oneMKL Technical Advisory Board

Session 8

November 11, 2020

Agenda

- Welcoming remarks – 5 minutes
- Updates from last meeting – 5 minutes
- Overview of oneMKL Vector Math domain - Andrey Stepin (30 minutes)
- Wrap-up and next steps – 5 minutes

Updates from last meeting

- oneAPI Developer Summit 2020: Nov. 12-13, 2020
  - Register: https://webinar.intel.com/oneAPIDeveloperSummit2020
- oneMKL TAB meeting
  - Calendar series ends with today's meeting
  - Planning for a December meeting
- oneMKL specification v. 1.0 released!
- oneMKL open source interfaces https://github.com/oneapi-src/oneMKL
  - RNG domain is available
  - Netlib backend for BLAS is available

Overview of oneMKL Vector Math (VM) domain

VM Structure (classic/OpenMP offload)

Vector math functions

- Supported languages
  - C/C++
  - Fortran
- Precisions
  - float
  - double
  - MKL_Complex8
  - MKL_Complex16
- Accuracies
  - HA (high accuracy = 1 ulp)
  - LA (low accuracy = 4 ulp)
  - EP (enhanced performance = half of mantissa bits of the result is correct)
- Target devices
  - x86 CPUs
  - Intel GPUs: OpenMP offload only

vsExp

vsSin

vsPow

vsDiv

vsMul

…

Service functions

- Computation mode control
  - accuracy (HA, LA, EP)
  - FTZ/DAZ
  - Precision & rounding mode
  - OpenMP threading settings
  - Error reporting
- Global error status access
- Error handler control

vmlGetMode

vmlSetMode

vmlGetErrStatus

vmlSetErrStatus

vmlClearErrStatus

vmlGetErrorCallBack

vmlErrorCallBack

vmlClearErrorCallBack

VM Structure (DPC++)

Free functions

Vector math functions

- Precisions
  - float
  - double
  - std::complex<float>
  - std::complex<double>
- Accuracies
  - HA (high accuracy = 1 ulp)
  - LA (low accuracy = 4 ulp)
  - EP (enhanced performance = half of mantissa bits of the result is correct)
- Target devices
  - x86 CPUs
  - optimized for Intel GPUs

oneapi::mkl::vm::exp

oneapi::mkl::vm::sin

oneapi::mkl::vm::pow

oneapi::mkl::vm::div

oneapi::mkl::vm::mul

...

Service functions

- Computation mode control
  - accuracy (HA, LA, EP)
  - global status reporting
- Global error status access

oneapi::mkl::vm::get_mode

oneapi::mkl::vm::set_mode

oneapi::mkl::vm::get_status

oneapi::mkl::vm::set_status

oneapi::mkl::vm::clear_status

Classes

Status code handler

```
template <typename T>
 struct  oneapi::mkl::vm::error_handler
 {
   error_handler( ) { .... }

   error_handler ( vm::status status_to_fix,
                   T fixup_value,
                   bool copy_sign = false) { ... }

   error_handler ( vm::status * array,
                   std::int64_t len = 1,...) { ... }

   error_handler ( sycl::buffer<one_vm::status, 1> & buf,
                   std::int64_t len = 1, ...)   { ... }
 }
```

- Report global status
- Local function aggregate status code
- Array of local function status codes for elements
- Fixup result

Note: all subsequent slides imply using namespace ***oneapi::mkl***

VM APIs

Classic

simple:

```
vsExp (MKL_INT     n,
       const float arg[],
       float       res[]);
```

with local mode:

```
vmsExp (MKL_INT     n,
        const float arg[],
        float       res[],
        MKL_INT64   mode);
```

OpenMP Offload

```
#pragma omp target data map(to:arg[0:n]) map(tofrom:res[0:n]) device(dev)
#pragma omp target variant dispatch device(dev) use_device_ptr(arg, res)
```

```
vsExp (MKL_INT     n,
       const float arg[],
       float       res[]);
```

```
vmsExp (MKL_INT     n,
        const float arg[],
        float       res[],
        MKL_INT64   mode);
```

DPC++ SYCL buffer

```
sycl::event exp(  sycl::queue &                 q,
                  int64_t                       n,
                  sycl::buffer<float> &         arg,
                  sycl::buffer<float> &         res,
                  vm::mode                      mode,
                  vm::error_handler<float>      handler);
```

DPC++ USM

```
sycl::event exp    (sycl::queue &                                q,
                     int64_t                                     n,
                     float *                                     arg,
                     float *                                     res,
                     sycl::vector_class<sycl::event> const &     deps,
                     vm::mode                                    mode,
                     vm::error_handler<float>                    handler);
```

yellow: optional C++ parameters with default values

VM Usage Models (classic/OpenMP offload)

C

```
#include "mkl_omp_offload.h"
#include “mkl_vml.h"

float a[N];
float y[N];

#pragma omp target data map(to:a[0:N]) map(tofrom:y[0:N])\
                                                  device(dev)
{
  #pragma omp target variant dispatch device(dev) \
                                use_device_ptr(a, y)
  {
    vmsExp(N, a, y, VML_LA);
  }
}
```

OpenMP offload directives to add

Fortran

```
include "mkl_omp_offload.f90“
include "mkl_vml.f90"

real      (kind=4) :: a(:)
real      (kind=4) :: y(:)

!$omp target data map(a,y)
!$omp target variant dispatch use_device_ptr(a,y)

call vmsexp(N, a, y, VML_LA)

!$omp end target variant dispatch
!$omp end target data
```

VM Usage Models (DPC++ simple)

y = exp(a): buffer API

```
#include <CL/sycl.hpp>
#include “oneapi/mkl/vm.hpp“

sycl::queue queue;
std::vector<float> a;
std::vector<float> y;

{
    sycl::buffer<float> buf_a (a.begin(),
                               a.end());
    sycl::buffer<float> buf_y (y.data(),
                               y.size());
    vm::set_mode(vm::mode::la);

    vm::exp(queue, a.size(), buf_a,
                             buf_y);
}
```

vm::exp actual computation will be completed in buf_y destructor

y = exp(a): USM API with host memory

```
#include <CL/sycl.hpp>
#include “oneapi/mkl/vm.hpp"

sycl::queue queue;
std::vector<float> a;
std::vector<float> y;

vm::set_mode(vm::mode::la);

vm::exp(queue, a.size(), a.data(),
                         y.data());

// y.data() has computational result
```

global VM mode

VM completes computation automatically if destination pointer is **host-based**: (USM “host”, C++ heap or stack). Internal memcpy is performed **from** and **to** device, if needed.

y = exp(a): USM API with shared memory

```
#include <CL/sycl.hpp>
#include “oneapi/mkl/vm.hpp“

sycl::queue queue;
int64_t n = 1024;

usm_a = sycl::malloc_shared<float>(n,
queue);
usm_y = sycl::malloc_shared<float>(n,
queue);

vm::exp(queue, n, usm_a, usm_y, {},
                          vm::mode::la);
queue.wait();
sycl::free(usm_y, queue);
sycl::free(usm_a, queue);
```

explicit mode for this call

Need to synchronize and complete the computation

VM Usage Models (DPC++ asynchronous)

$y = \sin^2(a) + \cos^2(a)$: buffer API

```
#include <CL/sycl.hpp>
#include “oneapi/mkl/vm.hpp"

sycl::queue queue;
std::vector<float> a;
std::vector<float> y;

{
    sycl::buffer<double> buf_a  (a.begin(), a.end());
    sycl::buffer<double> buf_t  (a.begin(), a.end());
    sycl::buffer<double> buf_y  (y.data(), y.size());

    vm::sin(queue, a.size(), buf_a, buf_a);
    vm::cos(queue, a.size(), buf_t, buf_t);
    vm::sqr(queue, a.size(), buf_a, buf_a);
    vm::sqr(queue, a.size(), buf_t, buf_t);
    vm::add(queue, a.size(), buf_a, buf_t, buf_y);
}
```

in-place calls supported by VM

Buffer dependencies are maintained automatically by SYCL

$y = \sin^2(a) + \cos^2(a)$: USM API with host memory

```
#include <CL/sycl.hpp>
#include “oneapi/mkl/vm.hpp"

sycl::queue queue;
int64_t n = 1024;
float *usm_a  = sycl::malloc_shared<float>(n, queue);
float *usm_y  = sycl::malloc_shared<float>(n, queue);
float *usm_t1 = sycl::malloc_shared<float>(n, queue);
float *usm_t2 = sycl::malloc_shared<float>(n, queue);

auto ev_s = vm::sin(queue, n, usm_a, usm_t1);
auto ev_c = vm::cos(queue, n, usm_a, usm_t2);

vm::add (queue, n, usm_t1, usm_t2,
           {
             vm::sqr(queue, n, usm_t1, usm_t1, {ev_s}),
             vm::sqr(queue, n, usm_t2, usm_t2, {ev_c})
           }
        );

queue.wait();
```

Explicit dependencies are needed for USM asynchronous calls to maintain correct order

VM Error Handler

Available scenarios:

1) Read global status code for sequence of VM routines

```
exp(queue, n, a, b, mode::global_status_report);
```

```
sin(queue, n, b, r, mode::global_status_report);
```

```
auto st = get_status(queue);
```

```
if (st != status::success) {
   // do something to handle computational errors
}
```

2) Inspect local aggregate status code for each VM routine

```
exp(queue, n, a, b, mode::la, error_handler<float> { &st });
```

```
if (has_any(st, status::overflow)) {
   // do something to handle overflow in exp()
}
```

```
if (has_any(st, status::underflow)) {
   // do something to handle underflow in exp()
}
```

3) Get array of status codes for VM routine

```
exp(n, a, r, mode::la, error_handler<float> { st[], n });
```

```
// Check full array of reported status codes
for(i =0, i < n, i++) {
   if (has_any(st[i], status::overflow)) {
      r[i] = FLT_MAX; // replace INF by FLT_MAX, e.g.
   }
}
```

4) Fix up results with non-success status code by desirable value

```
exp(queue, n, a, b, mode::la,
error_handler<float> { status::overflow, FLT_MAX, true }
);
```

```
// All overflow results with produced INF
// will be replaced by fixup value FLT_MAX
// with the same sign as argument (true)
```

Note: examples above for buffer API

Future Considerations for Spec 2.0+

DPC++ USM Strided – strides added in green

```
sycl::event exp (sycl::queue &                                  q,
                 int64_t                                        n,
                 float *                                        arg,
                 int64_t                                        incr_a,
                 float *                                        res,
                 int64_t                                        incr_r,
                 sycl::vector_class<sycl::event> const &        deps,
                 vm::mode                                       mode,
                 vm::error_handler<float>                       handler);
```

DPC++ STL std::vector

```
sycl::event exp (sycl::queue &                                  q,
                 std::vector<T>                                 & arg,
                 std::vector<T>                                 & res,
                 sycl::vector_class<sycl::event> const & deps,
                 vm::mode                                       mode,
                 error_handler<T>                               handler);
```

Fusion kernel (global & retained modes)

Global for queue

```
vm::start(queue);
auto t1 = vm::temporary(queue);
vm::exp(queue, n, a, t1);
vm::ln(queue, n, t1, y);
vm::finish(queue);
```

Queue-based object

```
{
  vm::retained_mode rtm ( queue );
  auto t1 = rtm.temporary();
  vm::exp(rtm, n, a, t1, vm::mode::la);
  vm::ln(rtm, n, t1, y);
}
```

Next Steps

- Focuses for next meeting(s):
  - Sparse linear algebra
  - Discrete Fourier transforms
  - Batched linear algebra
  - Any topics from oneMKL TAB members?

Resources

- oneAPI Main Page: https://www.oneapi.com/
- Latest release of oneMKL Spec (currently v. 1.0): https://spec.oneapi.com/versions/latest/elements/oneMKL/source/index.html
- GitHub for oneAPI Spec: https://github.com/oneapi-src/oneAPI-spec
- GitHub for oneAPI TAB: https://github.com/oneapi-src/oneAPI-tab

- GitHub for open source oneMKL interfaces (currently BLAS and RNG domains): https://github.com/oneapi-src/oneMKL