基于D-Hpctoolkit的显存泄露检测算法

# 摘要

基于GPU加速的异构体系发展越来越快，越来越多的高性能程序以及深度学习框架使用GPU进行加速。具有显式显存管理的语言要求程序员手动释放不再需要的显存空间。程序员经常在编写程序的过程中忘记释放申请的空间从而导致程序崩溃等不可预测的现象发生。本文针对具有显式显存管理的程序提出了一种显存泄漏静态检测算法，在程序运行前通过静态检测能够及时发现是否存在显存泄漏问题。本文的算法支持多种异构体系编程语言，在NVIDIA GPU以及AMD DCU上的实验验证了算法的正确性，算法具有跨平台性。

# 关键词

显存管理；显存泄漏；代码分析；静态检测

# 引言

新兴的超级计算机越来越多地使用GPU加速器。 这些GPU加速的系统能够提供比仅使用常规多核处理器构建的系统更高的性能。 此类异构体系加速系统的使用日趋广泛，程序运行的安全性显得尤为重要，资源泄漏就是其中的一个方面。资源泄露可以粗略的分为：内存泄露和显存泄露。 内存泄露，一般是堆上的数据创建后，在程序退出时未释放导致的。类似于内存泄漏，显存泄露一般是显存资源在程序退出时未释放导致，显存的申请和释放需要程序员手动调用相应的函数进行操作，针对长时间运行的应用程序，如果忘记释放一块显存空间，到后期显存资源可能会不足，导致程序乃至系统的崩溃。

针对内存泄漏有静态和动态2种方式。动态监测因为易于操作设计简单等原因，较早的被工业界所使用，所以发展的相对比较完善。而静态检查工具因为算法复杂且对硬件要求较高，目前处于一个持续发展的阶段。静态检测技术可以尽早的发现软件中的安全漏洞，节约了开发成本，提高了软件系统的质量。【5】

针对显存泄漏，对于CUDA异构编程框架，可以使用NVIDIA提供的cuda-memcheck工具进行检测，是CUDA工具包中包含的功能正确性检查套件，可以检测cudaMalloc申请的尚未释放的显存空间，但是这个软件只能局限使用在NVIDIA平台，不能跨平台使用。

为了解决异构体系中显存泄漏的问题，本文提出了显存泄漏静态检测算法，该算法参考内存泄露静态检测方法，具有以下几个特点：

静态检测：在程序运行前进行分析，尽可能早的发现程序是否有显存泄漏问题。

对应用程序性能没有影响：动态检测由于需要在程序运行的过程中插桩或者设置中断捕获一些信息，对应用程序的实际运行效率是会产生或多或少的影响；静态检测不涉及应用程序的实际运行，所以不会影响实际运行效率。

跨平台性：由于是静态分析，只针对源码进行检测，所以可以将检测软件部署到不同的异构平台中应用。

# 相关工作

# 背景

## D-Hpctoolkit

Hpctoolkit是一套集成的工具套件，可支持对顺序和并行程序进行度量、分析，可视化展示应用程序性能。本文扩展了Hpctoolkit工具，使之能够静态分析NVIDIA GPU以及AMD DCU的应用程序是否有显存泄漏问题。扩展后的工具称为D-Hpctoolkit。

Hpctoolkit支持对顺序和并行程序的应用程序性能的度量、分析、属性和表示。在完全优化的并行程序中，Hpctoolkit可以精确地确定和量化可伸缩性瓶颈，其测量结果仅为几个百分点。最近，Hpctoolkit中增加了新的功能，可以在不需要任何编译器支持的情况下收集完全优化的代码的调用路径配置文件，可以在多线程程序中精确定位和量化瓶颈。Hpctoolkit使用硬件性能计数器的统计采样，并将度量标准归因于它们发生的调用上下文和程序结构，其中包括循环和内联过程。

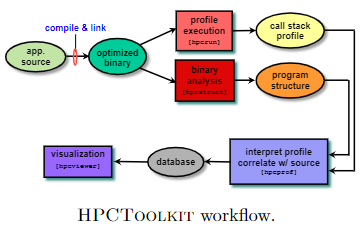
HPCToolkit通过以下几个步骤分析应用程序的相关性能[2]:

hpcrun以极低的开销（1-5％）收集未经修改的完全优化的应用程序的准确而精确的调用上下文敏感的性能度量。 它使用由系统计时器和性能监视单元事件触发的异步采样来驱动呼叫路径配置文件和可选跟踪的收集。

hpcstruct：将调用上下文敏感的度量与源代码结构相关联，hpcstruct分析完全优化的应用程序二进制文件，并恢复有关它们与源代码关系的信息。 特别是，hpcstruct将目标代码与源代码文件，过程，循环嵌套相关联，并标识内联代码。

hpcprof：hpcprof将调用路径配置文件和跟踪与hpcstruct计算的程序结构重叠，并将结果与​​源代码相关联。 hpcprof / mpi通过并行执行此关联来处理并行执行中的数千个配置文件。 hpcprof和hpcprof / mpi生成一个性能数据库，可以使用hpcviewer和hpctraceviewer用户界面进行浏览。

hpcviewer：hpcviewer是一个图形用户界面，它以三个互补的以代码为中心的视图（自上而下，自下而上和平坦）以交互方式呈现性能数据，以及使一个人可以评估线程和进程之间的性能可变性的图形视图。hpcviewer的设计目的是通过使用突出显示可伸缩性损失和效率低下的度量的派生指标来促进快速的自上而下的分析，而不是专门关注程序热点。



# 显存泄露静态检测算法

## 研究基础及概念定义

### 源程序抽象节点（Abstract Syntax Tree，AST）

AST是程序抽象表达式的一种形式，属于程序中间表达式，这个格式可以充分的表示代码中不同的词法单元如何组织在一起。语法树可以通过语法分析过程得到，对于分析代码的语法结构起到重要作用。完全符合文法规格要求的语法树（分析树）对于进一步处理并不是最合适的，所以在实际使用中，一般会对语法树进行适当的简化和修改。由于抽象语法树的表示简单，且方便使用，所以被广泛的应用在编译器开发和静态分析领域。

针对源程序的每一行进行如下抽象：

<node>

<lineNum>i</lineNum>

<code>code</code>

<nextLine>j1(,j2,j3…jn)</nextLine>

</node>

其中<node>代表一个节点，表示程序源代码中的某一行，i代表行号，code是源代码语句，j1~jn代表当前语句下一条语句的行号（一般针对表达式语句，nextLine为下一行语句的行号；针对控制语句，nextLine为多个分支行号）；用变量nodei代表第i行的<node>节点。

### 控制流图（Control Flow Graph，CFG）

控制流图是在代码解析过程中产生的一种中间表达式，是程序分析和静态检测中非常重要的一种代码表示方式。控制流图是一种反应程序逻辑控制流程的有向图。通常一个函数的控制流图可以表示为（N，E，Entry，Exit）。其中N代表节点的集合反映程序中的简单语句和复合语句的条件判断以及控制流汇合点等，E代表有向边的集合，反映程序中语句间的控制流关系。Entry为函数的固定唯一入口节点，Exit为函数唯一的推出节点。简单通俗的说：控制流图即时具有单一的、固定的入口节点和出口节点的有向图。对于非单入口和单出口的程序，可以通过人为添加一个统一入口和出口的方法解决。控制流图可以用于分析程序源代码的分支流向，用于确定代码块之间的调用关系。控制流图中的节点为上一节中抽象语法树生成的节点

### 指针地址映射集合（P-M Set）

显存中的每一个地址都会有一个变量与之对应，显存的申请和释放都是基于指针变量进行的操作。

指针地址集合定义如下。令P表示程序中显存指针变量集合，M表示显存地址集合，对于Pi∈P，Mi∈M，{<{Pi},Mi>}表示指针地址映射集合。

本文将通过分析静态程序的抽象语法树，对指针地址映射集合进行增删改操作，来记录程序对显存的操作。最后通过分析指针地址映射集合里是否有空指针或者还未释放的显存地址。

设每一次操作的指针变量为pointer，显存地址为memory(由于是静态分析，所以不能获取真实的显存地址，本文用当前时间进行代替)。

针对指针地址映射集合的操作如下。

（1）malloc(<pointer,memory>)：将指针pointer和显存地址插入到P-M Set中，主要针对cudaMalloc以及hipMalloc函数申请显存。

算法：向P-M Set中添加指针地址映射

输入：指针变量名pointer，显存地址memory

输出：无

Begin

获取当前时间time，将time赋值给memory

新建指针地址映射关系<{pointer},memory>

将<{pointer},memory>插入到P-M Set当中

End

（2）free(pointer)：将指针pointer所指向的显存地址释放，主要针对cudaFree以及hipFree函数释放显存。

算法：从P-M Set中删除pointer指针以及pointer指向的显存地址

输入：指针变量名pointer

输出：显存检测错误码

Begin

遍历每一组<{pointers},memory>，取出指针变量集合{pointers}

判断pointer是否在集合{pointers}当中

如果存在，将当前<{pointers},memory>映射删除，退出循环并返回SUCCESS

如果不存在，继续下一次遍历

遍历结束都没有找到当前pointer，记录一个重复删除异常并返回DUPLICATE\_ERR

（3）modify(<pointeri,memoryi>,<pointerj,memoryj>)：对指针变量进行赋值修改。针对i = j这样的赋值操作。

算法：修改P-M Set集合的元素

输入：指针地址映射关系<pointeri,memoryi>,<pointerj,memoryj>

输出：无

Begin

遍历每一组<{pointers},memory>，取出指针变量集合{pointers}

寻找pointeri所在的集合<{pointers}i,memoryi>

寻找pointerj所在的集合<{pointers}j,memoryj>

如果memoryi和memoryj是同一块显存地址

不做任何操作

如果memoryi和memoryj不是同一块显存地址

将pointeri从{pointers}i中移除，并添加到{pointers}j当中

（4）check()：在程序结束的时候对指针地址映射集合进行遍历，查看是否还有未释放的显存空间。

算法：遍历P-M Set集合

输入：无

输出：显存检测错误码

Begin

判断P-M Set是否为空

如果为空，返回SUCCESS

如果不为空，遍历每一组<{pointers},memory>

如果{pointers}为NULL，返回WILDPOINTER\_ERR

如果{pointers}不为NULL，返回REAMINPOINTER\_ERR

下面通过一段程序进行说明

1 int \*pointer1 = NULL;

2 int \*pointer2 = NULL;

3 size\_t size = 1024\*sizeof(int);

4 cudaMalloc((void\*\*)&pointer1, size);

5 cudaMalloc((void\*\*)&pointer2, size);

6 pointer1 = pointer2;

7 cudaError\_t err1 = cudaFree(pointer1);

8 cudaError\_t err2 = cudaFree(pointer2);

9 return;

指针地址映射集合状态转换表

|  |  |  |
| --- | --- | --- |
| 代码行号 | 注释及操作 | 指针地址映射集合状态 |
| 1、2、3 | 初始化指针以及显存大小 | {} |
| 4 | 为pointer1申请显存地址  malloc(<{pointer1}, 1576160766>) | {<{pointer1}, 1576160766>} |
| 5 | 为pointer2申请显存地址  malloc(<{pointer2}, 1576161000>) | {<{pointer1}, 1576160766>,<{pointer2}, 1576161000>} |
| 6 | 修改pointer1的值  modify(<{pointer1}, 1576160766>,<{pointer2}, 1576161000>) | {<{NULL}, 1576160766>,<{pointer1,pointer2}, 1576161000>} |
| 7 | 释放pointer1指向的显存  free(pointer1) | {<{NULL}, 1576160766>} |
| 8 | 释放pointer2指向的显存  free(pointer2) | {pointers}集合中不包含pointer2，报重复释放的错误 |
| 9 | 程序结束，遍历P-M Set  check() | 因为P-M Set不为空，并且包含NULL指针，所以返回WILDPOINTER\_ERR |

### 显存泄露故障检测基本思想

针对比较底层的显卡编程语言，比如CUDA以及HIP，应用程序需要cudaMalloc或hipMalloc等函数申请显存空间，当程序使用完相应的空间后，必须手动的调用cudaFree或hipFree等函数释放空间，否则随着程序不断的申请，显存总有被用完的时候，到时候会引起程序乃至系统的崩溃。

本文主要检测以下两个显存泄露类型：

1. 申请的显存空间忘记释放：针对只调用cudaMalloc、hipMalloc，而没有匹配的调用cudaFree、hipFree。
2. 申请的显存空间多次释放：针对申请的同一块显存空间被多次调用cudaFree、hipFree释放。

显存泄露故障检测流程如下：

1. 将源程序转化为抽象语法树的结构。
2. 逐行分析抽象语法树，分析的过程中建立并维护指针地址映射集合，针对分支判断变量的计算采用集合运算。
3. 如果变量集合符合判断条件，则进入该分支进行执行
4. 如果变量集合不符合判断条件，则跳过该分子往下执行
5. 循环（2）~（4）直到程序结束

# NVIDIA GPU显存分析

例举例说明：cudaDemo1.cu

#include <stdio.h>

#include <stdlib.h>

#include <cuda\_runtime.h>

#define CHECK(cmd) \

{\

cudaError\_t error = cmd;\

if (error != cudaSuccess) { \

fprintf(stderr, "error: '%s'(%d) at %s:%d\n", cudaGetErrorString(error), error,\_\_FILE\_\_, \_\_LINE\_\_); \

exit(EXIT\_FAILURE);\

}\

}

int main(){

int flag = 0;

int \*pointer1 = NULL;

int \*pointer2 = NULL;

int size1 = sizeof(int) \* 1024;

int size2 = sizeof(int) \* 2048;

CHECK(cudaMalloc((void\*\*)&pointer1, size1));

CHECK(cudaMalloc((void\*\*)&pointer2, size2));

if(flag == 0){

CHECK(cudaFree(pointer1));

}

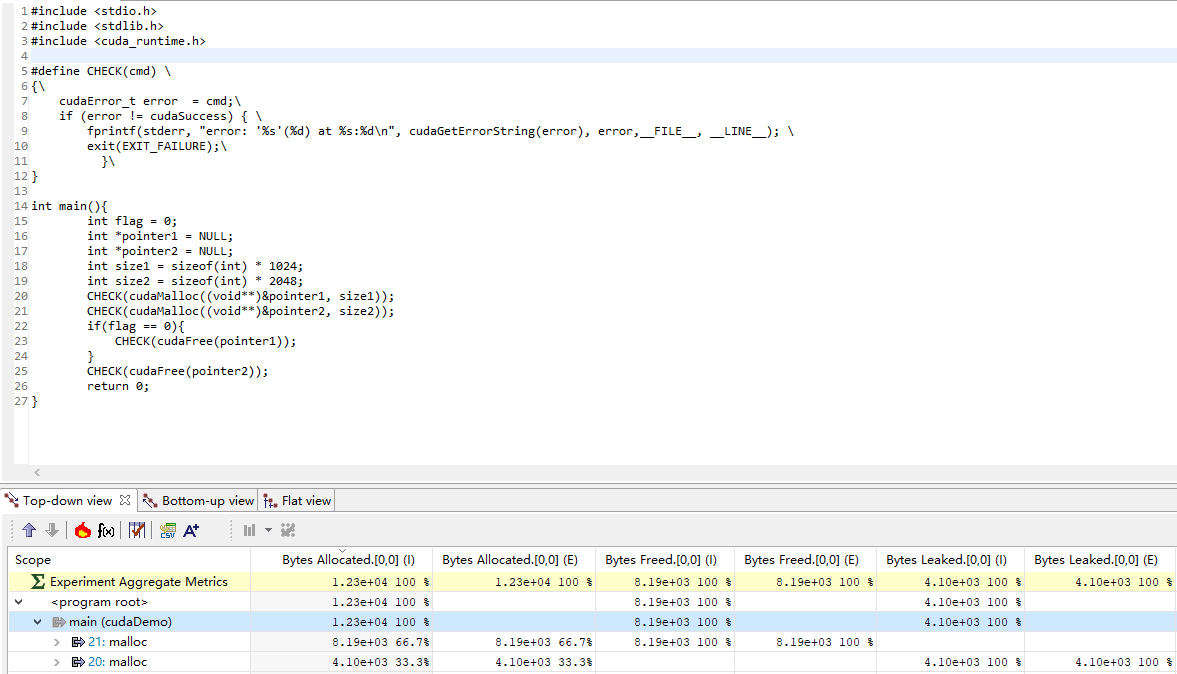
CHECK(cudaFree(pointer2));

return 0;

}

其中CHECK函数是检测所执行的cmd表达式返回值是否正确；在main函数中，首先为通过cudaMalloc分别申请了size1和size2大小的2块显存空间，分别存储在pointer1和pointer2变量中，然后进入条件判断，因为flag为0，所以程序正常释放pointer1和pointer2两块显存空间，无显存泄漏。结果如图1所示，xxxxxxx。

如果将flag改为1，再执行一遍显存泄露故障检测程序，结果如图2所示，发现pointer申请的显存没有释放导致显存泄露。



# AMD DCU显存分析

例举例说明：cudaDemo1.cpp

#include <stdio.h>

#include <stdlib.h>

#include <cuda\_runtime.h>

#define CHECK(cmd) \

{\

cudaError\_t error = cmd;\

if (error != cudaSuccess) { \

fprintf(stderr, "error: '%s'(%d) at %s:%d\n", cudaGetErrorString(error), error,\_\_FILE\_\_, \_\_LINE\_\_); \

exit(EXIT\_FAILURE);\

}\

}

int main(){

int flag = 0;

int \*pointer1 = NULL;

int \*pointer2 = NULL;

int size1 = sizeof(int) \* 1024;

int size2 = sizeof(int) \* 2048;

CHECK(cudaMalloc((void\*\*)&pointer1, size1));

CHECK(cudaMalloc((void\*\*)&pointer2, size2));

if(flag == 0){

CHECK(cudaFree(pointer1));

}

CHECK(cudaFree(pointer2));

return 0;

}

# 总结

## 未来工作

# 参考文献

[1]HPCTOOLKIT: Tools forperformance analysis ofoptimized parallel programs

[2] HPCToolkit: performance tools for scientific computing

[3] Rice University 2008 HPCToolkit performance tools.http://www.hipersoft.rice.edu/hpctoolkit

[4] Effective Sampling-Driven Performance Tools for GPU-Accelerated Supercomputers

【5】基于静态分析的软件内存漏洞检测——刘航源