

CERN Summer Student Report: Automatic translation from CUDA to C++

Student: Luca Atzori

Supervisors: Vincenzo Innocente, Felice Pantaleo

Introduction

This project consisted in the development of a tool for translating CUDA programs in C++.

CUDA is a parallel computing platform and application programming interface model created by NVIDIA. It allows software developers to use CUDA-enabled graphic processing unit (GPU) for general purpose processing.

The main goal of this work is to achieve performance portability, a major challenge faced today by the developers of the heterogeneous high performance computing community.

Our approach consists in writing the best possible code using the best possible frameworks and run it on the best possible hardware. This means that when we write our code, we don't think in advance about the legacy hardware on which our code will run, thus we can work with modern frameworks like CUDA.

Another fundamental aspect is that we want to discover if CUDA can be an effective data-parallel programming model for more than just GPU architectures.

1 Mapping

CUDA uses a data-parallel model of computation.

Compared to traditional CPUs, CUDA’s programming model takes advantage of the massively parallel nature of modern GPUs.

A CUDA program is composed by an host part, executed on a CPU, and a device part, executed on one or more GPUs. The host program launches one or more kernels, and the kernels are executed on the device concurrently by thousands of CUDA threads. CUDA threads are grouped into blocks, and blocks are grouped into a grid. Each thread has a unique local index in its block, and each block has a unique index in the grid. CUDA threads in a single block can synchronize and share data with threads in the same block. Blocks instead are asynchronous.

While the features of the model seem promising, the mapping of the computation to different architectures is not straightforward. The conceptually easiest implementation is to spawn a CPU thread for every CUDA thread specified in the programming model. However allowing CUDA threads within a block to execute on any available CPU core, mitigates the locality benefits and incurs in a large amount of scheduling overhead.

Therefore we should use a method of translating the CUDA program such that the mapping of the programming constructs maintains the locality expressed in the programming model also in the destination architecture. We also have to maintain the ordering semantics imposed by potential barrier synchronization points. Without modifying the operating system or architecture, this means that we must somehow manage the execution of the CUDA threads in the code explicitly.

Our approach consists in mapping each CUDA block to a CPU thread, and serialize the execution of each CUDA thread. The correspondence between CUDA elements and C++ language constructs is shown in Table 1

CUDA	C++	
block	std::thread / Task	asynchronous
thread	unrolled for loop	synchronous (barriers)

2 Techniques and tools

2.1 Source-to-source translation

From a more practical view, the translation will be done implementing a source-to-source translator, that takes the source code written in CUDA as its input and produces the equivalent source code in C++.

While a traditional compiler translator from a higher level programming language to a lower level programming language, we translate between programming languages that operate at approximately the same level of abstraction. This choice was made because we would like to analyze the obtained code more easily, in order to do further optimizations and debugging. Also, even if a traditional compiler approach would be a valid approach, we don't want to focus just on a certain target architecture, like x86 CPUs.

The key idea behind a source-to-source translation are the Abstract Syntax Trees, a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is abstract in not representing every detail appearing in the real syntax.

2.2 Clang

The tool on which we are relying to accomplish the syntactic analysis is Clang.

Clang is a compiler front end for the C/C++ languages family, and it uses LLVM as its back-end.

Clang's AST closely resembles both the written C++ code and the C++ standard, so this makes Clang's ASTs a good fit for refactoring tools. For example, parenthesis expressions and compile time constants are available in an unreduced form in the AST.

More importantly, Clang handles CUDA syntax.

3 Conclusion and future work

One of the most challenging aspects was obtaining the ASTs from the CUDA source code. It took a very long time just to study the feasibility of the problem, that finally was solved finding a way to include the CUDA headers in Clang. This allows us to match CUDA syntax very easily.

Another challenging aspect was the design of the software, both from the algorithmic point of view and the technical details concerning the translation of the CUDA constructs.

At the moment the kernel translation is completed but requires more intensive tests.

Some very preliminary experiments showed that the translated version slow-down, with respect to the native CUDA version, is acceptable. This is probably a symptom that the data parallel programming model of CUDA is effective also on other architectures.

Future work includes the management of the CUDA API in the host code, in particular the ones related to the memory management, and the atomic operations in the kernels. Also the insertion of vectorization compiler pragmas in the output source code is an aspect that needs to be explored.