GPU Programming

(in CUDA)

Summer 2020

Designed by Julian Gutierrez, Presented by Nicolas Agostini

Session 11

Outline

- Improving Host to Device Memory Transfers
  - Pinned Memory
  - 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

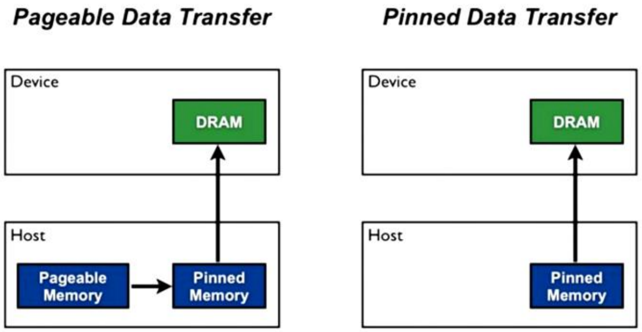
- We want to minimize the amount of time it takes to transfer data between host and device.
- How can we do this?
  - Reduce the amount of data we have to transfer! (if possible)
  - Achieve highest memory bandwidth between host and device. (Possible when using page-locked (or “pinned”) memory, we are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

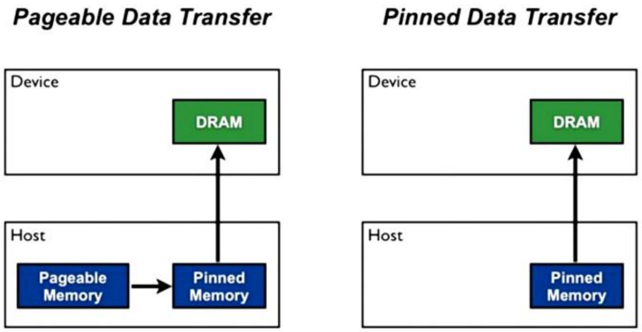
Pinned Memory (Page-Locked Data Transfers)

- Host (CPU) data allocations are pageable by default.
  - The GPU cannot access data directly from pageable host memory.
  - Due to this, the 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



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.



malloc

cudaMallocHost

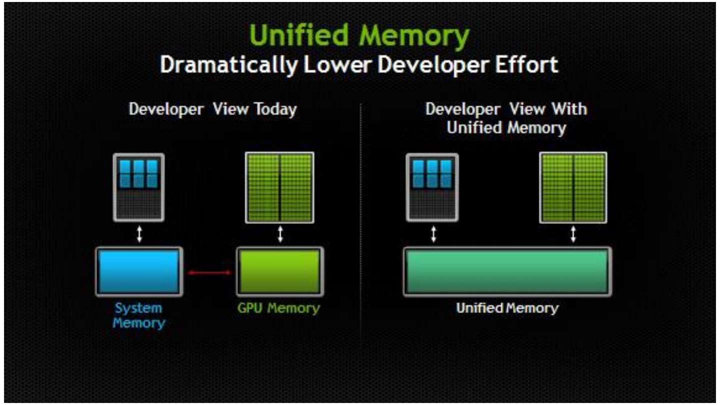
```
// allocate and initialize
h_aPageable = (float*)malloc(bytes); // host pageable
h_bPageable = (float*)malloc(bytes); // host pageable
checkCuda( cudaMallocHost((void**)&h_aPinned, bytes) ); // host pinned
checkCuda( cudaMallocHost((void**)&h_bPinned, bytes) ); // host pinned
checkCuda( cudaMalloc((void**)&d_a,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

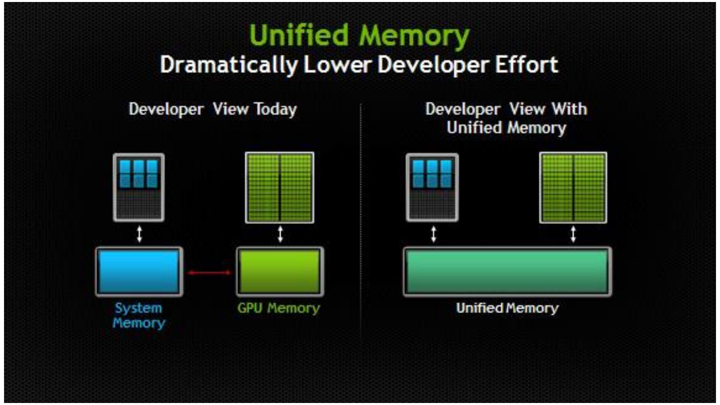
Unified Memory

- 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.



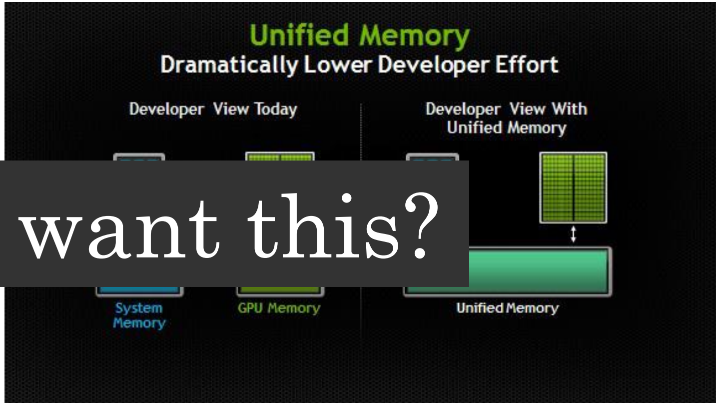
Unified Memory

- 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 `managed by runtime` 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 G
  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

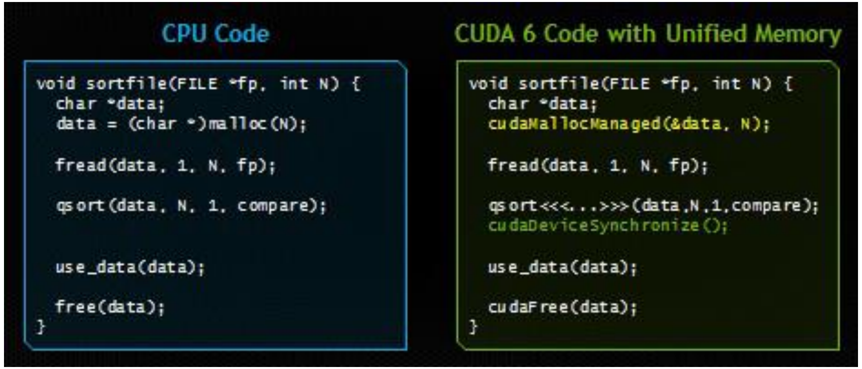
```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  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);
}
```

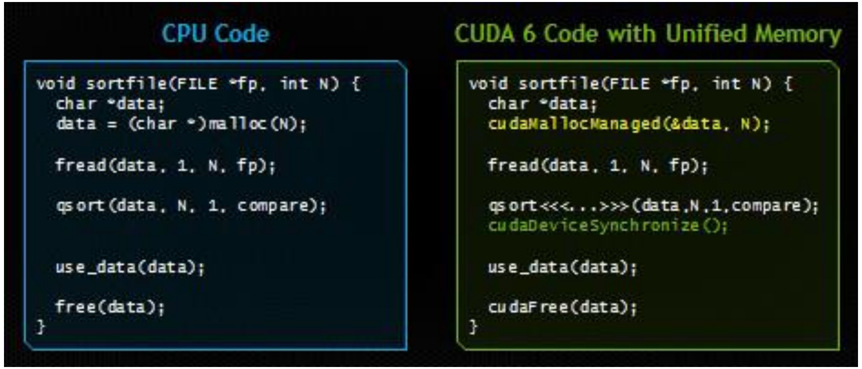
Unified Memory

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



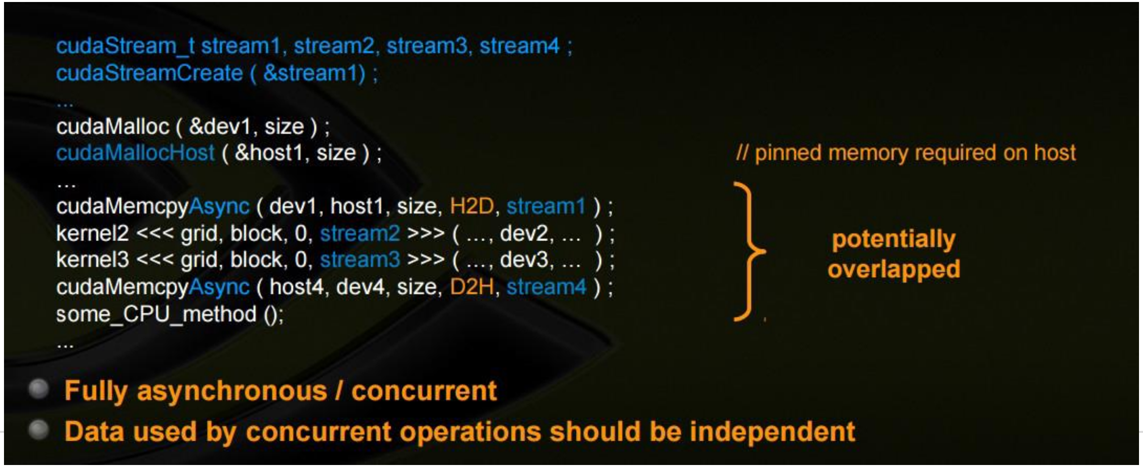
Unified Memory

- 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.



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

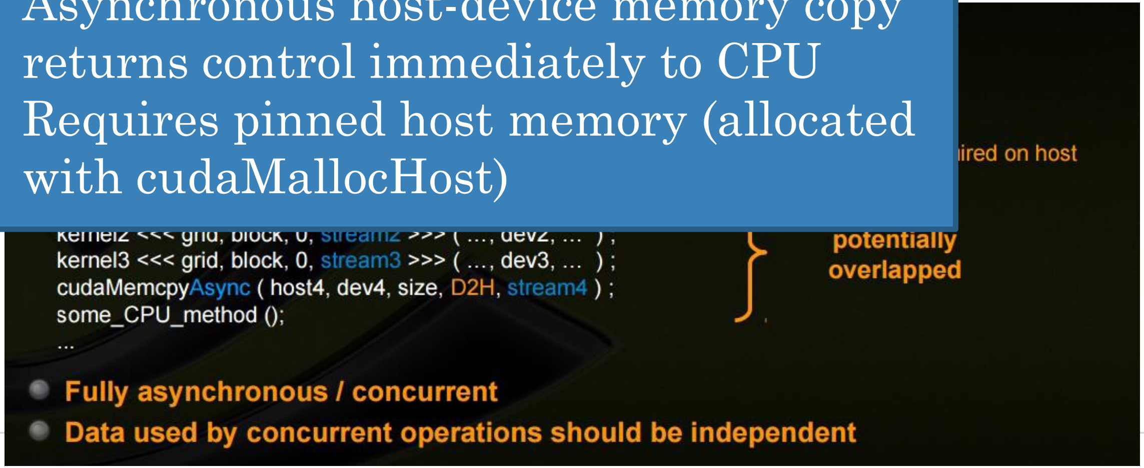


Streams

- Cuda stream:
  - A stream in CUDA is a sequence of operations that execute on the device in the order in which they
  - Operations fr
  - Stream IDs a nches

cudaMemcpyAsync

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



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

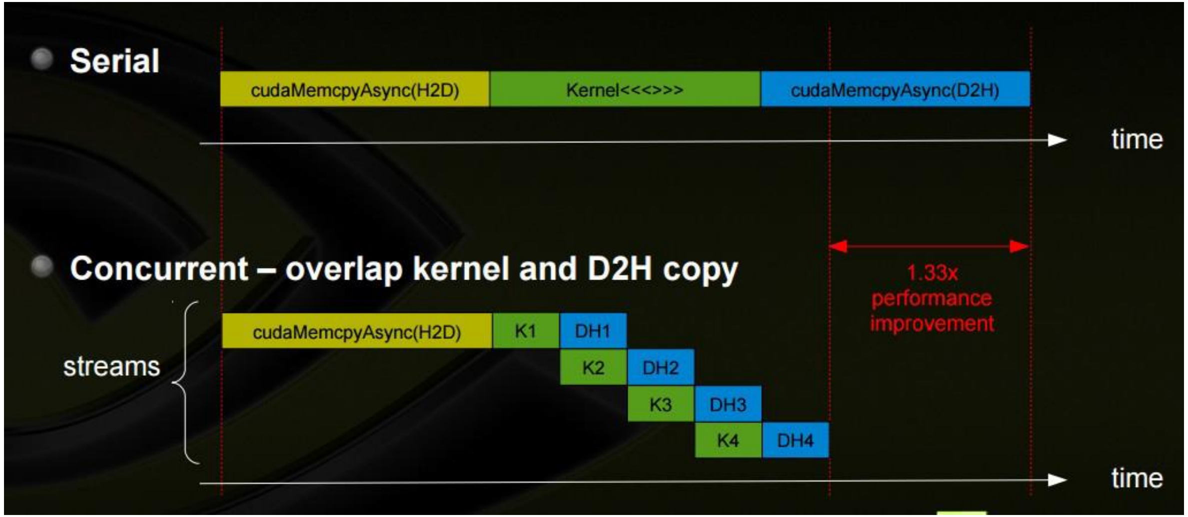
- If we have multiple streams, how many different ways exist to do a complete kernel operation (H2D/Kernel/D2H)?

Streams

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

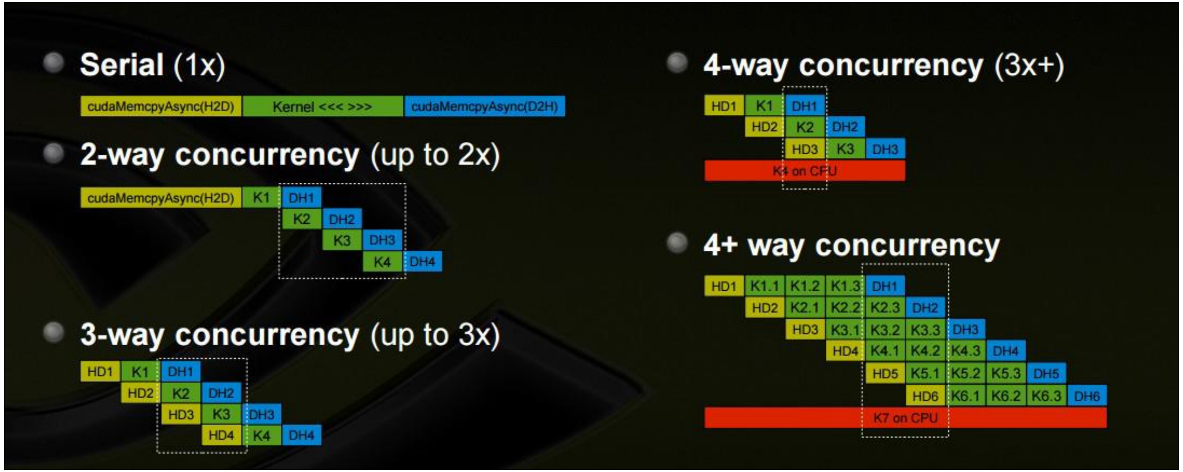
Streams

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



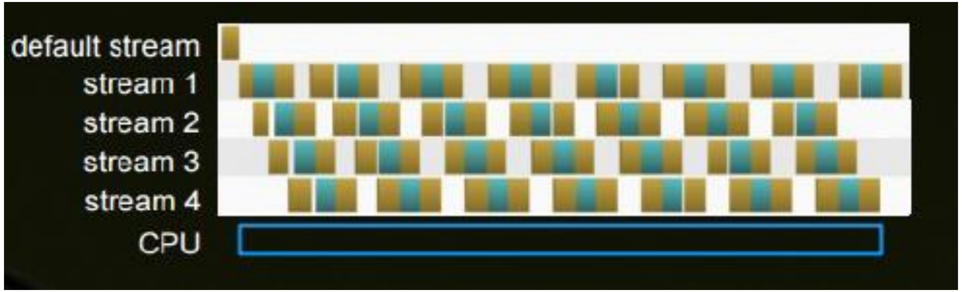
Streams

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



Streams

- Example results from a Tiled DGEMM
  - CPU
    - 43 Gflops
  - GPU
    - Serial: 126 Gflops (2.9x)
    - 2-way: 177 Gflops (4.1x)
    - 3-way: 262 Gflops (6.1x)
  - GPU + CPU
    - 4-way: 282 Gflops (6.6x)



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, cudaMemcpyHostToDev
  kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
  cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes, cudaMemcpyDeviceToH
}
```

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, cudaMemcpyHostToDev
  kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
  cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes, cudaMemcpyDeviceToH
}
```

Copy Engines

- Method 2

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

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, cudaMemcpyDeviceToHos
}
```

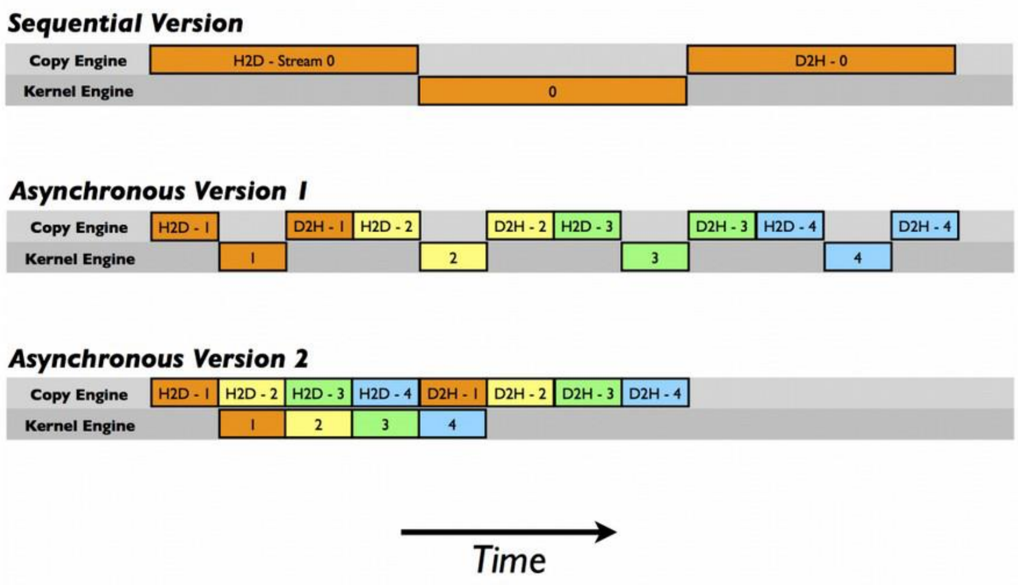
Copy Engines

- 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, cudaMemcpyHostToDevic
}

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, cudaMemcpyDeviceToHos
}
```

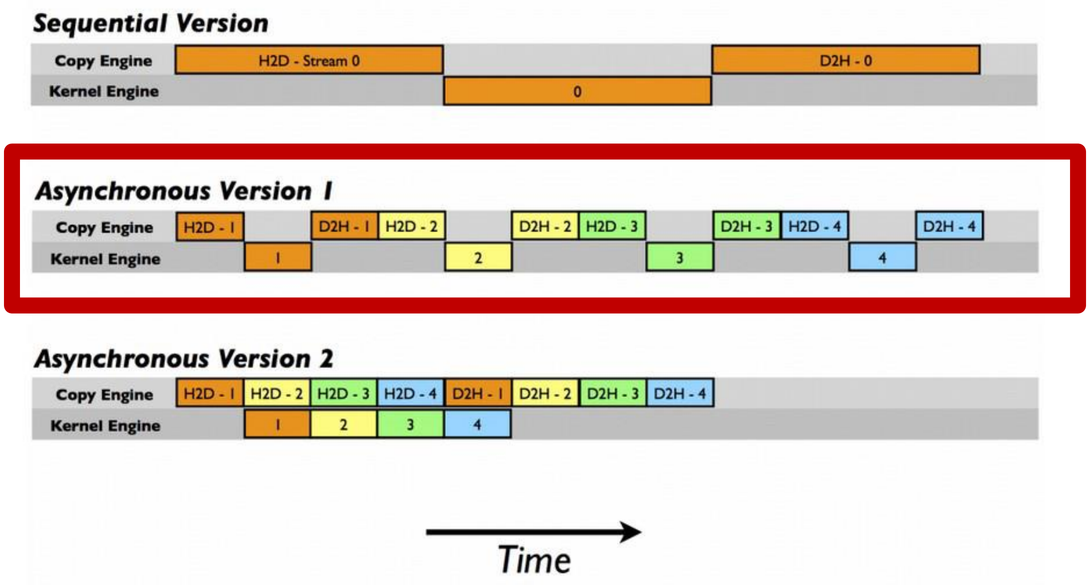
Copy Engines

- When the device has one copy engine



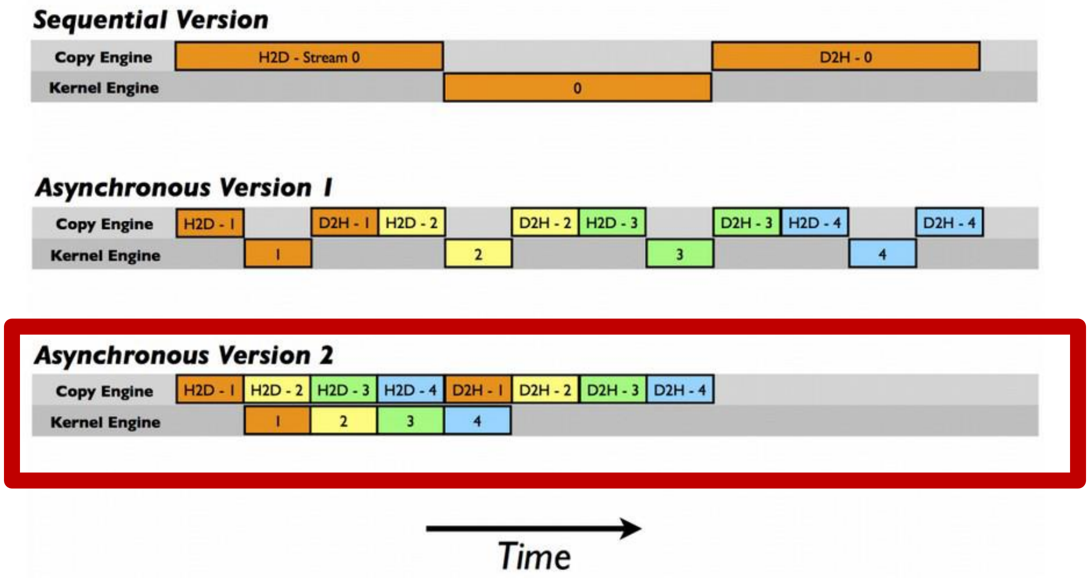
Copy Engines

- 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, …



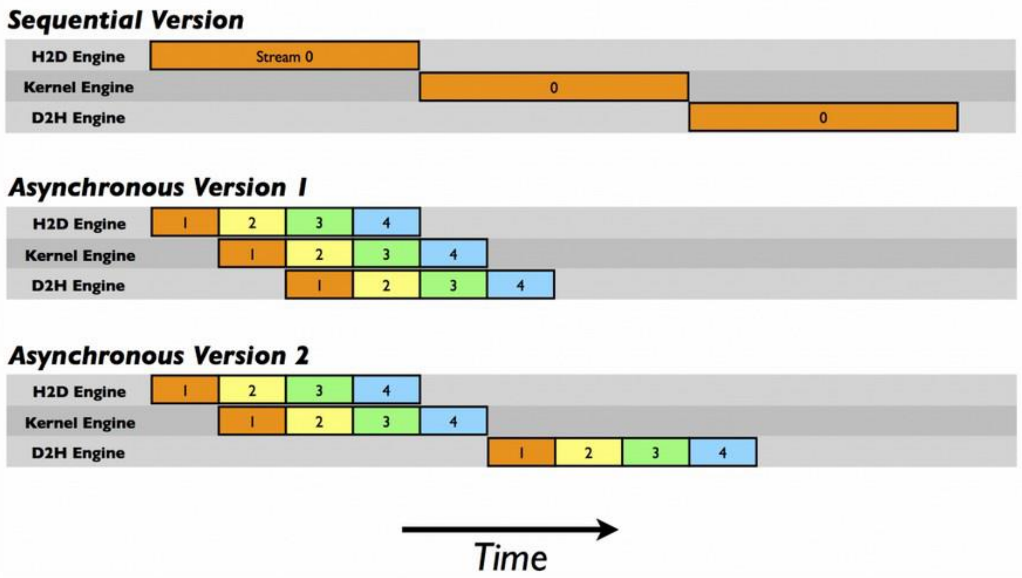
Copy Engines

- 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, ….



Copy Engines

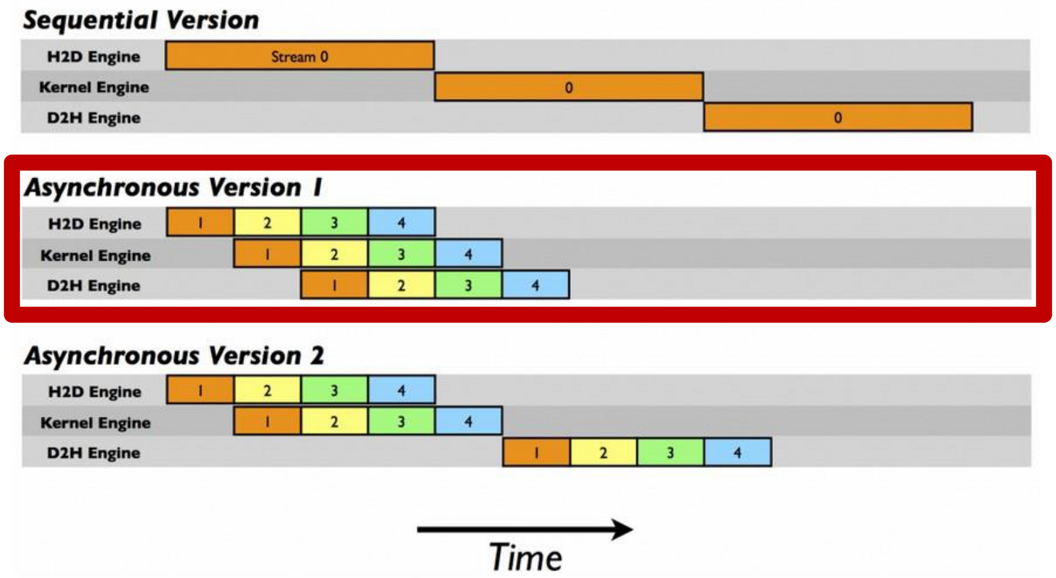
- When the device has two copy engines (with compute capability <3.5)



Copy Engines

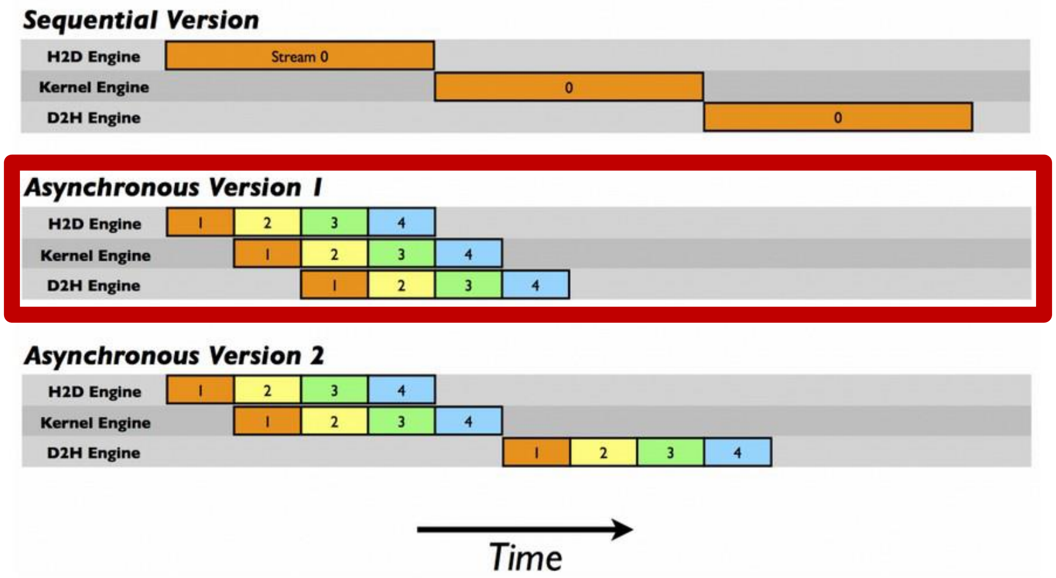
- When the device has two copy engines (with compute capability <3.5)

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



Copy Engines

- When the device has two copy engines (with compute capability <3.5)



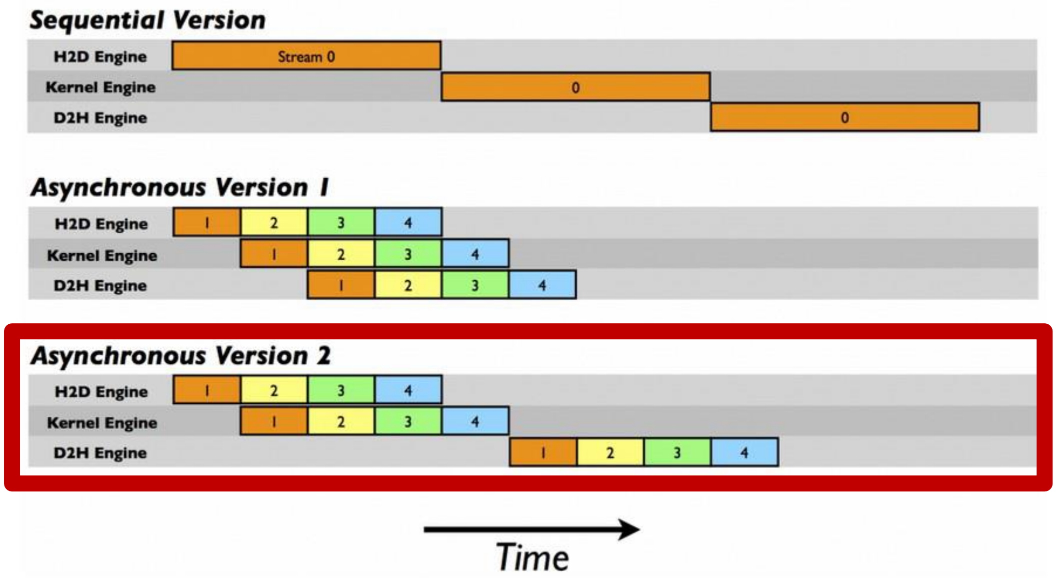
- Method 1
  - CHD1, K1, CDH1, CHD2, K2, CDH2, …
- Queues:
  - Copy engine
    - CHD1, CDH1, CHD2, CDH2, …
  - Kernel
    - K1, K2, …

Added to the other Copy Engine

Copy Engines

- When the device has two copy engines (with compute capability <3.5)

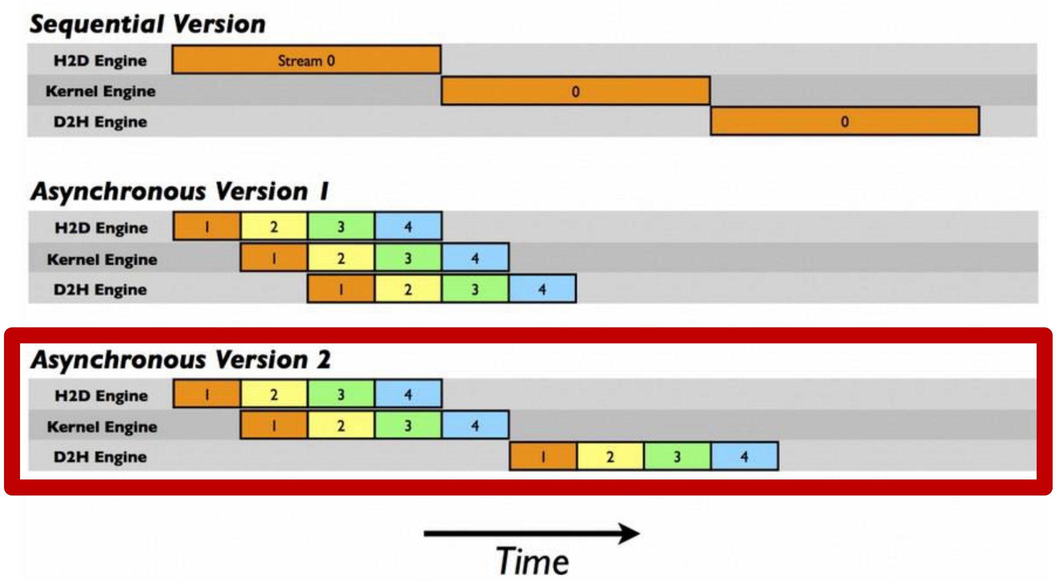
- METHOD 2
  - CHD1, CHD2, …, K1, K2, …, CDH1, CDH2, ….
- QUEUES
  - COPY ENGINE
    - CHD1, CHD2, … CDH1, CDH2, …
  - KERNEL
    - K1, K2, ….



Copy Engines

- When the device has two copy engines (with compute capability <3.5)

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

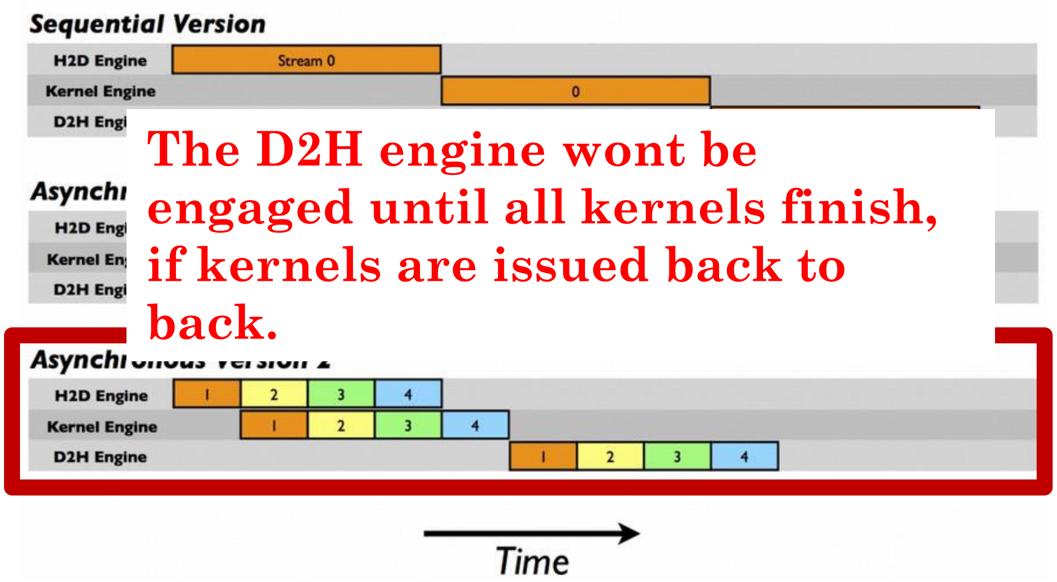


Added to the other Copy Engine

Copy Engines

- When the device has two copy engines (with compute capability <3.5)

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



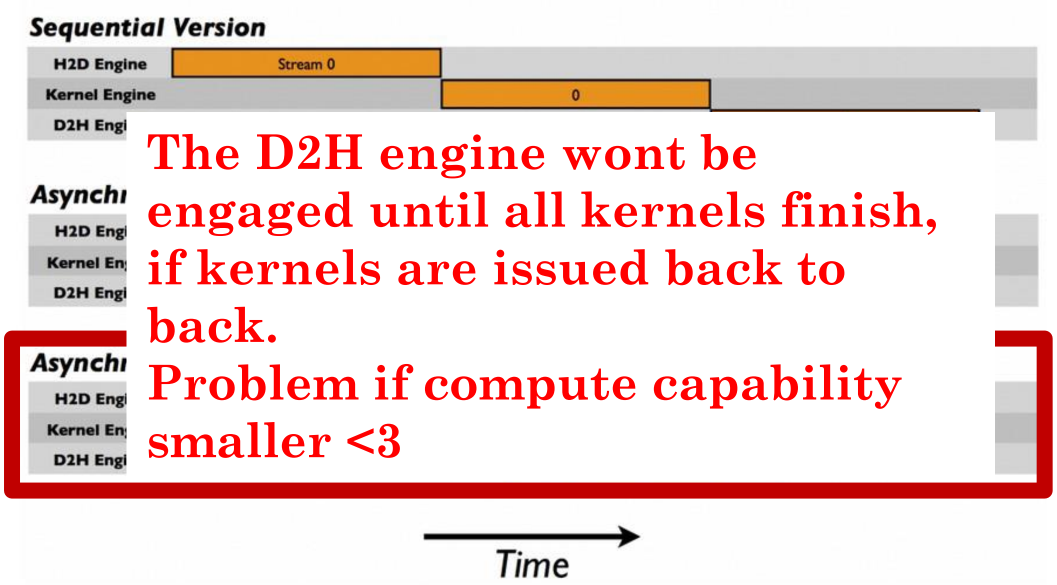
The D2H engine wont be engaged until all kernels finish, if kernels are issued back to back.

Added to the other Copy Engine

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, ….



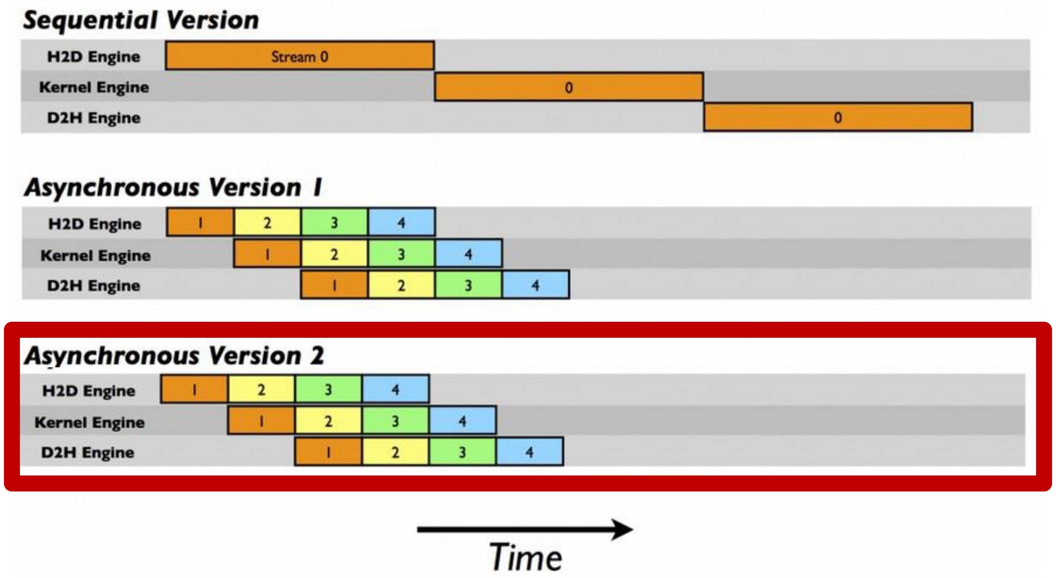
The D2H engine wont be engaged until all kernels finish, if kernels are issued back to back.
Problem if compute capability smaller <3

Added to the other Copy Engine

Copy Engines

- When the device has two copy engines (with compute capability >=3.5)

  - Method 1
    - CHD1, K1, CDH1, CHD2, K2, CDH2, …
  - Queues:
    - Copy engine
      - CHD1, CDH1, CHD2, [illegible], …
    - Kernel
      - K1, K2, …



Added to the other Copy Engine

Streams again

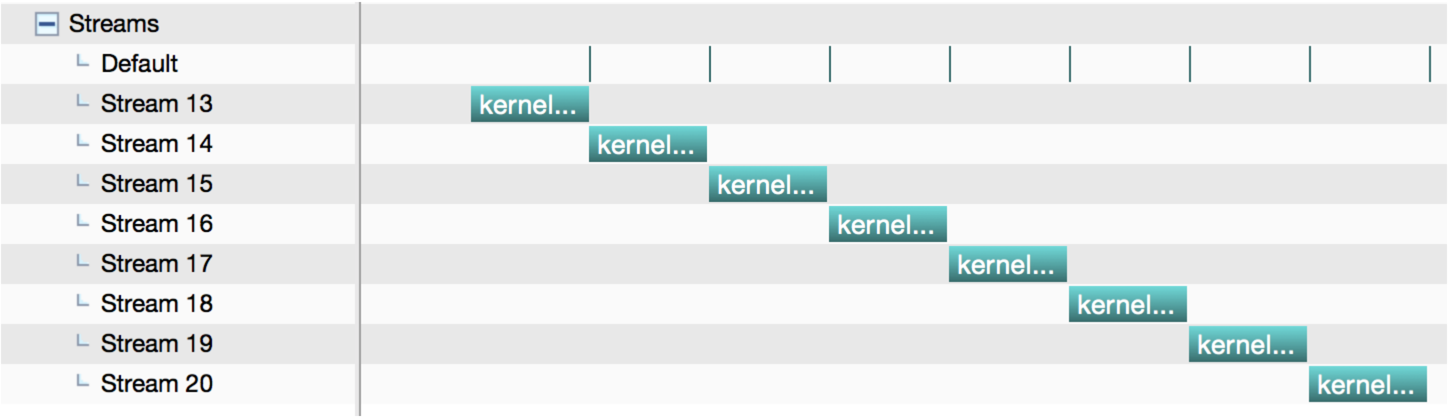
- Now that we've copied data efficiently and overlapped with execution. Another simple example:

```
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

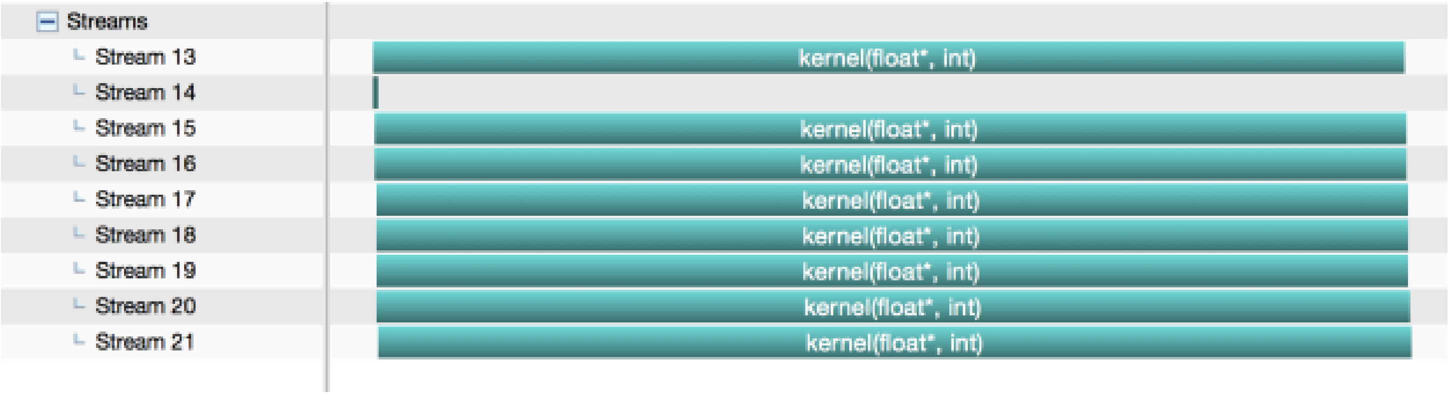
- When you execute asynchronous CUDA commands without specifying a stream:
  - runtime uses the default stream.
  - Before CUDA 7, the default stream is a special stream which implicitly synchronizes with all other streams on the device.
- CUDA 7 introduces a new option to use an independent default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

Streams again



Streams again

--default-stream per-thread

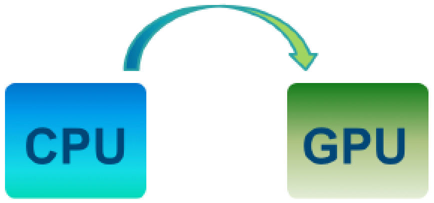


Dynamic Parallelism

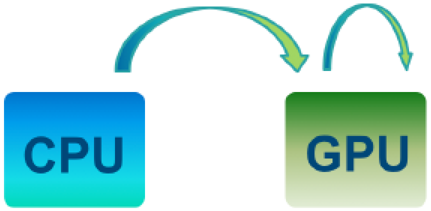
- Device-side kernel launches
  - Kepler GK110 architecture
  - Typical use cases
    - Dynamic load balancing
    - Data-dependent execution
    - Recursion
    - Library calls from kernels
  - Programmability and maintainability

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



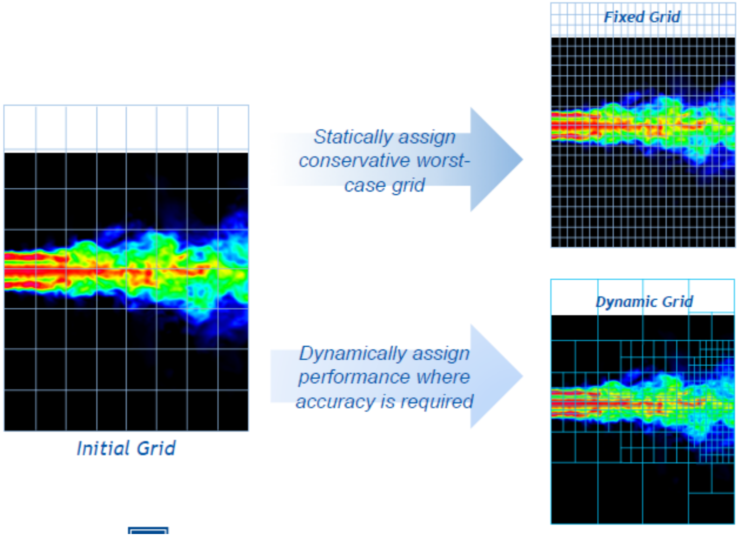
Fermi: Only CPU can generate GPU work



Kepler: GPU can generate work for itself

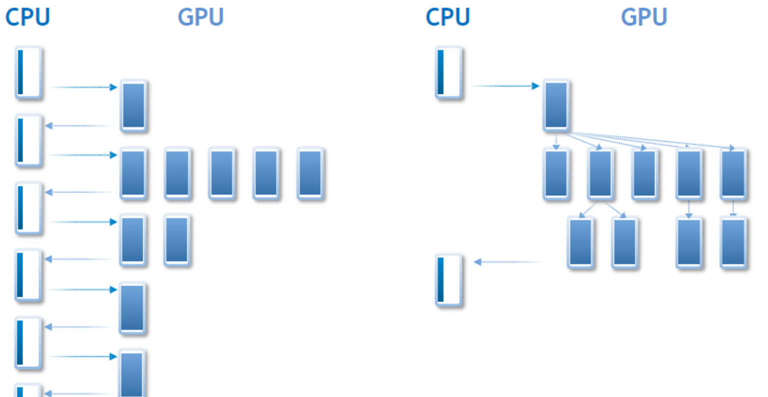
Dynamic Parallelism

- Fixed grid vs dynamic grid for a turbulence simulation model



Dynamic Parallelism

- CPU-GPU without and with dynamic parallelism



(a) Without Dynamic Parallelism

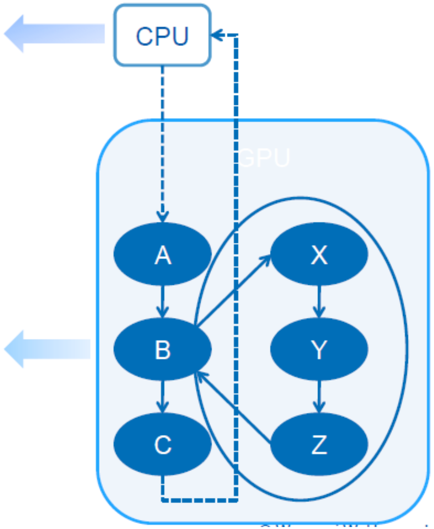
(b) With Dynamic Parallelism

Dynamic Parallelism

- 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

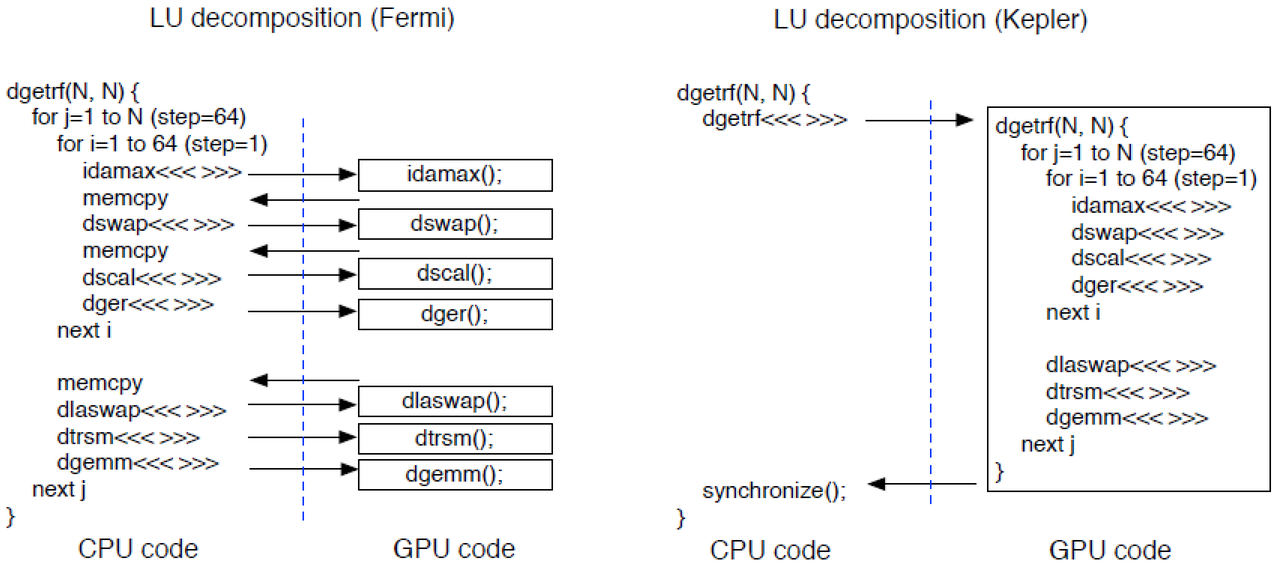
- Syntax

```
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

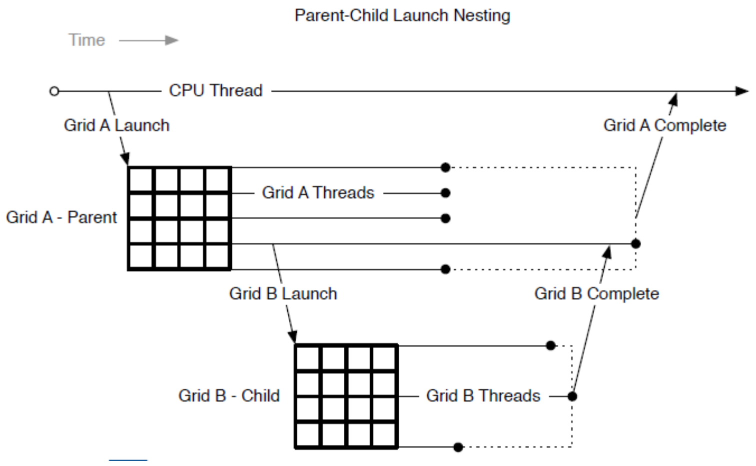
Dynamic Parallelism

- Example



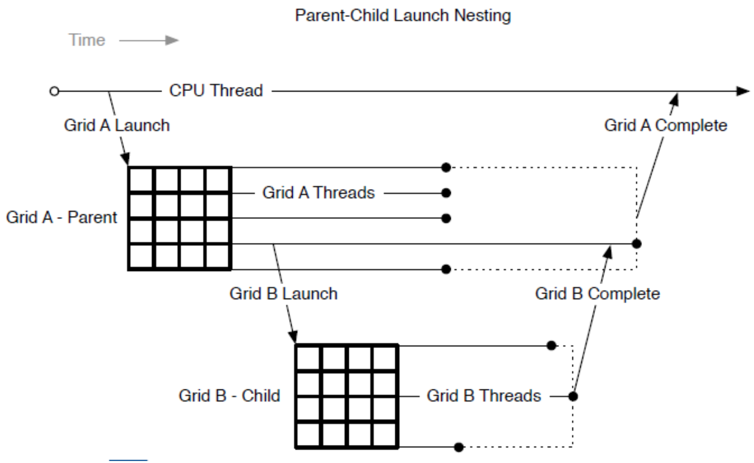
Dynamic Parallelism

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



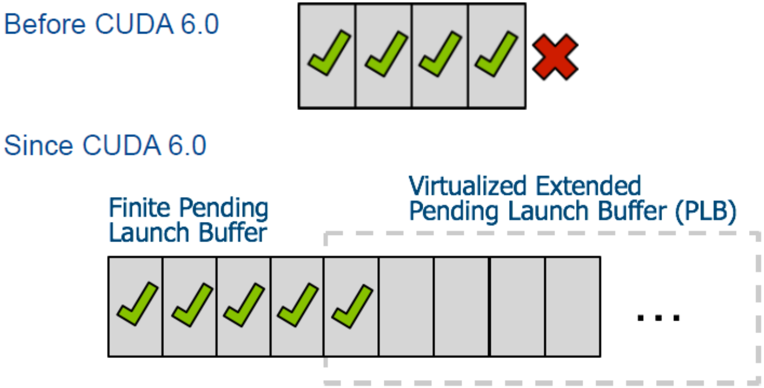
Dynamic Parallelism

- Memory Model
  - Child sees parent state at time of launch
  - Parent sees child writes after sync
  - Constants are immutable
  - Local and shared memory are private



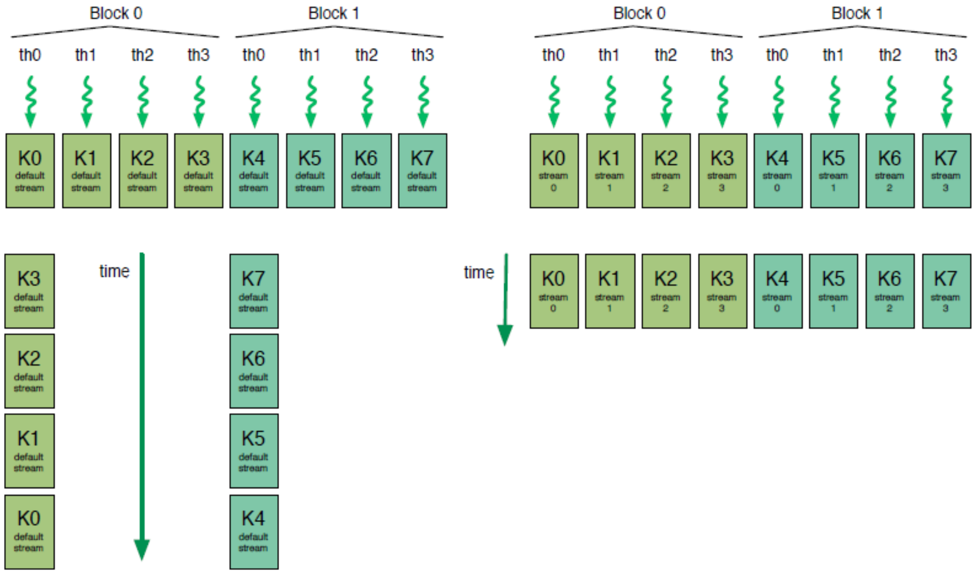
Dynamic Parallelism

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



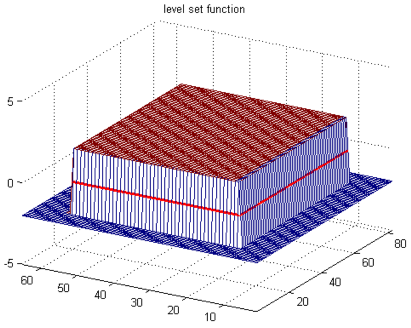
Dynamic Parallelism

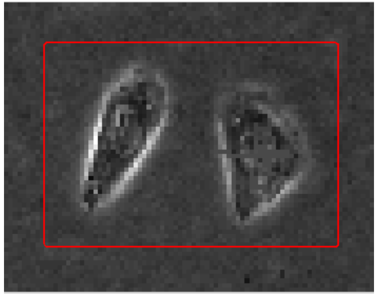
- Streams
  - To guarantee concurrency



Dynamic Parallelism

- Level Set Algorithm

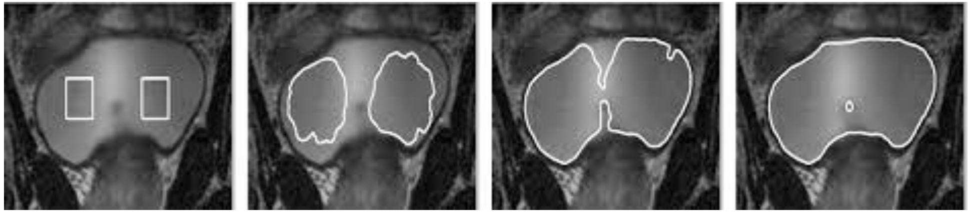




- Level Set Methods
  - Used to detect the border of an object on an image
  - They use partial differential equations to evolve the curve in the image
  - Positive = object
  - Negative = not object

Dynamic Parallelism

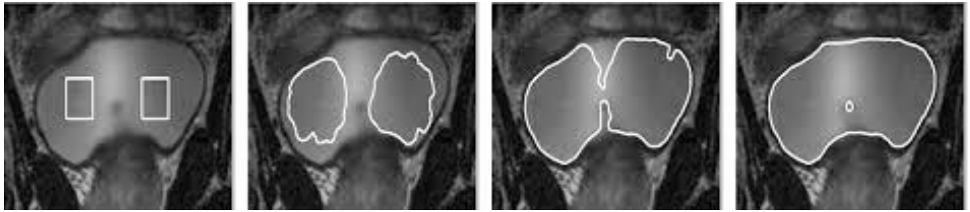
- Level Set Algorithm Implementation



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

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

checkCuda(cudaDeviceSynchronize());

#if defined(CUDA_TIMING)
        float Ktime;
        TIMER_CREATE(Ktime);
        TIMER_START(Ktime);
#endif

#if defined(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,
                                gridXSize,
                                gridYSize,
                                gpu.targetLabels,
                                gpu.lowerIntensityBounds,
                                gpu.upperIntensityBounds,
                                max_iterations,
                                gpu.globalBlockIndicator,
                                gpu.globalFinishedVariable,
                                gpu.totalIterations);

checkCuda(cudaDeviceSynchronize());

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

Dynamic Parallelism

- Level Set Algorithm Implementation

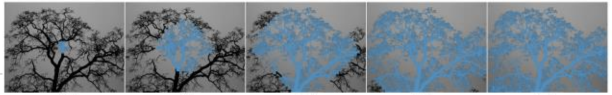


```
checkCuda(cudaMemcpy(gpu.intensity,
                    intensity,
                    gpu.size*sizeof(int),
                    cudaMemcpyHostToDevice));

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

checkCuda(cudaDeviceSynchronize());

#if defined(CUDA_TIMING)
        float Ktime;
        TIMER_CREATE(Ktime);
        TIMER_START(Ktime);
#endif

#if defined(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,
                                gridXSize,
                                gridYSize,
                                gpu.targetLabels,
                                gpu.lowerIntensityBounds,
                                gpu.upperIntensityBounds,
                                max_iterations,
                                gpu.globalBlockIndicator,
                                gpu.globalFinishedVariable,
                                gpu.totalIterations);
checkCuda(cudaDeviceSynchronize());

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

Dynamic Parallelism

- Level Set Algorithm Implementation

(a) (b) (c) (d) (e)



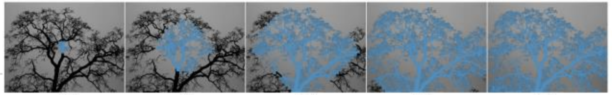
(a) (b) (c) (d) (e)

```
__global__ void evolveContour(unsigned int* intensity,
                              unsigned int* labels,
                              signed int* phi,
                              signed int* phiOut,
                              int gridXSize,
                              int gridYSize,
                              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 = (gridXSize*BLOCK_TILE_SIZE)*(gridYSize*BLOCK_TILE_SIZE);

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

    globalBlockIndicator = &globalBlockIndicator[tid*gridXSize*gridYSize];

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

    dim3 dimGrid(gridXSize, gridYSize);
    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

(a) (b) (c) (d) (e)



(a) (b) (c) (d) (e)

```
__global__ void evolveContour(unsigned int* intensity,
                              unsigned int* labels,
                              signed int* phi,
                              signed int* phiOut,
                              int gridXSize,
                              int gridYSize,
                              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 = (gridXSize*BLOCK_TILE_SIZE)*(gridYSize*BLOCK_TILE_SIZE);

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

    globalBlockIndicator = &globalBlockIndicator[tid*gridXSize*gridYSize];

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

    dim3 dimGrid(gridXSize, gridYSize);
    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 Basic Kernel

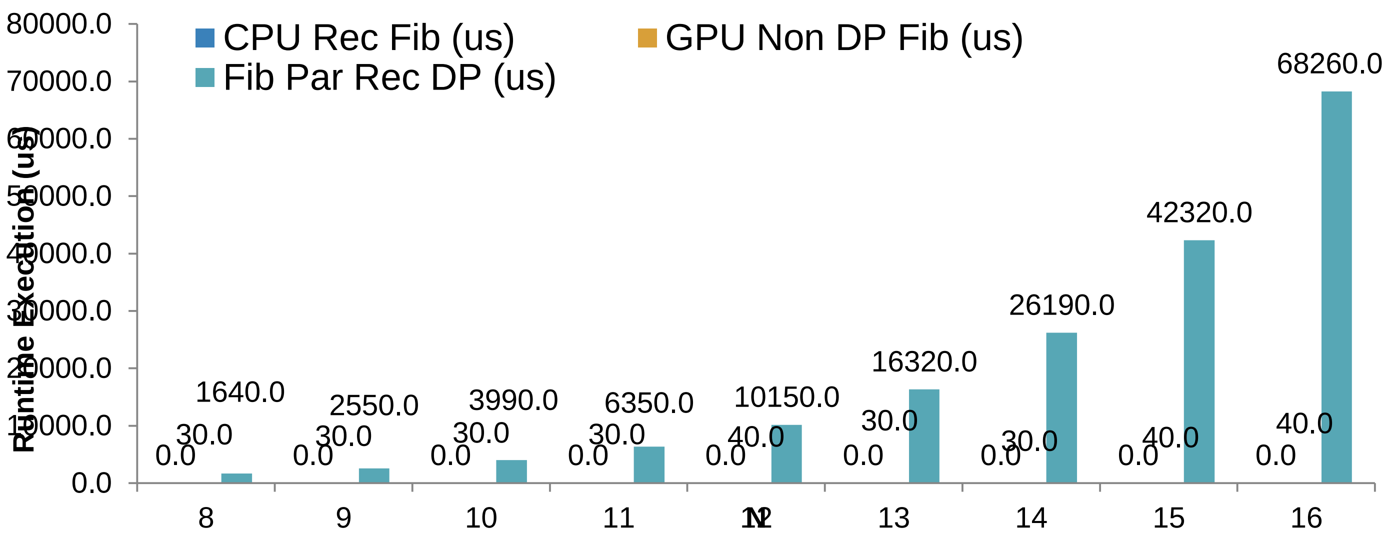
```
__global__ void fib_kernel_plain(int n, long int* vFib){
    int tid = threadIdx.x + blockDim.x * blockIdx.x;
     if (tid > n/32)
                 return;

        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];

}
```

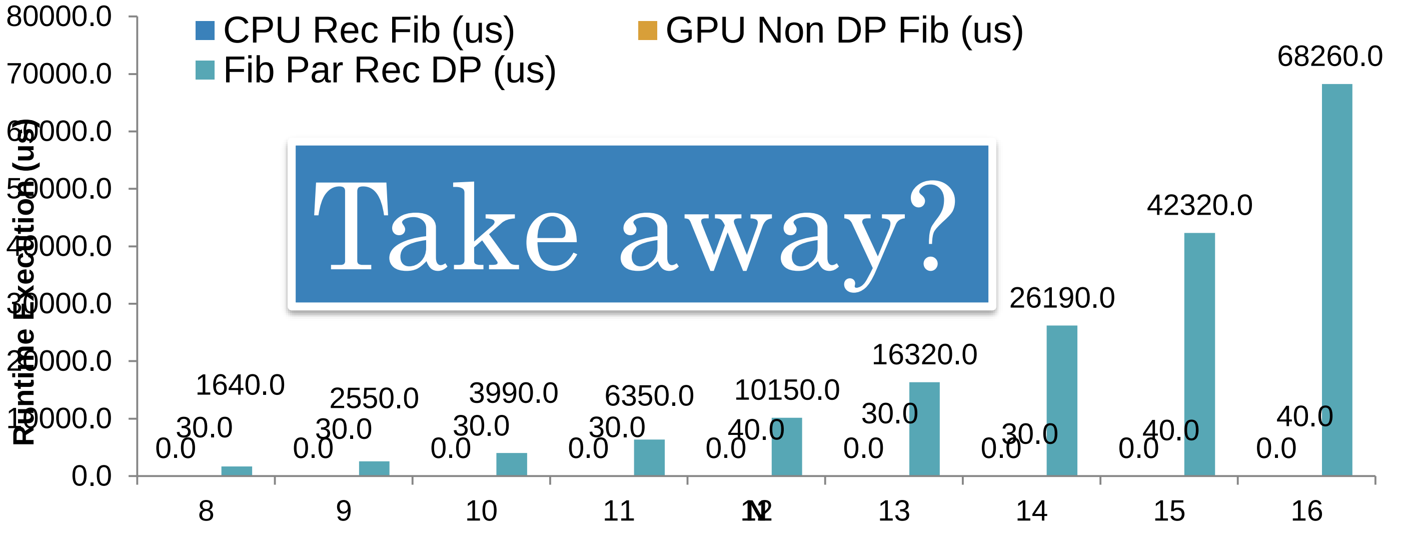
Results

- Performance CPU vs GPU non DP vs GPU DP



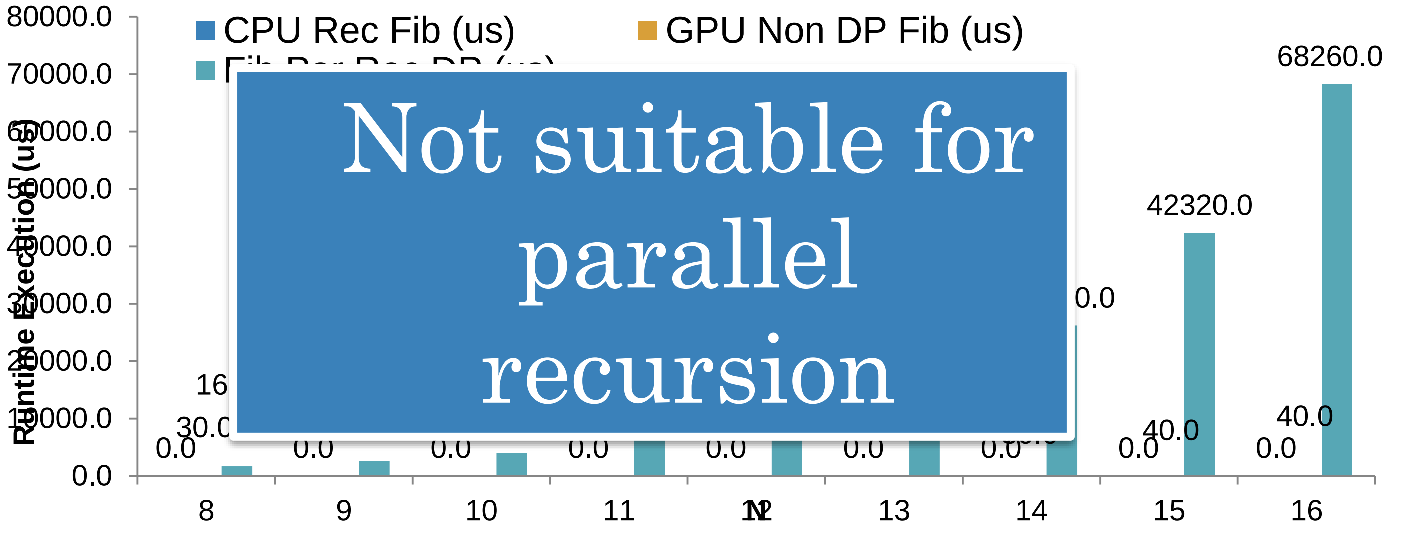
Results

- Performance CPU vs GPU non DP vs GPU DP



Results

- 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.