

```
ics
& (depth < MAXDEPTH)
= inside ? 1.0f : 0.0f;
nt = nt / nc; ddn = ddn * ddn;
s2t = 1.0f - nnt * ddn;
D, N );
)
at a = nt - nc, b = nt * nc;
at Tr = 1 - (R0 + (1 - R0) * s2t);
Fr) R = (D * nnt - N * (ddn * s2t));
E * diffuse;
= true;
-
efl + refr)) && (depth < MAXDEPTH)
D, N );
refl * E * diffuse;
= true;
MAXDEPTH)
survive = SurvivalProbability( diffuse, r1, r2, &R, $pdf );
estimation - doing it properly, closely following 3e-10
if;
radiance = SampleLight( &rand, I, &L, &light );
e.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH)
w = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance
random walk - done properly, closely following 3e-10
ive)
;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, $pdf );
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
```

/INFOMOV/

Optimization & Vectorization

J. Bikker - Sep-Nov 2018 - Lecture 10: "GPGPU (3)"

Welcome!



Introduction

Beyond “Let OpenCL Sort Them Out”

```
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).

*: <http://sa09.idav.ucdavis.edu/docs/SA09-opencl-dg-events-stream.pdf>



Introduction

Beyond “Let OpenCL Sort Them Out”

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



2D task:

- Improves data locality
- Improves flow coherence
- Also available in CUDA
- Fractal data stalls: few
- Fractal flow coherence: apparently not a big deal.



Introduction

Beyond “Let OpenCL Sort Them Out”

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



Task size:

- Use to balance thread count / registers per thread.
- Tune per device class.
- Auto-tuning possible?
- Do not trust OpenCL.



Introduction

Beyond “Let OpenCL Sort Them Out”

```

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



clFinish:

- Not a good idea: has CPU idling.
- Queue enforces order.
- Multiple queues are useful.
- How about running part of the fractal on the CPU?
- How do we balance CPU / GPU work?



Introduction

Beyond “Let OpenCL Sort Them Out”

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

A thread knows it's place in the global task set, but also the local group:

```
__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.:

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 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



```

ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn * ddn;
        cos2t = 1.0f - nnt * nnt;
        D, N );
    }
}

at a = nt - nc; b = nt * nc;
at Tr = 1 - (R0 + (1 - R0) *
Tr) R = (D * nnt - N * (ddn
E * diffuse;
= true;

efl + refr)) && (depth < MAXDEPTH)
D, N );
refl * E * diffuse;
= true;

MAXDEPTH)
survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &L, &light;
e.x + radiance.y + radiance.z) > 0) && (cosThetaOut > 0)
w = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance
random walk - done properly, closely following
ive)
;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf;
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```


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

Prefix Sum

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

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)

```
ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn * nc;
        ps2t = 1.0f - nnt * nnt;
        D, N );
    }
}

at a = nt - nc; b = nt * nc;
at Tr = 1 - (R0 + (1 - R0) * R);
Tr) R = (D * nnt - N * (ddn * nnt));

E * diffuse;
= true;

efl + refr)) && (depth < MAXDEPTH)
D, N );
refl * E * diffuse;
= true;

MAXDEPTH)

survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &L, &light
e.x + radiance.y + radiance.z) > 0) && (survive)

v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance

random walk - done properly, closely following
ive)

at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
```

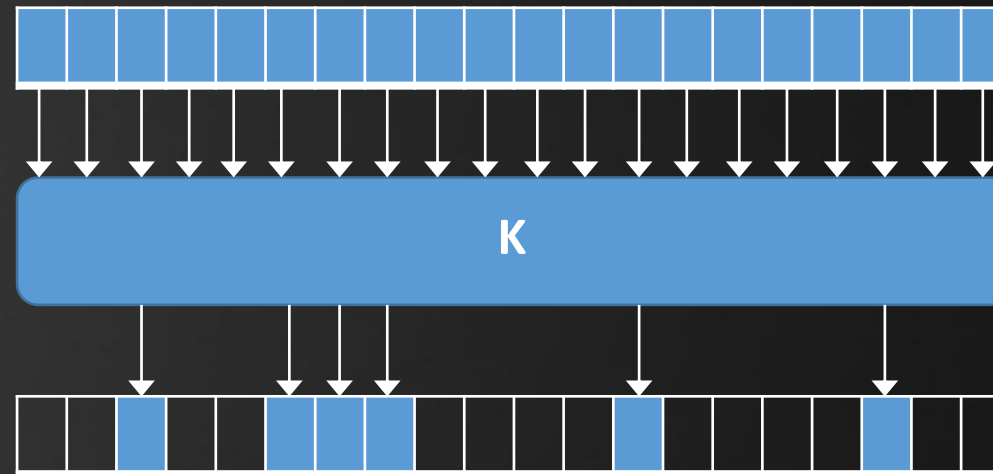


Prefix Sum

Prefix Sum

The prefix sum is used for *compaction*.

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



Prefix Sum

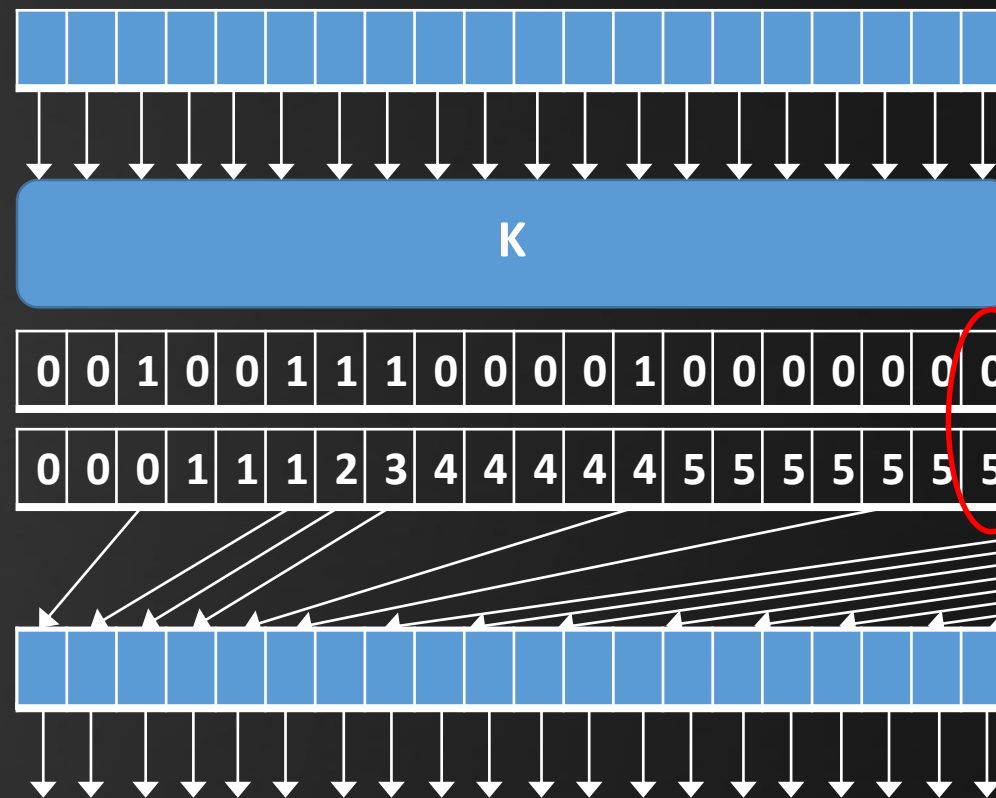
Prefix Sum - Compaction

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

```

...
    & (depth < MAXDEPTH)
...
    inside ? 1 : 0;
    nt = nt / nc; ddn = ddn / nc;
    ps2t = 1.0f - nnt * ddn;
    D, N );
    0);
...
    at a = nt - nc, b = nt + nc;
    at Tr = 1 - (R0 + (1 - R0) * ps2t);
    Tr) R = (D * nnt - N * (ddn *
...
    E * diffuse;
    = true;
...
    efl + refr)) && (depth < MAXDEPTH)
    D, N );
    efl * E * diffuse;
    = true;
...
MAXDEPTH)
survive = SurvivalProbability( diffuse, 1);
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &L, &light);
e.x + radiance.y + radiance.z) > 0) && (rand < 1);
...
    w = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
...
random walk - done properly, closely following 3rd
ive)
...
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf);
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;

```



output array size

boolean array

exclusive prefix sum

output array



Prefix Sum

Prefix Sum

```

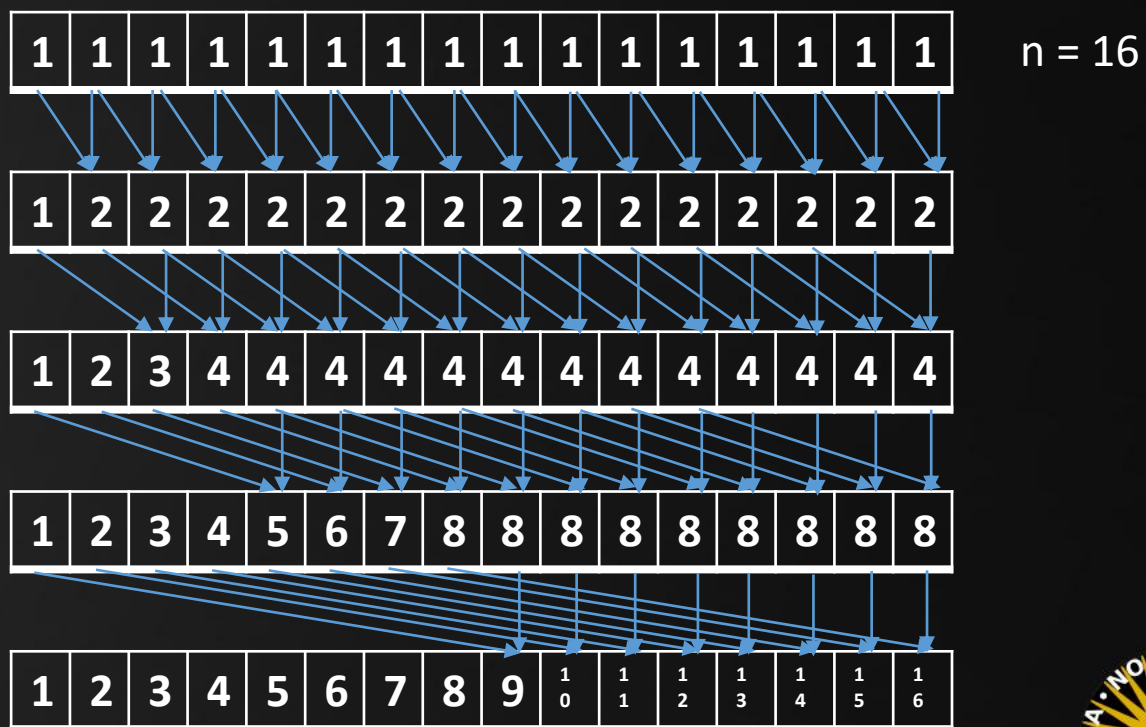
out[0] = 0;
for ( i = 1; i < n; i++ ) out[i] = in[i-1] + out[i-1];
    
```

In parallel:

```

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

- 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];

```

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 warp_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;
}

```



Prefix Sum

```

...ics
& (depth < MAXDEPTH)
...
c = inside ? 1 : 0;
nt = nt / nc; ddn = ddn / nd;
ps2t = 1.0f - nnt * nnt;
D, N );
...
)
...
at a = nt - nc, b = nt - nc;
at Tr = 1 - (R0 + (1 - R0) * R);
Tr) R = (D * nnt - N * nnt) / (D * nnt - N * nnt);
...
E * diffuse;
= true;
...
efl + refr) && (depth < MAXDEPTH)
D, N );
refl * E * diffuse;
= true;
...
MAXDEPTH)
survive = SurvivalProbability( diffuse, 1);
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &L, &light);
e.x + radiance.y + radiance.z) > 0) && (depth <
...
v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance
...
random walk - done properly, closely following
ive)
...
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf);
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```

```

int warp_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;
}

```



Prefix Sum

Take-away:

- A “scan” is useful for compacting arrays.
- The naïve scan has an obvious loop dependency.
- It is nevertheless possible to run the scan in parallel.
- Especially at the warp level, this heavily leans on a core GPU mechanism:

lockstep SIMT processing.

```

ics
& (depth < MAXDEPTH)
    c = inside ? 1 : 0;
    nt = nt / nc; ddn = ddn * c;
    ps2t = 1.0f - nnt * (ddn * c);
    D, N );
    )
    at a = nt - nc, b = nt * nc;
    at Tr = 1 - (R0 + (1 - R0) * c);
    Tr) R = (D * nnt - N * (ddn * c));
    E * diffuse;
    = true;
    -
    refl + refr)) && (depth < MAXDEPTH)
    D, N );
    refl * E * diffuse;
    = true;
    MAXDEPTH)
    survive = SurvivalProbability( diffuse, r1, r2, &R, &pdf );
    estimation - doing it properly, closely following 3e-10
    if;
    radiance = SampleLight( &rand, I, &L, &light );
    e.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH)
    w = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
    random walk - done properly, closely following 3e-10
    (survive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
    
```



Sorting

GPU Sorting

Observation:

- We frequently need sorting in our algorithms.

But:

- Most sorting algorithms are divide and conquer algorithms.

```

...
    & (depth < MAXDEPTH)
...
    if (inside ? 1 : 0) {
        nt = nt / nc; ddn = ddn * nc;
        cos2t = 1.0f - nnt * nnt;
        D, N );
    }
...
    at a = nt - nc; b = nt + nc;
    at Tr = 1 - (R0 + (1 - R0) * ddn);
    at R = (D * nnt - N * (ddn * ddn));
...
    E * diffuse;
    = true;
...
    refl + refr) && (depth < MAXDEPTH)
...
    D, N );
    refl * E * diffuse;
    = true;
...
MAXDEPTH)
...
survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &L, &light);
e.x + radiance.y + radiance.z) > 0) && (cosThetaOut > 0)
...
    w = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
...
random walk - done properly, closely following
survive)
...
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf);
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;

```



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

```

...ics
& (depth < MAXDEPTH)

c = inside ? 1.0f : 0.0f;
nt = nt / nc, ddn = ddn * c;
ps2t = 1.0f - nnt * ddn;
D, N );
0);

at a = nt - nc, b = nt * a;
at Tr = 1 - (R0 + (1 - R0) * a);
Tr) R = (D * nnt - N * ddn) * a;

E * diffuse;
= true;

efl + refr)) && (depth < MAXDEPTH)
D, N );
refl * E * diffuse;
= true;

MAXDEPTH)

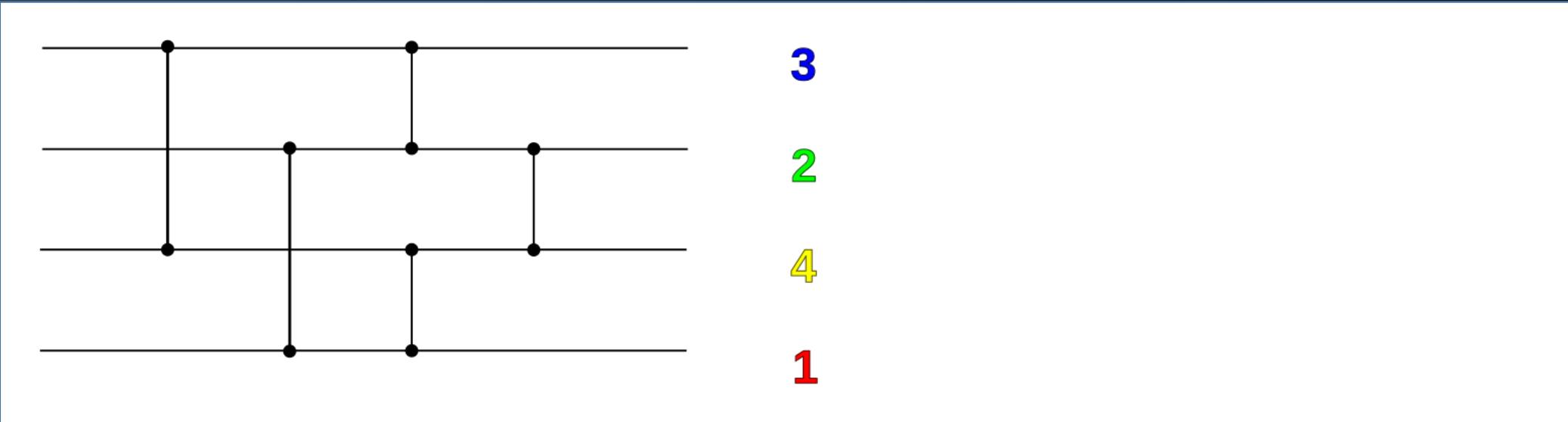
survive = SurvivalProbability( t,
estimation - doing it properly.
if;
radiance = SampleLight( &rand, 1);
e.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH)

w = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance

random walk - done properly, closely following Sedgewick's
(ive)

;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```



Sorting

GPU Sorting

```

...ics
& (depth < MAXDEPTH)

c = inside ? 1 : 0;
nt = nt / nc; ddn = ddn * ddn;
cos2t = 1.0f - nnt * nnt;
D, N );
0);

at a = nt - nc, b = nt * nc;
at Tr = 1 - (R0 + (1 - R0) * c);
Tr) R = (D * nnt - N * ddn)

E * diffuse;
= true;

efl + refr) && (depth < MAXDEPTH)

D, N );
refl * E * diffuse;
= true;

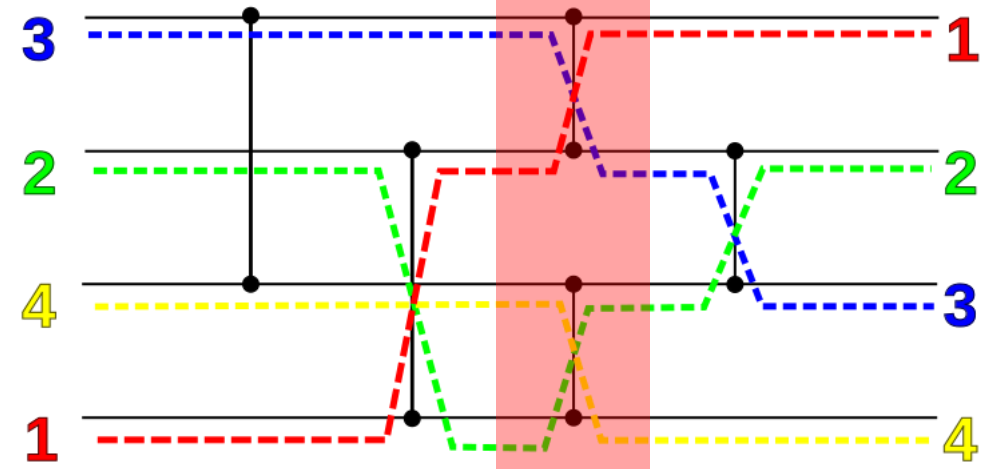
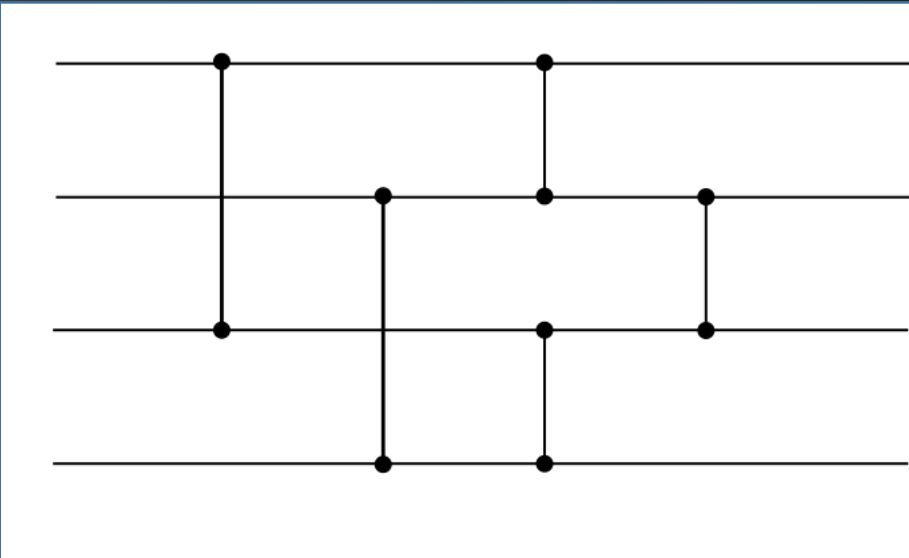
MAXDEPTH)

survive = SurvivalProbability( t,
estimation - doing it properly.
if;
radiance = SampleLight( &rand, 1
e.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH)

v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance

random walk - done properly, closely following Saito's
ive)

at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
    
```



Sorting

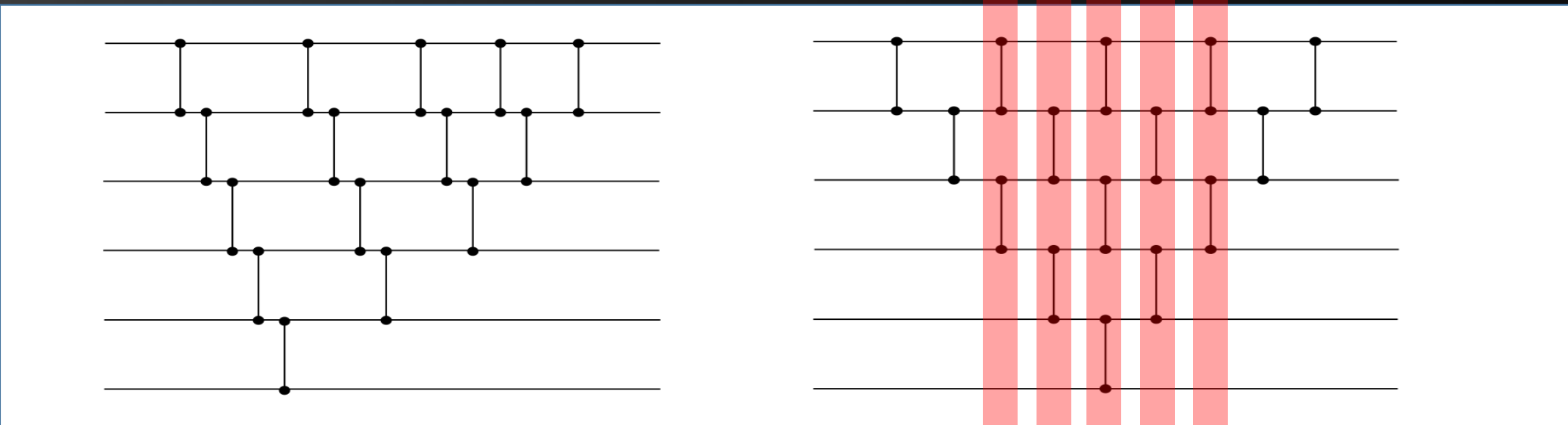
GPU Sorting

Bubblesort:

```

...ics
& (depth < MAXDEPTH)
...
c = inside ? 1 : 0;
nt = nt / nc; ddn = ddn * ddn;
ps2t = 1.0f - nnt * nnt;
D, N );
0)
at a = nt - nc; b = nt * nc;
at Tr = 1 - (R0 + (1 - R0) *
Tr) R = (D * nnt - N * (ddn
E * diffuse;
= true;
...
efl + refr) && (depth < MAXDEP
D, N );
refl * E * diffuse;
= true;
MAXDEPTH)
survive = SurvivalProbability( t
estimation - doing it properly
df;
radiance = SampleLight( &rand, 1
e.x + radiance.y + radiance.z) > 0) && (rand
w = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) *
...
random walk - done properly, closely followi
vive)
...
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```



Sorting network, **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

On a parallel device, the optimal sorting network
is the one with the smallest depth / smallest span / shortest critical path.

```

ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn * nc;
        ps2t = 1.0f - nnt * nnt;
        D, N );
    }
}

at a = nt - nc; b = nt * nc;
at Tr = 1 - (R0 + (1 - R0) * R);
Tr) R = (D * nnt - N * (ddn * nnt));

E * diffuse;
= true;

-
efl + refr)) && (depth < MAXDEPTH)
{
    D, N );
    refl * E * diffuse;
    = true;
}

MAXDEPTH)

survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following Serdar's
if;
radiance = SampleLight( &rand, I, &L, &light);
e.x + radiance.y + radiance.z) > 0) && (abs(radiance.x) +
w = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance.x +
random walk - done properly, closely following Serdar's
vive)
;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```

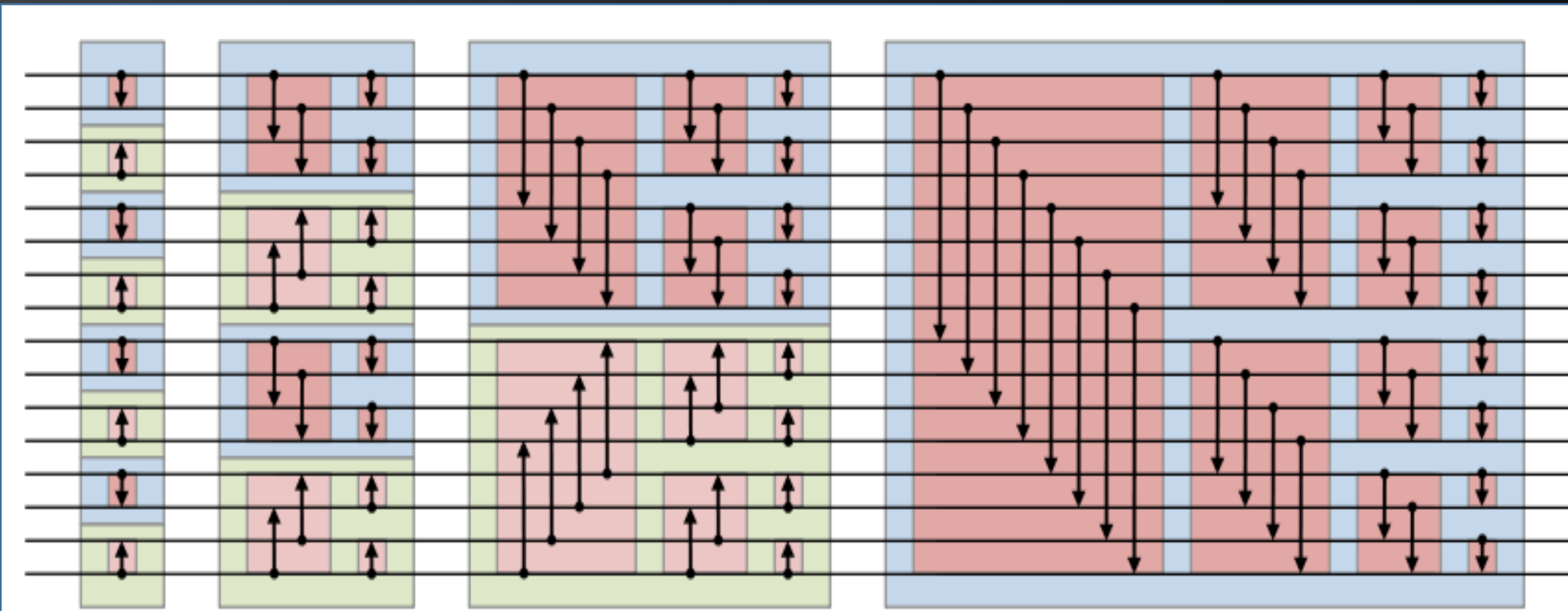


Sorting

GPU Sorting

Bitonic sort^{*,**}:

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



Compare element in top half with element in bottom half

Subdivide red box and recurse until a single comparison is left

Ensure that the largest number is at the arrow point

All boxes can execute in parallel.

```

ics
& (depth <
t = inside
nt = nt / n
s2t = 1.0f
D, N );
);
at a = nt -
at Tr = 1 -
Tr) R = (D
E * diffuse
= true;
efl + refr)
D, N );
refl * E *
= true;
MAXDEPTH)
survive = S
estimation
df;
radiance =
e.x + radia
w = true;
at brdfPdf
at3 factor
at weight =
at cosTheta
E * ((weight * cosThetaOut) / directPdf) * (radiance
random walk - done properly, closely following 3D
ive)
;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
    
```

*: 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>



Sorting

GPU Sorting

Full implementations of Bitonic sort for OpenCL:

<https://github.com/Juanjdurillo/bitonicsortopencl>

http://www.bealto.com/gpu-sorting_parallel-bitonic-1.html

Also efficient on GPU: Radix sort.

Side note:

<https://github.com/komrad36/SortingNetworks> (SSE sorting networks, 2-6 elements)

<http://pages.ripco.net/~jgamble/nw.html> (optimal sorting networks for $N \leq 32$)

(that last one tells us that data for a warp can be sorted in 31 parallel steps)



Compaction

Stream Filtering

```

...
    & (depth < MAXDEPTH)
...
    if (inside ? 1 : 0)
        nt = nt / nc; ddn = ddn * nc;
        ps2t = 1.0f - nnt * ddn;
        D, N );
    )
...
    at a = nt - nc, b = nt * nc;
    at Tr = 1 - (R0 + (1 - R0) *
    Tr) R = (D * nnt - N * (ddn
...
    E * diffuse;
    = true;
...
    refl + refr)) && (depth < MAXDEPTH)
    D, N );
    refl * E * diffuse;
    = true;
...
MAXDEPTH)
survive = SurvivalProbability( diffuse,
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &L, &light;
e.x + radiance.y + radiance.z) > 0) && (depth <
w = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * radiance;
...
random walk - done properly, closely following
survive)
...
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf;
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```

```

__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

```

...
    & (depth < MAXDEPTH)
...
    = inside ? 1.0f : 0.0f;
    nt = nt / nc; ddn = ddn / nc;
    ps2t = 1.0f - nnt * ddn;
    D, N );
    )
...
    at a = nt - nc, b = nt - nc;
    at Tr = 1 - (R0 + (1 - R0) *
    Tr) R = (D * nnt - N * (ddn
...
    E * diffuse;
    = true;
...
    refl + refr)) && (depth < MAXDEPTH)
    D, N );
    refl * E * diffuse;
    = true;
...
MAXDEPTH)
survive = SurvivalProbability( diffuse,
estimation - doing it properly, closely
if;
radiance = SampleLight( &rand, I, &L, &light
e.x + radiance.y + radiance.z) > 0) && (survive
{
    w = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * radiance;
...
random walk - done properly, closely following
survive)
...
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```

```

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.



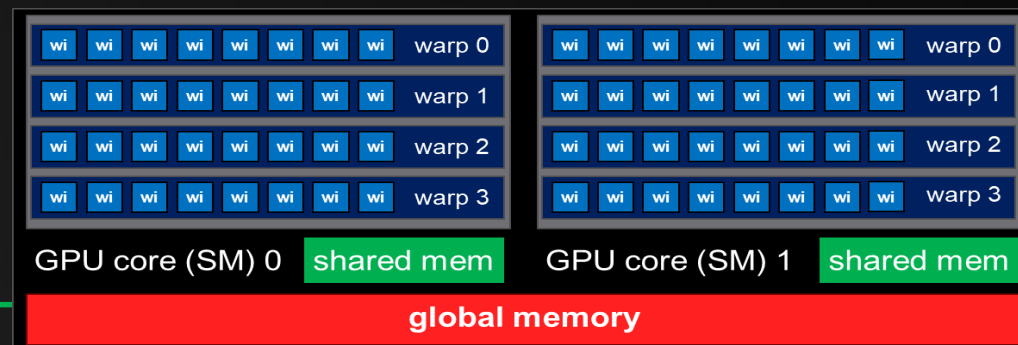
Compaction

Stream Filtering

```

__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
    int count = WarpScan( &array[(lidx >> 5) << 5] );
    if (lidx & 31 == 0)
        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

```

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

```

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.



Optimizing GPGPU

Faster OpenCL

1. Optimize memory usage

- Read data from global memory once
- Use local memory when possible
- Careful: reading the same global address in 256 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

```
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
```

```

...ics
& (depth < MAXDEPTH)
...
= inside ? 1.0f : 0.0f;
nt = nt / nc; ddh = ddh / nd;
s2t = 1.0f - nnt * nnt;
D, N );
0);
at a = nt - nc, b = nt - nc;
at Tr = 1 - (R0 + (1 - R0) *
Tr) R = (D * nnt - N * (ddh
E * diffuse;
= true;
...
efl + refr)) && (depth < MAXDEPTH)
D, N );
refl * E * diffuse;
= true;
...
MAXDEPTH)
survive = SurvivalProbability( diffuse,
estimation - doing it properly, closely
if;
radiance = SampleLight( &rand, I, &L, &light
e.x + radiance.y + radiance.z) > 0) && (survive
v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) *
...
random walk - done properly, closely following
ive)
...
at3 brdf = SampleDiffuse( diffuse, N, r1, r2,
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```



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

Cache line: 128B

```
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);
```



```
ics
& (depth < MAXDEPTH)
{
    if ( ! inside )
        return 0;
    Vec nt = n * t;
    Vec ddn = d * n;
    float r0 = 1.0f - nnt * ddn;
    Vec D, N );
    Vec R = (D * nnt - N * ddn) * r0;
    Vec E * diffuse;
    Vec refl * E * diffuse;
    Vec refl + refr)) && (depth < MAXDEPTH)
    Vec D, N );
    Vec refl * E * diffuse;
    Vec refl + refr)) && (depth < MAXDEPTH)
    Vec survive = SurvivalProbability( diffuse );
    Vec estimation - doing it properly, closely following
    Vec if;
    Vec radiance = SampleLight( &rand, I, &L, &light );
    Vec e.x + radiance.y + radiance.z > 0) && (depth < MAXDEPTH)
    Vec w = true;
    Vec at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    Vec at3 factor = diffuse * INVPI;
    Vec at weight = Mis2( directPdf, brdfPdf );
    Vec at cosThetaOut = dot( N, L );
    Vec E * ((weight * cosThetaOut) / directPdf) * (radiance
    Vec random walk - done properly, closely following
    Vec survive)
    Vec ;
    Vec at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    Vec survive;
    Vec pdf;
    Vec n = E * brdf * (dot( N, R ) / pdf);
    Vec sion = true;
}
```

Today's Agenda:

- Don't Trust the Template
- The Prefix Sum
- Parallel Sorting
- Stream Filtering
- Optimizing GPU code



/INFOMOV/

END of “GPGPU (3)”

next lecture: “fixed point”

```
ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn * ddn;
        cos2t = 1.0f - nnt * nnt;
        D, N );
    }
    at a = nt - nc, b = nt + nc;
    at Tr = 1 - (R0 + (1 - R0) * r);
    (Tr) R = (D * nnt - N * (ddn *
    E * diffuse;
    = true;
    -
    refl + refr)) && (depth < MAXDEPTH)
    D, N );
    refl * E * diffuse;
    = true;
    MAXDEPTH)
    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, closely following
    if;
    radiance = SampleLight( &rand, I, &L, &light);
    e.x + radiance.y + radiance.z) > 0) && (cosThetaOut > 0)
    w = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
    random walk - done properly, closely following
    (survive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
}
```

