

Towards Performance Portable GPU Programming with RAJA

[Extended Abstract]

Arpith C. Jacob, Samuel F. Antao, Hyojin Sung, Alexandre E. Eichenberger, Carlo Bertolli,
Gheorghe-Teodor Bercea, Tong Chen, Zehra Sura, Georgios Rokos, Kevin O'Brien

IBM T.J. Watson Research Center
1101 Kitchawan Rd.
Yorktown Heights NY, U.S.A.

{acjacob,sfantao,hsung,alexe,cbertol,gbercea,chentong,zsura,grokos,caomhin}@us.ibm.com

ABSTRACT

High-performance machines of today are already heterogeneous in nature with traditional multicores and accelerators such as GPUs and Xeon PHIs. It is a challenge to program these machines in a performance portable way with code that is maintainable across large production applications. RAJA is a recently introduced C/C++ programming approach targeting fine-grained parallelism in loops that is intended to be platform and vendor agnostic. Developed at the Lawrence Livermore National Laboratory, RAJA has shown promise on traditional multicores for ASC hydrodynamics codes.

In this work we extend RAJA to GPUs using the new OpenMP 4.1 standard. We document some of the unique issues when RAJA is used to program a heterogeneous system with distributed code and data spaces, and the efficiency challenges with RAJA when using the offload model of OpenMP 4.1. We then introduce several execution policies for RAJA that exploit the high performance features of OpenMP. We show that when RAJA is used to offload a loop with large amounts of work it matches performance of programs written with low-level OpenMP 4.1 constructs. When a loop has limited work the overhead of offloading dominates and the approach of RAJA is a limiter for performance. We are investigating an asynchronous execution policy that targets such cases, which may help hide the cost of offloading.

Categories and Subject Descriptors

D.1.3 [Programming Techniques]: Concurrent Programming – *parallel programming*.

General Terms

Performance.

Keywords

Performance-portable, GPU programming, RAJA, OpenMP, LCALS.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Portability Among HPC Architectures for Scientific Applications,
November 15, 2015, Austin, TX, USA.
Copyright 2015 HPCPort.

1. MOTIVATION

As HPC systems incorporate diverse accelerators such as GPUs and Xeon PHIs™ with traditional CPUs it becomes challenging for application developers to achieve performance portability and maintainability of their codebases. In particular, to exploit fine-grained parallelism in loops on these and other emerging architectures developers must express parallelism in varied computing paradigms such as Vector, Single Instruction Multiple Data (SIMD), and Single Instruction Multiple Thread (SIMT).

RAJA [1] is a C/C++ programming approach developed at the Lawrence Livermore National Laboratory that allows an application developer to express fine-grained parallelism found in loops in a manner that is agnostic to the computing paradigm. Platform and vendor specific optimizations along with low-level programming models can be hidden behind the RAJA framework. Once an application is written in the RAJA style of programming it can be ported to a variety of platforms with minimum code disruption. Initial reports on using RAJA for ASC hydrodynamics codes show promising results [1].

To achieve these goals RAJA requires compiler support for recently introduced C++ language features and integration with emerging programming models. In this work we investigate how RAJA may be used to exploit systems with GPUs using the OpenMP® 4.1 [5] programming model. We use a Clang®/LLVM® based OpenMP compiler and runtime [2] that we are developing to accelerate RAJA programs on GPUs. We describe ongoing work to extend the RAJA framework to fully exploit GPU devices and the compiler/runtime functionality added to support RAJA.

Our work is done in the context of the Livermore Compiler Analysis Loop Suite (LCALS) [3], which contains kernels written in OpenMP and RAJA for execution on CPUs. We first port LCALS to the GPU using OpenMP 4.1 and RAJA and then compare the performance of the two programming styles on an NVIDIA® Kepler K40m GPU.

We find that RAJA compares well with programs written in pure OpenMP 4.1 when the amount of work per offloaded loop is substantial. For a kernel with sequences of loops that have limited work in their body, the overhead of offloading them individually with RAJA dominates, whereas in pure OpenMP they can be placed in a single offload region. We are investigating techniques to minimize this overhead.

This study is broadly relevant to other template based programming approaches such as KOKKOS [6].

2. THE RAJA APPROACH

In this work we focus on RAJA as used in C++. RAJA uses C++ lambda functions and templates to separate loop traversal from a loop body. The mechanism enabling various loop traversal schemes such as sequential, SIMD, or in parallel across multiple cores is hidden within RAJA and is invisible to the application developer.

Figure 1 shows how a user can convert a loop to RAJA. The loop body is expressed as a lambda function parameterized by the iteration variable. The lambda is said to *capture-by-reference* variables used within it, which are visible at the scope of the call site. The RAJA template *forall* is passed the loop body, its iteration bounds and an execution policy.

```
C++ style loop
for (int i=0; i<len; i++) {
    bvc[i] = cls * (compression[i] + 1.0);
}

RAJA loop
typedef simd_exec exec_policy;
forall<exec_policy>(0, len, [&](int i) {
    bvc[i] = cls * (compression[i] + 1.0);
});
```

Figure 1: A RAJA loop corresponding to a C/C++ loop.

The execution policy is defined in RAJA using templates. Figure 2 shows two templates for serial SIMD and parallel execution on a multicore. RAJA hides vendor specific compiler directives and lower level programming models such as OpenMP, OpenACC[®] or CUDA[®], only exposing an abstract execution intent to the user.

```
template <typename LOOP_BODY> inline void forall(
simd_exec, int begin, int end, LOOP_BODY loop_body) {
    #pragma clang loop vectorize(enable)
    for (int i = begin; i < end; i++) {
        loop_body(i);
    }
}

template <typename LOOP_BODY> inline void forall(
omp_parallel_exec, int begin, int end, LOOP_BODY loop_body)
{
    #pragma omp parallel for
    for (int i = begin; i < end; i++) {
        loop_body(i);
    }
}

template <typename EXEC_T, typename LOOP_BODY> inline
void forall (int begin, int end, LOOP_BODY loop_body) {
    forall (EXEC_T(), begin, end, loop_body);
}
```

Figure 2: RAJA templates to specify serial execution on a SIMD unit using the LLVM compiler or multicore execution with OpenMP.

3. OPENMP ACCELERATOR MODEL

The OpenMP 4.0 [5] accelerator model defines a default host device and one or more *target* devices that are selectively invoked by the programmer. A device is associated with an independent data environment, i.e., a collection of variables accessible to it. On encountering a structured block annotated by a *target* pragma the host offloads the associated code and data to a device for execution. Variables are communicated explicitly, or *mapped*,

from the host's data environment to that of the target using a *map* clause associated with the target construct.

A device starts execution with one thread but may fork additional threads on encountering parallel constructs. To efficiently exploit massively threaded devices such as GPUs, where threads are grouped into blocks, the *teams* construct allows the creation of a group of independent teams each with a single thread. A *distribute* construct partitions iterations of its associated loop across teams. When a team encounters a parallel construct it may create additional threads on the device that can be used for worksharing.

A map clause specifies variables that are to be made accessible by the target device. Arrays are mapped using the syntax *array[base:length]*. Motion types *alloc*, *to*, *from*, and *tofrom* of a map clause specify that a variable is to be allocated, initialized, and copied back to the host after device execution. If a variable accessed in a target region is not listed in its map clause and is not present in the device's data environment, a default motion type of *tofrom* is assumed.

```
#pragma omp target map(to: compression[0:len], e_old[0:len],
                       vnewc[0:len]) map(from: bvc[0:len], p_new[0:len]) {
    #pragma omp parallel for
    for (int i=0 ; i<len ; i++) {
        bvc[i] = cls * (compression[i] + 1.0);
    }
    #pragma omp parallel for
    for (int i=0 ; i<len ; i++) {
        p_new[i] = bvc[i] * e_old[i];
        if ( fabs(p_new[i]) < p_cut ) p_new[i] = 0.0 ;
        if ( vnewc[i] >= eosvmax ) p_new[i] = 0.0 ;
        if ( p_new[i] < pmin ) p_new[i] = pmin ;
    }
}
```

Figure 3: OpenMP PRESSURE kernel in LCALS extended with offload directives to execute on a GPU.

Figure 3 shows the PRESSURE kernel in LCALS originally written for a CPU multicore environment modified for execution on a GPU using the target construct. Our compiler offloads the two loops in a single kernel on the GPU and uses all threads in a thread block for parallel execution of each loop.

4. OFFLOADING WITH RAJA

```
template <typename LOOP_BODY> inline void forall(
omp_gpu_parallel_exec, int begin, int end, LOOP_BODY
loop_body) {
    #pragma omp target
    #pragma omp parallel for schedule(static, 1)
    for (int i = begin; i < end; i++) {
        loop_body(i);
    }
}
```

Figure 4: RAJA template using OpenMP 4.0 for GPU offloading on a thread block with coalesced memory accesses.

To enable GPU execution we created a new execution policy, *omp_gpu_parallel_exec*, that exposes an intent for parallel GPU execution. The corresponding RAJA template in Figure 4 uses the OpenMP 4.0 target construct to offload a loop to the GPU, along with a parallel worksharing construct to execute the loop on threads of a GPU thread block. The parallel worksharing construct uses a static schedule with a chunk size of 1 to generate coalesced

memory accesses across GPU threads and fully exploit the memory bandwidth of a GPU. RAJA allows an application developer to ignore these implementation details when executing on a system with GPUs.

4.1 Establishing the GPU Data Environment

The RAJA template to support offloading cannot explicitly setup a device data environment with a map clause because variables accessed within the lambda function are not visible inside the template [7]. OpenMP implicitly maps all unmapped variables referenced in the lambda function (including arrays) as scalars with a motion type of *tofrom*. This obviously leads to invalid references on the GPU when an array is accessed through the RAJA abstraction.

OpenMP 4.1 [6] allows an elegant solution to this problem through the introduction of standalone unstructured data mapping clauses that can be used to setup the device data environment prior to offloading with RAJA. A *target enter data* construct is used to map variables to the device data environment and the *target exit data* construct to copy them back to the host.

Typical Physics codes store a mesh along with element and node field data in an encapsulated data structure called a Domain. Figure 5 shows such a Domain data structure in the DEL_DOT_VEC_2D kernel of LCALS. A structured mesh that is accessed by this kernel is stored as a member variable (along with other variables not required by the kernel) of this structure. We can use the unstructured data mapping clauses in the constructor and the destructor of the class to selectively setup and teardown the device data environment.

When a RAJA loop is offloaded to the GPU our OpenMP runtime detects the presence of the earlier mapped array *real_zones* on the device and simply sets up a device reference in the kernel. In this manner data definition and motion *to* or *from* an accelerator device can be hidden and separated from the computation specified in RAJA.

```

struct ADomain {
  ADomain( LoopLength ilen, Index_type ndims )
  : ndims(ndims), NPML(2), NPNR(1) {
    // Initialization code
    // Map data onto the device
    #pragma omp target enter data map(to: real_zones[0:
                                     n_real_zones])
  }
  ~ADomain() {
    // Release data from the device
    #pragma omp target exit data map(release: real_zones[0:
                                     n_real_zones])
    if (real_zones) delete []real_zones;
  }
  ...

  int *real_zones;
  int n_real_zones;
};

```

Figure 5: Establishing a data environment for RAJA using unstructured data mapping constructs.

4.2 Code Offloading

The RAJA approach is designed by its authors to express fine-grained parallelism in loops to achieve performance portability with minimum code disruption. When used to program a

heterogeneous system we have found a few deficiencies that we highlight in this section.

4.2.1 Offloading Arbitrary Code Regions

On a system with an accelerator we require a mechanism to express the arbitrary offloading of code in addition to the current mechanism for expressing fine-grained parallelism.

Consider the example of the VOL3D kernel in LCALS. Figure 6 shows the relevant code where several pointers used in the kernel are initialized to reference distinct chunks of pre-allocated working memory.

```

#define NDPTRSET(v,v0,v1,v2,v3,v4,v5,v6,v7) \
  v0 = v ;   v1 = v0 + 1; \
  v2 = v0 + domain.jp; \
  ...
  v7 = v3 + domain.kp;

  Real_ptr x = loop_data.array_1D_Real[0];

  // The following two statements must be executed on the GPU
  UnalignedReal_ptr x0,x1,x2,x3,x4,x5,x6,x7;
  NDPTRSET(x,x0,x1,x2,x3,x4,x5,x6,x7);

  typedef omp_gpu_parallel_exec exec_policy;
  forall<exec_policy>(fpz, lpz + 1, [&] (int i) {
    Real_type x71 = x7[i] - x1[i];
    Real_type x72 = x7[i] - x2[i];
    ...
  }

```

Figure 6: Code snippet from VOL3D showing the need for a RAJA construct to offload a serial section to an accelerator.

One way to offload this code is to first map the working memory referenced by the pointer *x* onto the device and then initialize pointers *x0-x7* on the device so that they reference the device variable. This requires a new RAJA construct that can hold serial code and express the intent of offloading to an accelerator.

```

#pragma omp declare target
  Real_type trap_int_func(Real_type x, Real_type y,
                        Real_type xp, Real_type yp) {
    Real_type denom = (x - xp)*(x - xp) + (y - yp)*(y - yp);
    denom = 1.0/sqrt(denom);
    return denom;
  }
#pragma omp end declare target

typedef omp_gpu_parallel_exec exec_policy;
forall<exec_policy>(0, len, [&] (int i) {
  ... = trap_int_func(x, y, xp, yp);
});

```

Figure 7: RAJA requires a new construct to enable the compilation of functions for an accelerator similar to the declare target construct of OpenMP 4.0.

Second, recall that in OpenMP 4.0 code must be explicitly marked for execution on an accelerator. This also applies to functions that may be called on a device, which must be tagged with a *declare target* construct as shown in Figure 7. The snippet is from the TRAP_INT kernel of LCALS and shows a RAJA loop calling the function *trap_int_func*.

Without the *declare target* construct the function will not be made available on the GPU by an OpenMP compiler (this is particularly

relevant if the function is defined in a different compilation unit) and the RAJA loop will fail in the link stage. RAJA will require an abstraction to support functions on an accelerator.

4.3 Optimizing RAJA GPU Programs

The efficiency and feasibility of RAJA relies on two factors:

- a. The ability of the compiler to aggressively inline instances of the template and lambda functions, and
- b. The ability of the compiler to collect the requisite information about the computation and data expressed in the lambda.

These two factors pose a challenge to the Clang compiler due to its sequence of actions, as summarized below.

- a. During template instantiation, the lambda function is also created and passed as an argument. Unlike a regular function, lambda functions capture references to data accessed within them. In Clang, lambda functions are represented as a unique C++ record (class) whose fields are the captured references. During emission of the lambda body the captured data references are loaded from the fields of the lambda object.
- b. Code generation now occurs, which is also when the OpenMP directives are processed.
- c. Finally, optimizations such as function inlining occur.

We enable aggressive function inlining of RAJA programs to produce efficient code. However, a consequence of the sequence outlined above is that during OpenMP code emission the template and lambda functions are not yet inlined¹. Hence, the OpenMP code generator inside the RAJA template cannot see outside the template boundary to automatically determine the data mapping information required for OpenMP 4.0.

A user may explicitly map data accessed by the lambda onto the GPU with the *enter* and *exit* data constructs as described in Section 4.1. However, the lambda object (C++ record) must be treated as a special case for mapping by the compiler.

4.3.1 Mapping Capture-by-Reference Variables in a Lambda

The lambda function as used in RAJA captures variables by reference. They are represented as pointer fields of a C++ record describing the lambda as noted above. When a RAJA loop is offloaded using the *target* construct a variable of this type is mapped using the default motion type of *tofrom*. Unfortunately the pointer fields of this structure are not translated; they still refer to addresses in the host's address space. This leads to invalid references when the RAJA loop is run on a GPU.

This is a so-called deep copy issue that is still the subject of standardization in OpenMP. To successfully map a structure of arrays, as in this case, the runtime must map not just the structure itself but traverse deeper to also map the fields (pointers) within it.

We modified our compiler and runtime to add limited deep copy support for the special case of a lambda function. After mapping the lambda structure, the runtime checks for the presence of each member pointer within the device data environment. If present,

¹ Inlining before OpenMP code generation may enable additional optimizations, e.g., fusion of RAJA loops. We are investigating how this can be done with minimal disruption to Clang.

the corresponding device address replaces the host pointer value in the field of the mapped lambda structure.

4.3.2 Efficient Offloading of Sequences of RAJA Loops

Consider the PRESSURE kernel in Figure 3 offloaded to an accelerator using OpenMP 4.0. The target region contains a sequence of two parallel loops. As a general rule the larger the target region, the lower the startup overhead since the costs involved in invoking an accelerator can be amortized over a larger execution time.

The equivalent program in RAJA uses two calls to the RAJA abstraction as shown in Figure 8. This results in two back-to-back invocations of the accelerator and therefore increased overheads. In this section we explore various OpenMP 4.1 offload constructs to see how such sequences of RAJA loops may be optimized.

```
typedef omp_gpu_parallel_exec exec_policy;
forall<exec_policy>(0, len, [&] (int i) {
    bvc[i] = cls * (compression[i] + 1.0);
});
forall<exec_policy>(0, len, [&] (int i) {
    p_new[i] = bvc[i] * e_old[i];
    if ( fabs(p_new[i]) < p_cut ) p_new[i] = 0.0;
    if ( vnewc[i] >= eosvmax ) p_new[i] = 0.0;
    if ( p_new[i] < pmin ) p_new[i] = pmin;
});
```

Figure 8: RAJA equivalent of the PRESSURE kernel invokes the GPU twice, once for each loop.

4.3.2.1 RAJA on a GPU Thread Block

We first modified the RAJA template for parallel execution on a GPU to use the *target teams distribute parallel for* construct as shown in Figure 9. As mentioned in Section 3, this construct partitions iterations in a loop across teams for parallel execution. On the GPU these iterations are concurrently executed by multiple thread blocks. OpenMP restricts the placement of this offload construct to only a single loop nest that is to be offloaded so this is an ideal match for RAJA.

```
template <typename LOOP_BODY> inline void forall(
    omp_gpu_parallel_exec, int begin, int end, LOOP_BODY
    loop_body) {
    #pragma omp target teams distribute parallel for
    schedule(static, 1)

    for (int i = begin; i < end; i++) {
        loop_body(i);
    }
}
```

Figure 9: RAJA template using OpenMP 4.0 for offloading onto multiple GPU thread blocks.

4.3.2.2 Minimizing OpenMP Overhead in a RAJA GPU Kernel

Correctly executing general OpenMP programs on a GPU requires a complex thread co-ordination scheme due to the limitations of the architecture. On a GPU the smallest unit of organization is a group of 16 or 32 threads termed a *warp*. A warp of threads must execute in lock-step for maximum performance. Furthermore, synchronization always occurs at the level of a block of threads and must occur at the same syntactic location. Finally, a thread on a GPU has a private stack. To share stack variables between threads, for example, between a master and its team, they must be

promoted to GPU shared memory so that a reference to the corresponding data is valid in the different threads.

General OpenMP target regions may contain serial and parallel sections, sequences of parallel regions with varying numbers of threads, sharing of stack variables between a team master and its team, and nested parallel or simd constructs. Supporting all of this functionality requires a complex control loop formulation on the GPU that essentially guides a subset of threads from serial to parallel regions. This machinery increases the registers and shared memory used in a kernel and reduces the GPU occupancy, i.e., the number of concurrently executing threads on a GPU core, which can result in poor performance.

We can overcome these limitations in the case of RAJA where only a single parallel loop is present in a target region. We know, for example, that once threads are invoked they execute in parallel and only synchronize at kernel termination. Data is also not shared between threads in this construct. Our compiler detects the offload construct used in RAJA and generates a simplified control mechanism with minimal overhead. In Section 5 we study the impact of this simplified code generation scheme on performance.

4.3.2.3 Pipelining RAJA Loops on GPUs

Programs written in RAJA are expected to contain numerous back-to-back target regions, each with small-scale computations. GPU kernels with small-scale computations underutilize the device resources. We are investigating a technique to pipeline consecutive RAJA loops on an accelerator to more fully utilize the GPU resources.

One way to overcome the startup overhead when offloading target regions is to invoke the device asynchronously with respect to the host thread. OpenMP 4.1 provides the *nowait* clause that can be used to initiate the execution of a target region on an accelerator but continue execution of the encountering host thread before the accelerator completes.

```
typedef omp_gpu_parallel_exec async_exec_policy;
template <typename LOOP_BODY> inline void forall(
omp_gpu_async_parallel_exec, int begin, int end,
LOOP_BODY loop_body) {
    #pragma omp target teams distribute parallel for nowait
    schedule(static, 1)

    for (int i = begin; i < end; i++) {
        loop_body(i);
    }
}

void main() {
    ...
    forall<async_exec_policy>(0, len, [&] (int i) {.....} );
    forall<async_exec_policy>(0, len, [&] (int i) {.....} );
    ...
    #pragma omp taskwait
}
```

Figure 10: RAJA loops with the asynchronous execution policy allows a host thread to execute concurrently with its target region.

5. RESULTS

In this section we compare the performance of RAJA against pure OpenMP. We study the LCALS [3] benchmark, a collection of floating-point scientific kernels released by the co-design center at the Lawrence Livermore National Laboratory to interface with

platform vendors. The benchmark is designed to measure SIMD and OpenMP multithreaded performance. It also includes variants to compare multithreaded performance of pure OpenMP and RAJA kernels.

We first ported LCALS to OpenMP 4.0 by adding directives to offload each kernel to the GPU. We were able to successfully do so for all the multithreaded kernels except for COUPLE, and PIC_2D. COUPLE requires the Complex STL library, which is currently not supported on the GPU. PIC_2D requires the data mapping of an array of pointers onto the GPU, which is currently not supported by the map clause of the OpenMP standard.

We run our experiments on an OpenPower system with IBM Power8 CPUs and NVIDIA Kepler K40m GPUs. The kernels are compiled using our Clang/LLVM based compiler and is available online [8]. We employ the Lightweight OpenMP (LOMP) library developed at IBM for the host OpenMP runtime and the GPU offload logic. We use an OpenMP runtime for NVIDIA GPUs that we have developed and released online [9].

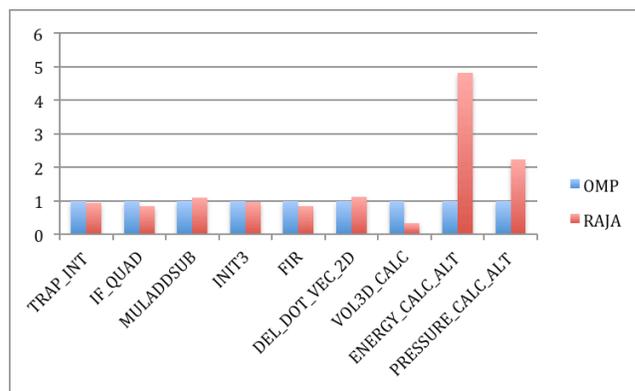


Figure 11: RAJA performance normalized to pure OpenMP 4.0 on LCALS kernels with a small dataset.

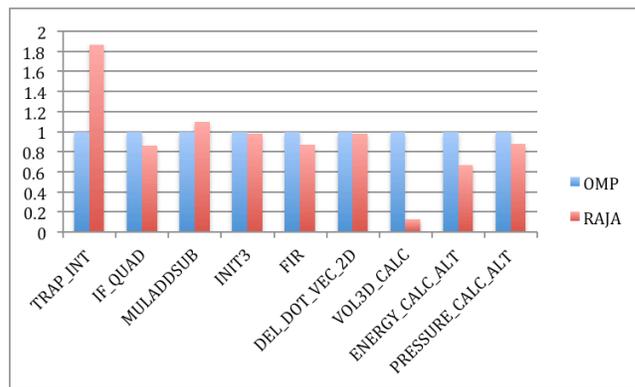


Figure 12: RAJA performance normalized to pure OpenMP 4.0 on LCALS kernels with a large dataset.

Figure 11 compares the performance of the kernels written using pure OpenMP and the RAJA abstraction. In this case we used a small dataset so there is limited work in each kernel for most cases. The runtime is dominated by the overhead of offloading as described in Section 4.3.2.

We see comparable performance in all but three cases. PRESSURE_CALC_ALT² and ENERGY_CALC_ALT are two

² Currently results of PRESSURE_CALC_ALT with RAJA are incorrect. We are investigating this correctness issue.

kernels with 2 and 6 loops respectively. The pure OpenMP version can offload each kernel in a single target region but RAJA prescribes an offload for each loop, which explains some of the poorer performance. We expect some of this overhead to be hidden once we have implemented the asynchronous execution policy as described in Section 4.3.2.3.

For some of these benchmarks RAJA versions produce less optimized code due to two reasons. First, there are multiple load instructions to access array elements transferred from host to target in the main parallel loops. Many of them are loop invariants dereferencing wrapper pointers to get base addresses of arrays. While the Loop Invariant Code Motion (LICM) pass properly hoists these instructions out of the loop for the pure OpenMP versions, it fails to do so for RAJA.

Second, some of the loops have if-else statements represented as branches to true and false basic blocks. The instruction combine optimization tries to eliminate the branches with predicated instructions (e.g., select), but it fails with RAJA version.

The common cause for the optimization failures is conservative alias analysis results for RAJA versions. The additional levels of indirection when referencing target data with RAJA prevents alias analysis from correctly deducing alias sets. Both LICM and predicated if-else rely on alias analysis results to determine if code changes are safe. With the conservative alias sets, the safety of optimizations cannot be proved and so are not applied.

We plan to improve the compiler to better handle the anonymous class object argument for lambda functions to produce more accurate alias sets.

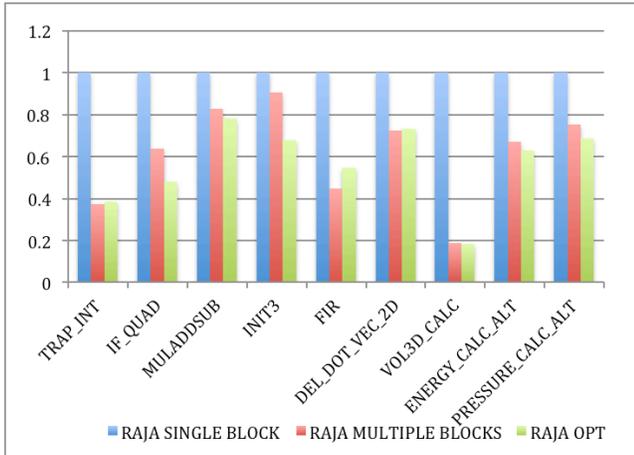


Figure 13: Comparison of optimizations developed for RAJA. Performance is normalized to RAJA SINGLE BLOCK.

Figure 12 shows the results of the experiment with a larger dataset. With more work per kernel the startup overhead can be hidden and the performance of RAJA is comparable to pure OpenMP. Several kernels are faster with RAJA because it splits the offloaded regions to distinct loops that can be individually offloaded to multiple GPU blocks using the teams OpenMP construct as described in Section 4.3.2.1 (of course, the loops in the pure OpenMP target regions can also be rewritten to execute on multiple regions as for RAJA).

These results offer evidence for the merits of the RAJA philosophy. For sequences of loops with enough work the right

approach is to offload them individually to multiple GPU blocks as the overhead of offloading can easily be amortized.

In Figure 13 we compare the two optimizations developed for RAJA programs. The baseline offloads a RAJA loop to a single GPU block; the first optimization to multiple GPU blocks; and the final one uses our simplified OpenMP code generation scheme.

Using multiple GPU blocks is always beneficial for performance, particularly for compute bound kernels like VOL3D. The simplified code generation scheme also improves performance as is evident for the first four kernels, which have limited number of instructions in the loop body. An application developer is oblivious to these optimizations, which are hidden by RAJA.

6. CONCLUSIONS

In this paper we have described how RAJA, a new abstraction for fine-grained parallelism in loops, can be extended to offload loop programs onto a GPU. We identified unique challenges in porting RAJA to a heterogeneous system with CPUs and GPUs due to their distributed code and data spaces, and the offloading model of low-level programming abstractions such as OpenMP.

We introduced several execution policies for efficient code and data offloading in RAJA using OpenMP 4.1. The philosophy of RAJA is to target a single parallel loop for its fundamental unit of abstraction. This choice is validated when offloading a loop with large amounts of work as we can exploit the *teams* construct to offload to the entire GPU and an efficient code generation scheme to minimize OpenMP overhead.

When a loop has limited work the overhead of offloading dominates. We described an asynchronous execution policy that is designed to pipeline the execution of a sequence of RAJA loops. We expect this policy will minimize the overhead of offloading.

7. ACKNOWLEDGEMENT

This work is partially supported by the CORAL project LLNS Subcontract No. B604142.

8. REFERENCES

- [1] R. D. Hornung, and J. A. Keasler. 2014. The RAJA Portability Layer: Overview and Status. LLNL-TR-661403 .
- [2] C. Bertolli et al. 2014. Coordinating GPU threads for OpenMP 4.0 in LLVM. LLVM compiler infrastructure in HPC.
- [3] LCALS. 2015. Retrieved from <https://codesign.llnl.gov/LCALS.php>
- [4] H. C. Edwards, and C. R. Trott. 2013. Kokkos: Enabling Performance Portability Across Manycore Architectures. In Extreme Scaling Workshop.
- [5] OpenMP ARB. OpenMP version 4.0, May 2013.
- [6] OpenMP ARB. OpenMP version 4.1, July 2015.
- [7] W. Scogland et al. 2015. Supporting Indirect Data Mapping in OpenMP. 11th International Workshop on OpenMP.
- [8] Clang with support for OpenMP 4.0. 2015. https://github.com/clang-omp/clang_trunk
- [9] GPU OpenMP runtime. 2015. <https://github.com/clang-omp/libomptarget>