



LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

A Case for Improved C++ Compiler Support to Enable Performance Portability in Large Physics Simulation Codes

R. D. Hornung, J. A. Keasler

April 24, 2013

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

A Case for Improved C++ Compiler Support to Enable Performance Portability in Large Physics Simulation Codes

Richard D. Hornung and Jeffrey A. Keasler
Lawrence Livermore National Laboratory, Livermore, CA 94551*
(email: hornung1@llnl.gov, keasler1@llnl.gov)

Executive Summary

Vendors of modern computer platforms typically include the capacity of SIMD vector units in advertised theoretical peak system performance. However, the extensive, *non-portable* source code modifications required for compilers to generate SIMD vector instructions are not manageable in large simulation codes. Thus, realized performance is substantially less than one quarter to one eighth of machine capability on current platforms. In this paper, we argue that to increase efficiency of machine usage, compilers must improve their support for SIMD vectorization and other optimizations. We show that good vectorization is possible with current compiler technology but that it is impractical to achieve in large codes, due to maintenance and portability concerns. We enumerate specific corrective actions that compiler vendors could pursue to resolve some key issues. These compiler improvements would also allow us to exploit other important forms of parallelism in a portable manner through high-level software constructs.

1. Introduction

A few specific C++ compiler improvements would significantly boost the prospects for achieving *portable* high performance with large physics codes. Efforts at DOE Laboratories and in industry are exploring software abstractions based on standard C++ features to solve HPC performance and portability problems for current and future platforms. A major hurdle hampering compiler optimizations is the pervasive use of C-style data pointers in simulation codes. This forces compilers to make conservative assumptions about data aliasing and alignment that typically prevent optimizations, such as SIMD vectorization. Compilers do allow programmers to provide precise source code hints to circumvent conservative assumptions. For example, every C++ compiler recognizes the “`__restrict__`” pseudo-keyword and provides directives for data alignment that can assist SIMD vectorization. Indeed, all compiler vendor optimization guides recommend using such code decorations, thus acknowledging their necessity. However, these constructs are not innate to the

* This work was performed under the auspices of the U.C. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. UCRL-TR-635681.

programming language and, thus, are inconsistently supported and maintained across compilers. See the Appendices for details.

We view consistent encapsulation of compiler and platform-specific directives and attributes within appropriate abstraction layers and *typedefs* as a critical enabler of performance portability. The current inability to fully encapsulate all necessary information is unsustainable, as it requires pervasive non-portable code adornments. Even when encapsulation is possible with current compilers, information is not consistently propagated through the compilation process and yields “hit or miss” optimization, likely due to the informal support for these features. Optimization quality varies across compilers, as well as different versions of the same compiler. Thus, programmers face few perceptible clues to the impact of source code implementation choices on performance. Success on future platforms depends heavily on addressing these problems. Substantial performance improvements on existing machines would also result.

This document begins with a brief overview of performance portability concerns and challenges facing code developers as hardware advances. Then, we enumerate common compiler deficiencies that make high performance difficult to achieve and which hinder portability. We make concrete recommendations to fix the problems. Next, we describe how C++ language features may be used to build performance portable software abstractions by insulating application code from compiler attributes and programming model details. Several efforts in the HPC community are exploring similar concepts. Then, we describe a benchmark we have developed to reveal key issues and demonstrate that, if compiler support were improved, many issues we encounter are eliminated. We recommend that such benchmarks be used for compiler evaluation and scoring. In the Appendices, we provide detailed source code examples that illustrate some of the more vexing issues we observe.

2. What is the Performance Portability Problem?

The single most daunting issue facing developers of HPC application codes is discerning a practical path-forward to achieve high performance *portably* on increasingly diverse hardware platforms. A typical multi-physics code, for example, runs on a variety of machines over a wide range of scales from single CPU laptops to petascale systems containing millions of processor cores. Such codes have scaled successfully in the recent past primarily because processor architectures and memory systems have remained relatively homogeneous and consistent while CPU clock rates have increased considerably. Also, compilers have applied optimizations that exploit sophisticated processor features, such as instruction level parallelism (ILP), to hide latency. Consequently, application scientists have been able to focus on coarse-grained algorithm and model development with reasonably little concern for particular hardware details.

Although precise knowledge of looming architecture changes is unknown, general hardware trends are clear. Most notable is the emergence of hardware features that enable substantial fine-grained concurrency, such as multi-core CPUs, hardware threads, vector units, and many-core GPUs. To exploit these capabilities, codes must express different and varied forms of *on node* parallelism, such as multi-core threading, SIMD vectorization, and SIMT. Also, data locality and memory access patterns must be managed more carefully than in the past due to increasing system complexity and heterogeneity.

SIMD vectorization is perhaps the best way to optimize for power constraints and performance. Hardware vendors are building increasingly wider vector units into processors. Thus, neglecting SIMD optimizations becomes more wasteful with each advance in this hardware technology. *Theoretical peak FLOPS* are measured assuming full SIMD vectorization. Real application codes typically achieve little or no SIMD vectorization, resulting in a maximum performance of 1/4th, 1/8th, or even 1/16th of advertised peak FLOPS depending on vector unit width. Translating source code into SIMD instructions is the job of a compiler. However, the current state of compiler-specific pragmas and intrinsic directives that programmers must provide are unmanageable in large codes (e.g., a large multiphysics code containing a million statements may require 100,000 additional compiler-specific directives be added by hand; see Appendix A).

Large simulation efforts at DOE Laboratories have made huge investments in C and C++ code development, which should be preserved. Achieving high levels of concurrency while limiting data motion will likely require tuning parallelization strategies and memory usage in these codes for particular hardware features. A typical application code may contain tens of thousands of loops and routines that require restructuring or rewriting. Programming models that simplify writing parallel code and which enable compilers to optimize well are necessary to make high performance code development a tractable problem. Robust software abstractions that insulate domain scientist developers from implementation details specific to compilers, platforms, and programming models are necessary for portability.

3. C++ Optimization Issues and Recommendations

This section enumerates C++ compiler deficiencies that make it difficult to achieve highly optimized code and which hinder aspects of performance portability. Often, the issues are rooted in compiler heuristics that decide when certain optimizations are beneficial, or in vendor implementation choices when language standards allow for interpretation. Appendix B provides source code examples that illustrate many of these issues.

- **Specialized directives.** We have emphasized that current compilers require programmer assistance to generate highly optimal, efficient code. Generally, this involves heavy use of specialized directives and intrinsic

functions that disambiguate *aliasing*, or data sharing. For example, compilers will emit no SIMD vector instructions, or possibly sub-optimal instructions, without detailed information about data alignment and aliasing disambiguation via the *restrict* pseudo-keyword. See Example A in Appendix B.

Rather than adorn individual data pointers with directives and attributes wherever they are used in a code, this information can be more effectively encapsulated in a *typedef*, for instance. Compilers already support this ability to varying degrees. See Example B in Appendix B.

Recommendations.

- Compilers will supply data directives and attributes to disambiguate pointer aliasing and *consistently* apply robust SIMD vectorization optimizations in response to those directives.
 - Compilers will enable the `__attribute__` mechanism to be applied to pointers declared in *any* scope to specify data-centric information. The goal will be to obviate the need for similar pseudo-function based directives where possible.
 - Compilers will enable data-centric `__attribute__` annotations to be embedded within a typedef declaration.
- **“Hit-or-miss” optimization.** Compilers do not consistently propagate directives and attributes through all elements of type systems or optimization phases. For example, the `__restrict__` attribute should be honored when attached to pointers that are fields of structures, class members, or global variables. Typically, when structure fields are used directly in expressions, type attributes are lost and so certain optimizations do not occur. Certain optimizations can be recovered by assigning fields to local variables declared with `__restrict__`, but this requires extra code and makes software less readable and maintainable. See Example C in Appendix B.

Recommendations.

- Compilers will fully propagate all information expressed in a typedef through *all* optimization machinery, providing all compilation phases with full knowledge to apply appropriate optimizations.
 - Compilers will rigorously test optimization directives as though they were standard language features. Often, optimizations work for some time (e.g., a few compiler releases) and then stop working because they are not tested and maintained at the same level of rigor as the core language.
- **Insufficient inline support.** Robust abstraction layers often use

lightweight functions to hide implementation details. To avoid excessive function call overhead, programmers need more control over function inlining. The C++ language standard leaves to the compiler to decide whether a function will be inlined; the “inline” keyword is considered a *suggestion* from a programmer. Compilers also provide various compile line flags and arguments to influence the extent of inlining that may occur, but these are applied indiscriminately throughout each translation unit (i.e., source file) when specified.

Recommendations.

- Compilers will provide the ability to attach inline attributes to functions using the “`__attribute__`” mechanism (e.g., “`__attribute__((always_inline))`”). Then, guarantee that such inlining will always occur.
- **Minimum “trip-count” requirements.** It is common in physics codes to gather mesh-based data into short arrays to process for cache efficiency. For example, we may iterate over zones on a mesh, pack data in each zone into temporary arrays, and pass them to a function that includes a statically bound loop over the array elements. However, compilers often decide that loops with a *trip count* less than some heuristically set value will not benefit from SIMD vectorization, without examining the actual amount of work being done in the loop. Flop-intensive code that occurs in a leaf function rather than directly in a loop body may also be passed over for SIMD optimization.

Recommendations.

- Compiler will give programmers more control over when to apply SIMD vectorization. For example, a compiler could provide an option to always apply SIMD vectorization *when it does not prevent correctness* and allow programmers to disable it, using “`#pragma novector`” for example, to override this behavior.
- Compilers will attempt to SIMD vectorize algorithms that do not appear in the context of a loop body, such as in the case of “leaf functions” with sufficient amounts of work.
- **Automatic data alignment.** Heap allocated data using the C-standard library “`malloc`” functions and the C++ “`new`” operator are not required by any language standard to be aligned with SIMD vector boundaries. The same is true for local, stack-allocated temporary arrays, which are used commonly in short inner loops that are nested in longer loops. Compilers should be able to automatically align this stack data properly for SIMD vectorization when appropriate and pass the alignment information to the optimizer so it is applied to the extent possible.

Recommendations.

- Compilers will supply programmers with a mechanism to guarantee that all heap data allocated through standard allocation operations will meet a minimum alignment restriction.
 - Compilers will automatically SIMD align local scope stack-based arrays whenever they are used in operations that have the potential for SIMD vectorization.
- **Proper treatment of function attributes.** Some compilers allow programmers to attach attributes to functions with the intent of helping the compiler optimize function calls. For example, “pure” and “const” `__attribute__` declarations indicate that a function depends only on its arguments (and possibly global variables for “pure” functions) and have no side effects apart from its return value. Properly annotated functions should not inhibit certain optimizations. Using these attributes with function prototypes in header files should be sufficient to disambiguate aliasing assumptions for these functions in the calling routine. See Example D in Appendix B.

Recommendations.

- Compilers will properly apply function attributes as supplied by programmers in header function prototype declarations to disambiguate aliasing in the calling functions. The link phase could optionally verify that such contracts were correctly specified.
- **Insufficient lambda function support.** We defer this to Section 4.

4. Promising C++ Abstractions for Performance Portability

In this section, we describe basic elements of powerful and flexible approaches to portability that employs C++ language features. The viability of these approaches requires resolution of fundamental, yet largely straightforward, compiler deficiencies, discussed here, in Section 3, and illustrated in Appendix B.

Software engineers working on large applications possess deep knowledge and insight about how to build software abstractions that can insulate much of the source code from concerns specific to compilers, platforms, and programming models. For example, the early standardization of MPI and encapsulation of its functionality in codes has enabled excellent coarse-grained parallel scalability on platforms to date. We contend further advances in encapsulation of other parallelization and platform concerns are necessary to make large codes performance portable now and into the future. Programming models, such as OpenMP, OpenACC, OpenCL, CUDA, etc., do simplify the process of exploiting other finer-grained forms of parallelism. However, these models differ in maturity and there is no available, or proposed, programming model that provides a “one size fits all” solution to all potential parallelization strategies. In

addition, each requires specialized syntax and programming considerations. The diversity of potential programming models and constructs, with non-uniform usage considerations, exacerbates the already difficult problem of increasing the amount of application concurrency.

DOE Laboratory and industry efforts are exploring the potential of standard C++ language features to solve performance portability issues. Efforts include: RAJA at LLNL, the KokkosArray Library at SNL, Threading Building Blocks (TBB) from Intel, the open source Thrust Library promoted by Nvidia, and the Bolt Library from the Heterogeneous System Architecture Foundation, to name a few. These models share a common approach to use high-level C++ features to decouple algorithm representations from their implementations. Such decoupling enables compile time specialization of data layouts and access patterns as well as parallelization strategies. Other aspects of execution, such as work scheduling and dispatch, can be definable at run time. The key point is that implementation details tuned for particular architectures are embedded in software abstractions. These abstractions, in turn, rely on the basic level of compiler functionality discussed in Section 3 to enable machine specific optimizations.

Well-designed abstractions allow application developers to be insulated from details associated with compilers and platforms. Abstraction layers also provide the ability to employ various programming models without wedding a code to any particular one. For example, a lightweight C++ API can be inserted into an existing C++ code with less code modification compared to incorporating some programming models directly. The result is that applications can be more flexible and achieve portable high performance without major code changes for new hardware.

As a basic illustration, consider a simple “daxpy” operation implemented using a C/C++ for-loop:

```
for ( int i = begin; i < end; ++i ) {
    y[i] += a*x[i];
}
```

Here, “x” and “y” are arrays of floating point data and “a” is a scalar. Suppose we decouple the loop iteration from the loop body by encoding the loop traversal in a generic function template, such as:

```
template < typename LOOP_BODY >
void forall( int begin, int end, LOOP_BODY body )
{
    for ( int i = begin; i < end; ++i ) body( i );
}
```

Then, we can instantiate the loop in a code by calling the method and passing the loop body as an argument (here we use a *C++ lambda function*):

```
forall( begin, end, [&] (int i) {  
    y[i] += a*x[i];  
} );
```

Note that the loop body is written verbatim as in the original code. An index range (“beg”, “end”) and a loop variable (“i”) passed to the template method define the traversal.

What have we accomplished with this software decoupling? It may appear that we have done nothing more than write the operation using more complicated C++ syntax. However, this example shows some important connections between software abstractions and performance portability. Note that the original loop implementation *explicitly* defines the iteration order and the pattern of data access and execution scheduling *in the application code*. The forall() method approach expresses the same operations, but separates the implementation details from the application code. In particular, we can hide compiler directives and other code decorations from the source code by placing them in a generic traversal method. Different traversal methods, defined in platform-specific header files for instance, can be used transparently in the code for machine specific optimizations. Many (possibly thousands) of loop bodies may share the same traversal method. Likewise, we may pass a given loop body to various “forall” templates that apply different execution strategies. For data parallel loops, for instance, we can add an OpenMP “parallel for” directive to yield thread parallelization. We can also apply different iteration patterns based on different array data layouts. Centralized control of traversal is important for portability, especially where different memory subsystems have different chunking and timing rules for data access.

While not every loop is amenable to the sort of loop abstraction layer shown here, a substantial majority of loops in large physics codes are. To apply such loop abstractions, application developers would characterize performance-critical loops in their codes according to their loop structure and parallel execution possibilities. Then, they would convert the for-expression portion of a for-statement, to a call to a traversal method and pass the unchanged loop body statement as a lambda function. At that point, implementation alternatives can be explored easily within the centralized traversal method.

Lambda functions are a centerpiece feature of the C++11 language standard. They greatly simplify the use of generic C++ algorithms based on function templates, which have been widely used for well over a decade. Here, we suggest that lambda functions can be an important tool for building software abstractions that address performance portability problems. Unfortunately, some C++ compilers do not support lambda functions yet. For those compilers,

one could use C++ functors instead, but this is a highly unattractive alternative in this context due to cumbersome syntax, scoping rules, and poor code readability. We expect that all vendors will implement the C++11 language standard eventually as an increasing number of C++ developers will want to more easily exploit powerful capabilities in the C++ standard library and others such as Boost, Thrust, and Bolt libraries.

The *compile-time polymorphism*, which enables separation of the traversal and the algorithm, allows code operations to be tailored to different programming models and execution environments with little or no change to application code. Since loops are instantiated at compile time, all appropriate optimizations are possible, as long as compiler functionality is robust. However, all C++ compilers are deficient in their ability to yield high quality optimizations when using lambda functions or functors with function templates. The issues are similar to the treatment of structure fields and class members described in Section 3. See Example C and Example E in Appendix B.

Our recommendations to compiler vendors with respect to their support for lambda functions are:

- Compilers will fully support lambda functions and other related elements of the C++11 language standard. Compiler vendors that do not will alienate a large portion of the scientific computing community that rely on C++ abstractions for architectural portability in addition to many other C++ developers.
- Compilers will properly propagate data type attributes completely through the lambda variable capture mechanism so that when lambda functions are passed as arguments to function templates, these functions will optimize as though they were inline code.
- Compilers will look for opportunities to do an inlining pass before lambda function instantiation, allowing for potentially better optimizations. For example, inlining a traversal method first may be more amenable to optimization than first converting a lambda function to an internal functor and then applying the inline pass.

5. New “Livermore Loops”

We have developed a new “21st Century Livermore Loops” test suite. It is based in part on “Livermore Loops Coded in C”, developed by Steve Langer at LLNL in the mid-1990s, which was derived from “Livermore FORTRAN Kernels”, by Frank McMahon which appeared a decade earlier. These older suites were used to evaluate floating-point performance of hardware platforms prior to porting larger application codes.

The new suite is geared toward assessing C++ compiler optimizations. It contains 20 of 24 loop kernels from the older suites, plus various others

representative of loops found in current production application codes at LLNL. The latter loops emphasize more complex operations, loop constructs, and data access patterns than the others, such as multi-dimensional finite difference stencils. The loop suite framework is configurable; for example, it is straightforward to control compilation, loop sampling for execution timing, which loops are run and their lengths. It generates timing statistics for analysis and comparing variants of individual loops.

An initial goal of the new loop suite was to compare performance of commonly used C language “for”-loop constructs and alternatives that employ abstractions based on more advanced C++ language features as described in Section 4. With the suite, we have demonstrated that code abstractions used in this way can be optimized fully by a C++ compiler with robust lambda function and inline support. Figure 1 below illustrates the point for a particular compiler. Other compilers show similar qualitative behavior. The plot compares loop runtimes for variants where we pass the loop body as a lambda function to a generic “forall” method (see Section 4) to traditional C-style for-loops. The vertical axis is relative run time of the forall-lambda variant normalized to the C-style variant.

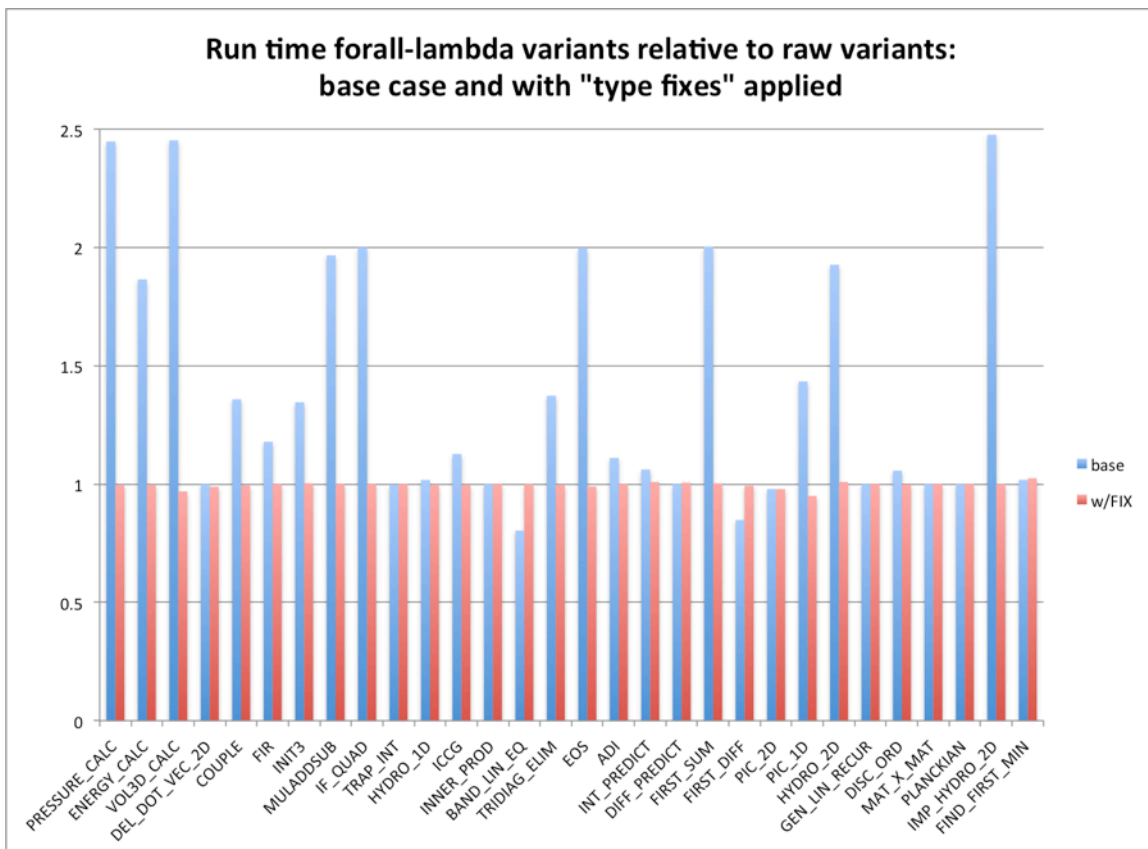


Figure 1. This chart compares runtimes of forall-lambda loop variants (see Section 4) to C-style for-loop variants (normalized value of 1 on vertical axis). Blue bars are runtimes when loop bodies are translated verbatim to lambda functions. Red bars are times when the “fix” in Example E in Appendix B is applied. Intel compiler version 13.0.117 used with options: “-O3 -mavx -inline-max-total-size=10000 -inline-forceinline -ansi-alias -std=c++0x”.

Blue and red bars indicate forall-lambda variants with and without a “fix” where type attributes are re-declared within the lambda expression using local pointer variables, respectively. For example, the chart shows that the runtime of the forall-lambda variant of the “EOS” loop without the “fix” is twice that of the C-style loop, whereas the run time is the same as the C-style variant when the “fix” is applied. See Example E in Appendix B for an explanation of the “fix”.

The most important thing to note about this plot is that when the “fix” is applied, all forall-lambda loop variants run as fast as C-style for loops. Without the “fix”, the performance impact is substantial, up to 2.5x slower in several cases. From experiments like this, we conclude that compilers have sufficient machinery to fully optimize the code abstractions. However, they require excessive source code hints from programmers to optimize properly in their current state.

The ability of compilers to optimize through abstraction layers is essential to achieve performance portability with standard programming languages. The loop suite has helped us expose and understand compiler optimization issues and platform capabilities. Code snippets extracted from the suite are the basis for our continuing interactions with compiler vendors to resolve issues. It would be immensely beneficial to the HPC community to include a similar loop suite in future platform procurements. This would provide a way to score compiler optimization capabilities as they apply to larger production codes and help us in our procurement decisions.

A. Appendix: Source code decorations required for SIMD vectorization

This appendix shows the variety of source code needed by compilers to generate optimal SIMD vector instructions. All code examples are variations of the following simple data parallel “daxpy” routine:

```
void daxpy(double *y, double a, double *x, int len) {  
    for ( int i=0; i<len; ++i )  
        y[i] += a*x[i] ;  
}
```

Compilers typically will not optimize such an operation in this “undecorated” form; for example, SIMD vector instructions will not be generated. Data pointers in C and C++ force compilers to make conservative assumptions about data aliasing and alignment to insure correct execution. Such assumptions typically prevent many optimizations. The following code examples show variations among code adornments for three different compilers that programmers may use to provide precise source code hints to enable SIMD optimizations.

Intel (v13.0.117):

```
void daxpy(double * __restrict__ y, double a,  
           double * __restrict__ x, int len) {  
    __assume_aligned(x, 16) ;  
    __assume_aligned(y, 16) ;  
  
    for ( int i=0; i<len; ++i )  
        y[i] += a*x[i] ;  
}
```

IBM (older BG/L compiler):

```
void daxpy(double * __restrict__ y, double a,  
           double * __restrict__ x, int len) {  
#pragma disjoint (*x, *y)  
    __alignx(16, x) ;  
    __alignx(16, y) ;  
  
    for ( int i=0; i<len; ++i )  
        y[i] += a*x[i] ;  
}
```

GNU (v4.7):

```

void daxpy(double *yy, double a, double *xx, int len) {
    double * __restrict__ x =
        (double * __restrict__ )__builtin_assume_aligned(xx, 16);
    double * __restrict__ y =
        (double * __restrict__ )__builtin_assume_aligned(yy, 16);

    for (int i=0; i<len; ++i)
        y[i] += a*x[i] ;
}

```

Although the “__restrict__” pseudo-keyword is not part of the C++ language standard (it is in the C99 standard), most compilers support it to express data alias disambiguation. Typically, the restrict qualifier is sufficient to achieve some level of SIMD vectorization. Data alignment information is required to generate efficient vector instructions. Data alignment annotations are often unique for each compiler and may involve pragma directives or intrinsic functions as shown here.

Several important points should be taken away from these examples. First, decorations that are generally required to enable compiler optimizations vary significantly across compilers. Second, inserting such decorations in a large code with many loops (tens of thousands of loops in a large physics code is common) for a single compiler is a substantial task. Third, when the code must be built with different compilers to execute on a variety of platforms, it is very difficult to enable all compiler optimization features in a maintainable manner.

One of key recommendations we make in this document is that compiler vendors support the attachment of attributes for data aliasing disambiguation and alignment to data pointers using a typedef declaration. See Example B in Appendix B for details. By specifying compiler-specific information in a single location, such as a header file, it can be propagated easily to many appropriate sites in a large code. Then, for example, our daxpy routine would appear in a much more maintainable form in a code as:

```

void daxpy(Real_ptr y, double a, Real_ptr x, int len) {

    for ( int i=0; i<len; ++i )
        y[i] += a*x[i] ;
}

```

Here, “Real_ptr” is a typedef that decorates a double pointer with necessary compiler-specific data restriction and alignment information.

B. Appendix: Compiler optimization issues

This appendix illustrates compiler optimization issues discussed in Section 3 using simple code examples.

Example A

This example shows that compilers typically require detailed information about data aliasing disambiguation to generate optimal SIMD vectorized code.

```
void muladdsub1(int len,
               double* __restrict__ out1,
               double* __restrict__ out2,
               double* __restrict__ out3,
               double* in1, double* in2)
{
    for ( int i=0; i<len ; ++i ) {
        out1[i] = in1[i] * in2[i];
        out2[i] = in1[i] + in2[i];
        out3[i] = in1[i] - in2[i];
    }
}
```

The `muladdsub1()` routine contains a simple data parallel loop. Without the `__restrict__` qualifier on the “out” variables, compilers will not generate SIMD instructions typically since the possibility of data aliasing automatically disqualifies SIMD operations. The addition of `__restrict__` enables suboptimal SIMD operations (Example B shows what is required for better SIMD generation). Without `__restrict__`, this loop will read six input double values (“in1” and “in2” redundantly for each statement). Here, `__restrict__` indicates that the written values do not overlap memory with the input values, and so the “in” variables only need to be read once per loop iteration, curtailing six memory read operations to two, and reducing memory bandwidth pressure.

Example B

This example shows how attributes for data aliasing disambiguation and alignment can be attached to data pointers using a typedef declaration. This is a much cleaner and maintainable alternative since a single typedef can be placed in a header file and then used throughout a large code easily.

Every C++ compiler supports the attachment of the “`__restrict__`” qualifier to pointers via a typedef. The additional use of alignment attributes can enable better SIMD instruction generation. However, not all compilers will allow alignment attributes to be specified in a typedef declaration as shown in function `muladdsub2()` below. Compilers may instead require the use of a specialized intrinsic function for each non-aliased pointer in a code (recall Appendix A). For a large code with 20,000 loops and five array variables per loop, this amounts to 100,000 extra lines of non-portable code to achieve full optimization.


```

typedef double* __restrict__
               __attribute__((align_value (32))) Real_ptr;

// ...

void muladdsub2(int len,
               Real_ptr out1, Real_ptr out2, Real_ptr out3,
               Real_ptr in1, Real_ptr in2)
{
    for ( int i=0; i<len ; ++i ) {
        out1[i] = in1[i] * in2[i];
        out2[i] = in1[i] + in2[i];
        out3[i] = in1[i] - in2[i];
    }
}

```

Example C

This example shows that attributes attached to structure fields (or class members) may be lost during compilation preventing certain optimizations.

Assume that we use the “Real_ptr” typedef from Example B to attach attributes to pointers. Instead of passing individual data pointers to a function, we pass them as fields in a struct as shown in the mulsubadd3() function below. When this is done, a compiler will not generate packed data move or arithmetic SIMD instructions, indicating that the attributes were dropped during compilation. This has important implications for large codes, which often pass data pointers encapsulated in structures to functions to improve code readability and maintenance.

```

struct LoopData {
    Real_ptr in1;
    Real_ptr in2;
    Real_ptr out1;
    Real_ptr out2;
    Real_ptr out3;
};

// ...

void muladdsub3(int len, LoopData& d)
{
    for ( int i=0; i<len ; ++I ) {
        d.out1[i] = d.in1[i] * d.in2[i];
        d.out2[i] = d.in1[i] + d.in2[i];
        d.out3[i] = d.in1[i] - d.in2[i];
    }
}

```

```

void muladdsub3a(int len, LoopData& d)
{
    Real_ptr in1 = d.in1;
    Real_ptr in2 = d.in2;
    Real_ptr out1 = d.out1;
    Real_ptr out2 = d.out2;
    Real_ptr out3 = d.out3;

    for ( int i=0; i<len ; ++i ) {
        out1[i] = in1[i] * in2[i];
        out2[i] = in1[i] + in2[i];
        out3[i] = in1[i] - in2[i];
    }
}

```

The function `muladdsub3a()` is the same as `mulsubadd3()` except that we declare local pointer variables (using our typedef) and set them to the fields in the structure. These extra lines of code enable optimal SIMD vectorization.

Example D

This example shows that external functions without proper attributes may disrupt compiler optimizations.

```

double add(double a, double b) __attribute__((pure));

void muladdsub4(int len,
    double* __restrict__ out1,
    double* __restrict__ out2,
    double* __restrict__ out3,
    double* in1, double* in2)
{
    for ( int i=0; i<len ; ++i ) {
        out1[i] = add(in1[i], in2[i]);
        out2[i] = in1[i] + in2[i];
        out3[i] = in1[i] - in2[i];
    }
}

```

Here we have defined an external `add()` function that adds two doubles. The “pure” attribute tells the compiler that the `add()` function has no side effects; e.g., it will not affect the data arrays used in the `muladdsub4()` function. Without the pure attribute, the compiler *must* assume the external function can modify any memory location, disabling its ability to apply optimizations.

Example E

This example shows that attributes attached to pointers passed through the lambda function variable capture mechanism (or as members of a functor) are typically lost during compilation preventing certain optimizations.

```
template < typename LOOP_BODY >
void forall( int begin, int end, LOOP_BODY body )
{
    for ( int i = begin; i < end; ++i ) body( i );
}

//...

void muladdsub5(int len,
                Real_ptr out1, Real_ptr out2, Real_ptr out3,
                Real_ptr in1, Real_ptr in2)
{
    forall( 0, len, [&] (int i) {
        out1[i] = in1[i] * in2[i];
        out2[i] = in1[i] + in2[i];
        out3[i] = in1[i] - in2[i];
    } );
}
```

Here, we use the loop traversal template method from Section 4. The function `muladdsub5()` passes the loop body as a lambda function. Typically, a compiler will generate suboptimal SIMD vector instructions, if any are generated at all, for the loop in this routine. It will create an internal “functor” class/struct to represent the lambda function. As we have noted in Example C, type attributes are not attached to the associated data members/fields, typically.

Actually, we can achieve fully optimal SIMD vectorization with a “fix” that re-establishes type attributes with local variables in the lambda expression, as shown in the function `muladdsub5a()`:

```
void muladdsub5a(int len,
                Real_ptr out1, Real_ptr out2, Real_ptr out3,
                Real_ptr in1, Real_ptr in2)
{
    forall( 0, len, [&] (int i) {
        Real_ptr tout1 = out1;
        Real_ptr tout2 = out2;
        Real_ptr tout3 = out3;

        tout1[i] = in1[i] * in2[i];
        tout2[i] = in1[i] + in2[i];
        tout3[i] = in1[i] - in2[i];
    } );
}
```

This example clearly demonstrates that compilers contain sufficient machinery to optimize abstractions we discuss here, but only if awkward constraints are met. More formal treatment of types and attributes in compilers, and passing all type attributes through all compilation phases, would make it much less onerous for developers to write code that can be highly optimized.

For completeness, the function `muladdsub6()` below shows the loop body represented as a functor class, an object of which is passed to the loop traversal method. This version of the loop has the same optimization issues as the previous lambda function example and good optimization can be achieved using the same sort of local variable technique.

Finally, the functor code in this example illustrates a point that is worth emphasizing. Comparing the functor and lambda versions should make clear that the simplicity of lambda function usage and syntax is a much preferable alternative to the cumbersome code mechanics associated with functor classes.

```
class MULADDSUB_Functor
{
public:
    MULADDSUB_Functor(Real_ptr out1, Real_ptr out2, Real_ptr out3,
                      Real_ptr in1, Real_ptr in2)
    : m_out1(out1), m_out2(out2), m_out3(out3),
      m_in1(in1), m_in2(in2) { ; }

    void operator() (int i)
    {
        m_out1[i] = m_in1[i] * m_in2[i];
        m_out2[i] = m_in1[i] + m_in2[i];
        m_out3[i] = m_in1[i] - m_in2[i];
    }

    Real_ptr m_out1;
    Real_ptr m_out2;
    Real_ptr m_out3;
    Real_ptr m_in1;
    Real_ptr m_in2;
};

//...

void muladdsub6(int len,
                Real_ptr out1, Real_ptr out2, Real_ptr out3,
                Real_ptr in1, Real_ptr in2)
{
    MULADDSUB_Functor kernel(out1, out2, out3, in1, in2);

    forall( 0, len, kernel );
}
```