HCQC - HPC compiler quality checker

Masaki Arai masaki.arai@linaro.org

LEG HPC-SIG

Background and Purpose

- The quality of the kernel part is important in HPC applications(number crunch on supercomputer).
- Make it easy to check the quality of compiler optimizations and acquire data to improve them

HCQC : HPC compiler quality checker

Subject of Quality Check

- Configuration file defines the subject of quality check.
- Main items:
  - Compiler
  - Compiler version
  - Optimization flags

```
{
   “DISTRIBUTION" : "OpenSUSE Tumbleweed",
   "ARCH" : “aarch64",
   "CPU" : "AMD Opteron A1100 Cortex A57",
   "LANGUAGE" : "C",
   "COMPILER" : “GCC",
   "COMMAND" : "/usr/bin/gcc",
   "VERSION" : “7.1.1",
   "OPT_FLAGS" : ["-O2"],
   "ASM_FLAGS" : ["-S“,  “-fverbose-asm”],
   “FLAG_DB”  :  [[“?DEBUG_FLAG",  “-g”],
                  [“?C99_STANDARD",  “-std=c99”]]
}
```

Example of configuration file

Metrics for Quality Evaluation

- HCQC has the following metrics:
  - op

    # of mnemonics in an assembly code
  - kind

    The kind of mnemonics in an assembly code(memory, branch, other)
  - regalloc

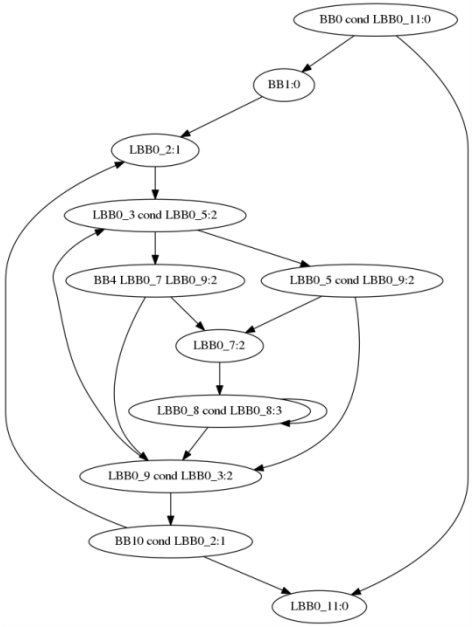
    The quality of register allocation(# of spill in/out instructions)
  - ilp

    Instruction level parallelism by instruction scheduler
  - vectorize

    Vectrization/SIMDization situation
  - swpl

    # of initiation interval by software pipelining

Investigation Result



DISTRIBUTION : OpenSUSE Tumbleweed
ARCH : aarch64
CPU : AMD Opteron A1100 Cortex A57
LANGUAGE : C
COMPILER : ClangLLVM
COMMAND : /usr/bin/clang
VERSION :4.0.1
OPT_FLAGS : -O2
TEST_PROGRAM: sample
KERNEL=FUNCTION=NAME : kernel
DATE: 2017/11/07

CFG	DEPTH	kind			regalloc		ilp	swpl	vectorize			
		memory	branch	other	spill in	spill out	IPC	II	kind	mem	arith	other
BB0 cond LBB0_11	0	3	1	3	0	0	0.5			0	0	0
BB1	0	0	0	4	0	0	0.5			0	0	0
LBB0_2	1	3	0	5	0	1	0.7			0	0	0
LBB0_3 cond LBB0_5	2	2	1	1	2	0	0.9		SLP	0	1	0
BB4 LBB0_7 LBB0_9	2	1	2	5	0	0	0.9		SLP	0	1	1
LBB0_5 cond LBB0_9	2	3	1	5	0	0	1.3		SLP	0	1	1
LBB0_7	2	3	0	5	0	3	1.7		SLP	0	1	2
LBB0_8 cond LBB0_8	3	7	1	5	0	0	2.5	5	LOOP,SLP	2	2	2
LBB0_9 cond LBB0_3	2	2	1	3	0	2	1.3		SLP	0	1	1
BB10 cond LBB0_2	1	1	1	2	0	0	0.8			0	0	0
LBB0_11	0	0	0	1	0	0	0.2			0	0	0
SUMMARY		25	8	39	2	6				2	7	7

Quality Evaluation by Comparison

- One investigation result has little meaning.
- Typical comparison examples:
  - GCC vs. LLVM(on AArch64)
  - LLVM 4.0.0 vs. LLVM 5.0.0(on AArch64)
  - LLVM with –O2 vs. LLVM with –O3(on AArch64)
  - LLVM on AArch64 vs. LLVM on x86_64

    Missing optimizations on AArch64
  - LLVM on AArch64 vs. ICC on x86_84

    Optimization hints for SVE from AVX codes

Example of Comparison(1)

- HimenoBMT-static(regalloc)
  - ✓ GCC is better than Clang/LLVM.

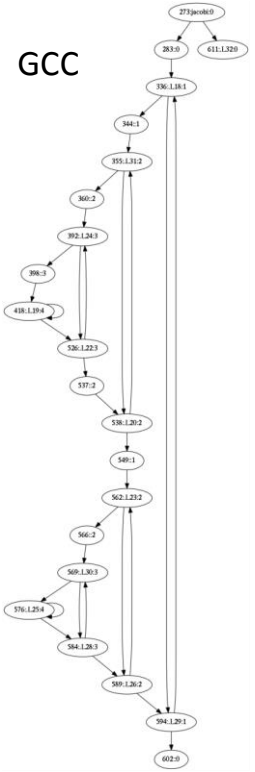
GCC

CFG	DEPTH	spill in	spill out
jacobi cond .L32	0	0	1
	0	0	8
.L18 cond .L29	1	1	0
	1	0	1
.L31 cond .L20	2	1	0
	2	3	1
.L24 cond .L22	3	0	0
	3	1	0
.L19 cond .L19	4	0	0
.L22 cond .L24	3	0	0
	2	1	0
.L20 cond .L31	2	2	1
	1	4	0
.L23 cond .L26	2	0	0
	2	0	0
.L30 cond .L28	3	0	0
.L25 cond .L25	4	0	0
.L28 cond .L30	3	0	0
.L26 cond .L23	2	0	0
.L29 cond .L18	1	2	1
end	0	0	0
.L32 end	0	0	0
SUMMARY	–	15	13

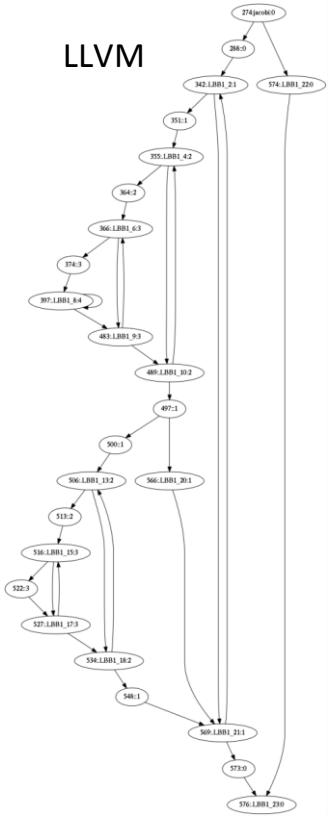
LLVM

CFG	DEPTH	spill in	spill out
jacobi cond .LBB1_22	0	0	7
	0	0	6
.LBB1_2 cond .LBB1_21	1	0	0
	1	0	1
.LBB1_4 cond .LBB1_10	2	1	1
	2	0	0
.LBB1_6 cond .LBB1_9	3	1	0
	3	0	0
.LBB1_8 cond .LBB1_8	4	0	0
.LBB1_9 cond .LBB1_6	3	1	0
.LBB1_10 cond .LBB1_4	2	2	0
cond .LBB1_20	1	1	0
	1	0	2
.LBB1_13 cond .LBB1_18	2	1	1
	2	2	0
.LBB1_15 cond .LBB1_17	3	1	0
	3	1	0
.LBB1_17 cond .LBB1_15	3	1	0
.LBB1_18 cond .LBB1_13	2	4	2
goto .LBB1_21	1	2	0
.LBB1_20	1	2	0
.LBB1_21 cond .LBB1_2	1	0	0
goto .LBB1_23	0	0	0
.LBB1_22	0	0	0
.LBB1_23 end	0	7	0
SUMMARY	–	27	20

GCC



LLVM



Example of Comparison(2)

- hpcg-3.0(op)
  - GCC generates `fmadd' and `fmsub' but Clang/LLVM dosesn't. (Need `-ffp-contract=fast')

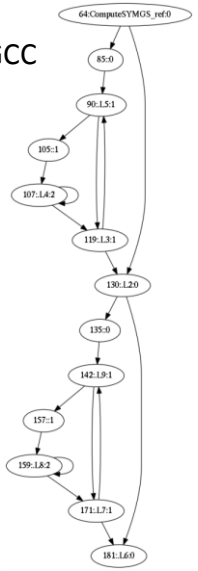
GCC

CFG	DEPTH			fdiv	fmadd	fmsub			
ComputeSYMGS_ref cond .L2	0			0	0	0			
	0			0	0	0			
.L5 cond .L3	1			0	0	0			
	1			0	0	0			
.L4 cond .L4	2			0	0	1			
.L3 cond .L5	1			1	1	0			
.L2 cond .L6	0			0	0	0			
	0			0	0	0			
.L9 cond .L7	1			0	0	0			
	1			0	0	0			
.L8 cond .L8	2			0	0	1			
.L7 cond .L9	1			1	1	0			
.L6 end	0			0	0	0			
SUMMARY	–			2	2	2			

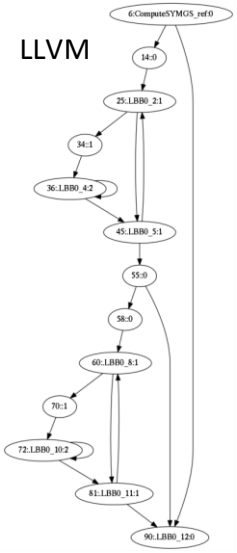
LLVM

CFG	DEPTH		fadd	fdiv			fmul	fsub	
ComputeSYMGS_ref cond .LBB0_12	0		0	0			0	0	
	0		0	0			0	0	
.LBB0_2 cond .LBB0_5	1		0	0			0	0	
	1		0	0			0	0	
.LBB0_4 cond .LBB0_4	2		0	0			1	1	
.LBB0_5 cond .LBB0_2	1		1	1			1	0	
cond .LBB0_12	0		0	0			0	0	
	0		0	0			0	0	
.LBB0_8 cond .LBB0_11	1		0	0			0	0	
	1		0	0			0	0	
.LBB0_10 cond .LBB0_10	2		0	0			1	1	
.LBB0_11 cond .LBB0_8	1		1	1			1	0	
.LBB0_12 end	0		0	0			0	0	
SUMMARY	–		2	2			4	2	

GCC



LLVM



Workflow of HCQC

① Compile one test program

② Run the executable file and verify its result by comparing output and answer data

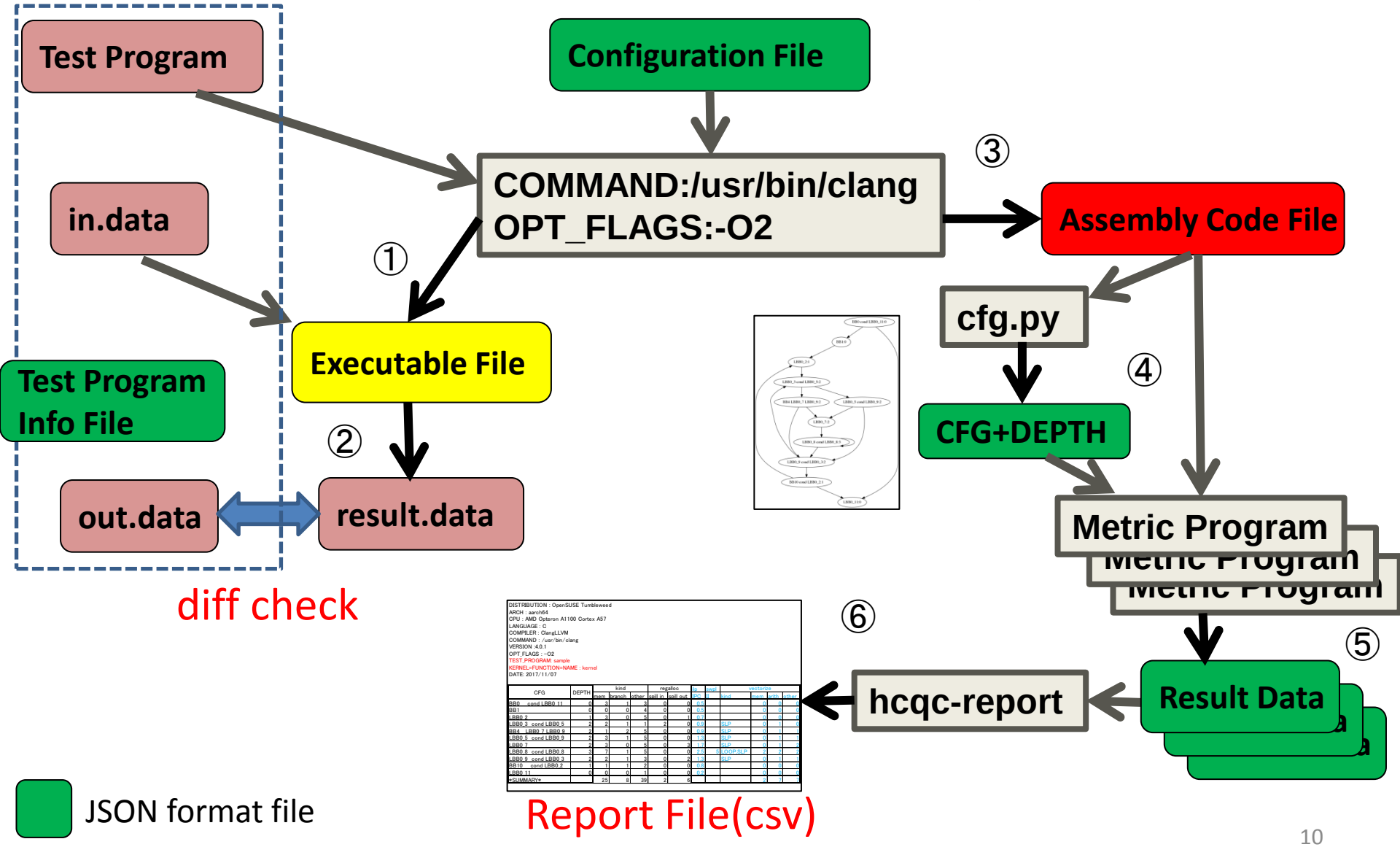
③ Generate the assembly code file

④ Make the control flow graph of the kernel part from the assembly code

⑤ Get result data using metric programs

⑥ Make the report file from data

Workflow of HCQC



Test Programs for HCQC

- Generate from programs that were problematic in Fujitsu's production compilers in the past
- Extract kernel parts and modify them to use under HCQC
  - Extract hot spots
  - If it is Fortran program, then convert them to C language(for comparison between GCC and Clang/LLVM)
  - Prepare the data to run and check those kernel parts
- This work is due to be completed by the end of this month.
  - Will push all under `hcqc/test-program’

Future Work

- Add supports for SVE(if available in GCC or LLVM)
- Implement metric programs:
  - vectorization(`vectorize`)
  - software pipelining(`swpl`)
  - instruction level parallelism(`ilp`)
- Add features for comparing with x86_64(SVE vs. AVX)

Thank you very much!

URL https://github.com/Linaro/hcqc

Any comments or suggestions are welcome.

Implementation of HCQC

- Commands by Python3 scripts: ~2300 LOC
- Information files by JSON format
- Result data files by JSON format
- Result report files by CSV format
- Control flow graphs by Graphviz`s dot


```
[
   [ "TITLE", ["CFG", "DEPTH", "memory", "branch", "other"]],
   [ "kernel ", [ "0", "0", "0", "0"]],
   [ ".LFB0  ", [ "0", "0", "0", "17"]],
   [ ".L3    ", [ "0", "0", "0", "3"]],
   [ ".L5    ", [ "0", "0", "0", "4"]],
   [ ".L4    ", [ "0", "0", "0", "21"]],
   [ ".L12   ", [ "0", "0", "0", "0"]],
   [ ".LFE0  ", [ "0", "0", "0", "0"]],
   [ "*SUMMARY*", [ "-", "0", "0", "45"]]]
```

Test Program Info Files

- Define information for building a test program and checking execution results

```
{
   "LANGUAGE" : "C",
   "MAIN_FLAGS" : [“?DEBUG_FLAG", “?C99_STANDARD"],
   "KERNEL_FLAGS" : ["-DFAST", “?C99_STANDARD "],
   "LINK_FLAGS" : [“?C99_STANDARD "],
   "LIB_LIST" : ["-lm"],
   "MAIN_FILENAME" : "main.c",
   "KERNEL_FILENAME" : "kernel.c“,
   "KERNEL_FUNCTION_NAME" : "kernel“,
    “INPUT" : [“STDIN", "in.data"],
    “OUTPUT" : [“STDOUT", “out.data"]
}
```

Example of test program info file

Metric Programs in HCQC

- Methods for collecting information depends on compilers.