

GPU Programming in Computer Vision

**Thomas Möllenhoff, Robert Maier,
Lingni Ma, Caner Hazırbaş**

Miscellaneous

**Technical University Munich, Computer Vision Group
Winter Semester 2015/2016, March 15 – April 15**

Outline

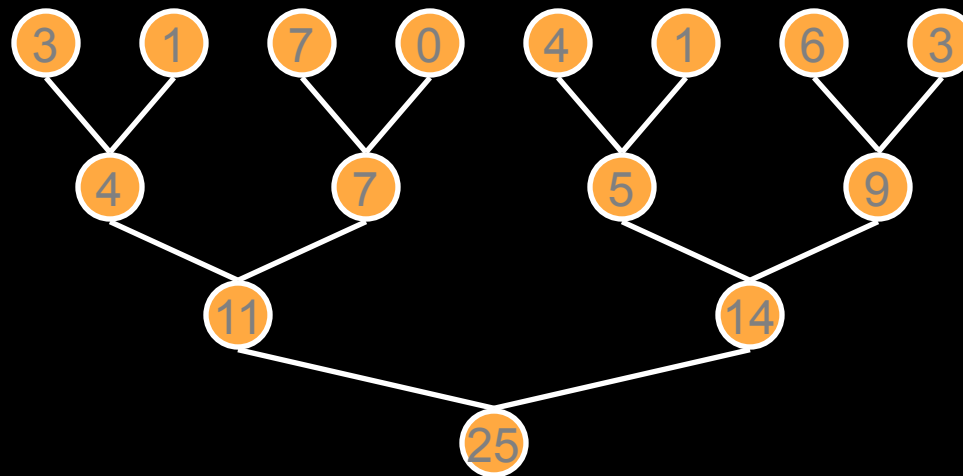
- **Parallel Reduction**
 - **Atomics**
 - **CUDA Streams and Events**
 - **Multi-GPU Programming**
-
- **See the Programming Guide for more details**

PARALLEL REDUCTION

Reduction

- **Reduce** vector to a single value
 - Via an associative operator (+, *, min/max, AND/OR, ...)
 - CPU: sequential implementation

```
for(int i = 0, i < n, ++i) ...
```
 - GPU: “tree”-based implementation

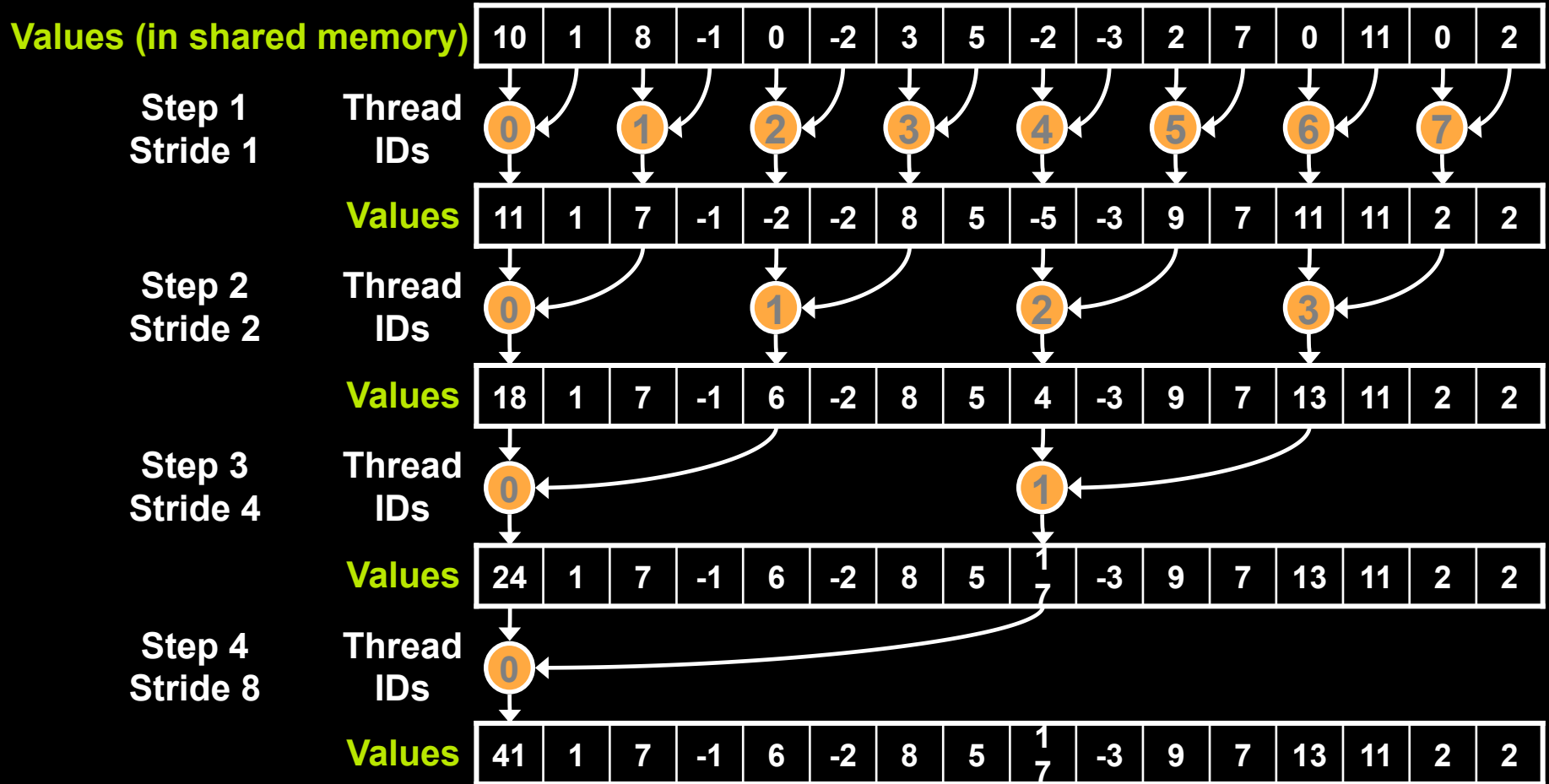


Serial Reduction

```
// reduction via serial iteration
float sum(float *data, int n)
{
    float result = 0;
    for(int i = 0; i < n; ++i)
    {
        result += data[i];
    }

    return result;
}
```

Parallel Reduction – Interleaved



Parallel Reduction – Contiguous

Values (in shared memory)

10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2
----	---	---	----	---	----	---	---	----	----	---	---	---	----	---	---

Step 1
Stride 8

Thread IDs



Values

8	-2	10	6	0	9	3	7	-2	-3	2	7	0	11	0	2
---	----	----	---	---	---	---	---	----	----	---	---	---	----	---	---

Step 2
Stride 4

Thread IDs



Values

8	7	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2
---	---	----	----	---	---	---	---	----	----	---	---	---	----	---	---

Step 3
Stride 2

Thread IDs



Values

21	20	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2
----	----	----	----	---	---	---	---	----	----	---	---	---	----	---	---

Step 4
Stride 1

Thread IDs



Values

41	20	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2
----	----	----	----	---	---	---	---	----	----	---	---	---	----	---	---

CUDA Reduction

```
__global__ void block_sum(float *input,
                          float *results,
                          size_t n)
{
    extern __shared__ float sdata[];
    int i = ..., int tx = threadIdx.x;

    // load input into __shared__ memory
    float x = 0;
    if(i < n)
        x = input[i];
    sdata[tx] = x;
    __syncthreads();
}
```


CUDA Reduction

```
// block-wide reduction in __shared__ mem
for(int offset = blockDim.x / 2;
    offset > 0;
    offset >>= 1)
{
    if(tx < offset)
    {
        // add a partial sum upstream to our own
        sdata[tx] += sdata[tx + offset];
    }
    __syncthreads();
}
```

CUDA Reduction

```
// finally, thread 0 writes the result
if(threadIdx.x == 0)
{
    // note that the result is per-block
    // not per-thread
    results[blockIdx.x] = sdata[0];
}
}
```

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
}
```

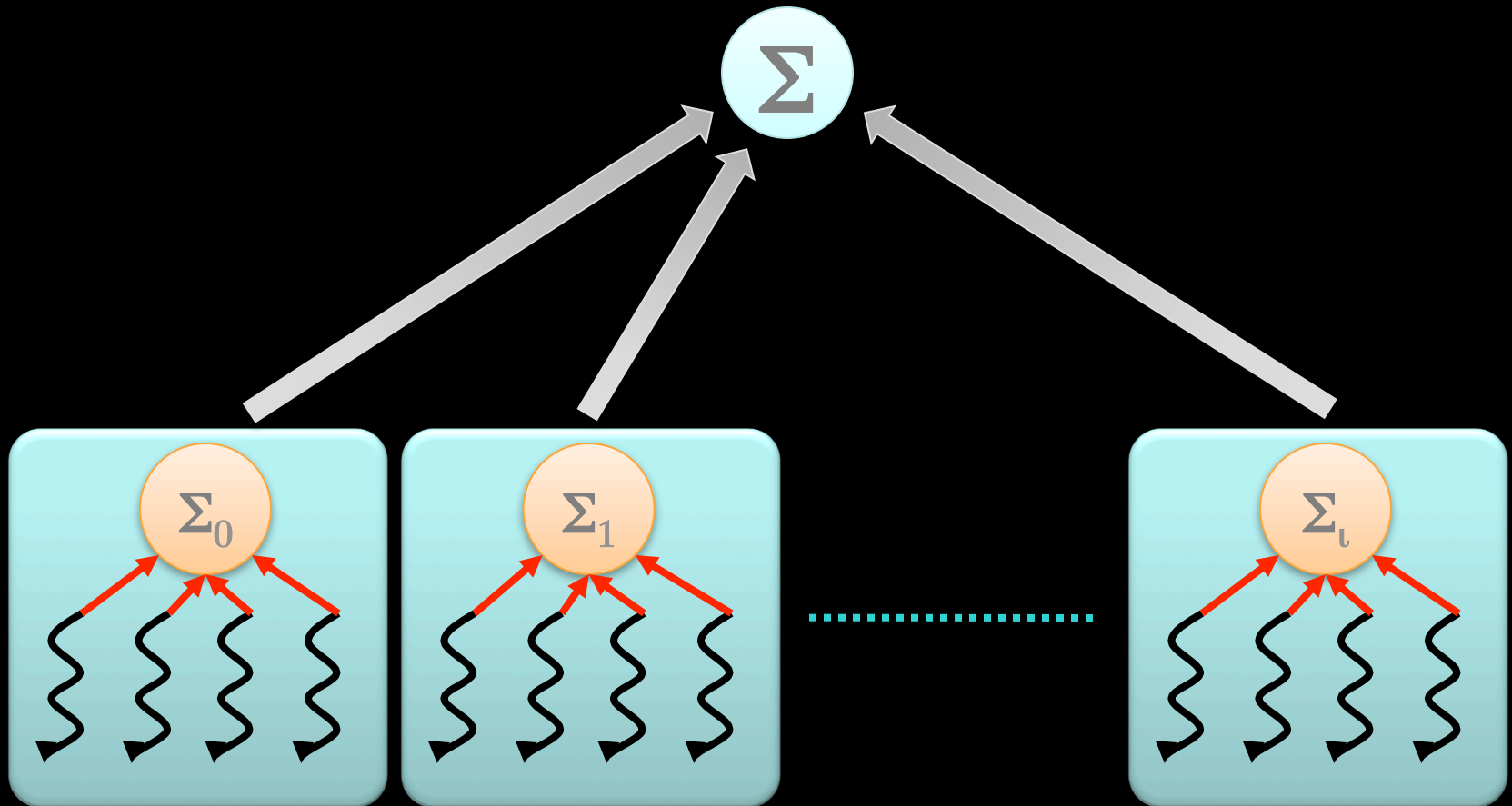
Atomics Imply Serialization

- 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);
}
```

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

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 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**
 - **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)
overlapping
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 \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

CUDA Libraries

- **Don't reinvent the wheel!**
 - **cuFFT, cuBLAS, cuSPARSE/cusp, thrust, ...**
 - <https://developer.nvidia.com/gpu-accelerated-libraries>
- **Many languages other than C/C++ offer CUDA support (python, MATLAB, ...)**

GPU Programming in Computer Vision

That's it!

Have fun
parallelizing your
applications
with CUDA!