电子科技大学

电子科技大学（深圳）高等研究院

GPU 并行编程

FFT 并行算法设计及优化

姓名	学号	签名
蔡畅	202122280514	
陈玉熙	202122280534	
黄德伟	202122280505	
李金键	202122280421	
李育泓	202122280515	
刘宗林	202122280315	
汪果	202122280519	
王宇哲	202122280535	

1 FFT 介绍

1.1 从 DFT 到 FFT

离散傅里叶变化(Discrete Fourier Transform)是傅里叶变换的简化版本，计算机科学家发明出离散傅里叶变换目的就是要让具有连续性质的傅里叶变换能在具有离散性质的计算机中被应用。而离散傅里叶变换算法时间复杂度高，使其在大规模场景中的应用受到限制，从而诞生了快速傅里叶变换(Fast Fourier Transform)。

FFT 是离散傅立叶变换的快速算法，可以将一个信号变换到频域。有些信号在时域上是很难看出什么特征的，但是如果变换到频域之后，就很容易看出特征了。FFT 的基本思想是把原始的 N 点序列，依次分解成一系列的短序列。充分利用 DFT 计算式中指数因子 所具有的对称性质和周期性质，进而求出这些短序列相应的 DFT 并进行适当组合，达到删除重复计算，减少乘法运算和简化结构的目的。此后，在这思想基础上又开发了高基和分裂基等快速算法，随着数字技术的高速发展，1976 年出现建立在数论和多项式理论基础上的维诺格勒傅里叶变换算法(WFTA）和素因子傅里叶变换算法。它们的共同特点是，当 N 是素数时，可以将 DFT 算转化为求循环卷积，从而更进一步减少乘法次数，提高速度。

1.2 FFT 预备知识

1.2.1 复数

在复数范围内令 ω^n =1，可以得到 n 个不同的复数根 $\{undefined\omega^0, \omega^1 \ldots . \omega^{n-1}\}$，且均匀分布在模长为 1 的圆上。

单位根具有以下性质：

$$(1)\omega_n^m = \omega_{2n}^{2m}$$

$$(2)\omega_n^m = -\omega_n^{m+\frac{n}{2}}$$

$$(3)\omega_n^m = cos\frac{m}{n}2\pi + isin\frac{m}{n}2\pi$$

1.2.2 蝶形变换

对于 A(x)= $\sum_{i=0}^{n-1} a_i x_i = a_0 + a_1 x_1 + \ldots\ldots + a_{n-1} x_{n-1}$：

按照 A(x)下标的奇偶性将其分成两组，右边提出 x：

$$A(x) = (a_0 + a_2 x^2 + \ldots\ldots + a_{n-2} x^{n-2}) + (a_1 + a_3 x^3 + \ldots\ldots + a_{n-1} x^{n-1})$$

$$= (a_0 + a_2 x^2 + \ldots\ldots + a_{n-2} x^{n-2}) + x(a_1 + a_3 x^2 + \ldots\ldots + a_{n-2} x^{n-2})$$

令$A_0(x)$=$a_0 + a_2 x + \ldots\ldots + a_{n-2} x^{\frac{n}{2}-1}$，$A_1(x) = a_1 + a_3 x + \ldots\ldots + a_{n-2} x^{\frac{n}{2}-1}$。

得证有$A(x) = A_0(x^2) + xA_1(x^2)$。

在 n=8 时，可以得到如下的蝶形图。

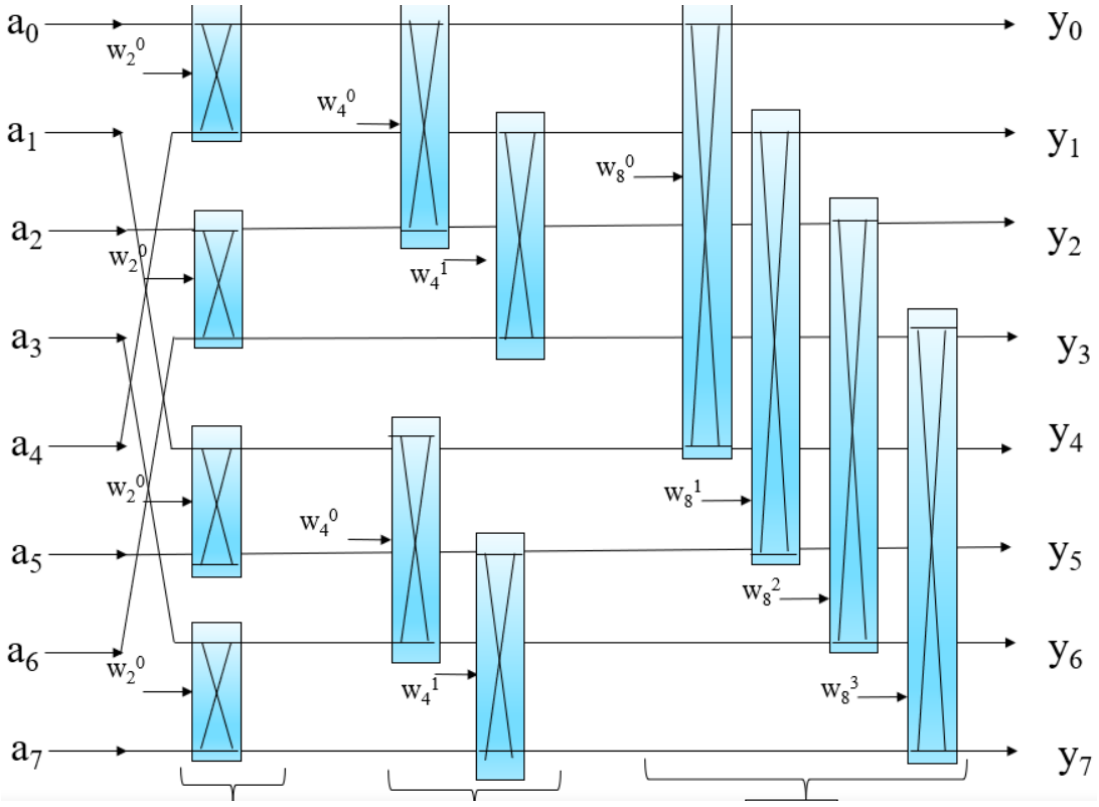

图 1-1 n=8 时蝶形图示意

对 A(x)代入ω_n^m可以得到以下结果：

$$A(\omega_n^m) = A_1(\omega_n^m)^2 + \omega_n^m A_2(\omega_n^m)^2 = A_1\left(\omega_{\frac{n}{2}}^m\right) + \omega_n^m A_2\left(\omega_{\frac{n}{2}}^m\right)$$

同样，再次代入$\omega_n^{m+\frac{n}{2}}$，可以得到结果：

$$A\left(\omega_n^{m+\frac{n}{2}}\right) = A_0\omega_n^{2m+n} + \omega_n^{m+\frac{n}{2}} A_2\omega_n^{2m+n} = A_0\omega_{\frac{n}{2}}^m - \omega_n^m A_1\omega_{\frac{n}{2}}^m$$

从上述的推导中可以得到两个结论：

$$A(\omega_n^m) = A_1\left(\omega_{\frac{n}{2}}^m\right) + \omega_n^m A_2\left(\omega_{\frac{n}{2}}^m\right)$$

$$A\left(\omega_n^{m+\frac{n}{2}}\right) = A_0\omega_{\frac{n}{2}}^m - \omega_n^m A_1\omega_{\frac{n}{2}}^m$$

当设定中 A0，A1 已知时，可以同时知道$A(\omega_n^m)$和$A\left(\omega_n^{m+\frac{n}{2}}\right)$的值，而 A0 和 A1 的规模是 A(x)的一半，所以 FFT 的时间复杂度可以控制在 O(nlogn)一级，是好算法。

1.2.3 码位倒序

当 n=8 时，可以得到一组典型的码位倒序。

表 1-1 典型码位倒序

顺序十进制数 I	顺序二进制数	倒序二进制数	倒序十进制数 J
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

倒序数和计算的顺序数一致，所以要在输入系数后对其进行码位倒序。

1.3 FFT 串行实现

在介绍 FFT 串行化实现之前，首先介绍卷积和离散卷积的定义：

$$(f * g)(t) \overset{\text{def}}{=} \int_{R_n}^{0} f(\tau)g(t-\tau)d\tau$$

$$(f * g)[n] = \sum_{m=-M}^{M} f[n-m]g[m]$$

给出[0,1,2,3]和[0,1,2]的卷积例子表示：

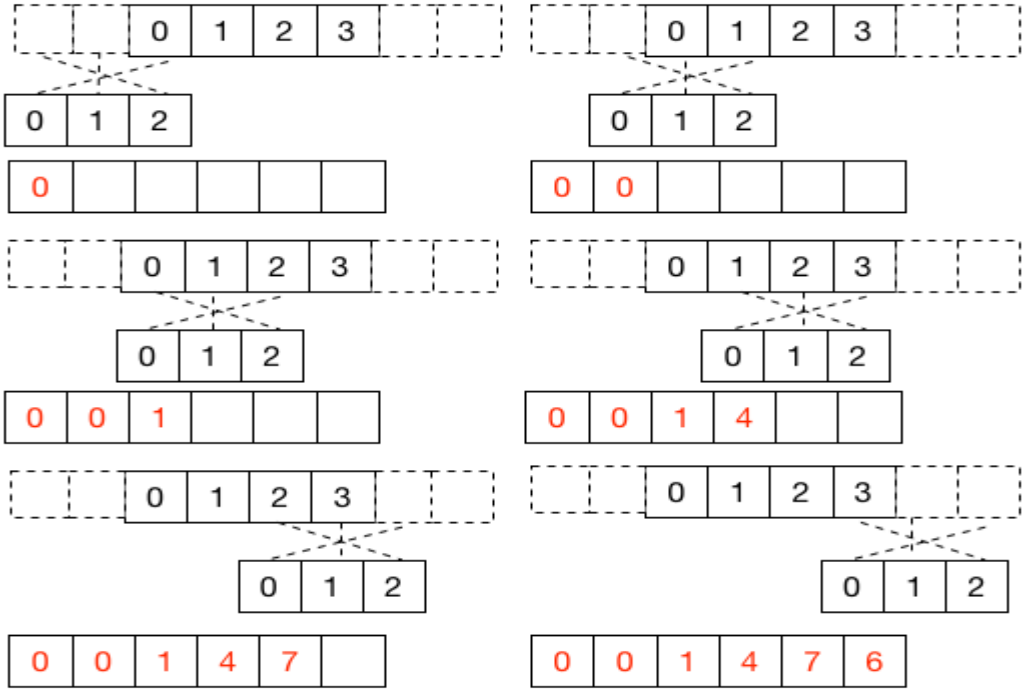

图 1-2 两种卷积例子

时域的卷积和频域的乘法是等价的，同时时域的乘法和频域的卷积也是等价的。基于这个这个前提，可以把待卷积的数组进行 FFT 变换，在频域做乘法，然后再进行 IFFT 变换即可得到卷积结果。算法流程描述如下：

1、设 N=len(a), M = len(b), 其中 a, b 为待卷积的数组，将长度增加到

$$L>=N+M-1,L=2n,n\in ZL>=N+M-1,L=2n,n$$

即

$$L=2logN+M-12+1L=2log2N+M-1+1$$

2、增加 a, b 的长度到 L，后面补零。

3、分别计算 afft=fft(a)afft=fft(a)，bfft=fft(b)bfft=fft(b)。

4、abfft=afft×bfftabfft=afft×bfft。

5、用 IFFT 计算 abaft 的 FFT 逆变换，取前（N + M - 1）个值即为卷积结果。

直接卷积的时间复杂度为 o(MN)，即 o(n^2) 。FFT 的时间复杂度为 o(nlogn)，FFT 卷积复杂度为 3 次 FFT+L 次乘法，3o(nlogn)+o(n)=o(nlogn) ，及 o(nlogn)。在实际应用中，卷积核（b）被提前计算，则只需 2 次 FFT 变换。

2 FFT 并行程序设计及优化

2.1 FFT 并行程序设计

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 2，长度为 $N=2^M$ 的 FFT 算法运算结构共 M 级,每级包含 $N/2$ 个蝶形单元。

因此，对于 N 点的 FFT 计算复杂度为（$N\log_2 N$），整个计算可以分为（$\log_2 N$）级，在每一级中要进行 $N/2$ 次蝶形运算。由于每个蝶形运算只与两个数据有关，而与其它数据无关。这样，对于同一级的 $N/2$ 个蝶形运算是相互独立，所以可并行执行；但是由于不同级的蝶形运算存在数据通信，故要串行运算，这样整个 FFT 运算用（$\log_2 N$）步骤可以完成。

根据上述分析结果，我们具体的 FFT 并行算法的设计步骤如下：

（1）把待处理的数据输入并行处理器；

（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。

而具体到 GPU 执行的过程中，在每级的 $N/2$ 蝶形运算里，为了避免数据的反复重排，在调用 FFT 变换 Kernel 函数前需要进行数据重排。我们将数据从主机 Host 端输入设备 Device 存储器，通过循环调用 GPU 的 FFT 核 M 次，最后将数从 Device 端输出到 Host 端。具体步骤如下：

（1）每行设定为一个线程块，块内有多个线程，用来计算一个或多个蝶形运算。即确定核内线程块和线程数量；

（2）变换前把数据重排，并将加权后的投影数据输入 GPU 全局存储器中。该步骤中不需额外开辟存储空间。

（3）循环 M 次调用 CUDA 的 Kernel 函数；

（4）为了减少传输时间，只需在循环开始前将数据从 Host 输入 Device，循环结束后，由 Device 输出到 Host。

2.2 并行程序设计优化

2.2.1 存储层次优化

众所周知，CUDA 共有六种不同的存储部件，其存储层次如图 2-1 所示：

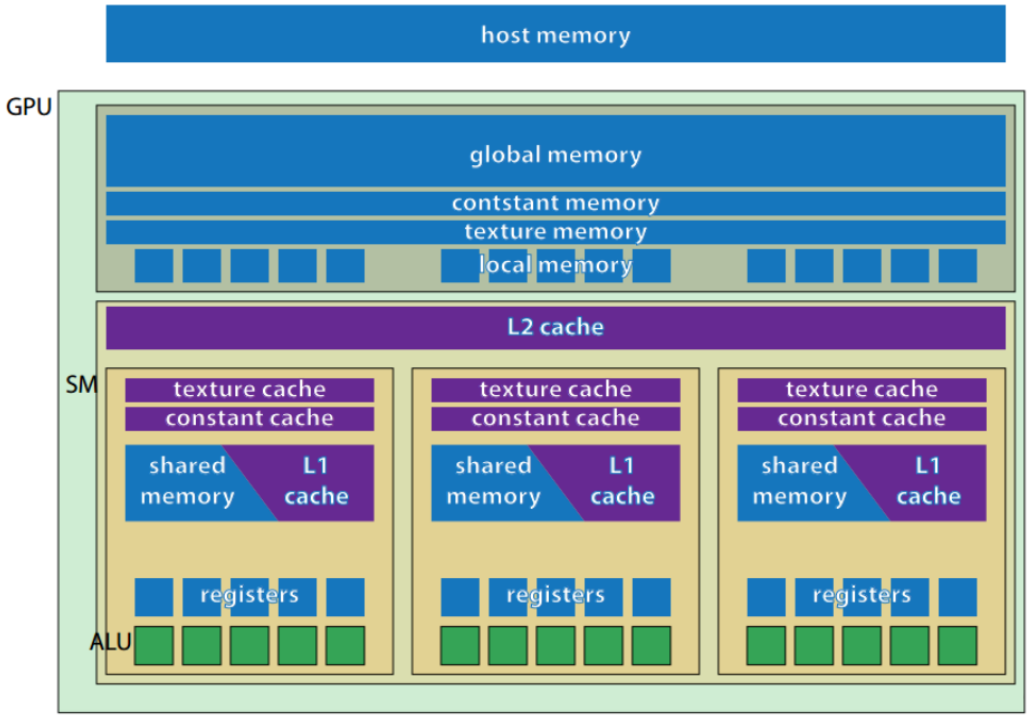

图 2-1 CUDA 存储层次

线程在执行时可以访问处于多个不同存储空间中的数据，根据这些存储器的位置、容量、访问权限和生存周期的特点，对算法实现进行优化，提高整体访存速度是有很大意义的。因此，我们针对存储调度进行了如下四点优化：

（1）由于每一个线程都拥有自己的私有寄存器（register）和局部存储器（local memory），而访问片上寄存器的时间约是 1~2cycles，远远高于访问显存上局部存储器的 200~300cycles。而当我们声明的局部变量太多不能在寄存器内分配时，系统自动将余出的局部变量放到局部存储器中，kernel 函数执行时频繁访问片外的局部变量，会降低程序运行速度。因此我们可以在 kernel 函数中使用尽可能少的局部变量，来提升算法的性能。

（2）共享存储器（shared memory）也是 GPU 片内的高速存储器，它是可以被同一线程块中所有线程访问的可读写存储器。在 FFT 算法中按级推进时，前级写入数据，次级读出数据，将这些数据存储在共享存储器（在容量允许范围内），

能在很大程度上提高所有线程的访存速度。因为访问片外全局存储器（global memory）具有较高的访存延时。在 kernel 函数设计中增加两个操作，计算开始前将显存的数据搬入共享存储器，待所有级运算完毕后再搬回显存。运算期间计算部件只与共享存储器交互，而无须访问片外的全局存储器。

针对共享内存优化过后的 FFT 计算流程图如下：

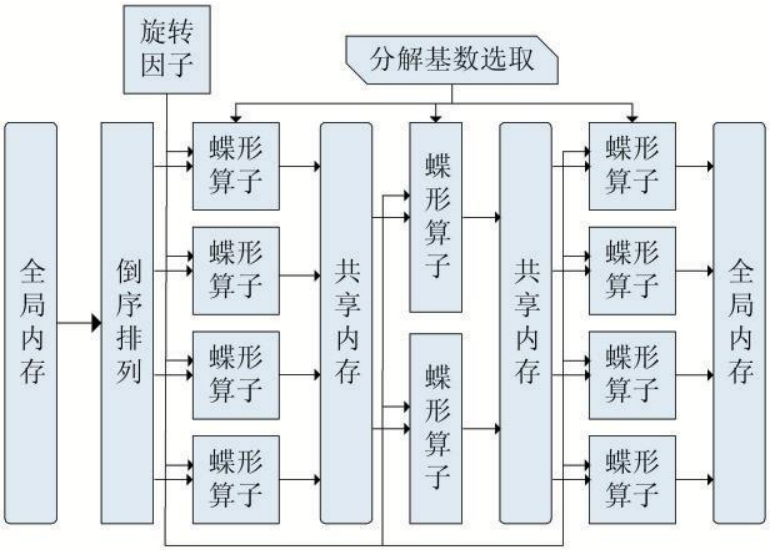

图 2-2 GPU 上一维 FFT 算法基于共享内存优化并行实现过程

（3）CUDA 存储模型中的只读存储器——纹理存储器（texture memory)，非常适合实现图像处理和查表，可以通过缓存加速访问。可以将旋转因子在计算前预先存储在纹理存储器中（在容量允许范围内），按级以矩阵的二维形式存储，这样可以降低重复计算旋转因子的计算量，同时利用纹理存储的高带宽提高运算精度。

（4）全局内存的访问是否满足合并条件是对 CUDA 程序性能影响最明显的因素之一，合并访问条件要求同一 half-warp 中的线程要按照一定字长访问经过对齐的段。FFT 实现时将采样点数据在存储器中对齐连续存储，旋转因子按二维矩阵存储，读取时数据存放满足不冲突要求，避免硬件自动将冲突访存转换为串行访存。

最终，经过存储层次优化的 FFT 算法 kernel 程序代码段如下：

```
__global__ void FFT_T (Complex * const DataIn, Complex * DataOut, const unsigned int N)
{
    extern __shared__ Complex sdata [] ;
    const unsigned int tid_in_block = threadIdx.x;
    if (tid_in_block < N) {
```

```
        sdata [tid_in_block]  = DataIn [tid_in_block] ;
        sdata [tid_in_block + N/2]  = DataIn [tid_in_block + N/2] ;
        __syncthreads( ) ;
        if (tid_in_block < N/2) {
            int p, q;
            Complex Wn, Xp, XqWn;
            float stage = 0. 0;
            for (int Ns = 1; Ns < N; Ns = Ns* 2) {
                p = tid_in_block/ Ns * Ns * 2 + tid_in_block % Ns;
                q = p + Ns;
                Wn = tex2D( texRef, tid_in_block, stage + + ) ;
                XqWn = ComplexMul( sdata [q] , Wn) ;
                Xp = sdata [p] ;
                sdata [p]  = ComplexAdd( Xp, XqWn) ;
                sdata [q]  = ComplexSub( Xp, XqWn) ;
                __syncthreads( ) ;
            } / /end for
            DataOut [p]  = sdata [p] ;
            DataOut [q]  = sdata [q] ;
        } //end if
    } //end if
} //end kernel
```

2.2.2 显存访问优化

计算机性能的瓶颈之一就是存储器的带宽。全局内存是 GPU 中不可避免使用的存储器，CPU 和 GPU 都可对其进行读写操作。在 CUDA 内核程序中不用考虑 I/O 和事务处理，每个线程束中都只进行访问存储和运算两种指令。因此，通过优化显存访问来解决显存带宽的瓶颈问题。显存访问优化存在以下几种方式：

（1）优化显存访问最为有效的方式就是减少对显存的访问，因此可以将具有相同线程块或者网格维度的多个 kernel 进行合并。

（2）在访问显存的过程中可以使用合并访问来隐藏内存延迟。所谓的合并访问就是指程序中的所有线程对连续的对齐的内存块进行访问。在代码编写时，使用 cudaMallocPitch()或者 cudaMalloc3D()对显存进行分配，可以使得线程满足合并访问的要求；其次，可以将数据的类型进行对齐处理；或者保证线程从 16 的整数倍的首地址开始进行访问，尽量使得每次线程读取的数据字长为 32bit。当对某些数据只会进行一次访问时，在满足合并访问的情况下可以使用 zerocopy。

（3）使用其他存储器，例如常量存储器或者纹理存储器，利用它们自带的缓存来提高某些应用的实际带宽。

2.2.3 资源优化

在程序运行过程中，调整每个线程所需处理数据的数量、共享存储器以及寄存器的使用量，使得程序获得更高的 SM 占用率。若线程处理的任务之间存在完全相同的部分，则可以将一小部分的线程用于计算公共部分，然后将公用的数据通过共享存储器广播给所有的线程。同时对线程块的大小、算法和指令进行调整来提高共享存储器的使用效率。

在减少寄存器使用的方面，可以在使用共享存储器进行变量存储；通过使用括号来明确地表示每个变量的生存周期；使用占用寄存器较小的等效指令来代替原有的指令等。

2.2.4 指令流优化

一条 warp 指令在 SM 中的执行主要过程为首先为 warp 中的每个线程读指令操作数，执行指令接着将计算结果写入每个线程中。因此，有效的指令吞吐量非常重要。因此可以采用以下方面来提高指令吞吐量：

（1）为了避免多个线程同时运行占用过多的时间，采用类似“if threadId<N”的方法对线程进行约束。

（2）在确定了不会出现不可接受的误差的前提下，使用 CUDA 算术指令集中的快速指令。

（3）在确保运算结果的正确性的情况下使用原子函数来实现复杂的算法。

（4）避免出现由于多余的同步操作而浪费时间的情况。