Kalray's MPPA: Mathematical library and low level arithmetic optimizations

Kalray training at CERN, June 3^{rd}, Nicolas Brunie

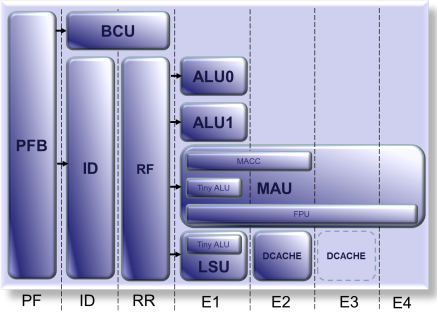
The objectives of this training are:

- Show you Kalray core arithmetic capabilities
- Teach you how to use basic math library on Kalray processor
- Teach you how to use advance function on K1
- Teach you how to write low-level optimized code

KALRAY

Overview of arithmetic on K1

K1 core implements a 5-issue VLIW



- 1 FP/MAU issue
- 4 32-bit ALU issues
- Between 1 and 4 cycles
- Bypasses
- 64-bit Load/Store

K1's Integer arithmetic

- One 64-bit ALU (ADD, SUB, SHIFT, ...)
- Four 32-bit ALU
  - Two full capabilities (ADD, SUB, SHIFT ..)
  - Two Reduced capabilities (ADD, SUB, LOGICAL)
- One 64-bit MAU: signed, unsigned, large accumulator
- Fixed-Point capabilities
- Operations with carry

K1's FPU Overview

- 4-stage main pipeline
- IEEE-754 compliant
- Extended capabilities (FMAWD, FDMA)
- Mixed-Precision

Operations	latency	throughput
fp32 FADD, FSUB, FMUL	4	1
$fp32 \to fp64$ conversions	4	1
fp32 FMA	4	1
fp64 FADD, FSUB	4	1
fp64 FMUL	5	2
FMAWD, FDMA	4	1

Original floating-point operations

Mixed Precision Fused Multiply-Add

- Computes $a \times b + c$ with a and b $fp32$ and c $fp64$
- Single rounding towards $fp64$
- FFAMWD, FFMSWD, FFMANWD, FFMSNWD instructions

Dual Fused Multiply-Add

- Computes $a \times c + b \times d$, with a, b, c and d $fp32$
- Single rounding towards $fp32$ or $fp64$
- FDMA, FDMS, FCMA, FCMS instructions

Floating Point Miscellaneous

FP operations in K1's ALU:

- Sign-based operations (abs, neg)
- Square root and Division seed
- $fp64 \rightarrow fp32$ conversions

Rounding modes and exceptions:

- 4 binary fp rounding mode supported
- 5 exceptions
- Default exception handling

Overview of mathematical library

Accesscore provides GCC and libm:

- GCC targets most of the operation introduced in Section 2
- GCC is delivered with libgcc (e.g. divsf3, divdf3)

External library:

- Newlib's libm
- Static library
- Compliant with C standard
- Implements the `math.h` API
- Usual function: exp, cosf, rint...

A few optimized implementations

Kalray's capabilities allow for efficient implementation

- FMA, FDMA
- Integrated conversions
- Pipelined FPUs

Current state:

- divsf3 and sqrtf
- More to come: priority driven by customer request

Pre-requesites: Kalray tools

- Build and link with **k1-gcc**
- Build with **make run_test TEST=test_name**
- Simulate executable with **k1-cluster**
  - Use `--cycle-based` to obtain better timing accuracy
  - Use `--profile` to generate execution traces
- Run on hardware with **k1-jtag-runner**
  - with option `--exec-file=C0:<executable>`
- Modify sources and Makefile, ask questions

Pre-requesites: timer measures

Before optimizing code, we need a metric: timing.
How to determine code execution time ?

- Traces can be used
- Performance monitors are more accurate

K1 performance monitoring support:

- Each K1 provides two performance monitors: PM0 and PM1
- Set them to count cycle using
  `__k1_counter_enable(cindex, _K1_CYCLE_COUNT, 0)`
- Retrieve current monitor value with
  `__k1_counter_num(cindex)`

Quick and Dirty complex multiplication

- FDMA and FCMA can be used to accelerate complex multiplication

```
__builtin_k1_fdma(a, b, c, d) = a * c + b * d
__builtin_k1_fdms(a, b, c, d) = a * c - b * d
__builtin_k1_fcma(a, b, c, d) = a * d + b * c
__builtin_k1_fcms(a, b, c, d) = b * c - a * d
```

Exercise: complex_product_empty

- Build and Run
- Open the source file
- Complete the implementation of `complex_mult_array_opt` Using `__builtin_k1_fdma, fdms, fcma, fcms`
- (Bonus) Develop assembly version of the function

Rounding modes and exceptions

- API can be found in:
  k1-elf/include/HAL/machine/core/common/cpu.h
- Provides R/W capabilities to **Compute Status** register fields
- Impact hardware operations (not libm)

Exercise: rnd_and_exceptions

- Build and Run
- Open and Modify sources
- Try to find simulator bugs (or at least generate a minus 0)

Rounding mode and mathematical function:

- Compute Status impacts optimized routines
- It does not impact most of the legacy functions

Using GCC built-in arithmetic support

Exercise: example_libgcc_empty

- Determine the options required to link with libgcc
- Build and run the example
- Open the source code
- Explain the timing differences

Using K1's libm

- Delivered with every accesscore
- Linked through k1-gcc, with **-lm** option

Exercise: example_libm_empty

- Try to build the example with **k1-gcc**
- Fix the problems which arise
- Build and run the example

Assembly development

For the next parts of this training, we will use low-level programming to optimize our programs and manipulate K1 arithmetic operations:

- Disassemble using **k1-objdump -D**
- Assenble using directly **k1-gcc**

- File is divided into section (.text, .data)
- GNU-asm like assembly syntax: **[op] [result] = [operand list]**
- Instruction bundles separated by **";;"**

Low-level exercise

Exercise: look at K1 assembly

- Dissasemble *build/example_libgcc_empty*
- Inspect the disassembled code, find the **main** function
- Build it once again but using **-S** options with **k1-gcc**
- Inspect the generated assemby code and find the call to division

What you need to know

To implement a function in assembly: You need to respect the calling convention:

- argument passing and result return interfaces
- callee and caller-saved registers
- stack and frame registers

Exercise: Observing the calling convention

- Let us have an other look at example_libgcc_assembly
- Find function calls
- Observe manifestation of the calling convention

Our goal is not to give you a full overview, but feel free to ask questions.

KALRAY

Half-Packed operations

K1's ALU and MAU implements 16-bit SIMD operations

- Add, Subtract, Multiply-Accumulate
- Compiler will select them (sometimes)

Exercise: compute_packed_array

- Compile with `k1-gcc -O3 -mcore=k1dp`
- Objdump with `k1-objdump -D`
- Look at the generated code for `compute_add_packed_array` and `compute_mac_packed_array`
- What part(s) implement the arithmetic computation ?

Optimizing using half-packed

Exercise: short_add

- Compile using k1-gcc
- Open short_add_opt_empty.S
- Finish the implementation of `short_add_opt`
- Compile, fix and compare

Operations with carry

- K1 ISA provides instruction for arithmetic with carry
- Those operations can be used to accelerate multi-precision computation

Exercise: op_with_carry

- Compile and Run
- Open large_addition_opt_empty.S
- Fill the gaps, Build, Run and Compare

KALRAY

Introduction to metalibm

- Kalray is involved in the Metalibm Project
- Metalibm is generator of mathematical functions
- Tuned for specific architecture
- Our current (on going) work is to optimize our libm

Using metalibm generated functions

Function generated from a private fork
Public software available at metalibm.org

- Metalibm aims at generating both libm and custom functions
- with hand-written level performances ...
- ... and much more flexibility

Exercise function_bench

- Open src/metalibm/bench/function_bench.c
- Build and Run
- Enable metalibm generated implementations, run once again
- Open function source files

The end.

Any questions ?

Kalray's OpenCL and libm

This training requires AccessCore 2.0

- Include `<math.h>` into kernels
- Add `-lm` to kernel build options
- Define macros to circumvent OpenCL-C missing features