Leverage Your OpenCL™ Investment on Intel® Architectures

By James Cownie, Principal Engineer, Intel Corporation, Simon McIntosh-Smith, University of Bristol, Arik Narkis, OpenCL Performance Architect, Intel Corporation, and Ayal Zaks, Manager, OpenCL MIC Compiler, Intel Corporation

Open Computing Language (OpenCL™) is a royalty-free standard for cross-platform parallel programming. The OpenCL programming model supports heterogeneous platforms with various types of devices including general-purpose CPUs, graphics processors, and coprocessors. The OpenCL specification includes several low-level features that specifically address GPUs, such as local memory, barriers, images, samplers, and workgroups. Such features may not exist in other devices. Intel introduced OpenCL support for its general-purpose CPUs in 2010, and added support for the Intel® Xeon Phi™ coprocessor, a highly parallel x86 coprocessor, early in 2013.

OpenCL mainly attracts developers because of its promise of cross-device portability and its broad support by hardware vendors. OpenCL code designed for a specific target GPU device will usually run on x86 seamlessly, or with only minor changes. Nevertheless, performance portability
across devices is not guaranteed. Specifically, OpenCL code tuned for a GPU would typically perform sub-optimally on an Intel Xeon Phi coprocessor, and vice versa. This is normally caused by the use of GPU hardware-specific features such as local memory and barriers; the use of too few workgroups (which exposes insufficient coarse-grain parallelism); or workgroups that are too small (leading to excessive fine-grain parallelism).

A typical Intel Xeon Phi coprocessor includes 60 Intel® architecture cores, each capable of running up to four hardware threads. The OpenCL runtime schedules these 240 hardware threads at the granularity of whole workgroups. Therefore, any OpenCL task (NDRange) which includes fewer than 240 work-groups will underutilize the available execution resources. To achieve high resource utilization and efficient load balancing, one should aim for at least one thousand workgroups in each NDRange.

With OpenCL on the Intel Xeon Phi coprocessor, fine-grained NDRanges may cause some runtime scheduling overhead. This overhead becomes noticeable when workgroups take less than 200 microseconds each. Therefore, coarser-grained workgroups are preferred. One way to achieve this is to increase the workgroup duration by fusing multiple kernels into a single larger kernel. Another option is to configure the NDRange to include fewer but larger workgroups, while maintaining the minimal number of workgroups as described above.

Intel's OpenCL compiler implicitly vectorizes each workgroup routine based on dimension zero of the NDRange (even though it cannot know its value). In effect, the dimension zero loop is unrolled by the target vector size. For example, a generated 3-D kernel code might look like this:

```c
_kernel kernel_name(/* kernel arguments */) {
  for (int dim2=0; dim2 < get_local_size(2); dim2++)
    for (int dim1=0; dim1 < get_local_size(1); dim1++)
      {
        for (int dim0=0; dim0 + VECTOR_SIZE < get_local_size(0);
             dim0 += VECTOR_SIZE)
          /* Vectorized_Kernel_Body */
          for (; dim0 < get_local_size(0); dim0++)
            /* Original_Kernel_Body, for remainder iterations */
      }
}
```

A kernel being vectorized across dimension zero of the workgroup.
On an Intel Xeon Phi coprocessor, the vectorization size is set to 16 regardless of the data types used. It is therefore good practice to make the local size at dimension zero a multiple of 16. Moreover, unit stride accesses to buffers along dimension zero should result in more efficient vector code.

Although existing OpenCL code that has been tuned for GPUs may perform sub-optimally on the Intel Xeon Phi coprocessor, with modest tuning effort the same OpenCL code can perform well on both Intel Xeon Phi and GPUs. One application that exhibits this behavior is the BUDE molecular docking code developed by Simon McIntosh-Smith and colleagues at the University of Bristol. BUDE's OpenCL implementation sustains over 32% of peak floating point performance on a wide range of many-core architectures, including the Intel Xeon Phi coprocessor. BUDE's approach to delivering performance portability with OpenCL is described in the recent paper *High Performance in silico Virtual Drug Screening on Many-Core Processors.*

Bristol joined the Intel Parallel Computing Center Program in February 2014, with a particular focus on exploiting OpenCL and OpenMP® to leverage Intel Xeon Phi. Codes being optimized in Bristol for these platforms include the BUDE molecular docking code, the ROTORSIM CFD code, and the CloverLeaf mini-app from the Mantevo benchmark suite.

**Summary**

OpenCL has evolved considerably since the release of its initial (1.0) specification at the end of 2008. The range of implementations currently available provides a de facto portable programming standard across disparate heterogeneous platforms, enabling customers to leverage their development investments. Both the specification and implementations are still actively advancing. Performance portability is one major challenge that is being pursued by many OpenCL teams in both research and industry. In this short article, we have outlined some of the performance portability considerations related specifically to the Intel Xeon Phi coprocessor.

References


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
In a previous post I discussed the Intel® Transactional Synchronization Extensions (Intel® TSX) technology released in the new generation of processors. I described the Intel® Threading Building Blocks (Intel® TBB) implementation of the HLE interface (speculative_spin_mutex). Now we can talk about the implementation of speculative_spin_rwlock_mutex, a Preview Feature of TBB 4.2 Update 2.

speculative_spin_rwlock_mutex uses RTM for mutual exclusion, and allows both concurrent reads and concurrent writes. It also contains a spin_rwlock_mutex because it may be necessary to perform the operation without speculation.

If concurrent executions of code protected by the mutex do not conflict, all reads complete and writes are atomically committed without explicitly taking the lock.

If there is a conflict or another problem that prevents speculative execution, and the transaction is aborted, speculative_spin_rwlock_mutex may retry the transaction or it may take the lock for real. If a writer takes the lock for real, all speculative readers and writers will abort the transaction and wait for the writer to complete, at which time the transactions may be retried. If a reader takes the lock for real all speculative writers will abort the transaction and wait for the reader to release the lock, at which time the transactions may be retried. Real readers and speculative readers may proceed in parallel. All this happens “under the covers,” as part of the TBB implementation.