ISSN 1004-9037,CODEN SCYCE4
Journal of Data Acquisition and Processing Vol. 38,No. 6,Nov. 2023,pp. 1380－1391
DOI:10. 16337/j. 1004-9037. 2023. 06. 013

http://sjcj. nuaa. edu. cn
E-mail:sjcj@nuaa. edu. cn
Tel/Fax:＋86-025-84892742

基于GPU的长轨SAR实时成像算法

谭运馨[1]，黄海风[2]，赖　涛[2]，但琪洪[2]，欧鹏飞[2]

(1. 中山大学系统科学与工程学院,广州 510006; 2. 中山大学电子与通信工程学院,深圳518107)

摘　要：为了满足长轨道超高分辨W波段合成孔径雷达(Synthetic aperture radar,SAR)的快速成像需求,本文提出了一种基于图形处理器(Graphics processing unit,GPU)的ω-K实时成像算法,该算法采用并行架构和双流多线程的处理方式。默认流沿着物理原理的方向进行数据处理,首先对距离补偿、误差校正和补零等操作进行并行化处理,然后采用一层嵌套的插值方式,通过维持上下层的依赖关系和同步管理就能达到约30的加速比。阻塞流与默认流同时启动,生成默认流所需的参数和函数,并在执行前将其存入显存,极大地缩小了算法的运行时间,同时通过在默认流上设置事件以保持双流的同步并行执行。实验结果表明,算法总的加速比可达13左右,幅值和相位相对误差趋近0,不仅具有良好的实时性、聚焦性,还保持了良好的成像效果。

关键词：长轨道合成孔径雷达;图形处理器;实时成像;Stolt插值;ω-K算法

中图分类号：TN957.52　　**文献标志码**：A

GPU-Based Real-Time Imaging Algorithm for Long-Track SAR

TAN Yunxin[1], HUANG Haifeng[2], LAI Tao[2], DAN Qihong[2], OU Pengfei[2]

(1. School of Systems Science and Engineering, Sun Yat-Sen University, Guangzhou 510006, China;2. School of Electronics and Communcation Engineering, Sun Yat-Sen University, Shenzhen 518107, China)

Abstract: To meet the fast imaging requirements of long-orbit ultra-high resolution W-band synthetic aperture radar(SAR), this paper proposes a graphics processing unit(GPU)-based ω-K real-time imaging algorithm which adopts parallel architecture and double stream multithreading processing. The default stream processes data along the direction of the physical principle. Firstly, it parallelizes the rang compensation, error correction, zero filling and other operations, and then adopts one-layer nested interpolation method. By maintaining the upper and lower dependencies and synchronization management, it can achieve a speed ratio of about 30. The blocking stream starts at the same time as the default stream, generates the parameters and functions required by the default stream, and stores them into video memory before execution, which can greatly reduce the running time of the algorithm. Meanwhile, by setting events on the default stream, the two streams can be executed synchronously in parallel. Experimental results show that the total acceleration ratio of the algorithm can reach about 13, and the relative errors of amplitude and phase are close to 0, which not only has good real-time performance and focusing performance, but also maintains good imaging effect.

基金项目:广东省重点领域研发计划项目(2019B111101001);国家自然科学基金(62071499);深圳市科技计划项目(JCYJ20190807153416984)。

收稿日期:2022-11-10;**修订日期**:2023-02-20

Key words: long-track synthetic aperture radar(SAR); graphics processing unit(GPU); real-time imaging; Stolt interpolation; ω-K algorithm

引 言

长轨道高分辨率成像雷达可用于机场异物检测、形变监测和科学研究等领域[1-2]。本文在前期开发完成的W波段厘米级分辨率长轨道合成孔径雷达(Synthetic aperture radar,SAR)和子孔径成像算法并行化的基础上,进一步探讨了基于图形处理器(Graphics processing unit,GPU)的实时成像方法。

快速实时成像算法是一个热门的研究领域,学者们的研究目标主要是传统机载或者星载体制的雷达。常见的成像算法有距离多普勒算法(Range-Doppler algorithm, RDA)、CSA(Chirp scaling algorithm)算法、ωKA等,RDA和CSA具有高效、成熟且易于实现等特点,但这两种算法由于采用了近似泰勒展开,因此难以适应大带宽、大波束角的成像场景。ωKA在二维频域通过Stolt插值来矫正了距离方位耦合与距离向时间和方位向频率的依赖关系,避免了RDA和CSA的缺陷。因此适用于轨道SAR实时成像系统[3-4]。

ω-K算法处理雷达图像需要较高的内存和处理时间。Sahay等[5]提出了一种改进时域参考函数的方法,缓解了SAR图像生成处理时间长和高内存的问题。参考函数的生成及运算要求更高的计算量,所以快速傅里叶变换(Fast Fourier transform, FFT)及快速傅里叶逆变换(Inverse fast Fourier transform, IFFT)需要更快的转换时间[6]。所以仅对参考函数进行改进,对整个系统的实时性来说远远不够。文献[7]在现场可编程逻辑门阵列(Field programmable gate array, FPGA)上实现了对ω-K算法的优化,其对ω-K算法中的Stolt插值部分采用数据循环存储,然后进行插值计算。与文献[5]一样,他们仅对ω-K算法中的插值部分进行优化,并没有对整个算法进行优化处理。当处理的数据量变大,分辨率变高,不只是插值部分的工作量变大,整个ω-K算法的复杂度也会变大,同时采用的FPGA平台的计算能力和并行化程度并不高。Zhu等[8]在FPGA上实现了极坐标格式算法(Polar format algorithm, PFA)实时处理,该方法相比CPU处理速度有所提升,但是依然无法满足实时成像的要求,并且FPGA时序难以规划、程序调试周期长、算法实现复杂。Zou等[9]在一种数学框架上提出了一种高效准确地基合成孔径雷达(Ground-based synthetic aperture radar, GB-SAR)算法,避免了庞大的计算量;但工程实现难度较大,应用于实际工程当中也未必能达到理想的效果。所以从硬件上进行加速方面的研究成了最佳选择。

GPU强大的处理能力、高度并行性及高显存带宽为ω-K算法提供了具有发展前景的新型运算平台[10]。因此众多学者在GPU平台上对SAR成像算法进行并行化处理[11-13]。本文提出了一种基于GPU平台的实时成像算法。与其他并行算法不同,本算法依据各个部分的独立性和并行性,或对算法进行改进使各个部分之间体现最大限度的独立性和并行性以进行整体加速。其中对插值部分提出了3种优化方案并进行对比分析,其中一层嵌套很好地适配了实时成像系统,将大数据量多重循环的插值计算优化成更细粒度的并发运算。利用双流方式将相关参数的生成与相关函数的执行并行进行,避免了设备端与主机端频繁的内存访问,将参数和函数的执行隐藏于默认流的执行当中,极大地缩减了实时成像算法的运行时间。

1 ω-K并行化优化处理策略

在速度恒定的条件下,ω-K算法具有在宽孔径或大斜视角范围内校正沿距离向的距离徙动变化的

能力[14]。本文提出了在GPU端实现的ω-K实时成像算法，如图1所示。图2为本算法雷达系统工作示意图，表1为该雷达系统的参数设置。

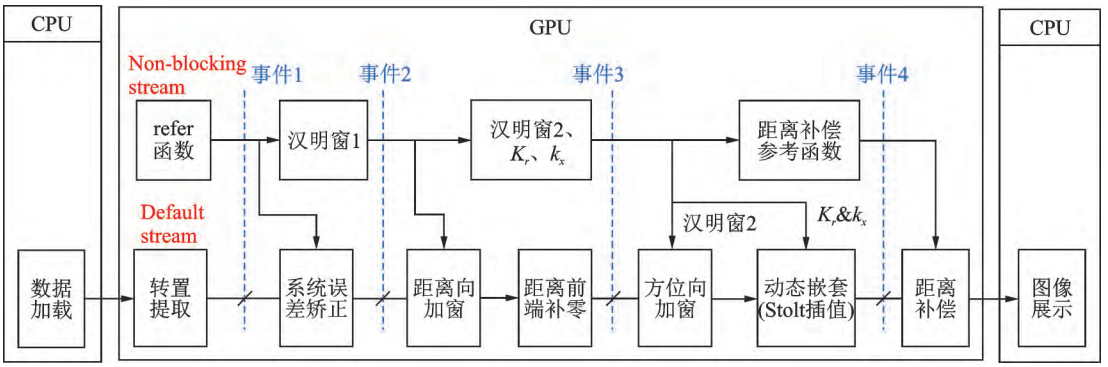

图1 长轨道SAR ω-K实时成像算法实现

Fig.1 Realization of ω-K real-time imaging algorithm for long-orbit SAR

1.1 基于非阻塞流的参数生成

在统一计算设备架构(Compute unified device architecture, CUDA)中，同一流中的操作有严格的执行顺序，而在不同流中的执行顺序不受限制[15]。如图1所示，创建一条与默认流相并行的非阻塞流。在相关函数执行之前，在非阻塞流上提前将所需函数(refer函数等)、变量(K_r, k_x)生成并保存至显存中。图3所示为采用默认流、阻塞流和非阻塞流执行生成refer函数的时间坐标。

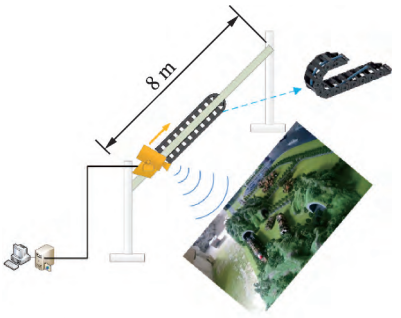

图2 雷达系统工作示意图

Fig.2 Radar system working diagram

图3中3个条形柱的宽度为refer函数生成的时间，为18.7 ms左右。由图3可知相比于串行的默认流，并行执行的非阻塞流要提前1 s运行；即使是阻塞流也要比默认流提前0.94 s；同时非阻塞流比阻塞流节省0.072 3 s的运行时间。如图1所示，refer函数、汉明窗1、汉明窗2等函数位于与默认流相并行的非阻塞流。两条流是并行执行，同时默认流上设有4个事件，他们会阻止默认流中的相关操作的执行，直到所需函数和变量已经提前保存在显存当中。

表1 回波数据相关参数

Table 1 Related parameters of echo data

系统参数	数值
载波频率/GHz	95
距离向带宽/GHz	6
脉冲重复频率/Hz	400
采样频率f_s/MHz	2.5
回波信号尺寸($N_a\times N_r$)	4 610×8 000
平台速度V_a/(m•s^{-1})	0.302 5

图4为非阻塞流和CPU上各个生成函数的运行时间，不难发现：在CPU上生成汉明窗的时间成本要更少一些。因为汉明窗生成的工作量很小，以至于内核的启动时间远大于核函数的运行时间。但考虑到CPU与GPU之间数据传输要花更长的时间，汉明窗的生成不能放到CPU端上执行。因为远距离目标的回波比较弱，在模拟域的补偿较少，所以要在数字域进行补偿，补偿参数为距离的1.5次方。如图4所示，距离补偿函数的生成，GPU可以加速4.18倍。采用双流并行的方式，从GPU端来说，总时长节省了61.34 ms；从CPU端来说，总时长可以节省364.395 ms。

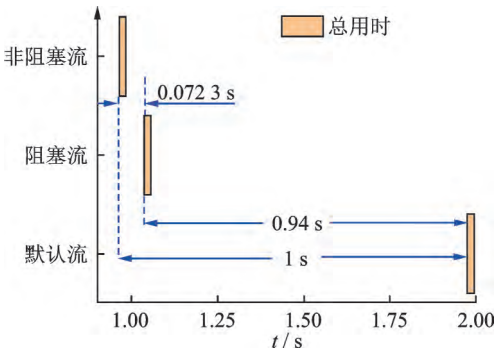

图3 默认流、阻塞流和非阻塞流生成refer函数的时间坐标

Fig.3 Time coordinate of refer function in default stream, blocking stream and nonblocking stream

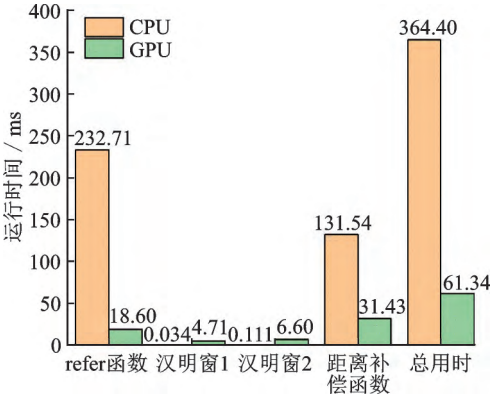

图4 非阻塞流运行各个参数函数的时间

Fig.4 Running time of each parameter generating function on nonblocking stream

1.2 方位向、距离向抽取二维频域并行优化策略

待处理的数据存在于CPU的内存中,需要将其读取并保存到GPU的显存当中;由于成像距离比较短,方位向和距离向过采样,所以要进行方位向和距离向抽取以降低采样率。为了进行插值前的数据处理,数据的读取、方位向和距离向抽取后需要进行FFT。同时ω-K算法需要在二维频域对信号进行处理。而在数据量较大的时候FFT计算量非常大,通常可以利用旋转因子的周期性、对称性等性质进行化简。对FFT/IFFT的快速计算进行相关的研究很早就开始进行,Cooley和Tukey[16]将离散傅里叶变换(Discrete Fourier transform, DFT)的算法复杂度从$O(N\cdot\lg N)$降低为$O(N^2)$。GPU因其计算能力强,众多学者尝试在GPU上进行FFT的快速实现[17-18]。本文将从CPU端和GPU端上分析FFT的加速方法。

表2 CPU端与GPU端加速效果对比

Table 2 Comparison of CPU and GPU acceleration effects

点数	CPU FFTW库耗时/ms	CUDA CUFFT库耗时/ms	加速比
1e+6	30.905 4	17.58	1.76
3e+6	235.752	32.013 3	7.36
5e+6	409.01	37.379 1	10.94
7e+6	629.255	49.980 4	12.60
1e+7	836.846	55.084	15.19
3e+7	2 736.72	101.207	27.04
8e+7	7 451.24	249.291	29.89
3e+8	33 304.8	224.497	148.35
5e+8	124 228	540.423	229.87

Frigo和Johnson[19]开发了一种在CPU端使用的快速计算离散傅里叶变换的标准C语言程序集FFTW。FFTW比CPU端其他傅里叶变换要更快、更高效,因而被广泛应用于如Intel的数学库和Scilib等软件上。GPU端的CDUA库提供了一系列库函数来提高开发人员的开发效率。其中CUFFT库提供了一个优化的快速傅里叶变换,其加速效果不亚于FFTW,并且避免了主机和设备之间频繁相互访问的时间消耗。

表3和图5对CPU和GPU上两种不同的FFT加速库进行了对比。可以发现随着点数的增加,CPU端FFT/IFFT时间也会随之增加,两者之间的加速差距也会明显变大。由图5右图的加速比曲线可以看出,随着点数增高,CUFFT的加速效果要优于FFTW。因此,在FFT加速方面,GPU端的加速

能力要远超FFTW。并且若从CPU端进行加速,需要将待处理的数据传到CPU,然后再将FFT/IFFT后的结果再传回GPU,导致多出了两次数据传输的时间消耗,不能实现处理过程中的结果暂存,且违背了GPU程序编程的3条通用法则中的第1和第3法则[20]。

表3 两种平台插值前的数据处理运行时间

Table 3 Data processing running time before interpolation of two platforms

项目	项目	数据量	GPU耗时/ms	CPU耗时/ms	加速比
数据读取	列FFT	4 608×8 000	50.954	1 221.210	23.97
	列IFFT	2 304×8 000	30.205	731.612	24.22
方位向抽取	列FFT	8 000×2 304	29.287	1 096.030	37.42
	列IFFT	8 000×2 304	37.277	1 099.490	29.50
	方位抽取	4 000×2 304	4.306	113.867	26.44
距离向抽取	行FFT	4 000×2 304	6.526	229.235	35.13
	行IFFT	4 000×2 304	12.555	249.797	19.90
	距离抽取	4 000×1 152	2.174	57.371	26.39
二维频域	列FFT	4 000×1 728	10.774	204.691	19.00
	频谱搬移	4 000×1 728	6.043	139.166	23.03

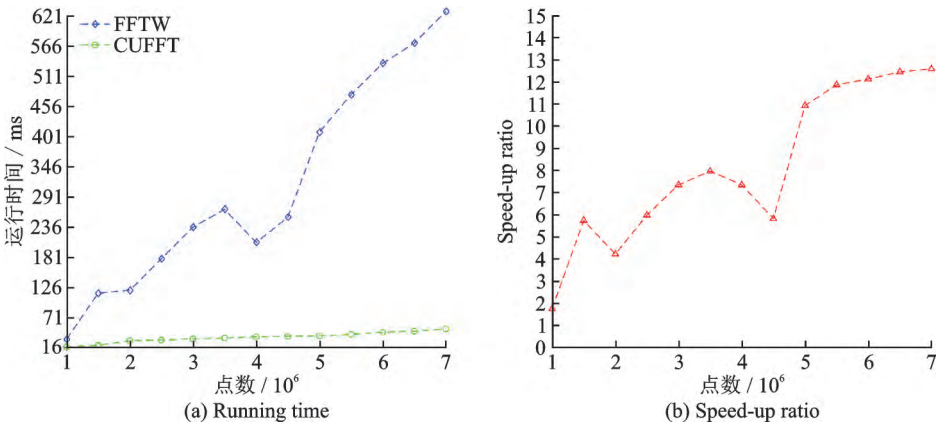

图5 FFTW和CUFFT运行时间对比及加速比

Fig.5 Speed-up ratio and comparison of running time for FFTW and CUFFT

CUDA库需要调用设备变量,普通主机函数无法直接访问设备变量,需要先将数据从显存传回CPU内存中。所以本文通过CUFFT库实现了距离向和方位向上的FFT和IFFT,并将其封装成一个宏函数可供主机端调用,因为宏函数的形参可以为设备变量或者主机变量。

表3为文件读取,距离向和方位向抽取、二维频域转换的时间花销对比。可以发现,在GPU上进行列FFT的时间要比行FFT时间要长一些。因为CUFFT采用行优先的数组存储方式,这会导致设计列FFT/IFFT宏函数时,要在FFT前后进行两次转置操作。行和列的逆傅里叶变换的结果需要除以点数进行矫正,故其运行时间要比正傅里叶变换的时间长。由表3可知从GPU端进行加速的效果更明显,加速比可达数十倍。因为本算法在离散化之前距离向就已经完成解线性调频,所以距离向已经处在频域。故只要进行方位向傅里叶变换即可进入二维频域。

1.3 系统误差矫正及距离频域前端补零并行优化策略

1.3.1 系统误差矫正

天线相位中心到目标的距离为$R_{sys}=0.08$ m,该距离使得目标回波在空中和射频电缆中的传输存在时间延迟T_{sys}。根据线性调频中的时频关系,频谱会产生一个F_{sys}的频移。这是一个与系统有关的固定误差,不会随数据的变化而更新。

$$T_{sys}=2\times R_{sys}/c \tag{1}$$

$$F_{sys}=K_r\times T_{sys} \tag{2}$$

式中:K_r为距离向调频率,c为光速。想要补偿频谱中产生的相位差,需要乘以如式(3)所示的补偿参考函数

$$H_{refer}=\exp(-j2\pi f_{sys}\tau) \tag{3}$$

式中:H_{refer}对应图1中的非阻塞流上生成的refer函数,τ为距离向快时间。图1中设置的事件1引入了默认流和非阻塞流之间的依赖关系;默认流在运行到事件1时,若非阻塞流还未运行到事件1,则会阻塞默认流的继续执行直到非阻塞流完成refer函数的生成。因此采用双流并行执行的方式,保证进行系统误差矫正前,refer参考函数已经提前完成生成并保存至GPU的显存之中,从而将生成参数的时间隐藏于默认流之中。如图6所示,refer生成函数(get_t)的运行时间几乎可以完全隐藏于CPU到GPU的数据传输(Memcpy HtoD)当中。

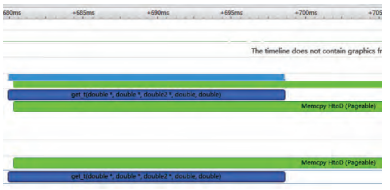

图6 refer函数的双流处理

Fig.6 Operation of refer function in two streams

表4所示为系统误差矫正的时长,由于补偿函数——refer函数的生成是相位补偿前在非阻塞流上生成的,所以GPU上系统误差矫正总时长仅包含相位补偿过程的时长6.911 ms。由表4可以看出,在GPU上进行相位补偿的加速比是212.73。整个系统误差矫正中,GPU加速比为246.40。

表4 GPU和CPU上系统误差矫正的运行时间

Table 4 Running time of system error correction on GPU and CPU ms

项目	参数生成	相位补偿	总时长
CPU	232.712	1 470.18	1 702.892
GPU	18.595(非阻塞流)	6.911(默认流)	6.911
加速比	12.51	212.73	246.40

1.3.2 距离前端补零、方位向、距离向加窗

插值前的数据是非均匀数据,无法直接与回波数据对应,所以无法直接进行逆傅里叶变换。因此需要通过Stolt插值将非均匀的数据重采样为均匀的数据。进行Stolt插值后的数据前端会向前突起,突起的部分超出了源数据的范围,如图7右图阴影部分。故在Stolt插值前,需要对源数据的前端进行补零,扩大源数据的范围以覆盖突起的部分。同时为了降低旁瓣对主瓣的影响,防止频谱信号能量泄露,提高精度,需要在方位向和距离向进行加窗聚焦。

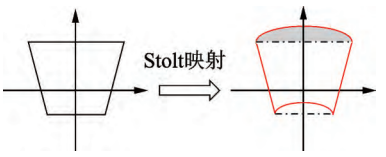

图7 Stolt映射

Fig.7 Stolt mapping

如表5所示为GPU和CPU端加窗和补零的运行时间。在GPU端上,距离前端补零和距离向加窗

表5 距离前端补零、方位向和距离向加窗对比结果

Table 5 Comparison results of zero padding at the distance front end, azimuth and range windowing

平台	项目	结果
CPU	距离前端补零	82.217 2 ms
	距离向加窗	261.874 ms
	方位向加窗	388.326
GPU	距离向补零加窗	2.901 ms
	加速比	118.61
	方位向加窗	2.372 ms
	加速比	163.71

同时进行。在GPU端进行距离向补零加窗的加速比为118.61,方位向加窗的加速比为163.71。由1.1小节分析可知,在窗函数的生成上,CPU的时间明显快于GPU时间。由于考虑到CPU和GPU之间数据传输的成本,不能在CPU端生成窗函数。同时GPU上窗函数生成时间可以隐藏于默认流之中,即GPU上生成窗函数是几乎没有时间成本。同时默认流中设置的事件2和事件3确保了窗函数的提前生成。

1.4 基于动态并行的插值运行

Stolt插值采用的插值因子通过距离向频率轴的映射和弯曲来完成[21],Stlot插值完成了残余距离徙动矫正、残余二次距离压缩和残余方位压缩。由于Stolt插值对成像质量的影响较大,所以对于Stolt插值要求精确计算,这就导致Stolt插值计算量大[22]。本算法对比了3种Stolt插值的并行化方法。

1.4.1 逐行进行插值(零层嵌套)

插值本身包含两重循环,在加上回波数据的一重循环,共3重循环。在对子孔径成像算法进行并行化处理中,采用从已知数据或者估计数据中的一个方向进行插值。已知数据为真实时域,是一维数据,而估计数据即虚拟时域为二维数据。如果从虚拟时域方向进行插值,可以同时对整个二维矩阵进行遍历、比较和插值,所以只要虚拟时域点数不超出GPU所限制的线程数,运行时间理论上会保持不变。在ω-K算法Stolt插值中,已知数据是二维数据矩阵$N_{a1}\times N_{r1}$(4 000×1 728),而估计数据是一维数据数组,且插值的比较范围是已知数据矩阵中的每一行。如果整个已知数据同时进行遍历、比较和插值,则在比较范围以外的其他行,有可能存在更接近于估计数据的临近点,导致该点进行错误的插值,进而导致整个估计数据进行错误的插值。所以ω-k算法中Stolt插值部分不能进行二维数据同时插值,但是可逐行进行插值。为了与一层动态嵌套和二层动态嵌套进行对比解释,本文将逐行进行插值称为零层动态嵌套。

如图8所示,零层嵌套对二维频域数据上的单个点进行插值运算,一个估计数据与N_{r1}个已知数据同时进行比较,筛选出临近点,进行线性拟合。二维数据总共有$N_{a1}\times N_{r1}$个点,故总共需要进行$N_{a1}\times N_{r1}$次点插值运算。

1.4.2 动态并行进行插值

动态并行允许GPU端在已有的GPU内核中直接创建和同步新的GPU内核[15]。实现核函数中嵌套其他核函数。插值部分的工作量在串行条件下以3重循环进行点对点的遍历、比较和插值。所以这就限制了其本身最多进行深度为3的嵌套。

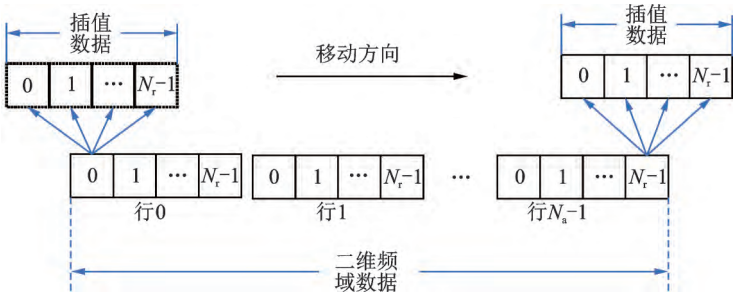

图8 零层嵌套插值

Fig.8 Zero-layer nested nested interpolation

(1)一层嵌套动态并行

图9为深度为2的嵌套示意图,父核函数为nestinterp1,子核函数为interp2_GPU。其中子核函数进行每一行的插值运算,父核函数在N_{a1}个线程上即②点处同时启动N_{a1}个子核函数,同时完成N_{a1}行插值运算。当运行至③处时,子核函数已经完成每一行的插值运算。

本实验平台为NVIDIA GeForce GTX 1060 6 GB,其多流处理器(Streaming multiprocessor,SM)的数量为10,流处理器(Streaming processor, SP)数量为128。估计数据的大小为$N_{a1}\times N_{r1}$。CUDA程序中要有足够的并行性,块的数量要多于SM的数量。父核函数和子核函数均采用一维网格。综上所述,父核函数和子核函数的线程块的中线程应满足

$$\lfloor(4\,000+\text{blocksize1}-1)/\text{blocksize1}\rfloor\geqslant 10 \tag{4}$$

$$\lfloor(1\,728+\text{blocksize2}-1)/\text{blocksize2}\rfloor\geqslant 10 \tag{5}$$

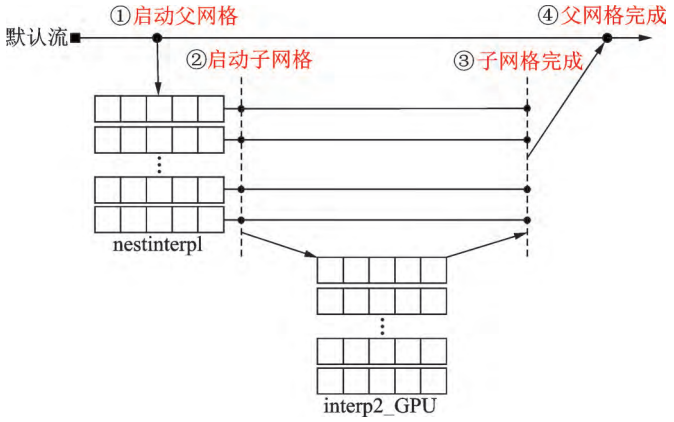

图9 一层嵌套工作原理图

Fig.9 One-layer nested working principle diagram

可以求出设备并行性的临界值为blocksize1≈444,blocksize2≈192。线程块尺寸越小于444和192,单位SM内的线程块的数量就越多。同时线程块的最大数量也受到硬件资源的限制,所以高占用率并不意味着更高的性能,其他因素同样也限制GPU的性能。为了避免线程块太小,每个块至少包含128个线程。为了能极大地提高加载效率,同时要保持每一个线程块中线程数量是线程束大小(32个线程)的倍数。所以从blocksize1≈444,blocksize2≈192的配置附近寻找最佳执行配置,如表6所示。

表 6 执行配置运行时间

Table 6 Time of execution configuration point operation ms

父核	352(85.94%)	384(93.75%)	416(81.25%)	448(87.50%)	480(93.75%)
子核 128(75%)	1 580.837	969.257	1 219.707	770.417	1 648.537
子核 160(70.31%)	1 998.487	1 588.957	1 042.057	804.967	1 778.747
子核 192(75%)	2 357.737	2 068.197	1 785.447	919.517	1 919.007

由表 6 可以看出，最佳执行配置是 blocksize1=448，blocksize=128。在 blocksize1=352，blocksize=128 的配置中，线程块数量最多，并行性也是最好的；但其占有率(85.94% 和 75%)却不是最高的。同理，最佳配置 blocksize1=448，blocksize=128 并行性不是最好的，占有率(87.50% 和 75%)也不是最高的。占有率最高的执行配置 blocksize1=384，blocksize=128 却不是最佳配置。所以没有一个单独的指标能直接优化性能，需要从多个相关指标和 GPU 硬件资源之间寻找一个恰当的平衡来实现最佳总体性能。占用率是每一个 SM 中活跃的线程束数量和最大线程束数量之间的比值，即

$$\text{占用率} = \frac{\text{活跃线程束数量}}{\text{最大线程束数量}} \tag{6}$$

NVIDIA GeForce GTX 1060 6 GB 上的每一个 SM 上的最大线程数量为 2 048，所以最大线程束数量为 2 048/32=64。由此可以算出最佳配置的父核函数和子核函数活跃线程束数量分别为

$$\text{父核活跃线程束数量} = 64 \times 0.875 = 56 \tag{7}$$

$$\text{子核活跃线程束数量} = 64 \times 0.75 = 48 \tag{8}$$

(2)双层嵌套动态并行

由(1)可知，子核函数的最佳线程块大小为 128，父核最佳线程块大小为 448。父网格 interp_lay3 即最外层循环的大小为 N_{a1}，中间循环 interp_lay2 即第 1 层嵌套和内层循环 interp_lay1 即第 2 层嵌套的大小为 N_{r1}。所以第 1、2 层嵌套的块大小设置为 128，父网格设置为 448。父网格调用第 1 层嵌套，然后第 1 层嵌套调用第 2 层嵌套，如图 10 所示，所以可以进行深度为 3 的嵌套。

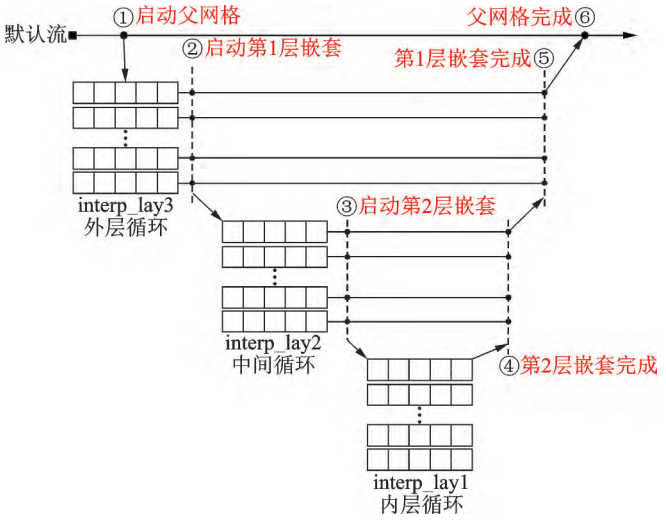

图 10 2层嵌套工作原理图

Fig.10 Two-layer nested working principle diagram

由表7可知,0层嵌套和1层嵌套都具有加速效果,而2层嵌套运行时间反而比CPU端的更慢。实际上,实时成像算法需要全天候、全天时对地质体进行连续监测,由此产生的数据量是巨大的。在新的一级网格中启动另一个网格需要额外消耗内存资源和时间花销,同时每一个上层嵌套和下层嵌套之间的同步管理又要耗费大量的设备内存和时间。在本算法的2层嵌套中,在父网格的所有线程中开辟一个新的网格,新网格的所有线程又启动另一个新的网格,需要额外开辟新的内存资源来启动新的网格和保持新旧网格之间的同步,同时嵌套深度越深,嵌套的各层之间的同步需求就更加频繁,需要维持的依赖关系也会变得更复杂。如图9、10所示,1层嵌套之间的同步管理只有上、下两层之间,只需维持上层和下层的依赖关系;而2层嵌套需要保持上中、上下、中下、上中下的同步管理和依赖关系,理论上同步管理的工作量增加了4倍以上,需要维持的依赖关系也增加到了4个。由表7可知2层以上嵌套的加速效果已经不足以抵消各层之间同步和维持各个层之间的依赖关系所带来的时间消耗。所以对于实时成像系统来说,1层嵌套的加速效果比零层嵌套和双层嵌套的加速效果要好得多。加速比可达30倍左右。

表7 采用0、1、2层嵌套的算法总用时

Table 7 Total time of using zero-layer, one-layer and two-layer nested algorithms

项目	0层嵌套	1层嵌套	2层嵌套	CPU插值
运行时间/s	13.038	0.770	319.746	22.931
加速比	1.76	29.78	0.072	

2 结果分析

表8为CPU端和GPU实时成像算法的运行时间,运行大小为4 610×8 000的实测数据,其中CPU端的运行时间为30.954 s,GPU的实时成像算法运行时间为2.440 s,加速比约13倍。图11是CPU与GPU处理结果的幅值和相位相对误差帕累托图。不难发现CPU端和GPU端处理结果的相位和幅值相对误差非常小,其中幅值相对误差有63.18%的点分布在$(10^{-13},10^{-12}]$的范围内,幅值相对误差分布在$(10^{-14},10^{-11}]$的点占99.28%。同理相位相对误差分布在$(10^{-14},10^{-12}]$范围内的点数占了85.99%,而分布在$(10^{-14},10^{-11}]$的点占了98.43%。因此GPU端的实时成像算法相位和幅值相对误差非常小,几乎为0。该误差主要来源于GPU端和CPU端舍入规则和数据截断方式的不同。以上分析说明:为了最大限度地体现GPU算法的并行性和独立性所作的并行化处理并没有破坏算法的物理原

表8 CPU端与GPU端实时成像算法运行时间

Table 8 CPU and GPU real-time imaging algorithm running time

CPU端/s	GPU端/s	加速比
30.954	2.440	12.69

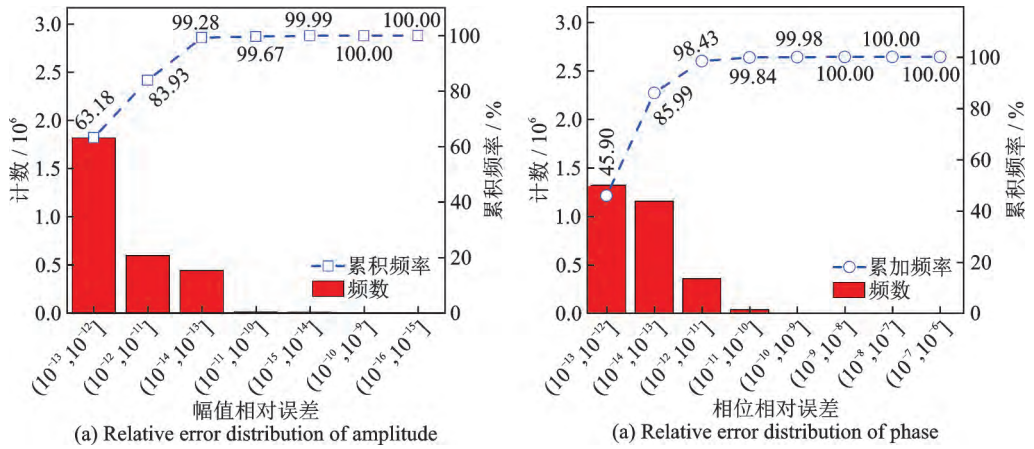

图11 相位和幅值相对误差分布

Fig.11 Relative error distribution of amplitude and phase

理,同时本算法的成像效果不会因为追求实时性而有所降低。图12为CPU成像算法和GPU成像算法的成像效果显示。不难发现,GPU实时成像算法具有良好的聚焦性、实时性和良好的成像效果。

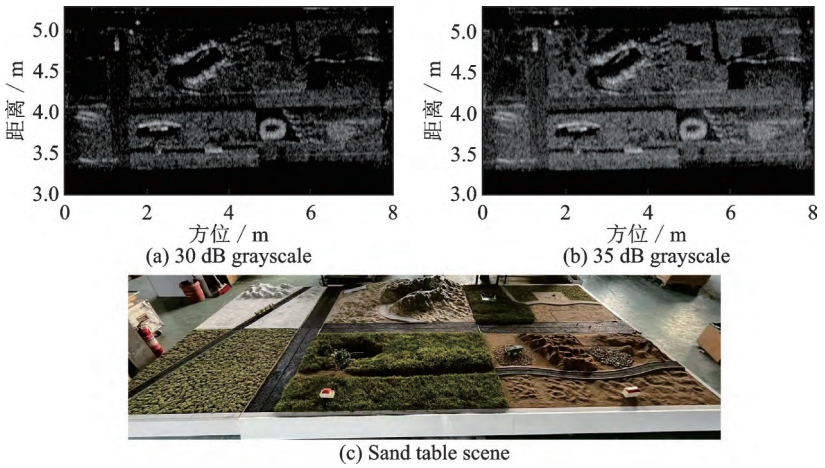

图12 GPU端和CPU端成像结果

Fig.12 GPU and CPU imaging results

3 结束语

本文提出了一种基于GPU的轨道SAR ω-K实时成像算法。本文对距离补偿、误差校正和补零等操作进行了优化处理。通过双流的并行执行方式,将系统误差、方位向和距离向加窗、插值和距离补偿所需的参考函数和窗函数放在与默认流相并行的流中执行。将CPU端需要364.395 ms完成的时间花销隐藏于默认流的各个操作和数据传输当中,大大缩小了系统误差矫正等操作的运行时间,同时避免了主机和设备之间的交互。通过CUFFT设计了行、列FFT和IFFT的宏函数,实验证明该宏函数比CPU端的加速库快数10倍。本算法提出了0层、1层和2层嵌套的插值方案,经过分析对比,发现零层和单层嵌套的插值方案均有加速效果。其中单层的嵌套加速比可达30倍左右。因为层数的增加,同步管理的需求变多以及依赖关系变得更为复杂,导致双层加速效果低于CPU端,所以双层的插值方式并不适合于实时成像系统。基于大小4 610×8 000的实测数据表明,本算法可以加速13倍左右,并且幅值和相位相对误差为百亿分之一,趋近于零。同时本算法具有良好的实时性、聚焦性和成像效果。

参考文献:

[1] SERRANO-JUAN A, VÁZQUEZ-SUÑÈ E, MONSERRAT O, et al. Gb-SAR interferometry displacement measurements during dewatering in construction works: Case of La Sagrera railway station in Barcelona, Spain[J]. Engineering Geology, 2016, 205: 104-115.

[2] LIU Xiangzeng, ZHENG Tian, QIANG Lu, et al. A new affine invariant descriptor framework in shearlets domain for SAR image multiscale registration[J]. AEU-International Journal of Electronics and Communications, 2013, 67(9): 743-753.

[3] ZHANG Lei, SHENG Jialian, XING Mengdao, et al. Wavenumber-domain autofocusing for highly squinted UAV SAR imagery[J]. IEEE Sensors Journal, 2011, 12(5): 1574-1588.

[4] XU Gang, XING Mengdao, ZHANG Lei, et al. Robust autofocusing approach for highly squinted SAR imagery using the extended wavenumber algorithm[J]. IEEE Transactions on Geoscience and Remote Sensing, 2013, 51(10): 5031-5046.

[5] SAHAY Peeyush, DHAVALKUMAR VAIDYA B, KADALI L K. Squint SAR algorithm for real-time SAR imaging[C]// Proceedings of 2022 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT). Bangalore, India: IEEE, 2022: 1-4.

[6] SAHAY P, VAIDYA DHAVALKUMAR B, DEB D, et al. High-squint SAR data generation and processing with validation on real data[C]//Proceedings of 2021 IEEE International Conference on Electronics, Computing and Communication

Technologies. Bangalore, India: IEEE, 2021: 1-6.
[7] 周萱, 喻忠军, 曹越, 等. 一种ω-k算法的FPGA实现[J]. 电子设计工程, 2020, 28(22): 141-146.
ZHOU Xuan, YU Zhongjun, CAO Yue, et al. A FPGA implementation of ω-k algorithm[J]. Electronic Design Engineering, 2020, 28(22): 141-146.
[8] ZHU Daiyin, ZHANG Jindong, MAO Xinhua, et al. A miniaturized high resolution SAR processor using FPGA[C]// Proceedings of EUSAR the 11th European Conference on Synthetic Aperture Radar. Hamburg, Germany: VDE, 2016: 1-4.
[9] ZOU Lilong, SATO Motoyuki. An efficient and accurate GB-SAR imaging algorithm based on the fractional fourier transform [J]. IEEE Transactions on Geoscience and Remote Sensing, 2019, 57(11): 9081-9089.
[10] COOK S. Cuda programming: A developer's guide to parallel computing with gpus[M]. San Francisco, CA, USA: Morgan Kaufmann, 2013.
[11] WANG Yuwei, LI Xingming, HU Shanqing, et al. The research of SAR processing performance based on multi-core GPU [C]//Proceedings of International Conference on Signal and Information Processing, Networking and Computers. Chongqing, China: Springer, 2017: 156-163.
[12] BHOGENDRA RAO P V R R, SHASHANK S S. Parallelization of synthetic aperture radar (SAR) image formation algorithm [C]//Proceedings of the 1st International Conference on Computational Intelligence and Informatics. Hyderabad, India: Springer, 2016: 713-722.
[13] 苟立婷, 李勇, 朱岱寅, 等. 基于GPU的圆迹视频SAR实时成像算法[J]. 雷达科学与技术, 2019, 17(5): 550-556,563.
GOU Liting, LI Yong, ZHU Daiyin, et al. A real time algorithm for circular video sar imaging based on GPU[J]. Radar Science and Technology, 2019, 17(5): 550-556, 563.
[14] CUMMING I G, WONG F H. Digital processing of synthetic aperture radar data[M]. Boston, USA: Artech House, 2005.
[15] CHENG J, GROSSMAN M, MCKERCHER T. Professional CUDA C programming[M]. Indianapolis, USA: John Wiley & Sons, 2014.
[16] COOLEY J W, TUKEY J W. An algorithm for the machine calculation of complex Fourier series[J]. Mathematics of Computation, 1965, 19(90): 297-301.
[17] KENNETH M, EDWARD A. The FFT on a GPU[C]//Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware. SanDiego, USA: ACM, 2003: 112-119.
[18] KINDRATENKO V. Numerical computations with GPUs[M]. Berlin, German: Springer, 2014.
[19] FRIGO M, JOHNSON S G. FFTW: An adaptive software architecture for the FFT[C]//Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing. Seattle, WA, USA: IEEE, 1998: 1381-1384.
[20] FARBER R. CUDA application design and development[M]. Burlington, USA: Morgan Kaufmann, 2012.
[21] STOLT R H. Migration by Fourier transform[J]. Geophysics, 1978, 43(1): 23-48.
[22] 王金波, 唐劲松, 张森, 等. 一种宽带大斜视STOLT插值及距离变标补偿方法[J]. 电子与信息学报, 2018, 40(7): 1575-1582.
WANG Jinbo, TANG Jinsong, ZHANG Sen, et al. Range scaling compensation method based on STOLT interpolation in broadband squint SAS imagin[J]. Journal of Electronics & Information Technology, 2018, 40(7): 1575-1582.

作者简介:

谭运馨(1996-),男,硕士,研究方向:基于GPU的实时成像算法研究,E-mail: tanyx33@mail2.sysu.edu.cn。

黄海风(1976-),男,博士,教授,研究方向:空间电子和智能感知领域的基础理论和关键技术攻关研究。

赖涛(1980-),**通信作者**,男,博士,副教授,研究方向:超宽带轨道式SAR成像雷达系统设计与信息处理等,E-mail:lait3@mail.sysu.edu.cn。

但琪洪(1997-),男,硕士,研究方向:雷达实时成像与系统矫正。

欧鹏飞(1996-),男,硕士,研究方向:雷达图像处理。

(编辑:陈珺)