GPU TECHNOLOGY CONFERENCE

Inside Kepler

Lars Nyland – Architecture

Stephen Jones – CUDA

NVIDIA Corporation

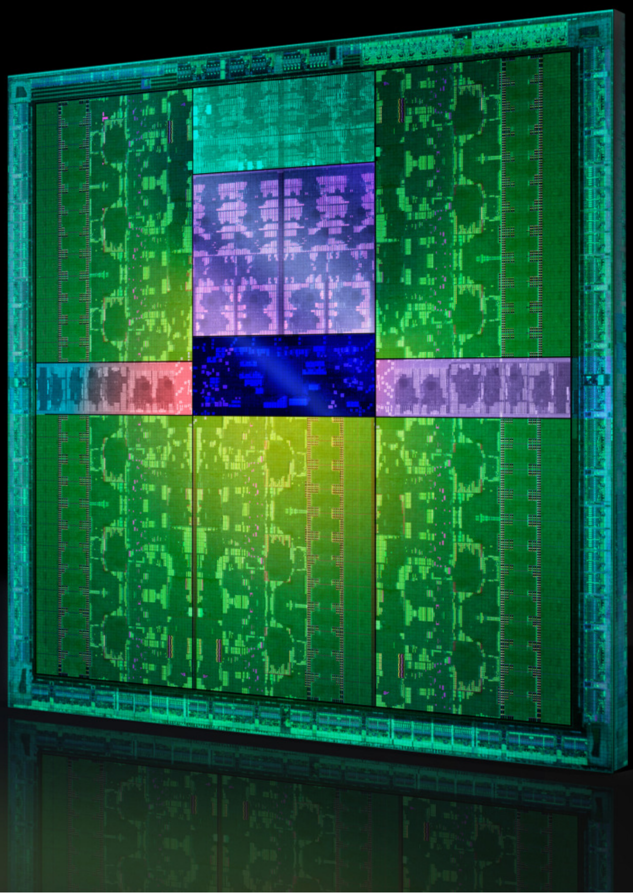
NVIDIA

Welcome the Kepler GK110 GPU

Performance

Efficiency

Programmability

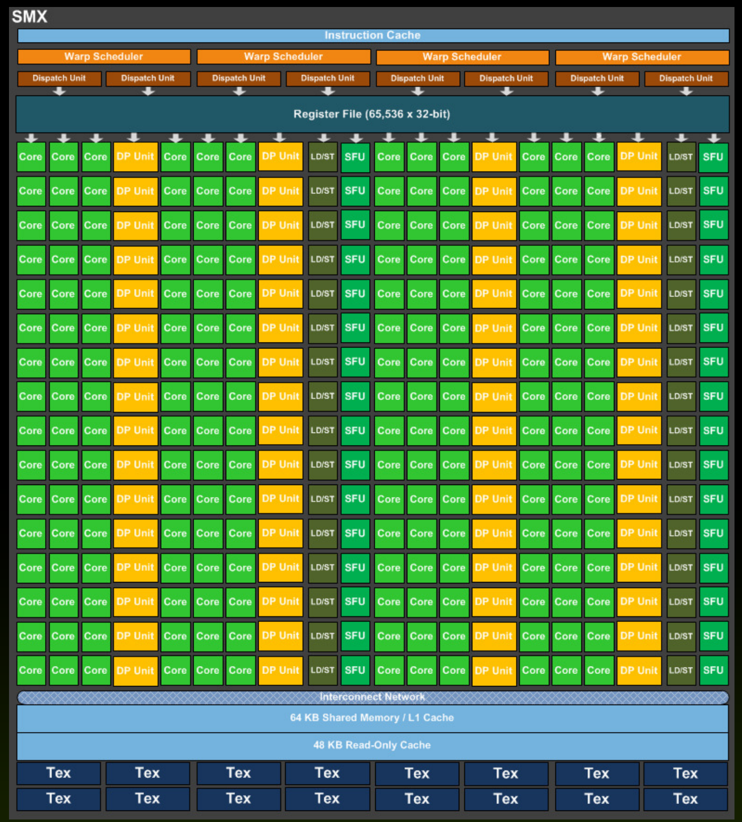


Kepler GK110 Block Diagram

Architecture

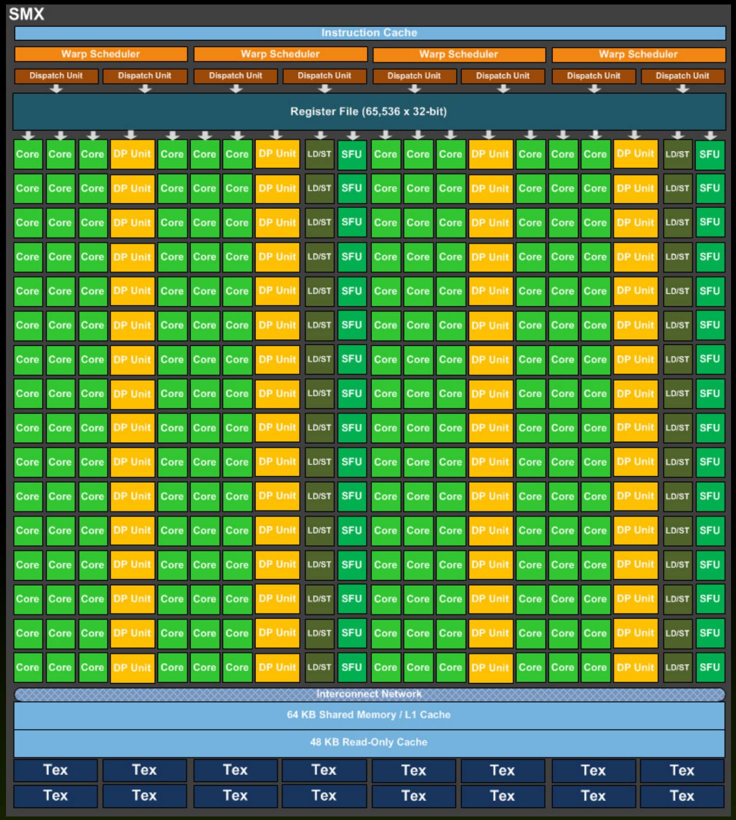
- 7.1B Transistors
- 15 SMX units
- > 1 TFLOP FP64
- 1.5 MB L2 Cache
- 384-bit GDDR5
- PCI Express Gen3

Kepler GK110 SMX vs Fermi SM

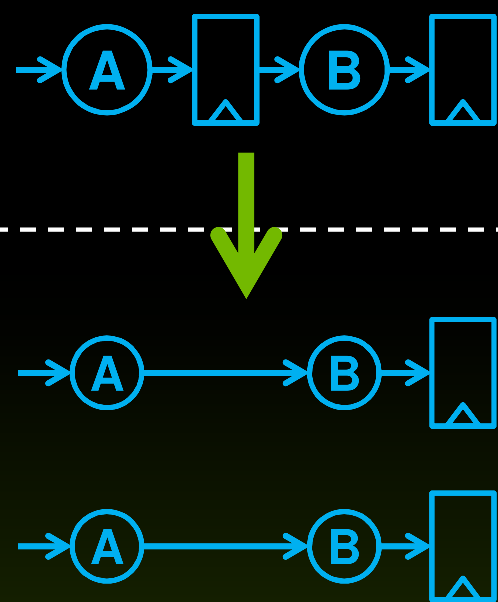


SMX: Efficient Performance

- **Power-Aware SMX Architecture**
- **Clocks & Feature Size**
- **SMX result -**
  - **Performance up**
  - **Power down**



Power vs Clock Speed Example



	Logic		Clocking	
	Area	Power	Area	Power
Fermi 2x clock	1.0x	1.0x	1.0x	1.0x
Kepler 1x clock	1.8x	0.9x	1.0x	0.5x

SMX Balance of Resources

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

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
- Ex. Quda QCD fp64 sample runs 5.3x faster
  - Spills are eliminated with extra registers

New High-Performance SMX Instructions

SHFL (shuffle) -- Intra-warp data exchange

ATOM -- Broader functionality, Faster

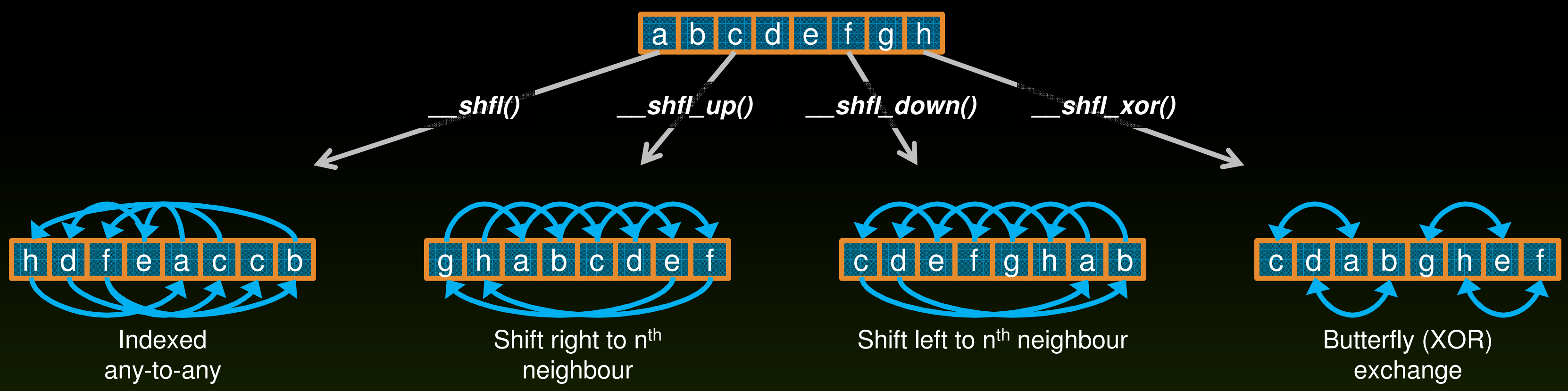
Compiler-generated, high performance instructions:

- 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:

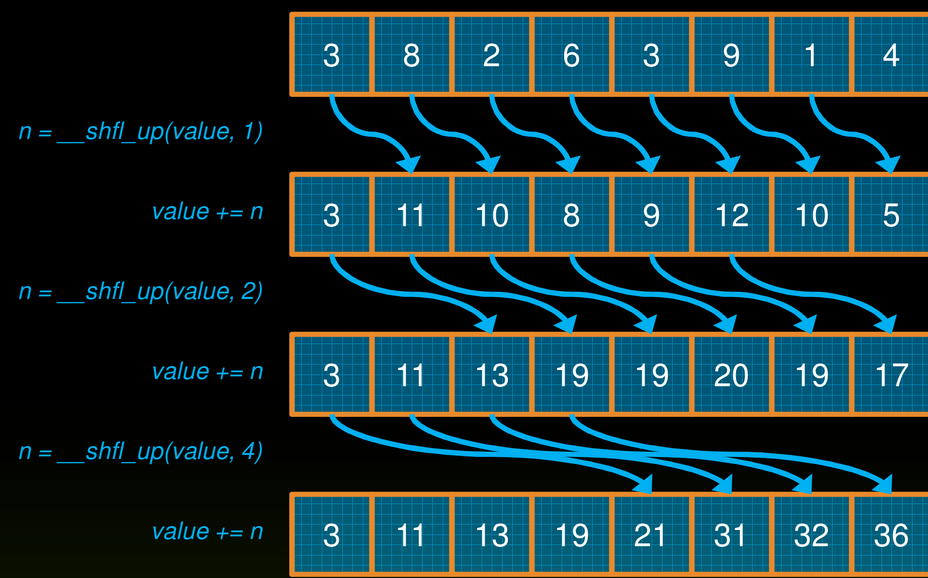


SHFL Example: Warp Prefix-Sum

```
__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**

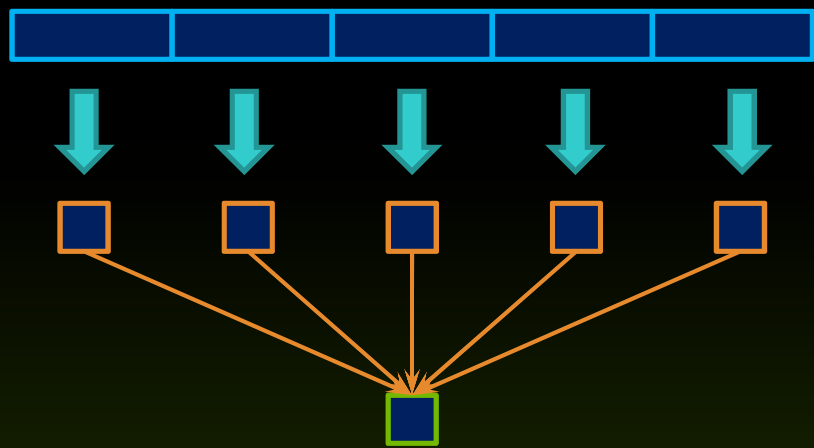
Atom Op	int32	int64
add	x	x
cas	x	x
exch	x	x
min/max	x	X
and/or/xor	x	X

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

High Speed Atomics Enable New Uses

Atomics are now fast enough to use within inner loops

- **Example: Data reduction (sum of all values)**



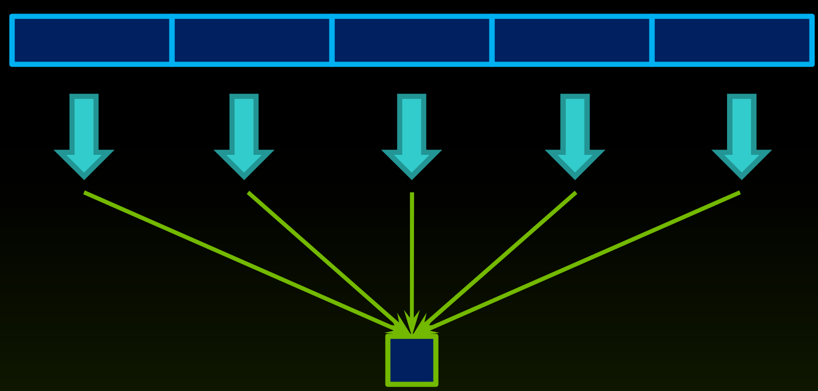
Without Atomics

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

Atomics are now fast enough to use within inner loops

- **Example: Data reduction (sum of all values)**

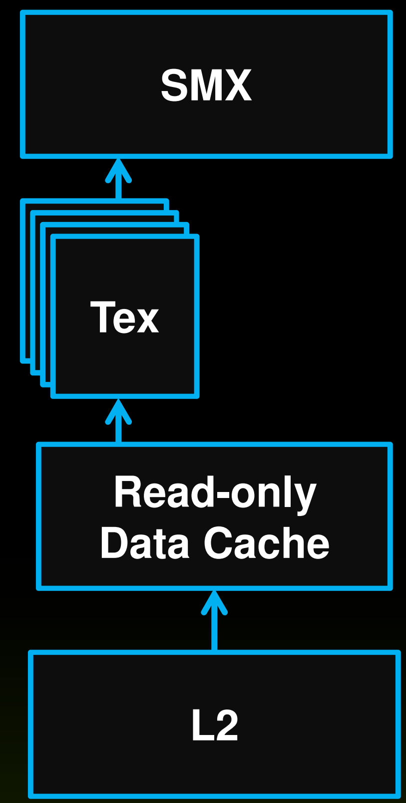


With Atomics

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.

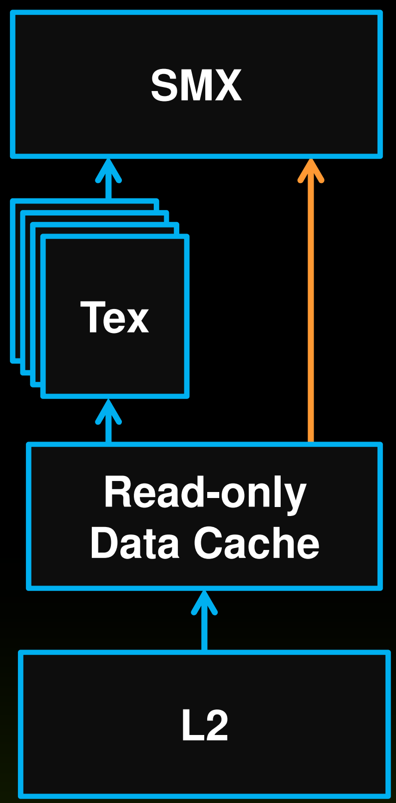
Texture performance

- 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 use texture
    // 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)

Bonsai GPU Tree-Code

Journal of Computational Physics,
231:2825-2839, April 2012

- Jeroen Bédorf, Simon Portegies Zwart
  - Leiden Observatory, The Netherlands
- Evghenii Gaburov
  - CIERA @ Northwestern U.
  - SARA, The Netherlands

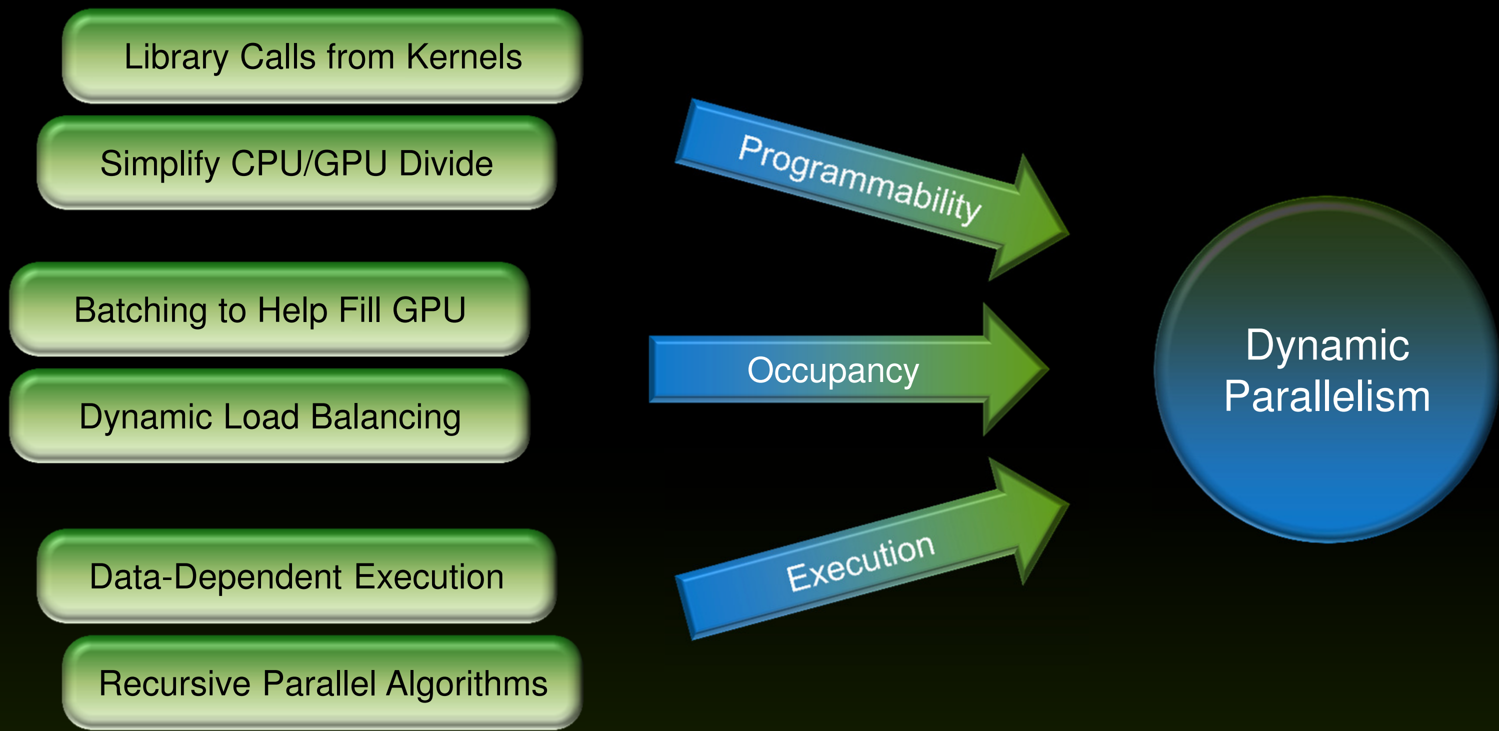
- Galaxies generated with: Galatics
  Widrow L. M., Dubinksi J., 2005,
  Astrophysical Journal, 631 838

Sterrewacht
Leiden

CIERA

sara

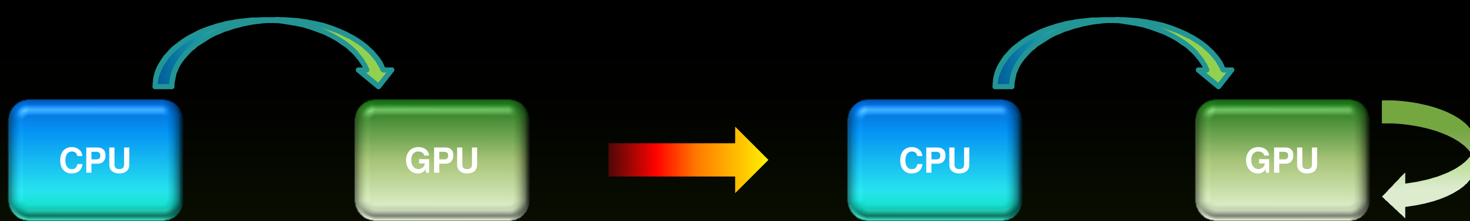
Improving Programmability



What is Dynamic Parallelism?

The ability to launch new grids from the GPU

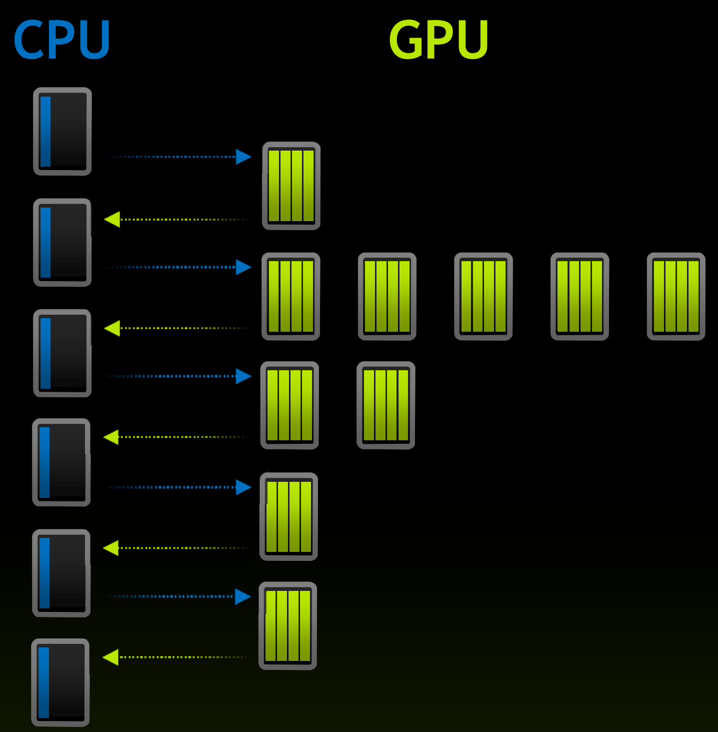
- Dynamically
- Simultaneously
- Independently



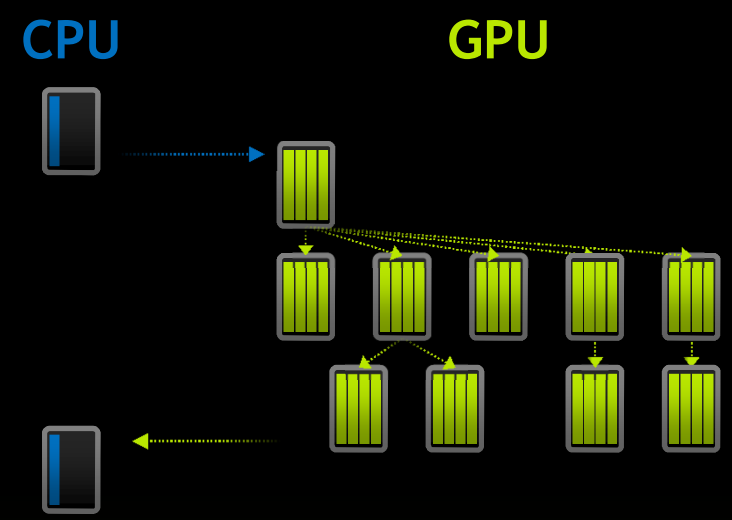
Fermi: Only CPU can generate GPU work

Kepler: GPU can generate work for itself

What Does It Mean?

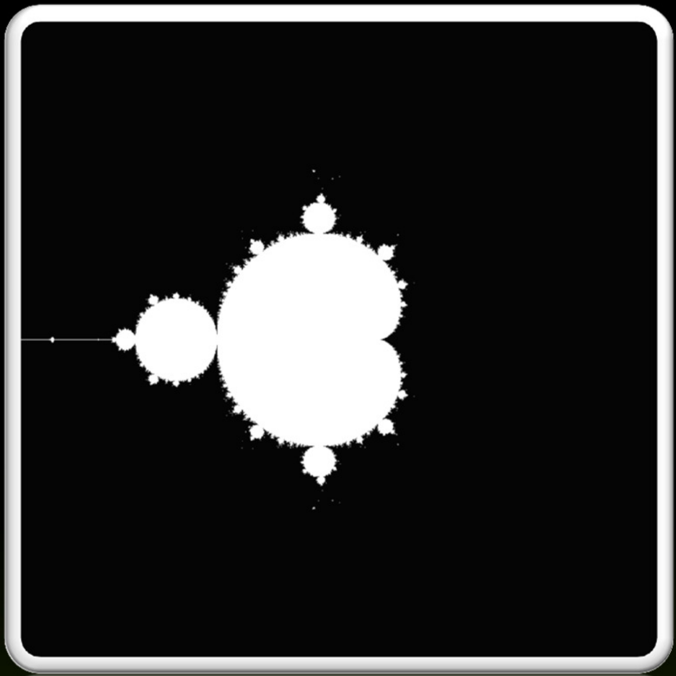


GPU as Co-Processor



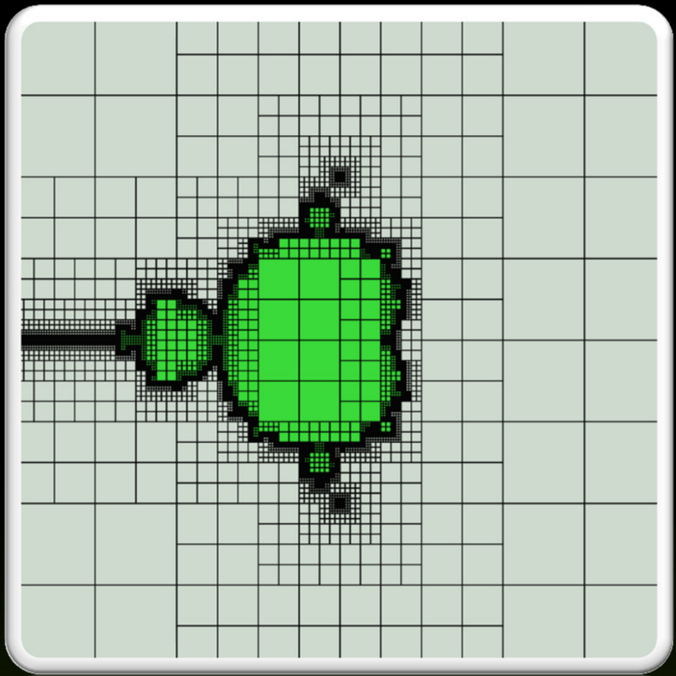
Autonomous, Dynamic Parallelism

Data-Dependent Parallelism



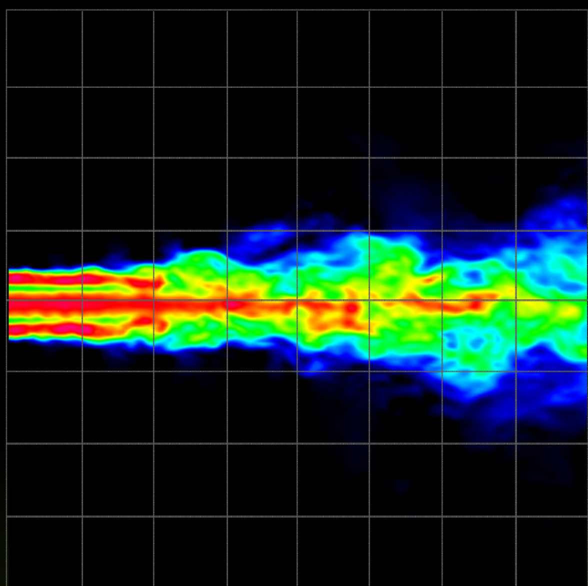
CUDA Today

Computational Power allocated to regions of interest



CUDA on Kepler

Dynamic Work Generation

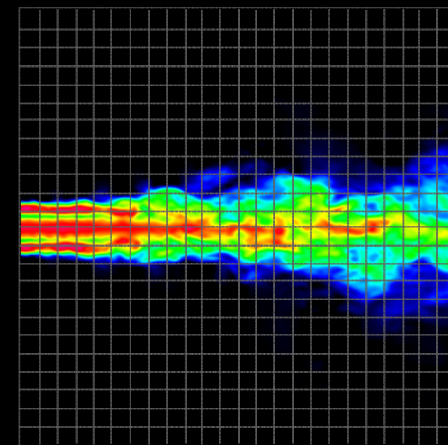


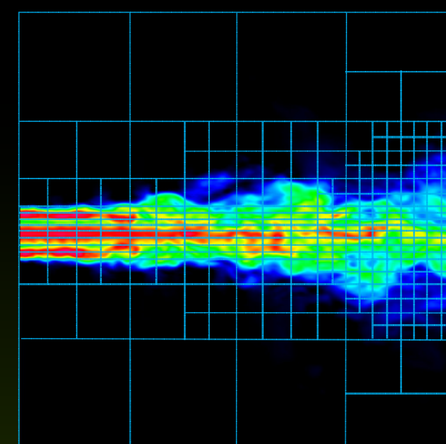
Initial Grid

Statically assign conservative worst-case grid

Dynamically assign performance where accuracy is required

Fixed Grid



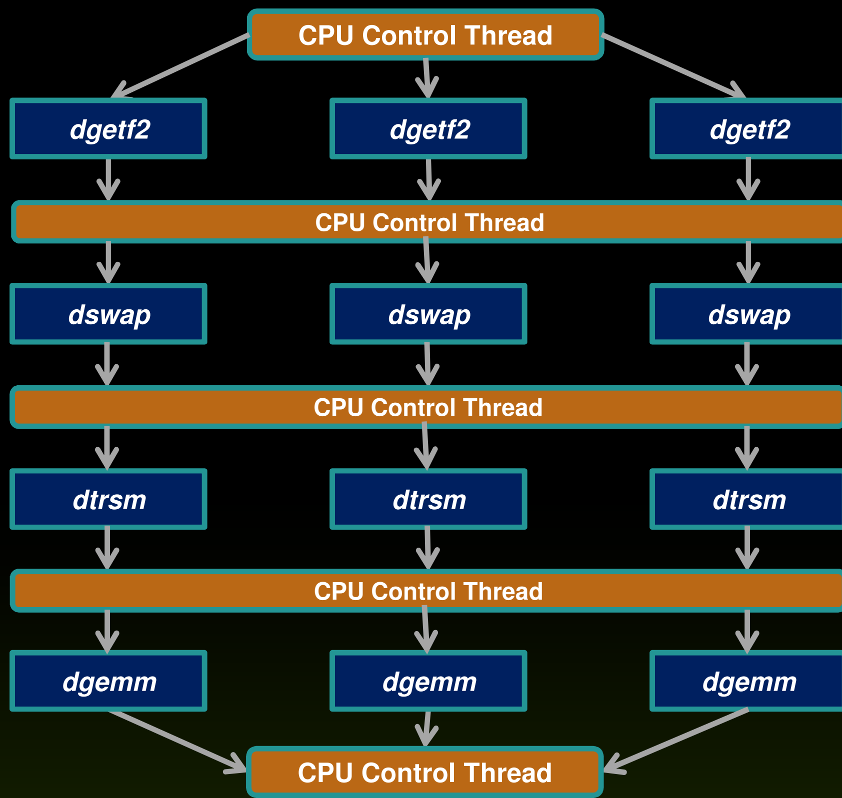


Dynamic Grid

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



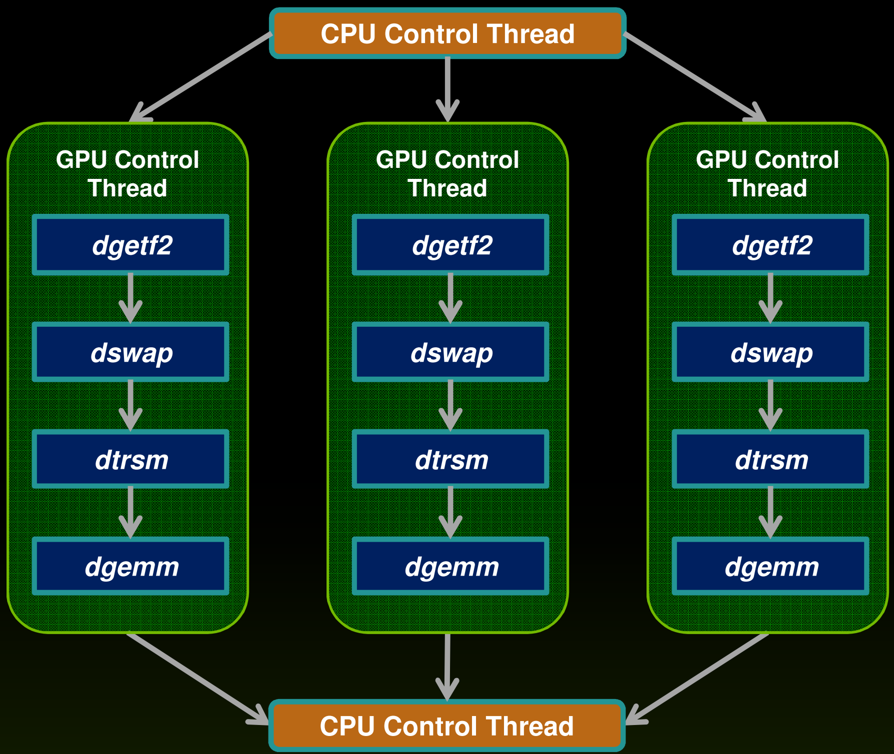
Multiple LU-Decomposition, Pre-Kepler

Algorithm flow simplified for illustrative purposes

Batched & Nested Parallelism

Batching via Dynamic Parallelism

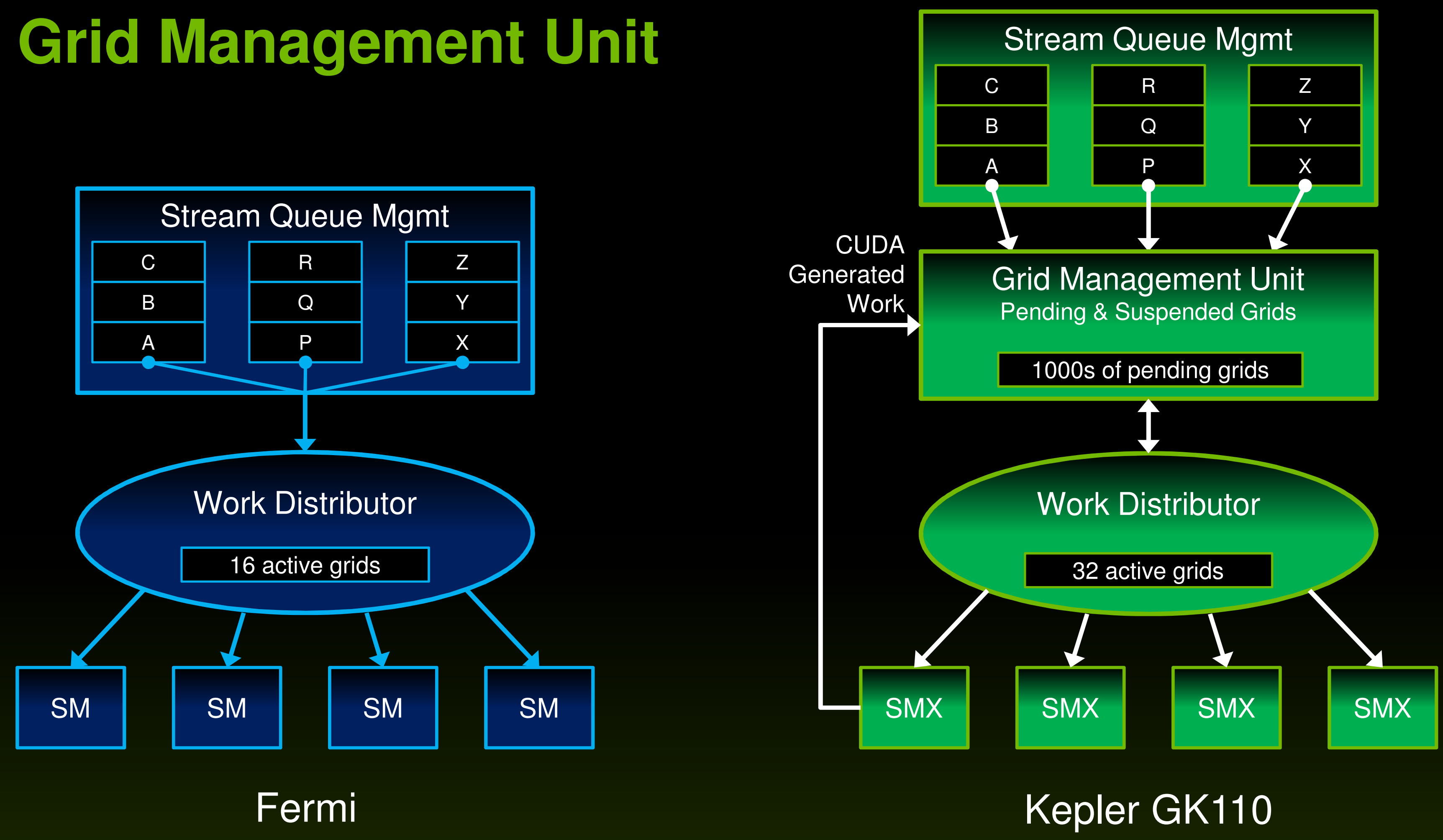
- Move top-level loops to GPU
- Run thousands of independent tasks
- Release CPU for other work



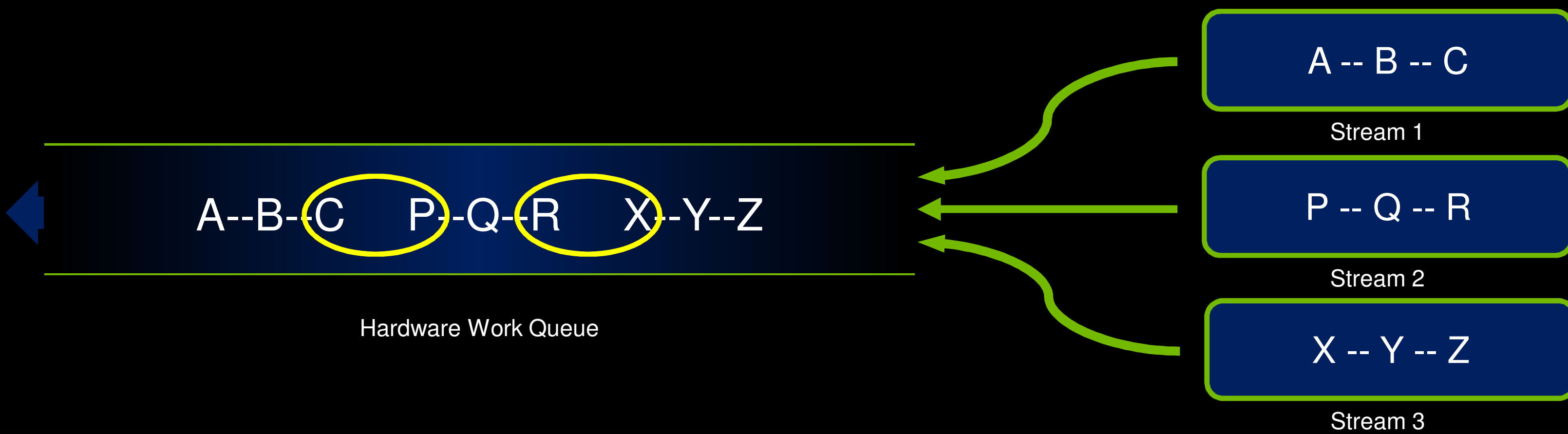
Batched LU-Decomposition, Kepler

Algorithm flow simplified for illustrative purposes

Grid Management Unit



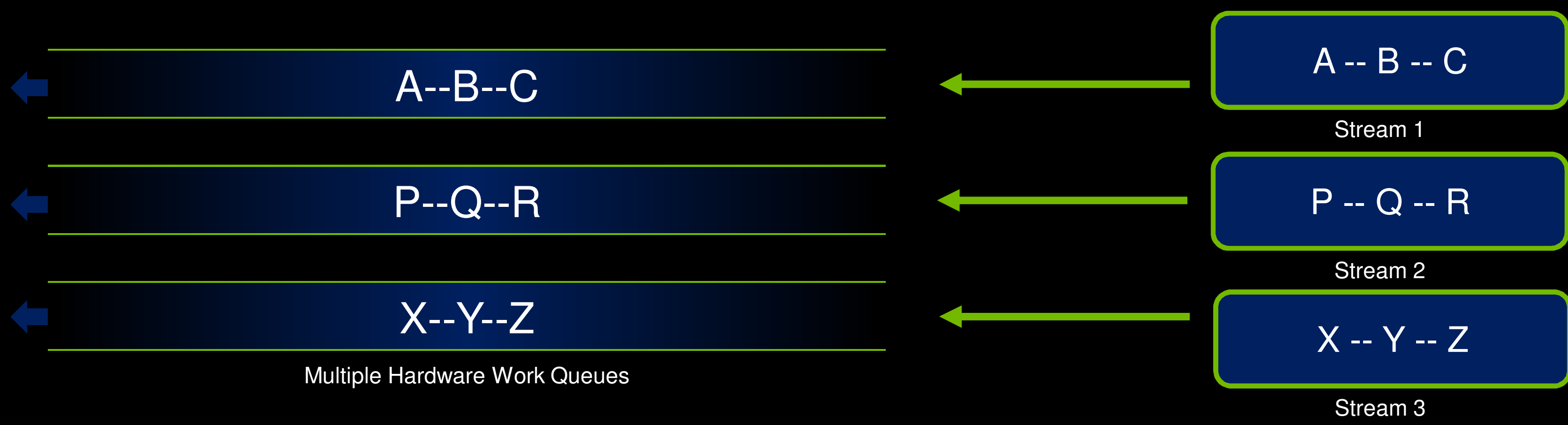
Fermi Concurrency



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

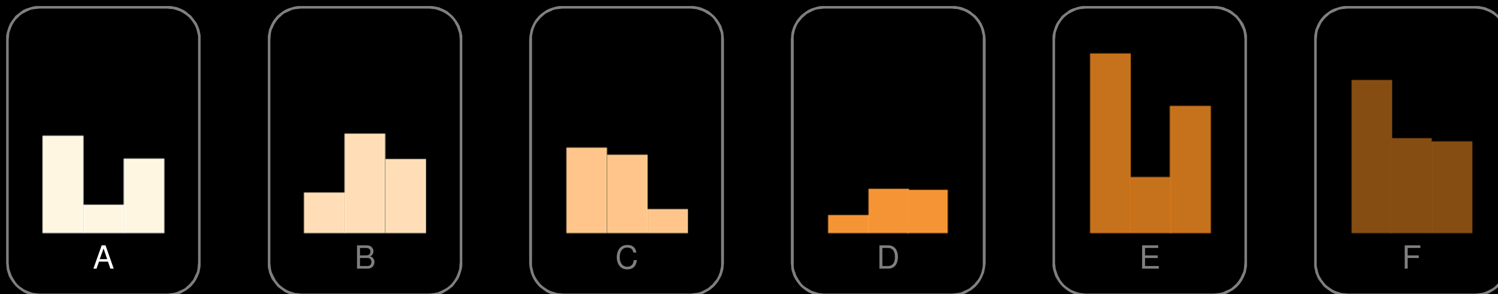
Kepler Improved Concurrency



Kepler allows 32-way concurrency

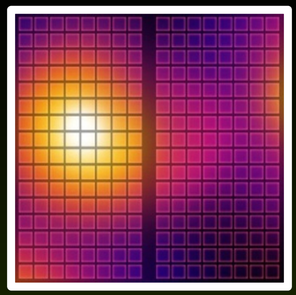
- One work queue per stream
- Concurrency at full-stream level
- No inter-stream dependencies

Fermi: Time-Division Multiprocess

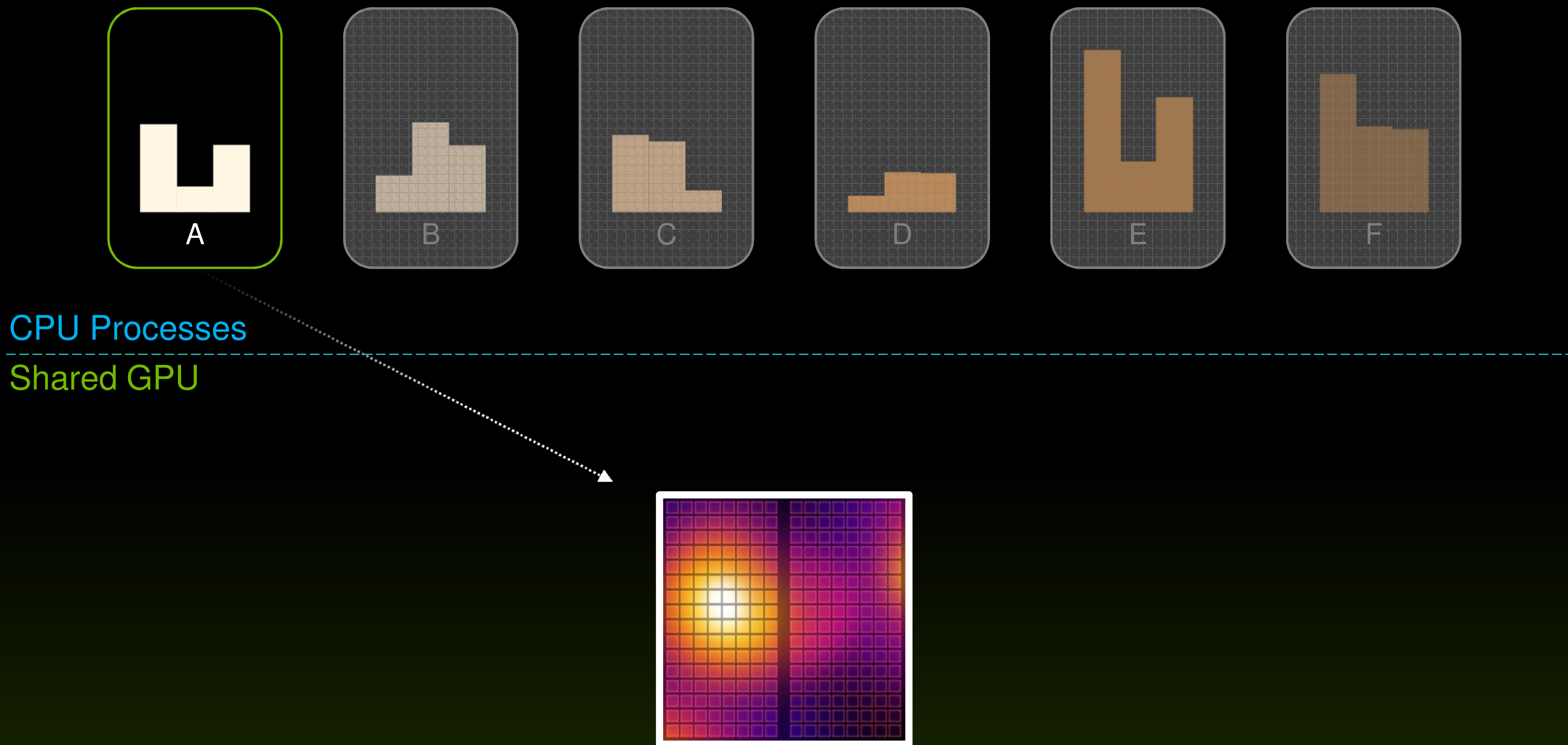


CPU Processes

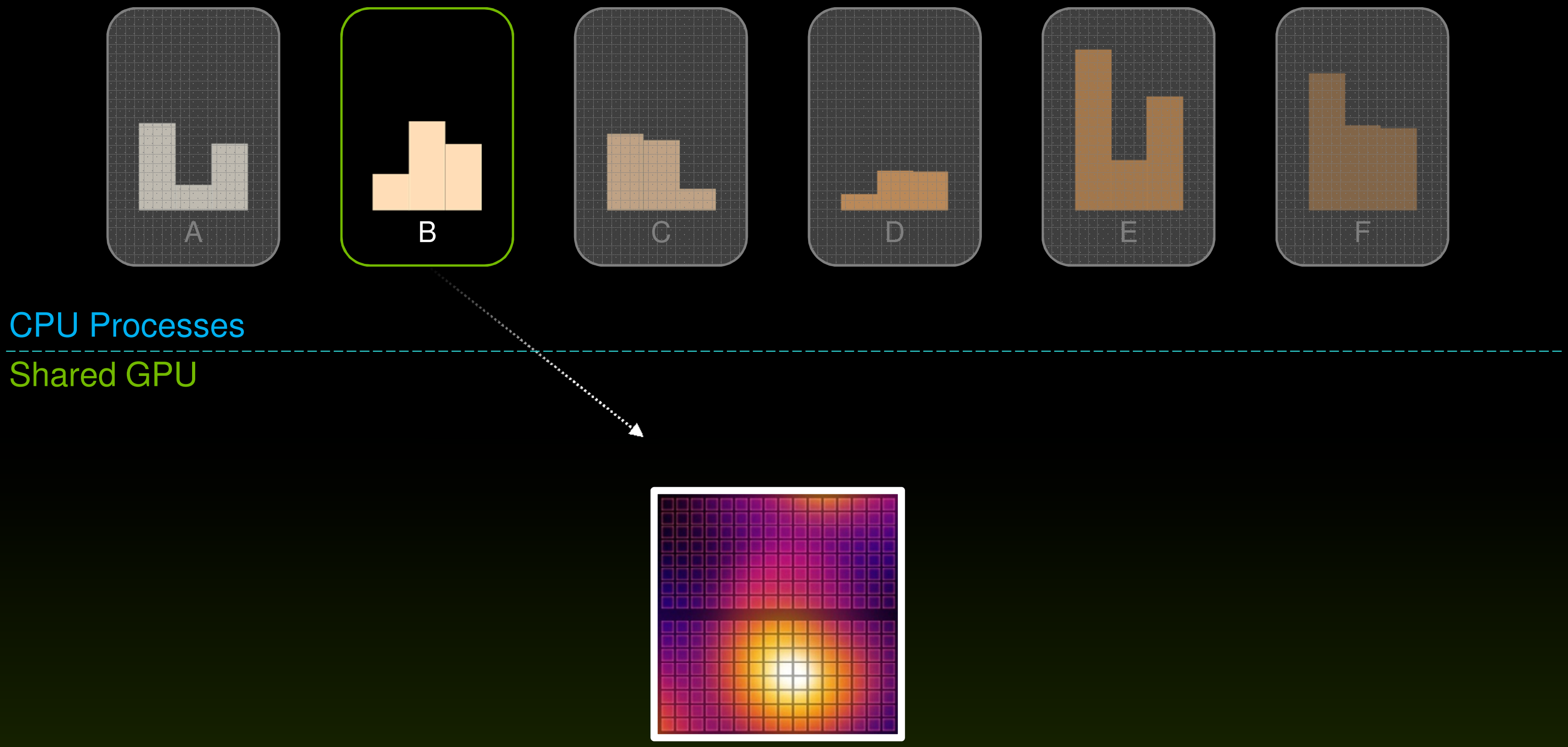
Shared GPU



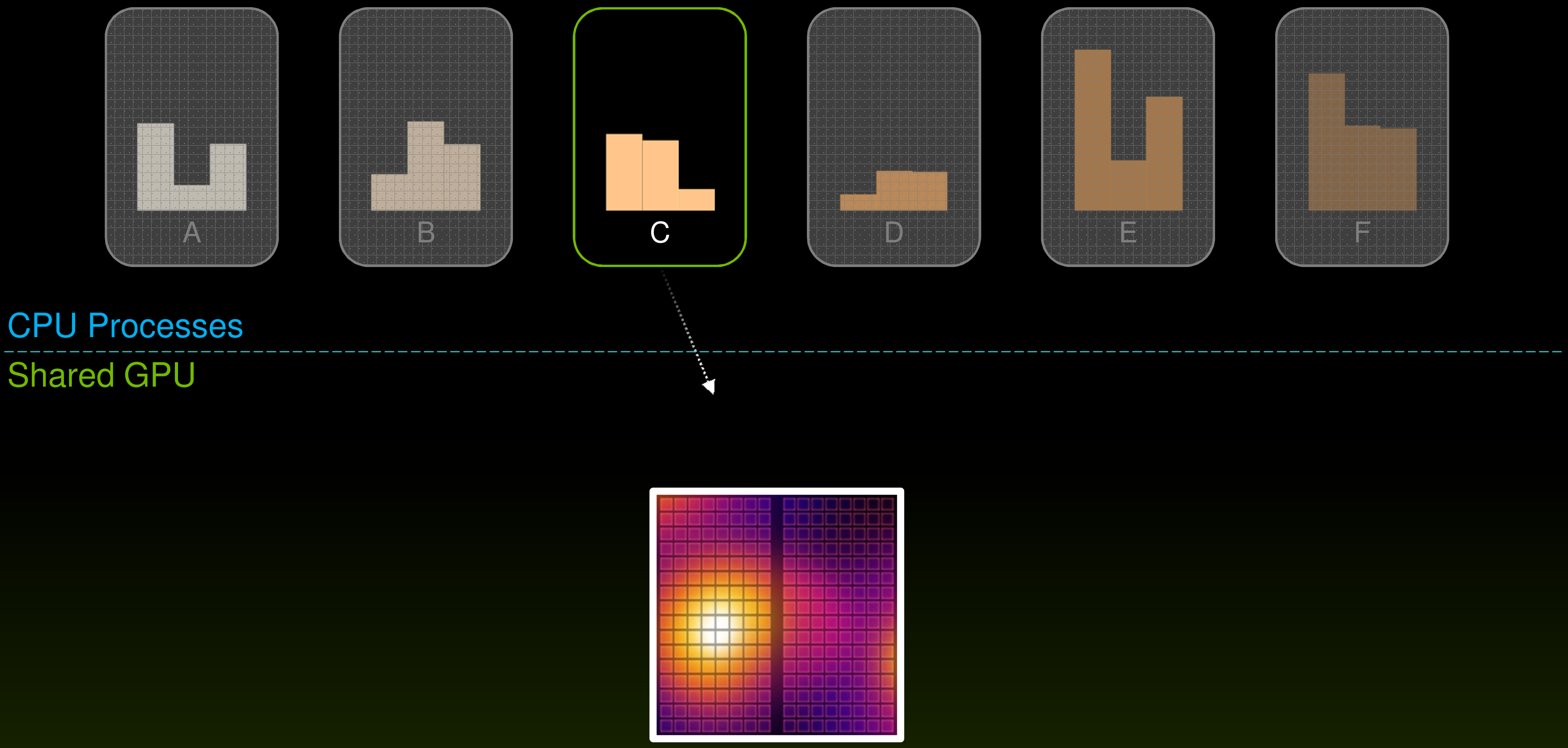
Fermi: Time-Division Multiprocess



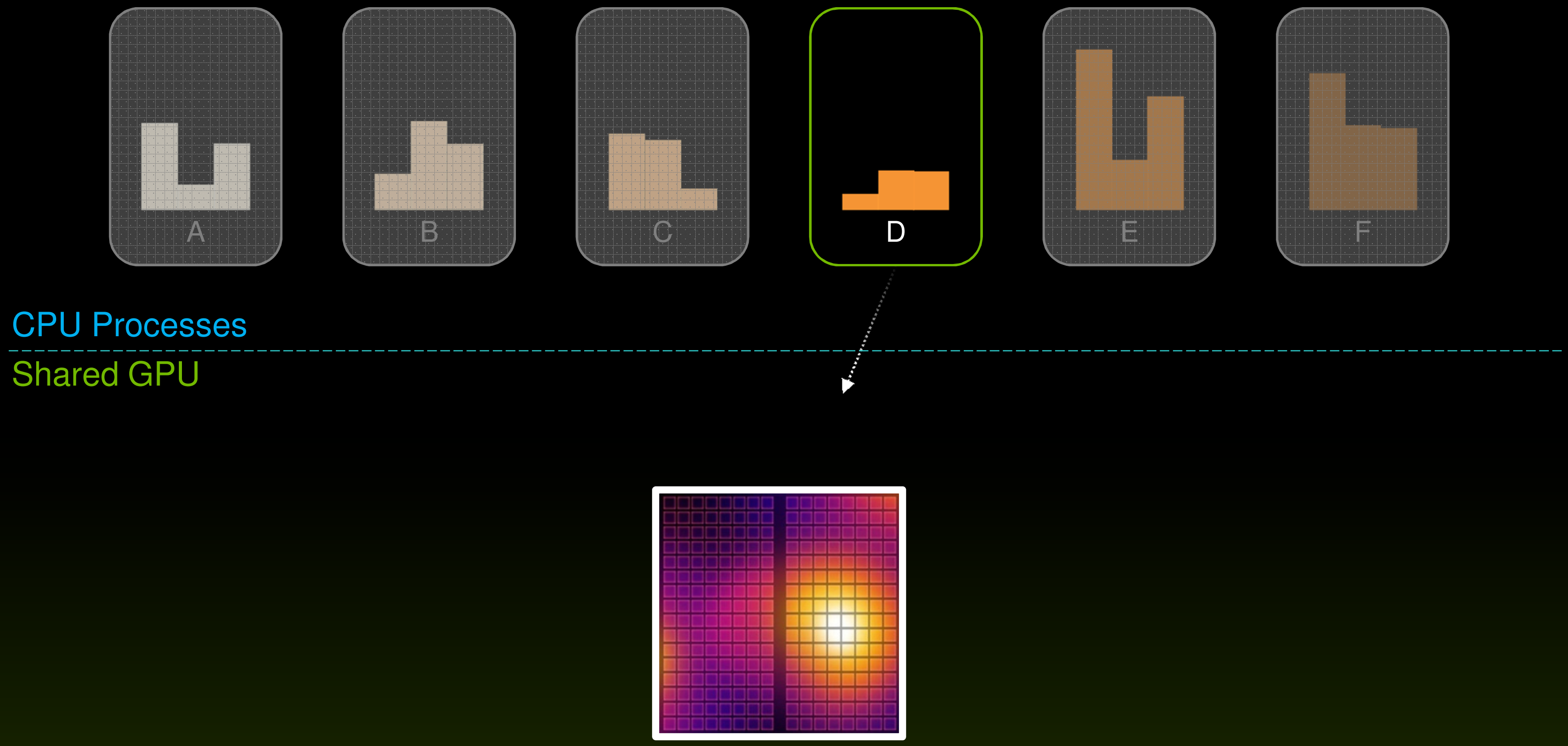
Fermi: Time-Division Multiprocess



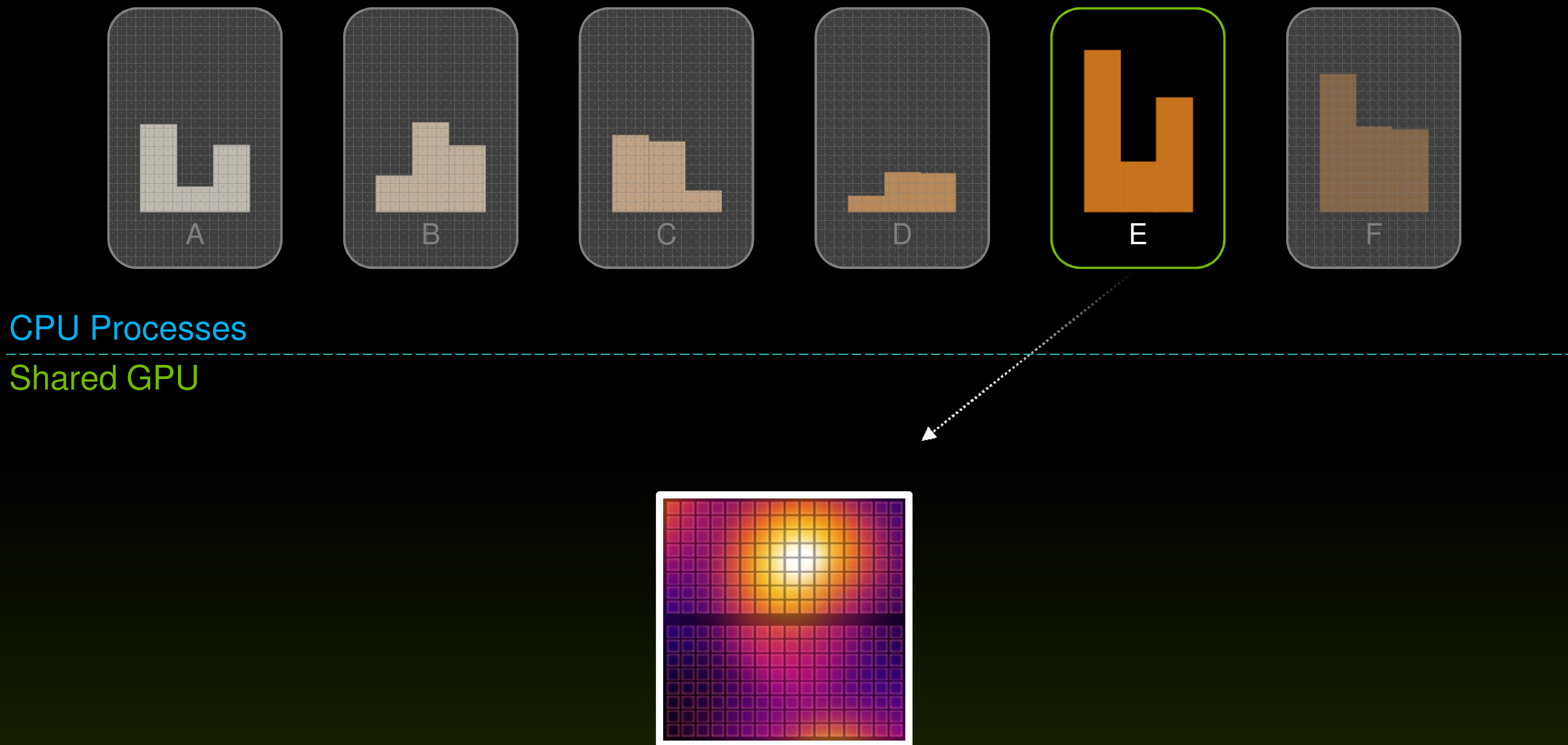
Fermi: Time-Division Multiprocess



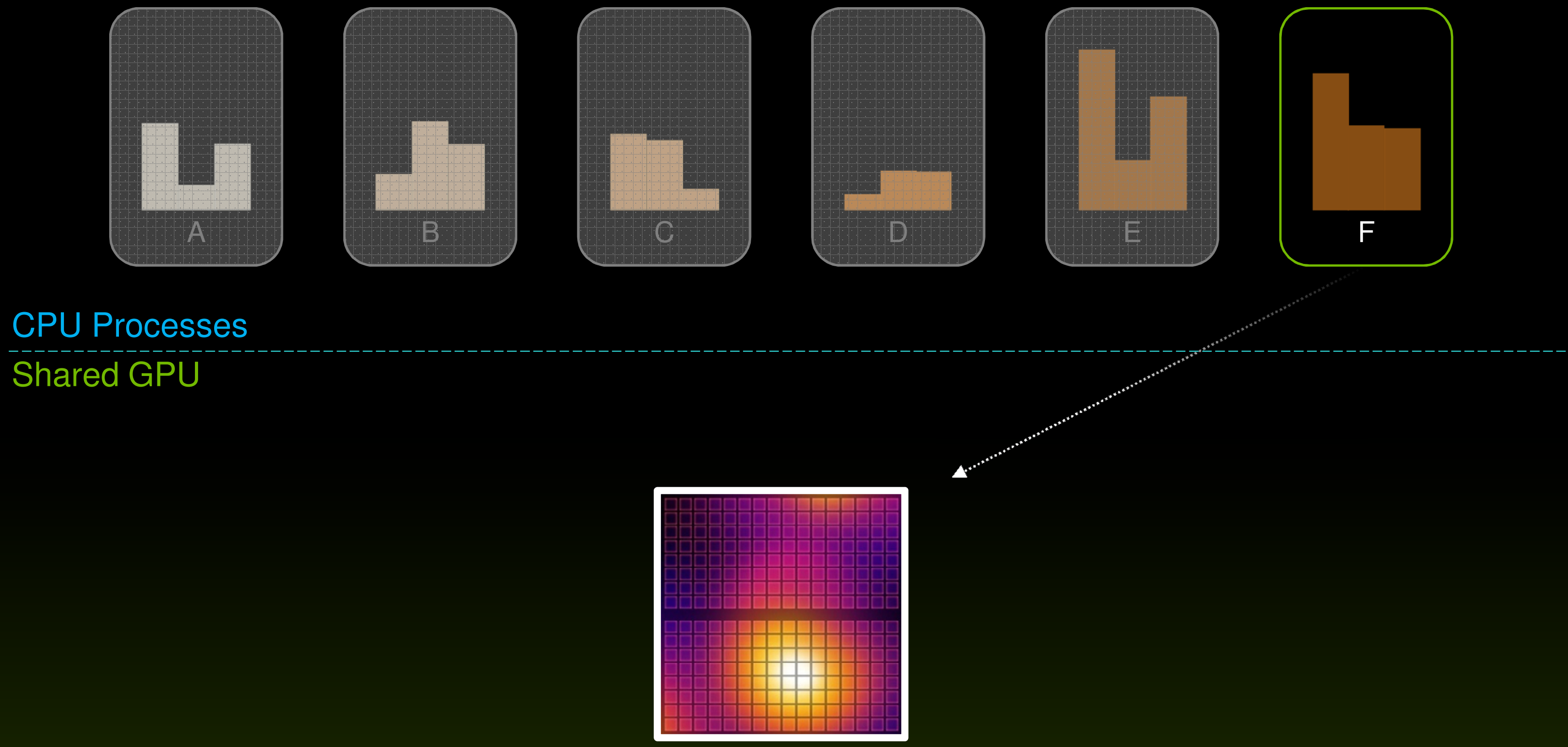
Fermi: Time-Division Multiprocess



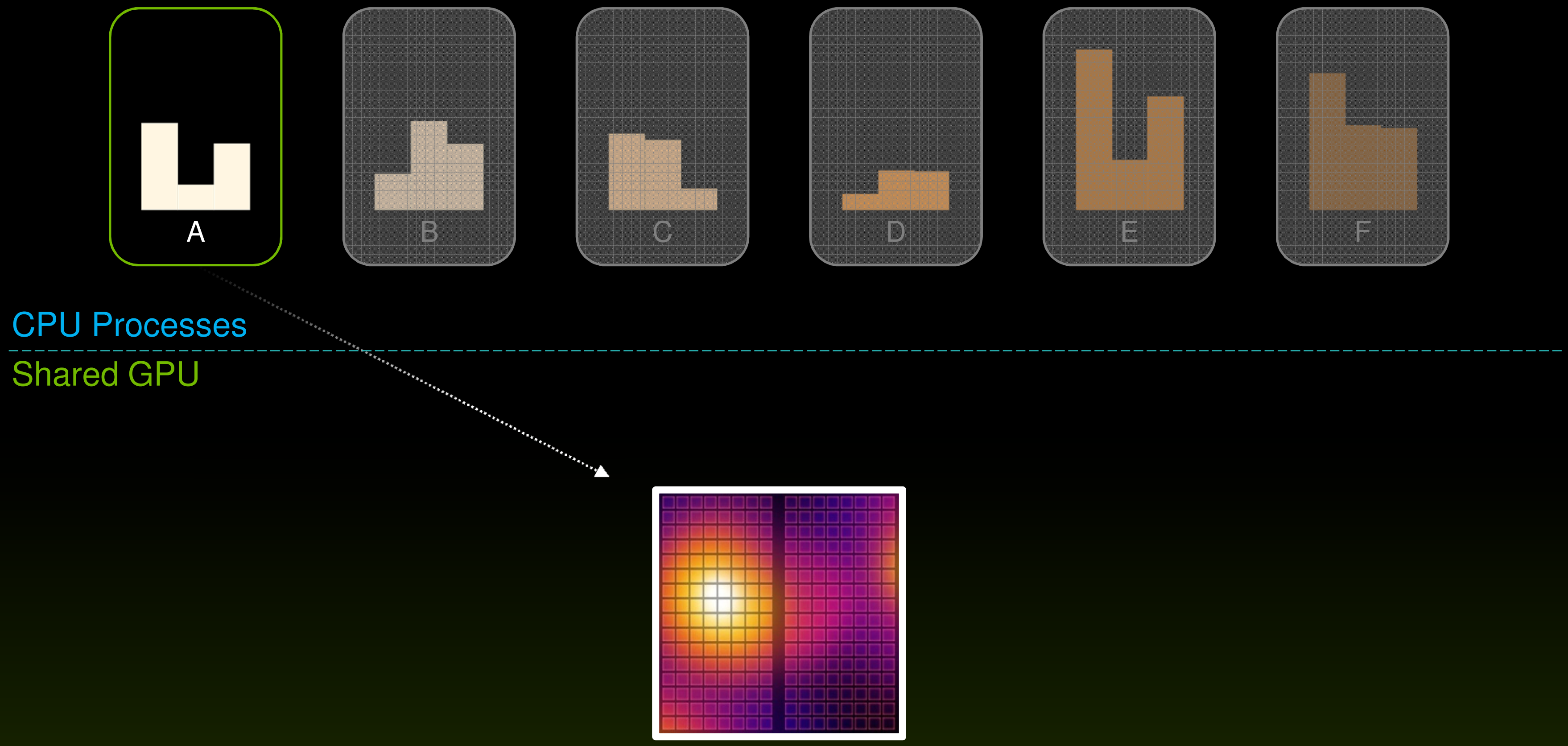
Fermi: Time-Division Multiprocess



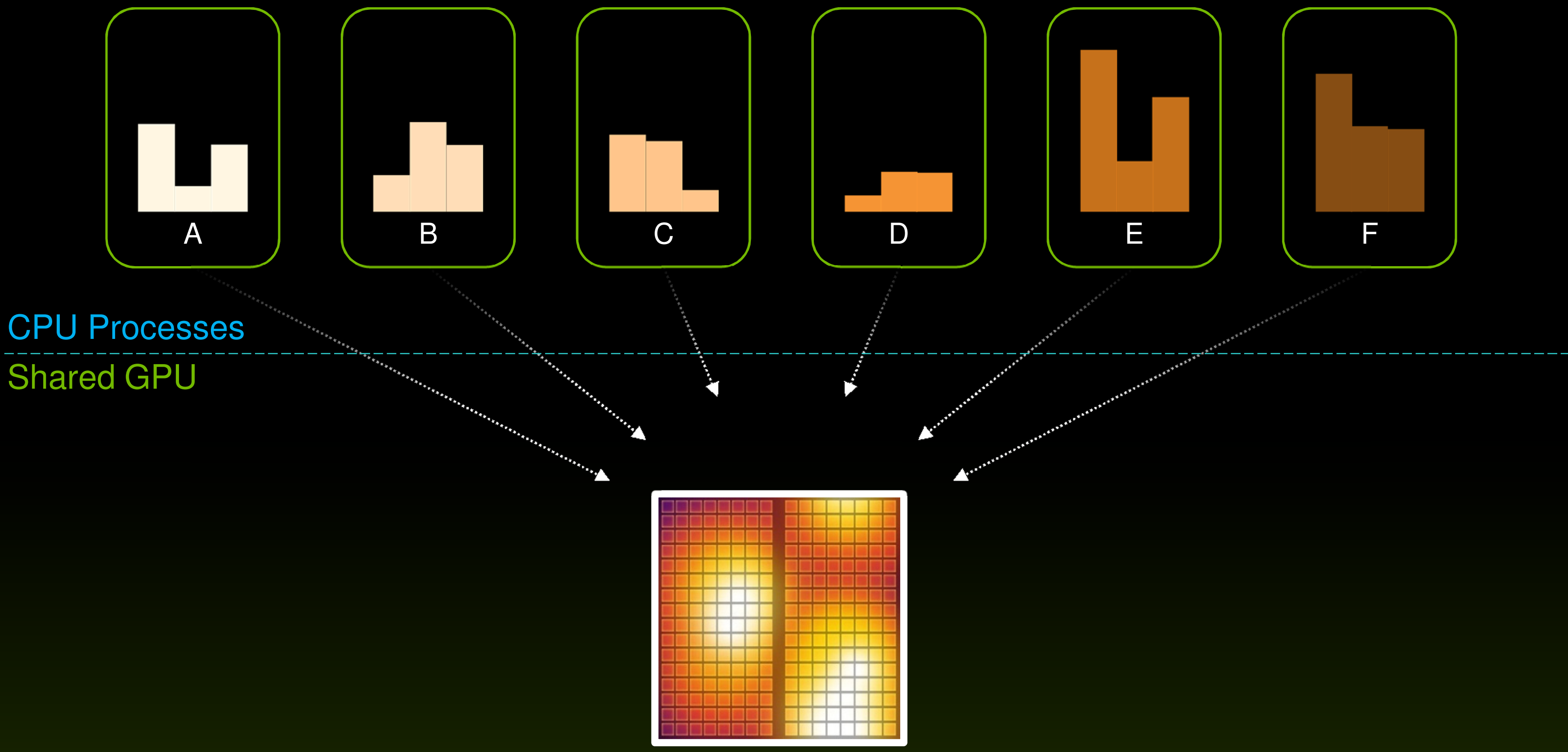
Fermi: Time-Division Multiprocess

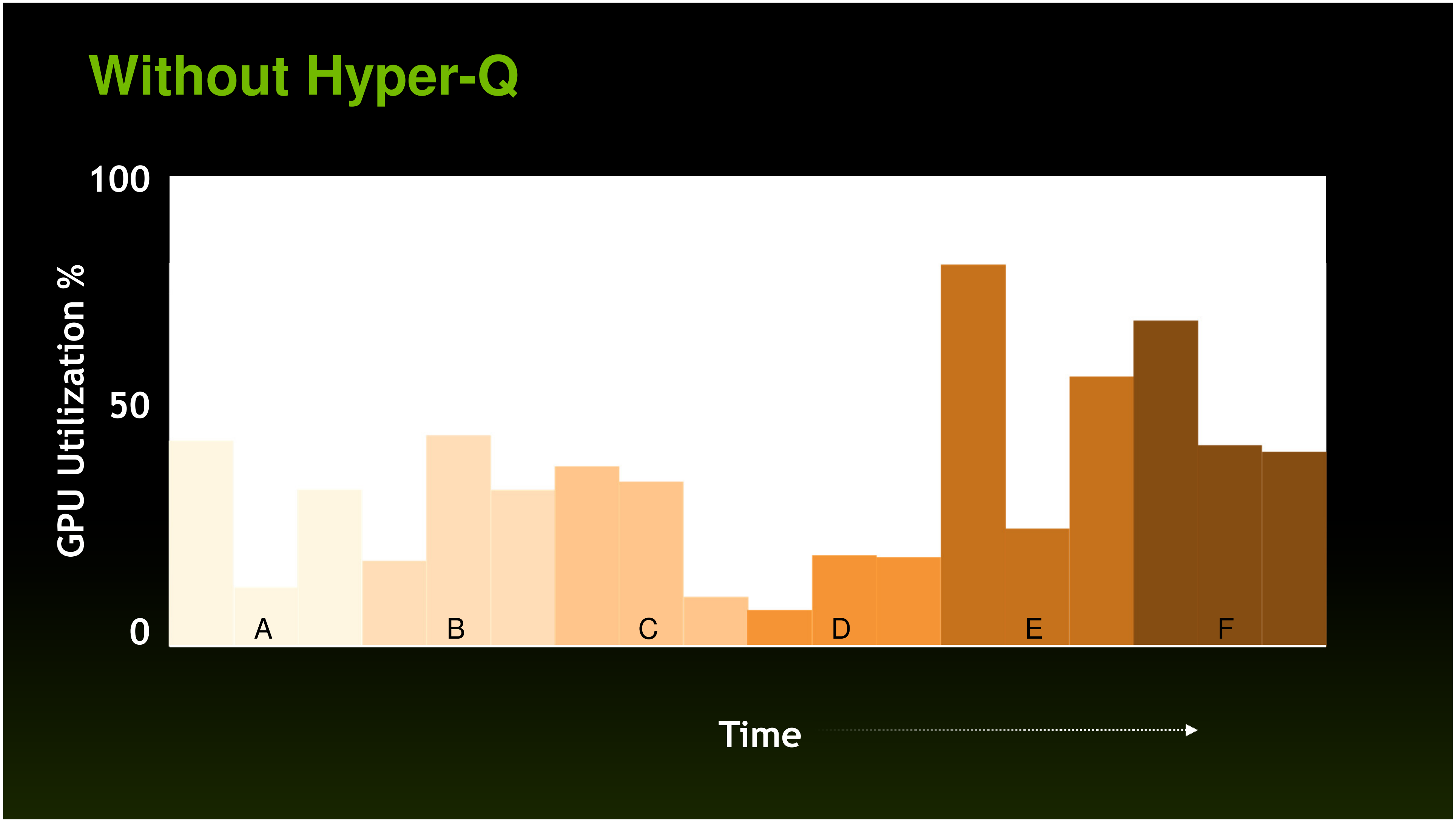


Fermi: Time-Division Multiprocess



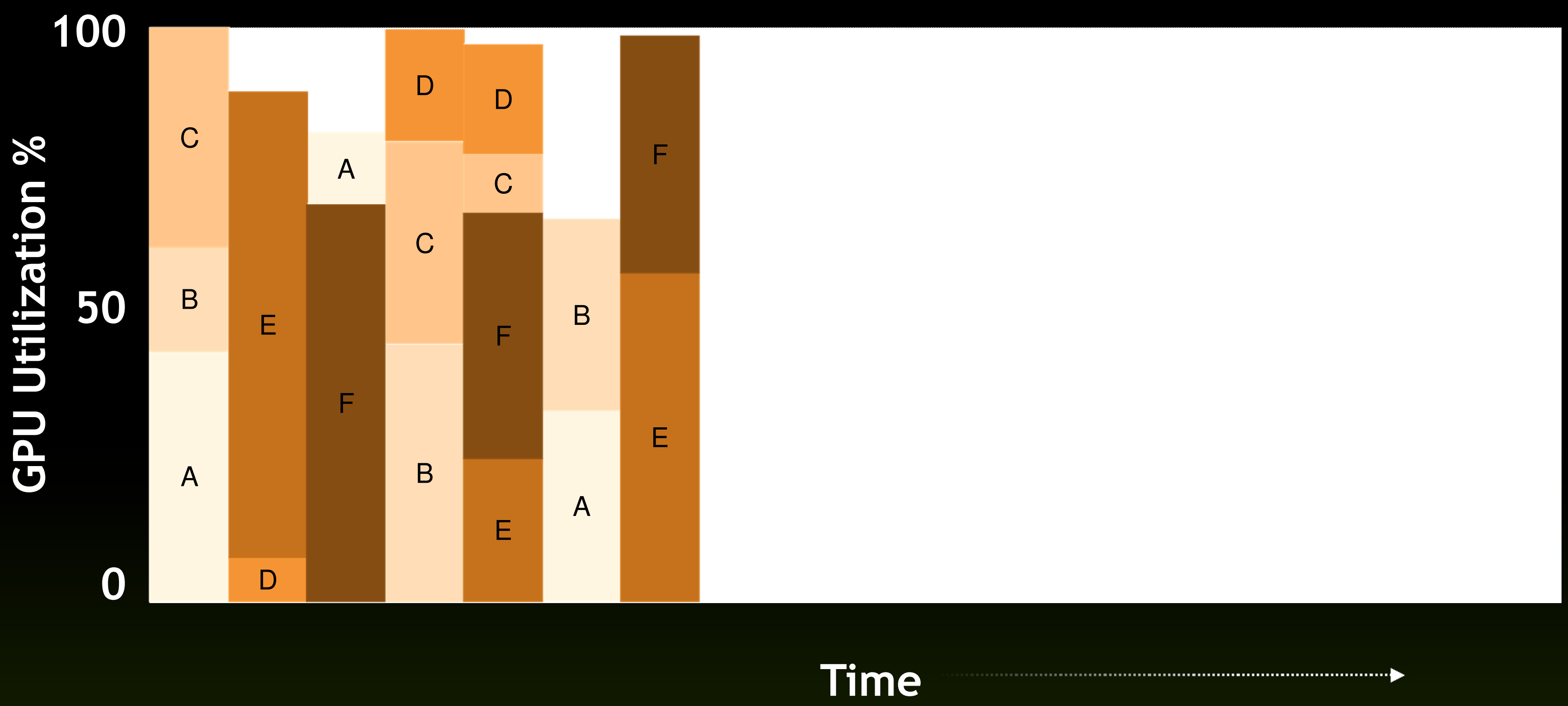
Hyper-Q: Simultaneous Multiprocess

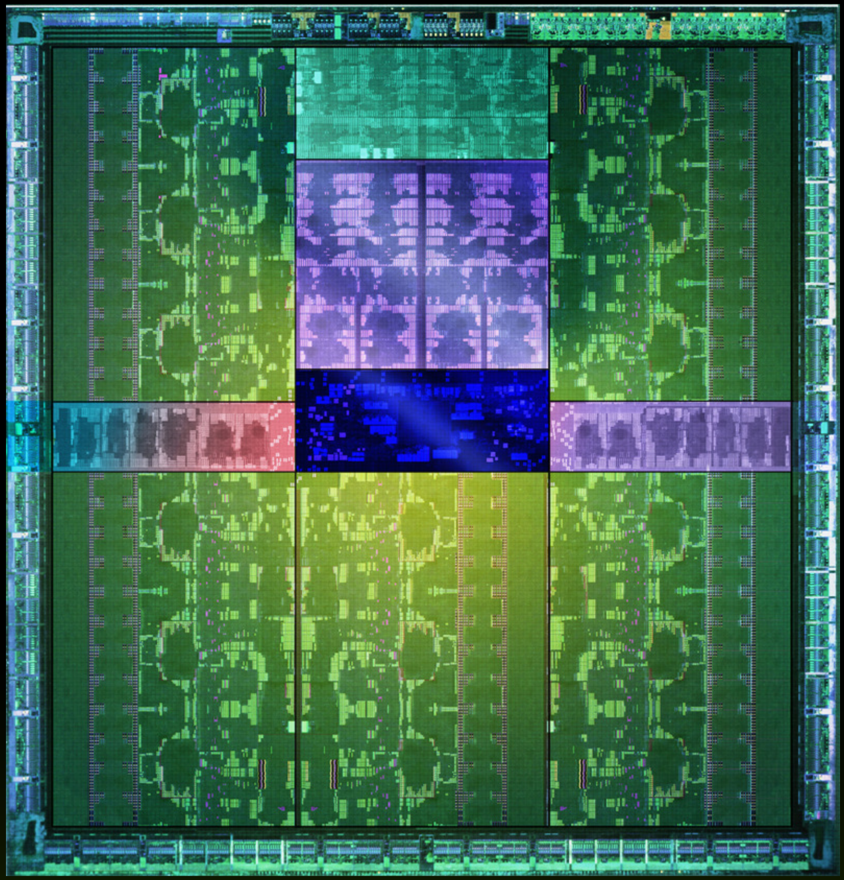




Without Hyper-Q
100
50
0
GPU Utilization %
A
B
C
D
E
F
Time

With Hyper-Q





Whitepaper: http://www.nvidia.com/object/nvidia-kepler.html