SIMD 编程：基于 neon 的 NTT 优化

2313911 黄尚扬

2025.4.26

目录

一．引言

SIMD(Single Instruction, Multiple Data)是一种并行计算技术，它通过向量寄存器存储多个数据元素，并使用单条指令同时对这些数据元素进行处理，从而提高了计算效率。SIMD 已被广泛应用于需要大量数据并行计算的领域，包括图像处理、视频编码、信号处理、科学计算等。许多现代处理器都提供了 SIMD 指令集扩展，例如 x86 平台的 SSE/AVX，以及 ARM 平台的 NEON。[1]

多项式乘法作为基础的数学运算，在信号处理，计算机图形学，密码学等领域有着广泛的应用。在该并行选题中，我们重点关注同态加密中的多项式乘法，同态加密可以在密文上进行运算，对运算后的密文进行解密的结果等于明文直接运算的结果，这在安全领域中是非常重要的技术，而同态加密的其中一项关键组成部分便是多项式乘法。

本研究基于 NEON，对已有的 NTT 算法进行优化。

二．问题描述

在本次选题中，我们并不需要关注同态加密的复杂数学原理，而只关注其中的多项式乘法部分及其 NTT 优化。下面为多项式乘法的基本定义：给定多项式 $f(x)=a_{n-1}x^{n-1}+...+a_1x+a_0$ 和多项式 $g(x)=b_{m-1}x^{m-1}+...+b_1x+b_0$，设 $h(x)=f(x)g(x)$，则有 $h(x)$的 x_k 项系数 h_k 为：

$$h_k = \sum_{i=0}^{k} a_i b_{k-i}$$

朴素的多项式乘法计算需要 $O(n^2)$的时间复杂度，这在实际应用中是无法接受的，所以实际应用中一般通过 FFT 或者 NTT 来加速多项式乘法的计算，由于本选题重点关注同态加密领域中的多项式乘法，所以我们重点研究 NTT 算法的并行加速。

对于 FFT 而言，其主要研究时域与频域上的转化，浮点操作可能会损失精度；而 NTT 则是在整数取模域上的操作，当取友好的模数时可以避免误差。接下来笔者将会摘录一段**我自己以前写的算法学习笔记**来描述 NTT 的一些原理。

“和 FFT 相似，$x(n)$表示的是一列整数，类似于时域系数；$X(m)$指的是一列变换后的整数，类似于频域点值。这里的 a^{mn} 就是我们使用 NTT 遍历求和时的

生成元的幂次遍历，我们可以将 a 类似地视为 n 次单位根，而当它在 n 上遍历时就相当于 FFT 中的转圈操作。

“对于某个满足 $p=c*2^r+1$ 的素数 p 以及一列 n<=k　(k=2^q)个的数列，我们可以用类似于 FFT 的操作，在对 p 取模域上构造出 $a=g^{(p-1)/k}$（其中 g 是 p 的一个原根），而它就可以被拿来直接代替单位根在模数域上进行 NTT 运算。特别地，对于 p=998244353，其原根 g 为 3。”

于是，朴素的串行 NTT 很好理解，就是把 FFT 模板中复数的部分等效替代为上面的 a，并且加上求快速幂和逆元的部分即可。进一步地，经过推导可以获知只要求 a^{p-2} 即可获知逆元。为了增强效率，我们会采用蝴蝶变换来将递归变为递推。

以上就是问题的基本描述以及 NTT 的朴素思路。

就**复杂度**分析而言，和 FFT 相似，NTT 本身是一个 nlogn 的算法；在实现了数据操作上的并行后，其时间复杂度量级不变，但用时可能降到原来的数分之一，从而达到优化的目的。

三．算法设计

首先我们会先实现串行的 NTT。代码如下：

```
int ksm(int a,int b,int p){//快速幂 modp
    if(!b)  return 1;
    int tmp=1;
    while(b>1){
        if(b&1) tmp=1ll*tmp*a%p;
        a=1ll*a*a%p;
        b>>=1;
    }
    return 1ll*a*tmp%p;
}


int k=0,lim=1;//k 表示最大幂次, lim 表示（大于等于 2n 的）2^k
int qwq[400000];//辅助数组，用于存储蝴蝶变换下标
void realntt(int *a,int p,bool opt){//算法主体,opt 表示正运算还是逆运算

    for(int i=0;i<lim;++i){
        if(i<qwq[i])    std::swap(a[i],a[qwq[i]]);
    }
```

```
    for(int mid=1;mid<lim;mid<<=1){
        int wn=ksm(3,(p-1)/(mid<<1),p);
        if(!opt){//负运算，需要求逆元
            wn=ksm(wn,p-2,p);
        }
```

```
        for(int j=0;j<lim;j+=(mid<<1)){//主体
            int w=1;
            for(int k=0;k<mid;++k,w=1ll*w*wn%p){
                int x=a[j+k];
                int y=1ll*a[j+k+mid]*w%p;
                a[j+k]=(x+y)%p;
                a[j+k+mid]=(x-y+p)%p;
            }
        }
    }
```

```
    if(!opt){//负运算除以长度
        int awa=ksm(lim,p-2,p);
        for(int i=0;i<lim;++i){
            a[i]=1ll*a[i]*awa%p;
        }
    }
    return;
}
void ntt(int *a,int *b,int *ab,int n,int p){//优化算法
```

```
    while(lim<n*2){//初始化幂次
        lim<<=1;
        ++k;
    }
```

```
    for(int i=0;i<lim;i++){//对每个位置下标的二进制位操作
        qwq[i]=((qwq[i>>1]>>1)|((i&1)<<(k-1)));
    }
```

```
    realntt(a,p,true);
    realntt(b,p,true);
```

```
    for(int i=0;i<lim;++i){
        ab[i]=1ll*a[i]*b[i]%p;
    }
    realntt(ab,p,false);//INTT
```

```
    return;
}
```

由于这不是本次实验的重点，在这里就简单带过。首先我们有快速幂，这个就不详细解释了。lim 是用于扩展数列长度到 2^k 所设置的，这是因为 NTT 要求如此。在本次作业中给出的数据无似乎需这个操作，但鉴于它是 NTT 模板的一部分，笔者还是写上了。

我们的 ntt 函数里会先将下标逆序置换，然后再调用 realntt 进入到计算主体部分。这里是采用了蝴蝶变换的递推版本。

接下来我们考虑将串行 NTT 转化为并行 NTT。NEON 提供了向量化的操作，允许我们使用单指令多数据操作，以此提高效率。

让我们一步一步优化，先从最基础的方法开始。

首先，我们有以下代码：

```
uint32x4_t mc(uint32x4_t A,uint32x4_t B,int mod){
    uint32x2_t al = vget_low_u32(A);
    uint32x2_t ah = vget_high_u32(A);
    uint32x2_t bl = vget_low_u32(B);
    uint32x2_t bh = vget_high_u32(B);

    uint64x2_t awa1=vmull_u32(al,bl);
    uint64x2_t awa2=vmull_u32(ah,bh);
    int tmpa=(uint32_t)(vgetq_lane_u64(awa1,0)%mod);
    int tmpb=(uint32_t)(vgetq_lane_u64(awa1,1)%mod);
    int tmpc=(uint32_t)(vgetq_lane_u64(awa2,0)%mod);
    int tmpd=(uint32_t)(vgetq_lane_u64(awa2,1)%mod);
    uint32x4_t tmpans={tmpa,tmpb,tmpc,tmpd};
    return tmpans;
}
```

这段代码是一个朴素的模乘，暴力地将向量内元素拆出来取模。值得注意的是，在最上方的 4 行中我们把 4 个元素的向量拆分为了 2 个元素的向量，这是为了避免乘法溢出而使用返回值为 64 位的向量导致的。

然后下面是我们的蝴蝶主体，用于替换相似部分：

```
uint32x4_t modv=vdupq_n_u32(p);//{p,p,p,p}，方便内部调用
        for(int j=0;j<lim;j+=(mid<<1)){//向量化的主体
            int w=1;
            int k=0;
            for(;k+3<mid;k+=4){
```

```
            int32x4_t x={a[j+k],a[j+k+1],a[j+k+2],a[j+k+3]};//加载
            int32x4_t b={a[j+k+mid],a[j+k+mid+1],a[j+k+mid+2],a[j+k+mid+3]};

            int w1=w;
            int w2=1ll*w1*wn%p;
            int w3=1ll*w2*wn%p;
            int w4=1ll*w3*wn%p;
            uint32x4_t wv={w1,w2,w3,w4};//预处理 4 个一组的单位根
            w=1ll*w4*wn%p;//将幅角转 4 倍单位根

            uint32x4_t y=mc(vreinterpretq_u32_s32(b),wv,p);//先做乘法

            int32x4_t sum=vaddq_s32(x,vreinterpretq_s32_u32(y));//x+y->a[j+k]
            int32x4_t diff=vsubq_s32(x,vreinterpretq_s32_u32(y));//x-y->a[j+k+mid]
```

```
        //从这里开始
            uint32x4_t mask1=vcgeq_s32(sum,vreinterpretq_s32_u32(modv));//比较 sum 的每一个元
素和 p
            sum=vreinterpretq_s32_u32(vsubq_u32(vreinterpretq_u32_s32(sum),vandq_u32(mask1,
modv)));//如果大于 p（对应位置为 0xFFFFFFFF），就-p

            uint32x4_t mask2=vcltq_s32(diff,vdupq_n_s32(0));//clt 和 cge 相反
            diff=vreinterpretq_s32_u32(vaddq_u32(vreinterpretq_u32_s32(diff),vandq_u32(mask
2,modv)));//类似
        //到这里为止
        //注意到我们的 sum=x+y,又有 diff=x-y,而 x 和 y 都小于 p，于是结果在-p 到 2p-1 间，只需要判断溢
出再做一次加减就行

            vst1q_s32(a+j+k, sum);
            vst1q_s32(a+j+k+mid, diff);//存回内存
        }
        for(;k<mid;++k,w=1ll*w*wn%p){//这里处理最后剩下的部分
            int x=a[j+k];
            int y=1ll*a[j+k+mid]*w%p;
            a[j+k]=(x+y)%p;
            a[j+k+mid]=(x-y+p)%p;
        }
    }
```

具体的分步操作我在注释中都已经标注出来了。其逻辑并不复杂，无非就是以 4 个元素为一个单位来处理循环求解的部分，最后再放回内存。实现了这部分并行代码后，我提交发现：虽然正确性没有问题，但跑得太慢了。**不过在实现进一步优化之前，我们已经可以认为，在串行 NTT 基础上实现的向量化底线任务**

已经完成。

接下来我们试图进行蒙哥马利规约优化。根据我的理解，这个优化的核心是将模乘转化为加法、乘法以及位运算的形式。

一般地我们会有：

$$\text{Mont}(x) = (x + ((x \times \text{mod\_inv}) \mod R) \times \text{mod})/R$$

而其中 R 会取 2^32,并且 mod_inv 是负逆元。

这样我们只需要取低 32 位来乘上 inv 再进行回乘即可。我的代码如下：

```
uint32_t getinv(uint32_t mod){//求一下逆元
    uint32_t ret=1;
    for (int i=0;i<5;i++)    ret*=2-ret*mod;
    return -ret;
}
//规约函数
inline uint32x2_t montgomery_reduce_u64(uint64x2_t x,uint32_t mod,uint32_t mod_inv) {
    uint32x2_t x_lo=vmovn_u64(x);
    uint64_t t0=(uint64_t)x_lo[0]*mod_inv;
    uint64_t t1=(uint64_t)x_lo[1]*mod_inv;
    t0=t0*mod;
    t1=t1*mod;
    uint64_t res0=x[0]+t0;
    uint64_t res1=x[1]+t1;
//    uint32x2_t result={(uint32_t)(res0>>32),(uint32_t)(res1>>32)};
//    uint32x2_t result={(uint32_t)res0,(uint32_t)res1};

    uint32_t r0=(uint32_t)(res0>>32);
    uint32_t r1=(uint32_t)(res1>>32);
    if (r0>=mod)r0-=mod;
    if (r1>=mod)r1-=mod;
    uint32x2_t result={r0,r1};
    return result;//取低 32bit 作为结果
}
```

```
uint32x4_t mc(uint32x4_t A,uint32x4_t B,int mod) {
    constexpr uint32_t R=0xFFFFFFFFu+1;//2^32
    uint32_t mod_inv=getinv(mod);
```

```
    uint32x2_t al=vget_low_u32(A);
    uint32x2_t ah=vget_high_u32(A);
    uint32x2_t bl=vget_low_u32(B);
    uint32x2_t bh=vget_high_u32(B);
```

```
    uint64x2_t awa1=vmull_u32(al,bl);
    uint64x2_t awa2=vmull_u32(ah,bh);
```

```
    uint32x2_t res1=montgomery_reduce_u64(awa1,mod,mod_inv);
    uint32x2_t res2=montgomery_reduce_u64(awa2,mod,mod_inv);
```

```
    uint32x4_t result=vcombine_u32(res1,res2);
    return result;
}
```

这里参考了 Newton-Raphson 迭代的思路来求逆元，避免了使用 exgcd。[2]

这份代码运行结果错误，但由于时间有限，截至我写下这段话，已经是 4.27 的凌晨 4 点钟了。故我没有更多的精力去进一步调试。

总的来说，我实现了从朴素多项式到串行 NTT，再到向量基本优化的并行 NTT 的实现；对于最后一块高难度目标蒙哥马利优化，我做出了自己的尝试，但并没有完全成功。

以下的性能分析，是基于我编写的代码中正确性无误的最高级算法而言的。

四．实验分析

由于我比较菜，不太会使用服务器，又怕搞崩了，所以性能方面并没有在服务器进行大量测试。更多地是在本地模拟环境下进行的性能测试。

根据要求，我需要**保留一份多项式乘法的朴素运行结果**。以下是结果截图：

```
    version=version.VERSION)
AttributeError: module 'version' has no attribute 'VERSION'
多项式乘法结果正确
average latency for n=4 p=7340033 : 0.00015 (us)
多项式乘法结果正确
average latency for n=131072 p=7340033 : 94498 (us)
```

可以看到，在连 NTT 都没有使用的时候，朴素的乘法效率及其低下，在 1e5 的规模下跑到了 9.4 万 us。

而以下是基于 perf 工具获得的串行版本 NTT 的性能测试数据，p 取 104857601：

n	时间 1（us）	时间 2（us）	时间 3（us）	时间 4（us）	时间 5（us）	平均（us）
1000	0.521	0.519	0.527	0.525	0.524	0.5232
10000	5.426	5.424	5.391	5.407	5.431	5.4158

100000	59.805	59.990	59.811	59.902	59.806	59.8628

表 1 串行耗时数据

以下是最终版本并行 NTT 的性能测试数据：

n	时间 1（us）	时间 2（us）	时间 3（us）	时间 4（us）	时间 5（us）	平均（us）
1000	0.509	0.510	0.502	0.521	0.523	0.513
10000	5.337	5.360	5.328	5.334	5.313	5.3344
100000	53.846	54.213	54.069	53.990	53.882	54.000

表 2 并行耗时数据

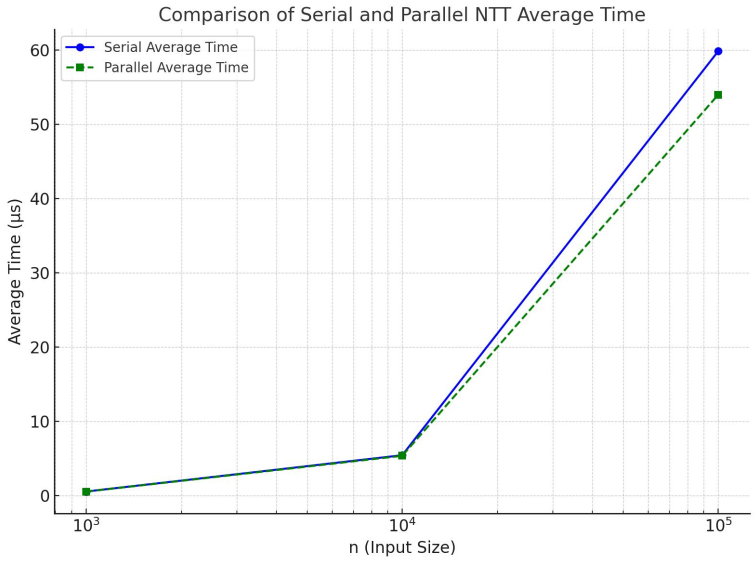

图 1 耗时对比

可以看到，在 1e5 的规模下，后者的效率比前者快了约 9.8%；并且差距会随着数据规模的增大而拉大，见图 1。这可以见得并行算法的优越性。如果进一步实现了规约优化，算法的效率还会变得更高。

根据我的理解，出现这样结果的**内在原因**是通过同时执行多个计算任务来最大化硬件资源的利用。通过采用 SIMD 指令集，多个数据元素可以在单个指令周期内并行处理，从而显著提高数据吞吐量。此外，合理的负载均衡和任务划分能有效减少处理器空闲时间和通信开销，优化内存访问策略（如减少缓存未命中和数据传输延迟）进一步减少了性能瓶颈。最终，通过这些优化，计算过程中的并

行度提高，延迟降低，从而实现显著的性能提升。

接下来我们就**可能的体系结构方面进行进一步优化分析**。由于 NTT 的计算涉及多个轮次，每一轮的频繁数据访问可能导致缓存失效。为了提高缓存命中率并减少内存访问延迟，似乎可以采用数组块化存储等策略，使得同一轮次或相邻轮次的数据能被放置在缓存友好的位置，从而有效提高缓存的利用率。

此外，NTT 的核心计算依赖蝴蝶变换（butterfly operation），在该过程中，数据需要频繁交换与操作。通过调整蝴蝶变换的执行顺序，可以最大程度地减少不必要的缓存失效。改进数据重排策略，确保相邻数据元素在处理时能够同时位于缓存中，能够有效降低缓存未命中的概率。

五. 总结

通过此次作业，我对使用 SIMD 指令优化 NTT 算法的整个过程有了更深刻的理解。首先是简单的引入向量化操作，随后我尝试引入蒙哥马利规约，在理论层面上完善了对其认识。

进一步地，我也对 ARM 平台的 neon 指令集有了更准确的认识。这些对我今后的学习打下了深刻的基础。

六. 项目链接

https://github.com/Hsy23333/NTT_neon

七. 参考文献

[1] moonzzz. 基于 NTT 的多项式乘法算法原理与实现 [EB/OL]. https://www.cnblogs.com/moonzzz/p/17806496.html, 2023-10-19.

[2] 逸珺. 数学之美：牛顿-拉夫逊迭代法原理及其应用， 知乎专栏. https://zhuanlan.zhihu.com/p/266566509 (accessed Apr. 27, 2025).