J2: Compilers Challenges for Computing In Memory

Software Side Perspectives

ACACES july 2022 summer school, HiPEAC, Fiuggi, Italy

Henri-Pierre CHARLES

CEA DSCIN department / Grenoble

july 12, 2022

Outline

1. GCC / Compilation
   - SOA C compil chain
   - Rappels Static Compilation chain
   - SW-SOA GCC
   - GCC : New ISA port a new architecture
   - GCC : ISA-FMA
   - GCC : GCC-Research
2. LLVM / C Compilation
   - SOA LLVM
   - Examples : LLVM
   - Examples of code
   - Examples LLVM3
   - Examples LLVM4
   - Examples LLVM4
   - LLVM : LLVM-Research
3. Java
   - SOA Java
   - SW-SOA Dalvik
   - SW-SOA Dalvik deploy
   - Example : Java-Bytecode
   - Java-Compilation
   - Example Java-VM
4. JavaScript
   - Examples Javascript engine : SpiderMonkey
   - Examples Javascript interpretation
   - Examples Trace based optimization
5. Simulation platforms
   - Arch : Architecture Simulation Why ?
   - Arch : Simulation Platforms and Levels
   - Arch : QEMU
   - Arch : Qemu Example “how the code is vectorized”
   - Arch : Qemu-Research

Scientific Evolution : Compilation Research Domains

Compilation Topics Map

Find code structure Extract parallelism : polyhedral approach

Assertion on legacy code correctness proof, hard realtime, model checking

Security HW attach counter mesure, obfuscation

Tools for scalability Systems and library for big parallel machines

Reproductibility Statistics tools and reproductible research

Ad hoc code optimization Application driven code optimization

Legacy compiler optimization follow the HW evolution

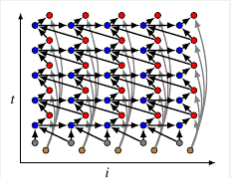
New code generation paradigm JIT, Dynamic code generation

https://top500.org

LAPACK June 2002 #1 : HPE Cray 8,730,112 cores, Linpack perf 1,102.00 PFlop/s, 65% of peak performance (usualy better)

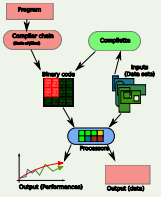
HPCG Fugaku 7630848 cores, 16,004 TFlops 2.98 % of peak performance !

Polyhedral model



(a) CDAG for $T = 4, N = 7$

Algorithmic accelerator



SOA C compil chain

Description

- Language support
  - C language (K&R), C90 up to C11
  - Normalized
  - gcc, clang, icc, tcc, ...
  - A lot of LEGACY
- Open source is mandatory : Apple, Intel
- Economic step !

Compilation chain

- Static compilation
  - Preprocessing (# stuff)
  - Compilation
    - Lexical / Syntaxic
    - IR form
    - Phases / passes
    - Instructions selection
  - Assembly
  - Load
- Dynamic part
  - OS : `ld`
  - Addresses resolution
  - Cache interaction ?

`https://en.wikipedia.org/wiki/C_(programming_language)`

Rappels Static Compilation chain

Static compilation (on C language):

1. Preprocessor (all # stuff : rewriting) `cc -E`
2. Compilation (from C to textual assembly) `cc -S`
3. Assembly (from textual asm to binary asm) `cc -c`
4. Executable (binary + dynamic library)

Optional

- Profiling: Compile (use `-pg`) produce `File`; Run `File`; Use `gprof`
- `cc -da` dump all intermediate representation

(Use gcc -v to see all the steps)
Don't stop at static time (Operating system + processor): Load in memory, dynamic linking; Branch resolution; Cache warmup

SW-SOA GCC

Description

- https://gcc.gnu.org/
- Many platforms supported
- Since 1987
- C, C++, Fortran, Objectice-C, Ada
- OpenMP, OPenACC
- GPL Licence (base compiler for Linux distro)
- Written in C, rewritten in C++ since some years

Compilation chain

- Front-end (L to syntax tree)
- GENERIC and GIMPLE (SSA) intermediate representation
- RTL intermediate representation
- Back-end

Features

- LTO
- Plugins
- Transactional memory support

GCC : New ISA port a new architecture

Machine description

- Write a `.md` file
- Write low level files (fdiv, low level OS)
- Build

Write new optimisation

- Recognize IR pattern
- Insert low level optimization

Extract from `gcc/config/aarch64/aarch64.md`

```
(define_insn "add<mode>3"
  [(set (match_operand:GPF_F16 0 "register_operand" "=w")
        (plus:GPF_F16
         (match_operand:GPF_F16 1 "register_operand" "w")
         (match_operand:GPF_F16 2 "register_operand" "w")))]
  "TARGET_FLOAT"
  "fadd\\t%<s>0, %<s>1, %<s>2"
  [(set_attr "type" "fadd<stype>")]
)
```

GCC : ISA-FMA

Extract from gcc/config/aarch64/aarch64.md

```
;; fma - expand fma into patterns with the accumulator operand first since
;; reusing the accumulator results in better register allocation.
;; The register allocator considers copy preferences in operand order,
;; so this prefers fmadd s0, s1, s2, s0 over fmadd s1, s1, s2, s0.

(define_expand "fma<mode>4"
  [(set (match_operand:GPF_F16 0 "register_operand")
        (fma:GPF_F16 (match_operand:GPF_F16 1 "register_operand")
                     (match_operand:GPF_F16 2 "register_operand")
                     (match_operand:GPF_F16 3 "register_operand")))]
  "TARGET_FLOAT"
)

(define_insn "*aarch64_fma<mode>4"
  [(set (match_operand:GPF_F16 0 "register_operand" "=w")
        (fma:GPF_F16 (match_operand:GPF_F16 2 "register_operand" "w")
                     (match_operand:GPF_F16 3 "register_operand" "w")
                     (match_operand:GPF_F16 1 "register_operand" "w")))]
  "TARGET_FLOAT"
  "fmadd\\t%<s>0, %<s>2, %<s>3, %<s>1"
  [(set_attr "type" "fmac<stype>")]
)
```

GCC : GCC-Research

Code Legacy

- Invent a new architecture for new application
- Port to a new architecture
- Write C application
- Run it ... on which platform ?
- What is your metric ?

Structure optimisation

- Invent a new optimization
  - Based on code structure
  - Based on application structure
- Use accelerator
- Performance portability

SOA LLVM

Description

- Low Level Virtual Machine `http://llvm.org`
- Many compilation project (compiler toolbox)
- C; C++; Objective-C
- Other frontend via dragonegg
- Licence BSD (Base compiler in FreeBSD)
- Written in C++ since the beginning
- Much better error handling (due to parsing technique)
- Huge usage by Apple, soon by Intel

http://llvm.org/docs/

Compilation chain

- Native compilation (clang) : gcc replacement
- LLVM bytecode interpretation (lli)
- Ahead of time compilation ()
- JIT

Examples : LLVM

LLVM Low Level Virtual Machine

- Created as low level platform optimization
- Multiple input language (C, C++, ObjectiveC, other languages)
- `clang` is a gcc replacement
- Powerfull Intermediate Representation, much powerfull than C
  - Vector representation
  - Word size

http://llvm.org/

LLVM Intermediate representation

- Register based : efficient
- SSA form : powerfull format for optimization

Examples of code

Argumentation

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]){
  int i = atoi (argv[1]);
  printf ("Hello␣%d\n", i+1);
  return 0;
}
```

LLVM bytecode (see file Hello.ll)

```
; ModuleID = 'Hello.c'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [10 x i8] c"Hello␣%d\0A\00", align 1

; Function Attrs: nounwind uwtable
define i32 @main(i32 %argc, i8** nocapture readonly %argv) #0 {
  %1 = getelementptr inbounds i8*, i8** %argv, i64 1
```

Examples LLVM3

Static compilation

- `clang -O3 -o Hello Hello.c`
- Error message more precise

Dynamic compilation

- `clang -emit-llvm -c -o Hello.bc Hello.c`
- `lli Hello.bc`

Examples LLVM4

C example

```
int sum (int a[], int len)
{
  int i, sum;
  sum = 0;
  for (i = 0; i < len; i++)
    {
      sum += a[i];
    }
  return sum;
}
```

Code intermédiaire -O0

```
; ModuleID = 'VectSumExample.c'
source_filename = "VectSumExample.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

../..
; <label>:11:
; preds = %7
../..
  %16 = load i32, i32* %15, align 4
  %17 = load i32, i32* %6, align 4
  %18 = add nsw i32 %17, %16
  store i32 %18, i32* %6, align 4
  br label %19

; <label>:19:
; preds = %11
  %20 = load i32, i32* %5, align 4
  %21 = add nsw i32 %20, 1
  store i32 %21, i32* %5, align 4
```

Examples LLVM4

Code intermédiaire -O0

```
; ModuleID = 'VectSumExample.c'
source_filename = "VectSumExample.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

../..
; <label>:11:
; preds = %7
../..
  %16 = load i32, i32* %15, align 4
  %17 = load i32, i32* %6, align 4
  %18 = add nsw i32 %17, %16
  store i32 %18, i32* %6, align 4
  br label %19

; <label>:19:
; preds = %11
  %20 = load i32, i32* %5, align 4
  %21 = add nsw i32 %20, 1
  store i32 %21, i32* %5, align 4
```

Code intermédiaire -O3

```
; ModuleID = 'VectSumExample.c'
source_filename = "VectSumExample.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

../..
  %41 = load <4 x i32>, <4 x i32>* %40, align 4, !tbaa !2
  %42 = getelementptr i32, i32* %39, i64 4
  %43 = bitcast i32* %42 to <4 x i32>*
  %44 = load <4 x i32>, <4 x i32>* %43, align 4, !tbaa !2
  %45 = add nsw <4 x i32> %41, %36
  %46 = add nsw <4 x i32> %44, %37
  %47 = or i64 %17, 24
../..
```

LLVM : LLVM-Research

Port to a new target

- Write platform description
- Write low level libraries ..
- Use JIT compiler
- ...

Use IR strenght

- Generate IR directly from high level application / programming language

Examples

- SPIR
- MLIR
- OpenCL

SOA Java

Description

- Java Language (Sun microsystems / Oracle), 95
- Normalized
- Stack machine
- Enormous API (+10000 classes) “Compile once, run anywhere”
- Oracle, IBM, Google (+/-)
- Licence GPL

Compilation chain

- javac (.java to .class bytecode)
- interpretation
- “Hotspot” compiler
  - trigger most used method
  - server or client profile
- class to executable

https://en.wikipedia.org/wiki/Java_(programming_language)

SW-SOA Dalvik

Description

- Java language
- Android API
- Normalized ...
- Google toolbox
- Security
- Store
- Dalvik is a register based machine

https://source.android.com/devices/tech/dalvik/

Compilation chain

- java to dalvik
- Dalvik VM optimized for small memory machines
- dalvik jitted

SW-SOA Dalvik deploy

Description

- Program application
- Verify functionnality agains Android version
- Deploy on google store

Compilation chain

- Download (pay ?) application
- Ahead of time compilation
- Run the code
- Report on VM failure

Example : Java-Bytecode

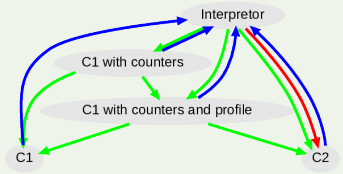
Java context usages

- Web browser : speed and reactivity
- Server : reasonnable speed quickly
- HPC : optimallity slowly

Optimization level

- Interpretor : slow but reactive
- C1 : JIT compiler
- C2 : Complex JIT compiler, very slow

Compiler transition



Green : Staged mode; Red : normal mode; Green : backtrack

Java-Compilation

Java source : javac Hello.java

```
public class Hello{
    static public void main(String[] arg){
        int i = Integer.parseInt(arg[0]);
        System.out.println ("Hello␣:␣"+(i+1) );
    }
}
```

Java “Bytecode”: javap -c Hello.class

```
Compiled from "Hello.java"
public class Hello {
  public Hello();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
```

Example Java-VM

Java Virtual machine

- Java input language
- Interpret byte codes
- ISA 0 adresses
- Implicit stack machine

Virtual machine

- Interpretation: straightforward
- Stack machine : inefficient

Bytecode Examples

opcode	code	semantic
nop	00	→
iconst_0	03	→0
iconst_1	04	→1
iadd	60	v1, v2 →r
ladd	61	v1, v2 →r
fadd	62	v1, v2 →r
dup	59	v →v, v
swap	5f	v2, v1 →v1, v2

Examples Javascript engine : SpiderMonkey

Javascript engine in Firefox

- C and C++
- Source Files
- Support : arm arm64 mips-shared mips32 mips64 x64 x86

Spidemonkey internals

Jit Strategy

- Read javascript
- Transform in bytecode
- Interpret (big switch)
- Trace based compilation

Examples Javascript interpretation

switch on bytecode

- Emulate a processor
- Register state, PCcounter, Stack, etc

Code example

```
#define MAX 1024
unsigned char program[MAX];
int stack[MAX];
int PCcounter, STACKcounter;
enum OPCODE {iADD, iSUB};

void functionInterpret ()
{
  int iresult;
  switch (program[PCcounter++])
    {
    case iADD:
      iresult = stack[STACKcounter] + stack[STACKcounter];
      stack[--STACKcounter] = iresult;
      break;
    case iSUB:
      iresult = stack[STACKcounter] - stack[STACKcounter];
      stack[--STACKcounter] = iresult;
      break;
      /* ../.. */
    }
}
```

Examples Trace based optimization

Trace

- Count every executed instruction (in the program)
- Observe histogram

Trace optimization

- Select a huge profile
- Binary compile this profile
- Replace instructions by call to binary
- (Time optimisation)

Arch : Architecture Simulation Why ?

Usual experiment for compilers writer

1. Wait until new architecture arrive
2. Wait until new application arrive
3. Write new compiler optimization

Emulation based

- Run new application on future platform
- Need performance model
- Could interact with processor designer
- Choose a correct level (Stay at ISA level)

Arch : Simulation Platforms and Levels

Argumentation

- Atomistic level / Physicists
- Electronic level / HW engineer
- Instruction level / Compiler & application
- System level /

Which level is our science ?

Name	In	Out	Time
BigDFT	Atoms position	Electonic behavior	ns
SPICE	Elect. behavior + layout	ms	
VHDL	Bloc behavior		
SystemC	System behavior	Event	
Gem5	Binary code	Functionnal behavior	few sec.
QEMU	Binary code	Functionnal behavior	real time

Arch : QEMU

Sytem level emulation

- Multiple possible usage :
  - System level
  - User level
  - KVM Hosting
  - Xen hosting
- GNU licence

Author : Fabrice Bellard

Working principle

1. Read a block of binary source instruction
2. Transform it in IR
3. Binary compile it in target instruction
4. Execute it
5. Keep it in a buffer for reuse

Arch : Qemu Plugin

Evolution

- Recent evolution:
  - 0.1 started in 2009
  - 4.2.0 in 2019 : TCG plugin
  - 5.0 2020
  - 7.0 2022
- Call back : register a fonction for an event

Usage https://wiki.qemu.org/Features/TCGPlugins

- `https://qemu.readthedocs.io/en/latest/devel/tcg-plugins.html#writing-plugins`
- Configure with `-enable-plugins`
- `qemu-risc32 -d plugin -plugin /path/to/plugin <riscv binary>`
- or use environment variable `QEMU_PLUGIN`

Main API calls

```
int qemu_plugin_install()                       // Register plugin
void qemu_plugin_register_atexit_cb();          // Register for exit
void qemu_plugin_register_vcpu_tb_trans_cb();   // Block to be compiler
void qemu_plugin_register_vcpu_tb_exec_inline() // Counter for an instruction
```

Arch : Qemu Example “how the code is vectorized”

https://github.com/qemu/qemu/blob/master/contrib/plugins/howvec.c

- Decode opcode
- Classify instruction
- Print statistics at the end

Illustration

`Arch/QemuExample/howvec.c`

Arch : Qemu-Research

Legacy

- Implement new architectures (examples : Kalray, riscv-128)
- Support new systems

Experiment

- System modelization: cache level
- Performance and power model at instruction level
- Experiment on full application, (run to completion)