计算机体系结构

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

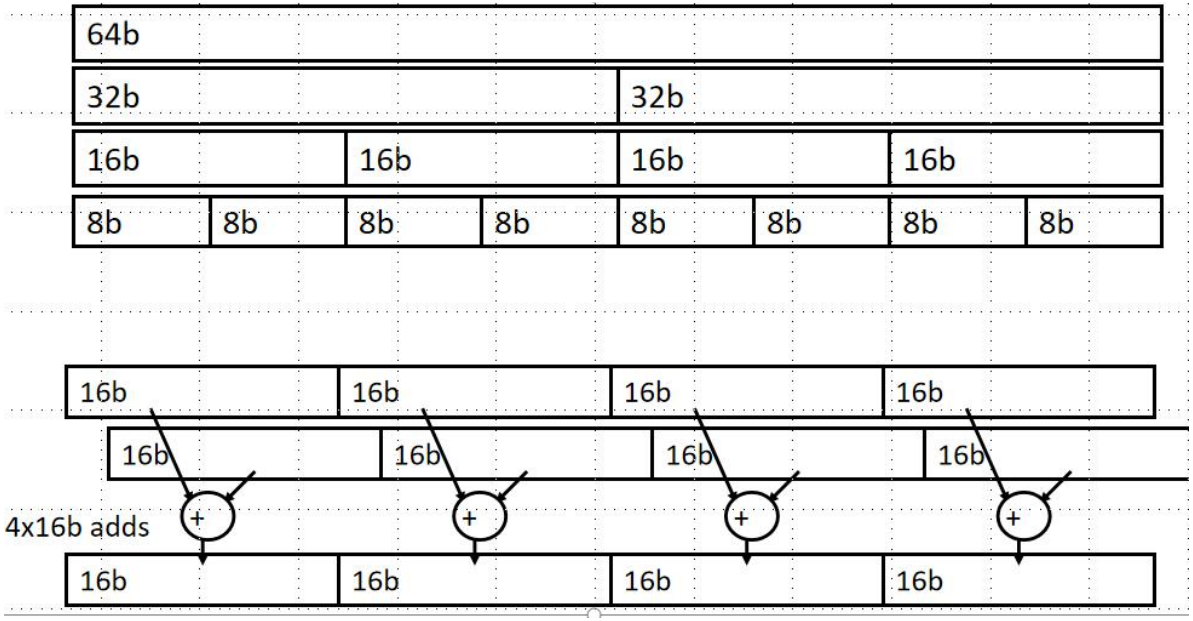
Review

- **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

Review: SIMD extensions

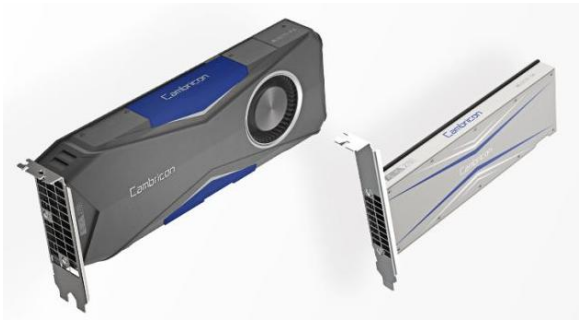
- **在已有的ISA中添加一些向量长度很短的向量操作指令**
- **将已有的 64-bit 寄存器拆分为 2x32b or 4x16b or 8x8b**
  - 1957年，Lincoln Labs TX-2 将36bit datapath 拆分为2x18b or 4x9b
  - 新的设计具有较宽的寄存器
    - 128b for PowerPC Altivec, Intel SSE2/3/4 (Sreaming SIMD Extensions)
    - 256b for Intel AVX (Advanced Vector Extensions)
- **单条指令可实现寄存器中所有向量元素的操作**



6.3 GPU I

GPU?

- **加速器**
  - 加速某一领域特定计算负载的专用的硬件模组
    - 早期用于浮点运算加速的浮点运算协处理器
    - 加速规整性计算负载的FPGA
  - 机器学习加速器
    - 寒武纪：思源270，思源100
    - 华为：昇腾系列 AI处理器
    - Google：TPU
    - ....




Tseng Labs ET4000/W32p 1991


Voodoo3 2000 AGP card 1999

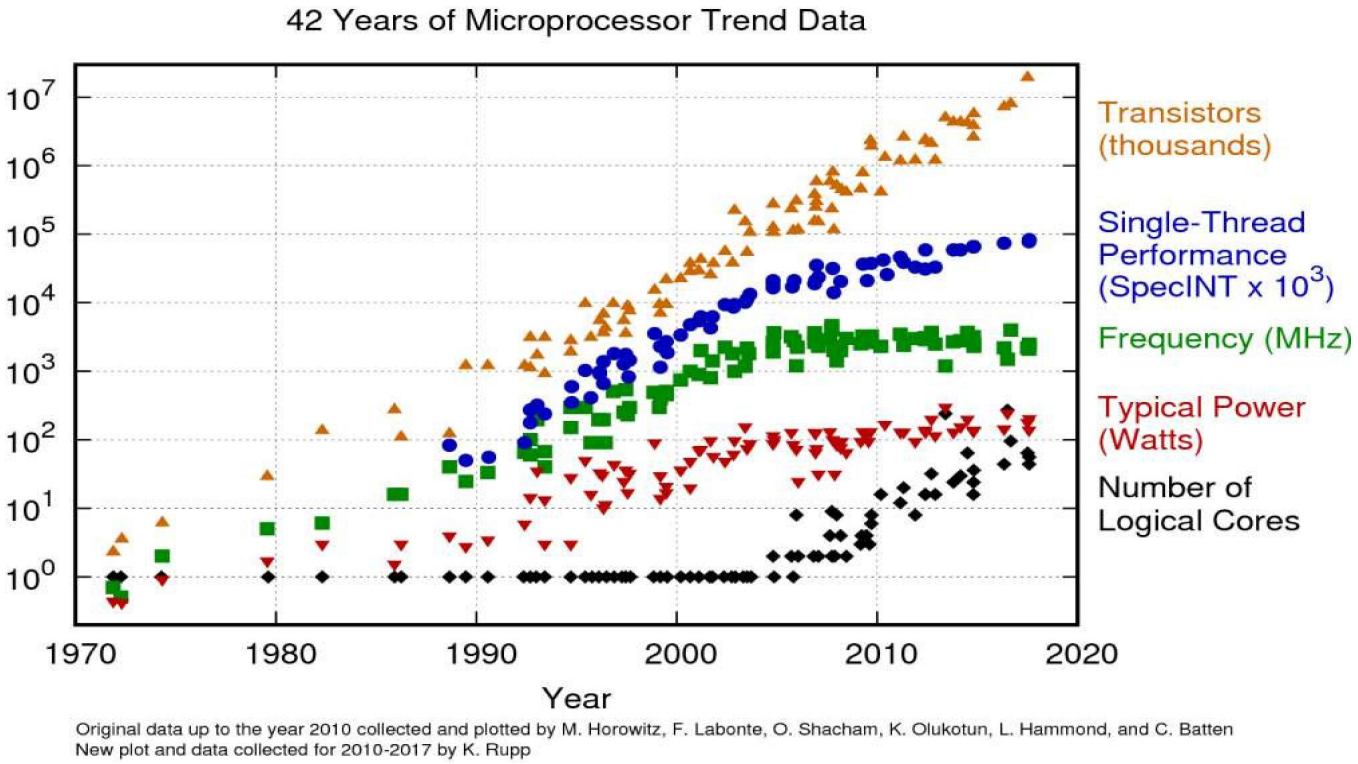

GeForce 6600 GT Personal Cinema 2004


NVIDIA GeForce GTX 280 2008

GPU起源

- **源于计算机图形学。1960年在Ivan Sutherland的Sketchpad项目中首先提到计算机图形学**
  - 早期，计算机图形学用于电影动画离线渲染，以及游戏中的实时渲染。
- **硬件支持渲染。显卡始于1981年 IBM Monochrome Display Adapter (MDA)，后来支持 2D, 3D加速；**
- **早期的3D加速卡（Graphics Processing Unit）：用来渲染图形的专用处理器**
  - 如1999年 NVIDIA GeForce 256 功能相对固定
  - 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)
  - 用户可以配置图形处理流水线，可编程性较弱
- **GPU的可编程性**
  - 2001年NVIDIA改善了GPU的可编程性（GeForce3）
  - 2003年 学术界研究将矩阵数据转换为纹理数据并使用shaders完成运算，从而在GPU上实现通用计算，这些努力激发了GPU制造商直接支持通用计算。
  - 第一个支持通用计算的GPU是GeForce 8 系列
  - Fermi architecture：具有缓存读写数据的能力
  - AMD的混合结构（GPU和CPU集成在一个芯片中，支持thread的动态调度）
  - Volta architecture：引入Tensor Cores 可有效加速机器学习应用
- **现代GPU(GP-GPU)是可编程的，可用于通用计算**

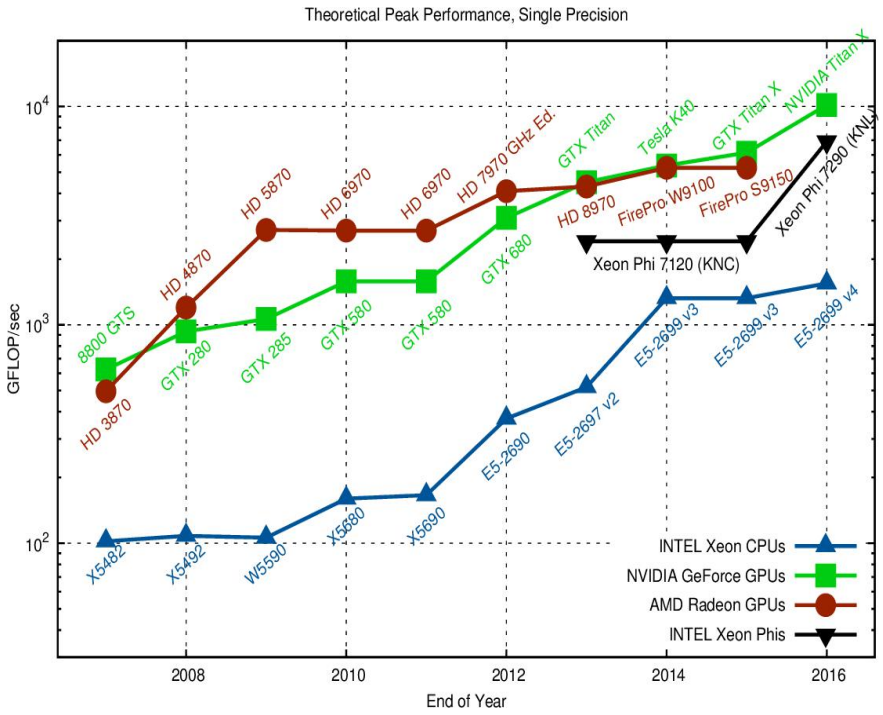
GPU得到普遍关注，why?

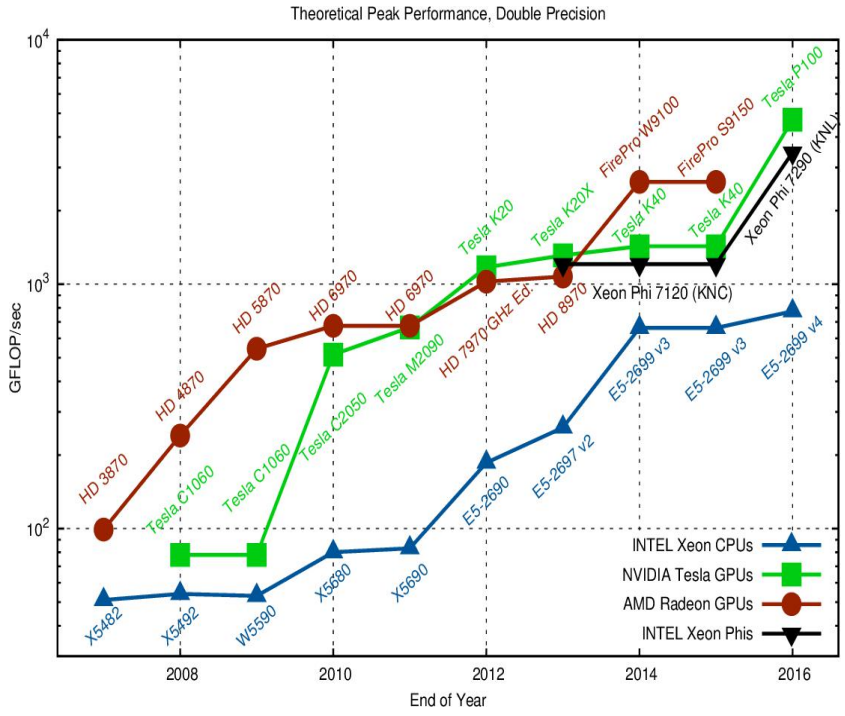


- **Moore's law：早期:IC中晶体管的数量每两年2X；目前：趋势放缓**
- **21世纪以来：时钟频率、单核的性能增加有限；性能提升主要依赖于单片上的“core”的数量；必须设计并行执行的代码**

Source: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

GPU得到普遍关注，why?

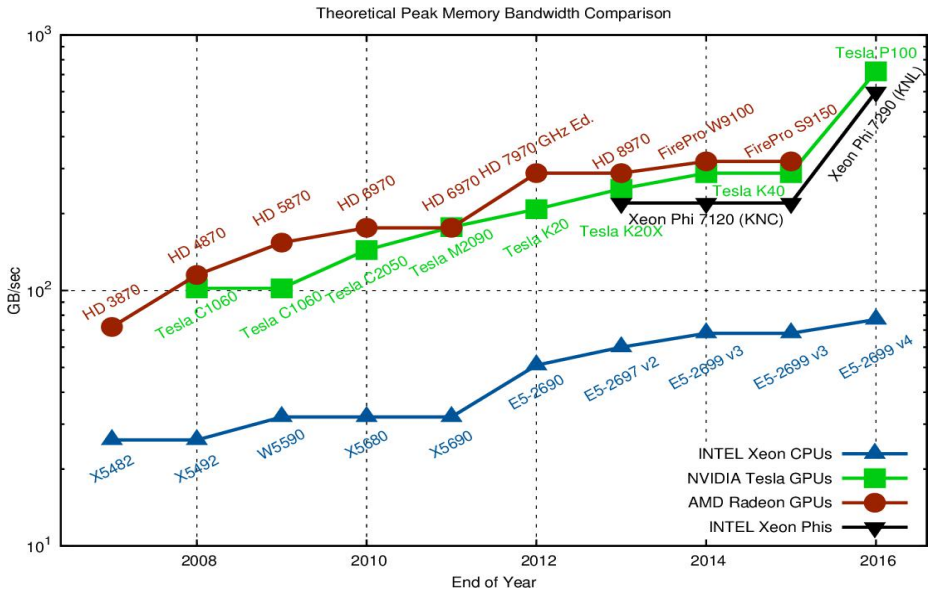


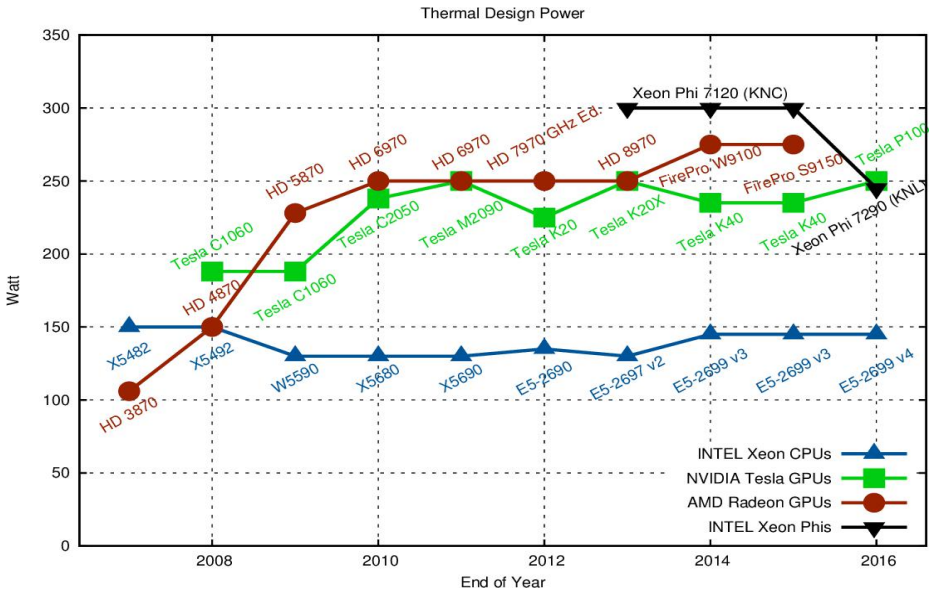


与cpu相比，gpu提供了更高的32位浮点数性能

数据中心GPU还提供比cpu更高的64位浮点性能

Figures source: https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/





GPU的访存带宽明显高于cpu

功耗的公平比较应该是：一个单独的GPU与2-socket CPU服务器比较

Figures source: https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

X86 Intel8180 vs Nvidia V100 GPU

Aggregate performance numbers (FLOPs, BW)	**Dual socket Intel 8180 28-core (56 cores per node)**	**Nvidia Tesla V100, dual cards in an x86 server**
Peak DP FLOPs	4 TFLOPs	14 TFLOPs (3.5x)
Peak SP FLOPs	8 TFLOPs	28 TFLOPs (3.5x)
Peak HP FLOPs	N/A	224 TFLOPs
Peak RAM BW	~ 200 GB/sec	~ 1,800 GB/sec (9x)
Peak PCIe BW	N/A	32 GB/sec
Power / Heat	~ 400 W	2 x 250 W (+ ~ 400 W for server) (~ 2.25x)
Code portable?	Yes	Yes (OpenACC, OpenCL)

A supercomputer in a desktop?

ASCI White (LLNL)

- 12.3 TFLOP/sec – #1 Top 500, November 2001.
- Cost – $110 Million USD (in 2001!)

SDSC Comet

- 2.8 PFLOP/sec aggregate
- 36 nodes 2 x Nvidia K80
  5.5 TFLOP/sec DP, 16.4 TFLOP/sec SP (each node)
- 36 nodes 4 x Nvidia P100
  18.8 TFLOP/sec DP, 37.2 TFLOP/sec SP (each node)
- Cost – $25 Million USD ($14 Million Hardware)

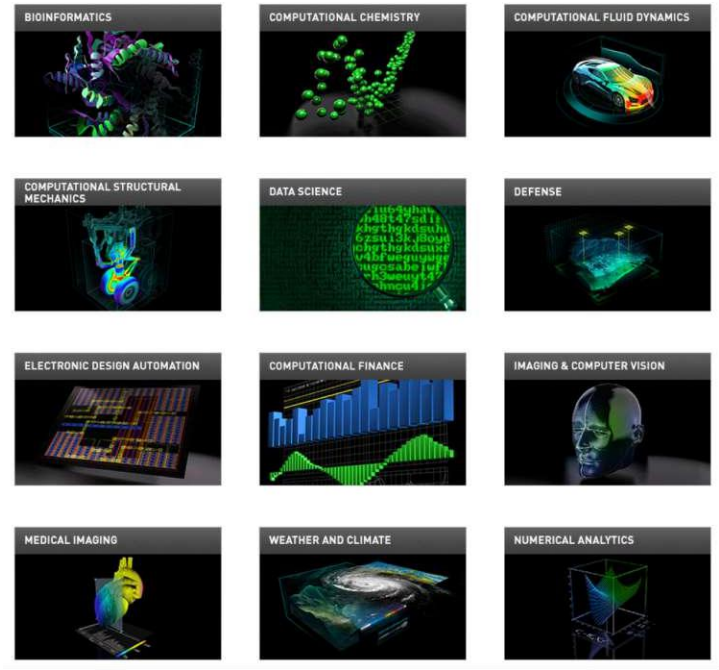
DIY 4 x Nvidia RTX 2080 box

- 1.3 TFLOP/sec DP
- 40.0 TFLOP/sec SP
- Cost – ~ $5 Thousand USD

GPU 应用领域

一些主要应用领域

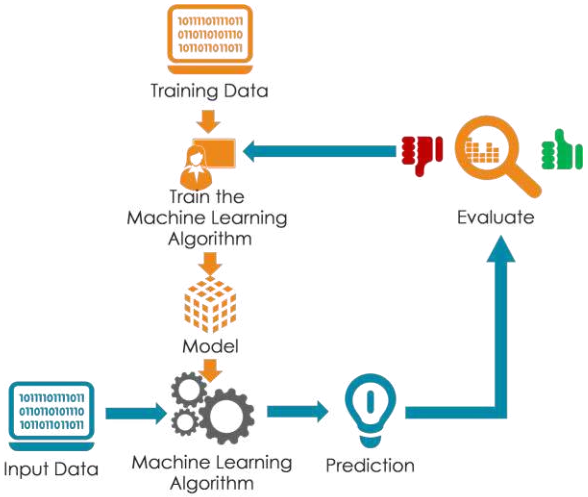
- Exhautive list on https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/
- 化学
- 生命科学
- 生物信息学
- 天体物理学
- 金融
- 医学图像
- 自然语言处理
- 社会科学
- 天气与气候
- 计算流体动力学
- **机器学习**
- etc...



Machine learning and GPUs

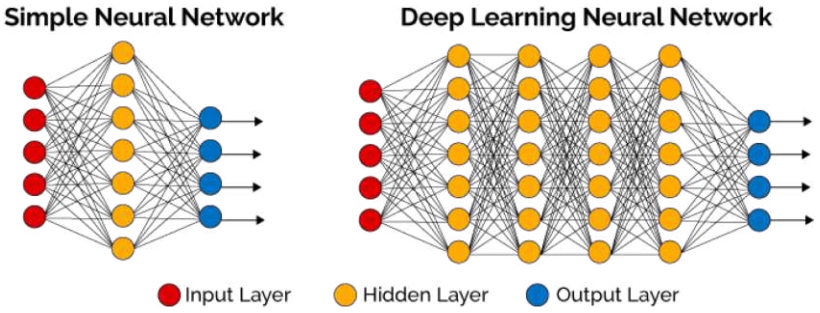
Machine learning

- Estimate / predictive model based on reference data.
- Many different methods and algorithms.
- **GPUs are particularly well suited for deep learning workloads**



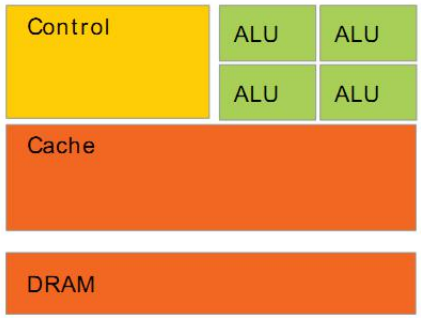
Deep learning

- Neural networks with many hidden layers.
- Tensor operations (matrix multiplications).
- GPUs are very efficient at these (4x4 matrix algebra is used in 3D graphics)
- Half-precision arithmetic can be used for many ML applications, at least for inference.
- ML frameworks provide GPU support (E.g. PyTorch, TensorFlow)

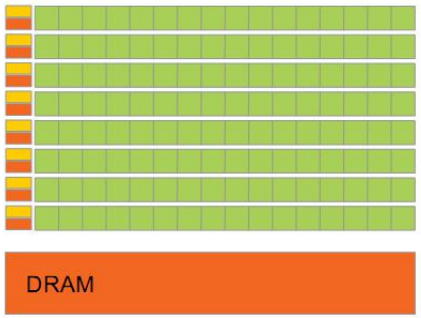


GPU基本硬件结构

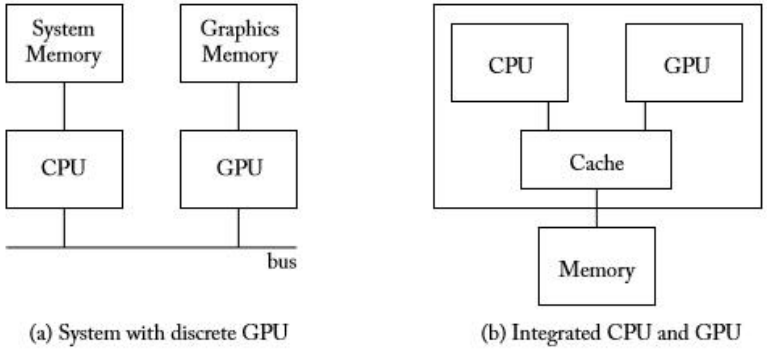
- CPU+GPU异构多核系统
  - 针对每个任务选择合适的处理器和存储器
- 通用CPU 适合执行一些串行的线程
  - 串行执行快
  - 带有cache，访问存储器延时低
- GPU 适合执行大量并行线程
  - 可扩放的并行执行
  - 高带宽的并行存取



强控制、弱计算

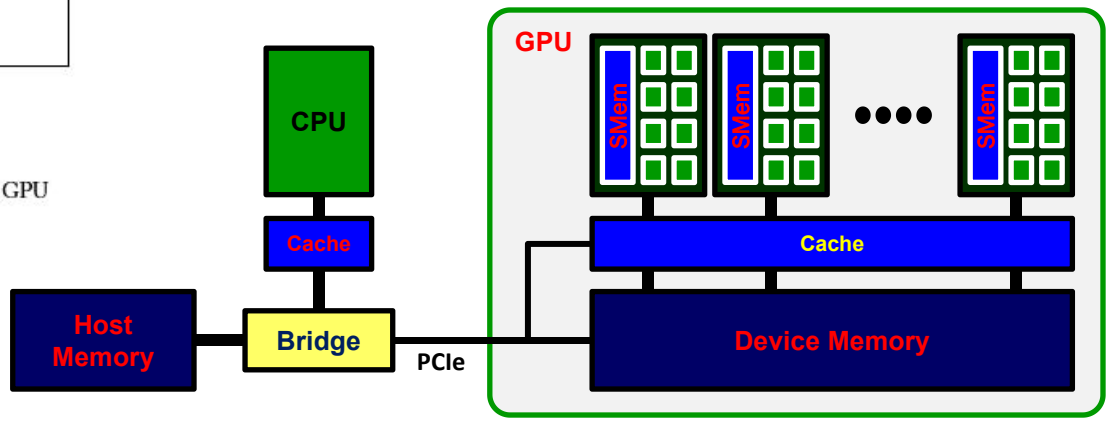


弱控制、强计算



(a) System with discrete GPU (b) Integrated CPU and GPU

Figure 1.1: GPU computing systems include CPUs.



现代GPU硬件结构

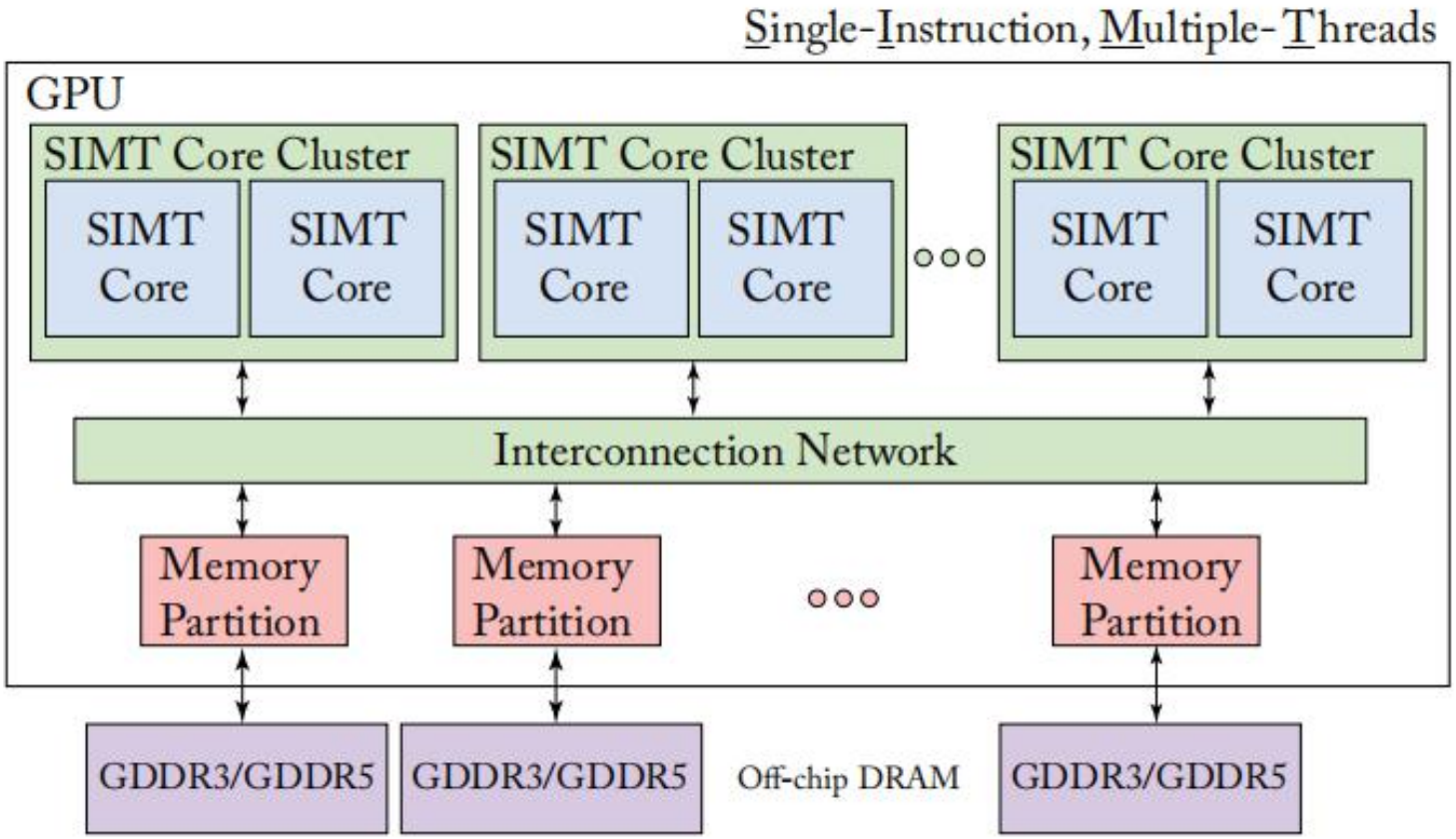


Figure 1.2: A generic modern GPU architecture.

Multicore vs. Multithreaded

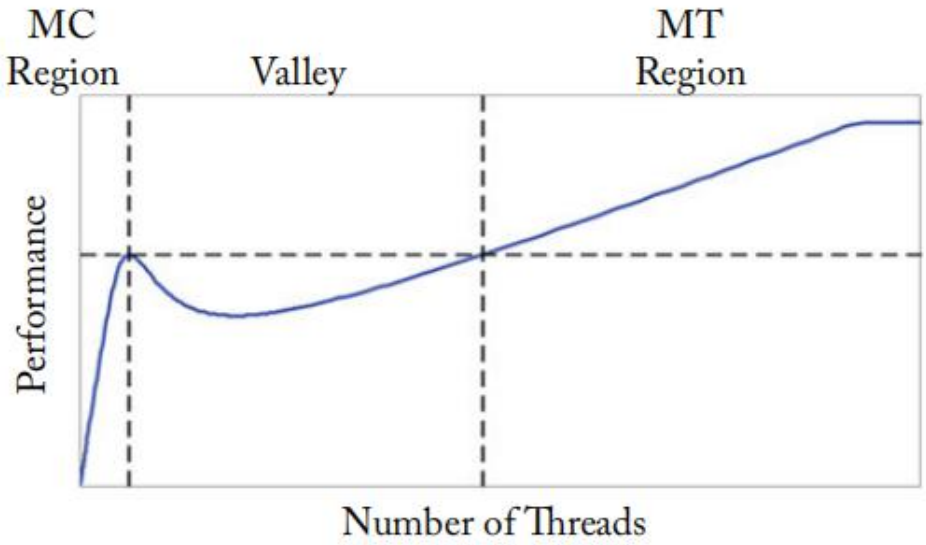


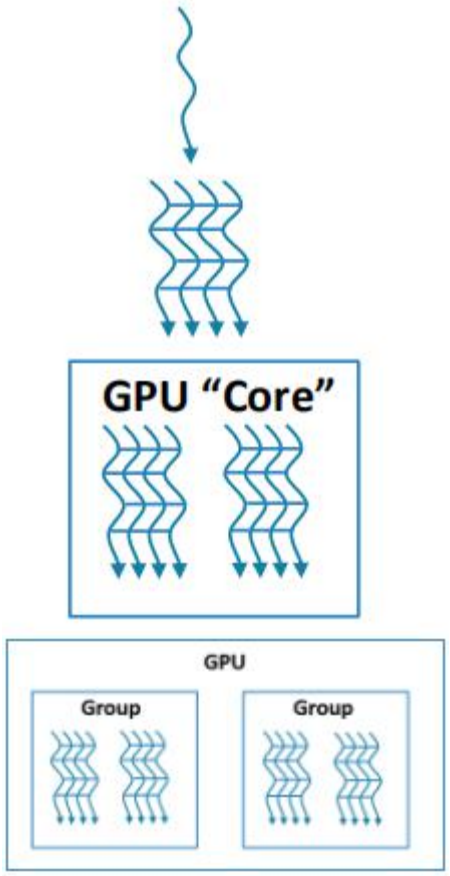
Figure 1.3: An analytical model-based analysis of the performance tradeoff between multicore (MC) CPU architectures and multithreaded (MT) architectures such as GPUs shows a "performance valley" may occur if the number of threads is insufficient to cover off-chip memory access latency (based on Figure 1 from Guz et al. [2009]).

- **多核处理器结构随着线程数的增加会落入性能低谷**
  - Cache容量不足时，性能就会下降。
- **当线程数较少时多线程结构（GPU）会落入性能低估**
  - 无法抵消对片外存储访问的延时

小结：General-Purpose GPUs (GP-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+**
  - 通过配置向量化硬件（例如GPU)，减少指令处理的额外开销，可获得10X的效率提升。
  - 通过引入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：如何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270、思源100
  - 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除了支持机器学习以外，可以支持图灵计算模型，具有更好的实用性**
  - 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）：CPU发射数据并行的kernels 到GP-GPU上运行
  - 通过优化GPU结构和编程模型，在增加其性能的同时，方便用户编程
- **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

一些术语



Nvidia/CUDA	AMD/OpenCL	Derek‘ CPU Analogy
CUDA Processor	Processing Element	Lane
CUDA Core	SIMD Unit	Pipeline
Streaming Multiprocessor	Computer Unit	Core
GPU Device	GPU Device	Device

6.3 GPU

GPUs are SIMD Engines Underneath

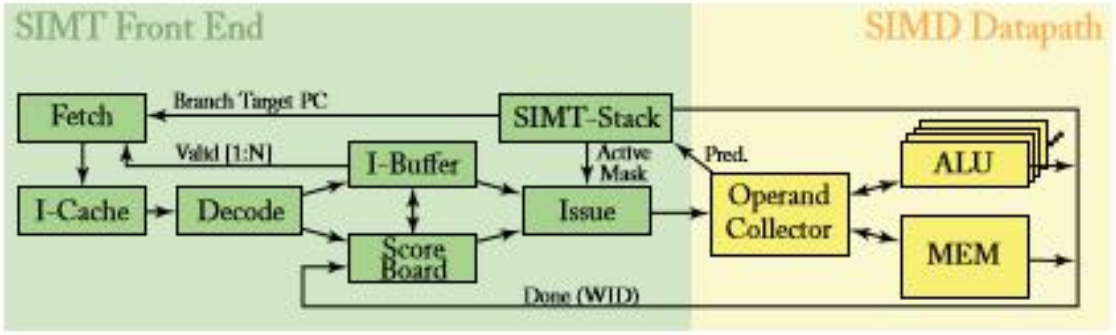


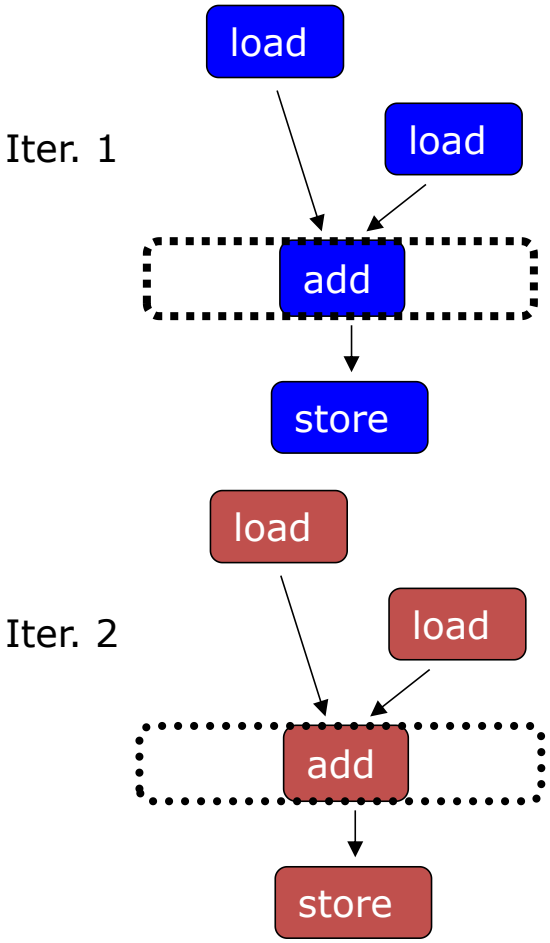
Figure 3.1: Microarchitecture of a generic GPGPU core.

- **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)

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];
```

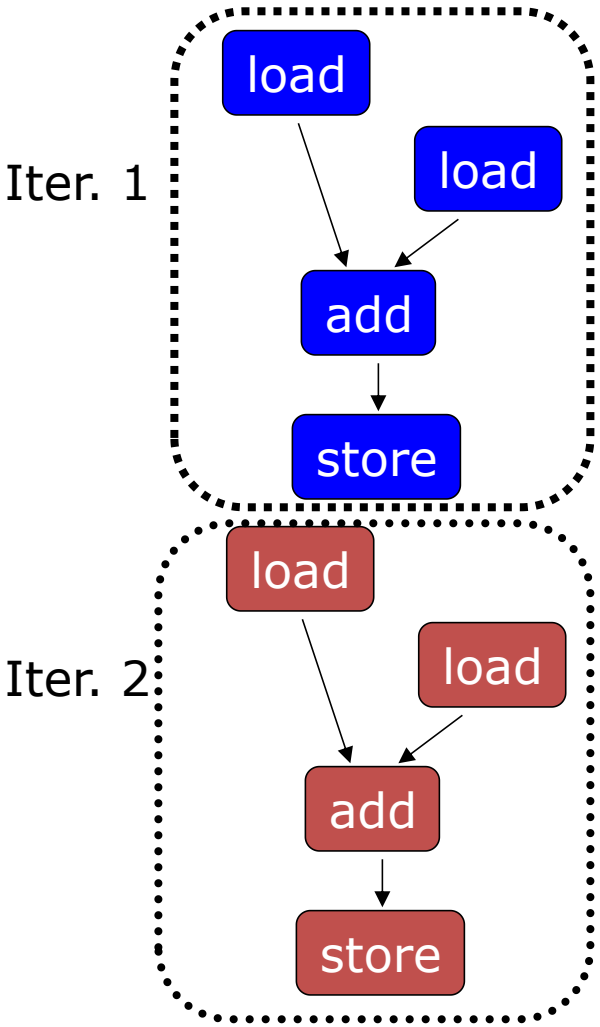


三种编程模式来挖掘程序的并行性:

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

Prog. Model 1: Sequential (SISD)

Scalar Sequential Code

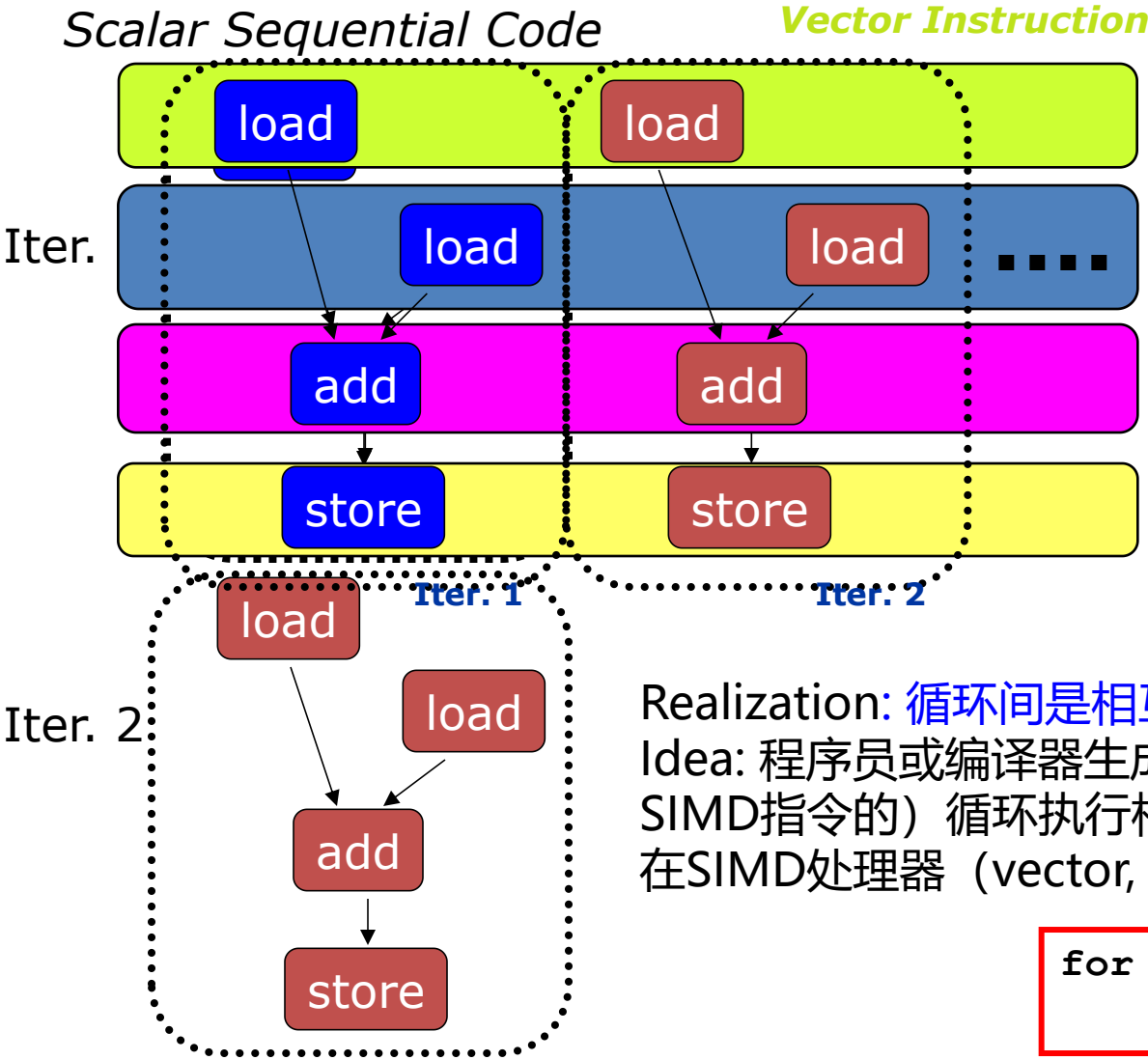


可以采用如下不同类型的处理器执行

- **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

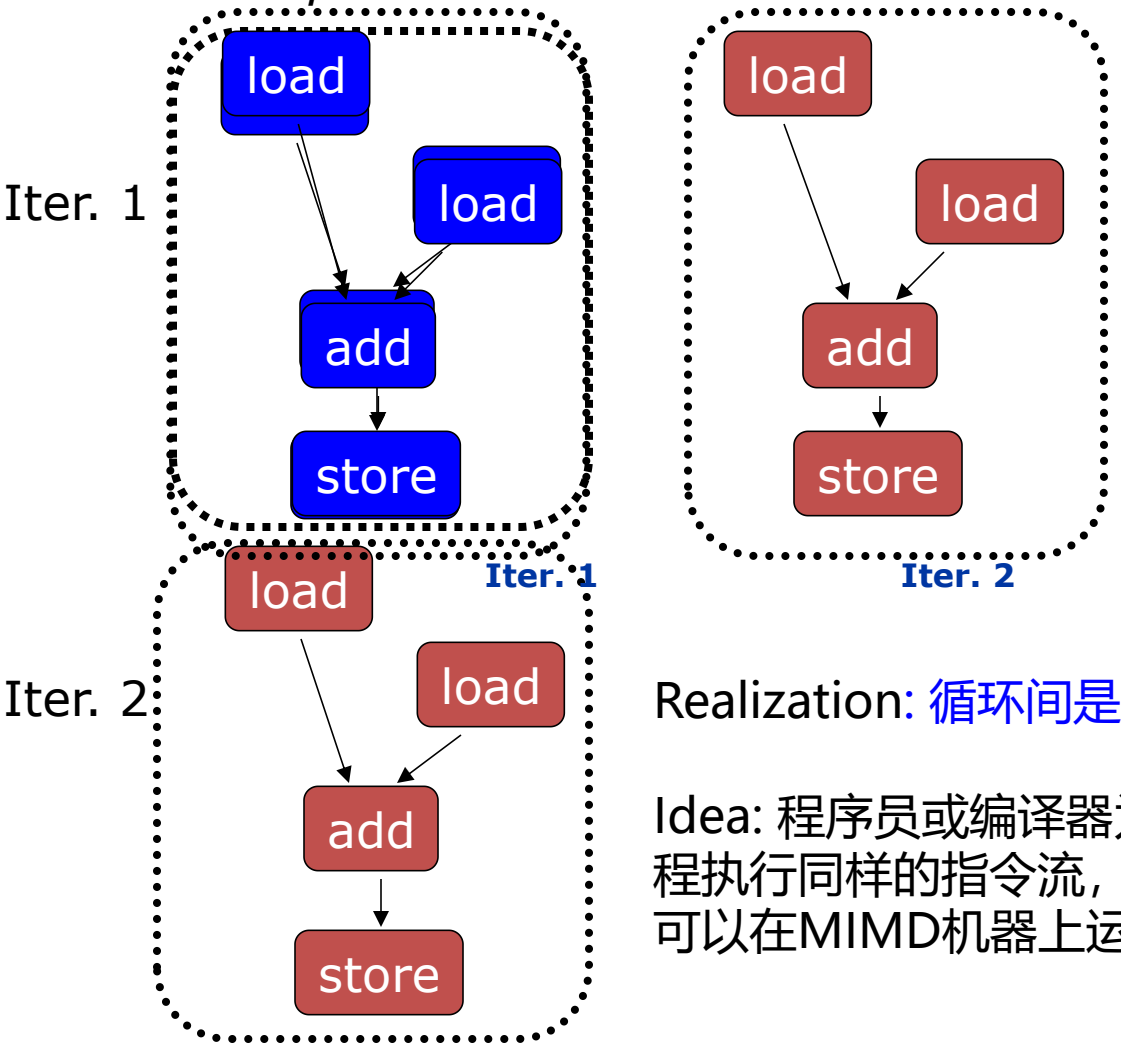
Realization: 循环间是相互独立的，循环间没有数据依赖

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

Scalar Sequential Code



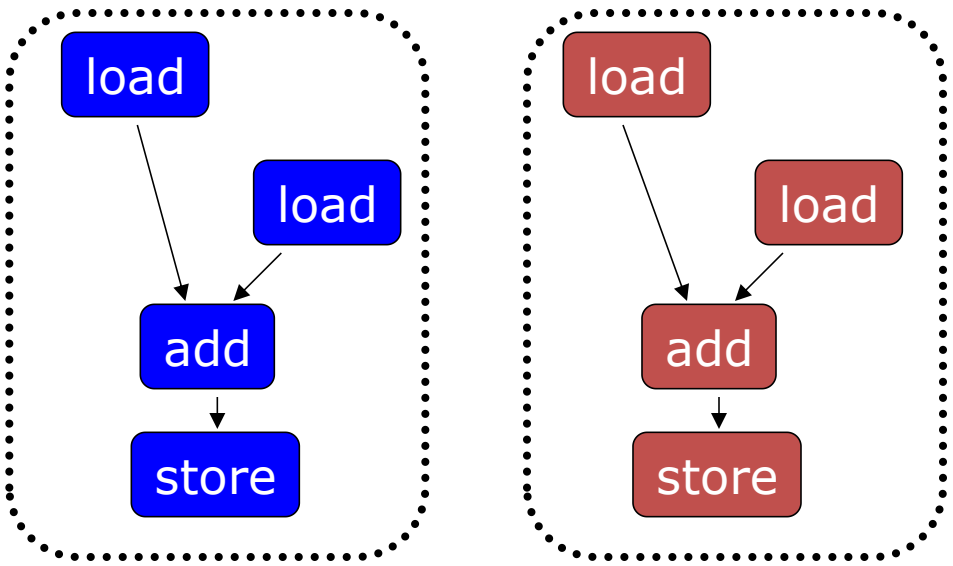

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

....

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




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

....

Reali
Idea:
行同

这种模式也称为:

SPMD: Single Program Multiple Data

可以

可以在SIMT 机器上运行
Single Instruction Multiple Thread

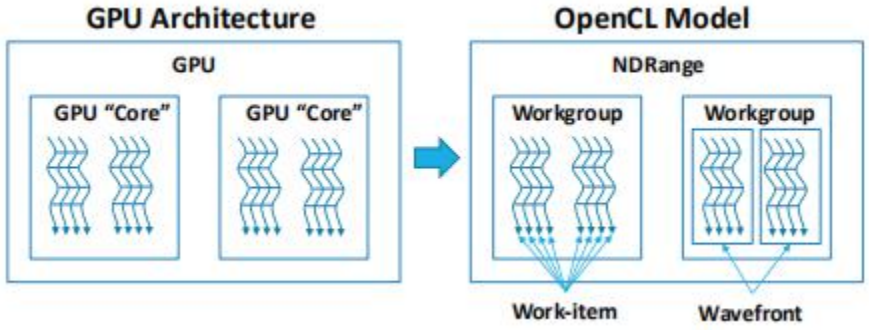
SPMD

- **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

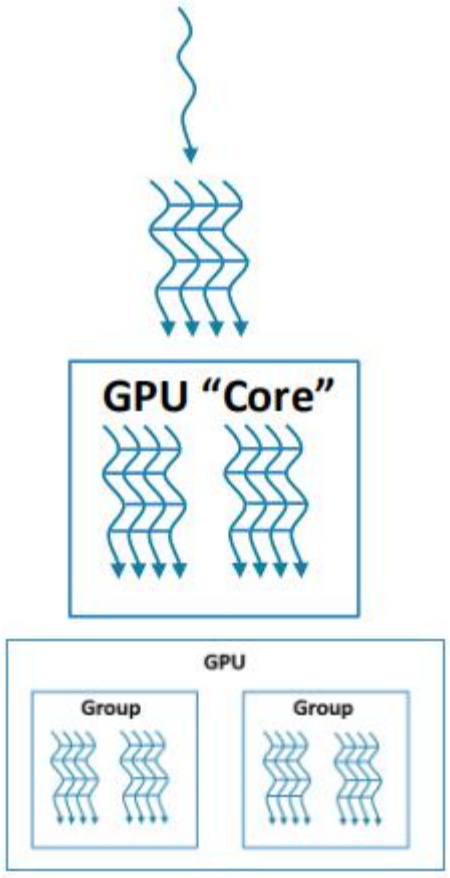
- **多条指令流执行相同的程序**
  - 每个程序/过程
    - 操作不同的数据
    - 运行时可以执行不同的控制流路径
  - 许多科学计算应用以这种方式编程，运行在MIMD硬件结构上 (multiprocessors)
  - 现代 GPUs 以这种类似的方式编程，运行在SIMD硬件上

GPU 编程模型

- **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
- **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**
  - Thread Block Scheduler
  - SIMD Thread Scheduler
  - Warp
    - SIMD线程
    - 线程调度的基本单位

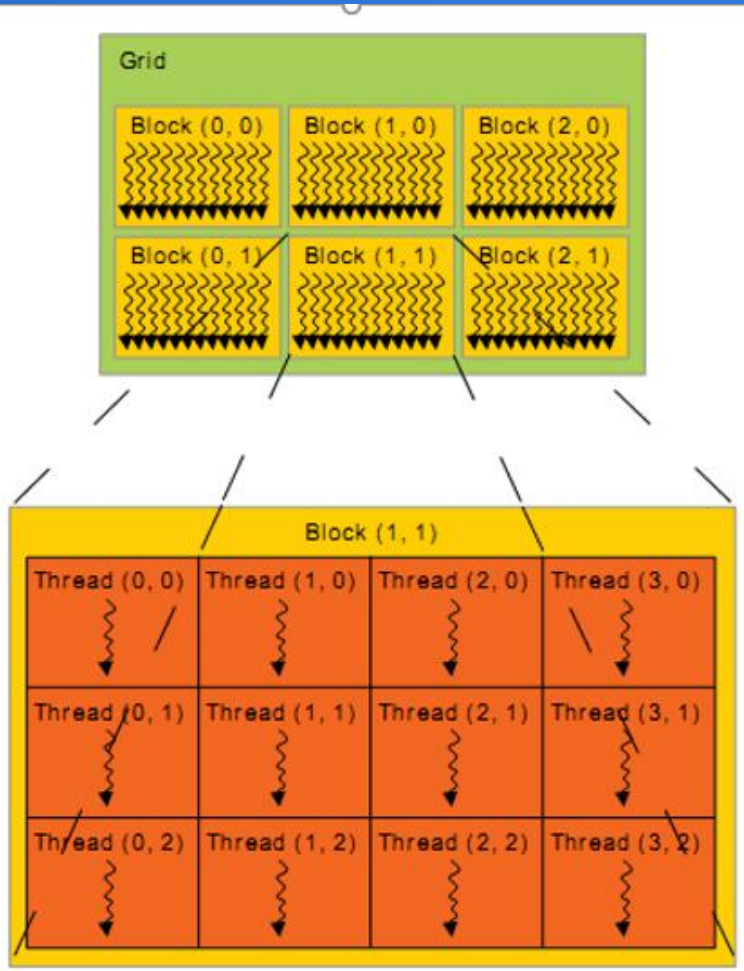
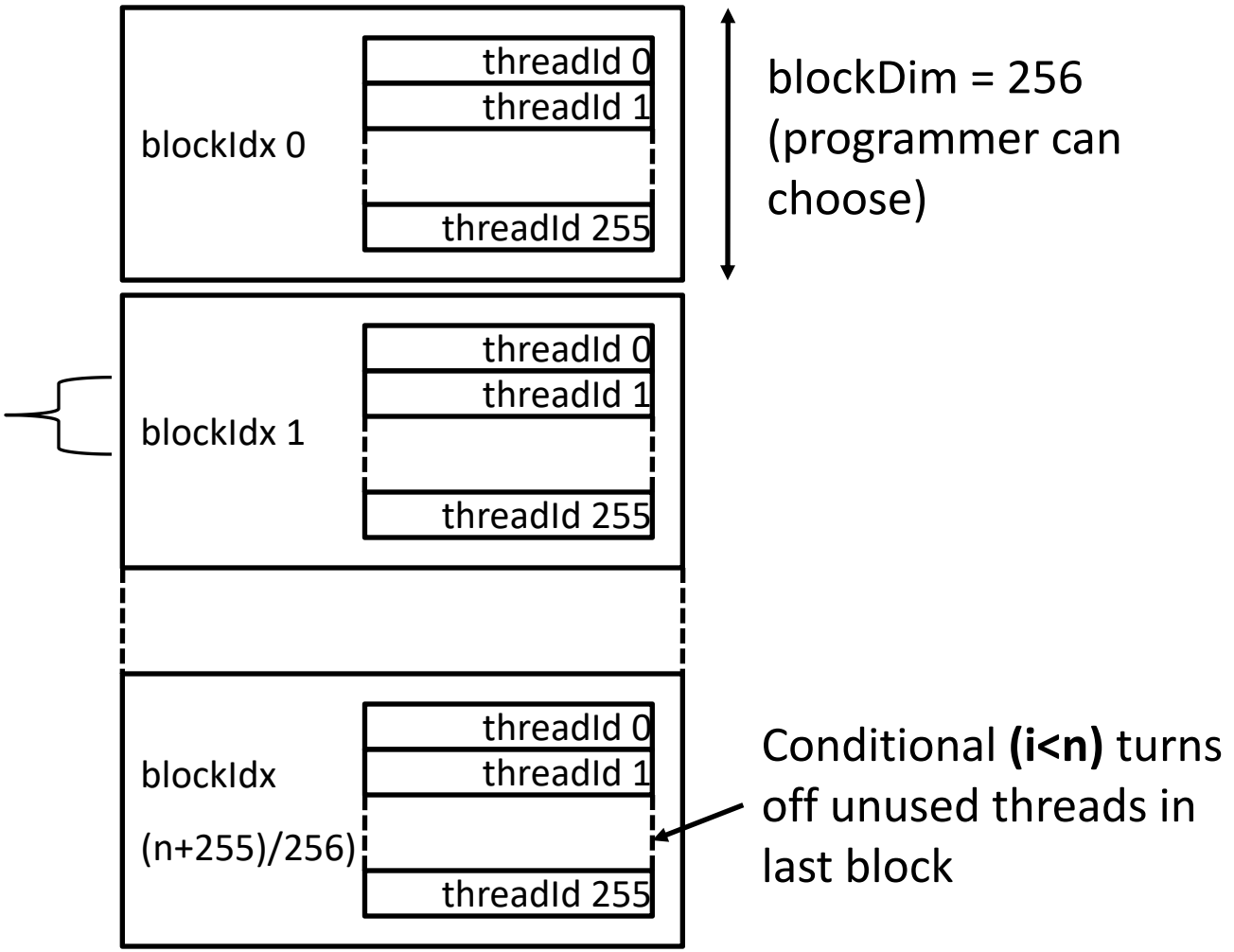


Figure 6 Grid of Thread Blocks

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*)



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 包含16个线程块（512个元素/Block）
Block包含16个SIMD 线程
32个CUDA 线程/SIMD线程
处理一个元素/CUDA线程

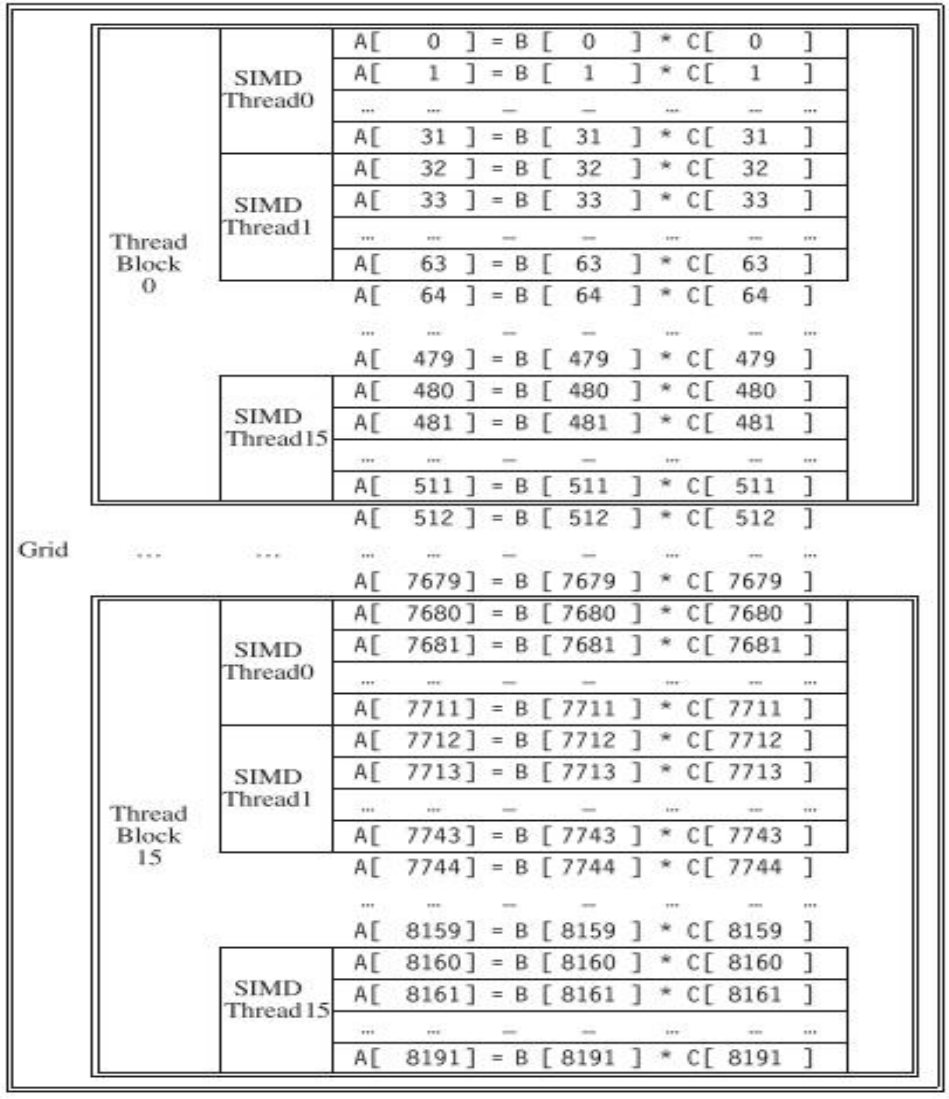
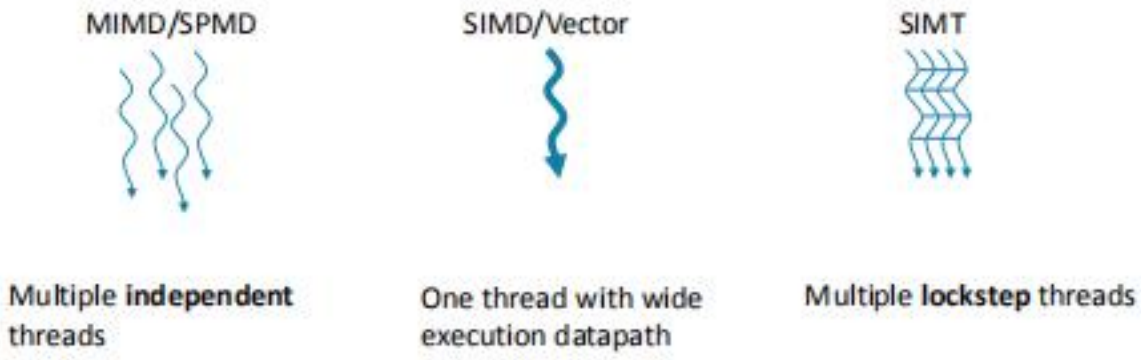


Figure 4.13 The mapping of a Grid (vectorizable loop), Thread Blocks (SIMD basic blocks), and threads of SIMD instructions to a vector-vector multiply, with each vector being 8192 elements long. Each thread of SIMD instructions calculates 32 elements per instruction, and in this example, each Thread 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.)

6.3 GPU

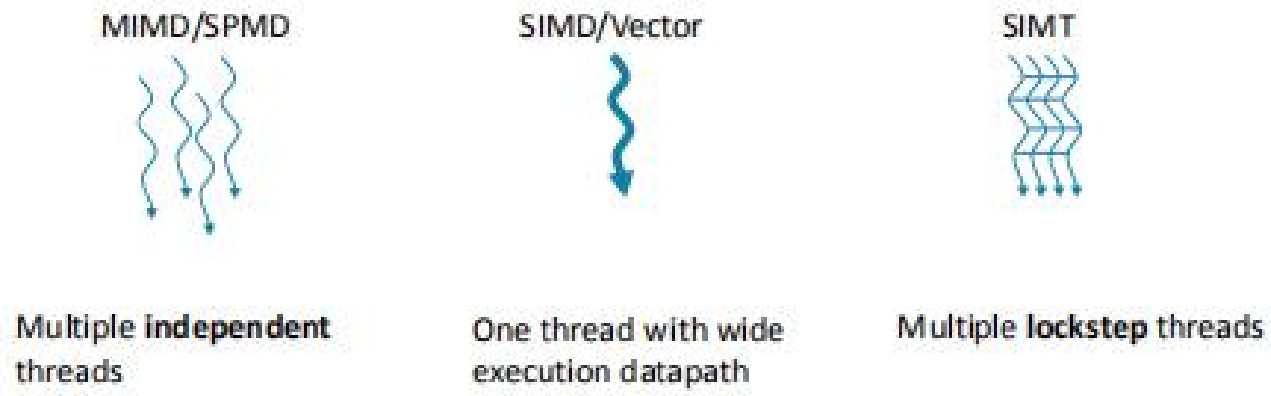
A GPU is a SIMD (SIMT) Machine

- **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



数据并行不同的执行模式

数据并行不同的执行模式比较

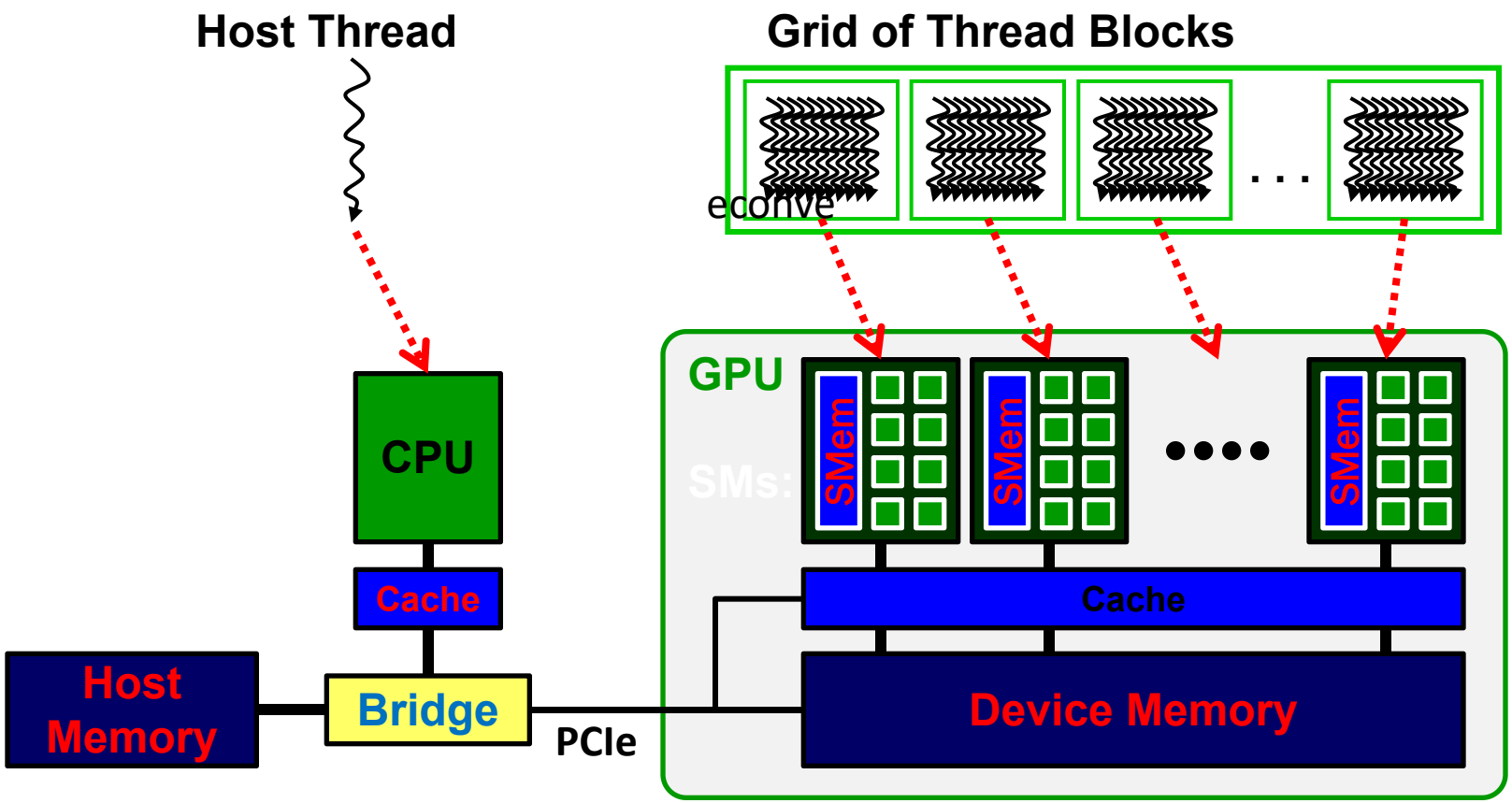


Example Architecture	Multicore CPUs	x86 SSE/AVX	GPUs
Pros	适用性强：支持线程级(任务级）并行	可以串-并行代码	容易编程进行 Gather/Scatter 操作
Cons	数据并行效率较低	不适合Gather/Scatter 操作	分支对性能影响明显

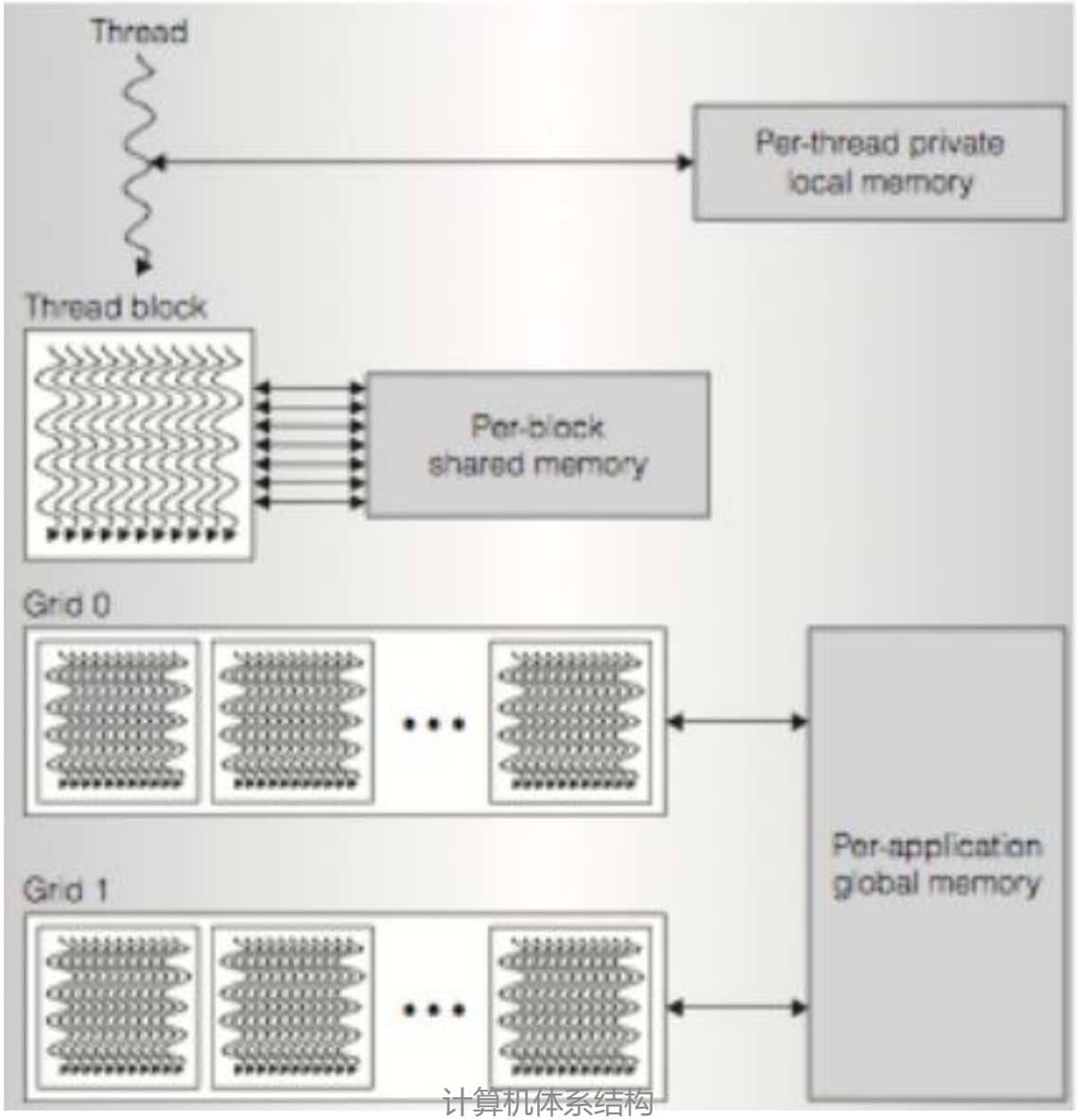
6.3 GPU

CUDA kernel maps to Grid of Blocks

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



GPU Memory Hierarchy



[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.

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**