Hewlett Packard Enterprise

CCE Offloading Models

Comprehensive General LUMI Course

April 23–26, 2024

Agenda

- Introduction to how to program with directive-based approach
  - This is not a presentation on how to program with OpenMP/OpenACC!

- Directive based approach for execution offloading with CCE
  - CCE OpenMP support
  - CCE OpenACC support

- Programming languages to accelerate applications

Approaches to Accelerate Applications

Accelerated Libraries

- The easiest solution, just link the library to your application without in-depth knowledge of GPU programming
- Many libraries are optimized by GPU vendors, eg. algebra libraries

Directive based methods

- Add acceleration to your existing code (C, C++, Fortran)
- Can reach good performance with somehow minimal code changes
- OpenACC, OpenMP

Programming Languages

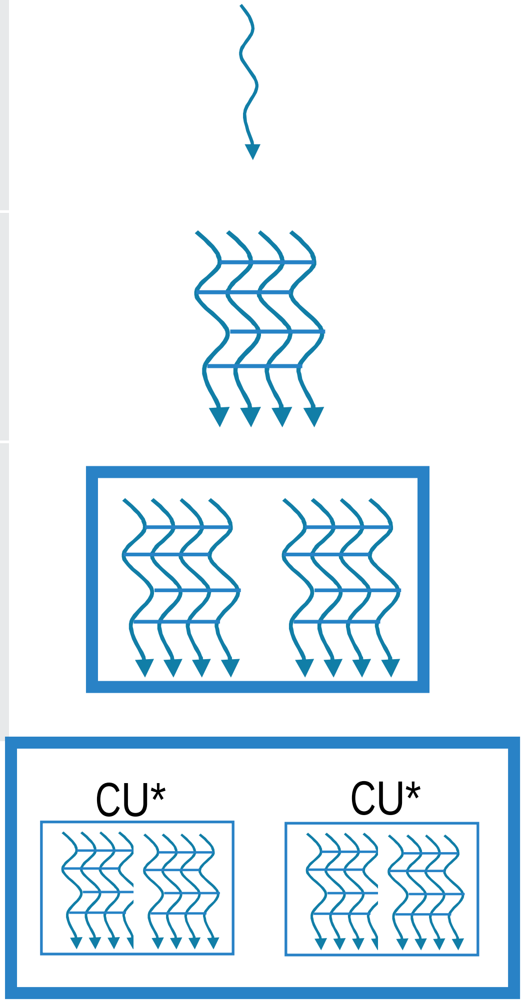
- Maximum flexibility, require in-depth knowledge of GPU programming and code rewriting (especially for Fortran)
- Kokkos, RAJA, CUDA, HIP, OpenCL, SYCL

Directive-based programming

A short introduction

The Multiple Dimensions of GPU Parallelism

AMD	NVIDIA	Description
Work item	Thread	• Fine-grained, lock-step parallelism • Performs best with stride-1 data accesses • Performs best with non-divergent control flow
Wavefront	Warp	• Fine-grained, independent parallelism • NVIDIA warp size is 32 threads • AMD wavefront size is 64 work items
Work group	Thread block	• Loosely-coupled, course-grained parallelism • Collective synchronization prohibited • Performs best with massive parallelism • Performance scales with more powerful GPUs



GPU

* Compute Unit

Directive-based programming models and Portability

- Huge potential to provide cross-architecture portability (CPUs and GPUs)
  - Code regions are offloaded from a host CPU to be computed on an accelerator
  - Performance portability across vendors (in principle)
- Standard specifications that all compiler vendors can implement
  - Define C/C++ and Fortran bindings
  - Has been critical for Fortran, especially for offloading
- Significant opportunities for improving construct-to-hardware mapping
  - Better cross-vendor consistency
- OpenMP offload and OpenACC support features to integrate with models such as CUDA and HIP
  - Optimize performance for a limited set of OpenMP/OpenACC constructs and APIs
  - Provide a portable model similar to existing kernel languages (e.g., CUDA or HIP)

Directive-based programming

- Directives provide high-level approach
  - \+ Based on original source code (Fortran, C, C++)
  - \+ Easier to maintain/port/extend code
  - \+ Users with OpenMP experience find it a familiar programming model
  - \+ Compiler handles repetitive boilerplate code
    - \+ Memory allocations, data transfers....
  - \+ Compiler handles default scheduling
    - \+ User can step in with clauses, but only where needed
- Possible performance sacrifice
  - – CCE: OpenMP/OpenACC aims to be close to native CUDA/HIP performance
  - – Small performance sacrifice is acceptable and attractive
    - – Trade this off against gains in portability and productivity
  - – Who handcodes in assembly language these days?
    - – After all, you could recode your CPU code in assembler to get additional boost...

Offloading: OpenMP VS OpenACC

- OpenACC is known to be more descriptive
  - The programmer uses directives to tell the compiler how/where to parallelize the code and to manage data between potentially separate host and accelerator memories
  - Example
    - An OpenACC parallel loop directive tells the compiler that it's a true parallel (data independent) loop, so it can spread its execution across threads or run them across SIMD lanes, choosing very different mappings depending on the underlying hardware

- OpenMP offloading approach, on the other hand, is known to be more prescriptive
  - Supports different hardware, not just accelerators
  - The programmer uses directives to tell the compiler more explicitly how/where to parallelize the code, instead of letting the compiler decides
  - Example
    - An OpenMP parallel loop directive doesn't guarantee that a loop is in fact a parallel loop. Rather, it instructs the compiler to schedule the iterations of that loop across the available resources according to either a default or user-specified scheduling policy
    - The programmer promises that the generated code is correct, and that any data races are handled by the programmer using OpenMP-supplied synchronization constructs

OpenMP/OpenACC offload example

- OpenACC (only Fortran support in CCE)

```
!$acc data copyin(B[1:n], C[1:n]) copyout(A[1:n])
!$acc parallel loop
do i = 1, n
  A(i) = B(i) + scalar * C(i)
end do
!$acc end parallel loop
!$acc end data
```

- It schedules the loop to be executed in parallel on the GPU
- Levels of parallelism are automatically decided and applied by the compiler to map the hardware resources

- OpenMP (C/C++/Fortran support in CCE)

```
#pragma omp target data map(to: B[0:n], C[0:n]) map(from: A[0:n])
{
  #pragma omp target \
              teams \
              distribute \
              parallel for simd
  for (size_t i = 0; i < n; i++) {
    A[i] = B[i] + scalar * C[i];
  }
}
```

- The `target` directive offloads the execution, no parallelization
- `teams` creates a league of teams and one master thread in each team, but <u>no worksharing</u> among the teams
- `distribute` distributes the iterations across the master threads in the teams, but <u>no worksharing</u> among the threads within one team
- `parallel do/for`: threads are activated within one team and worksharing among them
- `simd` enables SIMD instructions

CCE OpenACC/OpenMP construct mapping to GPU

<table>
<tr><th>NVIDIA</th><th>AMD</th><th>CCE Fortran OpenACC</th><th>CCE Fortran OpenMP</th><th>CCE C/C++ OpenMP</th></tr>
<tr><td>Thread block</td><td>Work group</td><td><code>acc gang</code></td><td><code>omp teams</code></td><td><code>omp teams</code></td></tr>
<tr><td>Warp</td><td>Wavefront</td><td><code>acc worker</code></td><td rowspan="2"><code>omp simd</code></td><td rowspan="2"><code>omp parallel</code>
<code>omp simd</code></td></tr>
<tr><td>Thread</td><td>Work item</td><td><code>acc vector</code></td></tr>
</table>

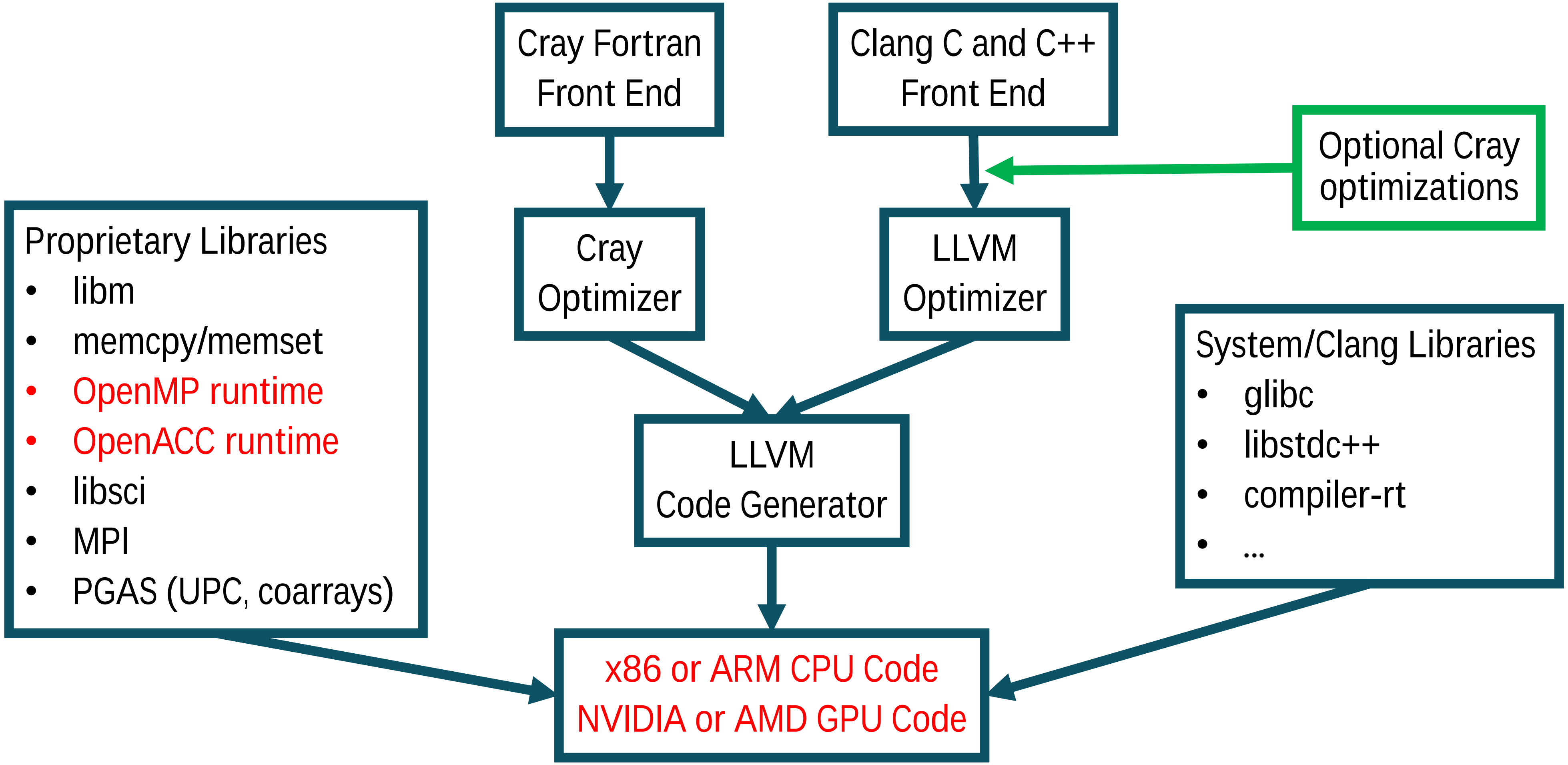
- Current best practice for OpenMP:
  - Use “teams” to express GPU thread block/work group parallelism
  - Use “parallel for simd” to express GPU thread/work item parallelism

  ➔ The loopmark listing file will indicate how each construct maps to GPU parallelism
- Future direction:
  - Improve CCE support for “parallel” and ”simd” in accelerator regions
  - Upstream Clang is expanding support for “simd” in accelerator regions

Long-term goal: let users express parallelism with any construct they think makes sense, and CCE will map to available hardware parallelism

CCE Offloading Feature Highlights and Best Practices

Current CCE architecture



CCE OpenMP Support

- Enabled with `-fopenmp` flag
  - An appropriate accelerator target module must be loaded in order to use target directives
- Uses proprietary OpenMP runtime libraries
- Supports cross-language and cross-vendor OpenMP interoperability
- Implements HPE-optimized code generation for OpenMP offload regions
- Supports OpenMP allocators (e.g., CPU “pinned”, GPU “shared” and “managed”)
- Full OpenMP 4.5 support for Fortran, C, and C++
- OpenMP 5.x – in progress, implementation phased in over several CCE releases
  - See release notes (accessible via `module help cce`) and `intro_openmp` man page for full list of supported features
  - OpenMP 5.0 is near complete as of CCE 14.0
  - OpenMP 5.1/5.2 support in progress for CCE 15.0+

CCE OpenMP 5.0 Status

CCE 10.0 (May 2020)

- OMP_TARGET_OFFLOAD
- reverse offload
- implicit declare target
- omp_get_device_num
- OMP_DISPLAY_AFFINITY
- OMP_AFFINITY_FORMAT
- set/get affinity display
- display/capture affinity
- requires
- unified_address
- unified_shared_memory
- atomic_default_mem_order
- dynamic_allocators
- reverse_offload
- combined master constructs
- acq/rel memory ordering (Fortran)
- deprecate nested-var
- taskwait depend
- simd nontemporal (Fortran)
- lvalue map/motion list items
- allow != in canonical loop
- close modifier (C/C++)
- extend defaultmap (C/C++)

CCE 11.0 (Nov 2020)

- noncontig update
- map Fortran DVs
- host teams
- use_device_addr
- nested declare target
- allocator routines
- OMP_ALLOCATOR
- allocate directive
- allocate clause
- order(concurrent)
- atomic hints
- default nonmonotonic
- imperfect loop collapse
- pause resources
- atomics in simd
- simd in simd
- detachable tasks
- omp_control_tool
- OMPT
- OMPD
- declare variant (Fortran)
- loop construct
- metadirectives (Fortran)
- pointer attach
- array shaping
- acq/rel memory ordering (C/C++)
- device_type (C/C++)
- non-rectangular loop collapse (C/C++)

CCE 12.0 (Jun 2021)

- device_type (Fortran)
- affinity clause
- conditional lastprivate (C/C++)
- simd if (C/C++)
- iterator in depend (C/C++)
- depobj for depend (C/C++)
- task reduction (C/C++)
- task modifier (C/C++)
- simd nontemporal (C/C++)
- scan (C/C++)
- lvalue list items for depend
- mutexinoutset (C/C++)
- taskloop cancellation (C/C++)

CCE 13.0 (Nov 2021)

- declare variant (C/C++)
- metadirectives (C/C++)
- mapper (C/C++)
- extend defaultmap (Fortran)
- close modifier (Fortran)
- mutexinoutset (Fortran)

CCE 14.0 (May 2022)

- task reduction (Fortran)
- task modifier (Fortran)
- target task reduction (Fortran)
- simd if (Fortran)

Future CCE Release

- loop construct (C/C++)
- mapper (Fortran)
- iterator in depend (Fortran)
- non-rectangular loop collapse (Fortran)
- depobj for depend (Fortran)
- uses_allocators
- concurrent maps
- taskloop cancellation (Fortran)
- scan (Fortran)
- target task reduction (C/C++)

Refer to CCE release notes or intro_openmp man page for current implementation status

OpenMP Interoperability

- OpenMP CPU interoperability
  - CCE's libcraymp behaves as drop-in replacement for Clang's libomp and GNU's libgomp
  - GNU OpenMP interface support is currently limited to OpenMP 3.1 constructs
- OpenMP GPU interoperability
  - CCE's libcrayacc behaves as drop-in replacement for Clang's libomptarget
  - No planned support for GNU OpenMP offload interface
  - Device code relies on each vendor's device runtime library
  - Each vendor's device code is linked into a separate "device image"
  - CCE OpenMP offload linker tool handles device unbundling and linking
  - Requires linking with CCE, or manually invoking the CCE OpenMP offload linker tool (`${CC_X86_64}/bin/cce_omp_offload_linker`)

CCE OpenACC Support

- CCE supports OpenACC 2.0+ for Fortran
- C/C++ OpenACC support was dropped in CCE 10.0
- Full OpenACC 3.2 Fortran support planned for a future CCE release
- CCE OpenMP and OpenACC implementations share a common codebase
  - Significant overlap in both compiler and runtime library
  - Same performance should be achievable with either model
- See release notes (accessible via `module help cce`) and `intro_openacc` man page for full list of supported features

“Offload” Host Execution

- Useful for debugging and possibly development with no accelerator hardware
- The target module `craype-accel-host` supports compiling and running an OpenMP/OpenACC applications on the host processor
  - This provides source code portability between systems with and without an accelerator
  - The OpenACC directives are automatically converted at compile time to OpenMP equivalent directives

CCE OpenMP/OpenACC Flags

Capability	CCE Fortran Flags	CCE C/C++ Flags
Enable/Disable OpenMP (disabled at default)	-f[no-]openmp -h[no]omp	-f[no-]openmp
Enable/Disable OpenACC (enabled at default)	-h[no]acc	N/A
Enable HIP	N/A	-x hip --rocm-path=$ROCM_PATH –L $ROCM_PATH/lib –lamdhip64

Offloading Target	All CCE Compilers (accel modules)	CCE C/C++ (optional flags)
Native Host CPU	craype-accel-host	(default without flags; no warning)
NVIDIA Volta	craype-accel-nvidia70	-fopenmp-targets=nvptx64 -Xopenmp-target -march=sm_70
AMD MI100	craype-accel-amd-gfx908	-fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx908
AMD MI250X	craype-accel-amd-gfx90a	-fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx90a

CCE - ROCm Compatibility/Interoperability

- CCE HIP offloading relies on ROCm headers, host libraries, and device bitcode libraries
- CCE OpenMP offloading relies on ROCm host libraries and device bitcode libraries
- Device bitcode libraries require a matching LLVM version between CCE and ROCm
- CCE OpenMP interoperability relies on compatible Clang OpenMP runtime ABI

	HIP/OpenMP (CCE Only)	OpenMP Interop (CCE + ROCm)
CCE 13.0.0	ROCm 4.1 – 4.5	ROCm 4.2 – 4.3
CCE 13.0.1	ROCm 4.1 – 4.5	ROCm 4.2 – 4.5
CCE 13.0.x	ROCm 4.1 – 4.5	ROCm 4.2 – 4.5
CCE 14.0.0	ROCm 5.0 – 5.2	ROCm 5.0 – 5.2
CCE 15.0.x	ROCm 5.0 – 5.4	ROCm 5.0 – 5.4

Runtime Offloading Messages

- Environment variable CRAY_ACC_DEBUG=[1-3]
- Emits runtime debug messages for offload activity (allocate, free, transfer, kernel launch, etc)

```
program main
 integer :: aaa(1000)
 aaa = 0
 !$omp target teams distribute map(aaa)
 do i=1,1000
  aaa(i) = 1
 end do

 if ( sum(abs(aaa)) .ne. 1000 ) then
  print *, "FAIL"
  call exit(-1)
 end if
 print *, "PASS"
end program main
```

```
ACC: Version 4.0 of HIP already initialized, runtime
version 3241
ACC: Get Device 0
ACC: Set Thread Context
ACC: Start transfer 1 items from hello_gpu.f90:4
ACC:       allocate, copy to acc 'aaa(:)' (4000 bytes)
ACC: End transfer (to acc 4000 bytes, to host 0 bytes)
ACC: Execute kernel main_$ck_L4_1 blocks:8 threads:128
from hello_gpu.f90:4
ACC: Start transfer 1 items from hello_gpu.f90:7
ACC:       copy to host, free 'aaa(:)' (4000 bytes)
ACC: End transfer (to acc 0 bytes, to host 4000 bytes)
 PASS
```

CCE OpenMP Allocator Specialization

Use Case	Allocator Mechanism	Notes
"Pinned" CPU memory	Allocator with "pinned" trait set	• Maps to hipMallocHost
"Shared" GPU memory	omp_cgroup_mem_alloc predefined allocator	• Maps to static allocation in LDS memory • Must be lexically specified on "allocate" clause on "teams" construct • Currently supported for Fortran only
"Managed" memory	cray_omp_get_managed_memory_allocator_handle()	• Maps to hipMallocManaged • CCE-specific extension • Topic of interest for OpenMP committee

OpenMP unified shared memory

- AMD Mi200 GPUs provide hardware support for managed memory where host and device memory is coherent
  - Avoid the burden of explicitly copy data between host and device, relying on the ROCM runtime for moving data
  - The same pointer to an object to be used both by the CPU and a GPU even if the physical location of the object were moved by the operating system or device driver
  - Expect some overhead, but less code (and pain)

- OpenMP 5.0 introduces the directive

```
omp requires unified_shared_memory
```

  - Uses standard system memory allocators (NOTE: it requires convenient memory alignment for GPU)
  - Do not need any `map` clause, OpenMP directives are used primarily for expressing parallelism

MI250X Recoverable page fault Modes (XNACK)

- XNACK allows GPU to recover from page faults, necessary for general unified memory support
- Runtime XNACK hardware mode is controlled by a ROCr environment variable, HSA_XNACK=1
- Each process can set the XNACK hardware mode independently, once at program startup
- GPU code must be compiled with a "compatible" XNACK software/compilation mode
  - CCE currently always compiles OpenMP with default XNACK mode
  - CCE HIP supports "target ID" synax for the "--offload-arch" flag

XNACK Compile Mode	Compiler Flags
any/both (default)	CC -x hip --offload-arch=gfx90a ...
on (xnack+)	CC -x hip --offload-arch=gfx90a:xnack+ ...
off (xnack-)	CC -x hip --offload-arch=gfx90a:xnack- ...
fat binary (xnack+,xnack-)	CC -x hip --offload-arch=gfx90a:xnack+ --offload-arch=gfx90a:xnack- ...

XNACK Compile and Runtime MODE Combinations

Compile Mode	HSA_XNACK=0	HSA_XNACK=1
any/both (default)	Functional, without demand paging and migration; performance overhead	Functional, with demand paging and migration; performance overhead
on (xnack+)	Runtime error due to XNACK mode mismatch	Functional, with demand paging and migration; performance overhead
off (xnack-)	Functional, without demand paging and migration; no performance overhead	Runtime error due to XNACK mode mismatch
fat binary (xnack+,xnack-)	Runtime selection of “xnack-” binary, resulting in same behavior as above	Runtime selection of “xnack+” binary, resulting in same behavior as above

CCE OpenMP unified memory support for AMD MI200

1. If you don't use unified memory, CCE's default runtime behavior for OpenMP map clauses is to allocate/transfer GPU memory
2. We can dynamically enable GPU managed memory for OpenMP map clauses
   - No code changes, i.e. automatic detection of data movement
   - Set env var CRAY_ACC_USE_UNIFIED_MEM=1 and `HSA_XNACK=1`
     - If `HSA_XNACK=1` is not set, the CCE OpenMP library will issue a runtime error when a variable is first mapped
   - Skips explicit allocate/transfer for all system memory
   - Global "declare target" variables will still be allocated separately (compiler statically emits a device copy)
   - Save data movement at the cost of ignoring the explicit `map` semantic, check the data movements with CRAY_ACC_DEBUG
3. Statically enable GPU unified memory for OpenMP map clauses
   - Compile with `requires unified_shared_memory` directive
     - The `CRAY_ACC_USE_UNIFIED_MEM` environment variable is implied when using `omp requires unified_shared_memory`, and therefore it does not need to be explicitly set
   - Set env var HSA_XNACK=1
     - If `HSA_XNACK=1` is not set, the CCE OpenMP library will issue a runtime error when a variable is first mapped

CCE Offload Summary

- Consistent development environment across a wide variety of CPU and GPU targets
- Support for the latest base language standards
  - Fortran 2018 support (including coarray teams)
  - C11 and C++17 support
- Support for several on-node parallel/offloading models
  - OpenMP 4.5, working towards 5.2
  - OpenACC 2.0, working towards 3.2
  - HIP
- Please reach out or file bugs if you have questions or encounter issues
- Man pages are your best friends:
  - `intro_openacc`
  - `intro_openmp`

Programming-language GPU offload

Setting the scene

Approaches to Accelerate Applications

Accelerated Libraries

- The easiest solution, just link the library to your application without in-depth knowledge of GPU programming
- Many libraries are optimized by GPU vendors, eg. algebra libraries

Directive based methods

- Add acceleration to your existing code (C, C++, Fortran)
- Can reach good performance with somehow minimal code changes
- OpenACC, OpenMP

Programming Languages

- Maximum flexibility, require in-depth knowledge of GPU programming and code rewriting (especially for Fortran)
- Kokkos, RAJA, CUDA, HIP, OpenCL, SYCL

HIP Compilation via PE wrappers (suggested method)

- Can use Cray Compiler (PrgEnv-cray) or AMD Compiler (PrgEnv-amd)
  - PrgEnv-amd provides access to clang installed into ROCm

```
module load PrgEnv-...
module load craype-accel-amd-gfx90a
module load rocm
```

- Compile HIP files (eg. with .cpp extension) with

```
CC -xhip
```

  - Do not use `-xhip` for linking
  - Promote compilation to C++ (`cc -xhip == CC -xhip`)
  - Specify all offload flags, i.e. `-D__HIP_PLATFORM_AMD__ --offload-arch=gfx90a`
  - Add paths to HIP includes and libraries
  - Add main HIP and ROCm libraries
  - Need to specify any other library, e.g. `-lhipblas`
  - As usual include other wrapper flags and libraries, e.g. MPI

AMD HIPCC

- AMD provides HIPCC to compile HIP code
- HIPCC is a wrapper around clang, coming from ROCM installation (set via PrgEnv-amd or rocm module)

```
> hipcc --version
HIP version: 5.2.21153-02187ecf
AMD clang version 14.0.0 (https://github.com/RadeonOpenCompute/llvm-project roc-5.2.3
22324 d6c88e5a78066d5d7a1e8db6c5e3e9884c6ad10e)
Target: x86_64-unknown-linux-gnu
Thread model: posix
InstalledDir: /opt/rocm/llvm/bin
```

- Need to specify all offload flags by hand, i.e. `-D__HIP_PLATFORM_AMD__ --offload-arch=gfx90a`
- Possible incompatibility between AMD-clang (v14) and CCE-clang (v15)
  - Suggested to use Cray compiler for linking
- Three cases when it worth using it:
  - Debugging
  - HIP targeting NVIDIA devices
  - GNU compiler mixing

GNU Compiler and HIP code

- The GNU compilers cannot be used to compile HIP code
  - All HIP kernels must be separated from CPU code

- HIP kernels must be compiled with hipcc (available via the `rocm` module)
- All non-HIP code must be compiled with the `PrgEnv-gnu` wrappers (ftn/cc/CC)
  - Linking must be performed with the wrappers

- Note about OpenMP library:
  - GNU is using libgomp library, while HIP (LLVM) is using libomp library
    - Cannot mix the two, e.g. do not use OpenMP in the HIP code

CCE HIP support

- CCE 11.0 (Nov 2020) introduced support for compiling HIP source files targeting AMD GPUs
- CCE HIP support leverages AMD's open-source HIP implementation in upstream Clang/LLVM
- CCE relies on HIP header files and runtime libraries from a standard AMD ROCm install
- CCE does not provide a "hipcc" wrapper – invoke the "CC" compiler driver directly

CCE HIP Flag	Description
-x hip	Enables HIP compilation for subsequent input files (avoid on link line or follow with "-x none")
--offload-arch=gfx90a	Specifies the MI200 offload target architecture
--rocm-path=<ROCM_PATH>	Specifies the location of a ROCm install; not required when $ROCM_PATH environment variable is set
-f[no-]gpu-rdc	Enables (disables) relocatable device code, producing bundled HIP offload object files and allowing cross-file references in HIP device code (default: -fno-gpu-rdc)
--hip-link	Enables device linking for bundled HIP offload object files; required when compiling with -fgpu-rdc
-mllvm -amdgpu-early-inline-all=true -mllvm -amdgpu-function-calls=false	Optimization flags that AMD's "hipcc" wrapper script provides; may provide additional performance benefit

Interoperability with OpenMP

- Mixing OpenMP and HIP in the same C/C++ compilation unit
  - No OpenMP offload supported (only CPU execution), the order of the flags matters: `-fopenmp –xhip`

  ➔Use different compilation units for OpenMP offload and HIP
    - Linking with `-fopenmp` only, don’t use `-xhip`

Multi-GPUs

- Driver associates a number for each HIP-capable GPU in a node, starting from 0
  - A process establishes a GPU context with each GPU can have access to
  - Several processes can create contexts for a single device, e.g. MPI ranks can share the same GPUs
    - Can be useful to maximize GPU occupancy
  - By default, threads on the same process share the primary context (for each device)

- Multi-GPU programming models

  a. One GPU per process
     - All HIP calls running on GPU 0

  b. Multiple GPUs per process
     - Process manages all context switching between devices

  c. One GPU per thread in multi-threaded applications
     - Syncing is handled through thread synchronization requirements
     - HIP API is threadsafe

Multi-GPUs per process

- The function `hipSetDevice()` is used for selecting the desired device
  - Each following HIP call will execute only on the selected device

```
for (unsigned int idev = 0; idev < deviceCount; ++idev) {
  hipSetDevice(idev);
  kernel<<<blocks, threads>>>(args[idev]);
}
```

➢ https://docs.amd.com/projects/HIP/en/latest/doxygen/html/group__device.html

- It is the user responsibility to make sure the data and execution are on the same device
  - Be aware that calls to external libraries can change device
  - OpenMP offload regions can set different devices
- Suggested approach
  - Use a single device per process via a proper binding procedure (see binding slides)

MPI and RCCL communications

- GPU-aware MPI communications
  - Pass GPU pointers to MPI calls to enable direct transfer between GPU buffers
  - Enable fast GPU Peer2Peer for intra-node MPI transfers
  - Some MPI calls will run operations via GPU kernels, e.g. `MPI_Allreduce`
  - ➢ See MPI slides for more details

- RCCL (ROCm Communication Collectives Library)
  - A stand-alone library of standard collective communication routines for GPUs, e.g. all-reduce
  - Maximize throughput and latency
  - Can be used in MPI applications, replacing MPI collective calls (different API though)
  - Only a single process per GPU
  - Particularly used in AI applications
  - ➢ https://github.com/ROCm/rccl
  - ➢ https://rocm.docs.amd.com/projects/rccl/en/latest/

Kokkos / Raja / Alpaka and SYCL

- Portability frameworks based on C++
  - CPUs & GPUs – AMD, Intel, NVIDIA
  - High-level abstraction for parallel processing via C++ constructors

- They not directly supported by the PE but can be built on top

- https://kokkos.org/
- https://raja.readthedocs.io/en/develop/
- https://alpaka.readthedocs.io/en/0.5.0/index.html
- https://www.khronos.org/sycl/

Questions?

