



Introduction to GPU Computing

CUDA

Gunter Roth

guntterr@nvidia.com

09 March 2016



Tesla Accelerated Computing Platform



Data Center Infrastructure

System Solutions



Communication



Infrastructure Management



Development

Programming Languages



Development Tools



Software Solutions



GPU
Accelerators
GPU Boost

Interconnect
*GPU Direct
NVLink*

System
Management
NVML

Compiler
Solutions
LLVM

Profile and
Debug
CUPTI

Accelerated
Libraries
cuBLAS

Enterprise Services Support & Maintenance

Resources

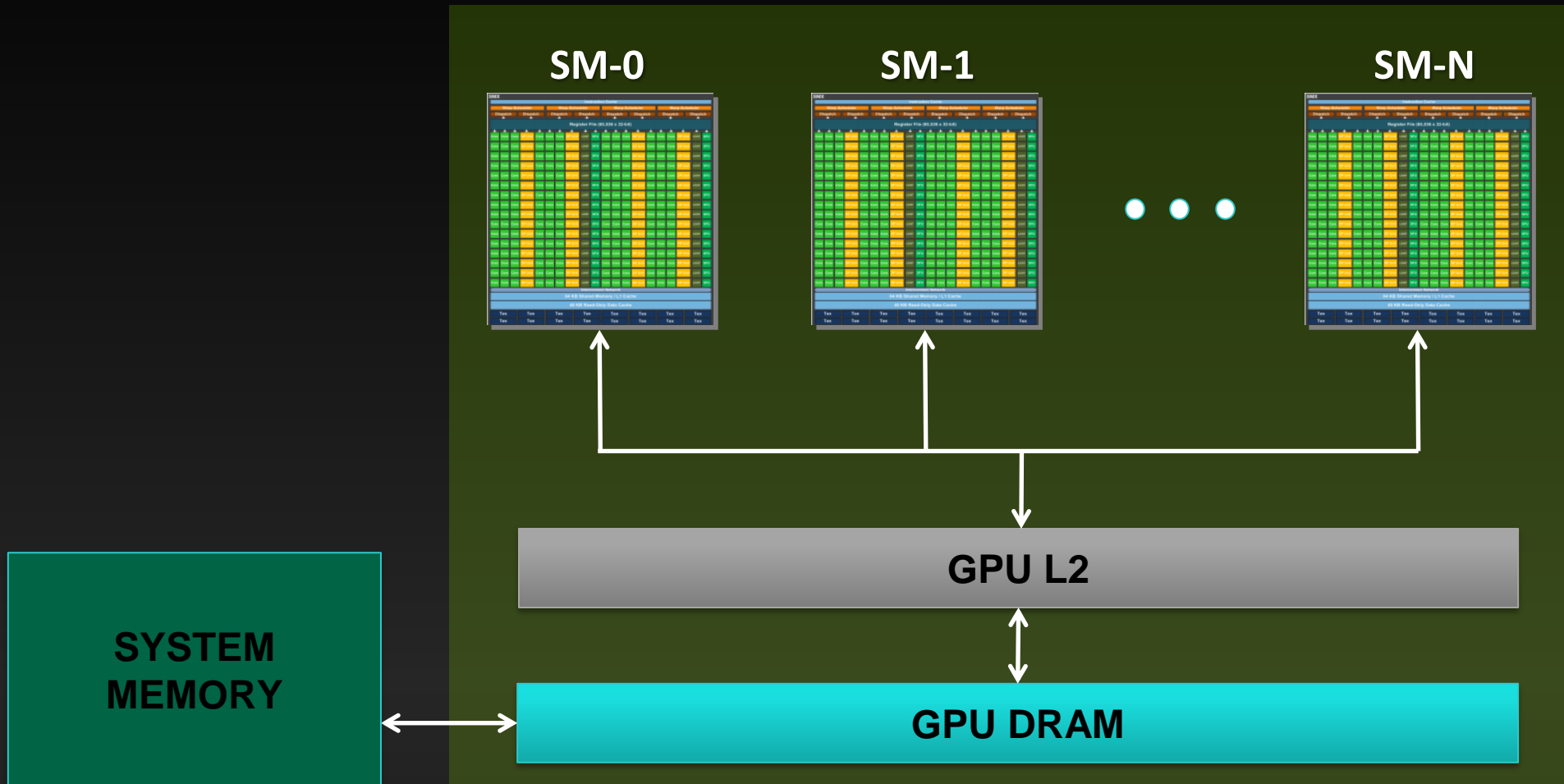
Learn more about GPUs

- ▶ CUDA resource center:
 - ▶ <http://docs.nvidia.com/cuda>
- ▶ GTC on-demand and webinars:
 - ▶ <http://on-demand-gtc.gputechconf.com>
 - ▶ <http://www.gputechconf.com/gtc-webinars>
- ▶ Parallel Forall Blog:
 - ▶ <http://devblogs.nvidia.com/parallelforall>
- ▶ Self-paced labs:
 - ▶ <http://nvlabs.qwiklab.com>



• CPU versus GPU architecture

GPU Architecture

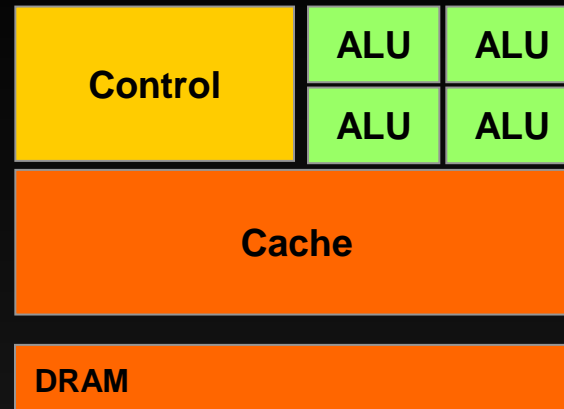


Low Latency or High Throughput?



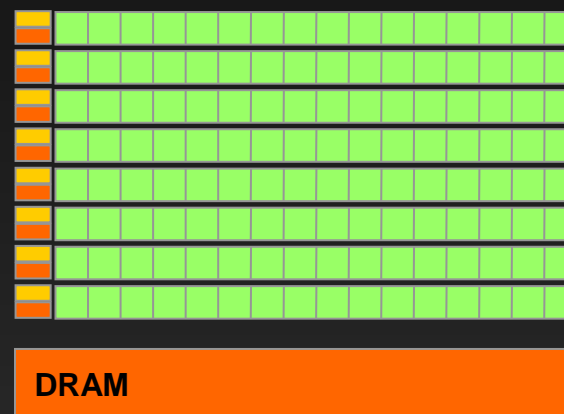
● CPU

- Optimised for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution
- **10's of threads**



● GPU

- Optimised for data-parallel, throughput computation
- Architecture tolerant of memory latency
- Massive fine grain threaded parallelism
- More transistors dedicated to computation
- **10000's of threads**



CPU / Accelerator Differences

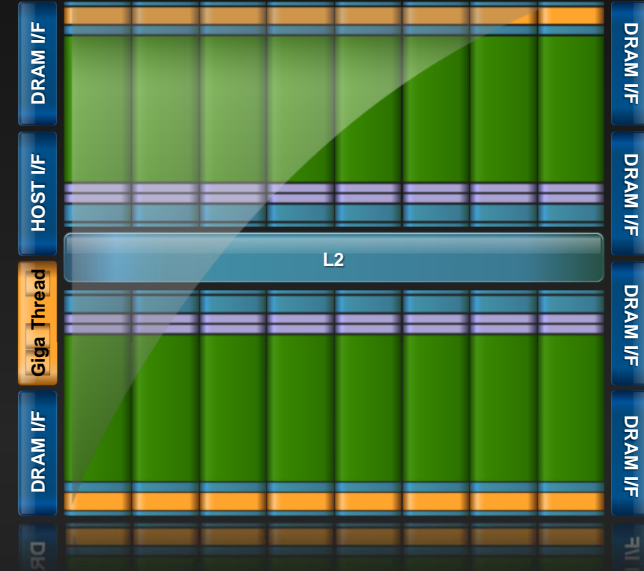


- **Faster clock (2.5-3.5 GHz)**
 - **More work per clock**
 - Pipelining (deep)
 - Multiscalar (3-5)
 - SIMD width (4-16)
 - More cores (6-12)
 - **Fewer stalls**
 - Large cache memories
 - Complex branch prediction
 - Out-of-order execution
 - Multithreading (2-4)
- **Slower clock (0.8-1.0 GHz)**
 - **More work per clock**
 - ▶ Pipelining (shallow)
 - ▶ Multiscalar (1-2)
 - ▶ SIMD width (16-64)
 - ▶ More cores (15-60)
 - **Fewer stalls**
 - ▶ Small cache memories
 - ▶ Little branch prediction
 - ▶ In-order execution
 - ▶ Multithreading (15-32)

GPU Architecture:

Two Main Components

- **Global memory**
 - Analogous to RAM in a CPU server
 - Accessible by both GPU and CPU
 - Currently up to **24 GB** per GPU
 - Bandwidth currently up to **~288 GB/s** (Tesla products)
 - **ECC on/off** (Quadro and Tesla products)
- **Streaming Multiprocessors (SMs)**
 - Perform the actual computations
 - Each SM has its own:
 - Control units, registers, execution pipelines, caches



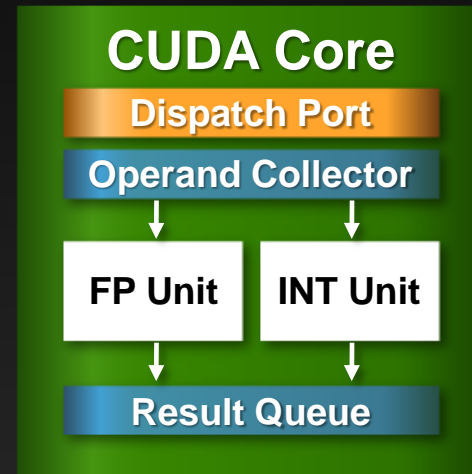
FERMI : SM Architecture

- 32 CUDA cores per SM (512 total)
- 8x peak double precision floating point performance
 - 50% of peak single precision
- Dual Thread Scheduler
- 64 KB of RAM for shared memory and L1 cache (configurable)



FERMI : CUDA Core Architecture

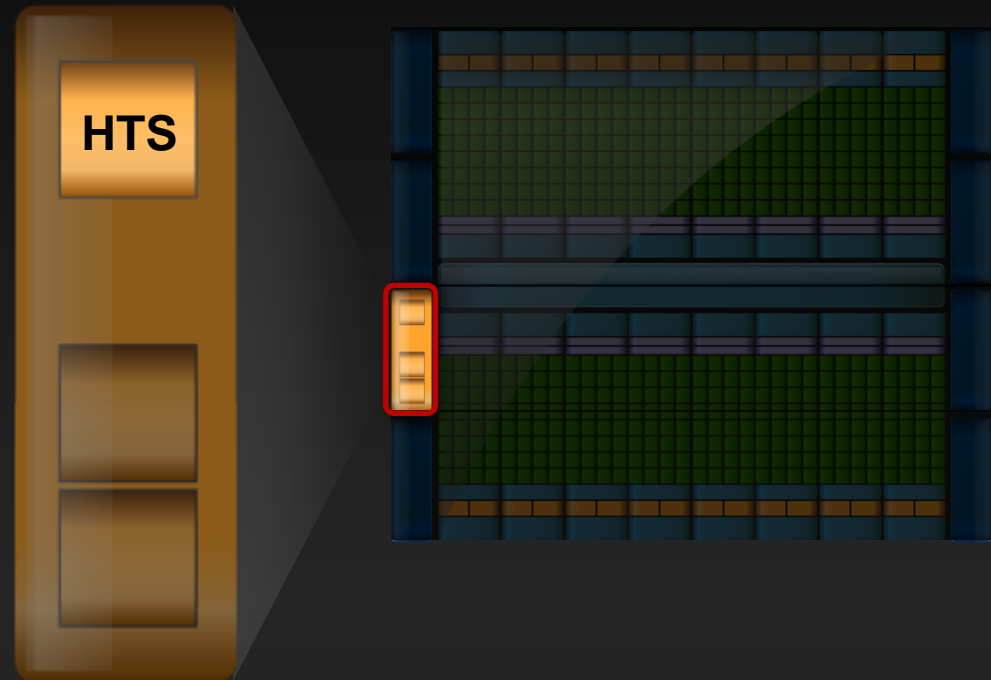
- New IEEE 754-2008 floating-point standard, surpassing even the most advanced CPUs
- Fused multiply-add (FMA) instruction for both single and double precision
- Newly designed integer ALU optimized for 64-bit and extended precision operations



GigaThread™ Hardware Thread Scheduler (HTS)



- Hierarchically manages thousands of simultaneously active threads
- 10x faster application context switching
- Concurrent kernel execution



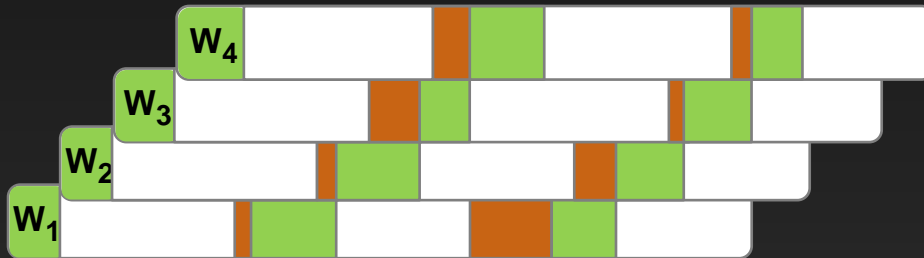
Low Latency or High Throughput?

- **CPU** architecture must **minimize latency** within each thread
- **GPU** architecture **hides latency** with computation from other threads

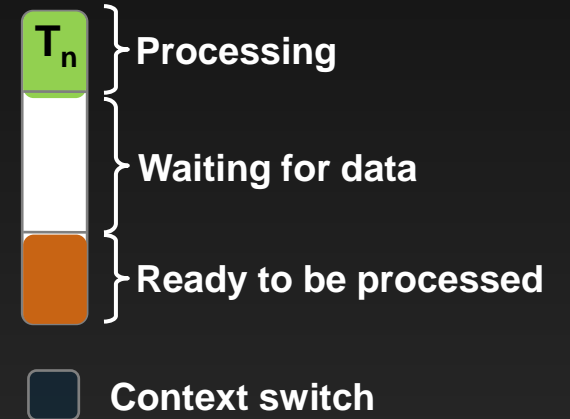
CPU core – Low Latency Processor



GPU Stream Multiprocessor – High Throughput Processor



Computation Thread/Warp



Fermi Memory Hierarchy Review



Local storage

- Each thread has own local storage
- Mostly registers (managed by the compiler)

Shared memory / L1

- Program configurable: 16KB shared / 48 KB L1 OR 48KB shared / 16KB L1
- Shared memory is accessible by the threads in the same threadblock
- Low latency
- Very high throughput (1.33 TB/s aggregate on Tesla M2090)

L2

- All accesses to global memory go through L2, including copies to/from CPU host
- 768 KB on Tesla M2090

Global memory

- Accessible by all threads as well as host (CPU)
- Higher latency (400-800 cycles)
- Throughput: 177 GB/s on Tesla M2090

Programming for L1 and L2



Short answer: DON'T

- **GPU caches are not intended for the same use as CPU caches**
 - Smaller size (especially per thread), so not aimed at temporal reuse
 - Intended to smooth out some access patterns, help with spilled registers, etc.
- **Don't try to block for L1/L2 like you would on CPU**
 - You have 100s to 1,000s of run-time scheduled threads hitting the caches
- If it is possible to block for L1 then block for SMEM
- Same size, same bandwidth, hw will not evict behind your back

Fermi Shared Memory



Uses

- Inter-thread communication within a block
- Cache data to reduce redundant global memory accesses
- Use it to improve global memory access patterns

Fermi organization:

- 32 banks, 4-byte wide banks
- Successive 4-byte words belong to different banks

Performance:

- 4 bytes per bank per 2 clocks per multiprocessor: **1.3 TB/s on M2090**
- smem accesses are issued per 32 threads (warp)
- serialization: if n threads in a warp access different 4-byte words in the same bank, n accesses are executed serially
- multicast: n threads access the same word in one fetch
- Could be different bytes within the same word

Fermi Constant and Texture Data



Constants:

- `__constant__` qualifier in declarations
- Up to 64KB
- Ideal when the same address is read by all threads in a warp (FD coefficients, etc.)
- Throughput is 4B per SM per clock

Textures:

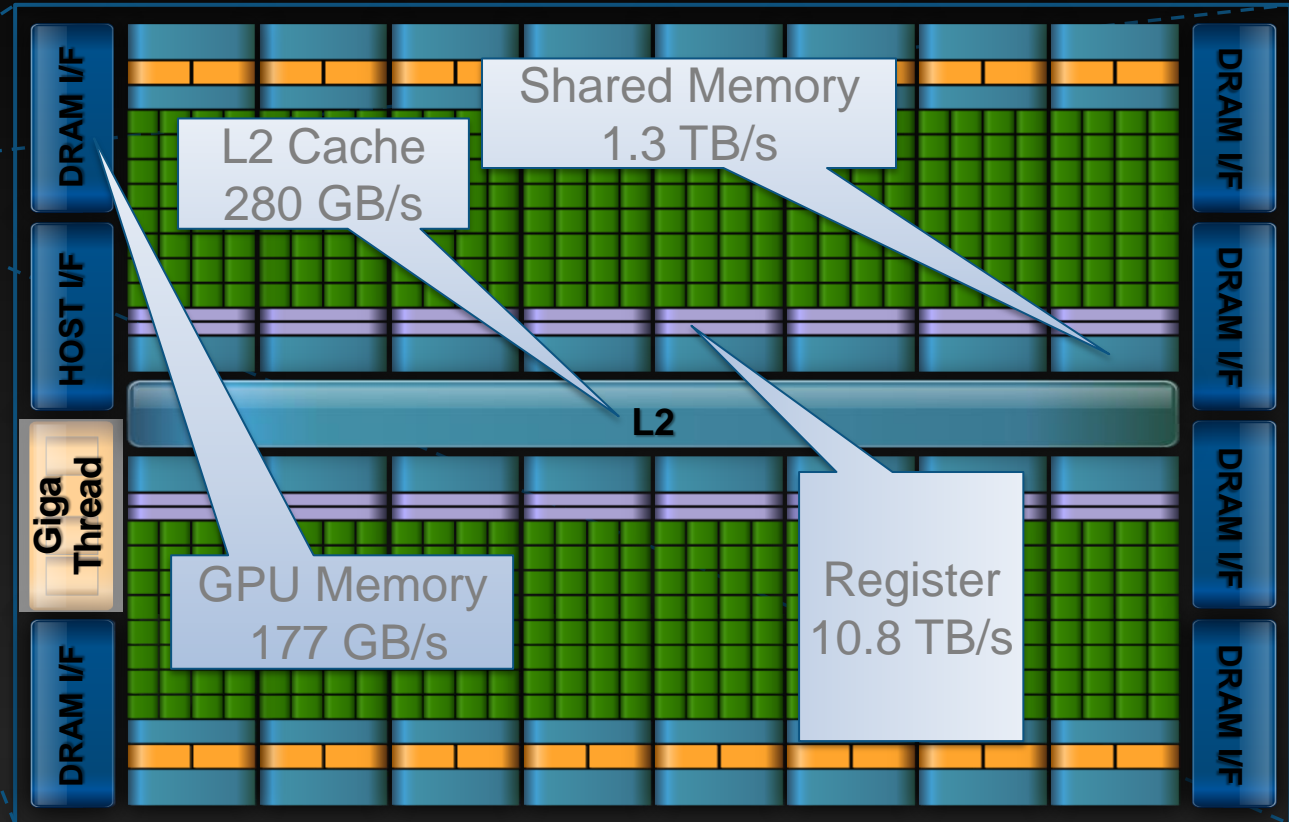
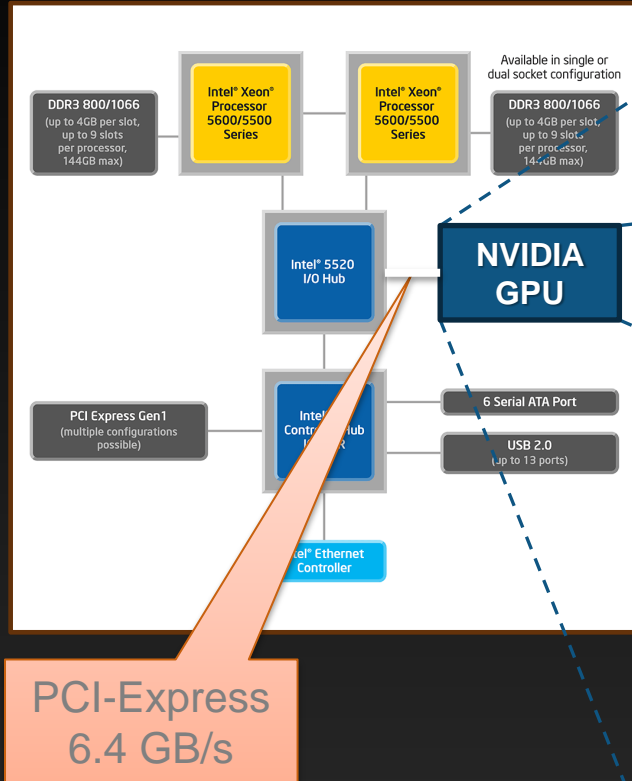
- Dedicated hardware for:
 - Out-of-bounds index handling (clamp or wrap-around)
 - Optional interpolation (think: using fp indices for arrays)
 - Linear, bilinear, trilinear
 - Optional format conversion
 - {char, short, int} -> float

Operation:

- Both textures and constants reside in global memory
- Both are read via dedicated caches

Scientific Computing Challenge: Memory Bandwidth

NVIDIA Technology Solves Memory Bandwidth Challenges

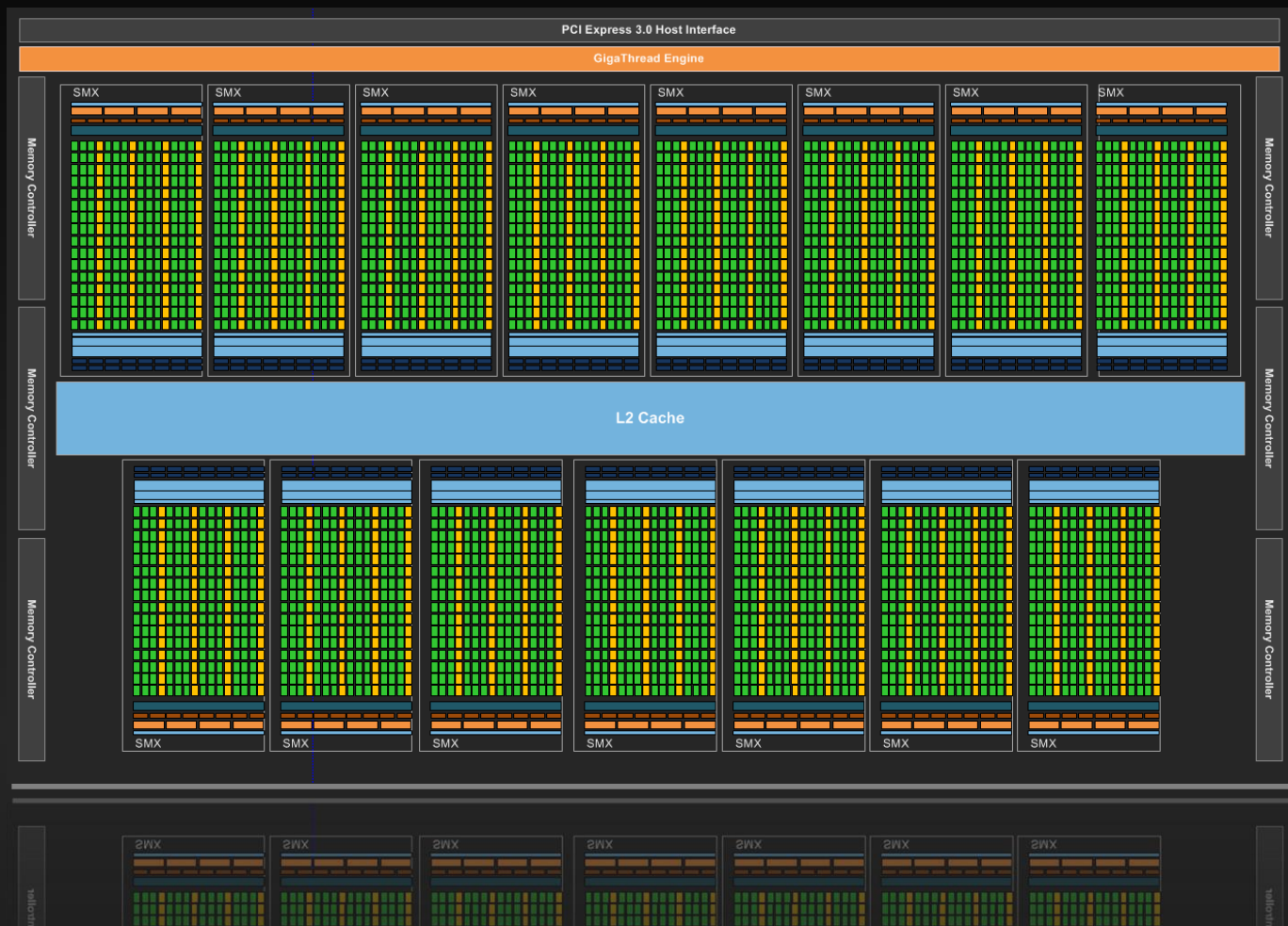


Kepler GK110 Block Diagram



Architecture

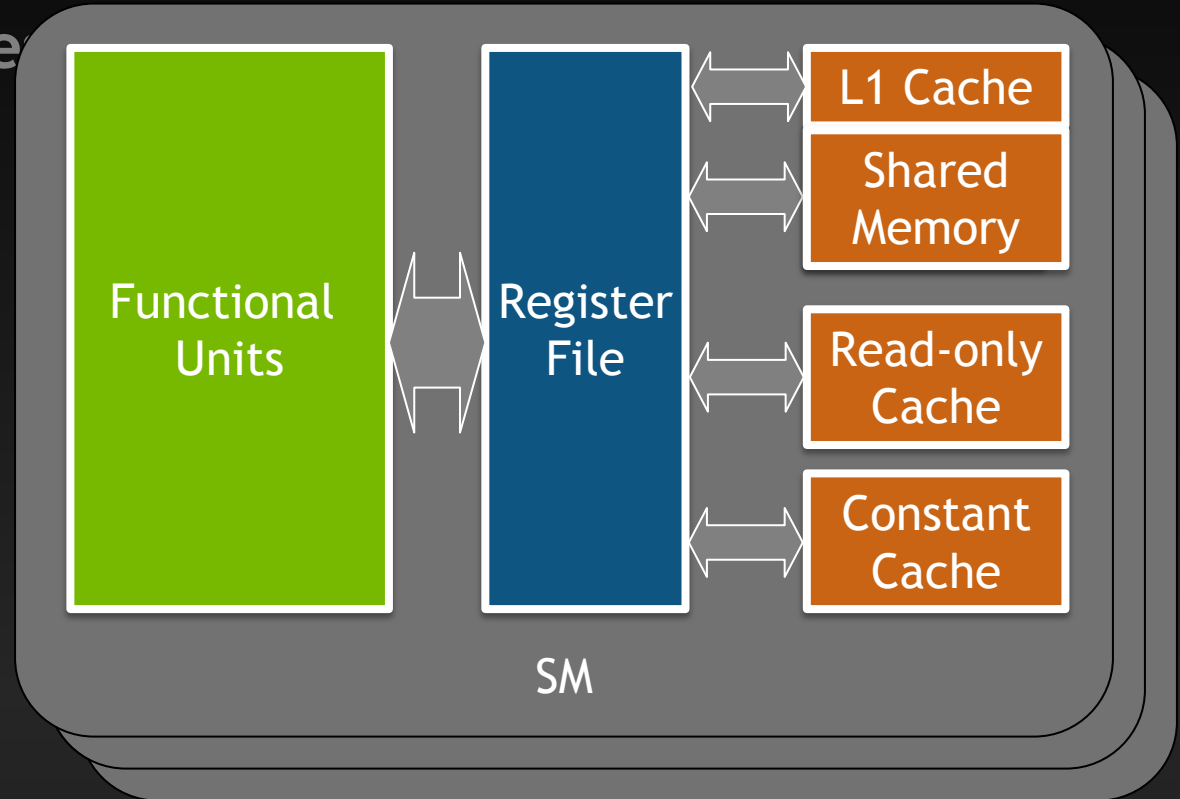
- 7.1B Transistors
- 15 SMX units
- > 1 TFLOP FP64
- 1.5 MB L2 Cache
- 384-bit GDDR5



GPU SM Architecture

Kepler SM

- ▶ **Functional Units = CUDA core**
 - ▶ 192 SP FP operations/clock
 - ▶ 64 DP FP operations/clock
- ▶ **Register file (256KB)**
- ▶ **Shared memory (16-48KB)**
- ▶ **L1 cache (16-48KB)**
- ▶ **Read-only cache (48KB)**
- ▶ **Constant cache (8KB)**



Best Practices

Optimize Data Locality: SM

- ▶ Minimize redundant accesses to L2 and DRAM
 - ▶ Store intermediate results in **registers** instead of global memory
 - ▶ Use **shared memory** for data frequently used within a thread block
 - ▶ Use `const __restrict__` to take advantage of **read-only cache**

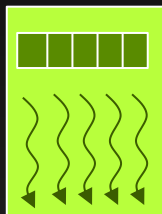


SIMT Execution Model

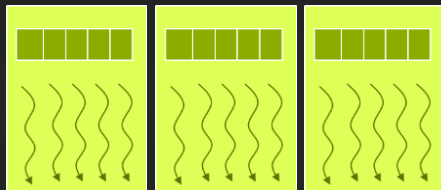
Software



Thread



Thread Block



Grid

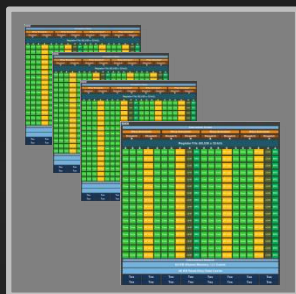
Hardware



CUDA
Core



Multiprocessor



Device

Threads are executed by CUDA Cores

Thread blocks are executed on multiprocessors

Thread blocks do not migrate

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)

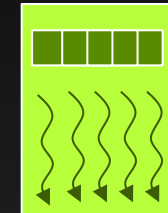
A **kernel** is launched as a grid of thread blocks

SIMT Execution Model

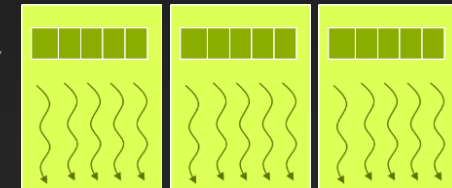
- ▶ **Thread**: sequential execution unit
 - ▶ All threads execute same sequential program
 - ▶ Threads execute in parallel
- ▶ **Thread Block**: a group of threads
 - ▶ Threads within a block can cooperate
 - ▶ Light-weight synchronization
 - ▶ Data exchange
- ▶ **Grid**: a collection of thread blocks
 - ▶ Thread blocks do not synchronize with each other
 - ▶ Communication between blocks is expensive



Thread



Thread Block



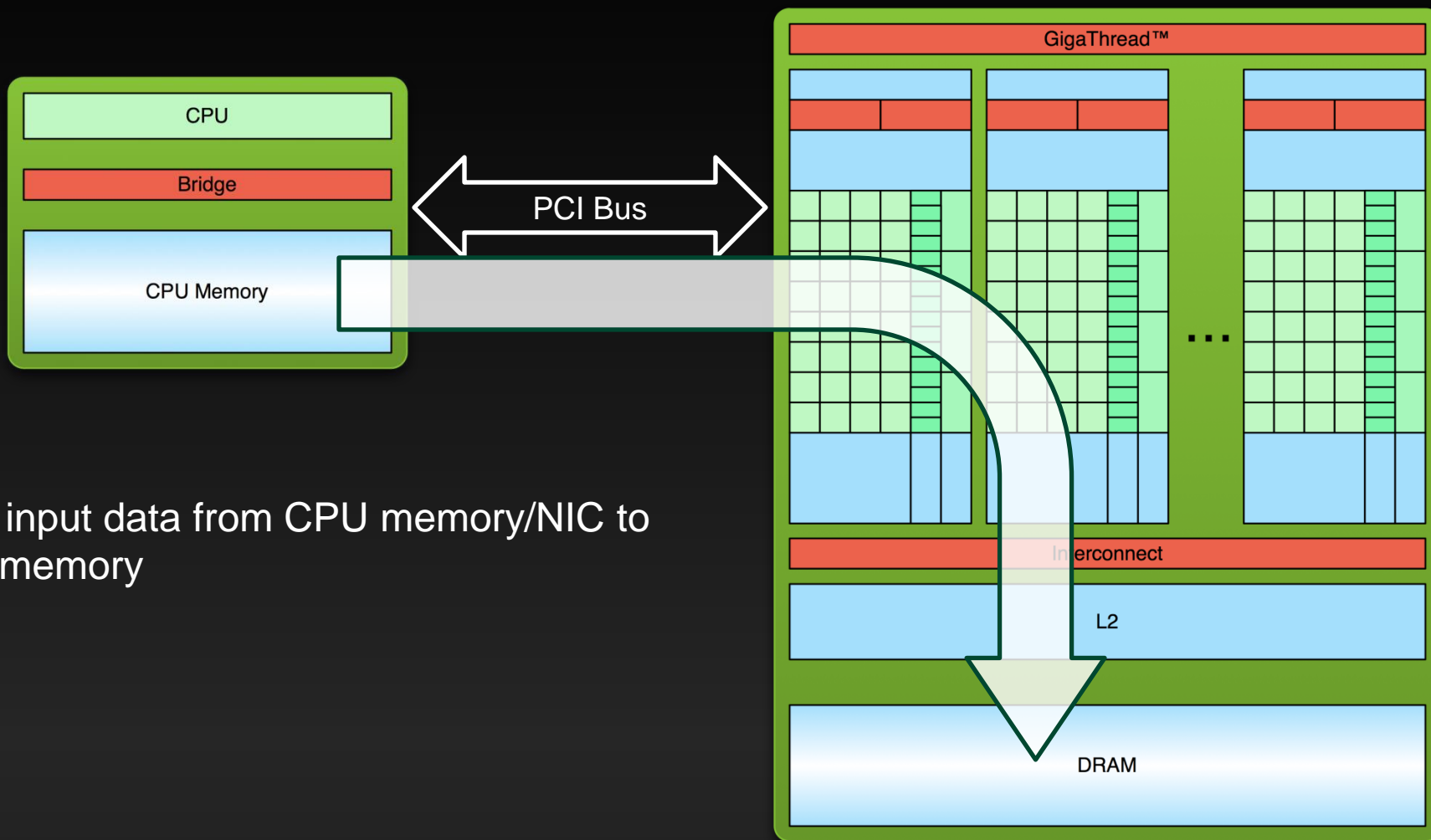
Grid

SIMT Execution Model

- ▶ Threads are organized into groups of 32 threads called “warps”
- ▶ All threads within a warp execute the same instruction simultaneously

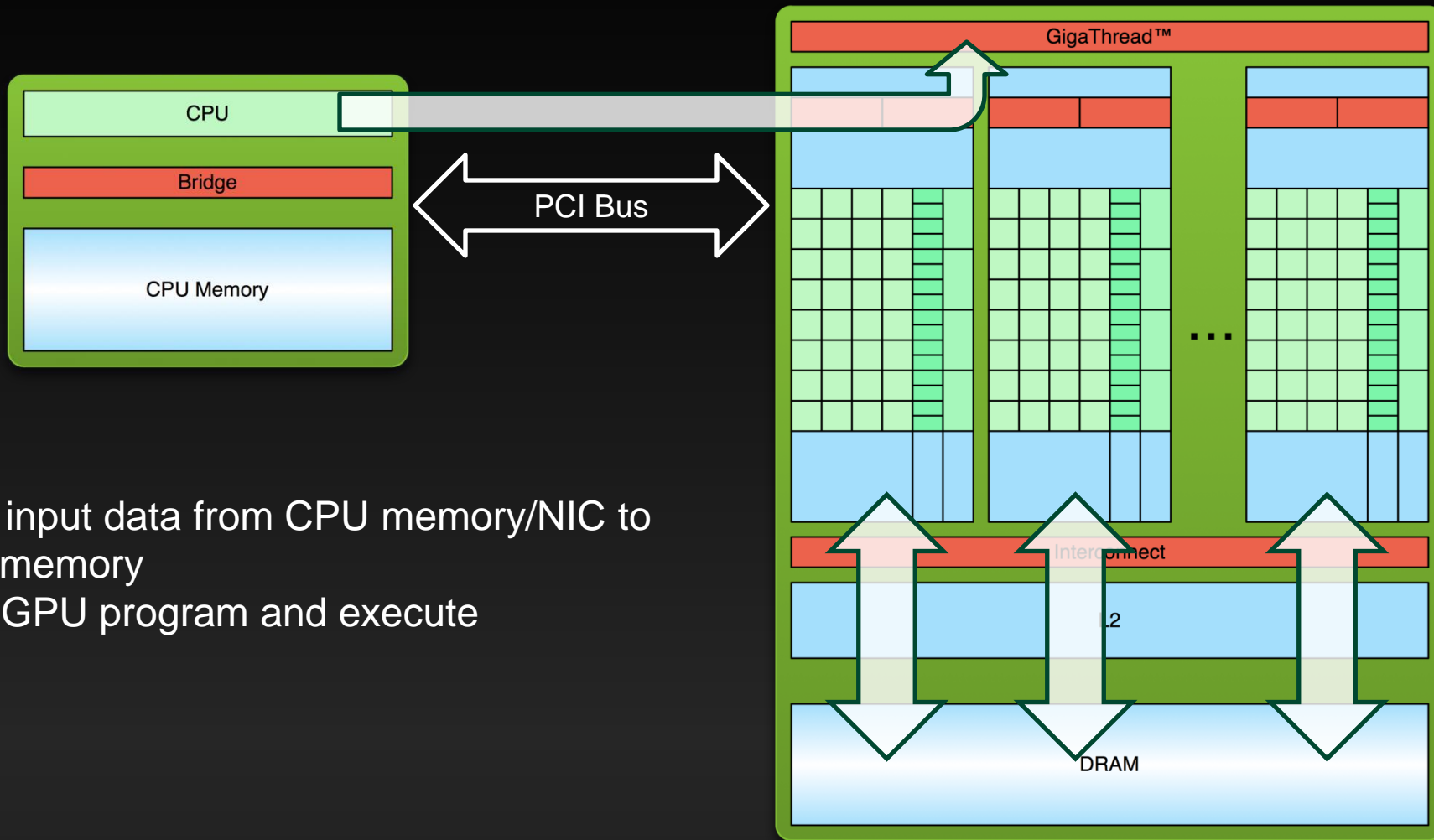


Simple Processing Flow



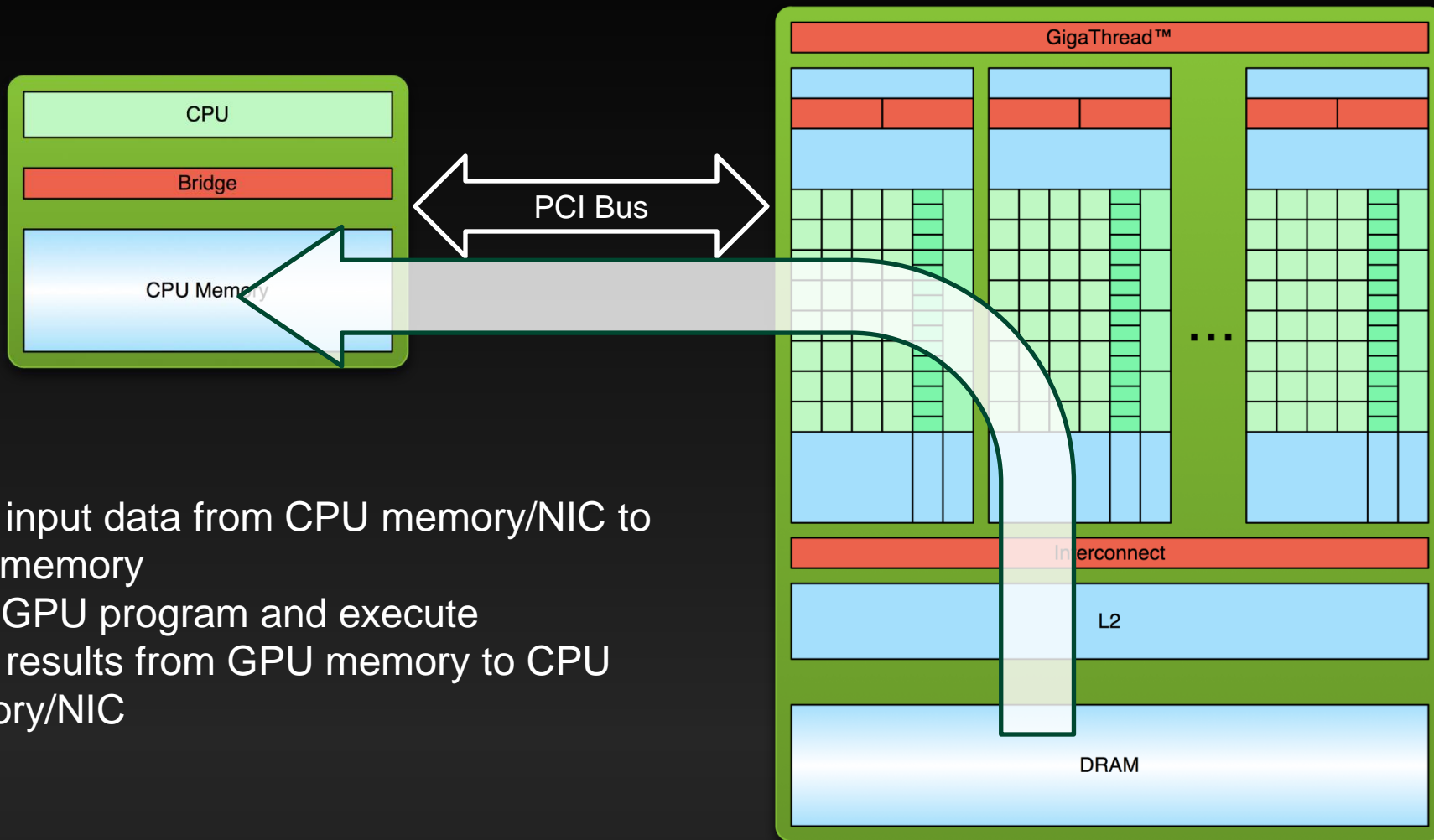
1. Copy input data from CPU memory/NIC to GPU memory

Simple Processing Flow



1. Copy input data from CPU memory/NIC to GPU memory
2. Load GPU program and execute

Simple Processing Flow

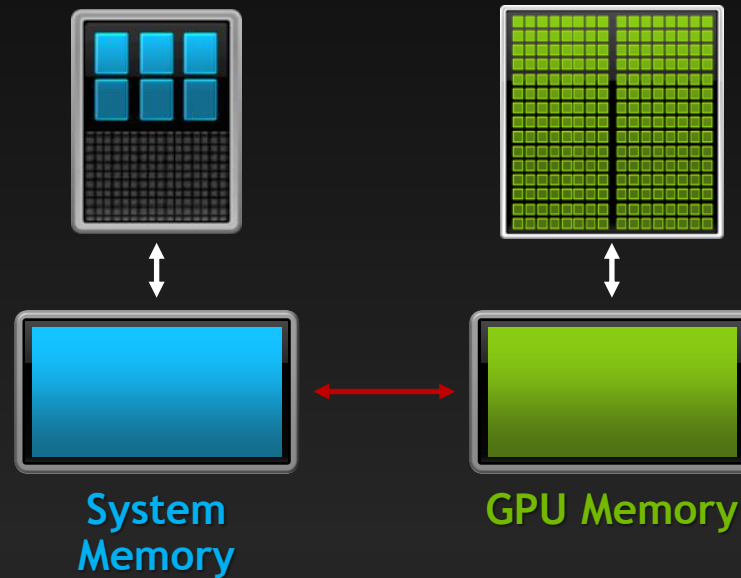


1. Copy input data from CPU memory/NIC to GPU memory
2. Load GPU program and execute
3. Copy results from GPU memory to CPU memory/NIC

Best Practices

Optimize Data Locality: GPU

- Minimize data transfers between CPU and GPU





Amount of Parallelism

GPUs issue instructions in order

- Issue stalls when instruction arguments are not ready

GPUs switch between threads to hide latency

- Context switch is free: thread state is partitioned (large register file), not stored/restored

Conclusion: need enough threads to hide math latency and to saturate the memory bus

- Independent instructions (ILP) within a thread also help

Very rough rule of thumb:

- Need ~512/1000 threads per SM
- At least a few 1,000 threads per GPU

Accelerator Fundamentals

- ▶ We must expose enough parallelism to saturate the device
 - ▶ Accelerator threads are slower than CPU threads
 - ▶ Accelerators have orders of magnitude more threads

Fine-grained parallelism is good

t0	t1	t2	t3
t4	t5	t6	t7
t8	t9	t10	t11
t12	t13	t14	t15

Coarse-grained parallelism is bad

t0	t0	t0	t0
t1	t1	t1	t1
t2	t2	t2	t2
t3	t3	t3	t3

Control Flow

- **Single-Instruction Multiple-Threads (SIMT) model**
 - A single instruction is issued for a warp (thread-vector) at a time
 - NVIDIA GPU: warp = a vector of 32 threads
- **Compare to SIMD:**
- SIMD requires vector code in each thread
- SIMT allows you to write scalar code per thread
 - Vectorization is guaranteed by hardware

Note:

- All contemporary processors (CPUs and GPUs) are built by aggregating vector processing unit

Kernel Execution

Threadblocks are assigned to SMs

- Done at run-time, so don't assume any particular order
- Once a threadblock is assigned to an SM, it stays resident until all its threads complete
- It's not migrated to another SM
- It's not swapped out for another threadblock

Instructions are issued/executed per warp

- Warp = 32 consecutive threads
- Think of it as a “vector” of 32 threads
- The same instruction is issued to the entire warp

Control Flow with divergent branches

Divergent branches:

- Threads within a single warp take different paths

if-else, ...

- Different execution paths within a warp are serialized
- Different warps can execute different code with no impact on performance
- Avoid diverging within a warp

Example with divergence:

- `if (threadIdx.x > 2) {...} else {...}`
- Branch granularity < warp size

Example without divergence:

- `if (threadIdx.x / WARP_SIZE > 2) {...} else {...}`
- Branch granularity is a whole multiple of warp size



Requirements for Maximum Performance

- **Have sufficient parallelism**
 - At least a few 1,000 threads per function (10,000 threads in total)
- **Coalesced memory access**
 - By threads in the same “thread-vector”
- **Coherent execution**
 - By threads in the same “thread-vector”

3 Ways to Accelerate Applications

Applications

Libraries

“Drop-in”
Acceleration

Compiler
Directives

Easily Accelerate
Applications

Programming
Languages

Maximum
Flexibility

Programming Languages

Numerical analytics ►

MATLAB, Mathematica, LabVIEW, Scilab, Octave

Fortran ►

OpenACC, CUDA Fortran

C ►

OpenACC, **CUDA C**

C++ ►

Thrust, **CUDA C++**, **KOKKOS**, **RAJA**, **HEMI**, **OCCA**

Python ►

PyCUDA, Copperhead, Numba, Numbapro

JAVA,C# ►

GPU.NET, Hybridizer (Altimesh), JCUDA, CUDA4J

Compile Python for Parallel Architectures

- **Anaconda Accelerate from Continuum Analytics**
 - NumbaPro array-oriented compiler for Python & NumPy
 - Compile for CPUs or GPUs (uses LLVM + NVIDIA Compiler SDK)
- **Fast Development + Fast Execution: Ideal Combination**



Free Academic
License

<http://continuum.io>



Numba Python Compiler

- Free and open source compiler for array-oriented Python
- NEW numba.cuda module integrates CUDA directly into Python

```
@cuda.jit("void(float32[:], float32, float32[:], float32[:])")
def saxpy(out, a, x, y):
    i = cuda.grid(1)
    out[i] = a * x[i] + y[i]

# Launch saxpy kernel
saxpy[griddim, blockdim](out, a, x, y)
```

- <http://numba.pydata.org/>



More C++ parallel for loops

GPU Lambdas Enable Custom Parallel Programming Models



Kokkos

<https://github.com/kokkos>

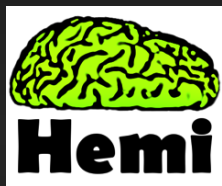
```
Kokkos::parallel_for(N, KOKKOS_LAMBDA (int i) {  
    y[i] = a * x[i] + y[i];  
});
```



RAJA

<https://e-reports-ext.llnl.gov/pdf/782261.pdf>

```
RAJA::forall<cuda_exec>(0, N, [=] __device__ (int i) {  
    y[i] = a * x[i] + y[i];  
});
```



Hemi
CUDA Portability
Library

<http://github.com/harrism/hemi>

```
hemi::parallel_for(0, N, [=] HEMI_LAMBDA (int i) {  
    y[i] = a * x[i] + y[i];  
});
```

STANDARD TEMPLATE LIBRARY

Higher-level library built around algorithms

```
for_each(begin, end, function);
```



Operator

A named pattern
of computation and
communication.



Data

One or more
collections to
operate on.



Function

Caller-provided
function object
injected in pattern.

PARALLEL STL

Algorithms + Execution Policies

```
for_each(par, begin, end, function);
```



Execution Policy

Specify *how* operation
may execute.

PARALLEL ALGORITHMS

Parallelizable algorithms in STL

`for_each`

`transform`

`copy_if`

`sort`

`set_intersection`

etc.

New additions for parallelism

`reduce`

`exclusive_scan`

`inclusive_scan`

`transform_reduce`

`transform_inclusive_scan`

`transform_exclusive_scan`

EXECUTION POLICIES

Specify how algorithms may execute

POLICY NAME	MEANING
<code>seq</code>	Sequential execution alone is permitted.
<code>par</code>	Parallel execution is permitted.
<code>par_vec</code>	Vectorized parallel execution is permitted.

parallel :: provided function objects can be executed in any order on one or more threads.

vectorized :: provided function objects can be also be interleaved when on one thread.

EXECUTION POLICIES

Provide opportunities for customization

Standard
Policies

POLICY NAME	MEANING
<code>seq</code>	Sequential execution alone is permitted.
<code>par</code>	Parallel execution is permitted.
<code>par_vec</code>	Vectorized parallel execution is permitted.

Custom
Policies
(non-standard)

<code>cuda::par</code>	Always choose a CUDA-capable GPU.
<code>on_some_fpga</code>	Permit execution on FPGA where possible.
<code>vendors_policy</code>	Custom semantics not listed above.



Portable, High-level Parallel Code TODAY

- ▶ Thrust library allows the same C++ code to target both:
 - ▶ NVIDIA GPUs
 - ▶ x86, ARM and POWER CPUs



- ▶ Thrust was the inspiration for a proposal to the ISO C++ Committee
- ▶ Committee voted unanimously to accept as official tech. specification working draft

N3960 Technical Specification Working Draft:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3960.pdf>

Prototype:

<https://github.com/n3554/n3554>

A Parallel Algorithms Library | N3724

Jared Hoberock Jaydeep Marathe Michael Garland Olivier Giroux
Vinod Grover {jhoberock, jmarathe, mgarland, ogiroux, vgrover}@nvidia.com
Artur Laksberg Herb Sutter {arturl, hsutter}@microsoft.com Arch Robison

Document Number: N3960
Date: 2014-02-28
Reply to: Jared Hoberock
NVIDIA Corporation
jhoberock@nvidia.com

**Working Draft, Technical
Specification for C++ Extensions for
Parallelism, Revision 1**

THRUST LIBRARY

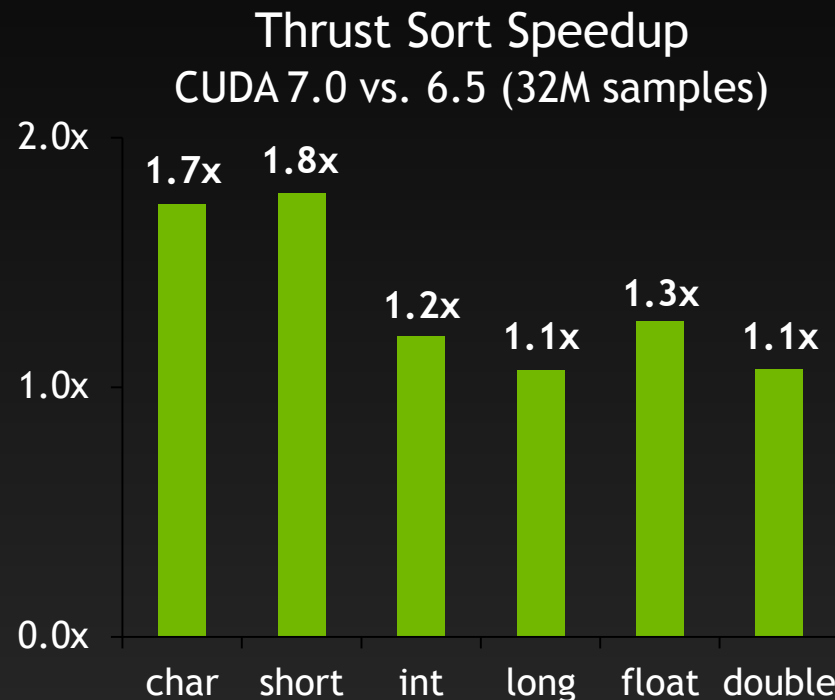
Programming with algorithms and policies today

Bundled with NVIDIA's CUDA Toolkit

Supports execution on GPUs and CPUs

Ongoing performance & feature improvements

Functionality beyond Parallel STL



From *CUDA 7.0 Performance Report*.

Run on K40m, ECC ON, input and output data on device

Performance may vary based on OS and software versions, and motherboard configuration

Thrust



- C++ template library for CUDA
 - Mimics Standard Template Library (STL)
- Containers
 - `thrust::host_vector<T>`
 - `thrust::device_vector<T>`
- Algorithms
 - `thrust::sort()`
 - `thrust::reduce()`
 - `thrust::inclusive_scan()`
 - etc.

Thrust



```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>

int main(void)
{
    // generate 16M random numbers on the host
    thrust::host_vector<int> h_vec(1 << 24);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```

What is CUDA?



- **CUDA Architecture**
 - Expose GPU parallelism for general-purpose computing
 - Retain performance
- **CUDA C/C++**
 - Based on industry-standard C/C++
 - Small set of extensions to enable heterogeneous programming
 - Straightforward APIs to manage devices, memory etc.
- **This session introduces CUDA C/C++**

Introduction to CUDA C/C++



- What will you learn in this session?
 - Start from “Hello World!”
 - Write and launch CUDA C/C++ kernels
 - Manage GPU memory
 - Manage communication and synchronization

Prerequisites



- You (probably) need experience with C or C++
- You don't need GPU experience
- You don't need parallel programming experience
- You don't need graphics experience

CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices



CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

__syncthreads()

Asynchronous operation

Handling errors

Managing devices

HELLO WORLD!

Memory Management



- Host and device memory are separate entities
 - **Device** pointers point to GPU memory
 - May be passed to/from host code
 - May *not* be dereferenced in host code
 - **Host** pointers point to CPU memory
 - May be passed to/from device code
 - May *not* be dereferenced in device code
- Simple CUDA API for handling device memory
 - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
 - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`



Some Terminology



- **Kernel** - A function which runs on the GPU
 - A kernel is launched on a **grid** of **thread blocks**.
 - The grid and block size are called the **launch configuration**.
- **Global Memory** - GPU's on-board DRAM
- **Shared Memory** - On-chip fast memory local to a thread block

Hello World!



```
int main(void) {  
    printf("Hello World!\n");  
    return 0;  
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

Output:

```
$ nvcc  
hello_world.cu  
$ a.out  
Hello World!  
$
```

Hello World! with Device Code



```
__global__ void mykernel(void) {  
}
```

```
int main(void) {  
    mykernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

- Two new syntactic elements...

Hello World! with Device Code



```
__global__ void mykernel(void) {  
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
 - Runs on the device
 - Is called from host code
- `nvcc` separates source code into host and device components
 - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
 - Host functions (e.g. `main()`) processed by standard host compiler
 - `gcc`, `cl.exe`

Hello World! with Device C0de



```
mykernel<<<1,1>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
 - Also called a “kernel launch”
 - We’ll return to the parameters (1,1) in a moment
- That’s all that is required to execute a function on the GPU!

Hello World! with Device Code



```
__global__ void mykernel(void) {  
}
```

```
int main(void) {  
    mykernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

Output:

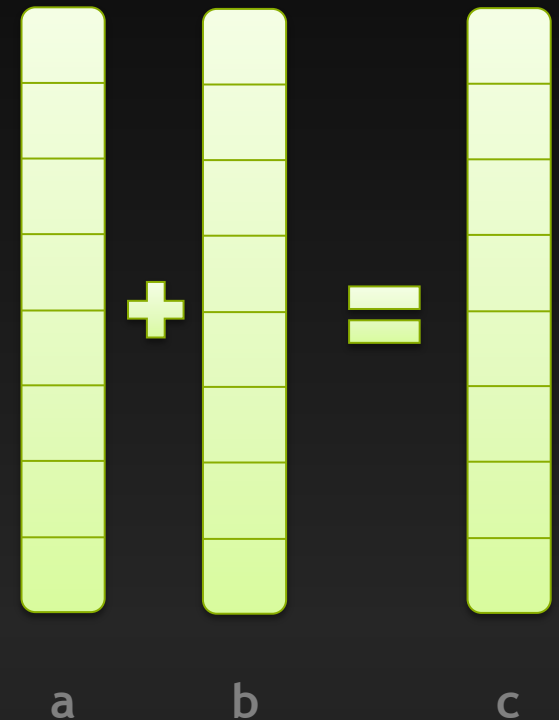
```
$ nvcc hello.cu  
$ a.out  
Hello World!  
$
```

- `mykernel()` does nothing, somewhat anticlimactic!

Parallel Programming in CUDA C/C++



- But wait... GPU computing is about massive parallelism!
- We need a more interesting example...
- We'll start by adding two integers and build up to vector addition



Vector Add: GPU's Hello World



- GPU is *parallel computation* oriented.
 - Vector add is a very simple parallel algorithm.
- **Problem: $C = A + B$**
 - C, A, B are length N vectors

```
void vecAdd(int n, float * a,  
            float * b, float * c)  
{  
    for(int i=0; i<n; i++)  
    {  
        c[i] = a[i] + b[i];  
    }  
}
```

Vector Add: GPU's Hello World



- GPU is *parallel computation* oriented.
 - Vector add is a very simple parallel algorithm.
- **Problem: $C = A + B$**
 - C, A, B are length N vectors

```
void vecAdd(int n, float * a,  
            float * b, float * c)  
{  
    for(int i=0; i<n; i++)  
    {  
        c[i] = a[i] + b[i];  
    }  
}
```

```
void main()  
{  
    int N = 1024;  
    float * a, *b, *c;  
    a = (float*)malloc(N*sizeof(float));  
    b = (float*)malloc(N*sizeof(float));  
    c = (float*)malloc(N*sizeof(float));  
    memset(c, 0, N*sizeof(float));  
    init_rand_f(a, N);  
    init_rand_f(b, N);  
  
    vecAdd(N, a, b, c);  
}
```

Moving Computation to the GPU



- Step 1: Identify parallelism.
 - Design problem decomposition
- Step 2: Write your GPU Kernel
- Step 3: Setup the Problem
- Step 4: Launch the Kernel
- Step 5: Copy results back from GPU

Remember: big font means important

Vector Add



- **Step 1: Parallelize**

- **Identify parallelism.**

- $c[i]$ depends only on $a[i]$ and $b[i]$.
 - $c[i]$ is not used by any other calculation
 - $c[i]$ can be computed in parallel

- **Assign Units of Work**

- Each thread will compute one element of c .
 - Will use a 1D grid of 1D threadblocks.

```
void vecAdd(int n, float * a,  
            float * b, float * c)  
{  
    for(int i=0; i<n; i++)  
    {  
        c[i] = a[i] + b[i];  
    }  
}
```

CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

__syncthreads()

Asynchronous operation

Handling errors

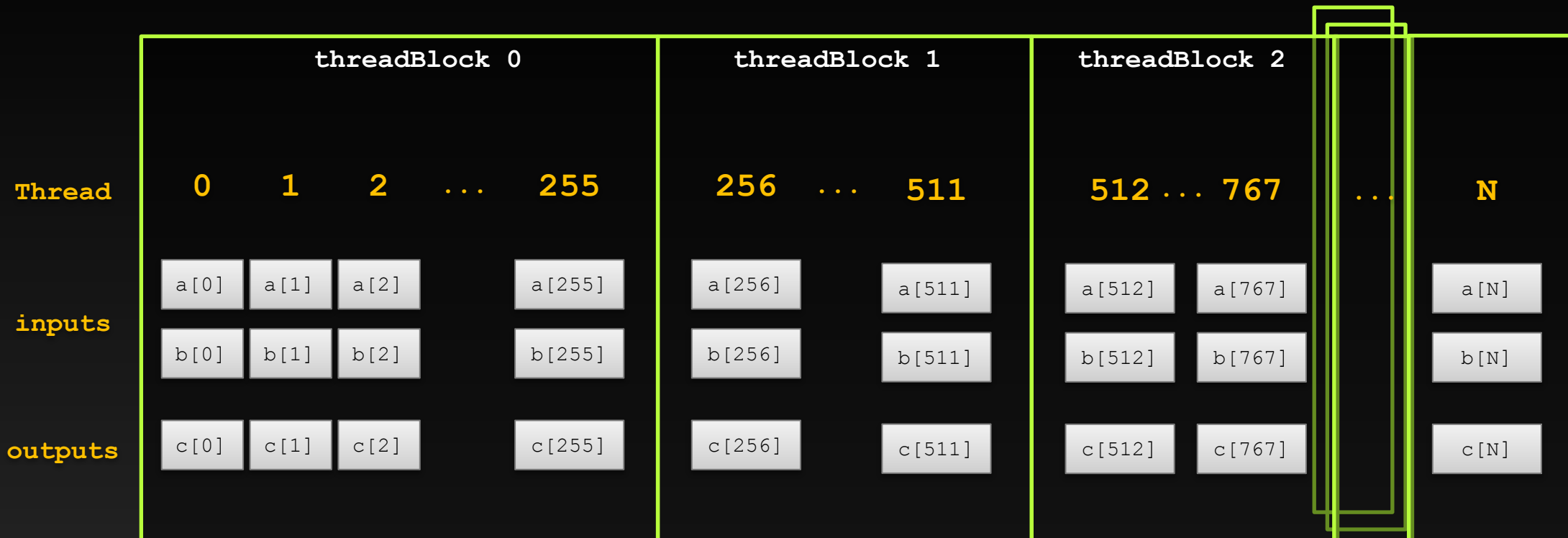
Managing devices

RUNNING IN PARALLEL BLOCKS & THREADS

Parallelization of VecAdd



Parallelization of VecAdd



IDs and Dimensions

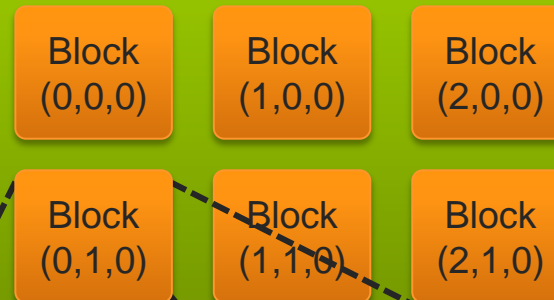


- Built-in variables:

- `threadIdx.[x y z]`
 - thread index within a thread block
- `blockIdx.[x y z]`
 - block index within the grid.
- `blockDim.[x y z]`
 - Number of threads in each block.
- `gridDim.[x y z]`
 - Number of blocks in the grid.

Device

Grid



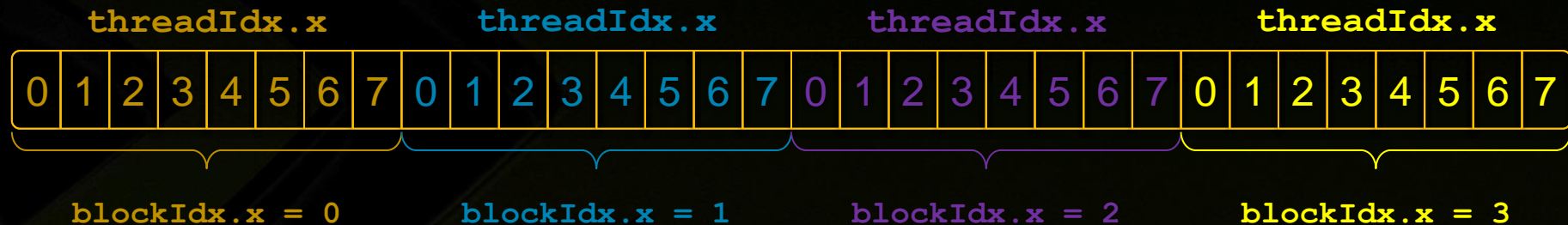
Block (1,1,0)

Thread (0,0,0)	Thread (1,0,0)	Thread (2,0,0)	Thread (3,0,0)	Thread (4,0,0)
Thread (0,1,0)	Thread (1,1,0)	Thread (2,1,0)	Thread (3,1,0)	Thread (4,1,0)
Thread (0,2,0)	Thread (1,2,0)	Thread (2,2,0)	Thread (3,2,0)	Thread (4,2,0)

Indexing Arrays with Blocks and Threads



- No longer as simple as using `blockIdx.x` and `threadIdx.x`
 - Consider indexing an array with one element per thread (8 threads/block)



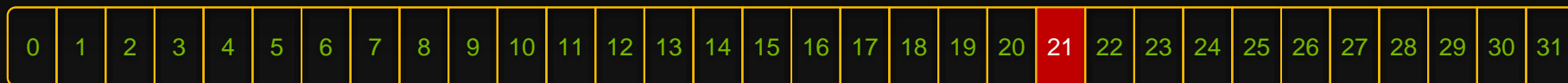
- With **blockDim.x** threads per block, a unique index for each thread is given by:

```
int index = blockIdx.x * blockDim.x + threadIdx.x;
```

Indexing Arrays: Example

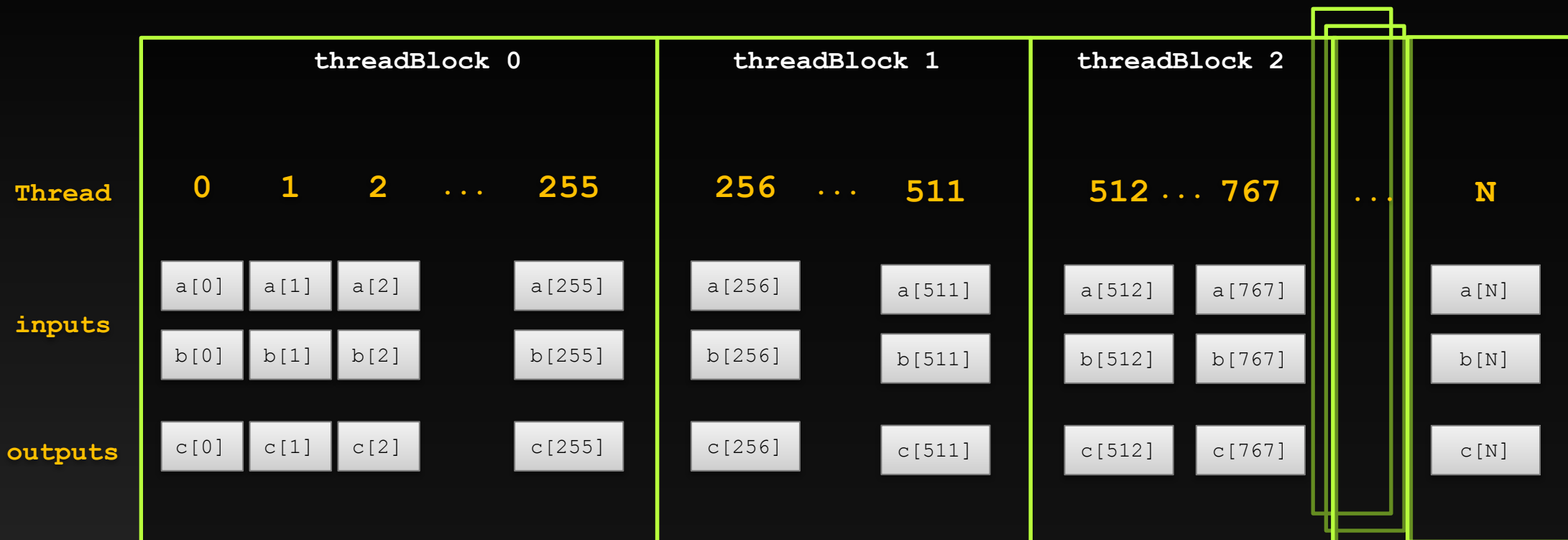


- Which thread will operate on the red element?



```
int index = blockIdx.x * blockDim.x + threadIdx.x;  
          = 2 * 8 + 5;  
          = 21;
```

Parallelization of VecAdd



work index $i = \text{threadIdx.x} + \text{blockIdx.x} * \text{blockDim.x};$

index of the thread within
a thread block

index of the threadblock
within the grid

number of threads within
each block

Vector Add: GPU's Hello World



- Step 2: Make it a GPU Kernel

Identify this function as something to be run on the GPU.

i is a different value for each thread.

```
__global__ void vecAdd(int n, float * a, float * b, float * c)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i < n){
        c[i] = a[i] + b[i];
    }
}
```

Protect against invalid access if too many threads are launched.

Step 3: Setup the problem



```
void main()
{
    int N = 1024;
    float *a, *b, *c;
    float *devA, *devB, *devC;
    a = (float*)malloc(N*sizeof(float));
    b = (float*)malloc(N*sizeof(float));
    c = (float*)malloc(N*sizeof(float));
    cudaMalloc(&devA, N*sizeof(float));
    cudaMalloc(&devB, N*sizeof(float));
    cudaMalloc(&devC, N*sizeof(float));

    memset(c, 0, N*sizeof(float));
    init_rand_f(a, N);
    init_rand_f(b, N);
    cudaMemcpy(devA, a, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(devB, b, N*sizeof(float), cudaMemcpyHostToDevice);
}
```

Pointers to storage on the GPU.

Allocate memory in the GPU Global Memory.

Copy data to the GPU.

Step 4: Launch the GPU Kernel



call function by name
as usual

Angle Brackets: Specify
launch configuration
for the kernel.

Normal parameter passing
syntax. Note that *devA*,
devB, and *devC* are
device pointers.
They point to memory
allocated on the GPU.

```
void main()
{
    ...
    vecAdd<<<(N+127)/128, 128>>>(N, devA, devB, devC);
    ...
}
```

First argument is
the number of
thread blocks
(rounding up)

Second argument is
the shape of
(i.e, number of threads in)
each **thread block**

Step 5: Copy data back.



```
void main()
{
    ...

    cudaMemcpy(c, devC, N*sizeof(float), cudaMemcpyDeviceToHost);

    ...
}
```


Handling Arbitrary Vector Sizes



- Typical problems are not even multiples of `blockDim.x`
- Avoid accessing beyond the end of the arrays:

```
__global__ void add(int *a, int *b, int *c, int n) {  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
    if (index < n)  
        c[index] = a[index] + b[index];  
}
```

- Update the kernel launch:

```
add<<<(N + M-1) / M,M>>>(a, b, c, N);
```

- `N` = problem size, `M` = block size

Multi-Dimensional Thread-Blocks



- CUDA supports up to 3-dimensional grids and blocks.
 - blockIdx.{x,y,z}
 - threadIdx.{x,y,z}
 - blockDim.{x,y,z}
 - gridDim.{x,y,z}
- Easily accelerate multiple nested loops

Order of loops is important for performance. Ideally threadIdx.x is consecutive in memory

```
for(int y=...) -> int y=blockIdx.y*blockDim.y+threadIdx.y;  
    for(int x=...) -> int x =blockIdx.x*blockDim.x+threadIdx.x;
```

- Launch a 2D grid using dim3 structures
 - kernel<<<dim3(32,32),dim3(32,4)>>>()

Blocks Size Guidelines



- Block size cannot be larger than 1024 (2048 Kepler) threads
 - Block size = `blockDim.x * blockDim.y * blockDim.z`
- For performance
 - Block size should be divisible by 32
 - Have at least 128 threads per block
- Tip:
 - start with 128 threads per block and tune up by increments of 32 if necessary

Threads vs Blocks



- Distinction is not clear from previous example
- Threads within a block can:
 - Communicate
 - Synchronize
- New example to illuminate this subject.

CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices

COOPERATING THREADS

Shared Memory



- Terminology: within a **block**, threads share data via **shared memory**
- Extremely fast **on-chip** memory
 - Bandwidth > 1 TB, latency ~ 10's of cycles
 - **Global memory**: Bandwidth ~ 250 GB/s, latency ~ 100's of cycles
- Common Uses:
 - User-managed cache
 - Sharing Data between threads
- Declare within a kernel using **__shared__**, allocated per block
 - Threads in the same block see the same memory
 - Threads in different blocks see different memory

Block Reduce



- Can such a kernel be parallelized?

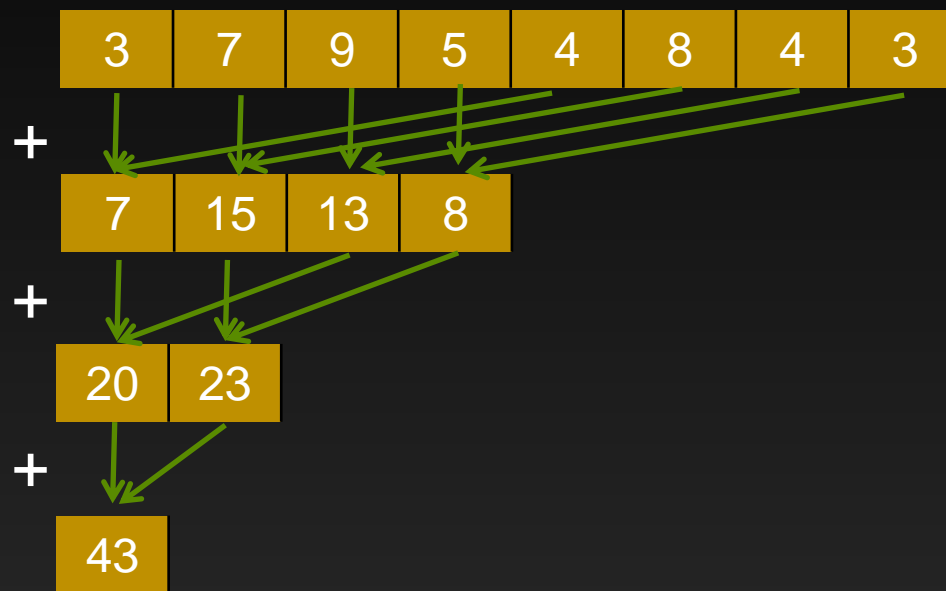
```
float sum = 0.0f;  
for(int i=0;i<N;i++) {  
    float v=a[i];  
    sum+=v*v;  
}  
sum=sqrtf(sum);
```

Block Reduce



- Can such a kernel be parallelized?

```
float sum = 0.0f;  
for(int i=0;i<N;i++) {  
    float v=a[i];  
    sum+=v*v;  
}  
sum=sqrtf(sum);
```



Requires communication between threads

Reduce + Atomic



Original Array



Block Reduce



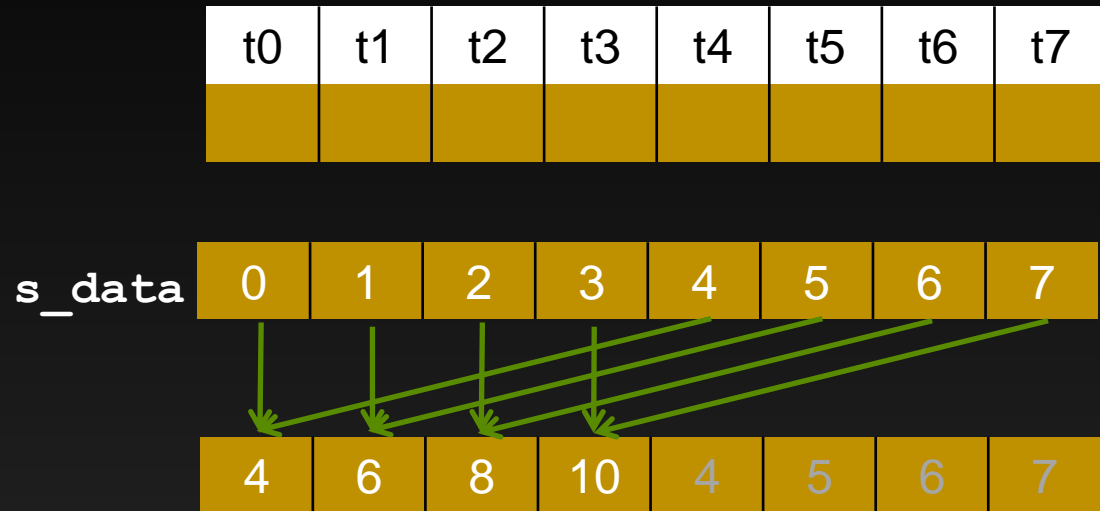
Final Result



Shared Memory Example

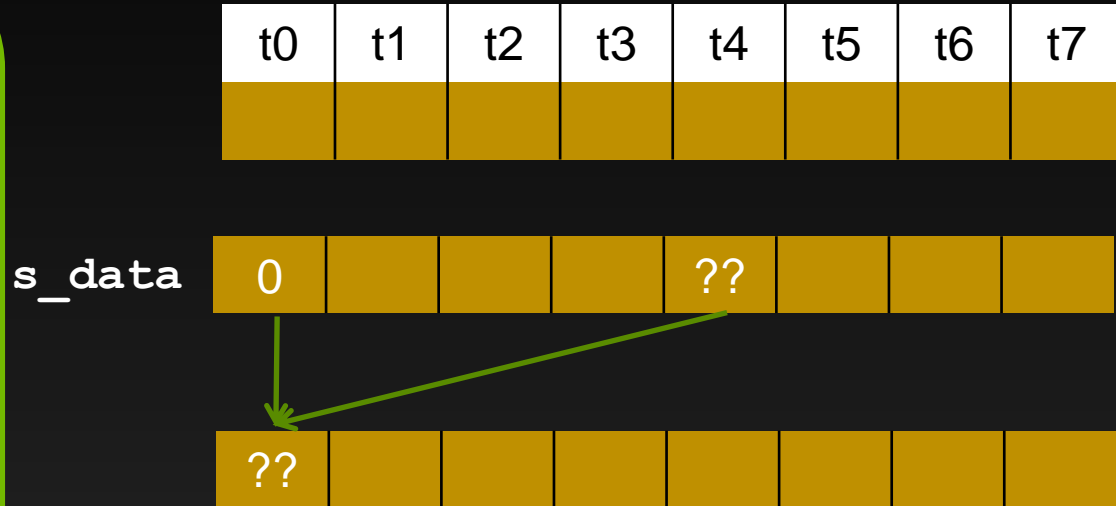


```
__global__ void kernel() {  
    int tid=threadIdx.x;  
    __shared__ s_data[BLOCK_SIZE];  
  
    s_data[tid]=tid;  
  
    int s=BLOCK_SIZE/2;  
    if(tid<s)  
        s_data[tid]+=s_data[tid+s];  
    ...  
}
```



Race Conditions

```
__global__ void kernel() {  
    int tid=threadIdx.x;  
    __shared__ s_data[BLOCK_SIZE];  
  
    s_data[tid]=tid;  
  
    int s=BLOCK_SIZE/2;  
    if(tid<s)  
        s_data[tid]+=s_data[tid+s];  
    ...  
}
```



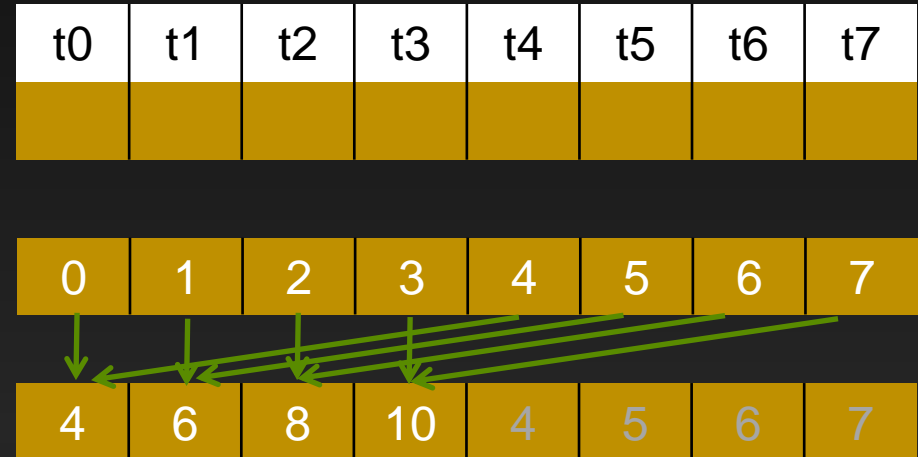
Synchronization



- The programming model does not guarantee thread execution order
- Enforce order via **__syncthreads()**
 - All threads in a block must call **__syncthreads()**

```
__global__ void kernel() {  
    int tid=threadIdx.x;  
    __shared__ s_data[BLOCK_SIZE];  
  
    s_data[tid]=tid;  
    __syncthreads();  
    int s=BLOCK_SIZE/2;  
    if(tid<s)  
        s_data[tid]+=s_data[tid+s];  
    ...  
}
```

s_data

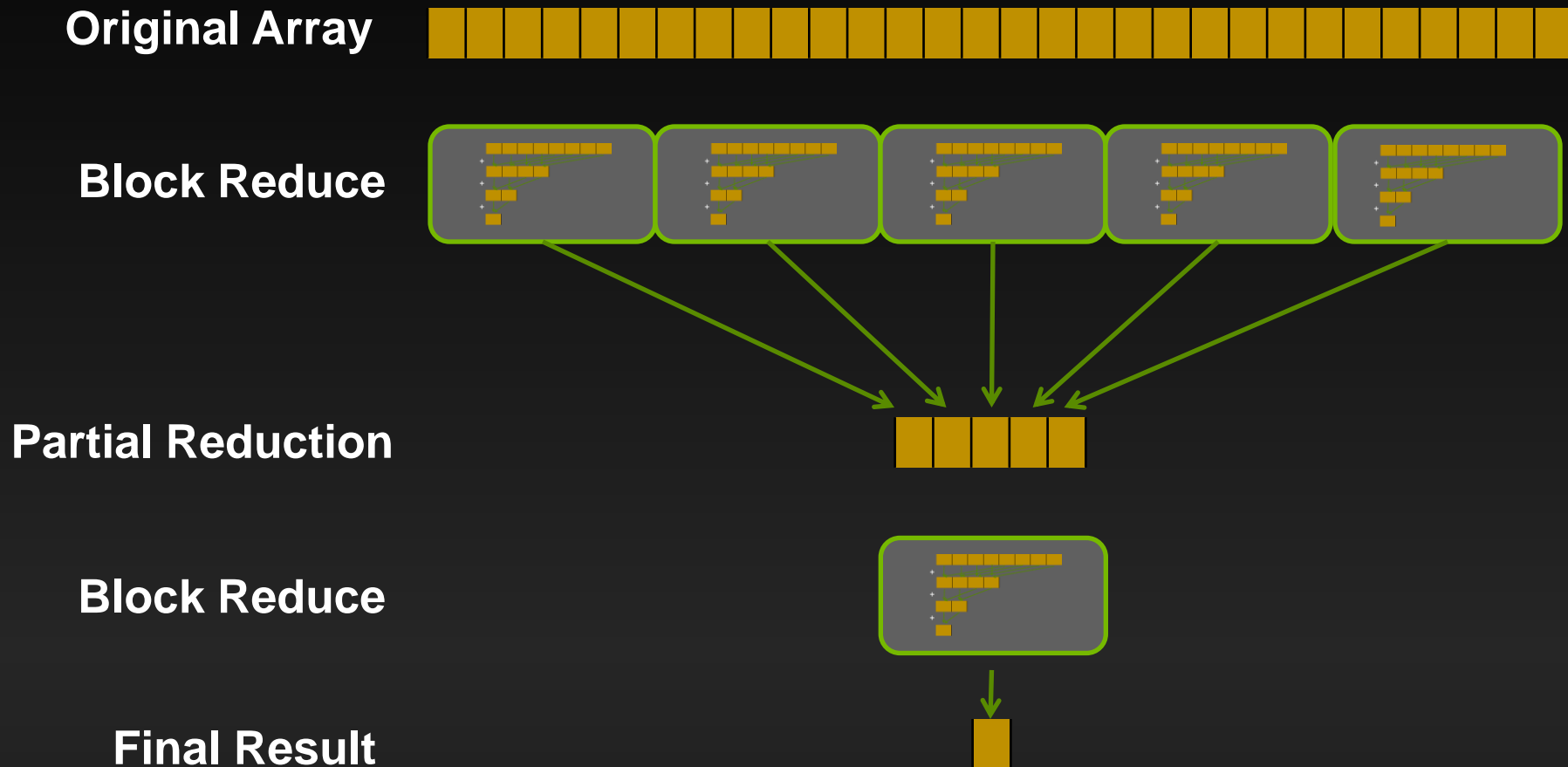


Parallel Reduction Across Blocks



- We should now be able to reduce within a block
 - But how do we reduce across multiple blocks?
- **CUDA: Blocks must be completely independent**
 - No synchronization
- **Reduce Across Blocks Options:**
 - Reduce within block and update atomically
 - Reduce within block, save partial results, repeat with new kernel

Reduce + Reduce



1D Stencil



- Consider applying a 1D stencil to a 1D array of elements
 - Each output element is the sum of input elements within a radius
- If radius is 3, then each output element is the sum of 7 input elements:



Implementing Within a Block



- Each thread processes one output element
 - `blockDim.x` elements per block
- Input elements are read several times
 - With radius 3, each input element is read seven times



Sharing Data Between Threads

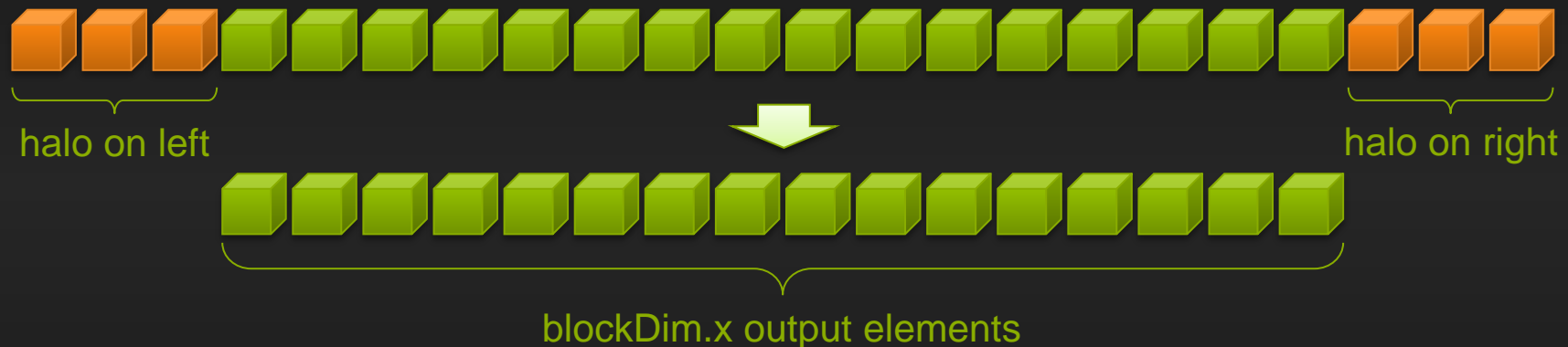


- Terminology: within a block, threads share data via **shared memory**
- Extremely fast on-chip memory, user-managed
- Declare using **__shared__**, allocated per block
- Data is not visible to threads in other blocks

Implementing With Shared Memory



- Cache data in shared memory
 - Read $(\text{blockDim.x} + 2 * \text{radius})$ input elements from global memory to shared memory
 - Compute blockDim.x output elements
 - Write blockDim.x output elements to global memory
- Each block needs a **halo** of radius elements at each boundary





Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {  
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];  
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;  
    int lindex = threadIdx.x + RADIUS;  
  
    // Read input elements into shared memory  
    temp[lindex] = in[gindex];  
    if (threadIdx.x < RADIUS) {  
        temp[lindex - RADIUS] = in[gindex - RADIUS];  
        temp[lindex + BLOCK_SIZE] =  
            in[gindex + BLOCK_SIZE];  
    }  
}
```



Stencil Kernel





```
// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

// Store the result
out[gindex] = result;
}
```

Data Race!



- The stencil example will not work...
- Suppose thread 15 reads the halo before thread 0 has fetched it...

```
temp[lindex] = in[gindex];      Store at temp[18]    
if (threadIdx.x < RADIUS) {  
    temp[lindex - RADIUS] = in[gindex - RADIUS];    Skipped, threadIdx > RADIUS  
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];  
}  
int result = 0;  
result += temp[lindex + 1];    Load from temp[19]  
```

__syncthreads()



- `void __syncthreads();`
- Synchronizes all threads within a block
 - Used to prevent RAW / WAR / WAW hazards
- All threads must reach the barrier
 - In conditional code, the condition must be uniform across the block

Stencil Kernel



```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();
}
```

Stencil Kernel



```
// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

// Store the result
out[gindex] = result;
}
```


Review (1 of 2)



- Launching parallel threads
 - Launch N blocks with M threads per block with `kernel<<<N,M>>> (...)` ;
 - Use `blockIdx.x` to access block index within grid
 - Use `threadIdx.x` to access thread index within block
- Allocate elements to threads:

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

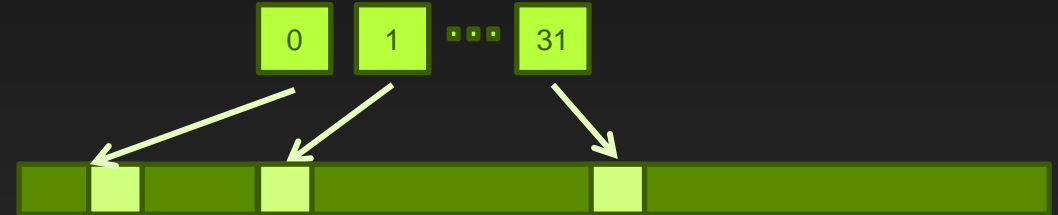
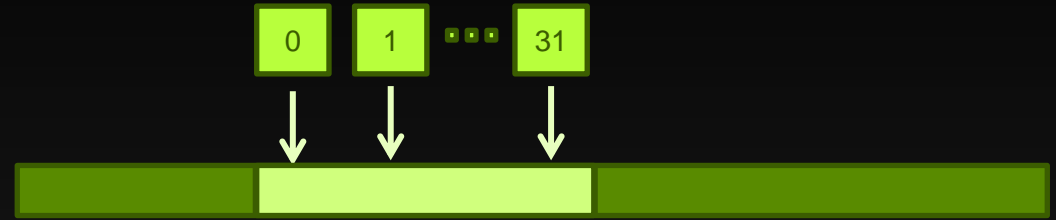
Review (2 of 2)



- Use `__shared__` to declare a variable/array in shared memory
 - Data is shared between threads in a block
 - Not visible to threads in other blocks
- Use `__syncthreads()` as a barrier
 - Use to prevent data hazards

What is an Uncoalesced Global Load?

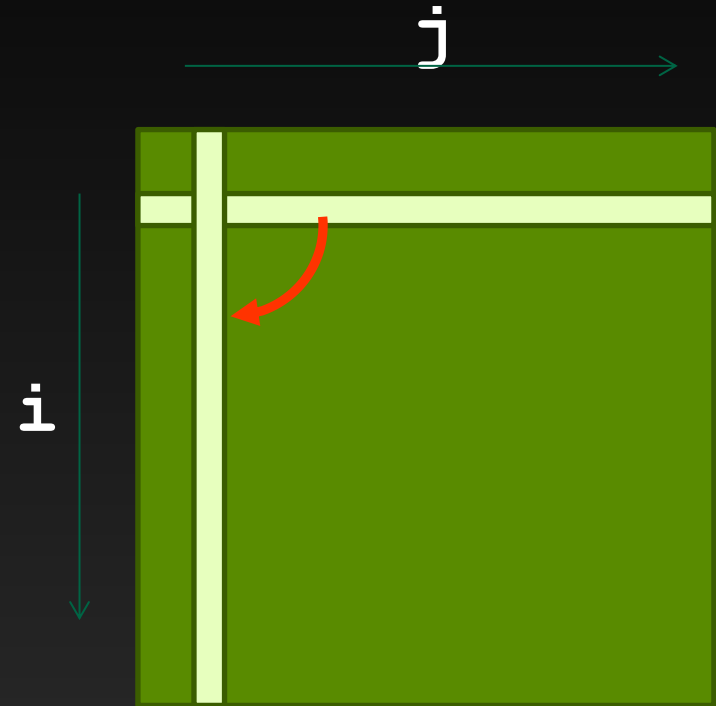
- Global memory access happens in transactions of 32 or 128 bytes
- *Coalesced* access:
 - A group of 32 contiguous threads (“warp”) accessing adjacent words
 - Few transactions and high utilization
- *Uncoalesced* access:
 - A warp of 32 threads accessing scattered words
 - Many transactions and low utilization



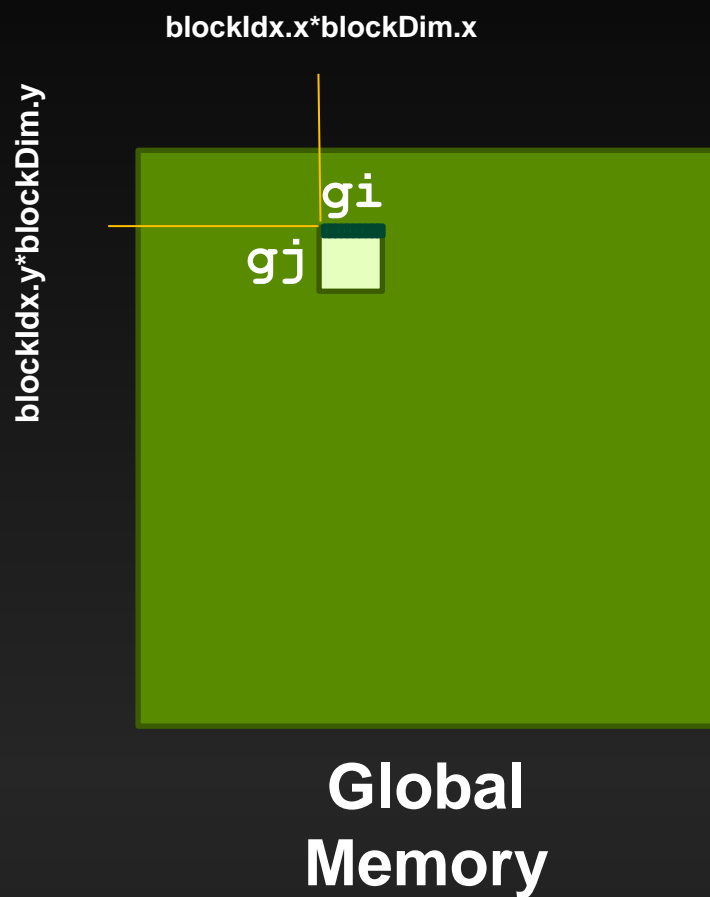
Transpose Memory Coalescing



- Either the read or the write will be uncoalesced
- Two options
 1. Use texture for loads
 2. Modify indexing to eliminate uncoalesced accesses
 - Fastest changing dimension should be threadIdx.x
 - Need to stage through shared memory



Optimal Global Memory Access

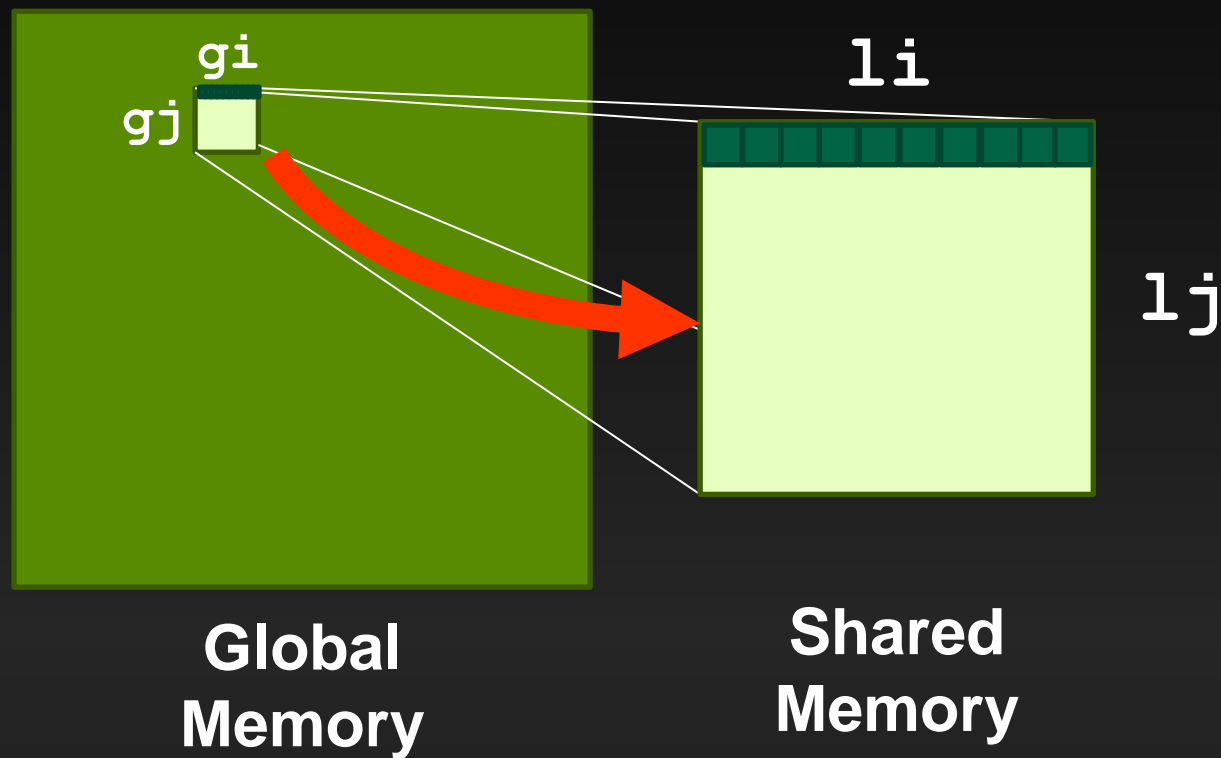


1. Load global (g_i, g_j)

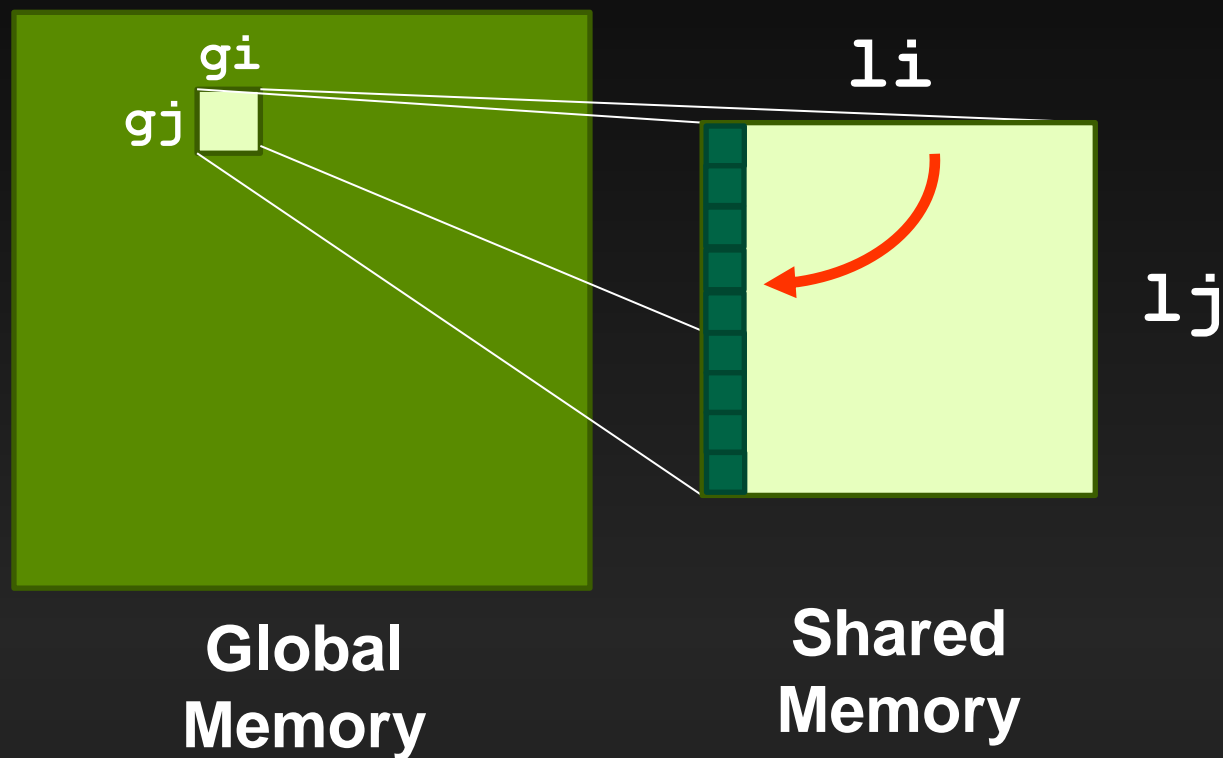
Optimal Global Memory Access



1. Load global (g_i, g_j)
2. Store shared (l_i, l_j)

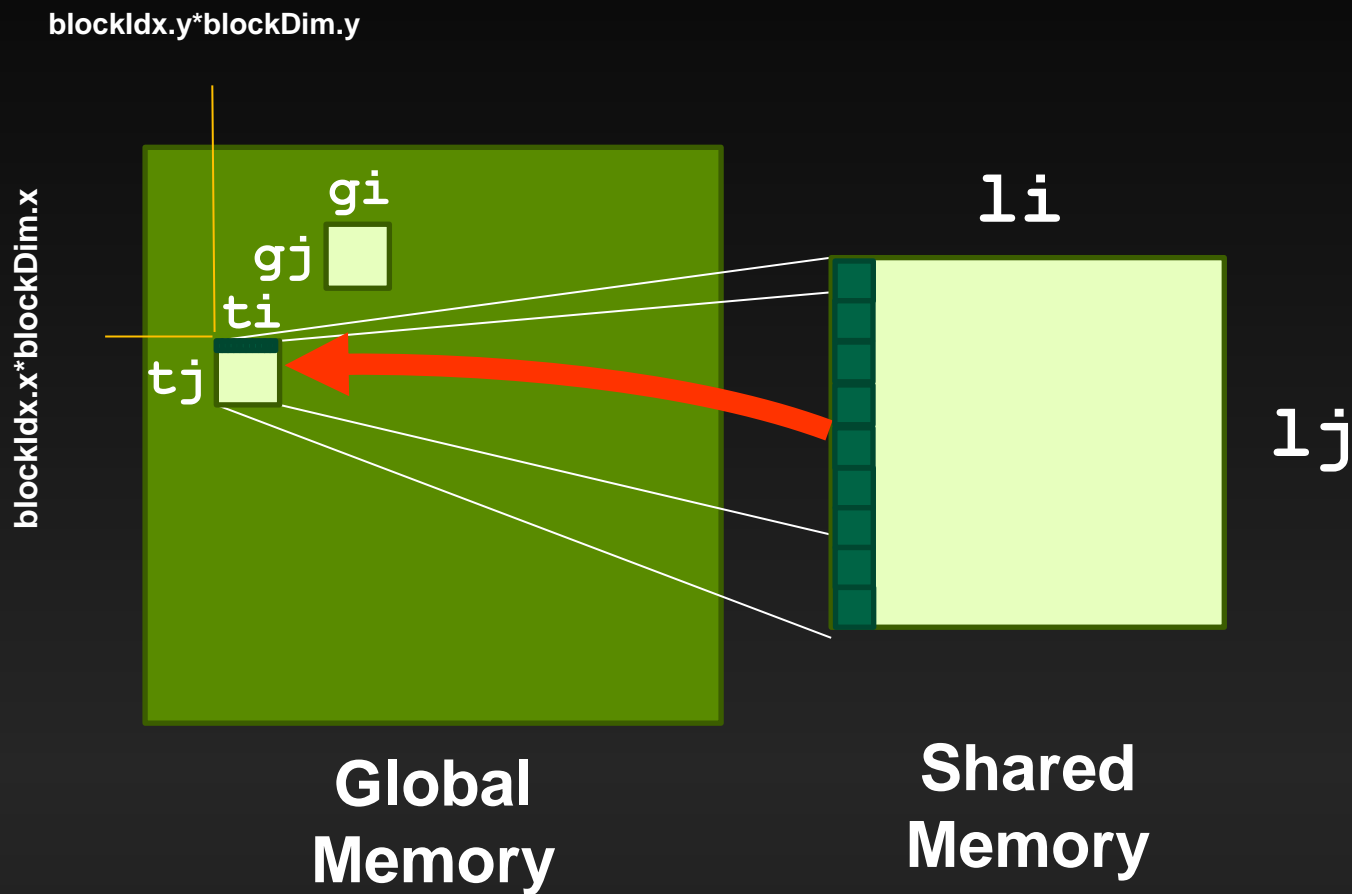


Optimal Global Memory Access



1. Load global (g_i, g_j)
2. Store shared (l_i, l_j)
3. Load shared transposed (l_j, l_i)

Optimal Global Memory Access



1. Load global (g_i, g_j)
2. Store shared (l_i, l_j)
3. Load shared transposed (l_j, l_i)
4. Store global (t_i, t_j)



CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices

MANAGING THE DEVICE

Coordinating Host & Device



- Kernel launches are **asynchronous**
 - Control returns to the CPU immediately
- CPU needs to synchronize before consuming the results

`cudaMemcpy()`

Blocks the CPU until the copy is complete
Copy begins when all preceding CUDA calls have completed

`cudaMemcpyAsync()`

Asynchronous, does not block the CPU

`cudaDeviceSynchronize()`

Blocks the CPU until all preceding CUDA calls have completed

Reporting Errors



- All CUDA API calls return an error code (`cudaError_t`)
 - Error in the API call itself
 - OR
 - Error in an earlier asynchronous operation (e.g. kernel)

- Get the error code for the last error:

```
cudaError_t cudaGetLastError(void)
```

- Get a string to describe the error:

```
char *cudaGetErrorString(cudaError_t)
```

```
printf("%s\n", cudaGetErrorString(cudaGetLastError()));
```

Device Management



- Application can query and select GPUs

```
cudaGetDeviceCount(int *count)
```

```
cudaSetDevice(int device)
```

```
cudaGetDevice(int *device)
```

```
cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
```

- Multiple threads can share a device
- A single thread can manage multiple devices

```
cudaSetDevice(i) to select current device
```

```
cudaMemcpy(...) for peer-to-peer copies†
```

[†] requires OS and device support

Introduction to CUDA C/C++



- What have we learned?
 - Write and launch CUDA C/C++ kernels
 - `__global__`, `blockIdx.x`, `threadIdx.x`, `<<<>>>`
 - Manage GPU memory
 - `cudaMalloc()`, `cudaMemcpy()`, `cudaFree()`
 - Manage communication and synchronization
 - `__shared__`, `__syncthreads()`
 - `cudaMemcpy()` VS `cudaMemcpyAsync()`, `cudaDeviceSynchronize()`

Compute Capability



- The **compute capability** of a device describes its architecture, e.g.
 - Number of registers
 - Sizes of memories
 - Features & capabilities

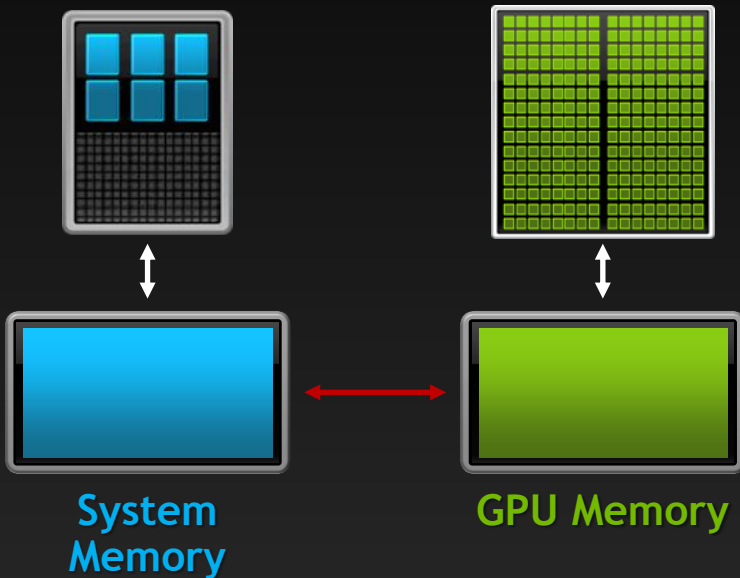
Compute Capability	Selected Features (see CUDA C Programming Guide for complete list)	Tesla models
1.0	Fundamental CUDA support	870
1.3	Double precision, improved memory accesses, atomics	10-series
2.0	Caches, fused multiply-add, 3D grids, surfaces, ECC, P2P, concurrent kernels/copies, function pointers, recursion	20-series

- The following presentations concentrate on Fermi and Kepler devices
 - Compute Capability ≥ 2.0

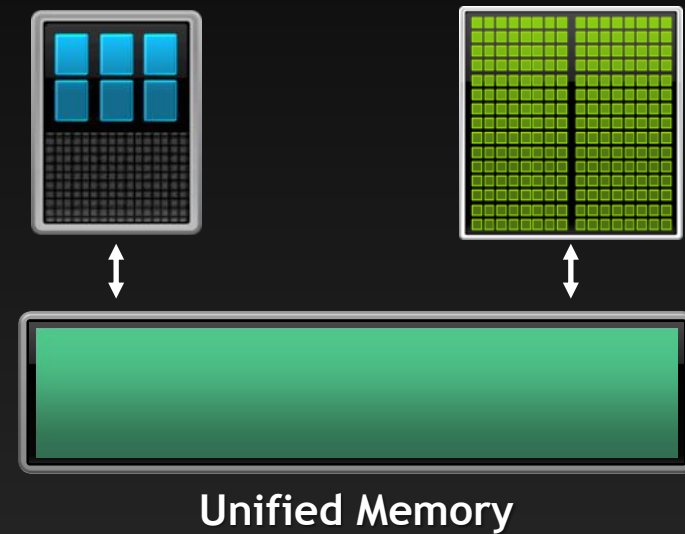
CUDA Memory Management



No Unified Memory



Unified Memory

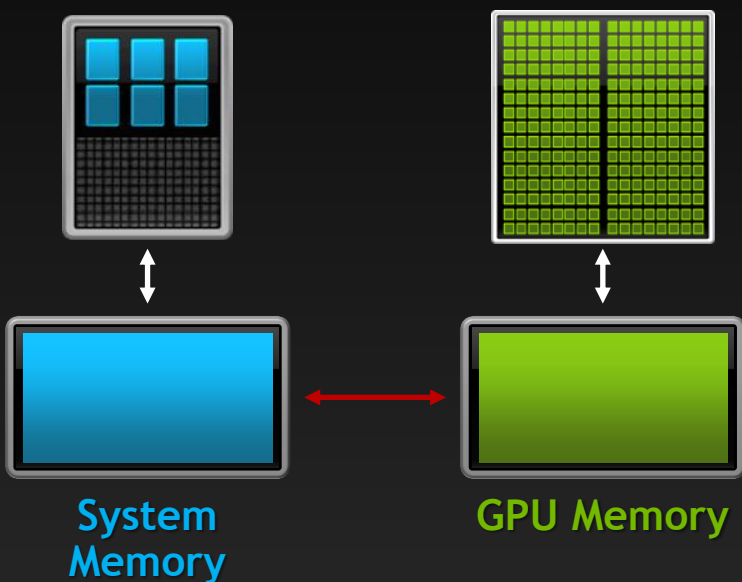


Requirements
Cuda 6+
Kepler+

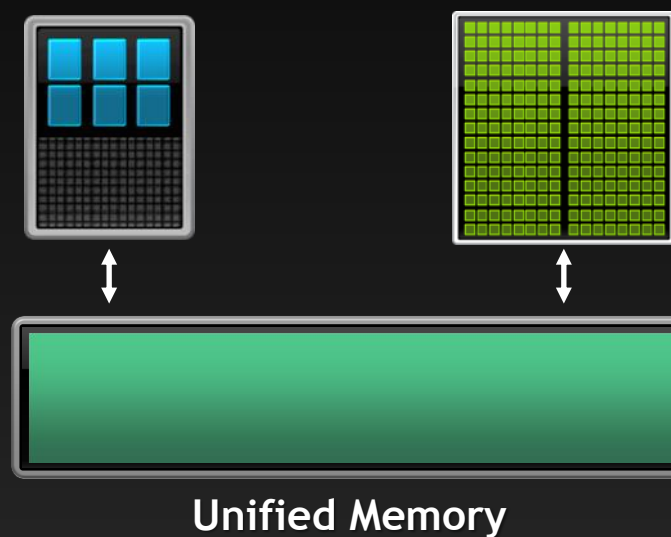
Unified Memory

Dramatically Lower Developer Effort

Developer View Today



Developer View With Unified Memory



Super Simplified Memory Management Code



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 Delivers



1. Simpler Programming & Memory Model

- Single pointer to data, accessible anywhere
- Tight language integration
- Greatly simplifies code porting

2. Performance Through Data Locality

- Migrate data to accessing processor
- Guarantee global coherency
- Still allows *cudaMemcpyAsync()* hand tuning

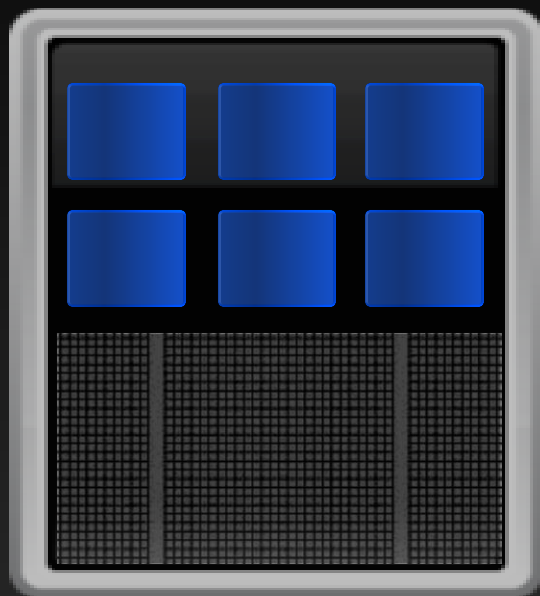


Questions?



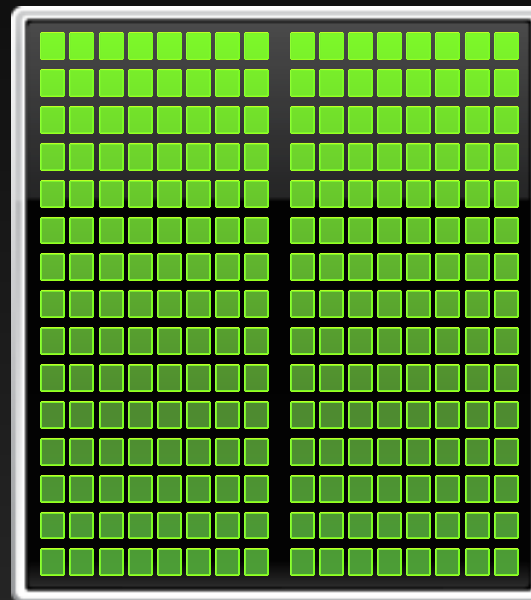
Add GPUs: Accelerate Science Applications

CPU

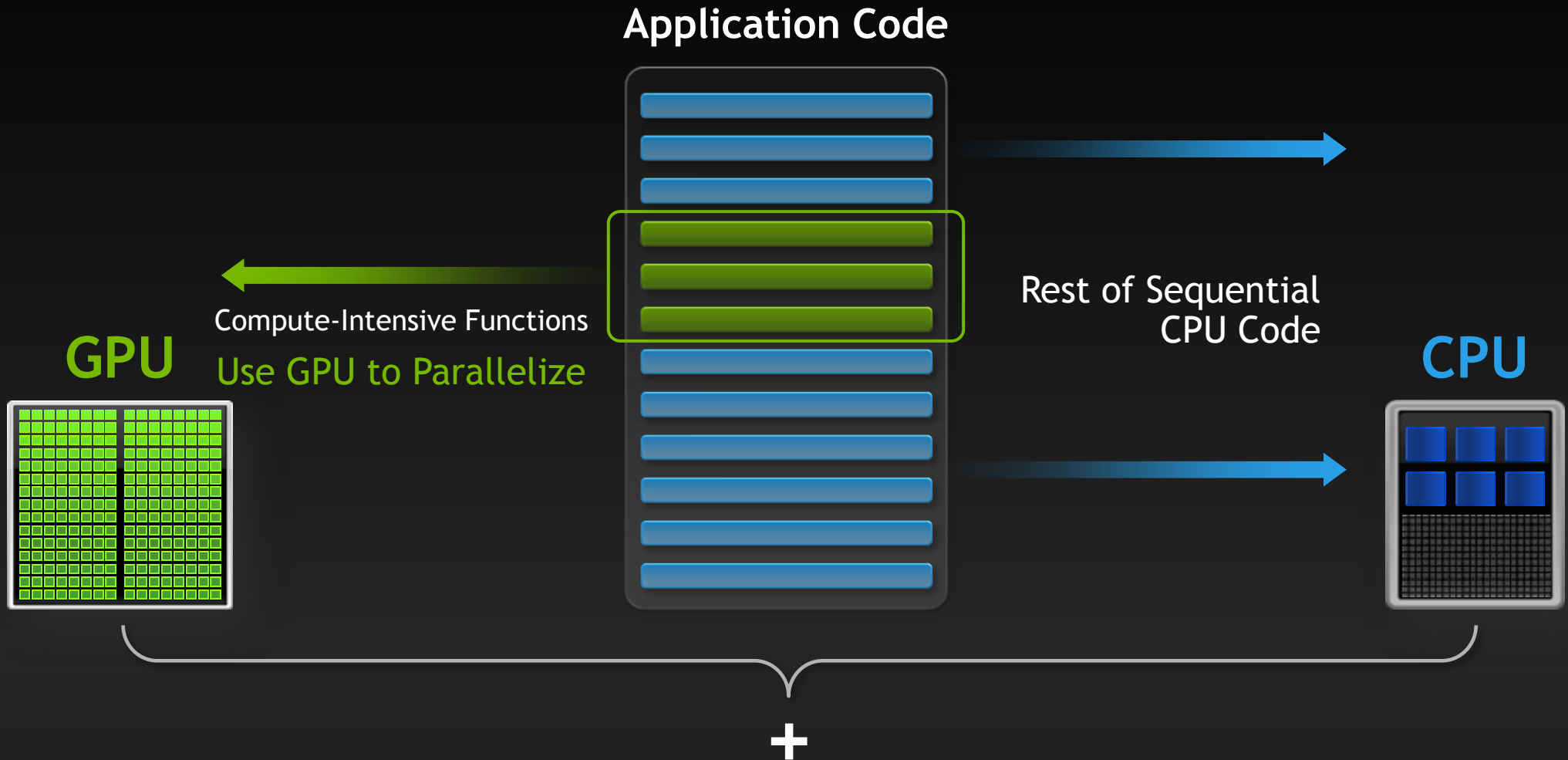


+

GPU



Small Changes, Big Speed-up



Libraries: Easy, High-Quality Acceleration

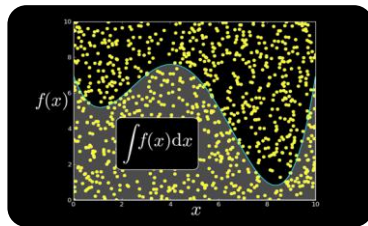


- **Ease of use:** Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
- **“Drop-in”:** Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- **Quality:** Libraries offer high-quality implementations of functions encountered in a broad range of applications
- **Performance:** NVIDIA libraries are tuned by experts

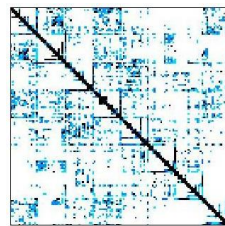
Some GPU-accelerated Libraries



NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE



NVIDIA NPP

GPU VSIPL

Vector Signal
Image Processing

CULA | tools

GPU Accelerated
Linear Algebra



Matrix Algebra on
GPU and Multicore



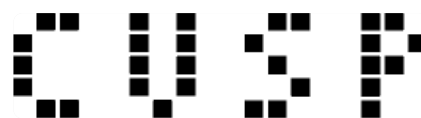
NVIDIA cuFFT



IMSL Library



ArrayFire Matrix
Computations



Sparse Linear
Algebra



C++ STL Features
for CUDA



3 Ways to Accelerate Applications

Applications

Libraries

“Drop-in”
Acceleration

Compiler
Directives

Easily Accelerate
Applications

Programming
Languages

Maximum
Flexibility