一种基于Dual-GPU的三次卷积插值并行算法研究

赖积保[1,2] 孟 圆[1,2] 余 涛[1] 王玉璟[2] 林英豪[1,2] 吕天然[1,2]
(中国科学院遥感应用研究所 北京 100101)[1] (河南大学计算机与信息工程学院 开封 475000)[2]

摘 要 针对传统三次卷积插值算法实现遥感图像放大在运算规模、计算速度等方面的不足,结合GPU的高性能计算优势,提出一种基于Dual-GPU(Graphic Processing Unit)的三次卷积插值并行算法(CCPA),即应用GPU的高性能计算技术将传统的三次卷积插值算法进行并行化处理,将图像的像素点个数平均分配给每个线程块,每个线程针对一个像素,线程在GPU中同时执行,以提高其插值效率。实验结果表明,该算法在保持放大后图像质量的同时,速度得到提升,随着图像分辨率的增大,该算法的优势更明显,在分辨率10240 * 10240的情况下,用GPU处理的速度比CPU提升了97.7%,用双GPU处理的速度是单GPU的2倍,并且在对放大遥感图像的质量和实时性均要求较高如地震、洪水等灾害的情况下,该算法具有实用价值。

关键词 三次卷积,CUDA,GPU,高性能计算
中图法分类号 TP39 **文献标识码** A

Research on Cubic Convolution Interpolation Parallel Algorithm Based on Dual-GPU

LAI Ji-bao[1,2] MENG Yuan[1,2] YU Tao[1] WANG Yu-jing[2] LIN Ying-hao[1,2] LV Tian-ran[1,2]
(Institute of Remote Sensing Applications, Chinese Academy of Sciences, Beijing 100101, China)[1]
(School of Computer and Information Engineering, Henan University, Kaifeng 475000, China)[2]

Abstract The traditional cubic convolution algorithm has to confront with the problems of large operational scale and slow efficiency, when it is used to realize the remote sensing image magnification. In this paper, GPU, as a burgeoning high performance computing technique, was proposed to make parallel processing of the traditional cubic convolution, which we call the Cubic Convolution Parallel Algorithm(CCPA). This algorithm that divides the pixels points equally to each block, guarantees each pixel point is executed by a thread and threads are executed simultaneously in GPU, improving the interpolation efficiency greatly. The experimental results show that compared with the traditional cubic convolution algorithm, this algorithm not only increases the calculation speed, but also achieves high quality image after zooming. Meanwhile, with the growth of image resolution, the advantages of the algorithm become more and more obvious, for instance, to the image of 10240 * 10240 resolutions, the speed processed by GPU is 97.7% higher than that by CPU, and the speed processed by double-GPU is twice than the speed processed by single GPU. Moreover, this algorithm also has profound practical value for remote sensing image processing under some emergency situations such as earthquakes, floods and other disasters, with the characteristic of good image quality and real-time mechanism.

Keywords Cubic convolution, CUDA, GPU, High-performance computing

1 引言

图像缩放在遥感科学领域有着广泛的应用,常用的图像插值包括最邻近点法、双线性插值和三次卷积插值[1]。虽然最邻近法和双线性插值这种传统的线性插值算法简单高效,但在图像放大时会导致锯齿效应、边缘模糊、高频信息丢失等问题。三次卷积插值能够克服以上两种算法的不足,改善低阶插值的缺点,并且计算精度高,但同时却引入了较多的计算量,导致图像处理速度降低。遥感图像数据的处理能力要远远滞后于遥感图像数据的获取能力,如何提高数据处理速度成为一个关键因素,这造成了选择插值方法的难度,早期的简单插值算法无法取得较好的效果,而具有较好效果的三次卷积插值算法因在处理过程中产生的速度上的局限,很多时候无法满足具体的实际需求。

为此,国内外研究人员一直在寻找各种加速三次卷积的方法,开展了一些相关研究,如设计三次卷积模板算法对算法进行离散化处理[2],利用增加缓冲区来提高三次卷积的速度[3],这样虽然得到了更好的图像质量,但增加了硬件成本,提速效果也不明显。针对三次卷积的提速问题,对并行加速也开展了一些研究工作,但该方法在系统维护上存在不便,因

到稿日期:2012-09-10 返修日期:2013-05-06 本文受国家重大科技专项高分辨率对地观测系统项目(E0101/1010/01/10,E0104/1112/XT-002),国家自然科学基金(60973126),国防科技工业民用专项科研技术研究项目(科工技2010A03A1000)资助。
赖积保(1982—),男,副教授,主要研究方向为空间数据处理,E-mail:laijibao@163.com;孟 圆(1988—),女,硕士生,主要研究方向为空间数据处理。

此无法满足其要求。然而，利用 CPU 实现三次卷积时受到 CPU 本身在浮点计算能力上的限制，对于高密度计算的三次卷积插值算法来说，过去传统的用 CPU 实现的方法，并未在处理性能与效率上有很大的进步。

目前，不降低算法复杂度而保持对遥感图像处理的高质量、高性能的实现是一个研究热点。GPU(Graphic Processing Unit)是特别为计算密集和高并行度计算设计的，将更多的晶体管用于数据处理而不是数据缓存和流控。2006 年 11 月，英伟达也推出了 CUDA(Compute Unified Device Architecture)，它利用 GPU 的并行计算引擎比 CPU 更高效地解决了许多复杂计算任务[4]，国内外现已将许多可以并行计算的算法交予 GPU 运算来提高算法速度[3]。因此，为了在图像放大效果和效率之间取得折中，本文提出了一种基于 Dual-GPU 的三次卷积插值并行算法，该算法针对三次卷积良好的局部性以及易于并行实现的特征，利用 GPU 对三次卷积算法进行加速，在具有三次卷积插值效果的基础上大大提高了三次卷积插值的效率。

2 三次卷积插值并行算法的分析与设计

2.1 三次卷积插值算法分析

三次卷积插值又称立方卷积插值，是一种更加复杂的插值方式，本质是一个三次多项式内插过程。目标图像的像素点经过计算得到源图像中对应的像素点$(i+u,j+v)$，利用采样点周围 16 个点的像素的信息作 3 次插值[3]，$(i+u,j+v)$周围相邻的 16 个点如图 1 所示。

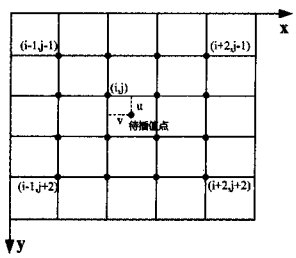

图 1 待插值点周围 16 个点

三次卷积插值算法需要选取插值基函数拟合数据，最常用的插值基函数 $s(w)$为理想插值函数 $sinc(w)=(sinw)/w$ 理论上的最佳三次逼近，即

$$s(w)=\begin{cases}1-2|w|^2+|w|^3, & |w|<1\\ 4-8|w|+5|w|^2-|w|^3, & 1\leqslant|w|\leqslant2\\ 0, & |w|>2\end{cases}\quad(1)$$

其插值公式如下：$f(i+u,j+v)=\boldsymbol{ABC}$。$f(i+u,j+v)$表示源图像待插值点$(i+u,j+v)$处的像素值。

其中，$\boldsymbol{A}$、$\boldsymbol{B}$、$\boldsymbol{C}$为矩阵，形式如下：

$$\boldsymbol{A}=[S(u+1)\quad S(u+0)\quad S(u-1)\quad S(u-2)]\quad(2)$$

$$\boldsymbol{B}=\begin{bmatrix}f(i-1,j-1)&f(i-1,j+0)&f(i-1,j+1)&f(i-1,j+2)\\ f(i+0,j-1)&f(i+0,j+0)&f(i+0,j+1)&f(i+0,j+2)\\ f(i+1,j-1)&f(i+1,j+0)&f(i+1,j+1)&f(i+1,j+2)\\ f(i+2,j-1)&f(i+2,j+0)&f(i+2,j+1)&f(i+2,j+2)\end{bmatrix}\quad(3)$$

$$\boldsymbol{C}=[S(v+1)\quad S(v+0)\quad S(v-1)\quad S(v-2)]^{\mathrm{T}}\quad(4)$$

该算法不仅考虑到 4 个直接相邻点的像素影响，而且考虑到各邻点间像素值变化率的影响。

由式(1)—式(4)容易计算得出，三次卷积插值算法每计算一个插值点像素值需要 70 次乘法、45 次加法；而双线性插值每个像素点的计算只需要 3 次乘法、6 次加法。显然三次卷积插值的运算量远大于双线性插值的运算量，放大效果也优于双线性插值。由此可见，三次卷积插值可得到更接近高分辨率图像的效果，但复杂的计算也导致了运算量的急剧增加，使得图像放大时的速度降低，达不到实时应用的需求。但是该算法对图像中的每一个像素进行插值的过程具有很好的独立性，其过程以及所利用的像素资源之间均不存在耦合性，于是本文用 GPU 的并行计算特性对该算法进行加速处理和相关优化。实验结果表明，基于 GPU 的三次卷积插值算法在保持图像放大质量的基础上，可有效地提高处理图像的效率。

2.2 GPU 并行化实现流程

CUDA 编程模型让 CPU 负责进行逻辑性强的事物处理和串行计算，GPU 则专注于执行能够被高度线程化的并行处理任务[5]。在用 CUDA 进行编程时，可将 GPU 看作是能够并行执行很多个线程的计算设备，将在 CPU 上运行的应用程序的数据并行、计算密集的部分卸载到此设备上，GPU 最后将计算的结果返回给 CPU 中的应用程序。

在 CUDA 中，线程的组织包括 3 个层次：线程网格(Grid)、线程块(Block)和独立线程(Thread)。如图 2 所示，一个可以并行化的 GPU 程序 Kernel 以 Grid 的形式组成，每个线程网格由若干个 Block 组成，而每个 Block 又由若干个 Thread 组成。不同的 Block 被分派到不同的流处理器上并行执行。由于块内的所有线程必须存在于同一个处理器核心中，且共享有限的存储器资源，因此一个块内的线程数目是有限的。在目前的 GPU 上，一个线程块可以包含多达 1024 个线程。然而，一个内核可被多个同样大小的线程块执行，所以总的线程数等于每个块内的线程数乘以线程块数[6]。

必须创建有足够块的网格，使得一个线程处理一个矩阵元素。为简便起见，假设网格每一堆上的线程数可被块内对应维上的线程数整除。网格内的每个块通过可在内核中访问的一维或二维索引唯一确定，此索引可通过内置的 blockIdx 变量获得。每个执行内核的线程也拥有一个独一无二的线程 ID，可以通过内置 threadIdx 变量在内核中访问。

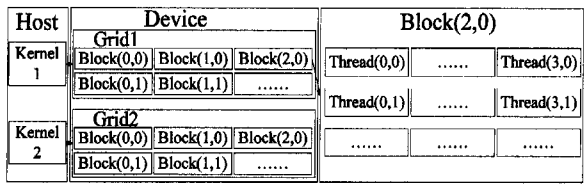

图 2 并行线程组织结构

通过利用 CUDA，GPU 便成为一种真正通用的数据并行计算设备。CUDA 通过允许程序员定义称为内核的 C 函数扩展了 C，内核调用时会被 N 个 CUDA 线程执行 N 次。内核使用_global_声明符定义，使用一种全新〈〈〈…〉〉〉语法指定执行某一指定内核调用的线程数。由 CPU 调用、在 GPU 上执行的函数被称为核心函数(Kernel function)。CPU 对 GPU 内部 Kernel 函数的调用形式如下：

Kernelname〈〈〈*blocknum*,*threadnum*〉〉〉(*A*,*B*,*C*)

其中，*blocknum* 代表本次计算使用 1 个 grid 里的 *blocknum* 个 blocks；*threadnum* 代表本次并行计算使用的每一个 block 里

均安排使用 *threadnum* 个 threads；*A*，*B*，*C* 则为 CPU 与 GPU 之间传递的参数[7]。

2.3 基于 GPU 的三次卷积插值算法

为了体现 GPU 的并行性，将三次卷积插值算法对每个像素点的插值过程交给不同的线程完成，即将插值过程转移给 GPU 运算。本文选用的测试图像的长与宽均可被 16 整除，所以为了最大限度地运用 GPU，每个线程块内包含有 256 个线程[8]。

2.3.1 实现步骤

利用 GPU 实现三次卷积插值算法的具体实现步骤如下：

Step1 利用 GDAL 组件读取待插值遥感图像，计算出图像的长与宽以及图像的长与宽均放大 3 倍后的大小；

Step2 在 CPU 上为待插值遥感图像和插值后的图像分配空间；

Step3 调用在 Core. cu 文件中自定义的 OpenUpGpu 函数，将计算所用的参数共 6 个（源图像，长，宽，目标图像，长，宽）传输到 GPU，定义 block 和 thread 的数目，开辟线程，获得 GPU 的 ID 号并启动 GPU 设备，在 GPU 上为传过来的待插值图像以及插值后的图像分别分配空间；

Step4 调用__global__ void ThreeInterpolation<<<grid,threads>>>(参数 1，参数 2，……)全局函数，获取插值后影像的每一个像素点的索引值，然后开始运用 GPU 进行三次卷积插值的一系列运算。本文选用图像均为 256 的倍数，所以一个线程块中分配 256 个线程，用插值后影像的高和宽分别除以 16，得到所需线程块在高和宽两个方向上的个数，如果有余数，则为余数再分配一个线程块计算；

Step5 将插值后的图像由显存 GPU 传给主存 CPU，释放显存 GPU 上的资源。

ThreeInterpolation 函数里面的形式参数和其他变量均是在 GPU 的存储器里面，那么在调用该函数之前利用 *cudamalloc*() 分配 GPU 的存储空间，用 *cudamemcpy* 复制 CPU 的内容到 GPU。释放显存 GPU 的资源用 *cudafree*() 函数。

2.3.2 GPU 核心部分代码实现

根据 2.1 节中提到的公式 $f(i+u,j+v)=\boldsymbol{ABC}$ 可计算出像素点 $(i+u,j+v)$ 处的像素值，本文首先按照式(1)计算出矩阵 ***A*** 和矩阵 ***C***，再将矩阵 ***A***、***B***、***C*** 相乘，即得结果。下述(1)、(2)均是在 core. cu 文件中定义的全局函数 *ThreeInterpolation* 中实现的。

(1)计算矩阵 ***A*** 和 ***C***

计算矩阵 ***A*** 和矩阵 ***C*** 对应像素点 $(i+u,j+v)$ 上的值，u、v 为浮点坐标 $(i+u,j+v)$ 的小数部分。SA[3]－SA[0]存放的是矩阵 ***A*** 中的元素，SV[3]－SV[0]存放的是矩阵 ***C*** 中的元素。

```
for(int m=0;m<=3;m++)
 {
   double W
   置 W 为 u+m-2
   置 AbsU 为绝对值|W|
   置 AbsU2 为|W|²
   置 AbsU3 为|W|³
   //按式(1)计算矩阵 A 中的元素
   //SA[0]:S(u-2) SA[1]:S(u-1)
   //SA[2]:S(u) SA[3]:S(u+1)
   if(AbsU<1)
   then 使 SA[m]= 1-2 * AbsU2+AbsU3
   else if(AbsU<2)
   then 使 SA[m]=4-8 * AbsU+5 * AbsU2-AbsU3
   else then 使 SA[m]= 0
 }
```

重复上述同样的运算，计算出矩阵 ***C*** 的 4 个元素。

(2)计算矩阵 ***ABC***

```
// 先将矩阵 A 与矩阵 B 相乘
置 AB1 为 A 与 B 相乘的第一个元素
置 AB2 为 A 与 B 相乘的第二个元素
置 AB3 为 A 与 B 相乘的第三个元素
置 AB4 为 A 与 B 相乘的第四个元素
//计算点(i+u,j+v)的像素值，即 ABC
待插值图像 f(i+u,j+v)的值为
AB1 * SC[3]+AB2 * SC[2]+AB3 * SC[1]+AB4 * SC[0]
```

利用 GPU 对图像中的所有像素点进行三次卷积插值的操作后，将插值后的图像传递给主机 CPU。

3 实验结果与分析

3.1 实验环境

为了测试算法的有效性，用 C++实现了本文算法。具体的软硬件系统配置如表 1 所列。

表 1 软硬件参数

名称	内容
处理器	Inter(R)Core(TM)i7 CPU 950 @ 3.07GHz
内存	16.0GB
硬盘	2.0TB
操作系统	Windows7 旗舰版，64 位操作系统
开发环境和语言	Microsoft Visual Studio 2010，C++
显卡	NVIDIA Tesla C2050，3GB 显存
CUDA 版本	4.0 版

3.2 实验过程

考虑到最邻近插值放大效果一般较差，本文仅选取传统的用 CPU 实现的双线性插值以及三次卷积插值与基于 GPU 的 CCPA 进行比较。为了进行多组数据的对比试验，选取环境星 CCD 影像的第一个波段，首先对原始图像数据进行裁减获得分辨率分别为 256 * 256、512 * 512、1024 * 1024、2048 * 2048、4096 * 4096、8192 * 8192 和 10240 * 10240 的图像。分别运用双线性插值、三次卷积插值与基于 GPU 的 CCPA 对 7 幅不同分辨率的图像进行放大 9 倍的插值对比试验，分别运行插值的 CPU 和 GPU 程序，记录插值处理时间，比较基于 GPU(Tesla C2050 和 GTS 450)和基于 CPU 的实现方法的时间差异和加速比。考虑到不同 GPU 的差异，本文也对比了 Tesla C2050(显存 3G)与 GTS 450(显存 1G)的处理速度，由于分辨率 8192 以及更大的图像在 450 上内存溢出，因此没有进行对比，结果如表 2 和图 3 所示，内存不足的部分用虚线框表示，如图 4 所示。

表 2 不同型号的 GPU 的处理时间对比

分辨率	NVIDIA Tesla C2050		NVIDIA GTS 450	
	总时间/(s)	数据传输时间/(ms)	总时间/(s)	数据传输时间/(ms)
256	0.063	2.859	0.088	5.956
512	0.075	10.156	0.111	24.692
1024	0.115	50.035	0.207	98.957
2048	0.248	105.322	0.57	384.017
4096	0.754	422.285	1.979	1527.355

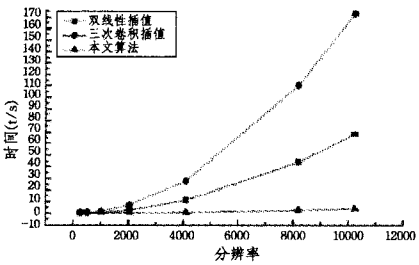

图 3　不同分辨率图像在放大 9 倍下的插值算法时间对比

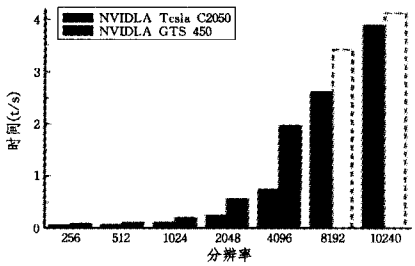

图 4　3 次卷积插值的 CPU 与两种型号 GPU 处理速度加速比(虚线框为内存不足)

3.3　算法优化

目前国内外有一些对双 GPU 技术应用的案例，本文也对双 GPU 在 CCPA 的应用进行了探索，以双 GPU 为例，取得了理想的效果。通过 3.2 节的实验可以看到 GPU 对三次卷积算法的速度提高很大，加速比达到了一百倍。故在本小节通过双 GPU 来实验，以进一步获取更快的速度，从而达到对算法优化的目的。

3.3.1　任务分配

将环境星 CCD 的第一个波段的影像拆分为两部分，分别放在两个 GPU 上并行处理，以此提高速度，也解决了内存溢出的问题。因为对于 NVIDIA GTS 450 显卡的计算能力来说，计算分辨率为 8192 和 10240 的图像会受到内存不足的限制。

3.3.2　任务管理

在单 GPU 程序中，GPU 的程序是由默认的主机线程控制的，我们并没有通过显示的创建线程去管理 GPU 程序，无需考虑线程的问题，但是在双 GPU 的环境下，我们就需要考虑任务管理。因为在双 GPU 程序中，每个 GPU 需要由不同的线程管理，所以须为每个 GPU 创建一个独立的线程。在不同的 GPU 线程中，通过调用不同的设备 ID 去驱动不同的 GPU，最后进行线程的同步，在 CPU 中合成算法结果。双 GPU 的环境下处理流程如图 5 所示。

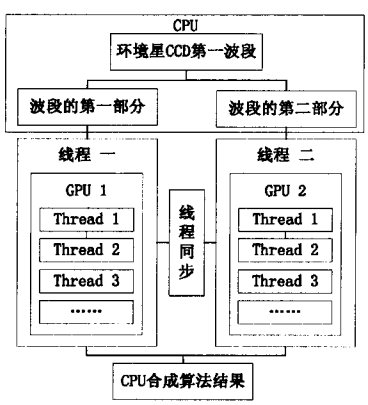

图 5　双 GPU 的三次卷积插值算法处理流程图

对双 GPU 的结果进行测试，使用同样的测试数据，结果如表 3 所列。

表 3　单 GPU 与双 GPU 的处理时间对比

分辨率	单 GPU 时间/(s)	双 GPU 时间/(s)	加速比
256	0.063	0.074	0.851
512	0.075	0.081	0.926
1024	0.115	0.120	0.958
2048	0.248	0.235	1.055
4096	0.754	0.412	1.830
8192	2.622	1.332	1.968
10240	3.887	1.828	2.126

3.4　结果分析

由表 2 可看出在保持了图像质量的基础上，利用 GPU 大大提高了三次卷积插值的处理速度，但在放大不同分辨率的图像时效果不尽相同。对于分辨率较小的遥感影像，GPU 算法在处理速度上的优势不明显，如 256 * 256 分辨率的图像的双线性插值结果优于 GPU 处理结果，这是由于显存和主存间的数据通信耗费了时间，线程数量不足以至于不能充分隐藏数据传输和读取延迟[8]。

从表 3 看出，Tesla C2050 由于显存大的优势，处理速度优于 GTS 450；Tesla C2050 数据传输时间占总时间的比例平均为 32%，GTS 450 的为 42%，Tesla C2050 的数据传输占用比例较少，这是由于 C2050 更大的数据集能够保存在直接附属于 GPU 的本地存储器上并且其具有高速的 PCIe Gen 2.0 数据传输率，从而实现了性能的最大化并减少了数据传输的情况。GPU 快速增长的性能使得 GPU 已经完全有实力与传统的 CPU 竞争。但从图 5 中可看出，随着遥感图像分辨率大小的增加，加速比也随之增加，分辨率越大的图像，加速比越高，Tesla C2050 的加速比甚至达到了 40 倍以上，说明分辨率越大，基于 GPU 运算的速度优势也越来越明显。

在双 GPU 环境下存在着两种并行，一种是任务级的两个 GPU 之间的并行处理，一种是每个 GPU 内部上百个计算单元之间的并行处理[9]。在分辨率低于 1024 的情况下，双 GPU 对速度提高的优势不明显，而在分辨率为 2048 及以上的情况下，驱动两个 GPU 并行计算则会比单 GPU 的效率提高一倍以上。

结束语　针对 CPU 环境下的三次卷积插值算法的加速问题，目前国内外已经做了相关研究，但是提速效果并不明显。同 CPU 相比，GPU 内对矩阵变换、三角运算等数学函数的支持，使其浮点运算能力比 CPU 高出 3 到 4 倍[10]。本文从 GPU 高性能并行计算的机理出发，在分析利用传统插值算法与三次卷积插值算法进行图像重采样的基础上，提出了基于 Dual-GPU 的三次卷积插值算法。该方法可充分利用 GPU 的并行计算引擎，有效地加快传统三次卷积算法的速度，解决了对遥感图像插值的速度与效率不可折中的问题，实现了对三次卷积插值算法进行提速的目标，相比基于传统的双线性插值与基于 CPU 的三次卷积插值算法有计算高效的优势。与此同时，针对 CPU 和 GPU 之间的数据通信速度及计算结果的保存与回读操作问题，还需要进一步研究。

参 考 文 献

[1]　高成敏，陈良，林永和. 双三次卷积模板算法[J]. 计算机工程与应用，2009，45(17)：151-154

(下转第 33 页)

结束语 针对浮点 FMA 部件对独立乘法和加法运算延迟不利的缺点，我们设计并实现了一种分离通路的乘加器(SPFMA)，其在保持 FMA 运算延迟 6 拍不变的情况下，独立乘法、加法运算延迟由 6 拍减为 4 拍。经过 DC 逻辑综合，结果表明该乘加器相对于基本 FMA 乘加器面积增加了31.30%，CTP 延迟减少了 3.19%。利用硬件仿真加速器验证系统运行 SPEC CPU2000 浮点测试课题的结果表明，所有浮点课题性能均有所提高，最大提高 5.54%，平均提高 1.66%。本文的主要贡献是，提出了一种新的乘加结构 SPFMA，它可以在保持 FMA 运算优点的同时克服其缺点，减小独立乘法和加法等运算的延迟，有利于浮点性能的进一步提升。此外，SPFMA 结构还可以支持独立乘法和加法运算的并行执行，只要将乘法和加法运算分配在独立的执行流水线上。相对于桥接 FMA 结构，SPFMA 结构硬件开销更小，而且不增加 FMA 运算延迟。下一步还可以对 SPFMA 的面积、延迟和流水线站台划分进行优化，实现更优化的设计。

参 考 文 献

[1] Montoye R K, Hokenek E, Runyon S L. Design of the IBM RISC System/6000 Floating-Point Execution Unit[J]. IBM Journal of Research and Development, 1990, 34: 61-62

[2] Eisen L, III J W W, Tast H-W, et al. IBM POWER6 Accelerators: VMX and DFU [J]. IBM Journal of Research and Development, 2007, 51: 663-684

[3] Boersma M, Kroener M, Layer C, et al. The POWER7 Binary Floating-Point Unit[C]// Proceedings of IEEE Symposium on Computer Arithmetic. Tübingen, Germany, IEEE Computer Society, 2011

[4] Sharangpani H, Arora K. Itanium Processor Microarchitecture [J]. IEEE Micro Magazine, 2000, 20(5): 24-43

[5] Maruyama T, Yoshida T, Kan R, et al. SPARC 6 4 VIIIfx: A New-generation Octocore Processor for Petascale Computing [J]. IEEE Micro, March-April 2010: 30-40

[6] Glaskowsky P N. NVIDIA's Fermi: The First Complete GPU Computing Architecture, Nvidia Fermi Whitepaper [EB/OL]. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf, 2012-09-27

[7] IEEE Computer Society. IEEE Standard for Floating-Point Arithmetic[S]. IEEE Standard 754-2008. New York, USA, August 2008

[8] Lutz D. Fused Multiply-Add Microarchitecture Comprising Separate Early-Normalizing Multiply and Add Pipelines[C]// Proceedings of IEEE Symposium on Computer Arithmetic. Tübingen, Germany, IEEE Computer Society, 2011

[9] Galal S, Horowitz M. Latency Sensitive FMA Design[C]// Proceedings of IEEE Symposium on Computer Arithmetic. Tübingen, Germany, IEEE Computer Society, 2011

[10] SPEC. CPF2000(Floating Point Component of SPEC CPU2000) [EB/OL]. http://www.spec.org/cpu2000/CFP2000, 2012-09-27

[11] Quach N, Flynn M J. An Improved Algorithm for High-Speed Floating Point Addition[R]. CSL-TR-90-442. Computer Systems Laboratory, Stanford University, Aug. 1990

[12] Schwarz E M, Floating-Point B. Unit Design: the fused multiply-add dataflow, High-Performance Energy-Efficient Microprocessor Design [M]// Oklobdzija V G, Krishnamurthy R K, eds. Springer, Printed in the Netherlands, 2006: 199-201

[13] Schmookler M S, Nowka K J. Leading Zero Anticipation and Detection A Comparison of Methods[C]// Proceedings of IEEE Symposium on Computer Arithmetic. Vail, CO, USA, IEEE Computer Society, June 2001: 11-17

[14] 梅小露. 浮点乘加部件中三操作数前导 1 预测算法的设计[J]. 微电子学与计算机，2005, 22(12): 16-20

[15] Lang T, Bruguera J. Floating-Point Fused Multiply-Add with Reduced Latency [J]. IEEE Transactions on Computer, 2004, 53(8): 088-1003

[16] Bruguera J D, Lang T. Floating-point fused multipy-add: reduced latency for floating-point addition[C]// Proc. 17th IEEE Symp. Computer Arithmetic. Hyannis, June 2005: 27-29

[17] Seidel P M. Multiple path IEEE floating-point fused multiply-add[C]// Proc. 46th Int. IEEE Midwest Symp. Circuits and Systems(MWS-CAS). 2003

[18] Quinnell E. Floating-Point Fused Multiply-Add Architectures [D]. University of Texas at Austin, 2007

[19] 靳战鹏，白永强，沈绪榜. 一种 64 位浮点乘加器的设计与实现[J]. 计算机工程与应用，2006, 28(18): 95-98

[20] 吴铁彬，刘衡竹，杨惠，等. 一种快速 SIMD 浮点乘加器的设计与实现[J]. 计算机工程与科学，2012, 34(1): 69-73

(上接第 27 页)

[2] 冯煌. GPU 图像处理的 FFT 和卷积算法及性能分析[J]. 计算机工程与应用，2008, 44(2): 120-129

[3] Wang Xiang, Ding Yong. Efficient implementation of a cubic-convolution based image scaling engine[J]. Zhejiang Univ-SciC (Compute & Electron), 2011, 12: 743-753

[4] 肖汉. 利用 GPU 计算的双线性插值并行算法[J]. 小型微型计算机系统，2010, 32(11): 2241-2245

[5] 郭一汉，史美萍，吴涛. 基于 GPU 的实时图像拼接[J]. 计算机科学，2012, 39(7): 257-261

[6] 卢凤顺. CPU/GPU 协同并行计算研究综述[J]. 计算机科学，2011, 38(3): 5-10

[7] Jason S, Edward K. CUDA by example: an introduction to general-purpose GPU programming[M]. Beijing: Tsinghua University Press, 2010: 64-74

[8] 刘建明. 基于全局优化的图像修复及其在 GPU 上实现[J]. 浙江大学学报：工学版，2011, 45(2): 247-252

[9] 杜刘革. 基于多 GPU 的 FDTD 并行算法机器在电磁仿真中的应用[D]. 济南：山东大学，2011

[10] 杨靖宇. 遥感影像 GPU 并行化处理技术与实现方法[D]. 郑州：解放军信息工程大学测绘学院，2008