OAK RIDGE National Laboratory

The Kokkos Ecosystem and its SYCL backend

Daniel Arndt

June 24, 2024

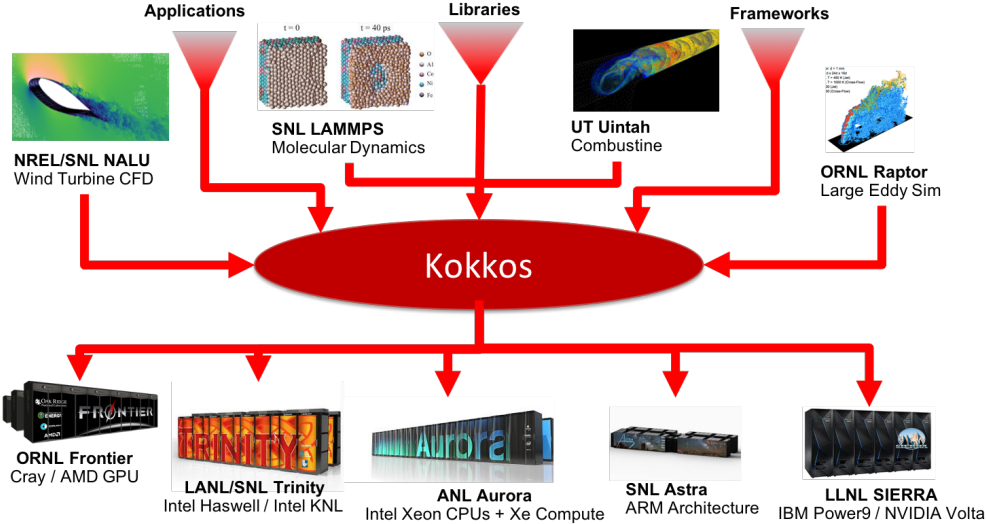
ORNL is managed by UT-Battelle LLC for the US Department of Energy

U.S. DEPARTMENT OF ENERGY

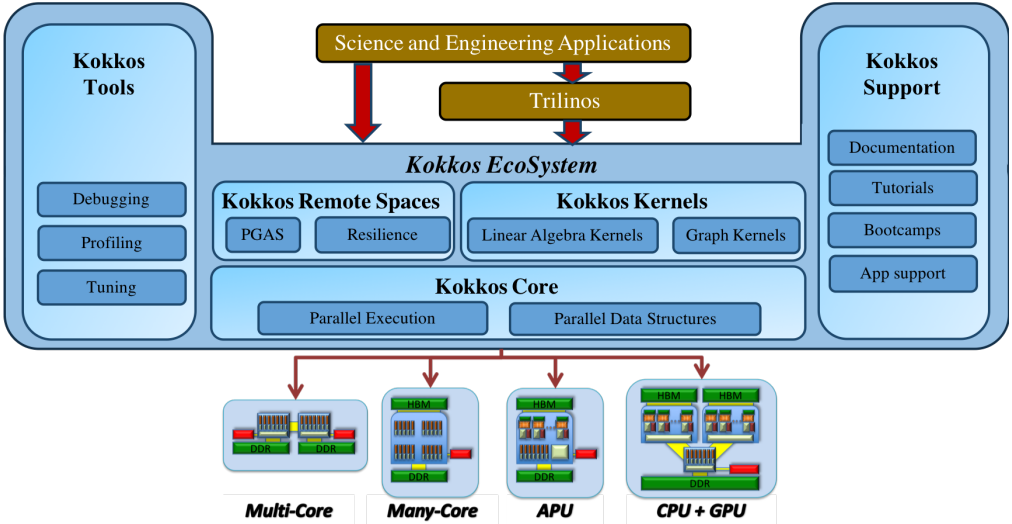
What is Kokkos?

- A C++ Programming Model for Performance Portability
  - Template library on top of CUDA, HIP, OpenMP, SYCL, ...
  - Aligns with developments in the C++ standard, e.g., `mdspan`, `atomic_ref`, `stdBLAS`
- Expanding solution for common needs of modern science and engineering codes
  - Math libraries based on Kokkos
  - Tools for debugging, profiling and tuning
  - Interoperability with Fortran and Python
- Open Source project with a growing community
  - Maintained and developed at `https://github.com/kokkos`
  - Hundreds of users at many large institutions

Kokkos as Portability Layer



Kokkos Ecosystem



Kokkos Core:	C. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, C. Clevenger, N. Ellingwood, R. Gayatri, E. Harvey, D. Ibanez, D. Lee, N. Liber, N. Morales, A. Powell, M. Simberg, B. Turcksin
Kokkos Kernels:	S. Rajamanickam, L. Berger-Vergiat, V. Dang, N. Ellingwood, E. Harvey, K. Kim, J. Loe, C. Pearson
Kokkos Tools	C.R. Trott, V. Kale, D. Lebrun-Grandie, D. Ibanez, S. Moore, A. Powell

Online Resources

- `https://github.com/kokkos`:
  - Primary Kokkos GitHub Organization
- `https://github.com/kokkos/kokkos-tutorials/wiki/Kokkos-Lecture-Series`:
  - Slides, recording and Q&A for the Full Lectures
- `https://kokkos.org/kokkos-core-wiki/`:
  - Wiki including API reference
- `https://kokkosteam.slack.com`:
  - Slack channel for Kokkos.
  - Please join: fastest way to get your questions answered.
  - Can whitelist domains, or invite individual people.

Kokkos' basic capabilities:

- Simple 1D data parallel computational patterns
- Deciding where code is run and where data is placed
- Managing data access patterns for performance portability
- Multidimensional data parallelism

Kokkos' advanced capabilities:

- Thread safety, thread scalability, and atomic operations
- Hierarchical patterns for maximizing parallelism
- Task based programming with Kokkos

Kokkos' tools and Kernels:

- How to profile, tune and debug Kokkos code
- Interacting with Python and Fortran
- Using KokkosKernels math library

Why Kokkos

- High performance computers are increasingly heterogenous
  MPI-only is no longer sufficient.
- For portability: OpenMP, OpenACC, ... or Kokkos.
- Kokkos obtains performant memory access patterns via architecture-aware arrays and work mapping.
  i.e., not just portable, performance portable.
- With Kokkos, simple things stay simple (parallel-for, etc.).
  i.e., it's no more difficult than OpenMP.
- Advanced performance-optimizing patterns are simpler with Kokkos than with native versions.
  i.e., you're not missing out on advanced features.
- Use vendor toolchains

Performance Portability

There's a difference between portability and performance portability.

Example: implementations may target particular architectures and may not be thread scalable.

(e.g., locks on CPU won't scale to 100,000 threads on GPU)

Goal: write one implementation which:

- compiles and runs on multiple architectures,
- obtains performant memory access patterns across architectures,
- can leverage architecture-specific features where possible.

Kokkos: performance portability across manycore architectures.

Experiences with implementing Kokkos' SYCL backend

Primary Programming Models: OpenMP, Cuda, HIP, SYCL

LANL Crossroads
Intel CPUs,
OpenMP, SYCL?

LBNL Perlmutter
AMD CPU, NVIDIA GPU
CUDA

ORNL Frontier
AMD CPU, AMD GPU
HIP

ANL Aurora
Intel CPUs, Intel GPUs
SYCL

LLNL El Capitan
AMD CPU, AMD GPU
HIP

Kokkos Core Functionalities, Mapping to SYCL

Constructs

- `parallel_for` ⟶ `sycl::parallel_for`
- `parallel_reduce` ⟶ `sycl::parallel_for`
- `parallel_scan` ⟶ `sycl::parallel_for`

Policies

- `RangePolicy` ⟶ `sycl::range`
- `MDRangePolicy` ⟶ `sycl::nd_range`
- `TeamPolicy` ⟶ `sycl::nd_range`

Memory

- `View` ⟶ `sycl::malloc/sycl::free`
- `View` is a reference-counted multi-dimensional-array

Feature Status Kokkos+SYCL targeting Intel GPUs

Feature complete apart from

- `WorkGraphPolicy`
- `Tasks`
- `Graphs` (in progress)
- Virtual functions/function pointer (lacking compiler support)

parallel_for RangePolicy

```
Kokkos::parallel_for(
  Kokkos::RangePolicy(execution_space, start, end)),
  KOKKOS_LAMBDA(int i) {/*...*/});
```

is mapped to SYCL code as

```
Functor functor;
q.parallel_for(sycl::range<1>(end - start),
  [=](sycl::id<1> idx) {
    int i = idx + start;
    functor(i);
});
```

Kokkos iota

```
#include <Kokkos_Core.hpp>
int main() {
  Kokkos::ScopeGuard scope_guard;
  Kokkos::View<int*> view("view", 100);
  Kokkos::parallel_for(100, KOKKOS_LAMBDA(int i){view(i) = i;});
}
```

⇒ Using `sycl::buffer` for `Kokkos::View` not feasible.

Problem:

- `Kokkos::View` is not trivially copyable.
- `sycl::is_device_copyable`?

`sycl::is_device_copyable`

> It is unspecified whether the implementation actually calls the copy constructor, move constructor, copy assignment operator, or move assignment operator of a class declared as *`is_device_copyable_v`* when doing an inter-device copy.
>
> [...]
>
> Likewise, it is unspecified whether the implementation actually calls the destructor for such a class on the device since the destructor must have no effect on the device.

Issue:

- Implementations actually call special member functions[1]
- We need another workaround!

[1] `https://github.com/intel/llvm/issues/5320`

`sycl::is_device_copyable` - Workaround I

```
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;
};
```

`sycl::is_device_copyable` - Workaround II

```
template <typename Functor>
class SYCLFunctionWrapper {
  union TrivialWrapper m_functor;
 public:
  SYCLFunctionWrapper(const Functor& functor, Storage&)
    : m_functor(functor) {}
  const Functor& get_functor() const {
    return m_functor.m_f;
  }
};

template <typename Functor>
struct sycl::is_device_copyable<
    SYCLFunctionWrapper<Functor, Storage, false>>
    : std::true_type {};
```

MDRangePolicy

`MDRangePolicy` maps up to 6 dimensions with tiling to three dimensions in `sycl::nd_range`

```
struct Functor{
  KOKKOS_FUNCTION void operator(
    int i, int j, int k, int l, int m) const {/*...*/}
};
Kokkos::parallel_for(
  Kokkos::MDRangePolicy(execution_space,
    {s0,s1,s2,s3,s4}, {e0,e1,e2,e3,e4}),
  Functor{});
```

TeamPolicy

```
parallel_for("Label", TeamPolicy<>(numberOfTeams, teamSize, vectorLength),
  KOKKOS_LAMBDA (const member_type & teamMember) {
    /* beginning of outer body */
    parallel_for(TeamThreadRange(teamMember, thisTeamsRangeSize),
      [=] (const int indexWithinBatch[, ...]) {
        /* begin middle body */
        parallel_for(ThreadVectorRange(teamMember, thisVectorRangeSize),
            [=] (const int indexVectorRange) {/* inner body */});
        /* end middle body */
      });
    parallel_for(TeamVectorRange(teamMember, someSize),
      [=] (const int indexTeamVector) {/* nested body */});
    /* end of outer body */
  });
```

TeamPolicy

Only policy to allow scratch allocations.

Mappings:

- team → `sycl::group`
- multiple threads → `sycl::subgroup`

Problem:

- thread synchronization requires a barrier on part of a subgroup → `tangle_group` or similar
- communicating address space information for scratch allocations

parallel_reduce

```
double result;
Kokkos::parallel_reduce(
  Kokkos::RangePolicy(execution_space, start, end)),
  KOKKOS_LAMBDA(int i, double& partial_sum) {
    partial_sum += i;
  }, result);
```

doesn't use SYCL's reduction variables. Features:

- simple reductions (sum)
- multiple reductions per parallel construct
- custom reductions with arbitrary value types and reduction operations
- runtime sized array reductions
- pre- and post-callbacks for reductions (`init`, `final`)

parallel_reduce

```
per_thread:
  value& tmp=init(local_tmp);
  for (i in local range)
    functor(i, tmp)
call join for merging values between threads
  in the same workgroup
let one (the last) workgroup merge all results
  from all workgroups
call final(result) on one thread
```

Shuffle-based implementation gives worse results than using local memory on Intel GPUs.

parallel_scan

```
Kokkos::parallel_scan(
  Kokkos::RangePolicy(execution_space, start, end),
  KOKKOS_LAMBDA (const int index, value_type& update,
                 const bool is_final) {
    const value_type local_value = in_data(i);
    // exclusive scan
    if (is_final)
      out_data_exclusive(i) = update;
    update += local_value;
    // inclusive scan
    if (is_final)
      out_data_inclusive(i) = update;
 });
```

parallel_scan

```
first kernel:
  per_thread:
    value& tmp=init(local_tmp);
    for (i in local range)
      functor(i, tmp, /*is_final*/ false)
  call join for implementing a prefix sum
    in the same workgroup
  let the last workgroup compute the prefix sum for the
    totals of all workgroups and store the result
  store intermediate results on each thread
second kernel:
  combine workgroup totals with thread intermediate results
  call the functor again for final result (with final=true)
```

As opposed to `parallel_reduce` a shuffle-based implementation is used.

Performance comparisons

AXPBY - parallel_for

DOT - parallel_reduce

SPMV - TeamPolicy

Conjugate Gradient

One node of `Sunspot` with Intel® Data Center GPU Max 1550 GPUs
peak memory bandwidth 3276.8 GB/s (2 tiles)

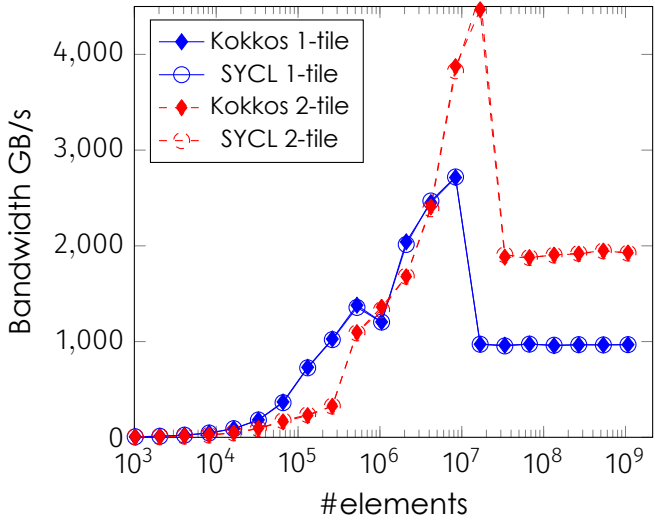
`https://github.com/kokkos/code-examples/tree/main/papers/IWOCL2024`

AXPBY benchmark

```
// Kokkos
for (int r = 0; r < R; r++) {
  Kokkos::parallel_for("axpby", N, KOKKOS_LAMBDA(int i) {
   z(i) = alpha*x(i) + beta*y(i);
  });
}

// SYCL
sycl::queue q{sycl::property::queue::in_order()} ;
for (int r = 0; r < R; r++) {
  q.parallel_for(sycl::range<1>(N), [=](sycl::id<1> idx){
    int i = idx;
    z[i] = alpha*x[i] + beta*y[i];
  });
}
```

Achieved effective bandwidth for the AXPBY benchmark

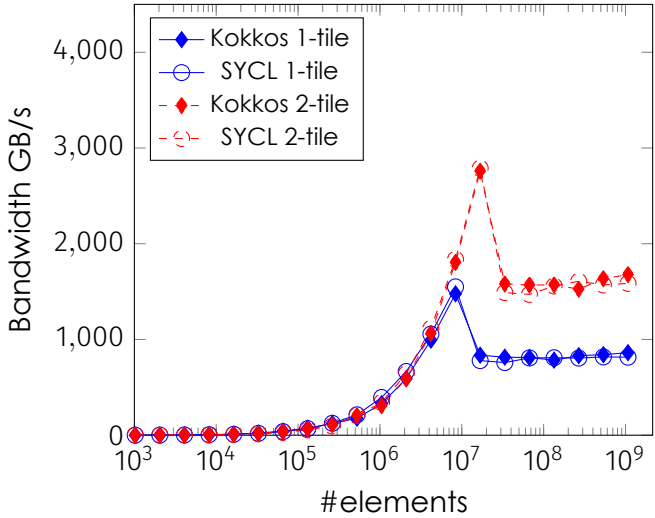


DOT benchmark

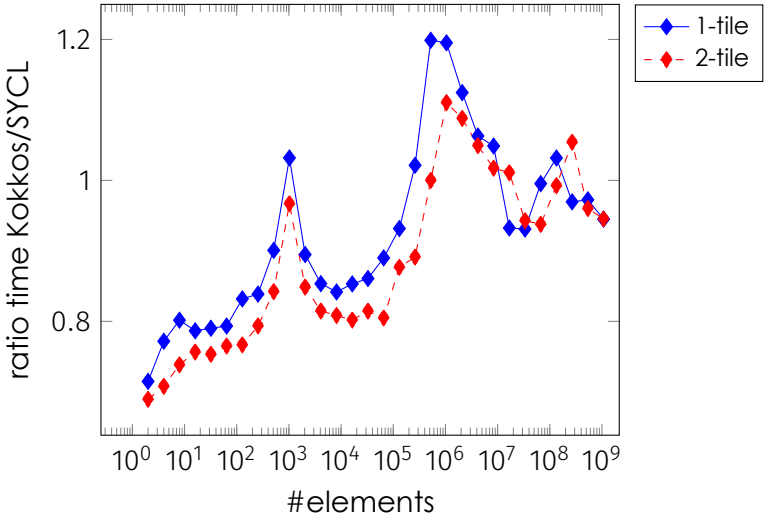
```
// Kokkos
for (int r = 0; r < R; r++) {
  Kokkos::parallel_reduce("dot", N,
    KOKKOS_LAMBDA(int i, double& sum) {sum += x(i) * y(i);},
  result);

// SYCL
sycl::queue q{sycl::property::queue::in_order()} ;
for (int r = 0; r < R; r++) {
  q.parallel_for(sycl::range<1>(N_),
    sycl::reduction(result_ptr, 0., sycl::plus<double>()),
      [=](sycl::id<1> idx, auto&sum) {sum += x[idx] * y[idx];});
  q.memcpy(&result, result_ptr, sizeof(double));
  q.wait();
}
```

Achieved effective bandwidth for the DOT benchmark



Run time ratio for DOT between Kokkos and SYCL



SPMV benchmark - Kokkos I

```
int rows_per_team = 32; //optimized for GPU
int team_size = 16;     //optimized for GPU
int vector_size = 4;    //optimized for GPU
int n_teams = (nrows + rows_per_team - 1)/rows_per_team;
using TeamMember = Kokkos::TeamPolicy<>::member_type;
// parallelize over the row blocks
Kokkos::parallel_for("SPMV",
  Kokkos::TeamPolicy<>(n_teams, team_size, vector_size),
    KOKKOS_LAMBDA(const TeamMember &team) {
      int64_t first_row=team.league_rank()*rows_per_team;
      int64_t last_row=first_row + rows_per_team < nrows
        ? first_row + rows_per_team : nrows;
```

SPMV benchmark - Kokkos II

```
// parallelize over rows owned by the team
Kokkos::parallel_for(
  Kokkos::TeamThreadRange(team,first_row,last_row),
    [&](const int64_t row) {
      const int64_t row_start = A.row_ptr(row);
      const int64_t row_length = A.row_ptr(row + 1) - row_start;
      // perform the dot-product of a matrix row with vector
      Kokkos::parallel_reduce(
        Kokkos::ThreadVectorRange(team,row_length),
          [=](const int64_t i, double &sum) {
            sum += A.values(i + row_start) * x(A.col_idx(i + row_start));
          }, y(row));
    });
});
```

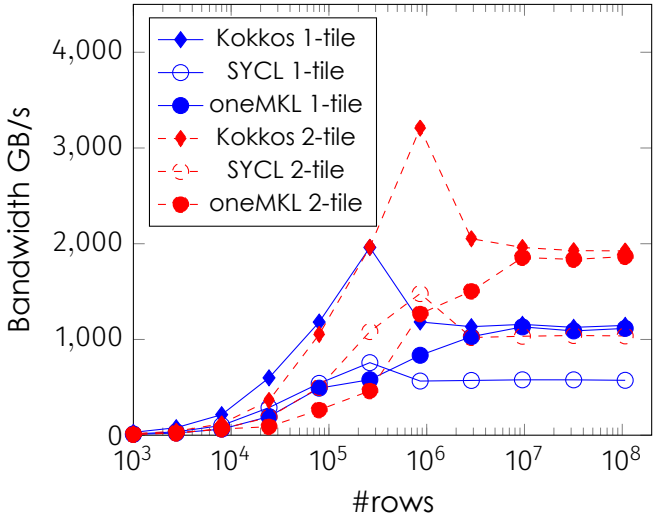
SPMV benchmark - SYCL I

```
int rows_per_team = 32; //optimized for GPU
int team_size = 16;     //optimized for GPU
int n_teams = (nrows + rows_per_team - 1)/rows_per_team;
q.submit([&] (sycl::handler& cgh) {
  // parallelize over the row blocks
  cgh.parallel_for_work_group(sycl::range<1>(n),
    sycl::range<1>(team_size), [=](sycl::group<1> g) {
      int64_t first_row= g.get_group_id(0)*rows_per_team;
      int64_t last_row=first_row + rows_per_team < nrows
        ? first_row + rows_per_team : nrows;
```

SPMV benchmark - SYCL II

```
        // parallelize over rows owned by the team
        g.parallel_for_work_item(
          sycl::range<1>(last_row-first_row),
          [&](sycl::h_item<1> item) {
            int64_t row = item.get_local_id(0)+first_row;
            int64_t row_start = row_ptr[row];
            int64_t row_length = row_ptr[row+1]-row_start;
            double y_row = 0.;
            for (int64_t i = 0; i < row_length; ++i)
              y_row += values[i + row_start] * xp[col_idx[i + row_start]];
            yp[row] = y_row;
      });
    });
  });
```

Achieved effective bandwidth for the SPMV benchmark on the GPU



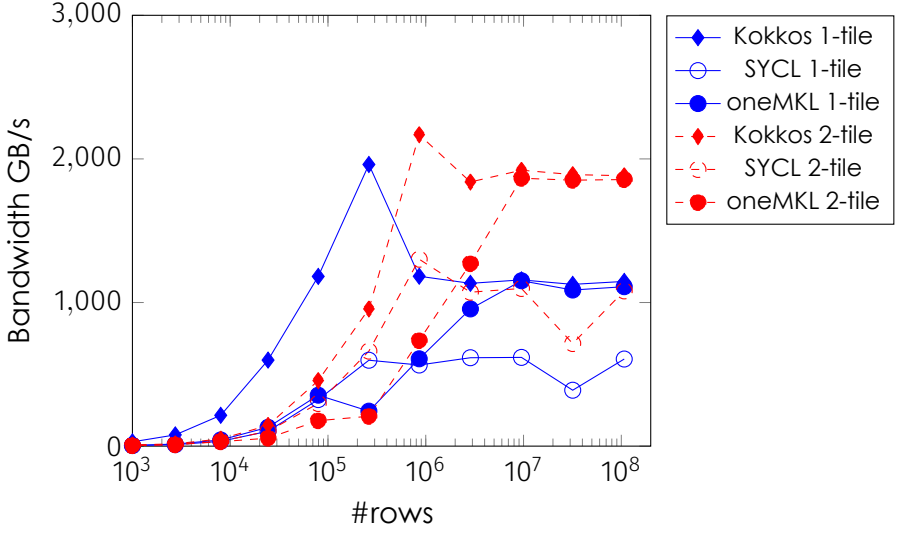
CG benchmark I

```
for (int64_t k = 1; k <= max_iter && normr > tolerance; ++k) {
  if (k == 1) {
    axpby(p, one, r, zero, r);
  } else {
    oldrtrans = rtrans;
    rtrans = dot(r, r);
    double beta = rtrans / oldrtrans;
    axpby(p, one, r, beta, p);
  }
  normr = std::sqrt(rtrans);
  double alpha = 0;
  double p_ap_dot = 0;
  spmv(Ap, A, p);
```

CG benchmark II

```
    p_ap_dot = dot(Ap, p);
    if (p_ap_dot < brkdown_tol) {
      if (p_ap_dot < 0) {
        std::cerr << "numerical breakdown!\n";
        return num_iters;
      } else
        brkdown_tol = 0.1 * p_ap_dot;
    }
    alpha = rtrans / p_ap_dot;
    axpby(x, one, x, alpha, p);
    axpby(r, one, r, -alpha, Ap);
    num_iters = k;
  }
```

Achieved effective bandwidth for the CG benchmark on the GPU



Summary

Despite some hiccups, `SYCL/DPC++`

- integration was pretty smooth
- works well on Intel GPUs (Aurora, Dawn, …)
- works much better than `OpenMPTarget`
- has better support for newer C++ features than `nvcc`

Questions?

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, 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.

- This work was done on a pre-production supercomputer with early versions of the Aurora software development kit.

Used Extensions

- `sycl::ext::oneapi::experimental::this_nd_item`
- `sycl::ext::oneapi::experimental::printf`
- `sycl::ext::oneapi::experimental::device_global`
- `sycl::ext::oneapi::experimental::properties`
- `sycl::ext::oneapi::experimental::device_image_scope`
- `sycl::ext::oneapi::experimental::this_sub_group`
- `sycl::ext::oneapi::group_ballot`
- `sycl::ext::oneapi::sub_group_mask`
- `sycl::ext::oneapi::experimental::bfloat16`
- `sycl::ext::oneapi::group_local_memory_for_overwrite`