Preparing HACC for Aurora

Esteban Rangel, Computational Science (CPS) Division

ALCF Developer Session

Argonne NATIONAL LABORATORY

Outline

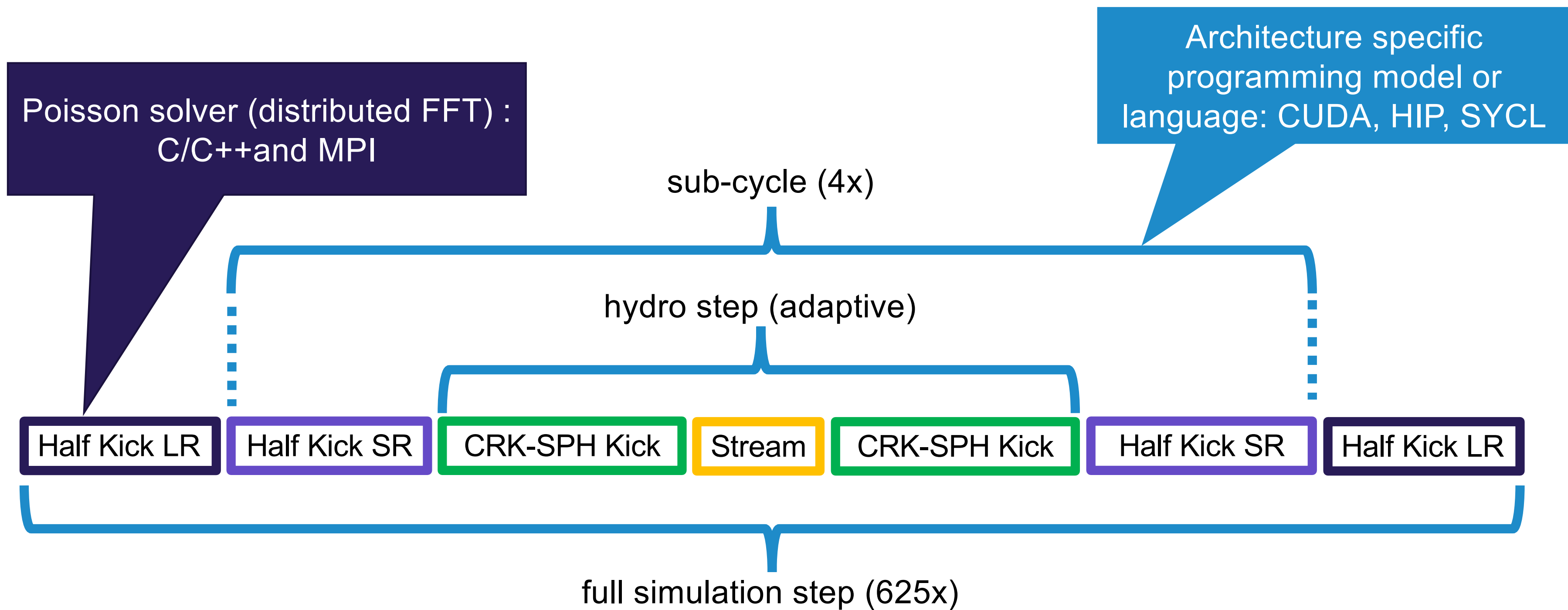
i. Preparing HACC for Exascale
ii. Development
  - Supporting multiple programming models for GPUs
  - Migration from CUDA to SYCL
  - Optimizations for Intel Xe GPUs
iii. Performance results on testbed systems
iv. Visualization (Movie) : cosmological adiabatic simulation used for verification

Argonne NATIONAL LABORATORY

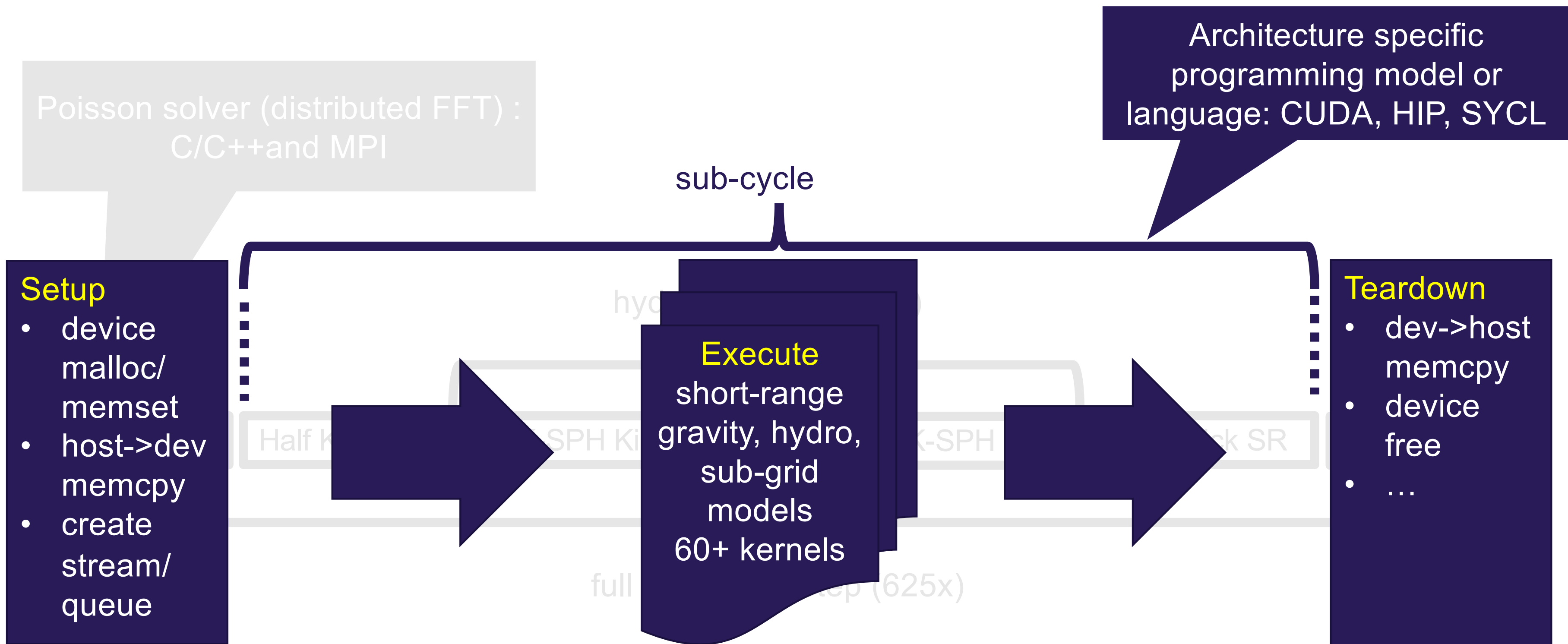
Preparing for Exascale

- Projects
  - ECP, *ExaSky*
    - Running on
      - Perlmutter and Summit (CUDA)
      - Frontier and Crusher (HIP)
      - Sunspot and JLSE testbed (SYCL)
  - Aurora ESP, *Extreme-Scale Cosmological Hydrodynamics*
    - Worked closely with ALCF and Intel through COE since early stages of Aurora through hackathons and an ongoing collaboration with Intel.
    - Early work began with OpenCL gravity-only HACC
    - Porting CUDA to SYCL and optimizing performance for Xe GPU architecture
    - Run on JLSE pre-Aurora systems: Iris, DGX, Arcticus, Florentia

HACC's Codebase (overview)



HACC’s Codebase (overview)



Argonne NATIONAL LABORATORY

Challenges

1. Evaluate Aurora's early hardware and SDK on a set of complex kernels (60+) primarily written in CUDA and under active development.
2. Minimize divergence between CUDA and SYCL versions of our codebase.
3. Identify configurations and implementation optimizations specific to Intel GPUs.
4. Identify implementation optimizations that are more generally applicable.

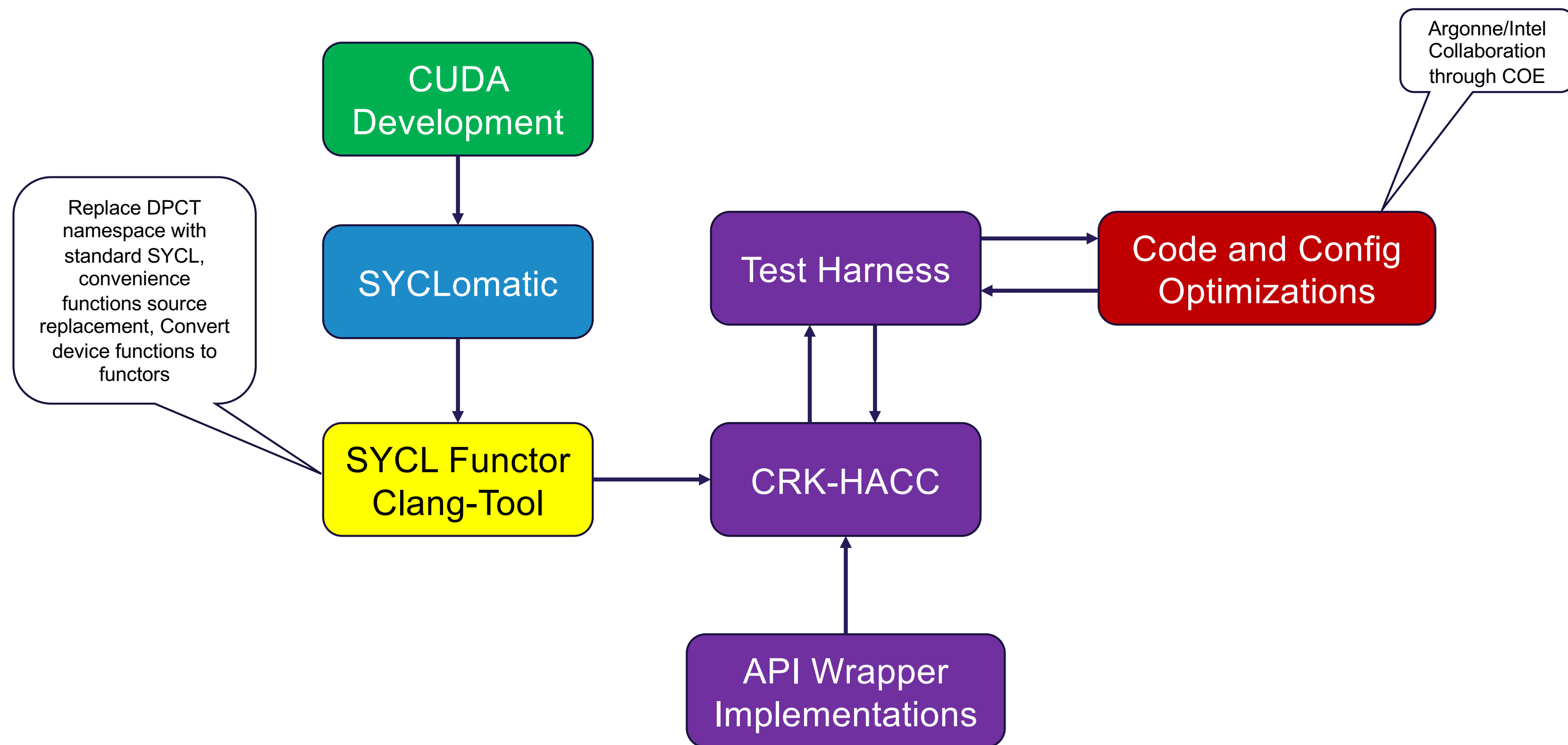
Supporting Multiple GPU Programming Models in HACC

- Host-side C++ (threaded) common for all systems.
  - Long-range spectral gravitational force solver, FFT based.
  - Ancillary routines: partitioning, rank data exchange, checkpointing, IO, etc.
  - Wrappers for specific API calls to CUDA, HIP, SYCL.
- HACC offloads large work blocks with enough concurrency to fill the GPU device with few memory transfers between host and device.
- Device kernels execute serially.

- HIP porting is largely handled with macros to redefine API calls.
- SYCL is used in a CUDA-like fashon
  - In-order SYCL queues
  - USM explicit methods, e.g., `malloc_device`, `malloc_host`, `memcpy`
- In general, there is a one-to-one correspondence between the CUDA and SYCL functionality needed by the application.

Argonne NATIONAL LABORATORY

Workflow Preparing for Aurora



Argonne NATIONAL LABORATORY

Common Host Interfaces to API Wrappers

```
// ==================================================================================================
// Update Baryon Acceleration (for 1st kick)
// ==================================================================================================
  InvokeGPUMemsetAsync((uint32_t*)DMEM->temp(),0,sizeof(float)*DMEM->maxNP*2,stream);//Zero out two temp arrays to work with

  //Note only running on nActive leafs.
  InvokeGPUKernelInter(updateBarAccel, nActive, m_groupSize, 0, stream, DMEM->leafs1(), DMEM->leafs2(),
                                               DMEM->xx(), DMEM->yy(), DMEM->zz(),
#ifdef COMPARE_HYBRID
                                               //DMEM->id(), Partition::getMyProc(),
#endif
                                               DMEM->hh(), DMEM->vol(), DMEM->mass(), DMEM->rho(),
                                               DMEM->vpx(),  DMEM->vpy(), DMEM->vpz(), DMEM->uup(),
#ifdef HY_CALC_RK0
                                               DMEM->A0(), DMEM->gA01(), DMEM->gA02(), DMEM->gA03(),
#endif
#ifdef HY_LIMITER
                                               DMEM->M1Mag(),
#endif
#ifdef HY_SPHGRADH
                 DMEM->omega(),
#endif
                                               DMEM->A(), DMEM->B1(), DMEM->B2(), DMEM->B3(),
                                               DMEM->gA1(), DMEM->gA2(), DMEM->gA3(), DMEM->gB11(),
                                               DMEM->gB12(), DMEM->gB13(), DMEM->gB21(), DMEM->gB22(),
                                               DMEM->gB23(), DMEM->gB31(), DMEM->gB32(), DMEM->gB33(),
                                               DMEM->gV11(), DMEM->gV12(), DMEM->gV13(),
                                               DMEM->gV21(), DMEM->gV22(), DMEM->gV23(),
                                               DMEM->gV31(), DMEM->gV32(), DMEM->gV33(),
#if defined(HYBRID_SG) || defined(HY_MOOD)
                                               DMEM->mask(),
#endif
                                               DMEM->lcount(),
#ifndef HYBRID_SORT_SPECIES
                 DMEM->lcountDM(),
#else
                 DMEM->lacountAGN(), //skip DM, Star, and AGN particles from initial leaf decomposition
```


Argonne NATIONAL LABORATORY

Common Host Interfaces : CUDA Implementation

```
template<typename ...Args>
struct KernelWrapper<void(Args...)>
{
    static void Invoke( void kernel(Args...), int numBlocks, int numThreads, size_t SMEM, cudaStream_t stream, Args ... args);
};

template<typename ...Args>
void KernelWrapper<void(Args...)>::Invoke( void kernel(Args...), int numBlocks, int numThreads, size_t SMEM, cudaStream_t stream, Args ... args)
{
    kernel<<<numBlocks, numThreads, SMEM, stream>>>( args ... );
};

//General invoke function for calling a GPU kernel
//Params: Function Name, number of cuda blocks (work groups), number of threads per block, shared memory bytes, and cuda stream.
template<typename K , typename ...Args>
void InvokeGPUKernel( K k, int nBlock, int BlockSize, size_t SMEM, cudaStream_t stream, Args ... args )
{
    int64_t n64 = nBlock; int64_t B64 = BlockSize;
    int64_t nThread = n64*B64;
    assert((nThread < (int64_t(1) << 31)) && (nThread >= 0)); /* assert that kernel number of threads is less than an int value */
    if(nThread == 0)return;/* no op if number of threads is zero. */
    typedef typename std::remove_pointer<K>::type KernelType;   // type of K is a _pointer_ to a function
    KernelWrapper<KernelType>::Invoke( k, nBlock, BlockSize, SMEM, stream, args ... );//invoke kernel
    cudaStreamSynchronize(stream); /* Synchronize Stream */
    cudaCheckError(); /* check for errors */
}
```

```
//GPU Asynchronous memset
//Params: void data pointer, int value, size_t count, stream
#define InvokeGPUMemsetAsync(devPtr, value, count, stream) { \
    cudaMemsetAsync(devPtr,value,count,stream); \
    cudaCheckError(); /* check for errors */ \
}
```

Credit: Nicholas Frontiere

Argonne NATIONAL LABORATORY

Common Host Interfaces : SYCL Implementation

```
//GPU Asynchronous memset
#define InvokeGPUMemsetAsync(devPtr, value, count, stream) {  \
  stream.memset(devPtr, value, count).wait();                 \
}

//Device memory malloc
#define InvokeGPUMalloc(devPtr, size, stream){        \
  *devPtr = (char*)sycl::malloc_device(size, stream);  \
  stream.wait();                                      \
}

//Memory copy between host and device
#define InvokeGPUMemcpy(dstPtr, srcPtr, count, direction, stream){  \
  stream.memcpy(dstPtr, srcPtr, count);                            \
  stream.wait();                                                   \
}

//Device memory free
#define InvokeGPUFree(devPtr, stream){  \
  sycl::free(devPtr, stream);           \
}

#define InvokeGPUKernelInter(NAME, nBlock, BlockSize, SMEM, sycl_queue, data1, data2, ...) {  \
  InvokeSYCLKernel<NAME>(nBlock, BlockSize, sycl_queue, data1, data2, __VA_ARGS__);   \
}

#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)
{
  auto sycl_kernel = K(args...);
  auto e = sycl_queue.parallel_for(sycl::nd_range<1>(nBlock*BlockSize, BlockSize), sycl_kernel);
  e.wait();
}
#endif // HACC_SYCL
```

- We cannot use a function pointer in this SYCL implementation, but a *functor* (C++ class) can be used as a template argument.
- To conform to the same host caller interface used by the CUDA implementation, we use a preprocessor macro to transform to the needed syntax.

SYCLomatic

https://github.com/oneapi-src/SYCLomatic

- Single file migration of CUDA kernels and manually write host code.
- Filed several bugs and made feature requests, e.g.,
  - Request for nd-range dimension to be specified
  - Bug translating `__syncwarp()` to `item_ct1.barrier()` instead of `item_ct1.get_sub_group().barrier();`
  - Request for functor form of function output, *outstanding*
  - Make DPCT open source.
    - Now SYCLomatic!
- We also use the incremental migration functionality for alternate code paths set by user macros.

```
/*CUDA*/
__global__ void updateBarAccel(const int* __restrict leafs1,…)
{
…
/*SYCL*/
void updateBarAccel(const int* __restrict leafs1,…, sycl::nd_item<1> item_ct1)
{
…
```


Argonne NATIONAL LABORATORY

SYCL Functors: source-to-source Translation

```
// SYCLomatic output
void updateBarAccel(const int* __restrict
leafs1,…, sycl::nd_item<1> item_ct1)
{
// function body
…
}
```

```
// Desired functor form
class updateBarAccel
{
private:
const int *__restrict leafs1;
…
public:
updateBarAccel(
const int *__restrict leafs1,
…
) : leafs1(leafs1), …
{}
[[intel::reqd_sub_group_size(HACC_SYCL_SG_SIZE)]]
void operator () (sycl::nd_item<1> item_ct1) const
{
//function body
```

Clang LibTooling

- The Intel oneAPI Data Parallel C++ compiler https://github.com/intel/llvm can support standalone clang-based tools via LibTooling.
- Using RecursiveASTVisitor based ASTFrontendActions we can parse SYCL files. https://clang.llvm.org/docs/RAVFrontendAction.html
  - Recursively traverse the Abstract Syntax Tree of the SYCL translation unit.
  - Identify functions and list function parameters.
  - Runs after the preprocessor, so retaining macros in source requires a little more effort.
- Builds using the documented instructions with little effort. Note this was using the provided Python scripts.

Performance Optimizations

Argonne
NATIONAL LABORATORY

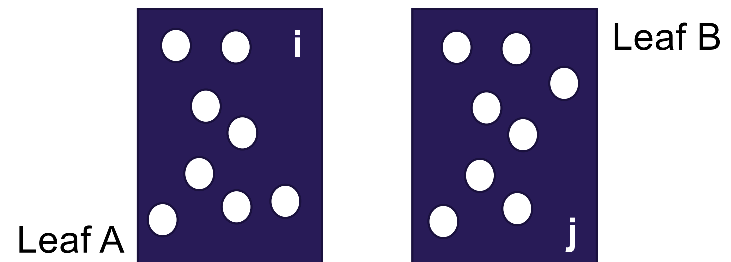
Computational Symmetries in CRK Hydro Kernels

- Half-warp and Full-warp algorithmic implementations have competing tradeoffs

CRK-SPH Estimate of Thermal Energy

$$\frac{Du_i}{Dt} = \frac{1}{2m_i} \sum_j V_i V_j (P_j + Q_j) v_{ij}^{\alpha} \left(\partial_\alpha \mathcal{W}_{ij}^R - \partial_\alpha \mathcal{W}_{ji}^R \right)$$

https://arxiv.org/abs/1605.00725



Half-warp lane layout

Lane	0	1	2	3	4	5	6	7
Leaf	A	A	A	A	B	B	B	B

Argonne NATIONAL LABORATORY

Symmetric Shuffles with Built-in group functions

iteration \ laneID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	17	16	19	18	21	20	23	22	25	24	27	26	29	28	31	30	1	0	3	2	5	4	7	6	9	8	11	10	13	12	15	14
2	18	19	16	17	22	23	20	21	26	27	24	25	30	31	28	29	2	3	0	1	6	7	4	5	10	11	8	9	14	15	12	13
3	19	18	17	16	23	22	21	20	27	26	25	24	31	30	29	28	3	2	1	0	7	6	5	4	11	10	9	8	15	14	13	12
4	20	21	22	23	16	17	18	19	28	29	30	31	24	25	26	27	4	5	6	7	0	1	2	3	12	13	14	15	8	9	10	11
5	21	20	23	22	17	16	19	18	29	28	31	30	25	24	27	26	5	4	7	6	1	0	3	2	13	12	15	14	9	8	11	10
6	22	23	20	21	18	19	16	17	30	31	28	29	26	27	24	25	6	7	4	5	2	3	0	1	14	15	12	13	10	11	8	9
7	23	22	21	20	19	18	17	16	31	30	29	28	27	26	25	24	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8
8	24	25	26	27	28	29	30	31	16	17	18	19	20	21	22	23	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7
9	25	24	27	26	29	28	31	30	17	16	19	18	21	20	23	22	9	8	11	10	13	12	15	14	1	0	3	2	5	4	7	6
10	26	27	24	25	30	31	28	29	18	19	16	17	22	23	20	21	10	11	8	9	14	15	12	13	2	3	0	1	6	7	4	5
11	27	26	25	24	31	30	29	28	19	18	17	16	23	22	21	20	11	10	9	8	15	14	13	12	3	2	1	0	7	6	5	4
12	28	29	30	31	24	25	26	27	20	21	22	23	16	17	18	19	12	13	14	15	8	9	10	11	4	5	6	7	0	1	2	3
13	29	28	31	30	25	24	27	26	21	20	23	22	17	16	19	18	13	12	15	14	9	8	11	10	5	4	7	6	1	0	3	2
14	30	31	28	29	26	27	24	25	22	23	20	21	18	19	16	17	14	15	12	13	10	11	8	9	6	7	4	5	2	3	0	1
15	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

sycl::permute_group_by_xor(sg, value, mask);
sycl::select_from_group(sg, value, id);
+ iteration);

mask=iteration
id=laneID ^ (HALF_SG_SIZE

Argonne NATIONAL LABORATORY

Using Shared Local Memory (SLM)

```
// SLM shuffles
template <typename T>
inline T sub_group_slm_exchange(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];
}
```

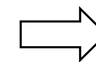
Warning: code assumes the amount of SLM available per work item can accommodate data of size T.

Argonne NATIONAL LABORATORY

Inline Assembly: experimental

```
#if HACC_SYCL_SG_SIZE==16
#define BUTTERFLY_SHFT(SRC_R, SRC_L) {                                        \
  __asm__ (                                                                   \
    "{\n"                                                                     \
    ".decl TMP0 v_type=G type=f num_elts=16 align=GRF\n"                      \
    ".decl TMP1 v_type=G type=f num_elts=16 align=GRF\n"                      \
    "mov (M1_NM, 16) TMP0(0,0)<1> %1(0,0)<0;8,1>\n"                           \
    "mov (M1_NM, 16) TMP1(0,0)<1> %1(0,8)<0;8,1>\n"                           \
    "mov (M1, 8) %0(0,8)<1> TMP0(0,"#SRC_R")<1;1,0>\n"                        \
    "mov (M1, 8) %0(0,0)<1> TMP1(0,"#SRC_L")<1;1,0>\n"                        \
    "}\n"                                                                     \
    : "=rw"(return_value)                                                     \
    : "rw"(value));                                                           \
}
#elif HACC_SYCL_SG_SIZE==32
#define BUTTERFLY_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
```

```
#if defined(__SYCL_DEVICE_ONLY__)
    switch (shift) {
#if SYCL_SG_SIZE==16
      case 0:
        BUTTERFLY_SHFT(0,8);
        break;
      case 1:
        BUTTERFLY_SHFT(1,7);
        break;
      case 2:
        BUTTERFLY_SHFT(2,6);
        break;
      case 3:
        BUTTERFLY_SHFT(3,5);
        break;
      case 4:
        BUTTERFLY_SHFT(4,4);
        break;
      case 5:
        BUTTERFLY_SHFT(5,3);
        break;
      case 6:
        BUTTERFLY_SHFT(6,2);
        break;
      case 7:
        BUTTERFLY_SHFT(7,1);
        break;
#endif
#if SYCL_SG_SIZE==32
      case 0:
        BUTTERFLY_SHFT(0,16);
        break;
      case 1:
        BUTTERFLY_SHFT(1,15);
        break;
…
```



```
inline float
butterfly_shuffle(sycl::sub_group sg,
                  const float value,
                  const int shift) const
    {
      float return_value = 0;


      return return_value;
    }
```


Argonne NATIONAL LABORATORY

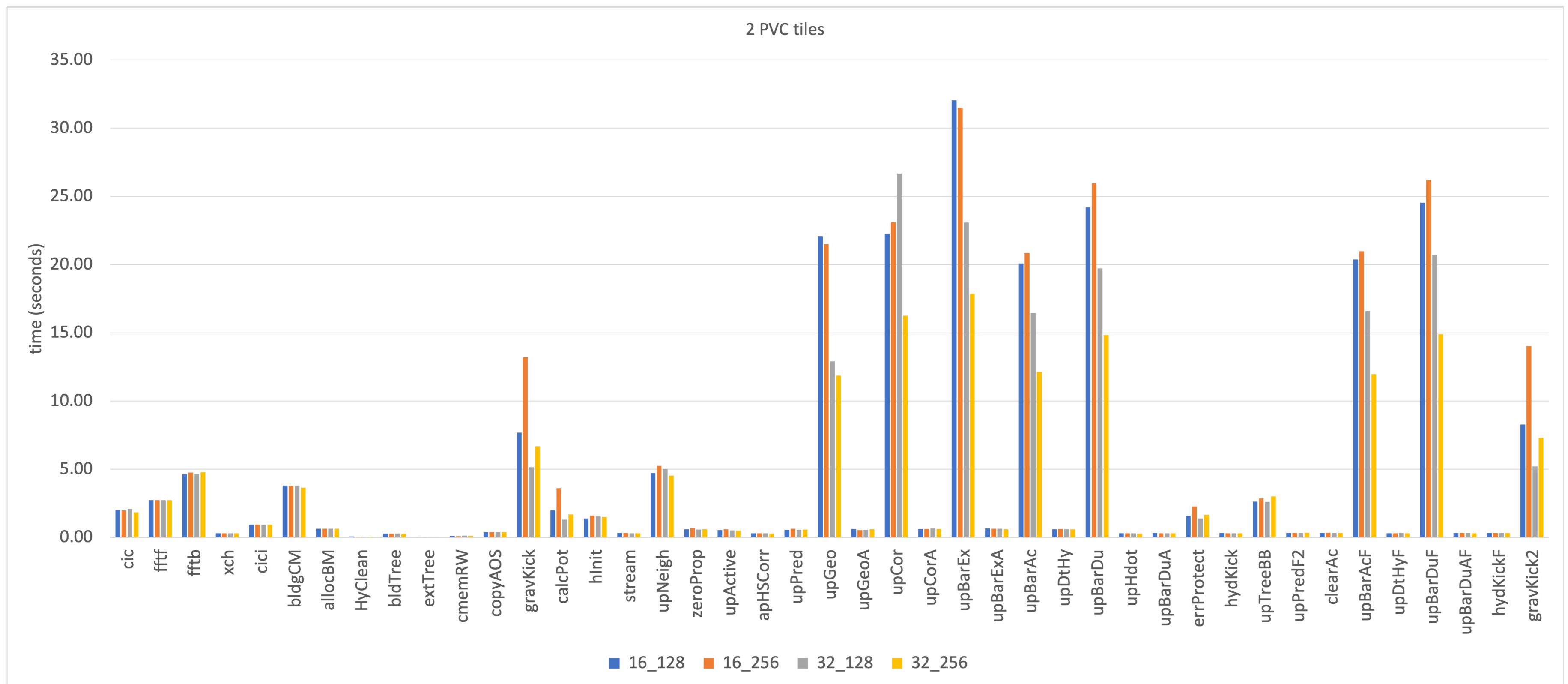
Results

All results are presented are using SYCLomatic for CUDA to SYCL kernel migration with the original half-warp algorithm used in CUDA and the SLM implementation for sub-group data exchange.

Comparison Run

Simulation	CRK-HACC Adiabatic
Dark Matter	256^3
Baryon	256^3
Initial redshift	200.0
Final redshift	50.0
Full simulation steps	5
Gravity sub-cycles	4
Hydro sub-cycles	fixed
System	Sunspot
Nodes	1
Ranks	8, 4 rank/PVC tile
Wall Time	~3min

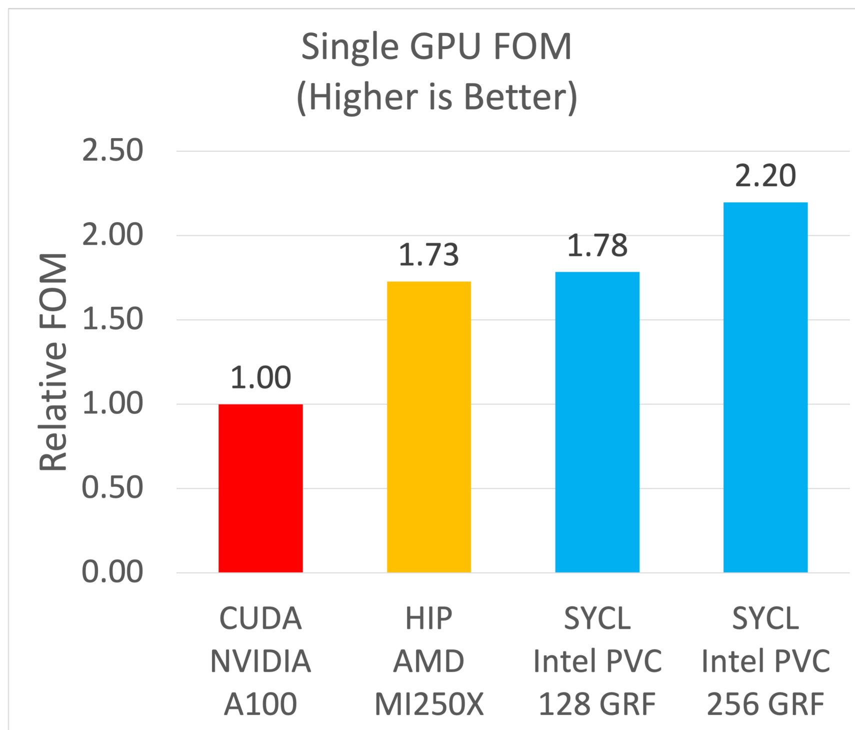
Timing Breakdown (per kernel)

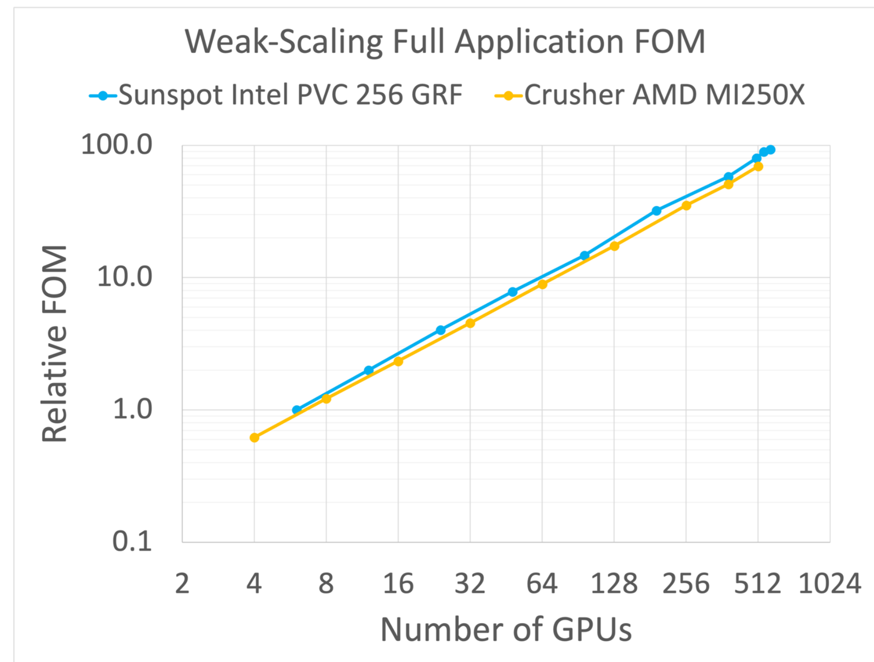


Credit: Adrian Pope

Argonne NATIONAL LABORATORY

Relative Performance Comparison





Credit: Adrian Pope

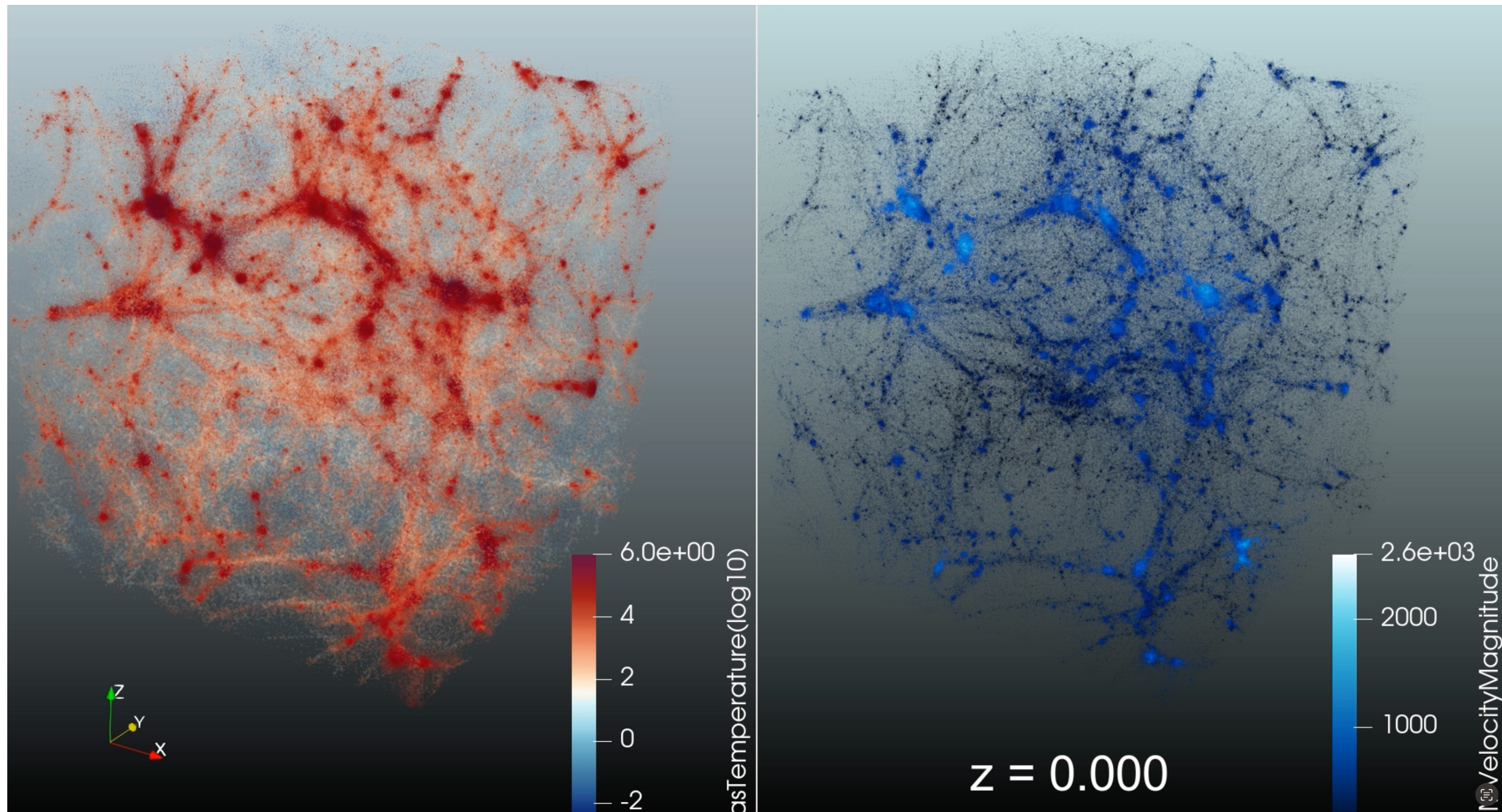
Verification and Visualization (Movie) Run

Simulation	CRK-HACC Adiabatic
Dark Matter	256^3
Baryon	256^3
Initial redshift	200.0
Final redshift	0.0
Full simulation steps	625
Gravity sub-cycles	4
Hydro sub-cycles	adaptive
System	Sunspot
Nodes	1
Ranks	8, 1 rank/PVC tile
Wall Time	~6hrs

Validation performed by comparing the matter power spectrum to a reference run using CUDA on A100.

Argonne NATIONAL LABORATORY

Sunspot Movie



Credit: Silvio Rizzi

Argonne NATIONAL LABORATORY

Questions

Thank you