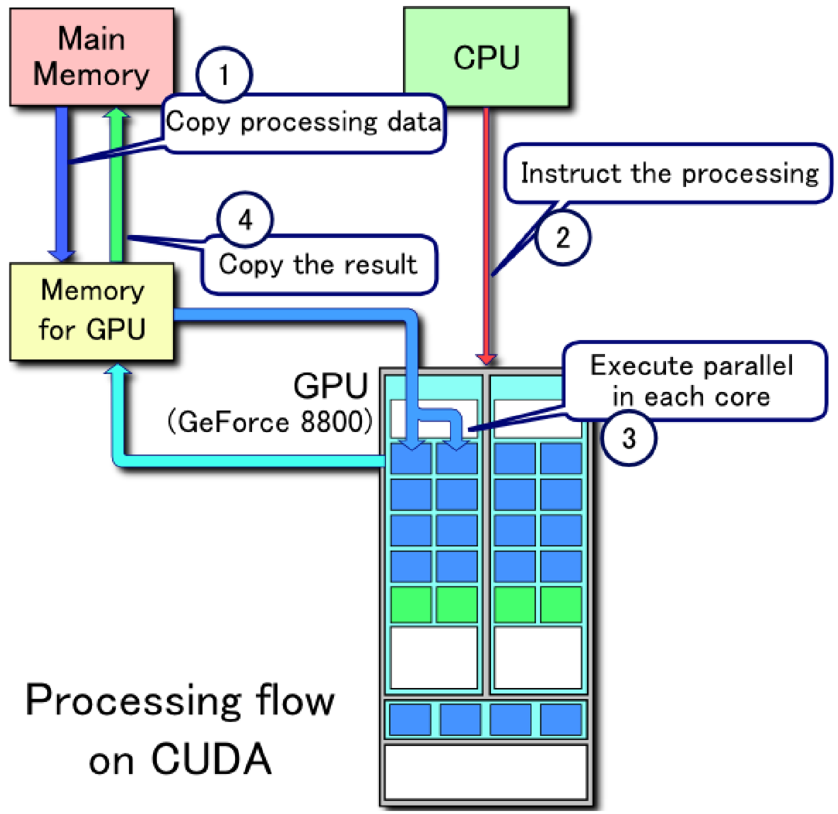
Dynamic Parallelism

Compute Unified Device Architecture

- Hybrid CPU/GPU Code
- Low latency code is running on CPU
  - Result immediately available
- High latency, high throughput code is running on GPU
  - Result on bus
  - GPU has many more cores than CPU



Types of Parallelism

- Different Types of Parallelism
  - Task parallelism (Task farming, Divide and conquer)
    - Problem is divided to tasks, which are processed independently.
  - Data parallelism (SPMD)
    - Same operation is performed over many data items
  - Other types of parallelism
    - Event driven, …

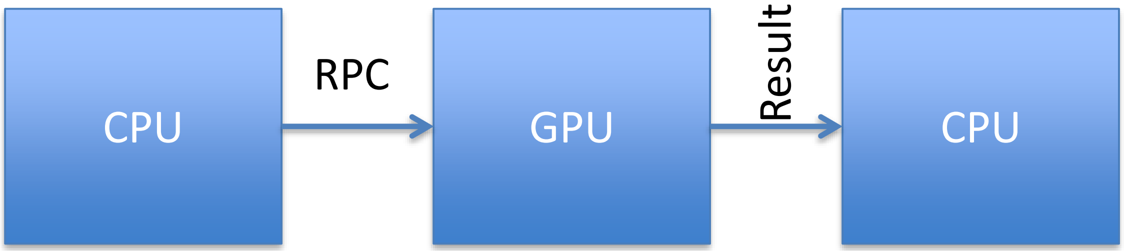
GPU Execution Model

- Parallelism in GPU
  - Data parallelism
    - The same kernel is executed by many threads
    - Thread process one data item
  - Limited task parallelism
    - Multiple kernels executed simultaneously (since Fermi)
  - But we do not have
    - Any means of kernel-wide synchronization (barrier)
    - Any guarantees that two blocks/kernels will actually run concurrently

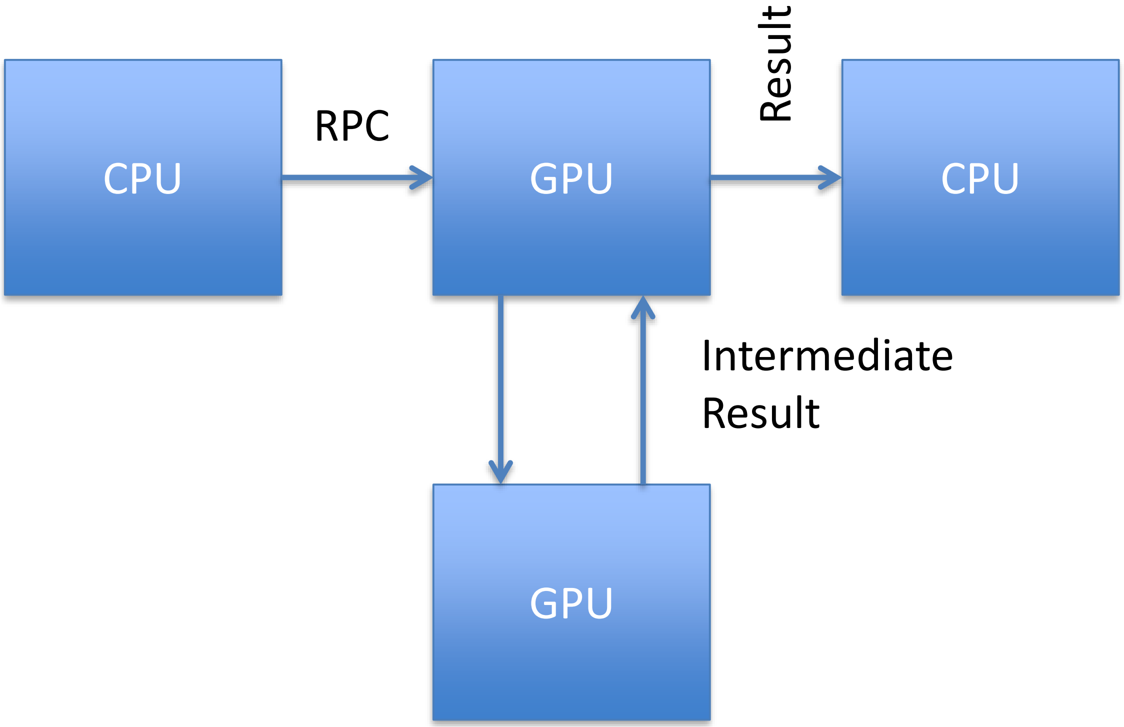
What is Dynamic Parallelism

- The ability to launch new kernels from the GPU
  - Dynamically
    - based on run-time data
  - Simultaneously
    - from multiple threads at once
  - Independently
    - each thread can launch a different grid
- Introduced with CUDA 5.0 and compute capability 3.5 and up

Execution Model (Overview)



Fermi: Only CPU can generate GPU work



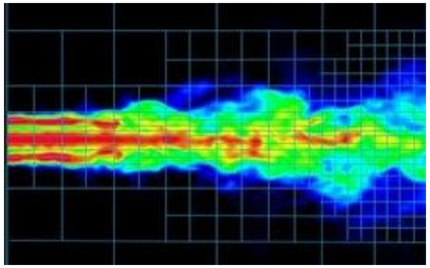
Kepler: GPU can generate work for itself

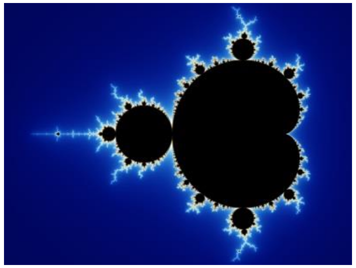
Dynamic Parallelism

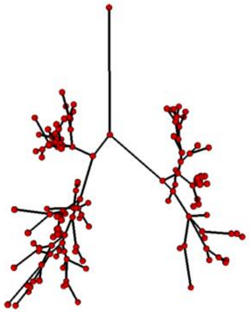
- Allows program flow to be controlled by GPU
- Allows recursion and subdivision of problems
- Interesting when data is not uniformly distributed
- Dynamic parallelism can launch additional threads in interesting areas
- Allows higher resolution in critical areas without slowing down others

Problematic Cases

- Unsuitable Problems for GPUs
  - Processing irregular data structures
    - Trees, graphs, …
  - Regular structures with irregular processing workload
    - Difficult simulations, iterative approximations
  - Iterative tasks with explicit synchronization





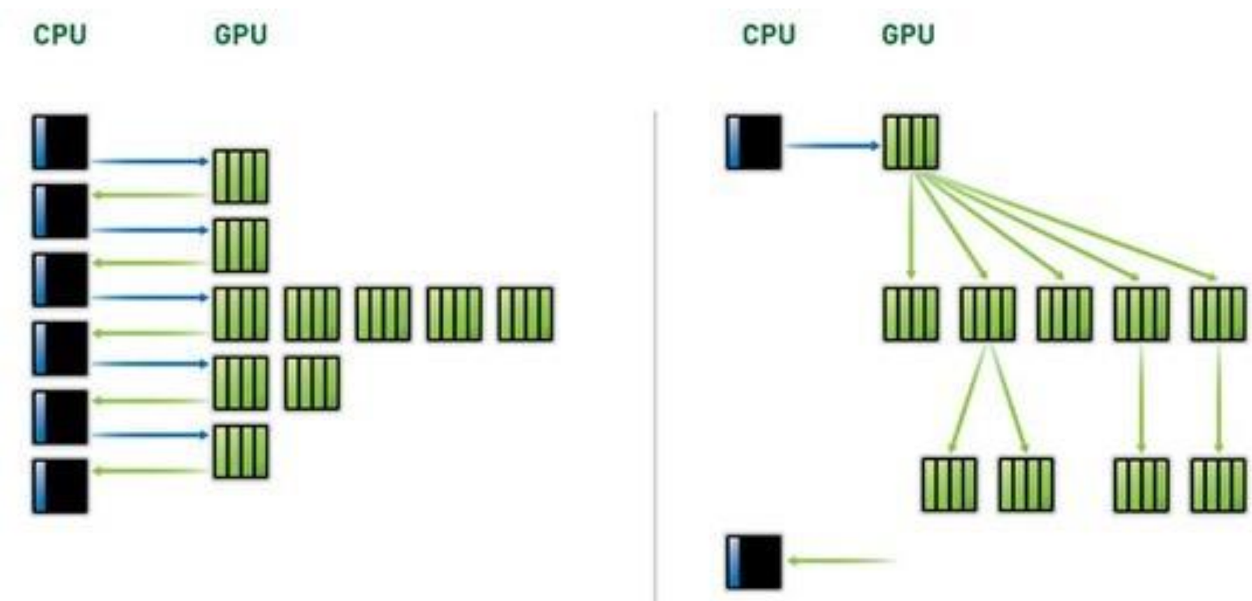


Problematic Cases

- Solutions
  - Iterative kernel execution
    - Usually applicable only for cases when there are none or few dependencies between the subsequent kernels
    - The state (or most of it) is kept on the GPU
  - Mapping irregular structures to regular grids
    - May be too fine/coarse grained
    - Not always possible
  - 2-phase Algorithms
    - First phase determines the amount of work (items, …)
    - Second phase process tasks mapped by first phase

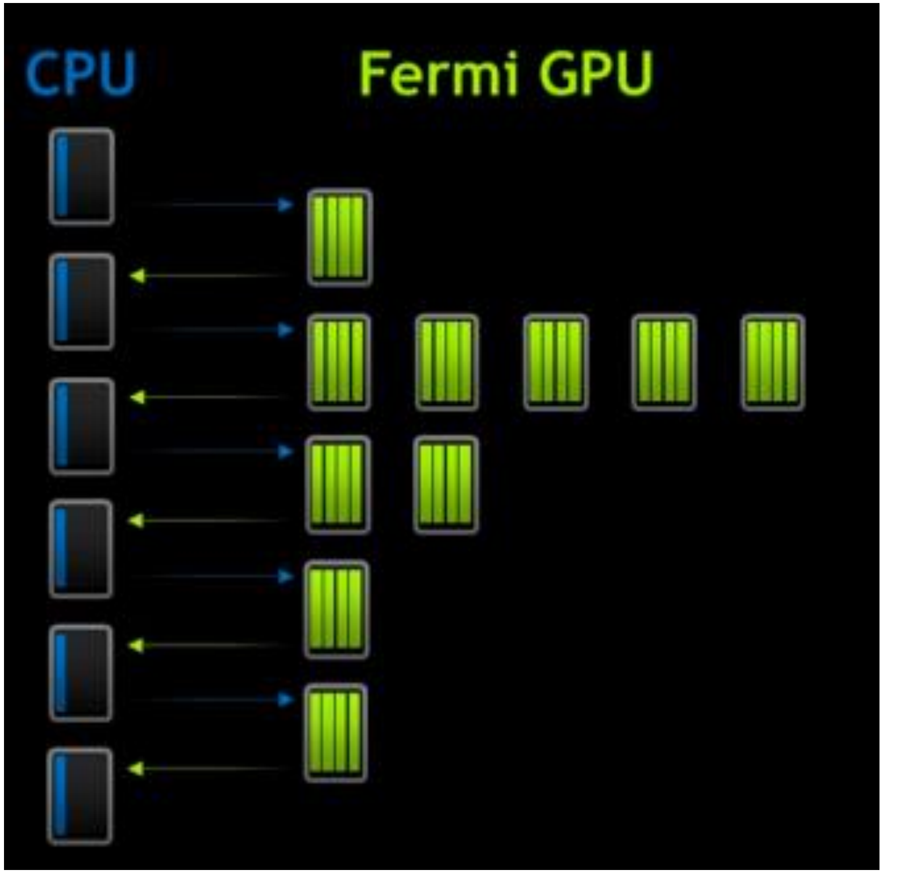
Dynamic Parallelism

- Dynamic Parallelism Purpose
  - The device does not need to synchronize with host to issue new work to the device
  - Irregular parallelism may be expressed more easily

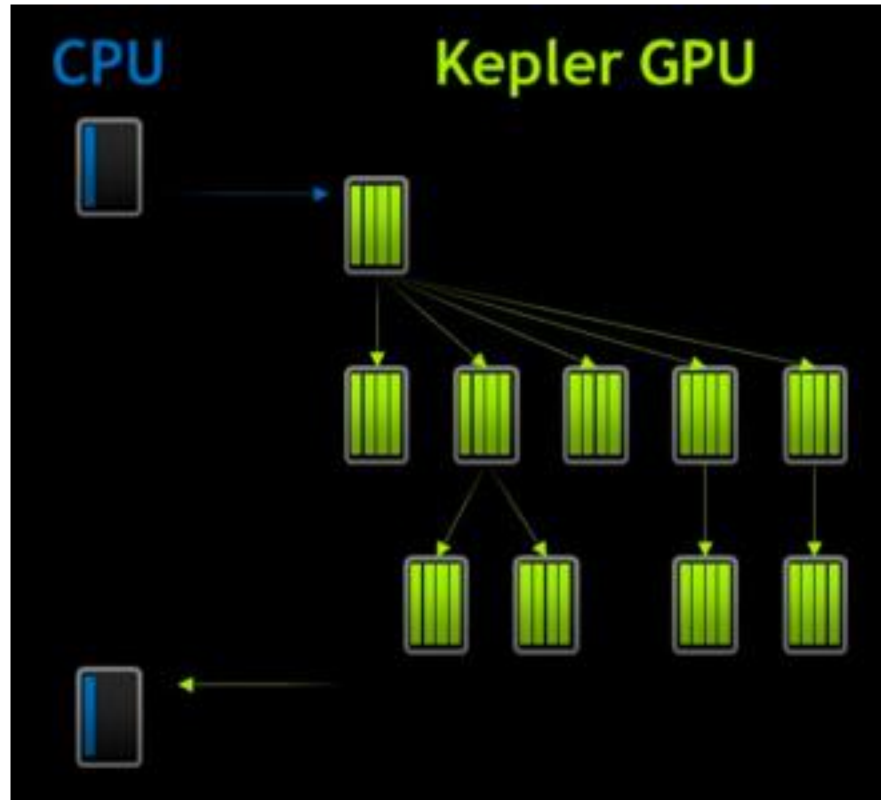


Dynamic Parallelism

- Without Dynamic Parallelism
  - Data travels back and forth between the CPU and GPU many times.
  - This is because of the inability of the GPU to create more work on itself depending on the data.



- With Dynamic Parallelism:
  - GPU can generate work on itself based on intermediate results, without involvement of CPU.
  - Permits Dynamic Run Time decisions.
  - Leaves the CPU free to do other work, conserves power.

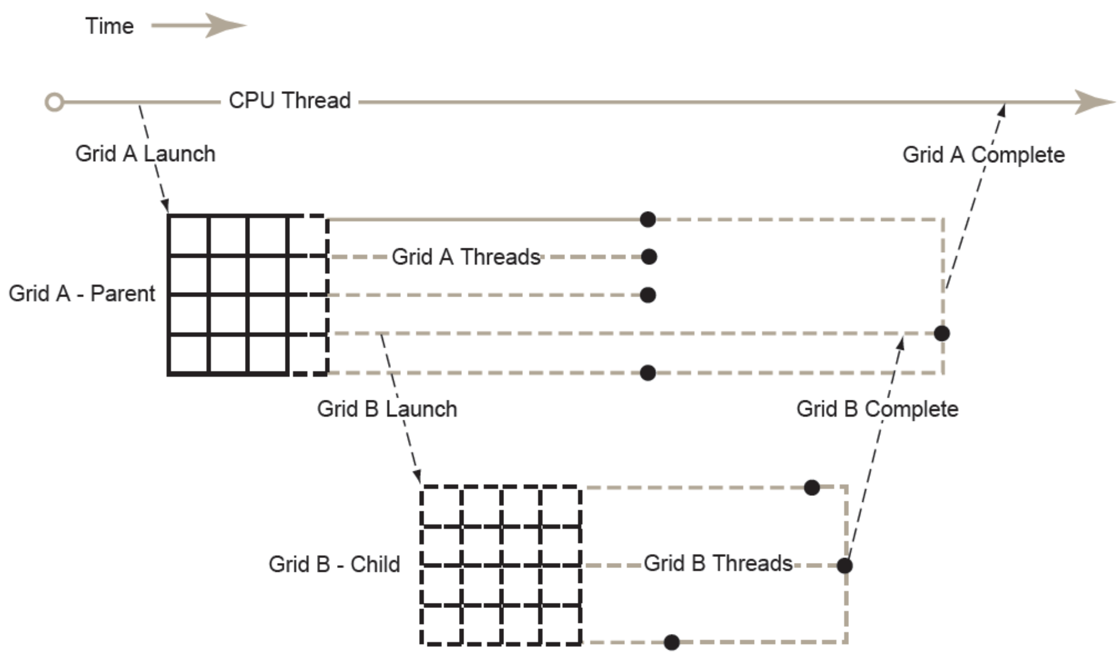


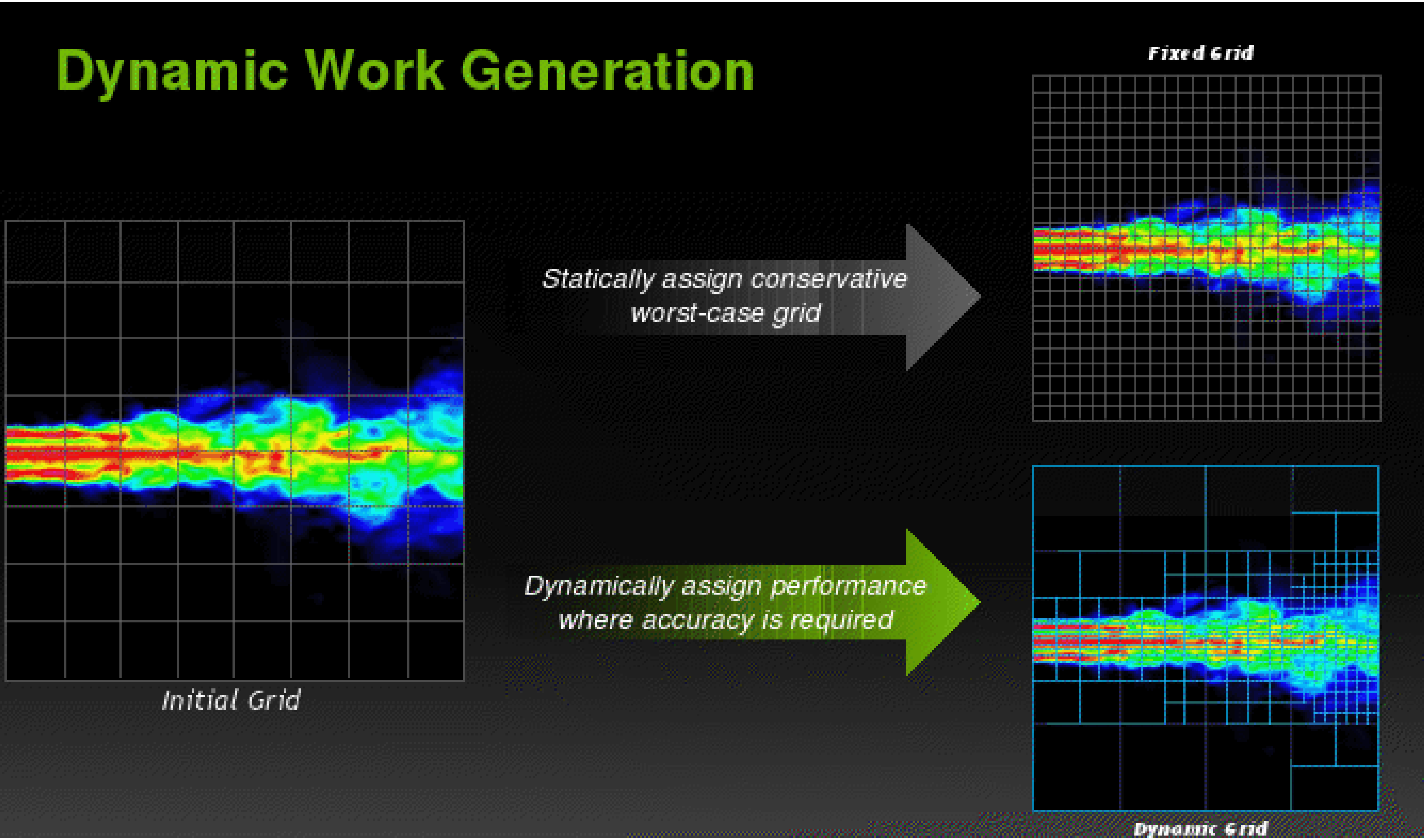
Dynamic Parallelism

- How It Works
  - Portions of CUDA runtime are ported to device side
    - Kernel execution
    - Device synchronization
    - Streams, events, and async memory operations
  - Kernel launches are asynchronous
    - No guarantee the child kernel starts immediately
    - Synchronization points may cause context switch
      - Entire blocks are switched on a SMP

Dynamic Parallelism

- CUDA Dynamic Parallelism
  - New feature presented in CC 3.5 (Kepler)





Source: NVIDIA

Dynamic Parallelism

- Example

```
__global__ void child_launch(int *data) {
    data[threadIdx.x] = data[threadIdx.x]+1;
}

__global__ void parent_launch(int *data) {
    data[threadIdx.x] = threadIdx.x;
    __syncthreads();
    if (threadIdx.x == 0) {
        child_launch<<< 1, 256 >>>(data);
        cudaDeviceSynchronize();
    }
    __syncthreads();
}

void host_launch(int *data) {
    parent_launch<<< 1, 256 >>>(data);
}
```

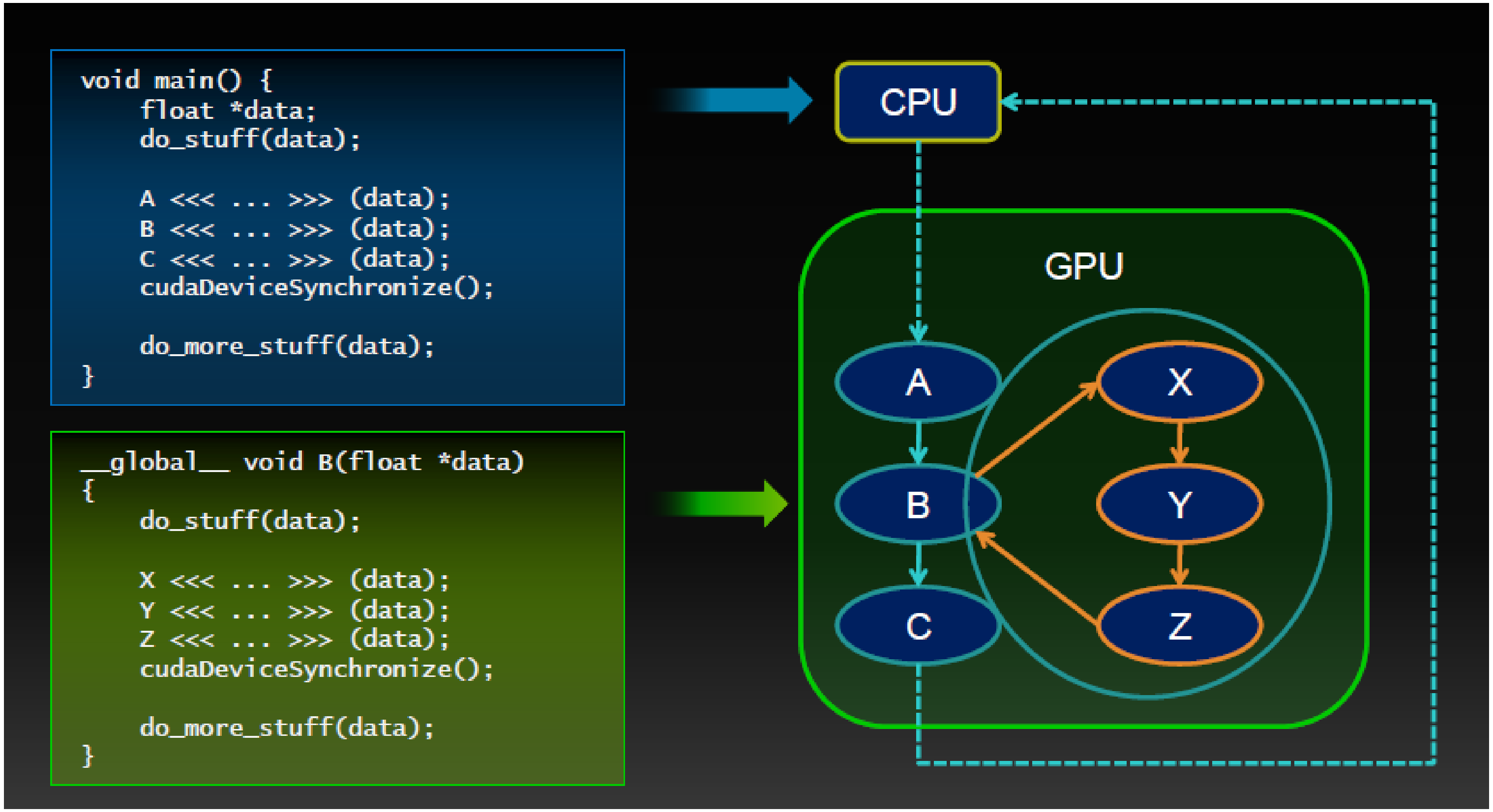
Thread 0 invokes a grid of child threads

Synchronization does not have to be invoked by all threads

Device synchronization does not synchronize threads in the block

Dynamic Parallelism

- Nested Dependencies



Source: NVIDIA

EASY TO PARALLELIZE PROGRAM

```
for i = 1 to N
        for j = 1 to M
                do_something(i, j)
         next j
next i
```

EASY TO PARALLELIZE PROGRAM

Serial Code

```
for i = 1 to N
        for j = 1 to M
                do_something(i, j)
        next j
next i
```

Parallel Code

```
__global__ void callKernel(...)
{
        do_something (blockIdx.x,
                                threadIdx.x)
}
```

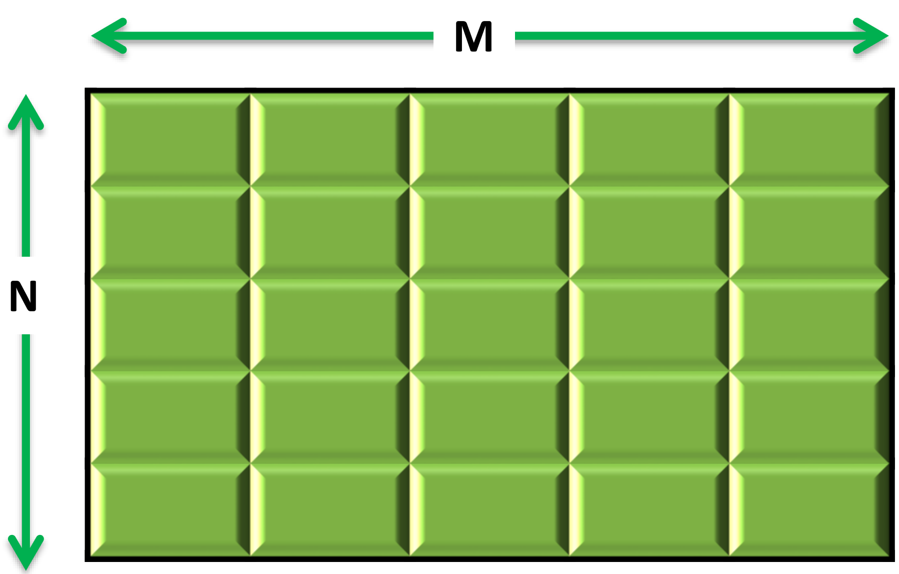
```
void main()
{
        callKernel<<< N, M >>>(...);
}
```

threadIdx.x								threadIdx.x								threadIdx.x								threadIdx.x							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
blockIdx.x = 0								blockIdx.x = 1								blockIdx.x = 2								blockIdx.x = 3							

EASY TO PARALLELIZE PROGRAM

Serial Code

```
for i = 1 to N
        for j = 1 to M
                do_something(i, j)
         next j
next i
```



Parallel Code

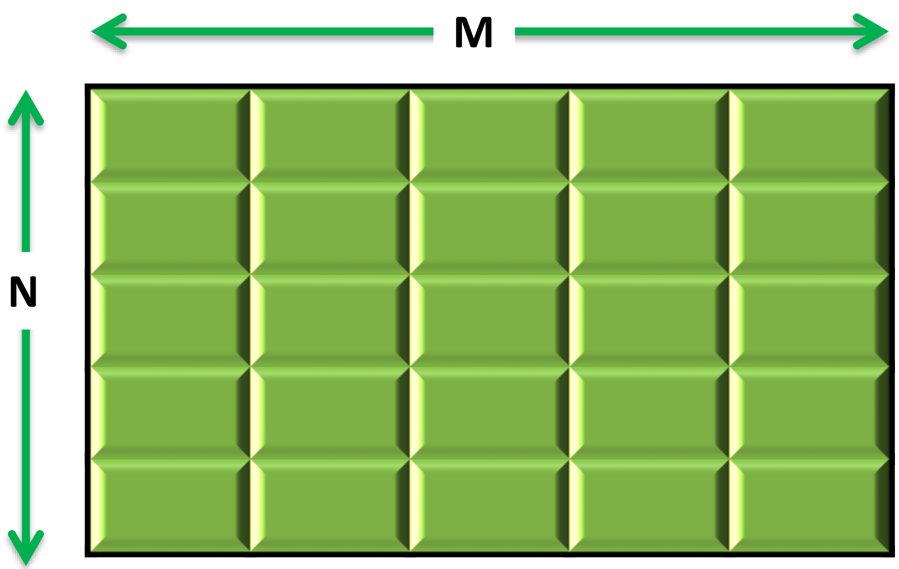
```
__global__ void callKernel( … )
{
        do_something (blockIdx.y,
                                  blockIdx.x)
}
```

```
void main()
{
    dim3 grid(N, M)
    callKernel<<< grid, 1 >>>( … );
}
```

EASY TO PARALLELIZE PROGRAM

Serial Code

```
for i = 1 to N
        for j = 1 to M
             do_something(i, j)
         next j
next i
```



Parallel Code

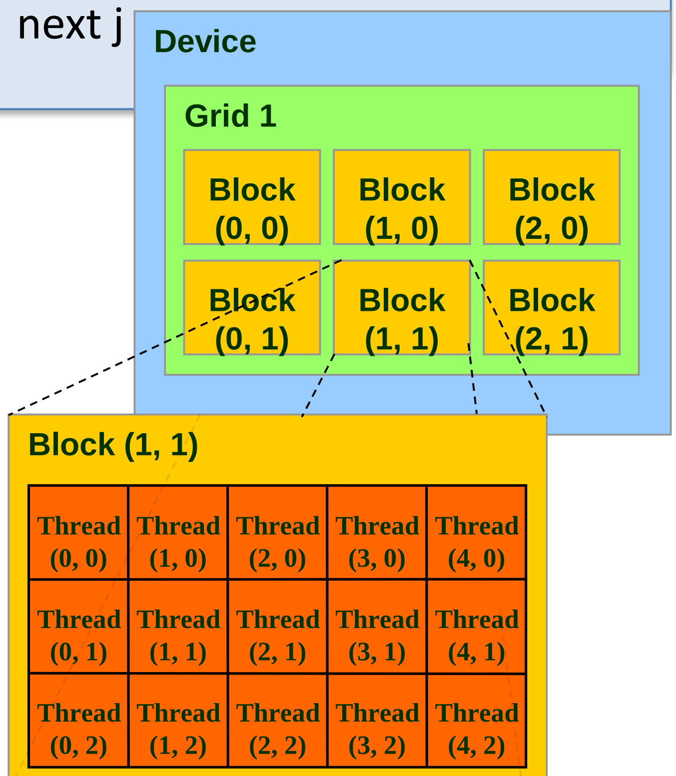
```
__global__ void callKernel( … )
{
     do_something (threadIdx.y,
                   threadIdx.x)
}
```

```
void main()
{
   dim3 block(N, M)
    callKernel<<< 1, block >>>(…);
}
```

EASY TO PARALLELIZE PROGRAM

Serial Code

```
for i = 1 to N
        for j = 1 to M
            do_something(i, j)
        next j
next i
```



Parallel Code

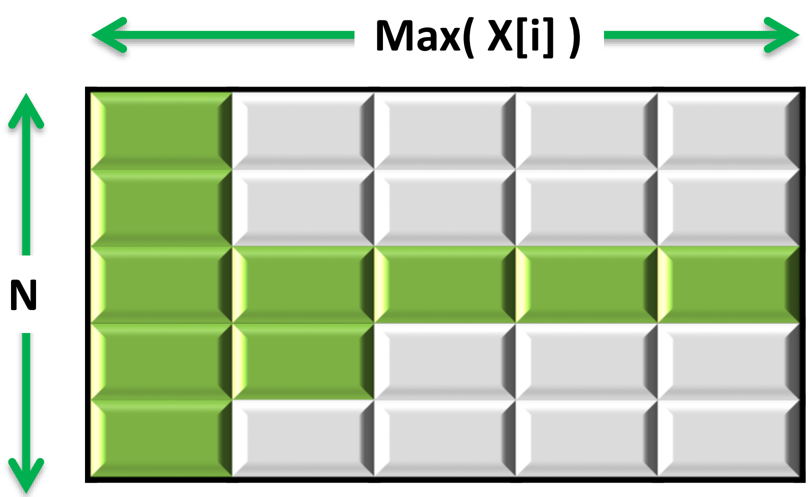
```
__global__ void callKernel( … )
{
    int i = ….;    int j = ……;
    do_something (i, j)
}
```

```
void main()
{  dim3 grid(A, B)
   dim3 block(X = N/A, Y= M/B)
   callKernel<<< grid, block >>>(…);
}
```

DIFFICULT TO PARALLELIZE PROGRAM

Serial Code

```
for i = 0 to N
        for j = 0 to x[i]
                do_something(i, j)
        next j
next i
```



Bad alternative : Idle Threads

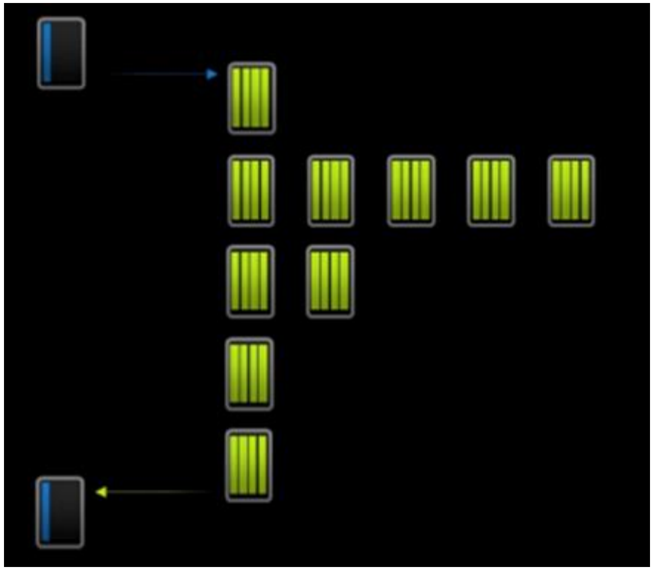
```
__global__ void callKernel(int x[], …)
{       int i = blockIdx.x;
        int j = threadIdx.x;
        if ( j < x[i])
                do_something (i, j)
}
```

```
void main()
{
        callKernel<<< N, max(x) >>>(x…);
}
```

DIFFICULT TO PARALLELIZE PROGRAM

Serial Code

```
for i = 1 to N
        for j = 1 to x[i]
                do_something(i, j)
         next j
next i
```



Dynamic Parallel Code

```
__global__ void ChildKernel(int i) {
        int j = threadIdx.x;
        do_something(i, j);
}
```

```
__global__ void kernel( … )
{       int i = threadIdx.x;
        Childkernel<<< 1, x[i]>>>(i)
}
```

```
void main()
{
      kernel<<< 1, N >>>(x);
}
```

Dynamic Parallelism

- Nested Dependencies - cudaDeviceSynchronize ()
- Can be used inside a kernel
- Synchronizes all launches by any kernel in block
- Does NOT imply __syncthreads()!

Dynamic Parallelism

- Kernel launch implies memory sync operation
- Child sees state at time of launch
- Parent sees child writes after sync
- Local and shared memory are private, cannot be shared with children