

ECE 498AL

Lecture 12: Application Lessons

When the tires hit the road...

Objective

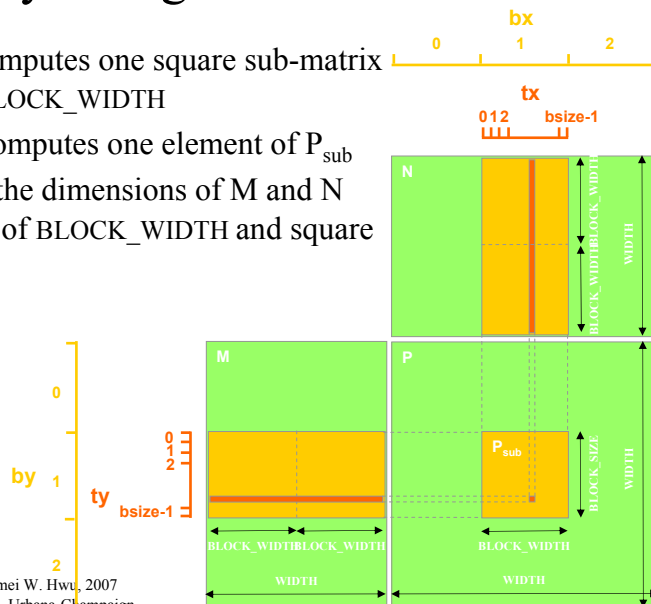
- Putting the CUDA performance knowledge to work
 - Plausible strategies may or may not lead to performance enhancement
 - Different constraints dominate in different application situations
 - Case studies help to establish intuition, idioms and ideas
- Algorithm patterns that can result in both better efficiency as well as better HW utilization

This lecture covers simple case studies on useful strategies for tuning CUDA application performance on G80.

Some Performance Lessons from Matrix Multiplication

Multiply Using Several Blocks

- One **block** computes one square sub-matrix P_{sub} of size `BLOCK_WIDTH`
- One **thread** computes one element of P_{sub}
- Assume that the dimensions of M and N are multiples of `BLOCK_WIDTH` and square shape



First-order Size Considerations

- Each thread block should have a minimal of 96 (768/8) threads
 - TILE_WIDTH of 16 gives $16*16 = 256$ threads
- A minimal of 64 thread blocks
 - A 1024*1024 P Matrix at TILE_WIDTH 16 gives $64*64 = 4096$ Thread Blocks
- Each thread block perform $2*256 = 512$ float loads from device memory for $256 * (2*16) = 8,192$ mul/add operations.

Shared Memory Usage

- Each SMP has 16KB shared memory
 - Each Thread Block uses $2*256*4B = 2KB$ of shared memory.
 - Can potentially have up to 8 Thread Blocks actively executing
 - For BLOCK_WIDTH = 16, this allows up to $8*512 = 4,096$ pending loads
 - In practice, there will probably be up to half of this due to scheduling to make use of SPs.
 - The next BLOCK_WIDTH 32 would lead to $2*32*32*4B = 8KB$ shared memory usage per Thread Block, allowing only up to two Thread Blocks active at the same time

Instruction Mix Considerations

```
for (int k = 0; k < BLOCK_SIZE; ++k)
    Pvalue += Ms[ty][k] * Ns[k][tx];
```

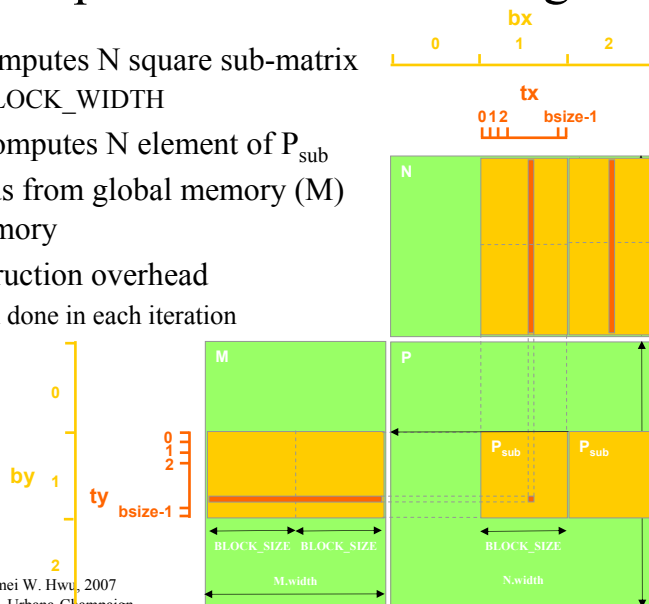
There are very few mul/add between branches and address calculation.

Loop unrolling can help.

```
Pvalue += Ms[ty][k] * Ns[k][tx] + ...
        Ms[ty][k+15] * Ns[k+15][tx];
```

More Work per Thread: 1xN Tiling

- One **block** computes N square sub-matrix P_{sub} of size BLOCK_WIDTH
- One **thread** computes N element of P_{sub}
- Reduced loads from global memory (M) to shared memory
- Reduced instruction overhead
 - More work done in each iteration



Prefetching

- One could double buffer the computation, getting better instruction mix within each thread
 - This is classic software pipelining in ILP compilers

```
Loop {  
  
Load current tile to shared  
memory  
  
syncthread()  
  
Compute current tile  
  
syncthread()  
}
```

©David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

```
Load next tile from global memory  
  
Loop {  
Deposit current tile to shared memory  
syncthread()  
  
Load next tile from global memory  
  
Compute current subblock  
  
syncthread()  
}
```

9

Some More Plausible Ideas

- One might be able to use texture memory for M accesses to reduce register usage
- Let us know if you get more than 120 GFLOPs (including CPU/GPU data transfers) for matrix multiplication

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

10

Scan – Algorithm Effects on Parallelism and Memory Conflicts

Parallel Prefix Sum (Scan)

- Definition:
The all-prefix-sums operation takes a binary associative operator \oplus with identity I , and an array of n elements

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

and returns the ordered set

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

- Example:
if \oplus 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

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:

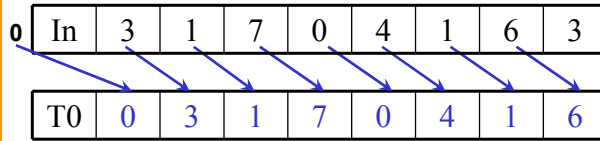
```
forall(j) { temp[j] = f(j) };  
scan(out, temp);
```
- Useful for many parallel algorithms:
 - radix sort
 - quicksort
 - String comparison
 - Lexical analysis
 - Stream compaction
 - Polynomial evaluation
 - Solving recurrences
 - Tree operations
 - Histograms
 - Etc.

Scan on the CPU

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

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

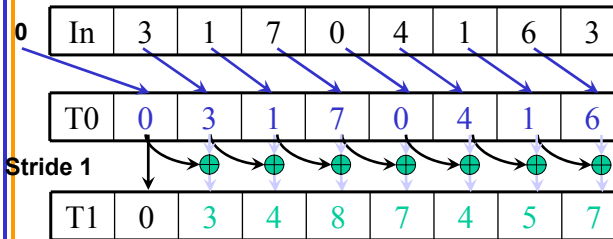
A First-Attempt Parallel Scan Algorithm



1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.

Each thread reads one value from the input array in device memory into shared memory array T0. Thread 0 writes 0 into shared memory array.

A First-Attempt Parallel Scan Algorithm

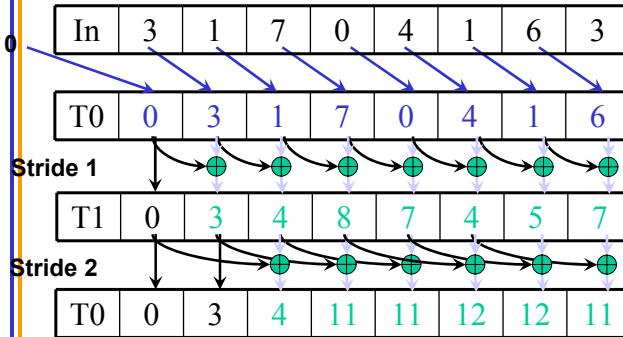


1. (previous slide)
2. Iterate $\log(n)$ times: Threads *stride* to n : Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared mem arrays)

Iteration #1
Stride = 1

- Active threads: *stride* to $n-1$ (n -*stride* threads)
- Thread j adds elements j and j -*stride* from T0 and writes result into shared memory buffer T1 (ping-pong)

A First-Attempt Parallel Scan Algorithm



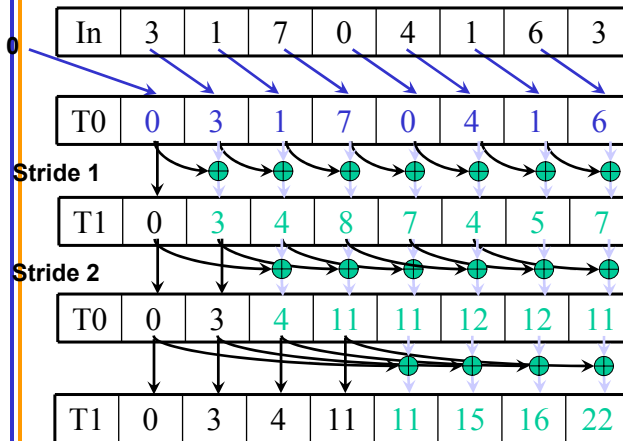
Iteration #2
Stride = 2

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.
2. Iterate $\log(n)$ times: Threads $stride$ to n : Add pairs of elements $stride$ elements apart. Double $stride$ at each iteration. (note must double buffer shared mem arrays)

17

A First-Attempt Parallel Scan Algorithm



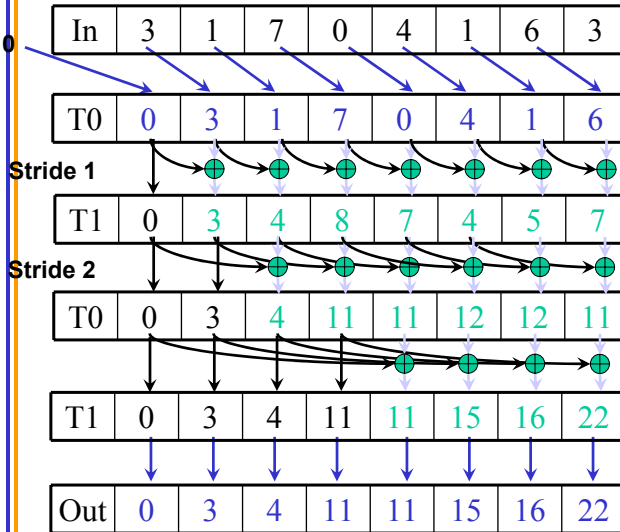
Iteration #3
Stride = 4

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.
2. Iterate $\log(n)$ times: Threads $stride$ to n : Add pairs of elements $stride$ elements apart. Double $stride$ at each iteration. (note must double buffer shared mem arrays)

18

A First-Attempt Parallel Scan Algorithm



1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.
2. Iterate $\log(n)$ times: Threads $stride$ to n : Add pairs of elements $stride$ elements apart. Double $stride$ at each iteration. (note must double buffer shared mem arrays)
3. Write output to device memory.

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

19

Code from CUDA SDK (naïve)

```

__global__ void scan_naive(float *g_odata, float *g_idata, int n)
{
    // Dynamically allocated shared memory for scan kernels
    extern __shared__ float temp[];
    int thid = threadIdx.x;
    int pout = 0;
    int pin = 1;
    // Cache the computational window in shared memory
    temp[pout*n + thid] = (thid > 0) ? g_idata[thid-1] : 0;
    for (int offset = 1; offset < n; offset *= 2) {
        pout = 1 - pout;
        pin = 1 - pout;
        __syncthreads();
        temp[pout*n+thid] = temp[pin*n+thid];
        if (thid >= offset)
            temp[pout*n+thid] += temp[pin*n+thid - offset];
    }
    __syncthreads();
    g_odata[thid] = temp[pout*n+thid];
}

```

Number of threads = n

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

20

Work Efficiency Considerations

- The first-attempt Scan executes $\log(n)$ parallel iterations
 - The steps do $(n/2 + n/2-1)$, $(n/4 + n/2-1)$, $(n/8 + n/2-1)$, ..., $(1 + n/2-1)$ adds each
 - Total adds: $n * (\log(n) - 1) + 1 \rightarrow O(n * \log(n))$ work
- This scan algorithm is not very work efficient
 - Sequential scan algorithm does n adds
 - A factor of $\log(n)$ hurts: 20x for 10^6 elements!
- A parallel algorithm can be slow when execution resources are saturated due to low work efficiency

Improving Efficiency

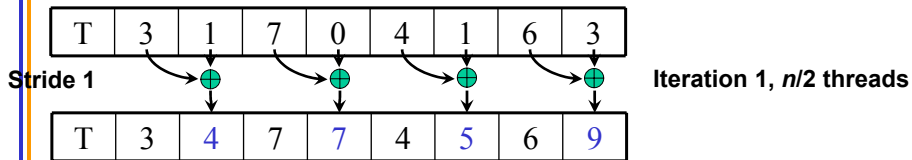
- A common parallel algorithm pattern:
 - Balanced Trees*
 - Build a balanced binary tree on the input data and sweep it to and from the root
 - Tree is not an actual data structure, but a concept to determine what each thread does at each step
- For scan:
 - Traverse down from leaves to root building partial sums at internal nodes in the tree
 - Root holds sum of all leaves
 - Traverse back up the tree building the scan from the partial sums


Build the Sum Tree

T	3	1	7	0	4	1	6	3
---	---	---	---	---	---	---	---	---

Assume array is already in shared memory

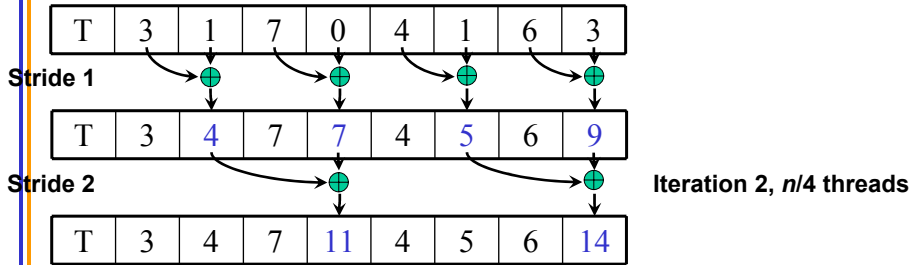
Build the Sum Tree



Each  corresponds to a single thread.

Iterate $\log(n)$ times. Each thread adds value *stride* elements away to its own value

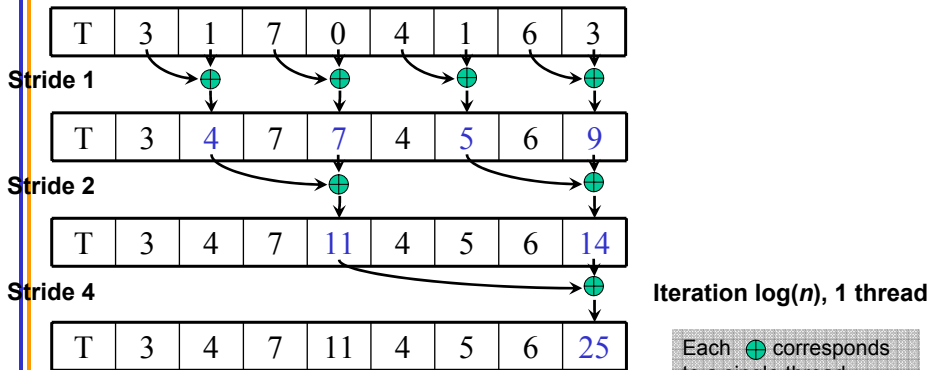
Build the Sum Tree



Each corresponds to a single thread.

Iterate $\log(n)$ times. Each thread adds value *stride* elements away to its own value

Build the Sum Tree



Each corresponds to a single thread.

Iterate $\log(n)$ times. Each thread adds value *stride* elements away to its own value.

Note that this algorithm operates in-place: no need for double buffering

Zero the Last Element

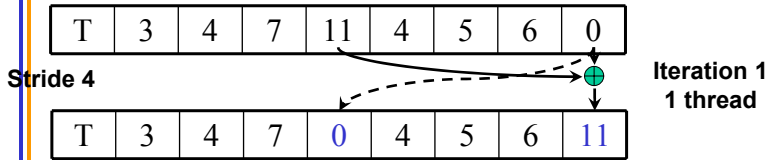
T	3	4	7	11	4	5	6	0
---	---	---	---	----	---	---	---	---


We now have an array of partial sums. Since this is an exclusive scan, set the last element to zero. It will propagate back to the first element.

Build Scan From Partial Sums

T	3	4	7	11	4	5	6	0
---	---	---	---	----	---	---	---	---

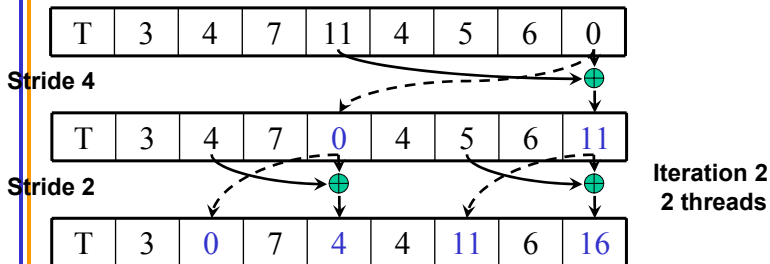
Build Scan From Partial Sums




Each  corresponds to a single thread.

Iterate $\log(n)$ times. Each thread adds value *stride* elements away to its own value, and sets the value *stride* elements away to its own *previous* value.

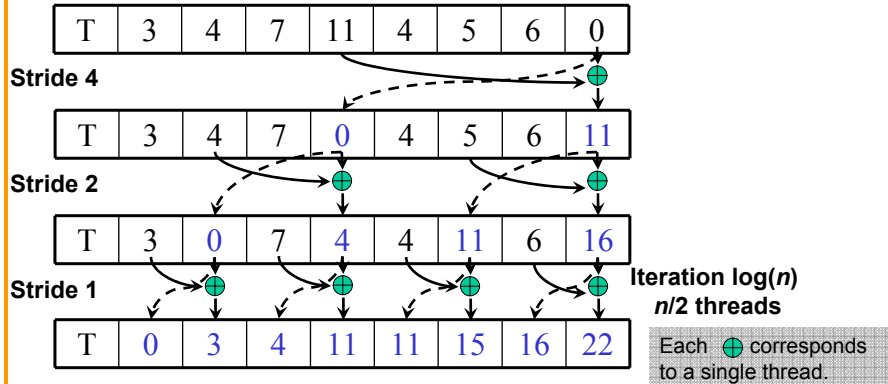
Build Scan From Partial Sums



Each  corresponds to a single thread.

Iterate $\log(n)$ times. Each thread adds value *stride* elements away to its own value, and sets the value *stride* elements away to its own *previous* value.

Build Scan From Partial Sums



Done! We now have a completed scan that we can write out to device memory.

Total steps: $2 * \log(n)$.

Total work: $2 * (n-1)$ adds = $O(n)$ **Work Efficient!**

Code (CUDA SDK work-efficient)

```

_global__ void scan_workefficient(float *g_odata, float *g_idata, int n){
    // Dynamically allocated shared memory for scan kernels
    extern __shared__ float temp[];
    int thid = threadIdx.x;
    int offset = 1;
    // Cache the computational window in shared memory
    temp[2*thid] = g_idata[2*thid];
    temp[2*thid+1] = g_idata[2*thid+1];
    // build the sum in place up the tree
    for (int d = n>>1; d > 0; d >>= 1) {
        __syncthreads();
        if (thid < d) {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;
            temp[bi] += temp[ai];
        }
        offset *= 2;
    }
}
    
```

Number of threads = $n/2$

Code (Cont.)

```
// scan back down the tree
// clear the last element
if (thid == 0) {
    temp[n - 1] = 0;
}
// traverse down the tree building the scan in place
for (int d = 1; d < n; d *= 2) {
    offset >>= 1;
    __syncthreads();

    if (thid < d){
        int ai = offset*(2*thid+1)-1;
        int bi = offset*(2*thid+2)-1;
        float t = temp[ai];
        temp[ai] = temp[bi];
        temp[bi] += t;
    }
}
```

Code (cont.)

```
__syncthreads();
// write results to global memory
g_odata[2*thid] = temp[2*thid];
g_odata[2*thid+1] = temp[2*thid+1];
}
```

Problem: Bank conflict

- Each thread loads two shared mem data elements
- Tempting to interleave the loads

```
temp[2*thid]    = g_idata[2*thid];
temp[2*thid+1] = g_idata[2*thid+1];
```
- Threads:(0,1,2,...,8,9,10,...)→banks:(0,2,4,...,0,2,4, ...)
- Solution: padding
- See the best implementation in CUDA SDK.