分　类　号：TP393　　单位代码：10183

研究生学号：2014534062　　密　　级：公 开

吉 林 大 学

硕士学位论文

（专业学位）

GPGPU 结构研究与性能分析

The Study of GPGPU Microarchitecture and Performance Analysis

作者姓名：邢千里

专　　业：计算机技术

研究方向：GPGPU 体系结构

指导教师：何立波　副教授

培养单位：计算机科学与技术学院

2017 年 4 月

GPGPU 结构研究与性能分析

The Study of GPGPU Microarchitecture and Performance Analysis

作者姓名： 邢千里

专业名称： 计算机技术

指导教师： 何立波 副教授

学位类别： 工程硕士

答辩日期： 2017 年 5 月 23 日

吉林大学博士(或硕士)学位论文原创性声明

本人郑重声明：所呈交学位论文，是本人在指导教师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：邢子里

日期：2017 年 5 月 25 日

摘　要

GPGPU 结构研究与性能分析

在过去的十几年里 GPU 处理性能的增长十分迅猛。GPU 在结构上与 CPU 有很大的不同，在 GPU 中有更多的晶体管用于计算，而 CPU 中更多的晶体管用于逻辑控制。因此在不同的设计目的之下，他们的作用也变得不同。更近一步，GPU 迅速从图像处理领域发展到通用计算领域，由此开启了一个新的领域叫做 GPGPU（General-Purpose Computing on the Graphic Processing Unit）。

GPGPU 是为处理并行任务而设计的，所以对并行计算模型的研究是很有意义的。虽然 PRAM 模型、BSP 模型和 logP 模型等经典的并行计算模型已经提出很多年，但是通过对这些模型的研究可以更加深刻的理解 GPGPU 结构。

从 GPGPU 这个概念被提出开始，很多的研究集中在利用其强大的计算能力，对于处理某一问题的效率进行大幅度提升。这一现象主要原因在于芯片的详细结构、流水线以及存储设计都涉及到商业机密，很难获得这些资料用于研究。英伟达和 AMD 是两家主要生产 GPGPU 的厂家，相比较之下英伟达的官方文档更加详细，其 CUDA 套件也更加完备，因此本文以英伟达的芯片作为研究重点。本文选择了开源的 GPGPU-Sim 模拟器，对英伟达的 GPU 进行模拟。

本文对一些并行计算模型，比如 PRAM 模型、BSP 模型和 logP 模型等进行了对比研究，比较了其参数的异同以及核心思想，并且对当前 GPU 的研究现状做了简单综述。随后，本文给出了一个全新的 NKGPGPU，对硬件结构、任务的逻辑结构、代码结构以及其中的映射关系做出了详细构架。整体上，NKGPGPU 包括五个子模型，分别是硬件结构子模型、任务结构子模型、任务组织子模型、任务执行子模型以及任务调度子模型。硬件结构子模型主要给出了 NKGPGPU 芯片中的主要组成部件。任务组织子模型主要给出了适用于 NKGPGPU 的代码结构以及代码和任务之间的映射，除此之外还给出了任务之间的启动关系模型。任务执行子模型这一部分给出了代码和硬件之间的映射。任务调度子模型给出了任务拓扑结构和硬件结构的映射。同时本文给出了一个性能分析模型，使它符合本文提出的 NKGPGPU。对于影响 GPGPU 性能的主要三个方面：GPGPU 流水线、共享存

储和全局存储，本文在不同线程数目的情况下进行了详细的实验。对 GPGPU 的流水线的实验主要是研究对于不同类型的指令的运行周期的差异，通过这个差异来判断指令与流水线之间的关系。研究共享内存和全局内存的方法类似，都是通过连续的访存指令测试完成周期。

本文提出的 NKGPGPU 丰富了 GPGPU 的理论模型，为 GPGPU 硬件工程师和软件编程人员提供了改进的依据，对于 GPGPU-Sim 的实验方法和思路可以作为进一步研究 GPGPU 的基础。

关键字：GPGPU，Fermi，PTX 语言，硬件结构

Abstract

The Study of GPGPU Microarchitecture and Performance Analysis

In the past ten years, GPU processing performance growth is very rapid. The GPU is structurally different from the CPU, with more transistors in the GPU for calculation, and more transistors in the CPU for logic control. So their role is different under different design goals. More quickly, GPU quickly from the field of image processing to the general field of computing, which opened a new field called GPGPU (General-Purpose Computing on the Graphic Processing Unit).

GPGPU is designed to handle parallel tasks, so the study of parallel computing model is very meaningful. Although classical parallel computing models such as PRAM model, BSP model and logP model have been proposed for many years, the GPGPU structure can be understood more deeply by studying these models.

Starting from the concept of GPGPU showing, a lot of research focused on using its powerful computing power to dramatically improve the efficiency of dealing with a single problem. The main reason for this phenomenon is that the detailed structure of the chip, pipeline and storage design are related to trade secrets, it is difficult to obtain such information for research. NVIDIA and AMD are the two major manufacturers of GPGPU, and compared with AMD, NVIDIA's official documents are more detailed. Besides its CUDA suite is more complete, so this article take NVIDIA's chip as a research focus. This paper chooses the open source GPGPU-Sim simulator to simulate the NVIDIA GPU.

In this paper, some parallel computing models, such as PRAM model, BSP model and logP model, are compared. The similarities and differences of the parameters and the core ideas are compared, and the current research status of the current GPU is briefly reviewed. Then, this paper gives a new NKGPGPU, the hardware structure, the logical structure of the task, the code structure and the mapping relationship between which made a detailed structure. On the whole, NKGPGPU includes five sub-models, namely hardware structure sub-model, task structure sub-model, task organization sub-model, task execution sub-model and task scheduling sub-model. The hardware

structure sub-model mainly gives the main components of the NKGPGPU chip. The task organization sub-model mainly gives the code structure suitable for NKGPGPU and the mapping between code and task. In addition, the starting relation model between tasks is given. Task execution sub-model gives the mapping between code and hardware. The task scheduling sub-model gives the mapping of the task topology and hardware structure. At the same time, this paper presents a performance analysis model to make it conform to the proposed NKGPGPU. For the three main aspects that affect the performance of GPGPU: GPGPU pipeline, shared storage and global storage, this paper has carried on the detailed experiment in the case of different thread number. The GPGPU pipeline is mainly to study the different types of instructions for the operation of the cycle of the difference, through this difference to determine the relationship between the instruction and the pipeline. Research on shared memory and global memory is similar to the method, through the continuous access command to complete the test cycle.

The proposed model of GPGPU is useful to GPGPU hardware engineers and software programmers, and the experimental methods and ideas for GPGPU-Sim can be used as a basis for further study of GPGPU.

Keywords: GPGPU, Fermi, PTX, Microarchitecture

目　录

第 1 章 研究意义及背景知识

1.1 研究意义

现在是大数据时代，顾名思义有大量的数据需要处理，同时也就需要与数据量匹配的“大”计算能力。CPU 的高速发展使得串行程序得到了极致发挥，然而在面对数据并行等计算密集型任务时，CPU 并不总能处理的十分高效。相比之下，作为为数据并行而生的 GPU 可以将性能提升 10 倍甚至 100 倍。英伟达公司公司给出的产品白皮书给出了 GPU 的峰值计算性能已经远超 CPU。时至今日，GPU 已经不仅局限于图像处理，在各个领域都有 GPU 的身影，所以称它为 GPGPU(General-Purpose Computing the Graphic Processing Unit) 更加合适。

GPGPU 的超级计算能力背后的硬件结构、流水线、存储等的设计令人十分好奇。对于 GPGPU 结构的研究可以使得人们不仅可以利用其计算能力，并且能深入理解其设计思想。对硬件设计者，对 GPGPU 结构的研究可以帮助他们设计出性能更加卓越的 GPU。对于编程人员，研究 GPGPU 结构可以在软件开发和硬件设计之间找到一个最佳性能点，进而发挥出硬件的极限能力。对 GPGPU 结构研究还可以在软硬件之前架起一道桥梁，硬件设计者和软件开发者可以共同设计出适应性更强的 GPGPU 编程模型。

1.2 GPU 体系结构

CPU 结构的核数量一般都比较少， 2 个、4 个、8 个的情况较为常见，而 GPU 则完全不一样，通常有几百甚至上千的的核。和 CPU 复杂的核相比，GPU 的核通常都比较简单，比如，GPU 的核在执行顺序指令时不会有旁路网络或者分支预测。现代 GPU 的设计都是把核封装在流处理器（Streaming Multiprocessor，简称 SM）中，SM 是一个相对独立的结构，包含了 GPU 活动的大部分过程，可以把它理解为一个小型 GPU，所以 SM 是一个需要重点研究的单元。不同的结构之间核数量相差很大，SM 数量差异也很大。SM 通过片上网络和 L2 缓存和片外内存通信，SM 之间并不能通过片上网络进行通信。内存和 L2 缓存是分块的，然后和 SM 相连接。

SM 的设计在每一代的进步也都很明显，下面一部分主要英伟达的 Fermi GPU 结构。英伟达的 Kepler 结构和 Maxwell 结构更加复杂，但是设计的基本思想与 Fermi GPU 类似，所以不再做详细介绍。

1.2.1 Fermi 结构

在 GPU 出现后，很多研究都希望把其卓越的计算能力用于其他非图像领域，也就是 GPGPU 编程。在 Fermi 结构发布之前，在 GPU 上使用过 DirtectX,OpenGL 和 Cg 等编程框架。但是使用这些框架对编程人员的要求很高，需要编程人员懂得相应的 API 以及 GPU 结构，同时编程过程也十分复杂。另一方面，一些功能，例如随机读写，双精度运算都还不能支持。Fermi 结构的出现解决了这一问题。对于 Fermi 结构，NVDIA 官方给出的白皮书[1]做了十分详细的介绍，本文只对 Fermi 结构的一部分进行简单介绍。下文会有一些英文缩写或者名称，为了保持其准确性，笔者就保留了其原有写法而没有翻译。

第一个使用 Fermi 架构的 GPU 包括 16 个 SM，每个 SM 包括 32 个 CUDA 核，一共 512 个 CUDA 核，有 6GB 的 GDDR5 DRAM，全局 GigaThread 调度器。每一个 SM 主要包含 16 个 load/store 单元，4 个特殊功能单元（special function units，简称 SFU）用来执行一些三角函数；还有对偶的 warp 调度器（对于 warp 会在 CUDA 编程结构给出），即两个 warp 调度器，这样的设计可以使两个 warp 同时执行，并且可以使得 Fermi 结构的 GPU 的性能达到最佳；每个 SM 还会有 64KB 的可编程的共享存储（shared memory）和一级缓存（L1 cache）。具体 SM 的结构如下图 1.1 所示。

Fermi 结构发布于 2009 年，后续还有 Kepler 结构，Maxwell 结构。在性能方面 Fermi 显然已经不是最优秀的结构，但是可以说它开启了 GPGPU 的可编程的新时代，并且本文的研究不留于计算性能表面，而是要深入到 GPU 内部，研究其内在结构以及流水线，而这些信息又往往涉及到商业机密很难获得，所以只能同过一个 GPGPU-Sim 模拟器[2]来研究。这个 GPGPU-Sim 本文也会做出较为详细的介绍， GPGPU-Sim 模拟器在设计之初就是 Fermi 结构，并且获得了英伟达的官方支持，所以对于其结构和流水线的研究可以在一定程度上反映真实 GPU 的情况。

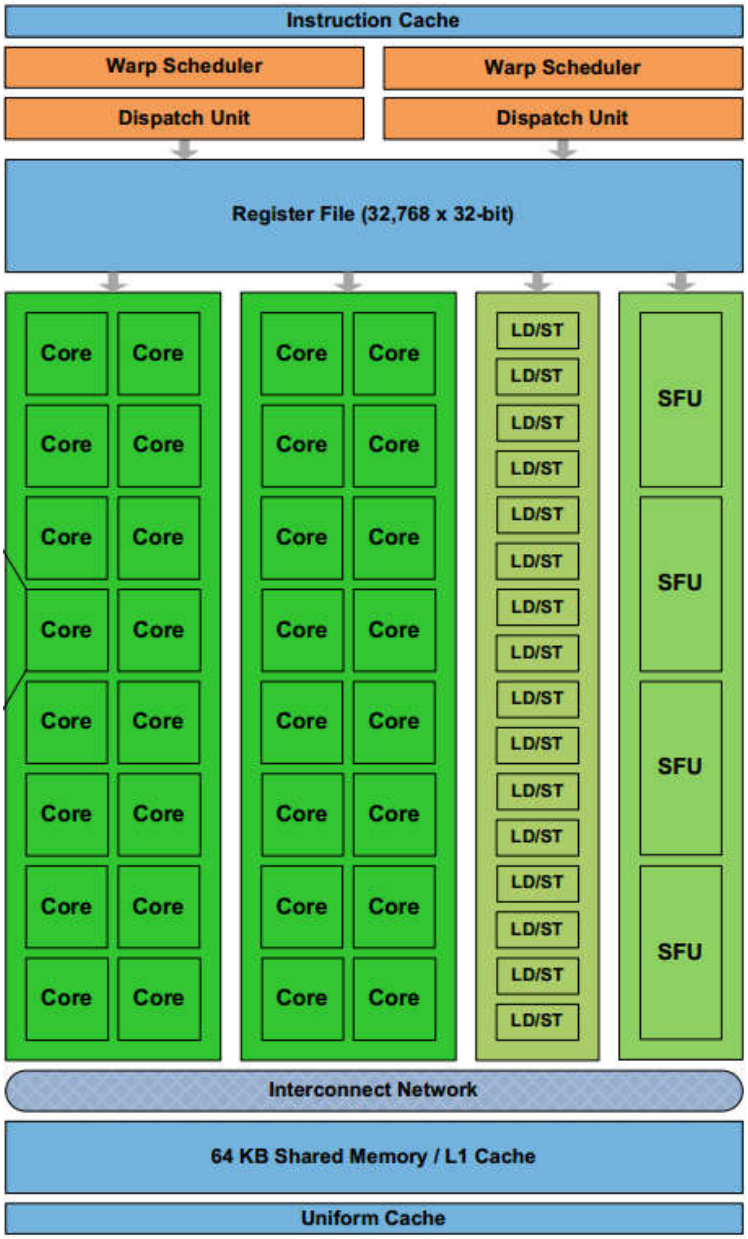

图 1.1 SM 结构图

1.2.2 GPGPU-Sim

GPGPU-Sim 是一个可以精确到周期的模拟器，它模拟了英伟达的 Tesla 和 Fermi 架构的 GPU 结构。用户可以在这两种结构之间进行选择，并且可以配置很多的参数，例如 warp 调度器，cache 的大小等。这个模拟器基于英伟达的官方文件，专利以及一些英伟达官方的支持和一些假设，由 UBC 大学的团队开发和维护。下图 1.2 给出了 SIMT 的微观结构，整体上结构分成了 SIMT 前端和 SIMD 后端。这一部分主要对 GPGPU-Sim 的内部结构进行一个详细的描述。

在 SIMT 前端，指令从指令 cache 中取出，然后进行译码，并且存在指令 buffer。然

后在计分板中，检查指令之间的 RaW 依赖和 WaW 依赖，由调度器确定哪一个 warp 流出到 SIMD 后端。SIMT-Stack 用于处理分支预测，SIMT-Stack 会存储每一个 warp 的分支点和汇合点。

SIMD 数据路径包含 4 个操作数收集器（Operand collector），两组 16 核的流处理器（streaming processor ,SP），4 个特殊功能寄存器，共享存储（shared memory）。

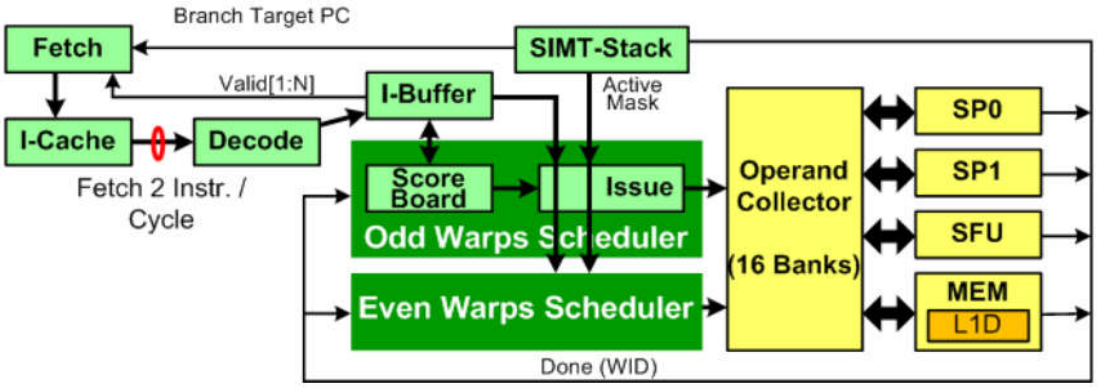

图 1.2 GPGPU-Sim 流水线

图 1.3 更详细的描述这个流水线。指令从指令高速缓存（instruction cache）被取出。然后指令被译码（decode），并且存在指令缓冲区（instruction buffer）。RaW 和 WaW 的依赖检查在得分板（scoreboard）中进行。在 GPGPU-Sim 中只有一个得分板，调度器选择准备好的 warp 进入执行单元。更进一步，SIMT-Stack 用于控制控制分歧（branch divergence）。它会存储每个 warp 的分支点（PR）和汇合点（RPC）。

SIMD 数据路径包含操作数收集器（operand collector），共享内存（shared memory）和 SIMD 执行单元（SP 和 SFU）。实际上寄存器文件内存（register file）是一个 4 组 bank 的 SRAM 内存，并且有一个逻辑单元来避免 bank 冲突。

蓝色部分表示存储，绿色部分表示系统的 buffer，计分板，所有线程的 PC，指令 buffer，SIMT Stack 和收集器单元。橙色部分是一个仲裁器结构，棕色部分是一个包含 4 组的寄存器文件存储结构。紫色部分展示了 SIMD 的执行单元。

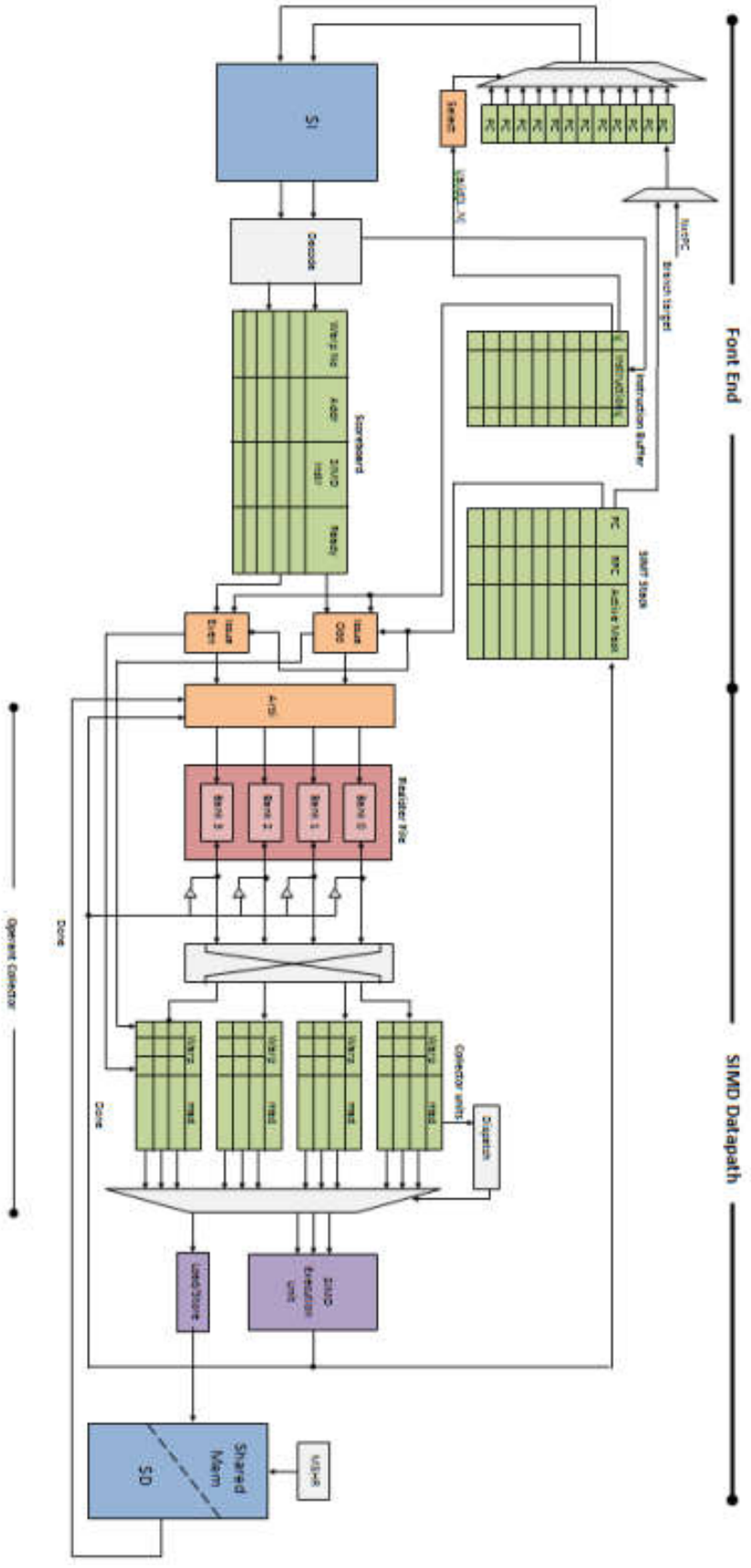

图 1.3 详细 GPGPU-Sim 流水线

1.3 GPU 编程

现在，GPU 不仅用于图形图像处理，还用在很多科学计算领域。在科学计算领域中，有很多的计算密集型程序，GPU 可以很好的对这种程序进行加速。GPU 通过编程模型，例如统一设备架构（CUDA）、OpenCL，来实现大规模并行，以加速程序。编程人员可以根据 GPU 的特性对算法进行相应的选择、优化，这样可以更好的利用硬件的性能。CUDA 和 OpenCL 都提供了高速缓存、线程 ID 以及同步机制。CPU 和 GPU 具有相对独立的存储单元。待处理的数据会明确的在 CPU 设备和 GPU 设备之间传输，如果 GPU 中的计算量非常小，那么传输时间将会严重影响程序的加速。

CUDA 和 OpenCL 都以 C 语言为基础，并做了一些扩展以适应 GPU。GPU 编程主要可分为两部分，第一部分为 host 代码，即运行在 CPU 中的代码，第二部分为 device 代码，即运行在 GPU 的代码。host 代码负责分配 GPU 的内存空间、在 CPU 和 GPU 之间进行数据传输，还要负责启动在 GPU 上运行的核函数（kernel）。kernel 也就是所谓的 device 代码，就是 GPU 执行的代码。CUDA 给出了一些概念，包括线程（thread）、线程组（warp）、线程块（block）、线程网格（grid），具体的解释如图 1.4 所示。

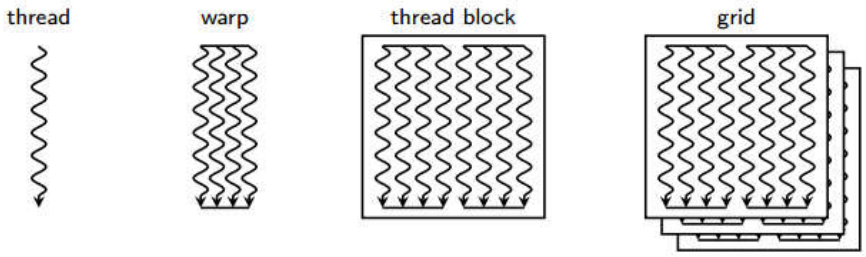

图 1.4 任务逻辑示意图

OpenCL 中和 CUDA 概念相对应的部分也在下表 1.1 中给出。

表 1.1 CUDA 和 OpenCL 对比示意图

CUDA	OpenCL
thread	work-item
warp	wavefront
thread block	work-group
grid	Computation domain
global memory	global memory
shared memory	local memory
local memory	private memory
streaming multiprocessor(SM)	compute unit
scalar core	processing element

在编程过程中，需要编程人员明确的给出每一个 block 中的 thread 数量，以及 block 的数量。grid 的大小就是 block 的数量乘以每一个 block 中的 thread 数量。每一个 block 通常包含几百个 thread。一个 block 只能执行在一个 SM 上，一个 SM 上可以执行多个 block。在 GPU 上，block 中的 thread 自动划分成若干个 warp，在 CUDA 中，一个 warp 包含 32 个 thread，warp 的执行方式类似于 SIMD。这样的模式中只需要一次取指和译码，warp 中的所有 thread 都可以执行，所以可以使 GPU 非常高效。但是基于这种模型，也使得 GPU 会有分支分歧（branch divergence）问题。在一个 warp 中，如果存在 if 的不同分支，则所有的分支都需要执行一遍。

几十到几百的 thread 构成一个 grid，GPU 会把 thread 动态分配到 SM 上，并且不同 block 之间的 thread 是不能通信的。block 的数量是独立于 SM 的，但是每个 SM 能承载的 block 数量是有限的。区分 GPU 计算性能主要看 GPU 包含 SM 的数量。

通常情况下 thread 的数量会远大于核的数量。为了隐藏流水线和数据转移的延迟，GPU 使用了细粒度的多线程方式。在执行一个 warp 指令后，GPU 会迅速切换到另一个 warp 指令。实现这一过程需要大量的寄存文件（register file）来保存上下文信息。这种方式也保证了“一条指令多次执行（SIMD）”的执行方式，并且可以在每一个指令执行后切换线程。

GPU 的更新换代的速度很快，并且现在也有着不同的硬件设计和结构，CUDA 语言作为一个扩展 C 语言的通用计算平台可以使得编程人员可以很方便的利用其计算能力。但是，由于 GPU 的内部的不同，同样的代码在不同架构的 GPU 上的性能可能会有很大出入。调试 GPU 程序（例如修改数据的访存模式）以及使 GPU 的计算能力发挥最大，就需要 PTX 语言了。PTX 目标在于提供一个稳定的编程模型和一个通用计算指令集。

PTX 包含一个低级别的执行虚拟机和一个指令集结构（ISA）。PTX 可以完全发挥 GPU 性能，使其成为了数据并行的计算设备。PTX 的目标在于提供一个稳定的编程模型和一个通用计算指令集。PTX 是 Parallel Thread Execution 的缩写，与之对应的编程模型是一个明确的并行程序，具体来看一个 PTX 程序会明确在线程队列中哪些线程是并行执行的。一组联合线程队列（CTA），就是一组可以同时执行的的线程。PTX 的硬件执行模型可以说与 CUDA 是完全一样的，因为 PTX 是 CUDA 中的 NVCC 编译器根据 GPU 型号而生成的。

其语言格式如图 1.5 所示，类似于汇编语言 SASS，但是比汇编语言更加容易理解，可读性更好，编程人员可以直接进行 PTX 编程。

Examples

```
        .reg      .b32 r1, r2;
        .global   .f32  array[N];

start:  mov.b32   r1, %tid.x;
        shl.b32   r1, r1, 2;          // shift thread id by 2 bits
        ld.global.b32 r2, array[r1];  // thread[tid] gets array[tid]
        add.f32   r2, r2, 0.5;        // add 1/2
```

图 1.5 PTX 语言程序例子

1.4 相关工作

1.4.1 并行计算模型对比研究

在英伟达的 GPGPU 芯片问世之前，就已经有了对于并行计算的模型研究。可以说 GPGPU 的研究属于并行计算的一个重要部分，并且其通用性和适应性在现在看来远好于其他的模型。但是一些经典的并行计算模型还是有一定的研究价值，其内在思想还是可以很好的指导 GPGPU 的设计。

在这一部分本文主要对比的是文献[3]和其他一些模型之间的关系。胡亮等人[3]给出了全新一代的 GPGGU 的描述模型，因此本文会称它为新模型，新模型解释了 GPGPU 微观的架构和运行机制，详细描述了其任务组织、硬件结构、调度原理运行原理和存储过程。其目的是在编程人员和硬件设计者之间架起一架桥梁来追求硬件能力的极限。

PRAM[4]、 BSP[5]和 logP[6]的扩展就是在扩展处理器数量，而新模型是在扩展完整一层。即之前这些模型都是在扩展处理器数量，而胡亮等[3]是在扩展一层中所有的部件。

从第 i 层确定第 i-1 层中的 PEm i-1 的数量：

在 BSP 模型（不支持并发读写）的分析中，在 p 物理处理器上模拟 v≥plogp 处理器时，一个超级步的最佳运行时间是 O（v/p）。那么对于新模型，第 i 层可以看做物理处理器，第 i-1 层可以看做虚拟处理器，BSP 模型中的 L 是对数的，BSP 模型采用散列的方式是访存均匀，新模型中都可以完成相应工作，那么应该可以对第 i-1 层的 PEm i-1 数量做出一个相似的结论。在支持并发读写的 BSP 模型中，v=p1+ ε , ε >0,L≥log p,一个超级步的最佳运行时间也可以达到 O（v/p）。

从硬件角度来看：PRAM 模型的寄存器单元以及处理器内部的各个单元在新模型中都有对照。在文献[3]的模型中，把处理器中 PRAM 模型中处理器内部的程序计数器、累加器、状态运行标志抽象成了 TB 部件。PRAM 部件的寄存器 0 用作输入数据，对应于新模

型中的 shared/local memory。

胡亮等人提出[3]的模型的本质就是一个冯诺依曼模型，如果我们去掉输入输出设备，则与 LogP 中对于硬件趋同的描述基本一致，因此我们可以采用 LogP 参数的思想，L 为通信延迟的上界限，o 是处理器开销时间，g 表示一个处理器两次接收（或者发送）的时间间隔，P 代表处理器个数。但是根据分层的想法，前面对于处理器的描述则变成对于一个 PE 单元的描述，并且不同层的硬件性能是不同的，因此参数同样需要分层，即 Li，oi，gi，Pi。对于这四个参数，不同层之间可能会有数量级的差别，而有一些则可能是同一数量级，所以同一参数，不同层层数之间的参数之间的关系也需要考量。我们可以假设 L、o、g、P 都是关于层数 i 的函数。这也比较符合一个发展的眼光来看问题。

BSPRAM 模型[7]中提出了两层的存储模型：BSPRAM 模型中为了充分利用数据的本地性，保留了处理器的存储单元，为了可以支持 PRAM 编程模型，所有处理器之间用一个整体的 shared memory 相连接。也就是说 BSPRAM 模型的存储就分成了两层，local memory 和 shared memory。在用 CUDA 编程时，发现在 block 粒度来看这个存储结构与 BSPRAM 基本一致，一个 block 中所有的 thread 会共享一块 shared-memory，每一个 thread 属于任务层面，其对应的物理结构就是流处理单元）会分配 local memory。新模型是基于发展的眼光来看问题，存储是可以分成很多层的，也就可以在一定程度上，既支持 shared-memory 的编程方式，也同样可以充分的利用数据的本地性。 图 1.6 给出了 BSP 型模型和 BSP-RAM 模型与新模型的对比。

胡亮等人在文献[8]对 GPU 硬件和 GPGPU 云进行了映射，给出了对于 GPGPU 任务的一些描述。更进一步，文献[3]的 Final Remarks 部分中对硬件结构的描述中提到了未来了 GPU 有可能打破计算任务和存储任务的边界，这一点上 BSP 模型对部件的分析就显得很有前瞻性，它的部件的功能就是处理计算或者访存的任务。

Task execute 子模型中的执行流水线分成了三个基准线：分配上下文；消耗代码；释放上下文；在这一部分存在一个[PE]的数量的问题，如果设计合适，就能是流水线发挥出最佳的优势。

BSP 模型的一个超级步的过程就是输入阶段、本地计算阶段、输出阶段。BSP 模型中一个超级步的计算公式为 w+hg+L，其中 w 为本地计算时间，h 代表接受/发送最多的数据，L 是一个同步周期。那么假设有 s 个超级步，则总共的计算时间如下公式:

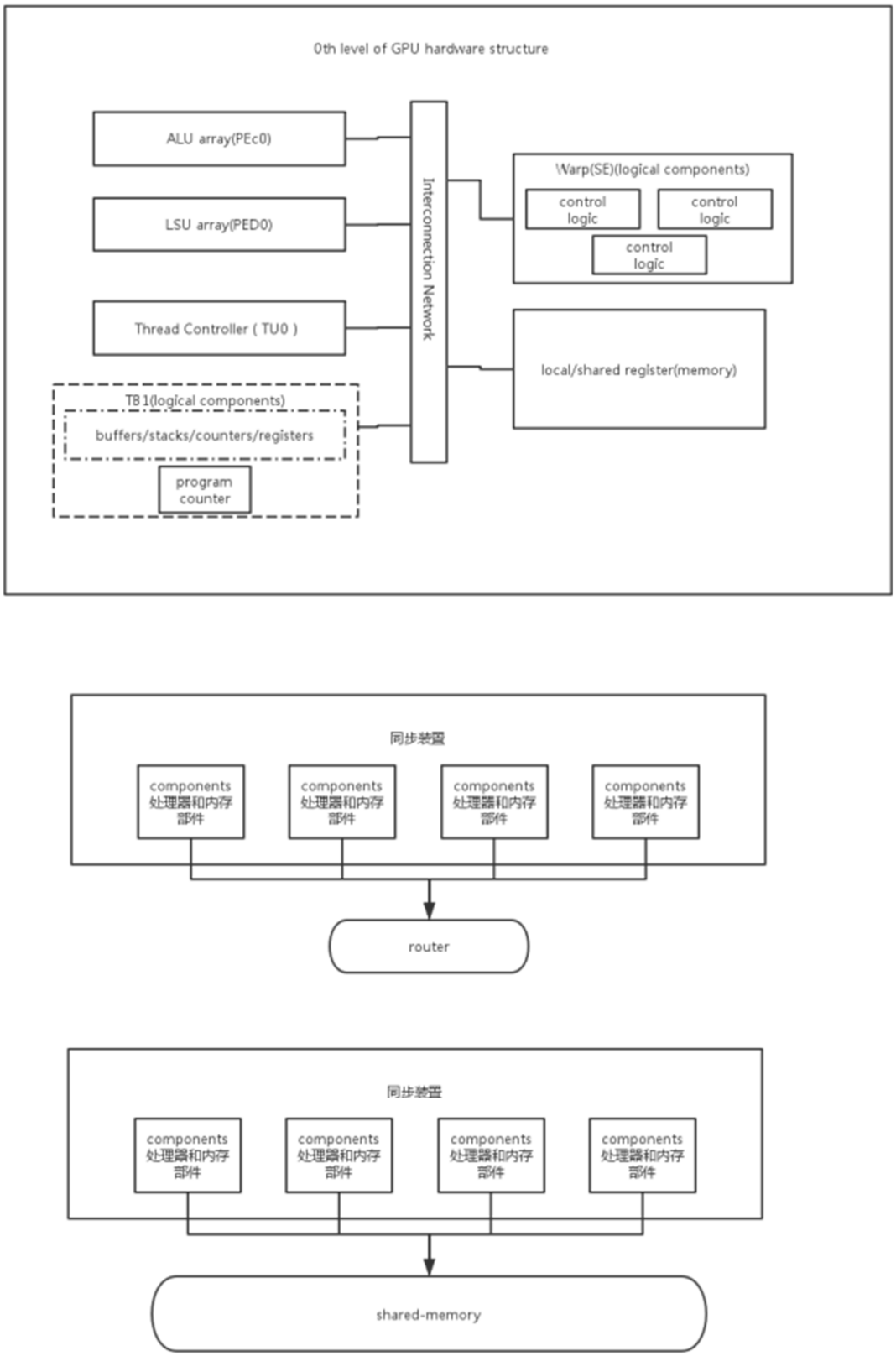

图 1.6 BSP 模型 BSP-RAM 模型对比

$$T_{BSP} = \sum_{k=0}^{s-1} w_k + g\sum_{k=0}^{s-1} h_k + sL$$

对应于 BSP 一个超级步的三个阶段则 w 对应于本地计算，而输入输出阶段就隐含在 $g\sum_{k=0}^{s-1} h_k$ 之中。因为一个超级步就相当于 BSP 模型的执行机制，那么我们或许可以从中受到一些启发，对新模型的执行流水线进行类似的一个定量分析，所以需要考量的问题就变成了如何抽象分配上下文和释放上下文，对本地计算和同步时间可以做以下保留。

BSP 模型强调了计算任务和通信任务的分开，而新模型强调了计算任务和访存任务的分开。也就是可以从访存的角度和上下午的角度来确定新的开销计算模型。对于 BSP 的研究有了更进一步的 Multi-BSP[9]模型，这一模型研究的焦点在于多核通信之间的问题。

1.4.2 定量分析

PRAM 模型、logP 模型、BSP 模型等主要从硬件角度来定量分析模型；朱俊峰等人[10]和 KOTHAPALLI.K 等人[11]则是从任务角度（或者叫指令的执行角度）分析了某一任务的执行需要的时间。两篇文章的核心思想十分类似，都定量分析了不同指令的时钟周期，以及访存不同部件的时钟周期；不同在于前一篇文献[10]中的 batch 作为基本的分析单元，KOTHAPALLI.K 等人[11]以 CUDA 中的 thread 作为基本单元。对于使用调度来隐藏访存时间这一部分连篇多没有做出一个很好的解释，文献[11]对此问题给出了一个上界和一个下界，即访存时间全部被隐藏 MAX 模型，访存时间全部没有被隐藏 SUM 模型，实际运行时间也确实在这这两个模型的预测时间之间。

结合新模型的任务模型和调度模型，对于 0th-level，需要考虑的就是 TI0，就是 thread，对于 1st-level 需要考虑的就是 TI1，就是 block。现在来考虑平均的等待时间，那么就需要考虑整体，也就是所有的 TIi-1，下面我以 TI1，即 block 为例进行分析。

我们借鉴一下超级步的思想，假设总共的 kernel 需要 Nsuperstep 个逻辑步完成。每个逻辑步中，所有的 block 可以并行；箭头表示依赖关系，第 n+1 个逻辑步中的 block 需要等待第 n 个逻辑步中的 block 完成才能运行如图 1 所示。这里面的逻辑步不是一个同步的概念，更像是一个逻辑时间先后的概念。不是说逻辑步中的 block 必须在开始和结束的时候同步，只是在逻辑上的先后。一个逻辑步中的 block 的总数可以理解一次可以同步执行的最大数量，但是这些 block 如果不同步执行也不会对结果产生影响，就是在一个逻辑时间上彼此独立的 block 的最大的数量。

那么假设第 i 超级步中的 block 数量是 ki，那么 block 的总数就是：$\sum_{i=0}^{i=Nsuperstep} k_i$

对于一个 block，编号 n，在超级步 Superstep（blockn）中执行，那么只能在它依赖的所有 block 执行完再工作。那么它的等待时间就是之前每一个超级步中 block 的最长时间（因为一个超级步中所有 block 可以并行），用 Tni 表示 i 层的 block n 的执行时间，那么等待时间就是 $\sum_{i=0}^{Superstep(blockn)} MAX(T_n^i)$。

那么总的等待时间就是关于超级步数量的函数就是每一个超级步等待开始的时间总和设这个时间是 Ttotal_wait_time，那么平均等待时间就可以表示为 Ttotal_wait_time/Nblock_num。

上面的分析是对处理器资源和调度能力都不受限制的情况。对于一般情况，分析如下对于 TB 中的 task queue 的数量是有限的，新模型中有两个衡量 TB 能力的参数，分别是深度(depth)和宽度(width)。对实际的情况的分析需要调度的理解:TUi 接收相应的 SLi，创建好上下文，然后判断接收的 kernel 是否可以进入相应的 TBi 的 FIFO task queue，如果相应的 task queue 没有达到相应的 depth 则可以进入该队列等候，TUi 中相应的 logical channel 被释放，准备接收新的 SLi。进入队列的 task 在一个队列中保持严格的顺序执行，队列之间可以无序。也就是说分配 task queue 的过程就已经完成了需要依赖的 block 之间的顺序执行，也就是说由于 task queue 的存在我们可以忽略那些 block 之间具有依赖。我们可以计算出所有所有 block 的串行的时间，然后减去最后 width 个 block 的执行时间，得到的就是总共的等待时间。然后除以 width，除以 block 的总数，就是一个 block 的平均等待时间。

MADOUGOU.S 等[12]对于 GPGPU 的性能分析模型进行了一个综述，将各种分析模型主要分成了以下几类：

分析模型，比如 PMAC 架构[13]、MVP-CWP[14]和部件模型[15]这一类主要分析硬件和任务，通过公式来预测核函数的执行周期；统计模型，比如 Eiger 架构[16]和 GPURandomForest[17]这一类模型主要用统计和机器学习的方法对待设计或者待优化的方法进行取样，找到对系统影响最大的特征，这种方法需要通常需要大量的人力物力；基于编译器的分析模型，这一类模型比如 WFG[18]，它融合了对程序的分析以及分析模型，即会预测执行时间，也会给出性能瓶颈。还有一些模型，比如 ScaleClass[19]和性能预测模型[20]都融合了几种方法

对 GPGPU 进行研究。

1.4.3 分析 GPU 结构的方法

WONG.H 等[21]相比较于前两篇看的定量分析的论文使用了 decuda 工具，可以深入到 native code level 对 machine code sequence 进行分析，得到了更加精确的结果。同时设计了一组很巧妙的测试程序来“解密”GPU 内部部件，找到了一些和 Programming Guide 有出入的地方，比如同步的粒度、对 SFU 的描述等。

定量分析模型应该与新模型的硬件模型思想保持一个，那么同样是一个分层模型。假设 level-i 的一个部件的执行时间为 Wlevel-i，则它的值与硬件模型的中的四个部分有关，即 SEi-1，PEi-1，TUi-1，TBi-1（内存 memory 隐含在 PEdi-1 的运行时间之中）。对计算指令和访存指令的执行时间的研究，也就是在分析 PE（包括 PEC 和 PED）的执行时间。

从执行流水线的角度建立模型时需要考虑的一些参数如下：Context allocation 过程中 TUi-1 给 SLi 创建环境；任务解 atomic 入 queue 是一个开销量，在 queue 中排到队头是一个开销量，这两个量显然与 Task Buffer 的中的 task queue 的 depth 和 width 有关。假设一个时钟周期能从走出 task queue 的 TI group 的数量从新模型角度来看就是 task queue 的 width，TI group 从队列中出来进入相应的 PE，如果没有达到能力上限，就可以进入 PE 等着执行。CUDA 也有相关的参数来描述这个性能。

Script Consumption 中 SEi-1 和 PEi-1 按照相应指令执行 SLi 的 PSi-1 和 SSi-1 在 PE 中走过流水是一个开销量：计算指令开销具体到某一个型号上面，与指令类型有关。

判断 PS（指令）执行时间需要的变量：thread 的数量、PE 的数量、指令的类型、指令的数量、GPU 核心的时钟频率 ，条件分支（本质上可以归类为计算开销）汇聚点。global memory 数据结合访问的 thread 的数量，一次全局存储器传输所需要的时钟周期数，访问间隔 stride。

Shared memory 访问冲突的路数，带宽，与 stride 有关。同时考虑 PE 队列和 SE 队列需要考虑 overlap 的问题，那就需要考虑访存和计算的比例 Context Deallocation 阶段 TUi-1 监视 SLi 的执行 SC 的情况，这一部分同样需要一个开销。

1.4.4 GPGPU 相关研究

对于 GPGPU 的研究，许多工作的目标在优化 global memory access 和实现尽可能多

的 thread-level parallelism。基于这一个目标有很多的编译框架[22-24]。Kepler 系列 GPU 支持了 Dynamic Parallelism，即在 GPU 端启动新的任务、同步结果、完成调度的过程都不在需要 CPU 的参与[25]。关于 Dynamic Parallelism 的研究也很多样。例如 DIMARCO.J 等人[26]给出了 Dynamic Parallelism 可以提高一些聚类算法的表现。文献[27] 中 YANG.Y 等人给出了解决 thread 内存在可并行代码的方法，并且希望可以用这种方法代替 CUDA 的 Dynamic Parallelism。NARASIMAN 等人和 PHOTHILIMTHANA 等人在文献[28, 29]给出了关于结构上的改进,ZAMBERNO 等[30]中提出了一个硬件方法，在运行时动态的生成 threads 来减少复杂的控制流的开销，这一方法与英伟达公司的 Dynamic Parallelism 十分类似。还有很多工作优化了应用[31, 32]。HONG 等[32]使用了一个 warp 负责一个工作来减少控制分歧。

第 2 章 NKGPGPU 模型描述

2.1 介绍

在过去的十几年里 GPU 的处理性能的增长十分迅猛。GPU 在结构上与 CPU 有很大的不同，GPU 的中更多的晶体管用于计算，而 CPU 中跟多的晶体管用于逻辑控制。因此在不同的设计目的之下，他们的作用也变得不同。更近一步，GPU 迅速从图像处理领域发展到通用计算领域，由此开启了一个新的领域叫做 General-Purpose Computing on the Graphic Processing Unit（GPGPU）。

对于 GPGPU 的研究，许多工作的目标在优化 global memory access 和实现尽可能多的 thread-level parallelism。基于这一个目标有很多的编译框架[22-24]。这一部分的研究主要基于现有硬件结构。

英伟达公司的 CUDA Toolkit 支持一种类似于 C 语言的编程语言，可以允许用户执行相应的 GPU 上执行代码。但是，有很多的涉及到递归以及数据依赖的应用并不能直接利用 GPU 的计算性能。在 Fermi 结构中，需要 CPU 端启动 GPU 任务，GPU 在把结果写回 CPU，CPU 再启动新的 GPU 任务，所以此类应用需要大量的重写才能运行在 Fermi 结构的 GPU 上。直到英伟达公司推出了开普勒（Kepler）系列的 GPU，这一问题才得到改善。Kepler 系列 GPU 支持了 Dynamic Parallelism，即在 GPU 端启动新的任务、同步结果、完成调度的过程都不在需要 CPU 的参与[25]。关于 Dynamic Parallelism 的研究也很多样。例如 DIMARCO 等[26]给出了 dynamic parallelism 可以提高一些聚类算法的表现。YANG 等[27]给出了解决 thread 内存在可并行代码的方法，并且希望可以用这种方法代替 CUDA 的 Dynamic Parallelism。NARASIMAN 等人和 PHOTHILIMTHANA 等人[28, 29]给出了关于结构上的改进，YANG 等提出了一个硬件方法，在运行时动态的生成 threads 来减少复杂的控制流的开销，这一方法与英伟达公司的 dynamic parallelism 十分类似。还有很多工作优化了应用[31, 32]。HONG 等[32]使用了一个 warp 负责一个工作来减少控制分歧。

但是 Dynamic Parallelism 还有两个主要的问题。第一是父子 kernel 的通信必须通过 global memory；第二是在 GPU 端启动 kernel 的开销十分巨大[27]。官方给出的白皮书更多的实在描述一个 GPU 的计算能力以及它能做到的事情，而不是部件之间如何相互协同作用。编程指南给出的是代码的语法和规则，而没有给出是什么样的微结构设计导致了这些规则。由于没有对硬件细节的详细描述，使得研究人员和开发人员不能充分发掘硬件

的潜力,进而提升应用的性能。

一些传统的定量分析模型[4-6]都不能很好的描述 GPGPU 的模型。本文给出的模型完整的描述了 GPGPU 运行机制，以及一个可以预测分析的模型。该模型展示了一种新型的硬件部件和内部的操作的机制，主要工作总结在下面：我们提出了一个适用于嵌套 kernel 的 NKGPGPU 模型，这个模型包括的主要方面有 task organization，hardware structure，scheduling mechanism 和 execution mechanism，其中在 task organization 给出了多个任务之间的数据结构。最后通过借鉴 KOTHAPALLI 等[11]的思路，给出了性能分析模型。

NKGPGPU 模型包括了 4 个主要的子模型，分别是硬件结构模型、任务组织子模型、任务执行子模型、任务调度子模型。描述模型主要用于解释在 NKGPGPU 在结构上的设计，以及在运行时的主要原则。

2.2 硬件结构模型（Hardware-Structure Submodel）

在硬件模型的设计过程中，我们总结出了需要在芯片设计和软件开发、以及两者交互过程中需要考虑的主要部件。NKGPGPU 的硬件结构子模型如图 2.1 所示，其中的每一个部件我们都会给出相应的解释。

Element for Computing(PEc):用来执行计算任务的部件。

Processing Unit for Computing(PUc)：用来执行计算任务的单元，是 PEc 部件的一部分。

Processing Element for data movement(PEc)：用来执行数据转移任务的部件。

Processing Unit for data movement(PUc)：用来执行数据转移的单元，是 PEd 部件的一部分。

Secheduling Element（SE)：负责任务调度的部件。

Task Unit（TU)：用来接收新任务的部件。

Buffer：用来存储状态信息的部件。

在存储体系的抽象过程中，先不考虑各级 cache 的存在及影响。图中每一个 PUc 和 PUd 部件拥有自己独立的 local memory。Shared memory 的可以被所有的 PEc 和 PEd 部件访问和修改，global memory 只可以被 PEd 部件修改和访问。具体来说，PEd 部件负责的是数据在 global memory 和 shared memory 之间的数据转移。

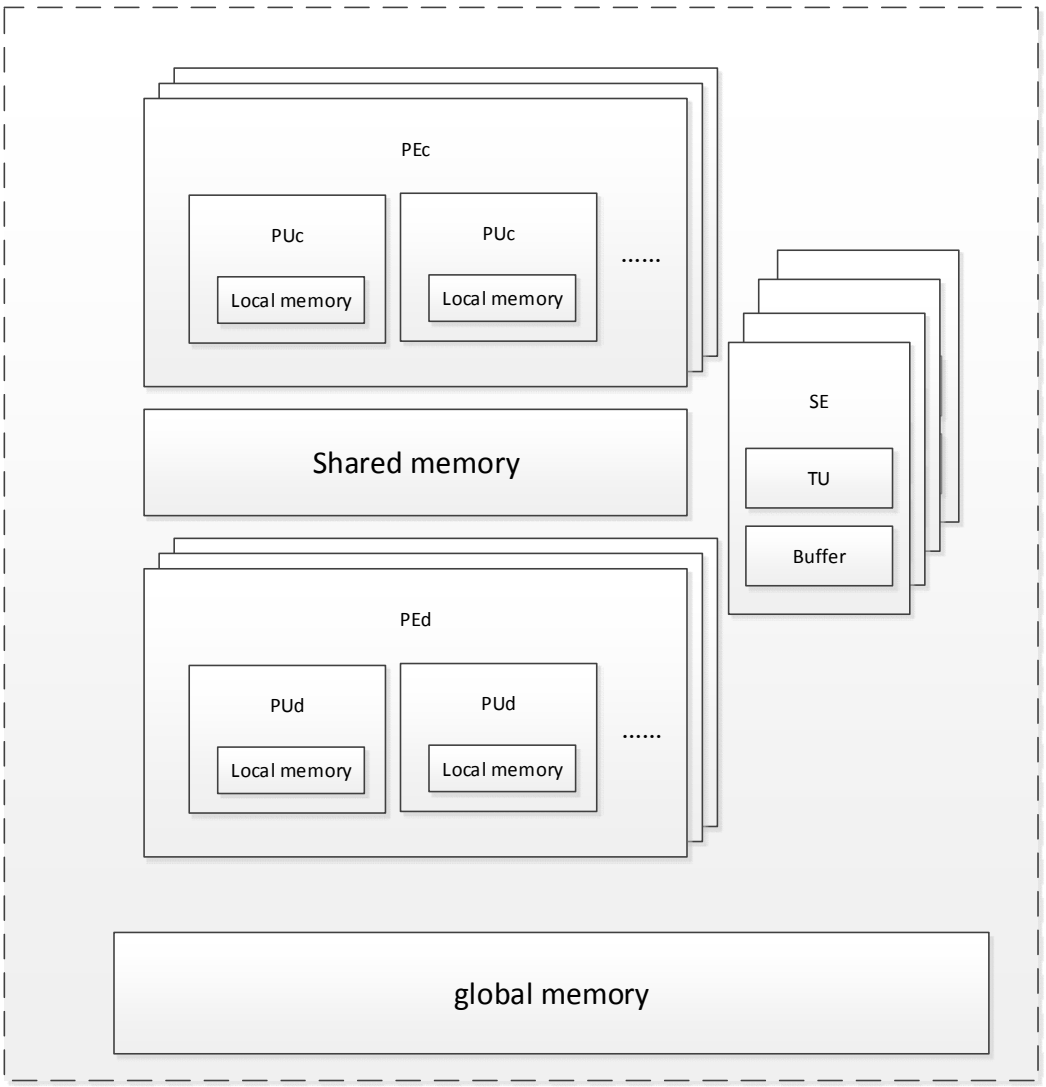

图 2. 1 硬件模型

2.3 任务组织子模型

胡亮等[3]提出了一个扁平化的多层的代码结构，我们借鉴了其中的部分定义。并且根据 NKGPGPU 模型修改了相关的概念，提出了一个 3 层的代码模型，这些概念在下面已经给出：

Processing Script for Computing(PSc)：负责计算任务处理的代码。

Processing Script for Data movement（PSd)：负责数据转移处理的代码。

Scheduling Script for Computing（SSc)：负责计算任务调度的代码。

Scheduling Script for Data movement(SSd)：负责数据转移任务调度的代码。

Script List for Computing(SLC):包含了所有的关于并行计算任务的信息的一段代码。一个 SLC 就是一列的 PSC 和 SSC。

Script List for Data movement(SLD) 包含了所有的关于并行计算任务的信息的一段代码。一个 SLC 就是一列的 PSD 和 SSD。

Script List for Program(SLP) 包含了所有的关于并行任务的所有信息的一段代码。一个 SLC 就是一列的 SLC 和 SLD。

在任务的逻辑结构上，借鉴 CUDA 的任务体系，目前设计为一个 3 层的任务结构。在下面给出最小粒度任务的描述。

Computing Task Instance(CTI)：一个带有独一无二索引的 SLC 运行实例。每一个 SLC 实例化一组 CTI,这个数量就是 SLC 的 task width。一个组内的 CTI 共享同一个 SLC，但是运行在基于 CTI 索引位置的内存地址上。

Data movement Task Instance(DTI)：一个带有独一无二索引的 SLD 运行实例。每一个 SLD 实例化一组 DTI，这个数量就是 SLD 的 task width。一个组内的 DTI 共享同一个 SLD，但是运行在基于 DTI 索引位置的内存地址上。

任务组织实际上就是一个代码结构与任务结构的映射，其对应关系如图 2.2 所示。每一个 SLD 会生成一个 DTI Grid，每一个 DTI Grid 包含了若干个 DTI Block，每一个 DTI Block 中包含了若干了 DTI。DTI 在 DTI block 中被分成 DTI warp 执行。每一个 SLC 会生成一个 CTI Grid，每一个 CTI Grid 包含了若干个 CTI Block，每一个 DTI Block 中包含了若干了 CTI。CTI 在 CTI block 中被分成 DTI warp 执行这样一个 SLP 就会实例化若干个 DTI Grid 和 CTI Grid。

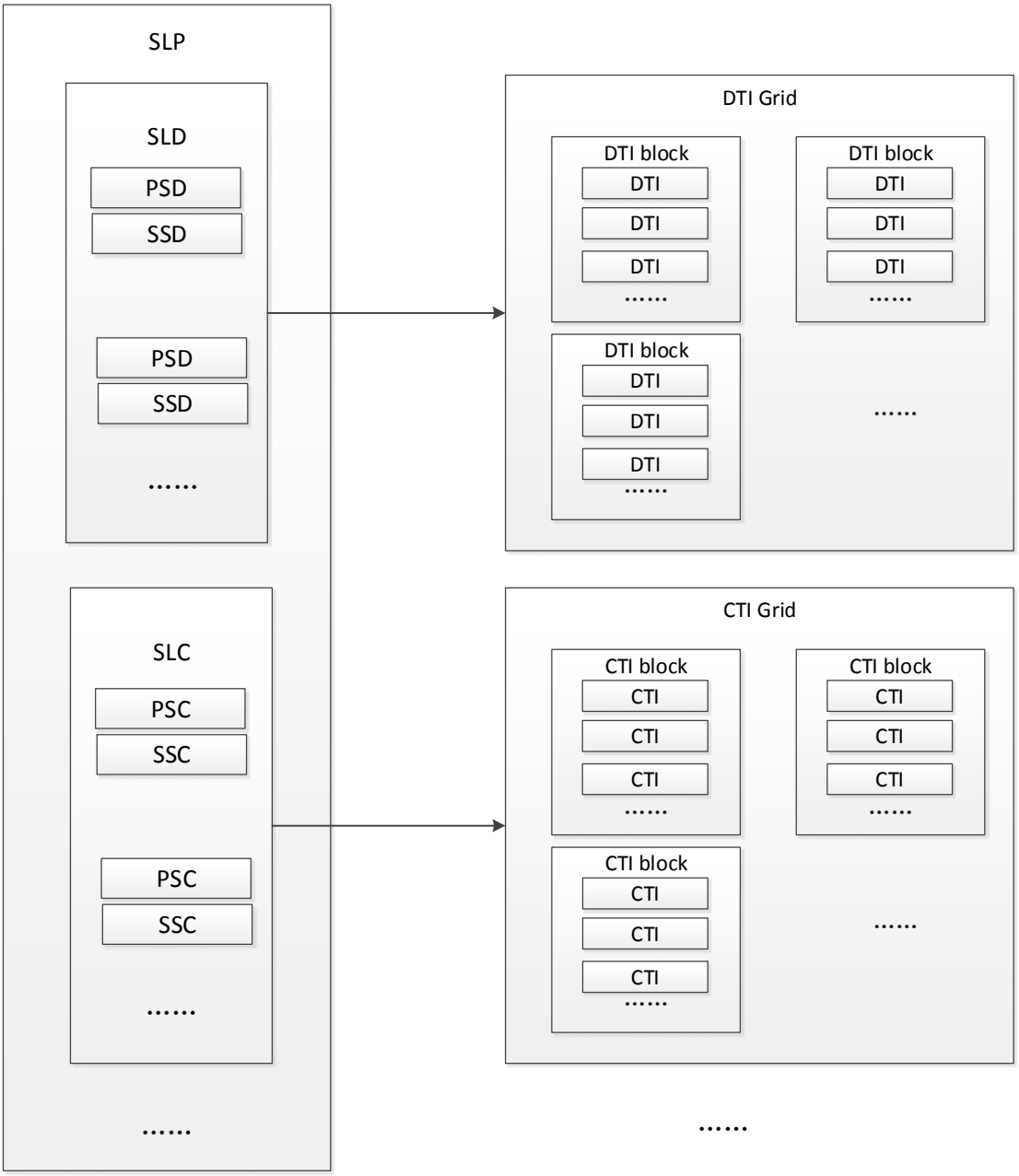

图 2.2 任务组织图

在一个 SLP（可以理解为 CUDA 编程模型中的 kernel）中，如果启动了新的 SLP，我们就称启动新 SLP 的 SLP 为父 SLP，被启动的 SLP 为子 SLP。由 SLP 之间的启动关系就构成了一棵启动树，树结构如图 2.3 所示，树中每一个节点表示一个 SLP，图中根节点为 Host 端，现有 GPU 还不能完全脱离 Host 端，所以这一设计是为了继承现有的产品，在以后可以扩展为 GPU 端的根节点。图 2.3 中，SLP A 启动了 SLP B、SLP C 、SLPD ，所以 SLP A 是 SLP B、SLP C 、SLPD 的父节点，SLP B、SLPC 、SLPD 是 SLP A 的子节点。SLP C 启动了 SLPE、SLP F 、SLP G ，所以 SLP C 是 SLP E、SLP F 、SLP G 的父节点，SLP E、SLP F 、SLP G 是 SLP C 的子节点。

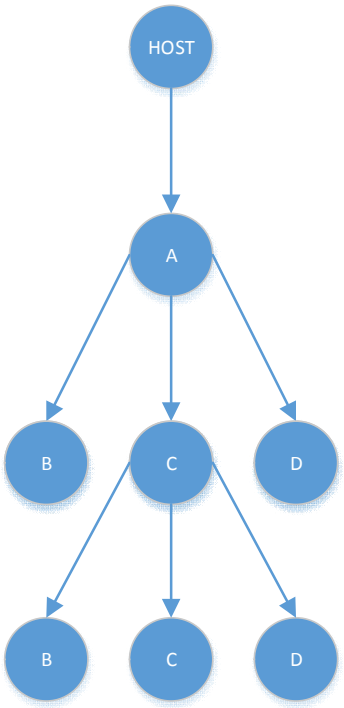

图 2.3 启动树

然而在启动关系树中，某一父节点的子节点之间的依赖关系需要进一步确认。我们把某一父节点以及其子节点之间的关系抽象成一个依赖关系树。在依赖树中，所以父节点的完成依赖于节点的完成，兄弟节点之间可以完全并行没有依赖。

父节点 SLP A（简称为 A）与子节点 SLP B、SLP C 、SLP D（简称为 B,C,D）之间的关系可以抽象成图 2.4 中的关系。

图 2-4 表示的是 BCD 之间没有依赖关系可以完全并行；图 2.4-B 表示 BD 之间没有依赖，B 需要依赖于 C，图 2.4-C 表示 CD 之间没有依赖，B 需要依赖于 CD；图 2.4-D 表示 BCD 之间都有依赖关系 B 依赖于 C,C 依赖于 D。

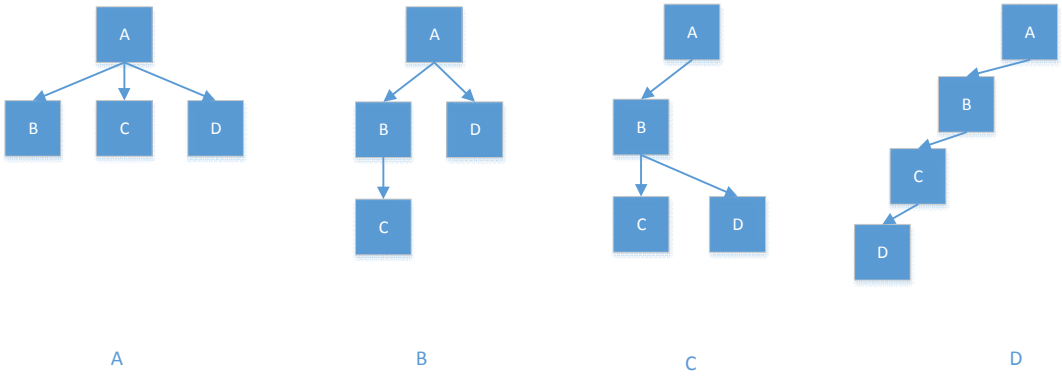

图 2-4 兄弟节点依赖关系图

对于每一个包含有子节点的父节点都会根据子节点之间的依赖关系，所有这些依赖

关系树合并在一起就会生成一棵完整的依赖树。图 2.5 给出一个例子。BCD 的依赖关系如图 2.5-A，EFG 之间的关系如图 2.5-B，合并的结果如图 2.5-C。以图 2.5 为例，在合并两棵依赖树时，只需要把 C 的子节点按照原有结构加入 A 的树中。

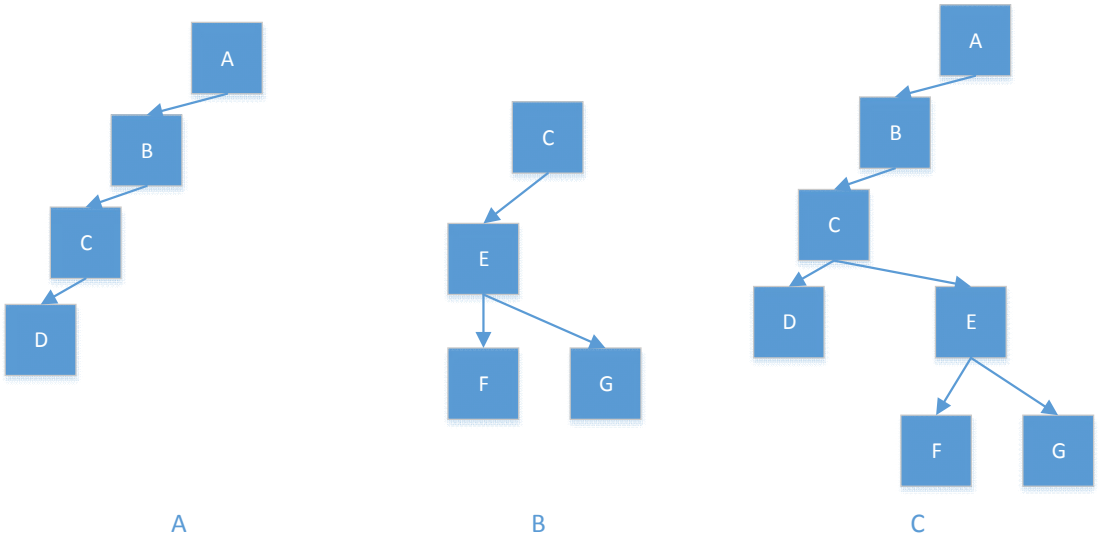

图 2.5 依赖树的合并

在依赖树生成之后就可以确定执行的顺序。按照层次遍历的顺序遍历依赖树，每遇到一个叶子节点，就把该节点加入到执行队列，并从依赖树中删除，重复这个过程知道依赖树中只剩下 host 节点。

2.4 任务执行子模型

任务执行子模型是一个从代码到硬件的映射，如图 2.6 所示。

PSc Consumption：逻辑上一组平行的 CTI 分配到一组可用的 PEc 上执行。可用表示 PEc 有足够的能力来接收一组新的 CTI。物理上，相关的 PSc 会被 SEc 部件解析，并且把相关信息存在 TB 部件中。

PSd Consumption：逻辑上一组平行的 CTI 分配到一组可用的 PEc 上执行。可用表示 PEc 有足够的能力来接收一组新的 CTI。物理上，相关的 PSc 会被 SE 部件中的 TU 解析，并且把相关信息存在 TB 部件中。

SS Consumption：SSd 和 SSc 都会在 SE 部件中进行执行。

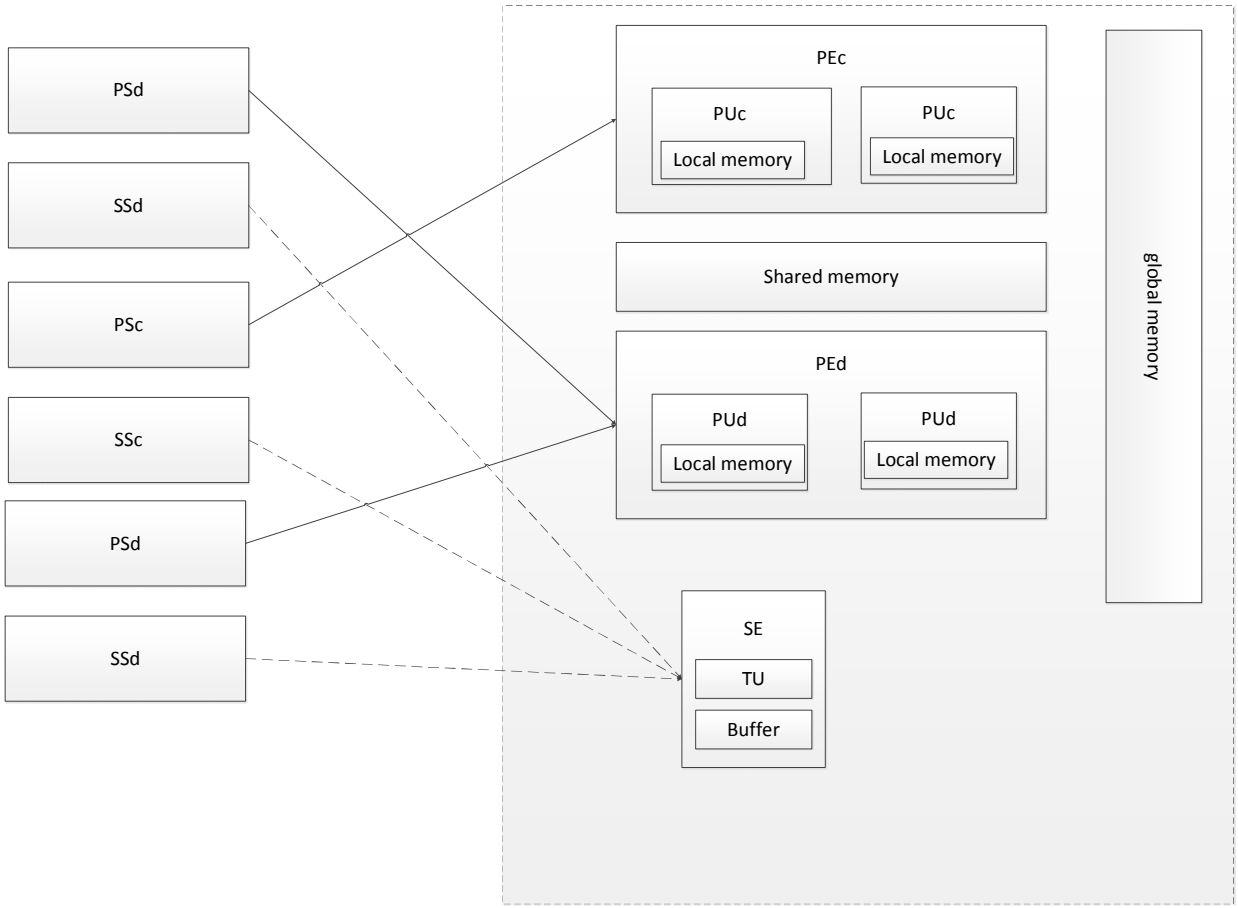

图 2.6 任务执行子模型

2.5 任务调度子模型

如图 2.7 所示，任务调度模型是一个任务拓扑结构到硬件结构的对应。多个任务组会分配到一个 PEc 或者 PEd 部件上，因此 PEc 或者 PEd 要保证这些任务顺序执行或者并行执行。为了保证依赖任务的顺序执行以及无依赖任务的并行，任务调度子模型将支持下面的特点：

Multiple Working Channels.每一个 TU 部件中包含多个 logical channel。每一个任务组会分配一个 channel 编号，任务是按照严格的顺序通过 channel 的。

Multiple FIFO Queues.每一个 TB 部件是多个 FIFO 队列。每一个队列与一个 channel 编号相对应。在一个队列中任务组通过队列是按照严格的先后顺序，但是多个队列之间是乱序的。

Intrinsic Synchronization.任务组的任务可能会异步执行，但是在任务组的开始和结束是需要同步的。

Hardware Multithreading.顺序执行机制保证了有依赖的任务之间不会叠，这也使得所有分配到 PEc 或者 PEd 的任务彼此独立。硬件可以通过多线程的方式在任务组之间进

行切换，使得硬件的占用率更高。

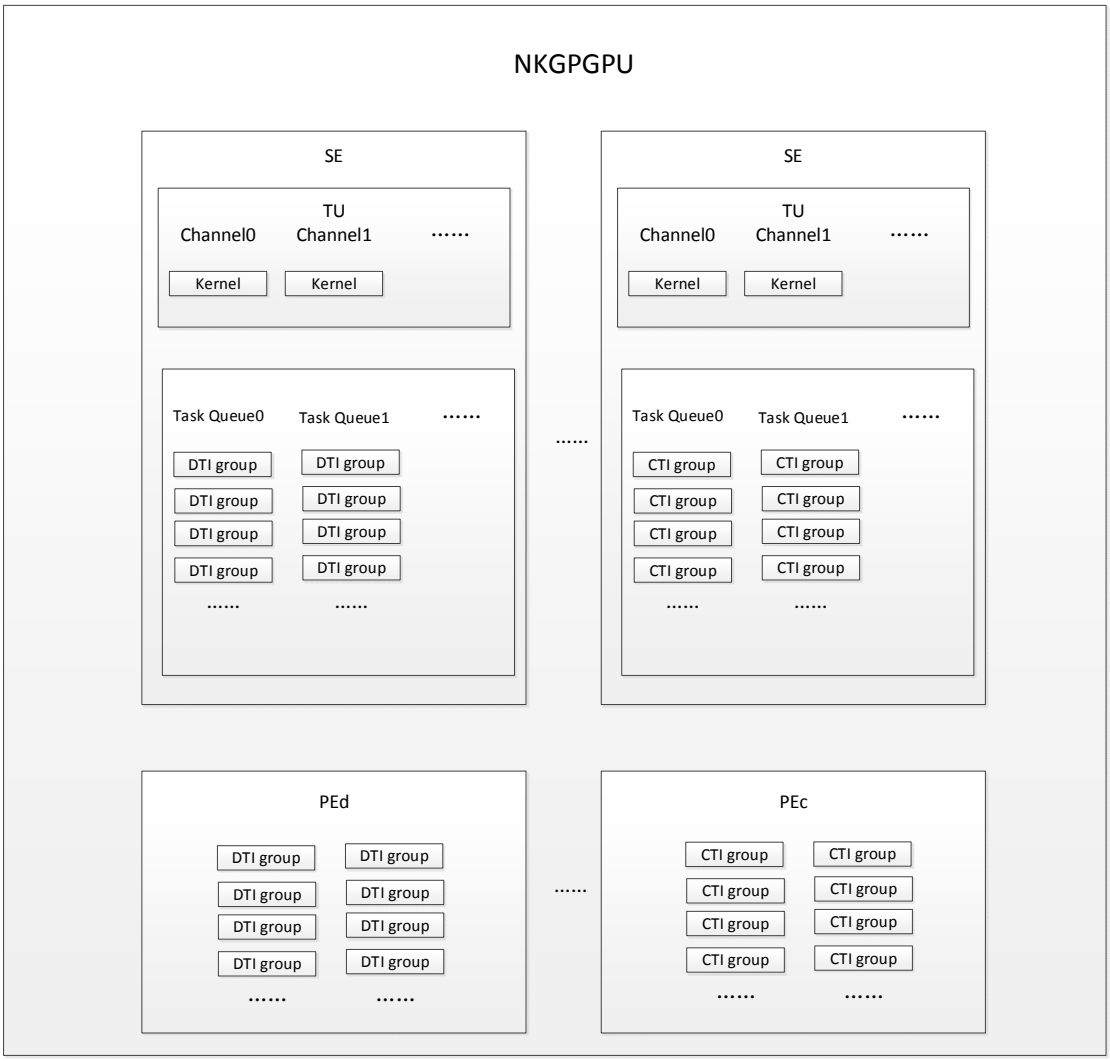

图 2.7 任务调度模型

第 3 章 NKGPGPU 性能分析模型

3.1 单个 warp 的性能分析

执行时间预测模型采用 PTX 语言作为输入，模型根据指令的调度和延迟来预测执行时间。首先指令之间主要有两种的依赖，第一种同一个 warp 的指令之间执行的间隔为 6 个周期。第二种是数据依赖，在一个 warp 有依赖的指令之间调度间隔为 18 个周期，这一部分会在实验中详细解释。

GPGPU 的执行以流水线的方式进行的执行，每两个周期可以调度一个 warp。根据 GPGPU-Sim 中的流水线结构，我们可以把流水线分成两个阶段。在 SIMT 前端，指令完成取指译码以及检查依赖的过程，在 SIMD 后端，指令完成操作数的读取和执行。同一个 warp 的下一条指令需要 6 个周期之后再调度，所以 SIMT 前端的延迟可以理解为 6 个周期。图 3.1 表示了这一过程。

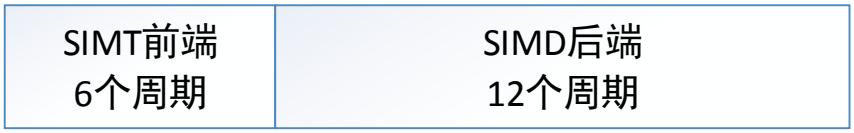

图 3.1 流水线周期表示

如果是有指令依赖的情况，下一条指令需要等待之前指令完成 SIMT 前端的过程才能开始。这一情况如图 3.2 所示。

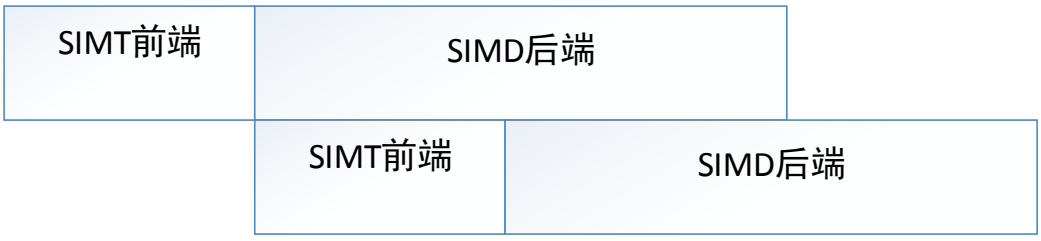

图 3.2 含有指令依赖的流水线表示

如果是一个数据依赖的情况，那么下一条指令必须等到上一条指令在 SIMD 后端完成才能开始执行，情况如图 3.3 所示。

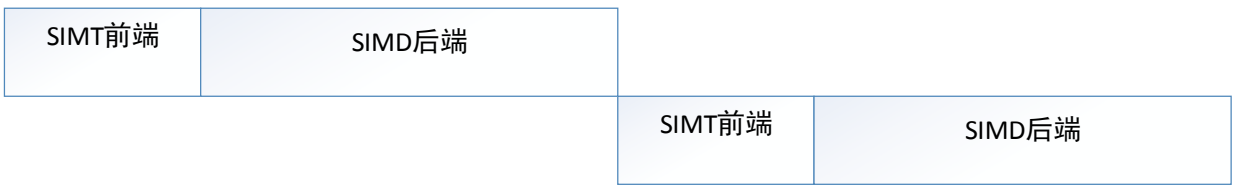

图 3.3 含有数据依赖的流水线表示

在我们有多个 warp 需要调度时，会用到两个调度器，奇数调度器和偶数调度器。举例来看，假如一个偶数调度器的情况如下图所示，由于是不同 warp 的指令，所以，指令之间没有依赖，所以每个两个周期可以调度一个 warp 的指令。这种情况如图 3.4 所示。

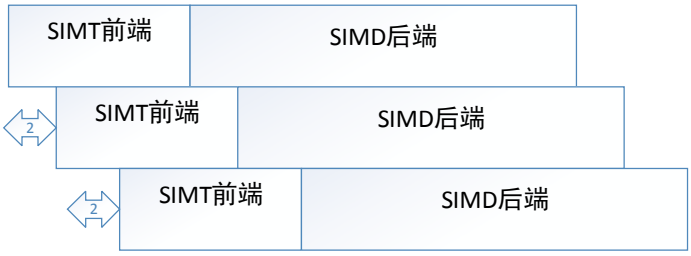

图 3.4 多个 warp 指令的流水线表示

图 3.5 给出了一个测试程序和它的时间线。最开始 S2R 指令开始执行，由于 SHL 指令和 S2R 指令有 RaW 依赖，SHL 指令只能在第 18 个周期开始执行。ADD 指令有指令依赖，所以只能在第 24 个周期开始执行。第二条 ADD 指令在第 30 个周期执行，S2R 指令再第 36 个周期执行。然后 SHL 指令由于 RaW 指令依赖，只能等到第 54 个周期开始执行。

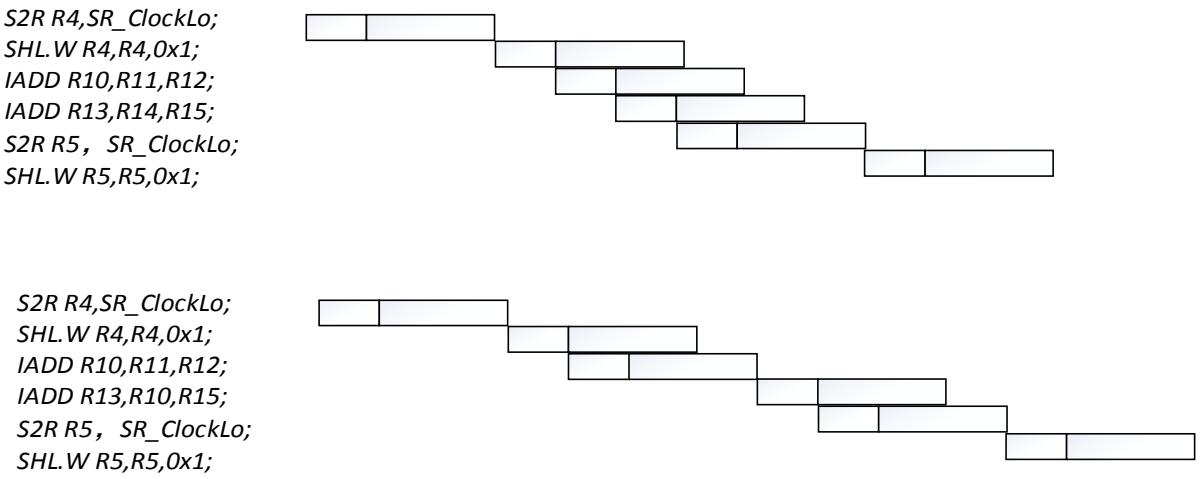

图 3.5 测试程序和流水线对应

根据实验对比，两条没有 RaW 依赖的测试程序的执行周期 36 和预测结果相同；两条有 RaW 依赖的执行周期为 48 个周期。

3.2 NKGPGPU 整体程序性能分析

文献[11]给出了一个 CUDA GPGPU 的性能预测模型，可以用来预测执行时间。本文参考[11]中的参数列表，做了一些修改来适应于我们的模型，具体参数如表 3.1 所示。下划线“-”表示下角标，次方符号“^”表示上角标。

表 3.1 参数列表

DTI 相关参数	
参数	定义
N_B^D(SLD)	一个 SLD 中分配到一个 PE_D 部件上的 DTI block 数量
N_w^D	每一个 DTI Block 中的 DTI warp 的数量
N_t^D	每一个 DTI warp 中的 DTI 的数量
D^D	PE_D 的流水线深度
N_c^D	每一个 PE_D 的 PU_D 的数量
C_T(SLD)	max 模型中表示一个 SLD 中某一 DTI 需要的最多的周期数
C(SLD)	一个 SLD 需要的周期数
CTI 相关参数	
参数	定义
N_w^C	每一个 CTI Block 中的 CTI warp 的数量
N_t^C	每一个 CTI warp 中的 CTI 的数量
D^C	PE_E 的流水线深度
N_c^C	每一个芯片上的 PE_C 的数量
C_T(SLC)	max 模型中表示一个 SLC 中某一 CTI 需要的最多的周期数
	sum 模型中表示一个 SLC 中所有的 CTI 运行周期的总和
(C(SLC)	一个 SLC 需要的周期数
SLP 相关参数	
C_SLD(SLP)	MAX 模型中表示一个 SLP 中某一 SLD 需要的最多周期数
	SUM 模型中表示一个 SLP 种所有的 SLD 需要的周期数
C_SLC(SLP)	MAX 模型中表示一个 SLP 中某一 SLC 需要的最多周期数
	SUM 模型中表示一个 SLP 种所有的 SLC 需要的周期数
C(SLP)	一个 SLP 需要的周期数
C_Total SLP	一颗依赖树中所有的 SLP 需要的运行周期
L	依赖树的层数

公式（1）和公式（2）分别给出了一个 SLD 和一个 SLC 需要执行的周期数的计算公

式。在公式（1）中的C_T (SLD)是一个 DTI 的代表量，公式中用所有 DTI 中的最大值表示。同理公式（2）中C_T (SLC)是一个 CTI 的代表量，公式中用所有 CTI 中的最大值表示。

$$C(SLD) = N_B^D(SLD) \cdot N_W^D(SLD) \cdot N_T^D(SLD) \cdot C_T\,(SLD) \cdot \frac{1}{N_C^D \cdot D^D} \quad \cdots\cdots(1)$$

$$C(SLC) = N_B^C(SLC) \cdot N_W^C(SLC) \cdot N_T^C(SLC) \cdot C_T\,(SLC) \cdot \frac{1}{N_C^C \cdot D^C} \cdots\cdots(2)$$

公式（3）给出了计算 SLP 的所需要的周期数。通过最好的调度可以实现C(SLD)和C(SLC)的相互隐藏，在这种情况下C(SLP)就可以通过 MAX 模型获得，表示为公式（3）;假如调度完全没起到作用，则C(SLP)就可以通过 SUM 模型获得，表示为公式（4）。

$$C(SLP) = MAX(C(SLD), C(SLC)) \cdots\cdots(3)$$

$$C(SLP) = SUM(C(SLD), C(SLC)) \cdots\cdots(4)$$

一颗依赖树中的所有 SLP 需要的周期表示成公式（5），对于依赖树中每一层中的所有节点可以并行，对于这些节点选择其中需要周期数最长的 SLP 作为这一层 SLP 需要的运行周期数。对于一个依赖树中的某一父节点，我们认为该节点需要等该他的孩子节点都完成才可以开始。因此某一个父节点的执行时间就是该节点的需要的周期数与它孩子节点周期数的和。

$$C_{Total}^{SLC} = \sum_{i=0}^{L} MAX\big(C(SLP_i)\big)\ , SLP_i\ \epsilon\ \text{level i} \cdots\cdots（5）$$

KOTHAPALL 等[11]中提出的性能预测模型，将一个 thread 的运行周期分成了计算周期和访存周期的。计算周期用 Ncomp 表示，访存消耗周期用 Nmemory 表示。在一个最好的调度情况下，计算任务和访存任务可以相互隐藏，则一个 thread 的运行周期为 C(T)=MAX(Ncomp,Nmemory)；如果调度没有起到作用，则 C(T)=SUM(Ncomp,Nmemory)。

KOTHAPALL 等[11]中提出的性能模型将存储和计算分开，与 NKGPGPU 中 PEd 和 PEc 的设计基本吻合。不同之处在于硬件结构和对应于硬件结构的任务结构。在 CUDA 的硬件结构体系中，计算和访存的任务都会交给 thread 来承担，而在 NKGPGPU 中计算任务会实例化 CTI，内存任务会实例化 DTI。一个 thread 在逻辑上完成的主要是三个任务准备数据，这一过程对应某一 DTI；处理数据，这一过程对应于 CTI、写回数据这一过程对应于 DTI，CUDA 由一个 thread 完成的任务，现在会分配给不同 DTI 和 CTI 去执行。

但是对于执行周期方面，在不考虑同步的情况下，还是可以采用 MAX/SUM 模型，MAX 模型表示 PEc 和 PEd 部件的任务之间没有依赖关系可以完全并行，SUM 模型表示 PEc 和 PEd 部件的任务之间有严格的顺序关系。Ncomp 在 NKGPGPU 中可以理解为 PEc 部件的执行

时间，Nmemory 表示 PEd 部件的执行时间。

KOTHAPALL 等[11]中对于一个 thread 的执行周期部分的分析，包括对各种计算指令以及访存指令所需要的周期的分析在本文中仍然使用，文中没有考虑 cache 和同步影响，在本文中也不做考虑。例如 CUDA 中一个大小为 N*N 的矩阵乘法的例子，设 block 的大小为 16*16 一个 thread 需要完成的计算操的周期作为 760N/16，需要完成的访存操作是 240N/16。在修改的性能模型中，设 CTI Block 和 DTI Block 的大小为 16*16，则$C_T(SLD) = 240N/16$, $C_T(SLC) = 760N/16$。公式中的其余部分也都按照 KOTHAPALL 等[11]的设置，可以得到对于 kernel，也就是本文中C(SLP)的分析。一个依赖树中所有 SLP 的分析也可以通过这一结果得到，在这里就不再赘述。

第 4 章 定量分析 GPGPU 结构中的时间变量实验

4.1 实验理论

为了能在低级别分析 GPU 的硬件行为，需要对于 GPGPU 硬件十分细节和准确的相关信息。然而由于这一部分涉及到专利和商业机密，所以很难获得，或者只有一个模糊的介绍。因此，对于时间的预测就变得十分困难。

本文给出的 NKGPGPU 中的代码结构与任务与当下的 GPGPU 并不完全一致，所以实验部分希望可以通过研究当下 GPGPU 的行为来近似模拟 NKGPGPU 的行为。

由于真正的 GPU 的细节难以获得，所以我们希望通过实验来分析 GPU 行为。本章节中的实验是为了研究 GPU 中不好分析的部分。实验结果通过基准程序来获得。在 GPGPU-Sim 中，我们的基准程序使用 PTX 语言和 PTXPlus 语言编写。PTX 语言虽然不是真正的机器语言，但是具有一定的通用性，并且可读性也更好。PTXPlus 是 GPGPU-Sim 团队对 PTX 语言的扩充，与 SASS 语言接近 1:1，PTXPlus 语言相较于 PTX 语言对于实验来说更加精确。

在 PTX 语言中，有一个专用计数器%clock，用来读时钟周期计数器。在硬件设备中，这个专用计数器叫 SR1。在 GPGPU-Sim 中 SR1 与 SIMT 核心频率一样，即每 SIMT 每过一个周期，SR1 就相应增加一个周期。

在 PTXPlus 语言中，英伟达的编译器会生成如下的指令来获得%clock。下面的指令对%clock 乘了 2。如果是 PTX 语言则需要下面的一条指令。

```
//PTXPlus accessing clock register
mov  %r1,  %clock
shl  %r1,  %r1, 0x1
```

图 4.1 PTXPlus 时钟周期指令

如果是 PTX 语言则只需要下面的一条指令。

```
//PTX accessing clock register
mov r1,  %clock
```

图 4.2 PTX 时钟周期指令

对于与时钟周期有关的结果一定要考虑 PTX 和 PTXPlus 之间的区别。下面给出一个 PTX 语言来计时的例子。

```
12.  asm (
13.      "{\n\t"
14.        ".reg .u32 time1;\n\t"
15.        ".reg .u32 time2;\n\t"
16.        //Register declaration
17.
18.        "mov.u32 time1, %clock;\n\t"
19.        "shl.b32 time1, time1, 1;\n\t"
20.
21.        //code to be measured
22.
23.        "mov.u32 time2, %clock;\n\t"
24.        "shl.b32 time2, time2, 1;\n\t"
25.
26.        "sub.u32    %0, time2, time1;\n\t"
27.        "}"
28.      : "=r"(time_res)
29.    );
```

图 4.3 使用 PTX 语言计时的例子

4.2 算数指令实验

算术指令对应于 NKGPGPU 的 Processing Script forComputing(PSc)，即负责计算任务的处理代码。由于算术指令在 PEc 部件完成，所以这一部分的研究主要是在研究 PEc 部件的行为。

对于算数指令操作，我们选择加法 ADD，乘法 MUL，余弦 COS，乘加 MAD。对于基准程序的设置为 32 个 Thread，所以只有一个 warp，一个 block。首先来看没有指令依赖的情况。下表给出了基准程序的运行周期，从表 4.1 中可以看出所有的指令需要的周期数是相同的，这就说明为了满足需求周期最大的指令把所有指令的延迟都做了修改。这样做的好处是可以让流水线的设计变得更加简单，进而可以提高吞吐量。除此之外，第一条指令的执行周期是 30，这是因为还需要把测试指令考虑在内。两条指令之间的间隔是 6 个周期，说明每条指令需要的译码、调度等过程至少需要 6 个周期，所以每 6 个周期开始一条新指令。

表 4.1 没有依赖情况下的算术指令执行周期

指令数量	ADD	MUL	COS	MAD
1	30	30	30	30
2	36	36	36	36
3	42	42	42	42
4	48	48	48	48
5	54	54	54	54

表 4.2 展示了在存在 RaW 依赖时指令所需要的执行周期。第一行和上一个实验的指

令相同，都是一条指令。对于下一条指令，则会发现有一个延迟，ADD，MUL,MAD 指令的延迟是 18 个周期。所以对于等待依赖的周期是 18-6=12 个周期。对于 COS 指令的数据依赖等待的周期则更长，是 22-6=16 个周期。COS 指令是在 SFU 单元进行的运算。

表 4.2 RaW 依赖的算术指令执行单元

指令数量	MUL ADD	MAD ADD	COS ADD
2	36	36	36
4	48	48	48
3	60	60	60
4	72	72	72

SFU 是一个更加复杂的功能单元，用来处理一些三角函数，例如 cosine 和 sine。SFU 的具体流水线实现并不清楚，但是本文可以假设 SFU 具有更深的流水线，所以数据依赖会影响流水线的更多的阶段。

表 4.3 中给出了混合这些指令的执行周期，分别是 MUL 和 ADD 的指令的组合，MAD 和 ADD 指令的组合，COS 和 ADD 指令的组合。通过这个实验本文验证指令之间是否是可以组合。在这个实验中，指令之间是没有任何依赖的。这个实验用于证明不同功能部件之间,也就是 SP 和 SFU 之间的关联会不会影响执行时间。从结果上可以看出，这个实验的结果和和两条的相同指令的结果是一样的。

表 4.3 无依赖混合指令执行周期

指令数量	MUL ADD	MAD ADD	COS ADD
2	36	36	36
4	48	48	48
3	60	60	60
4	72	72	72

下面的表 4.4 展示了具有数据依赖（这里还是指 RaW）的指令的结果。举例来说，假如有 4 条指令，就会有 3 个数据依赖问题。结果表示这个实验的结果和同一指令的数据依赖情况是一样的。本文主要关注常用的指令，对于在 SFU 上运行的指令不会做过多细节的实验。

表 4.4 RaW 依赖下算术指令执行单元

指令数量	MUL ADD	MAD ADD	COS ADD
2	48	48	52
4	84	86	92
3	120	122	132
4	156	160	172

4.3 Warp 调度器

Warp 调度器对应于 NKGPGPU 中的 SE 部件。

在硬件中，很难真实的去判断调度器中到底发生了什么。所以本文选择用 GPGPU-Sim 在每一周期追踪所有的指令。模拟器是确定的，并且每一次执行的结果应该是相同的。一些特殊情况本文暂且不做考虑，主要关注具有多数的情况。在每一个周期，本文对指令的执行进行监控，在 GPGPU-Sim 中 warp 的执行是基于 warpID 的。根据观测本文得出 warp 可能的三种状态。第一，调度状态：在这种状态下 GPU 可以执行不同 warp 的指令。调度器在检查得分板和 RaW 依赖后，每 2 个周期可以调度一个 warp 执行，也就是这种状态下 warp 可以找到可以调度的 warp。第二，由于 RaW 依赖导致的空闲状态：在这种状态下，调度器找不到可以执行的 warp，必须等待 RaW 依赖的解决，如果所有的 warp 都是平等的，并在严格的 RR 调度策略下，那么对于 RaW 依赖所有的 warp 都会等待相同的时间。第三，由于执行部件不可用而导致的空闲状态：调度器在调度新指令时会先检查 SP 和 SFU。如果执行单元没有做好准好准备是不流出指令的，例如在两个 warp 都要执行 LD/ST 单元，但是只有一个 LD/ST 单元，其中的一个 warp 必须进行等待。如果一个 warp 停滞了，那么它就需要从新去排队等待调度。

一个 warp 中的指令会按严格的顺序执行，在只有一个 warp 执行的情况下，warp 调度器只负责一个 warp 所以不需要进行调度。在多个 warp 的情况下，结果就会变得很复杂。Warp 之间是彼此独立的，所以他们可以乱序的进行动态调度，而不需要一个特定的顺序。调度器根据 warp 的状态进行动态选择执行。不同的调度策略会产生不同的调度时间。因此调度器的行为会影响对执行周期的预测。

前文的实验，每一个 block 只有一个 warp，即 32 个 thread。下面的实验将会测试不同 block 大小的执行时间，block 的大小分别是 32,64,128,256 和 512。纵坐标给出了执行周期的数量，横坐标给出了不同 block 大小的 warp ID。在没有控制分歧的情况下，一个 warp 中所有的 thread 执行的是同样的指令。图 4.4 展示了一条 ADD 指令在 block 包含的 thread 数量为 32,64,128，256 和 512 的情况下的执行周期。从图 4.4 可以看出，在一个 block 的 thread 数量小于等于 128 时，所有 warp 的执行时间是一样的。在 block 的 thread 数量为 256 和 512 时各个 warp 的的执行周期都变长了。在 warp 数量增加的情况下，需要更多的调度，所以需要更多的时间。

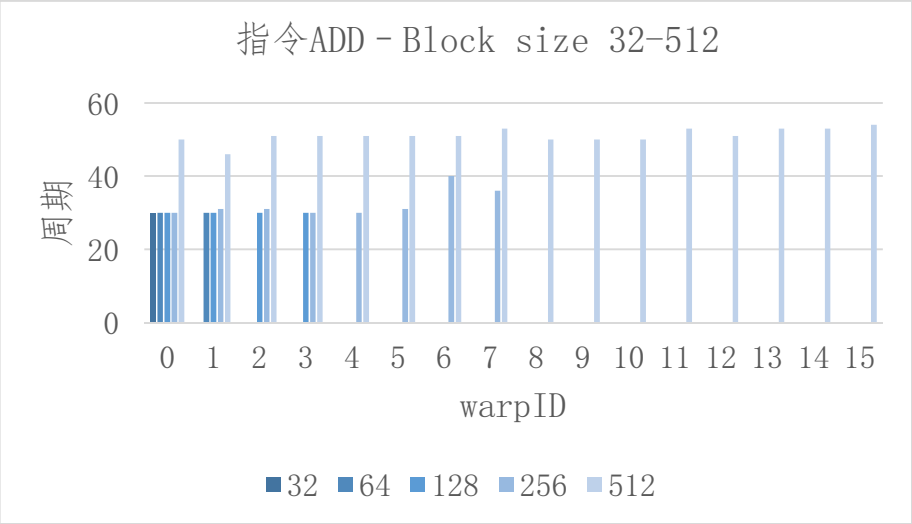

图 4.4 一条 ADD 指令在不同 block 大小的执行周期

本文执行了 8 次一个 block 包含 512 个 thread 的实验，为了验证这个过程是否是可以预测。

本文对 MAD 指令做了相似的实验，图 4.5 和图 4.6 两图展现了与 ADD 指令相似的实验结果。不同的是在一个 block 包含 256 和 512 个 thread 的情况下，每个 warp 的执行时间和和 ADD 指令的实验是不同的。但是由于这一部分缺乏相关文档描述，所以本文并不能对此区别做出相应的分析。

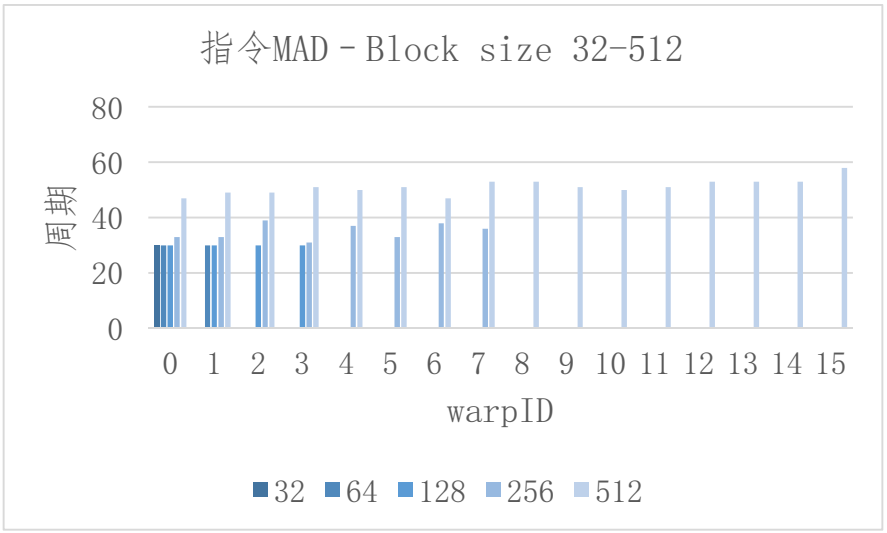

图 4.5 一条 MAD 指令在不同 block 大小的执行周期

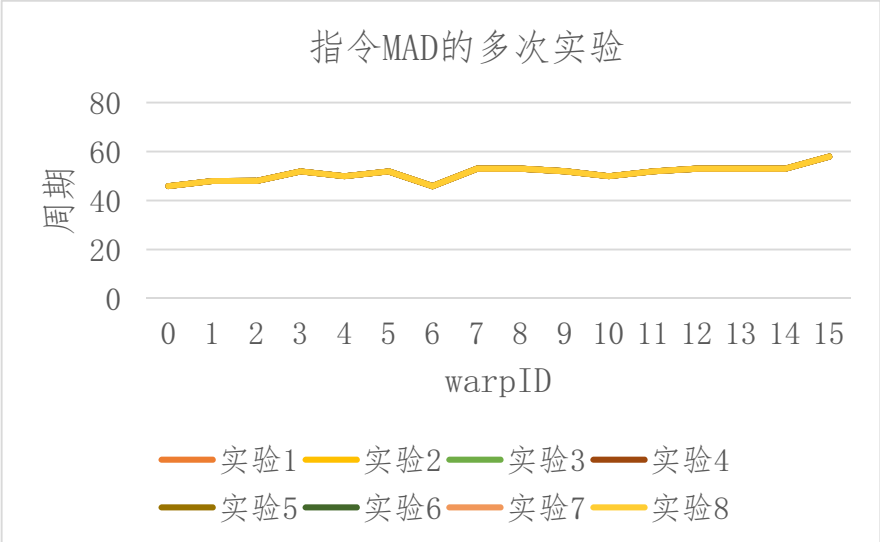

图 4.6 对 MAD 进行了 8 次重复试验

下一个实验使用了一条 MAD 指令和一条 ADD 指令，每个 block 包含的数量也同样是 32，64,128,256 和 512。图 4-7 展示了在没有数据依赖的情况下，在 128 个 thread 以下的实验结果和前面的实验很类似，在 256 和 512 个 thread 的情况下，warp 和 warp 之间的执行时间的与单个的 MAD 指令和 ADD 指令差距很大。

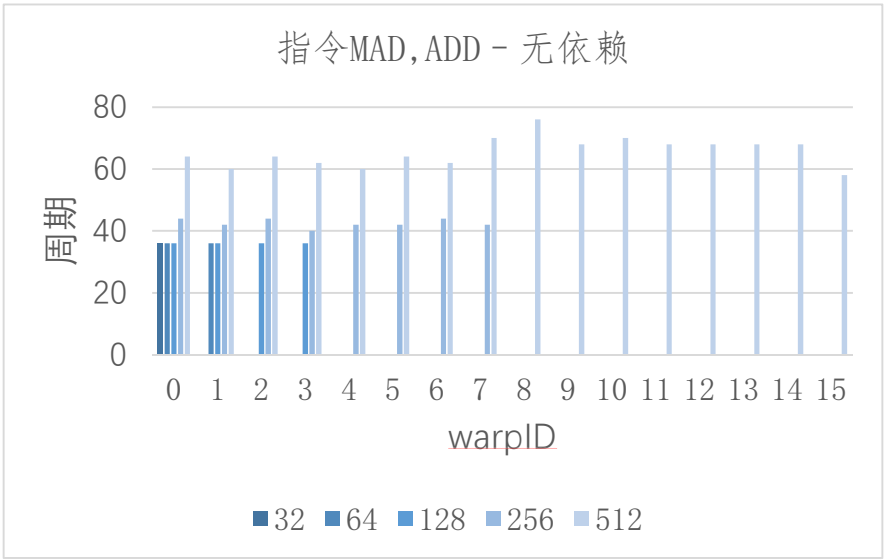

图 4.7 无依赖 MAD 和 ADD 指令的执行周期

同样本文还对不同的 512 个 thread 的情况进行了 8 次实验，来看一看两条指令的执行时间会不会在多次实验中有所改变。如图 4.8 所示，所有实验的执行时间相同，在多次实验中并没有观察到有不一样的执行时间。

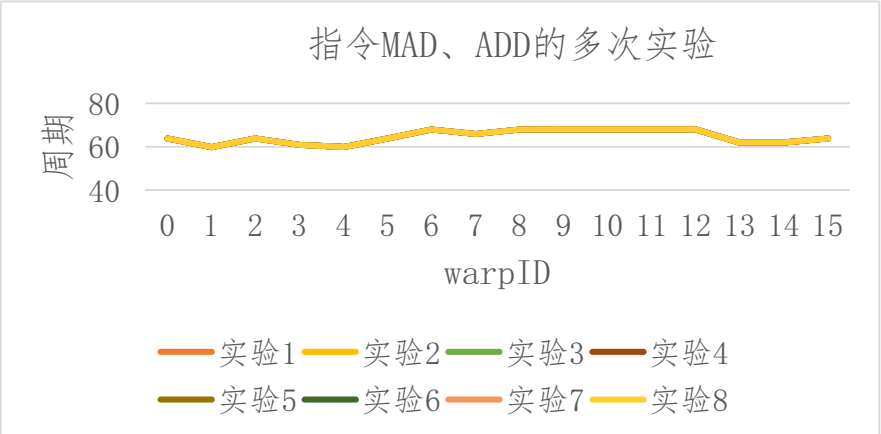

图 4.8 无依赖 MAD 和 ADD 指令的 8 次重复试验

下面的实验，本文对有数据依赖的 MAD 和 ADD 指令进行了实验，block 同样选取 6 个大小。同样在小于等于 128 个 thread 的情况下，每个 warp 的执行时间是相同的。整体结果如图 4.9 所示。每一个 warp 被调度的时间是两个周期，但是同一个 warp 的两条指令之间的调度需要 6 个周期。与此同时一个 warp 中的两个指令执行之间，其他两个 warp 的指令可以并行执行，并且不会增加执行时间，在 256 和 512 个 thread 的情况下所有的 warp 的执行时间的变化都很大。

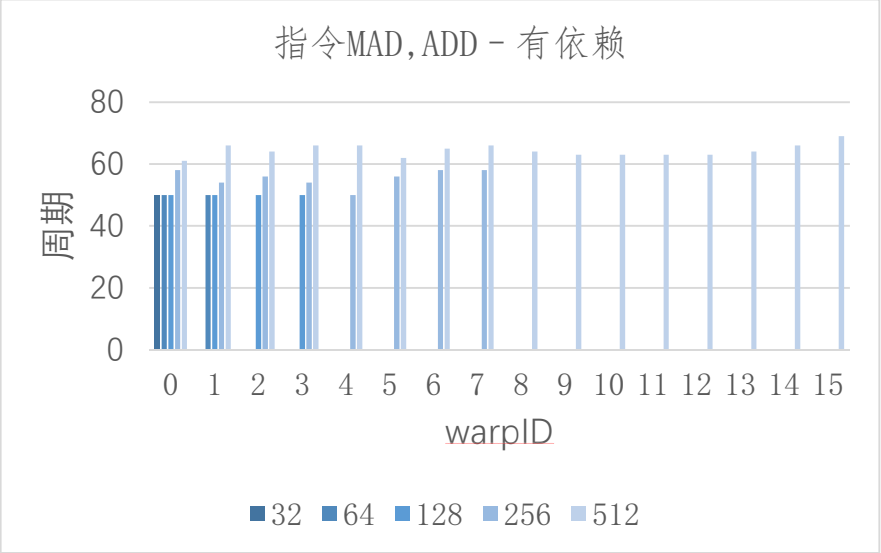

图 4.9 有依赖的 MAD 和 ADD 指令的执行周期

4.4 共享存储实验

共享存储部件对应于 NKGPGPU 中的 shared memory 部件。

在这一部分将会描述本文关于共享存储（shared memory）的一些实验。本文将用很少 store 和 load 指令来进行对共享内存的实验，然后会进行多次的实验来验证每次执行的结果是否相同。

图 4.10 展示了一条 load 指令，把操作数存入共享内存的执行周期。得到的结果与算术指令的结果十分类似，图中用纵坐标表示执行周期，横坐标表示线程 ID。在 block 的大小分别是 32,64,128 的情况下，每个 warp 的执行周期是相同的。在 block 大小为 256 和 512 的情况下 warp 的执行周期将增加，根据前文的结果，这是由于每个 warp 需要更多的调度。

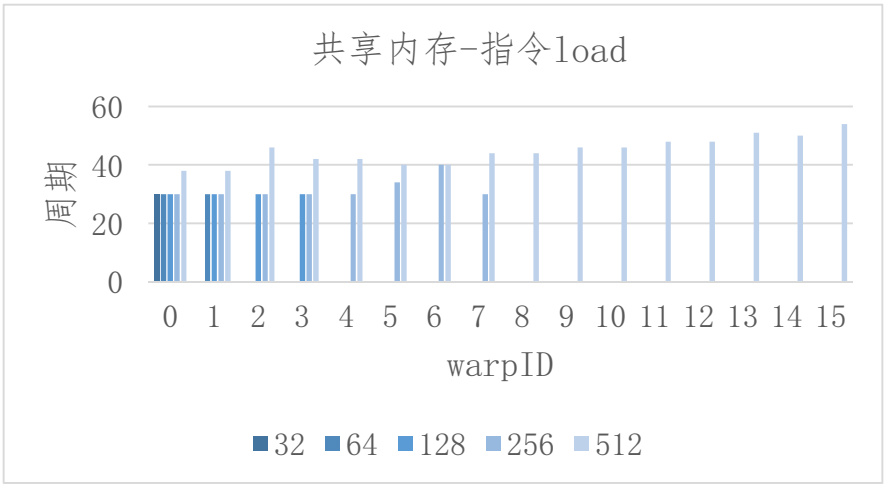

图 4.10 一条 load 指令的执行周期

随后，本文中进行了多次的实验。对于所有 8 次实验的结果也都是一样的，这一个结果表明 shared memory 是一个由软件控制的片上内存，并且与片上的其他部件没有相互影响。

对于 store 指令，本文采用了和 load 指令相同的实验方法，图 4.11 给出了一条 store 指令在不同 block 大小的执行周期数。和 load 指令一样，store 指令在 block 大小是 32，64,128 的情况下所有的 warp 执行周期都是一样的。同样由于调度的原因，在 block 大小为 128 和 256 的情况下每个 warp 都需要更高的执行周期。

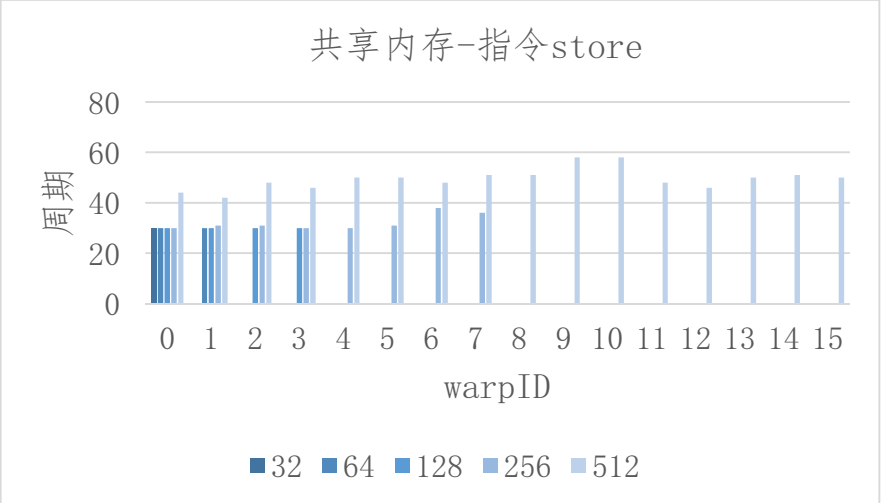

图 4.11 一条 load 指令的执行周期

在重复实验的情况下，我们还是得到了相同的结果。

在同样的 block 尺寸下，本文还验证了一下执行周期和 load、store 指令数量之间的关系。随着 load 和 store 指令数量的增加，单个指令的延迟产生了变化。纵坐标表示执行周期，横坐标表示指令的数量。本文只给出了执行周期最长的 warp 的执行周期，因为执行周期周期最长的 warp 决定了整体的执行时间。从下面两图可以看出在所有 block 大小的情况下，执行 store 和 load 指令的延迟随着 block 数量在线行增加。并且在 block 大小为 32, 64, 128, 256 的情况下 warp 最长执行周期是一样的。

总和上面的实验，将 10 条的 load 和 store 指令做一个对比，如图 4.12 纵坐标表示执行周期数，横坐标表示指令的数量。两种情况下，执行时间都随着指令数量的增加线行增加。更具体的说，store 指令的斜率更大，增速更快。

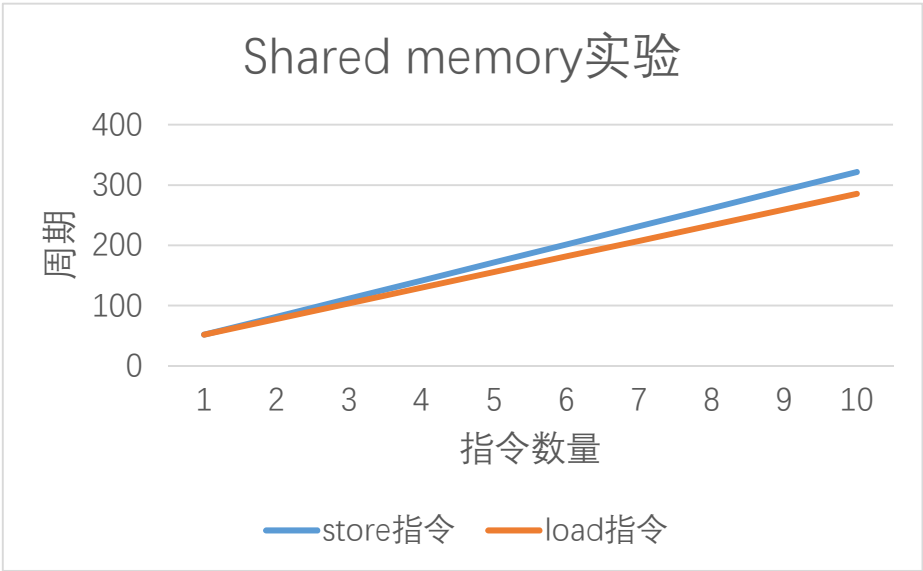

图 4.12 load、store 指令操作数在 shared memory 中的执行周期的对比

在一个 warp 的情况下，warp 中含有内存冲突（bank conflict）的情况。纵坐标表示执行的周期数，横坐标表示内存冲突的间隔（stride）。同样，执行的周期数随着冲突的数量的增加而增加。对于间隔为 64 和 128 得到了相同数量的冲突与间隔为 32 的执行周期保持一致。除此之外，我们还做了一个一条 load 指令一条 store 指令，两条指令之间具有 RaW 依赖的实验。执行周期与没有 RaW 依赖的执行时间十分相似，大约多出 24 个周期。

4.5 全局内存实验

全局内存实验对应于 global memory 部件。根据对 shared memory 和 global memory 的综合研究分析可以得到 PEd 部件的行为。

这一部分的实验主要研究一下 load、store 对 global memory 进行操作时的执行周期。我们采用了和 shared memory 相似的实验方法。

综合上面实验，本文做了一个 10 条 load、store 的实验，从图 4.13 可以看出 load 需要更多的周期。根据下图，我们可以得出执行一条新 load 指令需要 370 个周期，执行一条新 store 指令需要 34 个周期。

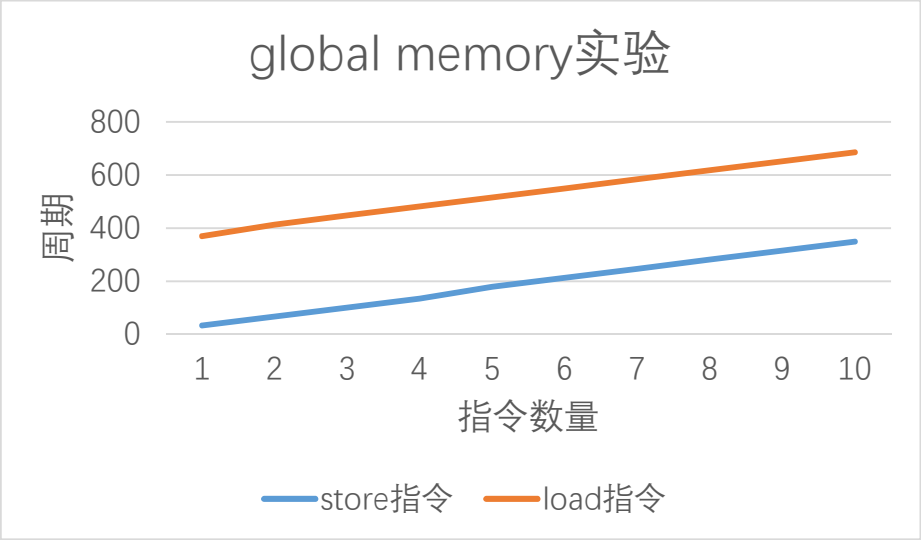

图 4.13 10 个 load、store 指令执行周期对比

在 block 大小为 512 时执行周期会有变化。Thread 会访问连续的内存空间，横坐标表示 warpID，纵坐标表示执行周期。所有的实验都有相似的趋势，但是不同次实验之间会有一些细小的差别，这说明所有 warp 访问存储的模式是相同的，但是还有一些其他因素影响了执行周期数。这些其他因素可能是 SM 内部网络请求，global memory 的刷新频率等等。

4.6 CUDA 编程实例

首先，本文实现了一个 1 至 8 的规约求和计算，没有考虑资源的使用情况，具体的规约过程如下：

```
for (int stride = 1;stride <= 4 ;stride *=2){
        __syncthreads();
        if (i%(stride)==0)
        a[2*i] =a[2*i]+a[2*i+stride];
```

（i 表示 threadIdx.x）

编码时，以 CUDA 的向量加法为基础。相比于 CUDA 中已经实现的各种算法，该算法十分简单，不需要理解很多的接口和复杂的错误控制，并且满足实验的要求。在编码时用了 shared memory。每一个数据只需要用一次，所以用 shared memory 不会同矩阵乘法效率一样高。

实现规约算法其的涵是在研究 thread 的使用，即更加合理的运用资源。对于最简单的规约和运算，从原理上十分容易理解，每一个 thread 负责计算两个数的值，对于一个长度为 n 的数组，通过 log2（n）次规约过程就可以得到结果，速度非常快。但是这种简单的算法是十分消耗资源的，需要大量的 thread 参与计算，但是很多的 thread 只会进行一次或者几次计算就会停止，对于此有很多改进的方法。

通过研究这一部分，warp 部件和 SIMP 理念的之间关系会更加明确。32-thread-warp 可以很好的实现 SIMP 理念，即根据同一个指令，32 个 thread 可以进行相同的操作。这也解释了控制分歧（control divergence）产生的原因。

启动 kernel 的代码如下，即 block 是 1 个，thread 有 4 个。

```
SimReKernel<<<1, 4>>>( dev_a);
```

在性能分析如图 4.14 中可以看到结果与设定相符。

	Function Name	Grid Dimensions	Block Dimensions	Start Time (μs)	Duration (μs)	Occupancy
1	SimReKernel	{1, 1, 1}	{4, 1, 1}	142,295.130	16.129	25.00 %

图 4.14kernel 启动情况

最终实验结果如图 4.15，最终结果在 a[0],即 thread0 计算了最后结果。

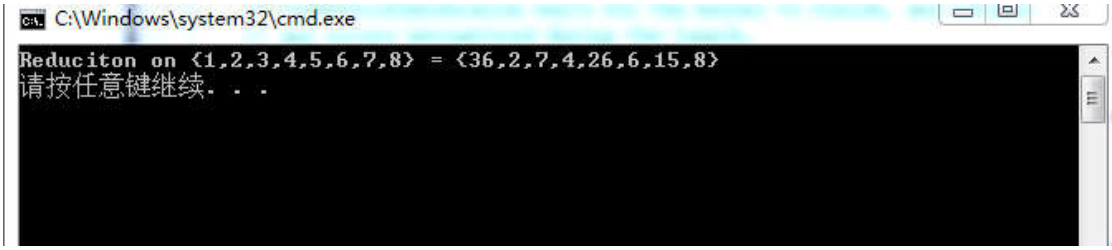

图 4.15 规约计算结果

为使编程更加快捷，本文实现了一个 4*4 的矩阵乘法，TILE_WIDTH=2,并且没有考虑越界的情况。这一实验调了好几天都没有结果， 最初的启动 kernel 的代码是这样的：

```
dim3 DimGrid=(2,2,1);
dim3 DimBlock=(2,2,1);
addKernel<<<DimGrid, DimBlock>>>(dev_c, dev_a, dev_b);
```

在研究启动 kernel 的实例的过程中，改变了写法，才得到正确结果。

```
addKernel<<<dim3(2,2,1), dim3(2,2,1)>>>(dev_c, dev_a, dev_b);
```

Function Name	Grid Dimensions	Block Dimensions	Dynamic Shared Mem/Block	Stream ID	Occupancy
addKernel	{2, 2, 1}	{2, 2, 1}	0	1	25.00 %

图 4.16 矩阵乘法使用的 block 设置

实验结果如下图：

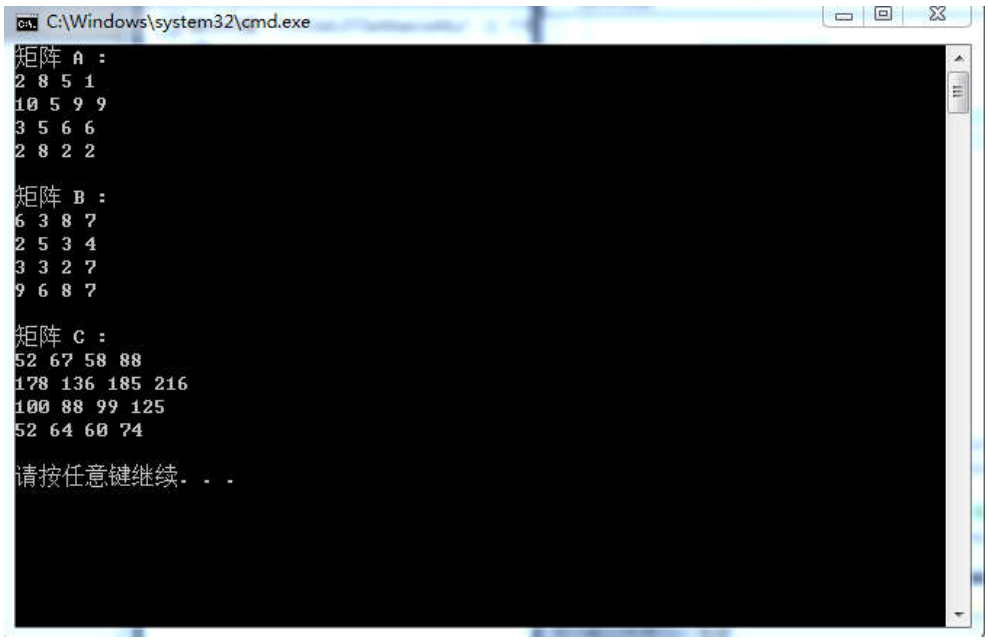

图 4.17 矩阵乘法结果

研究矩阵乘法，实际上是在研究存储器的相关内容。从粗粒度上看，分成 host memory 和 device memory。再进一步，device memory 中又有 global memory，shared memory 和

register。thread 读取 global memory 的速度较慢，所以想提高吞吐量的方法就在使用 shared memory。因为 shared memory 属于片上内存，读写速度非常快。分片（TILE）方法，就是对数据分片，让某一数据尽可能多的被 thread 使用，即这些 thread 从 shared memory 中读取数据，而不是从 global memory 中读取数据。对于 block 的长（宽），（blockDim.x（y））的设定应当分片长度（TILE_WIDTH）一致，方便计算。

4.7 在 GPGPU-Sim 平台运行基准程序

图 4.18 展示了 GPGPU-Sim 的执行过程。在 CUDA4.0 中，随着套件给出了 cuobjdump，可以从可执行文件（Executable）中提起出 PTX,SASS,ELF 信息，然后分为两种情况：1.直接使用 PTX 文件在 GPGPU-Sim 上运行；2 将 PTX,SASS,ELF 通过 cuobjdump_to_ptxplus 生成 PTXPlus，然后在 GPGPU-Sim 上执行。无论哪种情况，都需要先获得可执行文件。

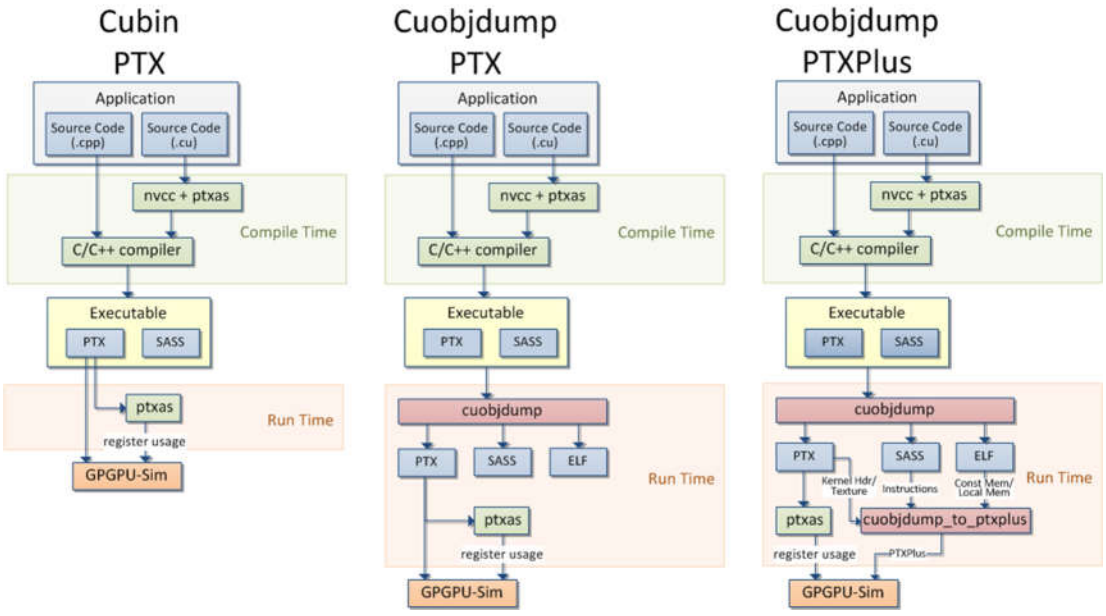

图 4.18 GPGPU-Sim 执行过程

GPGPU-Sim 在运行时可以简单的理解为一块支持 CUDA 的 GPU 芯片。在运行之前，仍然需要按照 CUDA4.0 的规范编写程序，并且使用 nvcc 编译通过。在本文使用的 ubuntu14.04 中，nvcc 可以启动 gcc 编译器来处理 c 代码，启动 PTX 编译器处理 CUDA 代码。基准程序 singleadd.cu，在 GPU 代码部分只有一条加法指令。可以通过下图 nvcc -o -singleadd singleadd.cu 指令获得可执行文件。如图 4.19 所示。

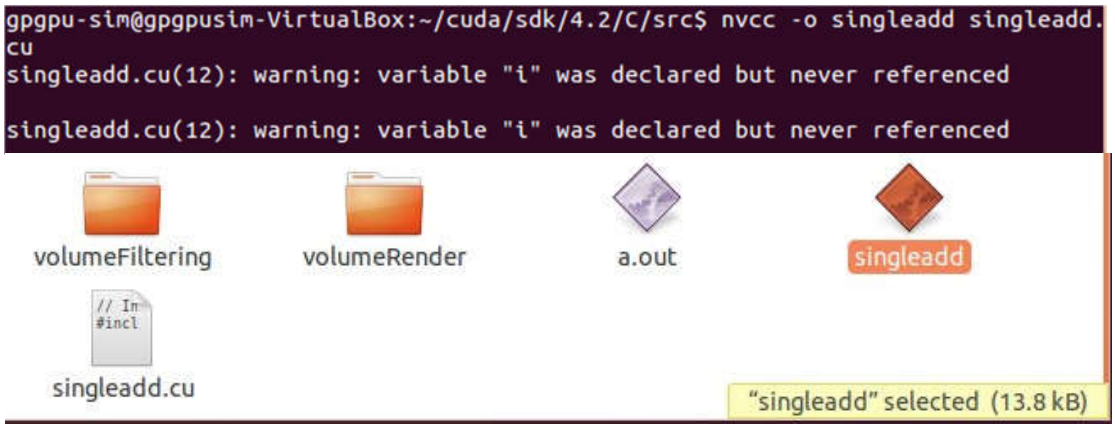

图 4.19 GPGPU-Sim 使用 NVCC 的编译结果

然后，根据官网给出的如下指令，运行该基准程序：

```
gpgpu-sim@gpgpusim-VirtualBox:~$ ./run_gpgpu-sim.sh /home/gpgpu-sim/singleadd
```

会得到十分详细的执行统计结果，截取其中的一部分如图 4.20 所示：

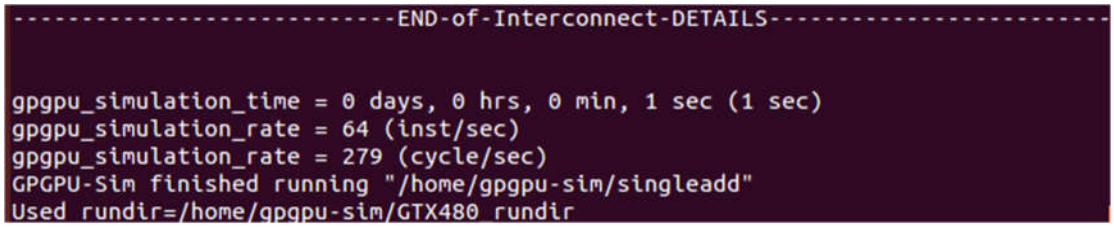

图 4.20 GPGPU-Sim 统计结果示意图

第 5 章 总结与展望

本文主要进行了以下几项工作：

1. 研究了现有 GPU 的结构,主要研究了 Fermi 结构的 GPU；
2. 通过 GPGPU-Sim 研究了 GPGPU 流水线并且对各个部件进行了实验；
3. 提出了 NK-GPGPU 模型；
4. 给出了一个 NKGPGPU 的性能分析模型；

目前各个行业对于并行计算的需求越来越多，GPU 也不在局限于完成图形图像计算，可以认为凡是有包含大规模数据并行的应用都可以使用 GPU 来完成。现在很多的超级计算机也都是以 GPGPU 芯片为核心。所以说 GPGPU 的发展会越来越好。

本文主要的焦点在于 Fermi 结构,相信如果能对英伟达公司的其他架构,比如 Kepler、Maxwell，还有其他厂家的 GPU 芯片，比如 AMD 和 Intel 公司的 GPU 的架构等，都做出一个详细的对比研究会得出更加卓越的结果。但是同样的，这将会是一个巨大的工作量。此外，现在的 CPU 已经发展的非常完善，如果能把 CPU 中的一些技术应用到 GPGPU 上，一定会有很好的效果。

本文使用到的编程模型是 CUDA，在英伟达公司的 GPGPU 上表现优异。由于发展时间较长，相关文档也比较健全，因此使用起来十分方便，但其通用性较差。现在蓬勃发展的 OpenCL 作为一个开源的 GPGPU 编程模型已经得到越来越多的厂家支持，对 OpenCL 的研究也会很好相信会促进对 GPGPU 的了解。

从科研领域看，现在如果是简单的对程序进行优化，或者说简单把 CPU 程序转移到 GPGPU 上从而获得一个 10 倍甚至 100 倍的性能提升已经不能成为一个非常有价值的科研问题。相比之下，对 GPGPU 内部的架构设计，比如存储、网络、调度等问题的研究价值会日益增高。

参考文献

[1] NVIDIA's Next Generation CUDATM Compute Architecture:FermiTM [J].

[2] BAKHODA A, YUAN G L, FUNG W W L, et al. Analyzing CUDA workloads using a detailed GPU simulator[C]. proceedings of the IEEE International Symposium on PERFORMANCE Analysis of Systems and Software, 2009.

[3] HU L, CHE X, ZHENG S Q. A Closer Look at GPGPU [J]. Acm Computing Surveys, 2016, 48(4): 60.

[4] FORTUNE S, WYLLIE J. Parallelism in Random Access Machines; proceedings of the ACM Symposium on Theory of Computing, May 1-3, 1978, San Diego, California, Usa, F, 1978 [C].

[5] VALIANT L G. A bridging model for parallel computation [J]. Communications of the Acm, 1990, 33(8): 103-11.

[6] CULLER D, KARP R, PATTERSON D, et al. LogP: towards a realistic model of parallel computation, F, 1993 [C].

[7] TISKIN A. The bulk-synchronous parallel random access machine [J]. Theoretical Computer Science, 1998, 196(1–2): 109-30.

[8] HU L, CHE X, XIE Z. GPGPU cloud: A paradigm for general purpose computing [J]. 清华大学学报自然科学版(英文版), 2013, 18(1): 22-3.

[9] VALIANT L G. A bridging model for multi-core computing [J]. Journal of Computer & System Sciences, 2011, 77(1): 154-66.

[10]朱俊峰, 陈钢, 张珂良, et al. 面向 OpenCL 架构的 GPGPU 量化性能模型 [J]. 小型微型计算机系统, 2013, 34(5): 1118-25.

[11]KOTHAPALLI K, MUKHERJEE R, REHMAN M S, et al. A performance prediction model for the CUDA GPGPU platform; proceedings of the International Conference on High PERFORMANCE Computing, Hipc 2009, December 16-19, 2009, Kochi, India, Proceedings, F, 2009 [C].

[12]MADOUGOU S, VARBANESCU A, DE LAAT C, et al. The landscape of GPGPU performance modeling tools [J]. Parallel Computing, 2016, 56: 18-33.

[13]SNAVELY, ALLAN, CARRINGTON, et al. A framework for performance modeling and prediction [J]. 2002, 6(3): 21.

[14]HONG S, KIM H. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness [J]. Acm Sigarch Computer Architecture News, 2009, 37(3): 152-63.

[15]SONG S, SU C, ROUNTREE B, et al. A Simplified and Accurate Model of Power-Performance Efficiency on Emergent GPU Architectures[C]. proceedings of the IEEE International Symposium on Parallel & Distributed Processing, 2013.

[16]KERR A, ANGER E, HENDRY G, et al. Eiger: A framework for the automated synthesis of statistical performance models[C]. proceedings of the International Conference on High PERFORMANCE Computing, 2012.

[17]ZHANG Y, HU Y, LI B, et al. Performance and Power Analysis of ATI GPU: A Statistical Approach[C]. proceedings of the IEEE Sixth International Conference on Networking, Architecture, and Storage, 2011.

[18]BAGHSORKHI S S, DELAHAYE M, PATEL S J, et al. An adaptive performance modeling tool for GPU architectures[C]. proceedings of the ACM Sigplan Symposium on Principles and Practice of Parallel Programming, 2010.

[19]WU G, GREATHOUSE J L, LYASHEVSKY A, et al. GPGPU performance and power estimation using machine learning[C]. proceedings of the IEEE International Symposium on High PERFORMANCE Computer Architecture, 2015.

[20]PALLIPURAM V K, SMITH M C, RAUT N, et al. A regression-based performance prediction framework for synchronous iterative algorithms on general purpose graphical processing unit clusters [J]. Concurrency & Computation Practice & Experience, 2014, 26(2): 532–60.

[21]WONG H, PAPADOPOULOU M M, SADOOGHI-ALVANDI M, et al. Demystifying GPU microarchitecture through microbenchmarking[C]. proceedings of the IEEE International Symposium on PERFORMANCE Analysis of Systems & Software, 2010.

[22]LEE S, MIN S-J, EIGENMANN R. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization [J]. Acm Sigplan Notices, 2009, 44(4): 101-10.

[23]YIXUN L, ZHANG E Z, SHEN X. A cross-input adaptive framework for GPU program

optimizations[C]. proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing, 23-29 May, 2009.

[24] YANG Y, XIANG P, KONG J, et al. A GPGPU Compiler for Memory Optimization and Parallelism Management [J]. Acm Sigplan Notices, 2010, 45(6): 86-97.

[25] Whitepaper NVIDIA's Next Generation CUDATM Compute Architecture: Kepler TM GK110 [J]. 2012.

[26] DIMARCO J, TAUFER M. Performance impact of dynamic parallelism on different clustering algorithms [M]. KELMELIS E J. Modeling And Simulation for Defense Systems And Applications Viii. 2013.

[27] YANG Y, ZHOU H. CUDA-NP: Realizing Nested Thread-Level Parallelism in GPGPU Applications [J]. 2014, 93-106.

[28] NARASIMAN V, SHEBANOW M, LEE C J, et al. Improving GPU performance via large warps and two-level warp scheduling [M]. Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture. Porto Alegre, Brazil; ACM. 2011: 308-17.

[29] PHOTHILIMTHANA P M, ANSEL J, RAGAN-KELLEY J, et al. Portable performance on heterogeneous architectures [J]. SIGARCH Comput Archit News, 2013, 41(1): 431-44.

[30] ZAMBRENO. M S A J. Dynamic Thread Creation for Improving Processor Utilization on SIMT Streaming Processor Architectures [J]. In MICRO, 2010,

[31] GOVINDARAJU N K, LLOYD B, DOTSENKO Y, et al. High Performance Discrete Fourier Transforms on Graphics Processors [M]. 2008.

[32] HONG S, KIM H, ACM. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness [M]. Isca 2009: 36th Annual International Symposium on Computer Architecture. 2009: 152-63.

致　谢

我最要感谢的是胡亮老师和车喜龙老师，同时还　要感谢何立波老师。下面包含了一些我的感悟，既是自己的一个总结，也希望读到它的同学能有些许启发。

三年的时间过得很快，确切的来说只有两年半。对于我来说，所在的校园是本科四年生活的校园，我熟悉这里的一切，各个楼的分布，校园周边哪里有好吃的，想去的各个地方要去要从学校怎么走。然而，随着学校里面开了一个集餐饮电影院游泳池于一体的日新楼投入使用，很多事情都改变了，我原本中午最长去玫瑰园食堂关门停业，曾经繁华的北门商圈现在很少过去。这些外在的改变其实越来越来方便了，所以让人很适应很舒服，仿佛它就是一直存在一样。

从本校读上来，身边是熟悉的老师和同学，仿佛有一种自豪感和优越感。然而这个东西容易让人迷失，忘却了自己最根本的学生身份，忘却了最根本的任务是学习。研究生阶段的学习和本科阶段的学习完全不一样，本科非常多的课程，把课程上好，准备好期末考试就好了，其实并不需要思考自己要干什么。所以刚开始读研究生的时候一下没有了本科那么大压力，一天天乐得逍遥自在，根本不懂得珍惜时间，总以为时间还很多。有人说，受教育的意义在于让人学会思考，而思考就会产生影响。这样一个看似自由自在的生活并没有让我快乐多久，我开始苦恼，我希望真正的做点事情。至此，良好的睡眠一去不返，我开始经常失眠。可是此时，我还是一个思想上的巨人，行动上的矮子，各种诱惑太强，我甚至沉迷于健身。

直到有一天，我正在吉大一院打针，接到了车老师的电话，让我进入讨论小组旁听。此时，我开始准备起了两周一次文献阅读的汇报。可以说此时开始正是接触文献阅读，或者是科研工作。然而此时的研究还是粗糙的，我此时还是把这个当成一项任务来应付。后来在开题报告的时候，算是对我的一次打击，由于之前工作的不认真不到位，并且私自修改了题目，导致老师们在开题报告很生气，严厉的批评了我。

然而车老师并没有放弃我，而是带我走入了全新的 GPU 领域。从那时开始，车老师与我的沟通保持在一周一次。每一周我把看的文献的总结，我的想法发给车老师，车老师总是很快的给我回复，甚至回复的内容比我写的还多。中间很多次想到放弃，GPU 这个领域确实入门比较难，但是车老师总是耐心的指导着我，让我充满希望。有一阶段车老师在外地，不方便邮件回复我，就打电话和我通话近一个小时来指导我，还有一次车老师在赶书

稿，在车上指导我的汇报，为此还耽误了接刚上幼儿园的孩子。这样的例子数不胜数，让我十分感激。可以说，车老师用自己的汗水把带我上了科研之路，让我能一直坚持下来，深刻体会科研。我为研究生阶段能有这样的老师而感到幸运。

这一段旅程已经接近尾声，下一段的人生路还很长。感谢研究生阶段的所有，让我能有所成长，现在还不是我最想要的样子，所以我仍然会继续努力，追赶上那个被基于厚望的自己。