GPU Programming

Rupesh Nasre.

High-Performance Parallel Computing
June 2016

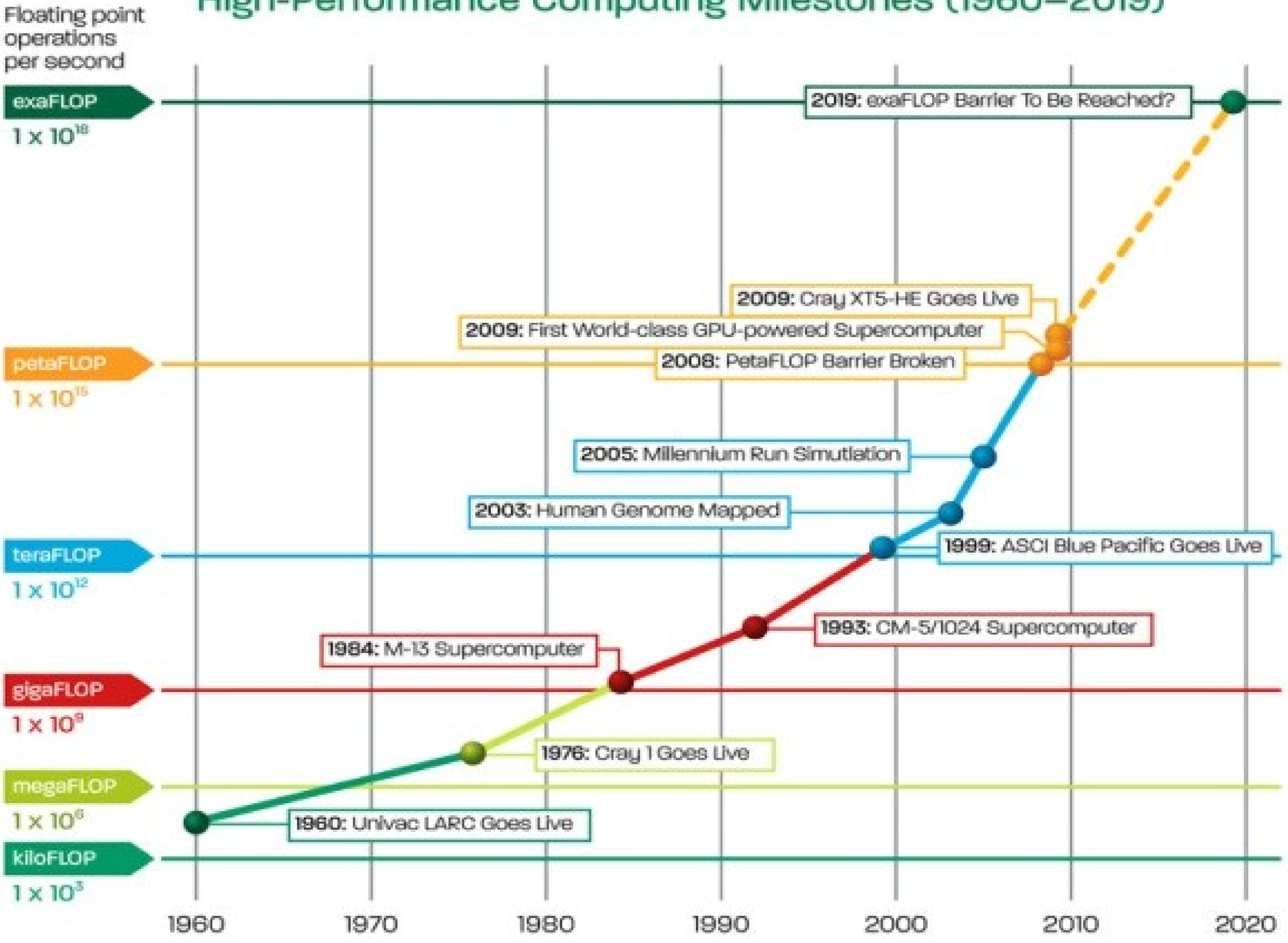
Outline

- **Basics**
  - History and Motivation
  - Simple Programs
  - Thread Synchronization
- **Optimizations**
  - GPU Memories
  - Thread Divergence
  - Memory Coalescing
  - ...
- **Case Studies**
  - Image Processing
  - Graph Algorithms

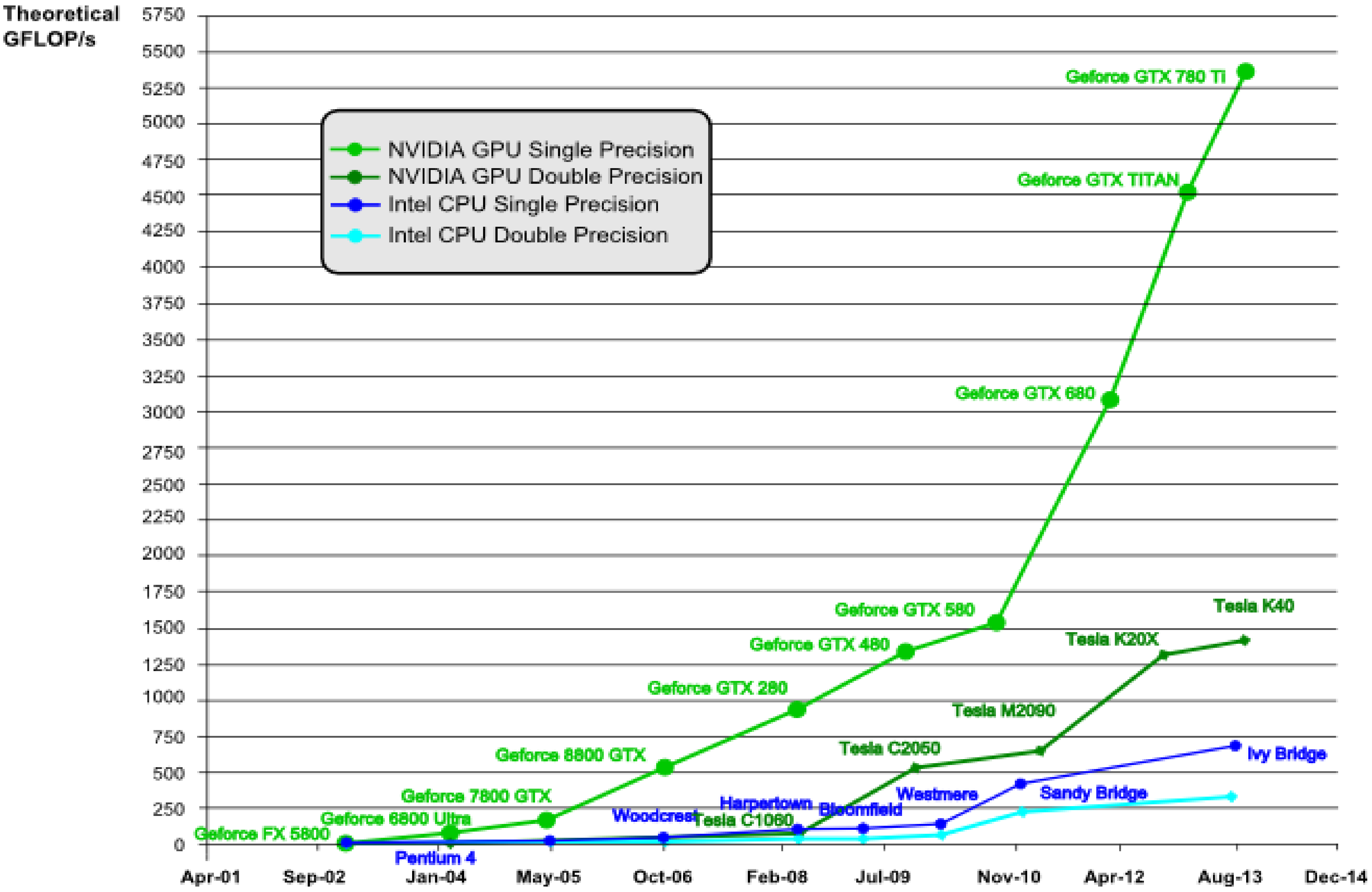
Some images are taken from NVIDIA CUDA Programming Guide.

AMD

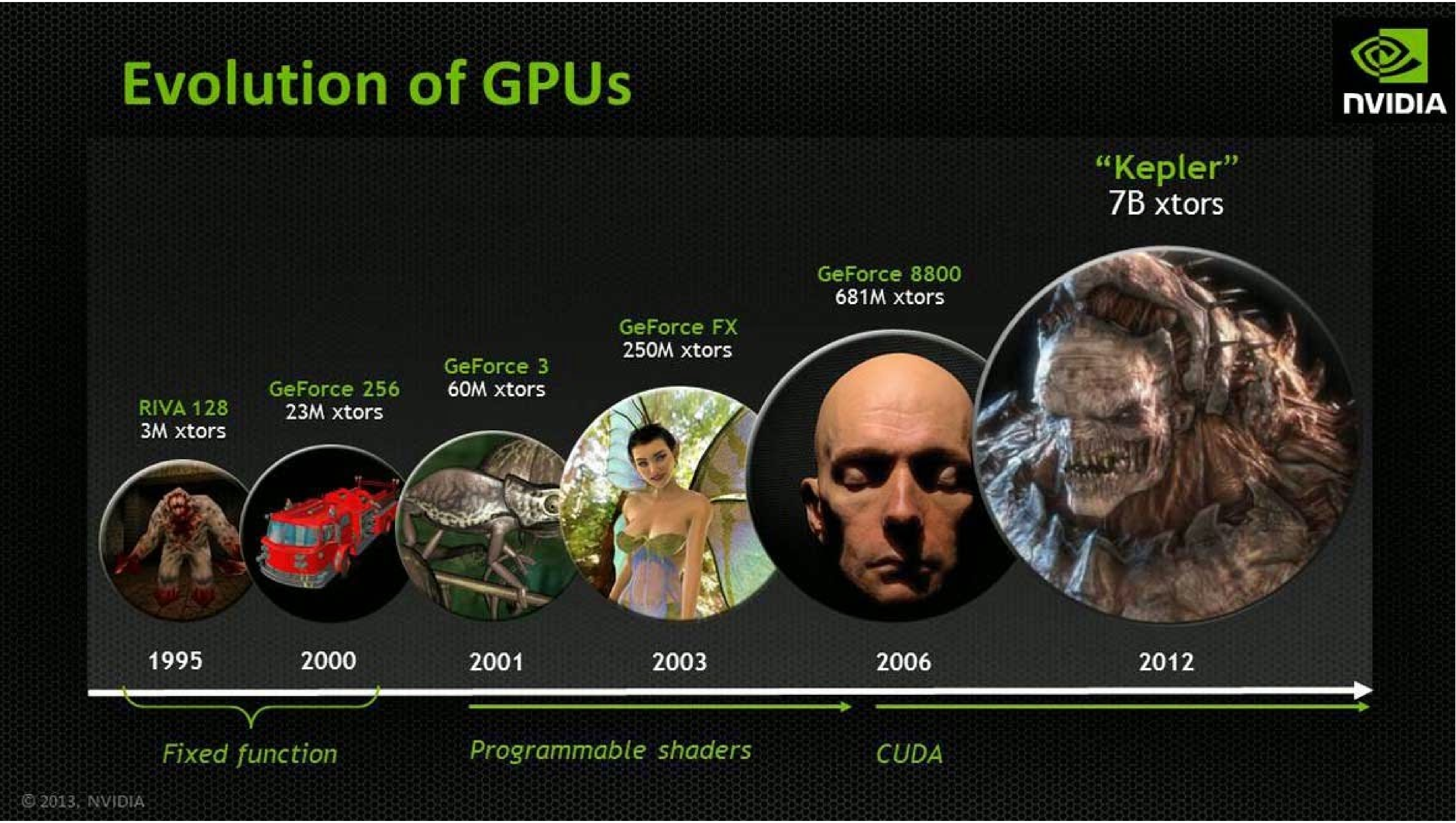
High-Performance Computing Milestones (1960–2019)



GPU-CPU Performance Comparison



Source: Thorsten Thormählen



Evolution of GPUs

NVIDIA

RIVA 128
3M xtors
1995

GeForce 256
23M xtors
2000

GeForce 3
60M xtors
2001

GeForce FX
250M xtors
2003

GeForce 8800
681M xtors
2006

"Kepler"
7B xtors
2012

Fixed function

Programmable shaders

CUDA

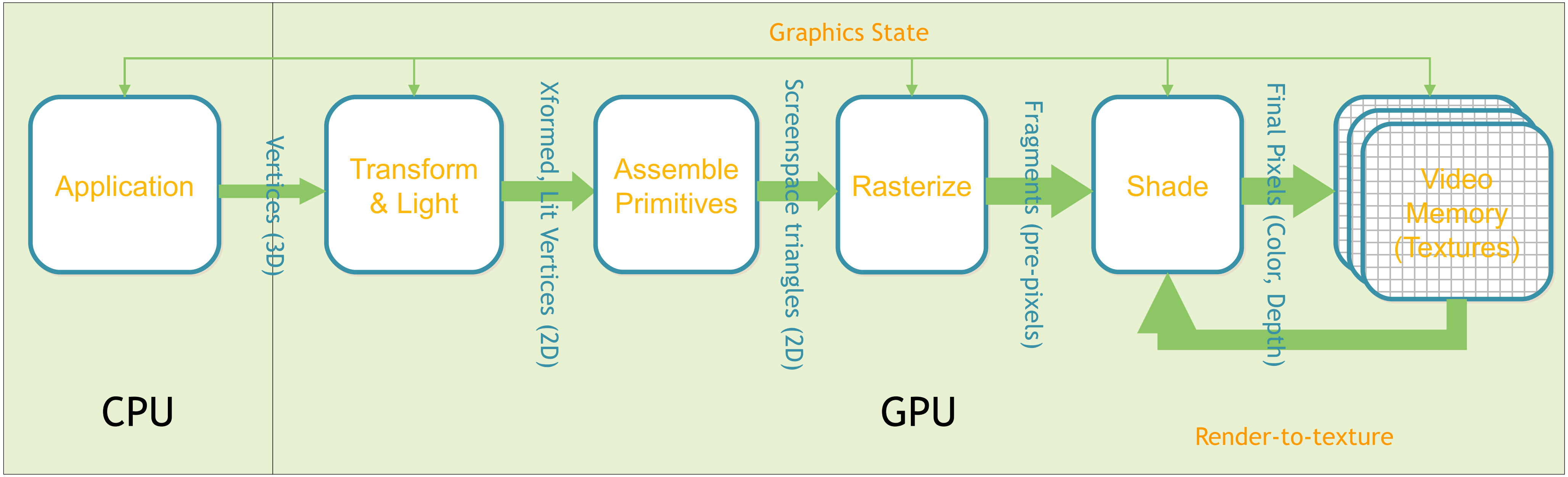
GPGPU: General Purpose Graphics Processing Unit

GPU Vendors

- NVIDIA
- AMD
- Intel
- QualComm
- ARM
- Broadcom
- Matrox Graphics
- Vivante
- Samsung
- ...

Earlier GPGPU Programming

GPGPU = General Purpose Graphics Processing Units.



- Applications: Protein Folding, Stock Options Pricing, SQL Queries, MRI Reconstruction.
- Required intimate knowledge of graphics API and GPU architecture.
- Program complexity: Problems expressed in terms of vertex coordinates, textures and shaders programs.
- Random memory reads/writes not supported.
- Lack of double precision support.

Kepler Configuration

Feature	K80	K40
# of SMX Units	26 (13 per GPU)	15
# of CUDA Cores	4992 (2496 per GPU)	2880
Memory Clock	2500 MHz	3004 MHz
GPU Base Clock	560 MHz	745 MHz
GPU Boost Support	Yes – Dynamic	Yes – Static
GPU Boost Clocks	23 levels between 562 MHz and 875 MHz	810 MHz 875 MHz
Architecture features	Dynamic Parallelism, Hyper-Q	
Compute Capability	3.7	3.5
Wattage (TDP)	300W (plus Zero Power Idle)	235W
Onboard GDDR5 Memory	24 GB	12 GB

rn-gpu machine:/usr/local/cuda/NVIDIA_CUDA-6.5_Samples/1_Utilities/deviceQuery/deviceQuery

Homework: Find out what is the GPU type on rn-gpu machine.

Configurations

In your login on rn-gpu, **setup the environment:**

```
$ export PATH=$PATH:/usr/local/cuda/bin:
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda/lib64:
```

You can also add the lines to .bashrc.

To **create**:

```
$ vi file.cu
```

To **compile**:

```
$ nvcc file.cu
```

This should create a.out in the current directory.

To **execute**:

```
$ a.out
```

GPU Configuration: Fermi

- Third Generation Streaming Multiprocessor (SM)
  - 32 CUDA cores per SM, 4x over GT200
  - 8x the peak double precision floating point performance over GT200
  - Dual Warp Scheduler simultaneously schedules and dispatches instructions from two independent warps
  - 64 KB of RAM with a configurable partitioning of shared memory and L1 cache
- Second Generation Parallel Thread Execution ISA
  - Full C++ Support
  - Optimized for OpenCL and DirectCompute
  - Full IEEE 754-2008 32-bit and 64-bit precision
  - Full 32-bit integer path with 64-bit extensions
  - Memory access instructions to support transition to 64-bit addressing
  - Improved Performance through Predication

- Improved Memory Subsystem
  - NVIDIA Parallel DataCacheTM hierarchy with Configurable L1 and Unified L2 Caches
  - First GPU with ECC memory support
  - Greatly improved atomic memory operation performance
- NVIDIA GigaThreadTM Engine
  - 10x faster application context switching
  - Concurrent kernel execution
  - Out of Order thread block execution
  - Dual overlapped memory transfer engines

CUDA, in a nutshell

- Compute Unified Device Architecture. It is a hardware and software architecture.
- Enables NVIDIA GPUs to execute programs written with C, C++, Fortran, OpenCL, and other languages.
- A CUDA program calls parallel kernels. A kernel executes in parallel across a set of parallel threads.
- The programmer or compiler organizes these threads in thread blocks and grids of thread blocks.
- The GPU instantiates a kernel program on a grid of parallel thread blocks.
- Each thread within a thread block executes an instance of the kernel, and has a thread ID within its thread block, program counter, registers, per-thread private memory, inputs, and output results.
- A thread block is a set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory.
- A grid is an array of thread blocks that execute the same kernel, read inputs from global memory, and write results to global memory.
- Each thread has a per-thread private memory space used for register spills, function calls, and C automatic array variables.
- Each thread block has a per-block shared memory space used for inter-thread communication, data sharing, and result sharing in parallel algorithms.

Hello World.

```
#include <stdio.h>

int main() {

    printf("Hello World.\n");

    return 0;

}
```

Compile: nvcc hello.cu
Run: a.out

GPU Hello World.

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel() {
    printf(“Hello World.\n”);
}
int main() {
    dkernel<<<1, 1>>>();
    return 0;
}
```

Kernel

Kernel Launch

Compile: nvcc hello.cu
Run: ./a.out
– *No output.* --

GPU Hello World.

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel() {
    printf(“Hello World.\n”);
}
int main() {
    dkernel<<<1, 1>>>();
    cudaThreadSynchronize();
    return 0;
}
```

```
Compile: nvcc hello.cu
Run: ./a.out
Hello World.
```

Takeaway

CPU function and GPU kernel run asynchronously.

GPU Hello World in Parallel.

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel() {
    printf("Hello World.\n");
}
int main() {
    dkernel<<<1, 32>>>();
    cudaThreadSynchronize();
    return 0;
}
```

```
Compile: nvcc hello.cu
Run: ./a.out
Hello World.
Hello World.
...
```

32 times

GPU Hello World with a Global.

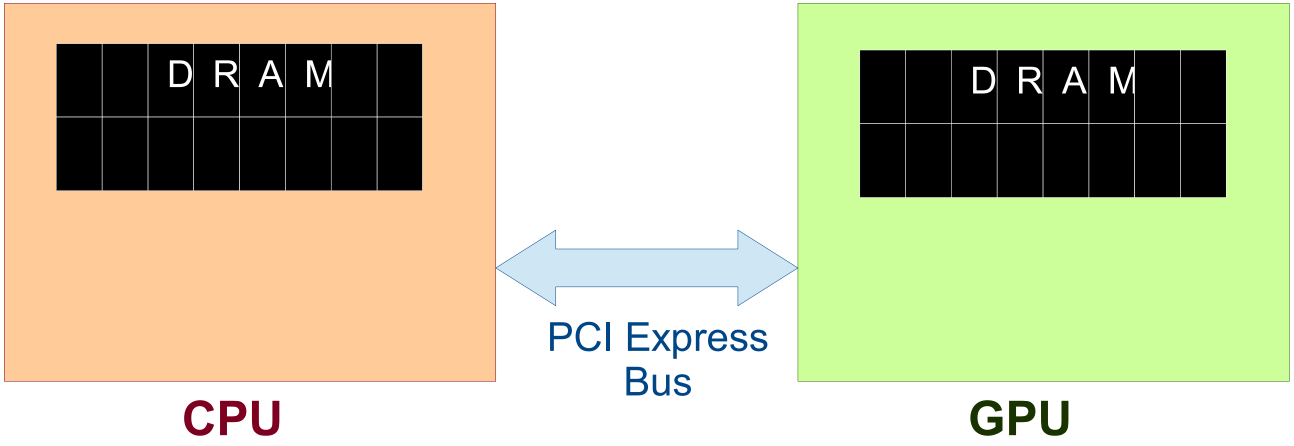
```
#include <stdio.h>
#include <cuda.h>
const char *msg = "Hello World.\n";
__global__ void dkernel() {
    printf(msg);
}
int main() {
    dkernel<<<1, 32>>>();
    cudaThreadSynchronize();
    return 0;
```

```
Compile: nvcc hello.cu
error: identifier "msg" is undefined in device code
```

Takeaway

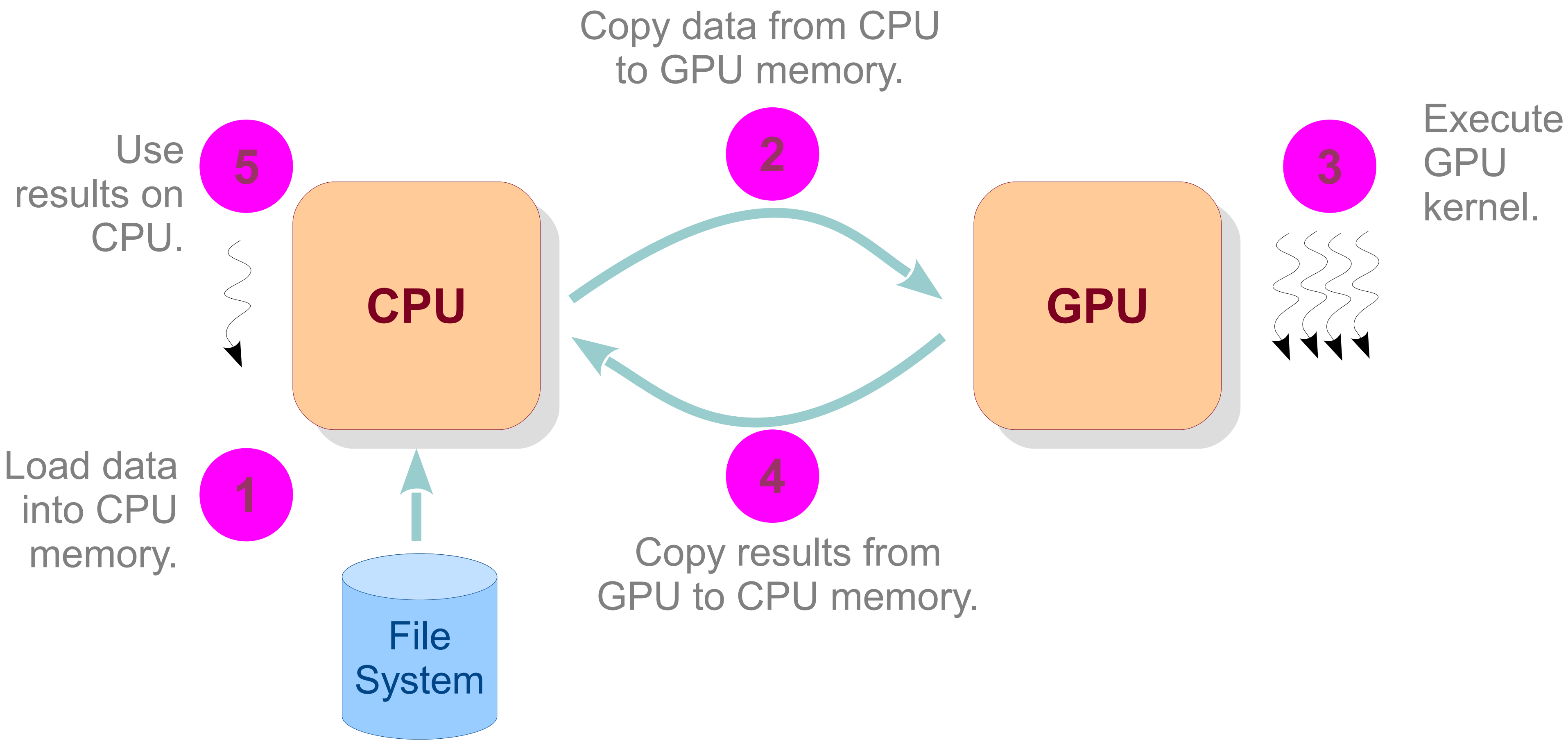
CPU and GPU memories are separate (for discrete GPUs).

Separate Memories



- CPU and its associated (discrete) GPUs have separate physical memory (RAM).
- A variable in CPU memory cannot be accessed directly in a GPU kernel.
- A programmer needs to maintain copies of variables.
- It is programmer's responsibility to keep them in sync.

Typical CUDA Program Flow



Copy data from CPU to GPU memory.

Use results on CPU.

Execute GPU kernel.

Load data into CPU memory.

Copy results from GPU to CPU memory.

Typical CUDA Program Flow

1 Load data into CPU memory.

- fread / rand

2 Copy data from CPU to GPU memory.

- cudaMemcpy(..., cudaMemcpyHostToDevice)

3 Call GPU kernel.

- mykernel<<<x, y>>>(...)

4 Copy results from GPU to CPU memory.

- cudaMemcpy(..., cudaMemcpyDeviceToHost)

5 Use results on CPU.

Typical CUDA Program Flow

2 Copy data from CPU to GPU memory.

- cudaMemcpy(..., cudaMemcpyHostToDevice)

This means we need two copies of the same variable – one on CPU another on GPU.

e.g., int *cpuarr, *gpuarr;

Matrix cpumat, gpumat;

Graph cpug, gpug;

CPU-GPU Communication

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel(char *arr, int arrlen) {
        unsigned id = threadIdx.x;
        if (id < arrlen) {
                ++arr[id];
        }
}
```

```
int main() {
     char cpuarr[] = "Gdkkn\x1fVnqkc-",
            *gpuarr;

     cudaMalloc(&gpuarr, sizeof(char) * (1 + strlen(cpuarr)));
     cudaMemcpy(gpuarr, cpuarr, sizeof(char) * (1 + strlen(cpuarr)), cudaMemcpyHostToDevice);
     dkernel<<<1, 32>>>(gpuarr, strlen(cpuarr));
     cudaThreadSynchronize();       // unnecessary.
     cudaMemcpy(cpuarr, gpuarr, sizeof(char) * (1 + strlen(cpuarr)), cudaMemcpyDeviceToHost);
     printf(cpuarr);

     return 0;
}
```

Classwork

1. Write a CUDA program to initialize an array of size 32 to all zeros in parallel.
2. Change the array size to 1024.
3. Create another kernel that adds *i* to *array[i]*.
4. Change the array size to 8000.
5. Check if answer to problem 3 still works.

Thread Organization

- A kernel is launched as a grid of threads.
- A grid is a 3D array of thread-blocks (gridDim.x, gridDim.y and gridDim.z).
  - Thus, each block has blockIdx.x, .y, .z.
- A thread-block is a 3D array of threads (blockDim.x, .y, .z).
  - Thus, each thread has threadIdx.x, .y, .z.

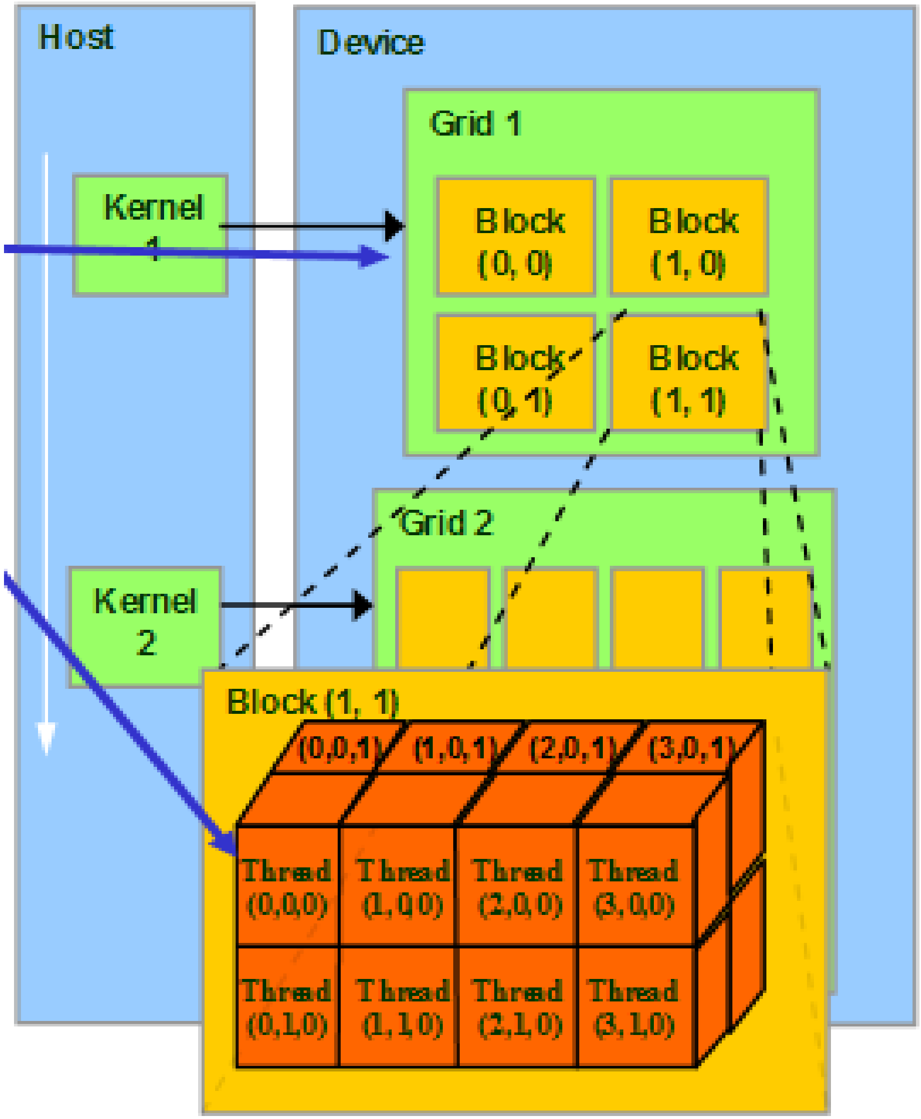
Grids, Blocks, Threads

- Each thread uses IDs to decide what data to work on
  - Block ID: 1D, 2D, or 3D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …
- Typical configuration:
  - 1-5 blocks per SM
  - 128-1024 threads per block.
  - Total 2K-100K threads.
  - You can launch a kernel with millions of threads.

CPU GPU

Grid with 2x2 blocks

A single thread in 4x2x2 threads



Accessing Dimensions

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel() {
        if (threadIdx.x == 0 && blockIdx.x == 0 &&
            threadIdx.y == 0 && blockIdx.y == 0 &&
            threadIdx.z == 0 && blockIdx.z == 0) {
                printf("%d %d %d %d %d %d.\n", gridDim.x, gridDim.y, gridDim.z,
                                                blockDim.x, blockDim.y, blockDim.z);
        }
}
int main() {
        dim3 grid(2, 3, 4);
        dim3 block(5, 6, 7);
        dkernel<<<grid, block>>>();
        cudaThreadSynchronize();
        return 0;
}
```

How many times the kernel printf gets executed when the *if* condition is changed to *if (threadIdx.x == 0)* ?

Number of threads launched = 2 * 3 * 4 * 5 * 6 * 7.
Number of threads in a thread-block = 5 * 6 * 7.
Number of thread-blocks in the grid = 2 * 3 * 4.

ThreadId in x dimension is in [0..5).
BlockId in y dimension is in [0..3).

2D

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel(unsigned *matrix) {
        unsigned id = threadIdx.x * blockDim.y + threadIdx.y;
        matrix[id] = id;
}
#define N       5
#define M       6
int main() {
        dim3 block(N, M, 1);
        unsigned *matrix, *hmatrix;

        cudaMalloc(&matrix, N * M * sizeof(unsigned));
        hmatrix = (unsigned *)malloc(N * M * sizeof(unsigned));

        dkernel<<<1, block>>>(matrix);
        cudaMemcpy(hmatrix, matrix, N * M * sizeof(unsigned), cudaMemcpyDeviceToHost);

        for (unsigned ii = 0; ii < N; ++ii) {
                for (unsigned jj = 0; jj < M; ++jj) {
                        printf("%2d ", hmatrix[ii * M + jj]);
                }
                printf("\n");
        }
    return 0;
}
```

```
$ a.out
 0  1  2  3  4  5
 6  7  8  9 10 11
12 13 14 15 16 17
18 19 20 21 22 23
24 25 26 27 28 29
```

1D

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel(unsigned *matrix) {
        unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
        matrix[id] = id;
}
#define N       5
#define M       6
int main() {
        unsigned *matrix, *hmatrix;

        cudaMalloc(&matrix, N * M * sizeof(unsigned));
        hmatrix = (unsigned *)malloc(N * M * sizeof(unsigned));

        dkernel<<<N, M>>>(matrix);
        cudaMemcpy(hmatrix, matrix, N * M * sizeof(unsigned), cudaMemcpyDeviceToHost);

        for (unsigned ii = 0; ii < N; ++ii) {
                for (unsigned jj = 0; jj < M; ++jj) {
                        printf("%2d ", hmatrix[ii * M + jj]);
                }
                printf("\n");
        }
    return 0;
}
```

Takeaway

One can perform computation on a multi-dimensional data using a one-dimensional block.

If I want the launch configuration to be <<<2, X>>>, what is X?
The rest of the code should be intact.

Launch Configuration for Large Size

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel(unsigned *vector) {
        unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
        vector[id] = id;
}
#define BLOCKSIZE       1024
int main(int nn, char *str[]) {
        unsigned N = atoi(str[1]);
        unsigned *vector, *hvector;
        cudaMalloc(&vector, N * sizeof(unsigned));
        hvector = (unsigned *)malloc(N * sizeof(unsigned));

        unsigned nblocks = ceil(N / BLOCKSIZE);
        printf("nblocks = %d\n", nblocks);

        dkernel<<<nblocks, BLOCKSIZE>>>(vector);
        cudaMemcpy(hvector, vector, N * sizeof(unsigned), cudaMemcpyDeviceToHost);
        for (unsigned ii = 0; ii < N; ++ii) {
                printf("%4d ", hvector[ii]);
        }
        return 0;
}
```

Access out-of-bounds.

Find two issues with this code.

Needs floating point division.

Launch Configuration for Large Size

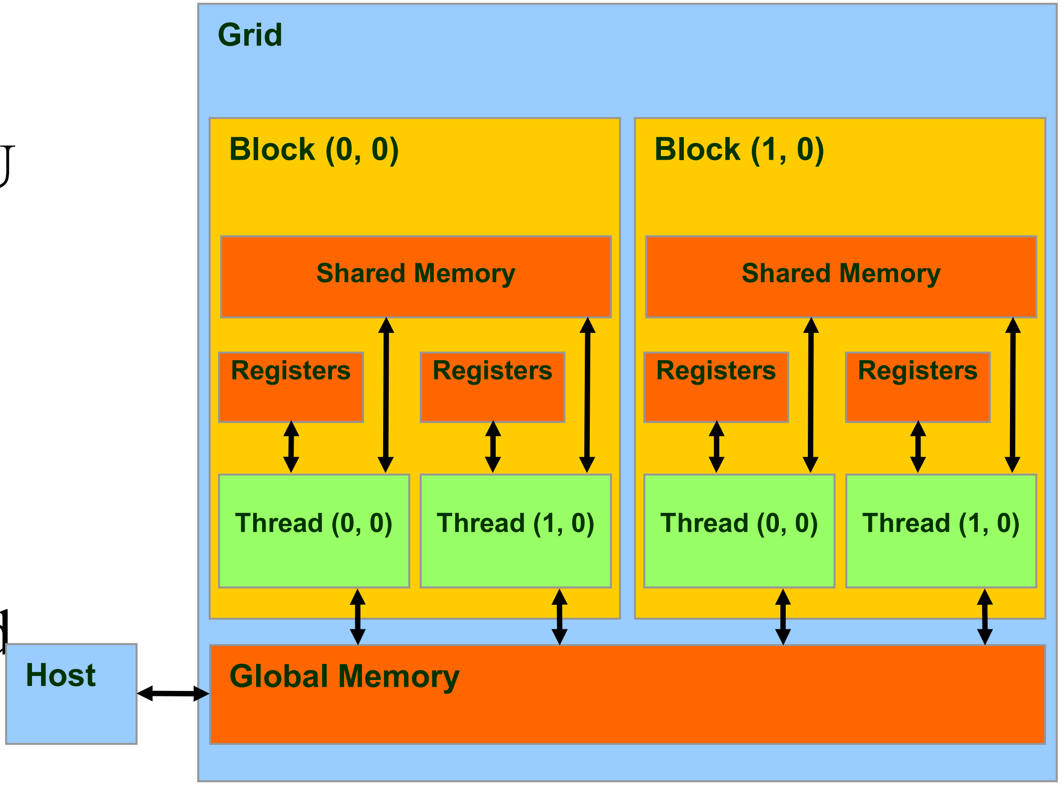
```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel(unsigned *vector, unsigned vectorsize) {
        unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
        if (id < vectorsize) vector[id] = id;
}
#define BLOCKSIZE       1024
int main(int nn, char *str[]) {
        unsigned N = atoi(str[1]);
        unsigned *vector, *hvector;
        cudaMalloc(&vector, N * sizeof(unsigned));
        hvector = (unsigned *)malloc(N * sizeof(unsigned));

        unsigned nblocks = ceil((float)N / BLOCKSIZE);
        printf("nblocks = %d\n", nblocks);

        dkernel<<<nblocks, BLOCKSIZE>>>(vector, N);
        cudaMemcpy(hvector, vector, N * sizeof(unsigned), cudaMemcpyDeviceToHost);
        for (unsigned ii = 0; ii < N; ++ii) {
                printf("%4d ", hvector[ii]);
        }
        return 0;
}
```

Classwork

- Read a sequence of integers from a file.
- Square each number.
- Read another sequence of integers from another file.
- Cube each number.
- Sum the two sequences element-wise, store in the third sequence.
- Print the computed sequence.

CUDA Memory Model Overview

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all GPU threads
  - Long latency access
- We will focus on global memory for now
  - There are also constant and texture memory.



CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void  KernelFunc()`	device	host
`__host__   float HostFunc()`	host	host

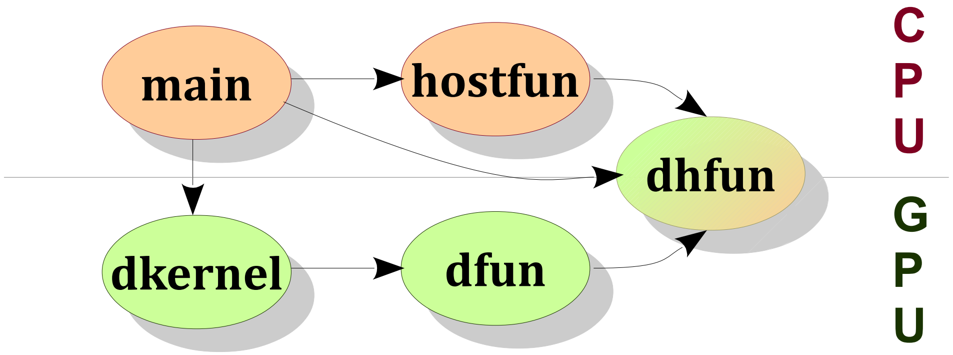
- __global__ defines a kernel. It must return void.
- A program may have several functions of each kind.
- The same function of any kind may be called multiple times.
- Host == CPU, Device == GPU.

Function Types (1/2)

```
#include <stdio.h>
#include <cuda.h>
__host__ __device__ void dhfun() {
        printf("I can run on both CPU and GPU.\n");
}
__device__ unsigned dfun(unsigned *vector, unsigned vectorsize, unsigned id) {
        if (id == 0) dhfun();
        if (id < vectorsize) {
                vector[id] = id;
                return 1;
        } else {
                return 0;
        }
}
__global__ void dkernel(unsigned *vector, unsigned vectorsize) {
        unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
        dfun(vector, vectorsize, id);
}
__host__ void hostfun() {
        printf("I am simply like another function running on CPU. Calling dhfun\n");
        dhfun();
}
```

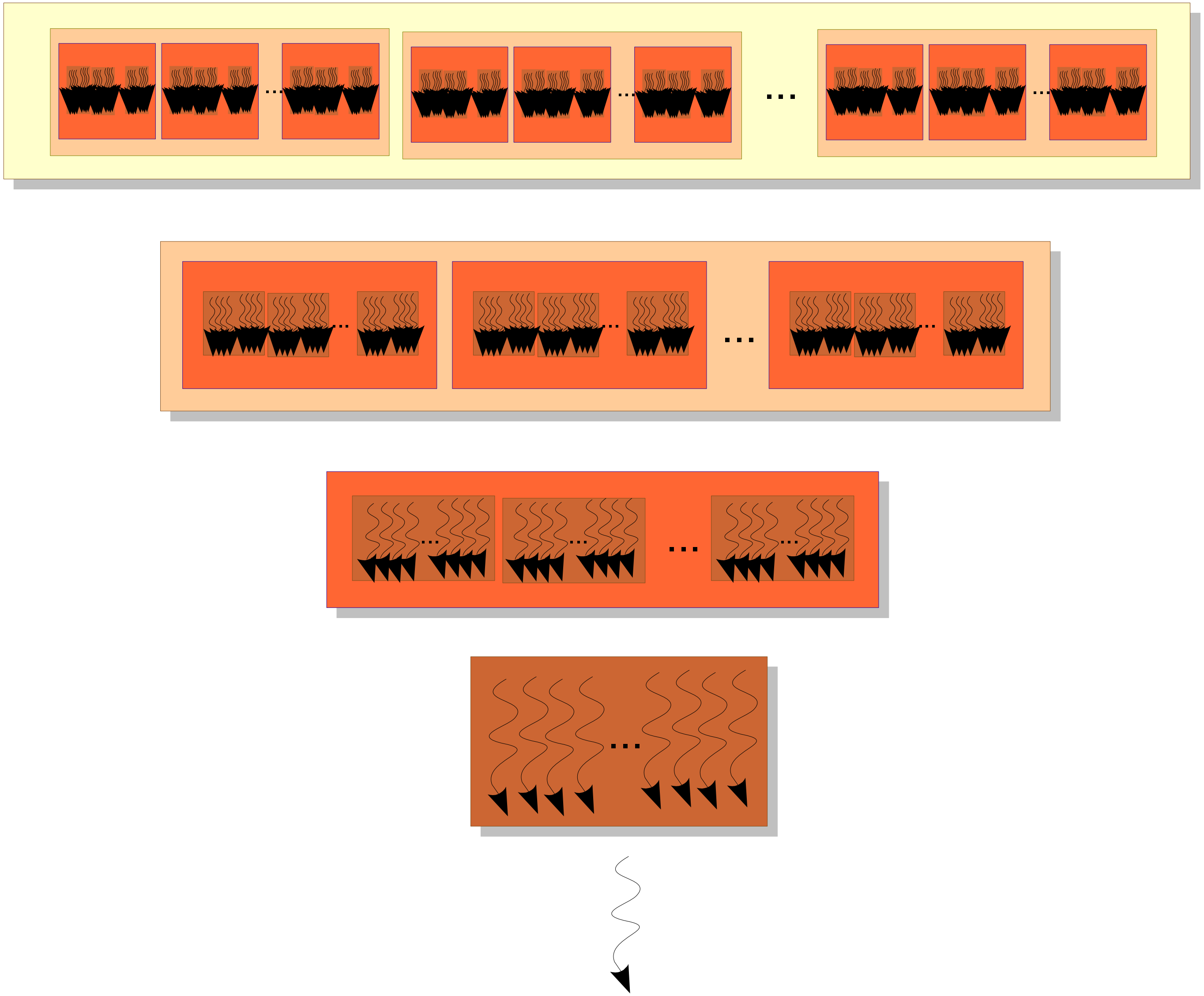
Function Types (2/2)

```
#define BLOCKSIZE       1024
int main(int nn, char *str[]) {
        unsigned N = atoi(str[1]);
        unsigned *vector, *hvector;
        cudaMalloc(&vector, N * sizeof(unsigned));
        hvector = (unsigned *)malloc(N * sizeof(unsigned));

        unsigned nblocks = ceil((float)N / BLOCKSIZE);
        printf("nblocks = %d\n", nblocks);

        dkernel<<<nblocks, BLOCKSIZE>>>(vector, N);
        cudaMemcpy(hvector, vector, N * sizeof(unsigned), cudaMemcpyDeviceToHost);
        for (unsigned ii = 0; ii < N; ++ii) {
                printf("%4d ", hvector[ii]);
        }
        printf("\n");
        hostfun();
        dhfun();
        return 0;
}
```



What are the other arrows possible in this diagram?

GPU Computation Hierarchy



GPU	Hundreds of thousands
Multi-processor	Tens of thousands
Block	1024
Warp	32
Thread	1

What is a Warp?

Source: Wikipedia

Warp

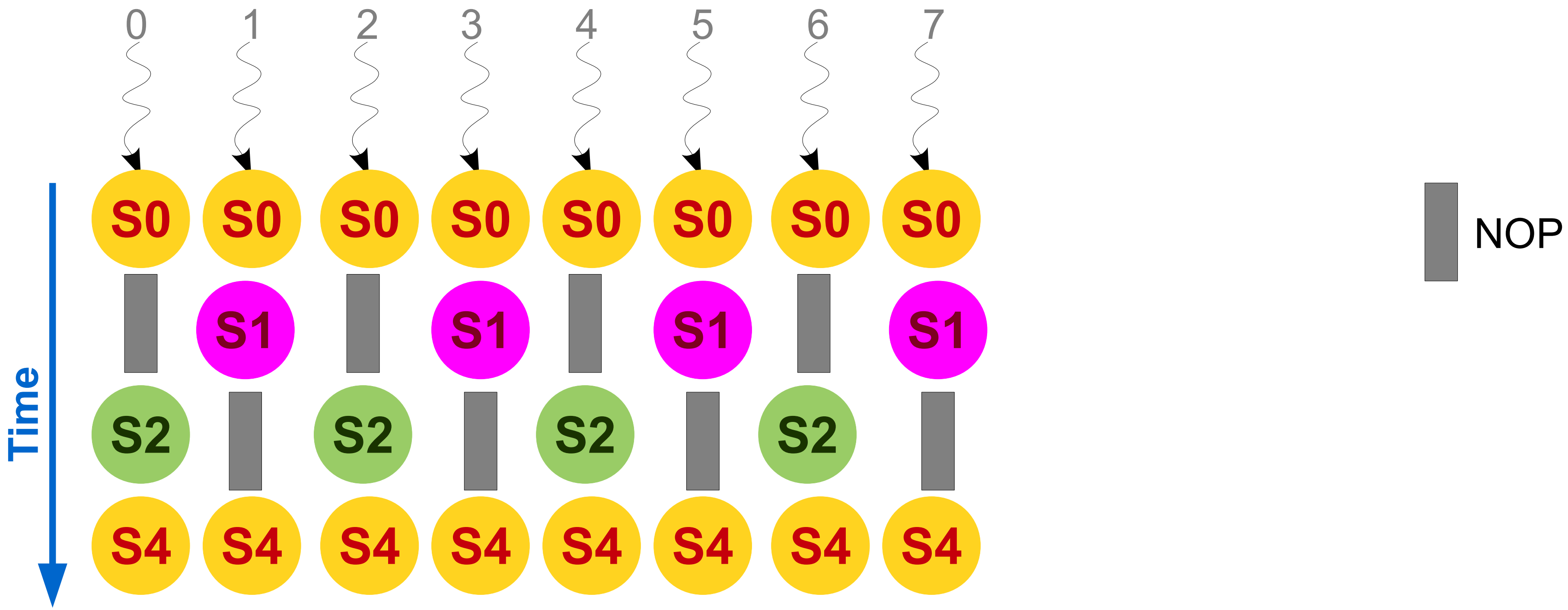
- A set of consecutive threads (currently 32) that execute in SIMD fashion.
- SIMD == Single Instruction Multiple Data
- Warp-threads are fully synchronized. There is an implicit barrier after each step / instruction.
- Memory coalescing is closedly related to warps.

Takeaway

It is a misconception that all threads in a GPU execute in lock-step. Lock-step execution is true for threads only within a warp.

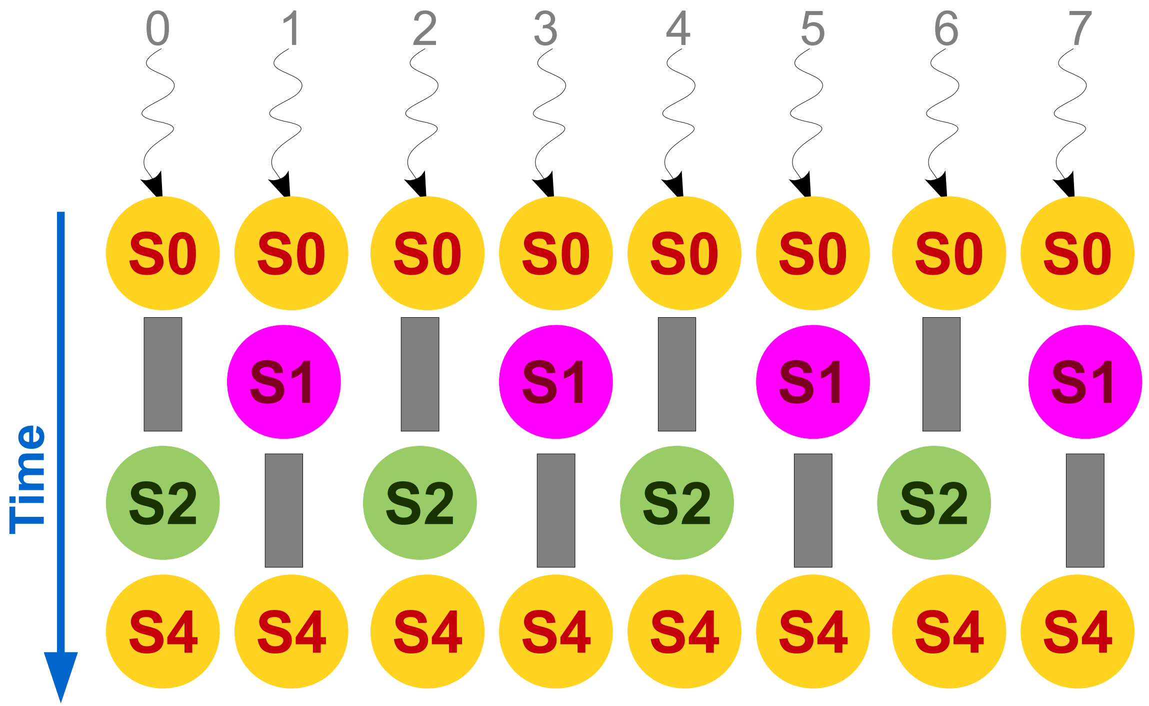
Warp with Conditions

```
__global__ void dkernel(unsigned *vector, unsigned vectorsize) {
        unsigned id = blockIdx.x * blockDim.x + threadIdx.x;  S0
        if (id % 2) vector[id] = id;  S1
        else vector[id] = vectorsize * vectorsize;  S2
        vector[id]++;  S4
}
```



Warp with Conditions

- When different warp-threads execute different instructions, threads are said to diverge.
- Hardware executes threads satisfying same condition together, ensuring that other threads execute a no-op.
- This adds sequentiality to the execution.
- This problem is termed as thread-divergence.



Thread-Divergence

```
__global__ void dkernel(unsigned *vector, unsigned vectorsize) {
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
    switch (id) {
    case 0: vector[id] = 0; break;
    case 1: vector[id] = vector[id]; break;
    case 2: vector[id] = vector[id - 2]; break;
    case 3: vector[id] = vector[id + 3]; break;
    case 4: vector[id] = 4 + 4 + vector[id]; break;
    case 5: vector[id] = 5 - vector[id]; break;
    case 6: vector[id] = vector[6]; break;
    case 7: vector[id] = 7 + 7; break;
    case 8: vector[id] = vector[id] + 8; break;
    case 9: vector[id] = vector[id] * 9; break;
}   }
```

Thread-Divergence

- Since thread-divergence makes execution sequential, conditions are evil in the kernel codes?

```
if (vectorsize < N) S1; else S2;
```

Condition but no divergence

- Then, conditions evaluating to different truth-values are evil?

```
if (id / 32) S1; else S2;
```

Different truth-values but no divergence

Takeaway

Conditions are not bad;
they evaluating to different truth-values is also not bad;
they evaluating to different truth-values for warp-threads is bad.

Classwork

- Rewrite the following program fragment to remove thread-divergence.

```
// assert(x == y || x == z);
if (x == y) x = z;
else x = y;
```

Locality

- Locality is important for performance on GPUs also.
- All threads in a thread-block access their L1 cache.
    - This cache on Kepler is 64 KB.
    - It can be configured as 48 KB L1 + 16 KB scratchpad or 16 KB L1 + 48 KB scratchpad.
- To exploit spatial locality, consecutive threads should access consecutive memory locations.

Matrix Squaring (version 1)

```
square<<<1, N>>>(matrix, result, N);    // N = 64
```

```
__global__ void square(unsigned *matrix,
                       unsigned *result,
                       unsigned matrixsize) {
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
    for (unsigned jj = 0; jj < matrixsize; ++jj) {
        for (unsigned kk = 0; kk < matrixsize; ++kk) {
            result[id * matrixsize + jj] +=
                        matrix[id * matrixsize + kk] *
                        matrix[kk * matrixsize + jj];
}   }   }
```

CPU time = 1.527 ms, GPU v1 time = 6.391 ms

Matrix Squaring (version 2)

```
square<<<N, N>>>(matrix, result, N);   // N = 64
```

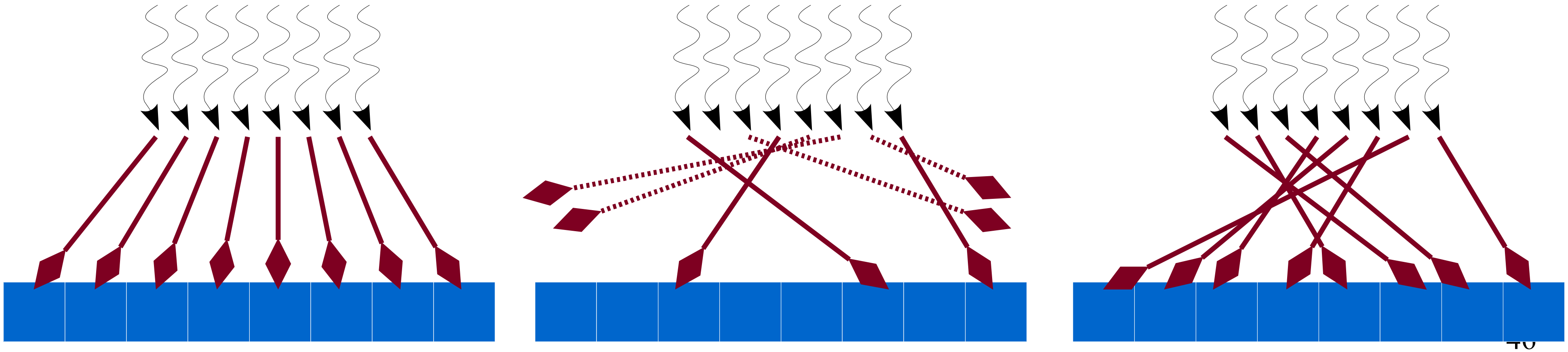
```
__global__ void square(unsigned *matrix,
                       unsigned *result,
                       unsigned matrixsize) {
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned ii = id / matrixsize;
    unsigned jj = id % matrixsize;
    for (unsigned kk = 0; kk < matrixsize; ++kk) {
      result[ii * matrixsize + jj] += matrix[ii * matrixsize + kk] *
                                      matrix[kk * matrixsize + jj];
}   }
```

Homework: What if you interchange ii and jj?

CPU time = 1.527 ms, GPU v1 time = 6.391 ms,
GPU v2 time = 0.1 ms

Memory Coalescing

- If *consecutive threads* access words from the same block of 32 words, their memory requests are clubbed into one.
  - That is, the memory requests are coalesced.
- This can be effectively achieved for regular programs (such as dense matrix operations).



Coalesced

Uncoalesced

Coalesced

Memory Coalescing

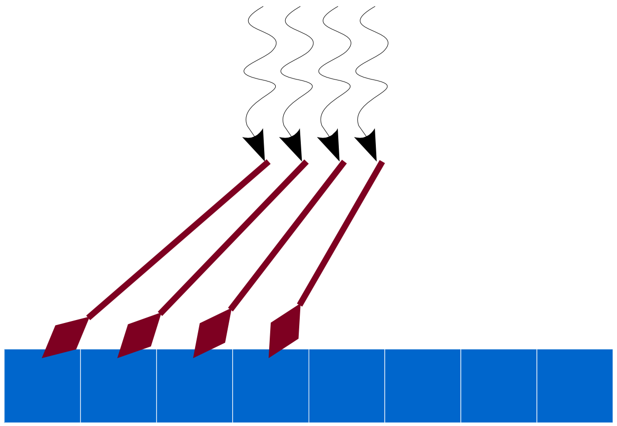
CPU

- Each thread should access consecutive elements of a chunk (strided).
- Array of Structures (AoS) has a better locality.

GPU

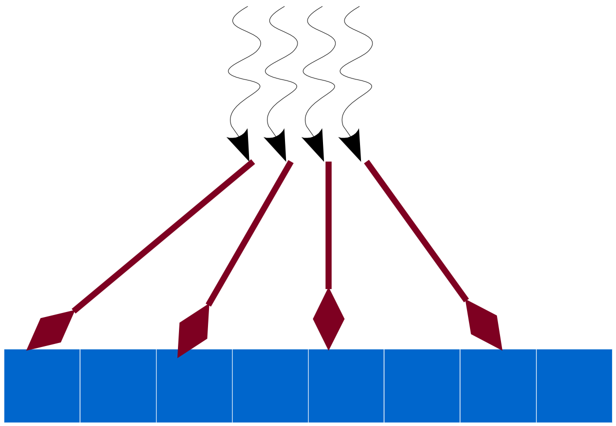
- A chunk should be accessed by consecutive threads (coalesced).
- Structures of Arrays (SoA) has a better performance.

... a[id] ...



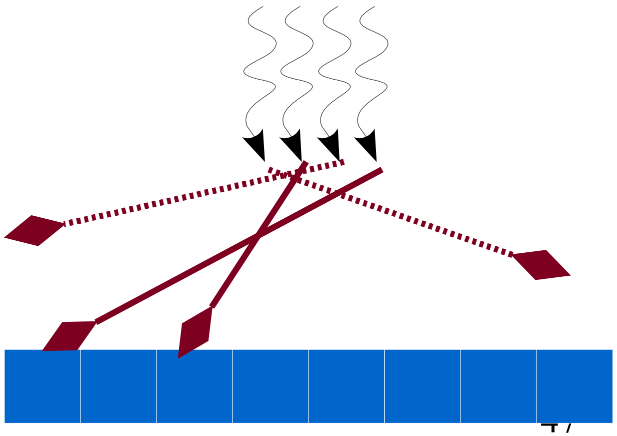
Coalesced

```
start = id * chunksize;
end = start + chunksize;
for (ii = start; ii < end; ++ii)
    ... a[ii] ...
```



Strided

... a[input[id]] ...



Random

AoS versus SoA

```
struct node {
        int a;
        double b;
        char c;
};
struct node allnodes[N];
```

```
struct node {
        int alla[N];
        double allb[N];
        char allc[N];
};
```

Expectation: When a thread accesses an attribute of a node, *it* also accesses *other attributes* of the *same node*.

Better locality (on CPU).

Expectation: When a thread accesses an attribute of a node, its *neighboring thread* accesses the *same attribute* of the *next node*.

Better coalescing (on GPU).

AoS versus SoA

```
struct node {
        int a;
        double b;
        char c;
};
struct node allnodes[N];
```

```
struct node {
        int alla[N];
        double allb[N];
        char allc[N];
};
```

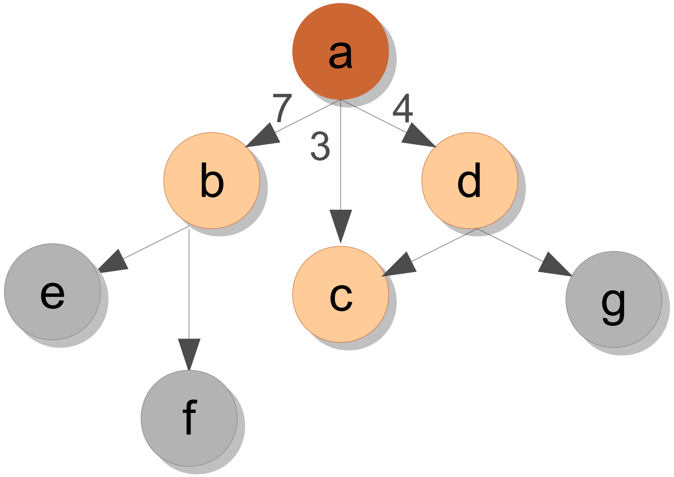
```
__global__ void
dkernelaos(struct nodeAOS
                    *allnodesAOS) {
        unsigned id = blockIdx.x *
          blockDim.x + threadIdx.x;

        allnodesAOS[id].a = id;
        allnodesAOS[id].b = 0.0;
        allnodesAOS[id].c = 'c';
}
```

```
__global__ void
dkernelsoa(int *a, double *b,
                    char *c) {
        unsigned id = blockIdx.x *
          blockDim.x + threadIdx.x;

        a[id] = id;
        b[id] = 0.0;
        c[id] = 'd';
}
```

AoS time: 0.000058 seconds
SoA time: 0.000021 seconds

Let's Compute the Shortest Paths

- You are given an input graph of India, and you want to compute the shortest path from Nagpur to every other city.
- Assume that you are given a GPU graph library and the associated routines.



```
__global__ void dsssp(Graph g, unsigned *dist) {
    unsigned id = ...
    for each n in g.allneighbors(id) {    // pseudo-code.
        unsigned altdist = dist[id] + weight(id, n);
        if (altdist < dist[n]) {
            dist[n] = altdist;
}   }   }
```

What is the error in this code?

Synchronization

- Atomics
- Barriers
- Control + data flow
- ...

atomics

- Atomics are primitive operations whose effects are visible either none or fully (never partially).
- Need hardware support.
- Several variants: atomicCAS, atomicMin, atomicAdd, ...
- Work with both global and shared memory.

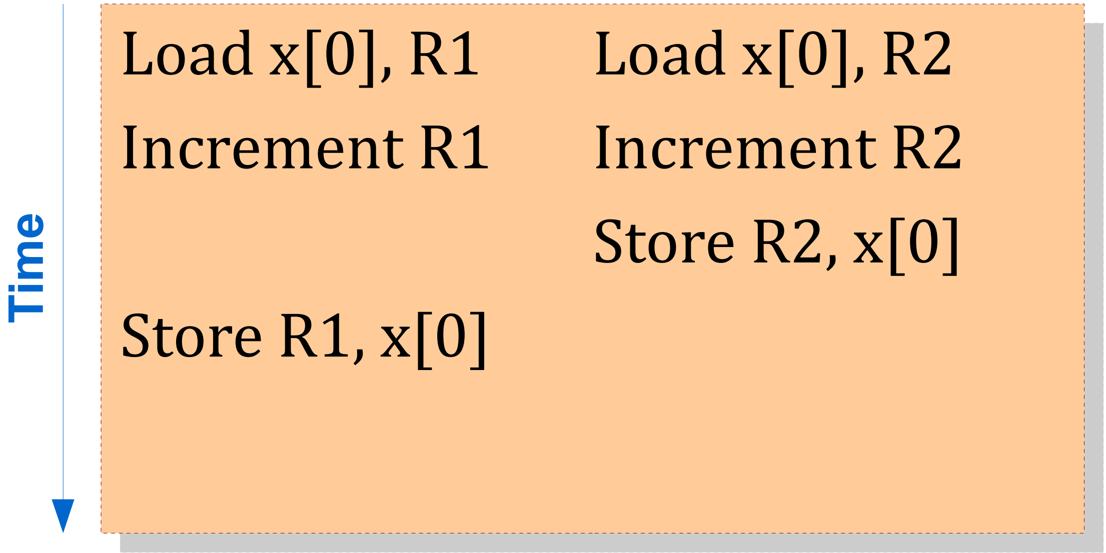
atomics

```
__global__ void dkernel(int *x) {
    ++x[0];
}
...
dkernel<<<1, 2>>>(x);
```

After dkernel completes, what is the value of x[0]?

++x[0] is equivalent to:

```
    Load x[0], R1
    Increment R1
    Store R1, x[0]
```



Load x[0], R1	Load x[0], R2
Increment R1	Increment R2
	Store R2, x[0]
Store R1, x[0]	

Final value stored in x[0] could be 1 (rather than 2).
What if x[0] is split into multiple instructions? What if there are more threads?

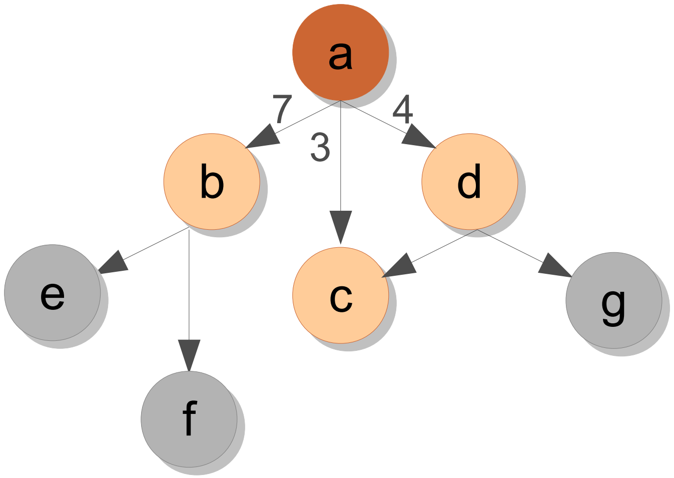
atomics

```
__global__ void dkernel(int *x) {
    ++x[0];
}
...
dkernel<<<1, 2>>>(x);
```

- Ensure all-or-none behavior.
  - e.g., atomicInc(&x[0], ...);
- **dkernel**<<<K1, K2>>> would ensure x[0] to be incremented by exactly K1*K2 – irrespective of the thread execution order.

Let's Compute the Shortest Paths

- You are given an input graph of India, and you want to compute the shortest path from Nagpur to every other city.
- Assume that you are given a GPU graph library and the associated routines.



```
__global__ void dsssp(Graph g, unsigned *dist) {
    unsigned id = ...
    for each n in g.allneighbors(id) {    // pseudo-code.
        unsigned altdist = dist[id] + weight(id, n);
        if (altdist < dist[n]) {
            ~~dist[n] = altdist;~~  atomicMin(&dist[n], altdist);
}   }   }
```

Classwork

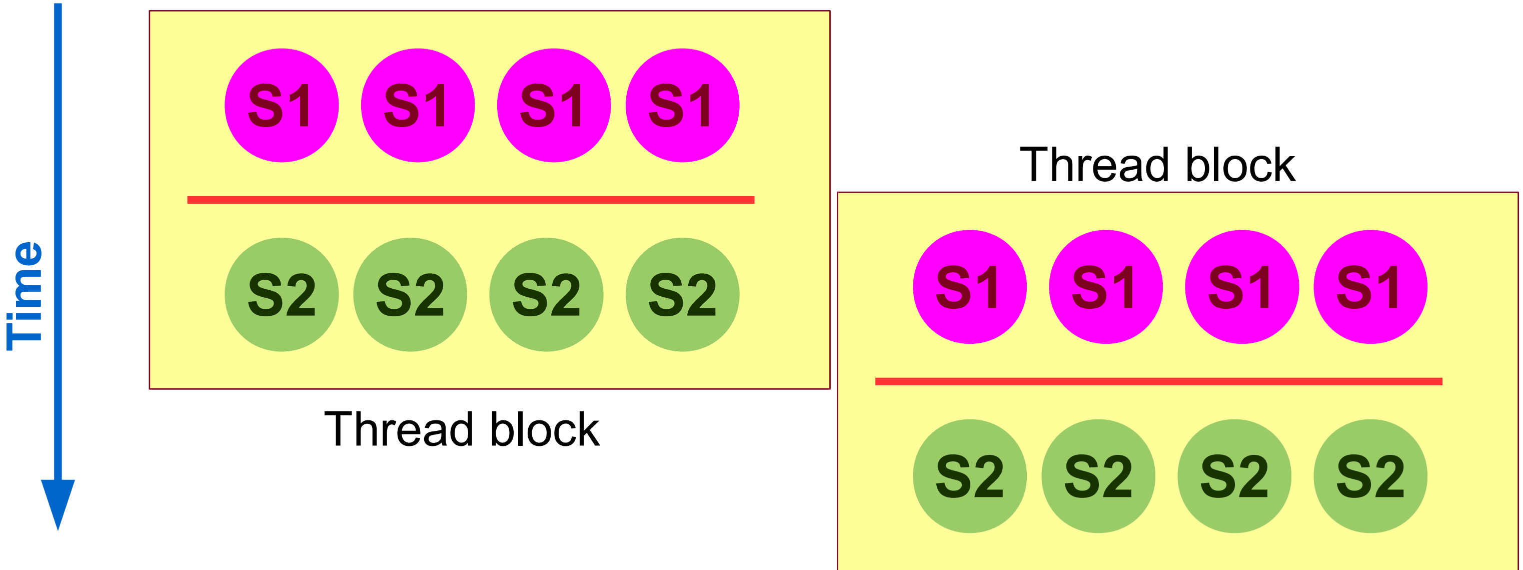
1. Compute sum of all elements of an array.
2. Find the maximum element in an array.
3. Each thread adds elements to a worklist.
   - e.g., next set of nodes to be processed in SSSP.

Barriers

- A barrier is a program point where all threads need to reach before any thread can proceed.
- End of kernel is an implicit barrier for all GPU threads (global barrier).
- There is no explicit global barrier supported in CUDA.
- Threads in a thread-block can synchronize using __syncthreads().
- How about barrier within warp-threads?

Barriers

```
__global__ void dkernel(unsigned *vector, unsigned vectorsize) {
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
    vector[id] = id; S1
    __syncthreads();
    if (id < vectorsize - 1 && vector[id + 1] != id + 1) S2
        printf("syncthreads does not work.\n");
}
```



Barriers

- __syncthreads() is not only about control synchronization, it also has data synchronization mechanism.
- It performs a memory fence operation.
  - A memory fence ensures that the writes from a thread are made visible to other threads.
- There is a separate __threadfence() instruction also.
- A fence does not ensure that other thread will read the updated value.
  - This can happen due to caching.
  - The other thread needs to use volatile data.

Classwork

- Write a CUDA kernel to find maximum over a set of elements, and then let thread 0 print the value in the same kernel.
- Each thread is given work[id] amount of work. Find average work per thread and if a thread's work is above average + K, push extra work to a worklist.
  - This is useful for load-balancing.
  - Also called work-donation.

Synchronization

- Atomics
- Barriers
- **Control + data flow**
- …

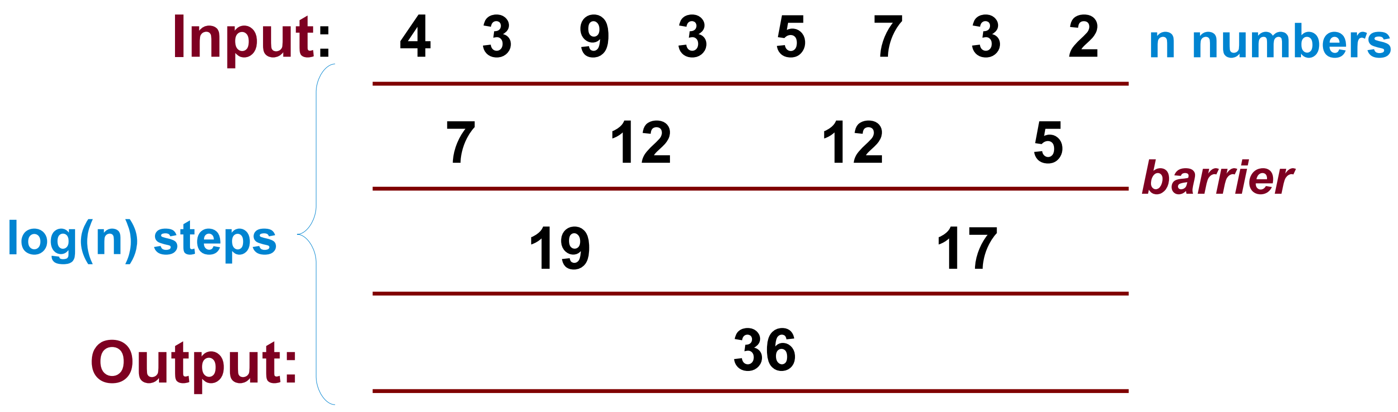
Initially, flag == false.

```
while (!flag) ;
S1;
```

```
S2;
flag = true;
```

Reductions

- What are reductions?
- Computation properties required.
- Complexity measures



Input: 4 3 9 3 5 7 3 2 n numbers

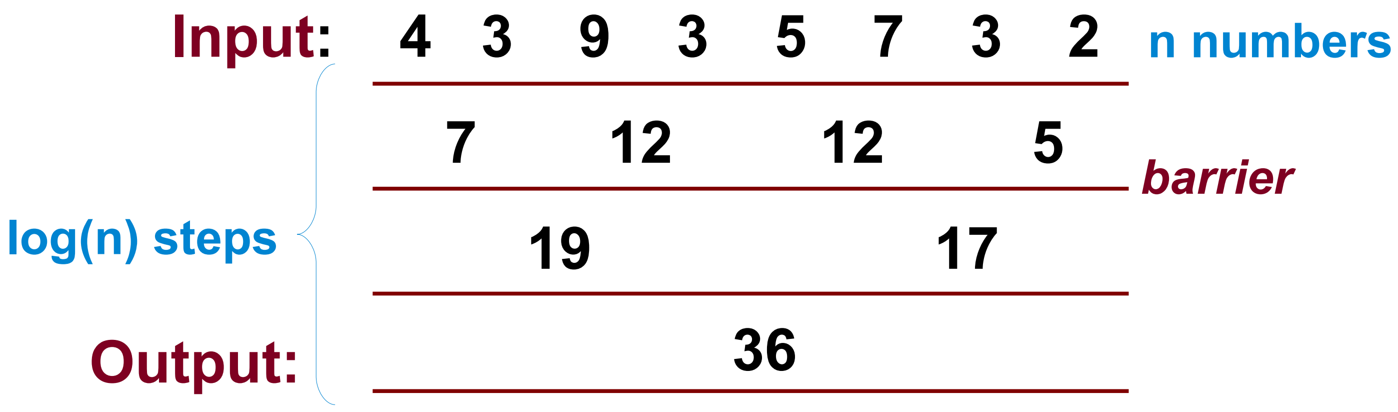
7 12 12 5 barrier

log(n) steps 19 17

Output: 36

Reductions

```
for (int off = n/2; off; off /= 2) {
        if (threadIdx.x < off) {
                a[threadIdx.x] += a[threadIdx.x + off];
        }
        __syncthreads();
}
```



Input: 4 3 9 3 5 7 3 2 n numbers

7 12 12 5 *barrier*

log(n) steps

19 17

Output: 36

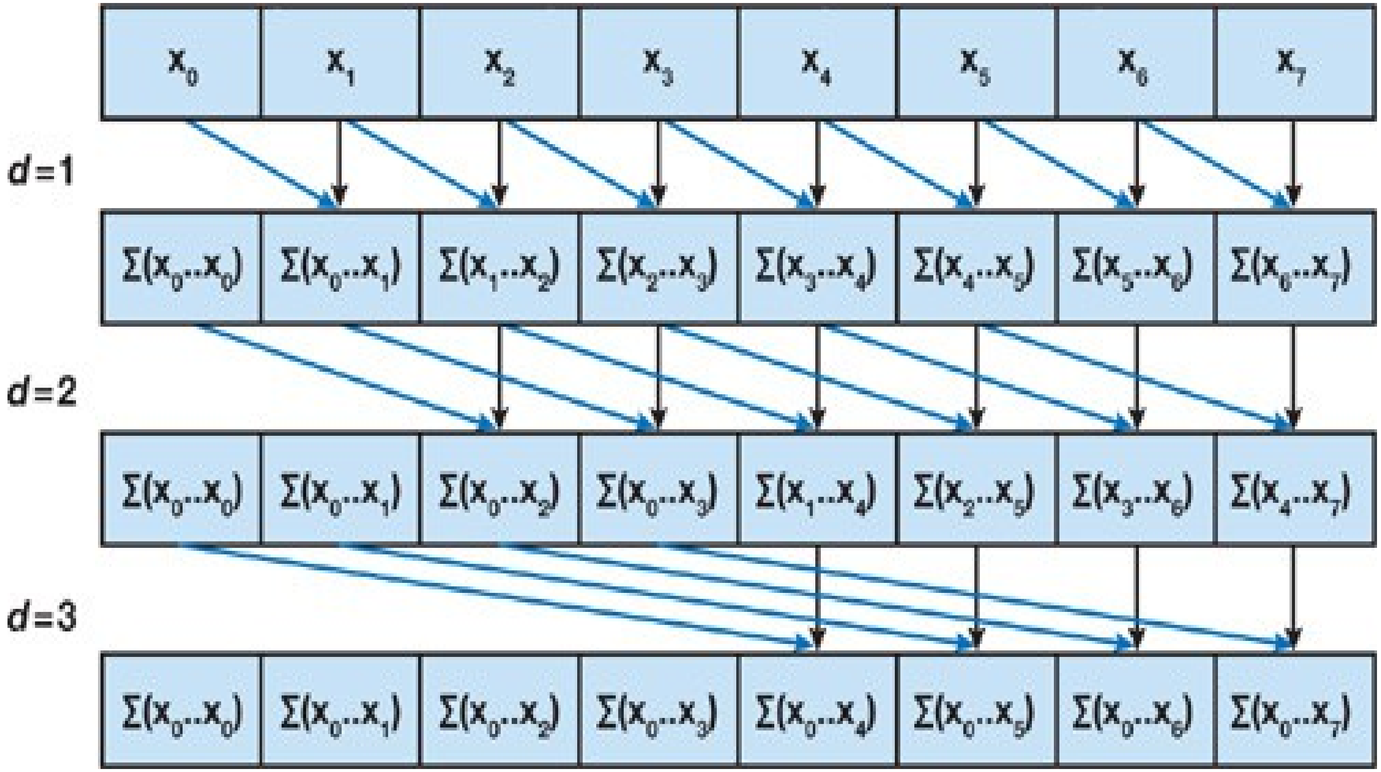
Prefix Sum

- Imagine threads wanting to push work-items to a central worklist.
- Each thread pushes different number of work-items.
- This can be computed using atomics or prefix sum (also called as *scan*).

Input:	**4**	**3**	**9**	**3**	**5**	**7**	**3**	**2**
Output:	**4**	**7**	**16**	**19**	**24**	**31**	**33**	**35**
OR								
Output:	**0**	**4**	**7**	**16**	**19**	**24**	**31**	**33**

Prefix Sum

```
for (int off = 1; off < n; off *= 2) {
        if (threadIdx.x >= off) {
                a[threadIdx.x] += a[threadIdx.x - off];
        }
        __syncthreads();
}
```


x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
$d=1$
$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_1..x_2)$ $\Sigma(x_2..x_3)$ $\Sigma(x_3..x_4)$ $\Sigma(x_4..x_5)$ $\Sigma(x_5..x_6)$ $\Sigma(x_6..x_7)$
$d=2$
$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_0..x_2)$ $\Sigma(x_0..x_3)$ $\Sigma(x_1..x_4)$ $\Sigma(x_2..x_5)$ $\Sigma(x_3..x_6)$ $\Sigma(x_4..x_7)$
$d=3$
$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_0..x_2)$ $\Sigma(x_0..x_3)$ $\Sigma(x_0..x_4)$ $\Sigma(x_0..x_5)$ $\Sigma(x_0..x_6)$ $\Sigma(x_0..x_7)$

Shared Memory

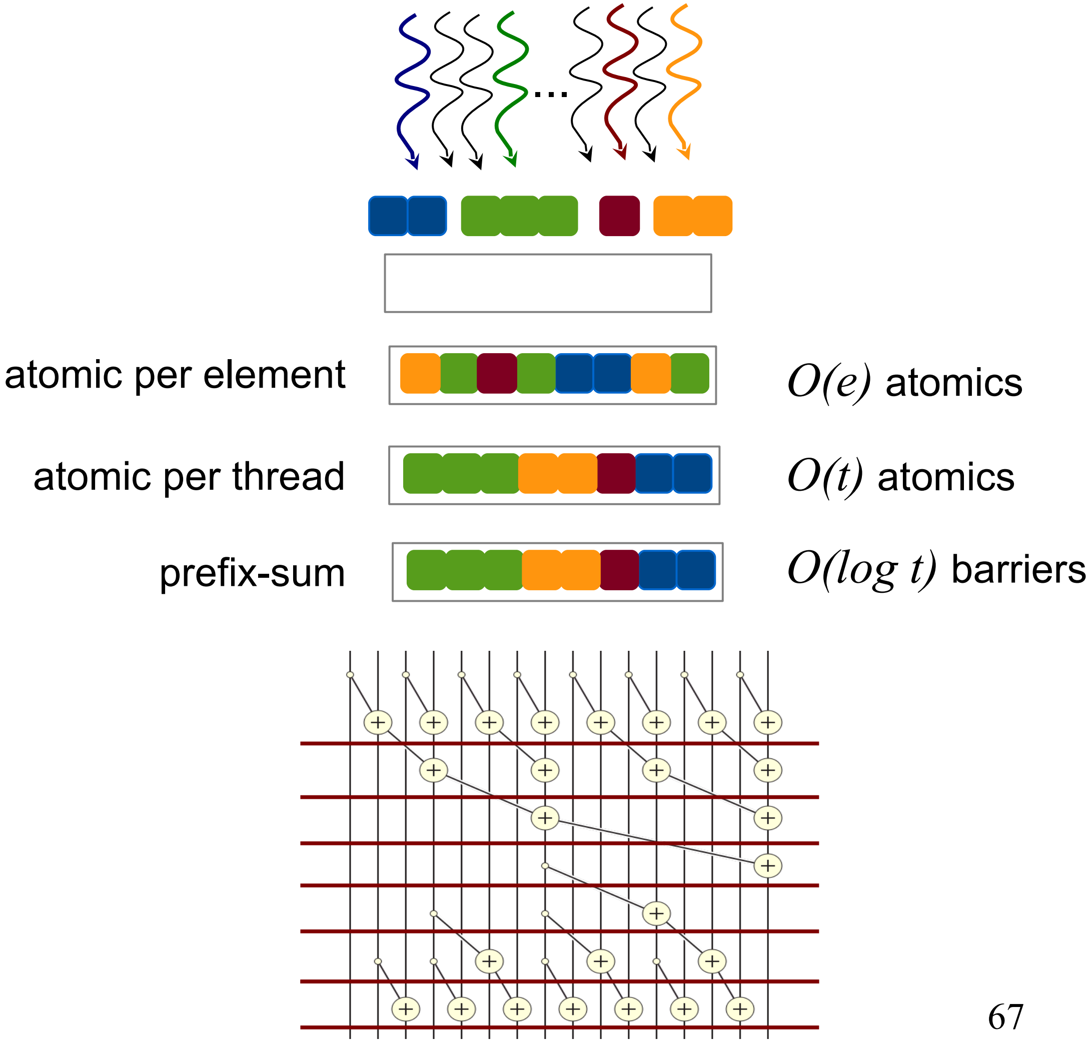
- What is shared memory?
- How to declare Shared Memory?
- Combine with reductions.

```
__shared__ float a[N];
a[id] = id;
```

Barrier-based Synchronization

- ➔ **Disjoint accesses**
- Overlapping accesses
- Benign overlaps

Consider threads pushing elements into a worklist



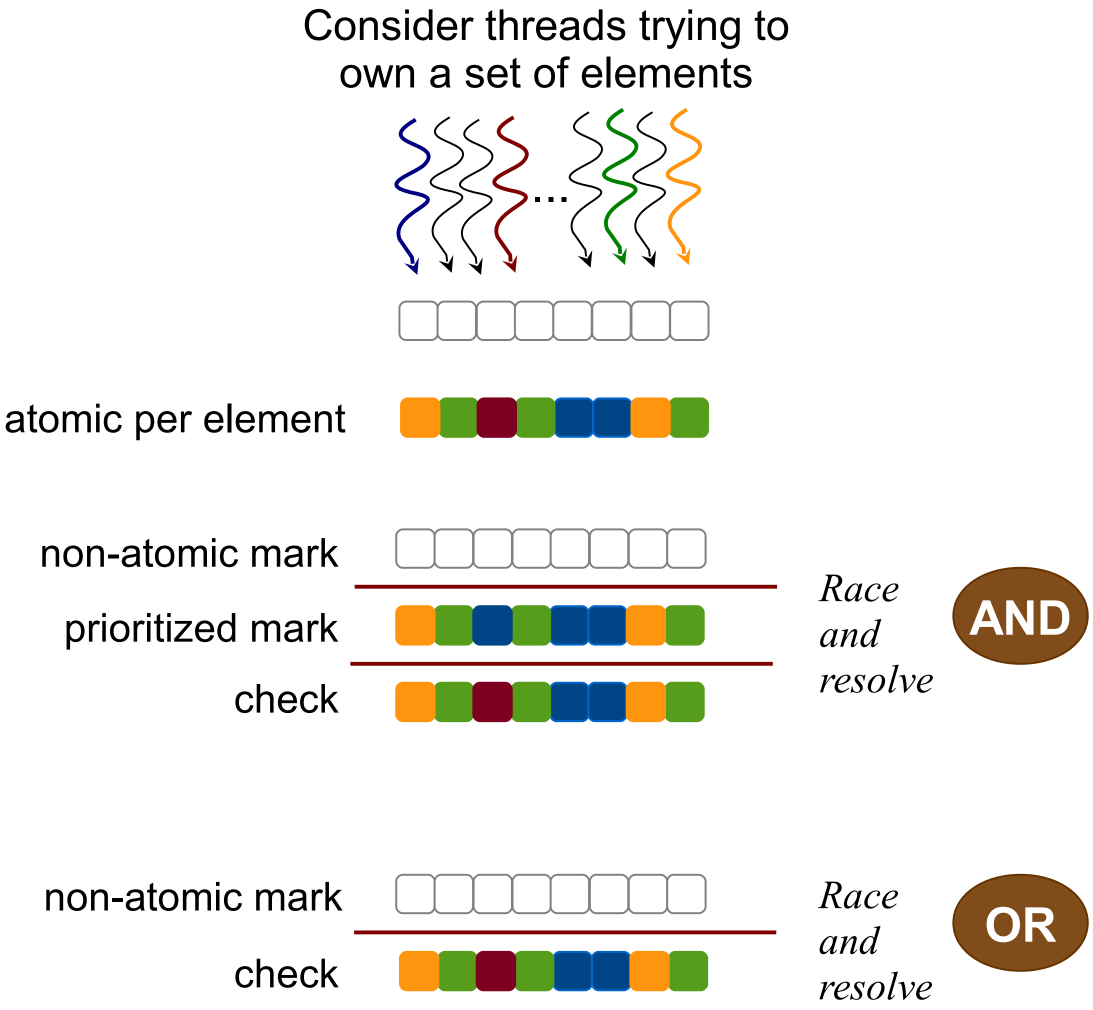
atomic per element — $O(e)$ atomics

atomic per thread — $O(t)$ atomics

prefix-sum — $O(log\ t)$ barriers

Barrier-based Synchronization

- Disjoint accesses
- → **Overlapping accesses**
- Benign overlaps



Consider threads trying to own a set of elements

atomic per element

e.g., for owning cavities in Delaunay mesh refinement

non-atomic mark

prioritized mark

check

Race and resolve

AND

e.g., for inserting unique elements into a worklist

non-atomic mark

check

Race and resolve

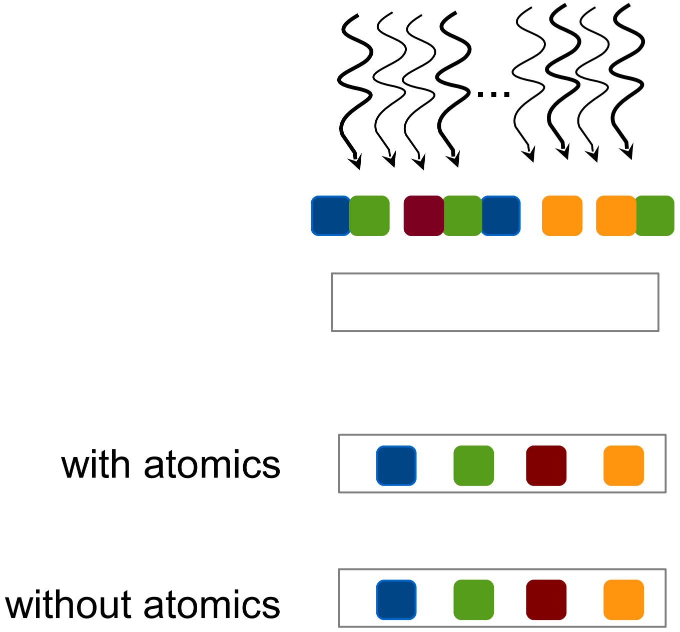
OR

Barrier-based Synchronization

- Disjoint accesses
- Overlapping accesses

➔ **Benign overlaps**

Consider threads updating shared variables to the same value

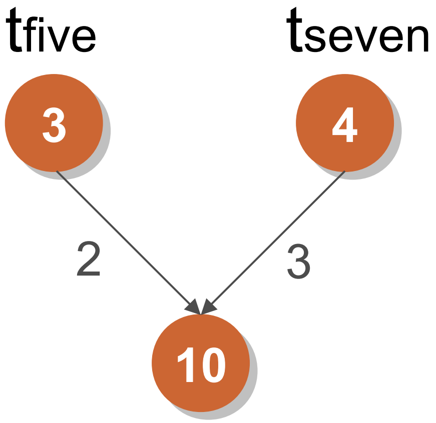


e.g., level-by-level breadth-first search

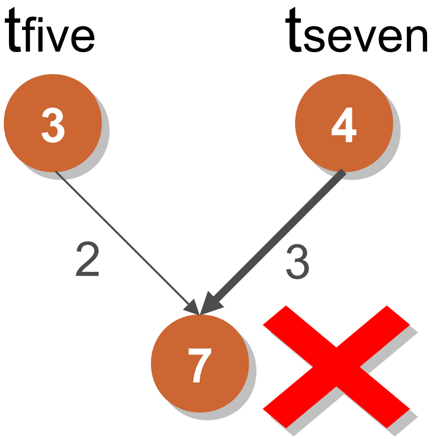
Exploiting Algebraic Properties

- ➔ **Monotonicity**
- Idempotency
- Associativity

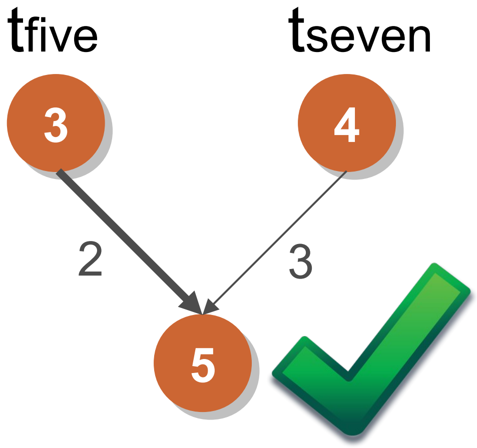
Consider threads updating distances in shortest paths computation



Atomic-free update



Lost-update problem

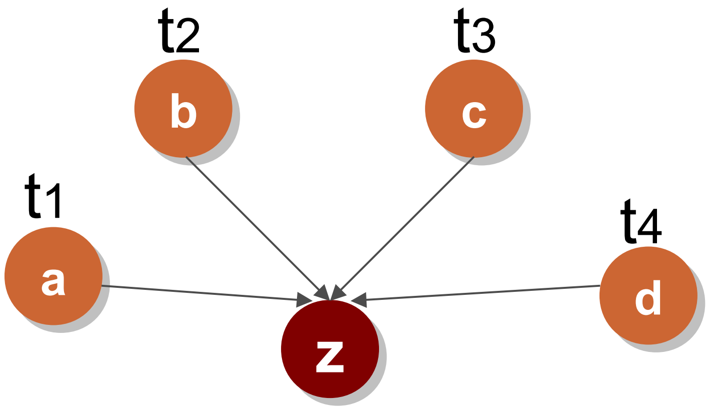


Correction by topology-driven processing, exploiting monotonicity

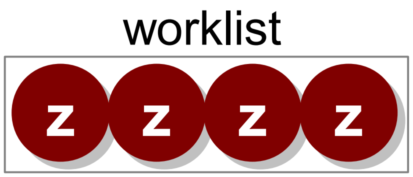
Exploiting Algebraic Properties

- Monotonicity
- ➔ **Idempotency**
- Associativity

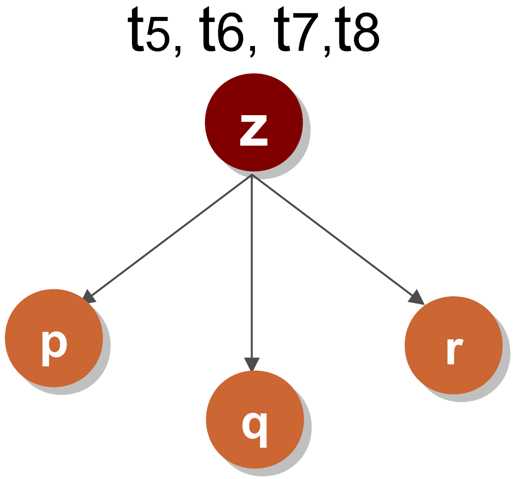
Consider threads updating distances in shortest paths computation



Update by multiple threads



Multiple instances of a node in the worklist



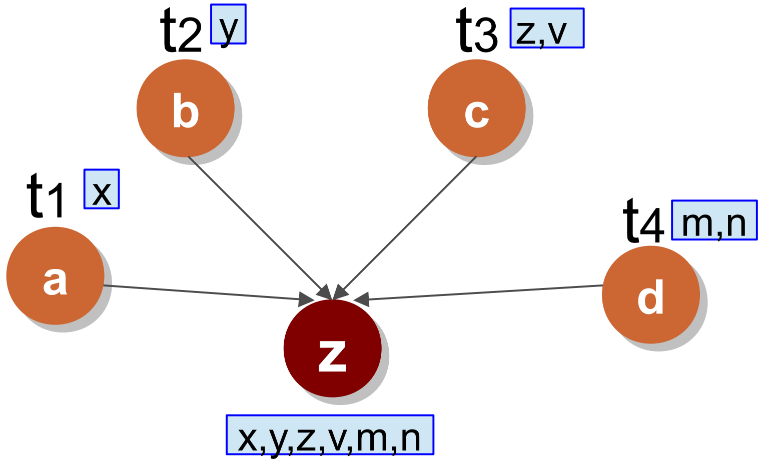
Same node processed by multiple threads

Exploiting Algebraic Properties

- Monotonicity
- Idempotency

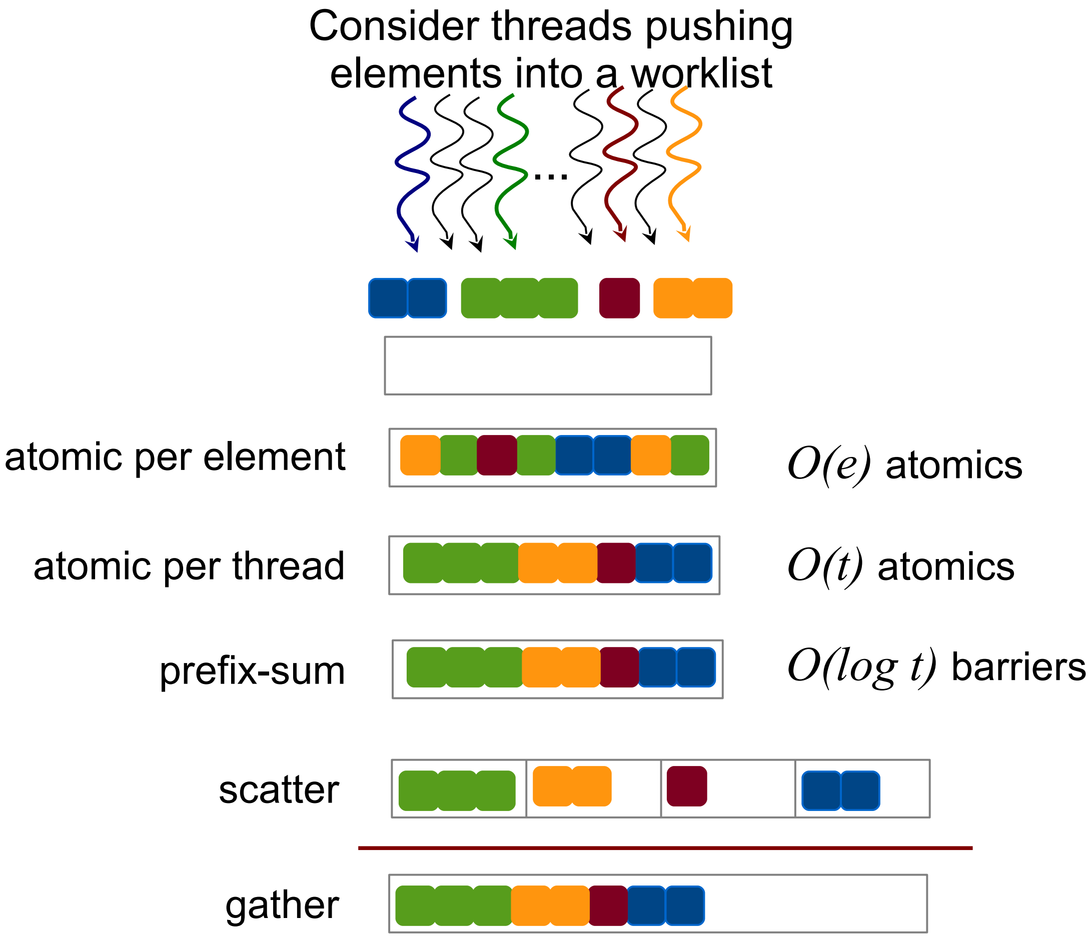
➔ **Associativity**

Consider threads pushing information to a node



Associativity helps push information using prefix-sum

Scatter-Gather



Consider threads pushing elements into a worklist

atomic per element — $O(e)$ atomics

atomic per thread — $O(t)$ atomics

prefix-sum — $O(\log t)$ barriers

scatter

gather

Other Memories

- Texture
- Const
- Global
- Shared
- Cache
- Registers

Thrust

- Thrust is a parallel algorithms library (similar in spirit to STL on CPU).
- Supports vectors and associated transforms.
- Programmer is oblivious to where code executes – on CPU or GPU.
- Makes use of C++ features such as functors.

Thrust

```
thrust::host_vector<int> hnums(1024);
thrust::device_vector<int> dnums;

dnums = hnums;    // calls cudaMemcpy

// initialization.
thrust::device_vector<int> dnum2(hnums.begin(), hnums.end());
hnums = dnum2;    // array resizing happens automatically.

std::cout << dnums[3] << std::endl;

thrust::transform(dsrc.begin(), dsrc.end(), dsrc2.begin(),
                  ddst.begin(), addFunc);
```

Thrust Functions

- find(begin, end, value);
- find_if(begin, end, predicate);
- copy, copy_if.
- count, count_if.
- equal.
- min_element, max_element.
- merge, sort, reduce.
- transform.
- ...

Thrust User-Defined Functors

```
// calculate result[] = (a * x[]) + y[]
struct saxpy {
    const float _a;
    saxpy(int a) : _a(a) { }

    __host__ __device__
    float operator()(const float &x, const float& y) const {
        return a * x + y;
    }
};

thrust::device_vector<float> x, y, result;
// ... fill up x & y vectors ...
thrust::transform(x.begin(), x.end(), y.begin(),
                  result.begin(), saxpy(a));
```

Thrust on host versus device

- Same algorithm can be used on CPU and GPU.

```
 int x, y;
thrust::host_vector<int> hvec;
thrust::device_vector<int> dvec;
// (thrust::reduce is a sum operation by default)
x = thrust::reduce(hvec.begin(), hvec.end());   // on CPU
y = thrust::reduce(dvec.begin(), dvec.end());   // on GPU
```

Challenges with GPU

- Warp-based execution
  - Often requires sorting of work or algorithm change
- Data structure layout
  - Best layout for CPU differs from the best layout for GPU
- Separate memory space
  - Slow transfers
  - Pack/unpack data

- Incoherent L1 caches
  - May need to explicitly push data out
- Poor recursion support
  - Need to make code iterative and maintain explicit iteration stacks
- Thread and block counts
  - Hierarchy complicates implementation
  - Optimal counts have to be (auto-)tuned

General Optimization Principles

- Finding and exposing enough parallelism to populate all the multiprocessors.
- Finding and exposing enough additional parallelism to allow multithreading to keep the cores busy.
- Optimizing device memory accesses for contiguous data.
- Utilizing the software data cache to store intermediate results or to reorganize data.
- Reducing synchronization.

Other Optimizations

- Async CPU-GPU execution
- Dynamic Parallelism
- Multi-GPU execution
- Unified Memory

Bank Conflicts

- Programming guide.

Dynamic Parallelism

- Usage for graph algo.

Async CPU-GPU execution

- Overlapping communication and computation
  - streams
- Overlapping two computations

Multi-GPU execution

- Peer-to-peer copying
- CPU as the driver

Unified Memory

- CPU-GPU memory coherence
- Show the problem first

Other Useful Topics

- Voting functions
- Occupancy
- Compilation flow and .ptx assembly

Voting Functions

Occupancy

- Necessity
- Pitfall and discussion

Compilation Flow

- Use shailesh's flow diagram
- .ptx example

Common Pitfalls and Misunderstandings

- GPUs are only for graphics applications.
- GPUs are only for regular applications.
- On GPUs, all the threads need to execute the same instruction at the same time.
- A CPU program when ported to GPU runs faster.

GPU Programming

Rupesh Nasre.

High-Performance Parallel Computing
June 2016