Google

Fixed Point Arithmetic

Replacing Soft Floats in the Pixel Buds ⚡

leonardchan@

August 2024

01

Problem Statement

Soft floats in Pixel Buds are expensive

Floats on the Pixel Buds

The Pixel Buds use the **Arm Cortex-M0+** processor, an energy-efficient Arm processor available for constrained embedded applications

It doesn't have an FPU, so many floating point operations targeting this CPU depend on software emulating floating point ops (“soft floats”), usually via lib calls (`__floatdisf`, `__aeabi_fmul`, `__aeabi_fcmpgt`, ...)

Soft floats can be pretty expensive when we need to work with decimal/fractional numbers

What do we use floats for?

Pixel Buds code running on this processor runs different algorithms for classifying touch types and determining on-head device (OHD) state

“Is this a short touch, long touch, single/double/triple tap, swipe up/down, ...?”
“Is the device on-head or off-head?”

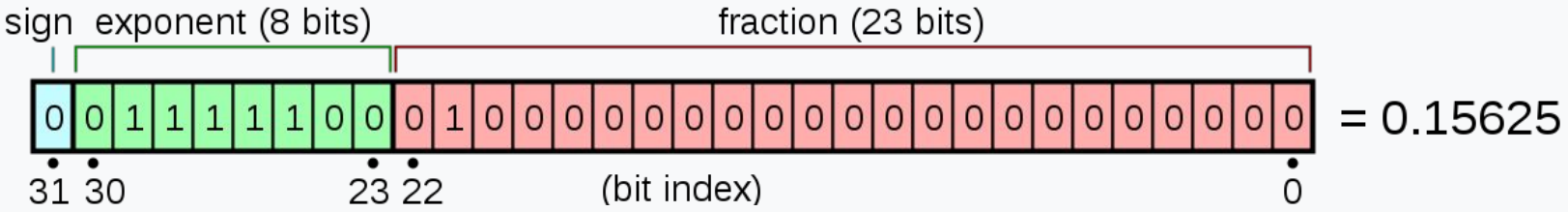
Algorithms include

- Logistic regression classifier
- Binary logistic regression classifier
- Softmax
- IIR Filter
- Centroid estimation
- Touch scaling

Each of these involves doing a lot of floating point ops which can be pretty slow

Why are soft floats expensive?

This is largely a consequence of how we represent them



This particular representation allows for greater range and precision at the cost of large/expensive operations

Addition involves *a lot* of steps

`__aeabi_fadd` in `libgcc.a` for this target is **444 bytes**

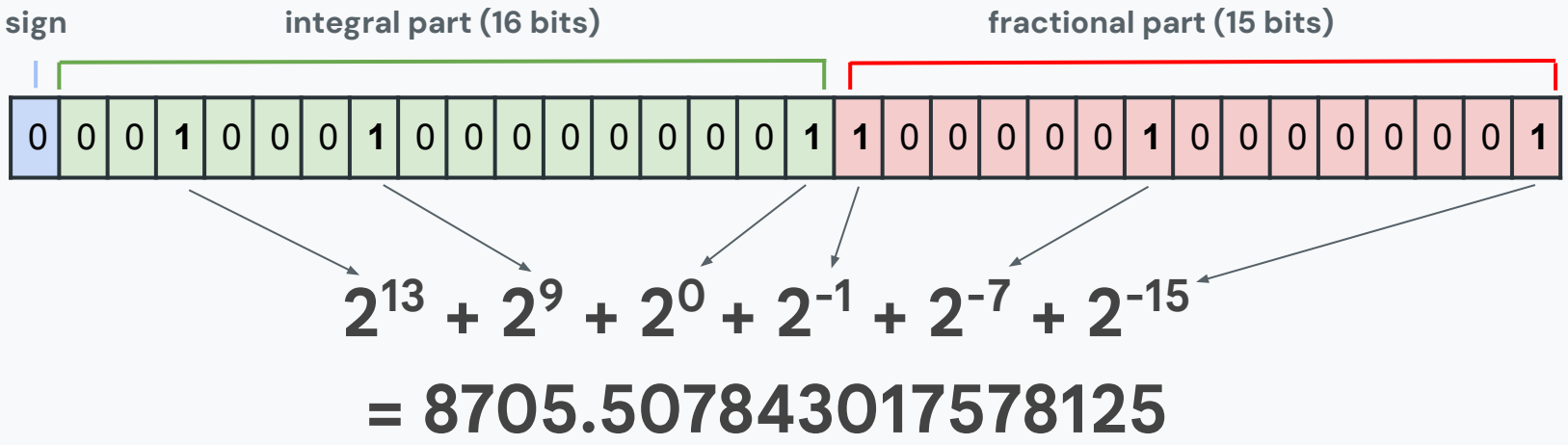
02

Fixed Point Arithmetic

For representing fractional numbers

Fixed Point Type Example Layout

	Sign bit	Integral bits	Fractional bits	Bit width
`signed _Accum`	1	16	15	32



ISO TR 18037: Fixed Point Types

This extension adds a bunch of new fixed point types into C:

```
unsigned short _Fract       unsigned short _Accum
unsigned _Fract             unsigned _Accum
unsigned long _Fract        unsigned long _Accum
signed short _Fract         signed short _Accum
signed _Fract               signed _Accum
signed long _Fract          signed long _Accum
```

Types prefixed with `_Sat` also default to saturation instructions.

Each of these types has a *unique number of integral and fractional bits* associated with it

Fixed vs Float Representations

Floating Point	Fixed Point
Specified by IEEE 754	Specified by ISO TR 18037
Hardware can supply native floating point ops (“hard floats”) Or emulate with library calls (“soft floats”)	Hardware can supply native fixed point ops Just integers under the hood
Range of **+/-inf**	Limited by integral and fractional bits
Very Small Precision (**10**$^{-38}$ for standard `float`)	Limited by the fractional bits (1/2^scale)
Hard floats - Typically fast *Soft floats* - Can be very slow/large	Typically fast Usually involves normal integral binary ops with some shifts
Always within 0.5 ULP of true result	Always within 1 ULP of the true result
May not always produce the same result for a given op due to rounding errors	Will always produce the same result for a given op

03

Implementation

Getting Fixed Point on Pixel Buds

The Plan

Clang/LLVM

- Ensure the compiler supports all features needed

libc

- Ensure llvm-libc provides all necessary functions

Pixel Buds

- Replace all the floats
- Benchmark, compare, and assert correctness

Fixed Point in IR - Design Decisions

How do we want to represent fixed point types under the hood?

Functionality

- We need to at least be able to support everything in ISO 18037

Complexity

- How much work will this take to implement?

Usability

- Can other frontends/languages take advantage of fixed point types also?

Optimizations

- Can the representation take advantage of existing opts?

Ease of lowering for backends

- If a backend target has native fixed point support, can the representation be lowered to take advantage of fixed point instructions?

Implement as integers

Fixed point types are essentially just scaled integers, so we can just reuse existing IR integers. More complex operations can be implemented via intrinsics.

```
_Accum result = a * b;
%result = call i32 @llvm.smul.fix.i32(i32 %a, i32 %b, i32 15)
```

Pros

- Can re-use many existing IR instructions
- Can take advantage of existing integer opts
- Easiest to implement
- Backends that support native fixed point types can lower the intrinsics to those instructions

Cons

- Unable to tell at the IR level if an int is actually a fixed point type
- Would need to implement a lot of intrinsics

Fixed Point Intrinsics

	Signed	Unsigned
Multiplication	`llvm.smul.fix.*` `llvm.smul.fix.sat.*`	`llvm.umul.fix.*` `llvm.umul.fix.sat.*`
Division	`llvm.sdiv.fix.*` `llvm.sdiv.fix.sat*`	`llvm.udiv.fix.*` `llvm.udiv.fix.sat.*`
Addition	`llvm.sadd.sat.*`	`llvm.uadd.sat.*`
Subtraction	`llvm.ssub.sat.*`	`llvm.usub.sat.*`

Addition and **Subtraction** intrinsics aren't limited to fixed point types

Clang - Frontend

- Taught Clang about all the fixed point types
  - Enabled via `-ffixed-point`
- Helper classes
  - APFixedPoint
- Precision macros
- Added all supported operations
- Semantic analysis
  - Largely conversion/casting checks
  - Updating `-Wformat` warnings for new specifiers
- DWARF Debug info
  - `DW_ATE_signed_fixed`
  - `DW_ATE_unsigned_fixed`

C++ Support

ISO 18037 is a C extension and provides no C++ support, but all C++ types need to have an encoding for name mangling

```
int func(float)    -> _Z4funcf
int func(int)      -> _Z4funci
```

Proposed and added encodings for each fixed point type in the Itanium C++ ABI

```
int func(_Accum)         -> _Z4funcDAi
int func(_Sat _Fract)    -> _Z4funcDSDRi
```

C Functions to add in llvm-libc

Need fixed point equivalents for

`sqrtf` $\sqrt{x}$

`expf` e^x

`roundf`

`printf` formatting

ISO 18037 defines many more, but we only implemented the ones used by Hearables due to limited time and resources

Pixel Buds - Replacing all the floats

A lot of instances, but not as much as expected (**~130**)

- No doubles
- Doesn't include 3p code
- Includes literals with `f` suffix

Precision:

- Tests and tuning constants use numbers on the order of **0.001**
- Some constants used for tuning have about **18** significant figures, but that much precision didn't seem needed

Range:

- Most tests use a range of `[-91.2, 12.3]`

`_Accum` works for most cases

04

Results

They’re pretty good

Size: 1.25KiB saved

```
   FILE SIZE        VM SIZE
--------------  --------------
 +0.5% +3.06Ki  [ = ]       0    .debug_line
 +0.1% +1.89Ki  [ = ]       0    .debug_str
 +0.2%    +496  [ = ]       0    .debug_abbrev
 +0.6%    +272  +0.6%    +272    .bss
 -0.3%     -16  -0.3%     -16    .data
 -0.3%    -232  [ = ]       0    .debug_frame
 -1.3%    -712  [ = ]       0    .debug_ranges
 -0.3% -1.11Ki  [ = ]       0    .debug_loc
 -1.2% -1.50Ki  -1.2% -1.50Ki    .text
 -1.6% -1.61Ki  [ = ]       0    .symtab
 -3.2% -3.06Ki  [ = ]       0    .pw_tokenizer.entries
 -1.2% -4.00Ki  [ = ]       0    .strtab
 -0.4% -19.6Ki  [ = ]       0    .debug_info
 -0.3% -26.1Ki  -0.7% -1.25Ki    TOTAL
```

Size: The Math Functions

No longer need to link in various floating point lib functions (`__floatdisf`, `__aeabi_fmul`, `__aeabi_fcmpgt`, ...)

The fixed point libc functions are much smaller and use functions that are already linked in.

`sqrtf` - makes calls to `__ieee754_sqrtf`, `__aeabi_fcmpun`, `__aeabi_fcmplt`, `__errno`, and `__aeabi_fdiv`

`isqrt_fast` - makes only calls to `__clzsi2`, `__aeabi_lmul`, and `__aeabi_uidiv`

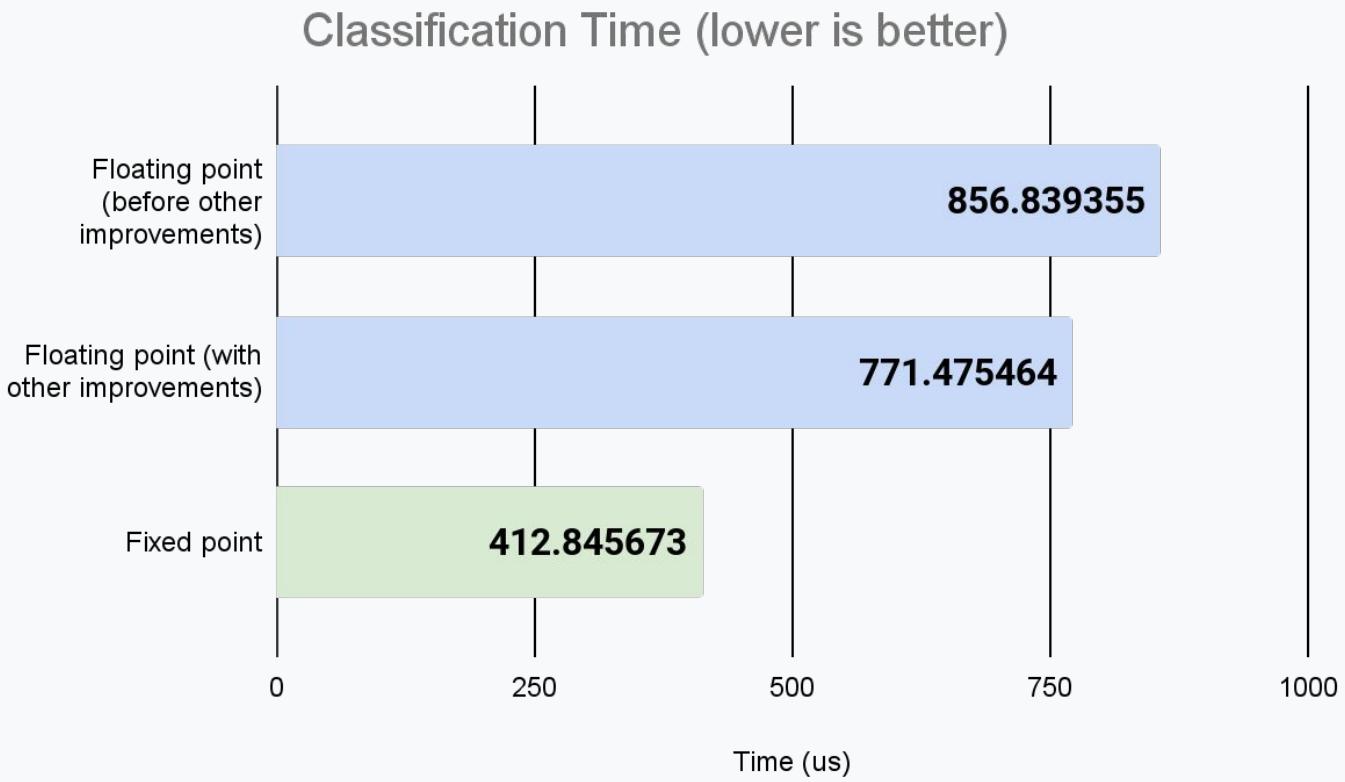
Size: Binary Operations

Note that individual binops are slightly larger since they involve a bunch of shifts or saturation checks. Floats tend to make a call to the same library function, but fixed point types emit instructions at every callsite.

```
float_add:
  push    {r7, lr}
  add     r7, sp, #0
  bl      __aeabi_fadd
  pop     {r7, pc}
```

```
sat_fixed_add:
  adds    r0, r0, r1
  bvc     .LBB1_2
  asrs    r1, r0, #31
  movs    r0, #1
  lsls    r0, r0, #31
  eors    r0, r1
.LBB1_2:
  bx      lr
```

Speed: ~2x improvement



Correctness

All of the 9336 gesture classification tests produce the exact same results as if using floats.

All of the 6058 OHD classification tests produce the exact same results as if using floats.

05

Future Improvements

Future Improvements

Support in other languages: Zig, Rust

Formal PW_LOG support format specifier support

See if we can use non-saturating types to reclaim some instructions + `.text` space (assuming we don't ever overflow)

Using fixed-point in other products and contexts (like replacing hard floats)

Including fixed point in some future version of C rather than it being an extension

C++ standard library support (like `std::format`)

Thanks to these people

Hearables: Max Koopman, Merrick McCracken

llvm-libc: Michael Jones, Tue Ly

Toolchain: Petr Hosek, Prabhu Karthikeyan Rajasekaran

Pigweed: Keir Mierle

Ericson: Bevin Hansson

Summary

1.25KiB saved
~2x speed improvement
Identical behavior

Use fixed point types in Clang with `-ffixed-point`!