计算机体系结构

周学海
xhzhou@ustc.edu.cn
0551-63606864
中国科学技术大学

第6章 Data-Level Parallelism in Vector, SIMD, and GPU Architectures

6.1 向量处理机模型

- 数据级并行的研究动机
- 数据级并行的种类
- 向量体系结构
  - 向量处理模型
  - 基本特性及结构
  - 性能评估及优化

6.2-1 向量处理机模型优化

6.2-2 面向多媒体应用的SIMD指令集扩展

6.3-1 GPU-I

6.3-2 GPU-II

- GPU简介
- GPU的编程模型
- GPU的存储系统

Review

- **GPU：多线程协处理器**
- **GPU编程模型：SPMD (Single Program Multiple Data)**
  - 使用线程 (SPMD 编程模型)，不是用SIMD指令编程
    - 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)
  - 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）
- **GPU执行模型：SIMT (Single Instruction Multiple Thread)**
  - 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
- **GPU存储器组织**
  - Local Memory， Shared Memory, Global Memory
- **GPU分支处理（发散与汇聚）**

Type	Vector term	Closest CUDA/NVIDIA GPU term	Comment
Program abstractions	Vectorized Loop	Grid	Concepts are similar, with the GPU using the less descriptive term
	Chime	—	Because a vector instruction (PTX instruction) takes just 2 cycles on Pascal to complete, a chime is short in GPUs. Pascal has two execution units that support the most common floating-point instructions that are used alternately, so the effective issue rate is 1 instruction every clock cycle
Machine objects	Vector Instruction	PTX Instruction	A PTX instruction of a SIMD Thread is broadcast to all SIMD Lanes, so it is similar to a vector instruction
	Gather/ Scatter	Global load/store (ld. global/st.global)	All GPU loads and stores are gather and scatter, in that each SIMD Lane sends a unique address. It's up to the GPU Coalescing Unit to get unit-stride performance when addresses from the SIMD Lanes allow it
	Mask Registers	Predicate Registers and Internal Mask Registers	Vector mask registers are explicitly part of the architectural state, while GPU mask registers are internal to the hardware. The GPU conditional hardware adds a new feature beyond predicate registers to manage masks dynamically
Processing and memory hardware	Vector Processor	Multithreaded SIMD Processor	These are similar, but SIMD Processors tend to have many lanes, taking a few clock cycles per lane to complete a vector, while vector architectures have few lanes and take many cycles to complete a vector. They are also multithreaded where vectors usually are not
	Control Processor	Thread Block Scheduler	The closest is the Thread Block Scheduler that assigns Thread Blocks to a multithreaded SIMD Processor. But GPUs have no scalar-vector operations and no unit-stride or strided data transfer instructions, which Control Processors often provide in vector architectures
	Scalar Processor	System Processor	Because of the lack of shared memory and the high latency to communicate over a PCI bus (1000s of clock cycles), the system processor in a GPU rarely takes on the same tasks that a scalar processor does in a vector architecture
	Vector Lane	SIMD Lane	Very similar; both are essentially functional units with registers
	Vector Registers	SIMD Lane Registers	The equivalent of a vector register is the same register in all 16 SIMD Lanes of a multithreaded SIMD Processor running a thread of SIMD instructions. The number of registers per SIMD Thread is flexible, but the maximum is 256 in Pascal, so the maximum number of vector registers is 256
	Main Memory	GPU Memory	Memory for GPU versus system memory in vector case

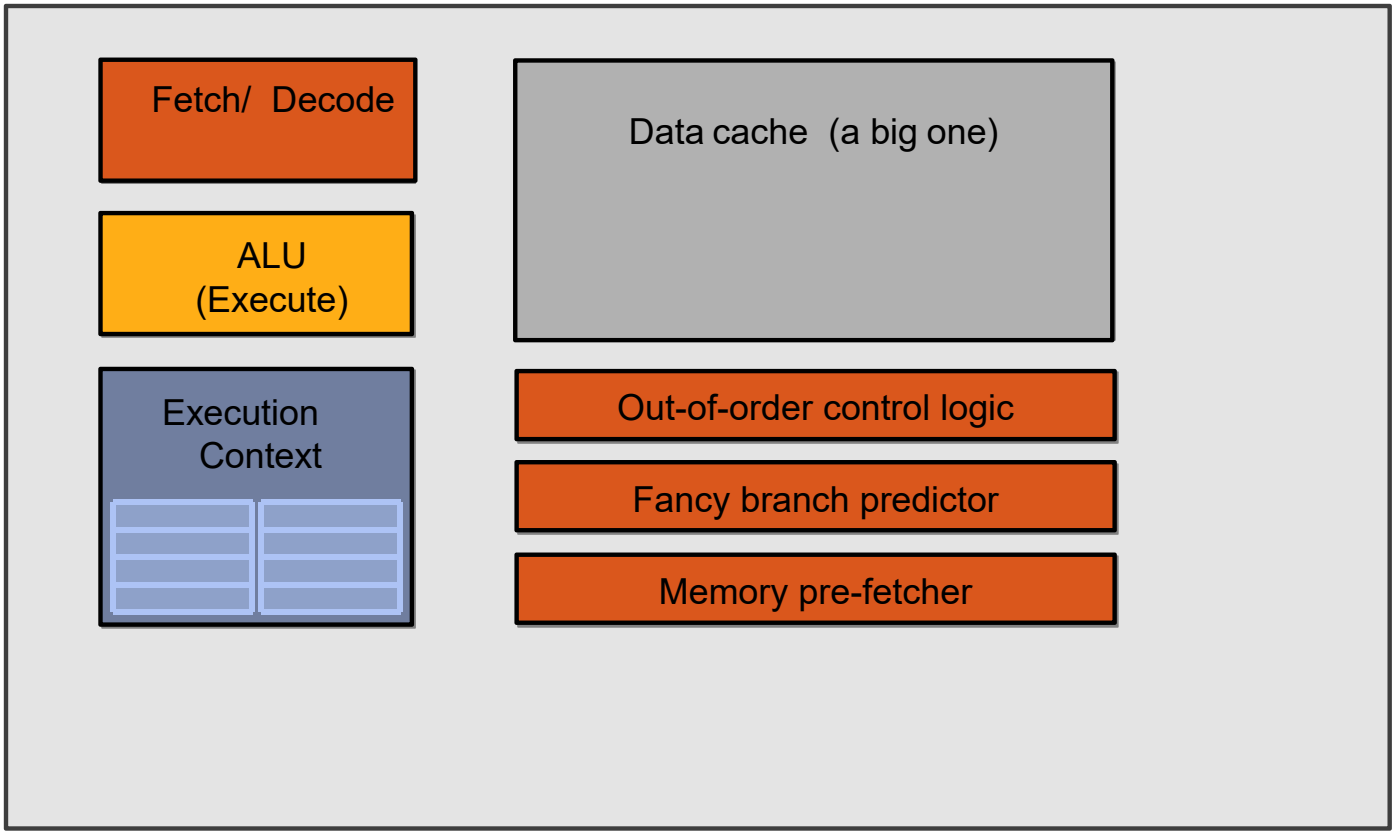
Figure 4.21 GPU equivalent to vector terms.

GPU: throughput processing

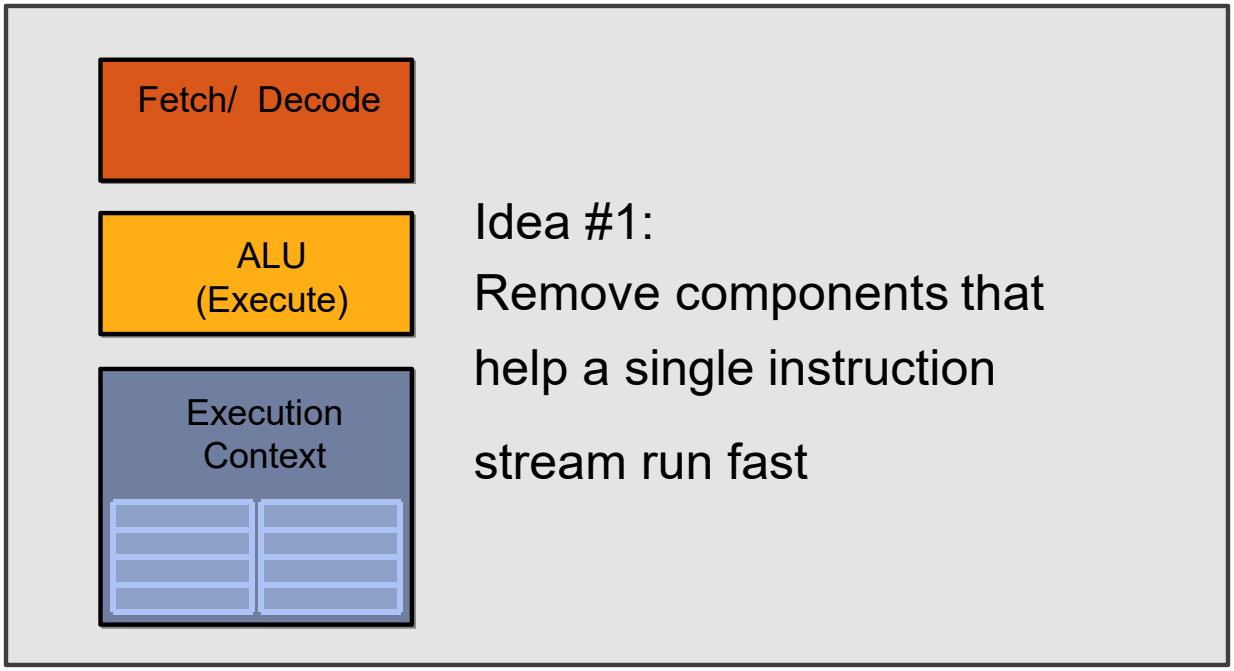
- **现代GPU处理核心运行代码背后的三个关键概念**
  - 使用许多“精简的core”并行运行
  - “装满alu的core” SIMD Processing
  - 通过交叉执行许多组片段来避免延迟(数据访问)

- **了解这些概念的作用：**
  - 了解GPU内核空间(和CPU内核吞吐量)设计
  - 优化着色器/计算内核
  - 建立直观印象:哪些工作负载可能会适合在这些体系结构（的机器上允许）？

“CPU-style” cores



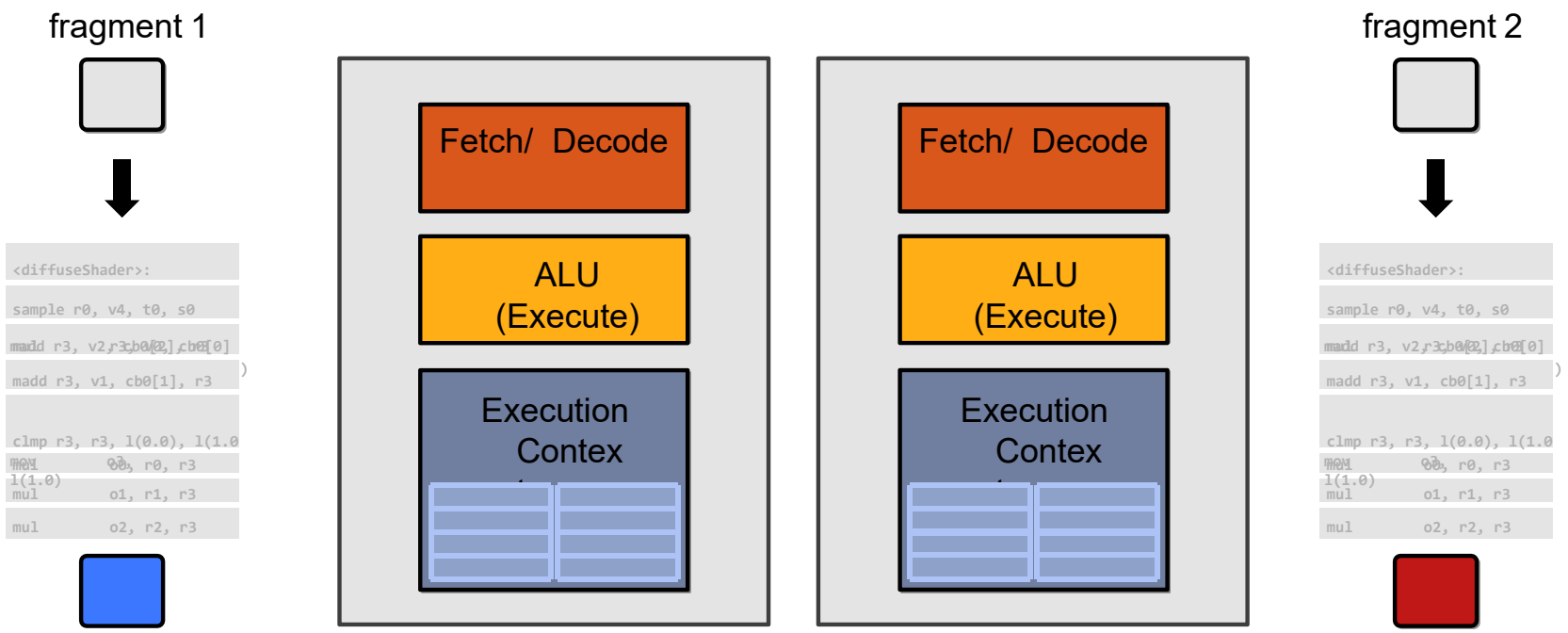
Slimming down



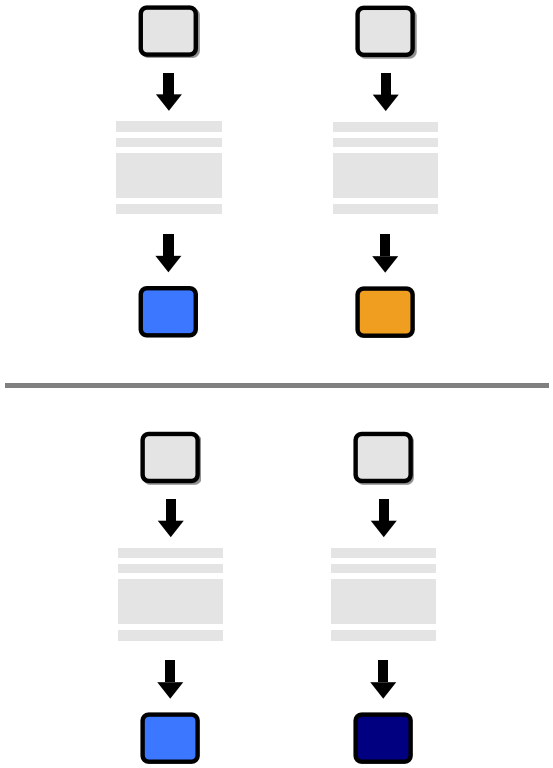
Idea #1:

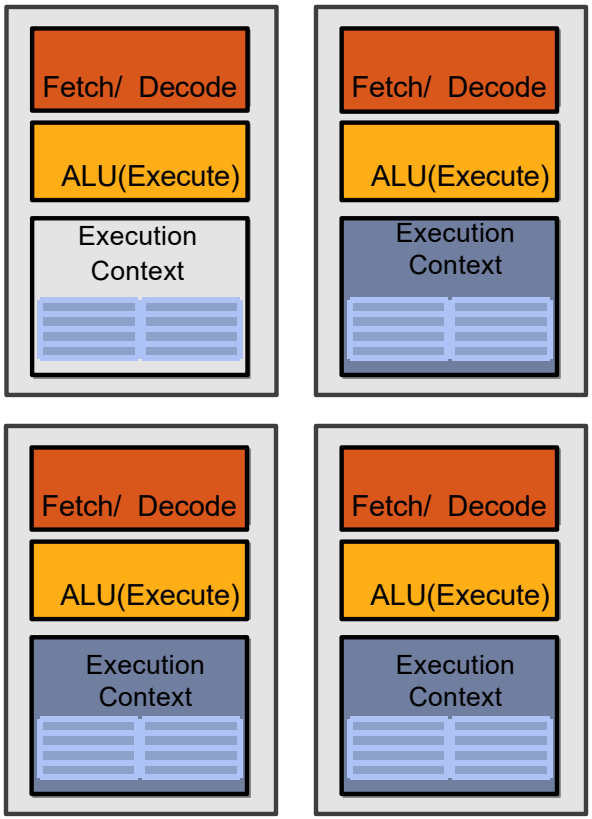
Remove components that help a single instruction stream run fast

Two cores (two fragments in parallel)

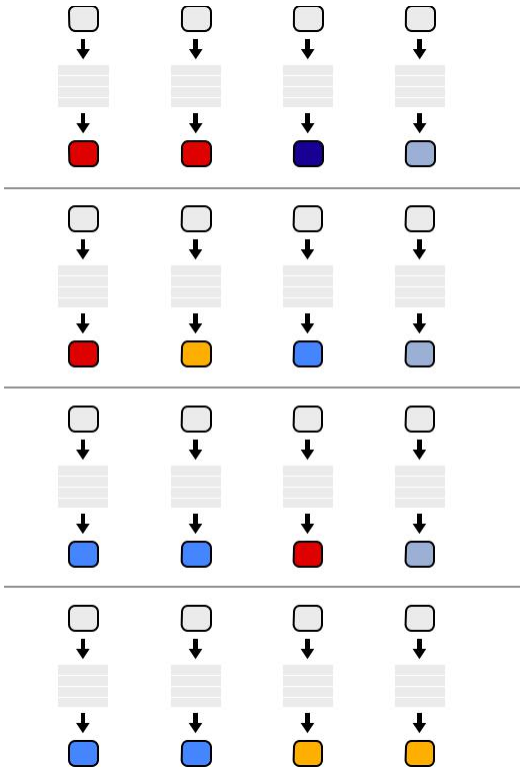


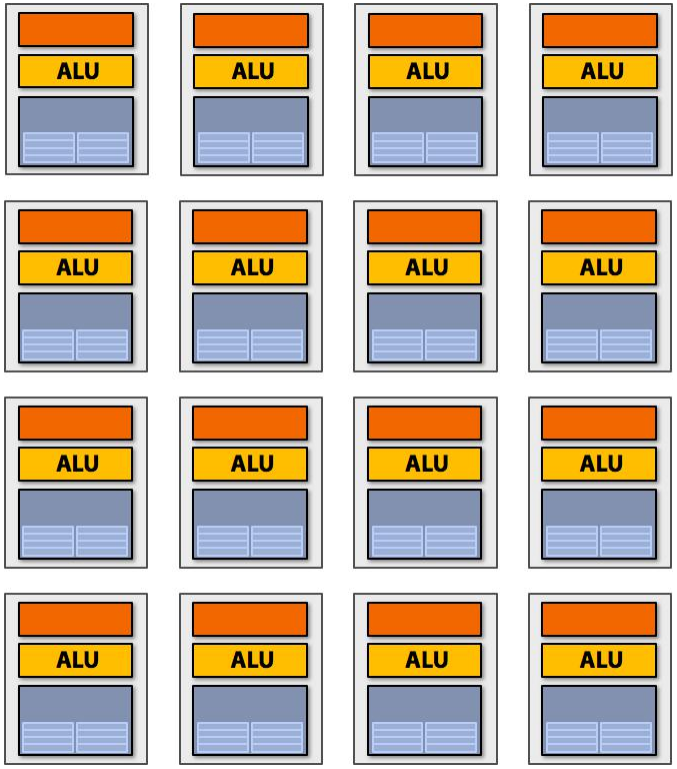
Four cores (four fragments in parallel)





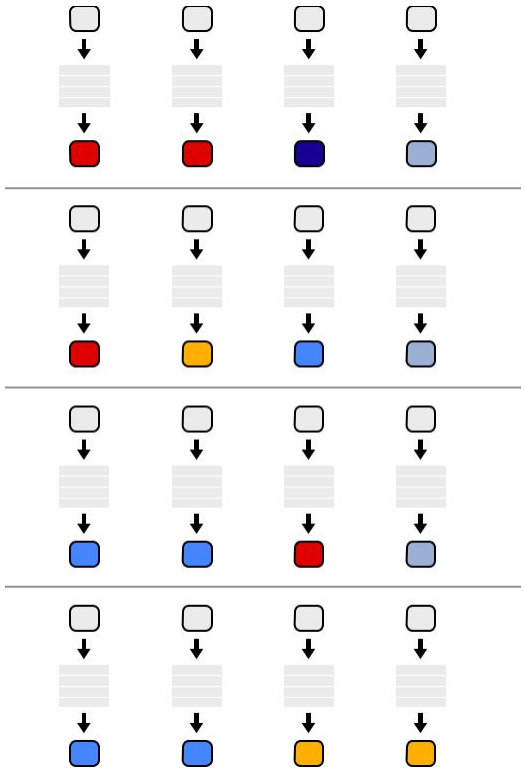
Sixteen cores (sixteen fragments in parallel)





16 cores = 16 simultaneous instruction streams

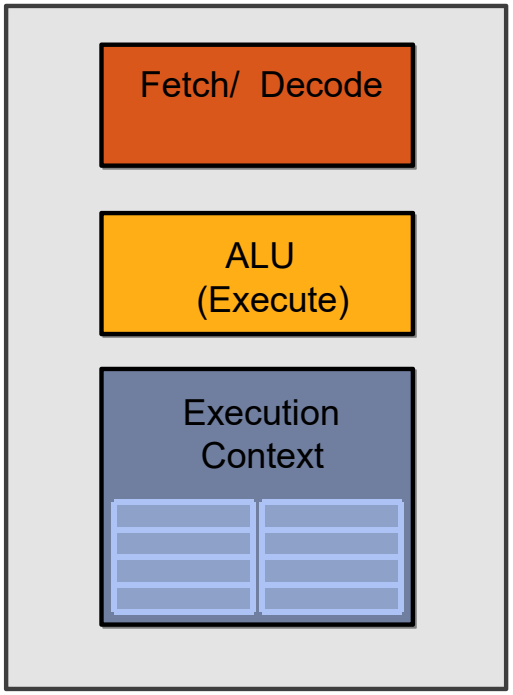
Instruction stream sharing



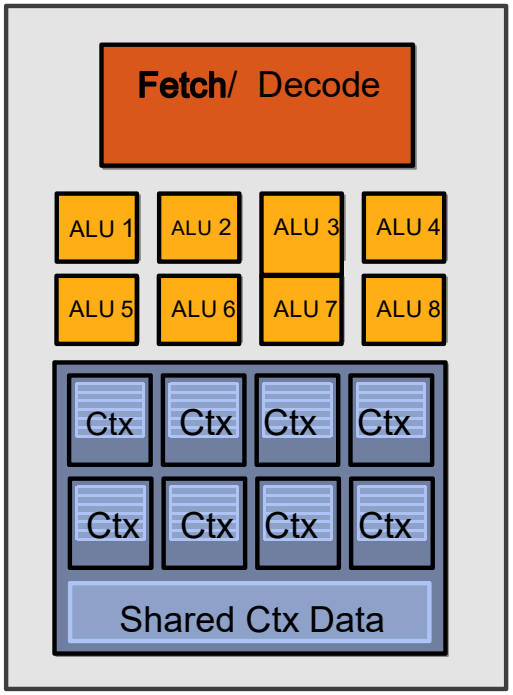
But ... many fragments should be able to share an instruction stream!

```
<diffuseShader>:

sample r0, v4, t0, s0

mul        r3, v0, cb0[0]

madd r3, v1, cb0[1], r3

madd r3, v2, cb0[2], r3

clmp r3, r3, l(0.0), l(1.0)

mul        o0, r0, r3

mul        o1, r1, r3

mul        o2, r2, r3

mov        o3, l(1.0)
```

Recall: simple processing core



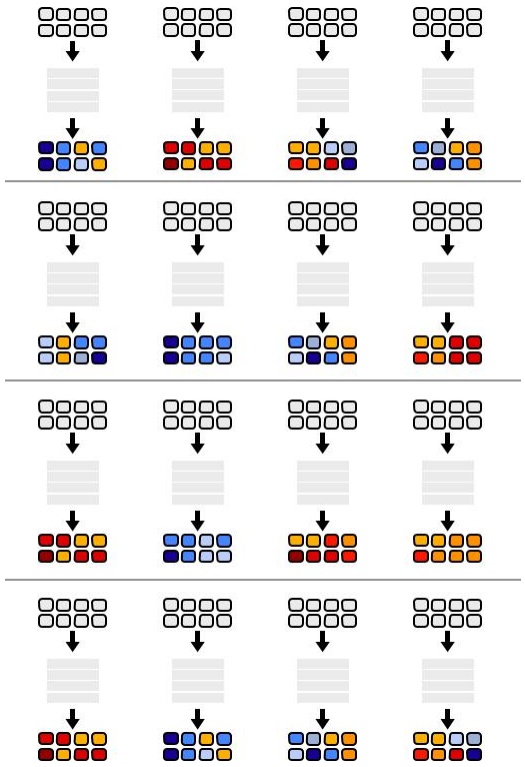
Add ALUs



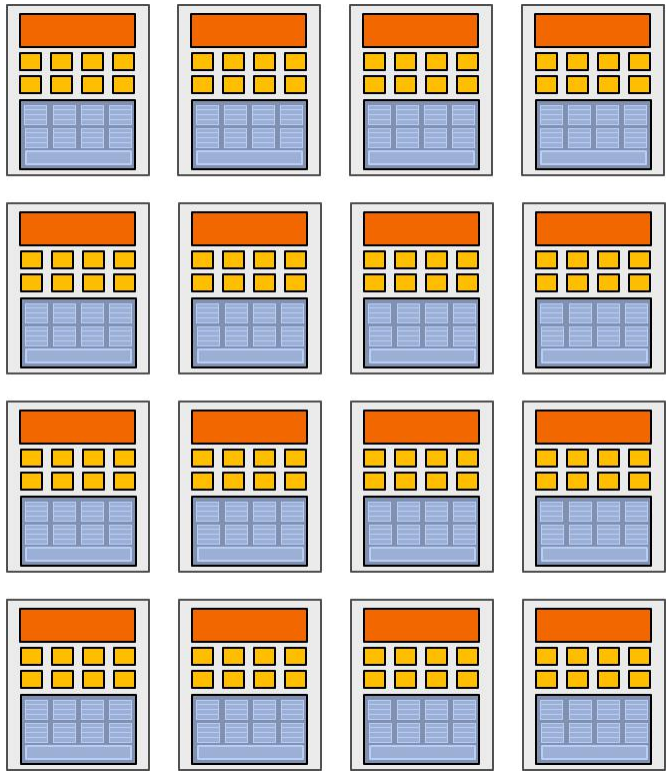
Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

128 fragments in parallel



16 cores = 128 ALUs

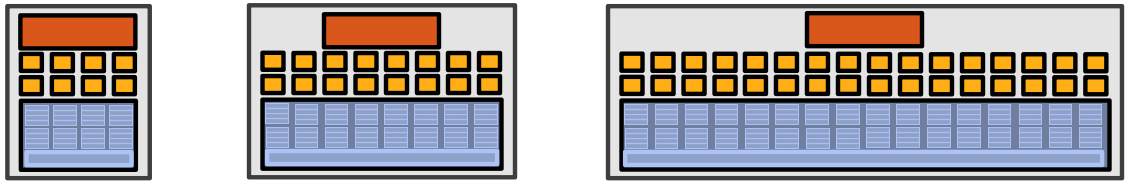


, 16 simultaneous instruction streams

Clarification

SIMD processing does not imply SIMD instructions

- Option 1: explicit vector instructions
  - x86 SSE, AVX, Intel Larrabee
- Option 2: scalar instructions, implicit HW vectorization
  - HW determines instruction stream sharing across ALUs (amount of sharing hidden from software)
  - NVIDIA GeForce (“SIMT” warps), ATI Radeon architectures (“wavefronts”)



In practice: 16 to 64 fragments share an instruction stream.

Stalls occur when a core cannot run the next instruction because of a dependency on a previous operation.

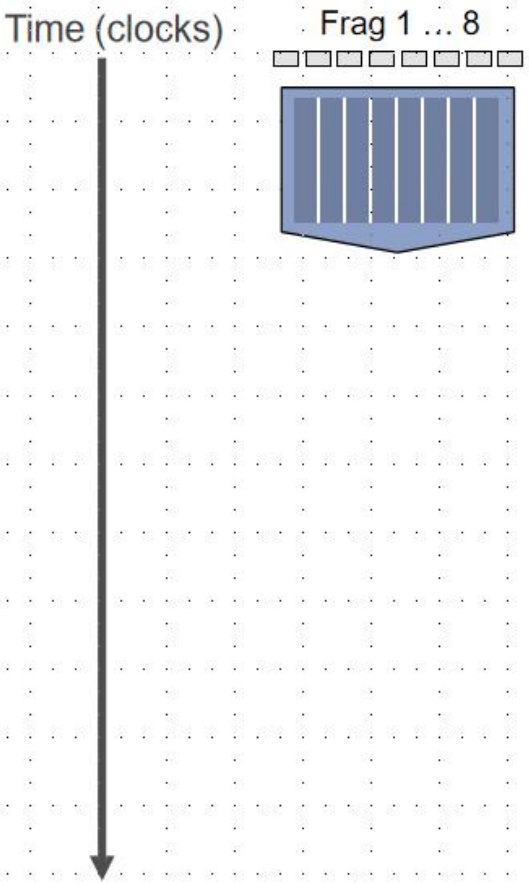
Texture access latency = 100's to 1000's of cycles

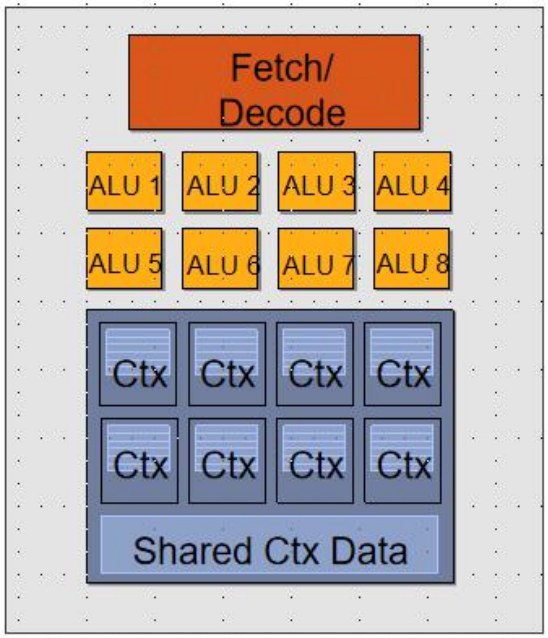
We've removed the fancy caches and logic that helps avoid stalls.

Idea #3:

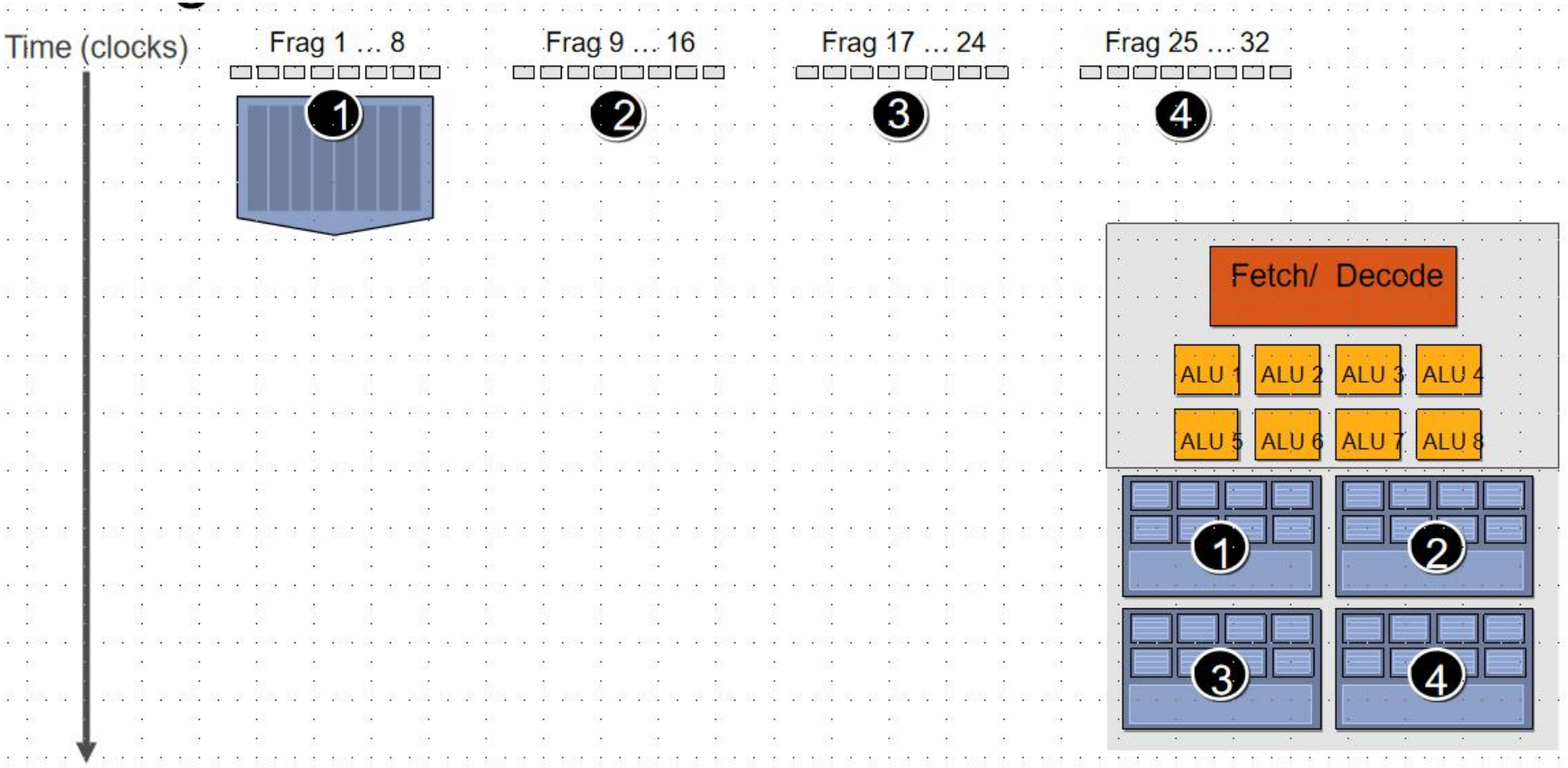
Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations
(在单个核上交错处理多组片段，以避免高延迟操作造成的停顿.)

Hiding shader stalls

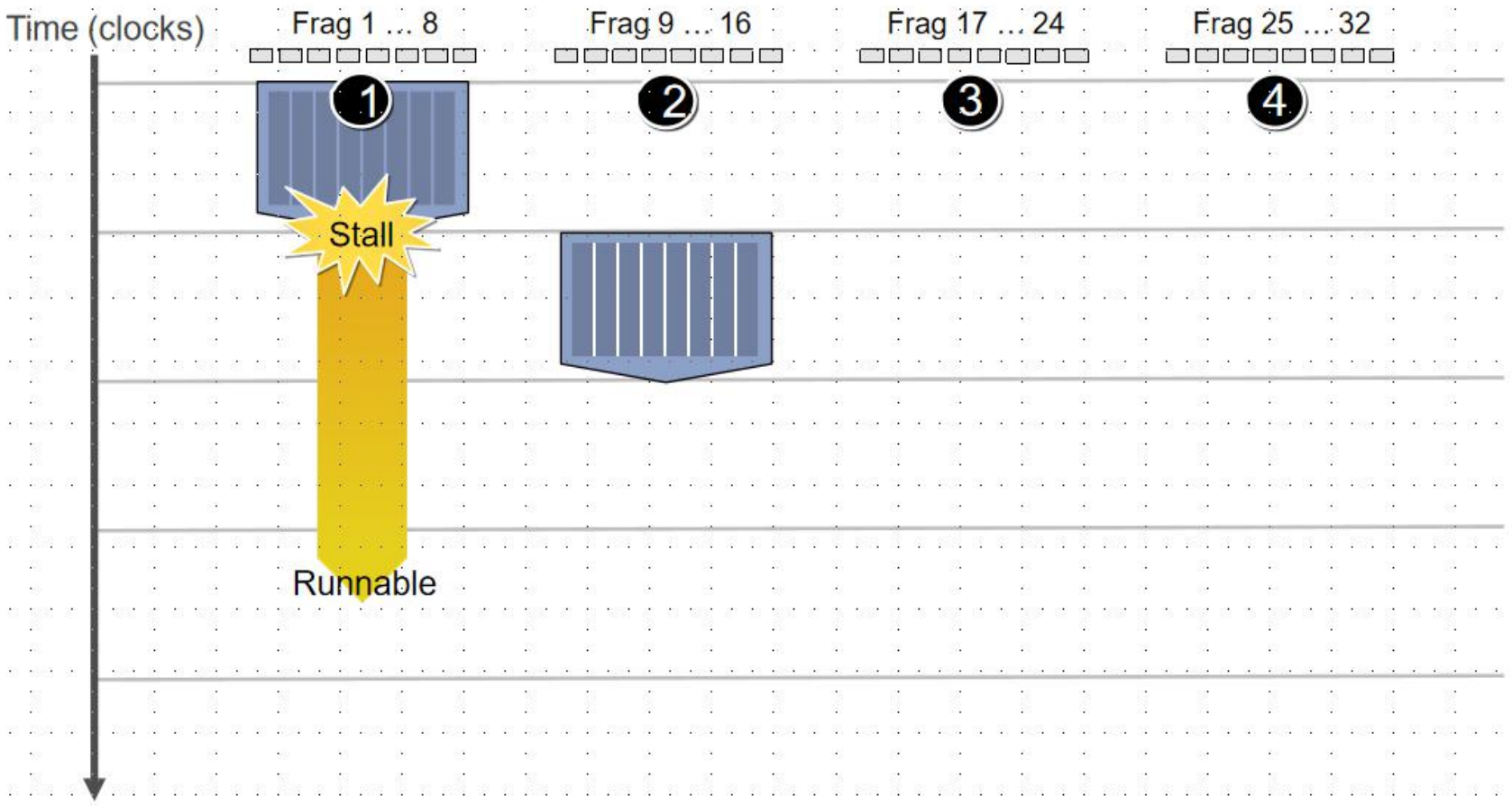




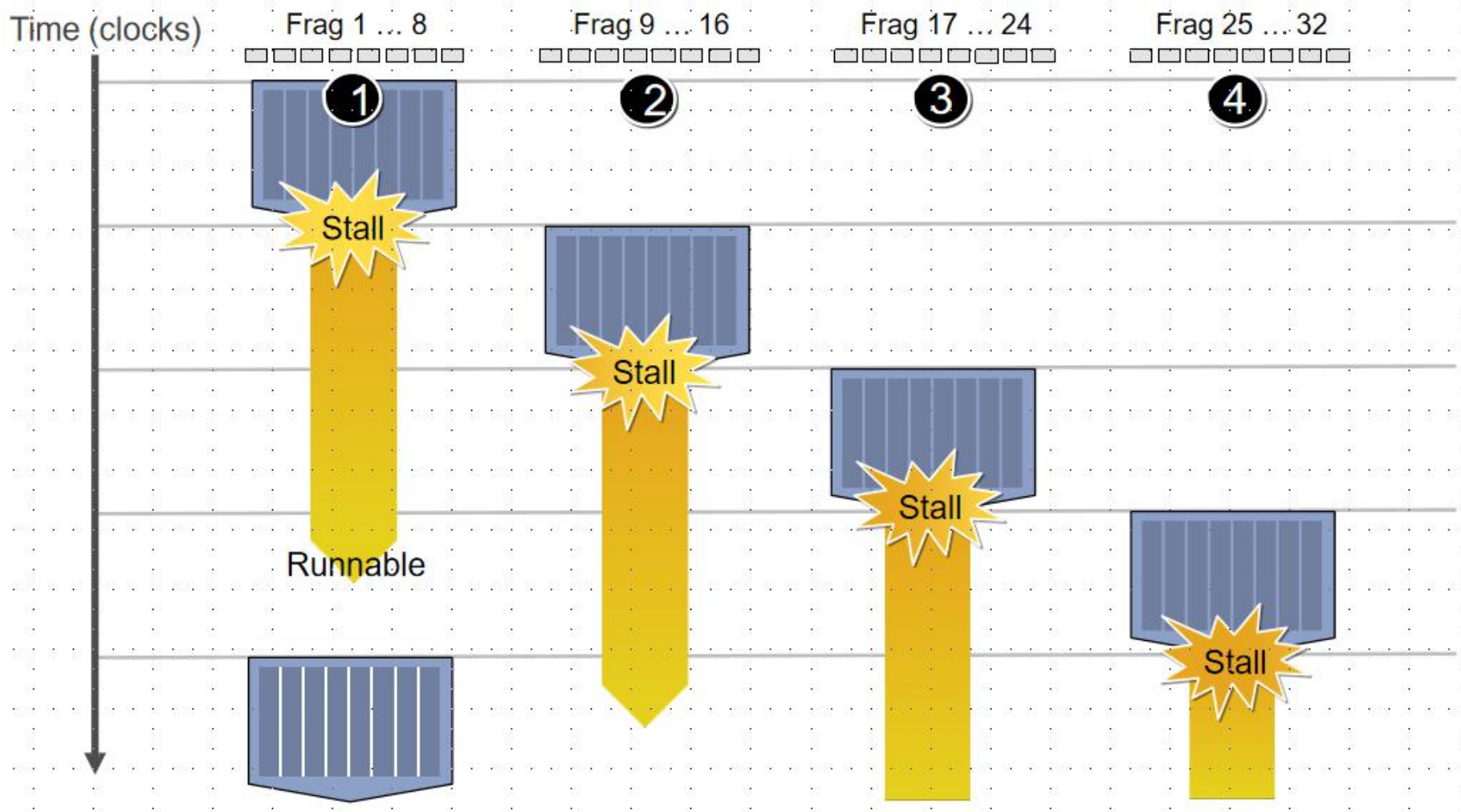
Hiding shader stalls



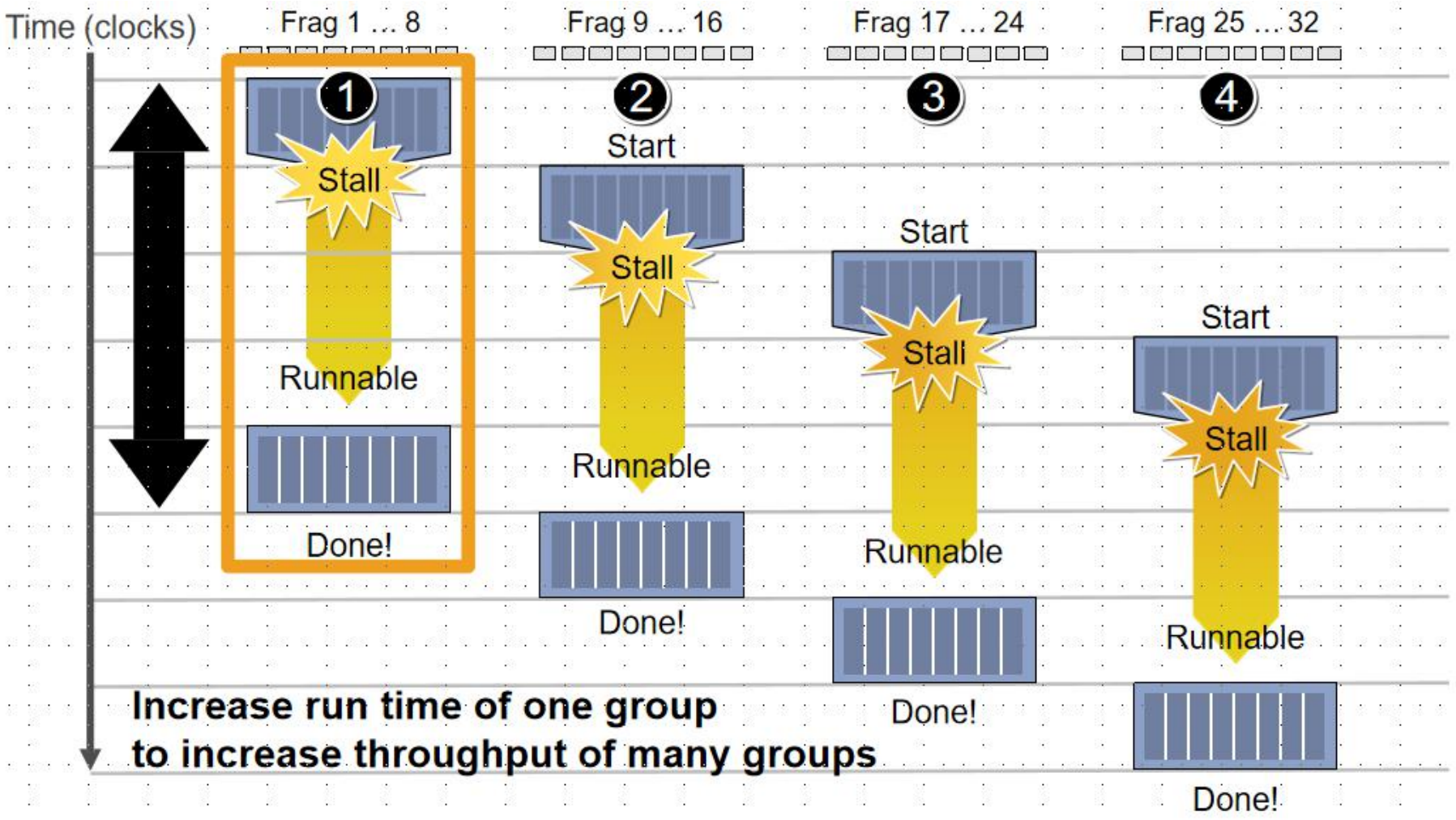
Hiding shader stalls



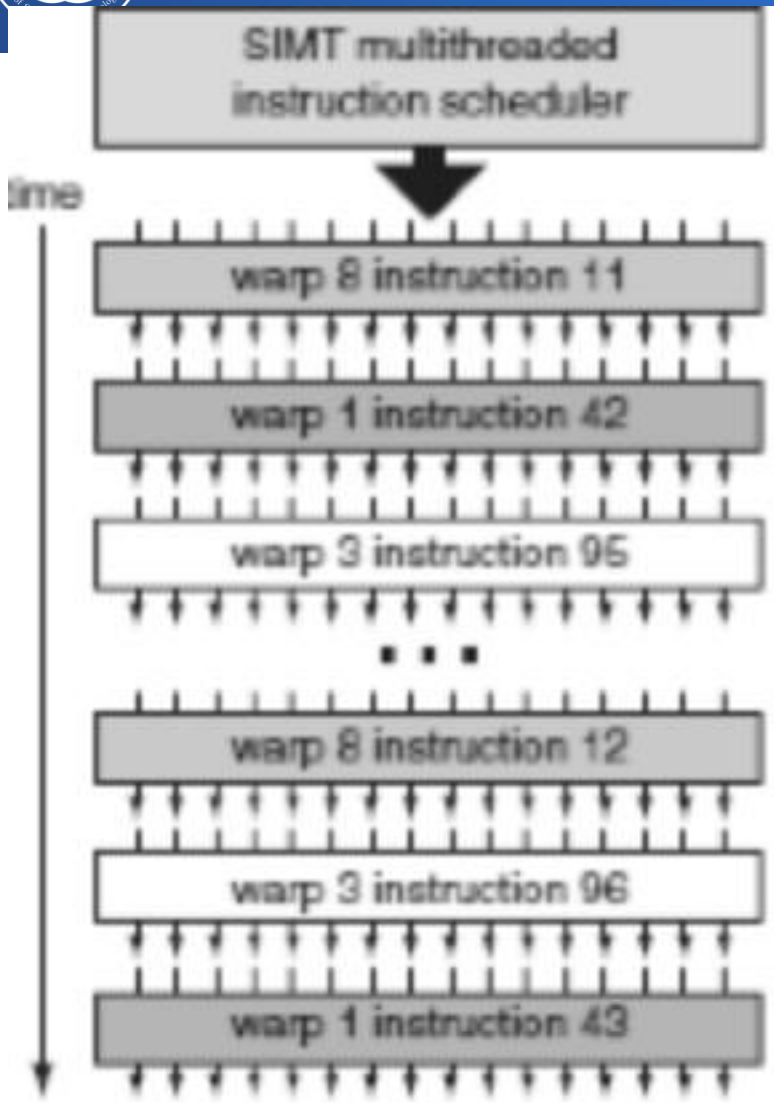
Hiding shader stalls



Throughput!


Time (clocks)
Frag 1 … 8
Frag 9 … 16
Frag 17 … 24
Frag 25 … 32
1
2
3
4
Stall
Start
Stall
Start
Stall
Start
Stall
Runnable
Runnable
Runnable
Runnable
Done!
Done!
Done!
Done!
Increase run time of one group
to increase throughput of many groups

Warps are multithreaded on core



- **一个 warp 由 32个 μthreads 构成**
- **多个warp线程在单个核上交叉运行，以隐藏存储器访问和功能部件的延迟**
- **单个线程块包含多个warp，这些warp都映射到同一个核上**
- **多个线程块也可以在同一个核上运行**

[Nvidia, 2010]

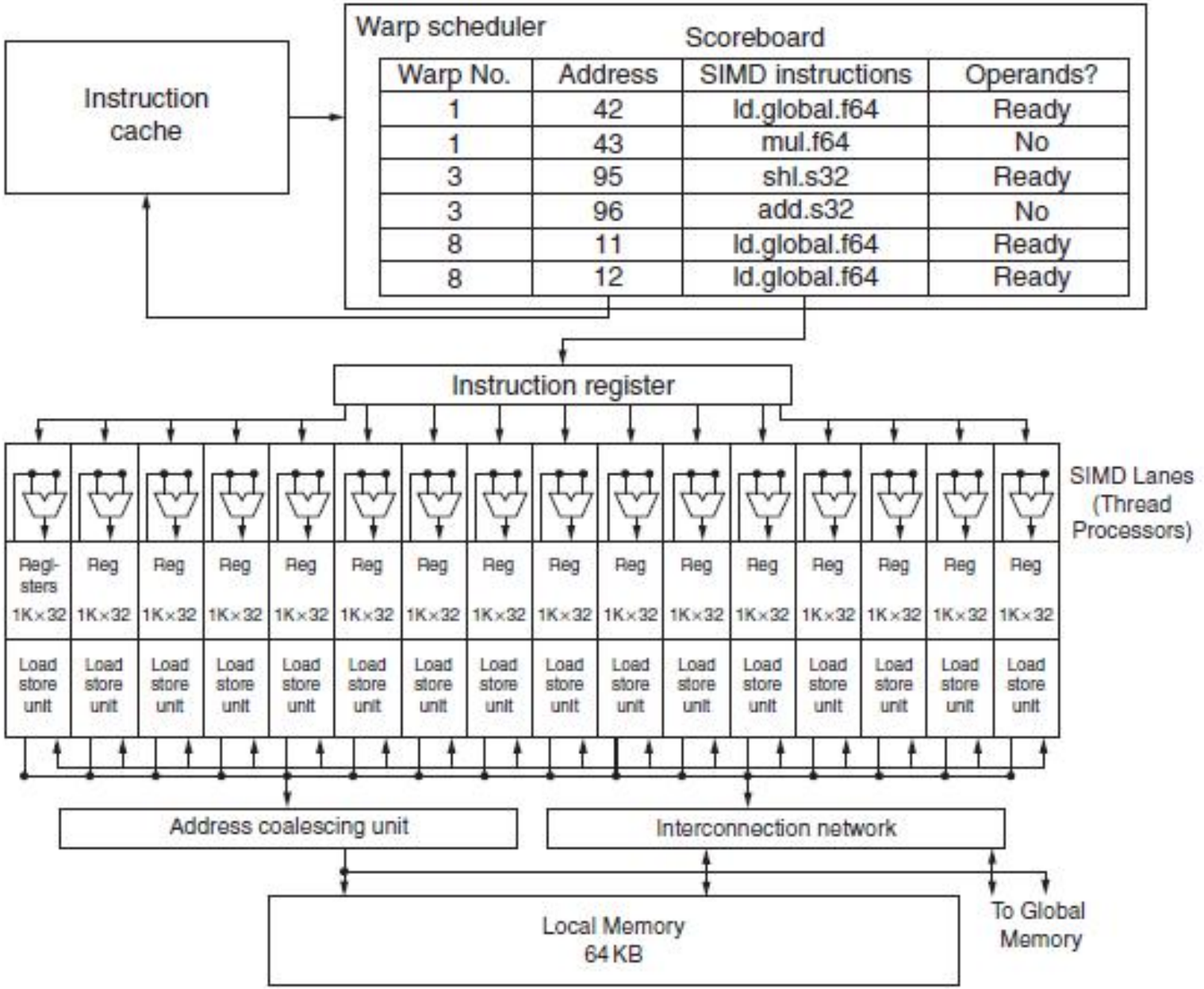
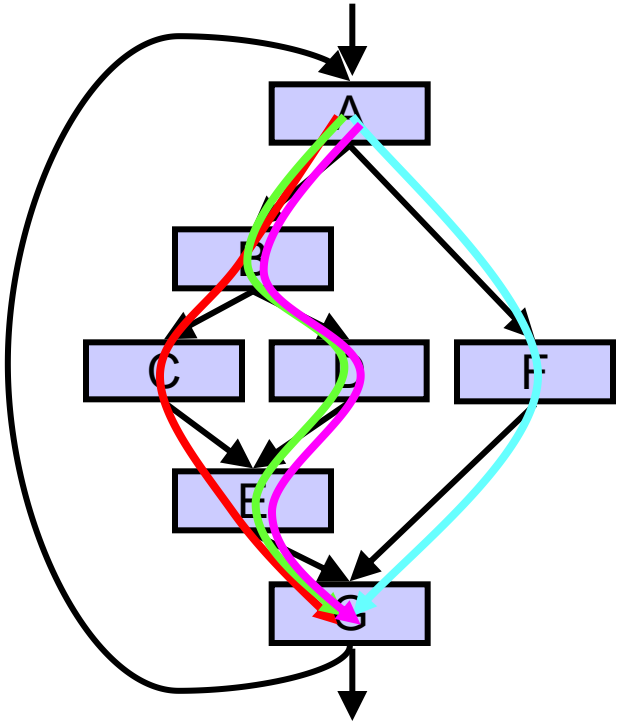


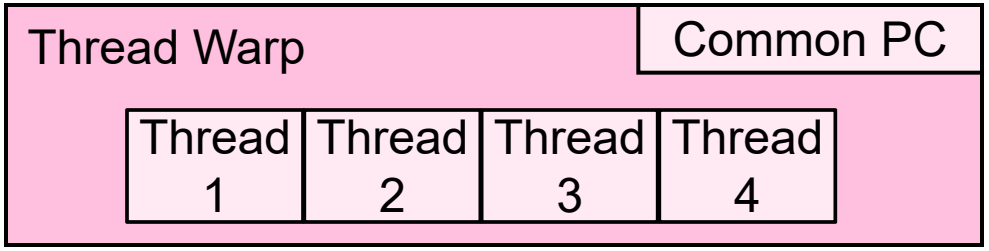
Figure 4.14 Simplified block diagram of a Multithreaded SIMD Processor. It has 16 SIMD lanes. The SIMD Thread Scheduler has, say, 48 independent threads of SIMD instructions that it schedules with a table of 48 PCs.

6.3-2 GPU II - 分支处理（发散与汇聚）

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



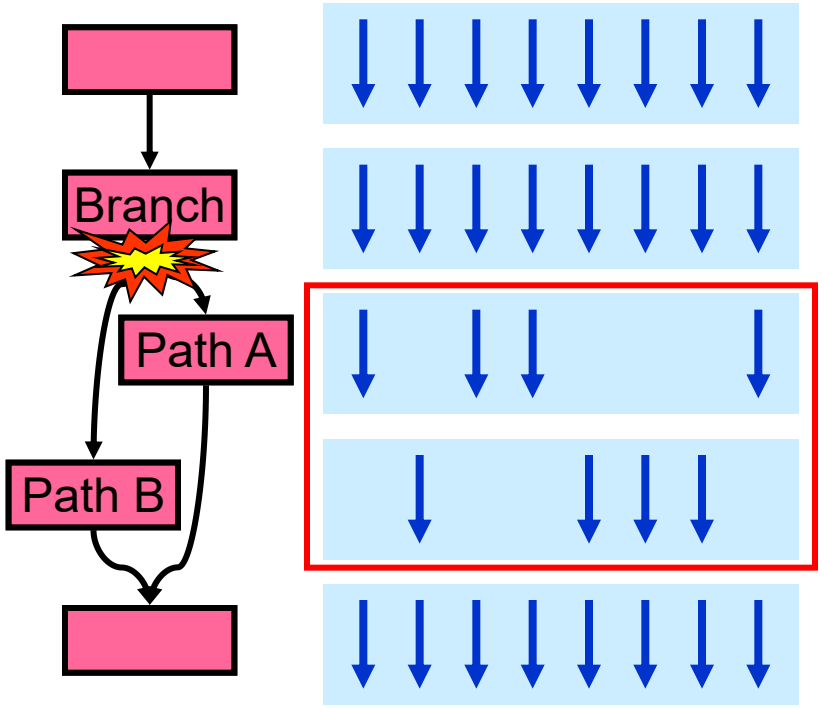


Slide credit: Tor Aamodt

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%

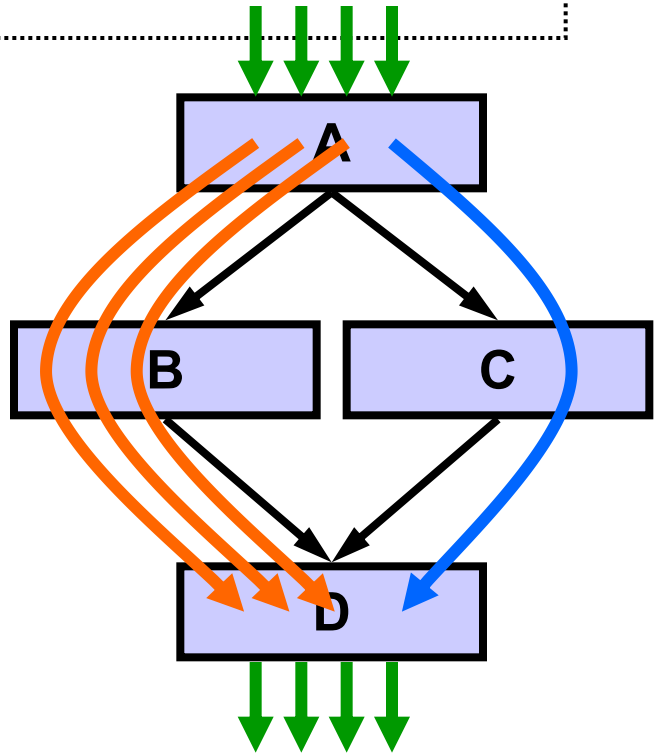
Example

```
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
```

Branch Divergence Handling

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

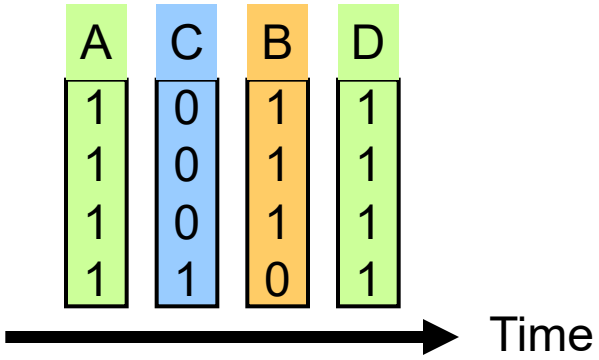


One per warp

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.

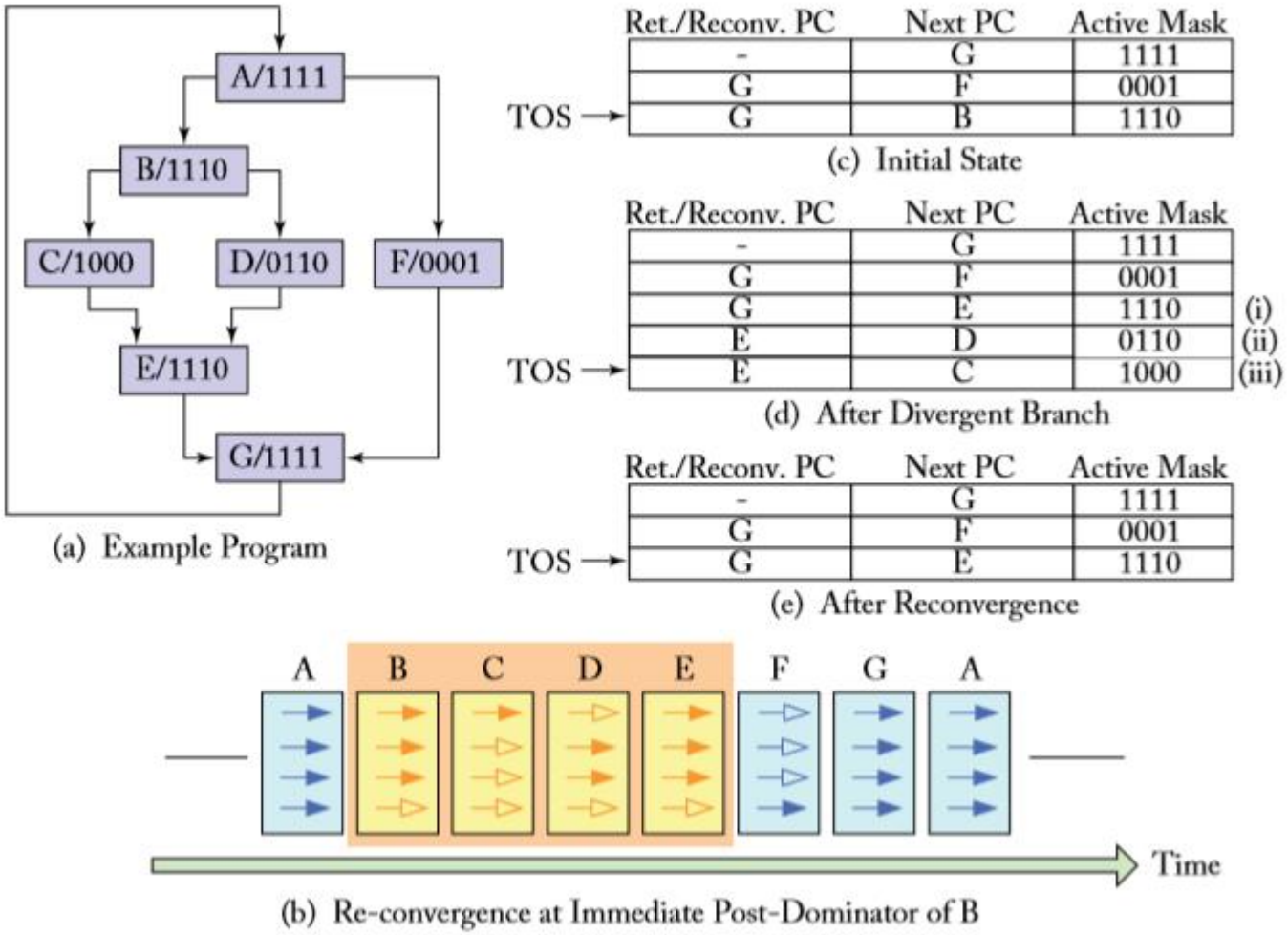


Figure 3.4: Example of SIMT stack operation (based on Figure 5 from Fung et al. [2007]).

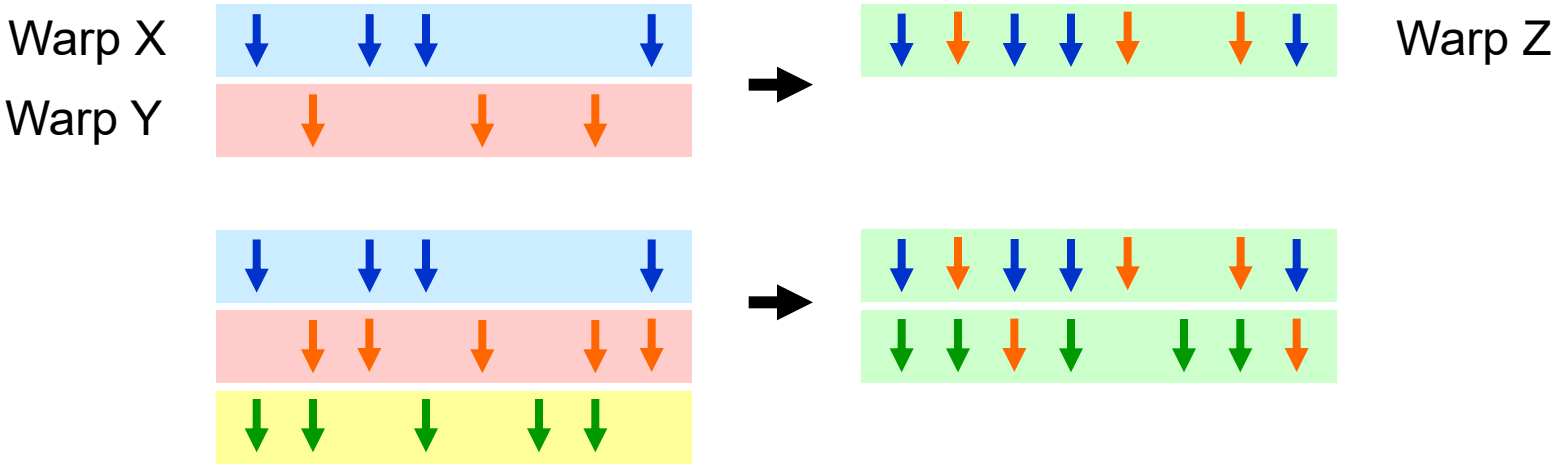
Remember: Each Thread Is Independent

- **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)

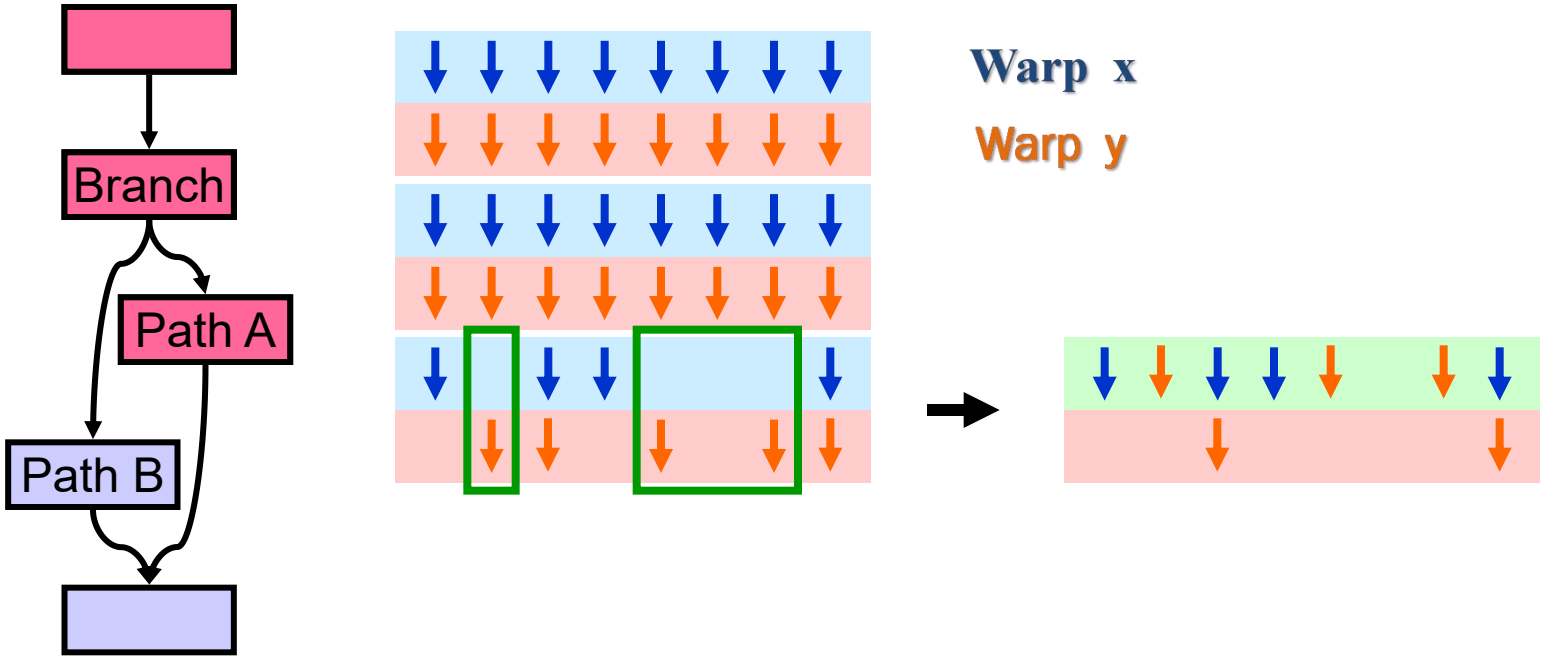
Dynamic Warp Formation/Merging

- **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



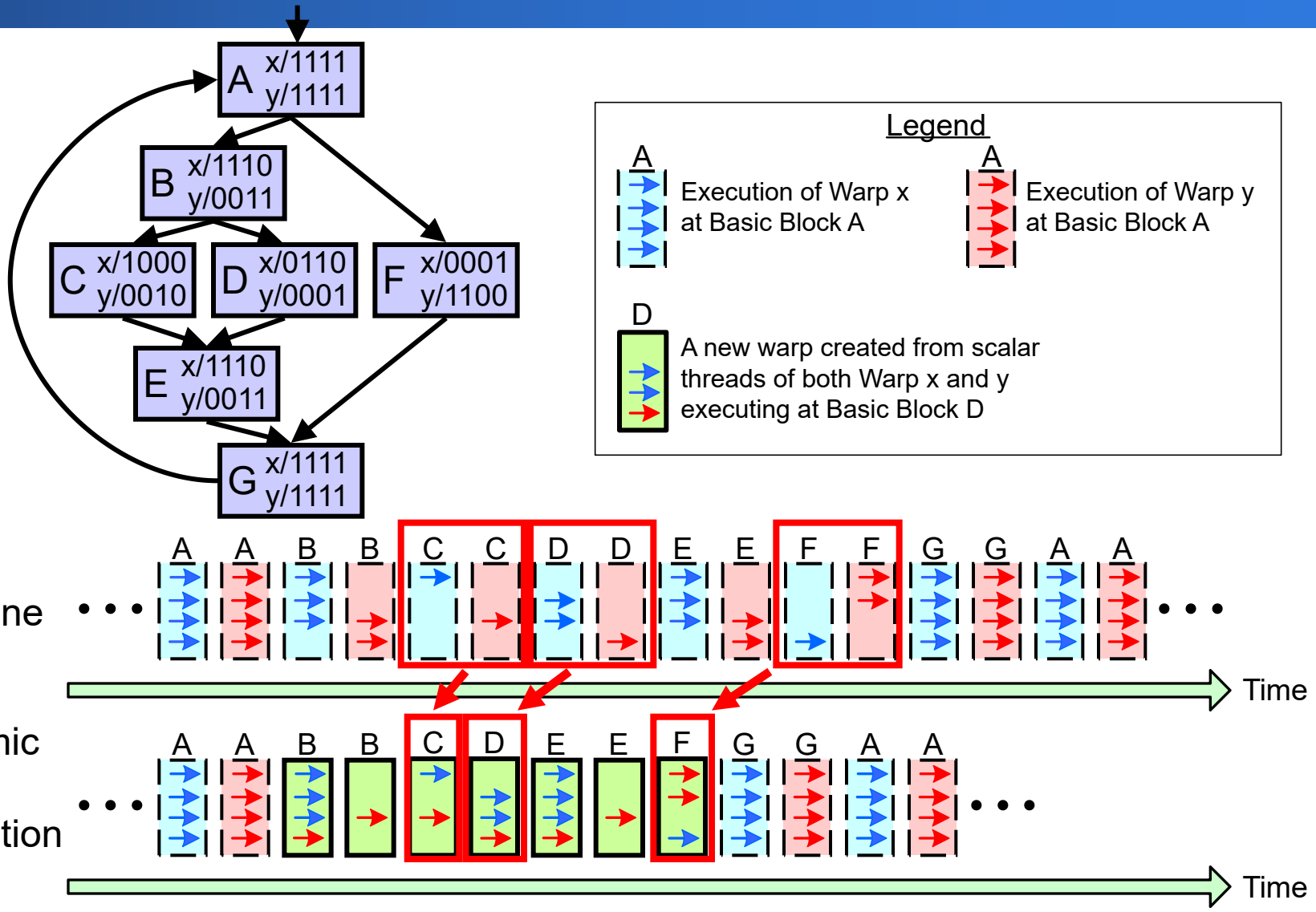
Dynamic Warp Formation/Merging

- Idea:
  - 分支发散之后，动态合并执行相同指令的线程

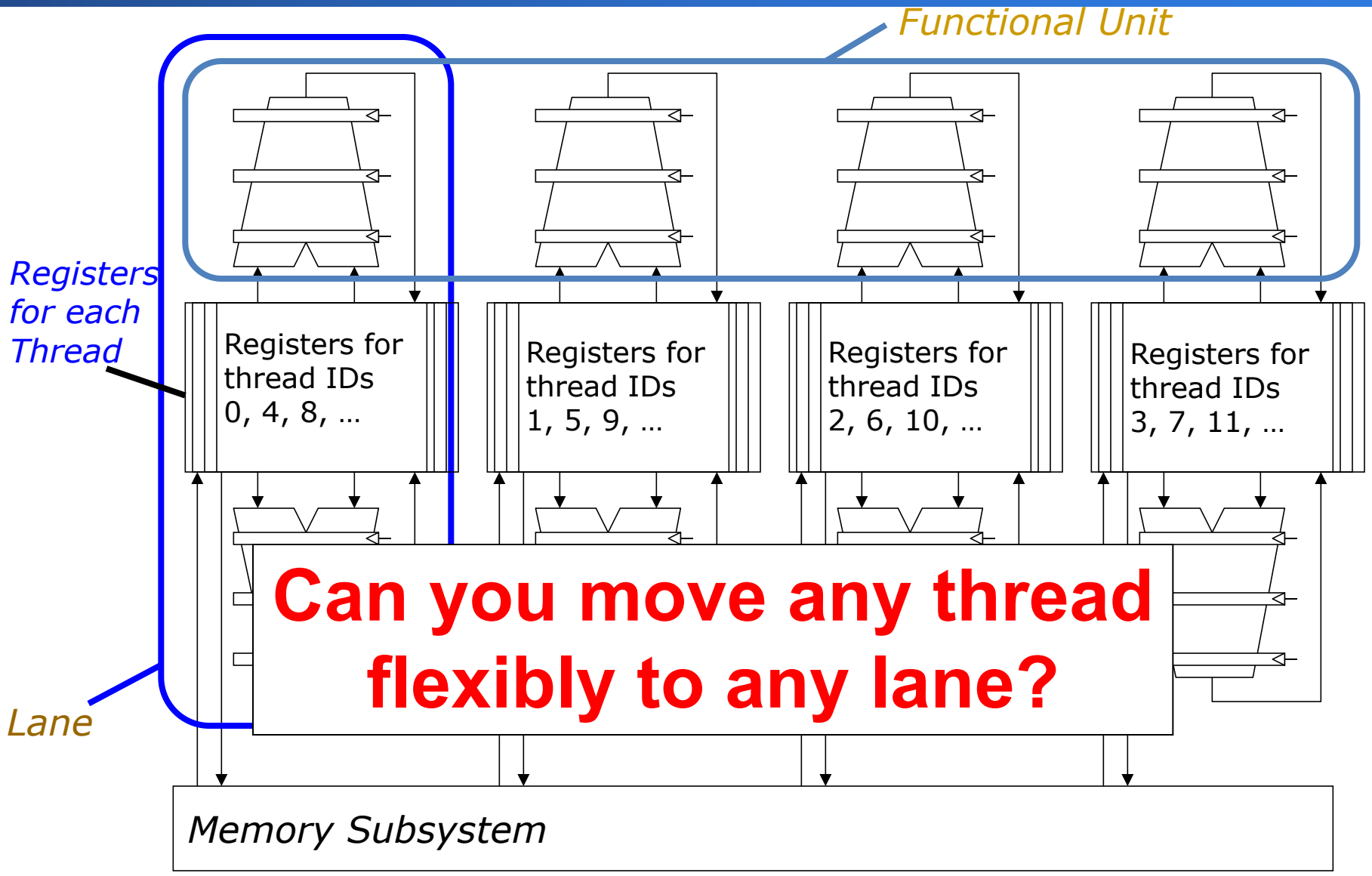


- Fung et al., "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," MICRO 2007.

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

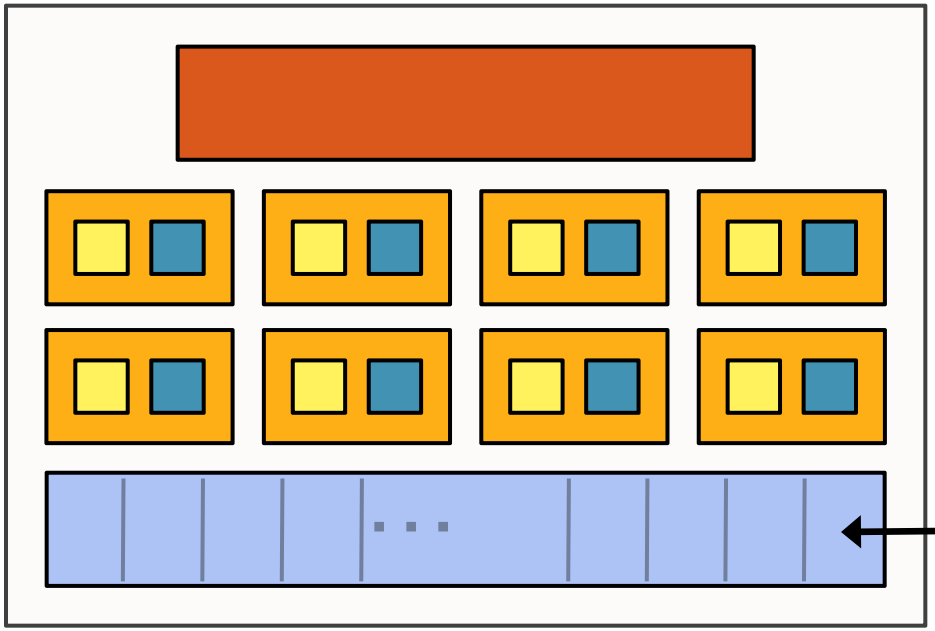
- 需要有相关技术来解决存储器发散访问问题

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"



64 KB of storage for thread contexts (registers)

= SIMD functional unit, control shared across 8 units

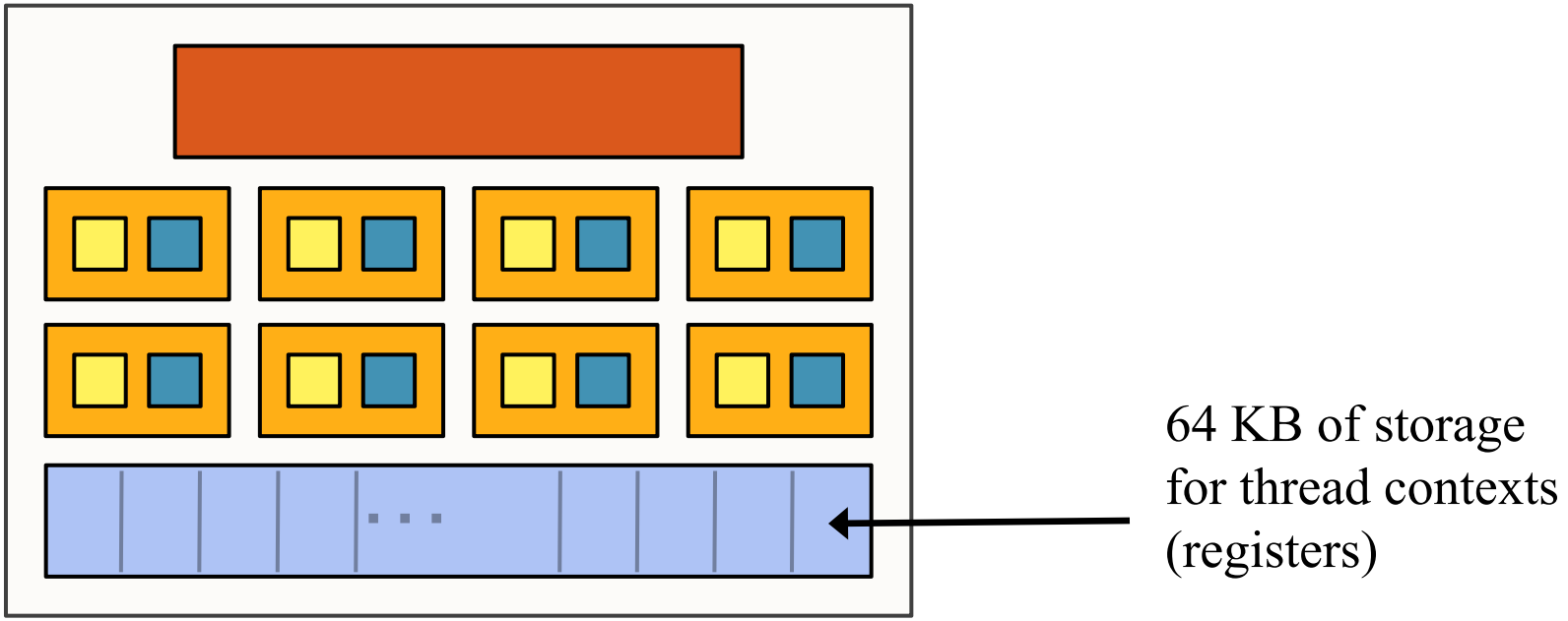
= multiply-add

= multiply

= instruction stream decode

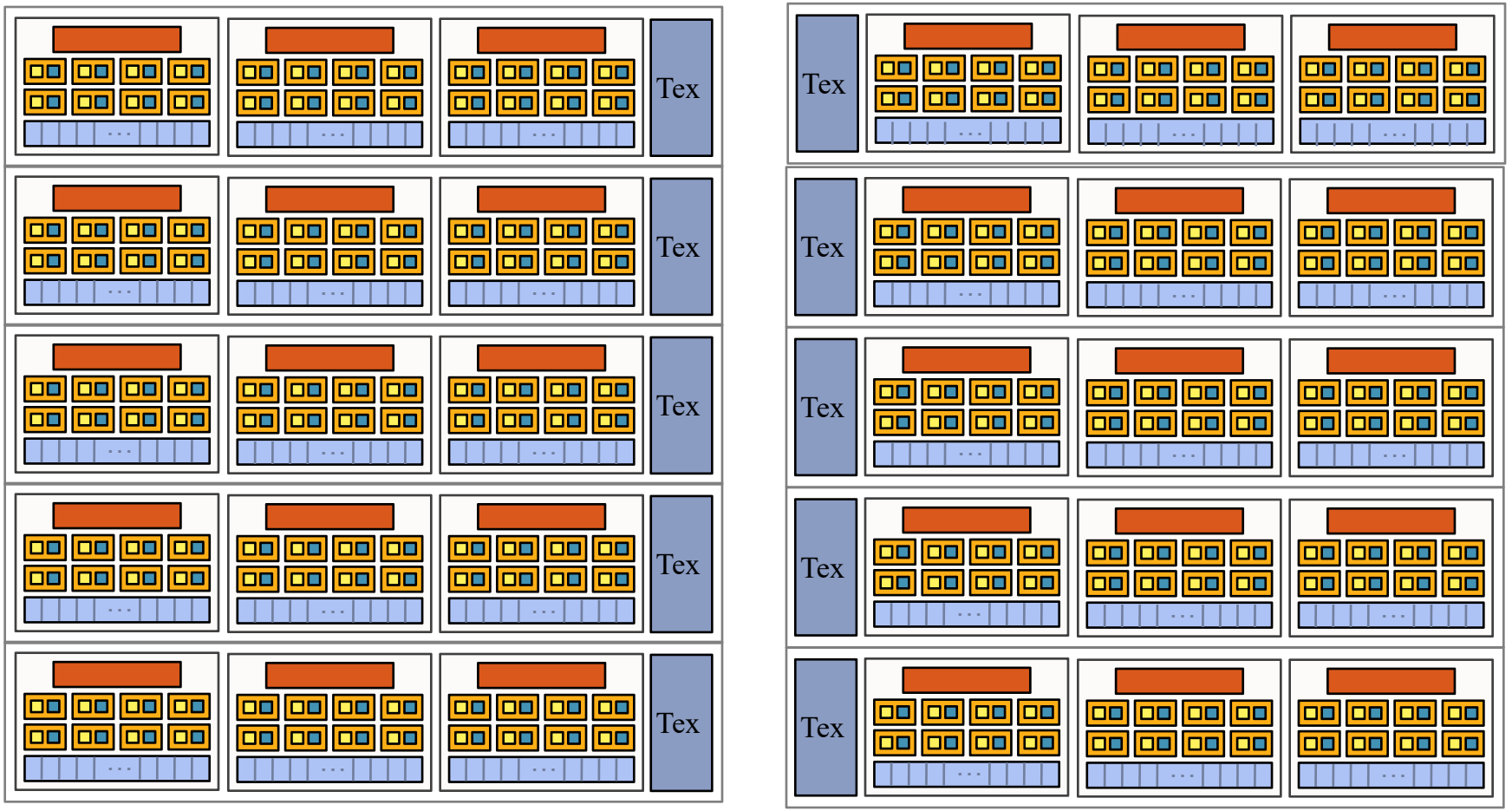
= execution context storage

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

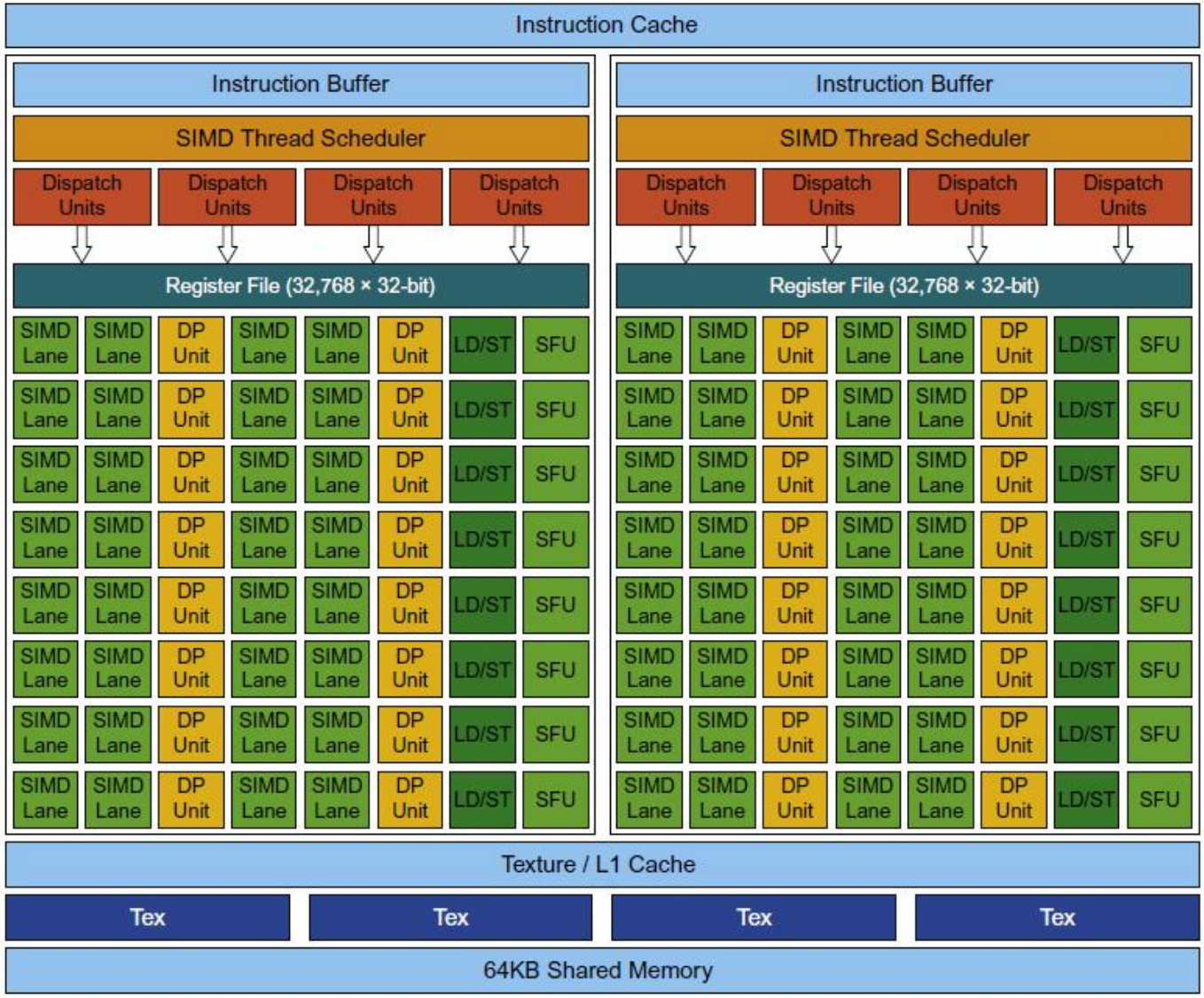


Figure 4.20 Block diagram of the multithreaded SIMD Processor of a Pascal GPU. Each of the 64 SIMD Lanes (cores) has a pipelined floating-point unit, a pipelined integer unit, some logic for dispatching instructions and operands to these units, and a queue for holding results. The 64 SIMD Lanes interact 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	Vector term	Closest CUDA/NVIDIA GPU term	Comment
Program abstractions	Vectorized Loop	Grid	Concepts are similar, with the GPU using the less descriptive term
	Chime	—	Because a vector instruction (PTX instruction) takes just 2 cycles on Pascal to complete, a chime is short in GPUs. Pascal has two execution units that support the most common floating-point instructions that are used alternately, so the effective issue rate is 1 instruction every clock cycle
Machine objects	Vector Instruction	PTX Instruction	A PTX instruction of a SIMD Thread is broadcast to all SIMD Lanes, so it is similar to a vector instruction
	Gather/ Scatter	Global load/store (ld. global/st.global)	All GPU loads and stores are gather and scatter, in that each SIMD Lane sends a unique address. It's up to the GPU Coalescing Unit to get unit-stride performance when addresses from the SIMD Lanes allow it
	Mask Registers	Predicate Registers and Internal Mask Registers	Vector mask registers are explicitly part of the architectural state, while GPU mask registers are internal to the hardware. The GPU conditional hardware adds a new feature beyond predicate registers to manage masks dynamically

Summary-向量处理机 vs. GPU

Type	Vector term	Closest CUDA/NVIDIA GPU term	Comment
Processing and memory hardware	Vector Processor	Multithreaded SIMD Processor	These are similar, but SIMD Processors tend to have many lanes, taking a few clock cycles per lane to complete a vector, while vector architectures have few lanes and take many cycles to complete a vector. They are also multithreaded where vectors usually are not
	Control Processor	Thread Block Scheduler	The closest is the Thread Block Scheduler that assigns Thread Blocks to a multithreaded SIMD Processor. But GPUs have no scalar-vector operations and no unit-stride or strided data transfer instructions, which Control Processors often provide in vector architectures
	Scalar Processor	System Processor	Because of the lack of shared memory and the high latency to communicate over a PCI bus (1000s of clock cycles), the system processor in a GPU rarely takes on the same tasks that a scalar processor does in a vector architecture
	Vector Lane	SIMD Lane	Very similar; both are essentially functional units with registers
	Vector Registers	SIMD Lane Registers	The equivalent of a vector register is the same register in all 16 SIMD Lanes of a multithreaded SIMD Processor running a thread of SIMD instructions. The number of registers per SIMD Thread is flexible, but the maximum is 256 in Pascal, so the maximum number of vector registers is 256
	Main Memory	GPU Memory	Memory for GPU versus system memory in vector case

Figure 4.21 GPU equivalent to vector terms.

Summary-Multimedia SIMD Computers and GPUs

Feature	Multicore with SIMD	GPU
SIMD Processors	4–8	8–32
SIMD Lanes/Processor	2–4	up to 64
Multithreading hardware support for SIMD Threads	2–4	up to 64
Typical ratio of single-precision to double-precision performance	2:1	2:1
Largest cache size	40 MB	4 MB
Size of memory address	64-bit	64-bit
Size of main memory	up to 1024 GB	up to 24 GB
Memory protection at level of page	Yes	Yes
Demand paging	Yes	Yes
Integrated scalar processor/SIMD Processor	Yes	No
Cache coherent	Yes	Yes on some systems

Figure 4.23 Similarities and differences between multicore with multimedia SIMD extensions and recent GPUs.

Type	More descriptive name used in this book	Official CUDA/ NVIDIA term	Short explanation and AMD and OpenCL terms	Official CUDA/NVIDIA definition
Program abstractions	Vectorizable loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more "Thread Blocks" (or bodies of vectorized loop) that can execute in parallel. OpenCL name is "index range." AMD name is "NDRange"	A Grid is an array of Thread Blocks that can execute concurrently, sequentially, or a mixture
	Body of Vectorized loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. These SIMD Threads can communicate via local memory. AMD and OpenCL name is "work group"	A Thread Block is an array of CUDA Threads that execute concurrently and can cooperate and communicate via shared memory and barrier synchronization. A Thread Block has a Thread Block ID within its Grid
	Sequence of SIMD Lane operations	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. AMD and OpenCL call a CUDA Thread a "work item"	A CUDA Thread is a lightweight thread that executes a sequential program and that can cooperate with other CUDA Threads executing in the same Thread Block. A CUDA Thread has a thread ID within its Thread Block
Machine object	A thread of SIMD instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results are stored depending on a per-element mask. AMD name is "wavefront"	A warp is a set of parallel CUDA Threads (e.g., 32) that execute the same instruction together in a multithreaded SIMT/SIMD Processor
	SIMD instruction	PTX instruction	A single SIMD instruction executed across the SIMD Lanes. AMD name is "AMDIL" or "FSAIL" instruction	A PTX instruction specifies an instruction executed by a CUDA Thread

Figure 4.24 Conversion from terms used in this chapter to official NVIDIA/CUDA and AMD jargon. OpenCL names are given in the book's definitions.

Type	More descriptive name used in this book	Official CUDA/ NVIDIA term	Short explanation and AMD and OpenCL terms	Official CUDA/NVIDIA definition
Processing hardware	Multithreaded SIMD processor	Streaming multiprocessor	Multithreaded SIMD Processor that executes thread of SIMD instructions, independent of other SIMD Processors. Both AMD and OpenCL call it a "compute unit." However, the CUDA programmer writes program for one lane rather than for a "vector" of multiple SIMD Lanes	A streaming multiprocessor (SM) is a multithreaded SIMT/SIMD Processor that executes warps of CUDA Threads. A SIMT program specifies the execution of one CUDA Thread, rather than a vector of multiple SIMD Lanes
	Thread Block Scheduler	Giga Thread Engine	Assigns multiple bodies of vectorized loop to multithreaded SIMD Processors. AMD name is "Ultra-Threaded Dispatch Engine"	Distributes and schedules Thread Blocks of a grid to streaming multiprocessors as resources become available
	SIMD Thread scheduler	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. AMD name is "Work Group Scheduler"	A warp scheduler in a streaming multiprocessor schedules warps for execution when their next instruction is ready to execute
	SIMD Lane	Thread processor	Hardware SIMD Lane that executes the operations in a thread of SIMD instructions on a single element. Results are stored depending on mask. OpenCL calls it a "processing element." AMD name is also "SIMD Lane"	A thread processor is a datapath and register file portion of a streaming multiprocessor that executes operations for one or more lanes of a warp

Type	More descriptive name used in this book	Official CUDA/ NVIDIA term	Short explanation and AMD and OpenCL terms	Official CUDA/NVIDIA definition
Memory hardware	GPU Memory	Global memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU. OpenCL calls it "global memory"	Global memory is accessible by all CUDA Threads in any Thread Block in any grid; implemented as a region of DRAM, and may be cached
	Private memory	Local memory	Portion of DRAM memory private to each SIMD Lane. Both AMD and OpenCL call it "private memory"	Private "thread-local" memory for a CUDA Thread; implemented as a cached region of DRAM
	Local memory	Shared memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors. OpenCL calls it "local memory." AMD calls it "group memory"	Fast SRAM memory shared by the CUDA Threads composing a Thread Block, and private to that Thread Block. Used for communication among CUDA Threads in a Thread Block at barrier synchronization points
	SIMD Lane registers	Registers	Registers in a single SIMD Lane allocated across body of vectorized loop. AMD also calls them "registers"	Private registers for a CUDA Thread; implemented as multithreaded register file for certain lanes of several warps for each thread processor

Figure 4.25 Conversion from terms used in this chapter to official NVIDIA/CUDA and AMD jargon. Note that our descriptive terms "local memory" and "private memory" use the OpenCL terminology. NVIDIA uses SIMT (single-instruction multiple-thread) rather than SIMD to describe a streaming multiprocessor. SIMT is preferred over SIMD because the per-thread branching and control flow are unlike any SIMD machine.

GPU Readings

- Lindholm et al., "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE Micro 2008.
- Fatahalian and Houston, "A Closer Look at GPUs," CACM 2008.
- Narasiman et al., "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," MICRO 2011.
- Fung et al., "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," MICRO 2007.
- Jog et al., "Orchestrated Scheduling and Prefetching for GPGPUs," ISCA 2013.

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/