

GPU 平台上面向性能和功耗的分支优化

于 齐 王博千 沈 立 王志英 陈 微

(国防科学技术大学计算机学院 长沙 410073)

摘 要 强大的计算能力使得 GPGPU 在通用计算领域得到了广泛的应用。然而, GPGPU 的 SIMT(Single Instruction Multiple Threads)工作方式, 使其执行效率受到应用中不一致分支行为(Branch Divergence)的严重影响。虽然人们提出了线程交换方法来减小分支带来的性能损失, 但这种方法往往会引入额外的访存操作, 不仅在一定程度上减少了线程交换优化的性能收益, 还增加了功耗。首先举例说明线程交换范围对程序性能和功耗的影响; 然后提出了一种减少线程交换所引入的额外访存操作的方法。实验表明, 对于 Reduction 程序, 当交换范围为 256 时, 在性能平均损失为 4% 的情况下功耗降低幅度最大为 7%; 而对于 Bitonic 程序, 当交换范围为 256 和 512 时, 在没有功耗开销的情况下, 性能分别最大提升了 6.4% 和 5.3%。

关键词 不一致分支行为, 访存, 线程交换

中图法分类号 TP303 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2016.5.004

Branch Divergence Optimization for Performance and Power Consumption on GPU Platform

YU Qi WANG Bo-qian SHEN Li WANG Zhi-ying CHEN Wei

(School of Computer, National University of Defense Technology, Changsha 410073, China)

Abstract Because of the tremendous computing power, general purpose graphics processing units(GPGPUs) have been widely accepted in general purpose computing area. However, as GPGPUs using an execution model called SIMT(Single Instruction Multiple Threads), their efficiency is subject to the presence of branch divergence in a GPU application. People have proposed a method based on thread swapping to reduce the performance loss brought by branch divergence, but these methods always bring extra memory accesses in return, which not only decrease the performance gains to a certain degree, but also increase power consumption. Firstly, an example was used to explain the influence thread swapping range has on performance and power consumption of a program. Secondly, a method was proposed to reduce the extra memory accesses brought by thread swapping. Experiments show that, for Reduction, this method reduces power consumption by 7% with average performance loss by 4% when swapping range is 256. While for Bitonic, this method improves performance by 6.4% and 5.3% when swapping range is 256 and 512 with no power consumption overheads, respectively.

Keywords Branch divergence, Memory access, Thread swapping

1 引言

近几年, GPGPU 在通用计算领域得到了广泛的应用, 成为当前高性能计算系统的重要组成部分。许多应用被成功移植到 CPU+GPU 的异构平台上, 得到了很好的加速效果。GPU 的优势在于其强大的并行计算能力以及良好的可编程性。以 NVIDIA 公司的 GPU 产品为例, 每个 GPU 中含有多个流多处理器(Streaming Multiprocessor, SM)。使用 CUDA^[1] 编程模型编写的程序在 GPU 上执行时会产生成百上千个线程。这些线程被划分成固定大小的线程束(warp)^[2], 若干个 warp 组成一个线程块, 线程块被分配到一个 SM 上执

行, 从而实现高度的线程级并行。

warp 中的线程按照 SIMD(Single Instruction Multiple Data)模式执行——它们执行相同的代码, 但处理不同的数据。当遇到分支指令时, 执行不同分支路径的线程只能串行执行。这种因不一致分支行为引起的串行执行现象被称作 Branch Divergence, 它的出现严重影响了 GPU 的执行效率, 降低了程序性能。针对这个问题, 人们提出了线程交换的优化方法^[3], 其主要思想为: 将具有相同分支行为(即执行相同分支路径)的线程放在同一个 warp 中, 这样 warp 中的线程就可以并行执行, 从而达到减少 Branch Divergence、提高性能的目的。

到稿日期: 2015-05-18 返修日期: 2015-07-18 本文受国家自然科学基金项目(61472431, 61202121), 教育部高等学校博士点新教师基金项目(20114307120013)资助。

于 齐(1990—), 男, 硕士, 主要研究方向为系统结构, E-mail: FishFlag@126.com; 王博千(1990—), 男, 硕士, 主要研究方向为系统结构; 沈 立(1976—), 男, 副教授, 主要研究方向为系统结构; 王志英(1956—), 男, 教授, 主要研究方向为系统结构、信息安全; 陈 微(1982—), 女, 讲师, 主要研究方向为高性能微处理器设计。

线程交换既可以在编译时静态完成,也可以在运行时动态进行。虽然这种方法能够减少 Branch Divergence 对程序性能的负面影响,但是这种方法也会带来额外的访存操作,比如非合并访存、访问共享存储器(Shared Memory)体的冲突等。这些额外的访存操作不仅在一定程度上抵消了线程交换在性能上的收益,而且还会带来额外的功耗开销。因此,本文关注如何减小线程交换带来的额外访存操作以及其引起的性能和功耗损失。

本文的工作主要包括以下几个方面:

1) 研究了线程交换对程序功耗的影响。目前针对 Branch Divergence 提出的软件优化方法^[3,4]都只关注性能的提升,没有考虑对功耗的影响。在计算系统的功耗越来越受到关注的今天,研究如何减小线程交换对程序功耗的影响具有重要的意义。

2) 研究了线程交换范围对程序性能和功耗的影响,并通过控制交换范围来控制性能收益和功耗损失。即在进行线程交换优化时,可以选择不同的交换范围,它们能够消除的 Branch Divergence 不同,引起的额外开销也不一样,优化时应该选择效果更好的交换范围。

在确定交换范围的基础上,为了说明如何减少线程交换带来的额外访存操作,选择 Reduction^[3]和 Bitonic^[5]两个程序进行线程交换优化以及性能测试。测试结果表明,Reduction 程序在性能损失较小的情况下有效降低了功耗,而 Bitonic 程序在没有额外功耗开销的情况下提升了性能。

2 GPGPU-Sim 简介

本文选择 GPGPU-Sim v3.2.2^[6] 模拟器作为实验平台,收集性能和功耗数据。GPGPU-Sim 是一个关注 GPU 通用计算的时钟级别的 GPU 性能模拟器^[7],其模拟的 GPU 结构如图 1 所示。

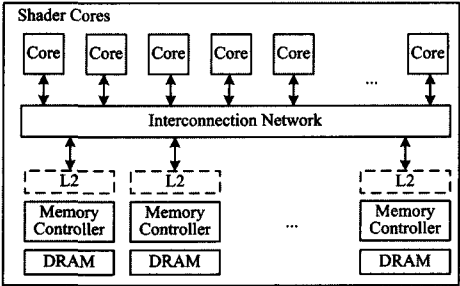


图 1 基础的 GPU 结构

本文使用的 GPU 型号为默认的 NVIDIA GTX-480,它采用 Fermi 架构,含有 15 个 SM,每个 SM 中含有两个 SP (Shader Processor)单元、一个 load/store 单元,以及一个用于完成复杂运算的特殊功能单元(Special Function Unit,SFU)。每个 SP 单元含有 16 个双倍时钟的 CUDA 核,每时钟周期可以执行 32 个线程,每个 CUDA 核含有一个独立的整型和浮点型流水线^[8]。

GPGPU-Sim 中内嵌了 GPUWattch^[9]来收集功耗数据。GPUWattch 的结构如图 2 所示。其中,McPAT 是一个用于精确估计片上多处理器(CMP)功耗和面积的结构模型,该模型被修改以适应 GPU 特殊的结构组件。工作时,GPGPU-Sim 将性能计数器的值通过接口传递给修改的 McPAT,GPGPU-Sim 估算出静态和动态功耗^[10]。

GPUWattch 的功耗输出文件中不仅列出了各个 kernel 在采样期间的平均功耗、最大功耗、最小功耗,还统计了各个性能计数器的采样值,根据这些采样值计算得到各个功能单元(取指、运算、访存等单元)的功耗(以上功耗均为动态功耗),这对于分析功耗变化的原因以及降低额外功耗开销具有重要作用。

此外,GPGPU-Sim 的输出数据中有一个叫做 gpgpu_n_stall_shd_mem 的性能计数器,其记录了在访存阶段由于非合并访存和访问共享存储器体冲突引起的停顿时钟数。本文以该性能计数器来衡量线程交换引起的访存开销。

3 线程交换范围

本节分析线程交换范围对程序性能和功耗的影响。所谓线程交换范围是指线程交换时交换单元中线程的数量,一般为 warp 中的线程数(32)的整数倍。例如交换范围为 64,就是在两个连续的 warp 内进行交换。一般来说,交换范围越大,Branch Divergence 被减少的可能性也会越大,收益也可能越高,但同时产生的额外访存的操作以及其带来的性能损失和功耗也越大。下面以图 3 所示程序为例来说明交换范围对收益与开销的影响。

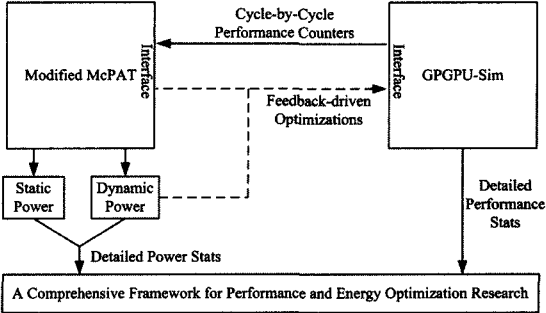


图 2 GPUWattch 结构

该程序根据线程号 *tid* 被 4 和 2 整除的情况执行不同的操作。在线程交换优化时,可以选择两种不同的交换范围,即 64(即 2 个相邻的 warp)和 128(即 4 个相邻的 warp)。

```
if(tid % 4 == 0){
    d_c[tid] = d_a[tid] * d_b[tid];
}
else if((tid % 4 != 0) && (tid % 2 == 0)){
    d_c[tid] = d_a[tid] + d_b[tid];
}
else{
    d_c[tid] = d_a[tid] - d_b[tid];
}
```

图 3 示例程序核心代码

每种交换范围的效果如下:

• 当交换范围为 64 时,即以连续 2 个 warp 为一组进行交换,线程映射关系(*i* 为偶数)如表 1 所列。

表 1 交换范围为 64 的线程映射表

warp _{<i>i</i>}	warp _{<i>i</i>+1}
tid % 4 == 0	tid % 2 != 0
tid % 4 != 0 && tid % 2 == 0	

从表 1 可以看出,优化前 warp_{*i*}、warp_{*i*+1} 都存在 Branch Divergence,优化后只有 warp_{*i*} 存在 Branch Divergence,总数减少了 50%。

对于访存的分析,以 $warp_i, warp_{i+1}$ 读取一次 $d_a[tid]$ 为例来说明。访存次数如表 2 所列。

表 2 交换范围为 64 的访问次数表

	$warp_i$	$warp_{i+1}$
优化前	3	3
优化后	4	2

虽然优化前后 $warp_i, warp_{i+1}$ 产生的总访存请求次数一样,但是优化前 $warp_i$ (或者 $warp_{i+1}$) 中线程访问的数据位于存储器的同一段中,只需一次合并访存即可获得全部数据;优化后 $warp_i$ (或者 $warp_{i+1}$) 中线程访问的数据地址不连续且位于存储器的不同段中,读取这些数据需要多次访存操作,这会导致访存阶段的停顿增加,从而会部分抵消由于减少 Branch Divergence 带来的性能收益。优化前后程序模拟时钟 (运行时间) 的差值与访存开销的差值之间的关系如图 4 所示。

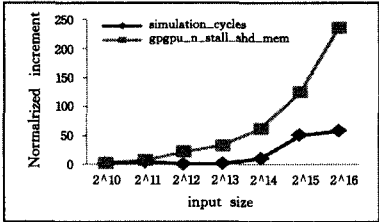


图 4 交换范围为 64 时访存开销与性能的关系

从图 4 可以看出,随输入数据集(input size)增大,非合并访存引起的性能开销也不断增大,并导致优化后时间(模拟时钟数目)增加。这说明线程交换引起的非合并访存开销成为影响最终性能收益的重要因素。

• 当线程交换范围为 128 时,即以连续 4 个 warp 为一组进行交换,线程映射关系(i 为 4 的倍数)如表 3 所列。

表 3 交换范围为 128 的线程映射表

$warp_i$	$warp_{i+1}$	$warp_{i+2}$	$warp_{i+3}$
$tid \% 4 = 0$	$tid \% 4 = 0 \& \&$ $tid \% 2 = 0$	$tid \% 2 = 0$	$tid \% 2 = 0$

优化前,4 个 warp 中都存在 Branch Divergence,而优化后这些 Branch Divergence 被完全消除了。

对于访存的分析,仍以 $warp_i, warp_{i+1}, warp_{i+2}, warp_{i+3}$ 读取一次 $d_a[tid]$ 为例来说明,访存次数如表 4 所列。

表 4 交换范围为 128 的访存次数表

	warp _i	warp _{i+1}	warp _{i+2}	warp _{i+3}
优化前	3	3	3	3
优化后	4	4	2	2

与交换范围为 64 时类似,虽然优化前后总访存次数一样,但优化后由非合并访存引起的停顿增加了。优化前后程序模拟时钟的差值与访存开销的差值之间的关系如图 5 所示。

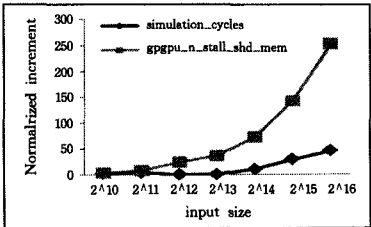


图 5 交换范围为 128 时访存开销与性能的关系

两种交换范围下性能以及功耗的变化如图 6 和图 7 所

示。从图 6 可以看出,两种交换范围获得的加速比小于 1,这是因为线程交换引起的访存开销过大,超过了其收益。从图 7 可以看出,对于两种交换范围,功耗降低较为明显,并且交换范围为 128 时功耗降低的幅度大于交换范围为 64 时功耗降低的幅度。

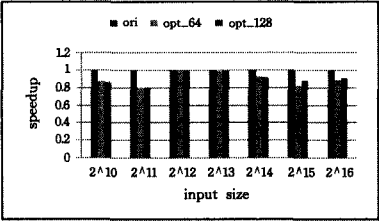


图 6 两种交换范围下的加速比

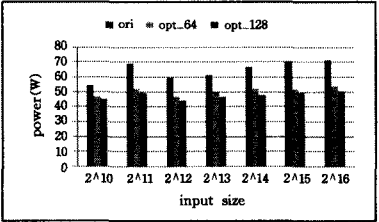


图 7 两种交换范围下功耗的变化

通过该示例程序可以看出,交换范围越大,降低 Branch Divergence 的程度越高,同时带来的访存开销也越大。因此,使用线程交换方法时,需要综合考虑,选择合适的交换范围,使得优化后的程序在性能和功耗之间达到更好的平衡。

4 测试结果及分析

本节选择 Reduction 和 Bitonic 两个应用程序来说明本文所提方法在性能和功耗方面的优化效果。其中,Reduction 采用基于树的方法计算数组的和,在很多领域都有应用;Bitonic 作为一种基础性的排序算法,在多个领域得到了广泛应用[5]。

4.1 Reduction

文献[3]介绍了使用线程交换方法在 GPU 平台上优化 Reduction 程序的过程。按照其描述,它所选择的交换范围为 64。通过对该交换范围进行测试,发现优化后获得的加速比较小,不超过 1.05,对于某些输入集,加速比甚至小于 1,性能反而有所下降;优化后功耗有所降低,幅度在 5% 以内。程序定义的块大小为 256,因此有效的交换范围为 64、128、256。在文献[3]的基础上,本文测试了另外两种交换范围的性能和功耗,结果如图 8 和图 9 所示。

从图 8 可以看出,加速比随交换范围的增大而增大,交换范围为 256 时获得的加速比最高;从图 9 可以看出,功耗随交换范围的增大而增大,交换范围为 64 时功耗最小,而交换范围为 128 和 256 时,功耗比优化前略有增加。通过分析各个功能单元的功耗,发现访问共享存储器的功耗增加是导致总功耗增加的主要原因。

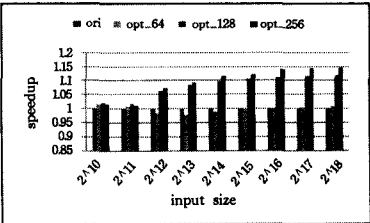


图 8 不同交换范围下 Reduction 的加速比

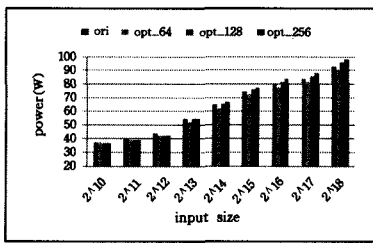


图 9 不同交换范围下 Reduction 的功耗

通过分析 kernel 代码,可以看出在执行归约操作前,需要先把数据从全局存储器拷贝到共享存储器,后续数据在共享存储器中读写,这样就会产生大量访问共享存储器的请求。GPGPU-Sim 的 GTX-480 处理器将共享存储器分成 32 个体。线程交换前, warp 中每个线程访问其中的 1 个体,不存在体冲突;而在优化后,出现了 warp 中的几个线程访问同一个体的情况,从而引起了体冲突。

与全局存储器不同,由于共享存储器访问速度接近寄存器的访问速度,虽然交换范围越大产生的体冲突越多,但是对性能造成的影响并不大,这就是交换范围为 256 时的加速比会大于交换范围为 64 和 128 时的加速比的原因。另一方面,体冲突增加会导致访问共享存储器的次数增加,引起共享存储器的功耗增加。因此,交换范围为 256 时的功耗大于交换范围为 64 和 128 时的功耗。

针对体冲突增加的问题,本文提出了一个简单的解决方法,即改变数据在共享存储器中存储的位置,使得 warp 中线程访问的数据存放在共享存储器中的不同体中。下面结合图 10 和图 11 的代码进一步说明。

```
unsigned int tid=threadIdx.x;
tid=dmap[tid];
unsigned int i=blockIdx.x*blockDim.x+tid;
sdata[tid]=(i<n)? g_idata[i]:0;
```

图 10 优化前的代码

```
unsigned int tid=threadIdx.x;
unsigned int newtid=dmap[tid];
unsigned int i=blockIdx.x*blockDim.x+newtid;
sdata[tid]=(i<n)? g_idata[i]:0;
```

图 11 优化后的代码

如图 10 所示,优化前,数据按照索引由小到大的顺序写入共享存储器,交换后的线程访问的数据可能位于同一个体中;而在图 11 中,交换后的线程要访问的数据按新线程号(newtid)的顺序写入共享存储器的不同体中,减少了大部分体冲突。此外,还需要在主机端代码中构造一个反向映射的数组,用于辅助对共享存储器的访问,这会使主机端产生额外的开销。由于反向映射数组的大小与块大小(256)相等,因此这一开销并不大;同时 GPU 端会产生额外的访存(访问反向映射数组),这会对优化效果产生一定影响。

图 12 和图 13 以交换范围为 256 来说明这种优化方法对性能和功耗的影响。从图 12 可以看出,该方法对性能产生一定的影响,即优化后加速比有所降低;而从图 13 可以看出,优化后功耗有所降低。总的来说,在性能平均损失 4% 的情况下,功耗最高降低了 7%。

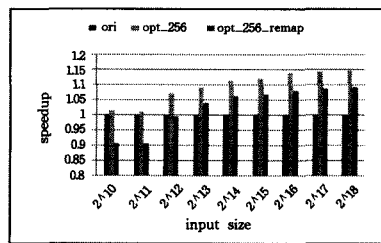


图 12 Reduction 再优化前后的加速比

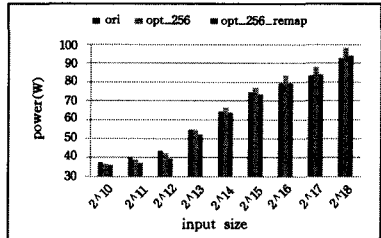


图 13 Reduction 再优化前后功耗的变化

4.2 Bitonic

文献[5]对 Bitonic 程序的原理以及优化策略做了详细说明,本文在此基础上做进一步优化。程序中块大小定义为 512,因此有效的交换范围为 64,128,256,512。文献[5]的优化根据是条件(1),即将满足该条件的线程号以 32 为一组映射到同一个 warp 中,而将不满足该条件的线程号映射到另外的 warp 中,以此达到减少 Branch Divergence 的目的。

$$idx^j > idx \quad (1)$$

通过对满足条件(1)的线程号做进一步分析,发现了下面的规律,即当满足条件(2)时,条件(1)恒成立。

$$j > idx \quad (2)$$

假设交换范围为 64,则当 $j \geq 64$ 时,63 以内的线程号恒满足条件(1),此时就不再需要做线程交换,这样会带来两个好处:1)可以减少主机端构建线程映射表的工作,减少主机向 GPU 传输线程映射表的数据量;2)由于线程交换需要在 kernel 中添加额外的线程映射语句,该语句需要访问全局存储器,这样就可以减少对全局存储器的访问,进而减少由于非合并访存引起的开销。

通过进一步优化,GPU 端的执行时间有一定程度的减少,功耗变化较小。以交换范围为 256 和 512 进行说明,如图 14 和图 15 所示。

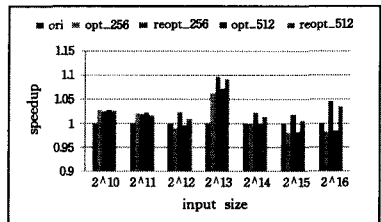


图 14 Bitonic 再优化前后的加速比

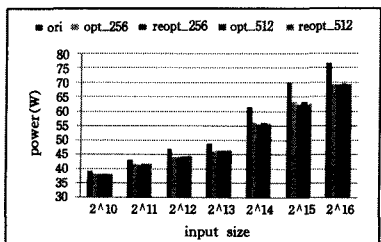


图 15 Bitonic 再优化前后功耗的变化

从图 14 可看出,在没有进一步优化前,线程交换获得的加速效果并不理想,对于某些输入集,线程交换引起的开销过大,导致加速比小于 1。进一步优化后,开销得到大幅度减小,加速比提高,在交换范围为 256 和 512 时,性能分别最高提高了 6.4% 和 5.3%。从图 15 可看出,通过线程交换,程序的功耗得到一定程度的降低,而进一步优化后,功耗变化不大。

结束语 本文首先研究了线程交换对程序性能和功耗的影响。通过实验发现,线程交换在减少 Branch Divergence 的同时会改变程序的访存行为,产生访存开销,这会在一定程度上影响由于 Branch Divergence 减少带来的收益,甚至完全抵消其收益。功耗方面,对于一些程序来说,线程交换影响访存行为并不一定会引起访存次数的增多,因此线程交换并不一定会使访存部分的功耗增加,并且由于线程交换减少了某些功能单元的访问次数,一般来说优化后程序的功耗会降低。

其次,本文研究了交换范围对程序性能和功耗的影响。不同的交换范围意味着不同的优化程度,一般来说,交换范围越大,优化程度越高,同时带来的开销也越大,所以并不是交换范围越大越好,如何选择合适的交换范围往往需要具体问题具体分析。

最后,本文选择 Reduction 以及 Bitonic 这两个含有较多 Branch Divergence 的程序进行优化和性能测试,研究了它们在不同交换范围内性能和功耗的表现,并针对优化过程中存在的问题进行了进一步分析及优化。通过优化,Reduction 程序在性能平均损失 4% 的情况下功耗最大降低了 7%,Bitonic 程序在交换范围为 256 和 512 时性能分别最高提升了 6.4% 和 5.3%。

接下来,将通过更多程序测试本文所提优化方法的效果,并将该优化策略集成在面向 CUDA 的编译器中。

参 考 文 献

- [1] NVIDIA CUDA[EB/OL]. [2015-5-15]. <http://www.nvidia.com/cuda>
- [2] Zhang E Z, Jiang Yun-lian, Guo Zi-yu, et al. On-the-Fly Elimina-

tion of Dynamic Irregularities for GPU Computing[C]// Proceedings of the 16th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS). Newport Beach, CA, USA, ACM, 2011: 369-380

- [3] Zhang E Z, Jiang Yun-lian, Guo Zi-yu, et al. Streamlining GPU application on the fly: thread divergence elimination through runtime thread-data remapping [C]// Proceedings of the 24th International Conference on Supercomputing. Tsukuba, Ibaraki, Japan, ACM, 2010: 115-126
- [4] Han T Y D, Abdelrahman T S. Reducing Branch Divergence in GPU Programs[C]// Proceedings of the 4th Workshop on General Purpose Processing on Graphics Processing Units. Newport Beach, CA, ACM, 2011: 1-8
- [5] Qian Cheng, Shen Li, Zhao Xia, et al. Thread Swapping Based Optimization Strategy for Sort Algorithms on GPUs[J]. Journal of Northeastern University(Natural Science), 2014, 35(1): 68-73(in Chinese)
钱程, 沈立, 赵夏, 等. GPU 上基于线程交换的排序算法优化策略[J]. 东北大学学报(自然科学版), 2014, 35(1): 68-73
- [6] Bakhoda A, Yuan G L, Fung W W L, et al. Analyzing CUDA workloads using a detailed GPU simulator[C]// IEEE International Symposium on Performance Analysis of Systems and Software. Boston, MA, USA, IEEE, 2009: 163-174
- [7] GPGPU-Sim Manual[EB/OL]. [2015-5-15]. http://gpgpusim.org/manual/index.php/GPGPU-Sim_3_x_Manual
- [8] Xu Qiu-min, Annaram M. PATS: Pattern Aware Scheduling and Power Gating for GPGPUs[C]// Proceedings of the 23rd international conference on Parallel Architectures and Compilation. Edmonton, AB, Canada, ACM, 2014: 225-236
- [9] Leng Jing-wen, Tayler H, Ahmed E, et al. GPUWatch: Enabling Energy Optimization in GPGPUs[C]// Proceedings of 40th Annual International Symposium on Computer Architecture. Tel-Aviv, Israel, ACM, 2013: 487-498
- [10] GPUWatch[EB/OL]. [2015-5-15]. <http://gpgpu-sim.org/gpuwatch>

(上接第 8 页)

- [37] Javassist[EB/OL]. <http://www.csg.ci.i.u-tokyo.ac.jp/~chi-ba/javassist/html/javassist/CtClass.html>
- [38] ASM[EB/CP]. <http://asm.ow2.org/>
- [39] DynamoRIO[EB/OL]. <http://www.dynamorio.org/>
- [40] DynInst[EB/OL]. <http://www.dyninst.org/>
- [41] Pin [EB/OL]. <https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html/>
- [42] Wang T, Wei T, Zou W. Checksum-Aware Fuzzing Combined with Dynamic Taint Analysis and Symbolic Execution[J]. Acm Transactions on Information & System Security, 2011, 14(2): 613-613
- [43] Atwood J W, et al. A new fuzzing technique for software vulnerability mining[D]. Concordia University, 2009
- [44] MSDN[EB/OL]. <https://msdn.microsoft.com/zh-cn/default.aspx>
- [45] Sulley[EB/OL]. <http://resources.infosecinstitute.com/sulley-fuzzing/>
- [46] Miller C, Peterson Z N J. Analysis of Mutation and Generation-Based Fuzzing [R/OL]. <http://securityevaluators.com/files/papers/analysisfuzzing.pdf>
- [47] Ganesh V, Leek T, Rinard M. Taint-based Directed Whitebox Fuzzing[C]// Proceeding International Conference on Software Engineering. 2009: 474-484
- [48] Lanzi A, Martignoni L, Monga M, et al. A Smart Fuzzer for x86 Executables[C]// ICSE Workshops 2007, Third International Workshop on Software Engineering for Secure Systems, 2007 (SESS'07). IEEE, 2007: 7-7
- [49] Liu G H, Wu G, Tao Z, et al. Vulnerability Analysis for X86 Executables Using Genetic Algorithm and Fuzzing[C]// International Conference on Convergence & Hybrid Information Technology. IEEE, 2008: 491-497
- [50] PaiMei[CP/OL]. <http://www.openrce.org/downloads/details/208/PaiMei>
- [51] Liu W. Research on DoS Attack and Detection Programming[C]// Workshop on Intelligent Information Technology Applications IEEE. 2009: 207-210
- [52] Hydera I, Sultan A B M, Zulzilil H, et al. Current state of research on cross-site scripting(XSS)-A systematic literature review[J]. Information & Software Technology, 2015: 170-186
- [53] Dukes L, Yuan X, Akowuah F. A case study on web application security testing with tools and manual testing[C]// Southeastcon, IEEE. IEEE, 2013: 1-6
- [54] WebScara[EB/OL]. https://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
- [55] ShareFuzz[CP/OL]. <http://www.immunitysec.com/resources>