Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 108C (2017) 2333–2337

Procedia Computer Science

International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland

Enabling efficient stencil code generation in OpenACC

Alyson D. Pereira[1], Rodrigo C. O. Rocha[2], Márcio Castro[1], Luís F. W. Góes[3], and Mario A. R. Dantas[1]

[1] Universidade Federal de Santa Catarina, Florianópolis, Brazil
alyson.pereira@posgrad.ufsc.br, marcio.castro@ufsc.br, mario.dantas@ufsc.br
[2] University of Edinburgh, Scotland, United Kingdom
r.rocha@ed.ac.uk
[3] Pontifícia Universidade Católica de 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 stencil extensions to enable efficient code generation in OpenACC. Our results show that our stencil extensions may improve the performance of OpenACC in up to 28% and 45% on GPU and CPU, respectively.

© 2017 The Authors. Published by Elsevier B.V.
Peer-review under responsibility of the scientific committee of the International Conference on Computational Science

Keywords: Stencil, parallel skeletons, source-to-source transformation, OpenACC, CUDA

1 Introduction

Structured parallel programs typically follow patterns of computation and coordination. Computation represent the application's logic and data flow control, whereas coordination handles parallelism, concurrency, communication, and load balancing. An important parallel pattern is stencil. This pattern operates on n-dimensional data structures, using an input data value and its neighbors to compute the corresponding output data element. The stencil pattern is critical in many scientific computing domains, including computational fluid dynamics and image processing.

The directive-based approach of OpenACC for GPU programming provides a simple yet powerful way for programmers to parallelize their code. Users need to identify portions of code that are profitable for parallel execution and annotate them using appropriate directives. For instance, the code fragment in Code 1 demonstrates the use of the OpenACC annotations. The `#pragma acc parallel loop` directive indicates that the loop should be executed by the GPU, while the `#pragma acc data` directive indicates which data should be allocated and copied from the host system to the GPU memory. An OpenACC compiler analyzes the annotated code and transparently maps the loops to GPU threads. However, the compiler may not generate very optimized binaries for such codes. This is because architectural-specific features such as software

1877-0509 © 2017 The Authors. Published by Elsevier B.V.
Peer-review under responsibility of the scientific committee of the International Conference on Computational Science
10.1016/j.procs.2017.05.155

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

Code 1: Jacobi stencil code with default OpenACC annotations.

managed caches (known as shared memory in the CUDA programming model) are usually not properly exploited by the compiler in the code generation [3].

A prior knowledge that the given code belongs to the stencil pattern could be used to achieve a better performance. A common approach that takes advantage of the pattern of applications is the use of 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. Some available skeleton-based programming frameworks includes: SkelCL [6], an OpenCL-based skeleton library for multi-GPU programming; SkePU [1], composed by a template library that is compatible to any C++11 compiler for sequential code generation and a source-to-source tool to target OpenMP, OpenCL and CUDA; and PSkel [5], a framework focused on the stencil pattern.

In this paper, we propose OpenACC extensions to enable efficient code generation and execution of stencil applications by parallel skeleton frameworks. 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 applications enhanced by the extended OpenACC stencil directives, generating optimized parallel code suitable for heterogeneous architectures through the PSkel framework.

2 The PSkel stencil framework

PSkel is a skeleton-based framework for high-level programming stencil computations, which offers parallel execution support on CPU and GPU. The PSkel API provides the `Array` templates classes for manipulating n-dimensional input and output data. The prototype function `stencilKernel` is the application specific method that describes the computation performed on each entry of the input array. The API also provides a set of classes for managing the whole execution of the stencil computation over the input data. Code 2 exemplifies the use of the PSkel framework to perform on the GPU the same stencil computation (Code 1).

Given the well-defined structure of skeletons, PSkel is able to perform several optimizations, including the use of the fast shared memory of GPUs using an overlapped trapezoidal tiling technique. Tiling partitions the iteration space into smaller regular blocks (tiles). When tiling stencil computation, neighborhood dependencies inherent to stencil computations 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 [2, 4].

```
1  __parallel__ void
2  stencilKernel(Array2D<float> A, Array2D<float> B, Mask2D<int> mask,
3                struct Arguments args, int x, int y){
4    B(x,y) = args.alpha * (A(x,y+1) + A(x,y-1) + A(x+1,y) + A(x-1,y) - args.beta);
5  }
6
7  void main(){
8    /* ... */
9    Array2D<float> input(A,M,N);
10   Array2D<float> output(B,M,N);
11   int neighbors = {{0,1}, {-1,0}, {1,0}, {-1,0}};
12   Mask2D<int> mask(4, neighbors);
13   struct Arguments args(alpha, beta);
14   /* ... */
15   Stencil2D<Array2D<float>, Mask2D<int>, Arguments> jacobi(A,B,args);
16   jacobi.runIterative(device::GPU, timesteps, 1.0);
17 }
```

Code 2: PSkel Stencil kernel function.

3 OpenACC stencil extensions

We propose directives for the extended OpenACC to inform the compiler about specific features of the stencil parallel skeleton as well as for its execution on a heterogeneous architecture. The use of the stencil directives is shown in Code 3 on line 1. The remaining code is exactly the same as the fragment shown in Code 1.

The `device` directive defines the target architecture for the execution of the stencil computation. The PSkel runtime system supports execution on CPU, GPU, or both simultaneously. For a partitioned execution between host CPU and GPU device, the percentage of the input data processed by the GPU must be informed, ranging from 0.0 (CPU-only) up to 1.0 (GPU-only).

The `iterations` directive 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 the aforementined code line, the number of iterations are defined by variable `timesteps`, and it is used as a bound to the `for` loop on line 3 of Code 1.

The `input` and `output` directives define, respectively, the stencil input and output data. Their values must be set to their corresponding array pointers used on the stencil computations, and their dimensions must be informed, indicating their respectively height, width and/or depth. Their value must be informed by the same variables that bounds the stencil parallel loop execution. In Code 1, the width and height directives are informed by the `N` and `M` variables, respectively. These variables are used as loop bounds on lines 5 and 7 of Code 1.

There is no need to inform in the `acc stencil` directive primitive data variables used on the stencil computation. The proposed source-to-source compiler detects if the stencil computation makes use of other variables to be able generate a correct and equivalent PSkel code. For instance, on Code 1 the `alpha` and `beta` variables used on the stencil computation will be acessed from a struct on 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` directive, which is similar to the `input` and `output` directives.

```
1  #pragma acc stencil device(gpu,1) iterations(timesteps) input(A[M,N]) output(B[M,N])
2  {
3      /* OpenACC code */
4  }
```

Code 3: Jacobi stencil code with extended OpenACC annotations.

4 Source-to-source compiler

The compilation process starts with an input C/C++ code which is enhanced by extended OpenACC annotations as discussed in Section 3 to take advantage of heterogeneous architectures. 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.

The **Frontend Parser** parses the received input C/C++ source code, enhanced by extended OpenACC annotations, which includes stencil specific directives, and produces an annotated Abstract Syntax Tree (AST). The **Extended OpenACC Preprocessor** traverses the stencil constructions using the extended OpenACC annotations. For each stencil construction found, it extracts information from the directives, which provides relevant information for identifying the target device, number of stencil iterations, and the input and output n-dimensional arrays, as well as their dimensions and sizes. Informed by stencil directives, the **Loop Analyzer** validates the semantics of nested loops, annotating the AST nodes for code generation. Based on information extracted from the stencil directives, the loop analyzer identifies the occurrences of the variables related to the input and output n-dimensional arrays, and also the nested loops responsible for the stencil iterations and traversing the entries of the input array.

The **Source Code Generator** traverses the annotated AST producing C/C++ code, transforming loops annotated with stencil directives into PSkel equivalent code. The body of the inner-most loop is transformed into an implementation of the stencil kernel function, the prototype function `stencilKernel` in PSkel; the array pointers are wrapped by PSkel n-dimensional array abstractions; 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 final stage invokes a **System Compiler** (*e.g.*, NVCC for CUDA backend, GCC with OpenMP for a multicore backend, etc.) to generate the target object code. Input to this stage is the PSkel runtime library and the PSkel-enabled C/C++ source code that was previously generated by the source-to-source compiler.

5 Experimental results

In this section, we present and discuss our experimental results. We compare the performance of the implementations with the original OpenACC code to the extended OpenACC code with PSkel as the backend for the stencil execution. We considered three stencil applications from different domains: a stencil implementation of the Conway's Game of Life(GoL), the Jacobi iterative method for solving matrix equations and a cloud dynamics simulator (Cloudsim). We carried out our experiments on a six-core Intel Xeon E5 processor with 64GB of RAM and a NVIDIA Tesla K20 GPU, running on CentOS 6.7. To build the applications, we used NVIDIA CUDA version 7.5, GCC version 4.9.2, and PGI Accelerator compiler version 16.5.

Figure 1 shows the execution time for each of the three stencil applications executing on CPU and GPU. The reported values represent the average of 10 executions with an input size of 8192^2 and 50 stencil iterations. For both CPU and GPU executions, PSkel presents a better performance than OpenACC. The main reason for the performance gain over OpenACC on CPU is the cache miss ratio on the last-level cache (LLC). The code generated by OpenACC has an average miss ratio of 32.25%, while the PSkel generated code has 21.85%. The GoL application achieves the best performance, corresponding to 45% of improvement over OpenACC, while Jacobi and Cloudsim improves the execution time by 27% and 20%, respectively.

For the GPU execution, PSkel is able to use the fast shared memory of GPUs. Due to this extra optimization, the PSkel runtime system achieves a read throughput between 500 GB/s and 1 TB/s as it uses the shared memory, instead on relying on the L2 cache memory as OpenACC

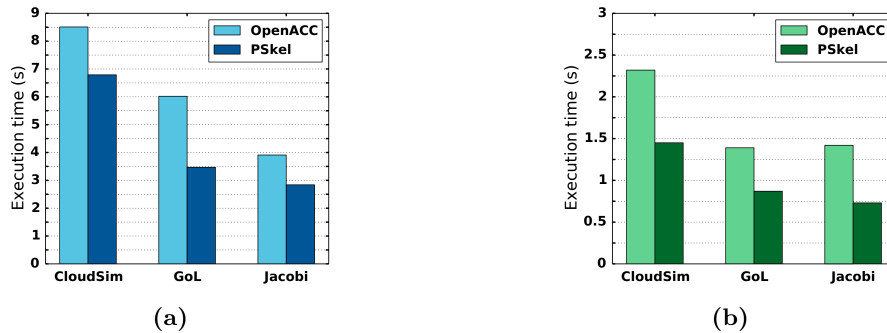

Figure 1: Execution times on CPU (a) and GPU (b) of the original OpenACC code and the generated PSkel code from the OpenACC stencil extensions.

does. Moreover, this technique reduces the amount of global memory loads by up to 92%. As a result, all applications executing with the PSkel backend achieved a better performance compared to OpenACC. Overall, the generated PSkel code improved the performance up to 28% compared to OpenACC for the GoL application, while Jacobi and Cloudsim improves the execution time by 18% and 17%, respectively.

6 Conclusions and future work

In this paper, we have proposed OpenACC extensions to enable efficient code generation and execution of stencil applications by means of the PSkel parallel skeleton framework as the runtime system. We developed a source-to-source compiler that receives as input stencil applications enhanced by the 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 the optimizations that are enabled by the stencil pattern. As future works, we intend to extend our source-to-source compiler to support different parallel skeletons. Moreover, we also plan to include support of the proposed stencil extensions to an open source OpenACC compiler.

References

[1] August Ernstsson, Lu Li, and Christoph Kessler. Skepu2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. *International Journal of Parallel Programming*, pages 1–19, 2017.

[2] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on GPU architectures. In *ACM International Conference on Supercomputing (ICS)*, pages 311–320, Venice, Italy, 2012. ACM.

[3] Ahmad Lashgar and Amirali Baniasadi. Employing software-managed caches in OpenACC: Opportunities and benefits. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 1(1):2:1–2:34, February 2016.

[4] Jiayuan Meng and Kevin Skadron. A Performance Study for Iterative Stencil Loops on GPUs with Ghost Zone Optimizations. *Int. Journal of Parallel Programming*, 39(1):115–142, 2011.

[5] Alyson D. Pereira, Luiz Ramos, and Luís F. W. Góes. PSkel: A stencil programming framework for CPU-GPU systems. *Concur. and Comp.: Practice and Experience*, 27(17):4938–4953, 2015.

[6] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. SkelCL: A Portable Skeleton Library for High-Level GPU Programming. In *IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW)*, pages 1176–1182, Shanghai, China, 2011. IEEE Computer Society.