Layered DSLs for Portable Manycore Scalability

Daniel Sunderland, Patrick G. Xavier, and H. Carter Edwards

SIAM Parallel Processing
February 21, 2014
SAND2014-1218C (Unlimited Release)
Motivating Problem

We want your awesome domain specific language (YADSL) to implement many useful domain features and work on various manycore architectures.
### Complexity of Naïve Implementation for YADSL

- Large amounts of code duplication
- Patchy rollout of new features to supported architectures
- Significant effort to add either new architectures or features
Can we do better?

Complexity of implementing features should not depend on number of architectures

Complexity of supporting architectures should not depend on number of features
**Desired Implementation for YADSL**

- **YADSL** is implemented on top of a manycore DSL
- **Complexity = Features + Architectures + Specializations**
  - Minimizes code duplication
  - New features automatically work with all supported architectures
  - Specialize algorithms only for performance
Kokkos: C++ Library

Manycore eDSL for Data Parallelism

- Portable to Manycore Architectures
  - Multicore CPU, NVidia GPU, Intel Xeon Phi, ...
  - Allows users to specialize code for any particular architecture
  - Only requires C++1998 standard compliant compiler

- Performant
  - Portable user code performs as well as architecture-specific
    - Thread scalable – not just thread safety (no locking!)

- Design
  - Small, straightforward API
    - Does not compromise portability or performance
Kokkos Abstractions

- Parallel Pattern – parallel_{for, scan, reduce} with functors
- Execution Space – where the code is executed
- Atomics for 32/64 bit types

- Data View – data type must be safe to memcpy
  - Reference counted on the host
  - Memory Space – where the data is stored
  - Memory Trait – how the data will be used
  - Data Layout – how the data is strided
    
    Device polymorphic
    (defaulted to optimal memory layout for the device)

- Containers – MD Array, Vector, Unordered Map, CRS Graph
Kokkos Example: Calculating the L2 Norm

```cpp
// declare vector with runtime length N on Device
View<double*, Device> vec("vec", N);

// fill vector with data
parallel_for(N, fill_vector<Device>(vec));

// calculate the L2 norm and bring it back to the host
// it is possible to just store it on the device to
// avoid the device->host communication
double norm;
parallel_reduce(N, l2norm<Device>(vec), norm);
```

How are the `fill_vector` and `l2norm` functors implemented?
template <class Device>
struct fill_vector {
    // Kokkos requires this typedef
typedef Device device_type;

    View<double*, Device> vec;

    fill_vector(View<double*, Device> v) : vec(v) {}  
    // this function must be const
    
    KOKKOS_INLINE_FUNCTION
    void operator(size_t i) const
    {
        ... // user performs some work
        vec(i) = ...;
    
    }
Kokkos Example: Calculating the L2 Norm

// declare vector with runtime length \(N\) on Device
View<double*, Device> vec("vec", N);

// fill vector with data
parallel_for(N, fill_vector<Device>(vec));

// calculate the L2 norm and bring it back to the host
// it is possible to just store it on the device to
// avoid the device->host communication
double norm;
parallel_reduce(N, l2norm<Device>(vec), norm);

How are the fill_vector and l2norm functors implemented?
template <class Device>
struct l2norm {
    // Kokkos requires these typedefs
    typedef Device device_type;

    // reduction type
    typedef double value_type;

    View<const double*, Device> vec;

    l2norm(View<double*, Device> v) : vec(v) {}  

    ... // on next slide

};
L2 Norm: parallel_reduce functor (cont)

```cpp
template <class Device>
struct l2norm {
    // these member functions must be const
    KOKKOS_INLINE_FUNCTION
    void init(value_type & value) const { value = 0; }
    KOKKOS_INLINE_FUNCTION
    void operator(size_t i, value_type & value) const
    { value += vec(i)*vec(i); }
    KOKKOS_INLINE_FUNCTION
    void join(volatile value_type & dst,
              const volatile value_type & src) const
    { dst += src }
    KOKKOS_INLINE_FUNCTION
    void final(value_type & result) const // optional
    { result = sqrt(result); }
};
```
Stokhos – an embedded DSL for Embedded (Intrusive) Uncertainty Quantification

- Stokhos is implemented on top of Kokkos to leverage its memory layout and architecture portability features
- Stokhos only requires a C++1998 compliant compiler
- The user templates their simulation code on a scalar type (normally double/float)
- Stokhos provides embedded UQ scalar types and overloads a large selection of operators and functions for these types
- The user then compiles their code with a Stokhos UQ type to perform a stochastic Galerkin UQ analysis of the code
The UQ types are stored in Kokkos MD Arrays
- Stored internally as a MD array of one higher rank

Kokkos handles the memory allocation and device portability
- Optimal memory layout for UQ data within the array improving memory access patterns
- Portable kernels leveraging additional fine-grained parallelism provided by extra UQ dimension(s)
Current Kokkos Users

Libraries and applications which use Kokkos

Tpetra
sparse matrix library

MiniMD

LAMMPS
molecular dynamics

MiniFE

Stokhos
embedded UQ

MiniContact

Stokhos
embedded UQ

MiniAero

Albany
ice Sheet Model

MiniDriver
Conclusions

- Kokkos can greatly simplify the complexity of implementing a DSL for manycore architectures.

- Reduces the programming burden by providing a single software interface to various architectures.

- Kokkos is open source and the latest release can be found at http://trilinos.sandia.gov/packages/kokkos
Thanks!

Carter Edwards
Christian Trott
Eric Phipps