Feldspar: Functional Embedded Language for DSP and Parallelism

Emil Axelsson

Advanced functional programming
2011-02-14

Motivation

- Signal processing for mobile communication gets increasingly demanding
- Old figures:
  - Expected bandwidth in 2014 is ***~10 times*** higher than in 2009
  - This requires ***~100 times*** more computational power
  - Compare to Moore's law: ***~10 times*** speed increase over the same period
- Industry needs to move to new, more parallel hardware architectures
  - Code portability a major concern
- Current signal processing code mostly written in C
  - Highly optimized for specific hardware – ***non-portable***
  - Not easily parallelized

Feldspar

- A joint project with Ericsson, Chalmers and ELTE University (Budapest)
- Aims to make digital signal processing (DSP) code more high-level
  - …to gain portability and maintainability
  - …ideally without sacrificing speed
- Method: Domain-specific language with associated compiler
- Initial application: Radio base stations for mobile communication
- Intension to be applicable for DSP in general

Hand-optimized loop (from AMR codec)

```
for (j = 0; j < L_frame; j++, p++, p1++)
{
   t0 = L_mac (t0, *p, *p1);
}
corr[-i] = t0;
```

ANSI-C specification

Hand-optimized loop (from AMR codec)

```
for (j = 0; j < L_frame; j++, p++, p1++)
{
   t0 = L_mac (t0, *p, *p1);
}
corr[-i] = t0;
```

ANSI-C specification

Equivalent loop optimized for specific processor

- Pragmas
- Intrinsic instructions
- Loop unrolling
- Etc.

```
#pragma MUST_ITERATE(80,160,80);
for (j = 0; j < L_frame; j++)
{
  pj_pj = _pack2 (p[j], p[j]);

  p0_p1 = _mem4_const(&p0[j+0]);
  prod0_prod1 = _smpy2 (pj_pj, p0_p1);
  t0 = _sadd (t0, _hi (prod0_prod1));
  t1 = _sadd (t1, _lo (prod0_prod1));

  p2_p3 = _mem4_const(&p0[j+2]);
  prod0_prod1 = _smpy2 (pj_pj, p2_p3);
  t2 = _sadd (t2, _hi (prod0_prod1));
  t3 = _sadd (t3, _lo (prod0_prod1));

  p4_p5 = _mem4_const(&p0[j+4]);
  prod0_prod1 = _smpy2 (pj_pj, p4_p5);
  t4 = _sadd (t4, _hi (prod0_prod1));
  t5 = _sadd (t5, _lo (prod0_prod1));

  p6_p7 = _mem4_const(&p0[j+6]);
  prod0_prod1 = _smpy2 (pj_pj, p6_p7);
  t6 = _sadd (t6, _hi (prod0_prod1));
  t7 = _sadd (t7, _lo (prod0_prod1));
}

corr[-i] = t0; corr[-i+1] = t1;
corr[-i+2] = t2; corr[-i+3] = t3;
corr[-i+4] = t4; corr[-i+5] = t5;
corr[-i+6] = t6; corr[-i+7] = t7;
```

Hand-optimized loop (from AMR codec)

```
for (j = 0; j < L_frame; j++, p++, p1++)
{
   t0 = L_mac (t0, *p, *p1);
}
corr[-i] = t0;
```

ANSI-C specification

Equivalent loop optimized for specific processor

- Pragmas
- Intrinsic instructions
- Loop unrolling
- Etc.

Probably not suited for different processor.
Non-portable!

```
#pragma MUST_ITERATE(80,160,80);
for (j = 0; j < L_frame; j++)
{
  pj_pj = _pack2 (p[j], p[j]);

  p0_p1 = _mem4_const(&p0[j+0]);
  prod0_prod1 = _smpy2 (pj_pj, p0_p1);
  t0 = _sadd (t0, _hi (prod0_prod1));
  t1 = _sadd (t1, _lo (prod0_prod1));

  p2_p3 = _mem4_const(&p0[j+2]);
  prod0_prod1 = _smpy2 (pj_pj, p2_p3);
  t2 = _sadd (t2, _hi (prod0_prod1));
  t3 = _sadd (t3, _lo (prod0_prod1));

  p4_p5 = _mem4_const(&p0[j+4]);
  prod0_prod1 = _smpy2 (pj_pj, p4_p5);
  t4 = _sadd (t4, _hi (prod0_prod1));
  t5 = _sadd (t5, _lo (prod0_prod1));

  p6_p7 = _mem4_const(&p0[j+6]);
  prod0_prod1 = _smpy2 (pj_pj, p6_p7);
  t6 = _sadd (t6, _hi (prod0_prod1));
  t7 = _sadd (t7, _lo (prod0_prod1));
}

corr[-i] = t0; corr[-i+1] = t1;
corr[-i+2] = t2; corr[-i+3] = t3;
corr[-i+4] = t4; corr[-i+5] = t5;
corr[-i+6] = t6; corr[-i+7] = t7;
```

Code generation

Optimized version could be generated from high-level algorithm + platform-specific annotations:

$$\sum p_i \cdot q_i$$

```
Data path width: 8;
Special instructions:
   _pack2,_smpy2,_sadd;
Must iterate: ...
Unroll: ...
```

```
#pragma MUST_ITERATE(80,160,80);
for (j = 0; j < L_frame; j++)
{
  pj_pj = _pack2 (p[j], p[j]);

  p0_p1 = _mem4_const(&p0[j+0]);
  prod0_prod1 = _smpy2 (pj_pj, p0_p1);
  t0 = _sadd (t0, _hi (prod0_prod1));
  t1 = _sadd (t1, _lo (prod0_prod1));

  p2_p3 = _mem4_const(&p0[j+2]);
  prod0_prod1 = _smpy2 (pj_pj, p2_p3);
  t2 = _sadd (t2, _hi (prod0_prod1));
  t3 = _sadd (t3, _lo (prod0_prod1));

  p4_p5 = _mem4_const(&p0[j+4]);
  prod0_prod1 = _smpy2 (pj_pj, p4_p5);
  t4 = _sadd (t4, _hi (prod0_prod1));
  t5 = _sadd (t5, _lo (prod0_prod1));

  p6_p7 = _mem4_const(&p0[j+6]);
  prod0_prod1 = _smpy2 (pj_pj, p6_p7);
  t6 = _sadd (t6, _hi (prod0_prod1));
  t7 = _sadd (t7, _lo (prod0_prod1));
}

corr[-i] = t0; corr[-i+1] = t1;
corr[-i+2] = t2; corr[-i+3] = t3;
corr[-i+4] = t4; corr[-i+5] = t5;
corr[-i+6] = t6; corr[-i+7] = t7;
```

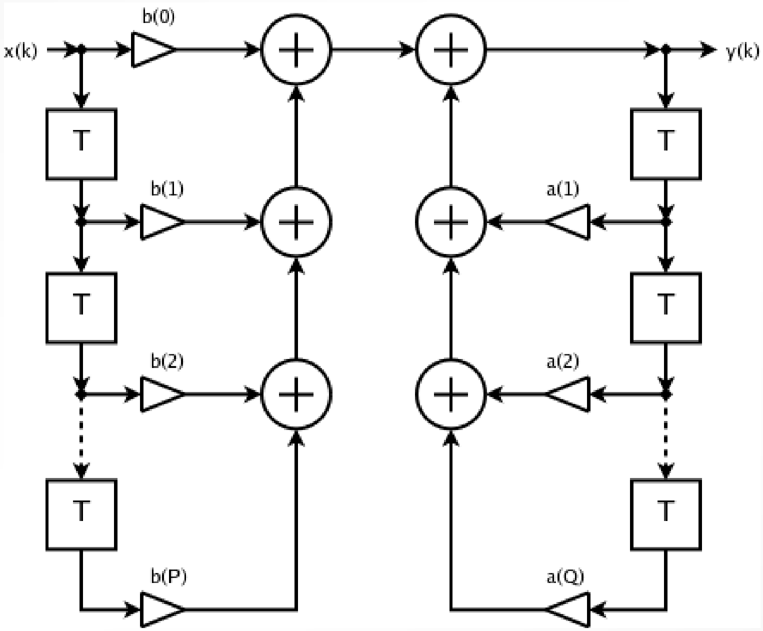
Code generation

- High-level language more readable and maintainable
- Flexible code generator makes code portable
- Platform-specific optimizations still needed, but should ideally be specified separately from algorithm

What kind of high-level language?

- DSP domain has well-established notation – calls for a domain-specific language (DSL)
- Existing DSP notation:

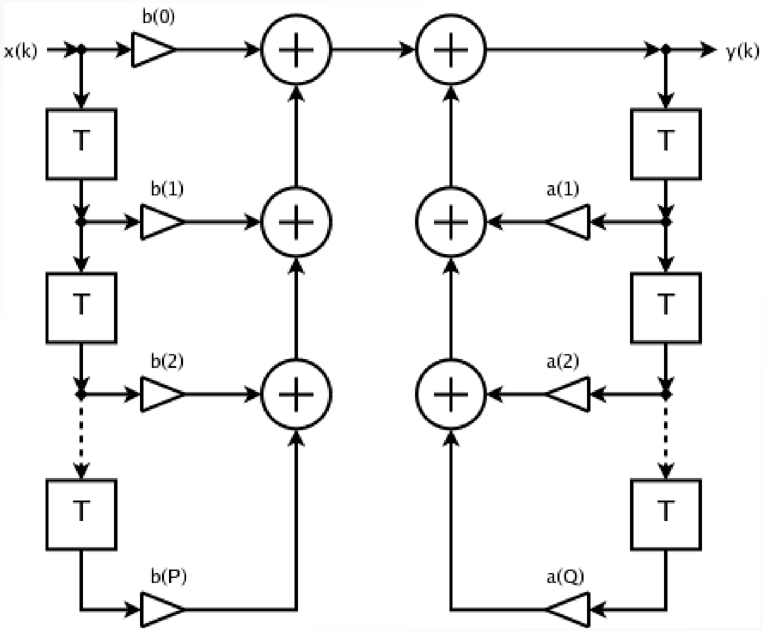
$$y[k] = \sum_{j=0}^{N} b_j x[k-j] - \sum_{p=1}^{M} a_p y[k-p]$$



What kind of high-level language?

- DSP domain has well-established notation – calls for a domain-specific language (DSL)
- Existing DSP notation:

$$y[k] = \sum_{j=0}^{N} b_j x[k-j] - \sum_{p=1}^{M} a_p y[k-p]$$



But also:

- Linear algebra
- Transforms
- Error detection/correction
- Encoding/decoding
- Etc.

Quite a broad domain

Feldspar – Functional Embedded Language for DSP and PARallelism

- High-level language:
  - Embedded in Haskell
  - Offers a functional programming style (higher-order functions, lists, etc.)
  - Domain specific constructs
  - Developed by Chalmers group
- Code generator developed by ELTE group

Feldspar – Functional Embedded Language for DSP and PARallelism

- “Feels” like Haskell:

```
square x = x*x

sumSq :: Data Index -> Data Index
sumSq n = sum (map square (1...n))
```

- Large part of Haskell's list library implemented in Feldspar
- Compare to standard Haskell:

```
square x = x*x

sumSq :: Int -> Int
sumSq n = sum (map square [1..n])
```

Generated C code (sumSq)

```
void sumSq(uint32_t in0, uint32_t * out1)
{
    uint32_t temp2;

    (* out1) = 0;
    {
        uint32_t i3;
        for(i3 = 0; i3 < ((in0 - 1) + 1); i3 += 1)
        {
            uint32_t v4;

            v4 = (i3 + 1);
            temp2 = ((* out1) + (v4 * v4));
            (* out1) = temp2;
        }
    }
}
```

Single for loop

No array allocation

Language structure

- Core language
  - Deeply embedded
  - Close to C but purely functional (explicit state)
  - Parallel arrays
  - Only static allocation, no recursion
  - Serves as input to the code generator
  - Low-level but flexible
  - Should not be used (much) by ordinary users
- Vector library
  - Data type for “virtual” vectors
  - No run-time representation
  - Interface similar to Haskell's list functions
  - Shallow implementation (does not require compiler support)
- Various other shallow high-level interfaces (extensible)

Implementation, Feldspar Light

- Code found on AFP course page
- deep-embedding library:
  - Back end: `Lambda.hs`
  - Front end: `Frontend.hs`
- Feldspar
  - Core language: `Feldspar.hs`
  - Vector library: `Vector.hs`

Real Feldspar vs. Feldspar Light

- Core language
  - Much more functions/types
  - Much more optimizations
- Vector library quite similar
  - Has a notion of “segments” to generate better code for concatenated vectors

Summary

- Embedded functional language seems promising for high-level, efficient DSP
  - Missing larger case studies...
- Low-level core language a success
  - Combines advantages of deep and shallow embedding
  - Good interface between Chalmers and ELTE groups
- Ongoing/future work:
  - Add “control layer”
  - Multi-core deployment
  - ...
- Open source:
  - http://hackage.haskell.org/package/feldspar-language
  - http://hackage.haskell.org/package/feldspar-compiler