# SIMD编程：基于neon的NTT优化

### 2313911 黄尚扬

### 2025.4.26

**目录**

[一． 引言 3](#_Toc30974)

[二． 问题描述 3](#_Toc23062)

[三． 算法设计 4](#_Toc19715)

[四． 实验分析 9](#_Toc11792)

[五． 总结 1](#_Toc10272)1

[六． 项目链接 1](#_Toc16954)1

[七． 参考文献 11](#_Toc31549)

##### 引言

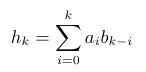
SIMD(Single Instruction, Multiple Data)是一种并行计算技术，它通过向量寄存器存储多个数据元素，并使用单条指令同时对这些数据元素进行处理，从而提高了计算效率。SIMD已被广泛应用于需要大量数据并行计算的领域，包括图像处理、视频编码、信号处理、科学计算等。许多现代处理器都提供了SIMD指令集扩展，例如x86平台的SSE/AVX，以及ARM平台的NEON。[1]

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。

##### 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)=an−1xn−1+...+a1x+a0和多项式g(x)=bm−1xm−1+...+b1x+b0，设h(x)=f(x)g(x)，则有h(x)的xk项系数hk为：



朴素的多项式乘法计算需要O(n2)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amn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\*2r+1的素数p以及一列n<=k (k=2q)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ap-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。

##### 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2k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(mask2,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：



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。

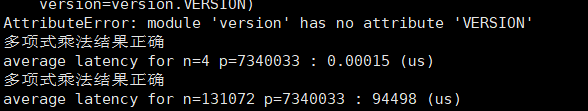
**总的来说，我实现了从朴素多项式到串行NTT，再到向量基本优化的并行NTT的实现；对于最后一块高难度目标蒙哥马利优化，我做出了自己的尝试，但并没有完全成功。**

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。

##### 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：



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

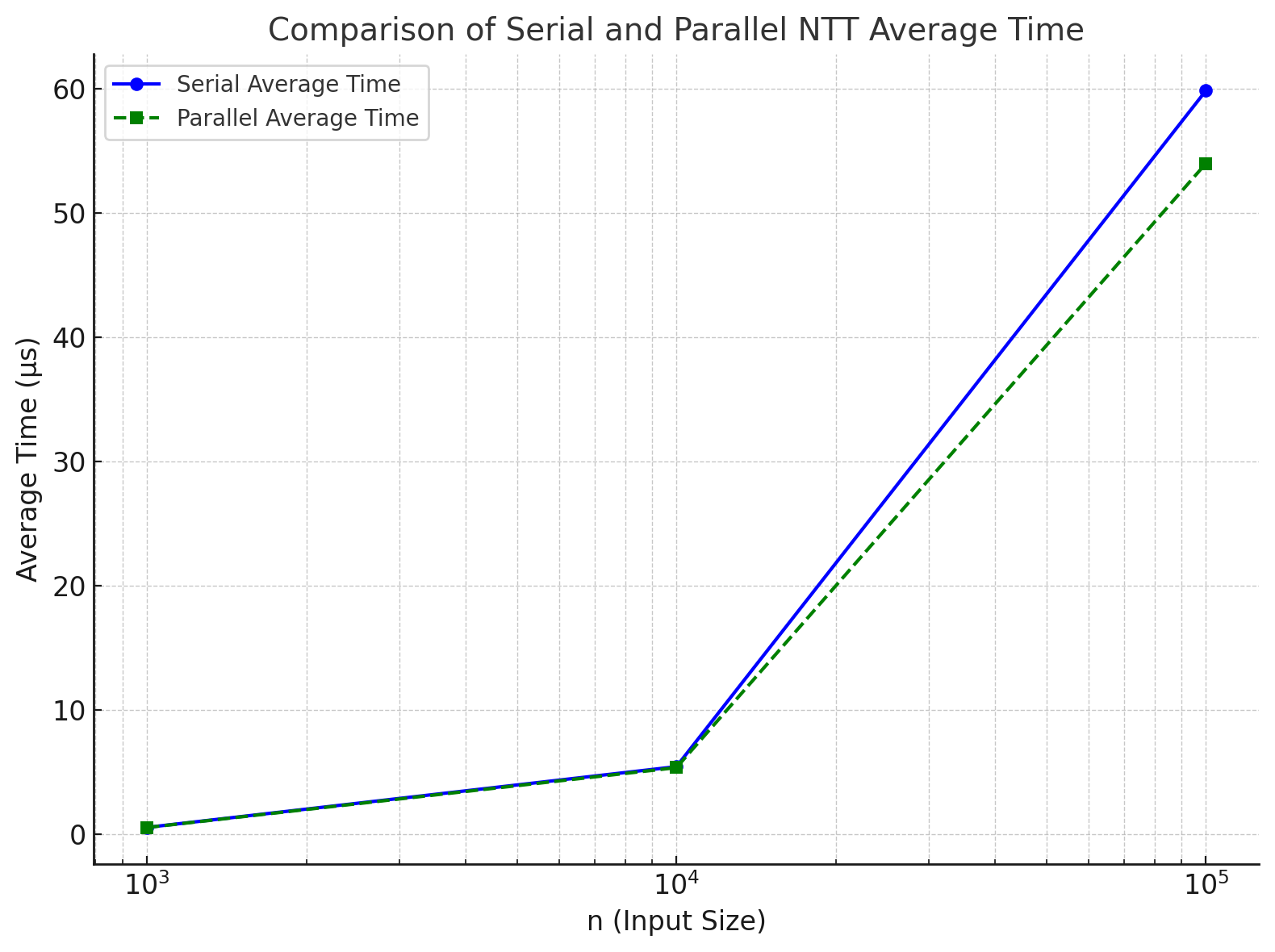
****

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。

##### 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。

##### 项目链接

https://github.com/Hsy23333/NTT\_neon

##### **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).