GPU Architecture in detail and Performance Optimization (Part II)

Bin ZHOU 2014/06

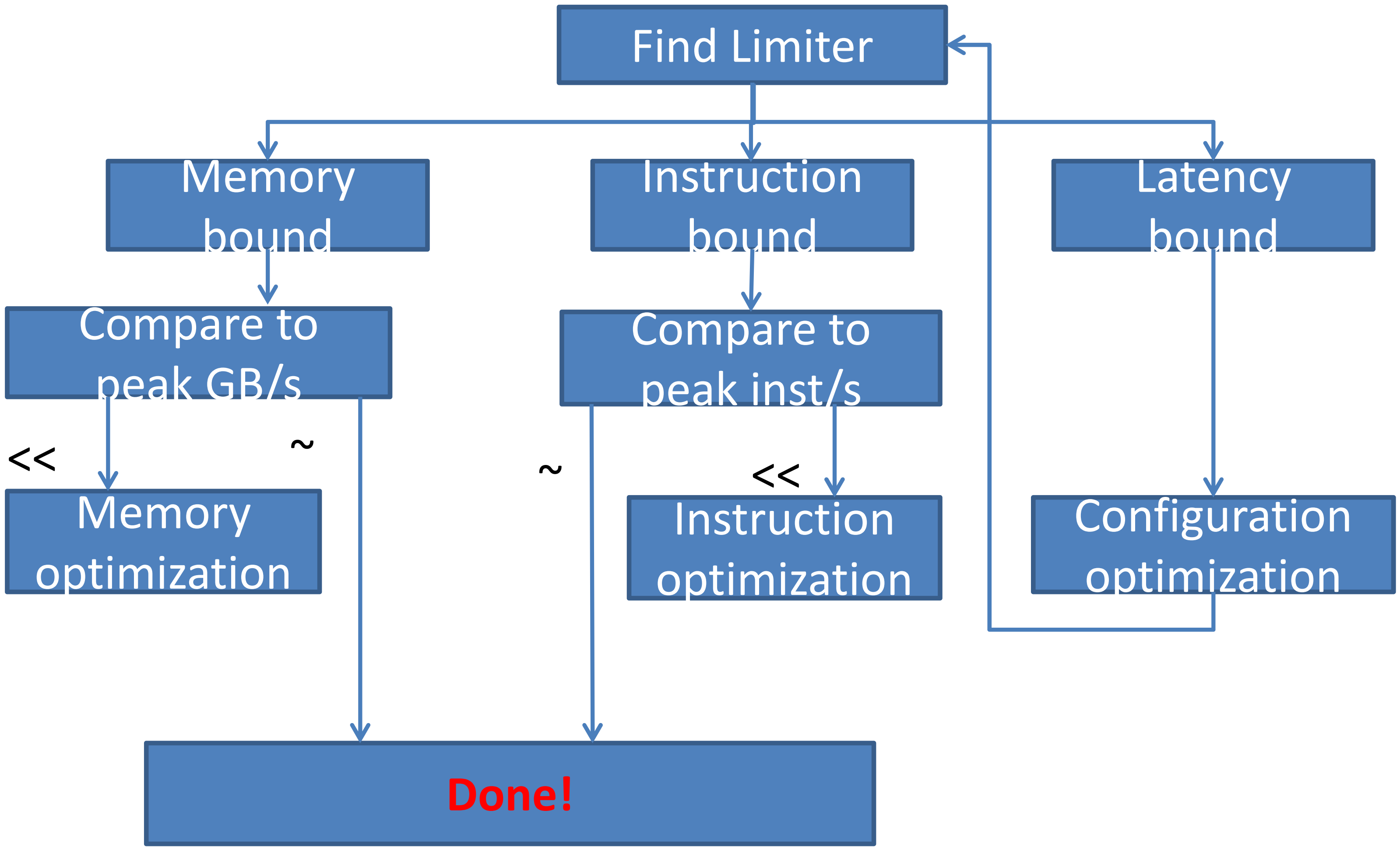
Outline

Optimization

- **General guideline II**
- **CPU-GPU Interaction**
- **Kepler in detail**

GENERAL GUIDELINE II

Kernel Optimization Workflow



General Optimization Strategies: Measurement

- **Find out the limiting factor in kernel performance**
  - Memory bandwidth bound (memory optimization)
  - Instruction throughput bound (instruction optimization)
  - Latency bound (configuration optimization)

- **Measure effective memory/instruction throughput**

Memory Optimization

- **If the code is memory-bound and effective memory throughput is much lower than the peak**
- **Purpose: access only data that are absolutely necessary**
- **Major techniques**
  - Improve access pattern to reduce wasted transactions
  - Reduce redundant access: read-only cache, shared memory

Instruction Optimization

- **If you find out the code is instruction bound**
  - Compute-intensive algorithm can easily become memory-bound if not careful enough
  - Typically, worry about instruction optimization after memory and execution configuration optimizations
- **Purpose: reduce instruction count**
  - Use less instructions to get the same job done
- **Major techniques**
  - Use high throughput instructions (ex. wider load)
  - Reduce wasted instructions: branch divergence, reduce replay (conflict), etc.

Latency Optimization

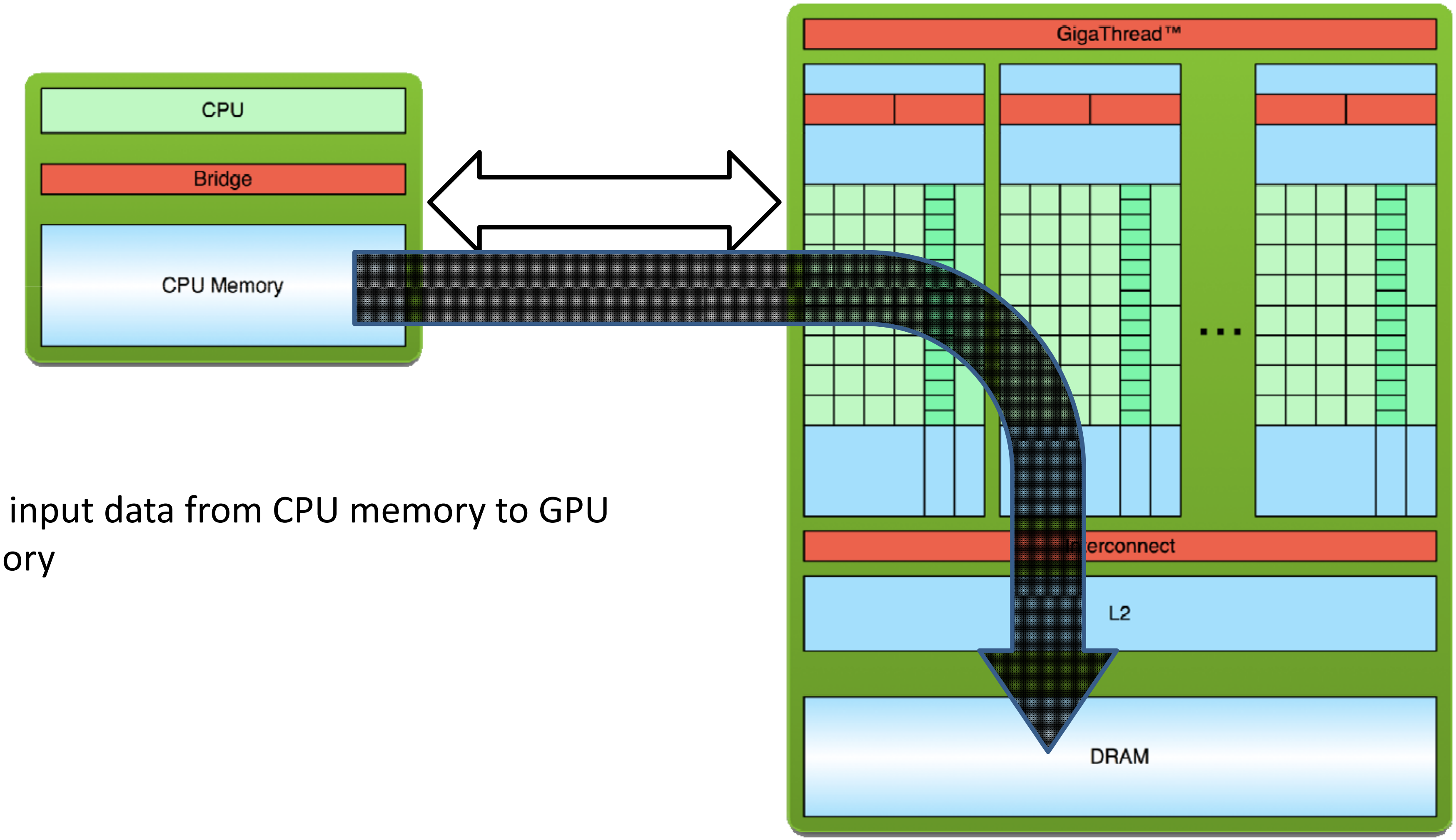
- **When the code is latency bound**
  - Both the memory and instruction throughputs are far from the peak
- **Latency hiding: switching threads**
  - A thread blocks when one of the operands isn't ready
- **Purpose: have enough warps to hide latency**
- **Major techniques: increase active warps, increase ILP**

CPU-GPU INTERACTION

Minimize CPU-GPU data transfer

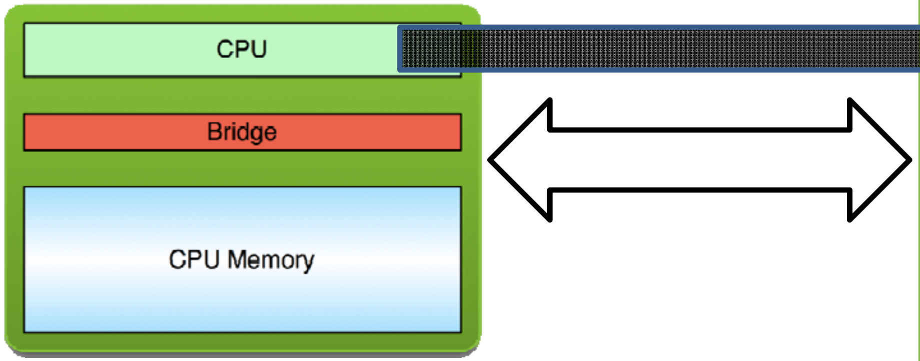
- **Host<->device data transfer has much lower bandwidth than global memory access.**
  - 16 GB/s (PCIe x16 Gen3) vs 250 GB/s & 3.95 Tinst/s (GK110)
- **Minimize transfer**
  - Intermediate data can be allocated, operated, de-allocated directly on GPU
  - Sometimes it's even better to recompute on GPU
  - Move CPU codes to GPU that do not have performance gains if it can reduce data transfer
- **Group transfer**
  - One large transfer much better than many small ones
  - Overlap memory transfer with computation

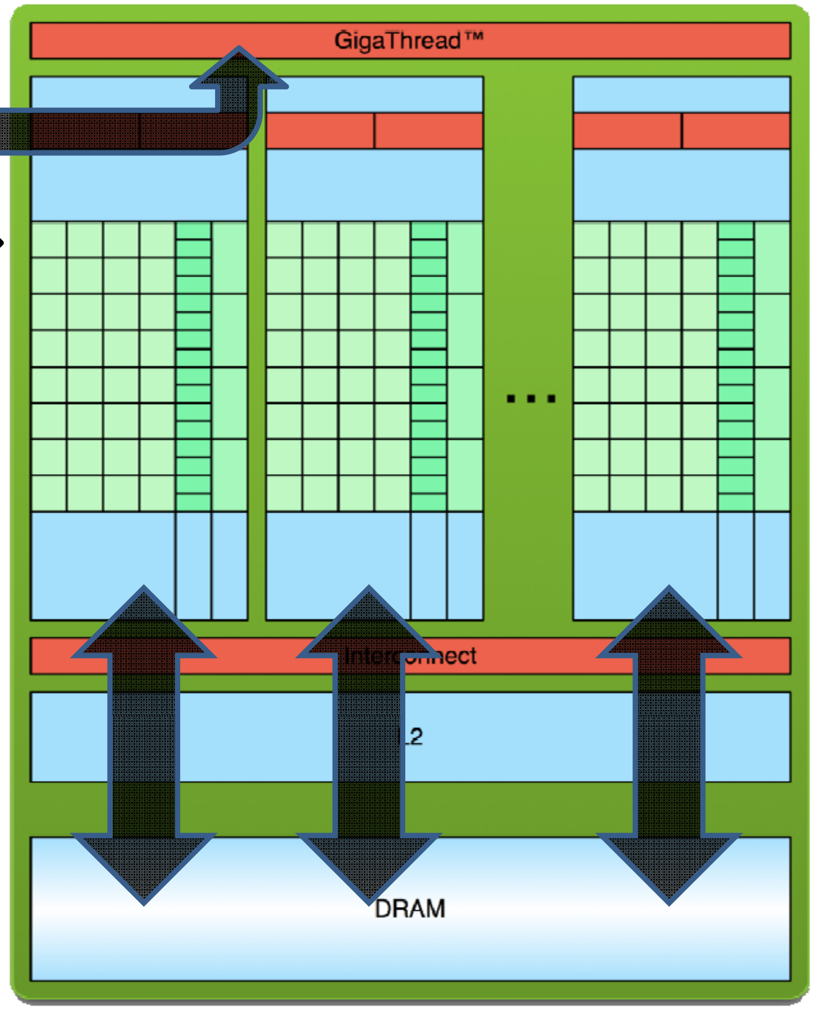
Revisit GPU Processing Flow



1. Copy input data from CPU memory to GPU memory

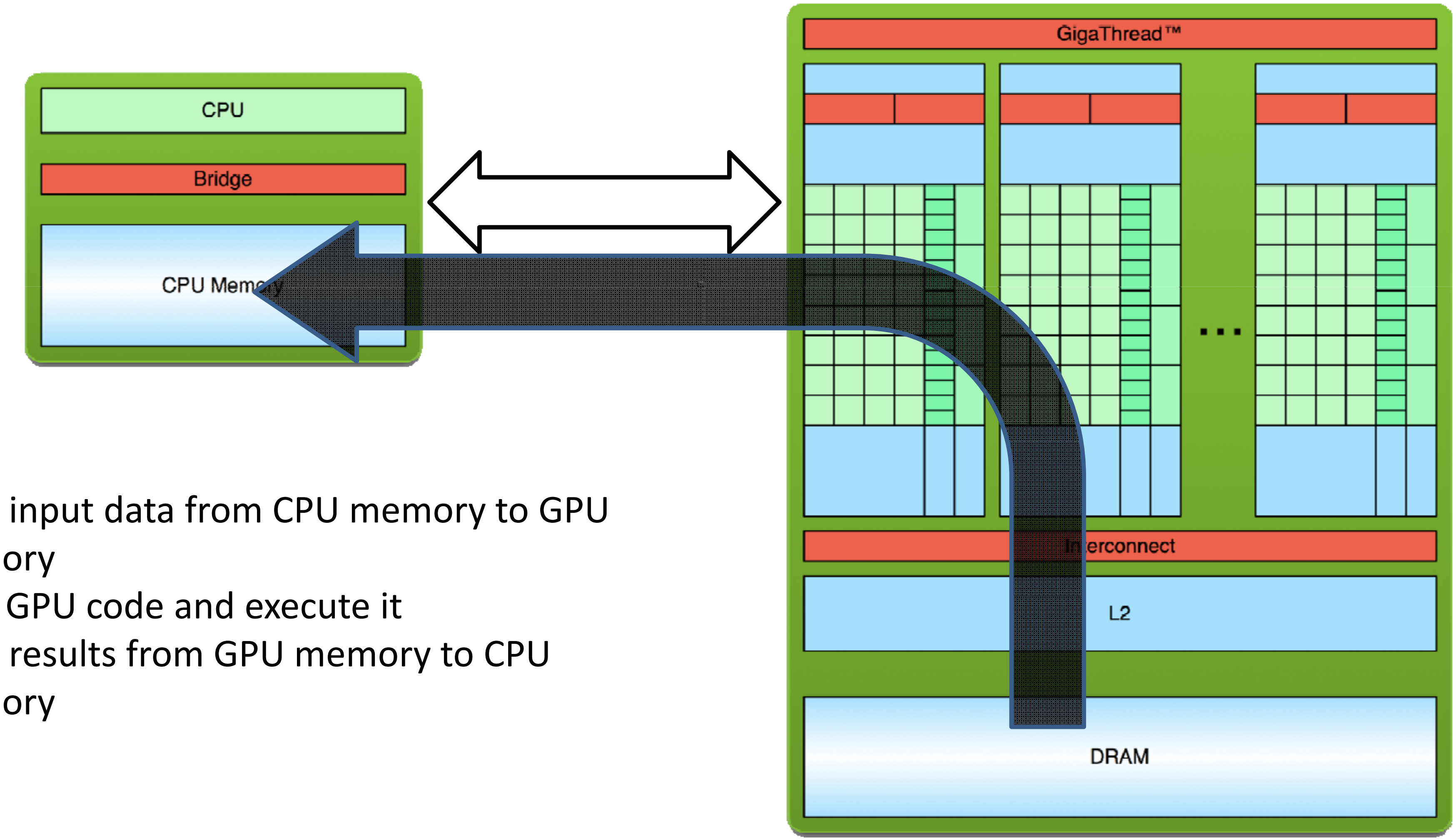
Revisit GPU Processing Flow





1. Copy input data from CPU memory to GPU memory
2. Load GPU code and execute it

Revisit GPU Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU code and execute it
3. Copy results from GPU memory to CPU memory

CUDA

- $T_{total} = T_{HtoD} + T_{Exec} + T_{DtoH}$

- **More Overlap?**

CUDA

Stream Example

```
cudaStreamCreate(&stream1);

cudaMemcpyAsync(dst1, src1, size, cudaMemcpyHostToDevice, stream1);

kernel<<<grid, block, 0, stream1>>>(…);

cudaMemcpyAsync(dst1, src1, size, stream1);

cudaStreamSynchronize(stream1);
```

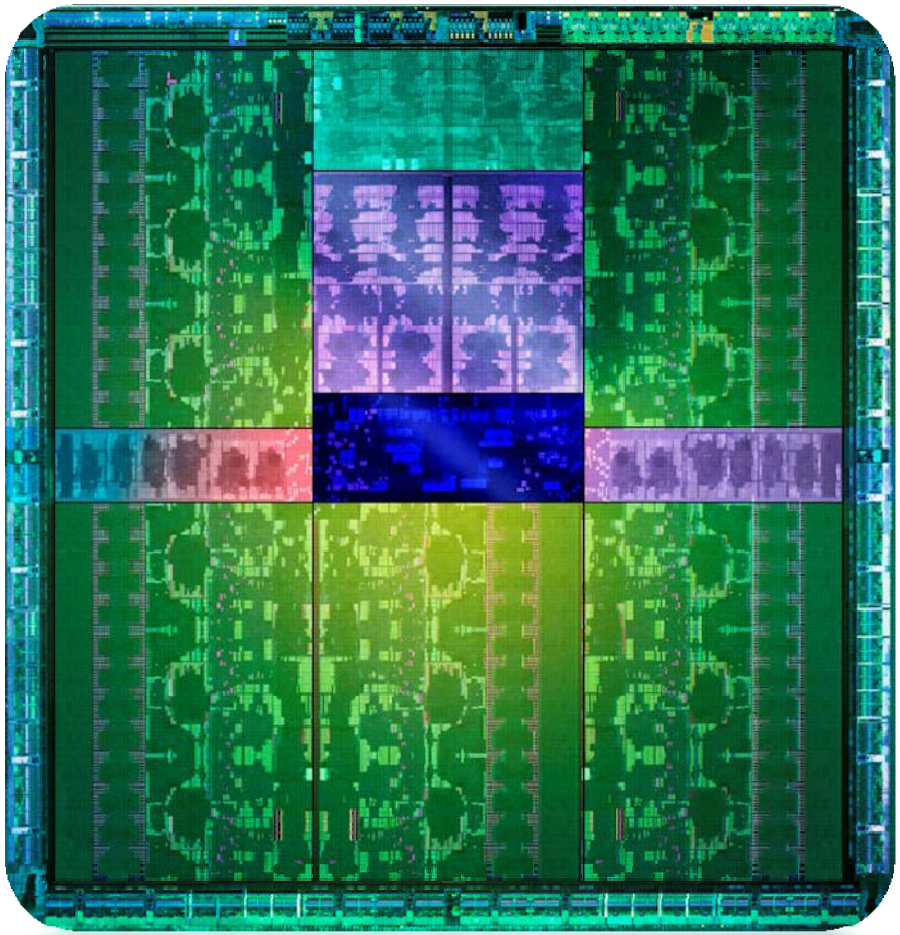
Stream Example

```
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync(dst1, src1, size, cudaMemcpyHostToDevice, stream1);
cudaMemcpyAsync(dst2, src2, size, cudaMemcpyHostToDevice,stream2);
kernel<<<grid, block, 0, stream1>>>(…);
kernel<<<grid, block, 0, stream2>>>(…);
cudaMemcpyAsync(dst1, src1, size, cudaMemcpyDeviceToHost, stream1);
cudaMemcpyAsync(dst2, src2, size, cudaMemcpyDeviceToHost, stream2);
cudaStreamSynchronize(stream1);
cudaStreamSynchronize(stream2);
```

KEPLER IN DETAIL

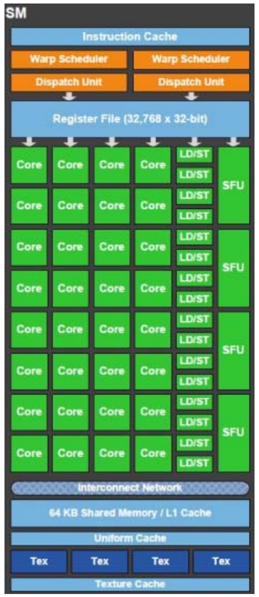
Kepler

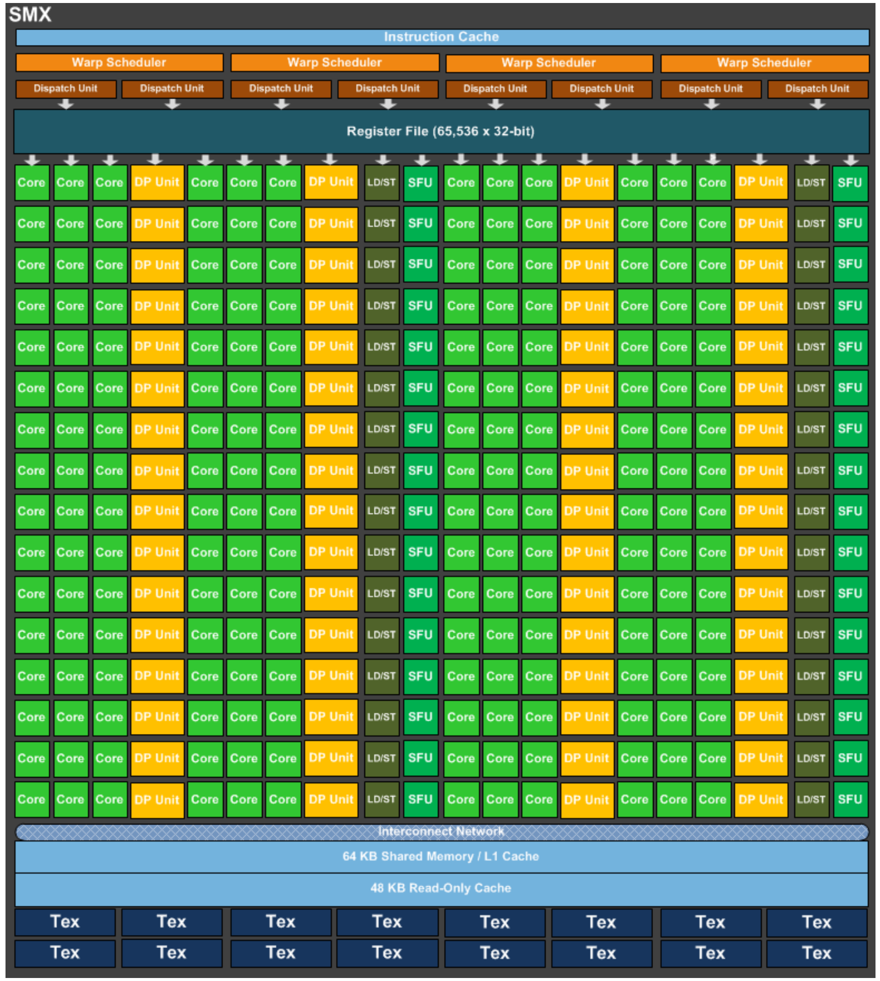
- **NVIDIA Kepler**
  - 1.31 tflops double precision
  - 3.95 tflops single precision
  - 250 gb/sec memory bandwidth
  - 2,688 Functional Units (cores)
- **~= #1 on Top500 in 1997**



NVIDIA GK110 - Kepler

Kepler GK110 SMX vs Fermi SM





3x perf

Power goes down!

New ISA Encoding: 255 Registers per Thread

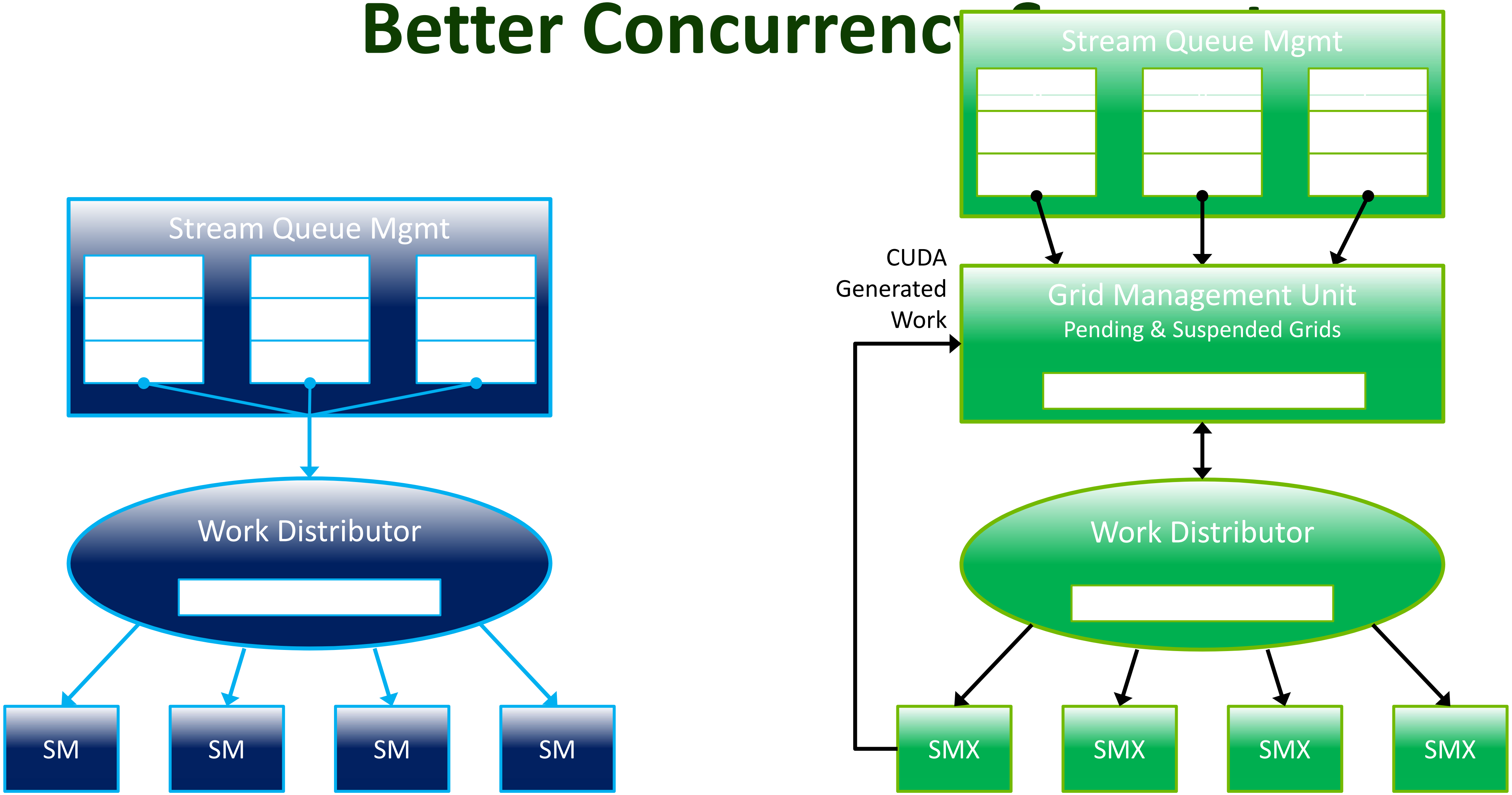
- **Fermi limit: 63 registers per thread**
  - A common Fermi performance limiter
  - Leads to excessive spilling
- **Kepler : Up to 255 registers per thread**
  - Especially helpful for FP64 apps

Hyper-Q

- ***Feature of Kepler K20 GPUs to increase application throughput by enabling work to be scheduled onto the GPU in parallel***

- **Two ways to take advantage**
  - CUDA Streams – now they really are concurrent
  - CUDA Proxy for MPI – concurrent CUDA MPI processes on one GPU

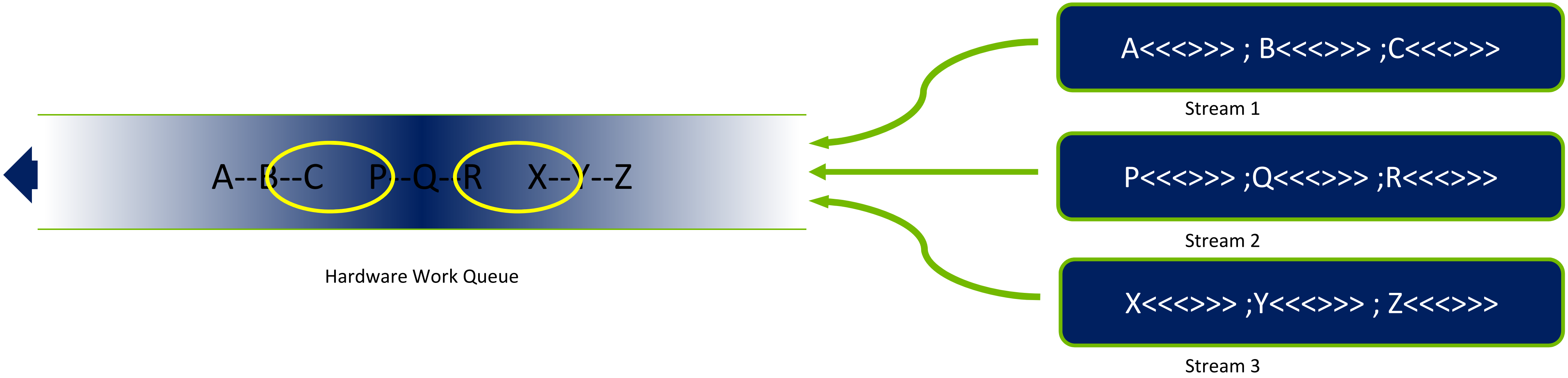
Better Concurrency



Fermi

Kepler GK110

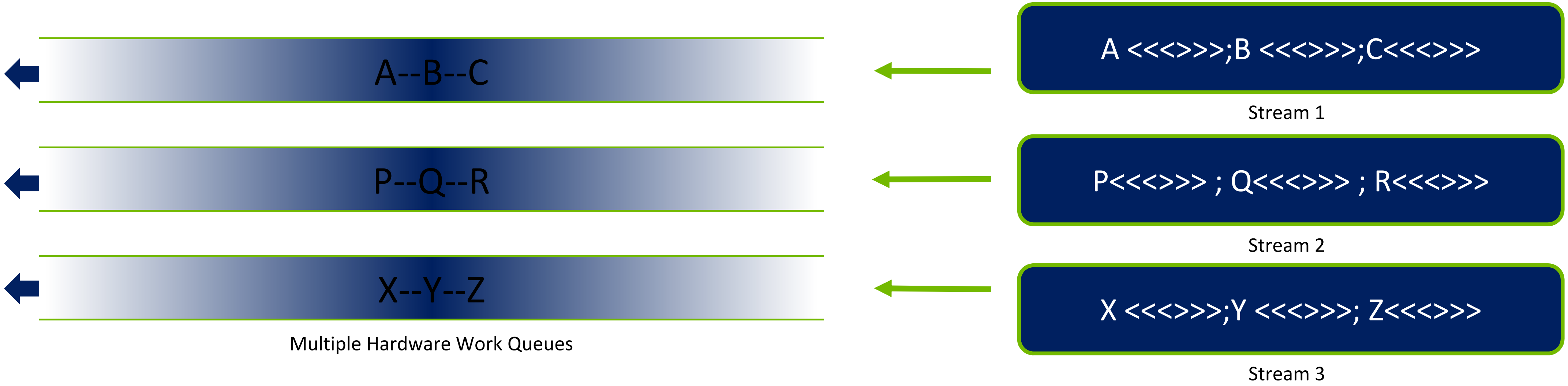
Fermi Concurrency



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

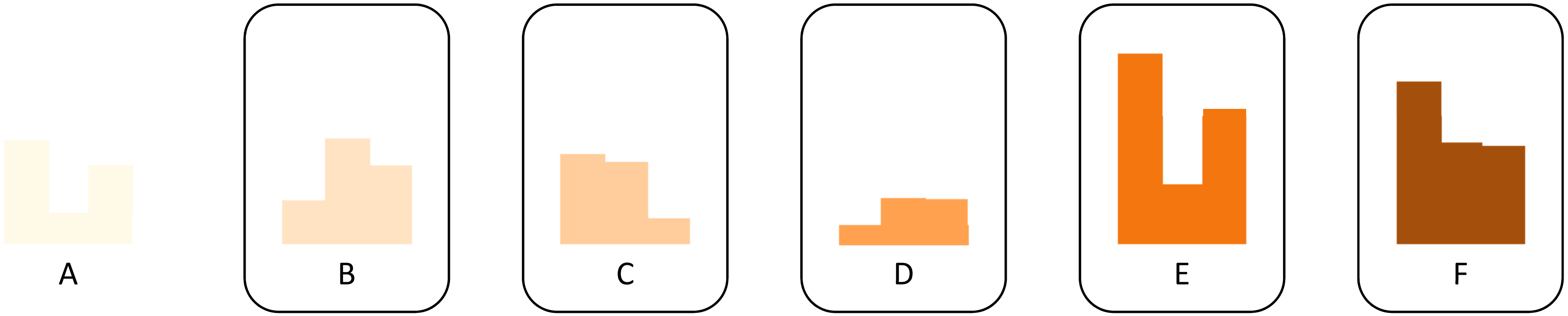
Kepler Improved Concurrency



Kepler allows 32-way concurrency

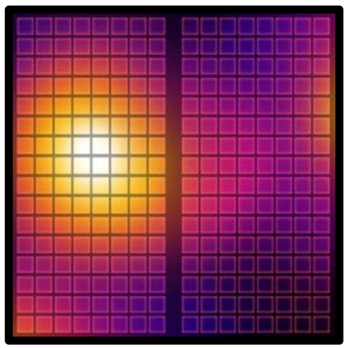
- One work queue per stream
- Concurrency at full-stream level
- No inter-stream dependencies

Fermi: Time-Division Multiprocess

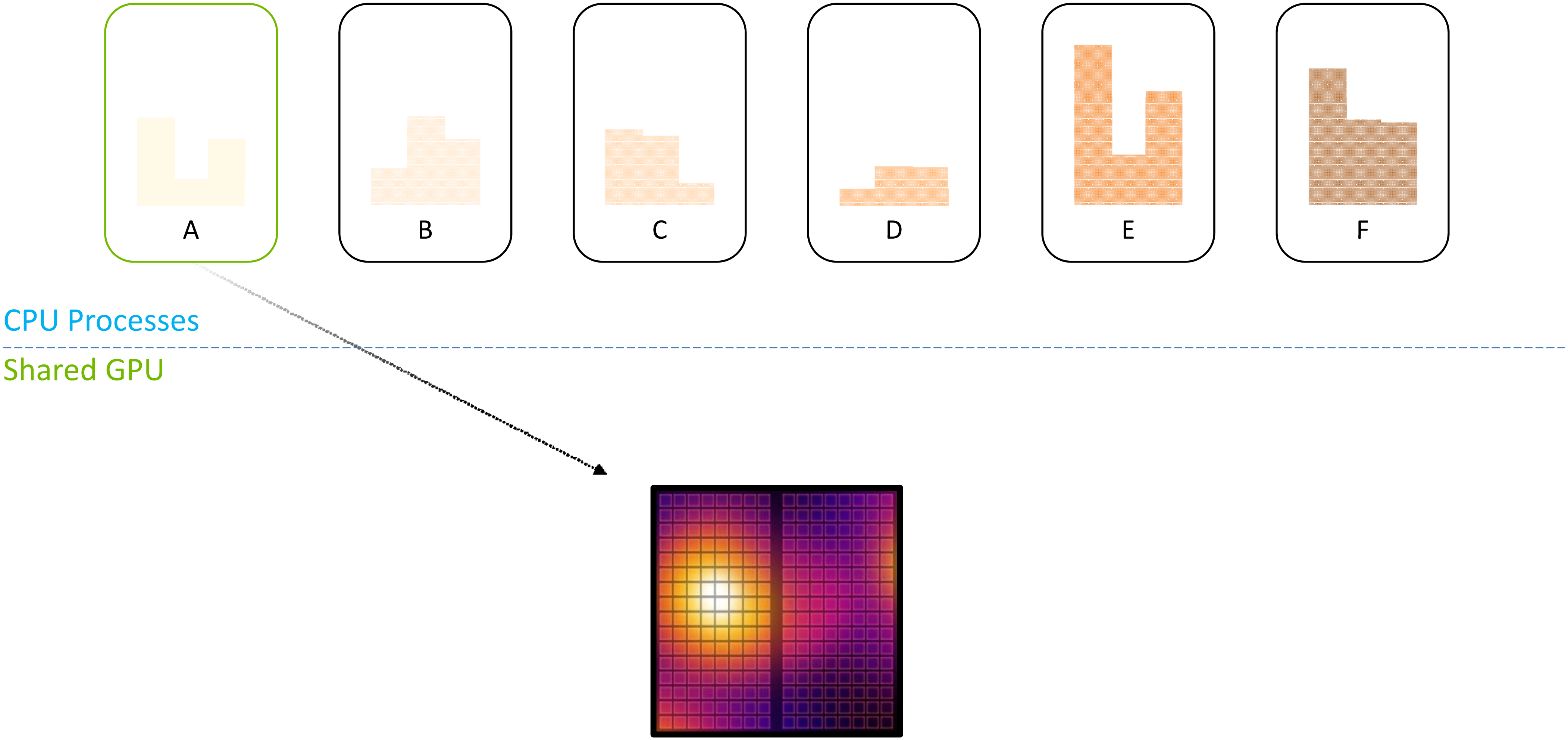


CPU Processes

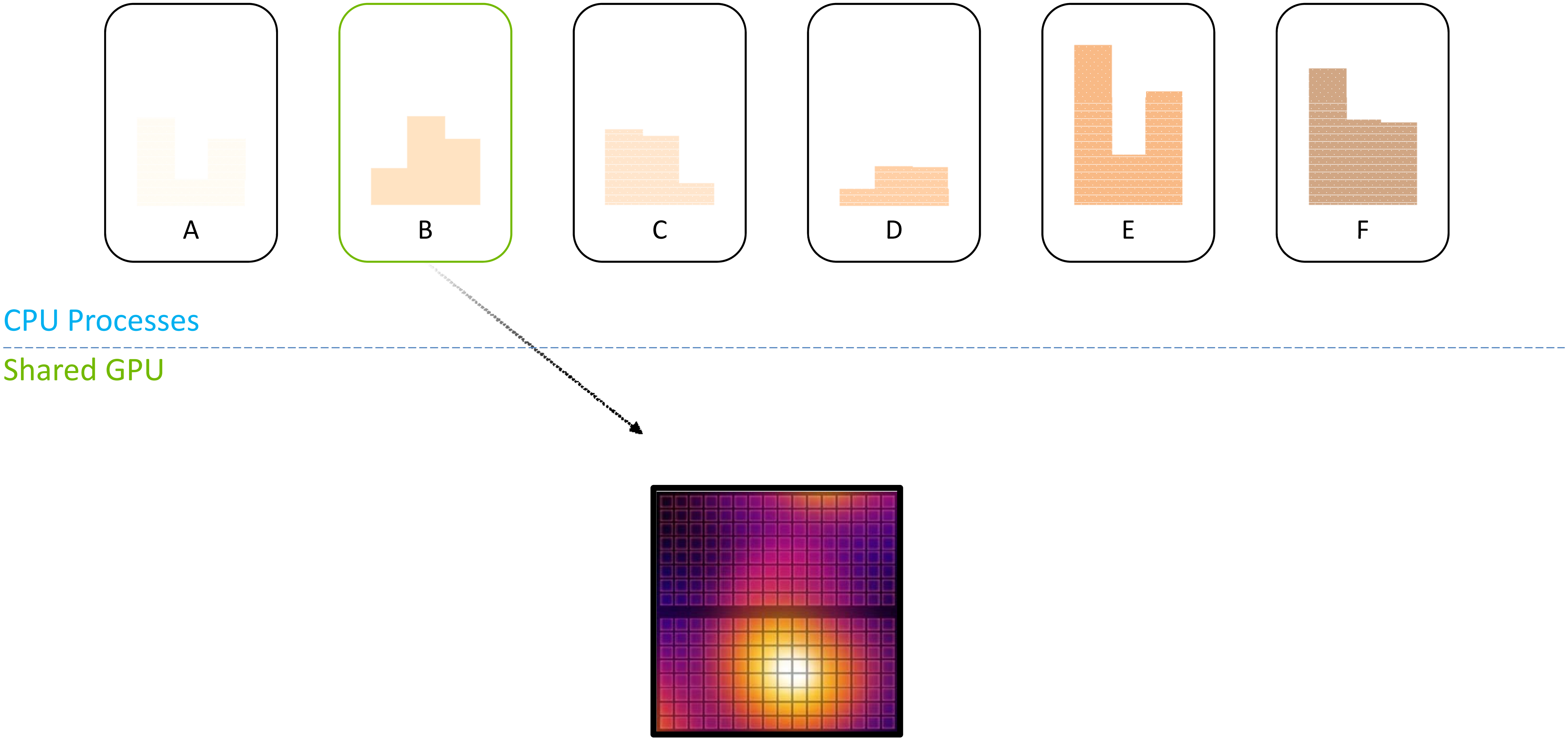
Shared GPU



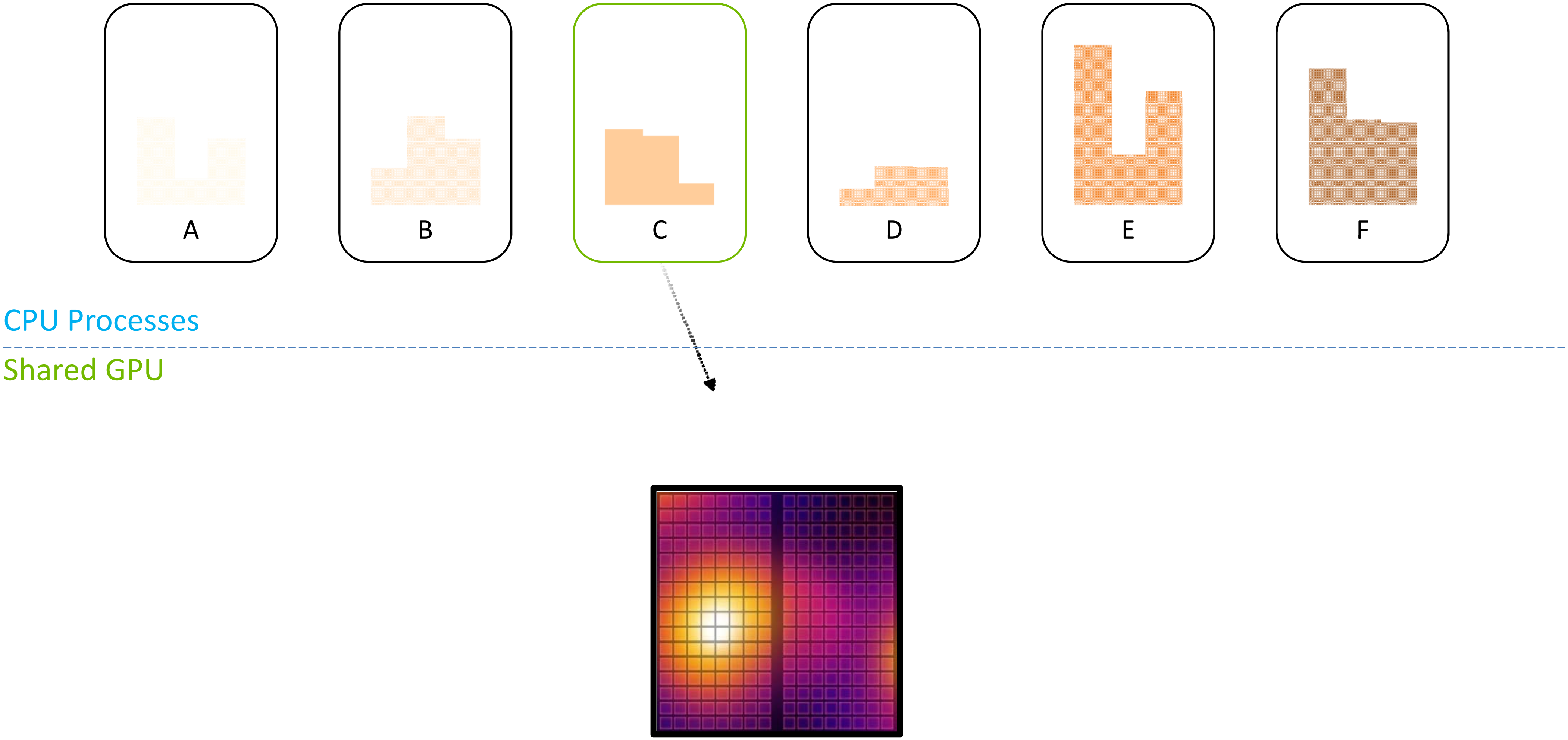
Fermi: Time-Division Multiprocess



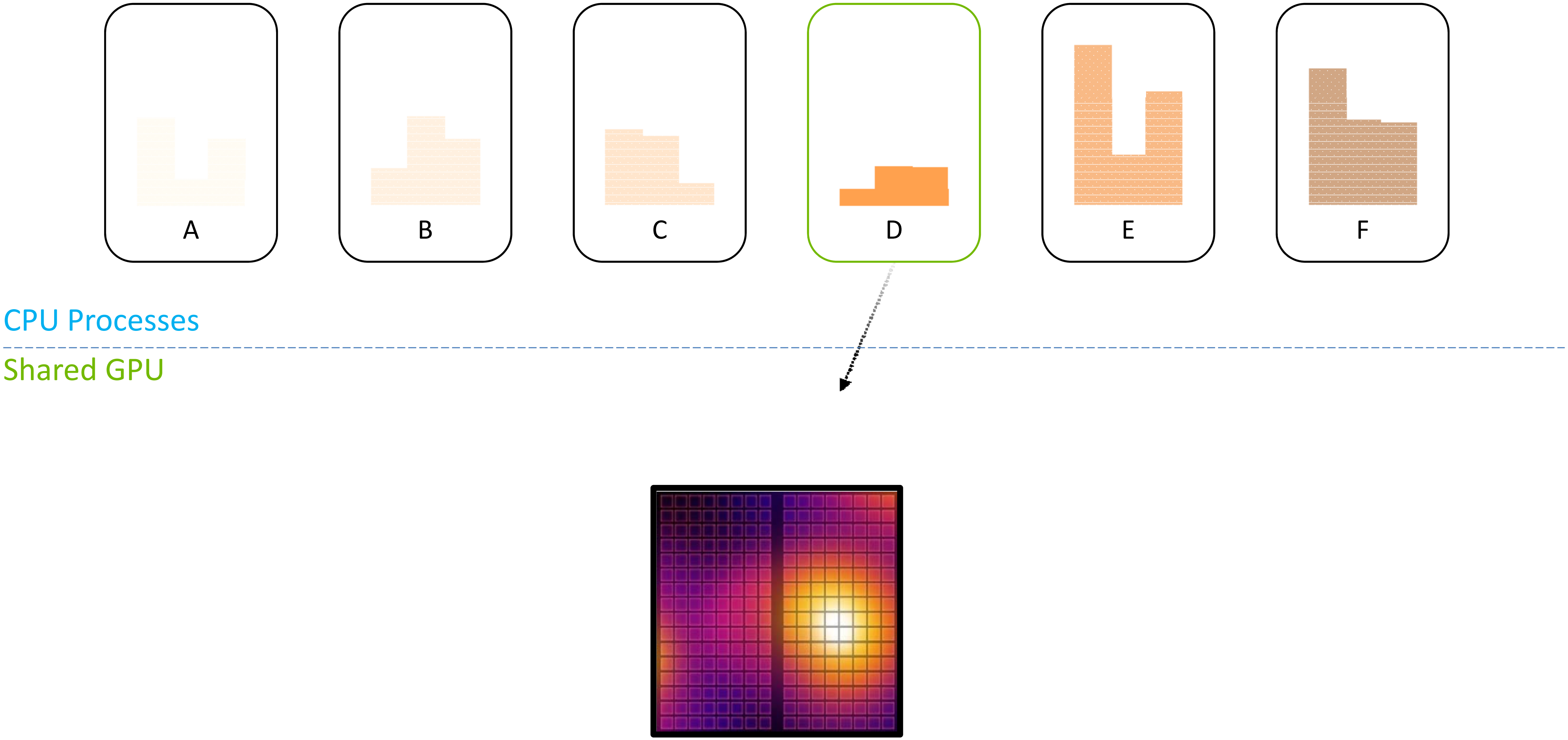
Fermi: Time-Division Multiprocess



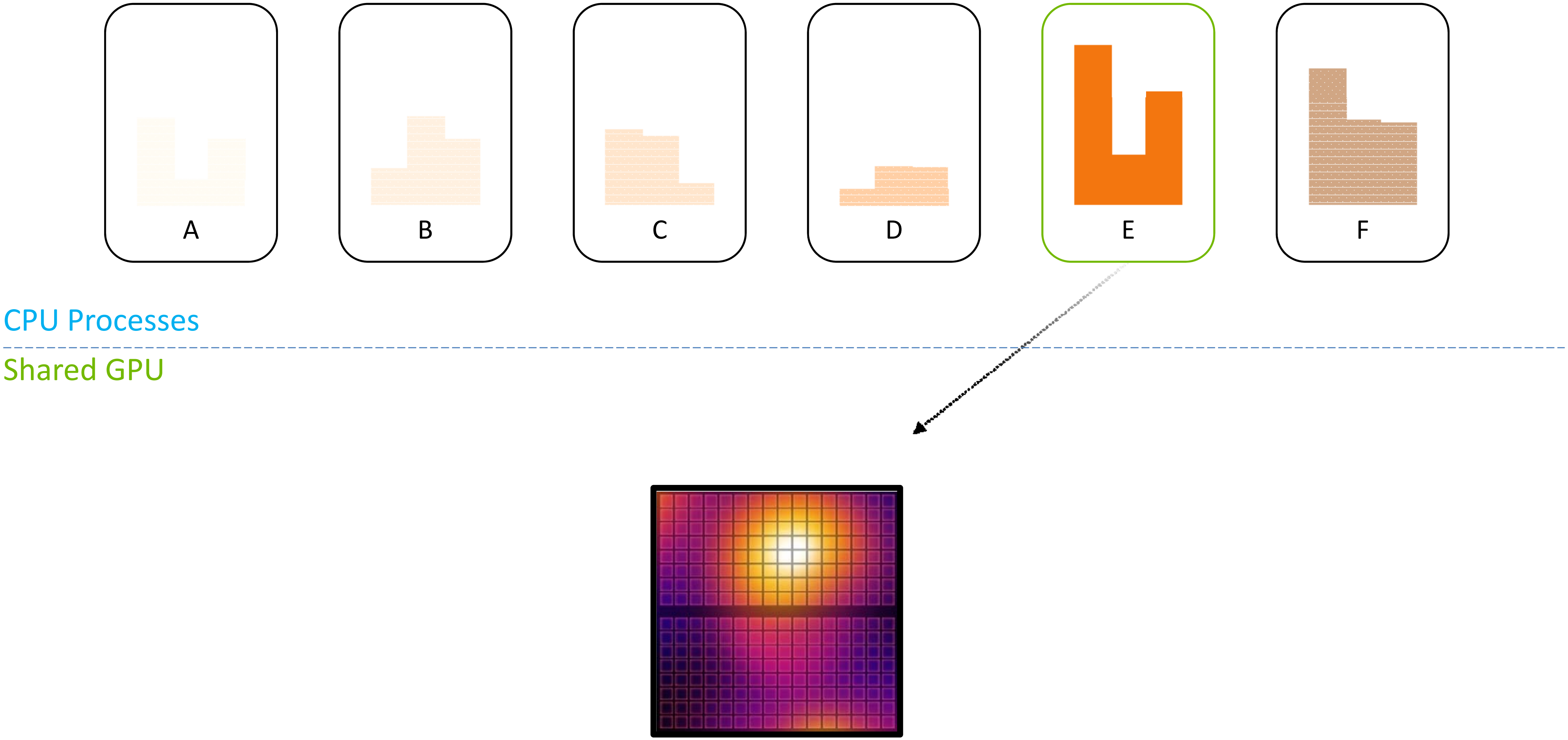
Fermi: Time-Division Multiprocess



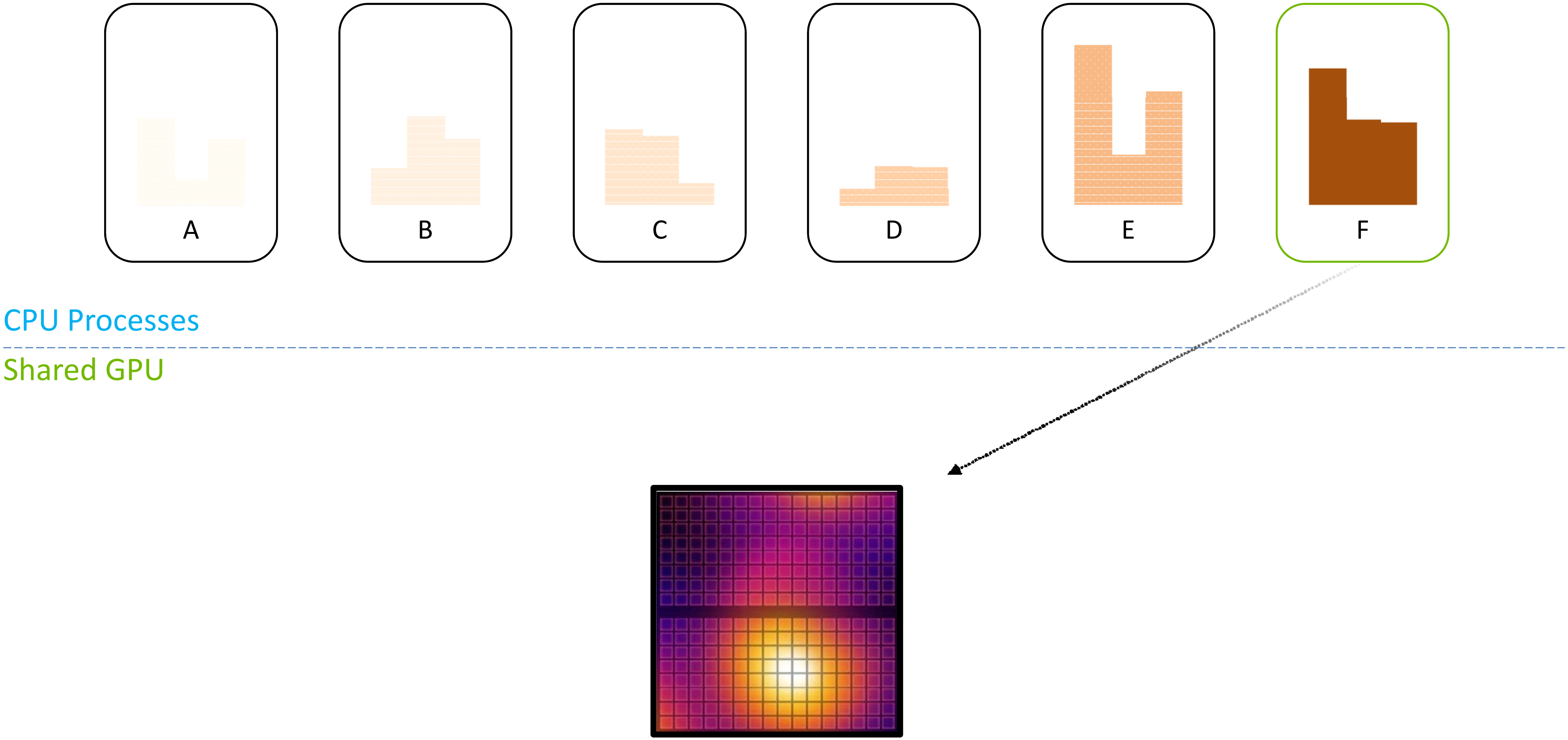
Fermi: Time-Division Multiprocess



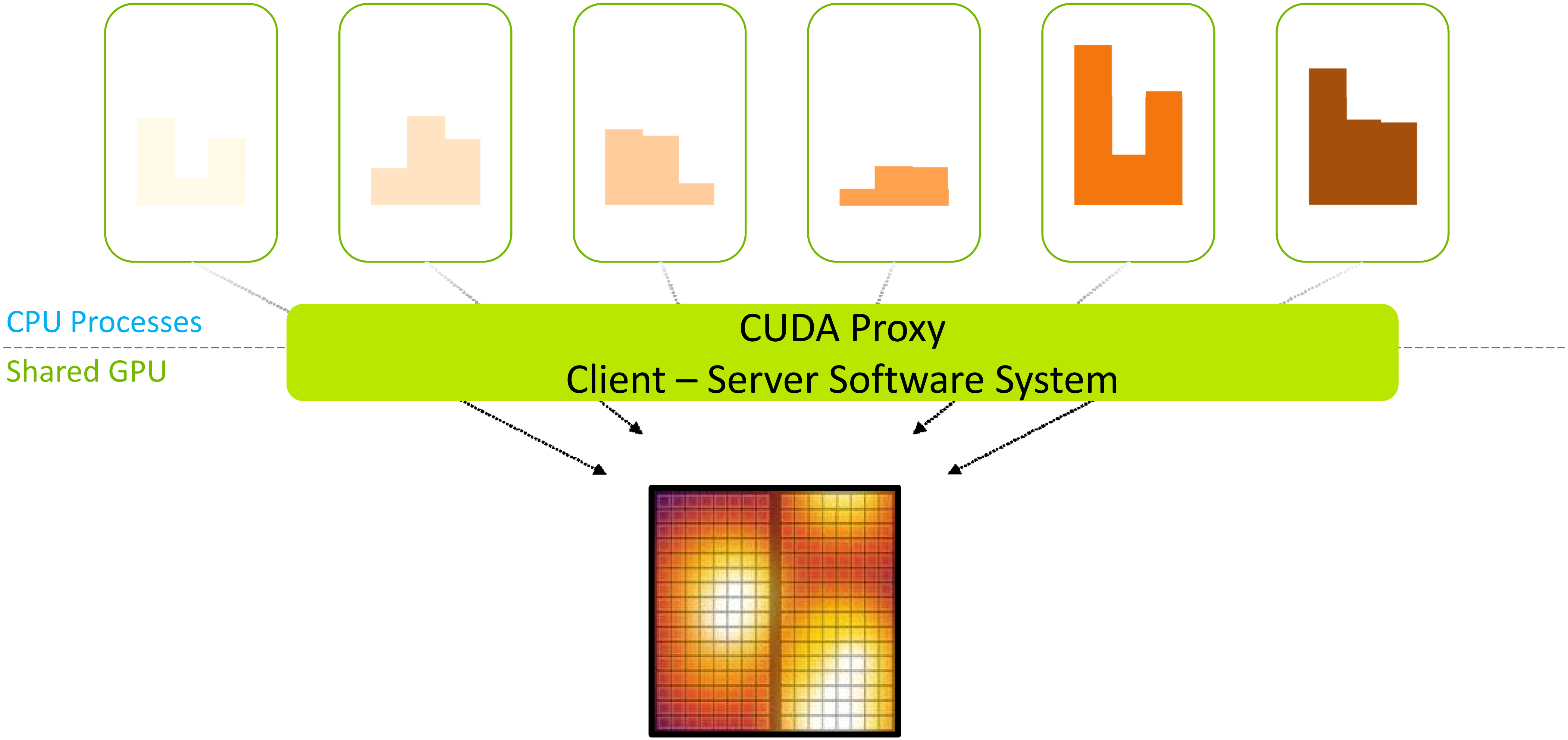
Fermi: Time-Division Multiprocess



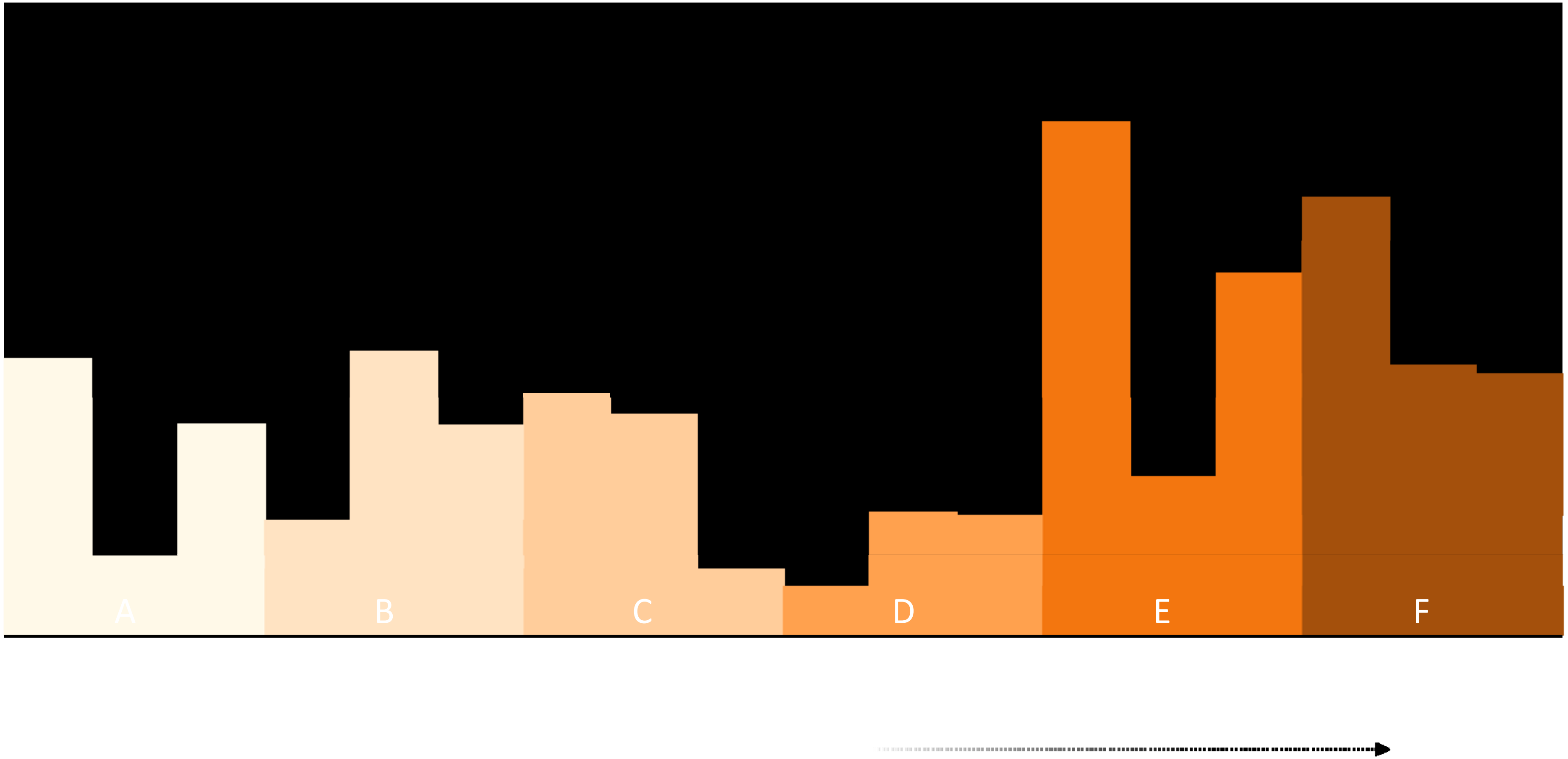
Fermi: Time-Division Multiprocess



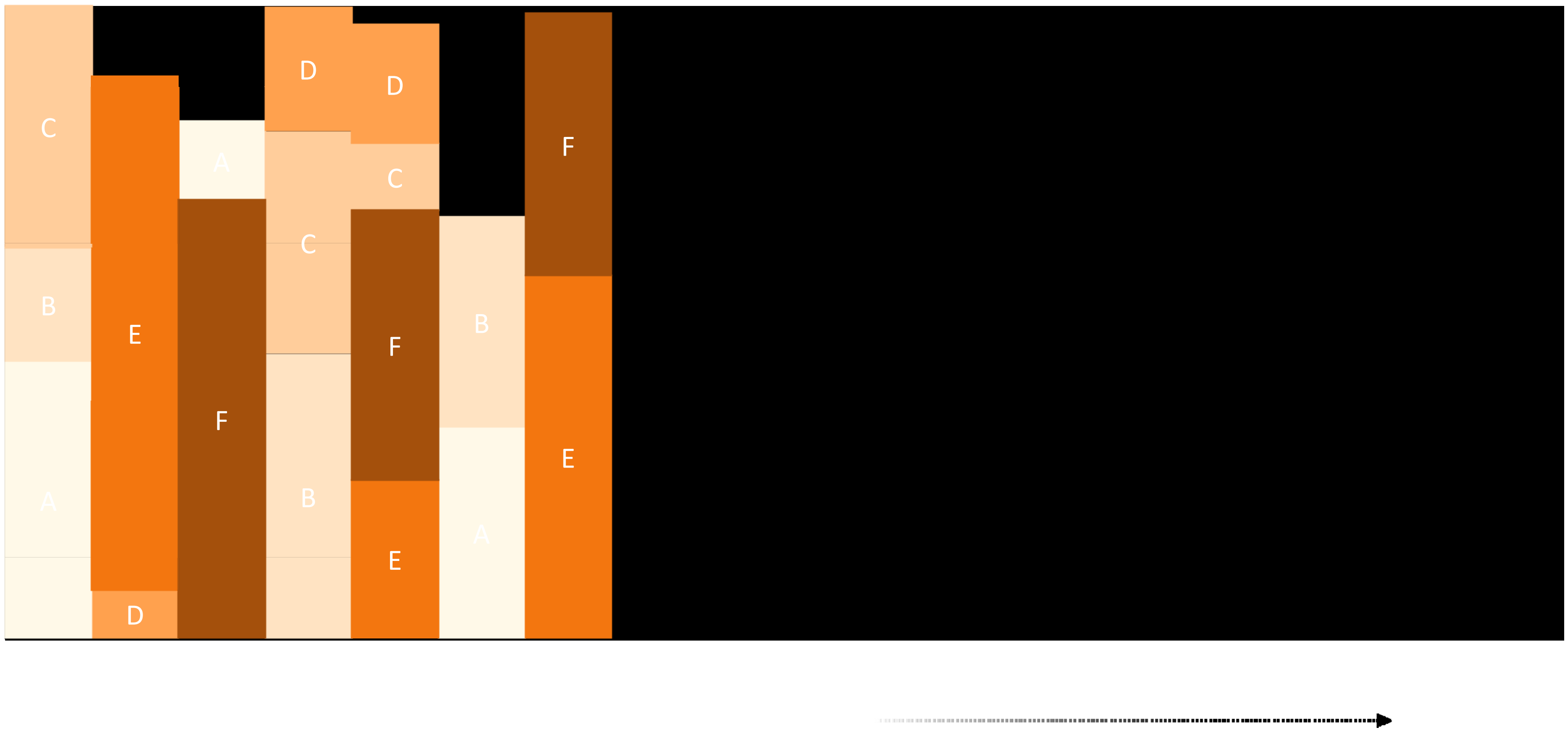
Hyper-Q: Simultaneous Multiprocess



Without Hyper-Q



With Hyper-Q



What is Dynamic Parallelism?

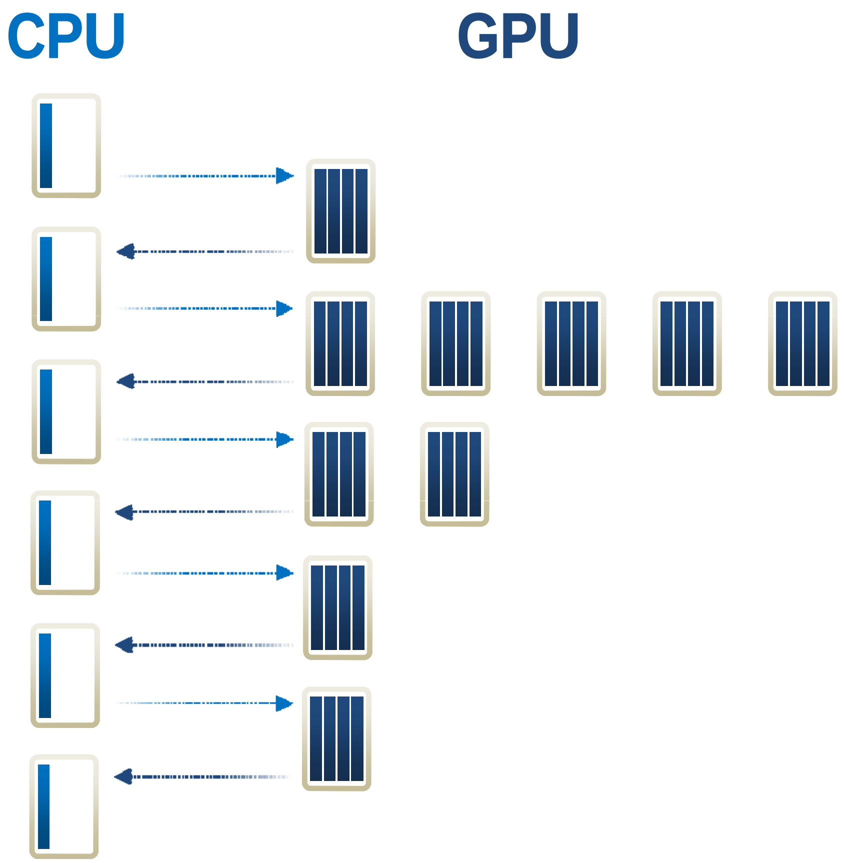
The ability to launch new kernels from the GPU

– Dynamically - based on run-time data
– Simultaneously - from multiple threads at once
– Independently - each thread can launch a different grid

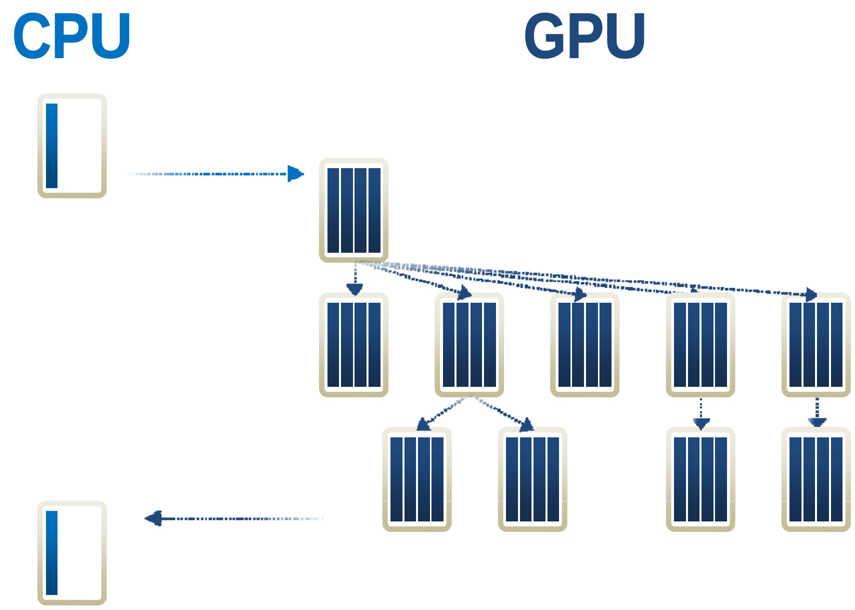
Fermi: Only CPU can generate GPU work

Kepler: GPU can generate work for itself

What Does It Mean?



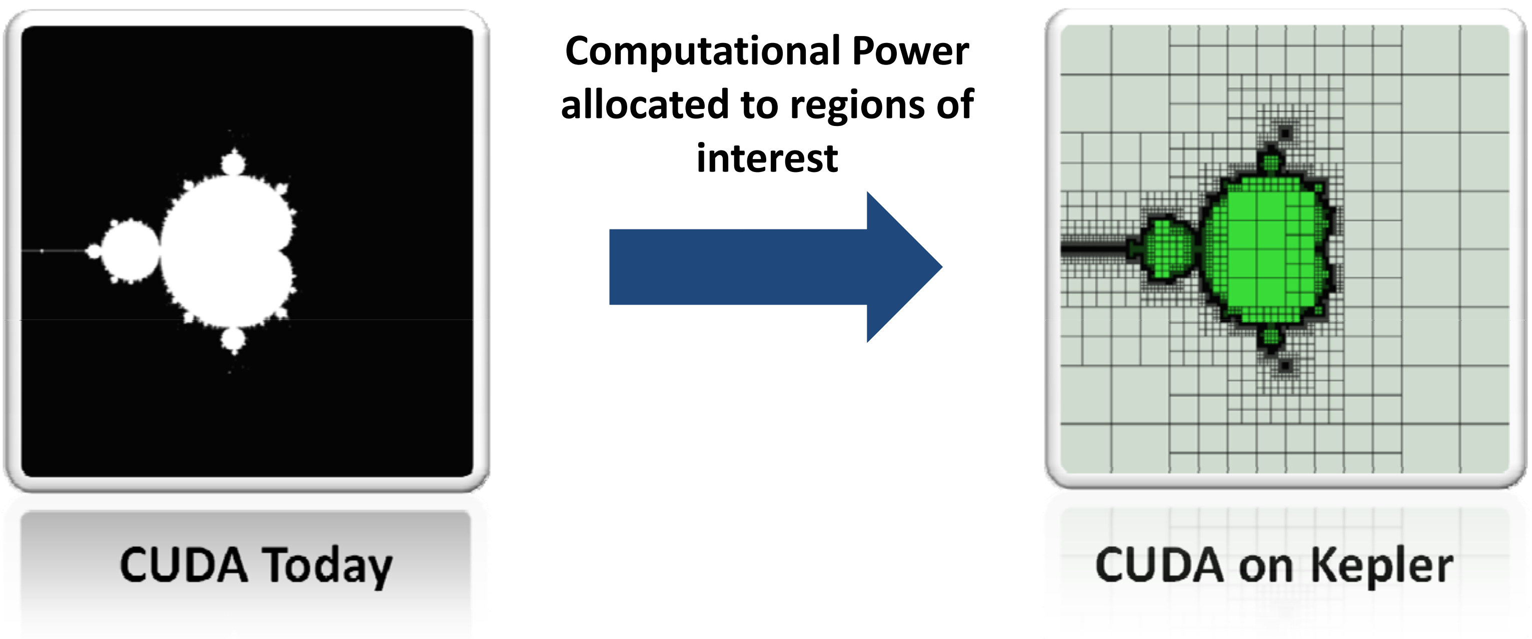
GPU as Co-Processor



Autonomous, Dynamic Parallelism

New Types of Algorithms

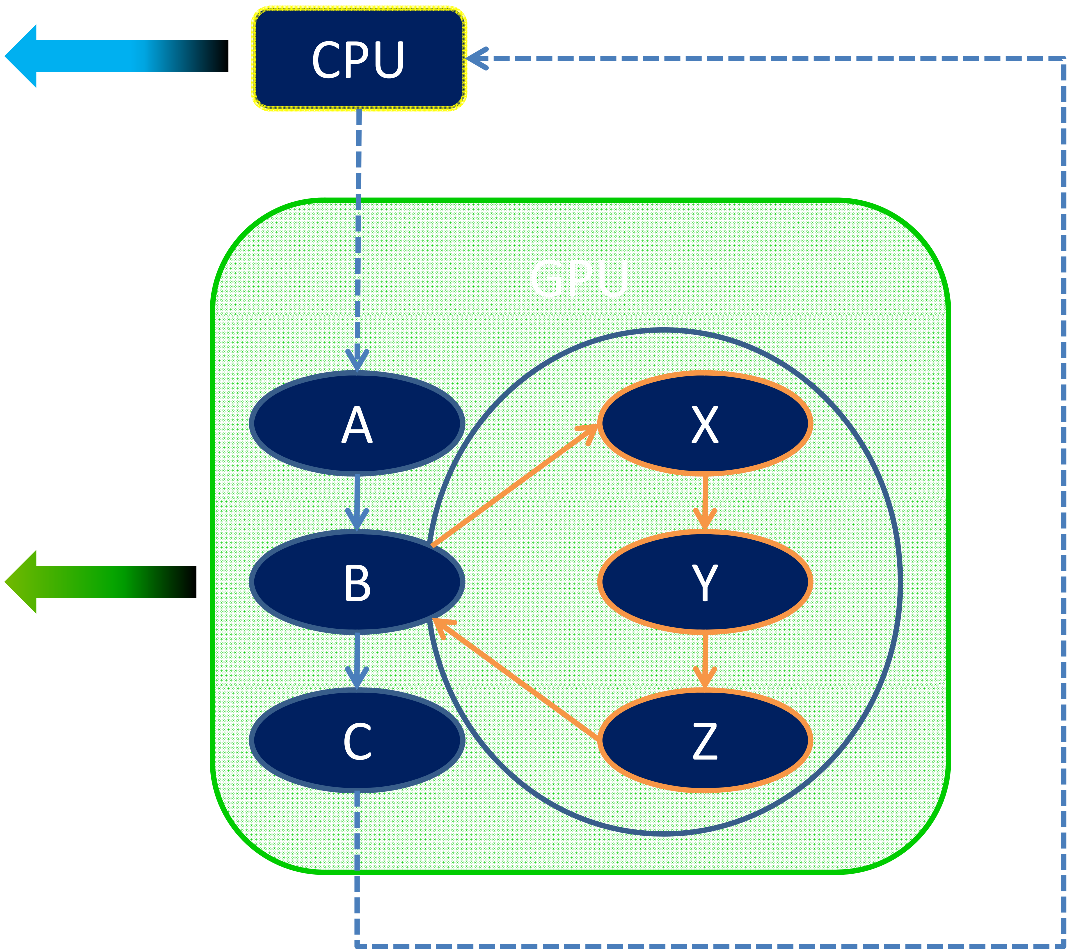
- **Recursive Parallel Algorithms like Quick sort**
- **Adaptive Mesh Algorithms like Mandelbrot**



Familiar Programming Model

```
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);
}
```



Programming Model

- Launch is per-thread and asynchronous

Code Example

```
__device__ float buf[1024];
__global__ void cnp(float *data)
{
    int tid = threadIdx.x;
    if(tid % 2)
        buf[tid/2] = data[tid]+data[tid+1];

    __syncthreads();

    if(tid == 0) {
        launch<<< 128, 256 >>>(buf);
        cudaDeviceSynchronize();
    }
    __syncthreads();

    cudaMemcpyAsync(data, buf, 1024);
    cudaDeviceSynchronize();
}
```

Programming Model

- Launch is per-thread
  and asynchronous

- CUDA primitives are per-block
- launched kernels and CUDA objects like streams are visible to all threads in a thread block
- cannot be passed to child kernel

Code Example

```
__device__ float buf[1024];
__global__ void cnp(float *data)
{
    int tid = threadIdx.x;
    if(tid % 2)
        buf[tid/2] = data[tid]+data[tid+1];

    __syncthreads();

    if(tid == 0) {
        launch<<< 128, 256 >>>(buf);
        cudaDeviceSynchronize();
    }
    __syncthreads();

    cudaMemcpyAsync(data, buf, 1024);
    cudaDeviceSynchronize();
}
```

Programming Model

- Launch is per-thread and asynchronous
- CUDA primitives are per-block
- Sync includes all launches by any thread in the block

Code Example

```
__device__ float buf[1024];
__global__ void cnp(float *data)
{
    int tid = threadIdx.x;
    if(tid % 2)
        buf[tid/2] = data[tid]+data[tid+1];

    __syncthreads();

    if(tid == 0) {
        launch<<< 128, 256 >>>(buf);
        cudaDeviceSynchronize();
    }
    __syncthreads();

    cudaMemcpyAsync(data, buf, 1024);
    cudaDeviceSynchronize();
}
```

Programming Model

- Launch is per-thread and asynchronous
- CUDA primitives are per-block
- Sync includes all launches by any thread in the block
- cudaDeviceSynchronize() does not imply syncthreads()

Code Example

```
__device__ float buf[1024];
__global__ void cnp(float *data)
{
    int tid = threadIdx.x;
    if(tid % 2)
        buf[tid/2] = data[tid]+data[tid+1];

    __syncthreads();

    if(tid == 0) {
        launch<<< 128, 256 >>>(buf);
        cudaDeviceSynchronize();
    }
    __syncthreads();

    cudaMemcpyAsync(data, buf, 1024);
    cudaDeviceSynchronize();
}
```

Memory Model

- Launch implies membar
  (child sees parent state at time of launch)

Code Example

```
__device__ float buf[1024];
__global__ void cnp(float *data)
{
    int tid = threadIdx.x;
    if(tid % 2)
        buf[tid/2] = data[tid]+data[tid+1];

    __syncthreads();

    if(tid == 0) {
        launch<<< 128, 256 >>>(buf);
        cudaDeviceSynchronize();
    }
    __syncthreads();

    cudaMemcpyAsync(data, buf, 1024);
    cudaDeviceSynchronize();
}
```

Memory Model

- Launch implies membar
  (child sees parent state at time of launch)
- Sync implies invalidate
  (parent sees child writes after sync)

Code Example

```
__device__ float buf[1024];
__global__ void cnp(float *data)
{
    int tid = threadIdx.x;
    if(tid % 2)
        buf[tid/2] = data[tid]+data[tid+1];

    __syncthreads();

    if(tid == 0) {
        launch<<< 128, 256 >>>(buf);
        cudaDeviceSynchronize();
    }
    __syncthreads();

    cudaMemcpyAsync(data, buf, 1024);
    cudaDeviceSynchronize();
}
```

Memory Model

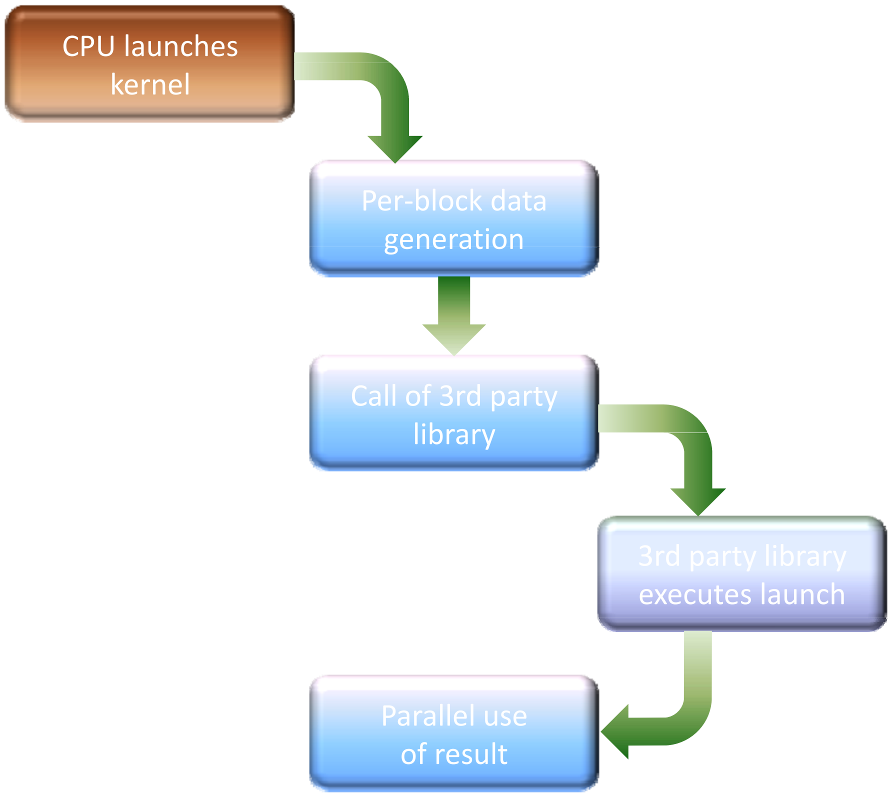
- Launch implies membar
  (child sees parent state at time of launch)
- Sync implies invalidate
  (parent sees child writes after sync)
- Local & shared memory are private
- Constants are immutable

Code Example

```
__device__ float buf[1024];
__global__ void cnp(float *data)
{
    int tid = threadIdx.x;
    if(tid % 2)
        buf[tid/2] = data[tid]+data[tid+1];

    __syncthreads();

    if(tid == 0) {
        launch<<< 128, 256 >>>(buf);
        cudaDeviceSynchronize();
    }
    __syncthreads();
    if (tid == 0) {
    cudaMemcpyAsync(data, buf, 1024);
    cudaDeviceSynchronize();
    }
```

Dynamic Parallelism and GPU Callable Libraries

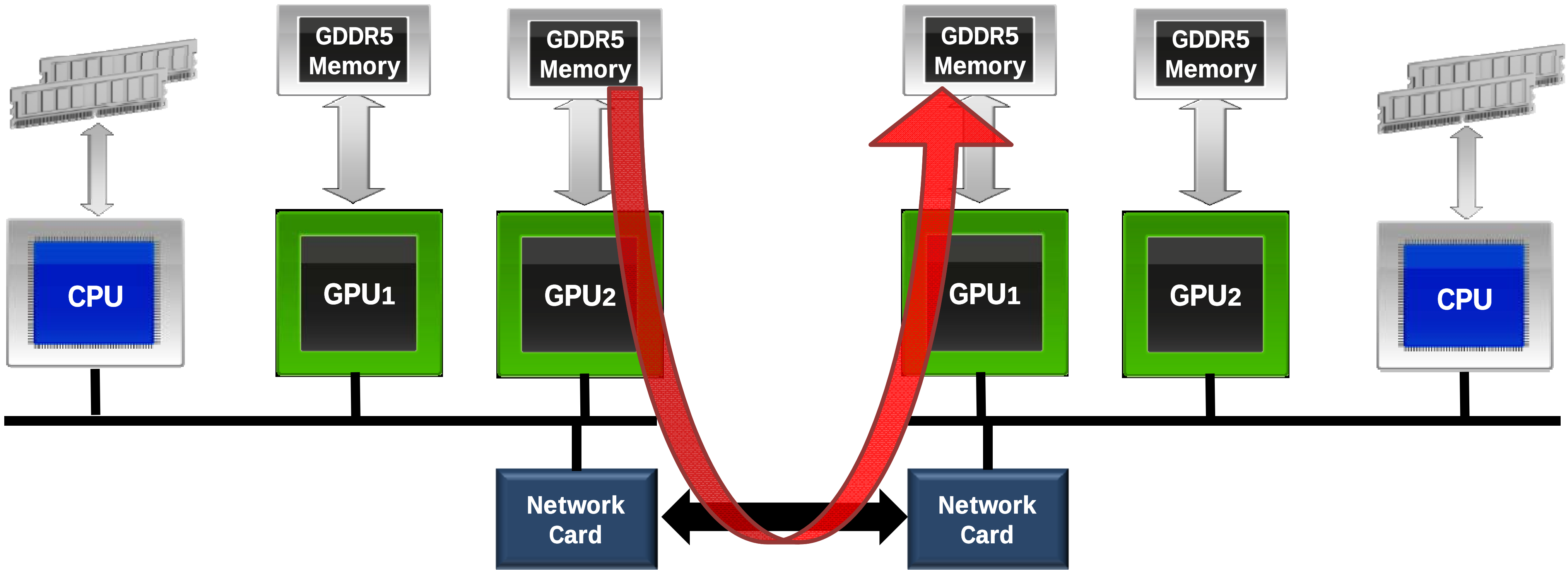
```
__global__ void libraryCall(float *a,
                            float *b,
                            float *c)
{
    // All threads generate data
    createData(a, b);
    __syncthreads();

    // Only one thread calls library
    if(threadIdx.x == 0) {
        cublasDgemm(a, b, c);
        cudaDeviceSynchronize();
    }

    // All threads wait for dtrsm
    __syncthreads();

    // Now continue
    consumeData(c);
}
```



NVIDIA GPUDirect™ RDMA

- ***Provides technology necessary to enable lower latency memory transfers between GPU and other PCIE devices without requiring custom hardware.***
- **API and documentation for device driver developers**
- **Available on Linux only**
- **Supported on Kepler Quadro and Telsa GPUs**

NVIDIA GPUDirect™ Now Supports RDMA



More threads are needed

- **2-3x throughput per clock per SM**
- **Memory bandwidth increasing**
- **Bigger SM have bigger stomach!**

More thread are needed

- **If you already launched enough threads, the following enhancement on kepler will ensure enough active warps on SMs.**
- **2x register file on each SM**
  - E.g. 63 registers per thread, blockDim 256
  - In Fermi 16 active warps
  - In Kepler 32 active warps
- **2x simultaneous blocks per SM**
  - E.g. 16 registers per thread, blockDim 96
  - In Fermi 96*8/32 = 24 active warps
  - In kepler 96*16/32 = 46 active warps
- **More flexible for shared memory configuration 16/32/48KB**

If one kernel can't launch enough threads

- **Concurrent Kernels**
  - GK110 allows up to 32 concurrent kernels to execute.
- **Hyper-Q**
  - Using MPI, Different processes can use the device at the same time.
  - Using Stream, there's no inter-stream dependencies any more.
- **Dynamic Parallelism**
  - Threads can launch kernels

Two GPUs on K10

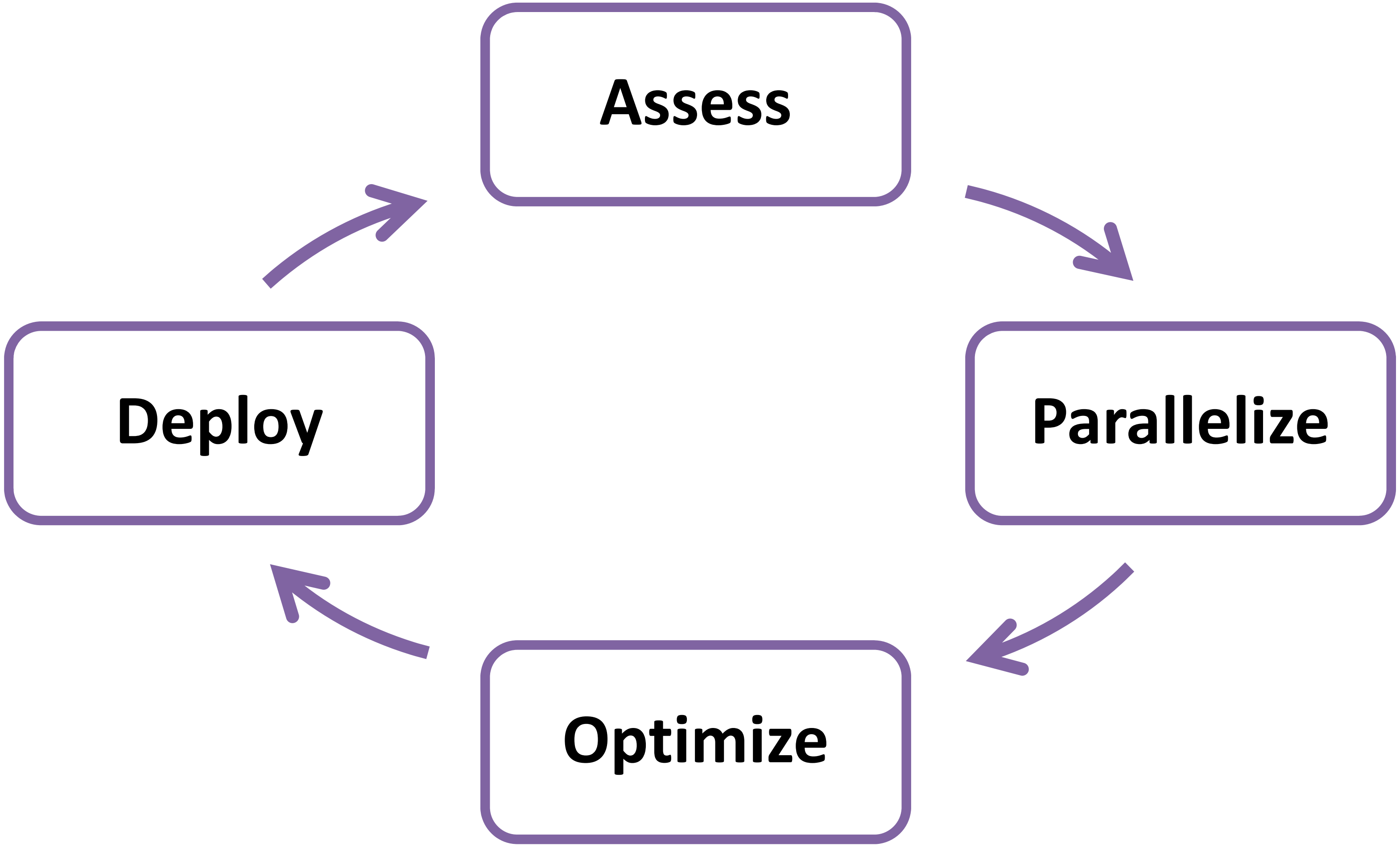
- K10 is a dual-GK104 Gemini board.
- Appear as two separate CUDA devices.
- Need multi-GPU paradigm.

New instructions for replacement

- **Communication in Shared memory -> shuffle**
  - Don’t need Shared memory
  - Lower latency
- **Complex reduction -> Fast Global Memory Atomics**
  - More easy way
- **L1 cache read -> Read-Only Data Cache**
  - L1 is reserved only for register spills and stack data
  - A separate pipe, relaxed memory coalescing rules

SUMMARY

APOD: A Systematic Path to Performance



Tools For Project

- **Linux SSH client**
  - Putty
- **cuda-gdb**
  - Along with GPU
- **Profiler**
  - Visual Profiler

cuda-gdb

- 编译程序时，使用 **–g –G**选项； **Linux**下停止 **x-server,**启动命令行
  - 常用的调试命令列表
  - **breakpoint**（**b**）：设置断点，使代码在指定位置暂停执行。其参数可以是方法名，也可以是行号。
  - **run**（**r**）：在调试器内执行程序。
  - **next**（**n**）：单步执行到下一行代码。
  - **continue**（**c**）：继续执行已暂停的程序至下一个断点或程序结尾处。
  - **backtrace**（**bt**）：显示当前方法调用的栈中的内容。
  - **thread**：列出当前的主机线程。
  - **cuda thread**：列出当前活跃的**GPU**线程（若有的话）。
  - **cuda kernel**：列出当前活跃的**GPU Kernel**，并允许将“焦点”转移到指定的**GPU**线程。