Performance Portable GPU Languages

Kokkos, Raja, and SYCL

Bob Robey
Parallel Computing Summer Research Internship
June 18th, 2021
Performance Portability and Productivity meetings

• DOE through their Centers of Excellence realized that the portability of software across architectures was the top concern for applications
  – First meeting held in April, 2016
  – Targeted topic: performance portability
  – Added as a workshop to Supercomputing Conference in 2018
  – productivity (programmer) was soon added to the topic
    • Now known as Performance, Portability, and Productivity (PPP)

• Portability includes CPUs, GPUs and other possibilities
  – not just one of each, but any of them

• Single-source goal
  – Reduces code duplication and maintenance

• An additional benefit is to avoid vendor lock-in
• Encourages both incremental and disruptive innovation
Portability Languages

• Pragma-based languages
  – OpenACC and OpenMP
  – limited by compiler and vendor support
  – good options for Fortran and C
• Native GPU languages
  – HIP and OpenCL
  – reality is two sources, so not a single-source solution
But what we usually mean by Portable GPU languages are the new C++ languages
• Kokkos: Sandia National Laboratory
• Raja: Lawrence Livermore National Laboratory
• SYCL: Khronos Group standard; multiple implementers including Intel
Our portability language heroes have had a lot of cross-fertilization and have a lot of similarities
Lambda construct in C++

• An anonymous function – a characteristic of lambda calculus
  – basically is just a loop body – maps well to the needs of GPU/CPU portability
  – actually can be named and often should be if reused
• Lambda functions are a more powerful generalization of functors or Fortran statement functions
• Lambda constructs have been added to many programming languages in recent years
• Lambda expressions were added to C++ in C++11 and improved in C++17
Example of C++ lambda expression

```cpp
int main() {
    const int N = 100;
    double a[N], b[N], c[N];
    double scalar = 0.5;

    // c, a, and b are all valid scope pointers on the device or host

    // We assign the loop body to the example_lambda variable
    auto example_lambda = [&] (int i) {
        c[i] = a[i] + scalar * b[i];
    };

    for (int i = 0; i < N; i++)
    {
        example_lambda(i);
    }
}
```

Lambda “closure” captures variables from the program context. The capture can be by value [=] or reference [&]

We use the named lambda expression as the loop body. Note that the index set for the loop is specified separately from the body of the loop.
Kokkos – a mature, complete library-based language

- Kokkos is a performance portability language
  - began in 2010 with limited functionality
  - evolved rapidly as hardware architectures diverged
- Has both compute and data constructs
- Compute construct
  - Execution spaces
    - Serial, OpenMP, Cuda, ROCm
  - Based on functors and lambdas
    - parallel_for, parallel_reduce, and parallel_scan
- Data construct
  - Memory spaces
    - HostSpace, CudaSpace
  - multi-dimensional Kokkos arrays basis for capability in C++ 2023 standard
  - allows for switching of multi-dimensional array ordering (polymorphic layout)
    - LayoutLeft and LayoutRight
Building a Kokkos program with CMake

cmake_minimum_required (VERSION 3.10)
project (StreamTriad)

find_package(Kokkos REQUIRED)

add_executable(StreamTriad StreamTriad.cc)

target_link_libraries(StreamTriad Kokkos::kokkos)

• Kokkos has streamlined their CMake support
• Has had some changes, but settling down
• The Kokkos_DIR variable has to point to the Kokkos cmake configuration file
• CMake flags can change the targeted execution platform
  – Kokkos::Serial     -DKokkos_ENABLE_SERIAL=On (default on)
  – Kokkos::Threads   -DKokkos_ENABLE_PTHREAD=On
  – Kokkos::OpenMP    -DKokkos_ENABLE_OPENMP=On
  – Kokkos::Cuda      -DKokkos_ENABLE_CUDA=On
  – Kokkos::HPX       -DKokkos_ENABLE_HPX=On
  – Kokkos::ROCm      -DKokkos_ENABLE_ROCm=On
Example of Kokkos sum reduction

```cpp
#include <Kokkos_Core.hpp>
using namespace std;
int main (int argc, char *argv[]) {
    Kokkos::initialize(argc, argv);{
        Kokkos::Timer timer;

        size_t nsizes = 1000000;
        Kokkos::View<double *> x( "x", nsizes);

        Kokkos::parallel_for(nsizes, KOKKOS_LAMBDA (int i) {
            x[i] = 1.0;
        });

        timer.reset();

        double xsum = 0.0;
        Kokkos::parallel_reduce(nsizes, KOKKOS_LAMBDA (const int i, double &xsum_thread) {
            xsum_thread += x[i];
        }, xsum);

        cout << "Runtime is " << timer.seconds()*1000.0 << " msecs " << endl;
    }
    Kokkos::finalize();
}
```
Example of Kokkos stream triad

```cpp
#include <Kokkos_Core.hpp>
using namespace std;
int main (int argc, char *argv[]) {
  Kokkos::initialize(argc, argv);

  Kokkos::Timer timer;
  double scalar = 3.0;
  size_t nsize = 1000000;
  Kokkos::View<double *> a( "a", nsize);
  Kokkos::View<double *> b( "b", nsize);
  Kokkos::View<double *> c( "c", nsize);

  Kokkos::parallel_for(nsize, KOKKOS_LAMBDA (int i) {
    a[i] = 1.0;
  });
  Kokkos::parallel_for(nsize, KOKKOS_LAMBDA (int i) {
    b[i] = 2.0;
  });
  timer.reset();
  Kokkos::parallel_for(nsize, KOKKOS_LAMBDA (const int i) {
    c[i] = a[i] + scalar * b[i];
  });
  cout << "Runtime is " << timer.seconds() * 1000.0 << " msecs " << endl;
}
Kokkos::finalize();
```
Raja
Raja – an easier language to add incrementally

- Raja provides a performance portable compute capability
  - `parallel_for`, `parallel_reduce`, and `parallel_scan`
- Additional packages provide data constructs
  - Chai – managed memory
  - Umpire – custom allocators and memory pools
- Raja allows some customization of usage for easier migration of existing codes
- Raja is a C++ library based on standard C++11 and lambda expressions
Raja CMake build

cmake_minimum_required (VERSION 3.0)
project (StreamTriad)

find_package(Raja REQUIRED)
find_package(OpenMP REQUIRED)

add_executable(StreamTriad StreamTriad.cc)
target_link_libraries(StreamTriad PUBLIC RAJA)
set_target_properties(StreamTriad PROPERTIES COMPILER_FLAGS ${OpenMP_CXX_FLAGS})
set_target_properties(StreamTriad PROPERTIES LINK_FLAGS "${OpenMP_CXX_FLAGS}")

Raja has good CMake build support
The targeted platform can be selected with CMake flags

-DENABLE_OPENMP=On (default on)
-DENABLE_TARGET_OPENMP=On (default Off)
-DENABLE_CUDA=On (default Off)
-DENABLE_TBB=On (default Off)
#include <chrono>
#include "RAJA/RAJA.hpp"
using namespace std;
int main(int RAJA_UNUSED_ARG(argc), char **RAJA_UNUSED_ARG(argv[]))
{
    chrono::high_resolution_clock::time_point t1, t2;

    const int nsize = 1000000;
    double scalar = 3.0;
    double* a = new double[nsize];
    double* b = new double[nsize];
    double* c = new double[nsize];

    for (int i = 0; i < nsize; i++) {
        a[i] = 1.0;
        b[i] = 2.0;
    }

    t1 = chrono::high_resolution_clock::now();

    RAJA::forall<RAJA::omp_parallel_for_exec>(RAJA::RangeSegment(0,nsize), [=](int i){
        c[i] = a[i] + scalar * b[i];
    });

    t2 = chrono::high_resolution_clock::now();

    double time1 = chrono::duration_cast<chrono::duration<double> >(t2 - t1).count();
    cout << "Runtime is " << time1*1000.0 << " msecs " << endl;
}
Example Raja stream triad

```cpp
#include <chrono>
#include "RAJA/RAJA.hpp"
using namespace std;

int main(int RAJA_UNUSED_ARG(argc), char **RAJA_UNUSED_ARG(argv[]))
{
    chronos::high_resolution_clock::time_point t1, t2;
    const int nsize = 1000000;
    double* x = new double[nsize];
    for (int i = 0; i < nsize; i++) {
        x[i] = 1.0;
    }
    t1 = chronos::high_resolution_clock::now();
    RAJA::ReduceSum< RAJA::omp_reduce, double > xsum_thread(0.0);
    RAJA::forall< RAJA::omp_parallel_for_exec >( RAJA::RangeSegment(0, nsize), [=](int i){
        xsum_thread += x[i];
    });
    xsum = xsum_thread.get();
    t2 = chronos::high_resolution_clock::now();
    cout << "xsum is " << xsum.get() << endl;
    double time1 = chronos::duration_cast< chronos::duration<double> > (t2 - t1).count();
    cout << "Runtime is " << time1 * 1000.0 << " msecs " << endl;
}
```
CHAI – managed memory plugin

• CHAI plugin provides an optional memory model for RAJA

```cpp
chai::ManagedArray<double> x(1000);
```

• Use Chai managed array on device or host and it will be copied there
  
  ```cpp
  // copied to device
  RAJA::forall<RAJA::cuda_exec<16>>(0, 1000, [=] __device__ (int i) {
    x[i] = (double)i * 2.0;
  });
  
  // copied back to host
  RAJA::forall<RAJA::seq_exec>(0, 1000, [=] (int i) {
    x[i] /= 2.0;
  });
  
  // can copy back to host to print
  RAJA::forall<RAJA::seq_exec>(0, 1000, [=] (int i) {
    std::cout << "array[" << i << "] is " << array[i] << std::endl;
  });
  ```
// get resource manager
auto& rm = umpire::ResourceManager::getInstance();
// get allocator
umpire::Allocator host_allocator = rm.getAllocator("HOST");
umpire::Allocator device_allocator = rm.getAllocator("DEVICE");
// allocate data on host and device
int nsize = 1024;
double* host_data =
    static_cast<double*>(host_allocator.allocate(nsize * sizeof(double)));
double* device_data =
    static_cast<double*>(device_allocator.allocate(nsize * sizeof(double)));
// initialize data on host
rm.memset(host_data, 0.0);
// copy data from host to device
rm.copy(device_data, host_data);
// dynamic memory pool
auto pooled_allocator = rm.makeAllocator<umpire::strategy::DynamicPool>("device_pool", device_allocator);
// allocate from pool
double* device_data =
    static_cast<double*>(pooled_allocator.allocate(SIZE * sizeof(double)));
What is SYCL?

• Introduced in 2014 as a single-source C++ language
• Originally developed as a higher-level C++ language on top of OpenCL
• Some implementations now use backends other than OpenCL
• Open collaborative effort with Khronos standard
  – still undergoing significant changes
  – OpenCL is also a Khronos-managed standard
• Largely a research effort until recently
• Intel announced it in Nov 2019 as their primary language for the DOE Aurora system with the release of their first discrete GPU
  – Released OneAPI and DPC++
SYCL Ecosystem

SYCL, OpenCL and SPIR-V, as open industry standards, enable flexible integration and deployment of multiple acceleration technologies.

SYCL enables Khronos to influence ISO C++ to (eventually) support heterogeneous compute.

From Khronos: https://www.khronos.org/blog/sycl-2020-what-do-you-need-to-know
Data Parallel C++ (DPCPP)

- We’ll use DPCPP for the SYCL example
- DPCPP is part of Intel’s OneAPI
- DPCPP implemented extensions to SYCL and proposed some of them as additions to SYCL standard
DPCPP Makefile

CXX = dpcpp
CXXFLAGS = -std=c++17 -fsycl -O3

all: StreamTriad

StreamTriad: StreamTriad.o timer.o
   $(CXX) $(CXXFLAGS) $^ -o $@

clean:
   -rm -f StreamTriad.o StreamTriad

Modifying a makefile to build with DPCPP version of SYCL is easy
• Set C++ compiler to dpcpp
• Add compile flags for SYCL and for C++17
SYCL stream triad example

```cpp
#include <chrono>
#include "CL/sycl.hpp"
namespace Sycl = cl::sycl;
using namespace std;

int main(int argc, char * argv[]) {
    chrono::high_resolution_clock::time_point t1, t2;
    size_t nsize = 10000;
    vector<double> a(nsize,1.0);
    vector<double> b(nsize,2.0);
    vector<double> c(nsize,-1.0);
    t1 = chrono::high_resolution_clock::now();
    Sycl::queue Queue(Sycl::cpu_selector{});
    const double scalar = 3.0;
    Sycl::buffer<double,1> dev_a { a.data(), Sycl::range<1>({a.size()}) };
    Sycl::buffer<double,1> dev_b { b.data(), Sycl::range<1>({b.size()}) };
    Sycl::buffer<double,1> dev_c { c.data(), Sycl::range<1>({c.size()}) };
    Queue.submit([&](Sycl::handler& CommandGroup) {
        auto a = dev_a.get_access<Sycl::access::mode::read>(CommandGroup);
        auto b = dev_b.get_access<Sycl::access::mode::read>(CommandGroup);
        auto c = dev_c.get_access<Sycl::access::mode::write>(CommandGroup);
        CommandGroup.parallel_for<class StreamTriad>(Sycl::range<1>({nsize}),
            [=] (Sycl::id<1> it){
                c[it] = a[it] + scalar * b[it];
            });
    });
    Queue.wait();
    t2 = chrono::high_resolution_clock::now();
    double time1 = chrono::duration_cast<chrono::duration<double>>(t2 - t1).count();
    cout << "Runtime is " << time1*1000.0 << " msecs " << endl;
}
```

Select device (CPU) to run on. Could also be:
- default_selector{}
- gpu_selector{}
- host_selector{}

Create SYCL device buffer and copy data
Start command group and assign handler
Get device array pointers and set access permissions
SYCL lambda
End command group
Wait for commands in queue to complete