软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2011,22(10):2538−2552 [doi: 10.3724/SP.J.1001.2011.04058]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

一种面向 CPU-GPU 异构系统的容错方法*

徐新海+, 杨学军, 林宇斐, 林一松, 唐 滔

(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)

Fault-Torlerance Method for CPU-GPU Heterogeneous System

XU Xin-Hai+, YANG Xue-Jun, LIN Yu-Fei, LIN Yi-Song, TANG Tao

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

+ Corresponding author: E-mail: xuxinhai@nudt.edu.cn

Xu XH, Yang XJ, Lin YF, Lin YS, Tang T.Fault-Torlerance method for CPU-GPU heterogeneous system. *Journal of Software*, 2011,22(10):2538−2552. http://www.jos.org.cn/1000-9825/4058.htm

Abstract: In recent years, heterogeneous parallel architecture has become an important development trend of supercomputer because it mitigates the problem of increasingly high power consumption. As a high performance and power efficiency accelerator, GPU (graphics processing unit) has been extensively used in HPC (high performance computing) area. However, the inherent unreliability of the GPU hardware deteriorates the reliability of supercomputer. Presently, most research of FT (fault-tolerance) techniques for CPU-GPU heterogeneous system isolates the GPU from the system, and does FT work for it at the granularity of a single GPU invocation. This paper proposes a new Lazy FT method for CPU-GPU heterogeneous system, introduces a FT framework and its constraints based on directives, and demonstrates the validity of the Lazy FT method. The experimental results show that, compared with existing FT methods, the cost of LazyFT is very cheap.

Key words: GPGPU; heterogeneous system; fault-tolerance; Lazy strategy; checkpointing

摘 要: 近年来,为了缓解日益严重的功耗问题,异构并行体系结构已成为超级计算机发展的一个重要趋势.图形处理器(graphics processing unit,简称 GPU)凭借其超高的计算性能和性能功耗比,作为一种高效的加速部件已被广泛应用于高性能计算领域.但是,GPU 先天的可靠性缺陷势必加剧超级计算机的可靠性问题.目前,国际上关于 CPU-GPU 异构系统容错技术的研究工作主要将 GPU 从异构系统中独立出来,以每次调用为粒度对其进行容错处理.设计了一种面向 CPU-GPU 异构系统的 Lazy 容错方法,给出了基于编译指导命令的容错框架及其约束,并讨论了相关的编译实现和优化方法,最后通过实验验证了该方法的正确性.实验结果表明,与现有的容错方法相比,利用所设计的 LazyFT 容错方法对 GPGPU(general purpose computation on graphics hardware)程序进行容错处理,可以明显降低容错代价.

关键词: GPGPU;异构系统;容错;Lazy 策略;检查点

中图法分类号: TP316 **文献标识码**: A

* 基金项目: 国家自然科学基金(60921062, 60873016)
收稿时间: 2010-04-28; 定稿时间: 2011-05-18

随着工艺的进步,单个芯片的计算性能不断提高.但是由于同构多核处理器存在严重的“存储墙”问题,且功耗问题也日益严峻[1],于是,HPC(high performance computing)领域的研究者们逐渐尝试利用专用处理器作为通用处理器的协处理器,组成异构系统以加速应用程序的执行[2,3].GPU(graphics processing unit)凭借其超高的计算性能和性能功耗比,如单个 AMD5870 显卡的单精度峰值计算性能已达到 2Tflops,受到了 HPC 研究者的广泛关注.近年来,随着 GPGPU(general purpose computation on graphics hardware)的流行[4,5],利用 CPU 和 GPU 构建异构系统已成为超级计算机发展的新趋势之一[6].在 2010 年 11 月发布的 top500 榜单中,排名第 1 的“TianHe-1A”超级计算机采用的就是 CPU-GPU 异构体系结构.

在提供强大的计算性能的同时,GPU 集成了大量的功能部件且运行时温度较高,容易出现瞬时故障[7],可靠性大大低于传统 CPU.由于 GPU 最初应用于图形处理领域,它的输出往往对应于图像中的像素,而少数像素点的错误并不会影响图像的整体效果,因此到目前为止,主流的 GPU 体系结构以及编程模型所提供的容错管理机制较弱[8].但是,通用计算领域尤其是科学计算程序对计算结果的正确性要求极其严格,这就需要对 GPU 上发生的瞬时故障进行容错处理.业界也开始注意到这一问题,比如 NVIDIA 公司就在 Femi 架构中为 GPU 的存储系统提供 ECC 校验,这虽然可以解决 GPU 存储部件的可靠性问题,但对于 GPU 上计算部件的瞬时故障仍缺乏保护,所以,针对 GPGPU 瞬时故障容错的研究仍具有重大意义.

由于目前广泛使用的 GPU 中缺乏容错硬件的支持,因此现有的针对 GPU 瞬时故障的容错方法主要采用纯软件的方法实现.然而现有的软件容错方法仍处于起步阶段,往往将 GPU 从 CPU-GPU 异构平台中孤立出来,以 GPU 的一次 Kernel 调用为粒度进行容错处理[8],这大大增加了 GPGPU 程序的容错代价.因此,我们结合 GPGPU 错误传播特点,提出了一种面向 CPU-GPU 异构平台的 Lazy 容错方法,本文的主要创新点是:

- 分析了 GPU 瞬时故障所产生的错误在 CPU-GPU 异构平台上的传播规律,并基于该传播规律提出了 Lazy 的容错思想;
- 设计了一种面向 CPU-GPU 异构系统的 Lazy 容错框架,并基于该框架实现了一种 LazyFT 容错方法;
- 建立了容错 GPGPU 程序的执行时间模型,并基于该时间模型给出了科学计算程序中两类典型程序段在使用 LazyFT 容错方法时的最优容错粒度选择方法;
- 通过实验验证了 LazyFT 容错方法的有效性.与现有的容错方法相比,使用 LazyFT 容错方法并结合相应的容错优化对 spec2000 中的 Swim 程序进行容错处理,可以在有故障发生的情况下将程序的总执行时间缩短 60%.

本文第 1 节介绍 GPGPU 程序的执行模式及其故障模型.第 2 节提出面向 CPU-GPU 异构系统的 Lazy 容错思想,并设计实现 LazyFT 容错方法.第 3 节建立容错 GPGPU 程序的执行时间模型,并在该模型的指导下给出两类典型科学计算程序在使用 LazyFT 容错方法时的最优容错粒度选择方法.第 4 节通过案例分析证实面向 CPU-GPU 异构系统的 Lazy 容错方法的正确性和性能优势.第 5 节介绍相关工作.第 6 节对本文工作进行总结.

1 系统结构及故障模型

为了更好地对面向 CPU-GPU 异构系统的容错方法进行研究,我们首先对 CPU-GPU 异构系统的结构和故障模型加以介绍.

1.1 系统结构和执行模式

一个典型的包含 GPU 的异构系统的体系结构如图 1 所示,它由一个 CPU 和一个 GPU 组成,CPU 和 GPU 之间通过 PCI-E 总线相连,各自的存储系统可以通过 DMA 操作进行大块的数据传送.如图 2 所示,当一个通用程序在 CPU-GPU 异构系统上运行时,CPU 和 GPU 之间是一种主从关系,CPU 负责主控程序的执行,以 Kernel 调用的方式启动 GPU 进行异步运算,并通过 Read 和 Write 操作与 GPU 进行数据传输;GPU 响应 CPU 发起的 Kernel 调用,并通过 SIMT 的方式启动大量的轻量级线程完成 Kernel 计算.

根据执行部件的不同,可以将 GPGPU 程序中的指令分为以下 3 类:

- GPU 指令:该类指令在 GPU 上完成操作,所涉及的数据都位于 GPU 端,这类指令仅包含 Kernel 调用;

- 通信指令:该类指令负责 CPU 和 GPU 之间的通信,所涉及的数据位于 CPU 和 GPU 两端,这类指令包括 Write 指令和 Read 指令;
- CPU 指令:该类指令在 CPU 上完成操作,所涉及的数据也都位于 CPU 端,除 Kernel,Write 和 Read 之外的所有指令都属于这类指令.

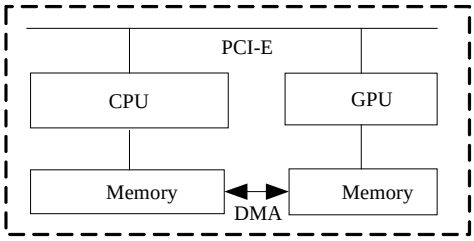


Fig.1 Architecture of CPU-GPU

图 1 CPU-GPU 体系结构

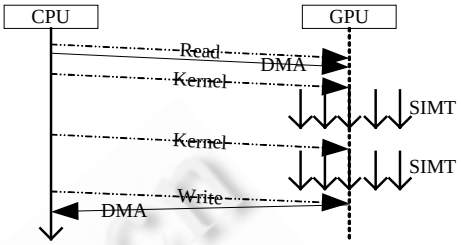


Fig.2 Executing model of GPGPU

图 2 GPGPU 执行模式

1.2 故障模型

在针对 GPGPU 异构系统进行容错方法研究之前,必须确定对应的故障模型,而一般故障模型中主要涉及故障、错误以及失效这 3 个概念[9,10].如图 3 所示,在本文所建立的故障模型中,主要研究 GPU 上计算部件的瞬时故障,假设 CPU 和 GPU 的存储部件都是可靠的.错误则特指因 GPU 计算部件在运算过程中发生瞬时故障所引起的数据错误,即 SDC(silent data corruption)错误[11].由于 SDC 错误并不会引起 GPU 乃至系统的死机,所以失效是指程序最终运行结果的不正确,同时本文认为,只要 CPU 指令使用了含有错误的数据,就会引起程序最终运行结果的不正确.故本文故障模型中的失效特指这类 CPU 指令使用了错误数据的失效,简称 CPU 指令失效.

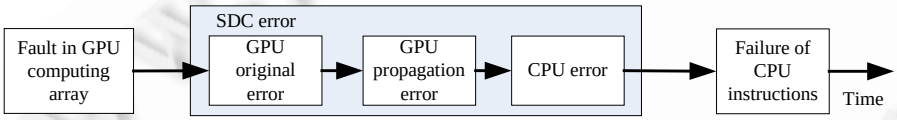


Fig.3 Fault-Tolerance model of GPGPU

图 3 GPGPU 故障模型

为了便于对 CPU-GPU 异构系统容错方法展开进一步研究,我们将 SDC 错误传播细分为以下 3 个阶段:首先,GPU 计算部件在 GPU 指令的计算过程中发生瞬时故障,可能导致其直接计算结果发生错误,我们称这类错误为 GPU 原生错误;然后,新的 GPU 指令可能因访问含有错误的 GPU 数据,从而产生了新的错误,即 GPU 传播错误;最后,Write 指令可能因访问了含有错误的 GPU 数据,从而将错误传播至 CPU 端,我们称其为 CPU 错误.如果在程序的后续运算过程中存在 CPU 指令访问了含有错误的 CPU 数据,则会引起 CPU 指令失效.

2 Lazy 容错方法

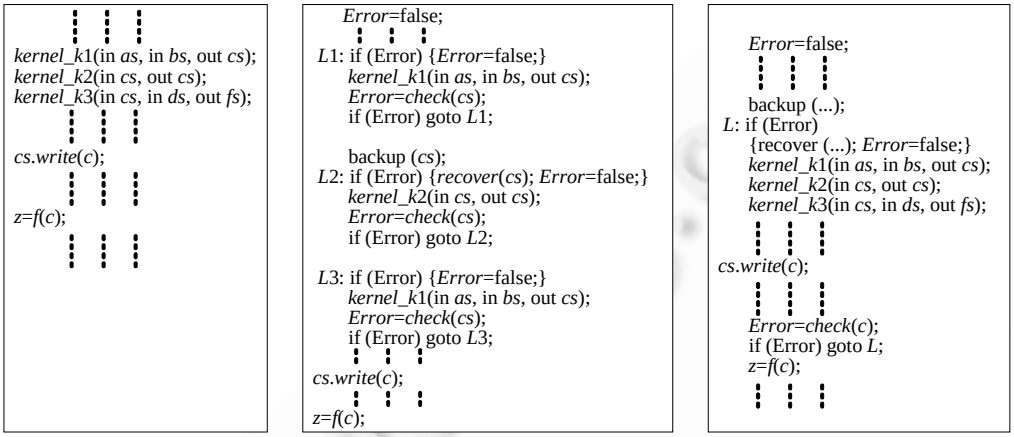
由于GPU故障所引发的错误是SDC错误,这类错误在程序执行过程中不易被程序员所发现,所以需要设计相应的故障检测机制以检测故障,本节就从故障检测的时机出发,对 CPU-GPU 异构系统的容错工作展开研究.

2.1 Lazy容错思想的提出

本文假设故障只发生在GPU指令的计算过程中,现有面向GPGPU的软件容错方法主要采用Eager的容错方法[8].如图 4(b)所示,Eager 容错方法在故障发生后的第一时间对 GPU 原生错误进行错误检测,以防止 GPU 传播错误以及 CPU 错误的产生,从而杜绝 CPU 指令失效.该方法以 GPU 指令为粒度,针对每个 Kernel 进行容错处理:首先,在 Kernel 执行前备份程序现场,即备份那些同时作为 Kernel 输出和输入变量的数据;其次,为 Kernel 调用准备故障恢复的入口,并在该入口进行与现场备份相对应的现场恢复工作;最后,在 Kernel 调用后立刻对其输

出结果进行错误检测,并根据错误检测的结果决定是否需要回滚程序至故障恢复入口.

虽然 Eager 容错方法实现简单且保证了程序运行结果的正确性,但其需要对每个 Kernel 调用进行容错处理,容错粒度较小,容错工作频繁,容错代价较大.通过对 GPGPU 故障模型的分析我们发现,只有 CPU 错误才可能直接引起 CPU 指令失效,从而影响程序的最终执行结果.因此,本文提出了一种 Lazy 容错思想.如图 4(c)所示,理想的 Lazy 容错方法允许包括 CPU 错误在内的所有 SDC 错误的存在,将错误检测的时机推迟到可能含有 CPU 错误的数据真正被 CPU 指令使用之前.



(a) Original program (a) 原始程序 (b) Eager FT (b) Eager容错 (c) Ideal Lazy FT (c) 理想Lazy容错

Fig.4 EagerFT and ideal LazyFT

图 4 EagerFT 和理想 LazyFT

显然,理想的 Lazy 容错方法只需对那些从 GPU 端写回 CPU 并被 CPU 指令所访问的数据进行错误检测,这可以大大降低错误检测的数据量.但是,这种将故障检测时机推迟至最后的理想 Lazy 容错方法也存在以下缺陷:一方面,如果使用最适合 GPGPU 容错的时间双模冗余方法[12]进行故障检测,为了对 CPU 数据进行比较,理想的 Lazy 容错方法除需要冗余 GPU 数据外,还需要冗余可能出现 CPU 错误的 CPU 数据,而且该方法除需进行冗余 Kernel 计算外,还需要进行冗余 Write 操作,这会同时增加存储空间和计算时间的容错额外开销;另一方面,由于理想的 Lazy 容错方法在每次错误检测时只检测那些即将被使用且可能含有 CPU 错误的数据,不能保证整个程序执行现场的正确性,如图 4(c)中对数据 *c* 进行错误检测后并不能保证 GPU 数据 *fs* 的正确性,因此在理想的 Lazy 容错方法中,单次错误检测不能保证程序运行现场的正确性,不利于形成稳定而可靠的现场备份点,从而导致现场备份点间隔增大,回滚代价增加.

2.2 LazyFT

针对理想的 Lazy 容错方法所存在的缺陷,我们在容错思想上对其进行如下改进:首先,为了避免引入冗余 Write 操作和冗余 CPU 数据空间,我们将错误检测的时机提前至 CPU 错误产生之前,即在 Write 操作之前进行错误检测,以确保所有写回 CPU 的数据都是正确的;其次,在每次错误检测时扩大检测对象,即检测所有可能存在错误的 GPU 数据,从而在错误检测后形成可靠的程序运行现场,以便进行现场备份.

2.2.1 容错框架及其约束

基于上述改进的基本思想,本文提出了如图 5 所示的面向 GPGPU 异构系统的改进 Lazy 容错方法,简称 LazyFT.在这种容错方法中,程序员通过在 GPGPU 程序中成对地插入“#Checkpoint”和“#ErrorDetect”编译指导命令以指定现场备份工作和错误检测工作的具体位置,从而进行容错处理.我们称那些位于编译指导命令对之间的程序段为容错目标程序段,以表示其是需要进行容错处理的;而称那些位于容错目标程序段之间的程序段

为非容错目标程序段,以表示在本文故障模型及 LazyFT 容错框架下,该段代码在运行过程中不涉及错误的产生与传播,无需容错处理.

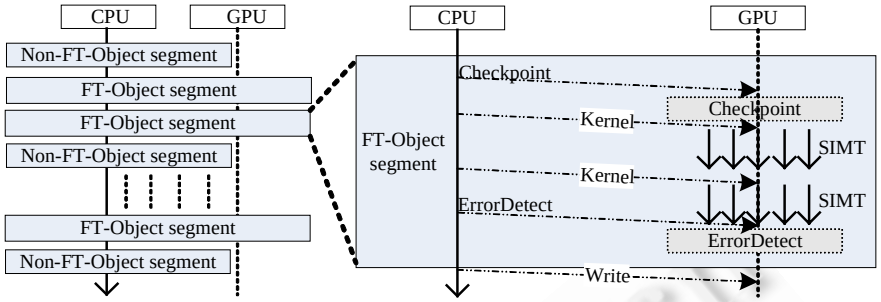


Fig.5 Fault-Tolerance framework of LazyFT

图 5 LazyFT 的容错框架

为了满足 LazyFT 在错误检测时机的要求,程序员在插入编译指导命令时必须满足以下约束:

约束 1. 所有的 Kernel 调用都应被封装于对应的容错目标程序段之中;

约束 2. 任何容错目标程序段之中都不允许出现 Write 操作.

约束 1 确保了所有故障都只发生在容错目标程序段之中;而约束 2 又保证了所有错误数据在被写回 CPU 端之前都进行了错误检测,杜绝 CPU 错误的产生,避免 CPU 指令失效的出现,从而保证了程序最终运行结果的正确性.

2.2.2 容错目标程序段的选择

在满足 LazyFT 的容错框架及其约束的前提下,考虑到在确保无 CPU 错误的情况下 CPU 指令的执行是不影响针对 GPU 数据的错误检测结果的,我们本着尽可能减少故障恢复代价的目的,原则上将现场备份点和错误检测点分别置于 Kernel 调用的前后,以避免容错目标程序段的头尾出现不影响故障检测结果的 CPU 代码.为此,本文通过如图 6 所示步骤对原程序进行静态分析,以帮助用户确定满足要求的现场备份点和错误检测点位置.

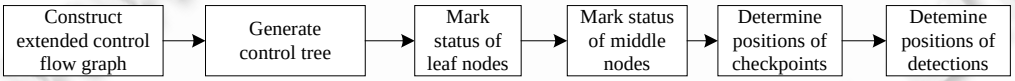
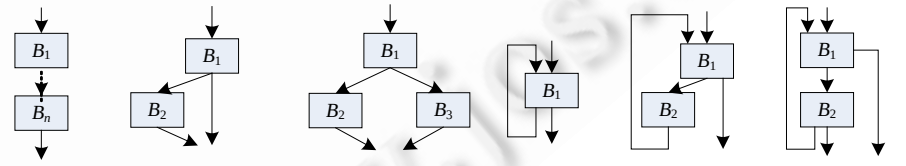


Fig.6 Flow of FT object segment selection

图 6 容错目标程序段选择流程

步骤 1.构造整个程序的扩展控制流图.与一般控制流图不同,该扩展控制流图中将所有的 Write 操作和 Kernel 调用都作为一个基本块来处理.我们假设程序中只存在如图 7 所示的 6 种控制结构.



(a) Block (b) If-Then (c) If-Then-Else (d) Self-Loop (e) While-Loop (f) Natural-Loop

Fig.7 Control structure

图 7 控制结构

步骤 2.采用结构分析方法[13]对控制流图进行结构分析,将控制流图中每种控制结构都标识成一个区域,并将其替换成一个抽象节点.如此递归地进行下去,直到最终将整个流图规约成一个单节点,同时得到结构分析过程对应的控制树.控制树的根节点代表整个程序,叶节点代表基本块,其他中间节点则代表各种对应的控制结构.需要特别指出的是,本文所形成的控制树中不存在嵌套 block 结构,所有的嵌套 block 结构都可以展开为一层

block 结构.

步骤 3.根据叶节点所对应基本块的类型,初始化控制树中所有叶节点的状态:如果基本块为 Kernel 调用,则该叶节点的状态为 *K*,以表示在该节点之前需要插入现场备份点;如果基本块为 Write 操作,则该叶节点的状态为 *W*,以表示该节点之前需要插入故障检测点;否则,该叶节点状态为 *N*,以表示该节点为一般节点,可以与其他类型的节点合并状态.

步骤 4.采用后序遍历方法遍历整个控制树,根据一定规则设置除叶节点外其他抽象节点的状态,并对相应节点进行待现场备份或待错误检测标记.抽象节点的状态除 *K*,*W* 和 *N* 外,还加入状态 *F* 状态,以表示该抽象节点所对应的控制树分支已实现容错功能.具体规则如下:

规则 1.1. 若该抽象节点的所有子节点状态都相同,则该节点状态与其子节点保持相同.

规则 1.2. 若该抽象节点的所有子节点除状态 *N* 外只含有一种状态,如 *K*(/*W*/*F*),则该节点的状态与该种状态相同,即为 *K*(/*W*/*F*).

规则 1.3. 若该抽象节点的所有子节点除状态 *N* 外至少还存在两种其他节点,则根据抽象节点控制类型不同,按规则 1.3.1 和规则 1.3.2 进行处理,并将该抽象节点的状态设为 *F* 状态.

规则 1.3.1. 若该节点控制类型为 block,假设其共有 *n* 个子节点,其中含有 *m* 个 *F* 状态节点(*m*>=0),则首先将 *n* 个子节点按 *m* 个 *F* 状态节点为间隔划分为 *m*+1 段,每段并不包括 *F* 状态节点;其次,针对每个段,顺序扫描其包含的子节点,将段中的第 1 个 *K* 节点和每个 *W* 节点后的第 1 个 *K* 节点标记为待现场备份;同时,将段中最后一个 *K* 节点和每个 *W* 节点前的最后一个 *K* 节点标记为待错误检测.需要注意的是,一个 *K* 节点可能同时被标记为待现场备份和待错误检测.

规则 1.3.2. 若该节点控制类型为 if-then-else,由于 Write 操作和 Kernel 调用所对应的基本块出度不大于 1,所以其不会出现在 B_1 子节点中,即 B_1 状态为 *N*.根据规则 3 的基本假设,B_2 和 B_3 两个子节点的状态为 *K*,*W*,*F* 中不同的两种,若其存在状态为 *K* 的子节点,我们将该 *K* 子节点同时标记为待错误检测和待现场备份.

对于 if-then 以及 3 种 loop 控制结构,基于与规则 1.3.2 中类似的原因,其 B_1 子节点的状态都为 *N*,不可能满足规则 1.3 关于子节点中除状态 *N* 外至少还存在两种其他节点的基本要求.

步骤 5.采用先序遍历方法遍历整棵控制树,对其中被标记为待现场备份的节点根据如下规则进行处理,以完成确定现场备份点工作:

规则 2.1. 节点为叶节点,则在该节点执行之前进行现场备份.

规则 2.2. 节点为抽象节点,且其控制类型为 block,则将其子节点中第 1 个类型为 *K* 的节点标记为待现场备份,以便后续处理.

规则 2.3. 节点为抽象节点,且其控制类型为 if-then 或 if-then-else,如果该抽象节点未被同时标记为待错误检测,则在 B_1 子节点之前进行现场备份;否则,删除该抽象节点的待错误检测属性,并将其所有类型为 *K* 的子节点同时标记为待错误检测和待现场备份.

规则 2.4. 若该节点为抽象节点,且其控制类型为 while-loop 或 natural-loop,在 B_1 子节点之前进行现场备份.

步骤 6.采用先序遍历方法遍历整棵控制树,对其中被标记为待错误检测的节点根据如下规则进行处理,以完成确定错误检测点工作:

规则 3.1. 节点为叶节点,则在该节点执行之后进行错误检测.

规则 3.2. 节点为抽象节点,且其控制类型为 block,则将其子节点中最后一个类型为 *K* 的节点标记为待错误检测,以便后续处理.

规则 3.3. 节点为抽象节点,且其控制类型为 if-then,if-then-else,hile-loop 或 natural-loop,在该抽象节点执行后进行错误检测.

通过上述步骤,编译器可以静态分析得到若干现场备份点和错误检测点,我们称这些点分别为强制现场备份点和强制错误检测点.

2.2.3 LazyFT 的有效性

本节对由第 2.2.2 节得到的强制现场备份点和强制错误检测点是否满足 LazyFT 容错框架及其约束进行证明.

(1) 使用第2.2.2节所设计的容错目标程序段的选择方法所得到的强制现场备份点和强制错误检测点是成对出现的.

在第 2.2.2 节所设计的容错目标程序段的选择方法中,涉及到给控制树中节点标记待错误检测和待现场备份属性的规则只有规则 1.3.1、规则 1.3.2、规则 2.2、规则 2.3 和规则 3.2.其中:规则 1.3.1 和规则 1.3.2 都是在控制树中成对地插入待错误检测和待现场备份属性;规则 2.2 和规则 3.2 只是分别将已有的待错误检测属性和待现场备份属性细化至其具体子节点;而规则 2.3 中的第 2 种情况则在去掉该抽象节点的错误检测和待现场备份属性对的同时,为其所有类型为 K 的子节点同时标记为待错误检测和待现场备份.因此,该方法保证了 GPGPU 程序中的强制现场备份点和强制错误检测点是成对出现的,这也就满足了 LazyFT 容错框架的基本要求,即在原程序中成对地插入编译指导命令,将原程序划分为容错目标程序段和非容错目标程序段.

(2) 所有 Kernel 调用都位于成对的强制现场备份点和强制错误检测点之间.

在第 2.2.2 节所设计的容错目标程序段的选择方法中,步骤 3 将所有 Kernel 调用基本块所对应的叶节点标记为 K 状态,步骤 4 确保了所有 K 节点的父节点或仍为 K 状态或为 F 状态.若存在某个 K 状态叶节点的所有父节点都是 K 状态,即在步骤 4 中只使用了规则 1.1 和规则 1.2,则说明整个控制树中不存在 W 状态节点,即该程序中不存在 Write 操作.显然,这种程序是没有意义的.因此,任意 K 状态叶节点都拥有 F 状态父节点.

根据步骤 4 的规则 1.3 我们不难发现,在后续遍历控制树时,任意 K 状态叶节点的第 1 个 F 父节点都只能是通过规则 1.3.1 或规则 1.3.2 所得到,而这两个规则都将其所有 K 状态子节点封装于被标记了待错误检测和待现场备份属性的 K 状态子节点对(可以为同一节点)之间.如果分别在这对 K 状态子节点的前后插入强制故障检测点和强制错误检测点,则所有 K 状态叶节点都被封装在强制故障检测点和强制错误检测点之间.

而在步骤 5 和步骤 6 中,规则 2.1、规则 3.1、规则 3.3 以及规则 2.3 的第 1 种情况都在步骤 4 分析的基础上不作任何改变地插入强制故障检测点或强制错误检测点;而规则 2.2、规则 3.2 以及规则 2.3 的第 2 种情况则是将不涉及容错处理的 N 状态节点排除在强制故障检测和强制错误检测点对之外,也不影响 K 状态叶节点与强制故障检测点和强制错误检测点对之间的关系.综上所述,通过第 2.2.2 节所设计的容错目标程序段的选择方法,所有 K 状态叶节点,即所有 Kernel 调用,都被封装在强制故障检测点和强制错误检测点之间.

(3) 所有 Write 操作都位于成对的强制现场备份点和强制错误检测点之外.

由于与证明(2)的情况类似的原因,任意 W 状态叶节点都拥有 F 状态父节点,且其第 1 个 F 父节点也只能是通过规则 1.3.1 或规则 1.3.2 所得到,而这两个规则仅将其所有 K 状态子节点封装于被标记了待错误检测和待现场备份属性的 K 状态子节点对之间.如果分别在这对 K 状态子节点的前后插入强制故障检测点和强制错误检测点,则所有 W 状态叶节点都不会被封装在强制故障检测点和强制错误检测点之间.而步骤 5 和步骤 6 中的所有规则都不会将 W 状态叶节点加入强制故障检测点和强制错误检测点之间,因此,所有 Write 操作都位于成对的强制现场备份点和强制错误检测点之外.

综上所述,使用第 2.2.2 节的容错目标程序段的选择方法所插入的强制现场备份点和强制错误检测点满足 LazyFT 的容错框架及其约束,能够保证 GPGPU 程序最终运行结果的正确性.

2.3 实现机制

对于 LazyFT,在确定错误检测点和现场备份点的具体位置后,需要进一步设计适合该方法的错误检测机制和故障恢复机制.为了便于对具体机制的介绍,我们首先给出如下定义:

定义 1(不可靠数据). GPU 指令产生的可能含有错误的计算结果数据.

定义 2(不可靠数据相关计算). 可能改变不可靠数据数值的计算.

定义 3(相对活跃变量). 如果一个变量在某个容错目标程序段中被定过值,且首次操作为被引用,则我们称这个变量是该容错目标程序段的相对活跃变量.

2.3.1 错误检测机制

基于 Gregerson 等人的研究,我们选择时间双模冗余的方法实现面向 GPGPU 的故障检测[12].时间双模冗余故障检测的主要思想是,将需要容错的代码先后分别执行两遍,再对两次运行的计算结果进行比较.若发现不一致的结果,就意味着该段代码的运行过程中出现了故障[14].因此,时间双模冗余故障检测往往包含 3 部分工作:数据复制、冗余执行以及计算结果比较.与传统时间双模冗余故障检测相比,在 LazyFT 中,时间双模冗余故障检测需要进行如下修改:

- 首先,数据复制时,该错误检测方法并不需要复制所有数据,而只需复制不可靠数据,即所有可能作为 Kernel 输出的 GPU 数据,复制数据后不可靠数据存在原始和冗余两个版本;
- 其次,冗余执行时,该错误检测方法并不需要重复执行所有计算,而只需冗余执行不可靠数据相关计算,即所有 Kernel 调用和涉及不可靠数据的 Read 操作,具体冗余执行时只需将所有不可靠计算在程序中连续执行两遍即可,两遍计算所涉及的不可靠数据分别使用其原始版本和冗余版本;
- 最后,在比较计算结果时,该错误检测方法也只需比较所有不可靠数据,即在错误检测点将所有不可靠数据的原始版和冗余版进行比较.

2.3.2 故障恢复机制

当错误检测发现错误后,我们需要对容错目标程序段进行故障恢复,从而实现完整的容错功能.当前最常使用的故障恢复机制是 checkpoint-recovery 机制[15,16],该机制的基本思想是:在程序正确执行过程中,定期备份所有与运行现场有关的活跃变量至稳定的存储介质;当故障检测机制发现错误后,程序回滚至最近的备份现场,从稳定的存储介质中恢复之前备份的数据并重新运行程序.我们结合 LazyFT 的特点,提出了面向 GPGPU 的应用级 checkpoint-recovery 机制:

- 首先,在我们的故障模型中,故障只是发生在 GPU 计算过程中的瞬时故障,GPU 上的存储器和 CPU 的内存都是稳定的存储介质,都可用于存储备份数据;
- 其次,虽然故障只会发生在 GPU 上,并且不会影响 CPU 的计算结果,但现场备份除了备份 GPU 现场之外,还需要备份 CPU 的现场,以控制 GPU 的复算.同时,为 GPU 提供 Read 操作的数据来源;
- 最后,本文所要处理的是仅引起 SDC 错误的瞬时故障,所以,这里所需要备份的活跃变量只是本容错目标程序段的相对活跃的变量.那些未被赋过值的数据显然在程序回滚时无需恢复处理,而那些虽然在本容错目标程序段中被赋过值但首次操作为写操作的变量,可以通过备份其他变量间接恢复得到.

2.3.3 容错目标程序段的执行流程

利用上述面向 GPGPU 的错误检测机制和故障恢复机制,我们可以将一个容错目标程序段的容错处理过程抽象成如下执行流程,如图 8 所示.

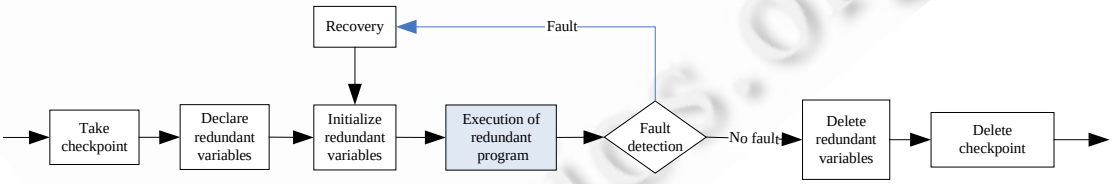


Fig.8 Executing flow of FT object segment

图 8 容错目标程序段执行流程

其中:现场备份是指将该容错目标程序段的 CPU 和 GPU 相对活跃变量分别备份至对应的备份变量中去;现场恢复是指将存储在备份变量中的数据恢复至对应的原变量中去;冗余变量声明是指为不可靠数据声明冗余版本;冗余变量初始化是指用原版不可靠数据初始化冗余版不可靠数据;原程序段冗余执行是指将原容错目标程序段进行计算,但在计算过程中必须将不可靠数据相关计算按第 2.3.1 节的要求先后执行两遍;故障检测是指通过比较所有 GPU 上原始和冗余不可靠数据对,只有当所有数据都一致时才判定为无故障;冗余变量删除就是删除这些冗余版不可靠数据;备份删除是指将现场备份时所声明的备份变量全部删除以释放空间.

需要指出的是,由于所有不可靠数据都驻留在 GPU 存储空间,为了提高故障检测和现场备份的效率,我们分别通过特定的 Kernel 函数实现 GPU 数据的错误检测和现场备份.与传统容错一样,我们假设在错误检测和现场备份期间不发生故障.

3 LazyFT 的优化

通过第 2.2 节所介绍的方法,程序员可以在编译器由静态分析得到的强制现场备份点和强制错误检测点插入编译指导命令,以阻止 CPU 错误的产生,从而确保程序最终运行结果的正确.但是,这些现场备份点和错误检测点是否能够使程序的总执行时间最优仍有待研究.

3.1 LazyFT执行时间模型

面向 GPGPU 的容错框架将原程序划分为若干容错目标程序段和非容错目标程序段,程序的运行过程就是容错目标程序段和非容错目标程序段交替执行的过程.假设程序的一次执行共经历了 N 次容错目标程序段或非容错目标程序段,则 $T=\sum_{i=1}^{N}T_i$, 其中,T_i 表示第 i 段的执行时间,而 T 则表示该程序的总执行时间.

容错目标程序段的执行时间主要由以下 3 部分组成[17]:原始执行时间、故障检测开销和复算开销.其中,原始执行时间是指不采取任何容错处理时原程序段的执行时间 t,故障检测开销主要包括冗余变量的初始化时间 t_init、冗余版不可靠数据相关计算的执行时间 $t\_modular$ 以及不可靠数据比较时间 t_detect,而复算开销主要包括现场备份时间 t_check、现场恢复时间 $t\_recover$ 和复算时间 t_redo.所以,如果假设一个容错目标程序段共进行了 n 次复算才通过故障检测,则该容错目标程序段的总执行时间可以由公式(1)来表示:

$$T_i=t\_check_i+t\_init_i+t_i+t\_modular_i+t\_detect_i+n_i\times(t\_recover_i+t\_redo_i) \tag{1}$$

由于现场备份和现场恢复所读写的数据是相同的,所以我们近似地认为 $t\_check\approx t\_recover$,而且 $t\_redo=t\_init+t+t\_modular+t\_detect$,则我们可以得到公式(2):

$$T_i=(n_i+1)\times(t\_check_i+t\_init_i+t_i+t\_modular_i+t\_detect_i) \tag{2}$$

为了在实际应用中通过上述公式对容错目标程序段的执行时间进行分析,我们必须获得公式中的各种参数.由于利用GPGPU计算的程序往往数据规模较大,所以我们可以通过profile的方法得到该容错目标程序段的原始执行时间 t 和冗余版不可靠数据相关计算的执行时间 $t\_modular$;再利用编译器根据第 2.3.1 节和第 2.3.2 节的要求静态分析不可靠数据和相对活跃变量的数据规模,从而根据数据复制速度、数据初始化速度以及数据比较速度分别计算得到现场备份时间 t_check、冗余变量初始化时间 t_init 和不可靠数据比较时间 t_detect.

通过上面的分析,公式(2)中只有参数 n 仍未获得.本文假设错误服从负指数分布,一个容错目标程序段计算过程中发生错误的概率 $p=\lambda\int_0^{t\_gpu}\mathrm{e}^{-\lambda t}\mathrm{d}t=1-\mathrm{e}^{-\lambda t\_gpu}$.其中:$\lambda$是负指数分布的参数,由 GPU 硬件自身的制造工艺以及运行时的电压频率所决定;t_gpu 表示该容错目标程序段的总 GPU 计算时间.而容错目标程序段的 GPU 计算时间主要包括 GPU 现场备份时间、GPU 冗余数据复制时间、原始版不可靠数据相关计算的执行时间、冗余版不可靠数据相关计算的执行时间以及故障检测比较时间.其中:GPU 现场备份时间是 t_check 的一部分,也可以通过与获得 t_check 的类似方法预估,令其为 $t\_checkgpu$;而两个版本的不可靠数据相关计算的执行时间是相同的,所以 $t\_gpu=t\_checkgpu+t\_init+2\times t\_modular+t\_detect$,则

$$p_i=1-\mathrm{e}^{-\lambda(t\_checkgpu_i+t\_init_i+2\times t\_modular_i+t\_detect_i)} \tag{3}$$

又因为

$$E(n_i)=\sum_{k=1}^{\infty}[kp_i^k(1-p_i)]=(1-p_i)\left(\sum_{k=1}^{\infty}p_i^k+\sum_{k=2}^{\infty}p_i^k+\ldots\right)=\frac{p_i}{1-p_i},\ 0<p<1,$$

所以,

$$E(T_i)=\frac{1}{1-p_i}\times(t\_check_i+t\_init_i+t_i+t\_modular_i+t\_detect_i) \tag{4}$$

至此,只要在容错目标程序段运行前通过诸如 profile 和编译器活跃变量分析等技术预估出该容错目标程序段的 *t_check*,*t_checkgpu*,*t_init*,*t*,*t_modular*,*t_detect* 等时间信息,并结合 GPU 硬件属性λ,就可以通过公式(3)和公式(4)求出该容错目标程序段执行时间的数学期望.

而当程序运行至非容错目标程序段时,其运行时间就是原始执行时间 *t*,但为了统一数学模型,我们仍使用公式(4)计算其执行时间的数学期望,只是非容错目标程序段的 *p*,*t_check*,*t_init*,*t_modular*,*t_detect* 都为 0.

3.2 容错目标程序段的优化

虽然通过第 2.2.2 节中给出的步骤,原程序已被分割成大粒度的容错目标程序段和非容错目标程序段,但是这些大粒度的容错目标程序段中可能包含多个 Kernel 调用和若干 CPU 代码,程序员是否有必要根据程序自身的特点插入一些备份点和检测点,从而将一个大容错目标程序段进一步分割成多个粒度较小的容错目标程序段和非容错目标程序段,是进一步需要解决的问题,也是优化容错代价的关键所在.为了便于下文介绍具体容错目标程序的优化选择,我们首先对两类程序段给出如下定义:

定义 4(顺序段). 位于一对强制备份点和强制检查点之间,且所有 Kernel 调用都按顺序执行的程序段.

定义 5(循环段). 循环体内不包含 Write 操作的循环程序段.

3.2.1 面向顺序段的最优容错

顺序段是强制备份点和强制检查点之间程序段的最基本程序结构,对其进行容错优化研究具有普遍的指导意义.顺序段进行容错的最小粒度就是将每个 Kernel 调用都独立地作为一个容错目标程序段,Kernel 调用之间的 CPU 代码段作为非容错目标程序段,我们称这种划分下的最小粒度计算段为基本计算段.优化的关键在于如何将基本计算段进行合并,从而组成粒度适中的容错目标程序段,使得整个顺序段执行时间的数学期望最小.为此,我们首先按照最小粒度将顺序段分成 *num* 个基本计算段,每个基本计算是由一个 Kernel 调用或 Kernel 调用间的 CPU 计算代码构成.

我们用 $T'_{x\sim y}$ 表示如果将第 *x* 段至第 *y* 段的连续基本段组合成一个容错目标程序段的执行时间,同时我们用 $T_{x\sim y}$ 表示从第 *x* 段至第 *y* 段这些连续段的最小执行时间,显然,

$$T_{x\sim y}=\begin{cases}T_x, & \text{if } (x=y)\\ \text{MIN}\{T'_{x\sim y},T_x+T_{x+1\sim y},T_{x+1}+T_{x+2\sim y},\ldots,T_{x\sim y-1}+T_y\}, & \text{else}\end{cases} \tag{5}$$

而 $T_{1\sim N}$ 就是整个顺序段的最优容错执行时间,至此,我们就可以基于第 3.1 节所分析的容错目标程序段和非容错目标程序段执行时间的数学期望,通过如下动态规划算法实现顺序段的最优备份点和检测点选择:

如图 9 所示,在进行最优容错目标段选择时,首先需要通过 profile 和编译器分析等方法预先得到所有可能容错目标段的 *t_check*,*t_checkgpu*,*t_init*,*t*,*t_modular*,*t_detect* 等信息,并基于这些信息通过公式(3)计算得到所有可能容错目标段的出错概率 *p*.

为此,我们将这些信息记录在二维数组之中,作为算法的输入,其中,下标为[*x*][*y*]的数据就表述由从第 *x* 段基本段至第 *y* 段基本段组成的容错目标程序段所对应的数据;接着,算法利用这些数据初始化每个基本计算段的执行时间和分割点;然后,算法通过动态规划的思想计算所有可能容错目标段的执行时间和分割点;最后,算法通过 outPath 函数跟踪分割点轨迹进行容错目标程序段的划分.

3.2.2 面向循环段的最优容错

顺序段是计算段的最基本的形式,但是很多科学计算程序的主体是一个循环迭代过程,虽然每次迭代中会顺序调用一系列 Kernel,但每次迭代计算时间很短.如果该循环是我们所定义的循环段,即循环中不存在 Write 操作,那么,我们可以将多次迭代组合成一个容错目标程序段,以多次迭代为粒度进行故障检测和现场备份,这样可以有效地降低容错开销.因此,我们通过如下数学模型指导用户如何选择循环段的最优容错粒度.

如果以一次迭代作为一个容错目标程序段进行数据备份和故障检测,令其相应的时间参数为 *t_check*(1),*t_checkgpu*(1),*t_init*(1),*t*(1),*t_modular*(1),*t_detect*(1),则其一次迭代的执行时间可以通过公式(3)、公式(4)计算得到.基于这些参数,我们建立以 *n* 次迭代为一个容错目标程序段时整个循环段的执行时间模型.首先,同一个循环

段的多次迭代与一次迭代所需要保存的现场是完全相同的,即 $t\_check(n)=t\_check(1)$,$t\_checkgpu(n)=t\_checkgpu(1)$;其次,多次迭代所需要的冗余初始化的数据与单次迭代也是相同的,即 $t\_init(n)=t\_init(1)$.类似地,我们可以得到 $t\_detect(n)=t\_detect(1)$;最后,n 次迭代的原始执行时间和冗余版不可靠数据相关计算执行开销都近似为一次迭代的 n 倍,即 $t(n)=n\times t(1)$,$t\_modular(n)=n\times t\_modular(1)$.所以,我们可以通过下列公式计算以 n 次迭代为一个容错目标程序段时整个循环段的执行时间.

$$p(n)=1-\mathrm{e}^{-\lambda(t\_checkgpu(1)+t\_init(1)+n\times 2\times t\_modular(1)+t\_detect(1))} \tag{6}$$

$$E(T(n))=\left[\frac{p(n)}{(1-p(n))^2}+1\right]\times[t\_check(1)+t\_init(1)+n\times(t(1)+t\_modular(1))+t\_detect(1)] \tag{7}$$

$$E(T)=\left|\frac{NUM}{n}\right|\times E(T(n))+E(T(rem)),\text{其中},rem=NUM\,\%n \tag{8}$$

基于上述公式不难发现,只要在循环段执行前通过 profile 方法得到以一次迭代作为容错目标程序段的相关时间开销,一个总共迭代 NUM 次循环的总执行时间只与容错目标程序段内的迭代次数 n 相关.由于公式比较复杂不利于求导,所以我们可以使用最简单的遍历方法得到循环段最优执行时间所对应的 n 值.

```
Input: bool CPU[num];   //whether segment num is CPU computing segment
       double t_check[][], t_checkgpu[][], t_init[][], t[][], t_modular[][], t_detect[][], p[][];
Output: bool CHECK[num], DETECT[num];   //whether a checkpoint(/detection point) should be
added before(/after) segment num
Algorithm:
int Path[][];   // Path[x][y] is used to record the best separation point between segment x and
segment y, and the value "-1" stands for no   separation point
{
    for (i=0; i<num; i++) {compute T[i][i] by formula (4);}
    for (i=0; i<num; i++) {Path[i][i]=−1;}
    for (i=1; i<num; i++)
                for (x=0; i+x<num; x++){
                y=x+i; Path[x][y]=−1;
                compute the execution time from segment x to segment y, and store it in
T[x][y];
                for (j=x; j<y; j++)   // traverse all possible   separation points
                        if (T[x][j]+T[j+1][y]<T[x][y])
                    T[x][y]=T[x][j]+T[j+1][y]; Path[x][y]=j;}
    outPath(0,n−1);   //determine checkpoint and detection point positions based on Path[][]
}
viod outPath(x,y){
    If (Path[x][y]==−1 && !(x==y &&   CPU[x])
                CHECK[x]=ture; DETECT[y]=ture; return;
    outPath(x,Path[x][y]); outPath(Path[x][y]+1,y);}
```

Fig.9 Optimization selective algorithm of optimization FT object for sequence segment

图 9 面向顺序段的最优容错目标段选择算法

4 案例分析

为了验证我们所提出的面向 GPGPU 的 LazyFT 容错方法的正确性并对其性能进行评价,我们首先用 Brook+语言实现了 spec2000 中的 Swim 测试程序,接着又利用现有的以单个 Kernel 为粒度的 Eager 容错方法和本文所设计的 LazyFT 容错方法分别实现了 Swim 程序的 Brook+容错版本,并将这些版本的程序分别进行了测试,测试平台及测试用例见表 1.

Table 1 Experiment platform and test specification

表 1 实验平台及测试用例

Heterogeneous platform	CPU	Intel Xeon E5405 4-core@2.0GHz, 256KB *L*1, 12MB *L*2
	GPU	AMD HD4870 RV770
Swim benchmark	Size Iterations	1023×1023 1 000

Swim 程序的主体是一个循环迭代,该迭代中不存在 Read/Write 操作,是一个循环段;每次迭代内部顺序调用 3 个计算段 cacl1,cacl2 以及 cacl3;每个 cacl 段内调用若干 Kernel 函数,是一个顺序段.为此,我们做了如下 4 组实验.

首先,我们分别测试了 Eager 容错和两种不同粒度的 LazyFT 容错方法在不发生故障时的容错代价.通过表 2 我们不难发现,以 cacl 顺序段为粒度进行容错处理的容错代价明显小于现有 Eager 的单 Kernel 容错.这主要是由于一个 cacl 顺序段内的多个 Kernel 调用之间存在大量的相同数据,所以与 cacl 顺序段容错相比,多个单 Kernel 容错时会对相同的数据进行重复备份、初始化以及比较检测;而单次迭代容错的容错代价却仅略优于 cacl 顺序段容错,这是由于一次迭代内的 3 个 cacl 顺序段容错之间重复的数据较少,因此单次迭代容错所能隐藏的容错处理也较少.

Table 2 Executing time of Swim without fault at different granularity

表 2 无故障情况下 Swim 测试在不同容错粒度下的执行时间

	Original program (s)	Eager FT (s)	LazyFT(s)	
Granularity of FT	None	Single Kernel	Cacl	Iteration
Backup time	0	5.955	2.949	2.523
Redundant variables initialization time	0	8.849	3.031	2.878
Redundant Kernel execution time	0	24.829	24.690	24.980
Fault detection time	0	70.564	25.127	23.865
Other time	24.653	25.411	25.036	25.322
Total time	24.653	135.608	80.833	79.568

其次,我们测试了当在无故障情况下以多次迭代为容错目标程序段时,LazyFT 容错代价随容错目标程序段粒度的变化情况.如图 10 所示,现场备份时间、冗余变量初始化时间以及故障检测时间与容错粒度的大小成反比,即上述 3 个时间与容错次数成正比.这是因为,无论以多少次迭代作为一个容错目标程序段,其对应的备份数据、冗余数据以及故障检测对象都是相同的;而冗余 Kernel 执行的时间只与程序固有的 Kernel 调用时间有关,并不随容错粒度的变化而变化.

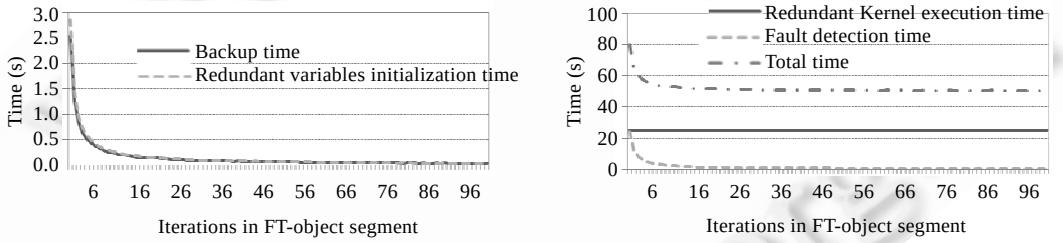


Fig.10 Cost curve of LazyFT for Swim without fault at different granularity

图 10 无故障情况下 Swim 测试面向循环段容错的额外开销随容错粒度变化曲线

接着,我们在假设 GPU 的平均无故障时间为 30s,即λ=1/30 的情况下,利用第 3.2.2 节所提出的面向循环段的最优容错方法进行最优容错目标程序段粒度的选择.首先,我们对 Swim 程序进行预处理,即以一次迭代为容错粒度对程序进行容错处理,并让程序执行一次迭代,进而得到以下信息:$t\_check(1)=t\_checkgpu(1)=0.002523$, $t\_init(1)=0.002878$,$t(1)=0.024653$,$t\_modular(1)=0.02498$,$t\_detect(1)=0.023865$.

然后,我们通过第 3.3 节中的公式计算得到程序总执行时间的数学期望 $E(T)$与容错粒度 n 之间的关系,如图 11 中“数学模型预估总时间”所示.为了验证这一结果的正确性,我们又通过在程序中模拟错误注入的方法,真实测得平均无故障时间为 30 时不同的容错粒度对程序最终执行时间的影响,如图 11 中“模拟执行实测总时间”所示.通过比较不难发现,第 3.2.2 节所提出的数学模型能够准确地预估真实的总执行时间.

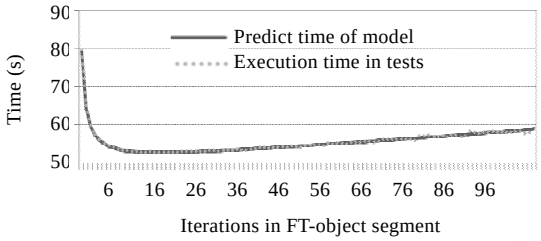


Fig.11 Executing time curve of loop segment at different granularity

图 11 面向循环段容错总执行时间随容错粒度变化曲线

最后,我们测试了 Swim 程序使用本文所提出的选择性容错方法,并基于第 3 节的数学模型进行容错目标程序段选择后,程序的容错性能情况.为此,我们以 Eager 容错方法作为比较的基准,忽略其故障恢复和复算时间,基准总执行时间为 135.608s,基准容错总开销为 110.955s.如图 12 所示,我们分别测试了不同平均无故障时间(mean time between failures,简称 MTBF)下,基于本文所提出的选择性容错方法,利用第 3.2.2 节数学模型指导选择的最优容错粒度进行容错处理的实际性能.实验结果表明,与现有的以单 Kernel 为粒度的 Eager 容错方法相比,使用最优容错粒度的选择性容错方法都可以得到较高的性能优势.随着平均无故障时间的增加,使用本文所设计的容错方法的容错 Swim 程序的容错开销减少 75%,总性能提高至 2.5 倍左右.

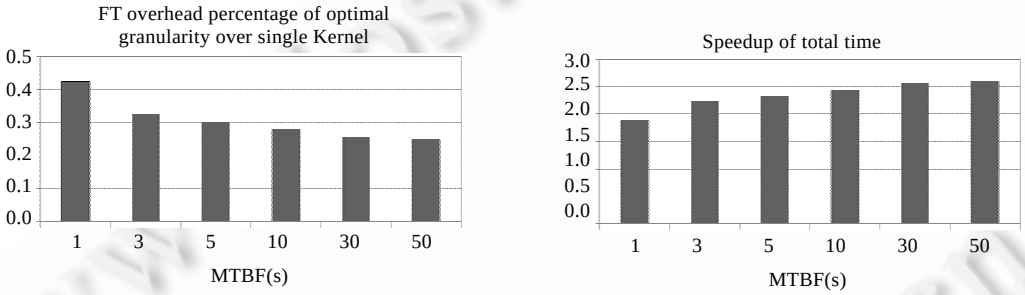


Fig.12 Performance of LazyFT at optimal granularity

图 12 最优容错粒度的选择性容错性能

5 相关工作

弗吉尼亚大学的 Sheaffer 和 NVIDIA 研究小组的 Luebke 等人在 2007 年指出,若在超级计算领域使用 GPGPU 并发挥其性能优势,就必须解决其可靠性问题.为此,他们设计了一种面向 GPGPU 可靠性的硬件冗余和恢复机制,该机制可以自动地检测故障并自动地启动故障恢复工作,而且故障恢复时只复算少量的错误数据[7].但是,他们所提出的容错机制是基于增加了冗余硬件支持的方法,并不适用于现在被广泛使用的普通 GPU.

中心佛罗里达大学的 Dimitrov 等人在 GPGPU 2009 Workshop 上针对 GPGPU 的可靠性问题提出了自己的软件容错方法,他们的工作主要是通过冗余执行 Kernel 的方式对 GPU 上发生的瞬时故障进行故障检测,并分析了不同的硬件支持对容错代价的影响[8].这样,他们就通过软件方法在一定程度上实现了对 GPGPU 的容错.但是,他们的工作主要集中在故障检测上,并未对故障恢复机制作深入研究,而且其容错工作的粒度也仅局限于单个 Kernel.

威斯康辛大学的 Gregerson 和 Abhyankar 等人分析了时间冗余、空间冗余、数据多样性冗余和基于算法的容错等多种传统软件容错方法在 GPU 上运用时的性能情况,得出如下结论:很多传统容错技术在 GPU 上的容错代价小于其在 CPU 上容错的代价;而且由于空间冗余往往需要额外的硬件支持,数据多样性冗余对浮点运算的支持不好,基于算法的容错又不具有通用性,而 GPU 的计算相对是比较廉价,所以 GPU 更适合使用时间冗

余计算的容错方法[12].

6 结 论

针对 CPU-GPU 异构系统的可靠性问题,为了降低 GPGPU 程序的容错代价,本文提出了一种全新的面向 GPGPU 的 LazyFT 容错方法.与现有的简单地面向单个 Kernel 调用的 Eager 容错方法不同,在我们的容错框架中,GPU 不再被孤立地作为一个加速部件,而是与 CPU 一起组成一个异构系统进行容错处理.本文首先建立了 GPGPU 故障模型,分析了 GPU 故障所引起的错误在 CPU-GPU 异构系统上的传播规律;并基于该规律提出了面向 GPGPU 的 LazyFT 容错框架,将异构系统上运行的程序区分为非容错目标程序段和容错目标程序段,提出了面向容错目标程序段的故障检测和故障恢复方法,并给出了容错目标程序段容错处理后的执行流程;最后,我们建立了面向 GPGPU 的容错执行时间模型,并在该模型下分析了面向顺序段和循环段两类典型科学计算程序的最优容错目标程序段选择方法.通过实验可以发现,本文所提出的面向 GPGPU 的 LazyFT 容错方法的容错代价明显低于现有的以单 Kernel 为粒度的 Eager 容错方法,而且基于我们所建立的面向 GPGPU 的容错执行时间模型可以比较准确地得到最优容错目标程序段的划分.

致谢 在此,我们向对本文的工作给予支持和建议的同行,尤其是国防科学技术大学计算机学院的唐玉华研究员、吴俊杰博士、王桂彬博士、贾佳博士和宋伟博士表示感谢.

References:

[1] Feng WC. The importance of being low power in high performance computing. CTWatch Quarterly, 2005,1(3):12−20. http://www.ctwatch.org/quarterly/articles/2005/08/

[2] Dally WJ, Hanrahan P, Erez M, Knight TJ, Labonté F, Ahn JH, Jayasena N, Kapasi UJ, Das A, Gummaraju J, Buck I. Merrimac: Supercomputing with streams. In: Proc. of the Supercomputing Conf. 2003 (SC 2003). 2003. 35−42. http://www.computer.org/portal/web/csdl/doi/10.1109/SC.2003.10043 [doi: 10.1109/SC.2003.10043]

[3] Pham D, Asano S, Bolliger M, Day MN, Hofstee HP, Johns C, Kahle J, Kameyama A, Keaty J, Masubuchi Y, Riley M, Shippy D, Stasiak D, Suzuoki M, Wang M, Warnock J, Weitzel S, Wendel D, Yamazaki T, Yazawa K. The design and implementation of a first-generation CELL processor. In: Proc. of the IEEE Int'l Solid-State Circuits Conf. (ISSCC 2005). 2005. 184−185. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1493930 [doi: 10.1109/ISSCC.2005.1493930]

[4] 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 Conf. on SIGGRAPH 2004 Course Notes. Los Angeles, 2004. 33−es. http://dl.acm.org/citation.cfm?id=1103933 [doi: 10.1145/1103900. 1103933]

[5] Owens JD, Luebke D, Govindaraju N, Harris M, Krüger J, Lefohn AE, Purcell TJ. A survey of general-purpose computation on graphics hardware. In: Proc. of the Eurographics 2005. 2005. 21−51. http://www.cg.informatik.uni-siegen.de/Teaching/Lectures/06_WS/Hauptseminar/02/GPUSurvey.pdf

[6] 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. 2004. 47. http://www.computer.org/portal/web/csdl/doi/10.1109/SC.2004.26 [doi: 10.1109/SC.2004.26]

[7] Sheaffer JW, Luebke DP, Skadron K. A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors. In: Proc. of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symp. on Graphics Hardware (GH 2007). 2007. 55−64. http://dl.acm.org/citation.cfm?id=1280104

[8] 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, New York: ACM Press, 2009. 94−104. http://dl.acm.org/citation.cfm?id=1513907 [doi: 10.1145/1513895.1513907]

[9] Gao L, Yang XJ. Error flow model: Modeling and analysis of software propagating hardware faults. Journal of Software, 2007,18(4):808−820 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/18/808.htm [doi: 10.1360/jos180808]

[10] Dubrova E. Fault Tolerant Design: An Introduction. Kluwer Acedemic Publischer, 2007. http://web.it.kth.se/~elena/draft.pdf

[11] Weaver C, Emer J, Mukherjee SS, Reinhardt SK. Techniques to reduce the soft error rate of a high-performance microprocessor. In: Proc. of the 31st Annual Int'l Symp. on Computer Architecture. München, 2004. 264. http://dl.acm.org/citation.cfm?id=1006723 [doi: 10.1109/ISCA.2004.1310780]

[12] Gregerson AE, Abhyankar AV. Performance cost analysis of software-implemented hardware fault tolerance methods in general-purpose GPU computing. 2009. http://homepages.cae.wisc.edu/~ece753/papers/Paper_4.pdf

[13] Muchnick SS. Advanced Compiler Design and Implementation. San Francisco: Morgan Kaufmann Publishers, 1998.

[14] Guerraoui R, Schiper A. Software-Based replication for fault tolerance. IEEE Computer, 1997,30(4):68−74. [doi: 10.1109/ 2.585156]

[15] Bronevetsky G, Fernandes R, Marques D, Pingali K, Stodghill P. Recent advances in checkpoint/recovery systems. In: Proc. of the Next Generation Systems Program Workshop at IPDPS. 2006. http://www.computer.org/portal/web/csdl/doi/10.1109/IPDPS.2006.1639575 [doi: 10.1109/IPDPS.2006.1639575]

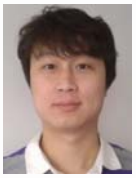
[16] Bronevetsky G, Schulz M, Szwed P, Marques D, Pingali K. Application-Level checkpointing for shared memory programs. In: Proc. of the Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2004. http://dl.acm.org/citation.cfm?id=1024421 [doi: 10.1145/1037947.1024421]

[17] Wu M, Sun XH, Jin H. Performance under failures of high-end computing. In: Proc. of the 2007 ACM/IEEE Conf. on Supercomputing. New York: ACM Press, 2007. 1−11. [doi: 10.1145/1362622.1362687]

附中文参考文献:

[9] 高珑,杨学军.错误流模型:硬件故障的软件传播建模与分析.软件学报,2007,18(4):808−820. http://www.jos.org.cn/1000-9825/18/808.htm [doi: 10.1360/jos180808]

徐新海(1984－),男,江苏镇江人,博士生,主要研究领域为计算机系统软件.



林一松(1983－),男,博士生,主要研究领域为计算机系统结构,编译优化.

杨学军(1963－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为计算机系统结构,计算机系统软件.

唐滔(1984－),男,博士生,CCF 学生会员,主要研究领域为计算机系统结构,编译优化.

林宇斐(1985－),女,博士生,主要研究领域为计算机系统软件.