Experiences Supporting DPC++ in AMReX

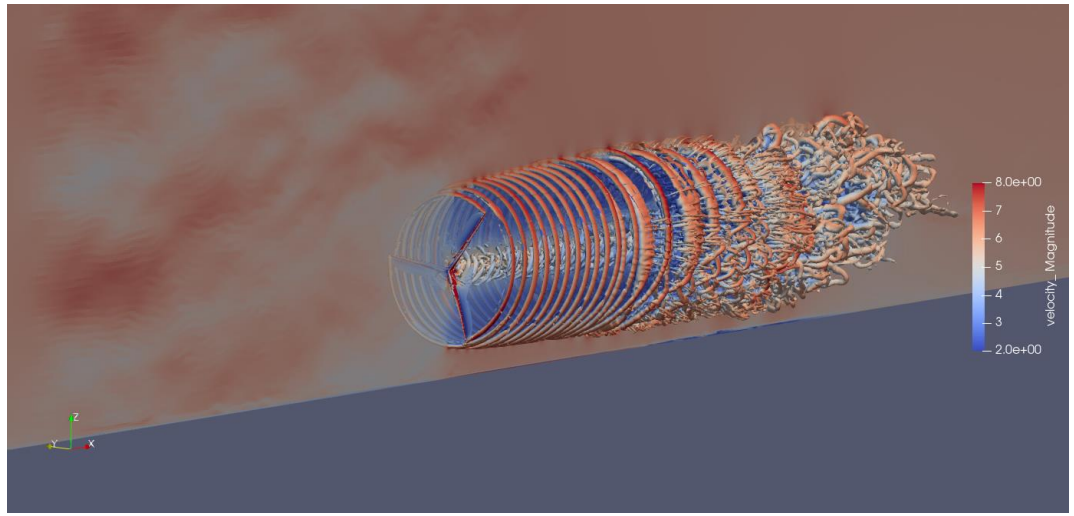
IWOCL & SYCLCon 2021

Weiqun Zhang, Kevin Gott
Lawrence Berkeley National Labs

Sravani Konda (Presenter), Dunni Aribuki, Christopher Lishka
Intel Corporation

Contents

- AMReX Overview
- DPC++ Features used in AMReX
- oneAPI/SYCL Specification and Product Influence
- Features missing in DPC++/SYCL
- Current Status
- Summary



Snapshot of the instantaneous flowfield for an NM-80 rotor using the hybrid ExaWind simulation solver suite. The image shows the tip vortices rendered using q-criterion and the contour colors show the magnitude of the velocity field.

Team: Ganesh Vijayakumar, Mike Brazell, Shreyas Ananthan, Ashesh Sharma, Jay Sitaraman, Ann Almgren, Weiqun Zhang, Mike Sprague

AMReX Overview

- Framework for building parallel, block-structured adaptive mesh refinement (AMR) applications
- Co-Design Center in the U.S. Department Of Energy's Exascale Computing Project (ECP)
- Key features:
  - Subcycling in time for time-dependent PDEs
  - Support for particles
  - Embedded boundary (EB) representations of complex geometries
  - Linear solvers
  - Parallel and asynchronous I/O
- MPI+X, where X is OpenMP, MPI, CUDA, HIP, and now DPC++
- Run on machines from laptop to supercomputers
- https://amrex-codes.github.io/amrex/

AMReX is used by numerous applications in national labs, academia and industry, including the following ECP projects

- ExaSky (cosmology)
- ExaStar (astrophysics)
- ExaWind (wind plant)
- MFiX-Exa (carbon capture)
- Pele (combustion)
- WarpX (accelerator)

AMReX is also being used to develop software tools in the greater community:

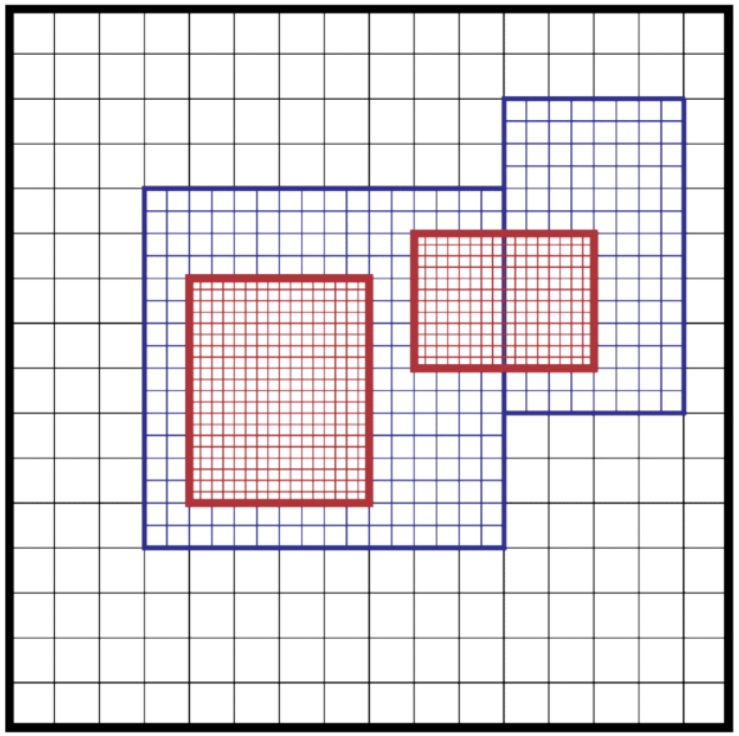
- HPCToolkit testing on Tulip and Iris
- Intel Performance Tools testing on Iris and DevCloud

Heterogeneous Computing

AMReX's main supercomputer targets:

- Different architectures usually require different programming models
- AMReX writes platform specific code using CUDA, SYCL and HIP, so that users can write optimally performant, fully portable applications

AMReX - Data Structures

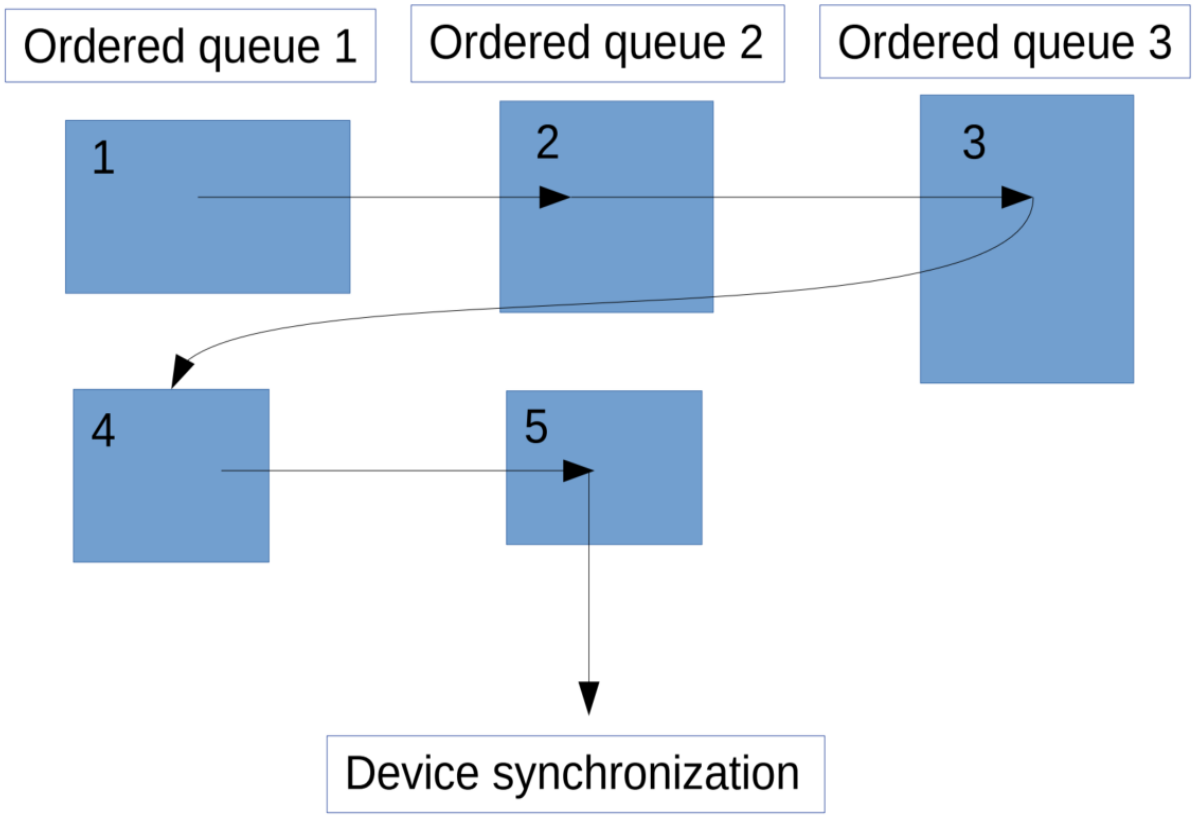


Example of three levels AMR grids

- Computational domain on each AMR level is decomposed into a union of rectangular domains
  - Hierarchy of logically rectangular grids
- Each level has Boxes: `MultiFab`
- Each Box's data are in a multi-dimensional array: `Fab`
- Non-owning accessor of Fab: `Array4`

```
template <typename T> struct Array4 {
   T* p;
   //multi-dimensional array bounds
};
```

AMReX - Iterator over Grids



- AMReX provides an iterator, `MFIter` for looping over the Boxes in MultiFabs
- Example with 5 Boxes. Kernels are launched for each Box using 3 ordered queues
- Launches in the same iteration (on the same Box) are placed in the same ordered queue, maintaining order in a simple way.
- Benefit of asynchronicity can be obtained with multiple queues

AMReX - ParallelFor

- Kernels are typically launched with `amrex::ParallelFor`
- The `Box` and `int` arguments are used to determine the number of threads to launch
- The captured `a` and `b` are trivial type containing non-owning pointers
- The memory resource management is handled by other classes using Unified Shared Memory in SYCL
- The DPC++ backend uses `sycl::parallel_for`
- `amrex::ParallelFor` is portable on CPUs and different GPU devices

```
//Launch over Box
ParallelFor(box, [=](int i, int j, int k)
{
   a(i,j,k) *= b(i,j,k);
};


//Launch over Box with components
ParallelFor(box, N,
            [=](int i, int j, int k, int n)
{
   a(i,j,k,n) *= b(i,j,k);
});


//Launch over number of elements
ParallelFor (N, [=] (int i)
{
   a[i] *= b[i];
});
```

AMReX - Reduction and Scan

- AMReX provides functionality for
  - Reduction on a `MultiFab` (a set of multi-D arrays)
  - Reduction on a `Fab` (a multi-D array)
  - Building custom reduction operations on user data structures
  - Prefix-sum functions
- DPC++ subgroup extension has allowed us to write efficient reduction and scan functions using subgroup primitives such as `shuffle_down`, `shuffle_up` and `shuffle_xor`

Note: Often need to do reduction on sections of a set of multi-dimensional arrays. Thus, the high-level SYCL Reduction APIs for 1D array do not work

```
//Min, Max and Sum reduction over a MultiFab

MultiFab mf(…);
ReduceOps<ReduceMin,ReduceMax,ReduceSum>reduce_op;
ReduceData<Real,Real,Real> reduce_data (reduce_op);
using ReduceTuple = typename decltype(reduce_data)::Type;
for (MFIter mfi(mf); mfi.isValid(); ++mfi) {
  auto const& a = mf.array();
  reduce_op.eval(mfi.box(),
   [=] (int i, int j, int k) -> ReduceTuple {
     return {a(i,j,k),a(i,j,k),a(i,j,k)};
   });
}
ReduceTuple hv = reduce_data.value();
std::cout << " Min: " << get<0>(hv)
          << " Max: " << get<1>(hv)
          << " Sum: " << get<2>(hv) << "\n";
```

AMReX Example

AMReX-based application code

```
MultiFab mfa(…), mfb(…);
for (MFIter mfi(mf); mfi.isValid(); ++mfi)
{
   auto a = mfa.array(mfi);
   auto b = mfb.array(mfi);
   amrex::ParallelFor(mfi.box(),
        [=](int i, int j, int k){
      a(i, j, k) += b(i, j, k);
   });
}
```

Code generated by DPC++ Backend

```
template <typename L>
void ParallelFor(Box const& box, L&& f) noexcept
{
   ……..
   auto& q = Gpu::Device::streamQueue();
   try {
      q.submit([&](auto &h) {
         h.parallel_for(nd_range<1>(range<1>(nthreads_total),
                                    range<1>(nthreads_per_block)),
             [=](nd_item<1> item)
             AMREX_REQUIRE_SUBGROUP_SIZE(Gpu::Device::warp_size)
             {
                …… // Calc indexing and launch the lambda.
              });
      });
    }
    catch (sycl::exception const& ex) {
           amrex::Abort(std::string("ParallelFor: ") +
                                    ex.what() + "!!!!!");
    }
……
}
```

DPC++ Features used in AMReX

- Features that were available in DPC++ and now in SYCL2020
  - Unified Shared Memory provides flexibility and fits naturally with other AMReX backends
  - In-order queues useful as a lot of operations in AMReX are naturally ordered
  - Sub-groups shuffles and collectives allows us to write custom reduction functions in an efficient way
  - Host task callback helps with memory management of temporary arrays
  - Device wide memory fence for synchronization in ParallelScan and similar funcs
- Unique DPC++/oneAPI Extensions used by AMReX
  - CXX standard library support - Enables the usage of a set of C and C++ std functions such as sqrt, fabs, sin, cos, etc., from std namespace in device code
  - Random Number Generation and Fast Fourier Transforms (via oneMKL Library) – Provides DPC++ interfaces for Uniform, Gaussian, Poisson distribution and 1,2 and 3-D FFT in device code

oneAPI/SYCL Specification Influence

- Level 0 Sysman API to query free memory on device - zesMemoryGetBandwidth
- Sub-group extension to set device's primary subgroup size attribute - SYCL_INTEL_sub_group
- Free functions to get id, item, nd_item, group, sub_group instances globally - SYCL_INTEL_free_function_queries
- Device APIs for random number generation – oneMKL Random Number Generators
- Reported a SYCL specification bug on `sycl::abs` being incompatible with C++ `std::abs` or C stdlib `abs`

Intel® oneAPI Product Influence

- Support for recursive function calls in device code
- Support for `assert()` in device code
- Increase in DPC++ kernel argument size from 1KB
- DPC++ interface to query device UUID which is available in Level0 as `ze_device_uuid_t` and OpenCL as `cl_khr_device_uuid`

Features Missing in DPC++/SYCL - 1

Local memory in device code

- Inconvenient to use local memory - requires object or pointer to be passed where local memory is used
- Workaround by creating a local accessor outside the kernel and capturing it
- Intel's extension proposal SYCL_INTEL_local_memory partially addresses the concern but still has the restriction that group-local variables must be defined at kernel functor scope

```
// Assume this function is in a header file included here.
inline void transpose (int* p)
{
    // This is what CUDA could do.
    __shared__ int shared_data[GROUP_SIZE];
    int id = threadIdx.x;
    shared_data[id] = p[id];
    __syncthreads();
    p[id] = shared_data[GROUP_SIZE-1-id];

}

int main (int argc, char* argv[])
{
    queue q;

    int* p = (int*)malloc_device(..);

    q.submit([&] (auto &h) {
        h.parallel_for(nd_range<1>(..), [=] (nd_item<1> item)
        {
            int id = item.get_global_id(0);
            p[id] = id;
        });
    });

    q.submit([&] (auto &h) {
        h.parallel_for(nd_range<1>(..), [=] (nd_item<1> item)
        {
            transpose(p);
        });
    });

    q.wait();
    free(p, q.get_context());
}
```

Features Missing in DPC++/SYCL - 2

Global Device Variables

- Many AMReX applications have the definition and declaration of the global variables spread over different files or translation units
- No support for accessing global variables in device memory and a memcpy function for copying data from host to the global variable
  - Supported by CUDA and HIP
- AMReX had to re-implement random number generation API to workaround the need for a global device variable that is implemented such that users are unaware of it

```
#include <CL/sycl.hpp>
…..
using namespace sycl;

inline __attribute__((opencl_constant)) int d_a = -1;
inline __attribute__((opencl_constant)) int d_b[] = { -1,-1,-1,-1 };

int main(int argc, char* argv[])
{
    queue q(gpu_selector{});

    int h_a = 3;
    std::vector<int> h_b{ 10,20,30,40 };
    GPU_MEMCPY_TO_SYMBOL(d_a, &h_a, sizeof(int));
    GPU_MEMCPY_TO_SYMBOL(d_b, h_b, sizeof(int)*4);

    int* data = static_cast<int*>(malloc_shared<int>(4 *
sizeof(int), q));
    {
        q.parallel_for(1, [=](id<1> i) {
            data[0] = d_a;
            data[1] = d_b[0];
            ….
            }).wait();
    }
    std::cout << data[0] << ", " << data[1] << ", " << data[2] << ",
" << data[3] << std::endl;
}
```

Current Status of AMReX DPC++ Backend

- All major AMReX capabilities - mesh, particle, embedded boundary, system linear solvers, reduction, random number generation are supported in DPC++ backend.
- Extensive testing done using existing tutorials and tests in AMReX and confirmed they work as expected
- ECP codes MFiX-Exa, WarpX, Nyx, AMR-Wind and PeleC have successfully run workloads with DPC++
- Actively working with AMReX users, ECP and the broader community to prepare to achieve scientific excellence on Aurora

Summary

- AMReX has chosen DPC++ as its backend for Intel GPUs.
- Successfully ported all major capabilities to DPC++ thanks to the power of C++, SYCL and DPC++ language extensions.
- DPC++ has enabled us to create an abstraction layer between the backend and the application. This provides existing AMReX codes with performance portability.
- Identified several limitations of DPC++ and have filed feature requests.
- We are looking forward to running on Aurora and other Intel GPU systems using DPC++ and testing cross platform capabilities on NVIDIA and AMD GPUs.
- Thanks to Intel® DevCloud and ANL's JLSE for providing resources for the development.

THANK YOU

Notices & Disclaimers

Intel technologies may require enabled hardware, software or service activation. Learn more at intel.com or from the OEM or retailer.

Your costs and results may vary.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

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. https://software.intel.com/en-us/articles/optimization-notice

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.

Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. See backup for configuration details. For more complete information about performance and benchmark results, visit www.intel.com/benchmarks.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.