Title

MYRSA IMPLEMENTATION YAYI2456

2020 年 6 月 29 日

目录

第 1 章　问题描述

1.1　问题引入

RSA 作为一种广泛使用的公钥加密算法，为了提供足够的安全性，随着计算能力的增强，密钥的长度也需要随之增加。目前，RSA 的推荐密钥长度一般未 1024 位，甚至需要 2018 位。密钥长度的增加不仅为 RSA 被破解增加了难度，同时也给正常加解密的过程增加了巨大的计算开销。而 RSA 广泛的在 SSL 等协议中被使用，算法的执行消耗时间需要被控制在一定范围之内，尤其是更加耗时的解密操作。为了尽量在保证足够的安全性的同时增加 RSA 算法执行效率，前人提出了一系列效率更高的算法来实现 RSA，诸如 RSA-S2 和基于 CRT 的 RSA 等等。除了在算法本身的设计上改进 RSA，在多核心处理器、GPU 技术快速发展的如今，也可以通过考虑对 RSA 算法进行并行化设计来充分利用并行资源，加快 RSA 的执行效率。

RSA 最为广泛的一种用途是对通信双方进行安全通信中会话密钥的加密解密。由于公钥密码算法相较于分组密码算法来说安全性高但加解密效率低，一般不用公钥密码算法进行大量消息的直接加密，而往往是通过使用公钥密码算法加密分组密码算法的密钥，分组密码算法加密消息的方式来使用。因此，这里主要考虑使用 RSA 进行单次数据加密的情况，尽管存在一些工作也研究了多消息下的并行化。

1.2　RSA 算法

具体地，RSA 算法的执行步骤如下所示：

- 密钥生成：

1. 实体选择两个大素数 p、q，满足 p 与 q 的差值较大，计算 $n = p * q$；
2. 计算 $\Phi(n) = (p-1)(q-1)$；
3. 选择大素数 e，满足 $gcd(e, \Phi(n)) = 1$，(e, n) 即为公钥；
4. 计算 d，使得 $e * d = 1 \pmod{n}$，(d, n) 即为私钥。

- 加密:

实体对消息 M 进行加密，计算密文 $C = M^e \pmod{n}$。

- 解密:

实体对消息进行解密，计算明文 $M = C^d \pmod{n}$。

1.3　RSA 的并行化

RSA 的计算主要包括：大素数选取、大数乘与加计算、大数模逆计算和大数模幂运算。针对每一种计算，前人都已经进行了一些研究，尤其是针对大数模幂运算，大量对 RSA 进行并行化的工作都是对模幂运算进行并行化设计，提出了一些有效的算法例如 SMM、LR/RL、Montgomery 算法等。

第 2 章　相关工作

关于 RSA 的并行化设计主要包括对消息分段进行的消息层面上的并行化设计以及对 RSA 算法本身进行的并行化设计。对 RSA 本身进行的并行化设计又包括使用 CRT 进行的并行化、对 Montgomery、RL 等模幂算法进行的并行化设计、利用同余系统分解大幂数进行的并行化设计、对密钥计算进行的并行算法设计等等。根据实现的不同，又分为利用分布式系统实现和单机实现，单机实现分为利用多核 CPU 进行的实现和利用 GPGPU 进行的并行实现，利用 GPGPU 进行并行实现的工作中，又以 OpenCL 和 CUDA 实现为主。

这里主要考虑对单个消息的一次加密的并行化，下面介绍一些相关的 RSA 并行化研究的工作。

2.1　RSA 本身的并行化

Mohammed 等人 [1] 的工作考虑了 RSA 在消息层面上的并行设计和对 RSA 本身做的效率提升。他们使用基于 CRT 的 RSA 算法，分别在单核 CPU，多核 CPU 和 CPU+GPU 平台上实现了 RSA-CRT 算法。由于 RSA-CRT 算法中 M_p 与 M_q、d_p 与 d_q 之间的独立性，允许了一定程度的并行。且由于 RSA-CRT 减小模数，相较于朴素 RSA 的性能有一定的提升。他们的工作更加完善，但仍然是基于已有的 RSA-CRT 算法进行的显而易见的并行化，而没有对更加耗时的模幂操作进行并行化设计。虽然在 CPU+GPU 平台上的算法已经取得了不错的效率，但是模幂算法的瓶颈依然存在。另外，从 Fan 等人 [2] 的工作可以看出，使 GPU 做整个的 RSA 算法的朴素运算是比较低效率的一种方式，RSA 算法本身已经有了许多更高效的算法变体，也更加适合并行化设计。

2.2　模幂运算并行化

除了朴素的模幂运算算法 $a^b \ mod \ c$，研究人员已经设计出了多种更加高效的模幂算法，比如 SMM、LR/RL、Montgomery 等。

从 RSA 这一场景中看模幂运算，一方面需要优化模幂运算本身的效率，另一方面也有研究人员尝试减小模幂运算中大数的长度、模数的长度，这催生出了 RSA-S2 和 RSA-CRT 算法。它们一个是将运算向量化，将一个 d 由许多 $d_{i,j}$ 取代，以更小的素数的运算来达成大素数运算的目标，另一个将模数 n 由 p 和 q 取代，减小了模数的长度。这两种都能减小 RSA 算法中模幂运算本身的复杂程度，也都带来了一定程度的并行空间。

2.2.1 多个模幂运算的分解

模幂运算的分解包括两种方式，一种方式是减小模数，另一种方式是讲一次模幂拆分成为多次。

Li 等人 [3] 设计了 EAMRSA 算法，是一种 RSA-S2 的变体算法。核心思想是：将密钥分解，利用 CRT 来将大数模幂运算转化为多个较小的数的模乘和模加运算，这样，一个完整的大数运算将被转化为多个数据，它们成向量化组织，向量化且彼此无关的数据计算使得算法能够更好的被并行执行。这一算法相较于原始的 RSA 算法更加适合并行化设计，同时也减小了解密过程中的计算量。但是，由于其传播出去的密文需要在 C 之上再做运算，因此增加了加密阶段的开销。但是，这部分计算由于可是向量化的，因此可以通过并行手段来尽量减小这部分开销对加密过程的影响类似的，Gao 等人 [4] 也是使用了 CRT 来将大模数减小，不过没有进行向量化的设计。但是，Gao 等人使用了 Montgomery 算法与 CRT 结合的方式来进行并行算法设计，可以有效地降低模幂运算中模数大小的同时，减小运算过程中的模幂运算的开销。这两者都使用了 CRT，可以看到，CRT 应用于 RSA，可以减小模数的长度，从而减小计算复杂度。

另一种分解方式即拆分模幂。这一方法利用同余，即 $a * b \ mod \ n = (a \ mod \ n) * (b \ mod \ n)$。Sonam 等人的工作 [5] 将一个大数模幂转换为多个大数的小幂数的模幂，他们使用了 CUDA 开发，使 CPU 完成密钥选取的工作，GPU 完成加密和解密的工作。这一方法能够有效减小模幂算法的计算资源消耗，多个相同的任务可以合并，从而大大减小了计算量。它们比较了不同消息大小下这一算法的执行时间和串行程序的执行时间，可以看到加速比始终维持在大于 1.66。但是，他们只对小素数做了对比实验，而对大素数并未给出与串行算法的对比。类似的，Asaduzzaman 等人的工作 [6] 也是通过将模幂运算分割而减小计算量开销。不同的是，他们根据线程数来得到任务划分的大小，且对如何划分任务给出了明确的定义。作者比较了不同线程个数下的运行时间，但是可以看到，该算法虽然减小了计算量，但由于并未使用更加有效的模乘算法，其执行时间并未得到较大的减小。其次，由于作者的实验给出的线程数较少，也是导致结果不理想的一个因素。Ayub 等人的工作也类似 [7]，只不过他们使用 OpenMP 在多 CPU 上做并行处理，而不是用 GPU。在线程数比较小，消息比较少的时候，这一方式是非常有效的。但是当消息比较大的时候，这一简单的改进能够节省的时间开销就比较少了。需要承认，这一方法确实是一种行之有效的并行方法，但是这种方法需要考虑好任务划分的粒度，几位作者均未对这一问题做出更加深入的研究讨论。Sonam 等人的工作 [5] 给出不同线程下同一批消息的执行时间变化，其实也算是对任务粒度不同的一种实验。

经典的模幂分解算法是快速模幂算法，或称为 LR/RL 算法。RL 算法针对模幂运算，它将幂数 e 分解为 $e = e_0 + e_1 * 2 + .. + e_{n-1} * 2^{n-1}$，于是 M^e 就可通过从 M 开始不断做对自身的模乘运算计算得到，能够将模乘的计算次数减小到 $(log_2 e)$ 次。C.K.Cok[8] 针对 RL 进行改进，通过增大快速模幂中的底数并使之长度可变来尽量减小模幂结果计算中的模乘次数。提出了滑动窗口算法。但是这样的方法需要预计算，同时窗口大小选

择策略确定比较困难。对于快速模幂中的每一次循环中都需要累乘计算的中间值 M^{2^i} 的计算次数也并不会减小。Abdul 等人 [9] 使用 CPU+GPU 来对 RL 算法进行并行化设计。RL 算法中由于每一次迭代执行都需要把大数进行平方计算，因此可以预先对大数进行平方计算，而判断是否需要将本次计算结果乘到结果变量上的操作也可以并行提前完成。这种方式需要额外的存储空间来存储中间数据，当大数长度比较大的时候，该空间消耗也会更大。

2.2.2 单个模乘运算的设计

针对模乘算法，正如前面所述，前人已经提出了许多高效、简洁的算法，Montgomery 算法是一种非常高效的模乘算法。它是一种模乘计算算法，它将对模数 N，一个素数的取模转化为对任意 R 的取模运算。当 R 选择为 2^i 时，容易将取模中大量的除法运算使用高效的位运算代替，大大提高了一次模乘运算的效率。它的缺点是，进行运算的大数需要是蒙哥马利空间中的大数，而将一个大数转入到这一空间也需要模乘运算。

Gao 等人 [4] 的工作，已经在前面提到，使用了 CRT 和 Montgomery 算法结合的方式来进行优化。他们使用了 Montgomery 进行模幂运算，并对原始的 Montgomery 算法尽心了并行化设计，但并未使用对并行化设计更加友好的变体算法。Robert 等人 [10] 的工作首先尝试了使用 CUDA 对 Montgomery 算法的变体 CIOS 做出了并行化实现，能够大大增加多个消息的情况下的消息处理速率，减小单个消息的处理时间。此外，针对单个模乘运算的算法设计，Tang 等人 [11]、McIvor 等人 [12] 使用 FPGA 进行了并行算法的硬件设计。

第 3 章　RSA 的高效实现

根据使用的模乘、模幂等运算的算法不同，RSA 也有相应的不同的实现方式。

3.1　RSA 的变体：CRT-RSA

CRT-RSA 利用中国剩余定理，将原始 RSA 中解密过程中对模数 N 的取模转换为对更短的 p 和 q 的取模运算。将模数的大小减小了几乎一倍，同时使幂数的大小减小到与 q 和 p 相似的大小。模数的减小有助于单次取模运算效率的提高，而幂数的减小有助于减小模乘运算的次数。

CRT-RSA 的加密过程与 RSA 相同，它的解密过程如公式3.1所示。虽然 CRT-RSA 的解密过程需要计算 d_p, d_q，最后也需要对 $M-p$ 和 M_q 进行计算得到最终结果 m，但 $qinv, d_p, d_q$ 都可以预计算得到，而且这部分计算的复杂度远远小于其中的模幂运算的复

杂度。

$$\begin{aligned} d_p = d\ mod\ (p-1),\ d_q = d\ mod\ (q-1) \\ C_p = C\ mod\ p,\ C_q = C\ mod\ q \\ M_p = C_p^{d_p}\ mod\ p,\ M_q = C_q^{d_q}\ mod\ q \\ m = M_q + ((qinv * (M_p - M_q))\ mod\ p) * q \end{aligned} \tag{3.1}$$

CRT-RSA 算法不仅带来了可观的计算速度提升，也带来了更大的并行潜力。其中 M_p 和 M_q 的计算是关键计算，且两部分计算之间没有任何依赖关系，很容易使用两个线程分别处理两部分，不需要任何额外的同步开销。

3.2 高效的模幂算法

3.2.1 RL 算法

RL 算法是一种非常高效的模幂算法 [1]，它利用了同余性质，把幂数进行分解，将一个模幂转化为多个模乘，总的模乘次数呈对数级别降低。

具体地，该算法对于一个模幂运算 $M = C^d\ mod\ N$，其运算过程如公式3.2所示。

$$\begin{aligned} d = d_0 + d_1 * 2^1 + .. + d_{n-1} * 2^{n-1} \\ M_i = C^{d_i * 2^i}\ mod\ N,\ i \in [0, n-1] \\ M = \prod_{i=0}^{n-1} M_i\ mod\ N \end{aligned} \tag{3.2}$$

其中计算 M_i 的过程中，有 $M_i \equiv M_{i-1} * M_{i-1}\ mod\ N$。于是，需要的模乘计算次数与 d 地长度 n 成线性关系。将模幂运算中模乘计算次数从 d 次减小到 $log_2 d + bits1(d)$ 次，其中 $bits1(d)$ 是幂数 d 的二进制表示中为 1 的比特个数。

其算法由算法 1 所示。

3.2.2 Montgomery 算法

蒙哥马利算法是一种高效的计算模乘的算法。蒙哥马利莫乘运算非常适合于做连续的模幂运算，它将模乘中的除法操作全部转化为移位操作，大大减小求模过程中的计算资源消耗。

针对模乘运算 $C = A * B\ mod\ N$，将乘数 B 展开，如果以 M 为进制，B 的长度是 L 的时候，可以被展开为 $B = b[0] + b[1] * M^1 + .. + b[L-1] * M^{L-1}$。根据同余性质，$C = (A * b[L-1]M^{L-1}\ mod\ N ... (A * b[0]\ mod\ N)\ mod\ N$。为了尽量削减大数的大小，考虑引入 $R = M^{L-1}\ mod\ N$，计算 $C1 = A * B * R^{-1}\ mod\ N$，其中

[1] 期中报告里面曾经的对幂数分层的想法，经分析实现与实验发现，在幂数较大的时候，由于除法的限制等因素，实际的效率提高几乎是线性的。可进行并行化设计，但并行化设计中的计算存在冗余计算的嫌疑，测试结果加速比也不高，算法设计比较繁琐。因此放弃对它进行设计，改用更加高效的 RL 算法。

Alg-1. RL

1. **input**: $d = d[0] + d[1] * 2 + .. + b[n-1] * d^{n-1}$, C, N
2. **output**: $M = C^d \ mod \ N$
3. $M := 1, \ X = C$
4. **for** i in **range** n:
5. **if** $d[i]$:
6. $M = M * X \ mod \ N$
7. $X* = X$

Alg-2. Montgomery

1. **input**: $BR = b[0] + b[1] * M + .. + b[L-1] * M^{L-1}$, AR, N
2. **output**: $C = AR * BR * R^{-1} \mod N$
3. $C := 0$
4. **for** i in **range** L:
5. $C+ = N * (C[0](M - N[0]^{-1}))$
6. $C+ = b[i] * AR$
7. $C/ = M$
8. $C\% = N$

$R^{-1}R = 1 \ mod \ N$，那么，$R^{-1} = M^{1-L} \ mod \ N$。仍然使用同余性质进行分解，得到 $C1 = (A * b[0] * M^{1-L} \ mod \ N)...(A * b[L-1] \ mod \ N) \ mod \ N$。这样产生了一个问题，那就是如何保证在移位的时候不会丢失信息。解决方式也很简单，利用同余，在第 i 次循环中可以求解式子 T，满足 $T \ mod \ N = 0$，这样，$C_{i-1} + T \ mod \ N = C_{i-1} \ mod \ N$ 且 $(C_{i-1}+T)[0] = 0$，只需要在移位之前加上 `T`，经过移位就不会丢失信息，也可以保持求解精确。为了保证 $T \ mod \ N = 0$ 且 $(C_{i-1}+T)[0] = 0$，可以直接求解 $C_{i-1} + qN \ mod \ M = 0 \ mod \ M$。求解可得 $q = C[0](M - N[0]^{-1})$。其中 $N[0]^{-1}N[0] = 1 \ mod \ M$。

其算法过程由算法 2 所示。

最后，计算 $CR^{-1} \ mod \ N$ 获取最终结果。

第 4 章　并行算法设计与实现

RSA 的并行化设计主要包括算法本身步骤的可并行性以及大数运算的可并行性。经过性能测试，选择表现最好的 CrtRSA+RL+Montogomery 组合实现的 RSA 算法进行并行化设计。

表 4.1: 2048bits 下解密过程各计算部分耗时 (ms)

解密总用时	Mp 计算用时	Mq 计算用时
2840.3	1306.9	1333.4
Mp 的 2^i-累乘用时	**Mp 的 res-累乘用时**	**Mp 中 Montgomery init 用时**
872.20	461.19	50.691

4.1 关键计算

在进行并行化设计之前，首先对串行算法中一次解密过程中的各个部分计算用时进行测试。在密钥长度为 2048bit 时，解密过程各个主要部分耗费的时间如表4.1所示。

将 RL 中的计算 $M_i = M_{i-1} * M_{i-1}$ 部分称为 2^i-累乘部分；将 RL 中的计算 $res = res * M_i$ 部分称为 res-累乘部分。

值得一提的是，其中由于 CRT-RSA 包括两个不同的底数和模数的蒙哥马利模乘器，因此需要进行两次初始化。但是，由于模数的减小和底数的减小，CRT-RSA 中一个蒙哥马利模乘器的初始化时间约为 RSA 中一个蒙哥马利模乘器初始化时间的 $\frac{1}{4}$。因此，实际上 CRT-RSA 在蒙哥马利模乘器初始化中仍然节省了几乎一半的时间。

首先从表4.1中可以看出，Mp 与 Mq 计算总用时时间差别不大，未给出的是，二者 2^i-累乘和 res-累乘时间也相差不大，因此只给出了 Mp 的 2^i-累乘和 res-累乘的计算时间、蒙哥马利模乘器初始化时间。

从表4.1可以看出，主要计算集中在 RL 的循环部分，占据了大约 93% 的时间，蒙哥马利模乘器的初始化占据了约 3.6% 的时间，剩余约 4% 的时间主要是 CRT-RSA 最后对 m 的计算，还包括蒙哥马利形式转出转入和其他程序执行函数调用等耗时。

蒙哥马利模乘器的初始化中，几乎所有时间都用来计算 $R^2 \ mod \ N$。该数据计算主要耗时为模运算中是用二分法寻找商的耗时。二分法寻找商，每一次迭代之间有存在依赖，上一次迭代结果直接决定下一次迭代中的取值上下界。运行时间为 $O(log(R^2))$。即使将之看作搜索算法，由于该算法一定只需要运行树高次，而不存在走错分支，并行化并不能有效帮助提高效率。因此，下面的并行化设计，主要集中在讨论 RL 的循环部分。

4.2 CRT-RSA 并行化

在讨论 RL 循环之前，容易看出，CRT-RSA 的 Mp 计算和 Mq 部分计算完全没有任何依赖，且根据表4.1可得，二者完成计算所需时间几乎相等，任务量相差不大。因此，很容易找到一种方法对之并行化，那就是使用两个不同的进程分别计算 Mp 相关数据和 Mq 相关数据。

如公式3.1所示，使进程 1 计算每一行左侧公式，进程 2 计算每一行右侧公式。这

样很容易得到一种可行的并行化方式，使用 openmp 也容易得到可执行代码。

接下来考虑每一个进程之间数据的并行可能性。以左侧为例，第一行公式计算属于预计算内容，这里不做考虑，因此主要运算集中在第三行公式的模乘。这部分的并行化将在下一小节讨论。

4.3 RL 算法并行化设计

如算法 1 所示，RL 算法的每一个循环之间都存在强依赖关系，即使展开循环也无法在保证计算量不增加的情况下消除依赖关系。因此多次迭代之间的并行几乎是一定无法实现。

虽然迭代之间的并行难以实现，但可以在迭代内部考虑并行。从表4.1中可以看出，2^i-累乘和 res-累乘所占时间比大约是 2:1，这个数据与幂数 d 相关，但是 2:1 的数值与预期相符，这代表幂数 d 的二进制表示中大约有一般的比特数是 1，另一半是 0。2:1 的计算时间比值，加上一次循环中的 2^i-累乘和 res 累乘之间只有读后写依赖，只需要通过提前数据复制就能很容易地解决，因此可以考虑在循环内部进行并行设计，使用两个进程在 $d[i] == 1$ 的时候分别运行本次迭代中的 2^i-累乘和 res-累乘部分。

使用两个进程分别进行 2^i-累乘和 res-累乘，一个长度为 $bit1(d)$ 的大数数组作为全局存储。2^i-累乘进程负责计算 M_i 并将 M_i 存入全局的大数数组对应位置，并在计算完成一个 d 的当前比特为 1 的 M_i 之后刷新全局存储。它无需进行任何等待。res-进程轮询下一个 d 的二进制表示比特位 1 位置的 M_i 是否计算完成，一旦就算完成，它就从全局存储中拿到这个大数，与最终结果大数进行模乘运算的更新。一般来说，它需要等待 2^i-累乘进程完成计算。

这一设计的缺点是，进程之间的同步和全局变量的刷新可能会带来一定的开销，另外，需要相当的一块存储空间来临时存储 2^i-累乘在各个迭代中计算得到的临时值。

但是，如果不使用全局存储空间，而是只依靠同步来保持 res 累乘计算进程读取数据的正确性，将会带来更多不必要的同步开销，将可观的时间花费在两个进程之间的相互等待，尤其是 2^i-累乘计算进程对 res-累乘计算进程的等待。由于 2^i 累乘进程的计算用时几乎不可能再降低，因此该进程的等待必然造成运行时间的增加。

两个进程执行内容可由算法 3 所示。

在 RL 内部，其每一个模乘，都是用蒙哥马利模乘计算，如算法 2 所示。观察算法 2 的特点，可以发现，类似于 RL，算法迭代之间依赖非常强，除此之外，每一个迭代内部也存在较强的依赖。当一次迭代中每一个操作都存在较强的依赖难以进行并行化时，考虑每一个操作本身是否可以被并行。观察迭代内部的计算，每一个迭代中的运算包括大数的乘运算、加运算、位运算。相较于大数乘运算和加运算，位运算的计算开销显然是极低的。因此可以考虑对大数乘和大数加进行并行化设计。

Alg-3. RL 并行

input: $d = d[0] + d[1] * 2 + .. + b[n-1] * d^{n-1}$, C, N
output: $M = C^d \ mod \ N$
$bigint \ array[n]$
2^i**-thread**:
for i in **range** n:
    if $d[i]$:
        $flush \ array$
    $X* = X$
    $d[i] = X$
res-thread:
for i in **range** n:
    if $d[i]$:
        while $!array[i]$
    $M* = d[i]$

4.4 大数乘法的并行化

由于这里针对的算法 2 中的大数乘法，因此观察算法 2，其主要的大数乘法运算实际上是一个大数和一个固定比特数数据的乘法。根据分解大数 B 的时候的进制的选择不同，这个固定的比特数也有变化。对于常用的 32 比特，实际上需要的大数乘法的并行化是对一个大数和一个长度位 32bit 的 uint 类型数据的乘法的并行化。因此这里考虑一个大数和一个整数的相乘的并行化设计。

大数运算本身的并行化设计显然需要依赖于 SIMD 技术进行。这里存在两种选择，一种是使用 AVX 技术，另一种是使用 GPGPU。在下一小节将会讨论到不使用 GPGPU 的原因，这里先讨论使用 AVX 对大数乘法进行并行化设计。

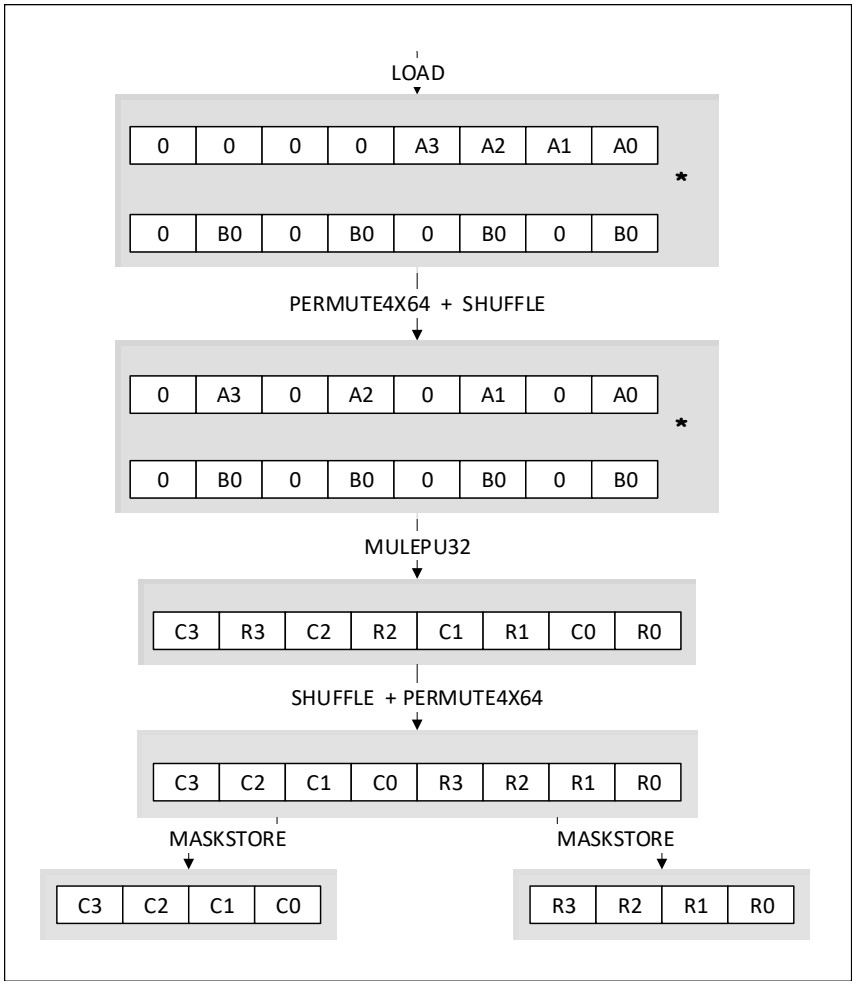
集合 RSA 的具体应用场景，可以确定大数的长度在 512 比特的倍数，诸如 1024 比特、2048 比特或更高。AVX 最多支持 256bits 的数据运算，根据乘法的性质，考虑积的长度是两个相同长度乘数的二倍，当进制为 2^{32} 一次最多可以处理 4 个大数片段。一次迭代中的具体处理过程可以由图4.1所示。

正如图4.1中所示，这里仅将大数乘法的得到的高位和低位都返回。由于 AVX 对加法中的进位支持比较差，使用 AVX 处理进位比较困难。这个困难包括：

1. 确定进位：两个相同长度整数相加，通过比较加数和和可以确定是否产生进位，进位必然为 1。

2. 进位传递：进位传递是最麻烦的一部分。其一，由于高位进位依赖于低位进位，因

图 4.1: AVX 大数乘并行一次迭代中处理过程



此加入对进位的处理对加速比的影响必然很大，需要进行四次串行的进位处理。较之直接串行处理加法，并没有明显的优势。

因此，对大数乘法的并行化仅限于得到高位和低位。随后可以直接使用大数加法进行加和。

4.5 大数加法的并行化

正如在前一个小节所述，AVX 对进位不太友好。类似的，在对大数乘法进行 AVX 的并行化之前，首先对大数加法进行了 GPGPU 的并行化设计。

设计思路是，尽管进位的处理必定是串行的，但加法计算可以是并行的。首先令每一个工作项计算被分配的整数之和，容易得到一个进位和一个本位数据。这部分只需要使用处理一个整数和的时间。每一个工作项将得到的本位数据自己存储，进位则存储到被分配给自己的共享内存，随后，每一个工作项检查在全局内存中，自己的前一个工作项是否产生了进位数据，一旦产生了进位数据，就将这个进位加到自己的本位数据，随后更新自己负责的存储进位数据的共享内存。

一个 kernel 函数的执行流程图如图4.2所示。

这一方法在最优情况下，只需要计算一个整数加的时间，最坏情况下则接近串行执行时间。

但是在测试中发现，当数据量比较少的时候，GPGPU 的优势并不明显。经过测试，在 2048bits 下，串行对两个大数相加，时间花费在 14us 左右波动，但是对 kernel 进行测试，执行时间则在 12ms 左右。加上各种初始化、内核的编译等工作，一次完整的时间竟达到 2s 以上，几乎等于一次完全串行的 CRT-RSA 的整个解密过程的耗时。在测试一节将会提到，在上述的并行化策略作用下，CRT-RSA 的最终解密时间需要 200ms 左右。这里的包括初始化、内核编译、数据传输等在内的一次简单的大数加就已经是其时间的 10 倍。当然，这种现象的发生，与设备本身的性能也有关系。但是，在性能差距如此悬殊的情况下，仅在该部分针对 GPGPU 的设计测试性能出现之后，就放弃了对它的 GPGPU 并行设计。

第 5 章 实验与分析

测试各个并行优化效果，每一个条件下以随机消息运行 100 次解密过程，使用平均使用时间作为该条件下的平均运行耗时，仍然在 2048bit 之下进行测试。测试结果如表5.1所示。其中 AVX 代表大数乘法并行起作用，TT 代表 RL 算法并行起作用，OMP 代表 Crt 的并行起作用。

另外，对每一中算法组合测试解密时间、Mp 计算时间和单次模乘时间，得到的测试结果如表5.2所示。

图 4.2: 大数加并行 kenel 处理过程

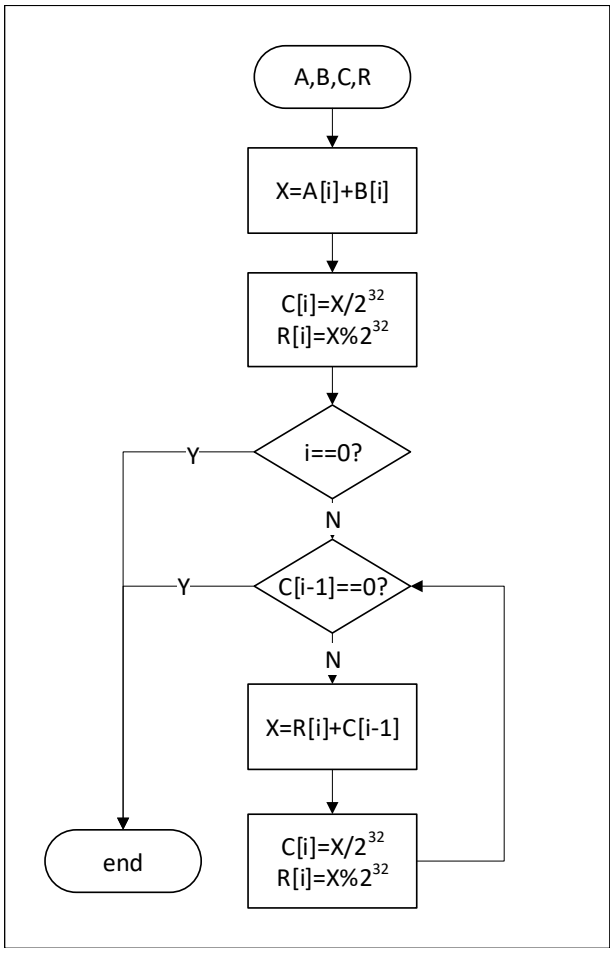


表 5.1: 2048bits 下解密过程并行化耗时 (ms)

串行解密	AVX	TT	OMP
2787.9	302.14	2094.8	1533.8
AVX+TT	**AVX+OMP**	**TT+OMP**	**AVX+TT+OMP**
274.20	187.03	1438.5	179.63

表 5.2: 不同并行算法下 CrtRSA 解密中各部分耗时 (ms)

算法	解密	Mp 计算	单次模乘
串行	2793.0	2763.1	0.8336
AVX	308.77	279.50	0.04201
TT	2058.1	2027.8	0.8739
OMP	1518.3	1486.1	0.8889
AVX+TT	293.32	262.85	0.05002
AVX+OMP	196.36	168.01	0.05498
TT+OMP	1444.1	1415.5	0.8673
AVX+TT+OMP	190.59	80.522	0.04874

首先对 AVX 超高的加速比进行分析。可以看到，只要当 AVX 并行被开启，整个解密过程的加速比就非常高。AVX 并行化负责的是大数的乘运算，该运算在蒙哥马利乘中是主要运算，且在表4.1中已经得到，解密中 93% 左右的计算量都是蒙哥马利乘，因此 AVX 必然会带来很高的加速比。但是，AVX 中处理大数乘，相较于串行程序之中每次处理 32bits 的数据，它可以每次处理 128bits 的数据。但是 AVX 并行中需要进行数据的读取和存储等操作，因此最高应该只有 4 倍加速，根据表5.1，显然 AVX 并行带来的加速比高于 4，甚至几乎达到了 9。这是由于，串行程序设计中的大数乘，是基于 `uint32_t` 数据的相乘与相加，而并非是基于 AVX、每次只处理一个 32bits 数据的大数乘。在串行程序中使用的实现和 AVX 并行的实现，其中的乘加操作可能本身就已经存在了性能的差异。如表5.2所示，可以看到这样的猜想是正确的。当 AVX 并行化被开启，单个模乘时间降低至 45us 左右，但 AVX 未开启的时候，单个模乘时间达到 850us 左右，二者相差近 20 倍。当单个模乘运算耗时急剧减小，蒙哥马利初始化等其他约 7% 占比的计算所耗时间不变的时候，容易计算得到，加速比 $=(93+7)/(\frac{93}{20}+7)\approx 8.6$，与表5.1中得到的实验结果相符合。此时，Mp 和 Mq 的计算循环的运行时间则减小了 10 倍，小于单次模乘运行时间减小。这是因为，Mp 和 Mq 的计算中还包括其他的取模运算。当单次模乘耗时间减小，其他较耗时的运算运行时间不变时，总的 Mp、Mq 循环时间占比减小并不会达到完全的 20 倍左右。在没有 AVX 并行下，模乘占据 Mp、Mq 循环时间的 94%，在 AVX 并行下，模乘占据 Mp、Mq 循环时间的 40%。

其次，对 OMP 并行，在理论上它应该能达到接近 2 倍的加速比，但无法达到或超过 2。这是因为，计算 Mp 和 Mq 的计算量显然在解密中占据近 93%，这二者的并行进程之间没有任何同步与数据依赖，应该能达到近 2 倍的加速比，实际上也确实是这样。

最后，对于 TT 并行，根据表4.1中的数据，res-累乘与 2^i-累乘的计算量大约是 1:2。这部分循环的总的计算量约 93%，则加速比预测为 $100/69\approx 1.4$。如表5.1中所示，实际实验中，测试出的加速比大约为 1.3，符合预期。

将这三种并行方式两两组合，首先是 AVX+TT。在 AVX 的基础上，TT 带来的加速并不高。相较于 AVX，AVX+TT 的加速比仍然来源于对 res-累乘的时间减小，实际上是蒙哥马利模乘运算执行时间的减小。在 AVX 加持下，蒙哥马利模乘所占时间占比缩小到约 40%，此时 TT 能够提供的加速比约为 $100/86.66 \approx 1.15$。实际实验中，这一数字会进一步缩小，在如表5.1中，加速比约为 1.1，略小于预期。但是实际上，TT 不可能为 Mp、Mq 循环部分的模乘减小 1/3 的耗时，因此 1.1 的加速比是在预期范围内的。

对于 AVX+OMP，仍然分析 OMP 能够带来的加速比。不同于 TT，OMP 几乎能够为 Mp、Mq 循环部分，而非仅仅是循环中的模乘部分，带来非常接近一半的运行时间削减。Mp、Mq 循环部分仍然占据着近 90% 的运行时间，因此，其加速比应在 $100/55 \approx 1.81$ 左右。实际上，在表5.1中，实际的加速比是 1.62。

对 TT+OMP，OMP 首先将 93% 左右的计算时间减小到几乎只占据 $(93/2)/(93/2+7) \approx 87\%$，TT 随后将 $87*0.94 \approx 82\%$ 左右的计算时间减小到约 $((82*2/3)/((82*2/3)+18) \approx 75\%$，将 $87*0.06 \approx 5\%$ 左右的计算时间减小到约 $((5)/((82*2/3)+18) \approx 7\%$，二者共占约 82% 的执行时间。最终相较于串行算法，应该达到 $100/(82*7/18+7)/approx2.5$ 左右的加速比。在表5.1中，得到的加速比为约 1.93。这个加速比相差有略大。考虑到对 OMP 和 TT 的加速比都是估计，且其中的各种同步开销也未考虑，预期加速比可作为上限存在。

最后，当三者结合的时候，一次解密的时间消耗大约是 179.63ms。这一数据比起串行中的数据已经有了很大的进步。为了便于分析，假设 AVX 并行化后的初始时运行时长为 100，其中 Mp、Mq 中的模幂运行时间 36，Mp、Mq 中其他运算运行时间 54，其他运算运行时间 10。OMP 并行化之后，时间变为 18、27、10；TT 并行化之后，时间变为 12、27、10。于是很容易计算得到，预期加速比为 2.0。实际上在表5.1中，加速比为 1.7。

第 6 章　总结

本文针对 CrtRSA 在使用 RL 模幂和蒙哥马利模乘的实现做了并行化设计，并通过分析和实验测试了性能。使用并行化技术后，解密算法能够更高效地完成一次解密过程，较完全串行的算法性能有较大的提升。

参考文献

[1] YOUNIS M I, FADHIL H M, JAWAD Z N. Acceleration of the RSA Processes based on Parallel Decomposition and Chinese RemainderTheorem[J]. International Journal of Application or Innovation in Engineering and Management, 2016, 5(1): 12–23.

[2] FAN W, CHEN X, LI X. Parallelization of RSA algorithm based on compute unified device architecture[C] // ANON. 2010 Ninth International Conference on Grid and Cloud Computing. 2010: 174–178.

[3] LI Y, LIU Q, LI T. Design and implementation of an improved RSA algorithm[C] // ANON. 2010 International Conference on E-Health Networking Digital Ecosystems and Technologies (EDT): Vol 1. 2010: 390–393.

[4] GAO S, ZHANG S, FU M, et al. Cost-Efficient Parallel RSA Decryption with Integrated GPGPU and OpenCL[J], 2016: 597–602.

[5] MAHAJAN S, SINGH M. ANALYSIS OF RSA ALGORITHM USING GPU PROGRAMMING[J]. International Journal of Network Security and Its Applications, 2014, 6(4): 13–28.

[6] ASADUZZAMAN A, GUMMADI D, WAICHAL P. A promising parallel algorithm to manage the RSA decryption complexity[J], 2015: 1–5.

[7] AYUB M A, ONIK Z A, SMITH S. Parallelized RSA Algorithm: An Analysis with Performance Evaluation using OpenMP Library in High Performance Computing Environment[J], 2019.

[8] KOC C K. Analysis of sliding window techniques for exponentiation[J]. Computers & Mathematics With Applications, 1995, 30(10): 17–24.

[9] RAZAQUE A, JINRUI W, ZANCHENG W, et al. Integration of CPU and GPU to accelerate RSA modular exponentiation operation[J], 2018.

[10] SZERWINSKI R, GÜNEYSU T. Exploiting the power of GPUs for asymmetric cryptography[C] // ANON. International Workshop on Cryptographic hardware and embedded systems. 2008: 79–99.

[11] TANG S, TSUI K, LEONG P H W. Modular exponentiation using parallel multipliers[C] // ANON. Proceedings. 2003 IEEE International Conference on Field-Programmable Technology (FPT)(IEEE Cat. No. 03EX798). 2003: 52–59.

[12] MCIVOR C, MCLOONE M, MCCANNY J V. Modified Montgomery modular multiplication and RSA exponentiation techniques[J]. IEE Proceedings-Computers and Digital Techniques, 2004, 151(6): 402 – 408.