造轮子——SHA256的指令集加速

https://zhuanlan.zhihu.com/p/393097506

None Mon Aug, 09 00:31

SHA256是很常用的一种信息摘要算法，具体算法我就不讲了，因为我也不很懂（笑），可以直接看站内文章：

反正因为过于常见，很多硬件都加入了加速计算的专属指令。

常见的使用方法是直接调用hash库。

一开始我用的是cryptopp，这个库覆盖了大量密码学算法，而且很C++。

但是这个库比较重，我只是想用的sha256，不太想额外编译那么多内容。

于是我又换了headeronly的库，digestpp：

但是这个库又不带硬件加速……

那就，自己造个轮子？

首先，**x86上的SHA256加速靠的是SHA-NI[1]指令集**，13年引入。

不过intel这边一开始只是为了atom加速而引入的，core上推进比较慢，加上10nm折戟沉沙，直到icelake才支持。

AMD这边则从zen开始一直支持，反而占了个先手。

至于ARM，**armv8将NEON设为必选扩展后，顺带加入了可选的密码学扩展**[2]，因此理论上至少要armv8往后才能支持。至于是运行在aarch32还是aarch64，倒是没差。

显然，这需要先动态判断CPU是否支持。

对于x86，cpuid指令[3]已经得到了充分发展，检查特性支持很方便。我常用的库是libcpuid。

但是这个库并不支持ARM平台，而ARM这边似乎也没有很广泛且规范的特性检查指令……

按照官方blog的一篇文章，推荐办法竟然是靠系统提供的支持，比如`/proc/info`和`getauxval()`……（对windows很不友好，而且系统又是怎么获得这些信息的呢）

就算是用`getauxval()`吧，还很麻烦，aarch32和aarch64下特性支持还放在不同的HWCAP下……

综合来看，还是找同时支持x86和arm的三方库比较省事，比如Google的cpu_features：

或者pytorch的cpuinfo：

总之，运行时特性检测搞定了，接下来开始看具体实现。

天下代码一大抄，**大多数x86的sha256实现都有同一个源头**：intel介绍SHA-NI的白皮书和blog。一开始是汇编，再被人改写回intrinsic。

但直接照抄呢，好像又少了点造轮子的意思，不如针对代码做一些结构上的调整。

SHA256以512bit数据为一个block，每个block要做64轮运算，运算的单位是32bit。而SHA-NI通过使用128bit的SSE指令，不但一次性往寄存器里塞下所有数据，还做到了把4轮计算融合在一起。常见的一轮（融合了4轮）计算是这样的：

```
// Rounds 32-35
MSG = _mm_add_epi32(TMSG0, _mm_set_epi64x(W64LIT(0x53380D134D2C6DFC), W64LIT(0x2E1B213827B70A85)));
STATE1 = _mm_sha256rnds2_epu32(STATE1, STATE0, MSG);
TMP = _mm_alignr_epi8(TMSG0, TMSG3, 4);
TMSG1 = _mm_add_epi32(TMSG1, TMP);
TMSG1 = _mm_sha256msg2_epu32(TMSG1, TMSG0);
MSG = _mm_shuffle_epi32(MSG, 0x0E);
STATE0 = _mm_sha256rnds2_epu32(STATE0, STATE1, MSG);
TMSG3 = _mm_sha256msg1_epu32(TMSG3, TMSG0);
```

简单分析一下，这里一共做了三件事：

- **目标数据`TMSG0`先和固定的key相加得到`MSG`，`MSG`用于更新state。**
- **目标数据`TMSG0`和前一段数据`TMSG3`做拼接移位，再和下一段数据`TMSG1`相加，经过运算得到新的下一段数据`TMSG1`。**
- **上一段数据`TMSG3`和目标数据`TMSG0`经过处理得到新的上一段数据`TMSG3`。**

但也并不是每一轮都是如此，比如第一轮就不需要更新`TMSG`（这里的shuffle是在做LE->BE的端序转换）：

```
// Rounds 0-3
MSG = _mm_loadu_si128(CONST_M128_CAST(data+0));
TMSG0 = _mm_shuffle_epi8(MSG, MASK);
MSG = _mm_add_epi32(TMSG0, _mm_set_epi64x(W64LIT(0xE9B5DBA5B5C0FBCF), W64LIT(0x71374491428A2F98)));
STATE1 = _mm_sha256rnds2_epu32(STATE1, STATE0, MSG);
MSG = _mm_shuffle_epi32(MSG, 0x0E);
STATE0 = _mm_sha256rnds2_epu32(STATE0, STATE1, MSG);
```

由此，不需要一口气写下64轮的计算，只要写下一轮计算，再根据当前轮次决定是更新上段数据或下段数据就好了。当然运行时的判断没有必要，constexpr if就很适合：

```
template<size_t Round>
forceinline void Calc([[maybe_unused]] U32x4& prev, U32x4& cur, [[maybe_unused]] U32x4& next) noexcept // msgs are BE
{
    const U32x4 adder(SHA256RoundAdders[Round]);
    auto msg = cur + adder;
    State1 = _mm_sha256rnds2_epu32(State1, State0, msg);
    if constexpr (SHA256RoundProcControl[Round][1]) // update next
    {
        const U32x4 tmp = _mm_alignr_epi8(cur, prev, 4);
        next = _mm_sha256msg2_epu32(next + tmp, cur);
    }
    msg = msg.Shuffle<2, 3, 0, 0>(); // _mm_shuffle_epi32(msg, 0x0E);
    State0 = _mm_sha256rnds2_epu32(State0, State1, msg);
    if constexpr (SHA256RoundProcControl[Round][0]) // update prev
    {
        prev = _mm_sha256msg1_epu32(prev, cur);
    }
}
```

这里的改动比较大，用了U32x4，这是我最近在捣鼓的同时兼容SSE/AVX和NEON的向量类。

此外，更新state调用了同一个指令`_mm_sha256rnds2_epu32`两次（两个state，A-H16个uint32），但中间msg做了shuffle。其实查一查这个指令就能明白为什么了：

指令只用到了msg的低64位，每次更新A-H中和ABEF，也就是每128bit的低64位。

正因如此，**x86的state是按ABEF、CDGH这样的格式交错存放的**。

看完了x86，再看看NEON。同样是拿cryptopp的代码举例：

```
// Rounds 32-35
MSG0 = vsha256su0q_u32(MSG0, MSG1);
TMP2 = STATE0;
TMP1 = vaddq_u32(MSG1, vld1q_u32(&SHA256_K[0x24]));
STATE0 = vsha256hq_u32(STATE0, STATE1, TMP0);
STATE1 = vsha256h2q_u32(STATE1, TMP2, TMP0);
MSG0 = vsha256su1q_u32(MSG0, MSG2, MSG3);
```

每4个round也是很相似的结构，也可以按前面提到的整合一下。

但前面已经整合过了，而且我还有了兼容两者的向量库，为什么不直接替换专用于计算SHA部分的函数呢？只不过**看起来和前面提到的差异有点儿大**啊，没有shuffle，没有使用add的结果，还拿TMP2备份了一下STATE0？

仔细一看可以发现，**SHA-NI用了3条指令** `_mm_sha256msg1_epu32`，`_mm_sha256msg2_epu32`，`_mm_sha256rnds2_epu32`。**而NEON为SHA2提供了4条指令** `vsha256su0q_u32`，`vsha256su1q_u32`，`vsha256hq_u32`，`vsha256h2q_u32`。

简单对比可以看到**前两条的功能是相似的，用于处理下一轮的信息**。只不过**NEON的附带了信息拼接加法的处理，更为简洁**。

而**更新state部分neon用了两条不同的指令**，而且根据介绍，**它们的输入和计算逻辑是相同的，只是两条指令用于更新不同的state**（当然实际硬件不会做重复计算），这也解释了为什么要“备份STATE0”。

所以稍做替换就得到了NEON版：

```
template<size_t Round>
forceinline void Calc([[maybe_unused]] U32x4& prev, U32x4& cur, [[maybe_unused]] U32x4& next) noexcept // msgs are BE
{
    const U32x4 adder(SHA256RoundAdders[Round]);
    auto msg = cur + adder;
    const auto state0 = State0;
    State0 = vsha256hq_u32 (State0, State1, msg);
    State1 = vsha256h2q_u32(State1, state0, msg);
    if constexpr (SHA256RoundProcControl[Round][1])
    {
        next = vsha256su1q_u32(next, prev, cur);
    }
    if constexpr (SHA256RoundProcControl[Round][0])
    {
        prev = vsha256su0q_u32(prev, cur);
    }
}
```

不过仔细看 `vsha256hq_u32` 的定义，还会发现一个重大区别：

NEON里state是按ABCD，EFGH的格式去存储的，不需要交织。那么state初始化和最后结果输出的代码也要有所区别了。

除去核心的计算部分，一些corner case也要考虑到。

前面提到了SHA256以512bit为block做计算，那**不足512bit怎么办**?

补零呗。

那么**怎么区分256bit的数据，和256bit+256bit零的数据**呢?

SHA256的做法是，**内容后面补一个最高位为1的uint8，然后最末尾添加长度信息**（64bit，但被拆分到两个uint32）。**在此前提下再去补零补足512bit**。

最简单的做法就是拷贝出最后一段数据，然后按需设置0x80和长度信息。但如果要把这一步也向量化呢？

SSE/NEON都能一次性载入128bit，但如果不足128bit，就会留有一部分垃圾数据（会不会访存越界出错暂且不考虑）。所以**要根据长度将这部分数据置0**。

其次还要**动态设置某个字节到0x80**，这一点SSE/NEON都不能直接做到，但可以**通过动态创建mask来做bit-select**。

由于两者都和长度相关，因此可以**构造一个长度mask，跟剩余的长度做比较，得到需要清除的mask以及要设置0x80的mask**。

至于0x80的引入，常见做法是构造一个全是0x80的寄存器，SSE里可以是movd+broadcast，或者直接生成好数据load。但LLVM还有更聪明的优化。

前面提到**比较长度生成了mask，mask是特定字节全0或全1**——那**直接对这个字节做移位不就能生成0x80了**？载入操作需要访存，开销可能偏大；movd+broadcast也延迟较高。

唯一的问题就是，**SSE/AVX都没有对字节做移位的指令支持，就算你想做乘法，也不支持8bit**……

那就**只有对16bit做移位了**。0xff放在uint16里可能是 `0x00ff` 或者 `0xff00`， 左移7位会得到 `0x0780` 或者 `0x8000`。后者是我们想要的，但前者引入了不想要的 `0x07`。这时候**刚好可以跟前面的原始mask做一个and，就只保留下我们想要的字节**。

大概是这个样子：

```
forceinline U32x4 Load128With80BE(const uint32_t* data, const size_t len) noexcept
{
    // Expects len < 16
    if (len == 0)
        return U32x4::LoadLo(0x80000000);
    const auto val = U32x4(data).SwapEndian();
    const U8x16 IdxConst(3, 2, 1, 0, 7, 6, 5, 4, 11, 10, 9, 8, 15, 14, 13, 12);
    const U8x16 SizeMask(static_cast<char>(len));                                                   // x, x, x, x
    const auto EndMask   = SizeMask.Compare<CompareType::Equal,       MaskType::FullEle>(IdxConst); // 00,00,ff,00 or 00,00,00,ff
    const auto KeepMask  = SizeMask.Compare<CompareType::GreaterThan, MaskType::FullEle>(IdxConst); // ff,ff,00,00 or ff,ff,ff,00
    const auto EndSigBit = EndMask.As<U16x8>().ShiftLeftLogic<7>().As<U8x16>();                     // 00,00,80,7f or 00,00,00,80
    const auto SuffixBit = EndSigBit & EndMask;                                                     // 00,00,80,00 or 00,00,00,80
    return SuffixBit.SelectWith<MaskType::FullEle>(val.As<U8x16>(), KeepMask).As<U32x4>();
}
```

如果是NEON的话，因为支持单个byte的shift，会更简单一些。

至此，SHA256的轮子就造得差不多了。

再找几个testvector写写测试吧：

参考

1. ^sha-ni https://en.wikipedia.org/wiki/Intel_SHA_extensions
2. ^neon-armv8 https://community.arm.com/developer/tools-software/oss-platforms/b/android-blog/posts/arm-neon-programming-quick-reference#_edn6
3. ^cpuid https://en.wikipedia.org/wiki/CPUID