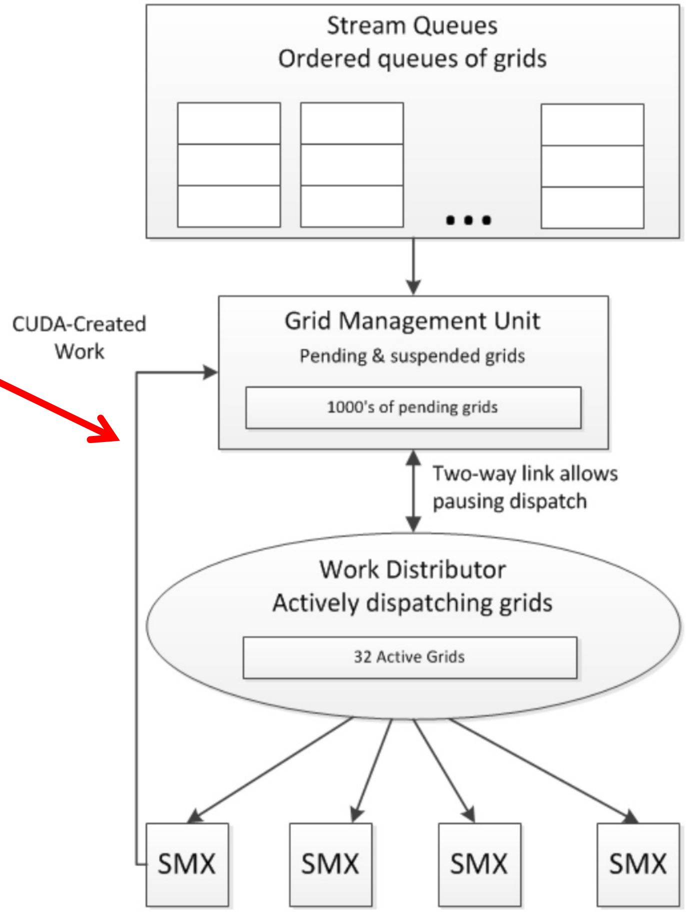
GPUs and Heterogeneous Systems
(programming models and architectures)

CUDA dynamic parallelism

Dynamic parallelism

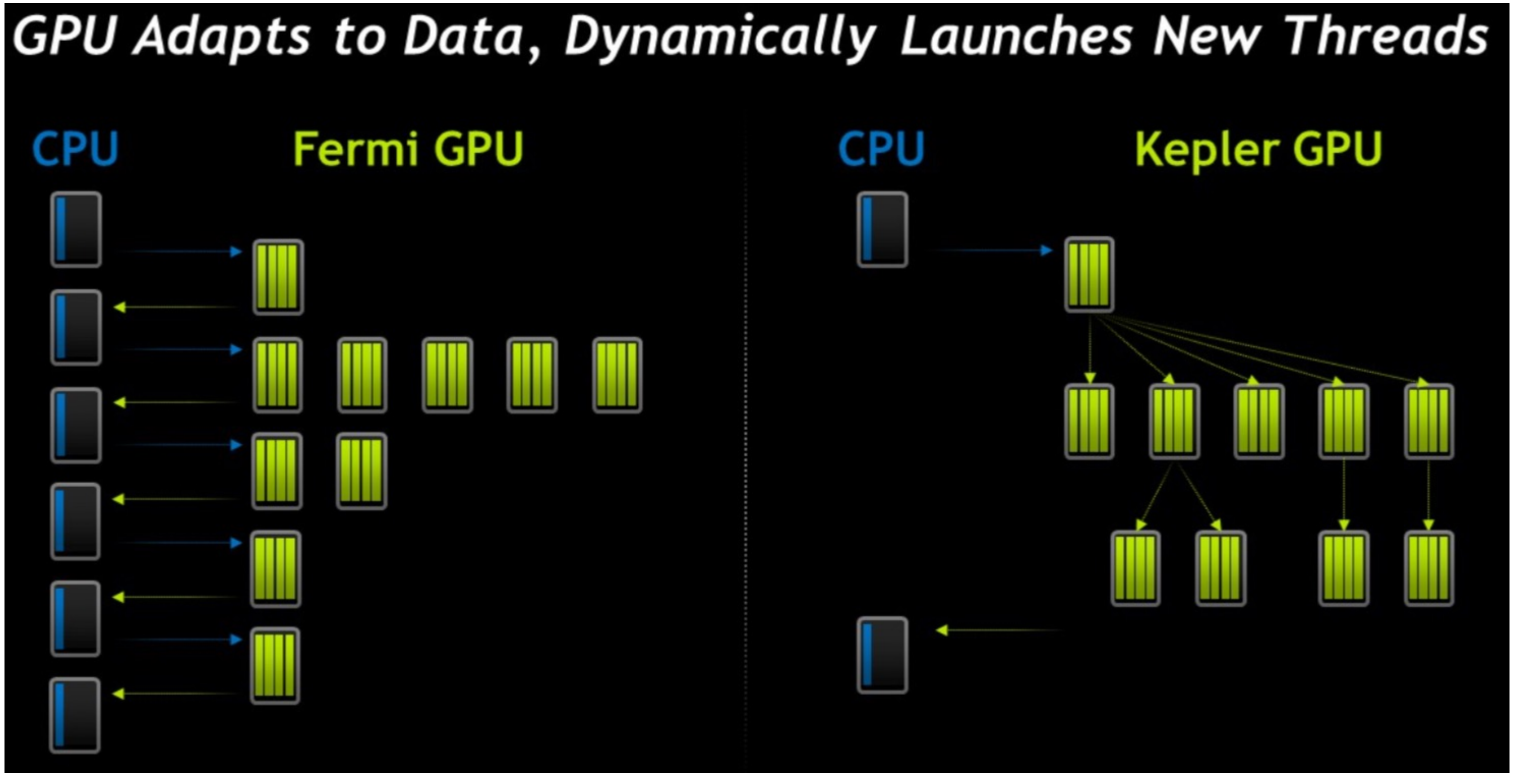
- NVIDIA GPUs (>= Kepler) provide mechanisms for dynamic parallelism
  - A thread can launch another kernel

- Advantages:
  - More natural implementation of recursive algorithms
  - Dynamically adapting to problems with unbalanced or data-driven load
    - Graphs, spare matrices, …
  - Reducing execution transfer control between host and device

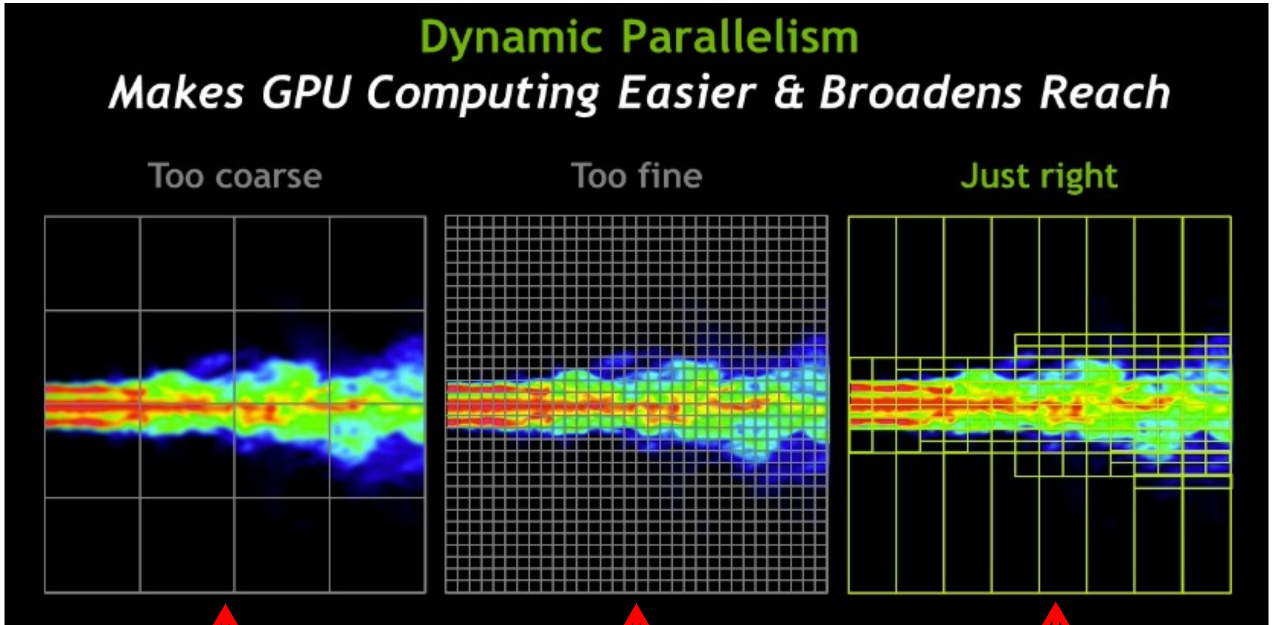


Dynamic parallelism

- The kernel running on the device can spawn child kernels



A motivating example: turbulence simulation



Too much (sequential) work to the single thread/block

Too many threads/blocks doing nothing

Adjust the work to be assigned to each thread/block

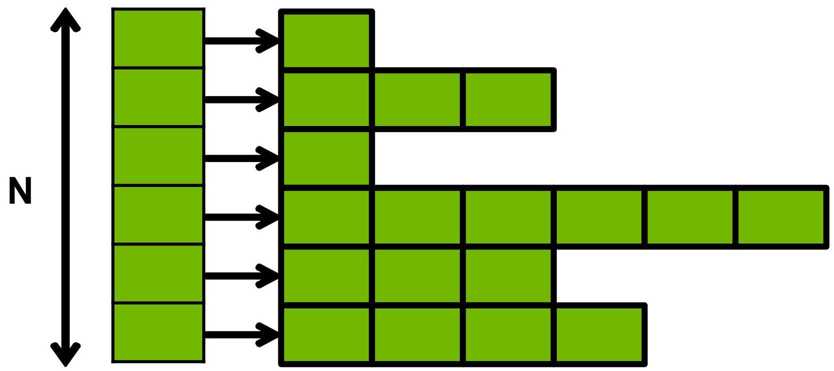
- Dynamic approach:
  - Kernel starts with the coarse-grain grid configuration
  - Then spawns multiple child grids

A motivating example: from the code point of view

- Coarse-grain solution:

All data to be processed are stored in a linearized array

```
__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]);
}
```

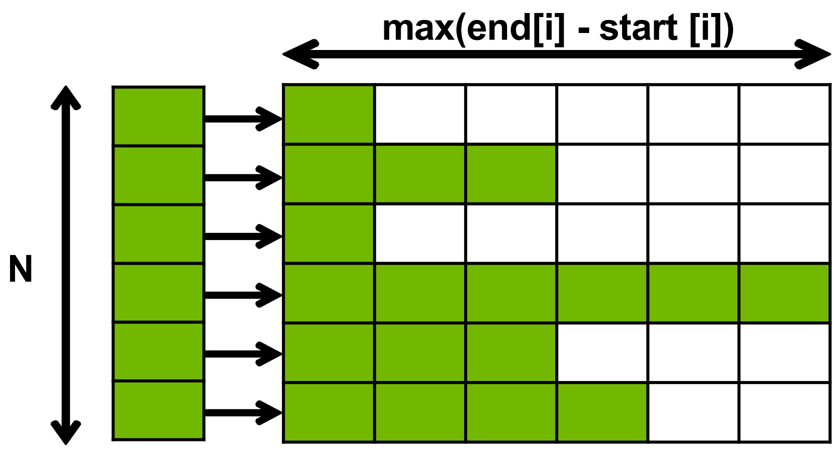


- N threads, each one working on an array
- Arrays have different length

- Sequential loop -> parallelization opportunity missed
- The number of iterations is different for each thread -> branch divergence!

A motivating example: from the code point of view

- Fine-grain solution:
  - Parallelize the loop obtaining a 2D grid (with a new subsequent kernel…)
  - Second dimension sized to the maximum “length” of the for loop
    - `max(end[i] - start[i])`



Several threads not working at all
-> waste of resources!

Code omitted for the sake of simplicity

A motivating example: from the code point of view

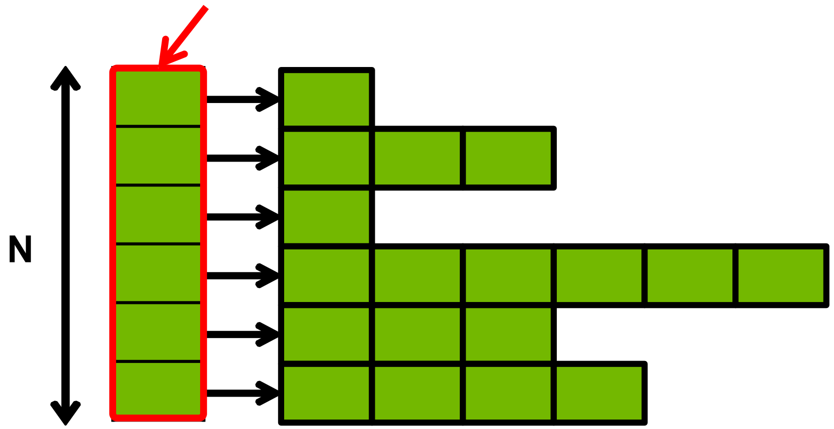
- Dynamic solution:

```
__global__ void parent_kernel(unsigned int* start,
            unsigned int* end, float* someData, float* moreData) {
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  doSomeWork(someData[i]);
  child_kernel <<< ceil((end[i]-start[i])/256.0), 256 >>>
                                   (start[i], end[i], moreData);
}

__global__ void child_kernel(unsigned int start,
                             unsigned int end, float* moreData) {
  unsigned int j = start + blockIdx.x*blockDim.x + threadIdx.x;
  if(j < end)
    doMoreWork(moreData[j]);
}
```

A motivating example: from the code point of view

- Dynamic solution:

```
__global__ void parent_kernel(unsigned int* start,
            unsigned int* end, float* someData, float* moreData) {
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  doSomeWork(someData[i]);
  child_kernel <<< ceil((end[i]-start[i])/256.0), 256 >>>
                                   (start[i], end[i], moreData);
}
```

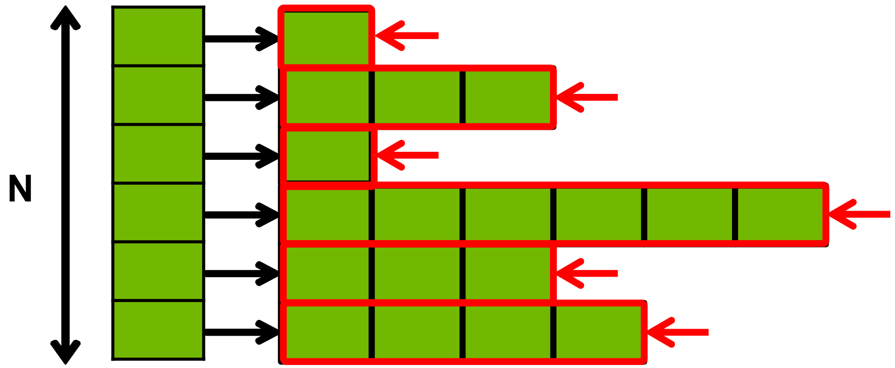


Parent kernel
- Run `doSomeWork`
- Dynamically sizes and launch grid for `doMoreWork`

A motivating example: from the code point of view

- Dynamic solution:

```
__global__ void child_kernel(unsigned int start,
                              unsigned int end, float* moreData) {
  unsigned int j = start + blockIdx.x*blockDim.x + threadIdx.x;
  if(j < end)
    doMoreWork(moreData[j]);
}
```



Child kernel runs `doMoreWork` on a single data item

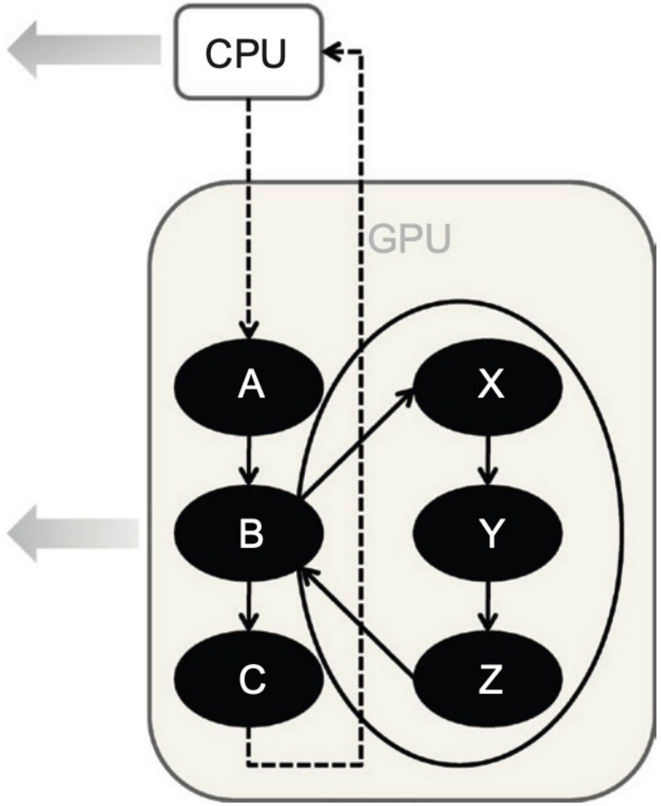
Coding with dynamic parallelism

- The kernel can do almost anything the host does:
  - Launch kernels
  - Allocate global memory
  - Synchronize with the child grid
  - Create streams

  - … the list of functions callable from the device side is restricted w.r.t. the host side

```
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);
}
```



Compiling the code

- Specific flags and libraries have to be specified to compile source code using dynamic parallelism:

```
nvcc -rdc=true -lcudadevrt myprogram.cu -o myprogram
```

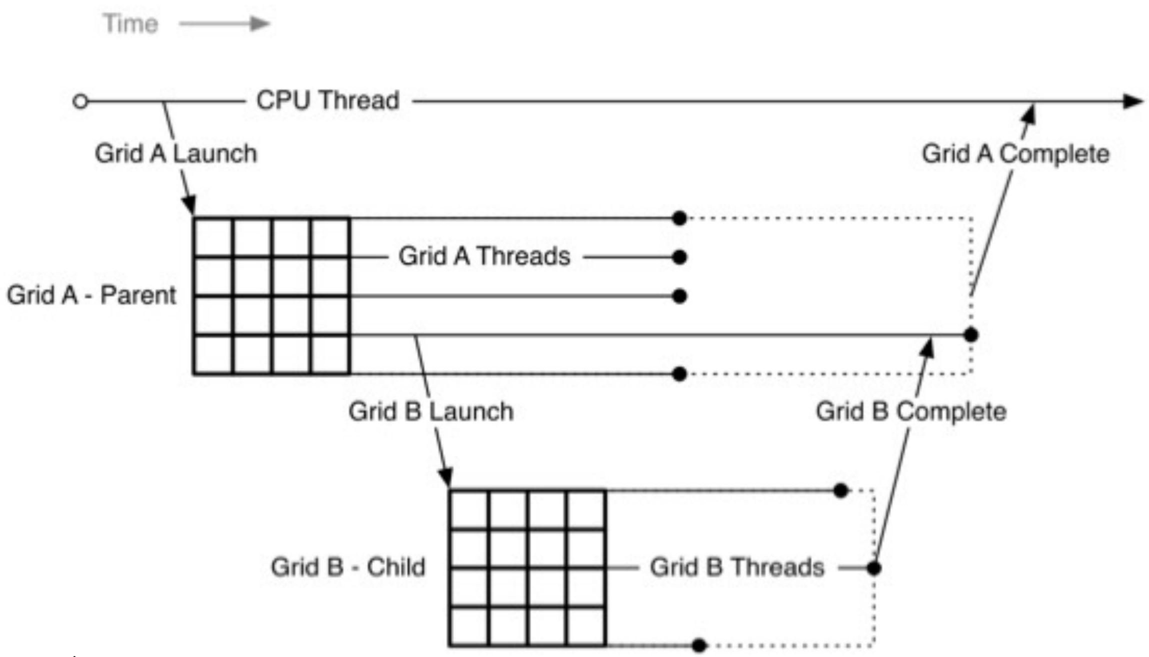
- The compute capability must be >= 3.5

Execution model

- Launch of child kernels is **per-thread and asynchronous**

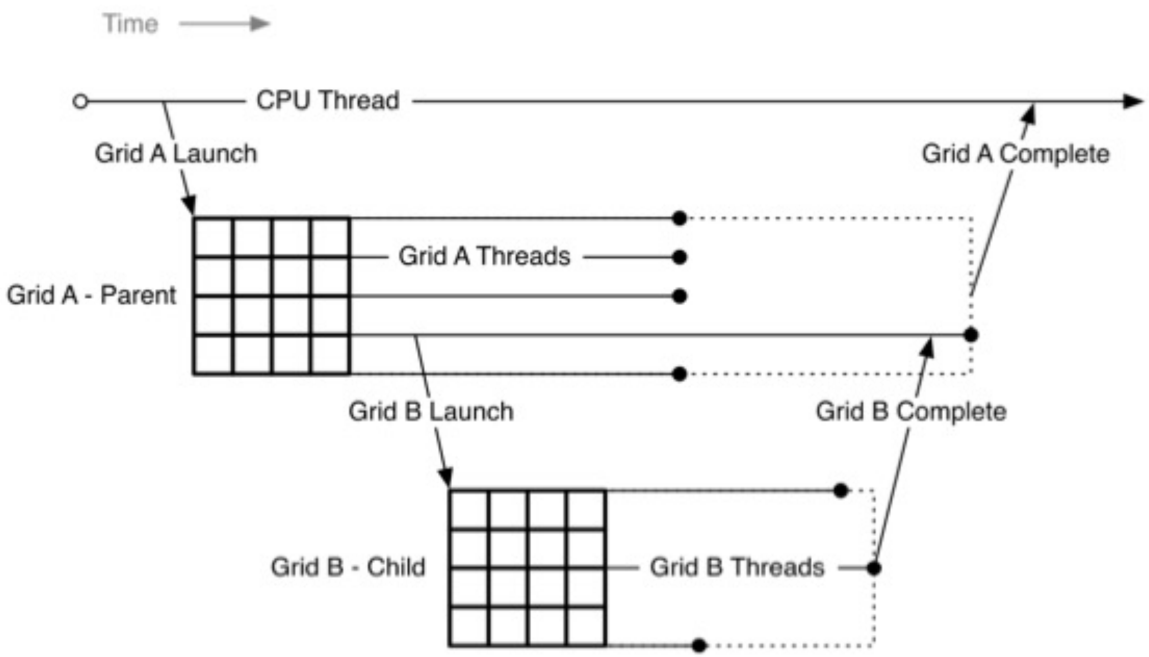
- To launch only a child grid per thread block (or grid), a single thread in the block (or grid) should launch the kernel:

```
if(threadIdx.x == 0)
  child_k <<< ... >>> (...);
```



Execution model

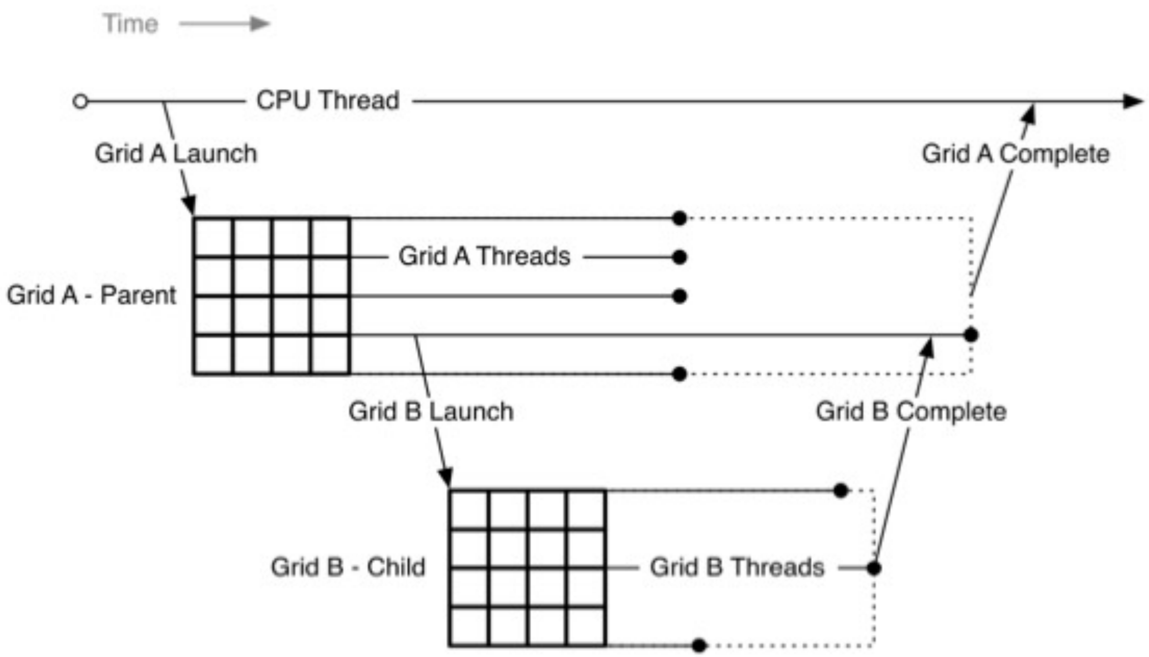
- There is no pre-defined order of execution of parent and child threads
  - Synchronization only at the launch and at the termination (see next slide)



Synchronization

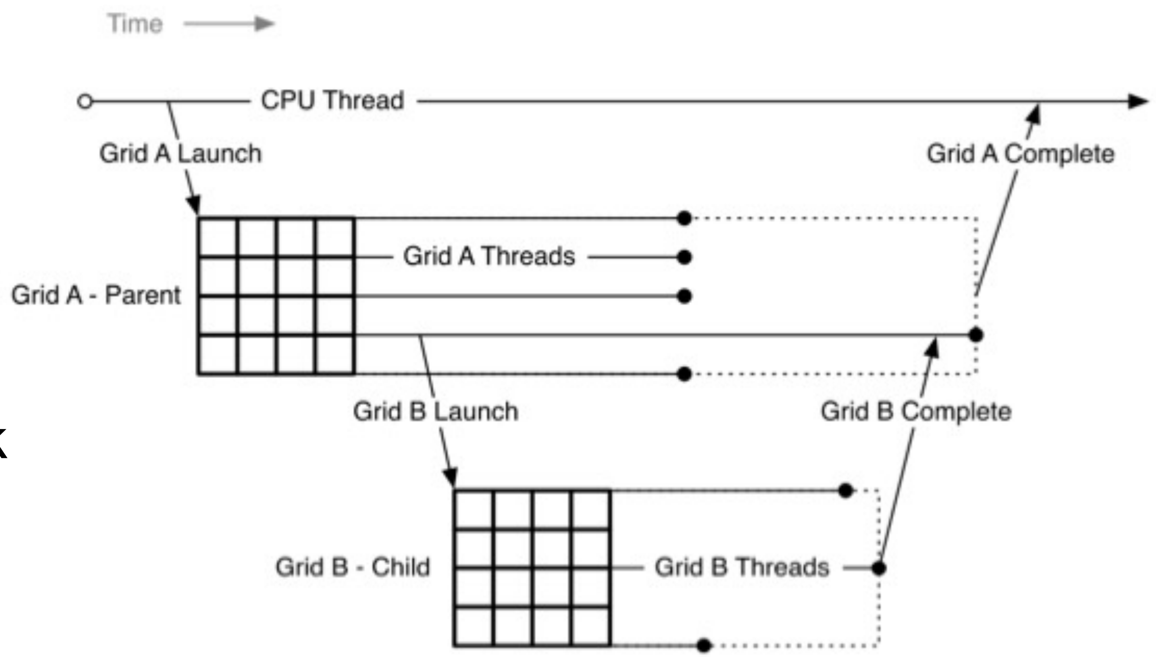
- Grids launched with dynamic parallelism are **fully nested**:
  - Child grids always complete before the parent thread (and block and grid) that launched them
  - An implicit barrier is added in the parent thread

- Child grid is guaranteed to be started only at parent thread synchronization point



Synchronization

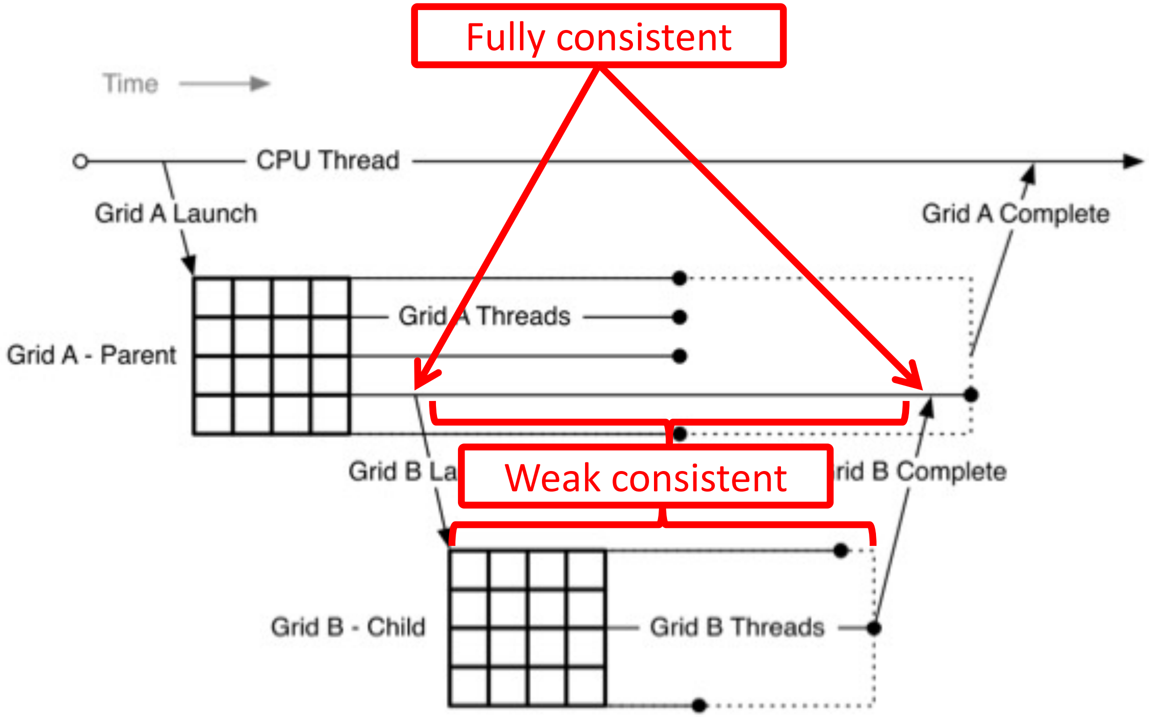
- Explicit synchronization of the parent thread to the all running child grids invoked in the parent block can be performed with `cudaDeviceSynchronize()`
- `__syncthreads()` may be needed to synchronize the block of the thread the launched the child grid



Memory data visibility and consistency

- Visibility of a memory region of a parent thread block to a child grid:
  - **Nonvisible**: local and shared memory data
  - **Visible**: global, texture and constant memory data

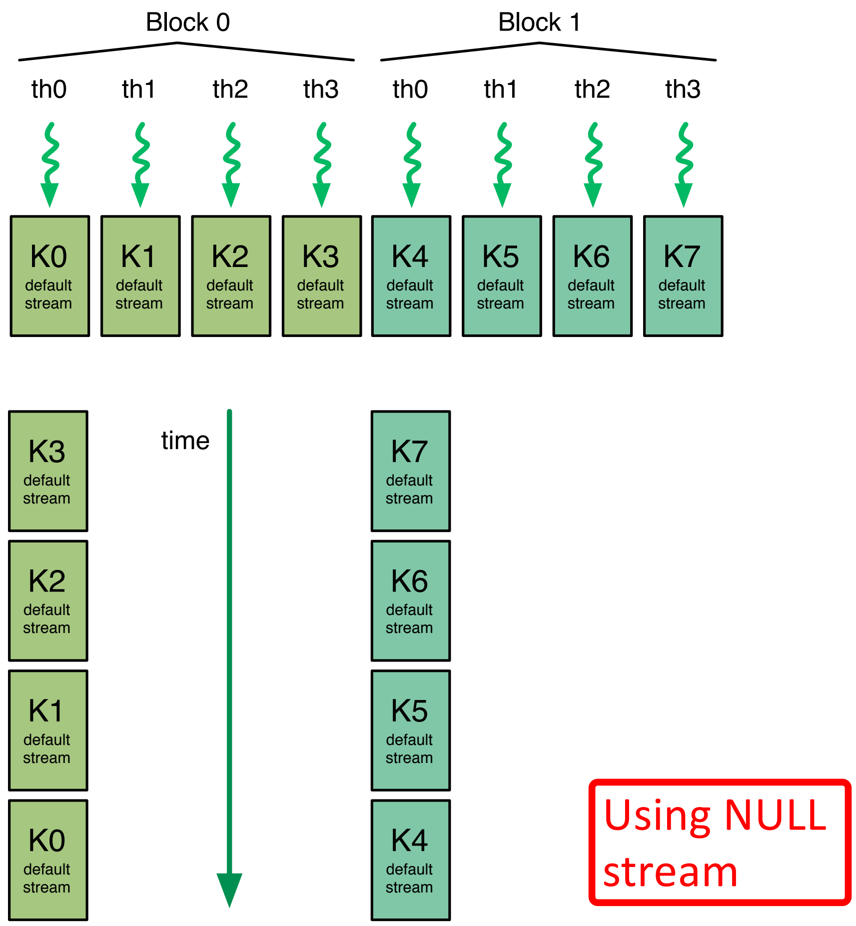
- Memory consistency w.r.t. the parent thread/block:
  - **Weak** while child grid is running
  - **Fully** at child launch time and synchronization



Streams

- Child grids launched within a thread block are executed sequentially
  - They are pushed in the NULL stream

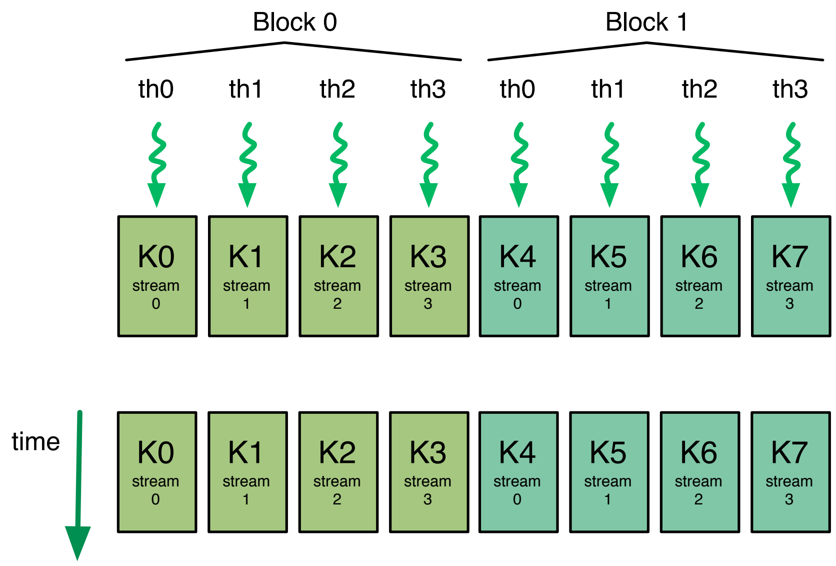
- Concurrent execution of multiple child grids of the same block can be achieved by means of streams



Streams

- Child grids launched within a thread block are executed sequentially
  - They are pushed in the NULL stream

- Concurrent execution of multiple child grids of the same block can be achieved by means of streams



Using user streams

Streams

- Kernel launch when using NULL stream:

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

- Kernel launch when using user streams:

```
cudaStream_t s;
cudaStreamCreateWithFlags(&s, cudaStreamNonBlocking);
kernel_child <<< ceil((end[i]-start[i])/256.0), 256, 0, s >>>
                                    (start[i], end[i], moreData);
cudaDeviceSynchronize();
cudaStreamDestroy(stream);
```

Limits of dynamic parallelism

- Maximum nesting depth of dynamic parallelism is 24
  - In general, device resources (and mainly memory to swap out parent kernels) definitely limit the nesting depth
  - Moreover, there is a limit on synchronization depth on `cudaDeviceSynchronize()` call

- The number of pending kernels is fixed as well (2048 by default)
  - To excide the limit a virtualized pool can be used … degrading performance

- Runtime errors in child kernels are only visible from the host side by calling `cudaGetLastError()`

References

- Slides mainly based on:
  - W.-m. W. Hwu , D. B. Kirk, I. El Hajj, **Programming Massively Parallel Processors: A Hands-on Approach**, **<u>Chapter 21</u>**
  - J. Chen, M. Grossman, T. McKercher, **Professional Cuda C Programming**, **<u>Chapter 3</u>**