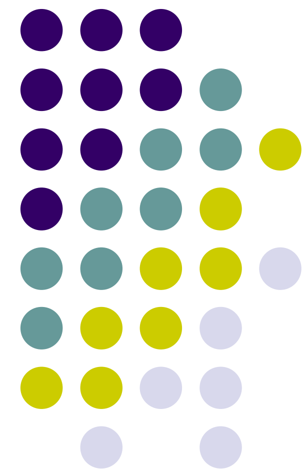
ME964
High Performance Computing for Engineering Applications



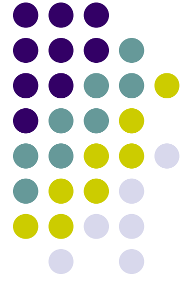
CUDA Optimization Rules of Thumb
Parallel Prefix Scan

March 8, 2011

"There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."
Sir C. A. R. Hoare

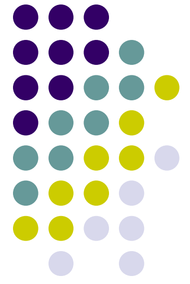
© Dan Negrut, 2011
ME964 UW-Madison

Before We Get Started…



- Last time
  - Thread divergence on the GPU
  - Execution Configuration Optimization
  - Instruction Optimization
- Today
  - Summarize lessons learned in a collection of optimization rules of thumb
  - Discuss parallel prefix scan on the GPU
    - This is your next homework, due 03/22
- Other issues
  - The "easy way out" Midterm Project was emailed to you
  - There are three students who haven't indicated their Midterm Project choice
  - HW6 was posted, due on 03/22
    - Please remember that you have reading assigned as well (see online Syllabus)

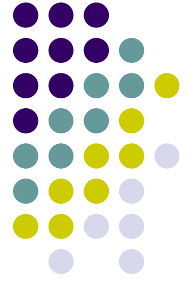
Performance Optimization



- Performance optimization revolves around three basic strategies:
  - Maximizing parallel execution
  - Optimizing memory usage to achieve maximum memory bandwidth
  - Optimizing instruction usage to achieve maximum instruction throughput

- Writing CUDA software is a craft
  - Sometimes having to deal with conflicting requirements
  - A list of recommendations is provided. Sections that are referenced are as in the CUDA C Best Practices Guide Version 3.2

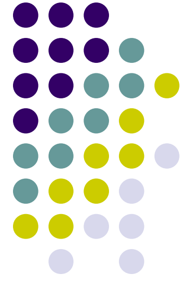
Writing CUDA Software: High-Priority Recommendations



1. To get the maximum benefit from CUDA, focus first on finding ways to parallelize sequential code. (Section 1.1.3)
2. Use the effective bandwidth of your computation as a metric when measuring performance and optimization benefits. (Section 2.2)
3. Minimize data transfer between the host and the 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. (Section 3.1)

http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf

Writing CUDA Software: High-Priority Recommendations



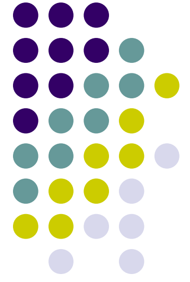
4. Ensure global memory accesses are coalesced whenever possible. (Section 3.2.1)

5. Minimize the use of global memory. Prefer shared memory access where possible. (Section 5.2)

6. Avoid different execution paths within the same warp. (Section 6.1)

http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf

Writing CUDA Software: Medium-Priority Recommendations



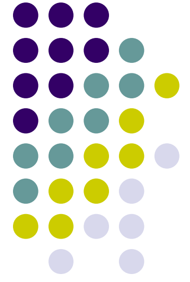
1. Accesses to shared memory should be designed to avoid serializing requests due to bank conflicts. (Section 3.2.2.1)

2. To hide latency arising from register dependencies, maintain sufficient numbers of active threads per multiprocessor (i.e., sufficient occupancy). (Sections 3.2.6 and 4.3)

3. The number of threads per block should be a multiple of 32 threads, because this provides optimal computing efficiency and facilitates coalescing. (Section 4.4)

http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf

Writing CUDA Software: Medium-Priority Recommendations



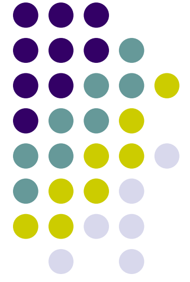
4. Use the fast math library whenever speed trumps precision. (Section 5.1.4)

5. Prefer faster, more specialized math functions over slower, more general ones when possible. (Section 5.1.4)

6. Use signed integers rather than unsigned integers as loop counters. (Section 6.3)

http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf

Writing CUDA Software: Low-Priority Recommendations



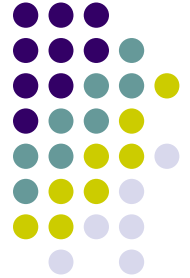
1. For kernels with long argument lists, place some arguments into constant memory to save shared memory. (Section 3.2.2.4)

2. Use shift operations to avoid expensive division and modulo calculations. (Section 5.1.1)

3. Avoid automatic conversion of doubles to floats. (Section 5.1.3)

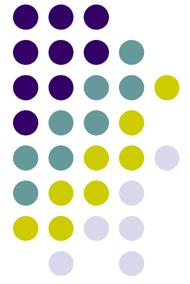
4. Make it easy for the compiler to use branch predication in lieu of loops or control statements. (Section 6.2)

http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf



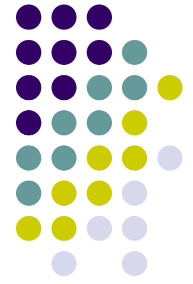
End Optimization Issues
Start Parallel Prefix Scan

Objectives of This Exercise



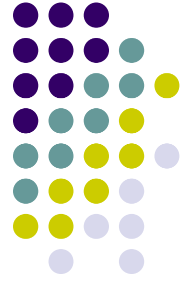
- The vehicle for the software design exercise: implementation of a parallel prefix sum operation
  - Recall first assignment, also the topic of assignment #6
- Goal 1: Putting your CUDA knowledge to work
- Goal 2: Understand that
  - Different algorithmic designs lead to different performance levels
  - Different constraints dominate in different applications and/or design solutions
  - Case studies help to establish intuition, idioms and ideas
- Goal 3: Identify parallel algorithm patterns that can result in superior performance
  - Understand that there are patterns and it's worth being aware of them
  - If you want, these are the tricks of the trade
  - When considering patterns, you can't lose sight of the underlying hardware

Running Code on Parallel Computers



- You come to rely on compiler to figure out the parallelism in a piece of code and then map it to an underlying hardware
  - VERY hard, the holy grail in parallel computing
- You rely on parallel libraries built for a specific underlying hardware
  - Very convenient, the way to go when such libraries are available
- You rely on language extensions to facilitate the process of generating a parallel executable
  - This is where you are with CUDA
  - Presents a great opportunity to screw up or to generate some tailored code that takes care of your *specific* application in an efficient fashion

Parallel Prefix Sum (Scan)



- Definition:

  The all-prefix-sums operation takes a binary associative operator ⊕ with identity ***I***, and an array of n elements

  $$[a_0, a_1, \ldots, a_{n-1}]$$

  and returns the ordered set

  $$[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})].$$

- Example:
  if ⊕ is addition, then scan on the set

  [3 1 7 0 4 1 6 3]

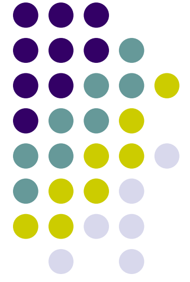
  returns the set

  [0 3 4 11 11 15 16 22]

Exclusive scan: last input element is not included in the result

(From Blelloch, 1990, "Prefix Sums and Their Applications)

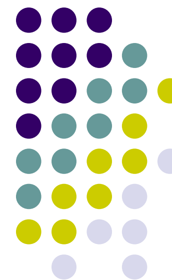
Scan on the CPU



```
void scan( float* scanned, float* input, int length)
{
  scanned[0] = 0;
  for(int i = 1; i < length; ++i)
  {
    scanned[i] = scanned[i-1] + input[i-1];
  }
}
```

- Just add each element to the sum of the elements before it
- Trivial, but sequential
- Exactly *n-1* adds: optimal in terms of work efficiency

Applications of Scan



- Scan is a simple and useful parallel building block
  - Convert recurrences from sequential …

```
for(j=1;j<n;j++)
   out[j] = out[j-1] + f(j);
```

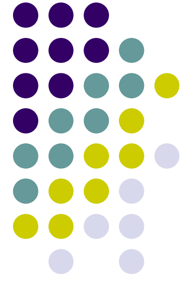
  - … into parallel:

```
forall(j) in parallel
     temp[j] = f(j);
scan(out, temp);
```

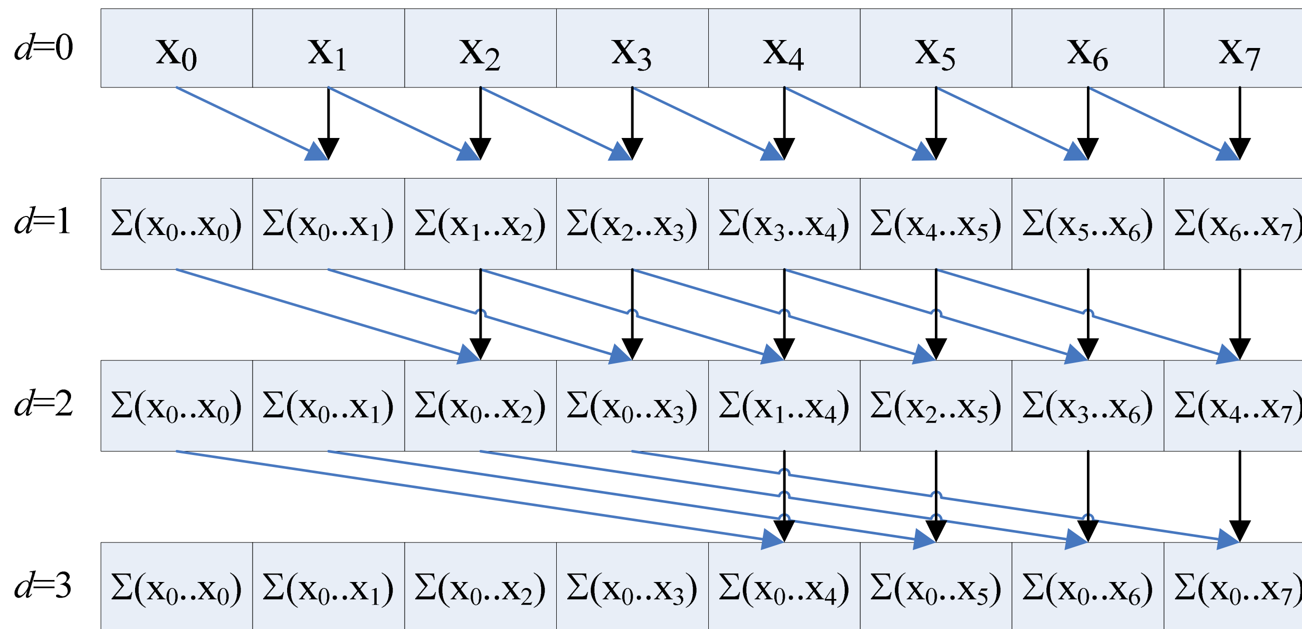
- Useful in implementation of several parallel algorithms:

• Radix sort	• Polynomial evaluation
• Quicksort	• Solving recurrences
• String comparison	• Tree operations
• Lexical analysis	• Histograms
• Stream compaction	• Etc.

Parallel Scan Algorithm: Solution One Hillis & Steele (1986)

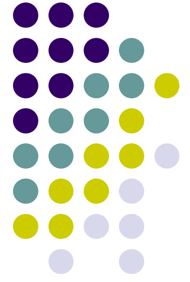


- Note that a implementation of the algorithm shown in picture requires two buffers of length n (shown is the case $n=8=2^3$)
- Assumption: the number n of elements is a power of 2: $n=2^M$



Picture courtesy of Mark Harris

The Plain English Perspective



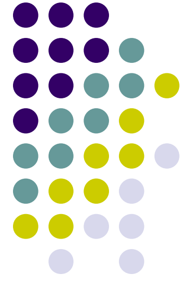
- First iteration, I go with stride $1=2^0$
  - Start at $x[2^M]$ and apply this stride to all the array elements before $x[2^M]$ to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have 2^M-1 additions

- Second iteration, I go with stride $2=2^1$
  - Start at $x[2^M]$ and apply this stride to all the array elements before $x[2^M]$ to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have $2^M - 2^1$ additions

- Third iteration: I go with stride $4=2^2$
  - Start at $x[2^M]$ and apply this stride to all the array elements before $x[2^M]$ to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have $2^M - 2^2$ additions

- … (and so on)

The Plain English Perspective



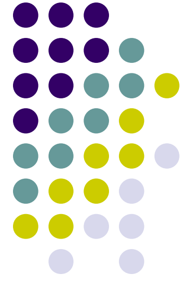
- Consider the kth iteration (where $1<k<M-1$): I go with stride 2^{k-1}
  - Start at $x[2^M]$ and apply this stride to all the array elements before $x[2^M]$ to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have 2^M-2^{k-1} additions

- …

- M^{th} iteration: I go with stride 2^{M-1}
  - Start at $x[2^M]$ and apply this stride to all the array elements before $x[2^M]$ to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have 2^M-2^{M-1} additions

- NOTE: There is no $(M+1)^{th}$ iteration since this would automatically put me beyond the bounds of the array (if you apply an offset of 2^M to "$\&x[2^M]$ " it places you right before the beginning of the array – not good…)

Hillis & Steele Parallel Scan Algorithm

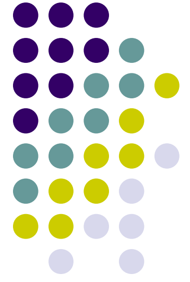


- Algorithm looks like this:

```
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:
  - $(2^M-2^0) + (2^M-2^1) + \ldots + (2^M-2^{k-1}) + \ldots + (2^M-2^{M-1})$
  - Final operation count:

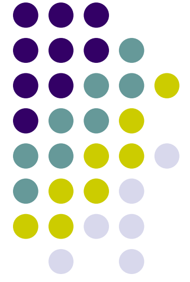
$$M \cdot 2^M - (2^0 + \ldots + 2^{M-1}) = M \cdot 2^M - 2^M + 1 = n(\log(n) - 1) + 1$$

  - 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
  - Additionally, you need two buffers…

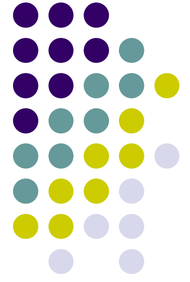
- A parallel algorithm can be slow when execution resources are saturated due to low algorithm efficiency

Hillis & Steele: Kernel Function



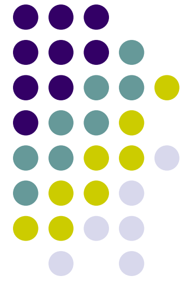
```
__global__ void scan(float *g_odata, float *g_idata, int n)
{
    extern  __shared__  float temp[]; // allocated on invocation

    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) ? g_idata[thid-1] : 0;
    __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 - offset];
           else
            temp[pout*n+thid] = temp[pin*n+thid];

        __syncthreads();
    }

    g_odata[thid] = temp[pout*n+thid1]; // write output
}
```

Hillis & Steele: Kernel Function, Quick Remarks



- The kernel is very simple, which is good
- Note the nice trick that was used to swap the buffers
- The kernel only works when the entire array is processed by one block
  - One block in CUDA has 512 threads, which means I can have up to 1024 elements (short of 16 million, which is your assignment)
  - This needs to be improved upon…

Improving Efficiency



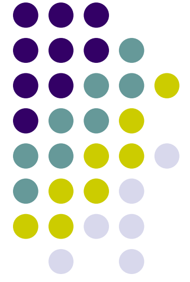
- A common parallel algorithm pattern:

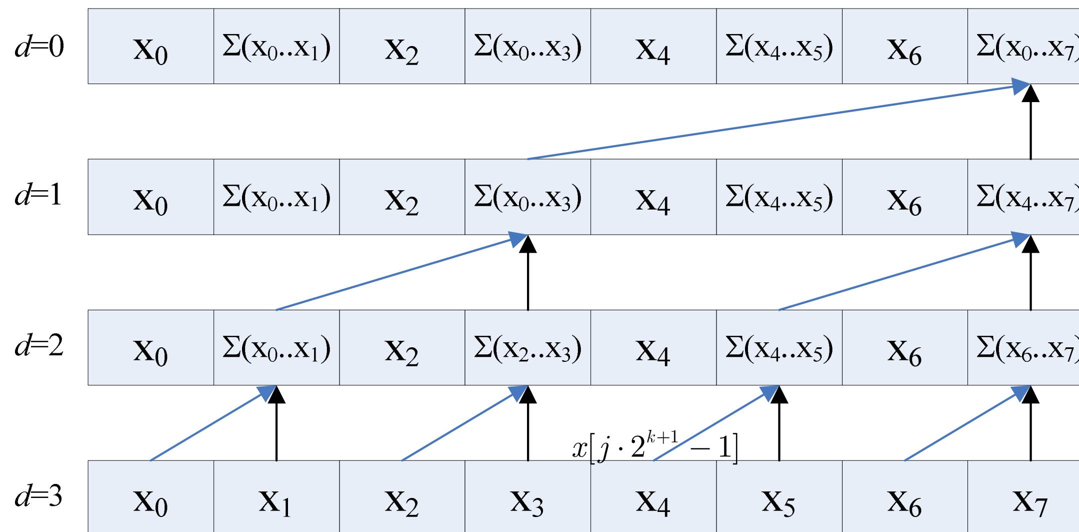
  Balanced Trees

  - 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

Picture and Pseudocode ~ Reduction Step~





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

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

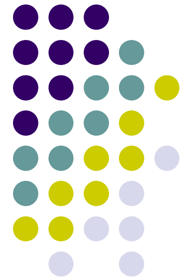
$j \cdot 2^{k+1}-2^k-1$ =

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

Operation Count, Reduce Phase

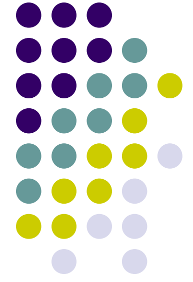


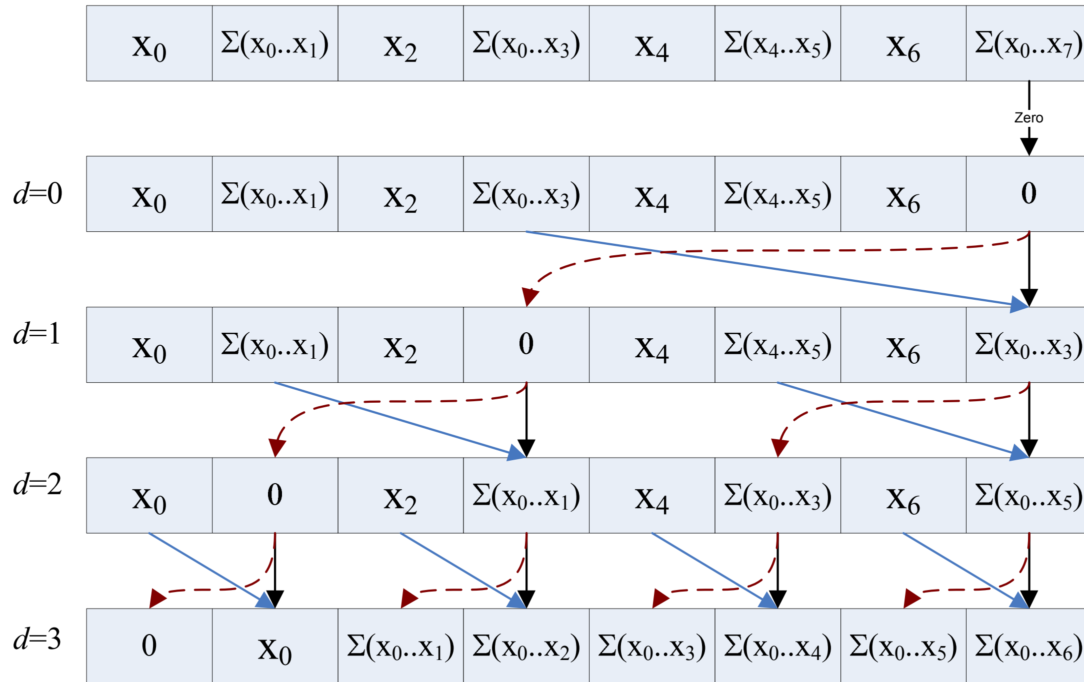
```
for k=0 to M-1
      offset = 2^k
      for j=1 to 2^(M-k-1) in parallel do
            x[j·2^(k+1)-1] = x[j·2^(k+1)-1] + x[j·2^(k+1)-2^k-1]
      endfor
endfor
```

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

Looks promising…

The Down-Sweep Phase

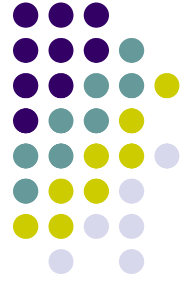




NOTE: This is just a mirror image of the reduction stage. Easy to come up with the indexing scheme…

```
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
```

Down-Sweep Phase, Remarks

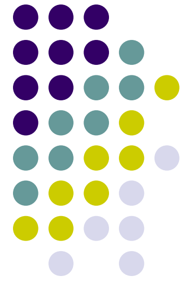


- Number of operations for the down-sweep phase:
  - Additions: n-1
  - Swaps: n-1 (each swap shadows an addition)

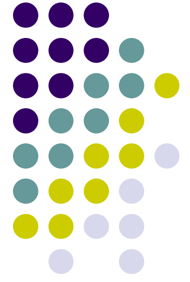
- Total number of operations associated with this algorithm
  - Additions: 2n-2
  - Swaps: n-1
  - Looks very comparable with the work load in the sequential solution

- 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  __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| }
```



Bank Conflicts

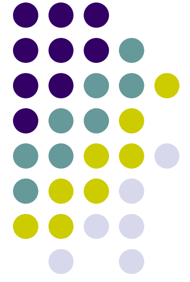


- No penalty if all threads access different banks
  - Or if all threads read from the exact same address (broadcasting)

- 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



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

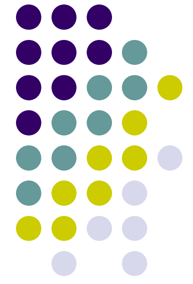
```
temp[2*thid]   = g_idata[2*thid];
temp[2*thid+1] = g_idata[2*thid+1];
```

  - 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

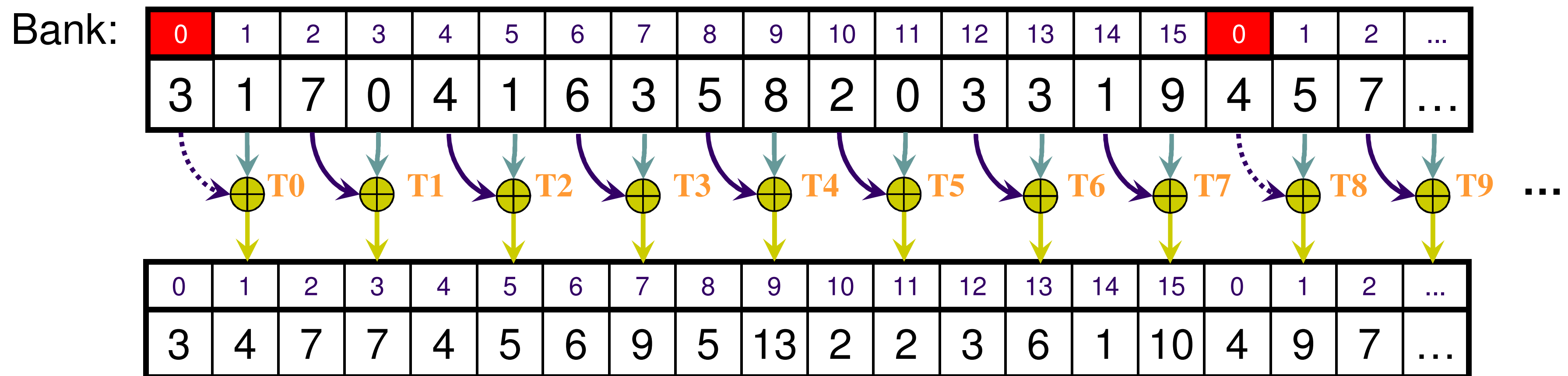
```
temp[thid]         = g_idata[thid];
temp[thid + (n/2)] = g_idata[thid + (n/2)];
```

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

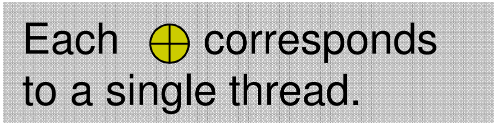
Bank Conflicts in the Tree Algorithm



- 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 conflicats: Threads (0 & 8) access bank 0 at the same time, and then bank 1 at the same time

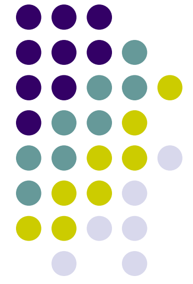


First iteration: 2 threads access each of 8 banks.

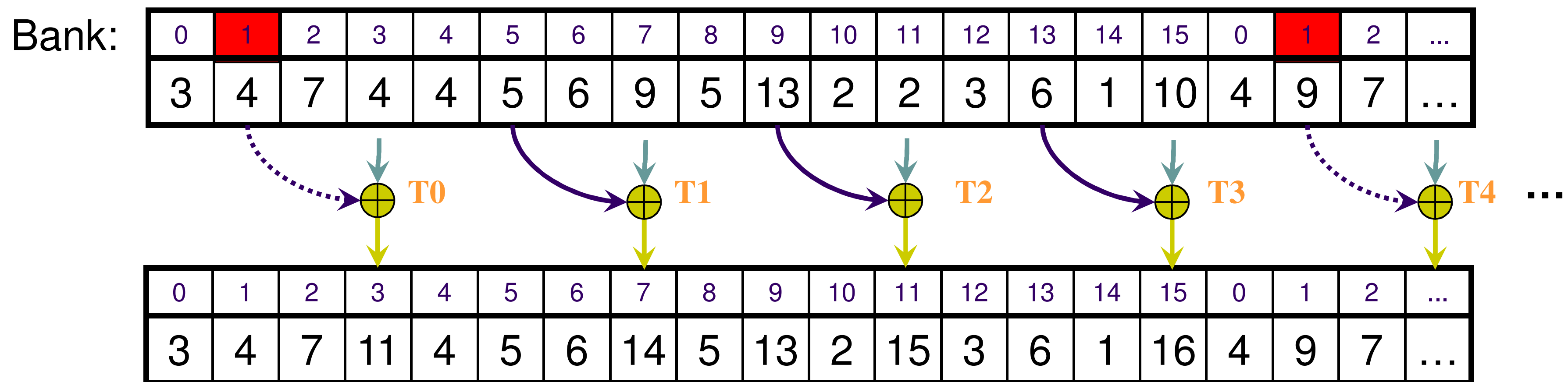
Each  corresponds to a single thread.

Like-colored arrows represent simultaneous memory accesses

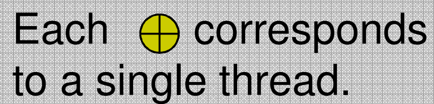
Bank Conflicts in the tree algorithm



- 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.

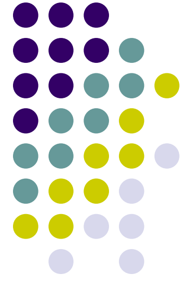


2nd iteration: 4 threads access each of 4 banks.

Each  corresponds to a single thread.

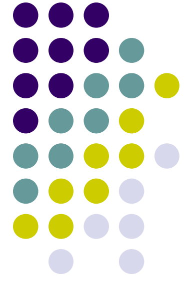
Like-colored arrows represent simultaneous memory accesses

Managing Bank Conflicts in the Tree Algorithm



- 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



- After you compute a ShMem address like this:

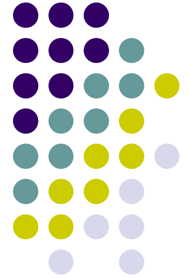
```
address = 2 * stride * thid;
```

- Add padding like this:

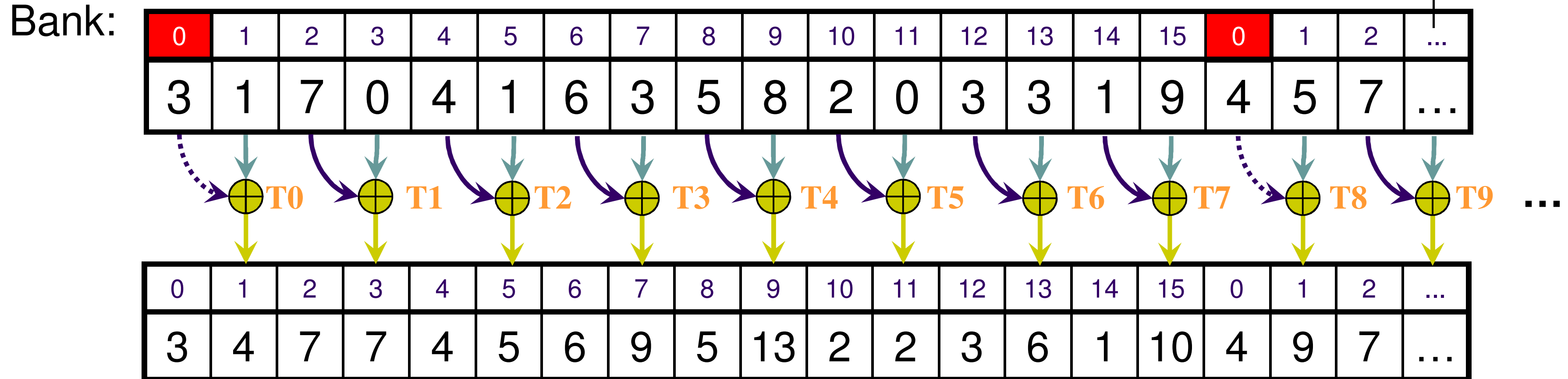
```
address += (address >> 4); // divide by NUM_BANKS
```

- 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

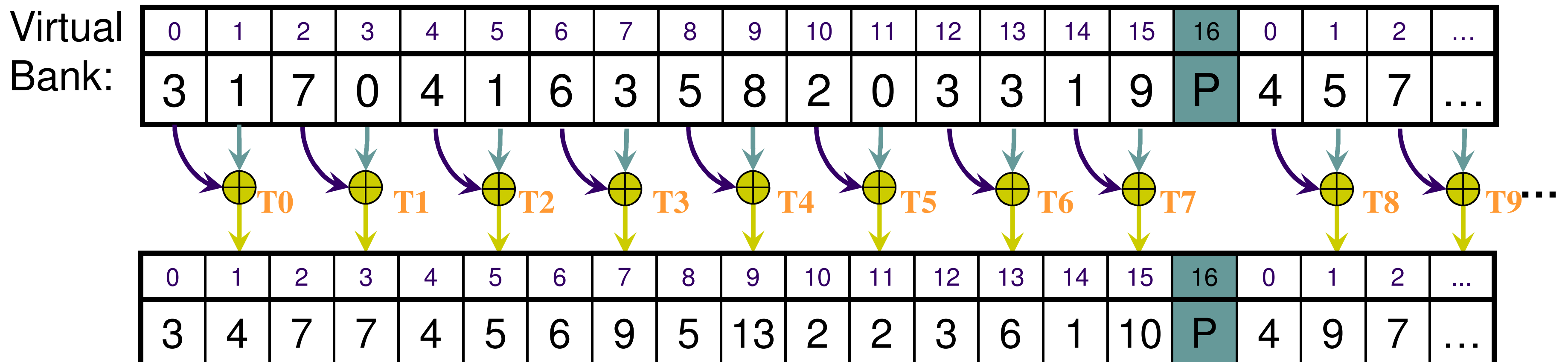


Original scenario.



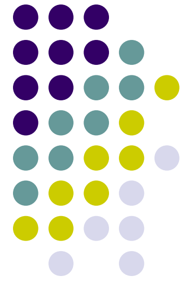
Modified scenario, virtual 17 bank memory layout.

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



Note that only arrows with the same color happen simultaneously.

Concluding Remarks, Parallel Scan

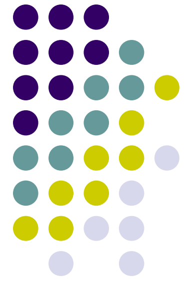


- Intuitively, the scan operation is not the type of procedure ideally suited for parallel computing
  - Even if it doesn't fit like a glove, leads to nice speedup:

# 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 scales like O(n)
  - 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