CAMP: Compiler Agnostic MetaProgramming, or Portable Performance at Compile Time

Thomas R. W. Scogland, scogland1@llnl.gov, Lawrence Livermore National Laboratory
https://github.com/llnl/camp

What is CAMP?

- A C++ metaprogramming library in the vein of:
  - Metal
  - Brigand
  - Kvasir MPL

- Provides types and templates for template-time calculation, type manipulation and control over overloaded resolution through helps for SFINAE and emulated Concepts
- Type lists: flat lists of types that can be iterated, searched, transformed and more
- Type maps: key-value type container like an associative list with lookup of a value by key
- Algorithms: transform, fold, select, apply and others
- A set of base types for use across RAJA framework and other projects built on those facilities
- Tuple: efficient cross-device tuple implementation
- Resources: a common low-level policy to represent a device or context for use with other RAJA framework projects
- The RAJA framework solution to portable, efficient compile-time primitives across compilers
- RAJA targets C++11, which lacks many common standards, and implementations lack efficient built-ins or even implementations of basic necessities like index_sequence
- CAMP provides these, C++11 compatible and supported on all RAJA compatible and supported.

Q: Why another metaprogramming library for C++?

A: RAJA, the kernel interface, and compiler portability

The kernel interface offers arbitrary loop permutations and a compile-time DSL for assembling and synchronizing nested loop kernels. Its capabilities offer single-source performance portability, but require substantial compile-time calculation to work, so we needed a way to not only maintain compile-time C++, but keep it portable across all RAJA compilers and efficient on compile times.

Usage: Generating test combinations:

```
// List index types to test using TestUtil::list<short, int, unsigned int, unsigned">

using EXEC_POL = KernelPolicy;
using OpenMP POL = OpenMPPolicy;
using CUDA POL = CUDAPolicy;
// Core abstract policy
using EXEC_POL = KernelPolicy:
// Core abstract policy
using EXEC_POL = OpenMPPolicy:
using EXEC_POL = CUDAPolicy:
```

List and map manipulation:

```
// Find index in typelist by type
static assert(camp::index_of(unsigned short, IdxTypedList<typelist>::::type::value == 1, ""));

// index into typelist
static assert(camp::same_as<camp::at<TypedList<typelist>::::type, 10>, unsigned short::::type::value, "");
```

Conclusions and the future

- CAMP has proven to provide a portable and efficient metaprogramming experience for RAJA
- Common types and features like tuple and resources provide both convenience and power across the suite
- There remain things to do:
  - Updating to newer style of alias would reduce verbosity and make it more accessible
  - Faster patterns for some constructs have been found, updates may be able to bring down compile times even further
  - Expanding resource to low level device access
  - Investigating making parts of camp available in the cross-dsl DESU suite as well

Portable (compile-time) performance

- CAMP's first concern is not performance, but keeping RAJA compile times reasonable is important
- Approaches:
  - Aliases over classes wherever possible
  - Support builtins for all compilers, particularly indexing and sequences
  - Significant improvement for compilers regardless of age of standard library
  - Avoiding recursion wherever possible
  - Tuple type is recursion-free, O(1) indexing by offset and type

Usage: Resources for async compute in RAJA

- As a common base component for the RAJA framework, CAMP also provides vocabulary and resource types across the framework, including the new resource types for asynchronous execution and low-level memory management
- These provide runtime functionality and even type-enabled wrappers to make writing generic code easier in the presence of allocators and the need for overlapping actions on a device

CAMP: Compiler Agnostic MetaProgramming, or Portable Performance at Compile Time

Thomas R. W. Scogland, scogland1@llnl.gov, Lawrence Livermore National Laboratory
https://github.com/llnl/camp

Usage:

```
void check_result_and_cleanup(double *p, camp::Resource r);
```

// Completely agnostic to sync/async and backend template<typename Policy>
void test(RAJA::RngRand r, double *data) {
  // Choose the resource type from the policy
  using Res = RAJA::resources::get_resource_policy::type;
  // Create strongly typed resource
  Res r;
  // Type erase for allocation routine, allows fewer template instantiations
  double *test_data = alloc_for_test(r);
  // Run forall with resource deps
  RAJA::forallPolicy(r, rng) [[(int i) {}}];
  // Run kernel dependent on first
  RAJA::forallPolicy(r, rng) [[(int i) {{"logic "}}];
  // sync if necessary, check result, free
  check_result_and_cleanup(test_data, r);
}
```

Library

<table>
<thead>
<tr>
<th>Compiler</th>
<th>CAMP</th>
<th>Metal</th>
<th>Kvasir MPL</th>
<th>Brigand</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>4.93</td>
<td>4.7</td>
<td>4.7</td>
<td>4.8</td>
</tr>
<tr>
<td>clang</td>
<td>3.8</td>
<td>3.8</td>
<td>3.5</td>
<td>3.5</td>
</tr>
<tr>
<td>XL</td>
<td>2019</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Intel</td>
<td>18+</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>nvcc</td>
<td>9.1</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>msvc</td>
<td>2015+</td>
<td>2017+</td>
<td>2015+</td>
<td>2017+</td>
</tr>
</tbody>
</table>

Table: Compiler compatibility comparison

<table>
<thead>
<tr>
<th>Speedup (Tsim/Tcomp)</th>
<th>Sequential</th>
<th>OpenMP</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>1.042</td>
<td>1.166</td>
<td></td>
</tr>
</tbody>
</table>