

The background of the slide features a large, stylized, metallic NVIDIA logo. The logo is composed of three curved, overlapping segments that form a partial 'V' shape. It has a brushed metal texture and is set against a dark, textured background that resembles a fine mesh or woven fabric.

# Kepler and CUDA5.0 on the Cray XK7

Peter Messmer

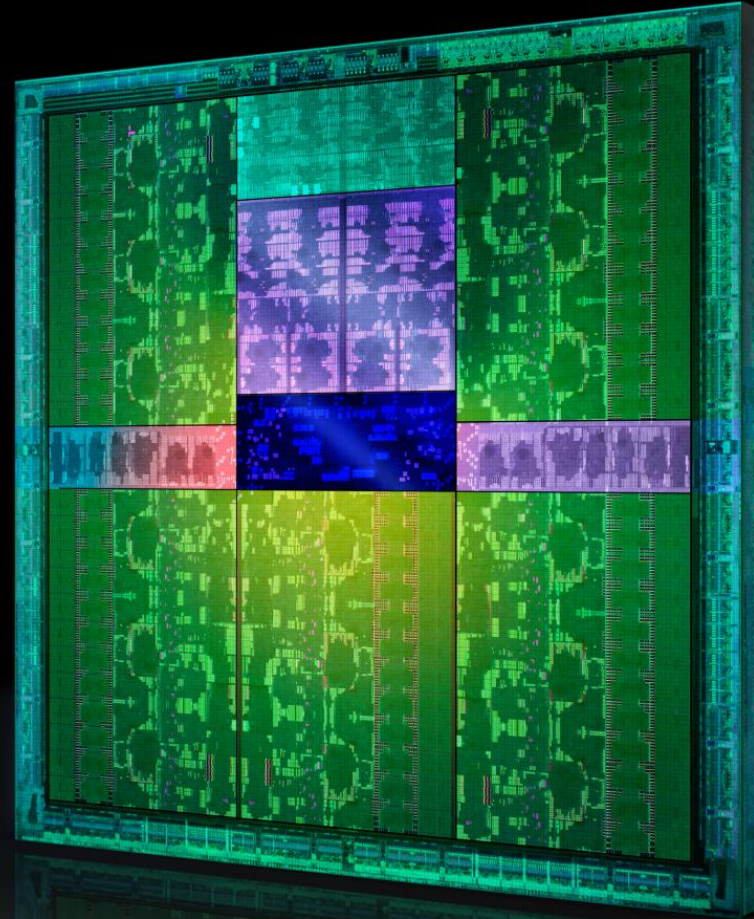


# The Kepler GK110 GPU

Performance

Efficiency

Programmability

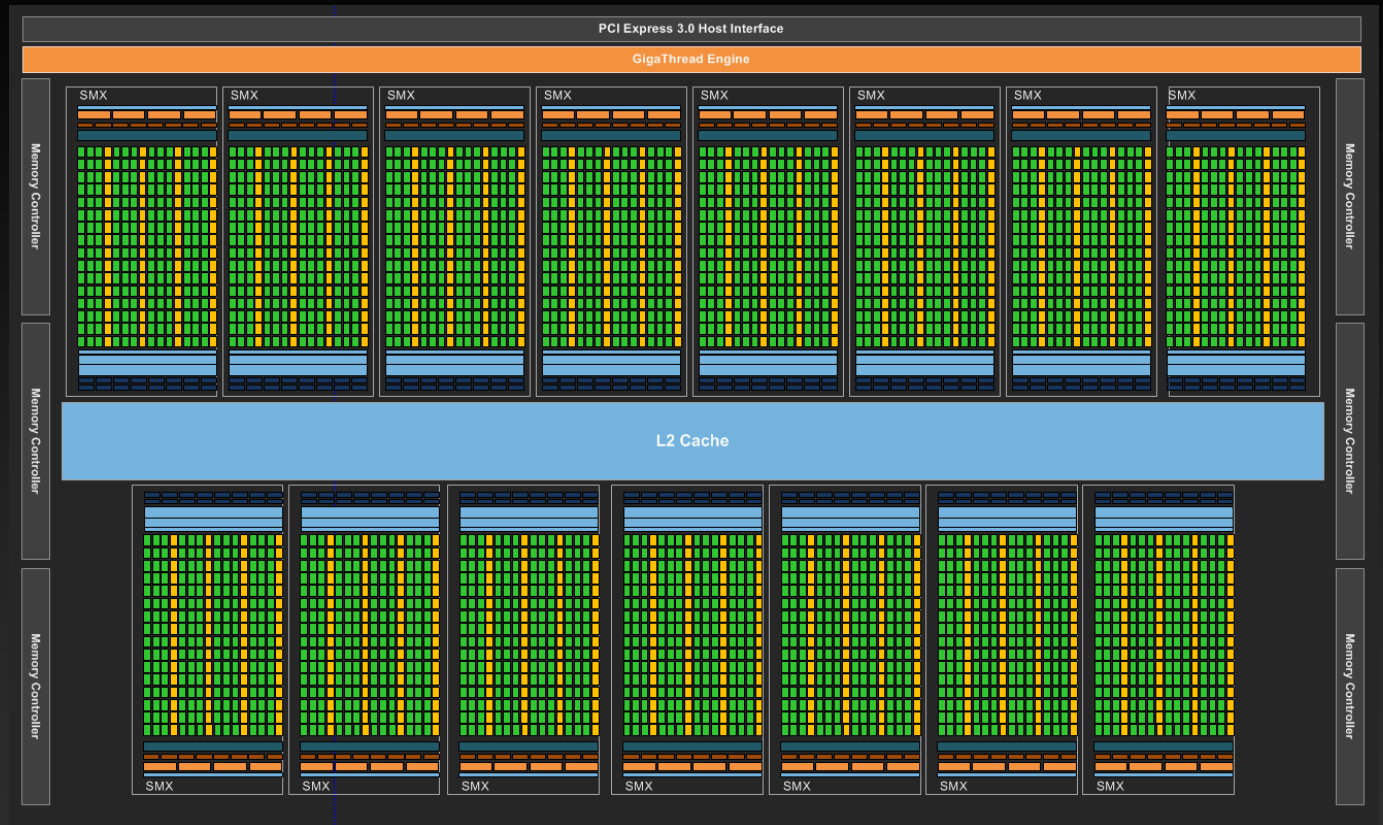




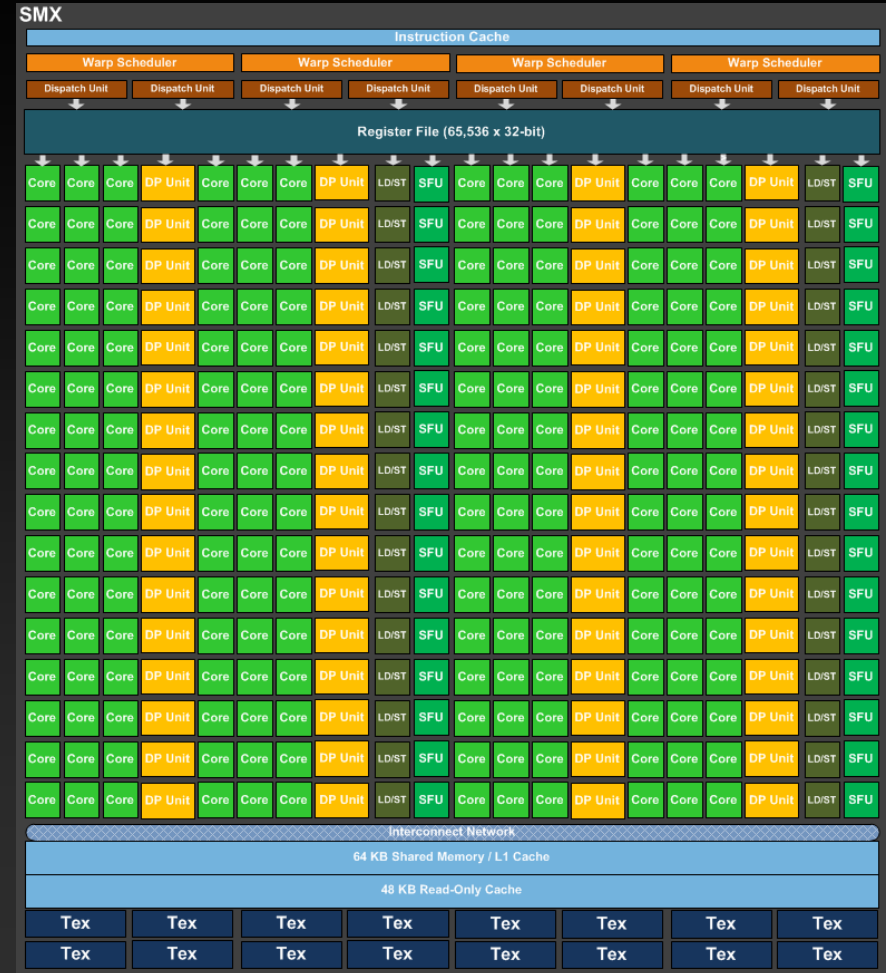
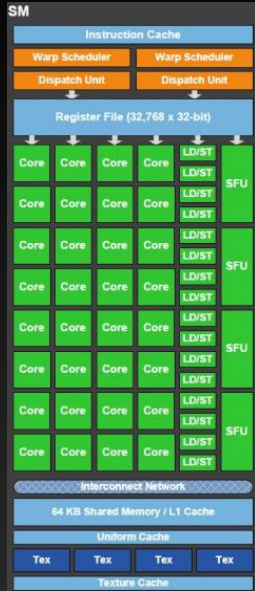
# Kepler GK110 Block Diagram

## Architecture

- 7.1B Transistors
- 15 SMX units
- > 1 TFLOP FP64
- 1.5 MB L2 Cache
- 384-bit GDDR5



# Kepler GK110 SMX vs Fermi SM



# SMX Balance of Resources

Resource	Kepler GK110 vs Fermi
<i>Floating point throughput</i>	<b>2-3x</b>
<i>Max Blocks per SMX</i>	<b>2x</b>
<i>Max Threads per SMX</i>	<b>1.3x</b>
<i>Register File Bandwidth</i>	<b>2x</b>
<i>Register File Capacity</i>	<b>2x</b>
<i>Shared Memory Bandwidth</i>	<b>2x</b>
<i>Shared Memory Capacity</i>	<b>1x</b>

# New ISA Encoding: 255 Registers per Thread

- **Fermi limit: 63 registers per thread**
  - A common Fermi performance limiter
  - Leads to excessive spilling
- **Kepler : Up to 255 registers per thread**
  - Especially helpful for FP64 apps
  - Spills are eliminated with extra registers

# New High-Performance SMX Instructions

**SHFL (shuffle) -- Intra-warp data exchange**

**ATOM -- Broader functionality, Faster**

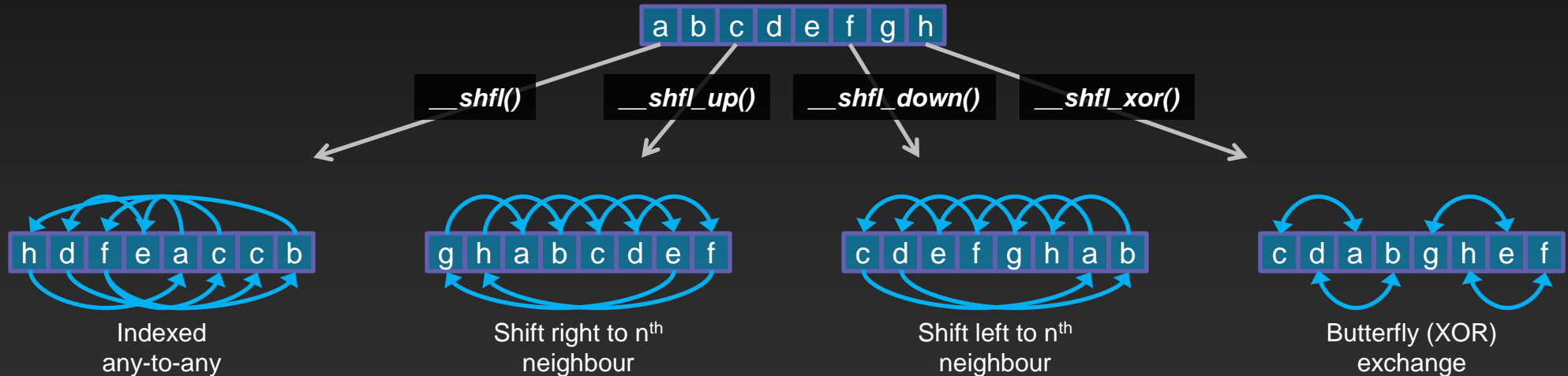
**Compiler-generated,  
high performance  
instructions:**

- ☐ bit shift
- ☐ bit rotate
- ☐ fp32 division
- ☐ read-only cache

# New Instruction: SHFL

## Data exchange between threads within a warp

- Avoids use of shared memory
- One 32-bit value per exchange
- 4 variants:



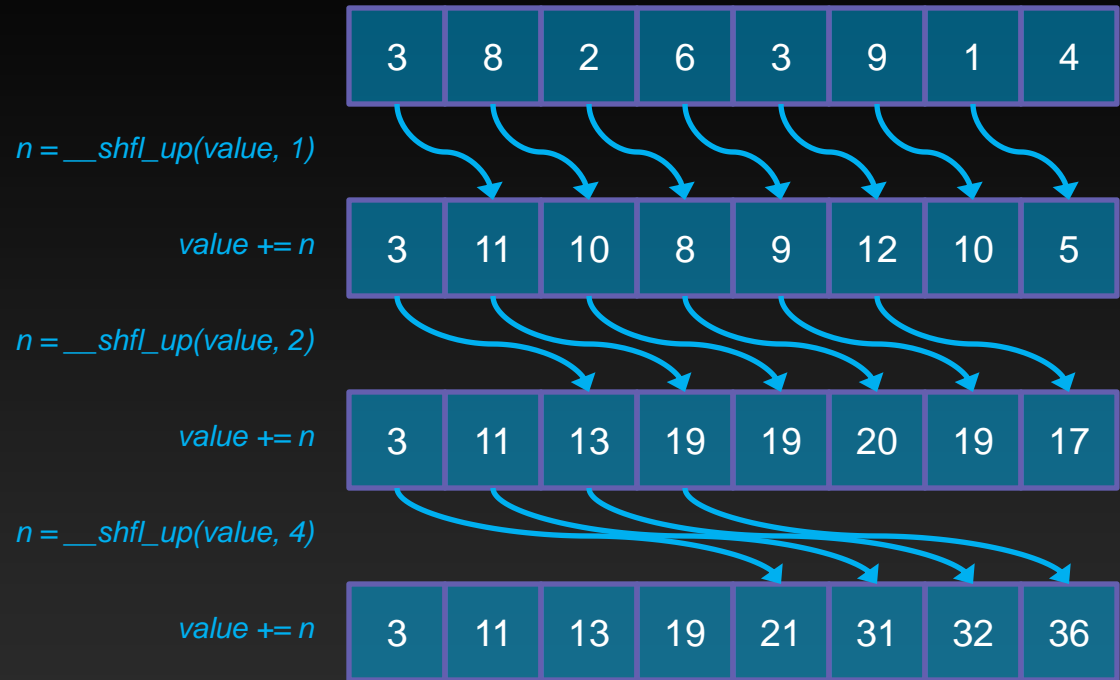


# SHFL Example: Warp Prefix-Sum

```
__global__ void shfl_prefix_sum(int *data)
{
    int id = threadIdx.x;
    int value = data[id];
    int lane_id = threadIdx.x & warpSize;

    // Now accumulate in log2(32) steps
    for(int i=1; i<=width; i*=2) {
        int n = __shfl_up(value, i);
        if(lane_id >= i)
            value += n;
    }

    // Write out our result
    data[id] = value;
}
```



# ATOM instruction enhancements

- Added int64 functions to match existing int32

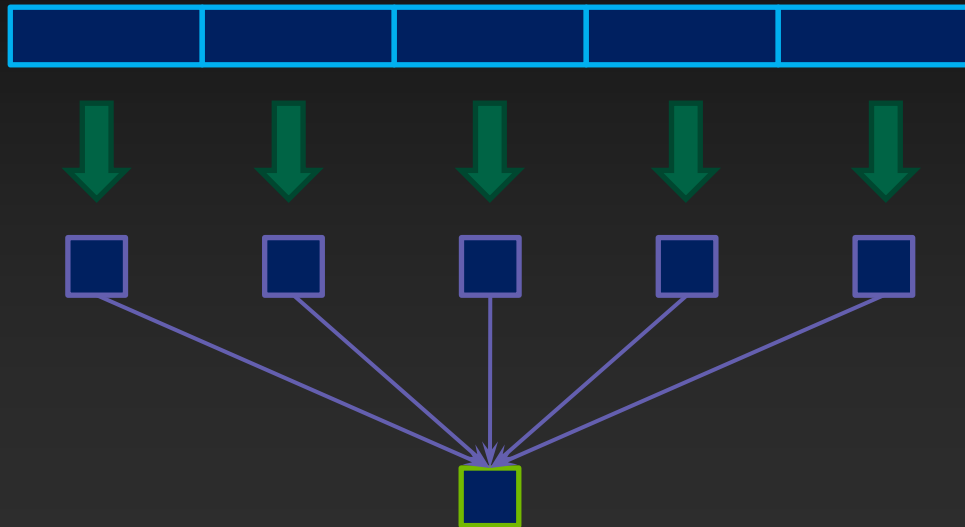
Atom Op	int32	int64
add	x	x
cas	x	x
exch	x	x
min/max	x	x
and/or/xor	x	x

- 2 – 10x performance gains
  - Shorter processing pipeline
  - More atomic processors
  - Slowest 10x faster
  - Fastest 2x faster

# High Speed Atomics Enable New Uses

Atoms are now fast enough to use within inner loops

- Example: Data reduction (sum of all values)



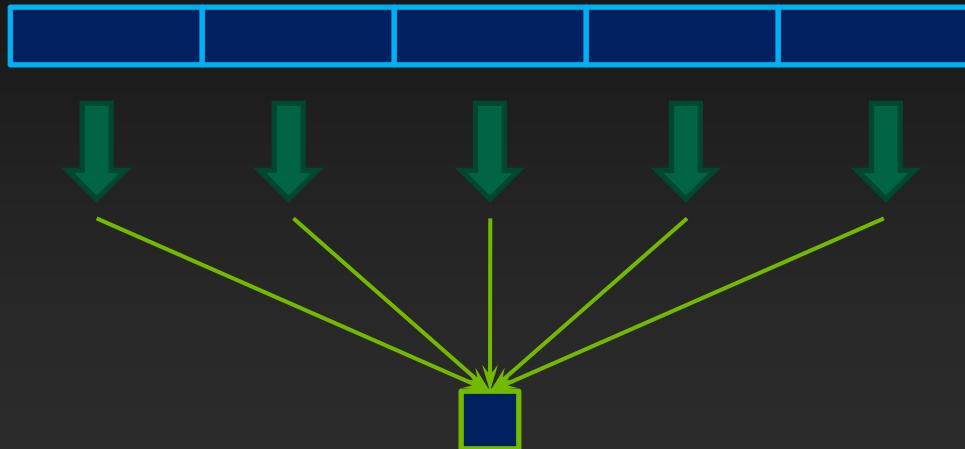
Without Atomics

1. Divide input data array into N sections
2. Launch N blocks, each reduces one section
3. Output is N values
4. Second launch of N threads, reduces outputs to single value

# High Speed Atomics Enable New Uses

Atoms are now fast enough to use within inner loops

- Example: Data reduction (sum of all values)

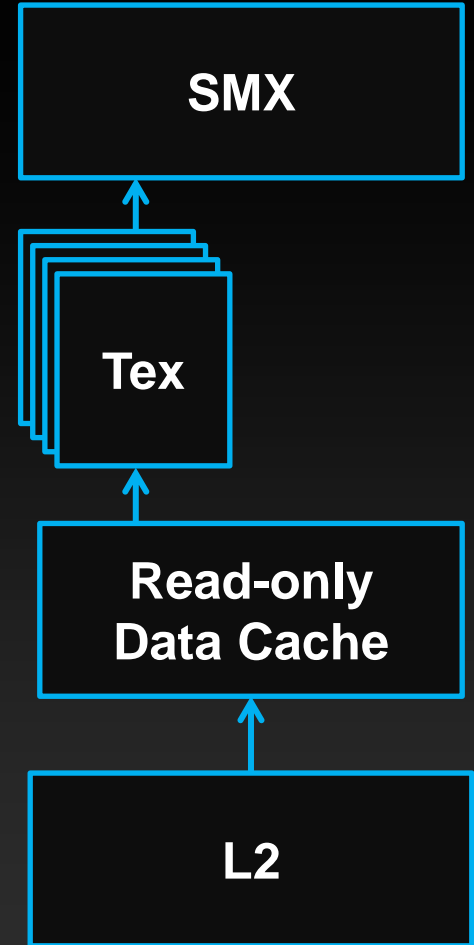


With Atomics

1. Divide input data array into N sections
2. Launch N blocks, each reduces one section
3. Write output directly via atomic.  
No need for second kernel launch.

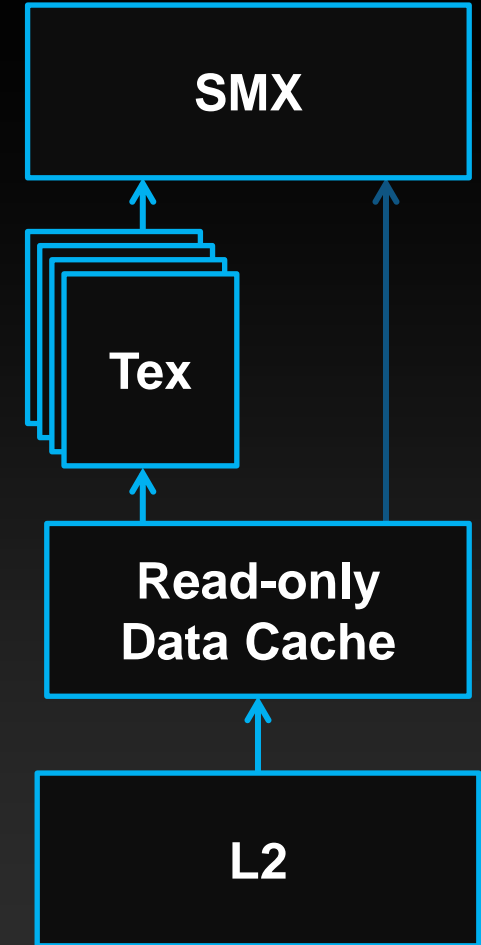
# Texture performance

- **Texture :**
  - Provides hardware accelerated filtered sampling of data (1D, 2D, 3D)
  - Read-only data cache holds fetched samples
  - Backed up by the L2 cache
- **SMX vs Fermi SM :**
  - 4x filter ops per clock
  - 4x cache capacity



# Texture Cache Unlocked

- Added a new path for compute
  - Avoids the texture unit
  - Allows a global address to be fetched and cached
  - Eliminates texture setup
- Why use it?
  - Separate pipeline from shared/L1
  - Highest miss bandwidth
  - Flexible, e.g. unaligned accesses
- Managed automatically by compiler
  - “const \_\_restrict\_\_” indicates eligibility





# const \_\_restrict\_\_ Example

- Annotate eligible kernel parameters with `const __restrict__`
- Compiler will automatically map loads to use read-only data cache path

```
__global__ void saxpy(float x, float y,  
                    const float * __restrict__ input,  
                    float * output)  
{  
    size_t offset = threadIdx.x +  
                    (blockIdx.x * blockDim.x);  
  
    // Compiler will automatically read-only  
    // data cache for "input"  
    output[offset] = (input[offset] * x) + y;  
}
```

# Kepler GK110 Memory System Highlights

- **Efficient memory controller for GDDR5**
  - Peak memory clocks achievable
- **More L2**
  - Double bandwidth
  - Double size
- **More efficient DRAM ECC Implementation**
  - DRAM ECC lookup overhead reduced by 66%  
(average, from a set of application traces)

# Optimizing for Kepler



Fermi code runs on Kepler as is



Better results – recompile code for Kepler



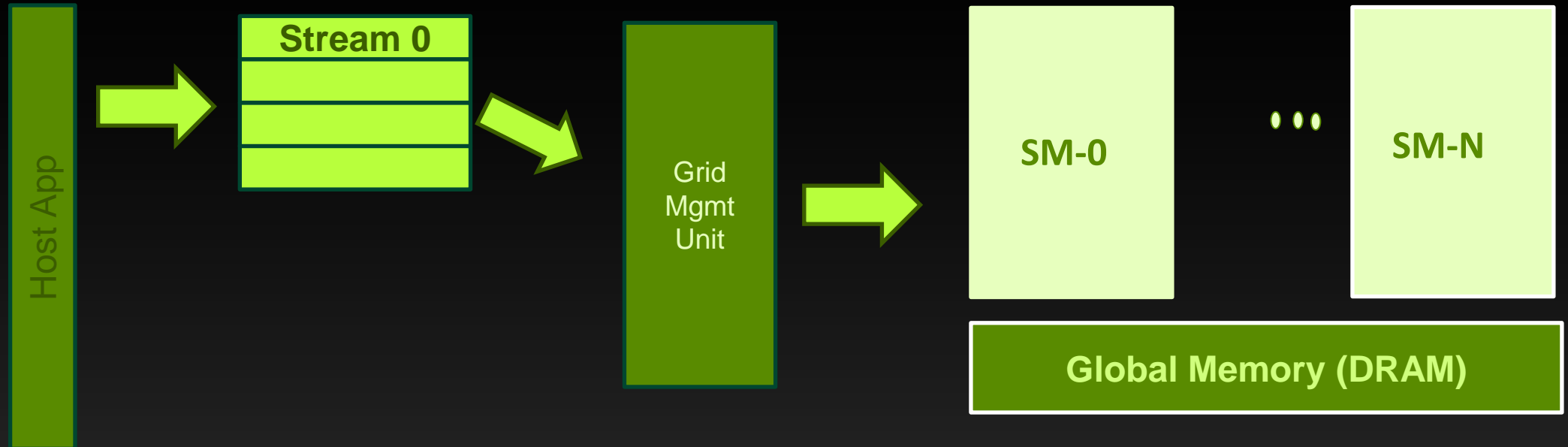
Best performance - tune code for Kepler

The background of the slide features a large, stylized, metallic NVIDIA logo. The logo is composed of three curved, overlapping segments that form a shield-like shape. It has a brushed metal texture and is set against a dark, textured background that resembles a fine grid or mesh.

# **Grid-Level Concurrency with Kepler**



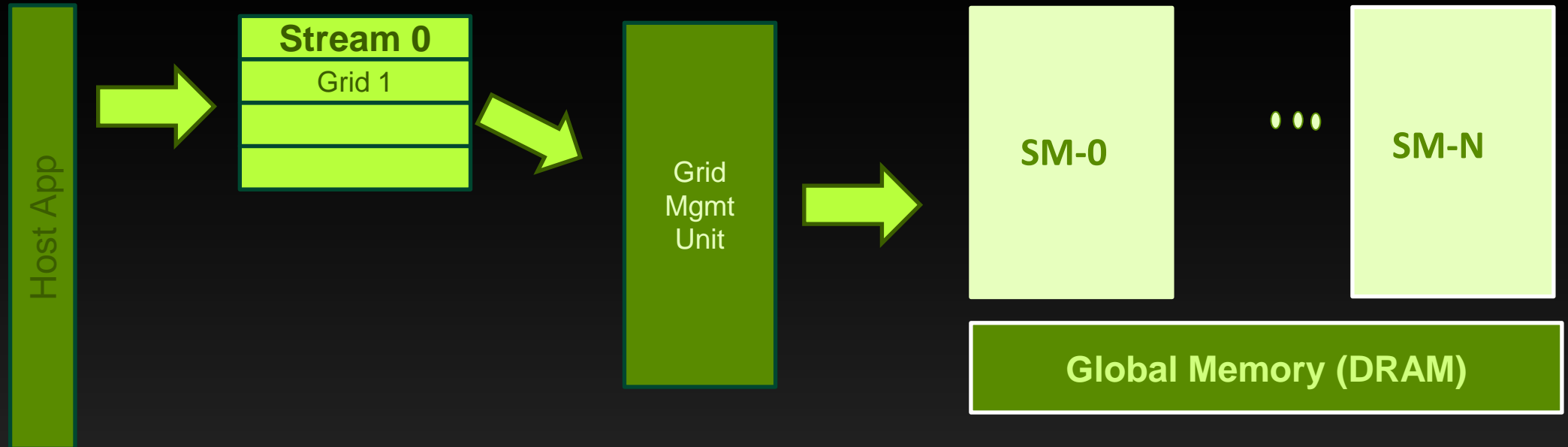
# Streams



Kernel launches

- within the same stream are in-order

# Streams

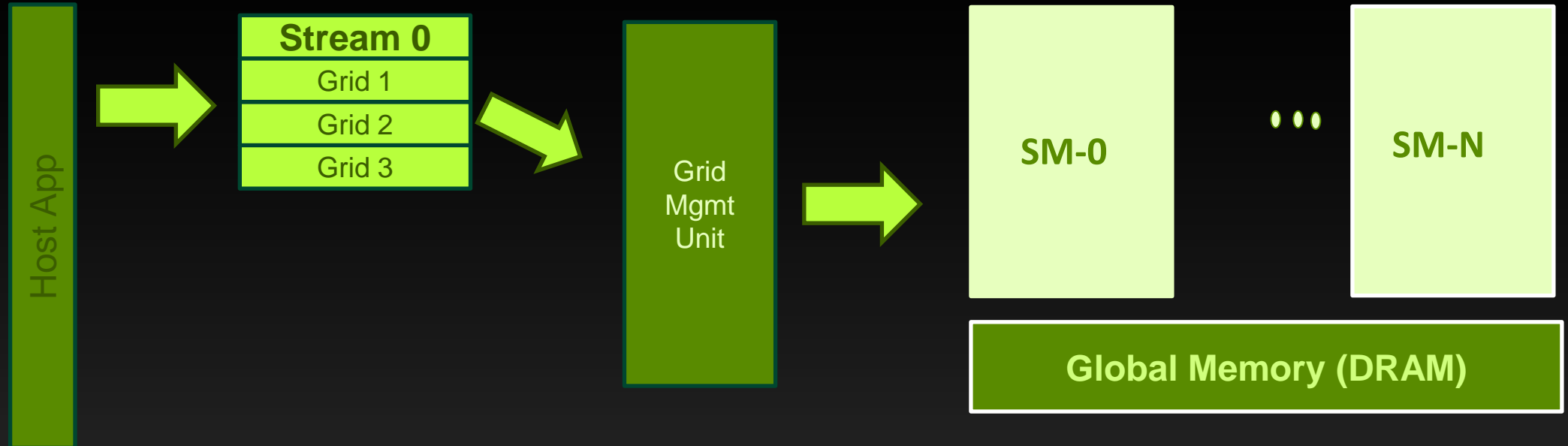


## Kernel launches

- within the same stream are in-order



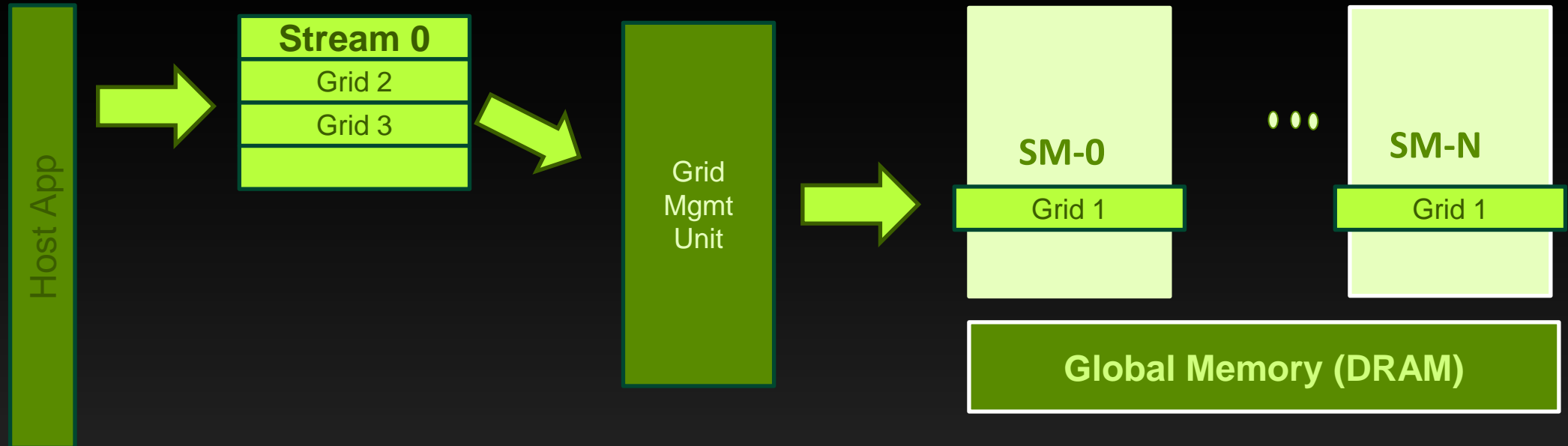
# Streams



## Kernel launches

- within the same stream are in-order

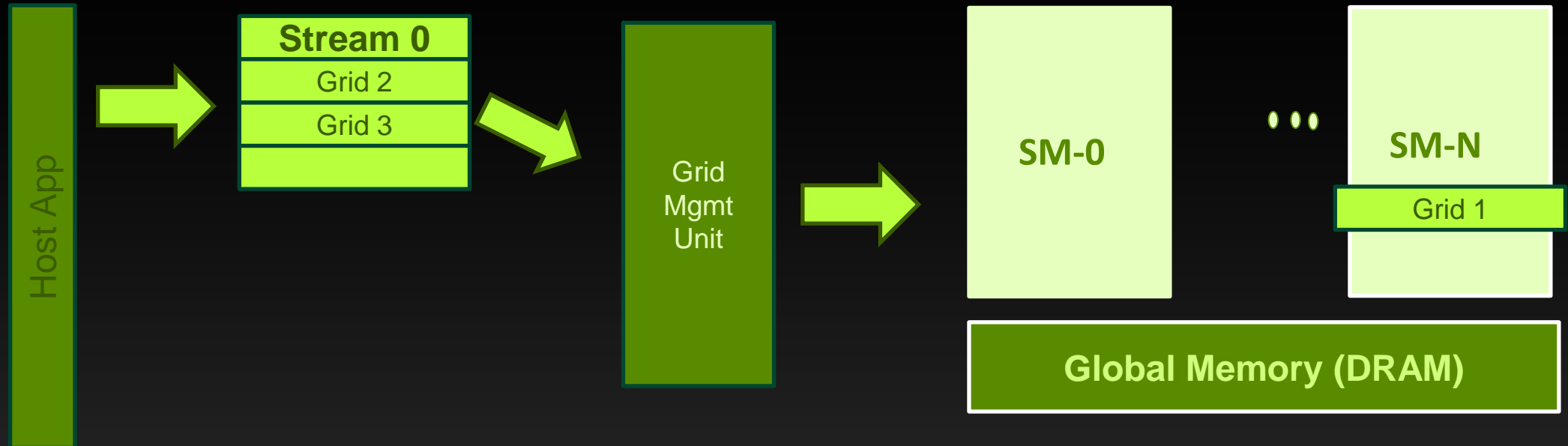
# Streams



## Kernel launches

- within the same stream are in-order

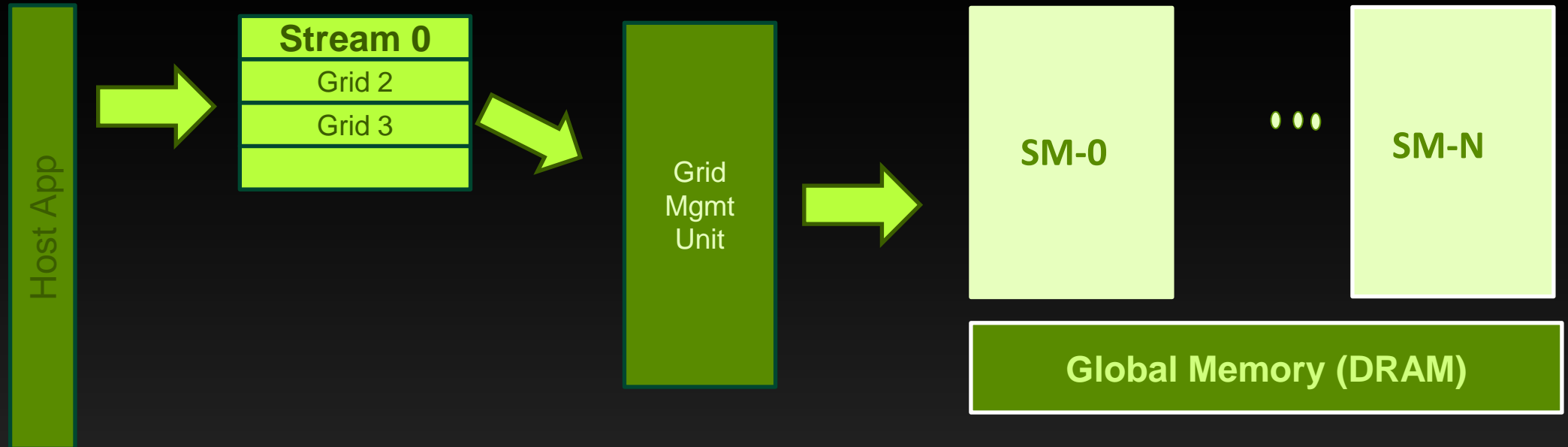
# Streams



## Kernel launches

- within the same stream are in-order

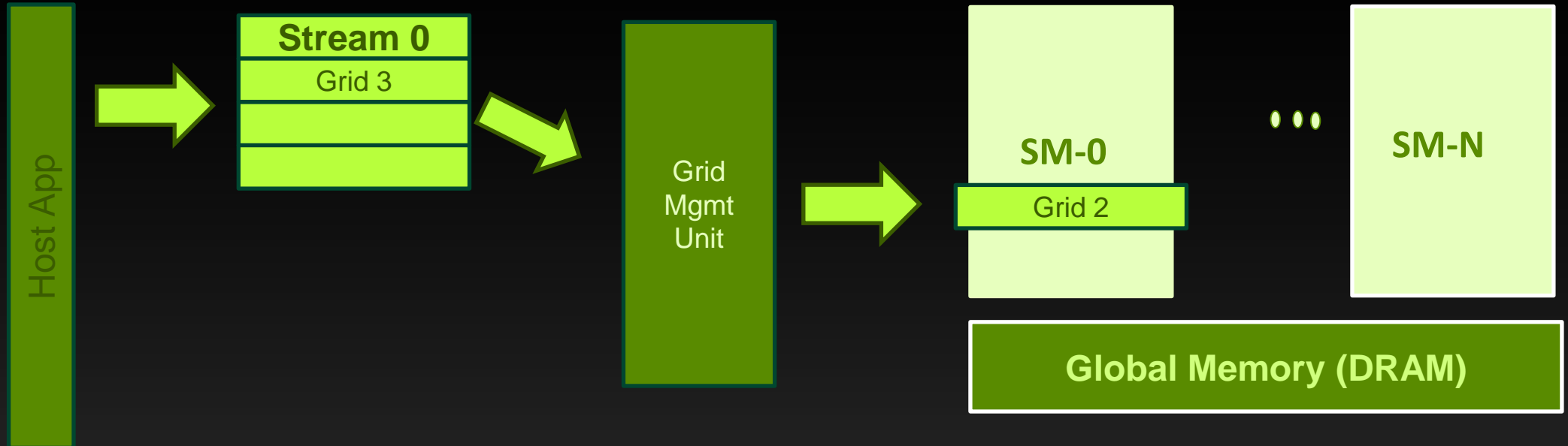
# Streams



## Kernel launches

- within the same stream are in-order

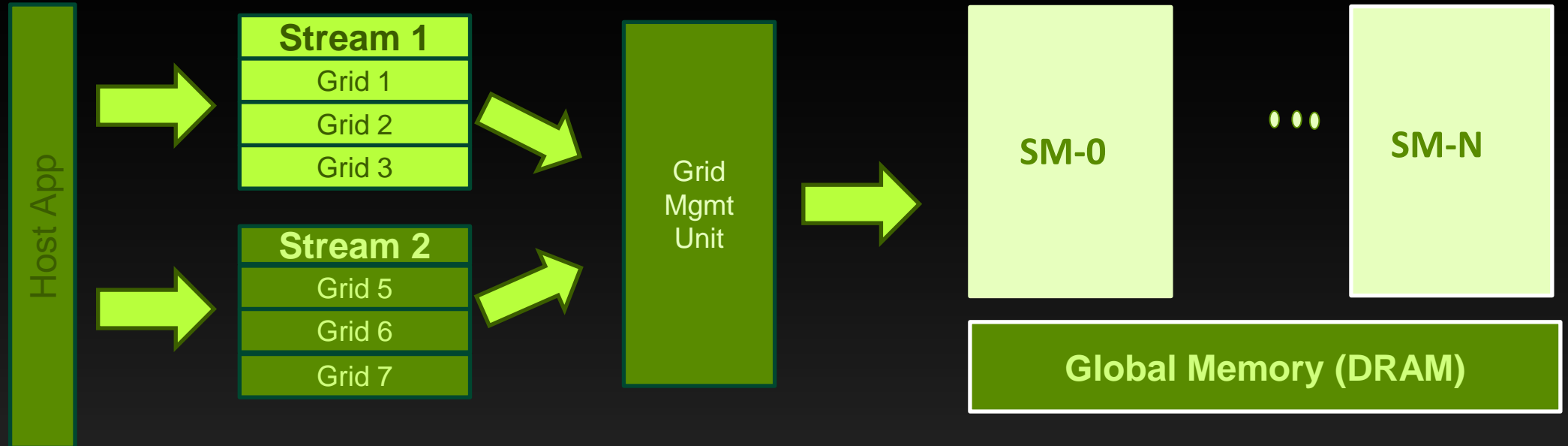
# Streams



## Kernel launches

- within the same stream are in-order

# Streams



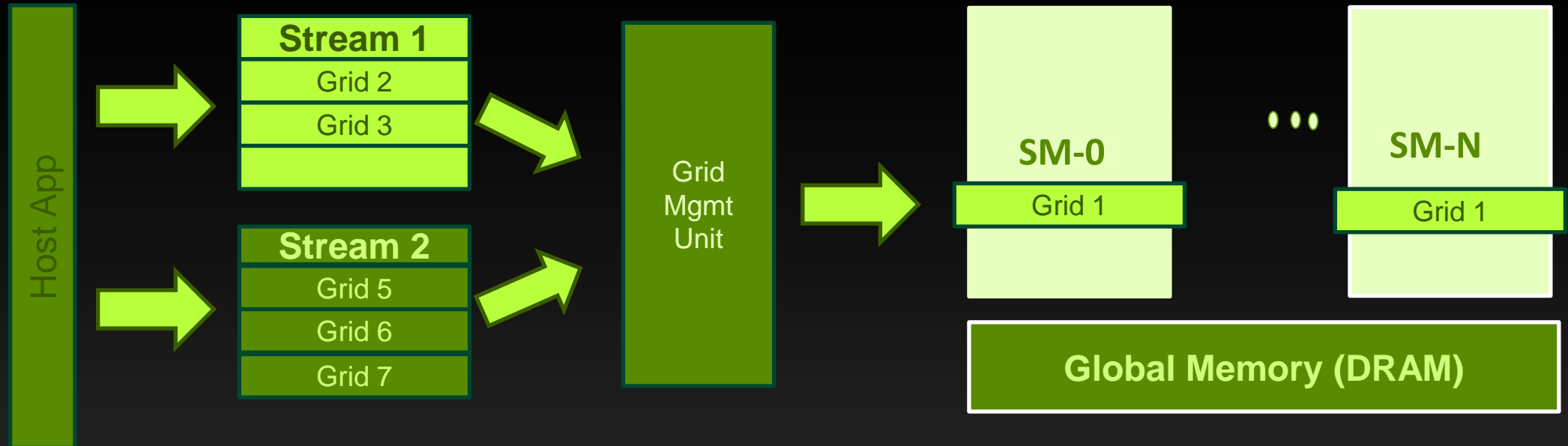
## Kernel launches

- within the same stream are in-order
- In different streams can be concurrent

All kernel launches are asynchronous to the host



# Streams

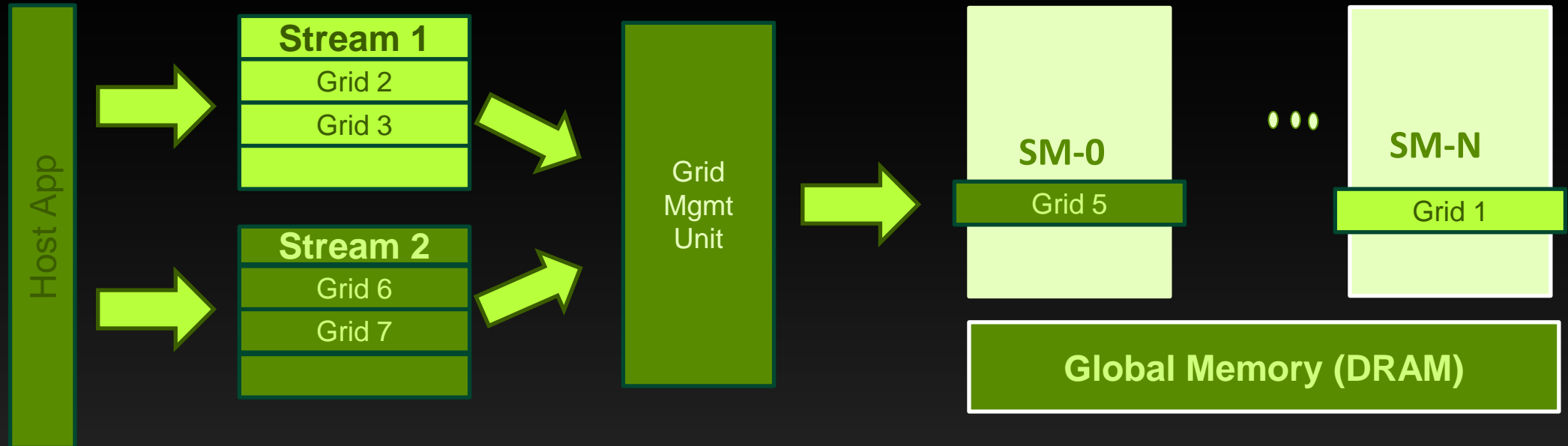


## Kernel launches

- within the same stream are in-order
- In different streams can be concurrent

All kernel launches are asynchronous to the host

# Streams

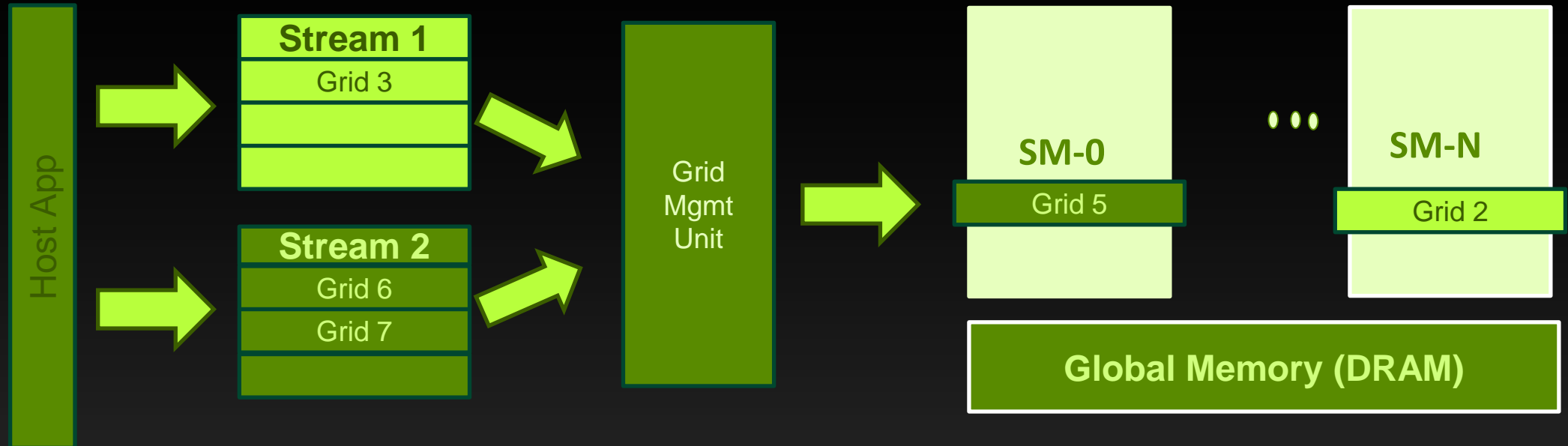


## Kernel launches

- within the same stream are in-order
- In different streams can be concurrent

All kernel launches are asynchronous to the host

# Streams

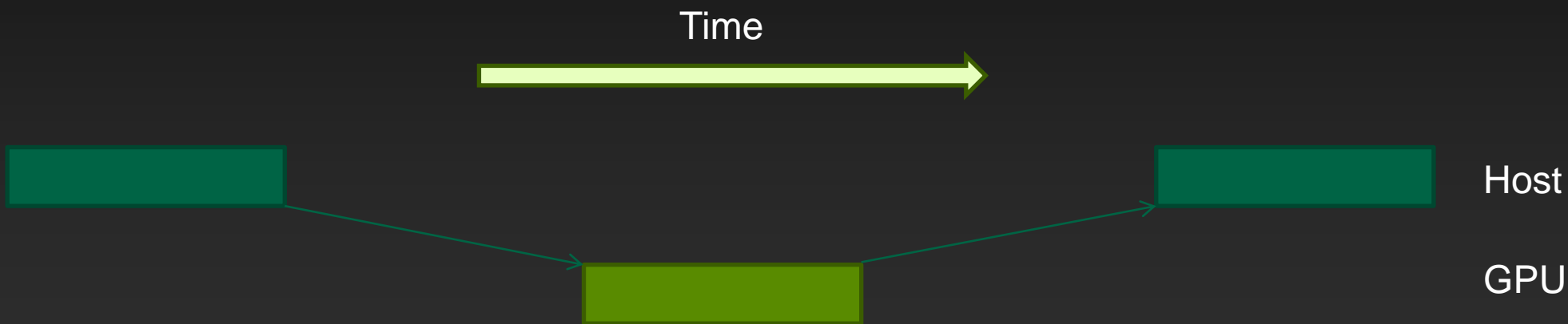


## Kernel launches

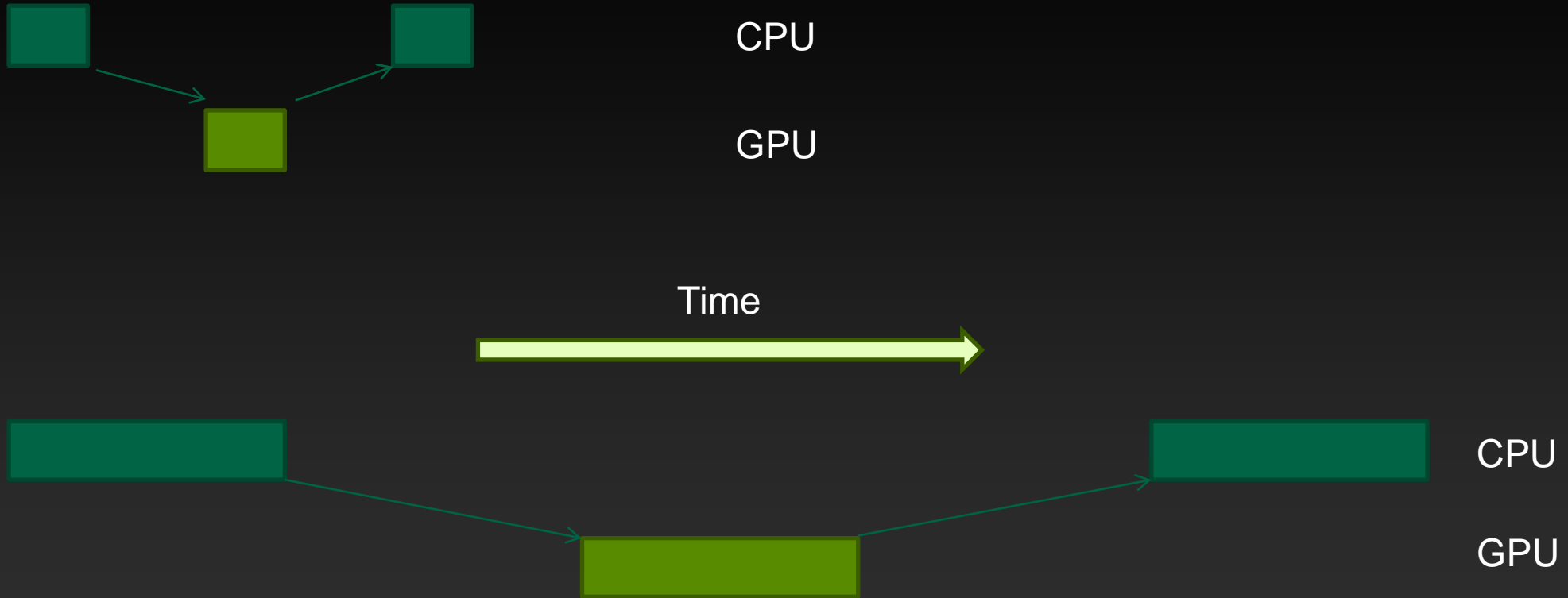
- within the same stream are in-order
- In different streams can be concurrent

All kernel launches are asynchronous to the host

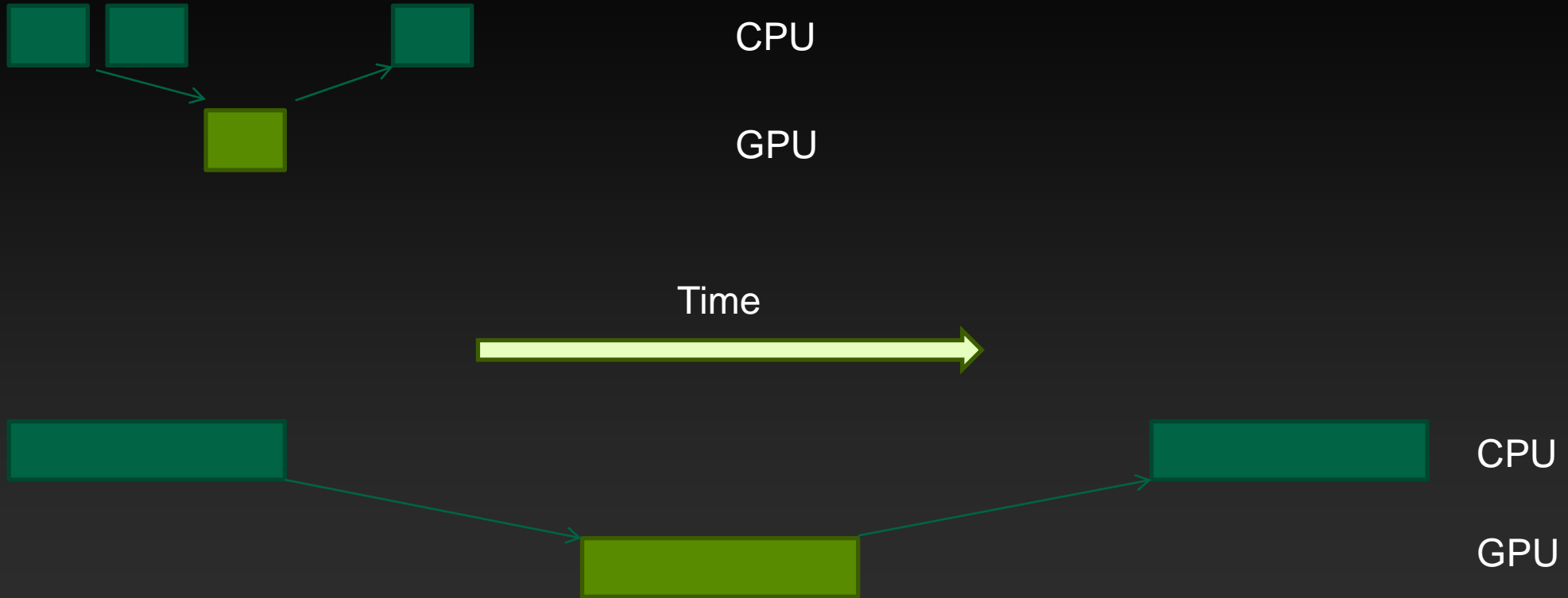
# Asynchronous Data Transfer / Pipelining



# Asynchronous Data Transfer / Pipelining

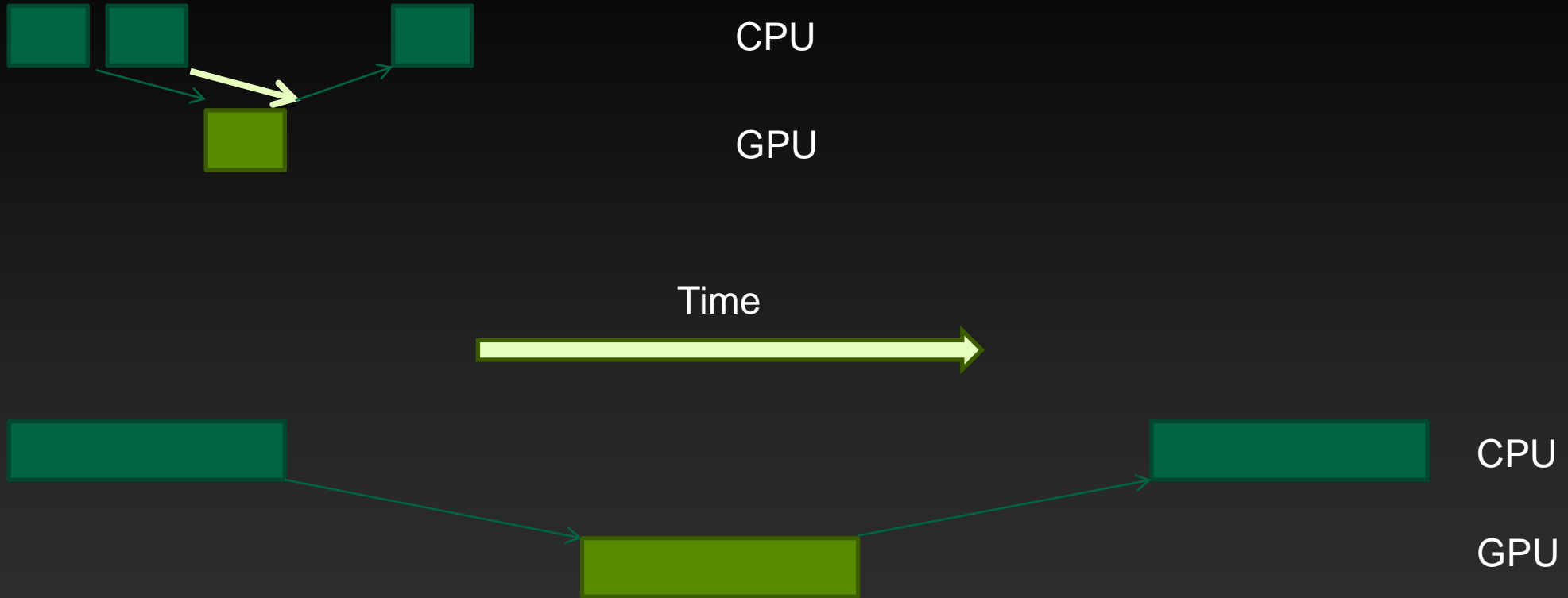


# Asynchronous Data Transfer / Pipelining

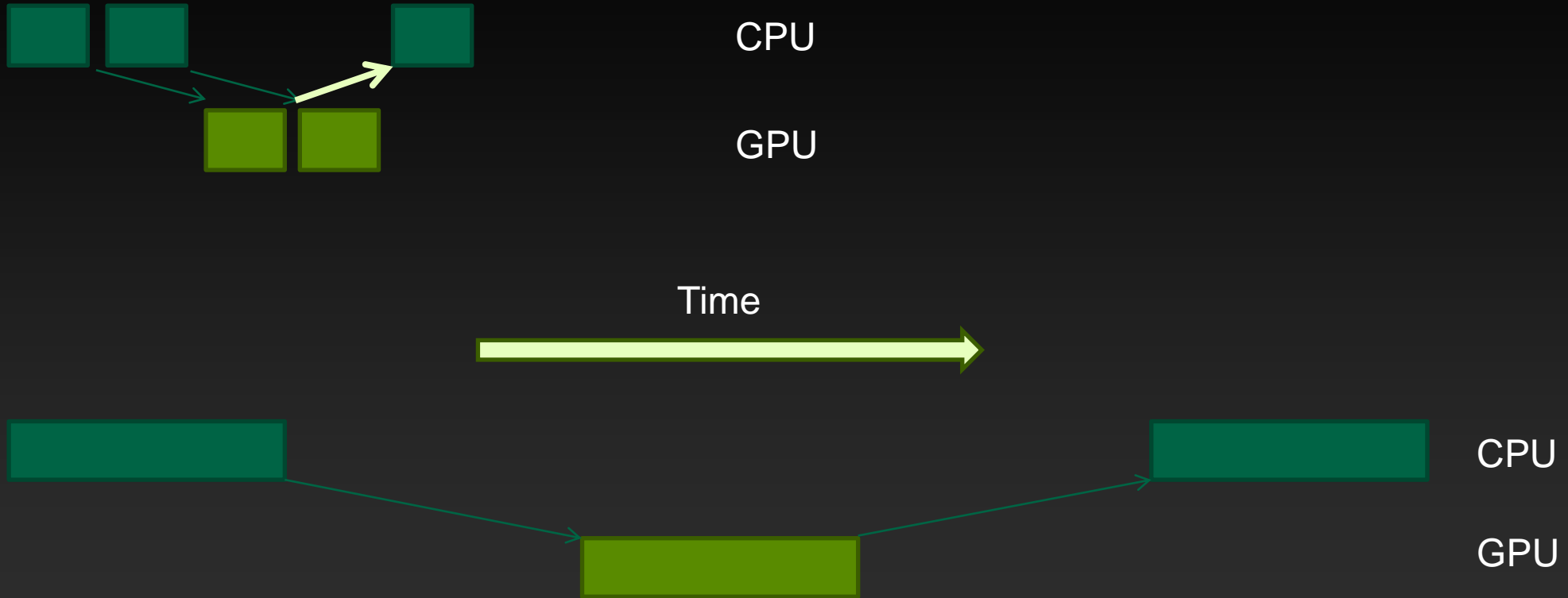




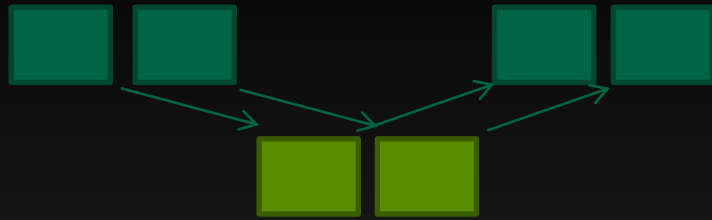
# Asynchronous Data Transfer / Pipelining



# Asynchronous Data Transfer / Pipelining



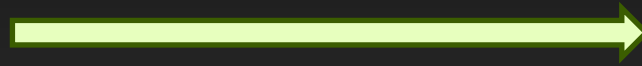
# Asynchronous Data Transfer / Pipelining



CPU

GPU

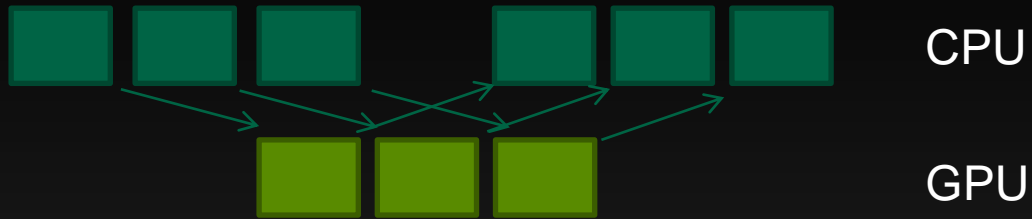
Time



CPU

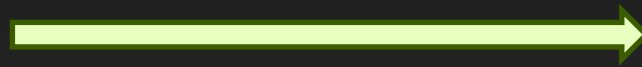
GPU

# Asynchronous Data Transfer / Pipelining

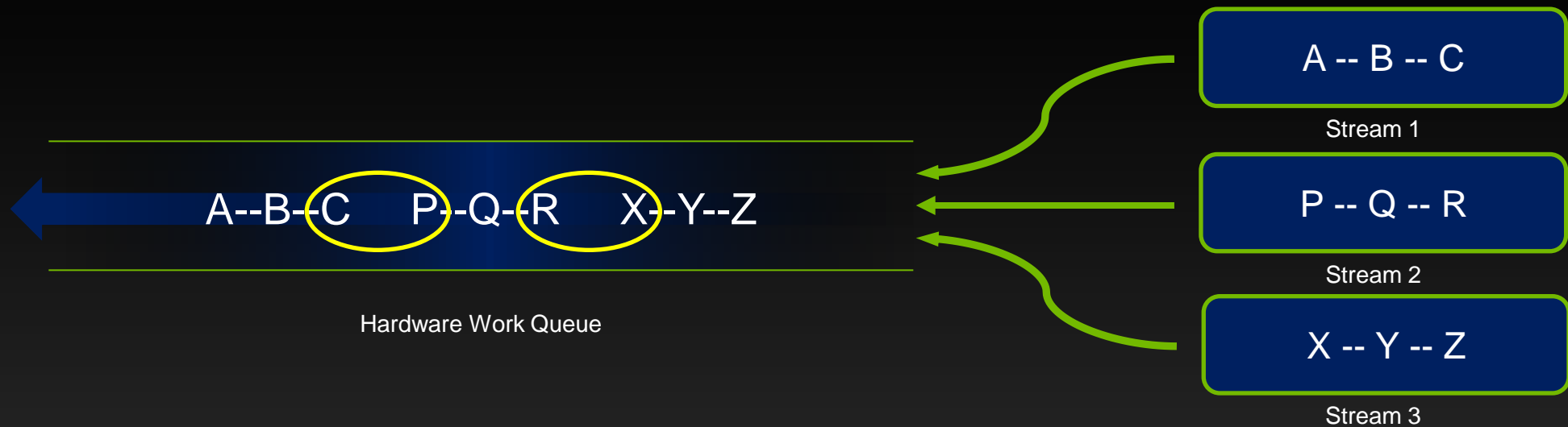


```
cudaStream_t    stream1, stream2;  
  
cudaMemcpyAsync( dst, src, size,  
                dir, stream1 );  
  
kernel<<<grid, block, 0, stream2>>>(...);
```

Time



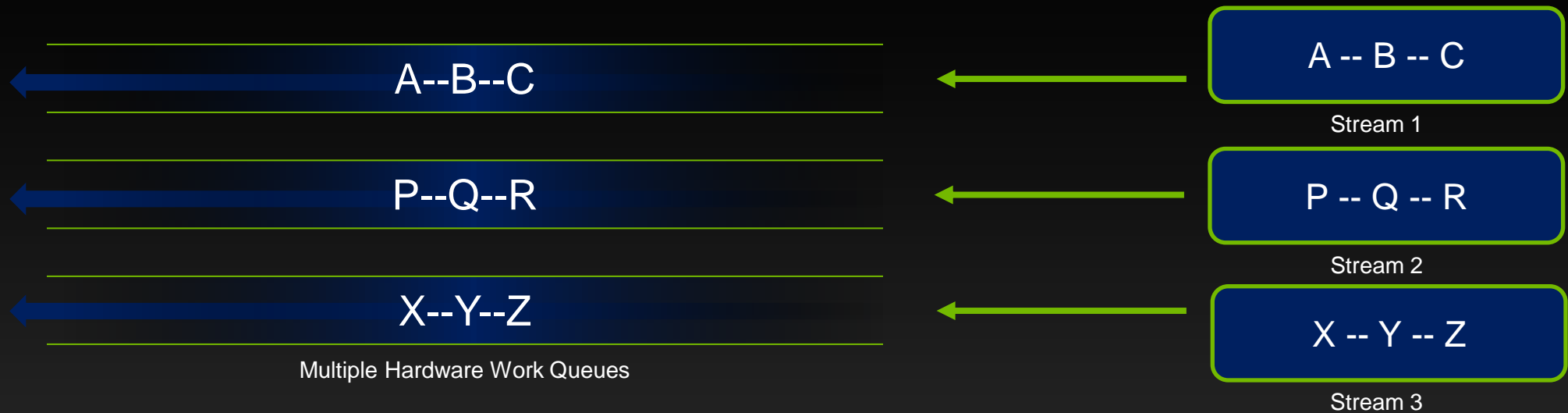
# Fermi Concurrency



## Fermi allows 16-way concurrency

- Up to 16 grids can run at once
- But CUDA streams multiplex into a single queue
- Overlap only at stream edges

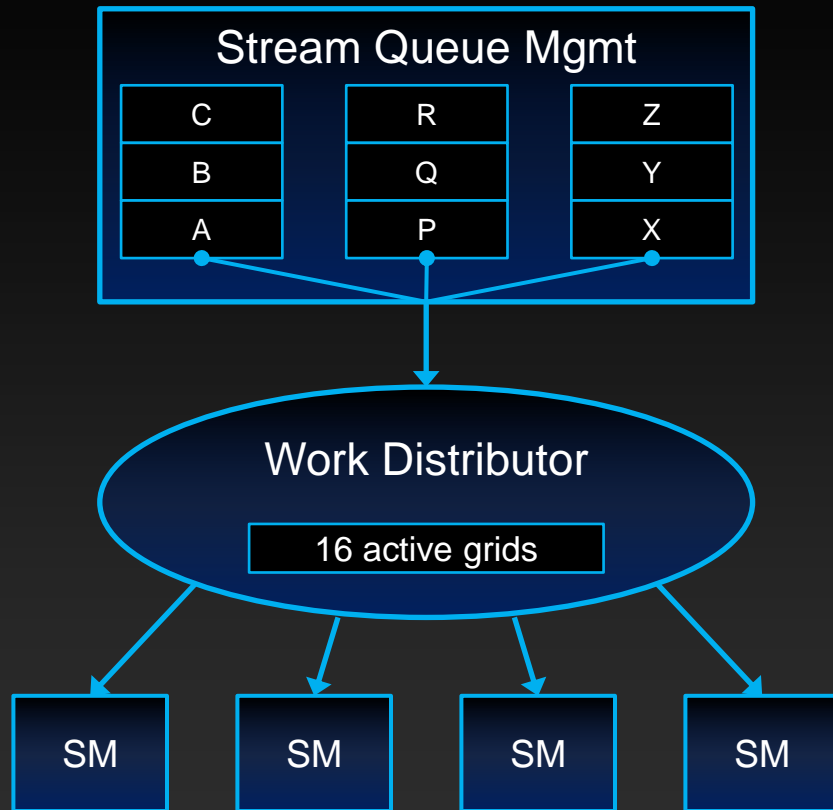
# Kepler Improved Concurrency



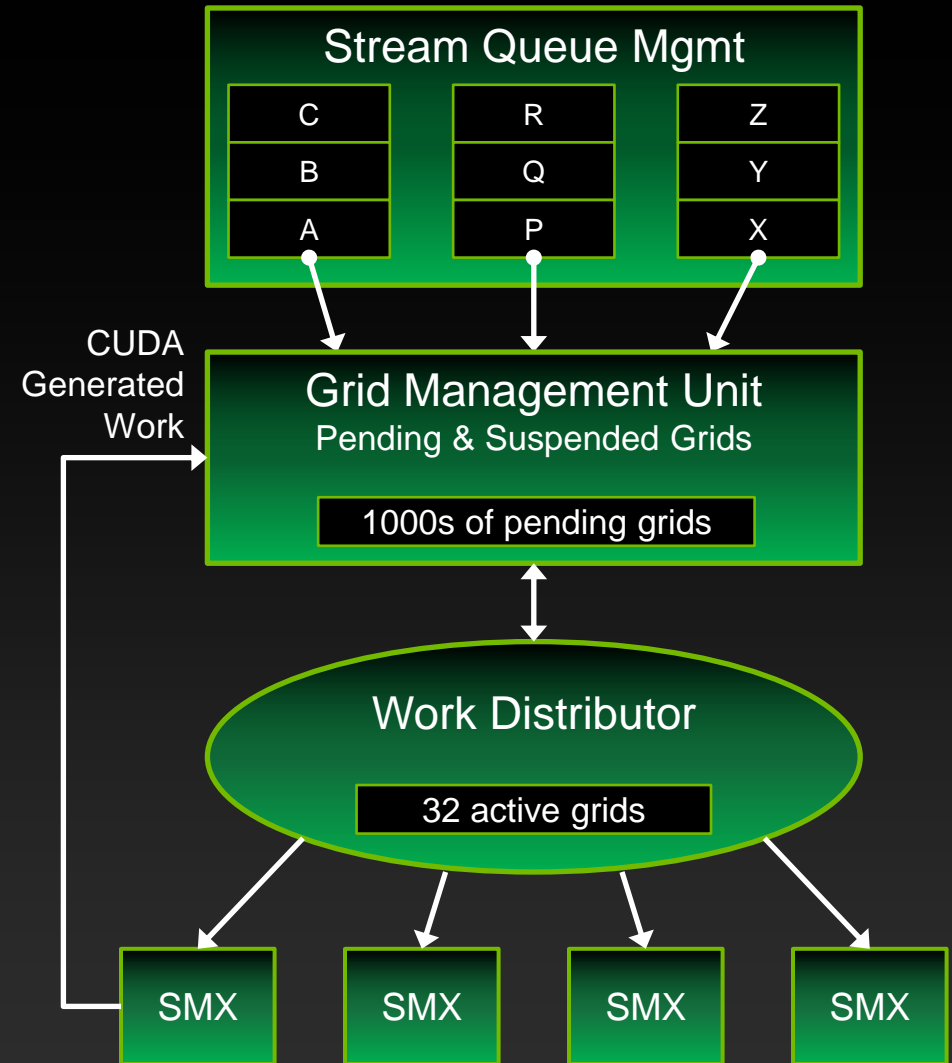
## Kepler allows 32-way concurrency

- One work queue per stream
- Concurrency at full-stream level
- No inter-stream dependencies

# Grid Management Unit



Fermi



Kepler GK110

# Hyper-Q Enables Efficient Scheduling

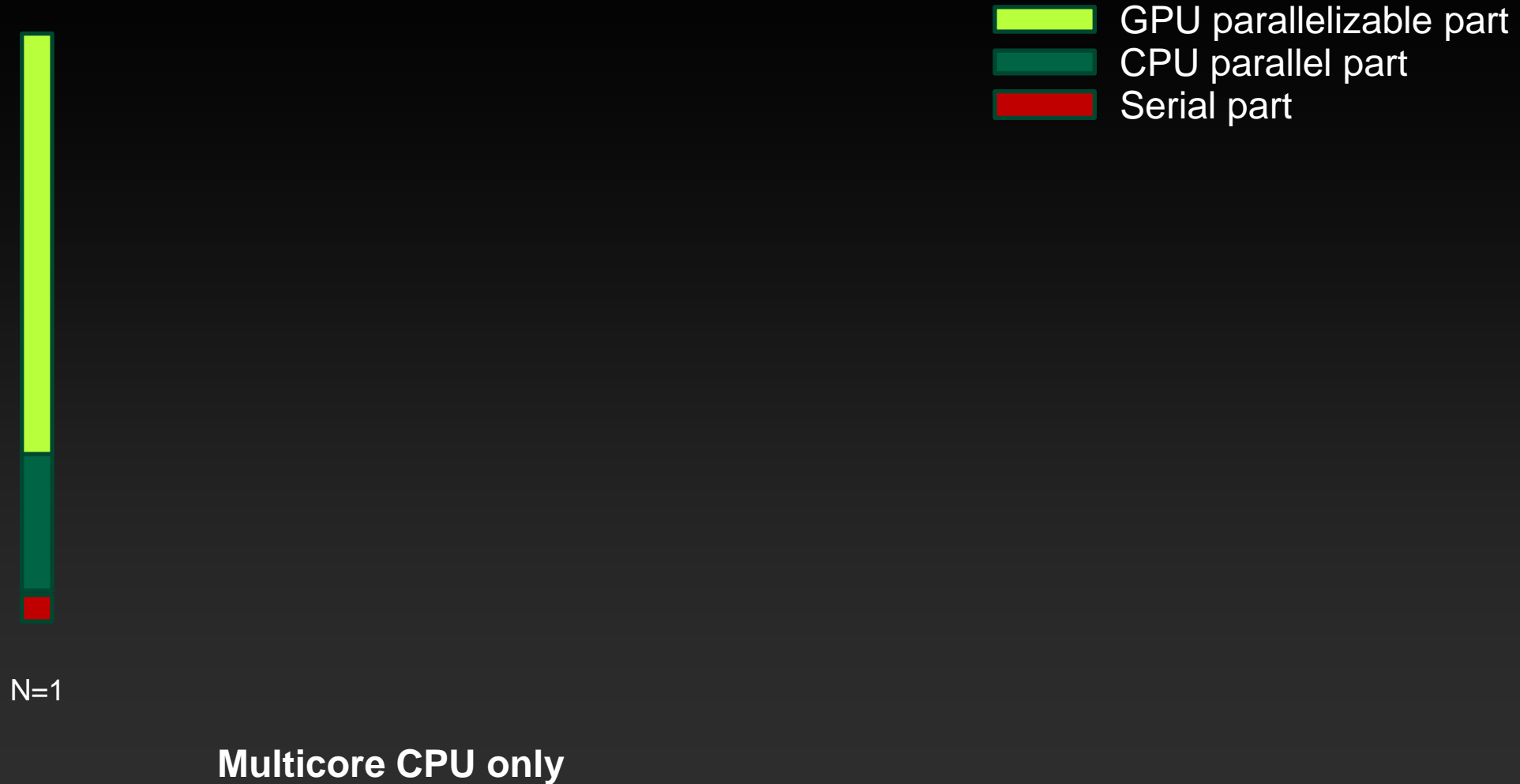
Grid management unit can select most appropriate grid from 32 streams

Improves scheduling of concurrently executed grids

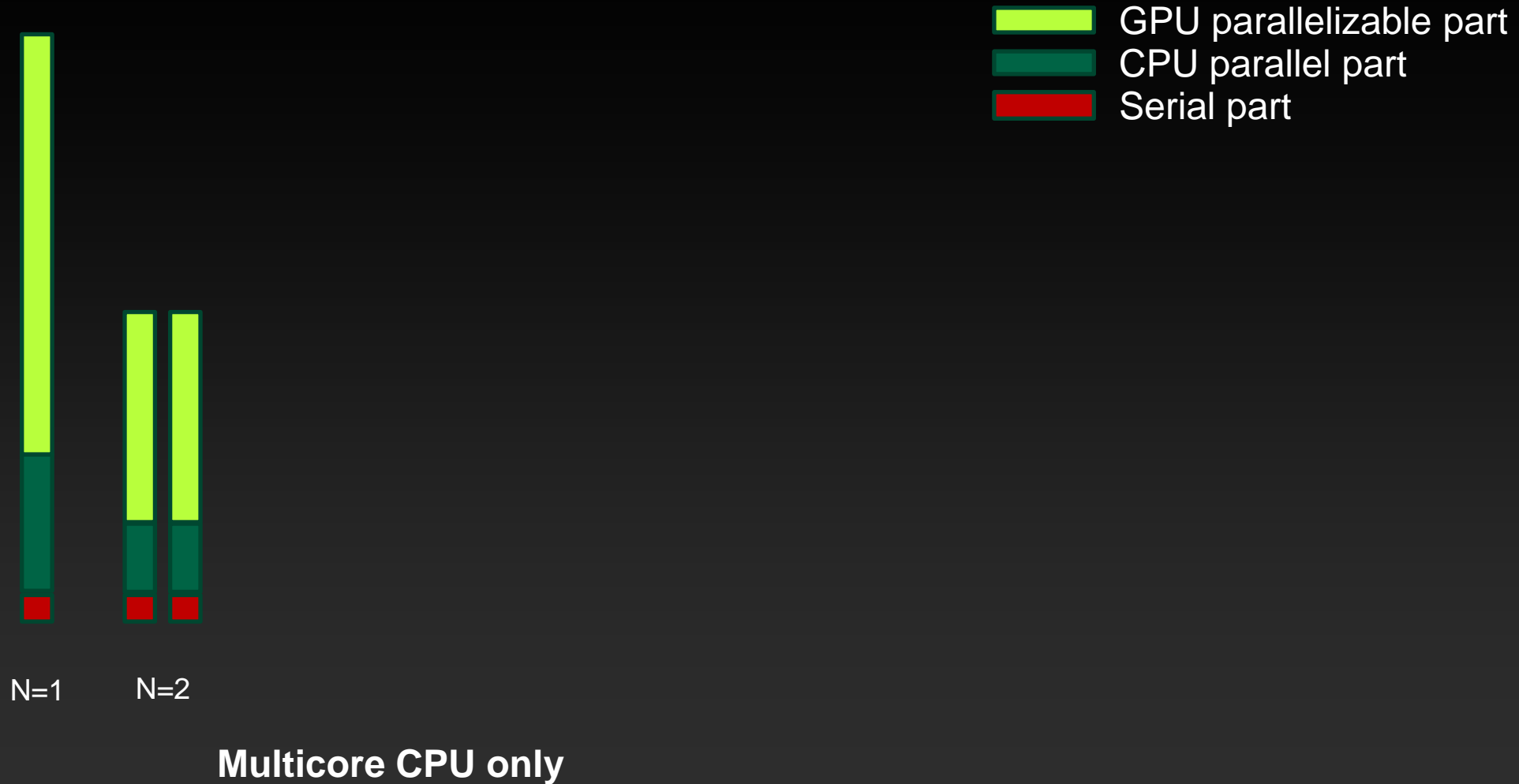
Particularly interesting for MPI applications



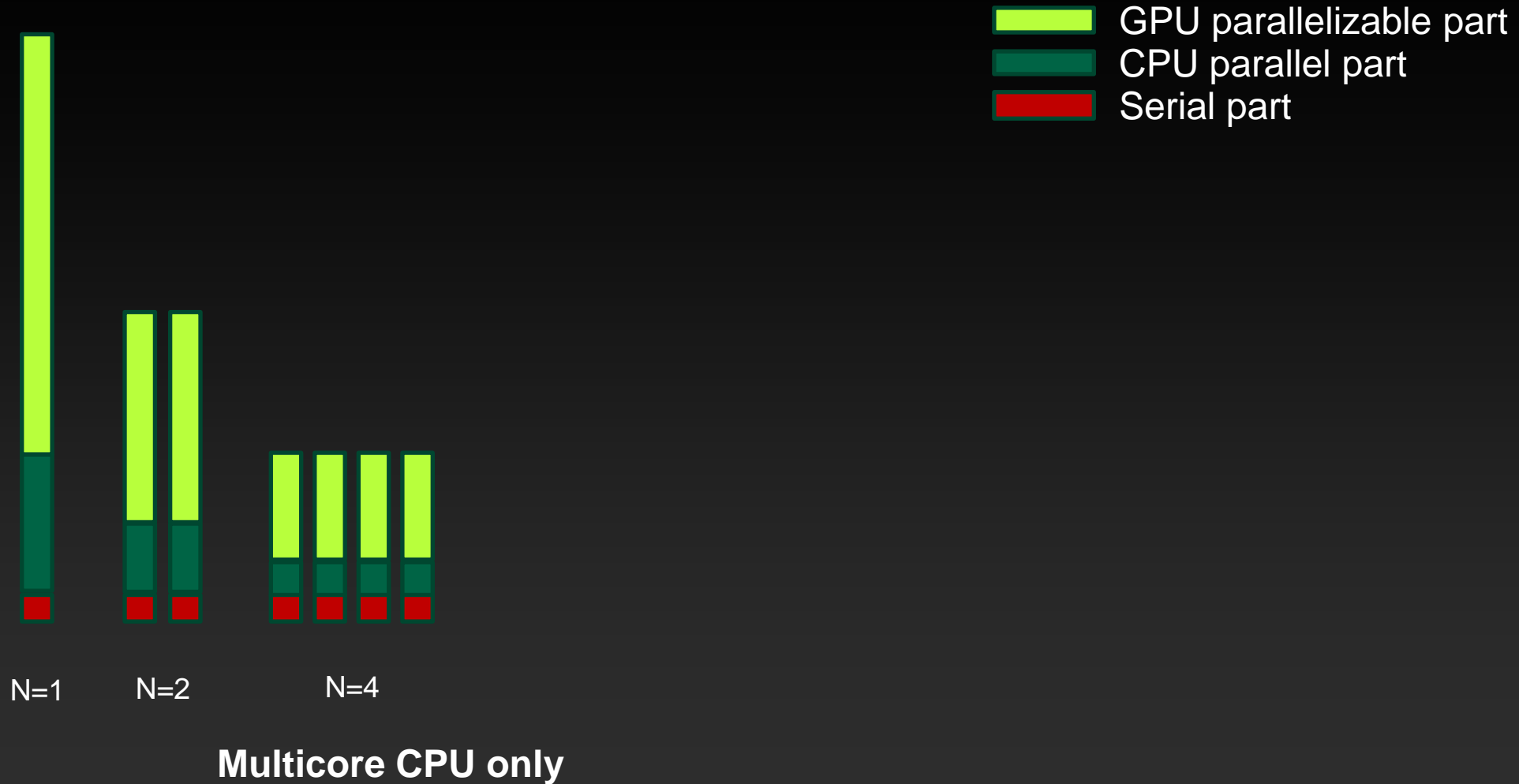
# Strong Scaling of MPI Application



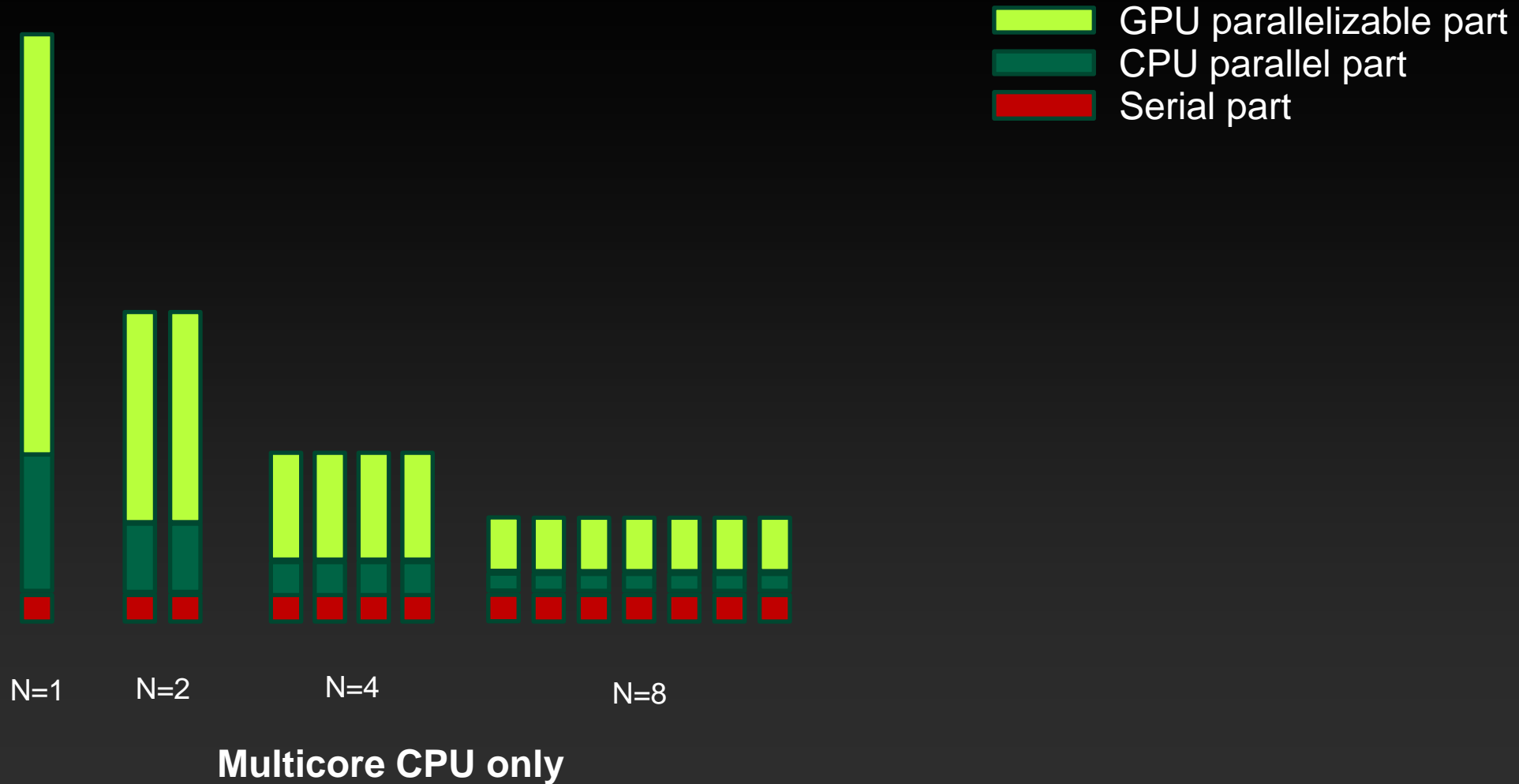
# Strong Scaling of MPI Application



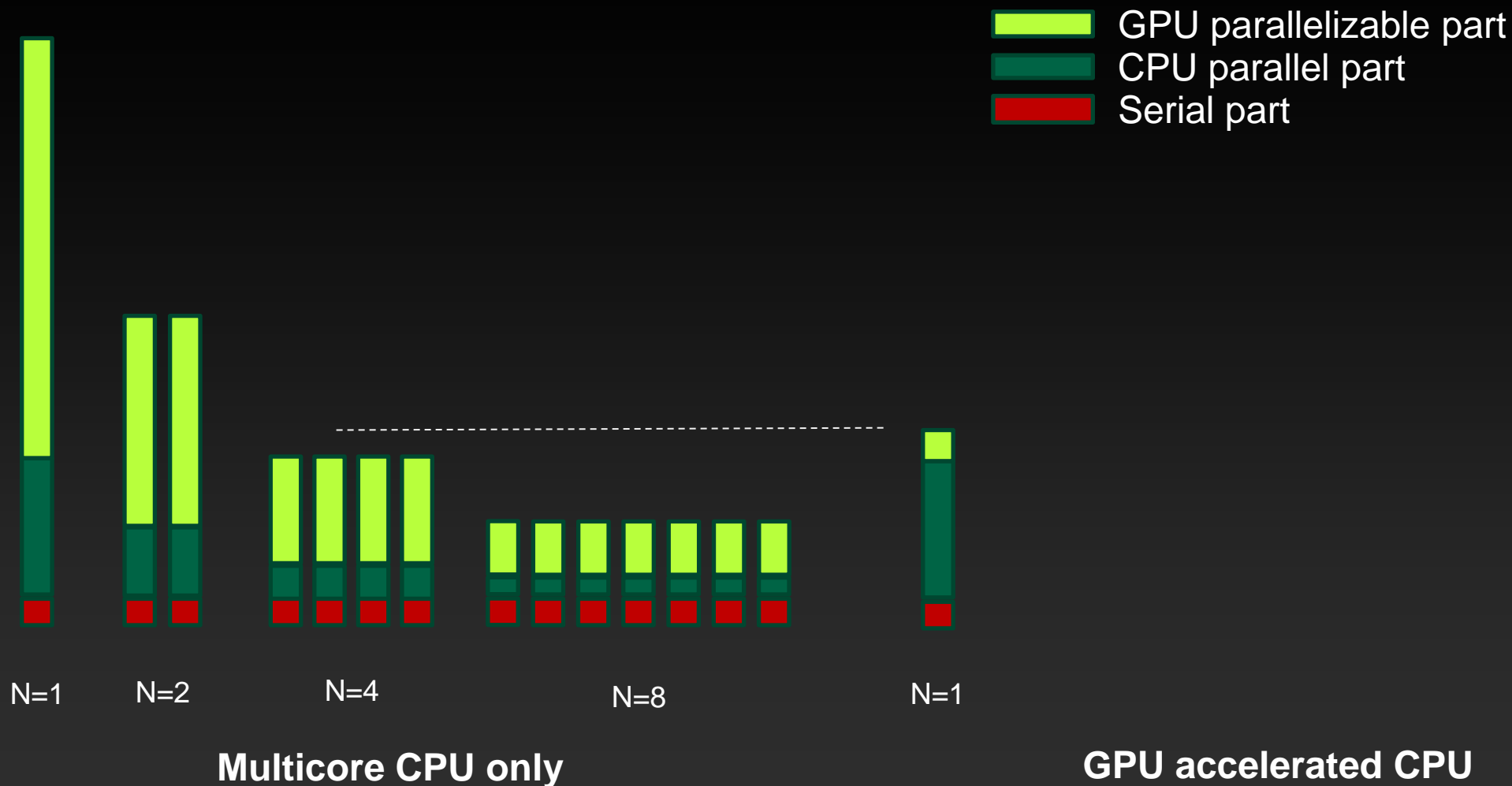
# Strong Scaling of MPI Application



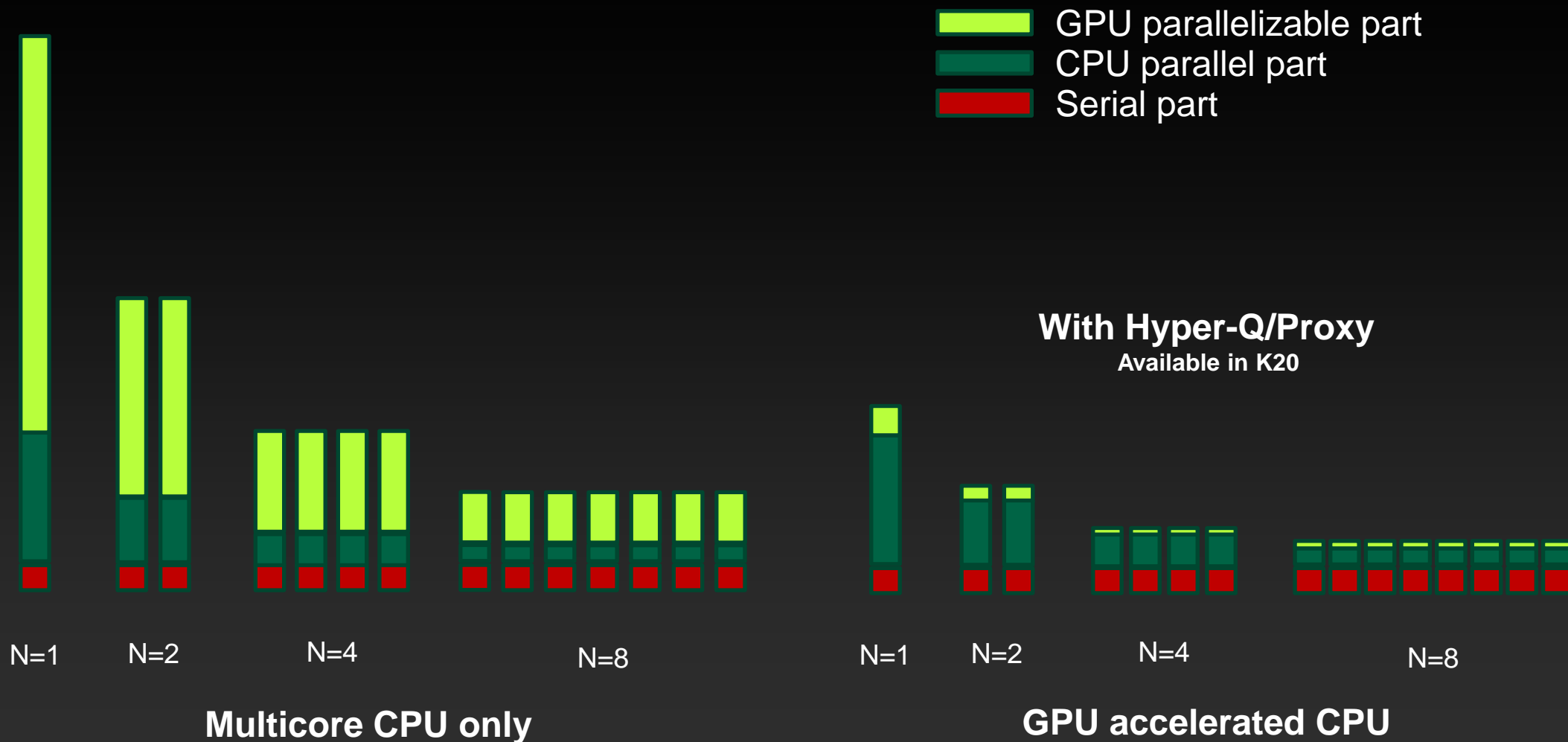
# Strong Scaling of MPI Application



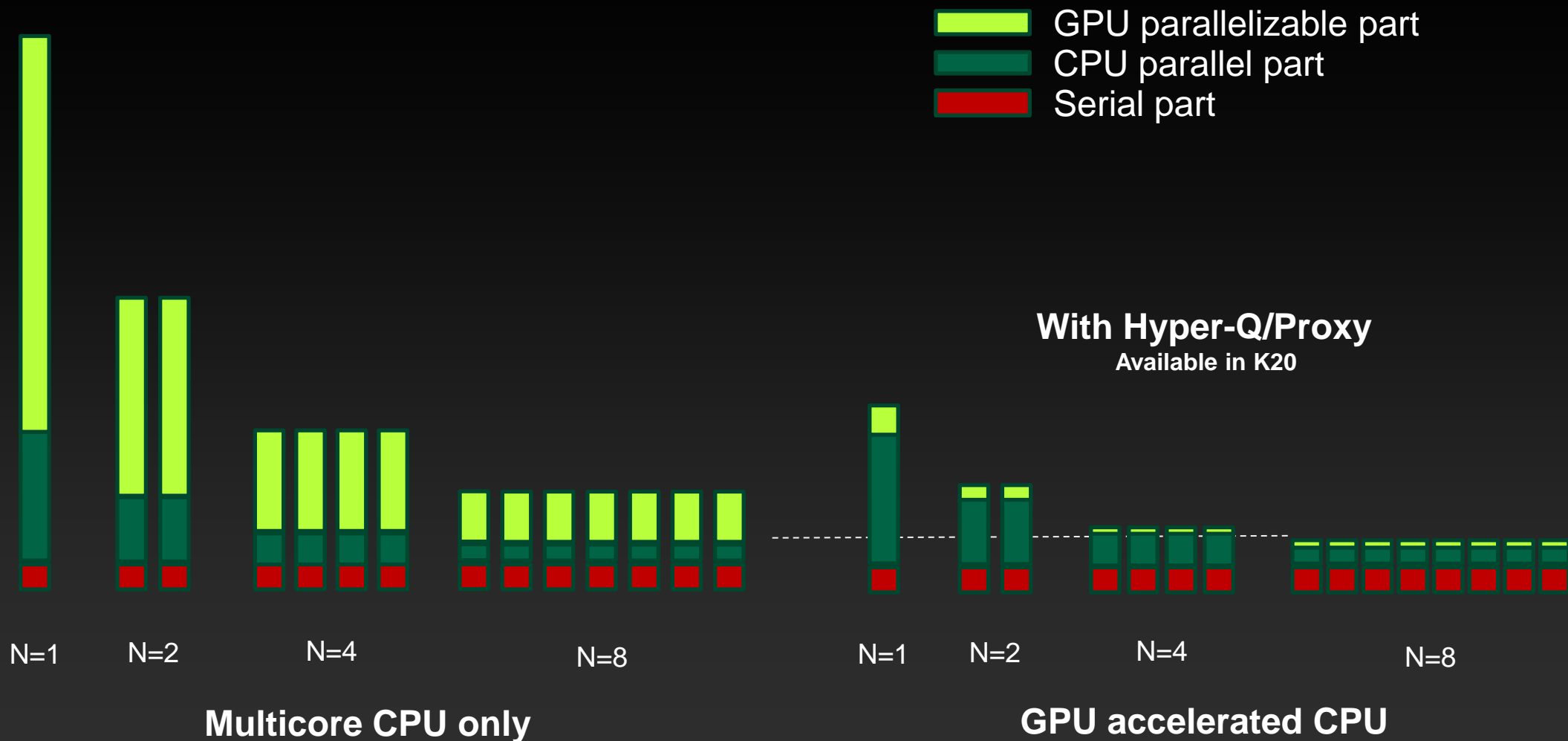
# GPU Accelerated MPI Application



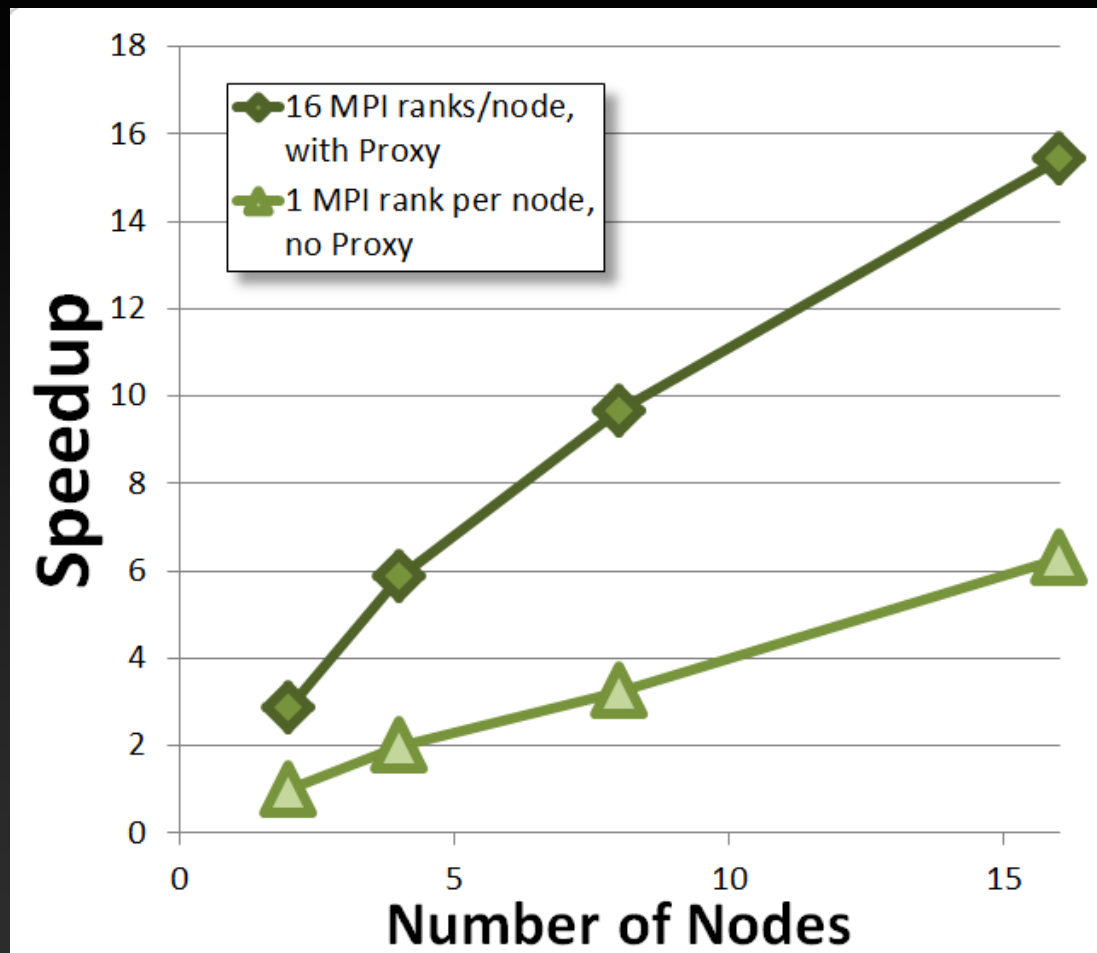
# GPU Accelerated Strong Scaling



# GPU Accelerated Strong Scaling



# Example: Hyper-Q/Proxy for CP2K



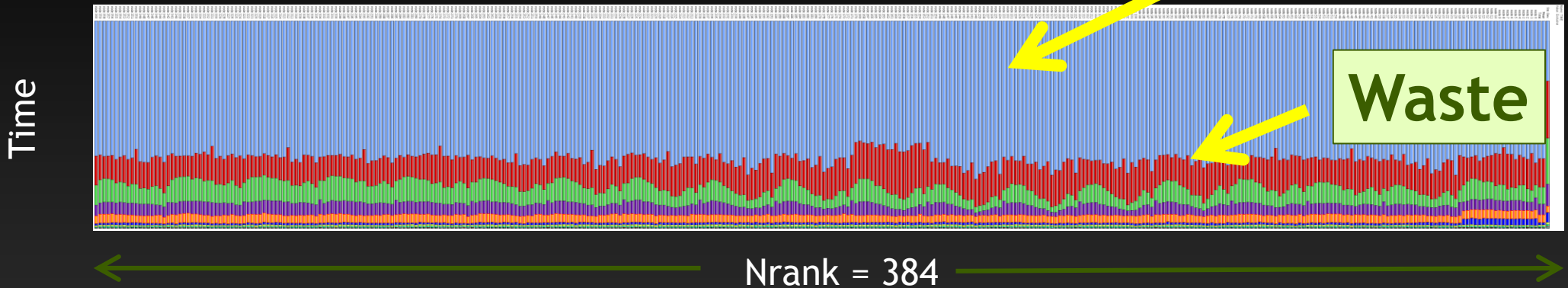


# How to use Hyper-Q

- No application modifications necessary
- Proxy process between user processes and GPU
- Enabled via environment variable  
`export CRAY_CUDA_PROXY=1`

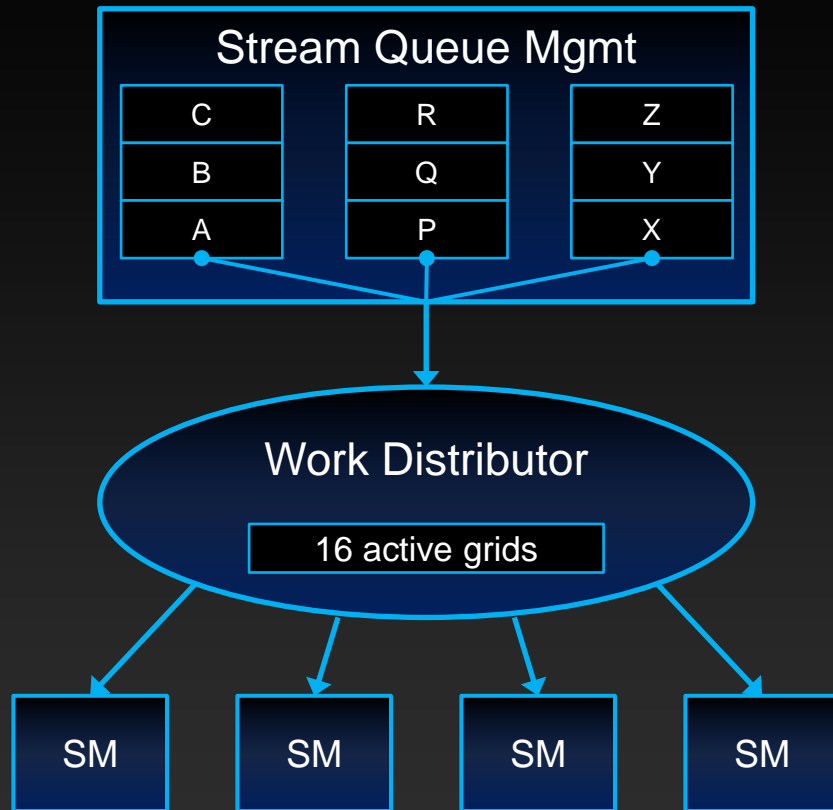
# Don't Forget Large-Scale Behavior

- Profile in realistic environment
  - Get profile at scale
  - Tau, Scalasca, VampirTrace+Vampir, Craypat, ..

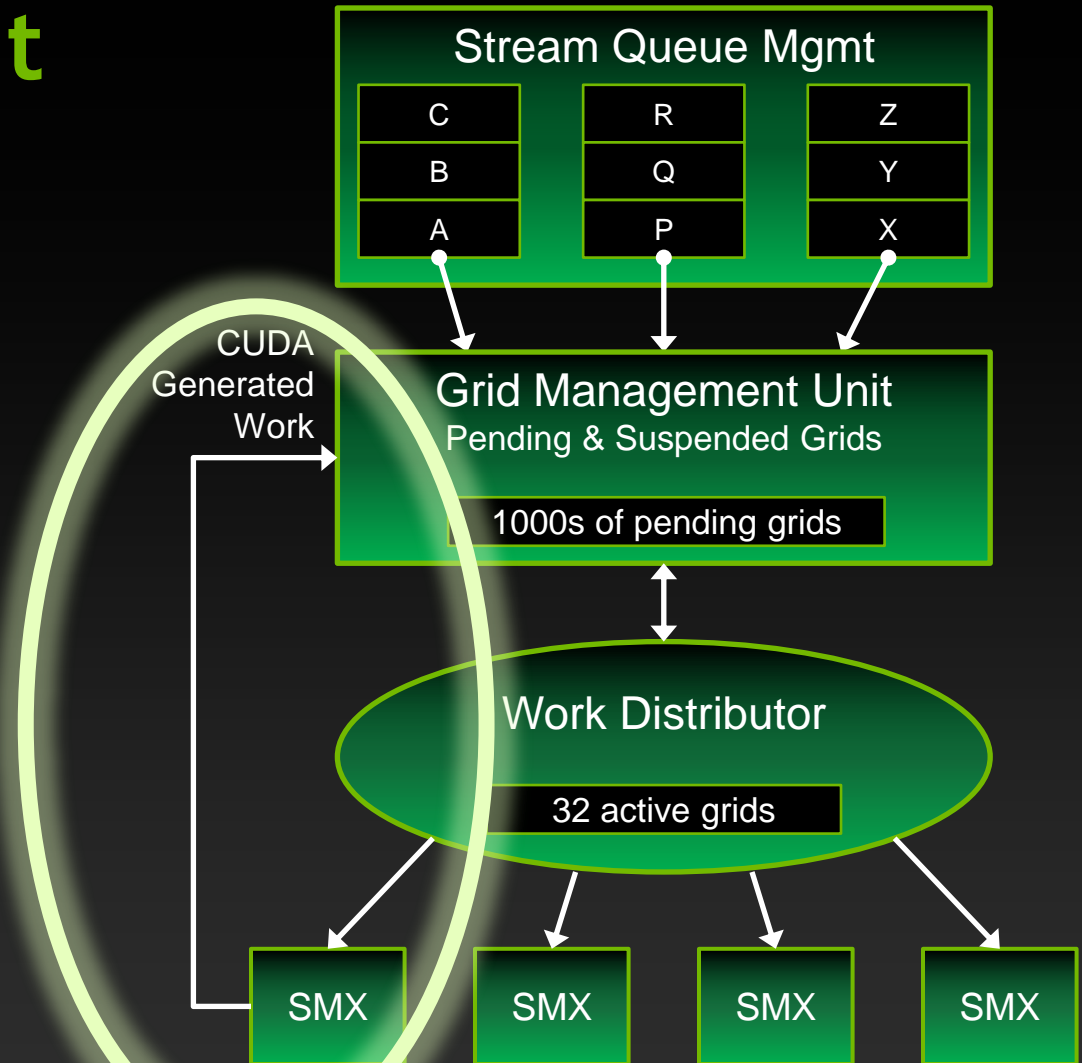


- Fix messaging problems first!
  - GPUs will accelerate your compute, amplify messaging problems
  - Will also help CPU-only code

# Grid Management Unit

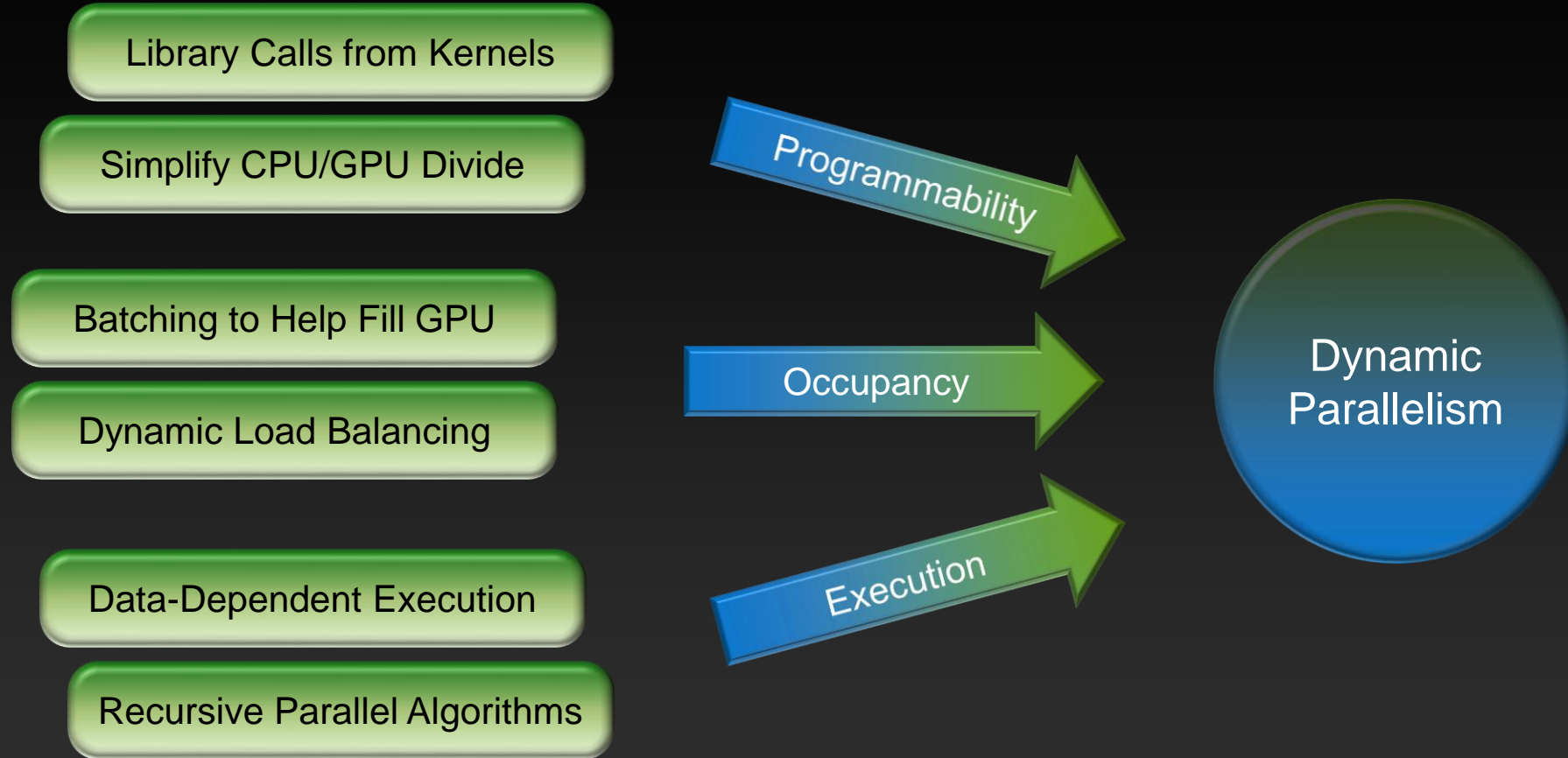


Fermi



Kepler GK110

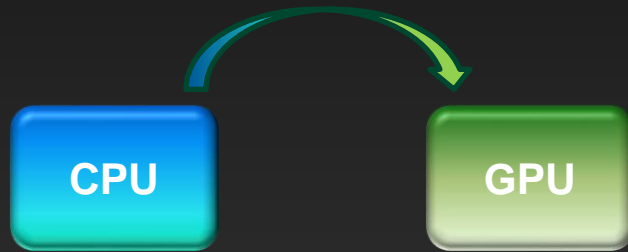
# Improving Programmability



# What is Dynamic Parallelism?

## The ability to launch new grids from the GPU

- Dynamically
- Simultaneously
- Independently

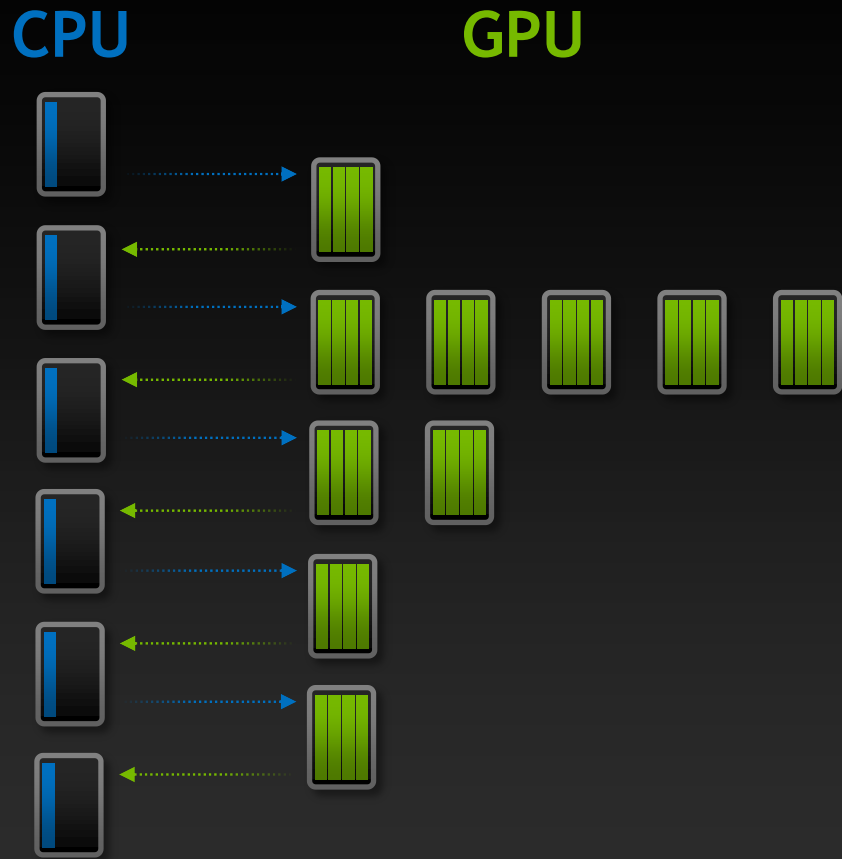


*Fermi: Only CPU can generate GPU work*

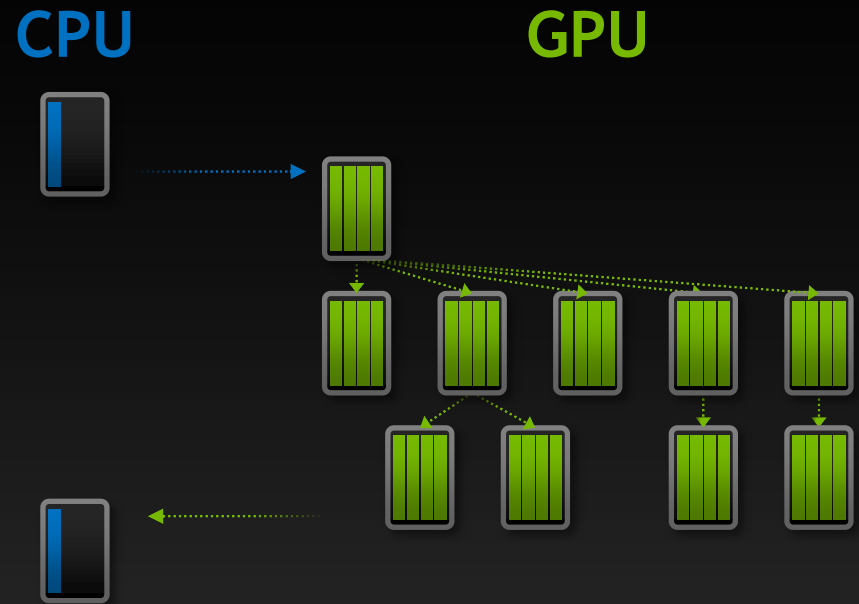


*Kepler: GPU can generate work for itself*

# What Does It Mean?



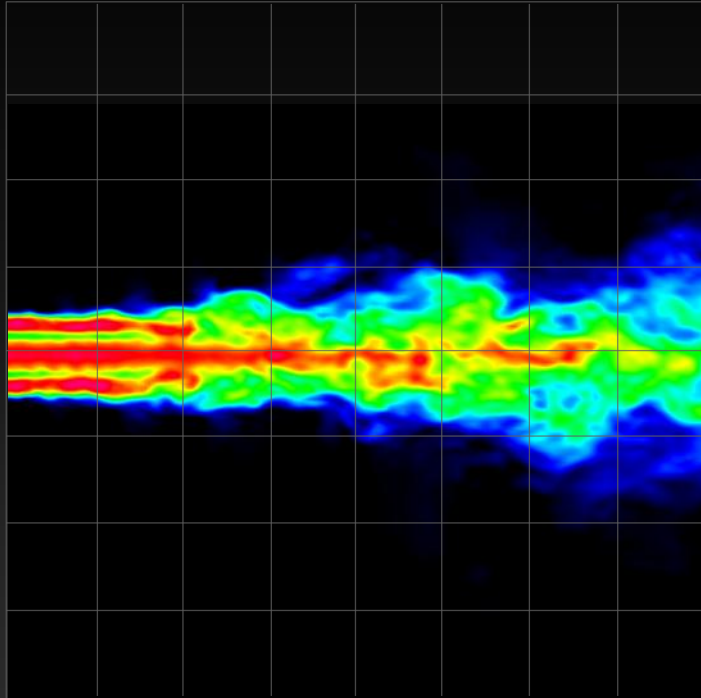
*GPU as Co-Processor*



*Autonomous, Dynamic Parallelism*

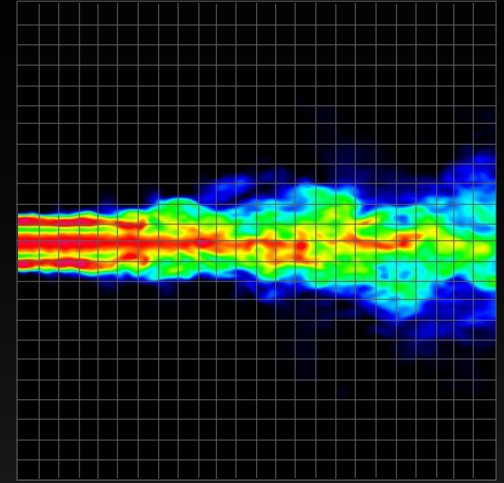
# Dynamic Work Generation

*Fixed Grid*

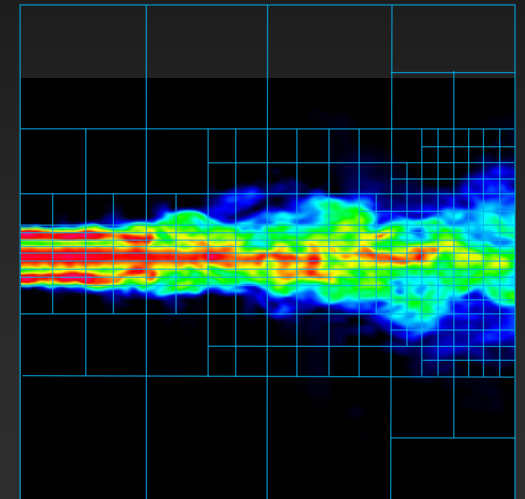


*Initial Grid*

*Statically assign conservative  
worst-case grid*



*Dynamically assign performance  
where accuracy is required*

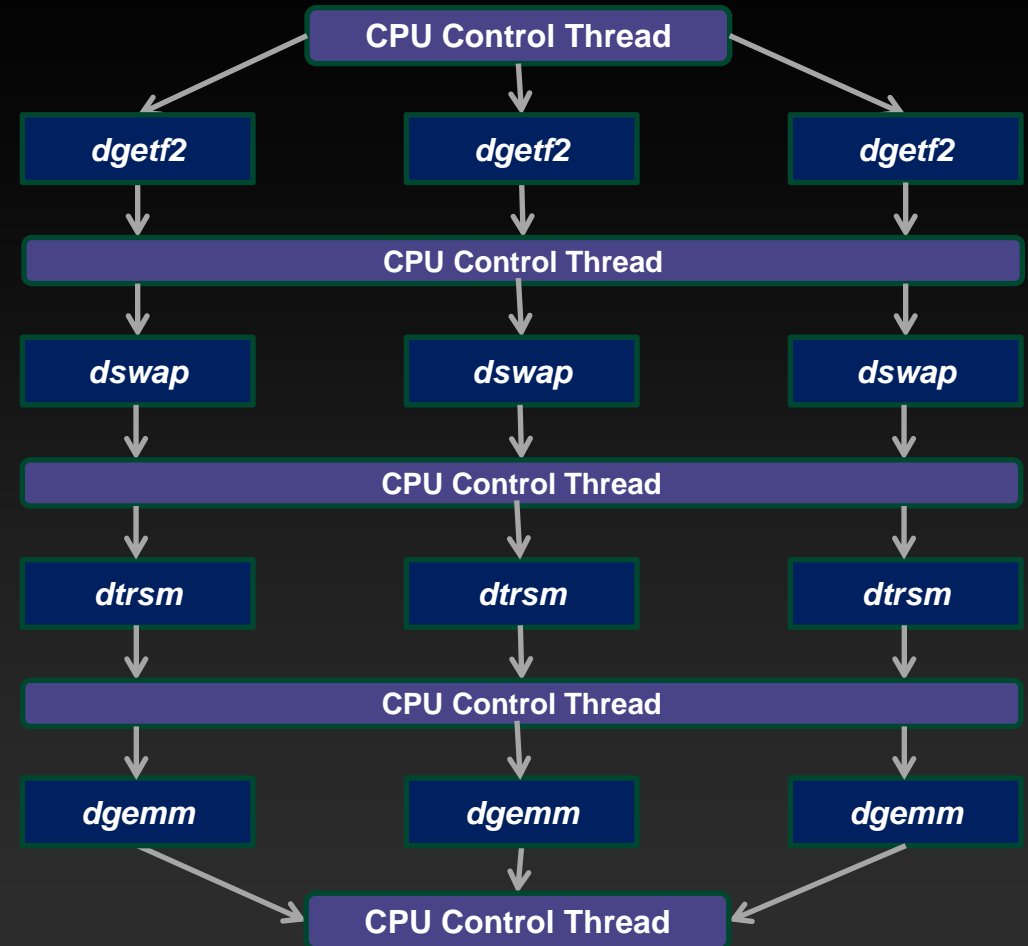


*Dynamic Grid*

# Batched & Nested Parallelism

## CPU-Controlled Work Batching

- CPU programs limited by single point of control
- Can run at most 10s of threads
- CPU is fully consumed with controlling launches



*Multiple LU-Decomposition, Pre-Kepler*

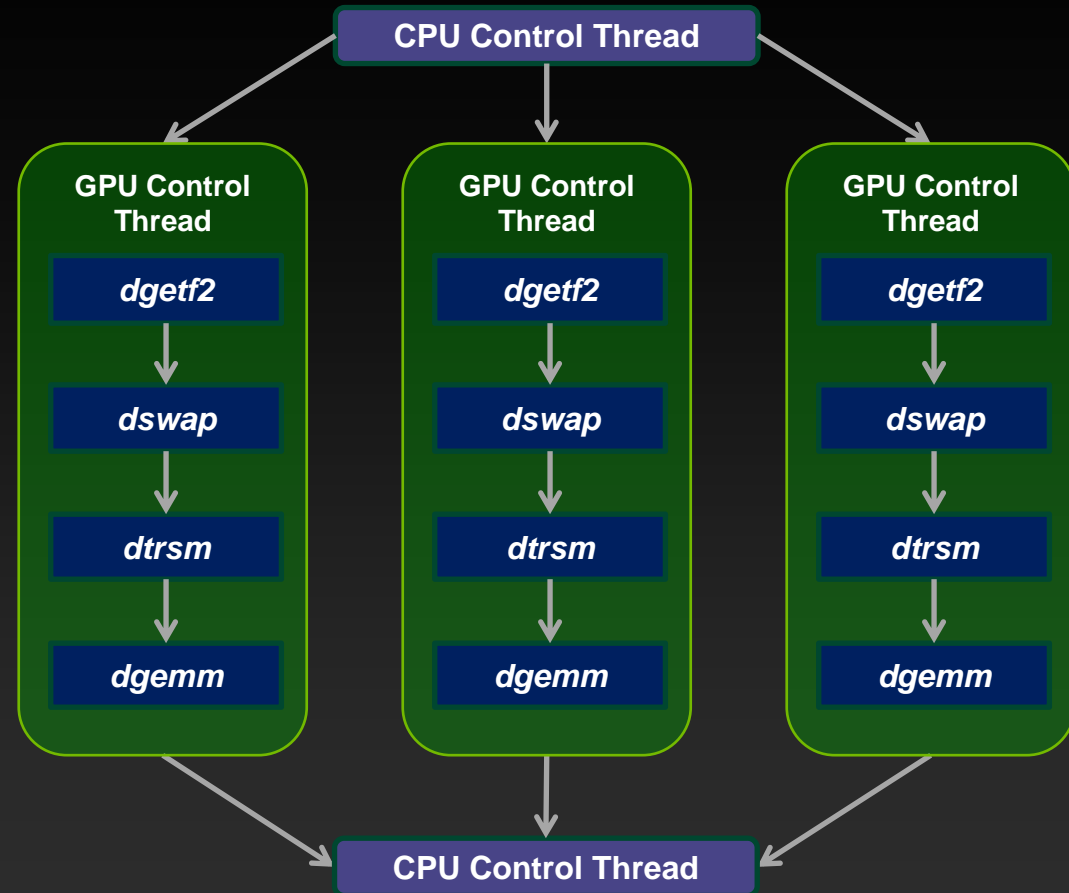
Algorithm flow simplified for illustrative purposes



# Batched & Nested Parallelism

## Batching via Dynamic Parallelism

- Move top-level loops to GPU
- Run thousands of independent tasks
- Release CPU for other work



*Batched LU-Decomposition, Kepler*

# CUDA Dynamic Parallelism

Kernel launches grids

Identical syntax as host

CUDA runtime function in  
cudadevrt library

Enabled via nvcc flag  
-rdc=true

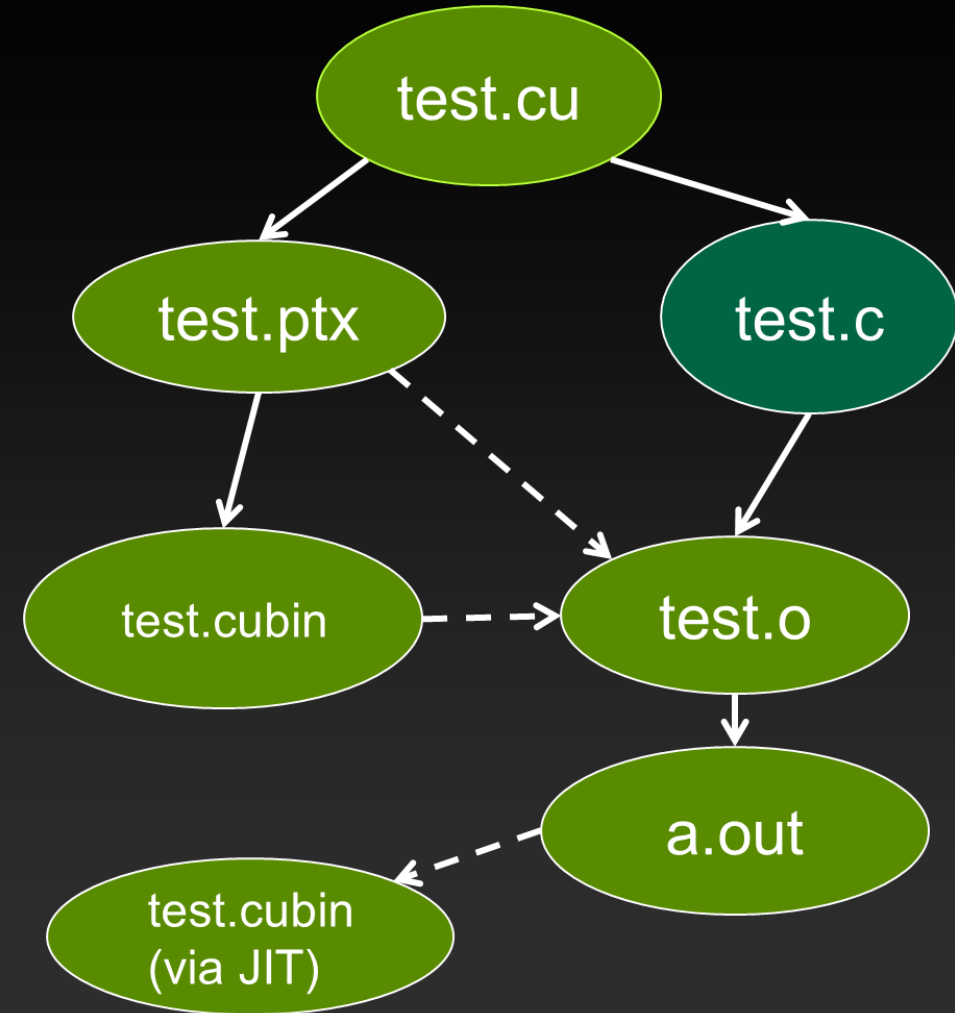
```
__global__ void childKernel()  
{  
    printf("Hello %d", threadIdx.x);  
}
```

```
__global__ void parentKernel()  
{  
    childKernel<<<1,10>>>();  
    cudaDeviceSynchronize();  
    printf("World!\n");  
}
```

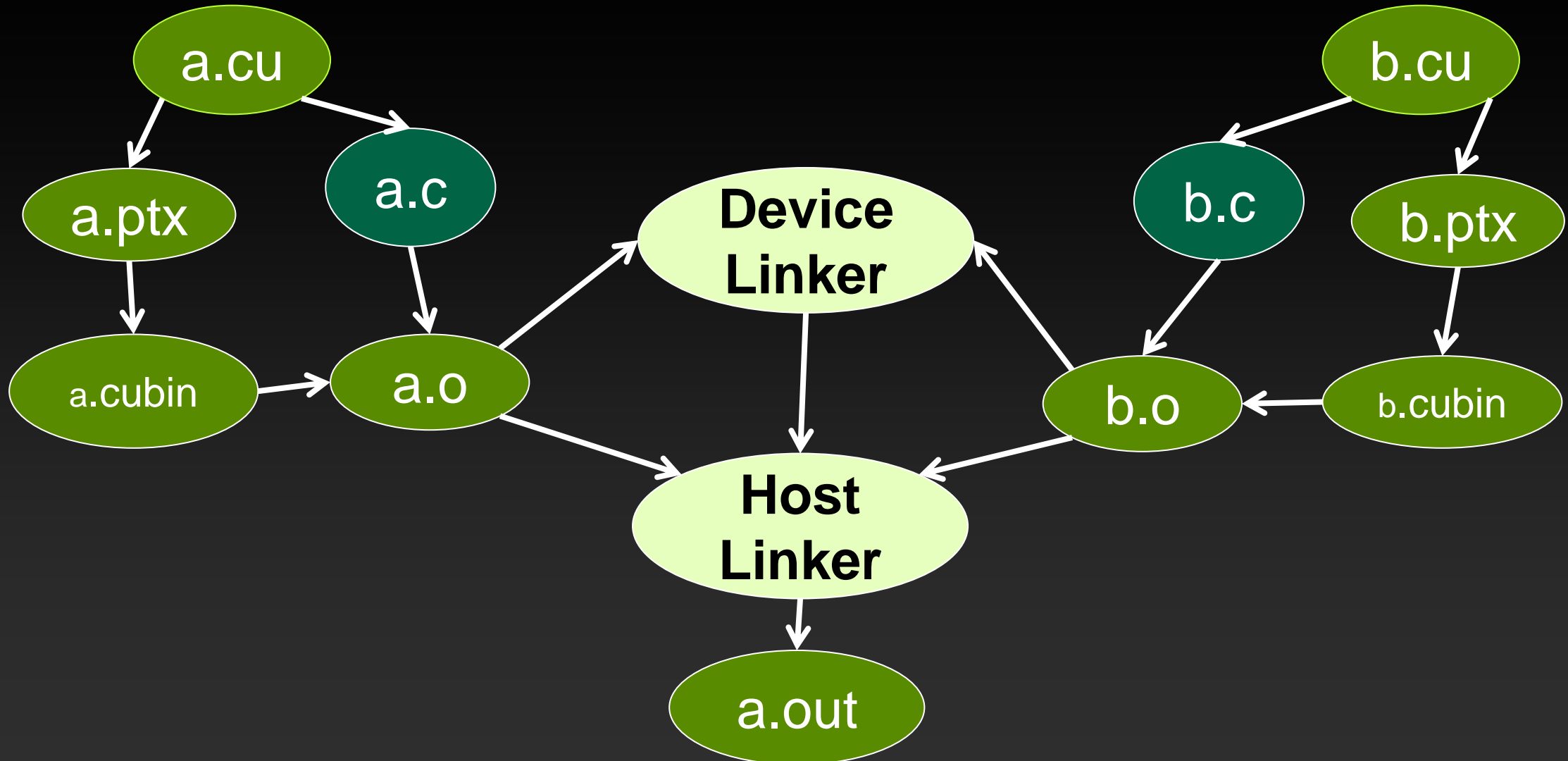
```
int main(int argc, char *argv[])  
{  
    parentKernel<<<1,1>>>();  
    cudaDeviceSynchronize();  
    return 0;  
}
```

# Compile Trajectory

- Separation of host and device code
- Device code translates into device-specific binary (.cubin) or device independent assembly (.ptx)
- Device code embedded in host object data



# CUDA 5 Introduces Device Code Linker



# Device Linker Invocation

- Introduction of an optional link step for device code

```
nvcc -arch=sm_20 -dc a.cu b.cu
```

```
nvcc -arch=sm_20 -dlink a.o b.o -o link.o
```

```
g++ a.o b.o link.o -L<path> -lcudart
```

- Link device-runtime library for dynamic parallelism

```
nvcc -arch=sm_35 -dc a.cu b.cu
```

```
nvcc -arch=sm_35 -dlink a.o b.o -lcudadevrt -o link.o
```

```
g++ a.o b.o link.o -L<path> -lcudadevrt -lcudart
```

- Currently, link occurs at cubin level (PTX not supported)

# Where to find additional information

## CUDA documentation [1]

- Best Practice Guide [2]
- Kepler Tuning Guide [3]

## Kepler whitepaper [4]

- [1] <http://docs.nvidia.com>  
[2] <http://docs.nvidia.com/cuda/cuda-c-best-practices-guide>  
[3] <http://docs.nvidia.com/cuda/kepler-tuning-guide>  
[4] <http://www.nvidia.com/object/nvidia-kepler.html>

The screenshot displays the NVIDIA Developer Zone CUDA Toolkit Documentation page. The left sidebar contains a navigation menu with the following items: CUDA Toolkit, CUDA C Best Practices Guide (selected), Preface, What Is This Document?, Who Should Read This Guide?, Assess, Parallelize, Optimize, Deploy, Assess, Parallelize, Optimize, Deploy, Assess, Parallelize, Optimize, Deploy, Recommendations and Best Practices, Heterogeneous Computing, Differences between Host and Device, What Runs on a CUDA-Enabled Device?, Application Profiling, Profile, Creating the Profile, Identifying Hotspots, Understanding Scaling, Getting Started, Parallel Libraries, Parallelizing Compilers, Coding to Expose Parallelism.

The main content area shows the 'Assess, Parallelize, Optimize, Deploy' (APOD) design cycle. It includes a paragraph: 'and that you have a basic familiarity with the CUDA C programming language and environment (if not, please refer to the *CUDA C Programming Guide*).' followed by the heading 'Assess, Parallelize, Optimize, Deploy'. The text describes the APOD design cycle for applications with the goal of helping application developers to rapidly identify the portions of their code that would most readily benefit from GPU acceleration, rapidly realize that benefit, and begin leveraging the resulting speedups in production as early as possible. It also states: 'APOD is a cyclical process: initial speedups can be achieved, tested, and deployed with only minimal initial investment of time, at which point the cycle can begin again by identifying further optimization opportunities, seeing additional speedups, and then deploying the even faster versions of the application into production.'

Below the text is a diagram illustrating the APOD design cycle. It consists of four rounded rectangular boxes arranged in a diamond shape, connected by curved arrows indicating a clockwise cycle: ASSESS at the top, PARALLELIZE on the right, DEPLOY at the bottom, and another ASSESS at the top. The diagram is shown twice, once in the main content area and once in a smaller, faded version at the bottom of the page.