CUDA Dynamic Parallelism

Objective

- To understand the CUDA Dynamic Parallelism (CDP) execution model, including synchronization and memory model
- To understand the benefits of CDP in terms of productivity, workload balance and memory regularity
- To understand the launching path of child kernels in the microarchitectural level
- To understand the overhead of CDP
Reading

- CUDA Programming Guide. Appendix C “CUDA Dynamic Parallelism”.

Recap: CUDA Execution Model

- Kernels/Grids are launched by host (CPU)
- Grids are executed on GPU
- Results are returned to CPU
Launching from device

- Kernels/Grids can be launched by GPU

Code Example

```c
__global__ void k1() {
    ... //get result of k1
    if(result) {
        k2<<<1, 1>>>();
    }
}

int main() {
    k1<<<1,1>>>();
    ... //get result of k1
    if(result) {
        k2<<<1, 1>>>();
        k1<<<1,1>>>();
    }
}
```

Launching k2 from CPU  
Launching k2 from GPU
CUDA Dynamic Parallelism (CDP)

- Introduced in NVIDIA Kepler GK110
- Launch workload directly from GPU
- Launch can be nested

Q1: How many k2 are launched? 32
Q2: How to change the code to launch only one k2?

CDP (2)

- Launches are per thread
  - Each thread can launch new kernels
  - Each thread can make nested launches

```c
__global__ void k1() {
    if(threadIdx.x == 0)
        k2<<<1, 1>>>();
}

__global__ void k2() {
}

int main() {
...
    k1<<<1,32>>>();
}
```

(7)
Compilation

- Needs special option to compile
  
nvcc test.cu -o test -arch=sm_35 -rdc=true -lcudadevrt

- `sm_35`: must be on device that have compute capability > 3.5
- `-rdc=true`: generate relocatable device code, must be used for CDP program
- `-lcudadevrt`: link CUDA device runtime library. Can be omitted on >CUDA 6.0

- Demo
  
  https://github.gatech.edu/jwang323/gpu_class/blob/master/src/cdp.cu

PTX Interface

- The “<<< >>>” in device code will be compiled into two PTX APIs
  - `cudaGetParameterBuffer` (cudaGetParameterBufferV2)
  - `cudaLaunchDevice` (cudaLaunchDeviceV2)

  call.uni (retval0), cudaGetParameterBufferV2,{
  param0,param1,param2,param3};

  call.uni (retval0),cudaLaunchDeviceV2,{
  param0,param1};

More on CUDA Programming Guide Appendix C
• Child execution is not guaranteed unless synchronization
  - Explicit
  - Implicit

```c
__global__ void k1() {
    if (threadIdx.x == 0)
        k2<<<1, 1>>>()
}
```

Q1: Does k2 start execution immediately after it is launched?

Don’t know!

```c
__global__ void k2() {
    
    int main() {
        ...
        k1<<<1,32>>>()
    }
```

CDP Synchronization (2)

• Explicit synchronization
  - API: cudaDeviceSynchronize()
  - All child kernels launched in the thread block before the sync API immediately start execution
  - If not enough GPU resources, parent kernel is suspended to yield to children
  - Blocking parent until child kernels finish

```c
__global__ void k1() {
    if (threadIdx.x == 0)
        k2<<<1, 1>>>();
    cudaDeviceSynchronize();
}
```
CDP Synchronization (3)

- cudaDeviceSynchronize and __syncthreads
  - Do NOT imply each other!

```c
__global__ void k1() {
  if(threadIdx.x == 0) k2<<<1, 1>>();
  // Only sync with child kernels, not other parent threads
  cudaDeviceSynchronize();
  // still necessary for parent barrier
  __syncthreads();
  // k2 result consumed by all parent threads
  consumeK2Result();
}
```

K2 result visible to (same block) thread 0? N Other threads? N

Other threads?

- How about this?

```c
__global__ void k1() {
  if(threadIdx.x == 0) k2<<<1, 1>>();
  __syncthreads();
  cudaDeviceSynchronize();
}
```

K2 result visible to (same block) thread 0? N Other threads? N

Other threads?

---

(13)
CDP Synchronization (4)

- Implicit Synchronization
  - If no explicit sync, child kernel must start execution when parent kernel execution ends
  - Parent kernel is not finished until all child kernels are finished

```c
__global__ void k1() {
    if(threadIdx.x == 0)
        k2<<<B2, T2>>>();

    otherOps();
}
```

```
int main() {
    k1<<<B1,T1>>>();
    k3<<<B3,T3>>>();
}
```

Q: When can k3 start execution? 
After k1 and k2 finish.

CDP Memory Model

- Global memory: visible to both parent and child
- Shared memory and local memory: private to parent/child, cannot be passed to each other.

```c
__global__ void k1(int * glmem) {
    __shared__ int shmem[...];

    k2<<<B2,T2>>>(shmem) //undefined behavior
    k2<<<B2,T2>>>(glmem) //ok
}
```
CDP Memory Model (2)

- Global memory is fully consistent when:
  - Child is invoked (parent->child)
  - Child completes after cudaDeviceSynchronize is invoked (child->parent)

```c
__global__ void k1(int * glmem) {
    // glmem is fully consistent between k1 and k2
    k2<<<B2,T2>>>(glmem)
    // No consistency guarantee
    cudaDeviceSynchronize();
    // glmem is fully consistent between k1 and k2
}
```

(17)

CDP Benefits

- What are the issues with the following CUDA code?
- Hint: control flow and memory accesses

```c
__global__ void k1(int * iterations, int * offsets, int * vals) {
    int it = iterations[threadIdx.x];
    int offset = offsets[threadIdx.x];
    for(int i = 0; i < it; i++) {
        val = vals[offset + i];
        process(val);
    }
}
```

(18)
Work load imbalance causes control flow divergence
Different offsets cause non-coalesced memory accesses

- Use CDP to implement
  - Launch a child kernel from each parent thread
  - Launch only when sufficient parallelism
CDP Benefits (3)

- Reduce control divergence
  - Reduce workload imbalance in parent kernel
  - Uniform control flow in child kernels

- Increase coalesced memory accesses

CDP Benefits (4)

- Compared with other optimization to reduce control and memory divergence:
  - Productivity
  - Fewer lines of code

```c
__global__ void k1(int * iterations,
                   int * offsets,
                   int * vals) {
    int it = iterations[threadIdx.x];
    int offset = offsets[threadIdx.x];

    if(it>para_threshold)
        child<<1,it>>>(vals, offset);
}
```
CDP Benefits (5)

- Other benefits (from S. Jones’ slides)
  - Data-dependent execution
  - Dynamic load balancing
  - Dynamic library kernel calls
  - Recursive kernel calls
  - Simplify CPU/GPU divide

Recap: Host-launched Kernel Execution
Device Kernel Launching

- Managed by Hardware (uArch) and Software (device driver)
- Pending kernel management in KMU
  - Managing pending child kernels and suspended kernels (due to parent-child yielding)
  - Kernel information are stored in memory
  - Tracked by driver

Warning: information speculated according to NVIDIA's documentations
Device Kernel Launching (3)

- New Launching path from SMX
  - Driver (software) initializes the child kernels and stores their information in the memory
  - SMX (hardware) notifies Kernel Management Unit
- Device kernel execution
  - Same as host-launched kernels

Device Kernel Launching (4)

- Parent-Child Synchronization
  - Device drive processes the synchronization request and perform necessary actions
    - Suspend parent kernels
    - Start execute child kernels
    - Complete parent kernels
  - SMX issues corresponding commands to KMU
CDP Launching Overhead

- Measured from CDP Program directly
  - Device kernel launching time ~35us
  - Recap: host kernel launching time ~10us

- Hardware (uArch)
  - Extra launching path

- Software (device driver)
  - Similar as driver for host-launched kernels, but executed on device
  - Kernel preallocation
  - Resource management

- No detailed breakdown

Warning: information speculated according to NVIDIA's documentations (29)

CDP Launching Overhead (2)

- Scales with kernel launch count
- 98.57ms for 256K kernel launches on K40 (VS nonCDP BFS kernel 3.27ms)
- 36.1%-80.6% of total execution time

![Graph showing CDP Launching Overhead vs Kernel Launching Count]
Potential benefits in control flow and memory behavior

However, overhead causes slow down instead of speedup

From Wang and Yalamanchili, IISWC’14

CDP Memory Footprint

- Require preallocation for child kernel information
  - ~8KB per child kernel launch

- Require preallocation for suspended parent status
  - Register
  - Shared memory
  - ~100MB per explicit synchronization level (cudaDeviceSynchronize)
CDP memory footprint

- Reserved global memory for kernel launch
  - scale with kernel count
  - Max 1.2GB reserved (K20c has 5GB global memory)

Summary

- CUDA Dynamic Parallelism (CDP) supports launching kernels from the GPU
- Good for productivity, control flow and memory behavior (irregular applications)
- Supported by extended uArch and driver
- Non-trivial overhead overshadows the performance benefit
Dynamic Thread Block Launch (DTBL)

Wang, Rubin, Sidelnik and Yalamanchili, [ISCA 2015]

Objectives

• To understand DTBL execution model
• To understand the microarchitectural support for DTBL
• To understand the benefits of DTBL
Example of Irregular Applications

- **Dynamically Formed** structured pockets of **Parallelism (DFP)**

Adaptive Mesh Refinement (AMR)          Graph Traversal

**Pockets of Structured Parallelism**

Workload Imbalance

Recap: CDP for Irregular Applications

- Launch a child kernel from each parent thread when sufficient parallelism

Parent Kernel: 10 11 12 13 14 15 16 17
Child Kernels:

Launched by t1, t2, t3, and t6
Features of DFP

- High Dynamic Workload Density: \(1000\sim200,000\) pockets of structured parallelism (\# of dynamic kernels)

- High kernel launching overhead

Features of DFP (2)

- Low Compute Intensity: Average degree of parallelism in dynamic workload is \(~40\) (\# of threads in dynamic kernels)

- Low GPU occupancy and utilization
Features of DFP (3)

- Workload Similarity: most dynamic workloads execute the same kernel code (with different configuration, i.e. TB size, shared memory size, etc...)

Dynamic Thread Block Launch (DTBL)

- Motivation:
  - Light-weight, efficient programming model and microarchitecture for DFP

- Propose: DTBL
  - Extend the current GPU execution model
  - Threads can launch fine-grained TBs on demand instead of kernels

- Goal:
  - Increase SMX execution efficiency for DFP
  - Minimize hardware and runtime overhead
DTBL Execution model

- Each thread in a kernel can launch new TBs
- New TBs are coalesced with the existing kernel
- **Coalescing:** new TBs belong to the same kernel and scheduled together

User Interface and Code Example

- **Program Interface:** One additional CUDA Device Runtime API

  ```c
  cudaMemcpyAsync

  __global__ void K1(...) {
      void * param = cudaMemcpyParameterBuffer(...);
      cudaMemcpyAsync(param, ...);
  }
  ```
Revisit: Graph Traversal

- Launch new TBs for DFP
  - All the TBs are doing the same work
- Launch only when sufficient parallelism

Parent Kernel

Dynamically Launched TBs

Launched by t1
Launched by t2
Launched by t3
Launched by t4

Similar benefit as in CDP for control flow and memory behavior

Revisit: Graph Traversal

- Launch new TBs for DFP
  - All the TBs are doing the same work
- Launch only when sufficient parallelism

Parent Kernel

DTBL: Coalesced with a new Kernel

Launched by t1
Launched by t2
Launched by t3
Launched by t4
TB Coalescing and Scheduling

- Achieve high SMX occupancy and utilization
- Recall DFP are fine-grained dynamic workloads with high density.
- TB coalescing allows enough TBs to be aggregated into one big kernel and scheduled to SMXs.
- In comparison: CDP only allows 32 concurrent kernel execution. SMXs are not fully occupied for small kernels.

Examples for TB coalescing

- One TB has 64 threads or 2 warps (in CDP, that is one kernel)
- On K20c GK110 GPU: 13 SMX, max 64 warps/SMX, max 32 concurrent kernels

CDP: All child kernels

SMXs

Kernel 1  Kernel 2  Kernel 3  .....  Kernel 100

SMX0  SMX1  SMX12
Examples for TB coalescing

- One TB has 64 threads or 2 warps (in CDP, that is one kernel)
- On K20c GK110 GPU: 13 SMX, max 64 warps/SMX, max 32 concurrent kernels

CDP: All child kernels

<table>
<thead>
<tr>
<th>SMXs</th>
<th>CDP: All child kernels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 1</td>
<td>Kernel 2</td>
</tr>
<tr>
<td>SMX0</td>
<td>SMX1</td>
</tr>
</tbody>
</table>

Examples for TB coalescing

- One TB has 64 threads or 2 warps (in CDP, that is one kernel)
- On K20c GK110 GPU: 13 SMX, max 64 warps/SMX, max 32 concurrent kernels

CDP: All child kernels

<table>
<thead>
<tr>
<th>SMXs</th>
<th>CDP: All child kernels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 1</td>
<td>Kernel 2</td>
</tr>
<tr>
<td>SMX0</td>
<td>SMX1</td>
</tr>
</tbody>
</table>

(50)
Examples for TB coalescing

- One TB has 64 threads or 2 warps (in CDP, that is one kernel)
- On K20c GK110 GPU: 13 SMX, max 64 warps/SMX, max 32 concurrent kernels

CDP: All child kernels

<table>
<thead>
<tr>
<th>SMXs</th>
<th>CDP: All child kernels</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMX0</td>
<td>Kernel 14</td>
</tr>
<tr>
<td>SMX1</td>
<td>Kernel 15</td>
</tr>
<tr>
<td>SMX2</td>
<td>Kernel 16</td>
</tr>
<tr>
<td>SMX3</td>
<td>Kernel 17</td>
</tr>
<tr>
<td>SMX4</td>
<td>Kernel 18</td>
</tr>
<tr>
<td>SMX5</td>
<td>Kernel 19</td>
</tr>
<tr>
<td>SMX6</td>
<td>Kernel 20</td>
</tr>
<tr>
<td>SMX7</td>
<td>Kernel 21</td>
</tr>
<tr>
<td>SMX8</td>
<td>Kernel 22</td>
</tr>
<tr>
<td>SMX9</td>
<td>Kernel 23</td>
</tr>
<tr>
<td>SMX10</td>
<td>Kernel 24</td>
</tr>
<tr>
<td>SMX11</td>
<td>Kernel 25</td>
</tr>
<tr>
<td>SMX12</td>
<td>Kernel 26</td>
</tr>
<tr>
<td>SMX13</td>
<td>Kernel 27</td>
</tr>
<tr>
<td>SMX14</td>
<td>Kernel 28</td>
</tr>
<tr>
<td>SMX15</td>
<td>Kernel 29</td>
</tr>
<tr>
<td>SMX16</td>
<td>Kernel 30</td>
</tr>
<tr>
<td>SMX17</td>
<td>Kernel 31</td>
</tr>
<tr>
<td>SMX18</td>
<td>Kernel 32</td>
</tr>
<tr>
<td>SMX19</td>
<td>Kernel 33</td>
</tr>
<tr>
<td>SMX20</td>
<td>Kernel 34</td>
</tr>
<tr>
<td>SMX21</td>
<td>Kernel 35</td>
</tr>
<tr>
<td>SMX22</td>
<td>Kernel 36</td>
</tr>
<tr>
<td>SMX23</td>
<td>Kernel 37</td>
</tr>
<tr>
<td>SMX24</td>
<td>Kernel 38</td>
</tr>
<tr>
<td>SMX25</td>
<td>Kernel 39</td>
</tr>
<tr>
<td>SMX26</td>
<td>Kernel 40</td>
</tr>
<tr>
<td>SMX27</td>
<td>Kernel 41</td>
</tr>
<tr>
<td>SMX28</td>
<td>Kernel 42</td>
</tr>
<tr>
<td>SMX29</td>
<td>Kernel 43</td>
</tr>
<tr>
<td>SMX30</td>
<td>Kernel 44</td>
</tr>
<tr>
<td>SMX31</td>
<td>Kernel 45</td>
</tr>
<tr>
<td>SMX32</td>
<td>Kernel 46</td>
</tr>
<tr>
<td>SMX33</td>
<td>Kernel 47</td>
</tr>
<tr>
<td>SMX34</td>
<td>Kernel 48</td>
</tr>
<tr>
<td>SMX35</td>
<td>Kernel 49</td>
</tr>
<tr>
<td>SMX36</td>
<td>Kernel 50</td>
</tr>
<tr>
<td>SMX37</td>
<td>Kernel 51</td>
</tr>
<tr>
<td>SMX38</td>
<td>Kernel 52</td>
</tr>
<tr>
<td>SMX39</td>
<td>Kernel 53</td>
</tr>
<tr>
<td>SMX40</td>
<td>Kernel 54</td>
</tr>
<tr>
<td>SMX41</td>
<td>Kernel 55</td>
</tr>
<tr>
<td>SMX42</td>
<td>Kernel 56</td>
</tr>
<tr>
<td>SMX43</td>
<td>Kernel 57</td>
</tr>
<tr>
<td>SMX44</td>
<td>Kernel 58</td>
</tr>
<tr>
<td>SMX45</td>
<td>Kernel 59</td>
</tr>
<tr>
<td>SMX46</td>
<td>Kernel 60</td>
</tr>
<tr>
<td>SMX47</td>
<td>Kernel 61</td>
</tr>
<tr>
<td>SMX48</td>
<td>Kernel 62</td>
</tr>
<tr>
<td>SMX49</td>
<td>Kernel 63</td>
</tr>
<tr>
<td>SMX50</td>
<td>Kernel 64</td>
</tr>
<tr>
<td>SMX51</td>
<td>Kernel 65</td>
</tr>
<tr>
<td>SMX52</td>
<td>Kernel 66</td>
</tr>
<tr>
<td>SMX53</td>
<td>Kernel 67</td>
</tr>
<tr>
<td>SMX54</td>
<td>Kernel 68</td>
</tr>
<tr>
<td>SMX55</td>
<td>Kernel 69</td>
</tr>
<tr>
<td>SMX56</td>
<td>Kernel 70</td>
</tr>
<tr>
<td>SMX57</td>
<td>Kernel 71</td>
</tr>
<tr>
<td>SMX58</td>
<td>Kernel 72</td>
</tr>
<tr>
<td>SMX59</td>
<td>Kernel 73</td>
</tr>
<tr>
<td>SMX60</td>
<td>Kernel 74</td>
</tr>
<tr>
<td>SMX61</td>
<td>Kernel 75</td>
</tr>
<tr>
<td>SMX62</td>
<td>Kernel 76</td>
</tr>
<tr>
<td>SMX63</td>
<td>Kernel 77</td>
</tr>
<tr>
<td>SMX64</td>
<td>Kernel 78</td>
</tr>
<tr>
<td>SMX65</td>
<td>Kernel 79</td>
</tr>
<tr>
<td>SMX66</td>
<td>Kernel 80</td>
</tr>
<tr>
<td>SMX67</td>
<td>Kernel 81</td>
</tr>
<tr>
<td>SMX68</td>
<td>Kernel 82</td>
</tr>
<tr>
<td>SMX69</td>
<td>Kernel 83</td>
</tr>
<tr>
<td>SMX70</td>
<td>Kernel 84</td>
</tr>
<tr>
<td>SMX71</td>
<td>Kernel 85</td>
</tr>
<tr>
<td>SMX72</td>
<td>Kernel 86</td>
</tr>
<tr>
<td>SMX73</td>
<td>Kernel 87</td>
</tr>
<tr>
<td>SMX74</td>
<td>Kernel 88</td>
</tr>
<tr>
<td>SMX75</td>
<td>Kernel 89</td>
</tr>
<tr>
<td>SMX76</td>
<td>Kernel 90</td>
</tr>
<tr>
<td>SMX77</td>
<td>Kernel 91</td>
</tr>
<tr>
<td>SMX78</td>
<td>Kernel 92</td>
</tr>
<tr>
<td>SMX79</td>
<td>Kernel 93</td>
</tr>
<tr>
<td>SMX80</td>
<td>Kernel 94</td>
</tr>
<tr>
<td>SMX81</td>
<td>Kernel 95</td>
</tr>
<tr>
<td>SMX82</td>
<td>Kernel 96</td>
</tr>
<tr>
<td>SMX83</td>
<td>Kernel 97</td>
</tr>
<tr>
<td>SMX84</td>
<td>Kernel 98</td>
</tr>
<tr>
<td>SMX85</td>
<td>Kernel 99</td>
</tr>
<tr>
<td>SMX86</td>
<td>Kernel 100</td>
</tr>
</tbody>
</table>

(51)
Examples for TB coalescing

- One TB has 64 threads or 2 warps (in CDP, that is one kernel)
- On K20c GK110 GPU: 13 SMX, max 64 warps/SMX, max 32 concurrent kernels

CDP: All child kernels

<table>
<thead>
<tr>
<th>SMXs</th>
<th>CDP: All child kernels</th>
<th>Wait till previous kernel finish</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMX0</td>
<td>Kernel 27</td>
<td></td>
</tr>
<tr>
<td>SMX1</td>
<td>Kernel 28</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Kernel 33</td>
<td></td>
</tr>
</tbody>
</table>

Achieved SMX Occupancy: 32 kernels * 2 warps/kernel / (64 warps/SMX * 13 SMXs)

= 0.077

SMX, max 32 concurrent kernels

CDP: All child kernels

<table>
<thead>
<tr>
<th>SMXs</th>
<th>CDP: All child kernels</th>
<th>Wait till previous kernel finish</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMX0</td>
<td>Kernel 27</td>
<td></td>
</tr>
<tr>
<td>SMX1</td>
<td>Kernel 28</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Kernel 33</td>
<td></td>
</tr>
</tbody>
</table>

(53)
Examples for TB coalescing

- One TB has 64 threads or 2 warps (in CDP, that is one kernel)
- On K20c GK110 GPU: 13 SMX, max 64 warps/SMX, max 32 concurrent kernels

**DTBL:** All TBs coalesced with one kernel. No restrictions in scheduling.

<table>
<thead>
<tr>
<th>SMXs</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SMX0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMX1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMX12</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

(55)
Examples for TB coalescing

- Achieved SMX Occupancy: $100 \times 2 \text{ warps/TB} / (64 \text{ warps/SMX} \times 13 \text{ SMXs}) = 0.24$
- Can be increased if more TBs!

Microarchitecture (Baseline GPU)
Microarchitecture Extension

Newly launched TBs

TB Coalescing on Microarchitecture

Tracking new TBs

Coalescing and scheduling new TBs
TB Coalescing on Microarchitecture

- Kernel Distributor
- Aggregated Group Table
- SMX Scheduler
- Memory Controller
- PC Dim Param ExeBL

Scheduled together to increase SMX Occupancy

TB Launching Latency

- Exclusive launching path from SMXs
- Simple runtime and driver support with low latency
  - Kernel information only has to be managed once
  - New TBs can reuse the kernel information
- In comparison, CDP uses
  - Longer launching path
  - More complicated runtime and driver for kernel management
Launch Path for CDP and DTBL

Launch path for CDP: ~70k cycles
Launch path for DTBL: ~10k cycles
(32 launches per warp)
### Speedup of DTBL

- **Ideal**: excluding launch latency, 1.63x over flat implementations and 1.14x over CDP (Benefits from Higher GPU Utilization)
- **Actual**: including launch latency, 1.21x over flat implementations and 1.40x over CDP (Benefits from less launching overhead)  

### Summary

- **DTBL**: New extension to GPU execution Model
- Efficient solution for **Dynamic Parallelism** in irregular applications

- Shows more benefit for dynamic, **fine-grained but sufficient** parallelism
- Dynamic **launching latency** is reduced from CDP