RPython's number crunching optimization

April 28, 2015

Abstract

1 Settings

Language: English
Duration: 30 min talk + Q/A
Audience level: Intermediate
Tags: PyPy, rpython, numpy, optimization, JIT

2 About me

I'm student at the technical university of Vienna currently finishing my master thesis. I have known Python for a while and it is one of my favorite languages.

Starting with my bachelor degree I specialized in compiler construction and computer languages in general which lead me to the PyPy project. I'm currently finishing my master thesis and I'm happy to share my experience with you!

I'm also very proud that I was accepted as a Google Summer of Code participant this year to implement the vectorization algorithm for PyPy/RPython.

3 RPython's number crunching optimization

The need for faster execution has driven CPU manufacturers to provide instructions to speed up Multimedia applications. Single Instruction Multiple Data (SIMD) is the terminology for the new CPU instruction set extensions that provided enhanced execution speed for numerical tasks. They are not only useful for multimedia applications but also for scientific applications. In theory, given a single precision floating point operation in a loop, if the loop is vectorizable on SSE2 (x86 ISA extension) the loop executes 4 times faster.

Currently the preferred way of optimizing numerical intensive applications is to write the critical routine in a low level language, compile it to the host computers target architecture and later invoke the routine in the language interpreter. SIMD extensions have been around for quite a while. Today's C/Fortran compiler optimization capabilities are astonishing but there is a catch: Using a JIT compiler these optimized SIMD routines are black boxes.

There are not many JIT compilers that use SIMD instructions to compute arithmetic on the packed items in parallel. Even less include this optimization in their production environment. Why don't we provide the means to let the JIT compiler utilize this opportunity to speedup the numerical kernels of NumPy? Why do only static ahead of time compiler vectorize their loops?

In this session I will present the new vectorizing algorithm and data structures that are implemented in the RPython backend. I will cover my journey on how to implement any optimization in the RPython backend, including any stumbling blocks I came along and potentially any newcomers might also encounter. The following questions and many more will be subject: Does it pay off to "just in time" vectorize your numerical application? Let's compare it to the conventional approach: What are the benefits?

4 Requirements / Good to know

Understanding how language interpreters and JIT compilers function. If you have attended any compiler lecture this should be sufficient to understand the presentation. It is an advantage (not required) if you understand how the RPython toolchain works.

5 Target audience

Everyone interested in compiler optimizations, code generation and utilizing SIMD instructions. Newcomers that are interested in writing their own optimization (for RPython interpreters). Newcomers that want to understand how the optimization in the RPython backend works.