Efficient audio signal processing using LLVM and Haskell

Henning Thielemann

2013-04-30
Thinking in terms of signal flow diagrams means thinking functional.

```
amplify (exponential halfLife amp) (oscillator Wave.saw phase freq)
```
Haskell and LLVM

Haskell

- strong type system
- purely functional
- lazy = stream processing
- efficiency is not primary

LLVM

- produces efficient code, especially vector instructions
- weak type system
- Just-In-Time compilation
  - transparent usage in Haskell
  - adaption to available vector instructions
Efficient audio signal processing using LLVM and Haskell

Embedded Domain Specific Language

```
amplify
  (exponential halfLife amp)
  (oscillator Wave.saw phase freq)
```

Direct interpretation:
- exponential and oscillator create infinite (lazy) lists of sample values
- amplify multiplies two lists element-wise

EDSL interpretation:
- exponential and oscillator provide LLVM IR code for generating values successively
- amplify appends the code provided by exponential and oscillator and multiplies their generated values
Embedded Domain Specific Language – Problems

Needed to solve more problems:

- sharing (→ causal arrows)
- feedback (→ causal arrows)
- cumbersome usage of arrows (→ functional interface)
- passing parameters to LLVM code (complicated by bug 8281)
- vector computing
- expensive computation of frequency filter parameters (→ opaque types)
Types of Vectorisation needed for Signal Processing

Given: Vectors of size $2^n$

- ideal speedup:
  $2^n$ scalar instructions $\rightarrow$ 1 vector instruction

- often speedup:
  $2^n$ scalar instructions $\rightarrow c \cdot n$ vector instructions

That is:

- Vectorisation not always optimization
- But: Assembling and disassembling vectors and conversion between different vector schemes also expensive
- Auto-vectorisation still possible?
Example: Cumulative Sum (cumsum)

Goal:

\[
\begin{align*}
    v_0 \rightarrow v_2 \\
    [a, b, c, d] \rightarrow [a, a + b, a + b + c, a + b + c + d]
\end{align*}
\]

Vectorisation:

\[
\begin{align*}
    v_0 \gg 1 & \quad +[a, b, c, d] \quad = [a, a + b, a + b + c, a + b + c + d] \\
    + v_0 & \quad = [a, a + b, b + c, c + d] \\
    = v_1 & \\

    v_1 \gg 2 & \quad +[a, a + b] \quad = [a, a + b, a + b + c, a + b + c + d] \\
    + v_1 & \quad = [a, a + b, b + c, c + d] \\
    = v_2 & \\
\end{align*}
\]

4 vector instructions instead of 3 scalar instructions
Where to do vectorisation in LLVM?

Different approaches:

- Program with vectors in Haskell, expand `cumsum` in Haskell (my current approach)
- Program with vectors in Haskell, expand `cumsum` in a custom LLVM pass (I’d prefer that)
- Program with scalars in Haskell, standard LLVM vectoriser detects `cumsum` (seems to be favorite of some LLVM developers)
Optimizations and JIT

- JIT compiles to host machine by default
- Optimizer does not optimize to host machine by default
  - Result: crashes
- I was told, I must set target data. Why?
  - And how, using the C interface?