CENG443
Heterogeneous Parallel Programming

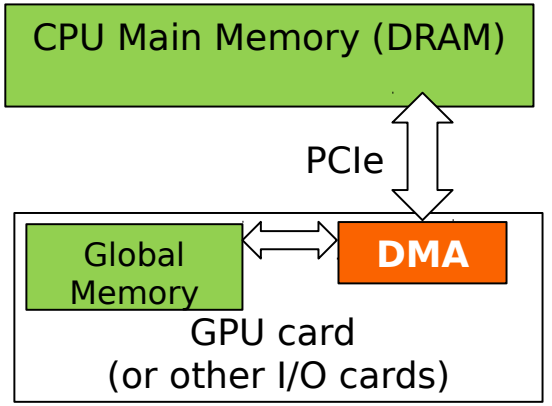
More Parallelism in CUDA

CPU-GPU Data Transfer

DMA (Direct Memory Access) hardware is used by cudaMemcpy() for better efficiency

Modern computers use virtual memory management

DMA uses physical addresses



- When cudaMemcpy() copies an array, it is implemented as one or more DMA transfers
- Address is translated and page presence checked for the entire source and destination regions at the beginning of each DMA transfer
- No address translation for the rest of the same DMA transfer so that high efficiency can be achieved

Pinned Memory

OS could accidentally page-out the data that is being read or written by a DMA and page-in another virtual page into the same physical location

Pinned memory are virtual memory pages that are specially marked so that they cannot be paged out (Page-Locked Memory)

Allocated with a special system API function call

CPU memory that serve as the source or destination of a DMA transfer must be allocated as pinned memory

Pinned Memory

If a source or destination of a cudaMemcpy() in the host memory is not allocated in pinned memory, it needs to be first copied to a pinned memory – extra overhead

cudaMemcpy() is faster if the host memory source or destination is allocated in pinned memory since no extra copy is needed

Zero-copy memory

Pinned Memory Allocation

cudaMallocHost()

Address of pointer to the allocated memory

Size of the allocated memory in bytes

cudaFreeHost()

Pointer to the memory to be freed

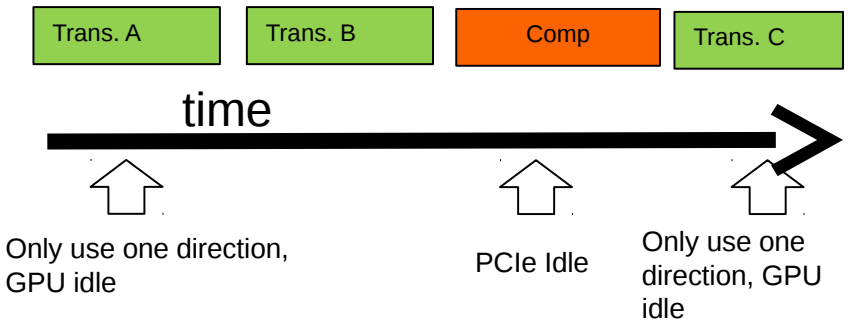
Pinned memory is a limited resource

Pinned Memory Example

```
int main()
{
   float *h_A, *h_B, *h_C;
…
   cudaMallocHost((void **) &h_A, N* sizeof(float));
   cudaMallocHost((void **) &h_B, N* sizeof(float));
   cudaMallocHost((void **) &h_C, N* sizeof(float));
…
   // cudaMemcpy() runs 2X faster
}
```

Serialized Data Transfer and Computation

So far, the way we use cudaMemcpy serializes data transfer and GPU computation



Device Overlap

Some CUDA devices support device overlap

Simultaneously execute a kernel while copying data between device and host memory

```
int dev_count;
cudaDeviceProp prop;

cudaGetDeviceCount( &dev_count);
for (int i = 0; i < dev_count; i++) {
  cudaGetDeviceProperties(&prop, i);
  if (prop.deviceOverlap) …
```

CUDA Streams

CUDA supports parallel execution of kernels and cudaMemcpy() with “Streams”

Each stream is a queue of operations (kernel launches and cudaMemcpy() calls)

Operations (tasks) in different streams can go in parallel “Task parallelism”

CUDA Streams

Create/destroy

```
cudaStream_t stream;
cudaStreamCreate( &stream );
cudaStreamDestroy( stream );
```

Launch

```
my_kernel<<<grid,block,0,stream>>>(...);
cudaMemcpyAsync( …, stream );
```

Synchronize

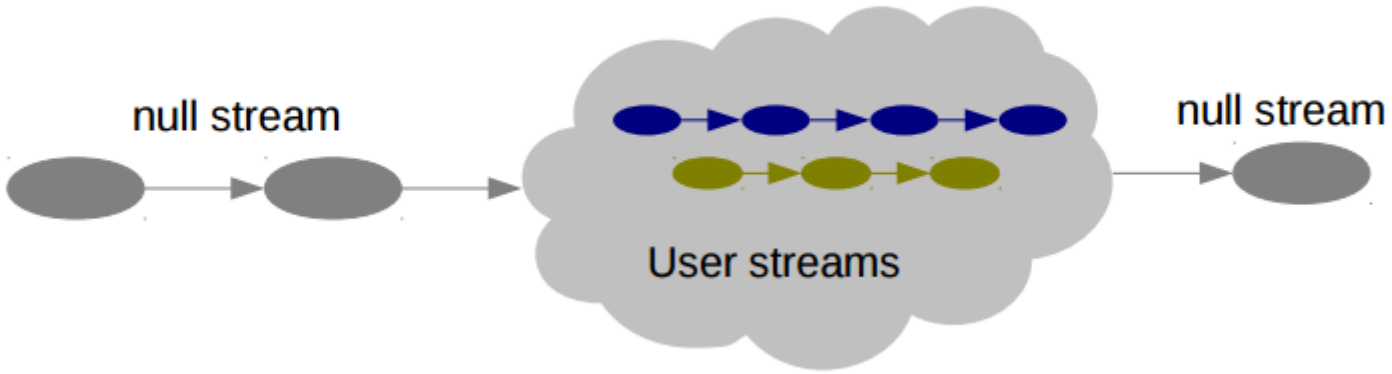
```
cudaStreamSynchronize( stream );
```

CUDA Streams

Which stream here?

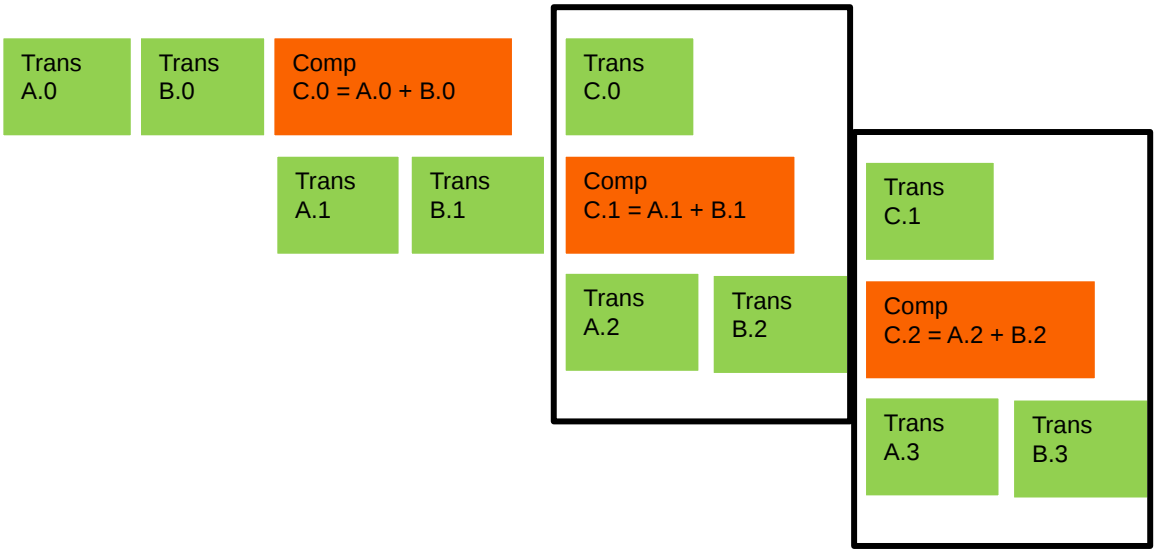
```
my_kernel<<<grid,block>>> (...);
```

The default (null) stream

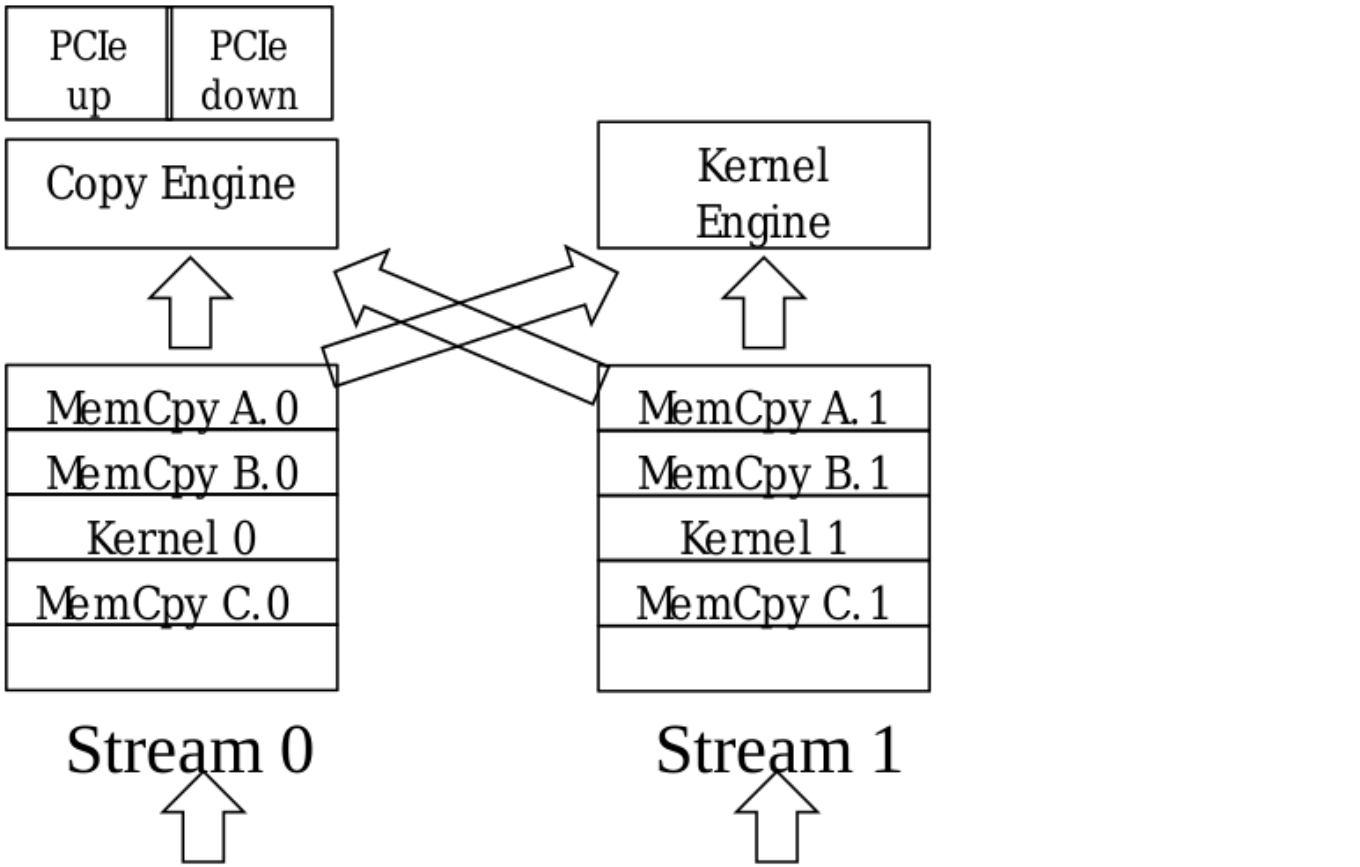


Multiple Streams

Data transfer/kernel execution overlap



Multiple Streams



Allocation for Streams

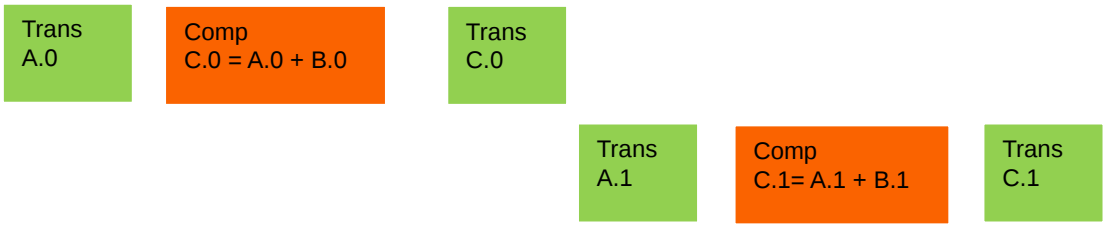
```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size); //in pinned memory
```

Stream Operations

```
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel <<<100, 512, 0, stream[i]>>>
          (outputDevPtr + i * size, inputDevPtr + i * size, size);
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
}

for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

No Overlap

On devices that do not support concurrent data transfers, C.0 (D2H issued to stream[0]) blocks A.1 (H2D issued to stream[1])

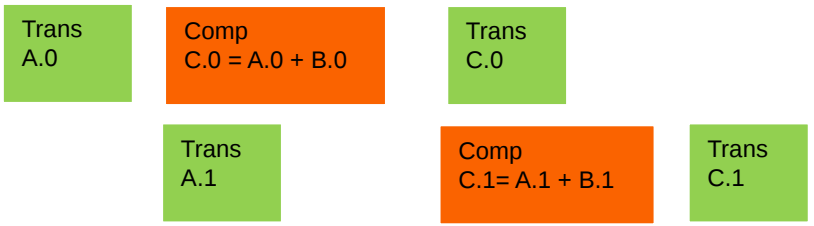


Stream Operations

```
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 2; ++i)
    MyKernel<<<100, 512, 0, stream[i]>>>
          (outputDevPtr + i * size, inputDevPtr + i * size, size);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
```

Overlap

If device supports overlap of data transfer and kernel execution



Simple Example: Synchronous

```
cudaMalloc ( &dev1, size ) ;
double* host1 = (double*) malloc ( &host1, size ) ;
…
cudaMemcpy ( dev1, host1, size, H2D ) ;
kernel2 <<< grid, block, 0 >>> ( …, dev2, … ) ;
kernel3 <<< grid, block, 0 >>> ( …, dev3, … ) ;
cudaMemcpy ( host4, dev4, size, D2H ) ;
...
```

All CUDA operations in the default stream are synchronous, no overlap

Simple Example: Asynchronous, No Streams

```
cudaMalloc ( &dev1, size ) ;
double* host1 = (double*) malloc ( &host1, size ) ;
…
cudaMemcpy ( dev1, host1, size, H2D ) ;
kernel2 <<< grid, block >>> ( …, dev2, … ) ;
some_CPU_method ();
kernel3 <<< grid, block >>> ( …, dev3, … ) ;
cudaMemcpy ( host4, dev4, size, D2H ) ;
...
```

GPU kernels are asynchronous with host by default, potentially overlap

Simple Example: Asynchronous with Streams

```
cudaStream_t stream1, stream2, stream3, stream4 ;
cudaStreamCreate ( &stream1) ;
...
cudaMalloc ( &dev1, size ) ;
cudaMallocHost ( &host1, size ) ; // pinned memory required on host
…
cudaMemcpyAsync ( dev1, host1, size, H2D, stream1 ) ;
kernel2 <<< grid, block, 0, stream2 >>> ( …, dev2, … ) ;
kernel3 <<< grid, block, 0, stream3 >>> ( …, dev3, … ) ;
cudaMemcpyAsync ( host4, dev4, size, D2H, stream4 ) ;
some_CPU_method ();
...
```

Fully asynchronous / concurrent, potentially overlap

Explicit Synchronization

Synchronize everything

cudaDeviceSynchronize ()

Blocks host until all issued CUDA calls are complete

Synchronize w.r.t. a specific stream

cudaStreamSynchronize (streamid)

Blocks host until all CUDA calls in streamid are complete

Synchronize using Events

Create specific 'Events', within streams, to use for synchronization

cudaEventRecord (event, streamid)

cudaEventSynchronize (event)

cudaStreamWaitEvent (stream, event)

cudaEventQuery (event)

Stream Scheduling

Fermi hardware has 3 queues

- 1 Compute Engine queue
- 2 Copy Engine queues – one for H2D and one for D2H

CUDA operations are dispatched to HW in the sequence they were issued

- Placed in the relevant queue
- Stream dependencies between engine queues are maintained, but lost within an engine queue

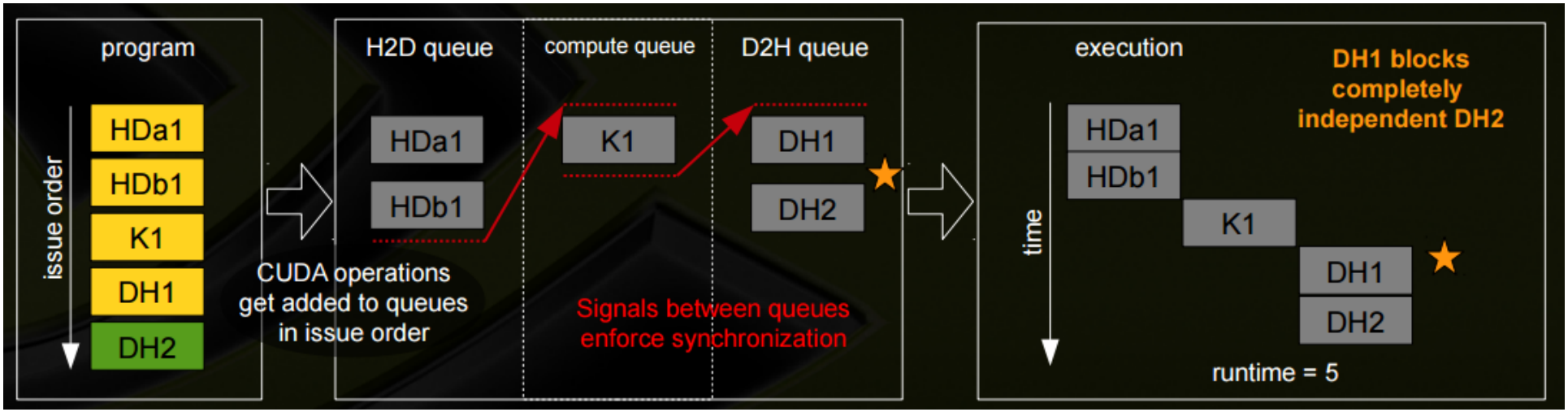
A CUDA operation is dispatched from the engine queue if:

- Preceding calls in the same stream have completed,
- Preceding calls in the same queue have been dispatched and resources available

Example - Blocked Queue

Stream 1 : HDa1, HDb1, K1, DH1 (issued first)

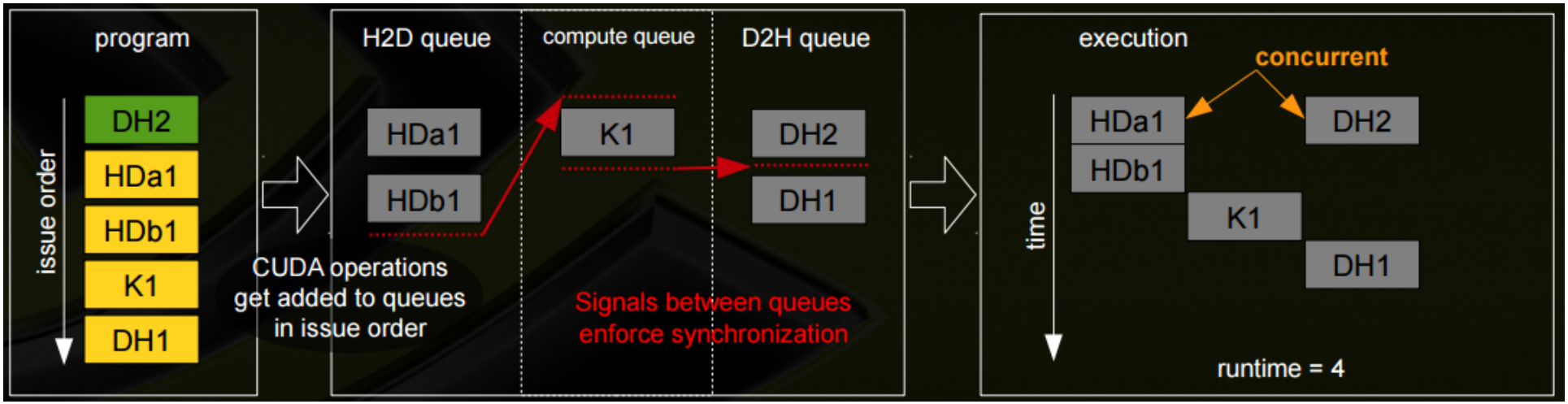
Stream 2 : DH2 (completely independent of stream 1)



Example - Blocked Queue

Stream 1 : HDa1, HDb1, K1, DH1

Stream 2 : DH2 (issued first)

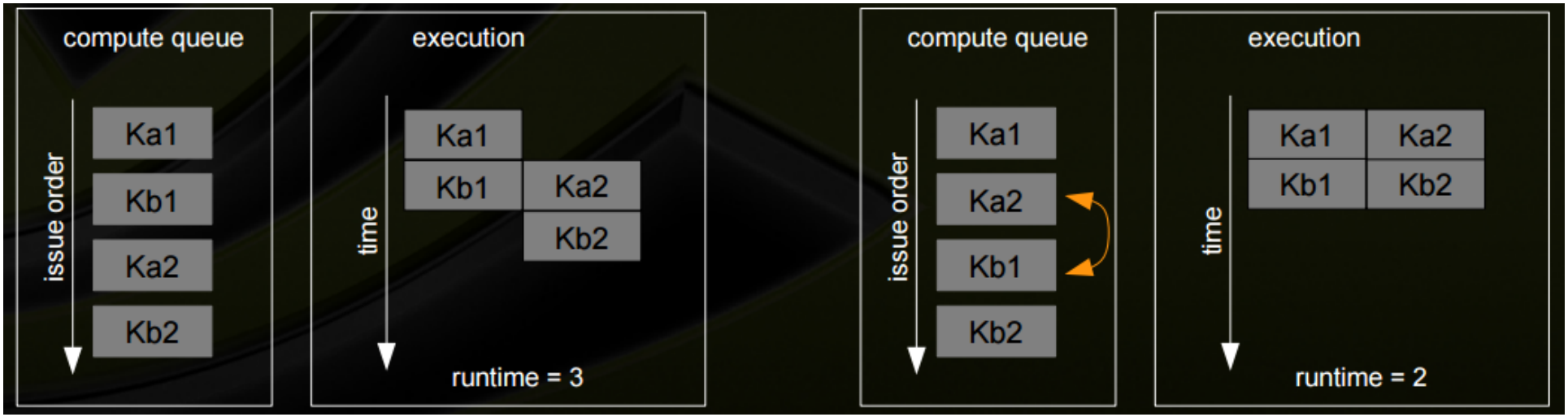


Example - Blocked Kernel

Stream 1 : Ka1, Kb1

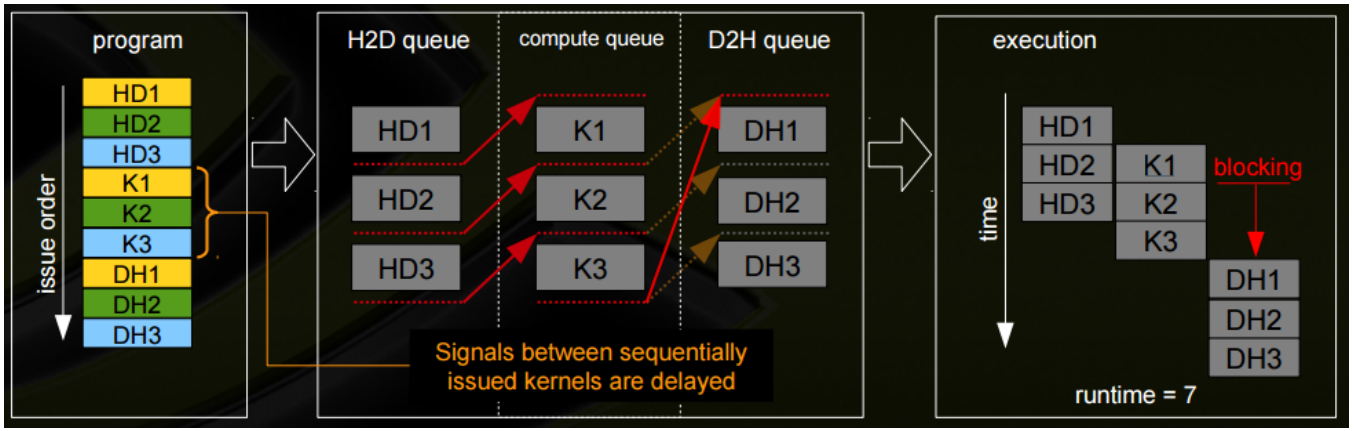
Stream 2 : Ka2, Kb2

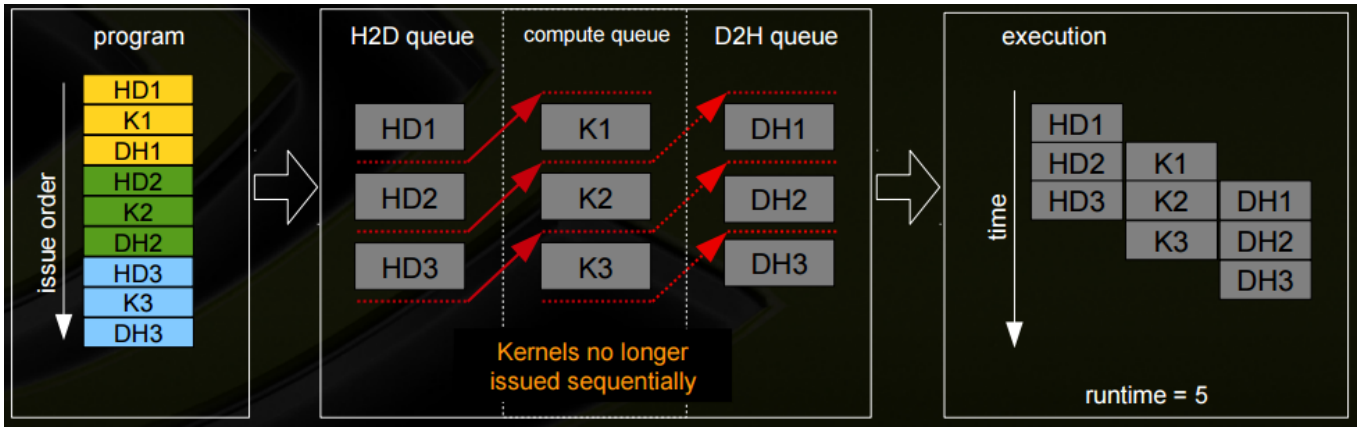
Kernels are similar size, use half of SM resources



Example - Concurrent Kernels

Three streams, each performing (HD, K, DH)

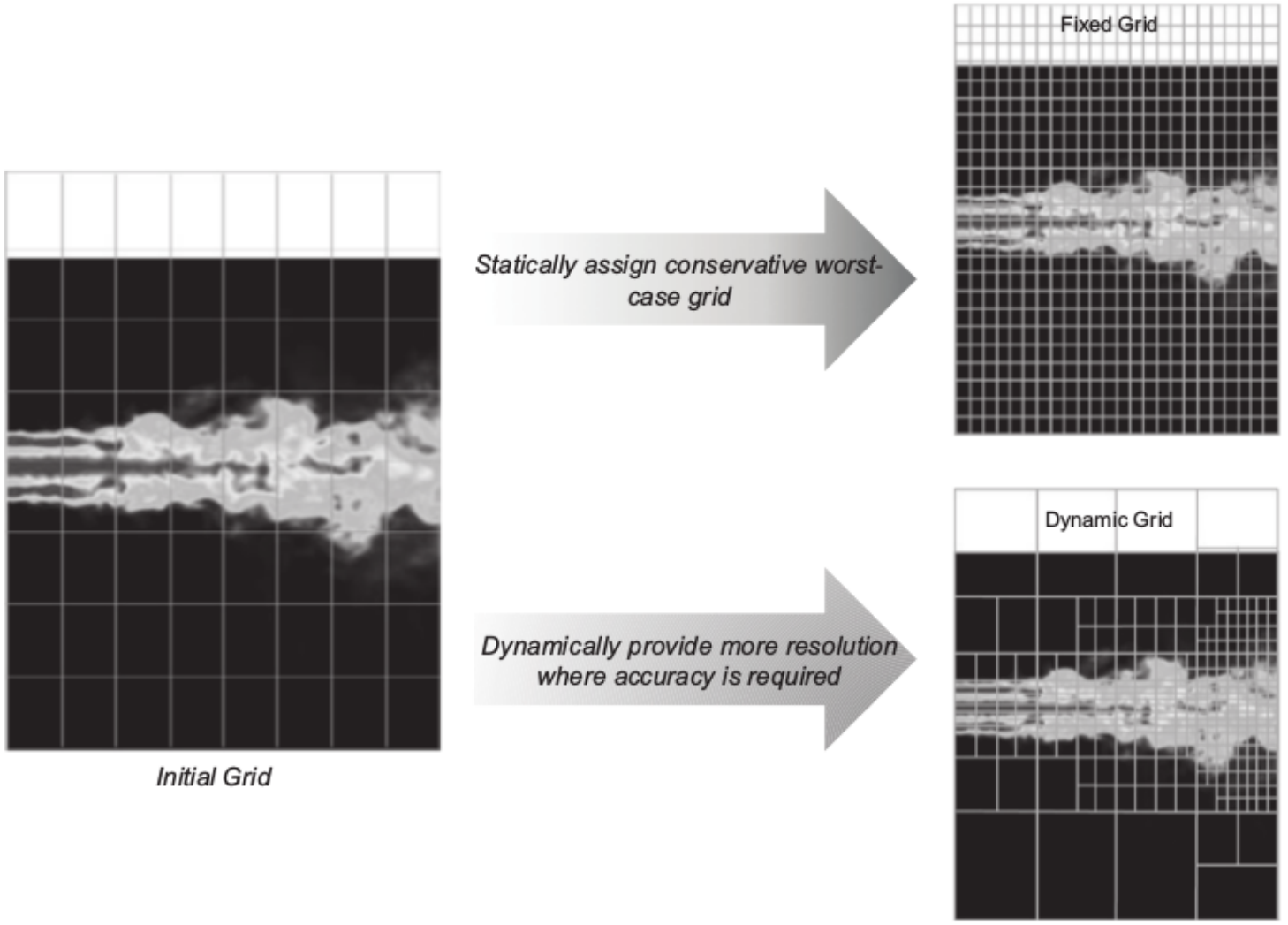




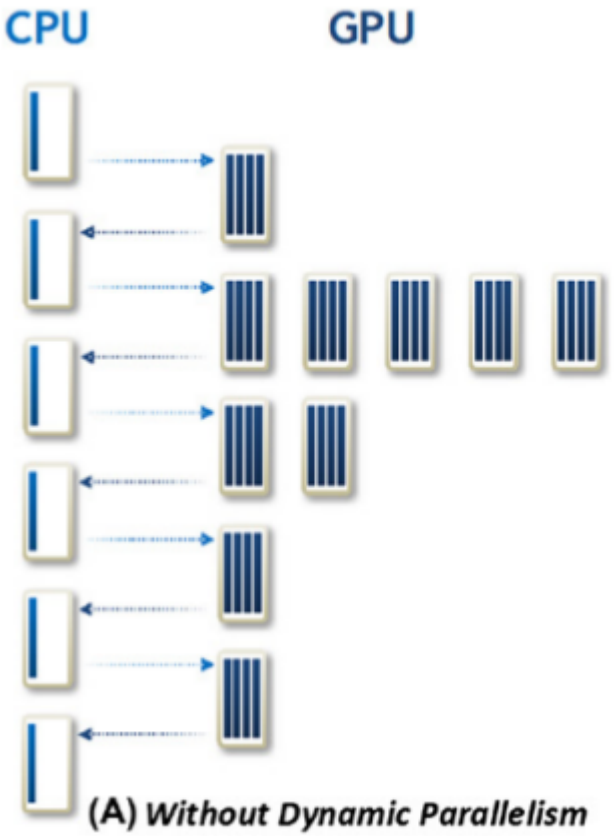
CUDA Events

```
cudaEventRecord(start, 0);
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDev + i * size, inputHost + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel<<<100, 512, 0, stream[i]>>>
              (outputDev + i * size, inputDev + i * size, size);
    cudaMemcpyAsync(outputHost + i * size, outputDev + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
}
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
```

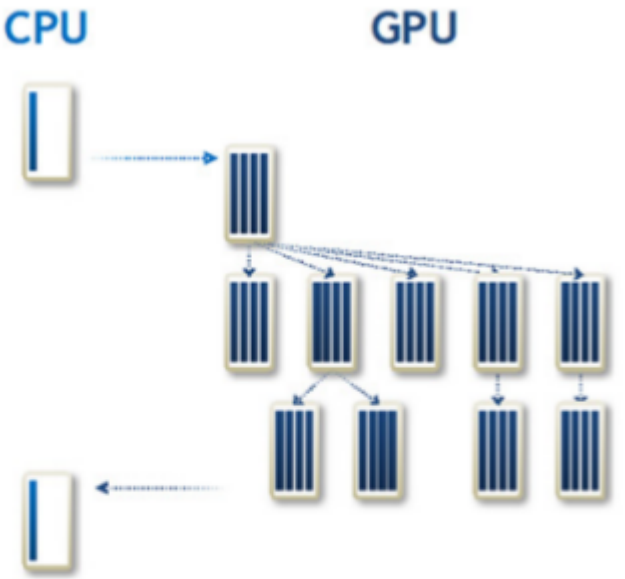
Dynamic Parallelism



Dynamic Parallelism



(A) *Without Dynamic Parallelism*



(B) *With Dynamic Parallelism*

Dynamic Parallelism

Algorithms using hierarchical data structures, such as adaptive grids;

Algorithms using recursion, where each level of recursion has parallelism, such as quicksort;

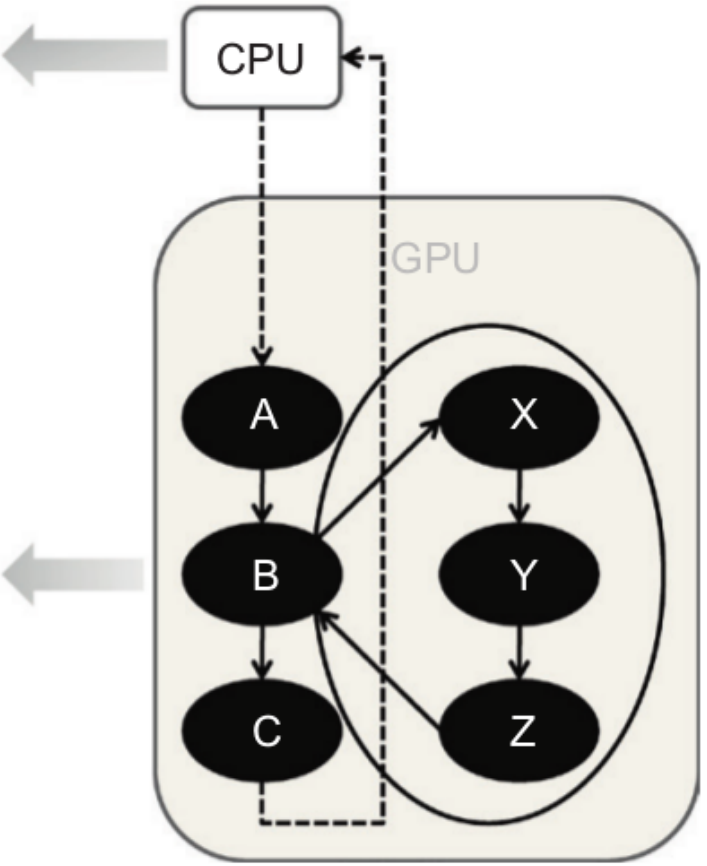
Algorithms where work is naturally split into independent batches, where each batch involves complex parallel processing but cannot fully use a single GPU

Available in CUDA 5.0 and later on devices of Compute Capability 3.5 or higher

Kernel Launch Inside a Kernel

```
int main() {
    float *data;
    setup(data);

    A <<< ... >>> (data);
    B <<< ... >>> (data);
    C <<< ... >>> (data);

    cudaDeviceSynchronize();
    return 0;
}
```

```
__global__ void B(float *data)
{
    do_stuff(data);

    X <<< ... >>> (data);
    Y <<< ... >>> (data);
    Z <<< ... >>> (data);
    cudaDeviceSynchronize();

    do_more_stuff(data);
}
```



Without Dynamic Parallelism

```
__global__ void kernel(unsigned  int* start, unsigned int* end,float* someData,
     float* moreData) {

     unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
     doSomeWork(someData[i]);

     for(unsigned int j = start[i]; j < end[i]; ++j) {
          doMoreWork(moreData[j]);
     }

}
```

The work in the loop can be profitably performed in parallel (but no parallelism)

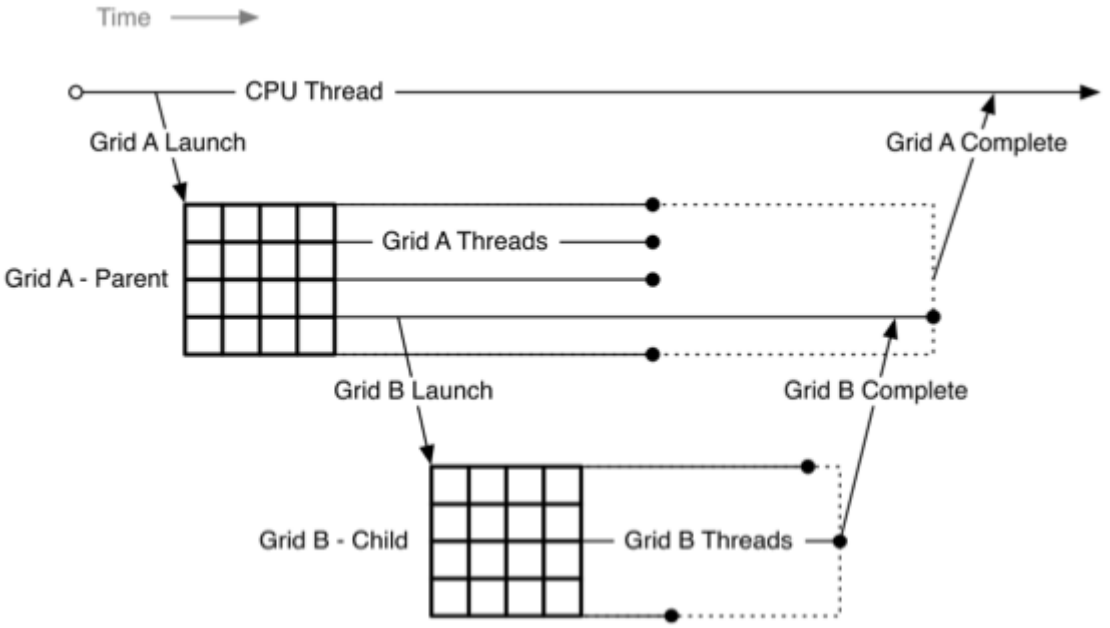
The number of iterations in the loop can vary significantly between threads in the same warp (control divergence)

With Dynamic Parallelism

```
__global__ void kernel_parent(unsigned int* start, unsigned int* end,
     float* someData, float* moreData) {

     unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
     doSomeWork(someData[i]);

     kernel_child <<< ceil((end[i]-start[i])/256.0) , 256 >>>
          (start[i], end[i], moreData);

}

__global__ void kernel_child(unsigned int start, unsigned int end,
     float* moreData) {

     unsigned int j = start + blockIdx.x*blockDim.x + threadIdx.x;

     if(j < end) {
          doMoreWork(moreData[j]);
     }

}
```

Parent-Child Grid

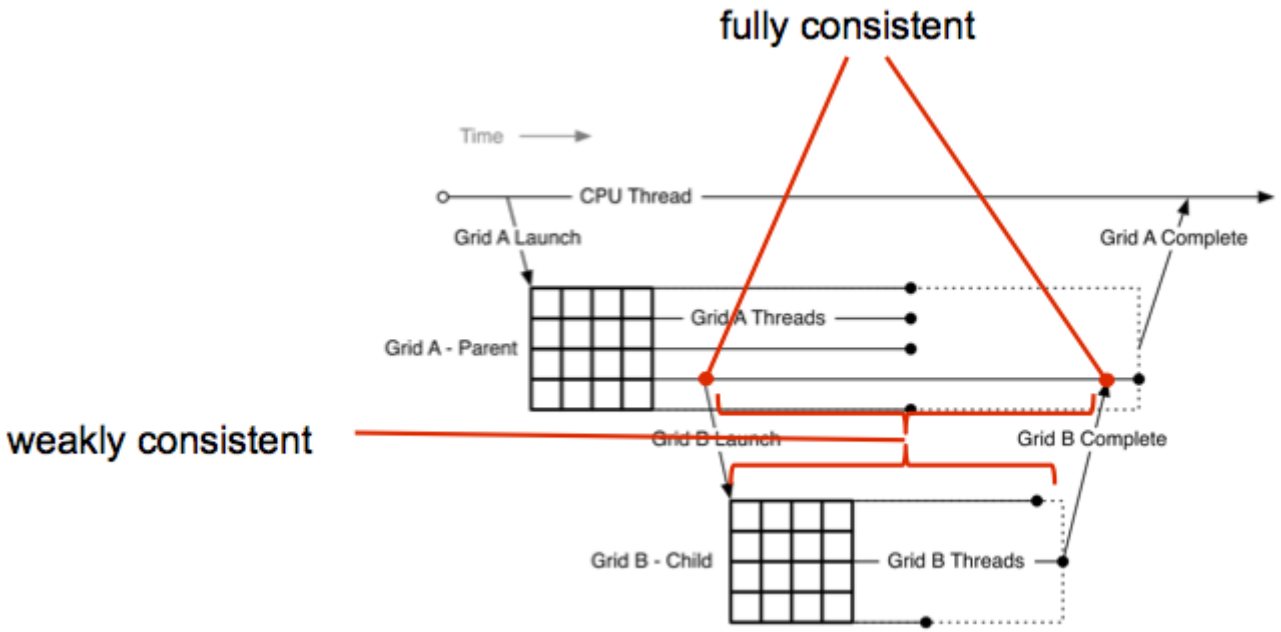
Grid: a group of blocks of threads that are running a kernel



Memory Consistency

When the child grid is created by a parent thread

When the child grid completes as signaled by the completion



Passing Pointers to Child Grids

+ Global memory
+ Zero-copy memory
+ Constant memory

- Shared memory
- Local memory

Local Memory Visibility

```
__device__ int value;
__device__ void x() {
    value = 5;
    child<<< 1, 1 >>>(&value);
}
```

(A) Valid–"value" is global storage

```
__device__ void y() {
    int value = 5;
    child<<< 1, 1 >>>(&value);
}
```

(B) Invalid–"value" is local storage

Synchronization

cudaDeviceSynchronize()

- A thread that invokes this call will wait until all kernels launched by any thread in the thread-block have completed
- Invoked by one thread in the block
- This does not mean that all threads in the block will wait
  - must also be followed by __syncthreads()
    - Invoked by all threads in the block

If a parent kernel launches other child kernels and does not explicitly synchronize on the completion of those kernels, then the runtime will perform the synchronization implicitly before the parent kernel terminates

Streams

The scope of a stream is private to the block in which the stream was created

Streams created by a thread may be used by any thread within the same thread-block

Stream handles should not be passed to other blocks or child/parent kernels

`cudaStreamCreate(): compile error`
`cudaStreamCreateWithFlags()` **with** `cudaStreamNonBlocking` **flag**

Multi Device System

CUDA calls are issued to the current GPU

cudaSetDevice() sets the current GPU

Current GPU can be changed while async calls (kernels, memcopies) are running

```
cudaSetDevice( 0 );
kernel<<<...>>>(...);
cudaMemcpyAsync(...);
cudaSetDevice( 1 );
kernel<<<...>>>(...);
```

References

https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf

https://docs.nvidia.com/cuda/cuda-c-programming-guide/