并行计算实验三报告
——快速傅里叶变换的并行实现

PB20111701 叶升宇 **PB20111689 蓝俊玮**

说明

本此实验由 PB20111701 叶升宇完成。

实验题目

快速傅里叶变换的并行实现。

实验环境

- 操作系统：Windows 11 22H2
- IDE 及工具链：CLion + VS2019 + Ninja
- CUDA：12.1 其他参数请参考实验一报告
- 处理器： Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59 GHz

原理介绍

串行 FFT

说明

本次实验考虑一维的 FFT，因为事实上傅里叶变换具有分离性，二维的傅里叶变换可以通过两次一维傅里叶变换得到，其它高维同理。

记 $A(x) = \sum_{j=0}^{n-1} a_j x^j$，定义 $y_k = A(w_n^k) = \sum_{j=0}^{n-1} a_j w_n^{kj}$，其中 $w_n^k = e^{\frac{2\pi k i}{n}}$ 是第 k 个单位负数根。向量 $y = (y_0, y_1, \ldots, y_{n-1})$ 就是系数向量 $a = a(a_0, a_1, \ldots, a_{n-1})$ 的离散傅里叶变换（DFT）。

而 FFT 则利用分治策略，利用 $A(x)$ 中偶数下标的系数与奇数下标的系数：

$$A^{[0]}(x) = a_0 + a_2 x + a_4 x^2 + \ldots + a_{n-2} x^{\frac{n}{2}-1}$$

$$A^{[1]}(x) = a_1 + a_3 x + a_5 x^2 + \ldots + a_{n-1} x^{\frac{n}{2}-1}$$

于是有

$$A(x) = A^{[0]}(x^2) + x A^{[1]}(x^2)$$

可以 **递归** 地求解子问题然后合并：

$$y_k = y_k^{[0]} + w_n^k y_k^{[1]} = A^{[0]}(w_n^{2k}) + w_n^k A^{[1]}(w_n^{2k}) = A(w_n^k)$$

$$y_{k+\frac{n}{2}} = y_k^{[0]} - w_n^k y_k^{[1]} = A^{[0]}(w_n^{2k+n}) + w_n^{k+\frac{n}{2}} A^{[1]}(w_n^{2k+n}) = A\left(w_n^{k+\frac{n}{2}}\right)$$

通常把因子 w_n^k 称为旋转因子，其中时间复杂度为 $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) = \Theta(n \lg n)$。

事实上上述递归的 FFT 并不够高效，下面再将其改为迭代实现。

首先注意到计算的时候存在公共表达式：旋转因子 w_n^k 乘以 $y_k^{[1]}$ 的值可以存入 t ，将 $y_k^{[0]} \pm t$ 的操作称为蝴蝶操作，如下图所示：

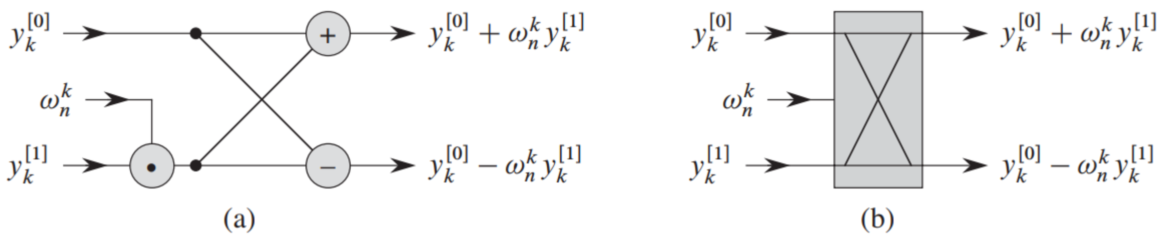


Figure 30.3 A butterfly operation. **(a)** The two input values enter from the left, the twiddle factor ω_n^k is multiplied by $y_k^{[1]}$, and the sum and difference are output on the right. **(b)** A simplified drawing of a butterfly operation. We will use this representation in a parallel FFT circuit.

下图是递归调用 FFT 产生的向量树:

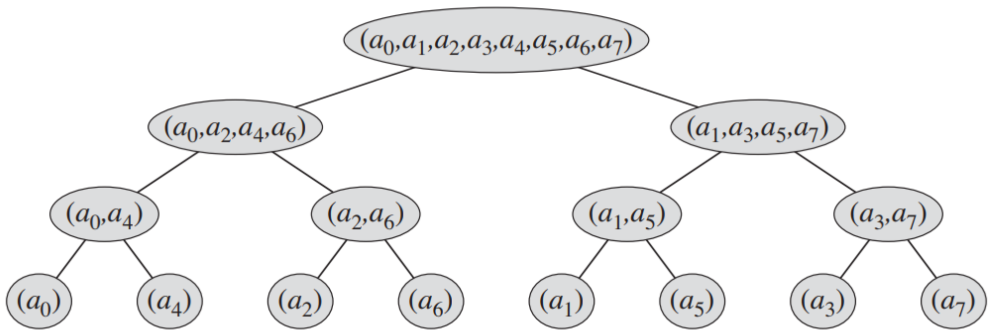


Figure 30.4 The tree of input vectors to the recursive calls of the RECURSIVE-FFT procedure. The initial invocation is for $n = 8$.

注意到，我们只需要将初始元素出现的顺序进行安排，就可以自底向上迭代进行运算。这里安排的顺序称为位逆序置换，例如 $6 = 110$ ，将二进制位反过来即为 011 ，即 6 应该调到位置 3。

修改后的迭代 FFT 时间复杂度为（展开的递归树有 $\lg n$ 层）

$$L(n) = \sum_{s=1}^{\lg n} \frac{n}{2^s} . 2^{s-1} = \sum_{s=1}^{lgn} = \Theta(n \lg n)$$

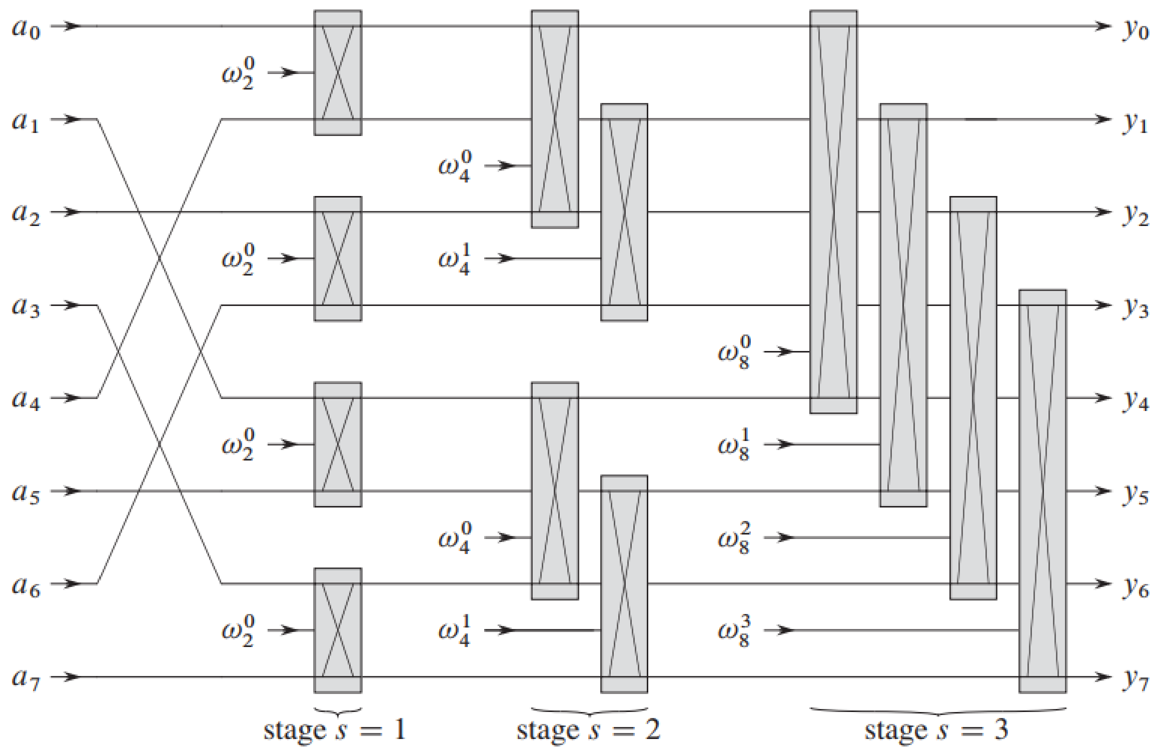
并行 FFT

基于迭代 FFT，可以将其改造为并行 FFT 算法。

首先对输入进行位逆序置换，然后将电路分为 $\lg n$ 级，每一级由 $\frac{n}{2}$ 个并行执行的蝴蝶操作组成。注意到这里同一级的每一个蝴蝶操作都是 **互不相关** 的，以两条不同线路上的数值和一个旋转因子作为输入，并且产生两条线路上的数值作为输出，因此才有并行的可行性，具体到 cuda 上，每一个 thread 执行同一级上的多个蝴蝶操作，每一级完成后都需要 **同步** 。

最后，分析一下时间复杂度，每一级只需要 $\Theta(1)$ 时间即可完成，深度为 $\Theta(\lg n)$ ，因此时间复杂度为 $\Theta(\lg n)$。（前提是有 $\Theta(n)$ 个线程可以保证每一级的并行，同时实际上总的还是做了 $\Theta(n \lg n)$ 个蝴蝶操作）。

一个计算 FFT 的并行电路可参照下图。



代码实现

首先是位逆转操作

```
int bitReverse(int x, int bits) {
    int res = 0;
    do {
        int bit = x % 2;
        --bits;
        res += bit << bits;
    } while (x /= 2);
    return res;
}
```

然后是由于 C++ 的 complex 类并没有支持 device 上的操作，所以需要自定义 **复数类** ，考虑到代码的可复用，实现了一个可以在 device 上操作的带模板(template) 的复数类，具体参照源代码，不在此处赘述。

接着就是核心的 FFT 的实现。

```
__global__ void FFT(Complex signal[], Complex transform[], int nbits) {
    int tid = threadIdx.x + blockDim.x * blockIdx.x;
    int n = 1 << nbits;
    for (int i = 2; i < 2 * n; i *= 2) {
        if (tid % i == 0) {
          for (int j = 0; j < i / 2; ++j) {
                butterfly(signal[bitReverse(tid + j, nbits)],
                signal[bitReverse(tid + j + i / 2, nbits)], Complex::W(i, j));
            }
        }
        __syncthreads();
```

```
    }
    transform[tid] = signal[bitReverse(tid, nbits)];
}
```

说明

不难发现这初始版本其实并没有按照原理介绍中的实现，因为一开始是先追求正确性，写出一个能运行的版本，后面才进行一步步改进。

代码优化

事实上，初始版本的实现有很多地方可以优化，以下分为三点进行描述。

优化一：结构调整

- 首先，注意到在最内层 `for` 循环里，每个蝴蝶操作需要先进行位逆序置换，这其实并没有必要，根据原理，只需要初始进行 **一次** 置换，后续可以都不用置换；
- 其次，假设有 n 个线程的情况下，上述实现其实是一个类似规约的操作，并没有充分利用每个线程，在第 k 级，只有 $\frac{n}{2^k}$ 个线程在运算，可以调整结构，使得每个线程都参与运算。

```
int n = 1 << nbits;
uint tid = threadIdx.x + blockDim.x * blockIdx.x;
for (uint i = tid; i < SIGNAL_SIZE; i += STRIDE) {
    transform[cubitReverse(i, nibts)] = signal[i];
}
__syncthreads();
for (int i = 2; i <= n; i *= 2) {
    int k = tid % (i / 2 - 1);
    int index = (2 * tid / i) * i + tid % (i / 2 - 1)
    Complex<T> factor = Complex<T>::W(m, k);
    butterfly(transform[index], transform[index + offset], factor);
    __syncthreads();
}
```

如上调整后，结构与并行 FFT 电路保持一致，每一级每个线程都能参与运算，不浪费算力。

说明

事实上，这个版本有两个问题

1. 由于为了让每个线程都参与运算，使得程序语义发生了部分变化，在线程数大于等于 $\frac{n}{2}$ 的情况下使能够正常得到结果的，但是如果线程数不足，上述程序语义是每个线程每一级只执行一次蝴蝶操作，会导致后面的变换没有进行！
2. 同样是由于线程不足，这里指的是一个 block 内 thread 至多只有 1024 个，因为每一级都要同步，而`__syncthreads();`只能在 block 内同步，没有跨 block 同步，因此输出结果会不正确！具体修复见下文。

优化二：去除同步

在这一步，将修复刚才提到的两个问题。首先是同步，实际上，同步本身会导致程序的性能下降，也表明刚才的并行粒度较粗，可以再细分，这里考虑将每已级的操作细分为一个 kernel ，进行并行，而这之间串行，重新组织，去除同步后，程序的逻辑结构更加清晰。

而刚才提到的第一个问题相对简单，只需要计算步长，将原先一个线程执行一次改为迭代执行若干次，本身同一级内不会互相干扰，因此保证了正确性。

说明

实际上去掉同步还有一个原因是 block 间的同步非常难实现，可行的方案有设置累加锁，但是这会导致性能下降严重。也曾尝试 cuda 的 `cooperative-groups` 库，但是针对我们实现的核函数，也不支持 grid 内所有线程的同步，所以最后不得不放弃这一想法。

考虑篇幅，修改后的代码与下一段合并展示。

优化三：指令优化

这一部分主要是利用 **位运算** ，对代码指令进行优化。注意到在之前的视线中，有不少乘除以及加减运算，实际啥那，除了蝴蝶操作之外，大部分都可以利用位运算优化。具体如下

```
__device__ uint cubitReverse(uint x, int n) {
    uint res = 0;
    while (n >>= 1) {
        res <<= 1;
        res |= (x & 1);
        x >>= 1;
    }
    return res;
}

__global__ void init(Complex<T> signal[], Complex<T> transform[], int n) {
    uint tid = threadIdx.x + blockDim.x * blockIdx.x;
    for (uint i = tid; i < SIGNAL_SIZE; i += STRIDE)
        transform[cubitReverse(i, n)] = signal[i];
}

__global__ void step(Complex<T> transform[], int m, int mask) {
    uint tid = threadIdx.x + blockDim.x * blockIdx.x, offset = m >> 1;
    for (uint i = tid; i < PROCESSOR_SIZE; i += STRIDE) {
        uint k = i & mask;
        uint index = ((i & ~mask) << 1) | k;
        Complex<T> factor = Complex<T>::W(m, k);
        butterfly(transform[index], transform[index + offset], factor);
    }
}

void FFT(Complex<T> signal[], Complex<T> transform[], int n) {
    init<T><<<BLOCKS_NUM, THREADS_NUM>>>(signal, transform, n);
    for (int m = 2, mask = 0; m <= n; m <<= 1, mask = (mask << 1) | 1)
        step<T><<<BLOCKS_NUM, THREADS_NUM>>>(transform, m, mask);
}
```

- 首先是 bitReverse，原先是传入位数 `nbits`，修改后传入 `n`，这里 `n` 是 2 的幂次，将除 2 操作用 **右移一位** 代替；将模 2 操作用与 1 相与代替；将加法用左移和或运算代替，将全部运算都用位运算实现，可以极大提高执行效率；
- 其次是第二部分提到的细粒度并行，`init` 为初始一次位逆序置换，`step` 为每一级操作，其中 `STRIDE` 为计算得到的步长，这符合 **内存模型** ，并行地读取同一块，保证 cache 命中率；
- 然后是 `step` 函数里，也做了大量的位运算优化，首先是 `FFT` 外层每一次通过位运算计算 `mask`，通过与 `mask` 相与，达到 **对 2 的幂次取模** 的效果(`mask` 的特点是后面几位全为1)；其次是计算 index 时针对 `(2 * tid / i) * i + tid % (i / 2)` 巧妙的利用 `mask` 取反再相与实现整除再相乘，特别地乘 2 用左移一位实现，加法利用或运算，因为正好是加到低位。

- 此外，在 butterfly 操作中，采用 **传引用** 来减少拷贝开销。

经过如上精心的设计和优化后，程序的性能得到了显著的提高，结构也变得精简，瓶颈的计算主要集中在蝴蝶操作（为复数乘法及加减），其余的除了有两处加法，剩下全部是位运算！

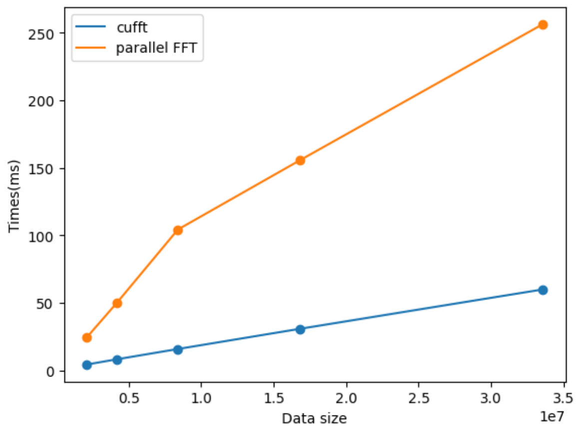
性能分析

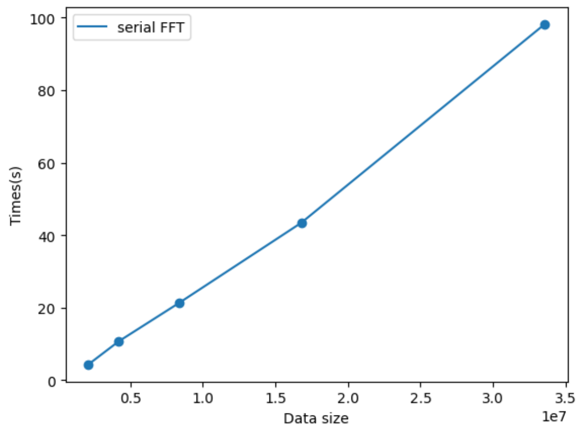
说明

本次实验通过对比串行 FFT、并行 FFT 依稀 CUDA 库提供的 cufft 三者对比来展现性能。同时与 cufft 的结果对比保证正确性。数据类型为 double，精度为 1e-6（但是在数据量过大后，例如 2^{25}，运行在 CPU 上的串行 FFT 精度会下降到 1e-5，而运行在 GPU 上的并行 FFT 以及 cufft 则没有影响）。

下面是针对不同数据量的测试结果，单位为 ms，默认 `1024 block 1024 thread`：

Data Size	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}
cufft	4.23728	8.15056	15.7385	30.6392	59.814
parallel FFT	24.2675	50.1064	104.178	155.305	256.189
serial FFT	4329	10718	21395	43404	98161





说明

右边将单位化为了秒。

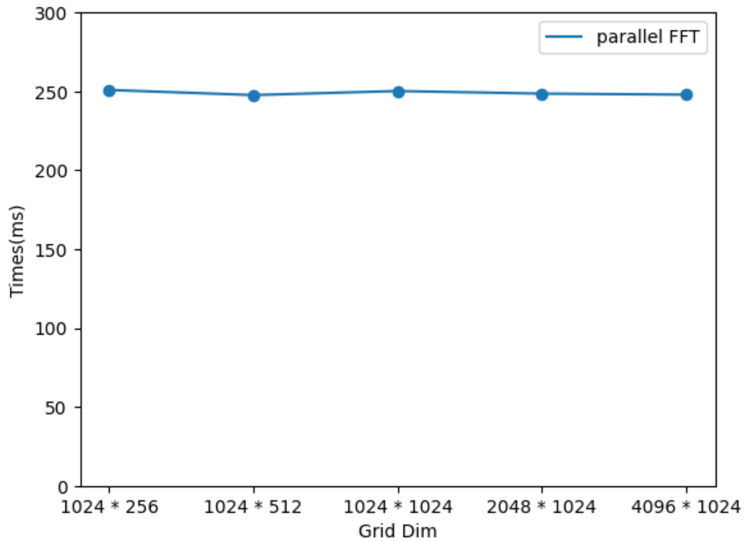
GPU 端的计时采用 CUDA 事件 API 来进行，而串行的则是用 `C++` 的高精度时钟 `chrono`。

可以发现，并行的加速比在 200 - 400，效果非常显著！此外对比 cufft，我们并行效果也不逊色，性能达到其 1/4，考虑到库是底层做了更多优化，因此我们效果能做到如此应该相当不错。

还有一点值得注意的是，虽然理论的时间复杂度是 $\Theta(\lg n)$，但是实际上从结果来看是更接近线性的，数据量翻倍，时间也几乎翻倍。猜测原因一是线程数不及数据量，无法达到理论那种程度；以及线程之间调度，调用核函数等等都存在额外开销。

下面还针对不同的 `block` 数和 `thread` 数的组合进行了测试：

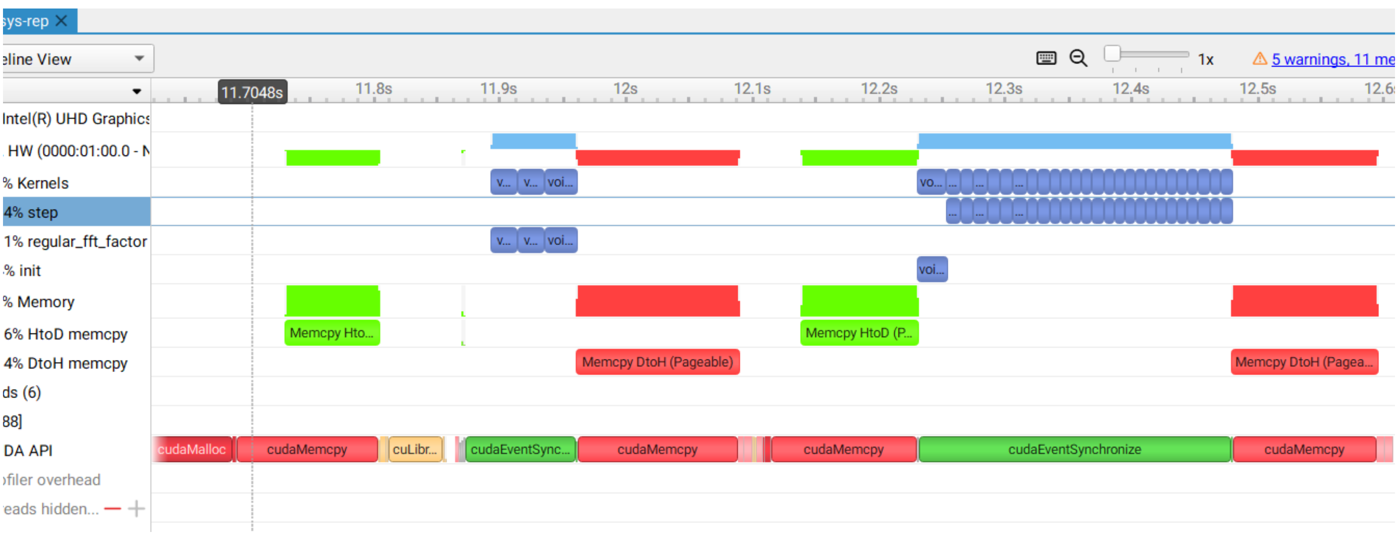
dim	$1024 * 256$	$1024 * 512$	$1024 * 1024$	$2048 * 1024$	$4096 * 1024$
Time	250.83	247.589	250.156	248.511	247.877



可以发现运行时间几乎持平，说明在达到一定程度后，增加 block 数或 thread 数也不一定能提高性能，甚至也有可能导致性能下降。

说明

下面还利用 nsys 对程序进行了分析（因为目前高版本不支持 nvprof，所以改用 nsys）。



实验总结

本次实验主要在于设计并行 FFT 的算法，以及根据 cuda 特点对程序进行优化，一步步设计以及优化后，程序性能得到了显著提升，在这个过程中也收益匪浅。