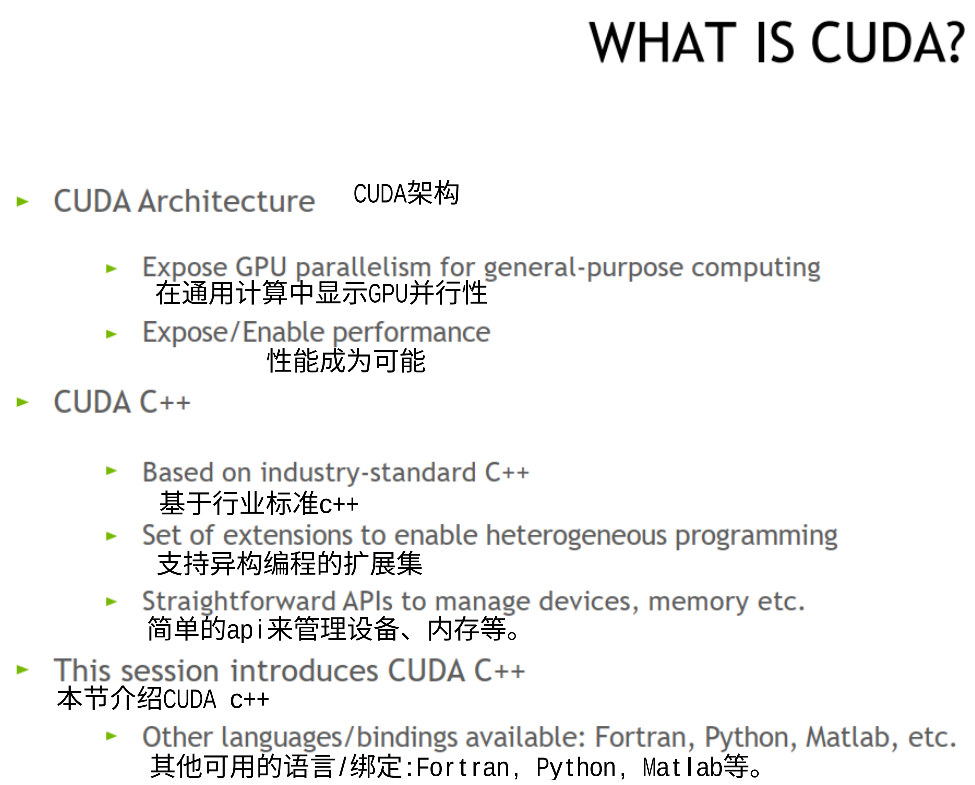
CUDA_C_Programming_Guide

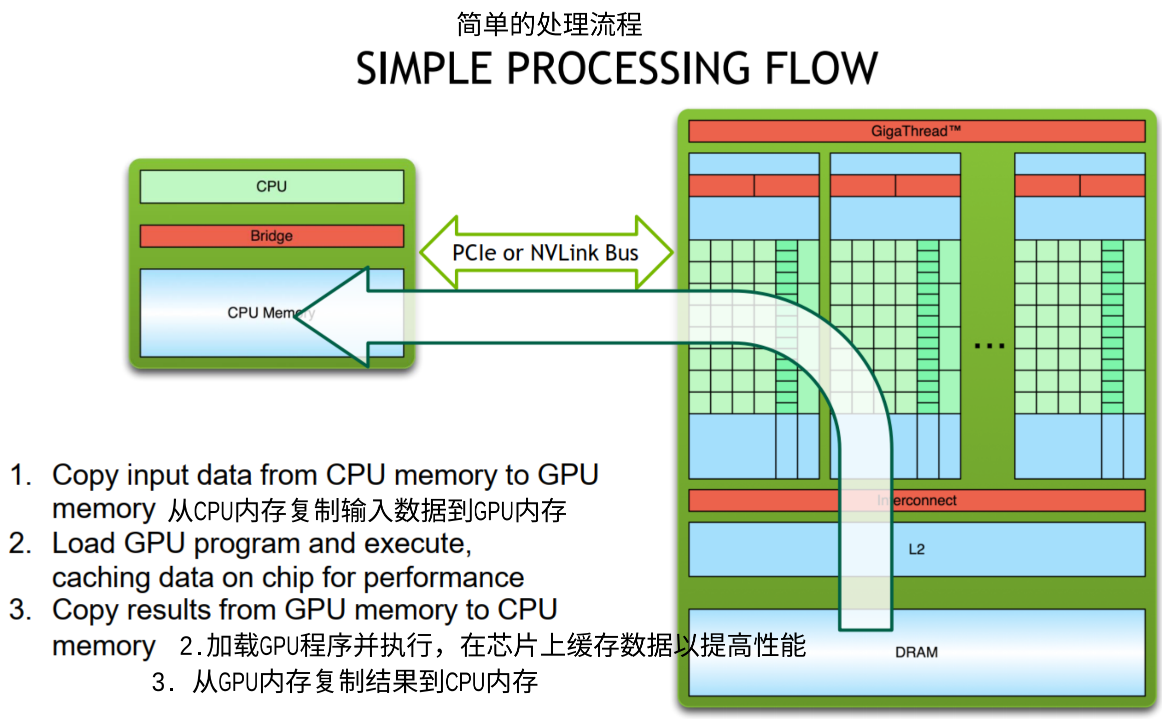
高性能考点总结 王若琪

《课件-01-CUDA-C-Basics》

Page 2 WHAT IS CUDA?



Page 8 SIMPLE PROCESSING FLOW



Page 10-11 GPU KERNELS: DEVICE CODE

GPU KERNELS: DEVICE CODE

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

- CUDA C++ keyword `__global__` indicates a function that:
  CUDA c++ keyword global表示一个函数:
  - Runs on the device
    在设备上运行
  - Is called from host code (can also be called from other device code)
    从主机代码调用(也可以从其他设备代码调用)
- `nvcc` separates source code into host and device components
  nvcc将源代码分离为主机和设备组件
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
    设备函数(如mykernel())由NVIDIA编译器处理
  - Host functions (e.g. `main()`) processed by standard host compiler:
    由标准主机编译器处理的主机函数(例如main()):
    - `gcc, cl.exe`

GPU KERNELS: DEVICE CODE

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

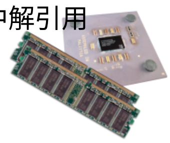
- Triple angle brackets mark a call to *device* code
  三尖括号表示对设备代码的调用
  - Also called a "kernel launch"
    也称为“内核启动”
  - We'll return to the parameters (1,1) in a moment
    稍后我们将回到参数(1,1)
  - The parameters inside the triple angle brackets are the CUDA kernel **execution configuration**
    三尖括号内的参数是CUDA内核执行配置
- That's all that is required to execute a function on the GPU!
  这就是在GPU上执行一个函数所需要的所有东西!

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

内存管理

MEMORY MANAGEMENT

- Host and device memory are separate entities 主机和设备内存是独立的实体
- Device pointers point to GPU memory 设备指针指向GPU内存
  - Typically passed to device code 通常传递给设备代码
  - Typically not dereferenced in host code 通常不会在宿主代码中取消引用
- Host pointers point to CPU memory 主机指针指向CPU内存
  - Typically not passed to device code 通常不传递给设备代码
  - Typically not dereferenced in device code 通常不会在设备代码中解引用



处理设备内存的简单CUDA API

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

RUNNING CODE IN PARALLEL

- GPU computing is about massive parallelism
  GPU计算是关于海量并行的
  - So how do we run code in parallel on the device?
    那么我们如何在设备上并行运行代码呢?

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

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

- Instead of executing `add()` once, execute N times in parallel
  不是执行一次add()，而是并行执行N次

VECTOR ADDITION ON THE DEVICE

- With `add()` running in parallel we can do vector addition
  通过并行运行add()，我们可以进行向量相加
- Terminology: each parallel invocation of `add()` is referred to as a **block**
  术语:每次并行调用add()都被称为一个块
  - The set of all blocks is referred to as a **grid**
    所有块的集合称为网格
  - Each invocation can refer to its block index using `blockIdx.x`
    每个调用都可以使用blockIdx.x引用它的块索引

```
__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
  通过使用blockIdx来索引数组，每个块处理不同的索引
- 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
  内置变量，如blockIdx。x是零索引的(C/ c++风格)，0..N-1，其中N来自内核启动时指示的内核执行配置

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》

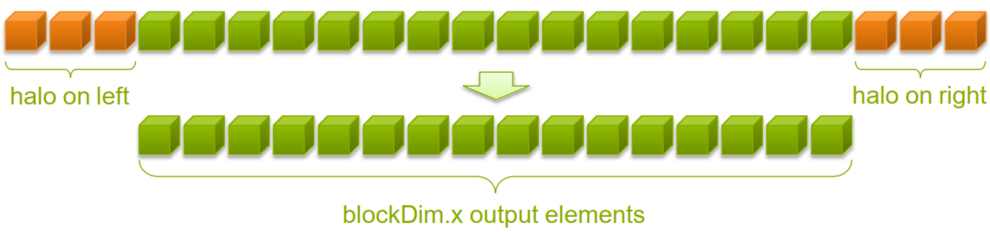
Page 6-13 SHARING DATA BETWEEN THREADS

SHARING DATA BETWEEN THREADS

- Terminology: within a block, threads share data via shared memory
  术语:在一个块中，线程通过共享内存共享数据
- Extremely fast on-chip memory, user-managed
  极快的片上存储，用户管理
- Declare using `__shared__`, allocated per block
  声明使用共享，分配每个块
- Data is not visible to threads in other blocks
  数据对其他块中的线程不可见

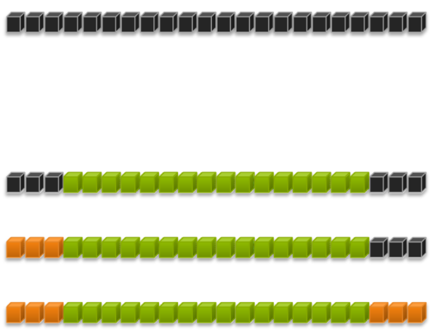
IMPLEMENTING WITH SHARED MEMORY

- Cache data in shared memory
  - Read `(blockDim.x + 2 * radius)` input elements from global memory to shared memory
  - Compute `blockDim.x` output elements
  - Write `blockDim.x` output elements to global memory
- Each block needs a halo of `radius` elements at each boundary



STENCIL KERNEL

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =
      in[gindex + BLOCK_SIZE];
  }
```



STENCIL KERNEL

```
// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
  result += temp[lindex + offset];

// Store the result
out[gindex] = result;
}
```

DATA RACE!

- The stencil example will not work...

- Suppose thread 15 reads the halo before thread 0 has fetched

```
temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex – RADIUS] = in[gindex – RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
  }
  int result = 0;
  result += temp[lindex + 1];
```

Store at temp[18]

Skipped, threadIdx > RADIUS

Load from temp[19]

__SYNCTHREADS()

- `void __syncthreads();`

- Synchronizes all threads within a block
  同步块内的所有线程
  - Used to prevent RAW / WAR / WAW hazards
    防止冒险

- All threads must reach the barrier
  所有的线程必须到达屏障
  - In conditional code, the condition must be uniform across the block
    在条件代码中，条件必须在整个块中是一致的

STENCIL KERNEL

```
__global void stencil_1d(int *in, int *out) {
    __shared int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
    // Synchronize (ensure all the data is available)
    __syncthreads();
```

STENCIL KERNEL

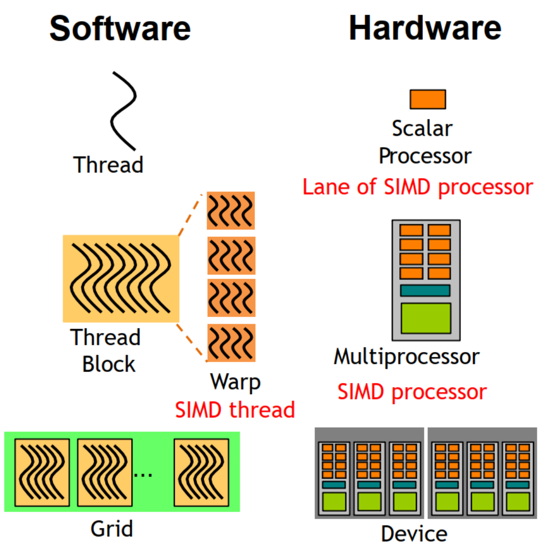
```
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}
```

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

Page 6 EXECUTION MODEL

执行模型

EXECUTION MODEL



Threads are executed by scalar processors (Lane of SIMD processor) 线程由标量处理器执行

Thread blocks are split to warps and executed on multiprocessors (SIMD processor) in SIMT.
线程块被分割成warps，并在SIMT中的多处理器(SIMD处理器)上执行。

Several concurrent thread blocks can reside on one multiprocessor (when the number of thread blocks greater than multiprocessors) - limited by multiprocessor resources (shared memory and register file)

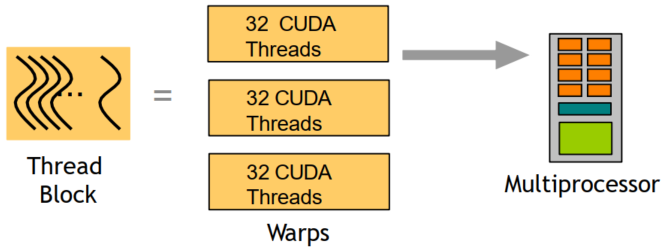
A kernel is launched as a grid of thread blocks

多个并发线程块可以驻留在一个多处理器上(当线程块的数量大于多处理器时)——受多处理器资源(共享内存和寄存器文件)的限制

内核以线程块grid的形式启动

Page 7 WARPS

WARPS



A thread block consists of 32-thread warps

A warp is executed physically in parallel (SIMD) on a multiprocessor

warp在物理上是平行执行的(SIMD)在多处理器上

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

Page 8-17 GPU MEM OPERATIONS

GPU MEM OPERATIONS

GPU MEM操作

- Loads:
  - Caching 超高速缓存
    - Default mode 默认模式
    - Attempts to hit in L1, then L2, then GMEM 尝试击中L1，然后是L2，然后是GMEM
    - Load granularity is 128-byte line 加载粒度为128字节行
- Stores:
  - Invalidate L1, write-back for L2 L1无效，L2回写

GPU MEM OPERATIONS

- Loads:
  - Non-caching 没有cache
    - Compile with -Xptxas -dlcm=cg option to nvcc
      使用-Xptxas -dlcm =cg选项编译nvcc
    - Attempts to hit in L2, then GMEM
      尝试在L2命中，然后GMEM命中
      Do not hit in L1, invalidate the line if it's in L1 already
      不要在L1中点击，如果它已经在L1中，就使这行无效
    - Load granularity is 32-bytes (segment)
      加载粒度为32字节(段)

We won't spend much time with non-caching loads in this training session

我们不会花太多时间讨论非缓存负载

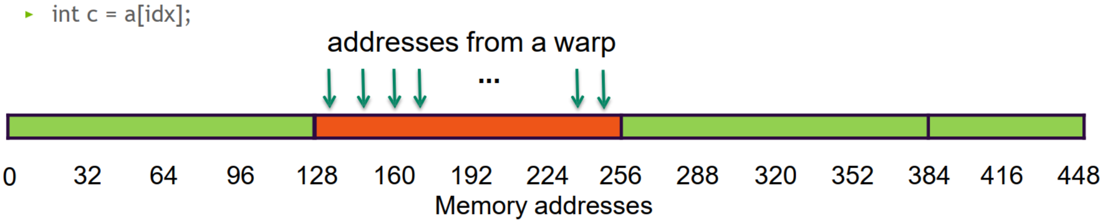
LOAD OPERATION

- Memory operations are issued per warp (32 threads)
  内存操作是每个warp(32个线程)发出的
  - Just like all other instructions
- Operation:
  - Threads in a warp provide memory addresses
    一个wrap中的线程提供内存地址
  - Determine which lines/segments are needed
    确定需要哪些行/段
  - Request the needed lines/segments
    请求所需的行/段

CACHING LOAD

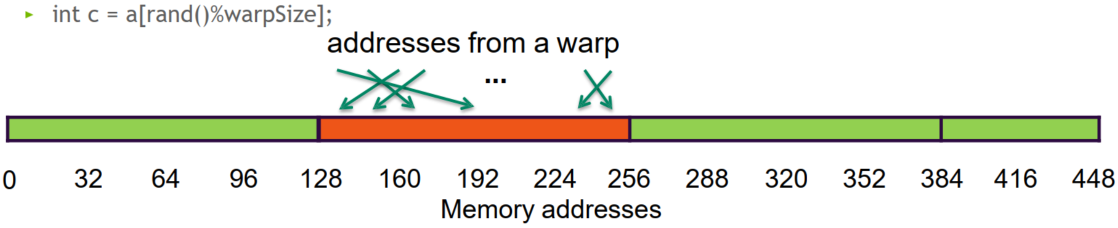
缓存加载

- Warp requests 32 aligned, consecutive 4-byte words
  wrap请求32个对齐，连续4字节的字
- Addresses fall within 1 cache-line
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
    128个字节在一次遗漏时穿过总线
  - Bus utilization: 100%
    总线利用率:100%
  - int c = a[idx];



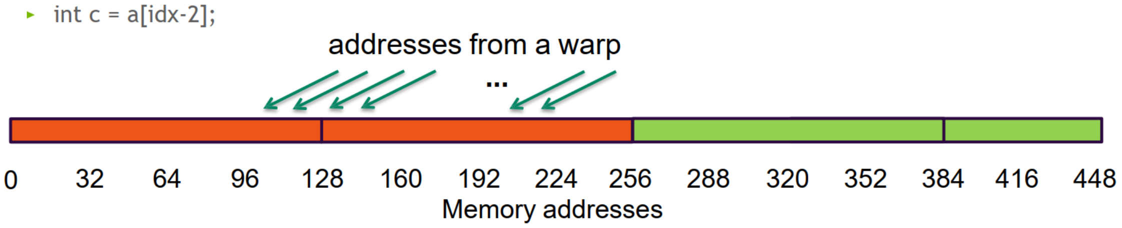
CACHING LOAD

- Warp requests 32 aligned, permuted 4-byte words
- Addresses fall within 1 cache-line
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%
  - int c = a[rand()%warpSize];



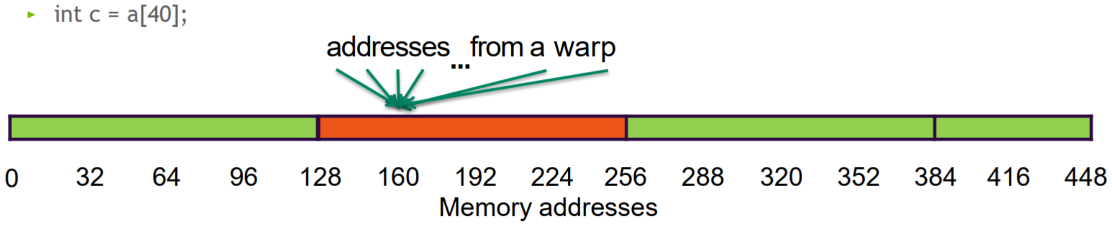
CACHING LOAD

- Warp requests 32 misaligned, consecutive 4-byte words
偏移的
- Addresses fall within 2 cache-lines
  - Warp needs 128 bytes
  - 256 bytes move across the bus on misses
  - Bus utilization: 50%
  - int c = a[idx-2];



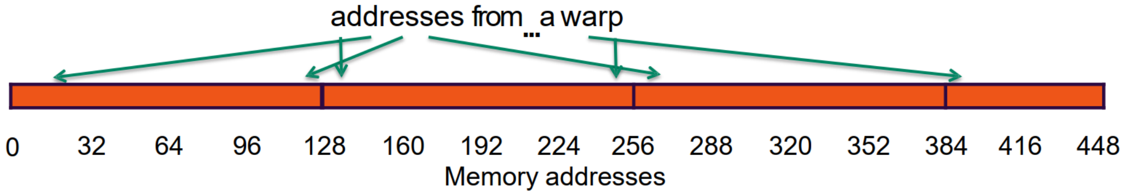
CACHING LOAD

- All threads in a warp request the same 4-byte word
- Addresses fall within a single cache-line
  - Warp needs 4 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 3.125%
  - int c = a[40];



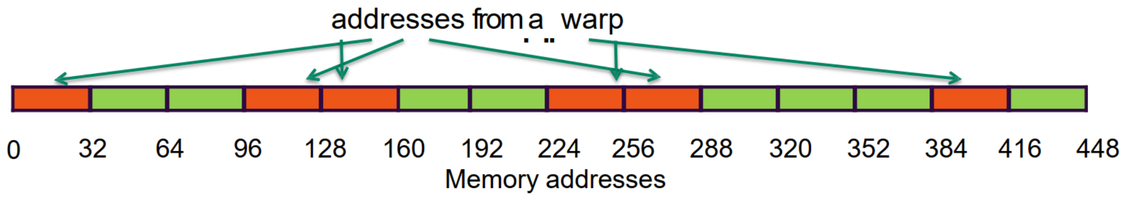
CACHING LOAD

- Warp requests 32 scattered 4-byte words
散布的
- Addresses fall within N cache-lines
  - Warp needs 128 bytes
  - N*128 bytes move across the bus on a miss
  - Bus utilization: 128 / (N*128) (3.125% worst case N=32)
  - int c = a[rand()];



NON-CACHING LOAD

- Warp requests 32 scattered 4-byte words
- Addresses fall within N segments
  - Warp needs 128 bytes
  - N*32 bytes move across the bus on a miss
  - Bus utilization: 128 / (N*32) (12.5% worst case N = 32)
  - int c = a[rand()]; -Xptxas -dlcm=cg



GPU MEM OPTIMIZATION GUIDELINES

GPU MEM优化指南

- Strive for perfect coalescing
  争取完美融合
  - (Align starting address - may require padding)
    对齐起始地址-可能需要填充
  - A warp should access within a contiguous region
    一个wrap应该在一个相邻区域内访问
- Have enough concurrent accesses to saturate the bus
  有足够的并发访问使总线饱和
  - Process several elements per thread
    每个线程处理几个元素
    - Multiple loads get pipelined
      多个load被流水线化
    - Indexing calculations can often be reused
      索引计算通常可以重用
  - Launch enough warps to maximize throughput
    发射足够的wraps来最大化吞吐量
    - Latency is hidden by switching warps
      交换wrap使得延迟被隐藏
- Use all the caches!
  使用所有的缓存!

Page 20-25 SHARED MEMORY

SHARED MEMORY

共享内存

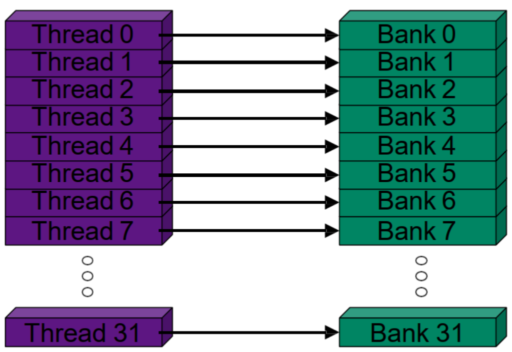
- Uses:
  用途
  - Inter-thread communication within a block
    块内的线程间通信
  - Cache data to reduce redundant global memory accesses
    缓存数据以减少冗余的全局内存访问
  - Use it to improve global memory access patterns
    使用它来改进全局内存访问模式
- Organization:
  组织
  - 32 banks, 4-byte wide banks
    32个组，4字节宽的组
  - Successive 4-byte words belong to different banks
    连续的4字节字属于不同的组

SHARED MEMORY

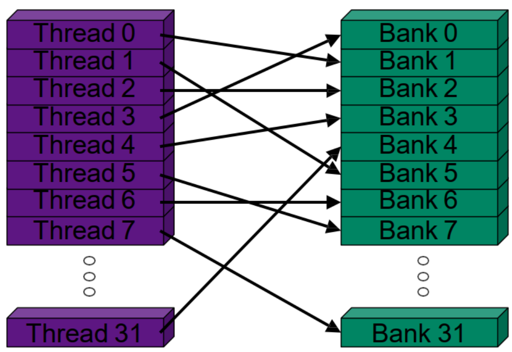
- Performance:
  性能
  - Typically: 4 bytes per bank per 1 or 2 clocks per multiprocessor
  - shared accesses are issued per 32 threads (warp)
    每32个线程（wrap）发出共享访问
  - serialization: if *N* threads of 32 access different 4-byte words in the same bank, *N* accesses are executed serially
    序列化:如果N个线程(32个线程中)访问同一bank中不同的4字节字元，那么N个访问将串行执行

BANK ADDRESSING EXAMPLES

No Bank Conflicts

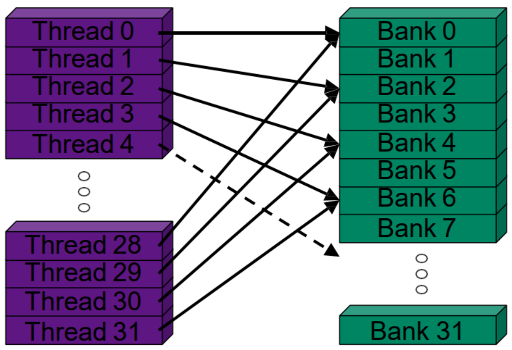


No Bank Conflicts

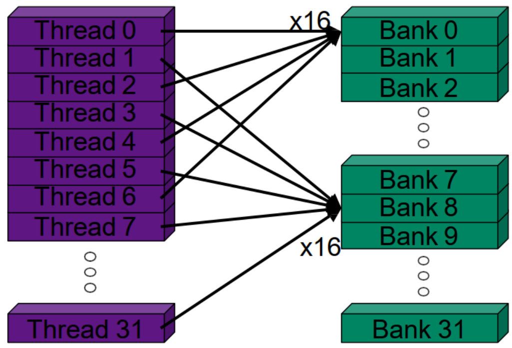


BANK ADDRESSING EXAMPLES

2-way Bank Conflicts



16-way Bank Conflicts

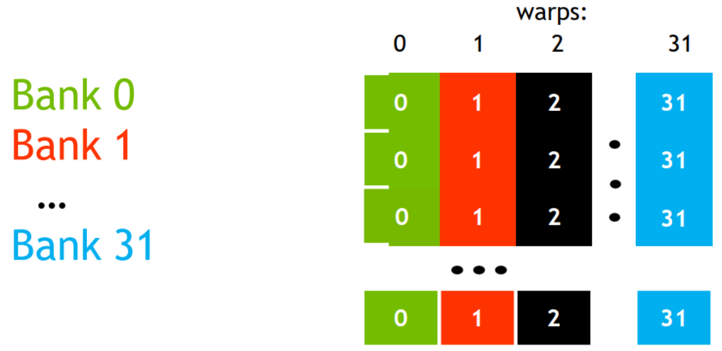


SHARED MEMORY: AVOIDING BANK CONFLICTS

共享内存：避免bank冲突

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

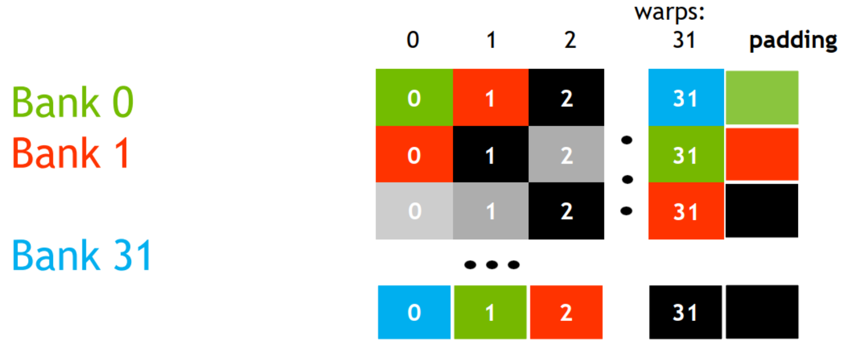
冲突



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》 见后面

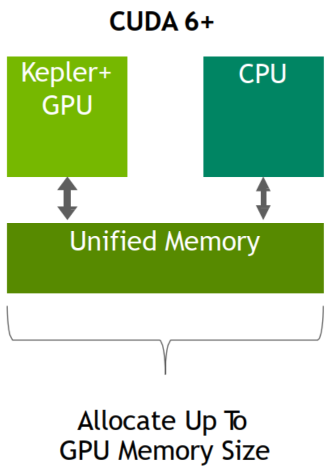
Page12-33 parallel reduction optimization

《课件-06_Managed_Memory》

Page 5-9 UNIFIED MEMORY

UNIFIED MEMORY 统一寻址

Reduce Developer Effort
减少开发人员的工作量



Simpler Programming & Memory Model
更简单的编程和内存模型

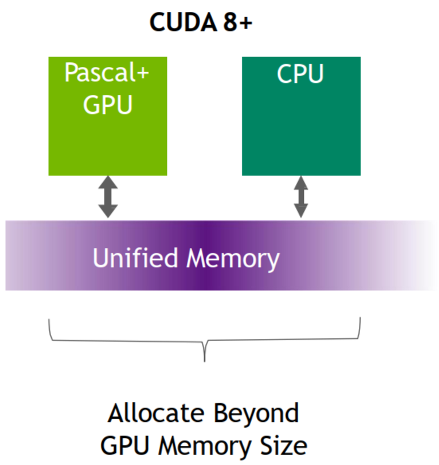
- Single allocation, single pointer, accessible anywhere
  单个分配，单个指针，在任何地方都可以访问
- Eliminate need for *explicit* copy
  消除对显式拷贝的需要
- Simplifies code porting
  简化了代码移植

Maintain Performance through Data Locality
通过数据本地化维护性能

- Migrate data to accessing processor
  将数据迁移到访问处理器
- Guarantee global coherence
  保证全局一致性
- Still allows explicit hand tuning
  仍然允许显式手动调优

CUDA 8+: UNIFIED MEMORY

Demand Paging For Pascal and Beyond
要求对Pascal和其他进行分页



Enable Large Data Models
enable大数据模型

- 超额认购GPU内存
  Oversubscribe GPU memory
- Allocate up to system memory size
  分配取决于系统内存大小

Simpler Data Accesss
简化数据访问

- CPU/GPU Data coherence CPU / GPU数据一致性
- Unified memory atomic operations
  统一存储原子操作

Tune Unified Memory Performance
调优 统一内存性能

- 通过cudaMemAdvise API使用提示
  Usage hints via cudaMemAdvise API
- Explicit prefetching API
  明确的预取API

SIMPLIFIED MEMORY MANAGEMENT CODE

简化的内存管理代码

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);
            (&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

按需分页

```
__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
统一内存分配

Access all values on CPU

Access one value on GPU