GPU Programming in Computer Vision

Evgeny Strekalovskiy, Maria Klodt
Jan Stühmer, Mohamed Souiai

Miscellaneous

Technical University Munich, Computer Vision Group
Winter Semester 2013/2014, March 3 – April 4

Outline

- **Atomics**
- **CUDA Streams and Events**
- **Multi-GPU Programming**

- **See the Programming Guide for more details**

ATOMICS

Communication Through Memory

- **Question:**

```
__global__ void race()
{
  __shared__ int my_shared_variable;
  my_shared_variable = threadIdx.x;

  // what is the value of my_shared_variable?
}
```

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**

```
// assume *p_result is initialized to 0
__global__ void sum(int *input, int *p_result)
{
  atomicAdd(p_result, input[threadIdx.x]);

  // after this kernel exits, the value of
  // *p_result will be the sum of the inputs
}
```

Atomics Imply Serialization

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

```
__global__ void sum(int *input, int *p_result)
{
  atomicAdd(p_result, input[threadIdx.x]);
}


// how many threads will contend
// for exclusive access to p_result?
sum <<<10,128>>> (input,p_result);
```

Atomics: Hierarchical Summation
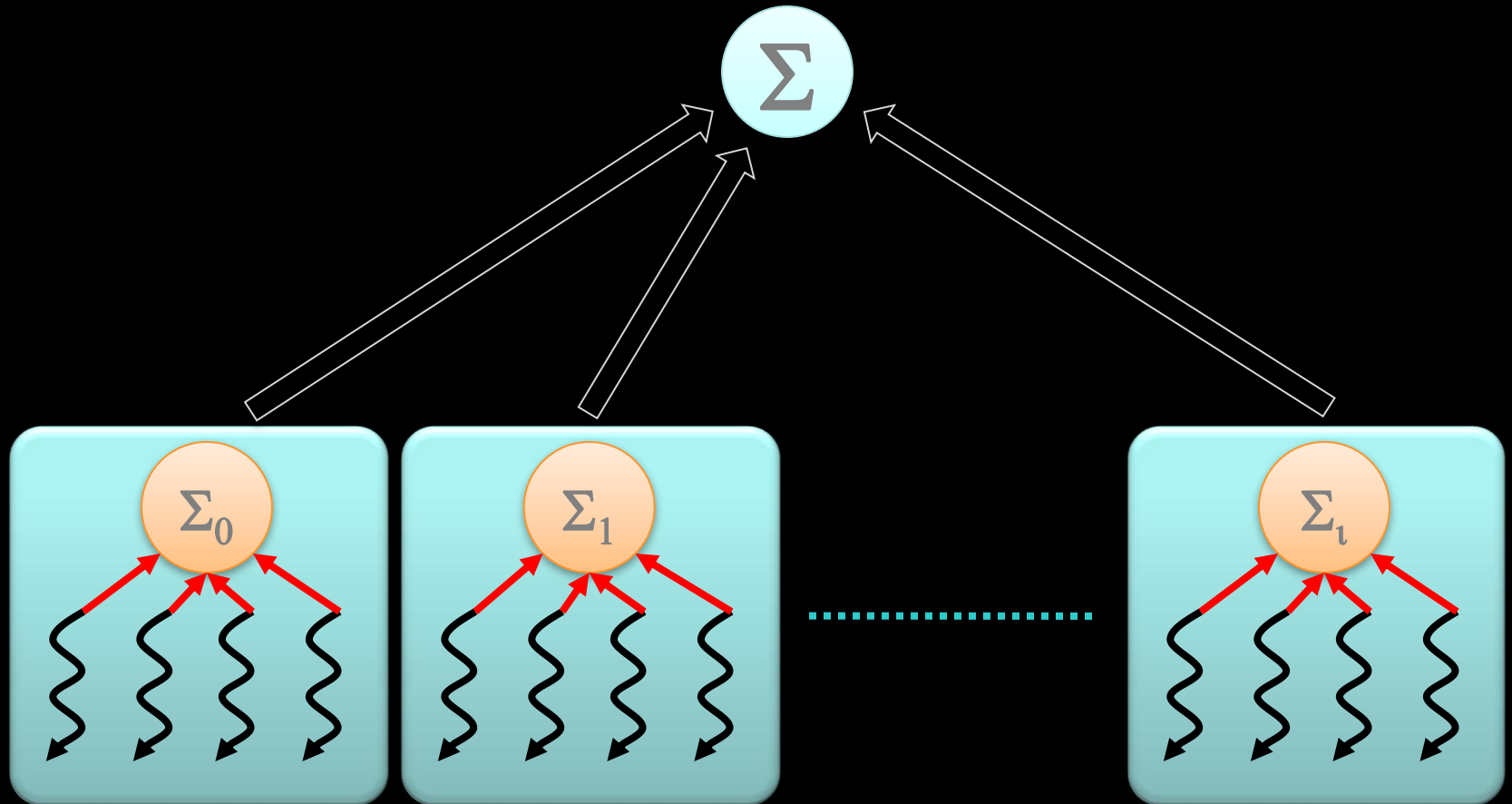
- **Divide & Conquer**
  - **__shared__ partial sums: `atomicAdd` per thread**
  - **global total sum: `atomicAdd` per block**

Atomics: Hierarchical Summation

```
__global__ void sum(int *input, int *result)
{
  __shared__ int partial_sum;

  // thread 0 is responsible for initializing partial_sum
  if(threadIdx.x == 0) partial_sum = 0;
  __syncthreads();

  // each thread updates the partial sum
  atomicAdd(&partial_sum, input[threadIdx.x]);
  __syncthreads();

  // thread 0 updates the total sum
  if(threadIdx.x == 0) atomicAdd(result, partial_sum);
}
```

CUDA STREAMS AND EVENTS

CUDA 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 certains kernels, but not for others

- **Stream = sequence of commands executed in order**
  - **different streams may execute cuncurrently, 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
  - possible: callbacks, relative priorities

CUDA Streams

```
cudaStream_t stream1; cudaStream_t stream2;
cudaStreamCreate(&stream1); cudaStreamCreate(&stream2);
float *h_ptr;  cudaMallocHost(&h_ptr, size);


cudaMemcpyAsync(h_ptr, d_ptr, size, dir, stream1);
kernel <<<grid,block,0,stream2>>> (...);
```

(potentially) overlaping execution

```
// check whether memcpy has finished
cudaError_t res = cudaStreamQuery(stream1);
if (res==cudaSuccess) { ... }

// or: wait for completion:
cudaStreamSynchronize(stream1);  // will only wait for the memcpy
cudaStreamSynchronize(stream2);  // will only wait for the kernel

cudaStreamDestroy(&stream1); cudaStreamDestroy(&stream2);
```

CUDA 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

```
cudaEvent_t start; cudaEvent_t stop;
cudaEventCreate(&start); cudaEventCreate(&stop);
cudaEventRecord(start,0);           // default stream
kernel <<<grid,block>>> (...);
cudaEventRecord(stop,0);            // default stream
cudaEventSynchronize(stop);        // block until "stop" recorded
float t; cudaEventElapsedTime(&t, start, stop);
cudaEventDestroy(start); cudaEventDestroy(end);
```

MULTI-GPU PROGRAMMING

Multi-GPU Programming

- **There may be more than one GPU installed**

- **CPU 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: Current Device

- **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**

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

Multi-GPU: Data Exchange

- **Data exchange between GPUs**
  - `cudaMemcpyPeer(ptr_to, dev_to,`
                   `ptr_from, dev_from, size);`

- **From CC>=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**

GPU Programming in Computer Vision

That's it!

Have fun
parallelizing your
applications
with CUDA!

And read the CUDA Programming Guide, really