

Practical Course: GPU Programming in Computer Vision

CUDA Miscellaneous

Björn Häfner, Benedikt Löwenhauser, Thomas Möllenhoff

Technische Universität München
Department of Informatics
Computer Vision Group

Summer Semester 2017
September 11 - October 8



Outline

- 1 Atomics
- 2 Streams and Events
- 3 Multi-GPU Programming
- 4 Third-party Libraries



Outline

- 1** Atomics
- 2 Streams and Events
- 3 Multi-GPU Programming
- 4 Third-party Libraries

Communication Through Memory

■ Question:

```
1  __global__ void race()
2  {
3      __shared__ int my_shared_variable;
4      my_shared_variable = threadIdx.x;
5
6      // what is the value of my_shared_variable?
7  }
```



Communication Through Memory

- This is a race condition
- The result is undefined
- The order in which threads access the variable is undefined without explicit coordination
- Use atomic operations (e.g., `atomicAdd`) to enforce well-defined semantics

Atomics

- Use atomic operations to ensure exclusive access to a variable

```
1 // assume *p_result is initialized to 0
2 __global__ void sum(int *input, int *p_result)
3 {
4     atomicAdd(p_result, input[threadIdx.x]);
5
6     // after this kernel exits, the value of
7     // *p_result will be the sum of the inputs
8 }
```



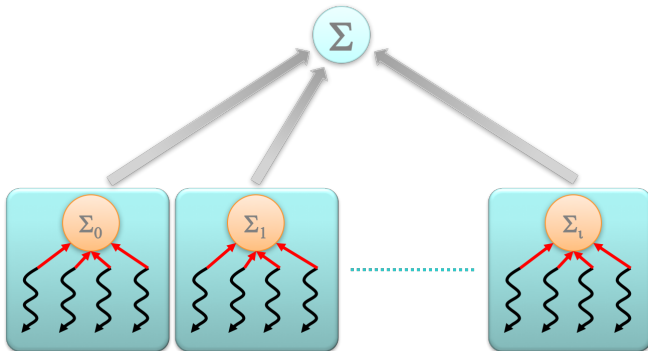
Atomics Imply Serialization

- Atomic operations are costly!
- They imply serialized access to a variable
- \Rightarrow use them only if there is no other better way to achieve your task

Atomics Imply Serialization

```
1  __global__ void sum(int *input, int *p_result)
2  {
3      atomicAdd(p_result, input[threadIdx.x]);
4  }
5
6  // how many threads will contend
7  // for exclusive access to p_result?
8  sum <<<10,128>>> (input,p_result);
```


Atomsics: Hierarchical Summation



Divide & Conquer:

- `__shared__` partial sums: `atomicAdd` per thread
- global total sum: `atomicAdd` per block

Atomics: Hierarchical Summation

```
1  __global__ void sum(int *input, int *result)
2  {
3      __shared__ int partial_sum;
4
5      // thread 0 is responsible for initializing partial_sum
6      if(threadIdx.x == 0) partial_sum = 0;
7      __syncthreads();
8
9      // each thread updates the partial sum
10     atomicAdd(&partial_sum, input[threadIdx.x]);
11     __syncthreads();
12
13     // thread 0 updates the total sum
14     if(threadIdx.x == 0) atomicAdd(result, partial_sum);
15 }
```



Advice: Shared Memory and Atomics

- Always use barriers such as `__syncthreads()`; to wait until `__shared__` data is ready
- Prefer barriers to atomics when data access patterns are regular or predictable
- Prefer atomics to barriers when data access patterns are sparse or unpredictable
- Atomics to `__shared__` variables are much faster than atomics to global variables



Outline

- 1 Atomics
- 2 Streams and Events
- 3 Multi-GPU Programming
- 4 Third-party Libraries

Disclaimer

- I haven't tried out most of what will follow myself
- Proceed with caution 😊
- Check out the samples in the SDK and look up documentation

Streams

- Concurrency is handled through **streams**
 - overlap kernel execution with another kernel execution
 - overlap kernel execution with a memcpy
 - overlap memcpy with another memcpy
 - wait for certain kernels, but not for others
- **Stream** = sequence of commands **executed in order**
 - different **streams** may execute concurrently, but not guaranteed
 - depends on hardware and the kind of operations executed in the streams
 - **default stream** is 0: if no stream specified
 - so everything without an explicitly specified stream executes in order

Streams

```
1  cudaStream_t stream1; cudaStream_t stream2;
2  cudaStreamCreate(&stream1); cudaStreamCreate(&stream2);
3  float *h_ptr; cudaMallocHost(&h_ptr, size);
4
5  // (potentially) overlapping execution
6  cudaMemcpyAsync(h_ptr, d_ptr, size, dir, stream1);
7  kernel <<<grid,block,0,stream2>>> (...);
8
9  // check whether memcpy has finished
10 cudaMemcpy_t res = cudaStreamQuery(stream1);
11 if (res==cudaSuccess) { ... }
12
13 // or: wait for completion:
14 cudaStreamSynchronize(stream1); // will only wait for the memcpy
15 cudaStreamSynchronize(stream2); // will only wait for the kernel
16
17 cudaStreamDestroy(&stream1); cudaStreamDestroy(&stream2);
```

Events

- Monitor device's progress
- Asynchronously record events at any point in the program
- Event recorded when all commands in stream completed
 - measure elapsed time for CUDA calls (clock cycle precision)
 - query the status of an asynchronous CUDA call
 - block CPU until CUDA calls prior to the event are completed

```
1  cudaEvent_t start; cudaEvent_t stop;
2  cudaEventCreate(&start); cudaEventCreate(&stop);
3  cudaEventRecord(start,0); // default stream
4  kernel <<<grid,block>>> (...);
5  cudaEventRecord(stop,0); // default stream
6  cudaEventSynchronize(stop); // block until "stop" recorded
7  float t; cudaEventElapsedTime(&t, start, stop);
8  cudaEventDestroy(start); cudaEventDestroy(end);
```




Outline

- 1 Atomics
- 2 Streams and Events
- 3 Multi-GPU Programming**
- 4 Third-party Libraries

Multi-GPU Programming

- There may be more than one GPU installed
- Host can query and select GPU devices
 - `cudaGetDeviceCount(int *count);`
 - `cudaSetDevice(int device);`
 - `cudaGetDevice(int *current_device);`
 - `cudaGetDeviceProperties(cudaDeviceProp *prop, int device);`
- Multi-GPU setting: device 0 is used by default

Multi-GPU Programming

- `cudaSetDevice(...)` can be called at any time
- Everything happens on the current device:
 - `cudaMalloc(...)` allocates on the cur. dev. only
 - `cudaFree(...)` frees memory of cur. dev.
 - Kernels execute only on the cur. dev.
 - `cudaDeviceSynchronize()` waits only for cur. dev.
- GPUs are independent: kernels run in parallel

```
1  cudaSetDevice(0); mykernel1 <<<grid1,block1>>> (d0_a, n0_a);  
2  cudaSetDevice(1); mykernel2 <<<grid2,block2>>> (d1_a, n1_a);
```



Multi-GPU Programming

- Data exchange between GPUs
`cudaMemcpyPeer(ptr_to, dev_to, ptr_from, dev_from, size);`
- From CC \geq 2.0: Direct access between GPUs
- Kernel on device x can read memory on device y
 - memcopies are done automatically
 - utilizes unified virtual addressing
 - must be explicitly enabled:
`cudaDeviceEnablePeerAccess(dev_peer, 0);`
 - enables current device to access memory of dev_peer

Outline

- 1 Atomics
- 2 Streams and Events
- 3 Multi-GPU Programming
- 4 Third-party Libraries**

Linear Algebra and Math Libraries

**cuBLAS**

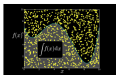
GPU-accelerated standard BLAS library

**CUDA Math Library**

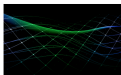
GPU-accelerated standard mathematical function library

**cuSPARSE**

GPU-accelerated BLAS for sparse matrices

**cuRAND**

GPU-accelerated random number generation (RNG)

**cuSOLVER**

Dense and sparse direct solvers for Computer Vision, CFD, Computational Chemistry, and Linear Optimization applications

**AmgX**

GPU accelerated linear solvers for simulations and implicit unstructured methods

Image Processing, Algorithms and Deep Learning



cuFFT

GPU-accelerated library for Fast Fourier Transforms



NVIDIA Performance Primitives

GPU-accelerated library for image and signal processing



NVIDIA Codec SDK

High-performance APIs and tools for hardware accelerated video encode and decode



NCCL

Collective Communications Library for scaling apps across multiple GPUs and nodes



nvGRAPH

GPU-accelerated library for graph analytics



Thrust

GPU-accelerated library of parallel algorithms and data structures



GPU-accelerated library of primitives for deep neural networks



TensorRT

GPU-accelerated neural network inference library for building deep learning applications



DeepStream SDK

Advanced GPU-accelerated video inference library

... and much more!



GPU-accelerated open-source library for computer vision, image processing and machine learning, now supporting real-time operation



Open-source multi-media framework with a library of plugins for audio and video processing



GPU-accelerated open source library for matrix, signal, and image processing



GPU-accelerated linear algebra routines for heterogeneous architectures, by Magma



GPU-accelerated open-source Fortran library with functions for math, signal and image processing, statistics, by RogueWave



Library for graph-processing designed specifically for the GPU



GPU-accelerated functions for sparse direct solvers, included in SuiteSparse linear algebra package authored by Prof.



GPU-accelerated linear algebra library by EM Photonics



GPU-accelerated linear algebra (LA) routines for the R platform for statistical computing supporting heterogeneous



GPU-accelerated computational geometry engine for advanced GIS, EDA, computer vision, and motion planning, by Flosters



GPU-accelerated library for sparse iterative methods by Paralution



Real-time visual simulation of oceans, water bodies in games, simulation, and training applications, by Trion

<https://developer.nvidia.com/gpu-accelerated-libraries>

Further Reading

CUDA Programming Guide (linked on course page)

- Appendix B.12 (atomics)
- Chapter 3, section 3.2.5 (streams & events)
- Appendix J (unified memory programming)