

Device Management and Error Handling

Device Management

- Application can query and select GPUs

```
cudaGetDeviceCount(int *count)
cudaSetDevice(int device)
cudaGetDevice(int *device)
cudaGetDeviceProperties(cudaDeviceProp *prop, int
device)
```

- Multiple threads can share a device
- A single thread can manage multiple devices
 - `cudaSetDevice(i)` to select current device
 - `cudaMemcpy(...)` for peer-to-peer copies

Error handling

- All CUDA API calls return an error code

`cudaError_t`

- Values different from `cudaSuccess` indicate errors
- Every function call should be checked:
- Useful macro is

```
#define CUDA_CALL(x) { const cudaError_t a = (x); If (a !=  
cudaSuccess) { printf("\n CUDA Error: %s (err_num=%d) \n",  
cudaGetErrorString(a),a); cudaDeviceReset(); assert(0);}}
```

```
CUDA_CALL(cudaMalloc(...));
```

Error handling

- Error handling with kernel is different, as kernel runs on device.
- To check for errors in the kernel **cudaPeekAtLastError()** is used.
- After the kernel call we can write:

```
If (cudaPeekAtLastError() != cudaSuccess)
{
    printf("\n%s%s%s", start, cudaGetErrorString(cudaGetLastError()),
    end) ;
    cudaDeviceReset() ;
}
```

Concurrency and asynchronous operations

Coordinating Host & Device

- **Kernel launches are asynchronous**
 - **Control returns to the CPU immediately**
- **CPU needs to synchronize before consuming the results**

`cudaMemcpy ()`

Blocks the CPU until the copy is complete
Copy begins when all preceding CUDA calls have completed

`cudaMemcpyAsync ()`

Asynchronous, does not block the CPU

`cudaDeviceSynchronize ()`

Blocks the CPU until all preceding CUDA calls have completed

Pinned (page-locked) memory

- **Pinned memory enables:**
 - **Direct Memory Access GPU->CPU without the involvement of the CPU**
 - **faster PCIe copies**
 - **memcpys asynchronous with CPU**
 - **memcpys asynchronous with GPU**
- **Usage**
 - **`cudaMallocHost` / `cudaFreeHost`**
 - **instead of malloc / free**

Streams and Async API

- **Default API:**
 - Kernel launches are asynchronous with CPU
 - Memcopies (D2H, H2D) block CPU thread
 - CUDA calls are serialized by the driver
- **Streams and async functions provide:**
 - Memcopies (D2H, H2D) asynchronous with CPU
 - Ability to concurrently execute a kernel and a memcopy
- **Stream = sequence of operations that execute in issue-order on GPU**
 - Operations from different streams may be interleaved
 - A kernel and memcopy from different streams can be overlapped

Overlap kernel and memory copy

- Requirements:
 - D2H or H2D memcopy from pinned memory
 - Kernel and memcopy in different, non-0 streams
- Code:

```
cudaStream_t    stream1, stream2;  
cudaStreamCreate(&stream1);  
cudaStreamCreate(&stream2);
```

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

} overlapp

Call Sequencing for Optimal Overlap

- **CUDA calls are dispatched to the hw in the sequence they were issued**
- **Fermi can concurrently execute:**
 - Up to 16 kernels
 - Up to 2 memcopies, as long as they are in different directions (D2H and H2D)
- **A call is dispatched if both are true:**
 - Resources are available
 - Preceding calls in the same stream have completed
- **Scheduling:**
 - Kernels are executed in the order in which they were issued

Example

Serial



time

Concurrent – overlap kernel and D2H copy

streams



1.33x
performance
improvement

time

Amount of concurrency

Serial (1x)



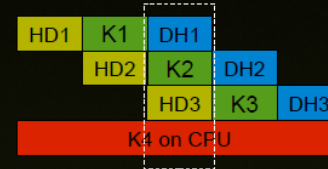
2-way concurrency (up to 2x)



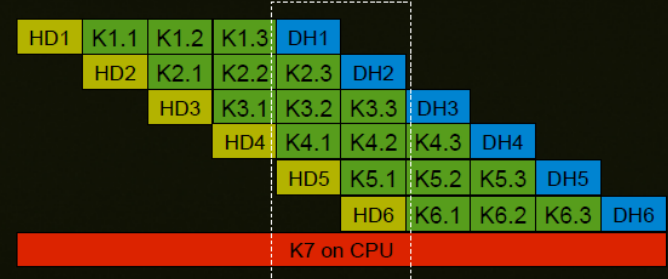
3-way concurrency (up to 3x)



4-way concurrency (3x+)



4+ way concurrency



Default stream

- **Stream used when no stream is specified**
- **Completely synchronous w.r.t. host and device**
 - As if `cudaDeviceSynchronize()` inserted before and after every CUDA operation
- **Exceptions – asynchronous w.r.t. host**
 - Kernel launches in the default stream
 - `cudaMemcpyAsync`
 - `cudaMemsetAsync`
 - `cudaMemcpy` within the same device
 - H2D `cudaMemcpy` of 64kB or less

Requirement for concurrency

- **CUDA operations must be in different, non-0, streams**
- **cudaMemcpyAsync with host from 'pinned' memory**
 - Page-locked memory
 - Allocated using cudaMallocHost() or cudaHostAlloc()
- **Sufficient resources must be available**
 - cudaMemcpyAsyncs in different directions
 - Device resources (SMEM, registers, blocks, etc.)

Synchronous

```
cudaMalloc ( &dev1, size ) ;  
double* host1 = (double*) malloc ( &host1, size ) ;  
...
```

```
cudaMemcpy ( dev1, host1, size, H2D ) ;  
kernel2 <<< grid, block, 0 >>> ( ..., dev2, ... ) ;  
kernel3 <<< grid, block, 0 >>> ( ..., dev3, ... ) ;  
cudaMemcpy ( host4, dev4, size, D2H ) ;  
...
```



**completely
synchronous**

- **All CUDA operations in the default stream are synchronous**

Asynchronous, No Streams

```
cudaMalloc ( &dev1, size ) ;  
double* host1 = (double*) malloc ( &host1, size ) ;  
...
```

```
cudaMemcpy ( dev1, host1, size, H2D ) ;  
kernel2 <<< grid, block >>> ( ..., dev2, ... ) ;  
some_CPU_method () ;  
kernel3 <<< grid, block >>> ( ..., dev3, ... ) ;  
cudaMemcpy ( host4, dev4, size, D2H ) ;  
...
```



**potentially
overlapped**

- **GPU kernels are asynchronous with host by default**

```
cudaStream_t stream1, stream2, stream3, stream4 ;  
cudaStreamCreate ( &stream1 ) ;  
...  
cudaMalloc ( &dev1, size ) ;  
cudaMallocHost ( &host1, size ) ;  
...  
cudaMemcpyAsync ( dev1, host1, size, H2D, stream1 ) ;  
kernel2 <<< grid, block, 0, stream2 >>> ( ..., dev2, ... ) ;  
kernel3 <<< grid, block, 0, stream3 >>> ( ..., dev3, ... ) ;  
cudaMemcpyAsync ( host4, dev4, size, D2H, stream4 ) ;  
some_CPU_method () ;  
...
```

// pinned memory required on host



**potentially
overlapped**

- **Fully asynchronous / concurrent**
- **Data used by concurrent operations should be independent**

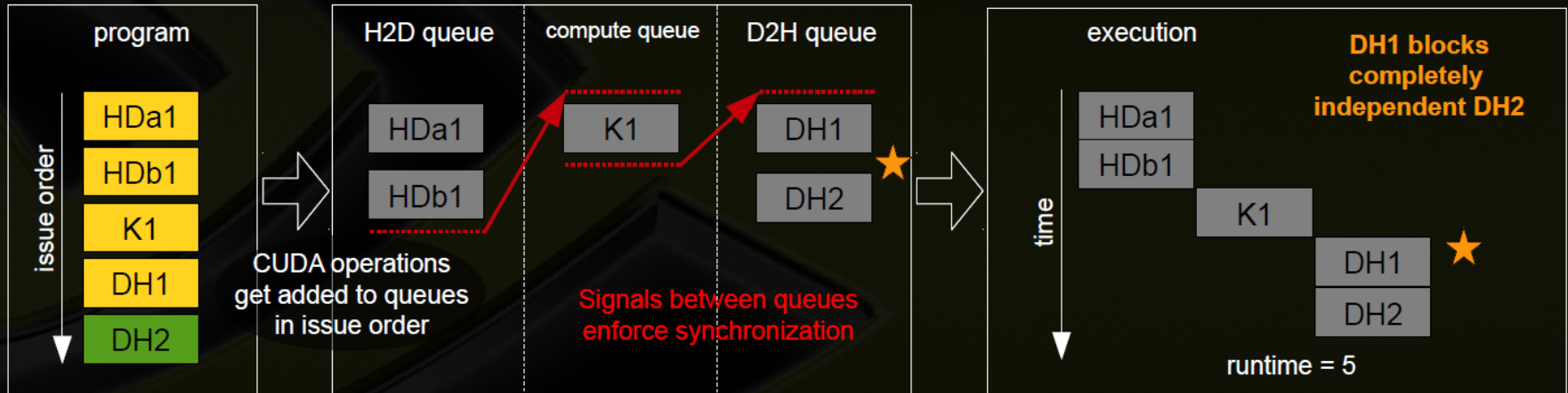
Stream scheduling

In Kepler each CUDA streams has its own hardware queue

- **Fermi hardware has 3 queues**
 - 1 Compute Engine queue
 - 2 Copy Engine queues – one for H2D and one for D2H
- **CUDA operations are dispatched to HW in the sequence they were issued**
 - Placed in the relevant queue
 - Stream dependencies between engine queues are maintained, but lost within an engine queue
- **A CUDA operation is dispatched from the engine queue if:**
 - Preceding calls in the same stream have completed,
 - Preceding calls in the same queue have been dispatched, and
 - Resources are available
- **CUDA kernels may be executed concurrently if they are in different streams**
 - Threadblocks for a given kernel are scheduled if all threadblocks for preceding kernels have been scheduled and there still are SM resources available
- **Note a blocked operation blocks all other operations in the queue, even in other streams**

Blocked Queue

- Two streams, stream 1 is issued first
 - Stream 1 : HDa1, HDb1, K1, DH1 (issued first)
 - Stream 2 : DH2 (completely independent of stream 1)



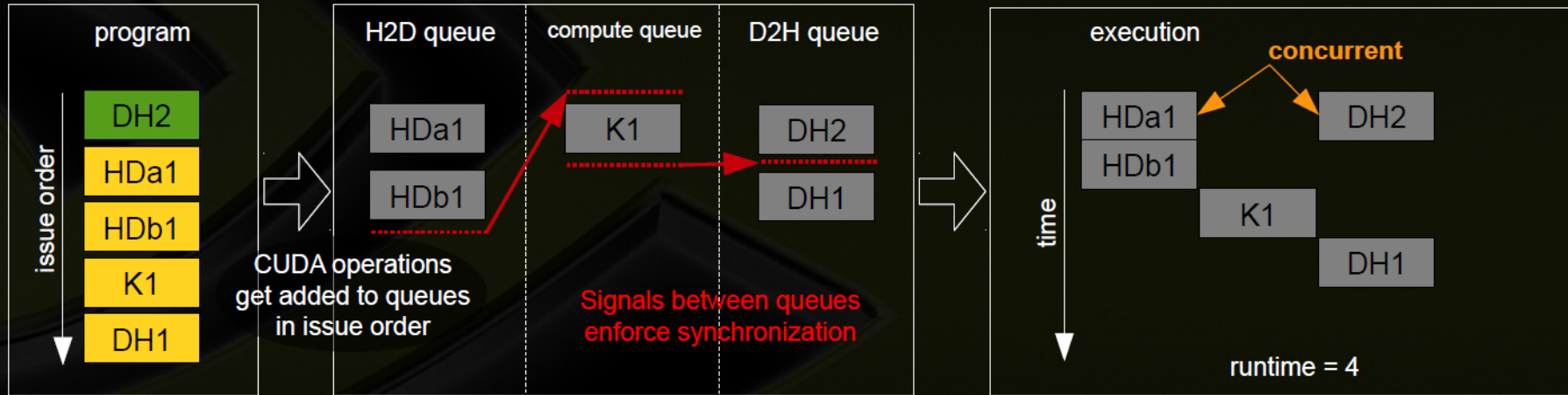
within queues, stream dependencies are lost

Blocked Queue

- Two streams, stream 2 is issued first

- Stream 1 : HDa1, HDb1, K1, DH1
- Stream 2 : DH2 (issued first)

issue order matters!



within queues, stream dependencies are lost

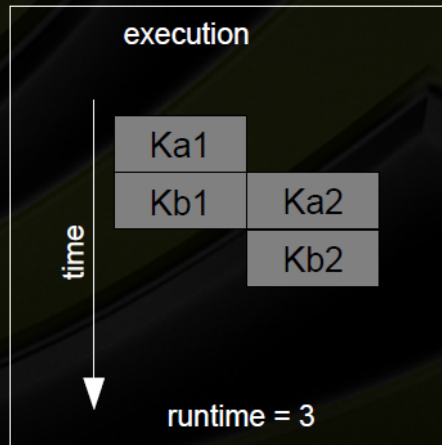
Blocked Kernel

- Two streams – just issuing CUDA kernels

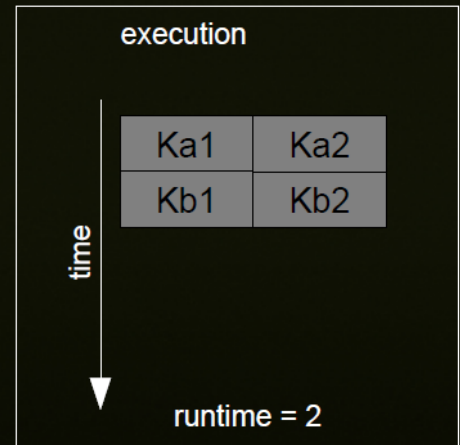
- Stream 1 : Ka1, Kb1
- Stream 2 : Ka2, Kb2
- Kernels are similar size, fill $\frac{1}{2}$ of the SM resources

issue order matters!

- Issue depth first



- Issue breadth first



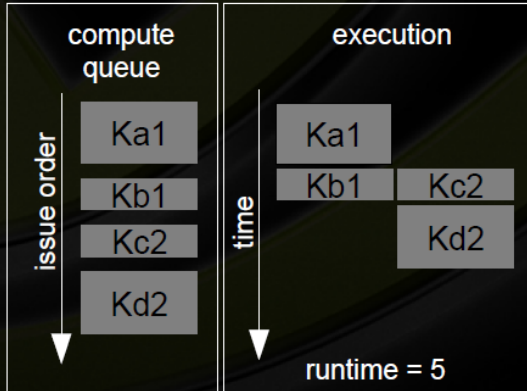
Optimal concurrency depends on kernel execution time

- Two streams – just issuing CUDA kernels – but kernels are different 'sizes'

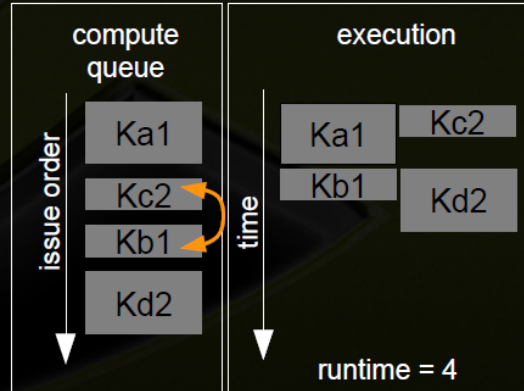
- Stream 1 : Ka1 {2}, Kb1 {1}
- Stream 2 : Kc2 {1}, Kd2 {2}
- Kernels fill $\frac{1}{2}$ of the SM resources

issue order matters!
execution time matters!

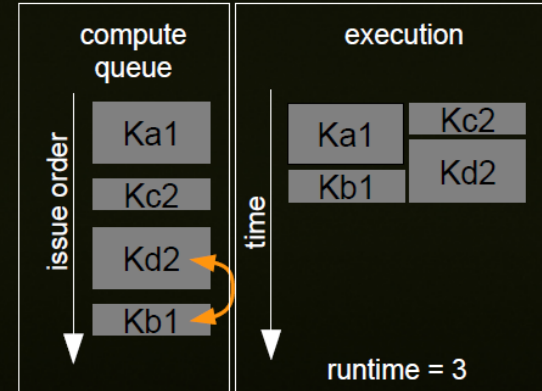
- Depth first



- Breadth first



- Custom



Kepler (compute capability 3.5)

Hyper-Q on Kepler

Hyper-Q enables multiple CPU threads or processes to launch work on a single GPU simultaneously.

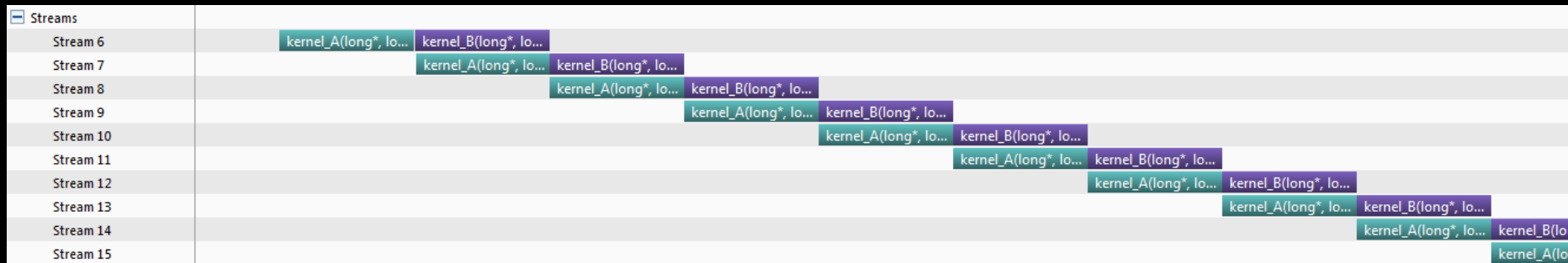
Applications that previously encountered false serialization across tasks, thereby limiting achieved GPU utilization, can see up to dramatic performance increase without changing any existing code.

Hyper-Q example

```
for (int i = 0 ; i < nstreams ; ++i)
{
    kernel_A<<<1,1,0,streams[i]>>>(.....);
    kernel_B<<<1,1,0,streams[i]>>>(.....);
}
```

Running without Hyper-Q

On a device without Hyper-Q, the single work pipeline in hardware means that we can only see concurrency between pairs of kernel_B from stream N and kernel_A from stream N+1



Running without Hyper-Q

On a device with Hyper-Q, the false dependencies are eliminated and all the kernel_As can execute concurrently (as can all the kernel_Bs).

Streams			
Stream 6		kernel_A(long*, long)	kernel_B(long*, long)
Stream 7		kernel_A(long*, long)	kernel_B(long*, long)
Stream 8		kernel_A(long*, long)	kernel_B(long*, long)
Stream 9		kernel_A(long*, long)	kernel_B(long*, long)
Stream 10		kernel_A(long*, long)	kernel_B(long*, long)
Stream 11		kernel_A(long*, long)	kernel_B(long*, long)
Stream 12		kernel_A(long*, long)	kernel_B(long*, long)
Stream 13		kernel_A(long*, long)	kernel_B(long*, long)
Stream 14		kernel_A(long*, long)	kernel_B(long*, long)
Stream 15		kernel_A(long*, long)	kernel_B(long*, long)
Stream 16		kernel_A(long*, long)	kernel_B(long*, long)
Stream 17		kernel_A(long*, long)	kernel_B(long*, long)
Stream 18		kernel_A(long*, long)	kernel_B(long*, long)
Stream 19		kernel_A(long*, long)	kernel_B(long*, long)
Stream 20		kernel_A(long*, long)	kernel_B(long*, long)

Dynamic Parallelism

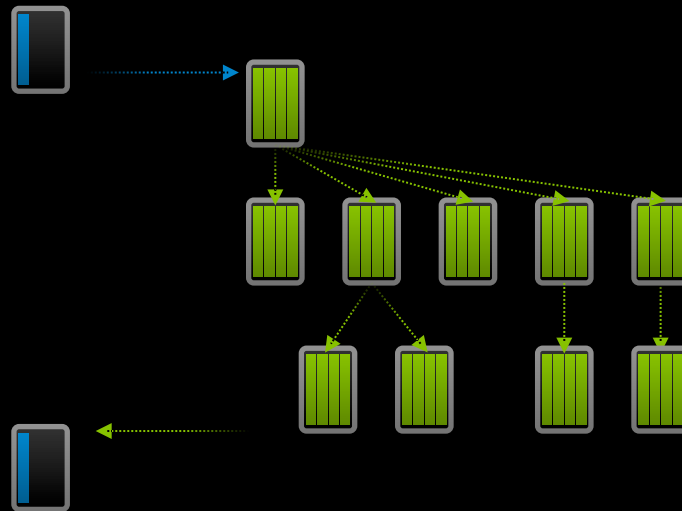
CPU

Fermi GPU



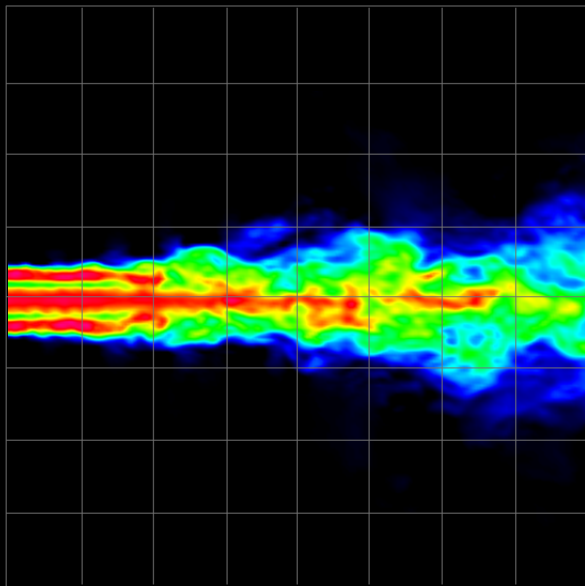
CPU

Kepler GPU



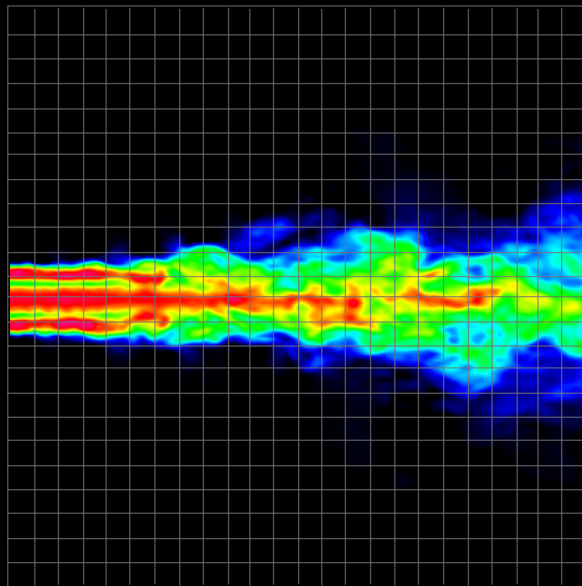
Dynamic Work Generation

Coarse grid



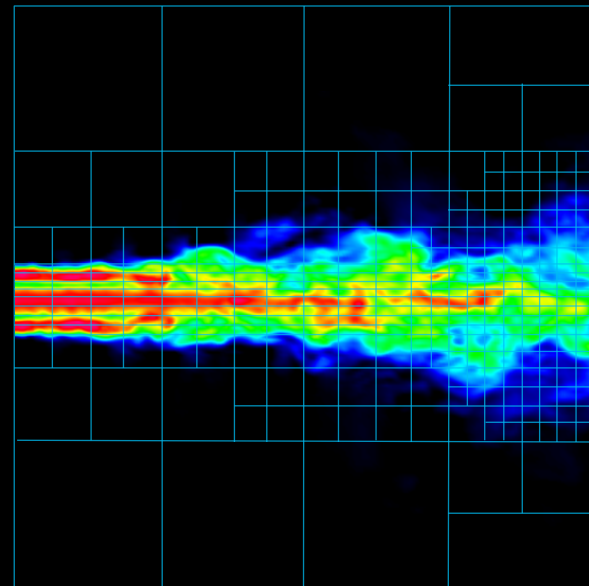
Higher Performance
Lower Accuracy

Fine grid



Lower Performance
Higher Accuracy

Dynamic grid

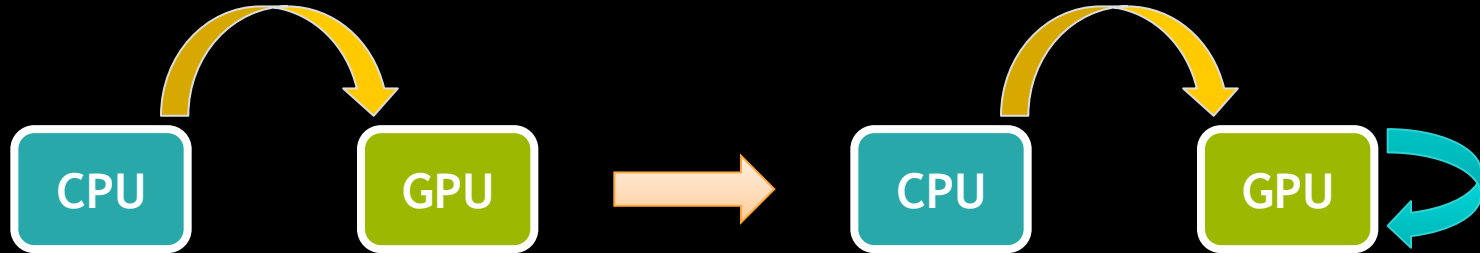


Target performance where
accuracy is required

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



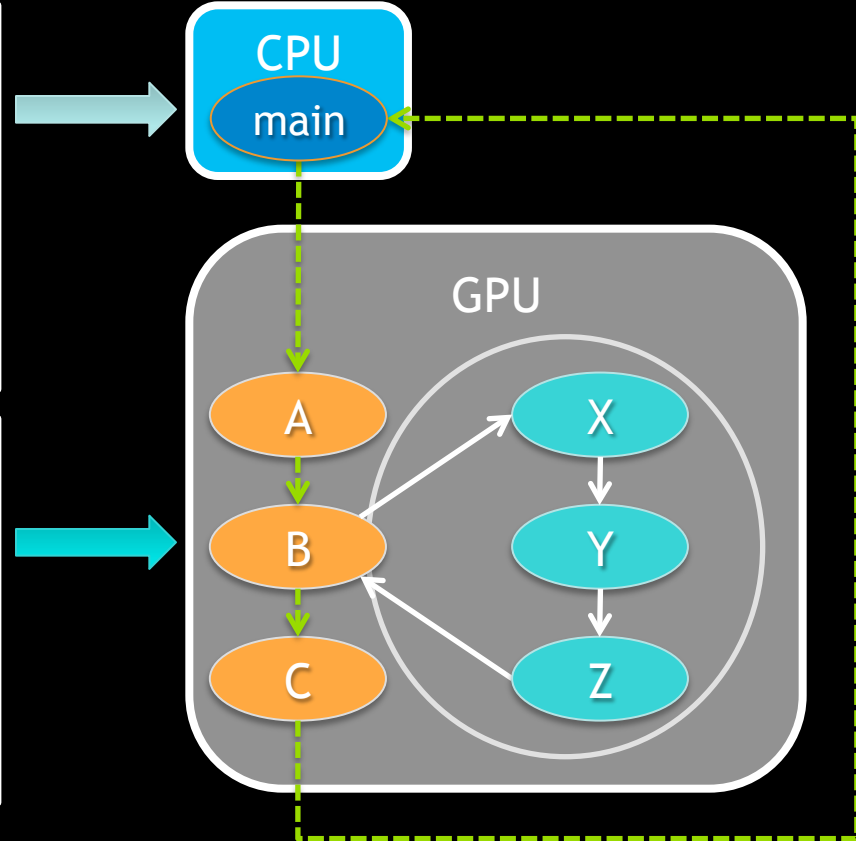
Fermi: Only CPU can generate GPU work

Kepler: GPU can generate work for itself

Familiar Syntax and Programming Model

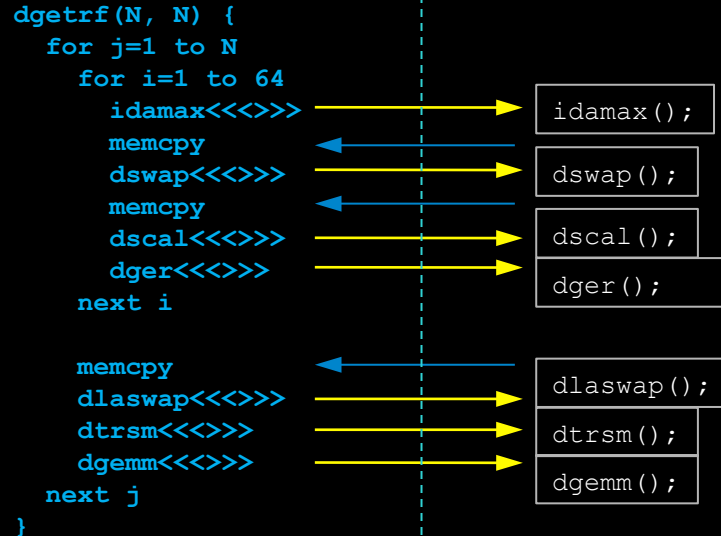
```
int main() {  
    float *data;  
    setup(data);  
  
    A <<< ... >>> (data);  
    B <<< ... >>> (data);  
    C <<< ... >>> (data);  
  
    cudaDeviceSynchronize();  
    return 0;  
}
```

```
__global__ void B(float *data)  
{  
    do_stuff(data);  
  
    X <<< ... >>> (data);  
    Y <<< ... >>> (data);  
    Z <<< ... >>> (data);  
    cudaDeviceSynchronize();  
  
    do_more_stuff(data);  
}
```



Simpler Code: LU Example

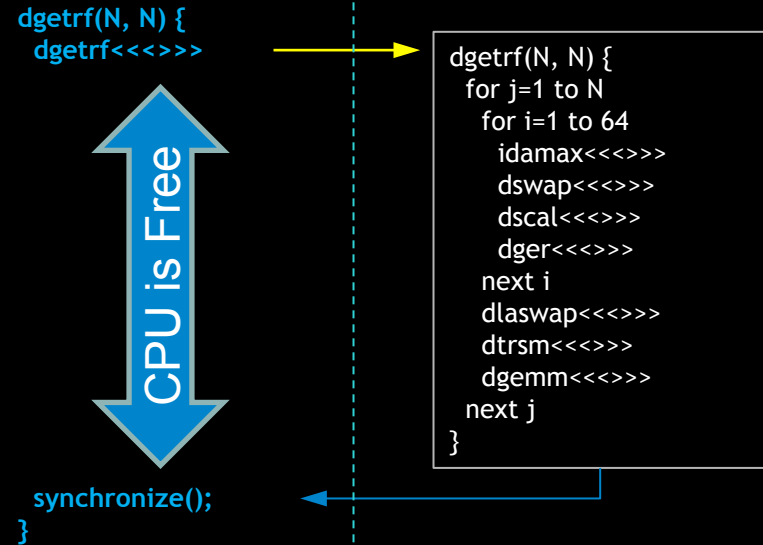
LU decomposition (Fermi)



CPU Code

GPU Code

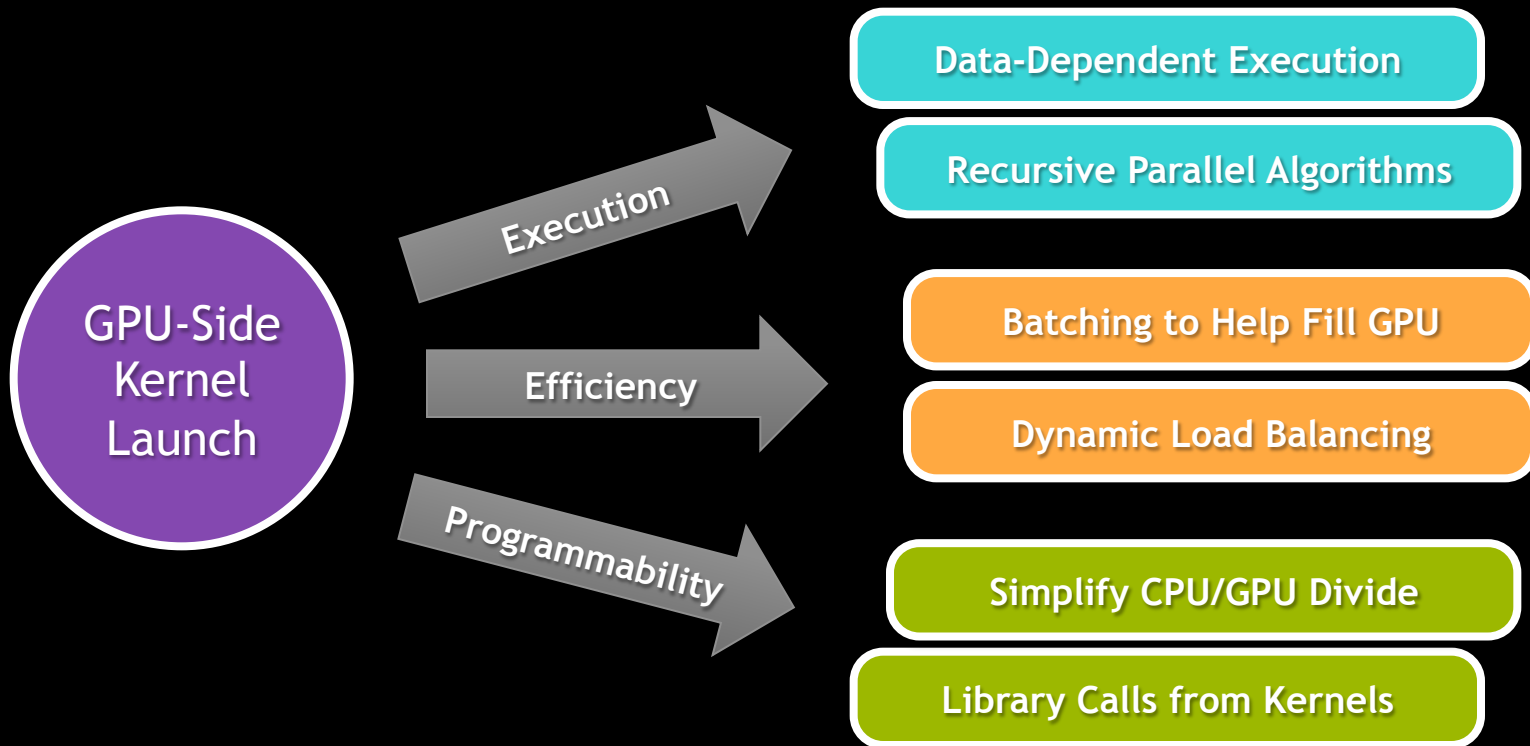
LU decomposition (Kepler)



CPU Code

GPU Code

CUDA Dynamic Parallelism



Profiling

Profiling

- NVIDIA Visual Profiler – nvvp
- Command line profiler – nvprof

\$> nvprof [options] [CUDA-application] [application-arguments]

Output redirected to *stderr*

- 3 profiling modes:

Summary mode

nvprof outputs a single result line for each kernel function and each type of CUDA memory copy/set performed by the application. For each kernel, **nvprof** outputs the total time of all instances of the kernel or type of memory copy as well as the average, minimum, and maximum time.

- 3 profiling modes:

GPU-trace

GPU-trace mode provides a timeline of all activities taking place on the GPU in chronological order. Each kernel execution and memory copy/set instance is shown in the output. For each kernel or memory copy detailed information such as kernel parameters, shared memory usage and memory transfer throughput are shown.

- 3 profiling modes:

API-trace

API-trace mode shows the timeline of all CUDA runtime and driver API calls invoked on the host in chronological order.