Programming Models on Polaris

Abhishek Bagusetty, Colleen Bertoni, Brian Homerding
Argonne Leadership Computing Facility

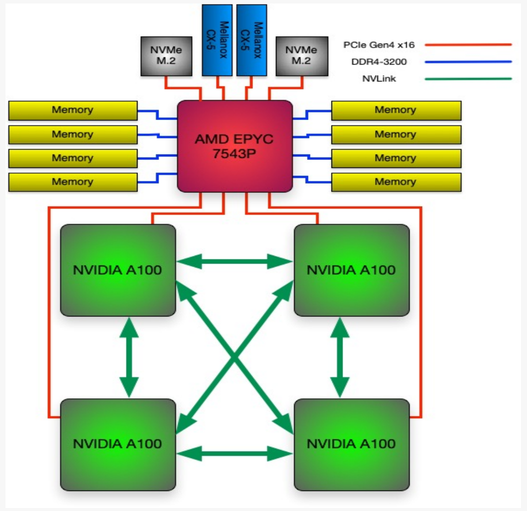
www.anl.gov

References

- Argonne documentation
  - https://www.alcf.anl.gov/support/user-guides/polaris/hardware-overview/machine-overview/index.html
- Perlmutter at NERSC is a similar system to Polaris
  - https://www.nersc.gov/assets/Uploads/ProgrammingModels.pdf

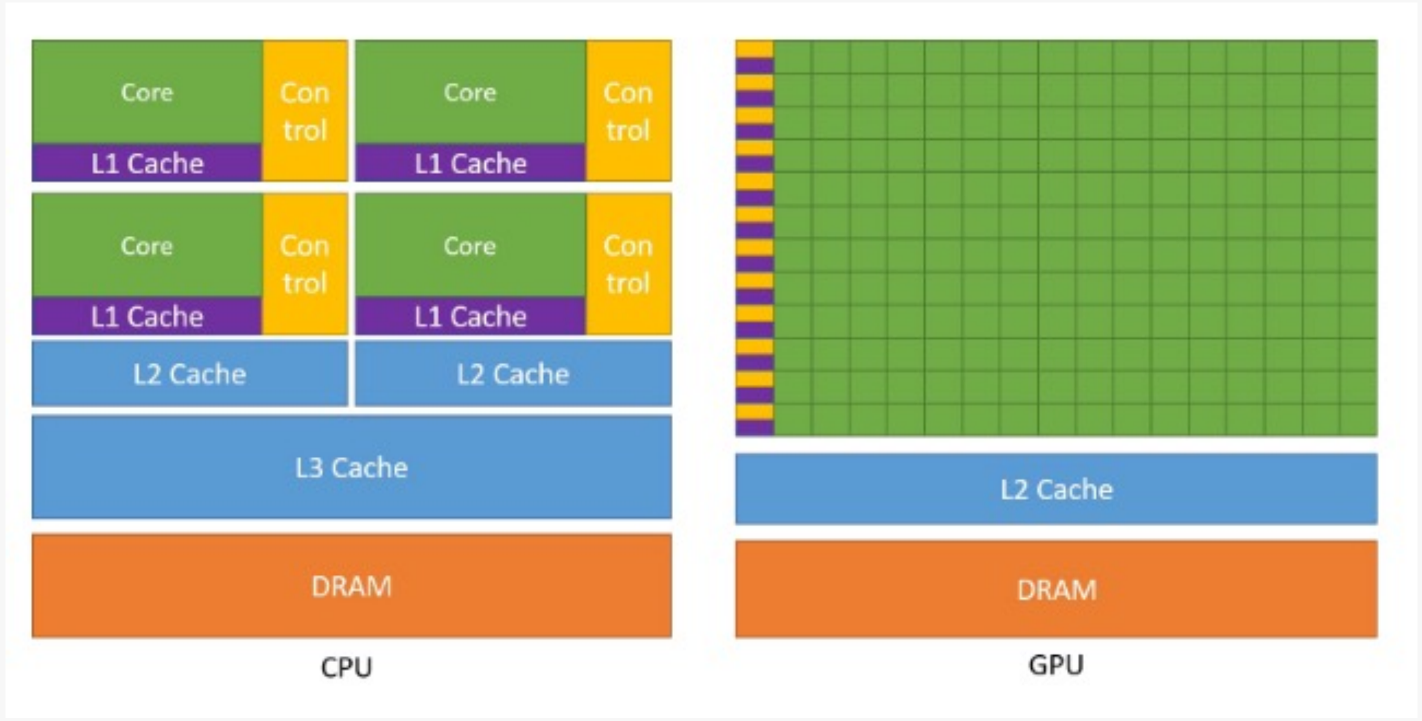
Polaris

- 40 Compute Racks
- 14 Nodes per Compute Rack (560 Nodes)
- Each node has 1 CPU and 4 GPUs
- 19.5 Tflops floating point operations per second per GPU (FP64 Tensor Core)
  - 44 Pflops for the whole system
- 1.6 TB/s memory bandwidth per GPU
  - 3.5 PB/s for the whole system



Reminder about CPU and GPU programming

- CPU
  - Optimized to reduce latency
  - Good for serial work
  - Relatively high clock frequency

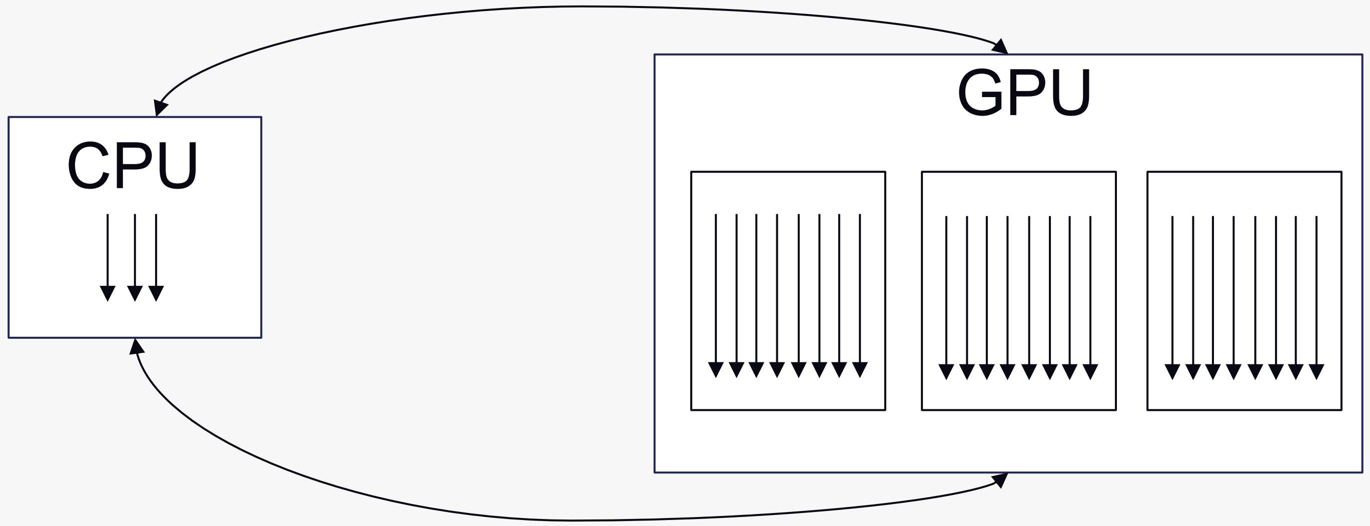


- GPU
  - Optimized for throughput
  - Good for parallel work
  - Relatively low clock frequency

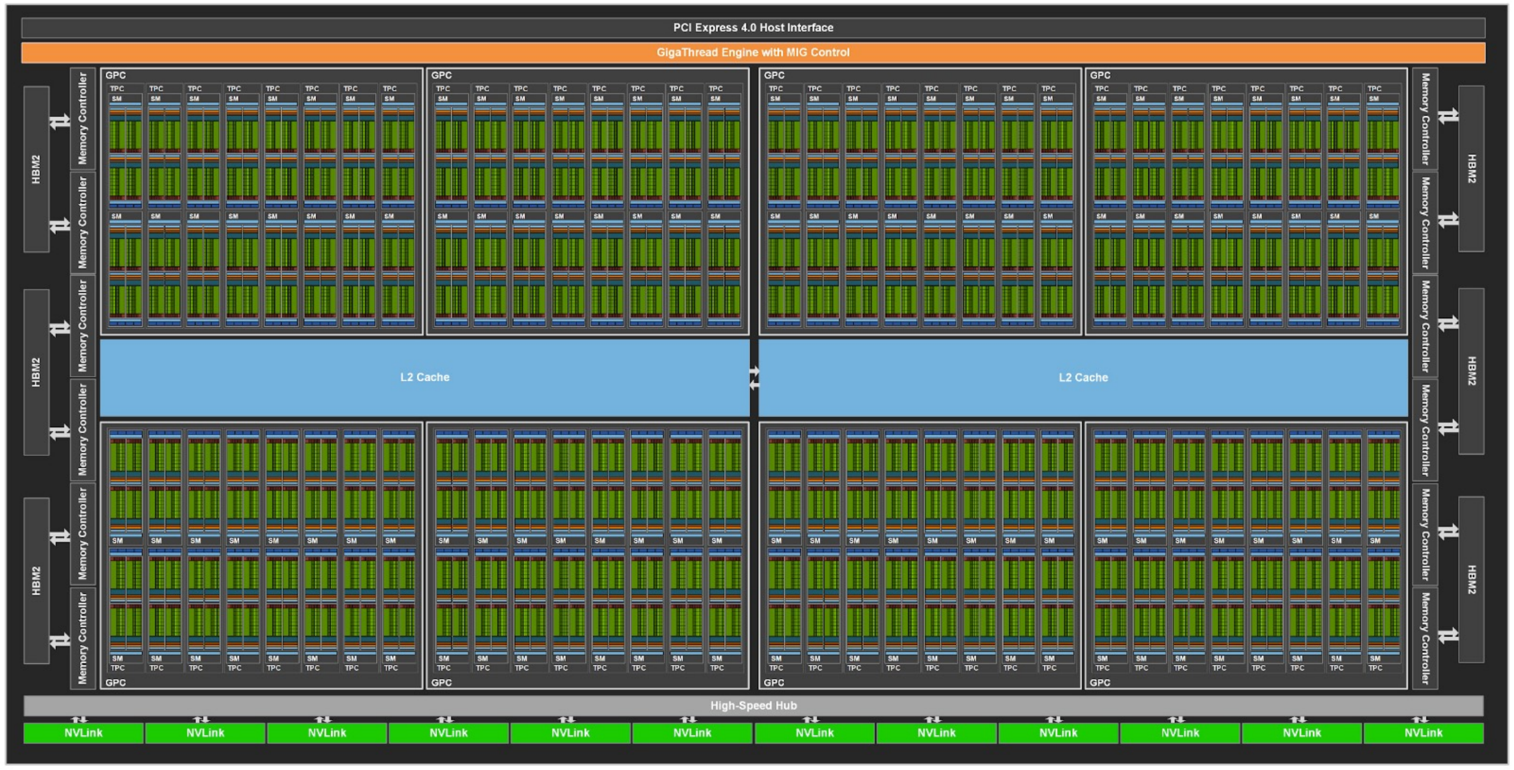
From: https://www.nersc.gov/assets/Uploads/ProgrammingModels.pdf

Reminder about CPU and GPU programming

- CPU+GPU Programming
  - High-level principles
    - Serial work runs on the CPU
    - Parallel work runs on the GPU
    - Minimize transferring data between CPU and GPU

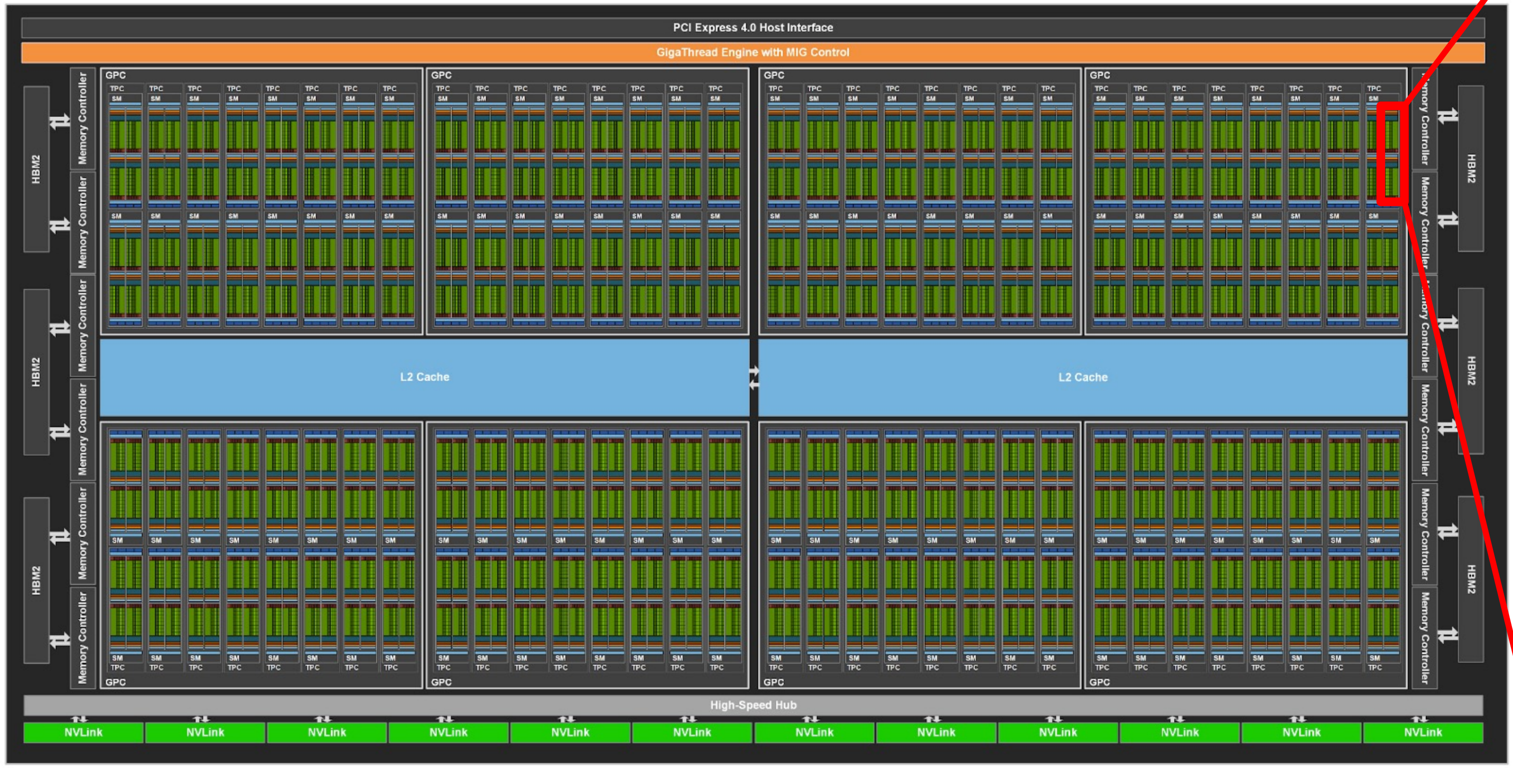


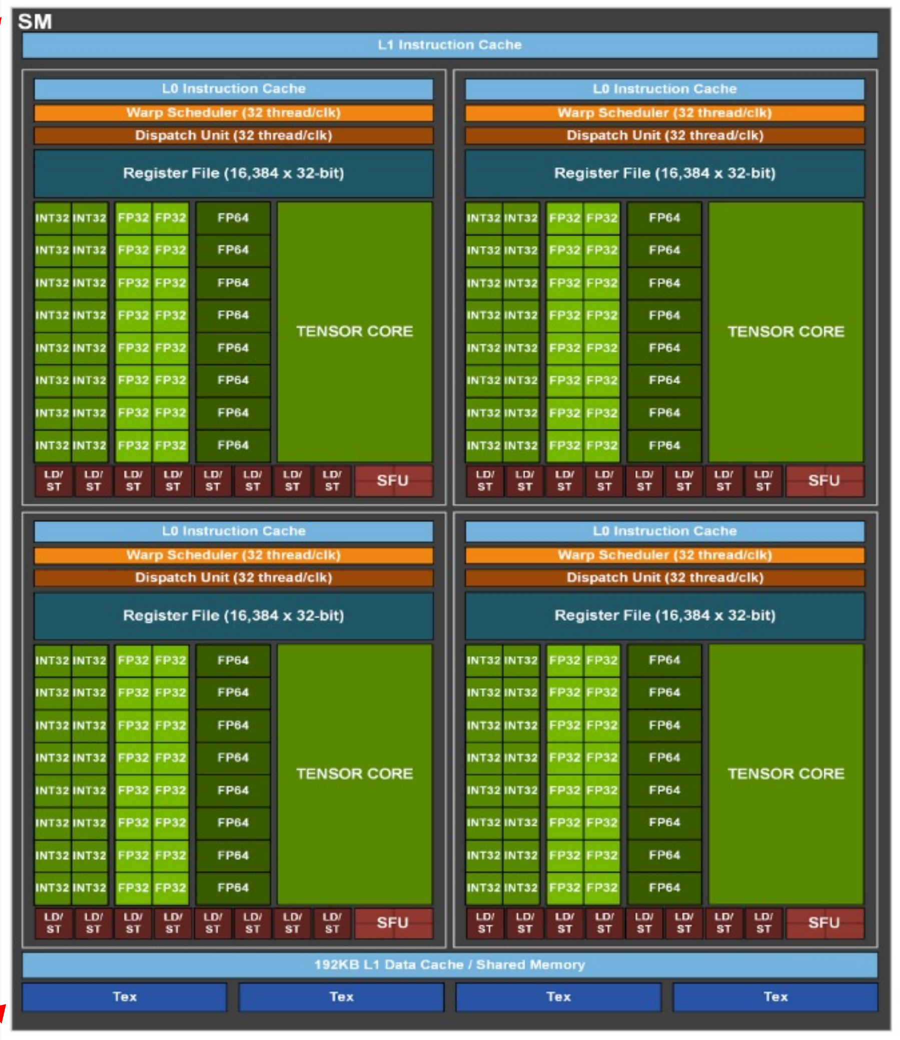
Nvidia GPUs


PCI Express 4.0 Host Interface
GigaThread Engine with MIG Control
GPC
TPC
SM
Memory Controller
HBM2
L2 Cache
High-Speed Hub
NVLink

https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

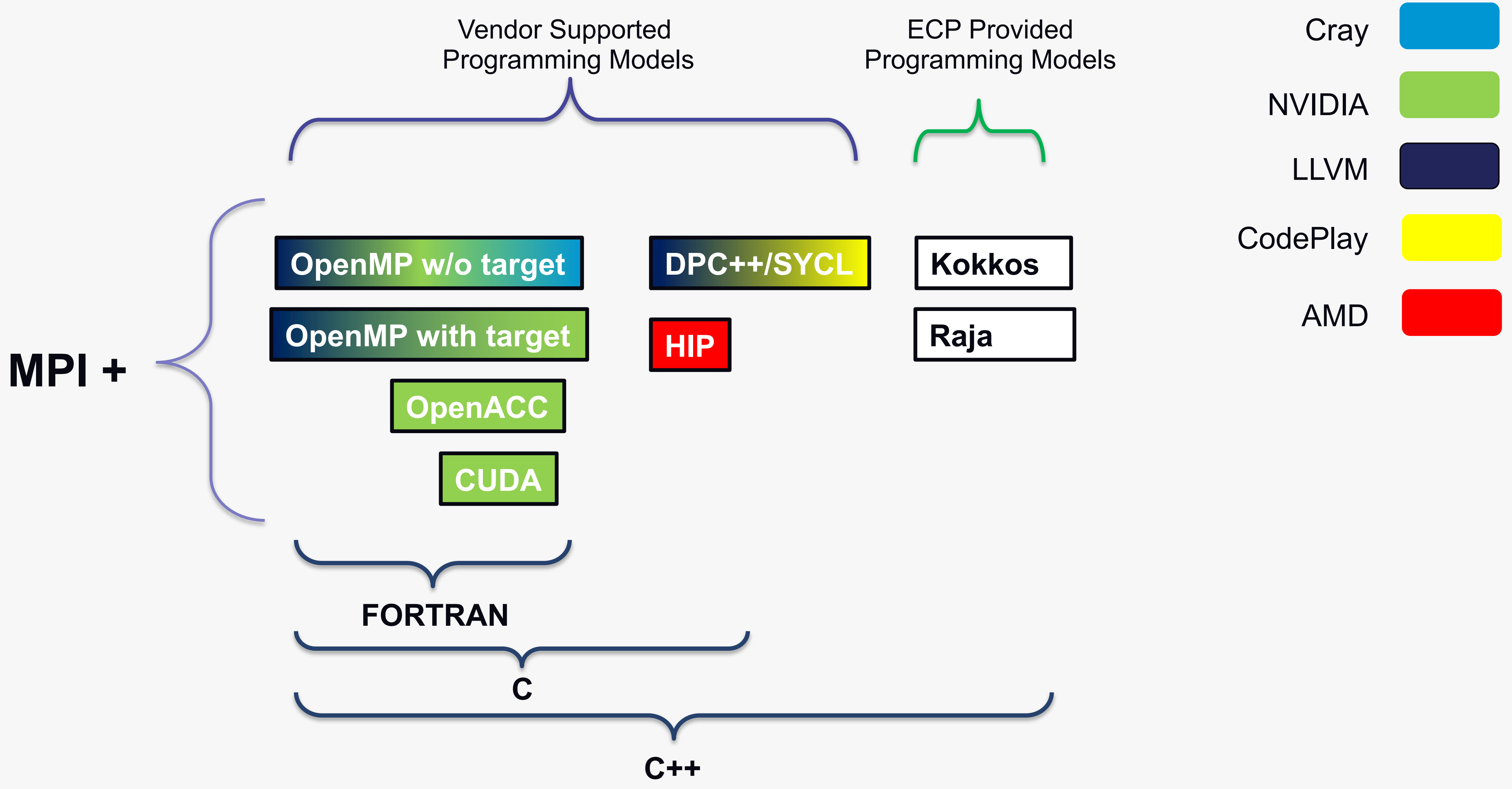
Nvidia GPUs


PCI Express 4.0 Host Interface
GigaThread Engine with MIG Control
GPC
TPC
SM
HBM2
Memory Controller
L2 Cache
High-Speed Hub
NVLink

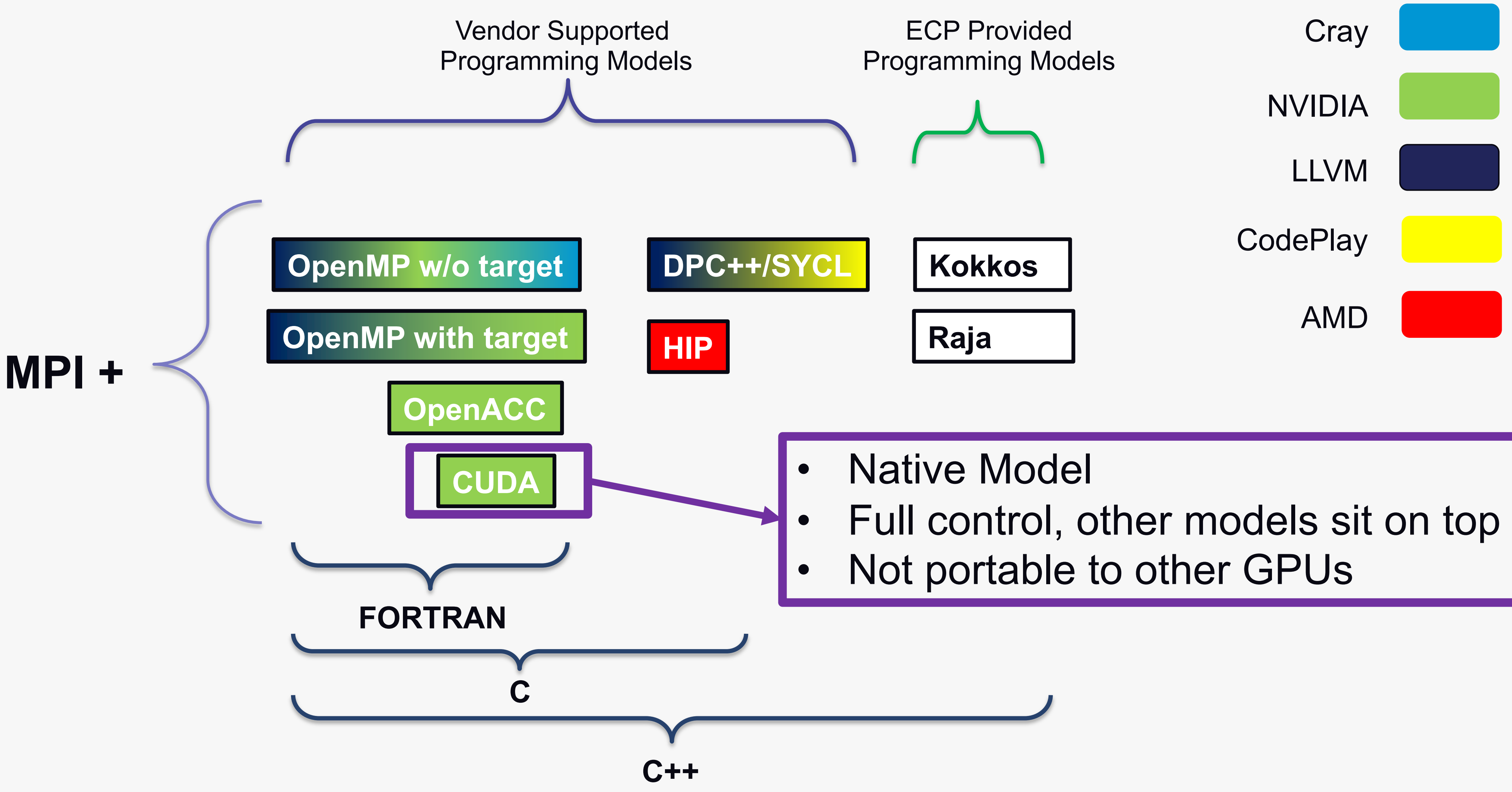

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ ST
SFU
192KB L1 Data Cache / Shared Memory
Tex

https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

Programming Model Landscape on Polaris

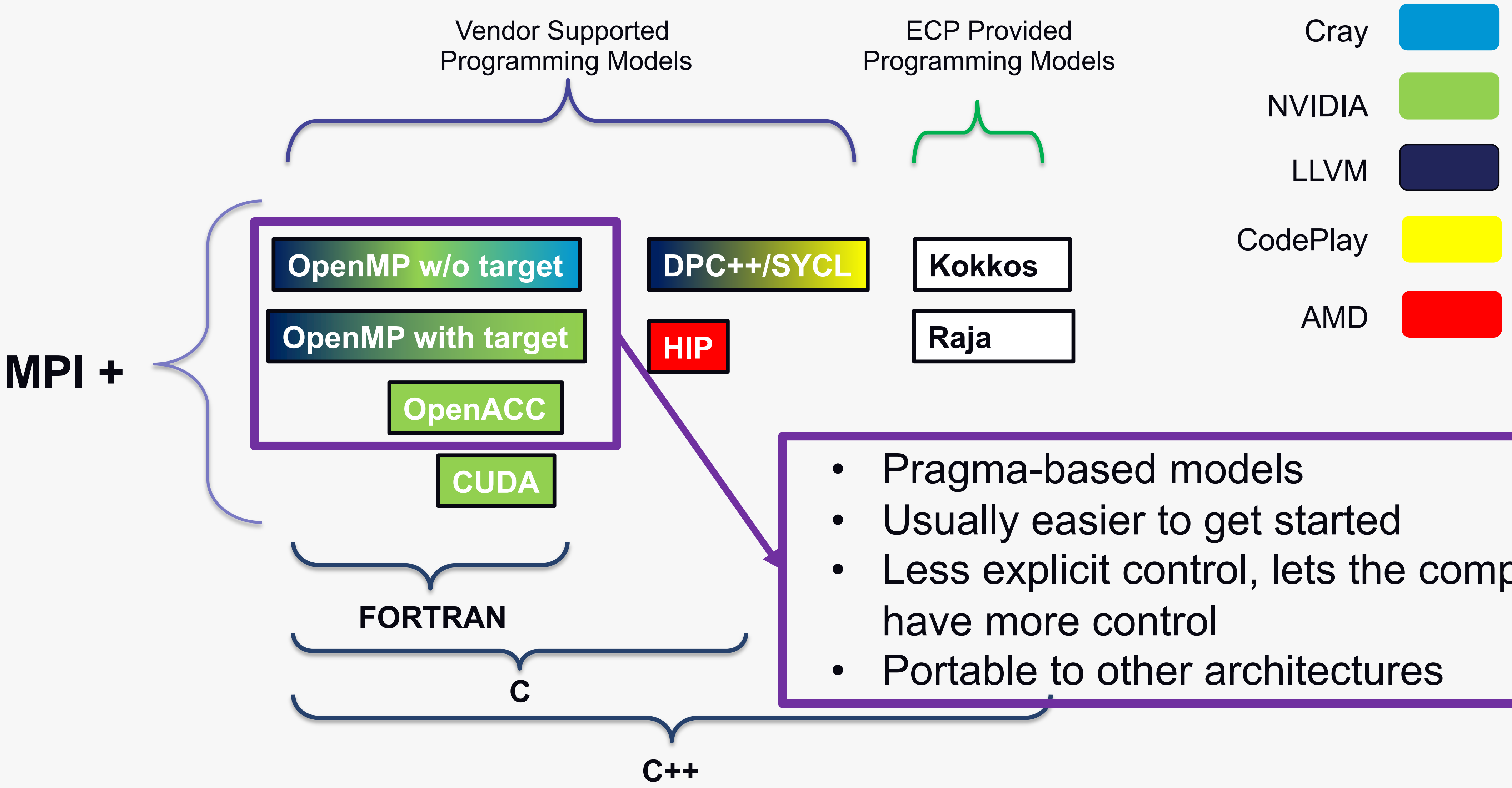


Programming Model Landscape on Polaris



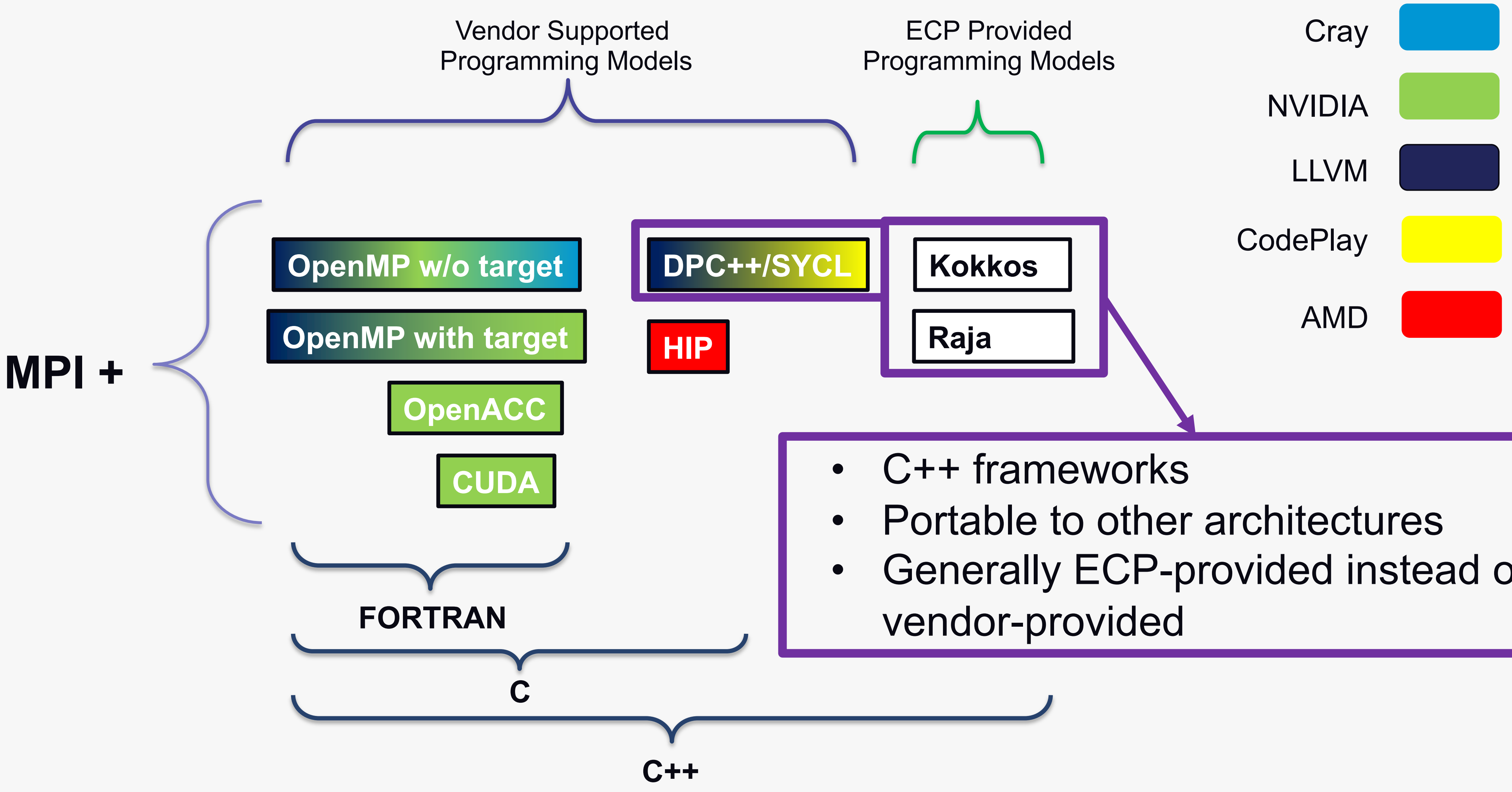
- Native Model
- Full control, other models sit on top of it
- Not portable to other GPUs

Programming Model Landscape on Polaris



- Pragma-based models
- Usually easier to get started
- Less explicit control, lets the compiler have more control
- Portable to other architectures

Programming Model Landscape on Polaris



- C++ frameworks
- Portable to other architectures
- Generally ECP-provided instead of vendor-provided

CUDA

Overview

- NVIDIA's proprietary programming model for the GPU
  - CC8.0 for Polaris's A100 GPUs
- Widely used with a robust toolchain
- Support for C, C++ and Fortran
- Proprietary and not supported on current and upcoming DOE exascale systems

CUDA example

```
__global__ void saxpy(int n, float a, float *x, float *y)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   if (i < n) y[i] = a*x[i] + y[i];
}

void hostFunction()
{
   {...}
   // Perform SAXPY on 1M elements
   saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
   {...}
}
```

CUDA example

```
__global__ void saxpy(int n, float a, float *x, float *y)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   if (i < n) y[i] = a*x[i] + y[i];
}

void hostFunction()
{
   {...}
   // Perform SAXPY on 1M elements
   saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
   {...}
}
```

- Kernel Launch

CUDA example

```
__global__ void saxpy(int n, float a, float *x, float *y)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   if (i < n) y[i] = a*x[i] + y[i];
}

void hostFunction()
{
   {...}
   // Perform SAXPY on 1M elements
   saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
   {...}
}
```

- Kernel Launch
- Iteration Space

CUDA example

```
__global__ void saxpy(int n, float a, float *x, float *y)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   if (i < n) y[i] = a*x[i] + y[i];
}

void hostFunction()
{
   {...}
   // Perform SAXPY on 1M elements
   saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
   {...}
}
```

- Kernel Launch
- Iteration Space
- Kernel Body

CUDA example

```
__global__ void saxpy(int n, float a, float *x, float *y)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   if (i < n) y[i] = a*x[i] + y[i];
}

void hostFunction()
{
   {...}
   // Perform SAXPY on 1M elements
   saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
   {...}
}
```

- Kernel Launch
- Iteration Space
- Kernel Body
- Iteration Handle

Hands-on

- https://github.com/argonne-lcf/sdl_workshop/tree/master/programmingModels/CUDA

- `git clone https://github.com/UoB-HPC/BabelStream.git`

- `git clone https://github.com/ParRes/Kernels.git`

CUDA Resources

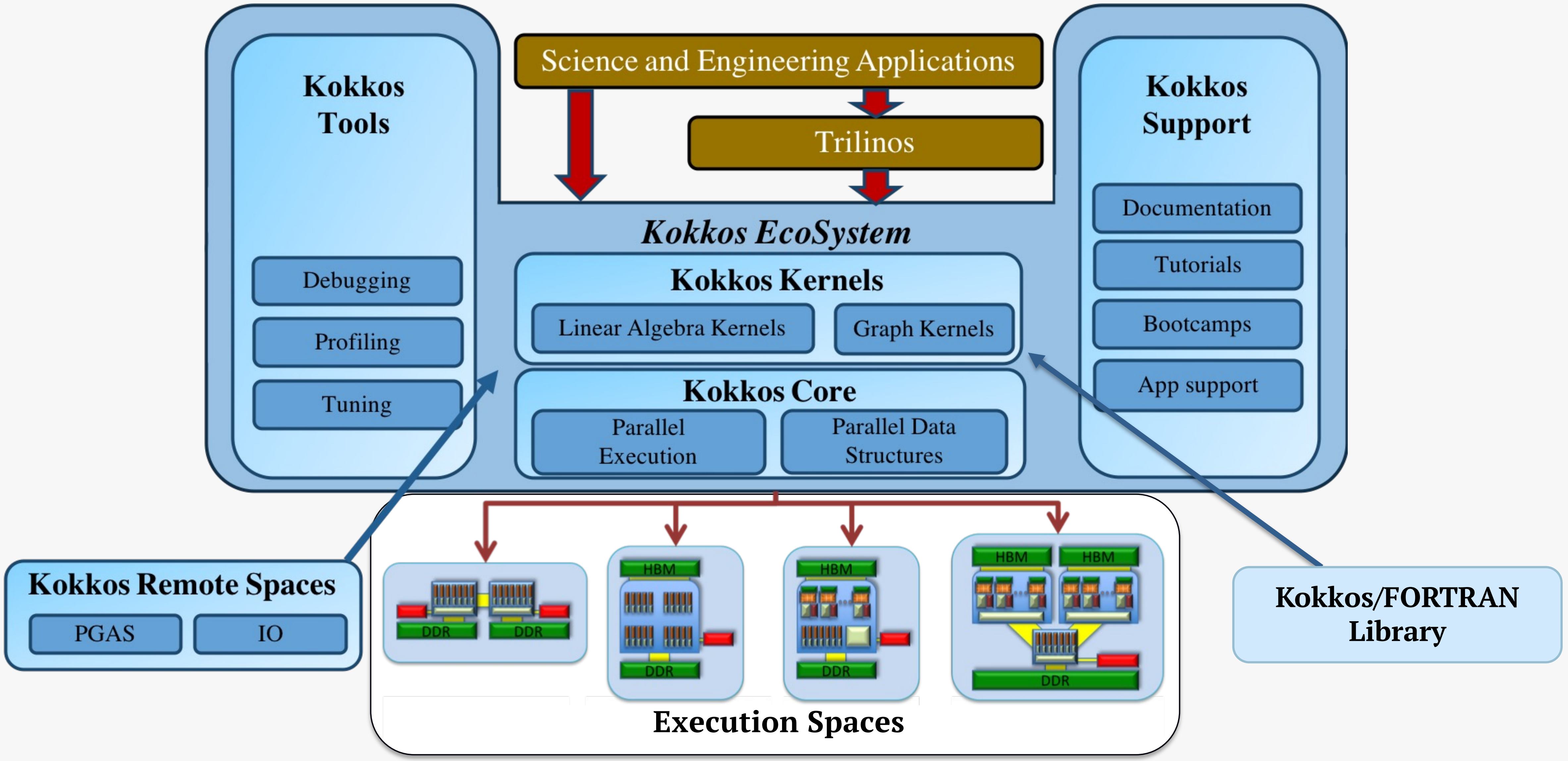
- CUDA resources from NVIDIA
  - https://developer.nvidia.com/cuda-toolkit
- Introduction to CUDA
  - https://developer.nvidia.com/blog/even-easier-introduction-cuda/
- Polaris Getting Started
  - https://www.alcf.anl.gov/support/user-guides/polaris/getting-started/index.html

RAJA/Kokkos

Overview

- Kokkos and RAJA are performance portable header/library solutions
- Built to leverage different backends depending on the target system
  - CUDA, HIP, SYCL, OpenMP, etc
- Utilize C++ to provide a uniform interface to the various backends
- Leverage the vendor developed toolchain while providing portability to the developer

Kokkos EcoSystem



CG Solve: The AXPBY

Simple data parallel loop: Kokkos::parallel_for

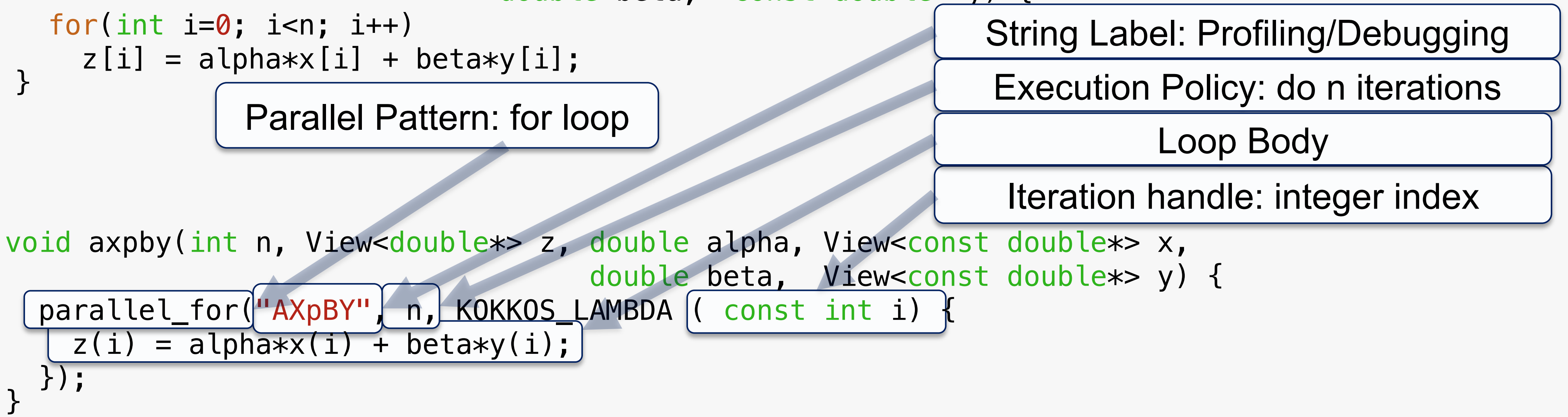
Easy to express in most programming models

Bandwidth bound

Serial Implementation:

Kokkos Implementation:

```
void axpby(int n, double* z, double alpha, const double* x,
                            double beta,  const double* y) {
  for(int i=0; i<n; i++)
    z[i] = alpha*x[i] + beta*y[i];
}
```

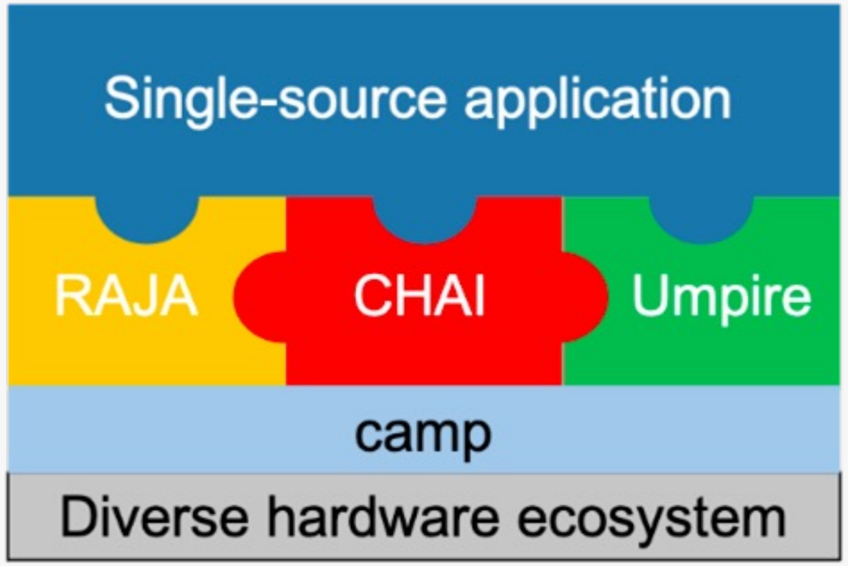

String Label: Profiling/Debugging
Execution Policy: do n iterations
Loop Body
Iteration handle: integer index
Parallel Pattern: for loop

```
void axpby(int n, View<double*> z, double alpha, View<const double*> x,
                                   double beta,  View<const double*> y) {
  parallel_for("AXpBY", n, KOKKOS_LAMBDA ( const int i) {
    z(i) = alpha*x(i) + beta*y(i);
  });
}
```

RAJA Portability Suite

RAJA: C++ kernel execution abstractions

- Enables apps to target various programming model back-ends while maintaining **single-source** app code



Umpire: memory API

- Provides high performance memory operations, such as pool allocations. **Native C++, C, Fortran APIs**

CHAI: C++ array abstractions

- Automates data copies, giving look and feel of unified memory

https://github.com/LLNL/RAJA
https://github.com/LLNL/CHAI
https://github.com/LLNL/Umpire
https://github.com/LLNL/camp

camp: low-level C++ metaprogramming facilities

- Focuses on HPC compiler compatibility

RAJA loop execution core concepts

```
RAJA::forall< EXEC_POLICY >( iteration_space,
   [=] (int i) {
      // loop body
   }
);
```

RAJA::forall method runs loop based on:

– **Execution policy type** (sequential, OpenMP, CUDA, etc.)

RAJA loop execution core concepts

```
RAJA::forall< EXEC_POLICY > ( iteration_space,
  [=] (int i) {
     // loop body
  }
);
```

RAJA::forall template runs loop based on:

– Execution policy type (sequential, OpenMP, CUDA, etc.)

– **Iteration space object** (stride-1 range, list of indices, etc.)

These core concepts are common threads throughout our discussion

```
RAJA::forall< EXEC_POLICY > ( iteration_space,
  [=] (int i) {
     // loop body
  }
);
```

RAJA::forall template runs loop based on:

- Execution policy type (sequential, OpenMP, CUDA, etc.)
- Iteration space object (contiguous range, list of indices, etc.)

Loop body is cast as a C++ lambda expression

- Lambda argument is the loop iteration variable

The programmer must ensure the loop body works with the execution policy; e.g., thread safe

Hands-on

- https://github.com/argonne-lcf/sdl_workshop/tree/master/programmingModels/Kokkos
- https://github.com/argonne-lcf/sdl_workshop/tree/master/programmingModels/RAJA
- `git clone https://github.com/kokkos/kokkos.git`
- `git clone --recursive https://github.com/LLNL/RAJA.git`
- `git clone --recursive https://github.com/LLNL/RAJAPerf.git`

Kokkos & RAJA Resources

- Kokkos
  - Documentation
    - https://kokkos.github.io/kokkos-core-wiki/
  - Tutorials
    - https://github.com/kokkos/kokkos-tutorials
- RAJA
  - Documentation
    - https://raja.readthedocs.io/en/develop/
  - Tutorials
    - https://github.com/LLNL/RAJA-tutorials

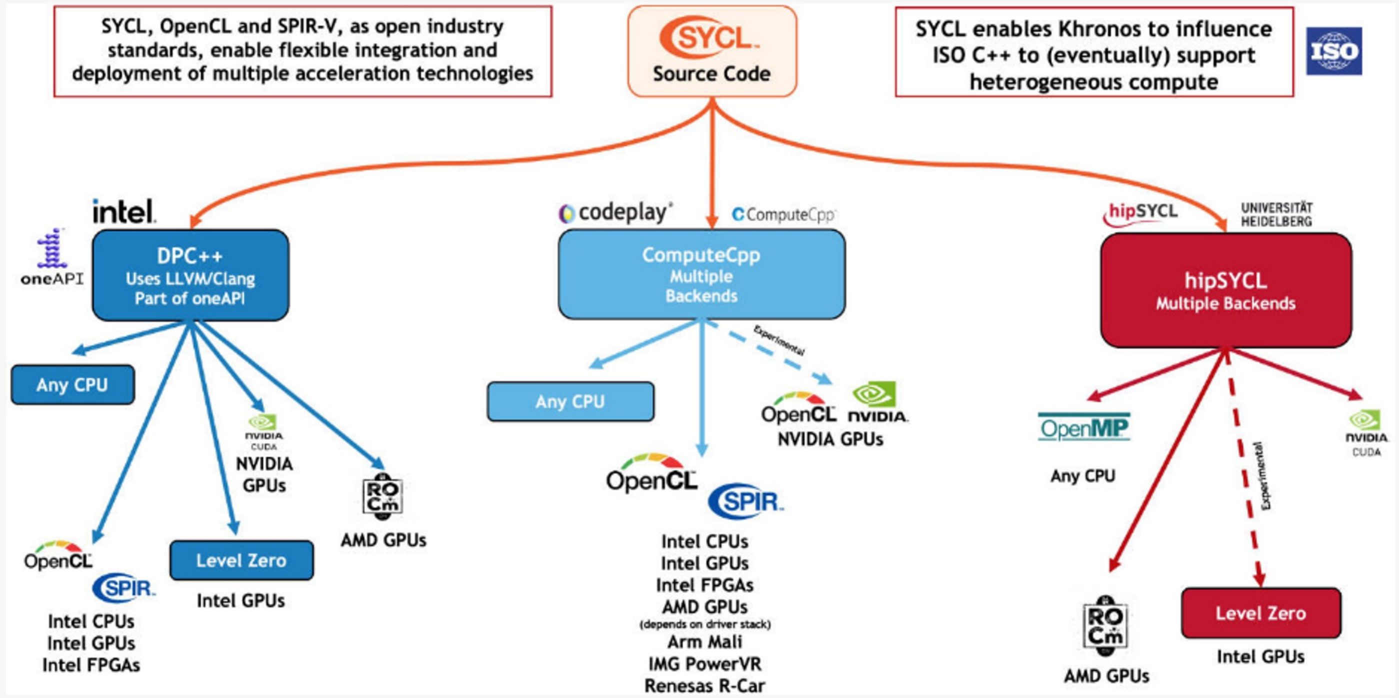
SYCL

Overview

A C++-based programming model for intra-node parallelism

- SYCL is a specification and "not" an implementation, currently compliant to C++17 ISO standards
- Cross-platform abstraction layer, heavily backed by industry
- Open-source, vendor agnostic
- Single-source model

SYCL – Compiler Players


SYCL, OpenCL and SPIR-V, as open industry standards, enable flexible integration and deployment of multiple acceleration technologies
SYCL Source Code
SYCL enables Khronos to influence ISO C++ to (eventually) support heterogeneous compute
intel
oneAPI
DPC++ Uses LLVM/Clang Part of oneAPI
Any CPU
NVIDIA GPUs
AMD GPUs
OpenCL SPIR Intel CPUs Intel GPUs Intel FPGAs
Level Zero Intel GPUs
codeplay
ComputeCpp
ComputeCpp Multiple Backends
Experimental
Any CPU
OpenCL NVIDIA NVIDIA GPUs
OpenCL SPIR Intel CPUs Intel GPUs Intel FPGAs AMD GPUs (depends on driver stack) Arm Mali IMG PowerVR Renesas R-Car
hipSYCL
UNIVERSITÄT HEIDELBERG
hipSYCL Multiple Backends
OpenMP Any CPU
Experimental
NVIDIA CUDA
AMD GPUs
Level Zero Intel GPUs

SYCL: Vector Addition

```
#include <sycl/sycl.hpp>
#include <iostream>

void main() {
  float A[1024], B[1024], C[1024];
  // initialize A, B, C with values on host

  sycl::queue myQueue;

  float* devA = sycl::malloc_device<float>(1024, myQueue);
  float* devB = sycl::malloc_device<float>(1024, myQueue);
  float* devC = sycl::malloc_device<float>(1024, myQueue);

  myQueue.memcpy(devA, A, 1024 * sizeof(float));
  myQueue.memcpy(devB, B, 1024 * sizeof(float));

  myQueue.parallel_for<class vector_add>(range<1> {1024}, [=](id<1> i) {
        devC[i] = devA[i] + devB[i];
     });

  myQueue.memcpy(C, devC, 1024 * sizeof(float));
  for (int i = 0; i < 1024; i++)
    std::cout << “C[“ << i << “] = “ << C[i] << std::endl;
}
```

Step 1: Create SYCL queue to create GPU

Step 2: Allocate device memory

Step 3: (H2D): copy inputs “A” & “B” to GPU

Step 4: (Compute): Run the kernel on device

Step 5: (D2H): Copy result “devC” back to host

SYCL compilers, flags & libraries on Polaris

Module	Compiler flags
llvm-sycl	clang++ -std=c++17 **-fsycl -fsycl-targets=nvptx64-nvidia-cuda -Xsycl-target-backend '--cuda-gpu-arch=sm_80'** sycl_main.cpp
oneMKL	clang++ -std=c++17 -fsycl -fsycl-targets=nvptx64-nvidia-cuda -Xsycl-target-backend '--cuda-gpu-arch=sm_80' **-lonemkl** onemkl_main.cpp
kokkos/3.7.00-sycl	clang++ -std=c++17 -fsycl -fsycl-targets=nvptx64-nvidia-cuda -Xsycl-target-backend '--cuda-gpu-arch=sm_80' **-lkokkoscore** kokkos_main.cpp

Compiling and running SYCL on Nvidia A100 GPUs

```
int main() {
  auto const& gpu_devices = sycl::device::get_devices(sycl::info::device_type::gpu);
  std::cout << "Number of GPUs: " << gpu_devices.size() << std::endl;

  sycl::queue* q{nullptr};
  int devID=0;
  for(const auto& d : gpu_devices) {

    q = new sycl::queue(d);
    sycl::device Dev = q->get_device();

    std::cout << "Found device [" << devID << "]: " << Dev.get_info<sycl::info::device::name>();
    auto global_mem_size = Dev.get_info<sycl::info::device::global_mem_size>();
    auto free_mem_size = Dev.get_info<sycl::ext::intel::info::device::free_memory>();

    std::cout << "Global, Free (in bytes) : " << global_mem_size << ", " << free_mem_size << std::endl;
    devID++;
  } // for-loop

  return 0;
}
```

Step 1: Discovery of 4 Nvidia A100

Step 2: Create a SYCL queue for each device

Step 3: Using SYCL queue, dispatch the work

```
abagusetty@x3004c0s13b0n0 /lus/eagle/projects/UINTAH_aesp/abagusetty $ clang++ -std=c++17 -O3 -fsycl -fsycl-targets=nvptx64-nvidia-cuda -Xsycl-target-backend '--cuda-gpu-arch=sm_80' test_sycl.cpp
clang-16: warning: CUDA version is newer than the latest supported version 11.5 [-Wunknown-cuda-version]
warning: linking module '/soft/compilers/llvm-sycl/llvm/build/lib/clang/16.0.0/../../clc/remangled-l64-signed_char.libspirv-nvptx64--nvidiacl.bc': Linking two modules of different target triples:
'/soft/compilers/llvm-sycl/llvm/build/lib/clang/16.0.0/../../clc/remangled-l64-signed_char.libspirv-nvptx64--nvidiacl.bc' is 'nvptx64-unknown-nvidiacl' whereas 'test_sycl.cpp' is 'nvptx64-nvidia-cuda'
 [-Wlinker-warnings]
1 warning generated.
abagusetty@x3004c0s13b0n0 /lus/eagle/projects/UINTAH_aesp/abagusetty $ ./a.out
Number of GPUs: 4
Found device [0] : NVIDIA A100-SXM4-40GB Global, Free (in bytes): 42505273344, 42070638592
Found device [1] : NVIDIA A100-SXM4-40GB Global, Free (in bytes): 42505273344, 42070638592
Found device [2] : NVIDIA A100-SXM4-40GB Global, Free (in bytes): 42505273344, 42070638592
Found device [3] : NVIDIA A100-SXM4-40GB Global, Free (in bytes): 42505273344, 42070638592
```

OneAPI Math Kernel Library (oneMKL) on Nvidia A100

- OneAPI Math Kernel Library (oneMKL) is designed to allow execution on a wide variety of computational devices: CPUs, GPUs, FPGAs, and other accelerators
- open-source implementation, interface works with multiple devices (backends) uses vendor device-specific libraries underneath
  Note: Apart of device-backend, supports host-CPU interface: Intel MKL, NETLIB

oneMKL APIs	Third-Party Library
BLAS	cuBLAS
Linear Solvers	cuSOLVER
Random Numbers	cuRAND
FFT	cuFFT (in-progress)

Example to run oneMKL GEMM on Nvidia A100

```
#include <sycl/sycl.hpp>
#include <oneapi/mkl.hpp>
…..
 // Initializing the devices queue with the GPU selector
  sycl::queue device_queue(sycl::gpu_selector{});

  // Creating device pointers for matrices (double)
  double* dev_a = sycl::malloc_device<double>(M*N, device_queue);
  double* dev_b = sycl::malloc_device<double>(N*P, device_queue);
  double* dev_c = sycl::malloc_device<double>(M*P, device_queue);

  // Transfer info from CPU to GPU
  device_queue.memcpy(dev_a, host_a, sizeof(double)*M*N);
  device_queue.memcpy(dev_b, host_b, sizeof(double)*N*P);
  device_queue.wait();

  // Launch oneMKL GEMM
  auto event = oneapi::mkl::blas::gemm(device_queue, transB, transA, n,
                      m, k, alpha, dev_b, ldB, dev_a, ldA, beta, dev_c, ldC);
  event.wait();

  sycl::free(dev_a, device_queue);
  sycl::free(dev_b, device_queue);
  sycl::free(dev_c, device_queue);
```

oneMKL header

oneMKL GEMM call

```
clang++ -std=c++17 -fsycl -fsycl-targets=nvptx64-nvidia-cuda -Xsycl-target-backend '--cuda-gpu-arch=sm_80' -lonemkl onemkl_main.cpp
```

Note: cublas code can be ported to SYCL::oneMKL using open-source tool SYCLomatic

(Experimental) Kokkos::SYCL on Nvidia A100

```
-- Built-in Execution Spaces:
--              Device Parallel: Kokkos::Experimental::SYCL
--              Host Parallel: NoTypeDefined
--              Host Serial: SERIAL
--
-- Architectures:
-- Found TPLLIBDL: /usr/include
-- Using internal desul_atomics copy
-- Kokkos Devices: SERIAL;SYCL, Kokkos Backends: SERIAL;SYCL
```

Note: Host OPENMP is not configured yet

Kokkos device-backends	Modules
Kokkos::Experimental::SYCL	module load kokkos/3.7.00-sycl
Kokkos::CUDA	module load kokkos/3.7.00-cuda

Resources

- SYCL documentation: https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html
  - Reference cheat sheet: https://www.khronos.org/files/sycl/sycl-2020-reference-guide.pdf

- Porting from CUDA to SYCL
  - opensource tool
    https://www.intel.com/content/www/us/en/developer/articles/technical/syclomatic-new-cuda-to-sycl-code-migration-tool.html#gs.dxdib9
    https://developer.codeplay.com/products/computecpp/ce/2.11.0/guides/sycl-for-cuda-developers/migrating-from-cuda-to-sycl

- Alternatives to cu** Math libraries: https://github.com/oneapi-src/oneMKL

Hands-on: https://github.com/argonne-lcf/sycltrain

OpenMP Offload

Overview

- Why OpenMP?
  - Open standard for parallel programming with support across vendors
  - OpenMP runs on CPU threads, GPUs, SIMD units
  - C/C++ and Fortran
  - Supported by Intel, Cray, GNU, LLVM compilers and others
  - OpenMP offload will be additionally supported on Aurora, Frontier, Perlmutter
- Four Important high-level features to express parallelism
  - Fork and join thread parallelism
  - SIMD parallelism (added in 4.0)
  - Device Offload parallelism (added in 4.0)
  - Tasking parallelism (added in 3.0)
- Why instead of CUDA?
  - Easy to get started and trivial to parallelize loops
  - The reduction clause simplifies data reduction

CPU OpenMP parallelism

Spawn threads in a thread team

Distributes iterations to the threads

```
#pragma omp parallel for private(x) reduction(+:sum)
 for( int i=0; i<=num_steps; i++){
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
```

GPU OpenMP parallelism

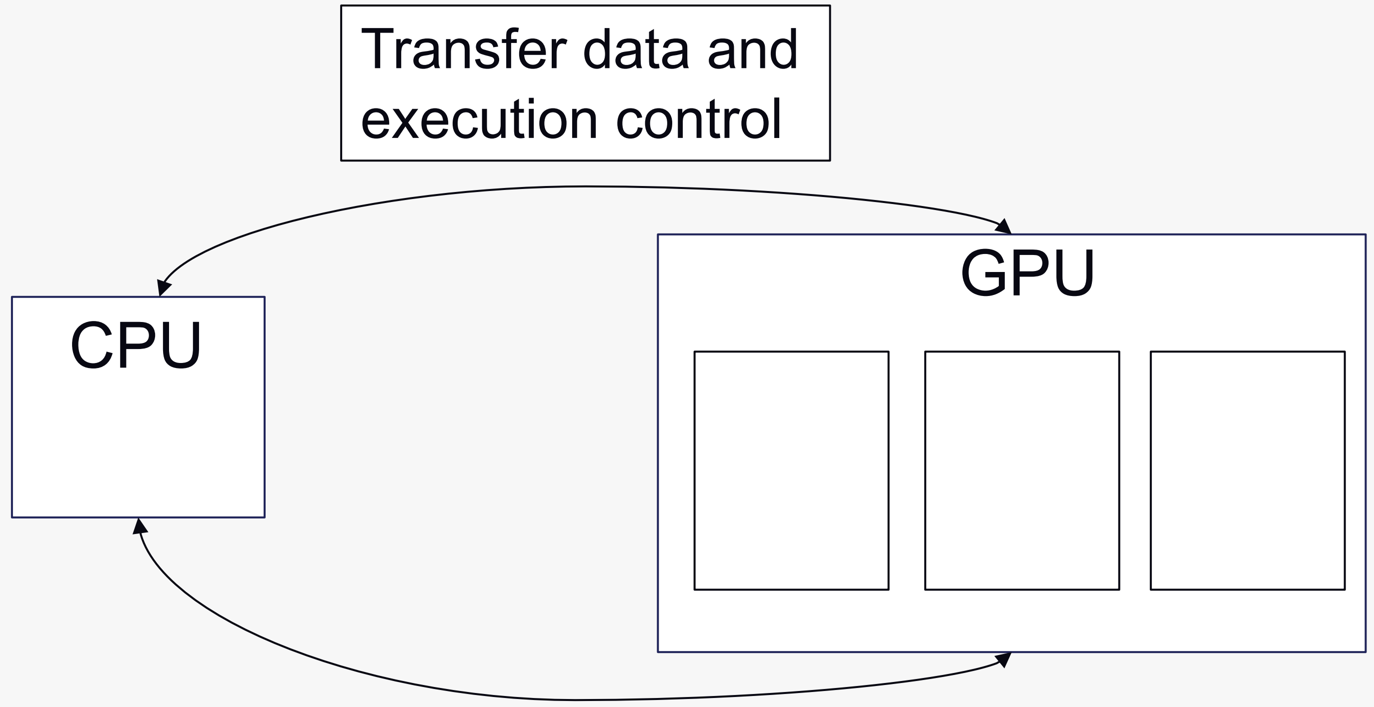
Creates teams of threads in the target device

Distributes iterations to the threads

```
#pragma omp target teams distribute parallel for private(x) reduction(+:sum)
 for( int i=0; i<=num_steps; i++){
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
```

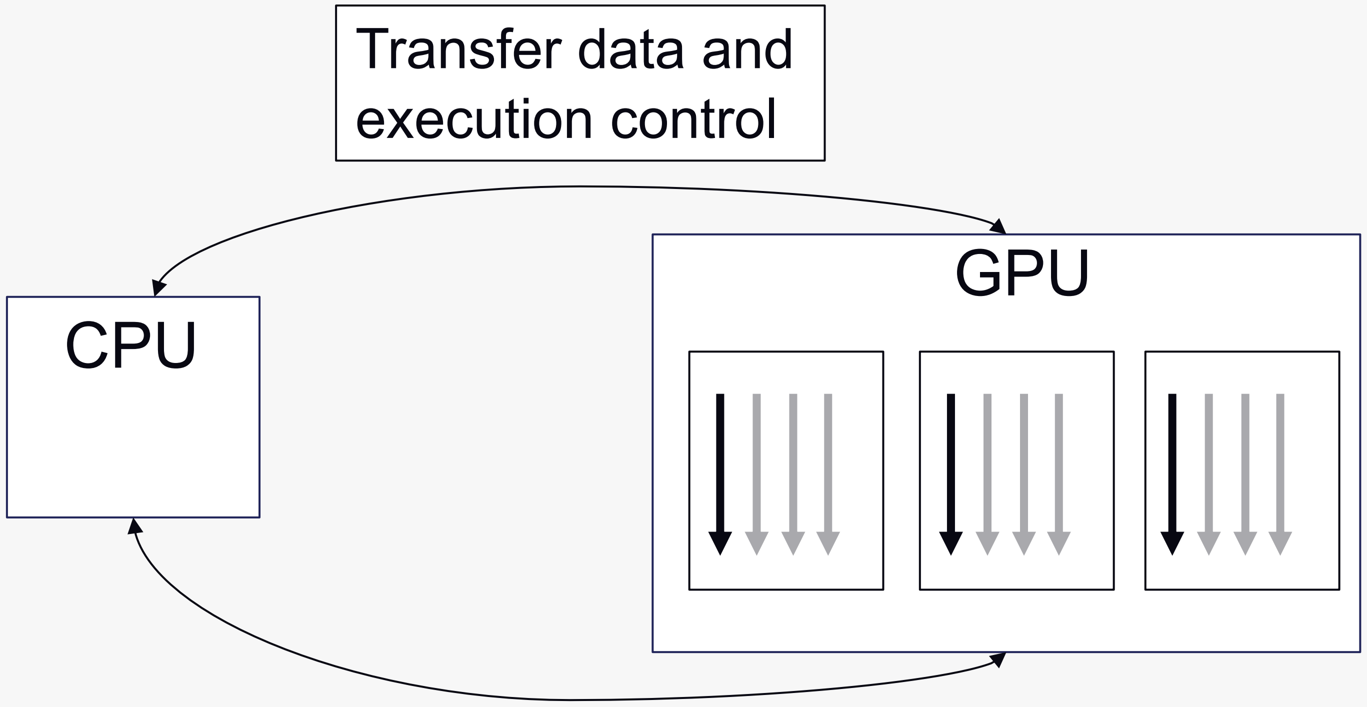
OpenMP Offload Introduction

- **Target construct**: offloads code and data to the device and runs in serial on the device



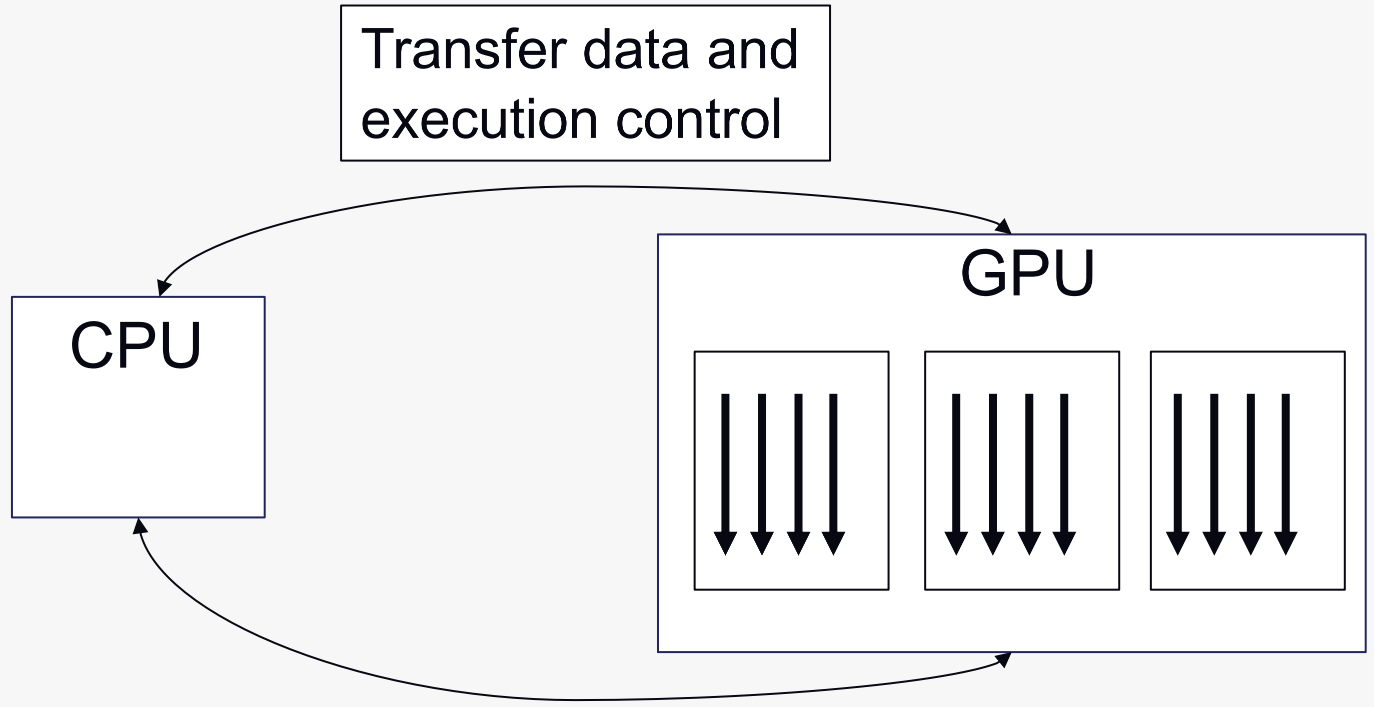
OpenMP Offload Introduction

- **Target construct**: offloads code and data to the device and runs in serial on the device
- **Teams construct**: creates a league of teams, each with one thread, which run concurrently on SMs (Nvidia terminology)



OpenMP Offload Introduction

- **Target construct**: offloads code and data to the device and runs in serial on the device
- **Teams construct**: creates a league of teams, each with one thread, which run concurrently on SMs (Nvidia terminology)
- **Parallel construct**: creates multiple threads in the teams, each which can run concurrently



GPU OpenMP parallelism

Creates teams of threads in the target device

Distributes iterations to the threads

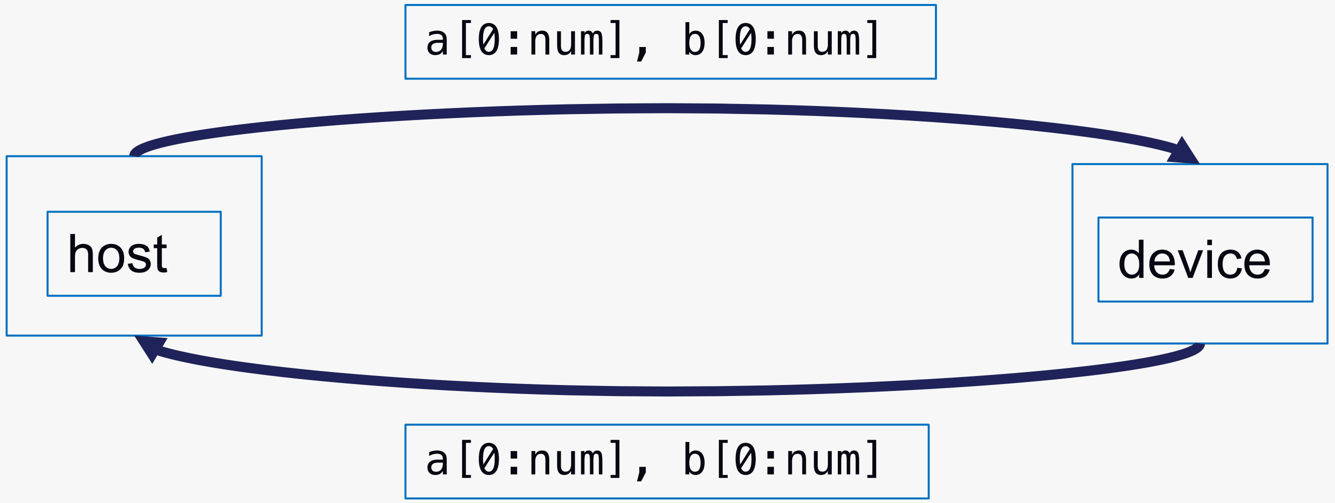
```
#pragma omp target teams distribute parallel for private(x) reduction(+:sum)
 for( int i=0; i<=num_steps; i++){
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
```

OpenMP and data transfer

```
...


#pragma omp target teams distribute parallel for map(tofrom:a[0:num], b[0:num])
     for (size_t j=0; j<num; j++) {
        a[j] = a[j]+scalar*b[j];

      }
…
```

OpenMP and data transfer

```
...


#pragma omp target teams distribute parallel for map(tofrom:a[0:num], b[0:num])
     for (size_t j=0; j<num; j++) {
        a[j] = a[j]+scalar*b[j];

      }
...
```

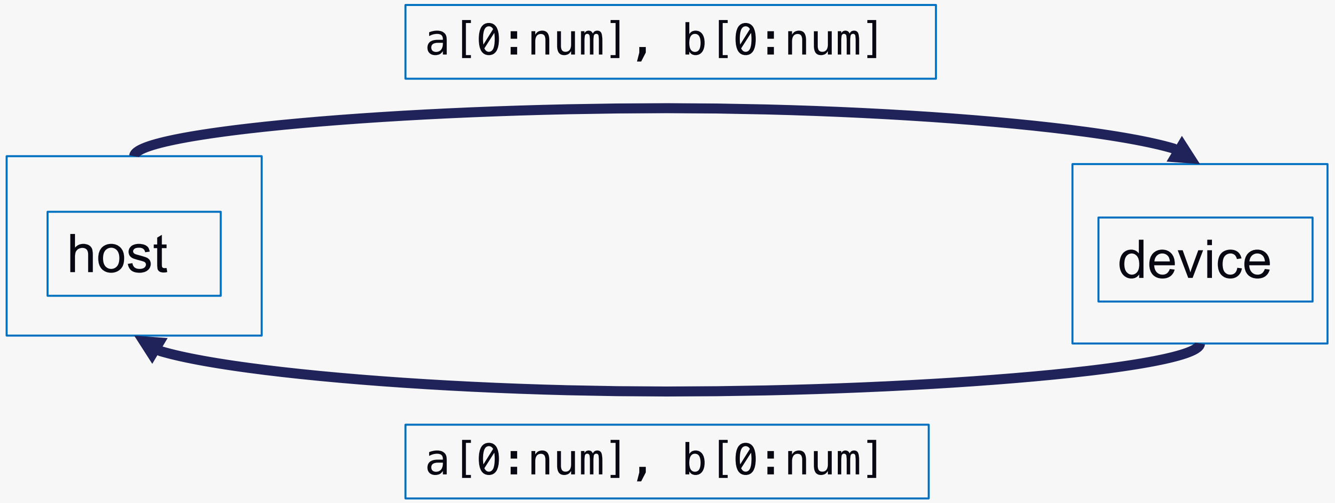


...

- Maps *a* and *b* to and from the device.
- These are shared and accessible by all of the threads on the GPU.

OpenMP and data transfer

```
...

#pragma omp target teams distribute parallel for map(tofrom:a[0:num], b[0:num])
     for (size_t j=0; j<num; j++) {
        a[j] = a[j]+scalar*b[j];

      }
...
```



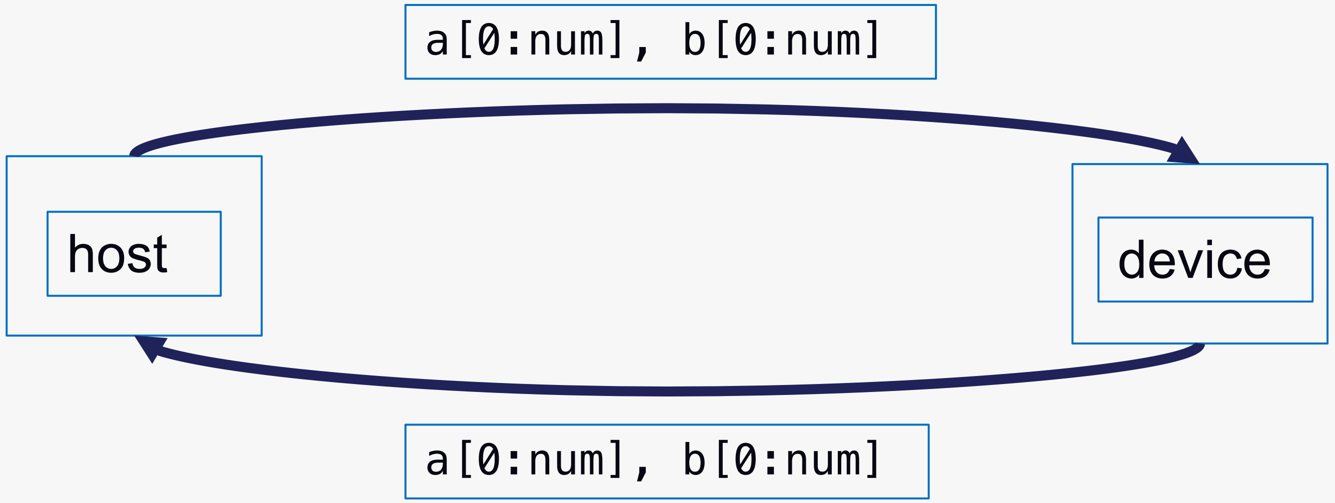
...

By default, scalars *num* and *scalar* are mapped as firstprivate, initialized to what they were on the host, and unique to each thread

OpenMP and data transfer

```
...


#pragma omp target teams distribute parallel for map(tofrom:a[0:num], b[0:num])
     for (size_t j=0; j<num; j++) {
        a[j] = a[j]+scalar*b[j];

      }
...
```

Loop variable *j* is a private scalar per thread on the device



...

OpenMP and data transfer

```
...
#pragma omp target enter data map(to:a[0:num],b[0:num])

#pragma omp target teams distribute parallel for
     for (size_t j=0; j<num; j++) {
        a[j] = a[j]+scalar*b[j];

      }
…
#pragma omp target exit data map(from:a[0:num])
```

OpenMP offload compilers and flags on Polaris

Vendor	Compiler	flags
Nvidia	Cc/CC/ftn (nvc++/ nvfortran under the wrapper)	-mp=gpu -gpu=cc80
LLVM	mpicxx/mpicc (clang++/clang under the wrapper)	-fopenmp -fopenmp-targets=nvptx64-nvidia-cuda
Cray	Cc/CC/ftn	-fopenmp

- Nvidia compilers are in the default environment on Polaris
- LLVM and Nvidia compilers are recommended
- https://www.alcf.anl.gov/support/user-guides/polaris/programming-models/openmp-polaris/index.html

Hands-on

```
$ git clone https://github.com/argonne-lcf/sdl_workshop.git
$ cd sdl_workshop/programmingModels/OpenMP
```

The goal is to show:

1. Offloading code to the device
2. Expressing parallelism
3. Mapping data

Resources

- OpenMP website: https://www.openmp.org/
- Using OpenMP – The Next Step by van der Pas, Stotzer and Terboven, MIT Press, 2017
- Polaris User Guide
  - https://www.alcf.anl.gov/support/user-guides/polaris/programming-models/openmp-polaris/index.html

Questions?

Backup

OpenMP and the loop directive

- Added in OpenMP 5.0
- Similar to “distribute” and “for”, it workshares loop iterations
- It also asserts that loop iterations can be run in any order (are independent)
- Can provide a performance advantage (specifically with the Nvidia compiler, which supports it well)

```
#pragma omp target teams distribute parallel for
     for (size_t j=0; j<num; j++) {
        a[j] = a[j]+scalar*b[j];

      }
```

```
#pragma omp target teams loop
     for (size_t j=0; j<num; j++) {
        a[j] = a[j]+scalar*b[j];

      }
```