JANUARY 28, 2026

Overview of CRK-HACC Development Efforts for Aurora

Esteban M. Rangel
Assistant Computational Scientist
Computational Science Division
Argonne National Laboratory

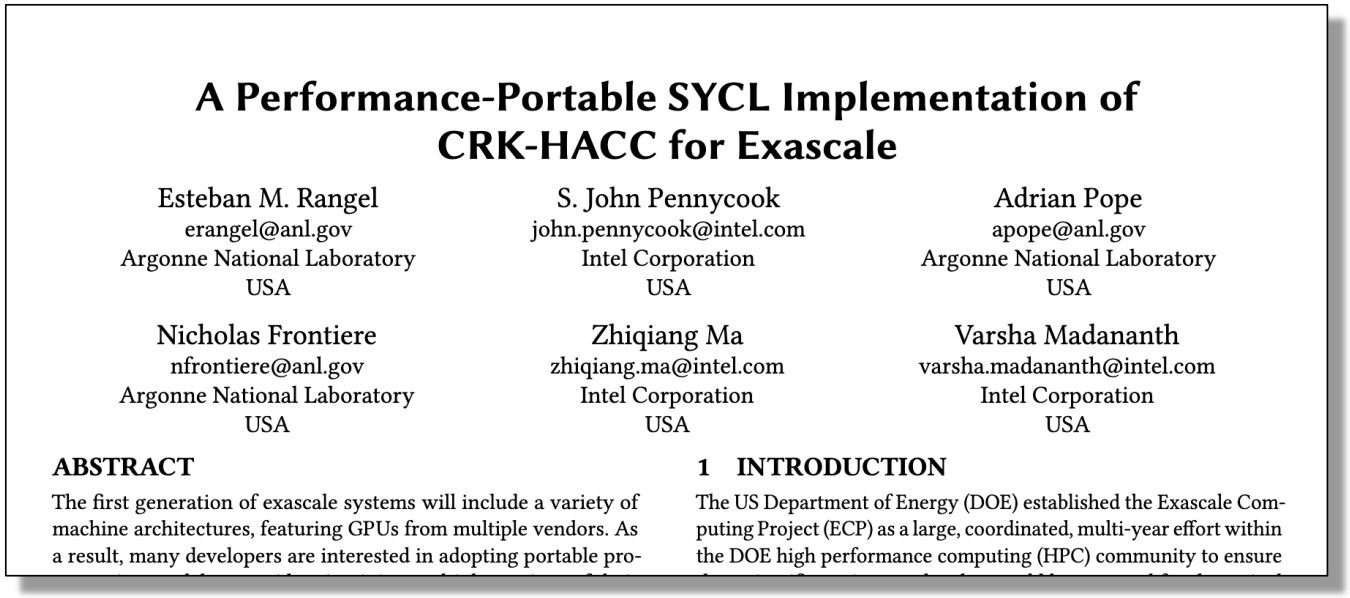
ALCF Developer Sessions

U.S. DEPARTMENT of ENERGY
Argonne National Laboratory is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC.

Argonne NATIONAL LABORATORY

PRIOR WORK – PERFORMANCE PORTABILITY OF CRK-HACC

Are CUDA, HIP, and SYCL equally performant across architectures?



A Performance-Portable SYCL Implementation of CRK-HACC for Exascale

Esteban M. Rangel
erangel@anl.gov
Argonne National Laboratory
USA

S. John Pennycook
john.pennycook@intel.com
Intel Corporation
USA

Adrian Pope
apope@anl.gov
Argonne National Laboratory
USA

Nicholas Frontiere
nfrontiere@anl.gov
Argonne National Laboratory
USA

Zhiqiang Ma
zhiqiang.ma@intel.com
Intel Corporation
USA

Varsha Madananth
varsha.madananth@intel.com
Intel Corporation
USA

ABSTRACT

The first generation of exascale systems will include a variety of machine architectures, featuring GPUs from multiple vendors. As a result, many developers are interested in adopting portable pro-

1 INTRODUCTION

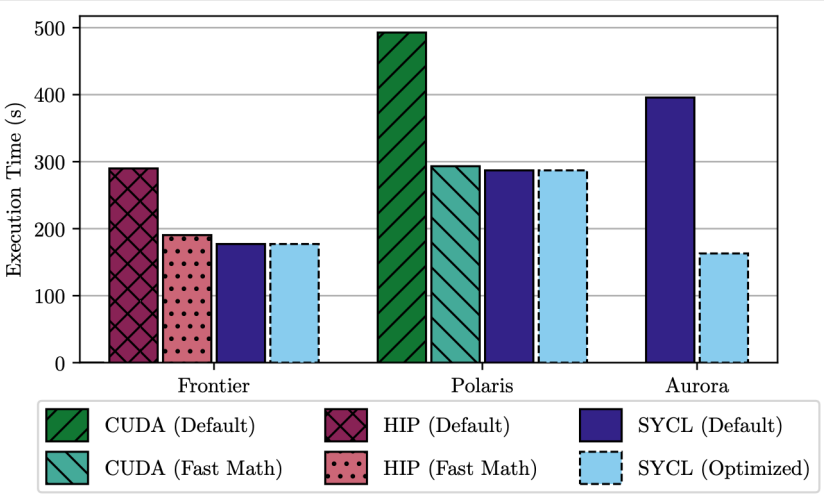
The US Department of Energy (DOE) established the Exascale Computing Project (ECP) as a large, coordinated, multi-year effort within the DOE high performance computing (HPC) community to ensure

P3HPC SC23

Short answer: *Yes! when tuned, all three models deliver* ***equivalent relative peak performance*** *across NVIDIA, AMD, and Intel GPUs for CRK-HACC's dominant kernels.*

RESULTS COMPARING SYCL TO CUDA AND HIP

Aggregate of all GPU Kernels



- Fast Math was not enabled by default on all compilers.
- Original CUDA ported to SYCL and optimized for the Intel GPU.

OPTIMIZING WARP / WORKGROUP SHUFFLE

Intel® Data Center GPU Max 1550 assembly snippets for `sycl::select-from-group`

Elements are gathered from the registers specified in **a0** and written into **r2** using *indirect register access*

```
...
shl (16|M0)  r24.0<1>:uw  r82.0<2;1,0>:uw  0x2:uw
add (16|M0)  a0.0<1>:uw   r24.0<1;1,0>:uw  0x640:uw
mov (16|M0)  r2.0<1>:ud   r[a0.0]<1,0>:ud
...
```

alternative instruction sequence employing *register regioning* is more performant but not always achievable by the compiler

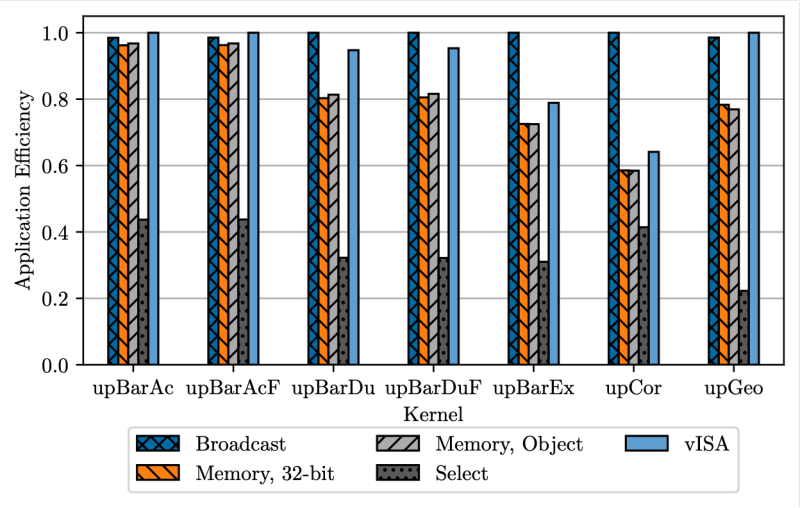
```
...
add (16|M0)  r24.0<1>:f  r68.0<1;1,0>:f  -r14.0<0;1,0>:f
add (16|M0)  r26.0<1>:f  r68.0<1;1,0>:f  -r14.1<0;1,0>:f
add (16|M0)  r30.0<1>:f  r68.0<1;1,0>:f  -r14.2<0;1,0>:f
...
```

Strategies Explored

- Broadcasts
  - Restructure loops so that sufficient information is known about the communication pattern at compile-time to generate more efficient assembly.
- Shared Local Memory
  - Uses sycl::local_accessor to reserve a small amount of work-group local memory per sub-group to communicate instead of via registers.
- Optimized Instruction Sequences
  - Explicitly code the assembly instructions for each communication step needed.

SYCL OPTIMIZATION RESULTS

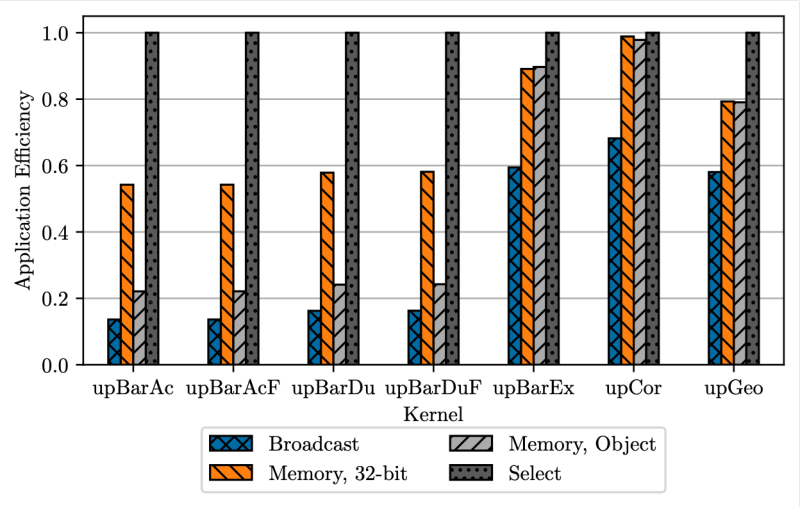
Aurora



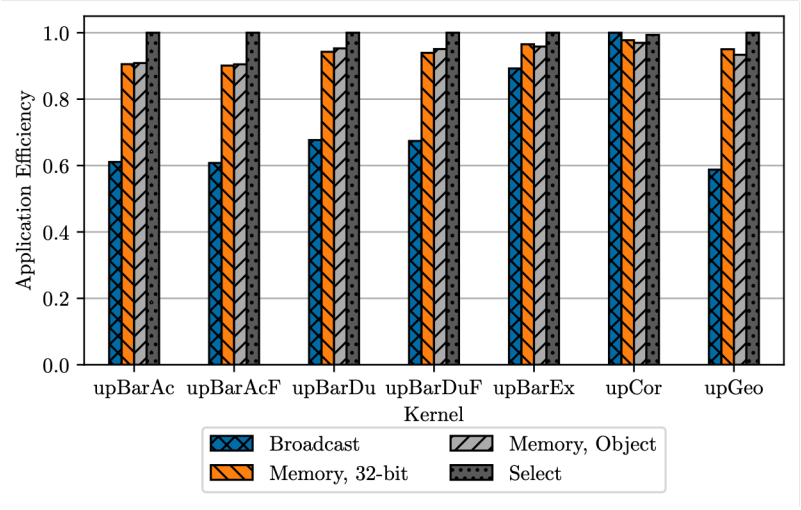
- Broadcast uses a sub-group size of 16, all other variants use a sub-group size of 32
- Restructuring the loops to use broadcasts also allows us to generate fewer atomic instructions, more noticeable in the Extras and Corrections kernels

SYCL OPTIMIZATION RESULTS

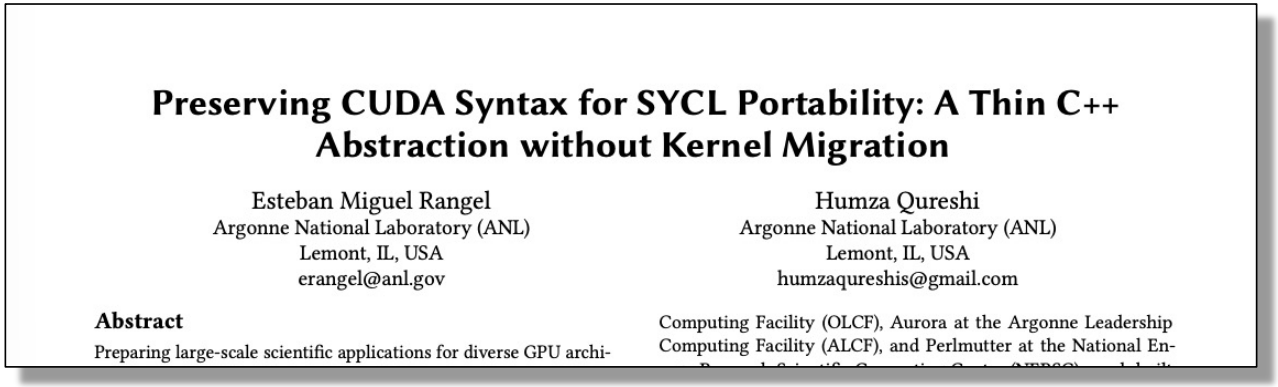
Polaris



Frontier



NEW GOAL



Preserving CUDA Syntax for SYCL Portability: A Thin C++ Abstraction without Kernel Migration

Esteban Miguel Rangel
Argonne National Laboratory (ANL)
Lemont, IL, USA
erangel@anl.gov

Humza Qureshi
Argonne National Laboratory (ANL)
Lemont, IL, USA
humzaqureshis@gmail.com

Abstract

Preparing large-scale scientific applications for diverse GPU archi-

Computing Facility (OLCF), Aurora at the Argonne Leadership Computing Facility (ALCF), and Perlmutter at the National En-

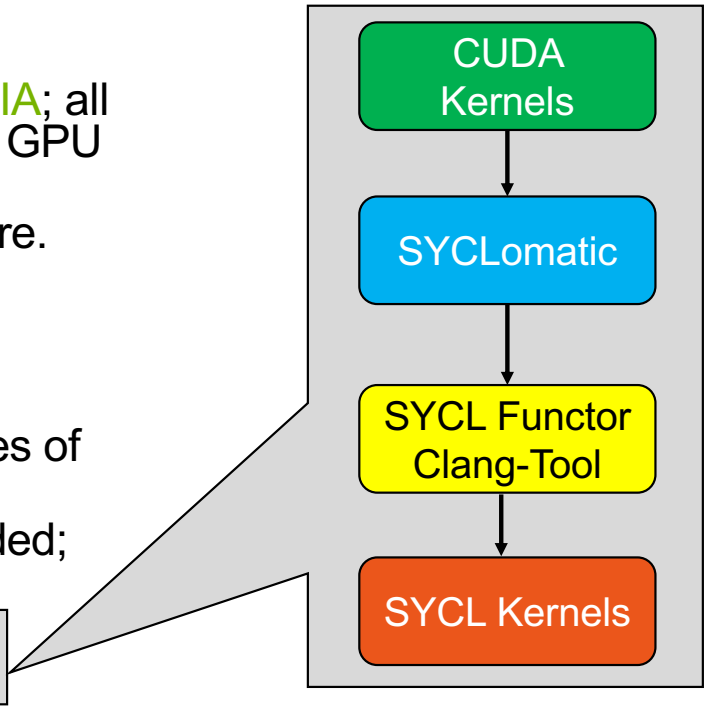
P3HPC SC25

A thin C++ layer can preserve CUDA syntax while enabling single-source SYCL portability, allowing more unified backend paths, reduced long-term maintenance costs, and retaining competitive performance across GPU vendors.

MOTIVATION & PROBLEM CONTEXT

Can we retain the highly tuned CUDA kernel bodies unchanged and still run them on SYCL targets?

1. Multi-vendor Exascale Landscape
   - Frontier = AMD, Aurora = Intel, Perlmutter = NVIDIA; all DOE open-science flagship systems use *different* GPU architectures.
   - Applications would like to run efficiently everywhere.
   - Portability models exist, but rewriting kernels or maintaining multiple backends is costly.
2. Case Study: CRK-HACC
   - Production cosmology code with thousands of lines of hand-tuned CUDA kernels.
   - New physics and optimizations are constantly added; repeated porting is not sustainable.
   - Prior SYCL ports required large source-to-source transformations and additional tuning.



MAIN IDEA

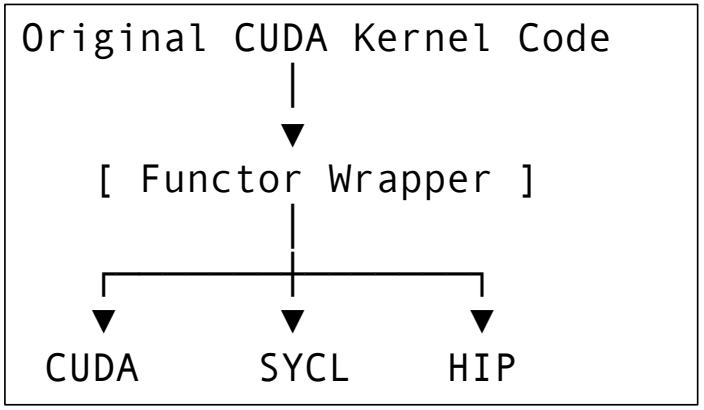
A thin, header-only C++ abstraction preserves CUDA kernel structure while compiling under either CUDA or SYCL toolchains.

- No DSL or new abstraction framework to learn.
- No AST-guided migration tools.
- No code duplication.
- Original CUDA syntax preserved.

DESIGN GOAL: PRESERVE CUDA SYNTAX, ENABLE SYCL

Preserve existing CUDA kernel bodies while exposing them to both CUDA and SYCL compilers.

- **Key Ideas**
  - — Developers keep writing CUDA-style kernels.
  - — No DSL, no rewriting, no AST/migration tools.
  - — A thin C++ layer replaces the `__global__` boundary with a functor boundary.
  - — Kernel body is unchanged.



FUNCTORIZATION: TURNING A CUDA KERNEL INTO A CALLABLE OBJECT

Each CUDA kernel becomes a C++ functor; the body is unchanged

- We replace:

```
__global__ void kernel(args...)
```

with:

```
struct kernel : KUtilKernelBase
{ void operator()(args...) }
```

- All CUDA constructs inside operator() remain intact.
- Functor instantiated by both CUDA and SYCL launchers.

```
__global__ void updateGeometry(const
    int* leafs1, ...) {
  int i = __ldg(&leafs1[blockIdx.x]);
  ...
}
```

```
struct updateGeometry : public
    KUtilKernelBase {
  using
    KUtilKernelBase::KUtilKernelBase;
  KUTIL_DEVICE void operator()(const
    int* leafs1, ...)
  {
    int i =
      __ldg(&leafs1[blockIdx.x]);
    ...
  }
};
```

Native CUDA

PRESERVING CUDA SEMANTICS THROUGH THE BASE CLASS

The base class binds CUDA-like semantics on SYCL: thread indices, warp intrinsics, atomics, fast math.

- threadIdx, blockIdx, blockDim mapped to sycl::nd_item
- Warp intrinsics mapped to SYCL subgroup or SLM operations
- CUDA math (`rsqrtf`, `__ldg`, `__popc`) aliased to SYCL native equivalents
- All inline, header-only; no overhead

Index binding

```
// In KUtilKernelBase (SYCL path)
threadIdx = {item.get_local_id(0), 0, 0};
blockIdx = {item.get_group(0), 0, 0};
```

Warp shuffle

```
template <typename T>
inline T __shfl(T v, int lane) {
  auto sg = m_item.get_sub_group();
  return sycl::select_from_group(sg, v, lane);
}
```

Fast math alias

```
inline float rsqrtf(float x) {
  return sycl::native::rsqrt(x);
}
```

LAUNCH INFRASTRUCTURE: ONE MACRO, TWO BACKENDS

Both CUDA and SYCL use the same launch macro; translation occurs entirely in headers

- InvokeGPUKernel(Kernel, grid, block, args...)
- Under CUDA: expands to a traditional <<<...>>> stub
- Under SYCL: creates a parallel_for with matching nd_range
- Optional shared memory forwarded correctly
- No device relocatable code (RDC) needed, avoids overhead

User code

```
InvokeGPUKernel(updateGeometry,
grid, block, leafs1, leafs2, xx);
```

CUDA backend

```
updateGeometry_kernel<<<grid, block>>>
(leafs1, leafs2, xx);
```

SYCL backend

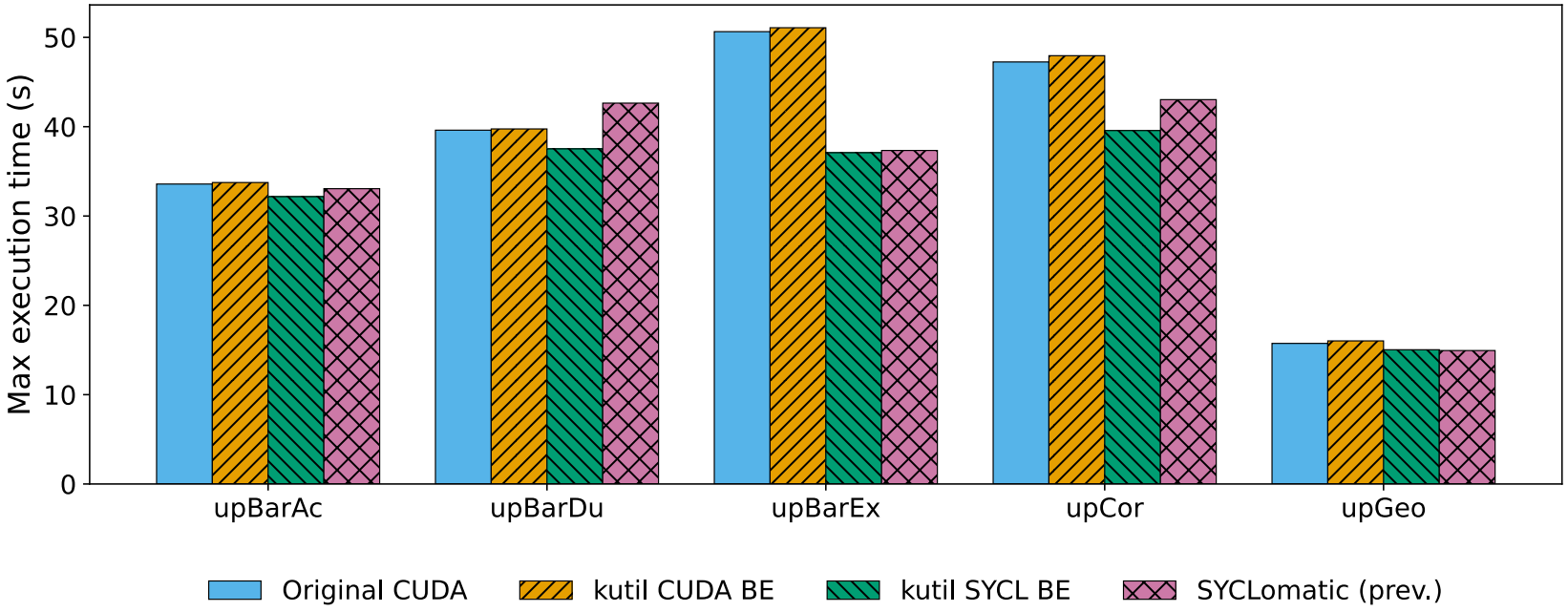
```
q.parallel_for(nd_range, [=](sycl::nd_item<1>
    item {
  updateGeometry(item)(leafs1, leafs2, xx);
});
```

CASE STUDY: CRK-HACC KERNEL SUITE

- **Five dominant kernels**
  - Geometry (upGeo)
  - Corrections (upCor)
  - Extras (upBarEx)
  - Acceleration (upBarAc)
  - Energy (upBarDu)
- Together they cover >85% of solver time

- **Code Similarity Results between native unmodified CUDA and using kutil library:**
  - Line/token Jaccard ≈ 0.87–0.93
  - Diff similarity ≈ 0.96–0.98

PERFORMANCE RESULTS: POLARIS (NVIDIA)

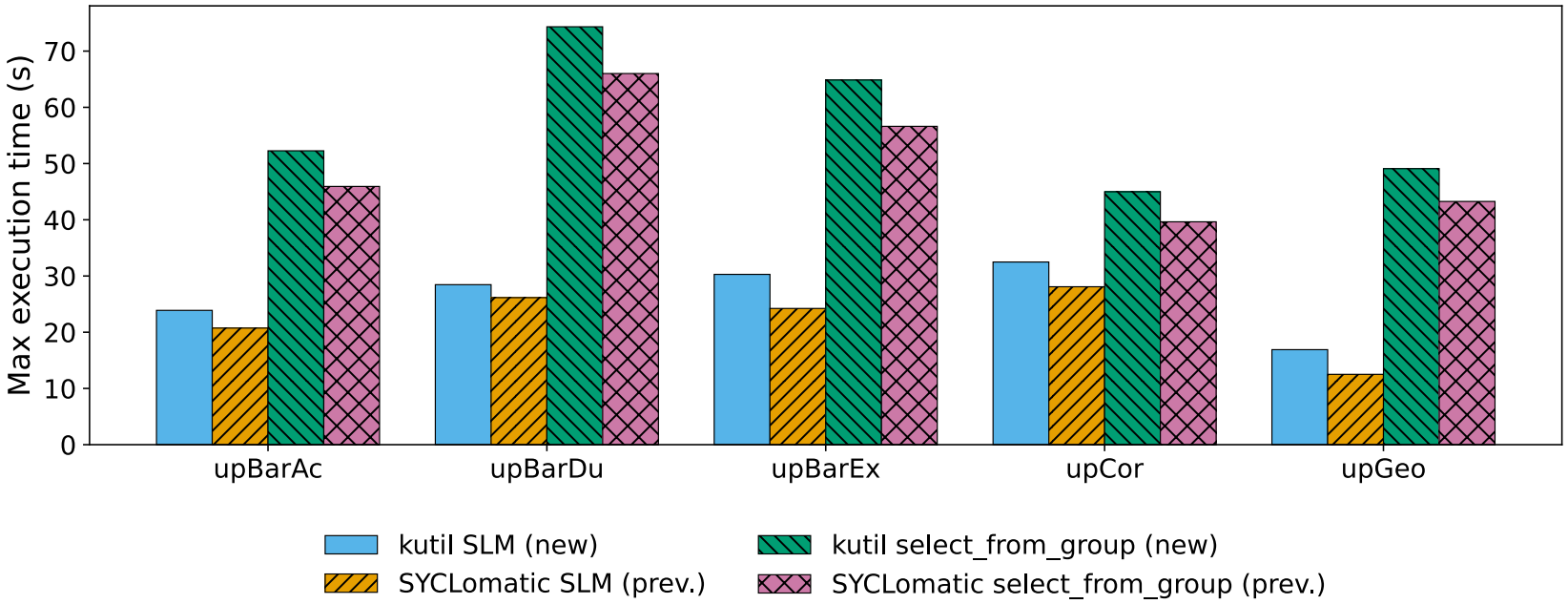


PERFORMANCE RESULTS: POLARIS (NVIDIA)

- **CUDA Backend Results**
  - Functorized CUDA shows **negligible overhead** (<1–2%).
  - Confirms the abstraction “compiles away” on NVIDIA.
- **SYCL Backend Results on NVIDIA**
  - kutil SYCL backend matches or slightly exceeds SYCLomatic + tuning implementation.
  - Observed competitive performance with CUDA.

- **RDC Study**
  - We observed RDC can increase runtime **1.1× to 2.5×**.
  - Our design intentionally avoids RDC.
  - Demonstrates the importance of TU-local instantiation.

PERFORMANCE RESULTS: AURORA (INTEL)



PERFORMANCE RESULTS: AURORA (INTEL)

- Shared-local-memory (SLM) shuffles outperform subgroup shuffles:
  - — **2×–3× faster**, consistent with prior work.
- kutil kernels show **10–15% slowdowns** relative to earlier SYCLomatic results.

- **Interpreting the Slowdown**
  - — Possible abstraction effects (e.g., inlining behavior).
  - — Almost certainly combined with SDK/runtime differences:
    - • compiler versions
    - • driver/firmware changes
  - — Not attributable solely to the abstraction; full apples-to-apples impossible due to environment drift

PRODUCTIVITY & MAINTAINABILITY IMPACT

- **For HACC developers**
  - Zero kernel rewrites.
  - SYCL compatibility hidden behind headers.
  - No dual-backend divergence.
  - No SYCLomatic churn for each release.

- **For the broader HPC community**
  - Demonstrates a *middle path* between:
  - full framework adoption (RAJA/Kokkos), and
  - full source migration to SYCL or HIP.
- Particularly suited for:
  - legacy codes,
  - codes with active CUDA development,
  - teams prioritizing single-source unification.

CONCLUSION

We show that you can keep your CUDA kernel body and still run on SYCL, without migration tools or DSL rewrites.

- Key outcomes:
  - — Unified single-source code base for CRK-HACC.
  - — Minimal abstraction overhead.
  - — Competitive performance across NVIDIA and Intel GPUs.
  - — Tiny abstraction footprint, large maintainability benefits.
- Future work will explore application beyond CRK-HACC.


Argonne
NATIONAL LABORATORY