IWOCL 2024

The 12th International Workshop on OpenCL and SYCL

Enabling RAJA on Intel GPUs with SYCL

Brian Homerding, Argonne National Laboratory

Arturo Vargas, Lawrence Livermore National Laboratory
Tom Scogland, Lawrence Livermore National Laboratory
Robert Chen, Lawrence Livermore National Laboratory
Mike Davis, Lawrence Livermore National Laboratory
Rich Hornung, Lawrence Livermore National Laboratory

RAJA Provides HPC Programming Portability

- US Department of Energy Exascale systems provide GPU-accelerated computing
  - —Intel
  - —AMD
  - —NVIDIA

- Scientific applications need to run efficiently across these different architectures

- Application developers have an increased interest in open portable programming models such as SYCL

- RAJA is another open approach with existing codes which provides portability at a higher level of abstraction.
  - —Abstracted parallel execution over other parallel programming models

Argonne NATIONAL LABORATORY

The RAJA Portability Suite

RAJA Portability Suite

RAJA

Performance portability abstraction

- Enable portability with manageable disruption
  - —Exposes tuning knobs
- Achieve performance comparable to using underlying programming model directly
  - —Low overhead

UMPIRE

Resource management library

- Abstraction over resources
- Memory management for NUMA memory hierarchies
- Memory discovery and provision

CHAI

C++ array abstractions

- Automates data copies
- Software support for unified memory

CAMP

Low-level C++ metaprogramming facilities

- Helps ensure compiler compatibility for HPC systems
- Supports abstract resource interface

RAJA Portability Suite

RAJA

Performance portability abstraction

- Enable portability with manageable disruption
  —Exposes tuning knobs
- Achieve performance comparable to using underlying programming model directly
  —Low overhead

UMPIRE

Resource management library

- Abstraction over resources
- Memory management for NUMA memory hierarchies
- Memory discovery and provision

CHAI

C++ array abstractions

- Automates data copies
- Software support for unified memory

CAMP

Low-level C++ metaprogramming facilities

- Helps ensure compiler compatibility for HPC systems
- Supports abstract resource interface

RAJA Execution Layer

- The RAJA execution layer is a set of portable abstractions for **simple and complex loops**.
- Allows applications to be developed with **decoupled loop bodies and execution**.
- Loop execution can be **tuned for a specific target** without modification of the loop body
- Loop execution is controlled through ex**ecution policy defined in template argument**.

Example: RAJA forall loop

```
// Defined in header file
using EXEC_POL= RAJA::omp_parallel_for_exec;

// Kernel code in application source.
RAJA::forall<EXEC_POL>(RAJA::RangeSegment(0, N),
                                   [=] (int i) {
     c[i] = a[i] + b[i];
});
```

Example: RAJA forall loop

Execution Template

```
// Defined in header file
using EXEC_POL= RAJA::omp_parallel_for_exec;

// Kernel code in application source.
RAJA::forall<EXEC_POL>(RAJA::RangeSegment(0, N),
                        [=] (int i) {
     c[i] = a[i] + b[i];
});
```

- Execution templates for different parallel patterns

Example: RAJA forall loop

```
// Defined in header file
using EXEC_POL= RAJA::omp_parallel_for_exec;

// Kernel code in application source.
RAJA::forall<EXEC_POL>(RAJA::RangeSegment(0, N),
                       [=] (int i) {
     c[i] = a[i] + b[i];
});
```

Execution Policy Type

- Execution policy specifies backend execution parameters

Example: RAJA forall loop

```
// Defined in header file
using EXEC_POL= RAJA::omp_parallel_for_exec;

// Kernel code in application source.
RAJA::forall<EXEC_POL>(RAJA::RangeSegment(0, N),
                         [=] (int i) {
     c[i] = a[i] + b[i];
});
```

Iteration Space

- Iteration space defined the space or spaces the kernel executes

Example: RAJA forall loop

```
// Defined in header file
using EXEC_POL= RAJA::omp_parallel_for_exec;

// Kernel code in application source.
RAJA::forall<EXEC_POL>(RAJA::RangeSegment(0, N),
                                              [=] (int i) {
          c[i] = a[i] + b[i];
});
```

Loop Body

- Loop body is executed as a C++ lambda

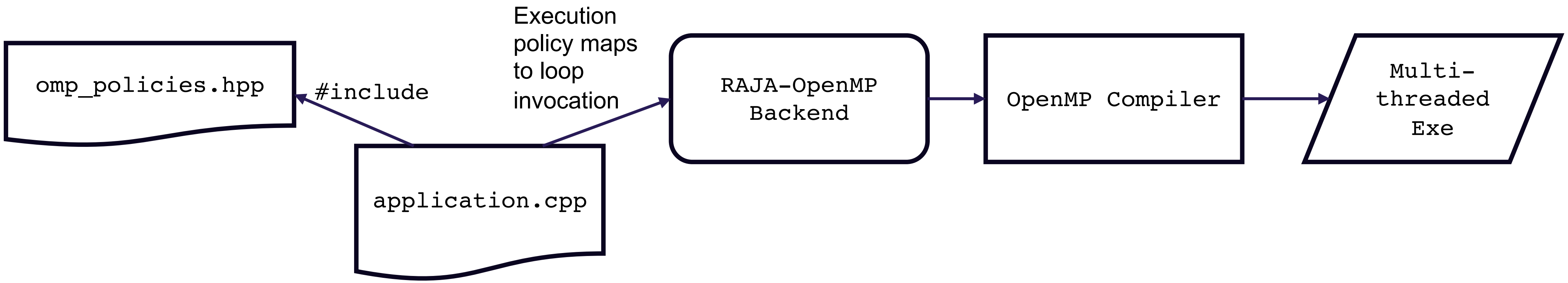
RAJA Execution Layer Design Goals

- Clean encapsulation of kernel code.
  - Separation of the implementation of the algorithm and the implementation of the execution.
- Easy customization of kernel execution.
  - Porting to new system is easy and doesn't modify kernel code.
- Incremental and selective adoption
  - Can utilize direct backend programming model in addition to RAJA
  - Can use backend language features in RAJA kernel
- Systematic performance tuning
  - Give control to the developer for kernel execution

High Level View of RAJA Application

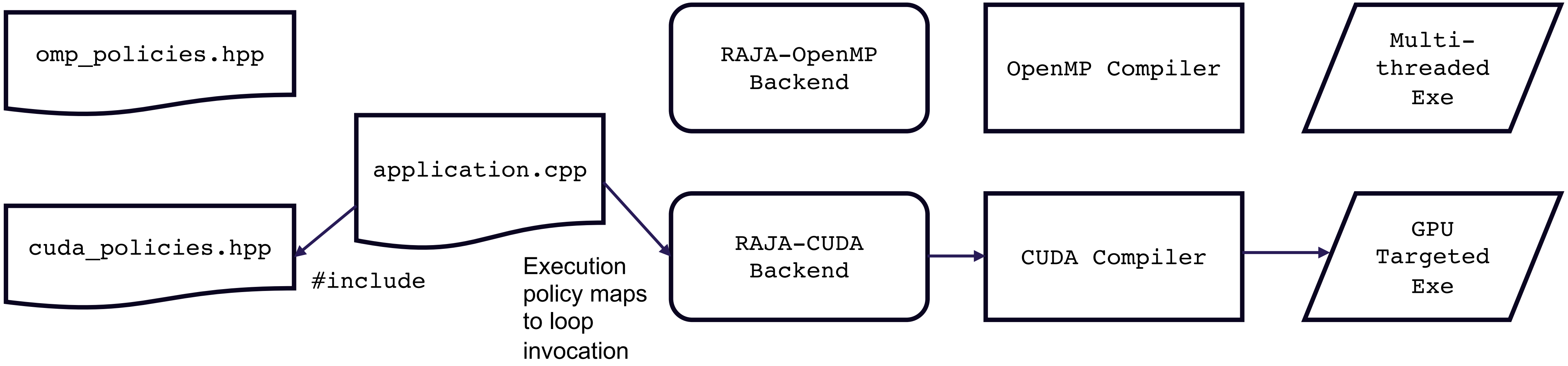
- Based on execution policy kernel will map to a given loop invocation (C++ lambda)

- Existing compiler with target backend support is utilized



High Level View of RAJA Application

- Based on execution policy kernel will map to a given loop invocation (C++ lambda)
- Existing compiler with target backend support is utilized
- When changing targets, new execution policies and appropriate compiler are needed



Adding support for Intel GPUs

- We developed a SYCL backend in RAJA for Intel GPU support
- SYCL provides explicit controls for kernel execution which can be leveraged as tuning parameters for RAJA
- Able to leverage Intel SYCL compiler support for Intel GPUs

SYCL Backend Implementation

Simple RAJA Kernel Mapping

```
using EXEC_POL= RAJA::sycl_exec<work_group_size, true /*async*/>;

RAJA::forall<EXEC_POL>(RAJA::RangeSegment(ibegin, iend),
                       [=] (Index_type i) {
        INIT3_BODY;
    });
```

Corresponding SYCL kernel execution

```
const size_t global_size = work_group_size
                           * RAJA_DIVIDE_CEILING_INT(iend, work_group_size);

        qu->submit([&] (sycl::handler& h) {
          h.parallel_for(sycl::nd_range<1>(global_size, work_group_size),
                                           [=] (sycl::nd_item<1> item ) {


            Index_type i = item.get_global_id(0);
            if (i < iend) {
              INIT3_BODY
            }

          });
        });
```

SYCL Backend – forall

```
template <typename Iterable, typename LoopBody, size_t BlockSize, bool Async, typename ForallParam,
          typename std::enable_if<std::is_trivially_copyable<LoopBody>{},bool>::type = true>
RAJA_INLINE
concepts::enable_if_t<resources::EventProxy<resources::Sycl>,
                      RAJA::expt::type_traits::is_ForallParamPack<ForallParam>,
                      RAJA::expt::type_traits::is_ForallParamPack_empty<ForallParam>>
forall_impl(resources::Sycl &sycl_res, sycl_exec<BlockSize, Async>,
            Iterable&& iter, LoopBody&& loop_body, ForallParam)
{

  using Iterator  = camp::decay<decltype(std::begin(iter))>;
  using LOOP_BODY = camp::decay<LoopBody>;
  using IndexType = camp::decay<decltype(std::distance(std::begin(iter), std::end(iter)))>;

  // Compute the requested iteration space size
  Iterator begin = std::begin(iter);
  Iterator end = std::end(iter);
  IndexType len = std::distance(begin, end);

  // Only launch kernel if we have something to iterate over
  if (len > 0 && BlockSize > 0) {

    // Compute the number of blocks
    sycl_dim_t blockSize{BlockSize};
    sycl_dim_t gridSize = impl::getGridDim(static_cast<size_t>(len), BlockSize);

    q = sycl_res.get_queue();
    q->submit([&](cl::sycl::handler& h) {

      h.parallel_for( cl::sycl::nd_range<1>{gridSize, blockSize},
                      [=]  (cl::sycl::nd_item<1> it) {

        IndexType ii = it.get_global_id(0);
        if (ii < len) { loop_body(begin[ii]); }
      });
    });

    if (!Async) { q->wait(); }
  }

  return resources::EventProxy<resources::Sycl>(sycl_res);
}
```

SYCL Backend – Unnamed Lambda, Trivially Copyable

```
template <typename Iterable, typename LoopBody, size_t BlockSize, bool Async, typename ForallParam,
          typename std::enable_if<std::is_trivially_copyable<LoopBody>{},bool>::type = true>
RAJA_INLINE
concepts::enable_if_t<resources::EventProxy<resources::Sycl>,
                      RAJA::expt::type_traits::is_ForallParamPack<ForallParam>,
                      RAJA::expt::type_traits::is_ForallParamPack_empty<ForallParam>>
forall_impl(resources::Sycl &sycl_res, sycl_exec<BlockSize, Async>,
            Iterable&& iter, LoopBody&& loop_body, ForallParam)
{

  using Iterator  = camp::decay<decltype(std::begin(iter))>;
  using LOOP_BODY = camp::decay<LoopBody>;
  using IndexType = camp::decay<decltype(std::distance(std::begin(iter), std::end(iter)))>;

  // Compute the requested iteration space size
  Iterator begin = std::begin(iter);
  Iterator end = std::end(iter);
  IndexType len = std::distance(begin, end);

  // Only launch kernel if we have something to iterate over
  if (len > 0 && BlockSize > 0) {

    // Compute the number of blocks
    sycl_dim_t blockSize{BlockSize};
    sycl_dim_t gridSize = impl::getGridDim(static_cast<size_t>(len), BlockSize);

    q = sycl_res.get_queue();
    q->submit([&](cl::sycl::handler& h) {

      h.parallel_for( cl::sycl::nd_range<1>{gridSize, blockSize},
                      [=]  (cl::sycl::nd_item<1> it) {

        IndexType ii = it.get_global_id(0);
        if (ii < len) { loop_body(begin[ii]); }
      });
    });

    if (!Async) { q->wait(); }
  }

  return resources::EventProxy<resources::Sycl>(sycl_res);
}
```

SYCL Backend – RAJA Resources

```
template <typename Iterable, typename LoopBody, size_t BlockSize, bool Async, typename ForallParam,
          typename std::enable_if<std::is_trivially_copyable<LoopBody>{},bool>::type = true>
RAJA_INLINE
concepts::enable_if_t<resources::EventProxy<resources::Sycl>,
                      RAJA::expt::type_traits::is_ForallParamPack<ForallParam>,
                      RAJA::expt::type_traits::is_ForallParamPack_empty<ForallParam>>
forall_impl(resources::Sycl &sycl_res, sycl_exec<BlockSize, Async>,
            Iterable&& iter, LoopBody&& loop_body, ForallParam)
{

  using Iterator  = camp::decay<decltype(std::begin(iter))>;
  using LOOP_BODY = camp::decay<LoopBody>;
  using IndexType = camp::decay<decltype(std::distance(std::begin(iter), std::end(iter)))>;

  // Compute the requested iteration space size
  Iterator begin = std::begin(iter);
  Iterator end = std::end(iter);
  IndexType len = std::distance(begin, end);

  // Only launch kernel if we have something to iterate over
  if (len > 0 && BlockSize > 0) {

    // Compute the number of blocks
    sycl_dim_t blockSize{BlockSize};
    sycl_dim_t gridSize = impl::getGridDim(static_cast<size_t>(len), BlockSize);

    q = sycl_res.get_queue();
    q->submit([&](cl::sycl::handler& h) {

      h.parallel_for( cl::sycl::nd_range<1>{gridSize, blockSize},
                      [=]  (cl::sycl::nd_item<1> it) {

        IndexType ii = it.get_global_id(0);
        if (ii < len) { loop_body(begin[ii]); }
      });
    });

    if (!Async) { q->wait(); }
  }

  return resources::EventProxy<resources::Sycl>(sycl_res);
}
```

SYCL queue/context

```
// Defined in header file.
using EXEC_POL = RAJA::sycl_exec<work_group_size>;
// Kernel code in application source.
RAJA::forall< EXEC_POL >(RAJA::RangeSegment(0, N), [=] (int i) {
  c[i] = a[i] + b[i];
});
```

RAJA Resources – New API

```
// Defined in header file.
using EXEC_POL = RAJA::sycl_exec<work_group_size>;
// Kernel code in application source.
RAJA::forall< EXEC_POL >(RAJA::RangeSegment(0, N), [=] (int i) {
  c[i] = a[i] + b[i];
});
```

```
// Defined in header file.
using EXEC_POL = RAJA::sycl_exec<work_group_size>;
using RESOURCE = RAJA::resources::Sycl;

// Kernel code in application source.
RESOURCE my_res;
RAJA::forall< EXEC_POL >(my_res, RAJA::RangeSegment(0, N), [=] (int i) {
    c[i] = a[i] + b[i];
});
```

RAJA Resources – CAMP

- In addition to the new API, applications can rely on CAMP to provide a default in order context.

```
namespace camp
{
namespace resources
{ ...

    class Sycl
    {
      static sycl::queue *get_a_queue(sycl::context &syclContext,
                                      int num,
                                      bool useContext)
      {
        static sycl::gpu_selector gpuSelector;
        static sycl::property_list propertyList =
            sycl::property_list(sycl::property::queue::in_order());
        static sycl::context privateContext;
        {...}
```

- A similar solution was introduced as an extension in the Intel OneAPI SYCL implementation

RAJA Reduction

```
// Defined in header file.
using EXEC_POL = RAJA::sycl_exec<work_group_size>;
using REDUCTION_T = RAJA::ReduceSum< RAJA::sycl_reduce, int >;

// Kernel code in application source.
REDUCTION_T sum(0);
RAJA::forall< EXEC_POL >(RAJA::RangeSegment(0, N),
                         [=] (int i) {

    sum += a[i];
});

int my_sum = sum.get();
```

RAJA Reduction

```
// Defined in header file.
using EXEC_POL = RAJA::sycl_exec<work_group_size>;
using REDUCTION_T = RAJA::ReduceSum< RAJA::sycl_reduce, int >;

// Kernel code in application source.
REDUCTION_T sum(0);
RAJA::forall< EXEC_POL >(RAJA::RangeSegment(0, N),
                         [=] (int i) {

    sum += a[i];
});

int my_sum = sum.get();
```

Allocate local memory?

Overloaded +=:
What is my index?

New RAJA Reduction Interface

```
// Defined in header file.
using EXEC_POL = RAJA::sycl_exec<work_group_size>;

// Kernel code in application source.
int sum = 0;
RAJA::forall< EXEC_POL >(RAJA::RangeSegment(0, N),
                         RAJA::expt::Reduce< RAJA::operators::plus >(&sum),
                         [=] (int i, int& _sum) {


    _sum += a[i];
});

doWork(sum);
```

- The new reduction interface allows the SYCL backend to leverage the compiler reduction support

RAJA SYCL Backend Support Features and Policies

```
RAJA::forall
RAJA::kernel
RAJA::launch

RAJA::Reduce
RAJA::expt::Reduce

RAJA::resources::Sycl

RAJA::sycl_exec<work_group_size, async>
Sycl::launch_t
RAJA::SyclKernel
RAJA::SyclKernelAsync


RAJA::sycl_reduce

RAJA::sycl_atomic
RAJA::sycl_atomic_explicit
```

```
RAJA::sycl_global_0<BLOCK_SIZE>
RAJA::sycl_global_1<BLOCK_SIZE>
RAJA::sycl_global_2<BLOCK_SIZE>

RAJA::sycl_global_item_0
RAJA::sycl_global_item_1
RAJA::sycl_global_item_2

RAJA::sycl_group_0_direct
RAJA::sycl_group_1_direct
RAJA::sycl_group_2_direct

RAJA::sycl_group_0_loop
RAJA::sycl_group_1_loop
RAJA::sycl_group_2_loop

RAJA::sycl_local_0_direct
RAJA::sycl_local_1_direct
RAJA::sycl_local_2_direct

RAJA::sycl_local_1_loop
RAJA::sycl_local_2_loop
RAJA::sycl_local_3_loop
```

Evaluation

Argonne
NATIONAL LABORATORY

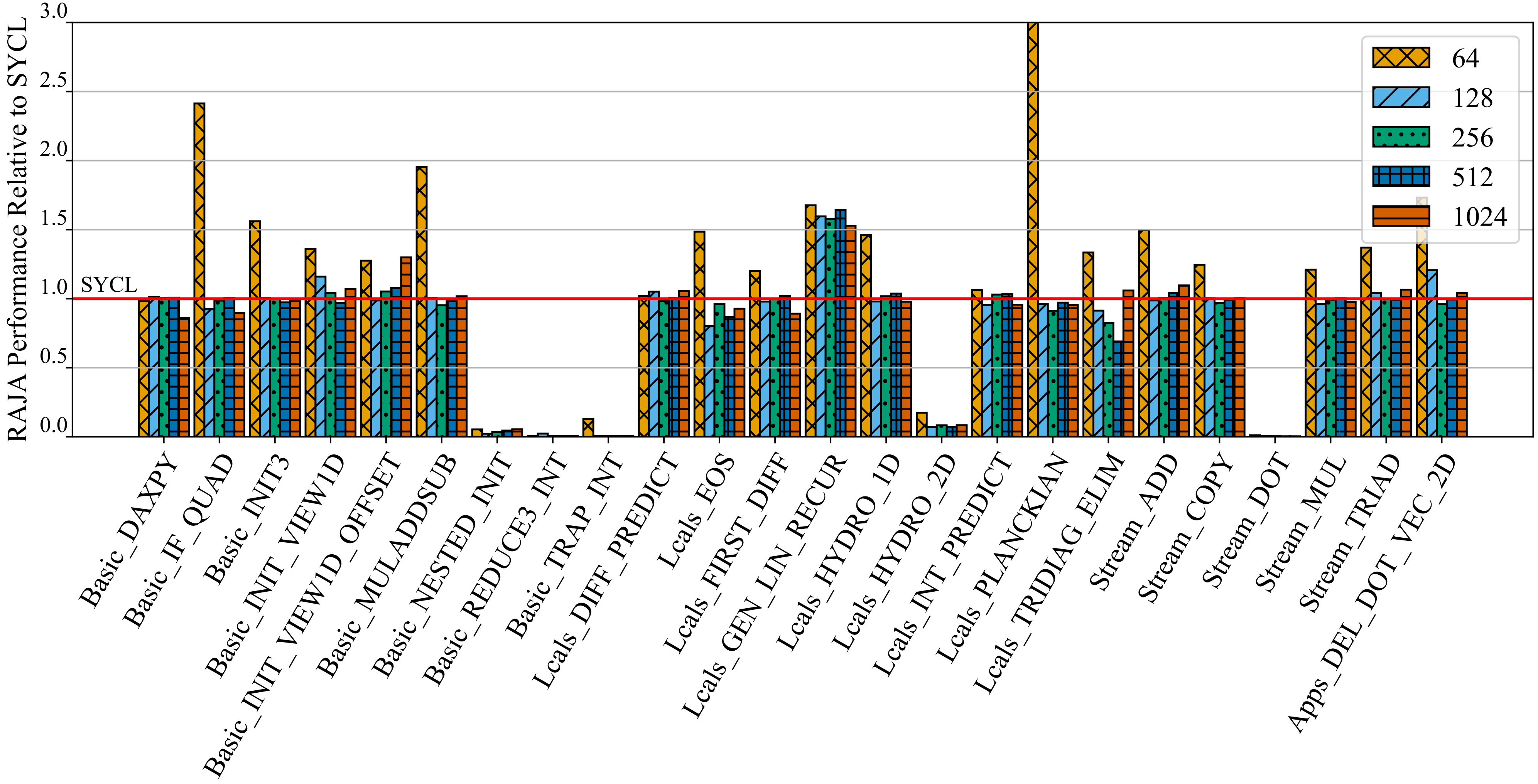
RAJA Performance Suite

- Primary developer – Rich Hornung (LLNL)
  - See RAJAPerf github page for full list of contributors
- Very good for compiler testing
- Built in timer and correctness testing.
  - Timer cover full execution of many repetitions the kernels
  - Correctness is done with checksum compared against sequential execution
  -
- Many "Variants"
  - `Base_Seq, Lambda_Seq, RAJA_Seq, Base_OpenMP, Lambda_OpenMP, RAJA_OpenMP, Base_OpenMPTarget, RAJA_OpenMPTarget, Base_CUDA, RAJA_CUDA, Base_SYCL, RAJA_SYCL, etc`

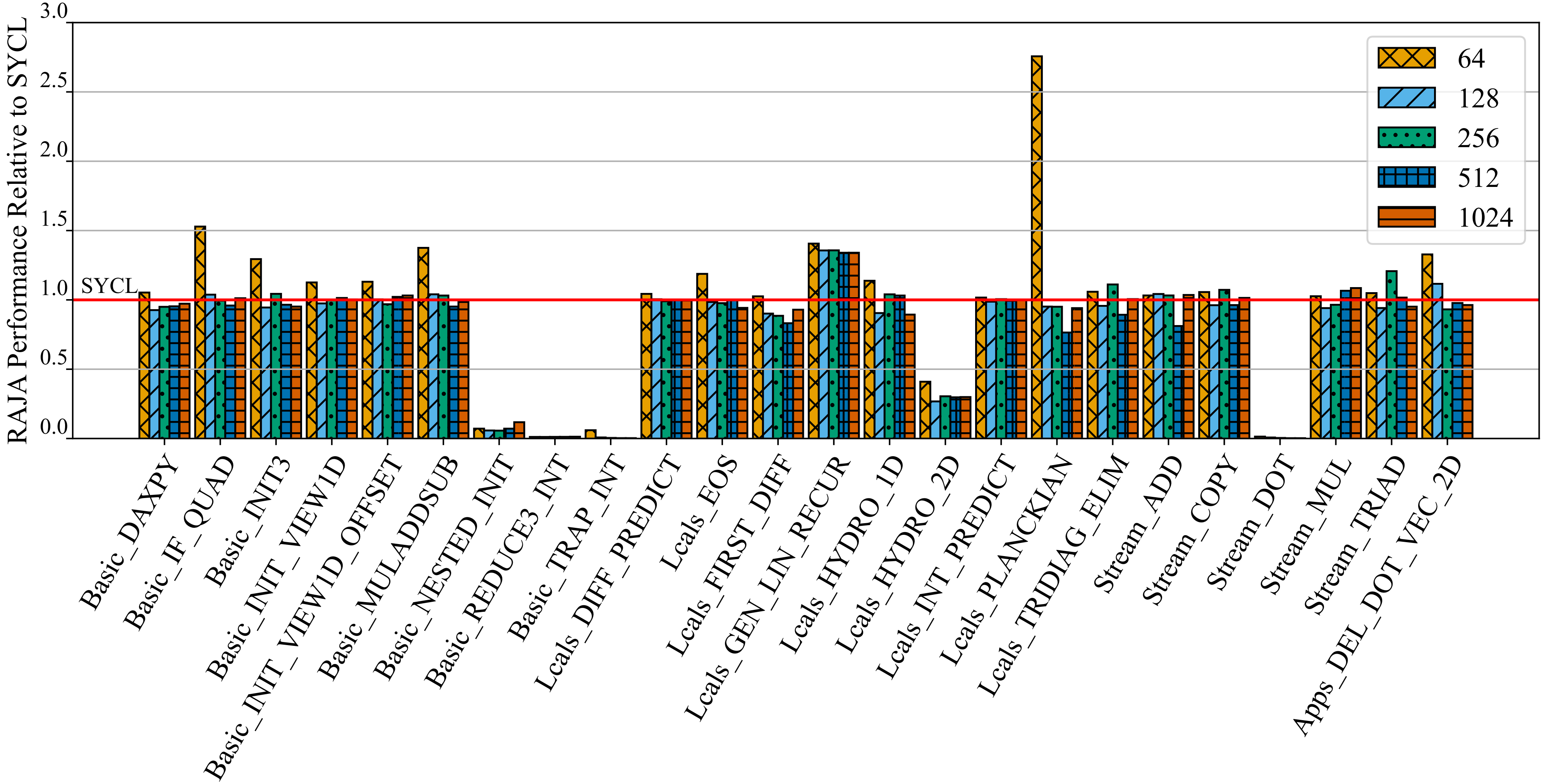
How the RAJA Performance Suite Works

- Variants, number of repetitions, size of kernels configurable as options
- Runs warm up kernels (subset of all kernels)
- GPU variants are templated on work group sizes at compile time
- Timer wraps repetition loop, computes averages
- Correctness is done with a checksum against baseline (Base_Seq by default)
    - —Considered a failure beyond 1e-6

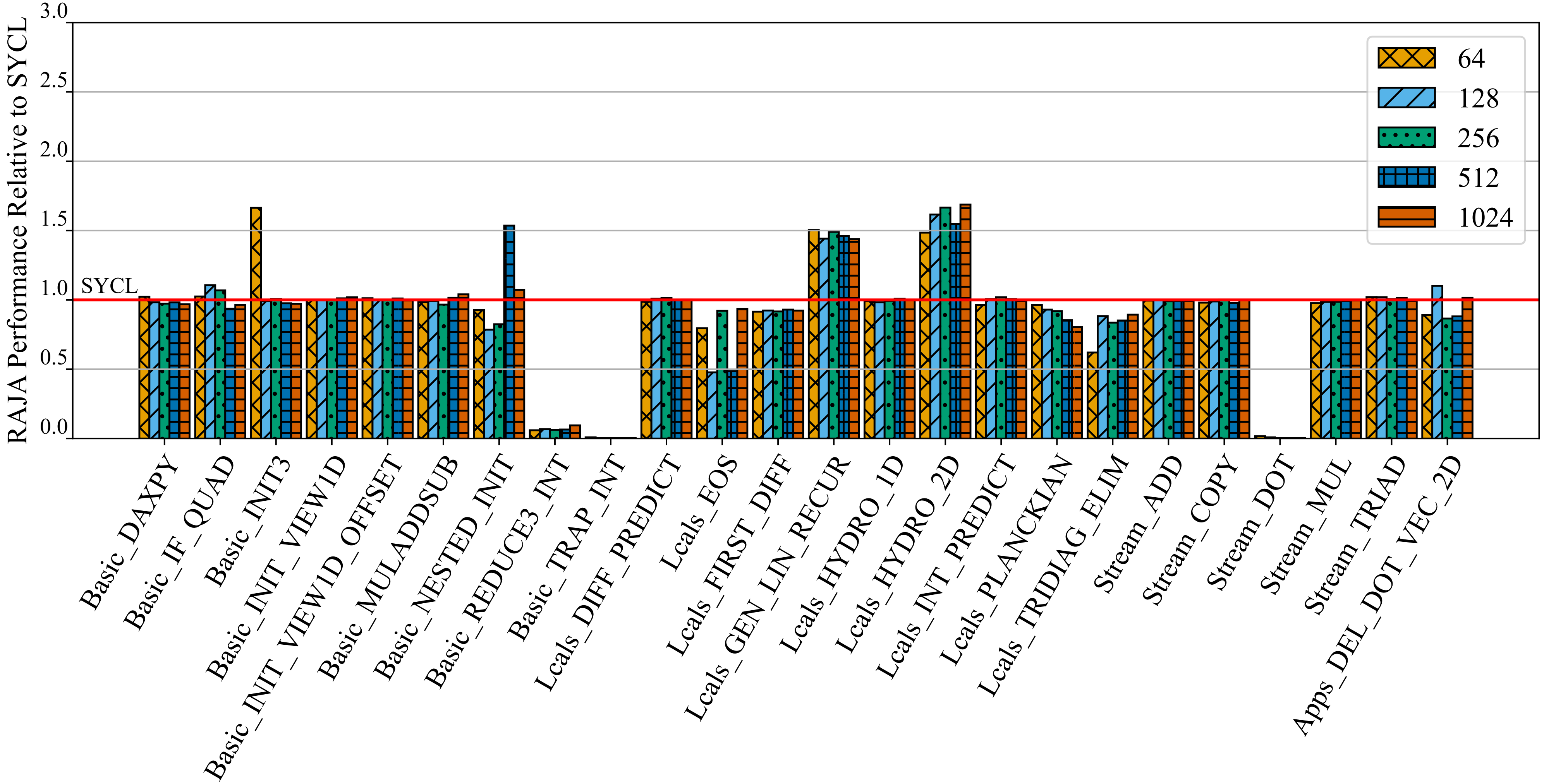
RAJA Performance vs SYCL Direct (Total overhead)



5X Problem Size RAJA Performance vs SYCL Direct



RAJA Kernel Performance vs SYCL Direct (iprof)



Conclusion

- RAJA SYCL backend enabled RAJA execution on Intel GPUs
  - Also works on AMD and NVIDIA
- Continued development to complete the support for the remaining RAJA features (eg. scans)
- Continuing to implement SYCL variants of RAJA Performance Suite kernels


Argonne
NATIONAL LABORATORY

Ackowledgements

- This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations - the Office of Science and the National Nuclear Security Administration, responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, to support the nation's exascale computing imperative.
- This work was also 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.
- This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE- AC02-06CH11357.
- We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.


Argonne
NATIONAL LABORATORY