电子科技大学（深圳）高等研究院

并行算法

课程设计报告

题目	《基于 MPI 的一维 FFT 并行化》
专业	控制科学与工程
学号	202122280534
姓名	陈玉熙
指导教师	段贵多
完成日期	2021.12.26

基于 MPI 的一维 FFT 并行化

摘　要：本文通过定量分析一维 FFT 的分解基数选取、旋转因子计算、倒序排列等各个阶段在 GPU 上并行执行时的特征，并提出以蝶形算子访存跨度为依据的共享内存访问机制，解决共享内存访问效率过低的问题。

一、引言

快速傅里叶变换（Fast Fourier Transform，FFT）算法是图像处理的基础算法之一，广泛应用于图像滤波、图像降噪、图像压缩等领域，同时也是基于频域分析的图像处理算法的基础。因此，针对 FFT 运算的硬件加速对提升系统性能有重要的意义。

主要的硬件加速平台包括图形处理器（Graphics Processing Unit，GPU）、现场可编程逻辑门阵列（Field Programmable Gate Array，FPGA）等。其中，GPU 因其先进的并行架构体系，能够支持海量的多线程、高内存带宽以及大计算能力，在数千个核心上分配大规模多线程，并行执行独立任务而备受青睐。由于 FFT 算法往往具有计算并行度高、数据流规整的特点，可以有效发挥 GPU 大规模并行计算的潜力，因而受到了广泛的关注。

目前，在 GPU 的 FFT 实现方面，MIT 开发了基于 CPU 的 FFT 算法库 FFTW(Fast Fourier Transform in the West，FFTW)，该库函数具有很强的移植性和自适应性，能够自动配合所运行的硬件平台，通过最优化搜索，找到最优的组合使运行效率达到最佳。2003 年 Kenneth Moreland 与 Edward Angel 利用 GPU 的着色器编译程序把 FFT 算法移植到了 GPU 平台上[①]。2007 年 NVIDIA 公司推出 CUDA 并行开发环境，同时也发布了基于 CUDA 的 FFT 库函数 CUFFT，该库进行 FFT 处理的速度大约是同期 CPU 进行 FFT 运行速度的 20 倍[②]。2008 年，Vasily Volkov 在 GPU 上对 FFT 算法进行改进使一维 FFT 算法效率比 CUFFT 1.1 提高 3 倍[③]。N.K.Govindaraju, B.Lloyd, Y.Dotsenko 等人在 Volkov 的基础上使得一维 FFT 效率比 CUFFT 1.1 提高 4 倍，并实现了二维 FFT[④⑤]。随后，NVIDIA 把 Volkov 等人改进的算法加入到下一版本的 CUFFT 中。

然而，上述这些算法库都是闭源的，无法对其进行修改、分解和组合，灵活性差，不能适应所有的应用场景。本文尝试自行实践 FFT 算法的并行化。

二、GPU 上一维 FFT 算法的整体结构设计

2.1 算法原理

1963 年，J.W.Cooley 和 J.W.Tukey 提出的 Cooley-Tukey 算法是 FFT 算法的早期版本，该算法以分治法为策略使 DFT 的运算量从 $O(N^2)$ 减小到了 $O(N\log_2^N)$，计算效率提高 1~2 个数量级。目前除了 Cooley-Tukey 算法以外，还涌现出许多高效的算法，包括素因子算法(Prime Factor)、分裂基算法，混合基算法等[6]。

本文选取 Cooley-Tukey 算法来实现 FFT。其原理是用多个长度较小的 DFT 来计算长度较大的 DFT，经过反复迭代，最终分解成多个最小尺寸 DFT。这个最小尺寸 DFT 称作蝶形运算单元，而最小尺寸称作基数。在这基础上，根据基数的性质，又有高基和分裂基等多种算法。FFT 分解基数的大小决定了分解后的级数。

一维 DFT 计算式：

$$X(k)=T_{DFT}\left[x(n)\right]=\sum_{n=0}^{N-1}x(n)W_N^{nk} \tag{1}$$

其中：$k=0,1,\ldots,N-1$，$W_N^{nk}=e^{-j\frac{2\pi nk}{N}}(n=0,1,2,\ldots,N-1)$ 称为选择因子，

N 为 DFT 计算长度。

Cooley-Tukey 算法将数据长度 N 分解成两个因子的乘积，即：$N=N_1\times N_2$，将时域信号索引 n 表示成：

$$n=N_2n_1+n_2 \tag{2}$$

其中：$n_1\in[0,N_1-1],n_2\in[0,N_2-1]$。

将频域输出信号索引表示为：

$$k=k_1+N_1k_2 \tag{3}$$

其中：$k_1\in[0,N_1-1],k_2\in[0,N_2-1]$。

经过推导得到基于 Cooley-Tukey 算法的 DFT 计算：

$$X\left(k_1,k_2\right)=\sum_{n_2=0}^{N_2-1}W_{N_2}^{n_2k_2}W_N^{n_2k_1}\sum_{n_1=0}^{N_1-1}x\left(n_1,n_2\right)W_{N_1}^{n_1k_1} \tag{4}$$

图 1 展示了以 2 为基数的 8 点 Cooley-Tukey FFT 算法，蝶形运算结构共 3 级。最后一级的输出结果是顺序的，而第一级的输入序列的顺序需要重新排列，称为倒序排列。每个蝶形运算单元有自己的旋转因子，蝶形单元的输入需要先乘上对应的旋转因子，再进行蝶形运算。旋

转因子的底数取决于 FFT 点数，而指数与蝶形单元所在的级数及其本身的序数有关。

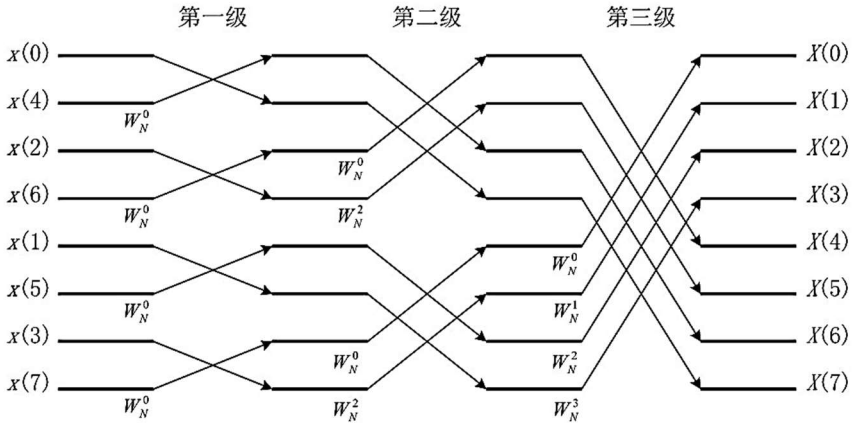

图 1. 基-2 Cooley-Tukey FFT 算法

2.2 算法整体结构设计

基于一维 FFT 算法的原理，我们首先分析一维 FFT 算法的可并行性。以单一基数的一维 FFT 为例，假设基数为 R，长度为 $N=R^M$ 的 FFT 算法运算结构共 M 级，每级包含 N/R 个蝶形单元。同一级的蝶形单元互不相关，可并行执行。但后一级的蝶形单元的输入数据来源于前一级中 R 个相关蝶形单元的输出数据，相邻两级存在依赖关系，不可并行，必须按顺序串行执行。

根据上述分析结果，本文编写了一维 FFT 算法在 GPU 上的并行实现过程，如图 2 所示。图中的主要模块包括分解基数选取、倒序排列、旋转因子计算以及蝶形运算。分解基数的选取决定了每一级的蝶形运算结构。每一级的基数不必相同，基数不同时，蝶形单元的结构也不同。倒序排列发生在输入序列的读取阶段。每一级中的蝶形单元结构相同，但是指向不同的内存地址。两级蝶形结构之间通过共享内存进行数据通信。

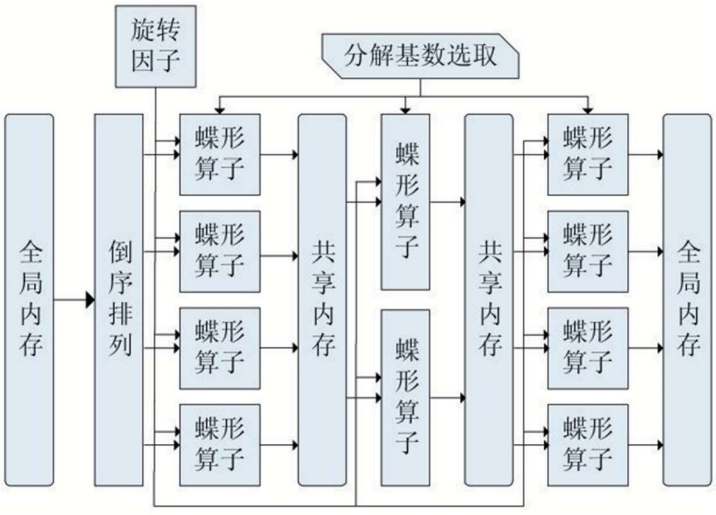

图 2. GPU 上一维 FFT 算法基于共享内存优化并行实现过程

各步骤分析如下：

1. 计算旋转因子：

使用常量内存查找表或 CUDA 内置函数库中的快速三角函数__sinf()、__cosf()来得到旋转因子 θ 的三角函数值。比较设备的运算能力和常量内存的带宽大小，从而决定到底使用哪种旋转因子 θ 的计算方式。计算密集型内核采取查找表方法，内密集型则采取实时计算方法。

2. 倒序排列：

倒序排列会导致全局内存的不连续访问。线程束中的线程访问全局内存不连续时，GPU 会发出超过原本次数的访存事务，称为全局负载重播。全局负载重播会消耗过多的内存带宽，导致严重的性能问题。不产生负载重播的全局内存访问称为合并访问。设计中引入一级额外的共享内存作为缓存，在此缓存空间中进行倒序排列，之后倒序的序列传向后递到第一级蝶形运算结构。这种方法实际上将不连续的内存访问风险从全局内存转嫁到共享内存。幸而共享内存在不连续访问的情况下仍然可以实现较高的带宽利用率，从而解决全局内存访问不连续的问题。

3. 蝶形运算：

蝶形运算包括两个部分：共享内存访问和运算。运算是简单的乘累加运算，而共享内存的访问则较为复杂。因为每一级蝶形结构的访存跨度在不断变化，所以共享内存访问过程中极容易发生 bank conflict。以图 1 为例，每一级蝶形单元的地址访问跨度 N_s 依次为 1、2、4，均小于 32，因此线程有可能同时占用两个或以上的存储体，导致共享内存发生 bank conflict。为了避免上述问题，本文设计了以蝶形单元访存跨度为依据的共享内存访问方式，写入数据时，每隔 $R \times N_s$ 个数据填充一个无效数据。

4. 分解基数选取：

FFT 算法分解过程中，基数越大，蝶形单元的级数越少，则同步操作次数越少，而且内存的访问次数降低。但是大基数也存在相应的劣势，基数越大，每个蝶形单元的规模更大，计算量更高，占用寄存器数量越大。由此可以推断，随着基数的增大，程序的性能瓶颈从内存密集型过渡到计算密集型。

2.3 基础算法设计

根据上述分析结果，我们以 2 为基数进行 FFT 算法编写，具体步骤如下：

（1）数据处理并输入并行处理器；

（2）完成倒位序重排并计算旋转因子；

（3）确定运算级数 M；

（4）在每一级并行 $N/2$ 个蝶形运算；

（5）重复 3～4 完成 M 级蝶形运算。

我们先对各子功能模块进行编写，结果如下：

1. 扩展输入系数

```
/*
 *@function 将系数项数扩展至2^X
 *@param num_coef: 输入的系数项数
 *@return 扩展后的项数
 * */
int Extend(int num_coef) {
    int m = log(num_coef) / log(2);
    if(num_coef == pow(2, m))
        return(num_coef);
    else
        return(pow(2, m + 1));
}
```

图 3. 扩展输入系数代码

2. 输入系数数组的实部、虚部

```
/*
 * @function 输入系数数组的实部和虚部
 * @param n: 系数个数
 * @param dataR: 实部
 * @param dataI: 虚部
 * */
void Input(int num_coef, int n, float* dataR, float* dataI){
    cout << "R: ";
    int i;
    for (int i = 0; i < num_coef; i++) {
        cin >> dataR[i];
    }
    cout << "I: ";
    for (i = 0; i < num_coef; i++) {
        cin >> dataI[i];
    }
    if (n > num_coef)
        for(; i < n; i++){
            dataR[i] = 0;
            dataI[i] = 0;
        }
}
```

图 4. 输入系数数组的实部、虚部代码

3. 码位交换

```
/*
 * @function 码位交换
 * @param n: 系数个数
 * @param coefR: 实部数组
 * @param coefI: 虚部数组
 *  */
void ReverseOrder(int n, float* coefR, float* coefI) {
    //码位个数
    int m = log(n) / log(2);
    //码位高位、低位
    int head, rear;
    //低位、高位为1 的数
    int a1, an;
    //码位交换后的数,用于交换的temp
    int j;
    float tempR, tempI;
    //遍历系数
    for (int i = 0; i < pow(2, m); i++) {
        j = 0;
        for (int k = 0; k < (m + 1) / 2; k++) {
            //第一位为1
            a1 = 1;
            //第M位为1
            an = pow(2, m - 1);
            //移位
            a1 = a1 << k;
            an = an >> k;
            //取高位、低位
            head = i & an;
            rear = i & a1;
            //交换码位模块
            if (0 != head)    j = j | a1;
            if (0 != rear)    j = j | an;
        }
        //数组元素交换
        if (i < j) {
            tempR = coefR[i];
            tempI = coefI[i];
            coefR[i] = coefR[j];
            coefI[i] = coefI[j];
            coefR[j] = tempR;
            coefI[j] = tempI;
        }
    }
}
```

图 5. 码位交换代码

4. 蝶形运算

```cpp
/*
 * @function 进行碟形计算
 * @param P: 旋转因子
 * @param n: 系数个数
 * @param r: 数组下标
 * @param B: 间隔
 * @param dataR: 实部
 * @param dataI: 虚部
 * */
void Calculate(int P, int n, int r, int B, float* dataR, float* dataI){
    /*进行蝶形运算*/
    //t = W * A1, W = cos(2 * PI * P / n) - i * sin(2 * PI * P / n)
    float tR = dataR[r + B] * cos(2 * M_PI * P / n) + dataI[r + B] * sin(2 * M_PI * P / n);
    float tI = dataI[r + B] * cos(2 * M_PI * P / n) - dataR[r + B] * sin(2 * M_PI * P / n);
    //A(r) = A0 + W * A1
    //A(r + B) = A0 - W * A1
    dataR[r + B] = dataR[r] - tR;
    dataI[r + B] = dataI[r] - tI;
    dataR[r] = dataR[r] + tR;
    dataI[r] = dataI[r] + tI;
}
```

图 6. 蝶形运算代码

5. 结果输出

```cpp
/*
 * @function 输出数组（复数形式）
 * */
void Print(int n, float* dataR, float* dataI){
    cout << "result:" << endl;
    for (int i = 0; i < n; i++) {
        if(dataI[i] >= 0)
            cout << dataR[i] << "+" << dataI[i] << "i" << "\t";
        else
            cout << dataR[i] << dataI[i] << "i" << "\t";
    }
    cout << endl;
}
```

图 7. 结果输出代码

2.4 串行算法设计

利用上述功能函数，我们可以编写出 FFT 的串行算法：

```
int main(int argc, char** argv) {
    //使用定点计数法，显示小数点后面位数精度为2
    cout.setf(ios_base::fixed, ios_base::floatfield);
    cout.setf(ios_base::showpoint);
    cout.precision(2);

    //系数输入个数
    int num_coef;
    cout << "the number of coefficients you want to input is: ";
    cin >> num_coef;
    //系数数组，进程数组
    float *dataR, *dataI;

    //扩展系数
    int n = Extend(num_coef);
    //分配空间
    dataR = new float[n];
    dataI = new float[n];

    //系数输入
    Input(num_coef, n, dataR, dataI);
    //码位倒序
    ReverseOrder(n, dataR, dataI);

    //蝶形级数
    int M = log(n) / log(2);
    //FFT蝶形级数L从1--M
    for(int L=1; L<=M;L++){
        /*第L级的运算*/
        //间隔 B = 2^(L-1);
        int B = pow(2, L - 1);
        for(int j = 0; j <= B - 1; j++){
            /*同种蝶形运算*/
            //增量k=2^(M-L)
            int k = pow(2,M-L);
            //计算旋转指数p，增量为k时，则P=j*k
            int P = j * k;
            for(int i = 0; i <= k - 1; i++){
            /*进行蝶形运算*/
            //数组下标定为r
            int r = j + 2 * B * i;
            Calculate(P, n, r, B, dataR, dataI);        }
        }
    }

    //输出
    Print(n, dataR, dataI);
}
```

图 8. FFT 串行代码

2.5 并行算法设计

我们基于 MPI 对串行代码并行化，主要是对串行代码的循环部分进行修改，修改如下：

```
//FFT蝶形级数L从1--M
for (int L = 1; L <= M; L++){
    MPI_Bcast(dataR, n, MPI_FLOAT, 0, MPI_COMM_WORLD);
    MPI_Bcast(dataI, n, MPI_FLOAT, 0, MPI_COMM_WORLD);

    /*第L级的运算*/
    //同一组蝶形计算的间隔 B = 2^(L-1);
    int B = (int)pow(2, L - 1);
    //旋转指数P的增量k=2^(M-L)
    int k = (int)pow(2, M - L);

    for (int j = 0; j < msize; j++){
        //数组下标定为r
        int r = start + j;

        //如果对应蝶形组的下方，即后半部分
        if ((r % (2 * B)) >= B){
            //如果不由本进程更新，则接收其他进程发送过来的结果
            if(j - B < 0) {
                MPI_Recv(&mdataR[j], 1, MPI_FLOAT, (r - B) / msize, r, MPI_COMM_WORLD, &status);
                MPI_Recv(&mdataI[j], 1, MPI_FLOAT, (r - B) / msize, r, MPI_COMM_WORLD, &status);
            }
            continue;
        }

        //计算旋转指数p
        int P = (r % B) * k;

        /*进行蝶形运算*/
        Calculate(P, n, r, B, dataR, dataI, tempR, tempI, mdataR[j], mdataI[j]);
        //如果data(r + B)不属于本进程mdata
        if(j + B + 1 > msize){
            //如果data(r + B) 已被分配所到进程中则将结果发送过去，否则不发送
            if(r + B < msize * size){
                MPI_Send(&tempR, 1, MPI_FLOAT, (r + B) / msize, r + B, MPI_COMM_WORLD);
                MPI_Send(&tempI, 1, MPI_FLOAT, (r + B) / msize, r + B, MPI_COMM_WORLD);
            }
        }
        //如果A(r + B)属于本进程mdata则直接进行更新
        else{
            mdataR[j + B] = tempR;
            mdataI[j + B] = tempI;
        }
    }

    /*处理未被分配的元素*/
    //未被分配元素开始的下标
    int remainStartID = msize * size;
    //在0号进程中直接处理未被分配的元素，当remainStartID == num_coef时说明全部被分配
    if((0 == myrank) && (remainStartID != n)){
        for(int j = 0; j < n - remainStartID; j++){
            int r = remainStartID + j;

            //计算旋转指数p
            int P = (r % B) * k;

            //如果位于蝶形组下方，即后半部分
            if ((r % (2 * B)) >= B) {
                //如果不由处理未被分配元素的模块更新
                if(j - B < 0) {
                    //t = W * A1, W = cos(2 * PI * P / n) - i * sin(2 * PI * P / n)
                    float tR = dataR[r] * cos(2 * M_PI * P / n) + dataI[r] * sin(2 * M_PI * P / n);
                    float tI = dataI[r] * cos(2 * M_PI * P / n) - dataR[r] * sin(2 * M_PI * P / n);
                    //A(r) = A0 + W * A1
                    //A(r + B) = A0 - W * A1
                    dataR[r] = dataR[r - B] - tR;
                    dataI[r] = dataI[r - B] - tI;
                }
                continue;
            }

            /*进行蝶形运算*/
            Calculate(P, n, r, B, dataR, dataI, dataR[r + B], dataI[r + B], dataR[r], dataI[r]);
        }
    }
    //等待此级蝶形变换全部计算完毕
    MPI_Barrier(MPI_COMM_WORLD);
    //从各个进程中收集结果，更新dataR, dataI
    MPI_Gather(mdataR, msize, MPI_FLOAT, dataR, msize, MPI_FLOAT, 0, MPI_COMM_WORLD);
    MPI_Gather(mdataI, msize, MPI_FLOAT, dataI, msize, MPI_FLOAT, 0, MPI_COMM_WORLD);
}
```

图 9. FFT 并行化修改

三、实验结果

运行上述代码，最终的实验结果截图如下所示：

```
root@kali:~/桌面/C/mpi# g++ -o FFT3 ./FFT3.cpp
root@kali:~/桌面/C/mpi# ./FFT3
the number of coefficients you want to input is: 5
R: 1 1 1 1 1
I: 0 0 0 0 0
result:
5.00+0.00i      0.00-2.41i      1.00+0.00i      0.00-0.41i      1.00+0.00i      -0.00+0.41i     1.00+0.00i      -0.00+2.41i
root@kali:~/桌面/C/mpi#
```

图 10. FFT 串行程序运行结果

```
root@kali:~/桌面/C/mpi# mpic++ -g -o myFFT myFFT.cpp
root@kali:~/桌面/C/mpi# mpirun -n 3 ./myFFT 5
R: 1 1 1 1 1
I: 0 0 0 0 0
result:
5.00+0.00i      0.00-2.41i      1.00+0.00i      0.00-0.41i      1.00+0.00i      -0.00+0.41i     1.00+0.00i      -0.00+2.41i
root@kali:~/桌面/C/mpi# mpirun -n 2 ./myFFT 5
R: 1 1 1 1 1
I: 0 0 0 0 0
result:
5.00+0.00i      0.00-2.41i      1.00+0.00i      0.00-0.41i      1.00+0.00i      -0.00+0.41i     1.00+0.00i      -0.00+2.41i
root@kali:~/桌面/C/mpi# mpirun -n 3 ./myFFT 8
R: 1 1 1 1 1 1 1 1
I: 0 0 0 0 0 0 0 0
result:
8.00+0.00i      0.00+0.00i      0.00+0.00i      0.00+0.00i      0.00+0.00i      0.00+0.00i      0.00+0.00i      0.00+0.00i
root@kali:~/桌面/C/mpi#
```

图 11. FFT 并行程序运行结果

四、总结

本文首先分析了 FFT 算法的基本原理，分析了 FFT 算法并行化的可实现性。进而提出了 GPU 上一维 FFT 算法的整体结构设计，包括计算旋转因子、倒序排序、蝶形运算、系数扩展等，并介绍各个主要模块的高效实现方法。最后，对代码进行性能对比测试，验证代码的并行化效果。

四、参考文献

① Moreland K, Angel E. The FFT on a GPU [C]//**Proceedings of the ACM Siggraph/Eurographics Conference on Graphics Hardware**, San Diego, California, July 26-27, 2003: 112-119.

② 赵丽丽，张盛兵，张萌，等. 基于的高速计算 [J]. 计算机应用研究，2011，28(4)：1556-1559.
ZHAO Lili，ZHANG Shengbing，ZHANG Meng，et al. High performance FFT computation based on CUDA [J]. **Application Research of Computers**，2011，28(4)：1556-1559.

③ Volkov V，Kazian B. **Fitting FFT onto the G80 Architecture** [M]. University of California，2008，E63(40)：1-12.

④ Naga K Govindaraju，Brandon Lloyd，Yuri Dotsenko，et al. High performance discrete Fourier transforms on graphics processors [C]// **Proceedings of the 2008 ACM/IEEE conference on Supercomputing**，Austin，Texas，November 15-21，2008：1-12.

⑤ Brandon L D，Boyd C，Govindaraju N. Fast computation of general Fourier Transforms on GPUs [C]// **IEEE International Conference on Multimedia and Expo**，Hannover，Germany，June 23-26，2008：5-8.

⑥ Rao K R, Kim D N, Hwang J J. 快速傅里叶变换:算法与应用[M].北京:机械工业出版社，2012: 1-33.