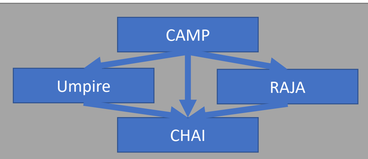
CAMP: Compiler Agnostic MetaProgramming, or Portable Performance at Compile Time

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 overload resolution through helpers 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 type 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 primitives, and implementations lack efficient builtins or even implementations of basic necessities like index_sequence
  - CAMP provides these, C++11 compatible and supported on all RAJA framework compilers

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 potability, 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.

Baseline seq variant

```
for ( int i = 0; i < ni; ++i ) {
   for ( int j = 0; j < nj; ++j ) {
      C[j + i*nj] *= beta;
      double dot = 0.0;
      for ( int k = 0; k < nk; ++k ) {
         dot += alpha * A[k + i*nk] * B[j + k*nj];
      }
      C[j + i*nj] = dot;
   }
}
```

Similar for OpenMP, CUDA, etc.

Note both loops "collapsed"

RAJA variants

```
kernel< EXEC_POL >( make_tuple( RangeSegment(0, ni), RangeSegment(0, nj) ),
   [=] (int i, int j) {
      C[j + i*nj] *= beta;
      double dot = 0.0;
      for ( int k = 0; k < nk; ++k ) {
         dot += alpha * A[k + i*nk] * B[j + k*nj];
      }
      C[j + i*nj] = dot;
   }
);
```

Sequential policy

```
using EXEC_POL =
  KernelPolicy<
    For<0, loop_exec,
      For<1, loop_exec,
        Lambda<0>,
      >
    >
  >;
```

OpenMP policy

```
using EXEC_POL =
  KernelPolicy<
    Collapse<omp_parallel_collapse_exec,
             ArgList<0, 1>,
        Lambda<0>,
    >
  >;
```

CUDA policy

```
using EXEC_POL =
  KernelPolicy<
    CudaKernelAsync<
      For<0, cuda_block_x_loop,
        For<1, cuda_thread_x_loop,
          Lambda<0>,
        >
      >
    >
  >;
```

Programming Model	Sequential	OpenMP	CUDA
Speedup (T_{Base} / T_{RAJA})	1.000	1.042	1.166

Compiler compatibility comparison

Library/ compiler	CAMP	Metal	Kvasir MPL	Brigand
gcc	4.9.3+	4.7+	4.7+	4.8+
clang	3.8+	3.8+	3.5+	3.5+
XL	2019+	?	?	?
Intel	18+	?	?	?
nvcc	9.1+	?	?	?
msvc	2015+	2017+	2015+	2017+

Usage:

Generating test combinations:

```
// List index types to test
using IdxTypeList = camp::list<short,
                               unsigned short,
                               int,
                               unsigned>;
// Resource types to test
using OMPResourceList = camp::list<camp::resource::OpenMP>;
// policies to test
using OMPForallExecPols = camp::list<RAJA::omp_parallel_for_exec,
                                     RAJA::omp_parallel_for_simd_exec>;
// Complete cross-product of lists to instantiate full tests
using test_types =
    camp::cartesian_product<IdxTypeList,
                            OMPResourceList,
                            OMPForallExecPols>;

// Passing into google test
INSTANTIATE_TYPED_TEST_SUITE_P(OMP,
                               ForallRangeSegmentTest, test_types);
```

List and map manipulation:

```
// find index in typelist by type
static_assert(camp::index_of<unsigned short,
                             IdxTypeList>::type::value == 1, "");

// index into typelist
static_assert(camp::is_same<camp::at_t<IdxTypeList, 1>,
                            unsigned short>::value, "");

// associative list indexing by type
using pairs = camp::list<camp::list<int, int_action>,
                         camp::list<char, char_action>>;
static_assert(camp::is_same<camp::at_key<pairs, char>,
                            char_action>::value, "");
```

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 on the system
  - 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-erased wrappers to make writing generic code easier in the presence of allocators and the need for overlapping actions on a device

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

// Completely agnostic to sync/async and backend
template <typename Policy>
void test(RAJA::RangeSegment rng, double *data){
  // Deduce 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::forall<Policy>(r, rng, [](int i) {
    // logic
  });
  // Run kernel dependent on first
  RAJA::forall<Policy>(r, rng, [](int i) {/* logic */});
  // sync if necessary, check result, free
  check_result_and_cleanup(test_data, r);
}

void run_tests(double *data){
  // Run on OpenMP backend
  test<omp_parallel_for_exec>(data);
  // Run on cuda backend synchronously
  test<cuda_exec<256>>(data);
  // Run on cuda backend asynchronously
  test<cuda_exec_async<256>>(data);
}
```

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
  - Investigate making parts of camp available in the cross-lab DESUL suite as well

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC: LLNL-POST-813930