软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2011,22(12):2853−2865 [doi: 10.3724/SP.J.1001.2012.04057]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

异构系统硬件故障传播行为分析及容错优化*

贾 佳+, 杨学军

(国防科学技术大学 计算机学院 并行与分布处理国家重点实验室,湖南 长沙 410073)

Propagation Behavior Analysis and Fault Tolerance Optimization of Hardware Fault in Heterogeneous Systems

JIA Jia+, YANG Xue-Jun

(National Key Laboratory for Parallel and Distributed Processing, College of Computer, National University of Defense Technology, Changsha 410073, China)

+ Corresponding author: E-mail: morpheux@163.com

Jia J, Yang XJ. Propagation behavior analysis in software and fault tolerance optimization of hardware fault in heterogeneous systems. *Journal of Software*, 2011,22(12):2853−2865. http://www.jos.org.cn/1000-9825/4057.htm

Abstract: Based on the inter-procedural dependence analysis, this paper studies the propagation behavior in software of hardware fault in heterogeneous systems. This research can be used for optimizing application-level checkpointing techniques. Experimental results demonstrate that this method is viable and can be very helpful for the research of fault tolerance optimization of heterogeneous systems.

Key words: general purpose GPU; heterogeneous system; inter-procedural dependence analysis; propagation behavior; fault tolerance optimization

摘 要: 以异构系统的过程间相关性分析为基础,研究分析异构系统硬件故障在软件之中的传播行为,指导优化基于异构系统的应用级 checkpointing 检查点保存问题,并通过实验验证其可行性及性能,对异构系统的容错优化研究具有重大意义.

关键词: 通用 GPU;异构系统;过程间相关性分析;传播行为;容错优化

中图法分类号: TP311 **文献标识码**: A

并行计算是实现超高性能计算的主要技术手段.当前,随着 GPGPU(general purpose computation on graphic processing units)[1]性能的不断提高,利用 CPU 和 GPU 构建的异构系统已经成为高性能计算机[2,3]领域的研究热点,我国首台千万亿次计算机天河[4]就是一个典型案例.

然而,随着并行计算系统规模的不断增长,系统可靠性越来越低,已成为并行计算向大规模扩展的一个不容忽视的制约因素.由于商用 GPGPU 容错能力较弱,所以由 CPU 和 GPU 构建的大规模异构并行系统的可靠性问题更为尖锐,尚缺乏实用的容错手段,本文针对这一现实难题展开研究,创新点主要包括:

* 基金项目: 国家自然科学基金(60921062, 61003087)
收稿时间: 2011-02-25; 定稿时间: 2011-04-28; jos 在线出版时间: 2011-09-09
CNKI 网络优先出版: 2011-09-09 13:54, http://www.cnki.net/kcms/detail/11.2560.TP.20110909.1354.001.html

© 中国科学院软件研究所 http://www.jos.org.cn

(1) 基于过程间相关性[5]理论,提出了由CPU和GPU构成的异构系统中硬件故障在软件中传播行为描述方法,我们称其为故障传播模型;

(2) 根据故障传播模型,设计了针对该系统的checkpointing[6,7]机制,并针对影响checkpoint/restart开销的主要问题之一——checkpoint保存数据量进行了优化;

(3) 我们以CUDA[8]为实验平台,通过6个典型的应用程序,验证本文提出的针对checkpoint保存数据量的优化方法可以有效地减小开销,提高容错性能.

本文第1节通过案例介绍并定义CG调用和CG调用流图,讨论异构系统故障传播与CG调用流图的关系.第2节通过分析CG间故障传递和G点内故障传递对异构系统的故障传播行为进行描述,并推导出其数据流方程.第3节实现异构系统的应用级checkpointing技术,并基于故障传播行为分析对checkpoint数据保存进行优化.第4节通过实验验证评估优化性能.第5节介绍相关工作.第6节总结全文.

1 问题提出

1.1 CG调用

对异构系统来说,目前比较热门的几类异构系统编程模型[9]主要包括AMD公司的Brook+[10]、Nvidia公司的CUDA和业界制定的统一编程标准OpenCL[11].为不失一般性,本文在描述编程模型时将各公司引入的面向其特定体系结构的细节特征隐去,只抽取异构编程模型中CPU对GPU调用时的相关特性.本文选取CUDA为研究对象,因此文中都统一采用CUDA专用术语[12]来描述.

图1给出了同构与异构两种系统下算法执行流程.在同构系统中,所有的调用都在CPU上执行,通过对多个子函数的调用来实现计算的分段执行.在异构系统中,程序分为CPU运行代码和GPU运行代码两部分,即流级代码和核心级代码两部分[13].流级代码运行于CPU上,负责组织数据,配置并调用核心(kernel)函数的执行;而核心级代码则全部运行于GPU上,负责具体的计算.如图2所示,这是一个CUDA上的矩阵乘算法,具体的$c=a\times b$矩阵乘计算都是由GPU计算完成,其中,a,b,c都是$n\times n$的矩阵.在CPU端的算法代码中,我们将a,b,c表示为ac,bc,cc.首先,CPU通过Read操作将ac和bc传送至GPU存储系统的对应变量a和b;其次,CPU调用Kernel函数执行计算,GPU为Kernel函数启动$n\times n$个线程来进行计算,每个线程负责计算c中的一个元素[14];最后,CPU通过Write操作将GPU端矩阵c传送至CPU端的矩阵cc中,作为整个计算段的结果.

在这一流程中,如果我们将运行在CPU上的流程序看作为主控部分,那么运行在GPU上一个kernel看作是一个子过程,从控制流角度来看,CPU上的流程序通过一种特定的手段把数据传递给GPU,并启动GPU上的kernel程序.这一过程和串行程序的子程序调用既有类似的地方也有不同,本文把这一过程称作CG调用(CPU调用GPU上的程序).

定义1(CG调用). 在CUDA中,包含有拷贝传递参数保留字*cudaMemcpy*(…,*HostToDevice*),对声明好的kernel分配线程并进行执行的保留字*kernelName*⟨⟨⟨…⟩⟩⟩(…)以及拷贝传回参数保留字*cudaMemcpy*(…,*DeviceToHost*)这3点构成的程序序列称为CG调用序列.*cudaMemcpy*(…,*HostToDevice*)称作为CG参数传递点,*kernelName*⟨⟨⟨…⟩⟩⟩(…)称为CG调用点,*cudaMemcpy*(…,*DeviceToHost*)称作CG调用返回点.

从广义上来看,所谓CG调用就是CPU启动GPU计算的过程,我们把这种关系称作为CG调用关系.对于CPU端产生这一调用的保留字语句称为调用点;而对于GPU的每段计算过程有效计算语句的出口入口,称为kernel的出口点入口点.

参照定义1,CG调用也可以推广到其他程序设计模型.在CUDA编程模型上,CG调用的传递参数并调用计算的程序流程与传统同构系统在语法上是有差别的,其语义与同构系统在参数传递的行为上也是有很大区别的.在同构系统上进行过程调用时,参数的传递不需要对数据进行拷贝而是直接进行引用的.通过图2的程序代码可以看出,CUDA程序在CG调用时,主程序首先在CPU端初始化矩阵ac和bc,并事先为ac和bc的全部元素在GPU上分配好地址空间,再将ac和bc的全部元素在kernel进行有效计算之前全部拷贝到GPU中,最后由CPU调用执行计算.由此可见,CG调用时的参数传递是通过数据拷贝的传值方式来实现的,我们称其为CG传值

© 中国科学院软件研究所 http://www.jos.org.cn

方式.

虽然这种利用拷贝来传递参数并调用 kernel 的行为方式不同于传统的过程调用,但是其实现效果是一样的.因此我们可以认为,CG 调用的行为与传统过程调用行为的作用是等效的,那么同样也可以使用针对过程调用的过程间分析方法来对 CG 调用的数据流传播行为进行分析.

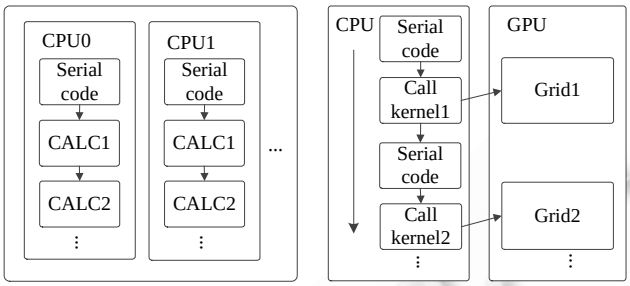

Fig.1 Program process on homogeneous system and heterogeneous system

图 1 同构与异构系统算法执行流程

```
//Main on CPU
clock_t matmultCUDA()
 {
  float *ac, *bc, *cc;  // array on the CPU
  clock_t start, end;
  start=clock();
  // allocate space on the GPU
  cudaMalloc((void**) & ac, sizeof(float)*n*n);
  ...
  // transfer array to GPU
  cudaMemcpy2D(ac,sizeof(float)*n, a,
    sizeof(float)*lda, sizeof(float)*n, n,
    cudaMemcpyHostToDevice);
  ...
  // execute the kernel
  matMultCUDA<<<blocks*n,
     NUM_THREADS>>>
     (ac,n,bc,n,cc,n,n);
  // transfer stream to CPU
  cudaMemcpy2D(c,sizeof(float)*ldc, cc, ...
    cudaMemcpyDeviceToHost);
  ...
 }
```

Data transfer

```
//Kernel on GPU
__global__ static void
  matMultCUDA(const float* a, size_t lda,
  const float* b, size_t ldb, float* c,
  size_t  ldc, int n)
  {
     const int tid=threadIdx.x;
     const int bid=blockIdx.x;
     const int idx=bid*blockDim.x+tid;
     const int row=idx/n;
     const int column=idx%n;
     int i;
     if (row<n && column<n) {
        float t=0;
        for (i=0; i<n; i++) {
           t+=a[row*lda+i]*b[i*ldb+column];
        }
        c[row*ldc+column]=t;
     }
```

Fig.2 Matrix multiply on CUDA

图 2 CUDA 上的矩阵乘算法

1.2 CG调用流图

在过程间分析问题中,描述程序中过程间调用关系的图称为调用流图.构建一个过程间调用流图可以直观明了、准确细致地表现出过程间调用关系,利于进行复杂的过程间分析.因此,为了更为直观、准确地对 CG 调用关系进行分析,我们基于过程间调用流图的构建方法,为异构系统程序构造了 CPU-GPU 间调用流图.

定义 2(CG 调用流图). 异构系统中,CPU 端流级程序中包含有 CG 调用序列的程序段为 CPU 端节点,记作 *C* 点节点,GPU 端核心级程序中每个 kernel 都为一个 GPU 端节点,记作 *G* 点节点.其中:连接 *C* 点内各节点,表示顺序执行的连线称为 *C* 点内边;而连接于 CPU 与 GPU 之间,用来表示 *C* 点与 *G* 点节点之间 CG 调用关系的连线称为 CG 调用边.那么,这种由节点与边构成的、描述异构程序中 CG 调用关系的数据流图称为 CG 调用流图,

© 中国科学院软件研究所 http://www.jos.org.cn

记作 $G=\langle C,G,E\rangle$.其中,C 为调用节点,G 为被调用的 kernel 节点,E 为表示这一 CG 调用边.

在调用流图中,我们将 CPU 端的节点标记为 C 点内的节点,则数据流在 CPU 端的传递称为 C 点内数据流传递;而 GPU 端的节点标记为 G 点内的节点,则数据流在 GPU 端的传递称为 G 点内数据流传递.那么,数据流在 CPU 与 GPU 之间的传递称为 CG 间数据流传递.在 CG 调用流图中,C 点节点只作为调用节点对 G 点内节点进行调用,G 点节点只能作为被调用节点且不存在嵌套调用.因此,G 点节点之间不存在边,整体调用关系比较简单.而本文所采用的 CUDA 语言是每次进行 kernel 调用时都是带变量名称的恒定过程的语言,因此构造 CG 调用流图并不复杂.我们只需检查程序中所有的过程体,即主程序中含有 CG 调用序列的程序段和全部的 kernel,为其中每个 CG 调用点加入一条从调用节点 C_1 到被调用的 kernel 节点 G_1 的边 E_1,记作$\langle C_1,G_1,E_1\rangle$.

以 CUDA 平台的 Swim 算法为例,我们为其构建了一个 CG 调用流图.传统的 Swim 算法主要由 4 个子过程构成(INITIAL 初始化过程除外),所有子过程都在 CPU 中串行调用执行,而在 CUDA 平台下,每个子过程作为一个 kernel 放入 GPU 进行计算,CPU 中只有 INITIAL 过程负责数组变量的初始化,并在每次调用 kernel 计算前对变量信息进行 GPU 端地址分配及拷贝传值.其中,Calc3ZKernel 只在主体循环的第一次迭代中调用一次,而其余 kernel 每次循环迭代时都会被调用.

在图 3 所示的 CG 调用流图中,我们构造了 7 个节点和 7 条边,其中,CPU 内有 3 个节点和 3 条 C 点内边,GPU 内 4 个节点,并且 C 点节点和 G 点 kernel 节点一一对应的调用关系都由它们之间的 4 条 CG 调用边清楚地标记出来.由此可见,除了节点标记存在差异外,CG 调用流图与传统过程调用流图是没有区别,完全等效的,且由于异构编程模型决定了 CG 调用中不存在传统过程调用中的嵌套调用,因此构造 CG 调用流图更为简单.

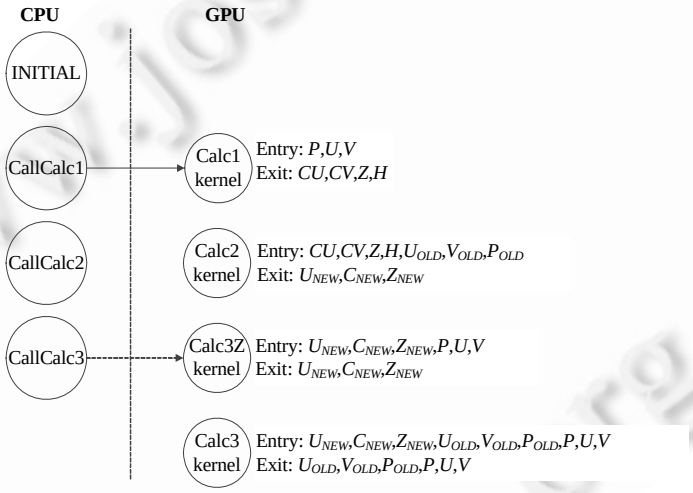

Fig.3 CG-CFG of Swim

图 3 SWIM 异构系统 CG 调用流图

1.3 异构系统的故障传播与CG调用流图

在 Swim 算法中,有 14 个二维数组变量和少量标量.图 3 列出了 GPU 中每个 kernel 出口/入口点处的所有数组变量.可以看出数据在 kernel 内的传播.那么,根据标记出的 CG 调用关系并列出调用点 CallCalc1 和与其对应的调用返回点的变量信息,则可以更为全面地描述这一计算段在 CG 间数据传播过程,如图 4 所示. CPU 通过 CallCalc1 对放入 GPU 的 Calc1Kernel 进行调用,首先将 CPU 端变量 *cpuP*,*cpuU* 和 *cpuV* 传值到 GPU 的变量 P,U,V 中,然后在计算段 Calc1Kernel 上,由传值到 kernel 入口点的变量 P,U,V 参与计算,在计算段的最后得到结果变量 CU,CV,Z 和 H,最后通过 kernel 出口点将结果传回给 CPU 主过程中的变量 *cpuCU*,*cpuCV*,*cpuZ* 和 *cpuH*,并继续由主过程将这些变量作为下一调用点 CallCalc2 处的实际参数传递给被调用的 Calc2Kernel 进行计算.以此类推,在主处理器串行调用过程中,Calc3Kernel 将最终计算结果传回 CPU 上的主过程.由此可见,CG 调用流图所显

© 中国科学院软件研究所 http://www.jos.org.cn

示的调用过程其实就是对这一算法数据流过程的完整描述.

基于此数据流过程,当变量信息在计算过程中被修改而产生变化时,由于被修改的变量与其他变量之间存在相关性,根据变量活跃性分析,其后,所有与这一变量存在依赖关系的计算结果及变量信息都将受到影响,并不断传递,这种行为就是数据流传播行为.同理,当系统出现硬件故障并导致在软件中出现错误时,在程序的计算及通信过程中都可能导致变量出现错误,其影响也将由活跃变量基于这一行为不断传播.在存在过程调用的程序中,故障由于过程间存在的相关性而在过程之间传播,因此可以说,在存在过程调用的数据流过程中故障是沿着调用流图传播的,那么异构系统的故障则是沿着 CG 调用流图传播的.

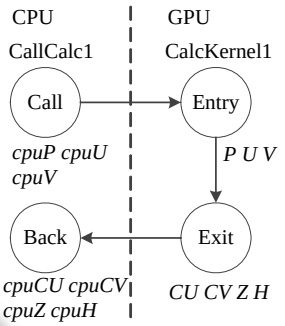

Fig.4 Data propagation in CG

图 4 CG 间的数据传播

根据这一结论,在 CG 调用流图中,故障不仅在 CPU 或 GPU 一端传播,还可以通过 CG 调用或 kernel 计算结果返回在 CPU-GPU 间传播.因此,CG 故障传播行为又分为 *C* 点内故障传播、*G* 点内故障传播和 CG 间故障传播.为了对故障传播行为问题进行更为具体的描述,我们需要分析故障在数据流中传播过程中的影响,即故障影响集合.

定义 3. 当系统出现故障时,由于相关性和活跃性导致其在之后的数据流中传递,那么在此故障点后的数据流过程中,任意一点处所由于故障影响而出现错误的变量集合,称为故障影响集合.

本文的主旨是分析异构系统的硬件故障在软件之中的传播行为,其主要面向由于故障导致计算过程中变量信息和参数出现的错误,而不考虑地址跳转错误.因此,文中主要针对数据流的故障而非控制流的故障进行分析.我们采用过程间分析方法,基于 CG 调用流图对故障传播的数据流进行分析,将 *C* 点内的调用点返回点以及 *G* 点内 kernel 出口/入口点作为关键点进行故障影响集合的求解,并以此对 CG 间故障传播和点内故障传播的行为进行总结.这一数据流分析方法对于瞬时故障或固定性故障都没有依赖关系,所以对这两种故障类型都适用.

2 异构系统的故障传播行为分析

故障在数据流中是依赖于活跃变量来传播的,因此,为了分析异构系统硬件故障在软件中的传播行为,必须基于 CG 调用流图.首先判断软件中所有用到的变量在 CPU 或 GPU 内以及 CPU 与 GPU 之间是否活跃,进而判定出这些由于故障引起的错误哪些会在 CPU 和 GPU 之间传播,哪些在单一芯片内传播,并通过求解 CG 调用中各个关键点的故障影响集合,总结出完整的异构系统故障传播行为.

为了利于数据流方程的形式化描述,根据 CG 调用流图以及 CG 间各节点间调用关系的分析,我们为 CG 间各关键点的故障影响集合进行定义.

定义 4. 对于给定的主过程 *P* 和过程调用点 *S*,故障影响调用集 *ErrorCall*(*S*)即指在 *S* 处被调用的所有由于故障影响而出现错误的变量集合.

定义 5. 对于给定的主过程 *P* 和返回点 *S*,故障影响返回集 *ErrorBack*(*S*)就是指由被调用 kernel 传值到返回点 *S* 处的所有由于故障影响而出现错误的变量集合.

定义 6. 对于过程(kernel)*P* 和传递给 *P* 的参数,入口故障影响集 *ErrorEntry*(*S*)是指在 kernel 入口点 *S* 处所有传递过来的由于故障影响而出现错误的变量集合.

定义 7. 对于过程(kernel)*P* 和 *P* 将要传递出去的参数,出口故障影响集 *ErrorExit*(*S*)即指在 kernel 出口点 *S* 处所有传递出去的由于故障影响而出现错误的变量集合.

2.1 CG间故障传播

在 CPU 对 GPU 进行传值调用时,如果存在故障影响集合,那么由故障引起的错误会从 CPU 传入 GPU 中,而当 GPU 对 CPU 传回数据时也是同理,这样就构成了 CG 间故障传递行为.本节主要针对 CPU 调用点到 GPU 的 kernel 入口点,以及出口点到 CPU 主过程返回点处的故障传播行为进行讨论.

根据 CG 调用定义,在 CPU 调用 kernel 时是通过传值实现参数传递的.传值到 GPU 中的变量名称与 CPU

© 中国科学院软件研究所 http://www.jos.org.cn

不同,且指向的地址空间是GPU上的,变量信息在kernel入口点也没有被重新赋值,因此与传统活跃变量分析不同.由于CG调用的特殊性,我们依然认为这些变量在CPU上的调用点处都是活跃的,变量信息是可传递的.同理,从kernel出口点处将结果变量信息传递回CPU也是如此.

我们将这一行为称为CPU-GPU调用数据传递,简称CG调用数据传递.为了利于表述,我们对其进行定义.

定义8. 对于给定的点S、过程P和通过传值传递给P的参数集合X,CG调用数据传递$CG_Trans(S,X)$就是在S点处所有因为CG传值而传递X的变量集合.

那么,我们根据数据流分析方法对CPU-GPU芯片间的故障传递进行分析,求导从CPU调用点到GPU的kernel入口点处故障传播行为.

给定调用点S,kernel入口点和调用点处的故障影响调用集$ErrorCall$,定义:

- $KILL(S)$=通过调用点S进入kernel时被注销(重新赋值)的所有变量集合,由于调用过程的传值不存在重新赋值,因此这里$KILL(S)$为∅;
- $Call(S)$=由于调用而通过过程调用点S的所有变量集合;
- $Entry(S)$=通过kernel入口点的所有变量集合;
- $CG\_Trans(S,X)$=对于调用点S,由于CG传值而给被调用kernel传递参数X的所有变量集合;
- $ErrorTrans(S)$=在CG调用中,对于给定调用点S在调用传值时存在于CPU-GPU之间的由故障引起错误的变量集合.

对于传统过程调用,给定被调用过程入口点S,如果S_p是调用点,那么,

$$Entry(S)=Call(S_p) \tag{2.1}$$

而对于CG调用,给定调用点S和与其对应的kernel入口点,在没有故障的情况下,根据定义8

$$Entry(S)=CG\_Trans(S,X) \tag{2.2}$$

入口点上所有变量都是调用点S处变量集合传值过来的,那么当存在故障时,根据公式(2.2),我们可以得出从过程调用点S到kernel入口点处的故障影响集合的数据流方程:

$$ErrorEntry(S)=(ErrorCall(S)\cap CG\_Trans(S,X))\cup ErrorTrans(S) \tag{2.3}$$

同理,给定返回点B和kernel出口点的故障影响集$ErrorExit$,定义:

- $CG\_Trans(B,X)$=对于返回点B,由于CG传值而从被调用kernel传递参数X回到返回点的所有变量的集合;
- $ErrorTrans(B)$=在CG调用中,对于给定返回点B在结果返回传值时存在于CPU-GPU之间的由故障引起错误的变量集合;
- $ErrorBack(B)$=由kernel传回并通过返回点B的故障影响集合.

那么,由kernel出口点至CPU返回点的故障影响集合数据流方程为

$$ErrorBack(B)=(ErrorExit(B)\cap CG\_Trans(B,X))\cup ErrorTrans(B) \tag{2.4}$$

由于在异构系统中GPU计算时不对CPU产生影响,因此CPU端从调用点到返回点路径上的故障始终传递,如果$ErrorCall\neq\varnothing$,那么根据公式(2.4),返回点的故障影响集合完整表示为

$$ErrorBack(B)=(ErrorExit(B)\cap CG\_Trans(B,X))\cup ErrorTrans(B)\cup ErrorCall(S) \tag{2.5}$$

其中,S是调用点.由此可见,公式(2.3)和公式(2.5)所得出的GPU上kernel入口点和CPU主过程返回点的故障影响集合数据流方程,即是对故障在CPU,GPU之间的传播行为的完整描述.

2.2 G点内故障传递

在进行一个kernel计算的过程中,如果主过程中没有分配计算,那么CPU内的变量信息是不变的,主过程中所有变量信息的更新都是由kernel计算结束后的由GPU将结果传回返回点实现的,那么故障在点内的传播行为也只可能出现在CPU的调用点和返回点而不会出现在过程中间.因此,对于点内故障传播行为的分析,我们主要针对G点内故障的传播进行讨论,即当计算过程中出现故障时是如何传播的.由于每个kernel都可以看作是一个被CPU调用的子过程,所以这也可以看成是一个过程内问题,不必考虑控制流.且其中变量的修改信息依赖

© 中国科学院软件研究所 http://www.jos.org.cn

关系判断并不明确,为一个流不敏感问题.我们针对 GPU 中 kernel 出入口点数据流传播行为进行分析,求解 kernel 入口和出口故障影响集.

在 kernel 内,数据流传递同样依赖于活跃变量,故障传播也是如此.由于判断数据流中变量活跃性取决于其是否被使用和修改,而 kernel 中数据流的使用和修改副作用都是由表达式的运算引起的,因此故障也是由表达式传播的.下面对依赖计算而传递的故障进行分析.

如图 5 所示,已知 kernel 以及它的入口点和出口点,在这里,我们可以将其看作一个基本块 S,其中的表达式语句为 L_1,L_2.首先由入口点传入变量 A,B,F,过程内 L_1,L_2 的运算包含 A~F 这 6 个变量,最后由出口点传出变量 E,D,F,则分析它们的相关性.在表达式 L_1 中,变量 A 和 B 参与计算 C 和 D,那么它们在入口点到基本块 S 的路径上被使用,$A\in USE(S)$且 $B\in USE(S)$;而 C 和 D 被修改,那么 $C\in MOD(S)$且 $D\in MOD(S)$.而在后面的表达式 L_2 中,C 和 D 又被变量 E 使用,E 被修改,则 $C\in MOD(S)\cup USE(S)$;D 同理,且 $E\in MOD(S)$.

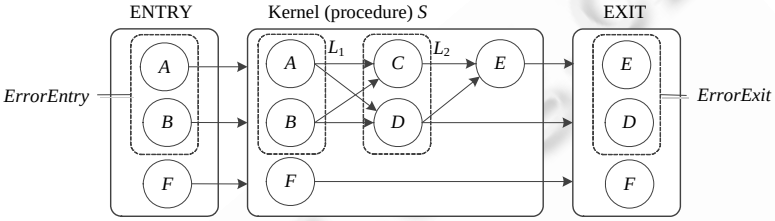

Fig.5 Error propagation in kernel

图 5 Kernel 内故障传递

通过分析可以看出,在基本块内,A~E 这 5 个变量都是相互依赖并存在相关性的,根据活跃变量分析,变量 A 和 B 在入口点到基本块 S 中的表达式路径上没被重新赋值且被使用,那么它们在入口点处是活跃的.如果 $D\in ErrorExit(S)\wedge E\in ErrorExit(S)$,那么被 D 和 E 所使用过的变量 A,B 也有可能携带或导致故障的传播;而由于其在入口点的活跃性,则 $A\in ErrorEntry(S)\wedge B\in ErrorEntry(S)$.最后一个变量 F 从基本块 S 入口开始一直到出口的路径上都没有参与任何计算,既没被使用也没被修改或注销,而是直接进入出口点输出,因此对于当前基本块 S 来说,我们认为它是不活跃的,不存在传递故障的行为,可以不予关注.

根据数据流分析方法对 kernel 中的故障传播行为进行分析,我们求导从 kernel 入口点到出口点的故障传递行为.

给定 kernel,基本块 S 及其出入口点的故障影响集——$ErrorEntry$ 和 $ErrorExit$ 在前文已定义过.在这里,$ErrorEntry(S)$为基本块 S 执行前的 kernel 入口点的故障影响集合,$ErrorExit(S)$为基本块 S 执行后的 kernel 出口点的故障影响集合.

- $RELY\_MOD(S)$=基本块 S 中表达式执行结束时产生的故障影响集合;
- $RELY\_KILL(S)$=基本块 S 中表达式执行结束时由于重新赋值而被消除的故障影响集合.

如果基本块 S 中没有与故障传播行为相关的表达式,则:

- $RELY\_MOD(S)=\varnothing$;
- $RELY\_KILL(S)=\varnothing$.

那么,综上可得出基本块 S 执行前,kernel 入口点处的故障影响集合:

$$ErrorEntry(S)=RELY\_MOD(S)\cup(ErrorExit(S)-RELY\_KILL(S)) \tag{2.6}$$

如果 S_p 是基本块 S 执行后的过程,则,

$$ErrorExit(S)=ErrorEntry(S_p) \tag{2.7}$$

其中,S_p 是基本块 S 的后继.根据公式(2.6)和公式(2.7),我们可以得出 kernel 入口点到出口点之间故障传播的数据流方程:

$$ErrorEntry(B)=RELY\_MOD(B)\cup(ErrorExit(B)-RELY\_KILL(B)) \tag{2.8}$$

© 中国科学院软件研究所 http://www.jos.org.cn

$$ErrorExit(B)=\bigcup_{B_p\in Succ(B)} ErrorEntry(B_p) \tag{2.9}$$

其中,B_p 和 B 代表基本块,$ErrorExit(B)$代表基本块 B 出口点的故障影响集,$ErrorEntry(B)$代表基本块 B 入口点的故障影响集.由此可见,G 点内故障传播行为的数据流方程与经典的传统过程内活跃变量传播的数据流方程是基本一致的,因此必为收敛的,不用再对收敛性进行证明.

3 异构系统容错技术

3.1 应用级Checkpointing技术

针对异构系统硬件故障在软件中实现容错[15,16]的需求,我们选取了同构系统中最为常用的 checkpointing 技术作为容错手段应用于 CUDA 平台,并重点对 GPU 进行容错处理.

Checkpointing 是一种广泛应用于大规模科学计算领域中的容错技术[17,18],该技术是在程序执行期间将计算状态周期性保存到可靠存储器上.如果某个进程失效,所有进程都必须回滚到最近一个检查点处继续计算.Checkpointing 分为两种基本的方法:系统级和应用级 Checkpointing.系统级 checkpointing 将应用状态在存储器上进行映像(core-dump-style snapshots),这种方法对应用是完全透明的.但是,在大规模并行系统中,应用状态可能占用大量的存储空间,因此所有进程同时读写 checkpoints 的 I/O 数据量很大,从而使 I/O 成为大规模并行系统中 checkpointing 技术的性能瓶颈.而应用级 checkpointing 技术通过保存一些关键数据结构来对整个计算状态进行恢复,因此其效率要远高于系统级 checkpointing 技术.本文选取了应用级 checkpointing 技术在异构系统上进行实现,并针对影响该技术容错开销的 checkpoint 保存数据量问题进行分析和讨论.

3.2 实现及优化

应用级 checkpointing 技术的基本思想是,通过程序员手动地修改应用代码,在整个计算过程中通过保存问题的关键数据结构而非整个系统状态来进行恢复.这一机制通过减少数据保存量,有效地改善了容错的性能和存储开销.由于该技术需要用户自己指定 checkpointing 的时机并选择尽可能少的信息进行保存,因此可优化的空间很大.本文针对异构系统故障传播行为的特点,将故障影响集合的分析应用于应用级 checkpointing 检查点数据信息筛选,从而对 checkpoint 保存数据量进行优化.

图 6 给出了应用级 checkpointing 技术应用于 CUDA 下的 Swim 算法的执行流程,图中只显示了其中一个 kernel 计算段 Calc1.当程序执行时,CPU 先将数据读取并传值给 GPU,GPU 在接收数据后即开始分析并进行一次 checkpoint 保存.也就是说,在每次的 kernel 调用执行前的入口点处进行一次 checkpoint 保存;其后与原始程序执行过程相同,最后在 kernel 的出口点处将结果数据写回 CPU 端;随后开始下一 kernel 的执行,如此往复.

以上我们只阐述了基本的 checkpointing 执行流程,且只在 GPU 上针对计算段进行 checkpoint 保存.由于存储空间的分离性,在对 GPU 进行 checkpoint 数据保存时不会对 CPU 产生任何影响.如果由于故障而在程序数据流中出现错误时,就回滚到离故障点最近的一次 checkpoint 来继续计算,以实现容错.由于在多数的大规模计算过程中即使是关键数据量也极为庞大,因此必须对这些数据进行分析和筛选,减小需要保存的 checkpoint 数据量,降低容错开销.根据前文的分析可以看出,通过故障传播行为分析所得的数据流方程可以对程序中关键点的数据信息进行分析,提取出通过这一点的故障影响集合,由此达到减少 checkpoint 保存数据量的目的.

在图 6 所示的流程中,我们将 checkpoint 放置于 kernel 入口点处,则依据前文得出的 kernel 入口点故障影响集合的数据流方程对入口点处的数据信息进行分析:

$$ErrorEntry(S)=(ErrorCall(S)\cap CG\_Trans(S,X))\cup ErrorTrans(S) \tag{3.1}$$

在为 checkpoint 分析需要保存的数据信息时,根据公式(3.1),针对不同的 kernel,我们只需对从每个调用节点传递给 kernel,并且为这一 kernel 计算时所必需的变量信息进行故障影响集合分析并提取出来,而无需将算法中全部变量信息都保存进去.

$$ErrorEntry(S)=RELY\_MOD(S)\cup(ErrorExit(S)-RELY\_KILL(S)) \tag{3.2}$$

根据前文的分析,如果计算段 Calc1Kernel 中出现错误时,那么其入口传入变量 P,U,V 中也有可能存在错误.

© 中国科学院软件研究所 http://www.jos.org.cn

通过公式(3.2),可以将可能携带错误并传递的变量提取出来进行保存,去除由于注销而被消除的错误变量,从而可以进一步减小 checkpoint 数据存储的空间及时间开销.

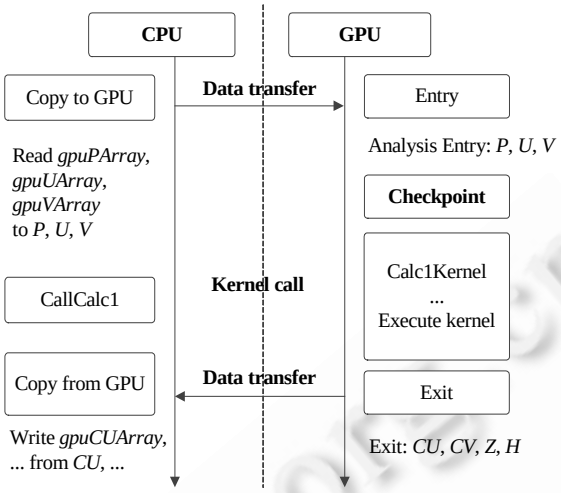

Fig.6 Checkpointing on CUDA program

图 6 CUDA 上的 checkpointing 流程

4 实 验

4.1 实验方法

本文实验主要针对由于硬件故障导致在软件中出现错误的情况来进行模拟验证的,由于异构系统绝大部分的计算都在 GPU 中完成,因此本文实现的应用级 checkpointing 技术也主要针对 GPU 进行容错.为了验证优化策略的可行性并评估优化效果,我们选取了 CUDA 平台的 Swim,Mgrid 等 6 个实例进行实验.通过代码修改,在 CUDA 上实现了异构系统的应用级 checkpointing 技术,使得算法在 GPU 上的运算过程中自主保存检查点,同时根据公式(3.1)和公式(3.2)的数据流方程对检查点保存数据量进行了优化处理.

应用级 checkpointing 技术中,程序员可以根据系统的平均故障时间间隔(mean time between failures,简称 MTBF)以及算法规模来手动调整检查点的最优保存周期.其主要需考虑各 checkpoint 处保存数据量的大小,并且要求连续 2 次 checkpointing 的时间间隔要小于系统的 MTBF.但是在本文实验中,我们主要是为了观察优化保存点存储开销所取得的性能提升,因此对于算法的检查点位置选择问题不予严格要求.由于算法规模不大,迭代次数不多,因此单次检查点的存储开销也并不明显.为了突显优化效果,我们在所有 kernel 每次迭代运算前的入口点处都进行一次 checkpointing,并使用我们提出的优化策略进行处理.

作为对比,我们将使用了优化策略与未经优化的异构系统应用级 checkpointing 技术的算法进行测试,通过 checkpoint 数据保存的时间开销以及存储空间开销的对比来对我们方法的优化效果进行评估.

实验测试的所有 6 个实例,其计算规模都是以运算时所需分配的最大显存空间可以占满全部 GPU 显存空间为基准来确定,是当前平台 GPU 执行所能负载的最大规模.需要强调的是,由于 GPU 显存空间的限制,我们无法将一次完整运算中的全部 checkpoint 数据信息完全存储在显存中,因此还需要对存储信息进行 GPU 到 CPU 的传输,但同时也增加了传输所需的时间开销.在测试中,我们将所有 checkpoint 数据信息都保存在显存空间,即无 CPU-GPU 传输状态的应用级 checkpointing 技术(CPU-GPU- communication -free checkpointing)与有 CPU-GPU 传输状态的应用级 checkpointing 技术(checkpointing with CPU-GPU- communication)也同时进行了有无优化时的对比评估.

此外,我们在实验中没有对错误注入以及 restart 机制的开销进行评估.由于恢复时的开销与 checkpoint 数据

© 中国科学院软件研究所 http://www.jos.org.cn

保存开销必成正比,其未经优化与优化后的开销对比结果与只对 checkpoint 数据保存优化前后的开销进行对比的结果是类似的.因此为了更为简单、直观地验证本文所提出的优化策略对于 checkpointing 技术的优化性能,就只对 checkpoint 数据保存开销的优化效果进行了对比和评估.

实验平台是由 i945+GTX295 搭建的 CUDA 服务器,四核 CPU 每核主频都是 2.66GHz,GPU GTX295 中有 30SM(stream multiprocessor),每个 SM 中有 8 个核,主存为 4GB,CPU 与 GPU 之间传输带宽为 3GB/s,在 Linux SUSE 11.2 下进行代码修改及测试.

4.2 实验结果

本节对 CUDA 平台的应用级 checkpointing 技术时间开销进行评估,并与使用了基于异构系统故障传播行为分析优化策略的应用级 checkpointing 技术进行了性能对比.

针对每个算法测试以下 7 组数据:

- 原始算法运行时间;
- 使用了未经优化的无 CPU-GPU 传输状态应用级 checkpointing 技术算法运行时间;
- 使用了经过优化的无 CPU-GPU 传输状态应用级 checkpointing 技术算法运行时间;
- 使用了未经优化的有 CPU-GPU 传输状态应用级 checkpointing 技术算法运行时间;
- 使用了经过优化的有 CPU-GPU 传输状态应用级 checkpointing 技术算法运行时间;
- 使用了未经优化的应用级 checkpointing 技术算法的状态点保存容量;
- 使用了经过优化的应用级 checkpointing 技术算法的状态点保存容量.

表 1 给出了 6 个科学计算程序的 checkpointing 时间开销,分别列出了算法原始执行时间(clean);无 CPU-GPU 传输时未优化的 checkpointing 算法执行时间(check_all)和优化过的 checkpointing 算法执行时间(check_opt);有 CPU-GPU 传输的未优化的 checkpointing 数据传输时间(all_trans)和优化过的 checkpointing 数据传输时间(opt_trans).

Table 1 Time overhead of checkpointing

表 1 检查点时间开销

	Clean (ms)	Check_All (ms)	Check_Opt (ms)	All_Trans (ms)	Opt_Trans (ms)
Mgrid	1.060	1.633	1.173	9 805.1	2 293.2
Swim	1 131.714	2 509.280	2 046.685	22 002.2	14 668.1
backprop	62.238	67.994	63.490	74.1	49.6
bfs	23.396	45.493	35.873	304.0	164.0
FFT2D	8.880	12.237	11.251	67.1	44.7
gaussan	394.109	425.059	414.588	4.2	2.8

图 7 给出了在无 GPU-CPU 传输状态下,对 checkpoint 的保存未经优化与优化后的算法分别相对于原始算法执行时间开销增加比值的对比.可以看出,在使用未经优化的 checkpointing 技术时,除了 backprop 和 gaussan 两个算法外,其余算法运行的时间开销都超出了原程序执行时间的至少 30%,甚至更高.其中,Swim 算法在不计原始算法执行时间、单独进行 checkpointing 时所用的时间还要略高于原始算法执行时间,这主要是由于我们选取的算法规模较小所造成的.实际上,大规模异构系统集群在运算大型计算程序时,其进行 checkpointing 的时间开销相对原始程序的执行时间开销比值会很小.通过对比可以看出,所有算法优化后的 checkpointing 时间开销相对原始算法执行时间开销的增加比都要明显低于未经优化的 checkpointing 时间开销增加比.

图 8 给出了无 CPU-GPU 传输状态下,优化后相对未经优化的 checkpointing 的存储时间开销减少百分比.可以看出,所有算法在经过优化后的 checkpointing 时间开销都比未经优化时减小了很多,其优化收益最少也提升了 29.4%(FFT2D),最大达到 80.4%(Mgrid).

© 中国科学院软件研究所 http://www.jos.org.cn

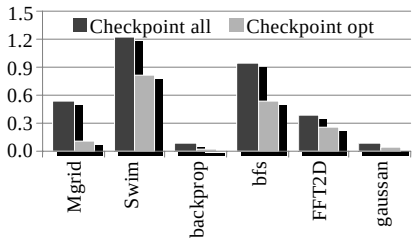

Fig.7 Increased percentage of execution time with CPU-GPU- communication -free checkpointing

图 7 无 CPU-GPU 传输状态检查点的执行时间增加比

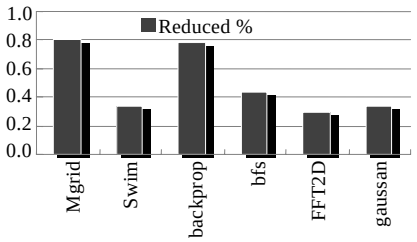

Fig.8 Reduced percentage of time overhead

图 8 时间开销减少百分比

表 2 给出了 checkpoint 数据信息的存储开销,列出了每个计算段单次迭代时未经优化的数据信息存储量和优化后的数据信息存储量以及迭代次数.图 9 显示了在有 CPU-GPU 传输情况下,优化后相对未经优化的应用级 checkpointing 技术在保存 checkpoint 数据时的时间和存储空间开销的优化收益情况.其中,在进行 GPU 到 CPU 的数据传输时,数据带宽为 3GB/s.可以看出,时间与存储空间开销这两者完全成正比,且开销减小的百分比都很接近.其中,优化收益最小的 Swim 和 FFT2D 也达到了 33.3%,而 Mgrid 优化收益达到了 76.6%.可见,其优化效果是十分显著的.

Table 2 Storage overhead of checkpointing

表 2 检查点存储开销

	CheckP/iter (Bytes)	CheckALL/iter (Bytes)	Iterations
Mgrid	687969536	2941543072	10
Swim	440043240	660064860	100
backprop	148898276	222298936	1
bfs	40999880	75999760	12
FFT2D	134217728	201326592	1
gaussan	8388608	12582912	1

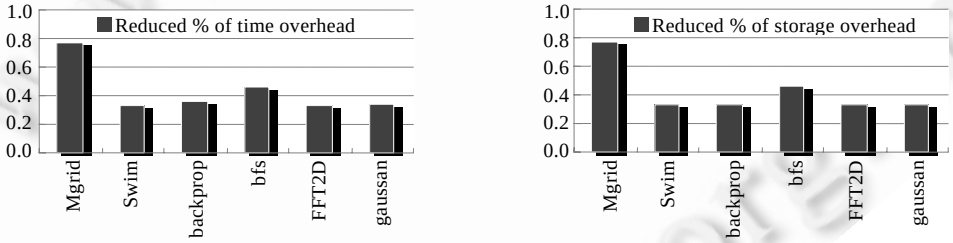

Fig.9 Optimization of checkpointing with CPU-GPU- communication

图 9 有 CPU-GPU 传输状态检查点的优化收益

由于一般的大型科学计算程序所需要保存的 checkpoint 数据信息是极为巨大的,因此显存空间肯定无法全部存储,必须将其传输到 CPU 端进行储存,这就需要额外的时间开销来进行数据传输.这也是异构系统与同构系统 checkpointing 技术的不同之处,也正因如此,其优化空间也是极为巨大的.通过实验结果可以看出,我们针对 CUDA 平台上的应用级 checkpointing 技术的 checkpoint 数据保存的优化策略效果明显,有效地减少了时间开销和存储空间开销,相对未经优化的 checkpointing 技术有很大的性能优势.

5 相关工作

当前,针对异构系统的容错问题所采用的大都是硬件冗余和软件冗余技术.Sheaffer 和 NVIDIA 研究小组的 Luebke 等人在 2007 年设计了一种面向超级计算领域的 GPGPU 可靠性的硬件冗余和恢复机制[19],George 等人也在 2008 年 DSN 会议上提出了基于硬件双模冗余和同步数据流技术的异构系统的容错方案[20],但是这些通过

© 中国科学院软件研究所 http://www.jos.org.cn

增加冗余硬件的容错方法并不适用于现在被广泛使用的普通 GPU.

在硬件冗余技术提出的同时,基于软件容错的机制也被越来越多的研究者所关注.Dimitrov 等人提出了一种冗余执行 kernel 的方式对 GPU 上发生的瞬时故障进行故障检测[21],Gregerson 等人分析了时间冗余、空间冗余、数据多样性冗余和多种传统软件容错方法在 GPU 上运用时的性能情况[22].实验结果表明,由于 GPU 计算的高效性,使用时间冗余的容错方法更加适用于 GPGPU 程序.

现有的异构系统容错技术中的冗余机制主要都只是用于故障检测,而容错的故障恢复机制的研究还并不深入,并未出现一套更为有效的容错技术.本文针对 CUDA 平台的编程模型,将应用级 checkpointing 技术应用于异构运算,验证了异构系统的应用级 checkpointing 技术的可行性.并以异构系统的故障传播行为分析为指导,提出了一套应用于异构系统的应用级 checkpointing 技术的检查点保存的优化策略,减小了检查点数据保存量,提高了容错性能.

6 结 论

随着 GPU 等加速器逐渐进入高性能计算领域,面向异构系统的容错研究必将得到更多的关注.本文以 CPU-GPU 异构系统为平台,研究了面向异构系统的故障传播行为方式.并以此为基础,针对异构系统的容错技术提供了更多的优化策略和一些全新的思路.

通过分析异构系统故障传播行为方式,提出一套基于异构系统的故障传播数据流方程,并初步建立了异构系统的故障传播模型.同时,对几个典型的科学计算应用进行了测试和分析.实验结果表明,本文提出的方法可以有效减少异构系统下应用级 checkpointing 技术中检查点的数据保存量,实现更高效的检查点存储.在未来的工作中,我们将进一步研究异构系统的故障传播模型以及面向异构系统的错误隔离技术及容错优化方法.

References:

[1] Luebke D, Harris M, Krüger J, Purcell T, Govindaraju N, Buck I, Woolley C, Lefohn A. GPGPU: General purpose computation on graphics hardware. In: Proc. of the ACM SIGGRAPH 2004 Course Notes. Los Angeles: ACM Press, 2004. 33. [doi: 10.1145/1103900.1103933]

[2] Fan Z, Qiu F, Kaufman A, Yoakum-Stover S. GPU cluster for high performance computing. In: Proc. of the 2004 ACM/IEEE Conf. on Supercomputing (SC 2004). Washington: IEEE Computer Society, 2004. 47. [doi: 10.1109/SC.2004.26]

[3] Dally WJ, Hanrahan P, Erez M, Knight TJ. Merrimac: Supercomputing with streams. In: Proc. of the Supercomputing Conf. (SC 2003). 2003. 35−42. [doi: 10.1109/SC.2003.10043]

[4] TOP500 supercomputing site. http://www.top500.org

[5] Burke MG, Cytron RK. Interprocedural dependence analysis and parallelization. In: Proc. of the 20 Years of the ACM/SIGPLAN Conf. on Programming Language Design and Implementation (1979~1999): A Selection. IBM Research Report RC11794. 2003. [doi: 10.1145/13310.13328]

[6] Ramkumar B, Strumpen V. Portable checkpointing for heterogeneous architectures. In: Proc. of the 27th Int'l Symp. on Fault-Tolerant Computing (FTCS'97). Washington: IEEE Computer Society, 1997. 58−67. [doi: 10.1109/FTCS.1997.614078]

[7] Beguelin A, Seligman E, Stephan P. Application level fault tolerance in heterogeneous networks of workstations. Journal of Parallel and Distributed Computing, 1997,43(2):147−155. [doi: 10.1006/jpdc.1997.1338]

[8] Kirk D. NVIDIA CUDA Software and GPU Parallel Computing Architecture. New York: ACM Press, 2007. 103−104. [doi: 10.1145/1296907.1296909]

[9] Kapasi UJ, Rixner S, Dally WJ, Khailany B, Ahn JH, Mattson P, Owens JD. Programmable stream processors. IEEE Computer, 2003,36(8):54−62. [doi: 10.1109/MC.2003.1220582]

[10] Advanced Micro Devices, Inc. AMD brook+. http://ati.amd.com/technology/streamcomputing/AMDBrookplus.pdf

[11] Open computing language. http://www.khronos.org/

[12] CUDA technical training volume I/II. Prepared and Provided by NVIDIA, 2008.

[13] NVIDIA CUDA computer unified device architecture programming guide. Version 2.1. Beta, 2008.

© 中国科学院软件研究所 http://www.jos.org.cn

[14] Halfhill TR. Parallel processing with CUDA. Nvidia's High-Performance Computing Platform Uses Massive Multithreading.

[15] Chen ZZ, Fagg GE, Gabriel E, Langou J, Angskun T, Bosilca G, Dongarra J. Fault tolerant high performance computing by a coding approach. In: Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP 2005). 2005. 213−223. [doi: 10.1145/1065944.1065973]

[16] Houghm PD, Glodsby ME, Walsh EJ. Algorithm dependent fault tolerance for distributed computing. Technical Report, SAND2000-8219, Sandia, 2000.

[17] Elnozahy EN, Alvisi L, Wang YM, Johnson DB. A survey of rollback-recovery protocols in message-passing systems. ACM Computing Surveys, 2002,34(3):375−408. [doi: 10.1145/568522.568525]

[18] Plank JS, Li K, Puening MA. Diskless checkpointing. IEEE Trans. on Parallel Distributed Systems, 1998,9(10):972–986. [doi: 10.1109/71.730527]

[19] Sheaffer JW, Luebke DP, Skadron K. A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors. In: Proc. of the Graphics Hardware. 2007.

[20] George N, Lach J, Gurumurthi S. Towards transient fault tolerance for heterogeneous computing platforms. In: Proc. of the 38th Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN 2008). 2008.

[21] Dimitrov M, Mantor M, Zhou HY. Understanding software approaches for GPGPU reliability. In: Proc. of the 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2), Vol. 383. Washington: ACM Press, 2009. 94–104. [doi: 10.1145/1513895.1513907]

[22] Gregerson AE, Abhyankar AV. Performance cost analysis of software-implemented hardware fault tolerance methods in general-purpose GPU computing. http://homepages.cae.wisc.edu/~ece753/papers/Paper_4.pdf

贾佳(1981－),男,北京人,博士生,主要研究领域为计算机体系结构,容错技术.

杨学军(1963－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为并行计算机体系结构与编译,容错并行算法,流处理器体系结构与编译.

© 中国科学院软件研究所 http://www.jos.org.cn