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

Dynamic Parallelism
Programmability
Occupancy
Execution
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?

CPU 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
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

CUDA on Kepler

Computational Power allocated to regions of interest
Dynamic Work Generation

Initial Grid

Statically assign conservative worst-case grid

Dynamically assign performance where accuracy is required

Fixed Grid

Dynamic Grid
LU decomposition (Fermi)

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

LU decomposition (Kepler)

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

GPU Code

CPU Code
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
Batched & Nested Parallelism

Batching via Dynamic Parallelism

- Move top-level loops to GPU
- Run thousands of independent tasks
- Release CPU for other work
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);
}
Reminder: Dependencies in CUDA

```c
void main() {
    float *data;
    do_stuff(data);

    A <<< ... >>> (data);
    B <<< ... >>> (data);
    C <<< ... >>> (data);
    cudaDeviceSynchronize();

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

__global__ void B(float *data) {
    do_stuff(data);
    X <<< ... >>> (data);
    Y <<< ... >>> (data);
    Z <<< ... >>> (data);
    cudaMemcpy(data);
    do_more_stuff(data);
}
__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);
        cudaMemcpyAsync(data, buf, 1024);
    }
    __syncthreads();

    cudaMemcpyAsync(data, buf, 1024);
}
Programming Model Basics

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

Code Example

```c
__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);
        cudaMemcpyAsync(data, buf, 1024);
    }
    __syncthreads();

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

Programming Model Basics

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

```c
__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);
    cudaMemcpyAsync(data, buf, 1024);
}
```
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

```c
__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);
        cudaMemcpyAsync(data, buf, 1024);
        cudaDeviceSynchronize();
    }
    __syncthreads();
}
```
__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

```c
__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);
        cudaMemcpy(c, c, cudaMemcpyDeviceToDevice);
    }

    // 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
- `cudaDeviceSynchronize()` by launching thread
- `__syncthreads()` before consuming data
### 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