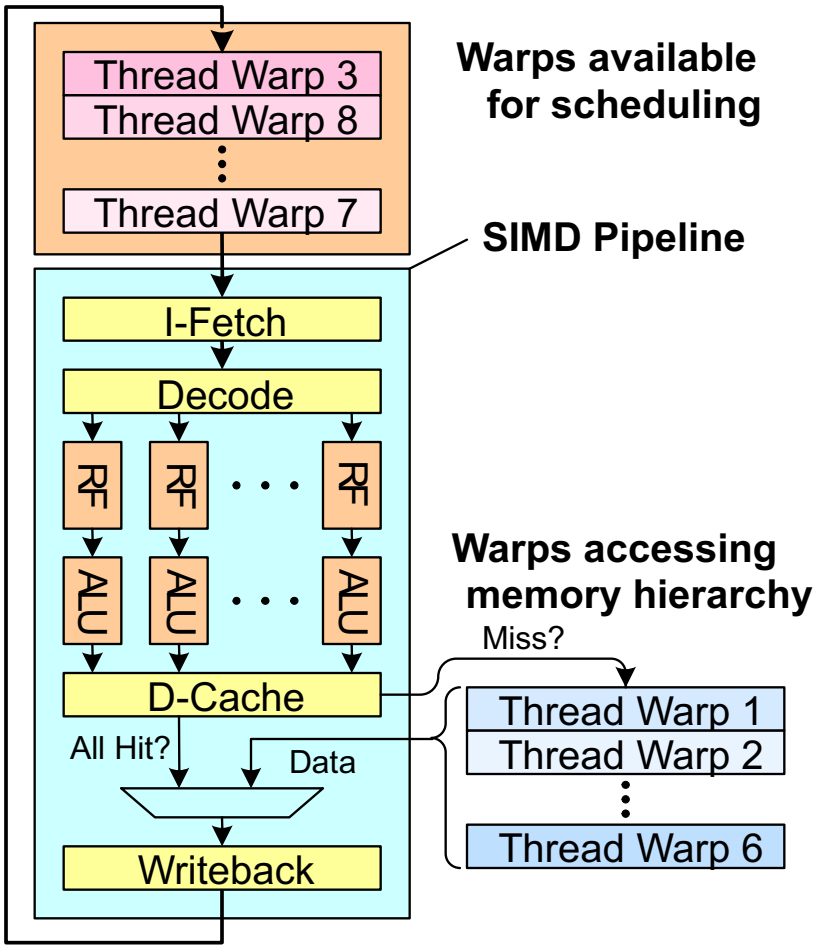
- Example machine has 32 threads per warp and 8 lanes
- Completes 24 operations/cycle while issuing 1 warp/cycle



Slide credit: Krste Asanovic

Latency Hiding via Warp-Level FGMT

- **Warp: 一组执行相同指令的线程（作用于不同的数据元素）**
- **Fine-grained multithreading**
  - 流水线上每次执行每个线程的一条指令 (No interlocking)
  - 通过warp的交叉执行来隐藏延时
- **所有线程的寄存器值保存在寄存器文件中**
- **FGMT 可以容忍长延时**
  - Millions of pixels



Warp-based SIMD vs. Traditional SIMD

- **传统的SIMD仅包含一个线程**
  - Lock step: 一条向量指令执行完，然后启动下一条向量指令
  - 编程模型为SIMD (no extra threads) → SW 需要知道向量长度
  - ISA 包含vector/SIMD指令

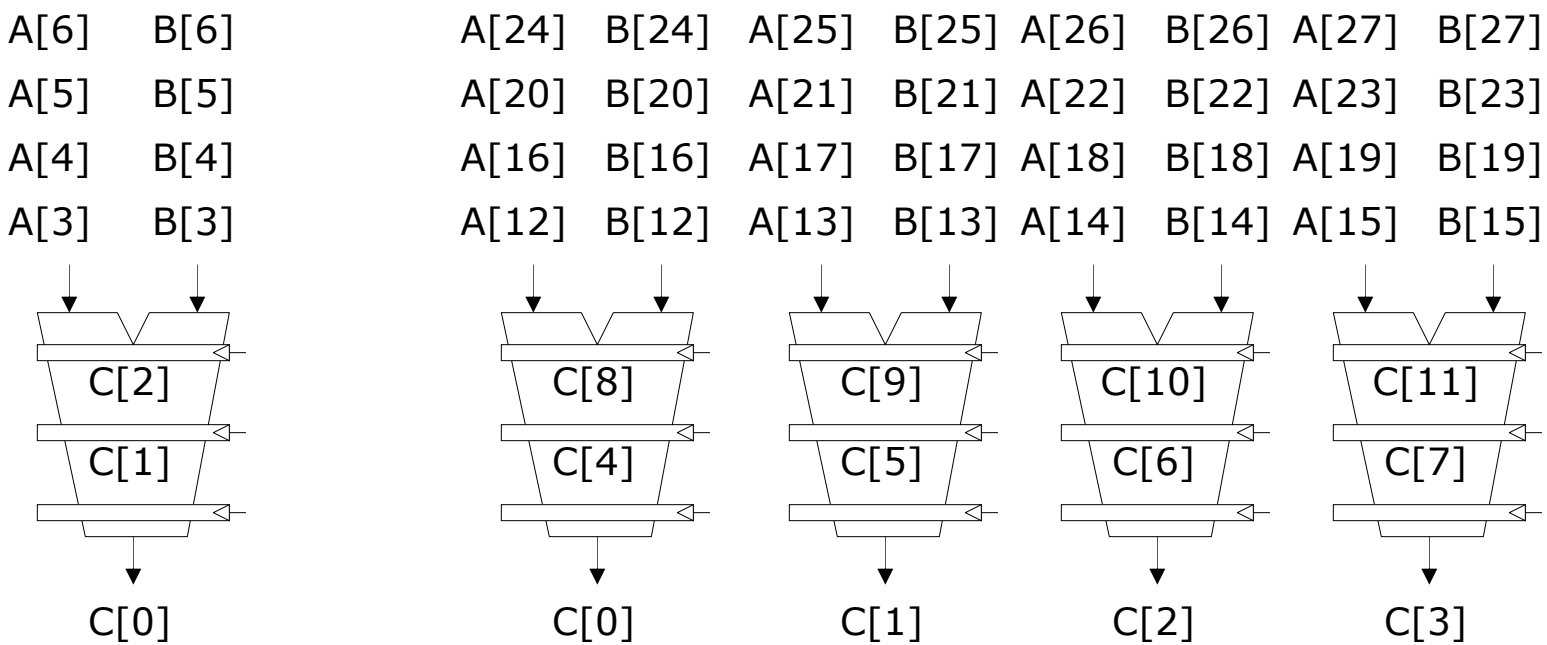
- **Warp-based SIMD 由多个标量线程构成，以SIMD方式执行(即所有的线程执行相同的指令)**
  - 不需要lock step
  - 每个线程可以单独对待(即可以映射到不同的warp中），编程模型不是SIMD
    - SW 不必知道向量长度
    - 多个线程可以动态构成warp
  - ISA是标量ISA → 可以动态形成逻辑上的向量指令
  - 是一种在SIMD硬件上实现的SPMD编程模型

Warp Execution (Recall the Slide)

32-thread warp executing ADD A[tid],B[tid] → C[tid]

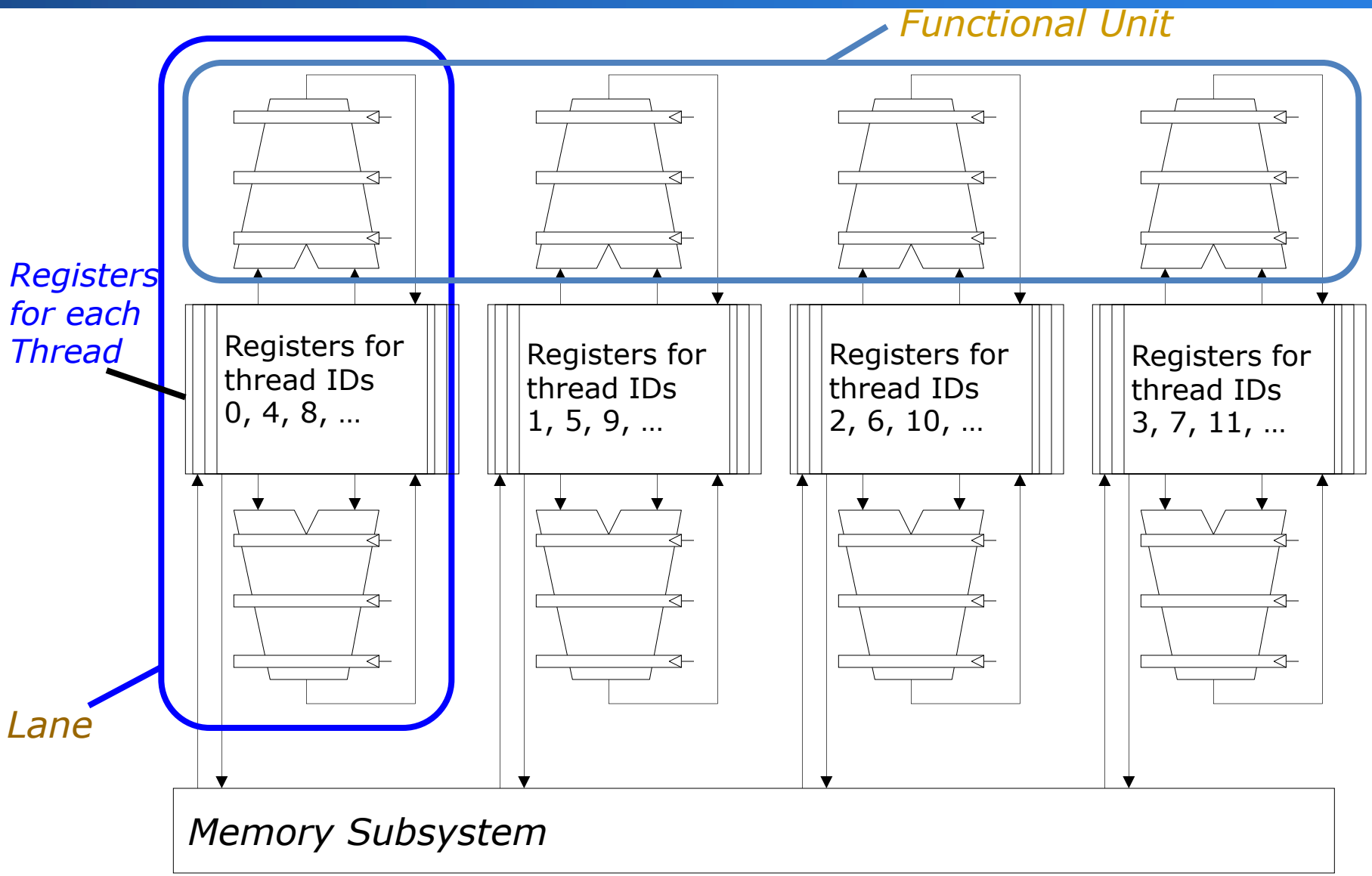
Execution using one pipelined functional unit

Execution using four pipelined functional units



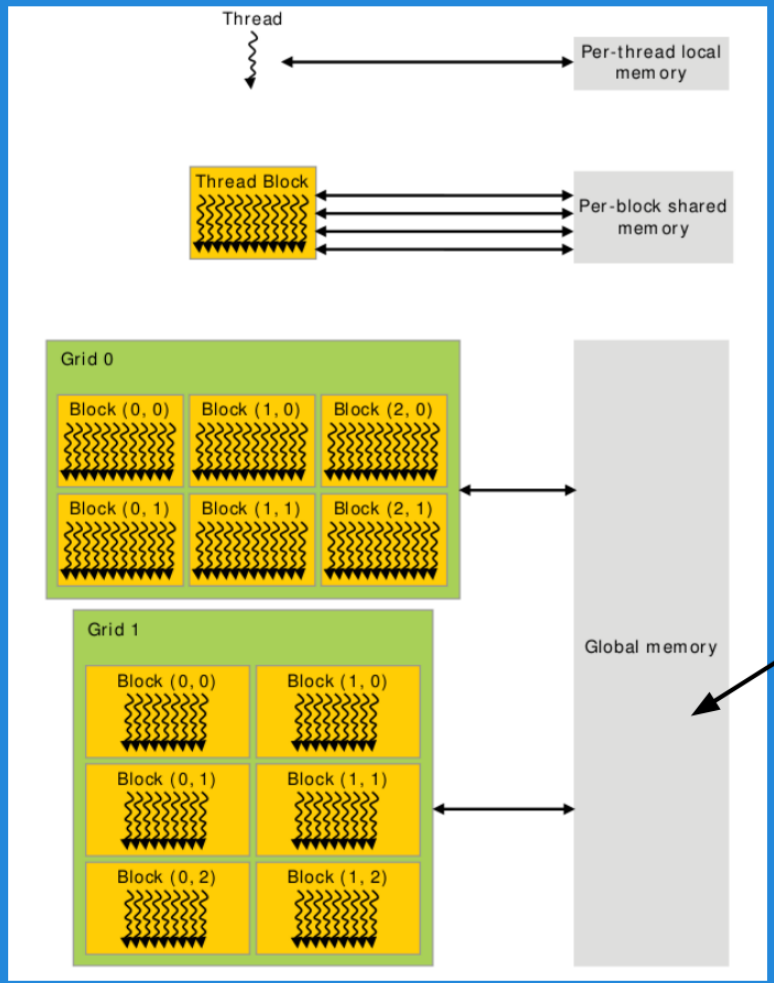
Slide credit: Krste Asanovic

SIMD Execution Unit Structure



Slide credit: Krste Asanovic

GPU Memory Hierarchy



constant memory

GPU Global Memory

Global memory is the “main” memory of the GPU. It has global scope and lifetime of the allocating program (or until cudaFree is called).

Global memory is similar to the heap in a C program.

Global memory is separate hardware from the GPU core (containing SM’s,
caches, etc). The vast majority of memory on GPU is global memory. If data doesn’t fit into global memory, you are going to process it in chunks that do fit in globalmemory.

GPUs have .5 - 24GB of global memory, with most now having ~2GB. Global memory latency is ~300ns on Kepler and ~600ns on Fermi

GPU Shared Memory

- Very fast memory located in the SM
- Same hardware as L1 cache, ~5ns of latency
- Maximum size of 48KB, but user configurable
- Scope of shared memory is the block

Registers

- Fastest “memory” possible, about 10x faster than shared Memory

- Most stack variables declared in kernels are stored in registers (example: float x)

- Statically indexed arrays stored on the stack are sometimes put in registers

Local Memory

- Local memory is everything on the stack that can't fit in registers. The scope of local memory is just the thread.

- Local memory is stored in global memory (much slower than registers!)

GPU L1 Cache

- Fermi - caches local & global memory (local memory's scope is local to the thread and it resides off-chip)
- Kepler, Maxwell - only caches local memory
- same hardware as shared memory
- configurable size (16, 32, 48KB)
- each SM has its own L1 cache

GPU L2 Cache

- caches all global & local memory accesses
- ~1MB in size
- shared by all SM's

Constant Memory

- Used for constants that cannot be compiled into program
- Constants must be set from host before running kernel
- Constant memory is global memory with a special cache
- 64KB for user, 64KB for compiler (kernel arguments are passed through constant memory)

Constant Cache

- 8KB cache on each SM specially designed to broadcast a single memory address to all threads in a warp (called static indexing)
- Can also load any statically indexed data
- through constant cache using “load uniform” (LDU) instruction

A few terms

- SM, SP, lane
- Grid, block, warp, thread
- SIMT, SIMD, SPMD, FGMT

flynn

programming

Nvidia

GPU Memory Hierarchy

SIMD

SIMT

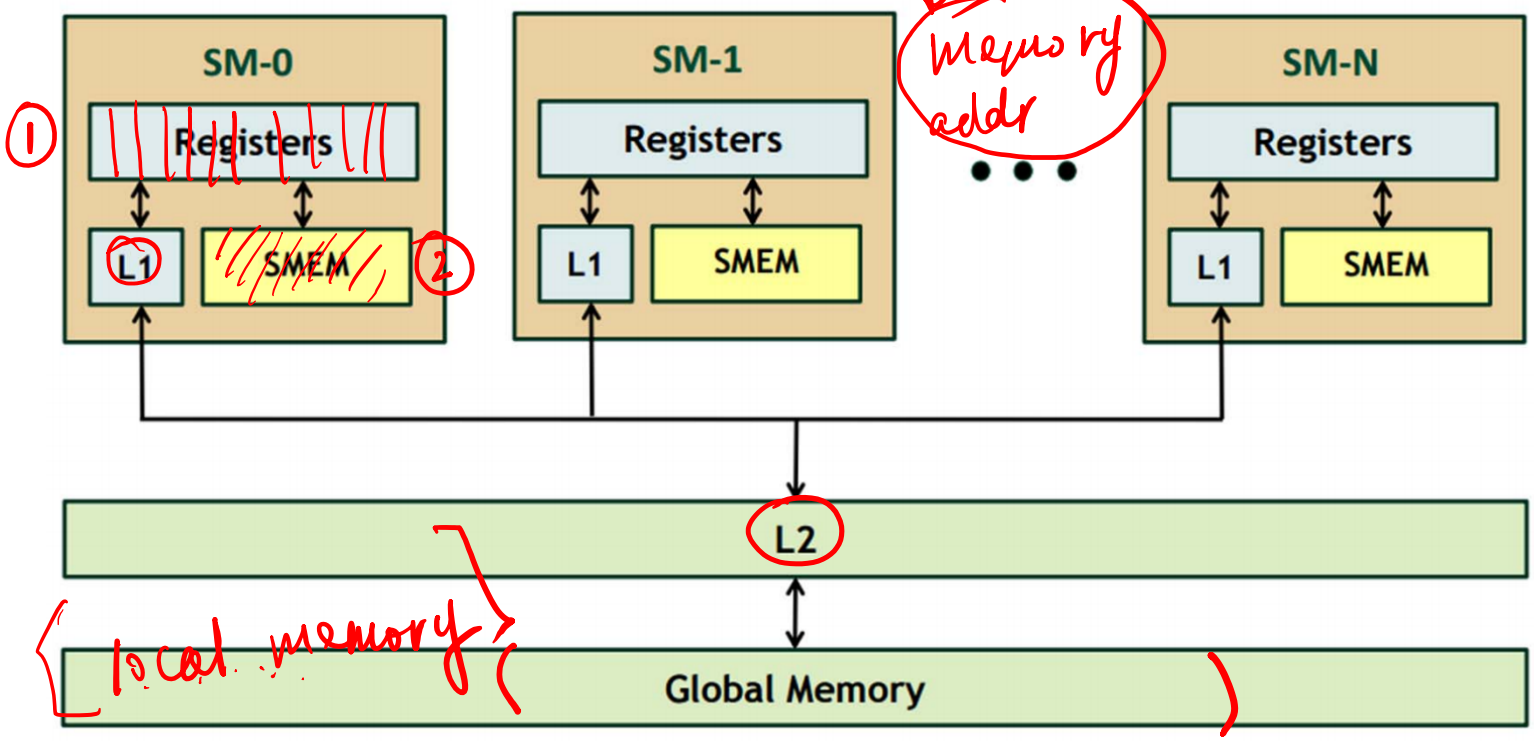
Ld $r0, 4 $(r1)

① CPU vs GPU

NVIDIA Fermi Memory Hierarchy

② why GPU is faster?

memory addr

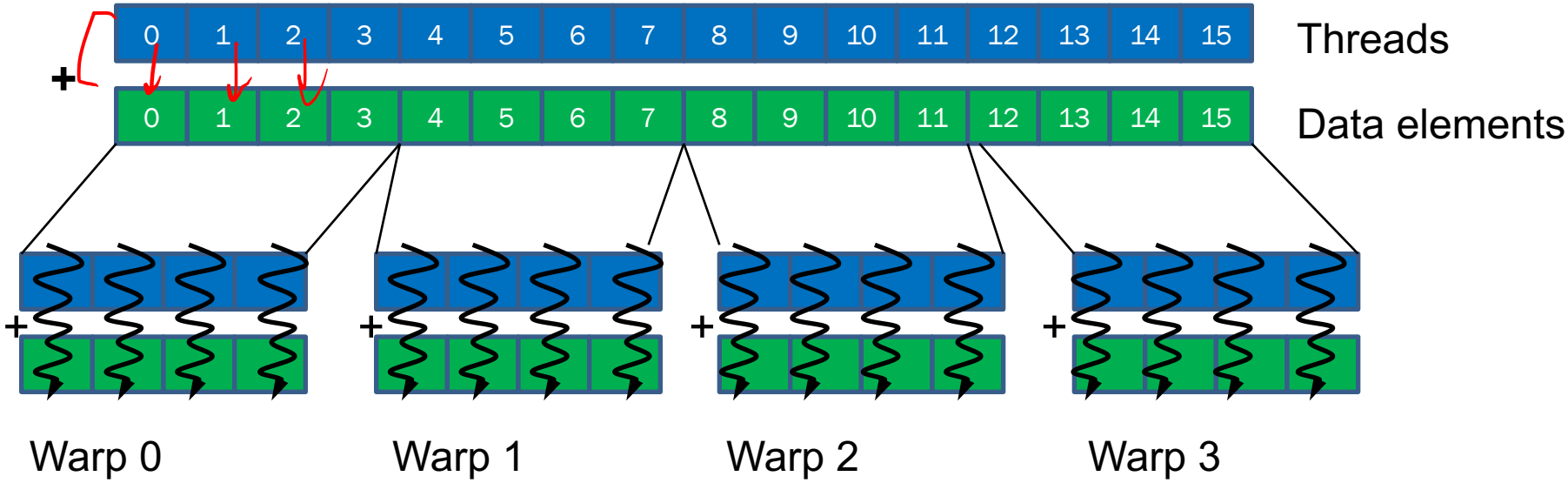


SIMT Memory Access

SIMT

- 不同线程的相同指令使用线程id来存取不同的数据元素

Let's assume N=16, 4 threads per warp → 4 warps



Slide credit: Hyesoon Kim

Global Memory Access

Each thread issues memory accesses to data types of varying sizes, perhaps as small as 1 byte entities

Given an address to load or store, memory returns/updates "segments" of either 32 bytes, 64 bytes or 128 bytes

Maximizing bandwidth:

- Operate on an entire 128 byte segment for each memory transfer

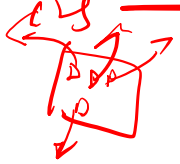
Understanding Global Memory Access

Start with memory request by smallest numbered thread.

Find the memory segment that contains the address (32, 64 or 128 byte segment, depending on data type)

Find other active threads requesting addresses within that segment and coalesce

SIMT

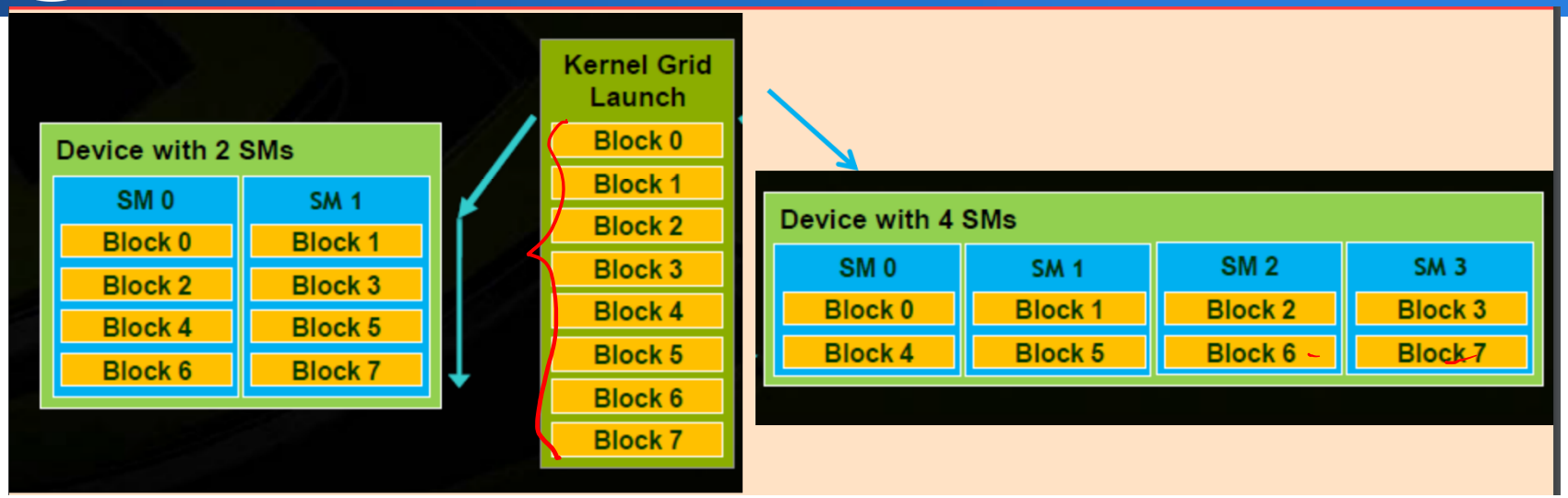


Reduce transaction size if possible

Access memory and mark threads as "inactive"

Repeat until all threads in half-warp are serviced
(Includes Tesla and GTX platforms)

Thread Block Scheduling



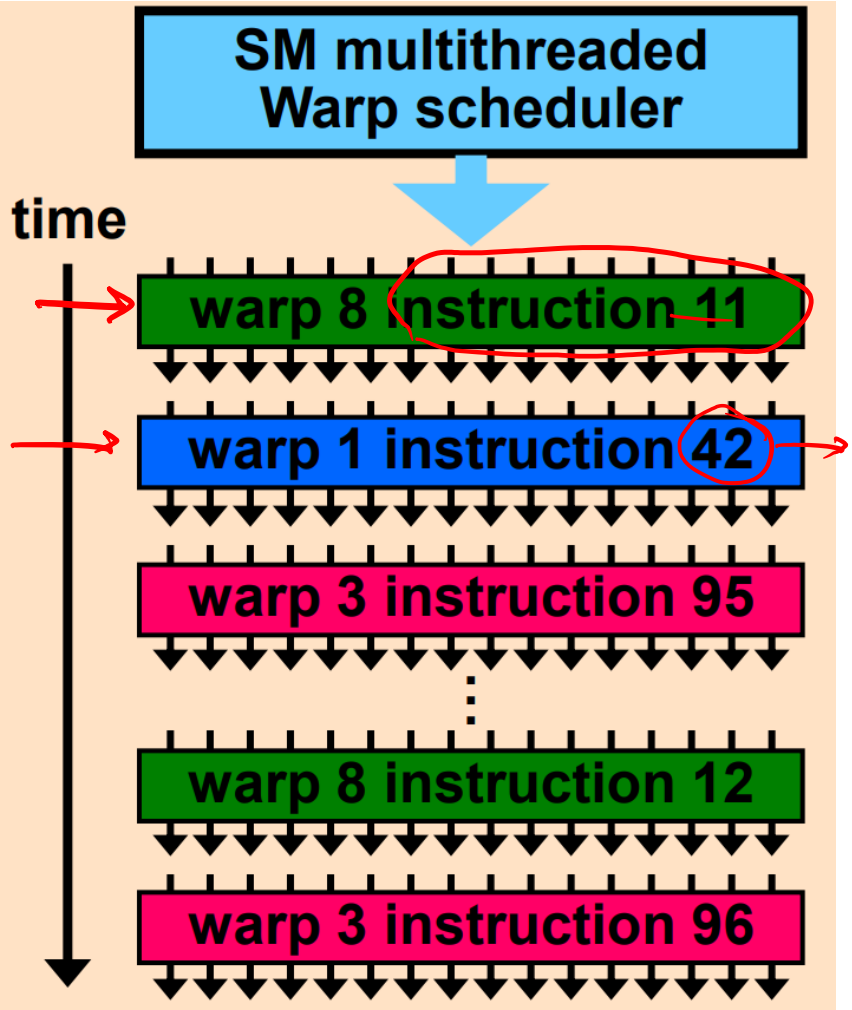
- Each TB allocated to an SM; Multiple TBs in a SM
- Threads in a thread block may need to cooperate
  - Load/Store from memory together
  - Share results (through shared memory)
  - Synchronize with each other
- Thread Blocks can execute in any order

Resident Thread Blocks

- No. of thread blocks allocated to SM is limited by
  - Maximum Thread blocks
  - Maximum number of threads
  - Registers (Registers per thread * Threads per Block * # Blks)
  - Shared Memory (Shared Memory per Block * # Blks)
- Warps of resident threadblocks are context switched to hide (long) latency

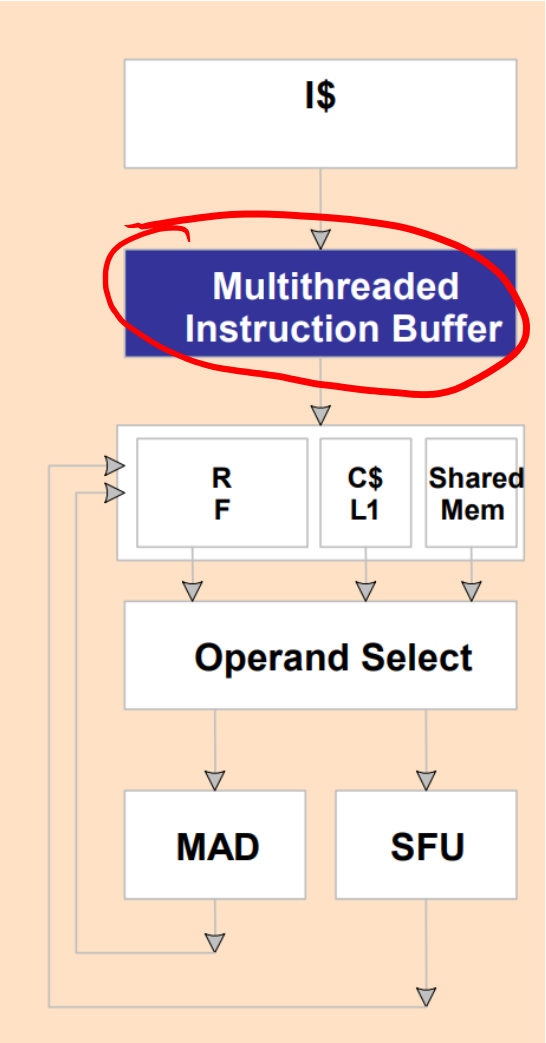
SM Warp Scheduling

- Each SM launches Warps of Threads warp level MT
  - 2 levels of parallelism
  - Shared instruction fetch per 32 threads (warps)
- SM hardware implements zero overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute the same instruction when selected



SM Instruction Buffer – Warp Scheduling

- Fetch one warp instruction/cycle
  - from instruction cache
  - into any instruction buffer slot
- Issue one "ready-to-go" warp instruction/cycle
  - from any warp - instruction buffer slot
  - operand scoreboarding used to prevent hazards
- Issue selection based on round robin or age of warp
- SM broadcasts the same instruction to 32 threads of a Warp

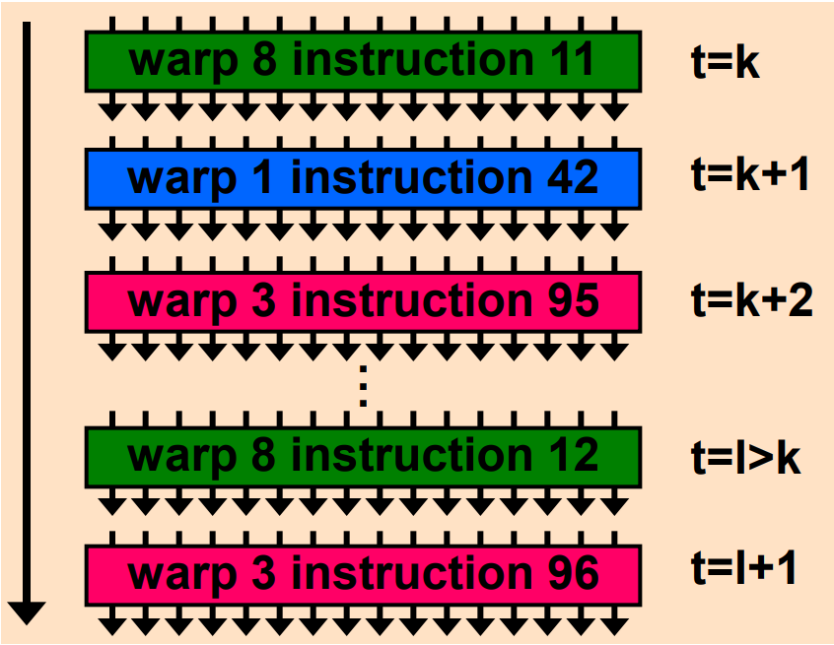


Scoreboarding

- How to determine if a thread is ready to execute?
- A scoreboard is a table in hardware that tracks
  - instructions being fetched, issued, executed
  - resources (functional units and operands) they need
  - which instructions modify which registers
- Old concept from CDC 6600 (1960s)

Scoreboarding Example

Consider three separate instruction streams: warp1, warp3 and warp8

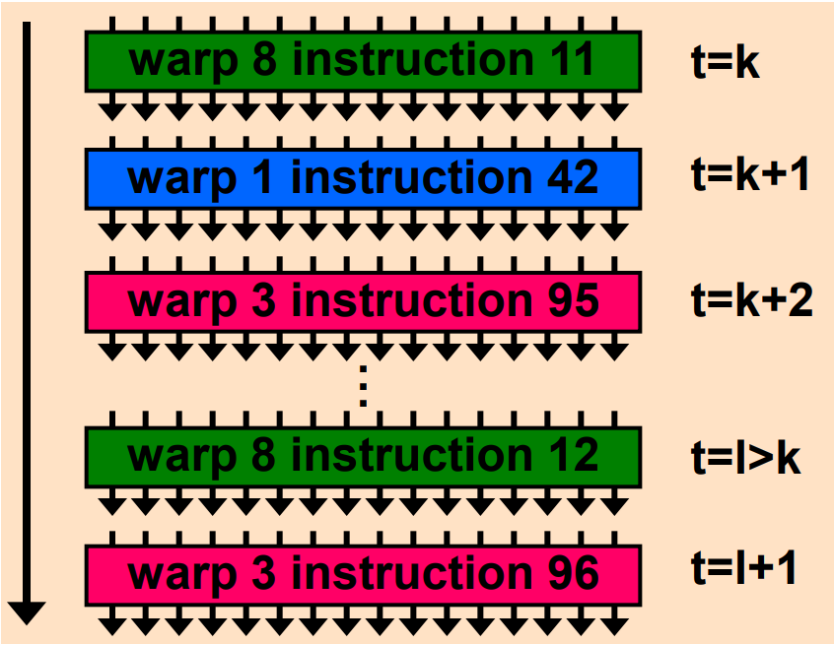


Warp	Current Instruction	Instruction State
Warp 1	42	Computing
Warp 3	95	Computing
Warp 8	11	Operands ready to go
...		

Schedule at time k

Scoreboarding Example

Consider three separate instruction streams: warp1, warp3 and warp8

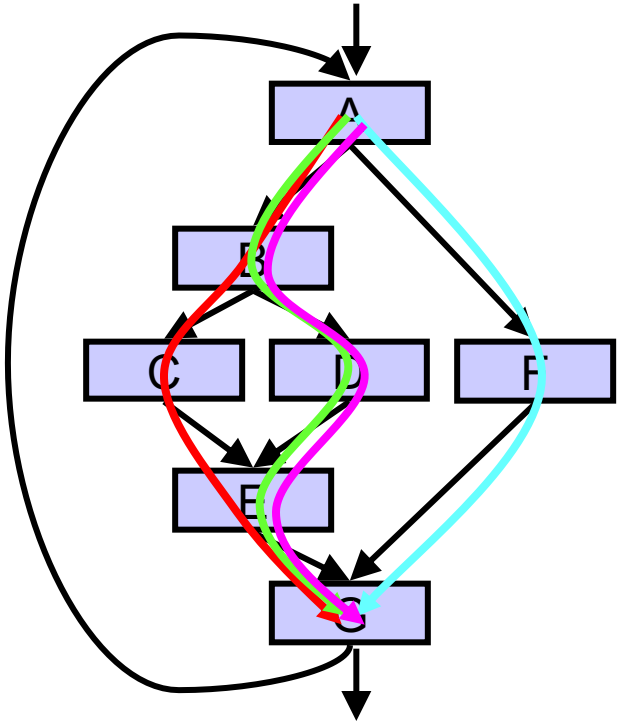


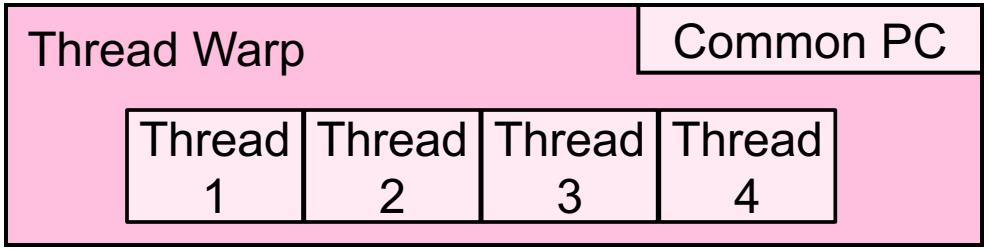
Warp	Current Instruction	Instruction State
Warp 1	42	Ready to write result
Warp 3	95	Computing
Warp 8	11	Computing
…		

Schedule at time k+1

Threads Can Take Different Paths in Warp-based SIMD

- 每个线程可以包含控制流指令
- 这些线程可以执行不同的控制流路径

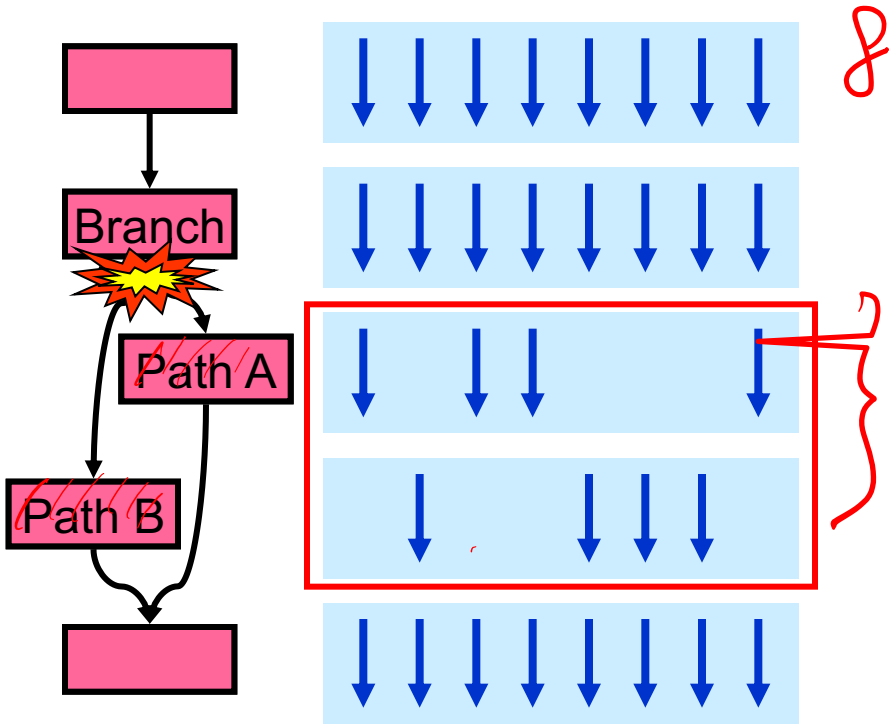




Control Flow Problem in GPUs/SIMT

- GPU 控制逻辑使用 SIMD流水线 以节省资源
  - 这些标量线程构成 warp

- 当一个WARP中的线程分支到不同的执行路径时，产生分支发散（Branch divergence）



与向量处理机模型的条件执行类似
（Vector Mask and Masked Vector Operations?）

Conditional Branching

- **与向量结构类似, GPU 使用内部的屏蔽字(masks)**
- **还使用了（SIMT-Stack）**
  - 分支同步堆栈
    - 保存分支的路径地址
    - 保存该路径的SIMD lane 屏蔽字(mask)
      - 即指示哪些车道可以提交结果
  - 指令标记(instruction markers）
    - 管理何时分支（divergence）到多个执行路径，何时路径汇合(converge)
- **PTX层**
  - CUDA线程的控制流由PTX分支指令(branch、call、return and exit）控制 以及
  - 由程序员指定的每个线程车道的1-bit谓词寄存器
- **GPU硬件指令层,控制流包括：**
  - 分支指令(branch,jump call return)
  - 特殊的指令用于管理分支同步栈
  - GPU硬件为每个SIMD thread 提供堆栈 保存分支的路径
  - GPU硬件指令带有控制每个线程车道的1-bit谓词寄存器

Branch divergence

- **硬件跟踪各μthreads转移的方向（判定哪些是成功的转移，哪些是失败的转移）**
- **如果所有线程所走的路径相同，那么可以保持这种 SIMD 执行模式**
- **如果各线程选择的方向不一致，那么创建一个屏蔽（mask）向量来指示各线程的转移方向（成功、失败）**
- **继续执行分支失败的路径，将分支成功的路径压入硬件堆栈（分支同步堆栈），待后续执行**
- **SIMD 车道何时执行分支同步堆栈中的路径?**
  - 通过执行pop操作，弹出执行路径以及屏蔽字，执行该转移路径
  - SIMD lane完成整个分支路径执行后再执行下一条指令 称为converge(汇聚）
  - 对于相同长度的路径，IF-THEN-ELSE 操作 的 效率平均为50%

Example

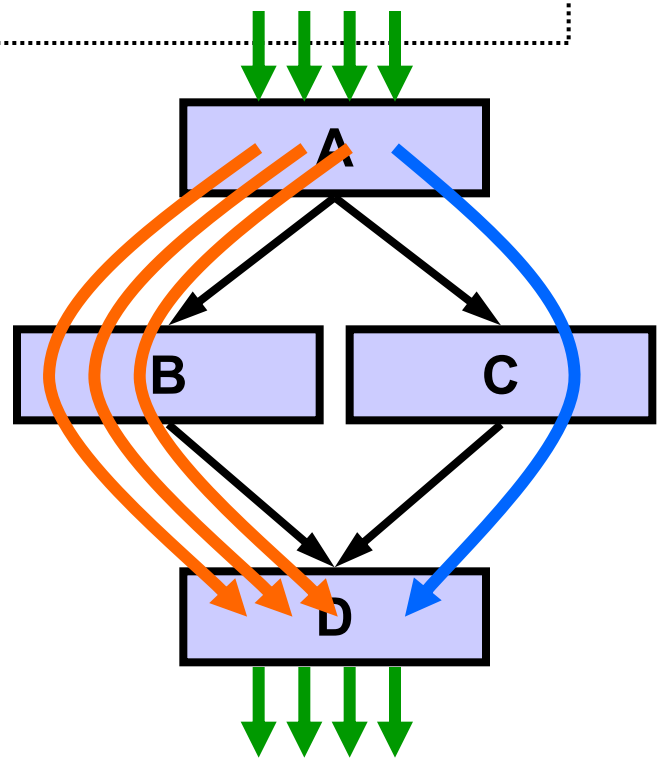
```
if (X[i] != 0)
        X[i] = X[i] – Y[i];
else X[i] = Z[i];
```

```
        ld.global.f64  RD0, [X+R8]  ; RD0 = X[i]
        setp.neq.s32  P1, RD0, #0   ; P1 is predicate register 1
        @!P1, bra  ELSE1, *Push     ; Push old mask, set new mask bits
                                    ; if P1 false, go to ELSE1
        ld.global.f64 RD2, [Y+R8]   ; RD2 = Y[i]
        sub.f64 RD0, RD0, RD2       ; Difference in RD0
        st.global.f64 [X+R8], RD0   ; X[i] = RD0
        @P1, bra  ENDIF1, *Comp     ; complement mask bits
                                    ; if P1 true, go to ENDIF1
ELSE1:  ld.global.f64 RD0, [Z+R8]   ; RD0 = Z[i]
        st.global.f64 [X+R8], RD0   ; X[i] = RD0
ENDIF1: <next instruction>, *Pop    ; pop to restore old mask
```

then

Branch Divergence Handling

```
A;
if (some condition) {
    B;
} else {
    C;
}
D;
```

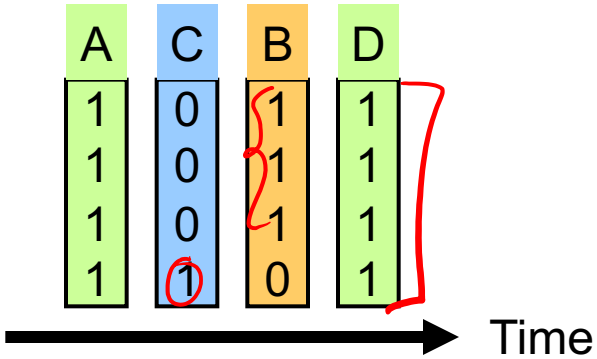


One per warp

Control Flow Stack

	Next PC	Recv PC	Active Mask
TOS →	D	--	1111
	B	D	1110
	C	D	0001

Execution Sequence



SIMT stack operation

```
do {
  t1 = tid*N;          // A
  t2 = t1 + i;
  t3 = data1[t2];
  t4 = 0;
  if( t3 != t4 ) {
    t5 = data2[t2]; // B
    if( t5 != t4 ) {
      x += 1;          // C
    } else {
      y += 2;          // D
    }
  } else {
    z += 3;            // F
  }
  i++;                 // G
} while( i < N );
```

Example CUDA C source code for illustrating SIMT stack operation

```
A:    mul.lo.u32      t1, tid, N;
      add.u32         t2, t1, i;
      ld.global.u32   t3, [t2];
      mov.u32         t4, 0;
      setp.eq.u32     p1, t3, t4;
@p1   bra             F;
B:    ld.global.u32   t5, [t2];
      setp.eq.u32     p2, t5, t4;
@p2   bra             D;
C:    add.u32         x, x, 1;
      bra             E;
D:    add.u32         y, y, 2;
E:    bra             G;
F:    add.u32         z, z, 3;
G:    add.u32         i, i, 1;
      setp.le.u32     p3, i, N;
@p3   bra             A;
```

Example PTX assembly code for illustrating SIMT stack operation.

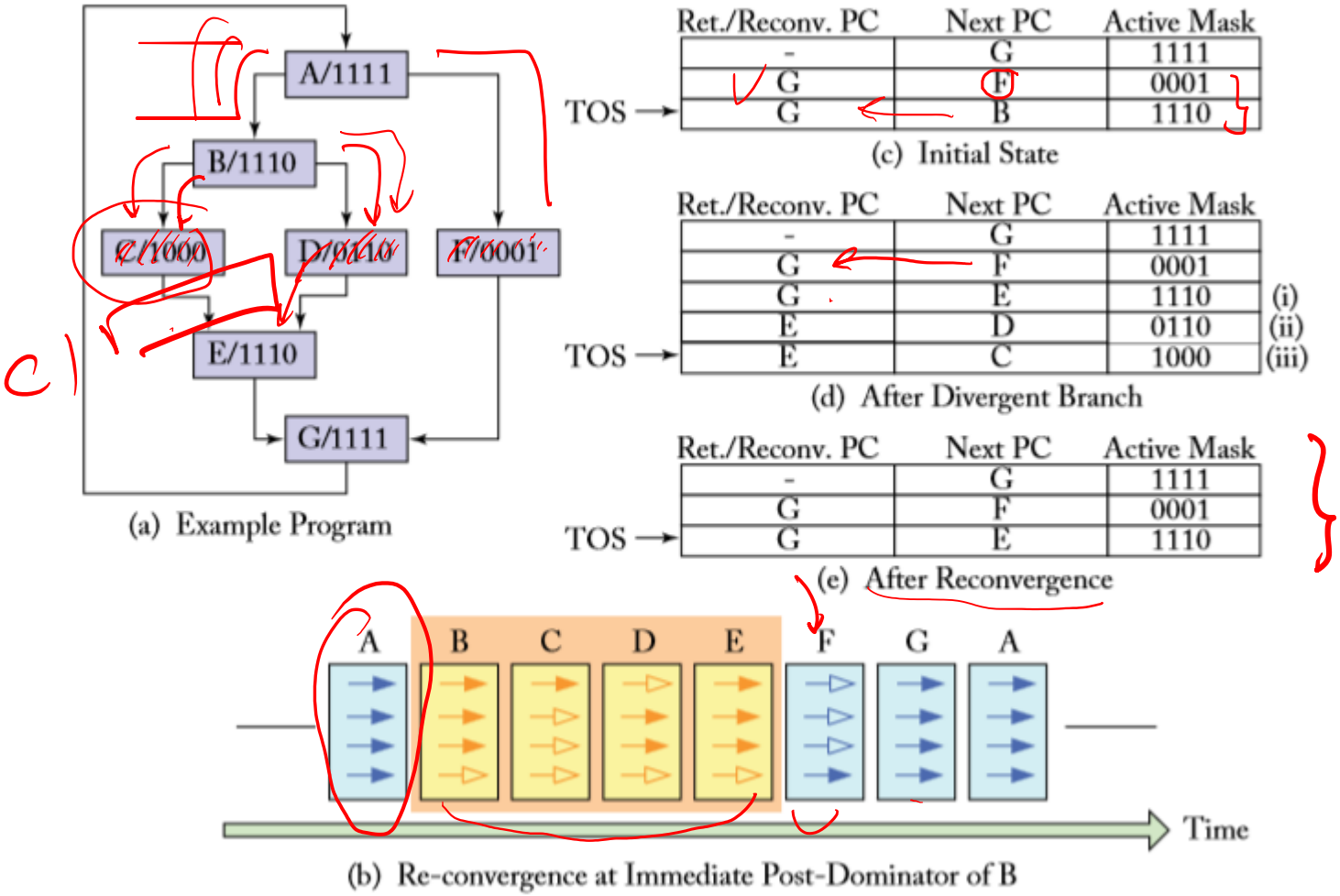


Figure 3.4: Example of SIMT stack operation (based on Figure 5 from Fung et al. [2007]).

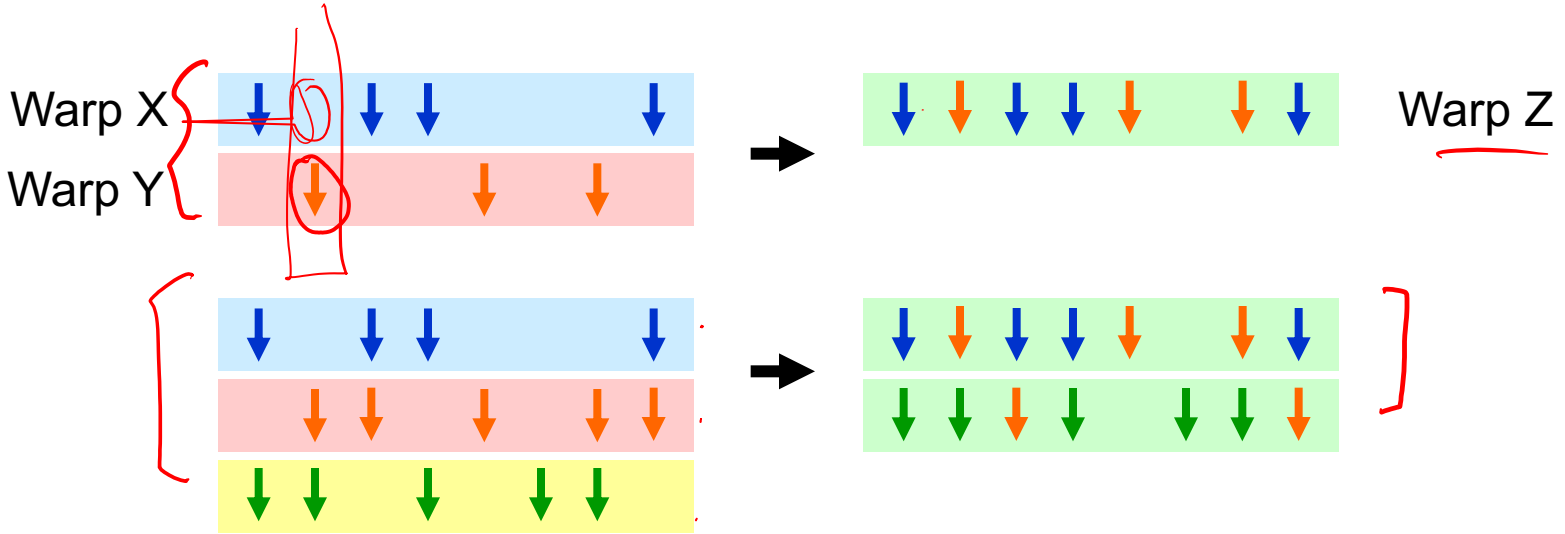
Remember: Each Thread Is Independent

- **SIMT 主要优点：**
  - 可以独立地处理线程,即每个线程可以在任何标量流水线上单独执行（ MIMD 处理模式）
  - 可以将线程组织成warp，即可以将执行相同指令流的线程构成warp，形成SIMD 处理模式, 以充分发挥SIMD处理的优势

- **如果有许多线程，对具有相同PC值的线程可以将它们动态组织到一个warp中**
- **这样可以减少“分支发散” →提高SIMD 利用率**
  - SIMD 利用率: 执行有用操作的SIMD lanes的比例 (即, 执行活动线程的比例）

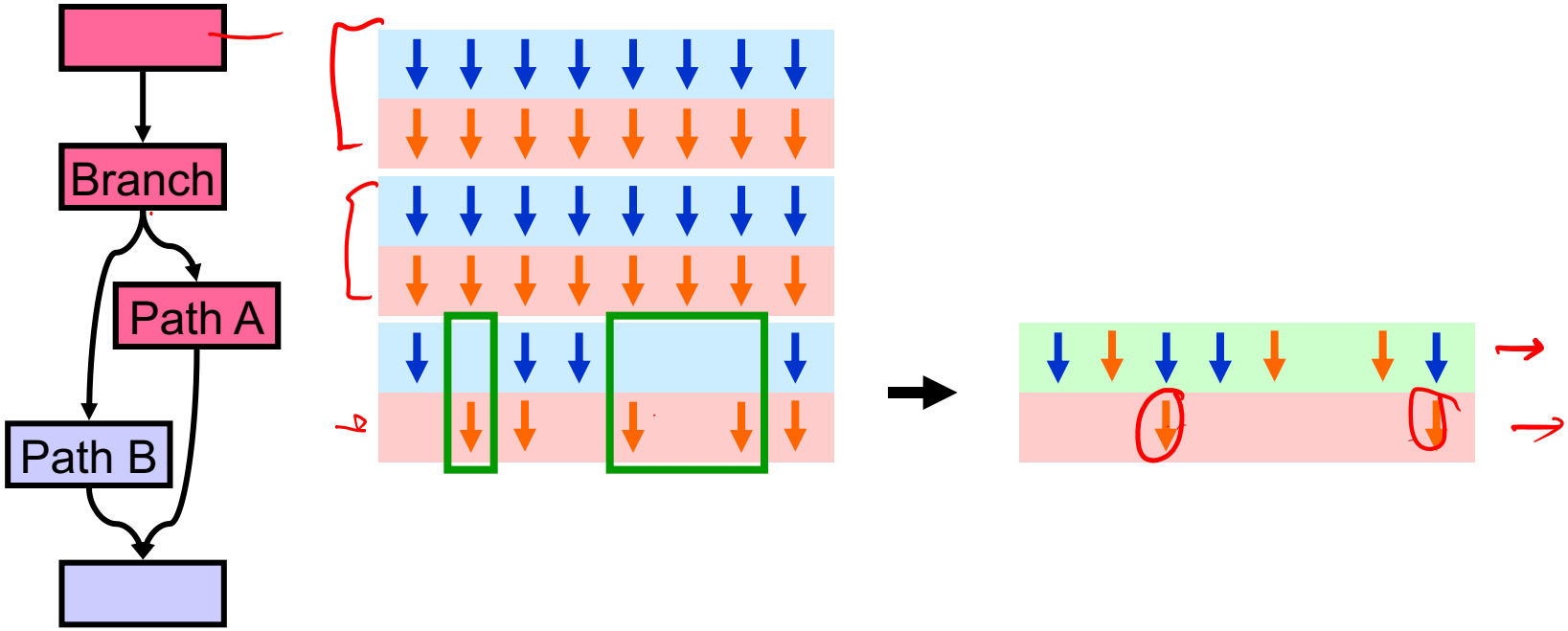
Dynamic Warp Formation/Merging

- **Idea:** 分支发散之后，动态合并执行相同指令的线程
  - 从那些等待的warp中形成新的warp
  - 足够多的线程分支到每个路径，有可能创建完整的新warp



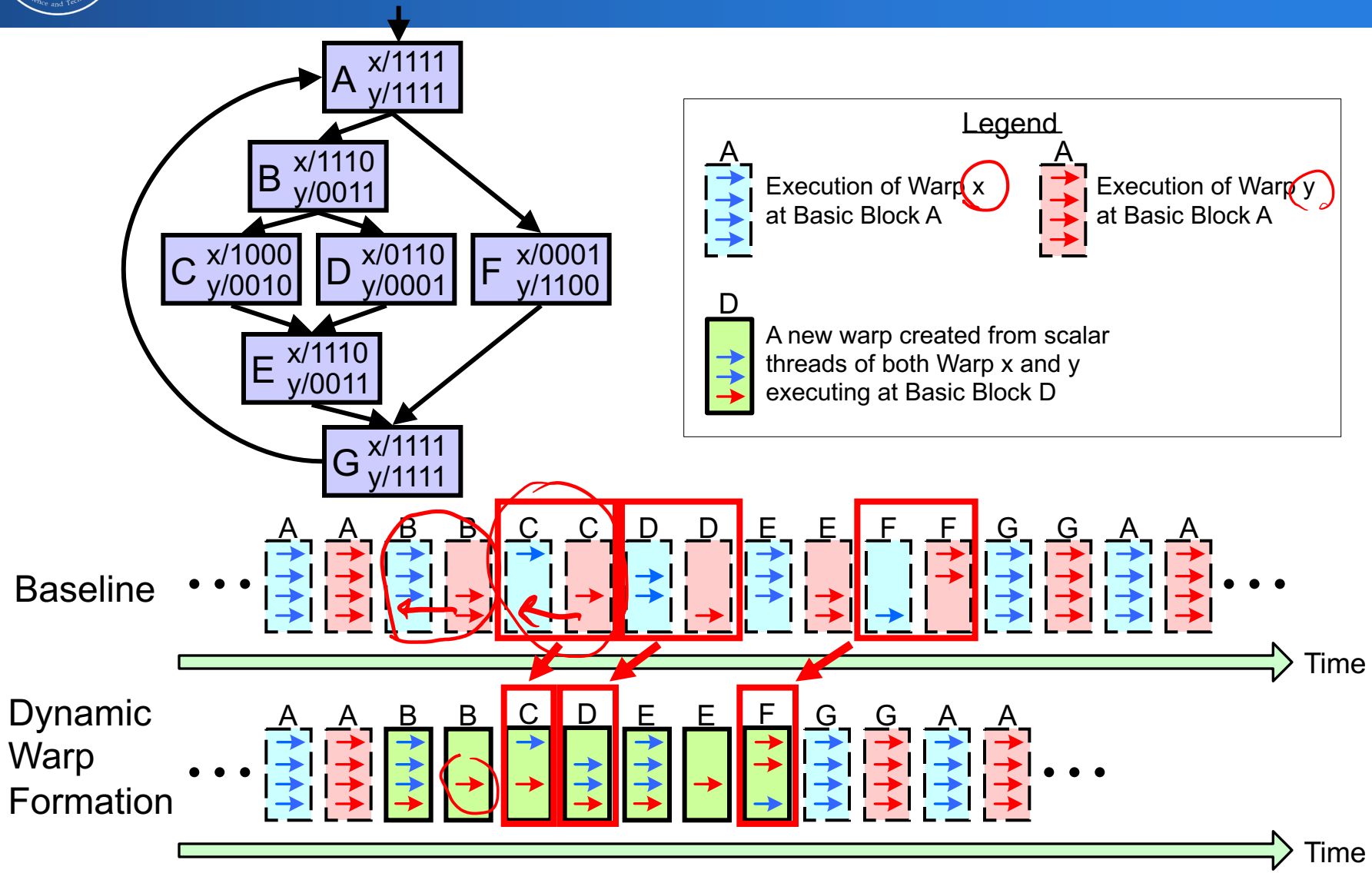
Dynamic Warp Formation/Merging

- Idea:
  - 分支发散之后，动态合并执行相同指令的线程

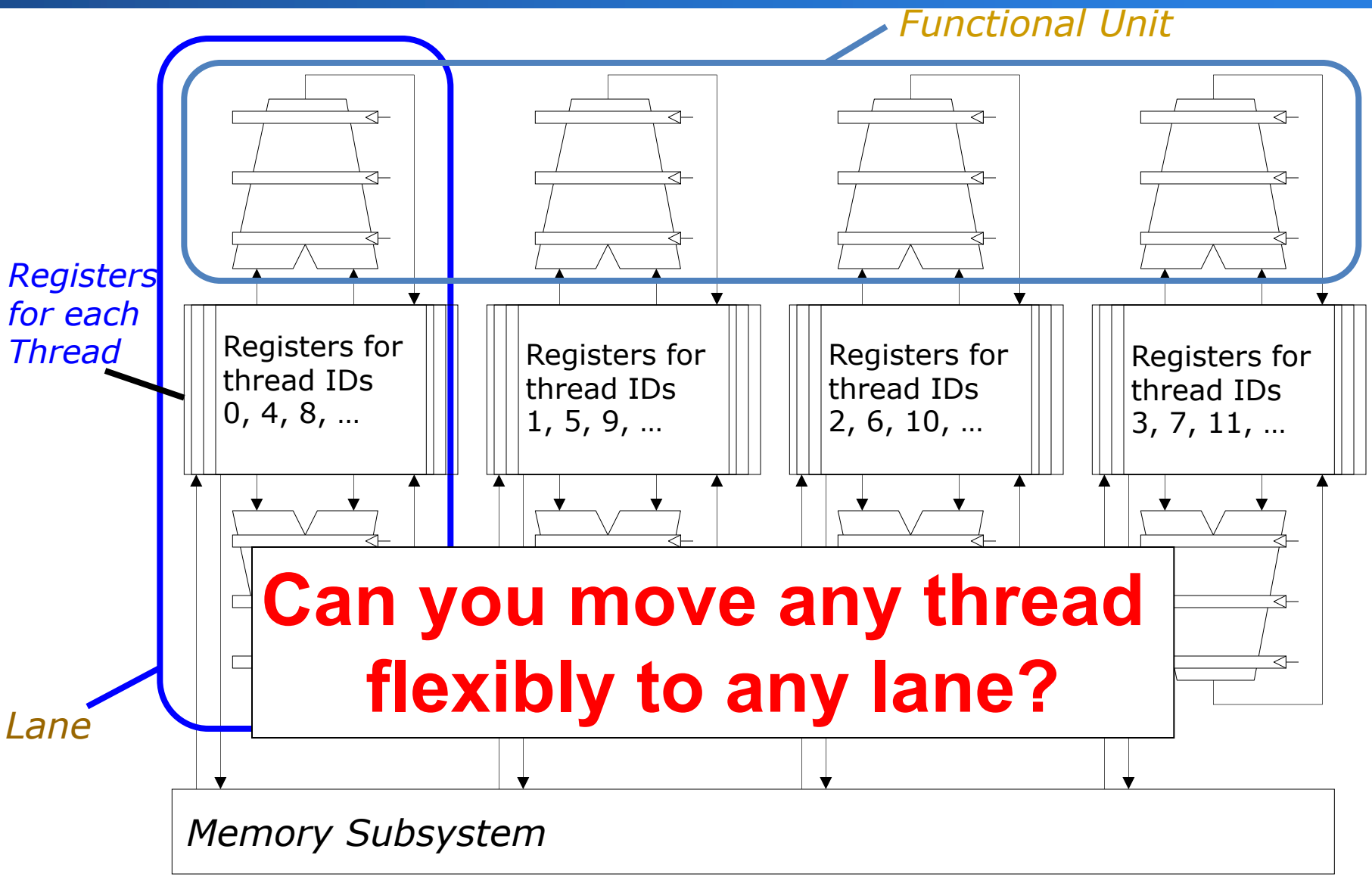


- Fung et al., "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," MICRO 2007.

Dynamic Warp Formation Example



Hardware Constraints Limit Flexibility of Warp Grouping



When You Group Threads Dynamically

- **存储器访问如何处理？**

- **固定模式的存储器访问相对简单，当动态构成warp时，使得访问模式具有随机性，使得问题变得复杂。→ 降低存储器访问的局部性**
  - → 导致存储器带宽利用率的下降

What About Memory Divergence?

- 现代 GPUs 包括高速缓存，减少对存储器的访问
- Ideally: 一个warp中的所有线程的存储器访问都命中 (互相没有冲突）
- Problem: 一个Warp中有些命中，有些失效
- Problem: 一个线程的stall导致整个warp停顿

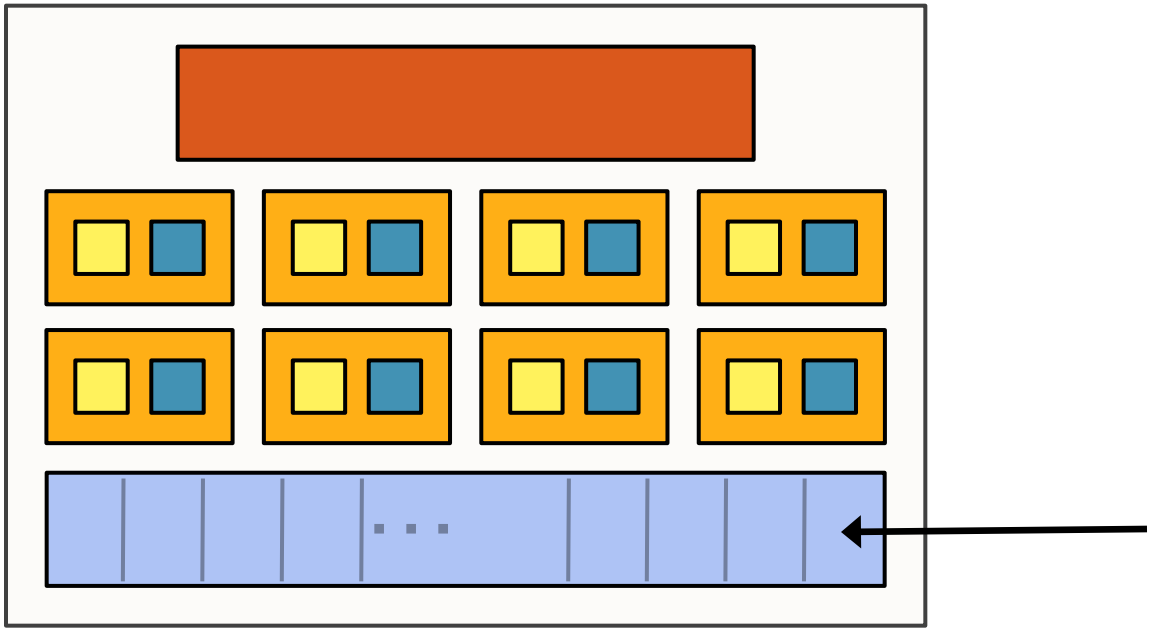
- 需要有相关技术来解决存储器发散访问问题

NVIDIA GeForce GTX 285

- **NVIDIA-speak:**
  - 240 stream processors
  - "SIMT execution"
  -

- **Generic speak:**
  - 30 cores
  - 8 SIMD functional units per core

NVIDIA GeForce GTX 285 "core"



= SIMD functional unit, control shared across 8 units

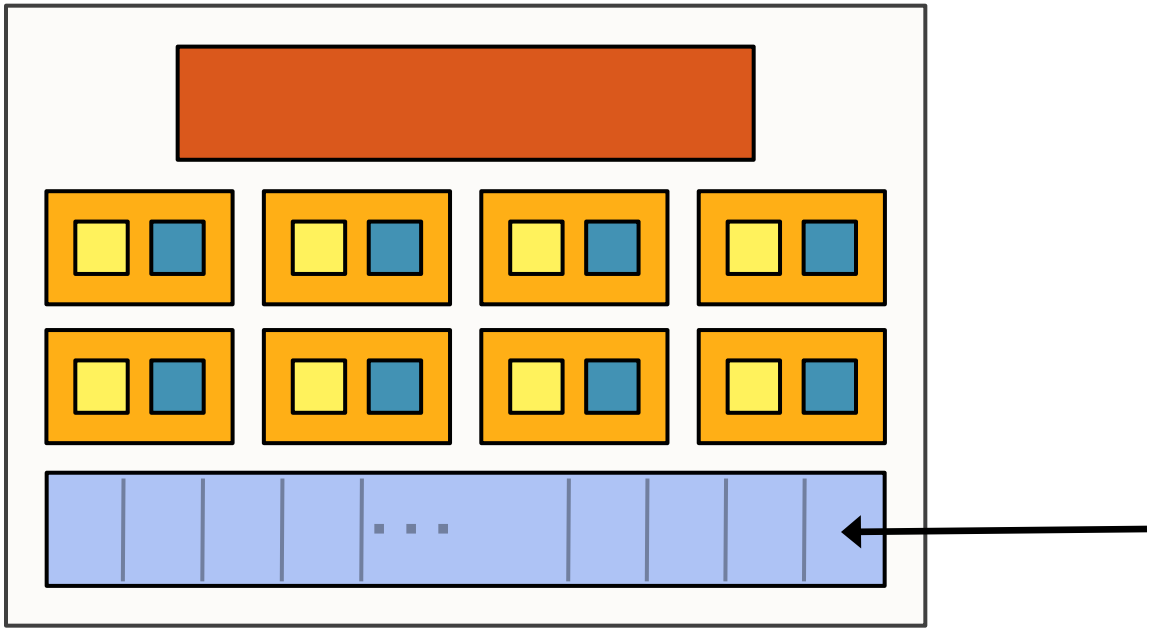
= multiply-add

= multiply

= instruction stream decode

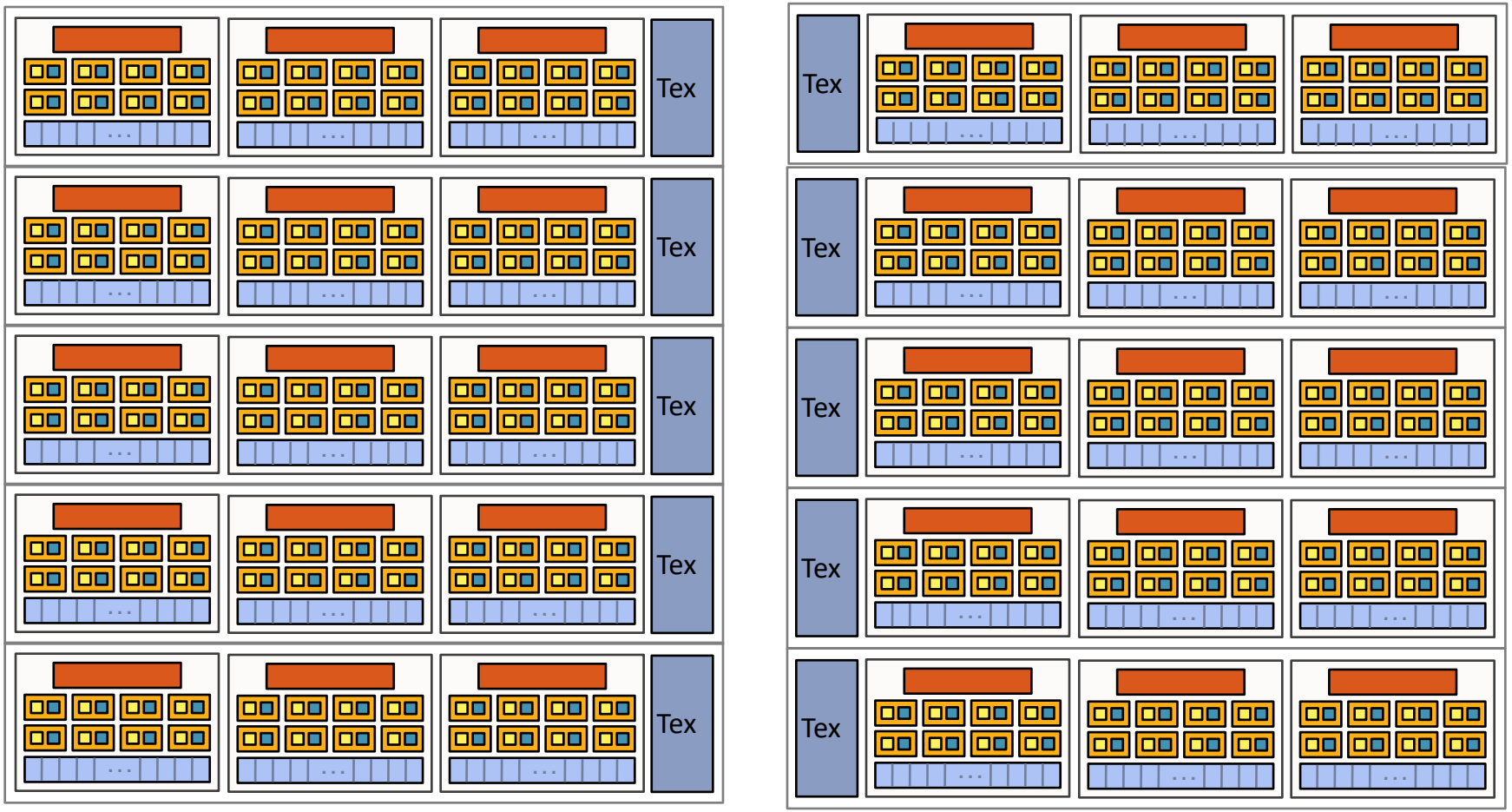
= execution context storage

NVIDIA GeForce GTX 285 “core”



- **Groups of 32 threads share instruction stream (each group is a Warp)**
- **Up to 32 warps are simultaneously interleaved**
- **Up to 1024 thread contexts can be stored**

NVIDIA GeForce GTX 285



- **30 cores on the GTX 285: 30,720 threads**

GP100 GPU

- 56 SMs
- 64 Lanes/SM

Or

- 3584 Stream Processors

Figure 4.20 Block diagram of the multithreaded SIMD Processor of a Pascal GPU. Each of the 64 SIMD Lanes (cores) has a pipelined floating-point unit, a pipelined integer unit, some logic for dispatching instructions and operands to these units, and a queue for holding results. The 64 SIMD Lanes interact with 32 double-precision ALUs (DP units) that perform 64-bit floating-point arithmetic, 16 load-store units (LD/STs), and 16 special function units (SFUs) that calculate functions such as square roots, reciprocals, sines, and cosines.

Summary- 向量处理机 vs. GPU

- **不同层次相近的术语比较**

Type	Descriptive name	Closest old term outside of GPUs	Official CUDA/NVIDIA GPU term	Short explanation
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via local memory
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it only contains SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask
	SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes

Summary-向量处理机 vs. GPU

Type	Descriptive name	Closest old term outside of GPUs	Official CUDA/NVIDIA GPU term	Short explanation
Processing hardware	Multithreaded SIMD Processor	(Multithreaded) Vector Processor	Streaming Multiprocessor	A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors
	Thread Block Scheduler	Scalar Processor	Giga Thread Engine	Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors
	SIMD Thread Scheduler	Thread Scheduler in a Multithreaded CPU	Warp Scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution
	SIMD Lane	Vector Lane	Thread Processor	A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask
Memory hardware	GPU Memory	Main Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU
	Private Memory	Stack or Thread Local Storage (OS)	Local Memory	Portion of DRAM memory private to each SIMD Lane
	Local Memory	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors
	SIMD Lane Registers	Vector Lane Registers	Thread Processor Registers	Registers in a single SIMD Lane allocated across a full Thread Block (body of vectorized loop)

Figure 4.12 Quick guide to GPU terms used in this chapter. We use the first column for hardware terms. Four groups cluster these 11 terms. From top to bottom: program abstractions, machine objects, processing hardware, and memory hardware. Figure 4.21 on page 312 associates vector terms with the closest terms here, and Figure 4.24 on page 317 and Figure 4.25 on page 318 reveal the official CUDA/NVIDIA and AMD terms and definitions along with the terms used by OpenCL.

Acknowledgements

- **These slides contain material developed and copyright by:**
  - John Kubiatowicz (UCB)
  - Krste Asanovic (UCB)
  - John Hennessy (Standford)and David Patterson (UCB)
  - Chenxi Zhang (Tongji)
  - Muhamed Mudawar (KFUPM)
- **UCB material derived from course CS152、CS252、CS61C**
- **KFUPM material derived from course COE501、COE502**
- **CMU Introduction to Computer Architecture**

http://www.ece.cmu.edu/~ece447/