OpenARC: Open Accelerator Research Compiler for directive-based, efficient heterogeneous computing

Conference Paper · June 2014

DOI: 10.1145/2600212.2600704

2 authors:

Seyong Lee
Oak Ridge National Laboratory
45 PUBLICATIONS **1,028** CITATIONS
SEE PROFILE

Jeffrey Vetter
University of Tennessee
256 PUBLICATIONS **5,423** CITATIONS
SEE PROFILE

OpenARC: Open Accelerator Research Compiler for Directive-Based, Efficient Heterogeneous Computing

Seyong Lee
Oak Ridge National Laboratory
Oak Ridge, TN 37831, USA
lees2@ornl.gov

Jeffrey S. Vetter
Oak Ridge National Laboratory
Oak Ridge, TN 37831, USA
Georgia Institute of Technology
Atlanta, GA 30332, USA
vetter@computer.org

ABSTRACT

This paper presents Open Accelerator Research Compiler (OpenARC): an open-source framework that supports the full feature set of OpenACC V1.0 and performs source-to-source transformations, targeting heterogeneous devices, such as NVIDIA GPUs. Combined with its high-level, extensible Intermediate Representation (IR) and rich semantic annotations, OpenARC serves as a powerful research vehicle for prototyping optimization, source-to-source transformations, and instrumentation for debugging, performance analysis, and autotuning. In fact, OpenARC is equipped with various capabilities for advanced analyses and transformations, as well as built-in performance and debugging tools. We explain the overall design and implementation of OpenARC, and we present key analysis techniques necessary to efficiently port OpenACC applications. Porting various OpenACC applications to CUDA GPUs using OpenARC demonstrates that OpenARC performs similarly to a commercial compiler, while serving as a general research framework.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*code generation, compilers*

Keywords

OpenACC; compiler; source-to-source translation; CUDA; GPU; OpenARC

1. INTRODUCTION

Today's architectures are growing more complex as hardware architects respond to the constraints of energy, density, facilities, and device technology trends [4]. Scalable Heterogeneous Computing (SHC) platforms, enabled by graphics processors, Intel Xeon Phi, etc., are clearly emerging as solutions to these challenges [13]. However, this trend comes at a cost of portability and productivity: to make use of these platforms, scientists must program and optimize a collection of multiple programming models (e.g., MPI, OpenMP, and CUDA) simultaneously to use these scalable heterogeneous systems. Consequently, the community is rethinking the design of the entire programming system in the hope of solving, or at least mitigating, this challenge.

In this regard, several efforts are investigating directive-based programming models [5, 6, 7, 9, 10] as a solution to this productivity and portability challenge. Among them, OpenACC [10] is the first standardization effort to provide portability across device types and compiler vendors. A major benefit of using directive-based programming models is that they transparently relieve programmers from dealing with complexity of low-level programming languages, such as CUDA or OpenCL, and they also hide most of the complex optimizations that are specific to underlying architectures, such as managing data movement into scratchpad memories.

Conceptually, however, these abstractions must strike a delicate balance. At one extreme, too much abstraction in the models restricts the ability of scientists to debug, port, and tune applications. On the other extreme, too little abstraction results in applications without portability. As a compromise, in many of these programming systems, scientists have added performance-critical hints to inform compilers of various compile-time and runtime optimizations to better utilize the underlying architectural resources, such as cores and specialized memory. In previous work, we investigated existing compilers for GPUs in order to better understand this balance on these emerging programming models; we identified several important issues, such as functionality, scalability, tunability, and debuggability. [8].

The lessons learned from the directive-based programming model study [8], along with observations from a comprehensive study of DOE applications, have motivated us to develop a new research compiler, called OpenARC [1], to allow us to investigate these issues in directive-based, high-level programming models. Here are the main contributions of this paper:

- We present the first open-source compiler that supports *full features* in the OpenACC standard V1.0 [10]. OpenARC has several salient features. First, OpenARC's very high-level IR allows high-level source-to-source translation, offering better readability of the output codes and more enhanced debugging environment than existing OpenACC compilers. Second, modular design of OpenARC enables clear separation between compiler passes, which communicate with each other through a rich set of annotations. Hence, within OpenARC, we can implement various traceability mechanisms to establish connections between input directive models and output performance.

- The OpenARC framework is shipped with additional types of directives. Combined with built-in tuning tools, these extensions allow programmers to have a fine-grained control over the overall OpenACC-to-GPU translation and optimizations in an abstract manner.

- We design and implement compiler analyses, code transformations, and runtime support for OpenACC-to-GPU translation. We also discuss how traditional parallelization techniques should be adjusted to the OpenACC execution model to preserve the correct program semantics.

- We evaluate thirteen OpenACC programs from various application domains and compare the results against those translated by the PGI-OpenACC compiler. The results show that OpenARC performs similarly to the commercial compiler.

2. OPENARC: OPEN ACCELERATOR RESEARCH COMPILER

OpenARC is the first open-source compiler supporting full features of OpenACC V1.0, which takes C-based OpenACC programs as inputs and generates accelerator-specific output codes. There exist open-source implementations partially supporting OpenACC [11, 12], but they are developed as fast-prototyping tools with limited contexts, while OpenARC supports full research across all OpenACC functionality.

2.1 Extensible Program Representation

Built on top of the Cetus compiler infrastructure [2], OpenARC's program representation inherits several of its predecessor's salient features. OpenARC's IR is implemented in the form of a Java class hierarchy, and it provides an Abstract Syntax Tree (AST)-like syntactic *view* of the input program that makes it easy for compiler-pass writers to analyze and transform the input program. The hierarchical IR class structure provides complete data abstraction such that compiler-pass writers can manipulate the objects only through access functions. OpenARC supports the following important features derived from Cetus.

- *Traversable IR objects.* All OpenARC IR objects extend a base class *Traversable*, which provides the basic functionality to iterate over IR objects. Combined with various built-in iterators (e.g., *BreadthFirst*, *DepthFirst*, *Flat*, etc.), these provide easy traversal and search over the AST-like, n-ary tree structures of the program representation.

- *Rich Annotations.* *Annotation* is a base class type to represent any type of annotations used in OpenARC. By deriving this base class, various types of information, such as comments, directives, raw codes to be inlined, etc., can be added to the OpenARC IRs. An annotation can be associated with any *Annotatable* IR objects (e.g., OpenMP/OpenACC directives attached to a *ForLoop* statement) or can be stand-alone like a comment statement.

- *Flexible Printing.* The printing functions in each IR class type allow flexible rendering of the program representation, depending on the target languages and translation goals.

As a high-level representation, OpenARC represents program semantics in a language-independent way. The OpenARC class hierarchy, which is more general (abstract) than AST structure, serves as a base vehicle to convey program semantics. The base OpenARC classes can capture semantics common among traditional general-purpose languages, such as Fortran or C/C++, in a generic way. However, traditional general-purpose languages are inherently sequential, and thus the base OpenARC class hierarchy cannot capture other important properties such as concurrency, parallelism, data distribution, etc. Moreover, new parallel programming models such as CUDA and OpenCL introduce new languages features that do not exist in traditional general-purpose languages (e.g., a CUDA kernel function has a mechanism to express the execution configuration for the kernel call).

To address these issues, OpenARC provides two alternative solutions: 1) semantic annotation and 2) class hierarchy extension. The semantic annotation augments existing OpenARC objects with rich semantic information. OpenARC already supports various directives, including OpenMP, OpenACC, and several internal directives, with which OpenARC can capture both task and data parallelisms, data sharing rules, accelerator regions, synchronizations, etc.

The OpenARC class hierarchy extension offers an alternative method to realize richer semantics; it can easily embrace new language constructs. Emerging parallel programming languages, such as X10 and Chapel, support various language constructs to explicitly express data locality, data distributions, etc. Study on extending existing accelerator programming models with these new language constructs may be possible by creating a new IR class or by extending existing classes in OpenARC.

This extensible high-level class hierarchy, with its rich semantic annotation provisions, allows OpenARC to be used as a base framework and starting point for quickly prototyping and exploring the trade-offs in realizing productive programming environment for scalable heterogeneous computing.

2.2 Compiler Analyses

OpenARC exploits various advanced analysis techniques to efficiently port OpenACC applications to a target accelerator. However, some of these techniques cannot be directly applied to the OpenACC translation due to the unique semantics of the OpenACC execution model. This section discusses how to customize general automatic parallelization techniques to the OpenACC context.

2.2.1 Scalar and Array Privatization

Identifying privatizable variables, along with reduction recognition, is one of the most important enabling techniques for automatic parallelization. For this, Cetus provides a general array privatizer, which can detect both scalar and array privatizable variables in a loop nest [2]. If an OpenACC compute region is a perfectly-nested loop, the general privatizer can be directly used to detect OpenACC gang/-worker/vector private variables. However, if the region is not a perfectly nested loop, which may be common in OpenACC *parallel* regions, additional analyses are necessary to identify private variables as follows. 1) If a *parallel* region contains multiple *gang* loops, a variable can be put in the private clause of the region only if the variable is privatizable in all the outermost gang loops contained in the region. 2) The variable should not be *upward-exposed* (used before it is defined) at entry to its enclosing compute region. 3) Variables appearing in any data clauses of the enclosing compute region cannot be privatized unless they are explicitly included in OpenACC *private/firstprivate* clauses (shared by default). 4) Local variables declared in a gang/worker/private loop are private to the enclosing loop, and local variables declared inside a *parallel* region but outside of any work-sharing loop are gang-private by default. A privatization transformation pass should also consider the execution context and a target architecture. For example, the CUDA memory system has several special memories whose visibility and capacity are different; depending on the size and the scope of a private variable, different memory allocation strategies should be applied, which is an important performance tuning subject. For this, OpenARC provides several additional directives and environment variables to control this mapping.

```
#pragma acc parallel \
reduction(+gang_num)
{ int lnum = 0;
  #pragma acc loop gang
  for(i=0; i<8; i++) {
    ...
    lnum = lnum + 1;
  }
  gang_num += lnum;
} //gang_num will be 8.
```

(a) Before IVS

```
#pragma acc parallel \
reduction(+gang_num)
{ int lnum = 0;
  #pragma acc loop gang
  for(i=0; i<8; i++) {
    ...
  }
  lnum += 8;
  gang_num += lnum;
} //gang_num will be 64.
```

(b) After IVS

Figure 1: Induction Variable Substitution (IVS) Example, which shows IVS can be unsafe depending on the execution modes of OpenACC.

2.2.2 Induction Variable Substitution

The induction variable substitution algorithm is another parallelization-enabling technique, which recognizes an induction form (e.g., $iv = iv + expr$), similar to a reduction form, and converts into a closed form that does not induce data dependence. Cetus provides a powerful induction variable substitution pass that can detect generalized induction variables [2]. However, the traditional substitution algorithm may not be safe in certain OpenACC execution contexts. For example, induction variable substitution may not be applicable to a gang loop in a *parallel* region; in Fig. 1 (a), if the parallel region is executed by eight gangs, the output value of *gang_num* will be 8, since the iterations of the inner gang loop will be partitioned to each gang, and *lnum* is gang-private. However, in Fig. 1 (b), the output value of *gang_num* will be 64, since each gang will redundantly execute the substituted expression ($lnum+ = 8;$). This problem occurs because the code in an OpenACC *parallel* region is redundantly executed by a set of participating gangs (i.e., gang-redundant mode) unless work-sharing constructs are encountered. Therefore, the induction substitution algorithm should be applied selectively, depending on the execution modes of OpenACC.

2.2.3 Reduction Recognition and Transformation

Reduction operations are used in many scientific applications, and thus Cetus supports a general reduction variable analyzer that detects additive reduction patterns (e.g., $sum = sum + expr$) in a loop for both scalar and array variables [2]. OpenARC extends the base analyzer to adapt to execution modes in OpenACC; 1) assignment expression outside a loop can be a reduction operation if the expression is executed in a *gang-redundant* mode, meaning that each gang executes the same code redundantly. 2) If a reduction variable recognized in a work-sharing loop is private to the loop, reduction transformation should not be applied to the variable. 3) If an OpenACC *parallel* region is not a loop, additional checking is necessary to make sure that a gang reduction variable is used only for reduction operations within the region.

2.3 OpenARC Directives and Environment Variables

Table 1: Directives Supported by OpenARC

`#pragma acc accdirective [clause [,] clause]...]`
`#pragma omp ompdirective [clause [,] clause]...]`
`#pragma cetus [clause [,] clause]...]`
`#pragma acc internal [clause [,] clause]...]`

Table 1 shows the list of directives that OpenARC supports; both OpenACC and OpenMP directives are accepted. In OpenARC, however, OpenMP programs are not translated into low-level output codes, such as Pthreads codes; instead, information in the OpenMP directives are used during the OpenACC translation to preserve correct OpenMP multithreading semantics. In addition to these standard directives, OpenARC uses several internal directives; *cetus* annotations are used to convey general analysis outputs generated by various built-in Cetus passes, and *acc internal* directives are used by OpenARC passes to communicate various analysis outputs and configurations specific to the OpenACC translation. Table 2 shows a partial list of the *acc internal* clauses: *kernelConfPt(kernel)* is used by an analysis pass to direct where the translator puts configuration statements for a kernel; *gangconf(list)* is used to generate a multi-dimensional grid when porting to CUDA. These rich set of internal annotations, which can be easily extended to adopt any user-provided directives, are used by internal OpenARC passes to communicate with each other, enabling clear separation between analysis passes and transformation passes. This modular design allows convenient framework for researchers to integrate/debug new compiler passes.

OpenARC environment variables control the program-level behavior of various compiler optimizations, transformations, and execution/tuning configurations. Table 3 shows a partial list of supported OpenARC environment variables; *showInternalAnnotations* controls types of annotations to be included in the output file, varying from nothing to all annotations. This selective annotation printing, combined with

Table 2: A Partial List of OpenARC Internal Clauses

Clause	Description
accglobal(list)	contains global symbols
accexplicitshared (list)	contains symbols that user inserted in data clauses
accreadonly(list)	contains R/O shared symbols
kernelConfPt (kernel)	indicates where to put kernel-configuration statements
gangconf(list)	contains sizes of each gang loop in nested gang loops

other OpenARC-pass-control options (e.g., *AccAnalysisOnly*), provides a simple, but efficient way to trace how OpenARC translates an input OpenACC program to a target device. OpenARC also offers various environment variables to be used for efficient tuning; e.g., *UserDirectiveFile* allows users or external tuning engines to add directives through a separate file, rather than inserting to the source program directly. If *extractTuningParameters* option is on, OpenARC will generate a list of optimizations applicable to a given input program, which can be used to prune the optimization search space that a tuning system should navigate. *genTuningConfFiles* generates a set of tuning configurations files based on the information extracted from other tuning-related environment variables. Each tuning configuration file can be used to customize the overall OpenACC translation. Combined with other built-in tuning tools, these environment variables offer powerful building blocks, with which custom tuning system can be easily built.

Table 3: A Partial List of OpenARC Environment Variables

Parameter	Description
AccParallelization	detect parallelizable loops automatically
AccReduction	control reduction variable types (none/scalar/array) to recognize automatically
AccAnalysisOnly	run only selected analysis passes and exit
showInternalAnnotations	control types of annotations to be included in the output file
UserDirectiveFile	specify a file containing user directives
extractTuningParameters	generate a file containing tuning parameters applicable to a given input program
defaultTuningConfFile	specify a file containing default tuning configurations that control tuning configuration generation
genTuningConfFiles	generate tuning configuration files, each of which can be fed to OpenARC compiler

2.4 Overall Compilation Flow

The OpenARC compiler consists of the following major passes, each of which provides one or more checkpoints. Within checkpoints, intermediate results can be saved as output codes with annotations. This is useful for manual debugging or for implementing traceability mechanisms.

Cetus parser calls C preprocessor to handle header files, macro expansion, etc., and converts the preprocessed OpenACC program into internal representations (OpenARC IR).

Input preprocessor parses OpenACC directives and performs initial code transformations for later passes, including selective procedure cloning to enable context-sensitive, interprocedural analyses/transformations.

OpenACC loop-directive preprocessor interprets loop directives, extracts necessary implicit information from the loop constructs and stores them as internal/external annotations, and performs initial loop transformations according to explicit/implicit rules.

OpenACC analysis checks the correctness of the overall OpenACC directives and derives sharing rules for the data not explicitly specified by programmers.

User-directive handler interprets additional annotations provided as a separate file, and stores them into IR.

Optimization pass performs various optimizations such as privatization, reduction recognition, locality analysis, etc. All the optimization results are stored as annotations to inform later transformation passes.

Transformation pass conducts several pre-transformations according to the results passed from the optimization pass.

OpenACC-to-Accelerator translation generates output accelerator codes with post-transformations that are possible only at output codes.

3. EVALUATION

For evaluation, we use thirteen OpenACC programs from diverse application domains, which were manually ported from several OpenMP benchmarks (two NAS Parallel Benchmarks (*EP* and *CG*), three kernel benchmarks (*MATMUL*, *JACOBI* and *SPMUL*), and eight Rodinia Benchmarks [3] (*BACKPROP*, *BFS*, *CFD*, *HOTSPOT*, *KMEANS*, *LUD*, *NW*, *SRAD*)) [8]. The OpenACC programs were translated to output CUDA programs by OpenARC and then compiled using NVCC V5.0 and GCC V4.4.6 . The same OpenACC programs were also ported by the PGI-OpenACC compiler (V13.6). For the programs that don't have manual CUDA versions (*JACOBI*, *SPMUL*, *EP*, and *CG*), locally developed CUDA versions were used. The compiled programs were executed on a platform with Intel Xeon X5660 host CPUs and NVIDIA Tesla M2090 GPU.

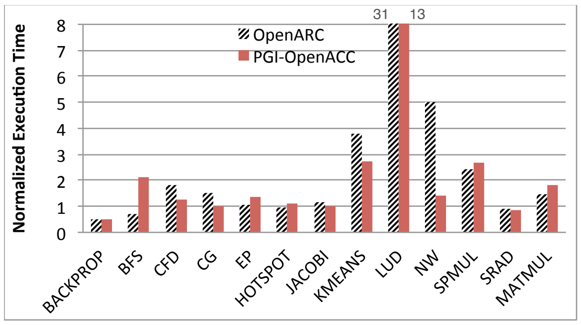

Figure 2: Performance of benchmarks translated by OpenARC and by the PGI-OpenACC compiler. The execution times are normalized to those of manual CUDA versions. Lower is better.

Fig. 2 presents the performance of the proposed OpenARC compiler; execution times are normalized to the manual CUDA versions, and thus a normalized value less than one indicates better performance than the manual CUDA versions. The results show that OpenARC performs similarly to the PGI-OpenACC compiler, while our compiler provides much better and extensible environment for program debugging and optimizations. For some benchmarks (e.g., BFS, EP, HOTSPOT, and SPMUL), OpenARC outperforms the PGI-OpenACC compiler, because OpenARC allows more fine-grained control over compiler-specific or GPU-specific features than the PGI-OpenACC compiler. However, the current implementation of OpenARC does not contain some compiler optimizations such as the automatic tiling transformation in the PGI-OpenACC compiler. Research and implementation of various analysis/transformation techniques to enable better utilization of GPUs will be future work.

The figure also indicates that OpenARC and PGI-OpenACC versions perform similarly to the manual CUDA version, except for *LUD*, where OpenARC and PGI-OpenACC perform significantly worse than the manual version. The excellent performance of the CUDA version is due to complex manual optimizations, which partition the whole matrix into many small blocks, and apply different software-caching strategies using complex thread-access patterns. These complex data access patterns to exploit the CUDA shared memory are not expressible in the standard OpenACC model, which motivates research on extending existing directive models to express more architecture-specific features still in high level.

4. CONCLUSIONS

Open Accelerator Research Compiler (OpenARC) is an open-source framework that supports the full feature set of OpenACC V1.0 and serves as a powerful research vehicle for various source-to-source transformation and instrumentation study to address important issues in directive-based, high-level programming models for scalable heterogeneous computing. In this paper, we have provided an overview of the overall design, implementation, and performance of OpenARC. Implementing a reference OpenACC compiler has led us to identify several key analysis techniques required for efficient porting of OpenACC applications. Porting thirteen OpenACC applications from diverse scientific domains shows that OpenARC performs similarly to a commercial compiler, and OpenACC can achieve performance comparable to that of low-level device programming models in many cases.

5. ACKNOWLEDGMENTS

The paper has been authored by Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract #DE-AC05-00OR22725 to the U.S. Government. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes. This research is sponsored by the Office of Advanced Scientific Computing Research in the U.S. Department of Energy.

6. REFERENCES

[1] OpenARC: Open Accelerator Research Compiler. [Online]. Available: http://ft.ornl.gov/research/openarc. (accessed April 11, 2014).

[2] H. Bae, L. Bachega, C. Dave, S.-I. Lee, S. Lee, et al. Cetus: A source-to-source compiler infrastructure for multicores. In *Proc. of the 14th Int'l Workshop on Compilers for Parallel Computing (CPC'09)*, page 14 pages, 2009.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[4] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60, 2011.

[5] T. D. Han and T. S. Abdelrahman. hiCUDA: High-level GPGPU programming. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):78–90, 2011.

[6] HMPP. OpenHMPP directive-based programming model for hybrid computing. [Online]. Available: http://www.caps-entreprise.com/openhmpp-directives/, 2009. (accessed April 11, 2014).

[7] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *SC'10: Proceedings of the 2010 ACM/IEEE conference on Supercomputing*. IEEE press, 2010.

[8] S. Lee and J. S. Vetter. Early evaluation of directive-based GPU programming models for productive Exascale computing. In *the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. IEEE press, 2012.

[9] A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 51–61. ACM, 2010.

[10] OpenACC. OpenACC: Directives for Accelerators. [Online]. Available: http://www.openacc-standard.org, 2011. (accessed April 11, 2014).

[11] R. Reyes, I. López-Rodríguez, J. Fumero, and F. Sande. accULL: An OpenACC implementation with CUDA and OpenCL support. In *Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 871–882. Springer Berlin Heidelberg, 2012.

[12] X. Tian, R. Xu, Y. Yan, Z. Yun, S. Chandrasekaran, and B. Chapman. Compiling a High-level Directive-Based Programming Model for GPGPUs. *Annual Workshop on Languages and Compilers for High Performance Computing (LCPC)*, 2013.

[13] J. S. Vetter, editor. *Contemporary High Performance Computing: From Petascale Toward Exascale*, volume 1 of *CRC Computational Science Series*. Taylor and Francis, Boca Raton, 1 edition, 2013.