CUDA Memory Model 2

Prof. Seokin Hong

Agenda

- **Matrix Multiplication**
  - **Basic Version**
  - **Tiled Version**

- Review: Memory Hierarchy
- Importance of Memory Access Efficiency

- **GPU Memory Hierarchy**
- **Improving Tiled Matrix Multiplication**
- **Impact of Memory on Parallelism**

GPU Memory Hierarhcy

Review: Thread Organization

- Hierarchical Structure
- Grid, block, thread

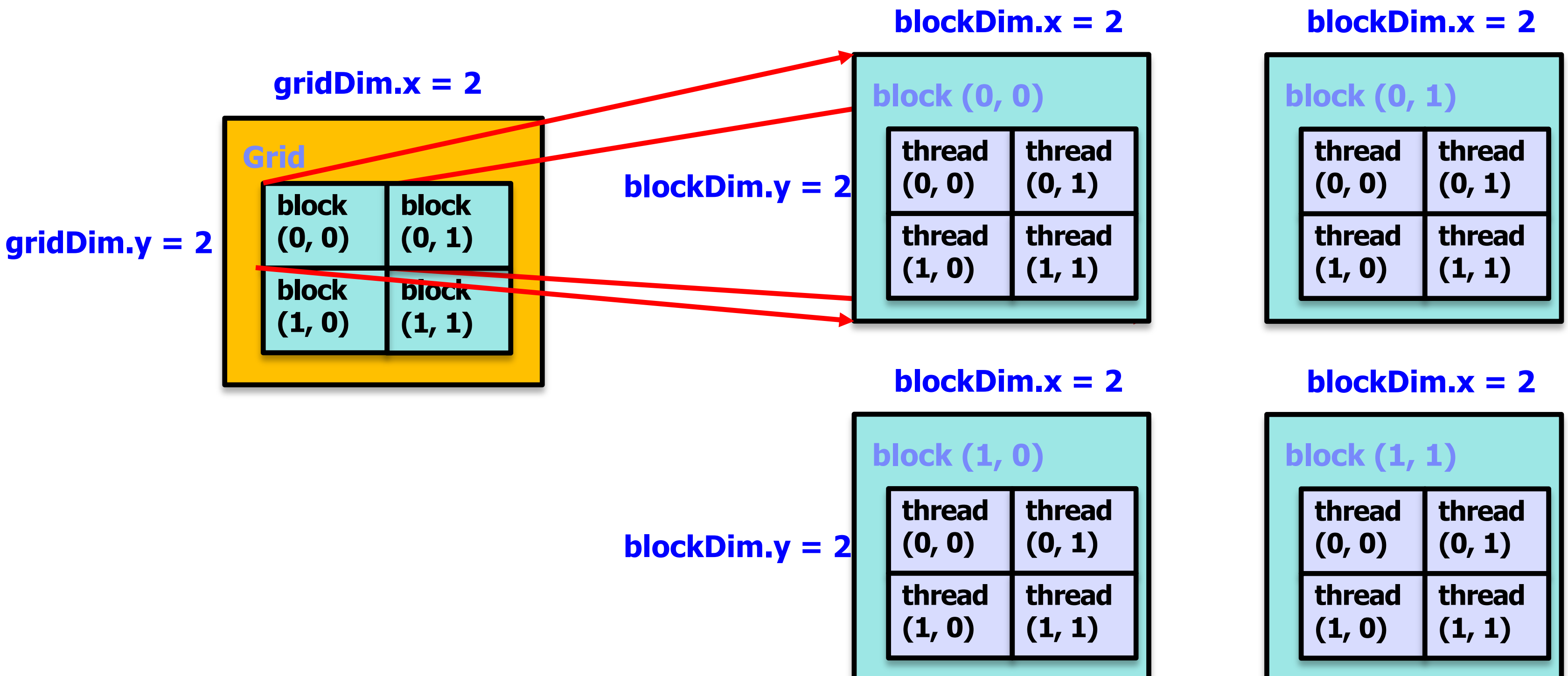
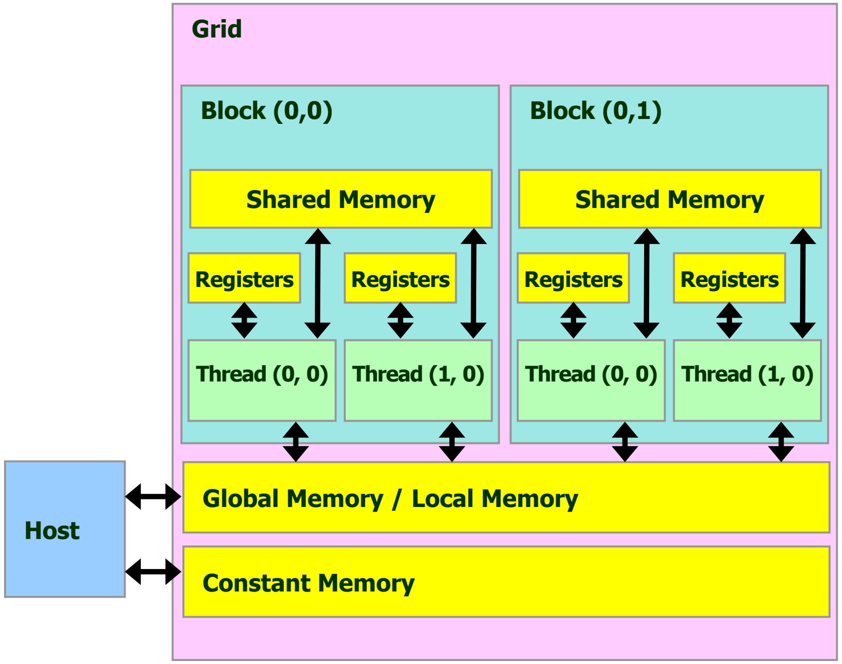


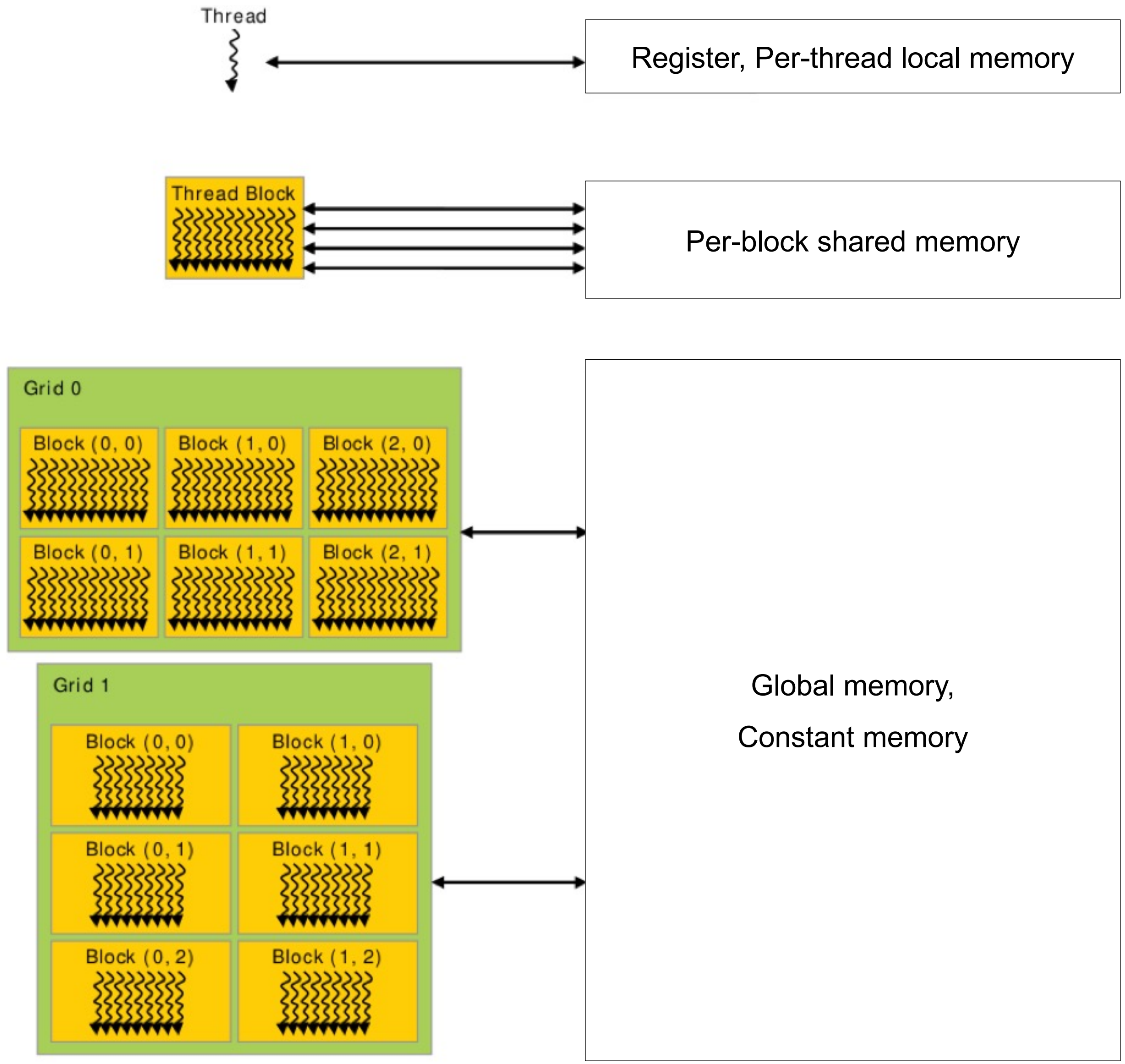
Image from http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx

Programmer's View of CUDA Memories

- Each thread can:
  - per-thread **registers**
    - (~1 cycle)
  - per-block **shared memory**
    - (~5 cycles)
  - per-grid **global memory**
    - (~500 cycles)
  - per-thread **local memory**
    - (~500 cycles)
    - actually, located on the global memory
  - per-grid **constant memory**
    - (~5 cycles with caching)
    - Read-only, allocated by host on device, cached on on-chip memory (fast)

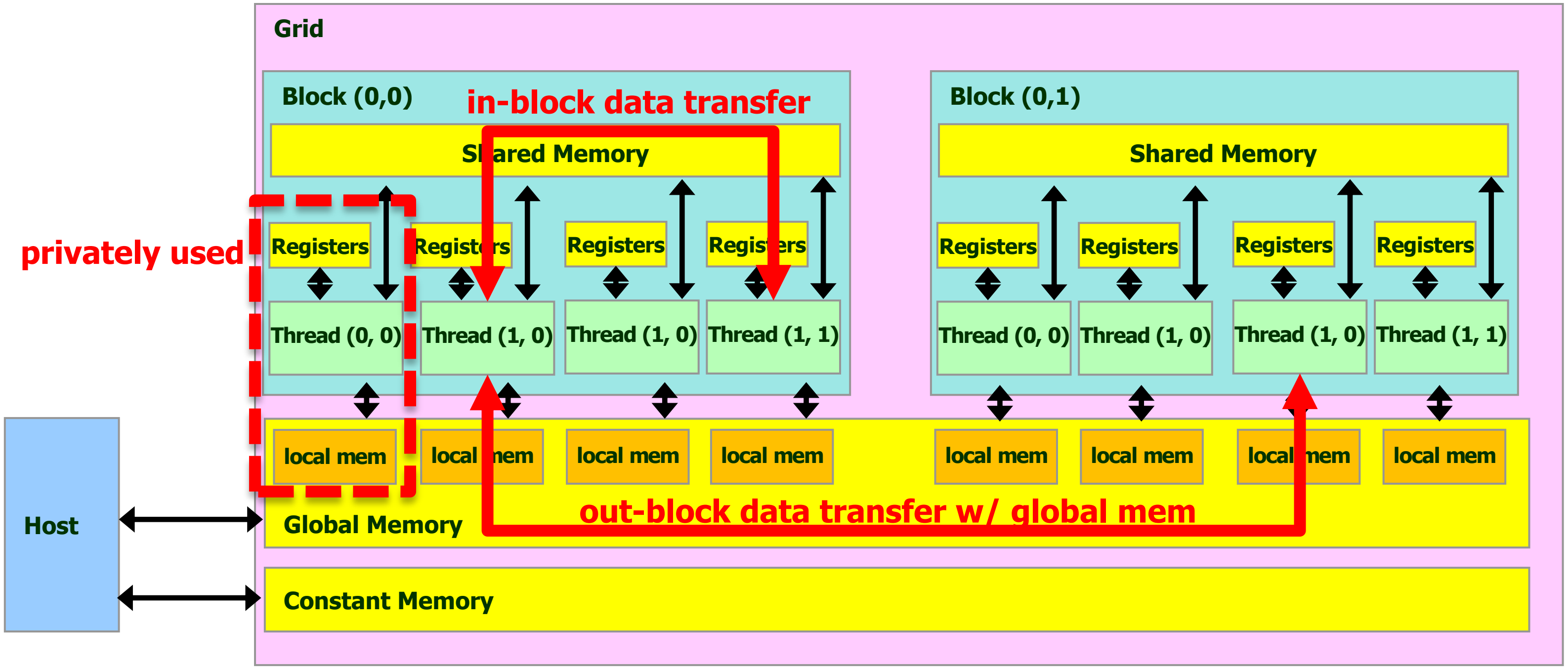


Programmer's View of CUDA Memories

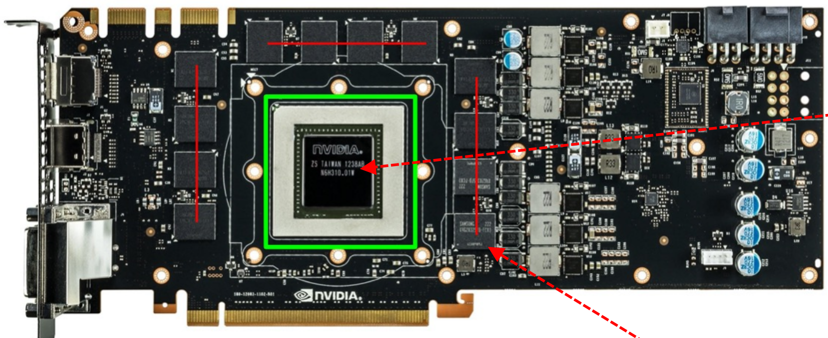


Programmer’s View of CUDA Memories

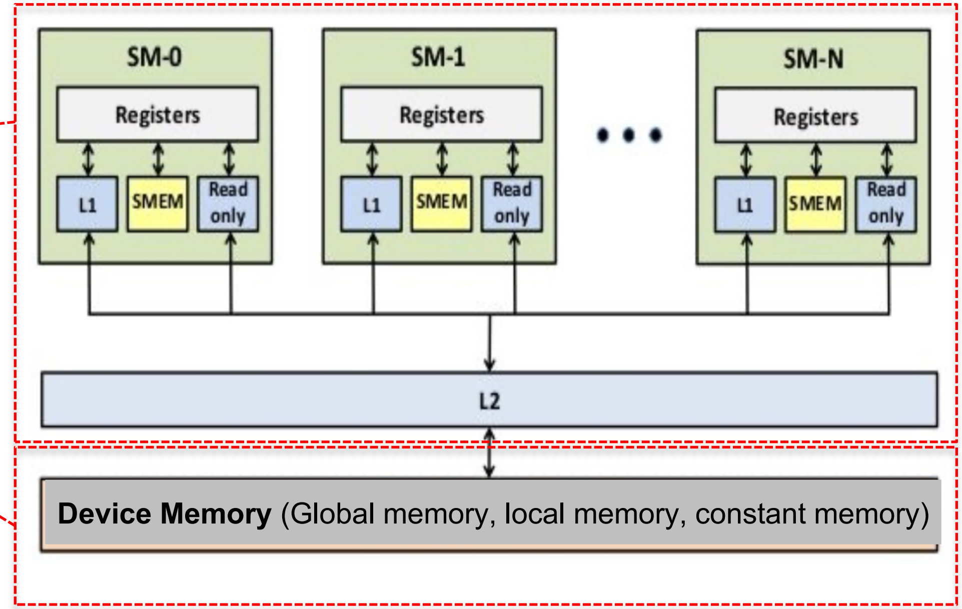
- Each thread has its own private registers and local memory
- **Threads within a block share the shared memory**
  - Shared memory is used for Inter-thread communication within a block
- **Thread blocks share the global memory**
  - Global memory is used for Inter-block (or inter-grid) communication



Architectural View of CUDA Memories



Green box is GK110, red lines are global memory

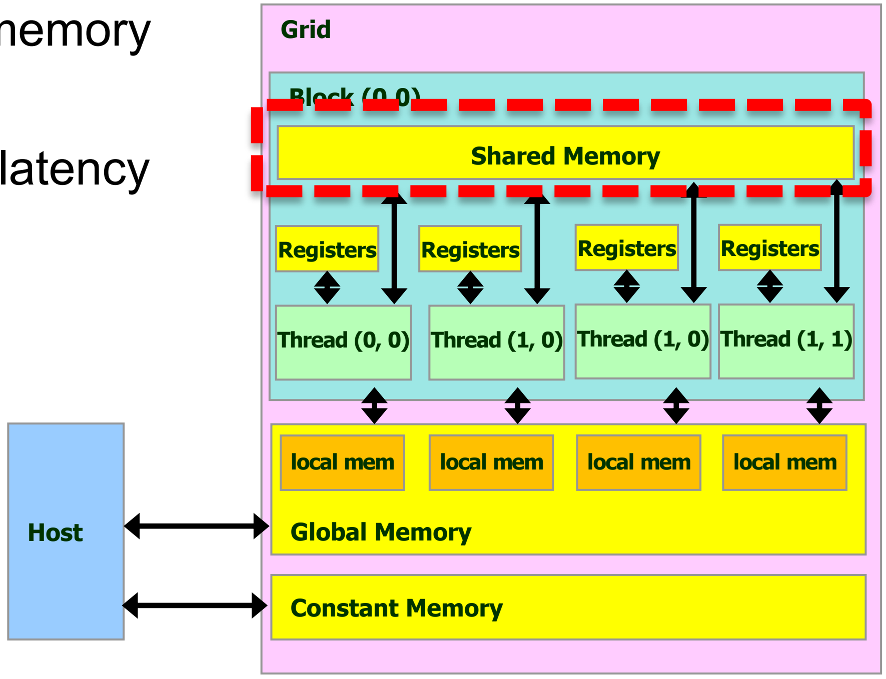


	Global Memory	Shared Memory	Constant Memory	Register
Capacity	16GB (Device memory)	64KB/SM	64KB/SM (Device memory)	256KB / SM
Cache	L1, L2	None	Special cache	None
Access	GPU-wide	SM-wide	GPU-wide (cached on each SM)	Private to each thread
Latency	200~400 cycles	1~5 cycle	1 cycle (hit), 200~400 cycle (miss)	1 cycle

* Numbers can be different depending on GPU architecture

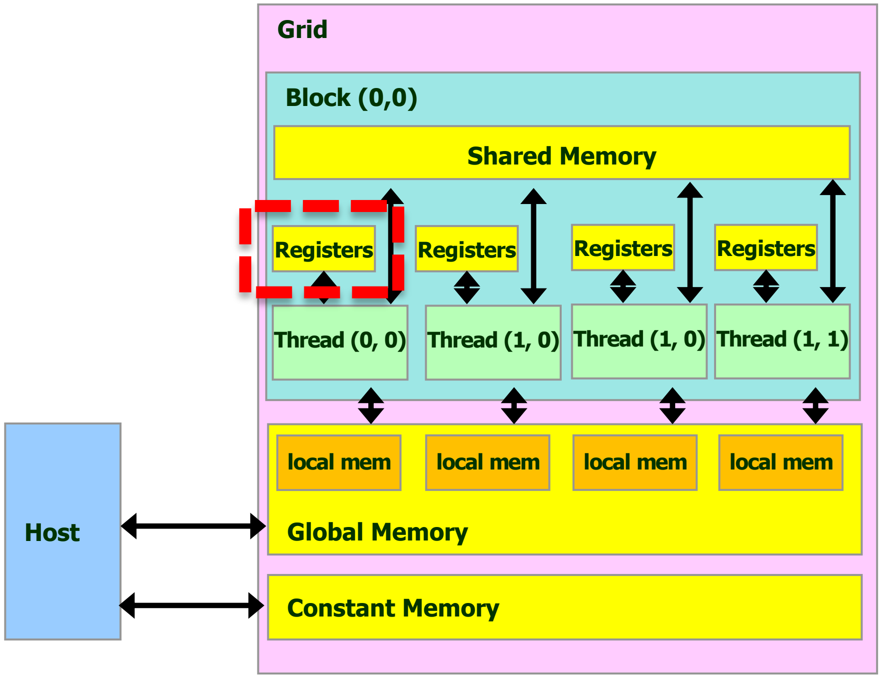
Shared Memory

- A special type of memory whose contents are explicitly declared
  - **Shared by threads of the same block**
  - Located in the processor (SM)
  - Accessed **at much higher speed** (in both latency and throughput)
  - **Shared memory is partitioned among the blocks**
  - Commonly referred to as scratchpad memory in computer architecture
  - Same hardware as L1 cache, ~5ns of latency



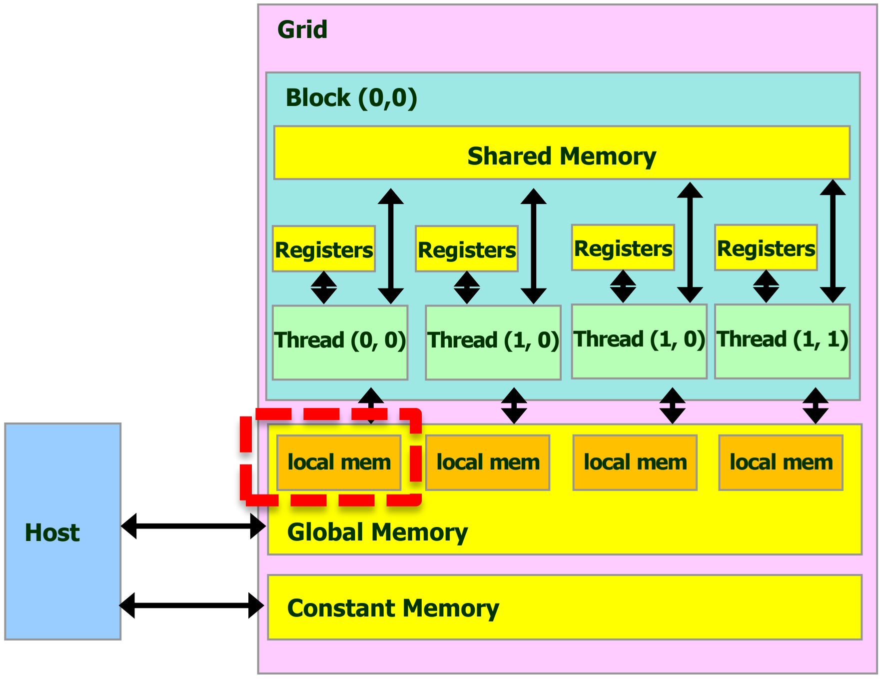
Registers

- **Fastest memory**
  - **Private per thread**
  - Located in the processor (SM)
  - About 10x faster than shared memory
  - **Registers are partitioned among the threads**
  - Most local variables declared in kernels are stored in registers (e.g., float x)



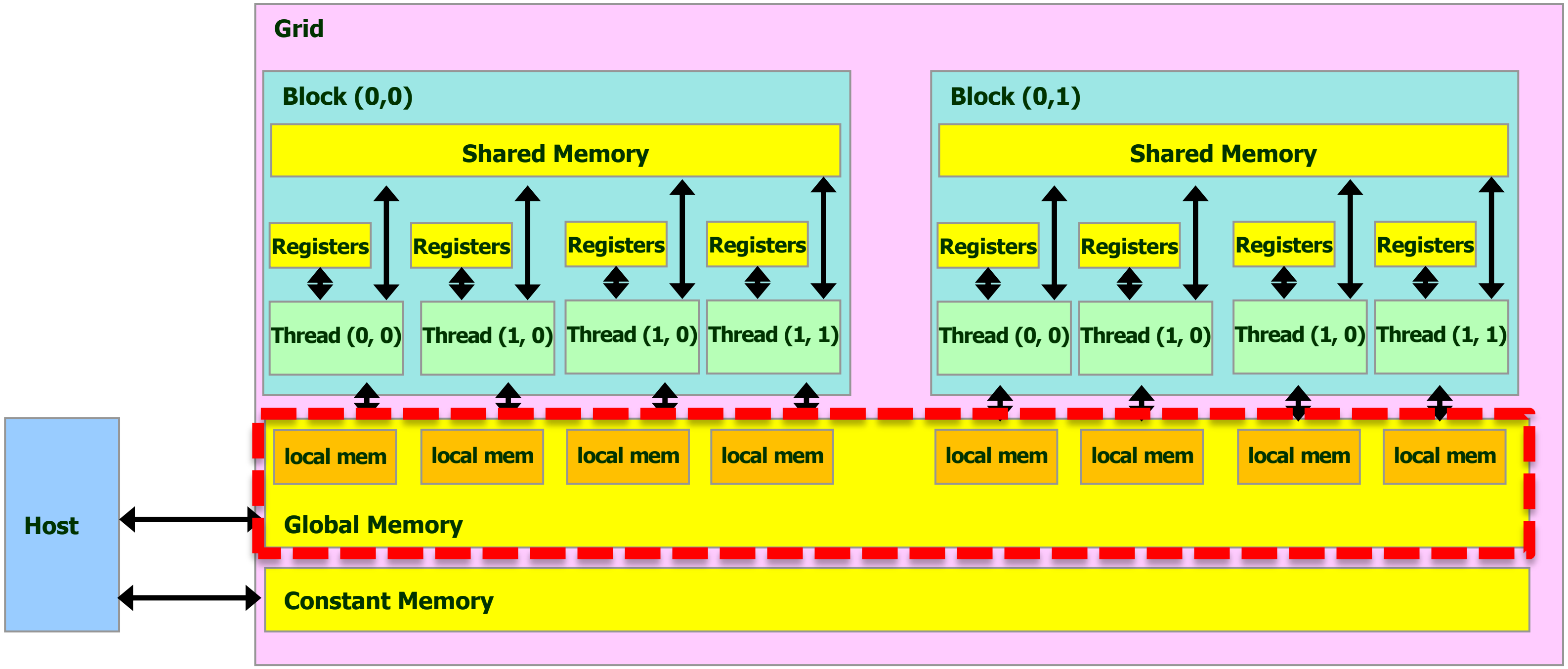
Local Memory

- Local memory stores data that can’t fit in registers
  - **Private per thread**
  - Stored in the global memory → much slower than registers
  - Used for local variables, register spilling



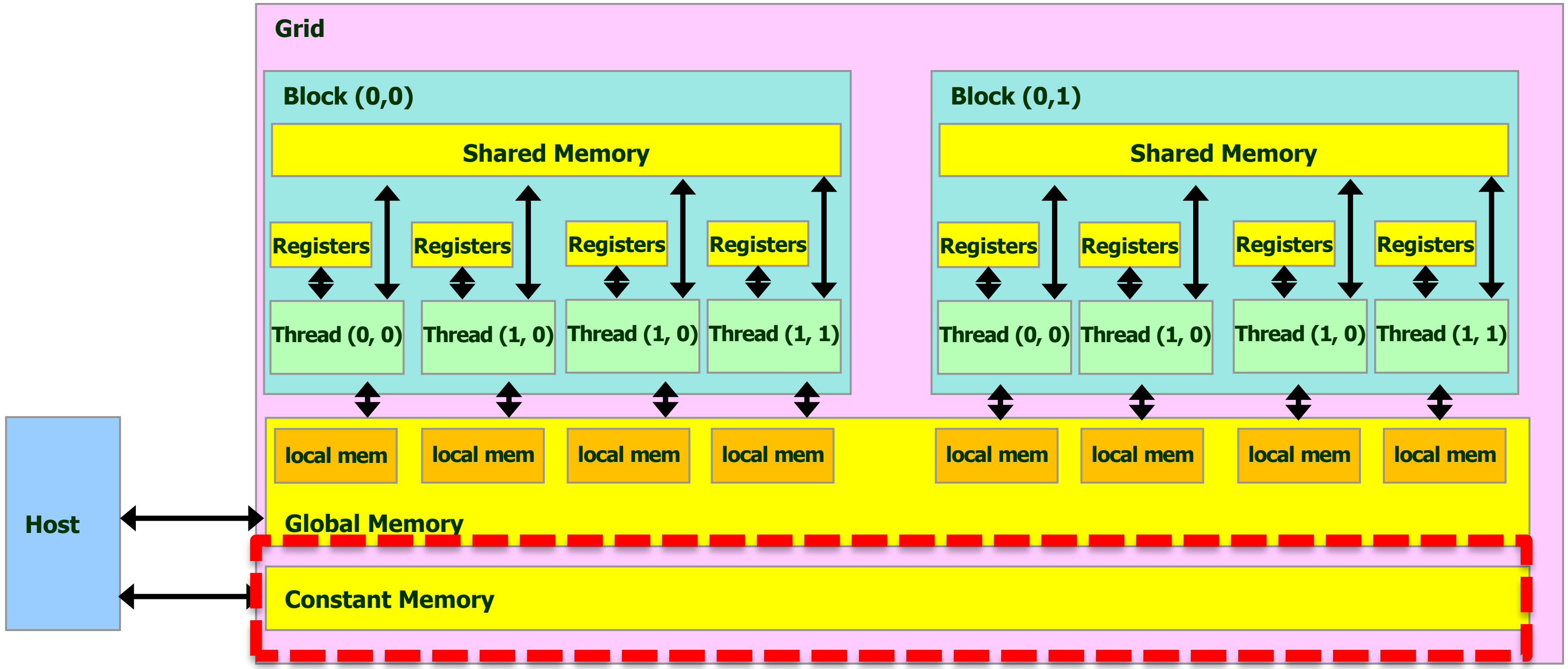
Global Memory

- Global memory is separate hardware from the GPU core
  - **Shared by all threads**
  - Located in off-chip device memory
    - → much slower than registers
  - Majority of data is in global memory
  - Both Host and GPU can access



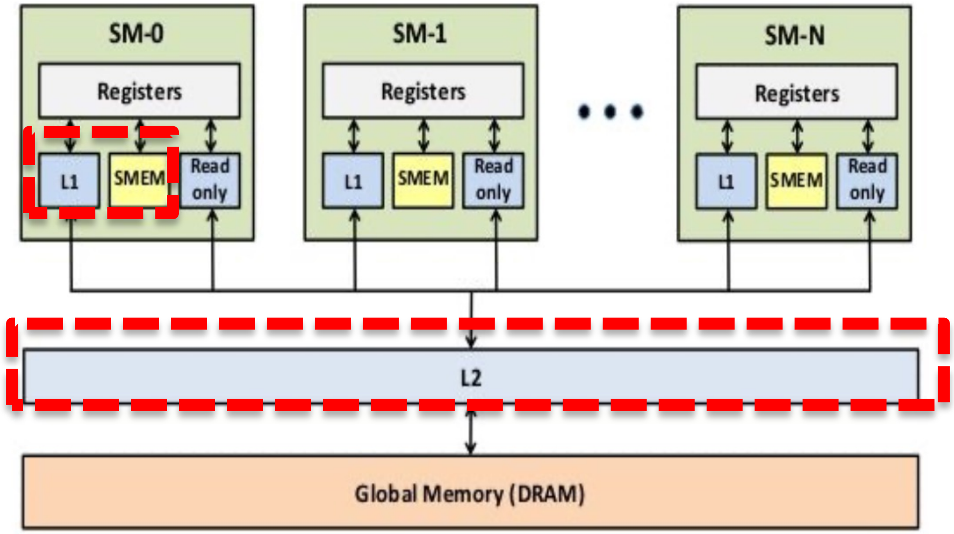
Constant Memory

- **Used for constant values** (read-only data)
  - **Shared by all threads**
  - Located in off-chip device memory
    - → much slower than registers
  - Both Host and GPU can access
  - Constants must be set from host before running kernel



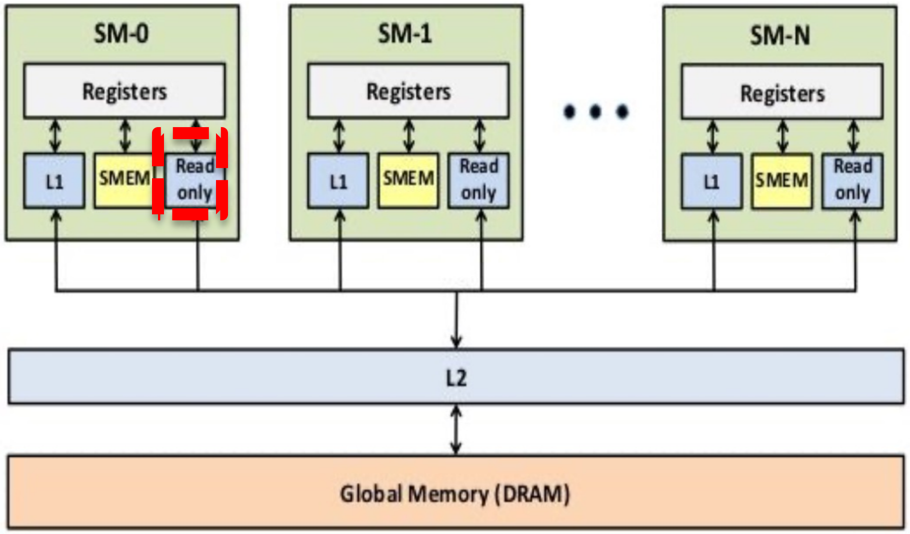
L1/L2 Cache (Hardware-managed)

- **On-chip Cache memory**
  - Store recently accessed data in the global memory
    - Store local & global memory data
  - **L1 Cache**
    - Same hardware as shared memory
    - Configurable size (16, 32, 48KB)
    - Each SM has its own L1 cache
  - **L2 Cache**
    - 1MB~ in size
    - Shared by all SM's



Constant Cache (Hardware-managed)

- **On-chip Cache memory**
  - Store recently accessed constant in the constant memory
  - Hardware-managed



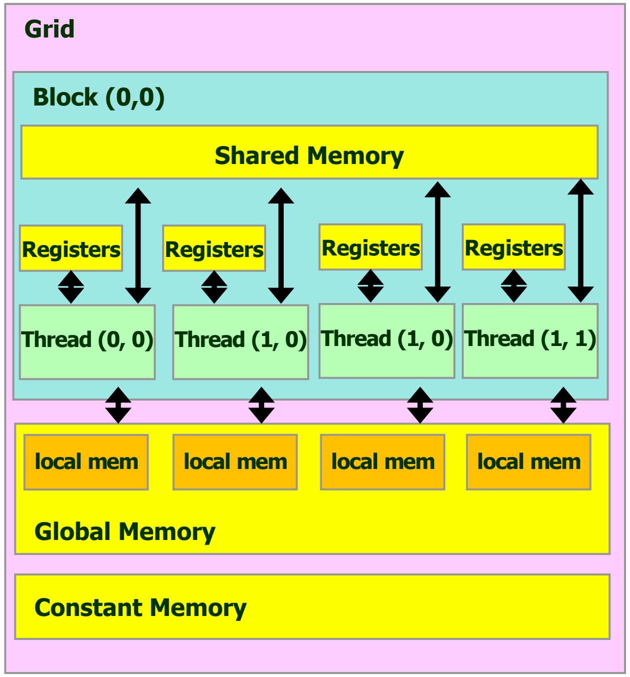
GPU Memory in AWS server

```
seokin@ip-172-31-41-4:/usr/local/cuda/samples/1_Utilities/deviceQuery$ ./deviceQuery
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "Tesla T4"
  CUDA Driver Version / Runtime Version          11.0 / 11.0
  CUDA Capability Major/Minor version number:    7.5
  Total amount of global memory:                 15110 MBytes (15843721216 bytes)
  (40) Multiprocessors, ( 64) CUDA Cores/MP:     2560 CUDA Cores
  GPU Max Clock rate:                            1590 MHz (1.59 GHz)
  Memory Clock rate:                             5001 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 4194304 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1024
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 3 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 0 / 30
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
int var;	register	thread	thread
int array_var[10];	local	thread	thread
__shared__ int shared_var;	shared	block	block
__device__ int global_var;	global	grid	application
__constant__ int constant_var;	constant	grid	application

- scalar variables without qualifier reside in a register
  - compiler will spill to thread-local memory
- array variables without qualifier reside in local memory

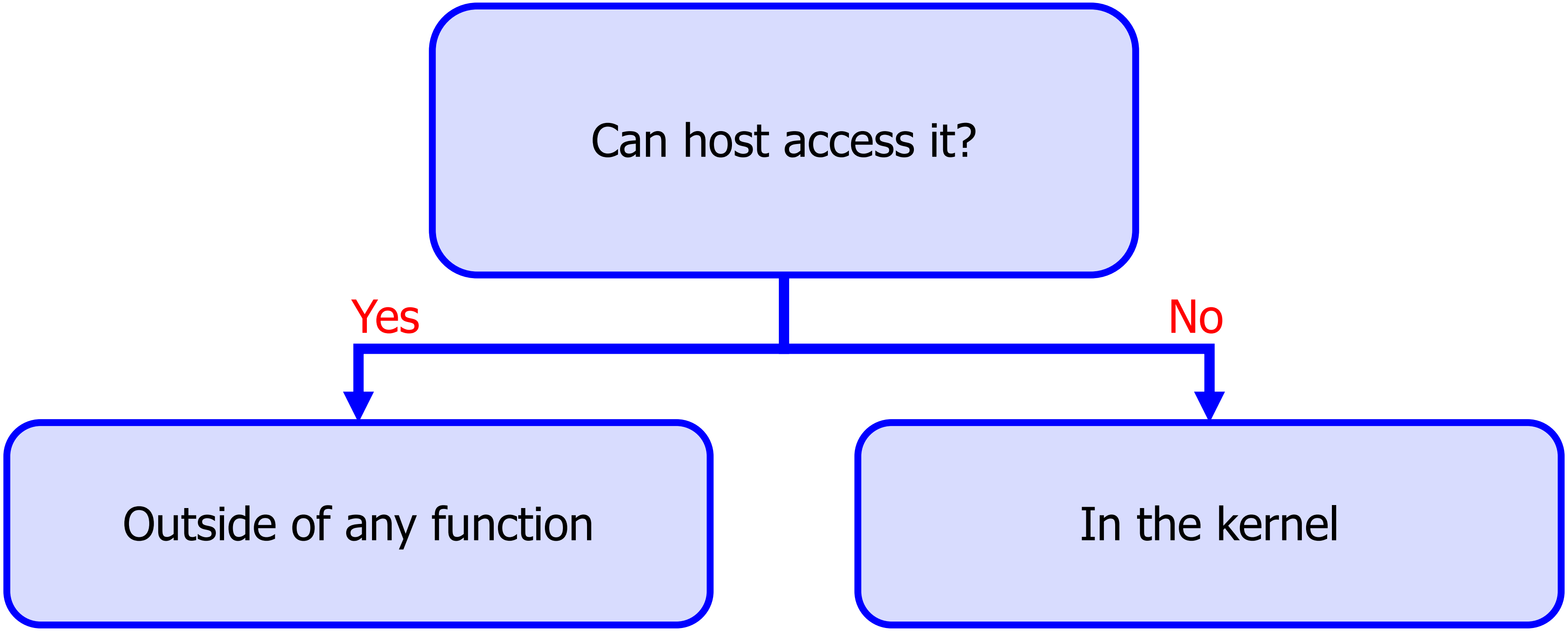


CUDA Variable Type Performance

Variable declaration	Memory	Penalty
int var;	register	1x
int array_var[10];	local	100x
__shared__ int shared_var;	shared	1x
__device__ int global_var;	global	100x
__constant__ int constant_var;	constant	1x

- scalar variables reside in on-chip registers → fast
- shared variables reside in on-chip memories → fast
- Thread-local & global variables reside in uncached off-chip memory → slow
- Thread-local & global variables reside in on-chip cache memory (L1, L2) → fast
- constant variables reside in on-chip cache memory (constant cache) → fast

Where to declare variables?

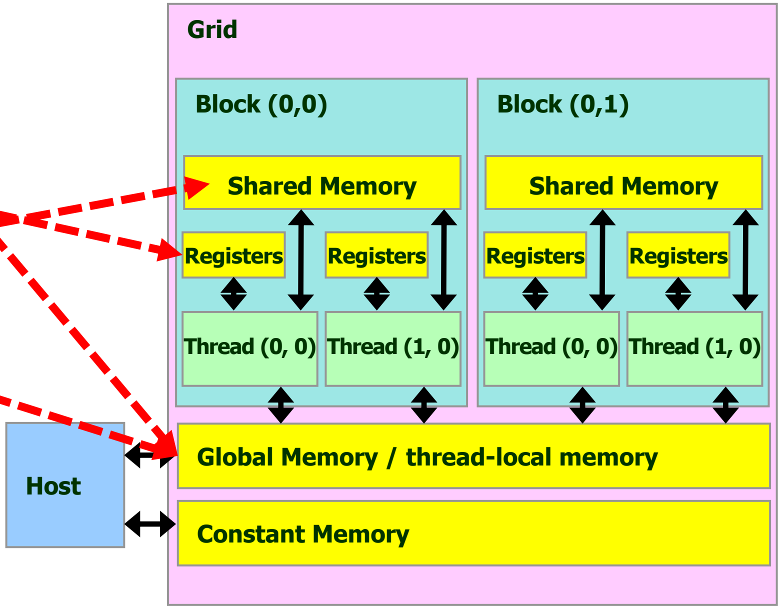



```
__constant__ int constant_var;
__device__   int global_var;
```

```
int var;
int array_var[10];
__shared__   int shared_var;
```

Example: thread-local variables

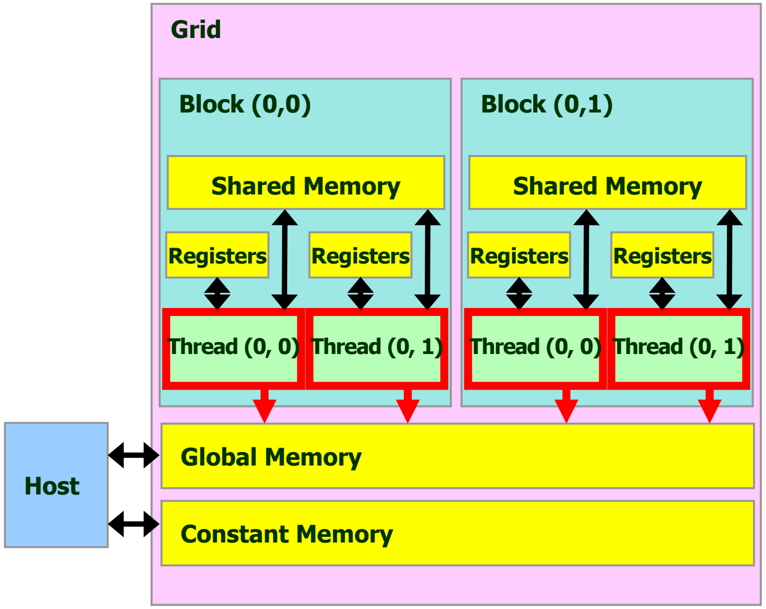
```
__global__   void   kernelFunc(float* dst,  const float*  src) {
   // p goes in a register
   float  p = src[threadIdx.x];
   // per-thread heap goes in off-chip memory
   float  heap[10];
   // shared variables
   __shared__   float   partial_sum[1024];
   // now actions
   …
}
```



Race Condition: Global Memory Case

- Question:

```
__global__  void  raceGlobal( int*  dst ) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    dst[0] = idx;
    // what is the value of dst[0] ?
}
```

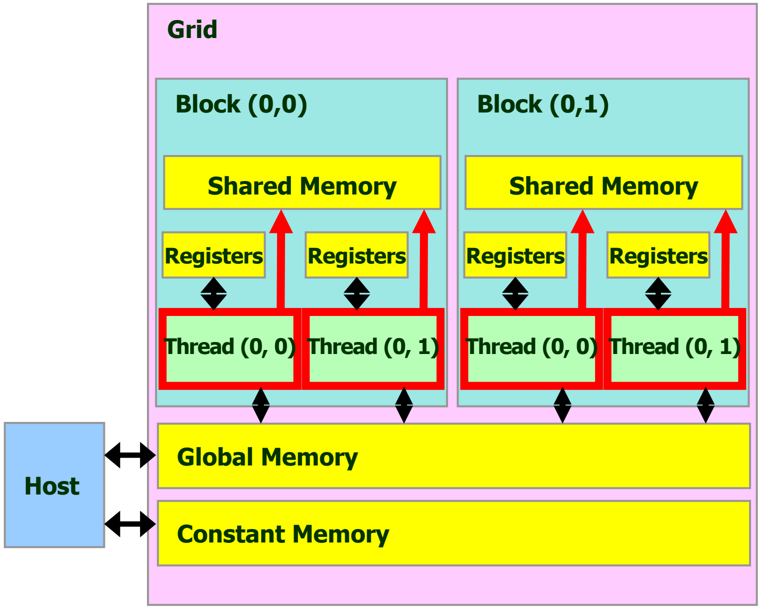


- undecidable !

Race Condition: Global Memory Case

- Question:

```
__global__  void  raceShared(void) {
    __shared__ int  shared_dst;
    shared_dst = threadIdx.x;
    // what is the value of shared_dst ?
}
```



- undecidable !

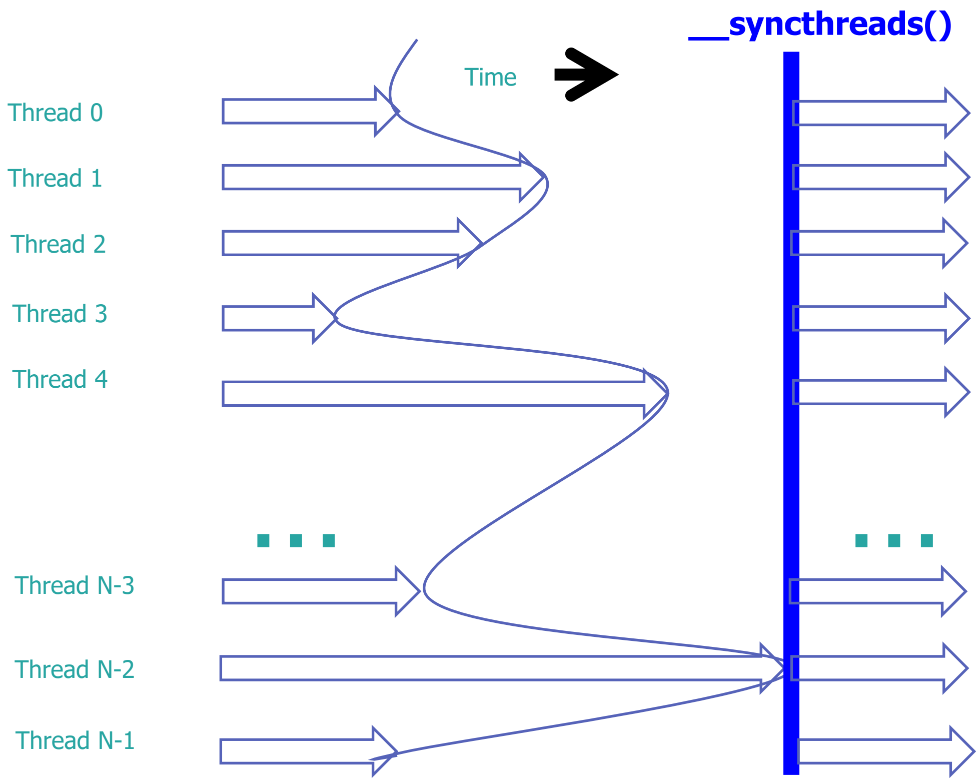
Communication Through Memory

- The order in which threads access the variable is undefined without explicit coordination

- Use barriers
  - for shared variables
  - __syncthreads() function in CUDA
  - All threads in the same block must reach the __syncthreads() before any can move on

- or atomic operations
  - for shared variables and global variables → explained later

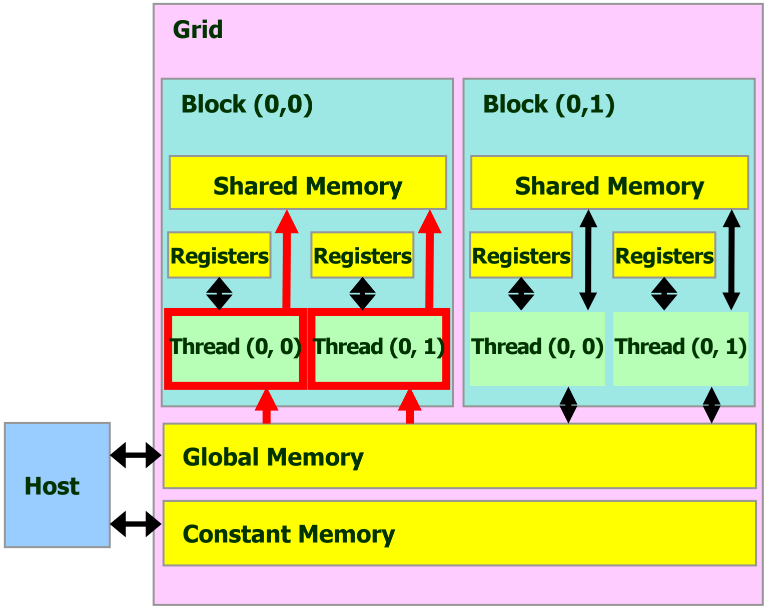
Barrier Synchronization



Barrier Synchronization

- Use **__syncthreads** to ensure data is ready for access

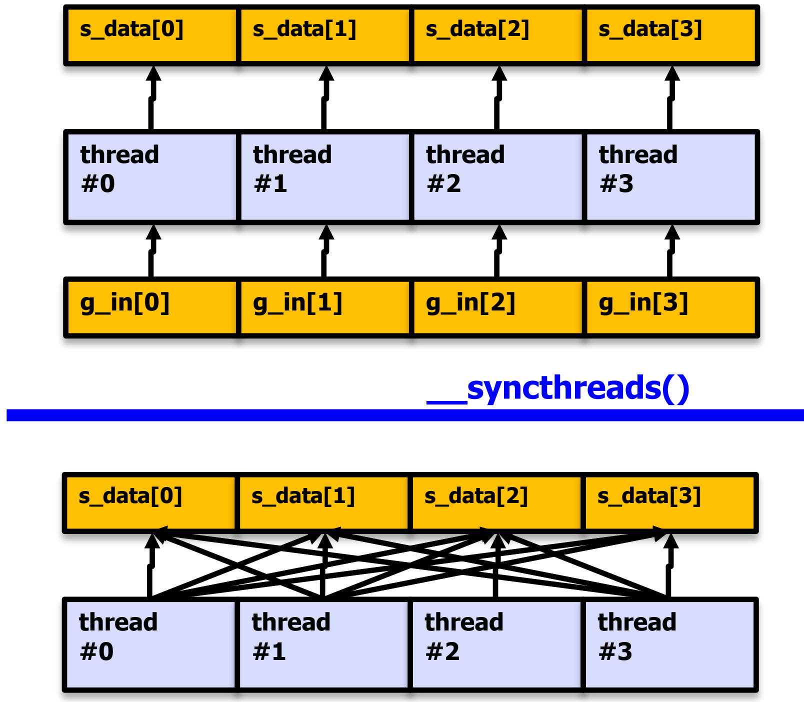
```
__global__ void  kernelFunc( int* g_input ) {
    __shared__  int  s_data[BLOCK_SIZE];
    s_data[threadIdx.x] = g_input[threadIdx.x];
    __syncthreads();
    // all data available for all threads in the block
}
```



Barrier Synchronization

- Use __syncthreads to ensure data is ready for access

```
__global__ void  kernelFunc( int* g_input ) {
    __shared__  int  s_data[BLOCK_SIZE];
    s_data[threadIdx.x] = g_input[threadIdx.x];
    __syncthreads();
    // all data available for all threads in the block

    … actions …
    // every thread can use the shared data
}
```



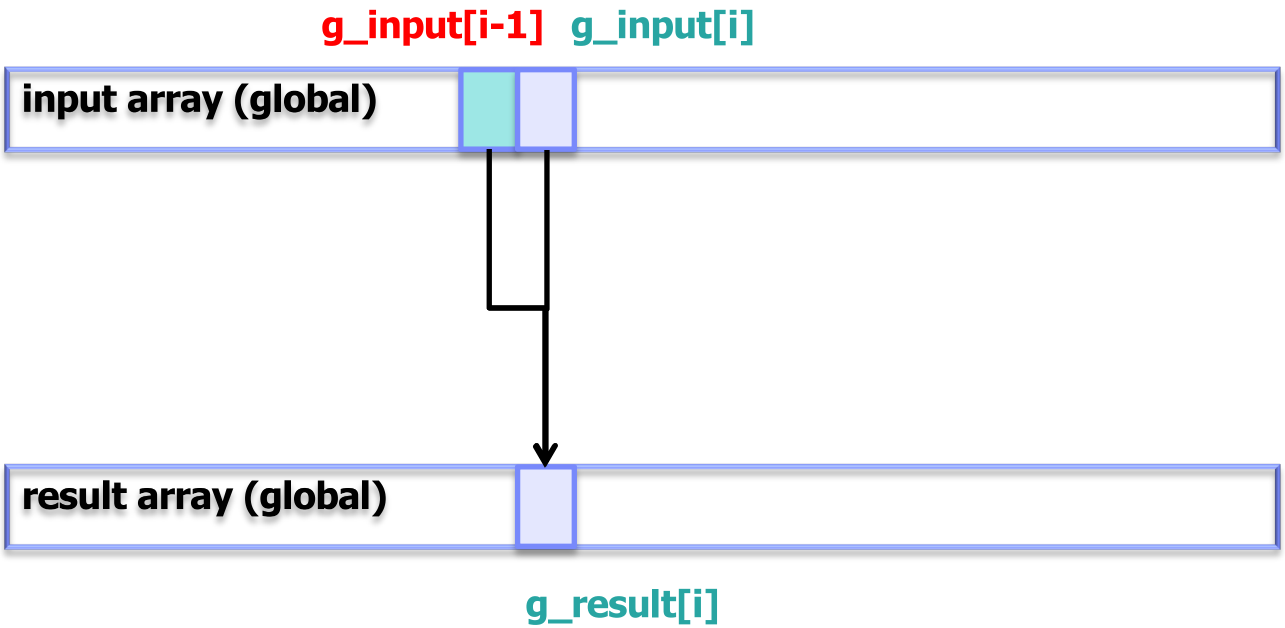
Barrier Synchronization

- Use barriers such as `__syncthreads` to wait until `__shared__` data is ready

- **Don’t synchronize or serialize unnecessarily**
  - degrade the program speed ...
  - heavy operation !

Adjacent Difference Example

- g_result[i] = g_input[i] – g_input[i – 1];




```
void getDiff(float* dst, const float* src, unsigned int size) {
   for (int i = 1; i < size; ++i) {
               dst[i] = src[i] - src[i-1];
   }
}
```

Example: CPU version

- Complete Version (adj_diff_cpu.cu)

```
#include <stdio.h>
#include <stdlib.h>   // for rand(), malloc(), free()
#include <fcntl.h>      // for open(), write()
#include <sys/stat.h>
#include “common.h”
#include <sys/time.h>

#define GRIDSIZE              (64 * 1024)
#define BLOCKSIZE 1024
#define TOTALSIZE            (GRIDSIZE * BLOCKSIZE)  // 32M byte needed!

void genData(float* ptr, unsigned int size) {
                while (size--) {
                        *ptr++ = (float)(rand() % 1000) / 1000.0F;
                }
}

// compute result[i] = input[i] – input[i-1]
void getDiff(float* dst, const float* src, unsigned int size) {
                for (int i = 1; i < size; ++i) {
                                dst[i] = src[i] - src[i-1];
                }
}

….
```

Example: CPU version

- Complete Version (adj_diff_cpu.cu)

```
.....
 int main(void) {
    float* pSource = NULL;
    float* pResult = NULL;
    int i;
   struct timeval start_time, end_time;

    // malloc memories on the host-side
    pSource = (float*)malloc(TOTALSIZE * sizeof(float));
    pResult = (float*)malloc(TOTALSIZE * sizeof(float));
    // generate source data
    genData(pSource, TOTALSIZE);
    // get current time
    gettimeofday(&start_time, NULL);
    getDiff(pResult, pSource, TOTALSIZE);
    // get end time
    gettimeofday(&end_time, NULL);
    double operating_time = (double)(end_time.tv_sec)+(double)(end_time.tv_usec)/1000000.0 -
((double)(start_time.tv_sec)+(double)(start_time.tv_usec)/1000000.0);
     printf("Elapsed: %f seconds\n", (double)operating_time);

    // print sample cases
    i = 1;
    printf("i=%2d: %f = %f - %f\n", i, pResult[i], pSource[i], pSource[i - 1]);
    i = TOTALSIZE - 1;
    printf("i=%2d: %f = %f - %f\n", i, pResult[i], pSource[i], pSource[i - 1]);
    i = TOTALSIZE / 2;
    printf("i=%2d: %f = %f - %f\n", i, pResult[i], pSource[i], pSource[i - 1]);
    // free the memory
    free(pSource);
    free(pResult);
 }
```

Example: GPU version using global memory

```
// compute result[i] = input[i] – input[i-1]
__global__  void  adj_diff_naive( float*  g_result, float*  g_input )  {
   // compute this thread's global index
   unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;


   if (i > 0)  {
      // each thread loads two elements from global memory
      int x_i    =              g_input[i];
      int x_i_minus_1 = g_input[i-1];


      g_result[i] = x_i – x_i_minus_1;
   }
}
```

Example: GPU version using global memory

```
// compute result[i] = input[i] – input[i-1]
__global__  void  adj_diff_naive( float*  g_result, float*  g_input )  {
  // compute this thread's global index
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

  if (i > 0)  {
    // each thread loads two elements from global memory
    int x_i     =          g_input[i];
    int x_i_minus_1 = g_input[i-1];

    g_result[i] = x_i – x_i_minus_1;
  }
}
```

Two Global Memory Reads

One Global Memory Write

Global memory is slow!

g_input[i] is used twice !

Example: GPU version using global memory

- Complete Version (adj_diff_naive.cu)

```
#include <stdio.h>
#include <stdlib.h>   // for rand(), malloc(), free()
#include <fcntl.h>      // for open(), write()
#include <sys/stat.h>
#include “common.h”
#include <sys/time.h>

#define GRIDSIZE              (8 * 1024)
#define BLOCKSIZE 1024
#define TOTALSIZE             (GRIDSIZE * BLOCKSIZE)  // 32M byte needed!

void genData(float* ptr, unsigned int size) {
                while (size--) {
                        *ptr++ = (float)(rand() % 1000) / 1000.0F;
                }
}

// compute result[i] = input[i] – input[i-1]
__global__  void   adj_diff_naive( float*  g_result, float*  g_input )  {
    ……
}
….:
```

Example: GPU version using global memory

- Complete Version: Host code

```
int main(void) {
    float* pSource = NULL;
    float* pResult = NULL;
    int i;
   struct timeval start_time, end_time;

    // malloc memories on the host-side
    pSource = (float*)malloc(TOTALSIZE * sizeof(float));
    pResult = (float*)malloc(TOTALSIZE * sizeof(float));
    // generate source data
    genData(pSource, TOTALSIZE);
    // CUDA: allocate device memory
    float* pSourceDev = NULL;
    float* pResultDev = NULL;
    CUDA_CHECK( cudaMalloc((void**)&pSourceDev, TOTALSIZE * sizeof(float)) );
    CUDA_CHECK( cudaMalloc((void**)&pResultDev, TOTALSIZE * sizeof(float)) );
    // CUDA: copy from host to device
    CUDA_CHECK( cudaMemcpy(pSourceDev, pSource, TOTALSIZE * sizeof(float), cudaMemcpyHostToDevice) );
    // get current time
    cudaThreadSynchronize();
    gettimeofday(&start_time, NULL);
    // CUDA: launch the kernel: result[i] = input[i] - input[i-1]
    dim3 dimGrid(GRIDSIZE, 1, 1);
    dim3 dimBlock(BLOCKSIZE, 1, 1);
    adj_diff_naive<<<dimGrid, dimBlock>>>(pResultDev, pSourceDev);
    // get end time
    cudaThreadSynchronize();
    gettimeofday(&end_time, NULL);
    double operating_time = (double)(end_time.tv_sec)+(double)(end_time.tv_usec)/1000000.0 -
((double)(start_time.tv_sec)+(double)(start_time.tv_usec)/1000000.0);
     printf("Elapsed: %f seconds\n", (double)operating_time);
```

Example: GPU version using global memory

- Complete Version: Host code

```
........
    // CUDA: copy from device to host
    CUDA_CHECK( cudaMemcpy(pResult, pResultDev, TOTALSIZE * sizeof(float), cudaMemcpyDeviceToHost) );

    // print sample cases
    i = 1;
    printf("i=%2d: %f = %f - %f\n", i, pResult[i], pSource[i], pSource[i - 1]);
    i = TOTALSIZE - 1;
    printf("i=%2d: %f = %f - %f\n", i, pResult[i], pSource[i], pSource[i - 1]);
    i = TOTALSIZE / 2;
    printf("i=%2d: %f = %f - %f\n", i, pResult[i], pSource[i], pSource[i - 1]);
    // CUDA: free the memory
    CUDA_CHECK( cudaFree(pSourceDev) );
    CUDA_CHECK( cudaFree(pResultDev) );
    // free the memory
    free(pSource);
    free(pResult);
}
```

Shared Memory Use Strategy

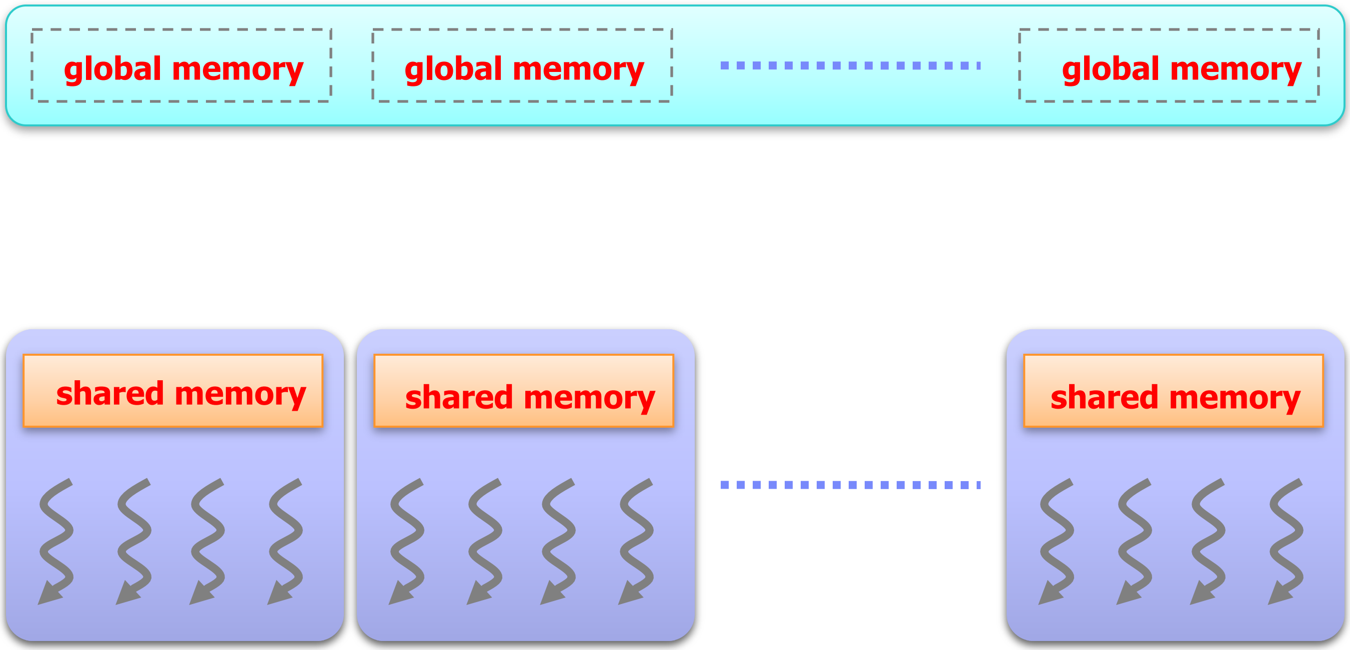
- Global memory resides in device memory (DRAM)
  - Much slower access than shared memory

- Tile data to take advantage of fast shared memory:

Shared Memory Use Strategy

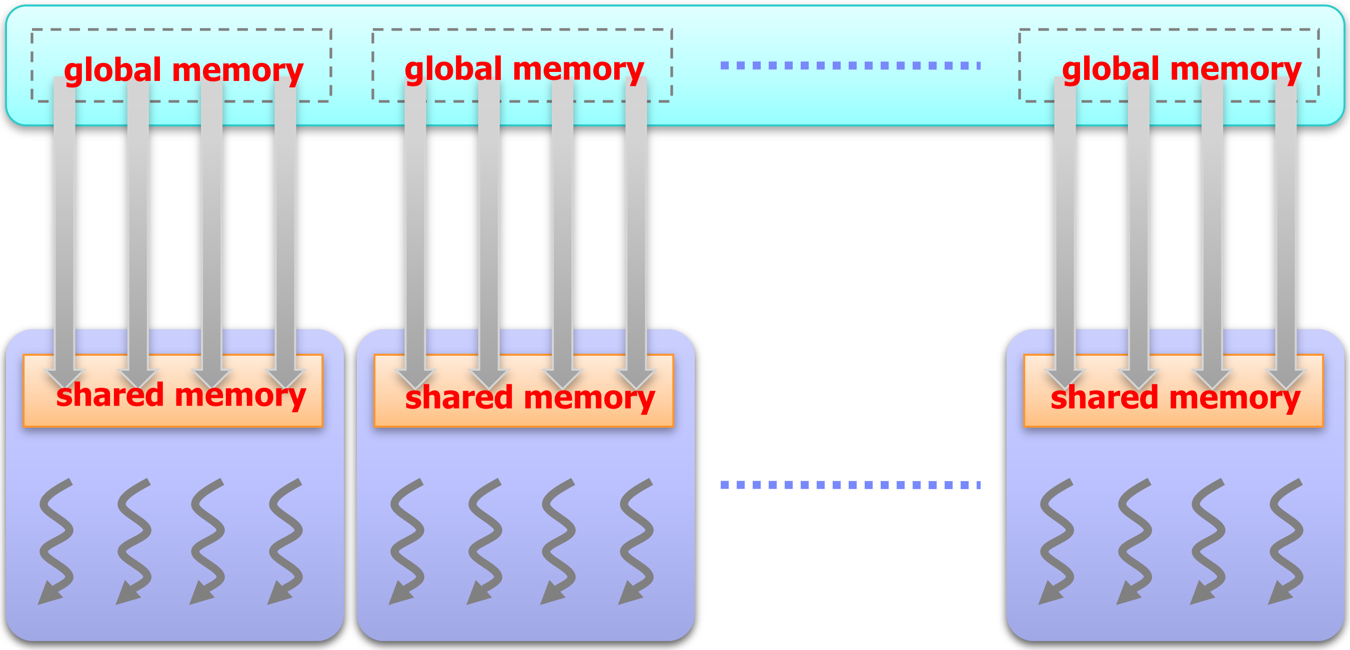
- Partition data into subsets that fit into shared memory

Shared Memory Use Strategy



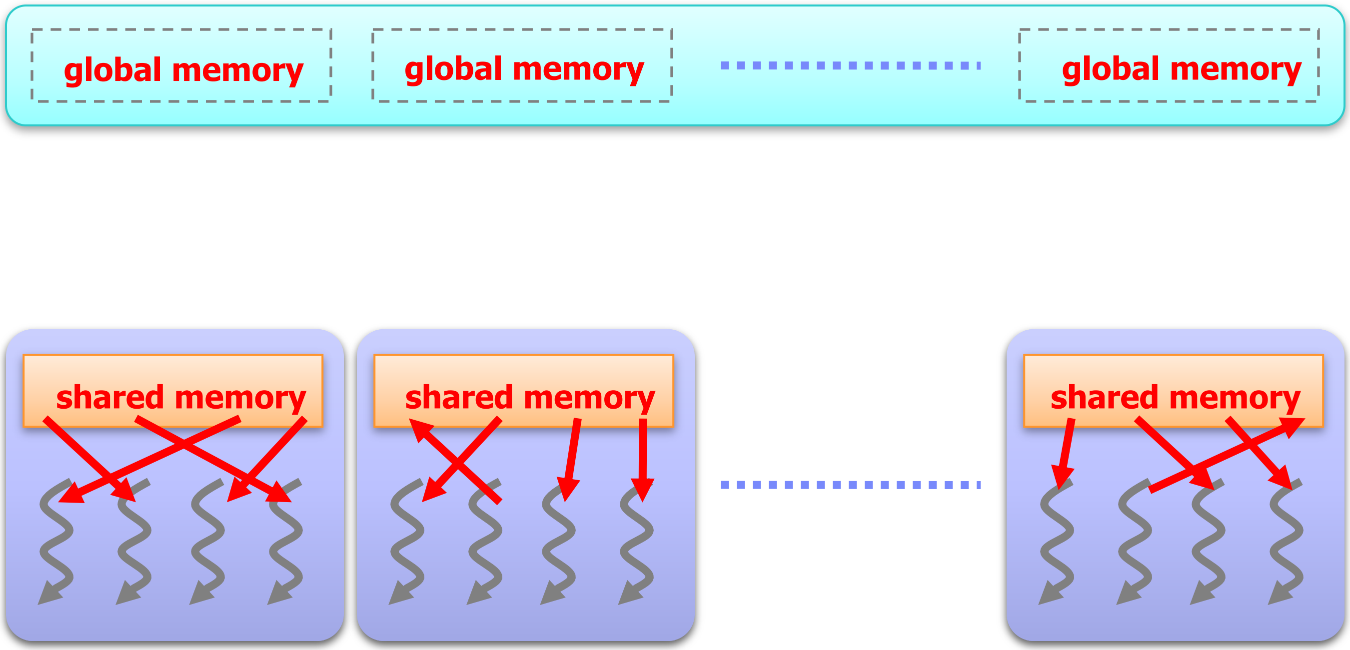
- Handle each data subset with one thread block

Shared Memory Use Strategy



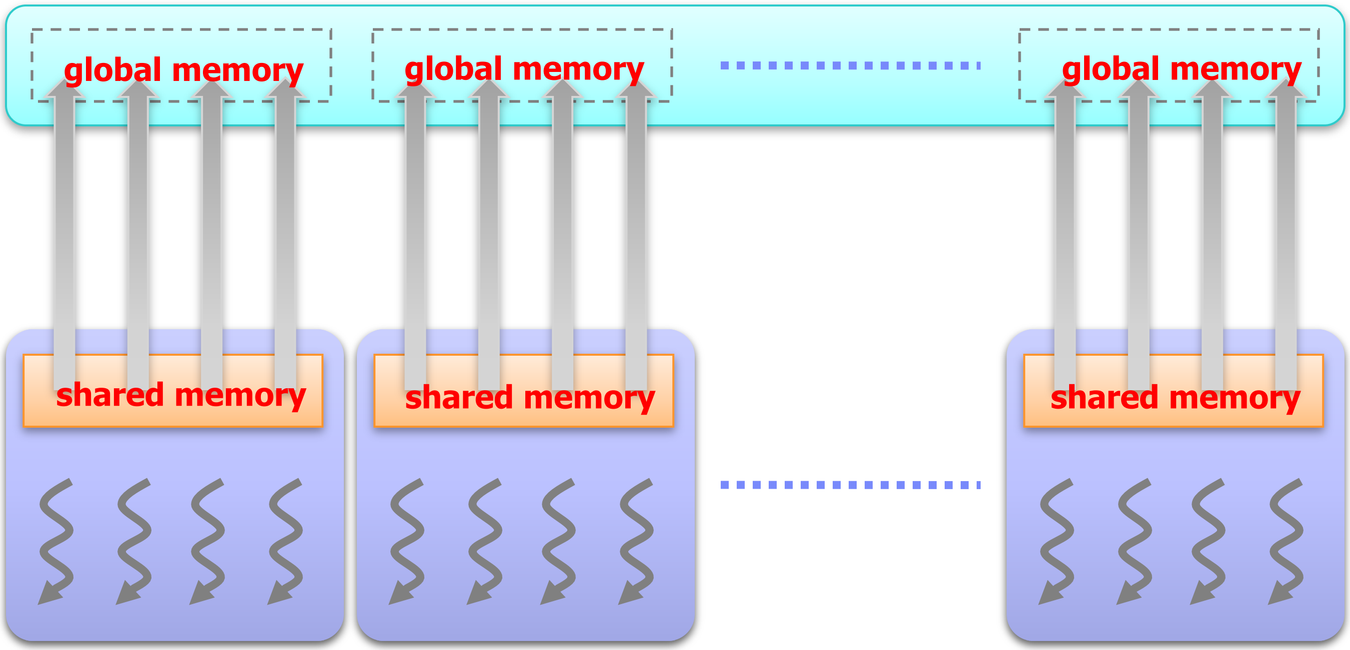
- Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism

Shared Memory Use Strategy



- Perform the computation on the subset from shared memory

Shared Memory Use Strategy



- Copy the result from shared memory back to global memory

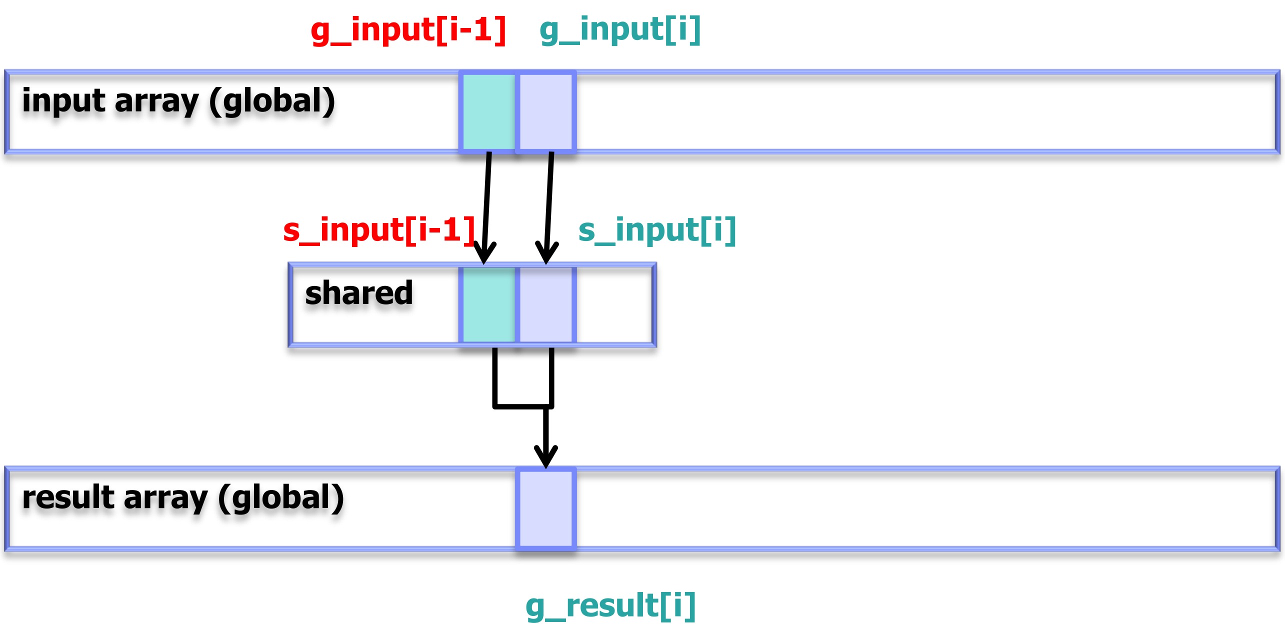
Shared Memory Use Strategy

- Carefully partition data according to access patterns

- Read-only ➔ __constant__ memory (fast)
- R/W & shared within block ➔ __shared__ memory (fast)
- R/W within each thread ➔ registers (fast)
- R/W inputs/results ➔ __global__ memory (slow)

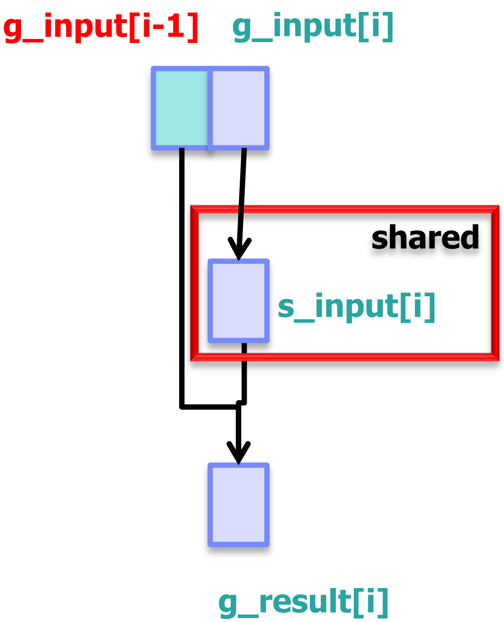
Example: GPU version using shared memory

- g_result[i] = s_input[i] – s_input[i – 1];



Example: GPU version using shared memory

```
__global__ void adj_diff(float* g_result, float* g_input) {
    int tx = threadIdx.x;
   // allocate a __shared__ array, one element per thread
   __shared__ float s_data[BLOCKSIZE];
   // each thread reads one element to s_data
   unsigned int i = blockDim.x * blockIdx.x + tx;
   s_data[tx] = g_input[i];
   // avoid race condition: ensure all loads complete before continuing
   __syncthreads();
   // now action
   if (tx > 0) {
      g_result[i] = s_data[tx] – s_data[tx–1];
   } else if (i > 0)   {
      // handle thread block boundary (for tx == 0 case)
      g_result[i] = s_data[tx] – g_input[i-1];
    }
}
```



Optimization Analysis

Implementation	Original	Improved
Global Loads	2N	N + N/BLOCK_SIZE
Global Writes	N	N
Throughput	36.8 GB/s	57.5 GB/s
source line of codes (SLOC)	18	35
Relative Improvement	1x	1.57x
Improvement/SLOC	1x	0.81x

- Optimizations tend to come with a development cost

Compare the Execution Time

- Host version: 0.260803 seconds
- CUDA global memory: 0.002964 seconds
- CUDA shared memory: 0.000005 seconds

- Use shared memory if possble!!!

Dynamically allocated shared memory

- when the size of the array isn't known at compile time...

```
__global__ void adj_diff(int *result, int *input) {
  // use extern to indicate a __shared__ array will be
  // allocated dynamically at kernel launch time
 extern __shared__ int s_data[];
  ...
}

// pass the size of the per-block array, in bytes, as the third
// argument to the triple chevrons
adj_diff<<<num_blocks, block_size, block_size * sizeof(int)>>>(r,i);
```

Next?

- **Matrix Multiplication**
  - **Basic Version**
  - **Tiled Version**

- Review: Memory Hierarchy
- Importance of Memory Access Efficiency

- **GPU Memory Hierarchy**
- **Improving Tiled Matrix Multiplication using Shared Memory**
- **Impact of Memory on Parallelism**