Compiler development for the OpenACC programming model using the LLVM compiler infrastructure

Ανάπτυξη μεταγλωττιστή για το μοντέλο προγραμματισμού OpenACC με χρήση της υποδομής μεταγλώττισης LLVM

Εμμανουήλ Μαρούδας

emmanouil.maroudas@gmail.com

Πανεπιστήμιο Θεσσαλίας

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Βόλος, Οκτώβρης 2013

Author

Emmanouil Maroudas

Thesis Supervisors

Christos Antonopoulos,
Assistant Professor at University of Thessaly

Nikos Bellas,
Associate Professor at University of Thessaly

University of Thessaly
Department of Computer & Communication Engineering

Volos, October 2013

Table of Contents

List of Figures

List of Tables

List of Code Examples

Abstract

The evolution of General-Purpose Graphics Processing Units (GPGPUs) in conjunction with the increasing need for performance and low power consumption, gave birth to heterogeneous systems, parallel architectures and led to the creation of many parallel programming models like OpenCL [4], OpenMP [5], OpenACC [1], MPI [6], Unified Parallel C (UPC) [7], CUDA [22] and more. Their purpose is to offer the programmer a high level access to devices capable of parallel execution, efficient utilization of all parallel resources in a system, high portability and easier maintainability of source code.

This thesis presents a source to source transformation tool named "ACCLL" which translates a program from the relatively new OpenACC programming model [1] to the well established OpenCL Standard [4] using the LLVM [2] compiler infrastructure and mainly their native C/C++ compiler Clang [3].

ACCLL is an implementation of the OpenACC programming model [1] on top of the OpenCL Standard [4]. Its purpose is to give the programmer the ability to express the parallelism of their programs in a more abstract representation in the source code and allow access to all OpenCL advantages (offline compiler) and analysis tools at the same time.

Some examples of tools are Profilers (Intel® Vtune [12], Intel® Graphics Performance Analyzer [13], AMD APP Profiler [14]), formal analysis tools (GPUVerify [9]), Automatic Hardware Generation tools (SOpenCL [10]) and more.

The following pages describe the structure, flow and implementation decisions that have been taken during the development phase.

Περίληψη

Η εξέλιξη των μονάδων επεξεργασίας γραφικών γενικού σκοπού (GPGPUs) σε συνδυασμό με την ανάγκη για αυξανόμενη αποδοτικότητα και μειωμένη κατανάλωση ισχύος, έδωσαν το έναυσμα για την δημιουργία ετερογενών συστημάτων, παράλληλων αρχιτεκτονικών και μοντέλων παράλληλου υπολογισμού όπως το OpenCL [4], OpenMP [5], OpenACC [1], MPI [6], Unified Parallel C (UPC) [7], CUDA [22] και άλλα. Σκοπός τους είναι να δώσουν στον προγραμματιστή πρόσβαση στις δυνατότητες των νέων παράλληλων αρχιτεκτονικών με αφηρημένο και συνεπή τρόπο, να προσφέρουν αποτελεσματική αξιοποίηση των πόρων του εκάστοτε συστήματος, φορητότητα και ευκολότερη συντήρηση του πηγαίου κώδικα.

Στα πλαίσια της διπλωματικής μου εργασίας ανέπτυξα ένα εργαλείο μετατροπής πηγαίου κώδικα από το υψηλού επιπέδου, σχετικά νέο μοντέλο παράλληλου προγραμματισμού OpenACC [1] στο χαμηλότερου επιπέδου, καθιερωμένο μοντέλο OpenCL [4] χρησιμοποιώντας την υποδομή μεταγλωττιστών LLVM [2] και κυρίως τον Clang μεταγλωττιστή [3] ο οποίος είναι βασισμένος σε αυτήν την υποδομή. Αυτό το εργαλείο (accll) είναι μια υλοποίηση του OpenACC μοντέλου προγραμματισμού [1]η οποία πατάει επάνω στο μοντέλο OpenCL [4]. Σκοπός του είναι μετατρέπει μια υψηλού επιπέδου παράλληλη εφαρμογή γραμμένη σε OpenACC [1], σε μία αντίστοιχης λειτουργικότητας εφαρμογή γραμμένη στο χαμηλότερου επιπέδου OpenCL μοντέλο [4].

Με αυτόν τον τρόπο, ο προγραμματιστής αποκτά πρόσβαση σε όλα τα εργαλεία ανάλυσης τα οποία έχουν σχεδιαστεί για το ήδη υπάρχων μοντέλο OpenCL [4] όπως Profilers (Intel® Vtune [12], Intel® Graphics Performance Analyzer [13], AMD APP Profiler [14]), εργαλεία ανάλυσης εγκυρότητας (GPUVerify [9]), εργαλεία αυτόματης δημιουργίας υλικού (SOpenCL [10]) και πολλά άλλα.

Οι παρακάτω σελίδες παρουσιάζουν την δομή, ροή και τις σχεδιαστικές αποφάσεις που πάρθηκαν κατά την ανάπτυξη αυτού του εργαλείου.

Chapter 1

Introduction

Over the last years the computational needs of many scientific applications, rapidly grown databases and the Entertainment Industry, combined with some technology constraints such as power consumption issues, transistor size and cooling limitations, motivated computer scientists on the quest for solutions which offer increased performance. We walk the path of parallel architectures like multicore processors, vector instruction sets (SIMD), GPGPU's and system configurations with different types of computational units also known as Heterogeneous Systems [11]. A computational unit could be a general purpose processor, a graphics processing unit (GPU), a digital signal processor (DSP), a co-processor or even a ASIC or FPGA. All these types of processors can be installed in a system today and each one may have different instruction set architecture (ISA).

One great challenge the parallel architectures and heterogeneous systems [11] introduce, is the ability for the programmer to write portable programs capable of exploiting all the computational resources present to any particular system they run on. Many Parallel programming models and frameworks have been designed to fill the void from the classic sequential computing to the new parallel computing era.

OpenACC [1] is a relatively new programming model that came out on November 2011. It's design purpose is to simplify parallel programming on heterogeneous systems [11] compared to other programming models like OpenCL [4] which can be considered quite verbose. An advantage of OpenACC regarding the OpenCL language is that with the former it is not necessary to alter the sequential version of the source code too much beyond importing some compiler directives in the appropriate places. If the compiler supports them it generates code that can be run on any supported device, if not, they are being ignored and the compiled program will run sequentially. This is not achievable with OpenCL, in which as a rule, the parallel version of a program differs a lot from the sequential.

This thesis presents a new tool named "ACCLL" that translates a OpenACC program to its equivalent OpenCL program at source code level with the same functionality, using the LLVM [2] compiler infrastructure and mainly their native C/C++ compiler front end, Clang [3]. Its purpose is to give the programmer the ability to express the parallelism of their programs in a more abstract representation in the source code and allow access to all OpenCL advantages (e.g. offline compiler) and analysis tools at the same time, like Profilers (Intel® Vtune [12], Intel® Graphics Performance Analyzer [13], AMD APP Profiler [14]), formal analysis tools (GPUVerify [9]), Automatic Hardware Generation tools (SOpenCL [10]) and more.

Having such a tool available, also gives the programmer the opportunity to familiarize themselves with all the benefits the modern parallel architectures can offer, using a more abstract parallel programming model with smoother learning curve. Another interesting use case of this tool can be to evaluate, test and strengthen the new OpenACC programming model on top of a robust, well defined and popular parallel framework such as OpenCL. As time passes and new parallel programming models appear, these type of tools may become essential in the design process and improvement of the aforementioned programming models.

The following chapters provide some background information, some implementation decisions taken during the development, the structure of the presented tool “ACCLL”, among with code examples for the most common source code patterns in a OpenACC program.

Chapter 2 provides some brief background information about the discrete technologies and tools this project puts together. These are the OpenCL and OpenACC programming models and the LLVM [2]/Clang [3] compiler infrastructure.

Chapter 3 elaborates in depth the internal changes to Clang [3], some key design decisions with more technical details. It contains a presentation of the newly added classes and their usefulness, changes to the Parser and the semantic checking phases of a OpenACC directive.

Chapter 4 explains the mapping between the two individual program models (OpenCL, OpenACC) and some limitations of the selected approach.

Chapter 5 is an introduction to the main tool (accll), giving the general design structure of the whole project and the reasons behind the selection of LLVM/Clang infrastructure among other choices. It also covers the individual Stages from the original source code through its final form, the OpenACC Runtime support and the ACCLL Runtime library, discussing the decisions made during development and their importance. In the end of the chapter there are details about the asynchronous execution model, its differences with the synchronous execution model and the way these two are expressed via the OpenCL API.

For each Stage there is a detailed presentation about its actions and changes it induces among any technical problems and design decisions taken during the development phase. The most critical and ‘hard working’

Stages are the one that implements the data movements and the one that creates the OpenCL kernels from the OpenACC Compute Constructs.

Finally, chapter 6 provides evaluation and testing information gathered from a collection of test files of various online benchmarks. It also points some outstanding testing issues and the implementation’s decisions on each one of them.

In the end there is also a complete example of a simple vector addition program in the original OpenACC form and the final OpenCL form this tool produces.

Chapter 2

Background

As the time of writing, the latest versions of these two programming interfaces are OpenACC 2.0 [1] and OpenCL 2.0 [4]. The versions described in this document are the OpenACC 1.0 [1] and OpenCL 1.2 [4] except if explicitly stated otherwise.

2.1 The OpenCL Standard

One of the most embraced technologies in the industry is the OpenCL Standard [4] which provides a clean and consistent API, Framework and Libraries for developing parallel programs. It gives the programmer full control over the device’s resources. A key feature of the OpenCL Standard [4] is that the device code can be compiled at Runtime, providing portability and flexibility among systems with different compute devices.

2.1.1 Execution Model

An OpenCL device consists of one or more compute units each one of one or more processing elements. The code executes within the processing elements.

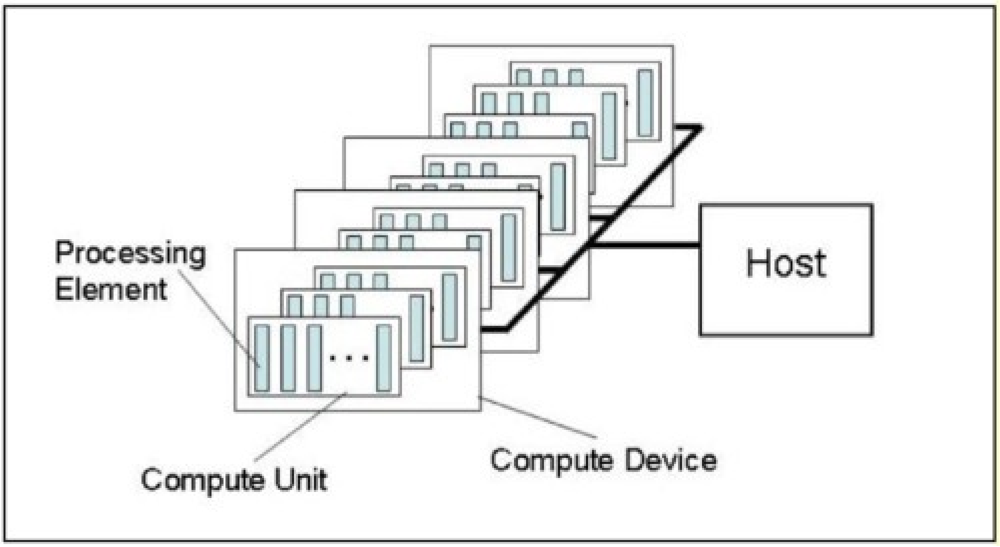

Figure 1: OpenCL Platform Model

Source: The OpenCL Specification - Khronos Group

An OpenCL application involves a host program and one or more kernels containing the computations. The host program submits commands to the device which executes them on the processing elements.

An OpenCL kernel is the core computational representation of the parallel program. Typically it expresses the parallelism at the finest granularity possible, consuming a single unit of input data and producing a single unit of output data.

Each kernel execution has an index space (NDRange) where an instance of the computation code runs for each work-item in this index space. Every work-item operates on different data using its unique index information (SIMD execution) like global id and local id. Work-items can be organized into work-groups by the user to provide different scales of granularity in a flexible way.

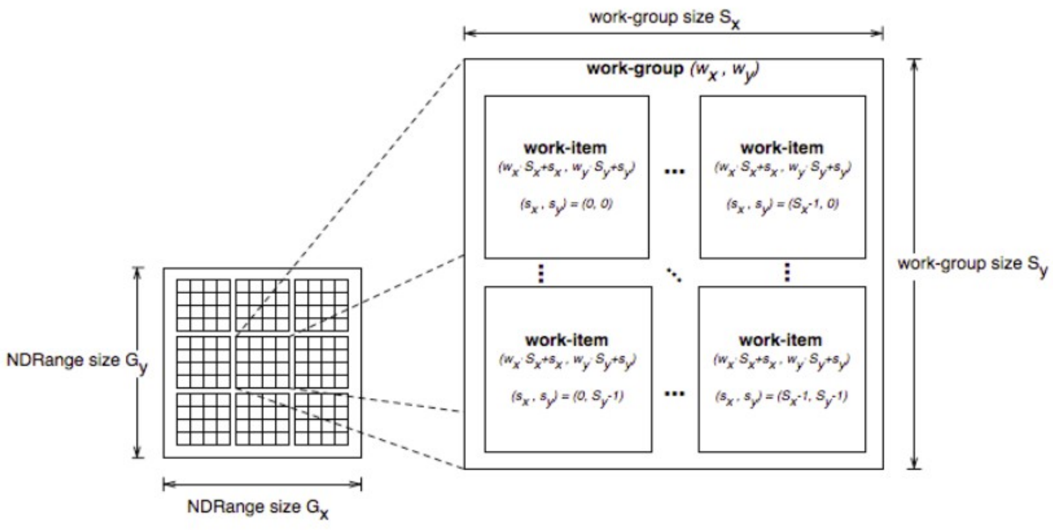
work-group size S_x
work-group (w_x, w_y)
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (0, 0)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (S_x-1, 0)$
work-group size S_y
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (0, S_y-1)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (S_x-1, S_y-1)$
NDRange size G_y
NDRange size G_x

Figure 2: An example of NDRange index

Source: The OpenCL Specification - Khronos Group

The host program creates a command-queue to coordinate execution of the kernels and data transfers on the devices in two ways:

- In-order Execution
  Commands are launched in the order they appear in the command-queue and complete in order.
- Out-of-order Execution
  Commands are issued in order, but do not wait to complete before following commands execute.

2.1.2 Memory Model

There are four distinct memory regions in a OpenCL device:

- Global Memory (big, slow):
  This memory region permits read/write access to all work-items in all work-groups.
- Constant Memory (cached):
  A region of global memory that remains constant during the execution of a kernel. The host allocates and initializes memory objects placed into constant memory.
- Local Memory (small, fast):
  A memory region local to a work-group. This memory region can be used to allocate variables that are shared by all work-items in that work-group.
- Private Memory:
  A region of memory private to a work-item. Variables defined in one work-item's private memory are not visible to another work-item.

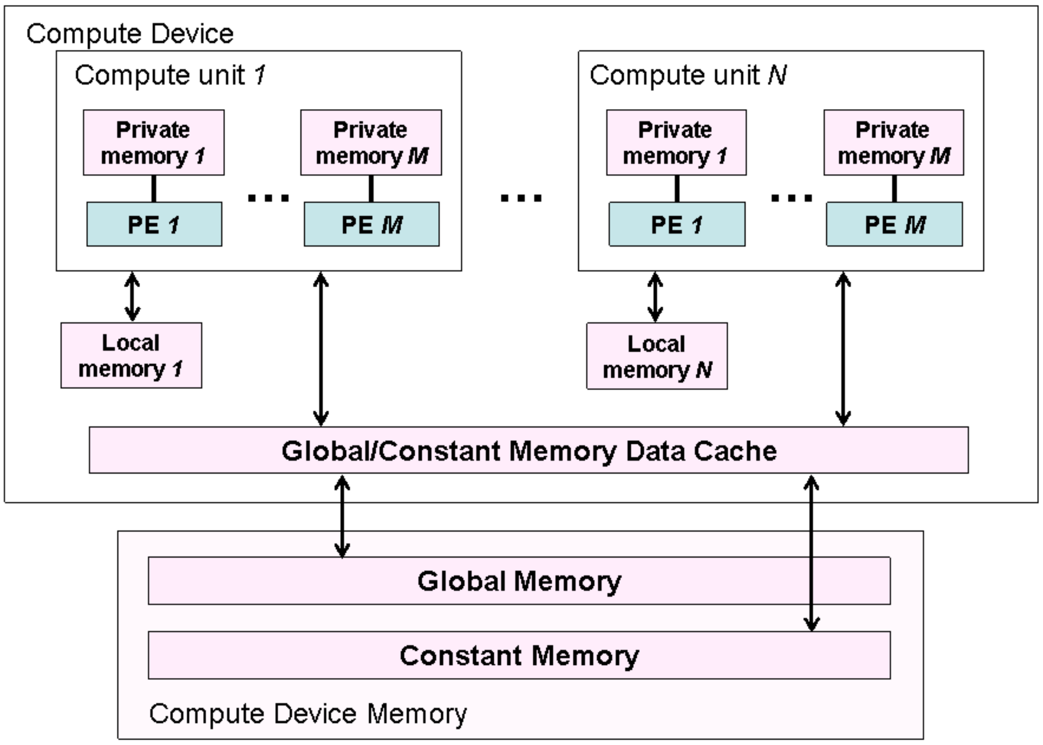

Figure 3: OpenCL Memory Model

Source: The OpenCL Specification - Khronos Group

2.1.2.1 Synchronization

There are 3 means of synchronization in OpenCL:

1. Command-queue barrier
   Ensures that all previously queued commands have finished execution and any resulting updates to memory objects are visible to subsequently enqueued commands before they begin execution.
2. Waiting on an event
   All OpenCL API functions that enqueue commands return an event that identifies the command and memory objects it updates. A subsequent command waiting on that event is guaranteed that updates to those memory objects are visible before the command begins execution.
3. Barrier synchronization in work-items in a single work-group
   For the purposes of this document, this case can be ignored. The reason is that there is no way to express this type of synchronization using the OpenACC programming model. The workaround in this case is to break the Compute Region at the synchronization points creating two or more Compute Regions (see Table 1). This works because there is an implicit barrier by default between Compute Regions.

For example, the following codes are algorithmically equivalent although performance may be different:

OpenCL kernel	OpenACC kernels loop Construct
__kernel foo() {	#pragma kernels loop
//do this	{
//sync for	//do this
work-items	}
//do that	//implicit barrier at host code
}	#pragma kernels loop
	{
	//do that
	}

Table 1: Work-item synchronization equivalent in OpenACC

2.1.2.2 Memory Objects

Kernels take memory objects as parameters which can be either input or output or both. OpenCL memory objects can be either buffer objects or image objects. Their main differences is that a simple buffer object contains its elements in a one-dimension collection whereas an image object can contain elements organized in a two- or three- dimensional collection. Also elements in an image object follow an internal storage format and reads/writes are performed with specific built-in functions and not directly as in the case of a simple buffer object. For the needs of this document, image objects can be ignored. The reason is that there is no way to express this type of object using the OpenACC programming model.

A buffer object stores a one-dimensional collection of elements. Elements of a buffer object may have scalar, vector, or a user-defined data type. The minimum number of elements in a memory object is one.

2.2 The OpenACC Programming Model

OpenACC [1] is a programming standard for parallel computing developed by Cray [18], CAPS [19], Nvidia [20] and PGI [21]. The standard is designed to simplify parallel programming of heterogeneous CPU/GPU systems [11].

The OpenACC API describes a collection of compiler directives and runtime routines to specify loops and regions of code in standard C, C++ and FORTRAN to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPU's and accelerators.

Programmers can create high-level host+accelerator programs without the need to explicitly initialize the accelerator, manually allocate/free device memory, offload programs to the accelerator, or initiate accelerator startup and shutdown. All these details are implicitly integrated into the programming model letting the compilers and runtimes manage all the hard work. The programming model allows the programmer to provide performance related details to the compiler, like data locality information, loop mapping strategies, and more.

2.2.1 Execution Model

The OpenACC execution model is quite similar to the OpenCL execution model. There is a host program managing the attached devices which offloads code regions specified by the user to the currently connected device. The device executes parallel regions and kernel regions which both may contain loop regions. Each loop region nested in a kernels region is executed as a distinct kernel. On the other hand the parallel region defines a single kernel as it is, even if it contains nested loop regions.

The host program is responsible for memory allocation/deallocation on the device, data initialization and transfer, sending the computation to the device, passing arguments to the parallel region, queuing the device code, synchronization and transferring results back to the host.

In the hardware level, most current accelerators support two or three levels of parallelism. There may be limited support for synchronization across coarse-grain parallel operations. Some accelerators also support fine-grain parallelism, often implemented as multiple threads of execution within a single execution unit. Finally, most accelerators support SIMD or vector operations within each execution unit.

2.2.2 Memory Model

The accelerator usually has its own completely separate address space from the host. In this case, the host may not be able to access device memory directly and all data movement between host memory and device memory must be performed by the host through runtime library calls.

The concept of separate host and accelerator memories is visible in OpenCL, where one has to declare one host and one device buffer before any data movement between the two different address spaces. These data movements often overshadow user code. The OpenACC programming model, implicitly performs all these data movements instead, without the need of the second explicit declaration for the device buffer.

2.2.3 Directive Format

An OpenACC directive applies to the immediately following statement, structured block or loop. More Information about clauses for each directive exists in the OpenACC Specification [1]. In C and C++, OpenACC directives are specified with the #pragma mechanism. The general syntax of an OpenACC directive is:

#pragma acc directive-name [clause [[,] clause]...] new-line

Table 2 has a brief presentation of the supported directives among their specific syntax.

Directive Type	C/C++ Syntax
Compute Constructs	#pragma acc parallel [clause [[,] clause]...] new-line structured block
Compute Constructs	#pragma acc kernels [clause [[,] clause]...] new-line structured block
Data Construct	#pragma acc data [clause [[,] clause]...] new-line structured block
Host_Data Construct	#pragma acc host_data [clause [[,] clause]...] new-line structured block
Loop Construct	#pragma acc loop [clause [[,] clause]...]new-line for loop
Cache	#pragma acc cache(list) new-line
Combined	#pragma acc parallel loop [clause [[,] clause]...]new-line for loop
Combined	#pragma acc kernels loop [clause [[,] clause]...]new-line for loop
Declare	#pragma acc declare declclause [[,] declclause]... new-line
Update	#pragma acc update clause [[,] clause]... new-line
Wait	#pragma acc wait [(scalar-integer-expression)] new-line

Table 2: OpenACC directives and syntax

2.2.4 Conditional Compilation

The _OPENACC macro name is defined to have a unique value for each version of the OpenACC Standard.

2.2.5 Internal Control Variables

An OpenACC implementation defines two internal control variables (ICVs) that control the behavior of the program. These ICVs are initialized by the implementation, or through environment variables or OpenACC API routines. The ICVs are:

- acc-device-type-var : controls which type of accelerator device is used.
- acc-device-num-var : controls which accelerator device of the selected type is used.

2.2.6 Subarray Support

In order to minimize the amount of data transfers between host and device, the programmer can specify a subarray of an array to be transferred from/to a device, minimizing the usage of memory bandwidth. This results to less bottlenecks and performance loss.

In C and C++, a subarray has the following syntax

arr[start:length]

2.3 The LLVM Infrastructure

The LLVM Project [2] is a collection of modular and reusable compiler and tool-chain technologies. It is designed to support transparent, lifelong program analysis and transformation for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time, and in idle time between runs.

LLVM defines a common, low-level intermediate code representation (IR) in Static Single Assignment (SSA) form [16] and the LLVM compiler framework together provide a combination of key capabilities that are important for practical, lifelong analysis and transformation of programs.

2.4 Clang Compiler - The LLVM Front end for C/C++

The Clang [3] Compiler is a compiler front end for the C, C++, Objective-C and Objective-C++ programming languages. It uses LLVM [2] as its back end and its goal is to offer a replacement to the GNU Compiler Collection (GCC) [15]. Development is completely open-source, with several major software development companies involved, including Google and Apple. Clang builds on the LLVM optimizer and code generator, allowing it to provide high-quality optimization and code generation support for many targets.

2.4.1 Abstract Syntax Tree (AST)

Clang’s AST [8] closely resembles both the written C++ code and the C++ standard. For example, parenthesis expressions and compile time constants are available in an unreduced form in the AST. This makes Clang’s AST a good fit for refactoring tools.

2.4.2 AST Context & AST Nodes

All the necessary information about the AST for a translation unit is bundled up in the ASTContext class [8]. Clang’s AST [8] nodes are modeled on a class hierarchy that does not have a common ancestor. Instead, there are multiple larger hierarchies for basic node types like Decl and Stmt.

The two most basic nodes in the Clang AST [8] are statements (Stmt [8]) and declarations (Decl [8]). Expressions (Expr [8]) are also statements in Clang’s AST [8]. This design allows recursive traversal of the whole translation unit using the RecursiveASTVisitor class [8]. Thus, to traverse the full AST, one starts from the TranslationUnitDecl [8] and then recursively traverses everything that can be reached from that node.

2.4.3 RecursiveASTVisitor

A class that does preorder depth-first traversal on the entire Clang AST and visits each node.

This class performs three distinct tasks:

- TraverseNode(Node *x)
  Traverse the AST (i.e. go to each node). It is the entry point for traversing an AST rooted at x.
- WalkUpFromNode(Node *x)
  Walk up the class hierarchy, starting from the node's dynamic type, until the top-most class (e.g. Stmt [8] ,Decl [8], or Type [8]) is reached.
- VisitNode(Node *x)
  given a (node, class) combination, where 'class' is some base class of the dynamic type of 'node', call a user-overridable function to actually visit the node.

2.4.4 LibTooling Library & Replacements Mechanism

Clang Tooling [8] provides several infrastructures to write tools that are interested in syntactic and semantic information about a program. LibTooling [8] is one of them, it is a library to support writing standalone tools based on Clang [3] giving full access to the AST Tree. This source to source transformation tool is based on LibTooling [8].

One of the core refactoring utilities is the Replacement class [8] defined for the Refactoring framework [8] of Clang. Replacement represents an independent replacement of a range of text in a specific file. Its constructor takes as parameters a range of text to be replaced and a string with the new text which is going to replace the old one.

Chapter 3

ACCLL - OpenACC to OpenCL Transformation Tool

Considering the the purpose and needs which led to the creation of such a tool, as long as the features it is going to offer, the choice of the LLVM [2] & Clang [3] Infrastructure among other options was made for the following outstanding features:

First of all its modular design and clean codebase offers the ability to choose specific features and extend them independently. Second, being an open source project with an active and growing community provides a satisfying amount of internal documentation and feedback. It also produces code that meets quality/production standards using state of the art code transformation techniques.

The OpenACC Specification [1] defines directives for both C/C++ and FORTRAN languages. The accll tool aims only at C language support. C++ support is very experimental and not tested.

The development of accll is divided in two major components. One consists of all the necessary internal changes to Clang in order to represent information related with the OpenACC Constructs (support component), and the other consists of the transformation logic that produces the final code (the actual tool component).

First of all it was necessary to make Clang aware of these new OpenACC directives by applying some internal changes mainly at the parsing phase to gather all this new information. This was achieved by extending the Clang AST [8] with new classes to represent all the needed information about any OpenACC directive, clause and argument in the original source code.

The Clang Preprocessor and Parser were extended with new methods that recognize the start of an OpenACC directive and parse it until they reach the End of Directive token (EOD). New Diagnostic messages were added to report possible warnings or errors to the programmer due to syntax errors.

The Clang's Sema class [8], responsible for all semantic checks on syntactically correct code, was extended too to support OpenACC directives. New Diagnostic messages were added to report possible warnings or errors to the programmer due to semantic errors. If there are errors in the source code, the appropriate Diagnostics are printed and accll exits with an error code.

The Clang AST was extended with a new kind of Statement, the AccStmt to represent all the new OpenACC structures. After the semantic checking, if the code is error free, the collected OpenACC information goes into the AST making all the OpenACC structures reachable from any external tool which traverses the AST using the RecursiveASTVisitor.

The second component is the actual implementation of this external tool (accll). After a series of passes for each input file, the tool splits the initial code annotated with OpenACC directives into two new source files. One that contains the OpenCL [4] host program (creation of platform, context, queue, data transfers, etc) and one that contains the OpenCL device code (OpenCL kernels).

For example:

OpenACC Original Input Filename	foo.c
OpenCL host program Filename	foo_accll.c
OpenCL device code Filename	foo_accll.cl

Table 3: Filenames

Support for multiple input files is still in experimental state.

Some important details worth mentioning at this point are:

- The original input file is not altered by the transformation process.
- If there are no OpenACC Compute Constructs in a specific input file, only the OpenCL host code file is being created.
- If there are no OpenACC directives and no OpenACC Runtime calls at all in a specific input file, this file is being ignored by ACCLL.

More details and in depth explanation of the behavior of the tool comes in the following chapters.

One high priority goal during development was to keep the Clang internal changes as limited as possible and put most of the work in the external tool.

Some additional concerns were the reuse of already existing features of LLVM and Clang as much as possible, like LLVM data containers, the LLVM Runtime Type Information system (RTTI), code style and feedback to the community.

3.1 Clang Internal Changes

This chapter describes the internal changes to Clang in more detail. One of the goals of these changes was not to alter the expected behavior of Clang when there are no OpenACC directives in the source code.

3.2 Driver & Front end Support

The modified Clang ignores by default any OpenACC directives to maintain compatibility with the mainstream ‘vanilla’ version. A new command line option flag “-fopenacc” has been added to explicitly enable the parsing of OpenACC directives. If this flag is not present in the Clang invocation, the OpenACC directives are being ignored.

The OpenACC Specification [1], in section 2.2 Conditional Compilation, demands the definition of the macro _OPENACC with the value 201111. If the “-fopenacc” option flag is present, the Driver defines this macro to the Preprocessor.

3.3 Representation of Directives

3.3.1 Class DirectiveInfo

This class keeps information about the kind of the directive, its name, clauses or arguments (if any), the associated AccStmt with this directive and the parent directive (if it is nested).

There are also helper methods to determine what sort of directive is (Compute, Combined, Executable, Data), whether it has optional clauses or not, what types of clauses are valid for this directive and a couple other more which make easier the identification later in the transformation process.

For example the method *hasOptionalClauses()* returns true if the specific directive can be used without any clauses, else returns false.

3.3.2 Class ClauseInfo

This class keeps information about the kind of the clause, its name, arguments (if any), the associated directive with this clause and whether it is implementation defined (implicit or explicit by the user).

There are also helper methods to determine whether this is a data or private clause, if it takes optional arguments and their number (none, one, list of arguments).

3.3.3 Class Arg and Derived Classes

This class is the base class for the different kinds of arguments a clause or directive can have. Each argument kind corresponds to a derived class as shown by the following table:

Argument kind	Derived Class Name
Raw Expression	RawExprArg
Variable Reference	VarArg
Array Reference	ArrayArg
Array Subscript Expression	ArrayElementArg
Subarray Expression	SubarrayArg
Constant Integer Value	ConstArg

Table 4: OpenACC Argument Subclasses

The OpenACC Specification does not dictate such a separation. The reason behind this decision was the simplification of development and extensibility while improving readability and avoiding code duplication.

These classes keep information about the original expression in the source code, the nesting of the argument, the parent type clause or directive, type specific information and whether this is an implementation default argument or not (implicit or user defined).

Probably the most important method is the Matches() method with the following signature:

*bool Arg::Matches(Arg *Target);*

This method checks whether the calling argument is semantically equal to the Target argument passed as parameter, for example if both refer to the same variable, array, struct data member or constant value. A current limitation of this method is that It does not perform alias analysis over pointers and data between the calling argument and the Target.

3.3.4 Class AccStmt

This class represents an OpenACC directive existent in the source code, as an AST Node in the Clang's AST. It keeps information about the directive kind and its sub-statement if that exists. The kind of sub-statement conforms to the OpenACC Specification.

3.4 Parser Support

In principle any kind of compiler directive (pragma) is not part of the language. They merely specify a compiler's behavior for the given input. Typically the Preprocessor takes care of these pragmas and Clang takes this approach. The changes in Clang Parser among some minor tweaks in the Preprocessor, give Clang the desired ability to recognize OpenACC directives, clauses and arguments in places specified by the OpenACC Specification.

Because OpenACC directives and clauses contain arguments which are valid C/C++ expressions, it was inevitable to move most of the OpenACC directive handling from the Preprocessor to the Parser. This was almost a one way decision. The alternative approach was to bloat the Preprocessor's code base with duplicated code from the Parser which was non intuitive and contradictory to the general concept of keeping the internal changes as less as possible.

Therefore, the new pragma handler for OpenACC directives is implemented at two stages.

The first (Preprocessor) stage watches the input token stream for the beginning of a new OpenACC directive by the '#pragma acc' string sequence. Then It generates a new annotation token of kind tok::annot_pragma_acc and inserts it into the token stream of the current translation unit (the Lexer considers each input file as a stream of tokens). This way the Preprocessor is bypassed and the Parser can take control over the directive, recognizing any valid C/C++ expression as normal.

Next, the second (Parser) stage reads this newly added token from the input token stream. From now on the Parser takes action until the End Of Directive (EOD) token. The Parser is also responsible for cleaning any pending tokens in case of error. This is normally a work for the Preprocessor, but this is a special case as explained earlier.

Note that although this design is far from ideal it still is the less intrusive strategy feasible at the moment, given the current design structure of Clang's Preprocessor and Parser. As this time of writing, there are some discussions in the cfe-dev mailing list [17] about this issue but a proper solution implies a big refactor/rewriting of core components of the Parser and Preprocessor code base.

New Parser methods consume any kind of OpenACC directive or clause. The argument parsing utilizes already implemented methods such as ParseExpression() [8].

Some changes extend the Parser in order to understand the subarray syntax as specified by the OpenACC Standard.

The Clang front end to LLVM aims to keep as much Source Location information as possible by not altering or ignoring any explicit user written code (such as obsolete parentheses or casts). The idea behind this behavior is to make Clang suitable for source transformation tools like the one it is described here, and to shift code modification from the front end to the LLVM IR level. By this structure it is straightforward that all optimizations take place at the LLVM IR level. The new changes honor this philosophy by keeping as much Source Location information as needed for a decent refactoring from the external tool.

During the parsing phase, syntactic checks look through the user's source code and the Diagnostics mechanism [8] emits the appropriate warnings/errors if necessary.

3.5 Semantic Checking Support

Clang performs all the semantic checking routines from the Sema class [8]. Rather than 'polluting' this class with new methods and data structures necessary to perform OpenACC semantic checking, a new helper class 'ACCInfo' has been defined to act as a wrapper to Sema. This design decision allows any semantic checks relative to OpenACC to be part of the external tool rather than hide in the Clang's internals [8]. Such a move is still under consideration; while it will leverage the ability to modify the semantic checks, it will be harder to take advantage of the powerful internal Diagnostics mechanism.

The OpenACC directives appear in the source code before the statement they annotate, therefore after parsing a directive, the ACCInfo wrapper remembers it until the next statement's parsing finishes. This is done with the member pointer

*DirectiveInfo *PendingDirective;*

As the name indicates, this pointer remembers the last semantically correct and unconsumed directive the Parser discovered.

After parsing the next statement, a semantic check takes place to confirm whether the PendingDirective (if exists) can annotate it. If there is a conflict, the Diagnostics mechanism prints an error and the ACCInfo wrapper discards the PendingDirective.

The update, wait, declare, cache directives do not apply to any statements. In Particular the Executable directives (update, wait) and cache directive are treated as standalone statements. The declare directive provides a compact way to express data movements; it really annotates variable declarations.

The useful statements OpenACC directives can annotate are the Compound statement (CompoundStmt [8]) and the For statement (ForStmt [8]). The OpenACC Specification allows the nesting of OpenACC Constructs with some rules and restrictions. For example the nesting of Data Constructs is allowed but Compute Constructs nesting is disallowed.

A container structure called ‘RegionStack’ with LIFO semantics keeps track of the currently open Constructs at any location in the source code. Each time the AccInfo wrapper detects an opening of a Construct, it performs a validity check before this Construct enters the RegionStack. Notice that even if there are syntax or semantic violations in a clause or argument, the associated directive still gets inserted into the RegionStack to avoid false positive Diagnostics later in the annotated code. Each time a Construct reaches its end, the respective entry gets removed from the RegionStack.

The OpenACC Specification compels the implementation to give implicit values to some clauses (num_gangs, num_workers, vector_length) if the user did not. These values are not constrained by the Specification; the implementation can choose any valid value. The semantic check takes care of this issue. These clauses act more like an advice to compiler. The compiler can use different values depending on the analysis results.

Currently the default value for each one of them is '1'. There is also a heuristic method which tries to find more suitable values but it is still in experimental state. There are some issues with this approach in this version of the tool that are discussed with more details on chapter 7.

There is also a check for duplicate arguments inside data clauses. This is mostly a sanity check not mandatory by the OpenACC Standard, but very useful as it exposes ill defined code and informs the programmer with an error.

Chapter 4

Map OpenACC Constructs to OpenCL Structures & API Calls

Before any more implementation details, it is vital to describe the relations between the execution model, memory model and data structures between the two programming interfaces, OpenACC [1] and OpenCL [4] respectively.

The Execution Model of OpenACC dictates a In-Order Execution Model of device code, with implicit barriers by default. The device code regions execute in the same order they appear in the source code. This behavior in OpenCL corresponds to an In Order Command Queue.

Mainly for performance enhancement reasons there is support for asynchronous execution with the use of the async clause. The device in this case executes the device code regions out of order in general, but still in a relative order depending on the async clause's argument. Asynchronous device code regions with matching async arguments execute in the same order as they appear in the source code. This behavior in OpenCL corresponds to an Out of Order Command Queue using Barriers and Markers for synchronization. This is the approach this tool follows. Any synchronous device code can be expressed with this approach by issuing explicit Barriers to the Out of Order Command Queue after each kernel or data move command.

The gangs and workers in OpenACC correspond to one dimensional index space of global and local workers respectively. The biggest difference between the two execution models is that in OpenCL multi dimensional index spaces can be expressed, instead of only one dimensional index space in OpenACC. The compiler may recognize patterns of nested loop regions as multi dimensional index spaces but this is still an experimental feature.

The Memory Model of both programming models is quite similar. OpenCL offers more flexibility as a low level API. OpenACC does not define any new data structures, therefore there is not a unique data type mapping between the two memory models.

Features like Memory Mapping or Constant Memory in OpenCL do not have an OpenACC dual, however It is up the implementation's decision whether to exploit these features or not depending on performance improvements.

Apart from that, any resource allocations, deallocations and data transfers between host and device are done in the same spirit.

The implementation maps each host memory region (variable, array) to be offloaded to the associated device to an OpenCL Memory Object of type cl_mem. Using this object any data transfers take place by the appropriate OpenCL API calls. When the cl_mem object is not needed any more it is released.

Arguments of async clauses, either explicit or implicit (implementation defined), map to a cl_event OpenCL object. Later on, this object can be passed as parameter to a clEnqueueBarrierWithWaitList() OpenCL API call, which is a map for the wait directive.

In some cases it is more beneficial to use a Marker to group multiple asynchronous OpenCL Commands, each one with its own cl_event, to a single common cl_event. This simplifies code generation and provides better performance by reducing the number of Barriers needed to be inserted between execution and data transfer Commands.

Consider for example an asynchronous update directive which updates more than one memory regions. A later wait directive associated with this asynchronous update directive has to wait only for the cl_event of the Marker instead of each cl_event from the update directive separately.

Table 5 below summarizes these mappings.

OpenACC	OpenCL
Memory buffer (variable)	cm_mem object

Update directive	ClEnqueueReadBuffer()
	clEnqueueWriteBuffer()

Async clause	cl_event
	clEnqueueBarrierWithWaitList()
	clEnqueueMarkerWithWaitList()

Gangs, workers	NDRange

Table 5: Mapping from OpenACC to OpenCL

Chapter 5

External Transformation Tool

The external transformation tool (accll) utilizes Clang's LibTooling library [8]. It takes as parameter one or more input files of C source code (*.c) with OpenACC directives, and outputs two files. One that contains the device code (OpenCL kernels) and one that contains the host program (OpenCL API calls, instrumentation of execution).

The OpenCL development header files must be accessible from a standard system include directory or in a directory provided by the -I option. The tool neither emits binary code (object files) nor performs any linking, therefore beyond the development header files, an OpenCL Runtime [4] is not a requirement for the execution of this tool.

The tool sets up the command line options that will finally invoke the code transformation phases. These options consist of any input parameters the programmer explicitly requested and some implicit options like the "-fopenacc" option for Clang.

Depending on the transformation Stage the tool enables/disables all Clang Diagnostics [8] over the input files. Diagnostics are enabled by default at least for the first and last Stage to indicate any warnings/errors before the code transformation starts and after when it completes. In some intermediate Stages where the transformation is in progress and the final code is incomplete, the default behavior (Diagnostics enabled) floods the output with false warnings, therefore these Stages disable the Diagnostics mechanism using the extra "-w"command line argument. Each Stage can run either on 'full Warnings' or 'no Warnings' mode.

5.1 Transformation Stages

The tool consists of several Stages before it completes the transformation. Each Stage focuses on a specific transformation task, writes its changes to the output file and finishes (Figure 4). In case of error the program terminates with an error message indicating the Stage which generated the failure and omits any following Stages.

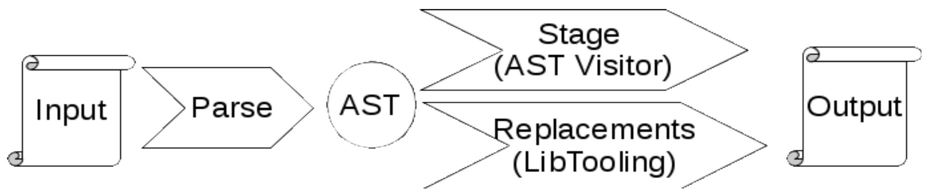

Figure 4: Stage data flow

The structure of a Stage consists of the following Clang components:

A derived class from RecursiveASTVisitor [8] which contains all the user defined actions for a Translation Unit. An ASTConsumer [8] class which handles a Translation Unit by calling the derived RecursiveASTVisitor object over it. One ConsumerFactory [8] object that creates a new ASTConsumer object to consume the current Translation Unit. It is used in combination of the following component, one RefactoringTool [8] object whose purpose is to run the refactoring actions based on the LibTooling infrastructure [8] and to coordinate the refactoring of all translation units.

The invocation happens as shows in Figure 4.

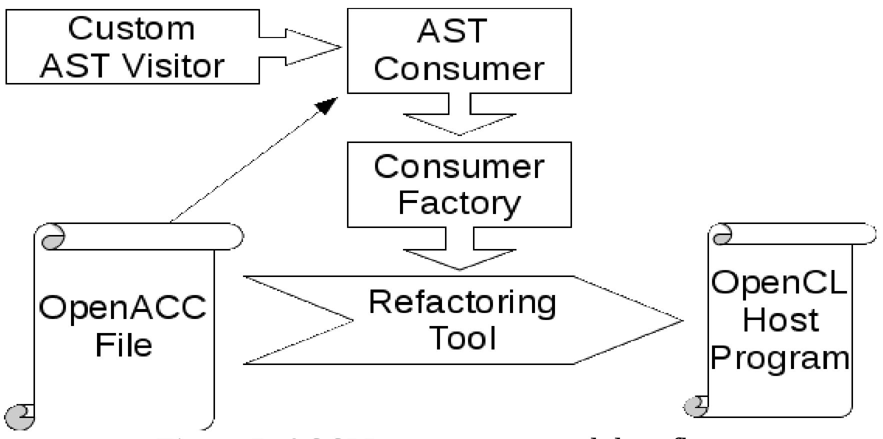

Figure 5: ACCLL components and data flow

The next subsections discuss each Stage separately.

5.2 Input File Validation

This Stage takes the initial list of input files provided by the user and checks whether they contain OpenACC directives. It is also responsible for the creation of any new files. The following Stages write to these new files and do not create other files.

The common case is to create two new files for each input file, one containing the OpenCL host program and one containing the OpenCL device code (kernels). In case there are no OpenACC Compute Constructs in a specific input file, the tool creates only the host program and in case there are no OpenACC directives at all, the tool ignores this input file. The original input files remain untouched in any case.

At the end of this Stage the new file which is going to contain the OpenCL host program is just a copy of the original file. The new file with OpenCL device code (kernels) is empty at the moment. The next Stages take the list of new files and modify them appropriately.

5.3 Unraveling of the If Clause

This Stage acts only on directives which contain a 'if' clause. It constructs the control flow of the OpenACC region at C/C++ level based on the 'if' clause's semantics. A simple If statement which tests the condition expression of the 'if' clause argument is enough. The ‘then’ statement contains the parallel device version of the original code region and the ‘else’ statement contains the host version.

The host version remains untouched as it is the one that executes the region sequentially in the host, ignoring any OpenACC Constructs nested in the annotated code region. The device version is the one that sends the computation for execution to a device according to the OpenACC execution model. The selection between these two versions depends on the boolean value of the argument of the If clause which is a valid C/C++ condition expression.

For example the tool restructures the following code:

```
#pragma acc parallel if (cond)
        { /* do something */ }
```

Code Example 1: Directive with 'if' clause

into this code:

```
if (cond) {
        #pragma acc parallel
                { /* do something */ }
}else
{ /* do something */ }
```

Code Example 2: Restructured directive with 'if' clause

This Stage reveals the control flow of the code region by restructuring the original C code with a transformation at the language level. It does not perform any other kind of transformation.

5.4 Resolve Ambiguity of Parallelization Method

In some cases the OpenACC Standard does not specify the kind of parallelization a segment of code is subject to. The implementation has to decide in these cases the desired behavior.

For example the *x = 10;* statement below can be either a separate kernel running on multiple workers or part of the previous kernel running on a single worker:

```
#pragma acc kernels
        {
                //device code;
                #pragma acc loop
                for ()
                {
                        //distribute computation between workers/groups
                }

                x = 10;  //statement with unspecified parallelization method
        }
```

Code Example 3: Ambiguous parallelization method

These cases of unspecified parallelization are an example of unspecified behavior in the OpenACC Specification. Hopefully this kind of code does not appear very often in a real life application. This implementation takes a safe path by keeping these statements of unclear parallelization status to be executed only by one worker at device. The programmer is responsible for restructuring the code to make sure these unspecified statements are going to be parallelized correctly on device. The Stage emits a warning if the code suffers from this issue encouraging the programmer to rewrite the specific code segment with no ambiguities.

This Stage rewrites the above example as follows:

```
#pragma acc kernels
	{
		//device code;
		#pragma acc loop
		for ()
		{
			//distribute computation among workers/groups
		}

		#pragma acc loop gang(1), worker(1)
		for (int __i__ = 0; __i__ < 1; ++__i__)
		{
			x = 10;  //executed by one worker only
		}
	}
```

Code Example 4: Resolved ambiguity of parallelization method

Note that this behavior mimics the semantics of the parallel directive and may change in later versions of this tool or later OpenACC Revisions.

This Stage acts exclusively on a kernels directive Construct. While other Stages rewrite the OpenACC Constructs to the lower OpenCL API, this Stage performs a higher level transformation by restructuring OpenACC Constructs at the OpenACC programming model level.

This Stage and the previous one are preparation Stages before the actual transformation from the OpenACC programing model to the OpenCL programing model.

5.5 Data Transfers

This Stage lowers the OpenACC data clauses and the update directive into OpenCL API calls. It also searches for any implicit data movements and generates code for them according to the specified semantics.

For example in the following OpenACC region, the movement of x is explicit but the movement of y is implicit:

```
    #pragma acc parallel copy(x)
        {
                x = y;
        }
```

Code Example 5: Implicit and explicit data movements

For each directive which can hold data clauses a wrapper scope is created (an opening bracket ‘{‘ before entry to the region, and a closing bracket ‘}’ when the region is complete). With this method we can reuse variable names for memory objects and other useful structures for every different directive, without caring about possible variable redeclarations.

There are four main different categories (types) of data clauses for a data buffer and four actions which complete a particular data movement. Some of them are optional depending on the data clause type's semantics.

Data clause types	
copy	present_or_copy
copyin	present_or_copyin
copyout	present_or_copyout
create	present_or_create

Table 6: Data clause types

Data movement actions
create a cl_mem object
move the data to the device before entry to the region
move the data back to the host when the region is complete
release the cl_mem object

Table 7: Data movement actions

The clause variants with the ‘present’ prefix do not create/release a new cl_mem object. Such an object is supposed to be present from a previously defined data clause of an already open implicit or explicit data region. If the implementation cannot find a matching cl_mem object in this case, a compile or runtime error will be triggered depending on the implementation’s approach. In the current version of this implementation this triggers a compile time error.

For example the following code:

```
#pragma acc data copy(x)
        { /* code */ }
```

Code Example 6: Data movement in OpenACC

transforms to:

```
{
        cl_mem __accll_x = clCreateBuffer(context,
                                          CL_MEM_READ_WRITE,
                                          sizeof(int), NULL, &error);
        clCheckError(error, "create buffer for 'x'");
        error = clEnqueueWriteBuffer(queue, __accll_x, CL_TRUE, 0,
                                  sizeof(x), &x, 0,NULL, NULL);
        clCheckError(error, "write buffer 'x'");
        { /* code */ }
        error = clEnqueueReadBuffer(queue, __accll_x, CL_TRUE, 0,
                                  sizeof(x), &x, 0,NULL, NULL);
        clCheckError(error, "read buffer 'x'");
        error = clReleaseMemObject(__accll_x);
        clCheckError(error, "release '__accll_x'");
}
```

Code Example 7: Data movement in OpenCL

This Stage also handles any wait directive on asynchronous data movements (from update directives) transforming them to the appropriate OpenCL API calls as Table 5 on chapter 4 shows. It is a design decision not to handle the case of asynchronous Compute Constructs here but in a separate stage (subsection 5.7) in concern of a more sane organization of the actions each stage performs.

The examples below show an asynchronous data movement due to an asynchronous update directive with the creation of the device buffers (cl_mem objects) before and after transformation from OpenACC to OpenCL. Note that the copy clause transfers the data buffers to device at the beginning of the region and back to the host at the end of the region.

```
int a, c;
#pragma acc data copy(a)
{
        //do something
        #pragma acc update host(a) async(c)
}
```

Code Example 8: OpenACC asynchronous data movement

```
int a, c;
//#pragma acc data copy(a)
{
        cl_mem __accll_a =
        clCreateBuffer(context, (1 << 0), sizeof(int), ((void *)0), &error);
        clCheckError(error, "create buffer for 'a'");
        error = clEnqueueWriteBuffer(queue, __accll_a, CL_TRUE, 0,
                                sizeof(int), &a, 0, NULL, NULL);
        clCheckError(error, "write buffer 'a'");

        //#pragma acc update host(a) async(c)
        {
                cl_event __accll_update_event_tmp_0;
                error = clEnqueueReadBuffer(queue, __accll_a, CL_FALSE, 0,
                                        sizeof(int), &a, 0, NULL,
                                        &__accll_update_event_tmp_0);
                clCheckError(error, "read buffer 'a'");
                const cl_event __accll_event_wait_list[] = {
                __accll_update_event_tmp_0 };
                error = clEnqueueMarkerWithWaitList(queue, 1,
                                __accll_event_wait_list,
                                &__accll_update_event_c_0);
                clCheckError(error, "marker with wait list");
        }
        error = clEnqueueReadBuffer(queue, __accll_a, CL_TRUE, 0,
                                sizeof(int), &a, 0, NULL, NULL);
        clCheckError(error, "read buffer 'a'");
        error = clReleaseMemObject(__accll_a);
        clCheckError(error, "release 'a'");
}
```

Code Example 9: OpenCL asynchronous data movement

5.6 Kernel Preparation

This Stage deploys the work-item functions like get_global_id() and get_local_id() as specified at 6.12.1 of OpenCL Specification to reveal geometry related information to each worker on the device (location inside the index-space).

This is the appropriate time to initialize the loop-control-variables of loop directives with the worker’s local or global ID depending on the semantics of the current Compute Construct.

The loop-body-statement of an OpenACC annotated For sub-statement is the code segment of device code about to become a unique discrete OpenCL kernel.

The next step is to enclose the device code in a If statement checking the condition-expression of the For sub-statement. This enhances code correctness taking into account the possibility of differences between for-range and kernel index space geometry during execution.

For example the following loop directive:

```
#pragma acc loop gang(100)
        for (int i=0; i<100; ++i)
        {
                z[i] = x[i] + y[i];  //kernel body
        }
```

Code Example 10: Device Code in OpenACC

will become:

```
__kernel
void uniqKernelName(__global int *x, __global int *y, __global int *z)
{
        int i = get_global_id(0);
        if (i < 100) {
                z[i] = x[i] + y[i];
        }
}
```

Code Example 11: Device Code in OpenCL (kernel)

A similar wrapping with geometry information is done in the body of the parallel directive as well. All the kernel parameters are passed as pointers even if they are constant scalar values. The reason for this approach is that there is no analysis Stage at the time to mark them as read-only. This decision may produce final code with non optimal performance depending on the underlying OpenCL implementation and the way it passes the various kinds of kernel parameters.

5.7 Generation of OpenCL kernels and API calls

This is the stage in which most of the OpenACC to OpenCL transformation takes place. Having all the data transfers already present in the code, what is left is to populate the device code output file with the device code (kernels), enrich the host program with OpenCL API calls and perform some proper error checking.

The OpenACC Constructs that must be translated into OpenCL kernels are the parallel directive and the Combined directives (parallel loop, kernels loop). The kernels directive defines a collection of OpenCL kernels rather than a standalone OpenCL kernel. More details about some unspecified behavior regarding this directive are given in detail in section 5.4.

This Stage performs the following actions, where most of them are pretty straightforward at this point thanks to the work accomplished by the previous Stages:

It creates each kernel's signature (definition) and body (declaration) handling any private and firstprivate parameters with possible data initialization. The generation of any reduction algorithm also happens here. Kernels are written to a separate file (*.cl file).

It calls the generated kernels in the appropriate places in the host program by creating any cl_kernel objects, preparing their arguments and the proper geometry. This is the perfect time to orchestrate any event based synchronization needed with cl_event objects for each asynchronous kernel invocations, populating the wait lists of the proper OpenCL API calls, enqueue them to a command queue, waiting for them to complete and finally releasing any OpenCL resources (cl_kernel, cl_mem objects) used for the specific kernel execution command. Note that data movements asynchronous or not are handled in a previous stage (chapter 5.5).

The following examples show an asynchronous kernel invocation before and after transformation from OpenACC to OpenCL, omitting any data movements for simplicity.

```
int x, y;
#pragma acc parallel async(1)
{
        x = y;
}
```

Code Example 12: OpenACC asynchronous device code

```
Int x, y;
//#pragma acc parallel async(1)
{
        size_t global_ws = 1;
        size_t local_ws = 1;
        cl_kernel accll_kernel_parallel_0 =
                clCreateKernel(program, "accll_kernel_parallel_0", &error);
        clCheckError(error, "create accll_kernel_parallel_0");
        error = clSetKernelArg(accll_kernel_parallel_0, 0, sizeof(cl_mem),
                &__accll_x);
        clCheckError(error, "clSetKernelArg 0 for 'accll_kernel_parallel_0'");
        error = clSetKernelArg(accll_kernel_parallel_0, 1, sizeof(cl_mem),
                &__accll_y);
        clCheckError(error, "clSetKernelArg 1 for 'accll_kernel_parallel_0'");
        error = clEnqueueNDRangeKernel(queue, accll_kernel_parallel_0, 1,
                NULL,&global_ws, &local_ws, 0, NULL,
                &__accll_device_code_event_const_1_0);
        clCheckError(error, "enqueue accll_kernel_parallel_0");
        error = clReleaseKernel(accll_kernel_parallel_0);
        clCheckError(error, "release accll_kernel_parallel_0");
}
```

Code Example 13: OpenCL asynchronous kernel invocation

This Stage is also responsible for the loading & building actions over the OpenCL kernels in the host program. This feature is implemented as a library routine in the ACCLL Runtime library (chapter 7.2). Usually this code is placed at the top of the entrance point of the host program (main function).

5.8 Formatting the Output

The last Stage aims to polish the output by properly applying a formatting style over the final code like indentation, spaces newlines, etc. It does not aim to alter the functionality of the code, just to improve readability.

It was feasible to format the code at the same time with the Replacements mechanism [8], but this approach did not seem very attractive for two reasons. First of all it requires a great amount of data keeping for each Stage separately just to offer a reading-friendly representation in the intermediate states of code no one is typically expected to see, except for debugging purposes. It also obfuscates and bloats the implementation with slow code (calculating indentation, etc).

Having the style formatting logic in a distinct Stage is cleaner, less error prone, fits better with the whole separate Stage approach of the tool, allows changes to the formatting style in a more consistent way. Thankfully all these features are already implemented in a different Clang based tool named clang-format [8]. This Stage is really a stripped down version of this tool.

For the debugging purposes of an intermediate Stage, the missing formatting style in conjunction with very few newline characters, can be an issue (hard to read output). This can be easily circumvented by bypassing all the following Stages of the erroneous Stage but the formatting Stage (this one).

5.9 Runtime Support

The runtime system is a core component for most programming models and programming languages. It works together with the compiler usually to simplify the implementation and maintenance of the non-trivial features. While it is more flexible to implement a specific functionality at the runtime level, embedding it directly into the compiler can increase the optimization opportunities and final code quality overall. This is a tradeoff decision most of the times without a priori knowledge about the actual advantages and disadvantages in each case.

The following subsections describe some decisions taken about the OpenACC and ACCLL Runtime systems.

5.9.1 OpenACC Runtime Support

The implementation of the OpenACC Runtime is a wrapper for OpenCL API calls that offer the desired functionality. In most cases the necessary functionality is pretty straightforward to implement.

The tool replaces some of these routines in early Stages with the semantically equivalent directive.

For example the

acc_async_wait_all()

Runtime routine becomes

#pragma acc wait

In this version of the tool, the Runtime is not thread safe. It has been tested only with single host thread test applications. Currently the binary code of the runtime is embedded into the user's executable which may not be ideal. A shared library would surely be a better approach and a transition to that is under development.

5.9.2 ACCLL Runtime Library

This Runtime library contains useful helper functions for common actions in the final transformed code, like error checking, loading the OpenCL kernels source code from a separate file, building that source, initializing the OpenCL Runtime and shutting it down.

It is preferable to have these functionalities into a separate library for easier modification, extensibility and debugging.

5.10 Asynchronous Execution Model Details

In the context of the asynchronous execution model, each command executes without depending on the execution status of any previous commands. Asynchronous execution models provide a way to exploit all the compute power of a device by running many independent computations at the same time, without waiting one another. Even in these types of workloads there is need of coordination at some level. The next subsections outline the Asynchronous and Blocking Execution Model in OpenCL [4].

5.10.1 Blocking Execution

To express the blocking behavior over an Out-of-Order Command Queue, the implementation implicitly enqueues a Barrier after each Compute Command. The data movement Commands are used with the blocking_[read | write] parameter set to CL_TRUE.

5.10.2 Asynchronous Execution and Synchronization

The asynchronous execution in terms of OpenCL structures is expressed by an Out-of-Order Command Queue. Among the compute and data movement commands, the implementation inserts Barriers and Markers in proper places to control the control flow as demanded by the user.

More details and examples about the implementation's asynchronous execution support can be found on sub chapters 5.5 and 5.7 for asynchronous data movements and asynchronous kernel execution respectively.

As an extension of OpenACC 2.0, a single wait directive can wait for both asynchronous data movements and asynchronous Compute Constructs. The process of these asynchronous actions of different nature is splitted in different Stages. Doing this separation in each Stage independently is easier to implement, debug and optimize.

Chapter 6

Evaluation & Testing

The validity of the result OpenCL code is tested only with Intel's OpenCL runtime implementation version 3.0.67279 on a 64-bit GNU/Linux system running Debian unstable [30] and kernel version 3.11. Therefore some of the issues discussed below may behave differently using another operating system or underlying OpenCL runtime.

There is a number of different test files from various places like the yacf project [24] related to accULL project [25], the HydroBench Suite [26], the EPCC OpenACC Benchmark Suite [27], the NPB2.3 OpenACC-C [28] and the OpenACC Testcase [29]. The last three can be found on the public online repository of Pathscale [23].

All the above benchmark suites contain similar test cases for proper evaluation of all code transformations like kernel generation/calling, error handling data movements and updates this tool applies. The OpenACC Testcase [29] in particular consists of some simple yet complete test cases that cover all the aforementioned code transformations and the majority of its test files pass.

This first version of the ACCLL tool is mostly a research and experimentation project which is quite far from productive usage, limited and not feature complete. The next paragraphs point the most significant known limitations among the test files they affect.

The test file "alias_pointer.c" fails as it requires proper alias analysis, still a missing feature of this implementation as discussed in 3.3.3. Therefore aliasing of memory buffers is not supported in this version of this tool.

Another issue is that due to some pending bugs, Clang may return incorrect ending source locations for the preprocessor's macros in an expression, resulting one more test file ("macro_func.c") to fail because of the incorrect information Clang passes to the tool. This regression is triggered only when a macro is used at the end of an expression or statement. There is no general workaround in this version of the tool. The safest path to circumvent this issue is not to use macros inside OpenACC Compute Constructs. A rather more flexible workaround is not to use macros at the end of expressions or statements inside OpenACC Compute Constructs.

Inter-procedural data flow analysis and dynamic memory allocation analysis are also missing, causing one more test file to fail ("present_pointer.c").

There is also a collection of internal test files mostly for parser validation, semantics correctness and proper warning/error messages emission.

One common regression between all test cases is that because the tool generates hard coded index-spaces for all kernel invocations, the size of it may not be supported by an accelerator selected at runtime. This issue almost always appears when there are data buffers with more elements that the current accelerator's threads. Some of the test files contain such data buffers that issue and fail to pass. Using a smaller buffer size in order not to exceed the accelerator's number of threads, all of them pass without any other issue.

The only workaround at the moment is advising the programer to manually edit their programs by explicitly specifying the expected or desired geometry of the device code in the OpenACC directive using the gangs and workers clauses and also properly edit the device code to take care of this smaller index-space.

The drawbacks of this workaround are that in order to achieve maximum performance, the programer should provide different versions of the OpenACC Compute Constructs (parallel, kernel, loop directives and its combinations) for accelerators with different number of threads, with loss of portability and code duplication among other side effects.

Finally multi dimension arrays are not fully supported as there is no support yet for multi dimensional index spaces as discussed on chapter 4.

Table 8 below gives a detailed description of the OpenACC Testcase benchmark [29] and the status of each test file.

Test file name	Pass/Fail (Reason)
alias_pointer.c	FAIL (alias analysis)
async_wait.c	PASS
declare_update.c	PASS
dyn_cont_array_2d2.c	FAIL (multi dimensional array)
dyn_cont_array_2d.c	FAIL (multi dimensional array)
func_decl.c	PASS
macro_func.c	FAIL (bad source locations, Clang bug)
present_pointer.c	FAIL (inter-procedural data flow analysis)
reduction.c	PASS
simple_initialize.c	PASS
static_array_2d.c	FAIL (multi dimensional array)
struct_mem.c	PASS
var_duration.c	PASS

Table 8: OpenACC Testcase evaluation

As this implementation is subsequent to existing commercial implementations, one of its goals is to be compatible with them. These test files among the the correctness also check for compatibility between existing solutions. In the current version of the accll tool there are some small deviations in the data clauses manipulation causing failure to a significant number of relatively more complicated test files. As this project grows and matures these defects will be eliminated and full compatibility with other implementations is the main goal.

Conclusion

Given the increasing need for performance year by year, programmers will always be in the search for better tools, simpler programming models and faster system infrastructures to run their code.

Moving towards the new era of parallel computing, there are tons of ‘legacy’ sequential applications not exploiting the advanced capabilities of parallel architectures.

This project at the moment is mainly a Proof of Concept of the fact that a big majority of all these ‘legacy’ sequential applications can easily experience a huge performance boost using a user friendly way of minimum changes in their current codebase by inserting OpenACC directives. Also it shows that this new annotated code can be reliably transformed to a broadly adopted, well defined and tested framework like OpenCL.

A complete example: VectorAdd

Below there is a complete example of a simple vector addition program in the original OpenACC version and the same program in OpenCL this tool generates, both host program and device code (kernels):

OpenACC version

```
#include <stdio.h>
#define SIZE 100


int main(int argc, char **argv) {
    int i;
    int x[SIZE];
    int y[SIZE];
    int z[SIZE];
    int w[SIZE];
    //init
    for (i=0; i<SIZE; ++i) {
        x[i] = i;
        y[i] = SIZE - i;
        z[i] = 0;
    }

    //host computation
    for (i=0; i<SIZE; ++i) {
        w[i] = x[i] + y[i];
    }
    //device computation
#pragma acc kernels loop
copyin(x,y) copyout(z) gang(SIZE)
    for (i=0; i<SIZE; ++i) {
        z[i] = x[i] + y[i];
    }

    //check results
    for (i=0; i<SIZE; ++i) {
        if (w[i] != z[i]) {
            printf("w[%d] = %d    !=
z[%d] = %d\n",i,w[i],i,z[i]);
            return 1;
        }
    }

    printf("Success!\n");
    return 0;
}
```

Code Example 14: VectorAdd in OpenACC

OpenCL version (host program)

```
/* Generated by accll */
#include <stdio.h>
#include <__accll.h>
#include <stdio.h>
#define SIZE 100

int main(int argc, char **argv) {
  __accll_init_accll_runtime();
  cl_program program =
      __accll_load_and_build("/opt/accll/mine/test-vector-add_accll.cl");
  int i;
  int x[SIZE];
  int y[SIZE];
  int z[SIZE];
  int w[SIZE];

  //init
  for (i = 0; i < SIZE; ++i) {
    x[i] = i;
    y[i] = SIZE - i;
    z[i] = 0;
  }

  //host computation
  for (i = 0; i < SIZE; ++i) {
    w[i] = x[i] + y[i];
  }
```

Code Example 15: VectorAdd in OpenCL (host program)

```
//device computation
  {
    cl_mem __accll_z = clCreateBuffer(context, (1 << 0), sizeof(int[100]),
                                  ((void *)0), &error);
    clCheckError(error, "create buffer for 'z'");
    cl_mem __accll_y = clCreateBuffer(context, (1 << 0), sizeof(int[100]),
                                  ((void *)0), &error);
    clCheckError(error, "create buffer for 'y'");
    error = clEnqueueWriteBuffer(queue, __accll_y, CL_TRUE, 0,
sizeof(int[100]),
                              y, 0, NULL, NULL);
    clCheckError(error, "write buffer 'y'");
    cl_mem __accll_x = clCreateBuffer(context, (1 << 0), sizeof(int[100]),
                                  ((void *)0), &error);
    clCheckError(error, "create buffer for 'x'");
    error = clEnqueueWriteBuffer(queue, __accll_x, CL_TRUE, 0,
sizeof(int[100]),
                              x, 0, NULL, NULL);
    clCheckError(error, "write buffer 'x'");
    //#pragma acc kernels loop copyin(x, y), copyout(z), gang(100)
    cl_event __accll_device_code_event_implicit__0;
    size_t global_ws = 100;
    size_t local_ws = 1;
    cl_kernel accll_kernel_kernels_loop_0 =
        clCreateKernel(program, "accll_kernel_kernels_loop_0", &error);
    clCheckError(error, "create accll_kernel_kernels_loop_0");
    error = clSetKernelArg(accll_kernel_kernels_loop_0, 0, sizeof(cl_mem),
                          &__accll_x);
    clCheckError(error,
                "clSetKernelArg number 0 for 'accll_kernel_kernels_loop_0'");
```
Code Example 16: VectorAdd in OpenCL (host program), part 2

}

```
error = clSetKernelArg(accll_kernel_kernels_loop_0, 1, sizeof(cl_mem),
                         &__accll_y);
    clCheckError(error,
                 "clSetKernelArg number 1 for 'accll_kernel_kernels_loop_0'");
    error = clSetKernelArg(accll_kernel_kernels_loop_0, 2, sizeof(cl_mem),
                         &__accll_z);
    clCheckError(error,
                 "clSetKernelArg number 2 for 'accll_kernel_kernels_loop_0'");
    error = clEnqueueNDRangeKernel(queue, accll_kernel_kernels_loop_0,
1, NULL,
                                 &global_ws, &local_ws, 0, NULL,
                                 &__accll_device_code_event_implicit__0);
    clCheckError(error, "enqueue accll_kernel_kernels_loop_0");
    error = clEnqueueBarrierWithWaitList(
        queue, 1, &__accll_device_code_event_implicit__0, NULL);
    clCheckError(error, "barrier with wait list");
    error = clReleaseKernel(accll_kernel_kernels_loop_0);
    clCheckError(error, "release accll_kernel_kernels_loop_0");
    error = clReleaseMemObject(__accll_y);
    clCheckError(error, "release 'y'");
    error = clReleaseMemObject(__accll_x);
    clCheckError(error, "release 'x'");
    error = clEnqueueReadBuffer(queue, __accll_z, CL_TRUE, 0,
sizeof(int[100]),
                              z, 0, NULL, NULL);
    clCheckError(error, "read buffer 'z'");
    error = clReleaseMemObject(__accll_z);
    clCheckError(error, "release 'z'");
  }
```

Code Example 17: VectorAdd in OpenCL (host program), part 3

```
//check results
  for (i = 0; i < SIZE; ++i) {
    if (w[i] != z[i]) {
      printf("w[%d] = %d    !=    z[%d] = %d\n", i, w[i], i, z[i]);
      return 1;
    }
  }

  printf("Success!\n");
  return 0;
}
```

Code Example 18: VectorAdd in OpenCL (host program), part 4

OpenCL version (device code – kernels)

```
/* Generated by accll */

__kernel void accll_kernel_kernels_loop_0(__global int *__accll_x,
                                          __global int *__accll_y,
                                          __global int *__accll_z) {
  /*Main Body*/
  {
    int i = get_global_id(0);
    if (i < 100) {
      __accll_z[i] = __accll_x[i] + __accll_y[i];
    }
  }
}
```

Code Example 19: VectorAdd in OpenCL (device code - kernel)

Bibliography

[1] OpenACC: OpenACC directives for accelerators, CAPS Enterprise, Cray Inc., NVIDIA, and the Portland Group. The OpenACC Application Programming Interface, v1.0 (November 2011). http://www.openacc-standard.org/

[2] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in Proc. of the 2004 Intl. Symp. on Code Generation and Optimization (CGO’04), Palo Alto, CA, USA, Mar. 2004.

[3] Clang: a C language family front end for LLVM
http://clang.llvm.org/

[4] Khronos OpenCLWorking Group: The OpenCL Specification, Version 1.2, Rev. 15 (November 15, 2011)

[5] OpenMP Architecture Review Board
http://openmp.org/wp/ . August 2013

[6] MPI, Message Passing Interface Forum,"MPI: A Message-Passing Interface Standard", Version 3.0, September 2012. http://www.mpi-forum.org/docs/docs.html.

[7] The UPC Consortium: UPC language specifications (v1.2) (2005). http://www.gwu.edu/~upc/docs/upc_specs_1.2.pdf

[8] The LLVM/Clang documentation
http://clang.llvm.org/docs/

[9] Adam Betts , Nathan Chong , Alastair Donaldson , Shaz Qadeer , Paul Thomson, GPUVerify: a verifier for GPU kernels, Proceedings of the ACM international conference on Object oriented programming systems languages and applications, October 19-26, 2012, Tucson, Arizona, USA

[10] Muhsen Owaida , Nikolaos Bellas , Konstantis Daloukas , Christos D. Antonopoulos, Synthesis of Platform Architectures from OpenCL Programs, Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines, p.186-193, May 01-03, 2011

[11] Wikipedia, Heterogeneous Computing
http://en.wikipedia.org/wiki/Heterogeneous_computing

[12] Intel® VTune Performance Analyzer
http://software.intel.com/en-us/intel-vtune-amplifier-xe

[13] Intel® Graphics Performance Analyzers
http://software.intel.com/en-us/vcsource/tools/intel-gpa

[14] AMD APP Profiler
http://developer.amd.com/tools-and-sdks/heterogeneous-computing/archived-tools/amd-app-profiler

[15] GCC,The GNU Compiler Collection.
http://gcc.gnu.org/

[16] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. TOPLAS, 13:451–490, 1991.

[17] cfe-dev -- Clang Front end for LLVM Developers' List
http://lists.cs.uiuc.edu/mailman/listinfo/cfe-dev

[18] Cray Inc.: Cray, the Supercomputer Company
http://www.cray.com

[19] CAPS: The Many-Core Programming Company
http://www.caps-entreprise.com/

[20] NVIDIA
http://www.nvidia.com

[21] PGI: The Portland Group
http://www.pgroup.com/

[22] CUDA, 2013. NVIDIA CUDA 5. http://www.nvidia.com/object/cuda_home_new.html.

[23] PathScale Inc.
http://www.pathscale.com

[24] yacf, Yet Another Compiler Framework
https://code.google.com/p/yacf/

[25] accULL, A programming environment for heterogeneous architectures
http://cap.pcg.ull.es/en/accULL

[26] A 2D Hydro code for benchmarking purpose
https://github.com/HydroBench/Hydro

[27] EPCC OpenACC benchmark suite
https://github.com/pathscale/EPCC_OpenACC_benchmark-suite

[28] NAS Parallel Benchmark OpenACC C Version
https://github.com/pathscale/NPB2.3-OpenACC-C

[29] OpenACC Testcase
https://github.com/pathscale/OpenACC-Testcase

[30] Debian Project
http://www.debian.org/