题型： 概念题，简答题，程序设计

《CUDA_C_Programming_Guide》

《课件-01-CUDA-C-Basics》

Page 2 WHAT IS CUDA? （CUDA概念）

CUDA架构

- Expose GPU parallelism for general-purpose computing

  一种由NVIDIA推出的**通用并行计算架构**，该架构**利用**GPU的并行性解决复杂的计算问题

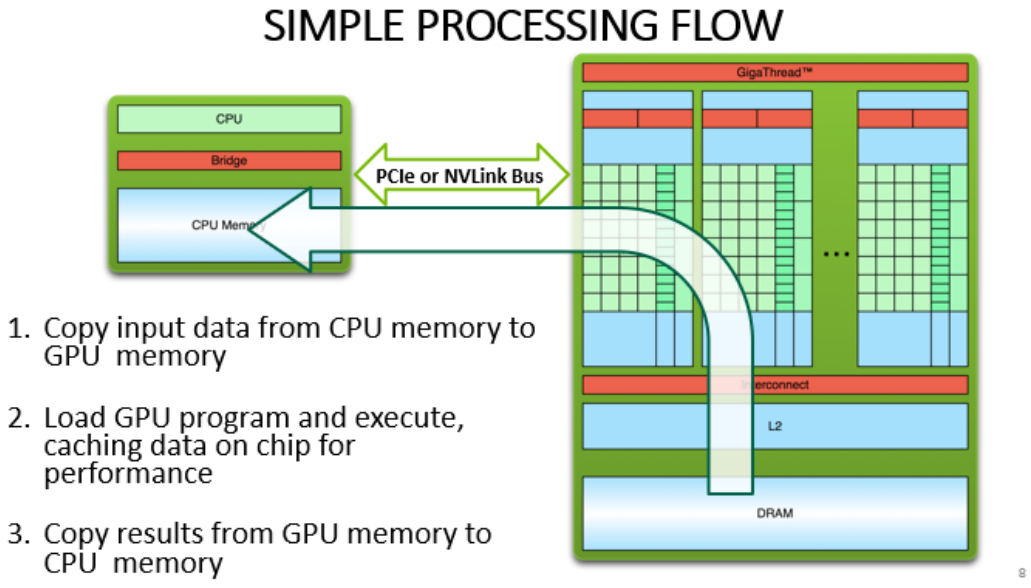
- Expose/Enable performance

  公开/启用性能

CUDA C/C++

- Based on industry-standard C/C++
- Small set of extensions to enable heterogeneous programming
- Straightforward APIs to manage devices, memory etc.

Page 8 SIMPLE PROCESSING FLOW （简单的处理过程）



1. 将输入数据从 CPU 内存复制到 GPU 内存
2. 加载 GPU 程序并执行，在芯片上缓存数据以提高性能
3. 将结果从 GPU 内存复制到 CPU 内存

Page 10-11 GPU KERNELS: DEVICE CODE

GPU KERNELS: DEVICE CODE

```
__global__ void mykernel(void) {
}
```

- CUDA C++ keyword `__global__` indicates a defined function that:
  - Runs on the device
  - Is called from host code (can also be called from other device code)
- `nvcc` separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler:
    - `gcc, cl.exe`

10

`__global__` 关键字修饰的函数：

- 运行在设备上
- 由主机调用

nvcc编译器会将源代码部署在主机和设备的器件上

- 设备的函数会被NVIDIA的编译器处理
- 主机函数会被主机编译器处理

GPU KERNELS: DEVICE CODE

```
mykernel<<<1,1>>>(); // run a kernel on GPU
```

- Triple angle brackets mark a call to *device* code
  - Also called a “kernel launch”
  - the number of CUDA threads that execute that kernel for a given kernel call is specified using a new <<<…>>> execution configuration syntax
  - Each thread that executes the kernel is given a unique thread ID (threadIdx) that is accessible within the kernel through built-in variables.
  - The parameters inside the triple angle brackets are the CUDA kernel **execution configuration**
- That’s all that is required to execute a function on the GPU!

11

三重尖括号标记对设备代码的调用

- 也称为“kernel launch”

- 为给定内核调用执行该内核的 CUDA 线程数使用新的 <<<...>>> 执行配置语法指定
- 执行内核的每个线程都被赋予一个唯一的线程 ID (threadIdx)，可通过内置变量在内核中访问该 ID。
- 三尖括号内的参数是CUDA内核执行配置

Page 12-16 RUNNING CODE IN PARALLEL，VECTOR ADDITION ON THE DEVICE

MEMORY MANAGEMENT

- Host and device memory are separate entities
- Device pointers point to GPU memory
  - Typically passed to device code
  - Typically not dereferenced in host code
- Host pointers point to CPU memory
  - Typically not passed to device code
  - Typically not dereferenced in device code

- Simple CUDA API for handling device memory
  - cudaMalloc(), cudaFree(), cudaMemcpy()
  - Similar to the C equivalents malloc(), free(), memcpy()

- 主机和设备内存各自分开的
- 设备的指针指向 GPU 内存
  - 通常传递给设备的kernel函数
  - 通常不在主机代码中解引用
- 主机的指针指向 CPU 内存
  - 通常不传递给设备代码
  - 通常不会在设备代码中解引用
- 用于处理设备内存的简单 CUDA API
  - cudaMalloc()、cudaFree()、cudaMemcpy() 类似于 C 中的 malloc()、free()、memcpy()

RUNNING CODE IN PARALLEL

- GPU computing is about massive parallelism
  - So how do we run code in parallel on the device?

```
add<<< 1, 1 >>>();
```



```
add<<< N, 1 >>>();
```

- Instead of executing `add()` once, execute N times in parallel

VECTOR ADDITION ON THE DEVICE

- With `add()` running in parallel we can do vector addition
- Terminology: each parallel invocation of `add()` is referred to as a **block**
  - The set of all blocks is referred to as a **grid**
  - Each invocation can refer to its block index using `blockIdx.x`

```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- By using `blockIdx.x` to index into the array, each block handles a different index
- Built-in variables like `blockIdx.x` are zero-indexed (C/C++ style), 0..N-1, where N is from the kernel execution configuration indicated at the kernel launch

在以上的例子中，共有N个block，他们的blockIdx不同，每个block中有一个线程，执行一次加法。

VECTOR ADDITION ON THE DEVICE

```
#define N 512
int main(void) {
   int *a, *b, *c;             // host copies of a, b, c
   int *d_a, *d_b, *d_c;       // device copies of a, b, c
   int size = N * sizeof(int);
   // Alloc space for device copies of a, b, c
   cudaMalloc((void **)&d_a, size);
   cudaMalloc((void **)&d_b, size);
   cudaMalloc((void **)&d_c, size);
   // Alloc space for host copies of a, b, c and setup input values
   a = (int *)malloc(size); random_ints(a, N);
   b = (int *)malloc(size); random_ints(b, N);
   c = (int *)malloc(size);
```

VECTOR ADDITION ON THE DEVICE

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
    // Launch add() kernel on GPU with N blocks
    add<<<N,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

- 《课件-02-CUDA-Shared-Memory》

Page 4, 7, 8 SHARING DATA BETWEEN THREADS

REVIEW (2 OF 2)

- Basic device memory management
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`
- Launching parallel kernels
  - Launch `N` copies of `add()` with `add<<<N,1>>>(...);`
  - Use `blockIdx.x` to access block index

Shared Memory: SHARING DATA BETWEEN THREADS

- Terminology: within a block, threads share data via shared memory
- Shared memory is equivalent to a user-managed cache:
  - The application explicitly allocates and accesses it.
- Extremely fast on-chip memory, user-managed
- Declare using `__shared__`, allocated a variable per block that:
  - Resides in the shared memory space of a thread block,
  - Has the lifetime of the block,
  - Has a distinct object per block,
  - Is only accessible from all the threads within the block,
  - Does not have a constant address.

- 在一个块内，线程通过共享内存共享数据
- 共享内存相当于一个用户管理的缓存，在程序中显式分配和访问它
- 特点：访问速度很快，由用户管理
- 使用 `__shared__` 关键字，为每个block申请一个共享的变量
  - 驻留在线程块的共享内存空间中，存在周期和blobk的存在周期相同
  - block之间是不共享的，每个block中都有一个实例
  - 只能被block内的所有线程访问
  - 没有固定的地址

Shared Memory: SHARING DATA BETWEEN THREADS

- Typical Programming pattern:
  - Load data from device memory to shared memory
  - Synchronize with all the other threads of the block so that each thread can safely readshared memory locations that were populated by different threads,
  - Process the data in shared memory
  - Synchronize again if necessary to make sure that shared memory has been updated withthe results,
  - Write the results back to device memory.

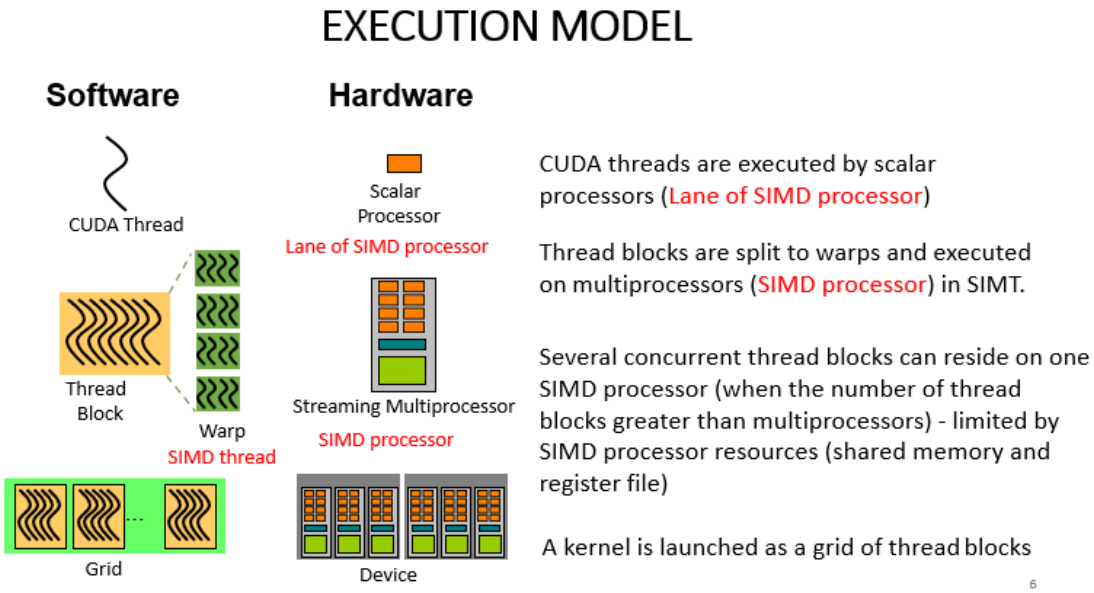
典型的编程模式：

- 将数据从设备内存加载到共享内存
- 与block中的所有其他线程同步，以便每个线程可以安全地读取由不同线程填充的共享内存位置
- 处理共享内存中的数据

- 如有必要，再次进行线程同步以确保共享内存已更新结果
- 将结果写回设备内存

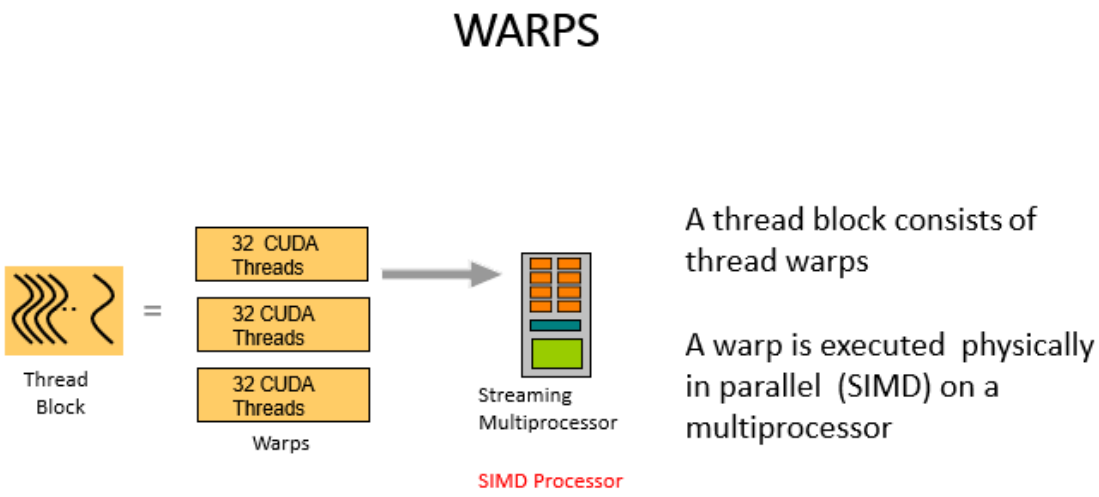
《课件-03-CUDA-Fundamental-Optimization-Part-1》

Page 6 EXECUTION MODEL



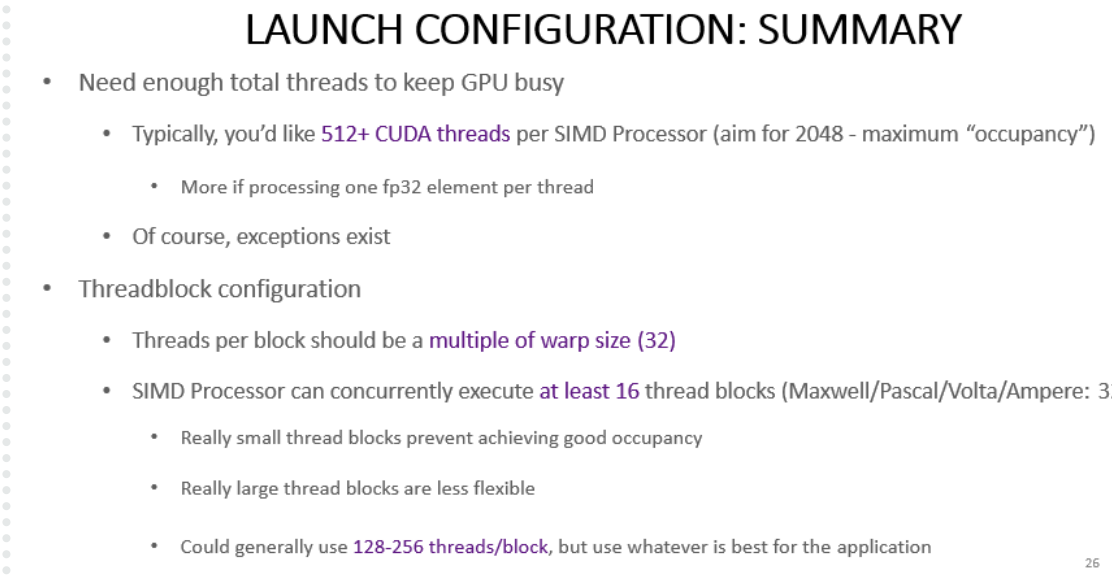
- CUDA threads在标量处理器上执行，标量处理器为SIMD处理器的车道
- block被分成warp，在SIMD处理器上执行（Streaming Multiprocessor）
- 多个并发线程块可以驻留在一个 SIMD 处理器上（当线程块的数量大于多处理器时）受 SIMD 处理器资源（共享内存和寄存器文件）的限制
- 一个grid发射一个kernel

Page 7 WARPS



- 一个block含有多个warps
- 实际上，一个warp在一个多处理器上执行

Page 26 LAUNCH CONFIGURATION: SUMMARY



为了充分利用GPU，需要足够多数量的线程来使GPU处于busy状态。

- 通常情况下，每个SIMD处理器处理512+CUDA threads

线程块配置

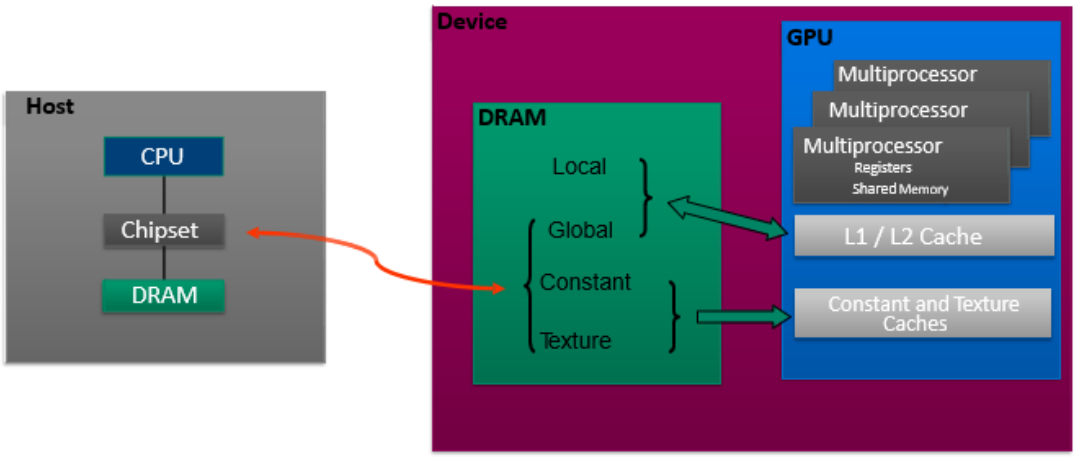
- 每个block的线程数应为一个warp的线程数的倍数
- SIMD处理器可以并发的处理至少16个block
    - block内的线程数太小则不能保持较高的GPU占用率
    - 太多会使灵活性降低
    - 通常情况下每个block使用128~256个线程，但最优的方案往往根据程序情况调整

《课件-04-CUDA-Fundamental-Optimization-Part-2》

Page 8-17 GPU MEM OPERATIONS

```
Loads:
    Caching
        Default mode
        Attempts to hit in L1, then L2, then GMEM（全局共享区域）
        Load granularity（粒度） is 128-byte line
    Non-caching
        Compile with -Xptxas -dlcm=cg option to nvcc
        Attempts to hit in L2, then GMEM
        Do not hit in L1, invalidate the line if it's in L1 already
        Load granularity is 32-bytes (segment)

Stores:
    Invalidate L1, write-back for L2
```



Load 操作

- 以warp为单位发起load操作
- 过程
  - 该warp中的CUDA thread提供需要访问的地址
  - 确定需要哪些行/段
  - 请求需要的行/段
- caching load（粒度为128字节）

  需要128字节
  - 32字节对齐
    - 连续：利用率100% `int c = a[idx];`
    - 不连续：利用率100% `int c = a[rand()%warpSize];`
  - 非32字节对齐
    - 连续：利用率50% `int c = a[idx-2];`
  - 疏散的
    - 散落在N个缓存行：利用率 128/N*128（3.125% when N=32） `int c = a[rand()];`

  需要4字节
  - 利用率 4/128=3.125% `int c = a[40];`
- Non-caching load（粒度为32字节）

  需要128字节
  - 疏散的
    - 散落在N个段：利用率 128/N*32（12.5% when N=32） `int c = a[rand()];`

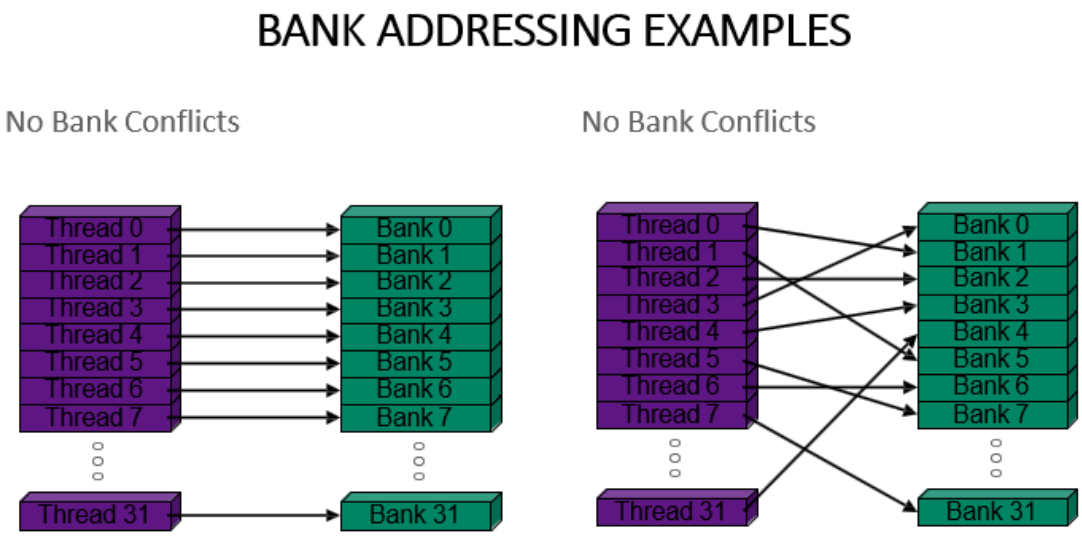
优化思路

- Strive for perfect coalescing
  - (Align starting address - may require padding)（避免非32字节对齐）
  - A warp should access within a contiguous region（避免疏散的）

- Have enough concurrent accesses to saturate the bus（足够的并发内存访问使总线饱和）
    - Process several elements per thread（每个线程处理多个元素）
        - Multiple loads get pipelined（多个负载流水线化）
        - Indexing calculations can often be reused（索引计算通常可以重复使用）
    - Launch enough warps to maximize throughput（发射足够多的warp以最大化吞吐量）
        - Latency is hidden by switching warps
- Use all the caches!

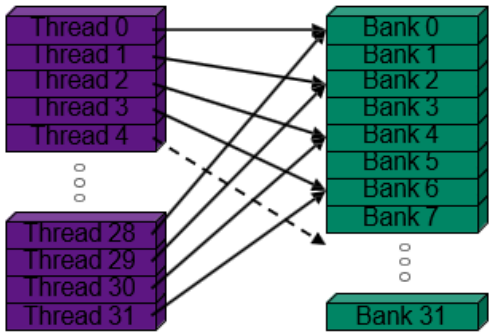
Page 20-25 SHARED MEMORY

- Uses:
    - Inter-thread communication within a block
    - Cache data to reduce redundant global memory accesses , like CPU cache
    - Use it to improve global memory access patterns
    - The shared memory is expected to be a low-latency memory near each processor core (much like an L1 cache)
    - Can be managed (allocate and free) via programming API
- Organization
    - 32 banks, 4-byte wide banks
    - Successive 4-byte words belong to different banks
- Performance
    - Typically: 4 bytes per bank per 1 or 2 clocks per multiprocessor
    - shared accesses are issued per 32 threads (warp)
    - serialization: if *N* threads of 32 access different 4-byte words in the same bank (bank conflict), *N* accesses are executed serially

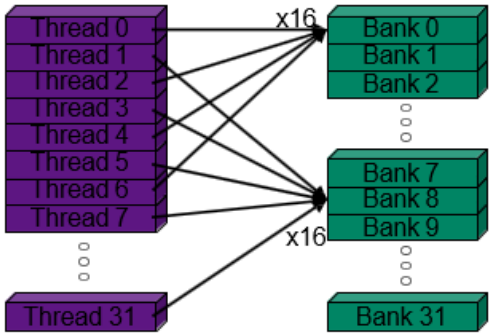


BANK ADDRESSING EXAMPLES

2-way Bank Conflicts

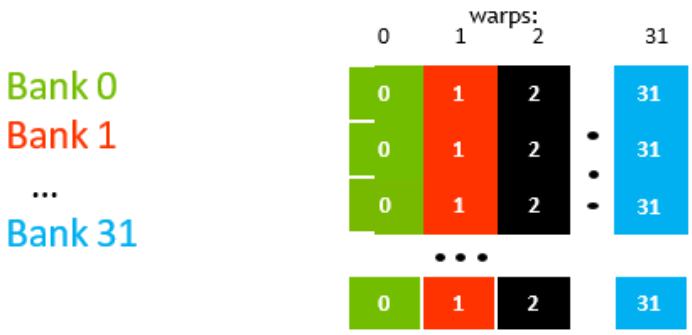


16-way Bank Conflicts



SHARED MEMORY: AVOIDING BANK CONFLICTS

- 32x32 Shared MEM array
- Warp accesses a column:
  - 32-way bank conflicts (threads in a warp access the same bank)



24

SHARED MEMORY: AVOIDING BANK CONFLICTS

- Add a column for padding:
  - 32x33 SMEM array
- Warp accesses a column:
  - 32 different banks, no bank conflicts

25

《课件-05_Atomics_Reductions_Warp_Shuffle》

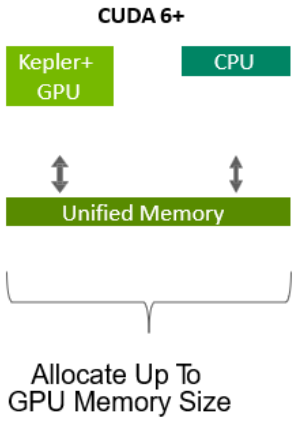
Page12-33 parallel reduction optimization （重要但不在考试范围）

《课件-06_Managed_Memory》

Page 5-9 UNIFIED MEMORY

UNIFIED MEMORY

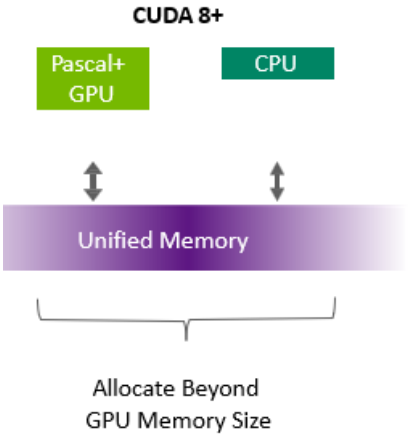
Reduce Developer Effort



Simpler Programming & Memory Model	Single allocation, single pointer, accessible anywhere Eliminate need for *explicit* copy Simplifies code porting
Maintain Performance through Data Locality	Migrate data to accessing processor Guarantee global coherence Still allows explicit hand tuning

CUDA 8+: UNIFIED MEMORY

Demand Paging For Pascal and Beyond



Enable Large Data Models	Oversubscribe GPU memory Allocate up to system memory size
Simpler Data Accesss	CPU/GPU Data coherence Unified memory atomic operations
Tune Unified Memory Performance	Usage hints via cudaMemAdvise API Explicit prefetching API

SIMPLIFIED MEMORY MANAGEMENT CODE

CPU Code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);

  use_data(data);

  free(data);
}
```

Ordinary CUDA Code

```
void sortfile(FILE *fp, int N) {
  char *data, *d_data;
  data = (char *)malloc(N);
  cudaMalloc(&d_data, N);
  fread(data, 1, N, fp);
  cudaMemcpy(d_data, data, N, …); // 1
  qsort<<<...>>>(data,N,1,compare); // 2
  cudaMemcpy(data, d_data, N, …); // 3

  use_data(data);
  cudaFree(d_data);
  free(data);
}
```

SIMPLIFIED MEMORY MANAGEMENT CODE

CPU Code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

CUDA Code with Unified Memory

```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```

UNIFIED MEMORY EXAMPLE

With On-Demand Paging

```
__global__
void setValue(int *ptr, int index,  int val)
{
  ptr[index] = val;
}


void foo(int size) {
  char *data;
  cudaMallocManaged(&data,  size);

  memset(data, 0, size);

  setValue<<<...>>>(data, size/2, 5);
  cudaDeviceSynchronize();

  useData(data);

  cudaFree(data);
}
```

Unified Memory allocation

Access all values on CPU

Access one value on GPU

总结：相当于CPU和GPU的共享存储器？

《并行程序设计导论》

第一章：

1.1

1.2

1.3

1.4

1.6

第二章：

2.1.1

2.1.2

2.2.1 至 2.2.6

2.3.1

2.3.2

2.3.4

2.3.5

2.4.2

2.4.3

2.4.4

2.5

2.6.1至2.6.4

2.7

2.9

第三章：

3.1.1至3.1.12

3.2.1

3.2.2

3.3.1

3.3.2

3.4.1至3.4.9

3.5

3.6.1至3.6.4

3.7.1

3.7.2

第四章：

4.1

4.2.1至4.2.7

4.3至4.7

4.8.1至4.8.3

4.9.1至4.9.4

4.10

4.11

第五章：

5.1至5.5

5.7

5.8

5.9

5.10

第六章：

6.1 N-body问题

《计算机体系结构-量化研究方法-第六版》

第四章（向量处理器、GPU）

1. PPT内容

Page 5 Flynn's Taxonomy 弗林分类法

- *SISD* - Single instruction stream, single data stream
- *SIMD* - Single instruction stream, multiple data streams
    - Extend pipelined execution of many data operations, superscalar（扩展许多数据操作的流水线执行，超标量）
    - Simultaneous parallel data operations in most instruction set. E.g. SSE(streaming SIMD extensions), AVX
    - New: *SIMT* – Single Instruction Multiple Threads (for GPUs)
- *MISD* - Multiple instruction streams, single data stream(nNo commercial implementation)
- *MIMD* - Multiple instruction streams, multiple data streams
    - Tightly-coupled MIMD (shared memory, NUMAs), OpenMP and Pthreads
    - Loosely-coupled MIMD (distributed memory system, e.g. cluster), MPI

Page 6-7 Advantages of SIMD architectures （SIMD架构的优势）

1. Can exploit significant data-level parallelism for:
    - A set of applications has significant data-level parallelism（具有显着的数据级并行性的程序）
    - Matrix-oriented scientific computing（面向矩阵的科学计算）
    - Media-oriented image and sound processors（面向媒体的图像和声音处理器）
2. More energy efficient than MIMD（相比于MIMD）
    - Only needs to fetch one instruction per multiple data operations, rather than one instr. per data op.（MIMD需要取出新的指令）
    - Makes SIMD attractive for personal mobile devices
3. Allows programmers to continue thinking sequentially

Page 11 Example of vector architecture

Example of vector architecture

- RV64V → RV64G, RV64V extended RV64G with vector instructions, here RV (RISC-V)
  - RISC-V (pronounced "risk-five") is an open standard instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles.
- Vector registers
  - Each register holds a 32-element, 64 bits/element vector
  - Register file has 16 read ports and 8 write ports, A register file is an array of processor registers in a central processing unit (CPU).
- Vector functional units – FP add and multiply
  - Fully pipelined
  - Data and control hazards are detected
- Vector load-store unit
  - Fully pipelined
  - One word per clock cycle after initial latency
- Scalar registers
  - 31 general-purpose registers
  - 32 floating-point registers
  - Provide data as input to the vector functional units
  - Compute addresses to pass to the vector load/store unit

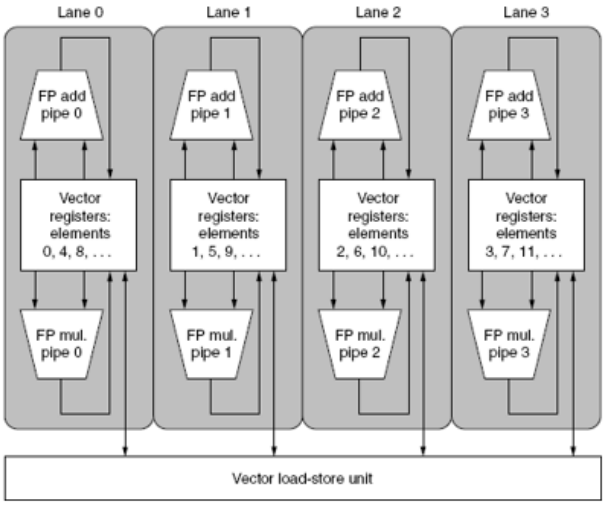
Page 26 Optimizations

1. *Multiple Lanes* *（多条车道）* → processing more than one element per clock cycle
2. *Vector Length Registers* *（向量长度寄存器）* → handling non-32 wide vectors
3. *Vector Mask Registers* *（向量遮罩寄存器）* → handling IF statements in vector code
4. *Memory Banks* *（内存组）* → memory system optimizations to support vector processors
5. *Stride* *（步幅）* → handling multi-dimensional arrays
6. *Scatter-Gather* *（分散-集中）* → handling sparse matrices
7. *Programming Vector Architectures* *（向量体系结构编程）* → program structures affecting performance

Question: how these optimizations map to MPI and OpenMP?

Page 28 1. A four lane vector unit

- RV64V instructions only allow element N of one vector to take part in operations involving element N from other vector registers→ this simplifies the construction of a highly parallel vector unit
  - Lane → contains one portion of the vector register elements and one execution pipeline from each functional unit
  - Analog with a highway with multiple lanes!!



lane 包含向量寄存器一部分元素，执行每个功能单元的一个流水线任务（每个向量功能单元使用多条流水线）

Page 38-39 Memory banks, 课本 page 298-299

- Memory system must be designed to support high bandwidth for vector loads and stores
  - Memory start-up time: get the first word from memory in to a register
- Spread accesses across multiple banks
  - Control bank addresses independently
  - Load or store non-sequential words
  - Support multiple vector processors sharing the same memory
- Example:
  - 32 processors, each generating 4 loads and 2 stores/cycle
  - Processor cycle time is 2.167 ns, DRAM bank busy time (bank latency) is 15 ns
  - How many minimum memory banks needed to allow all processors to run at the full memory bandwidth (no bank conflicts)?
    - 32 processors x 6 =192 memory accesses,
    - 15ns DRAM cycle /2.167ns processor cycle≈7 processor cycles
    - 7 x 192 →1344!

为了支持向量处理器的存取，存储系统需要进行相应的设计。

内存组：为向量载入/存储单元提供带宽

使用多个内存组的原因

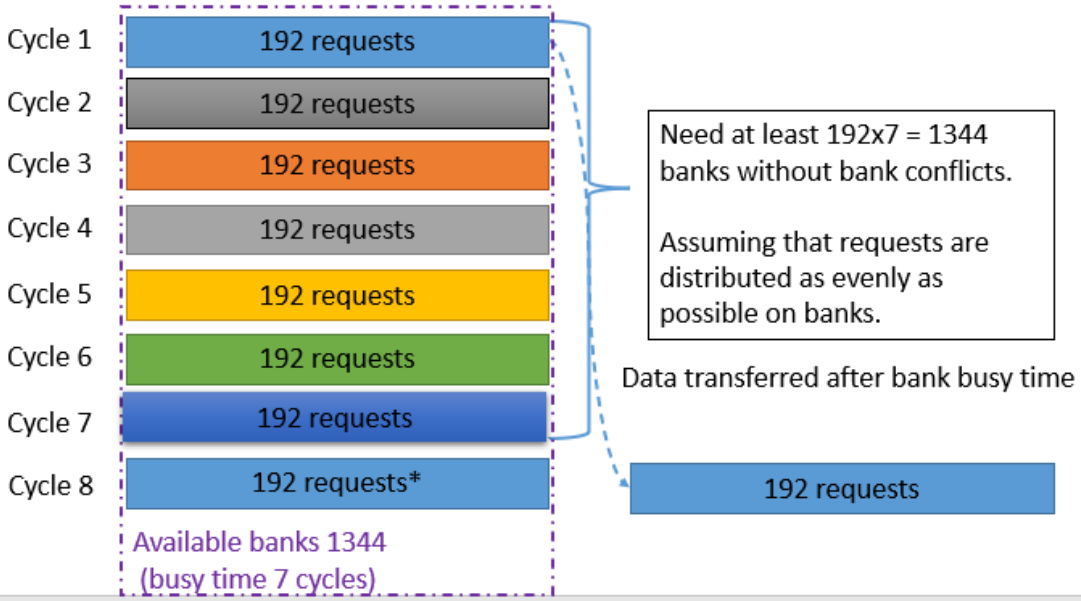
- 控制bank地址独立
- 存取不连续的字
- 多个向量处理器共享存储区域

举例

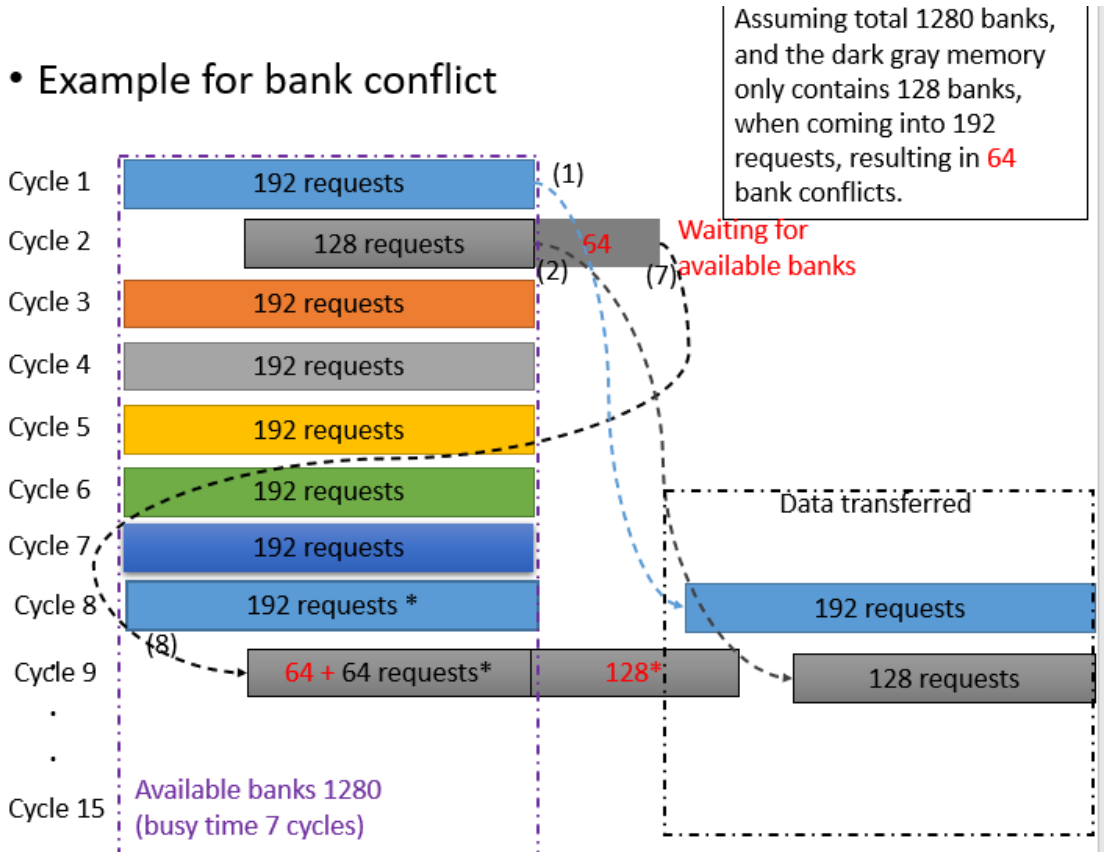
- 32个处理器，每个需要4 loads、2 stores
- 周期为2.167ns 你、bank延迟为15ns
- 若要使所有处理器以全内存带宽运行，最少需要多少个bank?
    - 32 * 6 = 192，需要192次内存访问
    - 15ns/2.167ns=7 7个处理器周期
    - 7*192=1344

Page 40-43

- No bank conflict is a critical condition for achieving full memory bandwidth.



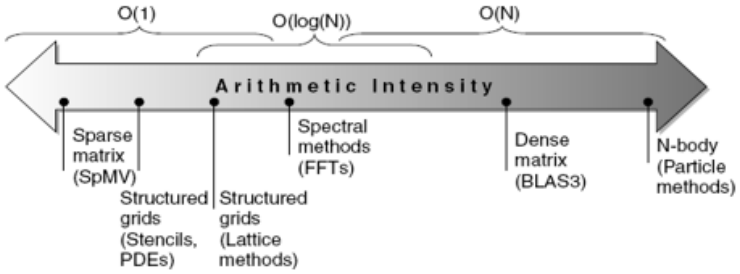
- Example for bank conflict



Page 57-58 Roofline Model 课本 Page 307-310

roofline可视性能模型

- Basic idea:
  - Peak floating-point throughput as a function of arithmetic intensity
  - Ties together floating-point performance and memory performance
- *Arithmetic intensity* (*计算密度、计算访存比*)→ Floating-point operations per byte read
- Attainable GFLOPs/sec = Min (Peak Memory BW × Arithmetic Intensity, Peak Floating Point Performance)
  Stream is a benchmark for memory performance



- Dense matrix operations scale with problem size but sparse matrix operations do not!!

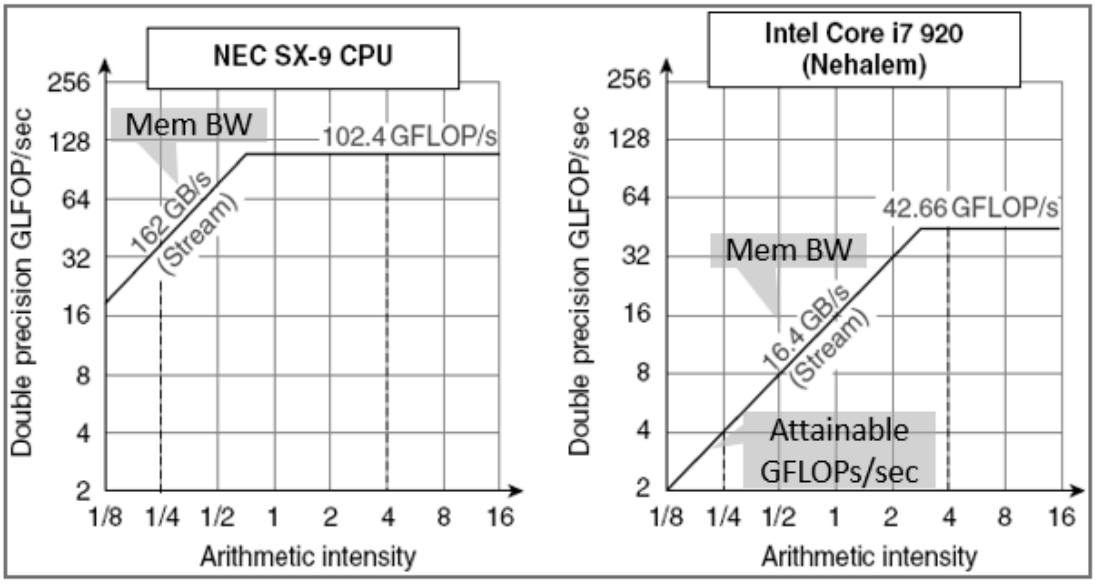
计算密度 = 浮点运算数/所访问的存储器字节数

（计算方法为：获取一个程序的总浮点数运算数，再除以在程序执行期间向主存储器传送的总数据字节）

峰值浮点性能 可以使用硬件规范求得，由缓存背后的存储器系统确定

峰值存储器带宽

可获得的$GFLOP/s = Min(\text{峰值存储器带宽} \times \text{运算密度}, \text{峰值浮点性能})$



on the sloped portion of the roof the performance is limited by the memory bandwidth, on the flat portion it is limited by peak floating point performance.

Page 59-60 C. Graphical Processing Unit GPU

GPU原用于图形处理，为了使其并行化处理的能力能被更多的应用程序使用：

基本思路：

- 异构执行模型
  - CPU是主机，GPU是设备
- 为 GPU 开发类似 C 的编程语言
  - 计算统一设备架构 (CUDA)
  - OpenCL 用于独立于供应商的语言
- 将所有形式的 GPU 并行性统一为 CUDA 线程
- 编程模型：“单指令多线程”（SIMT）

Threads, blocks, and grid (CUDA)

- A *thread* is associated with each data element
  - *CUDA threads* ➔ thousands of threads are utilized to various styles of parallelism: multithreading, SIMD, MIMD, ILP
- Threads are organized into *blocks*
  - *Thread Blocks*: groups of up to 512 elements
  - *Multithreaded SIMD Processor*: hardware that executes a whole thread block (32 elements executed per thread at a time)
- Blocks are organized into a *grid*
  - Blocks are executed independently and in any order
  - Different blocks cannot communicate directly but can *coordinate* using atomic memory operations in Global Memory
- Thread management handled by GPU hardware not by applications or OS
  - A multiprocessor composed of multithreaded SIMD processors
  - A Thread Block Scheduler

Page 62-63 Threads, blocks, and grid (CUDA) 课本 Page 313-320

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
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)

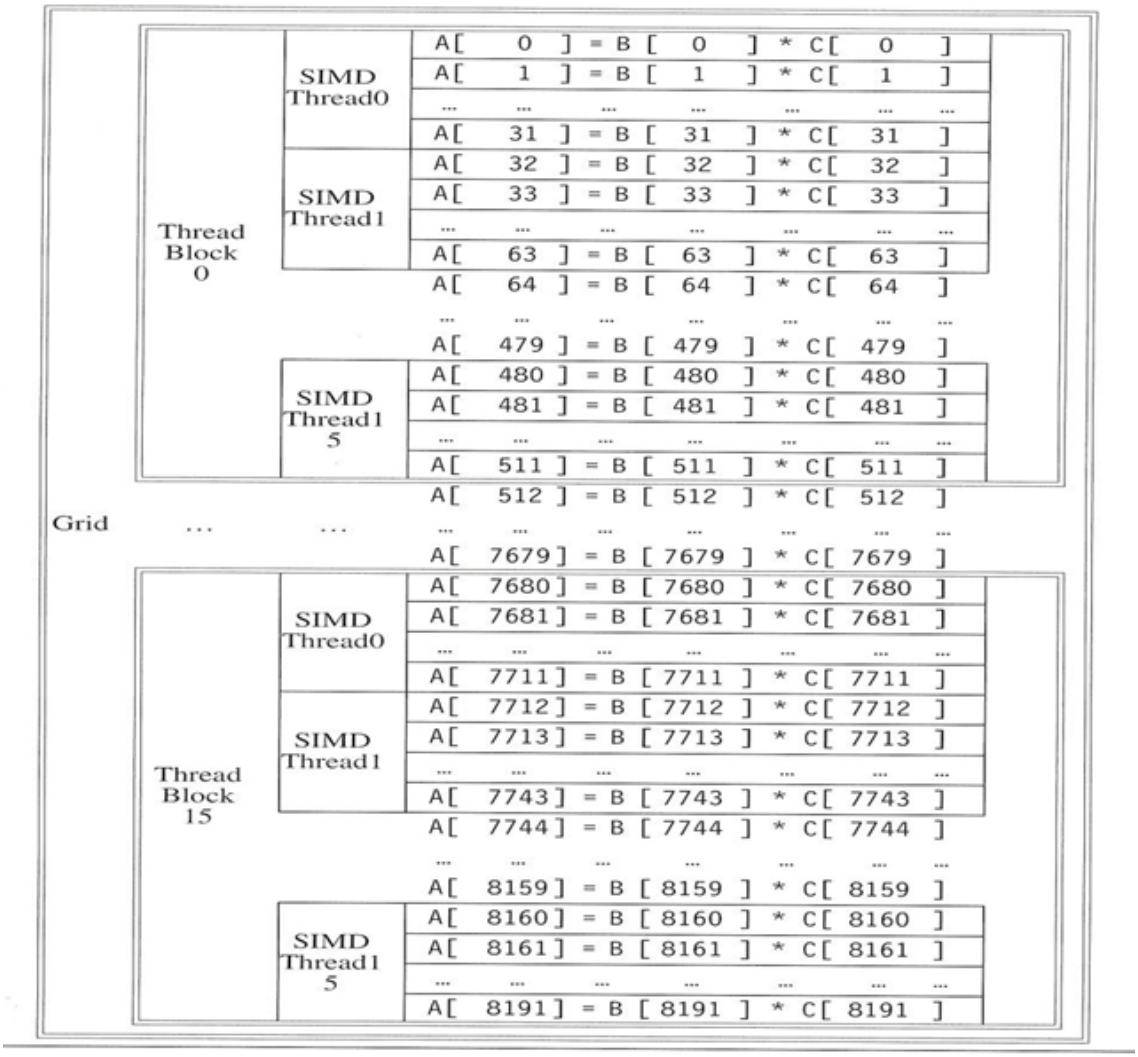
Page 64-67 Example: multiply two vectors of length 8192

c

```c
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n = 8192, double a, double *A, double *B, double *C){
 for (int i = 0; i < n; ++i)
    A[i] = B[i] * C[i];
}
```

cuda

```
// Invoke DAXPY with 512 threads per Thread Block
__host__
int nblocks = (n + 511) / 512;
daxpy<<<nblocks, 512 >>>(n, 2.0, A, B, C);
// DAXPY in CUDA
__global__
void daxpy (int n, double a, double *A, double *B, double *C)
{
 int i = blockIdx.x*blockDim.x + threadIdx.x;
 if (i < n)
        A[i] = B[i] * C[i];
}
```



Page 71-72 NVIDIA GPU memory structures 课本 Page 326-328

- Each SIMD Lane has private section of *off-chip* DRAM
  - "Private memory", not shared by any other lanes
  - Contains stack frame, register spilling, and private variables that don't fit in the registers.
  - Recent GPUs cache this private memory in L1 and L2 caches
- Each multithreaded SIMD processor also has local memory (Shared Memory) that is *on-chip*
  - Shared by SIMD lanes / threads *within a block only*
  - Latency is100x lower than GPU Memory.
  - Threads can access data in local memory loaded from global memory by other threads within the same thread block
- The *off-chip* memory shared by SIMD processors is *GPU* memory (Global memory)
  - Host can read and write GPU memory

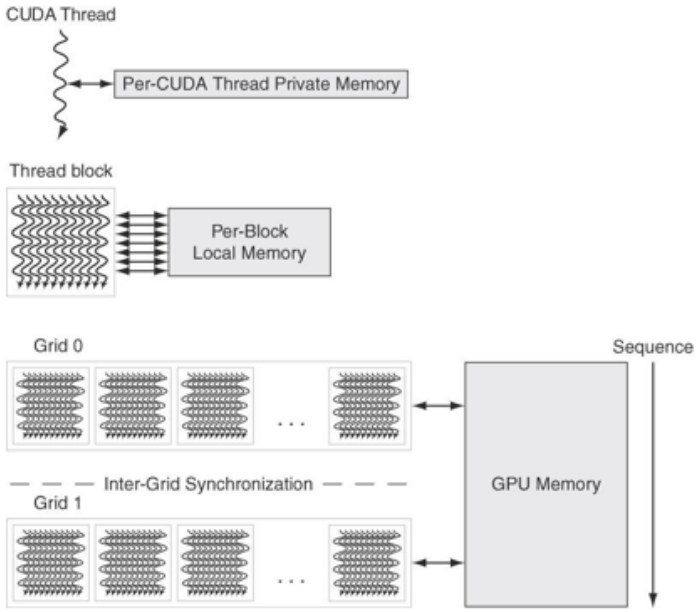
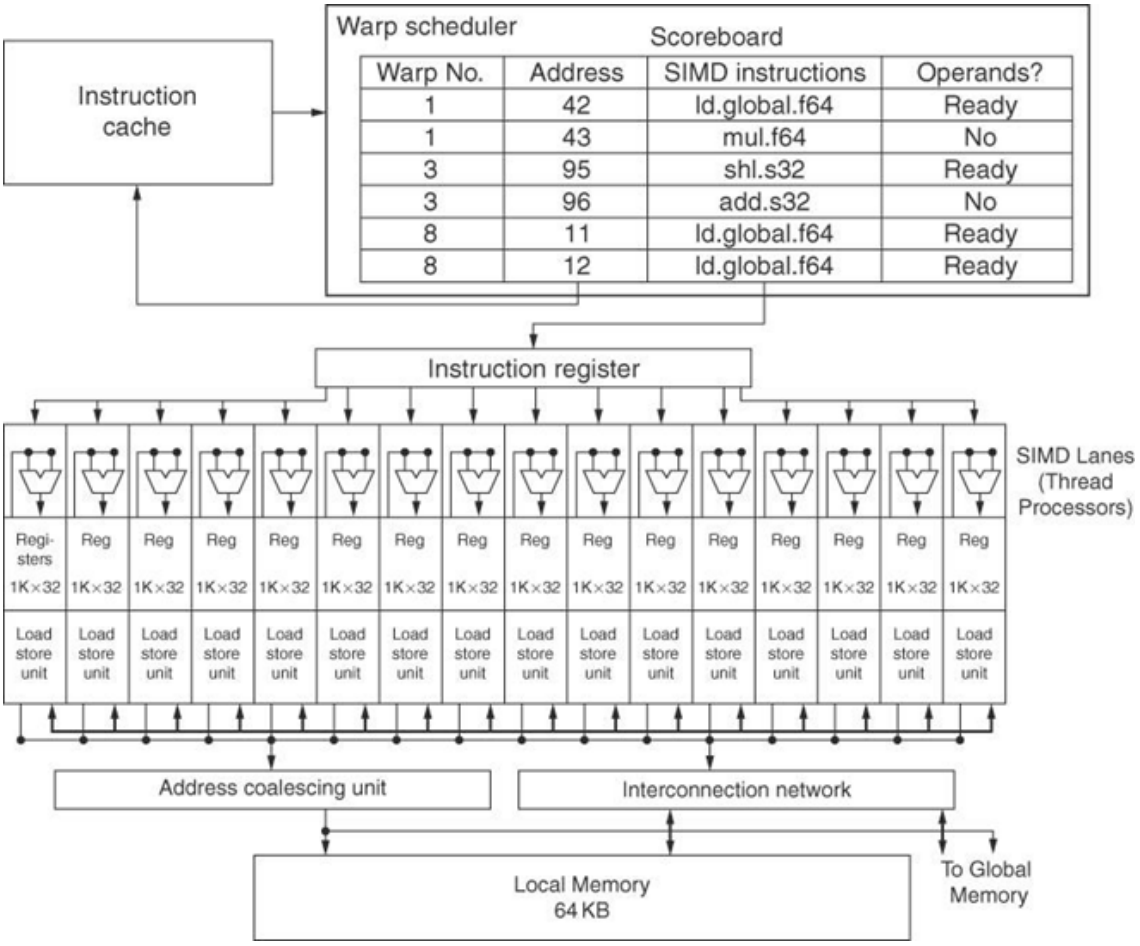


Figure 4.18 GPU Memory structures.

- *GPU Memory (Global Memory)* shared by all Grids (vectorized loops),
- *Local Memory (Shared Memory)* shared by all threads of SIMD instructions within a thread block (body of a vectorized loop).
- *Private Memory* private to a single CUDA thread.

- *Threads of SIMD instructions*
  - Each has its own PC
  - Thread scheduler uses scoreboard to dispatch
  - No data dependencies between threads!
  - Keeps track of up to 48 threads of SIMD instructions
    - Hides memory latency
- *Thread block scheduler*
  - Schedule thread blocks to SIMD processors, each thread block is split to sub-blocks (warps) (e.g. 32 CUDA threads / warp) and assigned to SIMD threads running on a SIMD processor.
- Within each SIMD processor:
  - 16 or 32 SIMD lanes
  - Wide and shallow compared to vector processors



Page 68-69 Conditional branching 课本 Page 323 – 326

Page 90, 93, 94 Loop level parallelism

Assume that a *1-D* array index *i* is **affine**:

- *a* * *i* + *b* (with constants *a* and *b*)

- An *n-D* array index is *affine* if it is affine in each dimension

Assume:

- Store to $a * i + b$, then
  - Load from $c * i + d$
  - i runs from m to n
- Dependence exists if:
  - Given j, k such that $m \le j \le n$, $m \le k \le n$
  - Store to $a \times j + b$, load from $c \times k + d$, and $a * j + b = c * k + d$

Generally cannot determine at compile time

- Test for absence of a dependence:
- GCD test:
  - If a dependency exists, GCD(c,a) must evenly divide (d-b)

2019：

概念

SIMD和MIMD的区别和联系

共享内存的两个优点和两个缺点

实现同步的两种方式

gpu Shared Memory和Global Memory的区别和联系

简答

解释链表访问冲突

画出Roofline模型、计算、计算矩阵乘法的计算密度，判断性能是由于CPU计算还是访存受限

omp parallel for循环依赖、修改程序、static调度chunksize为2时每个线程负责的元素

编程

MPI梯形法算积分