DEEP DIVE INTO DYNAMIC PARALLELISM

SHANKARA RAO THEJASWI NANDITALE, NVIDIA
CHRISTOPH ANGERER, NVIDIA
OVERVIEW AND INTRODUCTION
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
- Introduced with CUDA 5.0 and compute capability 3.5 and up

Fermi: Only CPU can generate GPU work

Kepler: GPU can generate work for itself
DYNAMIC PARALLELISM

CPU

GPU

CPU

GPU
AN EASY TO PARALLELIZE PROGRAM

```
for i = 1 to N
    for j = 1 to M
        convolution(i, j)
    next j
next i
```
A DIFFICULT TO PARALLELIZE PROGRAM
A DIFFICULT TO PARALLELIZE PROGRAM

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

Bad alternative #1: Idle Threads

Bad alternative #2: Tail Effect
DYNAMIC PARALLELISM

Serial Program

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

CUDA Program

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

void main()
{
    setup(x);
    convolution<<< N, 1 >>>(x);
}
```

With Dynamic Parallelism
EXPERIMENT

- **Device/SDK** = K40m/v7.5
- **K40m-CPU** = E5-2690

*Time (ms) lower is better*
LAUNCH EXAMPLE

Task Tracking Structures

A0 Tracking Structure

cudaLaunchDevice( B, 1, 1 );
LAUNCH EXAMPLE

Task Tracking Structures

A0 Tracking Structure

Allocate Task data structure

Grid Scheduler

Grid A

SM

A0

B<<<1,1>>()
LAUNCH EXAMPLE

Task Tracking Structures

A0 Tracking Structure

Fill out Task data structure
LAUNCH EXAMPLE

Task Tracking Structures

A0 Tracking Structure

B

Track Task B in Block A0

Grid Scheduler

Grid A

B<<<1,1>>>(())
LAUNCH EXAMPLE

Task Tracking Structures

A0 Tracking Structure

Launch Task B to GPU
LAUNCH EXAMPLE

Task Tracking Structures

A0 Tracking Structure

B

cudaLaunchDevice( C, 1, 1 );
Task Tracking Structures

A0 Tracking Structure

Allocate, fill out, and track Task C in block A0

Grid Scheduler

Grid A, Grid B

C<<<1,1>>>( )
LAUNCH EXAMPLE

Task Tracking Structures

Grid Scheduler

A0 Tracking Structure

B C

Task C is not yet runnable. Track C to run after B.
LAUNCH EXAMPLE

Task Tracking Structures

Task B completes. Scheduler kernel runs.

A0 Tracking Structure

B → C

Grid Scheduler

Grid A, Scheduler

SM

SM

SM

SM

SM

A0
LAUNCH EXAMPLE

Task Tracking Structures

A0 Tracking Structure

B -> C

Scheduler searches for work.
LAUNCH EXAMPLE

Task Tracking Structures

A0 Tracking Structure

Scheduler completes B, and identifies C as ready-to-run.
Task Tracking Structures

Scheduler frees B for re-use, and launches C to the Grid Scheduler.
LAUNCH EXAMPLE

Task Tracking Structures

A0 Tracking Structure

Task C now executes.
BASIC RULES

Programming Model

Essentially the same as CUDA

Launch is per-thread and asynchronous

Sync is per-block

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

cudaDeviceSynchronize() != __syncthreads()

Events allow inter-stream dependencies

Streams are shared within a block
Implicit NULL stream results in ordering within a block; use named streams

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
EXPERIMENTS
DIRECTED BENCHMARKS

Kernels written to measure specific aspects of dynamic parallelism

- Launch throughput
- Launch latency

As a function of different configurations

- SDK Versions
- Varying Clocks
RESULTS - LAUNCH THROUGHPUT
**LAUNCH THROUGHPUT**

* Device/SDK/mem-clk, gpu-clk = K40m/v7.5/875
* K40m-CPU = E5-2690
* Host launches are with 32 streams
LAUNCH THROUGHPUT

Observations

About an order of magnitude higher than from host

Dynamic parallelism is very useful when there are a lot of child kernels

Two major limiters of launch throughput

  Pending Launch Count

  Grid Scheduler Limit
PENDING LAUNCH COUNT

* Device/SDK/mem-clk: K40/v7.5/3004,875
* Different curves represent different pending launch count limits
Pre-allocated buffer in *Global Memory* to store kernels before their launch

Default value - 2048 kernels

Buffer overflow implies resize performed on-the-go

  Substantial reduction in launch throughput!

Know the number of pending child kernels!
CUDA API’S

Pending Launch Count

Setting Limit

```c
cudaDeviceSetLimit(cudaLimitDevRuntimePendingLaunchCount, yourLimit);
```

Querying Limit

```c
cudaDeviceGetLimit(&yourLimit, cudaLimitDevRuntimePendingLaunchCount);
```
GRID SCHEDULER LIMIT

* Different curves represent the total number of child kernels launched

* Device/SDK/mem-clk, gpu-clk = K40/v7.5/3004,875
GRID SCHEDULER LIMIT

Observations

Ability of grid scheduler to track the number of concurrent kernels

The limit is currently 32

If this limit is crossed, upto 50% loss in launch throughput
RESULTS - LAUNCH LATENCY
LAUNCH LATENCY

Device/SDK/mem-clk, gpu-clk = K40m/v7.5/3004,875
K40m-CPU = ES-2690
Host launches are with 32 streams
LAUNCH LATENCY

Observations

Initial and subsequent latencies are about 2-3x slower than that of host

Dynamic Parallelism may not be a good choice currently when:

- A few child kernels
- Serial kernel launches

We are working towards improving this**

LAUNCH LATENCY - STREAMS

*Device/SDK/mem-clk, gpu-clk = K40m/v7.5/3004,875*
LAUNCH LATENCY - STREAMS

Observations

Host streams affect device-side launch latency

Prefer device streams for dynamic parallelism
RESULTS - DEVICE SYNCHRONIZE
cudaDeviceSynchronize is costly
Avoid it when possible, example below

```c
__global__ void parent() {
    doSomeInitialization();
    childKernel<<<grid,blk>>>();
    cudaDeviceSynchronize();
}
```

Unnecessary. Implicit join enforced by the programming model!
DEVICE SYNCHRONIZE - COST

* Device/SDK = K40/v7.5
DEVICE SYNCHRONIZE DEPTH

Deepest recursion level until where `cudaDeviceSynchronize` works

CUDA limit `cudaLimitDevRuntimeSyncDepth` controls it

Default is level 2

At the cost of extra global memory reserved for storing parent blocks
DEVICE SYNCHRONIZE DEPTH

Memory Usage

Memory Reserved (MB)

Device Synchronize Depth
DEVICE SYNCHRONIZE DEPTH

Error Handling

cudaDeviceSynchronize fails **silently** beyond the set SyncDepth

Use *cudaGetLastError* on **device** to inspect the error

![Diagram showing error handling with SyncDepth=2](image-url)
DYNAMIC PARALLELISM - LIMITS
DYNAMIC PARALLELISM

Limits

Recursion depth is currently 24

Maximum size of formal parameters in the child kernel is 4096 B

Violation causes a compile-time error

Runtime exceptions in child kernel are only visible from host-side
ERROR HANDLING
Runtime exceptions in child kernels

Visible only from host-side

-lineinfo of nvcc along with cuda-memcheck to locate the error location

```c
__global__ void child(float* arr) {
    arr[0] = 1.0f;
}

__global__ void parent() {
    child<<<1,1>>>(NULL);
    cudaDeviceSynchronize();
    printf("%d\n", cudaGetLastError());
}

parent<<<1,1>>>(

cudaError_t err = cudaDeviceSynchronize();
```

Error caught here

Control never reaches here!
SUCCESS STORIES
FMM
Fast Multipole Method

- Solving the N-body problem
- Computational complexity $O(n)$
- Tree-based approach

Image source: http://www.bu.edu/pasi/courses/12-steps-to-having-a-fast-multipole-method-on-gpus/
FMM (2)

Performance

- Dynamic 1: launch child grids for neighbors and children
- Dynamic 2: launch child grids for children only
- Dynamic 3: launch child grids for children only; start only $p^2$ kernel threads; use shared GPU memory

From: FMM goes GPU A smooth trip or bumpy ride?, B. Kohnke, I. Kabadshow MPI BPC Göttingen & Jülich Supercomputing Centre, GTC2015
PANDA
anti-Proton ANnihilation at DArmstadt

- State-of-the-art hadron particle physics experiment
PANDA (2)

Performance and Reasons for Improvements

- Avoiding extra PCI-e data transfers.
  - Launch configuration data dependencies
- Higher launch throughput
- Reducing false dependencies between kernel launches.
  - Waiting on stream prevents enqueuing of work into other streams

Source: A CUDA Dynamic Parallelism Case Study: PANDA, Andrew Adinetz
http://devblogs.nvidia.com/parallelforall/a-cuda-dynamic-parallelism-case-study-panda/
SUMMARY
WHEN TO USE CUDA DYNAMIC PARALLELISM

Three Good Reasons

• Algorithmic: “Dynamically Formed Pockets of Structured Parallelism”*
  - Unbalanced load (e.g., vertex expansion in graphs, compressed sparse row)
  - Tree traversal (fat and shallow computation trees)
  - Adaptive Mesh Refinement

• Performance:
  - Improve launch throughput
  - Reduce PCIe traffic and false dependencies

• Maintenance:
  - Simplified, more natural program flow

REFERENCES


- FMM goes GPU, B. Kohnke and I. Kabadshow, GTC 2015, https://shar.es/1Y38Vf
THANK YOU

JOIN THE CONVERSATION
#GTC16  

JOIN THE NVIDIA DEVELOPER PROGRAM AT developer.nvidia.com/join