Extending OpenACC for Efficient Stencil Code Generation and Execution by Skeleton Frameworks

Alyson D. Pereira,
Márcio Castro, Mario A. R. Dantas
Federal University of Santa Catarina
Florianópolis, Brazil
alyson.pereira@posgrad.ufsc.br,
marcio.castro@ufsc.br, mario.dantas@ufsc.br

Rodrigo C. O. Rocha*
University of Edinburgh
Edinburgh, UK
r.rocha@ed.ac.uk

Luís F. W. Góes
Pontifical Catholic University of Minas Gerais
Belo Horizonte, Brazil
lfwgoes@pucminas.br

***Abstract*—The OpenACC programming model simplifies the programming for accelerator devices such as GPUs. Its abstract accelerator model defines a least common denominator for accelerator devices, thus it cannot represent architectural specifics of these devices without losing portability. Therefore, this general-purpose approach delivers good performance on average, but it misses optimization opportunities for code generation and execution of specific classes of applications. In this paper, we propose OpenACC extensions to enable efficient code generation and execution of stencil applications by parallel skeleton frameworks such as PSkel. Our results show that our stencil extensions may improve the performance of OpenACC in up to 28% and 45% on GPU and CPU, respectively. Moreover, we show that the work-partitioning mechanism offered by the skeleton framework, which splits the computation across CPU and GPU, may improve even further the performance of the applications in up to 18%.**

***Keywords*—stencil; skeleton frameworks; source-to-source compilation; CUDA; OpenACC**

I. Introduction

In recent years, Graphics Processing Units (GPUs) have been used in conjunction with general-purpose CPUs to enable High Performance Computing (HPC) with high energy efficiency. While modern CPUs use large caches and provide multiple out-of-order cores with branch prediction and speculation, GPUs are much richer in floating-point units and provide large amounts of simple processing cores. Despite being commonly found in the same hardware platform or even on the same chip, CPUs and GPUs typically have different application programming interfaces. OpenMP [1], CUDA [2] and OpenACC [3] are some of the many programming models currently available for programmers to choose when writing their applications. Leveraging from these different architectures with parallel programming is known to be difficult and error prone, imposing several challenges to the programmer [4].

Indeed, OpenMP and OpenACC are widespread programming models. They provide an easy way for programmers to parallelize their codes based on the use of annotations in the form of compiler directives, which should be added directly to code regions to enable parallel execution on the target platform. Although these directive-based programming models offer a simpler way of parallelizing code, they may generate unoptimized codes for certain classes of applications, which have predictable patterns of data access and communication. The prior knowledge about these patterns could be used to achieve high performance in heterogeneous architectures by minimizing contention for limited resources such as communication and memory bandwidth, keeping data close to processor and overlapping communications and computations during data transfers [5].

A common approach that takes advantage of the pattern of applications is parallel skeletons. Parallel skeletons model and abstract common parallel programming patterns (computation and coordination phases), thereby enabling the programmer to focus on algorithm design, rather than on runtime system details. Among existing parallel skeletons, the stencil pattern is critical in many scientific computing domains, including computational fluid dynamics and image processing. The large amount of recent work targeting GPU implementations of high-performance stencil computations highlights the importance of this pattern [5]–[13].

In this paper, we propose OpenACC extensions to enable efficient code generation and execution of stencil applications by parallel skeleton frameworks such as PSkel [14]. These extensions are designed to expose the stencil pattern to the compiler and runtime system, enabling specific optimizations for this class of applications. We developed a source-to-source compiler that receives as input stencil codes enhanced by the extended OpenACC annotations, generating optimized parallel code suitable for heterogeneous architectures.

We show that our approach is able to generate optimized PSkel source codes for stencil applications using extended OpenACC annotations. Our results show that our approach achieves up to 28% and 45% improvement over OpenACC on GPU and CPU, respectively. Moreover, we show that the work-partitioning mechanism available in PSkel, which splits computation across the CPU and the GPU, may improve even further the performance achieved by the GPU up to 18%.

This paper is organized as follows. Section II provides background on the stencil pattern and high-level programming approaches for GPUs. Section III presents the proposed stencil

* This author is supported by the UK Engineering and Physical Sciences Research Council (EPSRC) under grant EP/L01503X/1.

directives. Section IV describes the compilation process of the stencil annotations into PSkel code. In Section V we present experimental results and demonstrate the relevance of our proposed approach. Finally, Section VI concludes this paper.

II. Background and related work

In this section, we provide background on the stencil parallel pattern and discuss past efforts in providing high-level abstractions for CPU-GPU programming.

A. The stencil pattern

Structured parallel programs follow patterns of computation and coordination. Computation represents the application's logic and data flow control, whereas coordination handles parallelism, concurrency, communication, and load balancing.

An important parallel programming pattern is stencil [4]. The stencil pattern operates on n-dimensional data structures, using an input data value and its neighbors to compute the corresponding output data element. Specifically, a sliding window (also called *mask*) scans the entire set of input data and produces output data using a stencil function. The mask's size corresponds to a specific number of neighbors of each element of the input data. The stencil function performs computations using the mask and the neighbors to produce a corresponding element in the output data. The stencil application repeats this process on every element of the input data, except those at its limits (border cells). In case of iterative stencil applications, the whole procedure described above is repeated in each iteration (a.k.a. *timestep*). At the end of an iteration a *swap* operation occurs, since the output data of an iteration t must be used as the input data of an iteration $t+1$.

The code fragment written in Code 1 exemplifies a stencil computation performed in the Jacobi's method [15]. In this example, the variable `timesteps` holds the number of iterations that must be computed whereas variables `alpha` and `beta` are input parameters. We also assume that the input and output data (`A` and `B`) are 2D matrices of size `M*N` stored as 1D arrays. A more detailed description of this method is provided in Section V-A.

B. High-level GPU and CPU-GPU programming frameworks

Besides the existence of directive-based programming models, there has been extensive work in providing high-level abstractions for simplifying GPU programming. For example, previous works have proposed data-structure and control-flow abstractions via annotated code, compiler directives and new and extended programming languages [9], [10], [16]. In addition to providing high-level abstractions, other works have produced portable parallel code and improved the overall workload performance by exploiting parallel programming patterns and tuning the runtime system, hardware, and/or applications. Steuwer *et al.*, proposed SkelCL, an OpenCL-based skeleton library for multi-GPU programming [11]. In a similar approach, Enmyren & Kessler proposed SkePU [7], a C++ template library that provides genereric containers and implements several skeletons for GPU and CPU execution. Ernstsson *et al.*, [17] improved SkePU to provide a flexible and type-safe programming interface focused on C++11 and variadic templates. The new programming interface is compatible to any C++11 compiler for sequential code generation while a source-to-source tool transforms the code to target OpenMP, OpenCL and CUDA. In a similar approach to our work, Nguteren & Corportal proposed Bones, a source-to-source compiler based on algorithmic skeletons [18]. It transforms annotated C code to parallel CUDA or OpenCL using a translator written in Ruby. The skeleton set is based on a well-defined grammar and vocabulary. However, Bones places strict limitations on the coding style of input programs.

Due to the importance of the stencil pattern, some of those works focused specifically on optimizing stencil applications. For example, PATUS generates and provides auto-tuning abstractions for stencil programs on CPU-GPU systems [6]. The framework provides a domain specific language for kernel description and bandwidth-saving mechanisms, which the programmer can use to draw parallelization and optimization strategies. PSkel provides similar abstractions, with stencil kernels defined in C++ template functions and enables the fraction of computations executed in CPU or GPU to be specified at runtime [14]. Holewinski *et al.*, modified the compiler to automatically reduce memory bandwidth requirements for the GPU in exchange for redundant computations [10]. PARTANS generates optimized OpenCL code and automatically schedules stencil computations across GPUs based on characteristics of the stencil function, problem size, GPU heterogeneity, and PCI-express settings [8].

C. The PSkel stencil framework

PSkel is a framework for high-level programming stencil computations, based on the concept of parallel skeletons, which offers parallel execution support on heterogeneous architectures (CPU and GPU). PSkel offers a single programming interface, decoupled from the runtime back-ends, that releases programmers from the responsibility of writing boiler-plate code for parallel stencil computation. Instead, programmers are responsible for implementing a kernel describing solely the stencil computation, while the framework translates the abstractions described into low-level parallel C++ code, compatible with OpenMP and NVIDIA CUDA, where synchronization, memory management and data transfer is transparently handled by the framework [14].

Given the well-defined structure of skeletons, PSkel is able to perform several optimizations, including the use of fast shared memory of GPUs by means of overlapped trapezoidal tiling techniques. Tiling partitions the iteration space into smaller regular blocks, called tiles. When tiling a stencil computation, neighborhood dependencies must be considered before partitioning the data into smaller blocks. One of the main solutions for handling neighborhood dependencies is via overlapped blocks, resulting in redundant data and computation per tile [10], [12], [13].

Another optimization provided by the PSkel runtime is the partitioning of the stencil input data and the simultaneous

```
 1 void jacobi(float* A, float* B, int M, int N,
 2             float alpha, float beta, int timesteps){
 3   for(int t = 0; t < timesteps; t++){
 4    for(int y = 1; y < M-1; y++)
 5     for(int x = 1; x < N-1; x++)
 6      B[y*N+x] = alpha*(A[(y+1)*N+x] + A[(y-1)*N+x] +
 7                 A[y*N+(x+1)] + A[y*N+(x-1)] + beta);
 8    /* data swap */
 9    for(int y = 1; y < M-1; y++)
10     for(int x = 1; x < N-1; x++)
11      A[y*N+x] = B[y*N+x];
12   }
13 }
```

Code 1. Jacobi stencil computation.

execution of each partition on both CPU and GPU, through a work-partitioning mechanism. Some platforms can take benefit from this optimization, improving even further the performance os applications as shown in Section V-B3.

The PSkel Application Programming Interface (API) provides templates for manipulating input and output data via template classes for n-dimensional arrays, called `Array`, `Array2D`, and `Array3D`. The n-dimensional array abstractions provide methods that encapsulate the data management procedures, such as memory allocation, memory copy, and data transfer (*e.g.*, communication between host and GPU). Specifically to stencil parallel pattern, it also provides abstractions for specifying stencil masks, stencil kernel, and finally manage stencil execution. The stencil kernel (prototype function `stencilKernel()`) is the application's specific method that describes the computation performed on each entry of the input array and must be implemented by the programmer. Moreover, the API provides a set of classes (`Stencil`, `Stencil2D`, and `Stencil3D`) for managing the whole execution of the stencil computation over the input data, *i.e.*, the stencil class manages the input and output data during the execution of the stencil kernel over the input entries and the requested number of iterations. The prototype function `stencilKernel()` may also take additional user-defined parameters incapsulated by a struct (`Arguments`).

Finally, PSkel provides a method for CPU-only execution (`runIterativeCPU()`), GPU-only execution (`runIterativeGPU()`) and partitioned execution between CPU and GPU (`runIterativePartitioned()`). Those methods take the number of timesteps as input parameter. The latter takes an additional one (a number between 0.1 and 0.9) which indicates how the workload is partitioned between CPU and GPU. For example, using `runIterativePartitioned(timesteps, 0.6)` means that PSkel will schedule 60% of computation to GPU and 40% to CPU.

Code 2 exemplifies the use of PSkel framework to compute Jacobi's method (Code 1) on GPU. In this code we omitted the struct declaration for brevity.

D. Directive-based programming models

The directive-based approach of OpenMP and OpenACC provides a simple, yet powerful way for programmers to parallelize their code. Users typically need to identify portions of code that are profitable for parallel execution and annotate

```
 1 __parallel__ void
 2 stencilKernel(Array2D<float> A, Array2D<float> B,
 3               struct Arguments args, int x, int y){
 4   B(x,y) = args.alpha * (A(x,y+1) + A(x,y-1) +
 5            A(x+1,y) + A(x-1,y) + args.beta);
 6 }
 7
 8 void jacobi(float* A, float *B, int M, int N,
 9             float alpha, float beta, int timesteps){
10   Array2D<float> input(A,M,N);
11   Array2D<float> output(B,M,N);
12
13   struct Arguments args(alpha, beta);
14
15   Stencil2D<Array2D<float>, struct Arguments>
16            jacobi(input,output,args);
17   jacobi.runIterativeGPU(timesteps);
18 }
```

Code 2. PSkel Stencil kernel function.

them using appropriate directives. For instance, code sections containing nested loops are usually good candidates for parallelization on multicores or GPUs, and are annotated using `#pragma omp parallel for` and `#pragma acc parallel loop` directives for OpenMP and OpenACC, respectively. Nevertheless, these directive-based approaches have some shortcomings in terms of the generated code.

Consider the code fragment written in Code 1. A simple way to execute this on GPU is to annotate it with OpenACC pragmas as shown in Code 3. An OpenACC compiler analyzes the annotated code and transparently maps the loops to GPU threads according to an internal cost model. However, the compiler may not generate very optimized binaries for such codes. This is because architectural-specific features such as software managed caches (known as shared memory in the CUDA programming model) are usually not properly exploited by the compiler in the code generation [19].

III. OpenACC stencil extensions

In this section, we present the OpenACC extensions for the stencil pattern. We propose a `stencil` directive to inform the compiler about specific features of the stencil parallel skeleton as well as for its execution on a heterogeneous architecture. The syntax of the new OpenACC `stencil` directive proposed in this paper is

`#pragma acc stencil` *clause-list*

where *clause-list* is a list of all the following *clauses*, except *args* which is not mandatory. A *var-list* is a comma-separated list of one or more array names (*vars*) with their respective dimensions:

`device(` *device-type[, float]* **`)`**
`iterations(` *int* **`)`**
`input(` *var***`[`***int[, int[, int]]***`]`** **`)`**
`output(` *var***`[`***int[, int[, int]]***`]`** **`)`**
`args(` *var-list* **`)`**

The use of the `stencil` directive as well as its possible clauses are shown in Code 4 on lines 3 and 4. The remaining code is exactly the same as shown on lines 3 to 21 of Code 3.

The `device` clause defines the target architecture for the execution of the stencil computation. The *device-type* can be

```
1  void jacobi(float* A, float* B, int M, int N,
2              float alpha, float beta, int timesteps){
3   #pragma acc data copy(A[0:M*N]) create(B[0:M*N])
4   {
5    for(int t = 0; t < timesteps; t++){
6     #pragma acc parallel loop
7     for(int y = 1; y < M-1; y++){
8      #pragma acc loop independent
9      for(int x = 1; x < N-1; x++)
10      B[y*N+x] = alpha*(A[(y+1)*N+x] + A[(y-1)*N+x] +
11                 A[y*N+(x+1)] + A[y*N+(x-1)] + beta);
12    }
13    /* data swap */
14    #pragma acc parallel loop
15    for(int y = 1; y < M-1; y++){
16     #pragma acc loop independent
17     for(int x = 1; x < N-1; x++)
18      A[y*N+x] = B[y*N+x];
19    }
20   }
21  }
22 }
```

Code 3. Jacobi stencil code with OpenACC annotations.

either `cpu` or `gpu`, since the PSkel runtime system supports execution on CPU and GPU. For a partitioned execution between CPU and GPU, the percentage of the input data processed by the GPU or CPU must be informed (*float*) and must be between 0.1 and 0.9. In Code 4, the whole computation will be performed by the GPU.

The `iterations` clause defines the number of iterations performed by the stencil. Its value must be informed by the same variable that bounds the following loop of the stencil iterations. In Code 4, the number of iterations are defined by the variable `timesteps` that is used as a bound to the `for` loop on line 5 of Code 3.

The `input` and `output` clauses define, respectively, the stencil input and output data. Their values must be set to their corresponding array pointers used in the stencil computations, and their dimensions must be informed, indicating their respectively height, width and/or depth (at least the width must be informed). If the remaining dimensions are not informed, the compiler uses the default value of 1. Their value must be informed by the same variables that bounds the stencil parallel loop execution. In Code 4, width and height are informed by `M` and `N` variables, respectively. These variables are used as loop bounds on lines 7 and 9 of Code 3.

It is worth noting that there is no need to inform in the `stencil` directive any primitive data variables used in the stencil computation. The proposed source-to-source compiler presented in the next section detects if the stencil computation makes use of other variables to be able to generate a correct and equivalent PSkel code. For instance, in Code 3, `alpha` and `beta` variables used in the stencil computation will be accessed from a struct in the resulting PSkel source code. However, if the stencil computation uses additional data such as dynamic arrays, their pointers and dimensions must be informed using the `args` clause, which is similar to the `input` and `output` clauses.

IV. Source-to-source compiler

We propose a source-to-source compiler that takes an input C/C++ code, which is enhanced by extended OpenACC annotations, to take advantage of heterogeneous architectures as discussed in Section III. Since PSkel provides the necessary runtime system for executing stencil computations on heterogeneous architectures, the output of the source-to-source compiler is a PSkel C/C++ code targeted for multicore processors or CUDA-capable GPUs. In this section we discuss the details of the source code generation. Figure 1 shows the diagram of the compilation process which consists of four major stages. Our source-to-source compiler is currently implemented in Python using *pycparser*[1] to aid the compilation steps.

```
1 void jacobi(float* A, float* B, int M, int N,
2             float alpha, float beta, int timesteps){
3   #pragma acc stencil device(gpu) iterations(timesteps) \
4                       input(A[M,N]) output(B[M,N])
5   {
6      /* OpenACC code (lines 3 to 21 of Code 3) */
7   }
8 }
```

Code 4. Jacobi stencil code with extended OpenACC annotations.

The **Frontend Parser** receives an input C/C++ source code, enhanced by extended OpenACC annotations, which include a stencil specific directive and its clauses. Then, it extracts the functions composed by loops annotated with the aforementioned directive and produces an annotated Abstract Syntax Tree (AST) for each given function.

The **Extended OpenACC Preprocessor** traverses the stencil constructions using the extended OpenACC annotations. For each stencil construction found, it extracts information from the directive that provides relevant information for identifying the target device, number of stencil iterations, and the n-dimensional input and output arrays, as well as their dimensions and sizes.

Informed by stencil directive, the **Loop Analyzer** validates the semantics of nested loops, annotating the AST nodes for code generation. It first verifies that the nested loops are affine, with well defined induction variables, by means of a symbolic induction variable analysis using scalar evolution [20], [21]. An induction variable is a monotonic variable that increases (or decreases) by a fixed amount on every iteration of a loop. It can also be a linear function of another induction variable, also called induction expression. Scalar evolution is used primarily to symbolically analyze expressions involving induction variables in loops.

The Loop Analyzer expects nested loops with induction variables bounded by the sizes of the n-dimensional arrays. The outer-most loop can also be a loop responsible for counting the stencil iterations. We use induction variable analysis and data dependence analysis [22] in order to identify if the outer-most loop is responsible for the stencil iterations. For this loop, the induction variable must only be used for controlling the loop iterations, i.e., the induction variable must not be used inside the loop body. The Loop Analyzer also verifies data dependencies regarding the input and output arrays. If the Loop Analyzer is unable to fully verify all the aforementioned prerequisites, we fall back to standard OpenACC annotations.

[1]https://github.com/eliben/pycparser

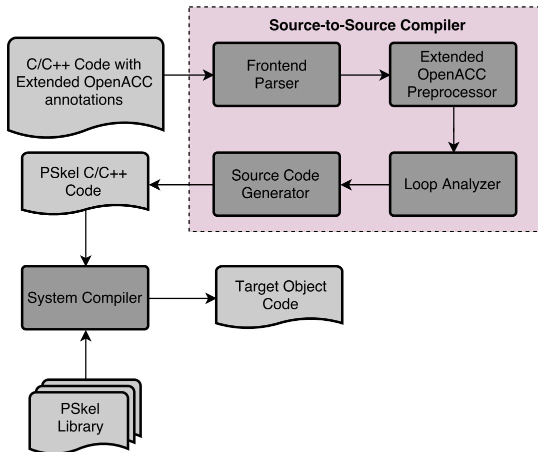

Fig. 1. Overall architecture of the compilation process.

In addition to the verification of the previously defined prerequisites, the Loop Analyzer also tries to infer the range of the stencil neighborhood. If this neighborhood information is present, the PSkel runtime framework is able to perform further optimisations [13]. Given that we know the size of the array dimensions, the Loop Analyzer uses scalar evolution to perform delinearization of the array access expressions (subscripts) [23]–[25]. After the array subscripts are delinearized, it tries to infer the neighborhood range by identifying the maximum displacement. Although this analysis works well in well-defined array subscripts, such as the example shown in Code 1, it is still limited and not always possible.

The **Source Code Generator** traverses the annotated AST producing C/C++ code, transforming loops annotated with stencil directives into PSkel equivalent code. For instance, a code equivalent to Code 2 is generated from Code 3 with the stencil annotations of Code 4. The body of the innermost loop is extracted into an implementation of the stencil kernel function, the prototype function `stencilKernel()` in PSkel (lines 1-6 of Code 2), and the original AST node of the whole stencil construction is replaced by a block of PSkel code that performs the setup of the stencil runtime system and the launch of the iterative stencil execution on the specified devices. The PSkel setup code consists of lines 10-17 of the example presented in Code 2. Array pointers are wrapped by PSkel n-dimensional array abstractions (lines 10 and 11) and the setup code, after wrapping the array pointers, instantiates a data structure for internalizing extra data variables necessary for the stencil computation. Every variable used inside the stencil nested loops, which was defined in a prior scope, is considered an extra argument during code transformation, and therefore passed to the stencil kernel function by the argument data structure. In Code 2, these extra variables are the `alpha` and `beta` variables (line 13). Finally, the stencil abstraction is instantiated (lines 15 and 16) for managing the execution of the iterative stencil computation (line 17).

The final stage invokes a **System Compiler** (*e.g.*, NVCC for CUDA back-end, GCC with OpenMP for a multicore back-end, etc.) to generate the target object code. Input to this stage is the PSkel runtime library and the PSkel-enabled C/C++ source code previously generated by the source-to-source compiler.

TABLE I
SYSTEM SETTINGS.

Features	Platform 1		Platform 2	
	CPU	GPU	CPU	GPU
Manufac.	Intel	NVidia	Intel	NVidia
Model	Xeon E5-2620	Tesla K20	Xeon E5-5645	Quadro 2000
Clock	2.3 GHz	706 MHz	2.4 GHz	1.25 GHz
Cores	6 + HT	2496	2×6	192
LLC size	15 MB	1.31 MB	12 MB	262 KB
DRAM	32 GB	5 GB	32 GB	1 GB

V. EXPERIMENTAL RESULTS

In this section, we present our experimental setup and results. Table I presents the settings of two computing platforms used throughout the experiments, both running CentOS 6.7 with NVIDIA CUDA version 7.5, GCC version 4.9.2, and PGI Accelerator compiler version 16.5, which supports OpenACC 2.5. We used Platform 1 to evaluate the performance gain when using the proposed stencil annotations compared to a pure OpenACC implementation, as described in Sections V-B1 and V-B2. Platform 2 is used to demonstrate the extra performance gain that can be obtained when using the PSkel work-partitioning feature, described in Section V-B3.

A. Stencil Applications

We considered three 2D stencil applications from different domains. **Game of Life** (GoL) is a cellular automaton implementing Conway's Game of Life [26] as a stencil application. The automaton is represented as a matrix where each element is a living or a dead individual and the stencil mask determines the interaction between an individual and its neighbors. Over the course of a pre-defined number of iterations, each individual analyzes the state of its neighbors to determine its own state in the next iteration according to GoL rules.

The **Jacobi** method is an iterative method for solving matrix equations [15]. The method is guaranteed to converge if the input matrix is strictly or irreducibly diagonally dominant, i.e., $|u_{i,i}| > \sum_{j \neq i} |u_{i,j}|$, for all i. Equation 1 defines the computation performed at each step of the Jacobi's iterative method for solving 2-dimensional Poisson's elliptical discretized equation, as shown in Code 1. The approximate solution is computed by discretizing the problem in a matrix of $n \times n$ evenly spaced points. Poisson's equation [15] is a partial differential equation of elliptic type largely used in theoretical physics.

$$u'_{i,j} = \frac{u_{i\pm1,j} + u_{i,j\pm1} + h^2 f_{i,j}}{4} \quad (1)$$

At each step, the new value of $u_{i,j}$ is obtained by averaging its neighbors with $h^2 f_{i,j}$, where $h = \frac{1}{n+1}$ and $f_{i,j} = f(ih, jh)$, for a given function f.

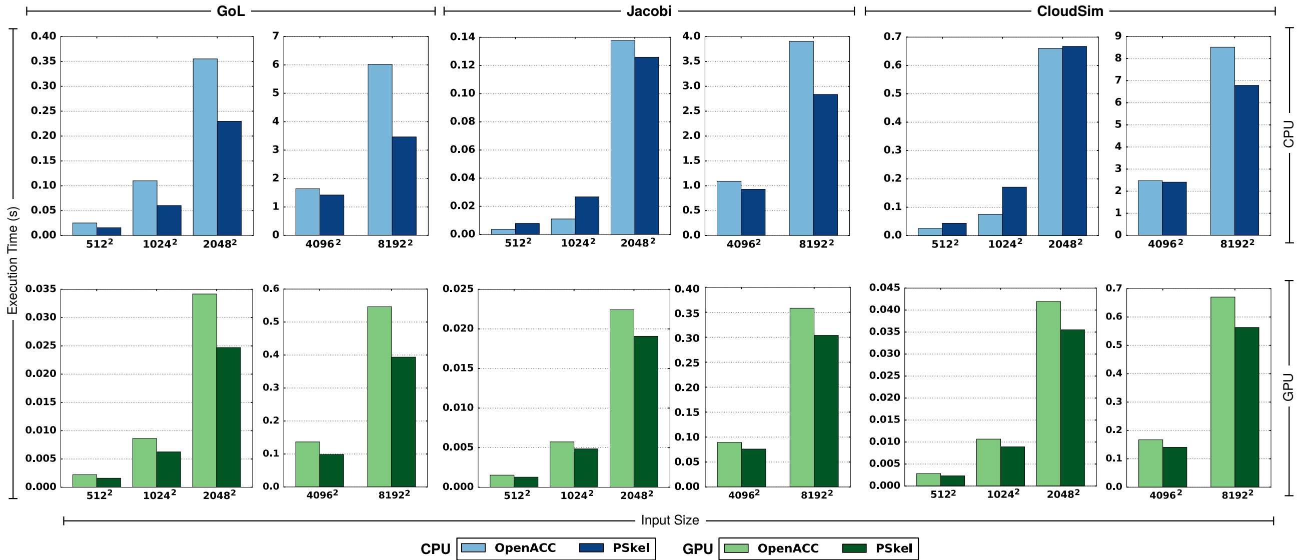

Fig. 2. Execution times of the original OpenACC code and the generated PSkel code from the OpenACC stencil extensions.

CloudSim simulates cloud dynamics based on cellular automaton [27]. The mathematical model uses neighborhoods of five cells. The model uses three weather properties: condensed cloud water particles, temperature and winds. The transition rules are based on the thermodynamic principles and weather concepts. The number of condensed cloud water particles in a cell is defined as a function of its current temperature. These condensed cloud water particles are displaced to a neighborhood in accordance with the wind direction. The cell temperature behavior is made from thermodynamic principles that provide a heat transfer between neighborhood cells.

B. Performance Evaluation

In this section, we compare the performance of the implementations with the original OpenACC code to the generated code from our source-to-source compiler, using the PSkel library as the back-end for stencil execution. Figure 2 shows the execution time for each of the three stencil applications executing on CPU and GPU. For the GPU execution, we only consider kernel computing time (i.e., we disconsider the time spent in memory transfers). The reported values represent the average of 10 executions of each application with a fixed number of iterations of 50. The results presented in Sections V-B1 and V-B2 were all executed in Platform 1.

1) CPU Performance: For the GoL application, PSkel has a better performance than OpenACC for all input data sizes, achieving up to 45% of improvement over OpenACC. The main reason for that is presented in Table II, which shows the last-level cache (LLC) miss ratio for the execution of the three stencil applications used in this paper. As it can be noticed, OpenACC has a higher LLC cache miss ratio while PSkel back-end maintains its consistent data cache miss ratio behavior, achieving a better performance.

For the Jacobi application, OpenACC shows better performance than PSkel back-end for small input sizes (53.35% and 58.74% improvement for 512^2 and 1024^2, respectively), but PSkel back-end outperforms OpenACC up to 27% for the remaining input sizes. Table II corroborates these results, showing that OpenACC has a lower miss ratio for small inputs, but higher values for the remaining inputs.

TABLE II
CPU LLC DATA CACHE MISS RATIO COMPARISON BETWEEN PGI OPENACC COMPILER GENERATED CODE AND PSKEL GENERATED CODE.

API	Input	LLC Miss Ratio (%)		
		GoL	Jacobi	CloudSim
OpenACC	512	7.74	9.57	4.76
	1024	1.40	1.79	1.10
	2048	26.62	27.52	29.49
	4096	38.19	34.51	32.38
	8192	32.27	34.33	30.16
PSkel	512	7.31	6.35	6.56
	1024	4.89	6.89	8.28
	2048	23.78	19.61	20.91
	4096	28.49	22.95	20.67
	8192	23.13	24.73	17.68

Finally, CloudSim showed a behavior similar to Jacobi. Again, the OpenACC code has a better performance for small input sizes with an speedup of $1.73\times$ and $2.27\times$ for input sizes 512^2 and 1024^2, respectively. As shown in Table II, for the same small input sizes, OpenACC has a lower cache miss ratio compared to PSkel. However, for larger input sizes, PSkel has a lower cache miss ratio, thus achieving a better performance, executing up to $1.25\times$ faster.

2) GPU Performance: Given the well-defined structure of the stencil pattern, PSkel is able to use the fast shared memory of GPUs by means of an overlapped trapezoidal tiling technique. Due to this extra optimization, the PSkel runtime system achieves a read throughput between 500 GB/s and 1 TB/s in the shared memory while OpenACC achieves a read throughput of up to 200 GB/s in the L2 cache memory. As

TABLE III
GPU DATA READ THROUGHPUT INTERVALS (MIN – MAX) IN GB/S OF THE ORIGINAL OPENACC AND PSKEL GENERATED CODES.

Application	API	DRAM Throughput	L2 Throughput	Shared Mem. Throughput
GoL	OpenACC	9.07 – 9.94	117.37 – 133.63	–
	PSkel	7.03 – 7.20	14.27 – 15.53	988.26 – 1000
Jacobi	OpenACC	40.72 – 49.13	150.59 – 180.36	–
	PSkel	26.91 – 30.43	34.31 – 39.09	648.71 – 736.91
CloudSim	OpenACC	68.34 – 81.84	176.22 – 199.87	–
	PSkel	47.86 – 58.52	88.16 – 100.91	579.25 – 591.30

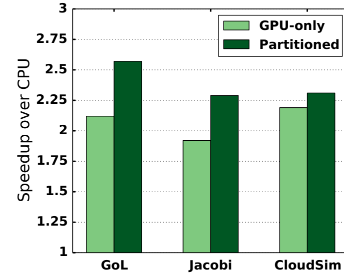

Fig. 3. Speedups of GPU-only and partitioned execution over CPU-only.

result, all applications executing with PSkel back-end achieved a better performance compared to OpenACC. Table III shows the data read throughput in GB/s (minimum and maximum) across the GPU memory hierarchy for the execution of the three stencil applications used in this paper.

We also analyzed the tiling technique, implemented in PSkel, regarding the trade-off between redundant computations and global data communication. For the CloudSim application, compared to the OpenACC code, the PSkel tiling technique increased the amount of floating point operations by 48% on average for all input sizes, while improving the amount of global load instructions with an average reduction of 63%. As shown in Table III, the PSkel code of Cloudsim reduces the DRAM and L2 cache throughput up to 30% and 50%, respectively, in relation to the OpenACC code. The CloudSim application has the highest DRAM read throughput among all applications for both PSkel and OpenACC because its kernel uses two additional matrices to store the wind data. Moreover, the PSkel framework has the limitation that only the main input and output data can make use of the shared memory, thus the extra wind data is cached only on L2. Still, the use of shared memory in PSkel achieves a load throughput up to $2.95\times$ higher than the equivalent read throughput on the L2 cache by OpenACC. Overall, the generated PSkel code improved performance up to 18% compared to OpenACC.

For the GoL application, PSkel back-end achieved the best performance gain, reaching a speedup of $1.39\times$ compared to OpenACC code. The tiling technique in PSkel increases the amount of arithmetic operations by an average of 69%, while achieving an average reduction of load instructions by 92%, when compared to OpenACC. As shown in Table III, GoL application has the lowest data throughput for DRAM and L2 cache among the applications, because it manipulates mainly boolean data (8 bits). When a load instruction is performed by a GPU thread, a block of 128 bits is fetched from the global memory and then cached in L2, therefore reading more data elements per load instruction, when comparing inputs of 8 bits boolean to 32 bits floating-point, resulting in an overall decrease in DRAM reads. The generated PSkel code has a 27% reduction on DRAM reads when compared to OpenACC and also a 88% reduction on L2 reads. Moreover, PSkel back-end achieves the best load throughput in shared memory (1 TB/s with input size of 8192^2).

Finally, for the Jacobi application, PSkel had an average increase of 56% on its floating-point operations and an average reduction of 83% on the total amount of load instructions, compared to OpenACC. PSkel had a reduction on DRAM and L2 load throughput of up to 38% and 78%, respectively. Overall, PSkel achieved a performance gain up to 17% compared to the OpenACC code on the Jacobi application.

3) Work-partitioning: As shown in Figure 2, applications may run up to $4.85\times$ faster on the GPU than the CPU on Platform 1. This result is expected because the GPU has a much higher theoretical peak performance than the CPU. In this case, the work-partitioning mechanism available in PSkel will not bring any performance improvement because the best performance is achieved by running all computations on GPU. However, when the performance of the GPU is modest compared to the CPU, which is the case in Platform 2, partitioning computations on both CPU and GPU can bring extra performance gains. Figure 3 illustrates the performance on Platform 2 of GPU-only and Partitioned (CPU+GPU) compared to CPU-only for all three stencil applications used in this paper. In these experiments we fixed the input size in 8192^2 and executed 50 stencil iterations.

For the GoL application, the Partitioned execution presents a speedup of $2.57\times$ compared to CPU-only, while the GPU-only execution achieved a speedup of $2.12\times$. The Partitioned performance is obtained when executing 70% of the input data size on GPU and the remaining input data on CPU, resulting in an 18% overall performance improvement over GPU-only.

Jacobi shows a speedup of $2.29\times$ for the partitioned execution compared to CPU-only, while the GPU-only execution achieves a speedup of $1.92\times$. The partitioned execution improves 16% over the GPU-only when 65% of computation is performed on GPU.

Finally, speedups of $2.31\times$ and $2.19\times$ can be observed for the Partitioned and GPU-only executions on CloudSim when compared to CPU-only, respectively. The partitioned execution achieves a 5% improvement over the GPU-only execution, which is obtained when executing 85% of the input data size on the GPU. Since the CloudSim application is richer in floating-point operations and the GPU device contains more floating-point units than the CPU, the work-partitioning mechanism achieves its best performance with a higher GPU partition when compared to Jacobi, which contains very few

floating-point operations, and GoL, which is composed of integer operations and conditional statements.

VI. Conclusion and Future work

In this paper, we proposed OpenACC extensions to enable efficient code generation and execution of stencil applications by means of the PSkel framework and runtime system. We developed a source-to-source compiler that receives as input stencil applications enhanced by extended OpenACC annotations with stencil directives and generates optimized parallel code suitable for heterogeneous architectures.

We demonstrated and evaluated the extraction and mapping process from extended OpenACC annotated codes to the PSkel framework, and showed that the resulting generated code achieves better performance when compared to the original OpenACC code due to optimizations that are enabled by the stencil pattern. We also demonstrated that it is possible to improve the performance by using the work-partitioning optimization that is provided by the skeleton framework, which splits computation across CPU and GPU. Another advantage of our approach is that it maintains the performance portability to target new platforms without modifying the original source code as long as a backend for the platform becomes available in the skeleton framework.

As future works, we intend to extend our source-to-source compiler to support different parallel skeletons, such as map, reduce, and scan. Moreover, we are currently porting the proposed source-to-source compiler into the industrial-strength compiler infrastructure Clang/LLVM. We plan to automatically generate optimized CUDA kernels for stencil computations while using PSkel as the runtime component.

Acknowledgments

The authors would like to thank FIP PUC Minas, CAPES, CNPq and FAPEMIG for funding this research. This work was also supported by STIC-AmSud/CAPES under EnergySFE research project grant No. 99999.007556/2015-02.

References

[1] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, 1998.

[2] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[3] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC: First experiences with real-world applications," in *International European Conference on Parallel and Distributed Computing (Euro-Par)*. Rhodes Island, Greece: Springer-Verlag, 2012, pp. 859–870.

[4] M. D. McCool, "Structured parallel programming with deterministic patterns," in *USENIX Conference on Hot Topics in Parallelism (HotPar)*. Berkeley, USA: USENIX Association, 2010, pp. 5–5.

[5] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, "Physis: An implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Seattle, USA: ACM, 2011, pp. 11:1–11:12.

[6] M. Christen, O. Schenk, and Y. Cui, "PATUS for convenient high-performance stencils: Evaluation in earthquake simulations," in *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. Salt Lake City, USA: IEEE Computer Society Press, 2012, pp. 11:1–11:10.

[7] J. Enmyren and C. W. Kessler, "SkePU: A multi-backend skeleton programming library for multi-GPU systems," in *International Workshop on High-level Parallel Programming and Applications (HLPP)*. Baltimore, USA: ACM, 2010, pp. 5–14.

[8] T. Lutz, C. Fensch, and M. Cole, "PARTANS: An Autotuning Framework for Stencil Computation on multi-GPU Systems," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 59:1–59:24, 2013.

[9] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "A stencil compiler for short-vector SIMD architectures," in *ACM International Conference on Supercomputing (ICS)*. Eugene, USA: ACM, 2013, pp. 13–24.

[10] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on GPU architectures," in *ACM International Conference on Supercomputing (ICS)*. Venice, Italy: ACM, 2012, pp. 311–320.

[11] M. Steuwer, P. Kegel, and S. Gorlatch, "SkelCL: A portable skeleton library for high-level GPU programming," in *IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW)*. Shanghai, China: IEEE Computer Society, 2011, pp. 1176–1182.

[12] J. Meng and K. Skadron, "A performance study for iterative stencil loops on GPUs with ghost zone optimizations," *International Journal of Parallel Programming*, vol. 39, no. 1, pp. 115–142, 2011.

[13] R. C. O. Rocha, A. D. Pereira, L. Ramos, and L. F. W. Ges, "TOAST: Automatic tiling for iterative stencil computations on GPUs," *Concur. and Comp.: Practice and Experience*, vol. 29, no. 8, pp. 1–16, 2017.

[14] A. D. Pereira, L. Ramos, and L. F. W. Góes, "PSkel: A stencil programming framework for CPU-GPU systems," *Concur. and Comp.: Practice and Experience*, vol. 27, no. 17, pp. 4938–4953, 2015.

[15] J. W. Demmel, *Applied numerical linear algebra*. Philadelphia, USA: Society for Industrial and Applied Mathematics, 1997.

[16] M. Steuwer, T. Remmelg, and C. Dubach, "Lift: A functional data-parallel IR for high-performance GPU code generation," in *International Symposium on Code Generation and Optimization (CGO)*. Austin, USA: IEEE, 2017, pp. 74–85.

[17] A. Ernstsson, L. Li, and C. Kessler, "SkePU2: Flexiblepol and type-safe skeleton programming for heterogeneous parallel systems," *International Journal of Parallel Programming*, pp. 1–19, 2017. [Online]. Available: http://dx.doi.org/10.1007/s10766-017-0490-5

[18] C. Nugteren and H. Corporaal, "Bones: An automatic skeleton-based C-to-CUDA compiler for GPUs," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 35:1–35:25, 2014.

[19] A. Lashgar and A. Baniasadi, "Employing software-managed caches in OpenACC: Opportunities and benefits," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 1, no. 1, pp. 2:1–2:34, 2016.

[20] M. P. Gerlek, E. Stoltz, and M. Wolfe, "Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form," *ACM Trans. Program. Lang. Syst.*, vol. 17, no. 1, pp. 85–122, 1995.

[21] S. Pop, A. Cohen, and G.-A. Silber, *Induction Variable Analysis with Delayed Abstractions*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 218–232.

[22] D. E. Maydan, J. L. Hennessy, and M. S. Lam, "Efficient and exact data dependence analysis," in *ACM Conf. on Program. Lang. Design and Implem. (PLDI)*. Toronto, Canada: ACM, 1991, pp. 1–14.

[23] V. Maslov, "Delinearization: An efficient way to break multiloop dependence equations," in *ACM Conf. on Program. Lang. Design and Implem. (PLDI)*. San Francisco, USA: ACM, 1992, pp. 152–161.

[24] A. Simbürger and A. Größlinger, "On the variety of static control parts in real-world programs: from affine via multi-dimensional to polynomial and just-in-time," in *International Workshop on Polyhedral Compilation Techniques*, Vienna, Austria, 2014, pp. 1–10.

[25] T. Grosser, S. Pop, J. Ramanujam, and P. Sadayappan, "On recovering multi-dimensional arrays in Polly," in *International Workshop on Polyhedral Compilation Techniques*, Amsterdam, Netherlands, 2015, pp. 1–9.

[26] M. Gardner, "Mathematical games: The fantastic combinations of John Conway's new solitaire game "Life"," *Scientific American*, vol. 223, no. 3, pp. 120–123, 1970.

[27] A. R. da Silva and M. M. Gouvêa, Jr., "Cloud dynamics simulation with cellular automata," in *Summer Computer Simulation Conference (SCSC)*. Ottawa, Canada: Society for Computer Simulation International, 2010, pp. 278–283.