/INFOMOV/

Optimization & Vectorization

J. Bikker - Sep-Nov 2016 - Lecture 12: “GPGPU (3)”

Welcome!

Today’s Agenda:

- Introduction
- The Prefix Sum
- Parallel Sorting
- Stream Filtering
- Optimizing GPU code

Introduction

Beyond “Many Independent Threads”

Many algorithms do not lend themselves to GPGPU, at least not at first sight:

- Divide and conquer algorithms
  - Sorting
- Anything with an unpredictable number of iterations
  - Walking a linked list or a tree
  - Ray tracing
- Anything that needs to emit data in a compacted array
  - Run-length encoding
  - Duplicate removal
- Anything that requires inter-thread synchronization
  - Hash table
  - Linked list

In fact, lock-free implementations of linked lists and hash tables exist and can be used in CUDA, see e.g.:

Misra & Chaudhuri, 2012, Performance Evaluation of Concurrent Lock-free Data Structures on GPUs.

Note that the possibility of using linked lists on the GPU does not automatically justify its use.

Introduction

Beyond “Many Independent Threads”

Many algorithms do not lend themselves to GPGPU.

In many cases, we have to design entirely new algorithms.

In some cases, we can use two important building blocks:

- Sort
- Prefix sum

Today’s Agenda:

- Introduction
- The Prefix Sum
- Parallel Sorting
- Stream Filtering
- Optimizing GPU code

Prefix Sum

Prefix Sum

The prefix sum (or cumulative sum) of a sequence of numbers is a second sequence of numbers consisting of the running totals of the input sequence:

Input: x_0, x_1, x_2
Output: $x_0, x_0 + x_1, x_0 + x_1 + x_2$ *(inclusive)* or $0, x_0, x_0 + x_1$ *(exclusive).*

Example:

input	1	2	2	1	4	3
inclusive	1	3	5	6	10	13
exclusive	0	1	3	5	6	10

Here, addition is used; more generally we can use an arbitrary binary associative operator.

Prefix Sum

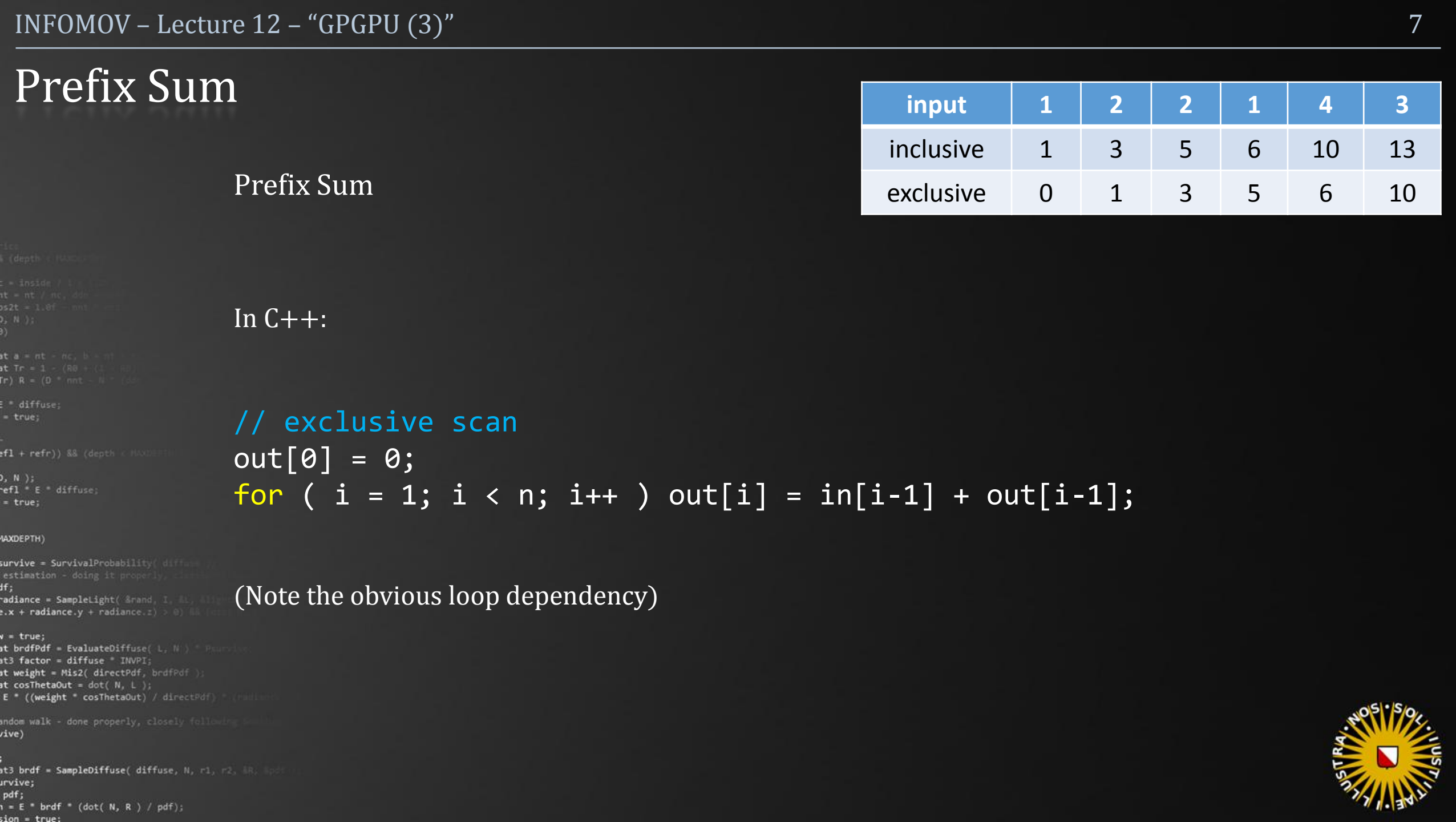
input	1	2	2	1	4	3
inclusive	1	3	5	6	10	13
exclusive	0	1	3	5	6	10

Prefix Sum

In C++:

```
// exclusive scan
out[0] = 0;
for ( i = 1; i < n; i++ ) out[i] = in[i-1] + out[i-1];
```

(Note the obvious loop dependency)

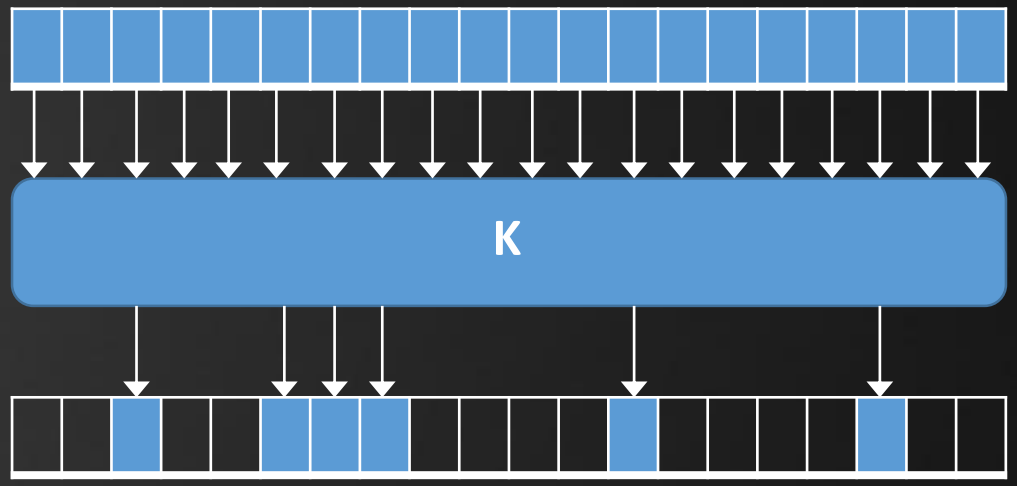


Prefix Sum

Prefix Sum

The prefix sum is used for *compaction*.

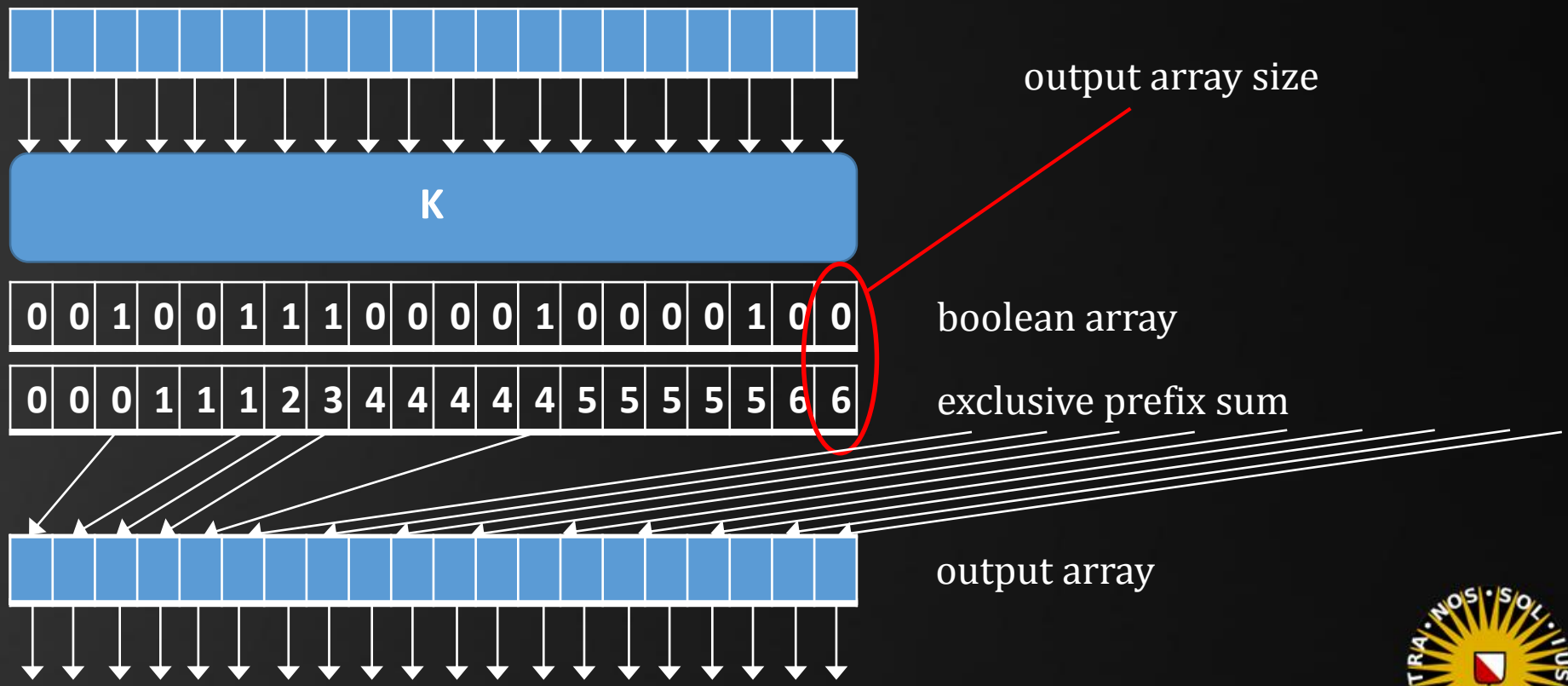
Given: kernel *K* which may or may not produce output for further processing.



Prefix Sum

Prefix Sum - Compaction

Given: kernel K which may or may not produce output for further processing.



Prefix Sum

Prefix Sum

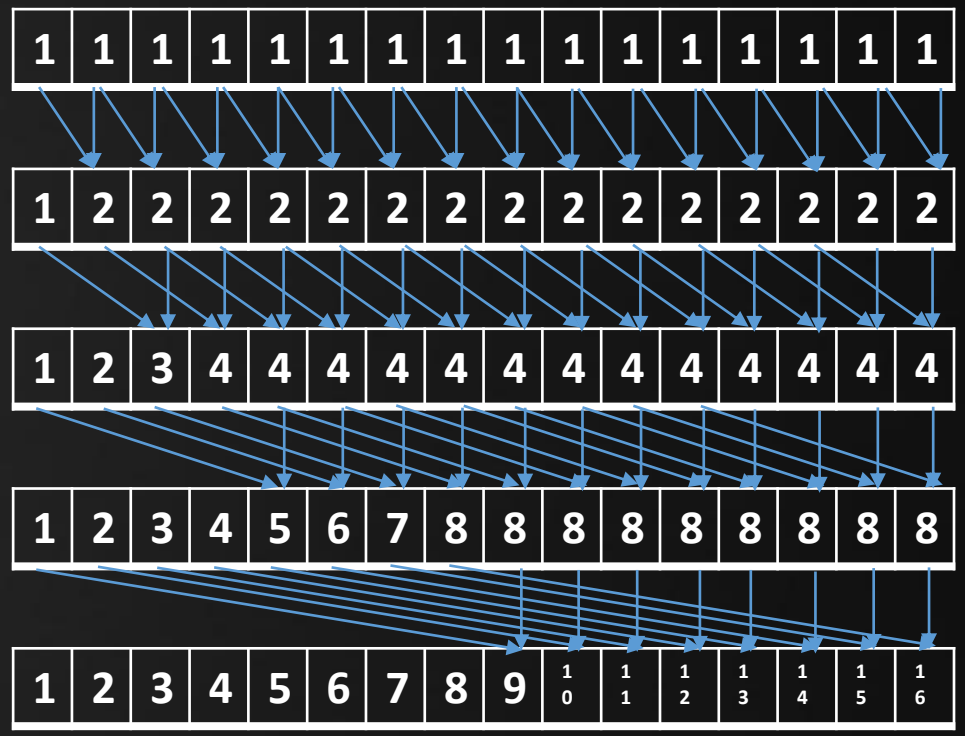
```
out[0] = 0;
for ( i = 1; i < n; i++ ) out[i] = in[i-1] + out[i-1];
```

In parallel:

```
for ( d = 1; d <= log2n; d++ )
   for all k in parallel do
      if k >= 2^(d-1)
         x[k] += x[k – 2^(d-1)]
```

For each pass:

- Each thread in the warp reads data
- Each thread in the warp sums 2 input elements
- Each thread in the warp writes data.



n = 16

Prefix Sum

Prefix Sum

```
out[0] = 0;
for ( i = 1; i < n; i++ ) out[i] = in[i-1] + out[i-1];
```

In parallel:

```
for ( d = 1; d <= log2n; d++ )
   for all k in parallel do
      if k >= 2^(d-1)
         x[k] += x[k – 2^(d-1)]
```

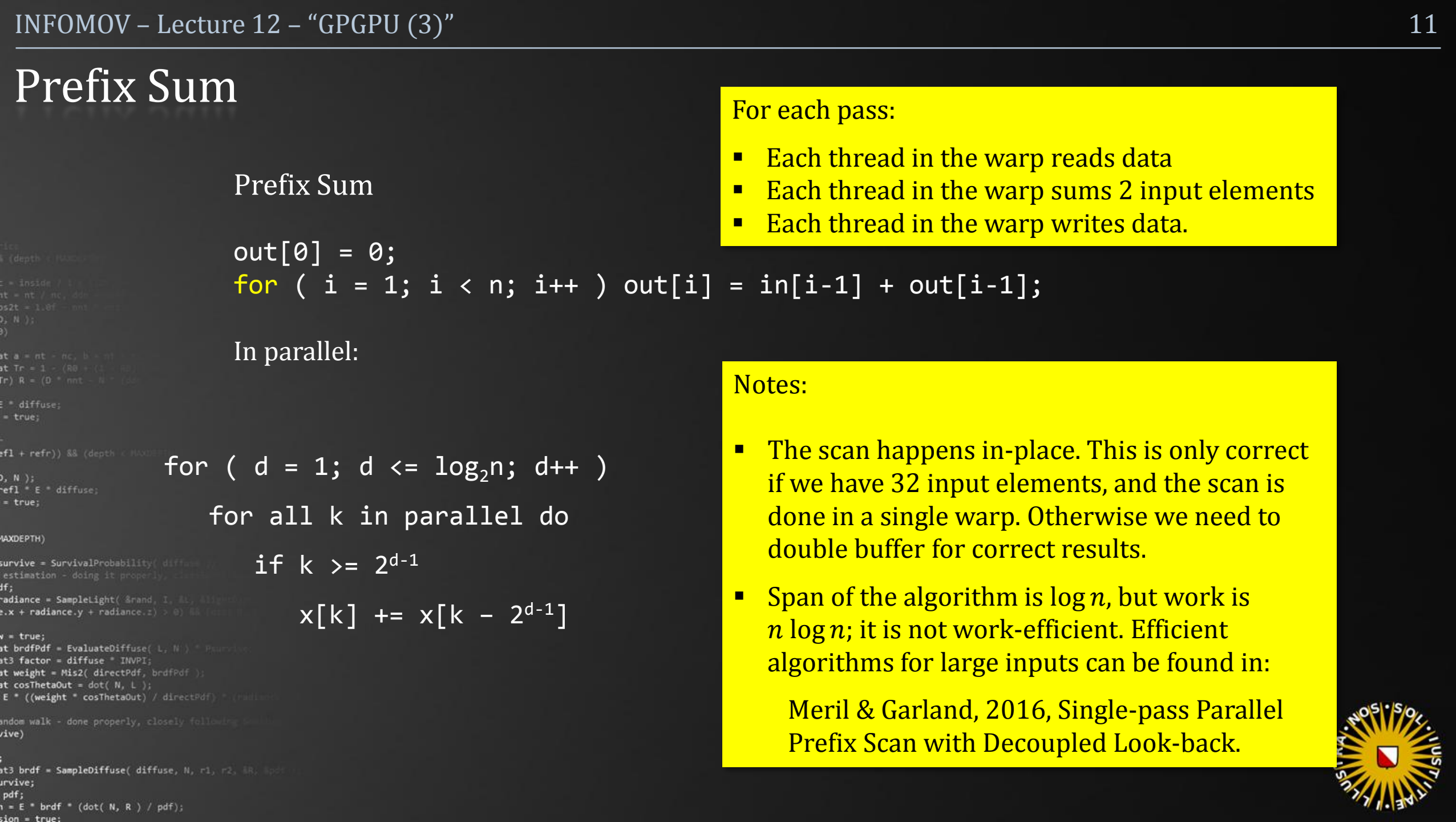
For each pass:

- Each thread in the warp reads data
- Each thread in the warp sums 2 input elements
- Each thread in the warp writes data.

Notes:

- The scan happens in-place. This is only correct if we have 32 input elements, and the scan is done in a single warp. Otherwise we need to double buffer for correct results.
- Span of the algorithm is $\log n$, but work is $n \log n$; it is not work-efficient. Efficient algorithms for large inputs can be found in:

  Meril & Garland, 2016, Single-pass Parallel Prefix Scan with Decoupled Look-back.



Prefix Sum

Prefix Sum

```
out[0] = 0;
for ( i = 1; i < n; i++ ) out[i] = in[i-1] + out[i-1];
```

In OpenCL:

```
int scan_exclusive( __local int* input, int lane )
{
    if (lane > 0 ) input[lane] = input[lane - 1] + input[lane];
    if (lane > 1 ) input[lane] = input[lane - 2] + input[lane];
    if (lane > 3 ) input[lane] = input[lane - 4] + input[lane];
    if (lane > 7 ) input[lane] = input[lane - 8] + input[lane];
    if (lane > 15) input[lane] = input[lane - 16] + input[lane];
    return (lane > 0) ? input[lane - 1] : 0;
}
```

Prefix Sum

Prefix Sum

You can find an implementation of the prefix sum for arbitrarily-sized arrays in the OpenCL template:

```
cl_int Buffer::ParallelScan()
```

This replaces the contents of a buffer with the prefix sum of the same buffer.

Today’s Agenda:

- Introduction
- The Prefix Sum
- Parallel Sorting
- Stream Filtering
- Optimizing GPU code

Sorting

GPU Sorting

Observation:

- We frequently need sorting in our algorithms.

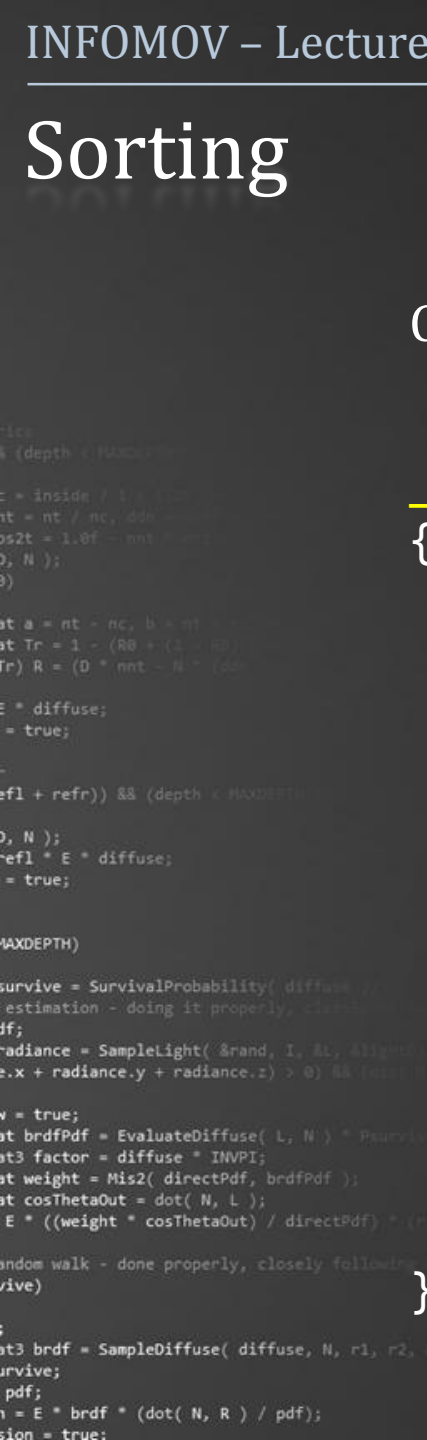
But:

- Most sorting algorithms are divide and conquer algorithms.

Sorting

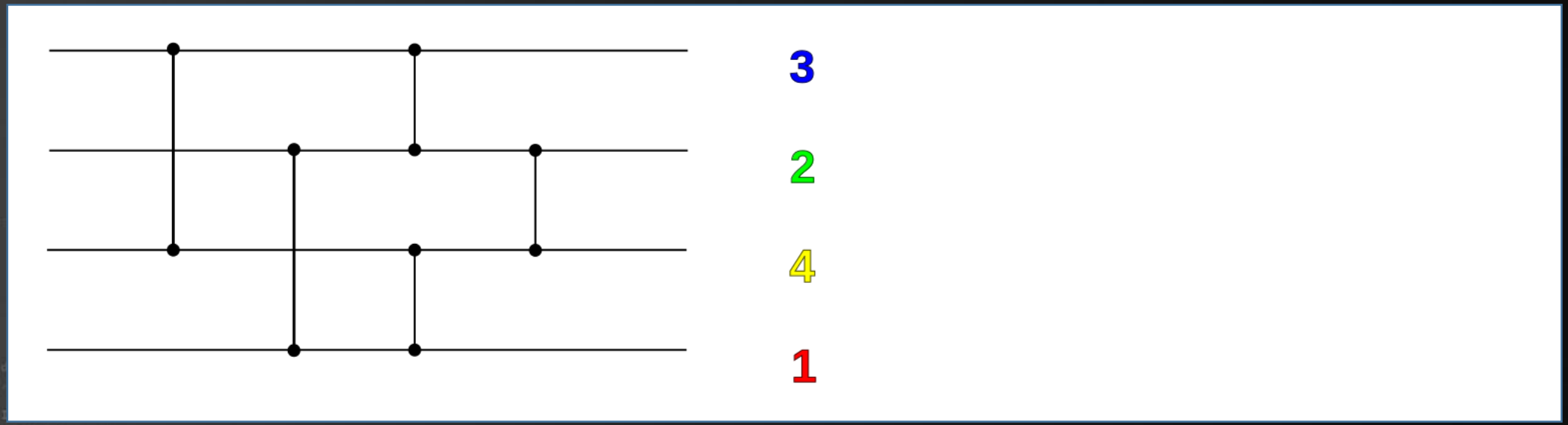
GPU Sorting: Selection Sort

```
__kernel void Sort( __global int* in, __global int* out )
{
  int i = get_global_id( 0 );
  int n = get_global_size( 0 );
  int iKey = in[i];
  // compute position of in[i] in output
  int pos = 0;
  for( int j = 0; j < n; j++ )
  {
    int jKey = in[j]; // broadcasted
    bool smaller = (jKey < iKey) || (jKey == iKey && j < i);
    pos += (smaller) ? 1 : 0;
  }
  out[pos] = iKey;
}
```



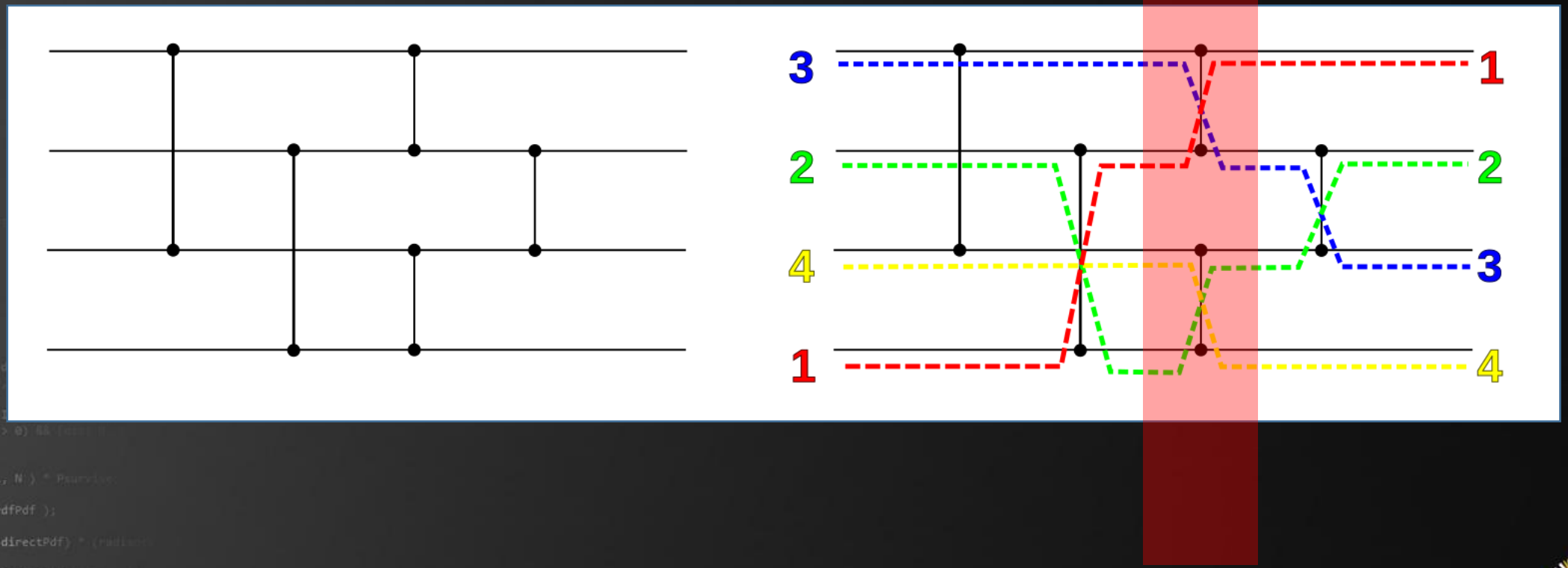
Sorting

GPU Sorting



Sorting

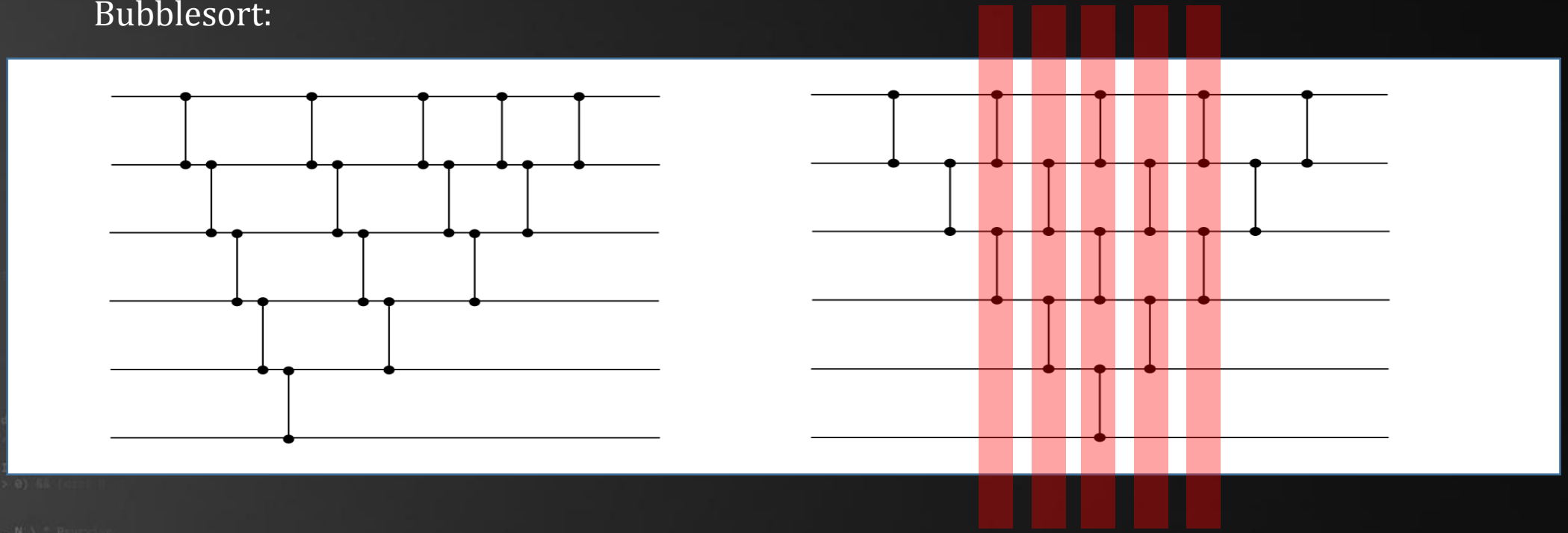
GPU Sorting



Sorting

GPU Sorting

Bubblesort:



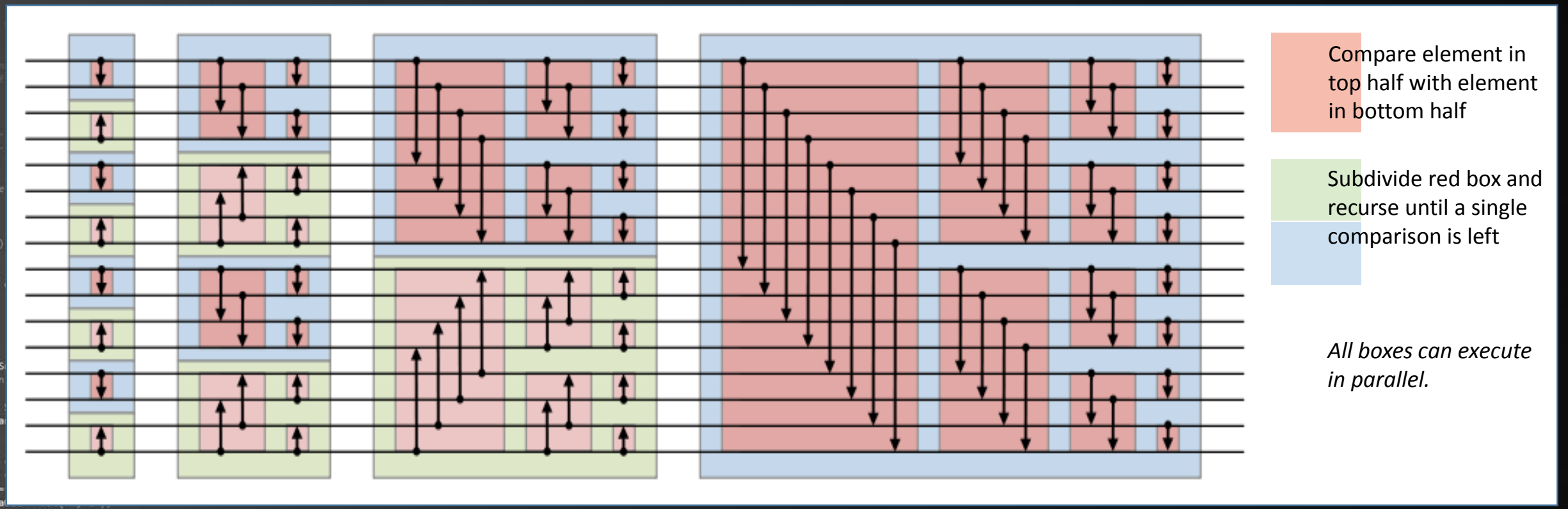
Size: number of comparisons (in this case: 5 + 4 + 3 + 2 + 1 = 15)
Depth: number of sequential steps (in this case: 9)

Sorting

GPU Sorting

Bitonic sort*,**:

- Work: $n \log(n)^2$
- Span: $\log(n)^2$



*: Batcher, ‘68, Sorting Networks and their Applications.
**: Bitonic Sorting Network for n Not a Power of 2;
http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/oddn.htm

Sorting

GPU Sorting

You can find an implementation of the bitonic sort in the OpenCL template:

```
cl_int Buffer::ParallelSort()
```

This replaces the contents of a buffer with the sorted values.

Sorting

Take-away:

GPGPU requires massive parallelism. Algorithms that do not exhibit this need to be replaced.
The parallel scan is an important ingredient that serves as a building block for larger algorithms, or between kernels.

Today’s Agenda:

- Introduction
- The Prefix Sum
- Parallel Sorting
- Stream Filtering
- Optimizing GPU code

Compaction

Stream Filtering

```
for ( int i = 0; i < items; i++ )
{
   // do something elaborate, ‘items’ can be 0..10
}

void ComplexTask( int taskID )
{
   // do generic work
   ...
   if (condition == true) // true 50% of the time
   {
      // do additional work
   }
}
```

Compaction

Stream Filtering

```
bool needsAdditionalWork[…];
void ComplexTaskPart1( int taskID )
{
   // do generic work
   ...
   if (condition == true) // true 50% of the time
   {
      // do additional work
      needsAdditionalWork[taskID] = true;
   }
}

void ComplexTaskPart2( int taskID )
{
   if (needsAdditionalWork[taskID])
   {
      ...
   }
}
```

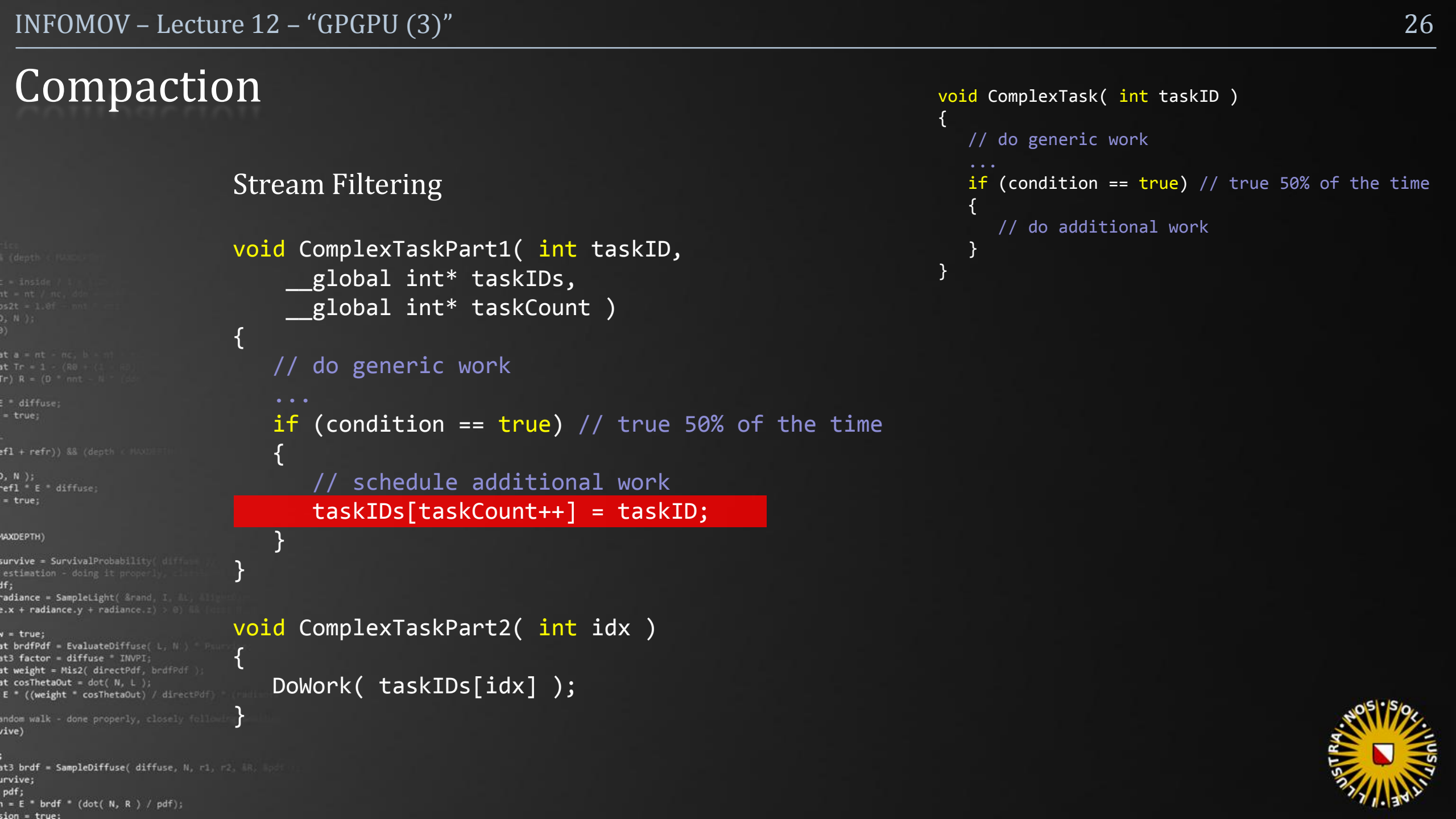
```
void ComplexTask( int taskID )
{
   // do generic work
   ...
   if (condition == true) // true 50% of the time
   {
      // do additional work
   }
}
```

Compaction

Stream Filtering

```
void ComplexTaskPart1( int taskID,
    __global int* taskIDs,
    __global int* taskCount )
{
   // do generic work
   ...
   if (condition == true) // true 50% of the time
   {
      // schedule additional work
      taskIDs[taskCount++] = taskID;
   }
}

void ComplexTaskPart2( int idx )
{
   DoWork( taskIDs[idx] );
}
```

```
void ComplexTask( int taskID )
{
   // do generic work
   ...
   if (condition == true) // true 50% of the time
   {
      // do additional work
   }
}
```



Compaction

Stream Filtering

```
void ComplexTaskPart1( int taskID,
    __global int* taskCount,
    __global int* taskIDs )
{
   // do generic work
   ...
   if (condition == true) // true 50% of the time
   {
      // schedule additional work
      int arrayIdx = atomic_add( taskCount, 1 );
      taskIDs[arrayIdx] = taskID;
   }
}

void ComplexTaskPart2( int idx )
{
   DoWork( taskIDs[idx] );
}
```

```
void ComplexTask( int taskID )
{
   // do generic work
   ...
   if (condition == true) // true 50% of the time
   {
      // do additional work
   }
}
```

Reducing the number of atomics:

- Store ‘1’ or ‘0’ in an array depending on ‘condition’;
- Do a prefix sum over this array;
- Do a single atomic_add, which yields the base index;
- Use the values in the array as offsets to this base index.

Compaction

Stream Filtering

Stream filtering is used in *multi-pass kernels*.

Examples:

- 10k threads need to find an element in a linked list or a tree
- 10k threads trace a path from the camera to the light
- 10k threads update tanks and decide if the tank needs to fire

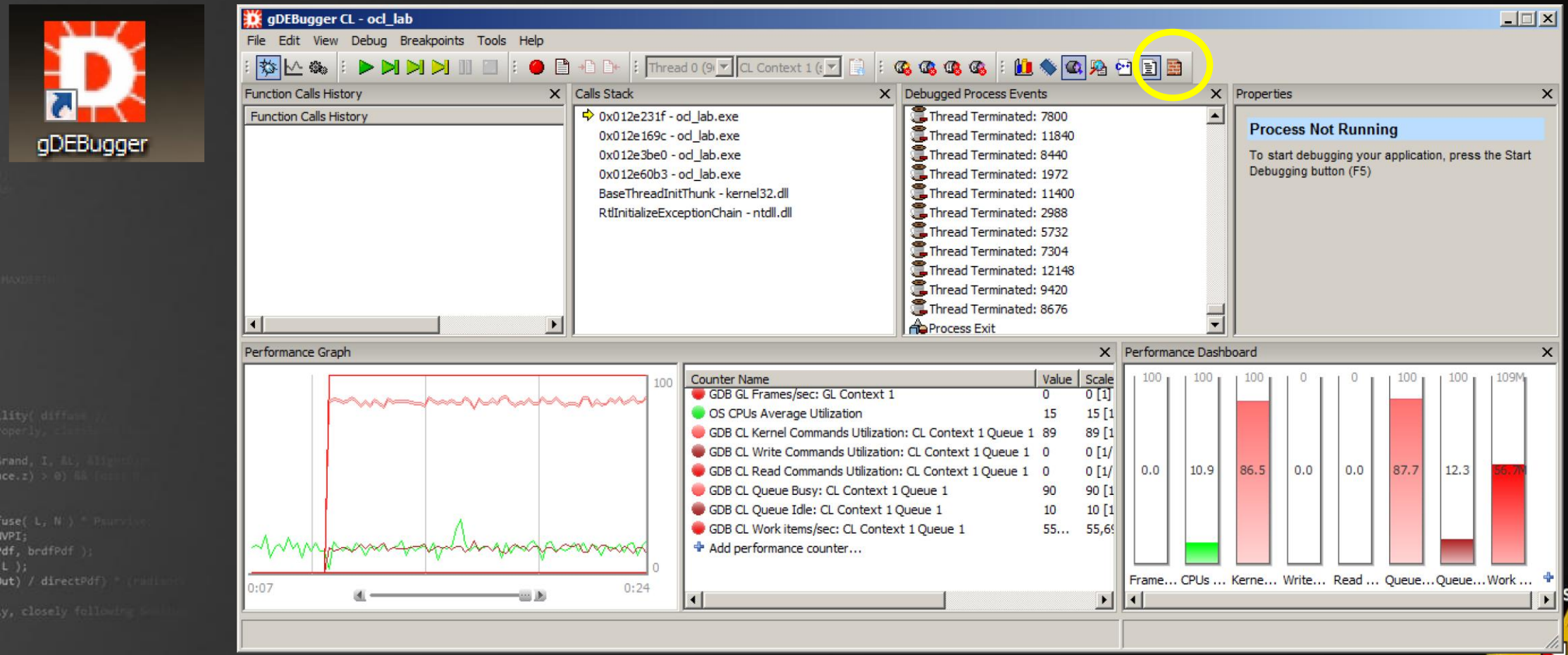
In all cases, the conditional code is executed by a continuous set of threads.

Compaction is used to restore *occupancy*.

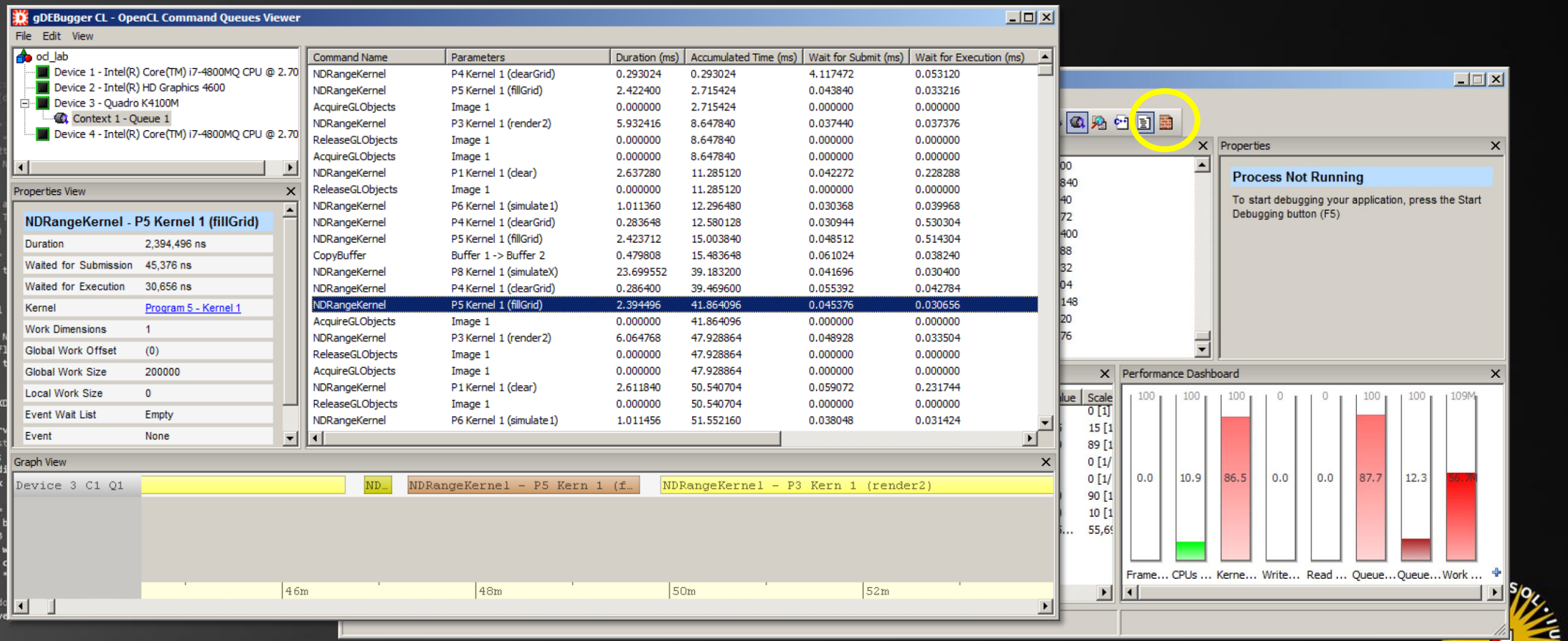
Today’s Agenda:

- Introduction
- The Prefix Sum
- Parallel Sorting
- Stream Filtering
- Optimizing GPU code

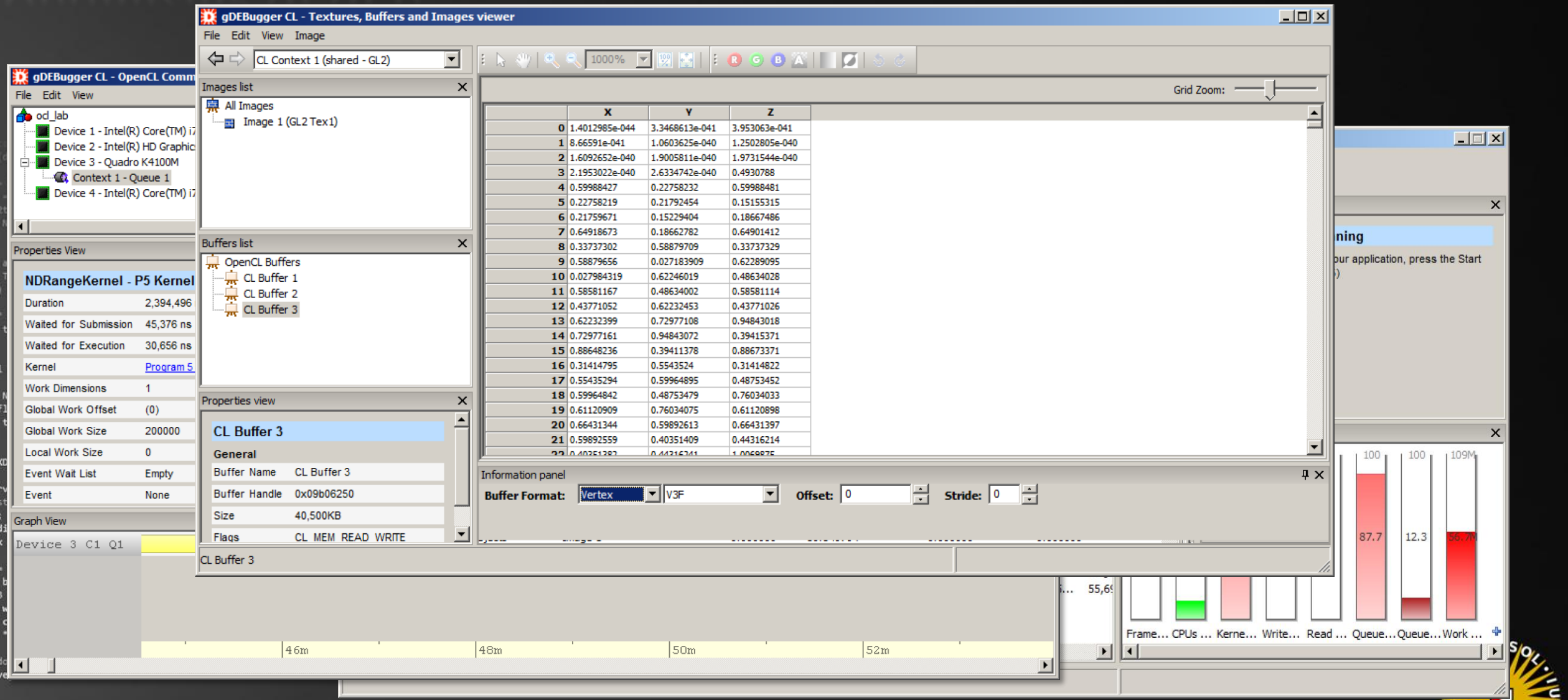
Optimizing GPGPU



Optimizing GPGPU



Optimizing GPGPU



Optimizing GPGPU

Faster OpenCL

```
temp = input[3] // input is in global mem

Instead, use:

if (get_local_id(0) == 0) local = input[3]
barrier(CLK_LOCAL_MEM_FENCE);
temp = local
```

1. Optimize memory usage

- Read data from global memory once
- Use local memory when possible
- Careful: reading the same global address in 32 threads is not a good idea!

2. Make sure there is enough work to hide latency

- On AMD: use multiples of 64 threads (called a ‘wavefront’)
- Tweak manually for performance, ideally per vendor / device

3. Minimize the number of host-to-device transfers, then their size

4. Minimize the number of kernel invocations

http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/opencl-optimization-guide/

Optimizing GPGPU

Faster OpenCL

Smaller things:

- Use float4 whenever possible
- Use predication rather than control flow
- Bypass short-circuiting
- Remove conditional code
- AOS vs SOA performance
- Reducing atomics
- Reduced precision math
- Pinned memory

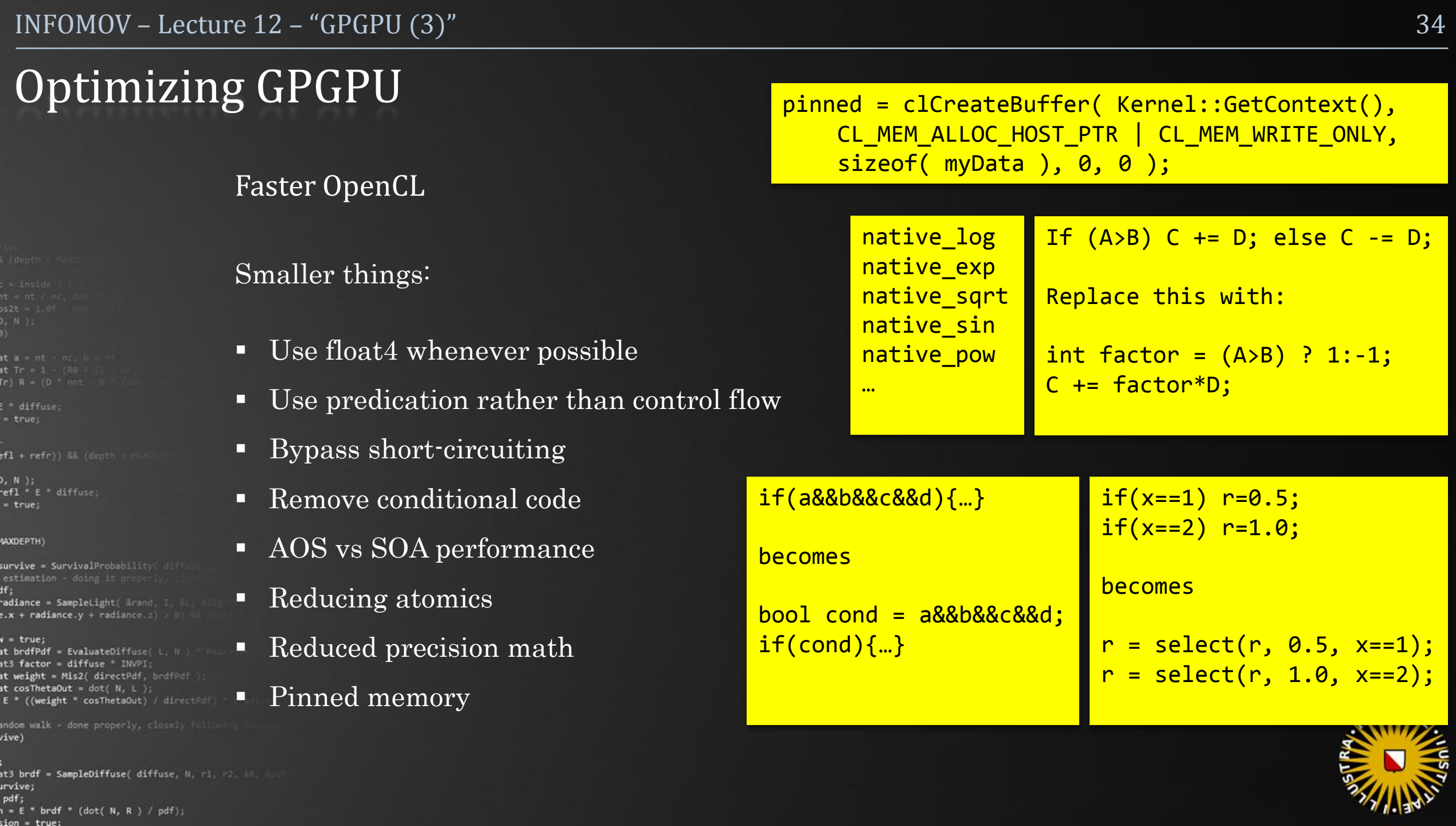
```
pinned = clCreateBuffer( Kernel::GetContext(),
    CL_MEM_ALLOC_HOST_PTR | CL_MEM_WRITE_ONLY,
    sizeof( myData ), 0, 0 );
```

```
native_log
native_exp
native_sqrt
native_sin
native_pow
…
```

```
If (A>B) C += D; else C -= D;

Replace this with:

int factor = (A>B) ? 1:-1;
C += factor*D;
```

```
if(a&&b&&c&&d){…}

becomes

bool cond = a&&b&&c&&d;
if(cond){…}
```

```
if(x==1) r=0.5;
if(x==2) r=1.0;

becomes

r = select(r, 0.5, x==1);
r = select(r, 1.0, x==2);
```



Today’s Agenda:

- Introduction
- The Prefix Sum
- Parallel Sorting
- Stream Filtering
- Optimizing GPU code

/INFOMOV/

END of “GPGPU (3)”

next lecture: TBD