Trellis: Portability Across Architectures with a High-level Framework

Lukasz G. Szafaryn[+] Todd Gamblin[++] Bronis R. de Supinski[++] Kevin Skadron[+]

[+]University of Virginia
{lgs9a, skadron}@virginia.edu

[++]Lawrence Livermore National Laboratory
{tgamblin, bronis}@llnl.gov

ABSTRACT

The increasing computational needs of parallel applications inevitably require portability across parallel architectures, which now include heterogeneous processing resources, such as CPUs and GPUs, and multiple SIMD/SIMT widths. However, the lack of a common parallel programming paradigm that provides predictable, near-optimal performance on each resource leads to the use of low-level frameworks with architecture-specific optimizations, which in turn cause the code base to diverge and makes porting difficult. Our experiences with parallel applications and frameworks lead us to the conclusion that achieving performance portability requires structured code, a common set of high-level directives and efficient mapping onto hardware.

In order to demonstrate this concept, we develop Trellis, a prototype programming framework that allows the programmer to maintain only a single generic and structured codebase that executes efficiently on both the CPU and the GPU. Our approach annotates such code with a single set of high-level directives, derived from both OpenMP and OpenACC, that is made compatible for both architectures. Most importantly, motivated by the limitations of the OpenACC compiler in transforming such code into a GPU kernel, we introduce a thread synchronization directive and a set of transformation techniques that allow us to obtain the GPU code with the desired parallelization that yields more optimal performance.

While a common high-level programming framework for both CPU and GPU is currently not available, our analysis shows that even obtaining the best-case performance with OpenACC, state-of-the-art solution for a GPU, requires modifications to the structure of codes to properly exploit braided parallelism, and cope with conditional statements or serial sections. While this already requires prior knowledge of compiler behavior the optimal performance is still unattainable due to the lack of synchronization. We describe the contributions of Trellis in addressing these problems by showing how it can achieve correct parallelization of the original codes for three parallel applications, with performance competitive to that of OpenMP and CUDA, improved programmability and reduced overall code length.

General Terms

Parallel Computation, Parallel Frameworks, Parallel Architectures, Loop Mapping

Keywords

Parallel Computation, Parallel Frameworks, Parallel Architectures, Loop Mapping

1. INTRODUCTION

1.1 Need for a Portable Parallel Framework

Parallelizing compilers still fall short in many cases, especially for the hierarchical parallelism that maps best onto GPUs, so parallelization still typically involves manual orchestration of execution and data transfer. For clarity and convenience, many programmers would prefer to parallelize code by using pragmas to annotate a traditional, serial programming language, such as Fortran or C, rather than using new languages or extensions that require them to rewrite the existing code.

While many solutions related to OpenMP [1] or inspired by its form facilitate this approach, none of them are universal. Currently, OpenMP is limited to multi-core CPUs. Although PGI [2] and OpenACC [3] feature directives that describe the two-level parallelism common in accelerators, they are not yet compatible with CPUs and lack a key, local synchronization primitive. The Cetus-based translator [4] achieves CPU-GPU portability, but its underlying OpenMP programming model does not provide directives suited for accelerators. Moreover, none of the above frameworks support codes with braided parallelism and complicated loop structures, a shortcoming that usually results in serialization, even when parallelism is present. The only framework that achieves portability and optimal performance on both multi-core CPU and GPU is OpenCL [5]. However, unlike high-level, directive-based approaches, it requires the use of low-level statements and extensive modifications to the code base. The performance of the high-level directive-based programming approaches must improve before programmers will stop using low-level solutions.

The lack of a convenient, portable, and capable framework forces programmers to use architecture-specific optimizations or to parallelize with low-level and/or non-portable languages, both of which yield non-portable code. Porting such code to a different architecture often results in an inefficient mapping onto hardware and thus suboptimal performance. Even if we manage to port such code across architectures, we are left with multiple semi-optimized and divergent versions, each expensive to maintain when changes are made to the underlying algorithm. In order to avoid this inefficient development cycle, we need a general programming paradigm that is convenient, portable across many computational resources and roughly equivalent in terms of performance.

1.2 Our Approach and Contributions

Our experience with modular programming and existing high-level accelerator frameworks has shown that we already have many fundamental building blocks available to realize the concept of a *portable*, multi-platform, high-level framework. We present Trellis, a prototype programming approach that demonstrates this

by deriving features from these components and integrates them into a single solution. Trellis consists of three features that we find crucial to performance portability: structured code, a single programming framework and code transformations capable of efficiently mapping parallel constructs onto diverse hardware. Each of these techniques alone improves portability, and together they improve the state of the art of portable parallel programming.

We first illustrate portability in terms of code structure. Although proper code structuring that exposes available, often multi-level, parallelism is needed for all parallel frameworks, we emphasize it in the case of Trellis to allow proper application of high-level directives. To increase concurrency, available parallelism should be maximized by combining independent tasks as well as avoiding conditional statements and synchronization, at a potential small cost of redundant computation.

We then demonstrate portability in terms of framework syntax. Drawing on previous work in this area, Trellis derives a single set of directives from both OpenMP and Open ACC, current high-level state-of-the-art solutions for multi-core CPU and GPU, and makes it compatible for both architectures. To implement execution on these architectures, Trellis performs a source-to-source translation of these directives to OpenMP and OpenACC, respectively.

Finally, we illustrate portability in terms of performance. GPUs pose a challenge to efficient hardware mapping by introducing a second hierarchical level of parallelism. While executing target OpenACC code, we identified constructs, such as braided structures, intermediate statements and interleaved serial code, that typically impede efficient mapping. We address this shortcoming in Trellis with prototype implementations of a thread synchronization directive and appropriate code analysis. The functionality of these features is obtained by translating directly to CUDA at the back end.

We apply Trellis to three applications that are representative of computationally intensive tasks: ddcMD (molecular dynamics) [6], Heart Wall (image processing) [7] and B+Tree (database search) [8]. These applications execute at the level of a cluster node, on multi-core CPU and GPU. Our results show that Trellis can achieve correct parallelization of these codes with performance competitive to that of OpenMP and CUDA, improved programmability and reduced overall code length.

Our work on Trellis illustrates that a common high-level framework is feasible and it can support efficient execution across heterogeneous resources. We discuss the benefits of this approach in terms of portability, performance, programmability and maintenance. Our GPU-based lessons should generalize to other accelerators with hierarchical structure. We make the following contributions:

- Providing a common set of directives, derived from the current state-of-the-art solutions, that are portable across architectures.
- Illustrating sufficient descriptive capability of directives to support efficient, portable parallelization of codes, thus obviating the need for low-level solutions.
- Complementing the OpenACC-derived paradigm with a thread synchronization directive and code analysis to enable more efficient mapping of kernels onto GPU hardware.
- Source-to-source translation of the common code base to OpenMP for multi-core CPU and OpenACC, augmented with CUDA for new transformations, for GPUs.

The remainder of this paper is organized as follows. Section 2 gives an overview of related work. Section 3 described the three applications that we analyze. Section 4 provides details of the Trellis framework. Section 5 gives an overview of our setup and methodology. Section 6 describes the performance and code length of different versions of our applications. Section 7 discusses the benefits of Trellis from the perspective of applications and architectures. This paper is best viewed in color.

2. RELATED WORK

Early frameworks for multicore CPUs, such as Cilk [9], use library functions to automate parallelization of tasks, such as loops and reductions. Later libraries such as TBB [10] also abstract many aspects of thread management. Unlike our approach, these solutions target specific functionality, require changes to the existing code, use parallelizing algorithms tailored to CPUs and lack features required for describing multilevel parallelism. Alternatively, OpenMP facilitates parallel execution on multicore CPUs via high-level directives that annotate the unchanged base code. However, in spite of its convenient form, its small set of directives only support single-level shared memory parallelism. Trellis uses OpenMP as a translation target for implementing CPU execution, as well as a point of reference.

The most popular native GPU framework, CUDA, facilitates efficient execution with GPUs' hierarchical parallel structure and internal memory. However, it explicitly orchestrates offloading of kernels, which in turn increases the learning effort and the amount of repetitive coding. PGI CUDA-x86 [11] and gpuOcelot [12] make CUDA portable to CPUs by compilation of the original code or translation of its assembly form, respectively. OpenCL extends CUDA's approach to support for both CPUs and GPUs portably. However, it also orchestrates kernel execution through a lower-level API. Similarly to these native frameworks, our solution provides more control over computational resources. However, we provide an orthogonal high-level directive-based approach that is convenient and portable.

The high-level APIs of Jacket [13] and ArrayFire [14], which are compatible with common CPU and GPU languages, offload common data-parallel operations to a GPU. These approaches combine multiple functions, with the corresponding data transfers into a single kernel. Although these solutions are convenient, they only target specific arithmetic functions and do not provide a generic programming framework.

The Cetus-based translator [4] converts OpenMP code to CUDA for execution on a GPU. However, its performance is limited by the efficiency of determining data transfer and kernel mappings that the original OpenMP code does not explicitly describe.

Flexible application-specific programming interfaces [15] [16] support heterogeneous platforms as well as algorithmic primitives developed for GPUs [17]. However, these are based on libraries that implement only specific algorithms and functionality while our approach provides a general, portable solution.

Both PGI [2] and its successor, OpenACC [3], adapt the high-level directive-based approach of OpenMP to accelerators by adding optional directives for describing multi-level parallelism. While similar in form and functionality to our approach, these solutions do not yet support CPUs. A future OpenMP interface [1] is expected to introduce accelerator directives, similar to those in OpenACC, in addition to its current support for CPUs. This solution may eventually become roughly equivalent to our

approach, so this paper formally evaluates the benefits of that possibility. Trellis uses OpenACC as a translation target for implementing GPU execution, as well as a point of reference.

In our prior, non-archival, workshop paper [18], we presented techniques for achieving structural and syntactic portability by maintaining a common code structure and using a common high-level directive-based framework with annotations derived from OpenMP and OpenACC. This paper extends that work by demonstrating how performance portability can be achieved for a GPU with additional synchronization directive and code analysis. We now present all of these techniques together in a single programming approach called Trellis.

3. OVERVIEW OF APPLICATIONS

3.1 ddcMD

The ddcMD application [6] calculates particle potential and relocation due to mutual forces between particles within a large 3D space. This space is divided into cubes, or *large boxes*, that are allocated to individual cluster nodes (Figure 1). The large box at each node is further divided into cubes, called *boxes*. Twenty-six *neighbor boxes* surround each box (the *home box*). Home boxes at the boundaries of the particle space have fewer neighbors. Particles only interact with those other particles that are within a cutoff radius, since those at larger distances exert negligible forces. Thus the box size *s* is chosen so that the cutoff radius does not span beyond any neighbor box for any particle in a home box, thus limiting the reference space to a finite number of boxes.

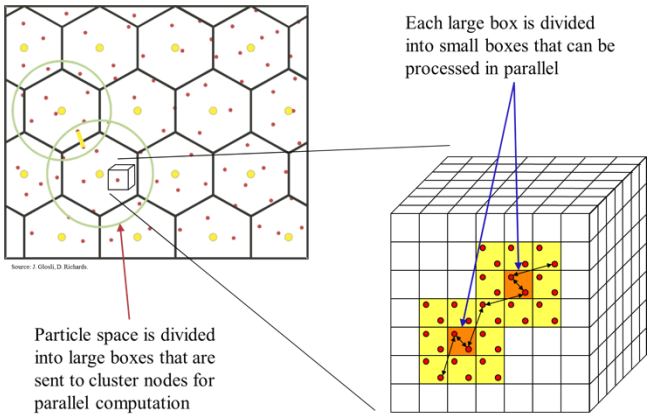


Figure 1. Partitioning of computation in ddcMD application. For every particle in an orange area, interactions with all particles in the surrounding yellow area are calculated.

Figure 2 shows the ddcMD code structure that executes on a node. The code has two groups of nested loops enclosed in the outermost loop, which processes home boxes. The processing of home boxes is independent and can proceed in parallel. For any particle in the home box, the 1st and 2nd groups of nested loops compute interactions with other particles in the home box and particles in all neighbor boxes, respectively. The processing of each particle consists of a single calculation stage in the innermost loop. The code is characterized by embarrassing parallelism, since the processing at both, home box and particle, levels can proceed in parallel within their own scope.

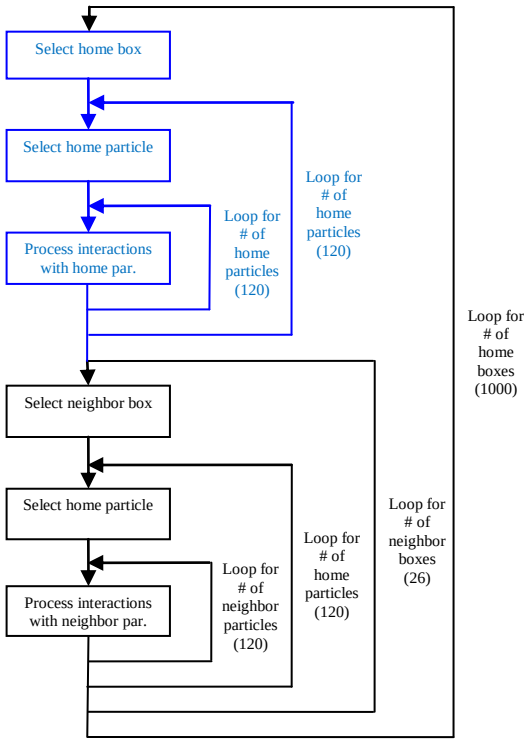


Figure 2. Original ddcMD code structure. Section in blue is merged with the remaining code into a braided structure to maximize parallelism.

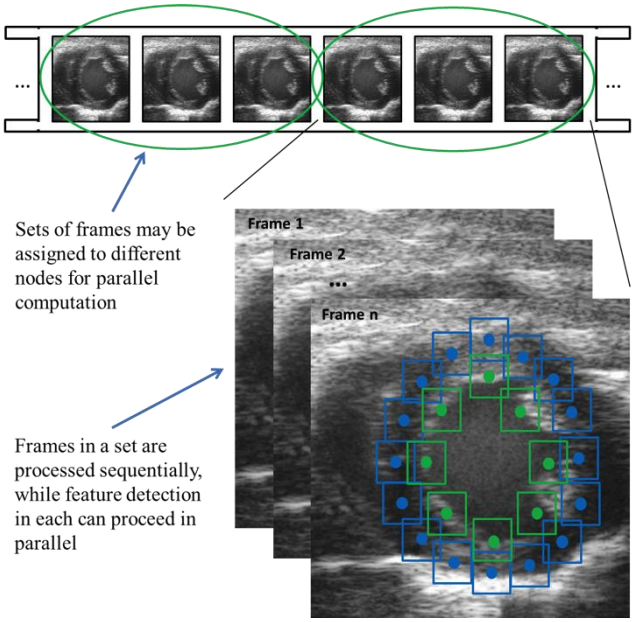


Figure 3. Partitioning of computation in Heart Wall application. Displacement of areas marked with blue and green squares is tracked throughout a frame sequence.

3.2 Heart Wall

The Heart Wall application [7] tracks the movement of a mouse heart over a sequence of 609x590 ultrasound frames (images) to observe response to a stimulus. Images are arranged into batches and offloaded to individual nodes for parallel processing (Figure 3). In the initial stage, not included in the diagram, the program performs image processing operations on the first frame in a batch

to detect initial, partial shapes of inner and outer heart walls and place sample points on them. The core of the application tracks the movement of heart walls by detecting displacement of image areas under sample points as the shape of heart walls changes throughout the remaining frames. Green and blue dots in Figure 3 indicate sample points that mark inner and outer heart walls.

Figure 4 shows the Heart Wall code structure that executes on a node. The code has two groups of nested loops enclosed in the outermost loop, which processes frames. The processing of frames is dependent and needs to proceed sequentially. The first and second groups of loops track features around sample points on inner and outer heart walls. The processing of each sample point, enclosed by the middle loop, consists of several sequentially dependent tracking stages interleaved by control statements. The code is characterized by braided parallelism, since the processing of inner and outer points can proceed in parallel.

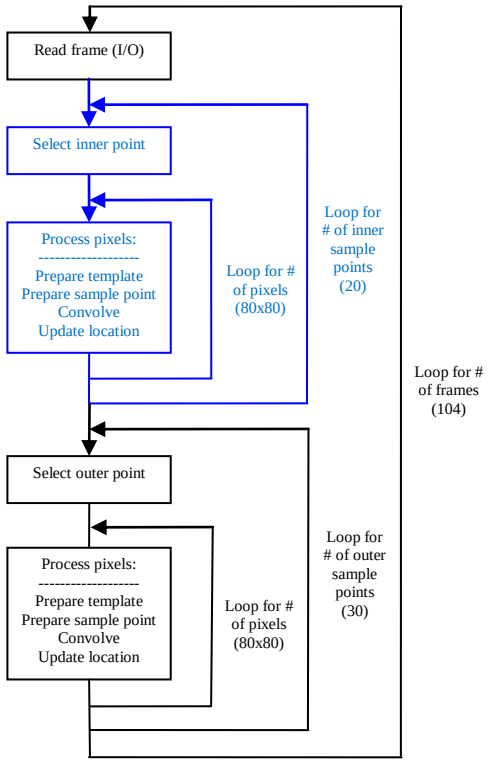


Figure 4. Simplified structure of the original Heart Wall code. Section in blue is merged with the remaining code into a braided structure to maximize parallelism.

3.3 B+Tree

The B+Tree application [8] performs search queries to a database. The pool of all requested queries is divided into groups, each offloaded to a node. The database is organized as a B+tree, optimized for efficient insertion and retrieval of records. The tree consists of many levels with multiple leaves at each level. A key value defines the database search criterion. The search proceeds down the tree structure across all levels to find leaf values that satisfy the search criterion (Figure 5). The response to the query is then created based on the individual search results.

Figure 6 shows the B+Tree code structure that executes on a node. The code has one group of nested loops enclosed in the outermost loop that processes individual queries. The processing of queries is independent and can proceed in parallel. The middle loop traverses levels of the data tree structure while the innermost loop compares the value of each leaf at every tree level to the key. The code is characterized by braided parallelism, since the processing of different queries can performed in parallel.

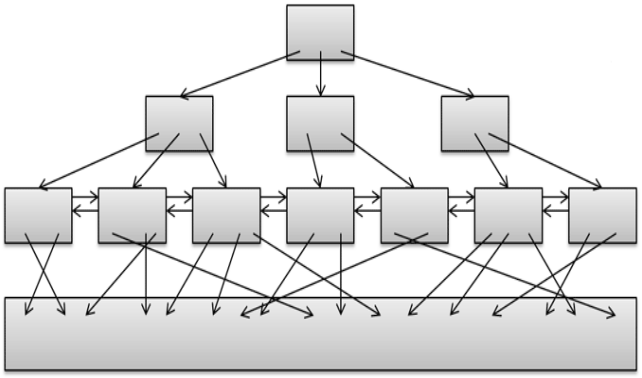


Figure 5. Partitioning of computation in the B+Tree application. The search proceeds down the tree structure consisting of levels and leaves.

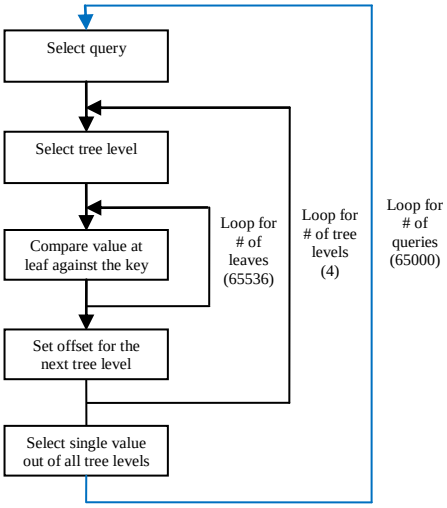


Figure 6. Original B+Tree code structure. Section in blue represents processing of different queries combined into a braided structure to maximize parallelism.

4. TRELLIS

4.1 Code Structure

Our previous work demonstrates that performance portability requires proper file organization and code order when manually transitioning between diverse frameworks by replacing relevant code sections. Trellis eliminates this requirement as it allows maintaining a single portable code base with generic structure where the available, often multi-level, parallelism can be clearly exposed to annotation for heterogeneous resources. When targeting parallel resources, parallelism should be maximized by grouping similar or different independent operations, usually expressed by individual loops, into a uniform or braided structure. Also, for GPUs, long tasks can be split to increase resource occupancy and to decrease register file footprint. As much as possible, conditional statements and synchronization should be avoided at the minimal cost of redundant computation to improve lock-step SIMD execution flow.

4.2 Common Framework

Trellis derives and extends a unified set of directives from OpenMP [1] and OpenACC [3], current high-level state-of-the-art solutions for multicore CPU and GPU, and makes it compatible for both architectures. To implement execution on these architectures, Trellis performs a source-to-source translation of these directives to OpenMP and OpenACC, respectively. For improved performance, it features a new thread synchronization directive and new code analysis (Section 4.3A) with functionality implemented via translation to CUDA [19]. The set of directives adopted by Trellis consists of those common for both CPU and GPU with specific clauses for the latter, as well as those that are specific to a GPU. In the remainder of this section we give only a short overview of annotations adapted by Trellis, and ask the reader to refer to [1] and [2] for more detailed information on this programming style. The naming for our annotations is derived mostly from [2].

Trellis uses annotations that consist of a directive and a corresponding clause (Figure 7). There are four types of **directives**: region, loop, data and executable (Figure 8). Data directives can only exist in combination with a region or loop directive that defines their scope. **Clauses** specify values that can refer to the number of tasks, vector widths, and the names and sizes of variables. While all directives can annotate the base code, the Trellis translator uses only those that are appropriate for the target framework (Section 4.4). Consider codes in Figures Figure 14, Figure 16 and Figure 18 as examples.

#pragma trellis directive (clause)

Figure 7. Directive format in Trellis framework.

Region directives *data* and *compute* encapsulate code regions in which data is exchanged with the GPU or in which computation is offloaded to the GPU, respectively. The *data* directive indicates that data should remain on the device for the duration of enclosed kernels. The *compute* directive instructs the compiler to try to offload the enclosed code region to an accelerator, regardless of whether particular parallel constructs, such as loops, are present.

Loop directives such as *parallel*, *vector* or *sequential*, specify execution flow for the annotated loop. The *parallel* directive is used for coarse-grained mapping onto cores in a CPU or multiprocessors in a GPU. The *vector* directive, on the other hand, is used for fine-grained mapping onto processing units in a CPU. In the case of a CPU, equivalent mapping onto vector units is performed implicitly by the compiler. Since the compiler can try to implicitly parallelize loops, even when no annotations present, sequential execution can be enforced via the *sequential* directive.

Data directives, such as *shared*, *private*, *cache*, *local*, *copy*, *copyin* or *copyout*, specify the types of data used in annotated code sections. The first two directives can be used for specifying the scope of variables for both CPU and GPU. The remaining annotations are used for caching and allocating space in GPU memory, as well as transferring data between GPU and system memories.

Executable directives support operations, such as device setup, as well as explicit thread and data synchronization, that are not associated with a particular construct but an implicit code region in which they appear. The GPU thread synchronization directive that is introduced in Trellis (Section 4.3A) is an example of this type of directive.

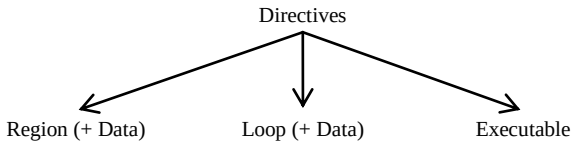
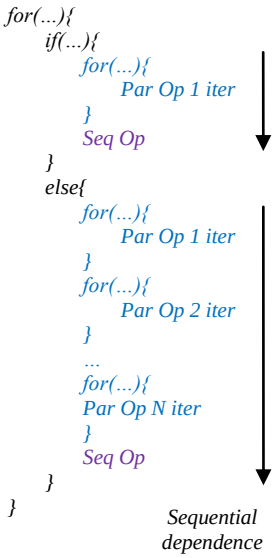


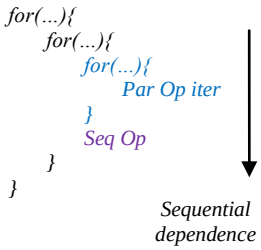
Figure 8. Types of directives in Trellis framework.

4.3 Efficient Mapping to GPUs

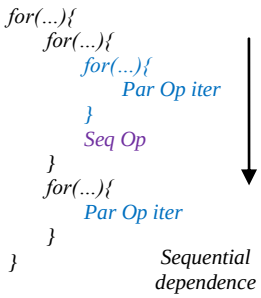
While running target OpenACC code, generated by Trellis at the back end, which is then further translated to CUDA by the OpenACC compiler, we observe that performance is suboptimal when compared to hand-written CUDA code. We discover that the desired hardware mapping (which is also the most optimal mapping in the case of our codes), specified by explicit annotations, cannot be obtained with OpenACC because it lacks support for thread synchronization and inefficiently handles code structure. This behavior is illustrated in the case of three main code constructs that we identify as braided loops, intermediate statements and interleaved serial code (Figure 9), discussed in detail below.



a) Heart Wall



b) ddcMD



c) B+Tree

Figure 9. Code structure of our applications.

The penalties of these shortcomings can be alleviated by first analyzing OpenACC compiler's behavior via trial and error and then, often unintuitively, modifying the source code only to obtain more efficient mapping. However, the optimal performance still cannot be obtained due to the lack of thread synchronization support. Since the aforementioned constructs appear in codes frequently, this is a major obstacle in achieving performance portability of a single code base. Therefore, we address this problem in Trellis with an introduction of a thread synchronization directive and required code transformations to achieve proper hardware mapping. The functionality of these is implemented at the back end via source-to-source translation directly to CUDA. Consider representative C-style outlines of kernels in our applications (Figure 9. Code structure of our applications.).

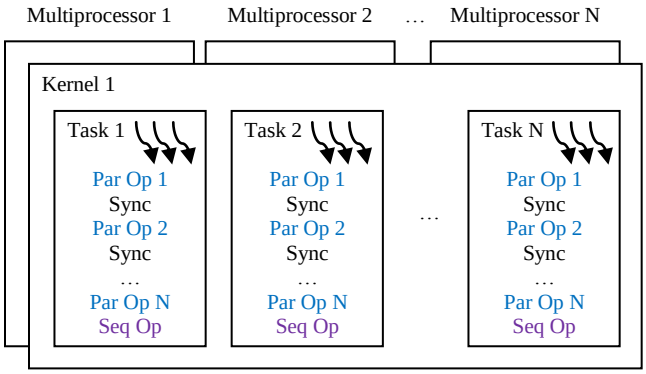


Figure 10. Efficient mapping of braided code onto GPU.

A. Braided Structure

We first focus on a braided code structure, such as that in Heart Wall (Figure 9a, disregard conditional statements). The outermost loop represents independent tasks that can execute in parallel, where each task consists of multiple operations that execute sequentially due to dependencies (black arrow). These operations are either parallel (marked blue) or scalar (marked purple) within their own scope. In order to achieve the best performance (highest occupancy and locality), the two-level braided structure should be mapped to the corresponding two-level hierarchy in a GPU (Figure 10) where tasks can take advantage of hardware synchronization for the sequentially dependent operations. The compiler should determine the optimal vector width, common for all operations.

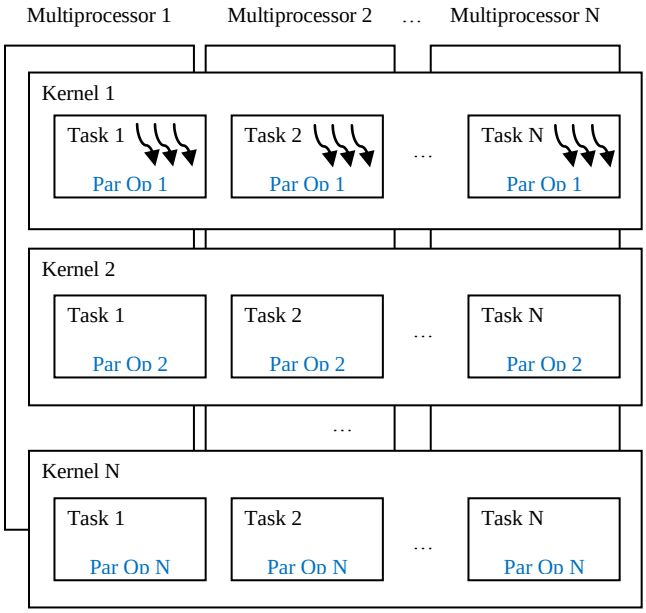


Figure 11. Inefficient mapping of braided code onto GPU due to the lack of synchronization support that results in a split kernel.

We observe that OpenACC cannot effectively map a braided structure most likely due to lack of support for thread synchronization within a GPU multiprocessor. In such case, synchronization can only be invoked via a global barrier at the end of the kernel. Therefore, as a result, each iteration of the loop corresponding to tasks is combined with those of each sequentially dependent operation into a separate kernel and spread across all multiprocessors (with no particular task-multiprocessor assignment, Figure 11). Most importantly, code offloaded to a GPU in such a way suffers from the overhead of additional kernel launches that could diminish the benefits of parallel execution [20]. Moreover, while this mapping tries to increase occupancy, it often suffers in terms of locality.

Following the approach of explicit annotation (Section 4.4), rather than relying on compiler dependence analysis, we address this shortcoming in Trellis with an introduction of a *threadSynchronize* directive (Figure 16, Figure 18) that allows the programmer to invoke thread synchronization explicitly. We then develop code analysis that takes advantage of this directive to construct a proper braided kernel as show in Figure 10.

B. Intermediate Statements

Let us now consider a braided code with intermediate statements such as conditionals (Heart Wall, Figure 9a) or loops (ddcMD, Figure 9b and B+Tree, Figure 9c) that add slight complexity to its structure. These statements can enclose individual operations, sets of operations (Heart Wall, B+Tree) or entire contents of a task (ddcMD). Similarly to Section 4.3A, in order to achieve optimal performance, all tasks with their contents (including the intermediate statements) should be executed in a single kernel (Figure 10), or at least as shown in Figure 11 (assuming limitation described in Section 4.3A). Although less deterministic, common vector width could be determined in case of a conditional statement.

We observe that OpenACC cannot yet efficiently map code structure that includes intermediate statements. The inclusion of the conditional statement or the additional loop (which could potentially be eliminated by the compiler via unrolling) currently prevents the compiler from offloading the code, even in the form shown in Figure 10 by breaking into multiple kernels. As a result, in the case of Heart Wall and ddcMD, operations within each task are serialized while tasks themselves are spread across multiprocessors (Figure 12). This mapping suffers from significant performance penalty due to serialization. In the case of B+Tree, on the other hand, the parallel region is not offloaded to a GPU, but executes in a CPU instead.

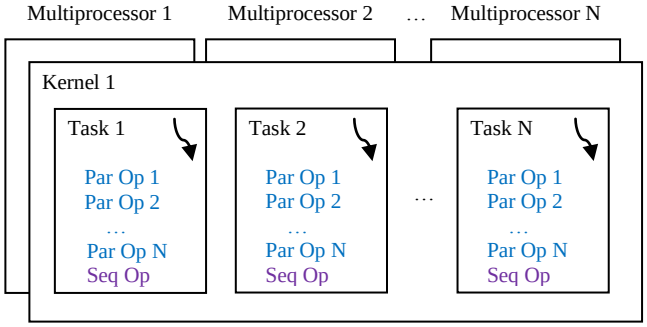


Figure 12. Inefficient mapping of a braided code onto GPU due to intermediate statements and interleaved sequential code that results in task serialization.

Trellis addresses this shortcoming with code analysis and transformations that can properly offload such code structure, as shown in Figure 10. The OpenACC compiler can be aided in achieving a more efficient mapping, such as the one in Figure 11, by moving conditional statements outside of the two-level loop structure and merging the intermediate loop with one of the

remaining loops. We make this manual modification to show the best-case OpenACC performance.

C. Interleaved Serial Code

Finally, we consider a braided code structure with parallel and sequential operations interleaved inside a task (Heart Wall, Figure 9a, ddcMD, Figure 9b, B+Tree, Figure 9c). The sequential operations usually represent initial, intermediate or final computation steps within a task (marked purple) that narrow the effective vector width to one thread. Similarly to Sections 4.3A and 4.3B, in order to achieve the best performance, all tasks with their contents should be executed in a single kernel (Figure 10), or at least as shown in Figure 11, while restricting execution of a serial section to a single thread.

We observe that OpenACC cannot efficiently map structures that include interleaved serial code, since unlike loops, it cannot be intuitively mapped onto a parallel resource. As a result, similarly to examples in Sections 4.3A and 4.3B, the remaining (parallel) operations within each task are serialized while the processing of tasks is spread across multiprocessors with no particular task-multiprocessor assignment that explores locality (Figure 12). This mapping suffers from significant performance penalty due to the serialization.

Trellis addresses this shortcoming with an appropriate code analysis and transformation that can offload this type of code structure as shown in Figure 10. The OpenACC compiler can be aided in achieving a more efficient mapping, such as the one in Figure 11, by enclosing interleaved serial code in a loop of unit width. We make this manual modification to show the best-case OpenACC performance.

4.4 Trellis Translator

Trellis translator implements execution of portable Trellis codes on multi-core CPU and GPU. This is achieved via source-to-source translation of C code annotated with Trellis directives (Figure 13a) to OpenMP and OpenACC (Figure 13b), respectively. Since Trellis directives are a subset of those derived from these two solutions, the translation process is straightforward. The additional synchronization directive and code transformations for improved GPU performance (Section 4.3) are implemented via translation to CUDA. Although straightforward as well, the process of translating to CUDA structure and syntax is beyond the scope of this paper. Trellis translator performs the following steps:

1) Determine target framework based on user input;
2) Perform code analysis with respect to loop structure;
3) Translate code to target CPU or GPU framework;

In its prototype form, unlike OpenACC, Trellis does not attempt to parallelize unannotated code, but it relies entirely on annotations. This approach is sufficient for the purpose of this work, as we are concerned with proper mapping of a fully annotated code rather than parallelization techniques via dependence analysis in the case of an unannotated or partially annotated code. Moreover, unlike OpenACC, Trellis does not split the inner loop iterations and map across multiprocessors for improved occupancy. This turns out to be problematic in terms of locality when a proper braided code is generated by Trellis. From all annotations present in the code, only applicable ones are used when generating kernel code for a CPU or a GPU. Also, for simplicity in this prototype version, Trellis expects certain ordering of variable declaration, memory allocation and computation code sections. The translator is available on the first author's webpage.

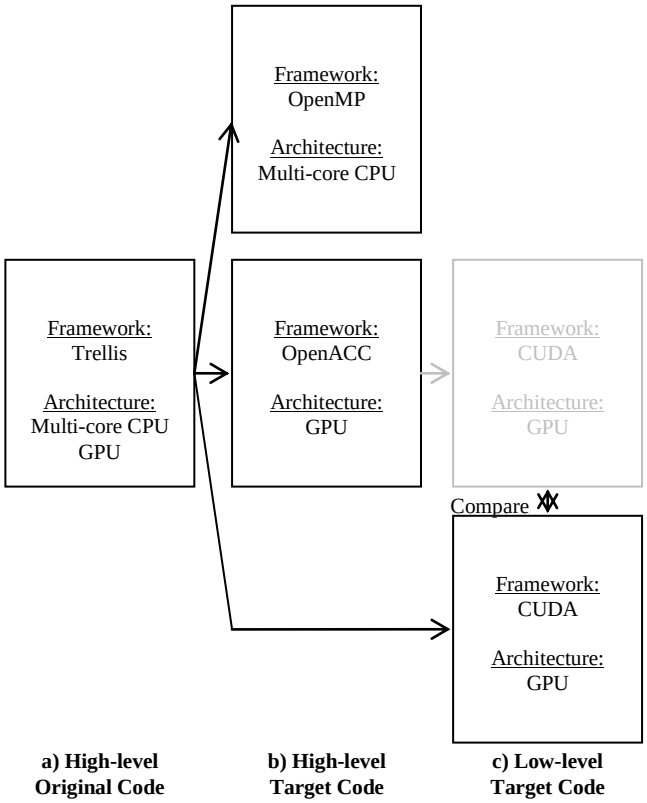


Figure 13. Translation of Trellis code to target frameworks.

5. SETUP AND METHODOLOGY

The baseline codes for our applications are written in C. They are extended with the Trellis framework, which is then translated to target frameworks that include: OpenMP, OpenACC and CUDA.

We configure ddcMD with 1000 small boxes with 120 particles in each. The Heart Wall application processes 104 video frames, 609x590 pixels each, with 20 inner and 30 outer sample points, 80x80 pixels each. B+Tree runs 65000 queries and its database is configured with 4 tree levels, each with 65536 leaves 128 keys.

Analysis and performance results are obtained from a single machine, equivalent to a cluster node, equipped with an 8-core (2 threads/core) Intel Xeon X5550 CPU and NVIDIA Tesla C2050 GPU. We compile C-only and OpenMP codes with GCC [21], using the highest optimization level. OpenACC codes are handled by PGCC [2], version 12.5, which translates them to CUDA that is further compiled by NVCC [19].

We try to ensure that OpenACC and CUDA codes generated by Trellis are equivalent in terms of memory usage, so that the difference in performance is related only to hardware mapping, kernel launch overhead, resource utilization and locality. We ensure that such mapping, that we compare against, is the most optimal for each application. For a fair line count comparison, we split compound annotations to include only one directive per line.

We choose the PGI compiler for OpenACC, as it appears to be the most mature option. Since the compiler does not allow examining the CUDA (Figure 13c) that it generates at the back end, we infer

its structure based on compiler messages and comparative performance. We use the hand-written CUDA code to estimate typical GPU overheads due to driver communication, data transfer and kernel launch.

We are only concerned with tasks processed at each node. Thus, we assume that the higher level code balances the amount of work at each node. Diagrams, code outlines and line counts for each application refer to the accelerated section of the code, which accounts for nearly all of execution time.

We refer to NVIDIA terminology [19] when describing GPU optimizations. Our results do not account for the overhead of source-to-source translation, which would be minimized if Trellis was supported natively by the compiler.

6. RESULTS

6.1 ddcMD

In order to prepare for the application of our framework, we first make several changes to the original code structure that include the following.

- **Increase Parallelism**: Since the processing of home and neighbor box interactions is almost identical, we combine the corresponding code and data structures to expose more parallelism (marked blue in Figure 2).
- **Improve Convergence**: We remove conditional statements related to the cut-off radius in order to improve convergence of GPU threads and data accesses at the cost of a small amount of additional work.
- **Simplify References**: To support GPU transfers, we consolidate box and particle data that originally used multiple structures referenced through nested pointers.
- **Remove Customization**: We remove optimizations, such as the explicit use of particular ISA instructions and memory padding, at insignificant performance cost.

We then apply our framework to the structured code as follows to arrive at a portable version (Figure 14).

- **Hardware Mapping**: We map box processing to CPU cores or GPU multiprocessors and parallelize particle processing across CPU vector units or GPU processors.
- **Execution Parameters**: To match computation size to available resources in GPU implementations, we set the number of blocks for tasks to 1000 and the vector width for operations to 128.

The portable Trellis code is translated to OpenMP or OpenACC for CPU or GPU execution, respectively. Since the annotations in the two follow from the relevant ones in Trellis, we do not show them in separate code listings, but only mark the relevant Trellis annotations with colors in Figure 14. The listing of target CUDA code that fully utilizes Trellis annotations is omitted because of being beyond the scope of the paper.

We make the following changes to OpenACC code to achieve the best-case performance (Figure 15).

- **Intermediate Loop**: We remove two sequential loops from the kernel (marked purple in Figure 15). However, the resulting implementation is still far from optimal as it narrows the scope of offloaded code (to two innermost loops) and increases the number of kernels.

Table 1 compares performance of ddcMD implementations. In our results, *Original* corresponds to the baseline sequential version, usually with custom optimizations, while *Structured* refers to the generalized parallel version of the *Original* code. These results illustrate code length and performance achievable with OpenMP (OMP) and OpenACC (OACC). *Structured Portable* code, on the other hand, is a *Structured* code that uses Trellis to achieve portability across architectures. These results show the code length and performance achievable with our portable high-level approach. Line 12 gives additional initialization and data transfer overhead incurred by GPU codes, as extracted from the hand-written CUDA code. Line 13 gives the best-case OpenACC performance, achievable with structural modifications to the source code. This number illustrates detriment to performance due to the lack of synchronization support (Section 4.3A). We normalize speedup to the OpenMP code that runs on a single core.

```
Conversion of indices to partitioned space
#pragma trellis data copyin(box[0:#_boxes]) \
                     copyin(pos[0:#_par.]) \
                     copyin(chr[0:#_par.]) \
                     copyout(dis[0:#_par.]){
    #pragma trellis compute{
        #pragma trellis parallel(1000) \
                         independent \
                         private(...) \
                         cache(home_box)
        for(i=0; i<#_home_boxes; i++){
            Home box setup
            #pragma trellis sequential
            for(j=0; j<#_neighbor_boxes; j++){
                Neighbor box setup
                #pragma trellis vector (128) \
                                 independent \
                                 private(...) \
                                 cache(neighbor_box)
                for(k=0; k<#_home_particles; k++){
                    #pragma trellis sequential
                    for(l=0; l<#_neighbor_particles;l++){
                        Calculation of interactions
                    }
                ...
}
```

Figure 14. Outline of portable ddcMD code, written in Trellis. During translation, statements in blue apply to both OpenMP and OpenACC, while those in brown only to OpenACC. Names of some directives change during translation.

```
Conversion of indices to partitioned space
#pragma acc data copyin(box[0:#_boxes]) \
                 copyin(pos[0:#_par.]) \
                 copyin(chr[0:#_par.]) \
                 copyout(dis[0:#_par.]){
    for(i=0; i<#_home_boxes; i++){
        Home box setup
        for(j=0; j<#_neighbor_boxes; j++){
            Neighbor box setup
            #pragma acc kernels{
                #pragma acc loop gang \
                                 independent \
                                 private(...) \
                                 cache(neighbor_box)
                for(k=0; k<#_home_particles; k++){
                    #pragma acc loop vector \
```

```
                    independent
            for(l=0; l<#_neighbor_particles;l++){
                Calculation of interactions
            }
        …
}
```

Figure 15. Outline of portable ddcMD code translated to OpenACC. Statements in blue represent structural changes required to achieve best-case OpenACC performance.

Table 1. Performance of ddcMD application.

	Arch.	Code Feature	Framework	Kernel Length [lines]	Exec. Time [s]	Speedup [x]
1	1-core CPU	Original	C	62	60.53e-1	1.52
2	1-core CPU	Structured	OMP	58	73.24e-1	1.00
3	8-core CPU	Structured	OMP	**65**	10.51e-1	**6.97**
4	GPU	Structured	OACC	**97**	91.62e-1	0.80
5	GPU	Structured	CUDA	88	5.59e-1	13.10
6				**162**		**6.97**
7	1-core CPU	Structured Portable	Trellis (-> OMP)	97	73.24e-1	1.00
8	8-core CPU	Structured Portable	Trellis (- > OMP)	97	10.51e-1	6.97
9	GPU	Structured Portable	Trellis (- > OACC)	97	91.62e-1	0.80
10	GPU	Structured Portable	Trellis (-> CUDA)	**97**	5.59e-1	**13.10**
11				**97**		**13.10**
12	GPU	Init/Trans Overhead	CUDA	---	3.36e-1	---
13	GPU	Modified Best-case	OACC	89	8.24e-1	8.88

6.2 Heart Wall

In order to prepare for the application of our framework, we first make several changes to the original code structure that include the following.

- **Increase Parallelism**: Since the processing of inner and outer points is almost identical, we combine the corresponding code and data structures to expose more parallelism (marked blue in Figure 4).
- **Improve Locality**: Since point-processing stages can share data, we arrange data accesses to maximize utilization of cache and shared memory in CPU and GPU, respectively.

We then apply our framework to the structured code as follows to arrive at a portable version (Figure 16).

- **Hardware Mapping**: We map sample point processing to CPU cores or GPU multiprocessors and parallelize processing of detection stages across CPU vector units or GPU processors.
- **Execution Parameters**: To match computation size to available resources in GPU implementations, we set the number of blocks for tasks to 50 and the vector width for operations to 512.
- **Thread Synchronization**: We instruct Trellis translator to invoke thread synchronization between sequentially dependent operations via new directive introduced in Trellis.

The portable Trellis code is translated to either OpenMP or OpenACC for CPU or GPU execution, respectively (relevant sections are marked in color in Figure 16).

We make the following changes to OpenACC code to achieve the best-case performance (Figure 17).

- **Conditional Statements**: We split code on each conditional statement (not shown in Figure 4 and Figure 9a for clarity) into groups of operations to avoid serialization.
- **Interleaved Sequential Code**: We enclose interleaved sequential code in a loop of unit width (purple in Figure 17) to enable proper parallelization of the outer loop.

Table 2 compares performance of Heart Wall implementations.

```
Processing of inputs from earlier stages.
for(i=0; i<#_frames; i++){
    Read frame
    #pragma trellis data copyin(frm[0:frm_siz]) \
                         copyin(ini_loc[0:#_smp_pnts.]) \
                         local(con/cor[0:#_pixels]) \
                         copyout[fin_loc[0:#_smp_pnts.]){
        #pragma trellis compute{
            #pragma trellis parallel(50) \
                             independent \
                             private
            for(j=0; j<#_sample_points; j++){
                #pragma trellis vector(512) \
                                 independent \
                                 private(…)
                for(i=0; i<#_pixels; i++){
                    Preparing inputs
                }
                #pragma threadSynchronize
                #pragma trellis vector(512) \
                                 independent \
                                 private(…)
                for(i=0; i<#_pixels; i++){
                    Convolving/correlating with templates
                }
                #pragma threadSynchronize
                Updating displacement
            …
}
```

Figure 16. Outline of portable Heart Wall code, written in Trellis. During translation, statements in blue apply to both OpenMP and OpenACC, those in brown only to OpenACC, while those in purple to CUDA. Names of some directives change during translation.

```
Processing of inputs from earlier stages.
for(i=0; i<#_frames; i++){
    Read frame
    #pragma acc data copyin(frm[0:frm_siz]) \
                     copyin(ini_loc[0:#_smp_pnts.]) \
                     create(con/cor[0:#_pixels]) \
                     copyout[fin_loc[0:#_smp_pnts.]){
        #pragma acc kernels{
            #pragma acc loop gang \
                             independent
```

```
    for(j=0; j<#_sample_points; j++){
        #pragma acc loop vector \
                                independent \
                                private(...)
        for(i=0; i<#_pixels; i++){
            Preparing inputs
        }
        #pragma acc loop vector \
                                independent \
                                private(...)
        for(i=0; i<#_pixels; i++){
            Convolving/correlating with templates
        }
        #pragma acc loop vector \
                                independent \
                                private(...)
        for(i=0; i<1; i++){
            Updating displacement
        }
        ...
}
```

Figure 17. Outline of portable Heart Wall code translated to OpenACC. Statements in blue represent structural changes required to achieve best-case OpenACC performance.

Table 2. Performance of Heart Wall application.

	Arch.	Code Feature	Framework	Kernel Length [lines]	Exec. Time [s]	Speed-up [x]
1	1-core CPU	Original	C	178	11.76e1	0.88
2	1-core CPU	Structured	OMP	183	10.35e1	1.00
3	8-core CPU	Structured	OMP	**197**	2.01e1	**5.15**
4	GPU	Structured	OACC	**205**	42.12e1	0.25
5	GPU	Structured	CUDA	202	1.21e1	8.55
6				**402**		**5.15**
7	1-core CPU	Structured Portable	Trellis (-> OMP)	209	10.35e1	1.00
8	8-core CPU	Structured Portable	Trellis (- > OMP)	209	2.01e1	5.15
9	GPU	Structured Portable	Trellis (- > OACC)	209	42.12e1	0.25
10	GPU	Structured Portable	Trellis (-> CUDA)	**209**	1.21e1	**8.55**
11				**209**		**8.55**
12	GPU	Init/Trans Overhead	CUDA	---	0.03e1	---
13	GPU	Modified Best-case	OACC	223	1.49e1	6.95

6.3 B+Tree

In order to prepare for the application of our framework, we first make several changes to the original code structure that include the following.

- **Simplify References**: To support GPU transfers, we consolidate data for queries and trees that originally used multiple structures referenced by nested pointers.

We then apply our framework to the structured code as follows to arrive at a portable version (Figure 18).

- **Hardware Mapping**: We map the processing of queries to CPU cores or GPU multiprocessors and parallelize the processing of leaves across CPU vector units or GPU processors.
- **Execution Parameters**: To match computation size to available resources in GPU implementations, we set the number of blocks for tasks to 65000 and the vector width for operations to 512.
- **Thread Synchronization**: We instruct Trellis translator to invoke thread synchronization between sequentially dependent operations via new directive introduced in Trellis.

The portable Trellis code is translated to either OpenMP or OpenACC for CPU or GPU execution, respectively (relevant sections are marked in color in Figure 18).

We make the following changes to OpenACC code to achieve the best-case performance (Figure 19).

- **Intermediate Loop**: We move the middle loop beyond the outermost loop to help the compiler recognize braided structure.
- **Interleaved Sequential Code**: We enclose interleaved sequential code in a loop of unit width (purple in Figure 19) to enable proper parallelization of the outer loop.

Table 3 compares performance of B+Tree implementations.

```
Construct data tree with multiple levels and leaves
#pragma trellis data copyin(records[0:#_records]) \
                     copyin(knodes[0:#_knodes.]) \
                     copyin(keys[0:#_queries.]) \
                     copyout(ans[0:#_queries.]){
    #pragma trellis compute{
        #pragma trellis parallel(65000) \
                         independent \
                         private
        for(i=0; i<#_queries; i++){
            #pragma trellis sequential
            for(i=0; i<#_tree_levels; i++){
                #pragma trellis vector(512) \
                                 independent \
                                 private(...)
                for(j=0; j<#_leaves; j++){
                    Compare value against the key
                }
                #pragma threadSynchronize
                Set offset for the next tree level
            }
            #pragma threadSynchronize
            Select single value out of all tree levels
        ...
}
```

Figure 18. Portable B+Tree code, written in Trellis. During translation, statements in blue apply to both OpenMP and OpenACC, those in brown only to OpenACC, while those in purple to CUDA. Names of some directives change during translation.

```
Construct data tree with multiple levels and leaves
#pragma acc data copyin(records[0:#_records]) \
                 copyin(knodes[0:#_knodes.]) \
                 copyin(keys[0:#_queries.]) \
                 copyout(ans[0:#_queries.]){
    for(i=0; i<#_tree_levels; i++){
        #pragma acc kernels{
```

```
            #pragma acc loop gang \
                              independent \
            for(i=0; i<#_queries; i++){
                #pragma acc loop vector \
                                  independent \
                                  private(...)
                for(j=0; j<#_leaves; j++){
                    Compare value against the key
                }
                #pragma acc loop vector \
                                  independent \
                                  private(...)
                for(i=0; i<1; i++){
                    Set offset for the next level
                }
            ...
    }
    #pragma acc kernels{
        #pragma acc gang \
                          independent \
            for(i=0; i<#_queries; i++){
                Select value out of all tree levels
            }
        ...
}
```

Figure 19. Portable B+Tree code translated to OpenACC. Statements in blue represent structural changes required to achieve best-case OpenACC performance.

Table 3. Performance of B+Tree application.

	Arch.	Code Feature	Framework	Kernel Length [lines]	Exec. Time [s]	Speed-up [x]
1	1-core CPU	Original	C	34	35.99e-2	0.92
2	1-core CPU	Structured	OMP	39	33.11e-2	1.00
3	8-core CPU	Structured	OMP	**42**	8.06e-2	**4.11**
4	GPU	Structured	OACC	**72**	29.56e-2	1.12
5	GPU	Structured	CUDA	81	1.57e-2	21.08
6				**114**		**4.11**
7	1-core CPU	Structured Portable	Trellis (-> OMP)	73	33.11e-2	1.00
8	8-core CPU	Structured Portable	Trellis (- > OMP)	73	8.06e-2	4.11
9	GPU	Structured Portable	Trellis (- > OACC)	73	29.56e-2	1.12
10	GPU	Structured Portable	Trellis (-> CUDA)	**73**	1.57e-2	**21.08**
11				**73**		**21.08**
12	GPU	Init/Trans Overhead	CUDA	---	0.35e-2	---
13	GPU	Modified Best-case	OACC	82	2.03e-2	16.31

7. DISCUSSION

7.1 Performance

The results summarized on line 6 in Table 1,

Table 2, and Table 3 illustrate the total code length and the best performance achievable with both OpenMP and OpenACC, the current high-level state-of-the-art solutions that implement CPU and GPU execution. When manually applied to the original code base, 8-core OpenMP yields the best performance that scales with the number of cores (rows 2, 3 in Table 1,

Table 2 and Table 3). The performance of OpenACC is suboptimal due to inefficient mapping onto a GPU (rows 4 in Table 1,

Table 2 and Table 3). However, neither of the two provides performance comparable to that of CUDA, a native GPU solution (rows 5 in Table 1,

Table 2 and Table 3).

The corresponding results for the target codes generated by Trellis are given on line 11. For all applications, target OpenMP code significantly improves the original performance (rows 7, 8 in Table 1,

Table 2 and Table 3, Figure 20 b, c). While the target OpenACC code (row 9 in Table 1,

Table 2 and Table 3, Figure 20 d) yields performance worse than that of the original code (row 1 in Table 1,

Table 2 and Table 3, Figure 20 a), manual modifications to the code structure allow achieving best-case performance (row 13, Table 1,

Table 2 and Table 3, Figure 20 e). Target CUDA code that reflects Trellis' ability to efficiently handle parallel constructs results in the best performance for all applications (row 10 in Table 1,

Table 2 and Table 3, Figure 20 g), equivalent to that of a hand-written CUDA (row 5 in Table 1,

Table 2 and Table 3, Figure 20 f). Within its parallelization limits, OpenACC compiler can efficiently utilize shared memory, as seen with Heart Wall (rows 10, 13 in

Table 2). Relative to CPU, GPU implementations incur an additional overhead due to communication with a driver and data transfer (row 12, Table 1,

Table 2, Table 3).

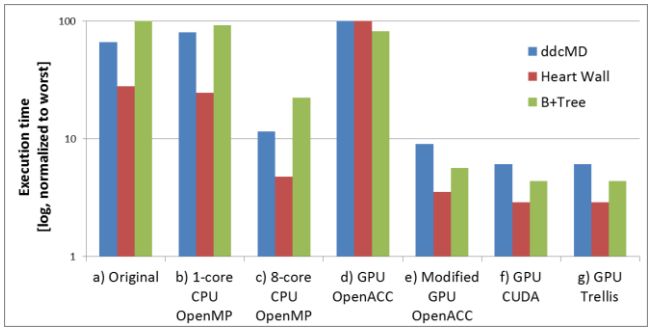


Figure 20. Execution time [log scale, normalized to worst]. Trellis allows achieving GPU performance equivalent to that of a hand-written CUDA code without modifications to the code base.

7.2 Portability

Our experiences with applications and Trellis lead us to the conclusion that the high-level directive-based framework is well suited to achieving high performance while supporting long-term annotation portability and overall ease of programming. The key is to provide intuitive abstractions that map naturally to the important architectural features (e.g., braided parallelism, local synchronization), yet remain general enough to apply across diverse architectures.

Achieving performance portability across a GPU, which poses additional mapping challenges due to its hierarchical structure, required implementation of the new hardware mapping capability in Trellis. This capability complements OpenACC, in its current state, by achieving performance (row 10, Table 1,

Table 2 and Table 3, Figure 20 g) equivalent to that of a hand-written CUDA code (row 5, Table 1,

Table 2 and Table 3, Figure 20 f) without manual modifications to the code base. In contrast, it is important to observe, that as of today, OpenACC requires manual modifications to obtain acceptable performance (row 13, Table 1,

Table 2 and Table 3, Figure 20 e). Unfortunately, these changes require prior knowledge of the compiler's behavior, are unintuitive to a common programmer and may no longer be optimal for a sequential CPU. However, even with such, OpenACC performance still remains inferior due to the lack of thread synchronization support. Since most accelerators are expected to have hierarchical structure, our conclusions regarding GPUs can be generalized.

7.3 Accuracy of Directives

As expected, the performance of Trellis codes for different target frameworks (rows 7, 8, 9 in Table 1,

Table 2 and Table 3) is identical to that of the corresponding manual implementations (rows 2, 3, 4 in Tables Table 1,

Table 2, Table 3). Since annotations used by Trellis are a superset of the relevant directives in OpenMP and OpenACC, the translation to those is straightforward and always results in the code that is as efficient as the hand-written one.

However, the most important observation is that high-level directives are sufficiently descriptive to allow generation of native GPU code (CUDA in our case, row 10 in Table 1,

Table 2 and Table 3) with deterministic performance that is equivalent to that of a hand-written code. Our codes show that many low-level statements can be replaced by higher-level abstractions for convenience, with no loss of performance, which contradicts the conventional view. Individual statements for copying to and from a GPU can be simplified to an enclosing data region. Explicit low-level thread control is unnecessary in most cases as, from coding and performance point of view, it is equivalent to introducing an additional loop of unit or limited width. All other aspects of execution, such as mapping to a hierarchical resource, execution width and the use of shared memory can be in fact efficiently described via directives and implemented by the compiler (or Trellis tool in our case).

7.4 Explicit Annotation

As already mentioned, our results illustrate sufficient descriptive capability of directives to allow proper parallelization even for a GPU. We show that this can be achieved via mere interpretation of comprehensive annotations with no need to rely on compiler dependence/occupancy analysis. Although such analysis is required in the absence of sufficient directives, it cannot be relied on due to its current immature state of the art.

As a part of this approach, we decide to include thread synchronization in the set of basic annotations in Trellis, which may not be supported by compilers in the near future due to the complexity of required dependence analysis. We believe that it is a fundamental high-level abstraction and should be available to allow the programmer to better express their intentions, and thus aid generation of the efficient native code (Figure 10) and avoid scenarios such as that in OpenACC, where kernels needs to be split to implement synchronization (Figure 11).

7.5 Generalization vs. Optimization

The restructuring that removed custom optimizations and regularized parallel control flow, in the case of ddcMD (row 2 in Table 1) to expose parallelism decreased its sequential performance by 17%. However, in the case of Heart Wall and B+Tree applications, similar techniques improved sequential performance by 14% and 9% mainly due to the removal of significant control flow overheads (row 2,

Table 2, Table 3). Thus, for most codes, structuring required for the efficient application of parallel frameworks (directive-based as well as lower-level) does not necessarily decrease performance with respect to that of that obtained with architecture-specific optimizations, while it gives potential of many-fold speedups when running on a parallel architectures. Therefore, programmers should focus their efforts on writing portable code rather than optimizing legacy sequential code.

7.6 Programmability

Our experiences with Trellis provide yet another illustration of the ease of programmability of the directive-based approach that we advocate. Although efficient utilization of directives still requires plentiful and exposed parallelism, unlike other solutions, this approach is characterized by simplicity and convenience. The most important advantage of the directive-based approach is that it allows the programmer to specify desired parallelization without making significant changes to the code base. This is so, because most directives and their clauses have simple forms and they annotate existing structures or regions in the original code base. Moreover, unlike in native approaches, lower-level operations in the form of runtime calls take place transparently. Consequently, many programmers are likely to avoid low-level languages if directive-based frameworks prove to support sufficient functionality and performance.

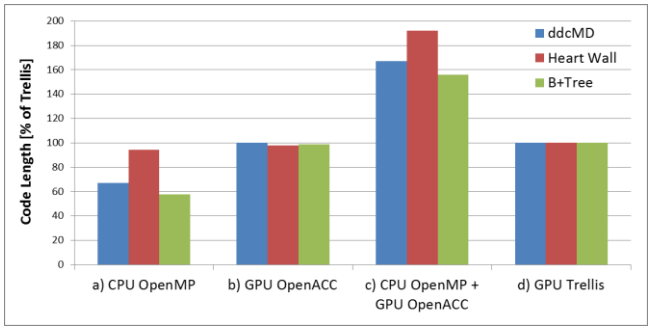


Figure 21. Length of OpenMP CPU code and OpenACC GPU code, normalized to portable Trellis code. Trellis minimizes the total code length required for portable execution.

7.7 Code Length and Maintenance

As of today, our solution decreases the overall code size required for CPU/GPU portability to that of single Trellis code base. The use of a single code base, as demonstrated by our codes written in Trellis, naturally reduces code maintenance costs (Figure 21). The overall code size decreased by 40%, 50%, 36% for ddcMD, Heart Wall and B+Tree (row 11 in Table 1,

Table 2 and Table 3, Figure 21 d) compared to separate multi-core CPU and GPU versions (row 6 in Table 1,

Table 2 and Table 3). Maintaining a single code base simplifies subsequent porting or optimization for any target architecture. While different in structure, both Trellis (row 5 in Table 1,

Table 2 and Table 3) and CUDA (row 10 in Table 1,

Table 2 and Table 3) codes have similar size. While CUDA requires more code to describe lower-level operations, Trellis must specify private variables for each parallelized loop.

8. CONCLUSIONS & FUTURE WORK

Our experiences with the applications presented in the paper, as well as development of the portable Trellis approach, lead us to the following conclusions.

- Most of the necessary building blocks for a common framework with syntactic portability, including high-level frameworks and structured programming, already exist.
- A directive-based approach has sufficient descriptive capability to support execution across multicore CPUs and accelerators.
- Synchronization support and additional hardware mapping capability added to OpenACC can yield performance comparable to that achieved by native solutions.
- Implementation of these concepts in Trellis allows programmers to develop a single code base with competitive performance and programmability as well as decreased code maintenance cost.
- Current solutions such as OpenMP and OpenACC can easily include features demonstrated in Trellis for CPU/GPU portability and improved hardware utilization.

Our future work will extend the Trellis with a more sophisticated transformation analysis and verification against a larger range of codes to demonstrate capabilities of a directive-based approach.

ACKNOWLEDGEMENTS

This work was supported in part by the US NSF under grant MCDA-0903471, SRC under GRC task 1972 and by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

REFERENCES

[1] OpenMP. [Online]. http://www.openmp.com

[2] PGI Accelerator API. [Online]. http://www.pgroup.com/resources/accel.htm

[3] OpenACC. [Online]. http://www.openacc-standard.org

[4] S Lee, S Min, and R Eigenmann, "OpenMP to GPGPU: a Compiler Framework for Automatic Translation and Optimization.," in *Proceedings of PPoPP'09*, New York, 2009.

[5] OpenCL. [Online]. http://www.khronos.org/opencl/

[6] F. H. Streitz et al., "100+ TFlop Solidification Simulations on BlueGene/L.," in *Proceedings SC'05*, Seattle, 2005.

[7] S. Che et al., "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads.," in *Proceedings of the ISWC'10.*, Atlanta, 2010.

[8] J. Fix, A. Wilkes, and K. Skadron, "Accelerating Braided B+ Tree Searches on a GPU with CUDA.," in *Proceedings of the 2nd A4MMC Workshop, in conjunction with ISCA.*, San Jose, 2011.

[9] R. D. Blumofe, C. F. Joerg, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System.," in *Proceedings of PPoPP'95.*, Santa Barbara, 1995.

[10] Threading Building Blocks (TBB). [Online]. http://threadingbuildingblocks.org/documentation.php

[11] CUDA-x86. [Online]. http://www.pgroup.com/resources/cuda-x86.htm

[12] G Diamos, A. Andrew, and M. Kesavan, "Translating GPU Binaries to Tiered SIMD Architectures with Ocelot.," Atlanta, 2009.

[13] Jacket. [Online]. http://www.accelereyes.com/jacket_tour

[14] ArrayFire. [Online]. http://www.accelereyes.com/arrayfire_tour

[15] P. Hanrahan. Domain-Specific Languages for Heterogeneous GPU Computing. [Online]. http://www.graphics.stanford.edu/~hanrahan/talks/

[16] Gromacs. [Online]. http://www.gromacs.org/

[17] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU Graph Traversal.," in *Proceedings of PPoPP'12.*, New Orleans, 2012.

[18] L. G. Szafaryn, T. Gamblin, B. de Supinski, and K. Skadron, "Experiences with Achieving Portability Across Heterogeneous Architectures.," in *Proceedings of WOLFHPC, in conjunction with ICS.*, Tucson, 2011.

[19] CUDA Programming Guide 3.2. [Online]. http://developer.download.nvidia.com

[20] Michael W. Boyer. CUDA Support Page. [Online]. https://www.cs.virginia.edu/~csadmin/wiki/index.php/CUDA_Support

[21] GCC. [Online]. http://gcc.gnu.org/

[22] Many Integrated Core (MIC) Architecture. [Online]. http://www.intel.com

[23] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, "Cell Broadband Engine Architecture and Its First Implementation: a Performance View.," in *IBM Journal of Research and Development.*, 2007.

[24] CSX700. [Online]. http://support.clearspeed.com/documentation/hardware/

[25] D. Tarjan, J. Meng, and K. Skadron, "Increasing Memory Miss Tolerance for SIMD Cores.," in *Proceedings of SC'09.*, Portland, 2009.