Accelerating Python on Heterogeneous Architectures using Staged Parallelisation

Dejice Jacob, Jeremy Singer, and Phil Trinder

School of Computing Science, University of Glasgow

08/02/2019

Abstract

GPU accelerators are now commodity items, targeted by domain-specific languages, dialects of existing C-like languages like CUDA and OpenCL or their bindings in other languages. End-user programming is now commonplace, as non-expert developers employ dynamic scripting languages like Python to develop custom applications and workflows. This research project harnesses the synergy between these two popular trends, to create a parallelising framework for Python code. Rather than requiring end-user programmers to learn how to develop CUDA kernels, we use an intelligent analysis framework to synthesise accelerator code from undecorated *Plain Old Python* scripts. This allows everyday coders to gain direct benefit from compute accelerators.

We follow classical parallelisation techniques, identifying data dependences in loop nest structures using classical dependence analysis and vectorisation algorithms for known data types. However our approach is unlike traditional auto-parallelising compilation approaches, e.g. for Fortran, since Python presents unique challenges and opportunities.

Interpreted languages with dynamic typing allow us to partially check for dependences ahead of time, but to confirm that code can be parallelised, we need to wait till execution time to resolve concrete types. By partially deferring analysis to runtime, we can take advantage of the knowledge of loop-invariant constants or "unknown at compile-time" loop-limits to unearth parallelisation optimisation opportunities. At runtime, we specifically look for loop-dependence calculations that were deferred to runtime, calculate the dependences by plugging-in the run-time values and if required rebuild the dependence graph for vectorisation. This "mixed static and dynamic" analysis and compilation could potentially change the structure of the dependence graph and hence unlock greater parallelisation of the loop.

This is a staged approach to auto-parallelisation, relying heavily on the introspective capabilities of the Python runtime. We anticipate using a light-weight cost model that is dependent on the physical characteristics of the accelerator to calculate the cost/benefit tradeoff for inspecting, analysing and optimising code for a target accelerator like a GPU. We aim to use compilation techniques to generate kernels for nested Python loops at run-time, specialised for the loop dependences and types of data.

We parallelise numerical kernel code that is computed within nested loop-nests. To achieve the dependence analysis and to check for parallelisation opportunities, we restrict ourselves to multi-dimensional homogeneous arrays of basic types like *floats*, *int32* etc. Type inspection at runtime on the bindings will indicate whether the code can be compiled or has to fall back to the interpreter for execution.

To compile the auto-generated kernels targeting the GPU, we use Numba, an LLVM based compiler for Python. Numba enables the compilation of specific kernels by the use of Python decorators. This allows us to target specific kernels of code we can potentially accelerate and *jit* for execution. We aim to make execution transparent to the user by replacing all applicable loop-nests in the kernel with closures that have access to the relevant data-structures from the static compilation stage and returning a module to the user that redirects to the dynamic code analysis and generation framework.

In this talk, we report on current progress in extending the Python runtime to support heterogeneous targets, demonstrate potential use case scenarios and provide some preliminary results.