Dynamic Compilation Everywhere

Henri-Pierre.Charles@cea.fr

Henri-Pierre Charles

CEA DACLE department / Grenoble

Introduction : Course Organization

Dynamic Code Generation

- Metrics
- HW architecture notion
- State of the art (SOA)
  - Architecture
  - Compiler
- IR for code generation

Evaluation

- Multiple Choice Questions tests at the end (whithout documents)
- "Short term memory"
- Questions topics are given : search for 

Assumption : you

- understand my broken froglish
- know C language
- know compiler structure
- know a script language

Intro Why Dynamic Compilation

Pro

- Dynamic application
- Memory hierarchy versus data set
- Unknown running architecture
- Specialized instructions
- Fast development cycle

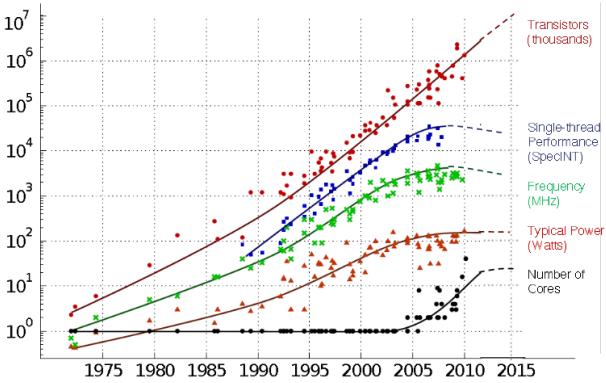
Cons

- Static Compilation
- Can verify, assert
- As many time as needed

Dynamic Code Generation ! = Dynamic Optimization (or programming)

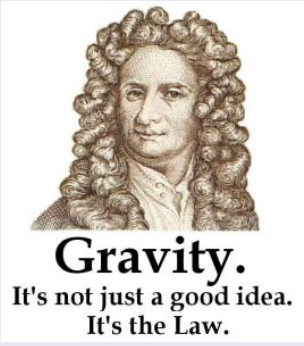
35 Years of microprocessor trend data

Natural evolution ?



Laws Vocabulary

Argumentation



Short list of importants laws / notions

- Moore Moore's_law
- Dennard Scaling Dennard_scaling (Frequency evolution)
- Amdahl Amdahl's_law
- Memory bound IO_bound
- CPU bound CPU_bound
- Roofline model Roofline_model

Vocabulaire Speedup

Notion

Speedup $S = 100 * \frac{T_{seq}}{T_{opt}}$

Can be between

- 1 and N processor
- 1 and vectorized
- non optimized versus optimized version

Mesure Quality what are the execution conditions : data set, computer workload, reproducibly, ...

Computer science has to use “human science” tools & methodology

Metrics How to Measure Performance ?

What scale ?

- Application level (TPS, Frame/s, .../)
- Run to completion
- Method level
- Instruction level

Tools

- Wall clock
- `gettimeofday`
- Performance counters

Full application or function call ?

- Cold start / warmup
- Statistic / multiple calls
- How many calls

Vocabulary Peak / Sustained

Notions

Peak performance : maximal theoretical performance, assuming no bubble

Sustain performance : real acheived performance, on a real benchmark

Cost

- What is the percentage you're ready to lose ? 90% 95% ?
- How many are you ready to pay (time, money) to minimise this loss ?

TOP500

Vocabulary : Cycle per seconds / Flops

Units

Mips	Million operation per second
Flops	Floating point operation per second `http://www.top500.org`
Flops/Watt	Floating point operation per watt per second `http://www.green500.org`
IPC	: Instructions per Cycle

How to mesure

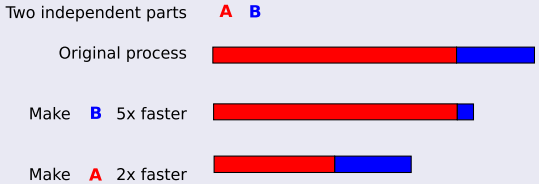
- Analytique (wall clock)
- Instrumentation
  - "Portable" (`gettimeofday()`)
  - Hardware (hardware performance counter)

Vocabulary : Amdahl

Amdahl law is forever

Assume that a task has two independent parts, A and B. B takes roughly 25% of the time of the whole computation. By working very hard, one may be able to make this part 5 times faster, but this only reduces the time for the whole computation by a little. In contrast, one may need to perform less work to make part A be twice as fast. This will make the computation much faster than by optimizing part B, even though B's speed-up is greater by ratio, (5x versus 2x)

Illustration

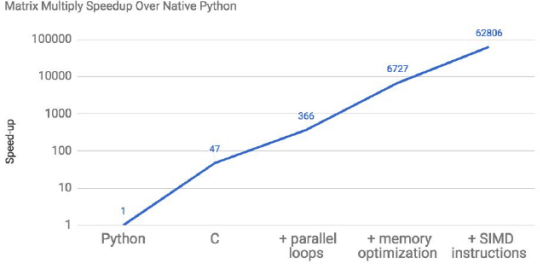


Presentation : Opportunity

Speed opportunity

What's the Opportunity?

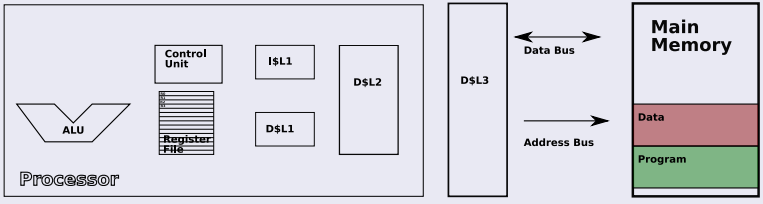
Matrix Multiply: relative speedup to a Python version (18 core Intel)



from: "There's Plenty of Room at the Top," Leiserson, et. al., *to appear*.

Architecture Schematic

Illustration



Instruction examples

- `add r0, r2, r3`
- `load r0, r2`

Arch Memory Cache Hierarchy

Memory type		Access time	Who care ?
Processor Register		ns	Compiler
Cache L1		ns/ms	Compiler
Cache L2		ns/us	Compiler
Cache L3		us/ms	Compiler /Application
RAM		10 ms	Application
Flash / SSD		.1 s	System
Hard drive		1-10 s	System
Tape Backup		mn/hr	User

Price / Access Time / Who care about

Arch Instruction Execution

Argumentation

- While true
  - Fetch instruction
  - Increment PC
  - Decode instruction
  - Fetch / Get Data
  - Execute operation
  - Store result

Vocabulary

- Register
- UAL
- PC
- RISC/CISC

https://en.wikipedia.org/wiki/Instruction_cycle

SW-SOA OOO

Out of Order principle

- "Intelligent Reordering"
- Add a dispatch queue
- CF Slide ARM big.LITTLE

Illustration

- Easy to compile
- Same ISA, evolution at micro architectural level (TICK/TOCK Intel)

- Difficult to compile very efficient code

https://en.wikipedia.org/wiki/Out-of-order_execution

SW-SOA MicroController

Description

Single chip with

- Compute node
- RAM / ROM
- I/O
- DAC Converter
- Chip radio (IoT)

Programming

- Bare metal 
- Cross compilation
- Interactive (Script)

`https://en.wikipedia.org/wiki/Microcontroller`

MPSoC : Multiprocessor System on Chip

One single chip which contains

- IP blocks (Intellectual property).
  - FFT / Turbo code / ../..
- Processors
- Memory

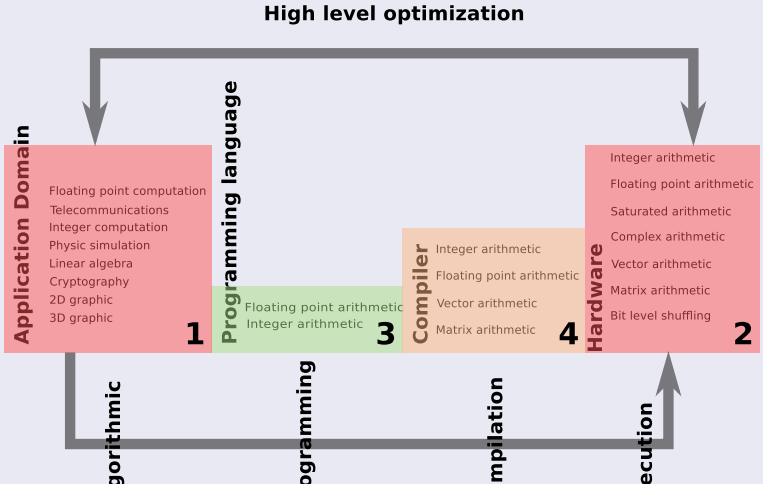
Why ?

- IP : Save energy
- Same chip : Save energy, ease integration
- Memory on chip : Save energy
- Multiple processors : parallelism to ... save energy

http://en.wikipedia.org/wiki/System_on_a_chip

From Application to binary code

Programming Language Semantic Bottleneck



SOA Arch : ISA

Instruction Set Architecture

- Lowest programming Level model : assembly language
- Allow to use processor full capacity
  - Special instructions (multimedia, vectors)
  - Strange memory access
- Binary representation
- Register access

What is the “instruction execution algorithm” ?
Is the assembly language still used ?

Instruction_set Comparison_of_instruction_set_architectures

Quizz Question 1

ADD : a "simple instruction"

Extract from an ARM databook

A8.6.4 ADD (immediate, Thumb)

This instruction adds an immediate value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

`ADDS <Rd>,<Rn>,#<imm3>` Outside IT block.

`ADD<c> <Rd>,<Rn>,#<imm3>` Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	imm3			Rn			Rd		

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

Encoding T2 ARMv4T, ARMv5T*, ARMv6*, ARMv7

Quizz Question 2

USADA8 : not a “simple instruction”

Extract from an ARM databook

A8.6.254 USADA8

Unsigned Sum of Absolute Differences and Accumulate performs four unsigned 8-bit subtractions, and adds the absolute values of the differences to a 32-bit accumulate operand.

Encoding T1 ARMv6T2, ARMv7

```
USADA8<c> <Rd>,<Rn>,<Rm>,<Ra>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	0	1	1	1	Rn				Ra				Rd				0	0	0	0	Rm			

```
if Ra == '1111' then SEE USAD8;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a == UInt(Ra);
if BadReg(d) || BadReg(n) || BadReg(m) || BadReg(a) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

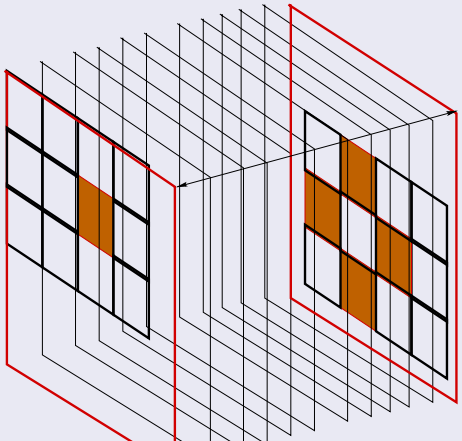
```
USADA8<c> <Rd>,<Rn>,<Rm>,<Ra>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	1	1	0	0	0	Rd				Ra				Rm				0	0	0	1	Rn			

```
if Ra == '1111' then SEE USAD8;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a == UInt(Ra);
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

Problem example : video compression

Illustration Video Compress



Problem example : video compression

```
1  #define PIXEL_SAD_C( name, lx, ly ) \
2  int name( uint8_t *pix1, int i_stride_pix1,  \
3                   uint8_t *pix2, int i_stride_pix2 ) \
4  {                                                   \
5      int i_sum = 0;                                  \
6      int x, y;                                       \
7      for( y = 0; y < ly; y++ )                       \
8      {                                               \
9          for( x = 0; x < lx; x++ )                   \
10         {                                           \
11             i_sum += abs( pix1[x] - pix2[x] );      \
12         }                                           \
13         pix1 += i_stride_pix1;                      \
14         pix2 += i_stride_pix2;                      \
15     }                                               \
16     return i_sum;                                   \
17 }
```

(Extract from x264 video compressor from `www.videolan.org` project, $(lx, ly) \in \{(4,4),(4,8),(8,4),(8,8),(8,16),(16,8),(16,16)\}$

SOA Arch : Language Classification / Addresses

Definition

- 0 address : stack machines ; all operators are implicit (bytecode java) (Java bytecode, postscript)
- Register machines :
  - 1 address : A add (other operand implicits)
  - 2 address : A += B (x86)
  - 3 address : A = B + C (itanium)
  - 4 address : A = B * C + D (power)

Compromise

- Code compacity, expressiveness
- “Simplicity” / efficiency
- Code generation speed

SOA ARCH : ARM big.LITTLE

big

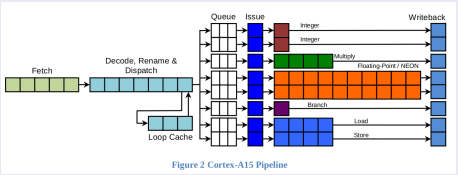
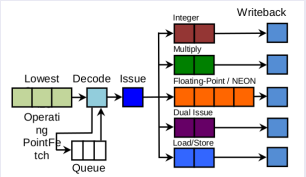


Figure 2 Cortex-A15 Pipeline

LITTLE



ARM : “more than 30 billion processors sold with more than 16M sold every day ARM” (Nov 2013) `http://www.arm.com/products/processors/index.php`

- 4 big processors + 4 little
- Same ISA, . . .
- (even for vector operation)
- Low latencie switch

big.LITTLE notion

SOA Architecture : Raspberry

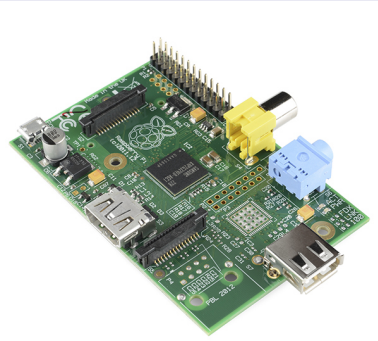
Building block for geek Web site

Raspberry

Characteristics :

- ARM1176JZF-S (ARMv6) 700MHz
- RAM : 256 Mo
- 2 video out ; audio in/out
- SDCARD mem ; 1 Port USB 2.0 ;
- 300 mA
- Full OS : linux, BSD, ...
- 20 $

Illustration



SOA Architecture : Texas Msp430

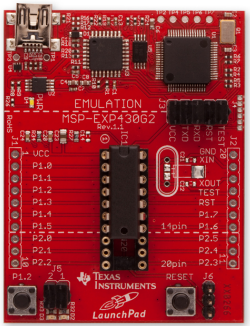
Building block for embedded system Arch Overview

Texas Microcontroller

Characteristics :

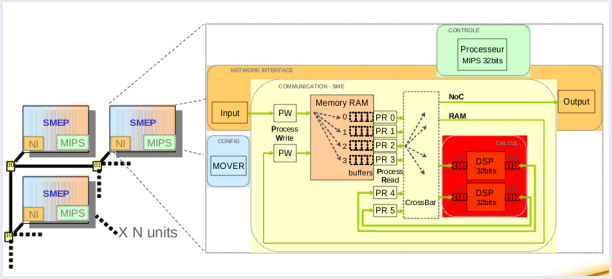
- 8 to 24 MHz
- 512 to 1024 bytes of ram
- Bare metal
- 1.75$ to 2.45$
- Cross compiler

Illustration



SOA Architecture : Genepy

Genepy Bloc diagram



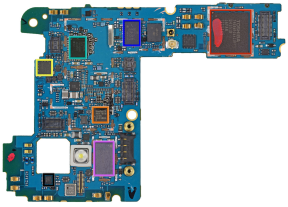
Genepy characteristics

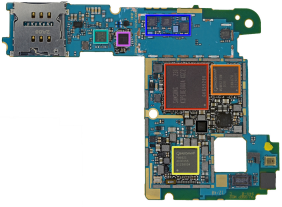
LTE modem processor

- Very low power (G Insn / mw)
- Hard to program (Kb of RAM), specialized operators

Architecture Nexus4

- Face
  - (red) Toshiba THGBM5G6A2JBA1R 8GB Flash
  - (orange) SlimPort ANX7808 SlimPort Transmitter (HDMI output converter)
  - (yellow) Invensense MPU-6050 Six-Axis (Gyro + Accelerometer)
  - (green) Qualcomm WTR1605L Seven-Band 4G LTE chip
  - (blue) Avago ACPM-7251 Quad-Band GSM/EDGE and Dual-Band UMTS Powe Amplifier
  - (violet) SS2908001
  - (black) Avago 3012 Ultra Low-Noise GNSS Front-End Module
- Back
  - Samsung K3PE0E00A 2GB RAM. We suspect the Snapdragon S4 Pro 1.5 GHz CPU lies underneath.
  - Qualcomm MDM9215M 4G GSM/CDMA modem
  - Qualcomm PM8921 Power Management
  - Broadcom 20793S NFC Controller
  - Avago A5702, A5704, A5505
  - Qualcomm WCD9310 audio codec
  - Qualcomm PM8821 Power Management





http://www.ifixit.com/Teardown/Nexus+4+Teardown/11781

Rappels Nexus4Compil

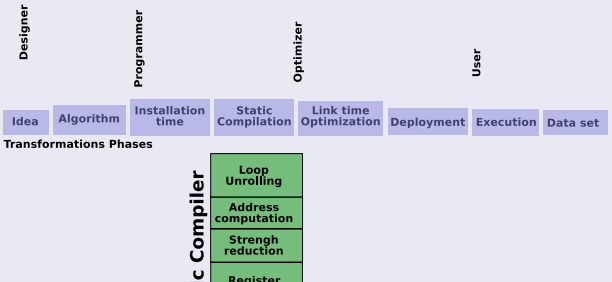
- Application SDK :
  - Java programming language
  - Dalvik virtual machine
  - Android Market / Google controled
- Native compilation
  - Modem stack
    - Native code
    - Cross compiler
    - Closed source
  - System stack mostly opensource Cyanogenmod build
    - Virtual machine Cyanogenmod
    - Building from source

Intro Compilers

Classical Compiler architecture : GCC, LLVM

- Driven by performance only
- Mono architecture
- Not energy aware

Before / After

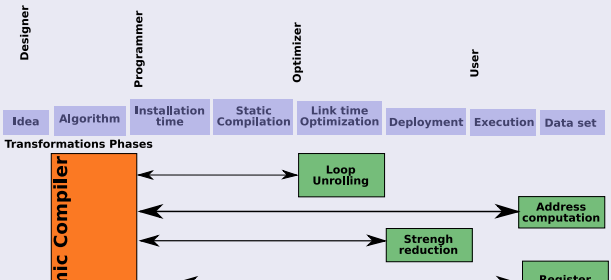


Intro Compilers

Future Compilers Architecture

- Multi objective (execution time, power, thermal constraints)
- Multi-target (Heterogeneous multi SoC)
- Data driven (dynamically)

Before / After



SOA C compil chain

Description

- C language (K&R), C90 up to C11
- Normalized
- gcc, clang, icc, tcc, ...

Compilation chain

- Preprocessing (# stuff)
- Compilation
  - Lexical / Syntaxic
  - IR form
  - Phases / passes
  - Instructions selection
- Assembly
- Load

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

Rappels ClassicalsCompilers

gcc	Gnu C compiler GCC
llvm/clang	Low Level Virtual Machine LLVM
open64	open64
sdcc	Small device C compiler sdcc
tcc	Tiny C Compiler TCC
icc	Intel C compiler x86 / Itanium Intel C Compiler
xlcc	IBM compiler power / powerpc IBM C Compiler
oracle solaris	sparc / x86 Oracle C compiler
../..	

What are the differences between them ?

SOA Java

Description

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

Java_(programming_language)

Compilation chain

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

SW-SOA Java JIT Strategy

Argumentation

- Count how many time methods are called
- Optimize with different levels

Illustration

- Does not take data into account
- Does not vectorize

SW-SOA JavaJit

How Java generate binary code

Execution starts with level 0
Different levels :

- 0 - interpreter
- 1 - C1 with full optimization (no profiling)
- 2 - C1 with invocation and backedge counters
- 3 - C1 with full profiling (level 2 + MDO)
- 4 - C2 (server)

Scenarios

- Common : 0, 3, 4
- C2 queue too long : 0, 2, 3, 4
- In queue change : 0, (3, 2), 4
- Trivial (small nothing to profile) : 0, 3, 1 or 0, 2, 1
- Fully profile in interpreter : 0, 4
- De-optimization : (1, 2, 3, 4), 0

SW-SOA Dalvik

Description

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

Compilation chain

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

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

SOA LLVM

Description

- Low Level Virtual Machine
  `http://llvm.org`
- Many compilation project (compiler toolbox)
- C ; C++ ; Objective-C
- Other frontend via dragonegg

Compilation chain

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

SOA JavaScript

Description

- Script language
- Trace based compilation 
- Tons of tools (server side and client side) 
- Used as back-end language
- Normalized, but with variants

Compilation chain

- Source interpretation
- JIT part of scripts 

JavaScript Comparison_of_server-side_JavaScript_solutions Node.js
`http://bellard.org/jslinux/`

SW-SOA JavaScriptVM

VM Category

- Web Browser
  - V8_(JavaScript_engine) Chrome browser
  - SpiderMonkey_(JavaScript_engine) Firefox
- Server side List_of_server-side_JavaScript_implementations
  - Node.js
- Embedded
  - “Smallest javascript”
  - Tiny-JS / STM32 C++

Differences

- Memory footprint
- Efficiency
- Extensions
- Standard support

SW-SOA OpenCL

"The open standard for parallel programming of heterogeneous systems"

- OpenCL Web
- "Embedded source" acceleration
- JIT compiled by accelerator driver
- Multiple technology provider
- "C-like programming model"

Cons

- Difficult to have portable performance
- Programmer need to organiz data choregraphy

SW-SOA OpenCL-Example

(OpenCL part only)

```
1  // Multiplies A*x, leaving the result in y.
2  // A is a row-major matrix, meaning the (i,j) element is at A[i*ncols+j].
3  __kernel void matvec(__global const float *A, __global const float *x,
4                       uint ncols, __global float *y)
5  {
6      size_t i = get_global_id(0);            // Global id, used as the row index.
7      __global float const *a = &A[i*ncols]; // Pointer to the i'th row.
8      float sum = 0.f;                        // Accumulator for dot product.
9      for (size_t j = 0; j < ncols; j++) {
10         sum += a[j] * x[j];
11     }
12     y[i] = sum;
13 }
```

SW-SOA CUDA

Argumentation

- NVIDIA programming language
- Computing for GPU device
- JIT compiled by GPU device drive
- Data parallel algorithm
- Cuda Developper

Cons

- Programmer need to organize “data choregraphy”
- Only one technology provider
- Difficult to acheive performance accross different devices

SW-SOA CUDA-Example

Python CUDA example

```
1  import pycuda.compiler as comp
2  import pycuda.driver as drv
3  import numpy
4  import pycuda.autoinit
5
6  mod = comp.SourceModule("""
7  __global__ void multiply_them(float *dest, float *a, float *b)
8  {
9  const int i = threadIdx.x;
10 dest[i] = a[i] * b[i];
11 }
12 """)
13
14 multiply_them = mod.get_function("multiply_them")
15
16 a = numpy.random.randn(400).astype(numpy.float32)
17 b = numpy.random.randn(400).astype(numpy.float32)
18
```

SW-SOA Other

Other Languages

- Python, PHP, LUA, FORTH, ...
- Fast development
- Huge library

Compilation chain

- None
- Script compression
- JIT
- Native Call Interaction

Intermediate representation (IR)

Represent a program

- Each compiler has a/many IRs
- Instruction List
- Programming model (0/1/2/3 address)
- Easy to manipulate (SSA)
- List of instructions

What for ?

- (re) Generate code
- Binary code at static compil time
- Binary code at run-time

IR LLVM

LLVM Intermediate representation

- SSA form
- "Easy" to read
- Used for OpenCL
- Many Possible Scenarii

Language référence

Examples

- Compile to native `clang file.c -o file`
- Compile to bytecode `clang -emit-llvm -S file.c -o file.bc`
- Interpret : `lli file.bc`
- Compile ahead of time : `llc test.bc`
- Optimize bytecode :`opt test.bc`

IR : LLVM example

C example

```
1  int sum (int a[], int len)
2  {
3    int i, sum;
4    sum = 0;
5    for (i = 0; i < len; i++)
6      {
7        sum += a[i];
8      }
9    return sum;
10 }
```

Code intermédiaire -O0

```
1  ; ModuleID = 'VectSumExample.c'
2  source_filename = "VectSumExample.c"
3  target datalayout = "e-m:e-i64:64-f80:128-n
4  target triple = "x86_64-pc-linux-gnu"
5
6  ../..
7  ; <label>:11:
   ; preds = %7
8  ../..
9    %16 = load i32, i32* %15, align 4
10   %17 = load i32, i32* %6, align 4
11   %18 = add nsw i32 %17, %16
12   store i32 %18, i32* %6, align 4
13   br label %19
14
15 ; <label>:19:
   ; preds = %11
```

IR Java ByteCode

Java Bytecode

- Stack based machine
- Byte Instructions
- Fast interpretor
- Bytecode Verifier/prover

Examples

- `javac File.java` "Bytecompile" java program
- `jar tf a.jar File.class` Link
- `java File.class or a.jar` Launch VM
  - `-server|-zero|-javm|avian` choose VM
  - `-XX:CompileThreshold=N` 1500=client, 10000=server
- `javah .h` for JNI call

Java_bytecode Java_bytecode_instruction_listings

IR Java ByteCode

Java VM interpretor

- Read an instruction (a byte)
- Choose a case (`switch`)
- Use statistics (method count)
- Method call :

```
if function call count > threshold
then Jit compile

if method native
then call native
else call interpretor
```

HotSpot JIT

- Choose threshold (client versus server)

- JIT compile a method
- Iterative compilation
- Context switch between native and virtual environment !

IR ART

Android Runtime

- Android Virtual Machine (Not JVM)
- Register based
- Dalvik was JITed, ART is ahead of time (install time)

Compiler options

- everything - compiles almost everything
- speed - maximizes runtime performance, (default option)
- balanced - performance/compilation investment.
- space - prioritizing storage space.
- interpret-only
- verify-none - should be used only for trusted system code.

https://en.wikipedia.org/wiki/Android_Runtime