

**GPU** TECHNOLOGY  
CONFERENCE

# Introduction to Dynamic Parallelism

**Stephen Jones**  
NVIDIA Corporation



# Improving Programmability

Library Calls from Kernels

Simplify CPU/GPU Divide

Batching to Help Fill GPU

Dynamic Load Balancing

Data-Dependent Execution

Recursive Parallel Algorithms

Programmability

Occupancy

Execution

Dynamic  
Parallelism

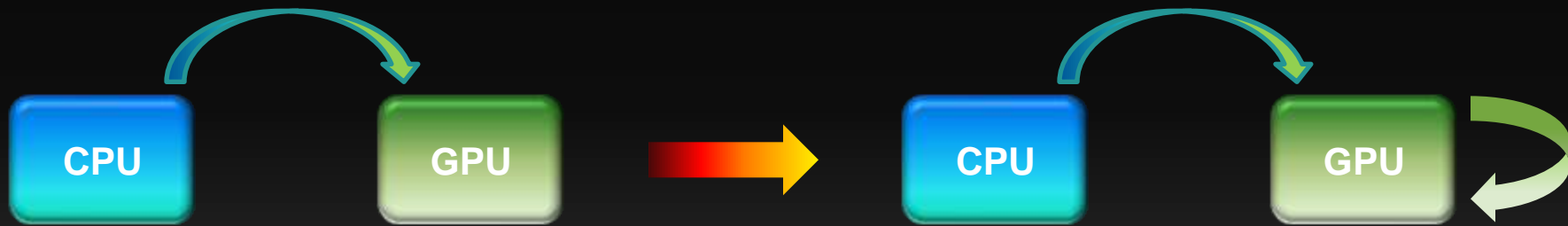
```
graph LR; A[Library Calls from Kernels] -- Programmability --> D((Dynamic Parallelism)); B[Simplify CPU/GPU Divide] -- Occupancy --> D; C[Batching to Help Fill GPU] -- Execution --> D; D1[Dynamic Load Balancing]; E[Data-Dependent Execution]; F[Recursive Parallel Algorithms];
```



# What is Dynamic Parallelism?

## The ability to launch new grids from the GPU

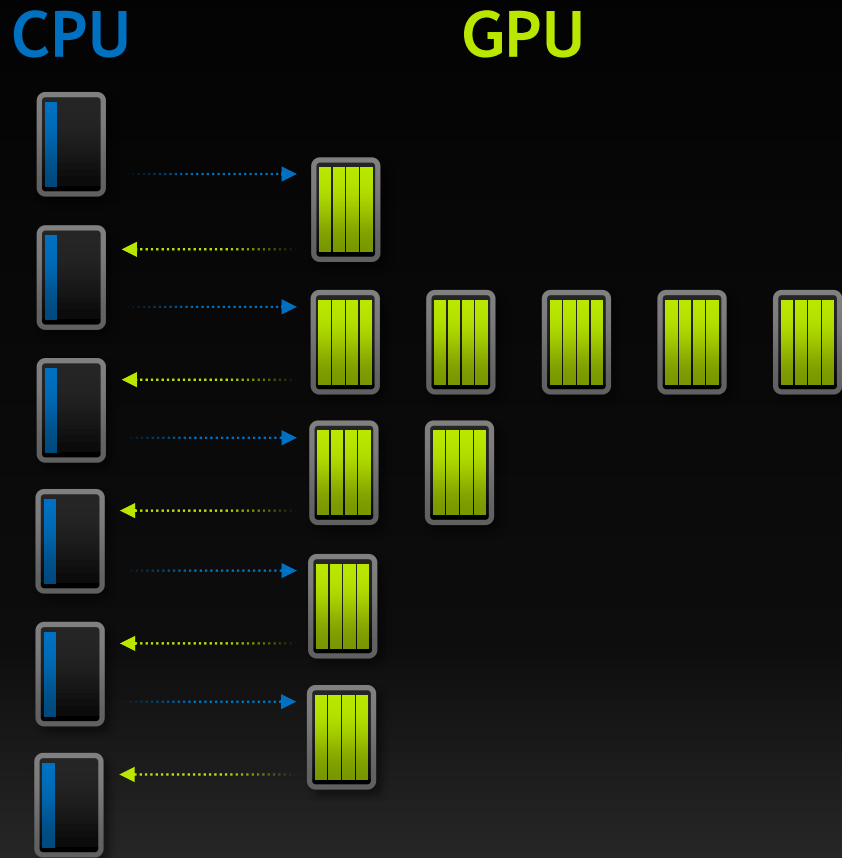
- Dynamically
- Simultaneously
- Independently



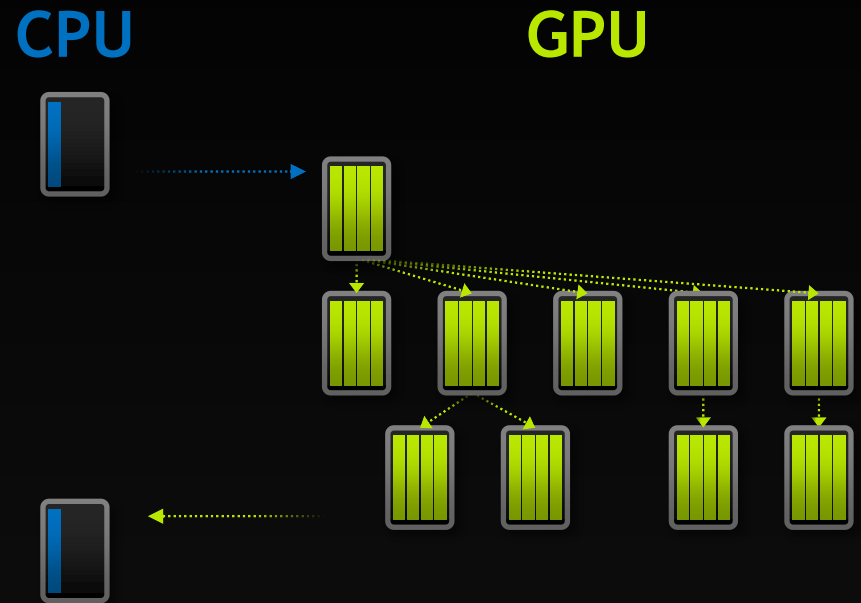
*Fermi: Only CPU can generate GPU work*

*Kepler: GPU can generate work for itself*

# What Does It Mean?



*GPU as Co-Processor*



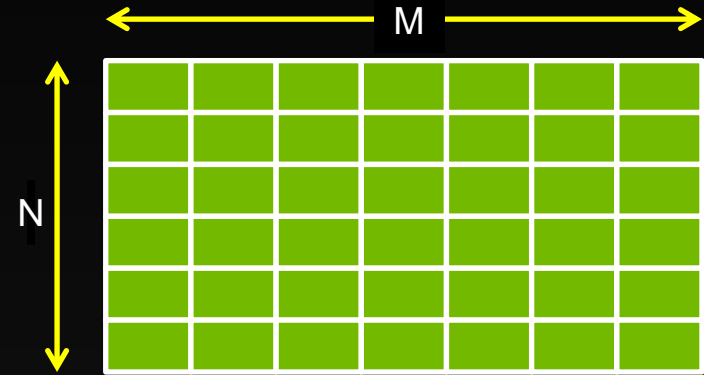
*Autonomous, Dynamic Parallelism*

# The Simplest Parallel Program

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

# The Simplest Parallel Program

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

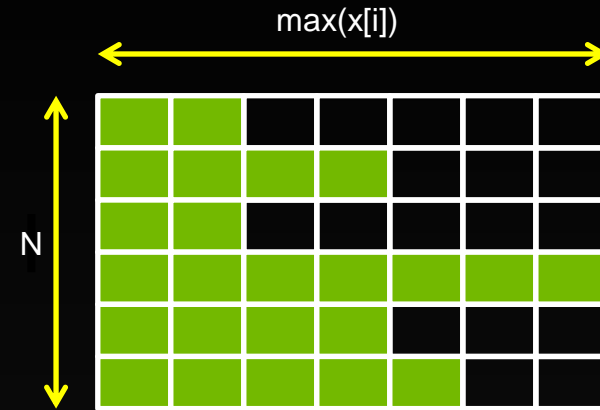


# The Simplest Impossible Parallel Program

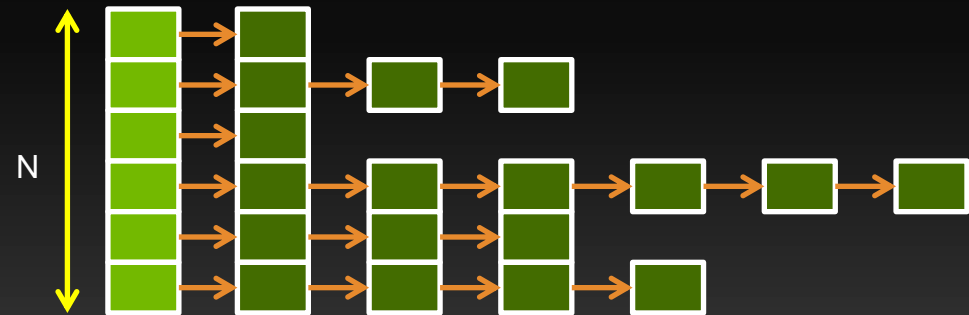
```
for i = 1 to N
  for j = 1 to x[i]
    convolution(i, j)
  next j
next i
```

# The Simplest Impossible Parallel Program

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



*Bad alternative #1: Oversubscription*



*Bad alternative #2: Serialisation*



# The Now-Possible Parallel Program

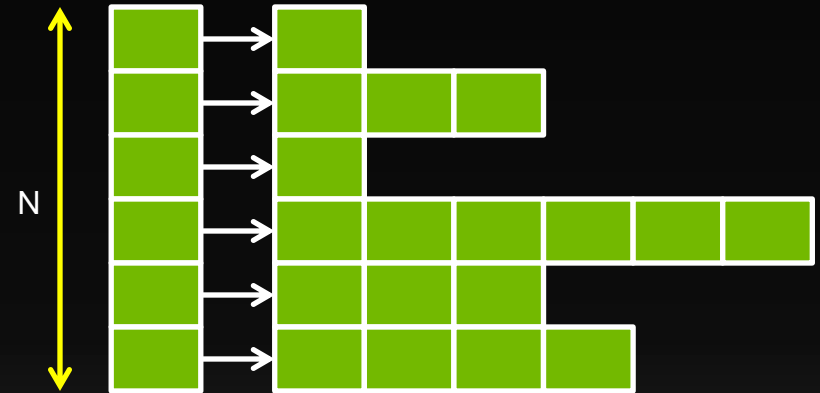
## Serial Program

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

## CUDA Program

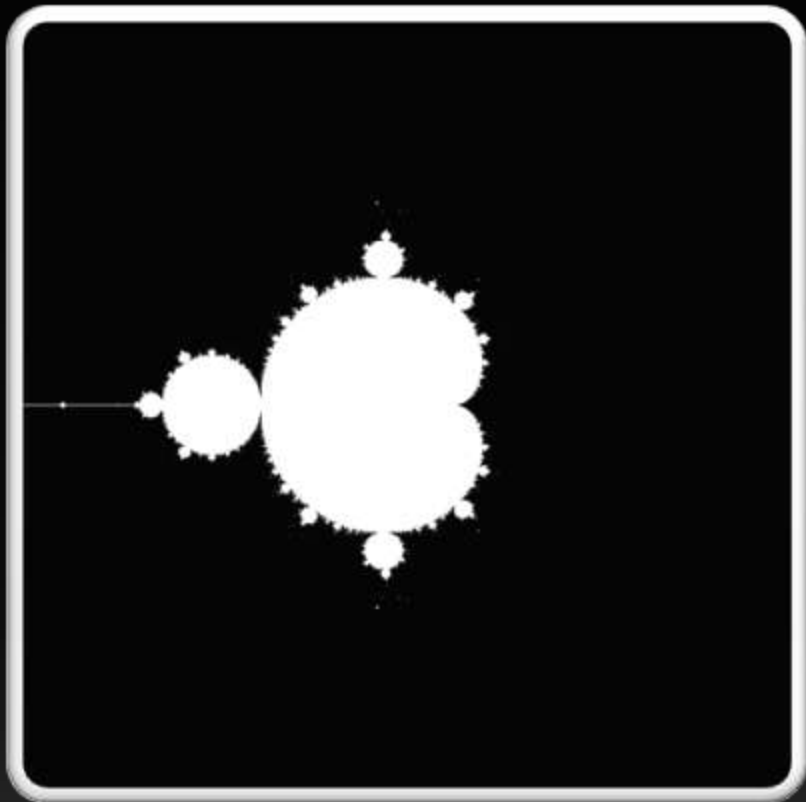
```
__global__ void convolution(int x[])
{
  for j = 1 to x[blockIdx]
    kernel<<< ... >>>(blockIdx, j)
}

convolution<<< N, 1 >>>(x);
```



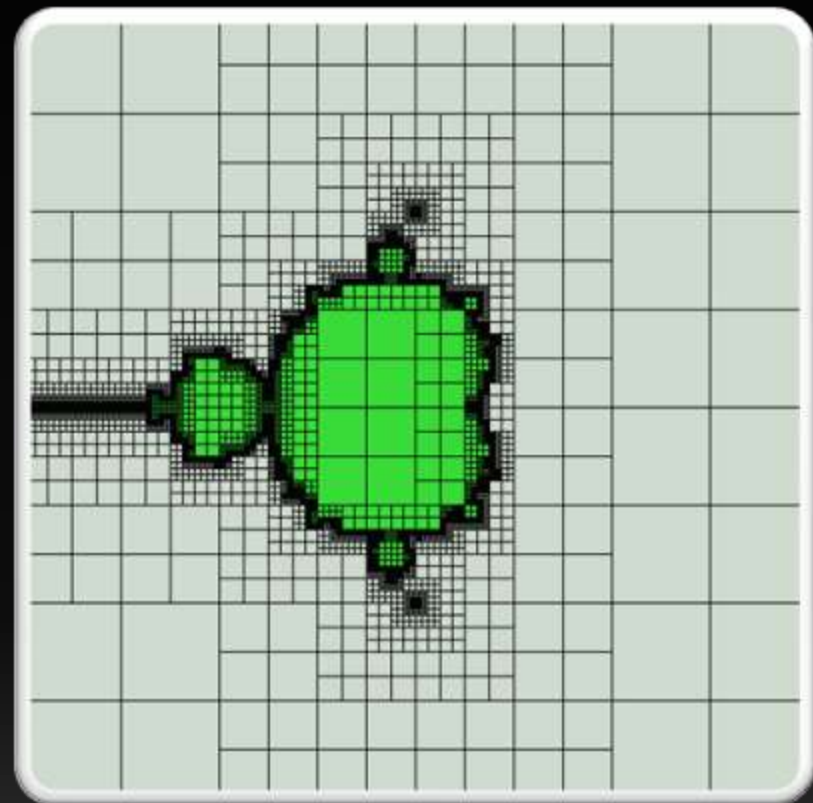
*Now Possible: Dynamic Parallelism*

# Data-Dependent Parallelism



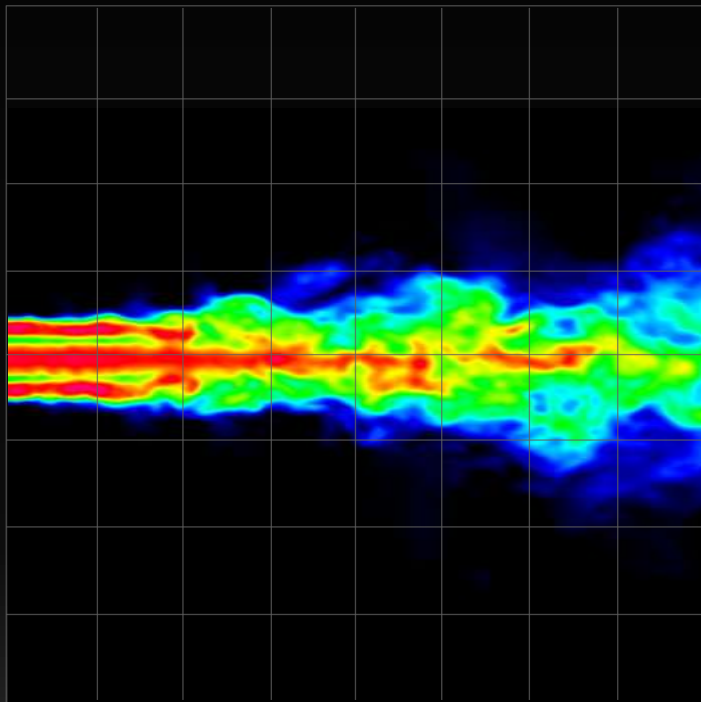
CUDA Today

Computational  
Power allocated to  
regions of interest



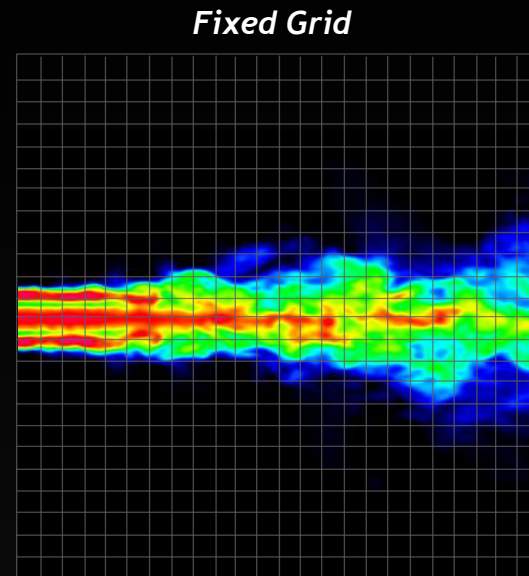
CUDA on Kepler

# Dynamic Work Generation



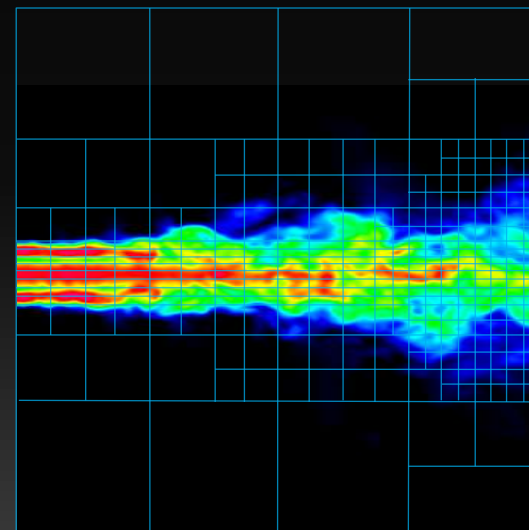
*Initial Grid*

*Statically assign conservative  
worst-case grid*



*Fixed Grid*

*Dynamically assign performance  
where accuracy is required*



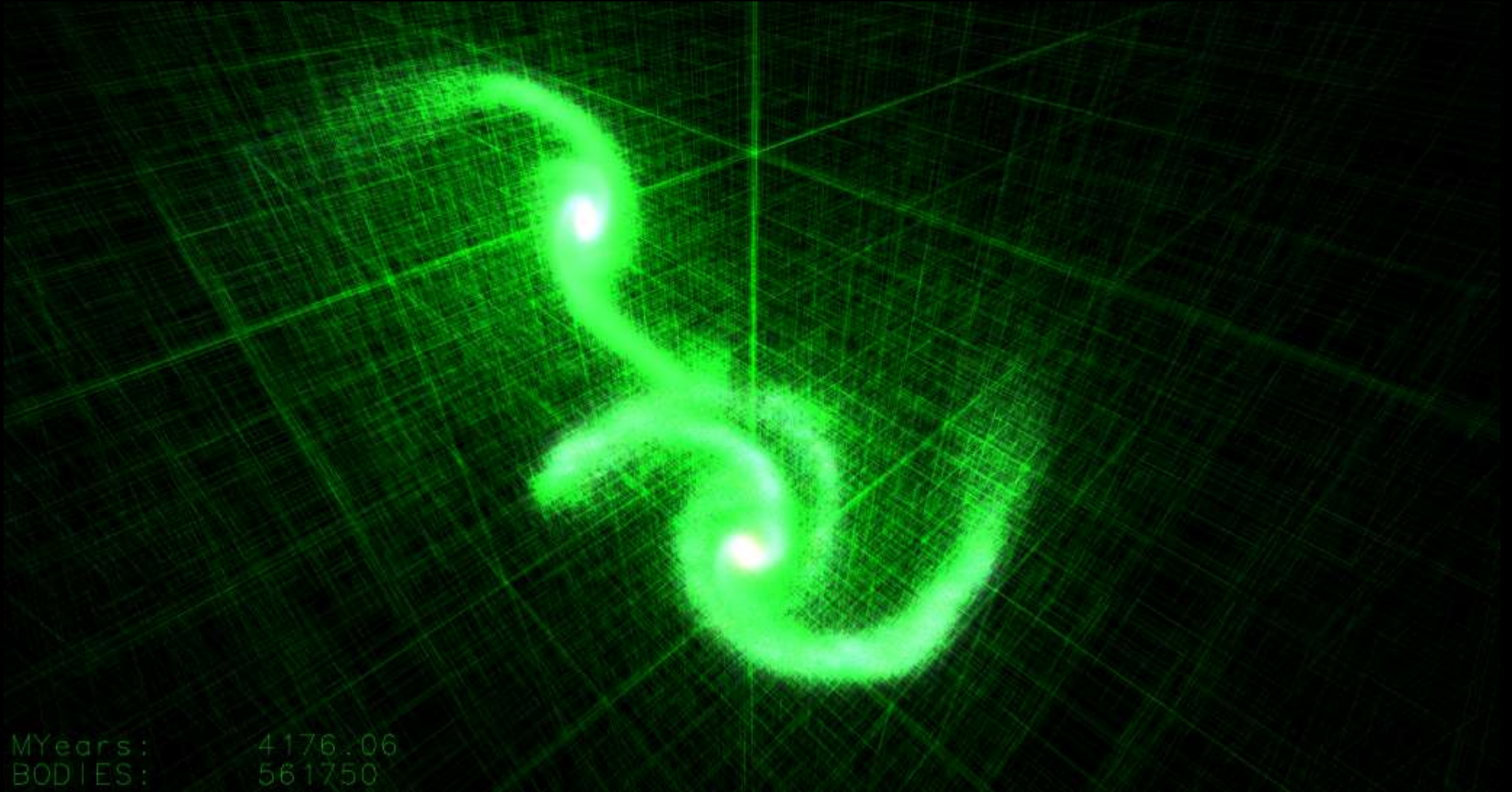
*Dynamic Grid*

# Mapping Compute to the Problem



MYears: 4176.06  
BODIES: 561750

# Mapping Compute to the Problem

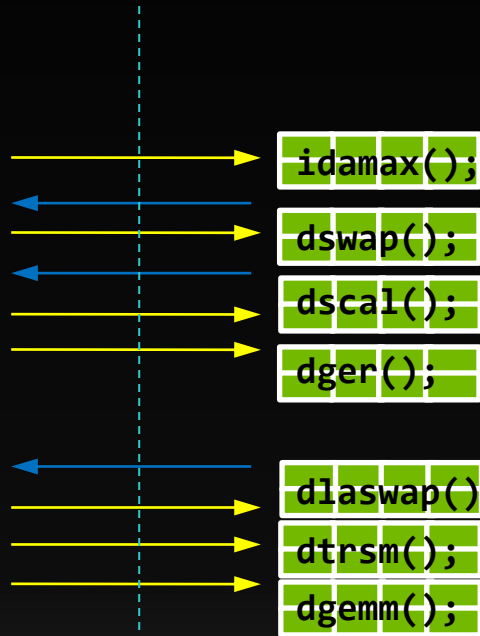


# Library Calls & Nested Parallelism

## LU decomposition (Fermi)

```
dgetrf(N, N) {  
  for j=1 to N  
    for i=1 to 64  
      idamax<<<<>>  
      memcpy  
      dswap<<<<>>  
      memcpy  
      dscal<<<<>>  
      dger<<<<>>  
    next i  
  
    memcpy  
    dlaswap<<<<>>  
    dtrsm<<<<>>  
    dgemm<<<<>>  
  next j  
}
```

CPU Code



## LU decomposition (Kepler)

```
dgetrf(N, N) {  
  dgetrf<<<<>>
```

CPU Code



GPU Code

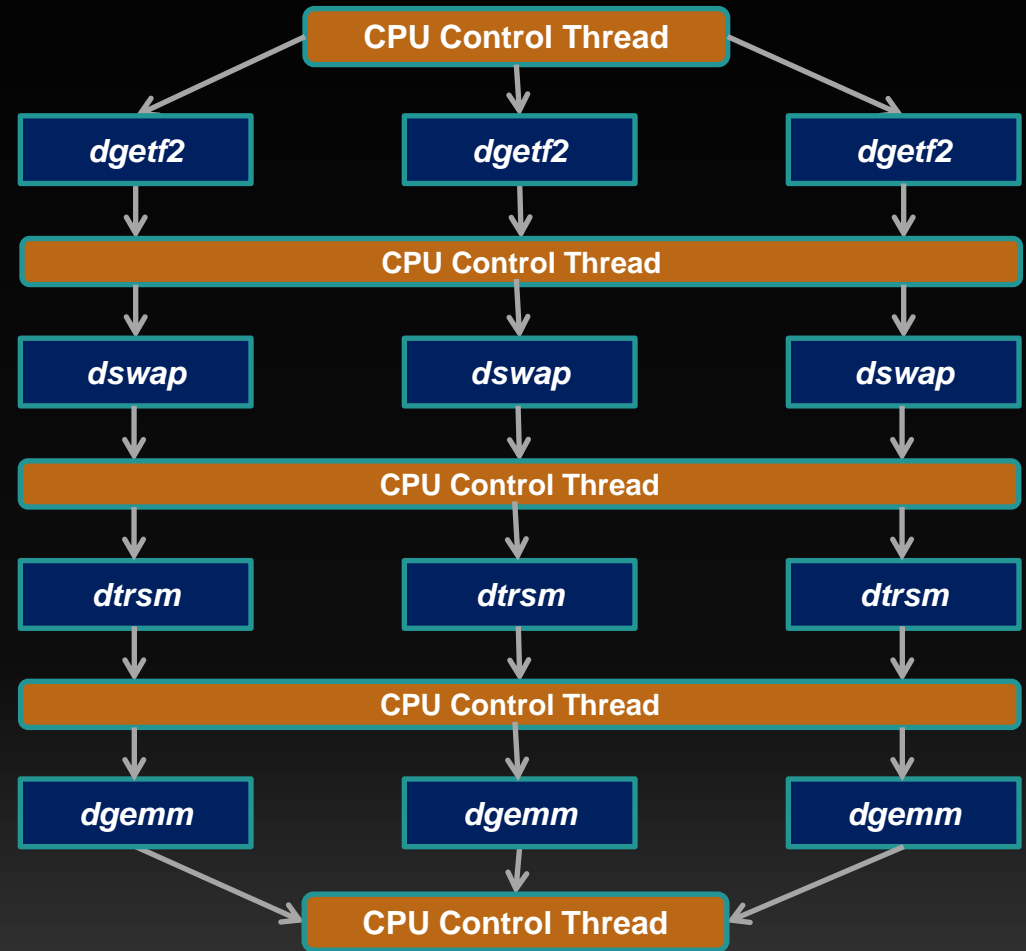
```
synchronize();  
}
```



# 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



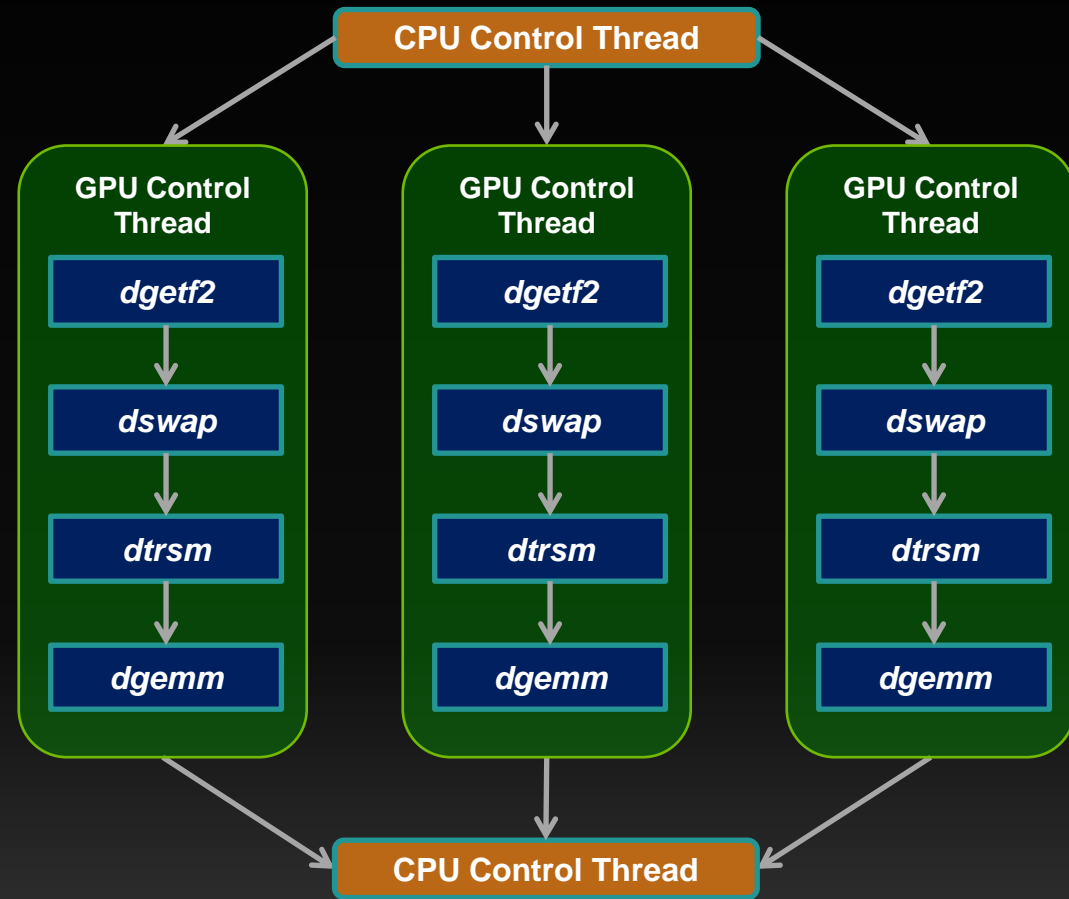
*Multiple LU-Decomposition, Pre-Kepler*



# 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*

# Familiar Syntax

```
void main() {  
    float *data;  
    do_stuff(data);  
  
    A <<< ... >>> (data);  
    B <<< ... >>> (data);  
    C <<< ... >>> (data);  
    cudaDeviceSynchronize();  
  
    do_more_stuff(data);  
}
```



**CUDA from CPU**

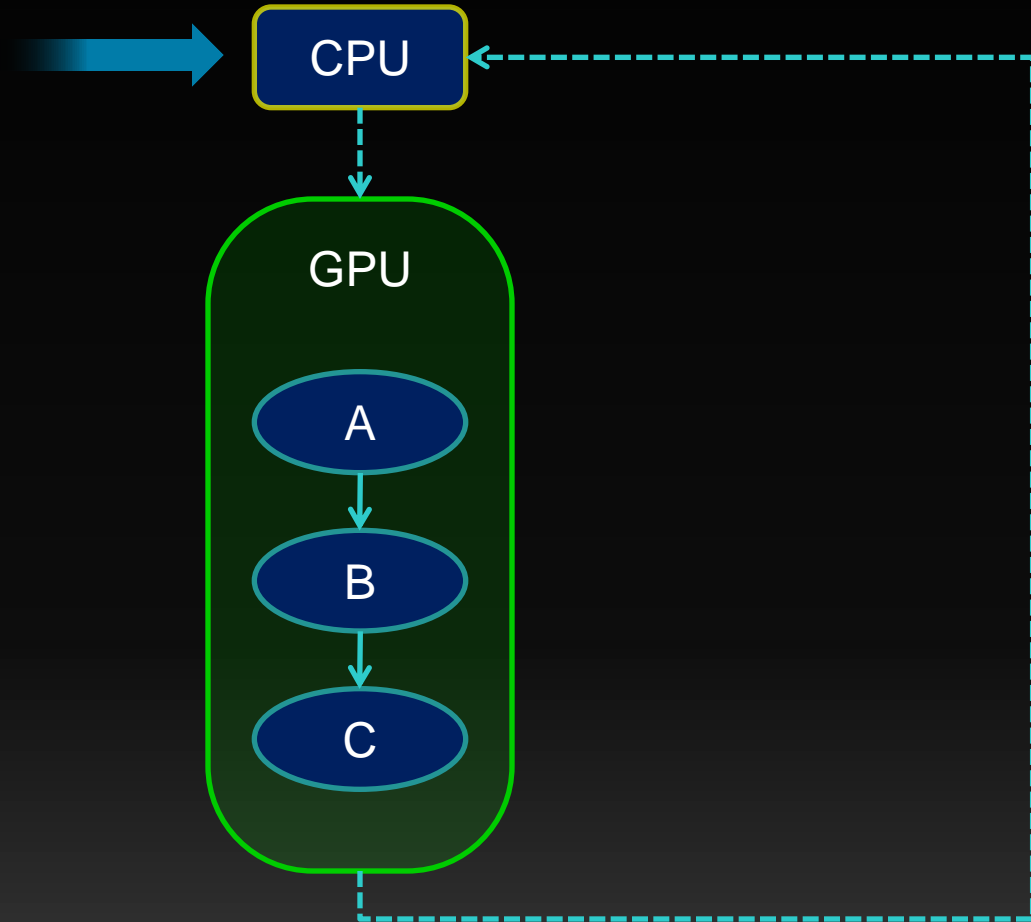
```
__global__ void B(float *data)  
{  
    do_stuff(data);  
  
    X <<< ... >>> (data);  
    Y <<< ... >>> (data);  
    Z <<< ... >>> (data);  
    cudaDeviceSynchronize();  
  
    do_more_stuff(data);  
}
```



**CUDA from GPU**

# Reminder: Dependencies in CUDA

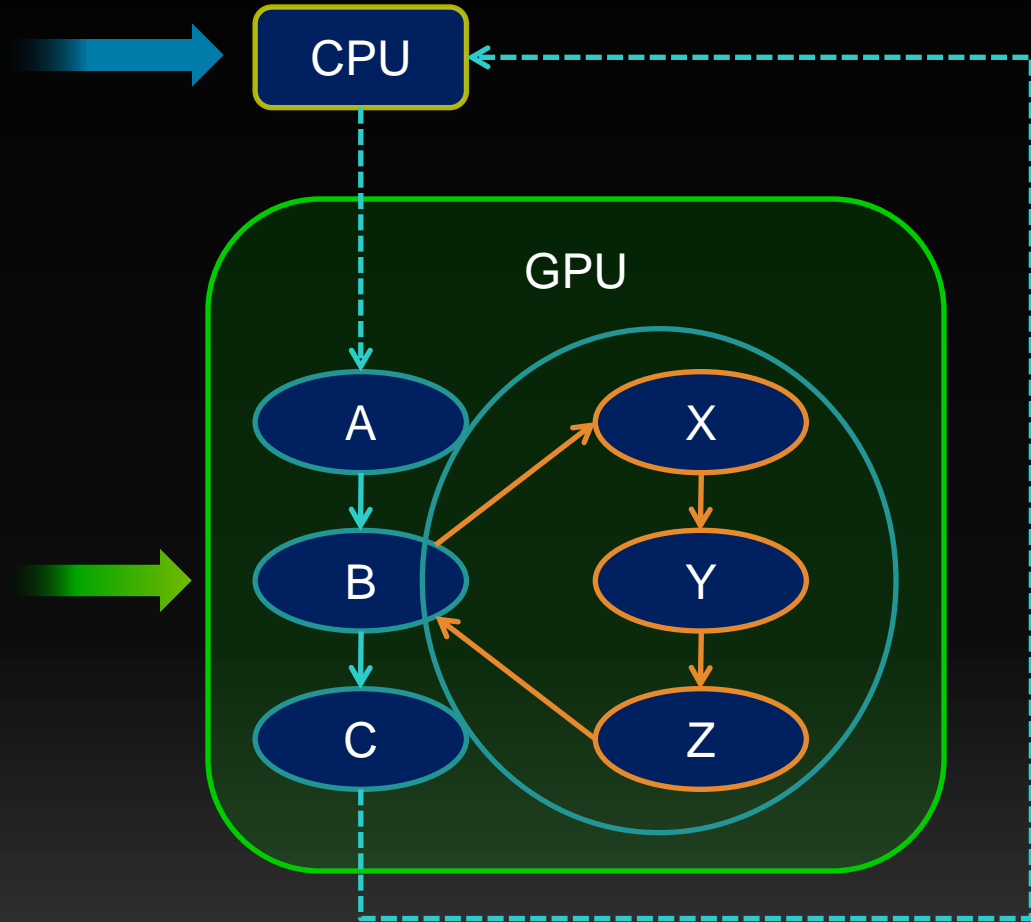
```
void main() {  
    float *data;  
    do_stuff(data);  
  
    A <<< ... >>> (data);  
    B <<< ... >>> (data);  
    C <<< ... >>> (data);  
    cudaDeviceSynchronize();  
  
    do_more_stuff(data);  
}
```



# Nested Dependencies

```
void main() {  
    float *data;  
    do_stuff(data);  
  
    A <<< ... >>> (data);  
    B <<< ... >>> (data);  
    C <<< ... >>> (data);  
    cudaDeviceSynchronize();  
  
    do_more_stuff(data);  
}
```

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



# Programming Model Basics

- CUDA Runtime syntax & semantics

## Code Example

```
__device__ float buf[1024];
__global__ void dynamic(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 Basics

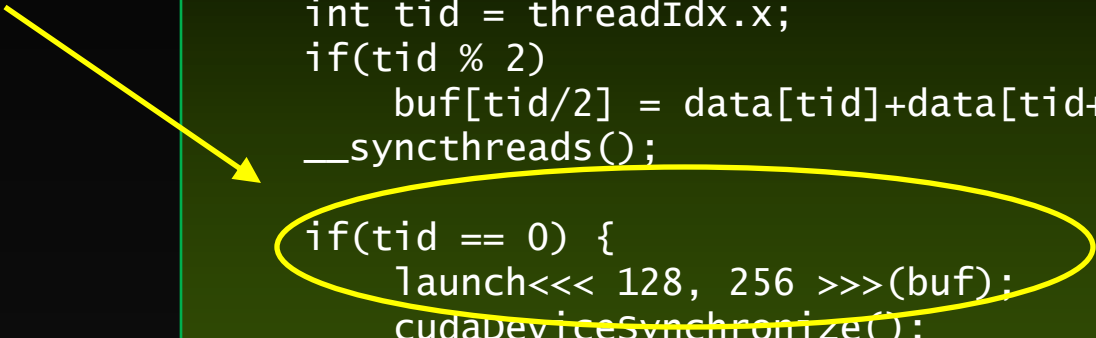
- CUDA Runtime syntax & semantics
- Launch is per-thread

## Code Example

```
__device__ float buf[1024];
__global__ void dynamic(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 Basics

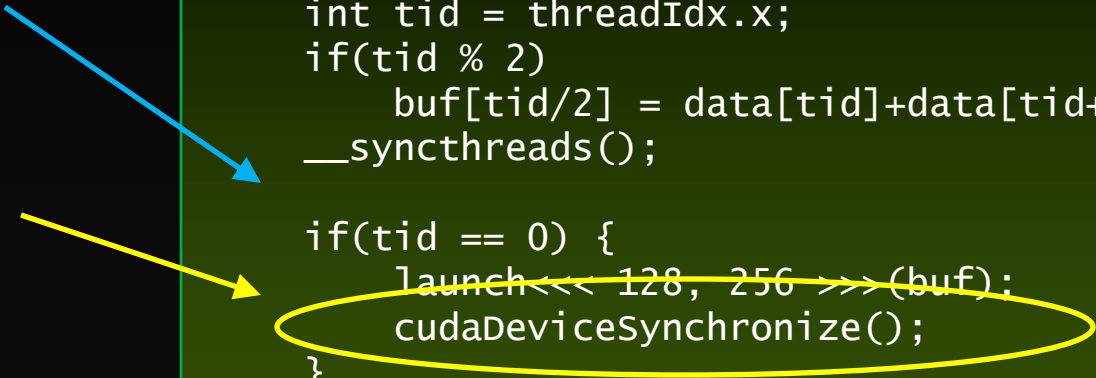
- CUDA Runtime syntax & semantics
- Launch is per-thread
- Sync includes all launches by any thread in the block

## Code Example

```
__device__ float buf[1024];
__global__ void dynamic(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 Basics

- CUDA Runtime syntax & semantics
- Launch is per-thread
- Sync includes all launches by any thread in the block
- *cudaDeviceSynchronize()* does not imply syncthreads

## Code Example

```
__device__ float buf[1024];
__global__ void dynamic(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 Basics

- CUDA Runtime syntax & semantics
- Launch is per-thread
- Sync includes all launches by any thread in the block
- *cudaDeviceSynchronize()* does not imply syncthreads
- Asynchronous launches only

## Code Example

```
__device__ float buf[1024];
__global__ void dynamic(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 Basics

- CUDA Runtime syntax & semantics
- Launch is per-thread
- Sync includes all launches by any thread in the block
- *cudaDeviceSynchronize()* does not imply syncthreads
- Asynchronous launches only  
(note bug in program, here!)

## Code Example

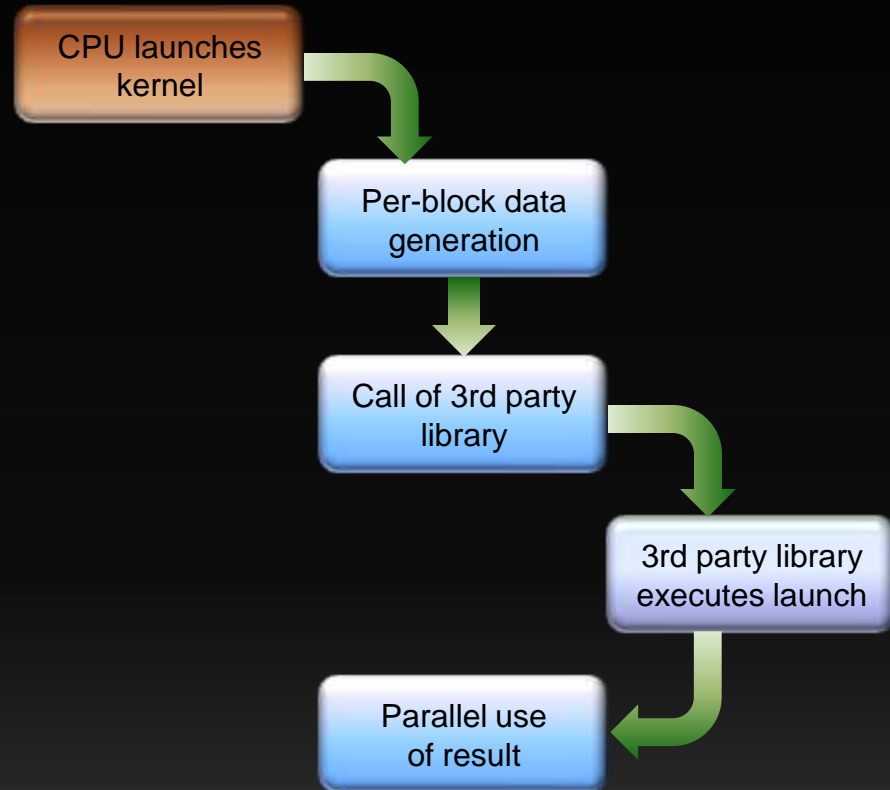
```
__device__ float buf[1024];
__global__ void dynamic(float *data)
{
    int tid = threadIdx.x;
    if(tid % 2)
        buf[tid/2] = data[tid]+data[tid+1];
    __syncthreads();

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

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

# Example 1: Simple Library Calls

```
__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);  
}
```



# Example 1: Simple Library Calls

```
__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 dgemm
    __syncthreads();

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

## Things to notice

Sync before launch to ensure all data is ready

Per-thread execution semantic

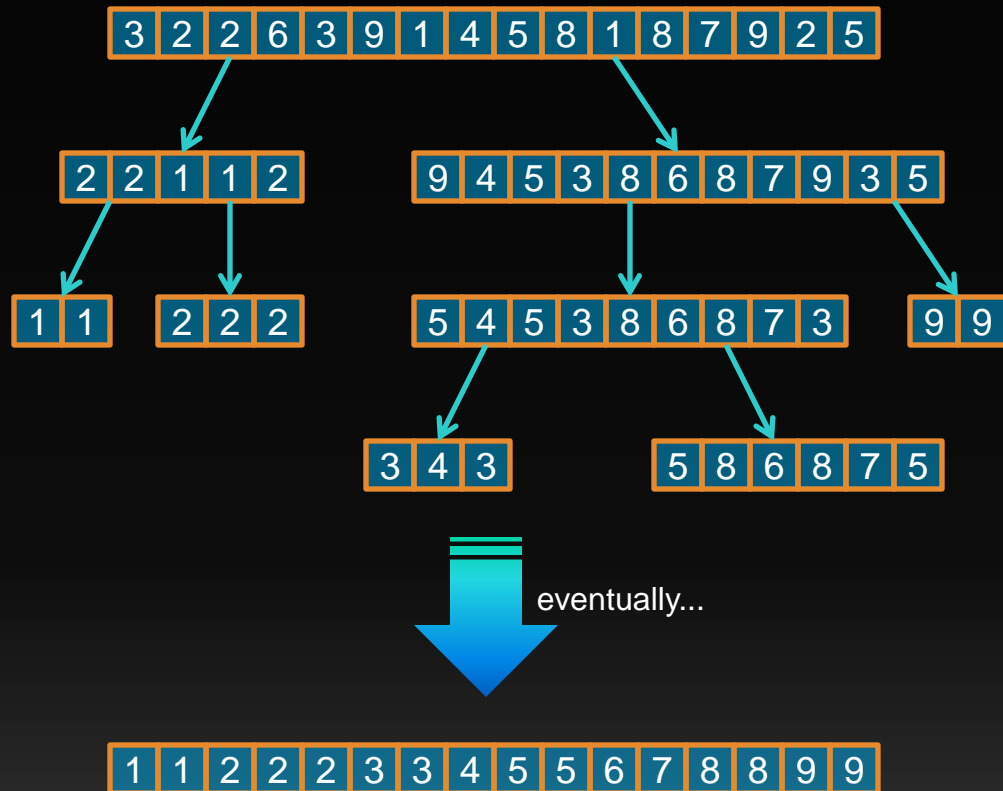
Single call to external library function

(Note launch performed by external library,  
but we synchronize in our own kernel)

*cudaDeviceSynchronize()* by launching thread

*\_\_syncthreads()* before consuming data

# Example 2: Parallel Recursion



## Simple example: Quicksort

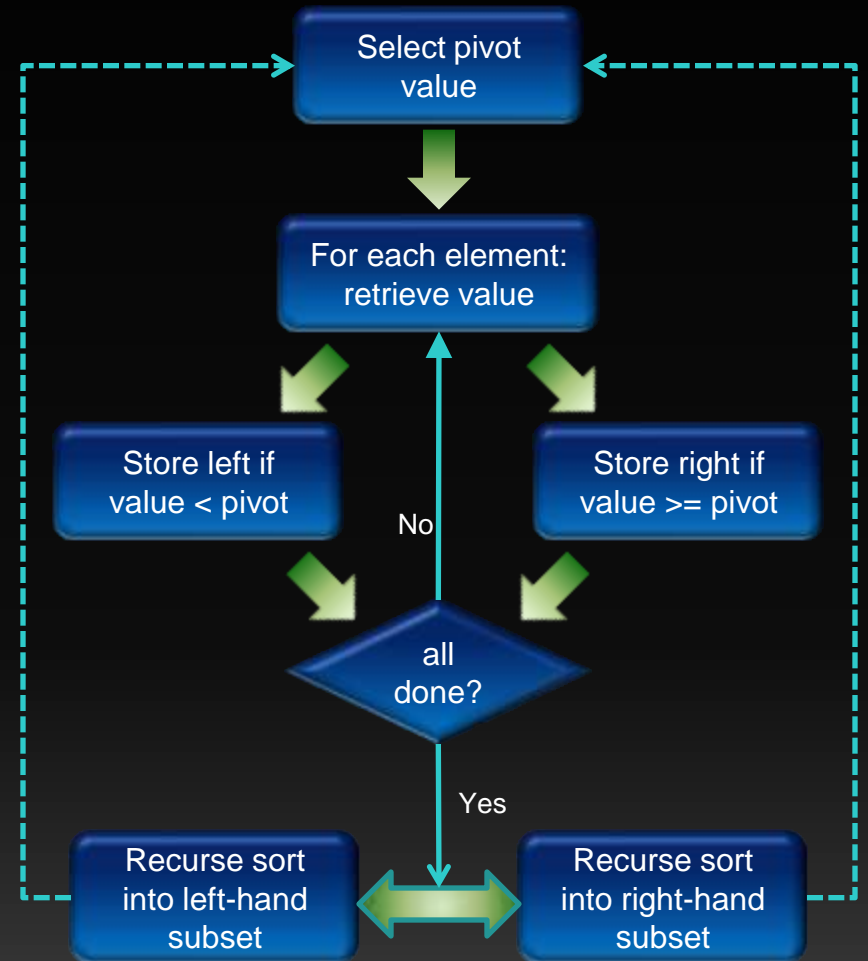
- Typical divide-and-conquer algorithm
- Recursively partition-and-sort data
- Entirely data-dependent execution
- Notoriously hard to do efficiently on Fermi

# Example 2: Parallel Recursion

```
__global__ void qsort(int *data, int l, int r)
{
    int pivot = data[0];
    int *lptr = data+l, *rptr = data+r;

    // Partition data around pivot value
    partition(data, l, r, lptr, rptr, pivot);

    // Launch next stage recursively
    if(l < (rptr-data))
        qsort<<< ... >>>(data, l, rptr-data);
    if(r > (lptr-data))
        qsort<<< ... >>>(data, lptr-data, r);
}
```





# Example 2: Parallel Recursion

```
__global__ void qsort(int *data, int l, int r)
{
    int pivot = data[0];
    int *lptr = data+l, *rptr = data+r;

    // Partition data around pivot value
    partition(data, l, r, lptr, rptr);

    // Now the recursive launch part.
    // Use streams this time!
    cudaStream_t s1, s2;
    cudaStreamCreateWithFlags(&s1, ...);
    cudaStreamCreateWithFlags(&s2, ...);

    int rx = rptr-data, lx = lptr-data;
    if(l < rx)
        qsort<<< ..., 0, s1 >>>(data, l, rx);

    if(r > lx)
        qsort<<< ..., 0, s2 >>>(data, lx, r);
}
```

Achieve concurrency by  
launching left- and right-hand  
sorts in separate streams

Compare simplicity of recursion to complexity  
of equivalent program on Fermi...

# Basic Rules

## *Programming Model*

Manifestly the same as CUDA

Launch is per-thread

Sync is per-block

CUDA primitives are per-block  
(cannot pass streams/events to children)

`cudaDeviceSynchronize() != __syncthreads()`

Events allow inter-stream dependencies

# Execution Rules

## *Execution Model*

Each block runs CUDA independently

All launches & copies are async

Constants set from host

Textures/surfaces bound only from host

ECC errors reported at host

# Memory Consistency Rules

## *Memory Model*

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

Sync implies invalidate  
(parent sees child writes after sync)

Texture changes by child are  
visible to parent after sync  
(i.e. sync == tex cache invalidate)

Constants are immutable

Local & shared memory are private:  
cannot be passed as child kernel args