Using OpenCL™ 2.0 Work-group Functions

Tutorial

Intel® SDK for OpenCL™ Applications - Tutorial

Legal Information

By using this document, in addition to any agreements you have with Intel, you accept the terms set forth below.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to:

http://www.intel.com/design/literature.htm.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. Go to: http://www.intel.com/products/processor_number/.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Intel, Intel logo, Intel Core, VTune, Xeon are trademarks of Intel Corporation in the U.S. and other countries.

* Other names and brands may be claimed as the property of others.

OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission from Khronos.

Microsoft product screen shot(s) reprinted with permission from Microsoft Corporation.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Among new OpenCL 2.0 features, several new and useful built-ins were introduced, called “work-group functions”. These built-ins provide popular parallel primitives that operate at the workgroup level. This article is a short introduction on work-group functions and their usage. It is also backed with some performance data gathered from Intel HD Graphics OpenCL device.

Work-group functions overview

For novice OpenCL programmers, the OpenCL 2.0 spec chapter dedicated to this topic might look too academic and full of formal notations. In short, work-group functions include three classic work-group level algorithms – **value broadcast**, **reduce**, and **scan**, plus two built-ins that evaluate boolean operation result over entire workgroup. **Reduce** and **scan** algorithms support **add**, **min** and **max** operations.

Functionality of work-group function built-ins is pretty obvious from their names:

- **work_group_broadcast()** built-in duplicates a value from the chosen workitem to all workitems of the workgroup
- **work_group_reduce()** group of built-ins evaluates **sum**, **min** or **max** among all items of workgroup, and then broadcasts it for every work item in the group
- **work_group_scan()** group of built-ins evaluates **sum**, **min** or **max** for all preceding work items, optionally including current
- **work_group_all()** returns logical AND over same boolean expression evaluated for every work-item
- **work_group_any()** is similar to work_group_all(), but logical OR is used instead

An important limitation about the listed built-ins is that they operate on scalar data types only (for example, popular int4 or float4 types are not supported). Also, 8-bit types such as char or uchar are not supported.

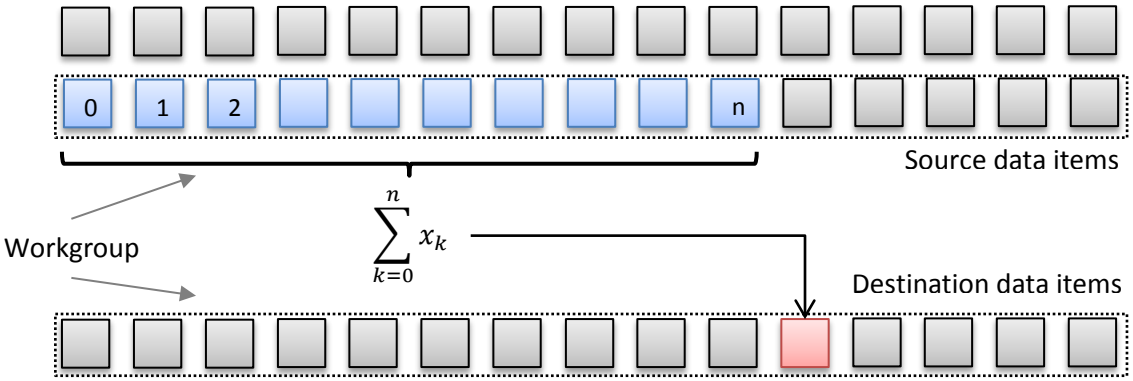
For a complete description, please see chapter 6.13.15 of OpenCL 2.0 C specification.

Work-group functions, as the name implies, always operate in parallel over entire work-group. An implicit consequence from this fact is that any work-group function call acts as a barrier.

Work-group functions usage brings two main benefits. First - work-group functions are convenient. It is much simpler to use a single built-in instead of a bulky piece of code that OpenCL 1.2 user has to write to implement such functionality. Second - work-group functions are more performance efficient, as they use hardware-specific optimizations internally.

Example

For illustration purposes, consider the following task (as a part of some algorithm)- computing prefix sums for equally-sized sub arrays of some larger array. So we have to compute prefix sum for every item of every sub array, and store it to destination memory of the same layout. The source and destination data layout is depicted on the scheme below:



A simple naïve OpenCL kernel for this task might work like this:

- every array (a line on the illustration) will be processed by single workgroup
- for every workitem scan is performed using just plain **for()** loop over preceding items, then cumulative prefix value is added, and finally the result stored to destination
- if workgroup size is smaller than input array for a workgroup, source and destination indices are shifted by workgroup size, cumulative prefix is updated, and process is repeated until the end of source line

The corresponding code is below:

```
__kernel void Calc_wg_offsets_naive(
                    __global const uint* gHistArray,
                    __global uint* gPrefixsumArray,
                    uint bin_size
                    )
{
   uint lid = get_local_id(0);
   uint binId = get_group_id(0);

   //calculate source/destination offset for workgroup
   uint group_offset = binId * bin_size;
   local uint maxval;

   //initialize cumulative prefix
   if( lid == 0 )  maxval = 0;
   barrier(CLK_LOCAL_MEM_FENCE);

   do
   {
      //perform a scan for every workitem
```

```
        uint prefix_sum=0;
        for(int i=0; i<lid; i++)
            prefix_sum += gHistArray[group_offset + i];

        //store result
        gPrefixsumArray[group_offset + lid] = prefix_sum + maxval;
        prefix_sum += gHistArray[group_offset + lid];

        //update group offset and cumulative prefix
        if( lid == get_local_size(0)-1 )  maxval += prefix_sum;
        barrier(CLK_LOCAL_MEM_FENCE);

        group_offset += get_local_size(0);
    }
    while(group_offset < (binId+1) * bin_size);
}
```

This naïve approach is very inefficient in most cases (except probably most tiny workgroup sizes). It is obvious that inner **for()** loop performs too many redundant loads and additions, which actually can be reused. And this redundancy will grow with growing workgroup size. To better utilize Intel HD Graphics hardware capabilities, we need a more work-efficient algorithm, such as Blelloch et al. We won't dive here into algorithm details, since it is perfectly described in classic GPU Gems article - http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html.

OpenCL 1.2 code of this work-efficient parallel scan will look like following:

```
#define WARP_SHIFT 4
#define GRP_SHIFT 8
#define BANK_OFFSET(n)     ((n) >> WARP_SHIFT + (n) >> GRP_SHIFT)

__kernel void Calc_wg_offsets_Blelloch(__global const uint* gHistArray,
                        __global uint* gPrefixsumArray,
                        uint bin_size
                        ,__local uint* temp
                       )
{
    int lid = get_local_id(0);
    uint binId = get_group_id(0);
    int n = get_local_size(0) * 2;

    uint group_offset = binId * bin_size;
    uint maxval = 0;
    do
    {
        // calculate array indices and offsets to avoid SLM bank conflicts
        int ai = lid;
        int bi = lid + (n>>1);
        int bankOffsetA = BANK_OFFSET(ai);
        int bankOffsetB = BANK_OFFSET(bi);

        // load input into local memory
        temp[ai + bankOffsetA] = gHistArray[group_offset + ai];
```

```
        temp[bi + bankOffsetB] = gHistArray[group_offset + bi];

        // parallel prefix sum up sweep phase
        int offset = 1;
        for (int d = n>>1; d > 0; d >>= 1)
        {
            barrier(CLK_LOCAL_MEM_FENCE);
            if (lid < d)
            {
                int ai = offset * (2*lid + 1)-1;
                int bi = offset * (2*lid + 2)-1;
                ai += BANK_OFFSET(ai);
                bi += BANK_OFFSET(bi);
                temp[bi] += temp[ai];
            }
            offset <<= 1;
        }

        // clear the last element
        if (lid == 0)
        {
            temp[n - 1 + BANK_OFFSET(n - 1)] = 0;
        }

        // down sweep phase
        for (int d = 1; d < n; d <<= 1)
        {
            offset >>= 1;
            barrier(CLK_LOCAL_MEM_FENCE);

            if (lid < d)
            {
                int ai = offset * (2*lid + 1)-1;
                int bi = offset * (2*lid + 2)-1;
                ai += BANK_OFFSET(ai);
                bi += BANK_OFFSET(bi);

                uint t = temp[ai];
                temp[ai] = temp[bi];
                temp[bi] += t;
            }
        }
        barrier(CLK_LOCAL_MEM_FENCE);

        //output scan result to global memory
        gPrefixsumArray[group_offset + ai] = temp[ai + bankOffsetA] + maxval;
        gPrefixsumArray[group_offset + bi] = temp[bi + bankOffsetB] + maxval;

        //update cumulative prefix sum and shift offset for next iteration
        maxval += temp[n - 1 + BANK_OFFSET(n - 1)] + gHistArray[group_offset + n - 1];
        group_offset += n;
    }
    while(group_offset < (binId+1) * bin_size);
}
```

Generally, this code is more work-efficient and hardware-friendly, but it has its own caveats.

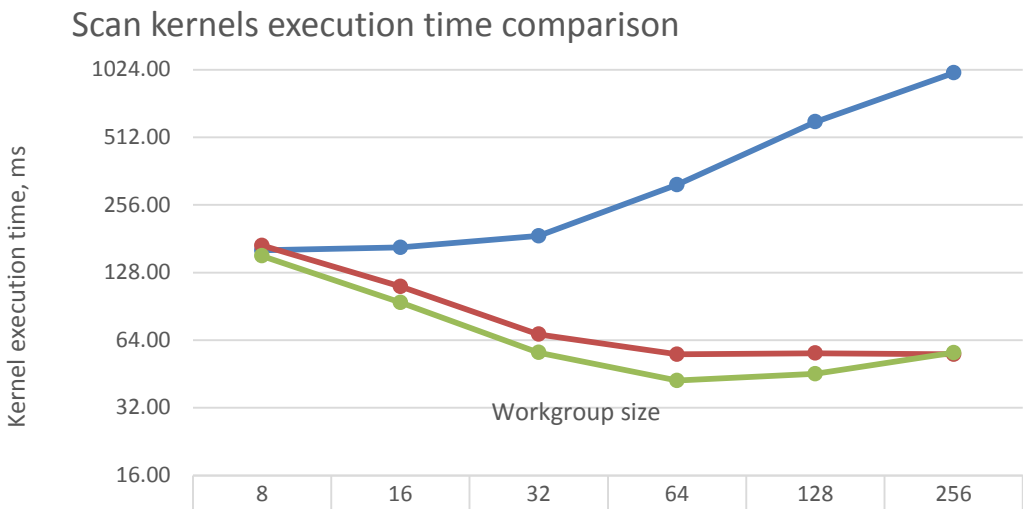
It introduces some overhead on data movement between local and global memory, plus multiple barriers. To amortize them and to be really efficient, this algorithm needs fairly large workgroup size. On tiny workgroup sizes (<16) it unlikely to deliver performance better than naïve loop.

Also note much increased code complexity and additional logic to avoid shared local memory bank conflicts (e.g. BANK_OFFSET macro).

Usage of workgroup functions overcomes all mentioned issues. Corresponding variant of optimized OpenCL code below:

```
__kernel void Calc_wg_offsets_wgf(
                    __global const uint* gHistArray,
                    __global uint* gPrefixsumArray,
                    uint bin_size
                    )
{
   uint lid = get_local_id(0);
   uint binId = get_group_id(0);

   uint group_offset = binId * bin_size;
   uint maxval = 0;

   do
   {
      uint binValue = gHistArray[group_offset + lid];
      uint prefix_sum = work_group_scan_exclusive_add( binValue );
      gPrefixsumArray[group_offset + lid] = prefix_sum + maxval;

      maxval += work_group_broadcast( prefix_sum + binValue, get_local_size(0)-1 );

      group_offset += get_local_size(0);
   }
   while(group_offset < (binId+1) * bin_size);
}
```

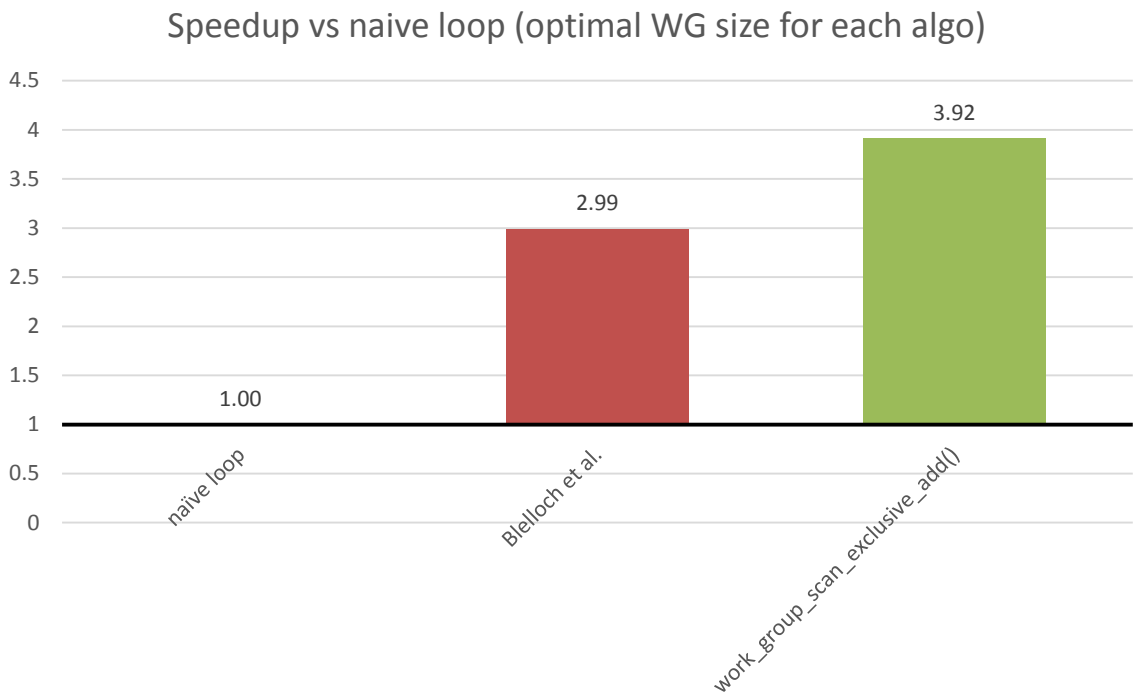
Performance results from both optimizations are measured on fairly large input data (each workgroup scans 65536 items, which corresponds to 8192 … 2048 outer loop iterations, depending on local size):



	8	16	32	64	128	256
naïve loop	160.71	165.64	186.23	314.84	599.30	990.50
Blelloch et al.	168.92	110.95	67.92	55.33	56.04	55.33
work_group_scan_exclusive_add()	151.69	94.20	56.40	42.30	45.31	56.25

As expected, naïve loop performs much worse with growing local size, while both optimized variants improve.

If workgroup size will be set to optimal for given algorithm, kernels comparison will look like this:



Notice how `work_group_scan_exclusive_add()` usage significantly improves performance on any workgroup size, while simultaneously simplifying the code.

Conclusion

Work-group functions addition is an important step forward with OpenCL 2.0 spec, equipping OpenCL developers with new great built-ins. Clever usage of this new feature can save a significant amount of investments in development of complex and performance-efficient OpenCL applications.