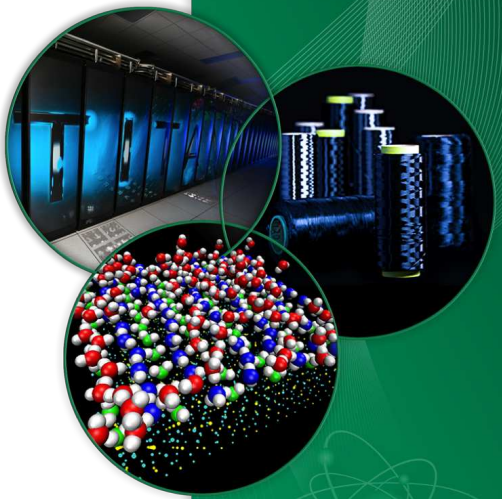
Early Experience of Application Developers with SYCL/Data Parallel C++

ECP Community BoF Days May 10, 2022 11.00 am Eastern Time

Approved for public release

Abhishek Bagusetty (ANL)
Michael D'Mello (Intel)
Daniel Arndt (ORNL)
Brian Homerding (ANL)
Brian Holland (Intel)
Thomas Applencourt (ANL)
Esteban Rangel (ANL)
Wayne Mitchell (LLNL)
Varsha Madanath (Intel)
Yasaman Ghadar (ANL)

Kokkos' Experience with SYCL



Daniel Arndt

Early Experience of Application Developers with SYCL/Data Parallel C++

OAK RIDGE National Laboratory

Feature Status

SYCL backend in Kokkos mostly feature-complete. Unimplemented

- atomics for big types (device global variables)
- WorkGraphPolicy
- Tasks
- Graphs

OAK RIDGE National Laboratory

Applications

Current features (seemingly) sufficient for a large number of ECP applications like

- `ArborX`
- `Cabana`
- `LAMMPS`
- `XGC`

OAK RIDGE National Laboratory

Missing SYCL Features

Features that are missing in `Intel`'s `SYCL` implementation, needed, e.g., for `Trilinos` and `ExaWind`:

- device global variables
- virtual functions on the device/RTTI
- querying device memory [1]
- `printf` in global namespace [2]
- better support for generic pointers

[1] Workaround kokkos-kernels#1225

[2] Workaround `KOKKOS_IMPL_DO_NOT_USE_PRINTF`

OAK RIDGE National Laboratory

Gotchas - Named kernel calls

https://github.com/kokkos/kokkos/pull/4593

- Final workgroup size must be divisible by range.
  $\Rightarrow$ `sycl::range`: wokgroup size 1 for prime number ranges
- Implict kernel range rounding only works well for named kernels. Not applicable for generic launch wrappers like

```
[=](sycl::item<1> item) {
    const typename Policy::index_type id = item.get_linear_id();
    m_functor(id);
}
```

  due to non-unique names.
- In short, use named kernel calls like

```
cgh.parallel_for<FunctorWrapper<Functor, Policy>>(range, f);
```

OAK RIDGE National Laboratory

Gotchas - Named kernel calls - AXPBY (V100)

- Unnamed call

```
cgh.parallel_for(range, f);
```

```
n = 524287: 8.868529e-01s 1.321383e+01 GB/s
n = 524288: 4.283028e-02s 2.736090e+02 GB/s
n = 524289: 3.133845e-01s 3.739423e+01 GB/s
```

- Named call

```
cgh.parallel_for<FunctorWrapper<Functor, Policy>>(range, f);
```

```
n = 524287: 4.648326e-02 s 2.521064e+02 GB/s
n = 524288: 4.585346e-02 s 2.555696e+02 GB/s
n = 524289: 4.766056e-02 s 2.458799e+02 GB/s
```

OAK RIDGE National Laboratory

Gotchas - device-copyable member variables

- `SYCL` doesn't allow arbitrary member variables, must be trivially copyable.
- `Kokkos` doesn't control functors passed in by users, even `Kokkos::View` can't be used.
- 1. Workaround: copy kernel to device explicitly.
- 2. Workaround: use `sycl::is_device_copyable`.

OAK RIDGE National Laboratory

Copy kernel explicitly

```
template <typename Functor, typename Storage>
class SYCLFunctionWrapper<Functor, Storage, false> {
  const Functor& m_kernelFunctor;

 public:
  SYCLFunctionWrapper(const Functor& functor, Storage& storage)
      : m_kernelFunctor(storage.copy_from(functor)) {}

  std::reference_wrapper<const Functor> get_functor() const {
    return {m_kernelFunctor};
  }

  static void register_event(Storage& storage, sycl::event event) {
    storage.register_event(event);
  }
};
```

OAK RIDGE National Laboratory

sycl::is_device_copyable

```
union TrivialWrapper {
  TrivialWrapper() {}
  TrivialWrapper(const Functor& f) { std::memcpy(&m_f, &f, sizeof(m_f)); }
  TrivialWrapper(const TrivialWrapper& other) {
    std::memcpy(&m_f, &other.m_f, sizeof(m_f));
  }
  TrivialWrapper& operator=(const TrivialWrapper& other) {
    std::memcpy(&m_f, &other.m_f, sizeof(m_f));
    return *this;
  }
  ~TrivialWrapper(){};

  Functor m_f;
};
```

OAK RIDGE National Laboratory

sycl::is_device_copyable II

```
template <typename Functor, typename Storage>
class SYCLFunctionWrapper<Functor, Storage, false> {
 public:
  SYCLFunctionWrapper(const Functor& functor, Storage&) :
    m_functor(functor) {}
  const Functor& get_functor() const { return m_functor.m_f; }
  sycl::event get_copy_event() const { return {}; }
  static void register_event(sycl::event) {}
};
```

OAK RIDGE National Laboratory

Gotchas - Non-device copyable member variables

1. Copy kernel to device explicitly.
   - need union to control special member functions
   - problems with `SYCL+CUDA`.
   - size restrictions
2. Use `sycl::is_device_copyable`.
   - need to manage lifetime of copied functors
   - no size restrictions

OAK RIDGE National Laboratory

Gotchas - Workgroup size `RangePolicy`

- `Kokkos` doesn't allow to explicitly set workgroup/block sizes in general
- Default workgroup sizes in general sufficiently good.
- `LaunchBounds` can be used to set
  - maximum number of threads in a block
  - minimum number of blocks in a subslice/streaming multiprocessor as optimization hints.
- kokkos#4875 allows specifying the workgroup size for a functor through `chunk_size` explicitly.

OAK RIDGE National Laboratory

Acknowledgments

- This manuscript has been authored by UT-Battelle, LLC, under Contract No. DE-AC0500OR22725 with the U.S. Department of Energy.
- This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

OAK RIDGE National Laboratory

ECP Community BOF Days
Early Experience of Application Developers with SYCL/Data Parallel C++

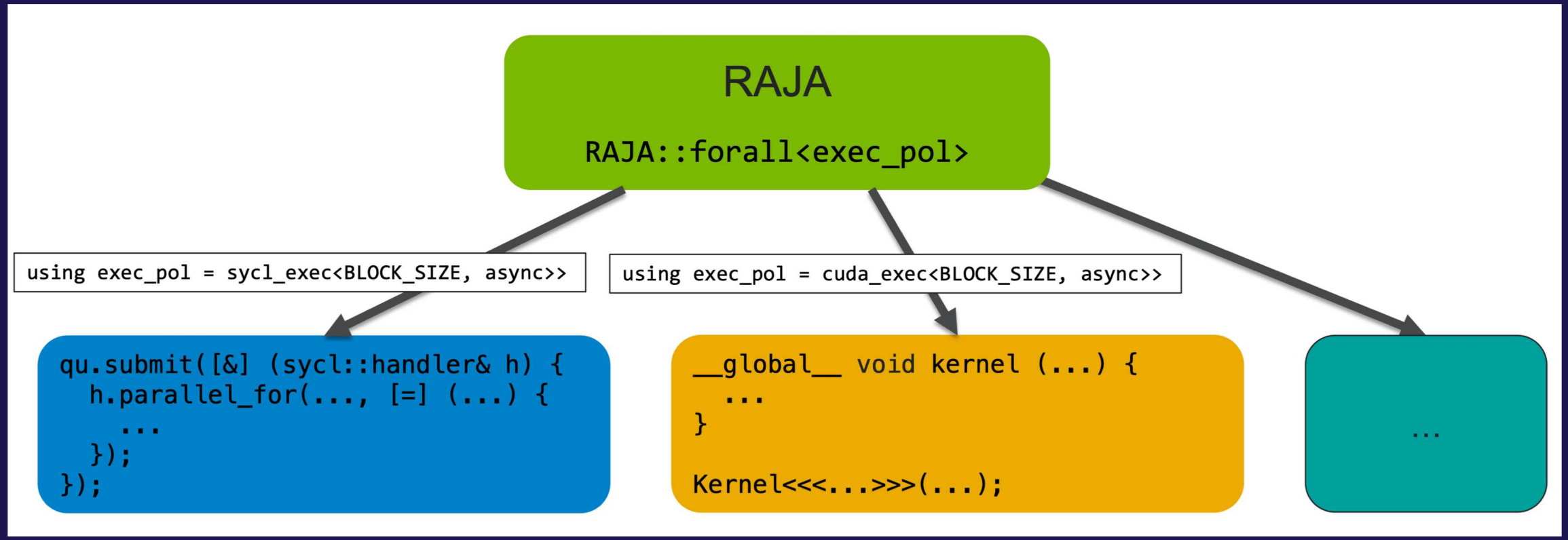
Argonne NATIONAL LABORATORY

Integrating SYCL/DPC++ into the RAJA Software Library

Brian Homerding
ALCF

RAJA

RAJA is a software library of C++ abstractions to support architecture and programming model portable parallel loop execution. It provides portable abstractions for simple and complex loops. Through the execution policy RAJA kernels are mapped to execution backends.



RAJA-SYCL Status

- RAJA – v2022.03.0
  - — Most recent release SYCL features include
    - RAJA::forall
    - RAJA::kernel
    - Reductions
    - Tiling
  - — Recent updates to the SYCL execution policies
    - Removed seperate non-trivial kernel launch interface
    - Tiling support
      - Supported in RAJA with mapping to groups and items
    - ForI and TileT policy support

Device Execution

```
sycl_exec<block_size, async>
SyclKernel<...>
```

Execution Policies

```
sycl_global_0<block_size>
sycl_global_1<block_size>
sycl_global_2<block_size>
sycl_group_0_direct
sycl_group_1_direct
sycl_group_2_direct
sycl_group_0_loop
sycl_group_1_loop
sycl_group_2_loop
sycl_local_0_direct
sycl_local_1_direct
sycl_local_2_direct
sycl_local_0_loop
sycl_local_1_loop
sycl_local_2_loop
```

Reduction Object

sycl_reduce

Porting RAJA kernels to new platform

omp_policies.h

```
typedef RAJA::omp_parallel_for_exec PREDFORT_LOOP_POL;
```

hip_policies.h

```
typedef RAJA::hip_exec<1024> PREDFORT_LOOP_POL;
```

cuda_policies.h

```
typedef RAJA::cuda_exec<1024> PREDFORT_LOOP_POL;

using RHS4_EXEC_POL_ASYNC =
    RAJA::KernelPolicy<RAJA::statement::CudaKernelFixedAsync<256,
      RAJA::statement::Tile<
        0, RAJA::statement::tile_fixed<4>, RAJA::cuda_block_z_loop,
          RAJA::statement::Tile<
            1, RAJA::statement::tile_fixed<4>, RAJA::cuda_block_y_loop,
              RAJA::statement::Tile<
                2, RAJA::statement::tile_fixed<16>, RAJA::cuda_block_x_loop,
                  RAJA::statement::For<
                    0, RAJA::cuda_thread_z_direct,
                      RAJA::statement::For<
                        1, RAJA::cuda_thread_y_direct,
                          RAJA::statement::For<
                            2, RAJA::cuda_thread_x_direct,
                              RAJA::statement::Lambda<0>>>>>>>>>;
    >;

using ODDIODDJ_EXEC_POL2_ASYNC = RHS4_EXEC_POL_ASYNC;
```

sycl_policies.h

```
typedef RAJA::sycl_exec<1024> PREDFORT_LOOP_POL;

using RHS4_EXEC_POL_ASYNC =
    RAJA::KernelPolicy<
      RAJA::statement::SyclKernel<
        RAJA::statement::For<0, RAJA::sycl_global_0,
          RAJA::statement::For<1, RAJA::sycl_global_1,
            RAJA::statement::For<2, RAJA::sycl_global_2
              RAJA::statement::Lambda<0>
            >
          >
        >
      >
    >;

using ODDIODDJ_EXEC_POL2_ASYNC = RHS4_EXEC_POL_ASYNC;

{...}
```

SYCL Kernel to RAJA-SYCL Kernel Mapping

```
qu->submit([&] (sycl::handler& h) {
  h.parallel_for(sycl::nd_range<3> (
                    sycl::range<3> (nk, nj, ni),
                    sycl::range<3> (1, 1, 256)),
                 [=] (sycl::nd_item<3> item) {


          Index_type i = item.get_global_id(2);
          Index_type j = item.get_global_id(1);
          Index_type k = item.get_global_id(0);


  });
});
```

```
using EXEC_POL =
  RAJA::KernelPolicy<
    RAJA::statement::SyclKernelAsync<
      RAJA::statement::For<2, RAJA::sycl_global_0<1>,
        RAJA::statement::For<1, RAJA::sycl_global_1<1>,
          RAJA::statement::For<0, RAJA::sycl_global_2<256>,
            RAJA::statement::Lambda<0>
          >
        >
      >
    >
  >;

  RAJA::kernel<EXEC_POL>( RAJA::make_tuple(RAJA::RangeSegment(0, ni),
                                           RAJA::RangeSegment(0, nj),
                                           RAJA::RangeSegment(0, nk)),
    [=] (Index_type i, Index_type j, Index_type k) {
```

0 1 2

Argonne NATIONAL LABORATORY

Early Experience

Useful Features - Flexibility

There are many features which provide flexibility which are useful for libraries

- `sycl::nd_range`
  - —RAJA by design exposes control over how to execute a kernel to the application. Utilizing nd_ranges allows us to expose fine grained control to the kernel execution policies
- unnamed lambda
  - —Libraries which launch kernels user defined kernels cannot provide useful and unique kernel names

Useful Features - Compatibility

There are several features which help the SYCL backend provide similiar functionality to the existing backends

- extended atomics
  - —Providing consistent support for atomics across backends requires support for additional types and memory scopes
- unified shared memory
  - —RAJA is all about kernel execution. In order to seperate the memory management from the kernel execution, USM is needed

Challenges Related to Integrating SYCL

- Providing a consistent context across the application, RAJA and the memory management software
- Getting access to local and group id in components which have a complicated call path with existing interfaces.
- Allocating shared local memory outside of the kernel launch

Argonne NATIONAL LABORATORY

Thank you

Argonne NATIONAL LABORATORY

ECP BoF: Early Experience of Application Developers with SYCL/Data Parallel C++ MFIX

Brian Holland, Intel Center of Excellence

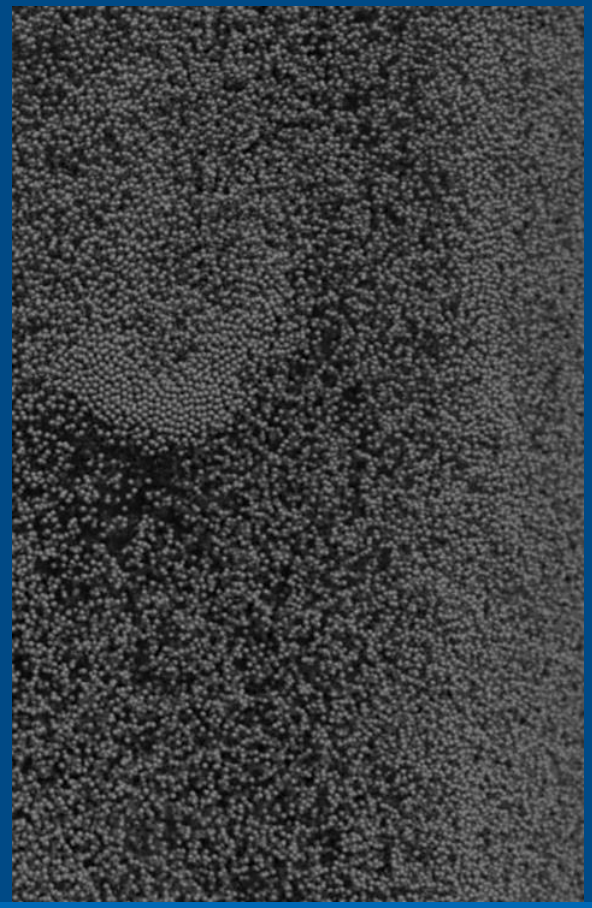
May 10, 2022

intel

Background

Computational Fluid Dynamics-Discrete Element Methods (CFD-DEM)

- Fluid Is a continuum solved on a grid, whereas particles are individually tracked (one-to-one)
- Fluid and particles are coupled through volume fraction and interphase interactions (e.g., drag force)
- Advantages
  - Use first principles to account for particle-particle and particle-boundary interactions
  - Fewer complex closures therefore less overall model uncertainty
- Limitations
  - Computationally expensive
  - Fluid-particular interactions are modeled



MFIX

- Written in C++
- Built on AMReX software framework
  - using AMReX data structures and performance portability constructs
- Fully ported to GPUs using amrex::Parallelfor
  - Normal for loop when running on CPUS
  - GPU Kernel launch with DPC++
- Users AMReX's Embedded Boundaries (EB) for specifying complex geometries
- AMReX native Multi-level Multi-Grid (MLMG) for linear solvers
- Supports in situ (i.e. real-time) visualization

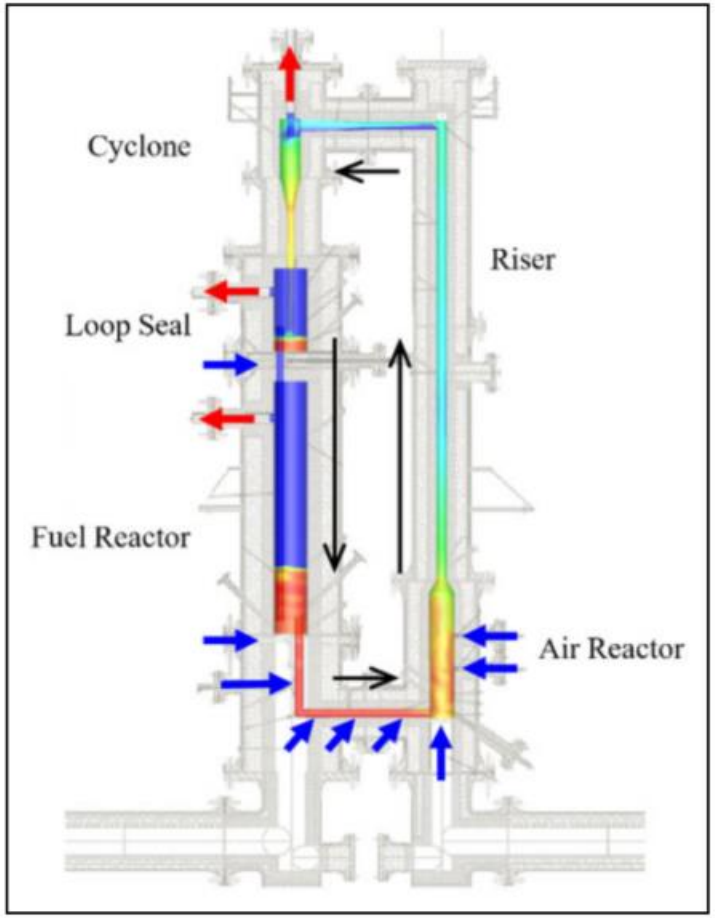


Figure 1. Illustration of the challenge problem: NETL's 50 kW chemical looping reactor. Blue arrows indicate gas inflows, and red arrows indicate gas outflows. Black arrows show the counter-clockwise solids circulation path. Image adapted from Bayham et al. (2016).

AMReX - DPC++ for Intel GPUs

https://github.com/AMReX-Codes/amrex/blob/development/Src/Base/AMReX_MFParallelForG.H

```
#include <AMReX_PlotFileUtil.H>
#include <AMReX_ParmParse.H>
using namespace amrex;
int main (int argc, char* argv[])
{
    amrex::Initialize(argc,argv);

    main_main();

    amrex::Finalize();
    return 0;
}
```

```
for (int step = 1; step <= nsteps; ++step) {
   phi_old.FillBoundary(geom.periodicity());

   // loop over boxes
    for ( MFIter mfi(phi_old); mfi.isValid(); ++mfi ){
      const Box& bx = mfi.validbox();
      const Array4<Real>& phiOld = phi_old.array(mfi);
      const Array4<Real>& phiNew = phi_new.array(mfi);

      // advance the data by dt
      amrex::ParallelFor(bx, [=] AMREX_GPU_DEVICE (int i, int j, int k) {
        phiNew(i,j,k) = phiOld(i,j,k) + dt *
          ( (phiOld(i+1,j,k) - 2.*phiOld(i,j,k) + phiOld(i-1,j,k)) / (dx[0]*dx[0])
           +(phiOld(i,j+1,k) - 2.*phiOld(i,j,k) + phiOld(i,j-1,k)) / (dx[1]*dx[1])
#if (AMREX_SPACEDIM == 3)
           +(phiOld(i,j,k+1) - 2.*phiOld(i,j,k) + phiOld(i,j,k-1)) / (dx[2]*dx[2])
#endif
          );
      });
    }

    time = time + dt;
    MultiFab::Copy(phi_old, phi_new, 0, 0, 1, 0);
    amrex::Print() << "Advanced step " << step << "\n";
    if (plot_int > 0 && step%plot_int == 0){
      const std::string& pltfile = amrex::Concatenate("plt",step,5);
      WriteSingleLevelPlotfile(pltfile, phi_new, {"phi"}, geom, time, step);
    }
  }
```

```
#elif defined(AMREX_USE_DPCPP)

   amrex::launch(nblocks, MT, Gpu::gpuStream(),
       [=] AMREX_GPU_DEVICE (sycl::nd_item<1> const& item) noexcept
       {
          int ibox, icell;
          int blockIdxx = item.get_group_linear_id();
          int threadIdxx = item.get_local_linear_id();
          if (dp_nblocks) {
             ibox = amrex::bisect(dp_nblocks, 0, nboxes,
static_cast<int>(blockIdxx));
             icell = (blockIdxx-dp_nblocks[ibox])*MT + threadIdxx;
          } else {
             ibox = blockIdxx / block_0_size;
             icell = (blockIdxx-ibox*block_0_size)*MT + threadIdxx;
          }
#endif
          Box const& b = dp_boxes[ibox];
          int ncells = b.numPts();
          if (icell < ncells) {
             const auto len = amrex::length(b);
             int k =  icell /   (len.x*len.y);
             int j = (icell - k*(len.x*len.y)) /   len.x;
             int i = (icell - k*(len.x*len.y)) - j*len.x;
             AMREX_D_TERM(i += b.smallEnd(0);,
                          j += b.smallEnd(1);,
                          k += b.smallEnd(2);)
             for (int n = 0; n < ncomp; ++n) {
                parfor_mf_detail::call_f(f, ibox, i, j, k, n);
             }
          }
       });
   }
   AMREX_GPU_ERROR_CHECK();
}
```

From amrex-tutorials:

https://github.com/AMReX-Codes/amrex-tutorials/blob/main/ExampleCodes/Basic/HeatEquation_EX0_C/Source/main.cpp

MFIX - Algorithm Structure within AMReX

MFIX

Eulerian Gas Phase (Fluid)

$$\frac{\partial \varepsilon_g}{\partial t} + \nabla \cdot (\varepsilon_g \boldsymbol{U}_g) = 0 ,$$

$$\frac{\partial}{\partial t}\left(\varepsilon_g \rho_g \boldsymbol{U}_g\right) + \nabla \cdot \left(\varepsilon_g \rho_g \boldsymbol{U}_g \otimes \boldsymbol{U}_g\right) = -\varepsilon_g \nabla p_g + \nabla \cdot \boldsymbol{\tau}_g + \varepsilon_g \rho_g \boldsymbol{g} - \beta(\boldsymbol{U}_g - \boldsymbol{U}_p) ,$$

fluid mass and momentum conservation

Phasic Coupling

Transferring Lagrangian (Particle) Data on the Eulerian Grid

Lagrangian Particle Phase

$$\frac{d\boldsymbol{X}_p}{dt} = \boldsymbol{V}_p ,$$

$$m_p \frac{d\boldsymbol{V}_p}{dt} = m_p \boldsymbol{g} + \boldsymbol{F}_{gp} + \boldsymbol{F}_{wp} + \sum_{q=1}^{N_p} \boldsymbol{F}_{qp} ,$$

$$I_p \frac{d\boldsymbol{\omega}_p}{dt} = \boldsymbol{T}_{wp} + \sum_{q=1}^{N_p} \boldsymbol{T}_{qp} ,$$

discrete position, linear and angular velocities

AMReX

Linear Solvers

`MaxProjector` — access linear solves and MLMG solvers for Fluid Convective Update

`NodalProjector` — access linear solves and MLMG solvers for fluid - node pressures and cell velocities

Mesh Data

`MultiFab`

`BoxArray`

array of boxes/grids at single level of refinement

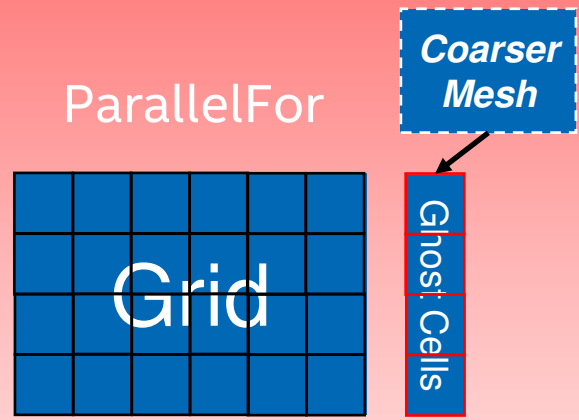
single grid

`FBoxArray`

⋮

`FBoxArray`

ParallelFor



Efficient stencil iteration with AMReX optimized communication to fill ghost cells

Thank You

Interoperability of SYCL & OpenMP

Thomas Applencourt
Argonne National Laboratory

EXASCALE COMPUTING PROJECT

OpenMP ↔ SYCL: An HPC Story

- Your HPC applications is written in C++ / OpenMP
- But you may want to be interfaced with SYCL:
  - Some SYCL API are more flexible than the OpenMP counterpart
    - OneMKL provide both an OpenMP and SYCL API,
      but SYCL API give access to more function (batched DGEMM for example)
  - Some API only exist in SYCL
    - For example, oneDPL (Intel oneAPI thrust)
  - You want to use SYCL to allocate memory (host, device, shared)

- Interoperability to the rescue!

```
#pragma omp target enter data map(to: data[0:N])
T* data_gpu;
#pragma omp target data use_device_ptr(data)  { data_gpu = data }
sycl::queue q = get_interopt_queue(); //Magic Function, more about it later
//SYCL parallel stl using an OpenMP device pointer
std::sort(oneapi::dpl::execution::make_device_policy(q), data_gpu, data_gpu + N);
```

OpenMP ➡ Backend ➡ SYCL

- Use `#pragma omp interop` to get Native Handler (OpenMP 5.1)
- Use those handlers to create SYCL Object (SYCL 2020)
- POC: Implementation using L0 API and ICPX:
  - https://github.com/argonne-lcf/HPC-Patterns/blob/main/sycl_omp_ze_interopt/interop_omp_ze_sycl.cpp

- Code example:

```
omp_interop_t o;
#pragma omp interop init(targetsync: o)
auto hDevice = static_cast<ze_device_handle_t>(
                  omp_get_interop_ptr(o, omp_ipr_device, &err));
#pragma omp interop destroy(o)

const sycl::device sycl_device =
          sycl::make_device<sycl::backend::ext_oneapi_level_zero>(hDevice);
```

OpenMP ⬌ SYCL Work

- Pro: It work! (something similar used by QMCPACK)

```
sycl::queue Q = get_intereopt_queue(); // Where the magic happens
      T *ompMem = (T*) malloc(N*sizeof(T));
T *syclMem = sycl::malloc_device<T>(N,Q);
```

  - OpenMP Target using “SYCL memory”

```
#pragma omp target is_device_ptr(syclMem) map(from:ompMem[0:N])
for (size_t i=0 ; i < N; i++)
            ompMem[i] = syclMem[i];
```

  - SYCL using “OpenMP memory”

```
T* ompMem_gpu;
#pragma omp target enter data map(to:ompMem[0:N])
#pragma omp target data use_device_ptr(ompMem) { ompMem_gpu = ompMem }
Q.copy<T>(cpuMem, ompMem_gpu, N).wait();
```

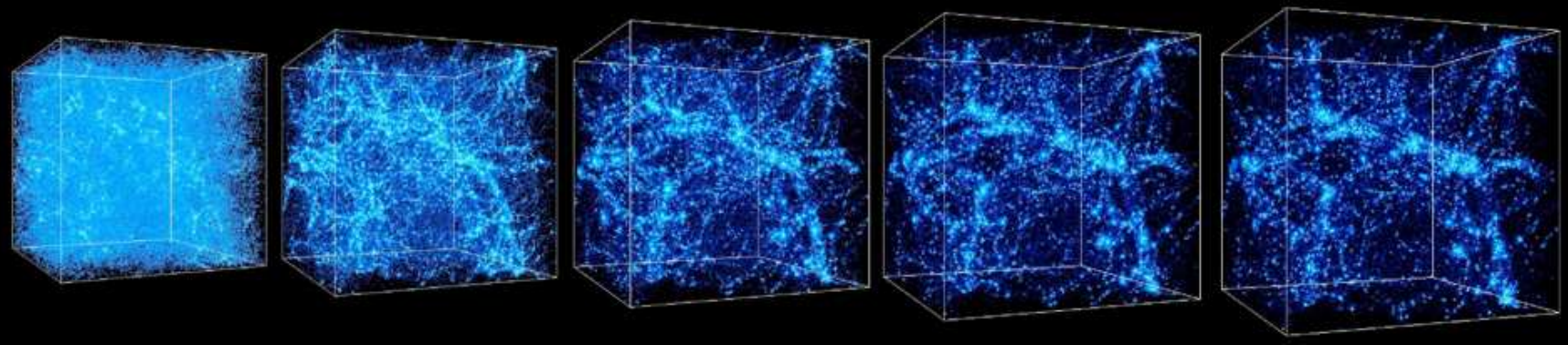
- Con:
  - Backend specific (need to cast pointer | specialize the templated API)
  - Only tested on ICPX & need to use non-standard interoperability API to workaround some bugs

Collaboration are welcome to tests/implements support for more compiler / backend!

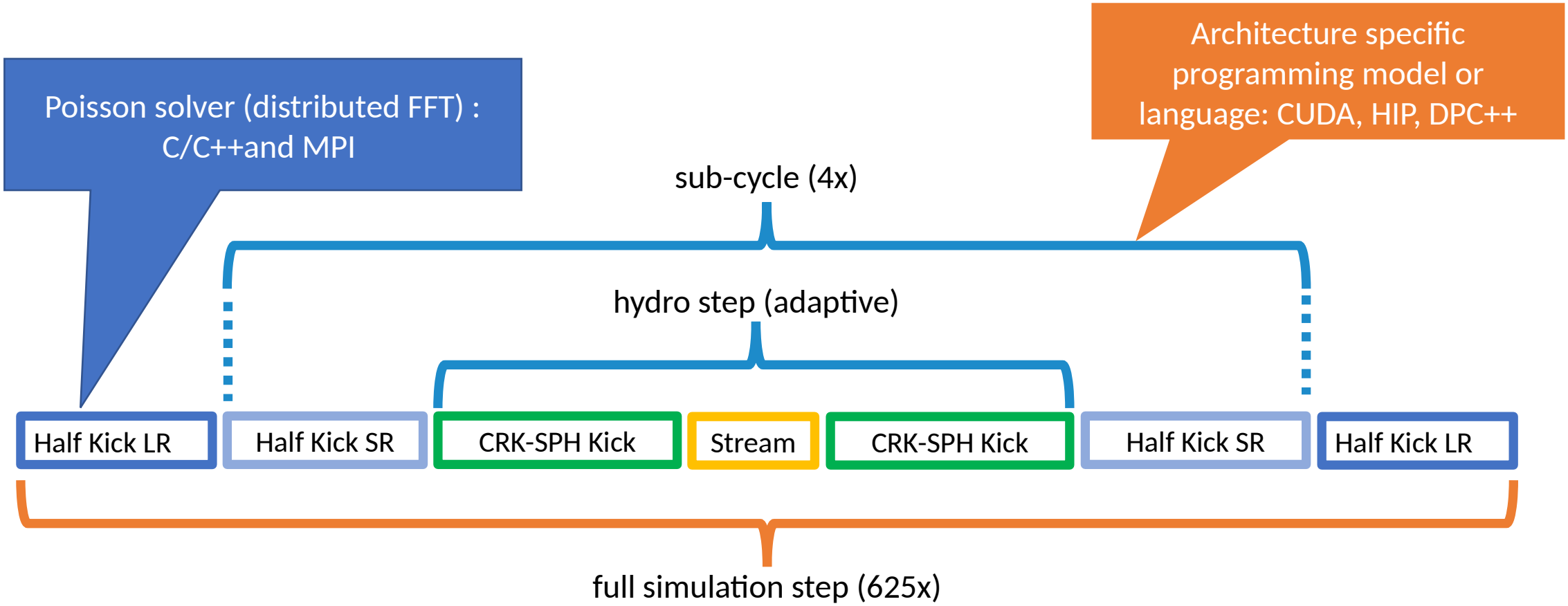
Supporting CUDA, HIP, and SYCL in HACC

Steve Rangel, CPS Division ANL

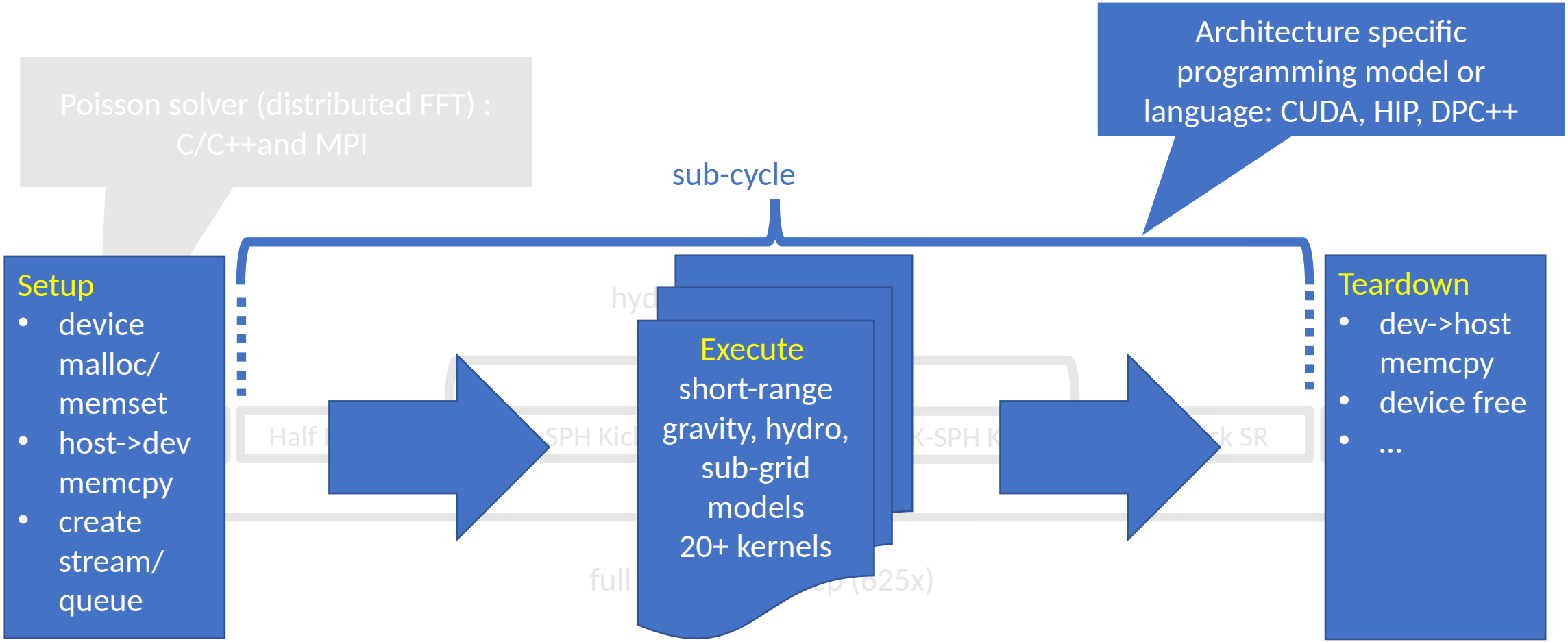
ECP BoF: Early Experience of Application Developers with SYCL/Data Parallel C++



HACC’s Codebase (overview)



HACC's Codebase (overview)



Supporting Multiple APIs

- HACC GPU kernels are implemented in the vendor preferred programming model (namely CUDA, HIP, SYCL/DPC++), with the goal to achieve maximum performance over portability.
- Primary development using CUDA.
- Porting to HIP requires minimal code changes.
- Porting to SYCL is semi-automated, done with the help of the Intel DPC++ Compatibility Tool (DPCT) for device kernels, file by file, and manually for host-side code.
- Calls to specific APIs are “wrapped” so **build targets can reuse the same host-side source files**.
  - Most functionality has a direct one-to-one mapping: https://developer.codeplay.com/products/computecpp/ce/1.1.4/guides/sycl-for-cuda-developers/migration

Wrapping Kernel Invocation

InvokeGPUKernel (CUDA)

```
#define InvokeGPUKernel(NAME, nBlock, BlockSize, SMEM, stream, ...) { \
    int64_t n64 = nBlock; int64_t B64 = BlockSize; \
    int64_t nThread = n64*B64;\
    assert((nThread < (int64_t(1) << 31)) && (nThread >= 0)); /* assert tha
    if(nThread == 0)return;/* no op if number of threads is zero. */ \
    NAME<<<nBlock, BlockSize, SMEM, stream>>>(__VA_ARGS__);\
    cudaStreamSynchronize(stream); /* Synchronize Stream */ \
    cudaCheckError(); /* check for errors */ \
}
```

- Variadic arguments are used to generalize the invocation wrapper.

InvokeGPUKernel (SYCL)

```
#define InvokeGPUKernel(NAME, nBlock, BlockSize, _, sycl_queue, ...) {
  InvokeSYCLKernel<NAME>(nBlock, BlockSize, sycl_queue, __VA_ARGS__);
}

template<class K, typename... T>
void InvokeSYCLKernel( int nBlock, int BlockSize, sycl::queue sycl_queue,
  T ...args)
{
  K sycl_kernel = K(args...);
  sycl::event e =
    sycl_queue.parallel_for(sycl::nd_range<1>(nBlock*BlockSize, BlockSize),
      sycl_kernel);
  e.wait();
}
```

- Note the above requires the functor implementation of a SYCL kernel, which is currently not supported by DPCT. We wrote an additional Clang-based tool to automate the transformation.

Wrapping Common API Calls

- In general, we take the union of function arguments used by the supported APIs to provide a common interface to the host.
- Notice how the stream argument in the CUDA wrapper for GPUMalloc is not used, but is necessary for SYCL.

```
#define InvokeGPUMalloc(devPtr, size, stream){ \
    cudaMalloc(devPtr,size); /* no stream needed */\
    cudaCheckError(); /* check for errors */ \
}
```

```
#define InvokeGPUMalloc(devPtr, size, queue){          \
  *devPtr = (char*)sycl::malloc_device(size, queue); \
  queue.wait();                                        \
}
```

Summery

- HACC is supporting CUDA, HIP, and SYCL for the upcoming exascale systems.
- The SYCL implementation uses the same explicit memory management model as the CUDA implementation.
- SYCL requires more effort for porting but is significantly helped by using the Intel DPC++ Compatibility Tool.
- The specific API calls are wrapped to promote code reuse and ease maintainability.

Support for Intel GPUs with SYCL in *hypre*

CASC
Center for Applied Scientific Computing

May 11th, 2022

ECP Annual Meeting BoF Sessions

Wayne Mitchell

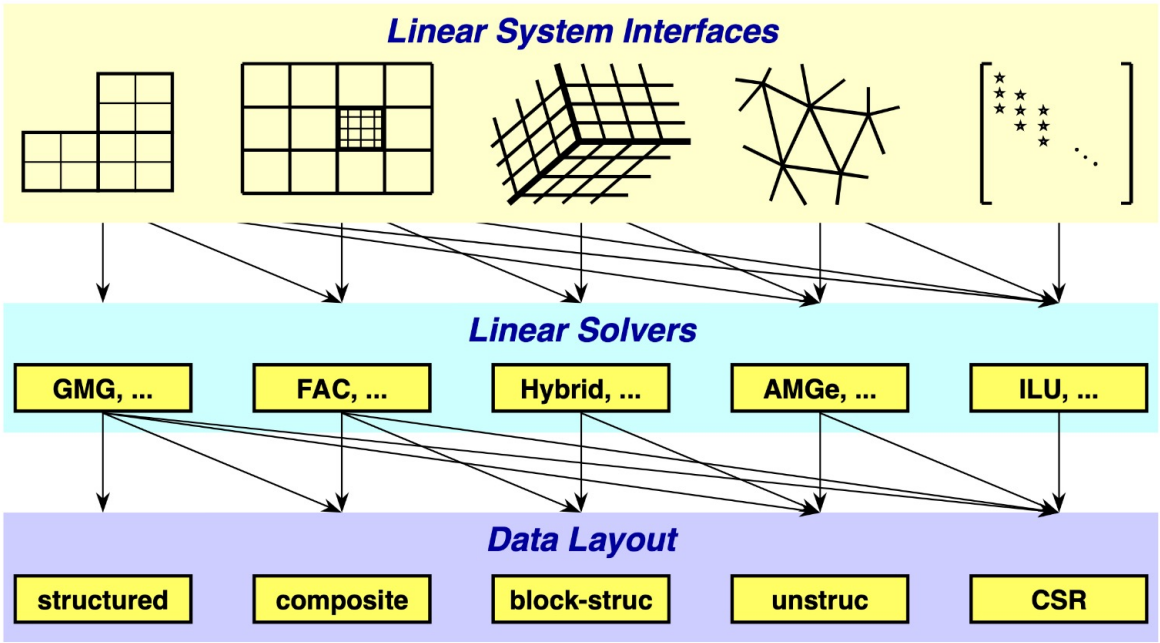
LLNL-PRES-834846

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

Introduction to *hypre*

- *hypre* is a library of scalable linear solvers produced by Lawrence Livermore National Lab (LLNL)
  - Geometric multigrid preconditioners for structured problems
  - BoomerAMG (algebraic multigrid) for unstructured problems



GPU support

- Most of *hypre* is written in plain C and was originally developed for large parallel CPU machines
- GPU support originally focused on CUDA
- Expanding support for HIP and SYCL
- *hypre* is designed to have minimal dependencies
  - Options for use of Kokkos or RAJA (with limited use cases) but we do not rely on these or other portability frameworks
  - Native use of CUDA/HIP and now SYCL

Current SYCL support in *hypre*

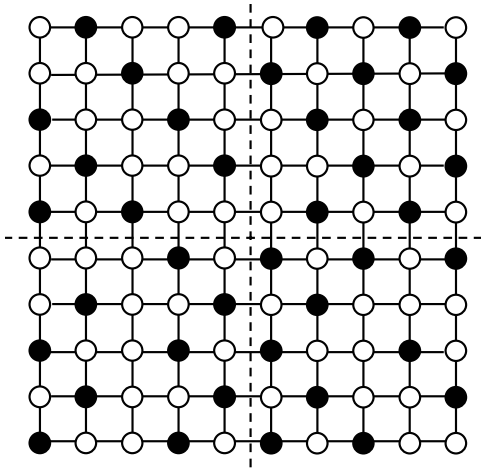
- Target Intel GPUs on upcoming Aurora system at Argonne
  - Use Intel's Data Parallel C++ (DPC++) implementation of SYCL
  - Currently testing on early access systems for Aurora
- *hypre* functionality running on Intel GPUs:
  - Structured interface solvers (full support)
  - Basic CSR infrastructure and linear algebra
  - BoomerAMG solve phase (with Jacobi relaxation)
  - In progress: BoomerAMG setup phase
    - Matrix-matrix multiplication
    - Interpolation routines
    - Coarsening routines

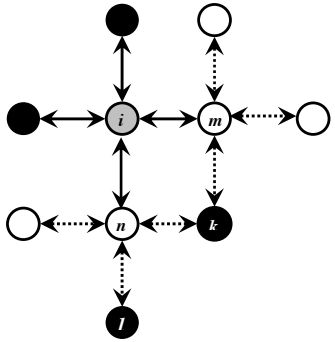
Usage of oneMKL in *hypre*

- The unstructured interface is based on a parallel CSR format for storing matrices
- Basic sparse linear algebra may be supplied by vendor libraries: cuSPARSE, rocSPARSE, oneMKL
- Solve phase for BoomerAMG is mostly matrix-vector products
  - No custom matrix-vector product
  - Rely on oneMKL sparse `gemv()`
- Setup phase for BoomerAMG utilizes matrix-matrix product
  - Custom matrix-matrix product is faster than cuSPARSE
  - Currently use oneMKL sparse `matmat()`

Porting BoomerAMG setup

- The setup phase of BoomerAMG involves
  - Generating a strength of connection matrix
  - Choosing a coarse grid (partition dofs into C/F points)
  - Build interpolation and restriction operators
  - Form a coarse-grid operator $A_c = RAP$





- Requires a lot of specialized kernels and use of Thrust algorithms!

Translating kernels

- Use macros to unify kernel launch syntax between CUDA/HIP/SYCL

```
#define HYPRE_GPU_LAUNCH(kernel_name, gridsize, blocksize, ...)
```

- SYCL kernels need an additional `sycl::nd_item` argument

```
__global__ void
hypre_ExampleKernel(
#if defined(HYPRE_USING_SYCL)
   sycl::nd_item<1> item,
#endif
   …
```

- Obtaining thread/subgroup IDs and subgroup collective operations (shuffle, etc.) also require translation

Translating Thrust algorithms

- Many Thrust algorithms have `std` or `oneapi` equivalents
- Notable exceptions:
  - `gather()` and `scatter()`
  - Conditional operations with an extra arguments for the stencil

```
thrust::copy_if(first, last, stencil, result, pred);
```

- These can be easily implemented using permutation/zip iterators
- Helpful to write wrapper functions that translate missing Thrust function calls

Conclusions and outlook

- *hypre* support for Intel GPUs with SYCL is well underway
  - Structured interface is completely supported
  - BoomerAMG solve phase is supported
  - BoomerAMG setup phase is coming soon
  - Additional specialized solvers will be continually added
- Additional focus going forward on achieving good performance on early access hardware for Aurora
- Special thanks to the Intel COE team!

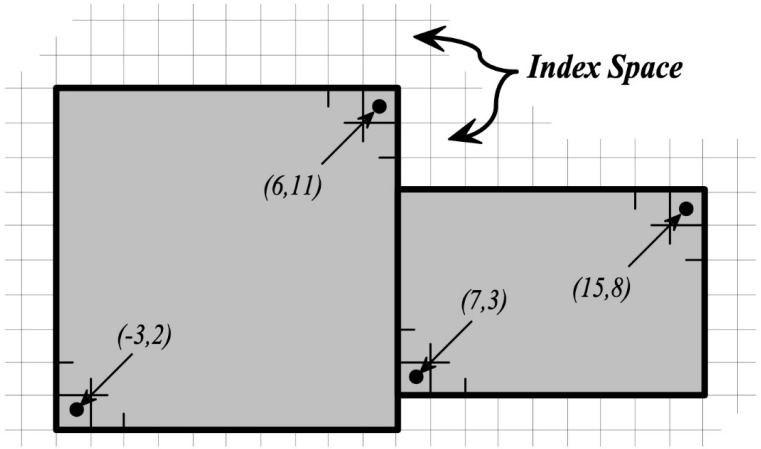
CASC

Center for Applied
Scientific Computing

Lawrence Livermore
National Laboratory

Structured solvers in SYCL

- Structured interface is based on boxes and stencils
- Computational kernels are defined as “BoxLoops”
- Porting to different backends involves rewriting the BoxLoops
- Loops that involve a reduction require special treatment



Early Experiences with SYCL in TestSNAP

Yasaman Ghadar
Argonne Leadership Computing Facility

We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.

ECP Community BoF, May 2022

Overview of EXAALT Project

- ECP EXAALT project seeks to extend accuracy, length and time scales of material science simulations for fission/fusion reactors using **LAMMPS**
  - Task management layer to create MD tasks, manage task queues, and store results in databases
  - Long-time, high-accuracy MD simulations with DFTB method
  - Long-time, large-scale MD simulations with machine learned **SNAP** (Spectral Neighbor Analysis Potential) potential

- Programming models:
  - ParSplice: C++
  - **LAMMPS: C/C++, OpenMP, GPU-enabled (Kokkos, CUDA, OpenCL, ROCm)**
  - LATTE: F90, OpenMP, GPU-enabled (CUDA)

- Performance directly depends on single-node performance for SNAP
  - Energy is a function of geometric descriptor of atomic environments

TestSNAP

Pseudo-code for TestSNAP

```
for ( int natom = 0; natom < num_atoms ; ++ natom )
{
// build neighbor - list for all atoms
 build_neighborlist ();

 // compute atom specific coefficients
 compute_U (); // Ulist [ idx_max ] and Ulisttot [ idx_max ]
 compute_Y (); // Ylist [ idx_max ]

 // for each (atom , neighbor ) pair
for ( int nbor = 0; nbor < num_nbor ; ++ nbor )
{
 compute_dU (); // dUlist [ idx_max ][3]
 compute_dE (); // dElist [3]
update_forces ()
 }
```

- TestSNAP is a stand-alone proxy app for the SNAP potential that can be run independently of LAMMPS and is written in C++ using OpenMP Target, CUDA, Kokkos and other programming models.

- Each of the compute routines iterate over the bi-spectrum components (id_max) and store their individual contributions in a 1D array.
  - For 2J14, *idx_max* is 15
  - For 2J8, *idx_max* is 9
  - For 2J2, *idx_max* is 3

TestSNAP Main Kernels

Start

loops over all the atoms in the simulation to compute the forces per time step.

Compute_U

The compute_U routine calculates the expansion coefficients for each (atom, neighbor) pair and stores the information in Ulist.

The expansion coefficients for each atom are summed over all its neighbors and stored in Ulisttot.

Compute_Y

The Clebsch-Gordan products for each atom are calculated in the routine compute_Y and stored in Ylist.

Compute_du

As a precursor to force calculations, derivatives of expansion coefficients which are stored in Ulist are computed by the compute_dU in all 3 dimensions using spherical coordinates and stored in dUlist.

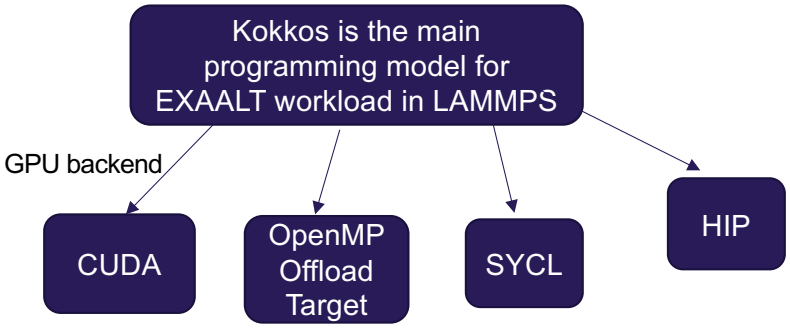
Compute_dE

Using dUlist and Ylist, the force vector for each (atom, neighbor) pair is computed by compute_dE and stored in dElist

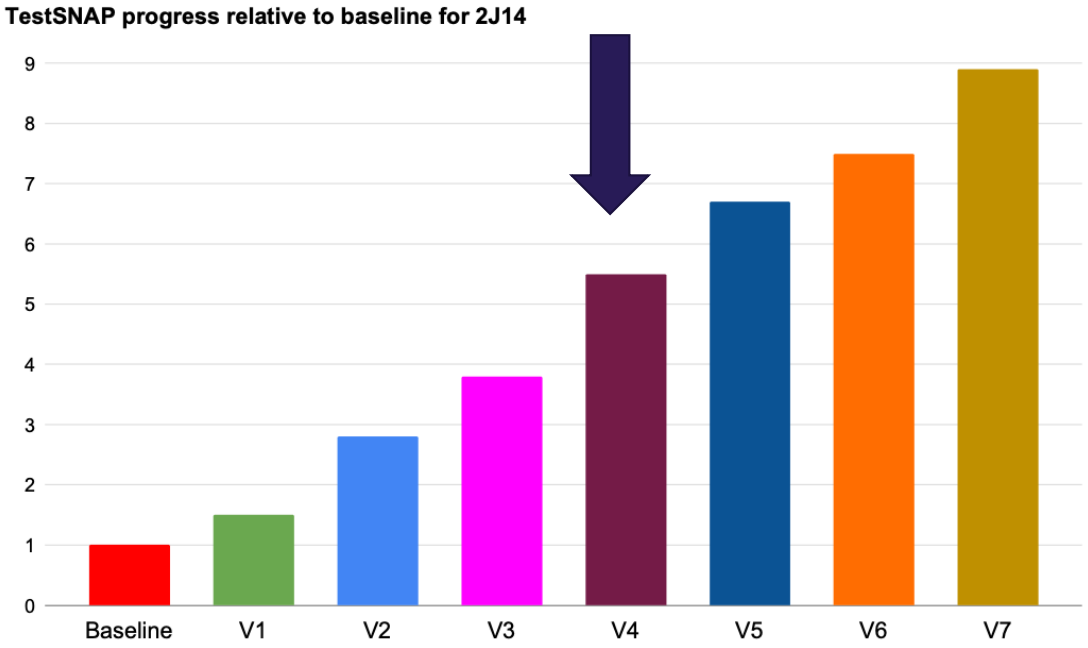
Final

Finally, the force on each atom is computed from dElist in update forces. A correctness check is built-in which compares the output against reference data

Kokkos Implementation of TestSNAP



Using Kokkos view and optimizing for shared memory, a new kernel which was fused of multiple kernels was created Compute Fused_Dei_Drj



Rapid Exploration of Optimization Strategies on Advanced Architectures using TestSNAP and LAMMPS
R. Gayatri, S. Moore, E. Weinberg, N. Lubbers, S. Anderson, J. Deslippe, D. Perez, and A. P. Thompson, Computer Science, Distributed, Parallel and Cluster Computing, 2020.

TestSNAP Kernels with Kokkos

GPU	Compute FusedDeiDrj	Compute Ui	Compute Yi
Intel Gen9	9.31	21.6	168
NVIDIA A100	0.097	0.0518	0.938
AMD MI100	0.298	0.409	2.08

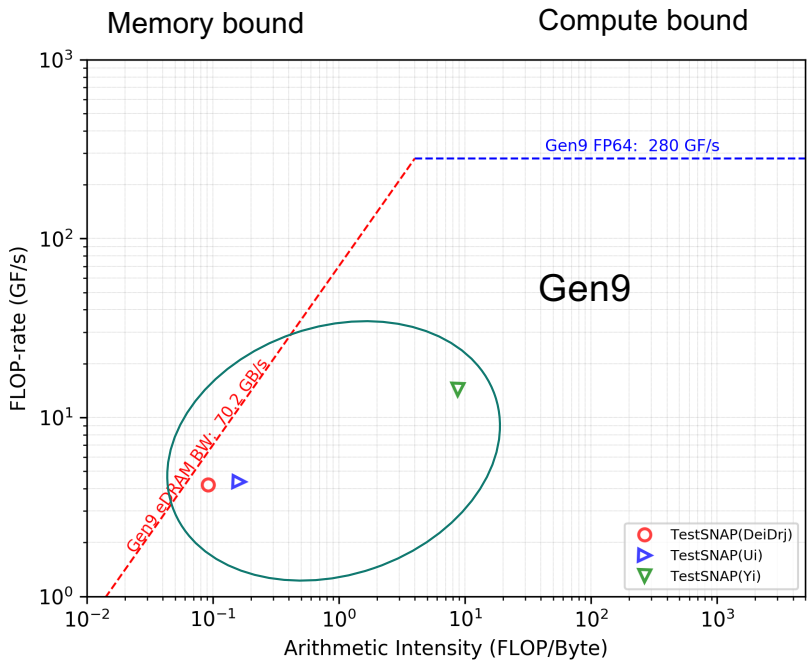
Intel: Kokkos/SYCL
NVIDIA: Kokkos/CUDA
AMD: Kokkos/HIP

Compute_Yi still is the most expensive kernel across arch.

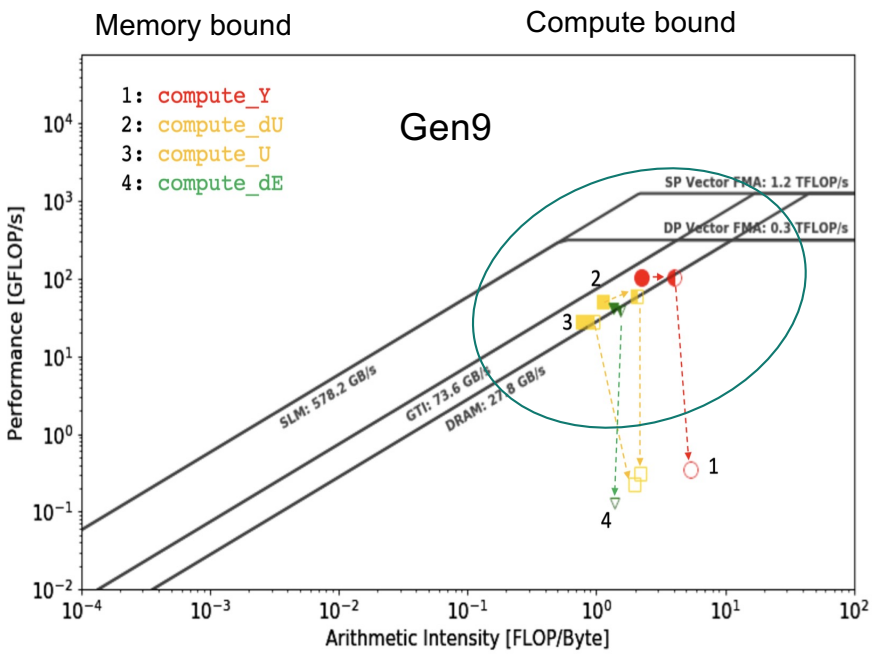
GPU	Kernel	AI_k	FR_k (GF/s)	Bound	P_k (GF/s)
Intel Gen9	FusedDeiDrj	0.0918	4.20	Memory	6.45
	Ui	0.161	4.37	Memory	11.3
	Yi	8.70	14.3	Compute	280.
NVIDIA A100	FusedDeiDrj	126.	3340	Compute	9390
	Ui	41.8	2030	Compute	9390
	Yi	319.	2540	Compute	9390
AMD MI100 (estimated)	FusedDeiDrj	123.	1090	Compute	10900
	Ui	24.2	257.	Compute	10900
	Yi	285.	1140	Compute	10900

Evaluation of Performance Portability of Applications and Mini-Apps across AMD, Intel and NVIDIA GPUs
*J. Kwack, J. Tramm, C. Bertoni, **Y. Ghadar,** B. Homerding, E. Rangel, C. Knight, and S. Parker, P3HPC Workshop @SC21*

Roofline Analysis of TestSNAP at Intel Gen9

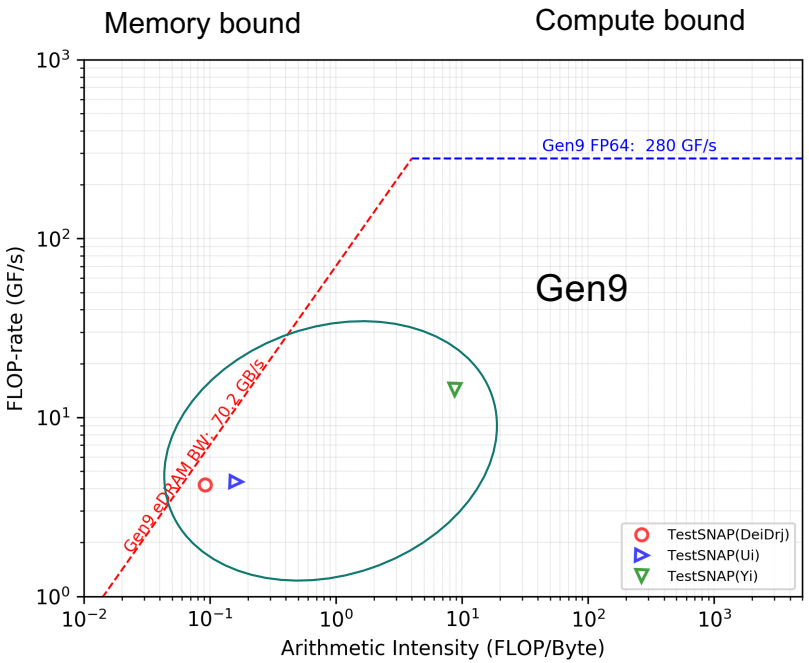


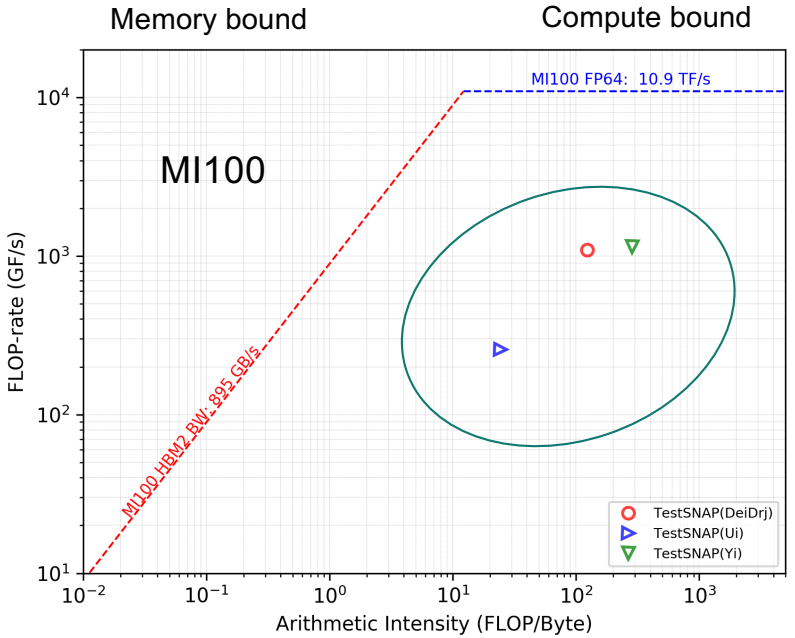
Kokkos/SYCL



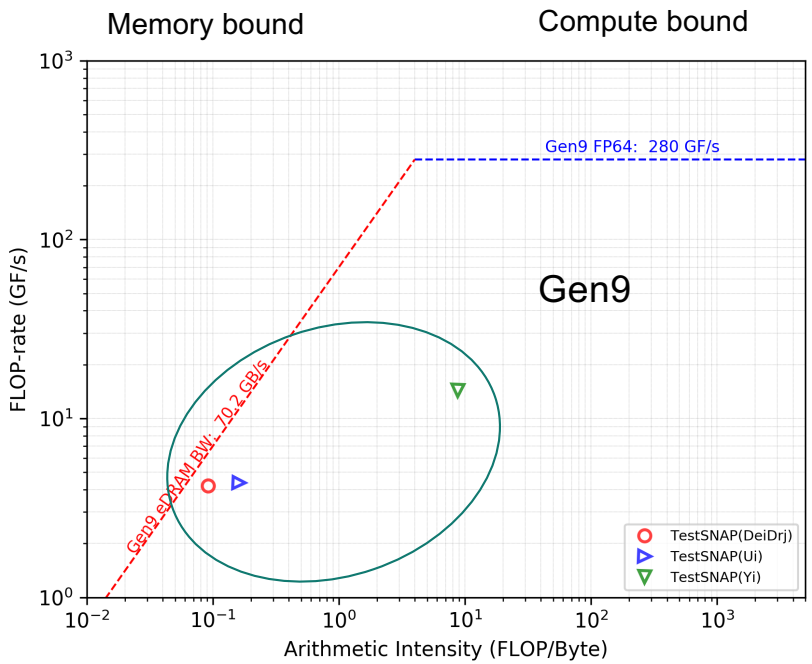
Pure OpenMP

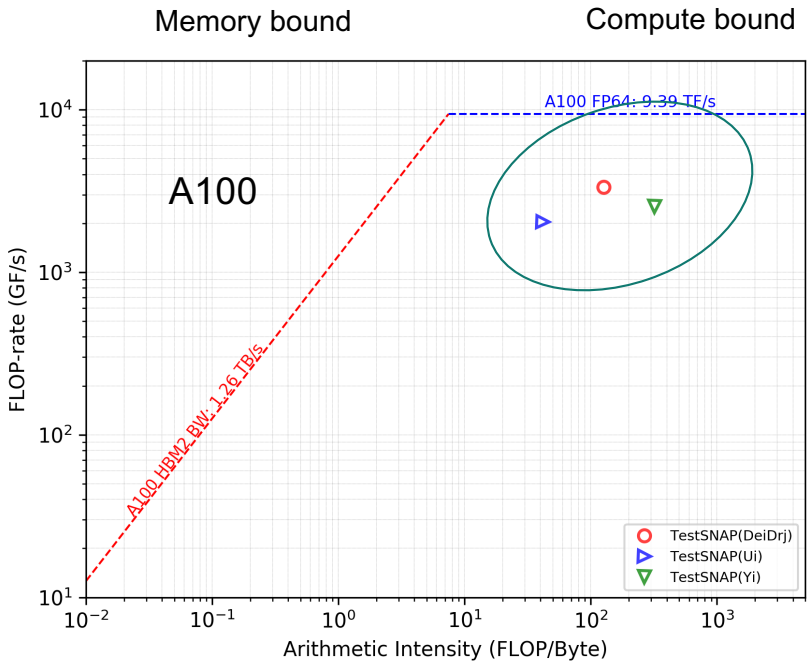
Roofline Analysis of TestSNAP with Kokkos/SYCL





Roofline Analysis of TestSNAP with Kokkos/SYCL





Using light weight profiling tools such as iprof indicated that the kernels are taking different path based on backend selected

TestSNAP has different code paths for each backend

Code path

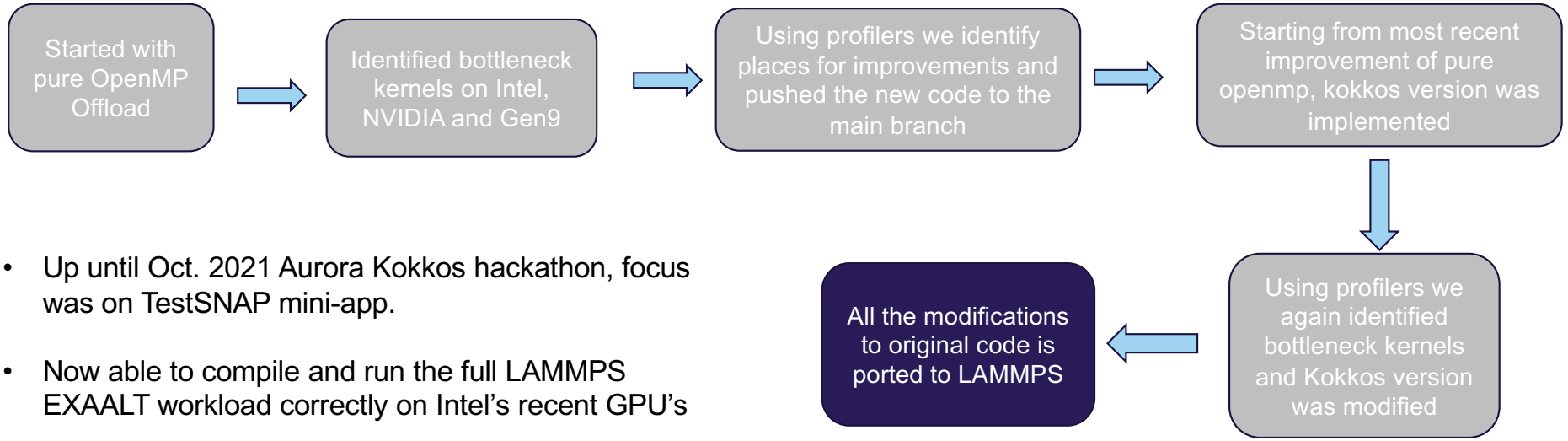
```
//*
#if defined(KOKKOS_ENABLE_CUDA) || defined(KOKKOS_ENABLE_HIP) || \
    defined(KOKKOS_ENABLE_SYCL)
    alist_gpu =
        SNAcomplex_View3DL("alist_gpu", vector_length, num_nbor, num_atoms_div);
    blist_gpu =
        SNAcomplex_View3DL("blist_gpu", vector_length, num_nbor, num_atoms_div);
    dalist_gpu = SNAcomplex_View4DL("dalist_gpu", vector_length, num_nbor,
                                    num_atoms_div, 3);
    dblist_gpu = SNAcomplex_View4DL("dblist_gpu", vector_length, num_nbor,
                                    num_atoms_div, 3);
    sfaclist_gpu = double_View4DL("sfaclist_gpu", vector_length, num_nbor,
                                  num_atoms_div, 4);

    ulisttot_re_gpu = double_View3DL("ulisttot_re_gpu", vector_length,
                                     idxu_half_max, num_atoms_div);
    ulisttot_im_gpu = double_View3DL("ulisttot_im_gpu", vector_length,
                                     idxu_half_max, num_atoms_div);
    ulisttot_gpu = SNAcomplex_View3DL("ulisttot_gpu", vector_length, idxu_max,
                                      num_atoms_div);

    ylist_re_gpu = double_View3DL("ylist_re_gpu", vector_length, idxu_half_max,
                                  num_atoms_div);
    ylist_im_gpu = double_View3DL("ylist_im_gpu", vector_length, idxu_half_max,
                                  num_atoms_div);

#else
```

Early Experience of Porting EXAALT (2022)



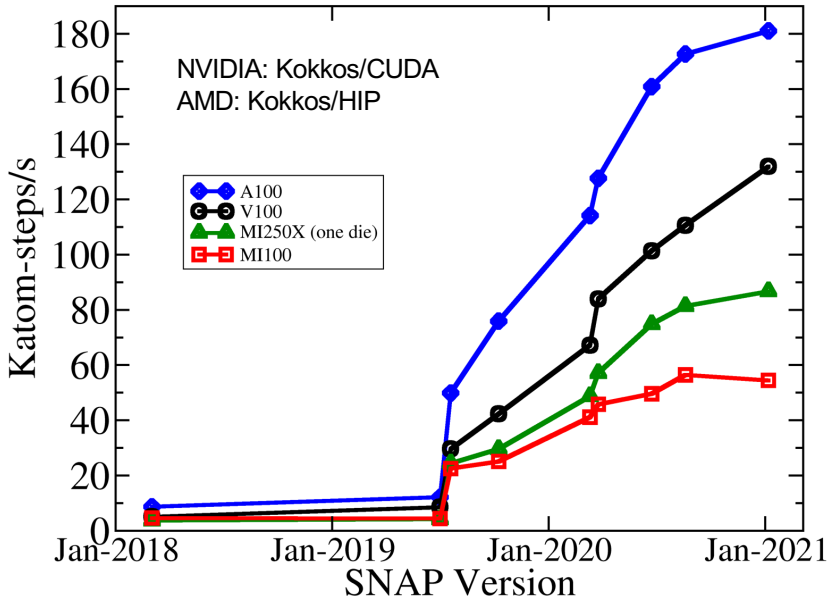
- Up until Oct. 2021 Aurora Kokkos hackathon, focus was on TestSNAP mini-app.
- Now able to compile and run the full LAMMPS EXAALT workload correctly on Intel's recent GPU's

This was done, through continuous open communication with EXAALT developers, Kokkos developers, Intel COE and so many more.

- EXAALT Team: Danny Perez, Rahul Gayatri, Stan Moore
- Intel CEO: Patrick Steinbrecher
- Kokkos Developer: Daniel Arndt
- Argonne: Chris Knight
- ….

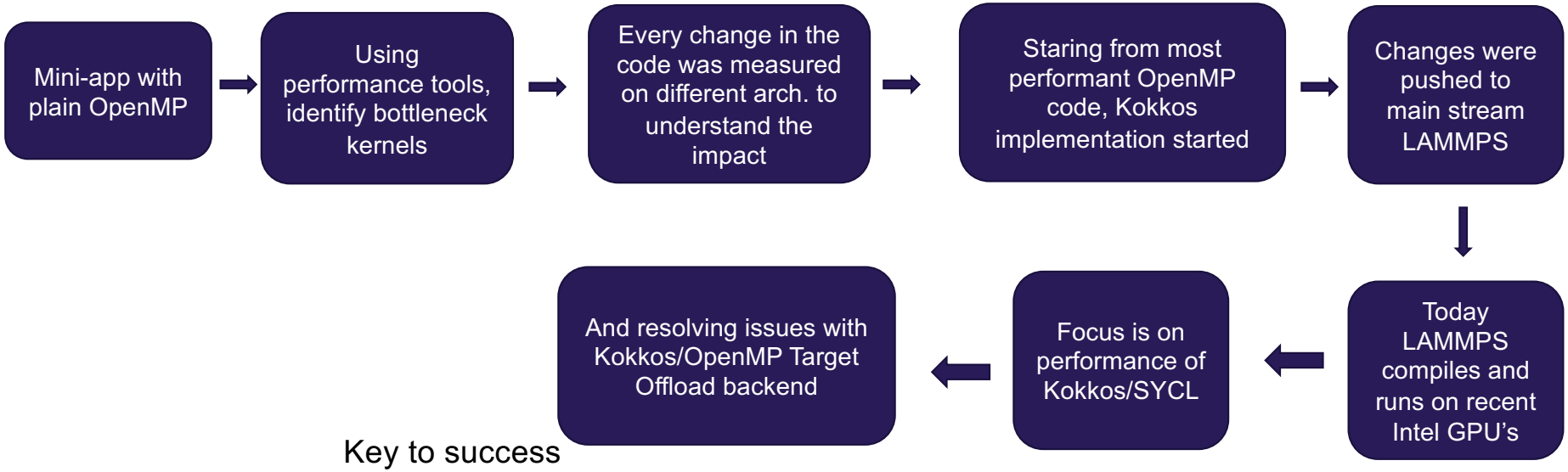
Current EXAALT Status

- **Today, we are focused on investigating performance of the EXAALT workloads in LAMMPS on Intel GPU's and also have been doing scaling studies as well.**
- There are multiple Kokkos backends enabling LAMMPS to work across DOE pre-exascale and exascale architectures.
- Many updates to enable the OpenMP Target and SYCL Kokkos backends in LAMMPS are being pushed into the public LAMMPS repo.



Work done by Stan Moore from Sandia NL

Lesson Learned and Future Plans



Key to success

Routinely running the code with new HW and SW and reporting issues

Open communication with developers and point of contact

Team Work

Acknowledgements

- Argonne Leadership Computing Facility and Computational Science Division Staff

- This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

- This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

Thanks!

Constant memory

Varsha Madananth

Constant memory

- Intel GPU does not have a dedicated memory space for constant
- Subset of device global memory address space
- `accessor` class specialized with target `target::constant_buffer` is deprecated
- Instead use device memory with read only access mode –
  - *auto inputimageacc = inputimagebuffer.get_access<access::mode::read, access::target::device_buffer>(cgh);*

Compile time constants

- Use constexpr and it will be passed to the device functions.

```
constexpr int Arows = 512;
constexpr int Acols = 512;
constexpr int Brows = 512;
constexpr int Bcols = 512

 q.submit([&](auto &h){
                h.parallel_for(range(Arows, Bcols), [=](auto index){
                        int row = index[0];
                        int col = index[1];
                        float sum = 0.0f;
                        for(int i=0; i < Acols; i++){
                          sum += A[row][i] * B[i][col];
                        }
                        C[row][col] = sum;
                });
        }).wait();
```

Setting runtime kernel constants – Specialization constants

- specialization constants are constants values can be set dynamically during execution of the application
- The values of these constants are fixed when a kernel is invoked, and they do not change during the execution of the kernel.
- Specialization constants must be declared using the `specialization_id` class
- Restrictions declaring specialization_id
  - the template parameter `T` must be a device copyable type;
  - the `specialization_id` variable must be declared as `constexpr`;
  - the `specialization_id` variable must be declared in either namespace scope or in class scope;
  - if the `specialization_id` variable is declared in class scope, it must have public accessibility when referenced from namespace scope;
  - the `specialization_id` variable may not be shadowed by another identifier `X` which has the same name and is declared in an `inline` namespace, such that the `specialization_id` variable is no longer accessible after the declaration of `X`;
  - if the `specialization_id` variable is declared in a namespace, none of the enclosing namespace names `N` may be shadowed by another identifier `X` which has the same name as `N` and is declared in an `inline` namespace, such that `N` is no longer accessible after the declaration of `X`.

Setting and getting the value of a specialization constant

- Functions of class kernel handler –
  - set_specialization_constant
  - get_specialization_constant

template<auto& SpecName> void set_specialization_constant(typename std::remove_reference_t<decltype(SpecName)>::value_type value);	*template<auto& SpecName> typename std::remove_reference_t<decltype(SpecName)> ::value_type get_specialization_constant();*

Example usage

```
#include <sycl/sycl.hpp>
using namespace sycl; // (optional) avoids need for
"sycl::" before SYCL names

using coeff_t = std::array<std::array<float, 3>, 3>;

// Read coefficients from somewhere.
coeff_t get_coefficients();

// Identify the specialization constant.
constexpr specialization_id<coeff_t> coeff_id;

void do_conv(buffer<float, 2> in, buffer<float, 2>
out) {
  queue myQueue;

  myQueue.submit([&](handler &cgh) {
    accessor in_acc { in, cgh, read_only };
    accessor out_acc { out, cgh, write_only };
```

```
    // Set the coefficient of the convolution as
constant.
    // This will build a specific kernel the coefficient
available as literals.

cgh.set_specialization_constant<coeff_id>(get_coe
fficients());

    cgh.parallel_for<class Convolution>(
        in.get_range(), [=](item<2> item_id,
kernel_handler h) {
          float acc = 0;
          coeff_t coeff =
h.get_specialization_constant<coeff_id>();
          for (int i = -1; i <= 1; i++) {
            if (item_id[0] + i < 0 || item_id[0] + i >=
in_acc.get_range()[0])
              continue;
            for (int j = -1; j <= 1; j++) {
              if (item_id[1] + j < 0 || item_id[1] + j >=
in_acc.get_range()[1])
                continue;
              // The underlying JIT can see all the values
of the array returned
              // by coeff.get().
              acc += coeff[i + 1][j + 1] *
                     in_acc[item_id[0] + i][item_id[1] + j];
            }
          }
```

References

- https://www.khronos.org/registry/SYCL/specs/sycl-2020/html/sycl-2020.html#_specialization_constants
- https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/proposed/sycl_ext_oneapi_device_global.asciidoc

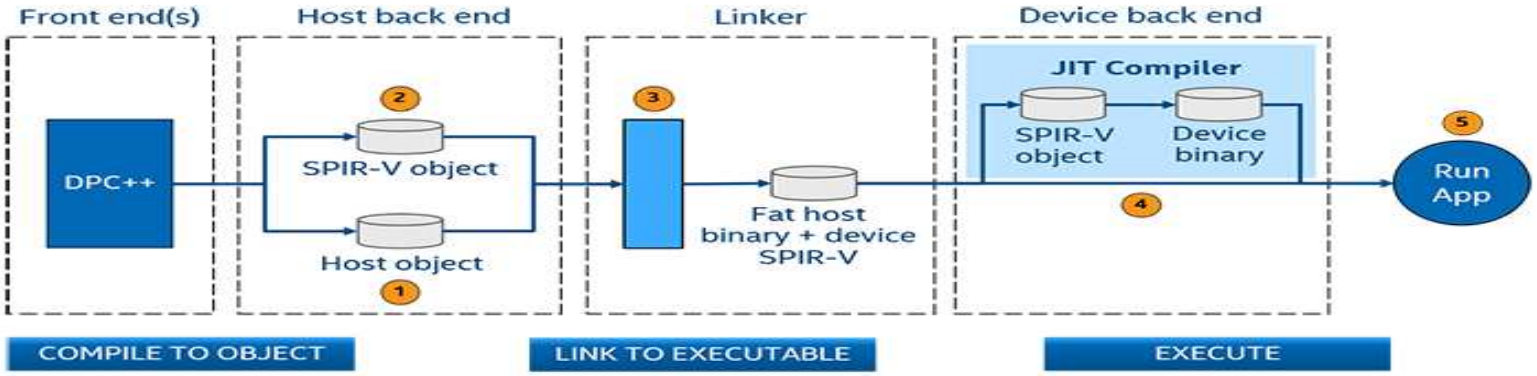
Tips and tricks for debugging your application with SYCL compiler

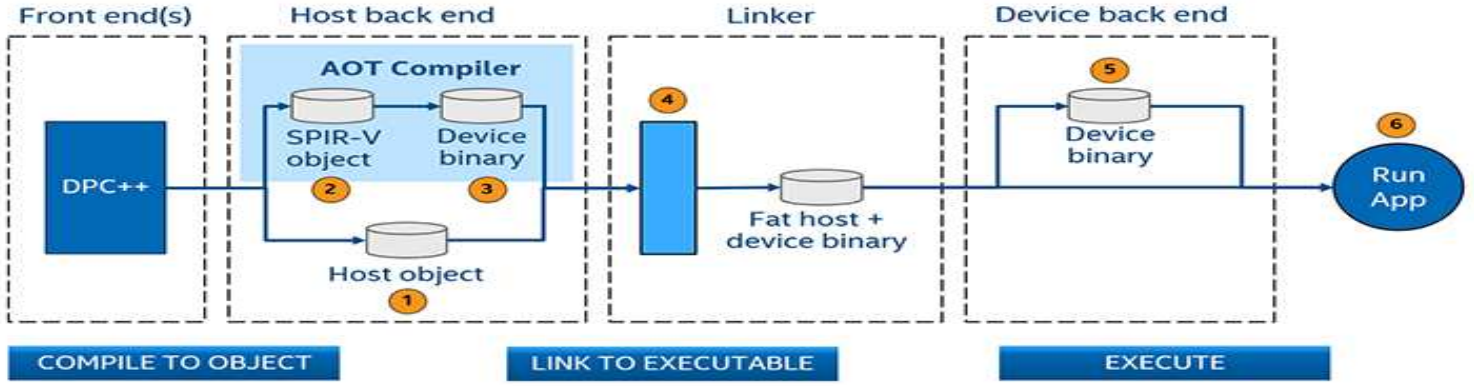
Varsha Madananth

compiler information options

- sycl compiler from oneAPI is based out of open-source clang.
- Preprocessor definitions defined by the compiler
  - icpx -fsycl test.cpp -dM -E
  - dpcpp test.cpp –dM –E
- To check the clang version the compiler is based out of –
  - icx -x c /dev/null -dM -E|grep clang
- Internal options passed by clang to the driver
  - dpcpp test.cpp -#
  - icpx test.cpp -#

JIT vs AOT compilation workflow





Identifying which phase the compilation is failing

- Use –### or –v options to see the underlying steps being done by compiler for JIT and AOT.
- --save-temps to store temporary files
- ocloc is a tool for managing Intel Compute GPU device binary format. It can be used for generation (as part of 'compile' command) as well as manipulation (decoding/modifying - as part of 'disasm'/'asm' commands) of such binary files.
  - To see all options supported by ocloc –
    - ocloc -compile –help
    - ocloc –internal_options – to add the Intel IGC specific options.

Triaging host / device optimizations

- Check optimization phases enabled by the compiler and control the optimization phases passed to the compiler.
  - Host and device side optimizations : icpx –mllvm –opt-bisect-limit=-1 test.cpp
  - Controlling Device side FE optimizations : feature request in process.
- Controlling IGC optimizations
  - SYCL_PROGRAM_COMPILE_OPTIONS="-cl-opt-disable" SYCL_LINK_LINK_OPTIONS="-cl-opt-disable“

Debugging calls at runtime

- To trace level zero calls and get basic profiling information.
  - https://github.com/intel/pti-gpu/tree/master/tools/onetrace

  --call-logging [-c] Trace host API calls

  --host-timing [-h] Report host API execution time

  --device-timing [-d] Report kernels execution time

- https://github.com/intel/llvm/blob/sycl/sycl/doc/EnvironmentVariables.md

Questions