HI Kyber:基于GPU的高性能KYber实现

HI-Kyber: A novel high-performance implementation scheme of Kyber based on GPU

汇 报 人： 袁 智 健

指 导 老 师： 董 建 阔

南京邮电大学 Nanjing University of Posts and Telecommunications

目录

CONTENT

研究背景-符合密码安全需求

量子计算

- 基于 Shor 和 Grover 算法的量子计算可以在多项式时间内求解离散对数、大整数因子分解等问题，这意味着在未来的量子计算时代，所有基于这些困难问题的**公钥密码算法都不能安全使用**

后量子密码

- NIST 从 2016 年便启动了全球范围内的后量子密码标准征集，于 2022 年公布了后量子密码算法标准化过程的第三轮结果，包括四个选定的算法和四个候选算法；**Kyber 是唯一的公钥加密算法**

CRYSTALS-Kyber
CRYSTALS-DILITHIUM
Falcon
SPHINCS+

性能问题

- 软硬件平台上的性能优化实现是后量子密码算法能够成功应用到工业界的关键；高复杂度算法增加了部署成本；**主流的后量子密码在吞吐性能、空间占用等方面存在诸多瓶颈与问题**

提高后量子密码计算效率，弥补密码迁移工作时空效率差距，是亟待解决的科学难题

GPU平台下的密码加速-符合业界研究需求

CPU VS GPU

- 强劲的算术逻辑单元，拥有复杂的ALU结构，主要设计用于通用计算
- 复杂的控制器
- 较大缓存Cache，采用更复杂的缓存管理策略

- 高效能的算术逻辑单元，拥有大规模的SIMD结构，更适用于大规模并行计算
- 简单的控制器，适合流式处理、相似操作
- Cache规模较小，局部性原理较为重要

业界已开展很多工作

Gupta 等人

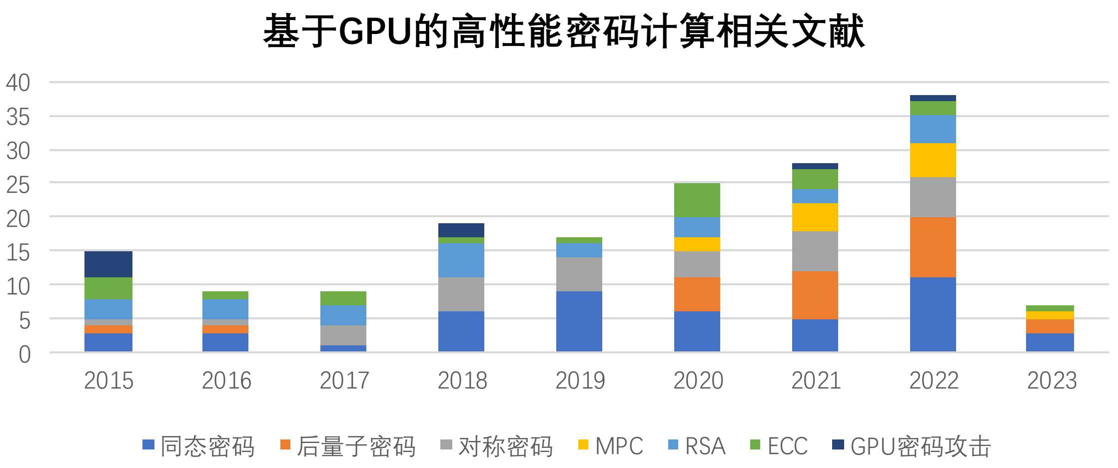
- 在 Volta 架构的 NVIDIA GPU Tesla V100中实现了 Kyber1024子函数的优化，特别是数论变换 NTT 和 Keccak

Lee 等人

- 在 NVIDIA GPU RTX2060上为数论变换 NTT 实现三种不同的细粒度并行方法

Wan 等人

- 利用AI加速器Tensor Core加速多项式乘法，获得了更好的性能提高



GPU平台的并行计算能力**强大**，适合加速**密码算法**等计算复杂度高的任务

背景知识-CRYSTAL-Kyber

Kyber 是一种 IND-CCA2 安全的密钥封装机制（KEM），用非对称加密的思想去封装密钥并进行密钥协商的算法，其安全性正是基于求解模格上错误学习问题困难假设。

算法 5 KYBER.CPAPKE.KeyGen(): 密钥生成

输出：私钥 sk，公钥 pk.

1: $d \leftarrow \boldsymbol{Random}()$ ▹ 随机数生成
2: $(\rho, \sigma) := \mathrm{G}(d)$ ▹ 通过确定性算法操作随机数以获得密钥参数
3: $\hat{\mathbf{A}} \leftarrow Gen\_matrix\_\hat{\mathbf{A}}(\rho)$, $\hat{\mathbf{A}} \in R_q^{k\times k}$ in NTT domain ▹ 生成公钥参数$\hat{\mathbf{A}}$
4: $\mathbf{s} \leftarrow Sample\_s(\sigma)$, $\mathbf{s} \in R_q^k$ from B_{η_1} ▹ 生成秘密 $\mathbf{s}$
5: $\mathbf{e} \leftarrow Sample\_e(\sigma)$, $\mathbf{e} \in R_q^k$ from B_{η_1} ▹ 生成噪声参数$\mathbf{e}$
6: $\hat{\mathbf{s}} := NTT(\mathbf{s})$ ▹ 对秘密参数进行自定义 NTT 变换
7: $\hat{\mathbf{e}} := NTT(\mathbf{e})$ ▹ 对噪声参数进行自定义 NTT 变换
8: $\hat{\mathbf{t}} := \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$ ▹ 逐项乘法，再加上噪声
9: **return** $pk := Encode(\hat{\mathbf{t}} \| \rho)$, $sk := Encode(\hat{\mathbf{s}})$ ▹ 对公私钥对进行编码

算法 6 KYBER.CPAPKE.Enc(): 加密

输入：公钥 pk，消息明文 m，随机数种子 r

输出：密文 c

1: $(\hat{\mathbf{t}}, \rho) \leftarrow Decode(pk)$ ▹ 对公钥进行解码
2: $\hat{\mathbf{A}}^T \leftarrow Gen\_matrix\_\hat{\mathbf{A}}^T(\rho)$, $\hat{\mathbf{A}}^T \in R_q^{k\times k}$ in NTT domain
3: $\mathbf{r} \leftarrow Sample\_r(r)$, $\mathbf{r} \in R_q^k$ from B_{η_1}
4: $\mathbf{e}_1 \leftarrow Sample\_e_1(r)$, $\mathbf{e}_1 \in R_q^k$ from B_{η_2}
5: $e_2 \leftarrow Sample\_e_2(r)$, $e_2 \in R_q$ from B_{η_2}
6: $\hat{\mathbf{r}} := NTT(\mathbf{r})$
7: $\mathbf{u} := NTT^{-1}(\hat{\mathbf{A}} \circ \hat{\mathbf{r}}) + \mathbf{e}_1$ ▹ 对相关参数进行逆自定义 NTT 变化
8: $v := NTT^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + Decompress(m)$ ▹ 对明文用到特定的处理方法
9: **return** $c_1 := Encode_u(\mathbf{u})$, $c_2 := Encode_v(v)$ ▹ 对加密结果以特定方法编码

算法 7 KYBER.CPAPKE.Dec(): 解密

输入：密钥 sk，密文 c

输出：消息明文 m

1: $\mathbf{u} := Decode_u(c)$ ▹ 先对参数解码
2: $v := Decode_v(c)$
3: $\hat{\mathbf{s}} := Decode(sk)$
4: **return** $m := Compress(v - NTT^{-1}(\hat{\mathbf{s}} \circ NTT(\mathbf{u})))$ ▹ 含有特定处理过程

背景知识-快速数论变换NTT

NTT（Number Theoretic Transform）在Kyber算法用于实现多项式的变换。
具体来说，Kyber使用NTT来进行多项式的前向变换和逆向变换，

多项式的乘法

1. 系数乘法

$A(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$

$B(x) = b_0 + b_1x + b_2x^2 + \cdots + b_{n-1}x^{n-1}$

$C(x) = c_0 + c_1x + c_2x^2 + \cdots + c_{2n-2}x^{2n-2}$

2. 点值乘法

给定 n 个点可以确定 $n-1$ 次函数曲线的系数

点值和系数存在映射关系，可以相互转化

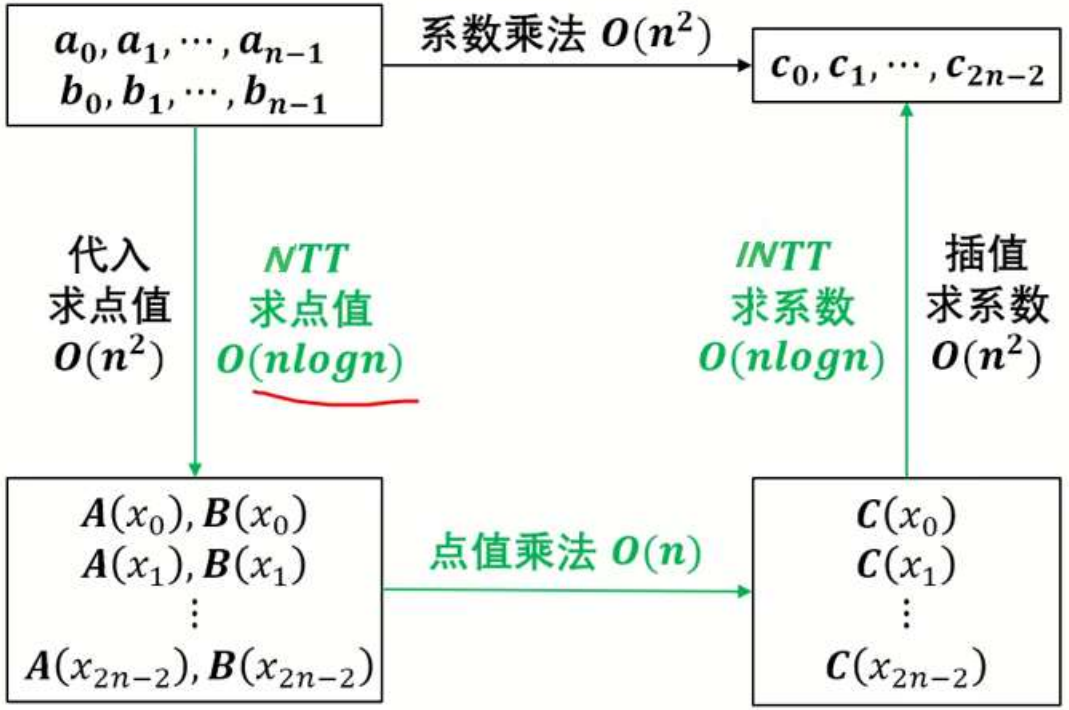
(1) 取 $x_0, x_1, x_2, \cdots x_{2n-2}$ 分别代入，求点值

$A(x)$: $y_0, y_1, y_2, \cdots, y_{2n-2}$

$B(x)$: $y'_0, y'_1, y'_2, \cdots, y'_{2n-2}$

(2) 相乘 $C(x)$: $y_0y'_0, y_1y'_1, y_2y'_2, \cdots, y_{2n-2}y'_{2n-2}$

(3) 插值 $C(x)$: $c_0, c_1, c_2, \cdots, c_{2n-2}$



时间复杂度分析

背景知识-快速数论变换NTT

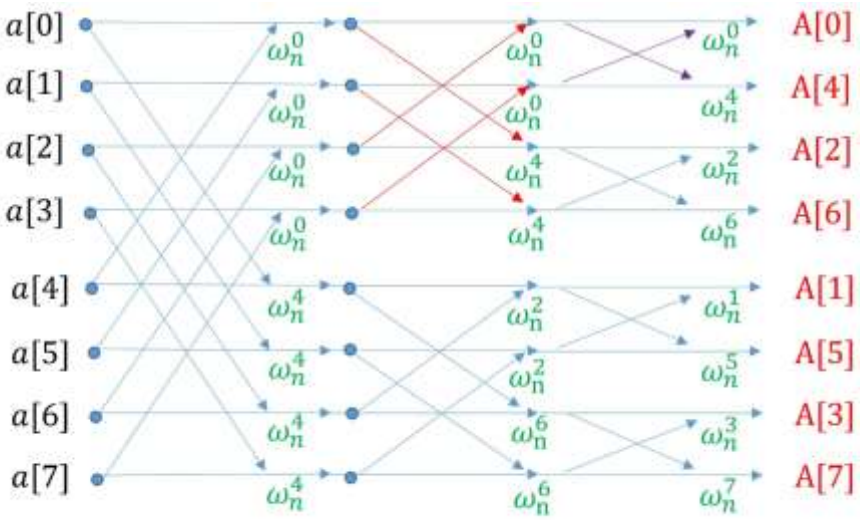
蝴蝶变换是在快速数论变换（FFT）和数论变换（NTT）等算法中的关键步骤之一，
用于实现多项式的点值表示和系数表示之间的转换。
核心思想是将一个多项式的点值表示划分为两个子集，然后通过乘上旋转因子进行组合。

蝴蝶变换的数学表达式如下：

$$X_k = W_N^{jk} \cdot (X_j + W_N^{N/2} \cdot X_{j+N/2})$$
$$X_{k+N/2} = W_N^{jk+N/2} \cdot (X_j - W_N^{N/2} \cdot X_{j+N/2})$$

其中：

- X_j 和 $X_{j+N/2}$ 是原始多项式的两个子集。
- X_k 和 $X_{k+N/2}$ 是结果多项式的两个对应子集。
- W_N 是N次单位根，也就是旋转因子。



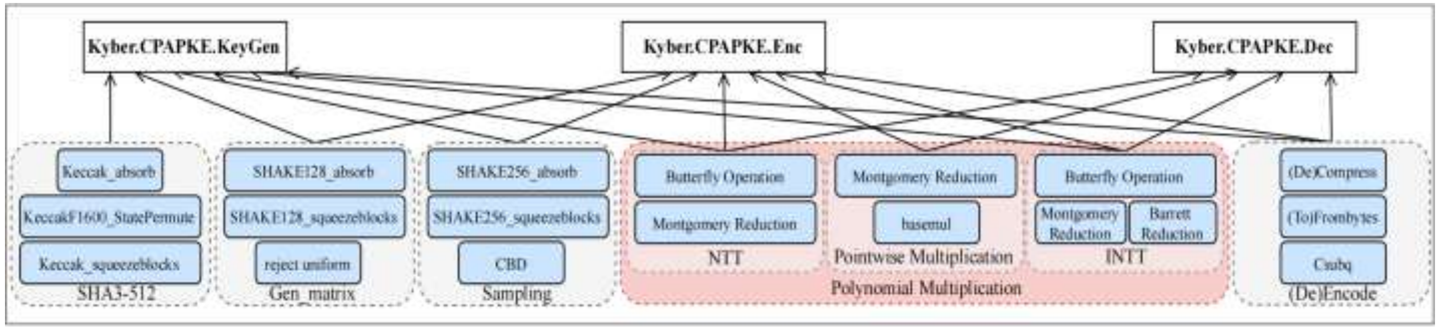
目录

CONTENT

02

总体框架- 基于 kernel 融合的设计

Kyber 整体的算法框架:



Kyber 分为三部分:
1） KeyGen
2） Encryption
3） Decryption

	NIST Security Levels
Kyber512	1
Kyber768	3
Kyber1024	**5**

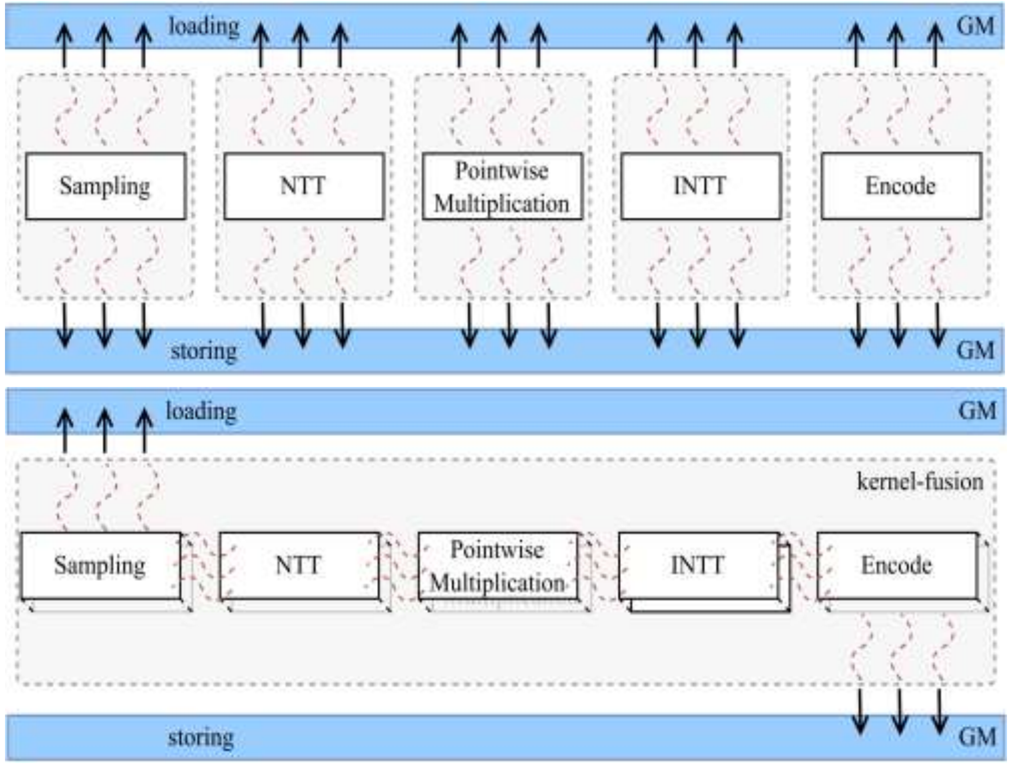
其中最耗时的算法之一：**多项式乘法**
1）**数论变换 NTT**
2）点乘算法 Pointwise Multiplication
3）逆向数论变换 INTT

NTT 一直是研究热点 （Dilithium、NTRU、Saber等）

HI-Kyber 主要针对 NTT 的 GPU 实现提出优化

本工作将与：
- ✓ 基于相同指令架构的 GPU [1] 实现做对比（**TPDS 2020** Gupta 等人的工作）
- ✓ 目前性能最优的 GPU [2] 实现做对比（**ESORICS 2022** Wan 等人的工作）

多 kernel 模式 [2] VS kernel 融合模式:



不同的并行方式导致访问全局内存的次数:
Kernel 融合：避免全局内存的访问

Parallelization	KeyGen	Encryption	Decryption
multi-kernel	12	17	8
kernel-fusion	1	1	1

NTT – 最耗时的核心算法之一

NTT: $\mathbf{v} = [v_0, v_1, \ldots\ldots, v_{n-1}] \Rightarrow \mathbf{V} = [V_0, V_1, \ldots\ldots, V_{n-1}]$

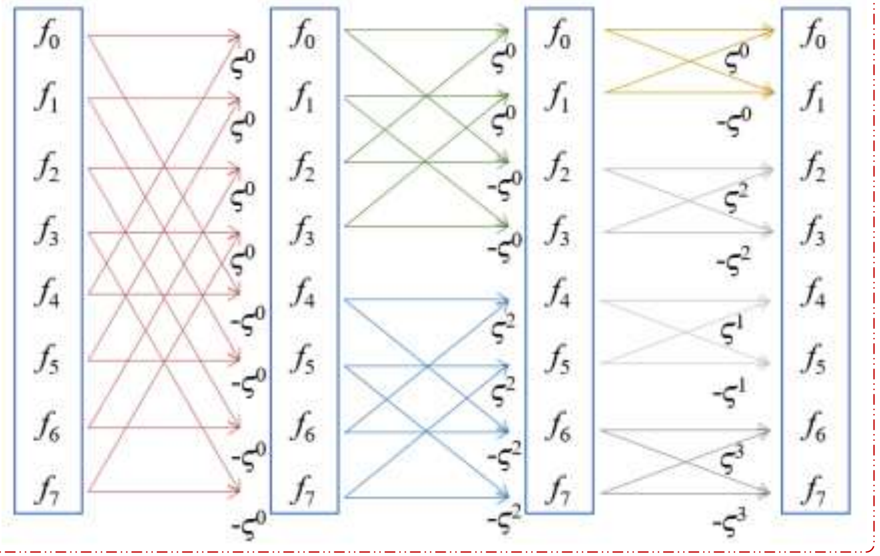
$$\mathbf{V_i} = \sum_{j=0}^{n-1} v_j \cdot \varsigma^{ij}$$

Chinese Remainder Theorem(CRT):

$$NTT(f) = \hat{f}_0 + \hat{f}_1 X^1 + \ldots\ldots + \hat{f}_{255} X^{255}$$

$$\begin{bmatrix} \hat{f}_0 \\ \hat{f}_2 \\ \hat{f}_4 \\ \vdots \\ \hat{f}_N \end{bmatrix} = \begin{bmatrix} 1 & \varsigma^{0\times1} & \cdots & \varsigma^{0\times(\frac{N}{2}-1)} \\ 1 & \varsigma^{1\times1} & \cdots & \varsigma^{1\times(\frac{N}{2}-1)} \\ 1 & \varsigma^{2\times1} & \cdots & \varsigma^{2\times(\frac{N}{2}-1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \varsigma^{\frac{N}{2}\times1} & \cdots & \varsigma^{\frac{N}{2}\times(\frac{N}{2}-1)} \end{bmatrix} \begin{bmatrix} f_0 \\ f_2 \\ f_4 \\ \vdots \\ f_N \end{bmatrix} \quad \begin{bmatrix} \hat{f}_1 \\ \hat{f}_3 \\ \hat{f}_5 \\ \vdots \\ \hat{f}_{N-1} \end{bmatrix} = \begin{bmatrix} 1 & \varsigma^{0\times1} & \cdots & \varsigma^{0\times(\frac{N}{2}-1)} \\ 1 & \varsigma^{1\times1} & \cdots & \varsigma^{1\times(\frac{N}{2}-1)} \\ 1 & \varsigma^{2\times1} & \cdots & \varsigma^{2\times(\frac{N}{2}-1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \varsigma^{\frac{N}{2}\times1} & \cdots & \varsigma^{\frac{N}{2}\times(\frac{N}{2}-1)} \end{bmatrix} \begin{bmatrix} f_1 \\ f_3 \\ f_5 \\ \vdots \\ f_{N-1} \end{bmatrix}$$

以8个点执行3层蝴蝶运算为例

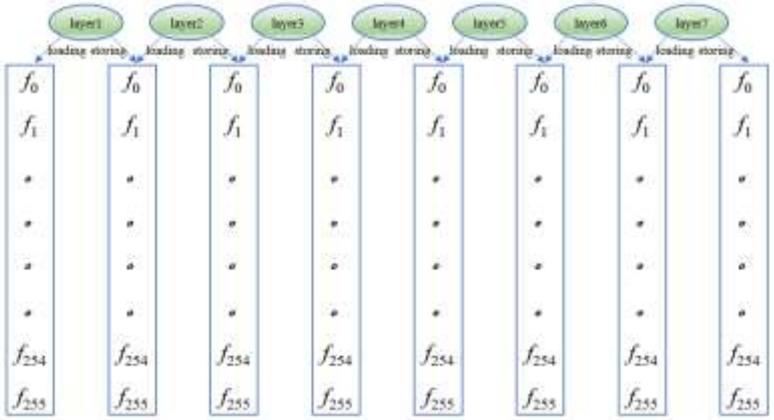


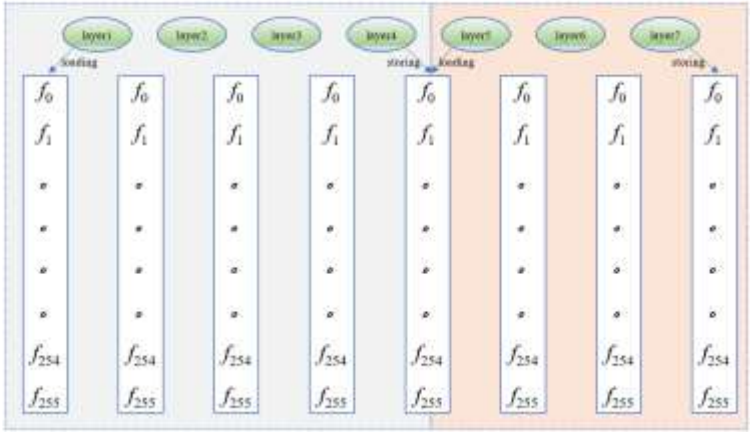
Kyber:

NTT Variant

$\log_2 N - 1$

$N = 256$





[1] 在 Cortex-M4 平台提出层合并的思想，通过重复利用数据多层减少取系数的次数

Loading and Storing:

$(\log_2 N - 1) * N \quad 2 * N$

基于此我们提出

- a. SLM
- b. SDFS
- c. EDFS

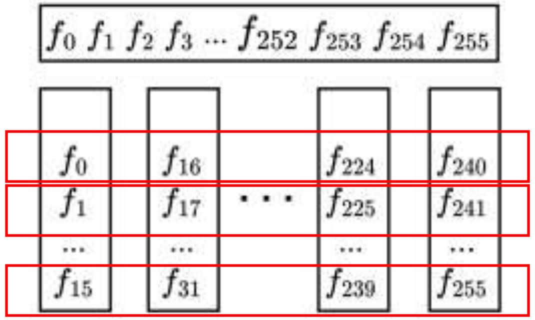
[1] E. Alkim, P. Jakubeit, and P. Schwabe, “Newhope on ARM Cortex-M,” in *Security, Privacy, and Applied Cryptography Engineering: 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings*. Springer, 2016, pp. 332–349.

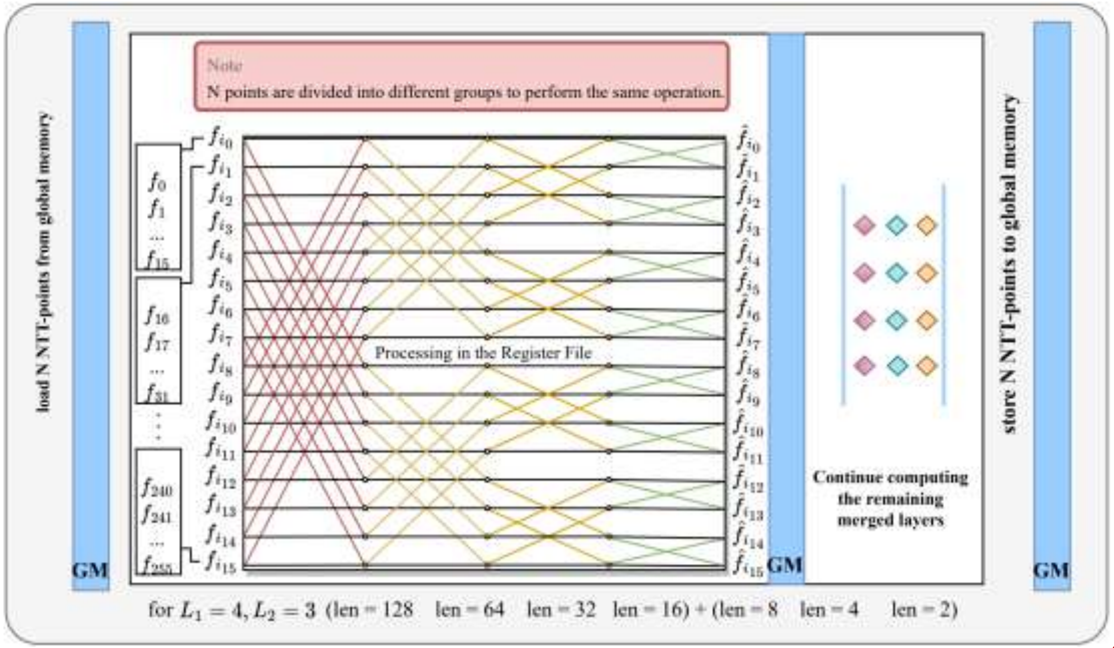
方案一：切片式层合并 SLM

切片的含义:
以 Kyber 中 N = 256 为例

共16组 | 每组16个系数

$Group_0$
$Group_1$
$Group_{15}$





$$\log_2 N - 1 = L_1 + L_2 + \cdots + L_n$$

n:合并层的个数　　L_i:每个合并层的层数

SLM:将 NTT 的系数量 N 分成 j 个切片
每个切片的第 k 个数据组成一组做蝶蝶运算

Algorithm 1: A sliced layer merging scheme of L_i for NTT

input : $f(x) \in \mathbb{Z}_q[X]/(X^n+1), \zeta^n \in \mathbb{Z}_q$
output: $\hat{f}(x) \in \mathbb{Z}_q[X]/(X^n+1)$, after L_i Layer Merging

```
L_finished ← Σ_{j=1}^{i-1} L_j
MAX_Group ← N ≫ (L_finished + L_i)
for Group ← 0 to MAX_Group do
    k ← N / (N≫(L_finished))
    for len ← N ≫ (L_finished + 1) to
      N ≫ (L_finished + L_i) do
        /* shift right one layer */
        for start ← Group to N do /* step by
          j + len */
            zeta ← ζ^{k++}
            for j ← start to start + len do
                /* step by MAX_Group */
                Butterfly Unit(f[j + len], f[j])
```

当分组的数据达到一定程度时，数据可以直接在寄存器中处理

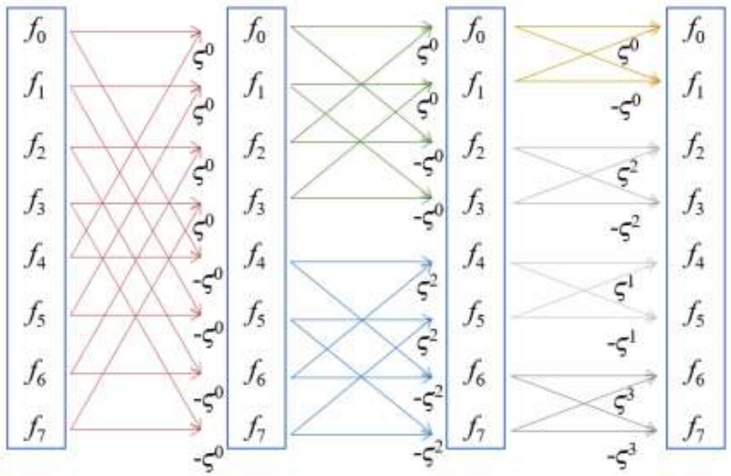
切片的片数 j 是根据合并层中最后一层的蝴蝶运算距离来决定的

寄存器是否冗余:
- 若分片数量太少，则每组需要处理的数据量可能会超过可用寄存器的数量;
- 若分片数量太多，则每组需要处理的数据量可能会发生寄存器冗余

默认情况是选择最大切片片数来使每组中数据的加载量为**最少可执行系数**

DFS-深度优先搜索

NTT 基于分而治之的思想



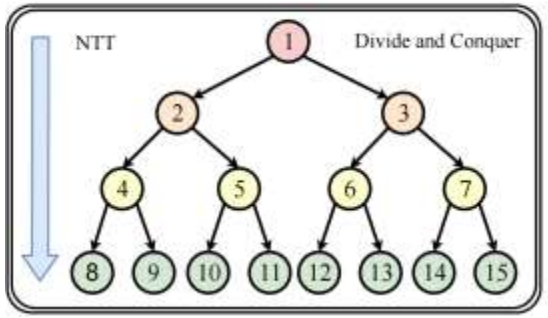
拥有相同旋转因子的系数看成一个节点

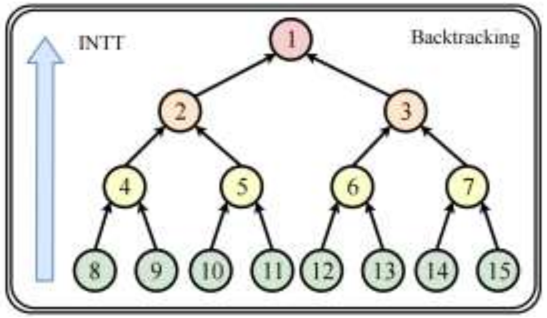
第一层 one node
第二层 two nodes
以此类推

NTT

执行 BFS 的满二叉树

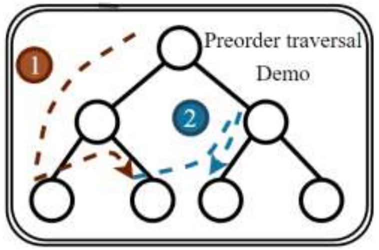
NTT：将 解决大问题 分解成 解决小问题
INTT：由 解决小问题 组成 大问题的解





提出**深度优先搜索策略**（the depth-first search，DFS），以先序方式处理节点

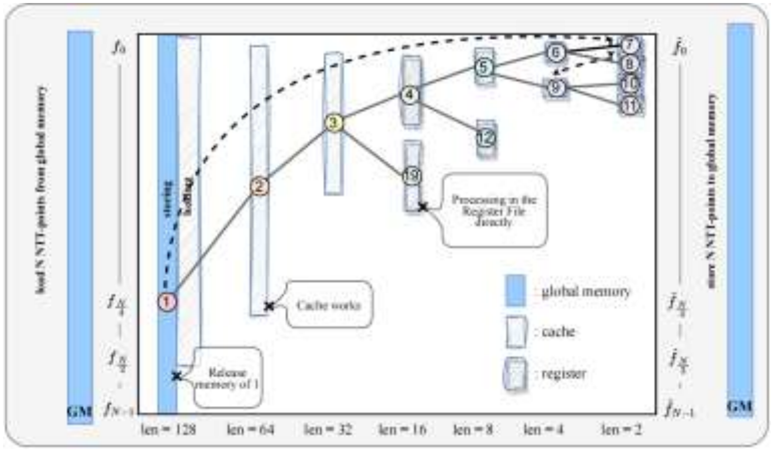
从根节点出发，沿着左子树的方法进行纵向遍历，直到找到叶子结点为止。然后回溯到前一个节点，进行右子树节点的遍历，直到遍历完所有可到达节点止。



在 DFS 中，每层需要处理的系数量是逐层减半的

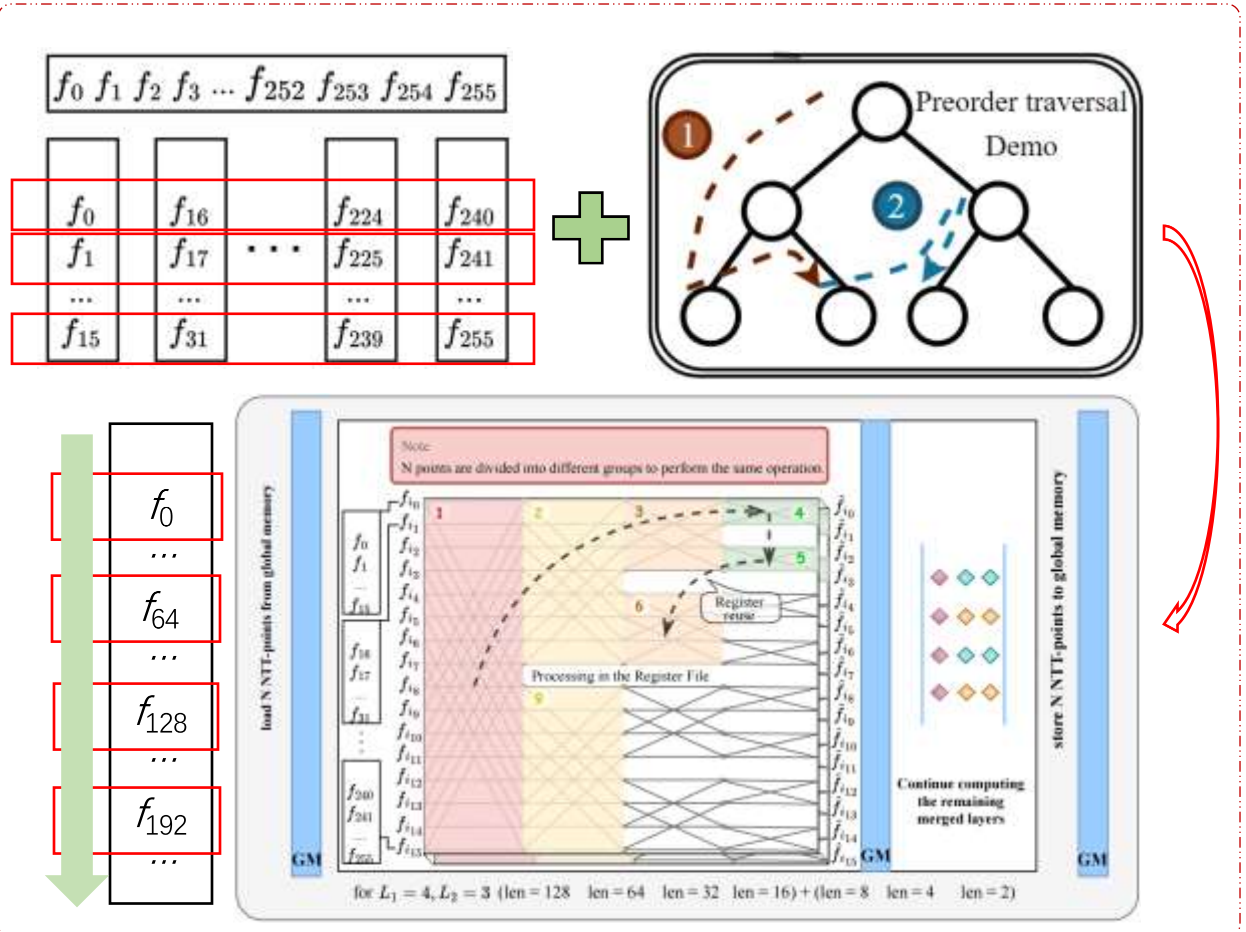
- 优化寄存器的使用
- 增加 Cache 击中率

层合并　　SDFS
DFS　　EDFS



方案二：切片式深度优先搜索SDFS

SLM ➕ DFS ➡ SDFS



DFS 与切片结合做切片遍历

即减少全局内存的访问次数

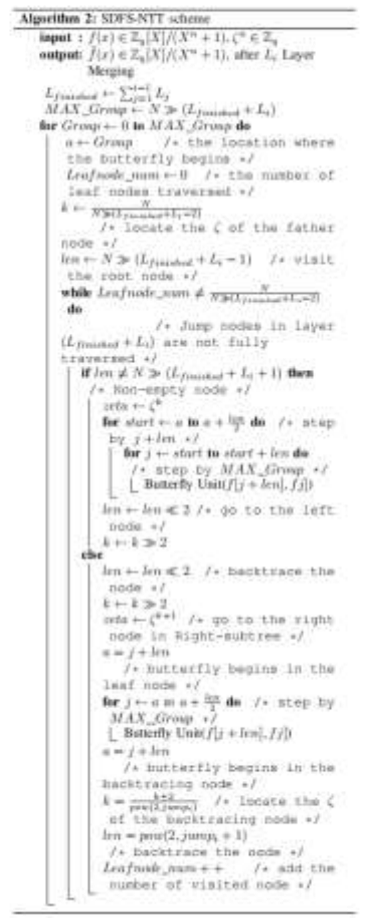
又能高效复用寄存器

当分组的数据达到一定程度时，数据可以直接在寄存器中处理

有了闲置的寄存器，编译器就会提前安排其他可用工作

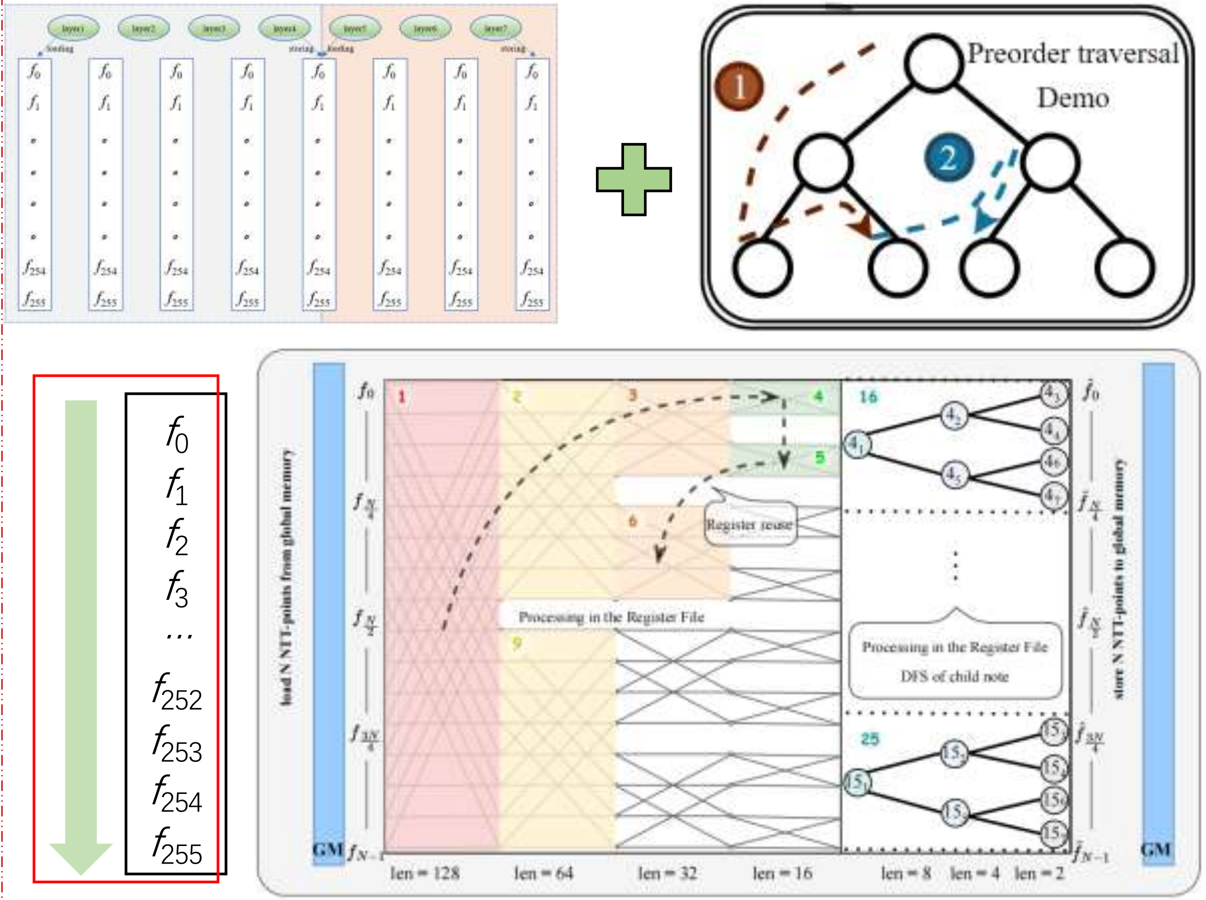
寄存器是否冗余:

- 过载（编译器的工作）
- 冗余（大量实验证明）



方案三：整体深度优先搜索EDFS

LM ➕ DFS ➡ EDFS



DFS 与分层的思想结合做整体遍历

即减少全局内存的访问次数

又能高效复用寄存器

还能访问连续的内存地址

重点注重

- 内存层次结构优化
- 高效利用 GPU 的计算型资源

在具体的实验方案中，作为 GPU 方案的设计人员，尽管我们无法精确控制每个寄存器保存的具体数据，尽可能地重复使用寄存器是至关重要的。

Algorithm 3: EDFS-NTT scheme

```
input : f(x) ∈ Z_q[X]/(X^n + 1), ζ^n ∈ Z_q
output: f̂(x) ∈ Z_q[X]/(X^n + 1), after L_i Layer
        Merging
L_finished ← Σ_{j=1}^{i-1} L_j
Leafnode_num ← 0        /* the number of
 leaf nodes traversed */
a ← 0         /* the location where the
 butterfly begins */
while Leafnode_num ≠ N / (N ≫ (L_finished + L_i − 2)) do
            /* Leaf nodes are not fully
traversed */
  if len ≠ N ≫ (L_finished + L_i + 1) then
    /* Non-empty node */
    zeta ← ζ^k
    for start ← a to a + len/2 do  /* step by
     j + len  */
      for j ← start to start + len do
        Butterfly Unit(f[j + len], f[j])
    len ← len ≪ 2     /* go to the left
     node */
    k ← k ≫ 2
  else
    len ← len ≪ 2      /* backtrace the
     node */
    k ← k ≫ 2 zeta ← ζ^{k+1}  /* go to the
     right node in Right-subtree */
    a = j + len
      /* CT butterfly begins in the
    leaf node */
    for j ← a to a + len do
      Butterfly Unit(f[j + len], f[j])
    a = j + len
      /* CT butterfly begins in the
    next node */
    k = (k+2) / pow(2, jump_i)  /* locate the ζ of
     the backtracing node */
    len = pow(2, jump_i + 1)
    Leafnode_num++           /* add the
     number of visited node */
```

Comparison-三种方案对比

- 对每一种方案都做了详细分析，考虑和设计；总的来说，设计了三种快速实现 NTT 的方案：SLM、SDFS 和 EDFS

LM的收益：
[1] 减少全局内存的访问次数

切片的收益：
[2] 当分组的数据达到一定程度时，数据可以直接在寄存器中处理

DFS的收益：
[3] 当逐层减半的数据达到一定程度时，数据可以直接在寄存器中处理
[4] 提前结束任务和高效复用寄存器
[5] 增大 Cache 的击中率

整体遍历的收益：
[6] 访问连续的内存空间

SLM

SDFS

EDFS

Scheme \ Benefit	[1]	[2]	[3]	[4]	[5]	[6]
SLM	√	√				
SDFS-NTT	√	√	√	√	√	
EDFS-NTT	√		√	√	√	√

旨在注重方案设计的合理性以及无论从并行、内存和指令，探索最适配于 GPU 的加速方案

南京郵電大學
Nanjing University of Posts and Telecommunications

目录

CONTENT

03

实验测试

- 对 SLM做了 1+6，2+5，3+4，4+3，5+2，6+1，2+2+2+1，3+3+1 的分层的数据吞吐测试；逐步增加数据量测试寄存器冗余

```
case 0:
    printf("TEST NTT_SLM_SPEED_TEST\n");
    NTT_SPEED_TEST(g_gridSize, g_blockSize,g_packSize);
    NTT_ML1P6(g_gridSize, g_blockSize,g_packSize);
    NTT_ML2P5(g_gridSize, g_blockSize,g_packSize);
    NTT_ML3P4(g_gridSize, g_blockSize,g_packSize);
    NTT_ML4P3(g_gridSize, g_blockSize,g_packSize);
    NTT_ML5P2(g_gridSize, g_blockSize,g_packSize);
    NTT_ML6P1(g_gridSize, g_blockSize,g_packSize);
    NTT_ML2P2P2P1(g_gridSize, g_blockSize,g_packSize);
    NTT_ML3P3P1(g_gridSize, g_blockSize,g_packSize);
    break;
```

```
TEST NTT ML SPEED TEST
NTT native OUT TimeUsed: 6498.228000 ms, Throughput: 47274426.197419 ops/sec, Latency: 0.649823 ms.
Checking resuts!!!
correctTest is 30720 in 30720
NTT GPU = CPU correctTest passed!
NTT ML1P6 OUT TimeUsed: 9936.648000 ms, Throughput: 30909636.803718 ops/sec, Latency: 0.993865 ms.
Checking resuts!!!
correctTest is 30720 in 30720
NTT_ML1P6 GPU = CPU correctTest passed!
NTT ML2P5 OUT TimeUsed: 8939.799000 ms, Throughput: 34363188.702565 ops/sec, Latency: 0.893980 ms.
Checking resuts!!!
correctTest is 30720 in 30720
NTT_ML2P5 GPU = CPU correctTest passed!
NTT ML3P4 OUT TimeUsed: 7772.217000 ms, Throughput: 39525401.825502 ops/sec, Latency: 0.777222 ms.
Checking resuts!!!
corectTest is 30720 in 30720
NTT_ML3P4 GPU = CPU corectTest passed!
NTT ML4P3 OUT TimeUsed: 6726.127000 ms, Throughput: 45672643.409796 ops/sec, Latency: 0.672613 ms.
Checking resuts!!!
correctTest is 30720 in 30720
NTT_ML4P3 GPU = CPU correctTest passed!
NTT ML5P2 OUT TimeUsed: 6741.749000 ms, Throughput: 45566810.630298 ops/sec, Latency: 0.674175 ms.
Checking resuts!!!
correctTest is 30720 in 30720
NTT_ML5P2 GPU = CPU correctTest passed!
NTT ML6P1 OUT TimeUsed: 7011.396000 ms, Throughput: 43814384.467801 ops/sec, Latency: 0.701140 ms.
Checking resuts!!!
correctTest is 30720 in 30720
NTT_ML6P1 GPU = CPU correctTest passed!
NTT ML2P2P2P1 OUT TimeUsed: 5949.728000 ms, Throughput: 51632612.448838 ops/sec, Latency: 0.594973 ms.
Checking resuts!!!
correctTest is 30720 in 30720
NTT_ML2P2P2P1 GPU = CPU correctTest passed!
NTT ML3P3P1 OUT TimeUsed: 6153.631000 ms, Throughput: 49921745.309023 ops/sec, Latency: 0.615363 ms.
Checking resuts!!!
correctTest is 30720 in 30720
NTT_ML3P3P1 GPU = CPU correctTest passed!
```

```
case 1:
    printf("TEST NTT_SLM_LOAD_NUM_SPEED_TEST\n");
    NTT_SPEED_TEST(g_gridSize, g_blockSize,g_packSize);
    NTT_ML1P6(g_gridSize, g_blockSize,g_packSize);
    NTT_ML2P5_LOAD4(g_gridSize, g_blockSize,g_packSize);
    NTT_ML2P5_LOAD8(g_gridSize, g_blockSize,g_packSize);
    NTT_ML2P5_LOAD16(g_gridSize, g_blockSize,g_packSize);
    NTT_ML2P5_LOAD32(g_gridSize, g_blockSize,g_packSize);
    NTT_ML2P5_LOAD64(g_gridSize, g_blockSize,g_packSize);
    NTT_ML2P5_LOAD128(g_gridSize, g_blockSize,g_packSize);
    NTT_ML3P4_LOAD8(g_gridSize, g_blockSize,g_packSize);
    NTT_ML3P4_LOAD16(g_gridSize, g_blockSize,g_packSize);
    NTT_ML3P4_LOAD32(g_gridSize, g_blockSize,g_packSize);
    NTT_ML3P4_LOAD64(g_gridSize, g_blockSize,g_packSize);
    NTT_ML3P4_LOAD128(g_gridSize, g_blockSize,g_packSize);
    NTT_ML4P3_LOAD16(g_gridSize, g_blockSize,g_packSize);
    NTT_ML4P3_LOAD32(g_gridSize, g_blockSize,g_packSize);
    NTT_ML4P3_LOAD64(g_gridSize, g_blockSize,g_packSize);
    NTT_ML4P3_LOAD128(g_gridSize, g_blockSize,g_packSize);
    NTT_ML5P2_LOAD32(g_gridSize, g_blockSize,g_packSize);
    NTT_ML5P2_LOAD64(g_gridSize, g_blockSize,g_packSize);
    NTT_ML5P2_LOAD128(g_gridSize, g_blockSize,g_packSize);
    NTT_ML6P1_LOAD64(g_gridSize, g_blockSize,g_packSize);
    NTT_ML6P1_LOAD128(g_gridSize, g_blockSize,g_packSize);
    NTT_ML2P2P2P1_LOADNUM_TEST(g_gridSize, g_blockSize,g_packSize);
    break;
```

```
NTT ML2P5 LOAD4 OUT TimeUsed: 8937.530000 ms, Throughput: 34371912.597776 ops/sec, Latency: 0.893753 ms.
Checking resuts!!!
correctTest is 30720 in 30720
NTT_ML2P5_LOAD4 GPU = CPU correctTest passed!
NTT ML2P5 LOAD8 OUT TimeUsed: 8916.290000 ms, Throughput: 34453791.879887 ops/sec, Latency: 0.891629 ms.
Checking resuts!!!
correctTest is 30720 in 30720
NTT_ML2P5_LOAD8 GPU = CPU correctTest passed!
NTT ML2P5 LOAD16 OUT TimeUsed: 9010.648000 ms, Throughput: 34092987.529146 ops/sec, Latency: 0.901065 ms.
Checking resuts!!!
correctTest is 30720 in 30720
NTT_ML2P5_LOAD16 GPU = CPU correctTest passed!
NTT ML2P5 LOAD32 OUT TimeUsed: 9207.881000 ms, Throughput: 33362724.822356 ops/sec, Latency: 0.920788 ms.
Checking resuts!!!
correctTest is 30720 in 30720
NTT_ML2P5_LOAD32 GPU = CPU correctTest passed!
NTT ML2P5 LOAD64 OUT TimeUsed: 9520.172000 ms, Throughput: 32268324.563880 ops/sec, Latency: 0.952017 ms.
Checking resuts!!!
correctTest is 30720 in 30720
NTT_ML2P5_LOAD64 GPU = CPU correctTest passed!
NTT ML2P5 LOAD128 OUT TimeUsed: 10725.572000 ms, Throughput: 28641829.079139 ops/sec, Latency: 1.072557 ms.
Checking resuts!!!
correctTest is 30720 in 30720
NTT_ML2P5_LOAD128 GPU = CPU correctTest passed!
```

```
NTT ML3P4 LOAD8 OUT TimeUsed: 7776.338000 ms, Throughput: 39504455.696242 ops/sec, Latency: 0.777634 ms.
Checking resuts!!!
corectTest is 30720 in 30720
NTT_ML3P4_LOAD8 GPU = CPU corectTest passed!
NTT ML3P4 LOAD16 OUT TimeUsed: 7772.264000 ms, Throughput: 39525162.880704 ops/sec, Latency: 0.777226 ms.
Checking resuts!!!
corectTest is 30720 in 30720
NTT_ML3P4_LOAD16 GPU = CPU corectTest passed!
NTT ML3P4 LOAD32 OUT TimeUsed: 8257.246000 ms, Throughput: 37203692.030453 ops/sec, Latency: 0.825724 ms.
Checking resuts!!!
corectTest is 30720 in 30720
NTT_ML3P4_LOAD32 GPU = CPU corectTest passed!
NTT ML3P4 LOAD64 OUT TimeUsed: 8694.066000 ms, Throughput: 35334445.396177 ops/sec, Latency: 0.869407 ms.
Checking resuts!!!
corectTest is 30720 in 30720
NTT_ML3P4_LOAD64 GPU = CPU corectTest passed!
NTT ML3P4 LOAD128 OUT TimeUsed: 10785.322000 ms, Throughput: 28483155.162173 ops/sec, Latency: 1.078532 ms.
Checking resuts!!!
corectTest is 30720 in 30720
NTT_ML3P4_LOAD128 GPU = CPU corectTest passed!
```

```
NTT ML4P3 LOAD16 OUT TimeUsed: 6725.024000 ms, Throughput: 45680134.375729 ops/sec, Latency: 0.672502 ms.
Checking resuts!!!
correctTest is 30720 in 30720
NTT_ML4P3_LOAD16 GPU = CPU correctTest passed!
NTT ML4P3 LOAD32 OUT TimeUsed: 7417.180000 ms, Throughput: 41417358.079486 ops/sec, Latency: 0.741718 ms.
Checking resuts!!!
correctTest is 30720 in 30720
NTT_ML4P3_LOAD32 GPU = CPU correctTest passed!
NTT ML4P3 LOAD64 OUT TimeUsed: 8082.184000 ms, Throughput: 38009528.117647 ops/sec, Latency: 0.808218 ms.
Checking resuts!!!
correctTest is 30720 in 30720
NTT_ML4P3_LOAD64 GPU = CPU correctTest passed!
NTT ML4P3 LOAD128 OUT TimeUsed: 10820.548000 ms, Throughput: 28390429.024482 ops/sec, Latency: 1.082055 ms.
Checking resuts!!!
correctTest is 30720 in 30720
NTT_ML4P3_LOAD128 GPU = CPU correctTest passed!
```

分层为 **2+2+2+1** 时，SLM性能最好

测试了 8 种分层情况的寄存器冗余，增加每组的数据量，实验结果**显示没有冗余**

实验测试

- 对SDFS、EDFS都做了1+6，2+5，3+4，4+3，5+2，6+1，2+2+2+1，3+3+1的分层的数据吞吐测试；最终选择EDFS进行Kyber性能测试

SDFS 性能测试

```
case 2:
    printf("TEST NTT_SlicedTREE_SPEED_TEST\n");
    NTT_ML1P6_TREE(g_gridSize, g_blockSize,g_packSize);
    NTT_ML2P5_TREE(g_gridSize, g_blockSize,g_packSize);
    NTT_ML3P4_TREE(g_gridSize, g_blockSize,g_packSize);
    NTT_ML4P3_TREE(g_gridSize, g_blockSize,g_packSize);
    NTT_ML5P2_TREE(g_gridSize, g_blockSize,g_packSize);
    NTT_ML6P1_TREE(g_gridSize, g_blockSize,g_packSize);
    NTT_ML2P2P2P1_TREE(g_gridSize, g_blockSize,g_packSize);
    NTT_ML3P3P1_TREE(g_gridSize, g_blockSize,g_packSize);
    break;
```

```
TREE NTT ML1P6 OUT TimeUsed: 5954.062000 ms, Throughput: 51589829.951725 ops/sec, Latency: 0.595406 ms.
Checking results!!!
correctTest is 30720 in 30720
NTT_ML1P6_TREE GPU = CPU correctTest passed!
TREE NTT ML2P5 OUT TimeUsed: 5384.267000 ms, Throughput: 57055120.037695 ops/sec, Latency: 0.538427 ms.
Checking results!!!
correctTest is 30720 in 30720
NTT_ML2P5_TREE GPU = CPU correctTest passed!
TREE NTT ML3P4 OUT TimeUsed: 5374.934000 ms, Throughput: 57154190.172382 ops/sec, Latency: 0.537493 ms.
Checking results!!!
corectTest is 30720 in 30720
NTT_ML3P4_TREE GPU = CPU corectTest passed!
TREE NTT ML4P3 OUT TimeUsed: 6245.925000 ms, Throughput: 49184068.012344 ops/sec, Latency: 0.624592 ms.
Checking results!!!
correctTest is 30720 in 30720
NTT_ML4P3_TREE GPU = CPU correctTest passed!
TREE NTT ML5P2 OUT TimeUsed: 5270.183000 ms, Throughput: 58291880.838458 ops/sec, Latency: 0.527018 ms.
Checking results!!!
correctTest is 30720 in 30720
NTT_ML5P2_TREE GPU = CPU correctTest passed!
TREE NTT ML6P1 OUT TimeUsed: 5349.390000 ms, Throughput: 57427108.511438 ops/sec, Latency: 0.534939 ms.
Checking results!!!
correctTest is 30720 in 30720
NTT_ML6P1_TREE GPU = CPU correctTest passed!
TREE NTT ML2P2P2P1 OUT TimeUsed: 5072.910000 ms, Throughput: 60556958.432142 ops/sec, Latency: 0.507291 ms.
Checking results!!!
correctTest is 30720 in 30720
NTT_ML2P2P2P1_TREE GPU = CPU correctTest passed!
TREE NTT ML3P3P1 OUT TimeUsed: 5480.066000 ms, Throughput: 56057719.012873 ops/sec, Latency: 0.548007 ms.
Checking results!!!
correctTest is 30720 in 30720
```

EDFS 性能测试

```
case 3:
    printf("TEST NTT_EntireTREE_SPEED_TEST\n");
    NTT_1P6_TREE(g_gridSize, g_blockSize,g_packSize);
    NTT_2P5_TREE(g_gridSize, g_blockSize,g_packSize);
    NTT_3P4_TREE(g_gridSize, g_blockSize,g_packSize);
    NTT_4P3_TREE(g_gridSize, g_blockSize,g_packSize);
    NTT_5P2_TREE(g_gridSize, g_blockSize,g_packSize);
    NTT_6P1_TREE(g_gridSize, g_blockSize,g_packSize);
    NTT_2P2P2P1_TREE(g_gridSize, g_blockSize,g_packSize);
    NTT_3P3P1_TREE(g_gridSize, g_blockSize,g_packSize);
    break;
```

```
TEST NTT_TREE_SPEED_TEST
NTT TREE OUT TimeUsed: 4718.175000 ms, Throughput: 65107160.033974 ops/sec, Latency: 0.471838 ms.
Checking results!!!
correctTest is 30720 in 30720
NTT TREE GPU = CPU correctTest passed!
TREE NTT 1P6 OUT TimeUsed: 4495.621000 ms, Throughput: 68333162.426281 ops/sec, Latency: 0.449562 ms.
Checking results!!!
correctTest is 30720 in 30720
NTT_1P6_TREE GPU = CPU correctTest passed!
TREE NTT 2P5 OUT TimeUsed: 4862.420000 ms, Throughput: 63030200.145868 ops/sec, Latency: 0.486242 ms.
Checking results!!!
correctTest is 30720 in 30720
NTT_2P5_TREE GPU = CPU correctTest passed!
TREE NTT 3P4 OUT TimeUsed: 5061.000000 ms, Throughput: 60689777.249552 ops/sec, Latency: 0.506181 ms.
Checking results!!!
corectTest is 30720 in 30720
NTT_3P4_TREE GPU = CPU corectTest passed!
TREE NTT 4P3 OUT TimeUsed: 5036.226000 ms, Throughput: 60990026.878306 ops/sec, Latency: 0.503623 ms.
Checking results!!!
correctTest is 30720 in 30720
NTT_4P3_TREE GPU = CPU correctTest passed!
TREE NTT 5P2 OUT TimeUsed: 4660.667000 ms, Throughput: 66055041.137110 ops/sec, Latency: 0.463067 ms.
Checking results!!!
correctTest is 30720 in 30720
NTT_5P2_TREE GPU = CPU correctTest passed!
TREE NTT 6P1 OUT TimeUsed: 4811.285000 ms, Throughput: 63840000.256388 ops/sec, Latency: 0.481128 ms.
Checking results!!!
correctTest is 30720 in 30720
NTT_6P1_TREE GPU = CPU correctTest passed!
TREE NTT 2P2P2P1 OUT TimeUsed: 5445.555000 ms, Throughput: 56412982.600733 ops/sec, Latency: 0.544555 ms.
Checking results!!!
correctTest is 30720 in 30720
NTT_2P2P2P1_TREE GPU = CPU correctTest passed!
TREE NTT 3P3P1 OUT TimeUsed: 5180.033000 ms, Throughput: 59304618.508137 ops/sec, Latency: 0.518003 ms.
Checking results!!!
correctTest is 30720 in 30720
NTT_3P3P1_TREE GPU = CPU correctTest passed!
```

Kyber 性能测试

```
Select compare pke_keypair of host and device
GPU GEN OUT TimeUsed: 30289.423000 ms, Throughput: 1604610.680756 ops/sec, Latency: 19.144711 ms.
CPU GEN OUT Throughput 10561.784 op/sec, Latency: 0.094681 ms.
Select compare pke_enc of host and device
GPU ENC OUT TimeUsed: 34609.026000 ms, Throughput: 1775259.436657 ops/sec, Latency: 17.304513 ms.
CPU ENC OUT Throughput 9329.083 op/sec, Latency: 0.107192 ms.
Select compare pke_dec of host and device
GPU DEC OUT TimeUsed: 7685.427000 ms, Throughput: 7994350.866907 ops/sec, Latency: 3.842713 ms.
key exchange 1336382.826698 ops/sec
CPU ENC OUT Throughput 46524.520 op/sec, Latency: 0.021494 ms.
```

分层为 **2+2+2+1** 时，SDFS 性能最好

分层为 **1+6** 时，EDFS 性能最好

选取 **EDFS** 作为 NTT实现方案，测试 Kyber 性能

实验结果-三种 NTT 实现方案的性能

- 对 SLM、SDFS、EDFS都做了 1+6，2+5，3+4，4+3，5+2，6+1，2+2+2+1，3+3+1 的分层的数据吞吐测试

TEST：10000次取平均值

NTT基于GPU的原始吞吐 48318 kops/s

1. SLM于

SLM 最优实现 51924 kops/s 提升 7.5%

寄存器冗余测试

以 SLM 中性能最优的分组 2+2+2+1 为例

每组中数据加载量最少时，性能最优

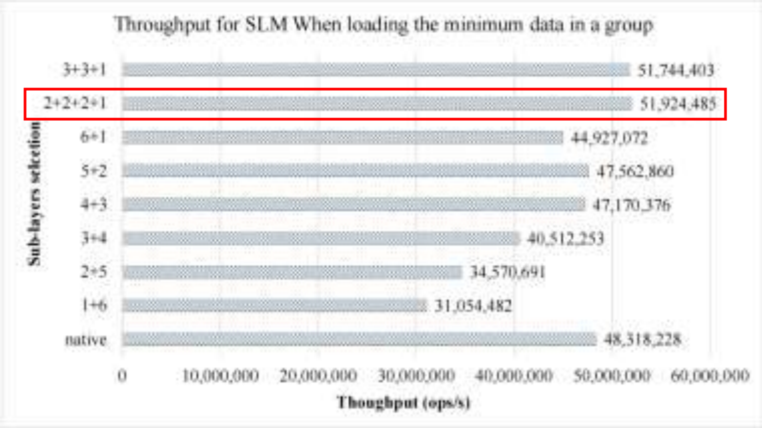
寄存器没有冗余

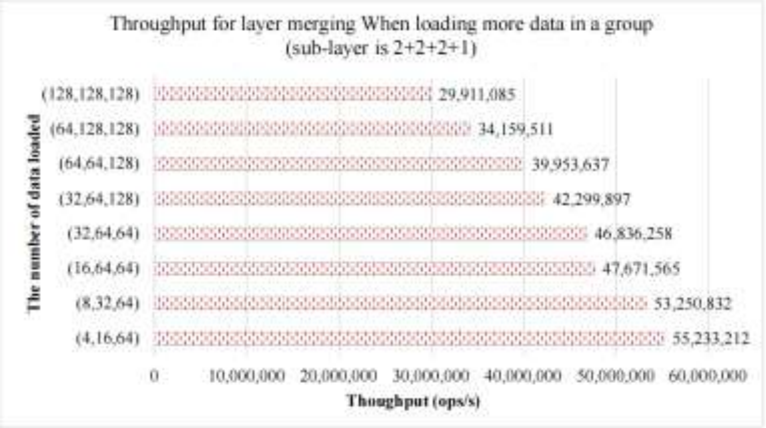
2. SDFS

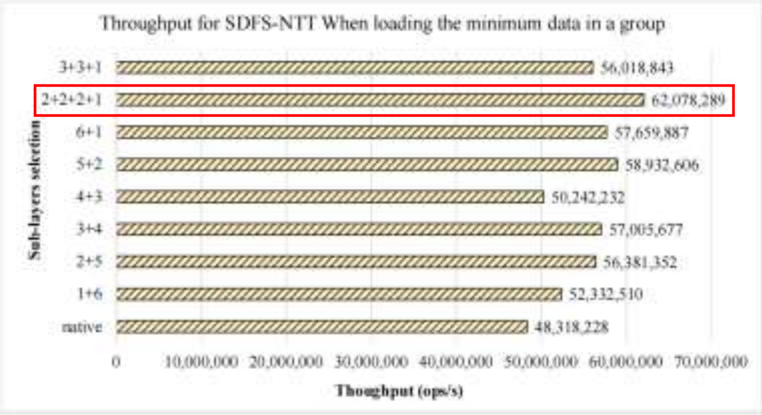
SDFS 最优实现 62078 kops/s 提升 28.5%

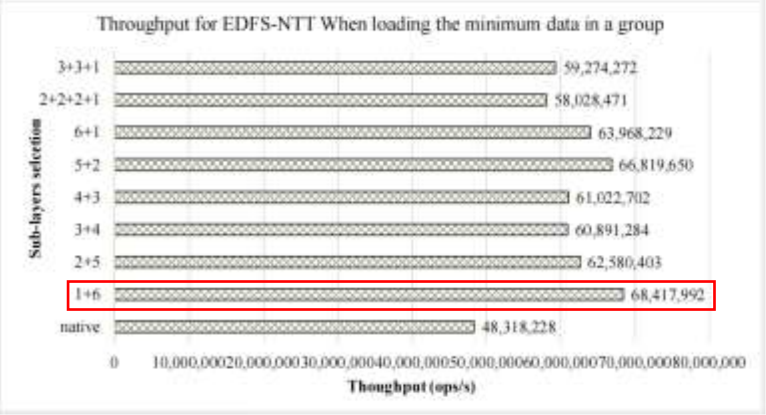
3. EDFS

EDFS 最优实现 68417 kops/s 提升 41.6%









实验性能 –HI-Kyber与其他工作对比

Nanjing University of Posts and Telecommunications

在相同的 GPU 平台上，HI-Kyber 的性能是 [4] 基于相同指令集 GPU 实现的 3.52 倍；是 [5] （目前最优性能实现）基于 AI 加速器 Tensor Core 实现的 1.78 倍

	平台	KeyGen/(ops/s)	Encryption/(ops/s)	Decryption/(ops/s)	Perf/(KX/s)
Sanal et al. [1]	Apple A12 2-cores Vortex at 2.490 GHz and 4-cores	26,157	26,774	27,360	13
Xing et al. [2]	Xilinx Artix7	17,182	14,728	11,600	7
C-Ref. [3]	Intel Core i7-4770K 3.5 GHz(Haswell), 4 Cores	11,390	10,101	8,826	5
AVX2-Ref. [3]	Intel Core i7-4770K 3.5 GHz(Haswell), 4 Cores	47,591	35,962	44,232	23
Gupta et al. [4]	NVIDIA Volta V100 GV100, 5120 CUDA cores	-	-	-	473
L. Wan et al. [5]	NVIDIA GeForce RTX 3080	1,250,000	1,298,701	2,380,952	820
Our work (EDFS-NTT)	NVIDIA Titan V Volta GV100, 5120 CUDA cores	1,639,949	1,763,898	7,885,157	**1,358**
	NVIDIA Tesla V100 Volta GV100,5120 CUDA cores	2,022,552	2,105,977	9,378,937	**1,664**
	NVIDIA GeForce RTX 3080	1,916,227	2,204,866	6,091,689	**1,458**
	NVIDIA Jetson AGX Xavier	160,130	187,832	539,450	**123**

同样适于
嵌入式 GPU

目录

CONTENT

实验总结 - 基于 GPU 的后量子密码 Kyber 加速技术

南京邮电大学 Nanjing University of Posts and Telecommunications

基于 GPU 的后量子密码 Kyber 加速技术设计

I. 构建 Kernel 融合架构 – 减少对全局内存的访问
II. NTT 算法设计 – GPU 寄存器的复用和减少对全局内存的访问
    I. SLM 提升 7.5%
    II. SDFS 提升 28.5%
    III. EDFS 提升 41.6%
III. 测试了**不同并行规模**下的 Kyber 最终性能吞吐以确定最佳并行参数（块数 = 80，线程数 = 384，寄存器数量 = 168）

最终密钥交换次数可达到 1358 kops/s

在相同的 GPU 平台上
- ✓ 是相同指令集架构 GPU 实现的 **3.52** 倍（**TPDS 2020** Gupta 等人的工作）
- ✓ 是基于AI加速器 GPU 实现（**目前最优实现**）的 **1.78** 倍（**ESORICS 2022** Wan 等人的工作）

Thank you for listening