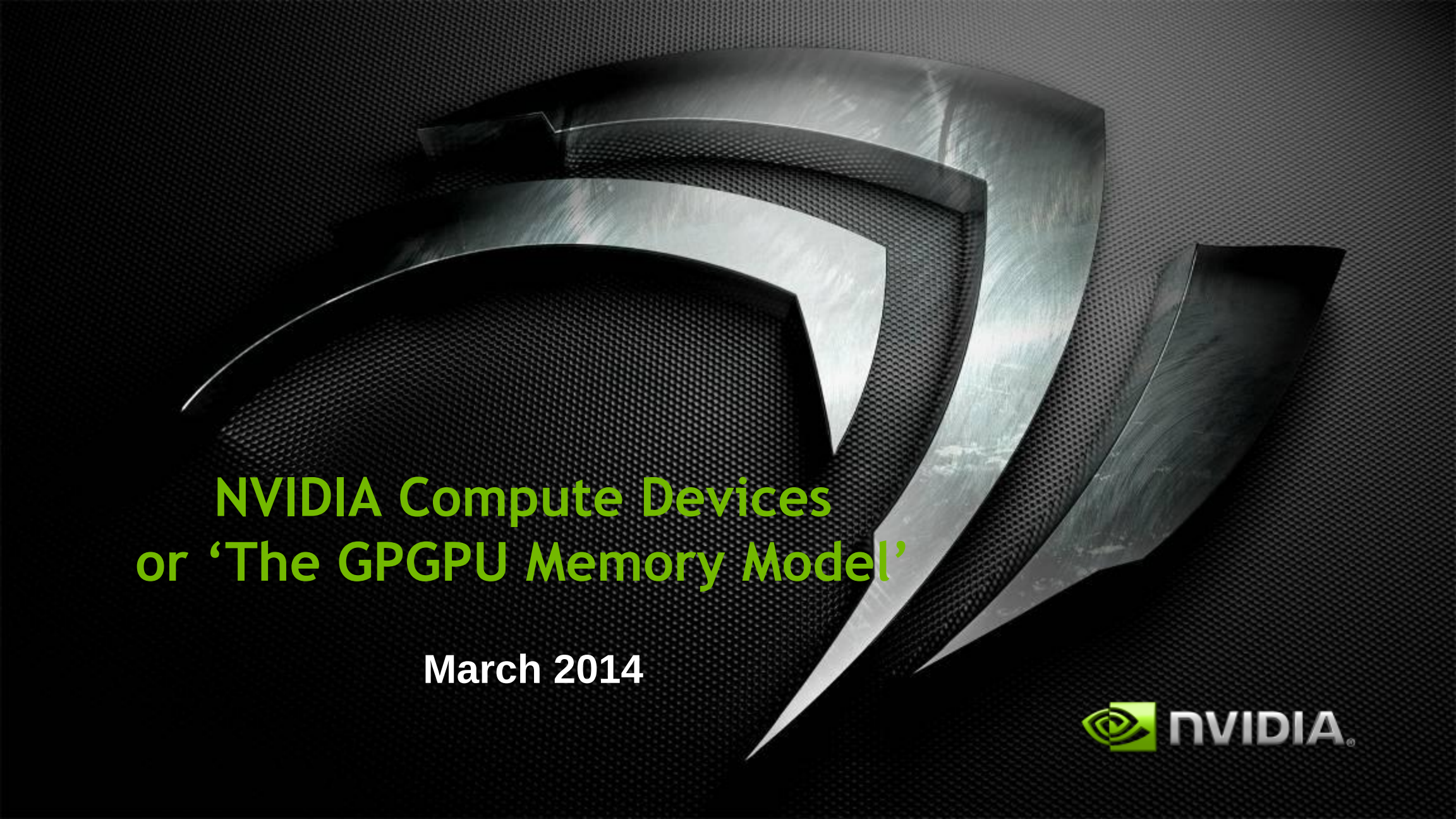

NVIDIA Compute Devices
or ‘The GPGPU Memory Model’
March 2014
NVIDIA

Goal

Goal: Educating the OpenSHMEM Community

- **CUDA: Compute Unified Device Architecture**
  - **CUDA C: C language extensions for parallelism**
- **CUDA memory and synchronization model**
- **CUDA process and thread model**
- **Future directions**

This talk the start of a discussion

- **Not a recommendation for immediate change.**

NVIDIA's Goal

Standard for one-sided communication in a parallel context

- **Move beyond host-managed communication**
- **Express communication with parallelism**
- **Improve Amdahl fraction**

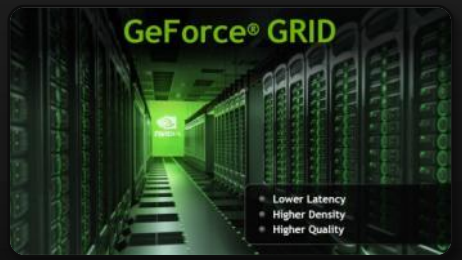
Drop-in library usable on both the CPU and GPU

Acronyms and Synonyms

- GPU == GPGPU == TOC
  - GPU is the historical name
  - GPGPU General Purpose GPU, modern GPU with compute capability
  - TOC Throughput Optimized Core
    - Power efficient parallel computation
- LOC Latency Optimized Core
  - CPU-like single threaded core
  - Trade higher power use for fewer, shorter clock cycles

NVIDIA

GPUs: GeForce, Quadro, TESLA



ARM SoCs: Tegra

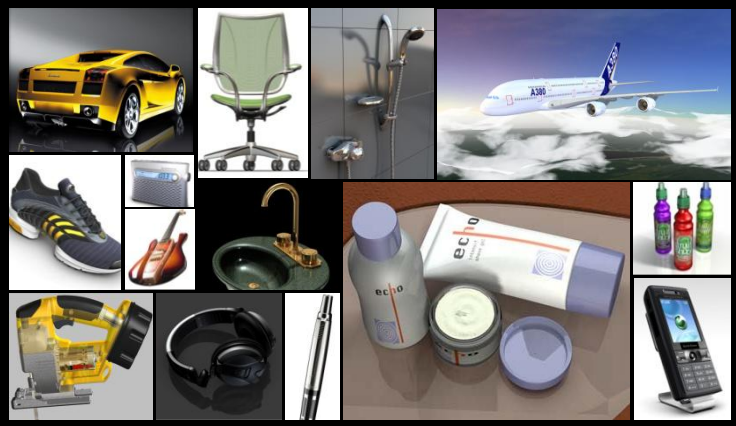
Windows 8 ARM

The Day Job That Makes It All Possible…

- Leverage volume graphics market to serve HPC
  - HPC needs outstrip HPC market's ability to fund the development
  - Computational graphics and compute are *highly* aligned

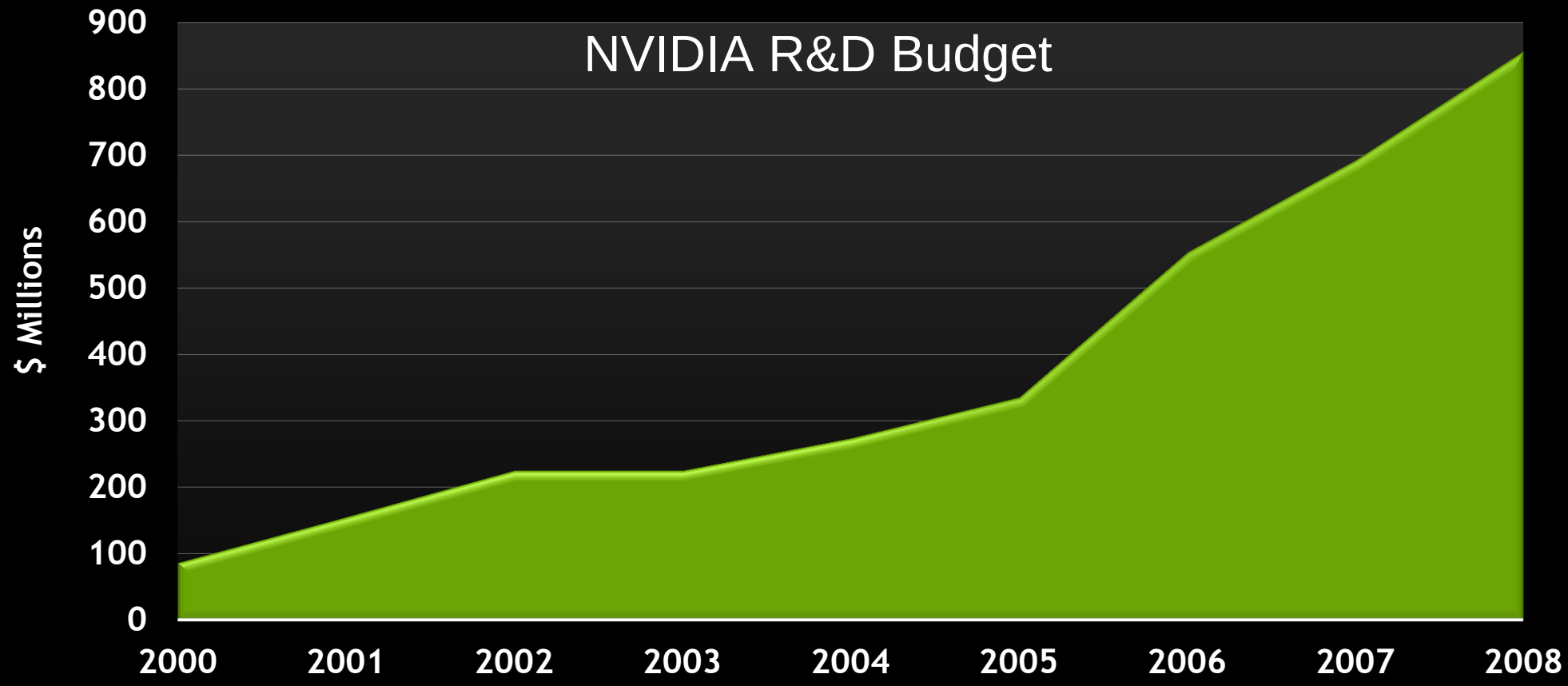
Tegra

GeForce



Quadro

GPU Innovation Accelerating



C for CUDA : C with a few keywords

```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

Standard C Code

```
__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)  y[i] = a*x[i] + y[i];
}
// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Parallel C Code

CUDA: C on the GPU

- A simple, explicit programming language solution
- Extend only where necessary

```
__global__ void KernelFunc(...);
__shared__ int SharedVar;
KernelFunc<<< 500, 128 >>>(...);
```

- Explicit GPU memory allocation
  - `cudaMalloc()`, `cudaFree()`
- Memory copy from host to device, etc.
  - `cudaMemcpy()`, `cudaMemcpy2D()`, ...

CUDA C with APK

Standard C Code

```
void saxpy_serial(int n,
                  float a,
                  float *x,
                  float *y)
{


for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}
// Perform SAXPY on 1M elements
saxpy_serial(4096*256, 2.0, x, y);
```

Parallel C Code

```
__global__
void saxpy_parallel(int n,
                    float a,
                    float *x,
                    float *y)
{
  int start = threadIndex();
  int stride = threadCount();
  for (int i = start; i < n; i+= stride )
      y[i] = a*x[i] + y[i];
}
// Perform SAXPY on 1M elements
saxpy_parallel(4096*256,2.0,x,y);
```

http://developer.nvidia.com/cuda-toolkit

Process and Thread Model

- **Many threads required for performance**
  - **Thousands to ~100K typical**
    - **Hardware scheduler keeps up to 32K active (Kepler K20)**
    - **2K threads to cover memory latency**
  - **Program in threads, hardware executes in SIMT**
    - **Supports complete divergence**
    - **Hardware/architecture vectorizes dynamically**
- **Implication: PE-per-thread (or even thread group) infeasible**
  - **… with PE-per-node too coarse-grained**

Memory Space Model

- Threads have multiple memory spaces
  - Private local - each thread
  - Shared - among threads in active thread block
    - Within the lifetime of the thread block
  - Read-only (visible to all threads, persistent across kernels)
    - Constant
    - Texture - additional addressing modes, data filtering
  - Global (all threads, persistent across kernels)
- Implication: Rich memory spaces vs OpenSHMEM memory spaces

Programming 100K nodes

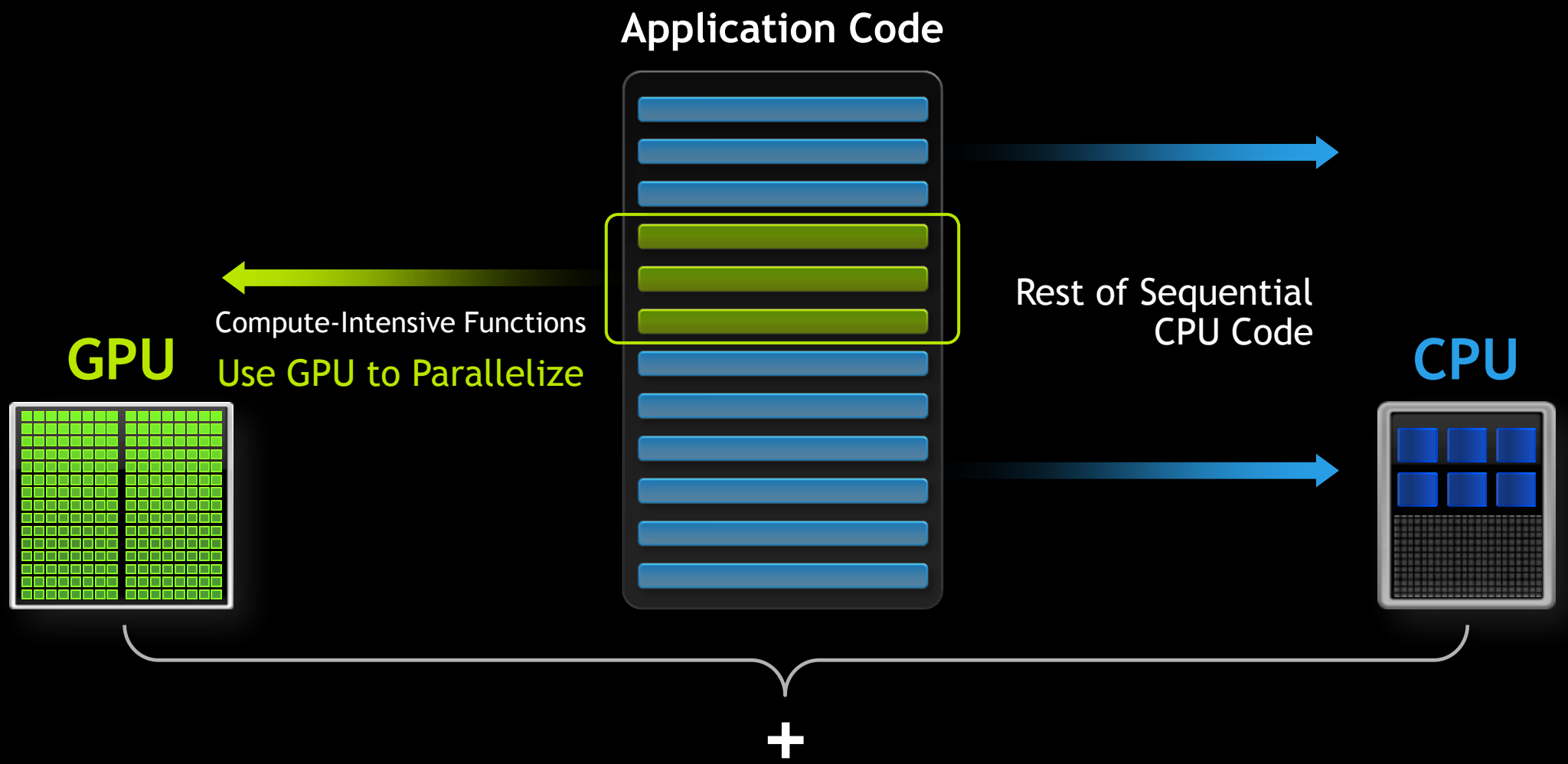
- Committed to legacy requirement: MPI + on-node parallelism
  - Hybrid Multicore
  - Directive or language solution
  - Explicit and implicit memory hierarchy management
- Exascale system wide programming model
  - Exascale solution must support *enabling technologies* for PGAS and GPUDirect
  - ATOMIC
  - Cluster wide addressing, paging
  - Active memory hierarchy

Kernel Acceleration (Offload) Model

- Current programming model
- CPU constructs work kernel
- Submits to a queue
- GPU interprets and executes the kernel
  - No concurrency guarantee
  - Spinwait within kernel == deadlock
  - Thread block has internal consistency
  - External or inter-kernel consistent only at exit sync

Work submission (launch) and exit (join) only points of consistency

GPUs Accelerate Computational Core

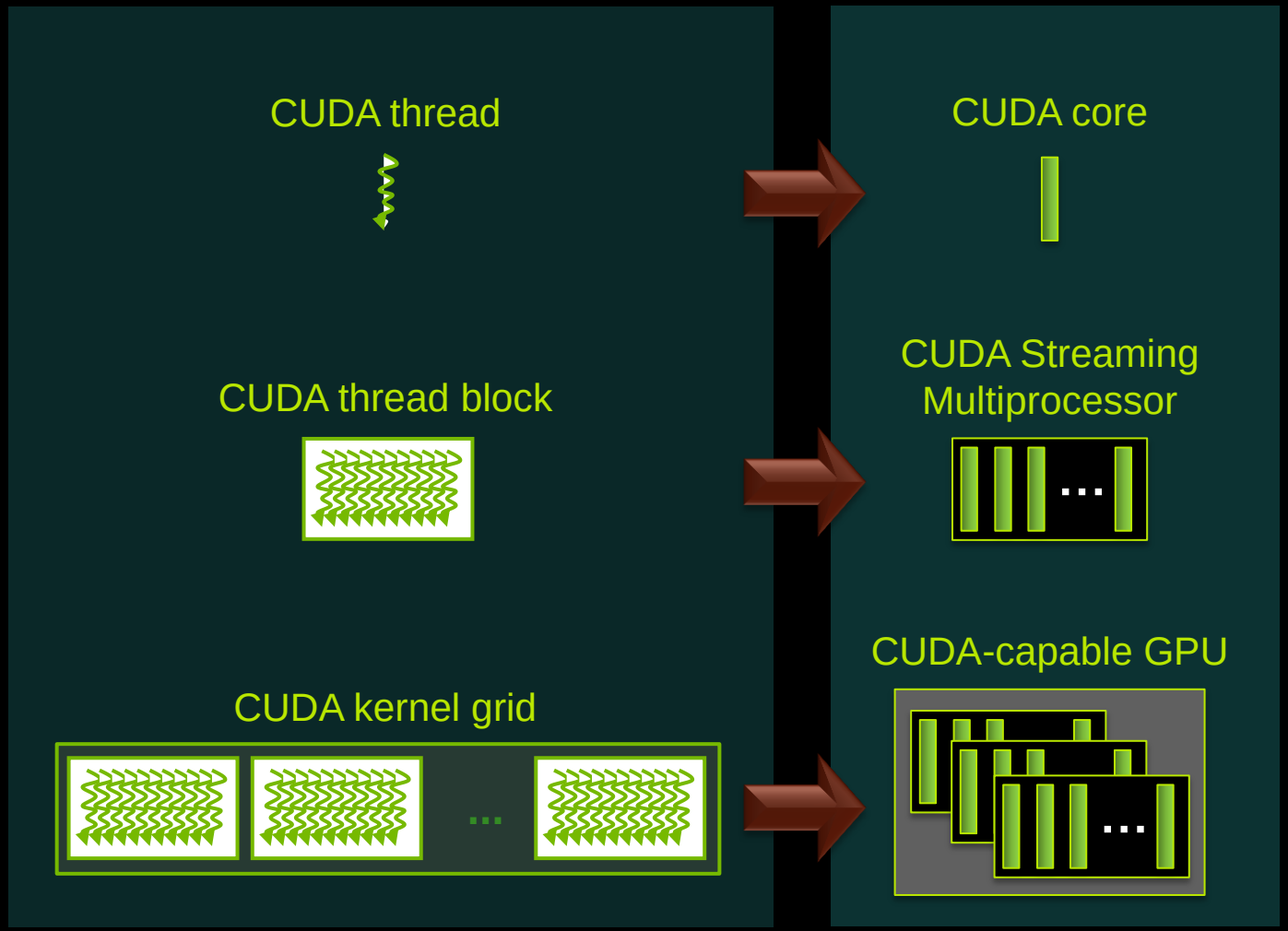


Kernel Acceleration (Offload) Model

Real life is a bit more sophisticated

- Hardware thread scheduling and dispatch
- Execution behavior introspection
  - Occupancy, effectiveness
  - Power and thermal
  - All influence behavior and performance
- GPU kernels can create new kernels
  - Architectural improvement that moves longer chunks to GPU

Kernel Execution

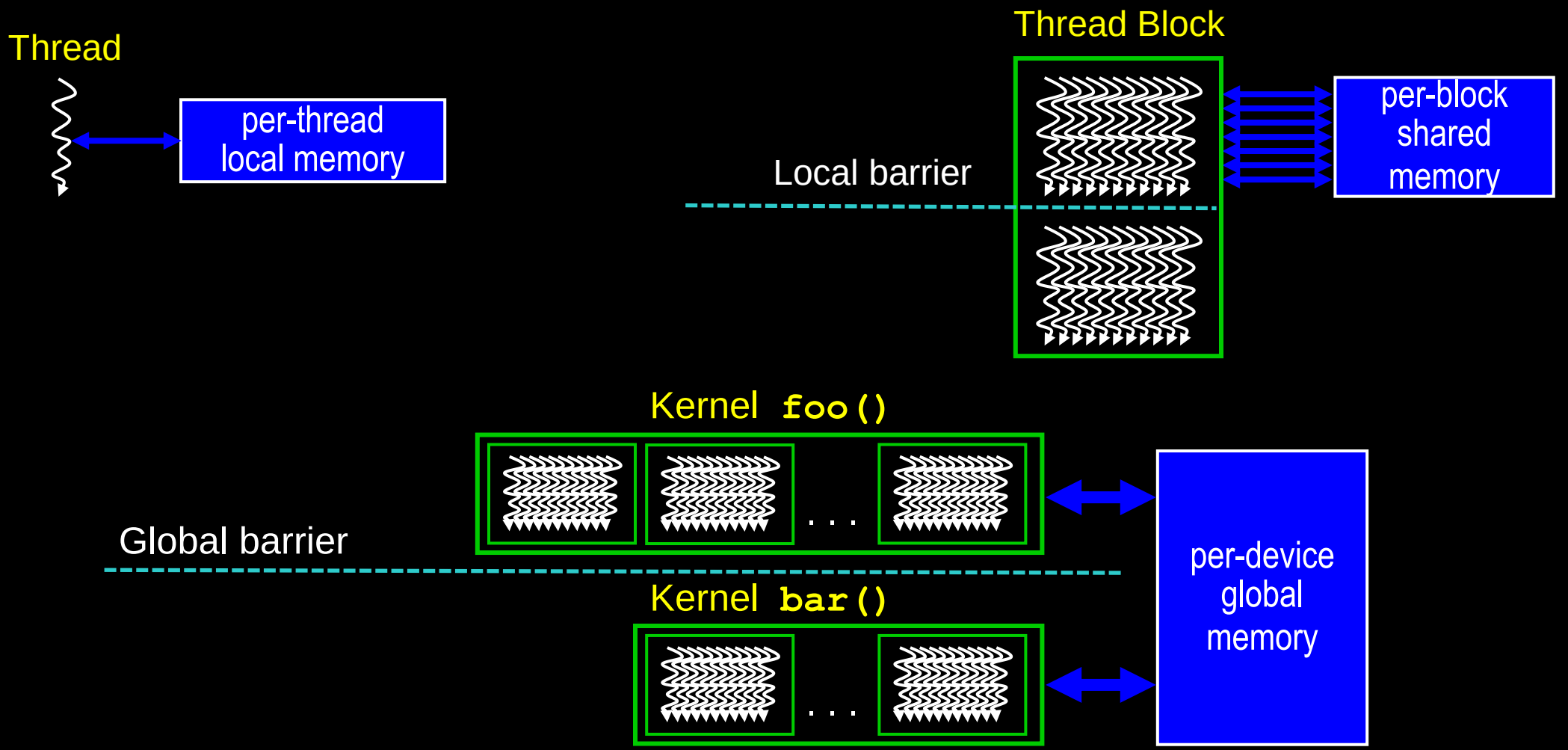


- Each thread is executed by a core

- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks' memory requirements and the SM's memory resources

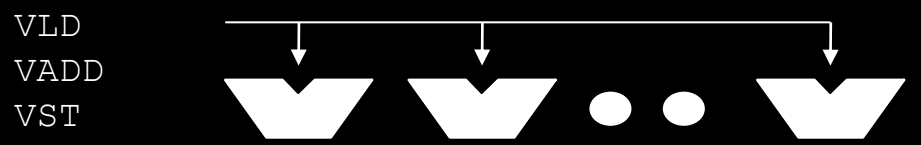
- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time

GPU Parallelism



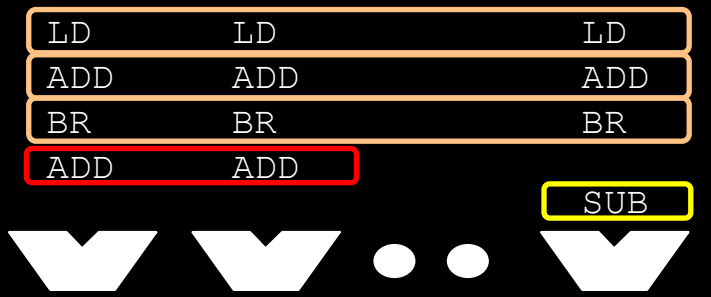
SIMD versus MIMD versus SIMT?

- **SIMD: Single Instruction Multiple Data**



- **MIMD: Multiple Instruction Multiple Data**

- **SIMT: Single Instruction Multiple Thread**



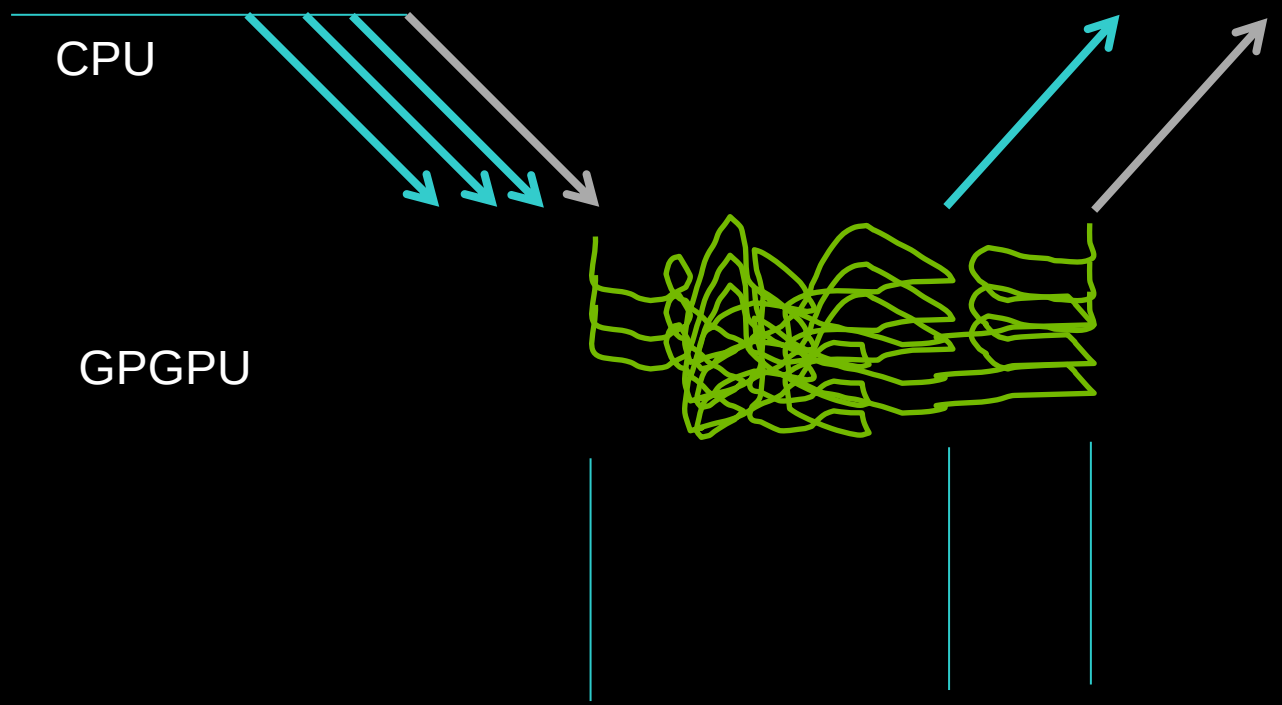
SIMT = MIMD Programming Model w/
SIMD Implementation Efficiencies

Divergence in Parallel Computing

- Removing divergence pain from parallel programming
- SIMD Pain
  - User or compiler required to SIMD-ify
  - User suffers when computation goes divergent
- GPUs: Decouple execution width from programming model
  - Threads can diverge freely
  - Inefficiency only when granularity exceeds native machine width
  - Hardware managed
  - Managing divergence becomes performance optimization
  - Scalable

Illustration of Launch/Join Consistency

- No (externally promised) consistency during offload execution



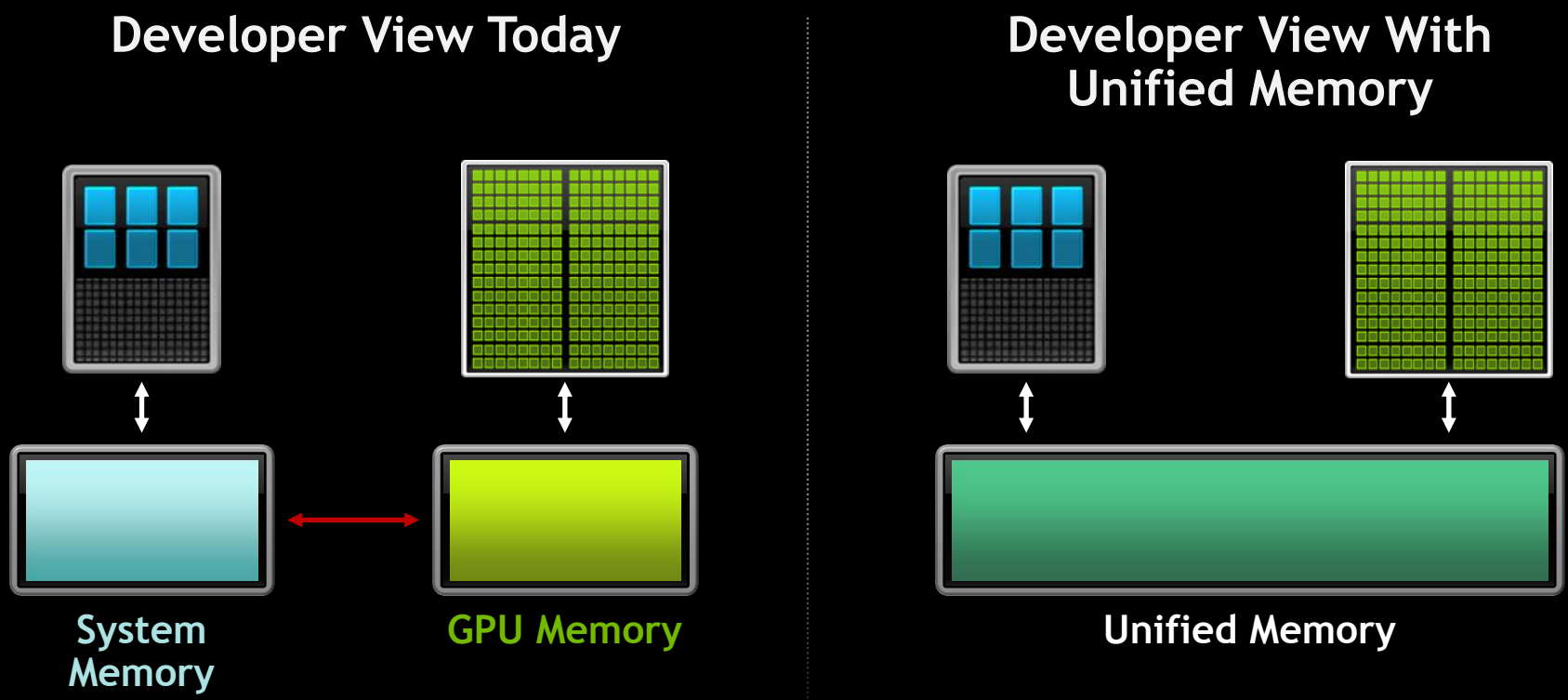
Undefined ordering implications

- Execution and completion ordering undefined
  - Allows hardware thread scheduling
  - Key element of performance
- Implication: GPU communication operations unordered

OpenSHMEM Integration Challenges

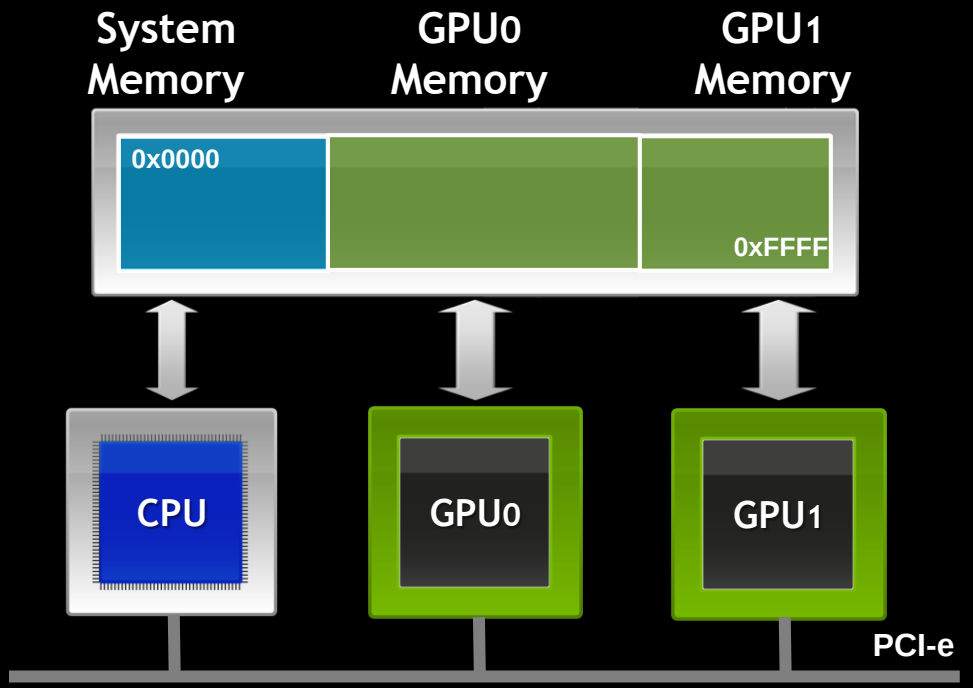
- Simple model is simple -- CPU driven
  - Potentially eave data on GPU-controlled memory
  - Use only after GPU sync join
- Performance and scalability limited
- Differentiated memory
  - CPU (DDR) vs GPU (GDR) local memory
  - Internal operational memory

Unified Memory
Dramatically Lower Developer Effort



Multi-GPU: Unified Virtual Addressing

Single Partitioned Address Space



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

Super Simplified Memory Management Code

CPU Code

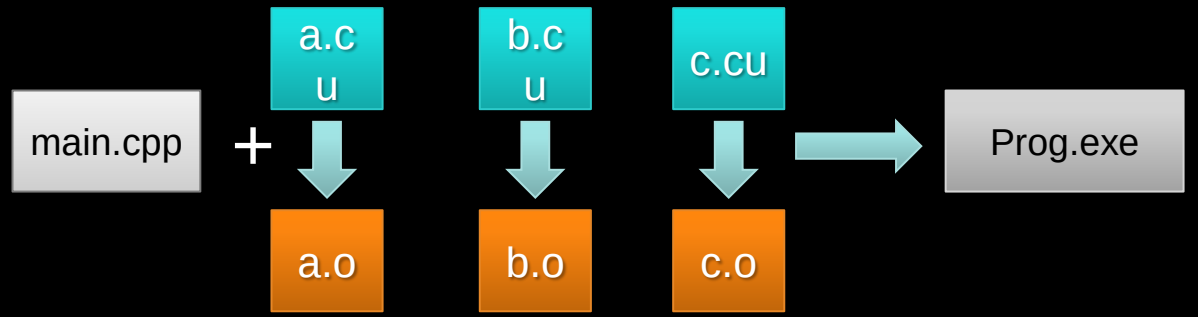
```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  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);
}
```

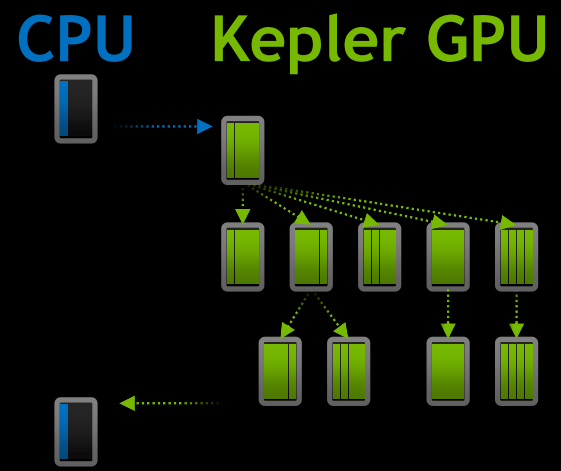
Impact of CUDA 5 and Kepler

- **Separate Compilation & Linking**



Compiler does not have to ‘see’ all functions
Link and externally call *device* code
Create reusable libraries

- **Dynamic Parallelism**



GPU can generate work for itself

Conclusion

- Sketch of CUDA memory and threading
- Concern about trading sideways with a communication model

MPI worked well with clusters

- Single core, strongly ordered TSO memory model
- Single path to NIC
- Rank == node, or rank == core
- Easy to send, hard to receive

Need a improved community API that matches upcoming machines

- Thousands of threads
- Hierarchy of locality and memory spaces