Evaluating Application Characteristics for GPU Portability Layer Selection

Charles Leggett
for HEP-CCE

ACAT 2024
March 13 2024

Selecting Languages for Heterogeneous Applications

There are now multiple language selections that can be made for most GPU architectures

- "native"
  - CUDA / HIP / SYCL
- portable
  - Kokkos / Alpaka / OpenMP / std::par
  - HIP and SYCL

	CUDA	Kokkos	SYCL	HIP	OpenMP OpenACC	alpaka	std::par std::exec
NVIDIA GPU				hipcc	nvc++ LLVM, Cray GCC, XL		nvc++
AMD GPU	ZLUDA prototype		openSYCL intel/llvm	hipcc	AOMP LLVM Cray		AdaptiveCpp ROCm stdpar oneapi::dpl
Intel GPU	~~ZLUDA prototype~~		oneAPI intel/llvm	CHIP-SPV: early prototype	Intel OneAPI compiler	advanced prototype	oneapi::dpl
x86 CPU	PGI CUDA for x86 (2010)		oneAPI intel/llvm openSYCL	via HIP-CPU Runtime	nvc++ LLVM, CCE, GCC, XL		
FPGA				via Xilinx Runtime	prototype compilers (OpenArc, Intel, etc.)	protytype via SYCL	

Portability layers have come a long way in the past few years, and can now support most backends.

How do we choose which language or portability solution to use?

Choosing a GPU Language

There is no overall "best" language

- Each language has its own characteristics and associated strengths and weaknesses

A specific application might perform better, or be better suited for one language depending on its characteristics and code

When selecting a portability solution, we first need to:

- Characterize the strengths and weaknesses of the languages
- Identify what characteristics of applications and code structures map best onto the language

HEP/CCE-PPS

Exploring portability solutions for HEP applications by porting to array of GPU languages

FastCaloSim
- ATLAS parameterized LAr calorimeter simulation
- 3 simple kernels,1-D and 2-D jagged arrays, atomics, ROOT dependency

Patatrack
- CMS pixel detector reconstruction
- 40 kernels of varying complexity and lengths (many are short)
  - good test for latency, concurrency, asynchronous execution, memory pools

Wirecell Toolkit
- LArTPC signal simulation
- 3 kernels: rasterization, scatter-add, FFT convolution, atomics

p2r
- CMS "propagate-to-R" track reconstruction in a single kernel

Sherpa and Madgraph
- Leading order Event Generators with auto-generated code

Analyzed applications to find commonalities and associations with GPU language usage

Kernel Runtime and Launch Latency

Launching a native kernel on a GPU can take from a few to several tens of microseconds, depending on

- GPU architecture
- GPU driver
- CPU speed and bus

Some portability layers can considerably increase launch latency

- Kokkos
- OpenMP, especially for first launch

For applications with short kernels, where latency cannot be hidden, this can significantly impact performance

Concurrency and Thread Pools

GPU kernels often work within a larger application framework that can use Multi-Threaded or Multi-Process concurrency.

GPU kernels can also be launched concurrently from different threads/processes.

Kokkos has significant incompatibilities with concurrency and thread pools

- Serial backend has a lock that serializes concurrent calls.
- Threads backend explicitly forbids calls from many external threads.
- Concurrent kernel launches only achievable with CUDA and HIP backends, using architecture specific APIs that limit portability
  - new experimental feature of partitioned execution spaces may fix this

SYCL implementations are inconsistent in their ability to launch concurrent GPU kernels.

- some implementations on the same architecture serialize concurrent kernel calls from different threads
- some architectures have no concurrent SYCL solution

External Library Compatibility

Most HEP applications use a variety of external libraries such as ROOT, Eigen.

- Kokkos headers cannot be directly exposed to nvcc
- Many versions of Eigen are incompatible with nvcc
  - this will percolate up to APIs like Kokkos and Alpaka
- Eigen is currently incompatible with OpenMP offload
- nvc++ (for std::par) cannot compile ROOT yet
  - cannot expose ROOT headers to nvc++

Care must be taken to ensure compatibility with portability layer, or isolate interactions at compile time.

- Can usually compile different parts of application with gcc/clang/portability layer, and link shared objs at runtime
  - may require copying of data between sections so all objects allocated and compiled with "native" compiler in shared lib
- Build rules become considerably more complicated

Data Structure Complexity and Memory Transfers

The prevalent use of complex C++ classes in HEP EDMs maps very poorly to efficient GPU memory usage which work best with simpler SoAs.

APIs like Kokkos, SYCL and Alpaka offer additional support for memory constructs like Kokkos::Views or SYCL buffers to enable portability across GPU and CPU architectures.

- Portability comes with a price - increased overheads for allocations and data transfers when using constructs like Kokkos::Views
  - especially for many small objects
- Pre-allocated objects can always be wrapped instead
- Automatic transfers using USM (for discrete GPUs) are invariably slower than explicit ones
  - std::par can only do USM transfers, by instrumenting constructors at compile time and triggering transfers on page faults

None of the APIs can gracefully represent jagged vectors.

- Heavily used by HEP codes

RNGs, FFTs, Atomics and Portability

Native compilers provide vendor specific hardware implementations

- eg cuFFT, rocRand, atomicAdd

Portability layers either don't (fully) support them, or offer their own portable implementations

- Kokkos has its own versions of RNGs and FFTs
  - more challenging to validate against native implementations
- Alpaka has no interface to vendor libraries
- Intel OneMKL has interfaces to call vendor RNG backends, but not FFT
- nvc++ std::par std::atomic<T> can require C++20
- OpenMP offload of atomics on NVIDIA hardware has wildly different performance with different compilers and compiler versions.

Microbenchmark Framework for Performance Evaluation of OpenMP Target Offloading

Compilation Time

Portability layers can considerably increase compilation time

- may be a concern for large projects

- nvc++ is 2x - 3x slower than gcc
- SYCL icpx bogs down with large kernels
  - sometimes > 1000x slower
- Kokkos adds about 10 - 20% overhead
- OpenMP variable depending on compiler
  - gcc and clang add little overhead
  - nvc++ > 5x slower

Runtime Provisioning

Device and host backends may need to be selected at compile time

- Kokkos needs to be built with very specific hardware architecture identifiers, only one device/host combination allowed
  - `-DKokkos_ARCH_SKX=On -DKokkos_ARCH_TURING75=On`
  - offloaded objects in shared libs must be visible to a single compilation unit
- Alpaka device code of each backend needs to be compiled with the backend-specific compiler.
  - Binaries of all backends can be loaded into the same process, as long as the user code guarantees unique symbols, and the backend technologies themselves don't conflict.
- OneAPI (SYCL) now supports multi-platform binaries, specified at compile time (though only 1 per Intel/AMD/NVIDIA architecture)
  - `-fsycl-targets=spir64,spir64_x86_64,nvidia_gpu_sm_75,amd_gpu_gfx1031`
- Some OpenMP implementations can specify multiple architectures at compile-time
  - Code will select backend at runtime by querying system to see what hardware to target.

Conclusions

When selecting a GPU portability layer, a careful analysis of the application's characteristics is important to achieve the best match to ensure:

- compatibility
- performance
- portability

There is no single solution that is optimal for all situations.

Portability layers continue to evolve, requiring careful monitoring of technologies and re-examination of code bases.

- seeing convergence of increasing feature support and optimized performance
- emerging C++ standards may simplify choices in 5-10 years
  - as may migration to machine learning solutions

fin