PQC 2025 Spring Final Project : Big Integer Multiplication

嘗試

我一開始的嘗試是自己用 C 寫一個 NTT（模一個 `uint64_t` 等級的質數）並讓一個 limb 裝盡量多 bit，但這樣試出來的 cycle 數大約是 GMP 的十倍甚至二十倍以上（在 2^{16}-bit 乘法的情形）。

於是因為這個差距實在太大，我就先去找有沒有別人已經寫過的跑得很快的大數乘法。
我從 https://judge.yosupo.jp/problem/multiplication_of_big_integers 的排行榜前幾名找，因為前幾名都是用 C++ 寫的，所以我設定了一下 `Makefile` 讓我可以測試他們的速度，另外也加了一個隨機產生數字來測試乘法 consistency 的檔案 `test.c`。

第一名和第三名（`andyli`、`Yuezheng_Ling_fans`）都是用 complex number FFT 做的，而且他的程式碼沒有使用 x64 的 `immintrin.h` 裡面的函式，可以直接在 Raspberry Pi 上面執行，一開始我想說連這都無法贏 `mpz_mul` 的話，那可能我自己寫完全贏不了，但只要把 N 拉大就可以贏過 `mpz_mul` 了，而且 N 越大贏越多（但超過一定程度之後 consistency 會開始壞掉）。意識到 GMP 的小數字乘法真的很快，設置不同切點是有道理的。

第二名（`grishared`）使用的是三個質數的 NTT 以及中國剩餘定理，但他的程式碼裡面有使用 x64 的 `immintrin.h` 的 AVX2 指令，所以並不能在 Raspberry Pi 4 上面跑，不過在我的 Intel CPU 筆電上是可以在 2^{19}-bit 左右的乘法就贏過 `mpz_mul` 的。

因為課堂上教的是 NTT，而且 FFT 的誤差比較難分析，我打算就模仿第二名的架構但改成 ARM NEON Intrinsics。因為 x64 的乘法邏輯與 ARM NEON 邏輯差很多，我並不是直接從這份程式碼開始改，而是先從一個比較 portable 的 NTT atcoder/convolution開始改，逐步加上 Montgomery、vectorize、Incomplete NTT 等優化。

最後決定的演算法

整體架構

選定一個 limb 為 32-bit（$R = 2^{32}$）。
要相乘兩個 N-limb 的數字，先計算不做進位的捲積，而捲積的每一項 $c_k = \sum_{i+j=k} a_i b_j < N \cdot R^2$。如果分別模 p_1, p_2, p_3 三個質數計算長度 $2N$ 的捲積 $\langle c_k \mod p_1 \rangle, \langle c_k \mod p_2 \rangle, \langle c_k \mod p_3 \rangle$，那麼根據中國剩餘定理，只要 $p_1 \cdot p_2 \cdot p_3 \geq N \cdot R^2$ 就可以唯一還原出 c_k。
最後再用 Garner's algorithm 一邊還原一邊計算進位得到正確的大數乘積。

> Garner's algorithm: 若 p_1, p_2, p_3 為相異質數且
>
> $$\begin{cases} x \equiv r_1 \pmod{p_1} \\ x \equiv r_2 \pmod{p_2} \\ x \equiv r_3 \pmod{p_3} \end{cases}$$
>
> 則
>
> $$\begin{cases} x_1 \equiv r_1 \pmod{p_1} \\ x_2 \equiv (r_2 - x_1)p_1^{-1} \pmod{p_2} \\ x_3 \equiv (r_3 - x_1)p_1^{-1} - x_2)p_2^{-1} \pmod{p_3} \\ x = x_1 + x_2 p_1 + x_3 p_1 p_2 \end{cases}$$
>
> 如果 $0 \leq x < p_1 p_2 p_3$ 且我們所有 mod 操作都化約到 $[0, p_i)$ 內，則 x 算出來的是精確值。

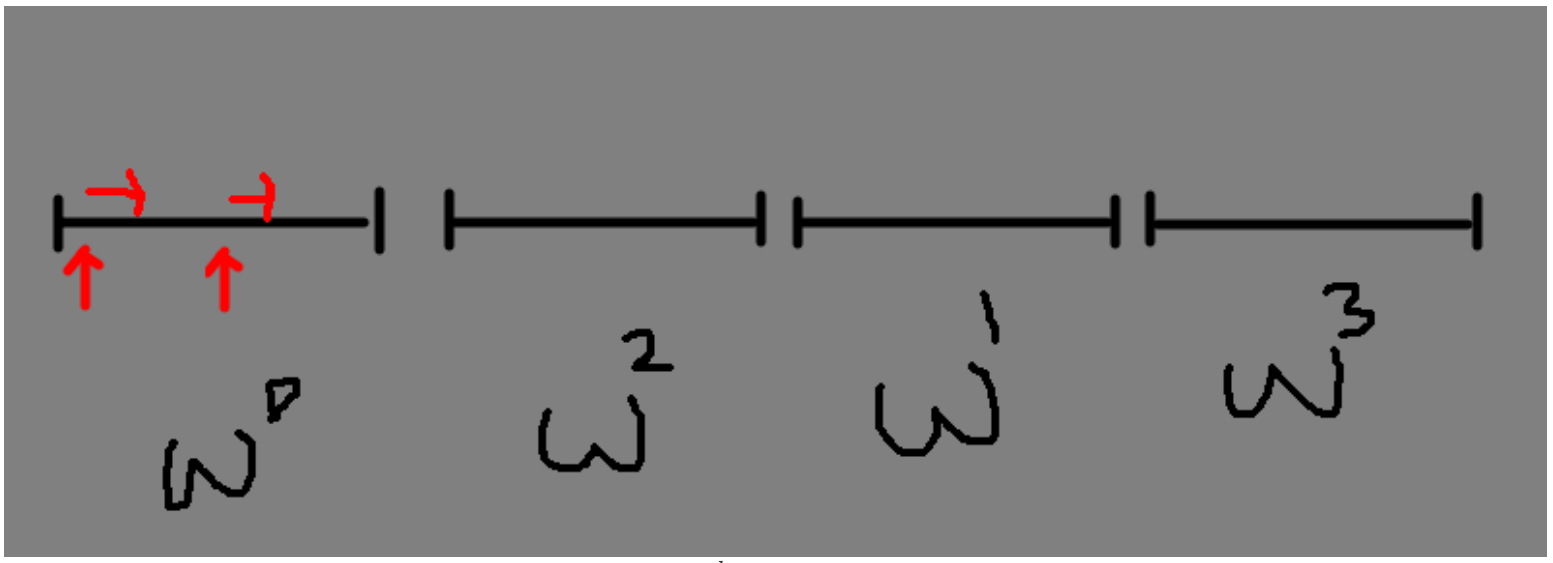
而在此架構中，最關鍵的部份就是計算模質數 p_i 的 NTT 了。我選定的三個質數大約都是 30-bit，且都是 NTT-friendly 的質數（可以支援到長度 2^{23} 的捲積，也就是說在此框架下可以計算到 2^{27}-bit 的大數字相乘）

```
static constexpr uint32_t M1 = 880803841;
static constexpr uint32_t M2 = 897581057;
static constexpr uint32_t M3 = 998244353;
static_assert(countr_zero(M1 - 1) == 23 && countr_zero(M2 - 1) == 23 &&
              countr_zero(M3 - 1) == 23);
```

NTT 捲積的優化

在前述提到的 atcoder/convolution 當中就有以下兩點優化

- 採用不預先 bitrev permute 的寫法
  random access 陣列是相當慢的記憶體操作。在此實做當中不會做 bitrev permute 而盡量讓我們對記憶體 sequential access。在這個寫法中，每一個連續的區段在 butterfly 要乘的因子都是一樣的，但從一塊進到下一塊時這個因子的次方是 bitrev 的順序。在此採用的方法是預處理一些關鍵的次方讓我們可以簡單的在途中一邊算出需要的 ω。



如圖，每次到新的一整塊，butterfly 要乘的因子會乘上一個 ω^k。只需要預處理 $\log(N)$ 種 ω 的次方就好，因為 k 一定是二進位 `00...01*** - 11...10***`

- 手動展開兩層 butterfly 變成 radix-4 的變換
  寫成線性變換的話就是

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \mapsto \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \\ 1 & -i & -1 & i \end{bmatrix} \begin{bmatrix} a_0 \\ a_1\omega \\ a_2\omega^2 \\ a_3\omega^3 \end{bmatrix}$$

雖然寫成矩陣形式，但程式碼中實際乘這個矩陣時做的乘法、加減法次數和做兩次 radix-2 幾乎一樣，不過應該對 cache 以及指令重排有幫助。

接著我以此 library 為基礎加上以下幾個優化：

Montgomery reduction

事先把 a, b 以及所有 NTT 要乘的那些單位根變換到 montgomery domain，並把幾乎所有乘法取模運算都改成在 montgomery domain 下的 `REDC` 操作。
變換到 montgomery domain 只需要做 $O(N)$ 次，且都是簡單的 sequential access（老師表示這裡也是花了多餘的時間）

vectorize

使用向量化的 intrinsic 指令，一次做 `uint32x4_t` 的 `REDC`、加減法取模。radix-4 和 radix-2 都可以 vectorize，但是最後兩層會因為每一塊的大小太小無法一樣套用 vectorize。
我採用的是 unsigned reduction $T \mapsto (T + (-M^{-1} \cdot T \mod R) \cdot M)/R$。一個 register 只有 128-bit，所以要做 64-bit 乘法得分成 low 和 high，vectorize 過的 `REDC` 大概寫得如下面這樣：

```
76  static uint32x4_t redc_64x4(uint64x2_t T_lo, uint64x2_t T_hi) {
 1    // Extract low 32 bits from T for multiplication with xinv
 2    uint32x4_t T_low = vcombine_u32(vmovn_u64(T_lo), vmovn_u64(T_hi));
 3
 4    // Compute m = T_low * xinv (mod 2^32)
 5    uint32x4_t m = vmulq_u32(T_low, vdupq_n_u32(xinv));
 6
 7    // Multiply m with mod
 8    // Add m * mod to T
 9    T_lo = vmlal_u32(T_lo, vdup_n_u32(mod), vget_low_u32(m));
10    T_hi = vmlal_u32(T_hi, vdup_n_u32(mod), vget_high_u32(m));
11
12    // Shift right by W (32 bits) - take high 32 bits
13    uint32x4_t result =
14        vcombine_u32(vshrn_n_u64(T_lo, 32), vshrn_n_u64(T_hi, 32));
15
16    // Conditional subtraction: if result >= mod then result -= mod
17    result = vminq_u32(result, vsubq_u32(result, vdupq_n_u32(mod)));
18
19    return result;
20  }
```

```
98  static uint32x4_t redc_32x4(uint32x4_t a, uint32x4_t b) {
 1    // Multiply a * b using widening multiplication
 2    uint32x2_t a_lo = vget_low_u32(a);
 3    uint32x2_t a_hi = vget_high_u32(a);
 4    uint32x2_t b_lo = vget_low_u32(b);
 5    uint32x2_t b_hi = vget_high_u32(b);
 6
 7    uint64x2_t ab_lo = vmull_u32(a_lo, b_lo);
 8    uint64x2_t ab_hi = vmull_u32(a_hi, b_hi);
 9
10    return redc_64x4(ab_lo, ab_hi);
11  }
12
13  static uint32x4_t from_32x4(uint32x4_t T) {
14    return redc_32x4(T, vdupq_n_u32(R2));
15  }
16
17  static uint32x4_t get_32x4(uint32x4_t T) {
18    return redc_32x4(T, vdupq_n_u32(1));
19  }
20
```

vectorize 的模下加減法則比較單純，就是加/減過後按情況把數字化約回 $[0, M)$ 當中。

Incomplete NTT

前面有提到在最後幾層中，每一塊的大小太小而不能 vectorize，因此就不分解最後幾層，而是直接 schoolbook multiplication 算 a, b 每一塊在 $\mathbb{Z}_p[x]/(x^{2^l} - \omega)$ 逐塊相乘的捲積

逐塊相乘的優化

以模 $(x^4 - \omega)$ 舉例來說，比較 naive 的寫法可以寫成下面這樣

```
u32 wb_b[8] = {};
for (int y = 0; y < 8; y++) {
  if (y < 4) wb_b[y] = redc(b[y], w);
  else wb_b[y] = b[y - 4];
}
u32 res[4] = {};
for (int x = 0; x < 4; x++)
  for (int z = 0; z < 4; z++) {
    res[z] = add(res[z], redc(a[x], wb_b[z + 4 - x]));
  }
```

但我們其實可以把 `a[x]` 和 `wb_b[z + 4 - x]` 先不 reduce 而直接加起來，最後再一次 reduce。這樣可以得到

```
u32 wb_b[8] = {};
for (int y = 0; y < 8; y++) {
  if (y < 4) wb_b[y] = redc(b[y], w);
  else wb_b[y] = b[y - 4];
}
u64 res_64[4] = {};
for (int x = 0; x < 4; x++)
  for (int z = 0; z < 4; z++) {
    res_64[z] += 1ULL * a[x] * wb_b[z + 4 - x];
  }
u32 res[4] = {};
for (int x = 0; x < 4; x++)
  res[x] = redc(res_64[x]);
```

上圖中，`res_64` 的值域大小是 $4 \cdot M^2$，小於 2^{64}（我們選的 $M < 2^{30}$），所以可以在 `uint64_t` 裡面直接加，最後再一次 `REDC`（Montgomery 只要求輸入要小於 $R \cdot M$ 就會是正確的）
此形式的乘法很容易 vectorize，且實際執行非常之快速。實際上，我選擇的是不分解最後三層，改計算模 $(x^8 - \omega)$ 的逐塊相乘，需要在最後化約一次才能保證小於 $R \cdot M$。

如此一來 NTT 的「所有」部份都被 vectorize 了。（這很重要，因為短板理論）
Incomplete NTT 以及此優化的想法來源是前述提到過的排行榜第二名的程式碼。

mulhi 優化

在我測試時發現主要瓶頸是 NTT 的正向變換（正向變換做的次數是逆向的兩倍，而逐塊相乘所花的時間也不多）
因為每一層的變換都有大量的跟某個常數做乘法取模的操作，所以我想到按照課堂上的簡報的兩次 mulhi 一次 mullo 的作法：`REDC(a * b) = mulhi(a, b) - mulhi(M, mullo(a, b'))` 其中 $b' = (bM^{-1} \mod R)$。在 ARM NEON 中最接近的 mulhi 指令是 `vqrdmulhq_s32`，除了被限制要用 signed 以外還需要做一個額外的除以 2，程式碼大概如下（同樣是一次對 `uint32x4_t` `REDC`）：

```cpp
struct MulConstContext {
  uint32_t b, b_prime;
  MulConstContext(u32 t_b) : b(t_b), b_prime(-t_b * xinv) {}
};

static uint32x4_t redc_32x4_by_context(uint32x4_t a,
                                       const MulConstContext &ctx) {
  // use signed reduction
  int32x4_t a_signed = vreinterpretq_s32_u32(a);
  uint32x4_t low = vmulq_u32(a, vdupq_n_u32(ctx.b_prime));
  int32x4_t low_signed = vreinterpretq_s32_u32(low);
  int32x4_t diff = vsubq_s32(vqrdmulhq_s32(a_signed, vdupq_n_s32(ctx.b)),
                             vqrdmulhq_s32(low_signed, vdupq_n_s32(mod)));
  uint32x4_t result = (uint32x4_t)vshrq_n_s32(diff, 1);
  return vminq_u32(result, vaddq_u32(result, vdupq_n_u32(mod)));
}
```

比較優化前後的指令的話，

- 直接 `REDC`：`vmull_u32 * 2 + vmulq_u32 + vmlal_u32 * 2 + vshrn_n_u64 * 2`
- 乘常數 `REDC`：預處理需要一個 `uint32_t` 乘法，之後只需要 `vmulq_u32 + vqrdmulhq_s32 * 2 + vshrq_n_s32`

template & constexpr & Ofast

編譯參數也是一個很重要的部份，從不加任何優化到 `-O2`、`-O3`、`-Ofast` 都有不同程度的 performance 提昇。

關於 template 與 constexpr，我一開始 Montgomery 的寫法是把 mod 以 constructor 傳給 Montgomery，而 Montgomery 常數都當作一個 member，這樣似乎編譯器沒辦法每次都知道這是一個編譯時就知道的常數。後來我改成 用 template 傳進 mod 以及使用 static constexpr 存 Montgomery 常數之後有得到一段明顯的 performance 提昇。

結果

以下的 N 都是指 bit 數，即把兩個 N-bit 的數字相乘得到 $2N$-bit 的數字。

在 N 拉到夠大的時候，經過以上優化的大數乘法就越來越打得過 `mpz_mul`。

選定 $N = 2^{21}$，且 `NWARMUP`, `NITERATIONS`, `NTESTS` 分別是 10, 60, 100 時的 benchmark 執行結果如下（需要跑八分鐘多）：

```
❯ make CYCLES=PERF && ./bench
g++ -Wall -Wextra -Werror=unused-result -Wpedantic -Werror -Wshadow -Wpointer-arith -Wredundant-decls -Wno-long-long -Wno-unknown-pragmas -Wno-u
nused-command-line-argument -Wconversion -fomit-frame-pointer -fno-stack-protector -std=c++2a -pedantic -MMD -Ofast -Wfatal-errors -march=native
  -c bigint.cpp -o bigint.o
g++ -Wall -Wextra -Werror=unused-result -Wpedantic -Werror -Wshadow -Wpointer-arith -Wredundant-decls -Wno-long-long -Wno-unknown-pragmas -Wno-u
nused-command-line-argument -Wconversion -fomit-frame-pointer -fno-stack-protector -std=c++2a -pedantic -MMD -Ofast -Wfatal-errors -march=native
  test.o bigint.o -o test -lgmp
g++ -Wall -Wextra -Werror=unused-result -Wpedantic -Werror -Wshadow -Wpointer-arith -Wredundant-decls -Wno-long-long -Wno-unknown-pragmas -Wno-u
nused-command-line-argument -Wconversion -fomit-frame-pointer -fno-stack-protector -std=c++2a -pedantic -MMD -Ofast -Wfatal-errors -march=native
  bench.o hal/hal.o bigint.o -o bench -lgmp
BENCH_SIZE = (1 << 21) = 2097152
       mpz cycles = 67859455

           percentile         1        10        20        30        40        50        60        70        80        90        99
       mpz percentiles:  67617877  67712071  67733428  67744318  67767663  67859455  67889410  67940435  68143591  68180010  70399611
       mul cycles = 59893464

           percentile         1        10        20        30        40        50        60        70        80        90        99
       mul percentiles:  59433914  59475364  59612336  59722877  59850306  59893464  59929435  59957249  59981744  60099457  60551357
 ~/aarch64-bench  main ↑29 !1 ?5                                                                    8m 35s  omelet@raspberrypi  00:27:11
❯
 [ 0 ]  |  vim  1:zsh*                                                                              [ omelet@raspberrypi ] 2025.06.09 00:28
```

選定 $N = 2^{22}$，且 `NWARMUP`, `NITERATIONS`, `NTESTS` 分別是 10, 60, 100 時的 benchmark 執行結果如下（需要跑約二十分鐘）：

```
❯ make CYCLES=PERF && ./bench
gcc -Wall -Wextra -Werror=unused-result -Wpedantic -Werror -Wmissing-prototypes -Wshadow -Wpointer-arith -Wredundant-decls -Wno-long-long -Wno-u
nknown-pragmas -Wno-unused-command-line-argument -fomit-frame-pointer -fno-stack-protector -std=c99 -pedantic -MMD -Ihal -Ofast -Wfatal-errors -
march=native  -DPERF_CYCLES -c test.c -o test.o
g++ -Wall -Wextra -Werror=unused-result -Wpedantic -Werror -Wshadow -Wpointer-arith -Wredundant-decls -Wno-long-long -Wno-unknown-pragmas -Wno-u
nused-command-line-argument -Wconversion -fomit-frame-pointer -fno-stack-protector -std=c++2a -pedantic -MMD -Ofast -Wfatal-errors -march=native
  -c bigint.cpp -o bigint.o
g++ -Wall -Wextra -Werror=unused-result -Wpedantic -Werror -Wshadow -Wpointer-arith -Wredundant-decls -Wno-long-long -Wno-unknown-pragmas -Wno-u
nused-command-line-argument -Wconversion -fomit-frame-pointer -fno-stack-protector -std=c++2a -pedantic -MMD -Ofast -Wfatal-errors -march=native
  test.o bigint.o -o test -lgmp
gcc -Wall -Wextra -Werror=unused-result -Wpedantic -Werror -Wmissing-prototypes -Wshadow -Wpointer-arith -Wredundant-decls -Wno-long-long -Wno-u
nknown-pragmas -Wno-unused-command-line-argument -fomit-frame-pointer -fno-stack-protector -std=c99 -pedantic -MMD -Ihal -Ofast -Wfatal-errors -
march=native  -DPERF_CYCLES -c bench.c -o bench.o
g++ -Wall -Wextra -Werror=unused-result -Wpedantic -Werror -Wshadow -Wpointer-arith -Wredundant-decls -Wno-long-long -Wno-unknown-pragmas -Wno-u
nused-command-line-argument -Wconversion -fomit-frame-pointer -fno-stack-protector -std=c++2a -pedantic -MMD -Ofast -Wfatal-errors -march=native
  bench.o hal/hal.o bigint.o -o bench -lgmp
BENCH_SIZE = (1 << 22) = 4194304
       mpz cycles = 165457687

           percentile         1        10        20        30        40        50        60        70        80        90        99
       mpz percentiles: 163844181 164634573 164686864 164746251 165134670 165457687 165475917 165521593 165559897 165600993 165669541
       mul cycles = 136065190

           percentile         1        10        20        30        40        50        60        70        80        90        99
       mul percentiles: 135873969 135961320 135983364 136015858 136035975 136065190 136083211 136105614 136147365 136191868 136319601
 ~/aarch64-bench  main ↑29 !2 ?5                                                                    20m 2s  omelet@raspberrypi
```

選定 $N = 2^{25}$，且 `NWARMUP`, `NITERATIONS`, `NTESTS` 分別是 1, 6, 10 時的 benchmark 執行結果如下（需要跑約五分鐘）：

```
❯ make CYCLES=PERF && ./bench
g++ -Wall -Wextra -Werror=unused-result -Wpedantic -Werror -Wshadow -Wpointer-arith -Wredundant-decls -Wno-long-lon
g -Wno-unknown-pragmas -Wno-unused-command-line-argument -Wconversion -fomit-frame-pointer -fno-stack-protector -st
d=c++2a -pedantic -MMD -Ofast -Wfatal-errors -march=native  -c bigint.cpp -o bigint.o
g++ -Wall -Wextra -Werror=unused-result -Wpedantic -Werror -Wshadow -Wpointer-arith -Wredundant-decls -Wno-long-lon
g -Wno-unknown-pragmas -Wno-unused-command-line-argument -Wconversion -fomit-frame-pointer -fno-stack-protector -st
d=c++2a -pedantic -MMD -Ofast -Wfatal-errors -march=native  test.o bigint.o -o test -lgmp
g++ -Wall -Wextra -Werror=unused-result -Wpedantic -Werror -Wshadow -Wpointer-arith -Wredundant-decls -Wno-long-lon
g -Wno-unknown-pragmas -Wno-unused-command-line-argument -Wconversion -fomit-frame-pointer -fno-stack-protector -st
d=c++2a -pedantic -MMD -Ofast -Wfatal-errors -march=native  bench.o hal/hal.o bigint.o -o bench -lgmp
BENCH_SIZE = (1 << 25) = 33554432
       mpz cycles = 2752603169

           percentile         1        10        20        30        40        50        60        70        80
    90        99
       mpz percentiles:2745386060274733094327490859132749138279275075391327526031692753059234275585218027564059232760775084260775084
       mul cycles = 1533934764

           percentile         1        10        20        30        40        50        60        70        80
    90        99
       mul percentiles:1515233920151669604615170990411522560251152936412615339347641542273538154709381415507806791562388714623887141562388714
```

$N = 2^{25}$ 時所花的時間已經是 `mpz_mul` 的大約一半，可以算是滿意的結果。

可能的下一步

- 透過調整 NTT 的係數把預先對 a, b 轉成 Montgomery domain 的操作給去掉
- 目前的 code 中，大部份的時候都有把數字化約到 $[0, M)$，這需要花費不少的 conditional add/sub 或 min + add/sub，也許可以在某些地方放寬到 $2M$ 或 $4M$ 之類的以減少這些化約
- 把 `NTT` 的 method 都改成 static，或者寫一些 template code 讓編譯器預先知道 transform 的一塊大小或者其他可能在編譯時先處理的資訊