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 application programming interface (API) completely seals the implementation and only forward-declares types which are beyond the language's built-in types. The entire API consists of below subcategories each illustrated by a small code snippet. The Function Interface for instance enables an own function to be enqueued for execution within a stream (via function pointer). A future release of the library will provide a native FORTRAN interface.

Data Types

Data types are forward-declared types used in the interface. Moreover, there is a mapping from the language's built-in types to the types supported in the Function Interface.

```
/** Boolean state. */
typedef int libxstream_bool;
/** Stream type. */
typedef struct libxstream_stream libxstream_stream;
/** Event type. */
typedef struct libxstream_event libxstream_event;
/** Enumeration of elemental "scalar" types. */
typedef enum libxstream_type { /** see libxstream.h */
  /** special types: BOOL, BYTE, CHAR, VOID */
  /** signed integer types: I8, I16, I32, I64 */
  /** unsigned integer types: U8, U16, U32, U64 */
  /** floating point types: F32, F64, C32, C64 */
} libxstream_type;
/** Function call behavior (flags valid for binary combination). */
typedef enum libxstream_call_flags {
  LIBXSTREAM_CALL_WAIT    = 1 /* synchronous function call */,
  LIBXSTREAM_CALL_NATIVE  = 2 /* native host/MIC function */,
  /** collection of any valid flags from above */
  LIBXSTREAM_CALL_DEFAULT = 0
} libxstream_call_flags;
/** Function argument type. */
typedef struct libxstream_argument libxstream_argument;
/** Function type of an offloadable function. */
typedef void (*libxstream_function)(LIBXSTREAM_VARIADIC);
```

Device Interface

The device interface allows to query the number of available devices as well as querying some important metrics (physical and available memory).

```
size_t ndevices = 0, allocatable = 0;
libxstream_get_ndevices(&ndevices);
libxstream_get_meminfo(-1/*host*/, &allocatable, NULL);
```

Beside of the minimalistic device-orientied functionality, this interface also allows to rely on the notion of an "active device". This way, multiple devices can be driven on a per host-thread basis.

```
/* Initialize active device on a per host-thread basis */
libxstream_set_active_device(omp_get_thread_num() % ndevices);
```

Relying on an active device is explicit for entirely library, and it is up to the user to make use of this notion.

```
int device = -1;
libxstream_get_active_device(&device);
libxstream_get_meminfo(device, &allocatable, NULL);
```

Memory Interface

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

```
const int hst = -1, dev = 0, i = 0;
libxstream_mem_allocate(hst, &ihst, sizeof(double) * nitems, 0/*auto-alignment*/);
libxstream_mem_allocate(hst, &ohst, sizeof(double) * nitems, 0/*auto-alignment*/);
/* TODO: initialize with some input data */
libxstream_mem_allocate(dev, &idev, sizeof(double) * nbatch, 0/*auto-alignment*/);
libxstream_mem_allocate(dev, &odev, sizeof(double) * nbatch, 0/*auto-alignment*/);

for (i = 0; i < nitems; i += nbatch) {
  const int ibatch = sizeof(double) * min(nbatch, nitems - i), j = i / nbatch;
  libxstream_memcpy_h2d(ihst + i, idev, ibatch, stream[j%2]);
  /* TODO: invoke user function (see Function Interface) */
  libxstream_memcpy_d2h(odev, ohst + i, ibatch, stream[j%2]);
}
```

It is possible to query properties of the allocated buffers using the base address as returned by the above allocation function:

```
size_t size = 0;  /*will be nitems x sizeof(double)*/
int device = -1;  /*will be 'dev' because 'idev' is allocated there*/
void* mapped = 0; /*will be an address only valid on device-side*/
libxstream_mem_info(idev, &mapped, &size, &device);
/* deallocating the above memory buffers... */
libxstream_mem_deallocate(hst, ihst);
libxstream_mem_deallocate(hst, ohst);
libxstream_mem_deallocate(dev, idev);
libxstream_mem_deallocate(dev, odev);
```

Stream Interface

The 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.

```
libxstream_stream* stream[2];
libxstream_stream_create(stream + 0, d, 0/*priority*/, "s1");
libxstream_stream_create(stream + 1, d, 0/*priority*/, "s2");
/* TODO: do something with the streams */
libxstream_stream_wait(NULL); /*wait for all streams*/
libxstream_stream_destroy(stream[0]);
libxstream_stream_destroy(stream[1]);
```

Event Interface

The 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.

```
libxstream_event* event[2/*N*/];
libxstream_event_create(event + 0);
libxstream_event_create(event + 1);

for (i = 0; i < nitems; i += nbatch) {
  const size_t n = i / nbatch, j = n % N, k = (j + 1) % N;
  /* TODO: copy-in, user function, copy-out */
  libxstream_event_record(event + j, stream + j);

  /* synchronize work N iterations in the past */
  libxstream_event_wait(event[k]);
}

libxstream_event_destroy(event[0]);
libxstream_event_destroy(event[1]);
```

Function Interface

The function interface is used to call a user function and to describe its list of arguments (signature). The function's signature consists of inputs, outputs, or in-out arguments. An own function can be enqueued for execution within a stream by taking the address of the function. In order to avoid repeatedly allocating (and deallocating) a signature, a thread-local signature with the maximum number of arguments supported can be constructed i.e., the thread-local signature is cleared to allow starting over with an arity of zero.

```
size_t nargs = 5, arity = 0;
libxstream_argument* args = 0;
libxstream_fn_signature(&args); /*receive thread-local function signature*/
libxstream_fn_nargs (args, &nargs); /*nargs==LIBXSTREAM_MAX_NARGS*/
libxstream_get_arity(args, &arity); /*arity==0 (no arguments constructed yet)*/
libxstream_fn_call((libxstream_function)f, args, stream, LIBXSTREAM_CALL_DEFAULT);
libxstream_fn_destroy_signature(args); /*(can be used for many function calls)*/
```

void fc(const double* scale, const float* in, float* out, const size_t* n, size_t* nzeros)

For the C language, a first observation is that all arguments of the function's signature are passed "by pointer"; even a value that needs to be returned (which also allows multiple results to be delivered). Please note that non-elemental ("array") arguments are handled "by pointer" rather than by pointer-to-pointer. The mechanism to pass an argument is called "by-pointer" (or by-address) to distinct from the C++ reference type mechanism. Although all arguments are received by pointer, any elemental ("scalar") input is present by value (which is important for the argument's life-time). In contrast, an elemental output is only present by-address, and therefore care must be taken on the call-side to ensure the destination is still valid when the function is executed. The latter is because the execution is asynchronous by default.

void fpp(const double& scale, const float* in, float* out, const size_t& n, size_t& nzeros)

For the C++ language, the reference mechanism can be used to conveniently receive an elemental argument "by-value" including elemental output arguments as "non-const reference" ("return value").

```
const libxstream_type sizetype = libxstream_map_to<size_t>::type();
libxstream_fn_input (args, 0, &scale, libxstream_map_to_type(scale), 0, NULL);
libxstream_fn_input (args, 1, in,  LIBXSTREAM_TYPE_F32, 1, &n);
libxstream_fn_output(args, 2, out, LIBXSTREAM_TYPE_F32, 1, &n);
libxstream_fn_input (args, 3, &n, sizetype, 0, NULL);
libxstream_fn_output(args, 4, &nzeros, sizetype, 0, NULL);
```

Beside of showing some C++ syntax in the above code snippet, the resulting signature is perfectly valid for both the "fc" and the "fpp" function. Mapping type definitions in C (and FORTRAN) similar to the above C++ code, one can rely on libxstream_get_autotype's "start" argument pointing into a "type category" (order in which types are enumerated). For example in the following code, the type "size_t" is considered to be at least an unsigned integer type but may be deduced to LIBXSTREAM_TYPE_U64 according to the result of sizeof(size_t).

```
libxstream_type sizetype = LIBXSTREAM_TYPE_U32;
libxstream_get_autotype(sizeof(size_t), sizetype, &sizetype);
```

Weak type information

To construct a signature with only weak type information, one may (1) not distinct between in-out and output arguments; even non-elemental inputs can be treated as an in-out argument, and (2) use LIBXSTREAM_TYPE_VOID as an elemental type or any other type with a type-size of one (BYTE, I8, U8, CHAR). Weak-typed arguments imply that extents are counted in Byte rather than in number of elements. Moreover, weak-typed scalar arguments need to supply a "shape" indicating the actual size of the element in Byte.

```
const size_t typesize = sizeof(float);
/* argument type in function signature: const float* or const float& */
libxstream_fn_input(args, 0,  &f1, LIBXSTREAM_TYPE_VOID, 0, &typesize);
/* argument type in function signature: unsigned char* */
libxstream_fn_inout(args, 1, data, LIBXSTREAM_TYPE_BYTE, 1, &numbytes);
```

Query Interface

This "device-side API" allows to query information about function arguments when inside of a user function (which is called by the library). This can be used to introspect the function's arguments in terms of type, dimensionality, shape, and other properties. In order to query a property, the position of the argument within the signature needs to be known. However, the position within the signature can be also queried (when inside of a library-initiated call context) by using the actual function argument (given by address). To refer a function's signature when inside of a function, a NULL-pointer is passed to designate the function signature of the current call context.

Revised function "fc" querying argument properties

As one can see, the signature of a function can often be trimmed to omit arguments which certainly describe the shape of an argument (below function signature omits the "n" argument shown in one of the previous examples).

```
LIBXSTREAM_TARGET(mic) void f(const double* scale, const float* in, float* out, size_t* nzeros)
{
  libxstream_type type = LIBXSTREAM_TYPE_VOID;
  size_t in_position = 0, n = 0;
  const char* name = 0;

  libxstream_get_argument(in, &in_position); /*in_position == 1*/
  libxstream_get_shape(NULL/*this call context*/, in_position, &n);
  libxstream_get_type (NULL/*this call context*/, 2/*out*/, &type);
  libxstream_get_typename(type, &name);
  printf("type=%s", name); /*f32*/
}
```

Variadic functions

The query interface allows enumerating and accessing function arguments including the raw data behind the argument. This applies regardless of whether the arguments are given explicitly or when supplied via a variadic part of the function's signature (ellipsis).

```
LIBXSTREAM_TARGET(mic) void fill(float* out, ...)
{
  size_t n = 0, i = 0, arity = 0;
  float fill_value = 0; /*default*/
  const void* data = 0;

  libxstream_get_shape(NULL/*this call context*/, 0/*out*/, &n);
  libxstream_get_arity(NULL/*this call context*/, &arity);
  if (1 < arity) { /*non-default fill value*/
    libxstream_get_data(NULL/*this call context*/, 1, &data);
    fill_value = *(const float*)data; /*better check the type*/
  }
  for (i = 0; i < n; ++i) out[i] = fill_value;
}
```

Performance

Prior to benchmarking more complex code, a copy-in and copy-out "microbenchmark" based on the copy sample code is performed. The performance should be only limited by the PCIe transfer speed (generation 2) rather than any other overhead induced by the library.

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". The performance is reproduced by 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 implementation producing the same output is likely able to outperform the benchmark.

Even the series of matrices with the largest problem size exhibits an insufficient amount of FLOPS in order to hide the cost of transferring the data. The data transfers are also including 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

Hybrid Parallelism

Additional scalability can be unlocked when running an application which is parallelized using the Message Passing Interface (MPI). In this case, the device(s) can be partitioned according to the number of ranks per host processor. To read more about this, please visit the MPIRUN WRAPPER project. To estimate the impact of this technique, one can scale the number of threads on the device until the performance saturates and then partition accordingly.

Implementation

Background

The library's implementation allows enqueuing 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.

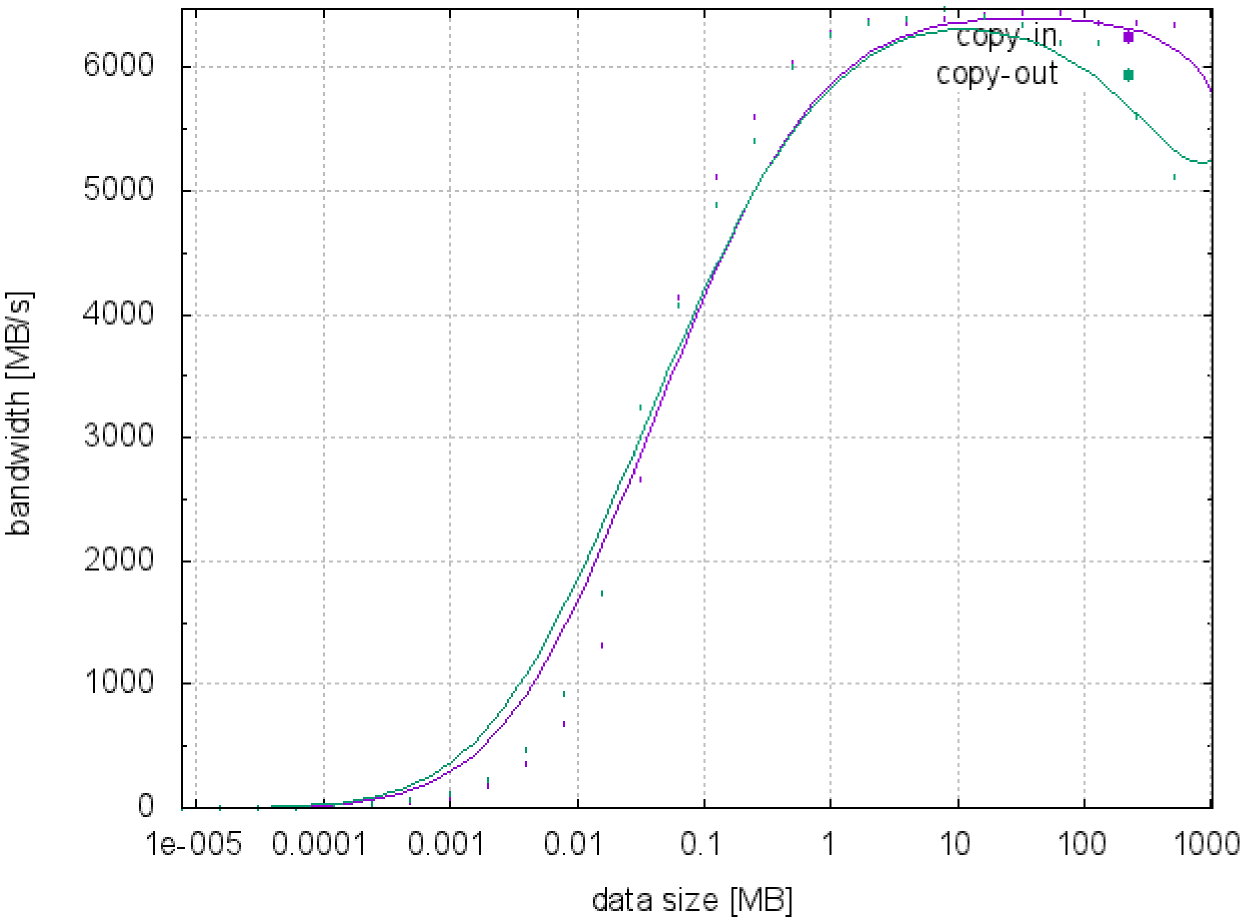

Figure 1: This graph shows the performance of an Intel Xeon Phi 3120 Coprocessor card performing the copy benchmark. The code has been built by running "./make.sh", and then running "./copy.sh i" and "./copy.sh o" on the host system. The benchmark performs a series of copy-in and copy-out operations followed by a synchronization operation after each transfer. The synchronization for every transfer is actually superfluous, however the benchmark is also attempting to capture an eventually induced overhead. The shown bandwidth is only limited by the transfer bandwidth of the PCIe gen. 2 interface, and the result is also expected to be independent of running on a 3, 5, or 7-series card.

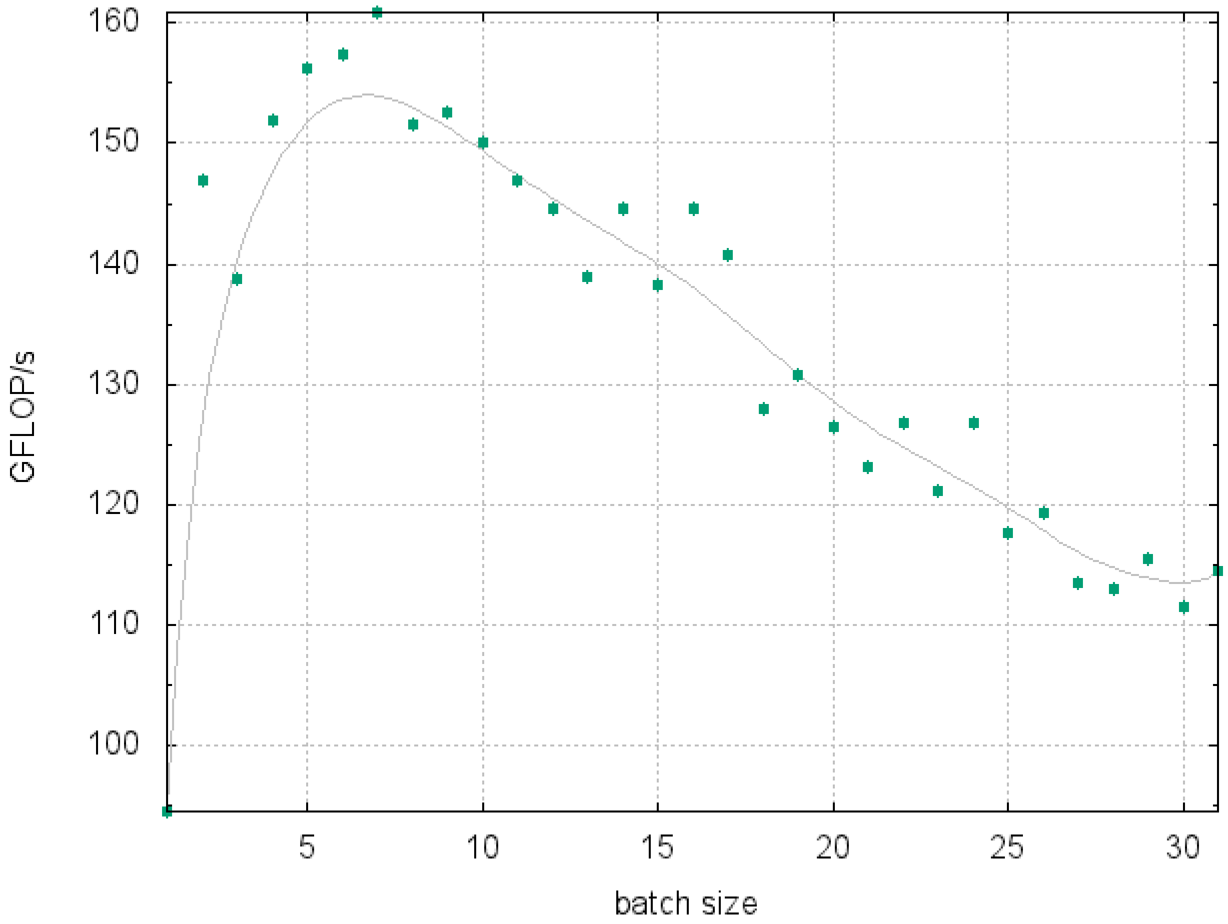

Figure 2: This graph shows the performance of a single Intel Xeon Phi 7120 Coprocessor card performing the multi-dgemm benchmark. The code has been built by running "./make.sh", and then running "env OFFLOAD_DEVICES=0 ./benchmark.sh" on the host system. The script varies the number of matrix-matrix multiplications 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 is showing a performance of more than 200 GFLOPS/s even with smaller batch sizes.

Limitations

There is a known performance limitations addressed by the Roadmap. Namely the asynchronous offload is currently disabled (see LIBXSTREAM_ASYNC in the configuration header) due to an improper synchronization infrastructure. This issue will be addressed along with scheduling work items. The latter is also a prerequisite for an efficient hybrid execution. Hybrid execution and Transparent High Bandwidth Memory (HBM) support will both rely on an effective association between host and “device” buffers in order to omit unnecessary memory copies.

Roadmap

Although the library is under development, the interface is stable. There is a high confidence that all features planned are mainly work “under the hood” i.e., code written now can scale forward. The following issues are being addressed in upcoming revisions:

- Transparent High Bandwidth Memory (HBM) support
- Hybrid execution (host and coprocessors)
- Native FORTRAN interface

Applications and References

[1] **http://cp2k.org/**: Open Source Molecular Dynamics application. An experimental branch at GitHub uses the library to offload work to an Intel Xeon Phi coprocessor (see https://github.com/hfp/libxstream/raw/master/documentation/cp2k.pdf).

[2] **https://github.com/01org/pyMIC**: Python module to offload computation to Intel Xeon Phi coprocessors.

[3] **http://software.intel.com/xeonphicatalog**: Intel Xeon Phi Applications and Solutions Catalog.

[4] **http://goo.gl/qsnOOf**: Intel 3rd Party Tools and Libraries.