Linaro

VECTORING IN ON QEMU'S TCG ENGINE

ALEX BENNÉE

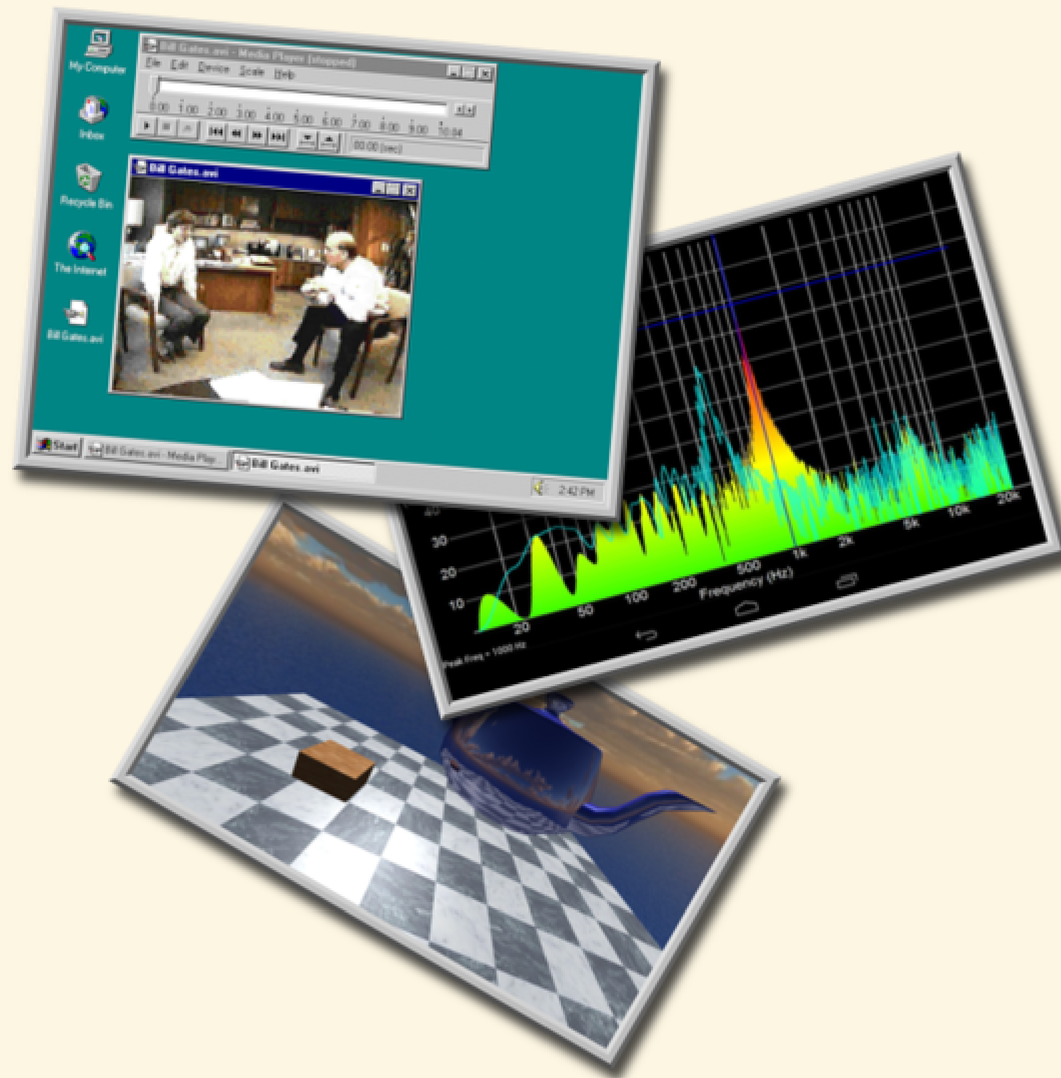
KVM FORUM 2017

Created: 2017-10-20 Fri 20:46

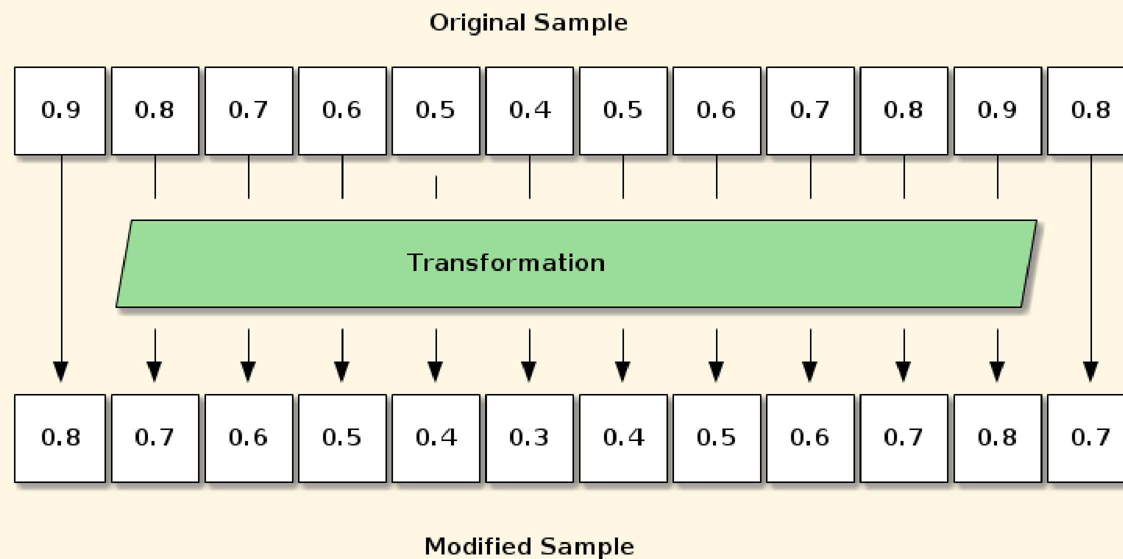
INTRODUCTION

- Alex Bennée
  - alex.bennee@linaro.org
  - stsquad on #qemu
- Virtualization Developer @ Linaro
- Projects:
  - QEMU TCG, KVM, ARM

WHY VECTORS?

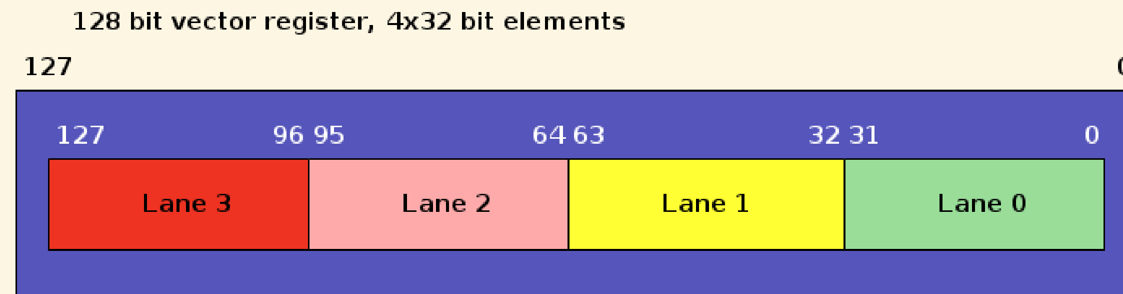


DATA PARALLELISM: AUDIO EXAMPLE



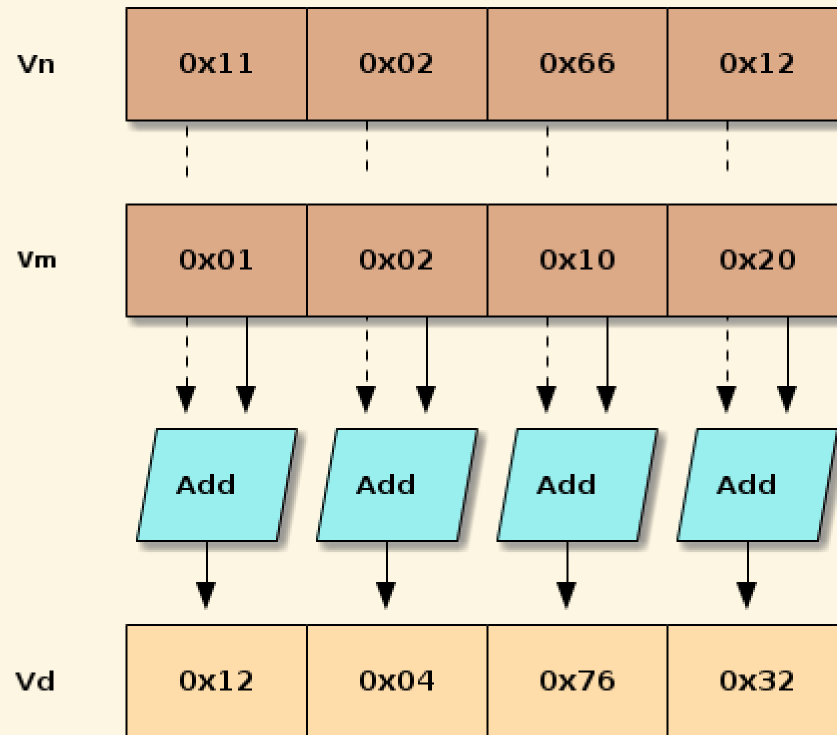
WHAT ARE VECTORS?

VECTOR REGISTER



VECTOR OPERATION

```
vadd %Vd, %Vn, %Vm
```



REGISTER LAYOUT

<table>
<tr><td>1x128</td><td colspan="16">.Q[0]</td></tr>
<tr><td>2x 64</td><td colspan="8">.D[1]</td><td colspan="8">.D[0]</td></tr>
<tr><td>4x 32</td><td colspan="4">.S[3]</td><td colspan="4">.S[2]</td><td colspan="4">.S[1]</td><td colspan="4">.S[0]</td></tr>
<tr><td>8x 16</td><td colspan="2">.H[7]</td><td colspan="2">.H[6]</td><td colspan="2">.H[5]</td><td colspan="2">.H[4]</td><td colspan="2">.H[3]</td><td colspan="2">.H[2]</td><td colspan="2">.H[1]</td><td colspan="2">.H[0]</td></tr>
<tr><td>16x 8</td><td>.B[15]</td><td>.B[14]</td><td>.B[13]</td><td>.B[12]</td><td>.B[11]</td><td>.B[10]</td><td>.B[9]</td><td>.B[8]</td><td>.B[7]</td><td>.B[6]</td><td>.B[5]</td><td>.B[4]</td><td>.B[3]</td><td>.B[2]</td><td>.B[1]</td><td>.B[0]</td></tr>
</table>

OTHER DATA PARALLELISM EXAMPLES

- General Matrix Multiplication (GEMM)
  - 3D modelling
  - AI/Machine learning
- Numerical simulation

HISTORY OF VECTORS IN COMPUTING

NAME THAT MACHINE?

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

ref: The Cray-1 Computer System, Richard M Russell, Cray Reasearch Inc, ACM Jan 1978, Vol 21, Number 1

VECTOR PROCESSORS/MACHINES

Year	Machine	Performance
1971	CDC Star-100	100 MFLOPS
1976	Cray 1	250 MFLOPS
1982	Cray X-MP	400 MFLOPS
1996	Fujitsu VP2600	5 GFLOPS
2001	NEC SX-6	8 GFLOPS/node

SX-6 (2001)

- 8 vector pipelines
- up to 72 Vector Registers
- up to 256 64 bit words per register
- 8 GFLOPS/node

NAME THAT MACHINE?

EARTH SIMULATOR (ES)

- 640 SX-6 nodes
- 5120 vector processors
- 35.86 TFLOPS
- Fastest Supercomputer 2002-2004

VECTORS COMES TO THE WORKSTATION

- Intel MMX (multi-media extensions*)
- 1997, Intel Pentium P5
- 8x64bit MMX registers
- 1x64/2x32/4x16/8x8
- Integer only

ARCHITECTURES WITH VECTORS

Year	ISA
1994	SPARC VIS
1997	Intel x86 MMX
1996	MIPS MDMX
1998	AMD x86 **3DNow!**
2002	PowerPC Altivec
2009	ARM NEON/AdvSIMD

VECTOR SIZE IS GROWING

Year	SIMD ISA	Vector Width	Addressing
1997	MMX	64 bit	2x32/4x16/8x8
2001	SSE2	128 bit	2x64/4x32/8x16/16x8
2011	AVX	256 bit	4x64/8x32
2015	AVX-512	512 bit	8x64/16x32/32x16/64x8

ARM SCALABLE VECTOR EXTENSIONS (SVE)

- IMPDEF vector size (128-2048* bit)
- nx64/2nx32/4nx16/8nx8
- New instructions for size agnostic code

TCG INTERNALS

CODE GENERATION

- decode target machine code
- intermediate form (TCG ops)
- generate host binary code

BYTEWISE EOR: C CODE

```
uint8_t *a, *b, *out;
...
for (i = 0; i < BYTE_OPS; i++)
{
    out[i] = a[i] ^ b[i];
}
```

BYTEWISE EOR: AARCH64 INNER LOOP ASSEMBLY

```
loop:
  ldr q0, [x21, x0]
  ldr q1, [x20, x0]
  eor v0.16b, v0.16b, v1.16b
  str q0, [x19, x0]
  add x0, x0, #0x10 (16)
  cmp x0, #0x400000 (4194304)
  b.ne loop
```

BYTEWISE EOR: REGISTER ALIASING

q(uad)	qN																VFP
d(ouble)	d2N								d2N+1								
vN.8b									.B[7]	.B[6]	.B[5]	.B[4]	.B[3]	.B[2]	.B[1]	.B[0]	NEON/AdvSIMD
vN.16b	.B[15]	.B[14]	.B[13]	.B[12]	.B[11]	.B[10]	.B[9]	.B[8]	.B[7]	.B[6]	.B[5]	.B[4]	.B[3]	.B[2]	.B[1]	.B[0]	

See also: target/arm/cpu.h

BYTEWISE EOR: ASSEMBLY BREAKDOWN

```
loop:
```

```
; load data from array
  ldr q0, [x21, x0]
  ldr q1, [x20, x0]
```

```
; do EOR
  eor v0.16b, v0.16b, v1.16b
```

```
; save result
  str q0, [x19, x0]
```

```
; loop conditition
  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 TYPES

Type	
TCGv_i32	32 bit integer type
TCGv_i64	64 bit integer type
TCGv_ptr*	Host pointer type (e.g. cpu->env)
TCGv*	target_ulong

TCG IR: EOR V0.16B, V0.16B, V1.16B

Compute v0.16b ^ v1.16.b into v0.16b

```
; load 1st half of v0 and v1
  ld_i64 tmp2,env,$0x898
  ld_i64 tmp3,env,$0x8a8
```

```
; do 64 xor
  xor_i64 tmp4,tmp2,tmp3
```

```
; same again for 2n halves
  ld_i64 tmp2,env,$0x8a0
  ld_i64 tmp3,env,$0x8b0
  xor_i64 tmp5,tmp2,tmp3
```

```
; store results in register file
  st_i64 tmp4,env,$0x898
  st_i64 tmp5,env,$0x8a0
```

BYTEWISE EOR: NATIVE VS QEMU

Native

```
[@aarch64] $ ./vector-benchmark -b bytewise-xor
bytewise-xor          : test took 718751 msec
                        1073741824 ops, ~669 nsec/kop
```

QEMU

```
[@corei7] $ qemu-aarch64 ./vector-benchmark -b bytewise-xor
bytewise-xor          : test took 340567 msec
                        1073741824 ops, ~317 nsec/kop
```

FLOAT MULTIPLY C CODE

```
float *a, *b, *out;
...
for (i = 0; i < SINGLE_OPS; i++)
{
    out[i] = a[i] * b[i];
}
```

FLOAT MULTIPLY ASSEMBLER

```
loop:
  ldr q0, [x0, x20]
  ldr q1, [x0, x19]
```

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

```
; store result and loop
  str q0, [x0, x1]
  add x0, x0, #0x10 (16)
  cmp x0, #0x400000 (4194304)
  b.ne loop
```

TCG IR: FMUL V0.4S, V0.4S, V1.4S

```
; get adddress of fpst
 movi_i64 tmp3,$0xb00
 add_i64 tmp2,env,tmp3
```

```
; first fmul.s
 ld_i32 tmp0,env,$0x898
 ld_i32 tmp1,env,$0x8a8
 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
 ld_i32 tmp0,env,$0x8a0
 ld_i32 tmp1,env,$0x8b0
 call vfp_muls,$0x0,$1,tmp8,tmp0,tmp1,tmp2
 st_i32 tmp8,env,$0x8a0
 ld_i32 tmp0,env,$0x8a4
 ld_i32 tmp1,env,$0x8b4
 call vfp_muls,$0x0,$1,tmp8,tmp0,tmp1,tmp2
 st_i32 tmp8,env,$0x8a4
```

VFP_MULS HELPER

```
float32 HELPER(vfp_muls)(float32 a, float32 b, void *fpstp)
{
    float_status *fpst = fpstp;
    return float32_mul(a, b, fpst);
}
```

See: target/arm/helper.c

HOST CODE TO CALL HELPER

```
; load values from env
  mov     0x898(%r14),%ebx
  mov     0x8a8(%r14),%r12d
; shuffle to C ABI and call
  mov     %ebx,%edi
  mov     %r12d,%esi
  mov     %rbp,%rdx
  callq   helper_vfp_muls
; store result
  mov     %eax,0x898(%r14)
```

WHAT ARE HELPERS USED FOR

- Complex instructions
- All floating point
- System instructions

INTRODUCING TCG_VEC

PREVIOUS WORK

- Kirill Batuzov
- {PATCH v2.1 00/20} Emulate guest vector operations with host vector operations
- Introduced TCGv_128 and TCGv_64, with MO_128
- Integer Add
  - add_i8x8, add_i16x4, add_i32x2, add_i64x1
- Early benchmarks
  - x264 video codec by 10%
  - simple vectorized for loop increased 2x

DESIGN PRINCIPLES

- Support multiple vector sizes
- Helpers dominate floating point
- Integer/Logic should still use host code

TCG_VEC

- Richard Henderson
- {PATCH v3 0/6} TCG vectorization and example conversion
- TCG_vec, pointer to guest CPU register
- Size agnostic
- Generic helpers for universal ops
- Generate code if backend supports it

GVEC_XOR HELPER

```
void HELPER(gvec_xor)(void *d, void *a, void *b, uint32_t desc)
{
    intptr_t oprsz = simd_oprsz(desc);
    intptr_t i;

    for (i = 0; i < oprsz; i += sizeof(vec64)) {
        *(vec64 *)(d + i) = *(vec64 *)(a + i) ^ *(vec64 *)(b + i);
    }
    clear_high(d, oprsz, desc);
}
```

GVEC_XOR COMPILED CODE (X86_64)

```
loop:
  ; load a
  movdqa (%rsi,%rax,1),%xmm0
  ; xor with b
  pxor   (%rdx,%rax,1),%xmm0
  ; store result
  movaps %xmm0,(%rdi,%rax,1)
  ; check loop
  add    $0x10,%rax
  cmp    %rax,%r8
  jg     loop
```

TCG IR: EOR V0.16B, V0.16B, V1.16B

Compute v0.16b ^ v1.16.b into v0.16b

```
ld_i64 tmp2,env,$0x898
ld_i64 tmp3,env,$0x8a8
xor_i64 tmp4,tmp2,tmp3
ld_i64 tmp2,env,$0x8a0
ld_i64 tmp3,env,$0x8b0
xor_i64 tmp5,tmp2,tmp3
st_i64 tmp4,env,$0x898
st_i64 tmp5,env,$0x8a0
```

TCG_VEC IR: EOR V0.16B, V0.16B, V1.16B

Compute v0.16b ^ v1.16.b into v0.16b

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

TCG_VEC CODE: EOR V0.16B, V0.16B, V1.16B

```
movdqu 0x8a0(%r14),%xmm0
movdqu 0x8b0(%r14),%xmm1
pxor   %xmm1,%xmm0
movdqu %xmm0,0x8a0(%r14)
```

BENCHMARKS (NSEC/KOP)

Benchmark	Native	QEMU	QEMU TCG_vec
bytewise-xor	672	364	615
wordwide-xor	1343	706	1283
bytewise-bit-fiddle	372	715	514
float32-mul	2700	8463	8689

FURTHER WORK

- Vector Load/Store
- ARM SVE Support
- Can we reliably interwork FP?