Pointers Inside Lambda Closure Objects in OpenMP Target Offload Regions

David Truby*, Carlo Bertolli†, Steven A. Wright‡*, Gheorghe-Teodor Bercea†, Kevin O'Brien†, Stephen A. Jarvis*
*Department of Computer Science, University of Warwick
†IBM Research
‡Department of Computer Science, University of York

***Abstract*—With the diversification of HPC architectures beyond traditional CPU-based clusters, a number of new frameworks for performance portability across architectures have arisen. One way of implementing such frameworks is to use C++ templates and lambda expressions to design loop-like functions. However, lower level programming APIs that these implementations must use are often designed with C in mind and do not specify how they interact with C++ features such as lambda expressions.**

This paper discusses a change to the behavior of the OpenMP specification with respect to lambda expressions such that when functions generated by lambda expressions are called inside GPU regions, any pointers used in the lambda expression correctly refer to device pointers. This change has been implemented in a branch of the Clang C++ compiler and demonstrated with two representative codes. This change has also been accepted into the draft OpenMP®specification for inclusion in OpenMP 5. Our results show that the implicit mapping of lambda expressions always exhibits identical performance to an explicit mapping but without breaking the abstraction provided by the high level frameworks.

I. Introduction

Over the last 10 years, High Performance Computing (HPC) architectures have diverged from the traditional multi-node CPU-based systems that had previously been prevalent. Emerging architectures such as GPUs bring new programming and portability challenges beyond the traditional performance concerns of scientific applications. As such, to ensure the performance portability of these codes, new frameworks are being developed to abstract parallelization, allowing backends for different performance APIs to be selected. Frameworks implemented in this way include RAJA [1] from Lawrence Livermore National Laboratory (LLNL) and Kokkos [2] from Sandia National Laboratories.

One way of implementing these frameworks is using higher-order functions, providing the programmer with a loop function that takes as arguments the iteration space and a function to perform. Historically these frameworks were arduous to use and usually required creating a class to hold the relevant local variables being used in the loop [3]. Another alternative is to develop a Domain Specific Language (DSL) on top of C++ or Fortran [4]. While these are not arduous to use, it requires maintaining a preprocessor for the DSL and keeping this up to date with any changes to the host language.

With the addition of lambda expressions in C++11, functions being passed as parameters can be specified inline and can automatically capture variables from the enclosing scope. Loops specified in this way, using lambda expressions, appear very similar to traditional C loops, and require much less modification to the original code. Additionally, since lambda expressions are part of the language a separate preprocessor does not need to be maintained.

Currently, there is one major issue with this approach: most HPC application programming interfaces are C based and only support C++ as a corollary of the fact that C++ is largely compatible with C. In particular, many of these interfaces do not specify how lambda expression captures should be handled. This is of particular concern when dealing with non-uniform memory address spaces such as those found on GPUs – the pointers captured by the lambda may not be in the same memory space as required. This can be solved in some APIs by using a unified memory space, such as the CUDA®Unified Memory interface. However, when using these the programmer loses control of when data transfers occur which may not be acceptable for some applications. In particular, in many cases programmers can easily identify optimized data movement schemes, and in order to implement these it is necessary to have full control of the data mapping [5]. When using a framework such as OpenMP that does not support a unified memory space, pointers inside lambda expressions will always refer to the host unless the programmer expicitly specifies device pointers must be captured, and specifying this is not possible in all cases.

This paper explores a change to how lambdas are handled when generating offload code in the OpenMP®API, which since version 4 has supported GPU offloading. Specifically, this paper makes the following contributions:

- We explore a change to the way that pointers are mapped in OpenMP target regions that addresses the issue of non-uniform memory address spaces within OpenMP target regions;
- We implement our design in a branch of the Clang compiler that supports OpenMP 4 offload directives for NVIDIA®GPUs;
- We validate our proposal on two applications that use the RAJA framework, showing no performance difference between implicit and explicit mapping of pointers when using either a simplified lambda-based forall implementation or the RAJA framework.

The behavior described in this paper has been accepted into

978-1-7281-0188-0/18/$31.00 ©2018 IEEE
DOI 10.1109/LLVM-HPC.2018.00005

the draft OpenMP specification for inclusion in OpenMP 5, as described in section 2.20.6.1 [6].

The remainder of this paper is structured as follows: Section 2 provides a summary of similar work; Section 3 provides a brief overview of OpenMP, lambda expressions and the RAJA framework; Section 4 describes our implementation in the Clang compiler; Section 5 contains an evaluation of our modified compiler against an explicitly mapped implementation and a pure OpenMP 4 implementation; and finally, Section 6 concludes this paper.

II. Related Work

The study of implicit data mapping techniques on systems including multiple accelerators with on-chip separate memory space is subject to several studies. One of the earliest comprehensive contributions is described as part of the OpenACC standard [7]. The report includes support for automatic deep copy, where the compiler adds mapping information for complex nested data structures without help from the user. Alternatively, users can explicitly identify fields to be mapped to minimize the amount of data being copied. We are not aware of specific studies based on this report that discuss the performance implications of the proposed techniques. This is instead the focus of our contributions.

Starting with version 4.0, OpenMP is capable of offloading computations to GPUs thus raising performance challenges for both on-device computation and host-device communication. Some of the early experiences with OpenMP are outlined by Karlin et al. [8] and Vergara Larrea et al. [9]. For some time, the OpenMP standard has required explicit handling of data between host and device using maps. This has proven to significantly lower performance and productivity. Newer versions of OpenMP support the management of data to occur implicitly. More recent studies based on OpenMP propose user-defined data mappers [10], which are composable and re-usable across translation units. Our proposed techniques are complementary to user-defined mappers, which could be used to map lambda expressions by describing them as structured data types.

Chen et al. focus on automatic deep copy for OpenMP programs [11]. This is as opposed to our proposal for automatic shallow copy specifically targeting C++ lambda expressions. The main relation with our contribution is a similar focus on performance.

Kokkos [2] offers similar functionalities to RAJA, with a sophisticated support for expressing access patterns through the concept of view. Kokkos has a definition of deep copy for arrays and it refers to the first level of memory indirection, rather than the ability to map nested data structures, such as lists. Calls to deep copy routines must be inserted by the application programmer.

In this paper we perform a performance analysis using two well known HPC mini-applications, namely TeaLeaf and LULESH. TeaLeaf is a mini-app solving the linear heat conduction equation [12]. It contains multiple different solvers for this equation, although this paper only considers the Conjugate Gradient solver. The performance of TeaLeaf has been extensively studied in previous work [13]–[15]. An evaluation of TeaLeaf using the CUDA and OpenMP host backends of RAJA has previously been performed [16], as well as an evaluation of the performance portability of RAJA and a preprocessor based framework called OPS [17]. Our contribution adds to this an evaluation of the OpenMP target offload backend on NVIDIA GPUs.

LULESH is a hydrodynamics code that solves a Sedov blast problem, designed to remain representative of the numerical algorithms and data motion used in larger C++ applications [18]. There are two versions of LULESH, this paper considers the updated second version [19]. LULESH kernels vary in terms of memory footprint and computational intensity and are representative of a large range of kernels encountered in practice. The performance of LULESH has been studied in previous work by comparing the performance of the OpenMP implementation with that of CUDA [20], [21] establishing that OpenMP can make up for most of the performance differences. Previous work has been focused on the computation time of the different kernels [22], [23] and less so on the overhead associated with the launch of kernels onto the GPU. Although only a proxy application, LULESH kernels consist of intricate data access patterns and dependencies which lend themselves to OpenMP's implicit mapping functionality.

To our knowledge, this is the first full report on the performance implications of relying on C++11 lambda expressions, and the analysis of the consequent overheads and their optimization.

III. Background

The OpenMP standard in recent years has added increased support for offloading to target devices, such as GPUs. These devices do not share memory with the host, and as such memory transfers between devices must be done manually by the programmer. However, the implementation is expected to deal correctly with pointer addressing in target regions, making sure that when a pointer is accessed on the device it points to the device memory for that object, not the host memory. The way this addressing should be performed is detailed in the specification. However, the specification currently only makes reference to the C++98 standard, and does not specify how device addressing interacts with features of the newer C++11 and C++14 standards. In particular, this paper is concerned with the interaction between OpenMP and lambda expressions.

A. Lambda Expressions

Lambda expressions are a feature added in the C++11 standard [24] that allow the programmer to provide function literals that are unnamed. This is mostly useful for providing functions as parameters to other functions without outlining the function elsewhere in the code. The lambda expression syntax in C++ allows the user to specify a 'capture' for variables that are defined outside the lambda such that those variables can be used inside. The compiler then generates a structure that contains all of the captured variables and the function provided, which is then passed at the point at which the lambda expression is used.

```
double* x;
int y;

// Capture by reference by default
[&](int i){
  x[i] = y * x[i];
};

// Conceptually generates the following type
class __Closure {
  double** __x; int* __y;

public:
  __Closure(double** x, int* y) : __x{x}, __y{y} {}

  void operator()(int i) {
    (*__x)[i] = *__y * (*__x)[i];
  }
};

// Capture by value by default
[=](int i) {
  x[i] = y * x[i];
};

// Conceptually generates the following type
class __Closure {
  double* __x; int __y;

public:
  __Closure(double* x, int y) : __x{x}, __y{y} {}

  void operator()(int i) {
    __x[i] = __y * __x[i];
  }
};
```

Listing 1: Generated closure object for a simple lambda expression

This structure, referred to in the standard as the closure type, contains any variables that are captured by-value or by-reference into the lambda. If the variables are captured by value, then they are represented in the closure type as members with the same type as the captured value and are copied into the closure object (the instance of the closure type for a specific lambda) when that lambda is constructed. If captured by reference, the variables are represented in the closure type as members with the same type as a pointer to the captured value, and these are initialized with the address of the captured variable on construction of the lambda. Variables of both types can exist in the closure object. If a default capture is used, the variables are only captured if used inside the lambda expression. A conceptual example of how the closure type is formed is shown in listing 1.

B. RAJA

RAJA is a performance portability abstraction framework developed by LLNL that uses loop-style functions and lambda expressions to allow performance portability with minimal changes to the original code. When using the RAJA framework different backends can be selected without modifying the application code, allowing the use of different implementations depending on the target hardware. The advantage of this approach over using a programming model like OpenMP directly is that the abstractions provided by RAJA allow porting applications to new hardware that doesn't support existing programming models by developing a single new backend targeting the programming model used by the hardware. Application codes written on top of RAJA can then run on the new hardware with very little modification.

```
RAJA::forall<ExecPolicy>(0, 100, [=](int i) {
  a[i] = b[i] + c[i];
});
```

Listing 2: Simple vector add in RAJA

```
template <typename Func>
void forall(size_t start, size_t end, Func body)
{
  #pragma omp target teams distribute parallel for
  for (size_t i = start; i < end; ++i) {
    body(i);
  }
}
```

Listing 3: Simplified forall implementation for OpenMP target offload

Using RAJA, the programmer replaces their traditional C-style for loops with a call to a loop function taking an execution policy, a start and end index, and a lambda expression, as shown in listing 2. The execution policy chooses which backend implementation is used to generate code. The backend implementations then call the function once for each loop iteration, passing the loop variable to the function. A simplified implementation of the OpenMP target offload backend is shown in listing 3. However, according to the current specification this implementation will not behave correctly as written. The programmer providing the lambda has no way to specify that the variables inside the lambda need to be device pointers if the lambda is called from the GPU, since the programmer cannot access properties of the closure object generated by the lambda expression. This can be addressed by adding a `#pragma omp use_device_ptr(p)` clause around the forall function call specifying each pointer captured by the lambda, which ensures that inside the relevant region the pointers mentioned in the clause always refer to the device memory space. However this breaks the abstraction that RAJA is designed to provide and is impractical to program for loops that access a large number of arrays, as each pointer would need to be listed in the clause. This clause cannot be put inside the forall function, as there is currently no way in C++ to access the member variables of the closure object that has been passed into the function. Functionality to iterate over the members of an object has been proposed as part of the Static Reflections working group of the C++ standards committee [25], however these proposals have yet to be accepted into any draft standard, meaning their inclusion into the published standard for C++ is a number of years away at best.

IV. Design

Our final design performs a mapping of pointers directly inside the closure object. This takes advantage of the requirement in the C++ standard that only the variables used inside the lambda are included in the closure object, meaning that even if a generic capture is specified only the arrays used are copied. However, this does have the disadvantage that applications such as LULESH must explicitly specify which pointers are to be captured from the Domain object. This can be done using a feature added to the C++ standard in 2014 that allows more complex closure captures to be specified, as shown in listing 4.

```
RAJA::forall(0, numElem, [fx=&domain.fx(0)](int i) {
  fx[i] = 0;
});
```

Listing 4: C++ 14 generic capture as used in LULESH

Recursively mapping the pointers could be ergonomically advantageous for the porting of applications such as LULESH, which in the reference implementation uses a large Domain object containing pointers to each array used in the application. However, if the Domain object is simply captured in a lambda closure, the runtime must perform a mapping of every pointer in the object. In the case of LULESH, this would require performing 150 pointer mappings. As shown in listing 5 each pointer mapping requires a load of the pointer in the closure object, followed by two stores into an offloading array. Additionally, when the runtime executes this mapping it will require a lookup for the pointer in the table connecting host addresses with their corresponding device address. For LULESH this would be 150 loads followed by 300 stores and 150 table lookups for each kernel, including kernels that only access a small number of arrays. An implementation of this design was developed at an early stage of the project and when tested on LULESH was demonstrated to have significant performance issues.

We implement our design in the compiler by scanning the list of types that the compiler generates for each offload region. We find the closure objects in this list and scan them for pointers, which are then passed to the runtime ensuring that accesses to these pointers refer to the device memory space and not the host. The runtime is also informed that the pointer is a member of the closure object to ensure the pointer mapping is correct. The IR generated by this process for a simple kernel with a single pointer is shown in listing 5. This example shows the four offload arrays that need to be passed to the runtime, and how they are prepared by the implicit mapping added to the compiler.

Line 1 contains the array of sizes for the objects being mapped to the device for the target region. In this case, the first two elements are the start and end indices of the loop, the third element is the closure object (of size 8 bytes since it contains a single pointer), and the final element is the pointer; a mapping of size 0 here means to just correct the address to a device address instead of a host address.

Line 4 contains the array of mapping types for the values described above. The exact meanings of these types are not relevant here, except to note that the large value at the end of the array is the mapping type for the pointer, in particular it informs the runtime that the pointer should be mapped to and from the device, and is a member of the previous element of the array.

Lines 11 to 16 store a pointer to the lambda itself in the two arrays that are passed to the runtime such that it knows where the actual values to map are. Lines 18 to 23 perform the same operation for the pointer inside the lambda itself. All four of these arrays are then passed to the runtime call on lines 28 to 34.

IBM S822LC Server
2 × IBM POWER8 3.259 GHz 8-core processors
4 × NVIDIA P100 GPUs
256 GB DDR4 memory

TABLE I: System configuration

V. Performance Analysis

In this section we provide a performance analysis for our implicit mapping scheme against an explicit mapping scheme (using `use_device_ptr`) on two representative HPC mini-applications, LULESH and TeaLeaf. In addition, we provide results for pure OpenMP implementations without any abstraction for comparison; however the main contribution of this paper is to demonstrate the performance implications of the implicit mapping relative to the explicit mapping workaround that was previously required. Note that this workaround is not being proposed as an alternative, since it breaks the abstraction that frameworks such as RAJA are attempting to provide.

The results in this section were obtained on an IBM®POWER8® S822LC system using NVIDIA Tesla®P100 GPUs (see Table I). All the target regions are executed on a single P100 GPU.

A. LULESH

From a data management perspective LULESH is a complex application with a large number of arrays accessed in some kernels on each iteration. Four kernels exhibiting different characteristics have been selected.

The CalcLagrangeElements kernel is a simple kernel containing only floating point operations and array assignments, with no branches. This kernel is included to demonstrate the flat effect that lambdas have on the performance of simple operations. The CalcMonotonicQGradientsForElems kernel is much larger, but still only contains simple floating point operations with no branches. The comparison of these two kernels shows the effects of the size of the kernel on performance when using lambdas. Larger kernels tend to lead to a larger number of registers being allocated thus having a direct impact on the runtime of that kernel on the GPU.

The other two kernels are included to test the performance in specific circumstances. The CalcMonotonicQRegionForElems kernel contains a large number of branches, both switch and if statements, allowing the impact of using lambda expressions on branching code to be tested. CalcSoundSpeedForElems is included as it is the kernel that dominates the application runtime; it is executed more frequently than the other kernels in the application.

The results for CalcLagrangeElements and CalcMonotonicQGradientsForElems, seen in fig. 1a and fig. 1b show that the difference between the implicit mapping and the `use_device_ptr` workaround is negligible and can be attributed to statistical error in the results. Since mapping affects the setup and tear-down of the kernel onto the device, the strictest performance test is running the experiment on small kernels. The smaller the kernel the larger the potential overhead incurred by the implicit mapping scheme. Our results show

```
forall(0, 100, [=](int i) {
  p[i] = 0;
});
@.offload_sizes = private unnamed_addr constant [4 x i64] [i64 4, i64 4, i64 8, i64 0]
; the last value in the following array corresponds to a bit pattern specifying
; that the pointer is a member of the lambda and must be mapped to and from the device
@.offload_maptypes = private unnamed_addr constant [4 x i64] [i64 800, i64 800, i64 673, i64 844424930131987]
; ...
; get the pointer from the lambda
%10 = getelementptr inbounds %class.anon, %class.anon* %loop_body, i32 0, i32 0
%11 = load i32*, i32** %10, align 8
; store the lambda in the offload arrays at index 2
; indices 0 and 1 will always contain the start and end loop counters respectively
%20 = getelementptr inbounds [4 x i8*], [4 x i8*]* %.offload_baseptrs, i32 0, i32 2
%21 = bitcast i8** %20 to %class.anon**
store %class.anon* %loop_body, %class.anon** %21, align 8
%22 = getelementptr inbounds [4 x i8*], [4 x i8*]* %.offload_ptrs, i32 0, i32 2
%23 = bitcast i8** %22 to %class.anon**
store %class.anon* %loop_body, %class.anon** %23, align 8
; store p in the offload arrays
%24 = getelementptr inbounds [4 x i8*], [4 x i8*]* %.offload_baseptrs, i32 0, i32 3
%25 = bitcast i8** %24 to i32***
store i32** %10, i32*** %25, align 8
%26 = getelementptr inbounds [4 x i8*], [4 x i8*]* %.offload_ptrs, i32 0, i32 3
%27 = bitcast i8** %26 to i32**
store i32* %11, i32** %27, align 8
; Prepare the two arrays for offloading
%28 = getelementptr inbounds [4 x i8*], [4 x i8*]* %.offload_baseptrs, i32 0, i32 0
%29 = getelementptr inbounds [4 x i8*], [4 x i8*]* %.offload_ptrs, i32 0, i32 0
; call the kernel on the device with the offload arguments
%30 = call i32 @__tgt_target_teams(i64 -1,
  i8*
  @"__omp_offloading_28_d5e3b8__Z6forallIZ6kernelPiE3$_0EviiT__l3.region_id",
  i32 4, i8** %28, i8** %29,
  i64* getelementptr inbounds ([4 x i64], [4 x i64]* @.offload_sizes, i32 0, i32 0),
  i64* getelementptr inbounds ([4 x i64], [4 x i64]* @.offload_maptypes, i32 0, i32 0),
  i32 0, i32 0)
```

Listing 5: Generated IR for kernel call

that even in this case our implicit mapping scheme does not lead to any additional overheads.

Repeating the comparison for small kernels with the application implemented with RAJA and the application implemented with pure OpenMP, we notice that the performance difference between the two is consistently 25% of the application runtime and does not change with the size of the problem domain being evaluated. This decrease in performance is due to the way RAJA is implemented; rather than using the indicies passed to the forall function directly, RAJA constructs C++ STL style iterators from the indicies and uses these to provide a unified interface for iterator-based and index-based iteration. However, this limits optimization opportunities for the compiler as the code is more opaque.

These results demonstrate that for the simple case of small kernels with no branches, there is no difference between the added implicit lambda mapping and explicit `use_device_ptr` mapping, and a measurable but constant performance difference between using lambdas and pure OpenMP.

The results for CalcMonotonicQRegionForElems, shown in fig. 1c, show a similar result to the simple kernels, with a constant performance difference between the lambda and pure OpenMP versions and no statistically significant difference between the implicit mapping and `use_device_ptr` mapping versions. This demonstrates that the addition of a large number of branches and a larger kernel size does not affect the performance of lambda-based implementations relative to pure OpenMP.

Figure 1d shows the results for the CalcSoundSpeed kernel diverge somewhat from the previously demonstrated results. In this case, as the size of the problem domain increases, the difference in performance between a lambda-based implementation and pure OpenMP implementation increases. This is indicative of the overhead incurred when using lambda expressions, since the lambda function is not currently being inlined correctly. In the OpenMP case the body of the kernel is directly contained in the loop body, and as such does not need inlining. Further investigation on this matter is required.

B. *TeaLeaf*

TeaLeaf is a simpler application with a small number of kernels, but since a small number of arrays are accessed in each iteration it can be run at larger problem sizes on a single GPU than LULESH. This implementation has also been written with and without the use of the RAJA library to test the overhead of RAJA relative to the simplified implementation given in the design section. TeaLeaf includes a number of solvers for linear heat conduction equations, however for simplicity this paper only considers the conjugate gradient solver [12][14].

As RAJA's design of reductions predates the use of OpenMP target offload as a backend for RAJA, the current reduction implementation does not work correctly with these pragmas. In particular it is not possible for RAJA to internally use OpenMP reductions with function objects, as the implementation has no way of accessing the variables within the closure object to specify them as being reduction variables. As a result, the only possible implementation currently is to manually use a critical section to perform the reduction, leading to poor performance. This is an issue that is being investigated by the developers of RAJA.

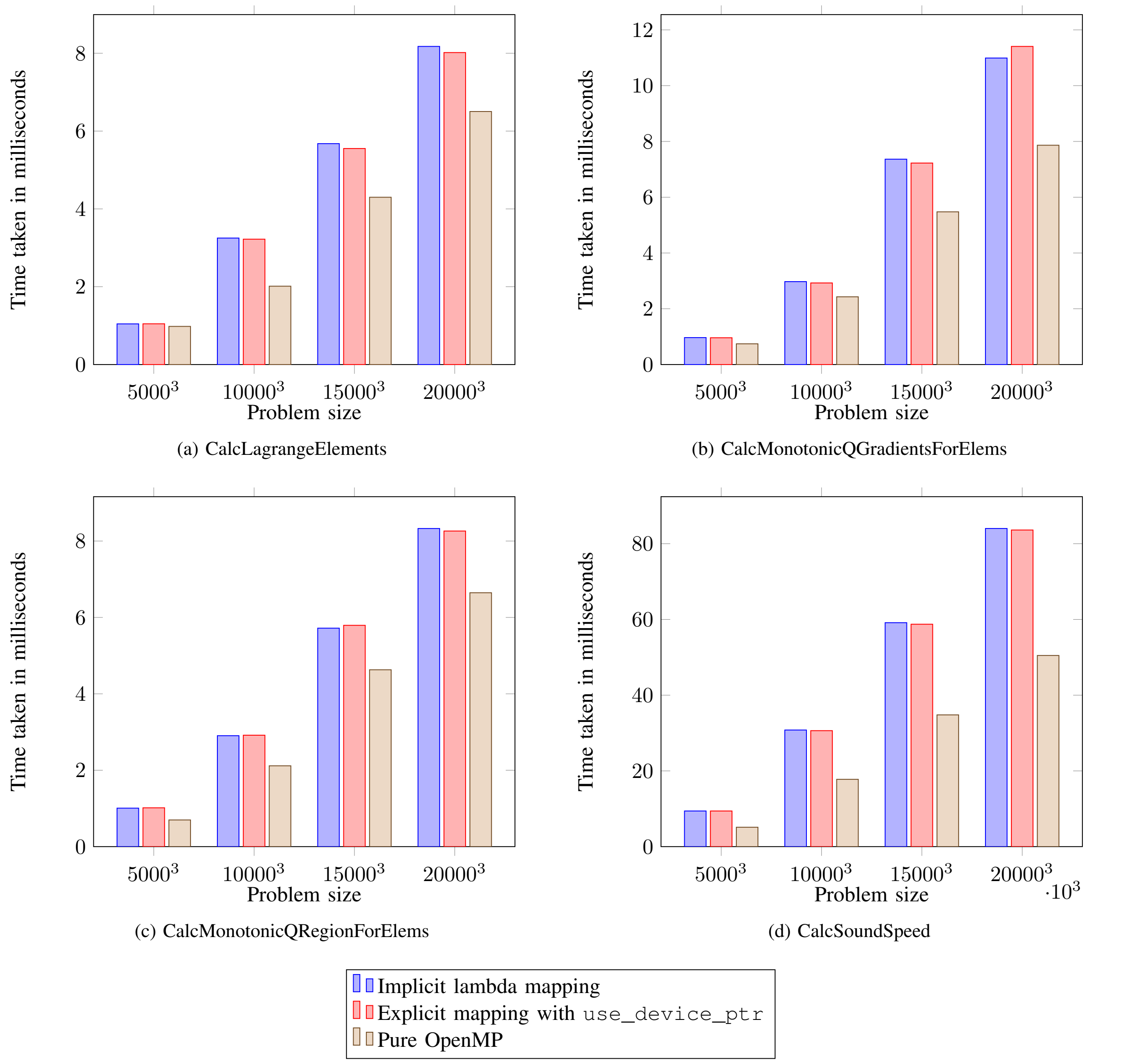

Fig. 1: LULESH kernel results

As a result of this, a simplified reduction implementation has been used here for all the kernels requiring reductions, including in the case where RAJA is being used rather than the simplified forall implementation. Because of this, the comparison between RAJA and plain lambdas is only included for the cg_calc_p kernel, since that kernel does not include a reduction.

The reduction design for lambdas used in this paper is simply to return the value to be added to the reduction variable from the lambda, such that the reduction variable exists in the OpenMP loop and normal OpenMP reductions can be used. The reduction variable is then returned from the function. The implementation of a sum reduction using this design and an example of using it are shown in listing 6.

The results for calc_w show that for a simple reduction kernel using the reduction design shown above, there is no statistically significant difference between a lambda implementation using the implicit mapping, an explicit mapping or pure OpenMP. Each case exhibits the same performance on all problem sizes tested.

The results for calc_p, shown in fig. 3, show the performance implications of using RAJA over a simplified lambda-based implementation. These results show that when using a simplified implementation, the performance is equivalent to pure OpenMP. When using RAJA, the performance degrades over time relative to OpenMP in a similar way to the results for the CalcSoundSpeed kernel in LULESH.

The reason for the performance difference here between the

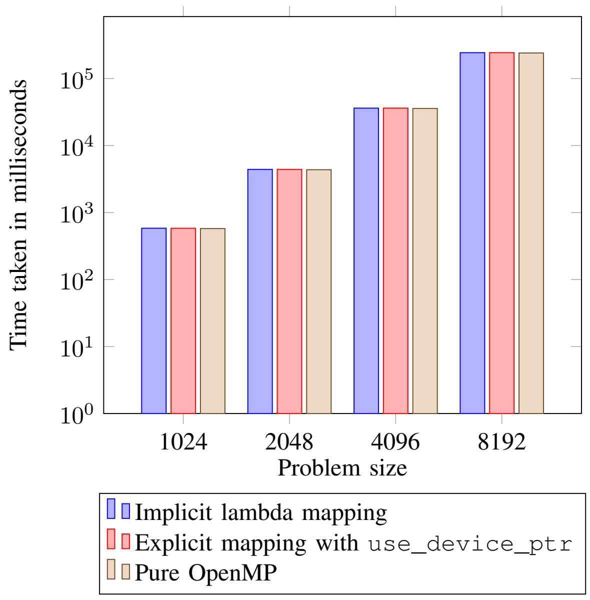

Fig. 2: calc_w reduction kernel

```
template <typename T, typename F>
T reduce_sum(int start, int end, F loop_body,
             T start_value = {})
{
  #pragma omp target teams distribute parallel for \
      reduction(+:start_value) \
      map(tofrom: start_value)
  for (int i = start; i < end; ++i) {
    start_value += loop_body(i);
  }
  return reduction_variable;
}

double sum_vec(double* a, int size) {
  double sum = reduce_sum<double>(0, size, [=](int i){
    return a[i]; // performs start_value += a[i]
  });
  return sum;
}
```

Listing 6: Sum reduction using lambdas

RAJA implementation and a simplified lambda loop implementation is due to the iterator-based indexing discussed in section V-A. These results show exactly the impact of this strategy, which causes the runtime of RAJA-based loop kernels to increase logarithmically with the problem size compared to a loop function simply using the passed indices directly. As a result it may be worth adding specializations to RAJA to use simple integer types when iterating over integer index-based problem spaces, as this would bring the performance of these loops in-line with the simple lambda forall implementation described here.

VI. Conclusion

As supercomputing moves towards the era of exascale computation with a diversification of architectures, it is becoming increasingly important to ensure that codes are performance portable. This performance portability requirement can be tackled by lambda-based frameworks such as RAJA and Kokkos which abstract across different platform API backends. Unfortunately, many of these backends are focussed on C

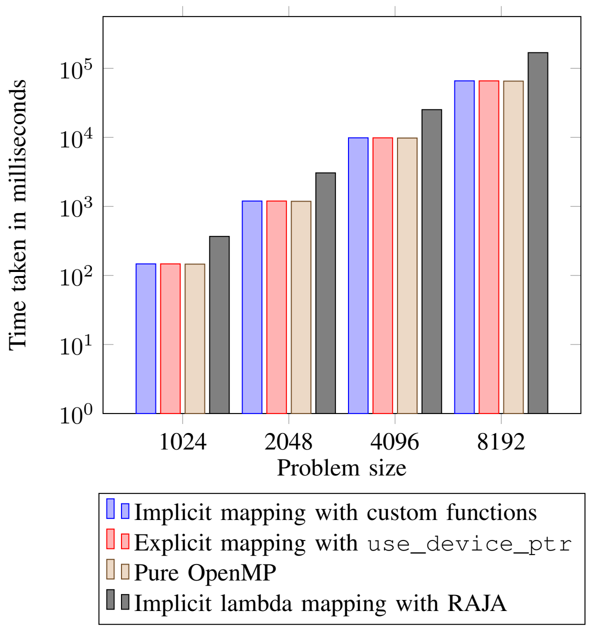

Fig. 3: calc_p kernel results

support and as such are unaware of some newer C++ features such as lambda functions. As such these frameworks need to be modified to be made aware of such features.

In this paper we implement an alternative to the mapping behavior of lambda expressions in OpenMP target regions that has been accepted into the draft OpenMP specification, and explore the performance implications of such a change. The new behavior maps pointers at the top level of a lambda closure type onto the device such that when accesses are performed inside the lambda, the pointers refer to the device memory address region.

We implement this behavior in a branch of the Clang compiler that supports OpenMP target offload directives for NVIDIA GPUs. We considered four possible design options before implementing one design that aims to find a balance between performance and convenience.

We implement key kernels from two applications using the lambda-based abstraction framework RAJA and demonstrate that the performance implications of such a mapping design are statistically insignificant compared to the abstraction breaking explicit mapping previously required to obtain correct behavior with lambda expressions in OpenMP target regions.

As a result, the experiments in this paper show that it is possible to implement a mapping to obtain correct behavior of lambda expressions containing pointers on OpenMP target regions without performance penalties, allowing lambda-based abstraction frameworks to implement OpenMP target offload backends without breaking the abstractions provided by such frameworks.

The results in this paper demonstrate that OpenMP can be modified to be aware of lambda expressions in offload regions with no performance penalty. This allows the development of lambda-based abstraction frameworks with OpenMP target offload backends without forcing a break of the abstraction to identify device pointers in lambdas, increasing the portability of these frameworks and programmer productivity when using them.

A. Future Work

The results presented in this paper show ... promising nature of X.... etc

Building on the work in this paper, we intend to apply the work to applications with other lambda-based frameworks such as Kokkos to demonstrate the implications of this mapping on these libraries. Further, we intend to extend the work by porting an entire application to a lambda-based framework, rather than just specific kernels, such that the performance of a full lambda-based application can be evaluated.

Acknowledgements

This work was completed as part of IBM's 2017 Summer Research Internship Program. This work was also partially supported by the Centre for Computational Plasma Physics, a collaboration between the University of Warwick and the UK Atomic Weapons Establishment. Prof. Stephen Jarvis is an AWE William Penney Fellow.

The authors would also like to express our gratitude to the team at the University of Warwick Scientific Computing Research Technology Platform for access to hardware and support.

References

[1] R. D. Hornung and J. A. Keasler, "The RAJA Performance Portability Layer: Overview and Status," Tech Report, LLNL-TR-661403, 09 2014.

[2] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731514001257

[3] J. Järvi and J. Freeman, "C++ Lambda Expressions and Closures," *Sci. Comput. Program.*, vol. 75, no. 9, pp. 762–772, Sep. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2009.04.003

[4] G. R. Mudalige, M. B. Giles, I. Reguly, C. Bertolli, and P. H. J. Kelly, "OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures," in *2012 Innovative Parallel Computing (InPar)*, May 2012, pp. 1–12.

[5] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of Unified Memory Access performance in CUDA," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2014, pp. 1–6.

[6] OpenMP Architecture Review Board, "OpenMP Technical Report 6: Version 5.0 Preview 2," Draft Standard, Nov. 2017. [Online]. Available: https://www.openmp.org/wp-content/uploads/openmp-tr6.pdf

[7] "Complex Data Management in OpenACC™Programs," OpenACC-Standard.org, Tech. Rep., 2014.

[8] I. Karlin, T. Scogland, A. C. Jacob, S. F. Antao, G. T. Bercea, C. Bertolli, B. R. de Supinski, E. W. Draeger, A. E. Eichenberger, J. Glosli, H. Jones, A. Kunen, D. Poliakoff, and D. F. Richards, "Early Experiences Porting Three Applications to OpenMP 4.5." International Workshop on OpenMP (IWOMP), 2016.

[9] V. G. V. Larrea, W. Joubert, M. G. Lopez, and O. Hernandez, "Early Experiences Writing Performance Portable OpenMP 4 Codes." Cray User Group (CUG), 2016.

[10] T. Scogland, C. Earl, and B. de Supinski, *Custom Data Mapping for Composable Data Management*. Cham: Springer International Publishing, 2017, pp. 338–347. [Online]. Available: https://doi.org/10.1007/978-3-319-65578-9_23

[11] T. Chen, Z. Sura, and H. Sung, *Automatic Copying of Pointer-Based Data Structures*. Cham: Springer International Publishing, 2017, pp. 265–281. [Online]. Available: https://doi.org/10.1007/978-3-319-52709-3_20

[12] S. McIntosh-Smith, M. Martineau, T. Deakin, G. Pawelczak, W. Gaudin, P. Garrett, W. Liu, R. Smedley-Stevenson, and D. Beckingsale, "TeaLeaf: A Mini-Application to Enable Design-Space Explorations for Iterative Sparse Linear Solvers," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2017, pp. 842–849.

[13] M. Martineau, S. McIntosh-Smith, and W. Gaudin, "Evaluating OpenMP 4.0's Effectiveness as a Heterogeneous Parallel Programming Model." International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS), 2016.

[14] M. Martineau, S. McIntosh-Smith, M. Boulton, and W. Gaudin, "An Evaluation of Emerging Many-Core Parallel Programming Models, Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores," ser. PMAM'16. New York, NY, USA: ACM, 2016, pp. 1–10, http://doi.acm.org/10.1145/2883404.2883420.

[15] M. Martineau, C. Bertolli, S. McIntosh-Smith, A. C. Jacob, S. F. Antao, A. Eichenberger, G. T. Bercea, T. Chen, T. Jin, K. O'Brien, G. Rokos, H. Sung, and Z. Sura, "Performance Analysis and Optimization of Clang's OpenMP 4.5 GPU Support." Piscataway, NJ, USA: IEEE Press, 2016.

[16] M. Martineau, S. McIntosh-Smith, and W. Gaudin, "Assessing the performance portability of modern parallel programming models using TeaLeaf," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 15, pp. n/a–n/a, 2017. [Online]. Available: http://dx.doi.org/10.1002/cpe.4117

[17] R. O. Kirk, G. R. Mudalige, I. Z. Reguly, S. A. Wright, M. J. Martineau, and S. A. Jarvis, "Achieving Performance Portability for a Heat Conduction Solver Mini-Application on Modern Multi-core Systems," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2017, pp. 834–841.

[18] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still, "Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application," in *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA, May 2013.

[19] I. Karlin, J. Keasler, and R. Neely, "LULESH 2.0 Updates and Changes," Tech. Rep. LLNL-TR-641973, August 2013.

[20] G. T. Bercea, C. Bertolli, S. F. Antao, A. C. Jacob, A. E. Eichenberger, T. Chen, Z. Sura, H. Sung, G. Rokos, D. Appelhans, and K. O'Brien, "Performance Analysis of OpenMP on a GPU Using a CORAL Proxy Application," in *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*, ser. PMBS '15. New York, NY, USA: ACM, 2015, pp. 2:1–2:11. [Online]. Available: http://doi.acm.org/10.1145/2832087.2832089

[21] S. F. Antao, A. Bataev, A. C. Jacob, G. T. Bercea, A. E. Eichenberger, G. Rokos, M. Martineau, T. Jin, G. Ozen, Z. Sura, T. Chen, H. Sung, C. Bertolli, and K. O'Brien, "Offloading Support for OpenMP in Clang and LLVM, Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC," ser. LLVM-HPC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 1–11, https://doi.org/10.1109/LLVM-HPC.2016.6.

[22] C. Bertolli, S. F. Antao, A. E. Eichenberger, K. O'Brien, Z. Sura, A. C. Jacob, T. Chen, and O. Sallenave, "Coordinating GPU Threads for OpenMP 4.0 in LLVM, Proceedings of the 2014 LLVM Compiler Infrastructure in HPC," ser. LLVM-HPC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 12–21, http://dx.doi.org/10.1109/LLVM-HPC.2014.10.

[23] C. Bertolli, S. F. Antao, G. T. Bercea, A. C. Jacob, A. E. Eichenberger, T. Chen, Z. Sura, H. Sung, G. Rokos, D. Appelhans, and K. O'Brien, "Integrating GPU Support for OpenMP Offloading Directives into Clang, 2015 llvm compiler infrastructure in hpc," ser. LLVM-HPC '15, 2015.

[24] "Information technology – Programming languages – C++," International Organization for Standardization, Geneva, CH, Standard, Sep. 2011.

[25] M. Chochlík, A. Naumann, and D. Sankel, "Static Reflection – Rationale, design and evolution," International Organization for Standardization JTC1/SC22/WG21/SG7, Proposal, 2017.