单节点多GPU集群下HPL动态负载均衡优化*

陈任之[1+]，黄立波[2]，陈顼颢[3]，王志英[4]

[1,2,3,4] (国防科学技术大学 计算机学院,长沙 410073)

Optimizing HPL Benchmark on Multi-GPU Clusters*

CHEN Ren-Zhi[1+], Huang Li-Bo[2], Chen Xu-Hao[3], Wang Zhi-Ying[4]

[1,2,3,4] (School of Computer National University of Defense technology, Changsha 410073, China)

+ Corresponding author: Phn: +86-13786134892, E-mail: chenrenzhi1989@gmail.com

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 proposes an experience-based dynamic load balancing algorism and a multi-GPU adaptive load balance algorithm. Besides, the article tests these two algorisms multi-GPU cluster and achieves a 6.3% acceleration compared with the latest NVIDIA's HPL accelerated by GPU.

Key words: HPL; GPU; dynamic load balancing algorithm

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,本文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平台上进行验证,对比现有基于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

随着半导体工艺发展,单处理器芯片内集成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.本文分析与优化目标为基于 NVIDIA 费米 GPU 的 HPL.

* Supported by the National Natural Science Foundation of China under Grant No. 61070037 and No. 61103016 (国家自然科学基金)

作者简介:陈任之,男,硕士,研究方向为高性能计算与体系结构;黄立波,男,博士,讲师,研究方向为微处理器体系结构;陈顼颢,男,博士,研究方向为微处理器体系结构;王志英,男,博士生导师,教授,研究方向为计算机体系结构.

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]提出通过一种基于实际运行性能指导的动态负载均衡算法,根据 HPL 运行时 GPU 与 CPU 的实际性能调整两者间的负载.现有基于 NVIDIA 费米 GPU 的 HPL 代码采用该算法.

然而在单节点多 GPU 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 的情况下的性能损失比 CPU 等待 GPU 的情况下更为严重.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反映出不同规模下 GPU 实际性能,尤其在 GPU 性能变化剧烈情况下,该算法对比原算法更接近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,有效使得 GPU 等待 CPU 的概率降低.与 NVIDIA 原算法相比,采用 ED 与 MA 算法时可获得 6.3%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.第 3 节介绍了针对单节点多 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.第 5 节总结全文.

1 算法分析

1.1 HPL算法

HPL 是通过求解一个稠密线性方程组来测试集群计算性能[3].如(1)式所示:

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

其中 $A = (a_{ij})_{N\times N}$, $b = (b_1, b_2, \cdots, b_N)^T$, $x = (x_1, x_2, \cdots, x_N)^T$,A 与 b 均已知,x 为待求向量.HPL 求解(1)式采用了 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 分解过程时间复杂度为 $O(N^3)$,回代求列向量 x 过程时间复杂度为 $O(N^2)$,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:

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

其中 A_{ij} 为已知矩阵块, L_{ij} 与 U_{ij} 为待求矩阵块.由(2)式可得:

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

因此求 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(3)式求解出 L_{11} 与 U_{11} 的基础上, L_{21} 与 U_{12} 亦可由(2)式求出:

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

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

由(2)式可得:

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

因此求解求 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(6)式中求出矩阵

$A_{22} - L_{21} \times U_{12}$ 过程.该过程通过调用双精度矩阵乘加函数(DGEMM)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,与 C_c 组成完整矩阵 C.

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:

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

其中 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.

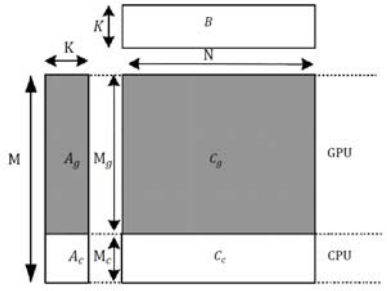

Fig.1 Task division between GPU and CPU
图 1 异构平台下 GPU 与 CPU 的任务划分

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 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:

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

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

其中 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性能提升,现有基于 NVIDIA 费米 GPU 的 HPL 已使用该算法.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,主要特点有如下两点:

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 4 GPU 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 4 GPU 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(右)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.

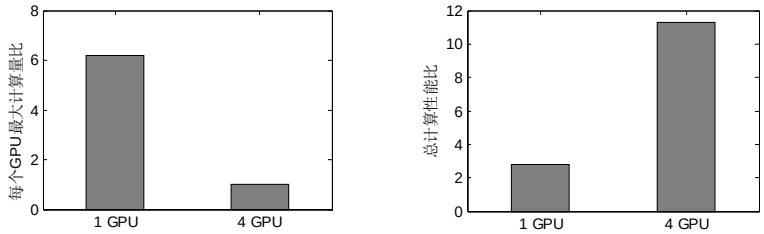

Fig.2 HPL features on multi-GPU cluster

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)

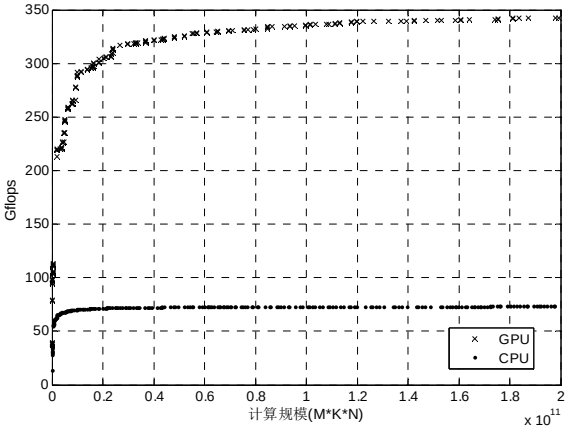

Fig.3 Performance curve of CPU and GPU in different calculating scales

图 3 GPU 与 CPU 在不同规模下的性能曲线

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} 情况下性能较为稳定,而在浮点运算次数小于 1×10^{11} 情况下性能明显下降[6]. CPU 在浮点总量小于 5×10^{9} 情况下才明显下降.

HPL 是一个递归求解过程[7],其计算规模不断减小.多 GPU 情况下,单个 GPU 总计算量较小,当计算规模减

少时,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 性能估算本次 GPU 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 性能与规模关系后通过函数拟合获得.

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 $g_{cpu}(N_1)$ 与 $g_{cpu}(N_1)$,如图4与图5.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 $g_{cpu}(N_1) = k_{cpu}^i \times N_1 + b_{cpu}^i$、GPU 性能-规模函数的第 j 段直线方程为 $f_{gpu}(N_2) = k_{gpu}^j \times N_2 + b_{gpu}^j$,则有:

$$\frac{M_{gpu}}{M} = \frac{k_{gpu}^j \times M_{gpu} + b_{gpu}^j}{k_{gpu}^j \times M_{gpu} + b_{gpu}^j + k_{cpu}^i \times (M - M_{gpu}) + b_{cpu}^i} \tag{12}$$

其中 M 已知, M_{gpu} 为待求量.整理(12)式可得一元二次方程:

$$(k_{gpu}^j - k_{cpu}^i) \times M_{gpu}^2 + (b_{gpu}^j + b_{cpu}^i + k_{cpu}^i \times M - k_{gpu}^j \times M) \times M_{gpu} - M \times b_{gpu}^j = 0 \tag{13}$$

M_{gpu} 可通过解(13)式求得.根据(9)式与求得的 M_{gpu} 即可求出 R_{split} .

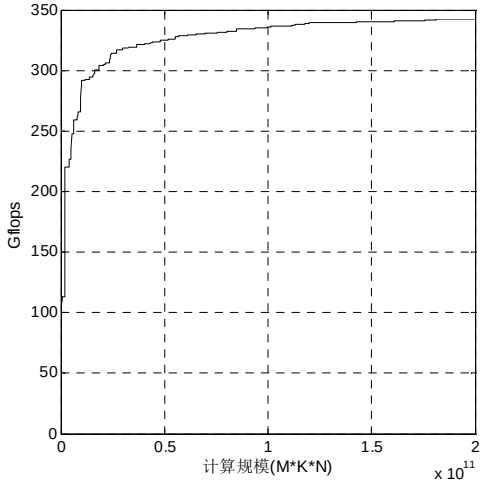

Fig.4 Fitting $f_{gpu}(N_2)$ by piecewise linear function

图 4 采用一次分段函数拟合的 $f_{gpu}(N_2)$

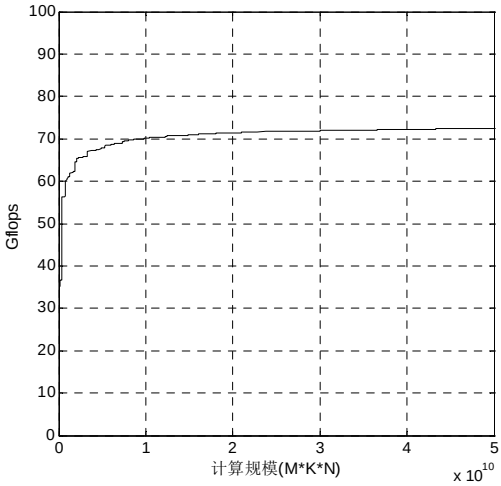

Fig.5 Fitting $g_{cpu}(N_1)$ by piecewise linear function

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,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 情况这个损失高达 8.1%.

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(7)

式中定义的 ratio 修正 CPU 与 GPU 间负载,以实现情况 2 发生概率远小于情况 1 发生的概率.

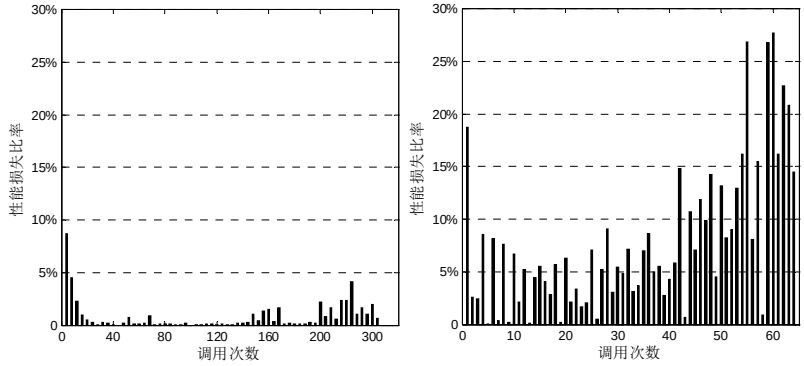

Fig.6 Performance drawback due to load imbalance on clusters with single GPU(left) and 4 GPUs(right)
图 6 单 GPU(左)与 4 GPU(右)的情况下单次 DGEMM 因负载不均衡带来的性能损失

下述介绍 MA 算法具体过程.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:

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.

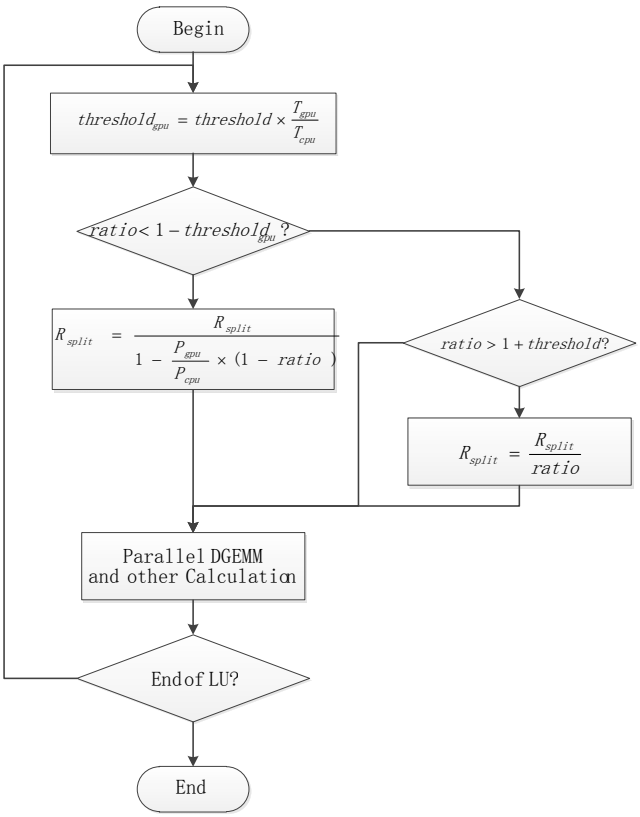

Fig.7 MA algorism flow chart
图 7 MA 算法流程图

需要注意的是修正过程中可能因为 GPU 与 CPU 性能突变会导致修正过度,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,当变化较大情况下可以进行适当限制.限制值需跟具体情况设定.

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与分析.

3.4 测试平台

本文使用浪潮 MF5588 服务器平台进行实验.每个节点主要配置如表 1:

Table 1 Software and hardware configuration of MF 5588

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} 比不同情况下,对比原版 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 所示.

Table 2 Test results under different P_{gpu} / P_{cpu}

表 2 不同 P_{gpu} / P_{cpu} 情况下测试结果

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.

从图 8 可以看出,随着(P_{gpu} / P_{cpu})的上升, 采用 ED 算法与 MA 算法得到加速比逐渐上升.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.

加速比变化趋势反应出 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加速效果.

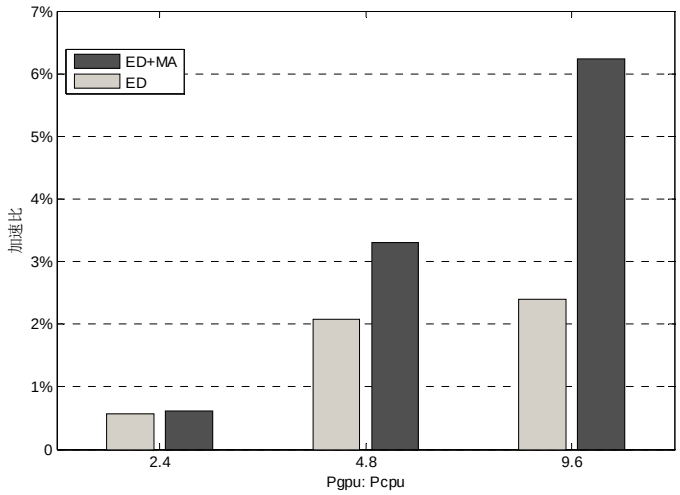

Fig.8 Acceleration of different algorism under different P_{gpu} / P_{cpu}

图 8 各算法在不同 P_{gpu} / P_{cpu} 时的加速效果

4 结论

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,可取得比 NVIDIA 最新针对费米架构的 HPL 有 6.3%的加速比,并通过对实验数据分析判断上述算法可在单节点多 GPU 集群环境取得较好效果.

致谢 感谢窦勇老师以及郭松博士的指导和帮助.

References:

[1] NVIDIA CUDA Compute Unified Device Architecture Programming Guide

[2] WANG Feng, YI Hui-Zhan: Optimizing Linpack Benchmark on GPU-Accelerated Petascale Supercomputer. Journal of Computer Science and Technology, 2011,V26(5): 854-865

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

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

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

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

[7] NVIDIA, “Fermi compute architecture whitepaper,” 2009.

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