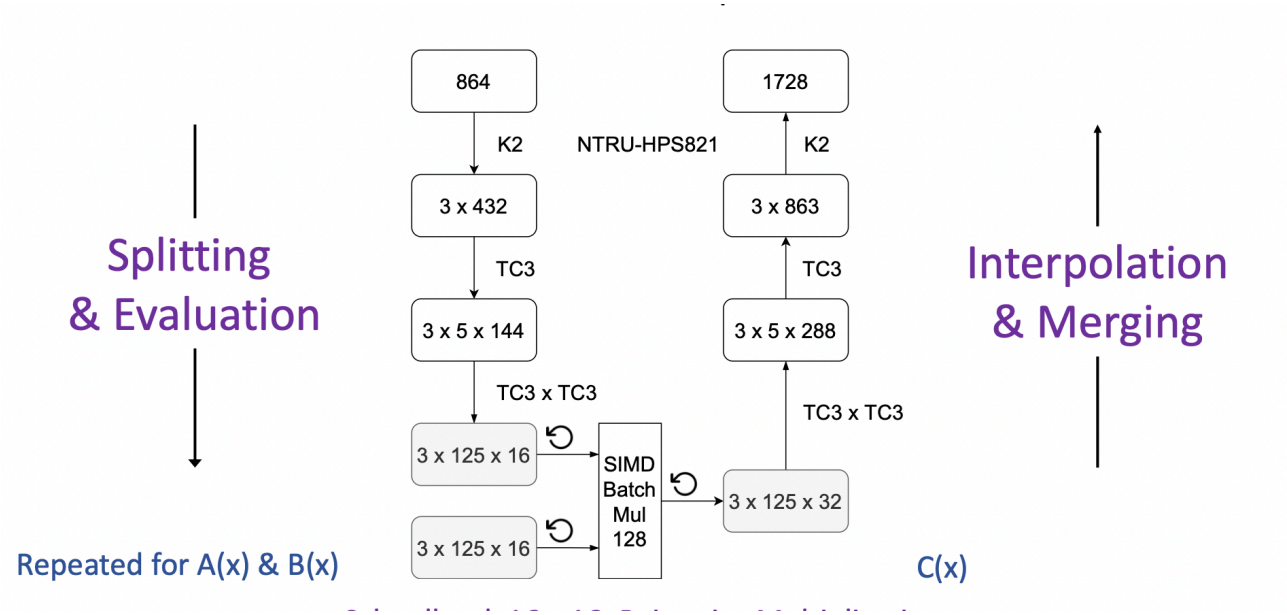
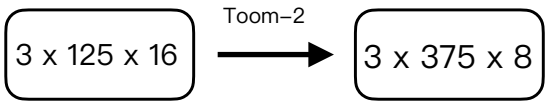
一、環境：
OS:GNU/Linux 11
Compiler:gcc 10.2.1

二、改進：



原先我們的改進是想改進上圖流程中的Splitting之分割方式，使其在最底層進行Multiplication的地方能完全在暫存器中進行運算（引用的code由於矩陣維度大而無法在暫存器中完成運算，因而會需要多次Load\Store記憶體），並且一同優化Toom-cook中Interpolation的部分。不過由於上述改進方法的code我們寫壞了（附在src/broken_code），所以我們最後僅僅在Evaluation多加一層Toom-2，使得底層進行Multiplication的地方能完全在暫存器中進行運算



Q：為何要將（方程式單位長度）壓到8?
A： Transpose：運算要8個暫存器(我們的code)、記錄要8個暫存器—>最多要24個暫存器
Multiplication：乘數要2*8=16個暫存器，結果要15個暫存器—>最多要31個暫存器
所以在Transpose-Multiplication-Transpose的過程中不需要進行多次Load\Store記憶體

三、結果：

```
pi@raspberrypi:~/Final/improved_SOTA $ make
cc -Wall -Wextra -Wpedantic -Wmissing-prototypes -Wredundant-decls -Wshadow -Wno-unused-result -mtune=native -mcpu=native -O3 -o test test.c hal-cortexa.c poly.c batch_
mul.c tran_mul_tran.c
cc -Wall -Wextra -Wpedantic -Wmissing-prototypes -Wredundant-decls -Wshadow -Wno-unused-result -mtune=native -mcpu=native -O3 -o speed speed.c hal-cortexa.c poly.c batc
h_mul.c tran_mul_tran.c
cc -Wall -Wextra -Wpedantic -Wmissing-prototypes -Wredundant-decls -Wshadow -Wno-unused-result -mtune=native -mcpu=native -O3 -o mod_inverse.s -S mod_inverse.c
pi@raspberrypi:~/Final/improved_SOTA $ ./test
poly_Rq_mul_small passed!
pi@raspberrypi:~/Final/improved_SOTA $ ./speed
poly_Rq_mul_small: 70647
pi@raspberrypi:~/Final/improved_SOTA $ ./speed
poly_Rq_mul_small: 68711
pi@raspberrypi:~/Final/improved_SOTA $ ./speed
poly_Rq_mul_small: 68062
pi@raspberrypi:~/Final/improved_SOTA $ ./speed
poly_Rq_mul_small: 70594
pi@raspberrypi:~/Final/improved_SOTA $ ./speed
poly_Rq_mul_small: 69694
pi@raspberrypi:~/Final/improved_SOTA $ ./test
poly_Rq_mul_small passed!
pi@raspberrypi:~/Final/improved_SOTA $ ./speed
poly_Rq_mul_small: 69623
```

如上圖，約落在68900 cycles(我們不太確定為何會是浮動的)。

四、工作分配：

張志謙(B08902128)：負責Splitting&Evaluation、Interpolating＆Merging的code和Presentation的報告
汪昱維(B08902060)：負責Transpose–Multiplication–Transpose的code和Report和Presentation的Slide

P.s.由於我們是各自處理不同任務最後再將其拼接，所以無法細分每一個改進分別減少了幾%的時間（不能各自直接跟原本的code對接）

五、參考：
Time-memory trade-off in Toom-Cook multiplication: an application to module-lattice based cryptography
Fast NEON-Based Multiplication for Lattice-Based NIST Post-quantum Cryptography Finalists
Source code:https://github.com/GMUCERG/PQC_NEON