**XMP Library**

XMP is a CUDA® accelerated(X) Multi-Precision library that enables batched multi-precision operations in CUDA.

The API is designed around a handle. A handle contains information necessary to execute operations. This includes but is not limited to the device, the stream, and scratch pad memory. A handle must be allocated for every device that a user will use. The handle is passed into every API.

In order to expose more parallelism XMP uses a batched interface where multiple instances of an operation are performed in parallel across the GPU. This allows the GPU to be fully utilized and achieve peak performance. The base type is an array of fixed precision integers (xmpIntegers\_t). The number of integers in an array is referred to as its count and the size of each integer in bits is referred to as its precision.

A number of other restrictions exist in this version of the library. A complete list of restrictions is listed after every API call. These restrictions may be relaxed in future releases.

The following is an API reference guide.

**Asynchronous Execution**

API calls are synchronous with respect to the host unless the asynchronous version of the API call is used. Asynchronous APIs are named with an “Async” on the end of their name. When using asynchronous APIs the user must ensure that all prior calls have completed execution by either calling a synchronous API call on the same handle or querying the handle for the stream and explicitly calling CUDA synchronization (see the CUDA Programming guide for more details).

**Execution Policies**

Execution policies control the behavior of the execution. A default execution policy is created with the handle. A new execution policy can be created using *xmpExecutionPolicyCreate*.

**Indexing**

By default indexing uses the natural order. That is the first instance executes at index 0, second index at index 1, and so on. In the case that an index exceeds the array size the index wraps back to 0 in a modular fashion. This allows for performing operations like multiplying every element in the array by the same number without having to replicate the number.

The indexing scheme can be arbitrarily modified using an execution policy. The API call xmpExecutionPolicySetIndices allows the user to specify an array of indices to be used when indexing the array. Using this the array can be traversed in any order. Like the default indexing scheme if the instance exceeds the index array length the modulus of the index will be taken. In addition, if the value in the index array exceeds the count of the array the modulus will be taken again.

Modular indexing on an output array is not supported. Having multiple output indices mapping to the same element of the integer array produces undefined results.

For clarity the indexing forumula is listed below:

Input arrays: idx = index[i%index\_count] % array\_count

Output arrays: idx = index[i]

Where i is the instance being executioned, index is the index array, index\_count is the size of the index\_array, and array\_count is the size of the integer array.

**Operators**

This library implements operators which work on arrays of integers. These operators use the indexing policy specified in the current execution policy.

All pointers passed into operators can exist in host memory or device memory. In order for asynchronous routines to execute asynchronously any host memory must be pinned to host memory (see the CUDA programming guide for more details).

**Performance Considerations**

The APIs provided below should be efficient on all hardware sm\_20 and greater. Maxwell performance in current releases of NVCC is poor due to compiler inefficiencies. These inefficiencies have been resolved in CUDA 8.0. We suggest using CUDA 8.0 or newer if you are are running on Maxwell or Pascal. If you are running with an older toolkit then build the library with the flag –DIMAD. This will provided greater performance on Maxwell but will not achieve the highest level of performance that you would see with CUDA 8.0.

Performance at different sizes can vary significantly. For maximum performance users should test multiple batch sizes and pick one that yields the best performance.

**Data Types**

XMP currently defines the following new data types. They are described below:

***xmpError\_t***: An error code that is returned from API calls. The possible error codes are as follows:

**xmpErrorSuccess**: The API call was successful and no error was detected.  
**xmpErrorInvalidParameter**: An invalid parameter was passed into an API call.  
**xmpErrorInvalidMalloc**: A system malloc failed (i.e. returned NULL).  
**xmpErrorInvalidCudaMalloc**: cudaMalloc failed.  
**xmpErrorInvalidCount**: An invalid count was passed into an API call. For example, count was larger than the output array.  
**xmpErrorInvalidDevice**: An invalid CUDA device was detected. This typically occurs when an integer array was allocated on a different device than the current handle.  
**xmpErrorInvalidFormat**: An invalid data format was encountered by a routine. This typically means an integer array was never initialized using xmpIntegersImport or by another API call which sets the integer array. **xmpErrorInvalidPrecision**: There was a precision mismatch in an operand. See the API call for restrictions on precision.  
**xmpErrorCuda**: an unspecified CUDA error has occurred. Use cudaGetLastError() for more information.  
**xmpErrorUnsupported**: The user tried to perform an unsupported operation (see restrictions for the API call for more details).  
**xmpErrorUnspecified**: An unspecified error occurred.

***xmpHandle\_t***: A handle which helps control execution of operations. All API calls take a handle as a parameter. The handle is opaque to the user.

***xmpIntegers\_t***: An array of large integers with constant sized precision. The array is opaque to the user.

***xmpExecutionPolicy\_t***: An execution policy which can control the execution behavior.

**API Guide**

const char\* ***xmpGetErrorString***(xmpError\_t e)

Converts an error code into an error string.

Returns:

A human readable error string for the error e.

Restrictions:

None

xmpError\_t ***xmpHandleCreate***(xmpHandle\_t \*handle)

Allocates and initializes *handle*. The currently active device in CUDA is stored in *handle* and all future API calls on this handle will be performed on this device regardless of what the active CUDA device is set to.

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidParameter: *handle* was NULL.  
xmpErrorInvalidMalloc: the system malloc routine returned NULL.  
xmpErrorInvalidDevice: the currently active device is not present or unsupported.

Restrictions:

Handles can only be created on CUDA architecture SM\_20 or greater.

xmpError\_t ***xmpHandleCreateWithAllocators***(xmpHandle\_t \*handle, xmpAllocFunc ha, xmpFreeFunc hf, xmpAllocFunc da, xmpFreeFunc df)

Allocates and initializes *handle*. The currently active device in CUDA is stored in *handle* and all future API calls on this handle will be performed on this device regardless of what the active CUDA device is set to.

The host allocator is set to ha, the host deallocator is set to hf, the device allocator is set to da, and the device deallocator is set to df. If a parameter to this function is NULL then the default allocator or deallocator will be used for that function.

xmpAllocFunc functions must take the form of void\* func(size\_t).

xmpFreeFunc functions must take the form of void func(void\*).

If the xmpAllocFunc returns 0 then an error XMP\_ERROR\_INVALID\_MALLOC or XMP\_ERROR\_INVALID\_CUDA\_MALLOC will be returned by the calling API call.

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidParameter: *handle* was NULL.  
xmpErrorInvalidMalloc: the system malloc routine returned NULL.  
xmpErrorInvalidDevice: the currently active device is not present or unsupported.

Restrictions:

Handles can only be created on CUDA architecture SM\_30 or greater.

xmpError\_t ***xmpHandleDestroy***(xmpHandle\_t handle)

Deallocates all memory associated with the *handle*. This does not include memory allocated in other API calls like xmpIntegersCreate. After this point the handle is no longer considered valid.

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.

Restrictions:

None

xmpError\_t **xmpHandleGetMemoryFunctions**(xmpHandle\_t handle, xmpAllocFunc \*ha, xmpFreeFunc \*hf, xmpAllocFunc \*da, xmpFreeFunc \*df);

Sets *ha* to the host allocation function, *hf* to the host deallocation function, *da* to the device allocation function, and *df* to the device deallocation function. If NULL is passed into any parameter than that function will not be returned.

xmpAllocFunc functions must take the form of void\* func(size\_t).

xmpFreeFunc functions must take the form of void func(void\*).

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.

Restrictions:

None

xmpError\_t ***xmpHandleSetStream***(xmpHandle\_t handle, cudaStream\_t s)

Sets the execution stream for *handle* to the stream passed in *s*.

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.

Restrictions:

The device that *s* was allocated on must match the device associated with *handle*.

xmpError\_t ***xmpHandleGetStream***(xmpHandle\_t *handle*, cudaStream\_t \*s)

Sets *s* to the stream associated with *handle*.

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidParamter: *s* was NULL.

Restrictions:

None

xmpError\_t ***xmpHandleGetDevice***(xmpHandle\_t handle, int32\_t \*d)

Sets *d* to the device associated with *handle*.

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidParamter: *d* was NULL.

Restrictions:

None

xmpError\_t ***xmpHandleSetExecutionPolicy***(xmpHandle\_t handle, xmpExecutionPolicy\_t policy)

Sets the execution *policy* associated with *handle*. If *policy* is null the default policy will be used.

Returns:

xmpErrorSuccess: no errors were detected.

Restrictions:

None

xmpError\_t ***xmpExecutionPolicyCreate***(xmpHandle\_t handle, xmpExecutionPolicy\_t \*policy)

Allocates an execution policy with default settings.

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidParamter: *policy* was NULL  
xmpErrorInvalidMalloc: the system malloc routine returned NULL.

Restrictions:

*None*

xmpError\_t ***xmpExecutionPolicyDestroy***(xmpHandle\_t handle, xmpExecutionPolicy\_t policy)

Deletes an execution policy.

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.

Restrictions:

*None*

xmpError\_t ***xmpExecutionPolicySetIndices***(xmpHandle\_t handle, xmpExecutionPolicy\_t policy, uint32\_t which, uint32\_t \*indices, uint32\_t count)

xmpError\_t ***xmpExecutionPolicySetIndicesAsync***(xmpHandle\_t handle, xmpExecutionPolicy\_t policy, uint32\_t which, uint32\_t \*indices, uint32\_t count)

Sets an indexing array for the execution policy. Each execution policy can specify up to four arrays which correspond to the four integer arrays in an API call. The parameter *which* specifies which indexing array is being set. The first integer parameter to appear in the corresponding API call will use *which*=0, the second will use *which*=1, and so on. Currently *which* cannot be larger than 3.

*Indices* is an array of length *count* which specifies the indexing array. The value at each element of this array specifies which index into the integers array will be used for each instance. For example instance i will use element indices[i] of the corresponding integer array.

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidParameter:  *which* was too large.  
xmpErrorInvalidCudaMalloc: cudaMalloc returned an error.

Restrictions

None

xmpError\_t ***xmpIntegersCreate***(xmpHandle\_t handle, xmpIntegers\_t \*x, uint32\_t p, uint32\_t c)

Allocates an array with of integers *x*. Where count(*x*)=c and precision(*x*)= *p*. Memory is allocated on the device associated with *handle*. The integers are unsigned and can store a value between [0,2^precision(*x*)).

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidParamter: *x* was NULL  
xmpErrorInvalidMalloc: the system malloc routine returned NULL.  
xmpErrorInvalidCudaMalloc: cudaMalloc returned an error.  
xmpErrorUnsupported: The precision was not divisible by 32.

Restrictions:

*p* must be divisible by 32.

xmpError\_t ***xmpIntegersDestroy***(xmpHandle\_t handle, xmpIntegers\_t x)

Frees the memory for the array of integers stored in *x*. After this call *x* is considered invalid.

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.

Restrictions

None

xmpError\_t ***xmpIntegersGetPrecision***(xmpHandle\_t handle, xmpIntegers\_t x, uint32\_t \*p)

Sets *p* to the precision of integers in *x.*

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidParameter: p was NULL.

Restrictions:

None

xmpError\_t ***xmpIntegersGetCount***(xmpHandle\_t h, xmpIntegers\_t x, uint32\_t \*c)

Sets *c* to the count of integers in *x.*

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidParameter: *c* was NULL.

Restrictions:

None

**Data Movement Routines**

The following describes the data movement routines which can move data into and out of the library. All pointers passed into these routines can exist in host memory or device memory. In order for asynchronous routines to execute asynchronously any host memory must be pinned to host memory (see the CUDA programming guide for more details). All integer arrays follow the indexing specified in the execution policy. With this elements can be reordered, duplicated, or filtered on import, export, and copy.

xmpError\_t ***xmpIntegersImport***(xmpHandle\_t handle, xmpIntegers\_t output, uint32\_t words, int32\_t order, size\_t size, int32\_t endian, uint32\_t nails, void\* input, uint32\_t count)  
xmpError\_t ***xmpIntegersImportAsync***(xmpHandle\_t handle, xmpIntegers\_t output, uint32\_t words, int32\_t order, size\_t size, int32\_t endian, uint32\_t nails, void\* input, , uint32\_t count)

Set *output* from an array of word data in *input*. The parameters specify the format of the word data. *count* is the number of integers, words is the number of words in each integer. *order* can be -1 for least significant words first and 1 for most significant words first.  *size* is the size of each word in bytes and *endian* specifies the endianess of each word (-1 little endian, 0 host native, and 1 big endian). The most significant *nails* bit of each word are set to zero, this can be 0 for full words.

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidParameter:  *input* was NULL, count was *invalid*, *size* was invalid, *order* was invalid, words was *invalid*, or *endian* was invalid.  
xmpErrorInvalidPrecision: the precision of out is not equal to words\*size  
xmpErrorInvalidDevice: the device associated with *handle* and the device associated with *output* do not match.

Restrictions

The device associated with *handle* must be the same device that was associated with *output* at allocation. *size* must be 1, 2, 4, or 8.

xmpError\_t ***xmpIntegersExport***(xmpHandle\_t handle, void\* output, uint32\_t \*words, int32\_t order, size\_t size, int32\_t endian, uint32\_t nails, xmpIntegers input, uint32\_t count)  
xmpError\_t ***xmpIntegersExportAsync***(xmpHandle\_t handle, void\* output, uint32\_t words, int32\_t order, size\_t size, int32\_t endian, uint32\_t nails, xmpIntegers input, uint32\_t count)

Set an array of word data *output* from *input*. The parameters specify the format of the word data. *count* is the number of integers. *order* can be -1 for least significant words first and 1 for most significant words first.  *size* is the size of each word in bytes and *endian* specifies the endianess of each word (-1 little endian, 0 host native, and 1 big endian). The most significant *nails* bit of each word are ignored set to zero, this can be 0 for full words. If *words* is not equal to NULL then it will be set to the number of words written.

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidParameter:  *input* was NULL, count was *invalid*, *size* was invalid, *order* was invalid, words was *invalid*, or *endian* was invalid.  
xmpErrorInvalidDevice: the device associated with *handle* and the device associated with *output* do not match.

Restrictions

The device associated with *handle* must be the same device that was associated with *input* at allocation. *size* must be 1, 2, 4, or 8.

xmpError\_t ***xmpIntegersSet***(xmpHandle\_t handle, xmpIntegers\_t output, xmpIntegers\_t input, uint32\_t count)  
xmpError\_t ***xmpIntegersSetAsync***(xmpHandle\_t handle, xmpIntegers\_t output, xmpIntegers\_t input, uint32\_t count)

Copies count integers from array i*nput* to array *output*. The buffers *input* and *output* do not need to reside on the same GPU.

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidDevice: the device associated with *handle* and the device associated with one of *output* or *input* do not match.  
xmpErrorInvalidFormat: an invalid format was encountered. This means that *input* was uninitialized.  
xmpErrorInvalidPrecision: the precision of *input* and *output* do not match.  
xmpErrorInvalidCount: the count of *input* or *output* is less than *count*.

Restrictions

The device associated with the handle must be the same device that was used to allocate either *input* or *output*.   
The precision of *input* and *output* must match.

**Math Operators**

All operators take the form of xmpError\_t xmpIntegersOp(xmpHandle\_t, …, uint32\_t instance). All integer arrays follow the indexing specified in the execution policy.

xmpError\_t **xmpIntegersAdd**(xmpHandle\_t handle, xmpIntegers\_t c, const xmpIntegers\_t a, const xmpIntegers\_t b, uint32\_t instances)  
xmpError\_t **xmpIntegersAddAsync**(xmpHandle\_t handle, xmpIntegers\_t c, const xmpIntegers\_t a, const xmpIntegers\_t b, uint32\_t instances)

Computes c=a+b for the *instance* integers in the arrays *a* and *b*. If (a+b)>2^precision(c) the result wraps such that c=(a+b)-2^precision(c).  
Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidDevice: the device associated with handle and the device associated with any of *c, a*, or *b* do not match.  
xmpErrorInvalidCount: The count(*c*) is less than *instances*.  
xmpErrorInvalidPrecision: The precision of *c* is less than the maximum precision of *a* and *b*.  
xmpErrorUnsupported: an unsupported combination of count or precision was detected.

Restrictions

The device associated with *handle* must be the same device associated with all input and output arrays.   
The precision of *c* must be greater than or equal to the maximum precision of *a* and *b*.  
count(*c*) must be greater than *instances*.

xmpError\_t **xmpIntegersSub**(xmpHandle\_t handle, xmpIntegers\_t c, const xmpIntegers\_t a, const xmpIntegers\_t b, uint32\_t instances)  
xmpError\_t **xmpIntegersSubAsync**(xmpHandle\_t handle, xmpIntegers\_t c, const xmpIntegers\_t a, const xmpIntegers\_t b, uint32\_t instances)

Computes c=a-b for the *instance* integers in the arrays *a* and *b*. If a<b the result wraps such that c=a-b+2^precision(c).  
Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidDevice: the device associated with handle and the device associated with any of *c, a*, or *b* do not match.  
xmpErrorInvalidCount: The count(*c*) is less than *instances*.  
xmpErrorInvalidPrecision: The precision of *c* is less than the maximum precision of *a* and *b*.  
xmpErrorUnsupported: an unsupported combination of count or precision was detected.

Restrictions

The device associated with *handle* must be the same device associated with all input and output arrays.   
The precision of *c* must be greater than or equal to the maximum precision of *a* and *b*.  
count(*c*) must be greater than *instances*.

xmpError\_t **xmpIntegersMul**(xmpHandle\_t handle, xmpIntegers\_t c, const xmpIntegers\_t a, const xmpIntegers\_t b, uint32\_t instances)  
xmpError\_t **xmpIntegersMulAsync**(xmpHandle\_t handle, xmpIntegers\_t c, const xmpIntegers\_t a, const xmpIntegers\_t b, uint32\_t instances)

Computes c=a\*b for the *instance* integers in the arrays *a* and *b*. If precision(a)+precision(b)>precision(c) only the lower precision(*c*) bits of the result are returned.  
  
Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidDevice: the device associated with handle and the device associated with any of *c, a*, or *b* do not match.  
xmpErrorInvalidCount: The count(*c*) is less than *instances*.  
xmpErrorInvalidPrecision: The precision of *c* is less than the maximum precision of *a* and *b*.  
xmpErrorUnsupported: an unsupported combination of count or precision was detected.

Restrictions

The device associated with *handle* must be the same device associated with all input and output arrays.   
The precision of *c* must be greater than or equal to the maximum precision of *a* and *b*.  
count(*c*) must be greater than *instances*.

xmpError\_t **xmpIntegersDiv**(xmpHandle\_t handle, xmpIntegers\_t c, const xmpIntegers\_t a, const xmpIntegers\_t b, uint32\_t instances)  
xmpError\_t **xmpIntegersDivAsync**(xmpHandle\_t handle, xmpIntegers\_t c, const xmpIntegers\_t a, const xmpIntegers\_t b, uint32\_t instances)

Computes c=floor(a/b) for the *instance* integers in the arrays *a* and *b*

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidDevice: the device associated with handle and the device associated with any of *c, a*, or *b* do not match.  
xmpErrorInvalidCount: The count(*c*) is less than *instances*.  
xmpErrorInvalidPrecision: The precision(*c*) is less than the precision(*a*).  
xmpErrorUnsupported: an unsupported combination of count or precision was detected.

Restrictions

The device associated with *handle* must be the same device associated with all input and output arrays.   
The precision of *c* must be greater than or equal to the precision of *a*.  
count(*c*) must be greater than *instances*.

xmpError\_t **xmpIntegersMod**(xmpHandle\_t handle, xmpIntegers\_t c, const xmpIntegers\_t a, const xmpIntegers\_t b, uint32\_t instances)  
xmpError\_t **xmpIntegersModAsync**(xmpHandle\_t handle, xmpIntegers\_t c, const xmpIntegers\_t a, const xmpIntegers\_t b, uint32\_t instances)

Computes c=a%b for the *instance* integers in the arrays *a* and *b*

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidDevice: the device associated with handle and the device associated with any of *c, a*, or *b* do not match.  
xmpErrorInvalidCount: The count(*c*) is less than *instances*.  
xmpErrorInvalidPrecision: The precision(*c*) is less than the precision(*b*).  
xmpErrorUnsupported: an unsupported combination of count or precision was detected.

Restrictions

The device associated with *handle* must be the same device associated with all input and output arrays.   
The precision of *c* must be greater than or equal to the precision of *b*.  
count(*c*) must be greater than *instances*.

xmpError\_t **xmpIntegersDivMod**(xmpHandle\_t handle, xmpIntegers\_t c, xmpIntegers\_t d, const xmpIntegers\_t a, const xmpIntegers\_t b, uint32\_t instances)  
xmpError\_t **xmpIntegersDivModAsync**(xmpHandle\_t handle, xmpIntegers\_t c, xmpIntegers\_t d, const xmpIntegers\_t a, const xmpIntegers\_t b, uint32\_t instances)

Computes c=floor(a/b) and d=a%b for the *instance* integers in the arrays *a* and *b*

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidDevice: the device associated with handle and the device associated with any of *c, a*, *b,* or *d* do not match.  
xmpErrorInvalidCount: The count(*c*) or count(*d*) is less than *instances*.  
xmpErrorInvalidPrecision: The precision(*c*) is less than the precision(*a*) or the precision(*d*) is less than the precision(*b*).  
xmpErrorUnsupported: an unsupported combination of count or precision was detected.

Restrictions

The device associated with *handle* must be the same device associated with all input and output arrays.   
The precision of *c* must be greater than or equal to the precision of *a*.  
The precision of *d* must be greater than or equal to the precision of *b.*  
count(*c*) and count(*d*) must be greater than *instances*.

xmpError\_t **xmpIntegersPowm**(xmpHandle\_t handle, xmpIntegers\_t o, const xmpIntegers\_t b, const xmpIntegers\_t e, const xmpIntegers\_t m, uint32\_t instances)  
xmpError\_t **xmpIntegersPowmAsync**(xmpHandle\_t handle, xmpIntegers\_t o, const xmpIntegers\_t b, const xmpIntegers\_t e, const xmpIntegers\_t m, uint32\_t instances)

Computes *o*=*b*^*e* % *m* for the *instance* integers in the arrays *o*,*b*,e, and *m*. count(*o*) must be greater than *instances*. Only odd moduli are supported if a modulus is even then the return value is undefined.

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidDevice: the device associated with handle and the device associated with any of *o*, *b*, *m*, or *e* do not match.  
xmpErrorInvalidCount: The count(*o*) is less than *instances*.  
xmpErrorInvalidPrecision: The precision of *o*, *a* and *m* do not match.  
xmpErrorUnsupported: an unsupported combination of count or precision was detected.

Restrictions

The device associated with *handle* must be the same device associated with all input and output arrays. The precision of all *o*, *a* and *m* must match. The modulus must be odd.

**Bit and Logical Operators**

xmpError\_t **xmpIntegersCmp**(xmpHandle\_t handle, int32\_t \*c, const xmpIntegers\_t a, const xmpIntegers\_t b, uint32\_t instances)  
xmpError\_t **xmpIntegersCmpAsync**(xmpHandle\_t handle, int32\_t \*c, const xmpIntegers\_t a, const xmpIntegers\_t b, uint32\_t instances)

Computes c=cmp(a,b) for the *instance* integers in the arrays *a* and *b*. Where cmp returns -1 if a<b, +1 if b>a, and 0 if a==b.

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidDevice: the device associated with handle and the device associated with any of *a* or *b* do not match.

Restrictions

The device associated with *handle* must be the same device associated with all input arrays.

xmpError\_t **xmpIntegersPopc**(xmpHandle\_t handle, uint32\_t \*c, const xmpIntegers\_t a, uint32\_t instances)  
xmpError\_t **xmpIntegersPopcAsync**(xmpHandle\_t handle, uint32\_t \*c, const xmpIntegers\_t a, uint32\_t instances)

Computes c=popc(a) for the *instances* integers in the array *a*. Where popc returns the number of set bits.

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidCount: count(*a*) is less than *instances*.  
xmpErrorInvalidDevice: the device associated with handle and the device associated with any of *a* or *b* do not match.

Restrictions

The device associated with *handle* must be the same device associated with all input arrays.

xmpError\_t **xmpIntegersIor**(xmpHandle\_t handle, xmpIntegers c, const xmpIntegers\_t a, const xmpIntegers\_t b, uint32\_t instances)  
xmpError\_t **xmpIntegersIorAsync**(xmpHandle\_t handle, xmpIntegers c, const xmpIntegers\_t a, const xmpIntegers\_t b, uint32\_t instances)

Computes c=a|b for the *instances* integers in the arrays *a* and *b*. Where | is the bitwise inclusive-or operation.

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidCount: the count(*c*) is less than instances.  
xmpErrorInvalidPrecision: the precision(c) is less than the max precision of a and b.  
xmpErrorInvalidDevice: the device associated with handle and the device associated with any of *a* or *b* do not match.

Restrictions

The device associated with *handle* must be the same device associated with all input and output arrays.

xmpError\_t **xmpIntegersXor**(xmpHandle\_t handle, xmpIntegers c, const xmpIntegers\_t a, const xmpIntegers\_t b, uint32\_t instances)  
xmpError\_t **xmpIntegersXorAsync**(xmpHandle\_t handle, xmpIntegers c, const xmpIntegers\_t a, const xmpIntegers\_t b, uint32\_t instances)

Computes c=a^b for the *instances* integers in the arrays *a* and *b*. Where ^ is the bitwise exclusive-or operation.

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidCount: the count(*c*) is less than instances.  
xmpErrorInvalidPrecision: the precision(c) is less than the max precision of a and b.  
xmpErrorInvalidDevice: the device associated with handle and the device associated with any of *a* or *b* do not match.

Restrictions

The device associated with *handle* must be the same device associated with all input and output arrays.

xmpError\_t **xmpIntegersAnd**(xmpHandle\_t handle, xmpIntegers c, const xmpIntegers\_t a, const xmpIntegers\_t b, uint32\_t instances)  
xmpError\_t **xmpIntegersAndAsync**(xmpHandle\_t handle, xmpIntegers c, const xmpIntegers\_t a, const xmpIntegers\_t b, uint32\_t instances)

Computes c=a&b for the *instances* integers in the arrays *a* and *b*. Where & is the bitwise and operation.

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidCount: the count(*c*) is less than instances.  
xmpErrorInvalidPrecision: the precision(c) is less than the max precision of a and b.  
xmpErrorInvalidDevice: the device associated with handle and the device associated with any of *a* or *b* do not match.

Restrictions

The device associated with *handle* must be the same device associated with all input and output arrays.

xmpError\_t **xmpIntegersNot**(xmpHandle\_t handle, xmpIntegers c, const xmpIntegers\_t a, uint32\_t instances)  
xmpError\_t **xmpIntegersNotAsync**(xmpHandle\_t handle, xmpIntegers c, const xmpIntegers\_t a, uint32\_t instances)

Computes c=!a for the *instances* integers in the array *a*. Where ! is the bitwise not operation.

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidCount: the count(*c*) is less than instances.  
xmpErrorInvalidPrecision: the precision(c) is less than the max precision of a.  
xmpErrorInvalidDevice: the device associated with handle and the device associated with *a* do not match.

Restrictions

The device associated with *handle* must be the same device associated with all input and output arrays.

xmpError\_t **xmpIntegersShf**(xmpHandle\_t handle, xmpIntegers c, const xmpIntegers\_t a, const int32\_t\* shift, const uint32\_t shift\_count, uint32\_t instances)  
xmpError\_t **xmpIntegersShfAsync**(xmpHandle\_t handle, xmpIntegers c, const xmpIntegers\_t a, const int32\_t\* sh, const uint32\_t sh\_count, uint32\_t instances)

Computes *c*=shift(*a*,*sh*) for the *instances* integers in the array *a*. Where shift(*a*,*sh*) shifts the bits of a left or right based on *sh*. If *sh* is > 0 then *a* is shifted left *sh* bits otherwise *a* is shifted right *sh* bits. If *sh\_count* < *instances* then elements of *sh* are repeated in a round-robin fashion.

Returns:

xmpErrorSuccess: no errors were detected.  
xmpErrorCuda: a CUDA error was returned.  
xmpErrorInvalidParameter: *sh* is NULL  
xmpErrorInvalidCount: the count(*c*) is less than instances.  
xmpErrorInvalidDevice: the device associated with handle and the device associated with *a* do not match.

Restrictions

The device associated with *handle* must be the same device associated with all input and output arrays.

**Example Program**

**#include "xmp.h"**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <cuda\_runtime\_api.h>**

**#define XMP\_CHECK\_ERROR(fun) \**

**{ \**

**xmpError\_t error=fun; \**

**if(error!=xmpErrorSuccess){ \**

**if(error==xmpErrorCuda) \**

**printf("CUDA Error %s, %s:%d\n",cudaGetErrorString(cudaGetLastError()),\_\_FILE\_\_,\_\_LINE\_\_); \**

**else \**

**printf("XMP Error %s, %s:%d\n",xmpGetErrorString(error),\_\_FILE\_\_,\_\_LINE\_\_); \**

**exit(EXIT\_FAILURE); \**

**} \**

**}**

**int main() {**

**int i,w;**

**int N=10000;**

**int bits=1024;**

**xmpIntegers\_t base, mod, exp, out;**

**uint32\_t \*b,\*m,\*e,\*o;**

**uint32\_t limbs=bits/8/sizeof(uint32\_t);**

**size\_t bytes=N\*bits/8;**

**b=(uint32\_t\*)malloc(bytes);**

**o=(uint32\_t\*)malloc(bytes);**

**m=(uint32\_t\*)malloc(bits/8);**

**e=(uint32\_t\*)malloc(bits/8);**

**xmpHandle\_t handle;**

**//allocate handle**

**XMP\_CHECK\_ERROR(xmpHandleCreate(&handle));**

**//allocate integers**

**XMP\_CHECK\_ERROR(xmpIntegersCreate(handle,&base,bits,N));**

**XMP\_CHECK\_ERROR(xmpIntegersCreate(handle,&out,bits,N));**

**XMP\_CHECK\_ERROR(xmpIntegersCreate(handle,&exp,bits,1));**

**XMP\_CHECK\_ERROR(xmpIntegersCreate(handle,&mod,bits,1));**

**//initialize base, exp, and mod**

**for(i=0;i<N;i++) {**

**for(w=0;w<limbs;w++) {**

**b[i\*limbs+w]=rand();**

**}**

**}**

**for(w=0;w<limbs;w++) {**

**m[w]=rand();**

**e[w]=rand();**

**}**

**//make sure modulus is odd**

**m[0]|=1;**

**//import**

**XMP\_CHECK\_ERROR(xmpIntegersImport(handle,base,N,limbs,-1,sizeof(uint32\_t),0,0,b));**

**XMP\_CHECK\_ERROR(xmpIntegersImport(handle,exp,1,limbs,-1,sizeof(uint32\_t),0,0,e));**

**XMP\_CHECK\_ERROR(xmpIntegersImport(handle,mod,1,limbs,-1,sizeof(uint32\_t),0,0,m));**

**//call powm**

**XMP\_CHECK\_ERROR(xmpIntegersPowm(handle,out,base,exp,mod,N));**

**//export**

**XMP\_CHECK\_ERROR(xmpIntegersExport(handle,o,N,&limbs,-1,sizeof(uint32\_t),0,0,out));**

**//use results here**

**//free integers**

**XMP\_CHECK\_ERROR(xmpIntegersDestroy(handle,base));**

**XMP\_CHECK\_ERROR(xmpIntegersDestroy(handle,out));**

**XMP\_CHECK\_ERROR(xmpIntegersDestroy(handle,exp));**

**XMP\_CHECK\_ERROR(xmpIntegersDestroy(handle,mod));**

**//free handle**

**XMP\_CHECK\_ERROR(xmpHandleDestroy(handle));**

**free(b);**

**free(o);**

**free(m);**

**free(e);**

**printf("done\n");**

**return 0;**

**}**