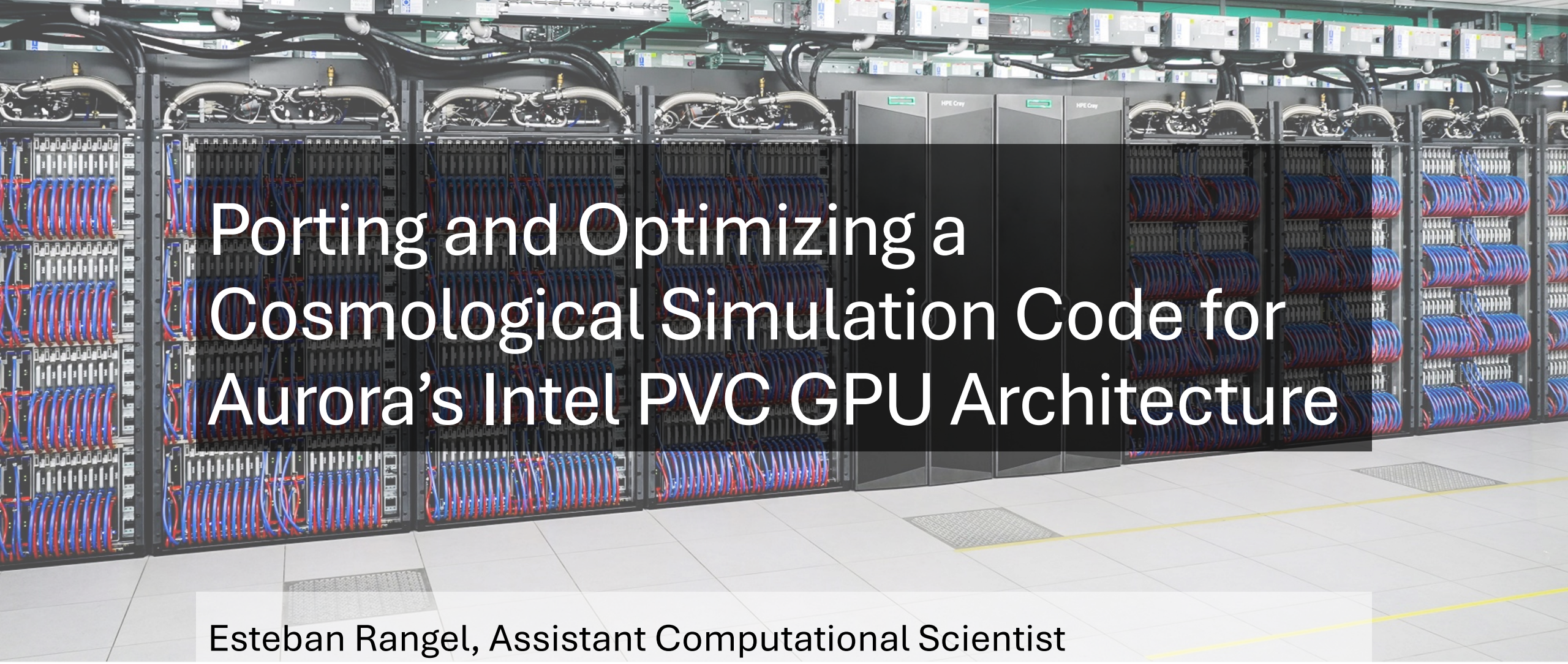


Porting and Optimizing a Cosmological Simulation Code for Aurora's Intel PVC GPU Architecture

Esteban Rangel, Assistant Computational Scientist
Computational Science (CPS) Division, Argonne National Laboratory

AMReX workshop

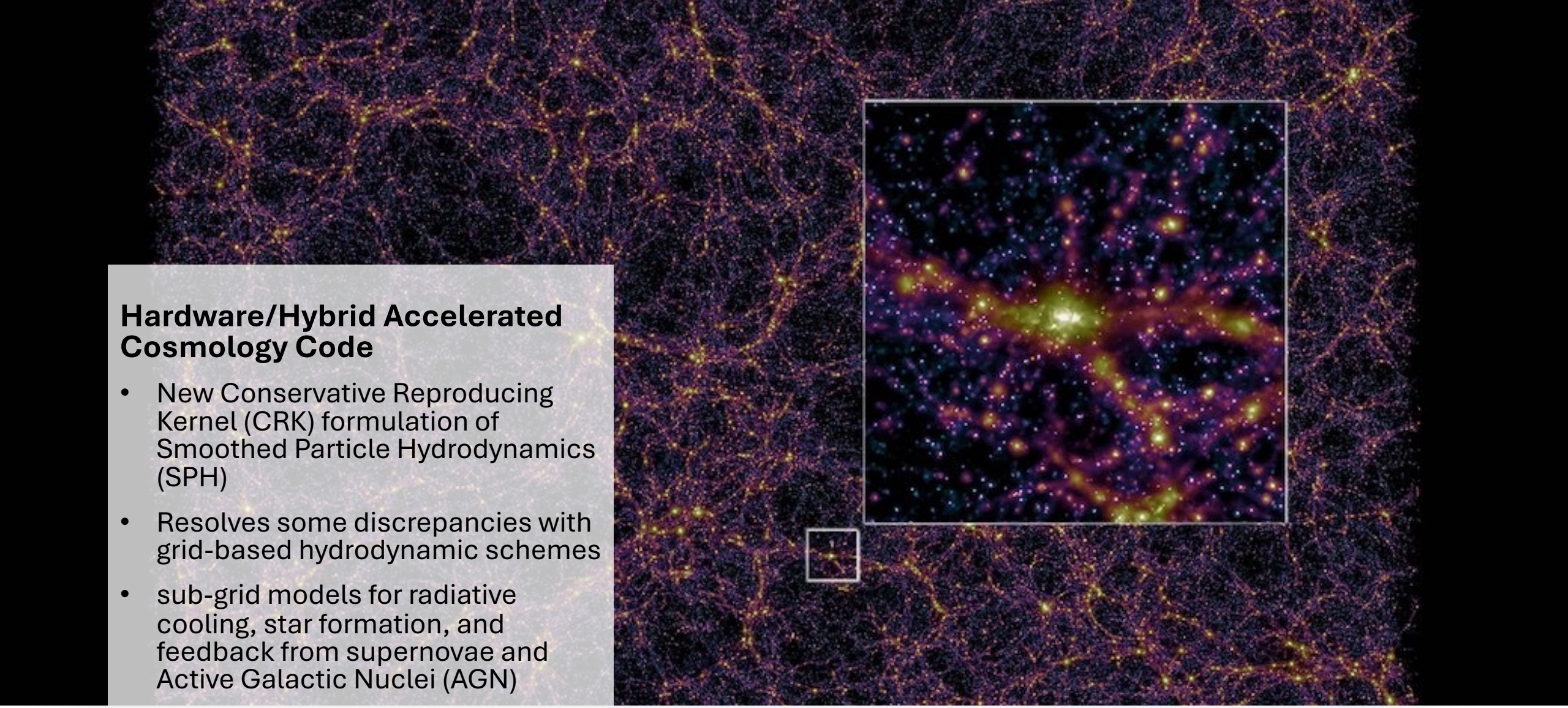
Goals for this talk

1. Demonstrate that SYCL can be used to achieve performance portability, using CRK-HACC as a case study.

2. Give some insight into the low-level programming capabilities of oneAPI using SYCL and inline vISA to target the Intel® Data Center GPU Max 1550 (PVC) .

Outline

1. Porting HACC from CUDA to SYCL
    1. Migration pipeline using SYCLomatic + LibTooling
    2. Abstracting host-side API calls to simplify maintainability
    3. Portability comparison with HIP, CUDA, and SYCL on AMD, NVDIA, and Intel GPUs
2. Optimizations for Intel® Data Center GPU Max 1550
    1. Shuffles using shared-local memory
    2. HACC-specific shuffles using inline vISA in SYCL
3. Early results from pre-production Aurora

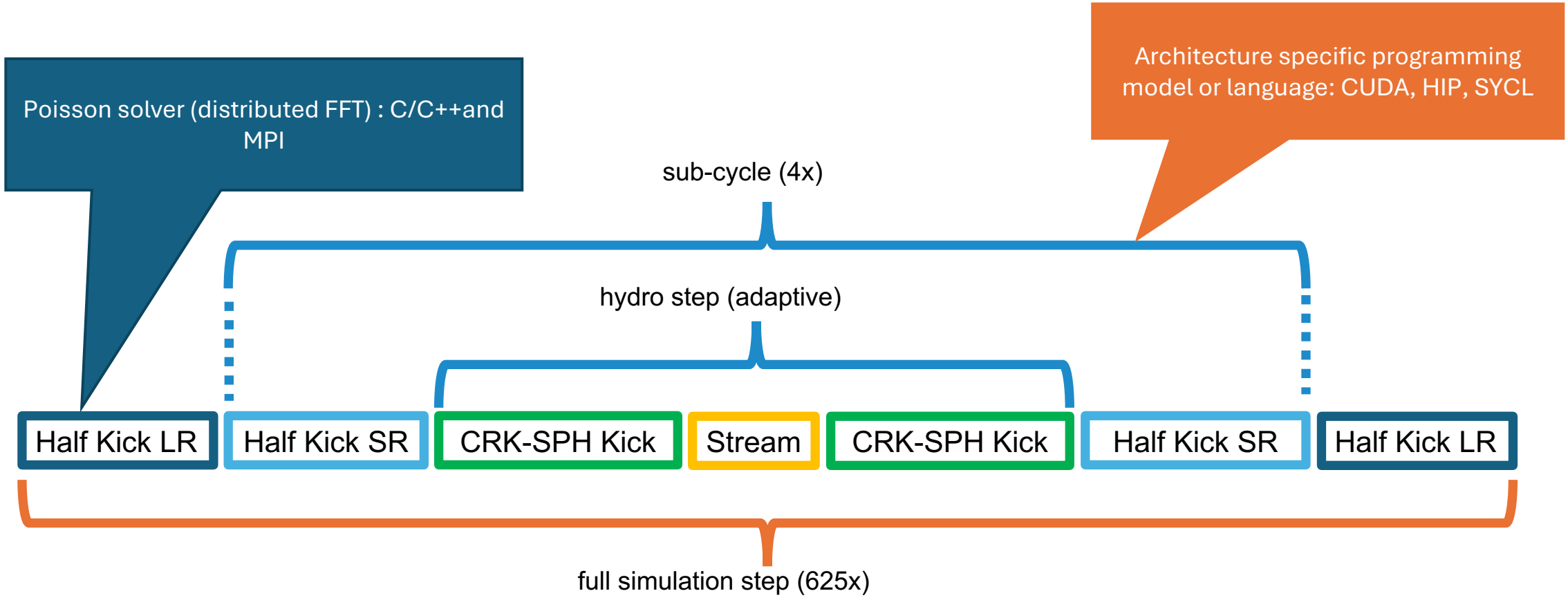


Hardware/Hybrid Accelerated Cosmology Code

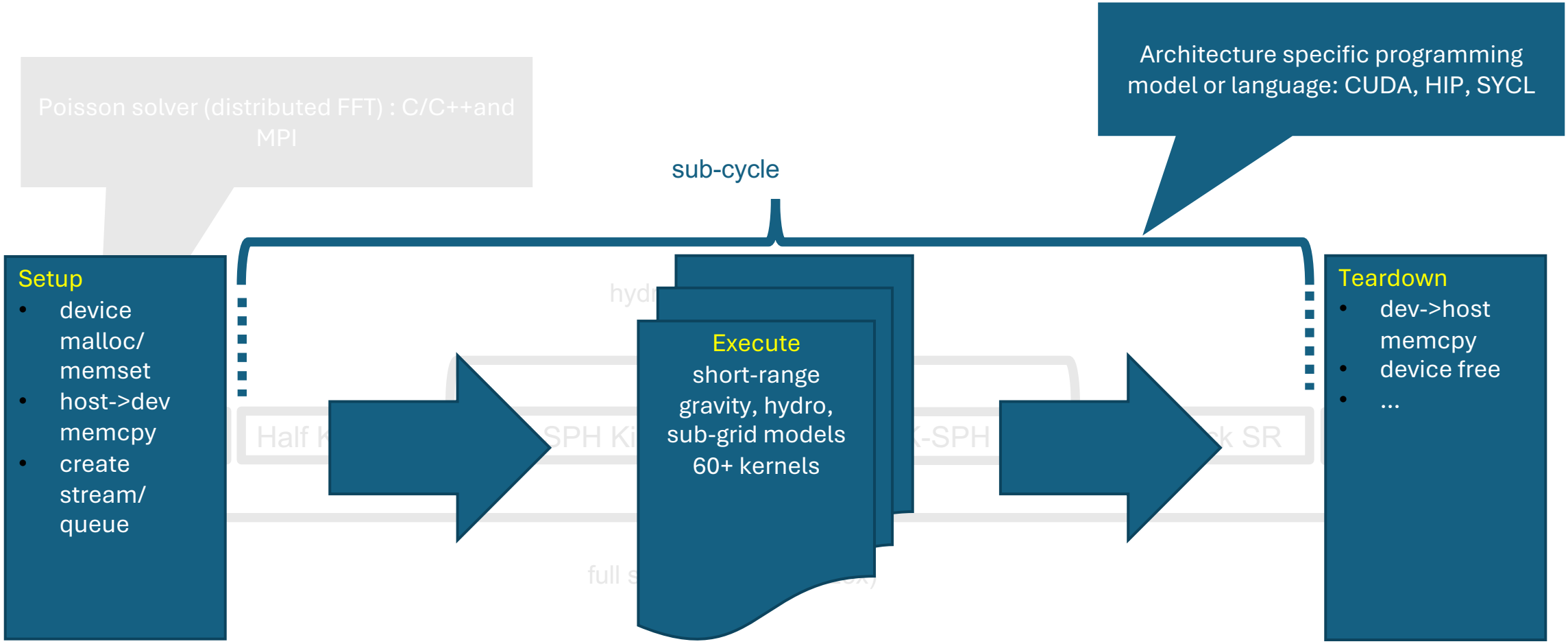
- New Conservative Reproducing Kernel (CRK) formulation of Smoothed Particle Hydrodynamics (SPH)
- Resolves some discrepancies with grid-based hydrodynamic schemes
- sub-grid models for radiative cooling, star formation, and feedback from supernovae and Active Galactic Nuclei (AGN)

CRK-HACC: N-body Cosmological Simulation

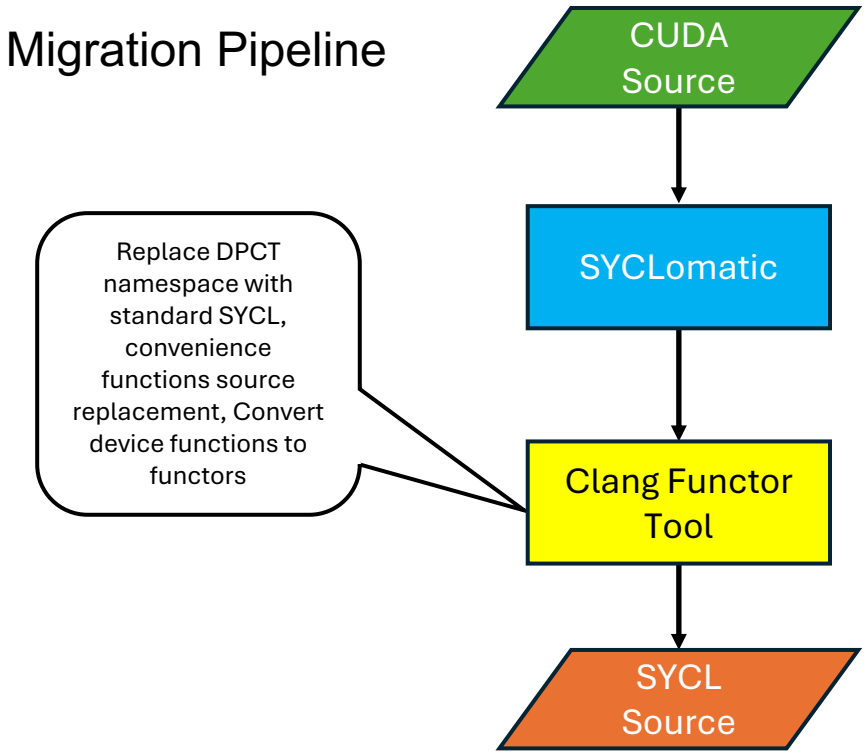
HACC's Execution (overview)

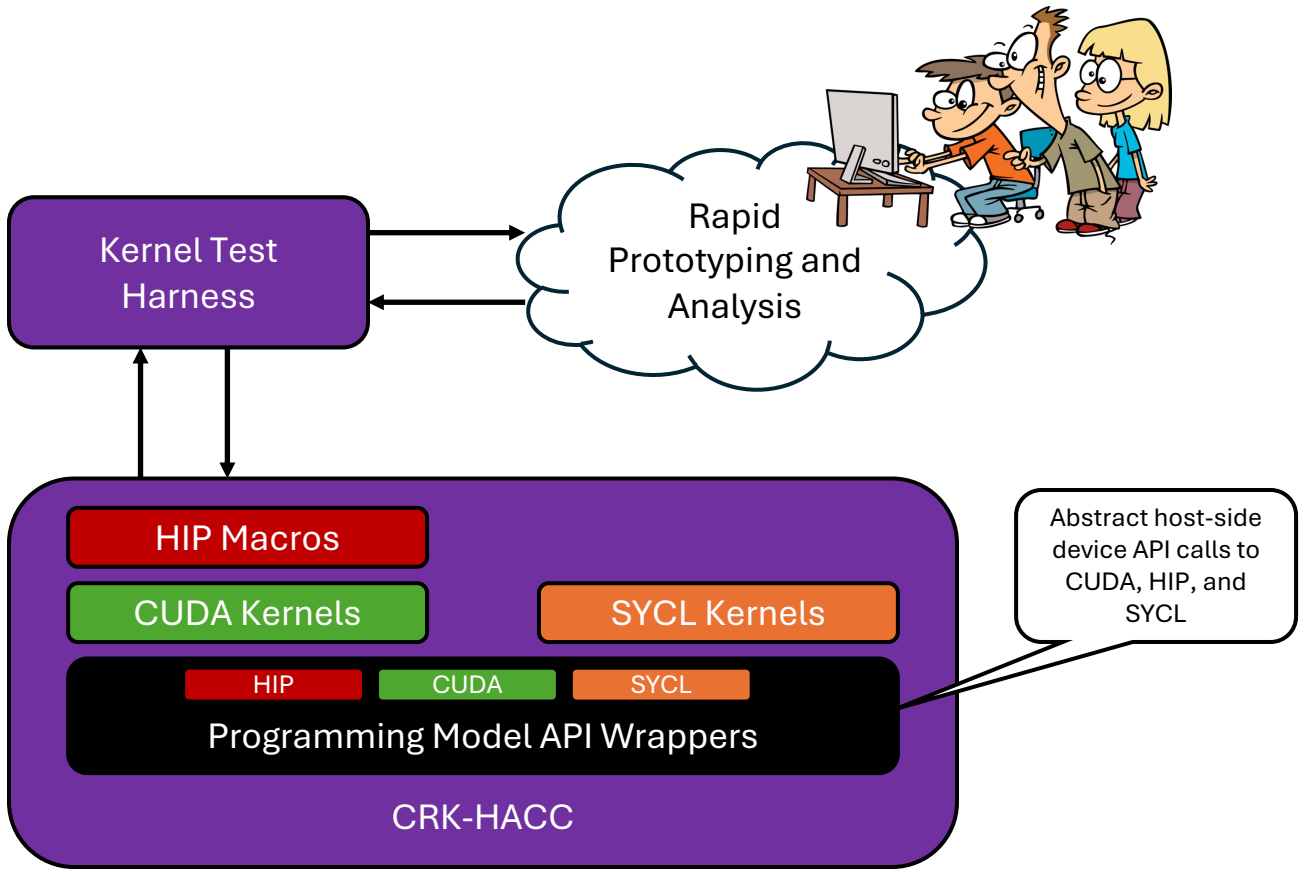


HACC's Execution (overview)



Porting HACC from CUDA to SYCL





CUDA/HIP/SYCL Kernel Launch Forms

CUDA/HIP

```
// CUDA kernel defined as a
function invoked by <<<>>>
__global__ void
cuda_kernel(float* a, int b) {
  /* kernel body */
}
...
cuda_kernel <<<...>>>(a, b);
```

SYCL lambda

```
// SYCL kernel defined as a
function invoked by a kernel
lambda
void sycl_kernel(float* a, int
b, sycl::nd_item<3> it) {
  /* kernel body */
}
...
q.parallel_for(sycl::nd_range
<3>(...),[=](sycl::nd_item <3>
it) {
  sycl_kernel(a, b, item_ct1);
});
```

SYCL function object

```
// SYCL kernel defined as a
function object invoked
directly
struct SYCLKernel
{
const float* a;
const int b;
void
operator()(sycl::nd_item<3> it)
{
  /* kernel body */
}
};
...
q.parallel_for(sycl::nd_range
<3>(...), SYCLKernel(a, b));
```

The SYCL 2020 standard does not allow function pointers, however, a C++ class from a function object can be used as a template parameter and is how our kernel launch wrapper is defined. For CUDA, our wrapper is implemented as a macro.

Programming Model API Wrappers (host)

SYCL Kernel Launch

```
template<class K, typename... T>

void InvokeSYCLKernel(int nBlock, int BlockSize,
sycl::queue sycl_queue, T ...args){

sycl::event e =
sycl_queue.submit([&](sycl::handler &cgh) {

sycl::local_accessor<char>
slm(sycl::range(SLM_BYTES), cgh);

K sycl_kernel = K(slm, args...);

cgh.parallel_for(sycl::nd_range<1>(nBlock*BlockSiz
e, BlockSize), sycl_kernel);

});

e.wait();

}
```

SYCL Memory (USM)

```
//GPU Memset

#define InvokeGPUMemset(devPtr,value,count,stream)
{ \

stream.memset(devPtr, value, count);          \

stream.wait();                                \

}

//GPU Asynchronous memset

#define InvokeGPUMemsetAsync InvokeGPUMemset

//Synchronize device

#define InvokeGPUSynchronize(stream) {  \

  stream.wait();                        \

}
```

Experimental Setup

System	CPU	GPU	FP32 (theoretical) Peak per GPU
Aurora	2 x Intel® Xeon® CPU Max 9470C, 52 cores	6 x Intel® Data Center GPU Max 1550	45.9 TFLOPS
Polaris	1 x AMD EPYC 7543P, 32 cores	4x NVIDIA A100-SXM4-40GB	19.5 TFLOPS
Frontier	1 x AMD EPYC 7A53, 64 cores	4 x AMD Instinct MI250X	53 TFLOPS

Simulation

- 2x 512^3 particles
- 5 timesteps (4 fixed sub-cycles)
- 8 MPI ranks

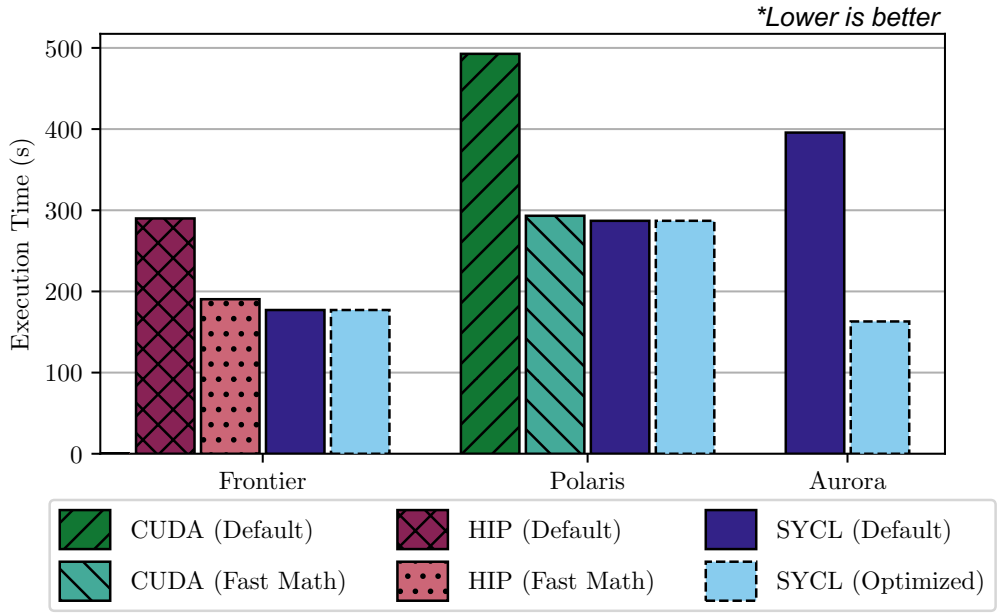
System Configurations (1-node)

- Aurora: 1 rank/stack
- Polaris: 2 ranks/GPU
  note: measured ~11% lower efficiency
- Frontier: 1 rank/GCD

Run Dates:
Aurora: September 15, 2023 : SYCL Compiler: Intel® oneAPI DPC++/C++ Compiler 2023.2.0 (2023.x.0.20230510)
Polaris: August 3, 2023 : CUDA Compiler: cuda_11.8.r11.8/compiler.31833905_0
Frontier: September 25, 2023 : HIP Compiler: roc-5.3.0 22362 3cf23f77f8208174a2ee7c616f4be23674d7b081

Initial Cross-GPU Comparison Results

Aggregate of all GPU Kernels



Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See slide 6 for configuration details.

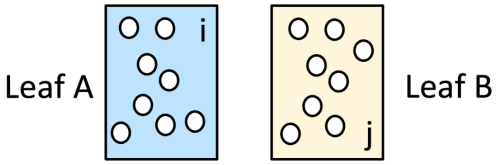
- The *fast-math* compiler option was not enabled by default with nvcc or hipcc but is with the DPC++ compiler.
- On Frontier, the HIP code uses wavefront size 64 and the SYCL code uses sub-group size 64.
- On Polaris, the CUDA code uses warp size 32 and the SYCL code uses sub-group size 32.
- On Aurora, the SYCL code uses sub-group size 32 or 16, whichever is optimal for the implementation.

Optimizations

Hotspot Kernels

1. *Geometry*: measures the volumes of gas particles
2. *Corrections*: computes the reproducing kernel coefficients of the higher order smoothed particle hydrodynamics (SPH) solver
3. *Extras*: evaluates the density and state gradients
4. *Acceleration*: calculates the momentum derivative
5. *Energy*: solves the derivative of the internal energy.

The SIMD lane data layout of the “half-warp” algorithm, implemented in the hotspot kernels.



Particles are organized in a tree with “fat leaves” containing an SYCL work-group/CUDA block-sized number of particles.

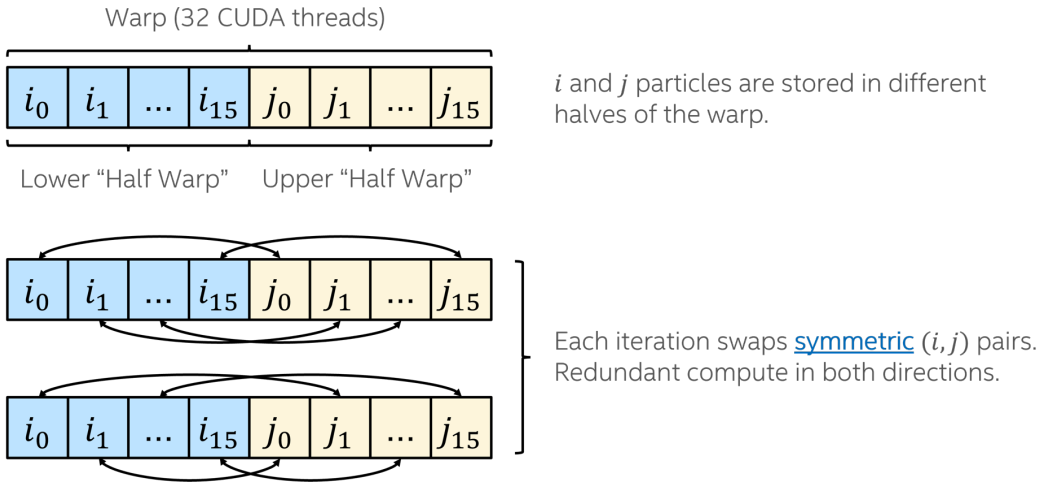
Particle SIMD layout

Lane ID	0	1	…	15	16	17	…	31
Leaf	A	A	A	A	B	B	B	B

Lanes [0-15] load and update particles from leaf A, while lanes [16-31] operate on particles from leaf B.

Optimizing Lane Data Exchanges

The data exchange pattern of the “half-warp” algorithm for interacting particles from leaves A and B within the same warp/sub-group.



The figure represents one of the total ($|LeafA| \times |LeafB|/warp\_size$) instances required.

- Notice the *pair-wise symmetry,* which is critically important for the correctness of the algorithm.
- XOR-based shuffle pattern implemented as:
  - `__shfl` intrinsic with CUDA
  - `sycl::select_from_group` with SYCL

Optimizing Lane Data Exchanges

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
...
```

Communication 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 using vISA[1]
  - Explicitly code the assembly instructions for each communication step needed.

1 Rangel et al., A Performance-Portable SYCL Implementation of CRK-HACC for Exascale

Optimizing Lane Data Exchanges (SLM)

Single element SLM shuffle

```
template <typename T>
inline T sub_group_slm_exchange_single(sycl::sub_group sg,
T value, int src) const {
char* slm_ptr = &slm[0];
const int sg_size = sg.get_max_local_range()[0];
const int offset = sg.get_group_id() * sg_size;
const int simd_lane = sg.get_local_id();
sycl::group_barrier(sg,sycl::memory_scope::sub_group);
((T*)slm_ptr)[offset + simd_lane] = value;
sycl::group_barrier(sg,sycl::memory_scope::sub_group);
return ((T*)slm_ptr)[offset + src];
}
```

Object SLM shuffle

```
template <typename T>
inline T sub_group_slm_exchange_obj(const sycl::sub_group
sg, const T value, const uint32_t src)
const {
const uint32_t simd_lane = sg.get_local_id();
char* slm_ptr = &slm[sg.get_group_id() * slm.size() /
sg.get_group_range()[0]];
sycl::group_barrier(sg,sycl::memory_scope::sub_group);
((T*)slm_ptr)[simd_lane] = value;
sycl::group_barrier(sg,sycl::memory_scope::sub_group);
return ((T*)slm_ptr)[src];
}
```

Optimizing Lane Data Exchanges (inline vISA)

Specialized HACC-shuffle exchange pattern, which provides the same pair-wise symmetry property of the XOR-based pattern (subset).

0	1	…	15	16	17	…	31
x_{16}	x_{17}	$x_{…}$	x_{31}	x_{0}	x_{1}	$x_{…}$	x_{15}

i = 0

After an initial upper- and lower-lane data exchange,

0	1	…	15	16	17	…	31
x_{31}	x_{16}	$x_{…}$	x_{30}	x_{1}	x_{2}	$x_{…}$	x_{0}

i = 1

lower lanes perform a cyclic shift-right(i) and upper lanes perform a cyclic shift-left(i)

…

0	1	…	15	16	17	…	31
x_{17}	x_{18}	$x_{…}$	x_{16}	x_{15}	x_{0}	$x_{…}$	x_{14}

i = 15

Optimizing Lane Data Exchanges (inline vISA)

```
#define HACC_SHFT(SRC_R, SRC_L) {                                   \
__asm__ (                                                           \
"{\n"                                                               \
".decl TMP0 v_type=G type=f num_elts=32 align=GRF\n"                \
".decl TMP1 v_type=G type=f num_elts=32 align=GRF\n"                \
"mov (M1_NM, 32) TMP0(0,0)<1> %1(0,0)<0;16,1>\n"                    \
"mov (M1_NM, 32) TMP1(0,0)<1> %1(0,16)<0;16,1>\n"                   \
"mov (M1, 16) %0(0,16)<1> TMP0(0,"#SRC_R")<1;1,0>\n"                \
"mov (M1, 16) %0(0,0)<1> TMP1(0,"#SRC_L")<1;1,0>\n"                 \
"}\n"                                                               \
: "=rw"(return_value)                                               \
: "rw"(value));                                                     \
}
#endif
```

Register view of the specialized HACC-shuffle

r0	X_0	X_1	X_2	X_3	X_4	...	X_{14}	X_{15}
r1	X_0	X_1	X_2	X_3	X_4	...	X_{14}	X_{15}
r2	X_{16}	X_{17}	X_{18}	X_{19}	X_{20}	...	X_{30}	X_{31}
r3	X_{16}	X_{17}	X_{18}	X_{19}	X_{20}	...	X_{30}	X_{31}

Example of performing the i=1 instance of the butterfly-shuffle.

The “wrap-around” feature of the register file is exploited.

Optimizing Lane Data Exchanges (inline vISA)

vISA Macro Caller Function

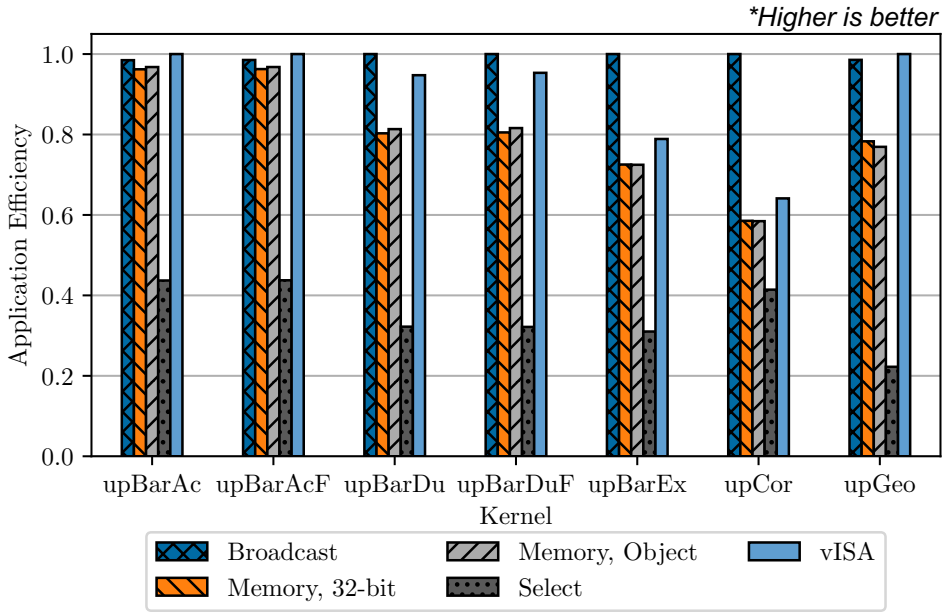
```
// HACC shuffle functions for PVC
inline float HACC_shuffle(sycl::sub_group sg, const float
value, const int shift) const
{
float return_value = 0;
#if defined(__SYCL_DEVICE_ONLY__)
switch (shift) {
#if HACC_SYCL_SG_SIZE==16
case 0:
HACC_SHFT(0,8);
break;
case 1:
HACC_SHFT(1,7);
break;
...
```

Kernel code

```
#ifdef HACC_SYCL_INLINE_VISA
// Compile-time loop with "constant expression" index
repeat<HALF_WARP>([&] (auto index) {
#else
#pragma unroll (HALF_WARP)
for (int i = 0; i < HALF_WARP; i++) {
#endif
…
#ifdef HACC_SYCL_INLINE_VISA
constexpr int src = index;
#else
int src = laneID ^ (HALF_WARP + i);
#endif
…
float rhoj = HACC_shuffle(item_ct1.get_sub_group(), rhoi, src);
```

Optimization Results

Aurora (PVC)



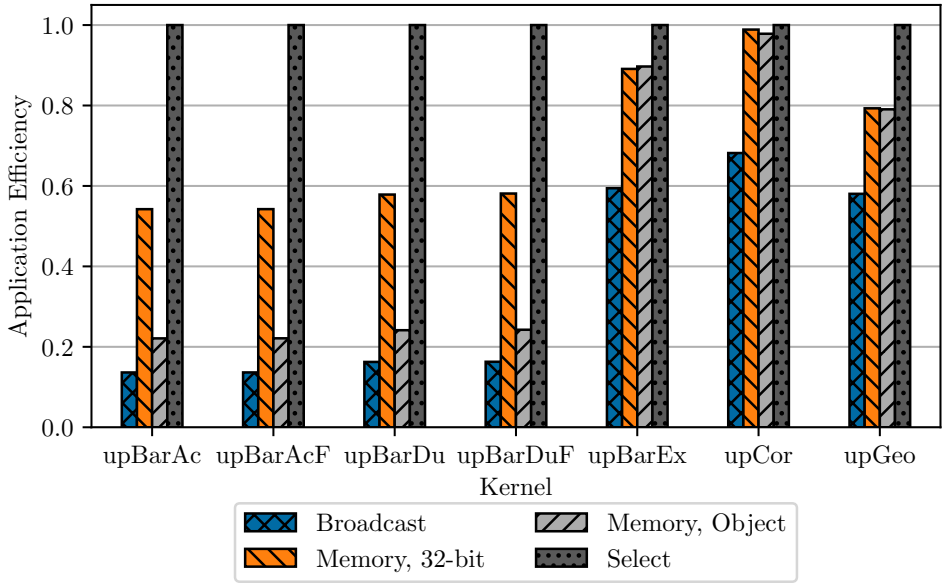
- Broadcast uses a sub-group size of 16, all other variants use a sub-group size of 32.
- Restructuring the loops to use broadcasts requires *fewer* atomic instructions, more noticeable in the Extras and Corrections kernels.
- Performance evaluation on NVIDIA and AMD architectures was also performed[1].

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See slide 6 for configuration details.

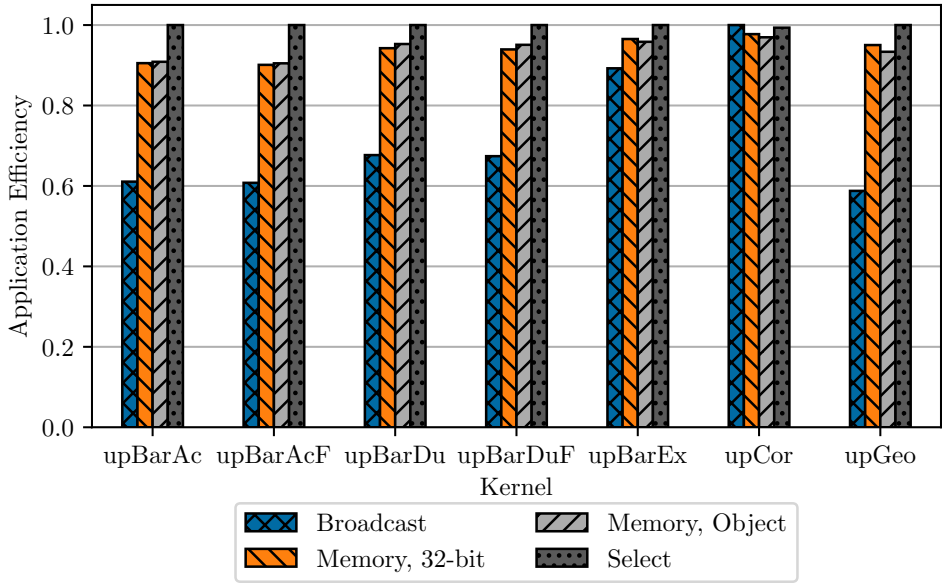
1 Rangel et al., A Performance-Portable SYCL Implementation of CRK-HACC for Exascale

Optimization Results

Polaris (A100)



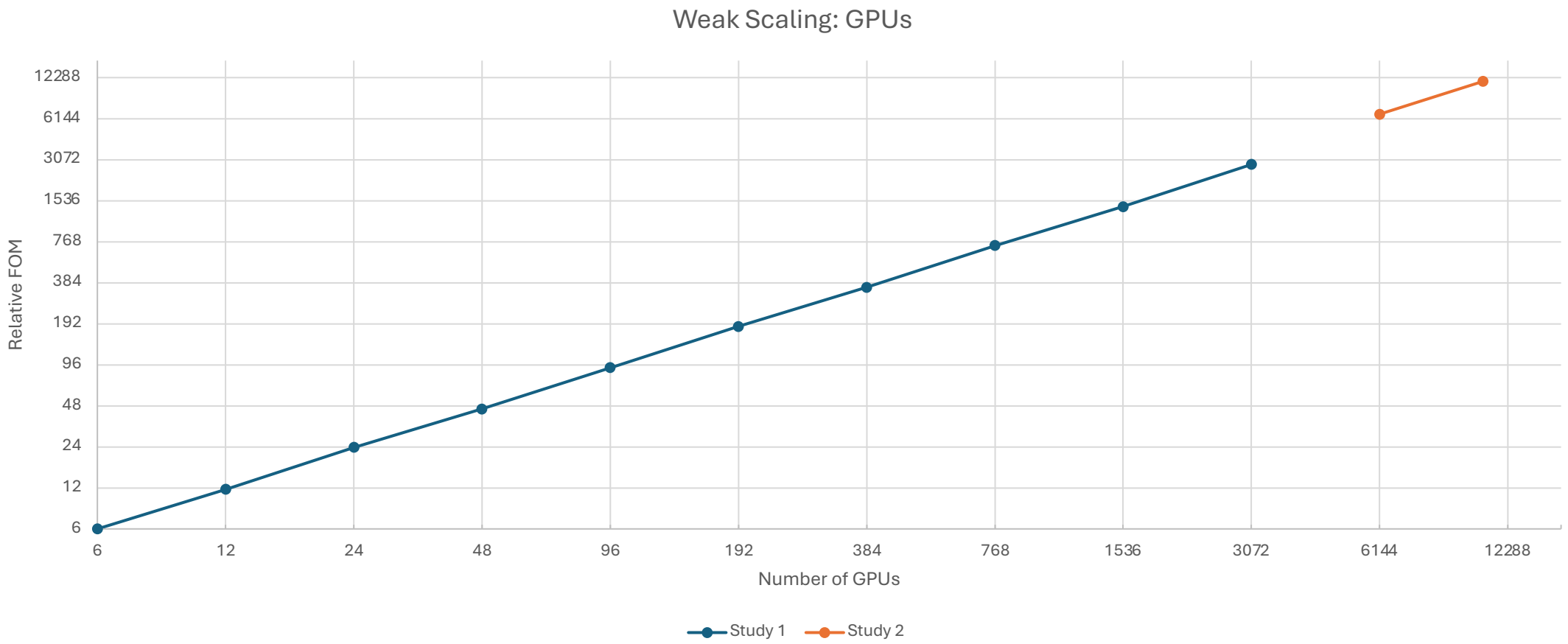
Frontier (MI250x)



Weak Scaling on Aurora

Figure-of-merit (FOM)
steps * sub-steps * number-of-particles / GPU time / 10^6

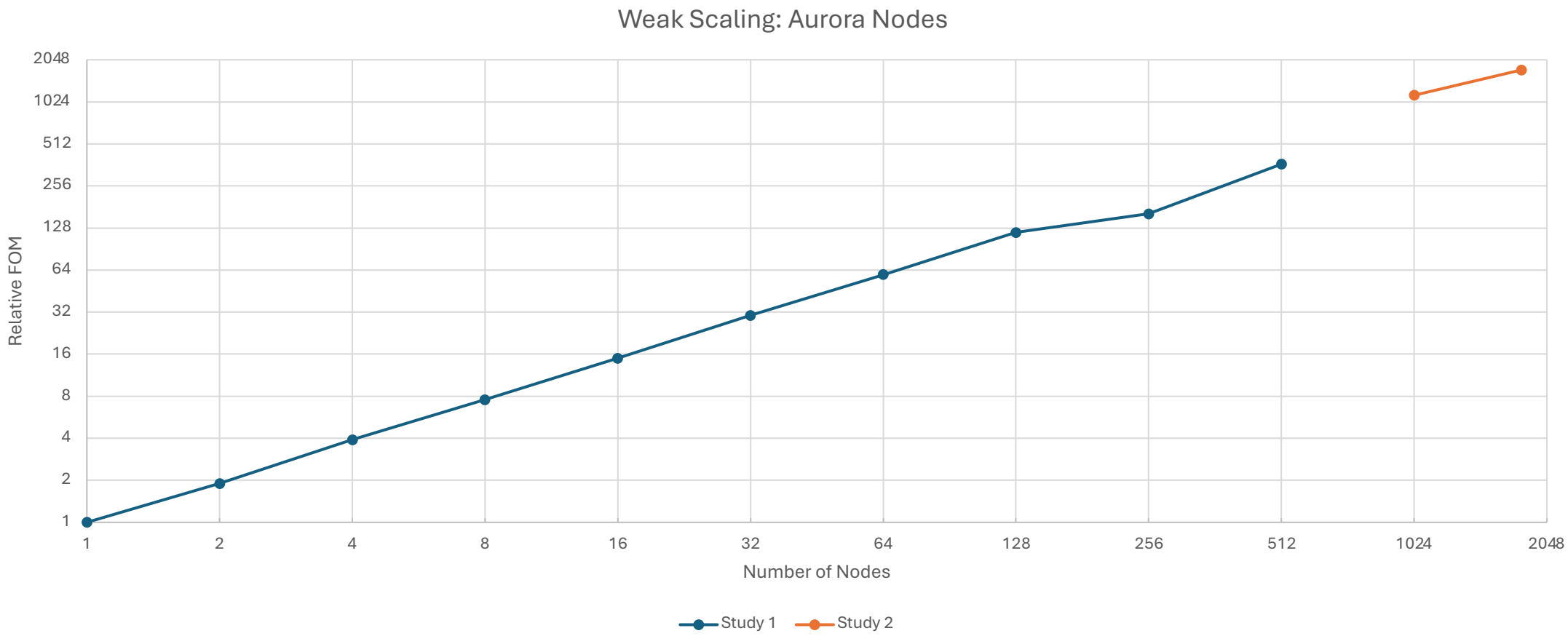
Disclaimer: Results were gathered on a pre-production state of Aurora with engineering versions of the SDK and system software. The two studies have different software versions and application configurations and are only intended to display a trend.



Weak Scaling on Aurora

Figure-of-merit (FOM)
steps * sub-steps * number-of-particles / total wall time / 10^6

Disclaimer: Results were gathered on a pre-production state of Aurora with engineering versions of the SDK and system software. The two studies have different software versions and application configurations and are only intended to display a trend.

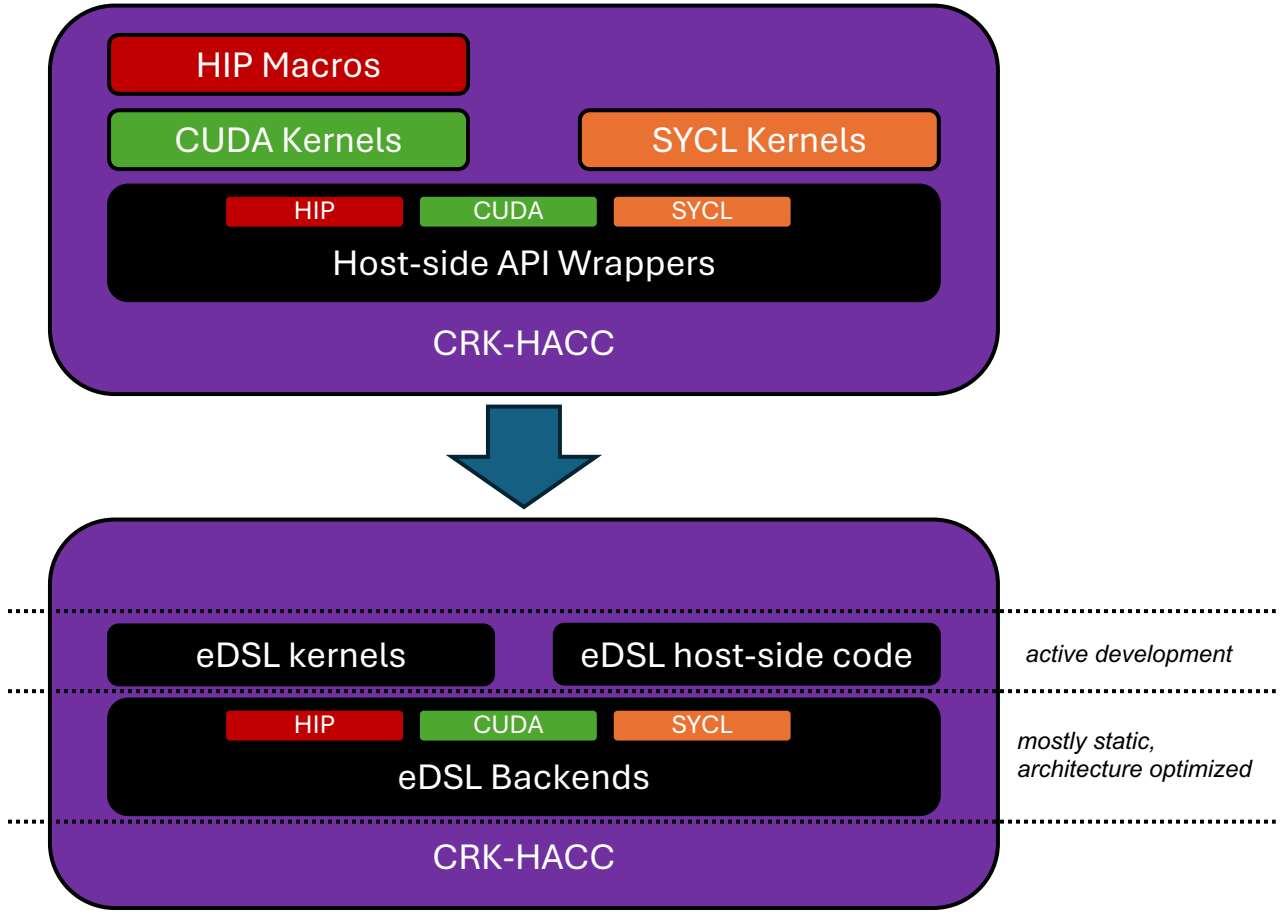


eDSL for Parallel Programming Model APIs

CRK-HACC is under active development, and since November 2023, the repository has had **377** files changed, **92,862** insertions(+), and **3,410** deletions(-) in **77** commits.

Goal: Design an embedded domain-specific language (eDSL) using C++ for HACC's supported programming models to have a single-source representation of kernels and host-side code.

1. Define the host-side abstractions, e.g., memory management, kernel invocation, data transfer, etc.
2. Define the kernel functionality abstractions, e.g., warp/sub-group functions, group functions, special math functions, etc.
3. Design the syntax for the eDSL.
4. Write the C++ wrappers that interface with the CUDA and SYCL APIs.



HACC eDSL Kernel Launcher

```
template<typename K>
INLINE_FUNCTION_ATTR void kernel_launch(K kernel, kernel_launch_params params)
{
#ifdef __NVCC__
cudaStream_t stream = params.dispatcher;
cuda_kernel_launch<<<params.num_blocks, params.block_size, params.local_memory_bytes, stream>>>(kernel);
cudaStreamSynchronize(stream);
#endif
#ifdef SYCL_LANGUAGE_VERSION
...
}
```

```
#ifdef __NVCC__
template<typename K>
__global__ void cuda_kernel_launch(K kernel)
{
hacc::item<1> it = hacc::item<1>();
kernel(it);
#endif
}
```

```
// example kernel invocation (host-side)
hacc::kernel_launch(\
  updatePositions{m_devPosBuff[buff_indx], m_devVelBuff[buff_indx], prefactor},
  hacc::kernel_launch_params{posBlocksPerGrid, m_groupSize, 0, devManager}
);
```

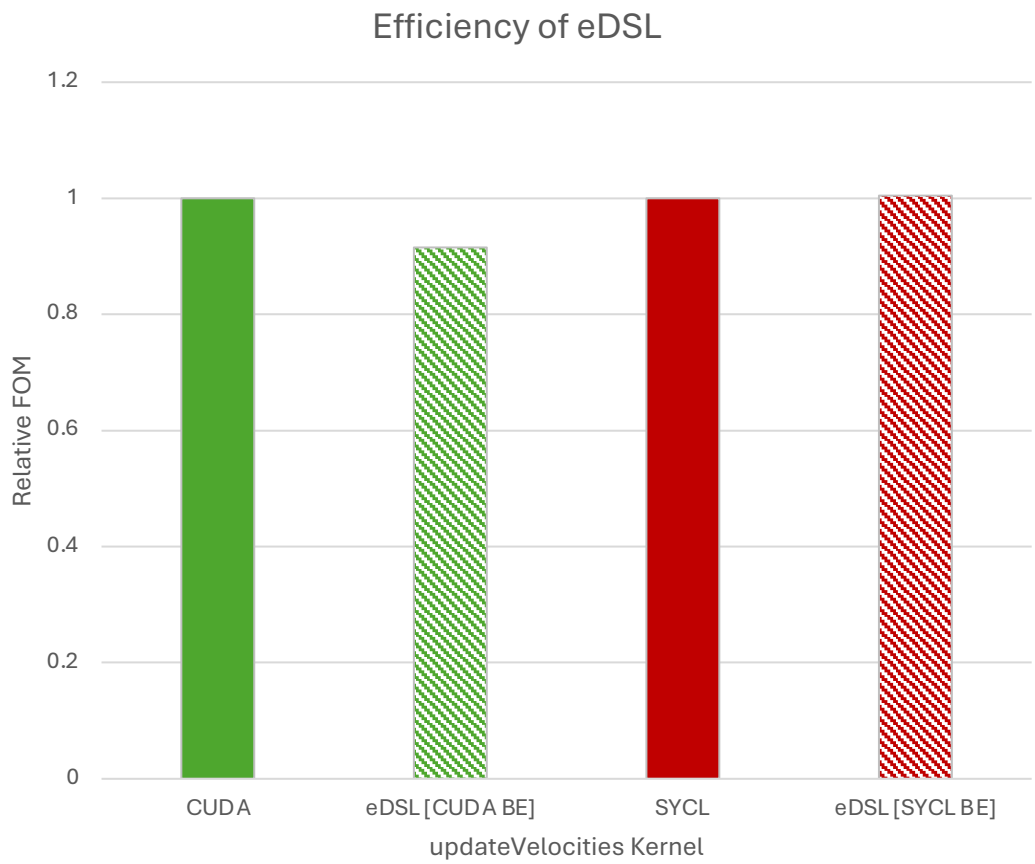
HACC eDSL Kernel Example

```
#include "updatePositions.h"
#include "eDSL_common.h"
#include "eDSL_kernel_launch.h"
#include "eDSL_math.h"
#include "eDSL_utility.h"

template void hacc::kernel_launch<updatePositions>(updatePositions,
kernel_launch_params);

KERNEL_FUNCTION_ATTR void updatePositions::operator () (hacc::item<1>
item) const
{
  /* kernel body */
}
```

The use of explicit instantiation of the launch wrapper in the kernel definition source file avoids the need to build CUDA relocatable device code.

HACC eDSL Results

- HACC gravity-only simulation
  - 512^3 particles
  - 8 MPI ranks
- Test Systems
  - ANL JLSE, (4x) NVIDIA V100
    - Evaluate CUDA and eDSL with CUDA back-end
  - Sunspot, Intel (4x) Intel® Data Center GPU Max 1550
    - Evaluate SYCL and eDSL with SYCL back-end



Conclusion

- Described porting HACC from a CUDA codebase to SYCL.
- The “shuffle” operations used by CRK-HACC are not performance-portable on the Intel® Data Center GPU Max 1550.
- A straightforward workaround for *shuffles* using shared-local memory was proposed.
- Example of using inline vISA in SYCL for Intel® Data Center GPU Max 1550.
- Scaling on pre-production Aurora was demonstrated on up to 1792 nodes.
- Previewed our ongoing eDSL effort to support code maintainability.

HACC Team

Argonne NATIONAL LABORATORY


Salman Habib


Nicholas Frontiere

Katrin Heitmann



Vitali Morozov


JD Emberson


Thomas Uram


Patricia Larsen


Adrian Pope


Esteban Rangel

Michael Buehlmann


Disclaimers

- Performance varies by use, configuration and other factors. Learn more at https://www.intel.com/performanceindex
- Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See Slide 6 for configuration details. No product or component can be absolutely secure. Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.
- Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others. Khronos is a registered trademark and SYCL and SPIR are trademarks of The Khronos Group Inc.
- No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document, with the sole exception that code included in this document is licensed subject to the Zero-Clause BSD open source license (OBSD), http://opensource.org/licenses/0BSD.

Acknowledgments

- This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.
- 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.
- This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.