Schedule

0:00 -- 0:10 : Introduction to the Tutorial, Theme, Speakers.

0:10 -- 0:30 : Basic Concepts -- CPU Architectures, GPUs -- evolution, comparison to earlier models of parallel computing

0:30 -- 0:55 : GPU Architectures in Detail -- NVidia architecture, Intel Larrabee architectural features

0:55 -- 1:30 : GPU Programming models with short examples, CUDA

B R E A K

1:50 -- 2:15 : Case studies of regular applications on the GPU

2:15 -- 2:45 : Case studies of irregular applications on the GPU

2:45 -- 3:20 : GPU Analytical Models

Design Space Optimization, Performance Prediction

3:20 -- 3:30 : Concluding remarks, discussion

© NVIDIA Corporation 2006

NVIDIA®

<<< GPU Programming >>>

Suryakant Patidar

spatidar@nvidia.com

Don’t just process, Compute !

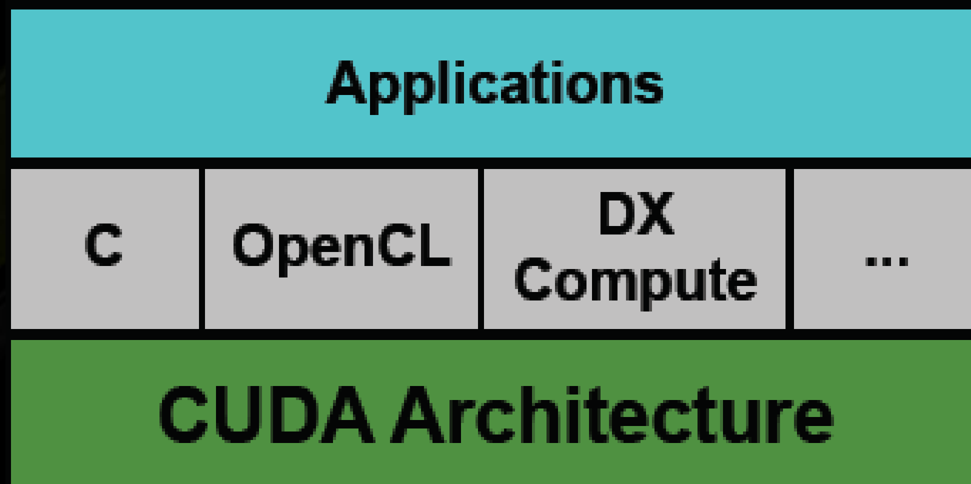
- Graphics Processing on GPU
  - OpenGL, DirectX …
  - Games, Visualizations …
- Classic GPGPU : General Processing on the GPU
  - OpenGL, DirectX etc. ? ?
  - General Problems <faked as> Graphics Rendering Problems
  - Easy – Data Parallel, image processing
  - Hard – Irregular Algorithms, MST for a Sparse Graph
- Compute on GPU
  - NVIDIA CUDA – C for GPUs
  - OpenCL – Open Compute Library

Take Away

- Compute Architecture
  - C, OpenCL
- Hardware/Software Model
  - Processor & Memory Organization
  - Execution Model
- API
  - Language and Limitations

Compute Architecture

- GPU as Highly Multi-Threaded Co-Processor
  - 1000s of threads, not 2, not 4, not 8
  - 100s of tiny processors
- Supports standard languages and APIs
  - C (CUDA)
  - OpenCL
  - DX Compute
- Supported on common operating systems
  - Linux
  - MacOS
  - Windows



CUDA/OpenCL Programming

- Heterogeneous programming model
  - CPU and GPU are separate devices with separate memory spaces
- CPU code is standard C/C++
  - OpenCL : Look alike as OpenGL(C based)
  - CUDA : C/C++
- GPU code
  - OpenCL/CUDA : Subset of C with extensions

CUDA H/W Architecture

SIMD Multi Processor #30

SIMD Multi Processor #2

SIMD Multi Processor #1

Shared Memory (16KB)

Registers

Registers

Registers

P_1

P_2

P_8

Instruction Unit

Texture Cache (8KB)

Constant Cache (8KB)

Device Memory (~1GB)

Software – Terminology

- Host – CPU
- Device – GPU

- Kernel – code which we wish to run on the GPU
- Thread/WorkItem – An Instance of a Kernel
- Block/WorkGroup – Group of Threads, bunched together
- Grid – Group of Blocks, one Grid – one Kernel

- OpenCL is very much inspired by CUDA, and given the GPU hardware is common to both, the APIs and approach are similar too

Kernels and Threads

- Parallel portions of an application are executed on the device as kernels
  - One kernel is executed at a time
  - Many threads execute each kernel
- Differences between CUDA and CPU threads
  - CUDA threads are extremely lightweight
    - Very little creation overhead
    - Fast switching
  - CUDA uses 1000s of threads to achieve efficiency
    - Multi-core CPUs can use only a few

CUDA S/W Architecture

CPU / Host

Kernel 1

Kernel 2

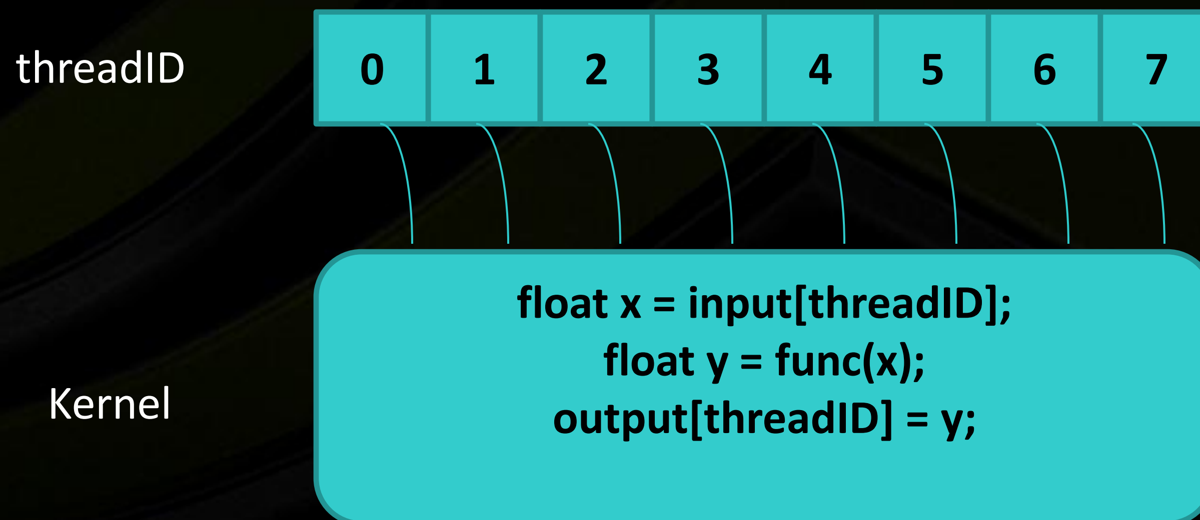
GPU / Device

Blocks (0,0)	Blocks (1,0)	Blocks (2,0)
Blocks (0,1)	Blocks (1,1)	Blocks (2,1)

Thread (0,0)	Thread (1,0)	Thread (2,0)	Thread (3,0)
Thread (0,1)	Thread (1,1)	Thread (2,1)	Thread (3,1)
Thread (0,2)	Thread (1,2)	Thread (2,2)	Thread (3,2)
Thread (0,3)	Thread (1,3)	Thread (2,3)	Thread (3,3)

Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code
  - Each thread has an ID that it uses to compute memory addresses and make control decisions

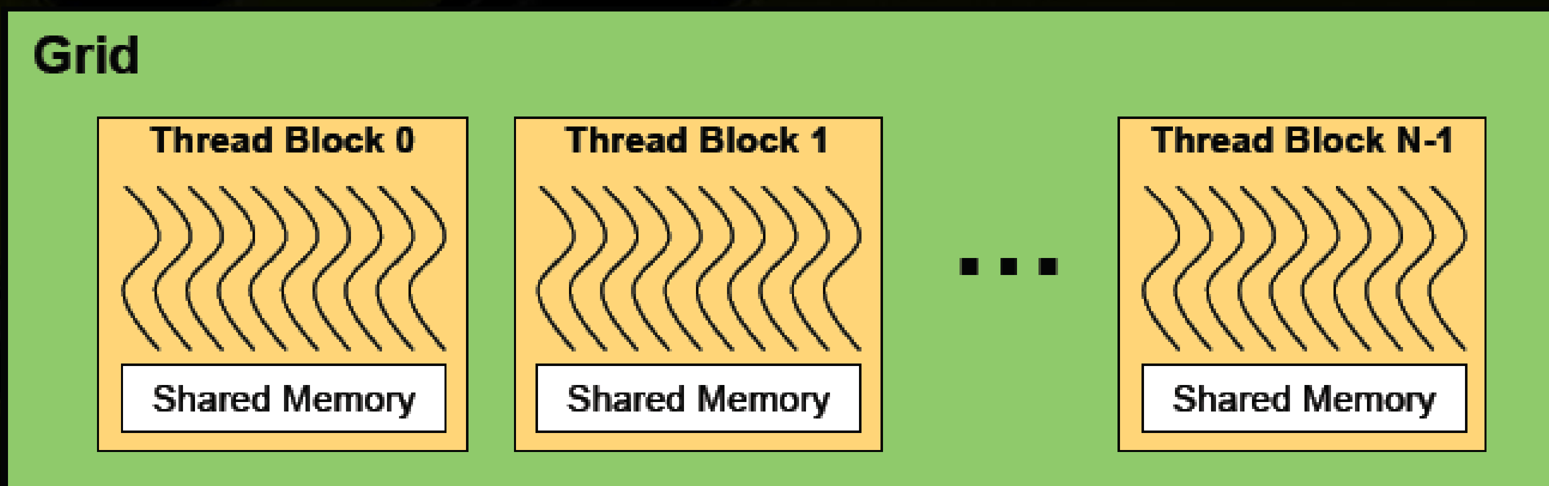


Thread Cooperation

- The Missing Piece: threads may need to cooperate
- Thread cooperation is valuable
  - Share results to avoid redundant computation
  - Share memory accesses
    - Bandwidth reduction
- Cooperation between a monolithic array of threads is not scalable
  - Cooperation within smaller batches of threads is scalable

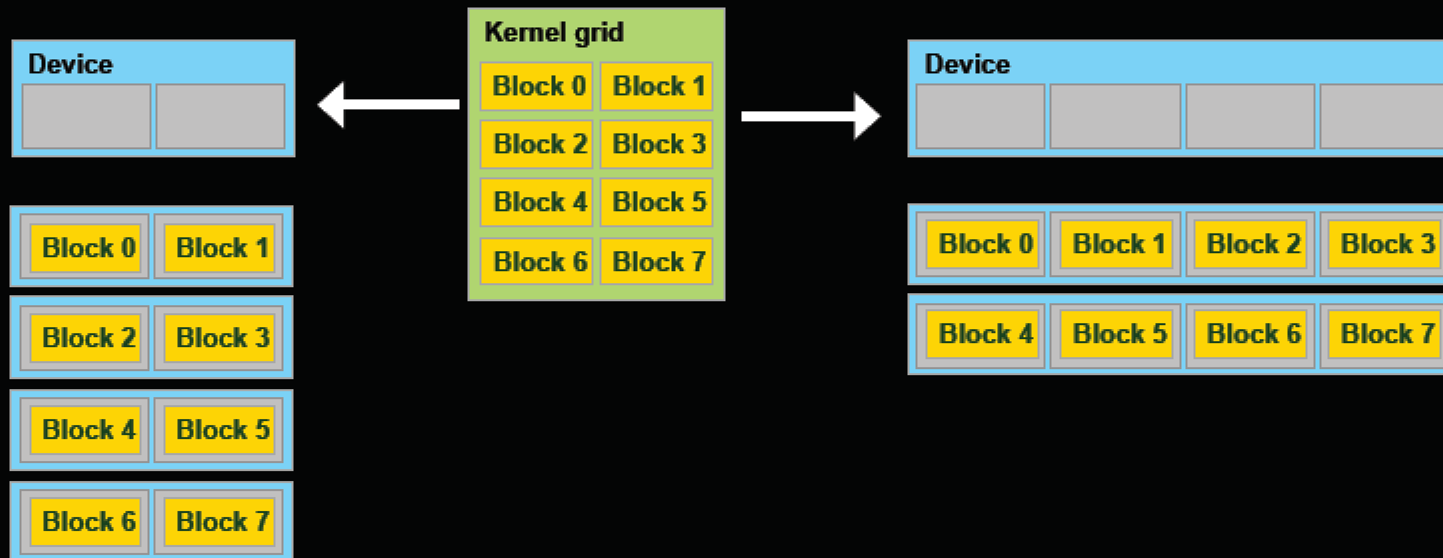
Thread Batching

- Kernel launches a grid of thread blocks
  - Threads within a block cooperate via shared memory
  - Threads within a block can synchronize
  - Threads in different blocks cannot cooperate
- Allows programs to *transparently scale to different* GPUs



Transparent Scalability

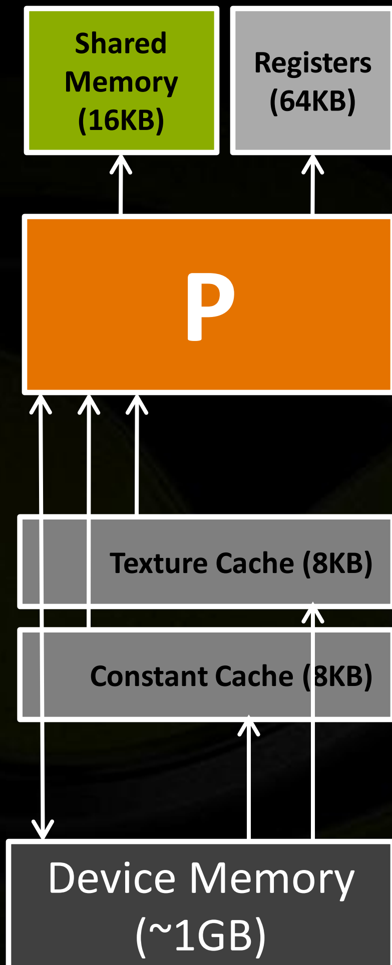
- **Hardware is free to schedule thread blocks on any processor**
  - A kernel scales across parallel multiprocessors



Kernel Memory Access

- Per-thread : Registers (fast)
- Per-block : Shared Memory (fast, on-chip)
- Per-device : Global Memory (Uncached, Off-chip, large persistent across kernel launches)

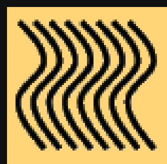
CUDA Memory Model (H/W)

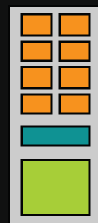


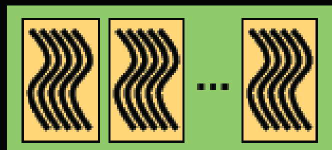
- P has access to Registers (private)
- P has access to Shared-Memory (common to 8 Ps, share and have fun)
- Caches (texture and constant) are not user-managed, true caches
- Device memory is for all Ps of all SMs ! Truly Chaotic !

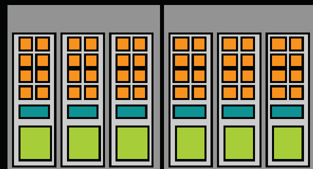
Execution Model

Software | Hardware









- **Threads are executed by thread processors**
- **Thread blocks are executed on multiprocessors**
  - **Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)**
- **A kernel is launched as a grid of thread blocks**
  - **Only one kernel can execute on a device at one time**

CUDA API

- An extension to the ANSI C programming Language
  - No interference of a Graphics API (OpenGL/DirectX)
  - Good learning curve

- Language Extensions in form of
  - Function type qualifiers (variety of functions)
  - Variable type qualifiers (types of variables)
  - Execution Configuration (parameters to kernel)
  - Built-In variables (block and thread Ids)

Function type qualifiers

- __device__ (internal functions needed by main device function)
  - Executed on the *device*
  - Callable from *device*
- __global__ (main Kernel function)
  - Executed on the *device*
  - Callable *only* from *host*
- __host__
  - Executed on the *host*
  - Callable only from *host*

- For functions executed on the device
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments

Variable type qualifiers

- __device__
  - Use with one of the options mentioned below
- __constant__
  - Resides in constant memory space
  - Has the lifetime of an application
  - Accessible from all threads and host
- __shared__
  - Resides in Shared Memory Space of thread block
  - Only accessible from threads within the block
  - Life time of a block

Built-In Variables

- gridDim
  - 3 Dimensional variable holding the dimensions of a *grid*
- blockIdx
  - An int3 type of variable holding the block index within the *grid*
- blockDim
  - 3 Dimensional variable holding the dimensions of a *block*
- threadIdx
  - An int3 type of variable holding the *thread* index within the *block*
- Can not assign values to them nor can you get the address of the above variables

Sample Code Snippets

- DeclSpecs
  - __global__ void convolve (float *image)
  - __shared__ float region[M]
- Keywords
  - region[threadIdx] = image[i]
  - __syncthreads()
- Memory management and Kernel Launch
  - void *myImage = cudaMalloc(bytes)
  - Convolve<<<100,100>>> (myImage);

Increment Array Example

```
void inc_cpu (int *a, int N) {

    int idx;

    for (idx = 0; idx<N; idx++)
        a[idx] = a[idx] + 1;
}

void main() {
    …
    inc_cpu(a, N);
    …
}
```

```
__global__ void inc_cpu (int *a_d, int N) {

    int idx = blockIdx.x * blockDim.x
                   + threadIdx.x;
    if ( idx < N )
        a_d[idx] = a_d[idx] + 1;
}
void main(){
    …
    dim3 dimBlock (num_threads);
    dim3 dimGrid(ceil(N/(float)num_threads));
    inc_gpu<<<dimGrid, dimBlock>>>(a_d, N);
    …
}
```

Q/A

NVIDIA

Schedule

0:00 -- 0:10 : Introduction to the Tutorial, Theme, Speakers.

0:10 -- 0:30 : Basic Concepts -- CPU Architectures, GPUs -- evolution, comparison to earlier models of parallel computing

0:30 -- 0:55 : GPU Architectures in Detail -- NVidia architecture, Intel Larrabee architectural features

0:55 -- 1:30 : GPU Programming models with short examples, CUDA

B R E A K

1:50 -- 2:15 : Case studies of regular applications on the GPU

2:15 -- 2:45 : Case studies of irregular applications on the GPU

2:45 -- 3:20 : GPU Analytical Models

Design Space Optimization, Performance Prediction

3:20 -- 3:30 : Concluding remarks, discussion

© NVIDIA Corporation 2006

<<< Data Primitives >>>

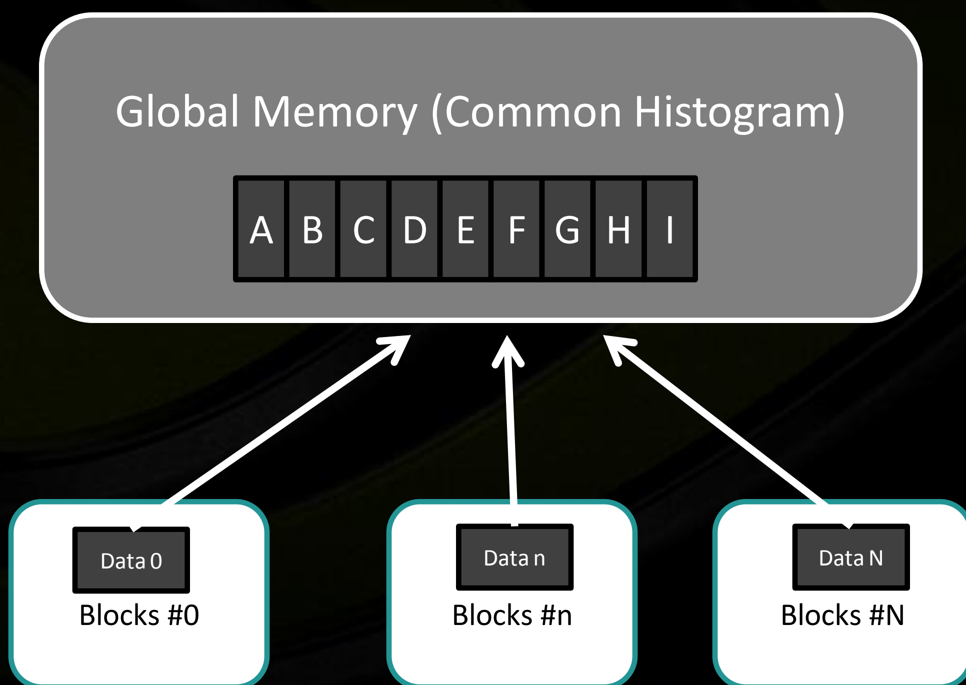
Suryakant Patidar
spatidar@nvidia.com

Histogram Computation

- Counting data based on a given property
  - Counting frequency of words in a book
- Large number of input data
  - Easy to parallelize
- Requires to access common memory areas
  - Memory operations on common area
  - Language provided Atomic Operations
- Widely used

Global Memory Histogram

- **Using Atomic operations on Global Memory**
- **'#Bins' sized array used in global memory to hold the histogram data**




```
for each thread in parallel
    for each element 'x' assigned
     to the thread, sequentially
{
     bin = category[x];
     atomicInc(&globalHist[bin]);
}
```

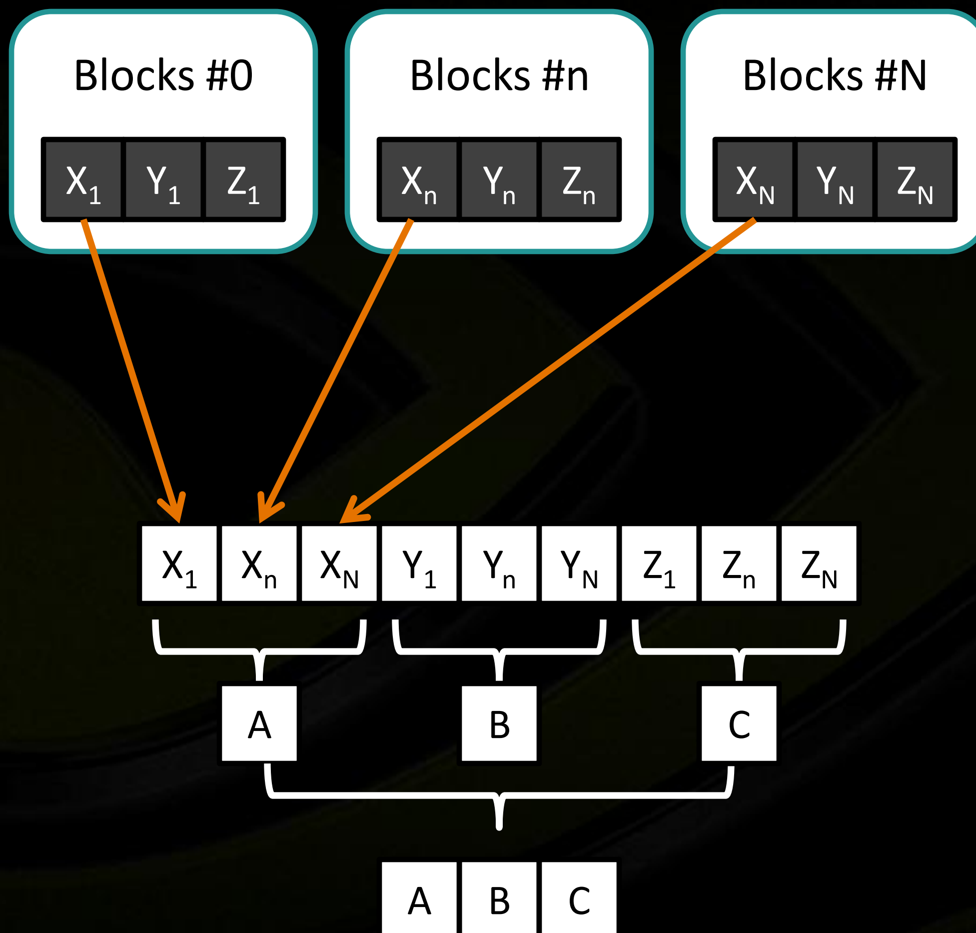
Global Memory Histogram (2)

- Low Memory requirement – one copy of histogram
- Number of Clashes ά Number of Active Threads
  - Highly data dependent, low number of bins tend to perform really bad
- Global Memory is high-latency, ~500cc I/O

Shared Memory Histograms

- A copy of the histogram for each Block
- Each Block counting its own data
- Once all done, we add the sub-histograms to get the final histogram as needed

Shared Memory Histograms (2)



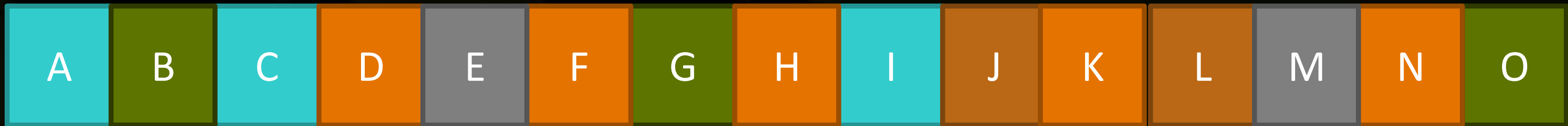
- **Local Shared Memory used to store the sub-histograms**
- **Fast to update and less number of conflicts (N times less conflicts)**
- **Experimentally found to be faster**

Split Operation

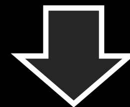
Split can be defined as performing ::

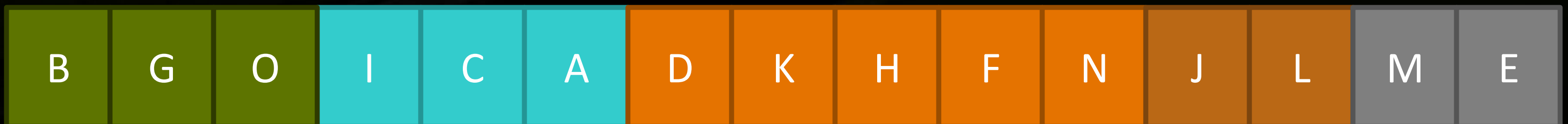
append(x,List[category(x)])

for each x, List holds elements of same category together



Split Operation





Split Sequential Algorithm

I. Count the number of elements falling into each bin

- for each element x of list L do
  - histogram[category(x)]++ [Requires Atomic]

II. Find starting index for each bin (Prefix Sum : Scan Primitive)

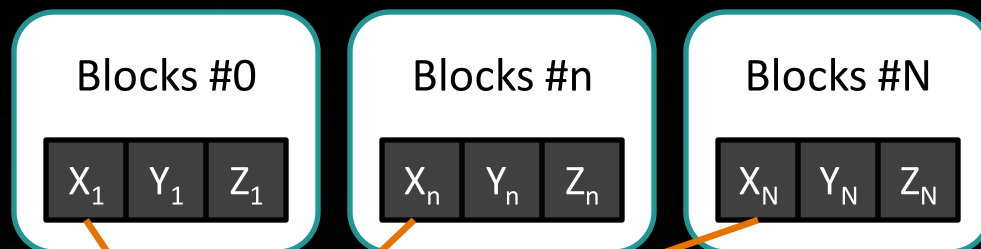
- for each category 'm' do
  - startIndex[m] = startIndex[m – 1]+histogram[m-1]

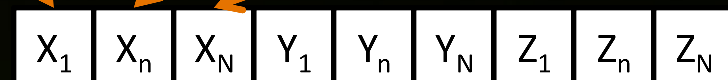
III. Assign each element to the output

- for each element x of list L do [Initialize localIndex[x]=0]
  - itemIndex = localIndex[category(x)]++ [Requires Atomic]
  - globalIndex = startIndex[category(x)]
  - outArray[globalIndex+itemIndex] = x

Split using Shared Atomic

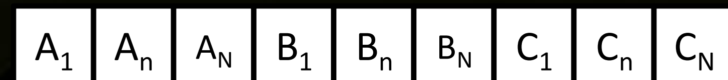
- **Shared Atomic Operations used to build Block-level histograms**
- **Parallel Prefix Sum used to compute starting index**
- **Split is performed by each block for same set of elements used in Step 1**

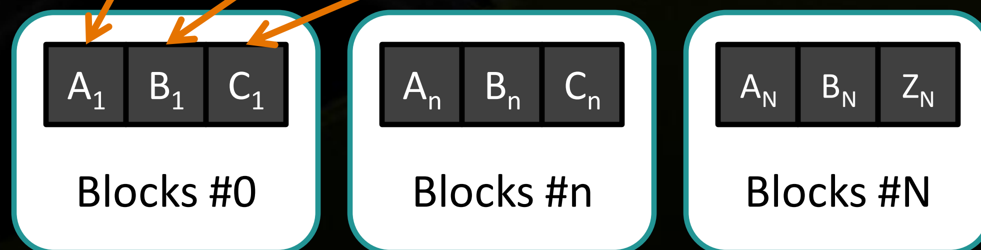




Local Histograms arranged in Column Major Order

Local Histograms arranged in Column Major Order





Matrix Multiplication

- Each thread block is responsible for computing one square sub-matrix Csub of C
- Each thread within the block is responsible for computing one element of Csub

B
BLOCK_SIZE
BLOCK_SIZE
wA
A
C
C_{sub}
BLOCK_SIZE
hA
BLOCK_SIZE BLOCK_SIZE
BLOCK_SIZE
wA
wB

Host Implementation

```
// Matrix multiplication on the (CPU)
    void MatrixMulOnHost(const Matrix M, const Matrix N,
   Matrix P)
   {
       for (int i = 0; i < A.height; ++i)
       for (int j = 0; j < B.width; ++j) {
               float sum = 0;
               for (int k = 0; k < A.width; ++k) {
                       float a = A.elements[i * A.width + k];
                       float b = B.elements[k * B.width + j];
                       sum += a * b;
               }
               C.elements[i * B.width + j] = sum;
         }
   }
```

Device Implementation

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(Matrix A, Matrix B, Matrix C)
{
	// 2D Thread ID
	int tx = threadIdx.x;
	int ty = threadIdx.y;
	// cvalue is used to store the element of the matrix that is computed by the thread
	float Cvalue = 0;
	for (int k = 0; k < A.width; ++k)
	{
		float Aelement = A.elements[ty * A.pitch + k];
		float Belement = B.elements[k * B.pitch + tx];
		Cvalue += Aelement * Belement;
	}
	// Write the matrix to device memory;  each thread writes one element
	C.elements[ty * C.pitch + tx] = Cvalue;
}
```

Q/A

NVIDIA