Personal Open source Business Explore Pricing This repository Search Sign in or Sign up

thenifty / **neon-guide**

Watch 7 Star 38 Fork 8

Code Issues 0 Pull requests 0 Projects 0 Pulse Graphs

Makes ARM NEON documentation accessible (with examples)

11 commits **1** branch **0** releases **1** contributor CC0-1.0

Branch: **master** New pull request Find file **Clone or download**

aledalgrande Fix vmlaq_n_f64 duplication and erroneous definition Latest commit `d96b046` on 6 Jun 2015

LICENSE	Update LICENSE	2 years ago
README.md	Fix vmlaq_n_f64 duplication and erroneous definition	2 years ago

README.md

NEON intrinsics guide

Makes ARM NEON documentation accessible (with examples). Born from frustration with ARM documentation and general lack of examples.

Intro

When you convert your iOS code to NEON, usually it's inside loops that can be written in parallel code. Also you have to keep in mind that the more load/store operations you have, the slower your code will be.

Assumptions

This guide is about inline *NEON intrinsics*, which should work on both 32bit and 64bit architectures. Vectors are always supposed to be of length 4, but you can generally just remove the letter *q* in the instruction name to use 2-vectors.

Syntax

Float

Arithmetic

- add: **vaddq_f32** or **vaddq_f64**

```
float32x4_t v1 = { 1.0, 2.0, 3.0, 4.0 }, v2 = { 1.0, 1.0, 1.0, 1.0 };
float32x4_t sum = vaddq_f32(v1, v2);
// => sum = { 2.0, 3.0, 4.0, 5.0 }
```

- multiply: **vmulq_f32** or **vmulq_f64**

```
float32x4_t v1 = { 1.0, 2.0, 3.0, 4.0 }, v2 = { 1.0, 1.0, 1.0, 1.0 };
float32x4_t prod = vmulq_f32(v1, v2);
// => prod = { 1.0, 2.0, 3.0, 4.0 }
```

- multiply and accumulate: **vmlaq_f32**

```
float32x4_t v1 = { 1.0, 2.0, 3.0, 4.0 }, v2 = { 2.0, 2.0, 2.0, 2.0 }, v3 = { 3.0, 3.0, 3.0, 3.0 };
```

```
float32x4_t acc = vmlaq_f32(v3, v1, v2);  // S = A + B * C
// => acc = { 5.0, 7.0, 9.0, 11.0 }
```

- multiply by a scalar: **vmulq_n_f32** or **vmulq_n_f64**

```
float32x4_t v = { 1.0, 2.0, 3.0, 4.0 };
float32_t s = 3.0;
float32x4_t prod = vmulq_n_f32(ary1, s);
// => prod = { 3.0, 6.0, 9.0, 12.0 }
```

- multiply by a scalar and accumulate: **vmlaq_n_f32** or **vmlaq_n_f64**

```
float32x4_t v1 = { 1.0, 2.0, 3.0, 4.0 }, v2 = { 1.0, 1.0, 1.0, 1.0 };
float32_t s = 3.0;
float32x4_t acc = vmlaq_n_f32(v1, v2, s);
// => acc = { 4.0, 5.0, 6.0, 7.0 }
```

- invert (needed for division): **vrecpeq_f32** or **vrecpeq_f64**

```
float32x4_t v = { 1.0, 2.0, 3.0, 4.0 };
float32x4_t reciprocal = vrecpeq_f32(v);
// => reciprocal = { 0.998046875, 0.499023438, 0.333007813, 0.249511719 }
```

- invert (more accurately): use a Newton-Raphson iteration to refine the estimate

```
float32x4_t v = { 1.0, 2.0, 3.0, 4.0 };
float32x4_t reciprocal = vrecpeq_f32(v);
float32x4_t inverse = vmulq_f32(vrecpsq_f32(v, reciprocal), reciprocal);
// => inverse = { 0.999996185, 0.499998093, 0.333333015, 0.249999046 }
```

Load

- load vector: **vld1q_f32** or **vld1q_f64**

```
float values[5] = { 1.0, 2.0, 3.0, 4.0, 5.0 };
float32x4_t v = vld1q_f32(values);
// => v = { 1.0, 2.0, 3.0, 4.0 }
```

- load same value for all lanes: **vld1q_dup_f32** or **vld1q_dup_f64**

```
float val = 3.0;
float32x4_t v = vld1q_dup_f32(&val);
// => v = { 3.0, 3.0, 3.0, 3.0 }
```

- set all lanes to a hardcoded value: **vmovq_n_f16** or **vmovq_n_f32** or **vmovq_n_f64**

```
float32x4_t v = vmovq_n_f32(1.5);
// => v = { 1.5, 1.5, 1.5, 1.5 }
```

Store

- store vector: **vst1q_f32** or **vst1q_f64**

```
float32x4_t v = { 1.0, 2.0, 3.0, 4.0 };
float values[5] = new float[5];
vst1q_f32(values, v);
// => values = { 1.0, 2.0, 3.0, 4.0, #undef }
```

- store lane of array of vectors: **vst4q_lane_f16** or **vst4q_lane_f32** or **vst4q_lane_f64** (change to **vst1...** / **vst2...** / **vst3...** for other array lengths);

```
float32x4_t v0 = { 1.0, 2.0, 3.0, 4.0 }, v1 = { 5.0, 6.0, 7.0, 8.0 }, v2 = { 9.0, 10.0, 11.0, 12.0 }, v3 =
float32x4x4_t u = { v0, v1, v2, v3 };
float buff[4];
vst4q_lane_f32(buff, u, 0);
// => buff = { 1.0, 5.0, 9.0, 13.0 }
```

Arrays

- access to values: **val[n]**

```
float32x4_t v0 = { 1.0, 2.0, 3.0, 4.0 }, v1 = { 5.0, 6.0, 7.0, 8.0 }, v2 = { 9.0, 10.0, 11.0, 12.0 }, v3 =
float32x4x4_t ary = { v0, v1, v2, v3 };
float32x4_t v = ary.val[2];
// => v = { 9.0, 10.0, 11.0, 12.0 }
```

Conditionals

- ternary operator: use vector comparison (for example **vcltq_f32** for *less than* comparison)

```
float32x4_t v1 = { 1.0, 0.0, 1.0, 0.0 }, v2 = { 0.0, 1.0, 1.0, 0.0 };
float32x4_t mask = vcltq_f32(v1, v2);  // v1 < v2
float32x4_t ones = vmovq_n_f32(1.0), twos = vmovq_n_f32(2.0);
float32x4_t v3 = vbslq_f32(mask, ones, twos);  // will select first if mask 0, second if mask 1
// => v3 = { 2.0, 1.0, 2.0, 2.0 }
```

Links

- summary of NEON intrinsics
- ARM NEON intrinsics reference

Contributing

Change README.md and send a pull request.

Author

This has been provided as part of the development that happens at Nifty.

With Nifty, the automated measurement app for easy and confident shopping, online shopping is a unique experience tailored to each shopper allowing them to buy garments with the perfect fit even on the go.