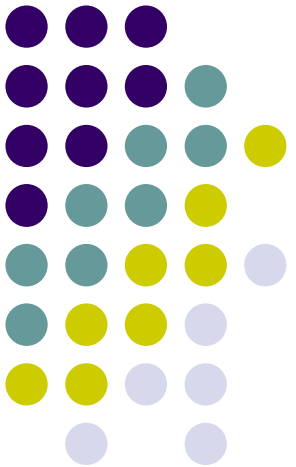
ME759
High Performance Computing for Engineering Applications

Parallel Prefix Scan Operation in CUDA

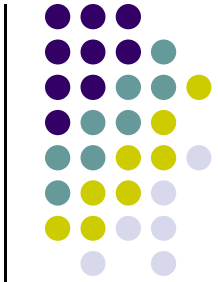
Using streams in CUDA

October 16, 2013



"Software is like entropy: It is difficult to grasp, weighs nothing, and obeys the Second Law of Thermodynamics; i.e., it always increases."
– Norman Augustine

Before We Get Started…

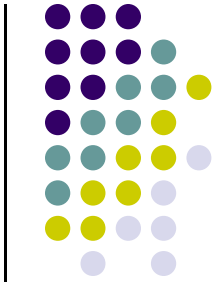


- Last time
  - Wrap up, vector reduction operation
  - CUDA specific issues that impact GPU computing performance
  - Example: Performing a scan operation

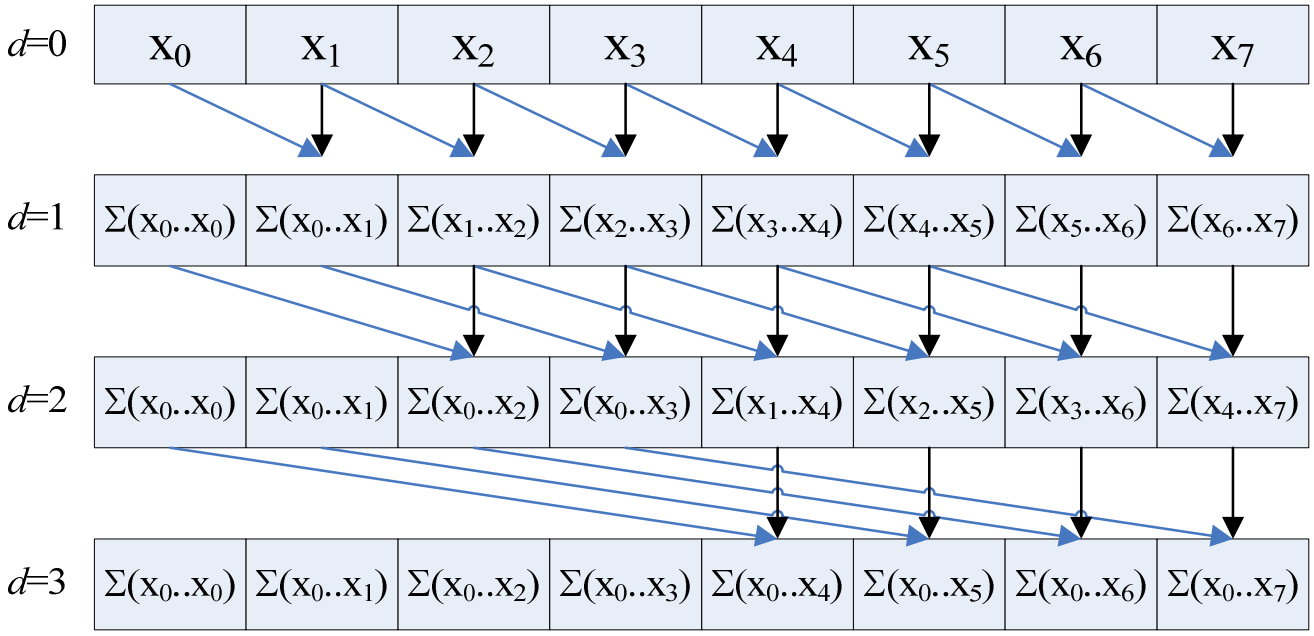
- Today:
  - Wrap up: scan operation
  - Streams in CUDA: hiding data movement with useful computation

- Miscellaneous
  - HW 6 posted online. Due on Mo, October 21 at 11:59 pm
  - Start thinking about midterm projects and final projects
    - Due date for midterm project topic selection was Oct 21
    - Moved back to Oct 23 since Oct 21 coincides w/ your due date for HW6

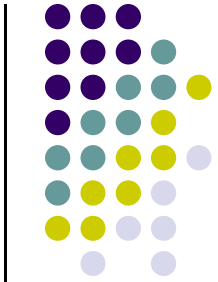
Parallel Scan Algorithm: Solution #1 Hillis & Steele (1986)



- Note that a implementation of the algorithm shown in picture requires two buffers of length n (shown is the case $n=8=2^3$)
- Assumption: the number n of elements is a power of 2: $n=2^M$



Hillis & Steele Parallel Scan Algorithm

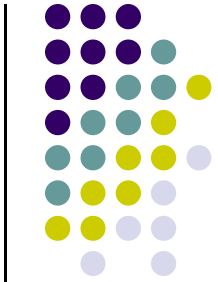


- Algorithm looks like this:

```
for d := 0 to M-1 do
    forall k in parallel do
            if k – 2^d ≥0 then
                    x[out][k] := x[in][k] + x[in][k – 2^d]
            else
                    x[out][k] := x[in][k]
    endforall
    swap(in,out)
endfor
```

Double-buffered version of the sum scan

Operation Count
Final Considerations



- The number of operations tally:

$$(2^M\text{-}2^0) + (2^M\text{-}2^1) + \ldots + (2^M\text{-}2^{k-1}) + \ldots + (2^M\text{-}2^{M-1})$$

  - Final operation count:

$$M \cdot 2^M - (2^0 + \ldots + 2^{M-1}) = M \cdot 2^M - 2^M + 1 = n(\log(n) - 1) + 1$$

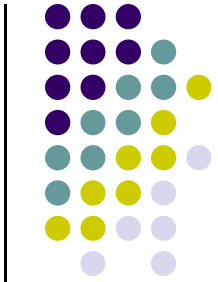
  - This is an algorithm with O(n*log(n)) work

- This scan algorithm is not that work efficient
  - Sequential scan algorithm only needs *n-1* additions
  - A factor of log(n) might hurt: 20x more work for 10^6 elements!
    - Homework requires a scan of about 16 million elements
  - Adding insult to injury: you need two buffers…

Hillis & Steele: Kernel Function

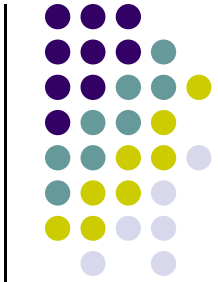
```
__global__ void scan(float *g_odata, float *g_idata, int n) {
    extern volatile __shared__  float temp[]; // allocated on invocation

    int thid = threadIdx.x;
    int pout = 0, pin = 1;

    // load input into shared memory.
    // Exclusive scan: shift right by one and set first element to 0
    temp[thid] = (thid == 0) ? 0: g_idata[thid-1];
    __syncthreads();

    for( int offset = 1; offset<n; offset <<= 1 ) {
        pout = 1 - pout; // swap double buffer indices
        pin  = 1 - pout;

        if (thid >= offset)
            temp[pout*n+thid] = temp[pin*n+thid] + temp[pin*n+thid - offset];
          else
            temp[pout*n+thid] = temp[pin*n+thid];

        __syncthreads(); // I need this here before I start next iteration
    }

    g_odata[thid] = temp[pout*n+thid]; // write output
}
```

Hillis & Steele: Kernel Function, Quick Remarks



- The kernel is very simple, which is good
- Note the `pin`/`pout` trick that was used to alternate the destination buffer
- The kernel only works when the entire array is processed by one block
  - One block in CUDA has at the most 1024 threads
  - In this setup we cannot handle yet 16 million entries, which is what your assignment will call for

Parallel Scan Algorithm: Solution #2 Harris-Sengupta-Owen (2007)



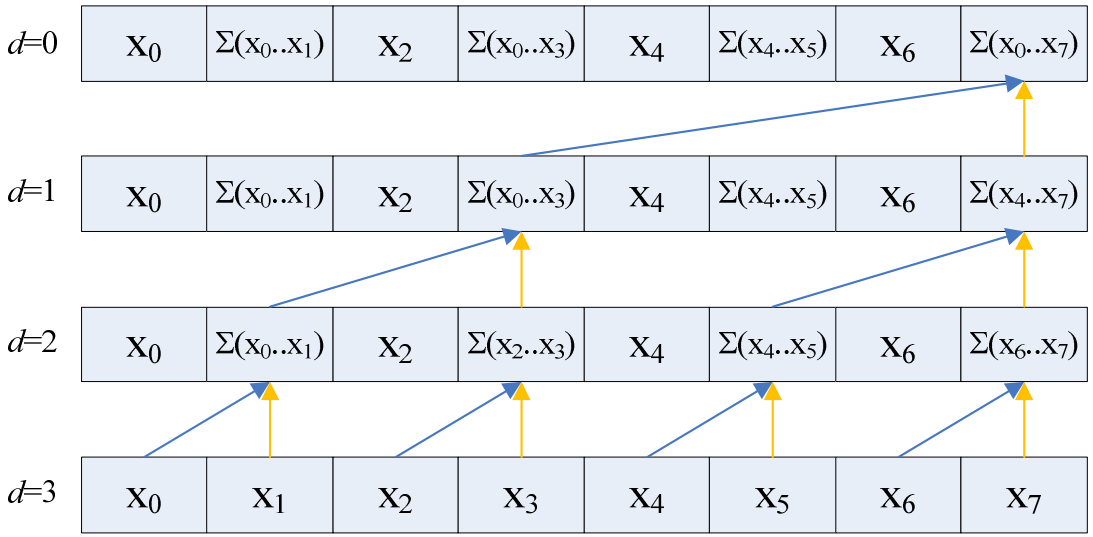
- A common parallel algorithm pattern:

  Balanced Trees

  - Build a balanced binary tree on the input data and sweep it to and then from the root
  - Tree is not an actual data structure, but a concept to determine what each thread does at each step

- For scan:
  - Traverse down from leaves to root building partial sums at internal nodes in the tree
    - Root holds sum of all leaves → this is a reduction algorithm
  - Traverse back up the tree building the scan from the partial sums
    - Called down-sweep phase

Picture and Pseudocode ~ Reduction Step~



$j \cdot 2^{k+1}-1 =$

1	3	5	7
3	7	-1	-1
7	-1	-1	-1

$j \cdot 2^{k+1}-2^{k}-1 =$

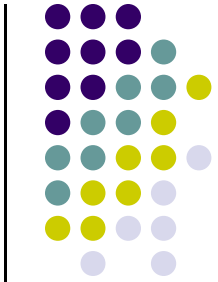
0	2	4	6
1	5	-1	-1
3	-1	-1	-1

```
for k=0 to M-1
        offset = 2^k
        for j=1 to 2^(M-k-1) in parallel do
                x[j·2^(k+1)-1] = x[j·2^(k+1)-1] + x[j·2^(k+1)-2^k-1]
        endfor
endfor
```

NOTE: "-1" entries indicate no-ops

NVIDIA [M. Harris]→

Operation Count, Reduce Phase

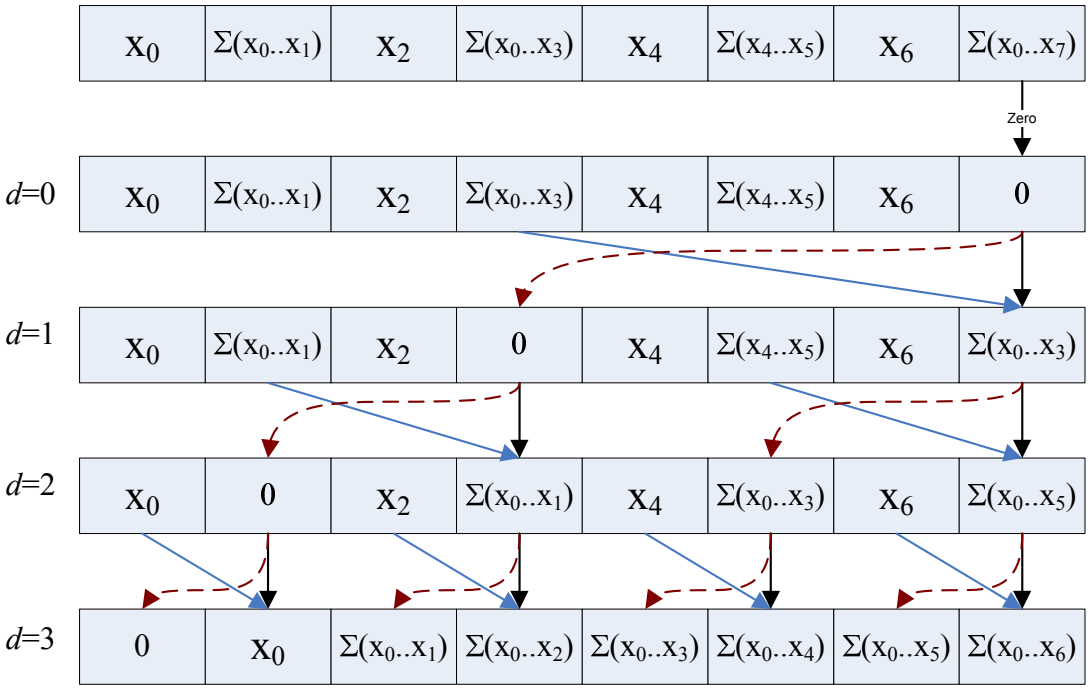


```
for k=0 to M-1
        offset = 2^k
        for j=1 to 2^(M-k-1) in parallel do
                x[j·2^(k+1)-1] = x[j·2^(k+1)-1] + x[j·2^(k+1)-2^k-1]
        endfor
endfor
```

By inspection: $\sum_{k=0}^{M-1} 2^{M-k-1} = 2^M - 1 = n - 1$

Looks promising…

The Down-Sweep Phase

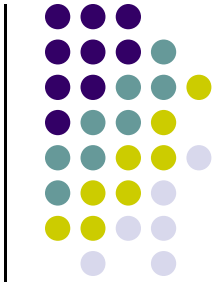


NOTE: This is just a mirror image of the reduction stage. Easy to come up with the indexing scheme…

```
for k=M-1 to 0
        offset = 2^k
        for j=1 to 2^(M-k-1) in parallel do
                dummy = x[j·2^(k+1)-2^k-1]
                x[j·2^(k+1)-2^k-1] = x[j·2^(k+1)-1]
                x[j·2^(k+1)-1] = x[j·2^(k+1)-1] + dummy
        endfor
endfor
```

NVIDIA [M. Harris]→

Down-Sweep Phase, Remarks

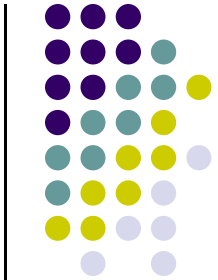


- Number of operations for the down-sweep phase:
  - Additions: n-1
  - Swaps: n-1 (each swap shadows an addition)

- Total number of operations associated with this algorithm
  - Additions: 2n-2
  - Swaps: n-1
  - Looks very comparable with the work load in the sequential solution

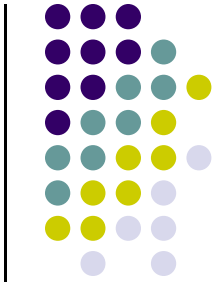
- The algorithm is convoluted though, it won't be easy to implement
  - Kernel shown on next slide

```
01|  __global__ void prescan(float *g_odata, float *g_idata, int n)
02| {
03|     extern volatile  __shared__  float temp[];// allocated on invocation
04|
05|
06|     int thid = threadIdx.x;
07|     int offset = 1;
08|
09|     temp[2*thid]   = g_idata[2*thid]; // load input into shared memory
10|     temp[2*thid+1] = g_idata[2*thid+1];
11|
12|     for (int d = n>>1; d > 0; d >>= 1) // build sum in place up the tree
13|     {
14|       __syncthreads();
15|
16|       if (thid < d)
17|       {
18|           int ai = offset*(2*thid+1)-1;
19|           int bi = offset*(2*thid+2)-1;
20|
21|           temp[bi] += temp[ai];
22|       }
23|       offset <<= 1;  //multiply by 2 implemented as bitwise operation
24|     }
25|
26|     if (thid == 0) { temp[n - 1] = 0; } // clear the last element
27|
28|     for (int d = 1; d < n; d *= 2) // traverse down tree & build scan
29|     {
30|         offset >>= 1;
31|         __syncthreads();
32|
33|         if (thid < d)
34|         {
35|            int ai = offset*(2*thid+1)-1;
36|            int bi = offset*(2*thid+2)-1;
37|
38|            float t   = temp[ai];
39|            temp[ai]  = temp[bi];
40|            temp[bi] += t;
41|          }
42|     }
43|
44|     __syncthreads();
45|
46|     g_odata[2*thid]   = temp[2*thid]; // write results to device memory
47|     g_odata[2*thid+1] = temp[2*thid+1];
48| }
```



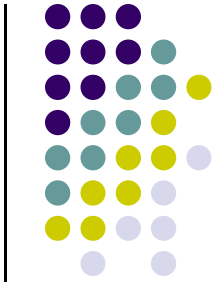
NVIDIA [M. Harris]→

Going Beyond 2048 Entries [1/3]



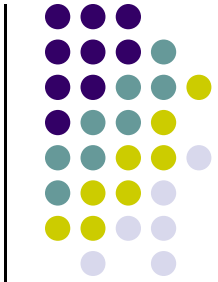
- Upon first invocation of the kernel (kernel #1), each will bring into shared memory 2048 elements:
  - 1024 “lead” elements (see vertical arrows ↑ on slide 9), and…
  - 1024 mating elements (the blue, oblique, arrows on slide 9)
  - Two consecutive “lead” elements are separated by a stride of $k=2^1$
  - A “lead” element and its “mating” element are separated by a stride of $k/2=1$
- Suppose you take 6 reduction steps in this first kernel and bail out after writing into the global memory the preliminary data that you computed and stored in shared memory
- The next kernel invocation should pick up the unfinished business where the previous kernel left…
  - Call this a “flawless reentry requirement”

Going Beyond 2048 Entries [2/3]



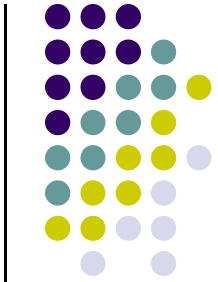
- Upon the second next kernel call, each block will bring into shared memory 2048 elements:
  - 1024 “lead” elements, and…
  - 1024 “mating” elements
  - Two consecutive “lead” elements will now be separated by a stride of $k=2^6$
  - A “lead” element and its “mating” element are separated by a stride of $k/2=2^5$
    - Thus, when brining in data from global memory, you are not going to bring over a contiguous chunk of memory of size 2048, rather you’ll have to jump 2^5 locations between successive “lead and mating element” pairs
  - However, once you bring data in shared memory, you process as before
  - Before you exit kernel #2 you have to write back data from shared memory into global memory
    - Again, you have to choreograph this shared to global memory store since there is a 2^5 stride that comes into play
  - If you exit kernel #2 after say 4 more reduction steps, the next time you re-enter the kernel (#3) you will have $k=2^{10}$

Going Beyond 2048 Entries
[3/3]



- You will continue the reduction stage until the stride is 2^{M-1}
  - At this point you are ready to start the down-sweep phase
  - Down-sweep phase carried out in a similar fashion: we will have to invoke the kernel several times
  - Always work in shared memory and copy back data to global memory before bailing out
- The challenges here are:
  - Understanding the indexing into the global memory to bring data to ShMem
  - How to loop across the data in shared memory
- There are very many shared memory bank conflicts since you move with strides that are power of 2
- Advanced topic: get rid of the bank conflict through padding

Concluding Remarks, Parallel Scan

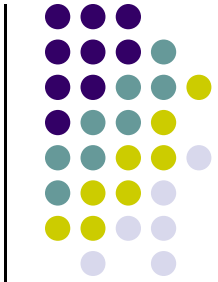


- Intuitively, the scan operation is not the type of procedure ideally suited for parallel computing
  - Even if it doesn't fit like a glove, leads to nice speedup:

# elements	CPU Scan (ms)	GPU Scan (ms)	Speedup
1024	0.002231	0.079492	0.03
32768	0.072663	0.106159	0.68
65536	0.146326	0.137006	1.07
131072	0.726429	0.200257	3.63
262144	1.454742	0.326900	4.45
524288	2.911067	0.624104	4.66
1048576	5.900097	1.118091	5.28
2097152	11.848376	2.099666	5.64
4194304	23.835931	4.062923	5.87
8388688	47.390906	7.987311	5.93
16777216	94.794598	15.854781	5.98

Source: 2007 paper of Harris, Sengupta, Owens

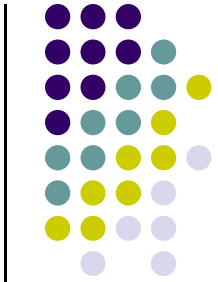
Concluding Remarks, Parallel Scan



- The Hillis-Steele (HS) implementation is simple, but suboptimal
- The Harris-Sengupta-Owen (HSO) solution is convoluted, but O(n) scaling
  - The complexity of the algorithm due to an acute bank-conflict situation
- Finally, we have not solved the problem yet: we only looked at the case when our array has up to 1024 elements
  - You will have to think how to handle the 16,777,216=2^{24} elements case
  - Likewise, it would be fantastic if you implement as well the case when the number of elements is not a power of 2

Bank Conflicts Discussion

[Advanced Topics – Supplementary Material]



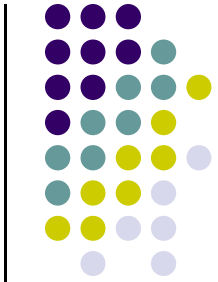
- No penalty if all threads access different banks
  - Or if threads read from the exact same address (multicasting/broadcasting)

- This is not the case here: multiple threads access the same shared memory bank with different addresses; i.e. different rows of a bank
  - We have something like $2^{k+1} \cdot j-1$
    - k=0: two way bank conflict
    - k=1: four way bank conflict
    - …

- Recall that shared memory accesses with conflicts are serialized
  - N-bank memory conflicts lead to a set of N successive shared memory transactions

Initial Bank Conflicts on Load

[Advanced Topics – Supplementary Material]



- Each thread loads two shared memory data elements
- Tempting to interleave the loads (see lines 9 & 10, and 46 & 47)

```
temp[2*thid]   = g_idata[2*thid];
temp[2*thid+1] = g_idata[2*thid+1];
```

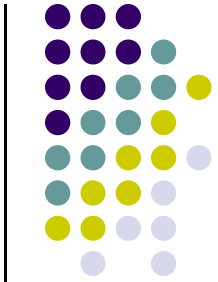
  - Thread 0 accesses banks 0 and 1
  - Thread 1 accesses banks 2 and 3
  - …
  - Thread 8 accesses banks 16 and 17. Oops, that's 0 and 1 again…
    - Two way bank conflict, can't be easily eliminated
- Better to load one element from each half of the array

```
temp[thid]         = g_idata[thid];
temp[thid + (n/2)] = g_idata[thid + (n/2)];
```

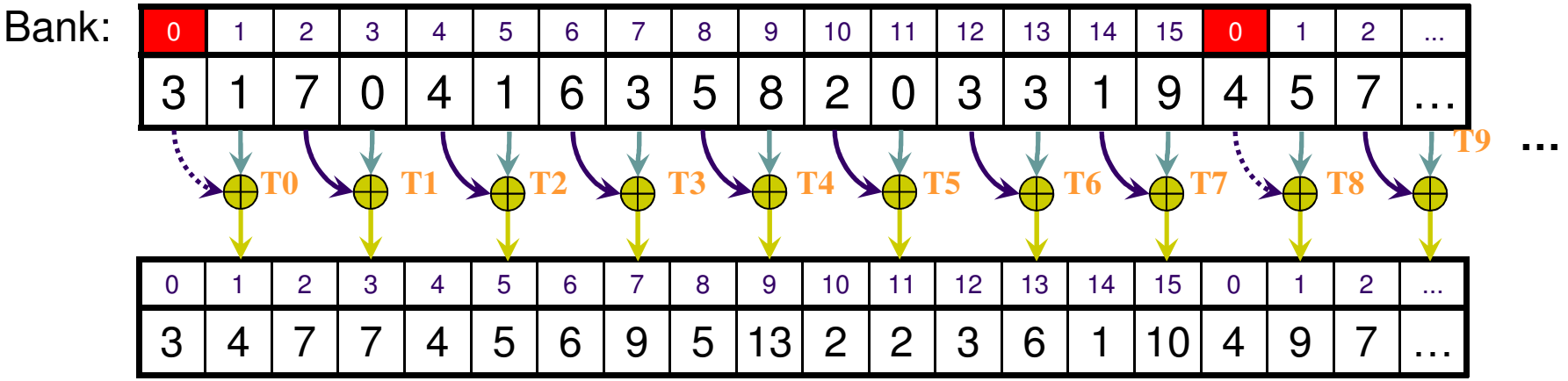
- Solution above is helping with the global memory bandwidth as well…

Bank Conflicts in the Tree Algorithm

[Advanced Topics – Supplementary Material]



- When we build the sums, during the first iteration of the algorithm each thread in a half-warp reads two shared memory locations and writes one
- We have bank conflicts: Threads (0 & 8) access bank 0 at the same time, and then bank 1 at the same time



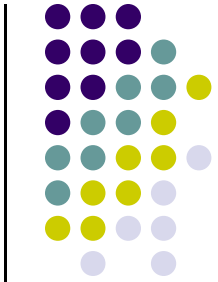
First iteration: 2 threads access each of 8 banks.

Each ⊕ corresponds to a single thread.

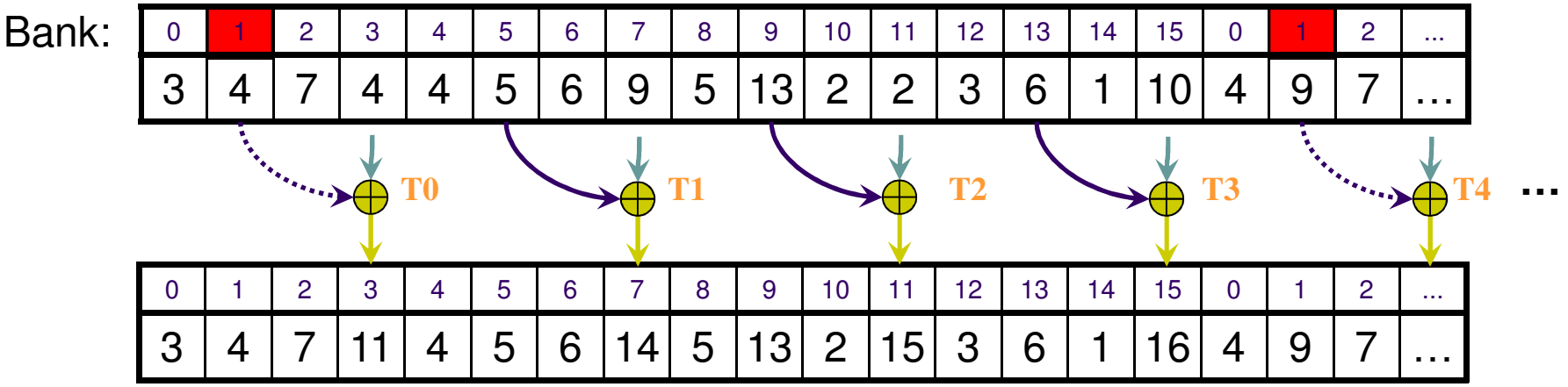
Like-colored arrows represent simultaneous memory accesses

Bank Conflicts in the tree algorithm

[Advanced Topics – Supplementary Material]



- 2nd iteration: even worse!
  - 4-way bank conflicts; for example:
    Th(0,4,8,12) access bank 1, Th(1,5,9,13) access Bank 5, etc.



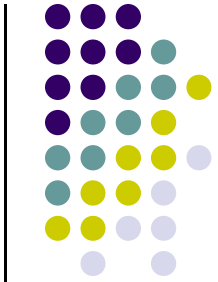
2nd iteration: 4 threads access each of 4 banks.

Each ⊕ corresponds to a single thread.

Like-colored arrows represent simultaneous memory accesses

Managing Bank Conflicts in the Tree Algorithm

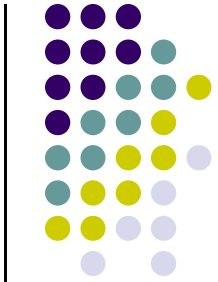
[Advanced Topics – Supplementary Material]



- Use padding to prevent bank conflicts
  - Add a word of padding every 16 words.
    - Now you work with a virtual 17 bank shared memory layout
  - Within a 16-thread half-warp, all threads access different banks
    - They are aligned to a 17 word memory layout
  - It comes at a price: you have memory words that are wasted
  - Keep in mind: you should also load data from global into shared memory using the virtual memory layout of 17 banks

Use Padding to Reduce Conflicts

[Advanced Topics – Supplementary Material]



- After you compute a ShMem address like this:

```
address = 2 * stride * thid;
```

- Add padding like this:

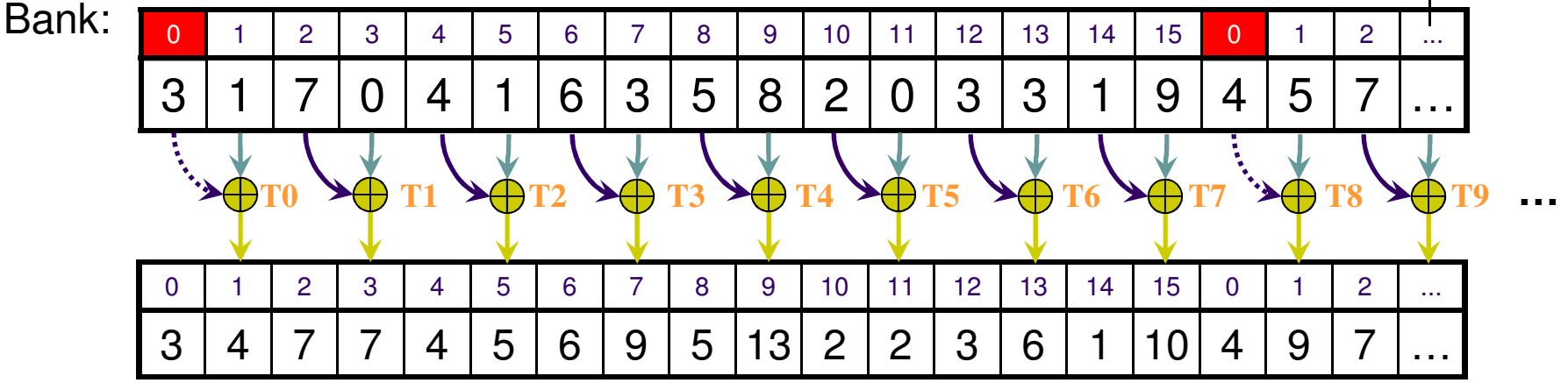
```
address += (address >> 4); // divide by NUM_BANKS
```

- This removes most bank conflicts
  - Not all, in the case of deep trees
  - Material posted online will contain a discussion of this “deep tree” situation along with a proposed solution

Managing Bank Conflicts in the Tree Algorithm

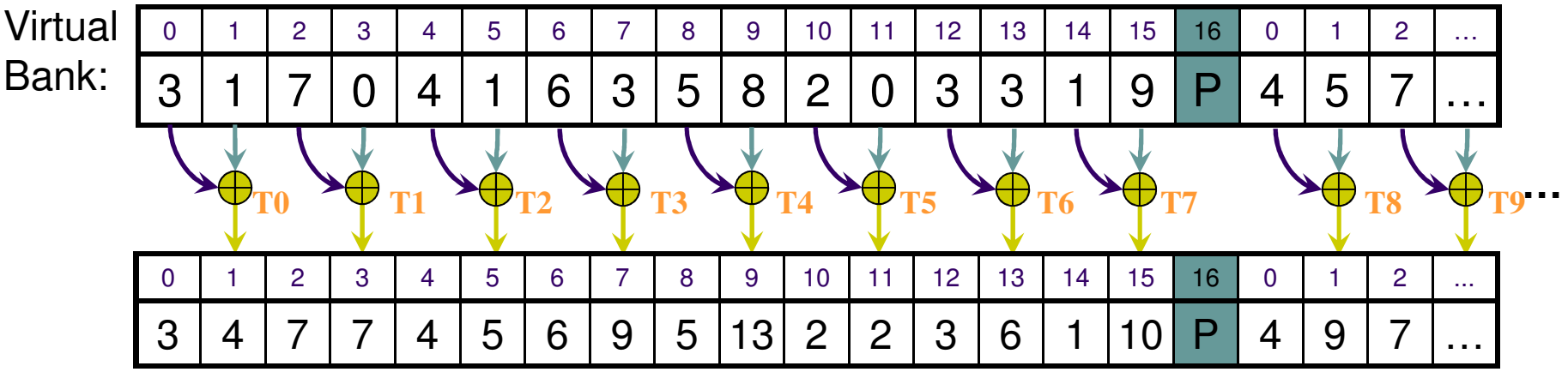
[Advanced Topics – Supplementary Material]

Original scenario.



Modified scenario, virtual 17 bank memory layout.

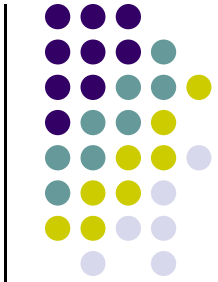
Actual physical memory (true bank number)
(0) (1) (2) (3)



Note that only arrows with the same color happen simultaneously.

Bank Conflicts Discussion

[Advanced Topics – Supplementary Material]



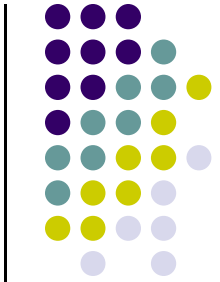
- No penalty if all threads access different banks
  - Or if threads read from the exact same address (multicasting/broadcasting)

- This is not the case here: multiple threads access the same shared memory bank with different addresses; i.e. different rows of a bank
  - We have something like $2^{k+1} \cdot j-1$
    - k=0: two way bank conflict
    - k=1: four way bank conflict
    - …

- Recall that shared memory accesses with conflicts are serialized
  - N-bank memory conflicts lead to a set of N successive shared memory transactions

Initial Bank Conflicts on Load

[Advanced Topics – Supplementary Material]



- Each thread loads two shared memory data elements
- Tempting to interleave the loads (see lines 9 & 10, and 46 & 47)

```
temp[2*thid]   = g_idata[2*thid];
temp[2*thid+1] = g_idata[2*thid+1];
```

  - Thread 0 accesses banks 0 and 1
  - Thread 1 accesses banks 2 and 3
  - …
  - Thread 8 accesses banks 16 and 17. Oops, that's 0 and 1 again…
    - Two way bank conflict, can't be easily eliminated

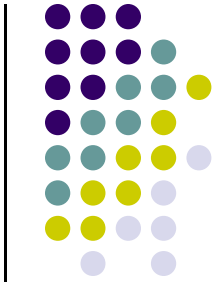
- Better to load one element from each half of the array

```
temp[thid]         = g_idata[thid];
temp[thid + (n/2)] = g_idata[thid + (n/2)];
```

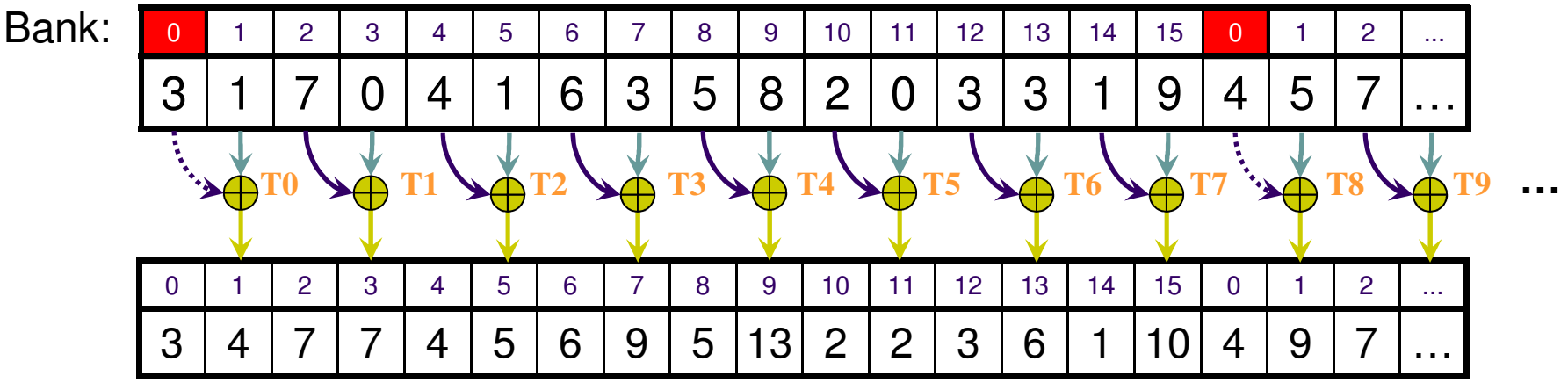
- Solution above is helping with the global memory bandwidth as well…

Bank Conflicts in the Tree Algorithm

[Advanced Topics – Supplementary Material]



- When we build the sums, during the first iteration of the algorithm each thread in a half-warp reads two shared memory locations and writes one
- We have bank conflicts: Threads (0 & 8) access bank 0 at the same time, and then bank 1 at the same time



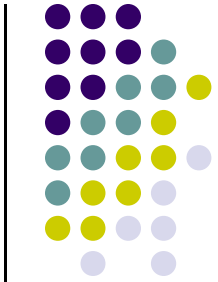
First iteration: 2 threads access each of 8 banks.

Each ⊕ corresponds to a single thread.

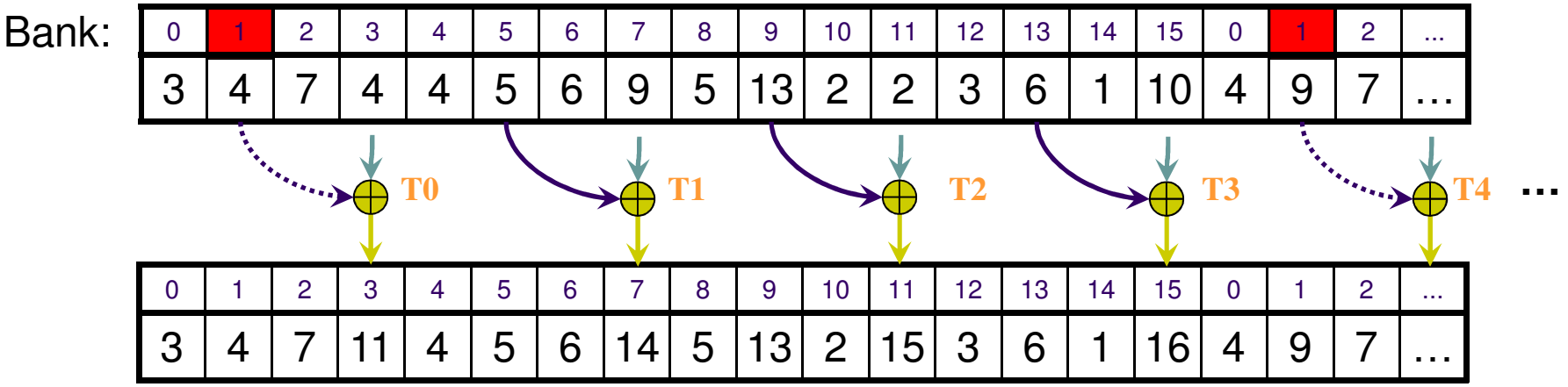
Like-colored arrows represent simultaneous memory accesses

Bank Conflicts in the tree algorithm

[Advanced Topics – Supplementary Material]



- 2nd iteration: even worse!
  - 4-way bank conflicts; for example:
    Th(0,4,8,12) access bank 1, Th(1,5,9,13) access Bank 5, etc.



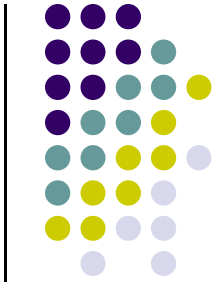
2nd iteration: 4 threads access each of 4 banks.

Each ⊕ corresponds to a single thread.

Like-colored arrows represent simultaneous memory accesses

Managing Bank Conflicts in the Tree Algorithm

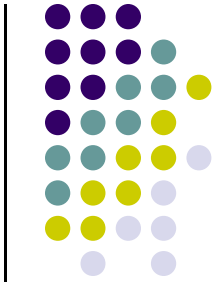
[Advanced Topics – Supplementary Material]



- Use padding to prevent bank conflicts
  - Add a word of padding every 16 words.
    - Now you work with a virtual 17 bank shared memory layout
  - Within a 16-thread half-warp, all threads access different banks
    - They are aligned to a 17 word memory layout
  - It comes at a price: you have memory words that are wasted
  - Keep in mind: you should also load data from global into shared memory using the virtual memory layout of 17 banks

Use Padding to Reduce Conflicts

[Advanced Topics – Supplementary Material]



- After you compute a ShMem address like this:

```
address = 2 * stride * thid;
```

- Add padding like this:

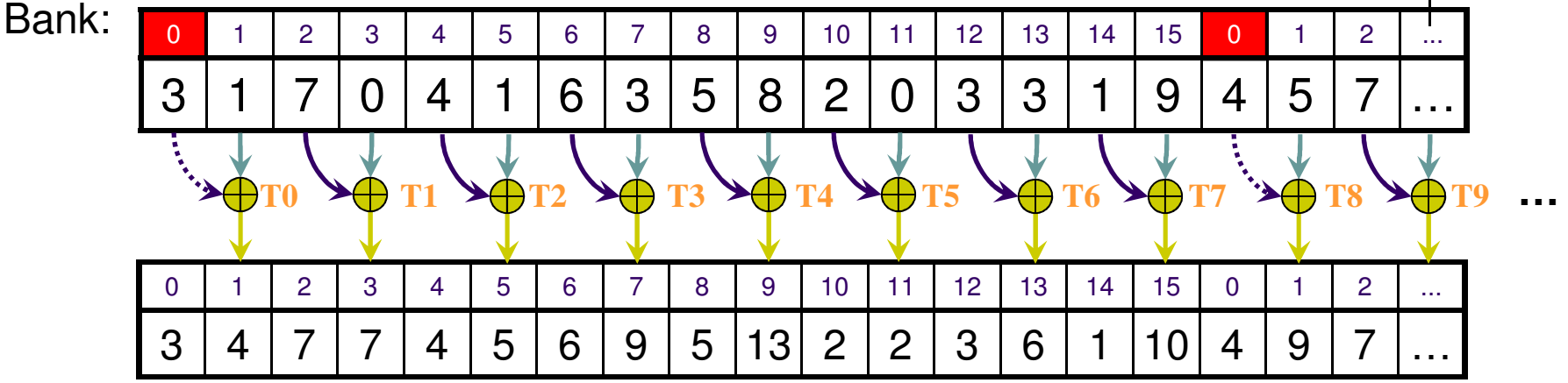
```
address += (address >> 4); // divide by NUM_BANKS
```

- This removes most bank conflicts
  - Not all, in the case of deep trees
  - Material posted online will contain a discussion of this “deep tree” situation along with a proposed solution

Managing Bank Conflicts in the Tree Algorithm

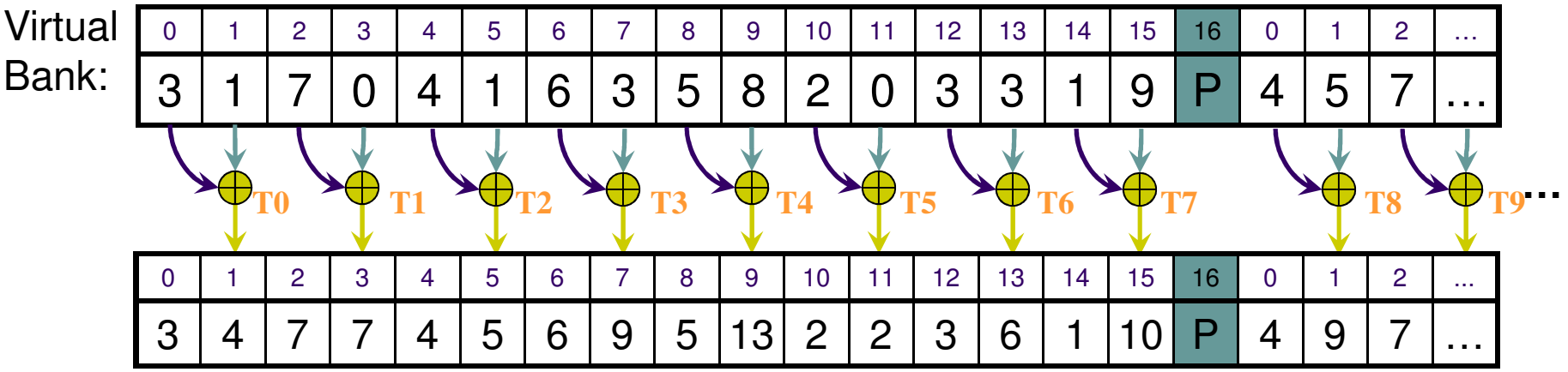
[Advanced Topics – Supplementary Material]

Original scenario.

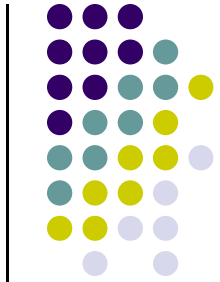


Modified scenario, virtual 17 bank memory layout.

Actual physical memory (true bank number)
(0) (1) (2) (3)

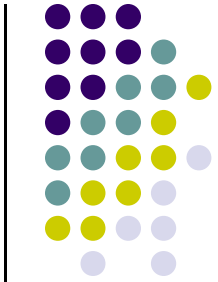


Note that only arrows with the same color happen simultaneously.



CUDA Streams

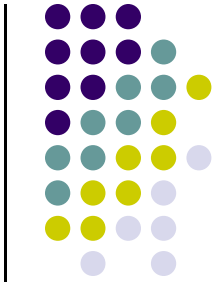
CUDA Streams: Why Bother?



- In the CPU-GPU interplay, a CUDA enabled GPU can count on two engines
  - An execution engine
  - A copy engine, which actually has 2 subengines that can work simultaneously
    - A H2D copy subengine
    - A D2H copy subengine
- Goal of this segment: learn how to use both engines at the same time
- Remark:
  - In this segment of the lecture the important things happen on the host side, not on the device side

[Preamble: 1/3]

Asynchronous Concurrent Execution



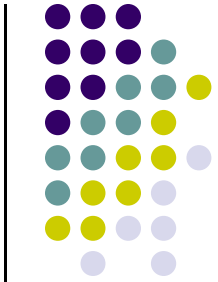
- In order to facilitate concurrent execution on host and device, some function calls are asynchronous
  - Control is returned to the host thread before the device has completed the requested task

- Examples of asynchronous calls
  - Kernel launches
  - Device ↔ device memory copies
  - Host ↔ device memory copies of a memory block of 64 KB or less
  - Memory copies performed by functions that are suffixed with Async

- NOTE: When an application is run via a CUDA debugger or profiler (**`cuda-gdb`**, **`nvvp`**, Parallel Nsight), all launches are synchronous

[Preamble: 2/3]

Host-Device Data Transfer Issues



- In general, host ↔ device data transfers using `cudaMemcpy()` are blocking
  - Control is returned to the host thread only after the data transfer is complete

- There is a non-blocking variant, `cudaMemcpyAsync()`

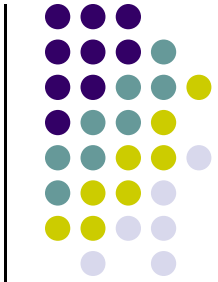
```
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, 0);
kernel<<<grid,block>>>(a_d);
cpuFunction();
```

  - The host does not wait on the device to finish the mem copy and the kernel call for it to start execution of `cpuFunction()` call
  - The launch of “kernel” only happens after the mem copy call finishes

- NOTE 1: the asynchronous transfer version requires pinned host memory (allocated with `cudaHostAlloc()`), and it contains an additional argument (a stream ID)
- NOTE 2: up until this point we are still not using the two GPU engines
  - We only make the CPU stay busy (which is nonetheless quite good)

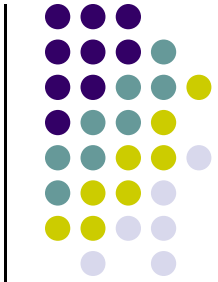
[Preamble: 3/3]

Overlapping Host ↔ Device Data Transfer with Device Execution



- When is this overlapping useful?
  - Imagine a kernel executes on the device and only works with the lower half of the device global memory
  - Then, you can copy data from host to device into the upper half of the device global memory
  - These two operations can take place simultaneously
- Note that there is an issue with this idea:
  - The device execution stack is FIFO, one function call on the device is not serviced until all the previous device function calls completed
  - This would prevent overlapping execution with data transfer
- This issue was addressed by the use of CUDA “streams”

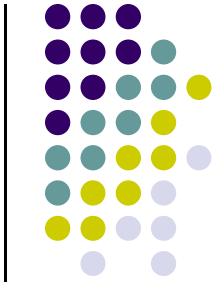
CUDA Streams: Overview



- A programmer can manage concurrency through *streams*

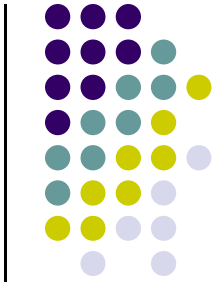
- A stream is a sequence of CUDA commands that execute in issue-order
  - Look at a stream as a queue of GPU operations
  - The execution order in a stream is identical to the order in which the GPU operations are added to the stream
  - NOTE: an operation in a stream does not commence prior to the previous operation being fully completed
    - There is a distinction between queuing an operation in a stream and the moment when it actually starts to be executed on the GPU

CUDA Streams: Overview
[Cntd.]



- One host thread can define multiple CUDA streams
- What are the typical operations in a stream?
  - Invoking a data transfer
  - Invoking a kernel execution
  - Handling events
- With respect to each other, different CUDA streams execute their commands as they see fit
  - Inter-stream relative behavior is not guaranteed and should therefore not be relied upon for correctness (e.g. inter-kernel communication for kernels allocated to different streams is undefined)
  - Another way to look at it: streams can be synchronized at barrier points, but correlation of sequence execution within different streams is not supported

CUDA Streams: Creation



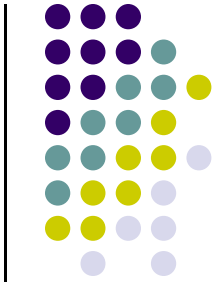
- A stream is defined by creating a stream object
  - It is subsequently used by specifying it as the stream parameter to a sequence of kernel launches and host ↔ device memory copies
- The following code sample creates two streams and allocates an array "`hostPtr`" of float in page-locked memory
  - `hostPtr` will be used in asynchronous host ↔ device memory transfers

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
        cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
```

- NOTE: As soon you invoke a CUDA function you create a default stream (stream 0)
  - If you don't explicitly state a stream in the execution configuration of a kernel it is assumed it's launched as part of stream 0

Notice the length of the array

CUDA Streams: Making Use of Them

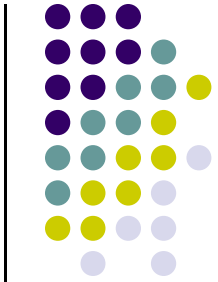


- In the code below, each of the two streams is defined as a sequence of
  - One memory copy from host to device,
  - One kernel launch, and
  - One memory copy from device to host

```
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size, cudaMemcpyHostToDevice, stream[i]);

    MyKernel<<<100, 512, 0, stream[i]>>>(outputDevPtr + i * size, inputDevPtr + i * size, size);

    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,size, cudaMemcpyDeviceToHost, stream[i]);
}
```

- There are some wrinkles to it, we'll revisit shortly…

CUDA Streams: Clean Up Phase

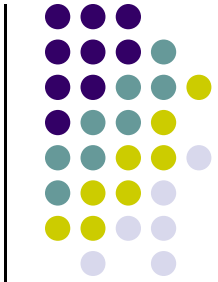


- Streams are released by calling `cudaStreamDestroy()`

```
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

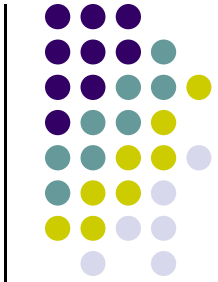
- `cudaStreamDestroy()` waits for all preceding commands in the given stream to complete before destroying the stream and returning control to the host thread

CUDA Streams: Caveats



- Two commands from different streams cannot run concurrently if either one of the following operations is issued in-between them by the host thread:
  - A page-locked host memory allocation,
  - A device memory allocation,
  - A device memory set,
  - A device $\leftrightarrow$ device memory copy,
  - Any CUDA command to stream 0 (including kernel launches and host $\leftrightarrow$ device memory copies that do not specify any stream parameter)
  - A switch between the L1/shared memory configurations

CUDA Streams: Synchronization Aspects



`cudaDeviceSynchronize()` halts execution on the host until all preceding commands in all CUDA streams have completed

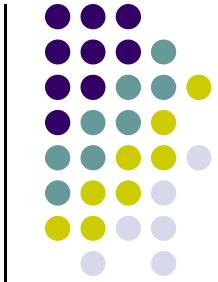
`cudaStreamSynchronize()` takes a stream as a parameter and halts execution on the host until all preceding commands in the given CUDA stream have completed. It can be used to synchronize the host with a specific stream, allowing other streams to continue executing on the device

`cudaStreamWaitEvent()` takes a CUDA stream and an event as parameters and makes all the commands added to the given stream after the call to `cudaStreamWaitEvent()` delay their execution until the given event has completed. Note: this halts the execution of tasks in a stream!

`cudaStreamQuery()` provides applications with a way to know if all preceding commands in a stream have completed

- NOTE: To avoid unnecessary slowdowns, all these synchronization functions are usually best used for timing purposes or to isolate a launch or memory copy that is failing

Example: Use of `cudaStreamWaitEvent`



- Assume `stream1` and `stream2` have been defined/initialized already
- The point of this example:
  - Use the two copy subengines at the same time
  - Wait onto the launching of the `myKernel` until the copy in stream 1 is finished

```
cudaEvent_t event;
cudaEventCreate (&event);                                   // create event

cudaMemcpyAsync ( d_in, in, size, H2D, stream1 );           // 1) H2D copy of new input
cudaEventRecord (event, stream1);                           // record event

cudaMemcpyAsync ( out, d_out, size, D2H, stream2 );         // 2) D2H copy of previous result

cudaStreamWaitEvent ( stream2, event );                     // wait for event in stream1
myKernel<<< 1000, 512, 0, stream2 >>> ( d_in, d_out );      // 3) GPU must wait for 1 and 2
someCPUfunction ( blah )                                    // this gets executed right away
```


[S. Rennich]→