Title: AD for an Array Language Supporting Nested Parallelism

Abstract:

Automatic differentiation (AD) is a practical way for computing derivatives of functions that are expressed as programs. AD has been recognized as one of the key pillars of the current machine learning (ML) revolution and has key applications in domains such as finance, computational fluid dynamics, atmospheric sciences, and engineering optimization.

This talk presents a solution for implementing reverse-mode AD for a high-level array language aimed at efficient GPU execution, in which programs are written as a nested composition of sequential loops and (explicitly) parallel constructs (e.g., map, reduce, scan, scatter).

In reverse-mode AD the original program is first executed to save all intermediate program values on a *tape*. The tape is subsequently used by the *return sweep* that computes in reverse program order the *adjoint* of each variable---i.e., the derivative of the result with respect to the given variable.

The talk is intended to provide a gentle introduction to AD, and then to highlight the two key ideas that lay the foundation of our solution:

First, parallel constructs can be differentiated at a high level, by specialized rules that are often more efficient in terms of both space and time than approaches that differentiate low level code.

Second, the tape can be eliminated by re-executing the original code of a scope whenever the return sweep enters that scope. This is important because (i) it is challenging to optimize the (spatial) locality of tape accesses in the context of GPU execution, (ii) the re-execution overhead can be optimized by known compiler transformations, e.g., flattening, and (iii) re-execution enables an important space-time trade-off that can be easily exploited by the user.

Finally, we present an experimental evaluation of ten relevant AI benchmarks that demonstrates that our approach is competitive with both state-of-the-art research solutions (Enzyme, Tapenade) and with popular tensor frameworks (PyTorch, JAX) in terms of both sequential-CPU and parallel-GPU execution.