LIBXSTREAM

Library to work with streams, events, and code regions that are able to run asynchronous while preserving the usual stream conditions. The library is targeting Intel Architecture (x86) and helps to offload work to an Intel Xeon Phi coprocessor (an instance of the Intel Many Integrated Core "MIC" Architecture). For example, using two streams may be an alternative to the usual double-buffering approach which can be used to hide buffer transfer time behind compute.

Interface

The library's C API completely seals the implementation and only forward-declares the types used in the interface. The C++ API allows to make use of the stream and event types directly although it is seen to be an advantage to only rely on the C interface. The C++ API is currently required for own code to be queued into a stream. However, a future revision will allow to only rely on a function pointer and a plain C interface. A future release may also provide a native FORTRAN interface.

Data Types: are forward-declarations of the types used in the interface. The C++ API allows to make use of the stream and event types directly although it is seen to be an advantage to only rely on the C interface.

```
/** Data type representing a signal. */
typedef uintptr_t libxstream_signal;
/** Forward declaration of the stream type (C++ API includes the definition). */
typedef struct libxstream_stream libxstream_stream;
/** Forward declaration of the event type (C++ API includes the definition). */
typedef struct libxstream_event libxstream_event;
```

Device Interface: provides the notion of an "active device" (beside of allowing to query the number of available devices). Multiple active devices can be specified on a per host-thread basis. None of the (main) API functions (see below) implies an active device. It is up to the user to make use of this notion.

```
/** Query the number of available devices. */
int libxstream_get_ndevices(size_t* ndevices);
/** Query the device set active for this thread. */
int libxstream_get_active_device(int* device);
/** Set the active device for this thread. */
int libxstream_set_active_device(int device);
```

Memory Interface: is mainly for handling device-side buffers (allocation, copy). It is usually beneficial to allocate host memory with below functions as well. However, any memory allocation on the host is interoperable. It is also supported copying parts to/from a buffer using below API.

```
/** Query the memory metrics of the given device (valid to pass one NULL pointer). */
int libxstream_mem_info(int device, size_t* allocatable, size_t* physical);
/** Allocate aligned memory (0: automatic) on the given device. */
int libxstream_mem_allocate(int device, void** memory, size_t size, size_t alignment);
/** Deallocate memory; shall match the device where the memory was allocated. */
int libxstream_mem_deallocate(int device, const void* memory);
/** Fill memory with zeros; allocated memory can carry an offset. */
int libxstream_memset_zero(void* memory, size_t size, libxstream_stream* stream);
/** Copy memory from the host to the device; addresses can carry an offset. */
int libxstream_memcpy_h2d(const void* mem, void* dev_mem, size_t, libxstream_stream*);
/** Copy memory from the device to the host; addresses can carry an offset. */
int libxstream_memcpy_d2h(const void* dev_mem, void* mem, size_t, libxstream_stream*);
/** Copy memory from device to device; cross-device copies are allowed as well. */
int libxstream_memcpy_d2d(const void* src, void* dst, size_t size, libxstream_stream*);
```

Stream Interface: is used to expose the available parallelism. A stream preserves the predecessor/successor relationship while participating in a pipeline (parallel pattern) in case of multiple streams. Synchronization points can be introduced using the stream interface as well as the event interface.

```
/** Query the range of valid priorities (inclusive bounds). */
int libxstream_stream_priority_range(int* least, int* greatest);
/** Create a stream on a device (demux<0: auto-locks, 0: manual, demux>0: sync.). */
int libxstream_stream_create(libxstream_stream**, int device, int demux, int, char*);
/** Destroy a stream; pending work must be completed if results are needed. */
int libxstream_stream_destroy(libxstream_stream* stream);
/** Wait for a stream to complete pending work; NULL to synchronize all streams. */
int libxstream_stream_sync(libxstream_stream* stream);
/** Wait for an event recorded earlier; NULL increases the match accordingly. */
int libxstream_stream_wait_event(libxstream_stream* stream, libxstream_event* event);
/** Lock a stream such that the caller thread can safely enqueue work. */
int libxstream_stream_lock(libxstream_stream* stream);
/** Unlock a stream such that another thread can acquire the stream. */
int libxstream_stream_unlock(libxstream_stream* stream);
```

Event Interface: provides a more sophisticated mechanism allowing to wait for a specific work item to complete without the need to also wait for the completion of work queued after the item in question.

```
/** Create an event; can be used multiple times to record an event. */
int libxstream_event_create(libxstream_event** event);
/** Destroy an event; does not implicitly waits for the completion of the event. */
int libxstream_event_destroy(libxstream_event* event);
/** Record an event; an event can be re-recorded multiple times. */
int libxstream_event_record(libxstream_event* event, libxstream_stream* stream);
/** Check whether an event has occurred or not (non-blocking). */
int libxstream_event_query(const libxstream_event* event, int* has_occured);
/** Wait for an event to complete i.e., work queued prior to recording the event. */
int libxstream_event_synchronize(libxstream_event* event);
```

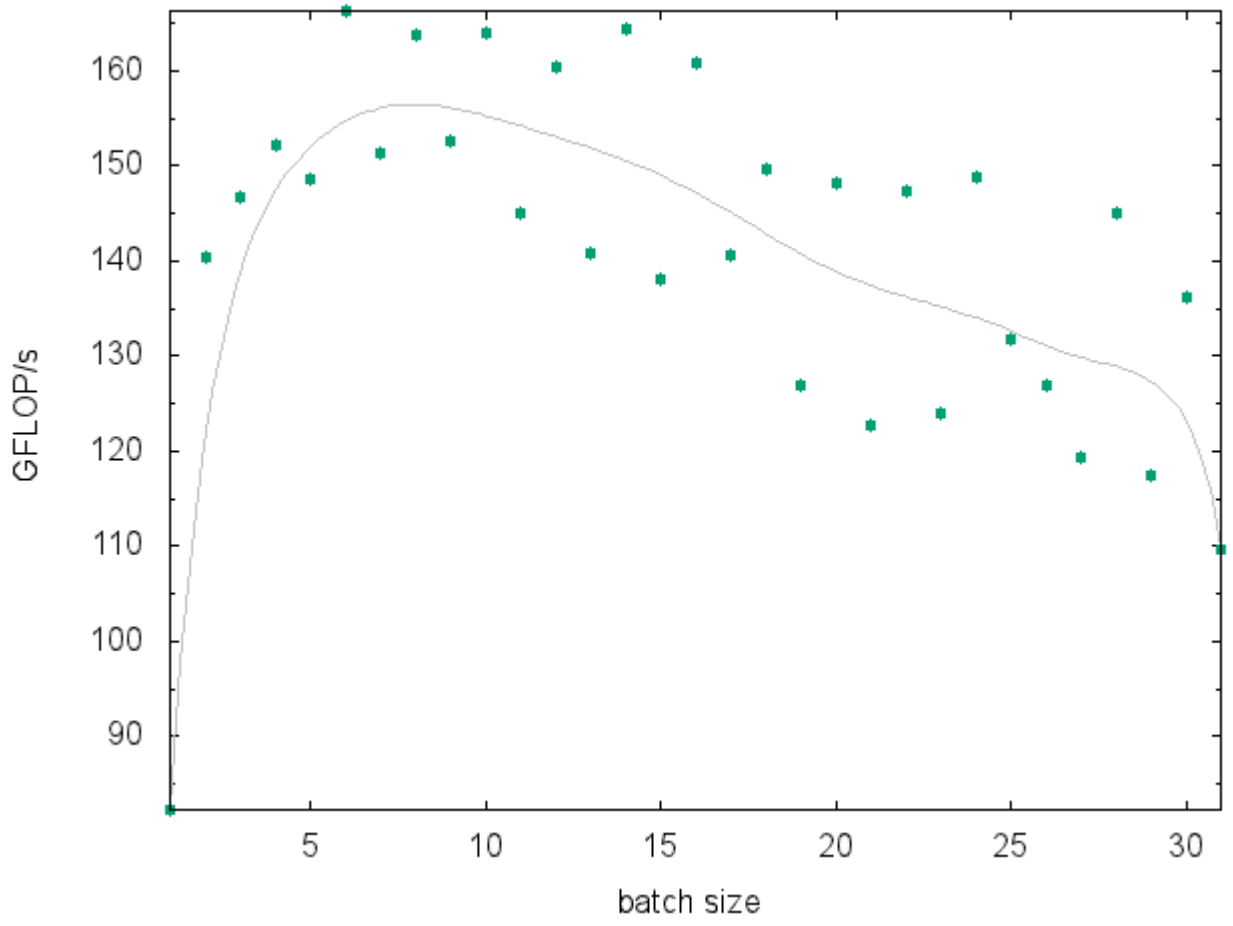
Implementation

The library's implementation allows queuing work from multiple host threads in a thread-safe manner and without oversubscribing the device. The actual implementation vehicle can be configured using a configuration header. Currently Intel's Language Extensions for Offload (LEO) are used to perform asynchronous execution and data transfers using signal/wait clauses. Other mechanisms can be implemented e.g., hStreams or COI (both are part of the Intel Manycore Platform Software Stack), or offload directives as specified by OpenMP.

The current implementation is falling back to host execution in cases where no coprocessor is present, or when the executable was not built using the Intel Compiler. However, there is no attempt (yet) to exploit the parallelism available on the host system.

Performance

The multi-dgemm sample code is the implementation of a benchmark (beside of illustrating the use of the library). The shown performance is not meant to be "the best case". Instead, the performance is reproduced by a program constructing a series of matrix-matrix multiplications of varying problem sizes with no attempt to avoid the implied performance penalties (see underneath the graph for more details). A reasonable host system and benchmark implementation is likely able to outperform below results (no transfers, etc.).

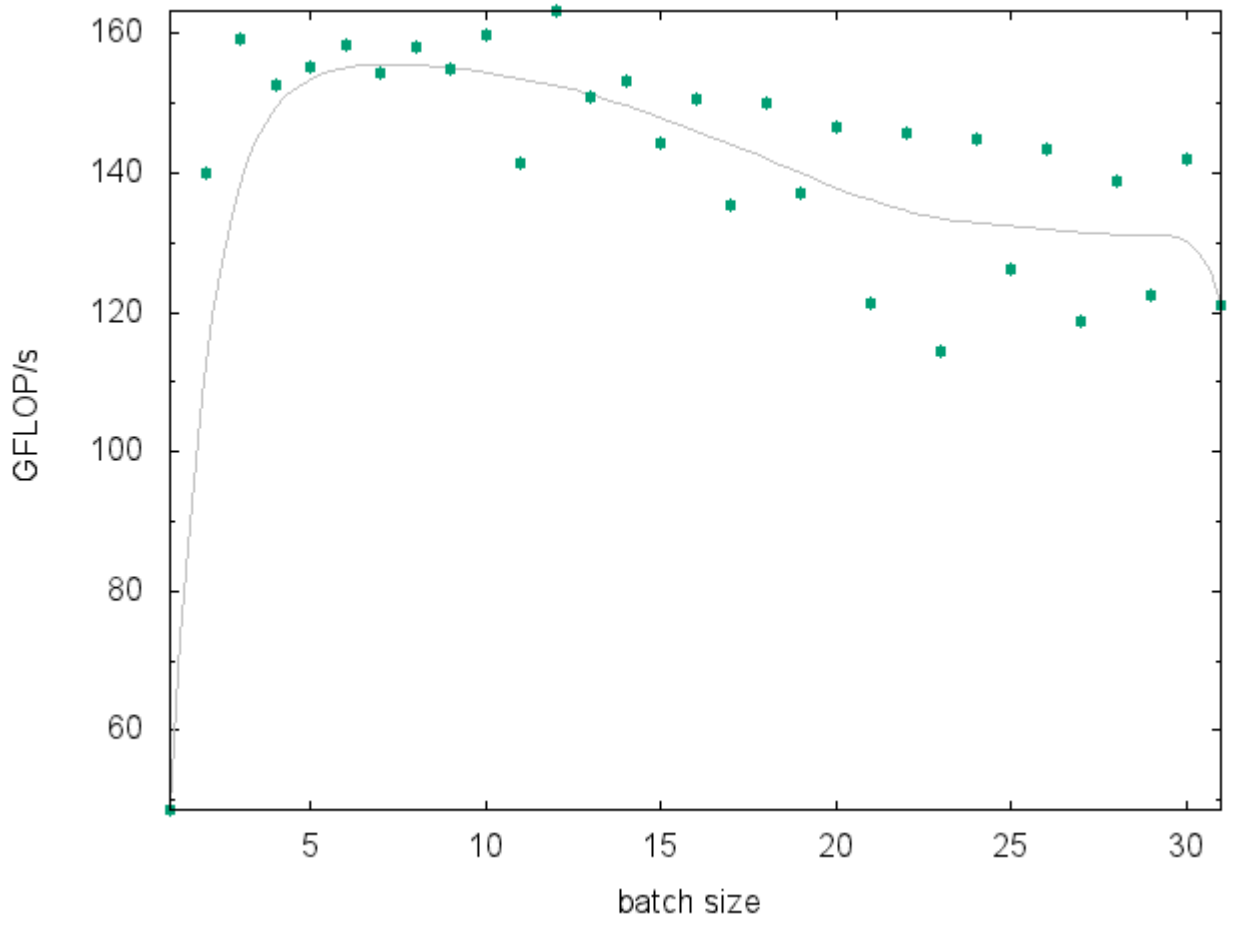


This performance graph has been created for a single Intel Xeon Phi 7120 Coprocessor card by running "OFFLOAD_DEVICES=0 ./benchmark.sh 250 1 2 1" on the host system. The script varies the number of matrix-matrix multiplications queued at once. The program is rather a stress-test than a benchmark since there is no attempt to avoid the performance penalties as mentioned below. The plot shows ~150 GFLOPS/s even with smaller batch sizes.

Even the series of matrices with the largest problem size of the mix is not close to being able to reach the peak performance, and there is an insufficient amount of FLOPS available to hide the cost of transferring the data. The data needed for the computation moreover includes a set of indices describing the offsets of each of the matrix operands in the associated buffers. The latter implies unaligned memory accesses due to packing the matrix data without a favorable leading dimension. Transfers are performed as needed on a per-computation basis rather than aggregating a single copy-in and copy-out prior and past of the benchmark cycle. Moreover, there is no attempt to balance the mixture of different problem sizes when queuing the work into the streams.

Tuning

The library supports a manual locking approach which can be requested at runtime on a per-stream basis instead of an automatic internal locking ("demux" mode). Manual locking also allows queuing work without the need for intermediate stream synchronization in case the effect of the work is not needed at this point in time. The locking approach effectively describes a logical group of work. In contrast, the automatic locking attempts to derive this information at the points where the stream synchronization function is called.



The above plot illustrates the impact of manual locking in contrast to the "demux" mode of operation. Results have been gathered by running "OFFLOAD_DEVICES=0 ./benchmark.sh 250 1 2 0" with otherwise the same conditions as mentioned in the Performance section. The plot shows ~155 GFLOPS/s and therefore a minor impact of manual locking.

Please note that the manual locking approach does not contradict the thread-safety claimed by the library; each queuing operation is still atomic. Synchronization and locking in general avoids intermixing work from different logical groups of work. An example where this becomes a problem (data races) is when the work is buffered only for a subset (work group) of the total amount of work, and when multiple host threads are queuing work items into the same stream at the same time.