Lecture 6 – CUDA Memory

THANH TUAN DAO

VNU-UET 12/03/2025

Today Agenda

Register

Shared Memory

Global Memory

GPU Memory System

GPU memory hierarchy

- Register ~ few - 20 cycles
- L1 caches, L2 caches ~ 20-40 cycles
- Shared memory ~ 20-40 cycles
- Global Memory ~ up to few hundreds cycles

ARCHITECTURE SPECIFICATIONS	COMPUTE CAPABILITY			
	2.0	2.1	3.0	3.5
Load/Store address width	64 B		64 B	
L2 cache	768 K		1,536 K	
On-chip memory per multiprocessor	64 K		64 K	
Shared memory per multiprocessor (configurable)	48 K or 16 K		48 K/32 K/16 K	
L1 cache per multiprocessor (configurable)	16 K or 48 K		48 K/32 K/16 K	
Read-only data cache	N/A		48 K	
Global memory	Up to 6 GB		Up to 12 GB	

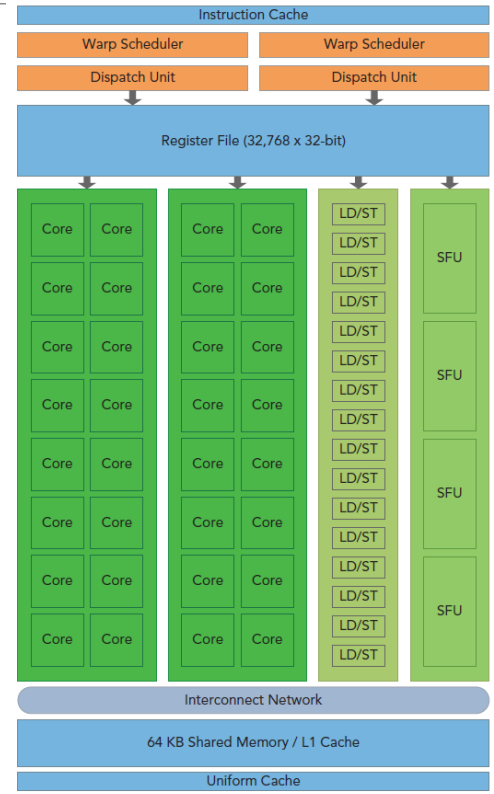


FIGURE 3-1

A Fermi Architecture

CUDA Example: Find the GPU registers

```
#include <iostream>
#include <cuda_runtime.h>

#define BLOCK_SIZE 256

// Naive Reduction (No Shared Memory)
__global__ void reductionNaive(float *input, float *output, int N) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    for (int stride = 1; stride < N; stride *= 2) {
        if (tid % (2 * stride) == 0 && tid + stride < N) {
            input[tid] += input[tid + stride];
        }
        __syncthreads();
    }
    if (tid == 0) {
        *output = input[0];
    }
}

// Optimized Reduction Using Shared Memory
__global__ void reductionShared(float *input, float *output, int N) {
    __shared__ float temp[BLOCK_SIZE];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int local_tid = threadIdx.x;

    temp[local_tid] = (tid < N) ? input[tid] : 0.0f;
    __syncthreads();

    // Reduction within block
    for (int stride = blockDim.x / 2; stride > 0; stride /= 2) {
        if (local_tid < stride) {
            temp[local_tid] += temp[local_tid + stride];
        }
        __syncthreads();
    }

    if (local_tid == 0) {
        atomicAdd(output, temp[0]);
    }
}
```

```
int main() {
    int N = 1 << 20; // 1 million elements
    size_t size = N * sizeof(float);

    // Allocate host memory
    float *h_input = (float *)malloc(size);
    float h_output = 0.0f;

    // Initialize input array
    for (int i = 0; i < N; i++) {
        h_input[i] = 1.0f;
    }

    // Allocate device memory
    float *d_input, *d_output;
    cudaMalloc((void **)&d_input, size);
    cudaMalloc((void **)&d_output, sizeof(float));

    // Copy data to device
    cudaMemcpy(d_input, h_input, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_output, &h_output, sizeof(float), cudaMemcpyHostToDevice);

    // Launch naive kernel
    int blocksPerGrid = (N + BLOCK_SIZE - 1) / BLOCK_SIZE;
    reductionNaive<<<blocksPerGrid, BLOCK_SIZE>>>(d_input, d_output, N);
    cudaMemcpy(&h_output, d_output, sizeof(float), cudaMemcpyDeviceToHost);
    std::cout << "Naive Reduction result: " << h_output << std::endl;

    // Reset output and launch shared memory optimized kernel
    h_output = 0.0f;
    cudaMemcpy(d_output, &h_output, sizeof(float), cudaMemcpyHostToDevice);
    reductionShared<<<blocksPerGrid, BLOCK_SIZE>>>(d_input, d_output, N);
    cudaMemcpy(&h_output, d_output, sizeof(float), cudaMemcpyDeviceToHost);
    std::cout << "Shared Memory Reduction result: " << h_output << std::endl;

    // Free memory
    cudaFree(d_input);
    cudaFree(d_output);
    free(h_input);

    return 0;
}
```

GPU Registers

Each SM has a fixed amount of registers

- Allocated (partitioned) to active threads
- Will be optimized by the compiler
- If the threads use more registers than available, the compiler will *spill* to local memory
  - Local memory can be cached by L1, L2 and global memory

The register usage may be limited by `nvcc –maxrregcount amount`

Using register smartly can greatly improve your programs!

GPU Shared Memory

Each SM has a fixed amount of shared memory

- Allocated (partitioned) to active blocks
- Threads in a block can access (read/write) to the allocated shared memory
- Threads in a block use block-level synchronization to guarantee memory consistency

Shared memory can be declared using either static or dynamic method

- Static: `__shared__ float temp[BLOCK_SIZE];`
- Dynamic: `extern __shared__ float temp[]; ... Kernel <<<grid, block, shared_mem_size>>>(...);`

Using shared memory smartly can greatly improve your programs!

GPU Shared Memory: Bank Conflict (1)

Shared memory is divided into 32 equally-sized banks (according to 32 threads in warp)

- Accesses to different banks can be serviced simultaneously
- A *memory transaction* can access one *location per bank*

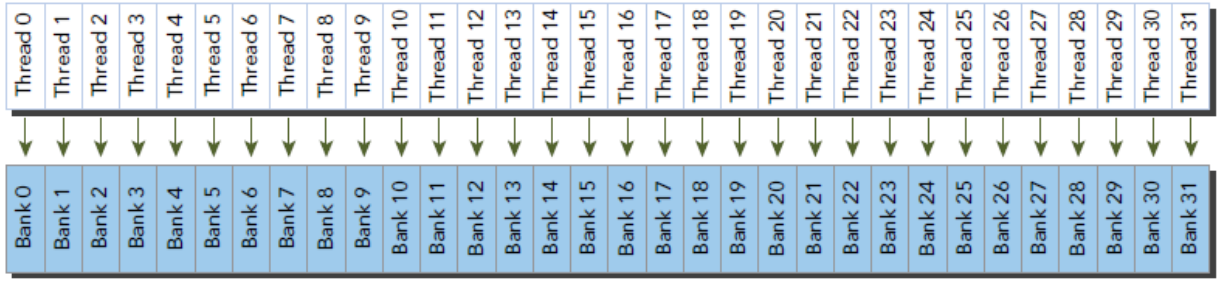
Three types of situations may happen when a request to shared memory issued by a warp

- Parallel access: Multiple addresses access across multiple banks
- Serial access: Multiple addresses access within the same bank
- Broadcast access: A single address read in a single bank

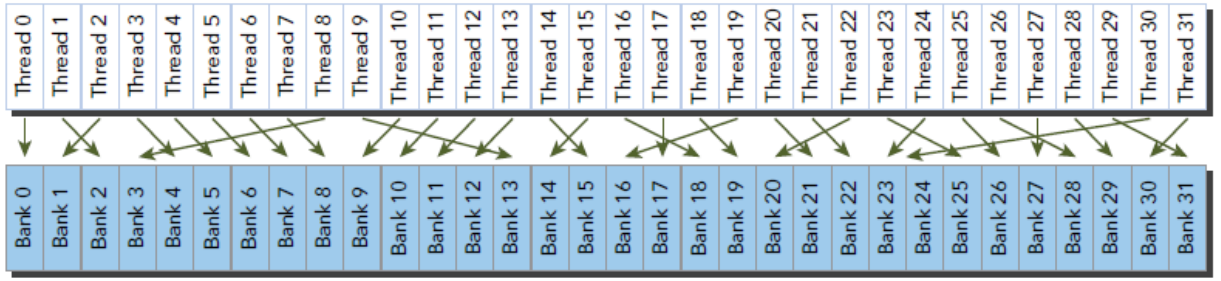
Bank conflicts arise when threads access different addresses in a bank

GPU Shared Memory: Bank Conflict (2)

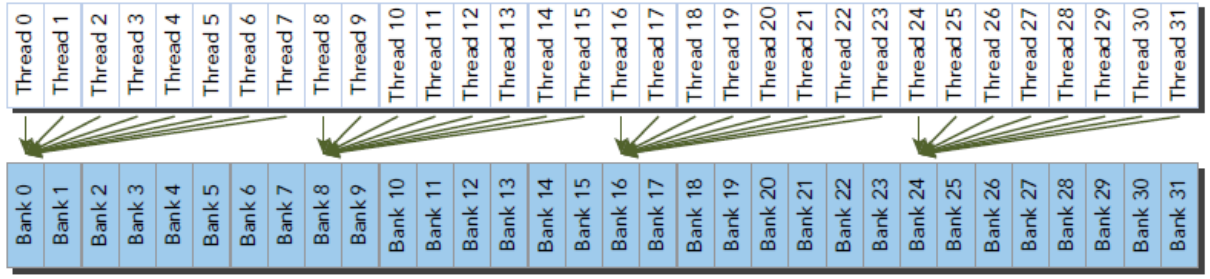
Parallel accesses



Serial accesses



Serial accesses (not necessarily broadcasting)



Mapping Address to Bank Index

Bank width is the number of bytes within the same bank

bank index = (byte address ÷ 4 bytes/bank) % 32 banks

bank index = (byte address ÷ 8 bytes/bank) % 32 banks

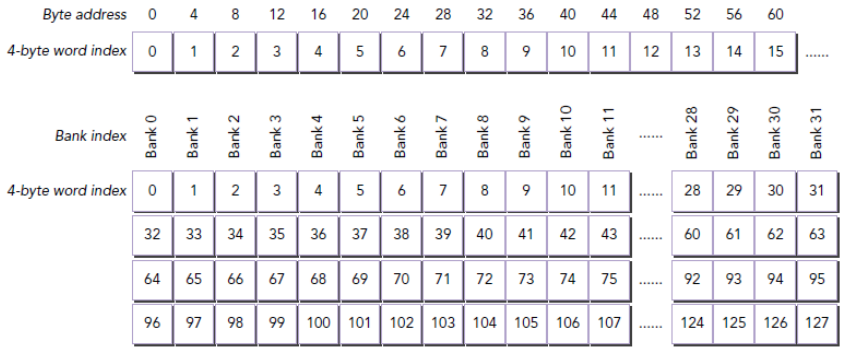
A bank conflict does not occur when two threads from the *same warp* access the *same address*.

Read access will be broadcast

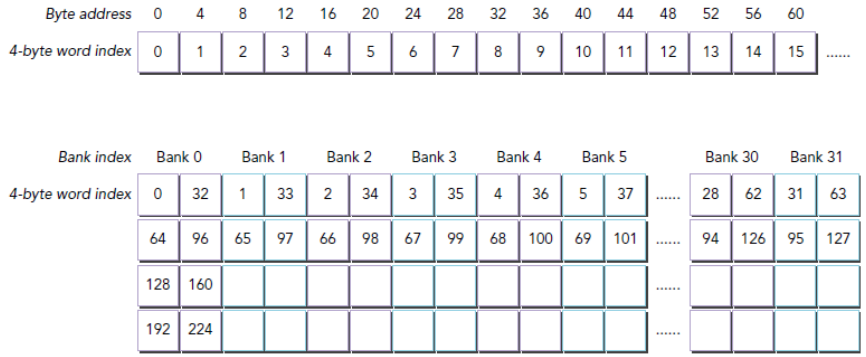
Write access will be written by one of the threads

Two modes: 32-bit mode and 64-bit mode (Kepler and later arch support both)

32-bit mode



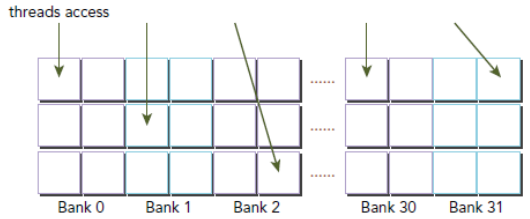
64-bit mode



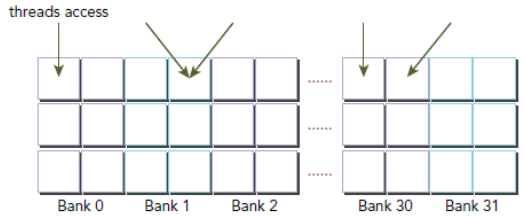
Bank Conflicts Examples

64-bit mode

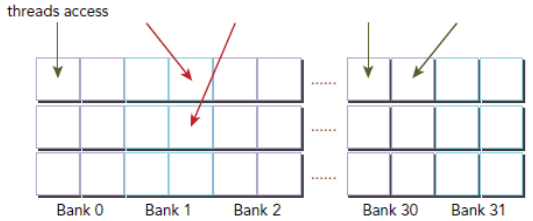
Conflict-free: Each thread access to a different bank



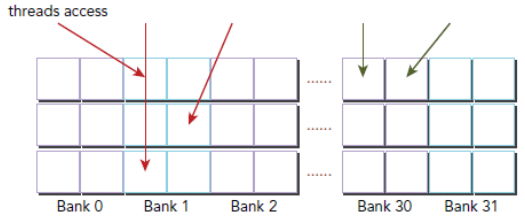
Conflict-free: Two threads accessing the same 8-byte word



2-way conflict



3-way conflict



Synchronization

Block-level synchronization

`__synchthreads()`: Wait for all threads in a block to arrive at this point

Affect all threads in the same block

Atomic operations

E.g., `float atomicAdd(float* addr, float amount)`

Perform read-modify-write atomic operations on global or shared memory

Program-level synchronization

E.g., `cudaDeviceSynchronize()`

Synchronizes operations (kernel launches, memory copies) in a program

Memory Fences

Ensure any memory write before the fence is visible to other threads after the fence

```
void __threadfence(); void __threadfence_block(); void __threadfence_system();
```

GPU Global Memory

Allocated by `cudaMalloc` in the host

Resides in device memory

Accessible via 32-byte, 64-byte or 128-byte memory transactions

The number of transactions required by a request depends on

- Distribution of memory addresses across threads in a warp
- Alignment of requested memory addresses

Optimizing the number of memory transactions is crucial for performance

Global memory access can be cached by

- L1, L2, read-only constant, read-only texture caches

Static Global Memory

Use `cudaMemcpyToSymbol` to transfer the data from host to device

```
#include <cuda_runtime.h>
#include <stdio.h>
__device__ float devData;

__global__ void checkGlobalVariable() {
  // display the original value
  printf("Device: the value of the global variable is %f\n",devData);
  // alter the value
  devData +=2.0f;
}
int main(void) {
  // initialize the global variable
  float value = 3.14f;
  cudaMemcpyToSymbol(devData, &value, sizeof(float));
  printf("Host: copied %f to the global variable\n", value);
  // invoke the kernel
  checkGlobalVariable <<<1, 1>>>();
  // copy the global variable back to the host
  cudaMemcpyFromSymbol(&value, devData, sizeof(float));
  printf("Host: the value changed by the kernel to %f\n", value);
  cudaDeviceReset();
  return EXIT_SUCCESS;
}
```

Memory Allocation and Transfer

Use `cudaMemcpy` to transfer the data from host to device

```
#include <cuda_runtime.h>
#include <stdio.h>
int main(int argc, char **argv) {
  // set up device
  int dev = 0;
  cudaSetDevice(dev);
  // memory size
  unsigned int isize = 1<<22;
  unsigned int nbytes = isize * sizeof(float);
  // get device information
  cudaDeviceProp deviceProp;
  cudaGetDeviceProperties(&deviceProp, dev);
  printf("%s starting at ", argv[0]);
  printf("device %d: %s memory size %d nbyte %5.2fMB\n", dev,
  deviceProp.name,isize,nbytes/(1024.0f*1024.0f));
  // allocate the host memory
  float *h_a = (float *)malloc(nbytes);
  // allocate the device memory
  float *d_a;
  cudaMalloc((float **)&d_a, nbytes);
  // initialize the host memory
  for(unsigned int i=0;i<isize;i++) h_a[i] = 0.5f;
  // transfer data from the host to the device
  cudaMemcpy(d_a, h_a, nbytes, cudaMemcpyHostToDevice);
  // transfer data from the device to the host
  cudaMemcpy(h_a, d_a, nbytes, cudaMemcpyDeviceToHost);
  // free memory
  cudaFree(d_a);
  free(h_a);
  // reset device
  cudaDeviceReset();
  return EXIT_SUCCESS;
}
```

Pinned memory

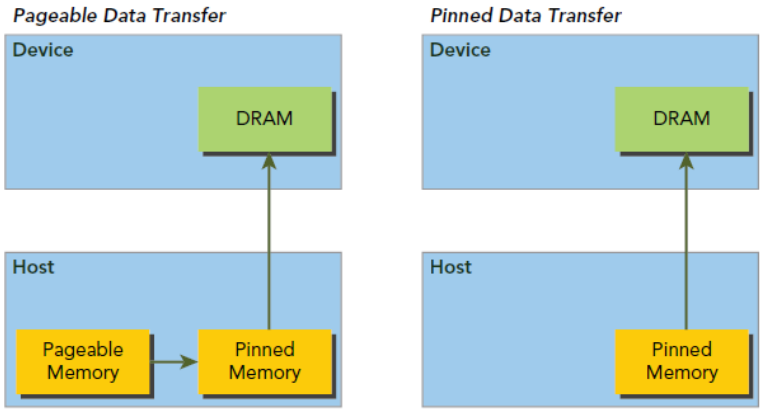
Default host memory is *pageable*

Can be paged out (due to fault operations) any time by the OS

GPU transfers data from pageable host memory to pinned (or page-locked) memory before transferring to GPU memory

Use `cudaMallocHost` to allocate pinned host memory

Using pinned memory is more expensive, but can copy faster



Zero-copy memory

Zero-copy memory is pinned memory that is mapped into the device address space

- No explicit copy is required
- The memory transfer is issued based on the requests in the kernel
- Slow when frequently used in the kernel

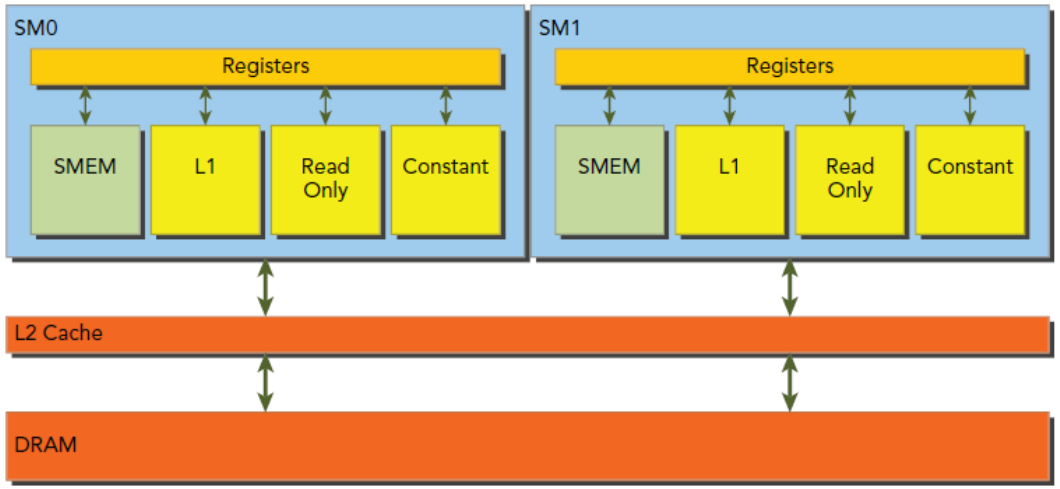
HW: rewrite VecAdd example with pinned memory, zero-copy memory and

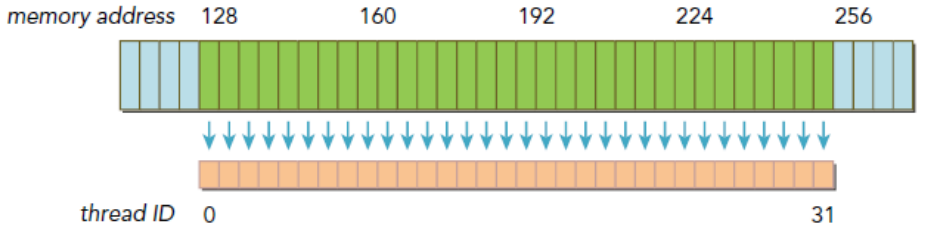
- Compare the performance
- Write a report to the TA

Memory Alignment and Coalescing

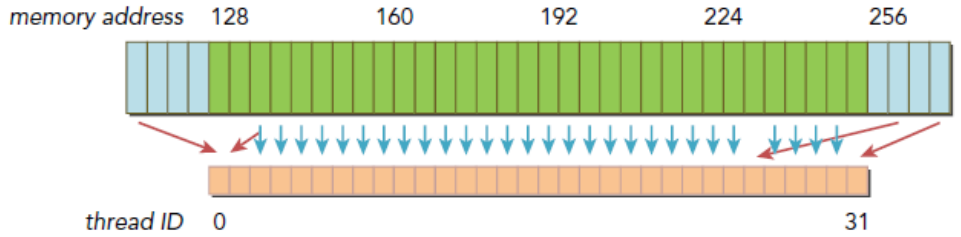
Aligned memory accesses occur when the first address of a device memory transaction is an even multiple of the cache granularity being used to service the transaction (32 bytes for L2 or 128 bytes for L1)

Coalesced memory access occur when all 32 threads in a warp access a contiguous chunk of memory (a memory segment)





Aligned and coalesced case



Misaligned and uncoalesced case

Thank you!