GPU TECHNOLOGY CONFERENCE

April 4-7, 2016 | Silicon Valley

GPU MEMORY BOOTCAMP III

COLLABORATIVE ACCESS PATTERNS

Tony Scudiero – NVIDIA Devtech

PRESENTED BY NVIDIA

Fanatical Bandwidth Evangelist

The Axioms of Modern Performance

#1. Parallelism is mandatory

Chips getting wider, not faster for 10+ years

#2. In real machines, as parallelism increases, real applications become limited by

"**<u>the rate at which operands can be supplied to functional units</u>**"

Flop/BW ratios increase

Memory latency increases, single-chip NUMA

Bootcamp 1: Best Practices

http://on-demand.gputechconf.com/gtc/2015/video/S5353.html

Loads in Flight

Coalescing

Shared Memory & Bank Conflicts

Memory Level Parallelism

A stream of C5 Galaxy aircraft filled with microSD cards has amazing bandwidth but latency leaves something to be desired

Bootcamp 2: Beyond Best Practices

http://on-demand.gputechconf.com/gtc/2015/video/S5376.html

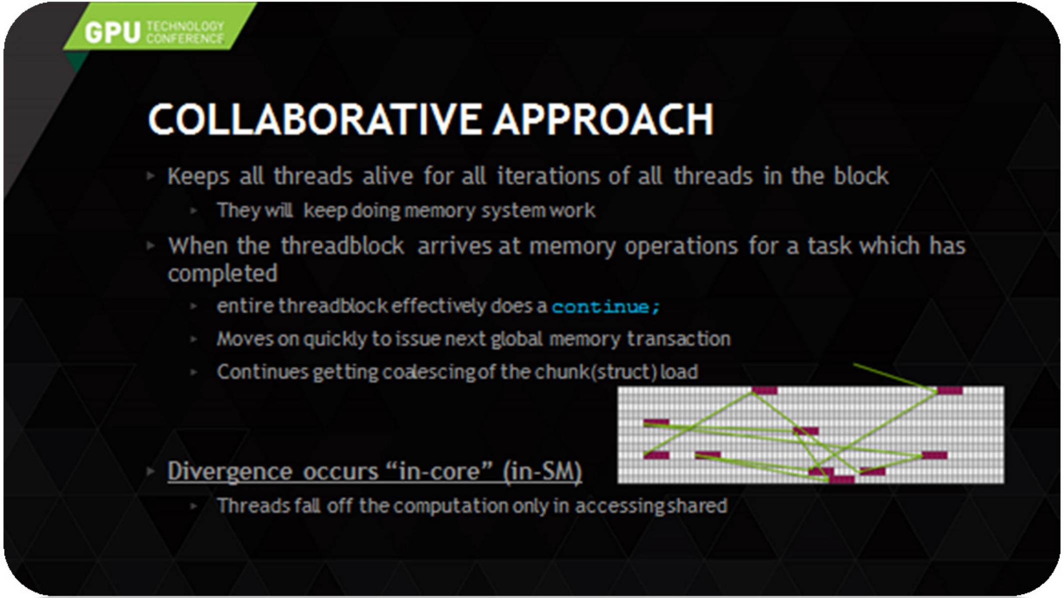
Opt-In L1 Caching – when it helps, when it hurts

Maximize data load size when applicable

Exploit Memory-Level-Parallelism:

- Hoist Loads, in-thread latency hiding
- Maximize loads in flight per thread

Bootcamp2: The Lost Slide



Bootcamp 2: The Lost Kernel

```
template<typename T, int MAX_SUMMANDS, int OPS_PER_PASS>
__global__ void collaborative_access_small_summandVersatile (long n, long m,
                                              long * offsetArray,
                                              T * dataArray,
                                              int * iterationArray,
                                              int summands,
                                              T * results)
{
    __shared__ long iterationOffset[COLABORATIVE_CTA_SIZE_MAX];
    __shared__ T sdata[OPS_PER_PASS][MAX_SUMMANDS+1];
    __shared__ T sResult[COLABORATIVE_CTA_SIZE_MAX];
    __shared__ int sIterCount[COLABORATIVE_CTA_SIZE_MAX];
    __shared__ int maxIt;
```

I Collect Quadro M6000s

All results are from Quadro M6000 (GM200)

AGENDA

~~Review of Bootcamp 1 and Bootcamp 2~~

Defining Collaborative Patterns

Reintroduce Bootcamp Kernel

Two collaborative versions of Bootcamp Kernel

Example: Signal Processing

Defining Collaborative Patterns

What Is Collaborative

Collaborate: To work together with others to achieve a common goal

Non-Exhaustive List of Collaborative Pattern Indicators:

- Data dependency between threads, especially operation results
- Number of outputs is less than number of active threads
- Thread works on data it did not load
- __syncthreads()

Collaborative Patterns: Memory Specific

Fundamental Principle: Minimize repeated or wasted memory traffic

* Achieve Coalescing

* Reuse data loaded by other threads

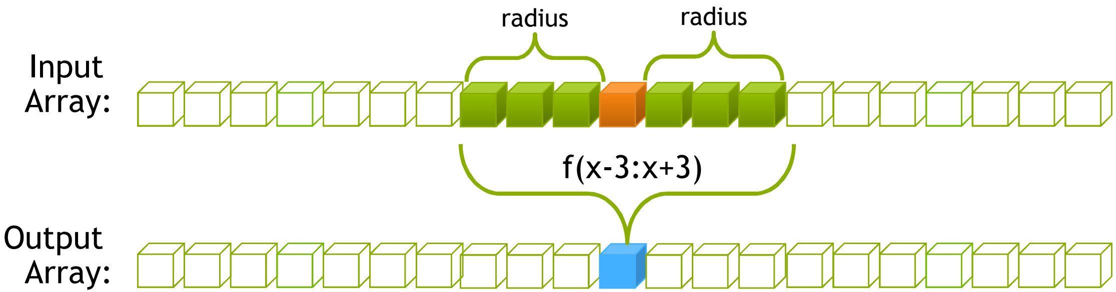
Focus on load/store structure of the algorithm

As opposed to inputs/outputs or compute structure

May result in threads that never issue 'math ops'

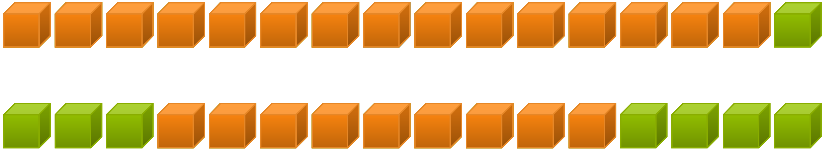
1D Stencil is Collaborative

Reminder: What is 1D Stencil



Reuse in 1D Stencil

Rather . . It should be collaborative



AGENDA

~~Review of Bootcamp 1 and Bootcamp 2~~

~~Defining Collaborative Patterns~~

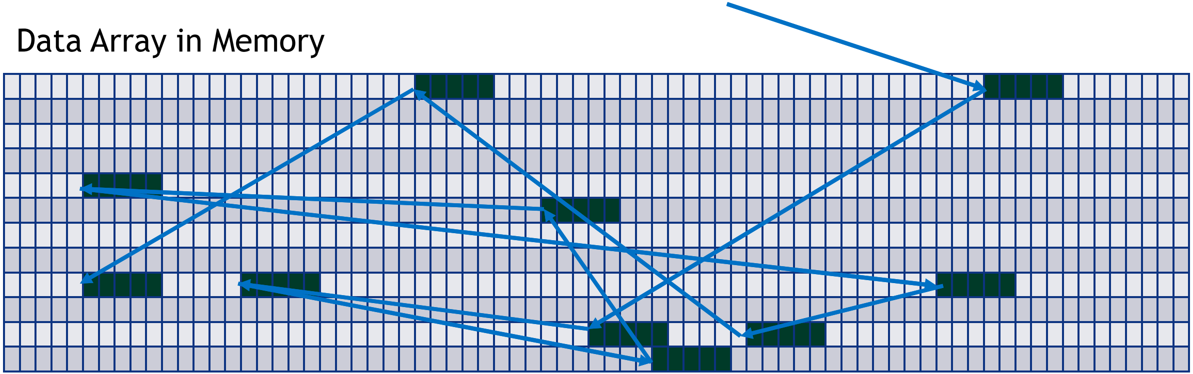
Reintroduce Bootcamp Kernel

Two collaborative versions of Bootcamp Kernel

Example: Signal Processing

THE BOOTCAMP CODE

Data Array in Memory



Access of a Single Task

```
template <typename T>
__global__ void reference_algo (long n,
                                long m,
                                long * offsetArray,
                                T * dataArray,
                                int * iterationArray, // Outer Loop Limit
                                int summands,         // Inner Loop Limit
                                 T * results)
{
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    int iterations =  iterationArray[idx];
    if (idx >= m)
    { return;}

    long offset = offsetArray[idx];
    for(int i=0; i<iterations; i++)
    {
        for(int s=0; s<summands; s++)
        {
            results[idx] += dataArray[offset + s];
        }
        offset = offsetArray[offset];
    }
}
```

Type Abstraction

Parallelism:
M executions

Random offset into dataArray

Outer Loop (iterations)

Inner Loop (Summands)

New dataArray offset each Iteration

Baseline Performance

Kernel	Float perf (M iters/s)	Float Bandwidth (GB/s)	Double perf (M iters/s)	Double Bandwidth (GB/s)
Reference	145.79	18.67	91.71	23.50
Bootcamp2: LoadAs_u2	405.5	51.95	158.33	40.577

AGENDA

~~Review of Bootcamp 1 and Bootcamp 2~~

~~Defining Collaborative Access~~

~~Reintroduce Bootcamp Kernel~~

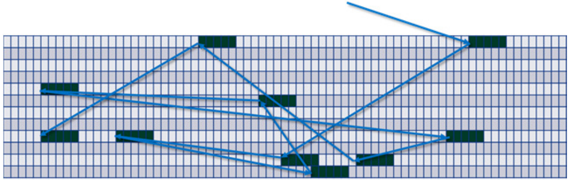
Two collaborative versions of Bootcamp Kernel

Example: Signal Processing

Collaborative Approach

Each task's starting location is random, but . . .

- Subsequent loads (after the first) are not random
  - Chunk of seqential/stride-1 memory
  - Simulates reading a multi-word data structure in AoS fashion



Could we load that sequential data (struct) in a coalesced fashion?

Assume we cannot parallelize across iterations

Collaborative Approaches

1. Each thread block performs one task
   - Vector-style pattern

2. Each thread does one task, thread block collaborates on load
   - Collaborative memory pattern only
   - Compute structure remains similar

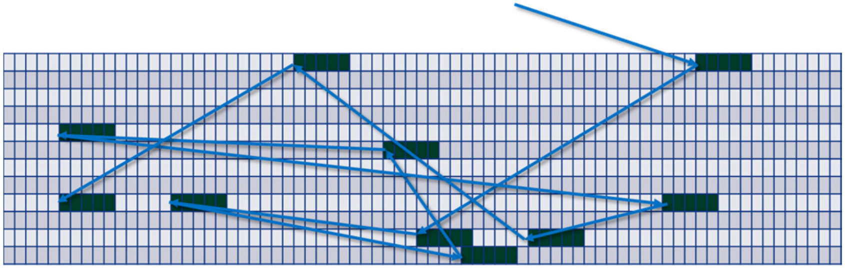
Collaborative Approach #1

Vector-Style Collaboration

"Restructure for Coalescing"

Each **block** performs the work of a single task

Instead of each **thread**



GPU TECHNOLOGY CONFERENCE

Collaborative Approach #1

Vector-Style kernel

```
__shared__ float sBuffer[MAX_SUMMANDS];

int offset = offsetArray[blockIdx.x];
for (int i=0; i<iterations; i++)
{
    float r = 0.0f;
    sBuffer[threadIdx.x] = dataArray[offset + threadIdx.x];
    __syncthreads();
    if(threadIdx.x == 0)
    {
        for(int s=0; s<summands; s++)
        {   r += sBuffer[s];    }
        results[blockIdx.x] += r;
    }
    offset = offsetArray[offset];
}
```

Assumes (summands == blockDim.x), Condition checks omitted from slide

Single Thread performs compute!

blockIdx.x == task id

Performance Report

Kernel	Float perf (M iters/s)	Float Bandwidth (GB/s)	Double perf (M iters/s)	Double Bandwidth (GB/s)
Reference	145.79	18.67	91.71	23.50
Bootcamp2: LoadAs_u2	405.5	51.95	158.33	40.577
Vector-Style	364.3	46.7	90.1	23.1

Sneaky Stuff

The code on the previous slide works as expected on

- Single precision on all hardware
- Double precision on full-throughput double precision hardware
  - Which GM200 is not

Single thread reduction is compute limiting due to double precision latency

- Using a parallel reduction returns to our expectations

Performance Report

Kernel	Float perf (M iters/s)	Float Bandwidth (GB/s)	Double perf (M iters/s)	Double Bandwidth (GB/s)
Reference	145.79	18.67	91.71	23.50
Bootcamp2: LoadAs_u2	405.5	51.95	158.33	40.577
Vector-Style	364.3	46.7	90.1	23.1
Vector-Style w/Reduce	338.1	43.32	266.84	68.37

A Second Version

A Hybrid Kernel

- Single thread per task (like original kernel)
- Whole block performs load (like vector kernel)

Collaborative memory structure

- Threads join up and collaborate for loads, split up for compute
- Can diverge in compute, must be kept alive

Collaborative Access Kernel

In Pseudocode

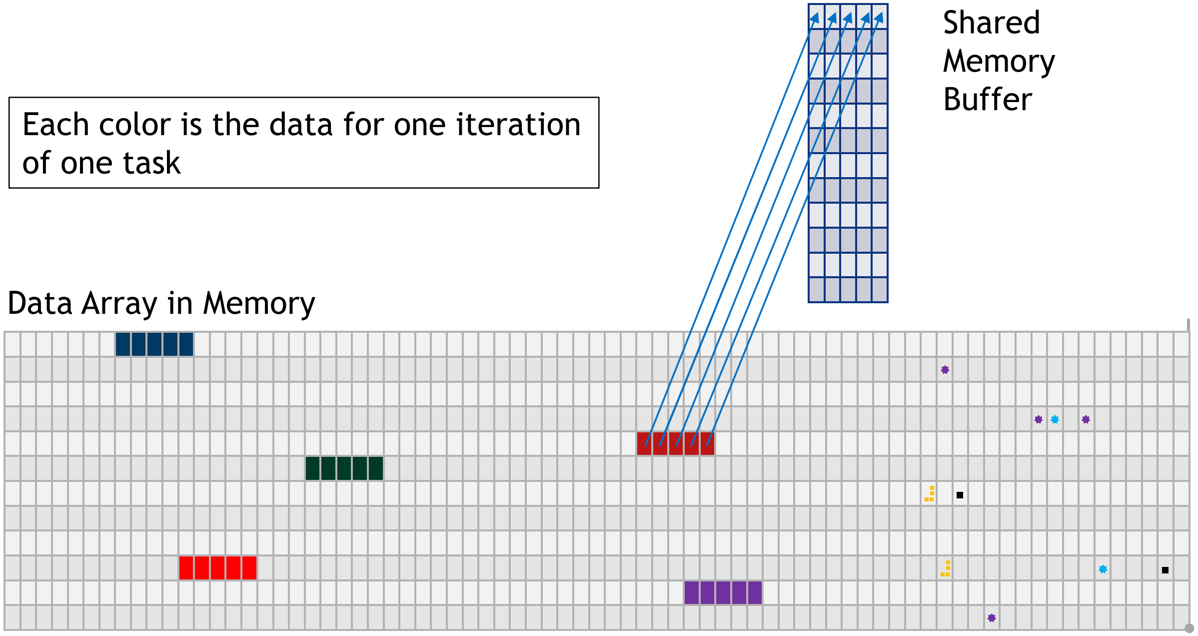
```
for i=1:iterations
  for each thread in the block
    coalesced load of data for this iteration into shared buffer
  end
  __syncthreads();
  for s=1:summands
     r += shared buffer[threadIdx.x][s];
  end
  update offsets for next iteration
 __syncthreads();
end for
```

Load Step

Compute Step

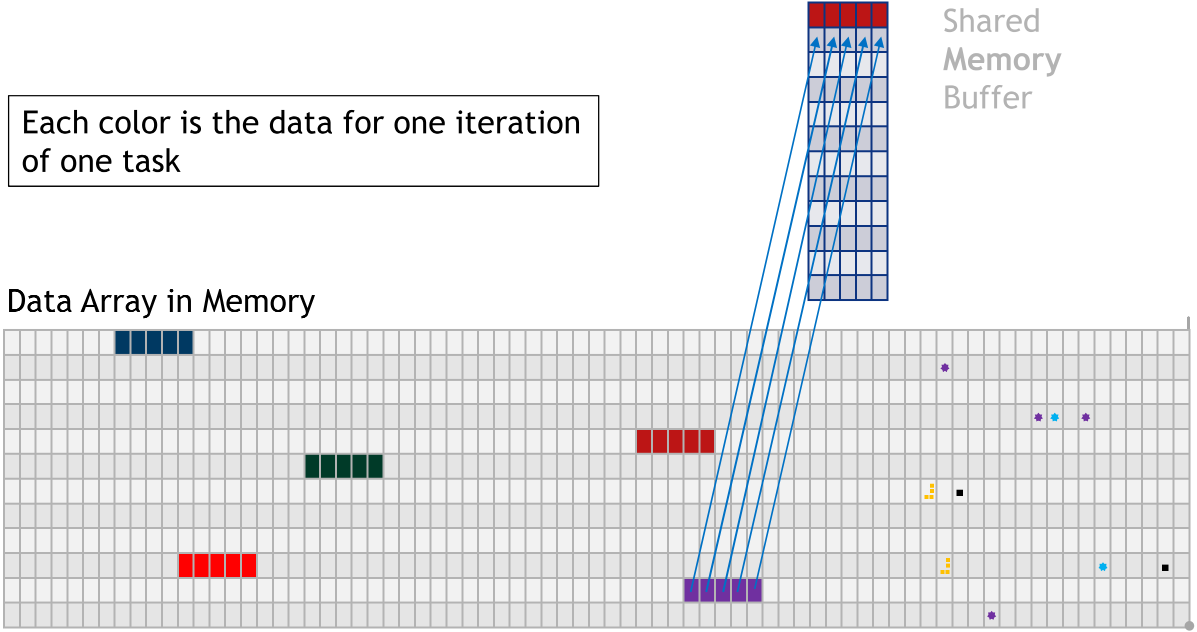
COLLABORATIVE LOAD

Each color is the data for one iteration of one task



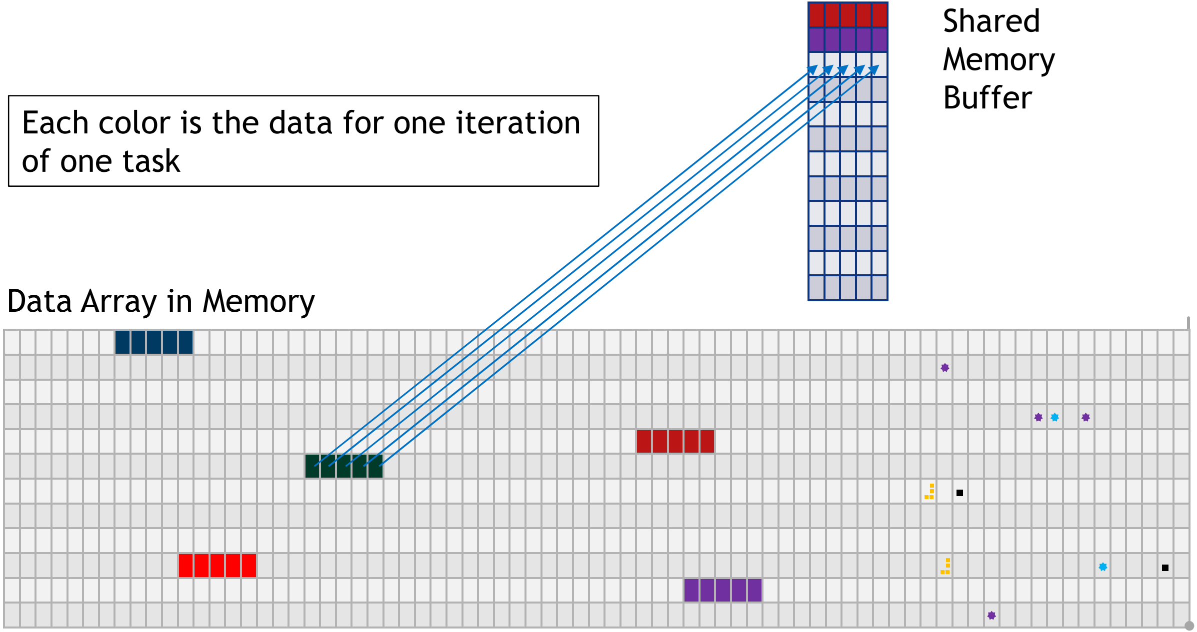
COLLABORATIVE LOAD

Each color is the data for one iteration of one task

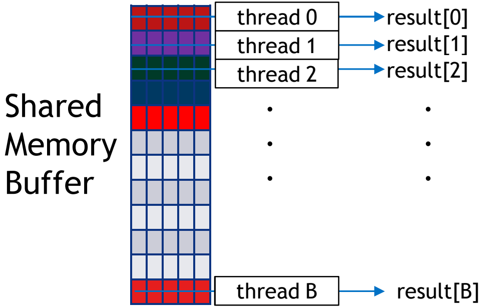


COLLABORATIVE LOAD

Each color is the data for one iteration of one task



Compute Step



B = blockDim.x

After the load step, each thread processes its data from shared memory

Collaborative Access Kernel

Assumes blockDim.x == summands, No bounds Checks on slide

```
__shared__ long iterationOffset[THREADS_PER_BLOCK];
__shared__ float sdata[THREADS_PER_BLOCK][MAX_SUMMANDS+1];
long idx = blockDim.x*blockIdx.x + threadIdx.x;
long offset = offsetArray[idx];

for(int i=0; i<iterations; i++)
{   iterationOffset[threadIdx.x] = offset;
    for(int s=0; s<THREADS_PER_BLOCK; s++)
    { sdata[s][threadIdx.x] = dataArray[iterationOffset[s] + threadIdx.x];}
    __syncthreads();

    float r = 0.0;
    for(int s=0; s<summands; s++)
        { r+= sdata[threadIdx.x][s]; }
    result[idx] += r;
    offset = offsetArray[offset];
    __syncthreads();
}
```

Load Step

Compute Step

What It really looks like

Even at 32 threads per block (1 warp), **`sdata`** is 4kb(float) or 8kb(double).

Max Occupancy 15.6%(f) 7.8%(d)

Solution: Multiple passes per thread block

Shared buffer is 32xK. for K=8, **`sdata`** is 1024b or 2048b per block

All threads load, threads 0-7 perform computation

Only 25% of threads are doing ‘computations’

K=8 and K=16 are best performers on GM200

GPU TECHNOLOGY CONFERENCE

Performance Report

Kernel	Float perf (M iters/s)	Float Bandwidth (GB/s)	Double perf (M iters/s)	Double Bandwidth (GB/s)
Reference	145.79	18.67	91.71	23.50
Bootcamp2: LoadAs_u2	405.5	51.95	158.33	40.577
Vector-Style	364.3	46.7	90.1	23.1
Vector-Style w/Reduce	338.1	43.32	266.84	68.37
Collaborative Load	736.0	94.3	452.76	116.02

Discussion Slide

32 threads per thread block, single precision

Kernel	Threads performing addition	Tasks per block	Performance GM200 (M iters/s)	Bandwidth GM200 (GB/s)
Reference	blockDim.x	blockDim.x	145.79	18.67
LoadAs	blockDim.x	blockDim.x	405.5	51.95
Vector-Style	1	1	364.3	46.7
Collaborative Access	$\frac{blockDim.x}{K}$	blockDim.x	736.0	94.3

Comparing the Kernels

Why is the collaborative access better?

<u>Loads In Flight</u>
(threadblock)

"But wait, there's more"

GM200 is Maxwell …

… which doesn't have full throughput double precision

Shouldn't double precision be 1/32nd the speed of float?

Kernel	Performance		Bandwidth	
	M Iters/s	Vs. Float	GB/sec	Vs. float
Vector w/Reduce (float)	338.1	-	43.32	-
Vector w/Reduce (double)	266.84	**78.9%**	68.37	**157.8%**
Collaborative(float)	738.131	-	94.57	-
Collaborative(double)	452.9	**61.4%**	116.06	**122.7%**

One More Thing

Kernel	Float perf (M iters/s)	Float BW (GB/s)	M iters/GB
Reference	145.79	18.67	7.809
Bootcamp2: LoadAs_u2	405.5	51.95	7.805
Vector-Style	364.3	46.7	7.800
Collaborative	338.1	43.32	7.805

Kernel	Doubleperf (M iters/s)	Double BW(GB/s)	M iters/GB
Reference	91.71	23.50	3.907
Bootcamp2: LoadAs_u2	158.33	40.577	3.902
Vector-Style w/Reduce	266.84	68.37	3.903
Collaborative	452.76	116.02	3.902

AGENDA

~~Review of Bootcamp 1 and Bootcamp 2~~

~~Defining Collaborative Patterns~~

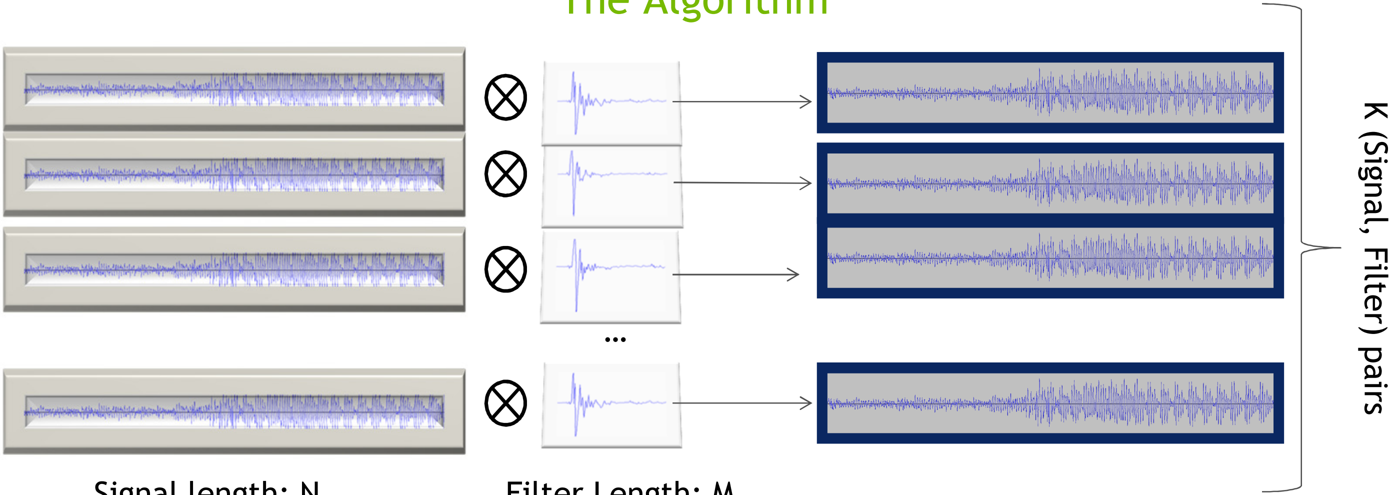
~~Reintroduce Bootcamp Kernel~~

~~Two collaborative versions of Bootcamp Kernel~~

Example: Signal Processing

Signal Processing: Multi-Convolution

The Algorithm



Signal length: N
Arbitrarily long

Filter Length: M
Very Small

Collaborative Implementation

One block per convolution, i.e., (filter, signal) pair

Each convolution is executed vector-like within the block

Filter is collaboratively loaded into shared memory (cached)

Signal is buffered in shared (cached)

Strong resemblance to 1D Stencil!

Signal Processing: Bank Convolution

Slide-Simplified kernel (no bounds checking)

```
__shared__ float filterBuffer[filterLength];
__shared__ float signalBuffer[2*BLOCK_SIZE];

int index = blockIdx.x;
filterBuffers[threadIdx.x] = filters[index][filterLength-threadIdx.x];}
signalBuffer[blockDim.x + threadIdx.x] = 0.0f;

for(int i=0; i<signalLength; i+=blockDim.x)
{
    __syncthreads();
    signalBuffer[threadIdx.x] = signalBuffer[blockDim.x + threadIdx.x];
    signalBuffer[blockDim.x + threadIdx.x] = signal[i*blockDim.x + threadIdx.x];
    __syncthreads();
    float r = 0.0f;
    for(int t=0; t<filterLength; t++)
    {   r+= filterBuffers[t]*signalBuffer[t+threadIdx.x];  }
    results[index][i*blockDim.x + threadIdx.x] = r;
}
```

Collaborative Loads are coalesced

Coalesced write

Operands & results all in shared mem and registers

~~AGENDA~~

~~Review of Bootcamp 1 and Bootcamp 2~~

~~Defining Collaborative Patterns~~

~~Reintroduce Bootcamp Kernel~~

~~Two collaborative versions of Bootcamp Kernel~~

~~Example: Signal Processing~~

Specific Results

Kernel Investigated

- At 1/32nd throughput, one thread/block doing compute is a limiter in this code
- At 1/32nd throughput, parallel compute is not a limiter in this code
- At full throughput, 1 thread/block doing compute is not a limiter in this code
- Collaborative pattern worked well for bootcamp kerenel

General Observations

Collaborative patterns seem to use __shared__ memory

Yes – in-block communication - collaboration requires communication

Collaborative patterns are good when there is data reuse

Yes – data reuse is a natural candidate for collaborative load structure

If code is *limited by the rate at which operands are delivered to functional units*:

a). Optimization efforts should focus on operand delivery

b). In collaborative kernels, some threads may not do any ‘compute’

TROVE

https://github.com/bryancatanzaro/trove

A GPU Library from Bryan Catanzaro (of Deep Learning fame)

- Library to allow access to AoS data structures
- Gather large structures from random locations in GPU memory

Is This Code on github?

By far, the most frequently asked bootcamp question

The wait is over. . .

https://github.com/tscudiero/MemBootcamp

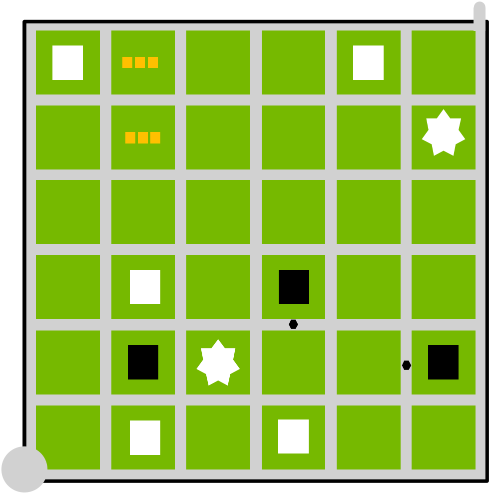
GPU TECHNOLOGY CONFERENCE

April 4-7, 2016 | Silicon Valley

THANK YOU

JOIN THE CONVERSATION

#GTC16



PRESENTED BY

NVIDIA