Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems

Ming Yang, Nathan Otterness, Tanya Amert, Joshua Bakita, James H. Anderson, F. Donelson Smith

All image sources and references are provided at the end.
Pitfalls for Real-Time GPU Usage

- Synchronization and blocking
- GPU concurrency and performance
- CUDA programming perils
## CUDA Programming Fundamentals

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i)</td>
<td>Allocate GPU memory</td>
<td>cudaMalloc(&amp;devicePtr, bufferSize);</td>
</tr>
<tr>
<td>(ii)</td>
<td>Copy data from CPU to GPU</td>
<td>cudaMemcpy(devicePtr, hostPtr, bufferSize);</td>
</tr>
<tr>
<td>(iii)</td>
<td>Launch the kernel <em>(kernel = code that runs on GPU)</em></td>
<td>computeResult&lt;&lt;&lt;numBlocks, threadsPerBlock&gt;&gt;&gt;(devicePtr);</td>
</tr>
<tr>
<td>(iv)</td>
<td>Copy results from GPU to CPU</td>
<td>cudaMemcpy(hostPtr, devicePtr, bufferSize);</td>
</tr>
<tr>
<td>(v)</td>
<td>Free GPU memory</td>
<td>cudaFree(devicePtr);</td>
</tr>
</tbody>
</table>
### CUDA Programming Fundamentals

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i)</td>
<td>Allocate GPU memory</td>
<td><code>cudaMalloc(&amp;devicePtr, bufferSize);</code></td>
</tr>
<tr>
<td>(ii)</td>
<td>Copy data from CPU to GPU</td>
<td><code>cudaMemcpy(devicePtr, hostPtr, bufferSize);</code></td>
</tr>
</tbody>
</table>
| (iii) | Launch the kernel  
(*kernel = code that runs on GPU*) | `computeResult<<<numBlocks, threadsPerBlock>>>(devicePtr);` |
| (iv) | Copy results from GPU to CPU | `cudaMemcpy(hostPtr, devicePtr, bufferSize);` |
| (v)  | Free GPU memory | `cudaFree(devicePtr);` |
### CUDA Programming Fundamentals

1. **Allocate GPU memory**
   - ```
cudaMalloc(&devicePtr, bufferSize);
```  
2. **Copy data from CPU to GPU**
   - ```
cudaMemcpy(devicePtr, hostPtr, bufferSize);
```  
3. **Launch the kernel**
   - ```
computeResult<<<numBlocks, threadsPerBlock>>>(devicePtr);
```  
4. **Copy results from GPU to CPU**
   - ```
cudaMemcpy(hostPtr, devicePtr, bufferSize);
```  
5. **Free GPU memory**
   - ```
cudaFree(devicePtr);
```
## CUDA Programming Fundamentals

1. **Allocate GPU memory**
   - Code: `cudaMalloc(&devicePtr, bufferSize);`

2. **Copy data from CPU to GPU**
   - Code: `cudaMemcpy(devicePtr, hostPtr, bufferSize);`

3. **Launch the kernel**
   - Code: `computeResult<<<numBlocks, threadsPerBlock>>>(devicePtr);`
   - **Note:** `kernel = code that runs on GPU`  

4. **Copy results from GPU to CPU**
   - Code: `cudaMemcpy(hostPtr, devicePtr, bufferSize);`

5. **Free GPU memory**
   - Code: `cudaFree(devicePtr);`
CUDA Programming Fundamentals

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i)</td>
<td>Allocate GPU memory</td>
<td><code>cudaMalloc(&amp;devicePtr, bufferSize);</code></td>
</tr>
<tr>
<td>(ii)</td>
<td>Copy data from CPU to GPU</td>
<td><code>cudaMemcpy(devicePtr, hostPtr, bufferSize);</code></td>
</tr>
<tr>
<td>(iii)</td>
<td>Launch the kernel (\textit{kernel} = code that runs on GPU)</td>
<td><code>computeResult&lt;&lt;&lt;numBlocks, threadsPerBlock&gt;&gt;&gt;(devicePtr);</code></td>
</tr>
<tr>
<td>(iv)</td>
<td>Copy results from GPU to CPU</td>
<td><code>cudaMemcpy(hostPtr, devicePtr, bufferSize);</code></td>
</tr>
<tr>
<td>(v)</td>
<td>Free GPU memory</td>
<td><code>cudaFree(devicePtr);</code></td>
</tr>
</tbody>
</table>
## CUDA Programming Fundamentals

1. **Allocate GPU memory**
   
   \[
   \text{cudaMalloc}(&\text{devicePtr}, \text{bufferSize});
   \]

2. **Copy data from CPU to GPU**
   
   \[
   \text{cudaMemcpy}(\text{devicePtr}, \text{hostPtr}, \text{bufferSize});
   \]

3. **Launch the kernel**
   
   \[
   \text{computeResult}<<<\text{numBlocks}, \text{threadsPerBlock}>>>(\text{devicePtr});
   \]

4. **Copy results from GPU to CPU**
   
   \[
   \text{cudaMemcpy}(\text{hostPtr}, \text{devicePtr}, \text{bufferSize});
   \]

5. **Free GPU memory**
   
   \[
   \text{cudaFree}(\text{devicePtr});
   \]
Pitfalls for Real-Time GPU Usage

- Synchronization and blocking
- GPU concurrency and performance
- CUDA programming perils
Explicit Synchronization

![Diagram showing Explicit Synchronization with different streams and time intervals.]
Explicit Synchronization

CPU threads ("tasks")
Explicit Synchronization

![Diagram showing explicit synchronization]

- **SM 1**
  - **K1**: 1
  - **K2**: 1
  - **K4**: 1

- **SM 0**
  - **K1**: 0
  - **K2**: 0
  - **K4**: 0

**Time (seconds)**: 0.2 to 3.0

**K1 starts** — **K1 completes**

**CPU Blocking**: Shows the period where a job is blocked due to CPU resource limitation.
Explicit Synchronization

Diagram showing the execution of streams on different SMs with 1024 and 256 threads, highlighting CPU blocking during certain phases.
Explicit Synchronization

Thread 3

1. Call `cudaDeviceSynchronize` (explicit synchronization).

2. Sleep for 0.2 seconds.

3. Launch kernel K3.
Explicit Synchronization

1. Thread 3 calls `cudaDeviceSynchronize` (explicit synchronization). *(a)*

2. Thread 3 sleeps for 0.2 seconds. *(c)*

3. Thread 3 launches kernel K3. *(d)*
Explicit Synchronization

1. Thread 3 calls `cudaDeviceSynchronize` (explicit synchronization). (a)
2. Thread 4 launches kernel K4. (b)
3. Thread 3 sleeps for 0.2 seconds. (c)
4. Thread 3 launches kernel K3. (d)
Explicit Synchronization

Pitfall 1. Explicit synchronization does not block future commands issued by other tasks.
Implicit Synchronization

CUDA toolkit 9.2.88 Programming Guide, Section 3.2.5.5.4, "Implicit Synchronization": Two commands from different streams cannot run concurrently [if separated by]:

1. A page-locked host memory allocation
2. A device memory allocation
3. A device memory set
4. A memory copy between two addresses to the same device memory
5. Any CUDA command to the NULL stream
Implicit Synchronization

→ Pitfall 2. Documented sources of implicit synchronization may not occur.

1. A page-locked host memory allocation
2. A device memory allocation
3. A device memory set
4. A memory copy between two addresses to the same device memory
5. Any CUDA command to the NULL stream
Implicit Synchronization

- Stream 1 (K1)
- Stream 2 (K2)
- Stream 3 (K3)
- Stream 4 (K4)

Time (seconds):
- 0.2
- 0.4
- 0.6
- 0.8
- 1.0
- 1.2
- 1.4
- 1.6
- 1.8
- 2.0
- 2.2
- 2.4
- 2.6
- 2.8

SM 1:
- K1: 1
- K2: 1
- K4: 1

SM 0:
- K1: 0
- K2: 0
- K4: 0

CPU Blocking
Implicit Synchronization

1. Thread 3 calls `cudaFree`. (a)
2. Thread 3 sleeps for 0.2 seconds. (c)
3. Thread 3 launches kernel K3. (d)
Implicit Synchronization

1. Thread 3 calls `cudaFree`. (a)

2. Thread 4 is blocked on the CPU when trying to launch kernel 4. (b)

3. Thread 4 finishes launching kernel K4, thread 3 sleeps for 0.2 seconds. (c)

4. Thread 3 launches kernel K3. (d)
Implicit Synchronization

➔ **Pitfall 3.** The CUDA documentation neglects to list some functions that cause implicit synchronization.

➔ **Pitfall 4.** Some CUDA API functions will block future, unrelated, CUDA tasks on the CPU.
Pitfalls for Real-Time GPU Usage

● Synchronization and blocking
  ○ Suggestion: use CUDA Multi-Process Service (MPS).

● GPU concurrency and performance

● CUDA programming perils
Pitfalls for Real-Time GPU Usage

- Synchronization and blocking
  - Suggestion: use CUDA Multi-Process Service (MPS).

- GPU concurrency and performance

<table>
<thead>
<tr>
<th></th>
<th>Multiple Process-based Tasks</th>
<th>Multiple Thread-based Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without MPS</td>
<td>MP</td>
<td>MT</td>
</tr>
<tr>
<td>With MPS</td>
<td>MP(MPS)</td>
<td>MT(MPS)</td>
</tr>
</tbody>
</table>

Nathan Otterness
Pitfalls for Real-Time GPU Usage

- Synchronization and blocking
  - Suggestion: use CUDA Multi-Process Service (MPS).
- GPU concurrency and performance

<table>
<thead>
<tr>
<th></th>
<th>Multiple Process-based Tasks</th>
<th>Multiple Thread-based Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without MPS</td>
<td>MP</td>
<td>MT</td>
</tr>
<tr>
<td>With MPS</td>
<td>MP(MPS)</td>
<td>MT(MPS)</td>
</tr>
</tbody>
</table>
70% of the time, a single Hough transform iteration completed in 12 ms or less.
This occurred when four concurrent instances were running in separate CPU threads.
The observed WCET under MT was over 4x the WCET under MP.
GPU Concurrency and Performance

Nathan Otterness
GPU Concurrency and Performance

The graph shows three different performance curves labeled as:
- **x4 MP**
- **x4 MP (MPS)**
- **x4 MT**

The x-axis represents time in milliseconds, ranging from 1 to 56. The y-axis on the left represents the percentage of tasks completed (% <= X), while the y-axis on the right represents density.

Nathan Otterness
Pitfall 5. The suggestion from NVIDIA’s documentation to exploit concurrency through user-defined streams may be of limited use for improving performance in thread-based tasks.
Pitfalls for Real-Time GPU Usage

● Synchronization and blocking
  ○ Suggestion: use CUDA Multi-Process Service (MPS).

● GPU concurrency and performance

● CUDA programming perils
Pitfalls for Real-Time GPU Usage

- Synchronization and blocking
  - Suggestion: use CUDA Multi-Process Service.
- GPU concurrency and performance
- CUDA programming perils
Pitfalls for Real-Time GPU Usage

- Synchronization and blocking
  - Suggestion: use CUDA Multi-Process Service (MPS).
- GPU concurrency and performance
- CUDA programming perils
Synchronous Defaults

if (!CheckCUDAError(
    cudaMemcpyAsync(
        state->device_block_smids,
        0, data_size)))) {
    return 0;
}
Synchronous Defaults

if (!CheckCUDAError(
    cudaMemcpyAsync(
        state->device_block_smids, 0, data_size))) {
    return 0;
}

• What about the CUDA docs saying that memset causes implicit synchronization?
Synchronous Defaults

if (!CheckCUDAError(
cudaMemsetAsync(
state->device_block_smids,
0, data_size))) {
    return 0;
}

• What about the CUDA docs saying that memset causes implicit synchronization?

• Didn't slide 22 say memset doesn't cause implicit synchronization?
Synchronous Defaults

if (!CheckCUDAError(
cudaMemsetAsync(
state->device_block_smids,
0, data_size))) {
return 0;
}

→ Pitfall 6. Async CUDA functions use the GPU-synchronous NULL stream by default.
Other Perils

➔ Pitfall 7. Observed CUDA behavior often diverges from what the documentation states or implies.

<table>
<thead>
<tr>
<th>Source</th>
<th>Blocks Other CPU Tasks</th>
<th>Implicit Sync. (Sec. 3.1.2)</th>
<th>Caller Must Wait for GPU</th>
<th>Documented Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>cudaDeviceSynchronize</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>cudaFree</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No (undoc.)</td>
</tr>
<tr>
<td>cudaFreeHost</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No (impl.)</td>
</tr>
<tr>
<td>cudaMalloc</td>
<td>?</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>cudaMallocHost</td>
<td>?</td>
<td>No</td>
<td>Yes</td>
<td>No (impl.)</td>
</tr>
<tr>
<td>cudaMemcpyAsync D-D</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>cudaMemcpyAsync D-H</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>cudaMemcpyAsync H-D</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>cudaMemcpyAsync (sync.)</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>cudaMemcpyAsync</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>cudaMemcpyAsync</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>cudaMemcpyAsync</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>cudaMemcpyAsync</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>cudaMemcpyAsync</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>cudaMemcpyAsync</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>cudaMemcpyAsync</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>cudaMemcpyAsync</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Nathan Ottersness
Other Perils

→ Pitfall 8. CUDA documentation can be contradictory.
Other Perils

➔ **Pitfall 8.** CUDA documentation can be contradictory.

CUDA Programming Guide, section 3.2.5.1:

*The following device operations are asynchronous with respect to the host: [...] Memory copies performed by functions that are suffixed with Async*

CUDA Runtime API Documentation, section 2:

*For transfers from device memory to pageable host memory, [cudaMemcpyAsync] will return only once the copy has completed.*
Other Perils

➔ **Pitfall 9.** What we learn about current black-box GPUs may not apply in the future.
Conclusion

- The GPU ecosystem needs clarity and openness!
- Avoid pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems
  - GPU synchronization, application performance, and problems with documentation
Thanks!
Questions?

Figure sources:
https://electrek.co/guides/tesla-vision/
https://www.researchgate.net/figure/Compute-unified-device-architecture-CUDA-threads-and-blocks-multidimensional_fig1_320806445?_sg=ziaY-qBKiKX4pliRq4yJSWZvDvdOidZ2aCRYnD1QVFBJDlx3MEO1l03c131e1It6pUr53qaS1L1w4Bt5fd8w