New Programming Model Features

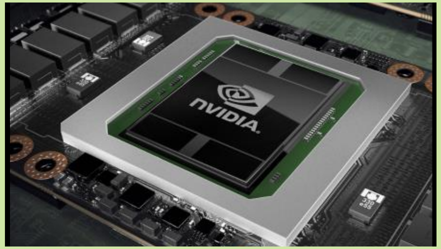
Cyril Zeller, Director of Developer Technology

NVIDIA

INTRODUCING TESLA P100

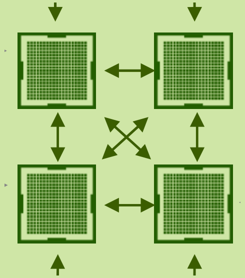
New GPU Architecture to Enable the World's Fastest Compute Node

Pascal Architecture



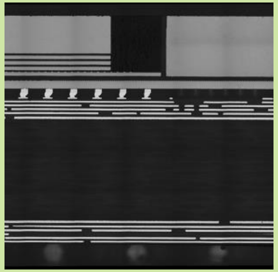
Highest Compute Performance

NVLink



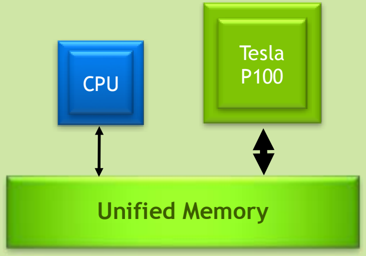
GPU Interconnect for Maximum Scalability

HBM2 Stacked Memory

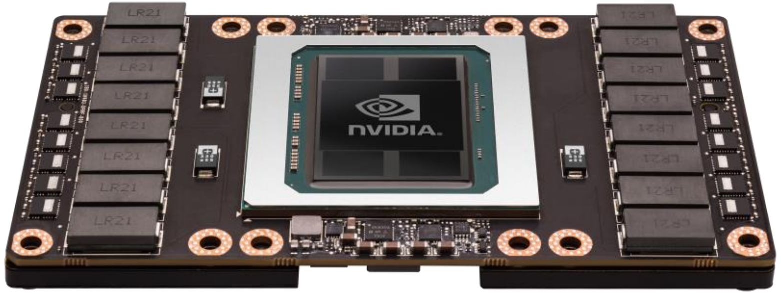


Unifying Compute & Memory in Single Package

Page Migration Engine



Simple Parallel Programming with 512 TB of Virtual Memory

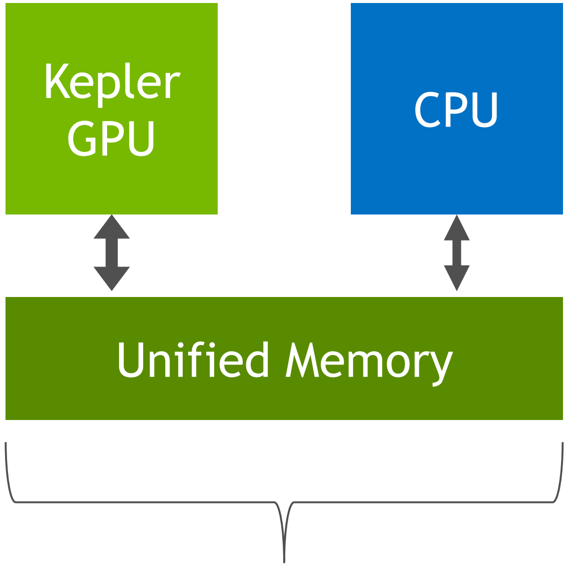


Unified Memory

Unified Memory

Dramatically Lower Developer Effort

CUDA 6+



Allocate Up To
GPU Memory Size

Simpler Programming & Memory Model

- Single allocation, single pointer, accessible anywhere
- Eliminate need for *explicit copy*
- Greatly simplifies code porting

Performance Through Data Locality

- Migrate data to accessing processor
- Guarantee global coherence
- Still allows explicit hand tuning

Simplified Memory Management Code

CPU Code

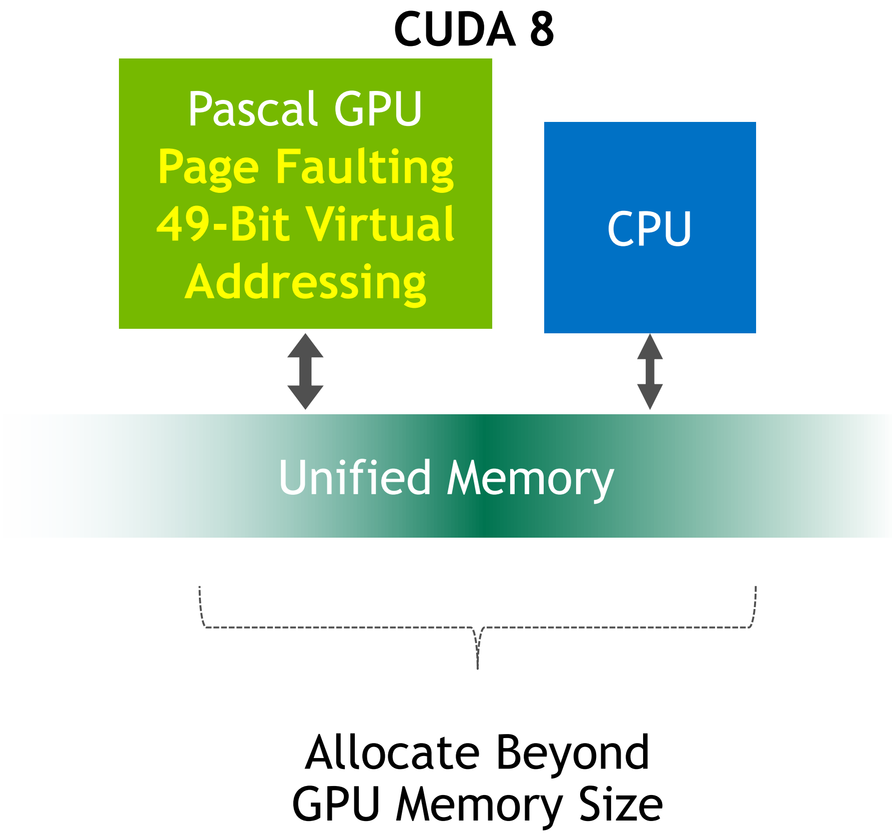
```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  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);
}
```

CUDA 8: Unified Memory on Pascal

Large datasets, simple programming, High Performance



Enable Large Data Models
- Oversubscribe GPU memory
- Allocate up to system memory size

Simpler Data Access
- CPU/GPU Data coherence
- Unified memory atomic operations

Tune Unified Memory Performance
- Usage hints via cudaMemAdvise API
- Explicit prefetching API

Unified Memory Example

On-Demand Paging

```
__global__
void setValue(int *ptr, int index, int val)
{
  ptr[index] = val;
}


void foo(int size) {
  char *data;
  cudaMallocManaged(&data, size);

  memset(data, 0, size);

  setValue<<<...>>>(data, size/2, 5);
  cudaDeviceSynchronize();

  useData(data);

  cudaFree(data);
}
```

Unified Memory allocation

Access all values on CPU

Access one value on GPU

How Unified Memory Works in CUDA 6

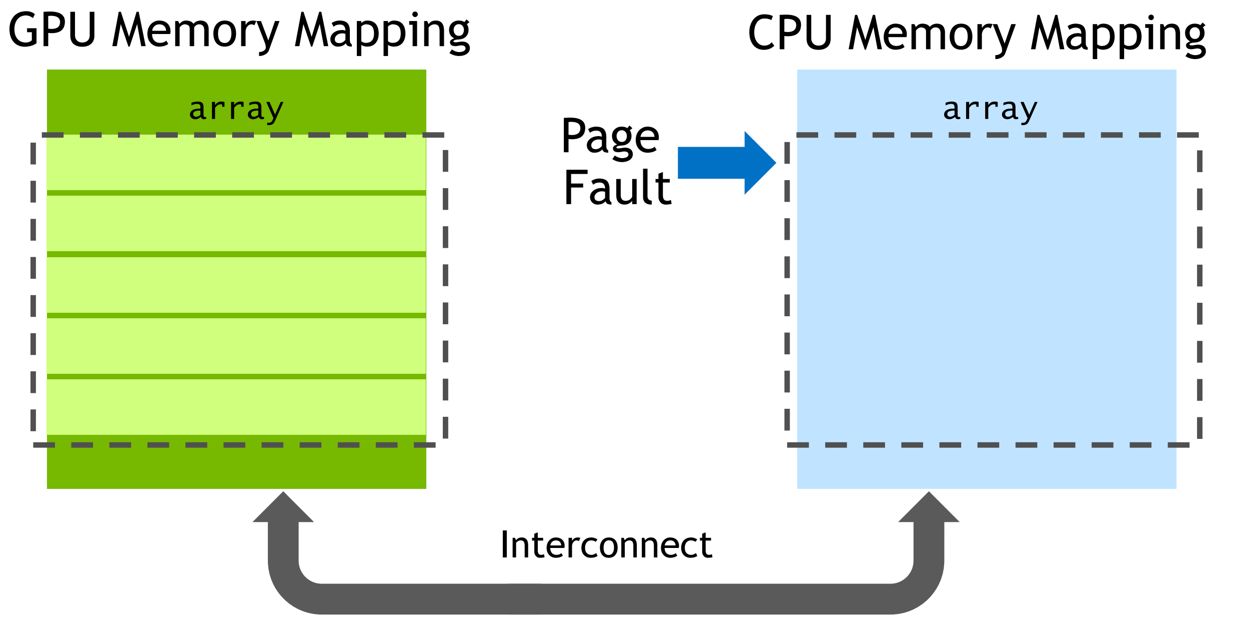
Servicing CPU page faults

GPU Code

```
__global__
void setValue(char *ptr, int index, char val)
{
  ptr[index] = val;
}
```

CPU Code

```
cudaMallocManaged(&array, size);

memset(array, size);

setValue<<<...>>>(array, size/2, 5);
```



How Unified Memory Works on Pascal

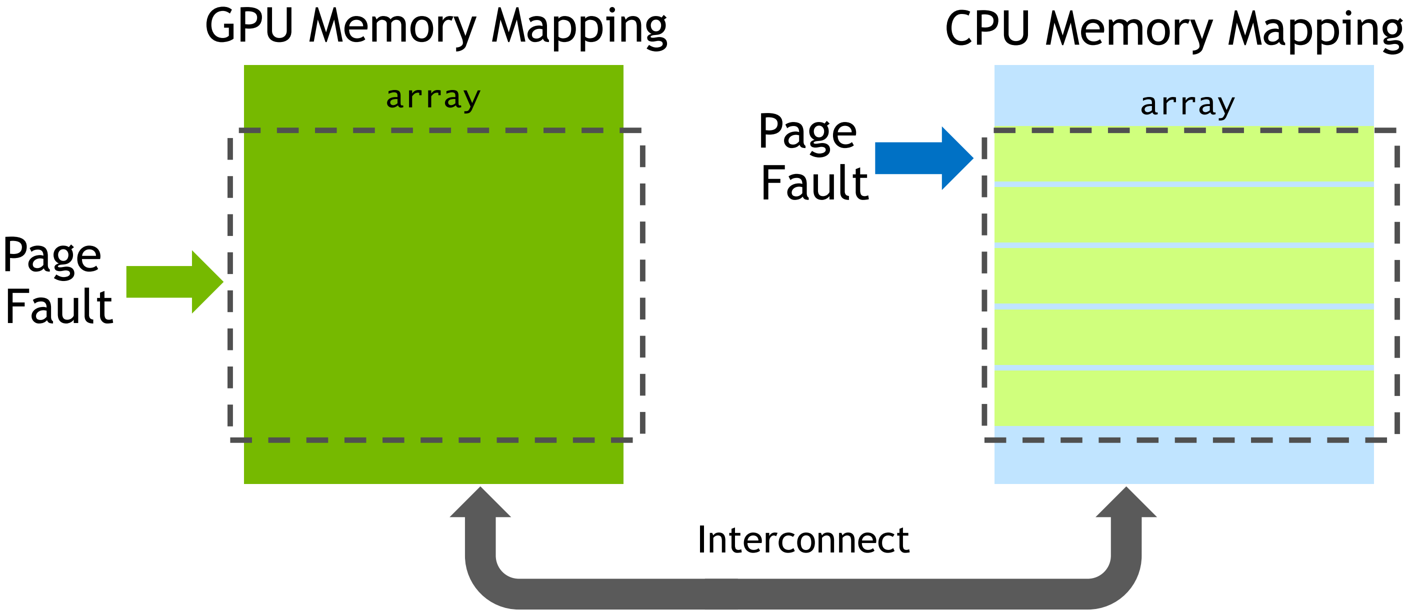
Servicing CPU *and* GPU Page Faults

GPU Code

```
__global__
Void setValue(char *ptr, int index, char val)
{
  ptr[index] = val;
}
```

CPU Code

```
cudaMallocManaged(&array, size);

memset(array, size);

setValue<<<...>>>(array, size/2, 5);
```



Unified Memory on Pascal

GPU memory oversubscription

```
void foo() {
  // Assume GPU has 16 GB memory
  // Allocate 32 GB
  char *data;
  size_t size = 32*1024*1024*1024;
  cudaMallocManaged(&data, size);
}
```

32 GB allocation

Pascal supports allocations where only a subset of pages reside on GPU. Pages can be migrated to the GPU when “hot”.

Fails on Kepler/Maxwell

Unified Memory on Pascal

Concurrent CPU/GPU access to managed memory

```
__global__
void setValue(int *ptr, int index, int val)
{
  ptr[index] = val;
}

void foo(int size) {
  char *data;
  cudaMallocManaged(&data, size);

  setValue<<<...>>>(data, size/2, 5);
  // no synchronize here
  data[0] = ‘c’;

  cudaFree(data);
}
```

OK on Pascal: just a page fault

Concurrent CPU access to ‘data’ on previous GPUs caused a fatal segmentation fault

Unified Memory on Pascal

System-Wide Atomics

```
__global__ void mykernel(int *addr) {
  atomicAdd(addr, 10);
}

void foo() {
  int *addr;
  cudaMallocManaged(addr, 4);
  *addr = 0;

  mykernel<<<...>>>(addr);
  __sync_fetch_and_add(addr, 10);
}
```

Pascal enables system-wide atomics

- Direct support of atomics over NVLink
- Software-assisted over PCIe

System-wide atomics not available on Kepler / Maxwell

Performance Tuning on Pascal

Explicit Memory Hints and Prefetching

Explicit prefetching with `cudaMemPrefetchAsync(ptr, length, destDevice, stream)`

Unified Memory alternative to cudaMemcpyAsync

Asynchronous operation that follows CUDA stream semantics

Advise runtime on known memory access behaviors with `cudaMemAdvise()`

`cudaMemAdviseSetReadMostly`: specify read duplication
`cudaMemAdviseSetPreferredLocation`: suggest best location
`cudaMemAdviseSetAccessedBy`: initialize a mapping

To Learn More:
S6216 “The Future of Unified Memory” by Nikolay Sakharnykh
Available at http://on-demand-gtc.gputechconf.com/

Heterogeneous C++ Lambda (CUDA 8)

Heterogeneous C++ Lambda

Combined CPU/GPU lambda functions

```
__global__ template <typename F, typename T>
void apply(F function, T *ptr) {
  *ptr = function(ptr);
}

int main(void) {
  float *x;
  cudaMallocManaged(&x, 2);

  auto square =
    [=] __host__ __device__ (float x) { return x*x; };

  apply<<<1, 1>>>(square, &x[0]);

  ptr[1] = square(&x[1]);

  cudaFree(x);
}
```

Call lambda from device code

__host__ __device__ lambda

Pass lambda to CUDA kernel

… or call it from host code

Experimental feature in CUDA 8.
`nvcc --expt-extended-lambda`

Heterogeneous C++ Lambda

Usage with Thrust

```
void saxpy(float *x, float *y, float a, int N) {
    using namespace thrust;
    auto r = counting_iterator(0);

    auto lambda = [=] __host__ __device__ (int i) {
      y[i] = a * x[i] + y[i];
    };

    if(N > gpuThreshold)
      for_each(device, r, r+N, lambda);
    else
      for_each(host, r, r+N, lambda);
}
```

__host__ __device__ lambda

Use lambda in thrust::for_each on host or device

Experimental feature in CUDA 8.
`nvcc --expt-extended-lambda`

Beyond CUDA 8: Cooperative Groups

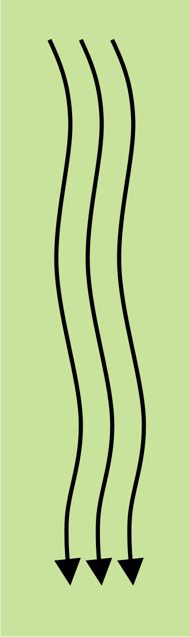
Cooperative groups

A Programming Model for Coordinating Groups of Threads

Support clean composition across software boundaries (e.g. libraries)

Optimize for hardware fast-path using safe, flexible synchronization

A programming model that can scale from Kepler to future platforms



Cooperative Groups Summary

Flexible, Explicit Synchronization

Thread groups are explicit objects in the program

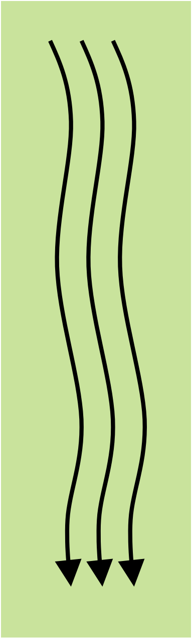
```
thread_group group = this_thread_block();
```

Collectives, such as barriers, operate on thread groups

```
sync(group);
```

New groups are constructed by partitioning existing groups

```
thread_group tiled_partition(thread_group base, int size);
```



Motivating Example

Optimizing for Warp Size

```
__device__
int warp_reduce(int val) {
  extern __shared__ int smem[];
  const int tid = threadIdx.x;

  #pragma unroll
  for (int i = warpSize/2; i > 0; i /= 2) {
    smem[tid] = val;      __syncthreads();
    val += smem[tid ^ i]; __syncthreads();
  }
  return val;
}
```

__syncthreads() is too expensive when sharing is only within warps

Motivating Example

Implicit Warp-Synchronous Programming is Tempting…

```
__device__
int warp_reduce(int val) {
  extern __shared__ int smem[];
  const int tid = threadIdx.x;

  #pragma unroll
  for (int i = warpSize/2; i > 0; i /= 2) {
    smem[tid] = val;
    val += smem[tid ^ i];
  }
  return val;
}
```

Barriers separating steps removed.
UNSAFE!

Motivating Example

Safe, Explicit Programming for Performance

Approximately equal performance to unsafe warp programming

```
__device__
int warp_reduce(int val) {
  extern __shared__ int smem[];
  const int tid = threadIdx.x;

  #pragma unroll
  for (int i = warpSize/2; i > 0; i /= 2) {
    smem[tid] = val;        sync(this_warp());
    val += smem[tid ^ i];   sync(this_warp());
  }
  return val;
}
```

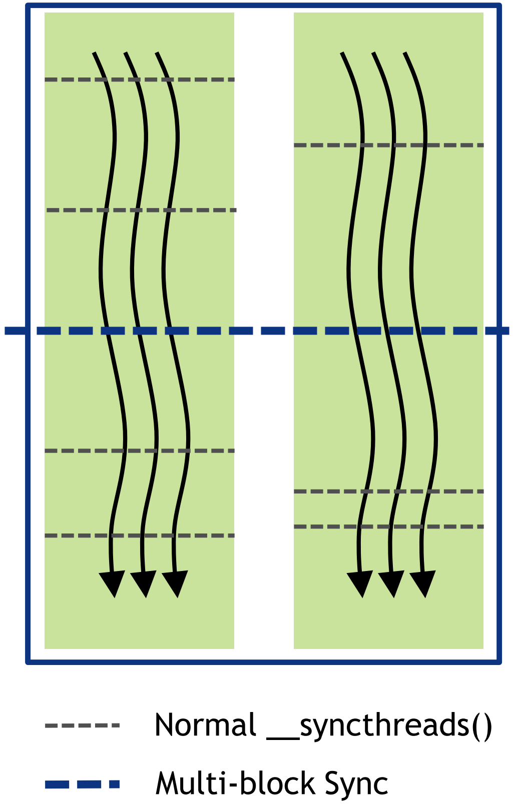
Safe *and* Fast!

Pascal: Multi-Block Cooperative Groups

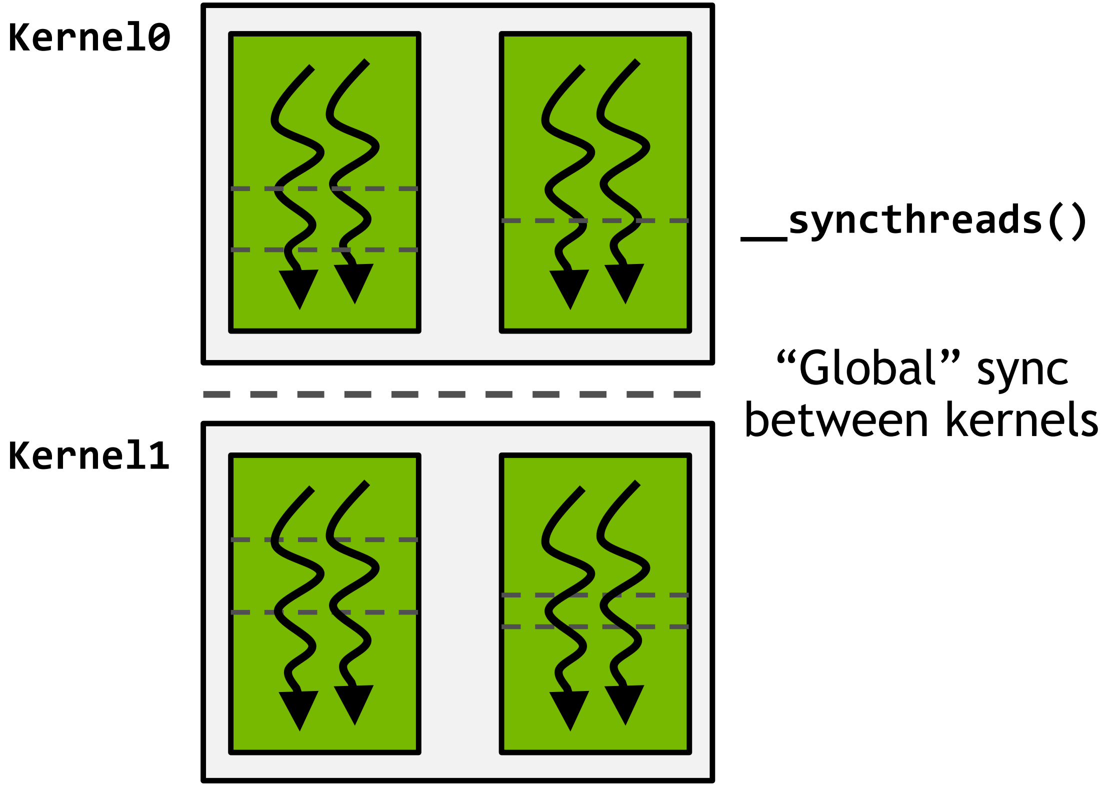
Provide a new launch mechanism for multi-block groups

Cooperative Groups collective operations like `sync(group)` work across all threads in the group

Save bandwidth and latency compared to multi-kernel approach required on Kepler GPUs



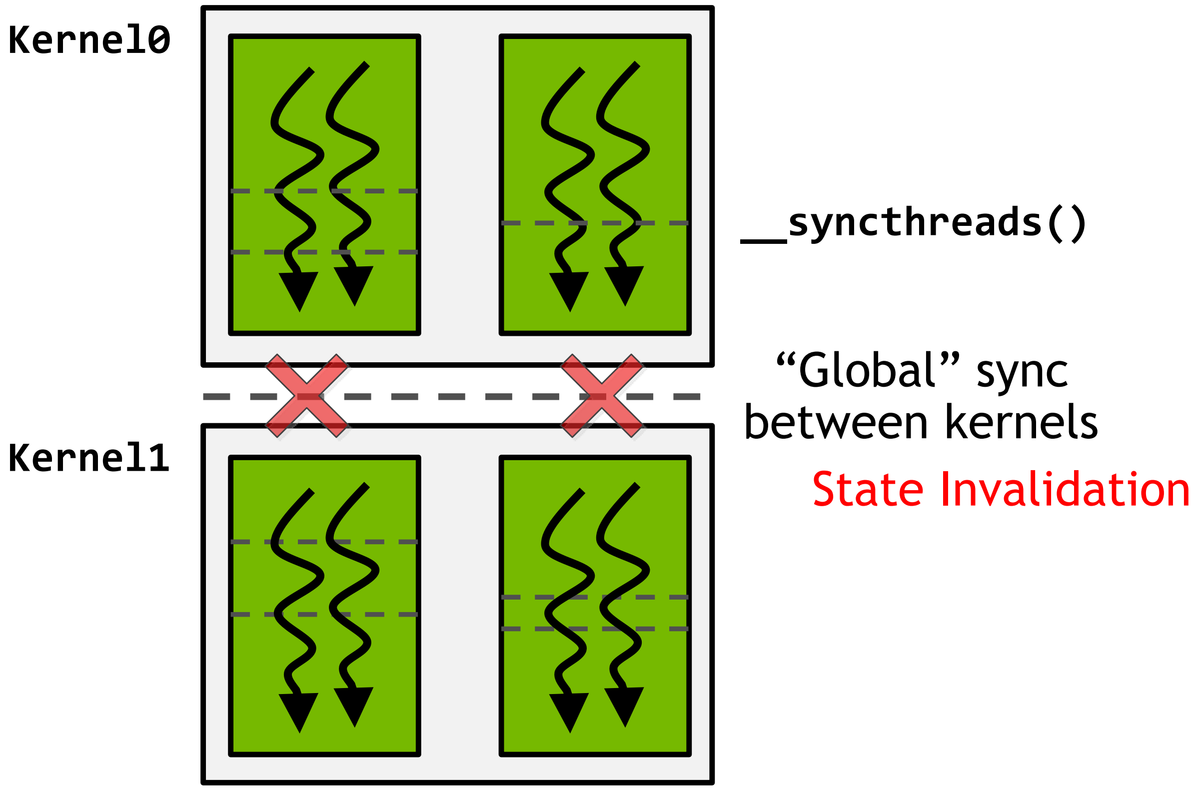
CUDA Synchronization Model



CUDA Blocks define the set of threads that communicate and synchronize

All other synchronization occurs at CUDA Grid boundaries

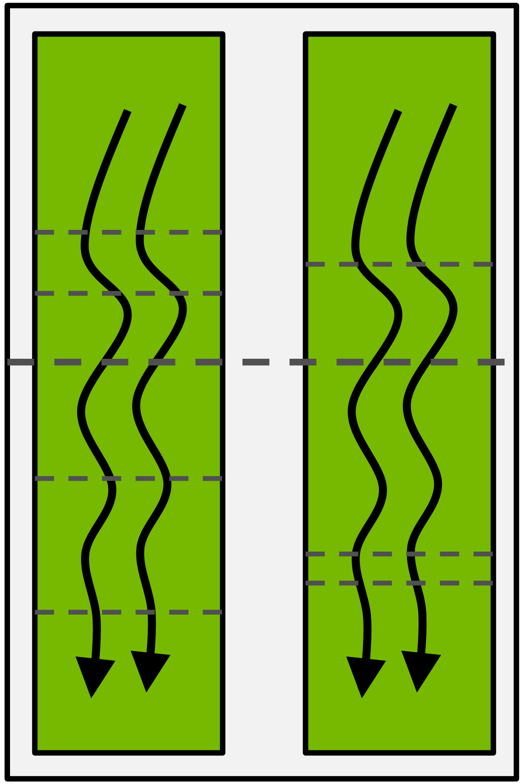
CUDA Synchronization Model



Lose register and shared memory state between kernels

Global Synchronization

Multi-Block Cooperative Groups



No state invalidation

Maintain register and shared memory state between phases of execution