COSC 407
Intro to Parallel Computing

Topic 15: Making Things Faster

Source: NVIDIA Best Practices Guide

Outline

Previous pre-recorded lecture:

- CUDA Scalability
- Thread Scheduling on the H/W: Thread Lifecycle
- zero-overhead and latency tolerance
- CUDA Memories Types (and Performance)

Today:

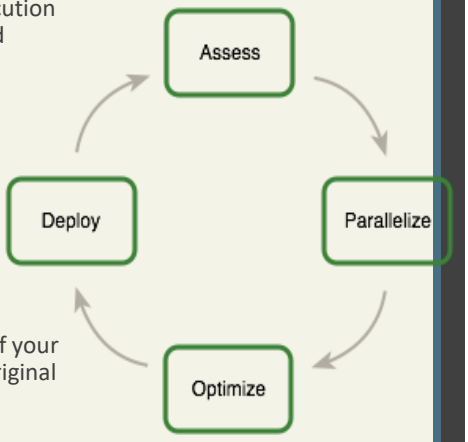
- Thread synchronization and barriers
- Atomic operations
- Memory Optimization
- Instruction Optimization
- Control Flow
- Example on Reduction

Next Lecture:

- Example: Improving Performance of Matrix Multiplication
- More optimizations
- Bandwidth

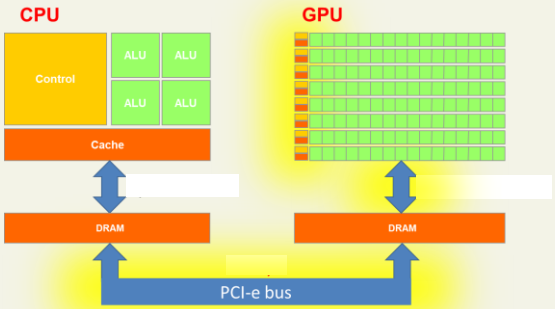
APOD Framework

- Assess, Parallelize, Optimize, Deploy
  - Locate parts responsible for most execution time. Decide if they can be parallelized
  - Determine expected speedup
    - Amdahl's and Gustafson's laws
  - Parallelize code
  - Optimize your implementation
    - **Memory optimization**
    - **Instruction optimization**
    - **Control flow**
  - Deploy any parallelized components of your program, ***measure***, and compare to original expectations.



Memory Optimization

- **The most important area for performance**
- Goal: maximize the use of hardware by maximizing bandwidth.
- **Two aspects:**
  (1) Transferring data between **host and device**
  (2) Using the **different memories of the device**



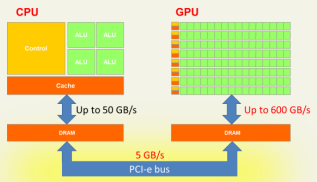
Transferring Data Between Host and Device

Guideline 1: *"Minimize data transfer between host and device:*

- *"even if it means running some kernels on the device that do not show performance gains when compared with running them on the host CPU."*
- Create and destroy ***intermediate data structures***, which are solely used by the device, ***on the device only***.
- Batch **small transfers** into one **large transfer** (to avoid the overhead associated with each transfer)

Legend

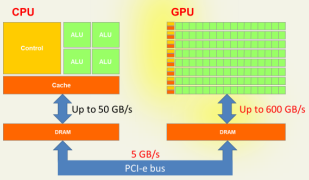
- **high priority guideline**
- **Medium or low priority guideline**



Using the Different Memories of the Device

There are several guidelines when you use the different memories of the device.

a) **Use *fast memory*** and **avoid *slow memory*** as much as possible.
b) Copy ***frequently accessed data to faster memory*** to reduce global memory traffic.
c) Access memory fast by using ***coalesced global memory access.***
   a) Also, ***reduce misaligned*** memory access.



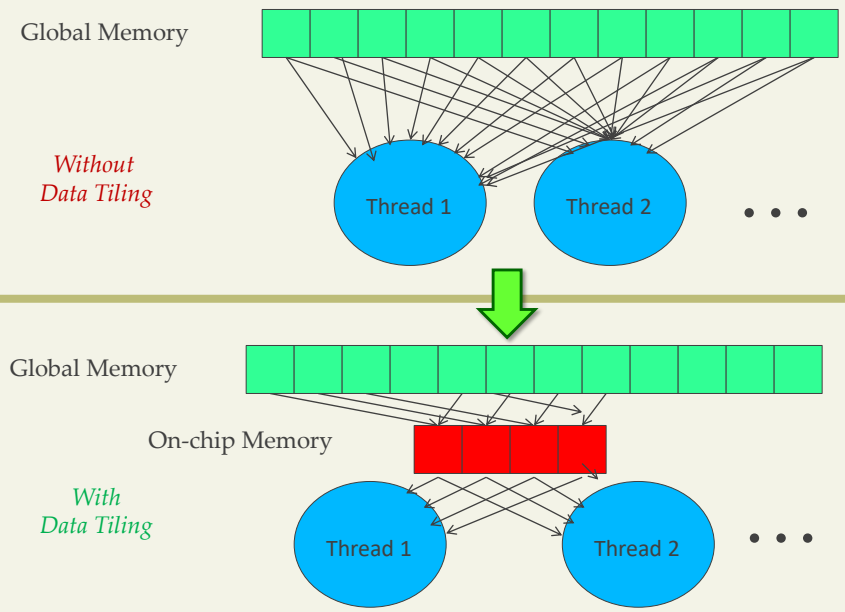
Reducing Global Memory Traffic

A profitable way of performing computation on the device is to *tile data* to take advantage of **fast shared memory**:

- **Basic idea**: partition data into subsets called ***tiles***, such that each *tile* **fits** into the ***shared memory***.
- Then, handle *each* ***data tile*** *with* ***one thread block***
  1. Load the subset from global memory to shared memory, **using multiple threads to exploit memory-level parallelism**
  2. Perform the computation on the subset from shared memory, then move to next subset, etc.
  3. Copy results from shared memory to global memory

- ***Restriction***: kernel **computations on these subsets of data (tiles)** can be done **independently from each other**
- Remember, we have different memories:
  - **Global memory: large but slow.**
  - **Shared memory: small but fast**

Shared Memory Blocking Basic Idea



Use Faster memory

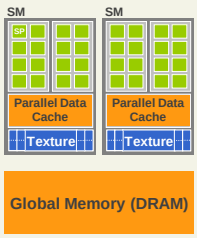
Guideline 2: "*use fast memory and avoid slow memory as mush as possible*"

Var. Declaration	Memory			Scope	Lifetime	speed
`int x;`	Register	on-chip		thread	Thread	very fast
`int array[10]`	Local	off-chip	uses L1 & L2 Cache	thread	Thread	slow
`_shared_ int x;`	Shared	on-chip	on configurable Mem	block	Block	fast
`_device_ int x;`	Global	off-chip	on DRAM,small cache	grid+host	application	slow
`_constant int x;`	Constant	off-chip	has dedicated Cache	grid+host	application	fast

- **e.g.** keep your scaler variables in registers unless you need other threads to access them.
- **e.g.** use constant memory for data that will never change on the host (limit is 64K).

Communication & Synchronization

- Having different types of "shared" memories provide an excellent way for threads to communicate. For example:
  - Shared memory:
    - Shared by all threads in the same block
    - Very fast
  - Global memory:
    - Shared by all threads in all blocks and across different kernels
    - Slow



- **Whenever there is communication** between threads, we need a mechanism for synchronizing threads.
  - e.g., if thread x reads a value that is supposed to be **written by** other threads, then thread x must ***wait*** until the value is written.
  - If we don't synchronize, we will end up with a **race condition**.

Synchronization

- Synchronization is required to avoid **data race**
  - Avoids RA**W**, **W**AR, **WAW** hazards when accessing shared or global memory
    - RAW : Read After Write
    - WAR : Write After Read
    - WAW : Write After Write
- **Synchronization** can be done in CUDA by means **barriers**.
  - Another technique is memory fences for ensuring a visibility of memory accesses to other threads, but we will *not discuss this in our course*.
- We will also discuss mutual exclusion
  - **Atomic operations**
  - **Critical sections**
    - although they are not recommended.

Where are the barriers?

Similar to OpenMP, we can use **barriers** to sync thread operations. We have two types of them:

1) Explicit barriers **within a block**

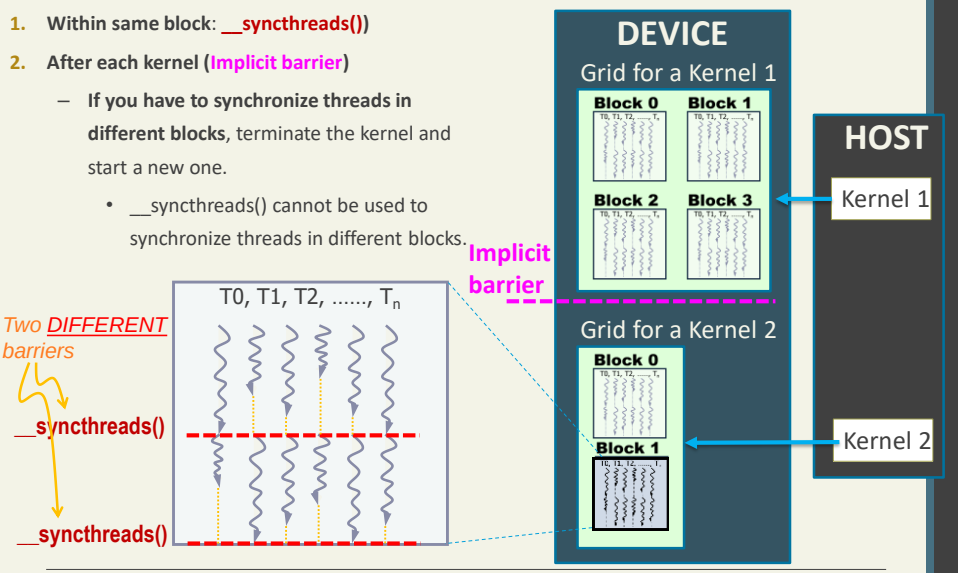
- A barrier can be placed using `__syncthreads();` which synchronizes all threads *within a block*
  - but NOT in different blocks.
  - **Example**: see the matrix multiplication kernel we saw before
- **All threads within a block must reach and execute __syncthreads()** before execution can resume
  - Note: having several __syncthreads() means having ***different barriers***. All threads must reach and execute the **same barrier** before execution can resume.

2) Implicit **barriers at the end of each kernel.**

- A kernel must complete before the next kernel can start

Where are the Barriers?

1. **Within same block: __syncthreads())**
2. **After each kernel (Implicit barrier)**
   - **If you have to synchronize threads in different blocks**, terminate the kernel and start a new one.
     - __syncthreads() cannot be used to synchronize threads in different blocks.



Example

Q1: Identify data race hazards

Q2: Modify to avoid data race

```
__global__ void foo(){
    __shared__ int array[100]; //create a shared array
    int i = threadIdx.x;
    array[i] = ...;                    //each thread initializes one element
    if(i > 1 && i < 100)
        array[i] = array[i-1];         //each thread shifts one element right
}
```

Write

Write

race cond

- Write = ALARM, there might be data race!
  - no problem if EVERY thread is reading from ITS OWN element (i.e., that is written by THIS thread, not by other threads).
  - Problem if there are threads reading from elements written by other threads.

Example: Answer for Q2

```
__global__ void foo(){
   int i = threadIdx.x;
   __shared__ int array[100];
   array[i] = ...;
   __syncthreads();      //so that data is written first before anyone
                         // reads it
   if(i > 1 && i < 100){
        array[i] = array[i-1];        //shift right by 1
    }
}
```

Write

Not enough!!
Q: Can you identify the data race?

Example: Answer for Q2

```
__global__ void foo(){
   int i = threadIdx.x;
   __shared__ int array[100]; __shared__ int temp[100];
   array[i] = ...;
   __syncthreads();      //so that data is written first before anyone reads it
   if(i < 100){
        temp[i] = array[i];  //to ensure reading original data
   }
   __syncthreads();
   if(i > 1 && i < 100){
        array[i] = temp[i-1];
   }

}
```

Q: do we really need to create a shared temp array? Can't we just use a register?

Example: Answer for Q2

```
__global__ void foo(){
   int i = threadIdx.x;
   __shared__ int array[100]; int temp;
   array[i] = ...;
   __syncthreads();      //so that data is written first before anyone reads it
   if(i > 1 && i < 100){
        temp = array[i-1];  //to ensure reading original data
   }
   __syncthreads();
   if(i > 1 && i < 100){
        array[i] = temp;
   }
   __syncthreads();    //if other reads/writes are done after that
   //...
}
```

Atomic Operations

- Data race could occur when multiple threads are accessing **same shared memory lactation.**
  - e.g., 1000 threads trying to increment the same variable stored in the shared (or global) memory.

```
__shared__ int x = 0;
x = x + 1;
```

- To avoid data race, **atomic operations** could be used.
- **Atomic functions:**
  - **serializes thread accesses** to shared data.
  - **read-modify-write** atomic operation on one word in global or shared memory.
  - Two types of functions:
    - **Arithmetic**: `atomicAdd()` , `atomicSub()` , `atomicExch()` , `atomicMax()` , `atomicIncr()`, `atomicDec()`, `atomicCAS`
    - **Bitwise**: `atomicAnd()`, `atomicOr()`, `atomicXor()`

Example: Atomic

```
__global__ void increment_naive(int* x) {
    *x = *x + 1;          // each thread runs this code
}
__global__ void increment_atomic(int* x) {
    atomicAdd(x, 1);  // each thread runs this code
}
int main() {
    int* h_x;             // declare and allocate host memory
    int* d_x;             // declare, allocate, and zero memory for x
    cudaMalloc(&d_x, sizeof(int));
    cudaMemset(d_x, 0, sizeof(int));
    // launch one of the two kernels:
    // increment_naive<<<NUM_THREADS/BLOCK_WIDTH, BLOCK_WIDTH>>>(d_x);
    increment_atomic<<<NUM_THREADS/BLOCK_WIDTH, BLOCK_WIDTH>>>(d_x);
    cudaMemcpy(&h_x, d_x, sizeof(d_x), cudaMemcpyDeviceToHost);
    printf("x = %d\n", h_x);
    free(h_x); cudaFree(d_x); return 0;
}
```

Atomics Limitations

- Slower!
  - Because of the serialized execution of threads
  - Don't over-use.
- No specific order
  - for the serial execution of threads
- Only certain operations are supported
  - Full list in a previous slide
  - `atomicCAS` (Compare-and-Swap) can be used to provide implementation of mostly any atomic operation
    - ***but we will not discuss this***.
- Only `int` is supported for most operations
  - Only `atomicAdd` and `atomicExch` support both int and float.

Use Faster Memory

Guideline 3: *Copy **frequently accessed data** to faster memory in order to reduce global memory traffic.*

- Even if this means more work for the threads.

- **Example:**
  - "Matrix Multiplication" from previous lecture – we used shared memory to avoid redundant transfers from global memory.

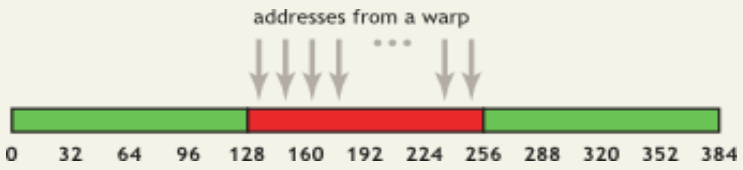
- **Basic technique:**
  - Copy data from global memory to shared memory ***IF*** data is going to be use ***frequently***.

```
__global__ void foo(float* arr){   // arr is pointing to 128 element 1D array
    __shared__ float shrArr[128];          // shrArr lives in shared
memory
    int idx = theadIdx.x;                  // idx lives in a register
    sh_arr[idx] = arr[idx];                // copy from global to shared
    __syncthreads();                       // ensure every thread finished
                          // copying its array element before proceeding
... //process the elements in sh_arr
```


Coalesced Global Memory Access

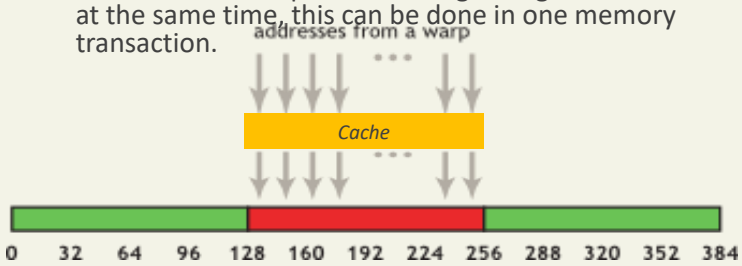
Guideline 4: Use **coalesced *global* memory access**

- One of the MOST important performance considerations.
- Accessing global memory will be fastest when **adjacent threads access contiguous memory** locations at the same time.
  - Global mem. accesses by *threads of a **warp*** can be performed in as few as ***one transaction*** if this guideline is followed



Coalesced Global Memory Access

- ***Explanation:***
  - Each ***memory transaction*** (R/W) gives access to a chunk of memory (***32-, 64-, or 128-byte***) at once, even *if you are reading/writing to a single memory location*.
  - If threads in a warp are accessing contiguous locations at the same time, this can be done in one memory transaction.

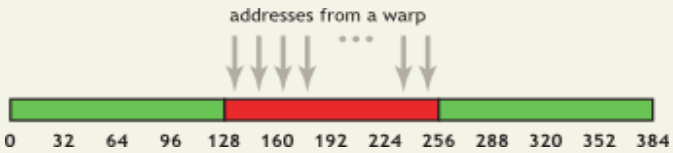


- *"the concurrent accesses of the threads of a warp will coalesce into a number of transactions equal to the number of **cache lines necessary to service all of the threads** of the warp."*

Coalesced Global Memory Access

Memory Access Patterns: (i) Simple (sequential, aligned)

- ***i-th*** *thread accesses* ***i-th*** *word in a cache line.*
- *Not all threads need to participate*
- *In the figure,* ***a single*** *128-byte L1* ***transaction*** *is needed.*
  - *Assume a warp has 32 threads, each reads a 4-byte float = 128 bytes*



Coalesced Global Memory Access

Memory Access Patterns: (ii) Sequential but Misaligned

- "If sequential threads in a warp access memory that is sequential but ***not aligned with the cache lines***, **two** 128-byte L1 transactions are needed"

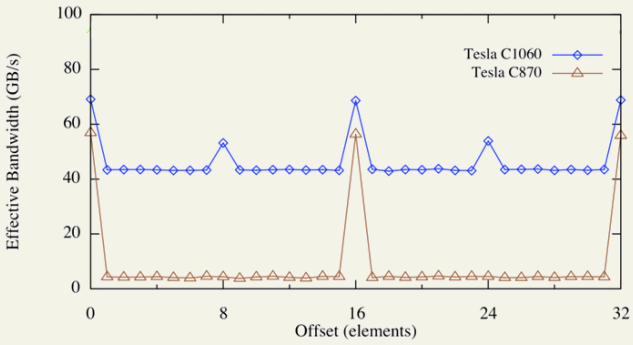
- Memory allocated via **cudaMalloc()** is aligned to at least 256 bytes. Therefore, **choosing sensible thread block sizes, such as multiples of the warp size**, facilitates memory accesses by warps that are aligned to cache lines

Guideline 5: choose sensible **thread block sizes**, such as **multiples** of the **warp size**, to avoid misaligned memory access.

Coalesced Global Memory Access

Memory Access Patterns: (ii) Sequential but Misaligned

The effect of misaligned accesses



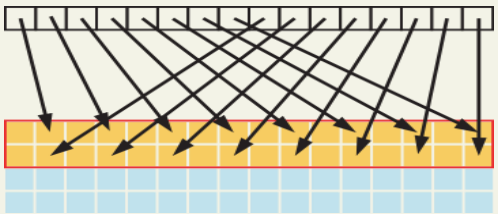

```
__global__ void foo(float* a, int offset){
      int i = blockDim.x * blockIdx.x + threadIdx.x + offset;
      a[i] = a[i] + 1.0;
}
```

Source: http://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/

Coalesced Global Memory Access

Memory Access Patterns: (iii) Strided

- A "stride" is number of locations between ***the beginnings* of successive array elements** being accessed. The figure shows adjacent threads accessing memory with a stride of 2

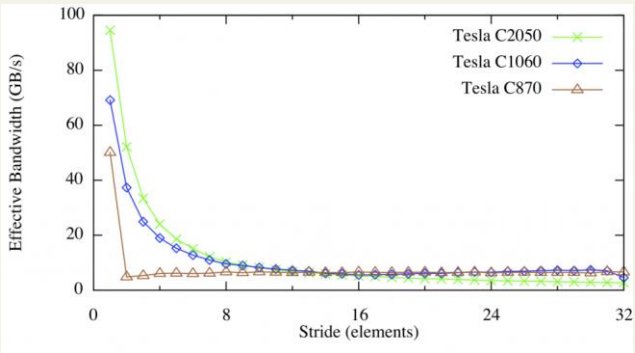


- How much does stride affects the bandwidth?
  - A stride of **2 results in a 50% of load/store efficiency**
    - since half the elements in the memory transaction are not used and represent wasted bandwidth.
  - As the stride increases, the effective bandwidth decreases until the point where 32 lines of cache are needed for the 32 threads in a warp

Coalesced Global Memory Access

Memory Access Patterns: (iii) Strided, *cont'd*

The Effect of Strided memory access




```
__global__ void foo(float* a, int stride){
      int i = blockDim.x * blockIdx.x + threadIdx.x * stride;
      a[i] = a[i] + 1.0;
}
```

Source: http://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/

Summary: Coalesced Global Memory Access

- Coalesced global memory access:
  - Simple access pattern gives the best performance
    - **Example:** threads 0 to 3 accessing a[0], a[1], a[2], a[3] at once
  - Misaligned memory access should be avoided
    - e.g., by choosing sensible thread block sizes e.g. multiples of warp size.
  - **Non-unit-stride** global memory accesses should be avoided (or at least minimize the stride) whenever possible
    - **Strided Example**: threads 0 to 3 accessing a[0], a[2], a[4], a[6] at once
    - How: e.g., use shared memory: load/store data in a coalesced pattern from global memory and then reorder it in shared memory.
      - there is no penalty for non-sequential or unaligned accesses by a warp in shared memory
  - Random **Global Memory Access** is the worst!
    - **Example**: threads 0 to 3 accessing a[0], a[30], a[10],a[71] at once. This is bad, and probably will be done in 4 separate memory transactions.

Instruction Optimization

Guideline 6: Optimize your code at the instruction level
(after you have completed higher level optimization e.g., memory access)

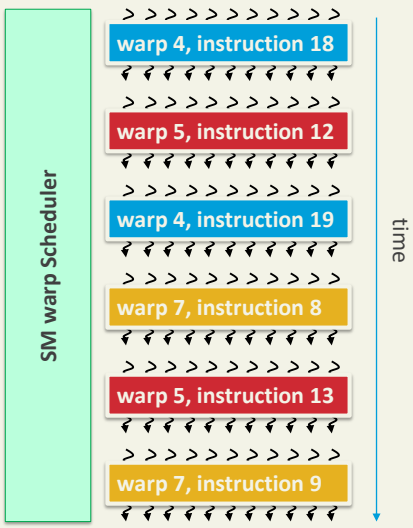
- **Basic idea**: write instructions in a more efficient way.
- **Examples**:
  - Use **shift operations** to avoid expensive division & modulo calculations
    - For example: multiplying i by 2 is same as i<<1; If `n` is a power of 2, `i/n` is equivalent to `i≫log2(n)` and `i%n` is equivalent to `i&(n-1)`.
      - The compiler will perform these conversions if `n` is literal.
  - Use **faster**, more specialized **math functions** over slower, more general ones when possible.
    - E.g., for exponentiation using base 2, use **`expf2()`** instead of **`expf()`**
  - Use **CUDA fast math library** whenever speed trumps precision
    - E.g., use **`__sin()`** instead of **`sin()`** (*see next slide*)
  - **Avoid automatic conversion** of doubles to floats
    - Automatic conversion requires more clock cycles
    - Accuracy issues

Instruction Optimization: Math Operations

- **Two types of runtime math operations**
  - `functionName()`
    - callable from **both device and host**
    - more accurate, but slower
    - **Example:**
      - `pow, sqrt, exp, log, log2, log10, log1p`
      - `sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, ...`
      - `ceil, floor, round`
      - Etc.
  - `__functionName()`
    - **device-only**
    - less accurate, but faster
    - Some mathematical functions only. Examples:
      - `__pow, __log, __log2, __log10, __exp, __sin, __cos, __tan`

Recall: Thread Scheduling

All threads in a warp **share a program counter**



Warp Divergence

Guideline 7: **Avoid (or minimize) warp divergence**
(i.e. to have different execution paths within the same warp).

Divergence happens when:

- **Threads in a warp take different paths.**
- **At least one thread in a warp takes longer to finish**

Details: flow control instruction (if, switch, for, while) may cause threads of the same warp to diverge. Why is this bad?

- ***Case 1 (if, switch):*** threads of a **warp share a program counter.** This means different execution paths are **serialized** (i.e., paths are traversed one at a time until there is no more.**)**
  - e.g., using "if", the "then" threads are executed first, then the "else" ones
- ***Case 2 (loops):*** If some threads execute **more iterations** than others, then those which finish first may be waiting in an idle state until all threads have finished.

Warp Divergence: Examples

- Code **WITH** divergence: the threads in a warp take different branches.
  - **Example1 *:**

```
if(threadIdx.x<5) p[i] = 10; //threads 0 to 4
else              p[i] = 5;  //threads 5, 6, ...
```

  - **Example 2 *:**

```
if (x < 0.0)   z = x - 2.0;    //if x is not the same for all warp threads
else           z = x * x;
```

- Code **WITHOUT** divergence: all the threads in a warp take the same branch.
  - **Example 3: condition depends on threadIdx and WARP_SIZE**

```
if(threadIdx.x/WARP_SIZE == 2){...}  //all threads in wrap2
else {...}                           //other warps
```

  - **Example 4:**

```
if(y < 0.0)    z = x * x;     //if y is the same for all threads in warp
else           z = x + 2.3;
```

* Assuming the code is not optimized by the compiler

Adapted from NVIDIA Best Practices Guide

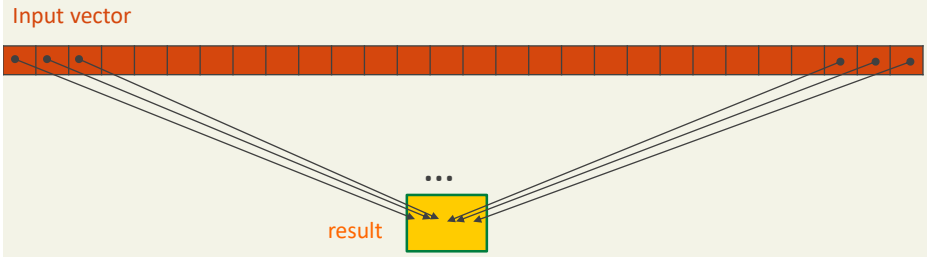
Reducing Thread Divergence and Balance Workload

- Sometimes warp divergence is unavoidable, but if possible do the following:
  - ***If the control flow depends on the thread ID***, minimize divergence by:
    - ***Using warp id*** to assign task (i.e., `threadIdx.x/WARP_SIZE`)
      - `e.g., if(threadIdx.x/WARP_SIZE == 2)` //all threads in warp 2
    - Assign the same task to all threads <n or >n (where n is warp size)
      - `e.g., if(threadIdx.x < n) …` //n is multiple of warp size
  - Try to ***equally distribute the workload*** to all threads in a warp
    - e.g., divide your tasks to expensive and inexpensive ones. Then assign each group to a different warp (or even to a different kernel)
  - Consider using the ***host (CPU) to carry out the part*** of the work that causes a load imbalance on the GPU
  - ***Modify the algorithm*** so that it does not cause warp divergence

Reduction

- Reduction is one of the common algorithms suitable for parallel programs.
  - Other common algorithms include: histogram, sort, and scan.
    - but they are outside the scope of this course.
- Reduction aims to reduce all elements to a single value
  - max, min, sum, etc.
- How:
  - Serial:
    - run a loop over every element.
  - Parallel:
    - Partition array into **segments**
    - Each segment is processed by a thread block to find a **partial result**
    - **Combine** results from different blocks

Vector Reduction

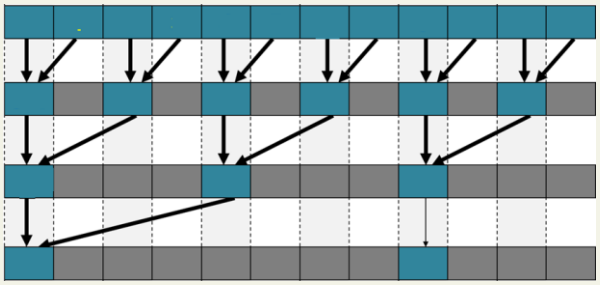


- We'll use **in-place** reduction using **shared memory**
  - The original vector is in device global memory
  - The shared memory used to hold a partial result (sum/min/etc) vector
  - Each iteration brings the partial result vector closer to the final result
  - The final solution will be in element 0

Vector Reduction - Basic Algorithm

- Here is the general algorithm (assume we are finding the sum)

  1) find the partial sum **within each block** and store
     - We need shared memory so that block threads can communicate

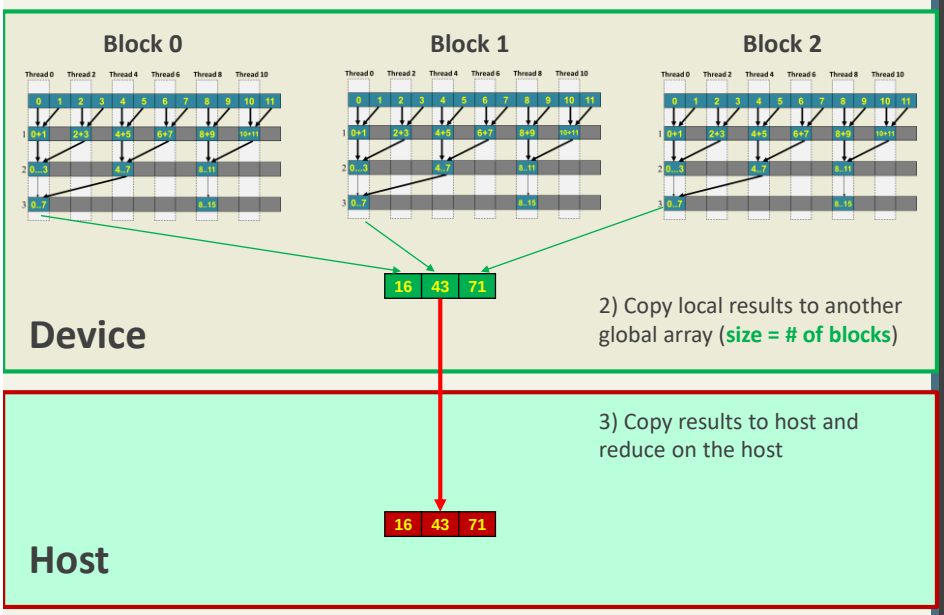


Inside **ONE** block

  2) copy the partial sum from each block into global array
     - global memory so that blocks can communicate

  3) find the sum of all partial results stored in the global array
     - could do this on the host

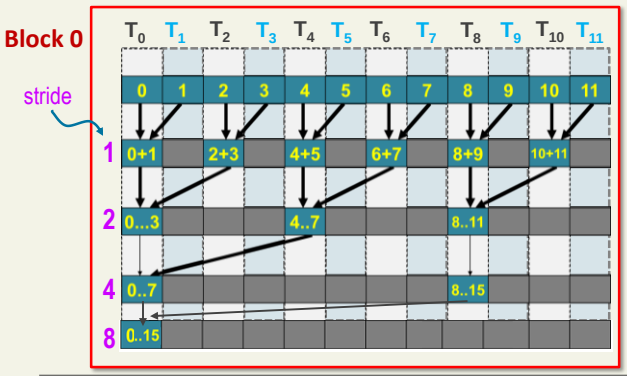
Find the Sum – Basic Algorithm



Find the sum: First Idea

```
__shared__ float partialSum[];
//assume we already loaded all array segments into block-scoped partialSum[]
int i = threadIdx.x;
for (int stride = 1; stride<blockDim.x; stride *= 2){
    if (i % (2 * stride) == 0)
        partialSum[i] += partialSum[i + stride];
    __syncthreads();
}//then save partialSum[0] to a global variable
```

For 512 elements, it takes 9 iterations



stride = 1: if(i%2==0) find sum

stride = 2: if(i%4==0) find sum

stride = 4: if(i%8==0) find sum

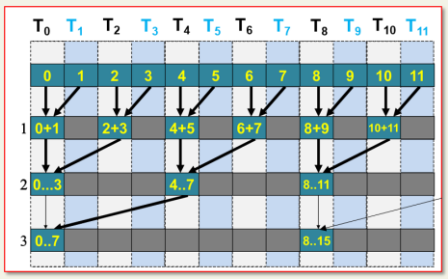
Problem: Divergence

```
__shared__ float partialSum[]
//assume we already loaded all array segments into block-scoped partialSum[]

int i = threadIdx.x;
for (int stride = 1; stride < blockDim.x;  stride *= 2){
    if (i % (2 * stride) == 0)
        partialSum[i] += partialSum[i + stride];
    __syncthreads();
}
//then copy result to global variable
```

We don't care about strided access here. Why?

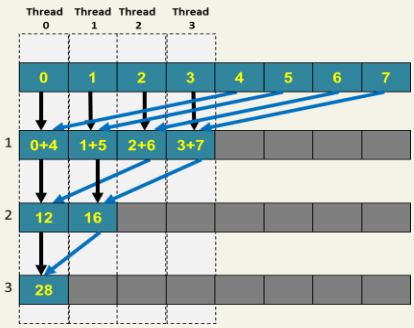
Divergence



Good Solution

```
__shared__ float partialSum[]
//assume we already loaded all array segments into block-scoped partialSum[]

int i = threadIdx.x;
for (int stride = blockDim.x/2; stride >= 1;  stride = stride/2){
    if (i < stride)              //all adjacent threads do the same thing
        partialSum[i] += partialSum[i + stride];
    __syncthreads();
}
```

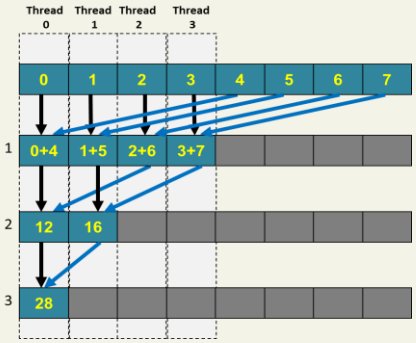


Good Solution

```
__shared__ float partialSum[]
//assume we already loaded all array segments into block-scoped partialSum[]

int i = threadIdx.x;
for (int stride = blockDim.x >> 1; stride >= 1;  stride >>= 1){
    if (i < stride)              //all adjacent threads do the same thing
        partialSum[i] += partialSum[i + stride];
    __syncthreads();
}
```

Instruction optimization



Conclusion

Today:

- Thread synchronization and barriers
- Atomic operations
- Memory Optimization
- Instruction Optimization
- Control Flow
- Example on Reduction

Next Lecture:

- Example: Improving Performance of Matrix Multiplication
- More optimizations
- Bandwidth
- iCLicker questions!