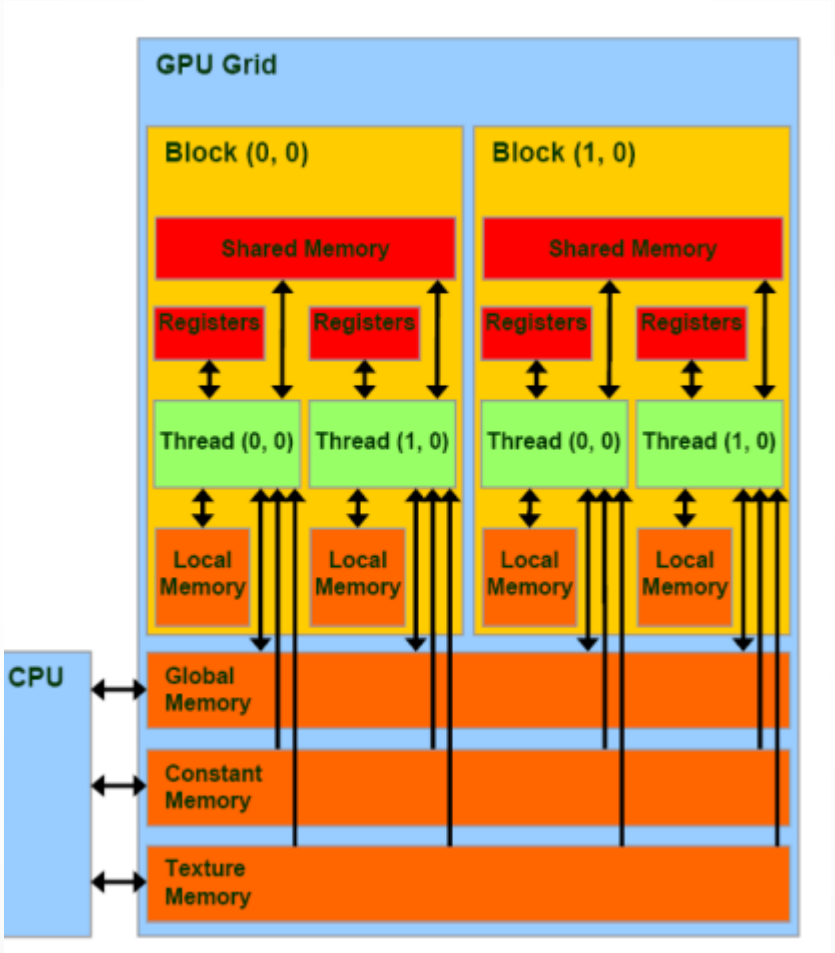
Pinned Memory, Unify Memory and Parallel Reduction

Leiming Yu
yu.lei@husky.neu.edu

Topics

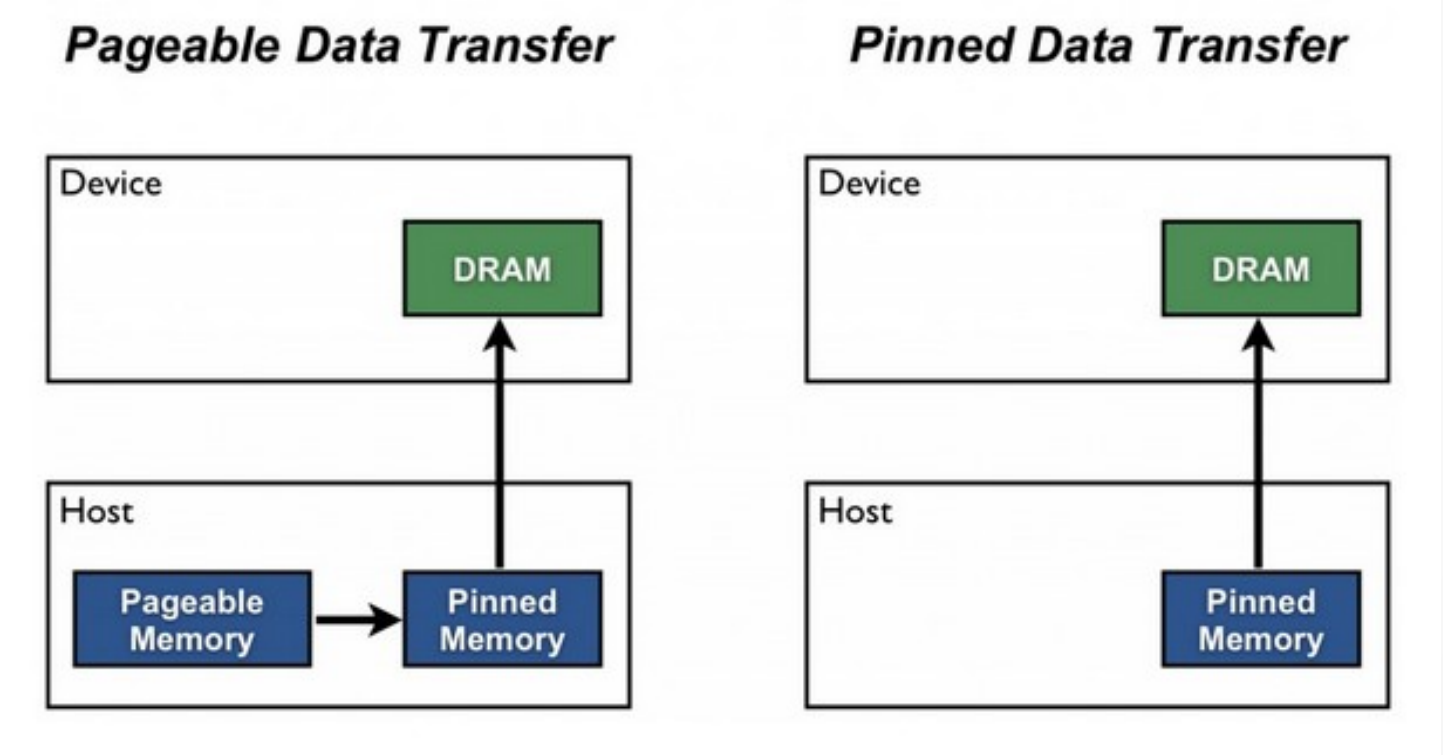
- Memory Transfer
  - Pinned/Nonpinned Memory/Unified Memory
- Stream / Concurrency / Copy Engines
- Parallel reduction

Catch up


GPU Grid
Block (0, 0)
Block (1, 0)
Shared Memory
Shared Memory
Registers
Registers
Registers
Registers
Thread (0, 0)
Thread (1, 0)
Thread (0, 0)
Thread (1, 0)
Local Memory
Local Memory
Local Memory
Local Memory
CPU
Global Memory
Constant Memory
Texture Memory

Pinned Memory

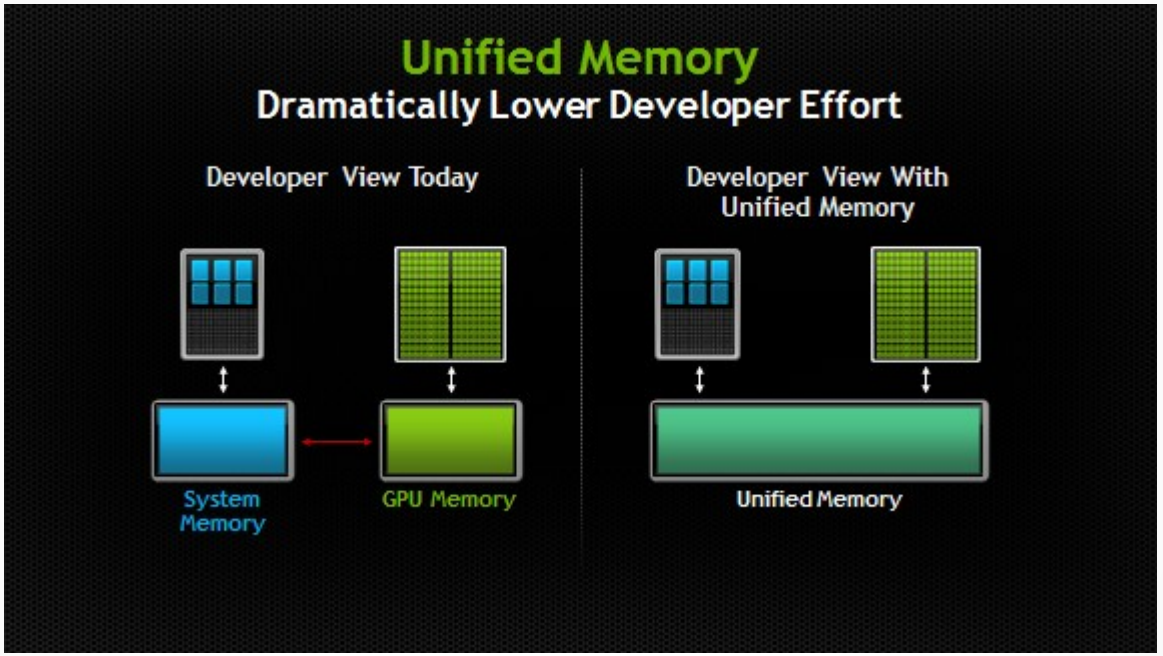
Host (CPU) data allocations are pageable by default.



`cudaMalloc` `cudaMallocHost`

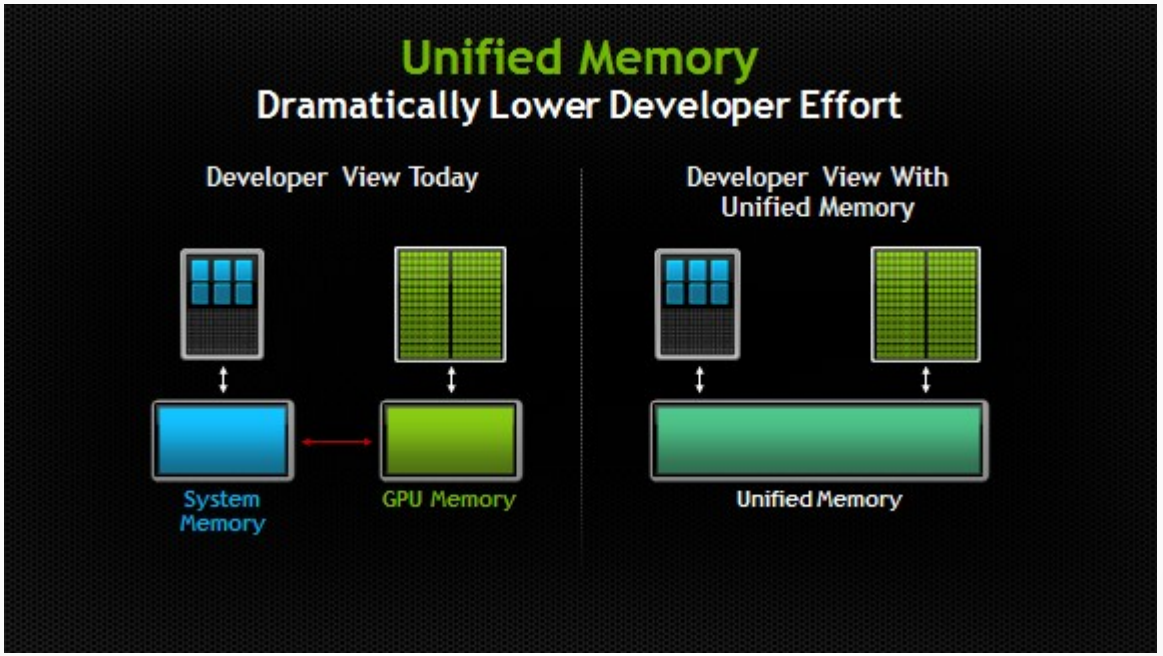
Unified Memory

Programming feature



Unified Memory

Programming feature



Unified Memory

Programming feature
Driver handles the lookup and transfer

CPU Code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

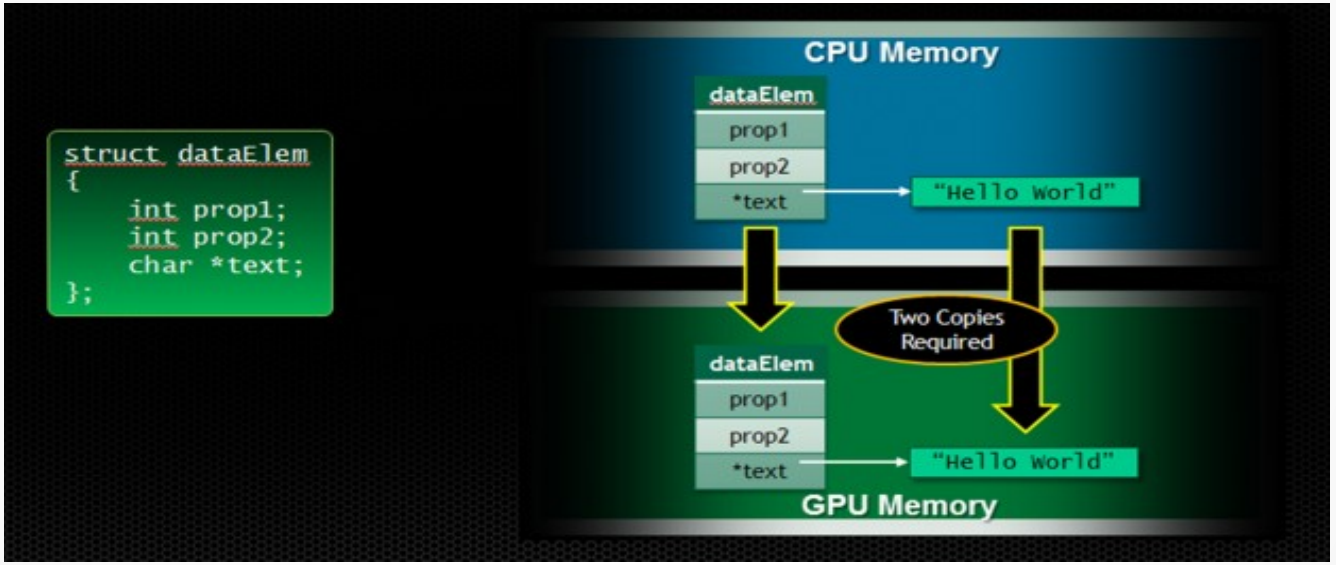
CUDA 6 Code with Unified Memory

```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```

Unified Memory / Unified Virtual Addressing

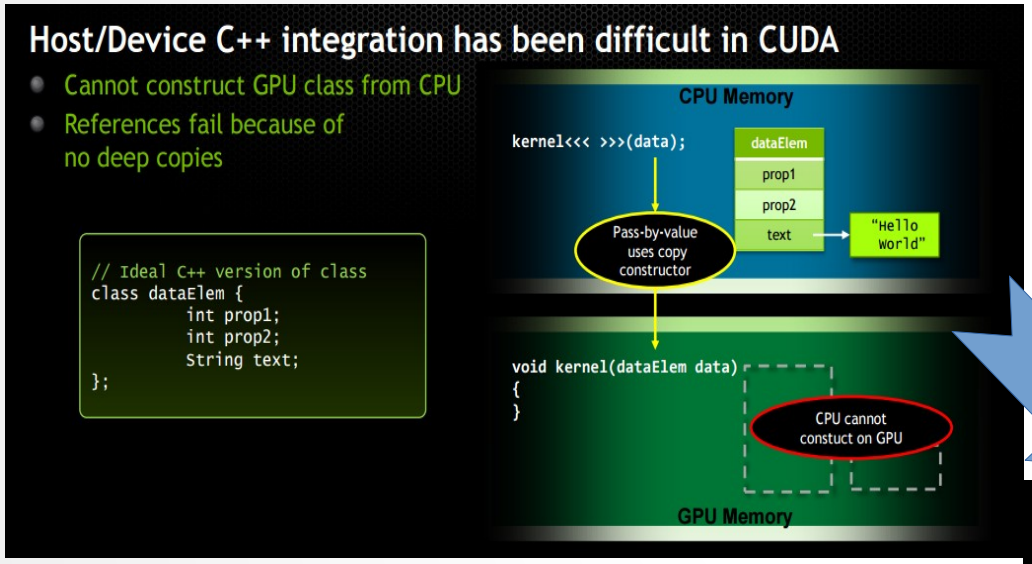
UVA , cuda 4.0

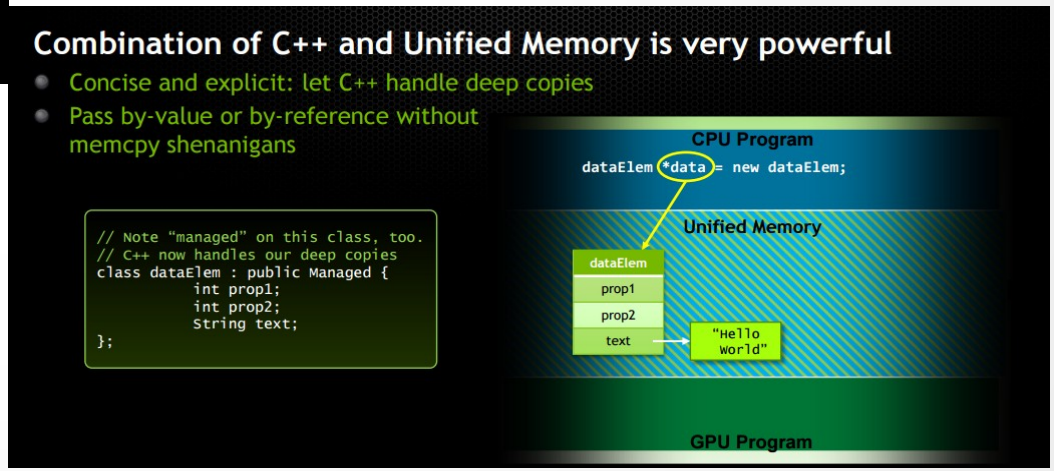
UM
* eliminate deep copy



Unified Memory

UM
* eliminate deep copy
* referencing / linking works now.

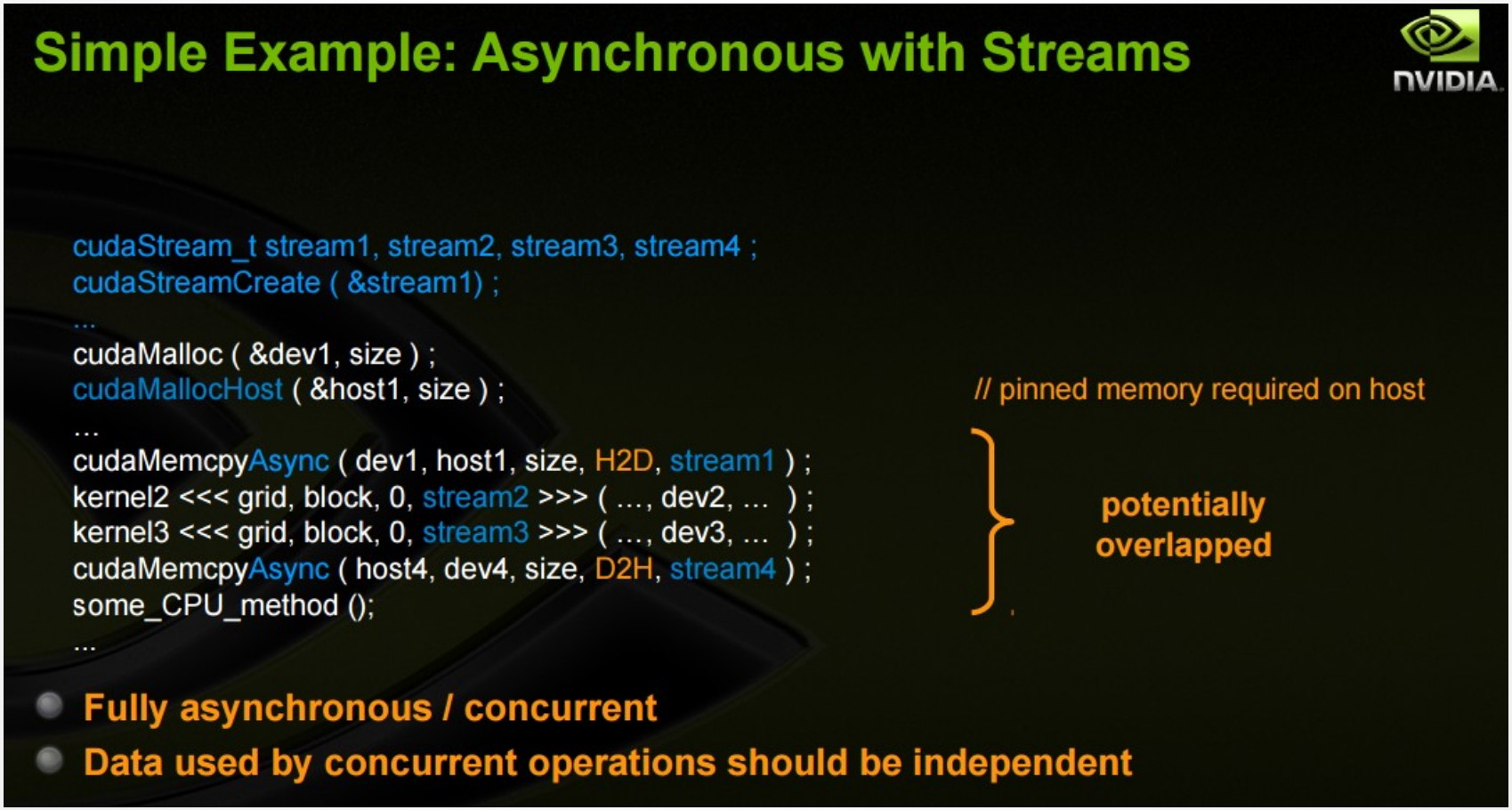




Streams

- Cuda stream :

  A stream in CUDA is a sequence of operations that execute on the device in the order in which they are issued by the host code



https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/

Streams

- Synchronization
  - CudaStreamSynchronize : block host threads
  - CudaStreamWaitEvent: wait for event in a stream
    - Build deepencies between streams

      (http://cedric-augonnet.com/declaring-dependencies-with-cudastreamwaitevent/)

How many different ways to do a complete kernel operation (H2D/Kernel/D2H) for multiple streams?

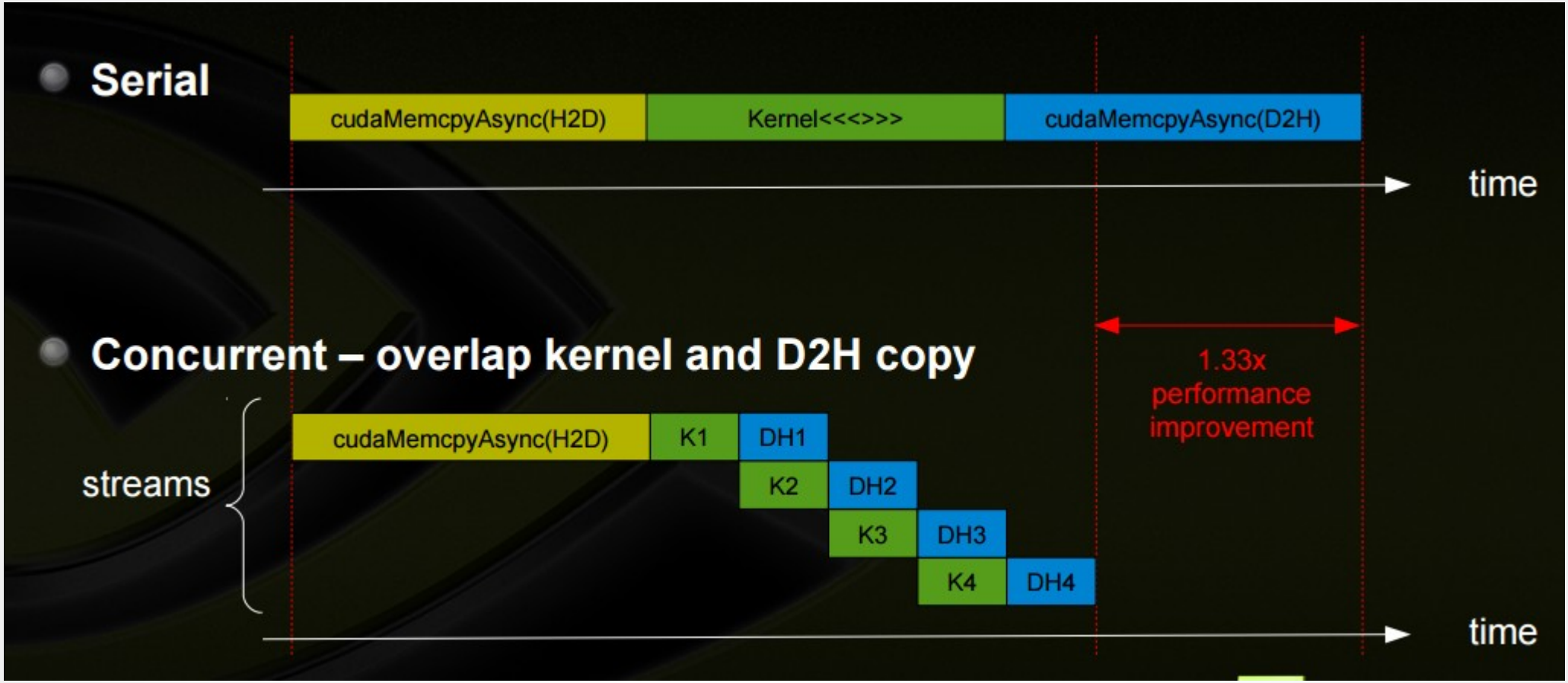
https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/

Concurrency

Left blank intentionally

https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/

Concurrency

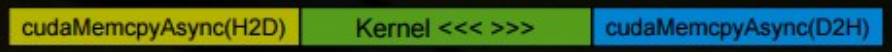


https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/

Concurrency

Amount of Concurrency

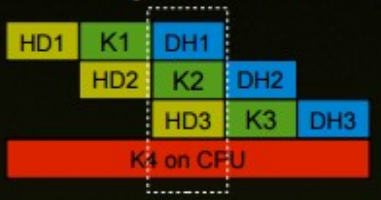
- **Serial** (1x)



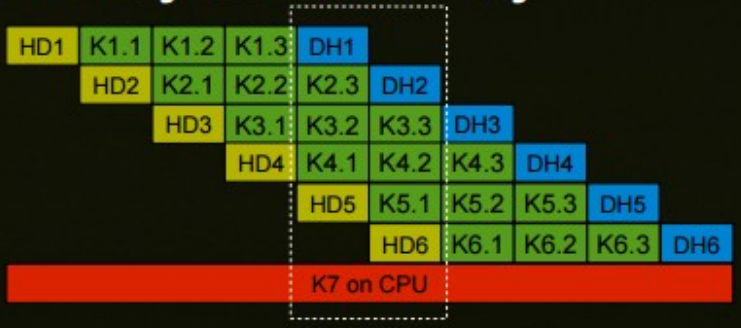
- **2-way concurrency** (up to 2x)

- **3-way concurrency** (up to 3x)

- **4-way concurrency** (3x+)



- **4+ way concurrency**



https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/

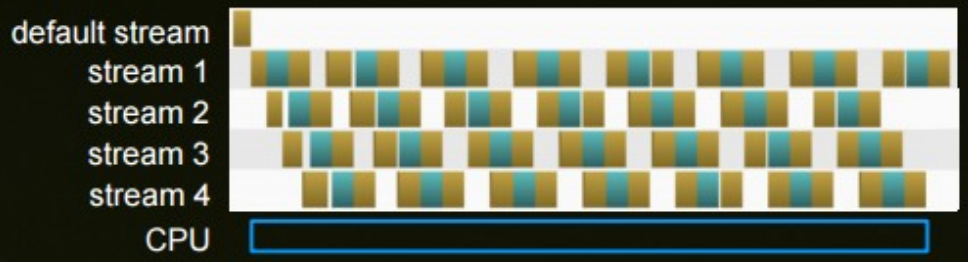
Concurrency

Example – Tiled DGEMM

- **CPU** (4core Westmere x5670 @2.93 GHz, MKL)
  - **43 Gflops**
- **GPU** (C2070)
  - **Serial : 125 Gflops (2.9x)**
  - **2-way : 177 Gflops (4.1x)**
  - **3-way : 262 Gfllops (6.1x)**
- **GPU + CPU**
  - **4-way con.: 282 Gflops (6.6x)**
  - **Up to 330 Gflops for larger rank**
- **Obtain maximum performance by leveraging concurrency**
- **All communication hidden – effectively removes device memory size limitation**

DGEMM: m=n=8192, k=288

Nvidia Visual Profiler (nvvp)



Why ? When ?

https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/

Copy Engines

How many different ways to do a complete kernel operation (H2D/Kernel/D2H) for multiple streams?

https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/

Copy Engines

- Method 1

```
for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes, cudaMemcpyHostToDev
  kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
  cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes, cudaMemcpyDeviceToH
}
```

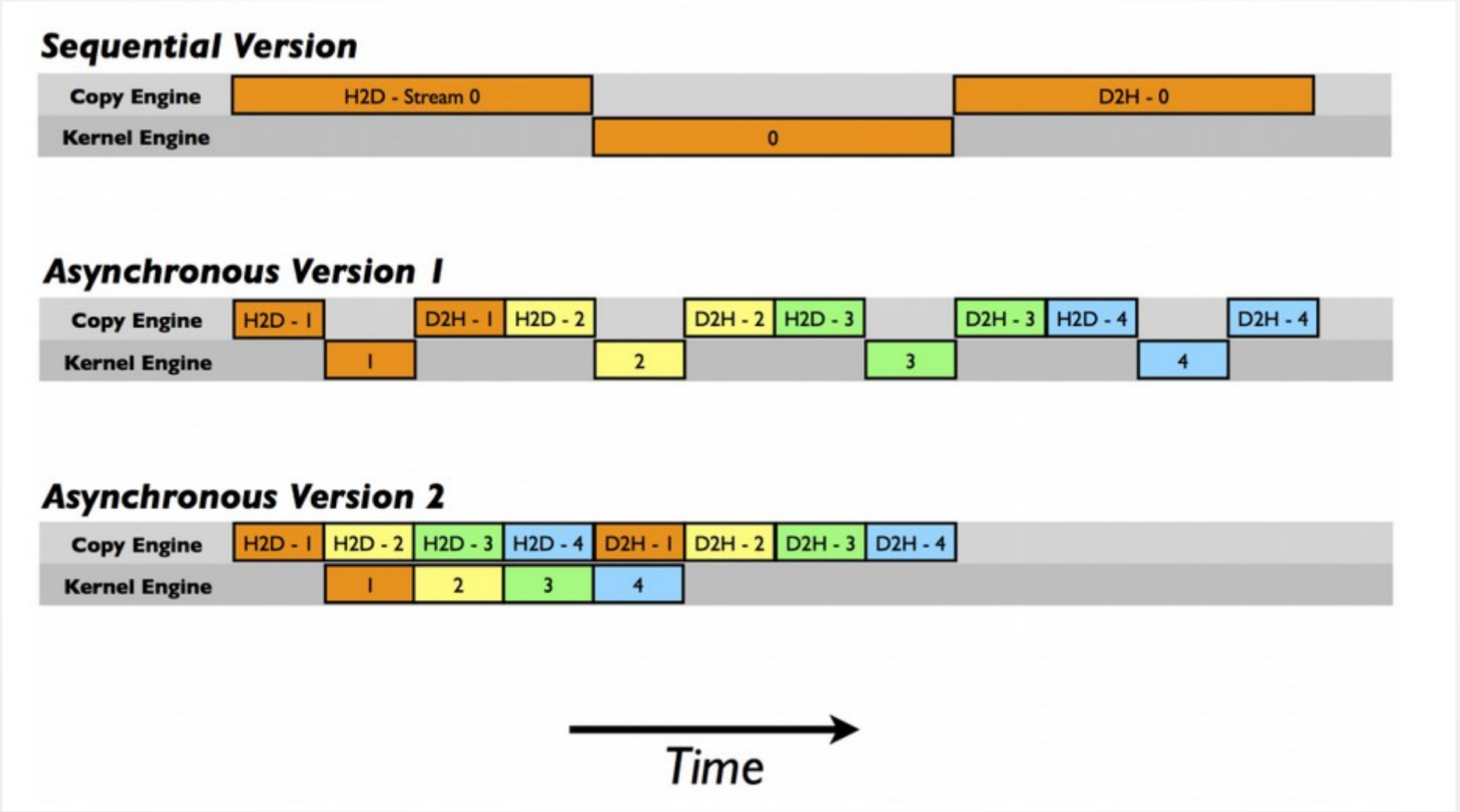
Copy Engines

- Method 2

```
for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&d_a[offset], &a[offset],
                  streamBytes, cudaMemcpyHostToDevice, cudaMemcpyHostToDevic
}

for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
}

for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&a[offset], &d_a[offset],
                  streamBytes, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToHos
}
```

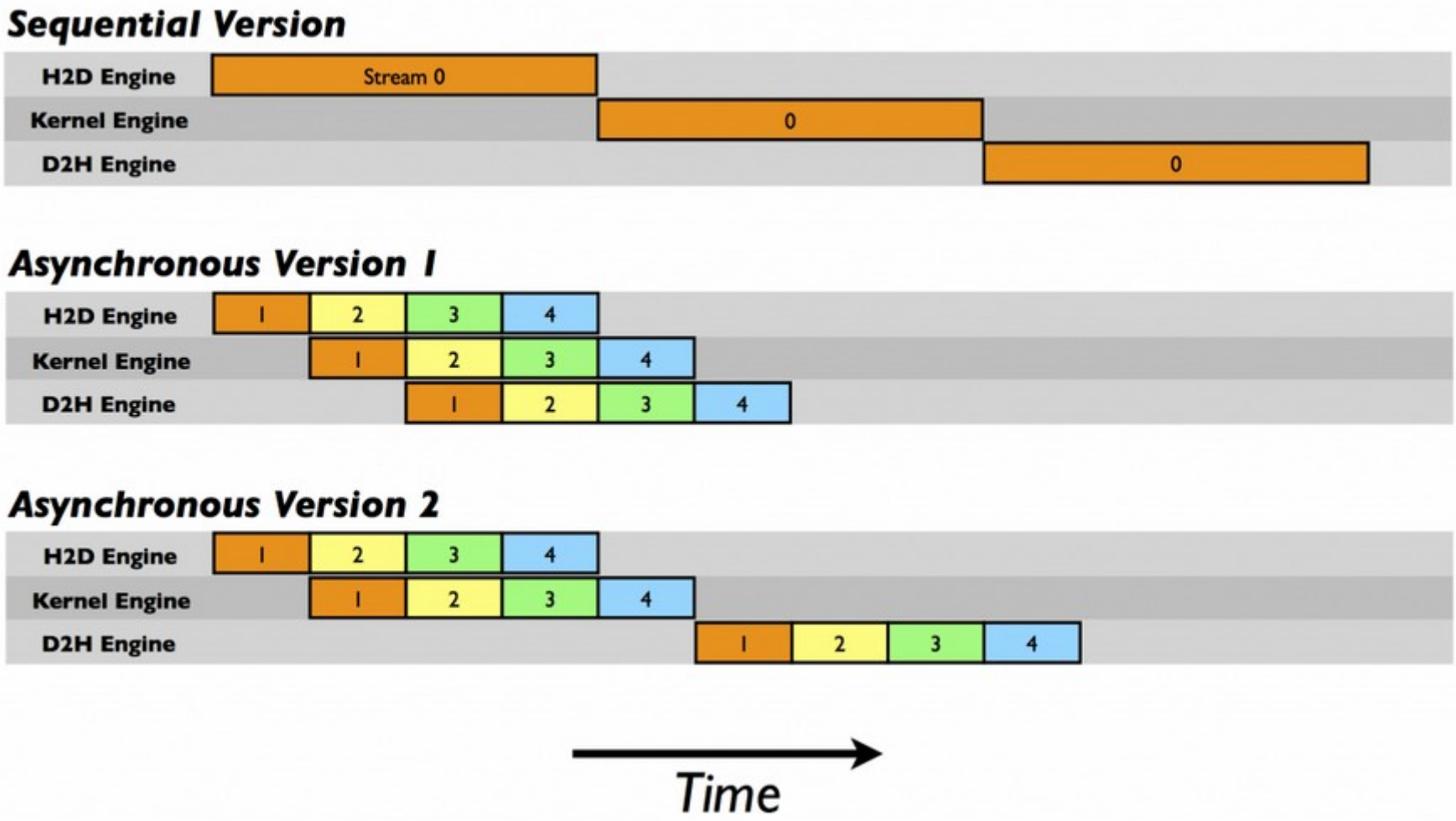
Copy Engines

Device has one copy engine



Copy Engines

Device has two copy engines



Parallel Reduction

Goal

To master the concept of control divergence through reduction trees

- A class of computation
- Control divergence
- Thread utilization
- Work efficiency

Parallel Reduction

commutative and associative

A commonly used strategy for processing large input data sets

- There is no required order of processing elements in a data set (associative and commutative)
- Partition the data set into smaller chunks
- Have each thread to process a chunk
- Use a reduction tree to summarize the results from each chunk into the final answer

- **Google and Hadoop MapReduce frameworks are examples of this pattern**
- **We will focus on the reduction tree step for now.**

Parallel Reduction

Reduction **is also needed to clean up after some commonly used parallelizing transformations**

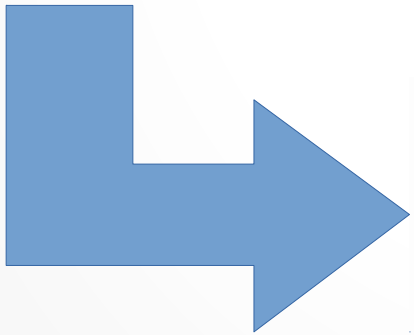
Privatization

- Multiple threads write into an output location
- Replicate the output location so that each thread has a private output location
- Use a reduction tree to combine the values of private locations into the original output location

Parallel Reduction

Summarize a set of input values into one value using a "reduction operation"

- Max
- Min
- Sum
- Product
- Often with user defined reduction operation function as long as the operation
  - Is associative and commutative
  - Has a well-defined identity value (e.g., 0 for sum)

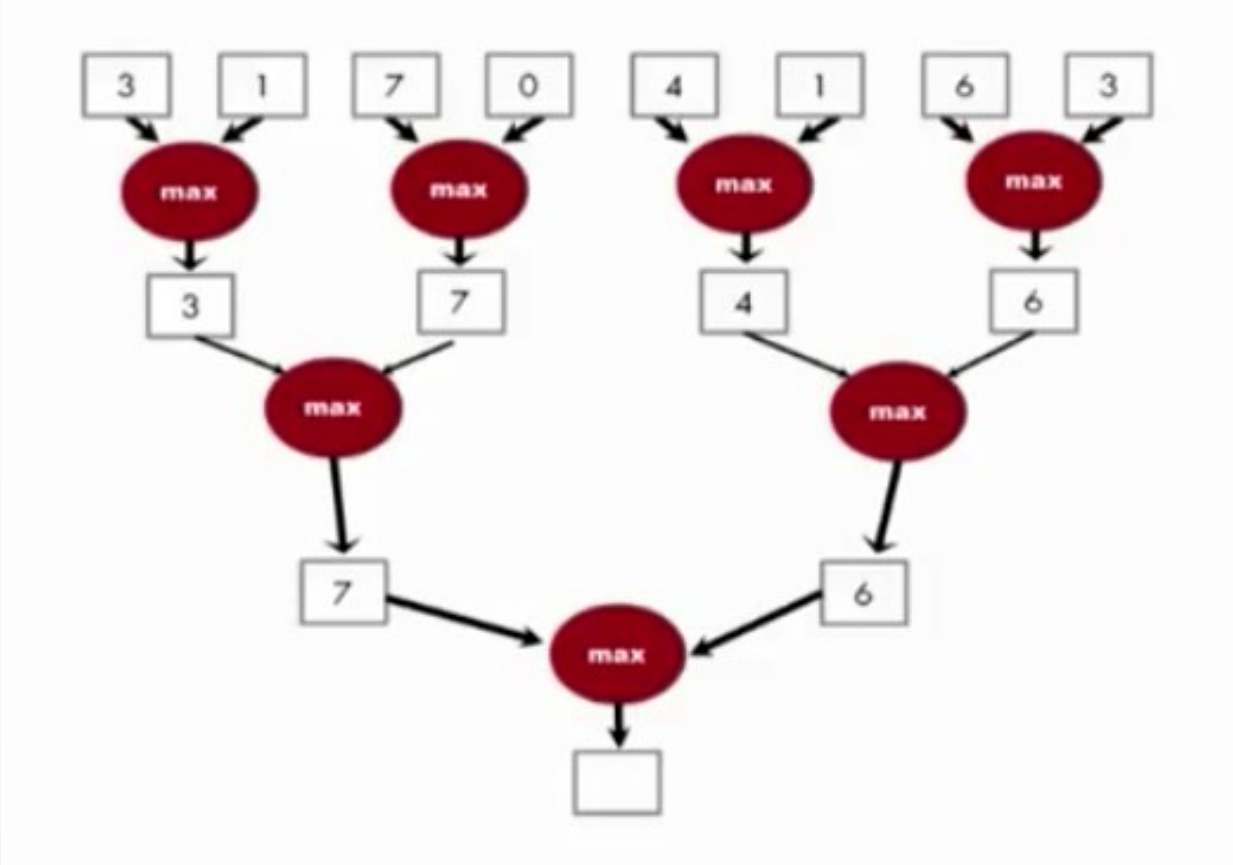


Initialize the result as an identity value for the reduction operation

- Smallest possible value for max reduction
- Largest possible value for min reduction
- 0 for sum reduction
- 1 for product reduction

Parallel Reduction

A parallel reduction tree algorithm performs N-1 operations in log(N) steps



Parallel Reduction

Analysis

For N input values, the reduction tree performs:

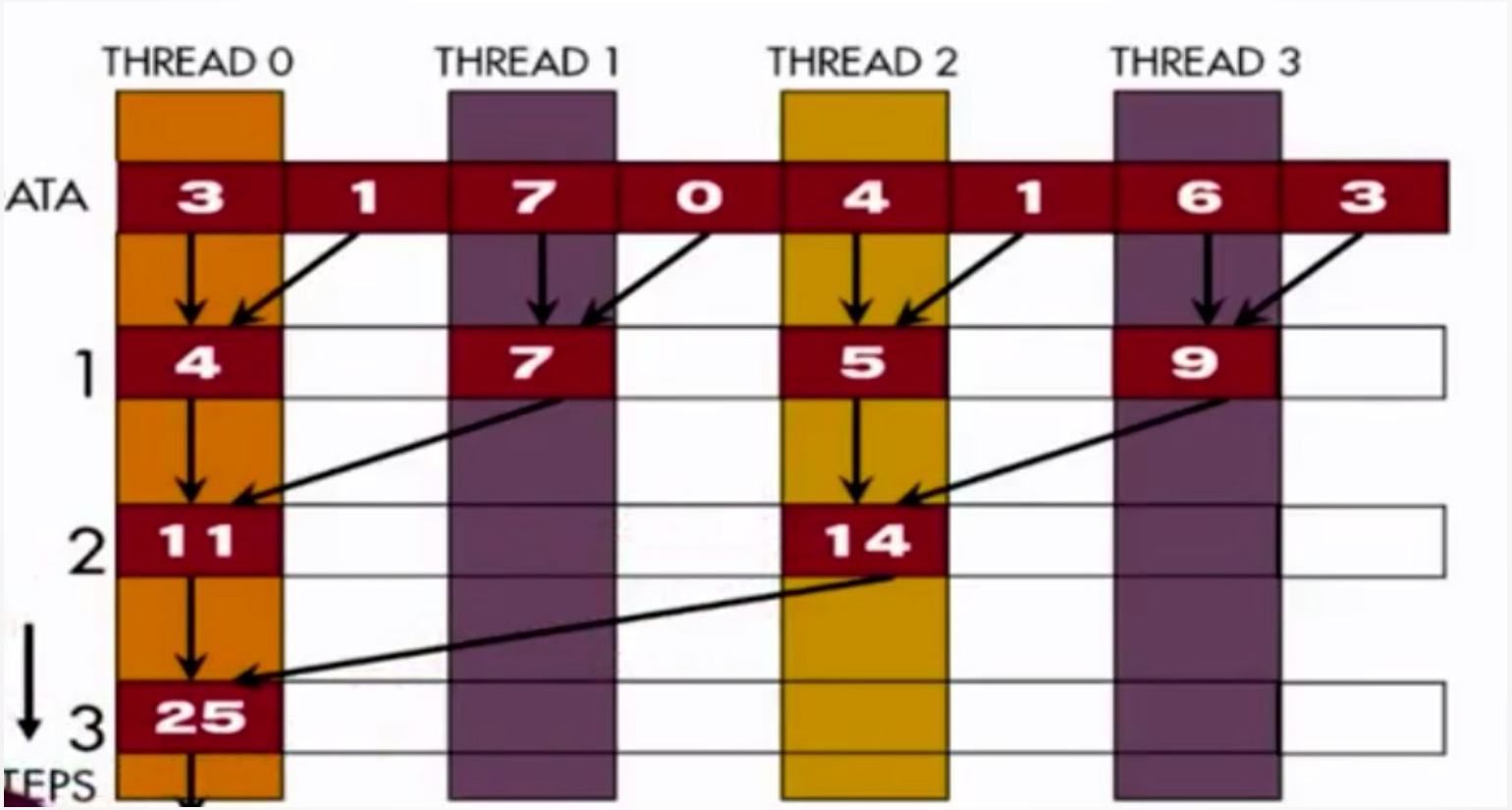
- $(1/2)N + (1/4)N + (1/8)N + \ldots (1/N) = (1- (1/N))N = N-1$ operations
- In Log (N) steps – 1,000,000 input values take 20 steps
  - Assuming that we have enough execution resources
- Average Parallelism (N-1)/Log(N))
  - For N = 1,000,000, average parallelism is 50,000
  - However, peak resource requirement is 500,000!

This is a work-efficient parallel algorithm

- The amount of work done is comparable to sequential
- Many parallel algorithms are not work efficient

Parallel Reduction

An exmaple



Parallel Reduction

An exmaple

- Each thread block takes 2* BlockDim input elements
- Each thread loads 2 elements into shared memory

```
__ shared __ float partialSum[2*BLOCK _ SIZE];

unsigned int t = threadIdx.x;
Unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[t] = input[start + t];
partialSum[blockDim+t] = input[start+ blockDim.x+t];
```

Parallel Reduction

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

Why do we need syncthreads()?

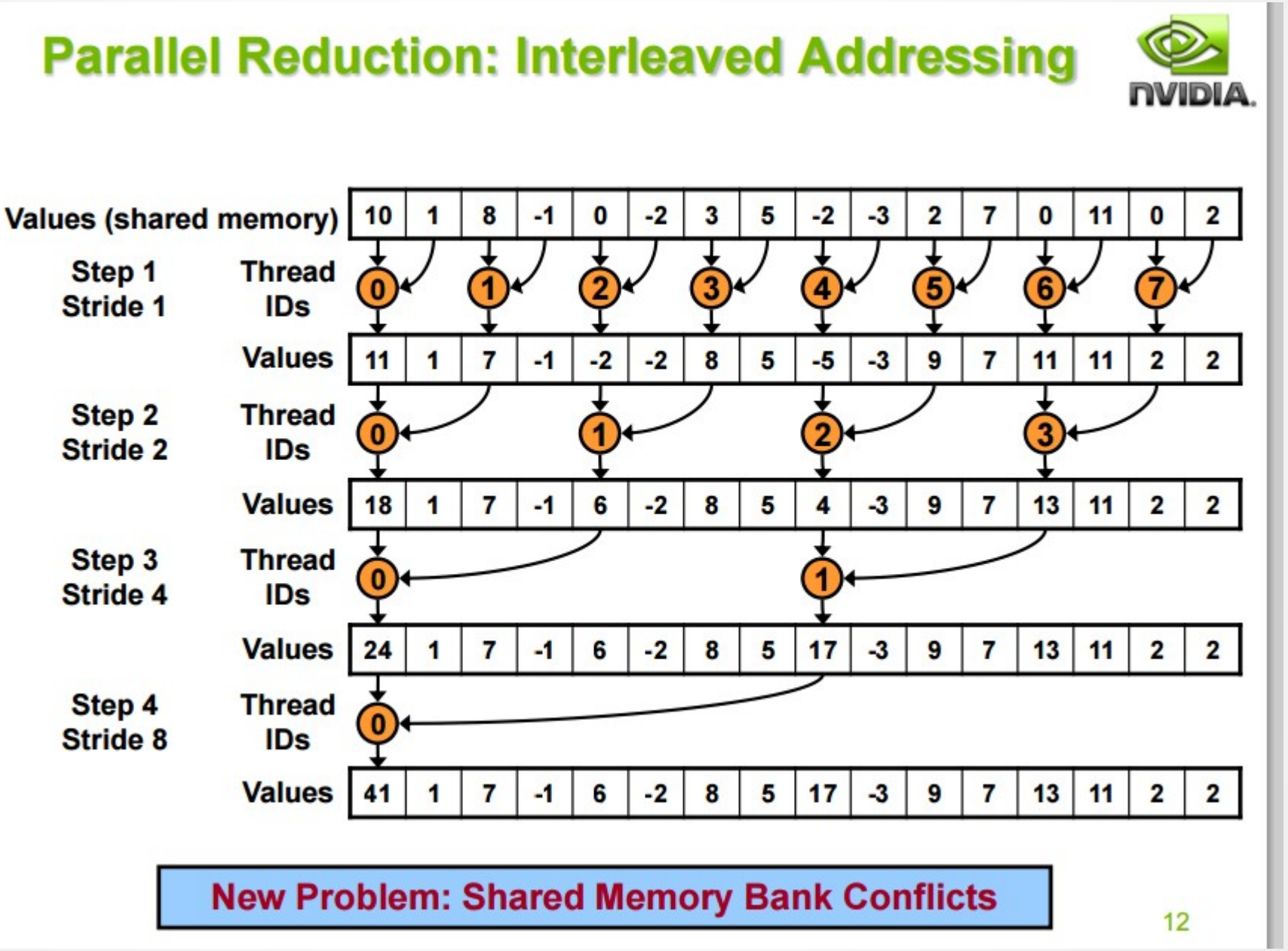
Thread 0 in each thread block write the sum of the thread block in partialSum[0] into a vector indexed by the blockIdx.x

There can be a large number of such sums if the original vector is very large

- The host code may iterate and launch another kernel

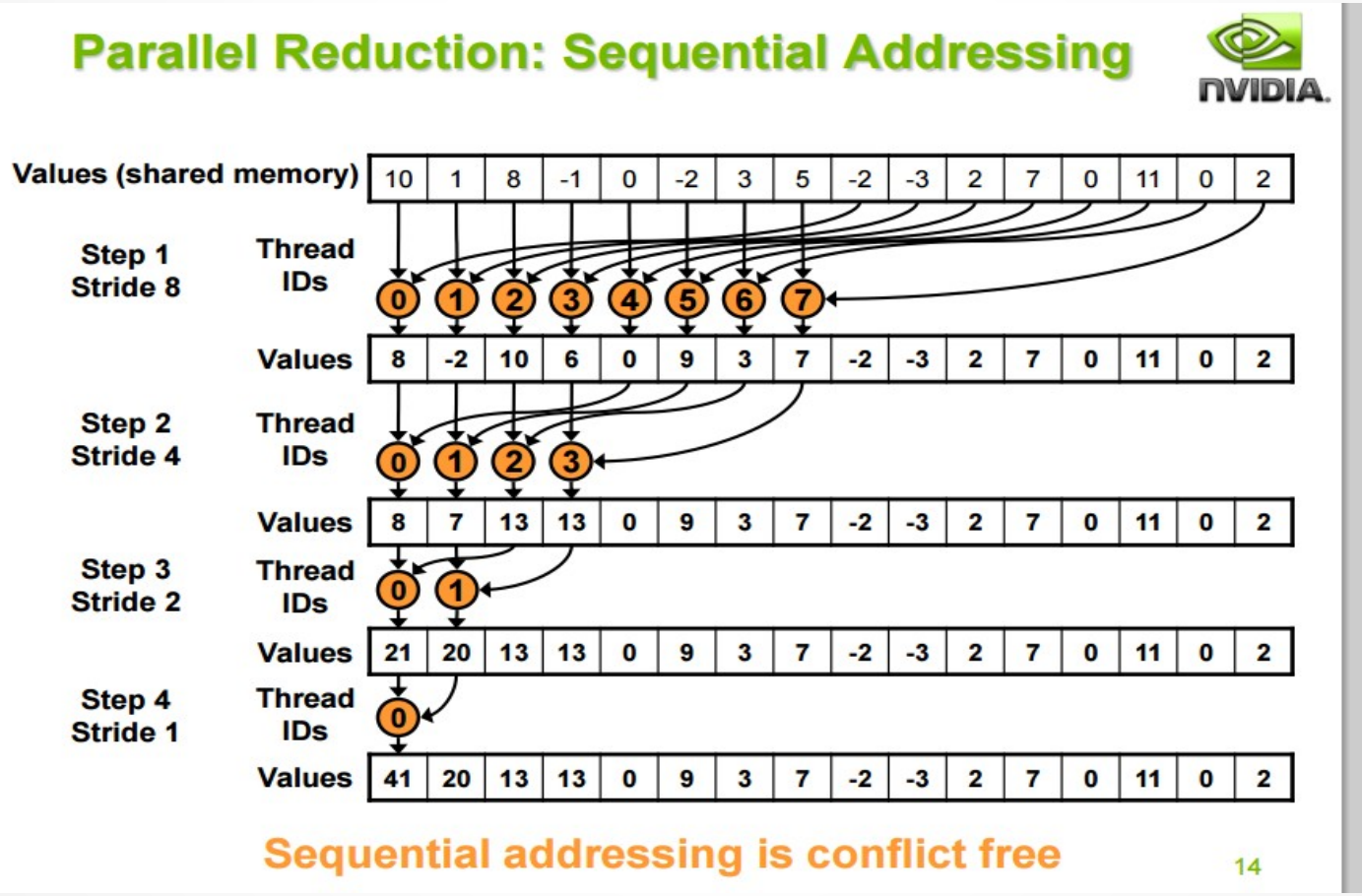
If there are only a small number of sums, the host can simply transfer the data back and add them together.

Parallel Reduction



Parallel Reduction

A better reduction ?



Parallel Reduction

Idle Threads

Problem:

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

Half of the threads are idle on first loop iteration!

This is wasteful…

Parallel Reduction

Reduction #4: First Add During Load

Halve the number of blocks, and replace single load:

```
// 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();
```

With two loads and first add of the reduction:

```
// perform first level of reduction,
// reading from global memory, writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

Parallel Reduction

Unrolling the Last Warp

- As reduction proceeds, # “active” threads decreases
  - When s <= 32, we have only one warp left
- Instructions are SIMD synchronous within a warp
- That means when s <= 32:
  - We don’t need to __syncthreads()
  - We don’t need “if (tid < s)” because it doesn’t save any work
- Let’s unroll the last 6 iterations of the inner loop

Parallel Reduction

Reduction #5: Unroll the Last Warp


```
__device__ void warpReduce(volatile int* sdata, int tid) {
    sdata[tid] += sdata[tid + 32];
    sdata[tid] += sdata[tid + 16];
    sdata[tid] += sdata[tid +  8];
    sdata[tid] += sdata[tid +  4];
    sdata[tid] += sdata[tid +  2];
    sdata[tid] += sdata[tid +  1];
}
```

IMPORTANT:
For this to be correct, we must use the "volatile" keyword!

```
// later…
for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
    if (tid < s)
        sdata[tid] += sdata[tid + s];
    __syncthreads();
}

if (tid < 32) warpReduce(sdata, tid);
```

Note: This saves useless work in *all* warps, not just the last one!
Without unrolling, all warps execute every iteration of the for loop and if statement

Parallel Reduction

```
template <unsigned int blockSize>
__device__ void warpReduce(volatile int *sdata, unsigned int tid) {
    if (blockSize >=  64) sdata[tid] += sdata[tid + 32];
    if (blockSize >=  32) sdata[tid] += sdata[tid + 16];
    if (blockSize >=  16) sdata[tid] += sdata[tid +  8];
    if (blockSize >=   8) sdata[tid] += sdata[tid +  4];
    if (blockSize >=   4) sdata[tid] += sdata[tid +  2];
    if (blockSize >=   2) sdata[tid] += sdata[tid +  1];
}
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n) {
    extern __shared__ int sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockSize*2) + tid;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = 0;

    while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize];  i += gridSize;  }
    __syncthreads();

    if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
    if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
    if (blockSize >= 128) { if (tid <   64) { sdata[tid] += sdata[tid +   64]; } __syncthreads(); }

    if (tid < 32) warpReduce(sdata, tid);
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

NVIDIA

Final Optimized Kernel

Parallel Reduction

```
template <unsigned int blockSize>
__device__ void warpReduce(volatile int *sdata, unsigned int tid) {
    if (blockSize >=  64) sdata[tid] += sdata[tid + 32];
    if (blockSize >=  32) sdata[tid] += sdata[tid + 16];
    if (blockSize >=  16) sdata[tid] += sdata[tid +  8];
    if (blockSize >=   8) sdata[tid] += sdata[tid +  4];
    if (blockSize >=   4) sdata[tid] += sdata[tid +  2];
    if (blockSize >=   2) sdata[tid] += sdata[tid +  1];
}
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n) {
    extern __shared__ int sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockSize*2) + tid;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = 0;

    while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize];  i += gridSize;  }
    __syncthreads();

    if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
    if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
    if (blockSize >= 128) { if (tid <   64) { sdata[tid] += sdata[tid +   64]; } __syncthreads(); }

    if (tid < 32) warpReduce(sdata, tid);
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

NVIDIA

Final Optimized Kernel

Homework 3

1. **Benchmark the performance between pinned and pageable memory** (including allocation, H2D/D2H, and free op). Justify when to use pinned memory. You should provide a figure with increasing transferred memory bytes on the x axis. (20 points)

2. Performance evaluation on parallel reduction by using different optimization schemes.

3. (optional) other way to improve reduction.