ACHIEVING DETERMINISTIC EXECUTION TIMES IN CUDA APPLICATIONS

Aayush Rajoria, Ashok Kelur
20th March 2019
CONTENTS

• CUDA Everywhere
• Deterministic Execution times
• Automotive Trade-offs
• Application execution flow
• Factors affecting Runtime determinism
CUDA EVERYWHERE
DETERMINISTIC EXECUTION TIMES

- Automotive use-cases and deterministic execution.
- How to write CUDA applications which are deterministic in nature.
AUTOMOTIVE TRADE-OFFS

- Determinism over Ease of programming
  - Example: `cudaMalloc` over `cudaMallocManaged` (CUDA unified memory)

- Determinism over GPU utilization
  - Example: Single context over MPS
AUTOMOTIVE TRADE-OFFS

• Different trade-offs need to be considered for every CUDA functionality.
  • Some CUDA functionality might be more deterministic than others.
• Trade-offs could be different for different phases of an application’s lifecycle.
• One simple application lifecycle is as below.

INIT

DETERMINISM

DEINIT

INNER LOOP
APPLICATION EXECUTION FLOW

// Init phase
Initialize Camera;
Do All memory allocation;
Sets up all the dependencies;

// Runtime phase
While() {
    Inference_Kernel<<< ..., stream1 >>>();
    Decision_Kernel<<< ..., stream1 >>>();
}

// Deinit phase
Free memory;
Free all the system resources;
FACTORS AFFECTING DETERMINISM OF THE RUNTIME PHASE
FACTORS AFFECTING DETERMINISM OF THE RUNTIME PHASE

- GPU work submission.
- GPU work scheduling
- Other factors
GPU work submission APIs are the most frequently used APIs in the runtime phase.

CUDA driver has done various improvements for making the GPU work submission time deterministic over the past few years.

CUDA DRIVER IMPROVEMENTS

<table>
<thead>
<tr>
<th>CUDA Versions</th>
<th>Avg submit time in us</th>
<th>Standard Deviation in us</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA 8.x</td>
<td>22.8</td>
<td>16.3</td>
</tr>
<tr>
<td>CUDA 9.x</td>
<td>7.17</td>
<td>5.1</td>
</tr>
<tr>
<td>CUDA 10.x</td>
<td>3.52</td>
<td>1.55</td>
</tr>
</tbody>
</table>

Source: Nvidia Internal micro benchmark ran on a Drive platforms on QNX
GPU WORK SUBMISSION

SUGGESTIONS FOR APPLICATIONS

• Using less number of GPU work submission to solve the problem at hand is always more deterministic as compared to more number of GPU work submissions.

• Number of GPU work submission can be reduced by:
  • Kernel fusion
  • CUDA graphs
GPU WORK SUBMISSION

Kernel Fusion

1. colorConversion_YUV.RGB<<< >>> ()
2. imageHistogram<<< >>> ()
3. edgeDetection<<< >>> ()

With Kernel Fusion

1. __device__ colorConversion_YUV.RGB()
2. __device__ imageHistogram()
3. __device__ edgeDetection()
4.
5. fusedKernel <<< >>> () {
6. colorConversion_YUV.RGB();
7. imageHistogram();
8. edgeDetection();
9. }
GPU WORK SUBMISSION

CUDA graphs

• CUDA graphs helps in batching multiple kernels, memcpy, memset into an optimal number of GPU work submission.

• CUDA graphs allows application to describe GPU work and its dependencies ahead of time. This allows CUDA driver to do all resource allocation ahead of the time.
GPU WORK SUBMISSION

Three-Stage execution model

Define + Instantiate

Execute

Destroy

Execution flow for deterministic applications.
EXECUTION OPTIMIZATIONS
Latency & Overhead Reductions

Launch latencies:

- Pre-defined graph allows launch of any number of kernels in one single operation

Source: Nvidia Internal benchmarks ran on a Drive platforms on QNX
HOST ENQUEUE TIME COMPARISON

Batching GPU work using CUDA graphs.

ResNet50 INT8: Host Enqueue time = 0.61 ms
ResNet152 INT8: Host Enqueue time = 1.49 ms
MobileNet INT8: Host Enqueue time = 0.31 ms

Source: Nvidia benchmarks ran on a Drive platforms on QNX with CUDA10.1
GPU WORK SCHEDULING
GPU WORK SCHEDULING

GPU Context switches

• Tasks in two GPU contexts can preempt each other which can affect the determinism of the application.
  
  • It is advised not to create multiple CUDA contexts on the same device in the same process.
  
  • In case the application has multiple contexts in the same process, the dependency between them can be established with:
    
    • cudaStreamWaitEvent()
  
  • In case the application has multiple contexts in different process, the dependency between them can be established with:
    
    • EGLSTREAMS
GPU WORK SCHEDULING

GPU Context switches

Context Save-Restore time
Inserted Dependency
CTX1
CTX2

Expected Deadline for Task1
Achieved Deadline for Task1
Saved time
Explicit Dependency
WORK SCHEDULING

CPU thread scheduling

If the CPU thread scheduling the GPU work gets switched out then it can result in increase in the launch overhead.

Potential solutions:

• Pin the CPU thread to the core and increase the thread priority of the thread submitting CUDA work

• Have a custom scheduler which guarantees that the CPU thread is active on a CPU core while submitting CUDA kernels
WORK SCHEDULING

CPU thread scheduling

Thread 1
- Launch A
  - CPU WORK
- Launch B
  - Launch C
  - CPU WORK
- Launch D
- Launch E

Thread 2
- Launch A
  - GPU IDLE
- Launch B
  - B
- Launch C
  - C
  - GPU IDLE
- Launch D
  - D
- Launch E
  - E

Thread 3
- Launch A
  - CPU WORK
- Launch B
  - B
- Launch C
  - C
- Launch D
  - CPU WORK
- Launch E
  - CPU WORK

Actual Finish
- Launch E

Expected Finish
- Launch E
WORK SUBMISSION ON NULL/DEFAULT STREAM

<table>
<thead>
<tr>
<th>Streams</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>- Default</td>
<td></td>
</tr>
<tr>
<td>- Stream 13</td>
<td>kernel...</td>
</tr>
<tr>
<td>- Stream 14</td>
<td>kernel...</td>
</tr>
<tr>
<td>- Stream 15</td>
<td>kernel...</td>
</tr>
<tr>
<td>- Stream 16</td>
<td>kernel...</td>
</tr>
</tbody>
</table>

Streams

<table>
<thead>
<tr>
<th>Streams</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>- Stream 13</td>
<td>kernel(float*, int)</td>
</tr>
<tr>
<td>- Stream 14</td>
<td>kernel(float*, int)</td>
</tr>
<tr>
<td>- Stream 15</td>
<td>kernel(float*, int)</td>
</tr>
<tr>
<td>- Stream 16</td>
<td>kernel(float*, int)</td>
</tr>
</tbody>
</table>
OTHER FACTORS
CUDA STREAM CALLBACKS

• cudaStreamCallback runs a CPU function in a helper thread in a stream order.

• Do not use cudaStreamAddCallback / cuStreamAddCallback.
  • It involves GPU interrupt latency
  • Application does not have control on the thread which executes callback.

• Potential solution:
  • Use explicit CPU synchronization to schedule the dependent CPU work.
PINNED MEMORY

• The page-locked host memory.

• All CPU memory used by the deterministic applications should be pinned (cudaMallocHost, cudaHostAlloc). Tradeoff between pinned memory usage and determinism. Without Pinned memory:
  • Asynchronous DMA transfers can not be done due to copying of pageable memory to staging memory involved.
LOCAL MEMORY RESIZES

• Use CU_CTX_LMEM_RESIZE_TO_MAX to avoid local memory resizes during kernel launches which can result in dynamic allocation. Tradeoff between resource utilization and determinism.

• In the init phase, run all kernels in the program at least once. This will ensure that enough local memory for the highest local memory requiring kernel has been allocated.
  • All calls to cuCtxSetLimit() for CU_LIMIT_STACK_SIZE should be made in the init phase. Changing the stack size also results in the local memory reallocation.
UNIFIED MEMORY

• Avoid using CUDA unified memory (created using cudaMallocManaged or cuMemAllocManaged). On current generation of hardware, managed memory results in dynamic behavior and resource allocations/deallocations.

• Tradeoff between ease of programming and determinism.
Do not use new, delete, malloc and free calls in CUDA kernels. Deterministic applications should allocate memory in the init phase and free/delete at the deinit phase.

Tradeoff between resource utilization vs determinism and also ease of programming vs determinism.
REFERENCES

• CUDA - New Features and Beyond by Stephen Jones - GTC Europe 2018

• Image Sources: Google Images
CONTACT US

• Aayush Rajoria - arajoria@nvidia.com
• Ashok Kelur - akelur@nvidia.com