Asian Journal of Computer and Information Systems (ISSN: 2321 – 5658)
Volume 04– Issue 01, February 2016

Review of LLVM Compiler Architecture Enhancements for CUDA

Munesh Singh Chauhan

College of Applied Sciences
Ibri, Oman
Email: munesh.ibr [AT] cas.edu.om

ABSTRACT— *Heterogeneous platforms are now becoming increasing omnipresent due to the availability of multicores at commodity prices. In order to benefit from the immense parallel capability of these multicores, more and more applications are now being developed as well as ported using the CUDA framework that programs these cores. Hence it becomes imperative to devise new compiler paradigms that cater varied languages and provide easy and flexible multicore programming. LLVM compilers have traditionally been the bedrock for such endeavors. New runtime processes are being researched to make CUDA platform more amenable for supporting a maximum set of language architectures. An analysis is made to understand the advantages and the accompanying pitfalls of various types of linking techniques.*

Keywords— Graphical Processing Unitt; Compute Unified Device Architetcure, LLVM Compiler

1. INTRODUCTION

The CUDA GPU compiler is based on the LLVM (Low Level Virtual Machine) Compiler infrastructure. LLVM compiler is the most commonly used compiler base for extending present-day languages as well for developing future languages. Most of the parallel programming languages in particular use LLVM IR for the back end.

GPU Computing in order to be ubiquitous need to embrace as many programming languages as possible. This can be achieved by creating a fertile eco-system based on CUDA platform. New languages can be supported on CUDA by creating new Front Ends (FEs) that compile the language into LLVM specified IR (Internal Representation). The IR thus generated can be further optimized for the target GPU hardware. The next step is to develop a hardware specific back-end that compiles the generated IR. LLVM has been modified by NVIDIA to support multithreading, which happens to be a fundamental multicore computing technique. The entire process is described in Figure 1.

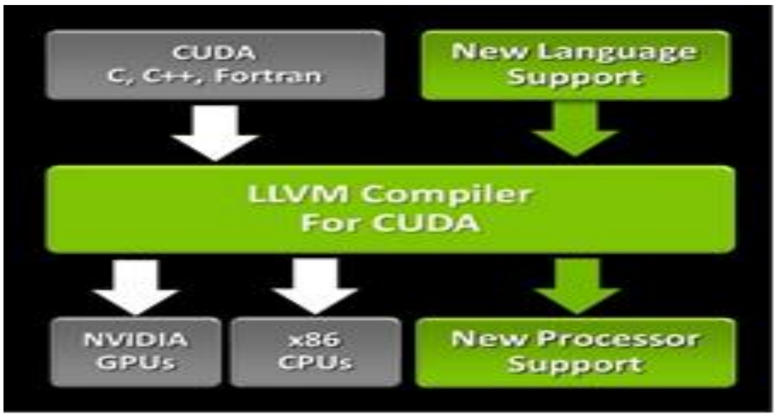

Figure 1: LLVM Compiler Description for CUDA

The benefits of supporting multiple languages using LLVM are outlined briefly as under:

1. Each language possesses its unique characteristic features that aid parallel execution, such as multithreading, modular procedures, etc.
2. CUDA based GPU computing can only become all-pervasive if it can support multiple language sets.

A parallel computing platform that supports multiple languages tends to encourage researchers from diverse fields and backgrounds. This in fact benefits in speeding-up legacy and erstwhile applications that have been languishing for want of phenomenal computing power.

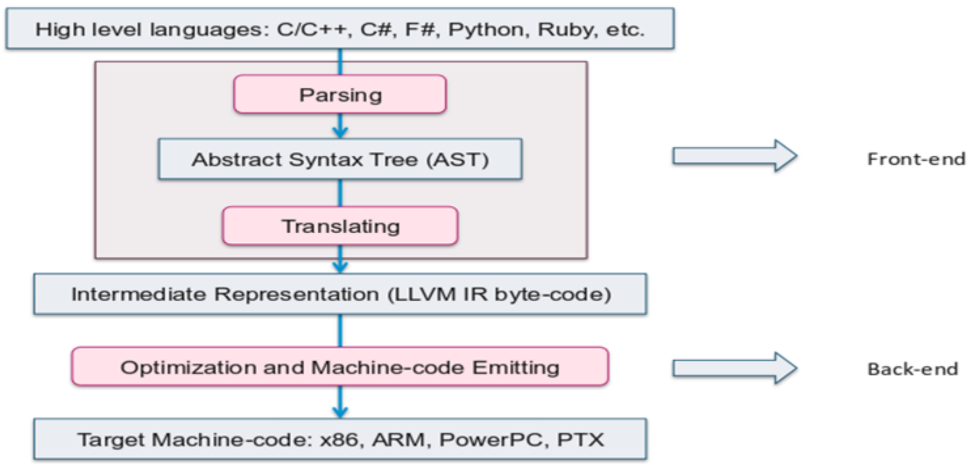
High level languages: C/C++, C#, F#, Python, Ruby, etc.
Parsing
Abstract Syntax Tree (AST)
Translating
Front-end
Intermediate Representation (LLVM IR byte-code)
Optimization and Machine-code Emitting
Back-end
Target Machine-code: x86, ARM, PowerPC, PTX

Figure 2: LLVM Compiler Description for CUDA

2. LLVM COMPILER

The LLVM compiler comprises of front and back end. The front end takes the lexical elements of the program syntax and parses it into an Intermediate Representation called LLVM IR byte code (Fig 2). The backend comprise code optimization and machine executable generation modules which specifically target a particular hardware. In order to cater CUDA parallel framework, LLVM backend has been modified by NVIDIA to provide parallel primitives.

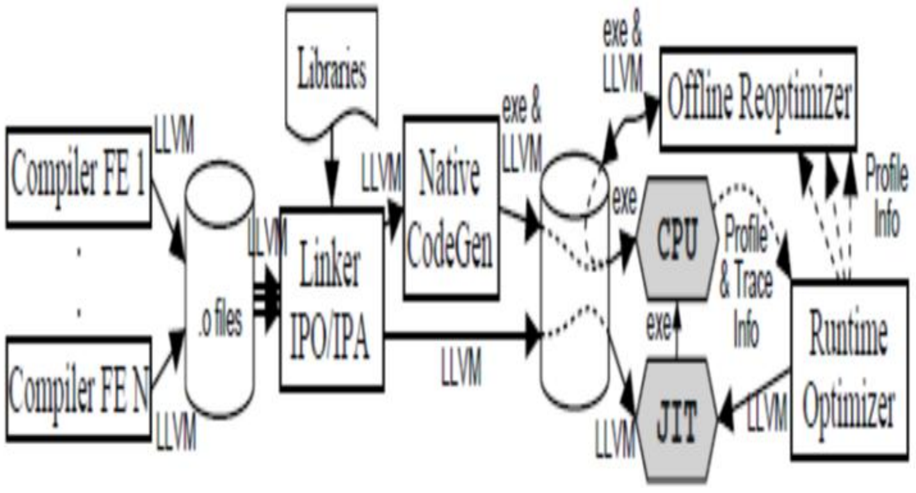

Figure 3: LLVM System Architecture Design

3. CUDA COMPILATION, WORKFLOW & ARCHITECTURE

CUDA stands for Compute Unified Device Architecture. The primary aim of CUDA framework is to provide a platform for harnessing the multithreading capabilities of GPUs (Figure 3). It provides a seamless integration of many well-known languages such as C, C++ into a parallel framework that provides both multi-threading and memory management techniques. Further, it constricts the learning curve which otherwise would have been quite steep if a new parallel language was designed from scratch.

Before going through the workflow, CUDA Compiler Architecture provides the blueprints necessary to describe the various compilation tools that go in executing a typical CUDA parallel source code.

In order to understand the workflow pattern of a CUDA NVCC compiler, Figure 4 dissects the program flow and explains the typical heterogeneous behavior of CUDA that utilizes both the CPU and the GPUs.

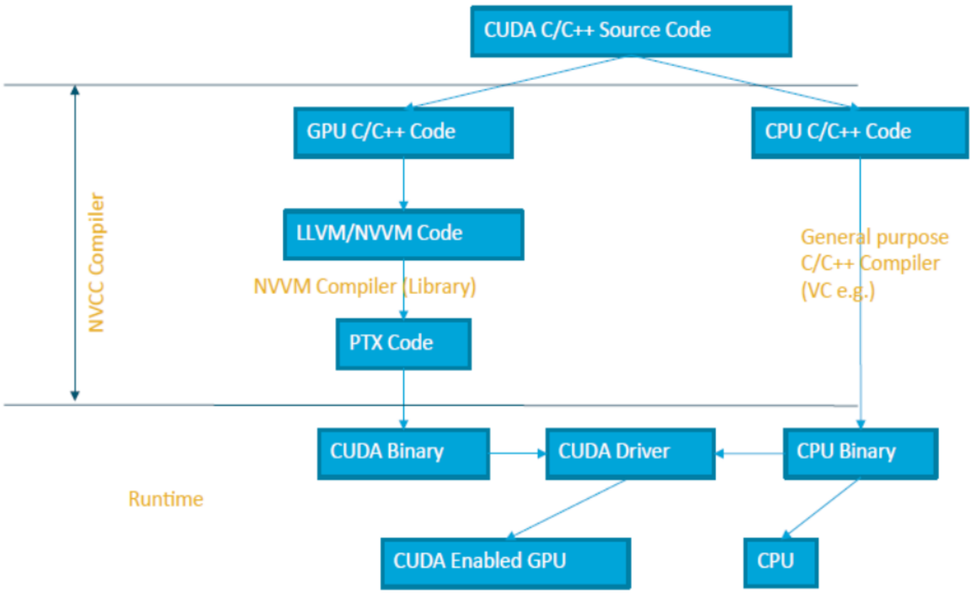

Figure 4: CUDA Backend Compiling & Linking (Source: QuantAlea)

The program is a composite one and proceeds to execute serial segment of code on the CPU whereas the parallel code is sent in the form of kernels that run synchronously on the GPUs. Within the kernel, the codes (mainly the hotspots) are asynchronously distributed over a grid hierarchy of threads that are capable of independent execution.

The specifications of CUDA framework are shown in Table 1 [4].

Table 1: CUDA Framework Specification

Kernel Language	PCUDA C/C++
IR	LLVM/NVVM
ISA	PTX
Machine Code	SASS
Front End Compiler	NVCC Compiler
Back End Compiler	NVVM Compiler
JIT Compiler	CUDA Driver

The CUDA NVVM compiler which is a backend compiler is built in conjunction with LLVM. NVVM has three parts: NVVM IR Specification, NVVM Compiler API, LibDevice. The backend of the CUDA compiler plays a vital role in converting the NVVM IR to the target architecture. Each generated IR module is matched to a template before being linked together into a single PTX Module. Finally the PTX Module is loaded and converted into a program object (Figure 5).

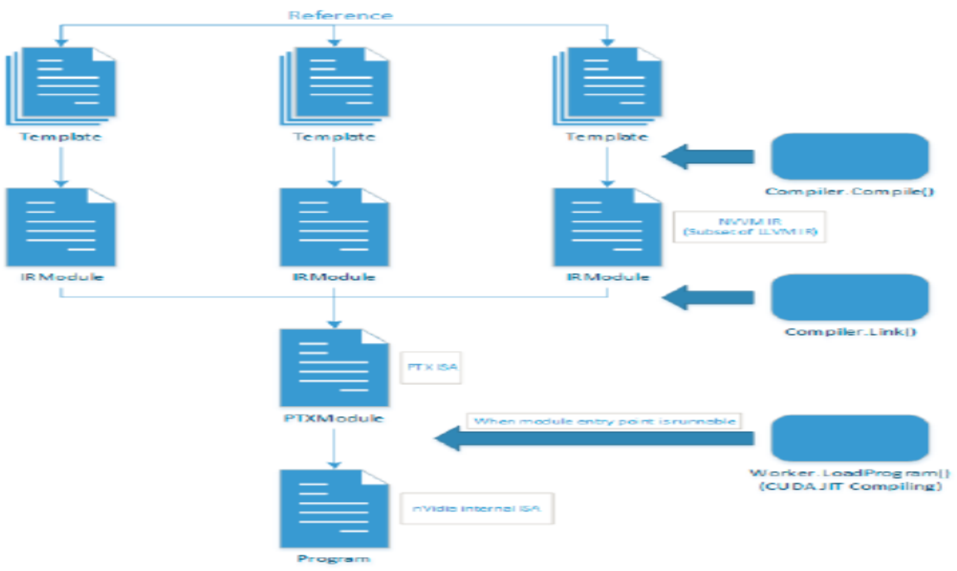

Figure 5: CUDA Backend Compiling & Linking (Source: QuantAlea)

4. NEED FOR A NEXT GENERATION COMPILER PLATFORM FOR GPUS

The clamor for pitching in GPUs for all computational need is short-sighted. GPUs are not the panacea for all general purpose computing tasks. Simple and relatively small applications do not need GPU multithreading model. In fact the applications suffer delays and bottlenecks as very few threads are spawned. Hence there is a need for a new programming model that deals with efficient and automatic in-lining using annotations. This is also referred to as directive based programming [5]. This method provides improved user controlled parallelism using time-proven libraries (directives). Examples of annotation based parallel programming are OpenACC and Open HMPP.

The major disadvantage of the directive based languages is that their compilers do not support generation and invoking of GPU kernels for hotspots such as iterative for and while loops. This severely limits the parallelization efficiency.

In order to mitigate the drawbacks of Directive-Based programming model, Domain Specific Languages (DSL) provide domain specific algorithm enhancements for parallelization. It provides another layer of abstraction and creates avenues for developer interventions in domain specific algorithm parallelization. One of a popular DSL is Halide that is used in image processing applications. But DSL carry an inherent drawback. It needs domain specific programming interventions from developers which at times can be quite challenging.

The other option widely used for enhancing parallelism through compilers is automatic analysis using polyhedral method. Polyhedral analysis for automatic code generation has been widely used in many language conversion tools.

Despite different options available for developing fast parallel compilers as outlined above all the above methods require manual intervention and modification in the code. Thus the challenge to develop efficient compilers still remains elusive.

The expected features needed in a parallel compiler are listed below [5]:

1. Support for all major programming languages.
2. Automatic conversion of parallel hotspots into GPU kernels.
3. Automatic code generation process.
4. Less memory traffic between global memory and GPU.
5. Compatible with other parallel programming paradigms, such as OpenCL, MPI, etc.

5. DESCRIPTION OF THE PRESENT NVCC (CUDA C/C++) BUILD PROCESS

The nvcc compiler generates three categories of code [6]:

1. Host object code (compiled with g++)
2. Device object code
3. Device assemble code (PTX)

All the above three categories of code are compiled into a single fat binary.

Three different pathways are taken when a fat binary code is compiled. The pathways taken depend on the following:

1. If the fat binary is made of device code, the code is executed immediately.
2. If the fat binary code is a PTX assembly then it is compiled by JIT compiler and run on the GPU.
3. If none, the code is not run.

A simplified diagram of the NVCC build process is shown in Figure 6.

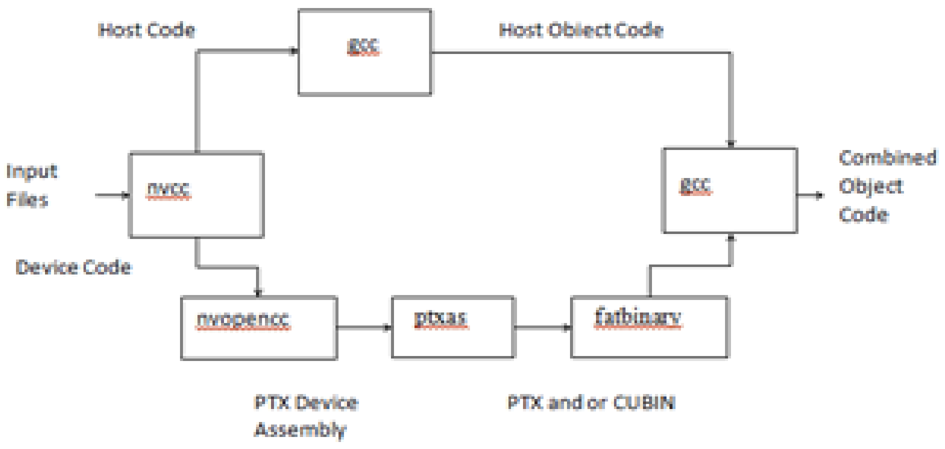

Figure 6: NVCC Build Process

CUDA Compute Compatibility and GPU Architecture

CUDA toolkits come with different Compute Capability (CC) versions as they evolve with time. CUDA compute compatibility provides insight into various features of CUDA compiler. Some of the salient features are listed below:

1. Latest computing features supported by the current generation of the GPUs.
2. Specific language intrinsic properties such as double data types.
3. Specific device related features such as maxThreadsPerBlock, shared, texture memory sizes, warp size, etc.
4. Assembly version of the PTX (Parallel Thread Execution) code.
5. Legacy compatibility.

Some of the specific intrinsic of GPU architecture are also outlined as under:

1. The machine code is architecture specific and varies with different architectures.
2. Object code version.
3. Optimization techniques vary across architectures.
4. Binary code is architecture specific and is not compatible across architectures.

6. PROPOSED COMPILER ROADMAP

As it is clearly seen, CUDA compiler (nvcc) suffers from the compatibility issue whenever a new generation GPU is launched. To maintain a persistent compilation technique is a challenging and tricky task. The major changes need to be done in the compiler build process. So far the compilers are modified to cater to drastic changes in the GPU architecture over a period in time. Some of these overhauls are outlined as under:

1. **Global memory addressing**
   The global memory also known as device memory has seen major changes in its hardwiring. The most prominent change is in the access pattern, such as memory coalescing, broadcasting, bank-conflict-free access and dynamic access. These major issues have already been addressed. As a result for a major part of future GPU Memory Architecture evolution cycles, the status quo will prevail. This means that no new memory specific

intrinsic shall be added to the PTX Assembly code. Hence this part of the compiler shall remain persistent across future architectures.

2. **Texture and Constant Memory**
   Texture and constant buffers mainly are relevant to graphics applications. Since the GPU is now being used for general-purpose parallel computing, graphic specific features are not used so often. Hence there exists least probability of any modification in these graphic-specific buffers.

3. **Thread Hierarchy**
   CUDA thread hierarchy is still under development and modification. In order to parallelize an application a great amount of programmer's intervention is needed. Parallelization on fly is still elusive. Hence CUDA thread hierarchy cannot be hardwired.

Since the silicon real estate prices are on the downtrend, certain radical changes even in the so-called stable segments (as outlined earlier) can be made without incurring massive costs. Many embedded tools have come up which provide dynamic upgrades in the form of firmware support, flash/BIOS memories and FPGAs.

Once the hardwiring or software embedding is done CUDA compiler can be tweaked for major speed-up changes and may provide enhanced application response times.

7. CONCLUSION

CUDA GPU based applications are getting popularity mainly due to the omnipresence of NVIDIA GPUs on commodity personal computing hardware. The major bottleneck that needs to be addressed is speed and ease of parallelization. Speed comes from GPU architecture enhancement and compiler optimizations. Compiler tweaks and optimizations have been discussed and a proposal is made to provide firmware based compiler support to complement the GPU architecture that remains stable through different evolution cycles. This will provide increased application throughput with minimal hardware changes.

8. REFERENCES

[1] Chris Lattner, Vikram Adve, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, University of Illinois at Urbana-Champaign.
[2] Yuan Lin, Building GPU Compilers with libNVVM, GPU Technology Conference.
[3] The LLVM Compiler Infrastructure, http://llvm.org.
[4] Xiang Zhang, Aaron Brewbaker, How to design a language integrated compiler with LLVM, GTC 2014, San Jose, California, March 27, 2014.
[5] Dmitry Mikushin, Nikolay Likhogrud, Eddy Z. Zhang, KERNELGEN – The design and implementation of a next generation compiler platform for accelerating numerical models on GPUs, Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International.
[6] Adam DeConinck, HPC Systems Engineer, NVIDIA, Introduction to CUDA Toolkit for Building Applications.