Experiences with Achieving Portability across Heterogeneous Architectures

Lukasz G. Szafaryn* Todd Gamblin† Bronis R. de Supinski† Kevin Skadron*

*University of Virginia
{lgs9a, ks7h}@virginia.edu

†Lawrence Livermore National Laboratory
{tgamblin, bronis}@llnl.gov

ABSTRACT

The increasing computational needs of parallel applications inevitably require portability across popular parallel architectures, which are becoming heterogeneous. The lack of a common parallel framework results in divergent code bases, difficulty in porting, higher maintenance cost, and, thus difficulty achieving optimal performance on target architectures.

Our paper examines two representative parallel applications and describes code structuring and annotations required to derive a single codebase that is parallelizable across representative heterogeneous architectures, such as multi-core CPU and GPU. Drawing on previous work in the area, we create a universal high-level directive-based framework that supports both of these architectures, and implements execution on each via translation to OpenMP and PGI Accelerator API, respectively.

We demonstrate that a high-level framework can support a common codebase that efficiently executes on heterogeneous architectures. Our results show that when combined with a state-of-the-art parallelizing compiler, such framework can yield performance comparable to custom code or a native language. Further, we show that the approach increases programmability, reduces code size and decreases maintenance cost.

General Terms

Measurement, Performance, Languages.

Keywords

Physical Systems, Simulation, Parallel Processing.

1. INTRODUCTION

1.1 Hardware and Application Development

Compute resources at major supercomputing centers have recently exhibited a steady trend of increasing heterogeneity. Today, many standalone machines and some clusters have both multi-core CPUs and GPUs. However, most legacy codes have not evolved to exploit these architectures fully. In this paper, we look at two scientific applications: ddcMD (molecular dynamics) [1] and Heart Wall (image processing) [2] and describe developments required to make them portable across the two architectures.

We have observed a common development pattern in these applications. Typically, large parallel applications are only coarsely parallelized with MPI for execution on homogeneous clusters, with serial execution at each node. When this programming model is extended to multiple cores at each node, we can only further optimize the serial code at the level of each node. However, the increasing computational needs of parallel applications can eventually lead to opportunities to improve efficiency by parallelizing the application for execution on multi-core CPUs or accelerators such as GPUs. These parallel architectures, especially GPUs, can significantly increase node computation capabilities while in many cases decreasing the requirement on the size of the cluster. Efficient utilization of these architectures requires proper parallelization of the serial code.

Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL- CONF-486291), and was supported in part by NSF grants CNS-0916908, MCDA-0903471, and the SRC under task 1972.

1.2 Code Structure and Optimizations

We encounter many programming difficulties in transitioning between architectures. For example, consider a serial code that consists of several tasks, some represented by loops, that are arranged intuitively according to sequential steps in the algorithm rather than dependence relationships between them. While this hardly makes a difference in case of serial execution, subsequent parallelization of such code is difficult because it does not expose available width of parallelism and potential for data sharing in order to utilize multi-core CPUs or GPUs fully. For the same reason, merely increasing the number of tasks offloaded to either of the two architectures is insufficient. The problem is exaggerated for GPUs, which require the developer to specify two hierarchical levels of parallelism. We could also imagine code that was only coarsely parallelized to match a small number of cores in the CPU. When coding for a GPU, we would have to split each task in order to exploit fine-grained parallelism and to decrease the register footprint for higher concurrency. In many cases, code structuring, that is required for parallelization, alone reduces control flow overhead and yields improved performance.

Many codes tailor optimizations to specific architectures. One such optimization bypasses the compiler to ensure the use of particular instructions by specifying them in the code. Some codes optimize serial execution by inlining functions and unrolling loops, both of which increase the register footprint and decrease the width of available parallelism. Other optimizations include memory access changes such as alignment via padding or orchestrating computation to match cache line sizes. While these custom optimizations may improve performance on a given target architecture, they often decrease it for other available architectures. In many cases, transitioning to a more efficient parallel architecture could provide performance that surpasses that of code customized for a particular-architecture.

1.3 Need for Portable Code

We conclude that the lack of a common framework to enable efficient utilization of parallel resources leads to code with poor overall structure and often necessitates reliance on custom

optimizations to the original serial code. However, code with custom structure, level of parallelization and optimization tailored to a specific architecture is often not portable. The process of porting, if attempted, often results in incorrect architectural and algorithmic tradeoffs, leading to suboptimal performance. Even if we manage to port the code across architectures, with all of the difficulties involved, we are left with multiple semi-optimized and divergent versions, each expensive to maintain when changes are made to the underlying algorithm. Therefore, we need mechanisms to support a single code that exhibits performance portability. The lack of such a programming model inhibits progress by scientists because they cannot easily achieve optimal performance on the available computational resources.

1.4 Optimal Framework

The capability of a compiler to parallelize sequential code automatically has not been sufficient in the past. Even current state-of-the-art parallelizing compilers are effective mainly at ILP (when compiling into VLIW or vector instructions). This limitation almost always forces manual parallelization to achieve good performance at DLP and TLP. For clarity and convenience, many programmers prefer to parallelize code by annotating a traditional, serial programming language such as Fortran or C rather than using new languages or extensions that require changes to the existing code. However, a lack of a common and efficient framework for heterogeneous architectures necessitates using new languages that are often hardware-specific in order to achieve better performance. Our goal is to demonstrate that the benefits of both can be combined in a single portable, convenient framework. We advocate the use of a directive-based framework and show that it can support efficient parallelization.

1.5 Our Approach and Contributions

We first demonstrate the concept of code portability across multiple architectures in terms of syntax, for which we create our own portable framework. Drawing on previous work in this area we follow the approach of OpenMP [3] and the PGI Accelerator API [4], popular directive-based frameworks for multi-core CPUs and NVIDIA GPUs [5]. Our framework combines functionality of the two frameworks in the form of a single set of directives that are portable across the two architectures. In order to implement execution of our framework on these architectures, we build a source-to-source translator that transforms our directives to those in OpenMP or the PGI Accelerator API.

Next, we illustrate the concept of portability in terms of performance. In order to achieve that, we structure the parallel code properly to facilitate use of our framework and efficient mapping of workload onto target architectures. We group similar independent tasks to maximize the width of parallelism. In order to make the code suitable for GPUs, we split long tasks to exploit fine-grained parallelism and to improve concurrency by limiting the register footprint. In order to make the code readable and amenable to the application of our directives, we ensure that the computation of the code, usually in nested loops, is kept generic. The remaining parts of the code, that include setup and allocation, as well as custom statements associated with particular APIs are all implemented in a separate code section. As much as possible, we avoid breaking loops, unrolling and conditional statements, at the cost of minimal redundant computation.

We validate our approach through performance evaluation of the two applications on multi-core CPUs and GPUs. We are only concerned with tasks processed at each node. Therefore we assume that the amount of work at each node is appropriately balanced by the higher-level code. We outline specific considerations for each application and architecture, and we discuss the benefits of our approach in terms of portability, programmability, performance and code maintenance. Also, based on our experiences with parallel applications, we propose extensions to our framework, for improved hardware utilization, which we plan to implement in the future. Our paper makes the following contributions.

- Demonstration of the feasibility of maintaining a generic version of code portable across multiple architectures via the use of a single high-level framework.
- Description of the modifications to our two applications that facilitate correct use of high-level directives.
- Presentation of a convenient framework that combines and extends features of other frameworks into a coherent set of directives that are portable across architectures.
- Demonstration of a source-to-source translation of our framework into OpenMP and the PGI Accelerator API for execution on multi-core CPUs and GPUs.
- Illustration of efficient cross-architecture parallelization with high-level directives aided by existing parallelizing compilers.
- Analysis of the tradeoffs between various aspects of the approach that include portability, programmability, performance and code maintenance.

1.6 Organization of the Paper

The remainder of this paper is organized as follows. Section 2 gives background information on parallel application structure and extraction of parallelism in different architectures. Section 3 describes related work. Section 4 details our annotations. Section 5 provides an overview of our experimental setup and methodology. Sections 6 and 7 present the two applications on which our study focuses including their portable implementation and the corresponding performance. Section 8 discusses the benefits of our programming approach and considerations for particular applications and architectures. Section 9 summarizes our work. Section 10 outlines possible future research. This paper is best viewed in color.

2. BACKGROUND

Parallel applications are usually structured as a system of nested loops, each corresponding to different levels of parallelism (TLP or DLP). While Amdahl's law provides a theoretical performance limit based on the code's parallelism, the amount of computation at each level of parallelism and its mapping to the architecture determine the actual speedup. A typical heterogeneous cluster node is equipped with a multi-core CPU and an accelerator. While multi-core CPUs only allow access to individual cores, accelerators such as GPUs, Cell BE [6] and ClearSpeed [7] expose a multi-level hierarchy of processing elements to the programmer. These elements are arranged into groups that naturally correspond to levels of parallelism in applications. Each group typically collectively executes a task, while elements inside each group exploit data parallelism within the task. In accelerators such as GPUs, the second group of elements executes in lock-step and incurs some penalty in terms of computation and memory latency when threads diverge on a conditional statement.

3. RELATED WORK

While experiences with application development and transformation described in this paper are original and specific to our work, we build on significant previous research in parallel frameworks. Inadequate functionality of existing parallel frameworks for recent architectures motivates our framework.

In the case of multi-core CPU execution, early languages such as Cilk [8] use library functions for automatic parallelization of typical tasks such as looping and reduction. Subsequent developments such as TBB [9] also abstract the aspects of thread management away from the programmer. Unlike our approach, these solutions make changes to the existing code, use parallelizing algorithms tailored for shared-memory and lack features required for describing multi-level parallelism. OpenMP, on the other hand, facilitates execution on multi-core CPUs via the use of high-level directives that annotate parallel constructs, which is similar to our approach. However, it provides only a small set of directives that limits its support only to single-level parallelism in a shared-memory system.

CUDA [5] and OpenCL [10] enable parallel execution on GPUs (and other devices, including CPUs, in the case of OpenCL). Both languages facilitate efficient parallelization but require the use of explicit low-level statements, which increase the learning effort and the amount of repetitive coding involved. Cetus [11] and proposed future OpenMP extensions [12], on the other hand, use high-level directives to implement execution on multiple architectures including CPUs and GPUs. The PGI Accelerator API uses similar concepts with significantly better compiler support, but its support is limited to GPUs. These frameworks rely on the parallelizing capability of the compiler when translating to the native language. While these frameworks are similar in form and functionality to our approach, they do not provide a single set of directives that can be used on both multi-core CPUs and GPUs.

We are aware of flexible application-specific programming interfaces [13] as well as those for molecular dynamics applications [14] that support heterogeneous platforms. However, these rely on the use of libraries that implement specific functionality and, unlike our approach, they do not provide a general framework for a wide range of applications.

4. OUR FRAMEWORK

4.1 Form and Functionality

The form of our framework closely follows that of the PGI Accelerator API, but its functionality is extended with respect to that in the PGI Accelerator API in order to support multi-core CPUs. The following section gives a general overview of the current features in our framework, which are largely derived from those in the PGI Accelerator API. Currently, the functionality of our framework is limited to that of OpenMP and the PGI Accelerator API, the underlying frameworks to which it is translated. This limitation is currently acceptable since we are still able to illustrate the concept of portable code. However, based on experiences with applications presented in the paper, we propose several extensions to our framework that go beyond the functionality of the two underlying frameworks. Their use is illustrated in Figure 2 and they are discussed in Section 10.

4.2 Structure

Our framework provides two types of directives: annotative and declarative. Annotative directives (Figure 1) describe parallel control flow and data transfers associated with a particular code construct. These directives include loop-mapping directives and data-mapping directives. Multiple annotative directives with corresponding clauses can be included on the same line. Alternatively, declarative directives support device setup or explicit data transfers that are not associated with a particular construct, but an implicit code region in which they appear.

```
#pragma api directive-name (clause)
```

Figure 1. Form of an annotative directive in our framework.

Loop-mapping directives specify the type of execution for an annotated code segment, such as parallel, vector or sequential. In the case of a multi-core CPU, a loop annotated with the parallel directive would execute concurrently in different cores, and any enclosed loops would execute in series. In case of a GPU, a loop annotated with the *parallel* directive would execute in multiprocessors, while any enclosed loops annotated with the *vector* directive would execute in individual processing units inside a multiprocessor. Data mapping directives such as *shared, private, cache, local, copy, copyin* or *copyout*, on the other hand, specify the types of data used in the segments,. The first two can be used with a reference to variables for both multi-core CPUs and GPUs. The last five are used for caching and allocating in GPU memory as well as transferring data between GPU and system memory. Clauses correspond to the number of parallel tasks, vector widths or the names of the actual variables, depending on the type of a corresponding directive.

```
#pragma api data copyin(input) copyout(output)
{
    Serial Code
    #pragma api compute{
        #pragma api id(0) device(1) proc(1:10) parallel(20)
                    copyin(input2) copyout(output2)
        for(i=0; i<x; i++){
            #pragma api vector(12) shared(variable A)
            for(j=0; j<x; j++){
                Parallel Code 1
                    …
            }
        }
        #pragma api parallel(60) device(1) proc(11:30)
                concurrent(0) copyin(input3) copyout(output3)
        for(i=0; i<x; i++){
            #pragma api vector(14) shared(variable B)
            for(j=0; j<x; j++){
                Parallel Code 2
                    …
            }
    …
}
```

Figure 2. Example of a structured code written with our framework (with proposed extensions discussed in Section 10).

Our framework assumes the use of both, generic directives that are applicable to all architectures, and specific directives that are supported only by some, architectures (e.g., GPUs). As of now, the framework requires specification of the target device in the code, so that it can generate code with appropriate directives. We envision that the future runtime with native support for our framework would be aware of the devices present in the system and determine applicability of directives automatically. We

provide an example of a code that uses both annotative and declarative directives in Figure 2.

As seen in the code example, *#pragma our_api data* specifies a region at the beginning of which data is copied into a GPU and at the end of which it is possibly copied out. Backed by the functionality of the PGI parallelizing compiler, our framework should only require a coarse grained annotation of the code, via *#pragma our_api compute*, that specifies the region to be parallelized for the target architecture. The fine-grained loop mapping directives described earlier should only help the compiler efficiently map the application onto the features of the underlying hardware. Since we expect that the compiler can also attempt to parallelize loops with no annotations, we annotate them with *sequential* directive to avoid parallelization.

5. SETUP AND METHODOLOGY

We configured 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.

We first structure the existing C code for these applications and then extend them with our framework directives. Our source-to-source translator, written in Perl, converts such codes to OpenMP and the PGI Accelerator API. We then compile the codes with PGCC [4] for multi-core CPUs and NVIDIA GPUs. We also developed GPU codes, written in CUDA, for each application and compiled with NVCC [5] for performance comparisons. We use the same CUDA codes for estimating GPU-specific overheads.

We present performance results that compare architecture-specific codes and our portable codes executing on three architectures (single-core CPU, multi-core CPU and GPU) in terms of kernel code lengths and performance. We obtain these results on a single machine, equivalent to a cluster node, equipped with an 8-core Intel Xeon X5550 2.67GHz CPU and NVIDIA GeForce GTX460 GPU.

Our results do not account for the overhead of source-to-source translation, which would disappear if the directives were supported natively by the compiler. We refer to NVIDIA terminology [5] when describing GPU optimizations. In order to make a fair comparison between line counts, we include one directive per line in our codes. Since translation of directives is mostly straightforward, as seen in Figures 5-7 and 10-12, we do not discuss it in the paper.

6. ddcMD APPLICATION

6.1 Functionality and Algorithm

The ddcMD application 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 3). The large box at each node is further divided into cubes, called *boxes*. 26 *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 ones at larger distances exert negligible forces. Thus the box size s chosen so that 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. Since particle interactions are mutual, the result can update 2 particles while reducing the required work.

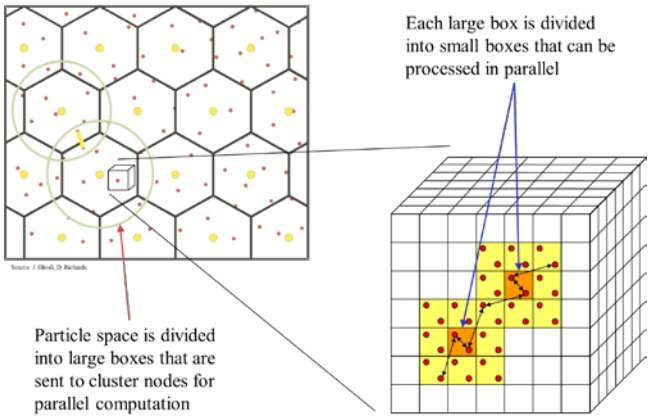

Figure 3. Partitioning of computation in ddcMD application. For every particle in orange area, interactions with all particles in the surrounding yellow area are calculated.

Figure 4 shows the ddcMD code structure that executes on a node. The code has 2 groups of nested loops enclosed in the outermost loop, which processes home boxes. For any particle in the home box, the 1st and 2nd nested loops process interactions with other particles in the home box and particles in all neighbor boxes. The processing of each particle consists of a single stage of calculation that is enclosed in the innermost loop.

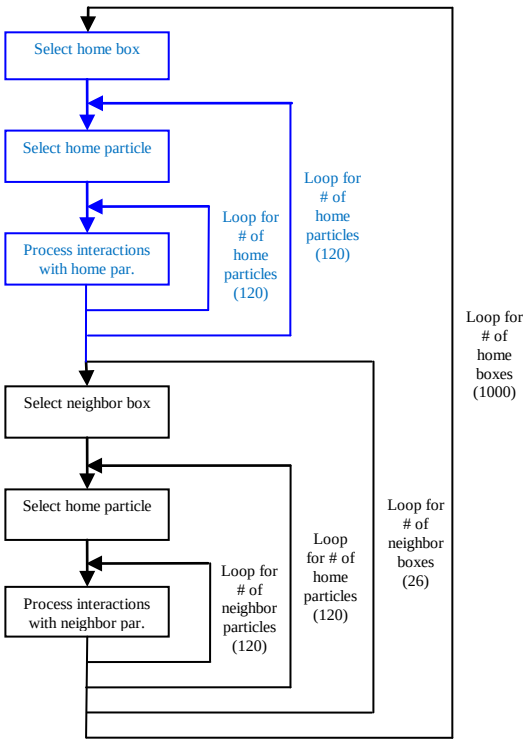

Figure 4. Original ddcMD loop structure. In portable code, section in blue is merged with the remaining structure.

6.2 Structure of Portable Code and Results

We made several changes to support performance portability. In order to simplify the loop structure for convenient application of high-level directives, we collapsed the 3D loop that processes boxes according to dimensions in the large box to a single dimension. We combined loops that calculate interactions with

particles in home and neighbor boxes to increase the width of parallelism. The lack of a fine-grained synchronization mechanism required that the parallel version of the code only updates one particle instead of two, which increased the amount of work. In order to facilitate efficient GPU execution, convergence of threads and data access were improved by avoiding conditionals on the cutoff radius, which resulted in a small amount of additional work. Also, we converted nested pointer structures to relative indices to make them valid when transferring to GPU memory. All ISA and memory-specific optimizations were removed at a small performance cost.

```
Conversion of indices to partitioned space
#pragma api data copyin(box[0:#_boxes-1]) \
        copyin(pos[0:#_par.-1]) copyin(chr[0:#_par.-1]) \
        copyout(dis[0:#_par.-1]){
    #pragma api compute{
        #pragma api parallel(30) private(…) \
                    cache(home_box)
        for(i=0; i<#_home_boxes; i++){
            Home box setup
            #pragma api sequential
            for(j=0; j<#_neighbor_boxes; j++){
                Neighbor box setup
                pragma api parallel vector (128) private(…) \
                            cache(neighbor_box)
                for(k=0; k<#_home_particles; k++){
                    pragma api sequential
                    for(l=0; l<#_neighbor_particles;l++){
                        Calculation of interactions
                    }
            …
}
```

Figure 5. Portable ddcMD code.

```
Conversion of indices to partitioned space
omp_set_num_threads(30);
#pragma omp parallel for private(…)
for(i=0; i<#_home_boxes; i++){
    Home box setup
    for(j=0; j<#_neighbor_boxes; j++){
        Neighbor box setup
        for(k=0; k<#_home_particles; k++){
            for(l=0; l<#_neighbor_particles;l++){
                Calculation of interactions
            }
      …
}
```

Figure 6. Portable ddcMD code translated to OpenMP. Statements in blue represent changes due to translation.

Figure 5 illustrates general structure of the resulting portable code written in our framework, while Figure 6 and Figure 7 show translations of this code to OpenMP and PGI Accelerator API, respectively. We rearranged the processing of home-neighbor particle interactions in the two innermost loops to increase locality and order of reference, which in turn improved cache utilization in multicore CPUs and stride memory access in GPUs. While, the processing of boxes was parallelized across cores in CPU or multiprocessors in GPU, the processing of enclosed particles was vectorized, but only in case of a GPU. We specified the upper bound of 30 processing units for the parallel task and 128-wide vector for the vector task in order to make a match between task sizes and available GPU resources. The code involves transferring of data between GPU and system memories, declaration of private variables and no explicit allocation on the device. Table 1 compares performance and code lengths, which we discuss in Section 8.

```
Conversion of indices to partitioned space
#pragma acc data region copyin(box[0:#_boxes-1]) \
        copyin(pos[0:#_par.-1]) copyin(chr[0:#_par.-1]) \
        copyout(dis[0:#_par.-1]){
    #pragma acc region{
        #pragma acc parallel(30) independent private(…) \
                    cache(home_box)
        for(i=0; i<#_home_boxes; i++){
            Home box setup
            #pragma acc sequential
            for(j=0; j<#_neighbor_boxes; j++){
                Neighbor box setup
                pragma acc parallel vector (128) private(…) \
                            cache(neighbor_box)
                for(k=0; k<#_home_particles; k++){
                    pragma acc sequential
                    for(l=0; l<#_neighbor_particles;l++){
                        Calculation of interactions
                    }
            …
}
```

Figure 7. Portable ddcMD code translated to PGI Accelerator API. Statements in blue represent changes due to translation.

Table 1.Performance of ddcMD Application.

	Arch.	Code Feature	Framework	Kernel Length [lines]	Exec. Time [s]	Speedup [x]
1	1-core CPU	Original	C	47	56.19	1
2	1-core CPU	Structured	C	34	73.73	0.76
3	8-core CPU	Structured	OpenMP	**37**	10.73	5.23
4	GPU	GPU-specific	CUDA	**59**	5.96	**9.43**
5				**96**		**9.43**
6	1-core CPU	Structured Portable	Our Framework	47	73.72	0.76
7	8-core CPU	Structured Portable	Our Framework	47	10.73	5.23
8	GPU	Structured Portable	Our Framework	**47**	7.11	**7.91**
9				**47**		**7.91**
10	GPU	Init/Trans Overhead	(CUDA)	---	0.87	---

7. HEART WALL APPLICATION

7.1 Functionality and Algorithm

The Heart Wall application tracks the movement of a mouse heart over a sequence of 609x590 ultrasound frames (images) to observe response to a stimulus. For a long sequence of frames, images are arranged into batches and offloaded to individual nodes for parallel processing (Figure 8). In its initial stage, not included in our code, 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. Due to dependency on the feature detection, the processing of subsequent frames in a batch must proceed sequentially. The application tracks the movement of heart walls by detecting displacement of image areas under sample points as shapes of heart walls change throughout the remaining frames. Green and blue dots in Figure 8 indicate sample points that mark inner and outer heart walls, respectively.

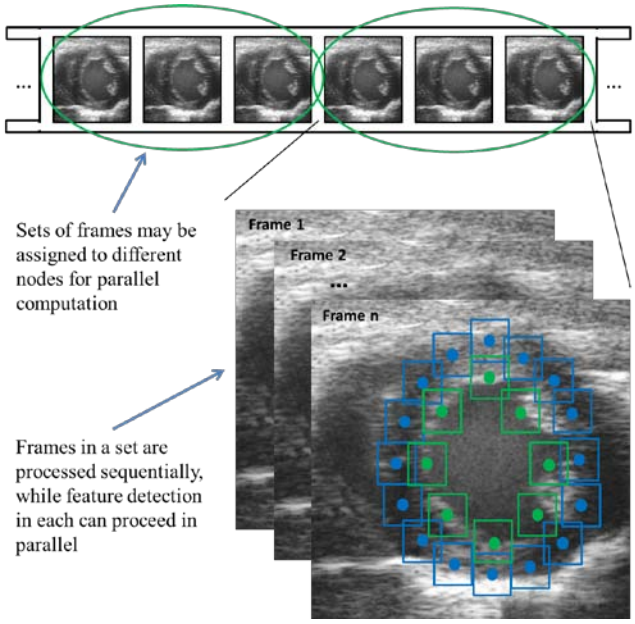

Figure 8. Partitioning of computation in Heart Wall application. Movement of areas marked with blue and green squares is tracked throughout a sequence of frames.

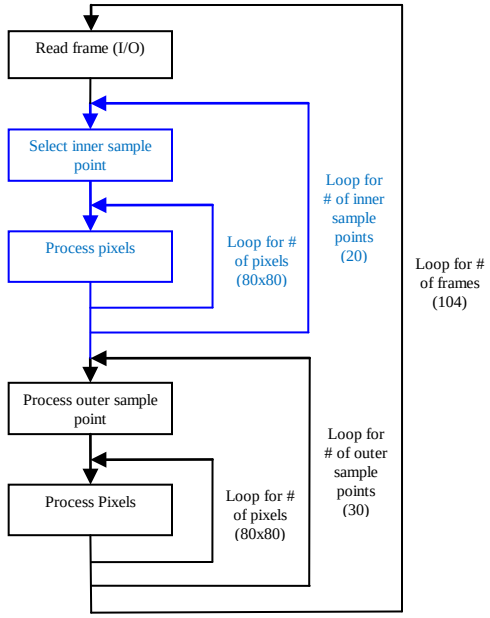

Figure 9. Original Heart Wall loop structure. In portable code, section in blue is merged with the remaining structure.

Figure 9 shows the Heart Wall code structure that executes on a node. The code has 2 groups of nested loops enclosed in the outermost loop. The outermost loop processes frames from the partitioned sequence of frames. The 1st and 2nd groups of loops track features around sample points on inner and outer heart walls, respectively. The processing of each sample point consists of several sequential tracking stages included in the innermost loops that are interleaved by control statements.

7.2 Structure of Portable Code and Results

We made several changes to support performance portability. Even though the processing of inner and outer points is almost identical, the original code separated the two and used two corresponding sets of variables and arrays. We consolidated this processing in order to simplify the structure of the loops for convenient application of high-level directives and to decrease the number of data-mapping directives. The lack of a fine-grained synchronization mechanism between sequential stages in feature tracking required full synchronization after each stage. In order to facilitate efficient GPU execution, we combined a few interleaved sequential code regions with the parallel ones and executed them with the full vector width specified.

```
Processing of inputs from earlier stages.
for(i=0; i<#_frames; i++){
    Read frame
    #pragma api data copyin(frm[0:frm_siz-1]) \
            copyin(ini_loc[0:#_smp_pnts.-1]) \
            local(con/cor[0:#_pixels]) copyout[fin_loc[0:#_.-1]){
        #pragma api compute{
            #pragma api parallel(30)
            for(j=0; j<#_sample_points; j++){
                #pragma api vector(512) private(…)
                for(i=0; i<#_pixels; i++){
                    Convolving/correlating with templates
                }
                #pragma api vector(512) private(…)
                for(i=0; i<#_pixels; i++){
                    Determining displacement
                }
            …
}
```

Figure 10. Portable Heart Wall code.

```
Processing of inputs from earlier stages.
for(i=0; i<#_frames; i++){
    Read frame
    omp_set_num_threads(30);
    #pragma omp parallel for private(…)
    for(j=0; j<#_sample_points; j++){
        for(i=0; i<#_pixels; i++){
            Convolving/correlating with templates
        }
        for(i=0; i<#_pixels; i++){
            Determining displacement
        }
        …
}
```

Figure 11. Portable Heart Wall code translated to OpenMP. Statements in blue represent changes due to translation.

Figure 10 illustrates general structure of the resulting portable code written in our framework, while Figure 11 and Figure 12 show translations of this code to OpenMP and PGI Accelerator API, respectively. Similarly to ddcMD, we rearranged processing of sequential tracking stages in the innermost loops to increase

locality and order of reference. We parallelized processing of sample points and vectorized processing of each detection stage. We set the upper bound of processing units to 30 and vector width to 512. In addition to transferring data between the GPU and system memories as well as declaring private variables, the code explicitly allocates temporary variables in GPU memory. Table 2 compares performance and code lengths, which we discuss in Section 8.

```
Processing of inputs from earlier stages.
for(i=0; i<#_frames; i++){
    Read frame
    #pragma acc data region copyin(frm[0:frm_siz-1]) \
            copyin(ini_loc[0:#_smp_pnts.-1]) \
            local(con/cor[0:#_pixels]) copyout[fin_loc[0:#_.-1]){
        #pragma acc region{
            #pragma acc parallel(30) independent
            for(j=0; j<#_sample_points; j++){
                #pragma acc vector(512) independent
                            private(…)
                for(i=0; i<#_pixels; i++){
                    Convolving/correlating with templates
                }
                #pragma acc vector(512) independent
                            private(…)
                for(i=0; i<#_pixels; i++){
                    Determining displacement
                }
            …
}
```

Figure 12. Portable Heart Wall code translated to PGI Accelerator API. Statements in blue represent changes due to translation.

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	132	117.21	1
2	1-core CPU	Structured	C	124	112.44	1.04
3	8-core CPU	Structured	OpenMP	**138**	15.84	7.40
4	GPU	GPU-specific	CUDA	**156**	6.54	**17.92**
5				**294**		**17.92**
6	1-core CPU	Structured Portable	Our Framework	144	112.44	1.14
7	8-core CPU	Structured Portable	Our Framework	144	15.84	8.09
8	GPU	Structured Portable	Our Framework	**144**	7.21	**16.25**
9				**144**		**16.25**
10	GPU	Init/Trans Overhead	(CUDA)	---	1.13	---

8. DISCUSSION

8.1 Performance

Improving programmability primarily motivates our work so we focus on our programming experiences. However, we evaluate the performance of our portable code to demonstrate that it is close to that of lower level implementations that require more coding effort.

The structuring of the original ddcMD code required for proper application of our directives decreased its performance by 31% due to removing of custom optimizations and regularization of the parallel control flow (row 1, Table 1). However, in case of Heart Wall, similar techniques improved the performance of the original code by 4% due to organizing the more regular parallelism better and removing some of the control flow overheads (row 1, Table 2). While CUDA uses more notations to describe parallelism, our framework requires additional notations for specifying private variables. While different in structure, due to that reason, these codes can have similar size (row 4, 8, Table 1, 2).

Our results for both applications illustrate that performance of our translated portable codes scales with the number of CPU cores (row 6, 7, Table 1, 2) and significantly increases with the use of a GPU (row 8, Table 1, 2). The GPU version requires longer compilation due to underlying translation from PGI Accelerator API to CUDA. It also incurs more delay at run time due to communication with the GPU driver. Nevertheless, our results show that these overheads are small (row 10, Table 1, 2) relative to the performance gain that the GPU provides. Due to the same overheads as well as efficiency of the PGI parallelizing compiler, there is an expected slight difference in performance between out GPU implementation and that of CUDA (row 4, 8, Table 1, 2). Although, we use NVIDIA GPUs for our work, many of the lessons learned apply to other accelerators that share the concept of hierarchical structure.

8.2 Feasibility

We conclude that generic code that uses high-level directives can exploit parallel node architectures to provide performance beyond that of any specifically optimized serial code, even when accounting for overheads due to regularization of control flow in parallel code (row 5, 9, col 6, Table 1, 2). Therefore, programmers should focus their efforts on writing portable code rather than optimizing legacy serial code.

However, the most important conclusion that we derive from our work is that we can achieve efficient execution on parallel architectures using high-level directives. While portable codes may incur some penalty due to the general way of describing parallel tasks, our results show that the performance difference is very small. This proves that high-level directives provide sufficient descriptive capability to achieve optimal performance. Therefore we can avoid low-level languages with specific calls to the runtime. The performance of the portable code depends on how well programmer structures the code and applies the directives. While good performance on a given architecture may still require the use of many specific annotations, we demonstrate how one coherent framework can support parallel computation on multiple architectures while keeping the code simple and generic.

8.3 Programming and Maintenance

Using a high-level framework improves programmability because it extends the language to target common code constructs, such as loops, to a range of devices. It also does not require a specific code structure. Portability across a range of architectures as well as the efficiency in describing parallelism for them even further illustrates the programmability of the approach. Due to these

factors, a high-level framework approach is likely to gain wide acceptance.

Having a single code base reduces code maintenance costs, which are significant for large codes bases that tend to change frequently. The overall size of each of the applications presented in the paper decreased by over 50% compared to when we maintain separate single/multi-core CPU and GPU versions (row 5, 9, col 4, Table 1, 2). Even if custom optimizations become required, maintaining a single code base makes subsequent adjustments or porting to any target architecture more feasible.

8.4 Implementation Problems

When developing our portable codes and source-to-source translator, we encountered several problems that are all related to the current state of the art of OpenMP and the PGI Accelerator API. OpenMP does not support array privatization, which requires the programmer to allocate private copies of the array. PGI does not support structures and function calls, which are common in parallel applications. While our source-to-source translator tries to compensate for these restrictions, they still limit our solutions. The lack of support for convenient directives such as loop collapsing in PGI compiler kept us from implementing these in our framework. Also, GPU translations of our code require the use of *independent* directives to help PGI compiler properly parallelize loops.

9. CONCLUSIONS

Our experiences with writing portable code for the applications presented in the paper as well as with developing our framework lead us to the following conclusions.

- A common high-level annotation framework can support efficient execution across architecture types.
- Correct use of this framework with the support of the current state-of-the-art parallelizing compilers can yield comparable performance to a custom, low-level code.
- Our approach results in increased programmability across architectures and decreased code maintenance cost.

10. FUTURE WORK

Based on our experiences with applications presented in the paper, we propose 3 extensions to our framework that are described below. For future work, we plan to extend our source-to-source translator to include code analysis and generation required to implement these. If implemented, the last 2 extensions would allow concurrent execution of two sequential sets of nested loops present in both ddcMD and Heart Wall. In order to show projected performance gain due to these features in our results, we made equivalent manual changes to these codes by merging the loops, as described in Sections 6.2 and 7.2.

One feature that we propose is assignment of tasks to particular devices. We envision that our future runtime will provide *get_devices()* statement for obtaining a list of available devices. The programmer would then annotate each outermost loop with the ID of the device on which to execute that segment of code via the *device(#)* directive, as seen in Figure 2. The code could be annotated with multiple IDs in order to request collaborative execution as presented in previous research [15].

Another feature that we propose is the explicit specification of concurrent code execution. In order to facilitate this feature, we expect that programmer labels code sections with particular IDs via the *id(#)* directive. Concurrent execution of two sections of code could be requested by specifying the ID of the second section via the *concurrent(#)* directive when annotating the first section.

In order for code to execute concurrently on the same accelerator device, as specified for each section of code by *device(#)* directives, the GPU kernel must be limited to a specific range of processing units through the *set_number_processors()* directive, which is another proposed feature of our framework. This directive can also confine a kernel to a general number of processing units, through a range defined by the compiler.

11. REFERENCES

[1] F. H. Streitz, J. N. Glosli, M. V. Patel, B. Chan, R. K. Yates, B. R. de Supinski, J. Sexton, J and A. Gunnels. 100+ TFlop Solidification Simulations on BlueGene/L. In *Proceedings of SC 05*. Seattle, WA. 2005

[2] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of* IISWC '09. IEEE Computer Society, Washington, DC, USA, 44-54.

[3] OpenMP. < http://openmp.org/wp/>

[4] PGI Accelerator API. <http://www.pgroup.com/resources/accel.htm>

[5] NVIDIA CUDA Programming Guide 3.2. <http://developer.download.nvidia.com >

[6] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. 2007. Cell broadband engine architecture and its first implementation: a performance view. *IBM J. Res. Dev.* 51, 5 (September 2007), 559-572.

[7] ClearSpeed CSX700. <http://support.clearspeed.com/documentation/hardware/ >

[8] Robert D. Blumofe, Christopher F. Joerg, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. "Cilk: An Efficient Multithreaded Runtime System." PPoPP '95. July 19-21, 1995, Santa Barbara, California, pp. 207-216.

[9] Intel Threading Building Blocks. <http://threadingbuildingblocks.org/documentation.php>

[10] Khronos Group. OpenCL. < http://www.khronos.org/opencl/>

[11] S. Lee, S. Min, and R. Eigenmann. 2009. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *Proceedings of* PPoPP '09. ACM, New York, NY, USA, 101-110.

[12] Accelerator support in future OpenMP. <http://openmp.org/wp/2011/02/31-draft-specs-ready-for-public-comment/>

[13] P. Hanrahan. *Domain-Specific Languages for Heterogeneous GPU Computing*. <http://www.graphics.stanford.edu/~hanrahan/talks/dsl/dsl1.pdf>

[14] Gromacs. < http://www.gromacs.org/>

[15] C. Luk, S. Hong, and H. Kim. 2009. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of* MICRO 42. ACM, New York, NY.