Multi GPU

In This Lecture

- Parallel Reduction Review
- Using Multi-GPU
- Example: Reduction using Multi-GPU
- Device Enumeration & Selection in Multi-GPU

Parallel Reduction Review

Problem: Summing Array Elements

- Sum all elements of the array.

2	4	3	1	4	4	5	0	7	9	2	3	7	1	7	8	3	7	4	2

Summing Array with CPU

- Sum all elements of the array.

2	4	3	1	4	4	5	0	7	9	2	3	7	1	7	8	3	7	4	2

CPU CODE

```
int sumArray(int* inputArray, int size){
     int tmpSum = 0;
    for(int i = 0; i < size; i++){
        tmpSum += inputArray[i];
    }
    return tmpSum;
}
```

Summing Array with GPU

- Sum all elements of the array.

2	4	3	1	4	4	5	0	7	9	2	3	7	1	7	8	3	7	4	2

CUDA CODE SIMPLE

```
__global__ void sumArray(int* inputArray, int* result, int size){
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if(i < size){
        result[0]+= inputArray[i];
    }
}
```

Summing Array with GPU

- Sum all elements of the array.

2	4	3	1	4	4	5	0	7	9	2	3	7	1	7	8	3	7	4	2

CUDA CODE SIMPLE

```
__global__ void sumA        t* inputArra        result, int size){
    int i = blockIdx.x*b         x +        dx.x;
    if(i < size){
        result[0]+= inputAr
    }
}
```

Wrong Way!

Summing Array with GPU: Atomic Op

- Sum all elements of the array.

2	4	3	1	4	4	5	0	7	9	2	3	7	1	7	8	3	7	4	2

CUDA CODE w/ Atomics

```
__global__ void sumArray(int* inputArray, int* result, int size){
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if(i < size){
        atomicAdd(&result[0],inputArray[i]);
    }
}
```

Summing Array with GPU: Atomic Op

- Sum all elements of the array.

2	4	3	1	4	4	5	0	7	9	2	3	7	1	7	8	3	7	4	2

CUDA CODE w/ Atomics

```
__global__ void sumArray(int* inputArray, int* result, int size){
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if(i < size){
        atomicAdd(&result[0],inputArray[i]);
    }
}
```

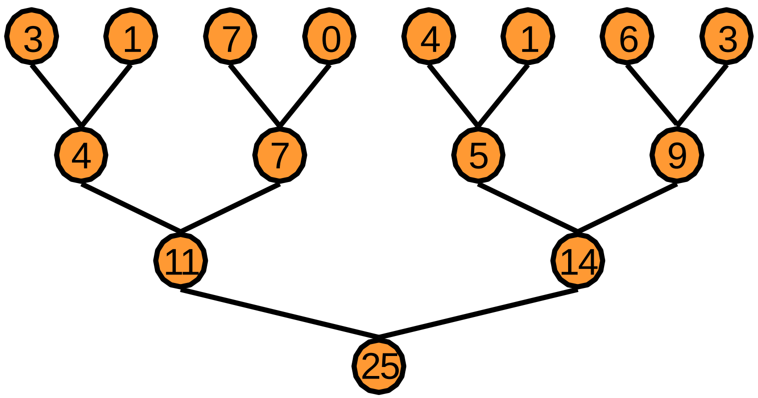
Serial Computation

Parallel Reduction

- **Common and important data parallel primitive**
- **Easy to implement in CUDA**
  - Harder to get it right
- **Serves as a great optimization example**
  - Demonstrates several important optimization strategies

Parallel Reduction

- **Tree-based approach used within each thread block**



- **Need to use multiple thread blocks**
  - To process very large arrays
  - To keep all multiprocessors on the GPU busy
  - Each thread block reduces a portion of the array

- **But how do we communicate partial results between thread blocks?**

Problem: Global Synchronization

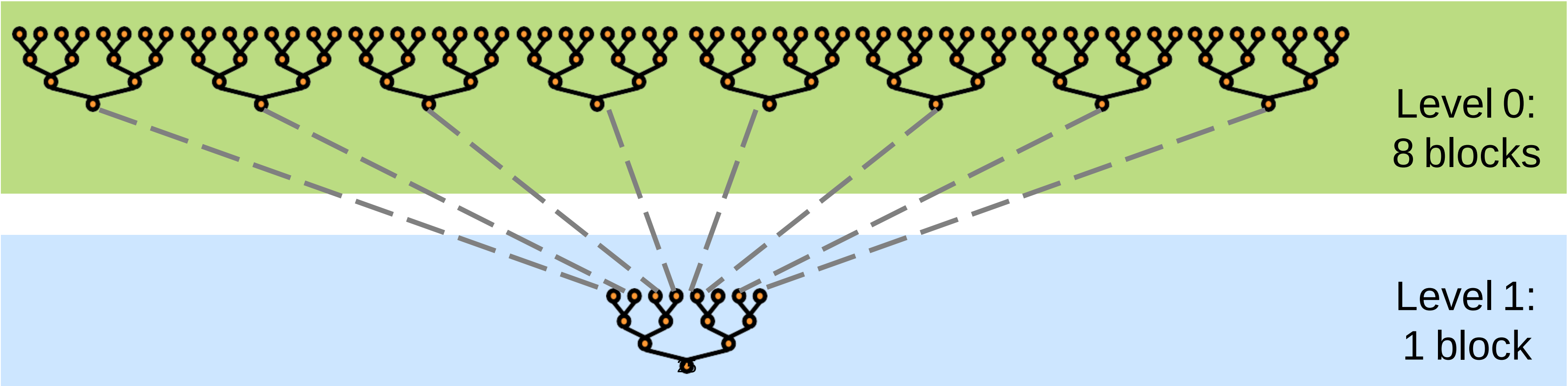
- **If we can synchronize across all thread blocks, we can easily reduce very large arrays**
  - Global sync after each block produces its result
    once all blocks reach sync, continue recursively

- **But CUDA has no global synchronization. Why?**
  - Expensive to build in hardware with high processor count
  - HW developer would force programmer to run fewer blocks
  - no more than total available resident block to avoid deadlock,
    which may reduce overall efficiency

- **Solution: decompose into multiple kernels**
  - Kernel launch serves as a global synchronization point
  - Kernel launch has negligible HW overhead, low SW overhead

Solution: Kernel Decomposition

Global sync by decomposing computation into multiple kernel invocations

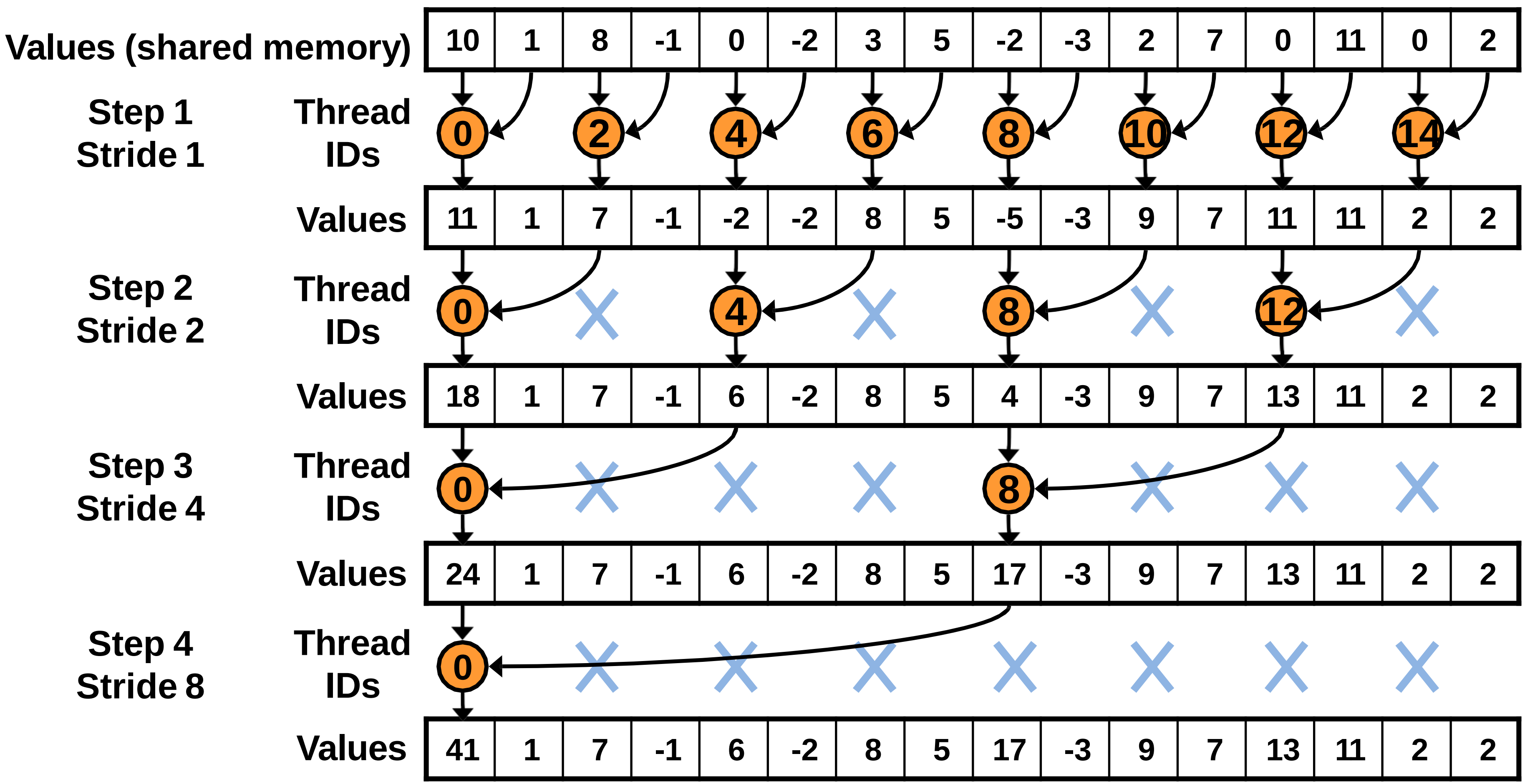


In the case of reductions, code for all levels is the same

Recursive kernel invocation

Parallel Reduction: Interleaved Addressing

This is Block-wise Computation.



Interleaved Addressing #1

```
__global__ void reduce1(int *g_idata, int *g_odata) {

  extern __shared__ int sdata[];

  // each thread loads one element from global to shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[i];
  __syncthreads();

  // do reduction in shared mem
  for (unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0)
    {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }

  // write result for this block to global mem
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Problem:
highly divergent warps, and % operator is very slow

Interleaved Addressing #2

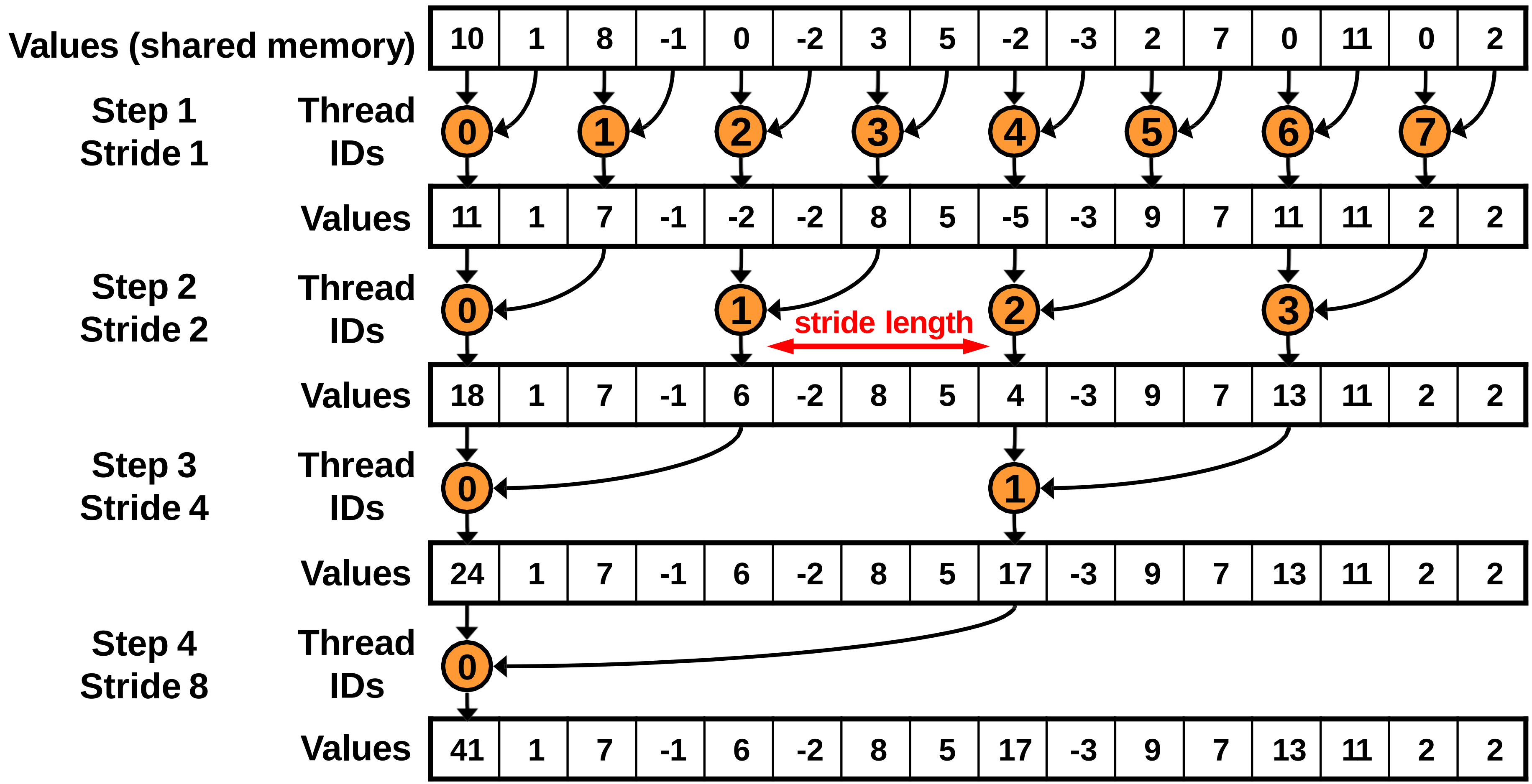
Just replace divergent branch in inner loop:

```
for (unsigned int s=1; s < blockDim.x; s *= 2){
  if (tid % (2*s) == 0) {
    sdata[tid] += sdata[tid + s];
  }
  __syncthreads();
}
```

With strided index and non-divergent branch:

```
for (unsigned int s=1; s < blockDim.x; s *= 2){
  int index = 2 * s * tid;
  if (index < blockDim.x) {
    sdata[index] += sdata[index + s];
  }
  __syncthreads();
}
```

Interleaved Addressing #2



New Problem: Shared Memory Bank Conflicts

Reduction #3: Sequential Addressing

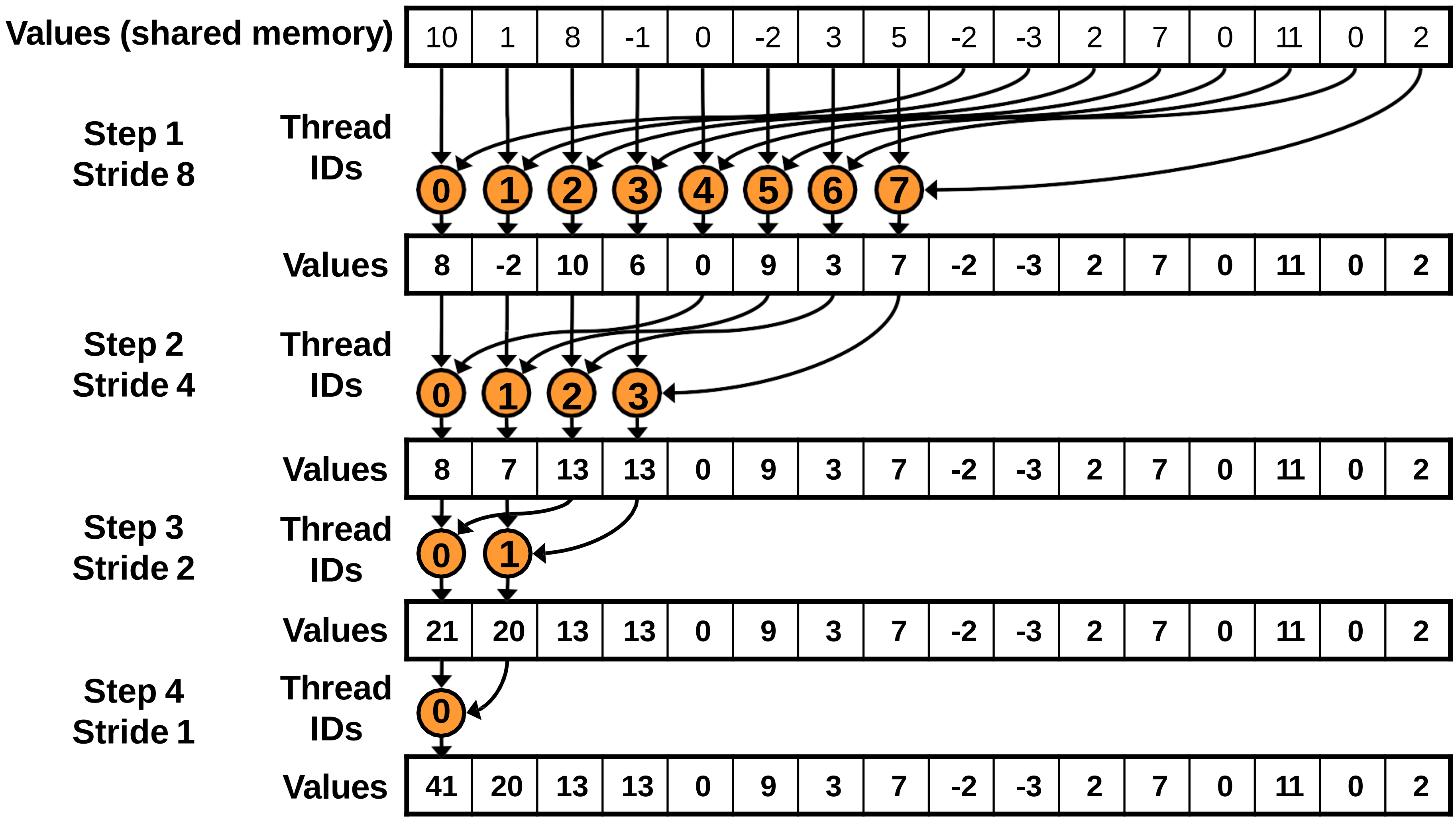
Just replace strided indexing in inner loop:

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {
  int index = 2 * s * tid;
  if (index < blockDim.x) {
    sdata[index] += sdata[index + s];
  }
  __syncthreads();
}
```

With reversed loop and threadID-based indexing:

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
  if (tid < s) {
    sdata[tid] += sdata[tid + s];
  }
  __syncthreads();
}
```

Reduction #3: Sequential Addressing



Sequential addressing reduces bank conflict

Reduction #3: Kernel Function

```
__global__ void reduction(int *g_idata, int *g_odata,  int n){
    // Dynamic allocation of shared memory
    extern __shared__ int sdata[];

    int tid = threadIdx.x;
    int i = blockIdx.x*blockDim.x + threadIdx.x;

    // Copy data into shared memory
    sdata[tid] = (i < n) ? g_idata[i] : 0;
    __syncthreads();

    // do reduction in shared mem
    for (int s=blockDim.x/2; s>0; s>>=1)
    {
        if (tid < s){
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }
    // Write result for this block to global mem
    if(tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Reduction #3: Recursive Kernel Launch

```
int main(){
        ........................
    // Set Grid/Block Dimensions
    int T = 1<<7;//2^7=128: T must be 2^n
    int B = (int)ceil((float)ARRAY_SIZE/T);
    int inputSize = ARRAY_SIZE;
    // Launch Kernel
    while(true){                                  // g_odata in Kernel
        reduction<<<B, T, T*sizeof(int)>>>(d_Array,d_Sum,inputSize);
        if(B == 1) break; //last Step
        inputSize = B;
        B=(int)ceil((float)B/T);
        /*pointer Swap*/
        {
            int* tmp = d_Sum;
            d_Sum = d_Array;
            d_Array = tmp;
        }
    }
        ........................
}
```

Performance for 4M elements reduction

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x

Using Multi-GPU

Count & Set Device

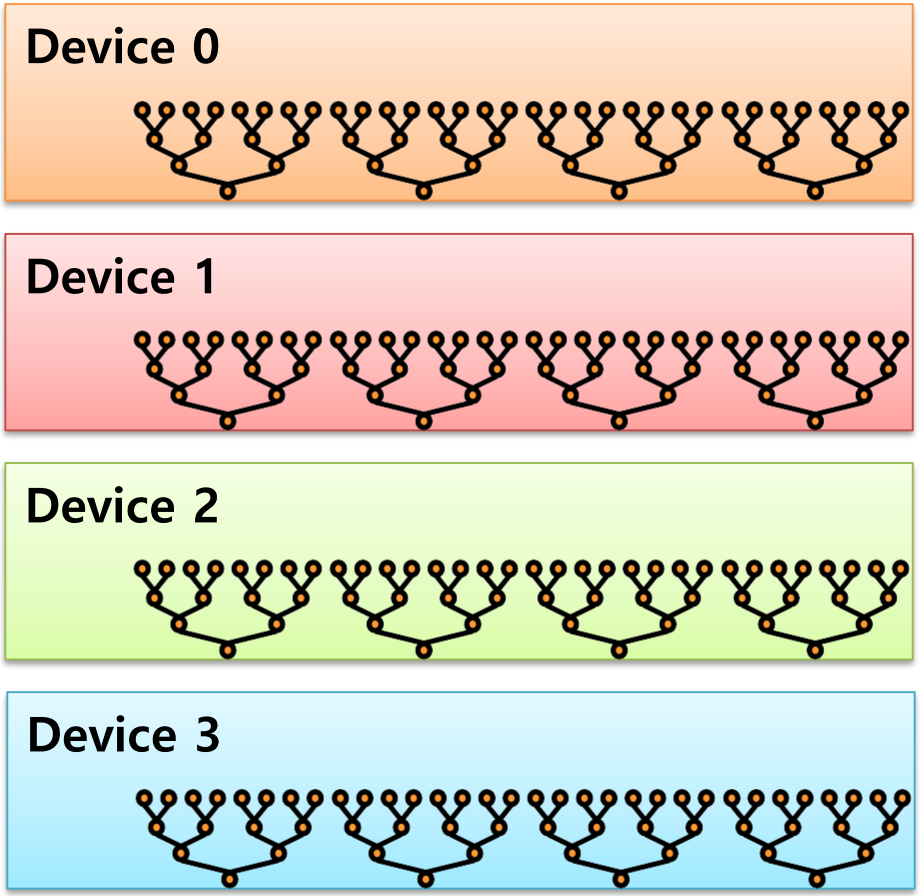
- **Until we have used only one Device at GPU Server.**
  - Our GPU Server has 4 devices (NVIDIA GTX 1080Ti)
  - Without "set device", CUDA automatically set device to no. **0**

- **You can count and set device with runtime function**
  - `cudaSetDevice(int deviceNum)`
    - **Choice device to run the CUDA program**
    - **Default is 0**
  - `cudaGetDeviceCount(int *count)`
    - **Get the number of installed devices at the computer**

  - **Above Functions are included at "cuda-runtime.h"**

Example: Multi-GPU Reduction

- Each device launch reduction kernels with partial data



Program Flow

Main

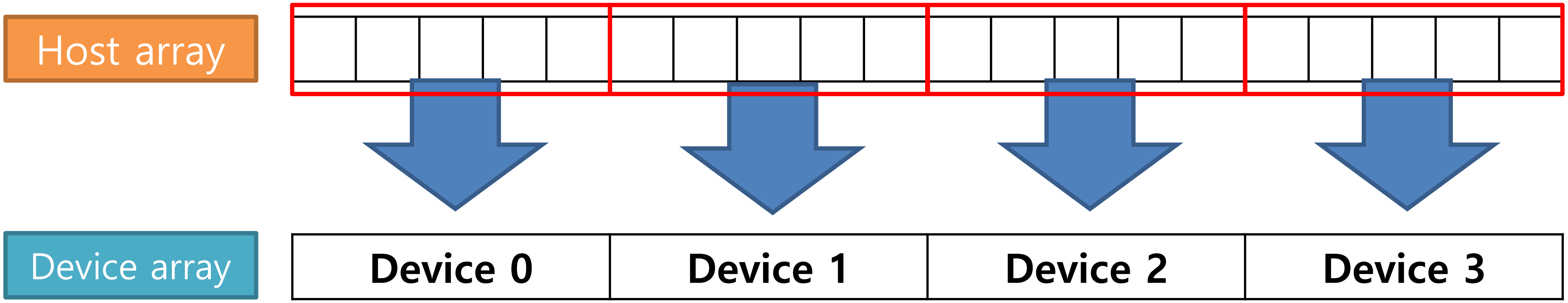
- Count Device
- Memory Initialize - Host
- Memory Initialize - Each Device
- Launch Kernel - Each Device
- Memory Copy - Each Device

```
for(int i=0; i<deviceCount; i++)
{
        cudaSetDevice(i);

        /*Do Something*/
}
```

Memory Initialization

- **Host Memory**
  - Same as previous
- **Device Memory**
  - Allocate memory to each device
    - We'll use double pointers for flexible program
  - Size of the array at each device is
    - `(int)ceil((float)ARRAY_SIZE/deviceCount)`



Code: Initialize

```
int main(){
    int  *h_Array,  *h_Sum; // Host Variables
    int **d_Array, **d_Sum; // Device Variables

    // Alloc & Initialize Host Input Array
    h_Array = (int*)malloc(sizeof(int)*ARRAY_SIZE);
    h_Sum = (int*)malloc(sizeof(int)*ARRAY_SIZE);

    // Omit input initialization

    // Allocate Device Memory
    d_Array = (int**)malloc(sizeof(int*)*deviceCount); // array for device array address
    d_Sum = (int**)malloc(sizeof(int*)*deviceCount);
    const int ARRAY_SIZE_PER_DEVICE = (int)ceil((float)ARRAY_SIZE/deviceCount);
    for(int i =0; i < deviceCount; i++){
        cudaSetDevice(i);
        cudaMalloc((void **) &d_Array[i], sizeof(int)*ARRAY_SIZE_PER_DEVICE);
        cudaMalloc((void **) &d_Sum[i], sizeof(int)*ARRAY_SIZE_PER_DEVICE);
        cudaMemcpy(d_Array[i], h_Array+ARRAY_SIZE_PER_DEVICE*i,
                        sizeof(int)*ARRAY_SIZE_PER_DEVICE, cudaMemcpyHostToDevice);
    }
    .........
}
```

Code: Launch Kernel

```
// Set Grid/Block Dimensions
int T = 1<<7;   // 2^7=128: T must be 2^n
int B = (int)ceil((float)ARRAY_SIZE_PER_DEVICE/T);
int inputSize = ARRAY_SIZE_PER_DEVICE;

// Launch Kernel
while(true){
    for(int i=0; i<deviceCount; i++){
        cudaSetDevice(i);
        reduction<<<B, T, T*sizeof(int)>>>(d_Array[i],d_Sum[i],inputSize);

    }
    if(B == 1) break; //last Step
    inputSize = B;
    B=(int)ceil((float)B/T);
    /*pointer Swap*/
    {
        int** tmp = d_Sum;
        d_Sum = d_Array;
        d_Array = tmp;
    }
}
```

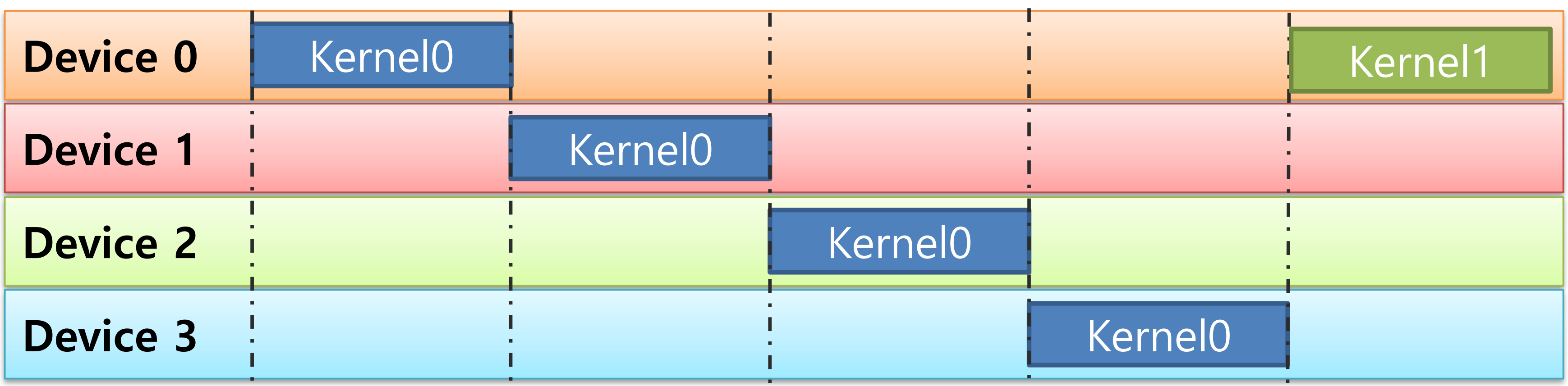
Code: Reduction Kernel

- We will use same Kernel function from Sequential Addressing

```
__global__ void reduction(int *g_idata, int *g_odata,  int n){
    extern __shared__ int sdata[];
    // perform first level of reduction,
    // reading from global memory, writing to shared memory
    int tid = threadIdx.x;
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = (i < n) ? g_idata[i] : 0;
    __syncthreads();
    // do reduction in shared mem
    for (int s=blockDim.x/2; s>0; s>>=1){
        if (tid < s){
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }
    if(tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Problem Analysis

- Kernels are running serially.



Synchronicity in CUDA

- **All CUDA calls can be either synchronous or asynchronous with respect to the host**
  - Synchronous: enqueue work and wait for completion
  - Asynchronous: enqueue work and return immediately

- **Kernel Launches are asynchronous to host and automatically overlapped with host**

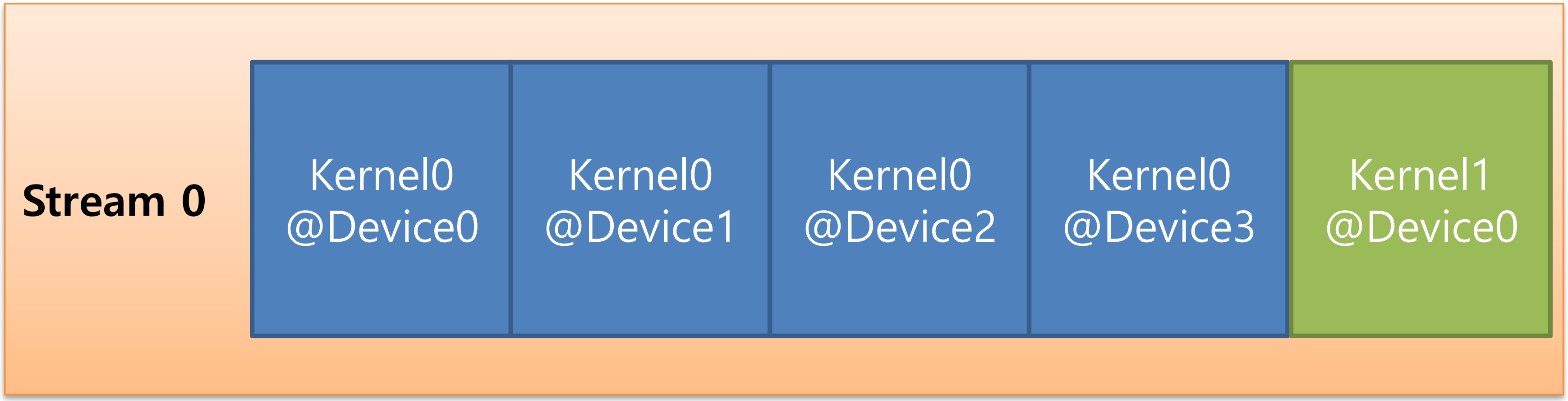
CUDA Stream

- **A stream is a queue of device work**
  - The host places work in the queue and continues on immediately
  - Device schedules work from streams when resources are free
- **CUDA operations are placed within a stream**
  - e.g. Kernel launches, memory transfers

- **Operations within the same stream are ordered (FIFO) and cannot overlap**
- **Operations in different streams are unordered and can overlap**

CUDA Stream(Contd.)

- **Without stream setup, Every CUDA Call is automatically controlled by Stream #0**
  - Kernels are launched serially



Managing Multi-Streams

- **cudaStream_t stream;**
  - Declares a stream handle
- **cudaStreamCreate(&stream);**
  - Allocates a stream
- **cudaStreamDestroy(stream);**
  - Deallocates a stream
  - Synchronizes host until work in stream has completed

Pacing Work into a Stream

- **Stream is the 4th parameter of kernel launch**
  - kernel<<< blocks, threads, sharedmem, **stream**>>>();

- **A stream contains its own API calls**
  - cudaMemcpyAsync(dst, src, size, dir, stream);

    - cudaMemcpy(...)
      - Places memory transfer operation into the default stream
      - Synchronous: Must complete prior to returning

    - cudaMemcpyAsync(..., stream)
      - Places memory transfer into a stream and returns immediately

Code with Stream: Initialize

```
int main(){
    int *h_Array, *h_Sum;   // Host Variables
    int **d_Array, **d_Sum; // Device Variables
    cudaStream_t *stream;

    // Input data initialization (omitted)

    // Allocate Device Memory
    d_Array = (int**)malloc(sizeof(int*)*deviceCount); //array for device array address
    d_Sum = (int**)malloc(sizeof(int*)*deviceCount);   //array for device array address
    const int ARRAY_SIZE_PER_DEVICE = (int)ceil((float)ARRAY_SIZE/deviceCount);
    stream = (cudaStream_t*)malloc(sizeof(cudaStream_t)*deviceCount);
    for(int i =0; i < deviceCount; i++){
        cudaSetDevice(i);
        cudaStreamCreate(&stream[i]);
        cudaMalloc((void **) &d_Array[i], sizeof(int)*ARRAY_SIZE_PER_DEVICE);
        cudaMalloc((void **) &d_Sum[i], sizeof(int)*ARRAY_SIZE_PER_DEVICE);
        cudaMemcpyAsync(d_Array[i], h_Array+ARRAY_SIZE_PER_DEVICE*i,
                    sizeof(int)*ARRAY_SIZE_PER_DEVICE, cudaMemcpyHostToDevice,stream[i]);
    }
    .........
}
```

Code with Stream: Launch Kernel

```
// Set Grid/Block Dimensions
int T = 1<<7;   // 2^7=128: T must be 2^n
int B = (int)ceil((float)ARRAY_SIZE_PER_DEVICE/T);
int inputSize = ARRAY_SIZE_PER_DEVICE;

// Launch Kernel
while(true){
    for(int i =0; i<deviceCount; i++){
        cudaSetDevice(i);
        reduction<<<B, T, T*sizeof(int),stream[i]>>>(d_Array[i],d_Sum[i],inputSize);

    }
    if(B == 1) break; //last Step
    inputSize = B;
    B=(int)ceil((float)B/T);
    /*pointer Swap*/
    {
        int** tmp = d_Sum;
        d_Sum = d_Array;
        d_Array = tmp;
    }
}
```

Code with Stream: Merge Result

```
// Copy Result to Host
int totalSum = 0;
for(int i=0; i<deviceCount; i++)
{
    cudaSetDevice(i);
    cudaMemcpy(h_Sum+i, d_Sum[i], sizeof(int), cudaMemcpyDeviceToHost );
    totalSum += h_Sum[i];    // d_Sum[i][0]
}

// Free Device Memory & Destroy CUDA streams
for(int i=0; i< deviceCount; i++)
{
    cudaSetDevice(i);
    cudaFree(d_Array[i]);
    cudaFree(d_Sum[i]);
    cudaStreamDestroy(stream[i]);
}
```

Device Enumeration & Selection in Multi-GPU

Monitor Device: Command Line

- **You can check device usage**
  - Command: nvidia-smi

penorchid@ubuntu: ~/cuda

```
Fri Nov 23 10:55:38 2018
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 396.54                 Driver Version: 396.54                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 108...  Off  | 00000000:05:00.0  On |                  N/A |
| 23%   36C    P2    55W / 250W |    310MiB / 11177MiB |      3%      Default |
+-------------------------------+----------------------+----------------------+
|   1  GeForce GTX 108...  Off  | 00000000:06:00.0 Off |                  N/A |
| 23%   38C    P8    15W / 250W |     12MiB / 11178MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  GeForce GTX 108...  Off  | 00000000:09:00.0 Off |                  N/A |
| 23%   38C    P8    17W / 250W |     12MiB / 11178MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  GeForce GTX 108...  Off  | 00000000:0A:00.0 Off |                  N/A |
| 23%   35C    P8     8W / 250W |     12MiB / 11178MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      6054      G   /usr/lib/xorg/Xorg                            85MiB |
|    0      6826      G   compiz                                        59MiB |
|    0     12347    C+G   ./particles                                  160MiB |
+-----------------------------------------------------------------------------+
penorchid@ubuntu:~/cuda$
```

- **Above device indices are different from runtime API index**
  - **Fortunately, our GPU Server has same device indices**

Device Index Check

- **Compare PCI Bus Id to check device index**
  - NVML & Run-time can get device PCI Bus information

penorchid@ubuntu: ~/cuda

```
penorchid@ubuntu:~/cuda$ nvcc device.cu  -o EXE2 -lnvidia-ml
penorchid@ubuntu:~/cuda$ ./EXE2
Found 4 devices

Listing devices:
NVML GPU id: 0
Bus id: 5
Runtime GPU id: 0
Bus id: 5
NVML GPU id: 1
Bus id: 6
Runtime GPU id: 1
Bus id: 6
NVML GPU id: 2
Bus id: 9
Runtime GPU id: 2
Bus id: 9
NVML GPU id: 3
Bus id: 10
Runtime GPU id: 3
Bus id: 10
All done.
Press ENTER to continue...
```

Device Monitoring with Library

- **You can check device usage with NVML**

- **NVML: NVidia Management Library**
  - API for monitoring and managing various states of the devices
    - Utilization
    - Memory
    - Fan
    - Power
    - PCI
    - Etc..
  - Compile with the option **-lnvidia-ml**

- **This device information can be used to select a device.**

NVML Init&Release Guide

- **`nvmlInit()`**
  - Initialize NVML
  - Communicate with any functional GPU while there are other GPUs unstable or in bad states.

- **`nvmlShutdown()`**
  - Shut down NVML by releasing all GPU resources previously allocated by **`nvmlInit()`**.

NVML Device Handle Guide

- **`nvmlDeviceGetHandleByIndex (unsigned int index, nvmlDevice_t* device )`**
  - Acquire the handle for a particular device, based on its index.

- **`nvmlDeviceGetSomething (nvmlDevice_t device, nvmlSomething_t *something)`**
  - These functions give corresponding properties of device

```
nvmlUtilization_t utilInfo;
nvmlDeviceGetHandleByIndex(i, &device);
nvmlDeviceGetUtilizationRates(device, &utilInfo);
```

Device Selection: Example

```
int getUsableDeviceID(){
    int deviceID = -1;
    int leastUsage = 100;
    int deviceCount = 0;

    cudaGetDeviceCount(&deviceCount);   // Get the number of devices
    nvmlInit();                         // Initialize NVML

    for(int i=0; i<deviceCount; i++){
        nvmlDevice_t device;            // NVML Device Structure to handle device
        nvmlUtilization_t utilInfo;     // Structure to save GPU & Memory Usage (in rate)
        nvmlDeviceGetHandleByIndex(i, &device);
        nvmlDeviceGetUtilizationRates(device, &utilInfo);
        if(leastUsage>utilInfo.gpu){
            leastUsage = utilInfo.gpu;
            deviceID = i;               // Find least used GPU then select
        }
    }
    nvmlShutdown(); // Release NVML
    if(deviceID < 0) printf("There is no usable GPU.\n");
    else printf("Usable device ID is %d.\n",deviceID);
    return deviceID;
}
```

Device Selection: Example

Utilization rate Comparison can be replaced to other properties comparisons.

- Remaining Memory
- Power Usage
- Temperature
- etc.

```
int getUsableDeviceI
    int deviceID = -
    int leastUsage =
    int deviceCount =

    cudaGetDeviceCou
    nvmlInit();

    for(int i=0; i<de
        nvmlDevice_t device;            // NVML Device Structure to handle device
        nvmlUtilization_t utilInfo;     // Structure to save GPU & Memory Usage (in rate)
        nvmlDeviceGetHandleByIndex(i, &device);
        nvmlDeviceGetUtilizationRates(device, &utilInfo);
        if(leastUsage>utilInfo.gpu){
            leastUsage = utilInfo.gpu;
            deviceID = i;               // Find least used GPU then select
        }
    }
    nvmlShutdown(); // Release NVML
    if(deviceID < 0) printf("There is no usable GPU.\n");
    else printf("Usable device ID is %d.\n",deviceID);
    return deviceID;
}
```

Memory Check

- **nvmlDeviceGetMemoryInfo**
  (nvmlDevice_t device, nvmlMemory_t * memory)
  - Structure **nvmlMemory_t** has
    - **unsigned long long total**
      - Total installed memory (in bytes)
    - **unsigned long long free**
      - Unallocated memory (in bytes).
    - **unsigned long long used**
      - Allocated memory (in bytes).

Power Check

- **nvmlDeviceGetPowerManagementLimit**
  (nvmlDevice_t device, unsigned int * limit)
  - Get Power Limitation

- **nvmlDeviceGetPowerUsage**
  (nvmlDevice_t device, unsigned int * power)
  - Get Current Power Usage

Temperature Check

- **nvmlDeviceGetTemperature**

```
(nvmlDevice_t device,
 nvmlTemperatureSensors_t sensorType,
 unsigned int * temp)
```

  - `sensorType` Flag that indicates which sensor reading to retrieve
    - Generally use **NVML_TEMPERATURE_GPU**

Check Running Process

- `nvmlDeviceGetComputeRunningProcesses`
```
    (nvmlDevice_t device,
     unsigned int * infoCount,
     nvmlProcessInfo_t * infos)
```
  - **infoCount** Reference in which to provide the infos array size, and to return the number of returned elements
  - **infos** Reference in which to return the process information
    - `unsigned int pid`
      - Process ID
    - `unsigned long long usedGpuMemory`
      - Amount of used GPU memory in bytes

Other NVML API Reference

- **You can search other API from**
  - http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/NVML/nvml.pdf