基于CUDA实现FFT(快速傅里叶变换)总结报告

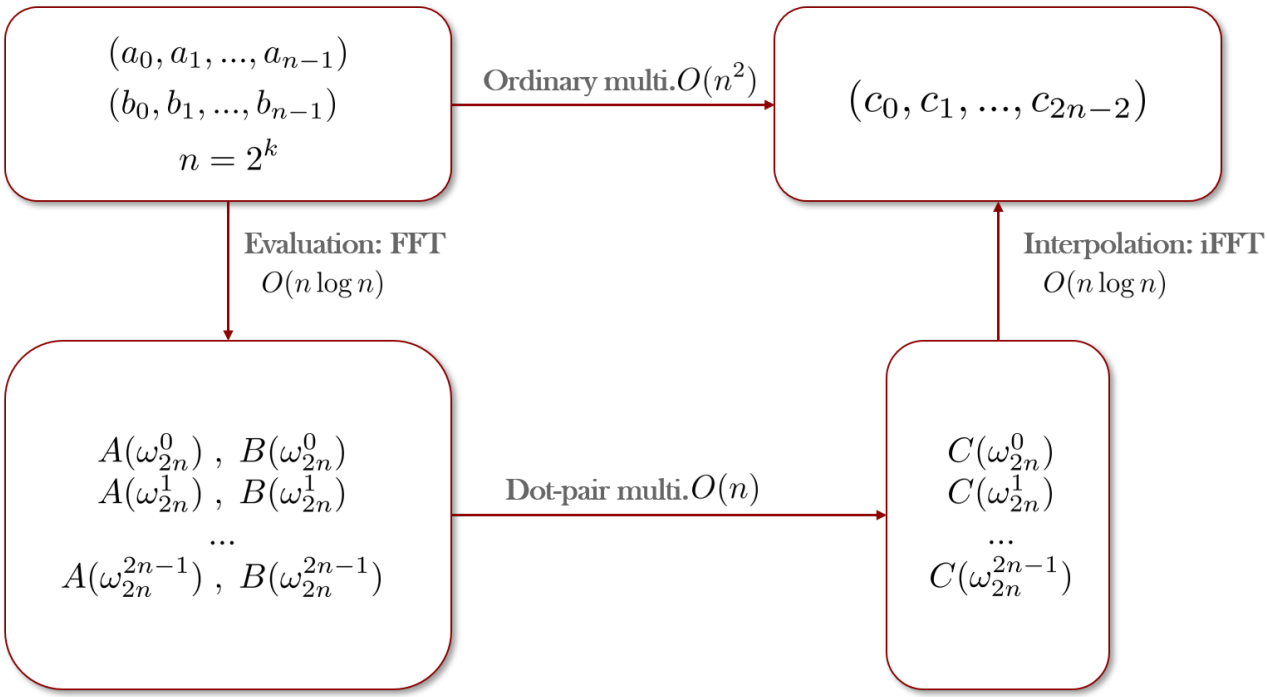
1 实验环境

所有程序的测试(benchmark)使用了两套环境，分别是在本地(Local)测试的CPU程序代码和在远端服务器(Server)测试的CUDA代码，环境如下：

	Local	Server
CPU	Intel i7-8550U	Intel(R) Xeon(R) Gold 6148 CPU
GPU	MX-250	Tesla V100
Language	C++	C++
Compiler	NVIDIA 11.3 ， g++ 8.3.0	NVIDIA 10.1 (with O3 optimization)
OS	Debian GNU/Linux 10 (buster)	Ubuntu 16
Benchmark	Google Benchmark	C++

2 算法描述

针对两个多项式的乘积，有如下3种算法，分别是普通的乘积算法，FFT算法和并行的FFT算法



2.1 OM算法

直接进行矩阵乘法的OM算法:

$$C(x) = \sum_{j=0}^{2n-2} c_j x^j \qquad c_j = \sum_{k=0}^{j} a_k b_{j-k}$$

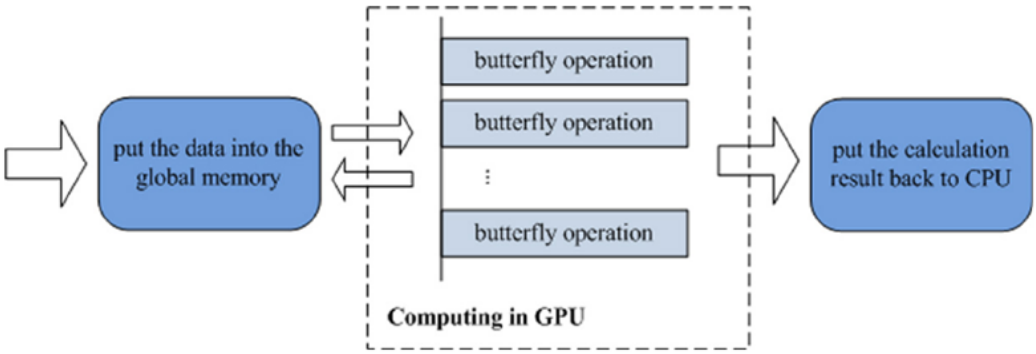
2.2 rFFT算法

基于递归实现的FFT算法：

$$P(\omega^j) = P_e(\omega^{2j}) + \omega^j P_o(\omega^{2j})$$
$$P(\omega^{j+\frac{n}{2}}) = P_e(\omega^{2j}) - \omega^j P_o(\omega^{2j})$$

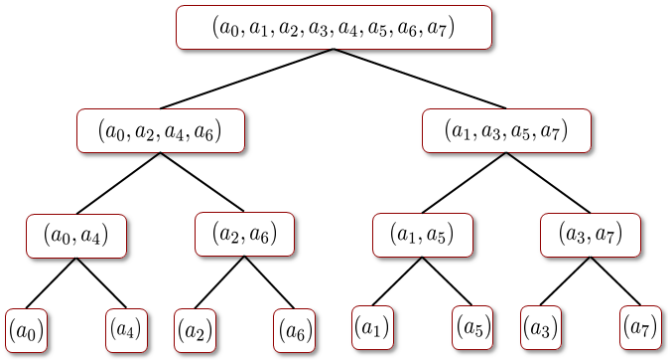
2.3 gFFT算法

基于GPU并行计算对rFFT进行的二重优化（去除递归，进行并行化处理）：



3 实现方法

3.1 rFFT实现



在递归后进行系数的计算，往上回溯从而得到整个系数的值表示(DFT)。

$$y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$$

整个过程的时间复杂度满足

$$T(N) = 2 * T(\frac{N}{2}) + \Theta(N) = \Theta(N \log N)$$

3.2 gFFT实现

为了实现GPU上的FFT算法，首先需要去除rFFT中的递归部分(具体代码，查看附件文件)。由递归树可知，如果按照最底层叶子节点的顺序对原数组进行排序，就可以做到in-place的计算模式。因此首先需要进行排序操作。进一步考虑可知，这个排序过程可以独立于FFT算法提前完成。考虑使用一个额外的数组空间作为排序后的数组，这个过程，数组元素之间没有相互干扰，因此这部分可以放到GPU进行执行。

```
__global__ void bits_rev(Complex* __restrict__ reordered, Complex* __restrict__ device, int s,
size_t threadN) {
    int id = blockIdx.x *threadN + threadIdx.x;
    reordered [__brev(id) >> (32 - s)] = device [id];
}
```

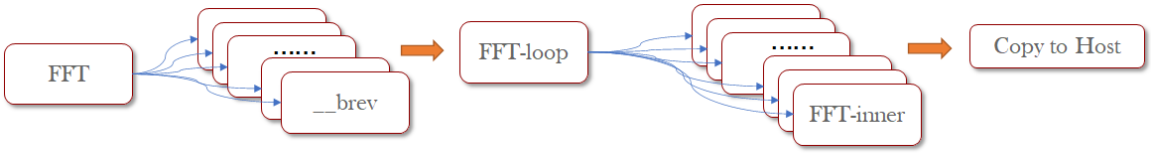
在已经排序的基础上，原地迭代的FFT框架如下：

```
for(half =1;half<N;half<<1)
        for(j=0;j<N;j+=2*half)
            for(k=0;k<half;k++)
                    ......
```

也就是说，每次对原递归树的一层的固定数目的元素操作，互不干扰，因此可以将内部循环并行化。伪代码如下：

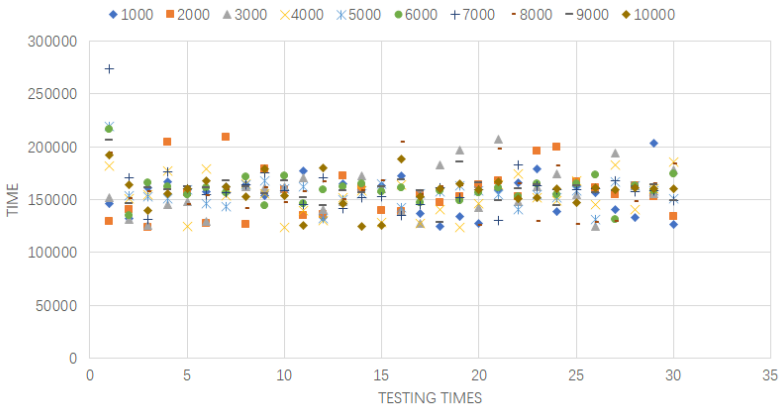
```
void FFT(Complex *__restrict__ data, size_t N, size_t threads,int flag) {
    sort<<< ceil(N / threads), threads >>> (...);
    cudaDeviceSynchronize();
    for (int i = 1; i <= s; i++) {
        int m = 1 << i; // for each layer
        FFT_core_loop_paralel<<< ceil((float)N / m / threads), threads >>>(...);
    }
}
```

因此，整个的算法模型如下：



4 算法特性的量化分析

对于gFFT，在多项式阶为1000~10000这10个分类下的测试结果如下：



计算时间随数据的波动较小，基本维持在150ms附近。

设计异构计算的函数有这三个，其中kernel function有两个，剩下一个函数作为device function call 会被编译器内联。

```
__global__ void iteration_FFT_core_loop_paralel
__device__ void iteration_FFT_core_loop
__global__ void bits_rev
```

在block 和 threads分配上，每次分配threads为固定的512，block大小根据数据规模进行变化。

```
int j = (blockIdx.x *threadN + threadIdx.x)*m;
```

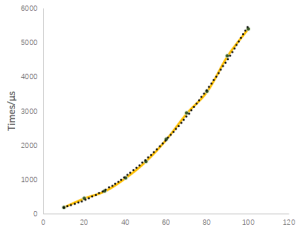
这种会分配寄存器存储。程序中，仅仅访问了全局内存，而不存在对于shared memory 或者 constant memory的访问。主要使用

```
cudaMalloc((void**)&reoredred, dataSize);
cudaMalloc((void**)&device, dataSize);
```

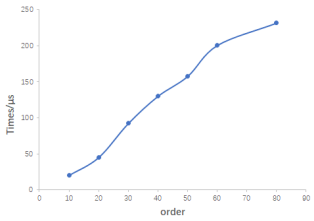
因此，内存空间的开销正比于数据量。

5 实验结果展示

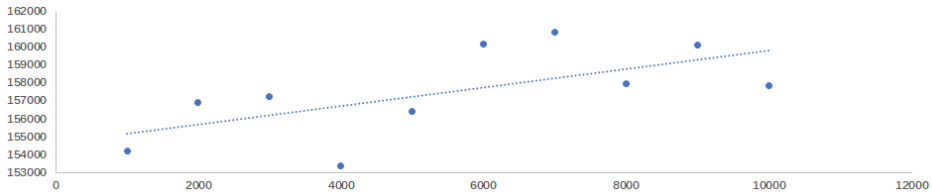
5.1 OM算法



5.2 rFFT算法



5.3 gFFT算法



6 分析与评价

FFT能并行化的关键在于，对于递归的修改，使其成为迭代算法。通过对于迭代算法多层循环的观察修改，发现可以按照树的Bottom-Up方法进行并行化处理，因为能利用CUDA程序，提高算法的效率。但是值得注意的是，虽然CUDA程序对于大规模数据的处理能力是OM算法，rFFT算法的数倍。但是对于小规模数据而言，大量的并行是没有意义且消耗巨大的，这种情况下gFFT的效率会慢数倍乃至数十倍。因此FFT算法在针对不同的数据量时，应该合理的选取计算模式。对于含有大规模数据的计算，gFFT算法的并行性会带来质一般的性能飞跃。