面向 OpenCL 架构的 GPGPU 量化性能模型

朱俊峰[1],陈　钢[2],张珂良[1],吴百锋[1]

[1] 复旦大学 计算机科学与技术学院,上海 200433)
[2] 中国电子科技集团公司 第三十八研究所,合肥 230088)
E-mail:10110240011@fudan.edu.cn

摘　要　为了评估数据并行(DLP)应用并行化后在GPU体系结构上的执行性能,针对OpenCL架构提出一种GPGPU量化性能模型.该模型充分考虑了影响GPGPU程序性能的各种因素:全局存储器访问、局部存储器访问、计算与访存重叠、条件分支转移和同步.通过对DLP应用的静态分析并设定具体的OpenCL执行配置,在无需编写实际GPGPU程序的前提下采用该模型即可估算出DLP应用在GPU体系结构上的执行时间.在AMD Radeon™ HD 5870 GPU和NVIDIA GeForce™ GTX 280 GPU上对矩阵乘法与并行前缀和的分析与实验结果表明:该性能模型能够相对准确地评估DLP应用并行化后的执行时间.

关 键 词　GPU;GPGPU;数据并行;OpenCL;性能模型
中图分类号　TP391　　**文献标识码**　A:　　**文 章 编 号**　1000-1220(2013)05-1118-08

Quantitative GPGPU Performance Model Targeting OpenCL Architecture

ZHU Jun-feng[1],CHEN Gang[2],ZHANG Ke-liang[1],WU Bai-feng[1]

[1] (Institute of Computer Science and Technology, Fudan University, Shanghai 200433, China)
[2] (China Electronics Technology Group Corporation No.38 Research Institute, Hefei 230088, China)

Abstract　For the sake of evaluating the potential execution performance of a data-level parallel application that will be parallelized onto GPU architecture, a quantitative GPGPU performance model targeting OpenCL architecture is proposed. The present model embodies various features of the GPU architecture which affect the performance of a GPGPU program such as global memory access, local memory access, overlapping memory access with useful computation, conditional branch divergence and synchronization. By statically analyzing a DLP application with considering of the specific OpenCL execution configuration, the present model can approximately estimate the execution time of a DLP application without the need of writing the actual GPGPU program. Analytical and experimental results for matrix multiplication and parallel prefix-sum on AMD Radeon™ HD 5870 GPU and NVIDIA GeForce™ GTX 280 GPU show that the present model can estimate the execution time of DLP applications relative accurately.

Key words　GPU;GPGPU;data-level parallel;OpenCL;performance model

1　引　言

近年来,数据并行(Data-Level Parallelism, DLP)应用受到学术界和工业界的广泛关注[1].这类应用具有数据与计算密集、可并行性高等特点,对运算能力和存储带宽要求较高,难以在多核体系结构上获得较好的性能[2].现代GPU(Graphics Processing Unit, GPU)体系结构充分利用先进的VLSI技术,在片上集成了大量的计算单元,能够提供强大的运算能力.同时,多级存储器层次为计算单元提供了较高的数据带宽,弥补了多核体系结构中的存储墙问题[3].随着AMD/ATI Stream™、NVIDIA CUDA™和OpenCL等高级编程模型的相继推出,GPU程序设计的复杂性得到大幅度降低,越来越多的DLP应用被移植到GPU上取得加速[4-8].如今,基于GPU的通用计算(General-purpose Computation on GPU, GPGPU)已经成为高性能计算领域的研究热点[9,10].

GPU硬件同时执行大量轻量级线程,通过快速线程切换以隐藏存储访问延迟,维持较高的系统吞吐率.尽管DLP应用受益于细粒度多线程的交替执行,但GPU体系结构的大规模并行特性容易产生复杂的动态行为,制约了GPGPU程序的实际性能[11].当DLP应用映射到GPU硬件上执行时,很多因素都会导致程序性能在不同程度上有所降低[12].反映GPU资源利用率的流处理器占用率并不能体现GPGPU程序性能的提升[13],即流处理器占用率的提高未必能够说明程序性能也得到了相应的提升.因此,建立一个能够全面反映GPU体系结构特性的性能模型,对于评估特定的DLP应用是否适合移植到这种大规模并行计算平台上至关重要.

为了预测DLP应用并行化后在GPU体系结构上的执行性能,本文针对OpenCL架构提出一种GPGPU量化性能分析模型.该模型建立在抽象GPU体系结构的基础上,充分考虑了影响GPGPU程序性能的各种因素,如全局存储器的接合

访问、局部存储器的冲突访问、计算与存储访问重叠、条件分支转移、同步)在无需编写实际 GPGPU 程序的前提下,通过对 DLP 应用的静态分析并结合 GPU 的性能参数设定具体的 OpenCL 执行配置,即可大致估算出该 DLP 应用并行化后在 GPU 体系结构上的执行时间.

2 相关工作

Goswami 等[14]的研究表明:即便一些应用程序具有丰富的数据并行性,也很难在 GPU 体系结构上取得较高的峰值性能.因此,为了充分发挥 GPU 体系结构的性能优势,需要不断地对 GPGPU 程序实施性能优化[15].AMD 和 NVIDIA 公司分别在各自的 GPGPU 开发环境中提供了可视化性能分析器 Stream Profiler[16]和 Parallel Nsight[17].受 GPU 硬件架构的限制,这些性能分析器只能反馈 GPU 中一个计算单元的性能统计数据.为了指导应用程序有效移植到多核体系结构,Williams 等[18]提出了面向软硬件性能分析的 Roofline 模型.尽管该模型能够判断给定应用程序是计算受限型还是存储受限型,但并没有结合 GPU 特有的体系结构特征,无法准确评估 GPGPU 程序的性能.针对计算机视觉应用,Mistry 等[19]提出了一种可用于分析 OpenCL 架构下的程序性能框架.该框架通过 OpenCL 事件对计算机视觉应用程序的执行过程进行分解,采用 OpenCL 内建的 API 函数调用实现细粒度的性能评测.然而,该框架并没有提供一个用于评估 GPGPU 程序性能的模型.

由于现代 GPU 体系结构的复杂性,BSP、PRAM 和 QRQW 等传统并行计算模型无法用于评估 GPGPU 程序的性能.Kothapalli 等[20]对上述三种模型加以组合并做适当扩展,提出了一种用于分析 CUDA 架构下 GPGPU 程序的性能模型.虽然这种模型体现了 GPU 体系结构的若干特性,如计算开销、存储器访问开销等,但并没有考虑同步和条件分支转移对 GPGPU 程序性能的影响.此外,该模型也没有考虑计算对于存储访问延迟的隐藏.Hong 等[21]认为评估存储访问开销是评估 GPGPU 程序性能的关键所在.在考虑运行线程数目和所消耗存储带宽的基础上,他们提出了一种基于并行存储访问请求的性能分析模型.该模型假设所有计算指令的执行开销相等,并且没有考虑条件分支转移和共享存储器访问冲突对于 GPGPU 程序性能的影响.为了缩小编译器在优化 GPGPU 程序时的搜索范围,Baghsorkhi 等[22]也提出了一种性能分析模型.该模型可以较为准确地识别 GPGPU 程序的性能瓶颈,但缺乏评估同步、计算与访存重叠等因素对于程序性能的影响.与 Hong 和 Baghsorkhi 等人的出发点不同,Zhang 等[23]提出了一种基于微基准测试程序的性能分析模型.该模型建立在 GPU 本地机器指令集的基础上,考虑了 GPU 体系结构的三个特性:指令流水线、共享存储器访问和全局存储器访问.虽然该模型可以用于评估 GPU 体系结构设计的改进,但是并没有考虑同步和条件分支转移.此外,完全基于微基准测试程序的评估方法无法定量评估 GPGPU 程序的执行时间.值得指出的是,上述工作[20-23]的性能评估都需要编写实际的 GPGPU 程序.

3 抽象的 GPU 体系结构与执行模型

为了统一 AMD/ATI GPU 与 NVIDIA GPU 体系结构的差异,本文采用 OpenCL 架构下的相关概念对现代 GPU 体系结构和执行模型进行抽象描述.

3.1 GPU 体系结构

GPU 是一个由若干计算单元(CU)构成的多处理器设备,每个 CU 又由若干处理元件(PE)组成,如图 1 所示.每个 CU 包含一块对内部所有 PE 都可见的局部存储器,通常用以

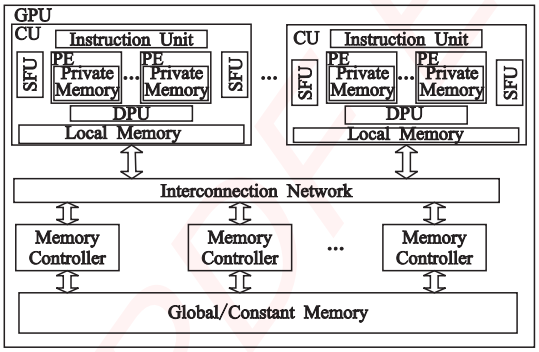


图 1 抽象的 GPU 体系结构

Fig. 1 Abstracted GPU architecture

在 PE 之间进行数据交换,可以较快的速度访问,但容量非常有限.PE 是 GPU 中最基本的执行单元,并拥有其独自的私有存储器,同一个 CU 中的 PE 之间是完全同步执行的.PE 可以执行逻辑运算、整型运算和单精度浮点运算.CU 中还包括特殊函数单元(SFU),可以执行一些超越函数 sin、cos 和 log 等.此外,每个 CU 中还包含双精度浮点运算器(DPU),用以支持 64 位的双精度浮点运算.全局存储器通过内部互连网络可被所有 CU 共享,该存储器的容量较大,但访问延迟较高.常数存储器是全局存储器中的一块区域,也可被所有 CU 共享,但该存储区域在片上被缓存,可以较快的速度访问.

3.2 GPU 执行模型

GPGPU 程序分为两部分:一部分是在主机(host)上执行的主程序,另一部分则是在 GPU 上执行的核心程序(kernel).Kernel 通过 host 提交命令,驱动 GPU 上的 PE 以 SIMD 的方式完成计算任务,并将计算结果返回给 host.工作项是运行在 PE 上的 kernel 实例,使用全局 ID 对其标识.若干工作项被组织在一个工作组内,每个工作组同样拥有唯一 ID.

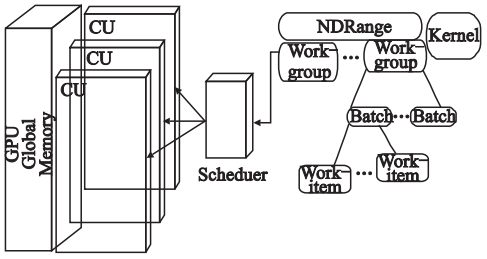


图 2 抽象的 GPU 执行模型

Fig. 2 Abstracted GPU execution model

如图 2 所示,工作组中的所有工作项被分配到一个 CU 中的若干 PE 上并发执行.为了充分利用 GPU 硬件资源,同一个工作组内若干连续的工作项被组织成一个批(batch)作为调度的基本单元.在 AMD/ATI GPU 中 batch 被称为 wavefront,大

小为 64 在 NVIDIA GPU 中 batch 被称为 warp，大小为 32 ），batch 内的工作项在同一指令地址开始执行并且执行相同的指令. Batch 被动态分配到 CU 中的 PE 上以时间片轮转方式运行，因此当一些 batch 因存储访问而阻塞等待时，其他 batch 可以立刻被调度到 CU 上运行，从而实现存储访问的延迟隐藏.

4 量化 GPGPU 性能评估模型

4.1 GPGPU 程序性能因素

对 GPGPU 程序的性能评估不但需要考虑 kernel 的执行时间，还需要考虑 kernel 的启动开销和数据传输开销. 通常情况下，kernel 启动的平均开销为几个微秒. 在数据规模较大时，该部分的开销相对于数据传输时间和 kernel 执行时间可以忽略不计[24].

在 OpenCL 架构下，工作项是 kernel 执行的基本元素. 在 kernel 的执行过程中，工作项主要完成数据 存储器 读取、数据计算和数据 存储器 写回等操作 首先，工作项对于全局存储器的访问以及计算与访存的重叠是影响 GPGPU 程序性能的重要因素 其次，局部存储器的模块访问冲突以及相关的同步开销也是影响 GPGPU 程序性能的重要因素 最后，条件分支转移使得分支代码段被各个工作项串行执行，降低了 GPGPU 程序的性能.

4.2 计算指令开销

本文采用实验的方法统计各类指令的平均执行开销. 实验测定一些常用指令在 AMD/ATI Radeon™ HD 5870 GPU 和 NVIDIA GeForce™ GTX 280 GPU 上的执行开销. 实验过程仅设定一个工作项，该工作项反复执行某种指令 10 次，以测量的平均值作为该类指令的执行开销. 表 1 按照 CU 中能够执行某种特定指令的运算单元的种类对 NVIDIA GeForce™ GTX 280 GPU 执行的常用指令及其执行开销 单位为时钟周期 进行了分类. 可以看出，每种类型指令的平均执行开销与 CU 中能够执行该指令的计算单元数目有关.

表 1 NVIDIA GeForce™ GTX 280 GPU 上的指令执行开销

Table 1 Instruction execution cost on Geforce™ GTX 280 GPU

指令类型	指令示例	执行单元数目	平均执行开销
1	ADD，SUB	8	24
2	MUL/DIV	10/NA	357
3	SIN，COS，LOG	2	45
4	DP-ADD，DP-DIV	1	378

每个 CU 在同一时刻只能执行一个 batch，多个 batch 在 CU 上以时间片轮转的方式执行. 若一个 CU 中包含 $N_{PE/CU}$ 个 PE，每个 batch 由 $BATCH\_SIZE$ 个工作项构成，那么为 batch 中所有工作项执行一条开销为 $cost_{ins}$ 的指令 ins，则需要连续的 $(BATCH\_SIZE/N_{PE/CU}) \times cost_{ins}$ 个时钟周期. 以 NVIDIA GeForce™ GTX 280 GPU 为例，一个 CU 包含 8 个 PE，每个 batch 中包含 32 个工作项，因此为一个 batch 中所有工作项执行一条加法指令则需要 $96\ (32/8 \times 24)$ 个连续的时钟周期. 可见，一个 batch 中所有工作项执行的各类计算指令的开销 分支语句中的计算指令除外，将在 4.5 节中单独评估 和时间可以表示为 ：

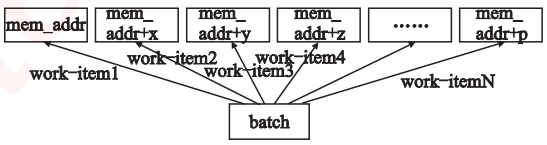
$$cost_{instruction/batch} = \frac{BATCH\_SIZE}{N_{PE/CU}} \times \sum_{i=1}^{4} cost\_ins_i \times num\_ins_i \quad (1)$$

$$time_{instruction/batch} = \frac{cost_{instruction/batch}}{freq_{shader}} \quad (2)$$

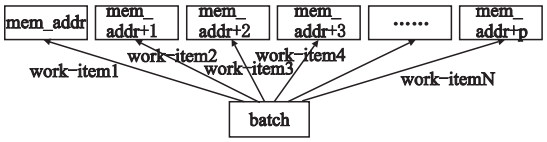
其中 $cost\_ins_i$ 为第 i 类计算指令的执行开销，num_ins_i 为第 i 类计算指令的数目，$freq_{shader}$ 为 GPU 核心时钟频率.

4.3 全局存储器访问开销

通常情况下，现代 GPU 全局存储器系统以连续的 128 个字节的对齐数据段作为一次传输单位. 如果执行存储访问的 batch 内工作项的存储访问模式满足一定条件，即所访问的数据地址在上述数据段中连续排列 工作项之间具有数据访问的空间局部性 那么这些工作项的多个存储访问请求可以在一次集中的存储器传输中完成 接合的存储访问 .)否则，每个分散的存储器访问都将分别产生一次不同的存储器传输 非接合的存储访问 .如图 3 a(所示，当工作项 1 访问 mem_addr 时，工作项 2 在访问 $mem\_addr + x$，工作项 3 在访问 $mem\_addr + y$，等等. 由于各工作项所访问的全局存储器地址不满足工作项之间的空间局部性，因此在这种非接合的访问方式下需要多次全局存储器传输才能满足所有工作项的存储访问请求. 在图 3 b(中，当工作项 1 访问 mem_addr 时，工作项 2 在访问 $mem\_addr + 1$，工作项 3 在访问 $mem\_addr + 2$，等等. 如此，工作项之间具有较好的数据访问局部性，从而有效提升了全局存储器的带宽利用率.



a( ) 非接合的全局存储器访问



b( ) 接合的全局存储器访问

图 3 非接合与接合的全局存储器访问

Fig. 3 Coalesced and non-coalesced global memory accesses

如果 batch 执行了非接合的存储访问，在所有工作项的存储器传输请求完成之前，该 batch 都不能被调度到 CU 上运行. 相比接合的全局存储器访问，非接合的访问需要更长的存储访问等待周期. 可见，在评估 GPGPU 程序的全局存储器访问开销时，需要考虑非接合存储访问对于性能的影响.

若一个 batch 中包含 $BATCH\_SIZE$ 个工作项，其中有 c 个工作项的存储访问请求可以在一次全局存储器传输中得到满足，则该 batch 总存储访问的平均延迟 单位为时钟周期 和时间为 ：

$$cost_{global\_memory/batch} = (BATCH\_SIZE - c) \times GM\_L + \frac{GM\_L + c}{c} \quad (3)$$

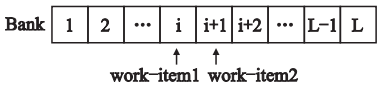
$$time_{global\_memory/batch} = \frac{cost_{global\_memory/batch}}{freq_{global}} \quad (4)$$

其中 GM_L 为一次全局存储器传输所需要的时钟周期数，$freq_{global}$ 为全局存储器时钟频率. 值得注意的是，一次全局存储器传输所能够满足的工作项存储访问请求数目 c 与访问间隔 $stride$ 有关.

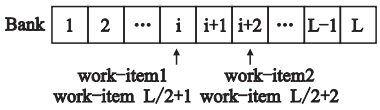
4.4 局部存储器访问开销

局部存储器是一块可被同一工作组内的所有工作项读写的快速存储器，可以利用该存储区域进行工作项之间的数据交换，也可以减少对全局存储器的访问. 局部存储器被划分成一些个大小相同、能够并行访问的存储器模块. 通常情况下，大小为 LKB 的局部存储器被组织成 L 个存储器模块，每个模块为 1KB. 相邻 32 位字被存放在相邻的模块中，每个模块在一个时钟周期能够提供 4 字节的访问带宽.

在同一时钟周期内，每个存储器模块都能够独立工作. 如果 batch 中有多个工作项同时访问处于一个存储器模块内的不同地址时，访问操作不能被并行执行，此时会出现存储器模块间冲突. 若 k 个工作项同时访问一个模块，会出现 k-路模块冲突访问. 硬件会把造成 k-路模块冲突的访问请求划分为 k 次无冲突的独立请求，此时有效带宽相应地降低 k 倍. 可见，局部存储器的带宽与模块冲突数的路数成反比.



a( ) 无冲突访问



b( ) 2-路访问冲突

图 4 局部存储器模块访问冲突

Fig. 4 Local memory access and memory bank conflict

如图 4 a(所示，当 batch 中的各工作项以间隔为 1 的方式访问局部存储器时，相邻的工作项访问相邻的存储器模块，因此没有模块访问冲突. 如图 4 b(所示，当工作项以间隔为 2 的方式访问时，即第一个工作项访问第 i 个存储器模块，第二个工作项访问第 i(+2 个存储器模块，第 L(/2 +1 个工作项访问访问第 i 个存储器模块，第 L(/2 +2 个工作项访问第 i(+2 个存储器模块，此时存在 2-路模块冲突. 可见，当访问间隔为奇数时不存在模块冲突 当访问间隔为偶数时存在模块冲突. 不难看出，冲突路数为访问间隔与模块数的最大公约数.

值得注意的是，对于一个无冲突的局部存储器访问，每个 batch 都需要两次数据传输. 因此，一个 batch 内的工作项访问局部存储器的开销和时间为 ：

$$cost_{local\_memory/batch} = 2 \times LM\_L \times GCD\ (stride, L\ ) \qquad 5\ (\ )$$

$$time_{local\_memory/batch} = \frac{cost_{local\_memory/batch}}{freq_{shader}} \qquad 6(\ )$$

其中 LM_L 为一次局部存储器传输所需要的时钟周期数，GCD 为最大公约数函数，$stride$ 为工作项之间的局部存储器访问间隔.

4.5 条件分支转移开销

Batch 内的工作项可以自由发生分支转移，一旦某个工作项发生了分支转移 此处称为分叉点 那么不同的分支路径将被 batch 内的所有工作项串行执行，直到所有工作项回到同一路径 此处称为汇聚点 时才能继续并行执行. 同时，batch 内没有发生分支转移的工作项执行该条分支路径的结果将被丢弃. 值得注意的是，循环结构产生的分支 例如循环控制指令 属于代码的固有属性，并不会造成 batch 内的工作项发生转移.

在图 5 所示的 if-else 分支中，只有当 if 条件在所有工作项都满足或都不满足时，才会造成仅有一个分支路径被执行 ；相反，如果 if 条件在某些工作项中满足而在某些工作项中不满足时，那么 if 和 else 两个分支将被串行执行，从而使得处于分支中的代码段的执行效率相应下降. 在图 5 中，o 代表活动的工作项 执行有效计算)k 代表非活动的工作项 执行无效计算 .)

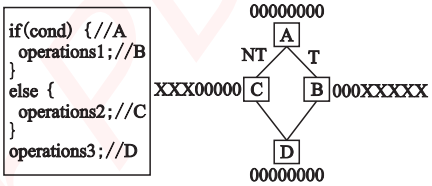


图 5 条件分支转移示例

Fig. 5 Conditional branch divergence

为了评估条件分支转移开销，本文把分叉点到聚合点之间的所有指令定义为顺序执行区域，其大小为该区域内的总的指令数目. 在图 5 中，A 为分叉点，D 为汇聚点，基本块 B 和基本块 C 构成顺序执行区域，大小为 operations1 与 operations2 之和. 若一个条件分支 branch 由 b 个路径组成 p(ath$_1$，…，path$_b$)发生分支转移的 batch 的百分比为 D_{batch}，则该 CU 上运行的 batch 的平均分支开销可以表示为 ：

$$cost_{branch/batch} = D_{batch} \times cost_{serial\_section} + 1(-D_{batch}\ ) * \frac{cost_{path_1} + cost_{path_2} + cost_{path_b}}{b} \qquad 7(\ )$$

$$time_{branch/batch} = \frac{cost_{branch/batch}}{freq_{shader}} \qquad 8(\ )$$

其中，$cost_{serial\_section}$ 为顺序执行区域的执行开销，$cost_{path}$ 为各分支路径的执行开销. 值得注意的是，式 7(中的每一项 $cost$ 都可以用式 1(计算求得.

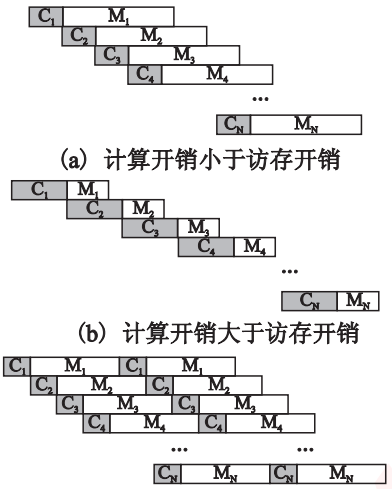
Fung 等[23] 通过研究大量基准测试程序发现，不同基准测试程序中 batch 发生条件分支转移的概率为 13% ~28%. 因此，本文设定 D_{batch} 取值为 1(3% +28% /)2 =20%. 虽然这种评估方法会产生误差，但是相比简单地假设所有的 batch 都执行顺序执行区域 通常会高估分支代码段的执行时间)要更为精确. 此外，Kerr 等[24] 对大量数据并行负载采用 PTX 指令集的模拟结果表明 在一个发生条件分支转移的 batch 中，活动线程与非活动线程的平均比值 活动因子 为 20%，从一定程度上反映了 D_{batch} 取值为 20% 的合理性.

4.6 计算与访存重叠的影响

在 4.2 节 ~4.5 节中，模型仅考虑了单个 batch 的计算指令和存储器访问开销. 在多线程调度环境下，较为精确地评估 GPGPU 程序的性能还需要考虑 GPU 体系结构对多个 batch

之间的调度情况（计算与访存重叠）.由于各个厂商对体系结构知识产权的保密，GPU 体系结构内部调度细节未在厂商发布的文档中具体说明. 因此，本文分析理想情况下的调度对于存储访问延迟影响的效果.

GPU 体系结构以 batch 为单位进行调度，因此 GPGPU 程序的性能主要由 batch 的计算开销和访存开销决定. 由于 batch 内的条件分支转移本质上可以归结为计算开销，因此可以将发生条件分支转移的附加开销与计算指令开销一起作为该 batch 的总体计算开销. Batch 的访存开销则由全局存储器访问开销和局部存储器访问开销构成. GPU 体系结构的设计原则是快速切换大量独立的 batch，尽可能实现 batch 存储访问延迟的完美隐藏. 此外，如果 GPGPU 程序的算术密度（计算操作与访存操作的比值）较高且运行时系统能够生成足够多的工作项，那么存储访问延迟几乎能够被有效的计算完全隐藏[27]. 在这种情况下，CU 上多个 batch 的执行如图 6 所示.



(a) 计算开销小于访存开销

(b) 计算开销大于访存开销

(c) 多次计算与访存

图 6 多线程调度对于性能的影响

Fig. 6 Performance impact on multi-threaded scheduling

不难看出，无论一个 batch 的计算开销小于访存开销还是计算开销大于访存开销，考虑计算和访存理想重叠因素的执行时间都可以表示为：

$$time_{batch/CU} = N_{batch/CU} \times time_{computation/batch} + time_{memory/batch} \quad (9)$$

$$time_{computation/batch} = time_{instruction/batch} + time_{branch/batch} \quad (10)$$

$$time_{memory/batch} = time_{global\_memory/batch} + time_{local\_memory/batch} \quad (11)$$

其中 $N_{batch/CU}$ 代表每个 CU 上的 batch 数，$cost_{computation}$ 代表一个 batch 的计算开销（由指令开销和条件分支转移开销构成），$cost_{memory}$ 代表一个 batch 的访存开销（由全局存储器访问开销和局部存储器访问开销构成）.

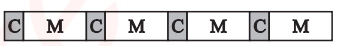


图 7 计算与访存串行执行

Fig. 7 Sequential execution and memory access

需要注意的是，控制私有存储器和局部存储器的使用数量，对于存储访问延迟的隐藏效果有着一定程度的影响. 在极端的情况下，当 CU 上没有足够的活动 batch 时（例如只有一个 batch）计算和访存将会串行执行，如图 7 所示. 因此，在没有足够多的 batch 时，本文所假定的存储访问开销能够完全被计算操作隐藏，将会低估 GPGPU 程序的执行时间.

4.7 同步开销

通常情况下，同步原语的硬件开销为几十个时钟周期[28]. 对于工作项加载数据至局部存储器时的同步，可以简单地对每条同步原语采取增加几十个时钟周期开销的评估方法. 然而在 GPU 体系结构的多线程环境下，如果对于其他类型的同步操作（例如在同步操作之前执行若干计算操作）都采用这种评估方法则会低估同步开销时间. 为此，本文采用图 8 所示的同步开销评估策略：若一个工作组中有 $N_{batch/work\text{-}group}$ 个 batch，当最后一个 batch 到达同步点时，所有的 batch 才可以继续执行，因此同步开销和时间可以表示为：

$$cost_{sync/work\text{-}group} = N_{batch/work\text{-}group} \times cost_{computation} + cost_{memory} - (cost_{computation} + cost_{memory}) + (N_{batch/work\text{-}group} - 1) \times cost_{computation} \quad (12)$$

$$time_{syne/work\text{-}group} = \frac{cost_{sync/work\text{-}group}}{freq_{shader}} \quad (13)$$

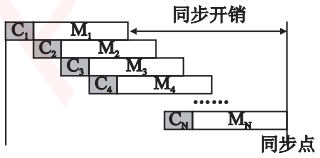


图 8 工作组内 batch 的同步开销

Fig. 8 Batch synchronize cost in a work-group

4.8 总体性能模型

GPGPU 程序的执行时间与应用程序的数据规模有着直接的联系，待处理数据的规模决定了 GPU 上运行的工作项数目. 一般情况下，GPU 上执行的工作项数目等于数据规模的大小（可由用户在 kernel 中使用显式编程模型指定）.假设一个 GPGPU 程序共有 $N_{work\text{-}item}$ 个工作项，这些工作项被组织成 $N_{work\text{-}group}$ 个工作组，每个工作组包含 $N_{batch/work\text{-}group}$ 个 batch，每个 batch 又由 $BATCH\_SIZE$ 个工作项组成，工作项在 CU 中的 $N_{PE/CU}$ 个 PE 上并行执行，则该 GPGPU 程序的执行时间为：

$$time_{GPGPU\_kernel} = \frac{N_{work\text{-}group} \times N_{batch/work\text{-}group} \times time_{batch/CU}}{N_{CU}} + \frac{time_{syne/work\text{-}group} \times N_{work\text{-}group}}{N_{CU}} = \frac{N_{work\text{-}group} \times (N_{batch/work\text{-}group} \times time_{batch/CU} + time_{syne/work\text{-}group})}{N_{CU}} \quad (14)$$

$$N_{batch/work\text{-}group} = \frac{N_{work\text{-}item}}{N_{work\text{-}group} \times BATCH\_SIZE} \quad (15)$$

$$N_{batch/CU} = \frac{N_{work\text{-}group} \times N_{batch/work\text{-}group}}{N_{CU}} \quad (16)$$

值得注意的是，batch 内的工作项在 CU 中的 $N_{PE/CU}$ 个 PE 上并行执行这一特性已经体现在式（9）的指令开销评估中. 因此，式（14）中的分母仅包含 N_{CU} 这一项即可.

5 实验及结果分析

为了验证本文 GPGPU 量化性能模型的准确性，选取了矩阵乘法（matrix multiplication，MM）和并行前缀和算法（parallel prefix-sum，PPS）作为应用实例进行评估. 实验所采用的 GPU 为 AMD/ATI Radeon™ HD 5870 GPU 和 NVIDIA Ge-

Force™ GTX 280 GPU 所采用的 CPU 为 Intel Pentium E( ) Dual-Core E6300 主频为 2.8GHz)系统内存为 2G 所采用的操作系统是 Windows XP SP3,开发环境是 Microsoft Visual Studio . Net 2008,AMD Stream SDK 2.1 版本,NVIDIA OpenCL SDK 3.0 版本.

为了侧重对 GPU 体系结构不同特性的评估 例如全局存储器的非接合访问、计算与访存重叠、局部存储器的访问冲突、同步、条件分支转移等 本文采用 OpenCL 编程语言实现了 MM 和 PPS 的两个 kernel 版本. Kernel 实际运行时间为实测 10 次的平均值.

5.1 矩阵乘法

本文实现了两个 GPU 版本的矩阵乘法核心 第一个版本 M(MM_Global 体现了工作项对于全局存储器非接合访问和计算与访存重叠的特性,第二个版本 MM_Local 则体现了工作项对于局部存储器的无冲突访问和相关的同步特性. 矩阵乘法 $A \times B = C$ 主个矩阵的矩阵规模都为 1024×1024 由一个三重循环迭代构成,将最外层的两个循环迭代实例映射为一个工作项.

在 MM_Global 版本中,每个工作项从全局存储器中读取矩阵 A 的一行和矩阵 B 的一列,通过最内层循环计算矩阵 C 的一个元素. 因此,为了计算矩阵 C 的一个元素,每个工作项需要从全局存储器中读取矩阵 A 的 1024 个数据元素和矩阵 B 的 1024 个数据元素,并完成 1024 次加法运算和 1024 次乘法运算. 可见在矩阵乘法程序中,一个 batch 的计算开销为 $BATCH\_SIZE \times 1024 \times (cost_{ADD} + cost_{MUL})$. 需要注意的是,工作项对矩阵 A 的访问为接合访问 按行方向访问 而对矩阵 B 的访问为非接合访问 按列方向访问 .)在 4.3 节中提到,现代 GPU 体系结构对于 batch 中工作项的接合全局存储器访问可以在一次传输内完成. 然而由于各大厂商的不同 GPU 产品存在差异,因此实际的实现可能存在一些差别. 例如在 NVIDIA GPU 中,如果一个 batch 中工作项的存储访问模式是接合的,对于不同型号的 GPU 可能需要不同的传输次数. 同样,对于非接合的全局存储器访问也存在差异. 基于这种差异,实验采取如下策略 如果 batch 内的工作项以接合的方式访问全局存储器,则需要一次传输 否则采用式 3(评估传输次数. 由于最内层循环一共迭代 1024 次,因此一个 batch 内所有工作项读取矩阵 A 的一行数据元素需要 1024 次数据传输,而读取矩阵 B 的一列数据元素则需要 $1024 \times BATCH\_SIZE$ 次数据传输,写回矩阵 C 的一个数据元素需要 1024 次数据传输. 因此,一个 batch 的全局存储器访问开销为 $1024 \times (2 + BATCH\_SIZE)$. 实验采用每个工作组包含 64、128 和 256 个工作项的执行配置,工作组的数目分别为 $1024^2/64$、$1024^2/128$ 和 $1024^2/256$,每个工作组中相应的 batch 数目分别为 $1024^2/(64 \times BATCH\_SIZE)$、$1024^2/(128 \times BATCH\_SIZE)$ 和 $1024^2/(256 \times BATCH\_SIZE)$. 图 9 展示了不同工作组执行配置下的 MM_Global kernel 在采用本文性能模型评估的执行时间和在两款 GPU 上实测的执行时间.

为了克服 MM_Global 版本中矩阵 B 的非接合全局存储器访问且反复从全局存储器中加载数据元素的缺陷,MM_Local 版本利用局部存储器实现了工作项之间的数据共享. 为此,本文对矩阵 A 和矩阵 B 主个矩阵的矩阵规模都为 1024×1024 采用了分块 $8(\times 8, 8 \times 16, 16 \times 16$ 的方式实现了 MM_Local kernel 每个工作组计算矩阵 C 的一个子矩阵,并且分块内的每个工作项从局部存储器中加载数据元素并计算结果子矩阵 即矩阵 C 的子矩阵 的一个元素,结果子矩阵为两个

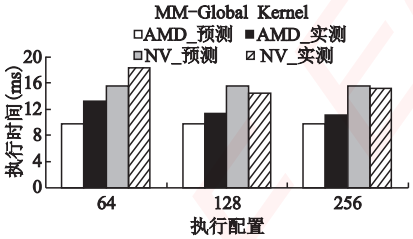


图 9 MM_Global kernel 的预测时间和实测时间对比

Fig. 9 Comparison of predicted time and actual time for MM_Global kernel

分块矩阵乘积的累加和. 以 8×16 的分块方式为例,分块大小为工作组内的工作项数目,即每个工作组内包含 128 个工作项. 因此工作组的数目为 $1024^2/128$,每个工作组中相应的 batch 数目为 $1024^2/(128 \times BATCH\_SIZE)$. 对于 batch 内的每个工作项而言,MM_Local kernel 的外层循环需要进行 1024/8 次迭代,完成数据从全局存储器至局部存储器的加载 内层循环则需要迭代 16 次,完成两个分块矩阵的乘加运算. 因此,一个 batch 的计算开销为 $BATCH\_SIZE \times 1024/8 \times 16 \times (cost_{ADD} + cost_{MUL})$. 值得注意的是,MM_Local kernel 的计算开销随执行配置和分块方式的不同而产生变化,因为每个工作项对应的内层循环迭代的计算量和分块大小有关. 存储访问开销为 3×1024 个接合的全局存储器访问加上 2×1024 个局部存储器访问. 需要指出的是,工作项对于局部存储器访问不存在冲突,因为在进行两个子矩阵相乘的内层循环中,对矩阵 A 和矩阵 B 的数据元素的访问都分布在不同的存储器模块中. 此外,访问局部存储器时工作项之间需要实施同步操作. 在 MM_Local kernel 版本中,一共有两次同步 第一次为加载数据至局部存储器的同步 保证整个分块中的数据都被加载至局部存储器)第二次为每个工作项计算子矩阵一个元素后的同步 保证计算下一个子矩阵元素同时开始 .)由于第一次同步发生之前不存在任何计算操作,因此在评估时简单的采用增加 50 个时钟周期的方法 对于第二次同步则采用式 1(中的方法. 可见,相比 MM_Global 版本,MM_Local 版本采用局部存储器消除了全局存储器的非接合访问,因而提升了 kernel 的执行效率. 图 10 见下页 展示了 1024×1024 矩阵规模下不同分块方式下的 MM_Local kernel 在采用本文性能模型评估的执行时间和在两款 GPU 上实测的执行时间.

5.2 并行前缀和

本文实现了两个 GPU 版本的并行 PPS 核心 数组大小为 65536)第一个版本 P(PS_Br 体现了 batch 的条件分支转移,第二个版本 P(PS_Conf 在则体现了工作项访问局部存储器时的模块冲突.

在 PPS_Br 版本中,工作组大小分别设定为 64、128 和 256,每个工作项负责一个数组元素的前缀和计算. 以工作组大小为 256 的执行配置为例,工作组数目为 65536/256,每个工作组中相应的 batch 数目为 $65536/(256 \times BATCH\_SIZE)$.

PPS_Br kernel 内每次求前缀和的操作偏移量 *ofs* 从 1 开始，步长为 2* *ofs*，因此对 65536 个数组元素求前缀和需要进行 $\log_2^{65536}$ 次循环迭代. 对于 batch 内的每个工作项而言，每次循环中所有工作项都需要完成一次局部存储器的拷贝，并且工作项还需要根据其 ID 是否大于 *ofs* 以决定是否进一步参与加和操作. 因此在 PPS_Br kernel 中，若 batch 内的工作项不满足条件分支，则仅仅完成两次局部存储器的访问（数据拷贝）和一次全局存储器的访问（结果写回）；若 batch 内的工作项满足条件分支，则还需要完成 $\log_2^{65536}-1$ 次加法操作. 对于 batch 内的条件分支转移，采用式（7）中的方法，发生分支转移的 batch 数目为 20% ×65536/*BATCH_SIZE*. 值得注意的是，PPS_Br kernel 中的顺序执行区域仅有一条加法语句，将此附加开销分摊到各个 batch 的计算开销中. 此外，PPS_Br kernel 中还有两处同步操作：第一次同步操作保证所有工作项都完成了局部存储器拷贝，第二次同步操作保证工作组中的各工作项都完成了 $\log_2^{65536}$ 次循环迭代. 对于 PPS_Br kernel 的同步开销评估，

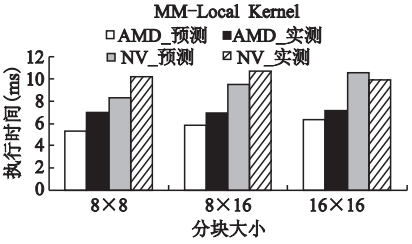


图 10 MM_Local kernel 的预测时间和实测时间对比

Fig. 10 Comparison of predicted time and actual time for MM_Local kernel

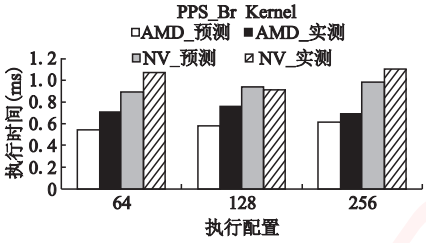


图 11 PPS_Br kernel 的预测时间和实测时间对比

Fig. 11 Comparison of predicted time and actual time for PPS_Br kernel

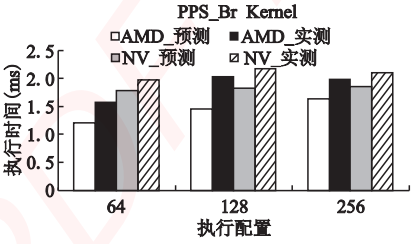


图 12 PPS_Conf kernel 的预测时间和实测时间对比

Fig. 12 Comparison of predicted time and actual time for PPS_Conf kernel

方法与 5.1 节中的 MM_Global kernel 类似. 图 11 展示了不同工作组执行配置下的 PPS_Br kernel 在采用本文性能模型评估的执行时间和在两款 GPU 上实测的执行时间.

在 4.6 节中提到，控制工作组所使用的局部存储器资源有利于系统生成更多的工作项，更好地实现计算与访存的延迟隐藏. 在 PPS_Br kernel 中，为了避免 batch 中工作项访问的共享数据被覆盖，每个工作项需要使用 8 字节空间的局部存储器来存储每一次循环迭代的输入和输出，并将上一次循环迭代的输出作为下一次的输入. 在 PPS_Conf 版本中，每个工作项负责两个数组元素的前缀和计算，只需要 4 字节空间的局部存储器. PPS_Conf kernel 的计算过程分为两步：第一步需要进行 $\log_2^{65536}$ 次循环迭代，每次循环迭代完成一次加和运算；第二步也需要进行 $\log_2^{65536}$ 次循环迭代，每次循环迭代完成一次局部存储器拷贝与一次加和运算. 然而在 PPS_Conf kernel 中，树形计算过程的循环迭代使得随着偏移量 *ofs* 的成倍增大，访问同一个局部存储器模块的工作项数目也随之成倍增大，造成越来越严重的访问冲突. 因此，对于每一次循环迭代中相应的 *ofs* 大小，采用式（5）的方法评估局部存储器访问冲突开销：当为 *ofs* 为 1 时 *stride* 为 2，造成 2-路局部存储器模块冲突；当为 *ofs* 为 2 时 *stride* 为 4，造成 4-路局部存储器模块冲突；以此类推. 值得指出的是，随着计算过程不断接近树根，即活动工作项数目不断减少，局部存储器模块冲突问题相应得到缓解. PPS_Conf kernel 中共有 3 次同步操作：前两次采用增加 50 个时钟周期的方法，第三次则采用式（12）中的方法：与 PPS_Br kernel 类似. 图 12 展示了不同工作组执行配置下的 PPS_Conf kernel 在采用本文性能模型评估的执行时间和在两款 GPU 上实测的执行时间.

6 结 论

为评估 DLP 应用并行化后在 GPU 上的执行性能，本文在抽象的 GPU 体系结构基础上提出一种 OpenCL 架构下的 GPGPU 量化性能分析模型. 该模型充分考虑了影响 GPGPU 程序性能的各种因素，包括全局存储器的非接合访问、局部存储器的冲突访问、计算与访存重叠、条件分支转移和同步. 采用该模型对 DLP 应用进行静态分析并结合 GPU 的性能参数设定具体的 OpenCL 执行配置，在无需编写 GPGPU 程序的前提下即可估算出该 DLP 应用并行化后在 GPU 上的执行时间. 在 AMD Radeon™ HD 5870 GPU 和 NVIDIA GeForce™ GTX 280 GPU 上对不同 GPU 版本的矩阵乘法与并行前缀和的评估结果表明，该模型能够相对准确地评估 DLP 应用移植到 GPU 体系结构之后的执行时间.

由于 GPU 体系结构和执行模型的复杂性，提出一种十分精确的量化性能模型较为困难. 为此，本文的下一步工作主要集中在以下两个方面：1）batch 的存储访问延迟不能完全被有效计算隐藏的情形下的性能评估；2）进一步改进条件对条件分支转移的评估.

References：

[1] Chas B. Data-parallel computing [J]. ACM Queue, 2008, 6(2): 30-39.

[2] Garland M, Kirk D B. Understanding throughput-oriented architectures [J]. Communications of the ACM, 2010, 53(11): 58-66.

[3] Feinbube F, Troger P, Polze A. Joint forces: from multithreaded programming to GPU computing [J]. IEEE Software, 2010, 28(1): 51-57.

[4] Feng Z, Zhao X Q, Zeng Z Y. Robust parallel preconditioned power grid simulation on GPU with adaptive runtime performance modeling and optimization [J]. IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems, 2011, 30(4): 562-573.

[5] Solomon S, Thulasiram R K, Thulasiraman P. Option pricing on the GPU [C]. Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications, Washington D C: IEEE Computer Society Press, 2010: 289-296.

[6] Herrero-Lopez S, Williams J R, Sanchez A. Parallel multiclass classification using SVMs on GPUs [C]. Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, New York: ACM Press, 2010: 2-11.

[7] Chen Gang, Wei Gang, Li Guo-bo, et al. Mega-base biological sequence alignment targeting OpenCL architecture [J]. Journal of Chinese Computer Systems, 2012, 33(2): 392-398.

[8] Park I K, Singhal N, Lee M H, et al. Design and performance evaluation of image processing algorithms on GPUs [J]. IEEE Transactions on Parallel and Distributed Systems, 2011, 22(1): 91-104.

[9] Nickolls J, Dally W J. The GPU computing era [J]. IEEE Micro, 2010, 30(2): 56-69.

[10] Wu En-hua, Liu You-quan. General purpose computation on GPU [J]. Journal of Computer Aided Design & Computer Graphics, 2004, 16(5): 601-612.

[11] Malony A D, Biersdorff S, Spear W, et al. An experimental approach to performance measurement of heterogeneous parallel applications using CUDA [C]. Proceedings of the 24th ACM International Conference on Supercomputing, New York: ACM Press, 2010: 127-136.

[12] Gharaibeh A, Ripeanu M. Size matters: space/time tradeoffs to improve GPGPU applications performance [C]. Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, Washington D C: IEEE Computer Society Press, 2010: 1-12.

[13] Lee J, Lakshminarayana N B, Kim H. Many-thread aware prefetching mechanisms for GPGPU applications [C]. Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture, Washington D C: IEEE Computer Society Press, 2010: 213-224.

[14] Goswami N, Ramkumar S, Madhura J, et al. Exploring GPGPU workloads: characterization methodology, analysis and microarchitecture evaluation implications [C]. Proceedings of the 2010 IEEE International Symposium on Workload Characterization, Washington D C: IEEE Computer Society Press, 2010: 1-10.

[15] Chen Gang, Li Guo-bo, Wu Bai-feng. GPU memory optimization through program restructuring methods [J]. Journal of Chinese Computer Systems, 2011, 32(10): 1921-1927.

[16] Purnomo B, Rubin N, Houston M. ATI stream profiler: a tool to optimize an OpenCL kernel on ATI Radeon GPUs [C]. Proceedings of the 2010 ACM SIGGRAPH Posters, New York: ACM Press, 2010: 1-1.

[17] NVIDIA Corporation. NVIDIA parallel nsight [OL]. http://www.nvidia.com/object/parallel-nsight.html, 2010.

[18] Williams S, Waterman A, Patterson D. Roofline: an insightful visual performance model for multicore architectures [J]. Communications of the ACM, 2009, 52(4): 65-76.

[19] Mistry P, Gregg C, Rubin N. Analyzing program flow within a many-kernel OpenCL application [C]. In Proceedings of the 4th ACM Workshop on General Purpose Processing on Graphics Processing Units, 2011: 1-8.

[20] Kothapalli K, Mukherjee R, Rehman M S, et al. A performance prediction model for the CUDA GPGPU platform [C]. Proceedings of the 2009 International Conference on High Performance Computing, Washington D C: IEEE Computer Society Press, 2009: 463-472.

[21] Hong S, Kim H. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness [C]. Proceedings of the 36th International Symposium on Computer Architecture, New York: ACM Press, 2009: 152-163.

[22] Baghsorkhi S S, Delahaye M, Patel S J, et al. An adaptive performance modeling tool for GPU architectures [C]. Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, New York: ACM Press, 2010: 105-114.

[23] Zhang Y, Owens J D. A quantitative performance analysis model for GPU architectures [C]. Proceedings of the 17th IEEE International Symposium on High-Performance Computer Architecture, Washington D C: IEEE Computer Society Press, 2011: 382-393.

[24] Volkov V, Demmel J W. Benchmarking GPUs to tune dense linear algebra [C]. Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, Washington D C: IEEE Computer Society Press, 2008: 1-11.

[25] Fung W W L, Aamodt T M. Thread block compaction for efficient SIMT control flow [C]. Proceedings of the 17th IEEE International Symposium on High-Performance Computer Architecture, Washington D C: IEEE Computer Society Press, 2011: 25-36.

[26] Kerr A, Diamos G, Yalamanchili S. A characterization and analysis of PTX kernels [C]. Proceedings of the 2009 IEEE International Symposium on Workload Characterization, Washington D C: IEEE Computer Society Press, 2009: 3-12.

[27] Cohen J, Garland M. Novel architectures: solving computational problems with GPU computing [J]. Computing in Science and Engineering, 2009, 11(5): 58-63.

[28] Wong H, Papadopoulou M M, Sadooghi-Alvandi M, et al. Demystifying GPU microarchitecture through microbenchmarking [C]. Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software, Washington D C: IEEE Computer Society Press, 2010: 235-246.

附中文参考文献：

[7] 陈　钢,韦　刚,李国波,等. 面向 OpenCL 架构的大规模生物序列比对 [J]. 小型微型计算机系统, 2012, 33(2): 392-398.

[10] 吴恩华,柳有权. 基于图形处理器(GPU)的通用计算 [J]. 计算机辅助设计与图形学学报, 2004, 16(5): 601-612.

[15] 陈　钢,李国波,吴百锋. 面向 GPU 存储优化的程序重构 [J]. 小型微型计算机系统, 2011, 32(10): 1921-1927.

嵌入式资源免费下载

总线协议：

1. 基于 PCIe 驱动程序的数据传输卡 DMA 传输
2. 基于 PCIe 总线协议的设备驱动开发
3. CANopen 协议介绍
4. 基于 PXI 总线 RS422 数据通信卡 WDM 驱动程序设计
5. FPGA 实现 PCIe 总线 DMA 设计
6. PCI Express 协议实现与验证
7. VPX 总线技术及其实现
8. 基于 Xilinx FPGA 的 PCIE 接口实现
9. 基于 PCI 总线的 GPS 授时卡设计
10. 基于 CPCI 标准的 6U 信号处理平台的设计
11. USB30 电路保护
12. USB30 协议分析与框架设计
13. USB 30 中的 CRC 校验原理及实现
14. 基于 CPLD 的 UART 设计
15. IPMI 在 VPX 系统中的应用与设计
16. 基于 CPCI 总线的 PMC 载板设计
17. 基于 VPX 总线的工件台运动控制系统研究与开发
18. PCI Express 流控机制的研究与实现
19. UART16C554 的设计
20. 基于 VPX 的高性能计算机设计
21. 基于 CAN 总线技术的嵌入式网关设计
22. Visual C 串行通讯控件使用方法与技巧的研究
23. IEEE1588 精密时钟同步关键技术研究
24. GPS 信号发生器射频模块的一种实现方案
25. 基于 CPCI 接口的视频采集卡的设计
26. 基于 VPX 的 3U 信号处理平台的设计
27. 基于 PCI Express 总线 1394b 网络传输系统 WDM 驱动设计
28. AT89C52 单片机与 ARINC429 航空总线接口设计
29. 基于 CPCI 总线多 DSP 系统的高速主机接口设计
30. 总线协议中的 CRC 及其在 SATA 通信技术中的应用
31. 基于 FPGA 的 SATA 硬盘加解密控制器设计
32. Modbus 协议在串口通讯中的研究及应用
33. 高可用的磁盘阵列 Cache 的设计和实现
34. RAID 阵列中高速 Cache 管理的优化

35. 一种新的基于 RAID 的 CACHE 技术研究与实现
36. 基于 PCIE-104 总线的高速数据接口设计
37. 基于 VPX 标准的 RapidIO 交换和 Flash 存储模块设计
38. 北斗卫星系统在海洋工程中的应用
39. 北斗卫星系统在远洋船舶上应用的研究
40. 基于 CPCI 总线的红外实时信号处理系统
41. 硬件实现 RAID 与软件实现 RAID 的比较
42. 基于 PCI Express 总线系统的热插拔设计
43. 基于 RAID5 的磁盘阵列 Cache 的研究与实现
44. 基于 PCI 总线的 MPEG2 码流播放卡驱动程序开发
45. 基于磁盘异或引擎的 RAID5 小写性能优化
46. 基于 IEEE1588 的时钟同步技术研究
47. 基于 Davinci 平台的 SD 卡读写优化
48. 基于 PCI 总线的图像处理及传输系统的设计
49. 串口和以太网通信技术在油液在线监测系统中的应用
50. USB30 数据传输协议分析及实现
51. IEEE 1588 协议在工业以太网中的实现
52. 基于 USB30 的设备自定义请求实现方法
53. IEEE1588 协议在网络测控系统中的应用
54. USB30 物理层中弹性缓冲的设计与实现
55. USB30 的高速信息传输瓶颈研究
56. 基于 IPv6 的 UDP 通信的实现
57. 一种基于 IPv6 的流媒体传送方案研究与实现
58. 基于 IPv4-IPv6 双栈的 MODBUS-TCP 协议实现
59. RS485CAN 网关设计与实现
60. MVB 周期信息的实时调度
61. RS485 和 PROFINET 网关设计
62. 基于 IPv6 的 Socket 通信的实现
63. MVB 网络重复器的设计
64. 一种新型 MVB 通信板的探究
65. 具有 MVB 接口的输入输出设备的分析
66. 基于 STM32 的 GSM 模块综合应用
67. 基于 ARM7 的 MVB CAN 网关设计

VxWorks：

1. 基于 VxWorks 的多任务程序设计
2. 基于 VxWorks 的数据采集存储装置设计
3. Flash 文件系统分析及其在 VxWorks 中的实现

4. VxWorks 多任务编程中的异常研究
5. VxWorks 应用技巧两例
6. 一种基于 VxWorks 的飞行仿真实时管理系统
7. 在 VxWorks 系统中使用 TrueType 字库
8. 基于 FreeType 的 VxWorks 中文显示方案
9. 基于 Tilcon 的 VxWorks 简单动画开发
10. 基于 Tilcon 的某武器显控系统界面设计
11. 基于 Tilcon 的综合导航信息处理装置界面设计
12. VxWorks 的内存配置和管理
13. 基于 VxWorks 系统的 PCI 配置与应用
14. 基于 MPC8270 的 VxWorks BSP 的移植
15. Bootrom 功能改进经验谈
16. 基于 VxWorks 嵌入式系统的中文平台研究与实现
17. VxBus 的 A429 接口驱动
18. 基于 VxBus 和 MPC8569E 千兆网驱动开发和实现
19. 一种基于 vxBus 的 PPC 与 FPGA 高速互联的驱动设计方法
20. 基于 VxBus 的设备驱动开发
21. 基于 VxBus 的驱动程序架构分析
22. 基于 VxBus 的高速数据采集卡驱动程序开发
23. Vxworks 下的冗余 CAN 通讯模块设计
24. WindML 工业平台下开发 S1d13506 驱动及显示功能的实现
25. WindML 中 Mesa 的应用
26. VxWorks 下图形用户界面开发中双缓冲技术应用
27. VxWorks 上的一种 GUI 系统的设计与实现
28. VxWorks 环境下 socket 的实现
29. VxWorks 的 WindML 图形界面程序的框架分析
30. VxWorks 实时操作系统及其在 PC104 下以太网编程的应用
31. 实时操作系统任务调度策略的研究与设计
32. 军事指挥系统中 VxWorks 下汉字显示技术
33. 基于 VxWorks 实时控制系统中文交互界面开发平台
34. 基于 VxWorks 操作系统的 WindML 图形操控界面实现方法
35. 基于 GPU FPGA 芯片原型的 VxWorks 下驱动软件开发
36. VxWorks 下的多串口卡设计
37. VxWorks 内存管理机制的研究
38. T9 输入法在 Tilcon 下的实现
39. 基于 VxWorks 的 WindML 图形界面开发方法
40. 基于 Tilcon 的 IO 控制板可视化测试软件的设计和实现
41. 基于 VxWorks 的通信服务器实时多任务软件设计
42. 基于 VXWORKS 的 RS485MVB 网关的设计与实现
43. 实时操作系统 VxWorks 在微机保护中的应用
44. 基于 VxWorks 的多任务程序设计及通信管理

Linux：

1. Linux 程序设计第三版及源代码
2. NAND FLASH 文件系统的设计与实现
3. 多通道串行通信设备的 Linux 驱动程序实现
4. Zsh 开发指南-数组
5. 常用 GDB 命令中文速览
6. 嵌入式 C 进阶之道
7. Linux 串口编程实例
8. 基于 Yocto Project 的嵌入式应用设计
9. Android 应用的反编译
10. 基于 Android 行为的加密应用系统研究
11. 嵌入式 Linux 系统移植步步通
12. 嵌入式 CC++语言精华文章集锦
13. 基于 Linux 的高性能服务器端的设计与研究
14. S3C6410 移植 Android 内核
15. Android 开发指南中文版
16. 图解 Linux 操作系统架构设计与实现原理（第二版）
17. 如何在 Ubuntu 和 Linux Mint 下轻松升级 Linux 内核
18. Android 简单 mp3 播放器源码
19. 嵌入式 Linux 系统实时性的研究
20. Android 嵌入式系统架构及内核浅析
21. 基于嵌入式 Linux 操作系统内核实时性的改进方法研究
22. Linux TCP IP 协议详解
23. Linux 桌面环境下内存去重技术的研究与实现
24. 掌握 Android 7.0 新增特性 Quick Settings
25. Android 应用逆向分析方法研究
26. Android 操作系统的课程教学
27. Android 智能手机操作系统的研究
28. Android 英文朗读功能的实现
29. 基于 Yocto 订制嵌入式 Linux 发行版
30. 基于嵌入式 Linux 的网络设备驱动设计与实现
31. 如何高效学习嵌入式
32. 基于 Android 平台的 GPS 定位系统的设计与实现
33. LINUX ARM 下的 USB 驱动开发
34. Linux 下基于 I2C 协议的 RTC 驱动开发
35. 嵌入式下 Linux 系统设备驱动程序的开发
36. 基于嵌入式 Linux 的 SD 卡驱动程序的设计与实现
37. Linux 系统中进程调度策略

38. 嵌入式 Linux 实时性方法
39. 基于实时 Linux 计算机联锁系统实时性分析与改进
40. 基于嵌入式 Linux 下的 USB30 驱动程序开发方法研究
41. Android 手机应用开发之音乐资源播放器
42. Linux 下以太网的 IPv6 隧道技术的实现
43. Research and design of mobile learning platform based on Android
44. 基于 linux 和 Qt 的串口通信调试器调的设计及应用
45. 在 Linux 平台上基于 QT 的动态图像采集系统的设计
46. 基于 Android 平台的医护查房系统的研究与设计
47. 基于 Android 平台的软件自动化监控工具的设计开发
48. 基于 Android 的视频软硬解码及渲染的对比研究与实现
49. 基于 Android 移动设备的加速度传感器技术研究
50. 基于 Android 系统振动测试仪研究
51. 基于缓存竞争优化的 Linux 进程调度策略

Windows CE：

1. Windows CE.NET 下 YAFFS 文件系统 NAND Flash 驱动程序设计
2. Windows CE 的 CAN 总线驱动程序设计
3. 基于 Windows CE.NET 的 ADC 驱动程序实现与应用的研究
4. 基于 Windows CE.NET 平台的串行通信实现
5. 基于 Windows CE.NET 下的 GPRS 模块的研究与开发
6. win2k 下 NTFS 分区用 ntldr 加载进 dos 源代码
7. Windows 下的 USB 设备驱动程序开发
8. WinCE 的大容量程控数据传输解决方案设计
9. WinCE6.0 安装开发详解
10. DOS 下仿 Windows 的自带计算器程序 C 源码
11. G726 局域网语音通话程序和源代码
12. WinCE 主板加载第三方驱动程序的方法
13. WinCE 下的注册表编辑程序和源代码
14. WinCE 串口通信源代码
15. WINCE 的 SD 卡程序[可实现读写的源码]
16. 基于 WinCE 的 BootLoader 研究
17. Windows CE 环境下无线网卡的自动安装
18. 基于 Windows CE 的可视电话的研究与实现
19. 基于 WinCE 的嵌入式图像采集系统设计
20. 基于 ARM 与 WinCE 的掌纹鉴别系统
21. DCOM 协议在网络冗余环境下的应用
22. Windows XP Embedded 在变电站通信管理机中的应用

23. XPE 在多功能显控台上的开发与应用
24. 基于 Windows XP Embedded 的 LKJ2000 仿真系统设计与实现
25. 虚拟仪器的 Windows XP Embedded 操作系统开发
26. 基于 EVC 的嵌入式导航电子地图设计
27. 基于 XPEmbedded 的警务区 SMS 指挥平台的设计与实现
28. 基于 XPE 的数字残币兑换工具开发

PowerPC：

1. Freescale MPC8536 开发板原理图
2. 基于 MPC8548E 的固件设计
3. 基于 MPC8548E 的嵌入式数据处理系统设计
4. 基于 PowerPC 嵌入式网络通信平台的实现
5. PowerPC 在车辆显控系统中的应用
6. 基于 PowerPC 的单板计算机的设计
7. 用 PowerPC860 实现 FPGA 配置
8. 基于 MPC8247 嵌入式电力交换系统的设计与实现
9. 基于设备树的 MPC8247 嵌入式 Linux 系统开发
10. 基于 MPC8313E 嵌入式系统 UBoot 的移植
11. 基于 PowerPC 处理器 SMP 系统的 UBoot 移植
12. 基于 PowerPC 双核处理器嵌入式系统 UBoot 移植
13. 基于 PowerPC 的雷达通用处理机设计
14. PowerPC 平台引导加载程序的移植
15. 基于 PowerPC 嵌入式内核的多串口通信扩展设计
16. 基于 PowerPC 的多网口系统抗干扰设计
17. 基于 MPC860T 与 VxWorks 的图形界面设计
18. 基于 MPC8260 处理器的 PPMC 系统
19. 基于 PowerPC 的控制器研究与设计
20. 基于 PowerPC 的模拟量输入接口扩展
21. 基于 PowerPC 的车载通信系统设计
22. 基于 PowerPC 的嵌入式系统中通用 IO 口的扩展方法
23. 基于 PowerPC440GP 型微控制器的嵌入式系统设计与研究
24. 基于双 PowerPC 7447A 处理器的嵌入式系统硬件设计
25. 基于 PowerPC603e 通用处理模块的设计与实现
26. 嵌入式微机 MPC555 驻留片内监控器的开发与实现
27. 基于 PowerPC 和 DSP 的电能质量在线监测装置的研制
28. 基于 PowerPC 架构多核处理器嵌入式系统硬件设计

ARM：

1. 基于 DiskOnChip 2000 的驱动程序设计及应用
2. 基于 ARM 体系的 PC-104 总线设计
3. 基于 ARM 的嵌入式系统中断处理机制研究
4. 设计 ARM 的中断处理
5. 基于 ARM 的数据采集系统并行总线的驱动设计
6. S3C2410 下的 TFT LCD 驱动源码
7. STM32 SD 卡移植 FATFS 文件系统源码
8. STM32 ADC 多通道源码
9. ARM Linux 在 EP7312 上的移植
10. ARM 经典 300 问
11. 基于 S5PV210 的频谱监测设备嵌入式系统设计与实现
12. Uboot 中 start.S 源码的指令级的详尽解析
13. 基于 ARM9 的嵌入式 Zigbee 网关设计与实现
14. 基于 S3C6410 处理器的嵌入式 Linux 系统移植
15. CortexA8 平台的μC-OSⅡ及 LwIP 协议栈的移植与实现
16. 基于 ARM 的嵌入式 Linux 无线网卡设备驱动设计
17. ARM S3C2440 Linux ADC 驱动
18. ARM S3C2440 Linux 触摸屏驱动
19. Linux 和 Cortex-A8 的视频处理及数字微波传输系统设计
20. Nand Flash 启动模式下的 Uboot 移植
21. 基于 ARM 处理器的 UART 设计
22. ARM CortexM3 处理器故障的分析与处理
23. ARM 微处理器启动和调试浅析
24. 基于 ARM 系统下映像文件的执行与中断运行机制的实现
25. 中断调用方式的 ARM 二次开发接口设计
26. ARM11 嵌入式系统 Linux 下 LCD 的驱动设计
27. Uboot 在 S3C2440 上的移植
28. 基于 ARM11 的嵌入式无线视频终端的设计
29. 基于 S3C6410 的 Uboot 分析与移植
30. 基于 ARM 嵌入式系统的高保真无损音乐播放器设计
31. UBoot 在 Mini6410 上的移植
32. 基于 ARM11 的嵌入式 Linux NAND FLASH 模拟 U 盘挂载分析与实现
33. 基于 ARM11 的电源完整性分析
34. 基于 ARM S3C6410 的 uboot 分析与移植
35. 基于 S5PC100 移动视频监控终端的设计与实现

Hardware:

1. DSP 电源的典型设计
2. 高频脉冲电源设计
3. 电源的综合保护设计
4. 任意波形电源的设计
5. 高速 PCB 信号完整性分析及应用
6. DM642 高速图像采集系统的电磁干扰设计
7. 使用 COMExpress Nano 工控板实现 IP 调度设备
8. 基于 COM Express 架构的数据记录仪的设计与实现
9. 基于 COM Express 的信号系统逻辑运算单元设计
10. 基于 COM Express 的回波预处理模块设计
11. 基于 X86 平台的简单多任务内核的分析与实现
12. 基于 UEFI Shell 的 PreOS Application 的开发与研究
13. 基于 UEFI 固件的恶意代码防范技术研究
14. MIPS 架构计算机平台的支持固件研究
15. 基于 UEFI 固件的攻击验证技术研究
16. 基于 UEFI 的 Application 和 Driver 的分析与开发
17. 基于 UEFI 的可信 BIOS 研究与实现
18. 基于 UEFI 的国产计算机平台 BIOS 研究
19. 基于 UEFI 的安全模块设计分析
20. 基于 FPGA NiosⅡ的等精度频率计设计
21. 基于 FPGA 的 SOPC 设计
22. 基于 SOPC 基本信号产生器的设计与实现
23. 基于龙芯平台的 PMON 研究与开发
24. 基于 X86 平台的嵌入式 BIOS 可配置设计
25. 基于龙芯 2F 架构的 PMON 分析与优化
26. CPU 与 GPU 之间接口电路的设计与实现
27. 基于龙芯 1A 平台的 PMON 源码编译和启动分析
28. 基于 PC104 工控机的嵌入式直流监控装置的设计
29. GPGPU 技术研究与发展
30. GPU 实现的高速 FIR 数字滤波算法
31. 一种基于 CPUGPU 异构计算的混合编程模型
32. 面向 OpenCL 模型的 GPU 性能优化
33. 基于 GPU 的 FDTD 算法
34. 基于 GPU 的瑕疵检测
35. 基于 GPU 通用计算的分析与研究

Programming:

1. 计算机软件基础数据结构 - 算法
2. 高级数据结构对算法的优化
3. 零基础学算法
4. Linux 环境下基于 TCP 的 Socket 编程浅析
5. Linux 环境下基于 UDP 的 socket 编程浅析
6. 基于 Socket 的网络编程技术及其实现
7. 数据结构考题 - 第 1 章 绪论
8. 数据结构考题 - 第 2 章 线性表
9. 数据结构考题 - 第 2 章 线性表 - 答案