单节点多GPU集群下HPL动态负载均衡优化

陈任之 黄立波 陈项颢 王志英
(国防科学技术大学计算机学院 长沙 410073)

摘 要 现有GPU加速的高性能Linpack基准测试程序(HPL)一般采用基于实际运算能力的动态负载均衡算法来实现。然而该算法在单节点多GPU的平台上表现不佳,其原因是单节点多GPU平台上单个GPU计算量小,并且GPU与CPU的总性能差距较大。为此,提出了经验指导的动态负载均衡算法以及多GPU自适应负载均衡算法,并且在单节点多GPU平台上进行了验证,结果显示,其比现有的基于NVIDIA费米GPU的HPL有6.3%的加速效果。

关键词 HPL,GPU,动态负载均衡算法
中图法分类号 TP301 **文献标识码** A

Optimizing HPL Benchmark on Multi-GPU Clusters

CHEN Ren-zhi HUANG Li-bo CHEN Xu-hao WANG Zhi-ying
(School of Computer, National University of Defense Technology, Changsha 410073, China)

Abstract The current high performance Linpack benchmark accelerated by GPU usually employs the performance-based dynamic load balancing algorism. However, this algorism does not perform well on multi-GPU cluster. The reason is that each GPU on this kind of node has a smaller calculating scale, and the gap of the total performance between GPU and CPU is larger. Therefore, this article proposed an experience-based dynamic load balancing algorism and a multi-GPU adaptive load balance algorithm. Besides, the article tested these two algorisms on multi-GPU cluster and it achieves a 6.3% acceleration compared with the latest NVIDIA's HPL accelerated by GPU.

Keywords HPL, GPU, Dynamic load balancing algorithm

随着半导体工艺的发展,单处理器芯片内集成的晶体管数目迅速增多,图形处理器(Graphics Processing Unit,GPU)的性能因此得到了飞速的提升并且远超于CPU,利用GPU加速HPL(High Performance Computing Linpack Benchmark)[1]逐渐成为热点。HPL,即高性能计算Linpack基准测试程序,是通过求解一个稠密线性方程组来测试集群计算性能的标准测试程序。本文分析与优化的目标为实现基于NVIDIA费米GPU的HPL。

GPU上的HPL将主要计算部分在GPU与CPU间并行完成。如何实现GPU与CPU间的任务负载均衡是充分发挥硬件计算能力的关键。王峰等[2]提出一种基于实际运行性能指导的动态负载均衡算法,其根据HPL运行时GPU与CPU的实际性能调整两者间的负载。现有的基于NVIDIA费米GPU的HPL代码采用该算法。

然而在单节点多GPU的情况下,基于实际运行性能指导的动态负载均衡算法加速效果并不理想。主要存在如下问题:第一,该算法是基于一个假设,即GPU与CPU性能变化不剧烈,然而单节点中存在多GPU时,由于单个GPU计算量较小,随着计算规模递减,GPU计算性能变化较大。第二,基于实际运行性能指导的动态负载均衡算法并没有考虑GPU与CPU计算能力差距对性能的影响。使用该算法时发生GPU等待CPU的可能性与CPU等待GPU的可能性是相同的。然而GPU的计算能力远远高于CPU,因此GPU等待CPU的情况的性能损失比CPU等待GPU的情况更为严重。当单个节点内包含多块GPU时,这种情况会更加严重。

本文针对上述两个问题,提出了经验指导的动态负载均衡算法(Experience-based Dynamic Load Balance Algorithm,简称ED)与多GPU自适应负载均衡算法(Multi-GPU Adaptive Load Balance Algorithm,简称MA)。在ED算法中,负载均衡主要依据GPU性能-规模函数。该函数可以通过多次实验获取GPU性能与规模的关系,再通过曲线拟合获得。GPU性能-规模函数可较为准确地反映出不同规模下GPU的实际性能,尤其在GPU性能变化剧烈的情况下,该算法比原算法更接近GPU实际性能。MA算法主要针对原算法中GPU等待CPU的可能性与CPU等待GPU的可能性相同而进行优化的。该算法根据GPU与CPU计算能力调整GPU等待CPU的概率,有效降低了GPU等待CPU的概率。与NVIDIA原算法相比,采用ED与MA算法时可获得6.3%的加速效果。

本文第1节简要介绍HPL算法;第2节介绍相关工作;

到稿日期:2012-10-19 返修日期:2012-12-21 本文受国家自然科学基金(61070037,61103016)资助。
陈任之 男,硕士,主要研究方向为高性能计算与体系结构,E-mail:chenrenzhi1989@gmail.com(通信作者);黄立波 男,博士,讲师,主要研究方向为微处理器体系结构;陈项颢 男,博士,主要研究方向为微处理器体系结构;王志英 男,教授,博士生导师,主要研究方向为计算机体系结构。

第3节介绍针对单节点多GPU平台的优化策略，包括ED算法与MA算法；第4节给出实验结果；最后总结全文。

1 算法分析

1.1 HPL算法

HPL通过求解一个稠密线性方程组来测试集群计算性能[3]。如式(1)所示：

$$Ax=b \tag{1}$$

式中，$A=(a_{ij})_{N\times N}$，$b=(b_1,b_2,\cdots,b_N)^{\mathrm{T}}$，$x=(x_1,x_2,\cdots,x_N)^{\mathrm{T}}$，$A$与$b$均已知，$x$为待求向量。HPL求解式(1)采用了LU分解算法。首先对待求的伴随矩阵进行LU分解，然后再回代求出列向量x。其中LU分解过程的时间复杂度为$O(N^3)$，回代求列向量x过程的时间复杂度为$O(N^2)$，因此主要考虑LU分解过程。

1.2 LU分解算法

(1)式中A矩阵进行LU分解可表示为：

$$A=\begin{pmatrix}A_{11} & A_{12}\\ A_{21} & A_{22}\end{pmatrix}=\begin{pmatrix}L_{11} & 0\\ L_{21} & L_{22}\end{pmatrix}\times\begin{pmatrix}U_{11} & U_{12}\\ 0 & U_{22}\end{pmatrix}=L\times U \tag{2}$$

式中，A_{ij}为已知矩阵块，L_{ij}与U_{ij}为待求矩阵块。由式(2)可得：

$$A_{11}=L_{11}\times U_{11} \tag{3}$$

求L_{11}与U_{11}等同为求矩阵A_{11}的LU分解，因此可以迭代求解。在通过式(3)求解出L_{11}与U_{11}的基础上，L_{21}与U_{12}亦可由式(2)求出：

$$L_{21}=A_{21}\times U_{11}^{-1} \tag{4}$$

$$L_{12}=A_{12}\times L_{11}^{-1} \tag{5}$$

由式(2)可得：

$$A_{22}-L_{21}\times U_{12}=L_{22}\times U_{22} \tag{6}$$

求解L_{22}与U_{22}等同为求矩阵$A_{22}-L_{21}\times U_{12}$的LU分解。因此可以首先求出矩阵$A_{22}-L_{21}\times U_{12}$，然后递归求解出$L_{22}$与$U_{22}$。

对程序运行时间进行分析可得到程序运行特性。实验数据表明，求解中主要耗时部分为式(6)中求解矩阵$A_{22}-L_{21}\times U_{12}$的过程。该过程通过调用双精度矩阵乘加函数(DGEMM)进行求解，占用总运行时间约94%。

1.3 GPU加速HPL

基于NVIDIA费米GPU的HPL主要是通过CPU与GPU协同完成DGEMM计算来实现加速[4]。以矩阵乘运算$C=A\times B$为例，矩阵划分如图1所示，先将矩阵B与A的一部分A_g传至GPU显存内，然后GPU计算$C_g=A_g\times B$，同时CPU计算$C_c=A_c\times B$。完成计算后，再将GPU显存中的C_g传至主存中，并与C_c组成完整矩阵C。

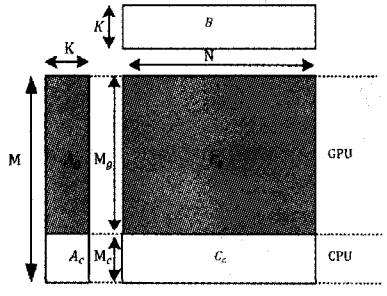

图1 异构平台下GPU与CPU的任务划分

由于GPU与CPU需要并行完成部分矩阵的求解，因此涉及到负载均衡问题。定义变量$ratio$：

$$ratio=T_{gpu}/T_{cpu} \tag{7}$$

$ratio$可以反映出GPU与CPU间的负载关系。负载完全均衡的情况下$ratio=1$；当$ratio>1$时，GPU负载过重，CPU等待GPU；当$ratio<1$时，CPU负载过重，GPU等待CPU。定义负载均衡因子如下：

$$R_{split}=\frac{M_{gpu}}{M_{gpu}+M_{cpu}}(0\leqslant R_{split}\leqslant 1) \tag{8}$$

式中，M_{gpu}与M_{cpu}在图1中已阐明含义。在HPL中，LU分解过程为递归求解，DGEMM将被多次调用，并且计算规模逐步递减。每次执行DGEMM时通过确定R_{split}将矩阵划分为两部分，并分别由GPU与CPU求解。通过调整R_{split}实现GPU与CPU的求解时间基本相等。

2 相关研究

张先轶提出静态的任务均衡方法[6]：通过多次的实验获得最佳的比例因子R_{split}，每次任务分割时，CPU与GPU负载比例固定为R_{split}。但是由于GPU与CPU的性能并不是恒定的，因此静态任务均衡效果并不理想。

王峰等提出R_{split}可根据上次DGEMM中GPU与CPU的实际浮点运算能力确定[2]：

$$R_{split}=\frac{M_{gpu}}{M_{gpu}+M_{cpu}}=\frac{P_{gpu}}{P_{gpu}+P_{cpu}} \tag{9}$$

P_{gpu}与P_{cpu}可通过如下公式计算：

$$P_{gpu}=M'_{gpu}\times N'\times K/T'_{gpu} \tag{10}$$

$$P_{cpu}=M'_{cpu}\times N'\times K/T'_{cpu} \tag{11}$$

式中，M'_{gpu}，M'_{cpu}，N'为上次DGEMM中在CPU与GPU上DGEMM求解矩阵的维度；T'_{cpu}与T'_{gpu}为上次DGEMM在CPU与GPU上的实际执行时间。

该动态均衡算法对比静态任务均衡有明显的性能提升，现有的基于NVIDIA费米GPU的HPL已使用了该算法。但是在多GPU情况下，由于没有考虑GPU总体性能远大于CPU总体性能以及单个GPU计算量下降的特点，任务均衡效果仍不理想。

3 单节点多GPU平台下HPL优化

3.1 单节点多GPU平台下HPL的特点

单节点内多GPU的集群中，主要特点有以下两点。

1)单个GPU的计算量下降

单个节点中内存大小固定，导致HPL总的计算规模为定值。当节点内GPU数增加时，每个GPU的计算量减少。以浪潮MF5588服务器为例，该服务器中有12条内存插槽。假设使用8G内存条，则总内存大小为96G。如果HPL占用80%的内存，则在单GPU的情况下，GPU需要求解的矩阵约占内存57G。而在单节点内4GPU的情况下，每个GPU需要求解的矩阵约占内存17G。根据HPL的空间复杂度$O(N^2)$以及时间复杂度$O(N^3)$，可求得单GPU与4GPU情况下每个GPU的计算量比为6.2∶1。

2)GPU总体计算能力远大于CPU总体计算能力

当节点内GPU个数增加时，GPU整体计算能力上升，导致GPU计算能力与CPU计算能力差距进一步拉大。以浪潮MF5588服务器为例，该服务器最多可容纳二路CPU与4块

GPU。如图 2(b)所示，CPU 与 GPU 分别以 Xeon 5675 与 M2090 计算时 GPU 与 CPU 的计算能力比为 11.3∶1，而在单 GPU 情况下该比例仅为 2.8∶1。

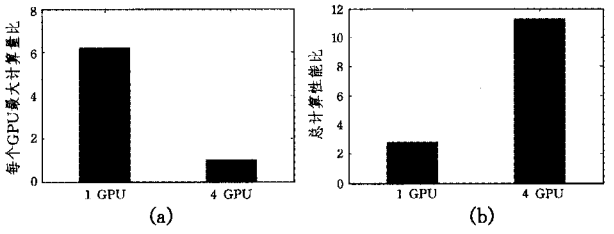

图 2　多 GPU 情况下 HPL 的特点

3.2　经验指导的动态负载均衡算法(ED 算法)

GPU 与 CPU 的实际浮点运算性能并不恒定，而是随着计算规模减小而下降的[5]。如图 3 所示，GPU 在浮点运算总量大于 1×10^{11} 的情况下，性能较为稳定，而在浮点运算次数小于 1×10^{11} 的情况下，性能明显下降[6]。而 CPU 在浮点总量小于 5×10^{9} 的情况下才明显下降。

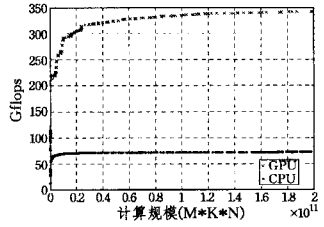

图 3　GPU 与 CPU 在不同规模下的性能曲线

HPL 是一个递归求解过程[7]，其计算规模不断减小。在多 GPU 情况下，单个 GPU 总计算量较小，当计算规模减小时，GPU 较快进入性能下降段。因此两次迭代过程中 GPU 性能差距较大。基于实际运行性能指导的动态负载均衡方法采用上次迭代中 GPU 的性能估算本次 GPU 的性能是不准确的。

可引入 ED 算法解决上述问题。实验表明，GPU 与 CPU 的性能是与规模相关的。在计算规模确定的情况下，GPU 与 CPU 的性能只在一个较小范围内波动。因此可引入 CPU 与 GPU 性能-规模经验函数 $g_{cpu}(N_1)$ 与 $f_{gpu}(N_2)$ 来求出负载均衡因子 R_{split}。其中 $g_{cpu}(N_1)$ 与 $f_{gpu}(N_2)$ 可通过实验获取 GPU 与 CPU 性能与规模的关系后再通过函数拟合获得。

给出一种 GPU 与 CPU 性能-规模函数，以及求解 R_{split} 的方法。通过多次实验获取 GPU 与 CPU 在不同规模下的实际性能数据。将数据按规模大小排序后，分组采用线性回归方程求解出每组的拟合直线。所有的拟合直线组成待求的一次分段函数 $f_{gpu}(N_2)$ 与 $g_{cpu}(N_1)$，如图 4 与图 5 所示。当 $g_{cpu}(N_1)$ 与 $f_{gpu}(N_2)$ 为一次分段函数时，CPU 性能-规模函数的第 i 段直线方程为 $g_{cpu}(N_1)=k^i_{cpu}\times N_1+b^i_{cpu}$，GPU 性能-规模函数的第 j 段直线方程为 $f_{gpu}(N_2)=k^j_{gpu}\times N_2+b^j_{gpu}$，则有：

$$\frac{M_{gpu}}{M}=\frac{k^j_{gpu}\times M_{gpu}+b^j_{gpu}}{k^j_{gpu}\times M_{gpu}+b^j_{gpu}+k^i_{cpu}\times(M-M_{gpu})+b^i_{cpu}} \tag{12}$$

式中，M 已知，M_{gpu} 为待求量。整理式(12)可得一元二次方程：

$$(k^j_{gpu}-k^i_{cpu})\times M^2_{gpu}+(b^j_{gpu}+b^i_{cpu}+k^i_{cpu}\times M-k^j_{gpu}\times M)\times M_{gpu}-M\times b^j_{gpu}=0 \tag{13}$$

M_{gpu} 可通过解式(13)求得。根据式(9)与求得的 M_{gpu} 即可求出 R_{split}。

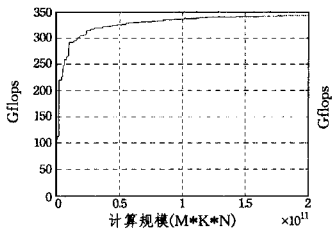

图 4　采用一次分段函数拟合的 $f_{gpu}(N_2)$

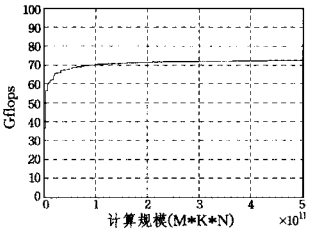

图 5　采用一次分段函数拟合的 $g_{cpu}(N_1)$

3.3　多 GPU 自适应负载均衡算法(MA 算法)

当 CPU 与 GPU 出现负载不均衡时，主要有两种情况：

情况 1　$ratio>1$，GPU 负载过重。CPU 空闲，CPU 等待 GPU。

情况 2　$ratio<1$，CPU 负载过重。GPU 空闲，GPU 等待 CPU。

空闲时间一定的情况下，由于 GPU 总体性能远大于 CPU 总体性能，情况 2 带来的性能损失更大。这种损失在单节点多 GPU 的情况更为严重。如图 6 所示，在单 GPU 情况下，由于负载不均衡带来的性能损失为 1.2%，而在 4 GPU 情况下，这个损失高达 8.1%。

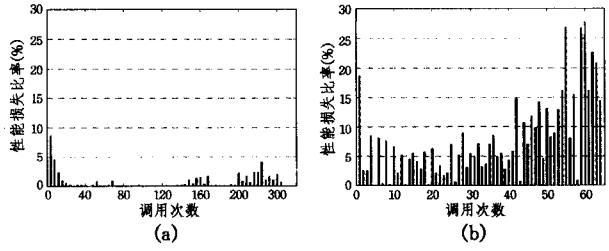

图 6　单 GPU(a)与 4GPU(b)的情况下单次 DGEMM 因负载不均衡带来的性能损失

在负载均衡中，GPU 与 CPU 不应该处于同等地位，均衡时应该尽量避免情况 2 发生。在 MA 算法中，通过式(7)中定义的 $ratio$ 修正 CPU 与 GPU 间负载，以实现情况 2 发生的概率远小于情况 1 发生的概率。

以下介绍 MA 算法的具体过程。DGEMM 过程前，判断上一次 DGEMM 过程中 $ratio$ 是否超出阈值 $threshold$。如果超出阈值，则进行修正。修正过程如下：

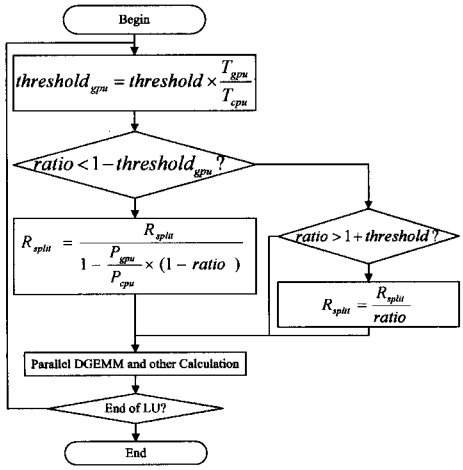

图 7　MA 算法流程图

若上一次 DGEMM 中发生情况 2，说明 R_{split} 过小，需根据 $ratio$ 以及 GPU、CPU 的性能比(P'_{gpu}/P'_{cpu})增大 R_{split}。$ratio$ 越大，R_{split} 增加量越大，(P'_{gpu}/P'_{cpu})越大，R_{split} 增量越大。

若上次 DGEMM 中发生情况 1，说明 R_{split} 过大，则需根据 *ratio* 减少 R_{split}。*ratio* 越小，R_{split} 减少量越大。具体算法流程图如图 7 所示。

需要注意的是，修正过程中可能因为 GPU 与 CPU 性能突变从而导致修正过度，使得 *ratio* 发散而不能收敛。出现这种情况时需要通过对 *ratio* 变化范围进行控制，在变化较大情况下可以进行适当限制。限制值需根据具体情况设定。

该算法使得 R_{split} 根据上次迭代计算的 *ratio* 以及 GPU 与 CPU 的性能比（P'_{gpu}/P'_{cpu}）进行调整，有效减少了情况 2 的发生，可以使得单节点内多 GPU 情况下有明显的性能提升。采用该算法实验结果进行分析。

3.4 测试平台

本文使用浪潮 MF5588 服务器平台进行实验。每个节点的主要配置如表 1 所列。

表 1 MF5588 服务器软硬件配置情况

类别	项目	型号	个数
硬件	CPU	Xeon 5675 (6 core @3.068GHz)	2
	GPU	Tesla C2070	2
	内存	Samsung 8G DDR3 1333MHz	16
软件	OS	RHEL 6.2	—
	MKL	Intel MKL 10.2	—
	Compiler	Intel ICC 12.1	—

实验主要考虑在单节点内 CPU 总的计算能力 P_{cpu} 与 GPU 总的计算能力 P_{gpu} 比不同的情况下，原版 NVIDIA HPL、采用 ED 算法的 HPL、同时采用 ED 算法与 MA 算法两种算法的 HPL 的加速效果。测试代码是在基于 NVIDIA 针对费米架构的 HPL 最新代码 hpl-2.0_feimi_v11 上进行修改优化。

3.5 性能

在 P_{gpu}/P_{cpu} 不同的情况下对上述方法进行测试。测试结果如表 2 所列。

表 2 不同 P_{gpu}/P_{cpu} 情况下的测试结果

GPU 个数	CPU 核数	内存占用率	P_{gpu}/P_{cpu}	原始 (Gflops)	ED 算法 (Gflops)	ED 算法+MA 算法 (Gflops)
1	12	80% (N=100000)	2.4	389.6	391.2	391.4
2	12	80% (N=100000)	4.8	770.7	786.1	795.4
2	6	40% (N=70000)	9.6	625.2	639.9	663.9

其中，原始算法为 NVIDIA 提供的 hpl-2.0_feimi_v11 代码中使用的算法。

3.6 数据对比与分析

单节点内 CPU 总的计算能力与 GPU 总的计算能力比（P_{gpu}/P_{cpu}）由节点硬件结构决定。当节点内 CPU 性能较强而 GPU 性能较弱时，（P_{gpu}/P_{cpu}）较小，传统的同构集群（P_{gpu}/P_{cpu}）=0；当节点内 CPU 较弱而 GPU 性能强，GPU 数量多时，（P_{gpu}/P_{cpu}）较大，单节点 4 GPU 的情况下（P_{gpu}/P_{cpu}）可以超过 11。

对表 2 数据进行分析。以 NVIDIA 原始算法为对比，比较采用 ED 算法的 HPL 以及同时采用 ED 算法与 MA 算法的 HPL 的加速效果，如图 8 所示。

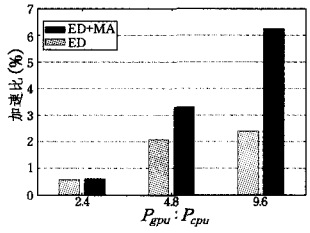

图 8 各算法在不同 P_{gpu}/P_{cpu} 时的加速效果

从图 8 可以看出，随着（P_{gpu}/P_{cpu}）的上升，采用 ED 算法与 MA 算法得到的加速比也逐渐上升。当（P_{gpu}/P_{cpu}）=9.6 时，同时采用 ED 算法与 MA 算法可达到 6.3%的加速比。

加速比变化趋势反映出 ED 算法与 MA 算法在节点内 GPU 数较多、性能较强的情况下可发挥出更大的优势。而 NVIDIA 本年底将推出的新一代开普勒架构 GPU 在性能方面将有 3 倍提升，（P_{gpu}/P_{cpu}）将进一步增大，使得上述算法将取得更好的加速效果。

结束语 本文主要分析现有 GPU 加速的 HPL 代码运行时特性，提出该代码采用的动态负载均衡算法在单节点多 GPU 情况下不能很好地实现负载均衡。针对该现象，本文提出了经验指导的动态负载均衡算法（EB 算法）与多 GPU 自适应负载均衡算法（MA 算法）。上述两种算法在单节点多 GPU 情况下，相比 NVIDIA 最新针对费米架构的 HPL 可取得 6.3%的加速比，并通过对实验数据的分析判断上述算法可在单节点多 GPU 集群环境取得较好的效果。

参 考 文 献

[1] NVIDIA CUDA Compute Unified Device Architecture Programming Guide[Z]

[2] Wang Feng, Yi Hui-zhan. Optimizing Linpack B enchmark on GPU-Accelerated Petascale Supercomputer[J]. Journal of Computer Science and Technology, 2011, 26(5): 854-865

[3] HPL-A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers[OL]. http://www.netlib.org/benchmark/hpl/

[4] Phillips E. CUDA Accelerated Linpack on Clusters, SC09[OL]. http://developer.nvidia.com/get-started-parallel-computing

[5] NVIDIA Tesla GPGPU- Introduction of tesla C2070 and C2050 [OL]. http://www.nvidia.cn/object/product_tesla_C2050_C2070_cn.html

[6] Linpack on NVIDIA GPU [OL]. http://tech.it168.com/a2010/0723/1081/000001081479_all.shtml

[7] NVIDIA. Fermi compute architecture whitepaper[Z]. 2009

[8] HPC Challenge Benchmarks[OL]. http://icl.cs.utk.edu/hpcc

(上接第 85 页)

[10] NVIDIA Corporation. NVIDIA CUDA 计算统一设备架构编程指南[M]. v2.0, 2008

[11] AMD Corporation. ATI Stream Computing OpenCL Programming Guide[M]. 2010

[12] 贾海鹏，张云泉，龙国平，等. 基于 OpenCL 的拉普拉斯图像增强算法优化研究[J]. 计算机科学，2011，39(5)

[13] 颜深根，张云泉，龙国平，等. 基于 OpenCL 的规约算法优化[J]. 软件学报，2011

[14] 张樱，张云泉，龙国平. 基于 OpenCL 的图像模糊化算法优化研究[J]. 计算机科学，2012，39(3)：260-264