Multi-Reduction: an Efficient Pattern for Simultaneous Independent Reductions on the GPU.

Christopher Parker
http://www.github.com/csp256

Computational Physics
University of Oslo
Norway

Allan MacKinnon
http://www.linkedin.com/in/allanmackinnon

PIXEL I/O, LLC
Seattle, WA

Abstract

We describe a flexible, warp-centric algorithm which permits multiple independent reduction operations to be efficiently performed in parallel with CUDA's $\_\_shfl\_xor()$ intrinsic on NVIDIA GPUs. Our technique can also be applied to both Intel IGP GPUs[3] and more recent AMD GCN GPUs[8]. We see significant performance gains by decreasing the number of instructions for the reduction by up to 60% and mitigating sequential data dependency further than previous best practices. We also provide a modification of this algorithm for platforms without warp shuffle operations, such as OpenCL and legacy NVIDIA hardware, as well as a variant for non-commutative operators. A use case is considered and benchmarked. Code is available online at: github.com/csp256/MultiReduction

1 Introduction

A core operation in parallel computing is to *reduce* a list of variables with an operator to yield a single value: $x_{0,n} = x_0 \circ x_1 \circ x_2 \circ \cdots \circ x_n$, where we use the notation $x_{i,j}$ to indicate a reduction of the variables on the interval $[i, j]$. This computational pattern is commonly used to compute sums, products, and max (min) values of a list. Note that while the reduction operator is not necessarily commutative or associative, in practice it is usually both and we restrict ourselves to this case except where noted. We consider addition as our specific associative, commutative reduction operator of interest without further loss of generality.

Assuming that such a reduction must be performed at the warp level, we can implement this simply as [5]:

```
for (int i=1; i<32; i<<=1) x += __shfl_xor(x, i);
```

We remind the reader that $\_\_shfl\_xor(x,i)$ returns the value of x held not by this thread, but by the thread number which equals i XOR this thread's number. Thus $i == 1$ causes adjacent threads to exchange their values of x. We implicitly assume all blocks are 32 threads wide here and throughout this document.

The above code snippet assumes that x is a single variable in a warp of 32 threads[1], and is thus represented by 32 values in 32 separate registers. Before the reduction operation each register holds a single meaningful value corresponding to the input data. After the reduction operation, all 32 registers hold the same value: the sum of all the input data (denoted as $x_{0,31}$). We can visualize this process:

0	1	2	3	4	5	6	7	8	⋯	31	Operation
x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	⋯	x_{31}	
$x_{0,1}$	$x_{0,1}$	$x_{2,3}$	$x_{2,3}$	$x_{4,5}$	$x_{4,5}$	$x_{6,7}$	$x_{6,7}$	$x_{8,9}$	⋯	$x_{30,31}$	$x+=\_\_shfl\_xor(x,1);$
$x_{0,3}$	$x_{0,3}$	$x_{0,3}$	$x_{0,3}$	$x_{4,7}$	$x_{4,7}$	$x_{4,7}$	$x_{4,7}$	$x_{8,11}$	⋯	$x_{28,31}$	$x+=\_\_shfl\_xor(x,2);$
$x_{0,7}$	$x_{0,7}$	$x_{0,7}$	$x_{0,7}$	$x_{0,7}$	$x_{0,7}$	$x_{0,7}$	$x_{0,7}$	$x_{8,15}$	⋯	$x_{24,31}$	$x+=\_\_shfl\_xor(x,4);$
$x_{0,15}$	$x_{0,15}$	$x_{0,15}$	$x_{0,15}$	$x_{0,15}$	$x_{0,15}$	$x_{0,15}$	$x_{0,15}$	$x_{0,15}$	⋯	$x_{16,31}$	$x+=\_\_shfl\_xor(x,8);$
$x_{0,31}$	$x_{0,31}$	$x_{0,31}$	$x_{0,31}$	$x_{0,31}$	$x_{0,31}$	$x_{0,31}$	$x_{0,31}$	$x_{0,31}$	⋯	$x_{0,31}$	$x+=\_\_shfl\_xor(x,16);$

The first shuffle-and-reduce operation computes 16 unique values (2 copies each) from 32 unique inputs. The second shuffle-reduce operation computes 8 unique values (4 copies each) from 16 unique inputs. This process quickly becomes increasingly wasteful of the available parallelism. The final shuffle-and-reduce operation computes only a single unique value, but it simultaneously does so 32 times in parallel. The total cost is 5 sequential shuffle-reduction operations.

[1] The CUDA Programming Guide warns that the number of threads in a warp is liable to change. This possibility would have no significant impact on the algorithms presented here.

2 Multi-Reduction

This waste of parallelism can be mitigated if we have multiple independent reductions to be performed in the same warp with the same reduction operator. Assume that we have two variables, a and b, in 32 threads, using total of 64 registers. Simply repeating the naive reduction operation results in twice as many (10) shuffle-reduction operations being required, though instruction level parallelism is improved:

```
for (int i=1; i<32; i<<=1) {
   a += __shfl_xor(a, i);
   b += __shfl_xor(b, i);
}
```

We can again create a table to visualizes this process, though this time we denote the variable of interest in the left column (*):

*	Thread Number											Operation
	0	1	2	3	4	5	6	7	8	⋯	31	
a	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	⋯	a_{31}	
b	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	⋯	b_{31}	
a	$a_{0,1}$	$a_{0,1}$	$a_{2,3}$	$a_{2,3}$	$a_{4,5}$	$a_{4,5}$	$a_{6,7}$	$a_{6,7}$	$a_{8,9}$	⋯	$a_{30,31}$	$a+=\_\_shfl\_xor(a,1);$
b	$b_{0,1}$	$b_{0,1}$	$b_{2,3}$	$b_{2,3}$	$b_{4,5}$	$b_{4,5}$	$b_{6,7}$	$b_{6,7}$	$b_{8,9}$	⋯	$b_{30,31}$	$b+=\_\_shfl\_xor(b,1);$
a	$a_{0,3}$	$a_{0,3}$	$a_{0,3}$	$a_{0,3}$	$a_{4,7}$	$a_{4,7}$	$a_{4,7}$	$a_{4,7}$	$a_{8,11}$	⋯	$a_{28,31}$	$a+=\_\_shfl\_xor(a,2);$
b	$b_{0,3}$	$b_{0,3}$	$b_{0,3}$	$b_{0,3}$	$b_{4,7}$	$b_{4,7}$	$b_{4,7}$	$b_{4,7}$	$b_{8,11}$	⋯	$b_{28,31}$	$b+=\_\_shfl\_xor(b,2);$
a	$a_{0,7}$	$a_{0,7}$	$a_{0,7}$	$a_{0,7}$	$a_{0,7}$	$a_{0,7}$	$a_{0,7}$	$a_{0,7}$	$a_{8,15}$	⋯	$a_{24,31}$	$a+=\_\_shfl\_xor(a,4);$
b	$b_{0,7}$	$b_{0,7}$	$b_{0,7}$	$b_{0,7}$	$b_{0,7}$	$b_{0,7}$	$b_{0,7}$	$b_{0,7}$	$b_{8,15}$	⋯	$b_{24,31}$	$b+=\_\_shfl\_xor(b,4);$
a	$a_{0,15}$	$a_{0,15}$	$a_{0,15}$	$a_{0,15}$	$a_{0,15}$	$a_{0,15}$	$a_{0,15}$	$a_{0,15}$	$a_{0,15}$	⋯	$a_{16,31}$	$a+=\_\_shfl\_xor(a,8);$
b	$b_{0,15}$	$b_{0,15}$	$b_{0,15}$	$b_{0,15}$	$b_{0,15}$	$b_{0,15}$	$b_{0,15}$	$b_{0,15}$	$b_{0,15}$	⋯	$b_{16,31}$	$b+=\_\_shfl\_xor(b,8);$
a	$a_{0,31}$	$a_{0,31}$	$a_{0,31}$	$a_{0,31}$	$a_{0,31}$	$a_{0,31}$	$a_{0,31}$	$a_{0,31}$	$a_{0,31}$	⋯	$a_{0,31}$	$a+=\_\_shfl\_xor(a,16);$
b	$b_{0,31}$	$b_{0,31}$	$b_{0,31}$	$b_{0,31}$	$b_{0,31}$	$b_{0,31}$	$b_{0,31}$	$b_{0,31}$	$b_{0,31}$	⋯	$b_{0,31}$	$b+=\_\_shfl\_xor(b,16);$

This has done nothing to reduce the computational load, and suffers from the same problem of redundant work. However, if we instead merge the two logical variables into a single variable, by predicating on the least significant bit of the lane number (denoted here by simply *threadIdx.x*), then the subsequent computational load is reduced by half. Note that this means that the variable 'a' holds the value of either $a_{i,j}$ or $b_{i,j}$, depending on if the thread is odd or even.

```
a += __shfl_xor(a, 1);
b += __shfl_xor(b, 1);
if (threadIdx.x&1) a = b;
for (int i=2; i<32; i<<=1) {
    a += __shfl_xor(a, i);
}
```

The table showing the changing values of each thread's variables helps make this optimization clear:

*	Thread Number											Operation
	0	1	2	3	4	5	6	7	8	⋯	31	
a	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	⋯	a_{31}	
b	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	⋯	b_{31}	
a	$a_{0,1}$	$a_{0,1}$	$a_{2,3}$	$a_{2,3}$	$a_{4,5}$	$a_{4,5}$	$a_{6,7}$	$a_{6,7}$	$a_{8,9}$	⋯	$a_{30,31}$	$a+=\_\_shfl\_xor(a,1);$
b	$b_{0,1}$	$b_{0,1}$	$b_{2,3}$	$b_{2,3}$	$b_{4,5}$	$b_{4,5}$	$b_{6,7}$	$b_{6,7}$	$b_{8,9}$	⋯	$b_{30,31}$	$b+=\_\_shfl\_xor(b,1);$
a	$a_{0,1}$	$b_{0,1}$	$a_{2,3}$	$b_{2,3}$	$a_{4,5}$	$b_{4,5}$	$a_{6,7}$	$b_{6,7}$	$a_{8,9}$	⋯	$b_{30,31}$	$if\ (threadIdx.x\ \&\ 1)\ a=b;$
a	$a_{0,3}$	$b_{0,3}$	$a_{0,3}$	$b_{0,3}$	$a_{4,7}$	$b_{4,7}$	$a_{4,7}$	$b_{4,7}$	$a_{8,11}$	⋯	$b_{28,31}$	$a+=\_\_shfl\_xor(a,2);$
a	$a_{0,7}$	$b_{0,7}$	$a_{0,7}$	$b_{0,7}$	$a_{0,7}$	$b_{0,7}$	$a_{0,7}$	$b_{0,7}$	$a_{8,15}$	⋯	$b_{24,31}$	$a+=\_\_shfl\_xor(a,4);$
a	$a_{0,15}$	$b_{0,15}$	$a_{0,15}$	$b_{0,15}$	$a_{0,15}$	$b_{0,15}$	$a_{0,15}$	$b_{0,15}$	$a_{0,15}$	⋯	$b_{16,31}$	$a+=\_\_shfl\_xor(a,8);$
a	$a_{0,31}$	$b_{0,31}$	$a_{0,31}$	$b_{0,31}$	$a_{0,31}$	$b_{0,31}$	$a_{0,31}$	$b_{0,31}$	$a_{0,31}$	⋯	$a_{0,31}$	$a+=\_\_shfl\_xor(a,16);$

With the small overhead of only a single conditional move, the number of shuffle-reductions necessary has decreased from 10 to 6. The even threads contain the result of one reduction, while the odd threads contain the result of the other. Afterward the result is held in the same register on N consecutive threads, and repeats 32/N times.

This can be extended to four independent reductions as such (also, see figure 1):

```
a += __shfl_xor(a, 1);
b += __shfl_xor(b, 1);
if (threadIdx.x&1) a = b;
c += __shfl_xor(c, 1);
d += __shfl_xor(d, 1);
if (threadIdx.x&1) c = d;

a += __shfl_xor(a, 2);
c += __shfl_xor(c, 2);
if (threadIdx.x&2) a = c;

for (int i=4; i<32; i<<=1) {
    a += __shfl_xor(a, i);
}
```

*	Thread Number											Operation
	0	1	2	3	4	5	6	7	8	⋯	31	
a	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	⋯	a_{31}	
b	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	⋯	b_{31}	
c	c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	⋯	c_{31}	
d	d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	⋯	d_{31}	
a	$a_{0,1}$	$a_{0,1}$	$a_{2,3}$	$a_{2,3}$	$a_{4,5}$	$a_{4,5}$	$a_{6,7}$	$a_{6,7}$	$a_{8,9}$	⋯	$a_{30,31}$	$a+=\_\_shfl\_xor(a,1);$
b	$b_{0,1}$	$b_{0,1}$	$b_{2,3}$	$b_{2,3}$	$b_{4,5}$	$b_{4,5}$	$b_{6,7}$	$b_{6,7}$	$b_{8,9}$	⋯	$b_{30,31}$	$b+=\_\_shfl\_xor(b,1);$
a	$a_{0,1}$	$b_{0,1}$	$a_{2,3}$	$b_{2,3}$	$a_{4,5}$	$b_{4,5}$	$a_{6,7}$	$b_{6,7}$	$a_{8,9}$	⋯	$b_{30,31}$	$if\ (threadIdx.x\ \&\ 1)\ a=b;$
c	$c_{0,1}$	$c_{0,1}$	$c_{2,3}$	$c_{2,3}$	$c_{4,5}$	$c_{4,5}$	$c_{6,7}$	$c_{6,7}$	$c_{8,9}$	⋯	$c_{30,31}$	$c+=\_\_shfl\_xor(c,1);$
d	$d_{0,1}$	$d_{0,1}$	$d_{2,3}$	$d_{2,3}$	$d_{4,5}$	$d_{4,5}$	$d_{6,7}$	$d_{6,7}$	$d_{8,9}$	⋯	$d_{30,31}$	$d+=\_\_shfl\_xor(d,1);$
c	$c_{0,1}$	$d_{0,1}$	$c_{2,3}$	$d_{2,3}$	$c_{4,5}$	$d_{4,5}$	$c_{6,7}$	$d_{6,7}$	$c_{8,9}$	⋯	$d_{30,31}$	$if\ (threadIdx.x\ \&\ 1)\ c=d;$
a	$a_{0,3}$	$b_{0,3}$	$a_{0,3}$	$b_{0,3}$	$a_{4,7}$	$b_{4,7}$	$a_{4,7}$	$b_{4,7}$	$a_{8,11}$	⋯	$b_{28,31}$	$a+=\_\_shfl\_xor(a,2);$
c	$c_{0,3}$	$d_{0,3}$	$c_{0,3}$	$d_{0,3}$	$c_{4,7}$	$d_{4,7}$	$c_{4,7}$	$d_{4,7}$	$c_{8,11}$	⋯	$d_{28,31}$	$c+=\_\_shfl\_xor(c,2);$
a	$a_{0,3}$	$b_{0,3}$	$c_{0,3}$	$d_{0,3}$	$a_{4,7}$	$b_{4,7}$	$c_{4,7}$	$d_{4,7}$	$a_{8,11}$	⋯	$d_{28,31}$	$if\ (threadIdx.x\ \&\ 2)\ a=c;$
a	$a_{0,7}$	$b_{0,7}$	$c_{0,7}$	$d_{0,7}$	$a_{0,7}$	$b_{0,7}$	$c_{0,7}$	$d_{0,7}$	$a_{8,15}$	⋯	$d_{24,31}$	$a+=\_\_shfl\_xor(a,4);$
a	$a_{0,15}$	$b_{0,15}$	$c_{0,15}$	$d_{0,15}$	$a_{0,15}$	$b_{0,15}$	$c_{0,15}$	$d_{0,15}$	$a_{0,15}$	⋯	$d_{16,31}$	$a+=\_\_shfl\_xor(a,8);$
a	$a_{0,31}$	$b_{0,31}$	$c_{0,31}$	$d_{0,31}$	$a_{0,31}$	$b_{0,31}$	$c_{0,31}$	$d_{0,31}$	$a_{0,31}$	⋯	$d_{0,31}$	$a+=\_\_shfl\_xor(a,16);$

This shows that only 3 merges and 9 shuffle-reductions are necessary to reduce 4 variables. Note that while the efficiency has improved, the amount of improvement has decreased.

To compute 32 reductions would cost only 62 shuffle-reductions and 31 merges, while the naive approach requires 160 shuffle-reductions.

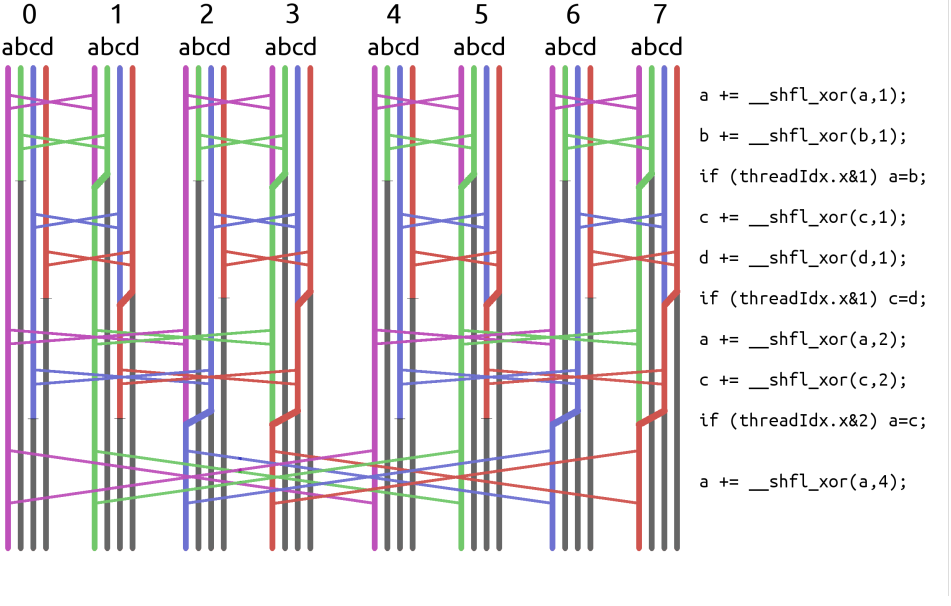

Figure 1: A multireduction is performed on 4 variables (a, b, c, d) across 8 threads (8 is used instead of 32 only for illustrative purposes). After the multireduction has concluded, variable a holds the results of each reduction on different threads. Reduction a is available on threads 0 and 1, reduction b is on threads 4 and 5, etc. After the multireduction has been carried out variables b, c, and d hold partial results which can be neglected (indicated by gray). Two additional shuffle-reductions would be necessary to perform the same task on 32 threads.

3 Iterative Multi-Reduction

With only a few registers of overhead, we can apply the multireduction pattern online.

```
register float r[6];
#pragma unroll
for (int i=0; i<32; i++) {
    r[0] = func(i);
    r[0] += __shfl_xor(r[0], 1);
    if ((i&1) == 0) {
        r[1] = r[0];
    } else {
        if (threadIdx.x & 1) {
            r[1] = r[0];
        }
        r[1] += __shfl_xor(r[1], 2);
        if ((i&2) == 0) {
            r[2] = r[1];
        } else {
            if (threadIdx.x & 2) {
                r[2] = r[1];
            }
            r[2] += __shfl_xor(r[2], 4);
            if ((i&4) == 0) {
                r[3] = r[2];
            } else {
                if (threadIdx.x & 4) {
                    r[3] = r[2];
                }
                r[3] += __shfl_xor(r[3], 8);
                if ((i&8) == 0) {
                    r[4] = r[3];
                } else {
                    if (threadIdx.x & 8) {
                        r[4] = r[3];
                    }
                    r[4] += __shfl_xor(r[4], 16);
                    if ((i&16) == 0) {
                        r[5] = r[4];
                    } else {
                        if (threadIdx.x & 16) {
                            r[5] = r[4];
                        }
                    }
                }
            }
        }
    }
}
// At this point the sum of func(N) is in thread N, where N is 0..31
```

That is to say, the above code efficiently solves problems of the form:

```
for (int i=0; i<32; i++) {
    float f = function(i, threadIdx.x);
    // some partial reduction process, dependent upon i
}
// At this point, all 32 threads have a unique reduction available.
// Specifically, thread i has the reduced value of the ith inner ↩
    loop iteration above.
```

Note that the above conditional statements are all constant at compile-time and exist in a fully unrolled loop, so there will be no branching at run-time. Of course, there are still predicated assignments based on the lane number.

Furthermore, the verbose code given above may be replaced by a relatively concise loop ostensibly without overhead. However, preliminary testing has show this to cause poorly optimized machine code to be generated on some compilers: the same code has half the speed using CUDA 7.5 than CUDA 8. It should be noted that on CUDA 8 the unrolled form given above is marginally slower than a more concise loop-based version. The reader is encouraged to profile closely and verify their expectations experimentally on performance critical kernels.

This technique requires 6 extra registers, but this number can be decreased if necessary by removing the inner parts of the above conditional and 'falling back' to the typical reduction method. This causes an increase in memory dependency stalls and number of operations, but can be crucial under significant register pressure. Note, that the efficiency loss per register removed in this manner is increasing: using only 5 registers results in only 1 more shuffle-reduction per 32 reduction operations. Conversely, additional registers are able to be utilized by the compiler effectively, and provide somewhat smooth scaling performance benefit.

Similar optimizations can be made, but without penalty, if each reduction operation to be performed spans only a portion of the warp. For example, if each consecutive quarter warp holds a sequence of values to be reduced, each independent (including across quarter-warp boundaries), then only 4 extra registers are needed. This is a savings of 1 register per subdivision of the full warp size.

The iterative multireduction pattern as presented above suffers from lack of instruction level parallelism. This can be easily added by "doubling up" the pattern so it operates on two independent sets of 6 registers and producers: after every line of code operating on r, add a nearly line of code instead operating on a new register array s. This was found to have a significant performance benefit in some cases, and is the technique implemented in our benchmark.

4 Swapping

In principle the multireduction pattern can reduce the number of shuffle-reductions necessary by a factor of two by using a predicated swap. The below example shows the swap-based multireduction pattern for 2 reductions.

```
if (threadIdx.x & 1) swap(a, b); // Assume swap() is a macro
a += __shfl_xor(b, 1);
for (int i=2; i<32; i<<=1) a += __shfl_xor(a, i);
```

*	Thread Number											Operation
	0	1	2	3	4	5	6	7	8	⋯	31	
a	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	⋯	a_{31}	
b	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	⋯	b_{31}	
a	a_0	b_1	a_2	b_3	a_4	b_5	a_6	b_7	a_8	⋯	b_{31}	$if\ (threadIdx.x\ \&\ 1)\ swap(a,b);$
b	b_0	a_1	b_2	a_3	b_4	a_5	b_6	a_7	b_8	⋯	a_{31}	Same as above.
a	$a_{0,1}$	$b_{0,1}$	$a_{2,3}$	$b_{2,3}$	$a_{4,5}$	$b_{4,5}$	$a_{6,7}$	$b_{6,7}$	$a_{8,9}$	⋯	$b_{30,31}$	$a+=\_\_shfl\_xor(b,1);$
a	$a_{0,3}$	$b_{0,3}$	$a_{0,3}$	$b_{0,3}$	$a_{4,7}$	$b_{4,7}$	$a_{4,7}$	$b_{4,7}$	$a_{8,11}$	⋯	$b_{28,31}$	$a+=\_\_shfl\_xor(a,2);$
a	$a_{0,7}$	$b_{0,7}$	$a_{0,7}$	$b_{0,7}$	$a_{0,7}$	$b_{0,7}$	$a_{0,7}$	$b_{0,7}$	$a_{8,15}$	⋯	$b_{24,31}$	$a+=\_\_shfl\_xor(a,4);$
a	$a_{0,15}$	$b_{0,15}$	$a_{0,15}$	$b_{0,15}$	$a_{0,15}$	$b_{0,15}$	$a_{0,15}$	$b_{0,15}$	$a_{0,15}$	⋯	$b_{16,31}$	$a+=\_\_shfl\_xor(a,8);$
a	$a_{0,31}$	$b_{0,31}$	$a_{0,31}$	$b_{0,31}$	$a_{0,31}$	$b_{0,31}$	$a_{0,31}$	$b_{0,31}$	$a_{0,31}$	⋯	$a_{0,31}$	$a+=\_\_shfl\_xor(a,16);$

However, no such swap operation is provided in hardware, and in practice implementing an equivalent implementation is slower than the above suggested method for both integer and floating point addition. However, if the reduction operator is particularly slow performing a swap might be more performant.

This method takes 31 conditional swaps and 31 shuffle-reductions to compute 32 reductions. The naive approach requires 160 shuffle-reductions.

5 Non-Commutative Operators

Non commutative operators on scalar data are somewhat rare. However, you can handle this case by replacing every occurance of the following code snippet in the iterative multireduction pattern (section 4) with the second code snippet. This has the effect of un-reversing the argument order of the reduction on every other group of 2^n threads. Care must of course be taken to adjust register indices and constant powers of two as appropriate.

```
r[0] += __shfl_xor(r[0], 1);
if ((i&1) == 0) {
```

Replace the above code with that on the following page.

```
if (i&1) {
    r[0] = __shfl_xor(r[0], 1) + r[0];
} else {
    r[0] += __shfl_xor(r[0])
}
if ((i&1) == 0) {
```

This replacement could be implemented by hand as such:

```
a += __shfl_xor(a, 1);
b = __shfl_xor(b, 1) + b;
if (threadIdx.x&1) a = b;
c += __shfl_xor(c, 1);
d = __shfl_xor(d, 1) + d;
if (threadIdx.x&1) c = d;

a += __shfl_xor(a, 2);
c = __shfl_xor(c, 2) + c;
if (threadIdx.x&2) a = c;

for (int i=4; i<32; i<<=1) {
    a += __shfl_xor(a, i);
}
// Only threads 0..3 hold valid results.
```

In the case where the reduction must be routed through shared memory, there are several options. The first round of the reduction might be performed like:

```
float r = b; // 'r' is scratch space
if (threadIdx.x & 1) {
    r = a;
    a = b;
}
s[threadIdx.x] = r;
r = s[threadIdx.x ^ 1];
b = r;
if (threadIdx.x & 1) {
    b = a;
    a = r;
}
a += b;
```

Note that two separate approaches are seen here: the first simply does two reduction operations at 50% efficiency each, while the above carefully rearranges the arguments so that a single reduction can be performed. The specifics of which method is favorable (ignoring issues of shared memory vs warp shuffle) depends upon how expensive the reduction operator is.

6 OpenCL & Shared Memory

The *__shfl()* instructions are not available when using OpenCL (except on Intel [3]) or older NVIDIA GPUs. However, it is still possible to use shared memory (referred to as local memory in OpenCL, not to be confused with CUDA's local memory) to perform a reduction. This requires a certain amount of shared memory per warp (typically 128 bytes to store 32 floats), and this requirement is doubled in our implementation. The increased shared memory usage can be removed if a swap is performed, and the number of shared memory writes halved, but this is less performant without significant register pressure. The shared memory variant of the multireduction pattern, for 4 data reductions, looks like this:

```
__shared__ float s[64];
s[threadIdx.x ] = a;
s[threadIdx.x+32] = b;
if (threadIdx.x&1) a = b;
a = a + s[(threadIdx.x ^ 1) | ((threadIdx.x&1)<<5)];
s[threadIdx.x ] = c;
s[threadIdx.x+32] = d;
if (threadIdx.x&1) c = d;
c = c + s[(threadIdx.x ^ 1) | ((threadIdx.x&1)<<5)];

s[threadIdx.x ] = a;
s[threadIdx.x+32] = c;
if (threadIdx.x&2) a = c;
a = a + s[(threadIdx.x ^ 2) | ((threadIdx.x&2)<<4)];

for (int i=4; i<32; i<<=1) {
    s[threadIdx.x] = a;
    a = a + s[threadIdx.x ^ i];
}
```

For 32 independent reductions 62 shared memory writes and 31 shared memory reads are required. Additionally, 31 predicated merges are required.

7 Benchmarks and Analysis

We compared the previous best practice reduction technique to our multireduction using an artificial benchmark: finding the sum of all consecutive subsets of length 32 in a long array. To emphasize the performance of the reduction operation we did this in the most naive way. Each subset was computed independently of each other directly from the input array. We investigated arrays of type `int`, `float`, and `double`. Code is available online: www.github.com/csp256/MultiReduction

Our particular choice of benchmarking task was necessary to prevent the kernel from becoming limited by global memory reads. By having such heavily overlapping subsets, we were better able to exploit the cache and thus reduce global memory bandwidth utilization

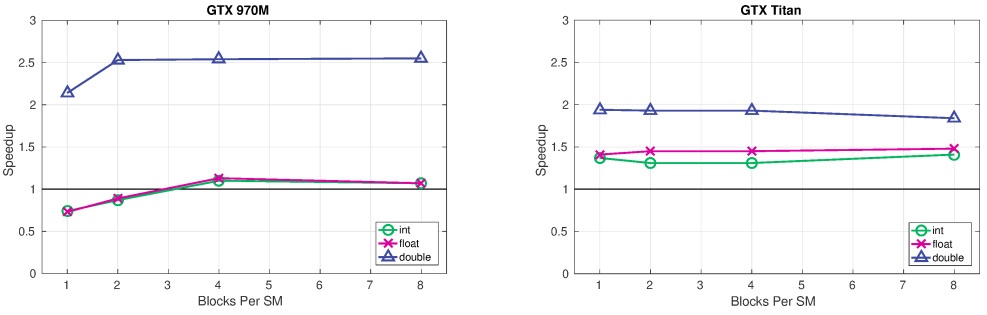

Figure 2: The multireduction pattern shows highest performance where it is needed most.

from 100% to 5%. Both kernels have been reasonably optimized are strongly limited by shuffle-reduction operations. Their results are validated on the CPU to ensure correctness.

Because we were only interested in relative, not absolute, performance between kernels we adapt our problem size such that a certain number of blocks are assigned per SM (maximum 8). Each block of 256 threads computes 16,384 = 2^{14} subsets (each subset containing 32 elements). Thus, when configured to launch 8 blocks per SM 131,072 = 2^{17} subset sums are computed per SM. Our GTX 970M and GTX Titan have 10 and 14 SMs respectively.

However, we note that especially poor compiler optimization on CUDA 7.5 forced us explicitly unroll the entire pattern. The program responsible for emitting the corresponding 344 lines of code is available online with the benchmark. The author has been told, but is not been able to verify, that CUDA 8 solves this issue. Similar unrolling was necessary in our kernel of comparison, specifically when preparing the results to be written out to global memory.

To time a kernel we first launched it 16 times to allow the GPU to (literally) warm up, and then immediately time 64 consecutive kernel launches using a high precision timer. Run times between 1 to 6 milliseconds (depending upon data type and hardware), with a variation near one percent.

Our kernel performed more consistent performance than the speedup alone implies. Integer and single precision floating point multireduction shows the same performance on both cards, however float performance drops about 10% on for the standard reduction technique on the Titan. Similarly, the decreased speedup or even slowdown at low utilizations was due to improved speed of the standard kernel, which gives evidence that the multireduction pattern can suffer resource contention more gracefully.

At maximum utilization both 32 bit types saw a speedup of 1.07 on the 970M, and 1.5 to 1.4 on the Titan.

However, the multireduction pattern excels when the shuffle-reduction operation is comparably slow, as is the case for doubles. Because only 32 bit values can be shuffled, doubles must be split, shuffled independently, and then stitched back together. Adding double precision numbers is half the speed of single precision numbers at best and often much worse

(hardware depending). When the more typical case of "much worse" is encountered on the 970M the multireduction pattern slightly exceeds its theoretically expected 2.5 speedup. The Titan has much better support for double precision arithmetic, so a somewhat more modest speedup of 1.9 is seen.

8 Use Case

A common, low level task in computer vision is to determine a correspondence between image pairs. This is commonly done only on a sparse set of salient points. An abstract representation of each point is computed. This is called a *descriptor* and takes the form of a 512 bit vector in our case. For a given descriptor in the first image, if its nearest neighbor in the second image is at least N bits more similar than the second nearest neighbor, then we establish a correspondence between the corresponding salient points. The following kernel establishes these correspondences, thus *matching* the descriptors.

```
__global__ void
#ifndef __INTELLISENSE__
__launch_bounds__(256, 0)
#endif
CUDAK2NN_kernel(const cudaTextureObject_t tex_q, const int num_q, ↩
    const uint64_t* const __restrict__ g_training, const int ↩
    num_t, int* const __restrict__ g_match, const int threshold) {
    uint64_t q[8];
    for (int i = 0, offset = ((threadIdx.x & 24) << 3) + ↩
        (threadIdx.x & 7) + (blockIdx.x << 11) + (threadIdx.y << ↩
        8); i < 8; ++i, offset += 8) {
        const uint2 buf = tex1Dfetch<uint2>(tex_q, offset);
        asm("mov.b64 %0, {%1,%2};" : "=l"(q[i]) : "r"(buf.x), ↩
            "r"(buf.y)); // some assembly required
    }
    int best_i, best_v = 100000, second_v = 200000;
#pragma unroll 7
    for (int t = 0; t < num_t; ++t) {
        const uint64_t train = g_training[(t << 3) + (threadIdx.x ↩
            & 7)];
        uint32_t dist[4];
        for (int i = 0; i < 4; ++i) dist[i] = ↩
            __byte_perm(__popcll(q[i] ^ train), __popcll(q[i + 4] ↩
            ^ train), 0x5410);
        for (int i = 0; i < 4; ++i) dist[i] += ↩
            __shfl_xor(dist[i], 1);
        if (threadIdx.x & 1) dist[0] = dist[1];
        if (threadIdx.x & 1) dist[2] = dist[3];
        dist[0] += __shfl_xor(dist[0], 2);
        dist[2] += __shfl_xor(dist[2], 2);
        if (threadIdx.x & 2) dist[0] = dist[2];
        dist[0] = __byte_perm(dist[0] + __shfl_xor(dist[0], 4), ↩
            0, threadIdx.x & 4 ? 0x5432 : 0x5410);
        if (dist[0] < second_v) second_v = dist[0];
```

```
        if (dist[0] < best_v) {
            second_v = best_v;
            best_i = t;
            best_v = dist[0];
        }
    }
    const int idx = (blockIdx.x << 8) + (threadIdx.y << 5) + ↩
        threadIdx.x;
    if (idx < num_q) g_match[idx] = second_v - best_v > threshold ↩
        ? best_i : -1;
}
```

On a GTX 970M extant GPU methods[1] of brute force matching descriptors of 256 bits have a peak throughput of 2.0 billion comparisons per second, while CPU methods can achieve 1.0 billion comparisons per second fast[6]. The above code reaches 18.6 billion comparisons per second for descriptors of twice the length on the same GPU. While the use of the multireduction pattern is not responsible for the entirety of this speedup, it does play a significant role. The performance scales well across hardware generations, as a GTX 760, GTX Titan, and GTX 1080 achieve 10, 20, and 64 billion comparisons per second.

We note that the performance difference is particularly stark because the given matcher kernel operates on descriptors (bitstrings) of twice the standard length, and thus requires twice as much computation, but remains an order of magnitude faster than previous methods. This is particularly important as automatic 3D reconstruction from image sets, referred to as the *structure from motion* task, may require several hours or days of computation on matching descriptors, even as part of a distributed system[2].

Similar, though only partially optimized, kernels were recently used by the author[7] to improve both the LATCH descriptor[4] extraction and matching speed by three and one orders of magnitude (respectively) relative to the best previous CPU and GPU implementations (respectively) by using the efficient multireduction pattern.

9 Conclusion

We have presented a flexible technique for optimizing a common task in high performance GPU computation, as well as explored several variations. This computational pattern, called multireduction, requires little to no resource overhead and improves upon previously extant best practices. We have provided source code, and shown a real-world speedup of an order of magnitude in performance-critical code, relative to other GPU implementations.

10 Acknowledgements

Kareem Omar kindly optimized the brute force matcher kernel. Further information and complete implementation is available: www.github.com/komrad36/CUDAK2NN

References

[1] Feature Detection and Description - OpenCV 2.4.13.1 documentation. URL http://docs.opencv.org/2.4/modules/gpu/doc/feature_detection_and_description.html#gpu-bruteforcematcher-gpu-base-knnmatch.

[2] Sameer Agarwal, Noah Snavely, Ian Simon, Steven M Seitz, and Richard Szeliski. Building rome in a day. In *2009 IEEE 12th international conference on computer vision*, pages 72–79. IEEE, 2009.

[3] Ben Ashbaugh, Allen Hux, Pranayini Gudali, Dawid Dominiak, and Biju George. Khronos Group. URL https://www.khronos.org/registry/cl/extensions/intel/cl_intel_subgroups.txt.

[4] Gil Levi and Tal Hassner. LATCH: learned arrangements of three patch codes. In *Winter Conference on Applications of Computer Vision (WACV)*. IEEE, 2016. URL http://www.openu.ac.il/home/hassner/projects/LATCH.

[5] Justin Luitjens. Faster parallel reductions on kepler. URL https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/.

[6] Kareem Omar. K2NN. URL https://github.com/komrad36/K2NN.

[7] Christopher Parker, Matthew Daiter, Kareem Omar, Gil Levi, and Tal Hassner. The CUDA LATCH binary descriptor: Because sometimes faster means better. In *European Conference on Computer Vision (ECCV) workshops*, 2016. URL http://www.openu.ac.il/home/hassner/projects/LATCH.

[8] Ben S and er. AMD GCN Assembly: Cross-Lane Operations, August 2016. URL http://gpuopen.com/amd-gcn-assembly-cross-lane-operations/.