Welcome!
Today’s Agenda:

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

Beyond “Let OpenCL Sort Them Out”

```c++
void Kernel::Run( const size_t count )
{
    cl_int error;
    CHECKCL( error = clEnqueueNDRangeKernel( queue, kernel, 1, 0, &count, 0, 0, 0, 0 ) );
    clFinish( queue );
}
```

Here:

- A queue is a command queue: we can have more than one*.
- ‘1’ is the dimensionality of the task (can be 1, 2 or 3).
- ‘count’ is the number of threads we are spawning (multiple of local work size, if specified).
- ‘0’ is the local work size (0 means: not specified, let OpenCL decide).

Introduction

Beyond “Let OpenCL Sort Them Out”

```c
void Kernel::Run( Buffer* buffer )
{
    glFinish();
    cl_int error;
    CHECKCL( error = clEnqueueAcquireGLObjects( queue, 1, buffer->GetDevicePtr(), 0, 0, 0 ) );
    CHECKCL( error = clEnqueueNDRangeKernel( queue, kernel, 2, 0, workSize, localSize, 0, 0, 0 ) );
    CHECKCL( error = clEnqueueReleaseGLObjects( queue, 1, buffer->GetDevicePtr(), 0, 0, 0 ) );
    clFinish( queue );
}
```

Here:

- We actually use the *device pointer* of a buffer (here: OpenGL data).
- ‘localSize’ is set: template default is \(\{32, 4\}\), i.e. 128 threads per SM.
- ‘Run’ is *synchronous* due to the use of `clFinish`. 
Beyond “Let OpenCL Sort Them Out”

A thread knows it’s place in the local group:

```c
__kernel void DoWork()
{
    // get the index of the thread in the global pool
    int idx = get_global_id( 0 );
    // get the index of the thread in the local set
    int localIdx = get_local_id( 0 );
    // determine in which warp the current thread is
    int warpIdx = localIdx >> 5;
    // determine in which lane we are
    int lane = localIdx & 31;
}
```
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.:


Note that the possibility of using linked lists on the GPU does not automatically justify their 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
- Persistent Threads
- Optimizing GPU code
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:

<table>
<thead>
<tr>
<th>input</th>
<th>1</th>
<th>2</th>
<th>2</th>
<th>1</th>
<th>4</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>inclusive</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>exclusive</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>10</td>
</tr>
</tbody>
</table>

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

```cpp
// 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

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

In parallel:

```c
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.
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 >= 2ᵈ⁻¹
            x[k] += x[k - 2ᵈ⁻¹]

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

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

In OpenCL:

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

```c
__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: Selection Sort

CAN WE DO BETTER?
GPU Sorting

```
// code snippet
```
**GPU 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.itl.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/oddn.htm
GPU Sorting

Full implementations of Bitonic sort for OpenCL:

https://github.com/Juanjdurillo/bitonicssortopencl

Also efficient on GPU:

Radix sort
Today’s Agenda:

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

Stream Filtering

```c
__kernel void UpdateTanks ( int taskID, __global Tank* tank )
{
    int idx = get_global_id( 0 );
    UpdatePosition();
    ConsiderFiring();
    Render();
    if (tank[idx].IsOffscreen())
    {
        RemoveFromGrid();
        Respawn();
        AddToGrid();
        ConsiderFiring();
    }
}
```
Compaction

Stream Filtering

```c
int offscreen[...]; int offscreenCount = 0;

__kernel void UpdateTanks ( int taskID, __global Tank* tank )
{
    int idx = get_global_id( 0 );
    UpdatePosition();
    ConsiderFiring();
    Render();
    if (tank[idx].IsOffscreen())
        offscreen[atomic_inc(&offscreenCount)] = idx;
}

__kernel void HandleOffscreenTanks( __global Tank* tank )
{
    ...}
```

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

```c
__local array[256], baseIdx[16];
int offscreen[...]; int offscreenCount = 0;
__kernel void UpdateTanks ( int taskID, __global Tank* tank ) {
    int idx = get_global_id( 0 );
    UpdatePosition();
    ConsiderFiring();
    Render();
    int isOffscreen = tank[idx].IsOffscreen() ? 1 : 0;
    // get index of thread in local group
    int lidx = get_local_id( 0 );
    // store in array
    array[lidx] = isOffscreen;
    // perform warp scan
    if (lidx & 31 == 0) {
        int count = WarpScan( &array[(lidx >> 5) << 5] );
        baseIdx[lidx >> 5] = atomic_add( &offscreenCount, count );
    }
    // store in ‘offscreen’ array
    if (isOffscreen) offscreen[baseIdx[lidx >> 5] + array[lidx]] = idx;
}
```

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

```c
int offscreen[...]; int offscreenCount = 0;
__kernel void UpdateTanks ( int taskID, __global Tank* tank )
{
    int idx = get_global_id( 0 );
    UpdatePosition();
    ConsiderFiring();
    Render();
    if (tank[idx].IsOffscreen())
        offscreen[atomic_inc( &offscreenCount )] = idx;
}
__kernel void HandleOffscreenTanks( __global Tank* tank )
{
    ...
}
```

How many threads execute this kernel?

*(CopyFromDevice() for just a single variable?)*
Compaction

Stream Filtering

```c
int offscreen[...]; int offscreenCount = 0;
__kernel void UpdateTanks ( int taskID, __global Tank* tank )
{
    int idx = get_global_id( 0 );
    UpdatePosition();
    ConsiderFiring();
    Render();
    if (tank[idx].IsOffscreen())
        offscreen[atomic_inc( &offscreenCount )] = idx;
}
__kernel void HandleOffscreenTanks( __global Tank* tank )
{
    if (get_global_id( 0 ) >= offscreenCount) return;
    ...
}
```

We start the kernel for all tanks.

This is fast, because all relevant tanks are handled by the first N threads; the remaining threads return immediately.
Compaction

Stream Filtering

Stream filtering is used in multi-pass kernels.

Examples:

- 10k threads need to Tick an instance of a specific class
- 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
- Persistent Threads
- Optimizing GPU code
How To Split N Jobs over M Cores

```c
int count = 1040;
int localSize = 128;
clEnqueueNDRangeKernel(queue, kernel, 1, 0, &count, &localSize, 0, 0, 0);
```

How does OpenCL split 1040 threads over 2 SMs?

- 128 + 128 (total: 256)
- 128 + 128 (total: 512)
- 128 + 128 (total: 768)
- 128 + 128 (total: 1024)
- 16 + 0

Actual process is more complex: warps get assigned to SMs when work is done: this is handled by the thread scheduler.
Persistent Threads

We can bypass the thread scheduler by using *persistent threads*.

For a GPU with 2 SMs, the ideal thread count is $32 \times 2 \times N$,
where $N$ depends on the register pressure and local memory demand in the kernel.

Each thread claims a job, and executes it.

Each thread terminates when the job pool is empty.

*A Study of Persistent Threads Style GPU Programming for GPGPU Workloads, Gupta et al.*
Persistent Threads

We can bypass the thread scheduler by using **persistent threads**.

```
__kernel void PersistentThread()
{
    while (1)
    {
        int jobIdx = atomic_inc( jobCounter );
        if (jobIdx >= totalJobs) return;
        ExecuteJob( jobIdx );
    }
}

void ExecuteJob( int jobIdx )
```

```
Persistent Threads

We can bypass the thread scheduler by using **persistent threads**.

```c
__local baseIdx[16]; // assuming max 16 warps per SM
__kernel void PersistentThread()
{
    while (1)
    {
        int localIdx = get_local_id( 0 );
        int warp = localIdx >> 5, lane = localIdx & 31;
        if (lane == 0)
            baseIdx[warp] = atomic_add( jobCounter, 32 );
        int jobIdx = baseIdx[warp] + lane;
        if (jobIdx >= totalJobs) return;
        ExecuteJob( jobIdx );
    }
}
```
Today’s Agenda:

- Introduction
- The Prefix Sum
- Parallel Sorting
- Stream Filtering
- Persistent Threads
- Optimizing GPU code
Faster OpenCL

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

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

Instead, use:

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

temp = local
```
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

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

Replace this with:

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

```
if(x==1) r=0.5;
if(x==2) r=1.0;
```
Today's Agenda:

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