

# GPU Programming

(in Cuda)

---

Julian Gutierrez

NUCAR

Session 7

# Outline

- Improving Host to Device Memory Transfers
  - Pinned Memory
  - Zero Copy
  - Unified Memory
- Concurrency
- Dynamic Parallelism

# Improving Host to Device Memory Transfers

# Memory Transfers

- We want to minimize the amount of time it takes to transfer data between host and device.
- Device to host memory bandwidth is much lower than device to device bandwidth

# Memory Transfers

- We want to minimize the amount of time it takes to transfer data between host and device.
- How can we do this?

# Memory Transfers

- We want to minimize the amount of time it takes to transfer data between host and device.
- How can we do this?
  - Reduce the amount of data we have to transfer! (if possible)

# Memory Transfers

- We want to minimize the amount of time it takes to transfer data between host and device.
- How can we do this?
  - Reduce the amount of data we have to transfer! (if possible)
  - Achieve highest memory bandwidth between host and device.  
(Possible when using page-locked (or “pinned”) memory, we are going to talk about it in this session)

# Memory Transfers

- We want to minimize the amount of time it takes to transfer data between host and device.
- How can we do this?
  - Reduce the amount of data we have to transfer! (if possible)
  - Achieve highest memory bandwidth between host and device. (Possible when using page-locked (or “pinned”) memory, we are going to talk about it in this session)
  - Batching many small transfers into one larger transfer performs much better because it eliminates most of the per-transfer overhead



# Memory Transfers

- We want to minimize the amount of time it takes to transfer data between host and device.
- How can we do this?
  - Reduce the amount of data we have to transfer! (if possible)
  - Achieve highest memory bandwidth between host and device. (Possible when using page-locked (or “pinned”) memory, we are going to talk about it in this session)
  - Batching many small transfers into one larger transfer performs much better because it eliminates most of the per-transfer overhead
  - Data transfers between the host and device can sometimes be overlapped with kernel execution and other data transfers (we will talk about this in this session as well)

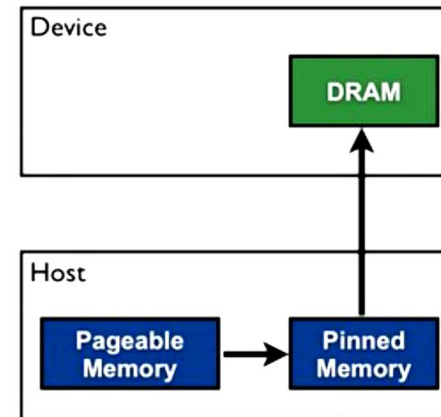
# Memory Transfers

- Remember
  - When deciding whether to run on the GPU or on the CPU, we should consider the kernel execution time and the time it takes to copy data back and forth.
  - We need to consider the cost of moving data across the PCI-e bus.

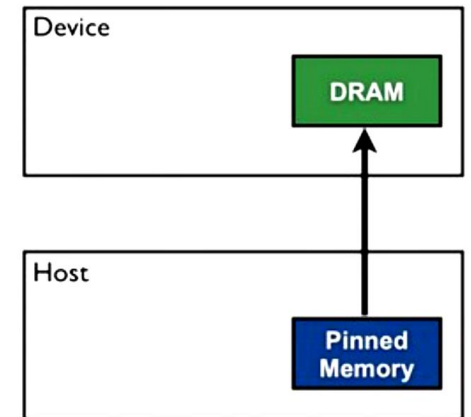
# Pinned Memory (Page-Locked Data Transfers)

- Host (CPU) data allocations are pageable by default.
  - The GPU cannot access data directly from pageable host memory.
- Due to this, Cuda driver must:
  - Allocate a temporary page-locked block, or “pinned”
  - Copy host data to the pinned block
  - Transfer from pinned to device
  - Delete pinned block

*Pageable Data Transfer*



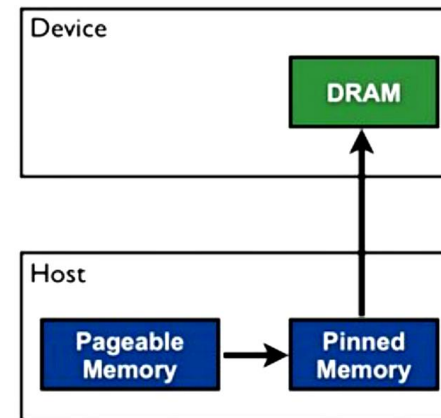
*Pinned Data Transfer*



# Pinned Memory (Page-Locked Data Transfers)

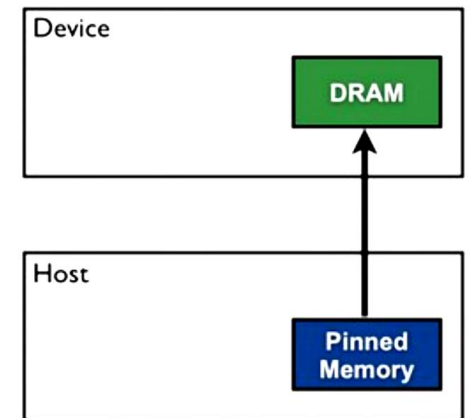
- `cudaMallocHost()`
  - Prevents OS from paging host memory
  - Allows PCI-e DMA to run at full speed
- `cudaFreeHost()`
- Allocations can fail so you need to check errors.

*Pageable Data Transfer*



`cudaMalloc`

*Pinned Data Transfer*



`cudaMallocHost`

```
// allocate and initialize
h_aPageable = (float*)malloc(bytes); // host pageable
h_bPageable = (float*)malloc(bytes); // host pageable
checkCuda( cudaMallocHost((void**)&h_aPinned, bytes) ); //
host pinned
checkCuda( cudaMallocHost((void**)&h_bPinned, bytes) ); //
host pinned
checkCuda( cudaMalloc((void**)&d_a, bytes) ); // device
```

# Pinned Memory (Page-Locked Data Transfers)

- You should not over-allocate pinned memory. Doing so can reduce overall system performance because it reduces the amount of physical memory available to the operating system and other programs.
- Transfers between the host and device are the slowest link of data movement involved in GPU computing, so you should take care to minimize transfers.

# Zero Copy

- Access host memory directly from device code
  - Transfers implicitly performed as needed by device code
- All set-up is done on host using mapped memory
- Use zero-copy operations on integrated GPUs for CUDA Toolkit version 2.2 and later.

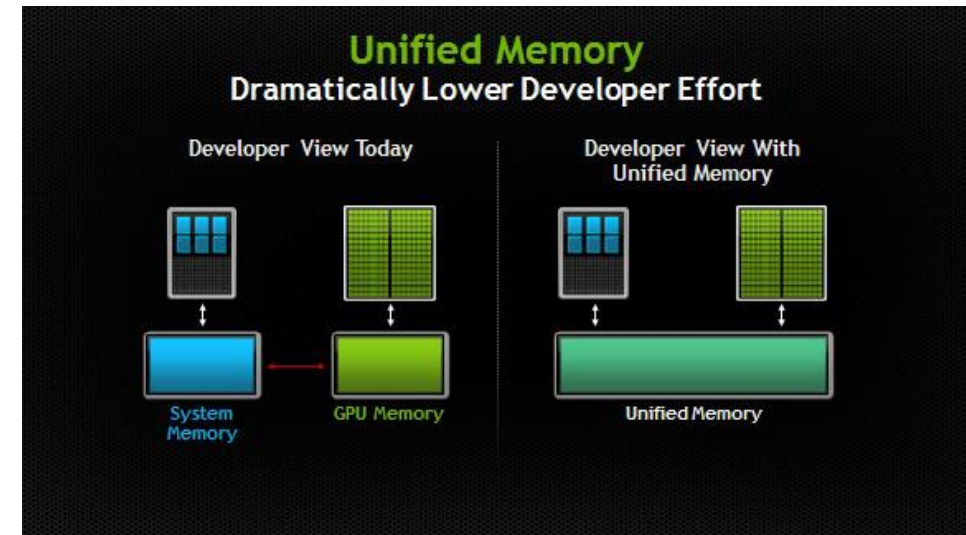
```
float *a_h, *a_map;  
  
...  
  
cudaGetDeviceProperties(&prop, 0);  
  
if (!prop.canMapHostMemory) exit(0);  
  
cudaSetDeviceFlags(cudaDeviceMapHost);  
  
cudaHostAlloc(&a_h, nBytes,  
cudaHostAllocMapped);  
  
cudaHostGetDevicePointer(&a_map, a_h, 0);  
  
kernel<<<gridSize, blockSize>>>(a_map);
```

# Zero Copy

- Zero copy will always be a win for integrated devices that utilize CPU memory
- Zero copy will be faster if data is only read/written from/to global memory once, for example:
  - Copy input data to GPU memory
  - Run one kernel
  - Copy output data back to CPU memory
- Potentially easier and faster alternative to using `cudaMemcpyAsync`.
- Some devices use pointers that are 32 bit so there is a limit of 4GB per context.
- **Note:** Mapped pinned host memory allows you to overlap CPU-GPU memory transfers with computation while avoiding the use of CUDA streams. But since any repeated access to such memory areas causes repeated PCIe transfers, consider creating a second area in device memory to manually cache the previously read host memory data.

# Unified Memory

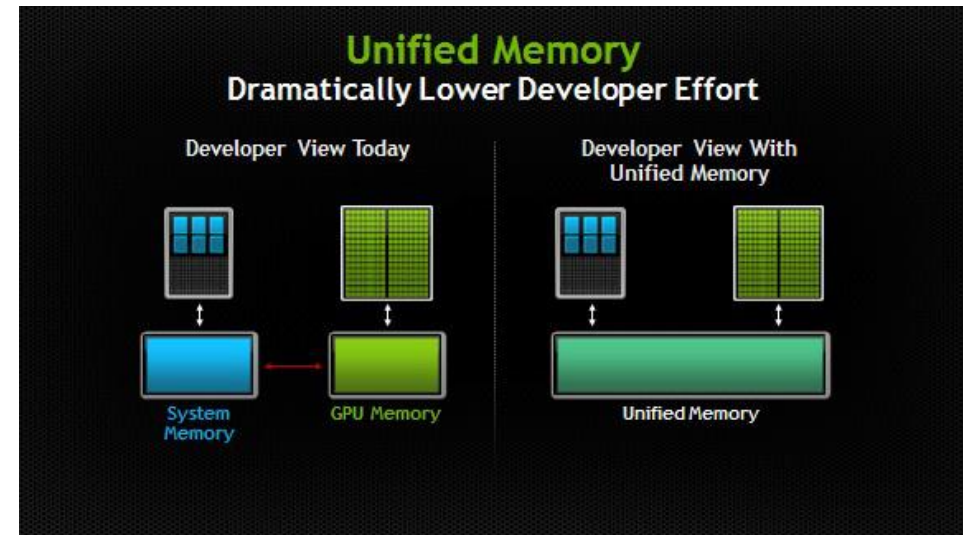
- In a typical PC or cluster node today, the memories of the CPU and GPU are physically distinct and separated by the PCI-Express bus.
- Data that is shared between the CPU and GPU must be allocated in both memories, and explicitly copied between them by the program.





# Unified Memory

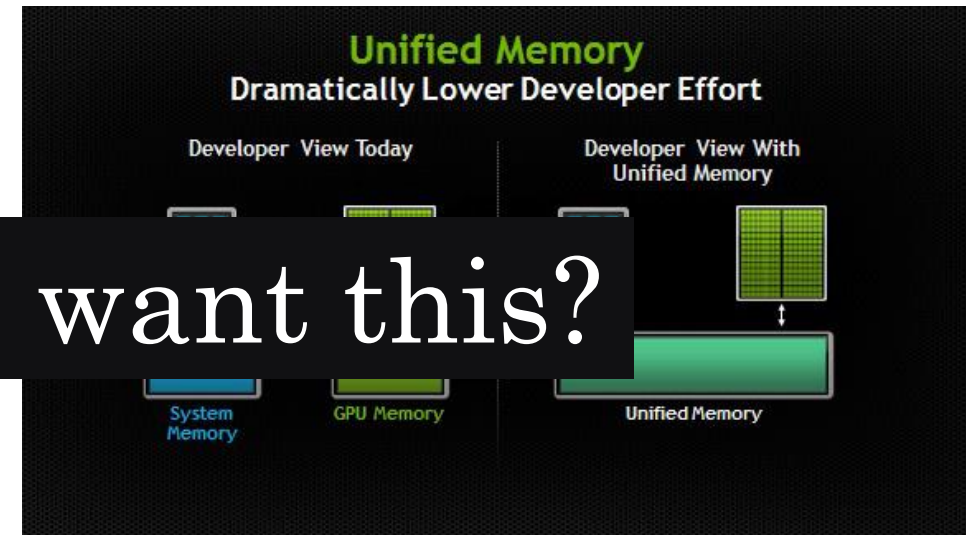
- Unified Memory creates a pool of managed memory that is shared between the CPU and GPU
- Managed memory is accessible to both the CPU and GPU using a single pointer
- The system automatically *migrates* data allocated in Unified Memory between host and device



# Unified Memory

- Unified Memory creates a pool of managed memory that is shared between the CPU and GPU
- Managed memory is accessible to both the CPU and GPU with a single pointer
- The system automatically *migrates* data allocated in Unified Memory between host and device

Why do we want this?



# Unified Memory

- The driver handles the lookup and transfers
- Code: CPU and CUDA 6 with UM is basically the same.

## 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

- Notice the simplicity
  - One pointer used by the Host (CPU), and the GPU (kernels)

## 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

- Allows Performance through data locality
  - Data migration happens on demand between CPU and GPU.
- Despite this, tuned programs that don't use UM will perform better due to added complexity in the Driver.

## 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);  
}
```

# Concurrency

# Streams

- Cuda stream:
  - A stream in CUDA is a sequence of operations that execute on the device in the order in which they are issued by the host code.
  - Operations from different streams can be interleaved
  - Stream IDs are used as arguments to async calls and kernel launches

```
cudaStream_t stream1, stream2, stream3, stream4 ;
cudaStreamCreate ( &stream1 ) ;
...
cudaMalloc ( &dev1, size ) ;
cudaMallocHost ( &host1, size ) ;
...
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 () ;
...
```

// pinned memory required on host

} potentially overlapped

- **Fully asynchronous / concurrent**
- **Data used by concurrent operations should be independent**



# Streams

- Cuda stream:

- A stream in which they are executed
- Operations follow the order in which they are submitted
- Stream IDs are used to identify streams

## cudaMemcpyAsync

- Asynchronous host-device memory copy returns control immediately to CPU
- Requires pinned host memory (allocated with cudaMallocHost)

order in

```
cudaStream_t stream1, stream2, stream3, stream4 ;
cudaStreamCreate ( &stream1 ) ;
...
cudaMalloc ( &dev1, size ) ;
cudaMallocHost ( &host1, size ) ;
...
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 () ;
...
```

// pinned memory required on host

} potentially overlapped

- Fully asynchronous / concurrent
- Data used by concurrent operations should be independent



# Streams

- Synchronization
  - `cudaThreadSynchronize()`
    - Blocks until all previously issued CUDA calls from a CPU thread are complete
  - `cudaStreamSynchronize (stream):`
    - Blocks until all CUDA calls issued to given stream have completed.
  - `cudaStreamQuery (stream)`
    - Indicates whether stream is idle.
    - Doesn't block CPU thread
  - `CudaStreamWaitEvent (event)`
    - wait for event in a stream

# Streams

- If we have multiple streams, how many different ways exist to do a complete kernel operation (H2D/Kernel/D2H)?

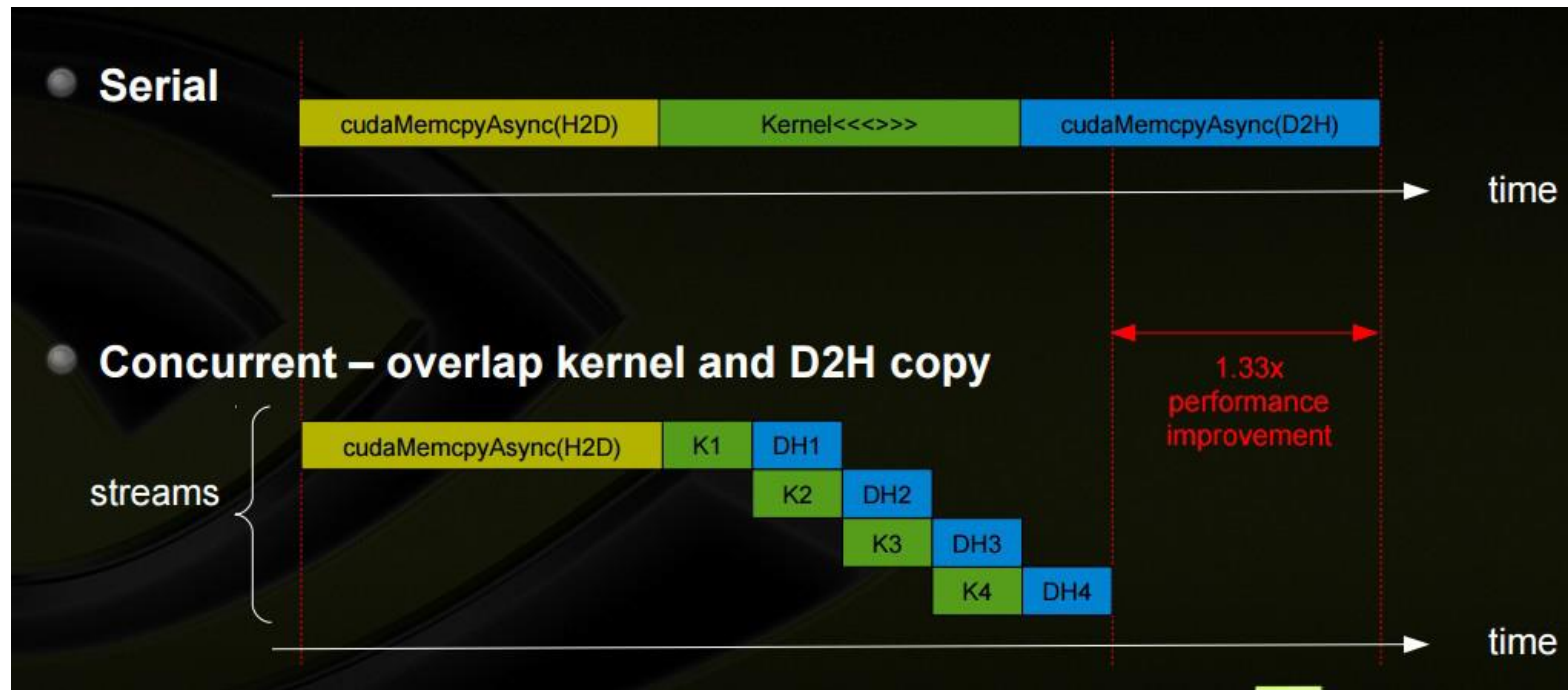
# Streams

- If we have multiple streams, how many different ways exist to do a complete kernel operation (H2D/Kernel/D2H)?



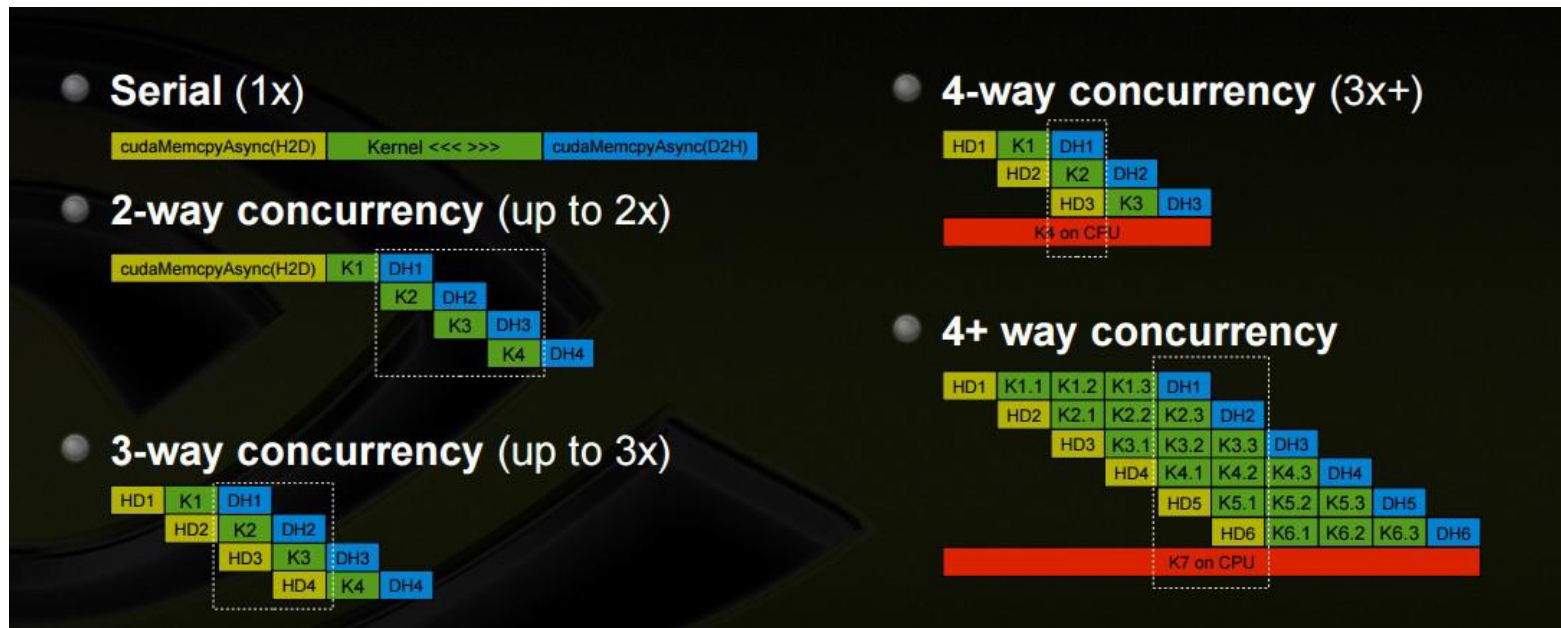
# Streams

- If we have multiple streams, how many different ways exist to do a complete kernel operation (H2D/Kernel/D2H)?



# Streams

- If we have multiple streams, how many different ways exist to do a complete kernel operation (H2D/Kernel/D2H)?





# Streams

- If we have multiple streams, how many different ways exist to do a complete kernel operation (H2D/Kernel/D2H)?

# Streams

- If we have multiple streams, how many different ways exist to do a complete kernel operation (H2D/Kernel/D2H)?
  - We need to take into account how many Copy Engines are available!



# Streams

- If we have multiple streams, how many different ways exist to do a complete kernel operation (H2D/Kernel/D2H)?
  - We need to take into account how many Copy Engines are available!
  - Lets look at 2 methods on how we could do these operations.

# Copy Engines

- Method 1

```
for (int i = 0; i < nStreams; ++i) {  
    int offset = i * streamSize;  
    cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes, cudaMemcpyHostToDevice,  
        kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);  
    cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes, cudaMemcpyDeviceToHost,  
        kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);  
}
```

# Copy Engines

- Method 1
  - CHD1, K1, CDH1, CHD2, K2, CDH2, ...
- Queues:
  - Copy engine
    - CHD1, CDH1, CHD2, CDH2, ...
  - Kernel
    - K1, K2, ...

```
for (int i = 0; i < nStreams; ++i) {  
    int offset = i * streamSize;  
    cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes, cudaMemcpyHostToDevice,  
        kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);  
    cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes, cudaMemcpyDeviceToHost,  
        kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);  
}
```

# Copy Engines

- Method 2

```
for (int i = 0; i < nStreams; ++i) {  
    int offset = i * streamSize;  
    cudaMemcpyAsync(&d_a[offset], &a[offset],  
                    streamBytes, cudaMemcpyHostToDevice, cudaMemcpyHostToDevice)  
}  
  
for (int i = 0; i < nStreams; ++i) {  
    int offset = i * streamSize;  
    kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);  
}  
  
for (int i = 0; i < nStreams; ++i) {  
    int offset = i * streamSize;  
    cudaMemcpyAsync(&a[offset], &d_a[offset],  
                    streamBytes, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToHost)  
}
```

# Copy Engines

- Method 2
  - CHD1, CHD2, ..., K1, K2, ..., CDH1, CDH2, ....
- Queues
  - Copy Engine
    - CHD1, CHD2, ... CDH1, CDH2, ...
  - Kernel
    - K1, K2, ....

```
for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    cudaMemcpyAsync(&d_a[offset], &a[offset],
                    streamBytes, cudaMemcpyHostToDevice, cudaMemcpyHostToDevice)
}

for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
}

for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    cudaMemcpyAsync(&a[offset], &d_a[offset],
                    streamBytes, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToHost)
}
```

# Copy Engines

- When the device has one copy engine

## Sequential Version



## Asynchronous Version 1



## Asynchronous Version 2



Time →

# Copy Engines

- When the device has one copy engine

- METHOD 1
  - CHD1, K1,  
CDH1, CHD2,  
K2, CDH2, ...
- QUEUES:
  - COPY ENGINE
    - CHD1,  
CDH1,  
CHD2,  
CDH2, ...
  - KERNEL
    - K1, K2, ...

## Sequential Version



## Asynchronous Version 1



## Asynchronous Version 2



Time →

# Copy Engines

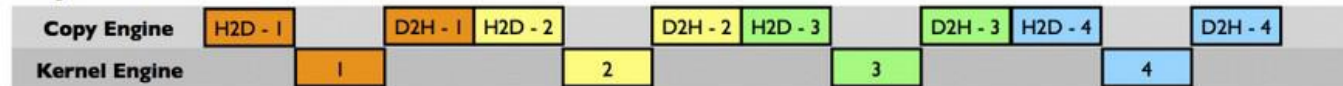
- When the device has one copy engine

- METHOD 2
  - CHD1, CHD2, ..., K1, K2, ..., CDH1, CDH2, ....
- QUEUES
  - COPY ENGINE
    - CHD1, CHD2, ... CDH1, CDH2, ...
  - KERNEL
    - K1, K2, ....

## Sequential Version



## Asynchronous Version 1



## Asynchronous Version 2



Time →



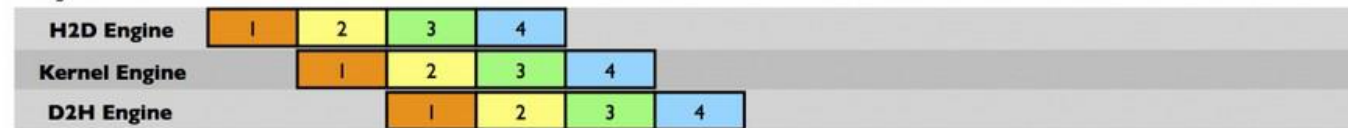
# Copy Engines

- When the device has two copy engines

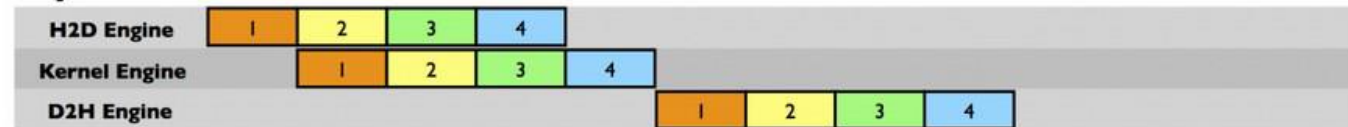
## Sequential Version



## Asynchronous Version 1



## Asynchronous Version 2



Time →

# Copy Engines

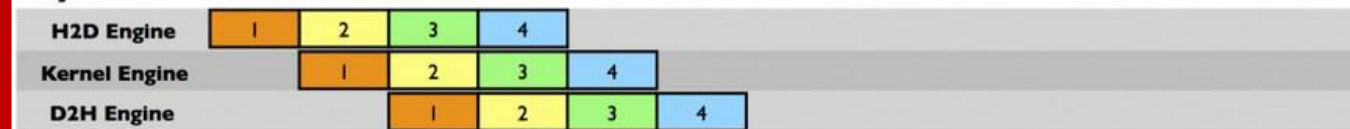
- When the device has two copy engines

- METHOD 1
  - CHD1, K1, CDH1,  
CHD2, K2, CDH2,  
...
- QUEUES:
  - COPY ENGINE
    - CHD1, CDH1,  
CHD2, CDH2,  
...
  - KERNEL
    - K1, K2, ...

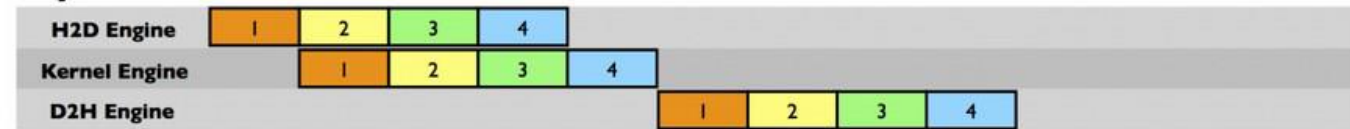
## Sequential Version



## Asynchronous Version 1



## Asynchronous Version 2



Time →

# Copy Engines

- When the device has two copy engines

- METHOD 1

- CHD1, K1, CDH1,  
CHD2, K2, CDH2,  
...

- QUEUES:

- COPY ENGINE

- CHD1, **CDH1**,  
CHD2, CDH2,  
...

- KERNEL

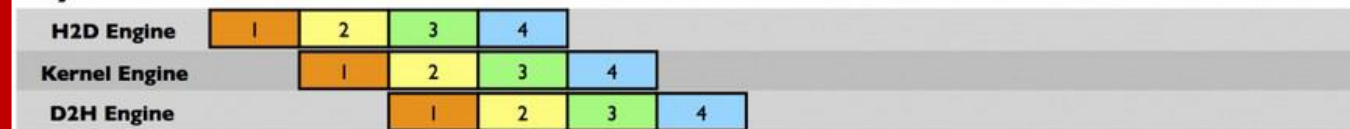
- K1, K2, ...

Added to the other Copy Engine

## Sequential Version



## Asynchronous Version 1



## Asynchronous Version 2



Time →

# Copy Engines

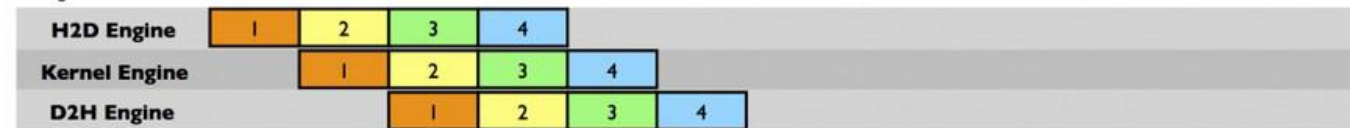
- When the device has two copy engines

- METHOD 2
  - CHD1, CHD2, ..., K1, K2, ..., CDH1, CDH2, ....
- QUEUES
  - COPY ENGINE
    - CHD1, CHD2, ... CDH1, CDH2, ...
  - KERNEL
    - K1, K2, ....

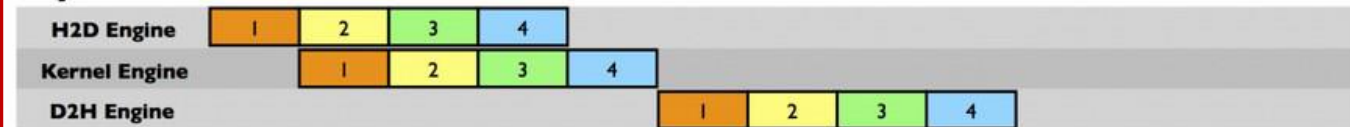
## Sequential Version



## Asynchronous Version 1



## Asynchronous Version 2



Time →

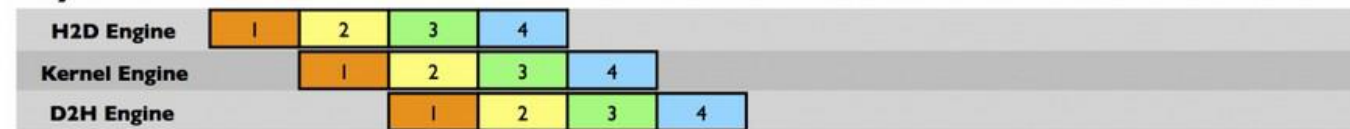
# Copy Engines

- When device has two copy engines

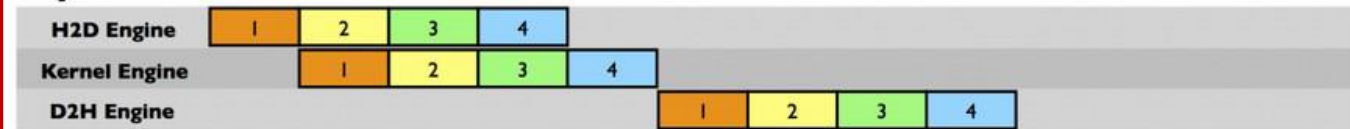
## Sequential Version



## Asynchronous Version 1



## Asynchronous Version 2



Time →

- METHOD 2
  - CHD1, CHD2, ..., K1, K2, ..., CDH1, CDH2, ....
- QUEUES
  - COPY ENGINE
    - CHD1, CHD2, ... CDH1, CDH2, ...
  - KERNEL
    - K1, K2, ....

Added to the other Copy Engine

# Copy Engines

- When device has two copy engines

- METHOD 2

- CHD1, CHD2, ..., K1, K2, ..., CDH1, CDH2, ....

- QUEUES

- COPY ENGINE

- CHD1, CHD2, ... CDH1, CDH2, ...

- KERNEL

- K1, K2, ....

Added to the other Copy Engine

## Sequential Version



The D2H engine won't be engaged until all kernels finish

## Asynchron



## Asynchronous Version 2



Time

# Streams again

- Now that we've copied data efficiently and overlapped with execution. Another simple example:

```
int main()
{
    const int num_streams = 8;

    cudaStream_t streams[num_streams];
    float *data[num_streams];

    for (int i = 0; i < num_streams; i++) {
        cudaStreamCreate(&streams[i]);

        cudaMalloc(&data[i], N * sizeof(float));

        // launch one worker kernel per stream
        kernel<<<1, 64, 0, streams[i]>>>(data[i], N);

        // launch a dummy kernel on the default stream
        kernel<<<1, 1>>>(0, 0);
    }

    cudaDeviceReset();

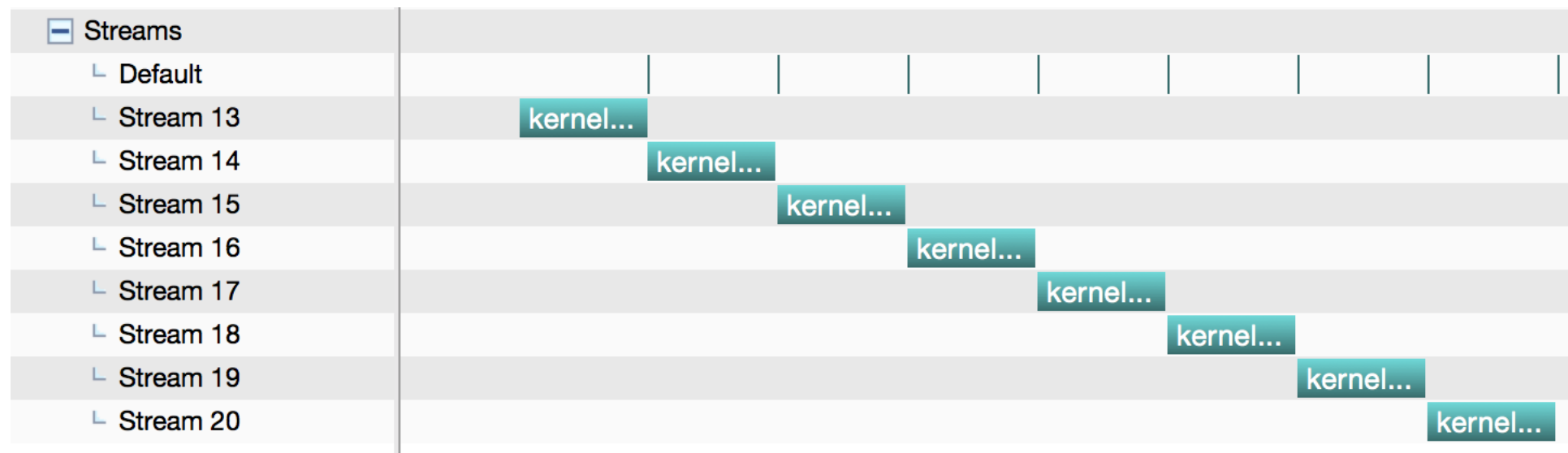
    return 0;
}
```

# Streams again

- When you execute asynchronous CUDA commands without specifying a stream:
  - runtime uses the default stream.
  - Before CUDA 7, the default stream is a special stream which implicitly synchronizes with all other streams on the device.
- CUDA 7 introduces a new option to use an independent default stream for every host thread, which avoids the serialization of the legacy default stream.
- `--default-stream`



# Streams again



# Streams again

--default-stream per-thread

Streams	
Stream 13	kernel(float*, int)
Stream 14	
Stream 15	kernel(float*, int)
Stream 16	kernel(float*, int)
Stream 17	kernel(float*, int)
Stream 18	kernel(float*, int)
Stream 19	kernel(float*, int)
Stream 20	kernel(float*, int)
Stream 21	kernel(float*, int)

# Dynamic Parallelism

# Dynamic Parallelism

- Device-side kernel launches
  - Kepler GK110 architecture
  - Typical use cases
    - Dynamic load balancing
    - Data-dependent execution
    - Recursion
    - Library calls from kernels
- Programmability and maintainability

# Dynamic Parallelism

- Device-side kernel launches
  - Kepler GK110 architecture
  - Typical use cases
    - Dynamic load balancing
    - Data-dependent execution
    - Recursion
    - Library calls from kernels
- Programmability and maintainability



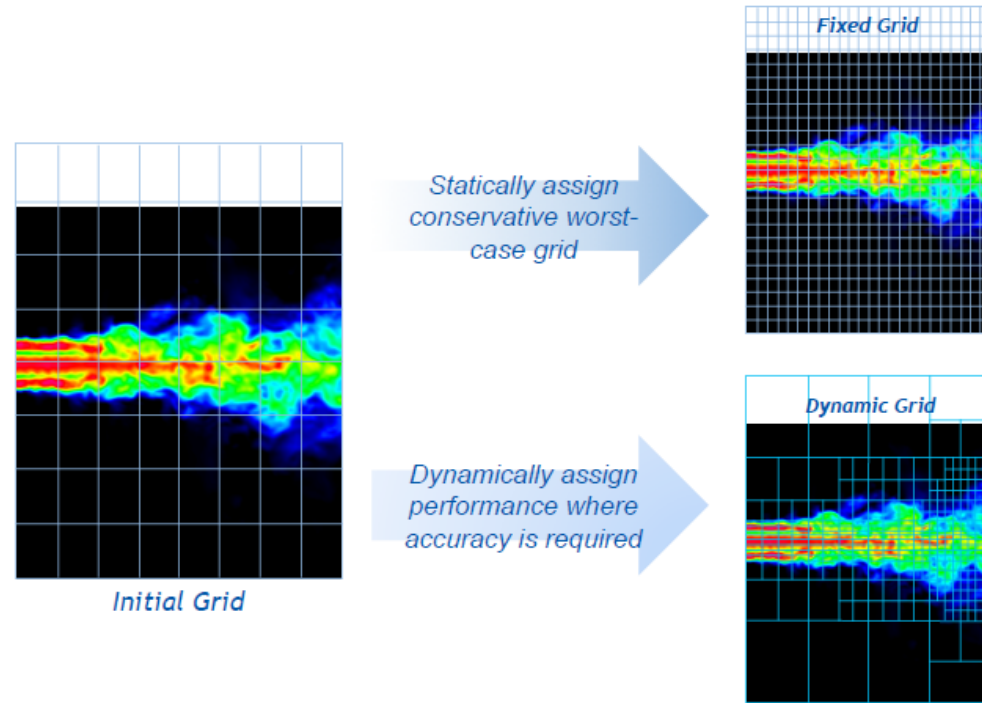
Fermi: Only CPU can generate GPU work



Kepler: GPU can generate work for itself

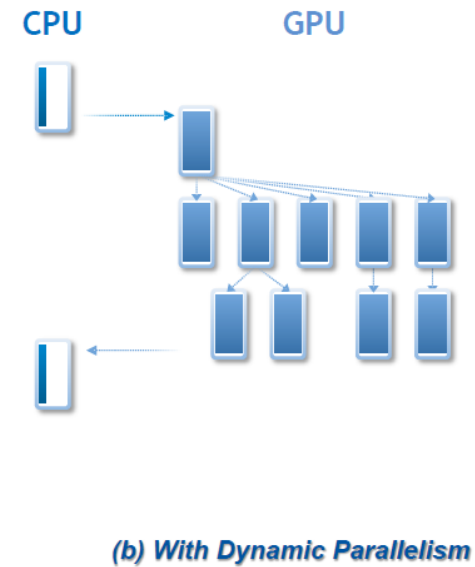
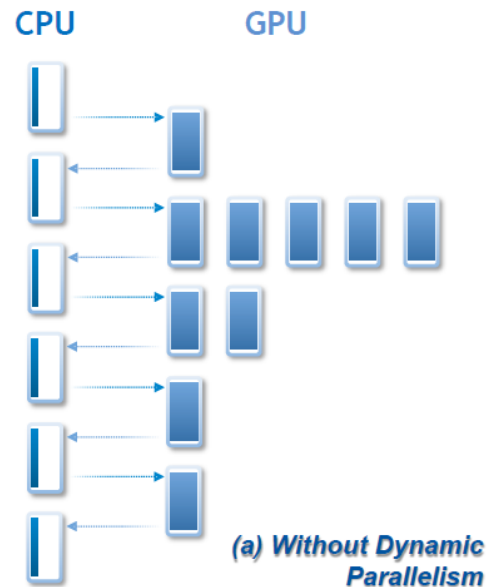
# Dynamic Parallelism

- Fixed grid vs dynamic grid for a turbulence simulation model



# Dynamic Parallelism

- CPU-GPU without and with dynamic parallelism



- Nested dependencies

- Nested dependencies

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

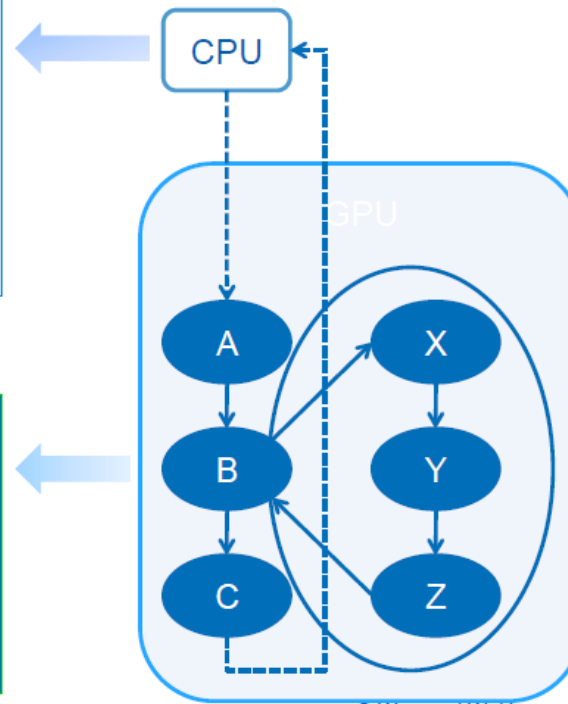
    A <<< ... >>> (data);
    B <<< ... >>> (data);
    C <<< ... >>> (data);

    cudaDeviceSynchronize();
    return 0;
}
```

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

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

    do_more_stuff(data);
}
```





# Dynamic Parallelism

## • Syntax

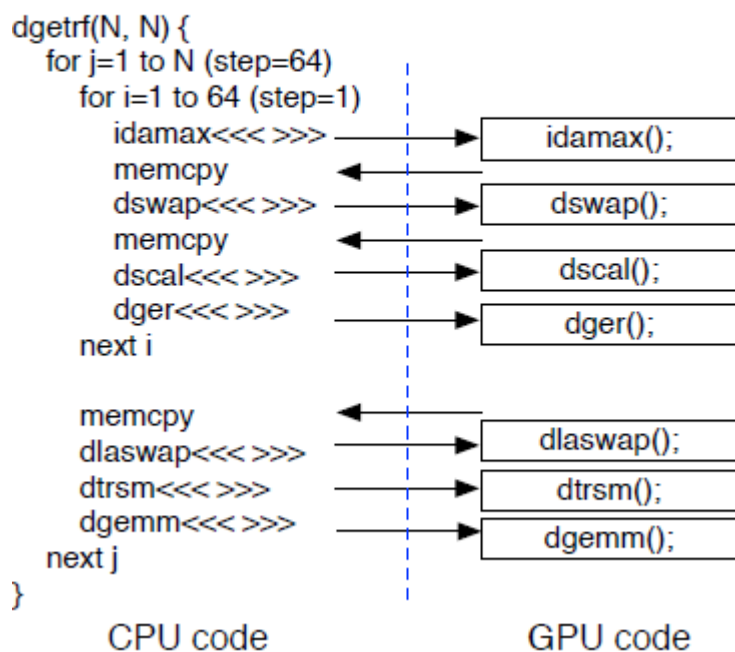
```
Kernel_name <<<Dg, Db, Ns, S>>> ([kernel arguments]);
```

- Dg is of type dim3 and specifies the dimensions and size of the grid
- Db is of type dim3 and specifies the dimensions and size of each thread block
- Ns is of type size\_t and specifies the number of bytes of shared memory that is dynamically allocated per thread block for this call
- S is of type cudaStream\_t and specifies the stream associated with this call

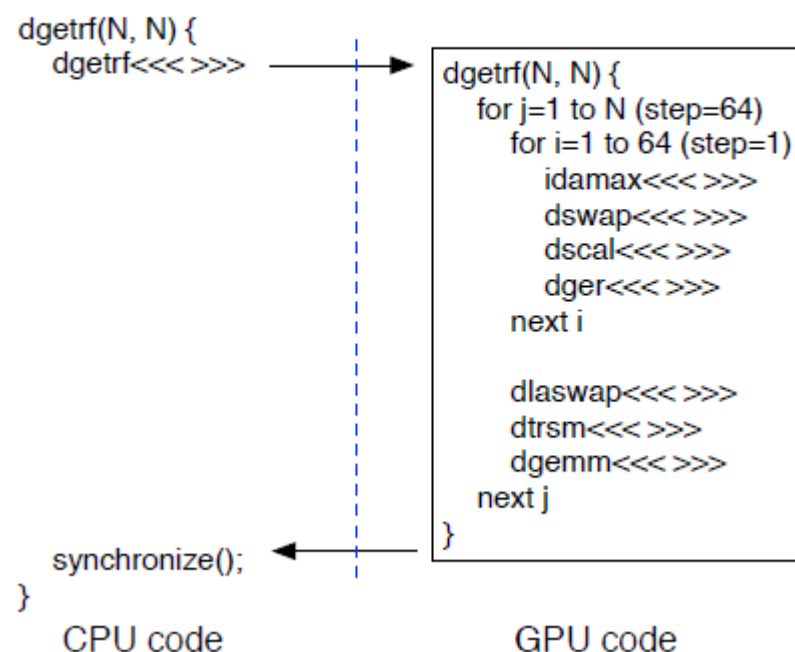
# Dynamic Parallelism

- Example

LU decomposition (Fermi)

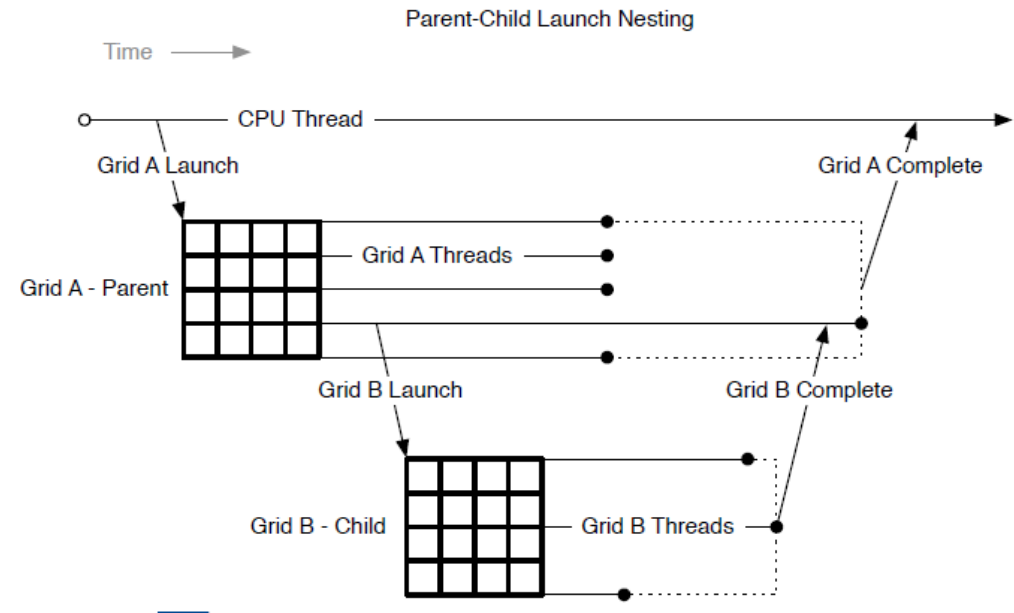


LU decomposition (Kepler)



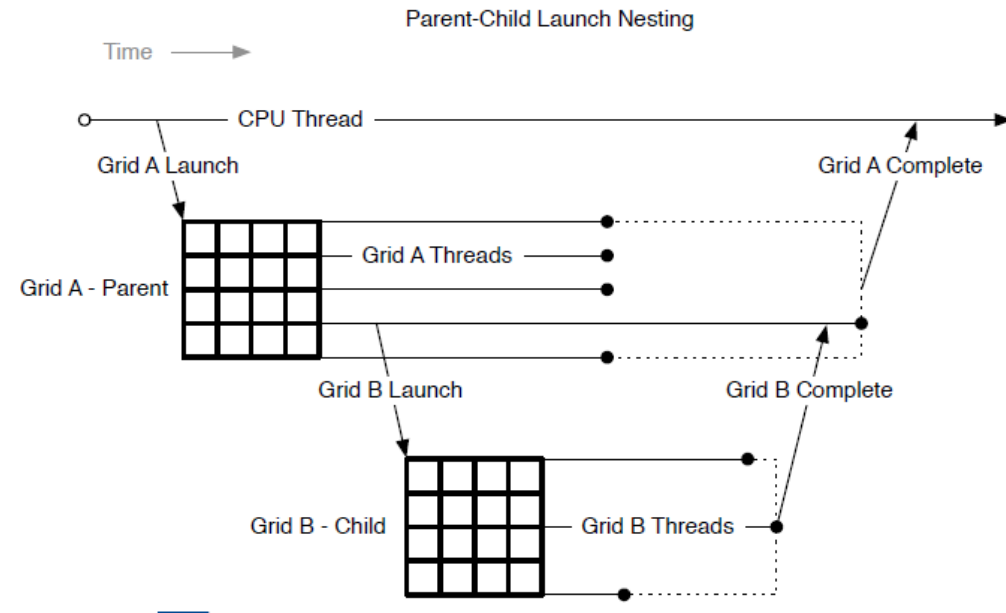
# Dynamic Parallelism

- Synchronization
  - Parent to child:  
memory consistency
  - Child to parent: after  
`cudaDeviceSynchronize()`



# Dynamic Parallelism

- Memory Model
  - Child sees parent state at time of launch
  - Parent sees child writes after sync
  - Texture changes by child are visible to parent after sync.
    - Sync == tex cache invalidate
  - Constants are immutable
  - Local and shared memory are private



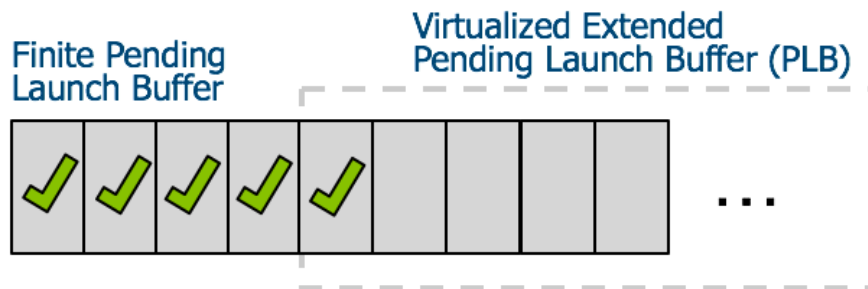
# Dynamic Parallelism

- Launch pool size
  - Fixed-size pool: default 2048
  - Variable-size pool

Before CUDA 6.0

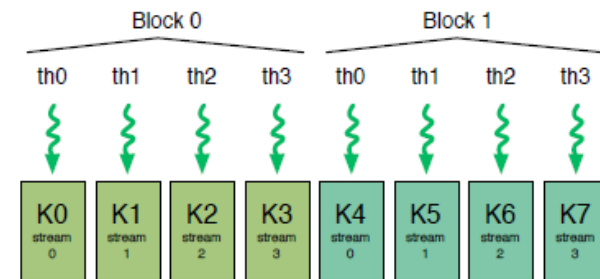
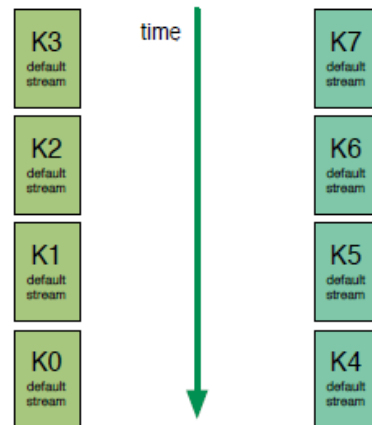
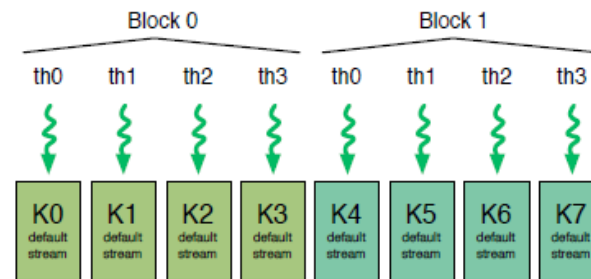


Since CUDA 6.0



# Dynamic Parallelism

- Streams
  - To guarantee concurrency



# Dynamic Parallelism

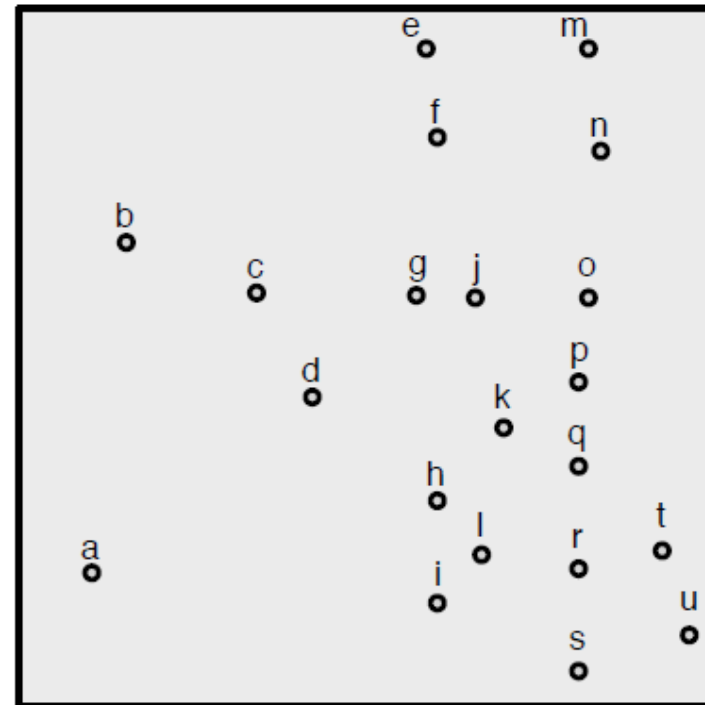
- Another Example
  - Quadtrees
    - A quadtree is a tree data structure in which each internal node has exactly four children.
    - Most often used to partition a two-dimensional space by recursively subdividing it into four quadrants or regions.



Example of image compression through Qtrees

# Dynamic Parallelism

- Another Example
  - Quadrees
    - Partitioning a 2D space by recursively dividing it into four quadrants

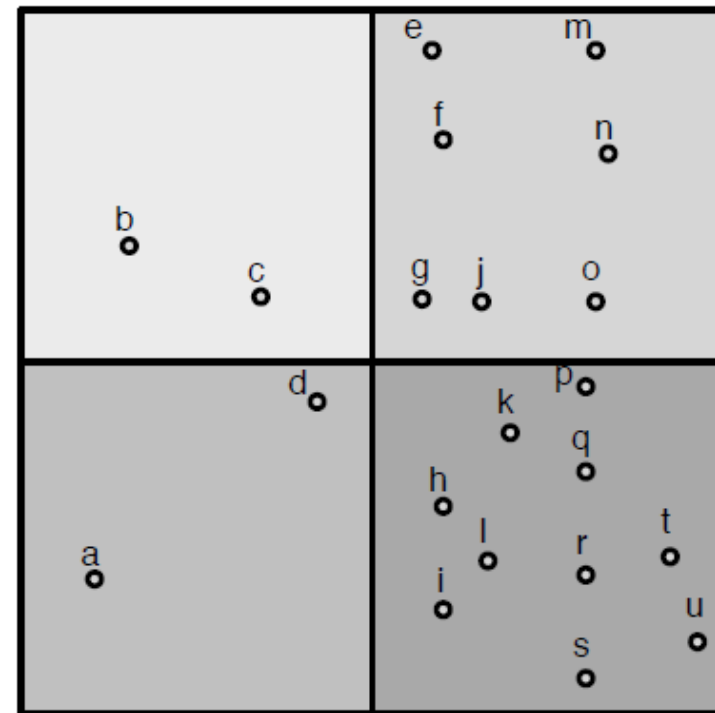


Depth = 0



# Dynamic Parallelism

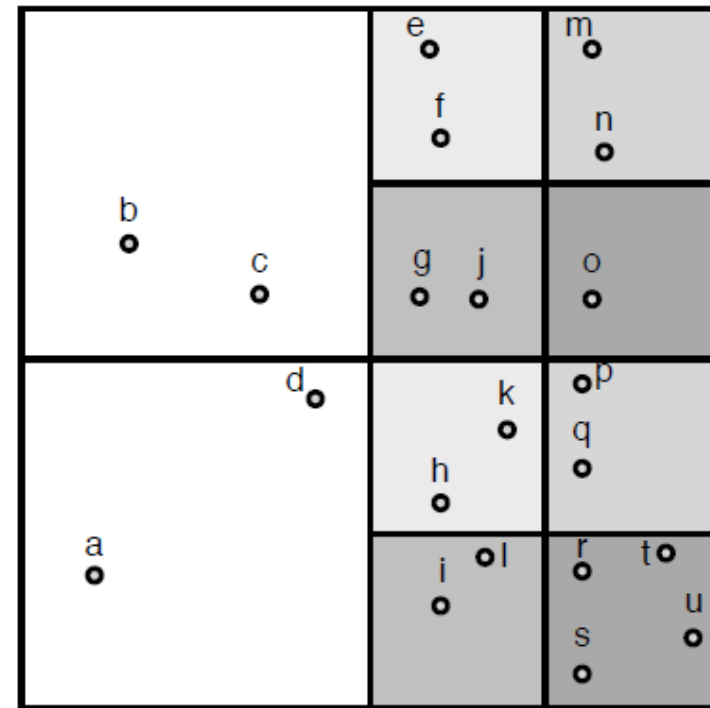
- Partitioning a 2D space by recursively dividing it into four quadrants



Depth = 1

# Dynamic Parallelism

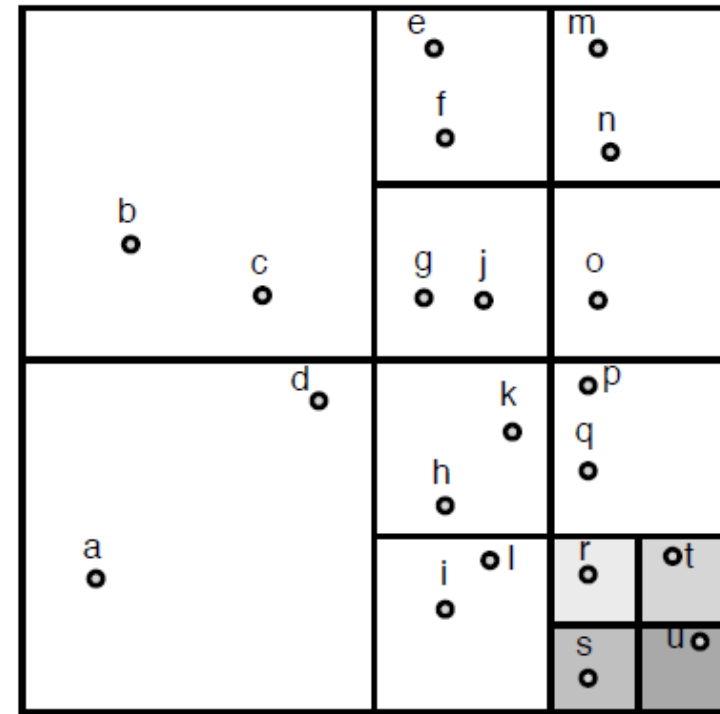
- Partitioning a 2D space by recursively dividing it into four quadrants



Depth = 2

# Dynamic Parallelism

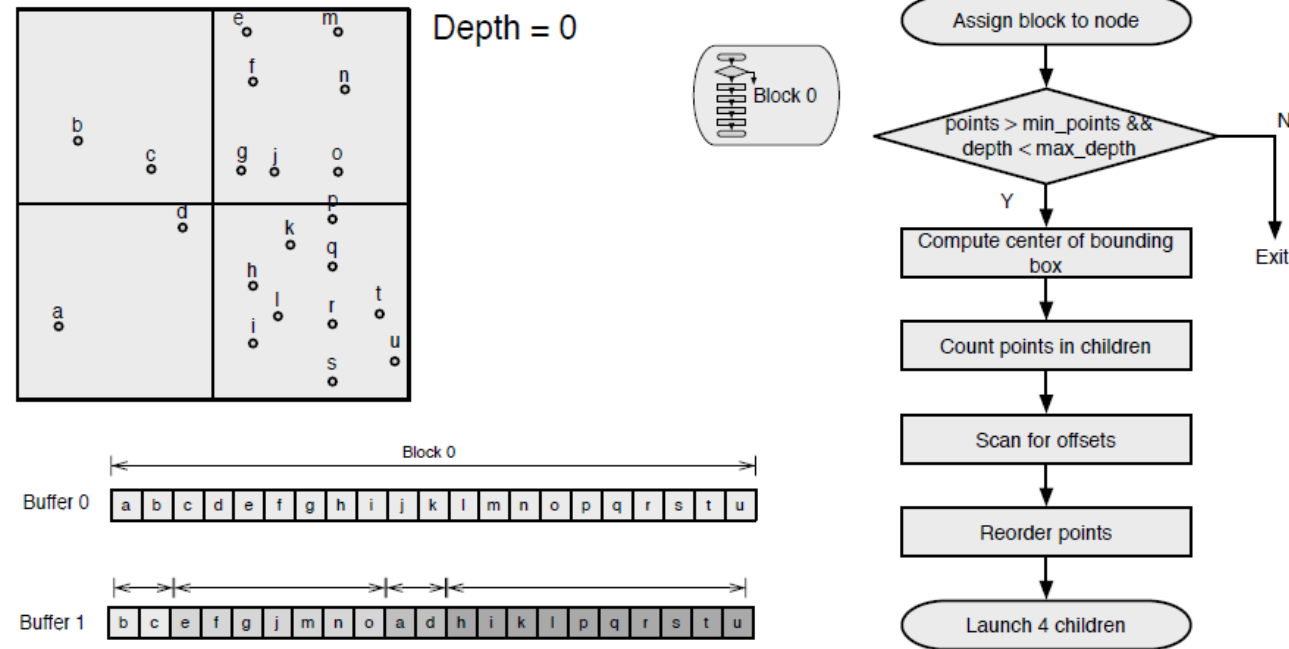
- Partitioning a 2D space by recursively dividing it into four quadrants



Depth = 3

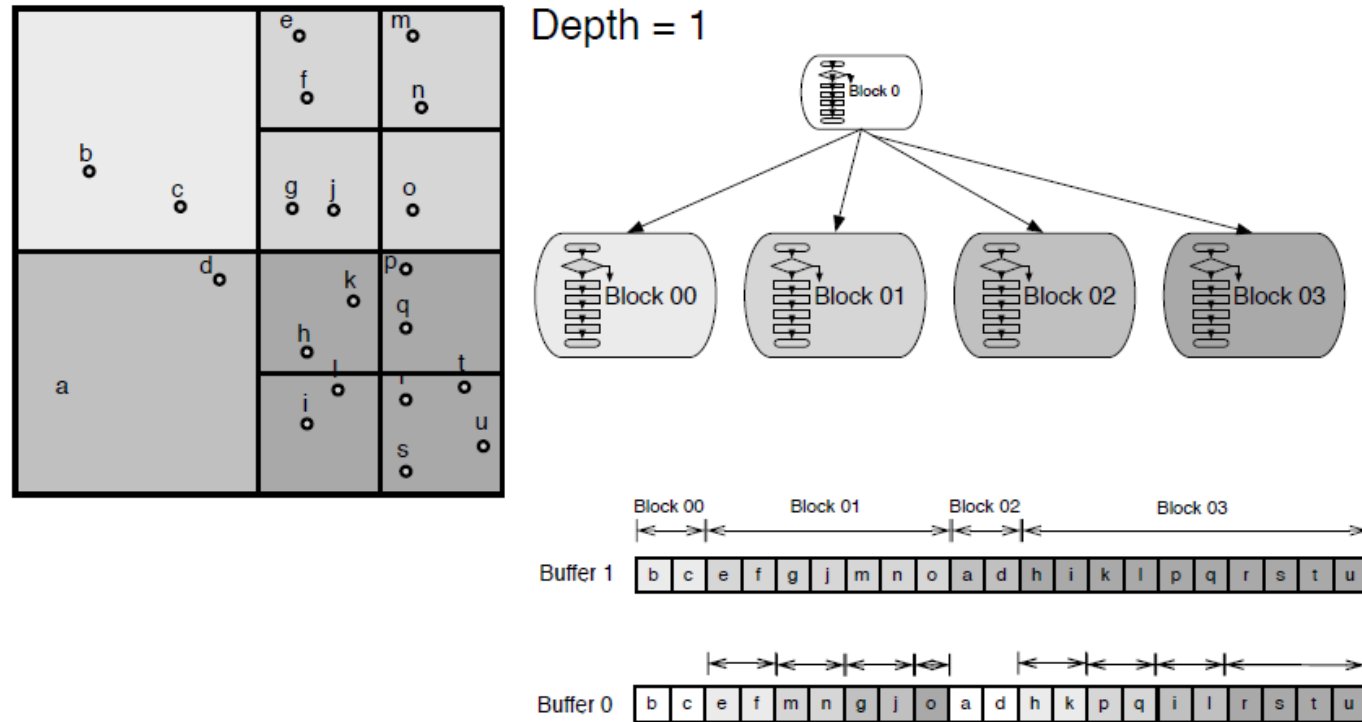
# Dynamic Parallelism

1 thread block is launched from host



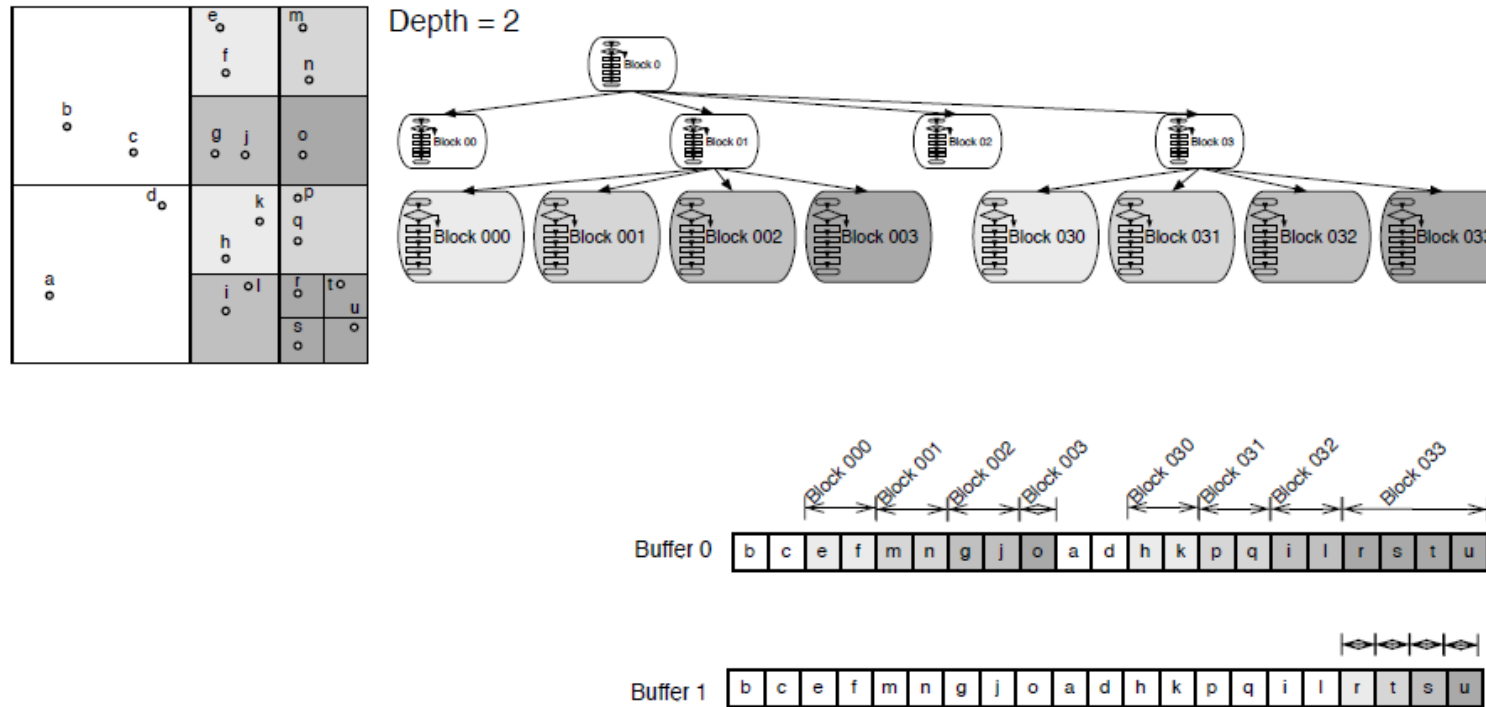
# Dynamic Parallelism

Each block launches 1 child grid of 4 blocks



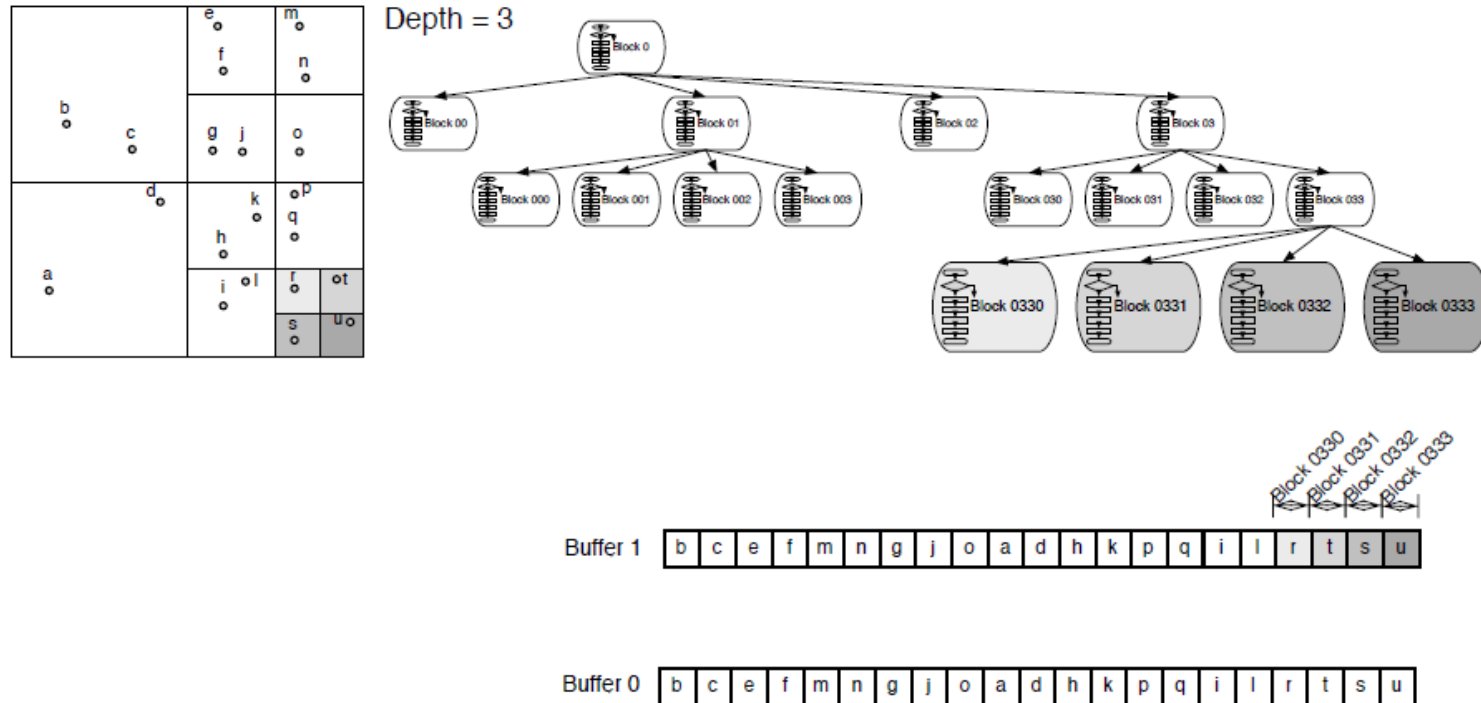
# Dynamic Parallelism

Each block launches 1 child grid of 4 blocks



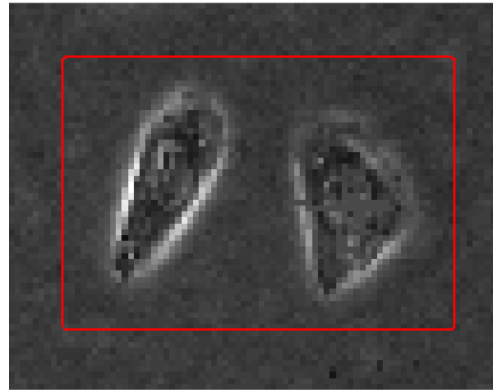
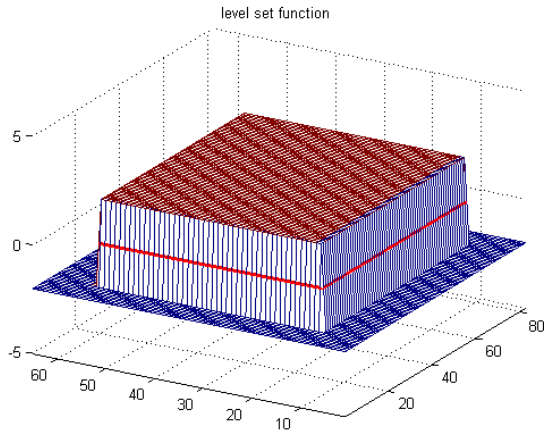
# Dynamic Parallelism

Each block launches 1 child grid of 4 blocks



# Dynamic Parallelism

- Level Set Algorithm



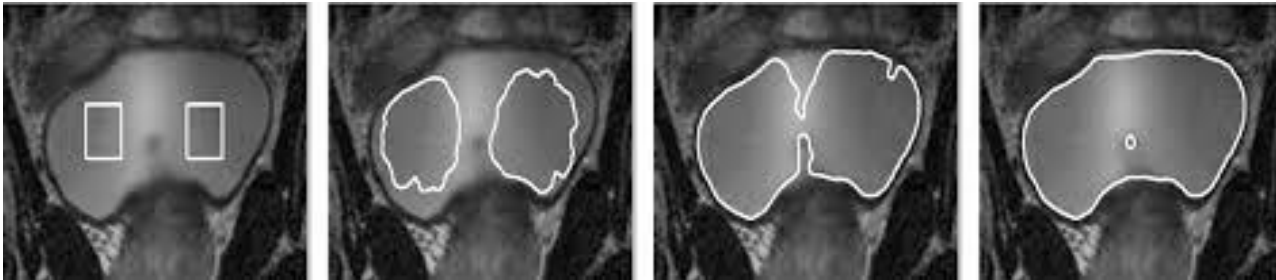
- Level Set Methods

- Used to detect the border of an object on an image
- They use partial differential equations to evolve the curve in the image
- Positive = object
- Negative = not object



# Dynamic Parallelism

- Level Set Algorithm Implementation



```
checkCuda(cudaMemcpy(gpu.intensity,
                    intensity,
                    gpu.size*sizeof(int),
                    cudaMemcpyHostToDevice));

checkCuda(cudaMemcpy(gpu.labels,
                    labels,
                    gpu.size*sizeof(int),
                    cudaMemcpyHostToDevice));

checkCuda(cudaDeviceSynchronize());

#ifdef CUDA_TIMING
    float Ktime;
    TIMER_CREATE(Ktime);
    TIMER_START(Ktime);
#endif

#ifdef VERBOSE
    printf("Running algorithm on GPU.\n");
#endif

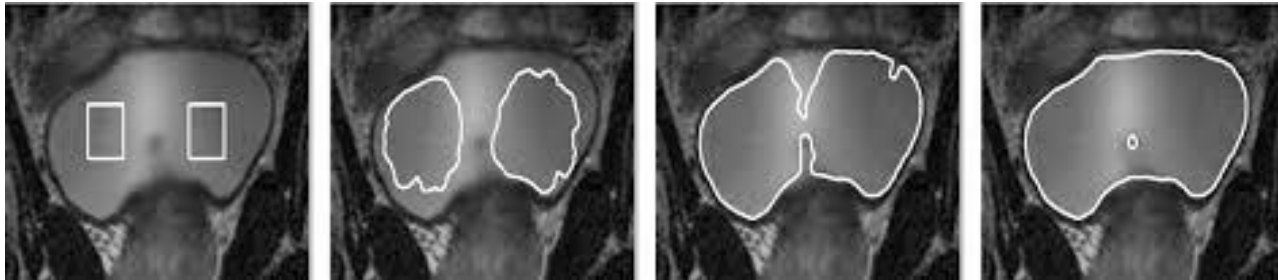
// Launch kernel to begin image segmentation
evolveContour<<<1, numLabels>>>(gpu.intensity,
                                gpu.labels,
                                gpu.phi,
                                gpu.phiOut,
                                gridSize,
                                gridSize,
                                gpu.targetLabels,
                                gpu.lowerIntensityBounds,
                                gpu.upperIntensityBounds,
                                max_iterations,
                                gpu.globalBlockIndicator,
                                gpu.globalFinishedVariable,
                                gpu.totalIterations);

checkCuda(cudaDeviceSynchronize());

#ifdef CUDA_TIMING
    TIMER_END(Ktime);
    printf("Kernel Execution Time: %f ms\n", Ktime);
#endif
```

# Dynamic Parallelism

- Level Set Algorithm Implementation



```
checkCuda(cudaMemcpy(gpu.intensity,
                    intensity,
                    gpu.size*sizeof(int),
                    cudaMemcpyHostToDevice));

checkCuda(cudaMemcpy(gpu.labels,
                    labels,
                    gpu.size*sizeof(int),
                    cudaMemcpyHostToDevice));

checkCuda(cudaDeviceSynchronize());

#ifdef CUDA_TIMING
    float Ktime;
    TIMER_CREATE(Ktime);
    TIMER_START(Ktime);
#endif

#ifdef VERBOSE
    printf("Running algorithm on GPU.\n");
#endif

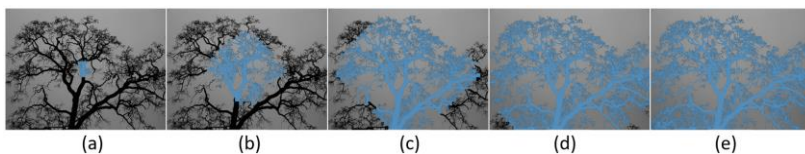
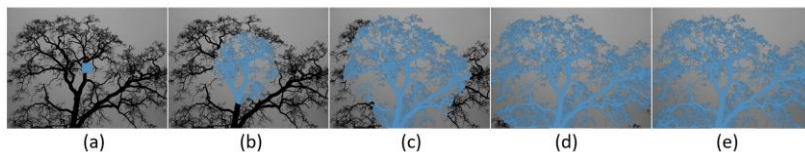
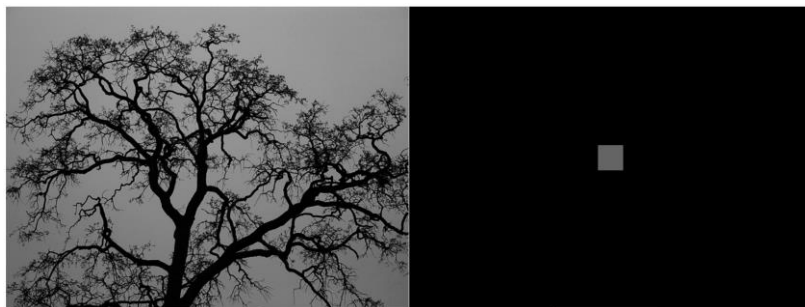
// Launch kernel to begin image segmenation
evolveContour<<<1, numLabels>>>(gpu.intensity,
                                gpu.labels,
                                gpu.phi,
                                gpu.phiOut,
                                gridSize,
                                gridSize,
                                gpu.targetLabels,
                                gpu.lowerIntensityBounds,
                                gpu.upperIntensityBounds,
                                max_iterations,
                                gpu.globalBlockIndicator,
                                gpu.globalFinishedVariable,
                                gpu.totalIterations);

checkCuda(cudaDeviceSynchronize());

#ifdef CUDA_TIMING
    TIMER_END(Ktime);
    printf("Kernel Execution Time: %f ms\n", Ktime);
#endif
```

# Dynamic Parallelism

- Level Set Algorithm Implementation



```
__global__ void evolveContour(unsigned int* intensity,
                             unsigned int* labels,
                             signed int* phi,
                             signed int* phiOut,
                             int gridSize,
                             int gridSize,
                             int* targetLabels,
                             int* lowerIntensityBounds,
                             int* upperIntensityBounds,
                             int max_iterations,
                             int* globalBlockIndicator,
                             int* globalFinishedVariable,
                             int* totalIterations ) {

    int tid = threadIdx.x;

    // Setting up streams for
    cudaStream t stream;
    cudaStreamCreateWithFlags (&stream, cudaStreamNonBlocking);

    // Total iterations
    totalIterations = &totalIterations[tid];

    // Size in ints
    int size = (gridSize*BLOCK_TILE_SIZE)*(gridSize*BLOCK_TILE_SIZE);

    // New phi pointer for each label.
    phi    = &phi[tid*size];
    phiOut = &phiOut[tid*size];

    globalBlockIndicator = &globalBlockIndicator[tid*gridSize*gridSize];

    // Global synchronization variable
    globalFinishedVariable = &globalFinishedVariable[tid];

    dim3 dimGrid(gridSize, gridSize);
    dim3 dimBlock(BLOCK_TILE_SIZE, BLOCK_TILE_SIZE);

    // Initialize phi array
    lssStep1<<<dimGrid, dimBlock, 0, stream>>>(intensity,
                                                labels,
                                                phi,
                                                targetLabels[tid],
                                                lowerIntensityBounds[tid],
                                                upperIntensityBounds[tid],
                                                globalBlockIndicator);

    int iterations = 0;
    do {
        iterations++;
        lssStep2<<<dimGrid, dimBlock, 0, stream>>>(phi,
                                                    globalBlockIndicator,
                                                    globalFinishedVariable );

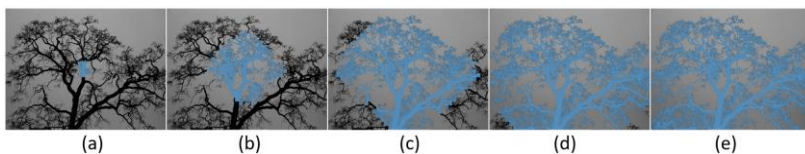
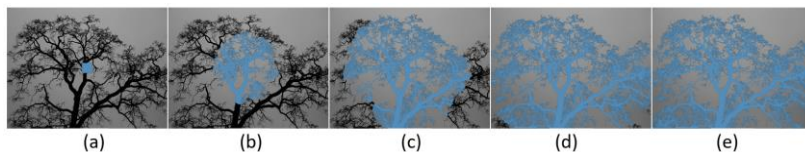
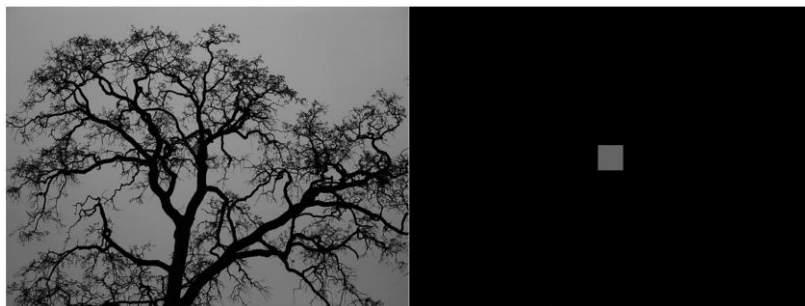
        cudaDeviceSynchronize();
    } while (atomicExch(globalFinishedVariable,0) && (iterations < max_iterations));

    lssStep3<<<dimGrid, dimBlock, 0, stream>>>(phi,
                                                phiOut);

    *totalIterations = iterations;
}
```

# Dynamic Parallelism

- Level Set Algorithm Implementation



```
__global__ void evolveContour(unsigned int* intensity,
                             unsigned int* labels,
                             signed int* phi,
                             signed int* phiOut,
                             int gridSize,
                             int gridSizeY,
                             int* targetLabels,
                             int* lowerIntensityBounds,
                             int* upperIntensityBounds,
                             int max_iterations,
                             int* globalBlockIndicator,
                             int* globalFinishedVariable,
                             int* totalIterations ) {

    int tid = threadIdx.x;

    // Setting up streams for
    cudaStream t stream;
    cudaStreamCreateWithFlags (&stream, cudaStreamNonBlocking);

    // Total iterations
    totalIterations = &totalIterations[tid];

    // Size in ints
    int size = (gridSize*BLOCK_TILE_SIZE)*(gridSizeY*BLOCK_TILE_SIZE);

    // New phi pointer for each label.
    phi = &phi[tid*size];
    phiOut = &phiOut[tid*size];

    globalBlockIndicator = &globalBlockIndicator[tid*gridSize*gridSizeY];

    // Global synchronization variable
    globalFinishedVariable = &globalFinishedVariable[tid];

    dim3 dimGrid(gridSize, gridSizeY);
    dim3 dimBlock(BLOCK_TILE_SIZE, BLOCK_TILE_SIZE);

    // Initialize phi array
    lssStep1<<<dimGrid, dimBlock, 0, stream>>>(intensity,
                                                labels,
                                                phi,
                                                targetLabels[tid],
                                                lowerIntensityBounds[tid],
                                                upperIntensityBounds[tid],
                                                globalBlockIndicator);

    int iterations = 0;
    do {
        iterations++;
        lssStep2<<<dimGrid, dimBlock, 0, stream>>>(phi,
                                                    globalBlockIndicator,
                                                    globalFinishedVariable );

        cudaDeviceSynchronize();
    } while (atomicExch(globalFinishedVariable,0) && (iterations < max_iterations));

    lssStep3<<<dimGrid, dimBlock, 0, stream>>>(phi,
                                                phiOut);

    *totalIterations = iterations;
}
```

# Another Recursive Example

- To understand recursive usage
  - Classic Fibonacci series: 0, 1, 1, 2, 3, 5...
  - CPU Recursive
  - GPU non recursive nonDP
  - GPU recursive DP

# FIBONACCI – CPU RECURSION

```
int fib(int n){  
    if (n == 0 || n == 1){  
        return n;  
    }  
    else{  
        return fib(n-1) + fib(n-2);  
    }  
}
```

# FIBONACCI – GPU PLAIN KERNEL

```
__global__ void fib_kernel_plain(int n, long int* vFib){  
    int tid = threadIdx.x + blockDim.x * blockIdx.x;  
    if (tid > n/32)  
        return;  
  
    if (n == 0 || n == 1){  
        return;  
    }  
    for(int i=tid*32 + 2; i <= n && i < tid*32 + 32; i++){  
        vFib[i] = vFib[i-1] + vFib[i-2];  
    }  
}
```

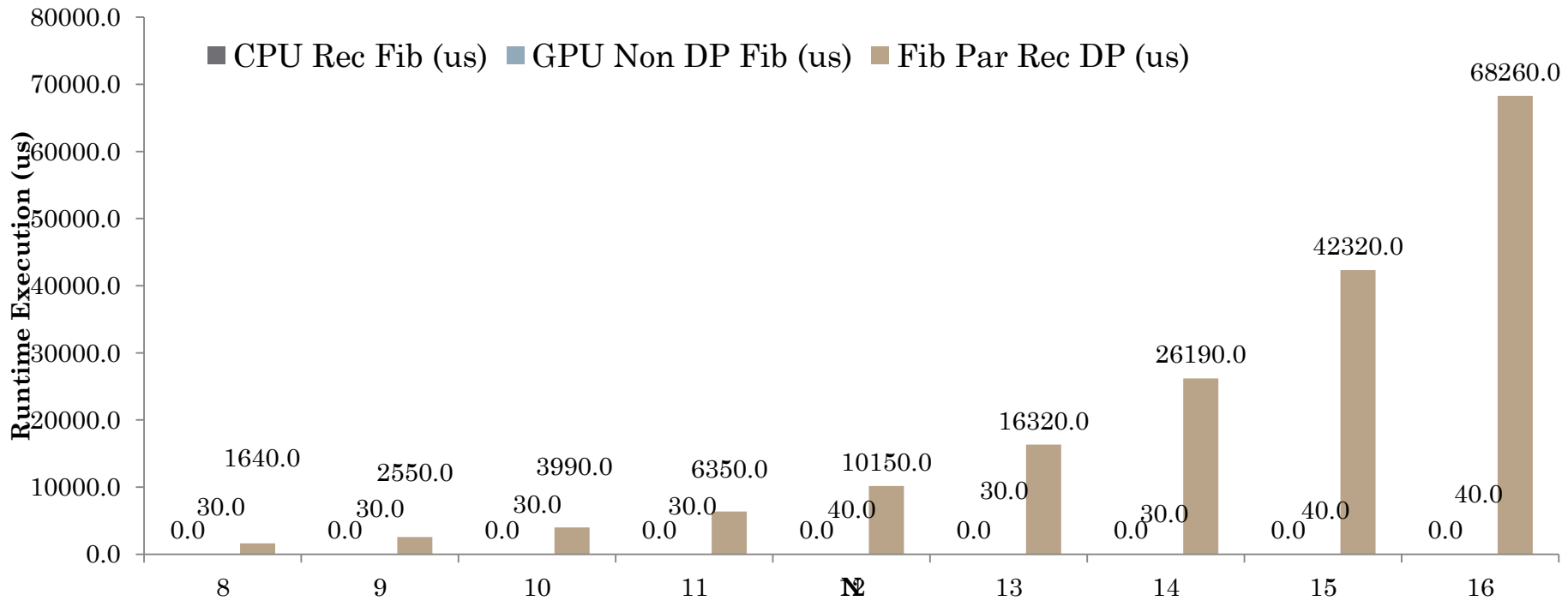
# FIBONACCI – GPU RECURSION KERNEL

```
__global__ void fib_kernel_par_rec(int n, unsigned long int* vFib){  
    if (n == 0 || n == 1)  
        return;  
  
    fib_kernel_par_rec<<<1, 1>>>(n-2, vFib);  
    fib_kernel_par_rec<<<1, 1>>>(n-1, vFib);  
    cudaDeviceSynchronize();  
    vFib[n] = vFib[n-1] + vFib[n-2];  
  
}
```



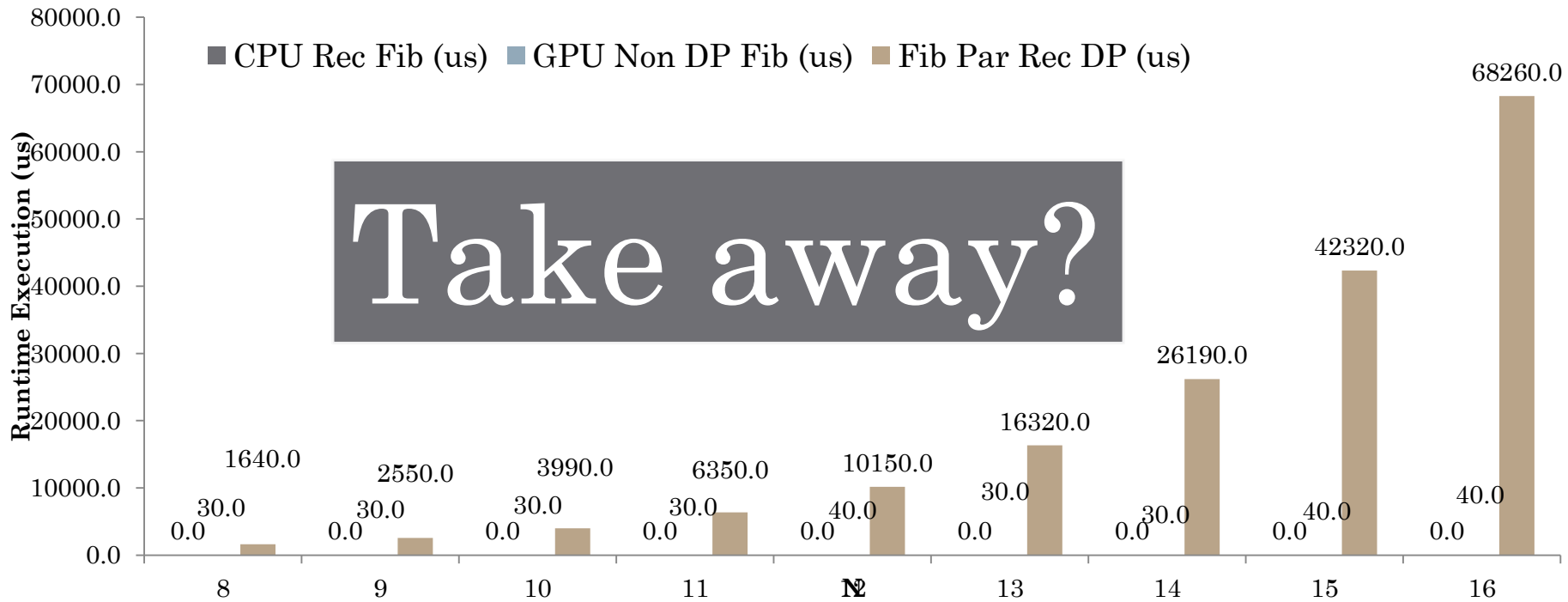
# FIBONACCI

- Performance CPU vs GPU non DP vs GPU DP



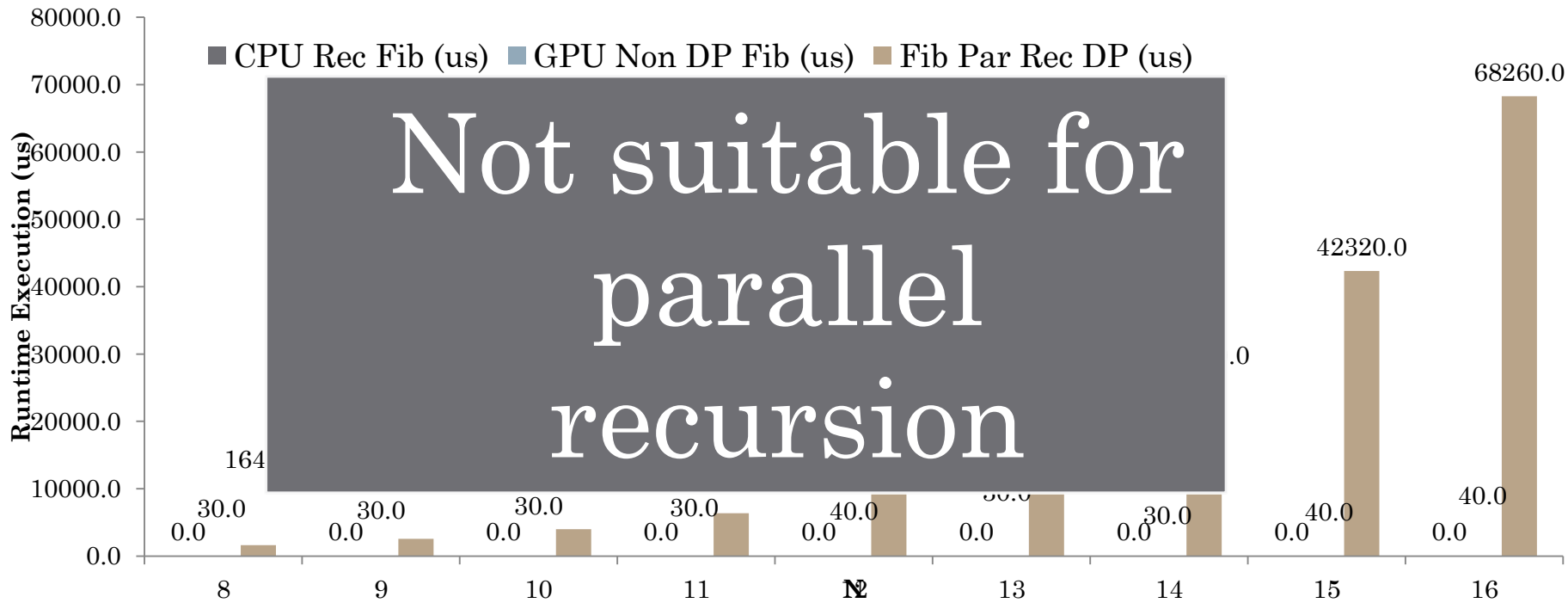
# FIBONACCI

- Performance CPU vs GPU non DP vs GPU DP



# FIBONACCI

- Performance CPU vs GPU non DP vs GPU DP



# What is and what is not dynamic parallelism

- CDP ensures better work balance, and offers advantages in terms of programmability
- However, launching grids with a very small number of threads could lead to severe underutilization of the GPU resources
- A general recommendation
  - Child grids with a large number of thread blocks,
  - Or at least thread blocks with hundreds of threads, if the number of blocks is small
- Nested parallelism for tree processing
  - Thick tree nodes (each node deploys many threads) work well
  - And/or when branch degree is large (each parent node has many children)
  - As the nesting depth is limited in hardware, only relatively shallow trees can be implemented efficiently.