工作量证明方案

本项目的目标是设计一种在 CPU 上可以较为高效工作，但是在 GPU、FPGA、ASIC 上却难以高效实现的工作量证明方案。即设计一个单向函数 H，满足 $y=CryptoHello(x)$。*CryptoHello* 应满足的要求类似于散列函数，即单向性、雪崩性、随机性，其计算效率在 CPU 上较高，但是在 GPU、FPGA、ASIC 上难以发挥出性能。

一、总体技术方案

1.1 基本原则

为了防止 GPU、FPGA、ASIC 上的高效实现，*CryptoHello* 的设计需要满足以下特征：

（1）内存受限。需要较大空间的内存容量，而且对内存有大量的随机访问。考虑到 CPU 和 GPU 的 cache 容量差异，内存容量大约在 1MB 左右。

（2）高度串行化。计算过程中基本没有可以并行执行的成分，必须严格按照串行执行；

（3）大量的控制相关性。即 *CryptoHello* 中需要执行比较多的条件分支指令，而且这些分支转移的概率均接近 50%；

（4）组合多种的散列函数。这主要是防止 ASIC 芯片中针对每种散列函数专门定制设计。如果是有很多种散列函数，将大幅度增加 ASIC 芯片的芯片面积。

1.2 总体技术方案

【定义 1】单向函数族 h_i，$0 \le i \le 15$，共计 16 种不同的单向函数。每个单向函数的输入固定为 256 位（h_0 的输入可以是任意长度），输出固定为 256 位。

【定义 2】工作存储器 M，总容量为$|M|$字节，$|M|$能被 32 整除，目前设置在为 1M。

【定义 3】伪随机函数发生器 *seed*(uint48 s)为设置随机数发生器种子，*rand*()为随机数发生器结果，返回 48 位无符号整数。该函数满足 $X_{n+1} = (aX_n + c) \bmod m$，$n \ge 0$，其中 $m = 2^{48}$，a =25214903917,c = 11（glibc 的 rand48()线性同余伪随机数发生器）

【定义 4】单向函数 $y=H(x)$，其中 y 为 256 位。

【定义 5】*reduce_bit*(*x*,*y*)将 x 的内容规约到 y 位。设 x 的位数为 l，应满足 $l \ge y$。

将 x 的长度补零扩展到 y 的整数倍，得到长度为 L 的二进制串 X，然后再将 X 分为 L/y 段长度为 y 的段，将这 L/y 段内容异或得到最终结果。

【定义 6】*RRS*(*x*, *y*)将 x 循环右移 y 位。

算法分为三步骤：1）初始化工作存储器 M；2）修改工作存储器 M 内容；3）根据工作存储器内容产生最后的结果。

【算法-1】初始化工作存储器 M

输入：x 单向函数 $y=H(x)$的输入

输出：M 工作存储器

参数：K 用于调整计算速度，K 越大计算速度越快

中间变量：

a,b 均为长度为 32 个字节的向量，$a[c_1:c_2]$表示其中的第 c_1:c_2 个字节；

i 为 32 位无符号整数；

1.$a[0:31]=h_0(x)$;

2. i 从 0 到$|M|/32$-1 循环　　//如果$|M|$为 1M，循环 2^{15} 次

2.1 如果 $i \bmod K \neq 0$　　//K 是一个可以选择的参数，用于调整产生速度

2.1.1　$b[0:7] = rand_0() \oplus (rand_0() << 16)$;

2.1.2　$b[8:15] = rand_1() \oplus (rand_1() << 16)$;

2.1.3　$b[16:23] = rand_2() \oplus (rand_2() << 16)$;

2.1.4　$b[24:31] = rand_3() \oplus (rand_3() << 16)$;

2.1.5　$b[0:31] = RRS(b[0:31], reduce\_bit(i,8))$;

2.1.6　$M[32*i:32*i+31]=b[0:31]$;

2.1.7　$a[0:31] = a[0:31] \oplus b[0:31]$;

2.2 否则

2.2.1　$t=reduce\_bit(a[0:31],4)$;

2.2.2　$a[0:31]=h_t(RRS(a[0:31], reduce\_bit(i,8)))$;

2.2.3　$seed_0(reduce\_bit(a[0:7],48))$;　$seed_1(reduce\_bit(a[8:15],48))$;

　　$seed_2(reduce\_bit(a[16:23],48))$;　$seed_3(reduce\_bit(a[24:31],48))$;

2.2.4$M[32*i:32*i+31]= a[0:31]$;

算法说明

本算法的目的是产生一个容量大小为$|M|$的伪随机内存内容。算法第 2 步中，每次循环每次产生 32 字节内容，其中连续 K-1 次由随机数发生器产生（2.1 步），1 次由单向函数产生（2.2 步）。单向函数的计算速度较慢，而随机数发生器的填充速度较快，通过增加 K 可以减少单向函数的调用时间，提升填充速度。

在随机数填充部分（2.1.1 到 2.1.4 步）中，有四路 48 位独立设置种子的随机数产生器，它们的结果合成一个 64 位的伪随机数。在合成 64 位随机数后，再使用不定长循环移位方法增加预测产生内容的难度。确定 32 字节随机数块的因素有两个：四个 48 位的随机数种子和 2^{15} 种循环变量 i 的可能性，因此决定这 32 字节内容的输入可能性总共有 $2^{96+15}=2^{111}$。这使得难以通过预计算的方法，预先确定每个存储块的内容。

调用单向函数族时（2.2.2 步），其输入内容不仅仅依赖于上一次单向函数的输出，而且依赖于前面 K-1 个存储块的内容（2.1.7 步）这使得单向函数的计算必须在随机数填充完成后才能进行。单向函数输入的内容依赖于 256 位的数值和 256 种移位数，其可能性达到 $2^{256+8}=2^{264}$，从而使得 2.2.2 步难以通过预计算方式存储。

2.2.3 步将为后续的随机数填充设定新的种子，这使得后续的随机数填充块必须在单向函数计算完成之后才能继续。

从上述分析可以看出，该方法满足以下特征：1）必须严格顺序执行；2）决定每 32 字节存储块内容的因素都足够大，难以通过预计算方法产生。

算法-1 流程图

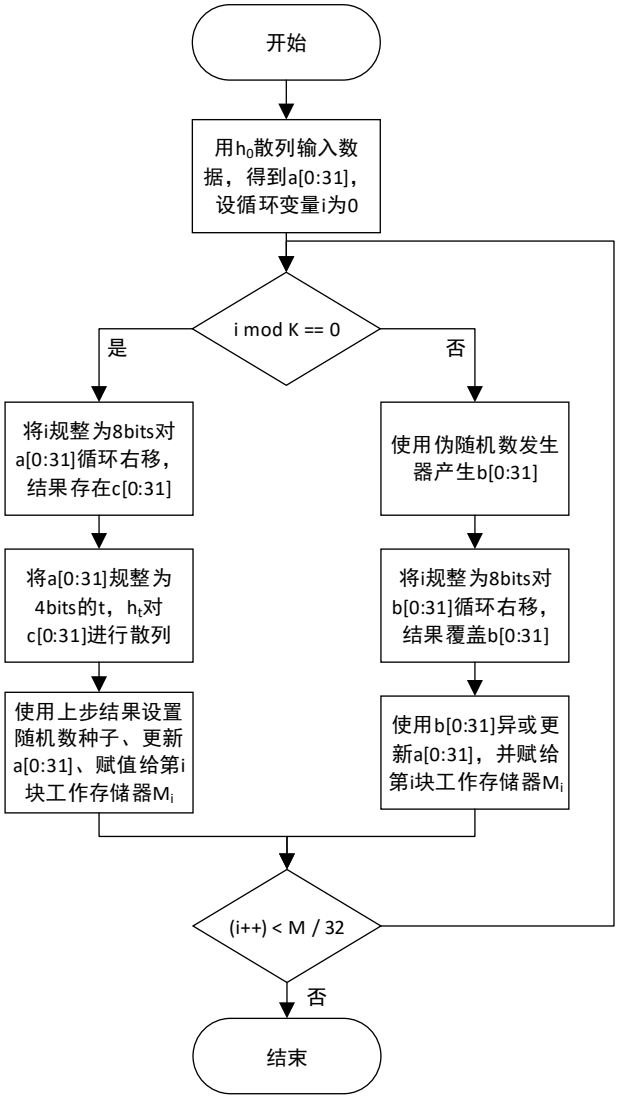

图 1-1 算法-1 流程图

【算法-2】修改工作存储器内容

输入：*MM*[*addr*]表示地址为 *addr* 的一个字节

参数：*L* 用于平衡存储器访问和散列函数计算之间的时间比例，目前设置为 1

C 用于调整计算轮数，不应该小于|*M*|/128

输出：*M*

c 32 字节数组，用于向下一个算法传递随机变量

中间变量：*s*,t_1, t_2 为 8 位无符号整数，*i* 为 32 位无符号整数，*r* 为 64 位无符号整数，*b* 为 64 字节数组，*a* 为 32 字节数组

1.*a*[0:31]=*c*[0:31]=h_0(*M*[|*M*|-32:|*M*|-1]); *r*=*reduce_bit*(*a*[0:31], 64); //对最后一块进行散列

2.*i* 从 0 到 *C* 次循环

2.1 *seed*(*reduce_bit*(*a*[0:31], 48)); //初始化随机数发生器

2.2 *j* 从 0 到 64**L*-1 循环 //修改存储器内容，并产生新的散列函数输入

2.2.1 *base*=(*rand*()+*r*) mod 2^{64}; *offset*= (*reduce_bit*(*r*,8)<<8)+1; //产生随机地址

2.2.2 $addr_1$=(*base*-*offset*) mod |*M*|; $addr_2$=(*base*+*offset*) mod |*M*|

2.2.3 t_1=*M*[$addr_1$]; t_2=*M*[$addr_2$]; *s*=*a*[*j* mod 32];

2.2.4 *M*[$addr_1$]=t_2⊕*s*; *M*[$addr_2$]=t_1⊕*s*; *b*[*j* mod 64]=t_1⊕t_2; //修改工作存储器

2.2.5 *r*=(*r*+*s*+t_1+t_2) mod 2^{64};

2.3 *t*=*reduce_bit*(*r*,4);

2.4 *a*[0:31]=*reduce_bit*(*b*[0:63], 256);

2.5 *a*[0:31]=h_t(*RRS*(*a*[0:31], *reduce_bit*(*i*+*r*,8)));

2.6 *c*[0:31]=*c*[0:31]⊕*a*[0:31]

算法说明

算法的第 1 步获取 *M* 的最后一块内容进行散列函数计算，这就需要完整地完成算法-1 之后才能进行本算法，保证了两个算法的顺序性。

算法的核心是第 2 步，其主要目的是随机化地修改 *M* 中的内容。分为两个阶段：1）根据参数随机地修改存储器内容（2.1~2.2 步）；2）根据排序后的内容使用单向函数族，重新产生存储器修改参数。

在存储器修改阶段，每次修改 2 个字节的内容，其地址的产生由上一次单向函数的结果 *a* 和 *r* 共同决定。其中 *r* 的修改（2.2.5 步）依赖于前面所有存储器访问的值和单向函数的结果。2.2 步的每次循环中，会产生两个存储器访问地址，它们的差距是 2×*offet*+2。由于 *offset* 的低 8 位为 0，使得在 *offset* 不等于 0 的情况下，两者的差距在 512 字节以上，此时这两次存储器访问不会落在同一个 cache 行中。当 *reduce_bit*(*r*,8)大于 4 时，两者的差距在 2K 以上，而 2KB 往往是 DRAM 的一个存储行大小。这使得两次存储器访问很大可能不是在一个存储行中，但是也存在着在一个存储行中的可能性，以增加 ASIC 存储器系统设计的复杂性。2.2.4 步中采用异或方法将两个存储器的内容和单向函数的内容进行相互交叉叠加，保持了原有内容的随机性。

2.5 步的单向函数输入取决于当次循环的存储器访问内容，当前循环次数 *i*，以及 *r*（包含了该算法前面执行过程的历史信息）。

2.6 步将所有单向函数的执行信息融合在 *c* 中，作为后续算法的随机数输入。

算法-2 流程图

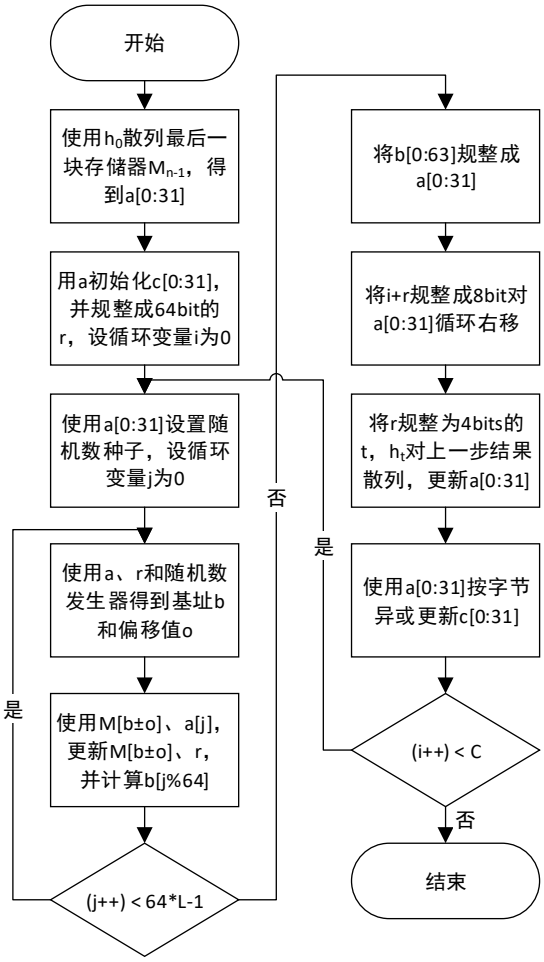

图 1-2 算法-2 流程图

【算法-3】根据工作存储器内容产生最后结果

输入：M 工作存储器

c 上一级算法的随机数种子

参数：D 用于调整单向函数的计算次数。D 越大，单向函数的计算次数越少，速度越快，一般设置为素数

输出：y

1.y[0:31]=c[0:31]; i=0;

2.循环

2.1t= *reduce_bit*(y[0:31],4);

2.2 d=*reduce_bit*(y[0:31], D)+1;

2.3 j 从 0 到 d 循环 //至少循环一次

2.3.1y[0:31]=y[0:31]⊕M[32*i, 32*i+31];

2.3.2i++;

2.3.3 如果 i=|M|/32-1，则

2.3.3.1y[0:31]=h_0(*RRS*(y[0:31], *reduce_bit*(i+t,8)));算法结束

2.4y=h_t(*RRS*(y[0:31], *reduce_bit*(i+t,8)));

算法说明

该算法的主要目的是快速地基于存储器 M 的内容产生最后结果，以 32 字节为单位一段顺序访问存储器 M。2.3 步中将上次单向函数的结果与 d 段存储器的内容通过异或方式叠加。d 是依赖于上次单向函数结果，且处于 0~2^D-1 之间的随机值。这里存在着并行计算的可能性，但是由于 2.3 步中起始的 i 和 d 都无法事先预知，无疑会增加判断的难度。

2.4 步中单向函数计算加入了循环变量 i 的因素，使得其输入不仅仅依赖于 y，而且和循环的位置相关。通过 2.3.1 步，单向函数的输入蕴含了所有存储器内容的信息。

2.3.3.1 和 2.5 步决定了在算法的最后一步必须执行单向函数 h_0。

算法-3 流程图

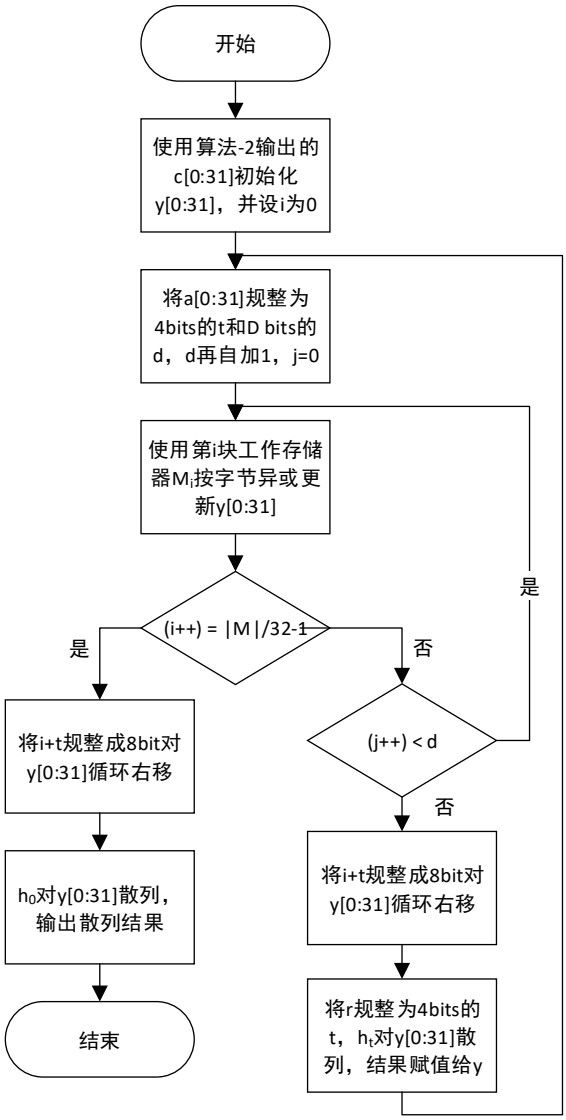

图 1-3 算法-3 流程图

1.3 算法操作分析

在特定参数的设置下，这三步算法的操作数量如表 1-1 所示。

表 1-1 单向函数 *CryptoHello* 的操作数量

算法	单向函数调用次数	存储器访问
算法 1	$\|M\|/32K$ 次	$\|M\|/32$ 次 256 位存储器写入
算法 2	C 次	$128CL$ 次 8 位存储器读出 $128CL$ 次 8 位存储器写入
算法 3	$\|M\|/(32*2^{D-1})$次	$\|M\|/32$ 次 256 位存储器读出

参数设置： $|M|$=1M, K=128, L=4,C=512,D=8

1.4 单向函数族

我们使用了如表 1-2 所示的单向函数族。表 1-3 给出了各种单向函数的实现细节。

表 1-2 单向函数族

t	类别	函数名	代码来源
0	SHA-3	SHA3-256	OpenSSL
1	SHA-1	SHA-1	OpenSSL
2	SHA-2	SHA-256	OpenSSL
3		SHA-512	OpenSSL
4	Whirlpool	Whirlpool	OpenSSL
5	RIPEMD	RIPEMD-160	OpenSSL
6	BLAKE2	BLAKE2s(256bits)	OpenSSL
7	AES	AES(128bits)	OpenSSL
8	DES	DES	OpenSSL
9	RC4	RC4	OpenSSL
10	Camellia	Camellia(128bits)	OpenSSL
11	CRC	CRC32	JTR
12	HMAC	HMAC(MD5)	OpenSSL
13	GOST	GOST R 34.11-94	JTR
14	HAVAL	HAVAL-256/5	JTR
15	Skein	Skein-512(256bits)	JTR

表 1-3 各种单向函数的实现细节

```
1. 1. SHA1
2. $hash[ 0:19] = sha1($input)
3. $hash[20:39] = sha1(~$input)  // ~为按字节取反
4. $output[0:31] = reduce_bit(hash[0:39], 256)
5.
6. 2. SHA256
```

```
$output[0:31] = sha256($input)

3. SHA512
$hash[0:63] = sha512($input)
$output[0:31] = reduce_bit(hash[0:63], 256)

4. SHA3-256
$output[0:31] = sha3-256($input)

5. Whirlpool
$hash[0:63] = whirlpool($input)
$output[0:31] = reduce_bit($hash[0:63], 256)

6. RIPEMD-160
$hash[ 0:19] = ripemd160($input)
$hash[20:39] = ripemd160(~$input)
$output[0:31] = reduce_bit(hash[0:39], 256)

7. BLAKE2s(256bits)
$output[0:31] = blake2s($input)

8. AES(128bits)
$hash[0:31] = sha256($input)
$hash2[0:15] = md5($hash[0:31])
$key = aes128_set_key($hash2[0:15])
$output[ 0:15] = aes128_encrypt($key, $hash[ 0:15])
$output[16:31] = aes128_encrypt($key, $hash[16:31])

9. DES
$hash[0:31] = sha256($input)
$hash2[0:15] = md5($hash[0:31])
$key = DES_set_key_unchecked($hash2[0:15])
$output[ 0: 7] = DES_encrypt($key, $hash[ 0: 7])
$output[ 8:15] = DES_encrypt($key, $hash[ 8:15])
$output[16:23] = DES_encrypt($key, $hash[16:23])
$output[24:31] = DES_encrypt($key, $hash[24:31])

10. GOST R 34.11-94
$output[0:31] = gost($input)

11. HAVAL-256/5
$output[0:31] = haval5_256($input)

12. Skein-512(256bits)
```

```
$output[0:31] = skein512_256($input)

13. CRC32
$hash[0:31] = sha256($input)
$output[0:31] = crc32($hash[0:31]) // 按字进行 冗余校验码

14. HMAC(MD5)
$hash[0:15] = hmac_md5($input, $input) // HMAC_MD5 的 key 和 message 均为输入
$output[0:31] = sha256($hash[0:15])

15. Camellia(128bits)
$hash[0:31] = sha256($input)
$hash2[0:15] = md5($hash[0:31])
$key = Camellia_set_key($hash2[0:15])
$output[ 0:15] = Camellia_encrypt($key, $hash[ 0:15])
$output[16:31] = Camellia_encrypt($key, $hash[16:31])

16. RC4(256bits)
$hash[0:31] = sha256($input)
$hash2[0:15] = md5($hash[0:31])
$key = RC4_set_key($hash2[0:15])
$output[0:31] = RC4($key, $hash[0:31])
```

1.5 单向函数 *CryptoHello* 测试

单向函数 *CryptoHello* 的标准输出：

CryptoHello(“0123456789”)=
cb98c372548618317a2dc286a7481701e5ea94892c9eb371d932c83d94ddd459

CryptoHello(“HelloWorld”)=
8d184a295c91aa46243c64452c0417fcff4d5ea67b30d43dd1e5a358171b9929

对单向函数 *CryptoHello* 执行 10^6 次。第 1 次输入为空值，第 i 次的输入 x_i 为上次输出 y_{i-1}，输出为 y_i。将 y_i 按照顺序拼接在一起，形成 32M 字节的数据流 Y。

对 Y 执行 NIST 的随机数测试工具集[1]，进行频数测试和游程测试。结果如表 1-3 所示，可以看到频率和游程测试的 P 值均大于 0.01，所以生成的序列是随机的。

表 1-3 单向函数 H 的随机性检查结果

测试项	**P - value**
频率测试（Frequency）	0.859141
游程测试（Runs）	0.876262

[1]https://csrc.nist.gov/Projects/Random-Bit-Generation/Documentation-and-Software

二、软件性能评测与分析

2.1 CPU 软件设计

CPU 平台

我们使用了服务器、PC 机、服务器 X 和嵌入式系统等四种，具体参数如表 2-1 所示。

表 2-1CPU 测试平台

硬件平台	平台 1	平台 2	平台 3	平台 4
	服务器	PC 机	服务器 X	嵌入式系统
处理器型号	E5-2609	Core i5 7500	E5-2692 v2	TX1
微结构	Haswell	Kaby Lake	Ivy Bridge	A57
SIMD 支持	SSE 2/3/4.1/4.2 AVX/AVX2	SSE 2/3/4.1/4.2 AVX/AVX2	SSE 2/3/4.1/4.2 AVX	NEON
物理核数	2*6	4	2*12	4
超线程	不支持	不支持	支持	不支持
主频	1.9 GHz	3.4 GHz	2.20GHz	2.1GHZ
主存容量	64 GB	8 GB	64GB	2GB
L1 Cache（每核）	32 KB+32 KB	32KB+32 KB	32 KB+32 KB	32 KB+32 KB
L2 Cache（每核）	256 KB	256KB	256 KB	2MB（共享）
L3 Cache	15MB	6MB	30MB	
操作系统	CentOS 6.6（64 bits）	Windows 10（64 bits）	Red Hat Enterprise 6.2	Ubuntu 14.04 LTS
编译器	gcc7 (GCC) 7.1.0	Microsoft VS2015 C/C++ 19.00.24215.1	icc 14.0.2	

单向函数族测试

在测试平台 1、2、3 上分别对单向函数族的性能进行了测试，如表 2-2 所示。

表 2-2 单向函数族的性能（单位：Mps，即每秒完成单向函数计算的次数）

t	类别	函数名	平台 1	平台 2	平台 3
0	SHA-3	SHA3-256	1.01	1.28	1.02
1	SHA-1	SHA-1	2.15	4.51	3.19
2	SHA-2	SHA-256	**2.81**	**5.26**	**3.72**
3		SHA-512	1.34	3.68	2.54
4	Whirlpool	Whirlpool	0.92	2.43	1.50
5	RIPEMD	RIPEMD-160	0.95	1.51	1.46
6	BLAKE2	BLAKE2s(256bits)	1.66	2.42	2.48
7	AES	AES(128bits)	0.92	1.79	1.28
8	DES	DES	0.66	1.25	0.91
9	RC4	RC4	0.72	1.42	1.06
10	Camellia	Camellia(128bits)	1.07	2.04	1.52
11	CRC	CRC32	2.16	3.98	2.94

12	HMAC	HMAC(MD5)	0.47	**0.66**	0.67
13	GOST	GOST R 34.11-94	**0.37**	0.73	**0.50**
14	HAVAL	HAVAL-256/5	1.67	2.43	1.81
15	Skein	Skein-512(256bits)	2.17	4.47	3.17

从上表可以看出，吞吐率最高的算法为 SHA256，最低的算法为 GOST 或者 HMAC。吞吐率最高者是最低者的 7.5~8.0 倍左右。

执行时间分布

在平台 1 上使用单核的情况下，单向函数 *CryptoHello* 的执行时间分布如表 2-3 所示。

表 2-3 单向函数 *CryptoHello* 的执行时间分布

步骤	算法 1 初始化存储器	算法 2 的 2.2 步 修改存储器	算法 2 的 2.5 步 单向函数族计算	算法 3 产生最后结果
比例	36%	53%	1%	10%

可以看出单向函数 *H* 中的主要计算开销在于算法 2 中的修改存储器内容。

性能

单向函数 *CryptoHello* 性能与线程数的关系如表 2-4 和图 2-1 所示。其中每个平台的最优性能如黑体所示，恰为对应平台的核数。

表 2-4 不同平台上单向函数 *CryptoHello* 的性能（单位：每秒的计算次数）

平台	线程数								
	1	**4**	**8**	**12**	**16**	**24**	**32**	**48**	**64**
服务器	80	315	628	**938**	837	836	867	858	763
PC 机	168	**641**	634	631	633	629	627	625	625
天河二号	115	443	797	1186	1578	**2337**	2208	2177	2171
嵌入式系统	73	**138**	133	133	133	133	133	132	133

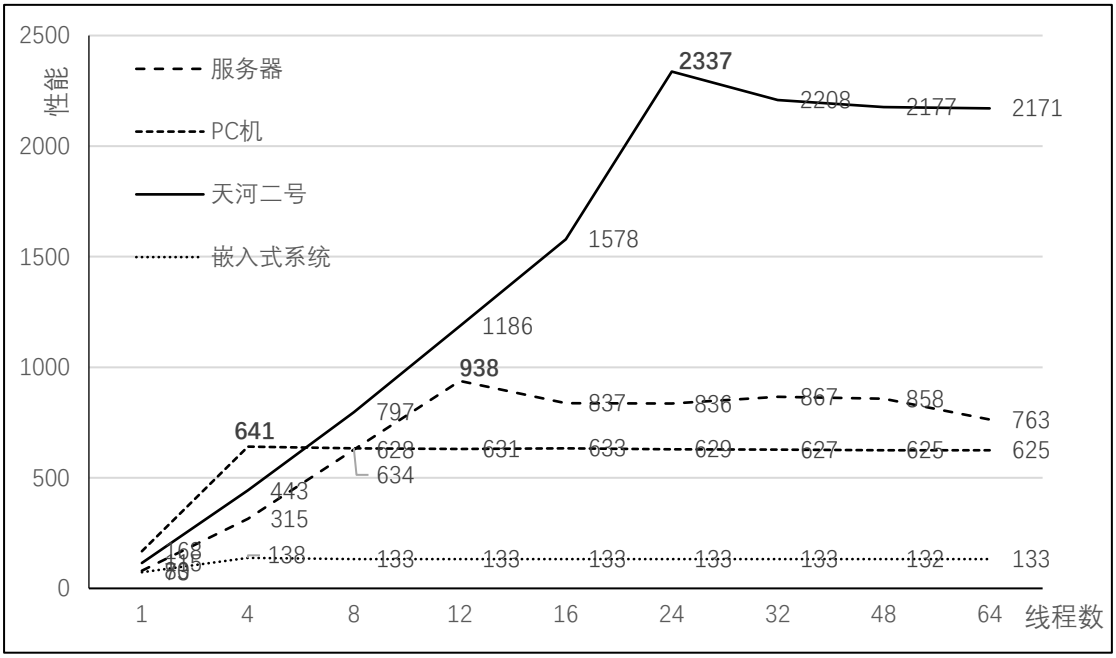

图 2-1 不同平台上单向函数 *CryptoHello* 的性能（单位：每秒的计算次数）

2.2 GPU 软件设计

GPU 平台选择当前主流的消费级显卡 GTX1080、 GTX Titan X 和 RX vega64，具体参数如表 2-5 所示。

表 2-5GPU 主要参数

显卡型号	GTX 1080	GTX Titan X	RX VEGA 64
架构	Pascal	Maxwell	Vega
核数	2560	3072	4096
主频(GHZ)	1.6	1.0	1.4
显存容量(GB)	8	12	8
显存宽度(bit)	256	384	2048
显存峰值带宽(GBps)	320	336	512
功耗(W)	180	250	295

1、线程数约束

由于显存容量有限，因此在一块 GPU 上最多同时执行 4K 个散列函数，占据显存 4K*1MB=4GB。

在 GPU 上的一个重要选择是多个线程同时执行一个单向函数 *CryptoHello*，还是一个线程执行一个单向函数。我们认为上述三个主要算法内部缺乏明显的并行性，难以实现多个线程同时执行一个单向函数，只能采用一个线程一个单向函数的执行方式。此时，全 GPU 系统中共计有 4K 个线程。这将导致限制 GPU 性能的第一个问题：**可以并行执行的线程数量太少，整体并行性有限。**

2、SIMT 约束

GPU 采用的是 SIMT（单指令多线程）方法，一个 Warp（=32）个线程执行同一条指令。在具有分支指令的情况下，如果线程之间的执行路径不同，则需要分别**串行**执行每个线程的所有路径，直至一个 Warp 中的所有线程执行到相同路径为止。

在上述算法中，不同输入的单向函数 H 需要从 16 个单向函数中近乎随机地选择一个执行。32 个线程同时能选择同一个单向函数的概率为 $\left(1/16\right)^{31}$ =4.7×10^{-38}，严重限制了 GPU 多线程并行性的发挥。

在算法-3 的 2.3 步中，循环的次数根据输入数据的不同在 1~2^{D}-1 之间变化，这也将导致一个 Warp 的线程执行时间依赖于最长的循环次数。

3、存储器访问约束

在算法-2 中，对工作存储器 M 的访问是以字节为单位，而且地址随机性强。由于 GPU 片上 Cache 容量非常有限，这使得算法-2 中的存储器访问基本上需要从显存中读取。虽然 GPU 显存的访问宽度高达 256 位（甚至 384 位），但是真正有效的仅仅只有 1 个字节，这使得 GPU 存储器访问带宽仅仅只有 3%的利用率。

由于大量依赖于显存访问，使得访存指令的延迟长达数百个周期，而且由于线程数量很少，难以掩盖这些访存指令的长延迟。

性能

GPU 版本的代码实现中支持以下几个动态调整功能：

1）根据机器中显卡个数动态创建多线程进行计算；

2）多线程互斥获取输入数据并行计算，支持任何规模的输入数据；

3）根据显卡的显存大小，动态设置 workitem 数目。

表 2-6 和图 2-2 给出了三种 GPU 卡的性能，输入数据从 1K~1000K 共测试 7 种数据规模，经过系统测试发现每次 kernel 计算的线程设置为 1024 时性能最优。

表 2-6 不同 GPU 上单向函数 *CryptoHello* 的性能

（单位：每秒的计算次数，单卡，线程设置为 1024）

GPU	同时运行的单向函数数量						
	1K	2K	3K	4K	10K	100K	1000K
GTX1080	538	538	538	538	536	532	532
GTX Titan X	538	538	538	538	533	527	525
Vega(windows)	409	455	438	455	445	449	443

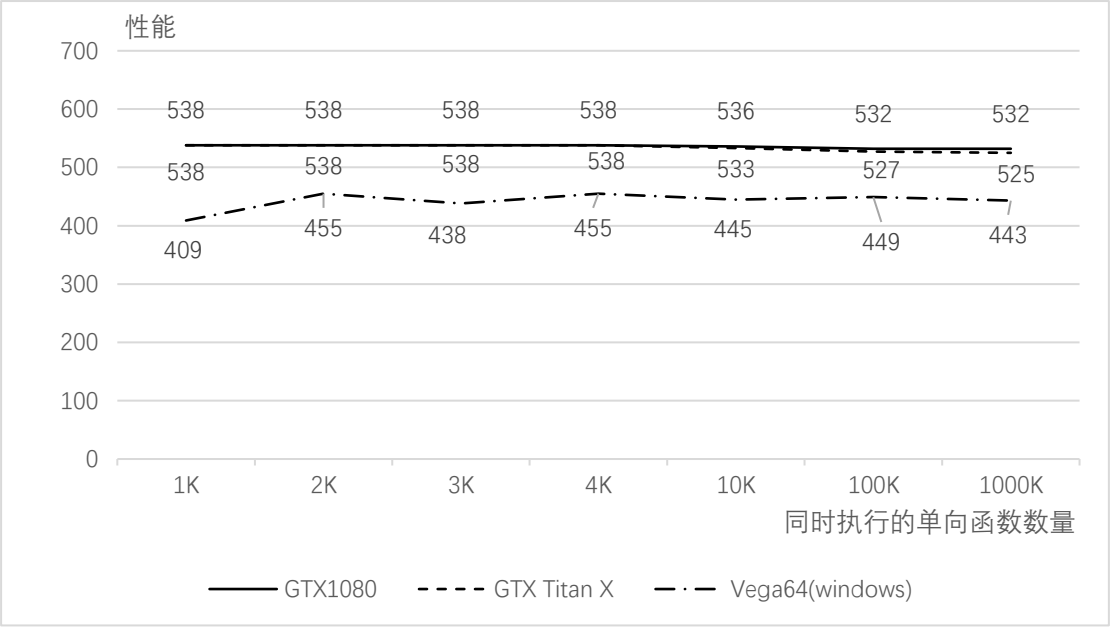

图 2-2 不同 GPU 上单向函数 *CryptoHello* 的性能

图 2-2 中，GTX 1080 和 Titan X 的性能接近，而 Vega64 的性能明显较低。我们认为，这主要是因为我们的算法是存储器带宽受限的，而与核数关系不大。由于在算法-2 中的存储器访问基本单位是字节，因此，虽然 GPU 的显存位宽很高，但是真正有效的数据传输率却非常低。三种 GPU 的存储器系统相关参数如表 2-7 所示。

表 2-7 GPU 存储器系统的主要参数

显卡型号	GTX 1080	GTX Titan X	RX VEGA 64
显存宽度(bit)	256	384	2048
显存峰值带宽(GBps)	320	336	512
字节访问时显存带宽利用率	3.13%	2.08%	0.39%
字节访问时有效显存带宽(GBps)	10	7	2

由此可以看出，虽然 VEGA64 的核数最多且显存峰值带宽最高，但是其在算法-2 中的

有效存储器带宽却是最低的。由于算法-2 在整个计算过程中占据了一半以上的时间，所以VEGA64 的性能反而较低。GTX 1080 和 Titan X 的有效显存带宽相近，所以两者的性能基本相同。

2.3 总结

表 2-8 对比了 CPU 和 GTX 1080 两者的最佳性能。可以看出，虽然显卡核数和存储器峰值带宽远远高于 CPU，但是由于本算法的设计，显卡的性能较一般 PC 机还有一定差距。而 CPU 的性能与其核数又成正比关系，核数越多性能越高。

表 2-8 CPU 和 GPU 上单向函数 *CryptoHello* 的性能对比（单位：每秒的计算次数）

分类	平台	最佳性能
CPU	服务器 X	2337
	服务器	938
	PC	641
GPU	GTX 1080	538

三、ASIC 设计评估

3.1 ASIC 加速芯片体系结构分析

3.1.1 ASIC 加速芯片的典型结构

典型的支持对称加解密运算和 Hash 运算的 ASIC 芯片，其结构如图 3-1 所示，分为水平方向上的以 CPU 为核心的控制流，和垂直方向上的运算数据流。

在垂直方向上，片外大容量存储器、片内高速缓存和加速运算模块阵列一起构成了实现加速运算的核心数据流。由于实现对称加解密运算和 Hash 运算的加速运算模块之间具有简单而规则的数据并行性，且单个加速运算模块的电路规模很小，仅为 1K 至 2K 门，因此这种 ASIC 加速芯片的设计重点是充分利用和扩展数据处理的并行性，从而获得更好的加速效果，具体措施有：

1、扩展片内高速数据缓存到片外大容量数据存储的带宽。这主要是通过采用更宽的数据总线和更多的芯片引脚来实现的（但同时也引入了昂贵的封装管壳开销）；

2、扩展片内的数据并行处理能力。这主要是通过在片内加入更多的加速运算模块，同时加速运算模块以并行运算的方式来提升处理能力。由于对称加解密运算和 Hash 运算的核心组合运算，这些运算的复杂度低，而且运算执行往往可以在一拍内完成，阵列式结构可以取得好的效果。

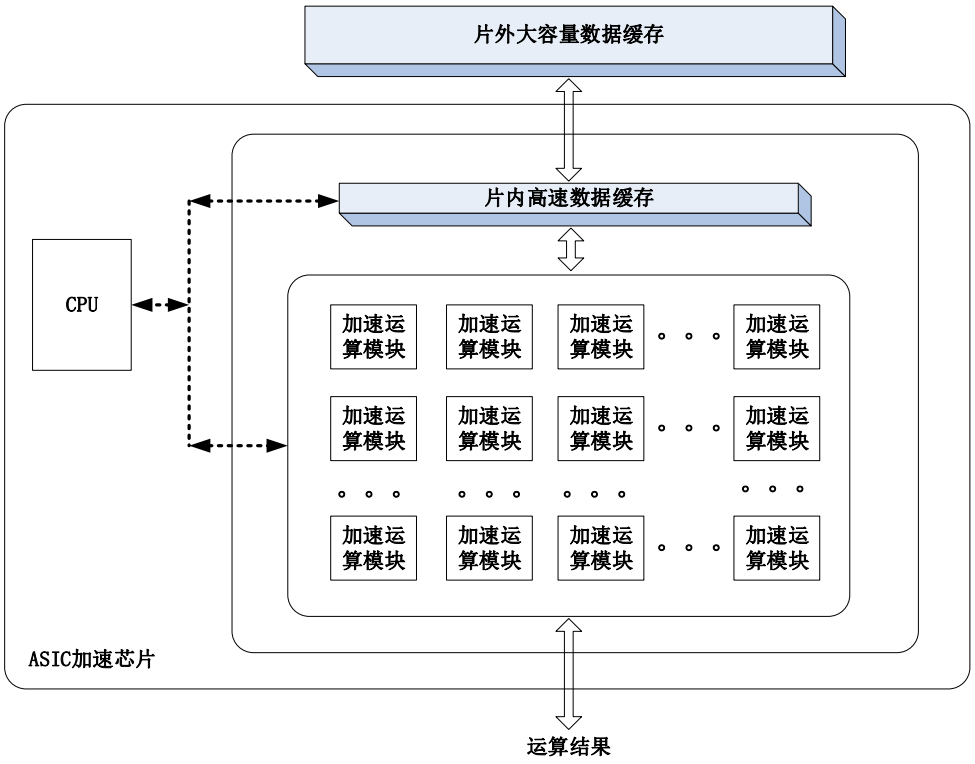

图 3-1 典型 ASIC 加速芯片的结构

在水平方向上，CPU 核负责执行控制流操作。主要包括片内数据缓存中的数据结构组织和调整，以及各个加速运算模块的工作模式的协调和管理。在这种芯片架构中，CPU 核没有处于影响整个 ASIC 加速芯片速度和性能的关键路径上。

3.1.2 抑制 ASIC 加速芯片的 Memory-Hard 算法改造

为了抑制采用图 4.1 中的 ASIC 芯片，以及基于 ASIC 芯片构造的矿机的发展，引入了 Memory-Hard 改进算法。Memory-Hard 改进算法的核心，是提高加速运算模块对片内高速缓存容量的需求。由于实现高速数字电路的 CMOS VLSI 工艺和大容量 SRAM 工艺存在较大的差异，在高速 ASIC 芯片中集成的 SRAM 存储器容量存在一个产品良率和性价比的工程上限。

通过修改算法，不断提升加速运算模块对片内 SRAM 容量的要求，导致片内最大容量的 SRAM 可以支持的加速运算模块的数量逐步减少，最终导致 ASIC 加速芯片因性价比问题退出挖矿。与此算法相对应的 ASIC 加速芯片的结构如下图 3-2 所示。

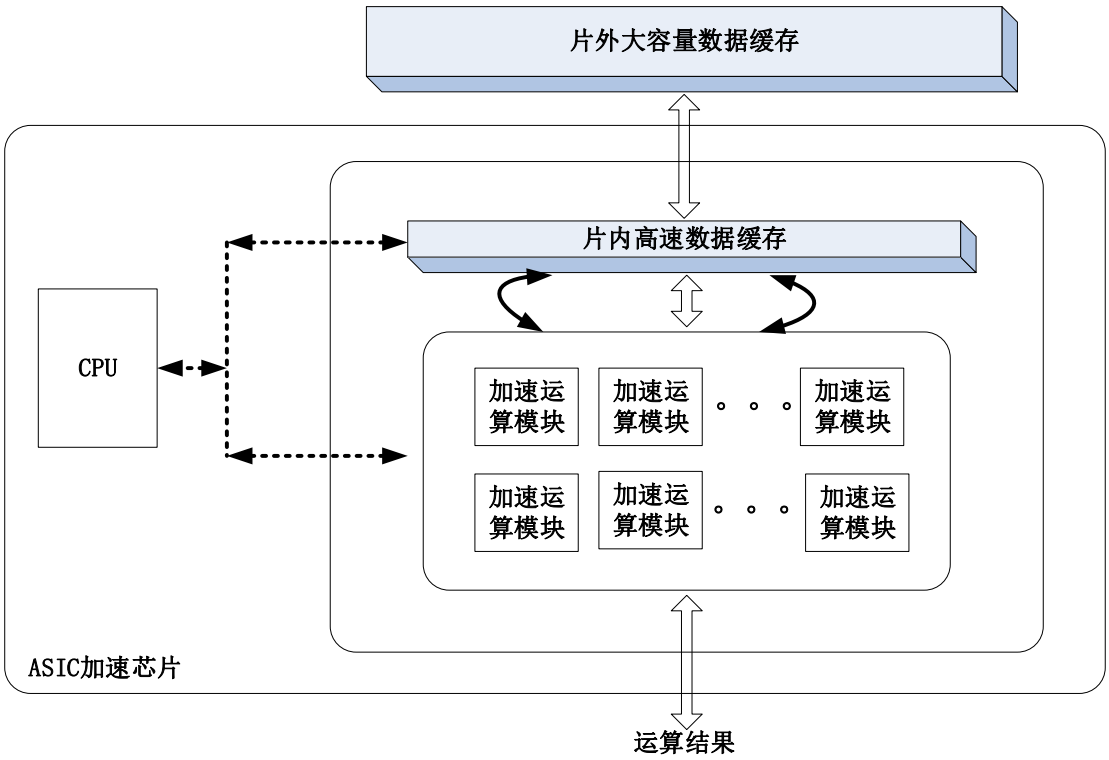

图 3-2 采用 Memory-Hard 算法的 ASIC 加速芯片结构

Memory-Hard 改进算法可以有效地降低单芯片中可以集成的加速运算模块的数量，但由于加速运算模块自身的电路规模小（1K 至 2K 门），这样单个 ASIC 加速芯片的成本可以有效控制，进而可以通过采用多芯片并行工作的方式来提升板级系统整体的算法执行速度。我们需要设计更加有效的 ASIC 芯片和矿机抑制算法。

3.1.3 抑制 ASIC 加速芯片的复杂控制流算法改造

实际上通过分析上述三个算法的原理和结构可知，通过综合采用这三个经过串行化改造的算法的组合，整体算法执行所消耗的时间中，复杂、串行化和密集访存的程序控制流行执行占用了 90%以上的时间，散列函数计算所占用的时间低于 10%。ASIC 加速芯片中加速运

算模块解决的是对应于散列函数指令执行的加速问题,这部分的性能提升仅限于算法执行的10%以下的部分。

这样的算法特征对 ASIC 芯片设计与实现产生了极大的限制，表现在以下三方面：

1、对于占 90%以上时间的程序控制流和指令串行执行部分，简单的加速部件方案难以直接实现；

2、对于常规的基于状态机控制器的 ASIC 设计方案，虽然状态机控制器从理论上能够实现复杂的控制结构，但是无法对必须顺序执行指令序列的有效加速；

3、专用加速模块的仅仅只能加速 10%的计算负载，对整体性能的提升没有太大帮助。

对于必须顺序执行的指令序列,现代微处理器提供深度指令流水线和多级 Cache 等机制,以有效开发多条指令之间的并行性，因此我们认为可行的加速 ASIC 芯片方案，应采用以高速 CPU 为核心的 SOC 芯片设计，来达到加速算法执行的目的。由此得到一种新的复杂控制流算法来抑制和改造 ASIC 加速芯片，对应的芯片结构如下图 3-3 所示。

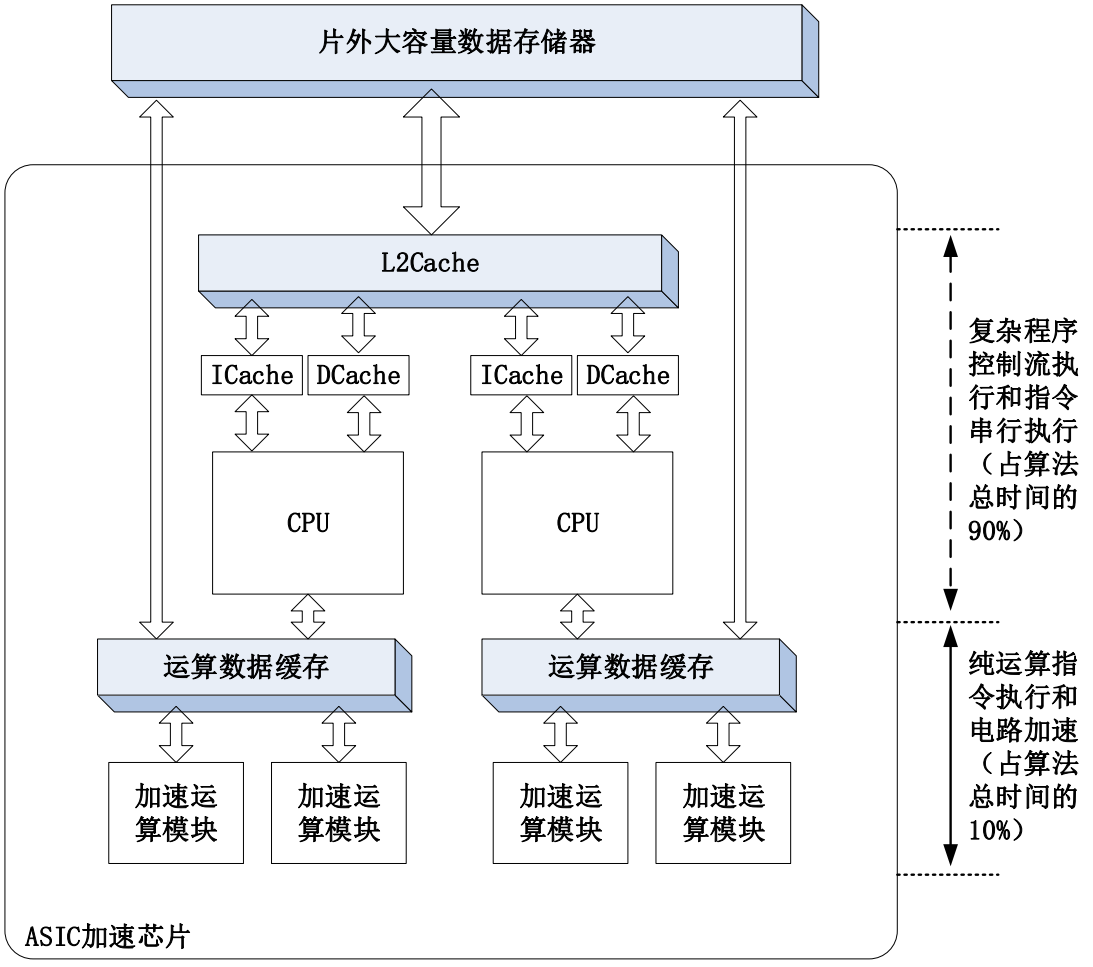

图 3-3 采用复杂控制流算法的 ASIC 加速芯片结构

3.2 基于 ARM 处理器核的 ASIC 方案

从图 3-1 的比较分析可知，集成了 4 个 ARM A57 核的嵌入式处理器 TX1，在执行我们专门设计的三个经过串行化改造的算法组合时，其加速效果按 1 核、2 核、3 核到 4 核的顺序逐步递增，采用更高核数的虚拟核调度和执行，其加速效果维持不变，这与算法的串行执

行机制是一致的，同时也达到了我们预期的算法设计目标。

同时从图 3-1 的对比分析可知，我们提出的单向函数 *CryptoHello* 算法，在不同的硬件平台上运行都遵循相同的规则，即：算法执行速度与单个物理 CPU 核的性能成正比，与系统集成的物理 CPU 核的数量成正比。

针对我们提出的单向函数 *CryptoHello*，可行的 ASIC 芯片设计方案应采用以高速 CPU 为核心的 SOC 芯片设计，且该 SOC 芯片应满足两个要求：

1、单个物理 CPU 核的性能应足够高；

2、芯片中应集成尽量多的物理 CPU 核。

目前市场上最主流 CPU 核为 ARM 公司提供的系列微处理器。按第一个要求，目前可用的高性能 CPU 核是 ARM 的 A73 核，最高主频可以达到 2.8GHZ；按第二个要求，单芯片可以集成的 A73 核可以达到 4 个。这是因为在 10nm 工艺下，集成 4 个 A73 核，采用 64KB 指令 L1 Cache 和 64KB 数据 L1 Cache，以及 2MB L2 Cache 的芯片面积已经达到 $5mm^2$，更多 CPU 核和 Cache 存储的集成将大幅提高芯片的单片成本。

在此 4 个 A73 核的设计方案下，其处理器核数远远少于表 2-1 中服务器和天河二号的处理器核数，因此其性能也将弱于已有 CPU 的实现方案。

与此同时，该 SoC 芯片投片成本高昂，主要包括以下两部分：

1、CPU 核的 License 费。ARM A73 核的单次使用 License 费是 5 百万美元，三年无限次使用 License 费是 1 千 5 百万美元；

2、芯片掩膜的 NRE 费用。支持最快速度的 7 纳米工艺，每层掩膜的费用约为 2 万美元，标准 SOC 芯片需要制作 50 层掩膜，费用在 100 万美元左右。

考虑到研发投入，研制该 SoC 的费用大约在 1 千万到 2 千万美元左右，而且需要大约 2 年的时间才能量产。