文章编号：150001

GPU 平台二维快速傅里叶变换算法实现及应用

张全[1,2,3]，鲍　华[1]，饶长辉[1]，彭真明[2]
(1. 中国科学院光电技术研究所，成都 610209；
2 电子科技大学光电信息学院，成都 610054;
3 中国科学院大学，北京 100049)

摘要：NVIDIA 在其 GPU 平台上开发的 FFT 库 CUFFT 经过几次升级，但在二维 FFT 实现上效率还有提升空间，而且对于特定不能与上下文的计算融合，导致多次对 Global memory 的访问。本文分析合并内存访问事务大小与占用率之间的关系，优化使用 GPU 存储器资源，对小数据量 2 次幂二维复数 FFT 在 GPU 上的实现进行改进，加速比最高达到 CUFFT 6.5 的 1.27 倍。利用实数 FFT 结果的共轭对称性，算法的效率比复数 FFT 算法运算量降低了 40%。最后将 FFT 的改进应用到光学传递函数（OTF）的计算中，采用 Kernel 融合的方法，使得 OTF 的计算效率比 CUFFT 计算方法提高了 1.5 倍。

关键词：快速傅里叶变换；CUDA；光学传递函数；图形处理器
中图分类号：　　**文献标志码**：A

Two-dimensional fast Fourier transform algorithm realization and application based on GPU

Quan Zhang[1, 2, 3], Hua Bao[1], Changhui Rao[1] , Zhenming Peng[2]
(1. *Institute of Optics and Electronics, Chinese Academy of Sciences, Chengdu* 610209;
2. School Optoelectronic Information，University of Electronic Science and Technology of China , Chengdu 610054;
3. University of Chinese Academy of Sciences, Beijing 100049)

Abstract: NVIDIA as the inventor of the GPU provides a library function CUFFT for computing FFT. After several generations update of CUFFT, there is still promotion space and it is not suit for kernel fusing on GPU to reduce the memory access and increase the instruction level parallelism (ILP). We develop our own custom GPU FFT implementation based on the well-known Cooley-Tukey algorithm. We analyze the relationship of coalesce memory access and occupancy of GPU and get the optimal configuration of thread block. The results show that the proposed method improved the computational efficiency by 1.27 times than CUFFT 6.5 for double complex data 512x512. And then it is used to the computation of OTF with kernel fusing strategy, it improved the efficiency of computation about 1.5 times than conventional method using CUFFT.

Key words: FFT; OTF; GPU; CUDA

0 引　言

快速傅里叶变换（Fast Fourier Transform, FFT），是离散傅里叶变换的快速算法。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。其中包括素因子算法（Prime Factor）、分裂基算法，混合基算法，专门针对长度为质数序列的 Rader 算法，以及用来分解质因数的 Good-Thomas 快速傅里叶变换算法和 Winograd 提出了一种素因子分解的快速算法等[1]。

在 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[2]。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[3]。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[4]。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[5][6]。随后，NVIDIA 把 Volkov 等人改进的算法加入到下一版本的 CUFFT 中。

本文针对 GPU 多核处理器架构，通过分析 FFT 的 Cooley-Tukey 算法框架，针对小数据量 2 的幂次方

收稿日期：2015-01-04；收到修改稿日期：2015−06−000
基金项目：国家自然科学基金（11178004）；中国科学院实验室创新基金（YJ14K018）
作者简介：张全(1985-)，男(汉族)，甘肃武威人。博士研究生，主要研究工作是 GPU 高性能计算，图像处理。E-mail: quanzhang100@126.com

http://www.gdgc.ac.cn

二维 FFT 在 GPU 上的实现进行分析，对列做一维 FFT 变换时，存在非合并内存访问的问题，使得访存效率下降。通过分析调整 Block 的大小，使得同时读入多列数据进行计算，尽量满足 128Byte 的 Cache line，将非合并内存访问的影响降到最低。最后将该算法应用到图像处理中常用的光学传递函数（OTF）的计算中，通过 Kernel 融合的方法提高了 OTF 的计算效率。

1 算法理论

1.1 Cooley-Tukey DFT算法框架[7]

离散傅里叶变换 DFT（Discrete Fourier Transform）将时域信号转换成频域信号。一维 DFT 计算公式如式(1)所示：

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

其中：k=0,1,…,N-1，$w_N^{nk} = \exp(-j2\pi \frac{nk}{N})$（$n$=0,1,2,...,$N$-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 表示成式(2)：

$$n = N_2 n_1 + n_2 \tag{2}$$

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式(3)

$$k = k_1 + N_1 k_2 \tag{3}$$

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 计算如式(4)所示：

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

Cooley-Tukey 算法主要包括两次索引变换和两次一维小点数 DFT，其计算分为如下 5 步。

Step1: 输入序列根据公式(2)做输入索引变换

Step2: 计算 N_2 个长度为 N_1 点的一维 DFT 运算（第一级变换）

Step3：对第一级变换结果乘以相应的旋转因子 $W_N^{n_2 k_1}$

Step4：计算 N_1 个长度为 N_2 点的一维 DFT 运算（第二级变换）

Step5: 对第二级变换结果按照公式(3)做输出索引变换

1.2 实数FFT算法

设 $x(n)$是 $2N$ 点的实序列，现将 $x(n)$分为偶数组 $x_1(n)$和奇数组 $x_2(n)$，如公式(5)所示

$$\begin{cases} x_1(n) = x(2n) \\ x_2(n) = x(2n+1) \end{cases} \quad n = 0,1,2,\ldots N-1 \tag{5}$$

然后将 $x_1(n)$及 $x_2(n)$组成一个复序列 $y(n)=x_1(n)+jx_2(n)$通过 N 点 FFT 运算可得到 $Y(k)=X_1(k)+jX_2(k)$，根据前面的讨论，得到公式(6)

$$\begin{cases} X_1(n) = \frac{1}{2}[Y(k) + Y^*(N-k)] \\ X_2(n) = -\frac{j}{2}[Y(k) - Y^*(N-k)] \end{cases} \quad k = 0,1,2,\ldots N-1 \tag{6}$$

为求 $2N$ 点 $x(n)$所对应的 $X(k)$，需求出 $X(k)$与 $X_1(k)$，$X_2(k)$的关系，具体见公式(7)

$$X(k) = \sum_{n=0}^{2N-1} x(n)W_{2N}^{nk} = \sum_{n=0}^{N-1} x(2n)W_N^{nk} + W_{2N}^{k} \sum_{n=0}^{N-1} x(2n+1)W_N^{nk} \tag{7}$$

而

$$\begin{cases} X_1(n) = \sum_{n=0}^{N-1} x_1(n)W_N^{nk} = \sum_{n=0}^{N-1} x(2n)W_N^{nk} \\ X_2(n) = \sum_{n=0}^{N-1} x_2(n)W_N^{nk} = \sum_{n=0}^{N-1} x(2n+1)W_N^{nk} \end{cases} \quad k = 0,1,\ldots N-1 \tag{8}$$

所以，得出 $X(k) = X_1(k) + W_{2N}^{nk} X_2(k)$。由 $x_1(n)$及 $x_2(n)$组成复序列，经 FFT 运算求得 $Y(k)$后，再利用共轭对称性求出 $X_1(k)$，$X_2(k)$，最后利用上式求出 $X(k)$，从而达到了用 N 点的 FFT 计算一个 $2N$ 点实数 DFT 的目的。

1.3 二维傅里叶变换

http://www.gdgc.ac.cn

二维离散傅里叶变换的定义为

$$X(k,l)=\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}x(n,m)\exp[-j2\pi(\frac{nk}{N}+\frac{lm}{M})]=\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}x(n,m)W_N^{kn}W_M^{lm} \tag{9}$$

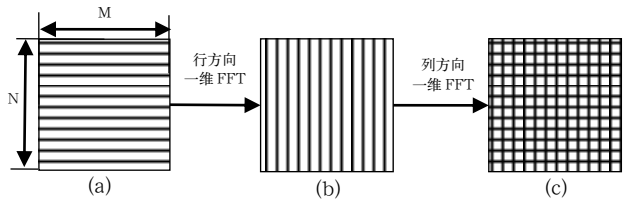

图 1. 二维复数傅里叶变换计算过程
Fig.1 Calculating procedure of 2-D FFT on complex number

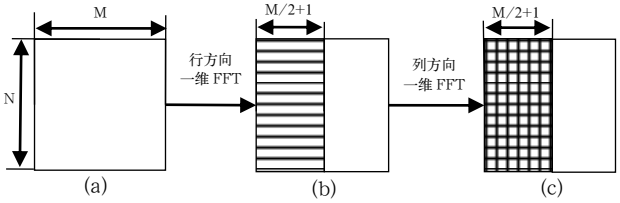

图 2. 二维实数傅里叶变换计算过程
Fig.2 Calculating procedure of 2-D FFT on real number

其中：$k\in[0,N-1]$，$l\in[0,M-1]$，$W_N^{kn}=\exp(-j2\pi\frac{nk}{N})$，$W_M^{lm}=\exp(-j2\pi\frac{lm}{N})$。二维离散傅里叶变换可通过两次一维离散傅里叶变换来实现，如图 1 所示，先对每个行 M 点一维 DFT（n 为行号，共 N 行），再对每个列 N 点一维 DFT（m 为列号，共 M 列）。这两次一维离散傅里叶变换都可以用快速算法实现。

由一维 DFT 性质知，当 $x(n)$为实序列时，满足复共轭对称性，所以行向量做一维 FFT 结果只保存 $M/2+1$ 列数据，如图 2(b)所示。然后对 $M/2+1$ 列作 N 点 DFT 得到图 2(c)结果。

1.4 GPU编程环境

在 CUDA 编程模型中，必须要有足够多的活动线程来隐藏访存延时，来提高计算性能。可以利用占用率来衡量活动线程数。但是程序优化的过程中不能满目的追求高占用率，高占用率不代表高效率[8]。高占用率虽然提高了并行度，但是由于高速存储器（寄存器，共享内存）资源有限，活动线程数量的增加导致每个线程分配的资源减少，从而导致了寄存器溢出又降低了计算效率，所以需要在占用率与资源使用之间做出一种折中。

此外，在 GPU 上进行程序优化主要遵循三个准则，其优先级由高到底。首先是 kernel 函数内部应尽量减少全局存储器的访问操作，同时全局存储器访问一定要满足合并访问；其次是设计应尽量使用共享内存（Share Memory, SM）来进行 block 内部通信，同时对于共享存储器的访问读写也要防止共享存储器的 bank 冲突；最后算法设计 block 内部应尽可能保持高密集度的寄存器计算，但同时也要协调好如何在片内寄存器资源有限的情况下选择最佳的 thread 数量[9]。

2　基于 GPU 平台的 FFT 工程实现

根据第二节中 Cooley-Tukey 算法思想，对于 2^m 点数的 FFT 可以由基-2、基-4、基-8、基-16 等多种组合实现，基数越高，数据重排的次数就越少，同时寄存器需求量也越大，在寄存器资源有限的情况下，高基数导致并行 thread 数受影响，降低了占用率。综合上述因素，对于 2^m 可以按顺序优先分解为基-16、基-8、基-4、基-2 的组合。

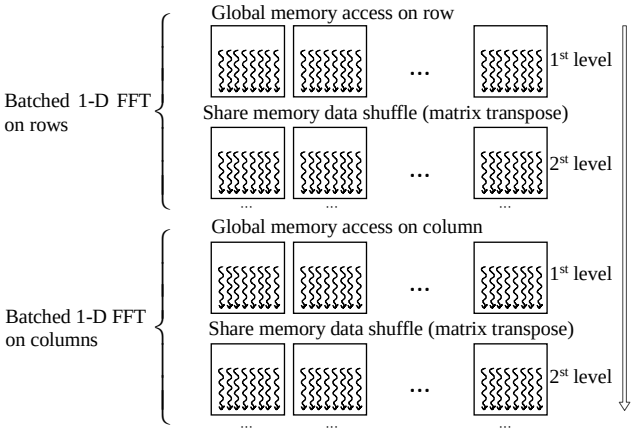

图 3. 二维傅里叶变换 GPU 实现过程
Fig.3 Implementation of 2-D FFT on GPU

对于一维复数 FFT，根据向量的大小，一维复数 FFT 的实现可以被分为三种情况：当向量长度为 8 或者 16 时，不需要进行线程之间的数据交换，其计算可以在寄存器中完成；当向量的长度为 64、256、512、1024、2048 和 4096 时，线程之间数据交换需要在 Share Memoy 中完成；当向量的长度超过 8192 时，必

http://www.gdgc.ac.cn

须通过两次 Kernel 调用，数据交换在 Global Memory 中完成。本文主要针对二维 FFT 计算进行分析，整个算法执行流程如图 3 所示，算法分为行方向一维傅里叶变换和列方向一维傅里叶变换。一维傅里叶变换又分为多级，相邻的级间需要同步并且通过 Share Memory 交换数据。

2.1 行方向一维FFT变换

对于小数据量的二维 FFT 算法，二维矩阵每一行数据可以放入 Share Memory，不需要通过 Global Memory 进行数据交换。假设数据长度为 N 的一维向量，将其排列成 $N_1 \times N_2$ 的二维矩阵。计算过程中首先选择合适的 N_2，使其能够在 Share Memory 中完成 FFT 计算。然后确定 $N_1=N/N_2$，如果 N_1 不能够在 Share Memory 中完成计算，则对 N_1 进一步分解为 $N_1=N_{11}\times N_{12}$，以此类推。另外一种解释就是相当于把长度为 N 的向量，重新排列为多维的矩阵，每一维的长度为 FFT 的一个固定基，其 FFT 计算称作整个 N 点 FFT 的一级。对 N 的每一次分解都要做一次矩阵转置。以 N=512 为例，首先，512 可以分解为 64×8，先做 64 行 8 点一维 FFT；其次，在 Share Memory 中进行数据转置，然后，再做 8 行 64 点一维 FFT，考虑到 64 点做 FFT 比较大，将其进一步分解为 8×8。通常 N 存在多种分解例如 512 还可以分解为 2×16×16，或者 4×8×16 不同的分解效率也会有差别，基越小运算量越大，尽量避免基-2 或者基-4 的分解。分解的级越多同步操作越多，所以尽量在满足共享存储要求的前提下采用大基。依据以上原则， N=128 的最优分解为 16×8 而非 4×4×8；N=256 的最优分解为 16×16，而非 8×8×4。

512 点 FFT 存储器访问如图 4 所示，完成一行 512 点一维 FFT 需要的线程数为 64。为了满足合并内存访问条件，512 个数据被分为 8 组，每组 64 个数据。每个线程从 8 组数中各取 1 个数据存入 8 个寄存器中。为提高 GPU 的占用率，Block 的大小取 64 的倍数，使一个 block 同时完成多行 512 点数据的一维 FFT。

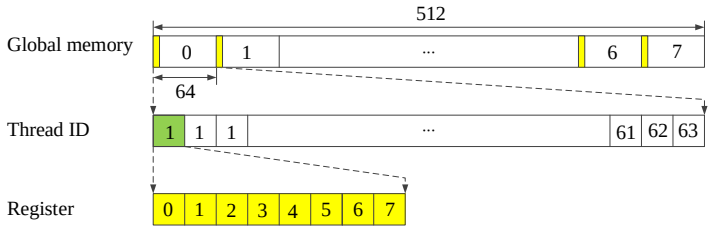

图 4. 512×512 行方向一维 FFT 变换全局存储器访问

Fig.4 Global memory access of 1-D FFT on row for array size of 512×512

2.2 列方向一维FFT变换

从 GPU 硬件的角度来看，GPU 上的内存延迟被不同线程束间的切换所隐藏。如果同一线程束的线程访问相邻内存位置，并且内存区域的开始位置是对齐的，这些访问请求会自动组合或者合并。计算能量 1.x 设备上，合并内存事务最小为 128 Byte。如果被合并线程访问的数据比较小，会导致内存带宽迅速下降。费米架构的设备支持 32 Byte 和 128 Byte 的合并内存事务。开普勒架构的设备支持 32 Byte、64 Byte 和 128 Byte 的合并内存事务。

列方向一维 FFT 变换与行方向一维 FFT 变换计算过程一样。区别在于对全局内存的访问方式不一样，在行方向一维 FFT 变换中线程访问 Global Memory 是满足合并内存访问的，而列方向一维 FFT 变换则不是。

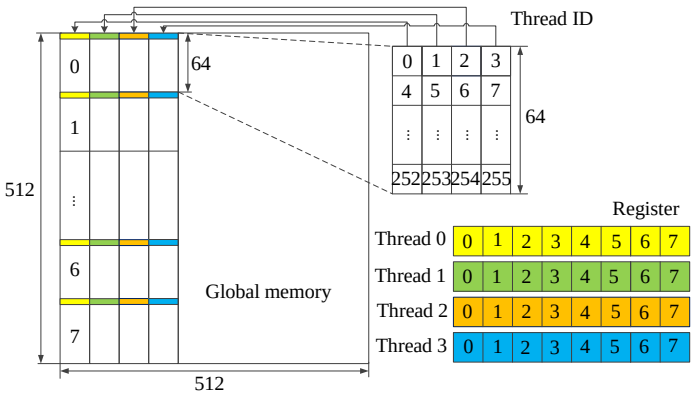

图 5. 512×512 列方向一维 FFT 变换全局存储器访问

Fig.5 Global memory access of batched 1-D FFT on column for array size of 512×512

http://www.gdgc.ac.cn

为了提高访存效率，需要对存储器访问方式进行调整。以 double 数据类型 512×512 二维矩阵列 FFT 变换为例，矩阵中每个数据占 16 Byte。本文实验采用的设备为计算能力为 3.5 的开普勒计算卡，其支持 32 Byte、64 Byte、128 Byte 三种缓存行模式，为了满足这三种内存合并模式，每个 Block 分别至少处理 2 列、4 列、8 列一维 FFT 计算，经过实验测试每个 Block 处理 4 列一维 FFT 时效率最佳。512×512 double 类型二维矩阵按列 FFT 变换存储器访问如图 5 所示，线程数排为 64×4 的二维矩阵，每一列线程访问 Global memory 中对应的一列数据，每个线程访问列方向间隔为 64 的 8 个数据。

2.3 FFT在OTF计算中应用[10]

光学成像系统在像面的复振幅可表示为：

$$h(x,y)=F^{-1}\{P(u,v)\} \tag{10}$$

式中，$F^{-1}\{\ \}$表示傅里叶逆变换，(u,v)和(x,y)分别为瞳面和像面坐标；$P(u,v)$为广义光瞳函数，可以表示为：

$$P(u,v)\ =\ p(u,v)\exp\{j[\phi(u,v)]\} \tag{11}$$

式中，j 为虚数单位，$\phi(u,v)$为波前相位，$p(u,v)$为孔径函数。对应成像系统的点扩散函数可表示为：

$$PSF(x,y)\ =\ h(x,y)\cdot h^{*}(x,y)\ =\mid F\{P(u,v)\}\mid^{2} \tag{12}$$

而光学传递函数定义为点扩散函数的傅里叶变换，即

$$OTF(\varepsilon,\eta)=F\{PSF(x,y)\} \tag{13}$$

由上所述，OTF 的计算分为 6 步：(1) 广义光瞳函数计算；(2) 对光瞳函数列方向批量一维 IFFT；(3) 行方向批量一维 IFFT；(4) 对取模平方运行得到点扩散函数；(5) 对点扩散函数列方向批量一维 FFT；(6) 行方向批量一维 FFT。在 GPU 上实现时，由于（1）和（2）计算过程中每个数据相互独立，可以合并，由一个 Kernel 完成计算；二维 FFT 计算中，交换行方向一维 FFT 和列方向一维 FFT 的计算次序不影响计算结果，交换（5）和（6）的次序，从而 Step 3~Step 5 行与行之间数据相互独立，三步可以合并由一个 Kernel 完成。整个合并过程使得 kernel 数由 6 个减小为 3 个。

3　实验结果分析

3.1 实验环境

本文实验所采用的硬件配置： Intel(R) core(TM) i7-3930k 主频 3.2GHz CPU，内存 8G， NVIDIA Telsa K20c GPU。软件配置： Windows 7 64-bit 操作系统，VS2010+CUDA6.5 编程环境。

3.2 FFT并行设计性能比较

二维 FFT 的 GPU 实现分两步执行，首先进行行方向批量一维 FFT，然后再列方向批量一维 FFT，行方向计算时 Global memory 满足合并内存访问，而列方向时不利于合并内存访问，根据不同的数据类型，尽量满足相邻线程访问事务大小的数据。对列方向一维 FFT 时，一次读入多列可以增加合并内存数据块，但是会消耗过多的共享内存和寄存器资源从而降低了占用率，所以需要在内存访问效率与 GPU 占用率之间做出折中的选择，针对不同的配置其占用率和内存访问效率都不一样，表 1 和表 2 分别针对 float2（float 型复数）和 double2（double 型复数）数据类型的列方向 FFT 在不同线程配置下做了测试。本实验采用的 Telsa K20c GPU，支持 32Byte、64Byte、128Byte 的合并内存事务。从实验数据我们知道占用率不代表高性能，也不是内存合并事务越大性能越高，需要在这两者之间进行折中。对于 256×256、512×512 的二维 FFT，64Byte 的合并内存事务使得占用率不会最低，同时内存访问效率也不是最低，是一种较优的配置。

表 1 Float2 型数据 Kernel 2 不同线程配置下性能比较

Table 1 Performance comparison of kernel2 for float2 under different thread configuration

	Block/ Grid	Share Memory	Occupancy	Coalesce memory access	Time/ us
128x128 float2	(256,1,1)/(4,1,1)	17KB	25%	32x8Byte	9.6
	(128,1,1)/(8,1,1)	**8.5KB**	**31.2%**	**16x8Byte**	**8.9**
	(64,1,1)/(16,1,1)	4.25KB	34.4%	8x8Byte	9.6
256x256 float2	(512,1,1)/(8,1,1)	34KB	25%	32x8Byte	15.2
	(256,1,1)/(16,1,1)	17KB	25%	16x8Byte	14.5
	(128,1,1)/(32,1,1)	**8.5KB**	**31.2%**	**8x8Byte**	**14.0**
512x512 float2	(1024,1,1)/(32,1,1)	36KB	50%	16x8Byte	56.7
	(512,1,1)/(64,1,1)	**18KB**	**50%**	**8x8Byte**	**49.7**
	(256,1,1)/(128,1,1)	9KB	62.5%	4x8Byte	54.8

表 2 Double2 型数据 Kernel 2 不同线程配置性能比较

Table 2 Performance comparison of kernel2 for double2 under different thread configuration

	Block/Grid	Share Memory	Occupancy	Coalesce memory access	Time/ us
128x128 double2	(128,1,1)/(8,1,1)	17KB	12.5%	16x16Byte	14.7
	(64,1,1)/(16,1,1)	**8.5KB**	**15.6%**	**8x16Byte**	**14.6**
	(32,1,1)/(32,1,1)	4.5KB	17.2%	4x16Byte	15.9
256x256 double2	(128,1,1)/(32,1,1)	17KB	12.5%	8x16Byte	28.0
	(64,1,1)/(64,1,1)	**8.5KB**	**15.6%**	**4x16Byte**	**21.7**
	(32,1,1)/(128,1,1)	4.25KB	17.2%	2x16Byte	27.9
512x512 double2	(512,1,1)/(64,1,1)	36KB	25%	8 x16Byte	97.8
	(256,1,1)/(128,1,1)	**18KB**	**25%**	**4 x16Byte**	**93.2**
	(128,1,1)/(256,1,1)	9KB	31.2%	2 x16Byte	104.6

对 CPU 平台下 FFTW 库函数，以及 GPU 平台下 CUFFT 和本文方法，分别做 1000 次二维 FFT 计算其平均值，三种方法都不考虑数据拷贝时间和内存分配时间，另外 FFTW 和 CUFFT 库函数时间只包括执行时间，不包括配置时间。表 3、表 4、表 5 分别为数据大小为 128×128、256×256、512×512 的二维 FFT 执行时间。从表中可以看出，数据较小时，CUFFT 与 FFTW 的加速比较小；数据量变大时，加速比随之增加，其范围在 2.5~53.7 之间。对于 128×128 大小的二维 FFT GPU 资源并没有充分利用，但是较之 FFTW 还是具有一定优势。本文的方法相对于 CUFFT 也有提高，双精度 512×512 大小二维复数 FFT 计算效率是 CUFFT 的 1.24 倍。利用实数 FFT 计算结果的共轭对称性，512×512 的二维双精度实数 FFT 计算效率是 CUFFT 的 1.36 倍。

表 3 128×128 二维 FFT 性能比较 (单位：微秒)

Table 3 Performance comparison of 2-D FFT for array of 128×128 (Unit:us)

		FFTW	CUFFT	Proposed method
2-D FFT of real number	Float	78.7	25.2	18.2 us
	Double	84.9	33.3	24.9 us
2-D FFT of complex number	Float2	75.3	19.1	18.9 us
	Double2	112.3	29.1	26.6 us

表 4 256×256 二维 FFT 性能比较 (单位：微秒)

Table 4 Performance comparison of 2-D FFT for array of 256×256 (Unit:us)

		FFTW	CUFFT	Proposed method
2-D FFT of real number	Float	375.5	31.7	23.4
	Double	461.6	48.8	34.1
2-D FFT of complex number	Float2	1114.6	30.2	27.6
	Double2	1290.4	48.1	42.4

表 5 512×512 二维 FFT 性能比较 (单位：微秒)

Table 5 Performance comparison of 2-D FFT for array of 512×512 (Unit: us)

		FFTW	CUFFT	Proposed method
2-D FFT of real number	Float	1599.5	68.1	52.4
	Double	1878.4	138.5	101.4
2-D FFT of complex number	Float2	4989.1	92.9	78.4
	Double2	7807.8	198.9	159.9

针对前面提到的 OTF 的计算，采用 CUFFT 库函数方法和本文 FFT 分别对其实现，其性能如图 6 所示，由于本文方法采用 Kernel 合并提高了 GPU 指令并行度，降低了 Global memory 的访问，对于 512×512 的双精度二维 OTF 计算效率是采用 CUFFT 方法的 1.51 倍。

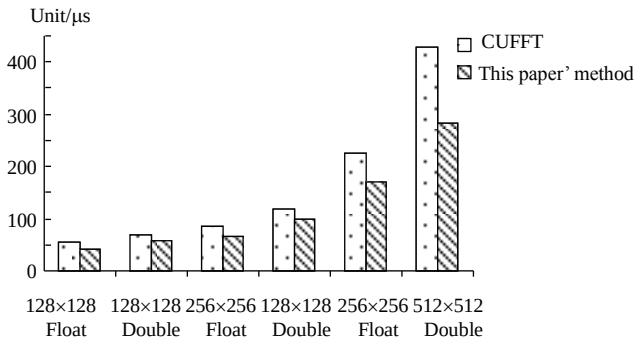

图 6 使用库函数与本文方法计算 OTF 执行时间比较

Fig.6 Comparison of execution time between library method and this paper' method

5 结论及展望

本文分析了 Cooley-Tukey FFT 计算框架，完成了基于 CUDA 编程模型的小数据量 2 的幂次方二维 FFT 在 GPU 上的实现，在合并内存访问与 GPU 占用率之间进行折中处理，实验结果表明，该方法效率是 CUFFT 的 1.24 倍。OTF 的 GPU 实现利用二维 FFT 计算特性，交换了一维行变换和列变换的次序，通过 Kernel 合并的方法，使 OTF 的计算效率相比 CUFFT 方法最多提高 1.5 倍。下一步工作主要针对大数据量的二维 FFT 并行化和非基-2 FFT 的实现。

参考文献：

[1] K.R. Rao，D.N Kim, J.J. Hwang. 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

[2] Moreland K, Angel E. The FFT on a GPU [C]//**Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware**, San Diego, California, July, 2003,112-119.

[3] 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 Li li, ZHANG Sheng bing, ZHANG Meng *et al*. High performance FFT computation based on CUDA [J]. **Application Research of computers**, 2011, 28(4):1556-1559.

[4] Volkov V, Kazian B. Fitting FFT onto the G80 architecture [Z]. University of California, 2008, E63(40):1-12

[5] 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.

[6] 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.

[7] 杨丽娟，张白桦，叶旭桢. 快速傅里叶变换 FFT 及其应用[J]. 光电工程, 2004, 31:1-3.
YANG Li-juan, ZHANG Bai-hua, YE Xu-zhen. Fast Fourier transform and its applications [J]. **Opto-Electronic Engineering**, 2004, 31:1-3.

[8] Volkov V. Better performance at lower occupancy [C]//**Proceedings of the GPU Technology Conference**, San Jose, California, September 20-23, 2010.

[9] Rob Farber. 高性能 CUDA 应用设计与开发：方法与最佳实践. [M]. 北京：机械工业出版社，2013:1-27

[10] Jim Schwiegerling. Relating wavefront error apodization, and the optical transfer function on-axis case [J] **Journal of the Optical Society of America A**(S1520-8532). 2014, **31**(11): 2476-2483