VECTORS MEET VIRTUALIZATION

ALEX BENNÉE

FOSDEM 2018
INTRODUCTION

- Alex Bennée
  - alex.bennee@linaro.org
  - stsqaud on #qemu
- Virtualization Developer @ Linaro
- Projects:
  - QEMU, KVM, ARM
WHAT IS QEMU?

From: www.qemu.org
"QEMU is a generic and open source machine emulator and virtualizer."
TWO TYPES OF VIRTUALIZATION

- Hardware Assisted Virtualization (KVM*)
- Cross Architecture Emulation (TCG)
HARDWARE ASSISTED VIRTUALIZATION

High Performance, Cloud, Server Consolidation

Diagram:
- **HOST**: QEMU
- **Guest**: App
- **EL0 (User)**
- **EL1 (Kern)**
- **EL2 (Hyp)**
- **Linux**
- **KVM**
- **Kernel**

Connections:
1. Connection from Kernel to KVM hypervisor
2. Connection from KVM hypervisor to Linux
3. Connection from Linux to QEMU
4. Connection from QEMU to App

Numbers 1, 2, 3, and 4 indicate the sequence of connections.
FULL SYSTEM EMULATION

Android Emulator, Embedded Development, New Architectures
LINUX USER EMULATION

Cross-development tools, Legacy binaries

- Guest Linux App (ARM/Linux binary)
- JIT Code (x86)
- QEMU
- Host Linux Kernel (x86)
WHAT ARE VECTORS?
CRAY 1 SPECS

Addressing 8 24 bit address
Scalar Registers 8 64 bit data
Vector Registers 8 (64x64bit elements)
Clock Speed 80 Mhz
Performance up to 250 MFLOPS*
Power 250 kW

## Architectures with Vectors

<table>
<thead>
<tr>
<th>Year</th>
<th>ISA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1994</td>
<td>SPARC VIS</td>
</tr>
<tr>
<td>1997</td>
<td>Intel x86 MMX</td>
</tr>
<tr>
<td>1996</td>
<td>MIPS MDMX</td>
</tr>
<tr>
<td>1998</td>
<td>AMD x86 3DNow!</td>
</tr>
<tr>
<td>2002</td>
<td>PowerPC Altivec</td>
</tr>
<tr>
<td>2009</td>
<td>ARM NEON/AdvSIMD</td>
</tr>
</tbody>
</table>
VECTOR REGISTER

128 bit wide, 4 x 32 bit elements
VECTOR OPERATION

\[ \text{vadd} \ %V_d, \ %V_n, \ %V_m \]
### Vector Size is Growing

<table>
<thead>
<tr>
<th>Year</th>
<th>SIMD ISA</th>
<th>Vector Width</th>
<th>Addressing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1997</td>
<td>MMX</td>
<td>64 bit</td>
<td>2x32/4x16/8x8</td>
</tr>
<tr>
<td>2001</td>
<td>SSE2</td>
<td>128 bit</td>
<td>2x64/4x32/8x16/16x8</td>
</tr>
<tr>
<td>2011</td>
<td>AVX</td>
<td>256 bit</td>
<td>4x64/8x32</td>
</tr>
<tr>
<td>2017</td>
<td>AVX-512</td>
<td>512 bit</td>
<td>8x64/16x32/32x16/64x8</td>
</tr>
</tbody>
</table>
ARM SCALABLE VECTOR EXTENSIONS (SVE)

- IMPDEF vector size (128-2048* bit)
- nx64/2nx32/4nx16/8nx8
- New instructions for size agnostic code
void strcpy(char *restrict dst, const char *src) {
    while (1) {
        *dst = *src;
        if (*src == '\0') break;
        src++; dst++;
    }
}

From:
**STRCPY (SVE ASSEMBLY)**

```
sve_strcpy:
  mov x2, 0
  ptrue p2.b

loop:
  setfr
  ldff1b z0.b, p2/z, [x1, x2]
  rddf1 b p0.b, p2/z
  cmpeq p1.b, p0/z, z0.b, 0
  brka p0.b, p0/z, p1.b
  st1b z0.b, p0, [x0, x2]
  incp x2, p0.b
  b.none loop
  ret
```

# header

# loop body

# set first fault register

# read ffr into p0

# break after

# function exit
PREDICATE REGISTERS

`vadd %Vd, %Vn, %Vm, %Pp`
STRCPY (SVE ASSEMBLY SETUP)

sve_strcpy:
; setup index and set p2 all true
  mov x2, 0
  ptrue p2.b

loop:
; clear first fault register, load into z0
  setffr
  ldff1b z0.b, p2/z, [x1, x2]
; did we truncate due to fault?
  rdffr p0.b, p2/z
**FIRST FAULT REGISTER**

<table>
<thead>
<tr>
<th>Page</th>
<th>Permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>(rwx)</td>
</tr>
<tr>
<td>n+1</td>
<td>(rwx)</td>
</tr>
<tr>
<td>n+2</td>
<td>( — )</td>
</tr>
</tbody>
</table>

"this is the start of a very long string of text that I want to copy that passed over several pages of memory. In fact, we might find that we are copying kilobytes at a time and we don't want to spend time checking bounds before each new page."

4.13
sve_strcpy:
 ; setup index and set p2 all true
   mov  x2, 0
   ptrue  p2.b
loop:
 ; clear first fault register, load into z0
   setffr
   ldff1b    z0.b, p2/z, [x1, x2]
 ; did we truncate due to fault?
   rdffr  p0.b, p2/z
 ; any 0's in z0.b
   cmpeq  p1.b, p0/z, z0.b, 0
   brka  p0.b, p0/z, p1.b
 ; store the string to destination
   st1b    z0.b, p0, [x0, x2]
 ; how many bytes did we copy?
   incp  x2, p0.b
 ; more?
   b.none loop
ret
RECAP

- Virtualization
  - many flavours
- Vectors
  - large registers
  - growing usage
  - data
  - parallelism
VECTORS MEET (TINY) CODE GENERATION

- QEMU's TCG Mode
- Software only virtualisation
THE X TO Y PROBLEM

- 20 guest architectures
- 7 TCG Backends
WHY CODE GENERATION?

• interpreting slow
• common processor functionality
  ▪ logic
  ▪ arithmetic
  ▪ flow control
• compiler for machine-code
CODE GENERATION

Guest Instructions (ARM)

Guest PC

Idr q0, [x21, x0]
Idr q1, [x20, x0]
tmul v0.4s, v0.4s, v1.4s
str q0, [x19, x0]
add x0, x0, #0x10 (16)
cmp x0, #0x400000
b.ne loop

TCG Micro Ops

mov i64 tmp2, x21
mov i64 tmp3, x0
add i64 tmp2, tmp2, tmp3
movi i64 tmp7, $0x8
add i64 tmp6, tmp2, tmp7
qemul ld i64 tmp5, tmp2, leq, 0
qemu ld i64 tmp5, tmp6, leq, 0
st i64 tmp4, env, $0x898
st i64 tmp5, env, $0x8a0
...

TCG Disas

TCG Opt

TCG Gen

Host Instructions (x86)

movl 0x14(%r14), %ebp
testl %ebp, %ebp
jcl 0x555559b8af
movq 0x8(%r14), %rbp
movq 0x40(%r14), %rbx
addq %rbx, %rbp
leaq 8(%rbp), %r12
movq (%rbp), %rbp
movq (%r12), %r12
movq %rbp, 0x898(%r14)
movq %r12, 0x8a0(%r14)
movq 0xe0(%r14), %rbp
addq %rbx, %rbp
leaq 8(%rbp), %rbx
movq (%rbp), %rbp
movq (%rbx), %rbx
movq %rbp, 0x8a8(%r14)
...

6.4
FLOAT MULTIPLY C CODE

```c
float *a, *b, *out;
...
for (i = 0; i < SINGLE_OPS; i++)
{
    out[i] = a[i] * b[i];
}
```
FLOAT MULTIPLY: ASSEMBLER BREAKDOWN

loop:

; load data from array
  ldr q0, [x0, x20]
  ldr q1, [x0, x19]

; actual calculation
  fmul v0.4s, v0.4s, v1.4s

; save result
  str q0, [x0, x1]

; loop condition
  add x0, x0, #0x10 (16)
  cmp x0, #0x400000 (4194304)
  b.ne loop
TCG IR: LDR Q0, [X0, X21]

Load q0 (128 bit) with value from x21, indexed by x0

; calculate offset
mov_i64 tmp2, x21
mov_i64 tmp3, x0
add_i64 tmp2, tmp2, tmp3

; offset for second load
movi_i64 tmp7, $0x8
add_i64 tmp6, tmp2, tmp7

; load from memory to tmp
qemu_ld_i64 tmp4, tmp2, leq, 0
qemu_ld_i64 tmp5, tmp6, leq, 0

; store in quad register file
st_i64 tmp4, env, $0x898
st_i64 tmp5, env, $0x8a0
TCG IR: FMUL V0.4S, V0.4S, V1.4S

; get address of fpst
movi_i64 tmp3, $0xb000
add_i64 tmp2, env, tmp3

; first fmul.s
ld_i32 tmp0, env, $0x898
ld_i32 tmp1, env, $0x8a8
; call helper
call vfp_muls, $0x0, $1, tmp8, tmp0, tmp1, tmp2
st_i32 tmp8, env, $0x898

; remaining 3 fmul.s
ld_i32 tmp0, env, $0x89c
ld_i32 tmp1, env, $0x8ac
call vfp_muls, $0x0, $1, tmp8, tmp0, tmp1, tmp2
st_i32 tmp8, env, $0x89c
...
...
## TCG Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCGv_i32</td>
<td>32 bit integer type</td>
</tr>
<tr>
<td>TCGv_i64</td>
<td>64 bit integer type</td>
</tr>
<tr>
<td>TCGv_ptr*</td>
<td>Host pointer type (e.g. cpu-&gt;env)</td>
</tr>
<tr>
<td>TCGv*</td>
<td>target_ulong</td>
</tr>
</tbody>
</table>
TCG TYPES AND TGC OPS

- TCGOOp has explicit sizes/params

```c
tcg_gen_addi_i32(TCGv_i32 ret, TCGv_i32 arg1, int32_t arg2);
tcg_gen_addi_i64(TCGv_i64 ret, TCGv_i64 arg1, int64_t arg2);
```
TYPES FOR VECTORS?

- Type for each Vector Size?
  - TCGv_i128,
    TCGv_i256...
- Type for each Vector Layout?
  - TCGv_i64x2,
    TCGv_i32x4...
PROBLEM

Each TCGType -> more TCGOps
TCG_VEC DESIGN PRINCIPLES

• Support multiple vector sizes
• without exploding TCGOp space
• Helpers dominate floating point
• avoid marshalling, pass pointers
TCG_VEC CODE GENERATION

Guest (ARM)

eor v0.16b, v0.16b, v1.16b

TCG Ops

ld_vec tmp8, env, $0x8a0, $0x1
ld_vec tmp9, env, $0x8b0, $0x1
xor_vec tmp10, tmp8, tmp9, $0x1
st_vec tmp10, env, $0x8a0, $0x1

Host (x86, SSE)

vmovdqu 0x8a0(%r14), %xmm0
vmovdqu 0x8b0(%r14), %xmm1
vpxor %xmm1, %xmm0, %xmm0
vmovdqu %xmm0, 0x8a0(%r14)
TCG_VEC GIVES US

- better code generation
- more efficient helpers
# Benchmarks (nsec/kop)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Native</th>
<th>TCG</th>
<th>TCG_vec</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytewise-xor</td>
<td>670</td>
<td>331</td>
<td>632</td>
</tr>
<tr>
<td>bytewise-xor-stream</td>
<td>235</td>
<td>330</td>
<td>450</td>
</tr>
<tr>
<td>wordwide-xor</td>
<td>1349</td>
<td>687</td>
<td>1260</td>
</tr>
<tr>
<td>bytewise-bit-fiddle</td>
<td>396</td>
<td>716</td>
<td>521</td>
</tr>
<tr>
<td>float32-mul</td>
<td>2717</td>
<td>8401</td>
<td>8665</td>
</tr>
</tbody>
</table>
BYTEWISE BIT FIDDLE: C CODE

```c
uint8_t *and, *add, *sub, *xor, *out;
...
for (i = 0; i < BYTE_OPS; i++)
{
    uint8_t value = out[i];
    value |= i & and[i];
    value += add[i];
    value ^= xor[i];
    value -= sub[i];
    out[i] = value;
}
```
BYTEWISE BIT FIDDLE: ASSEMBLY

; main loop
  mov  x0, #0x0
  mov  v1.16b, v29.16b
  add  v0.2d, v1.2d, v27.2d
  add  v17.2d, v1.2d, v26.2d
  add  v2.2d, v1.2d, v25.2d
  add  v16.2d, v1.2d, v23.2d
  add  v7.2d, v1.2d, v21.2d
  add  v20.2d, v1.2d, v24.2d
  xtn v19.2s, v1.2d
  xtn2 v19.4s, v0.2d
  add  v18.2d, v1.2d, v22.2d
  ... 
  ... 
  eor v0.16b, v0.16b, v3.16b
  sub  v0.16b, v0.16b, v2.16b
  str q0, [x19, x0]
  add  x0, x0, #0x10 (16)
  cmp  x0, #0x400000 (4194304)
  b.ne #-0x8c (addr 0x4011a0)
## Benchmarks (Nsec/Kop)

With `funroll-loops`

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>QEMU</th>
<th>QEMU TCG_vec</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte-wise-xor</td>
<td>332</td>
<td>338</td>
</tr>
<tr>
<td>byte-wise-xor-stream</td>
<td>169</td>
<td>185</td>
</tr>
<tr>
<td>word-wise-xor</td>
<td>670</td>
<td>631</td>
</tr>
<tr>
<td>byte-wise-bit-fiddle</td>
<td>661</td>
<td>469</td>
</tr>
<tr>
<td>float32-mul</td>
<td>7941</td>
<td>7634</td>
</tr>
</tbody>
</table>
FURTHER WORK

• ld/st handling
• better register liveliness
VECTORS MEET KVM*

- Xen
- HAXM (Windows)
- HVM (MacOS)
ARCHITECTURE
CPU RESOURCES

- Shared execution environment
- Virtualized resources for guest
  - Trap and Emulate
  - Context Switch
SWAPPING CONTEXT IN HOST KERNEL
SIZE OF ARMv8 CONTEXTS

- 32 x 64 bit integer regs (256 bytes)
- 32 x 2048 bit SVE regs (8192 bytes)
- 32 times bigger!
WHO USES SIMD (AND FP!)

- Userspace
  - dedicated vectorized workloads
  - accelerated library functions
- Kernel
  - Crypto
  - RAID
- Hypervisor
  - Not really
DETECTING USAGE

- Disable SIMD/FPU access
- First usage with Trap
  - swap context
  - enable SIMD/FPU
  - return to trapped insn
DEFERRED STATE BOOKEEPING

• per CPU variable
  ▪ fpsimd_last_state

• per Task Variables (task_struct)
  ▪ fpsimd_state
  ▪ TIF_FOREIGN_FPSTATE flag
VM IS MOSTLY THE SAME
ENABLING SVE ON ARM

- Kernel support in 4.15
- Enabling SVE for KVM guest
  - work in progress
SUMMARY

- Vectors are great
- Vectors are large!
- Need special handling by
  - Kernels
  - Hypervisors
  - Emulators
QUESTIONS?
BENCHMARK CODE

See:
https://github.com/stsquad/testcases/blob/master/aarch64/vector-benchmark.c