# Developing low-level assembly kernels with Peach-Py

Marat Dukhan

School of Computational Science and Engineering
College of Computing
Georgia Institute of Technology

Presentation on BLIS Retreat, September 2013

## Outline

## Outline

## The Problem

Peach-Py attacks the problem of generating multiple similar assembly kernels:

- Kernels which perform similar operations
  - E.g. vector addition/subtraction
- Kernels which do same operation on different data types
  - E.g. SP and DP dot product
- Kernels which target different microarchitectures or ISA
  - E.g. dot product for AVX, FMA4, FMA3
- Kernels which use different ABIs
  - E.g. x86-64 on Windows and Linux

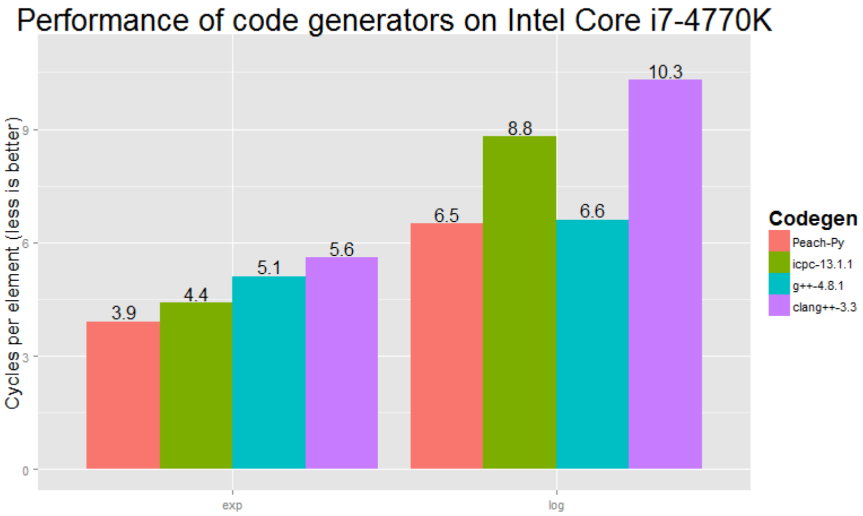
## Why Not C with Intrinsics?

### Experimental Setup

- “Full path” versions of vector elementary functions from Yeppp!
  - Originally written and tuned using Peach-Py
  - A single basic block which processes 40 elements
  - Two functions tested: log and exp
- Assembly instructions were one-to-one converted to C++ intrinsics
  - Compiler only had to do two operations: register allocation (and spilling) and instruction scheduling.

# Why Not C with Intrinsics?

## Benchmarking Results



## Outline

## What Is And Is Not Peach-Py

Peach-Py is a Python-based automation and metaprogramming tool for assembly programming.

### Peach-Py is

- An Assembly-like DSL: Peach-Py user is exposed to the same low-level details as assembly programmer
- A Python framework: some Python knowledge is required to use it

### Peach-Py is not

- A compiler: Peach-Py does not offer high-level programming abstractions
- An assembler: Peach-Py does not generate object code

## Peach-Py Philosophy

- Peach-Py is for writing high-performance codes
  - No support for invoke, OOP, and other "high-level assembly"
- Peach-Py is for writing assembly codes
  - Not a replacement for high-level compiler
- All optimizations possible to do in assembly should be possible to do in Peach-Py
- Whatever can be automated in assembly programming should be automated

## Outline

## Minimal Peach-Py Function

```
from peachpy.x64 import *

abi = peachpy.c.ABI('x64-sysv')
assembler = Assembler(abi)
x_argument = peachpy.c.Parameter("x",
    peachpy.c.Type("uint32_t"))
arguments = (x_argument)
function_name = "f"
microarchitecture = "SandyBridge"

with Function(assembler, function_name,
    arguments, microarchitecture):

    RETURN()

print assembler
```

## Modules and ABIs

- Modules
  - peachpy.x64 for x86-64 instructions
  - peachpy.arm for ARM instructions
  - Intended usage: `from peachpy.{arm|x64} import *`
- ABIs
  - peachpy.c.ABI("x64-sysv") for System V x86-64 ABI (Linux, Mac OS X)
  - peachpy.c.ABI("x64-ms") for Microsoft x86-64 ABI (Windows)
  - peachpy.c.ABI("arm-softeabi") for ARM EABI with SoftFP (armel, Android)
  - peachpy.c.ABI("arm-hardeabi") for ARM EABI with HardFP (armhf)
  - Indicate how parameters are passed to function, and which registers must be preserved in prologue

## Assembler and Function

- Assembler
  - Container for functions
  - Contains only functions with specified ABI
  - Normally may be saved as assembly file to disk
- Function
  - Created using with syntax: `with Function(...):`
  - Creates an active instruction stream
- Microarchitecture
  - String parameter for Function constructor
  - Restricts the set of supported instructions

## Instructions

- Names in uppercase
- All instructions are python objects
- Calling instruction as a Python function adds it to active Peach-Py function
- Most computational x86-64 and many ARM instructions are supported

**Traditional Assembly**

```
.loop:
    ADDPD xmm0, [rsi]
    ADD rsi, 16
    SUB rcx, 2
    JAE .loop
```

**Peach-Py**

```
LABEL( "loop" )
ADDPD( xmm0, [rsi] )
ADD( rsi, 16 )
SUB( rcx, 2 )
JAE( "loop" )
```

## Registers

- Names in lowercase
- x86 register classes GeneralPurposeRegister8/16/32/64, MMXRegister, SSERegister, AVXRegister
- ARM register classes GeneralPurposeRegister, SRegister, DRegister, QRegister

**Traditional Assembly**

```
MOVZX rax, al
PADD mm0, mm1
ADDPS xmm0, xmm1
VMULPD ymm0, ymm1, ymm2
```

**Peach-Py**

```
MOVSX( rax, al )
PADD( mm0, mm1 )
ADDPS( xmm0, xmm1 )
VMULPD( ymm0, ymm1, ymm2 )
```

## Outline

## Register Allocation

### Traditional Assembly

```
VMOVAPD ymm0, [rsi]
VMOVAPD ymm1, ymm0
VFMADD132PD ymm1, ymm13, ymm12
VFMADD231PD ymm0, ymm1, ymm14
VFMADD231PD ymm0, ymm1, ymm15
```

### Peach-Py

```
ymm_x = AVXRegister()
VMOVAPD( ymm_x, [xPointer] )
ymm_t = AVXRegister()
VMOVAPD( ymm_t, ymm_x )
VFMADD132PD( ymm_t, ymm_t, ymm_log2e, ymm_magic_bias)
VFMADD231PD( ymm_x, ymm_t, ymm_minus_ln2_hi, ymm_x )
VFMADD231PD( ymm_x, ymm_t, ymm_minus_ln2_lo, ymm_x )
```

## Using in-memory Constants

### Traditional Assembly

Right here:

```
section .rdata rdata
    c0 dq 3.141592, 3.141592
```

In a galaxy far far away:

```
section .text code
    MULPD xmm0, [c0]
```

### Peach-Py

```
MULPD( xmm_x, Constant.float64x2(3.141592) )
```

## Calling conventions
### The Problem

Consider Assembly implementations of C the function

```
uint64_t add(uint64_t x, uint64_t y) {
    return x + y;
}
```

**Microsoft x86-64 calling convention**

```
add:
    LEA rax, [rcx + rdx * 1]
    RET
```

**System V x86-64 calling convention**

```
add:
    LEA rax, [rdi + rsi * 1]
    RET
```

## Calling conventions

Peach-Py Approach

### Peach-Py code

```
from peachpy.x64 import *
asm = Assembler(peachpy.c.ABI('x64-ms'))
x_arg = peachpy.c.Parameter("x",
    peachpy.c.Type("uint64_t"))
y_arg = peachpy.c.Parameter("y",
    peachpy.c.Type("uint64_t"))

with Function(asm, "add", (x_arg, y_arg), "Bobcat"):
    x = GeneralPurposeRegister64()
    LOAD.PARAMETER( x, x_argument )
    y = GeneralPurposeRegister64()
    LOAD.PARAMETER( y, y_argument )
    LEA( rax, [x + y * 1] )
    RETURN()
```

## ISA-based runtime dispatching

- Peach-Py known the instruction set of each instruction
- Peach-Py also collects ISA information about each function
- This allows to do fine-grained runtime dispatching
    - More efficient vs recompiling the function for each ISA with high-level compiler

## Outline

## Parametrized Unroll

```
with Function(asm, "dotProduct", args, "SandyBridge"):
    xPointer, yPointer, zPointer, length = LOAD.PARAMETERS()

    reg_size = 32
    reg_elements = 8
    unroll_regs = 8

    acc  = [AVXRegister() for _ in range(unroll_regs)]
    temp = [AVXRegister() for _ in range(unroll_regs)]
    ...
    LABEL( "processBatch" )
    for i in range(unroll_regs):
        VMOVAPS( temp[i], [xPointer + i * reg_size] )
        VMULPS( temp[i], [yPointer + i * reg_size] )
        VADDPS( acc[i], temp[i] )
    ADD( xPointer, reg_size * unroll_regs )
    ADD( yPointer, reg_size * unroll_regs )
    SUB( length, reg_elements * unroll_regs )
    JAE( "processBatch" )
    ...
```

## SP and DP from same source

```
reg_size = 32
reg_elements = reg_size / element_size
unroll_regs = 8

SIMD_LOAD = {4: VMOVAPS, 8: VMOVAPD}[element_size]
if
SIMD_MUL  = {4: VMULPS, 8: VMULPD}[element_size]
SIMD_ADD  = {4: VADDPS, 8: VADDPD}[element_size]
...
LABEL( "processBatch" )
for i in range(unroll_regs):
    SIMD_LOAD( temp[i], [xPointer + i * reg_size] )
    SIMD_MUL( temp[i], [yPointer + i * reg_size] )
    SIMD_ADD( acc[i], temp[i] )
ADD( xPointer, reg_size * unroll_regs )
ADD( yPointer, reg_size * unroll_regs )
SUB( length, reg_elements * unroll_regs )
JAE( "processBatch" )
...
```

## Multiple ISA from same source

```
if Target.has_fma():
    VMLAPS  = VFMADDPS if Target.has_fma4() else VFMADD231PS
else:
    def VMLAPS(x, a, b, c):
        t = AVXRegister()
        VMULPS( t, a, b )
        VADDPS( x, t, c )

...
LABEL( "processBatch" )
for i in range(unroll_regs):
    VMOVAPS( temp[i], [xPointer + i * reg_size] )
    VMLAPS( acc[i], temp[i], [yPointer + i * reg_size], acc[i] )
ADD( xPointer, reg_size * unroll_regs )
ADD( yPointer, reg_size * unroll_regs )
SUB( length, reg_elements * unroll_regs )
JAE( "processBatch" )
...
```

## Outline

## Introduction

When writing in assembly, we usually want instructions to appear in the same order as we write them. However, the are several exceptions:

- ARM Cortex-A9 can decode one SIMD instruction + one scalar instruction per cycle. By interleaving SIMD and scalar processing we can achieve higher performance.
- We may use scalar instructions to detect special cases while SIMD unit are busy doing other calculations.
- Splitting the program into multiple instruction streams is helpful for software pipelining.

## Instruction Stream Objects

The Python with statement can be used to redirect generated instructions to an InstructionStream object.

```
scalar_stream = InstructionStream()

with scalar_stream:
    x = GeneralPurposeRegister64()
    MOV( x, [xPointer] )
    CMP.JA( x, threshold, "aboveThreshold" )

with vector_stream:
    ...
```

Instructions from instruction stream can then be re-issued to current instruction stream:

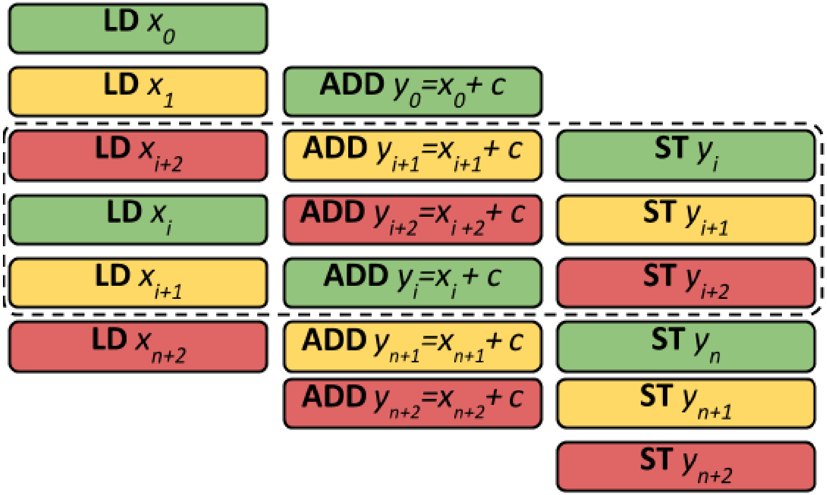
```
while scalar_stream or vector_stream:
    scalar_stream.issue()
    vector_stream.issue()
```

## Software Pipelining

Instruction streams are useful for implementing software pipelining

```
instruction_columns = [InstructionStream(), InstructionStream(),
for i in range(unroll_regs):
    with instruction_columns[0]:
        VMOVDQU( ymm_x[i], [xPointer + i * reg_size] )
    with instruction_columns[1]:
        VPADDD( ymm_x[i], ymm_y )
    with instruction_columns[2]:
        VMOVDQU( [zPointer + i * reg_size], ymm_x[i] )
with instruction_columns[0]:
    ADD( xPointer, reg_size * unroll_regs )
with instruction_columns[2]:
    ADD( zPointer, reg_size * unroll_regs )
```

## Software Pipelining



## Summary

- Assembly is not dead.
- Peach-Py provides a number of tools to simplify programming high-performance kernels in assembly.
- Hand-tuned assembly still beats optimizing compilers, even on manually vectorized and software pipelined codes.

## Links

- Peach-Py repository: bitbucket.org/MDukhan/peachpy
- Yeppp! library (a lot of Peach-Py codes): www.yeppp.info