Reduction

OpenCL Tutorial
Part 2.1

Guillermo Marcus
guillermo.marcus@gmail.com

Reduction operation

Reduces a set of values to a single output set, often to a single value.

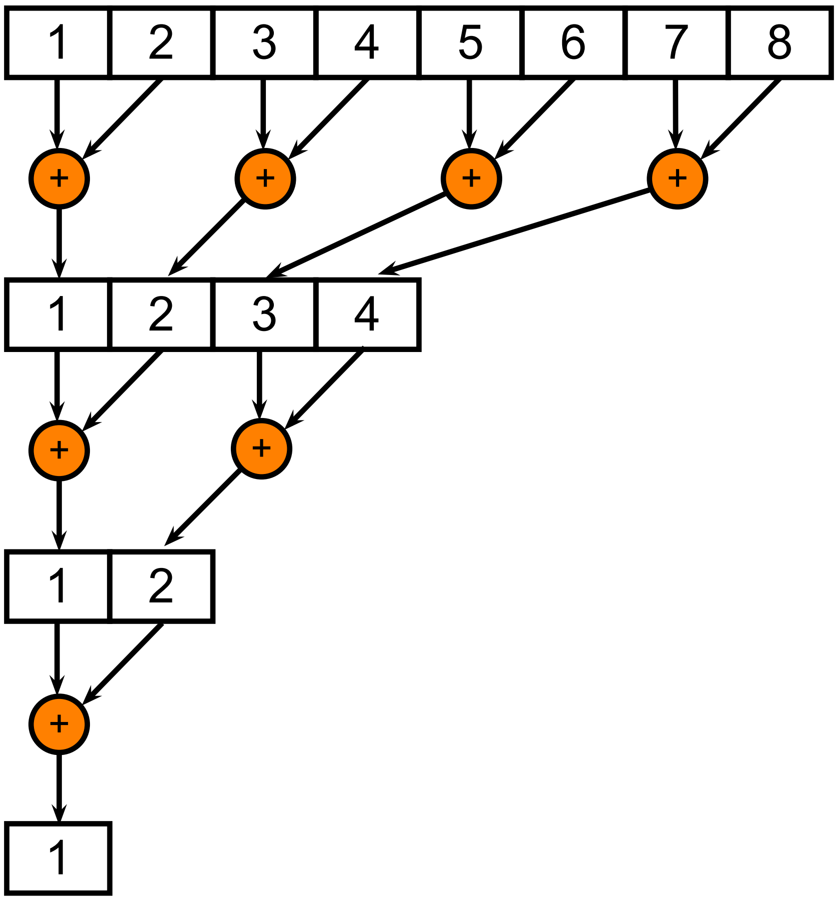
$$s = \sum_{i=0}^{N} f(x_i)$$

$$p = \prod_{i=0}^{N} f(x_i)$$

Whole set of operations, can be summation, product, histograms...

$$h_k = \sum_{i=0}^{N} (x_i = k)?1 : 0$$

Summation



For an array of size N, requires $\log_2 N$ steps, and N-1 operations.

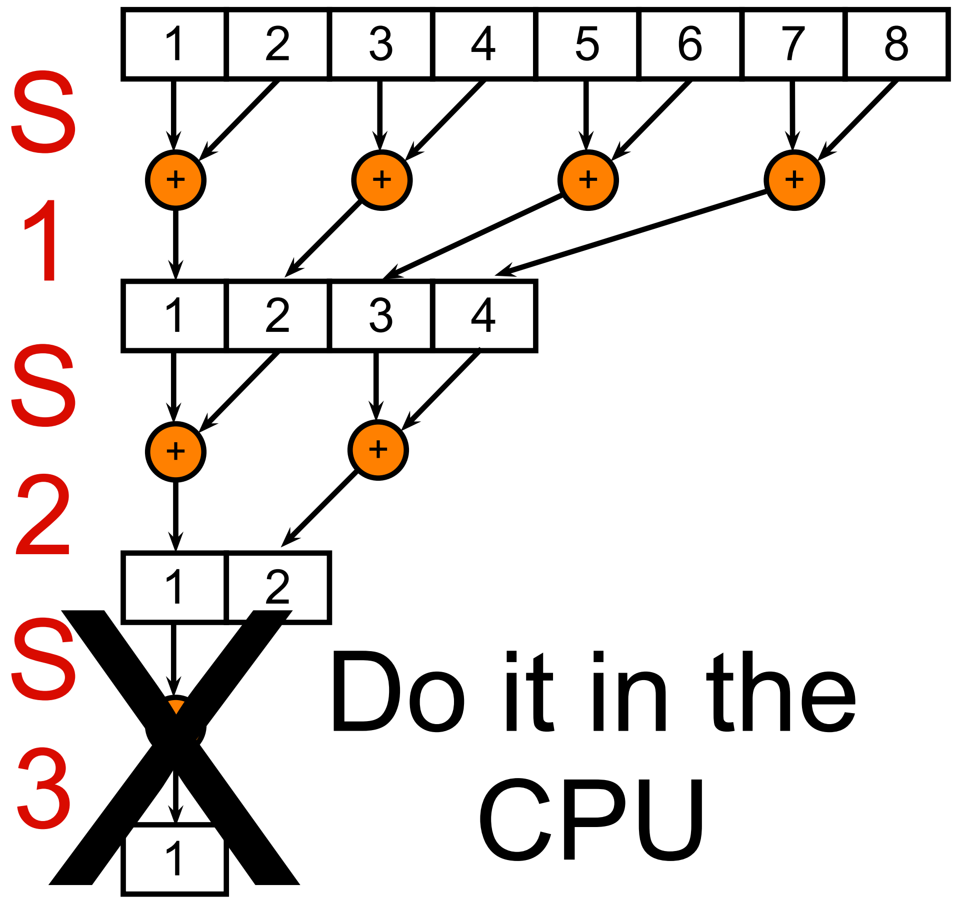
For this tutorial, we assume N is a power of 2 and the operations (+) are commutative.

Algorithm Overview

The kernel is called multiple times, once per $\log_2$ step.

Buffer pointers are switched on each iteration

Do not reduce to size 1, but to size M, then do it in the CPU.



Hands on!

Let's begin with testing the first version.

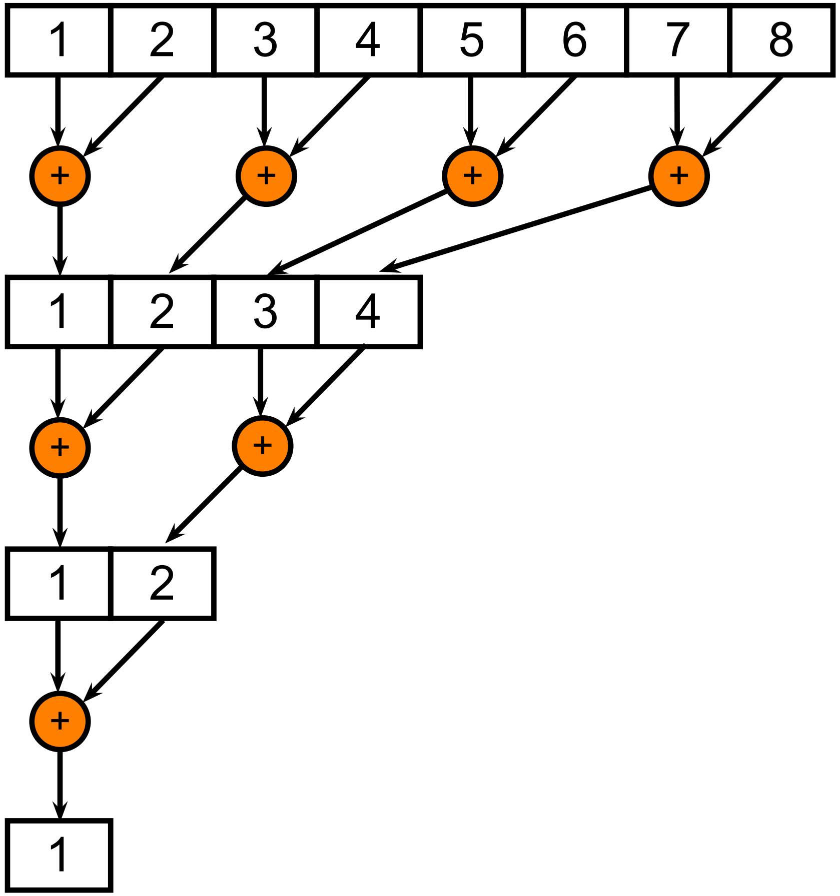
Compiled the same as the Hello Vector example.

So, what is this code doing?

Version 0

Add element i with element i+1

step is 1 (half size on each iteration)



Comparison

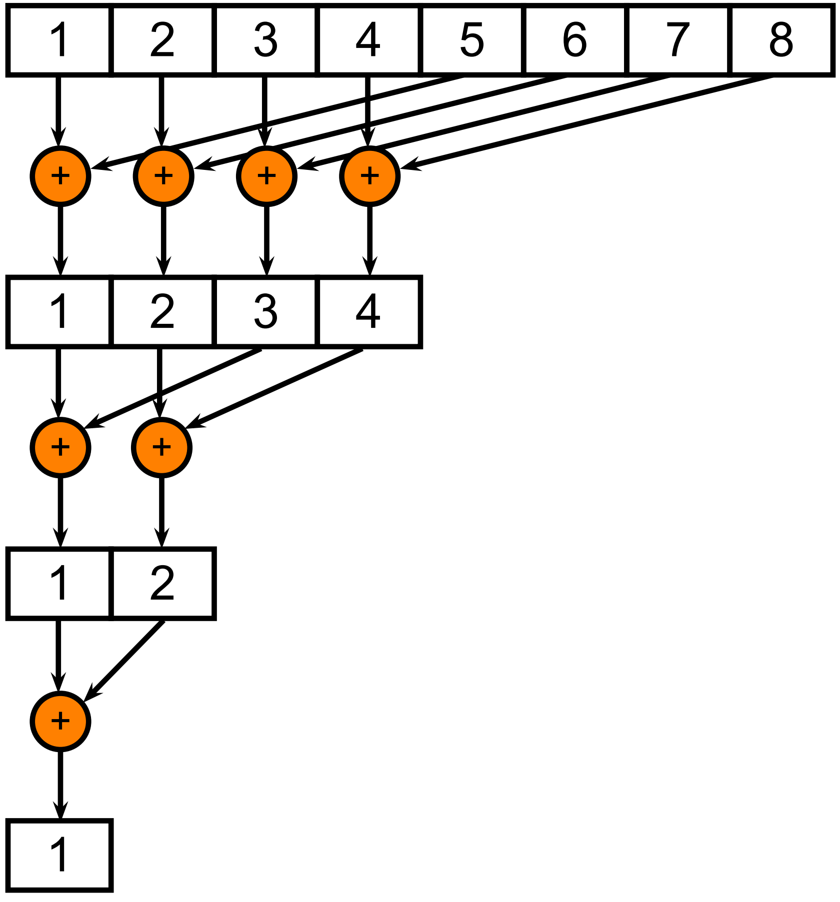
kernel	t gpu	t kernel	speedup
0	7.682 ms	6.376 ms	6.14

Task 1

Add element i with element i+(size/half)

Minimum size to add is 64

Step is still 1 (each iteration half of the previous)



Comparison

kernel	t gpu	t kernel	speedup
0	7.682 ms	6.376 ms	6.14
1	5.873 ms	4.088 ms	7.93

Task 2

On the previous versions, we read and write values on each iteration

Let's use the shared, local memory to process a block of values and reduce them to a single value.

Therefore, let's do minimum size of 1024, and a step size of 10 ($2^{10} = 1024$). Use 64 threads / block.

Task 2 - How to use local memory?

```
__kernel void reduce2(__const int size, __global int *A, __global int *B)
{
        __local int lcl[1024];
        unsigned int lidx = get_local_id(0);
        unsigned int idx = get_group_id(0);
        unsigned int step = get_local_size(0);
// Prefetch
        for(int i=0; i<1024;i+=64)
                lcl[ i + lidx ] = A[ idx*1024 + i + lidx ];
        barrier( CLK_GLOBAL_MEM_FENCE );

// Now, we can reduce the 1024-block to 1
        for(int s=(1024/2); s >= 1; s=(s>>1)) {
                for(int s2=0; s2 < s; s2 += step) {
                        if ( s2+lidx < s)
                                lcl[ s2+lidx ] += lcl[ s + s2 + lidx ];
                }
                barrier( CLK_LOCAL_MEM_FENCE );
        }
        barrier( CLK_LOCAL_MEM_FENCE);
        if (lidx == 0)
                B[ idx ] = lcl[0];
}
```

Comparison

kernel	t gpu	t kernel	speedup
0	7.682 ms	6.376 ms	6.14
1	5.873 ms	4.088 ms	7.93
2	5.795 ms	5.239 ms	8.65

Task 3

V2 is a good start, but inefficient. Let work on it in detail.

```
__kernel void reduce2(__const int size, __global int *A, __global int *B)
{
        __local int lcl[1024];
        unsigned int lidx = get_local_id(0);
        unsigned int idx = get_group_id(0);
        unsigned int step = get_local_size(0);
// Prefetch
        for(int i=0; i<1024;i+=64)
                lcl[ i + lidx ] = A[ idx*1024 + i + lidx ];
        barrier( CLK_GLOBAL_MEM_FENCE );

// Now, we can reduce the 1024-block to 1
        for(int s=(1024/2); s >= 1; s=(s>>1)) {
                for(int s2=0; s2 < s; s2 += step) {
                        if ( s2+lidx < s)
                                lcl[ s2+lidx ] += lcl[ s + s2 + lidx ];
                }
                barrier( CLK_LOCAL_MEM_FENCE );
        }
        barrier( CLK_LOCAL_MEM_FENCE);
        if (lidx == 0)
                B[ idx ] = lcl[0];
}
```

We are wasting local memory. We can load two values, reduce, and write into local memory

We do not need a compute barrier here, just a memory fence!

This IF is expensive, but only needed when we have a few last threads, so..we split this.

We do not need this barrier on each iteration. If the size is smaller than the warp size, it is already parallel. So we can unroll the last iterations.

Task 3 - Code

```
__kernel void reduce3(__const int size, __global int *A, __global int *B) {
        __local int lcl[1024];
        unsigned int lidx = get_local_id(0);
        unsigned int idx = get_group_id(0);
        unsigned int step = get_local_size(0);
// Prefetch
        for(int i=0; i<1024;i+=64)
                lcl[ i + lidx ] = A[ idx*1024 + i + lidx ] + A[ idx*1024 + i + lidx + size ];
        write_mem_fence( CLK_LOCAL_MEM_FENCE );
// Optimize the block, remove the if.
        for(int s=(1024/2); s >= 64; s=(s>>1)) {
                for(int s2=0; s2 < s; s2 += step) {
                        lcl[ s2+lidx ] += lcl[ s + s2 + lidx ];
                }
                barrier( CLK_LOCAL_MEM_FENCE );
        }
// do the last iterations from 64 to 1
        if (lidx < 32) lcl[ lidx ] += lcl[ lidx + 32 ];
        if (lidx < 16) lcl[ lidx ] += lcl[ lidx + 16 ];
        if (lidx < 8 ) lcl[ lidx ] += lcl[ lidx + 8 ];
        if (lidx < 4 ) lcl[ lidx ] += lcl[ lidx + 4 ];
        if (lidx < 2 ) lcl[ lidx ] += lcl[ lidx + 2 ];
        if (lidx < 1 ) lcl[ lidx ] += lcl[ lidx + 1 ];

        if (lidx == 0)
                B[ idx ] = lcl[0];
}
```

How far can we go?

We can completely unroll the loops, which gives a small but appreciable amount. (T4)

We need to fine tune the occupancy of each Symmetric Multiprocessor (SM) in the GPU. This is done in T5 and T6.

Comparison

kernel	t gpu	t kernel	speedup
0	7.682 ms	6.376 ms	6.14
1	5.873 ms	4.088 ms	7.93
2	5.795 ms	5.239 ms	8.65
3	2.556 ms	2.019 ms	18.19
4	2.334 ms	1.820 ms	19.87
5	2.102 ms	1.524 ms	22.43
6	1.737 ms	1.236 ms	27.07

End of Reduction Exercise

Comparison

kernel	t gpu	t kernel	speedup
0	7.682 ms	6.376 ms	6.14
1	5.873 ms	4.088 ms	7.93
2	5.795 ms	5.239 ms	8.65
3	2.556 ms	2.019 ms	18.19
4	2.334 ms	1.820 ms	19.87
5	2.102 ms	1.524 ms	22.43

Comparison

kernel	t gpu	t kernel	speedup
0	7.682 ms	6.376 ms	6.14
1	5.873 ms	4.088 ms	7.93
2	5.795 ms	5.239 ms	8.65
3	2.556 ms	2.019 ms	18.19

Version 4

- OK, now let's fully unroll the loops and see if there is any difference.....

Version 4

```
__kernel void reduce4(__const int size, __global int *A, __global int *B) {               __local int lcl[1024];            unsigned int lidx = get_local_id(0);    unsigned int idx =
get_group_id(0);   unsigned int step = get_local_size(0);              // Prefetch            int global_index1 = idx*1024 + lidx;  int global_index2 = idx*1024 + lidx + size;       lcl[ lidx + 0
] = A[ global_index1 + 0    ] + A[ global_index2 + 0    ];   lcl[ lidx + 64   ] = A[ global_index1 + 64   ] + A[ global_index2 + 64   ];          lcl[ lidx + 128 ] = A[ global_index1 + 128 ] + A[
global_index2 + 128 ];         lcl[ lidx + 192 ] = A[ global_index1 + 192 ] + A[ global_index2 + 192 ];       lcl[ lidx + 256 ] = A[ global_index1 + 256 ] + A[ global_index2 + 256 ];
lcl[ lidx + 320 ] = A[ global_index1 + 320 ] + A[ global_index2 + 320 ];       lcl[ lidx + 384 ] = A[ global_index1 + 384 ] + A[ global_index2 + 384 ];       lcl[ lidx + 448 ] = A[
global_index1 + 448 ] + A[ global_index2 + 448 ];         lcl[ lidx + 512 ] = A[ global_index1 + 512 ] + A[ global_index2 + 512 ];       lcl[ lidx + 512 + 64   ] = A[ global_index1 + 512
+ 64   ] + A[ global_index2 + 512 + 64   ];          lcl[ lidx + 512 + 128 ] = A[ global_index1 + 512 + 128 ] + A[ global_index2 + 512 + 128 ];     lcl[ lidx + 512 + 192 ] = A[
global_index1 + 512 + 192 ] + A[ global_index2 + 512 + 192 ];   lcl[ lidx + 512 + 256 ] = A[ global_index1 + 512 + 256 ] + A[ global_index2 + 512 + 256 ];     lcl[ lidx + 512 + 320
] = A[ global_index1 + 512 + 320 ] + A[ global_index2 + 512 + 320 ];        lcl[ lidx + 512 + 384 ] = A[ global_index1 + 512 + 384 ] + A[ global_index2 + 512 + 384 ];     lcl[ lidx +
512 + 448 ] = A[ global_index1 + 512 + 448 ] + A[ global_index2 + 512 + 448 ];                  write_mem_fence( CLK_LOCAL_MEM_FENCE );       // Completely remove the for
loops.     // 512      lcl[ lidx + 0     ] += lcl[ lidx + 512 + 0    ];             lcl[ lidx + 64    ] += lcl[ lidx + 512 + 64   ];          lcl[ lidx + 128   ] += lcl[ lidx + 512 + 128 ];         lcl[ lidx + 192   ] +=
lcl[ lidx + 512 + 192 ];          lcl[ lidx + 256   ] += lcl[ lidx + 512 + 256 ];         lcl[ lidx + 320   ] += lcl[ lidx + 512 + 320 ];         lcl[ lidx + 384   ] += lcl[ lidx + 512 + 384 ];          lcl[ lidx +
448   ] += lcl[ lidx + 512 + 448 ];         barrier( CLK_LOCAL_MEM_FENCE );           // 256      lcl[ lidx + 0     ] += lcl[ lidx + 256    + 0    ];            lcl[ lidx + 64    ] += lcl[ lidx + 256    + 64
];            lcl[ lidx + 128   ] += lcl[ lidx + 256    + 128 ];       lcl[ lidx + 192   ] += lcl[ lidx + 256    + 192 ];       barrier( CLK_LOCAL_MEM_FENCE );          // 128      lcl[ lidx + 0     ] += lcl
[ lidx + 128    + 0    ];             lcl[ lidx + 64    ] += lcl[ lidx + 128    + 64   ];        barrier( CLK_LOCAL_MEM_FENCE );           // 64        lcl[ lidx + 0     ] += lcl[ lidx + 64    + 0    ];
barrier( CLK_LOCAL_MEM_FENCE );           // do the last iterations from 64 to 1   if (lidx < 32) lcl[ lidx ] += lcl[ lidx + 32 ];            if (lidx < 16) lcl[ lidx ] += lcl[ lidx + 16 ];           if
(lidx < 8 ) lcl[ lidx ] += lcl[ lidx + 8 ];   if (lidx < 4 ) lcl[ lidx ] += lcl[ lidx + 4 ];            if (lidx < 2 ) lcl[ lidx ] += lcl[ lidx + 2 ];            if (lidx < 1 ) lcl[ lidx ] += lcl[ lidx + 1 ];           if
(lidx == 0)                      B[ idx ] = lcl[0];}
```

Version 4 - profiling

Terminal — ssh — 80×24

```
marcus@autana:~/code/oclReduction$ ./oclReduction -n 4
Using Kernel 'reduce4'
Number of elements: 33554432

Will use this device: GeForce GTX 260

Test PASSED!

GPU Memory IO: 9.229e+06 T memops/sec
GPU Memory bandwidth: 36.916 GB/s
T kernels: 1.82054 ms
T gpu: 2.3342 ms
T ref: 46.3934 ms
Speedup: 19.8755
marcus@autana:~/code/oclReduction$
```

- Kernel time = 2.02 % of total GPU time
- Memory copy time = 96.7 % of total GPU time
- Kernel taking maximum time = **reduce4** (2.0% of total GPU time)
- Memory copy taking maximum time = **memcpyHtoDasync** (96.7% of total GPU time)
- There is **no time overlap** between memory copies and kernels on GPU

Occupancy Analysis for kernel reduce4 on device GeForce GTX 260

- Kernel details: Grid size: [16384 1 1], Block size: [64 1 1]
- Register Ratio: 0.09375 (1536 / 16384) [8 registers per thread]
- Shared Memory Ratio: 0.84375 (13824 / 16384) [4124 bytes per Block]
- Active Blocks per SM: 3 (Maximum Active Blocks per SM: 8)
- Active threads per SM: 192 (Maximum Active threads per SM: 1024)
- Potential Occupancy: 0.1875 (6 / 32)
- Achieved occupancy: 0.1875 (on 27 SMs)
- Occupancy limiting factor: Block-Size

Comparison

kernel	t gpu	t kernel	speedup
0	7.682 ms	6.376 ms	6.14
1	5.873 ms	4.088 ms	7.93
2	5.795 ms	5.239 ms	8.65
3	2.556 ms	2.019 ms	18.19
4	2.334 ms	1.820 ms	19.87

Version 5

- We are using only 64 work items per workgroup, and only 3 groups are active per SM. This leads to only 192 threads of maximum of 1024. However, there is no problem to increase the number of work items, as the limiting factor is the local memory used.
- So let's increase the number of work items to 128 and see what happens

Version 5

```
__kernel void reduce5(__const int size, __global int *A, __global int *B) {     __local int lcl[1024];     unsigned int lidx
= get_local_id(0);unsigned int idx = get_group_id(0);unsigned int step = get_local_size(0);     // Prefetch  int
global_index1 = idx*1024 + lidx;     int global_index2 = idx*1024 + lidx + size;        lcl[ lidx + 0   ] = A[ global_index1 +
0   ] + A[ global_index2 + 0   ];        lcl[ lidx + 128 ] = A[ global_index1 + 128 ] + A[ global_index2 + 128 ];  lcl[ lidx +
256 ] = A[ global_index1 + 256 ] + A[ global_index2 + 256 ];        lcl[ lidx + 384 ] = A[ global_index1 + 384 ] + A[
global_index2 + 384 ];  lcl[ lidx + 512 ] = A[ global_index1 + 512 ] + A[ global_index2 + 512 ];  lcl[ lidx + 512 + 128 ] =
A[ global_index1 + 512 + 128 ] + A[ global_index2 + 512 + 128 ];        lcl[ lidx + 512 + 256 ] = A[ global_index1 + 512
+ 256 ] + A[ global_index2 + 512 + 256 ];        lcl[ lidx + 512 + 384 ] = A[ global_index1 + 512 + 384 ] + A[
global_index2 + 512 + 384 ];         write_mem_fence( CLK_LOCAL_MEM_FENCE );    // Completely remove the for
loops.        // 512lcl[ lidx + 0    ] += lcl[ lidx + 512 + 0   ];     lcl[ lidx + 128  ] += lcl[ lidx + 512 + 128 ]; lcl[ lidx + 256  ]
+= lcl[ lidx + 512 + 256 ];       lcl[ lidx + 384  ] += lcl[ lidx + 512 + 384 ]; barrier( CLK_LOCAL_MEM_FENCE );    // 256
        lcl[ lidx + 0    ] += lcl[ lidx + 256   + 0   ];  lcl[ lidx + 128  ] += lcl[ lidx + 256   + 128 ];       barrier(
CLK_LOCAL_MEM_FENCE );      // 128lcl[ lidx + 0    ] += lcl[ lidx + 128   + 0   ];  barrier(
CLK_LOCAL_MEM_FENCE );      // do the last iterations from 128 to 1       if (lidx < 64) lcl[ lidx ] += lcl[ lidx + 64 ];
barrier( CLK_LOCAL_MEM_FENCE );    if (lidx < 32) lcl[ lidx ] += lcl[ lidx + 32 ];    if (lidx < 16) lcl[ lidx ] += lcl[ lidx +
16 ];  if (lidx < 8 ) lcl[ lidx ] += lcl[ lidx + 8 ];       if (lidx < 4 ) lcl[ lidx ] += lcl[ lidx + 4 ];       if (lidx < 2 ) lcl[ lidx ] += lcl[
lidx + 2 ];    if (lidx < 1 ) lcl[ lidx ] += lcl[ lidx + 1 ];       if (lidx == 0)              B[ idx ] = lcl[0];}
```

Version 5 - profiling

Terminal — ssh — 80×24

```
marcus@autana:~/code/oclReduction$ ./oclReduction -n 5
Using Kernel 'reduce5'
Number of elements: 33554432

Will use this device: GeForce GTX 260

Test PASSED!

GPU Memory IO: 1.10236e+07 T memops/sec
GPU Memory bandwidth: 44.0946 GB/s
T kernels: 1.52416 ms
T gpu: 2.10262 ms
T ref: 47.1666 ms
Speedup: 22.4324
marcus@autana:~/code/oclReduction$
```

- Kernel time = 1.79 % of total GPU time
- Memory copy time = 96.9 % of total GPU time
- Kernel taking maximum time = **reduce5** (1.8% of total GPU time)
- Memory copy taking maximum time = **memcpyHtoDasync** (96.9% of total GPU time)
- There is **no time overlap** between memory copies and kernels on GPU

Occupancy Analysis for kernel reduce5 on device GeForce GTX 260

- Kernel details: Grid size: [16384 1 1], Block size: [128 1 1]

- Register Ratio: 0.28125 (4608 / 16384) [10 registers per thread]
- Shared Memory Ratio: 0.84375 (13824 / 16384) [4124 bytes per Block]

- Active Blocks per SM: 3 (Maximum Active Blocks per SM: 8)
- Active threads per SM: 384 (Maximum Active threads per SM: 1024)

- Potential Occupancy: 0.375 (12 / 32)
- Achieved occupancy: 0.375 (on 27 SMs)

- Occupancy limiting factor: Shared-memory

Version 6

- The profiler says the limiting factor is the occupancy
- Let's reduce the block size in order to increase the occupancy.
    - reduce min_size from 1024 to 512

Version 6

```
__kernel void reduce6(__const int size, __global int *A, __global int *B) {    __local int lcl[512];
unsigned int lidx = get_local_id(0);          unsigned int idx = get_group_id(0);         unsigned int step =
get_local_size(0);    // Prefetch      int global_index1 = idx*512 + lidx; int global_index2 = idx*512 + lidx
+ size; lcl[ lidx + 0   ] = A[ global_index1 + 0   ] + A[ global_index2 + 0   ];     lcl[ lidx + 128 ] = A[
global_index1 + 128 ] + A[ global_index2 + 128 ];         lcl[ lidx + 256 ] = A[ global_index1 + 256 ] + A[
global_index2 + 256 ];        lcl[ lidx + 384 ] = A[ global_index1 + 384 ] + A[ global_index2 + 384 ];
                    write_mem_fence( CLK_LOCAL_MEM_FENCE );       // Completely remove the for
loops. // 256  lcl[ lidx + 0    ] += lcl[ lidx + 256   + 0   ];   lcl[ lidx + 128  ] += lcl[ lidx + 256   + 128 ];
          barrier( CLK_LOCAL_MEM_FENCE );    // 128  lcl[ lidx + 0    ] += lcl[ lidx + 128   + 0   ];
barrier( CLK_LOCAL_MEM_FENCE );    // do the last iterations from 128 to 1       if (lidx < 64) lcl[ lidx
] += lcl[ lidx + 64 ]; barrier( CLK_LOCAL_MEM_FENCE );      if (lidx < 32) lcl[ lidx ] += lcl[ lidx + 32 ];
          if (lidx < 16) lcl[ lidx ] += lcl[ lidx + 16 ];    if (lidx < 8 ) lcl[ lidx ] += lcl[ lidx + 8 ];       if (lidx < 4 )
lcl[ lidx ] += lcl[ lidx + 4 ];    if (lidx < 2 ) lcl[ lidx ] += lcl[ lidx + 2 ];       if (lidx < 1 ) lcl[ lidx ] += lcl[ lidx +
1 ];      if (lidx == 0)             B[ idx ] = lcl[0];}
```

Version 6 - profiling

Terminal — ssh — 80×24

```
marcus@autana:~/code/oclReduction$ ./oclReduction -n 6
Using Kernel 'reduce6'
Number of elements: 33554432

Will use this device: GeForce GTX 260

Test PASSED!

GPU Memory IO: 1.36073e+07 T memops/sec
GPU Memory bandwidth: 54.429 GB/s
T kernels: 1.23658 ms
T gpu: 1.73764 ms
T ref: 47.0521 ms
Speedup: 27.0781
marcus@autana:~/code/oclReduction$
```

- Kernel time = 1.42 % of total GPU time
- Memory copy time = 97.3 % of total GPU time
- Kernel taking maximum time = **reduce6** (1.4% of total GPU time)
- Memory copy taking maximum time = **memcpyHtoDasync** (97.3% of total GPU time)
- There is **no time overlap** between memory copies and kernels on GPU

Occupancy Analysis for kernel reduce6 on device GeForce GTX 260

- Kernel details: Grid size: [32768 1 1], Block size: [128 1 1]

- Register Ratio: 0.5625 (9216 / 16384) [9 registers per thread]
- Shared Memory Ratio: 0.9375 (15360 / 16384) [2076 bytes per Block]

- Active Blocks per SM: 6 (Maximum Active Blocks per SM: 8)
- Active threads per SM: 768 (Maximum Active threads per SM: 1024)

- Potential Occupancy: 0.75 (24 / 32)
- Achieved occupancy: 0.75 (on 27 SMs)

- Occupancy limiting factor: Shared-memory

Final remarks

- We are comparing against a dummy in the CPU. A good CPU code can be 10x faster, leaving only a benefit of 2x - 3x of the GPU.
- This is only for big sets. Modify the code to plot variable data sizes.
- Optimizing code is system wide: depends on the host and the GPU
- A good optimization is also to know when to stop. Readable, modifiable code has value