KernelGen – multi-target accelerated kernels generator for Fortran programs

Dmitry N. Mikushin

maemarcus@gmail.com

August 27, 2011

Abstract

KernelGen is open source-to-source translator with Fortran frontend, multiple backends (currently CUDA, OpenCL and CPU) and common runtime library. Fortran frontend extracts computational loops out of original source code, replacing them with calls to equivalent GPU or APU kernels. Additionally frontend can generate functions for per-kernel automatic results comparison between different implementations of the same loop. Runtime library handles both static (modules data, global variables) and dynamic (variables, fixed shape or allocatable arrays) kernel dependencies, automatically performing data load into device memory and fetching back the results. Runtime is designed thread-safe to provide seamless multi-GPU support for multithread and MPI applications. This article includes both simple test case for demonstration and real application study. To become production-ready prototype a couple of important device code optimization techniques must be implemented in KernelGen: tiling for shared memory utilization and loops interchanging.

1 Motivation

During last 5 years GPGPU technologies became wide-spread. Although mature programming models exist as well as publicly available compilation toolkits, most of applications

ported to GPUs (considering real ones, not highly-tuned benchmarks) are rather simple or utilize GPU computations only for a small subset of code. There is a number of reasons behind this situation:

1. In complex applications it is often impossible to identify and port a single compute-intensive block of code, as soon as computations are distributed between multiple major units. For example in case of WRF or COSMO weather prediction models time of single block typically never exceeds 10-15% of total time. Thus, single block port cannot deliver impressive overall speedup. Still, choosing a well-localized block with minimum external dependencies is the only way to accomplish GPU port and present results shortly.

2. Consider the application block is ported. Then how to integrate it to the usual application development process, if core team has no much knowledge in GPU computations? How much support it would take to keep block updated? Without integration port becomes a branched case study.

This effort is dedicated to implementing hybrid CPU-GPGPU extendable Fortran compiler that can fully conserve original CPU source code, and therefore minimize implicit complications in user-side development process. Fully automatic compilation should not be confused with various existing directive-based approaches (PGI, HMPP), that still require too much hand-coding.

2   Strategy

No doubt, developing toolchain for automatic GPU kernels generation is a very complex long-term task. To move forward step-wise, the problem could be represented as a number of ring passes. On the first round we would need a rough mechanism for identifying computational loops and moving them into separate functions (host-device code split). Second, there should be a rough generator translating Fortran code into representation, that could be chained to existing GPU code compilers. Next we need previously implemented loop identification tool to insert calls to special runtime library functions that

would invoke kernels and pass their data dependencies between host and device memories. During passed rounds a collection of behavior tests should be collected for regressions checks. Next important group of rounds are optimizations. Toolchain needs to provide an approach for applying kernel code analysis and recompilation in runtime, taking in account additional information like actual compute grids sizes. With this mechanism implemented, a round of optimization passes development can start. According to known practices, the most important optimizations should be loops interchanging and tiling for threads and shared memory utilization. For better code coverage several specific limitations also need to be relaxed, such as inlining external calls in GPU kerenls. At this point complete functional prototype will be created, and development could switch back onto improving rough base components.

KernelGen, the compilation toolchain being developed, follows above plan and currently reached the step of implementing optimization passes. It means the tool already ports simple loops pretty well and can handle a large subsets of real source codes, falling back to host compiler in places where rough host-device split fails.

3 Code generator

The whole host and device code generation process is depicted on Fig. 1. Note it covers only static pipeline, when device code is compiled in binary form. KernelGen currenly also implements embedding kernel code IR for further compilation and optimization in runtime.

3.1 Kernels selection

To identify compute kernels and move them into separate compilation units, Fortran source code is marked up with XML into pretty-printed AST and transformed to target representation using a set of XSL transformations. XML markup is performed using g95-xml, a tool written by Philippe Marguinaud based on g95 compiler parser. Surprisingly, XSLT suited very well for fast development of rough implementation, thanks to its functional nature and builtin tree flow. However, this method is evil for human reading, quite slow on huge compilation units and suffers from various fails, such as g95 language

Original Fortran source code → Modules

Modules

Modules

g95-xml

LLVM IR → Optimized LLVM IR

host-part of Fortran code

device-part of Fortran code

C-backend

Compiling host Fortran code with target compiler

Compiling device C++ code with nvcc
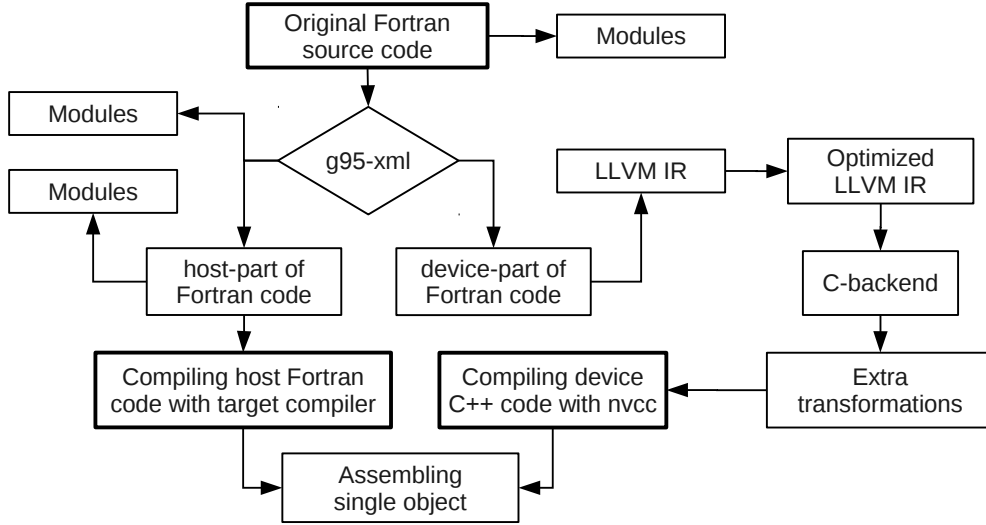
Extra transformations

Assembling single object

Figure 1: KernelGen static code generation pipeline

support issues or unhandled program logic cases in XSLT. The tool is known to provide good support for Fortran 90/95 and some features of Fortran 2003, like inline functions.

It is planned XML-based parser on some further step will likely be replaced by direct kernels generation out of LLVM IR. Although, currently it is not clear how to set up a split point with LLVM in a way that host code would still be Fortran, in order to utilize external Fortran compilers, other than LLVM over DragonEgg.

3.2   Backend

FF

3.3   Runtime vs compile-time generation

4   Runtime library

FF

4.1 Multithreading and MPI support

FF

5 Examples

FF

6 Special features

FF

7 Conclusions

FF

8 Acknowledgments

Generated code part	CPU	CUDA	OpenCL
Kernel in target language	x	x	x
Host side kernel init In C++	-	x	x
Host config and results checking in Fortran	x	x	x

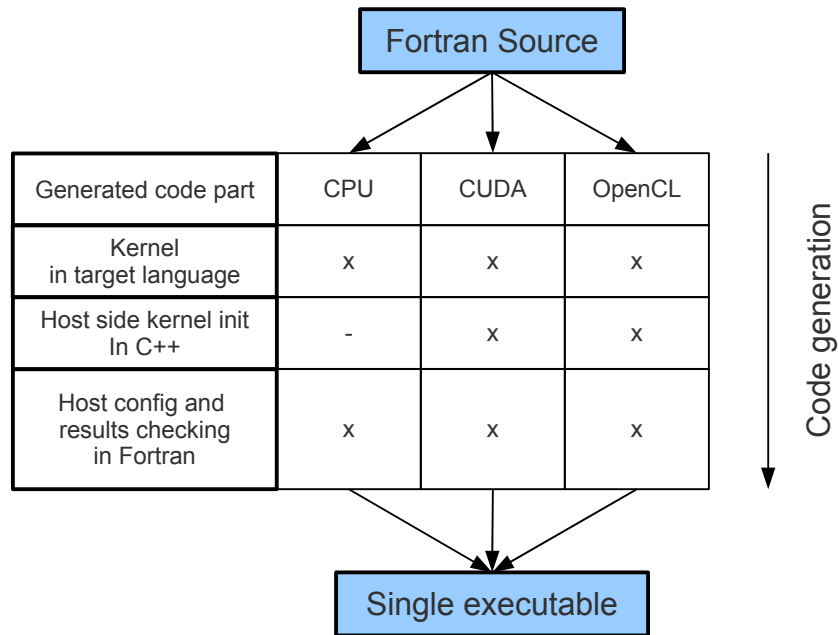Fortran Source → Single executable (Code generation)

Figure 2: Multiple targets support in a single executable.

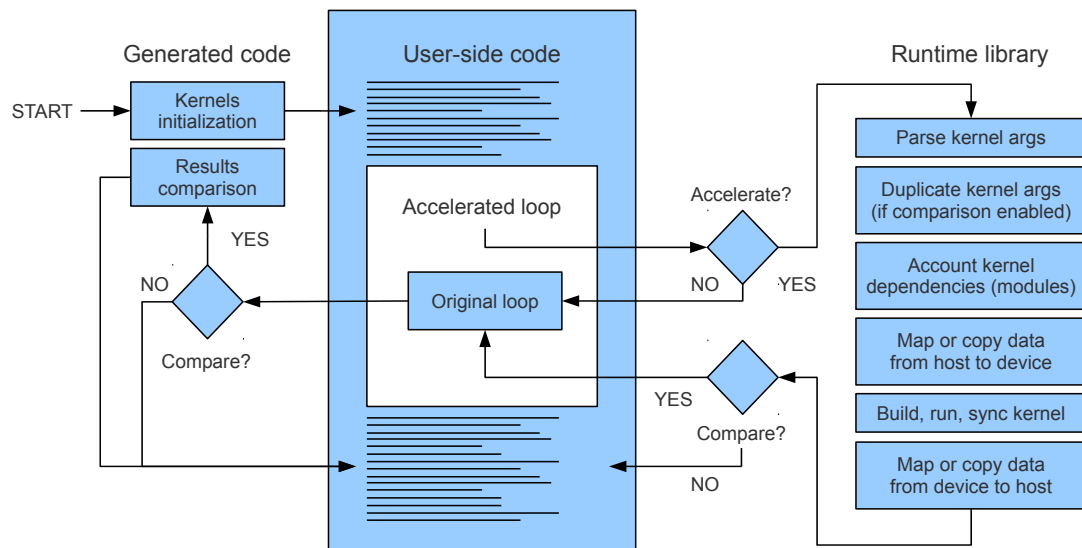on optimizations related to polyhedral analysis. Thanks to Christopher Bergström of Pathscale for discussions and criticism.

Figure 3: Execution model