计算机体系结构

周学海
xhzhou@ustc.edu.cn
0551-63492149
中国科学技术大学

Review：General-Purpose GPUs

- **随着Dennard Scaling定律的失效，通过提高主频提升性能越来越困难，需要探索更有效的硬件结构-通过专用硬件结构有可能使得能效提升500+**
  - 通过配置支持向量操作的部件（例如GPU），**减少指令处理的额外开销**
  - 通过引入较复杂的操作完成多个ALU运算避免大量访问存储器 **减少数据搬移**
  - **主要挑战：**平衡专用硬件加速器带来的性能提升与系统的适用性之间的矛盾
- **基于硬件加速器（DSA）的系统正成为体系结构发展的重要方向之一**
  - 寒武纪的智能处理器：思源370、思源270、思源290
  - Google： Tensor Processing Unit
  - 机器学习应用中大量的计算任务迁移到专用加速器上运行
- **现代GPU具备图灵完备性**
  - 支持在足够的存储空间内，用足够的时间可以完成的计算
  - 现代超级计算机大量采用GPU，以提高系统的能效 （性能/瓦）
- **GP-GPU的基本思想**
  - 发挥GPU**计算的高性能**和**存储器的高带宽**来加速数据并行性高的任务
  - 是一种协处理器（GPU作为附加设备）： CPU将数据并行的kernels迁移到GP-GPU上运行

Review: three key ideas

- **GPU使用大量”简单核心”（多核）并行执行**
- **核心中配置大量ALU部件形成SIMD处理模式**
  - Option 1: Explicit SIMD vector instruction
  - Option 2: Implicit sharing managed by hardware

- **通过交叉执行不同线程组处理不同数据片段避免指令流运行时的长延时（Stall)**
  - When one group stalls, work on another group

- **GPU如何处理程序中的分支？vs. 向量处理器模型中的Conditional Execution**

- **从用户的角度出发，GPU的编程模型?**

6.3-2 GPU II

GPU
编程模型

GPU
分支处理

GPUs are SIMD Engines Underneath

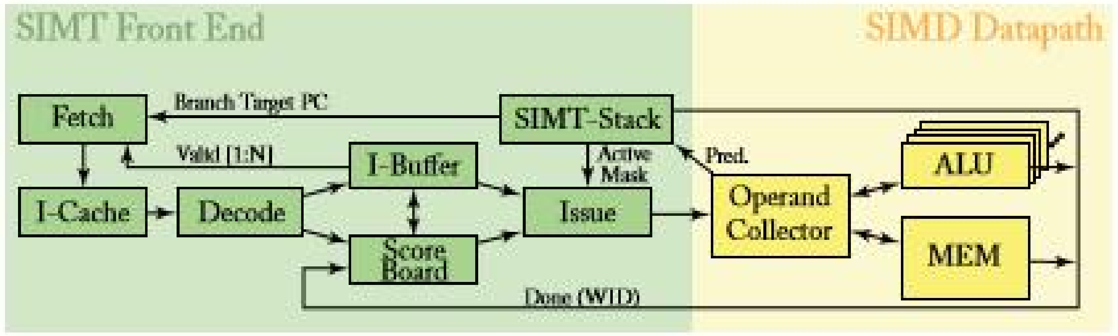
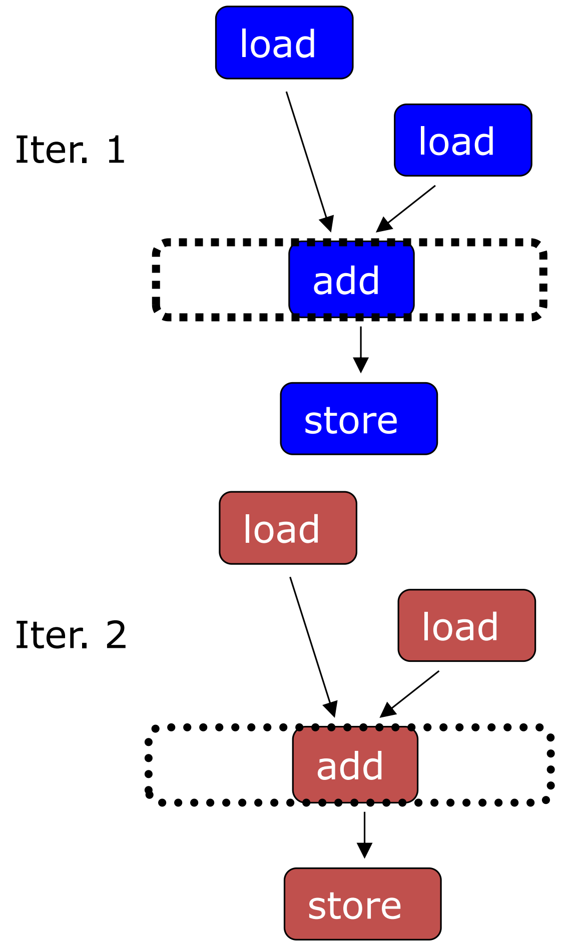


Figure 3.1: Microarchitecture of a generic GPGPU core.

- **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, 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)

如何挖掘程序的并行性?

Scalar Sequential Code



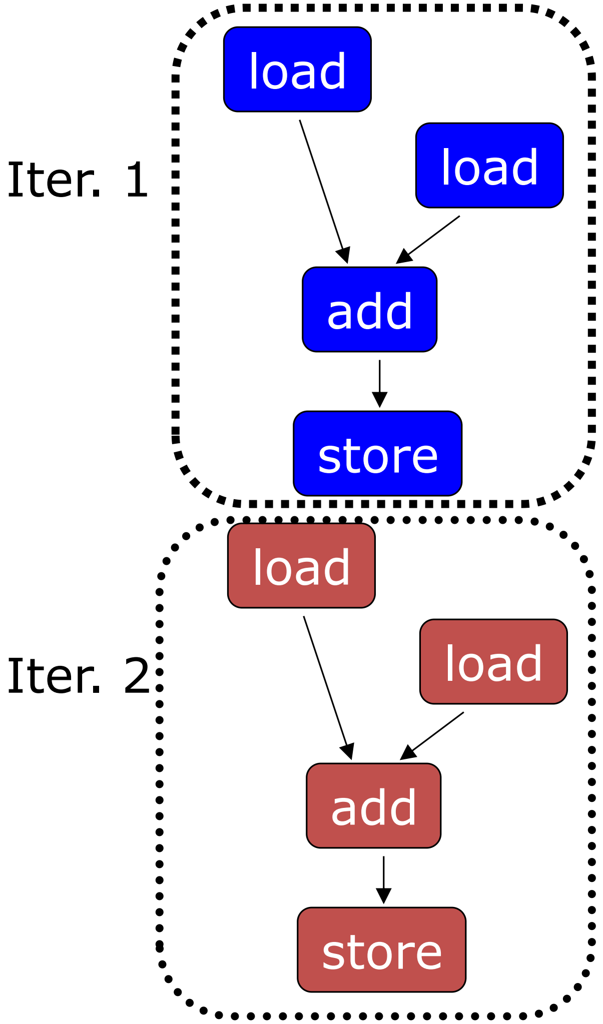
```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

三种编程模式来挖掘程序的并行性：

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

Prog. Model 1: Sequential (SISD)

Scalar Sequential Code



编程模型

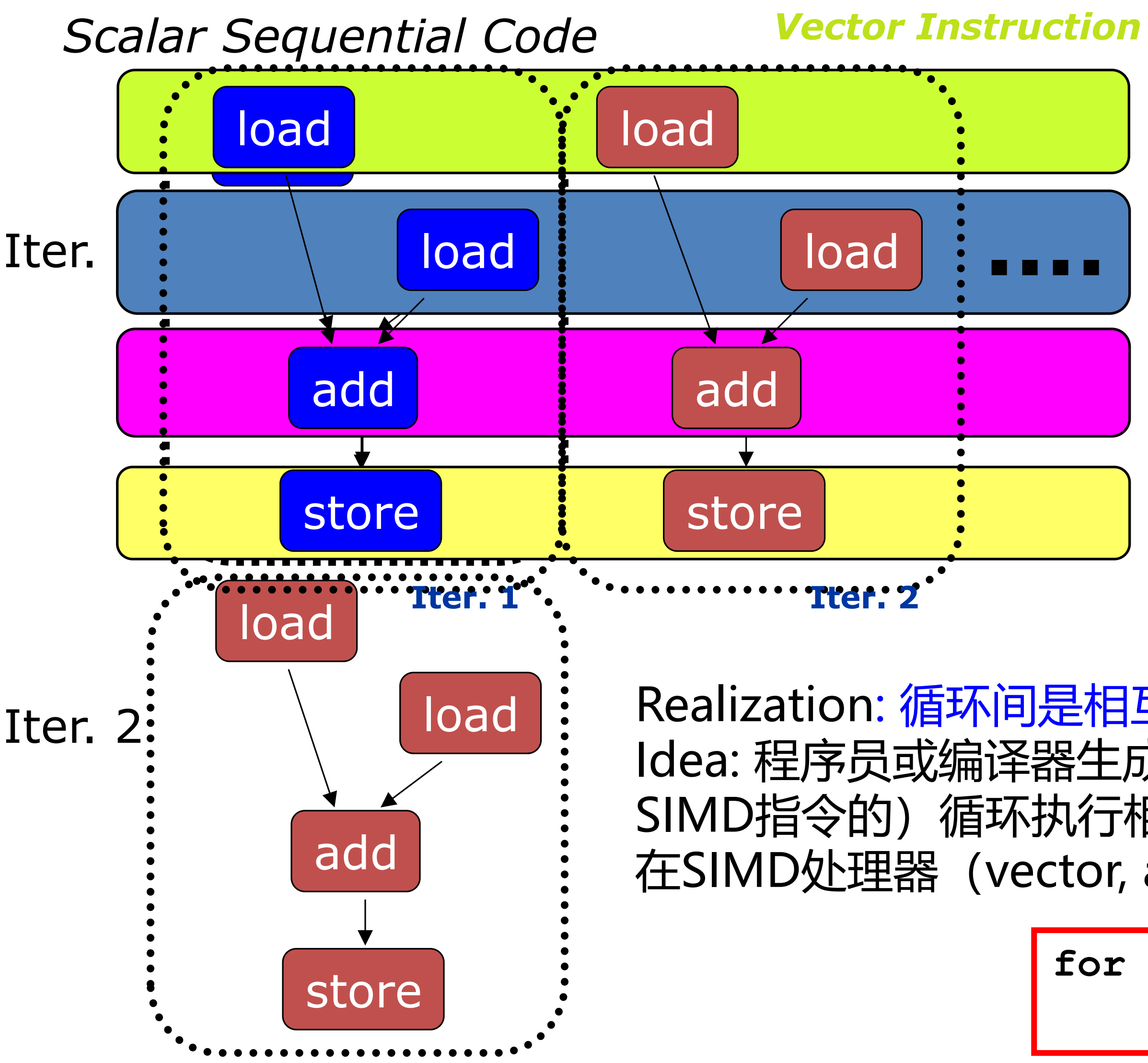
执行模型

可以采用如下不同类型的处理器执行

- **Pipelined processor**
- **Out-of-order execution processor**
  - 就绪的相互无关的指令
  - 不同循环的指令缓存在指令窗口中，多个功能部件可以并行执行
  - 即：硬件做循环展开
- **Superscalar or VLIW processor**
  - 每个cycle可以存取和执行多条指令

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

Prog. Model 2: Data Parallel (SIMD)



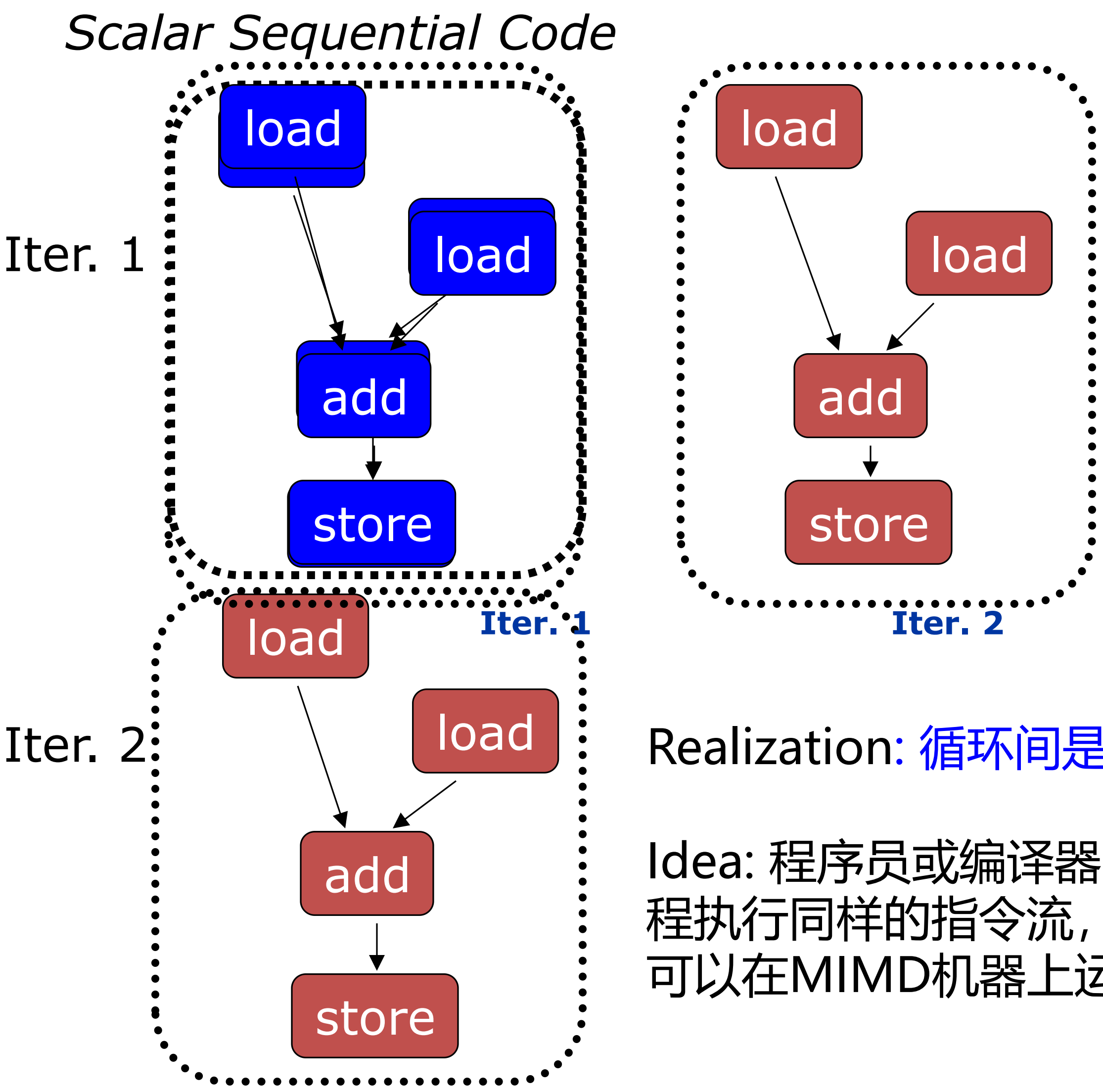
Vectorized Code

VLD	A → V1
VLD	B → V2
VADD	V1 + V2 → V3
VST	V3 → C

Realization: 循环间是相互独立的，循环间没有数据依赖
Idea: 程序员或编译器生成SIMD指令，所有的（聚合为SIMD指令的）循环执行相同的指令，处理不同的数据
在SIMD处理器（vector, array）上运行

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

Prog. Model 3: Multithreaded



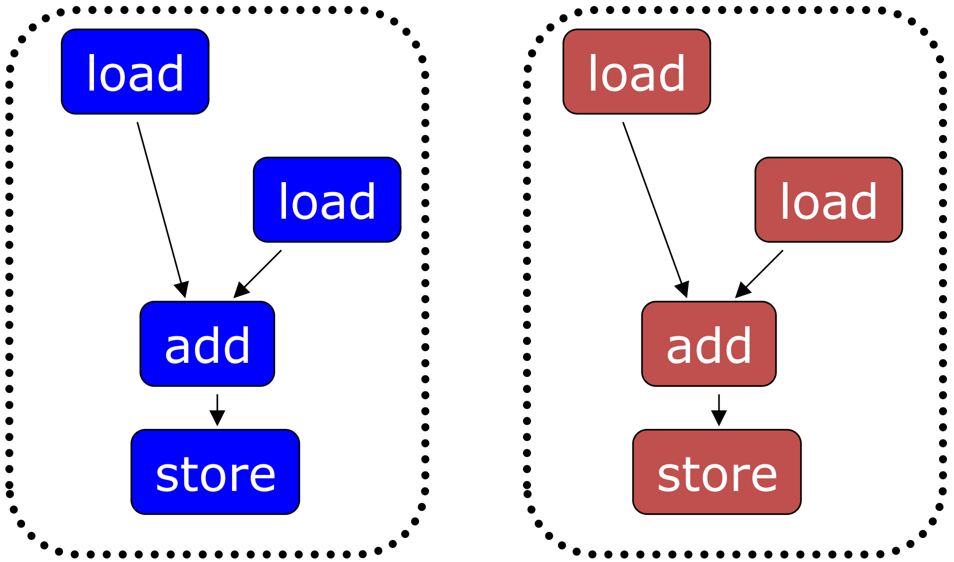
```
for (i=0; i < N; i++)
     C[i] = A[i] + B[i];
```

▪▪▪▪

Realization: 循环间是相互独立的，循环间没有数据依赖

Idea: 程序员或编译器为每次循环生成一个线程。每个线程执行同样的指令流，处理不同的数据
可以在MIMD机器上运行

Prog. Model 3: Multithreaded



Iter. 1　　Iter. 2

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

....

Reali

Idea:

行同

执

可以

这种模式也称为:

SPMD: Single Program Multiple Data

可以在SIMT 机器上运行

Single Instruction Multiple Thread

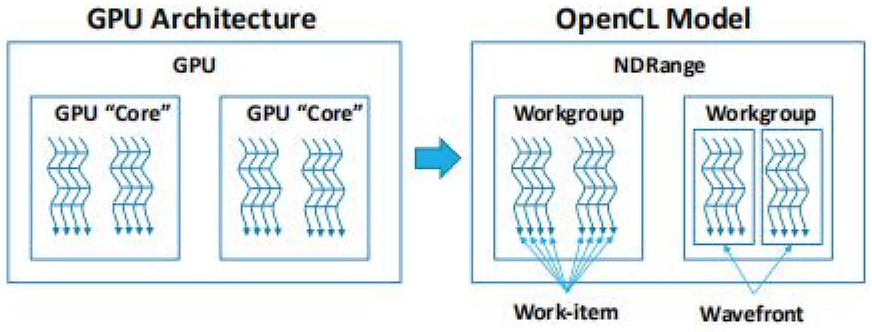
SPMD

- **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

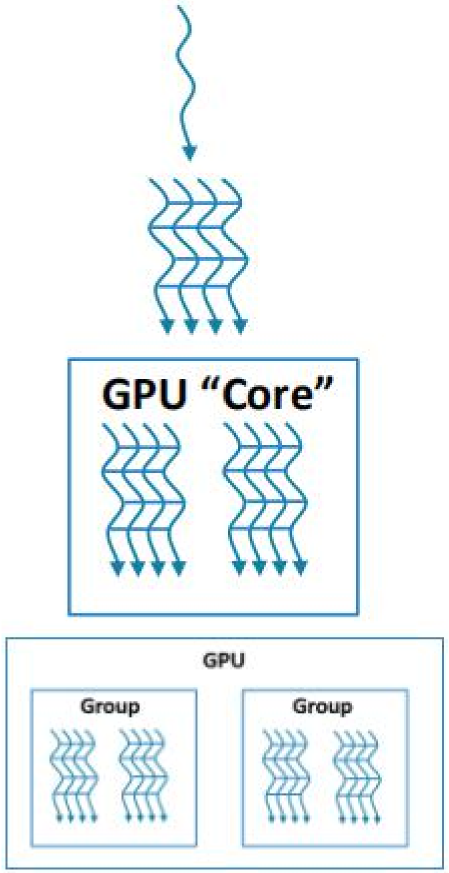
- **多条指令流执行相同的程序**
  - 每个程序/过程
    - 操作不同的数据
    - 运行时可以执行不同的控制流路径
  - 许多科学计算应用以这种方式编程，运行在MIMD硬件结构上 (multiprocessors)
  - 现代 GPUs 以这种类似的方式编程，运行在SIMD硬件上

GPU 编程框架

- **CUDA – Compute Unified Device Architecture**
  – Nvidia 研制开发的专用模型，第一个GPGPU编程环境
- **C++ AMP – C++ Accelerated Massive Parallelism**
  – 微软研制开发，CUDA/OpenCL的更高层抽象
- **OpenACC – Open Accelerator**
  – Like OpenMP for GPUs (semi-auto-parallelize serial code)，CUDA/OpenCL的更高层抽象
  – 2011年11月，Cray、PGI、CAPS和英伟达4家公司联合推出OpenACC 1.0编程标准
- **OpenCL**
  – 最早由苹果公司研制开发，后来形成开放的异构平台编程规范
  – 异构平台编程框架，OpenCL 用来编写设备端程序
  – OpenCL工作组的成员包括：3Dlabs、AMD、苹果、ARM、Codeplay、爱立信、飞思卡尔、华为、HSA基金会、GraphicRemedy、IBM、Imagination Technologies、Intel、诺基亚、NVIDIA、摩托罗拉、QNX、高通，三星、Seaweed、德州仪器、布里斯托尔大学、瑞典Ume大学



一些术语



CUDA/Nvidia	OpenCL/AMD	Henn&Patt
Thread	Work-item	Sequence of SIMD Lane Operations
Warp	Wavefront	Thread of SIMD Instructions
Block	Workgroup	Body of vectorized loop
Grid	NDRange	Vectorized loop

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（Block)**
- **许多线程块组成一个网格（Grid)**

- **GPU 由硬件对线程进行管理**
  - 两级调度
    - Thread Block Scheduler
    - SIMD Thread Scheduler
  - Warp
    - SIMD线程
    - 线程调度的基本单位

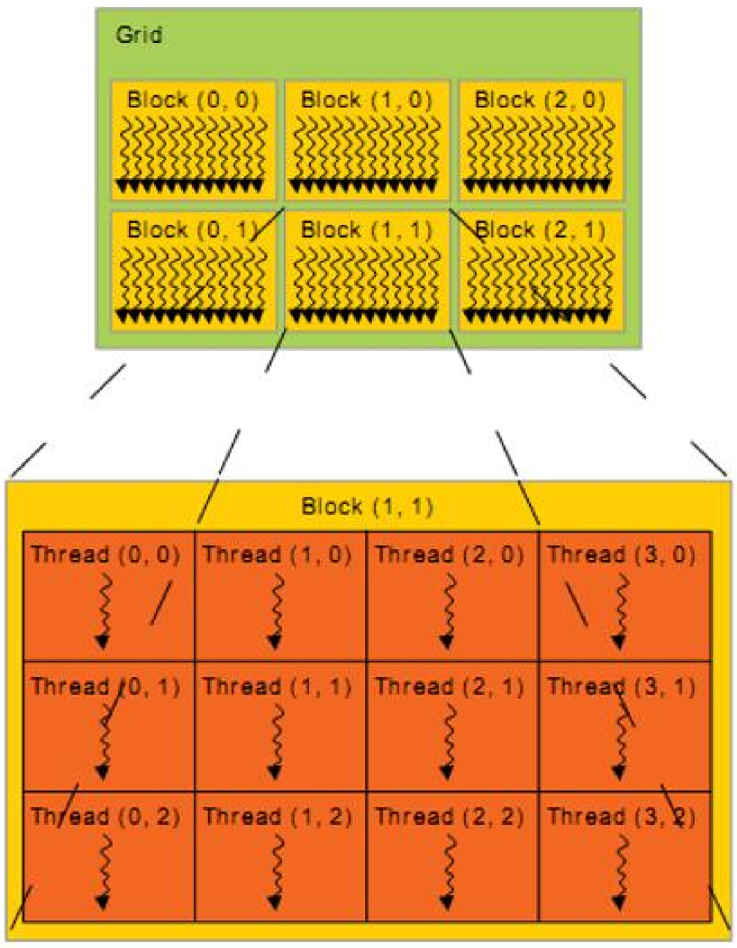
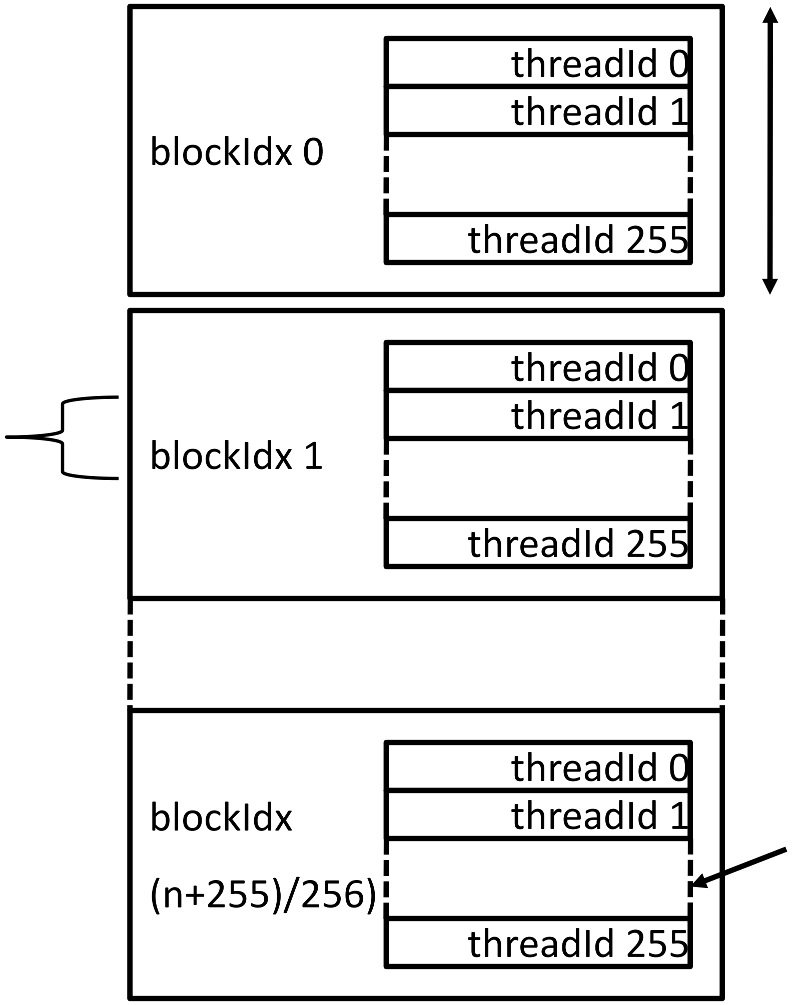


Figure 6 Grid of Thread Blocks

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

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])
```

处理向量长度为8192的程序组织：

Grid：（循环）
1、16个block/Grid
2、512个向量元素/Block

Block：（循环体）
3、16个SIMD线程/Block
4、32个CUDA线程/SIMD线程
5、处理一个元素/CUDA线程

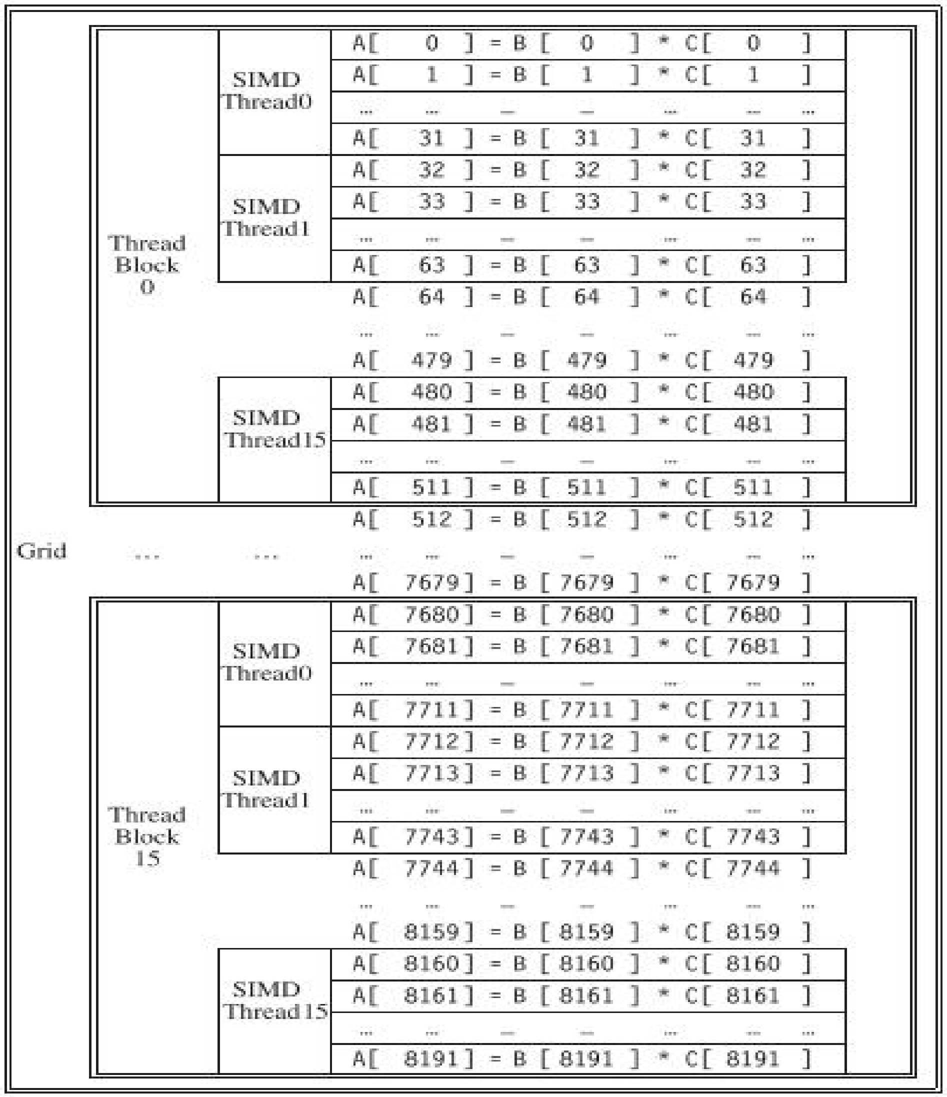
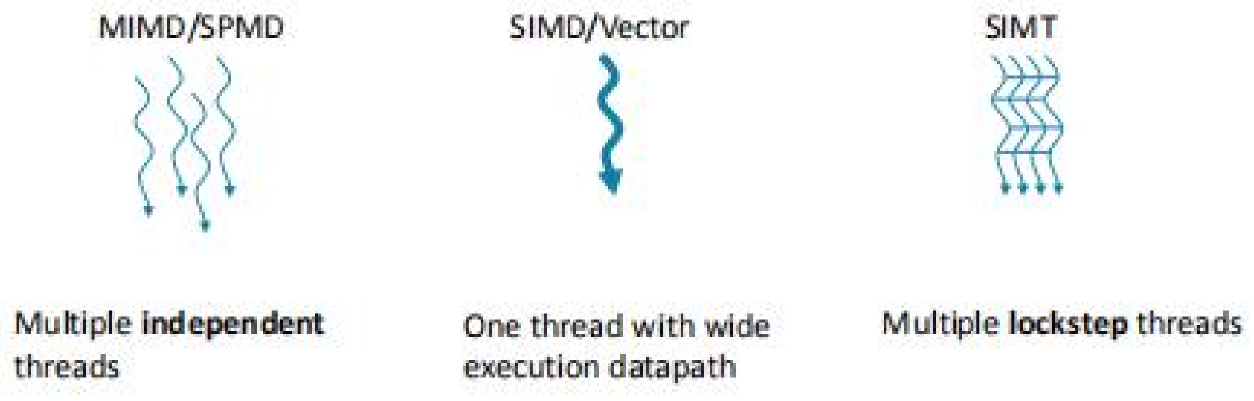

Grid
Thread Block 0
SIMD Thread0
A[0] = B [0] * C[0]
A[1] = B [1] * C[1]
A[31] = B [31] * C[31]
SIMD Thread1
A[32] = B [32] * C[32]
A[33] = B [33] * C[33]
A[63] = B [63] * C[63]
A[64] = B [64] * C[64]
A[479] = B [479] * C[479]
SIMD Thread15
A[480] = B [480] * C[480]
A[481] = B [481] * C[481]
A[511] = B [511] * C[511]
A[512] = B [512] * C[512]
A[7679] = B [7679] * C[7679]
Thread Block 15
SIMD Thread0
A[7680] = B [7680] * C[7680]
A[7681] = B [7681] * C[7681]
A[7711] = B [7711] * C[7711]
SIMD Thread1
A[7712] = B [7712] * C[7712]
A[7713] = B [7713] * C[7713]
A[7743] = B [7743] * C[7743]
A[7744] = B [7744] * C[7744]
A[8159] = B [8159] * C[8159]
SIMD Thread15
A[8160] = B [8160] * C[8160]
A[8161] = B [8161] * C[8161]
A[8191] = B [8191] * C[8191]

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.)

A GPU is a SIMD (SIMT) Machine

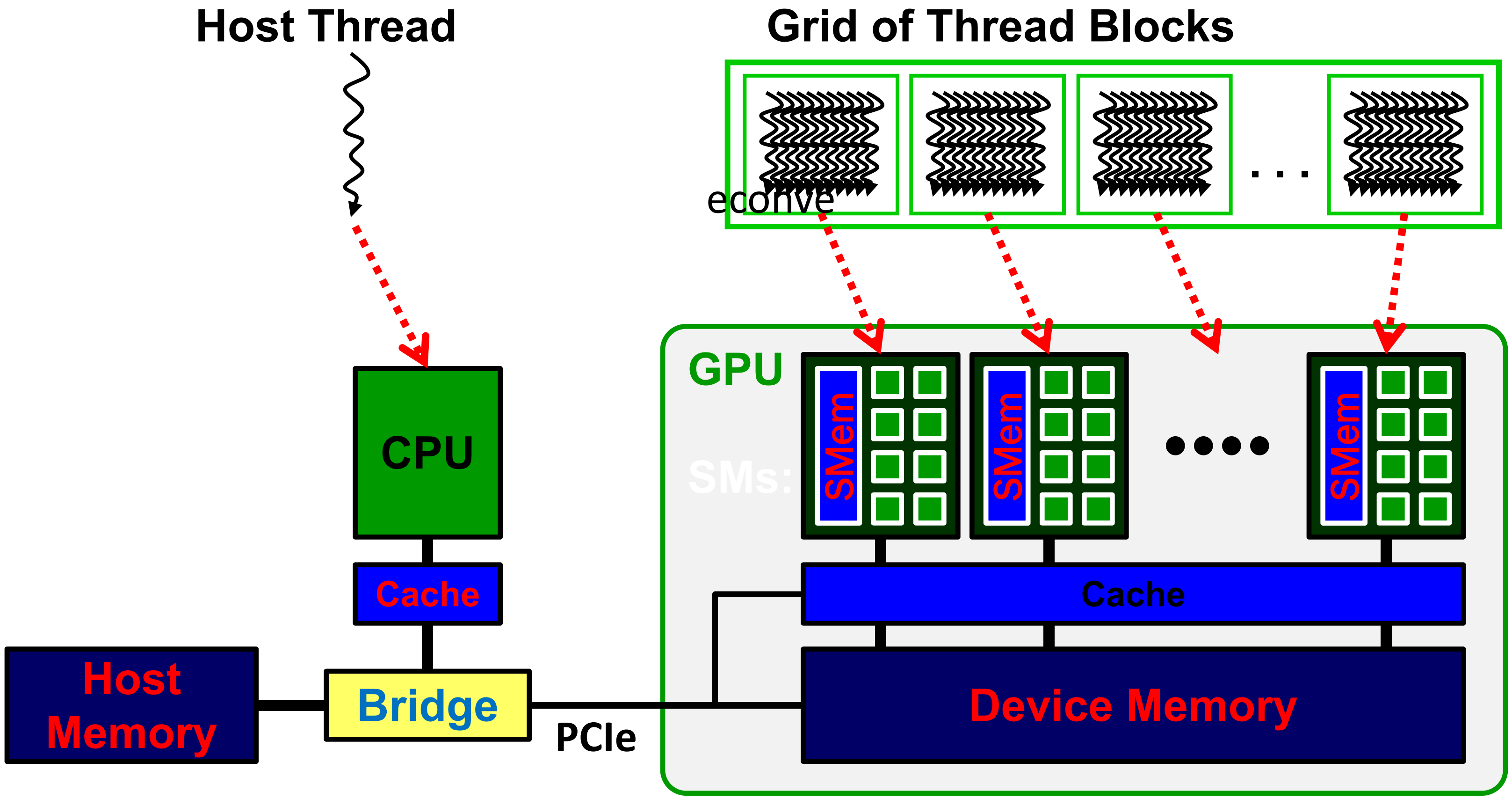
- **GPU不是用SIMD指令编程**
- **使用多线程模型 (一种SPMD 编程模型)**
  - 每个线程执行同样的代码，但操作不同的数据元素
  - 每个线程有自己的上下文(即可以独立地启动/执行等)
- **一组执行相同指令的线程由硬件动态组织成warp**
  - 一个warp是由硬件形成的SIMD操作
  - lockstep模式执行


MIMD/SPMD
SIMD/Vector
SIMT
Multiple **independent** threads
One thread with wide execution datapath
Multiple **lockstep** threads

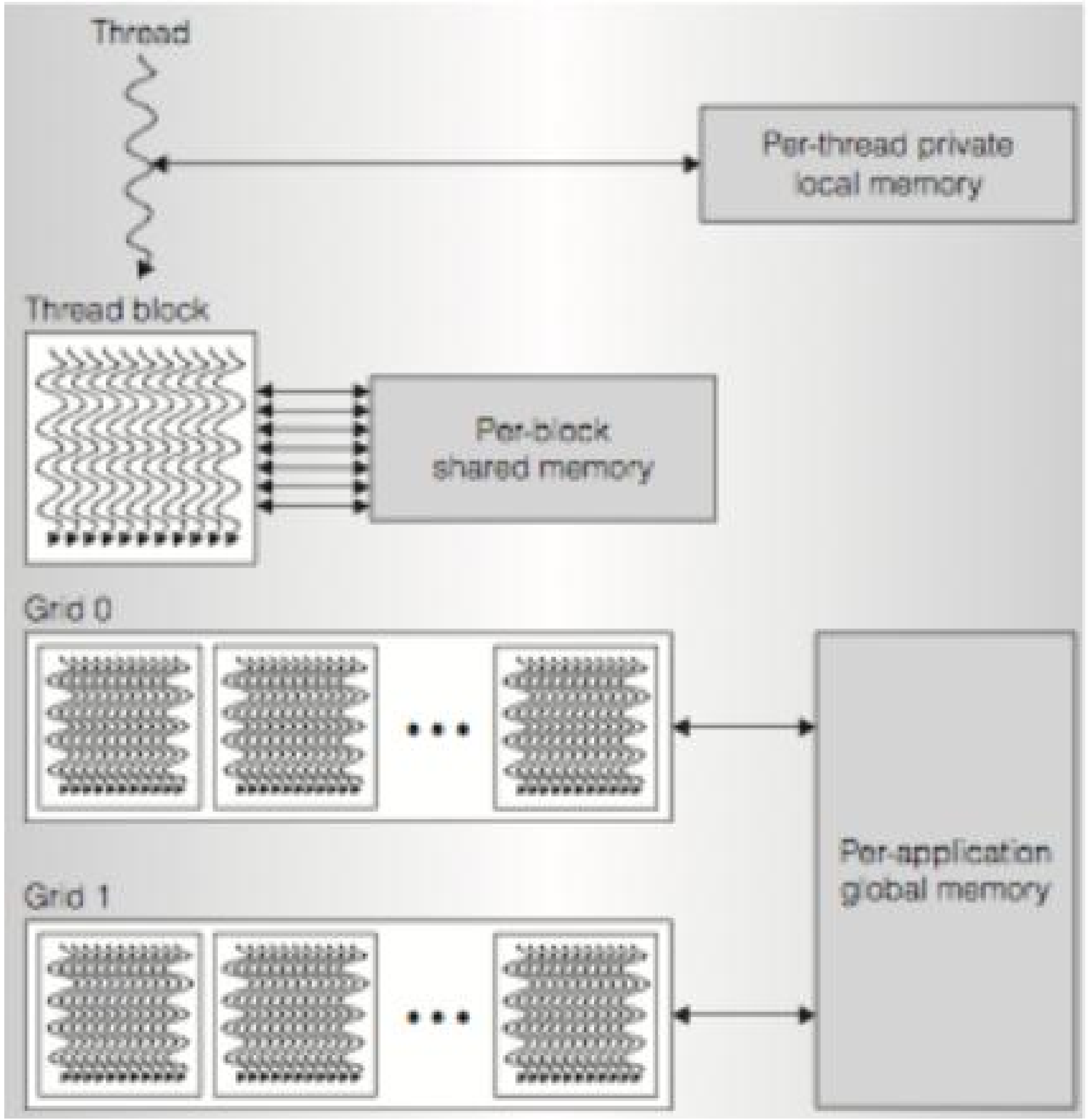
数据并行不同的执行模式

CUDA kernel maps to Grid of Blocks

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



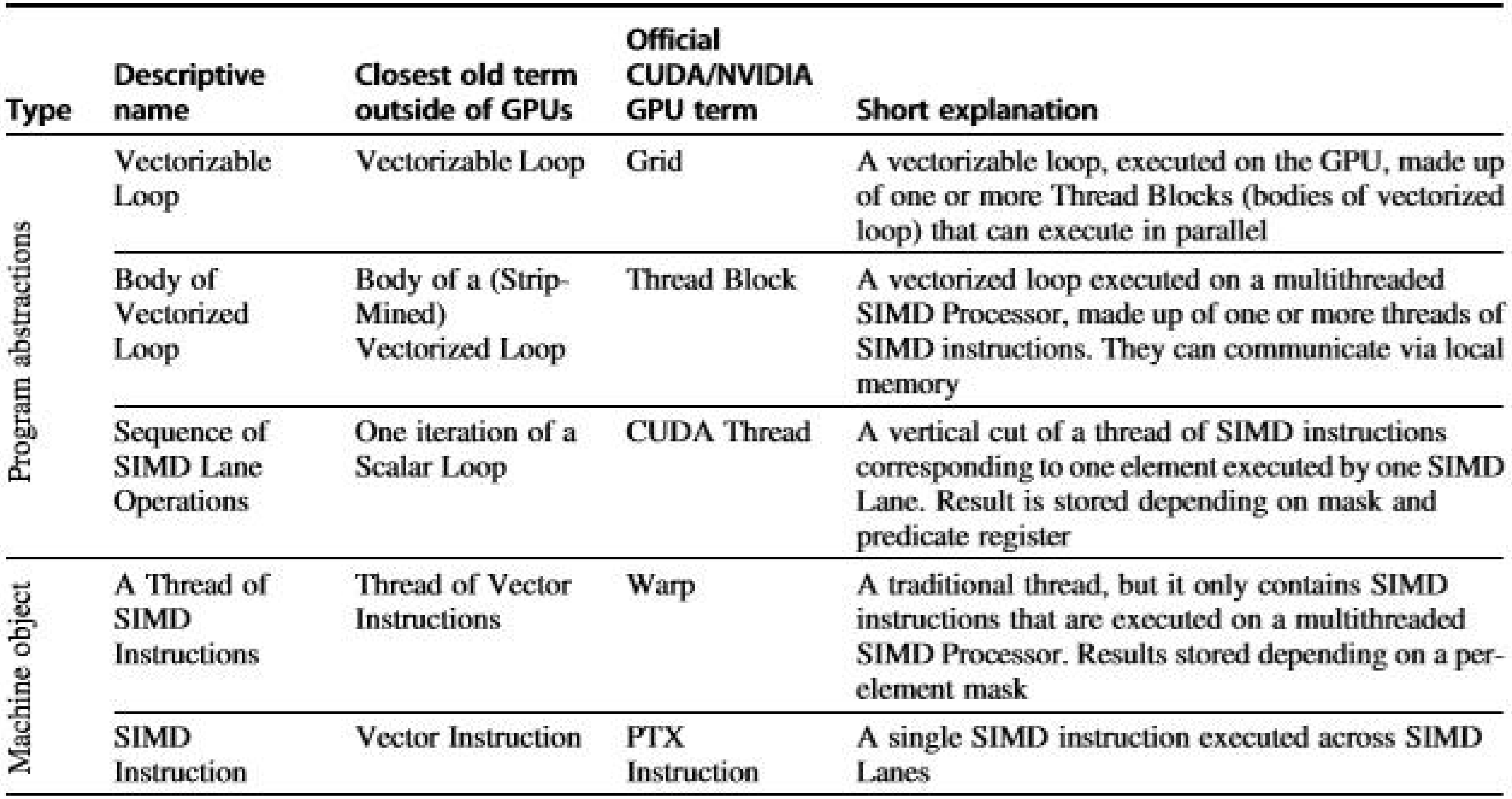
GPU Memory Hierarchy


Thread
Per-thread private local memory
Thread block
Per-block shared memory
Grid 0
Grid 1
Per-application global memory

[Nvidia, 2010]

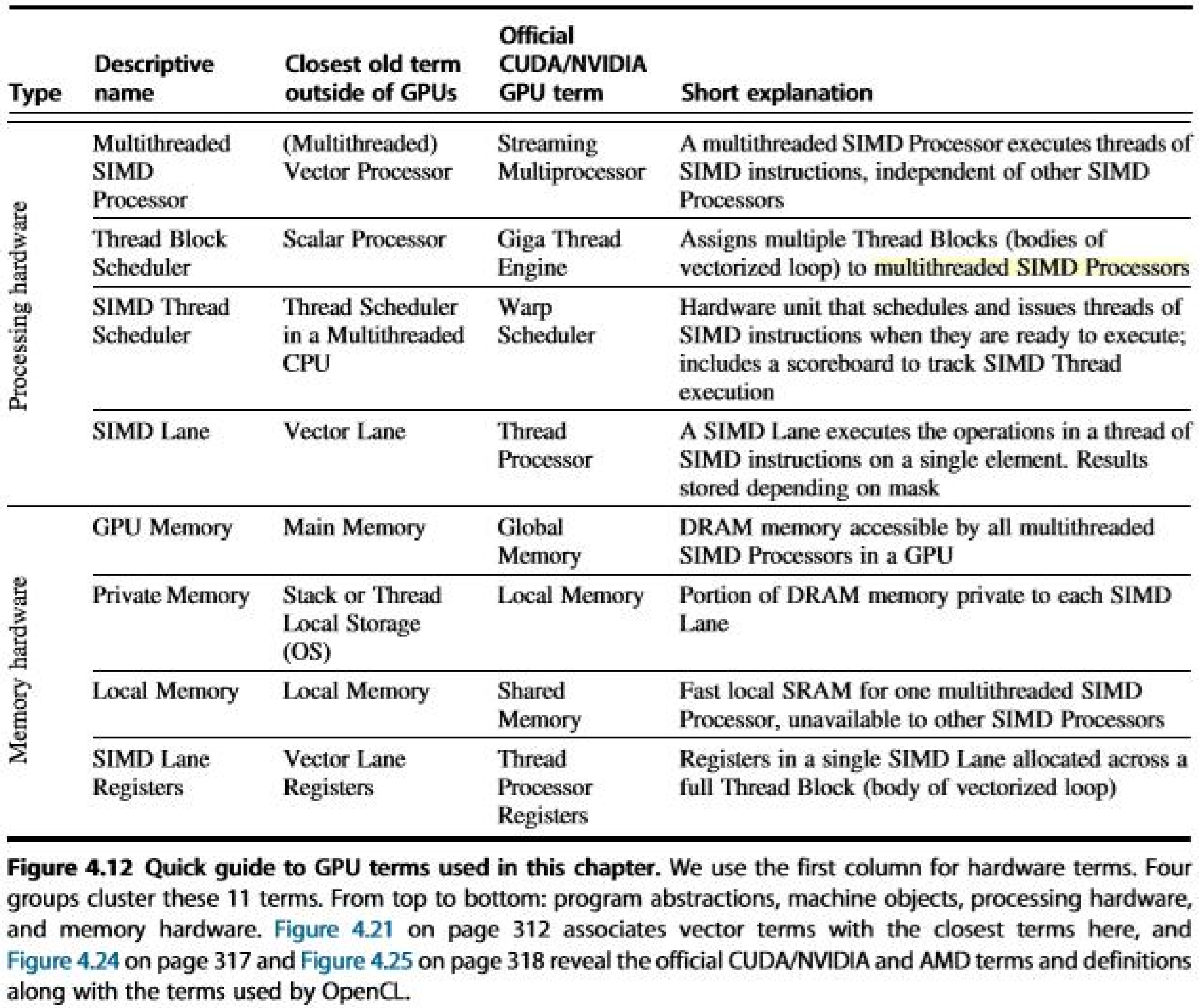
Summary- 向量处理机 vs. GPU

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.

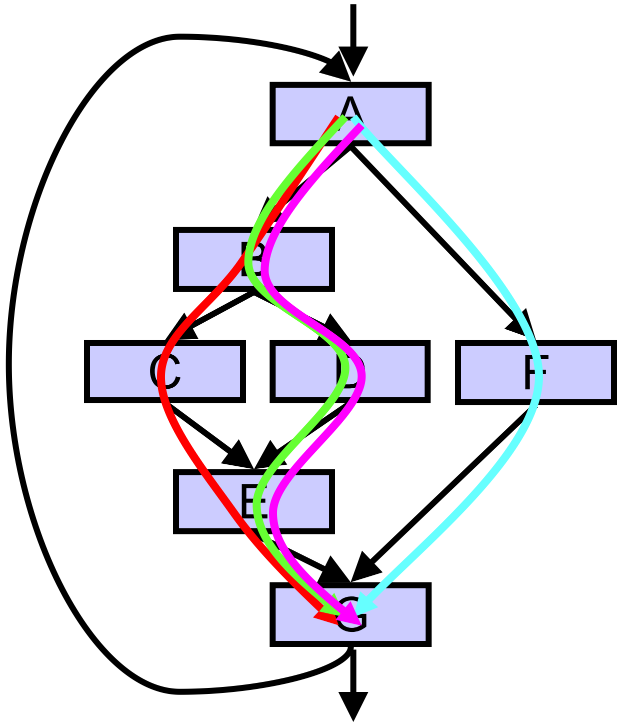
6.3-2 GPU II

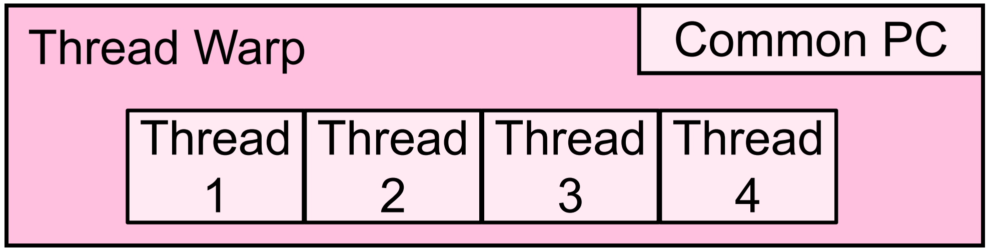
GPU
编程模型

GPU
分支处理

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



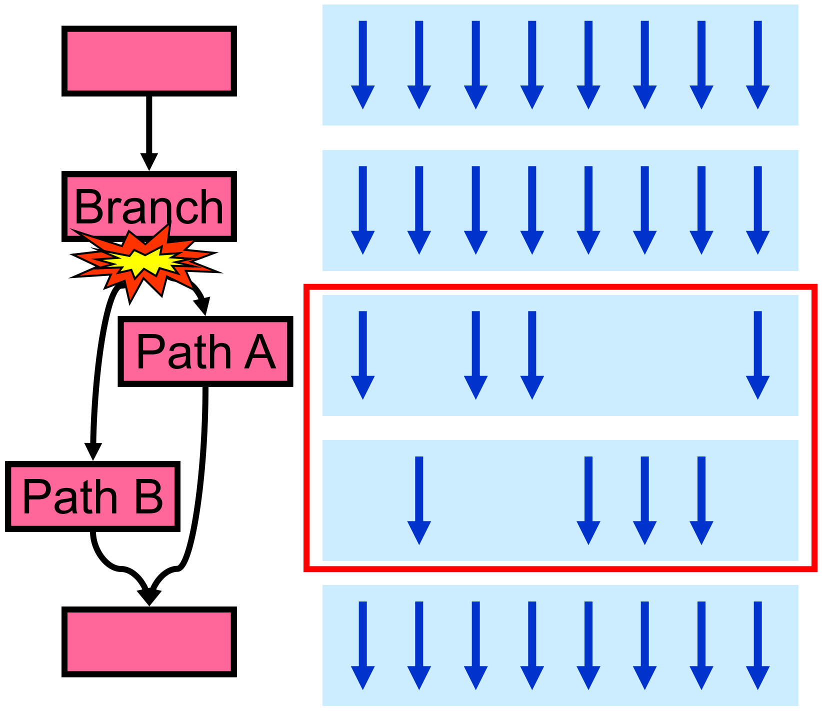


Slide credit: Tor Aamodt

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
  - 每个线程车道包含由程序员指定的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）**
- **如果所有线程路径相同，则保持这种 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%

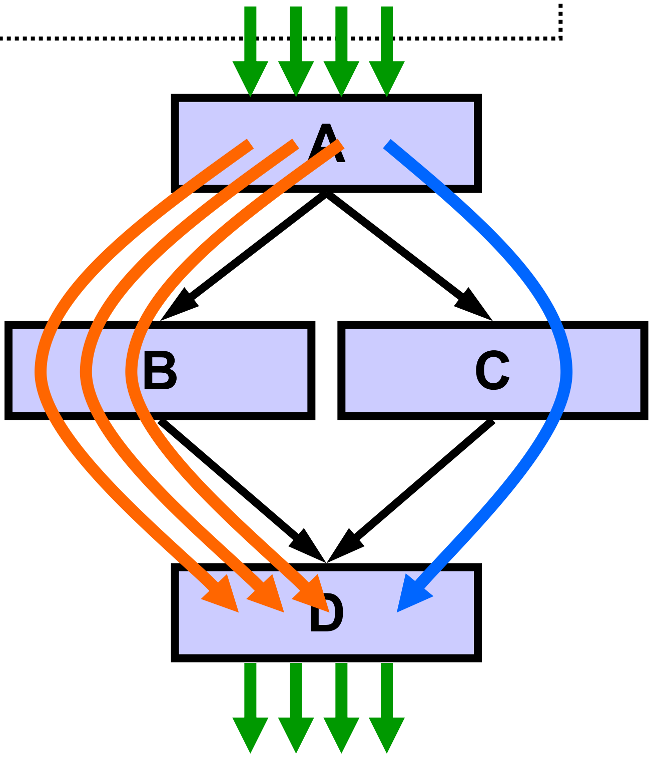
Example

```
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;
```

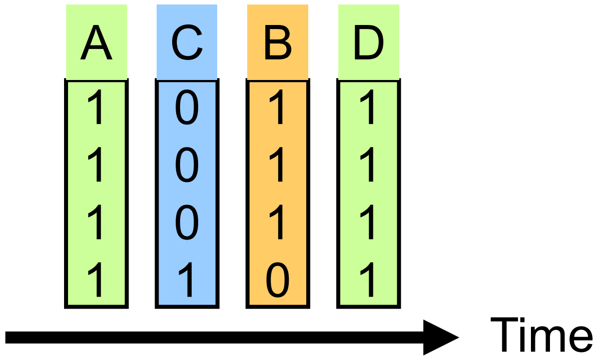


One per warp

Control Flow Stack

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

Execution Sequence



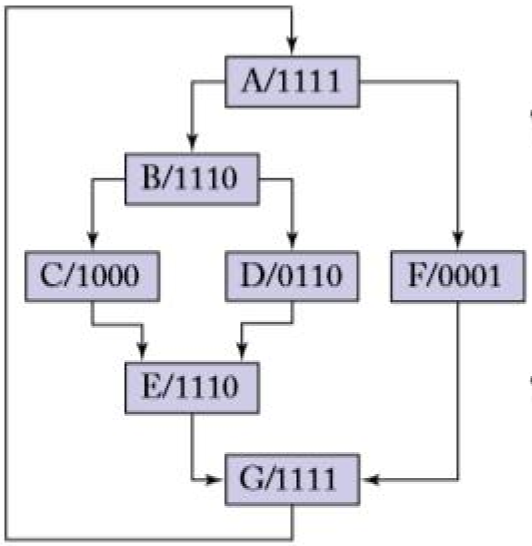
SIMT stack operation

```
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.



(a) Example Program

	Ret./Reconv. PC	Next PC	Active Mask
	-	G	1111
	G	F	0001
TOS →	G	B	1110

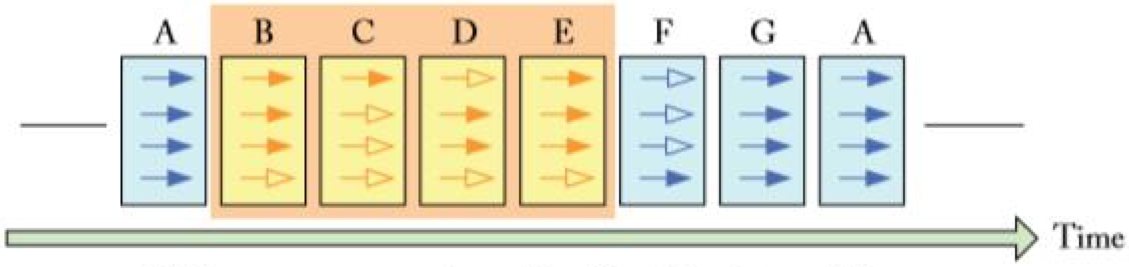
(c) Initial State

	Ret./Reconv. PC	Next PC	Active Mask	
	-	G	1111	
	G	F	0001	
	G	E	1110	(i)
	E	D	0110	(ii)
TOS →	E	C	1000	(iii)

(d) After Divergent Branch

	Ret./Reconv. PC	Next PC	Active Mask
	-	G	1111
	G	F	0001
TOS →	G	E	1110

(e) After Reconvergence



(b) Re-convergence at Immediate Post-Dominator of B

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

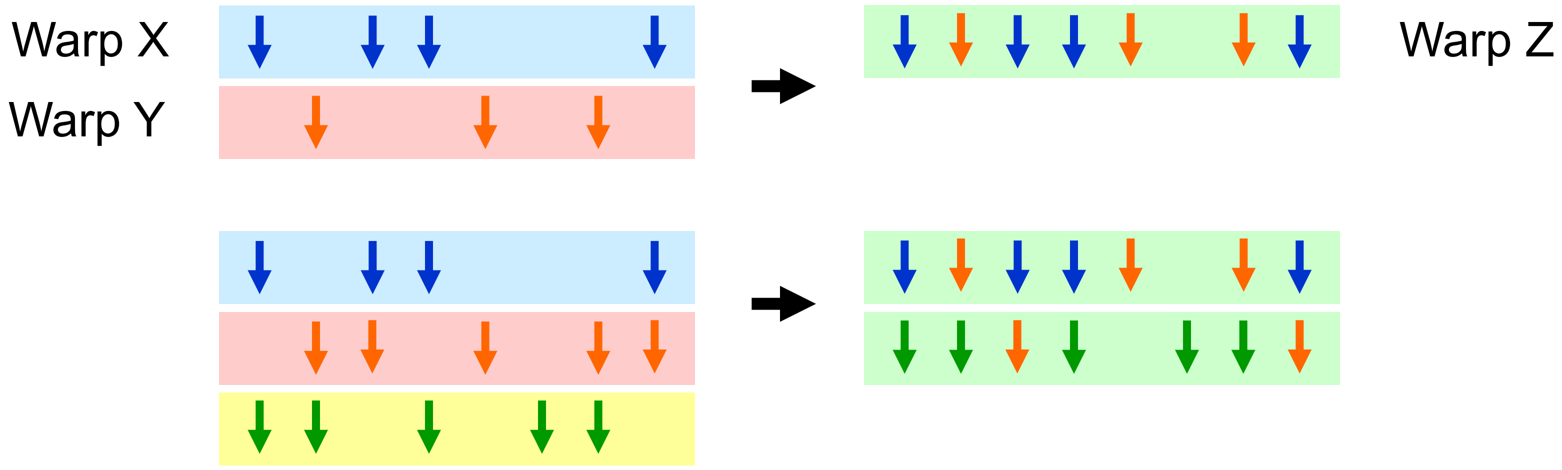
Remember: Each Thread Is Independent

- **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)

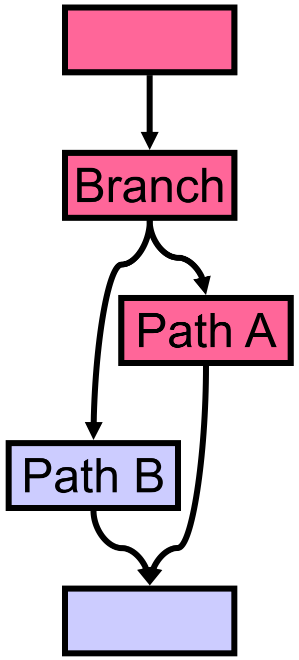
Dynamic Warp Formation/Merging

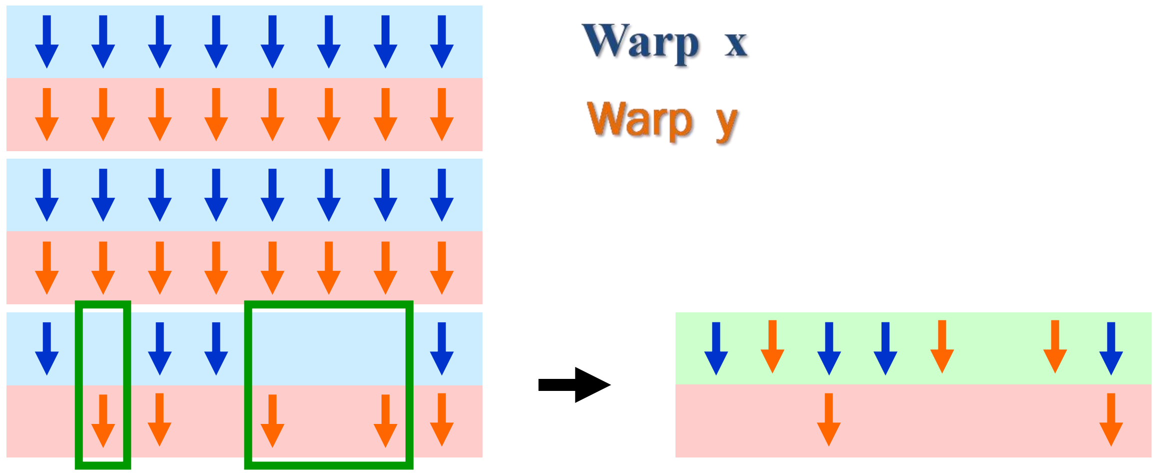
- **Idea:** 分支发散之后，动态合并执行相同指令的线程
  - 从那些等待的warp中形成新的warp
  - 足够多的线程分支到每个路径，有可能创建完整的新warp



Dynamic Warp Formation/Merging

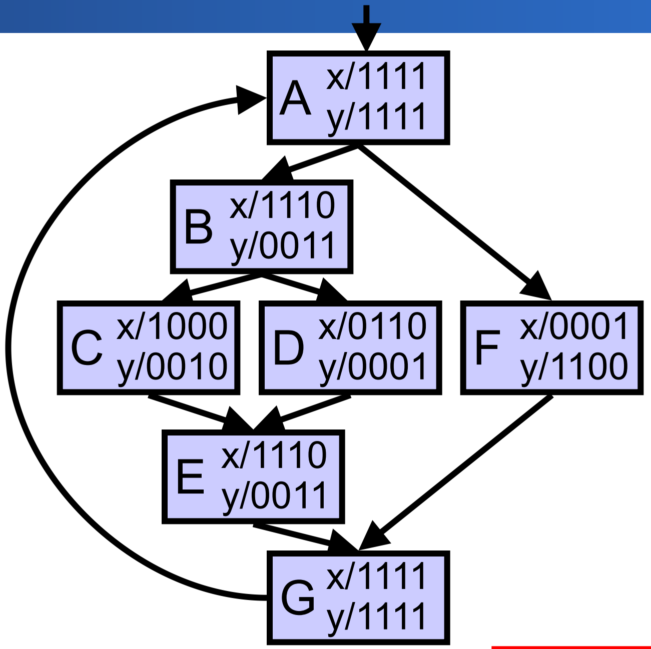
- Idea:
  - 分支发散之后，动态合并执行相同指令的线程

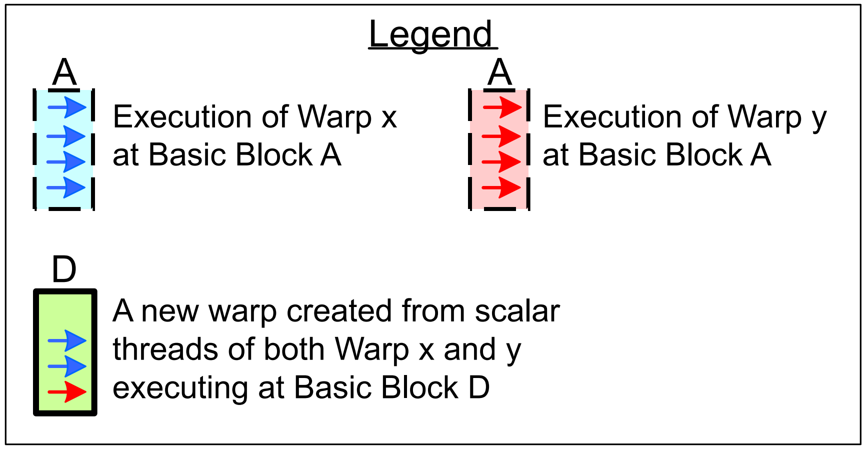


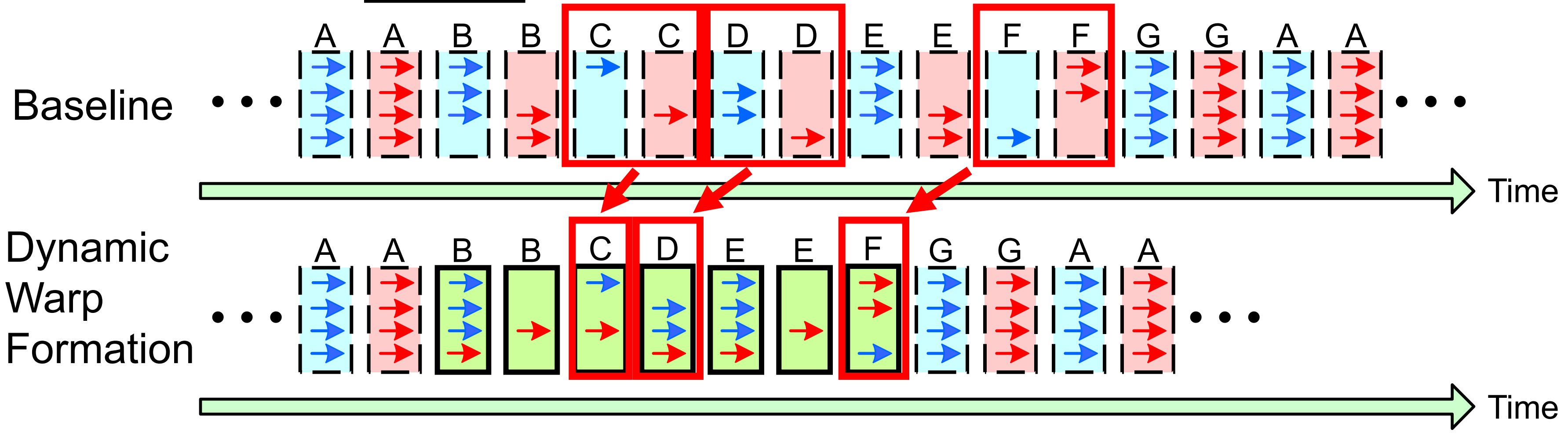


- Fung et al., “Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow,” MICRO 2007.

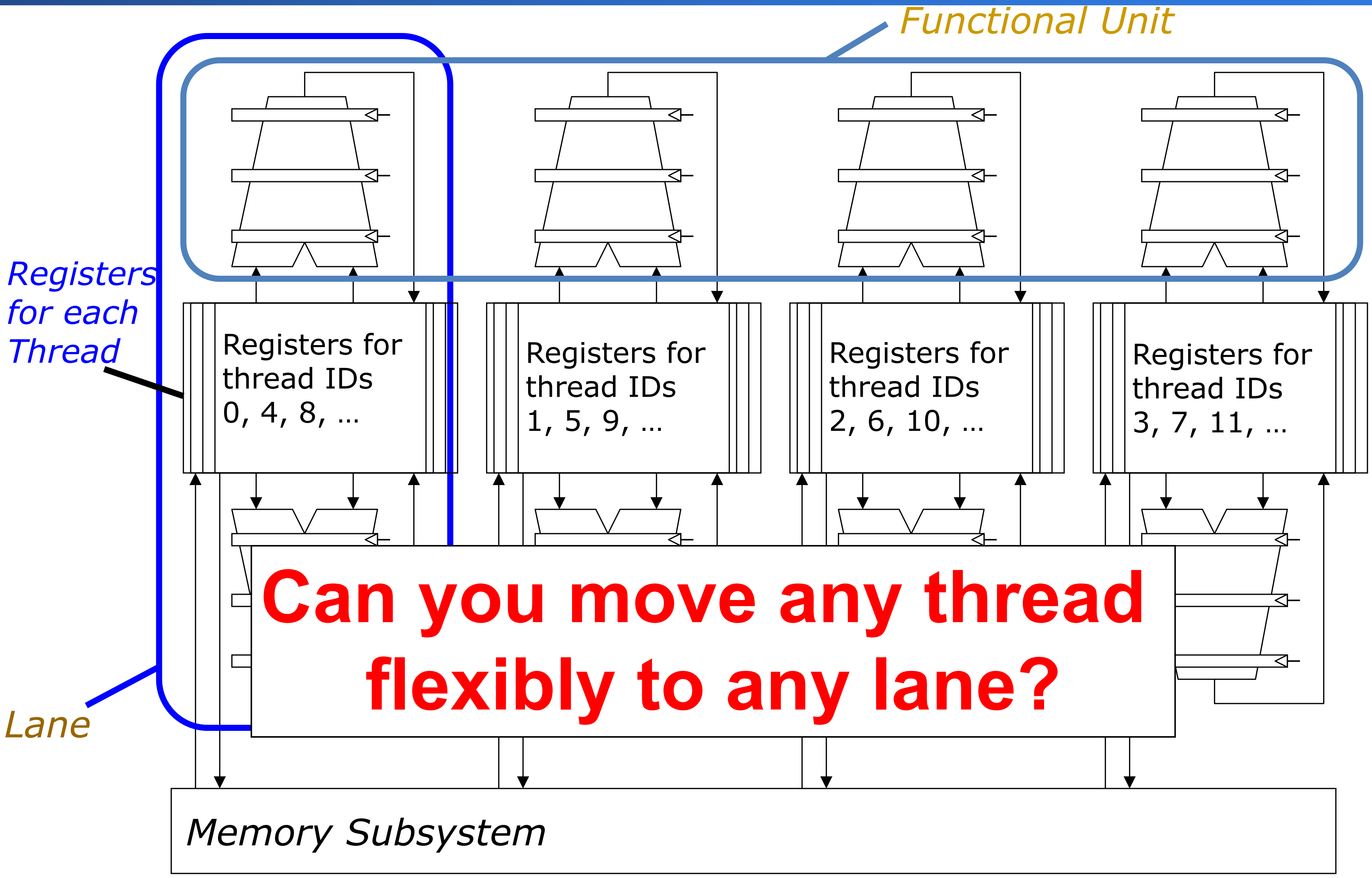
Dynamic Warp Formation Example




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



Hardware Constraints Limit Flexibility of Warp Grouping


Functional Unit
Registers for each Thread
Lane
Registers for thread IDs 0, 4, 8, …
Registers for thread IDs 1, 5, 9, …
Registers for thread IDs 2, 6, 10, …
Registers for thread IDs 3, 7, 11, …
Memory Subsystem

Can you move any thread flexibly to any lane?

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

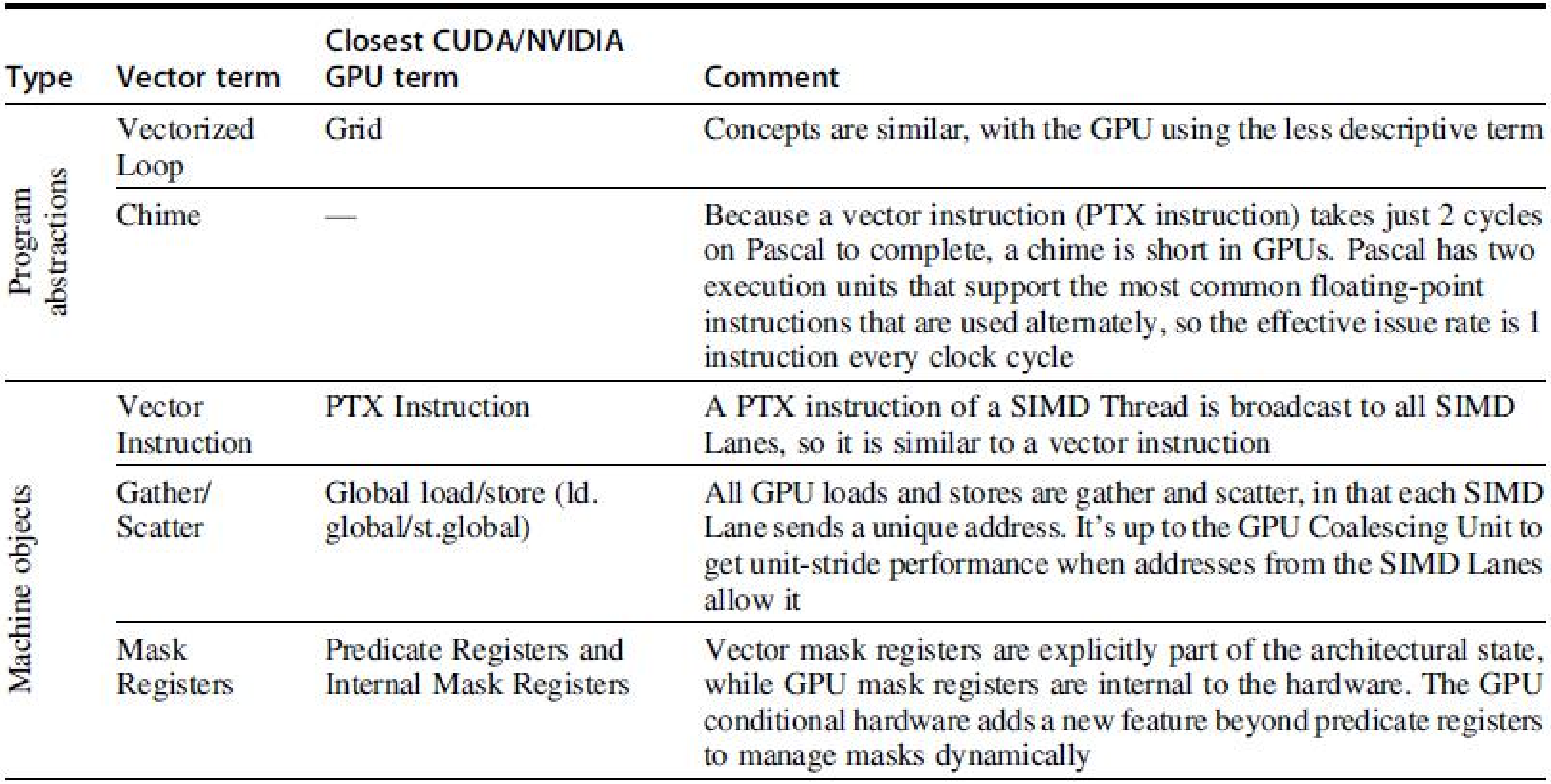
What About Memory Divergence?

- 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

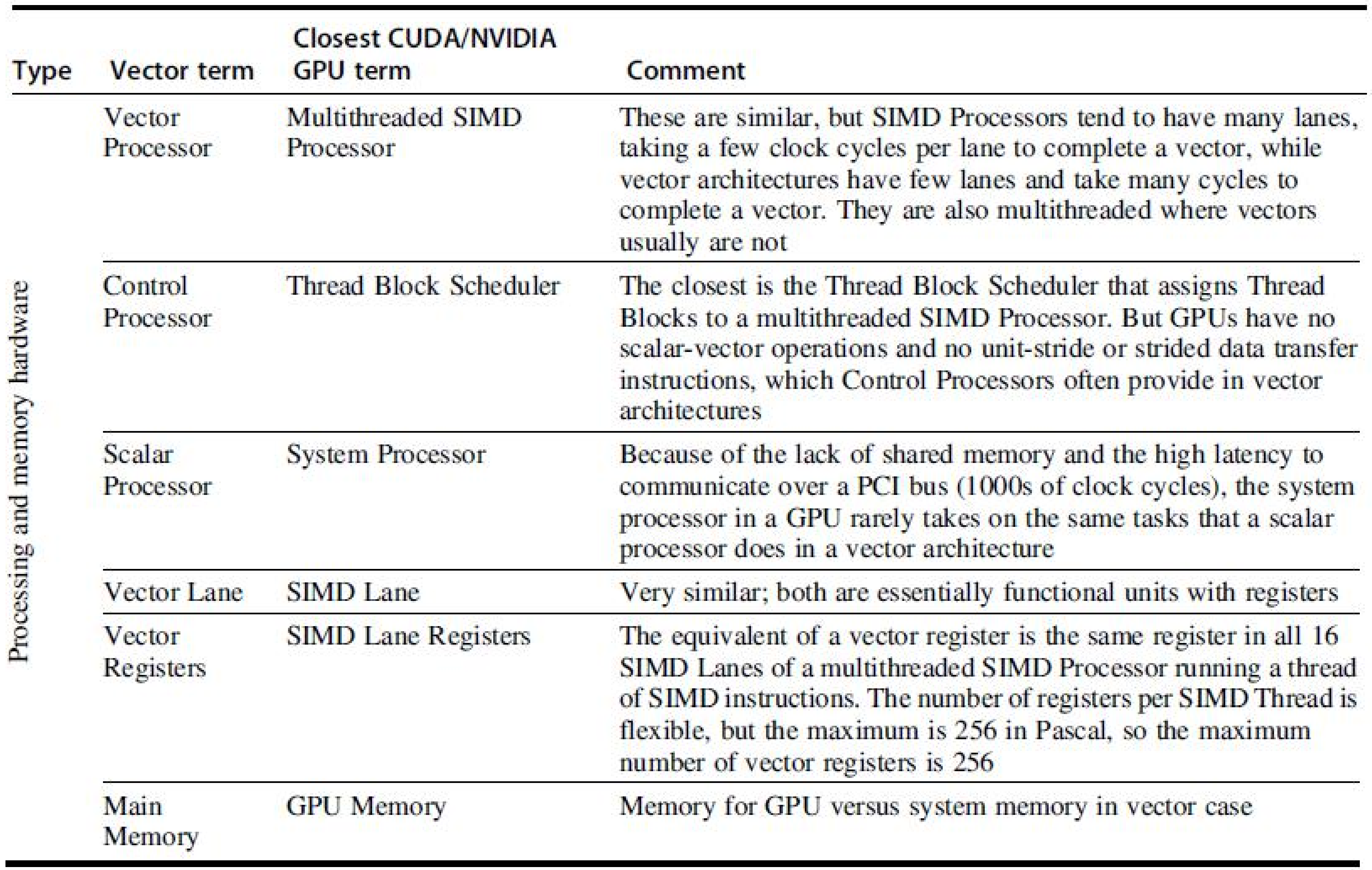
Summary- 向量处理机 vs. GPU

- **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

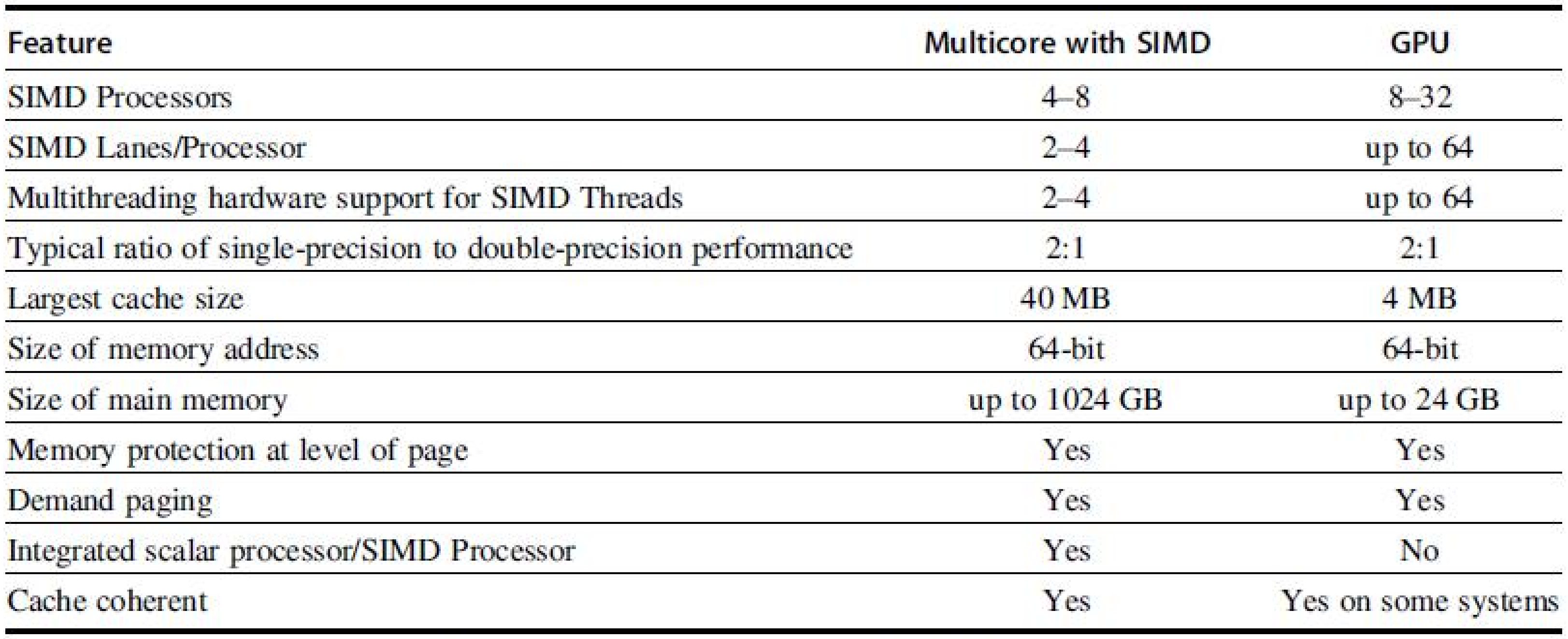
Summary-向量处理机 vs. GPU



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

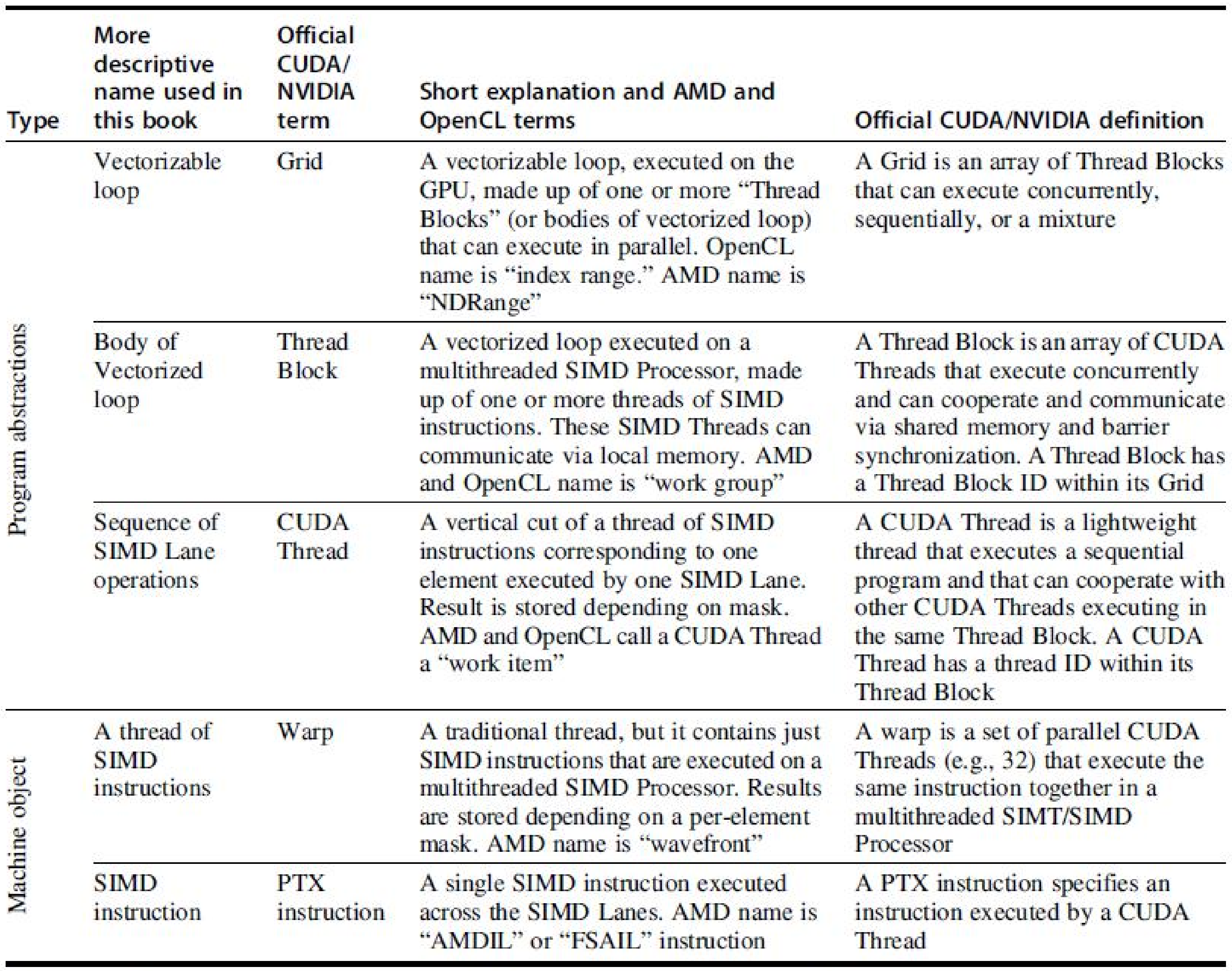
Figure 4.21 GPU equivalent to vector terms.

Summary-Multimedia SIMD Computers and GPUs



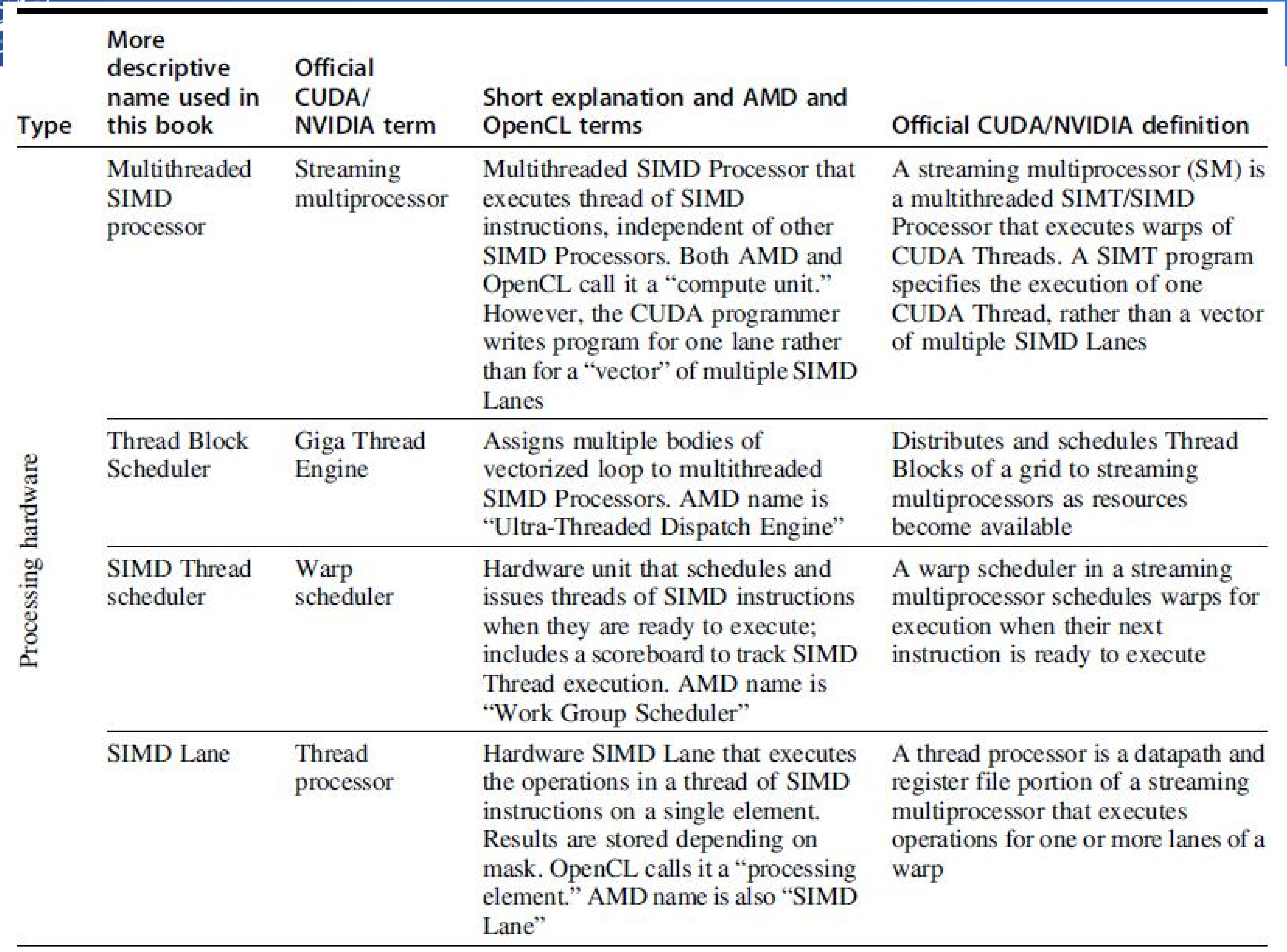
Feature	Multicore with SIMD	GPU
SIMD Processors	4–8	8–32
SIMD Lanes/Processor	2–4	up to 64
Multithreading hardware support for SIMD Threads	2–4	up to 64
Typical ratio of single-precision to double-precision performance	2:1	2:1
Largest cache size	40 MB	4 MB
Size of memory address	64-bit	64-bit
Size of 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.

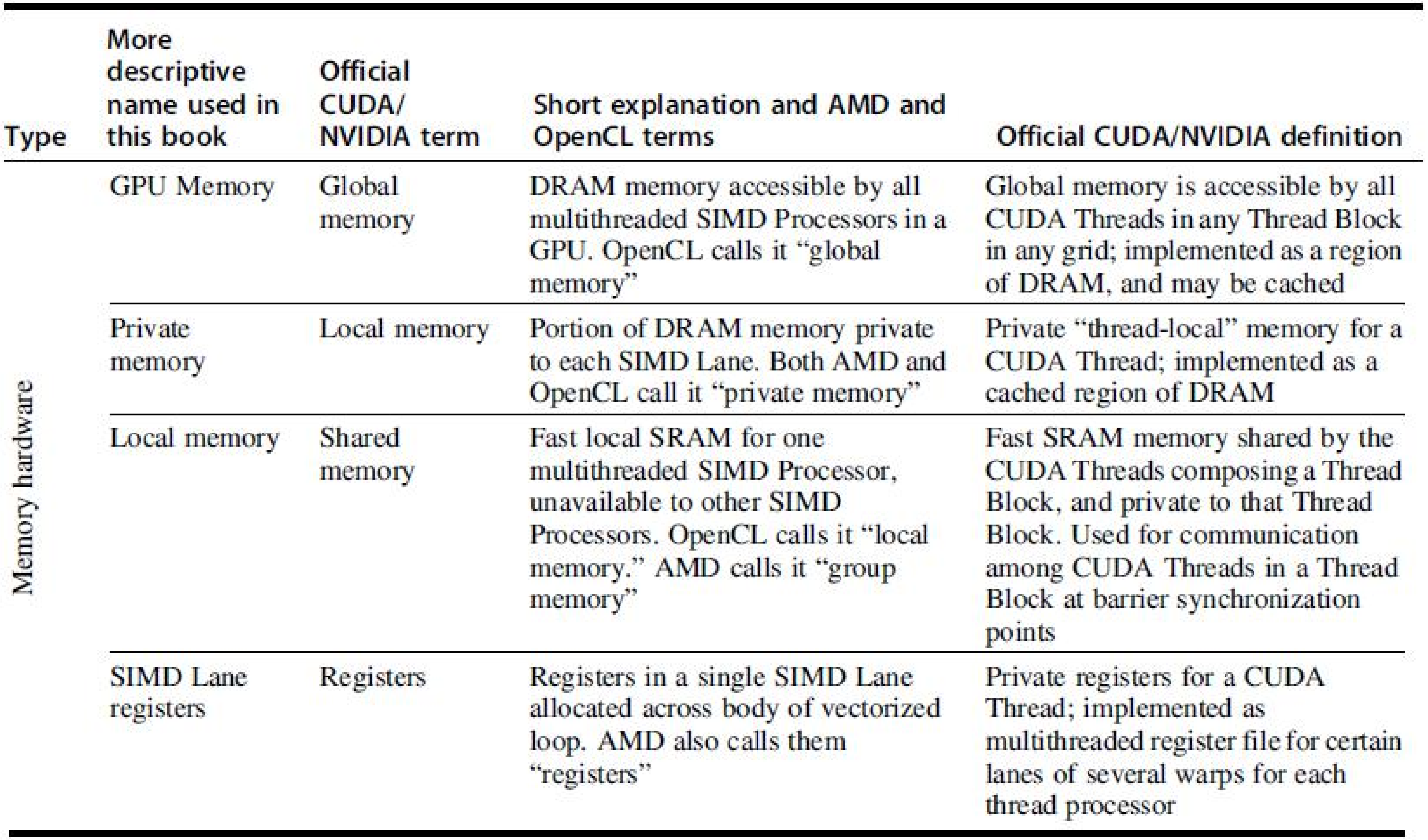


Type	More descriptive name used in this book	Official CUDA/ NVIDIA term	Short explanation and AMD and OpenCL terms	Official CUDA/NVIDIA definition
Program abstractions	Vectorizable loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more "Thread Blocks" (or bodies of vectorized loop) 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**
- **http://www.ece.cmu.edu/~ece447 CMU Introduction to Computer Architecture**
- **https://www.sdsc.edu/Events/training/webinars/gpu_computing_and_programming_2019/sdsc-gpu-webinar-goetz-2019-04-09.pdf**
- **http://www.haifux.org/lectures/267/Introduction-to-GPUs.pdf**
- **https://courses.cs.washington.edu/courses/cse471/13sp/lectures/GPUsStudents.pdf**
- **http://meseec.ce.rit.edu/551-projects/spring2015/3-2.pdf**