



中国科学技术大学
University of Science and Technology of China

并行计算课程报告
STENCIL 模板计算的 GPU 加速范例

谭邵杰 SA21011013

2022 年 6 月 23 日

摘要

在很多计算密集的应用中, Stencil 算法 (模板计算) 是耗时最多和最为重要的计算核心。随着 GPU 通用计算加速卡出色的计算能力、功耗控制和编程的简化, GPU 加速卡在科学研究中的工作站和超级计算机中得到普遍使用, 如何在 GPU 这样的异构设备上将 Stencil 算法进行性能优化和提高效率便成了一个亟待解决的关键问题。本文将通过优化经典的 5 点 Stencil 计算, 一方面总结的 GPU 程序的常见优化技巧, 另一方面结合程序的特点进行针对性的优化。借助性能分析工具——nvprof、nvcompute, 探讨了不同优化手段对于计算性能产生的影响, 并在 Tesla P40 和 Ampere A100 两种不同的架构上进行了性能对比。对我们深入了解 Pascal 架构和 Ampere 架构对有重要的借鉴意义。同时也为我们以后研究 Ampere 架构下的服务器级 GPU 性能分析与调优提供了参考。

关键字: Stencil, GPU, Optimization, 性能优化, 高性能计算

目录


1 绪论	4
1.1 选题缘由	4
1.2 研究背景	4
1.2.1 Stencil 模板计算简介	4
1.2.2 GPU 计算技术的发展与应用	5
1.3 研究意义	7
1.4 相关研究工作	8
2 GPU 计算介绍	9
2.1 GPU 计算概述	9
2.1.1 GPU 计算历史	9
2.2 CUDA 编程概述	10
2.2.1 流处理器简介	10
2.2.2 CUDA 线程模型	11
2.2.3 CUDA 存储模型	12
2.2.4 CUDA 执行模型	14
3 Stencil 相关优化方法	16
3.1 Stencil 算法优化方法	16
3.1.1 Blocking	17
3.1.2 Boundary Neat	18
3.1.3 Register Reuse	18
3.1.4 Max SMEM-Usage	19
3.1.5 Cache Oblivious	20
3.1.6 Time Skewing	21
3.1.7 Circular Queue	21
3.2 CUDA 程序性能优化	22
3.2.1 GPU 占用率	22
3.2.2 访存带宽	23
3.2.3 共享存储器访问效率	23
3.2.4 分支优化	23
4 实验结果和分析	25
4.1 实验环境	25
4.2 实验结果	26

4.3 结果分析	27
5 总结与展望	28
5.1 总结	28
5.2 未来工作的展望	28
5.3 开源代码	28

1 绪论

1.1 选题缘由

课程学习时接触到 Nvidia SC07 的一篇 CUDA 优化的介绍。

Performance for 4M element reduction 

	Time (2 ²² ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x
Kernel 7: multiple elements per thread	0.268 ms	62.671 GB/s	1.42x	30.04x

Kernel 7 on 16M elements: 72 GB/s!

图 1: SC07 CUDA 优化

想自己也实现一下，而且之前虽然有 CPU 的优化经历，但是没有写过 CUDA 的小项目代码。作为尝试选取了 Stencil 计算作为切入点，所以才有了这篇课程报告。

1.2 研究背景

1.2.1 Stencil 模板计算简介

Stencil 计算也被称为结构化网格计算，是一类常见的嵌套循环算法，在科学工程中的应用十分广泛，如计算电磁学、地球物理学、天体物理学、气候模拟、天气模拟、大气模拟、流体力学等。此外，Stencil 计算也被广泛应用于其他领域，如多媒体、图像处理等。

Stencil 在离散空间中的网格点按照预定义的模版在时间维度上进行更新，格点的更新依赖于其邻居格点。Stencil 问题可以是任意维度 d ，即其离散空间可以用 d 维数组来表示。Stencil 计算在时间维度上迭代更新的 d 维网格被称为数据空间， d 维数据空间迭代更新产生的 $(d + 1)$ 维空间被称为迭代空间，即 d 维数据空间加 1 维时间。

1 维 Stencil 计算如图 2 所示，所有的点都通过计算得到，设置一个 $A[T][NX]$ 的数组，保存

每一个迭代空间的点, 横坐标为数据空间, 纵坐标为时间维度, 深灰色圆点表示 1 次 Stencil 计算, 其中, 上面的点为需要更新的格点, 它依赖于其左右邻居格点和自身在上一时间步的值。[1]

Stencil 的类型十分丰富, 根据计算数据空间的边界条件, 可将其分为常量边界和周期性边界; 按照数据空间可以划分为 1 维、2 维以及更高维; 根据邻居的格点数可以分为 1 维的 3 点、1 维的 5 点等; 根据形状可分为星型和盒型; 根据格点的依赖类型可分为 Jacobi 和 Gauss-Seidel 等。

本文的优化对象是常量边界的二维 5 点 Stencil 计算。

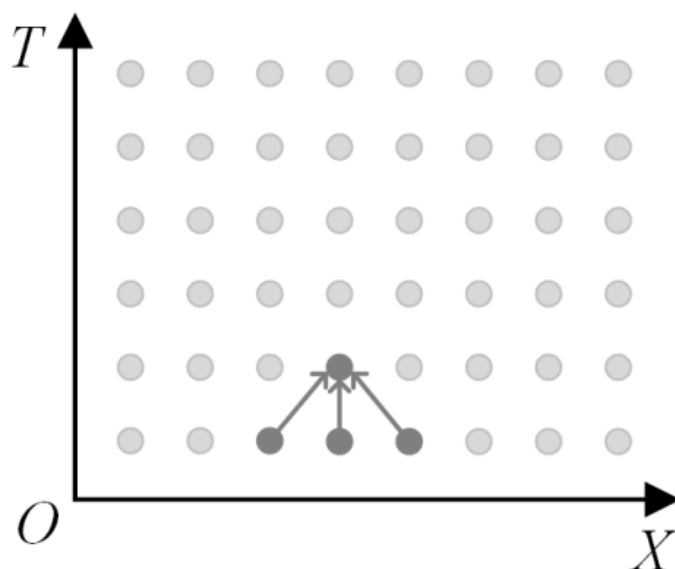


图 2: 1 维 Stencil 计算

1.2.2 GPU 计算技术的发展与应用

2003 年以前, 单核标量处理器的性能按照摩尔定律的规律不断地上升, 软件开发人员只需要一次性写好软件, 而性能的提升则只需被动地等待硬件的更新, 购买最新的硬件。2003 年之前的几十年里, 这种“免费的午餐”模式一直在持续。2003 年后, 由于发热量和功耗的原因, 处理器的频率停滞不前, 这种“免费的午餐”已经不复存在。标量单核的计算能力没有办法持续大幅度提升, 而伴随着人类在科学领域地不断深入, 数值计算的计算量增长迅猛, 气象预报、地质勘探、分子动力学等诸多领域的诸多应用对硬件计算能力的需求从没有停止。为了生存, 各 CPU 生产商不得不采用各种方式提高硬件的计算能力。目前最流行的

3 种方式如下 [2]:

- 让处理器在一个时钟周期内处理多条指令。如 Intel Haswell 处理器一个时钟周期内可执行 4 条整数加法指令、2 条浮点乘加指令，访存和运算指令也可同时执行。
- 使用向量指令，主要是 SIMD（单指令多数据）和 VLIW（超长指令字）技术。SIMD 技术将处理器一次能够处理的数据位数从字长扩大到 128 位或 256 位，甚至 512 位。从而提升了计算能力。
- 在同一个芯片中集成多个处理单元，根据集成方式的不同，相应地称为多核处理器或多路处理器。多核处理器的快速发展，以至于现在从服务器处理器、桌面版处理器到手机上的嵌入式处理器都已经是四核或八核的了。

同时，从 2006 年开始，由于其出色的功耗控制、计算能力和编程的简化，以及 GPU 的普及化，可编程的 GPU 越来越为大众所熟知认可。GPU 是图形处理单元（Graphics Processing Unit）的简称，设计的最初目的主要用于图形渲染，图形渲染的过程需要对多边形的各顶点和画面上各个像素反复进行向量矩阵乘法。而 GPU 的功能就是快速执行大量乘积累加运算、去除看不见的多边形、将可见多边形栅格化等。

3D 图形的巨大计算量与形成物体表面的多边形数量和面积等成比例，更高的分辨率要求多边形划分更加细致，高分辨率才能让画面更加真实，由于图形图像的性能需求没有止境，使得 GPU 的性能不断在提高。

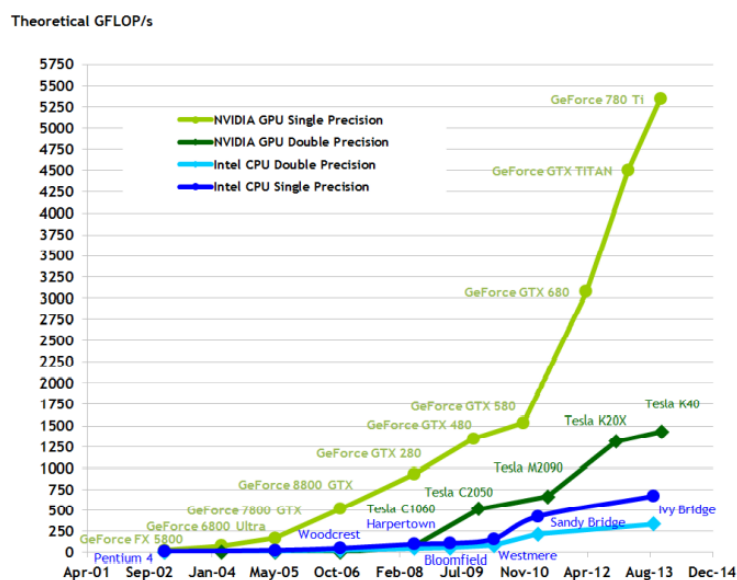


图 3: CPU 和 GPU 浮点数计算能力比较

最初的 GPU 用固定浮点数运算来计算坐标，但由于单位面积上晶体管数量依摩尔定律增加，GPU 硬件资源变得更为丰富，后来 GPU 改为使用 32 位单精度浮点数运算，以便处理范围更广的坐标值。因此，人们开始利用大量价格低廉的 SIMD 浮点数运算单元来组成 GPU。而且，随着图像真实性的提高，诸如循环判断等复杂的处理流程也浮现出来，因此 GPU 不仅要执行向量矩阵乘法，还要像通用 CPU 那样进行整数计算、逻辑判断等。然后，人们开始考虑使用功能不断丰富的 GPU 来进行图形计算之外的科学计算。

由于以上种种原因，GPU 脱离了图形处理范畴，慢慢变成了拥有通用数据处理能力的“General Purpose GPU”（GPGPU）。GPU 虽然工作频率较低，但是具有更多的内核数和并行计算能力，其总体性能-芯片面积比和性能-功耗比都很高。而且具有更高的访存带宽。

GPU 的发展使得异构计算异军突起，异构计算（Heterogeneous computing）主要是指使用不同类型指令集和体系架构的计算单元组成系统的计算方式。常见的计算单元有 CPU、GPU、DSP、FPGA 等。[3]

在最新的 TOP500 榜单中，使用异构加速卡的有 104 台系统，66 台使用了 NVIDIA 的 GPU 加速卡，还有 27 台使用了 Intel 的 Xeon Phi 加速卡，占到了全部异构系统的 63% 左右，占到了 TOP500 总数的 10% 以上。

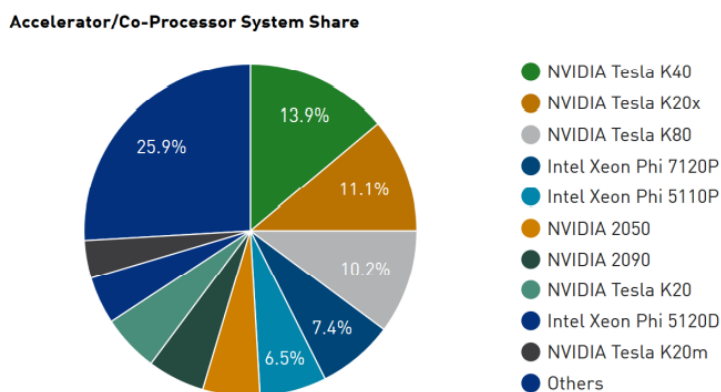


图 4: Top500 加速卡占比

1.3 研究意义

GPU 的特点是拥有较多的计算核心，但是其工作频率都不高以控制功耗，一般都要小于 1GHz。通过大量的计算核心来提高计算吞吐量和掩盖访存延迟，GPU 擅长处理规则数据结构和可预测存取模式，特别适合那些超大规模的并行密度高的运算程序。[3]

GPU 计算已经得到大量地普及，以及其可编程性变得简易，相关科研人员可以轻易地编写

一个可在 GPU 上运行的程序，然而在实际应用过程中，GPU 程序并不能取得与其理论计算峰值相匹配的计算性能。GPU 由于其与 CPU 迥异的体系结构，如其三层线程模型、复杂的存储模型等都与我们所熟知的常规的 CPU 编程模型不同，如何通过性能分析发现计算瓶颈和调优发掘 GPU 潜在的计算能力，已是一个重要的课题。

在很多计算密集型应用中，Stencil 计算是相对重要和耗时的计算核心部分，如何根据 GPU 的体系结构特性，对其进行性能分析与优化具有重要意义。

1.4 相关研究工作

近年来，国内外进行了许多关于模板计算在多核 CPU 和 GPU 上性能优化的研究。Matteo 等针提出了 cache oblivious 算法，该算法通过递归算法在时间和空间上计算最佳的 cache block 大小 [4]。最后该文将该算法推广到了 n 维的情形。

由于 Cache 大小限制，传统 tiling 算法在三维问题中不能有较好性能，Rivera 等提出了针对 3D Stencil 的 tiling 优化算法，该文首先提出二维情形下，编译器可以利用程序的局部性原理充分利用 cache 的使用率，进而说明 tiling 算法对二维程序性能提升有限。然而由于三维程序的数据量扩大以及访问区间的跨度扩大，使得编译器对三维 Stencil 程序很难进行自动优化，对 tiling 的形状和大小进行建模后，该文提出的算法提高了 cache 的命中率，平均得到了较好的性能 [5]。

Hikmet 等利用多线程和 SIMD 对高阶 Stencil 算法的优化进行了相关的研究 [6]。该文针对当今超级计算机主流架构，详细分析了如何通过不同层次的并行提升 Stencil 算法在集群的性能。该文通过空间分解实现了节点间并行，通过多线程实现了 CPU 核间并行，最后利用 SIMD 技术实现了数据并行。

Datta 等人 [7] 提出了利用 Circular Queue 算法在多核与单核 GPU 上实现 Stencil 算法，Yang 等 [8] 对该算法进行了改进，针对 GPU 提出了一种 Hybrid Circular Queue 的改进算法，相比只用共享内存的 Circular Queue 算法得到了较好的性能提升。

Meng 等人对 Stencil 优化中的 tiling 算法进行了建模，并提出根据不同的硬件特性和体系结构如何计算最佳的 ghost 区域 [9]。Maruyama 等提出了针对大规模多 GPU 下的 Stencil 编程模型，在 256 块 GPU 上取得的较好的拓展性 [10]。

Anamaria 等人对 Stencil 算法在 Fermi 和 Kepler 架构上的性能进行了比较与分析 [11]。文章给出了二维和三维情况下 GTX480 和 GTX680 对应的性能。但是该文并没有针对 Kepler 架构的特点给出具体的优化策略。

2 GPU 计算介绍

2.1 GPU 计算概述

在 20 世纪末，由于计算机芯片的频率和制造工艺水平的不断提高，CPU 性能依摩尔定律迅速提升。但是由于功耗过大和散热的原因导致不能无休止地提升 CPU 频率。从 20 世纪后，CPU 逐渐向多核及向量化发展，各种不同设计的 GPU 和加速器也被广泛使用。

2.1.1 GPU 计算历史

历史上，GPU 作为图形处理设备，专注于像素的渲染，GPU 的设计充分利用了像素渲染具有非常高的并行性的特性，通常 GPU 能够集成多达成百上千个计算核心。

从单位面积和单位功耗看，现代 GPU 的计算能力已经大大超过 CPU，另外 GPU 出货量也非常大，无形地降低了 GPU 的价格，这些因素促使科研人员把 GPU 作为通用计算研究对象，这统称为 GPGPU。早期的方式是通过把算法映射成图形渲染的过程，再使用图形编程接口来实现代码。这种方式编程难度大，不能充分利用硬件资源，难以调试和查错。因此这种传统的 GPGPU 编程没有被广泛使用。

2003 年，来自斯坦福大学的 Ian Buck 等人对 ANSI C 进行了拓展，开发了基于 CG 的 Brook 源到源的编译器，从而简化了通用图形计算的开发。Ian Buck 被称为 Brook 之父（现在在 NVIDIA 公司负责 CUDA 的开发）。但是虽然后来对 Brook 进行了升级，但是由于其借助的是图形接口，Brook 编译器效率不高，应用范围有限，并没有流行开来。2006 年，NVIDIA、AMD (ATI) 等 GPU 生产商观察到将 GPU 的并行性用于通用计算的潜力，2006 年 11 月，AMD 发布了 CTM (Close To Metal)，CTM 类似于机器语言，软件开发人员可以利用它来使用 AMD GPU 的所有资源，但是由于其太难于使用，也没有流行开来。2007 年 NVIDIA 公司推出了 CUDA (Compute Unified Device Architecture) 及 G80 通用计算显卡，CUDA 不再需要借助图形学编程接口，并且采用了类 C 语言进行开发，而且 CUDA 引入了同一的处理架构，从而大大地降低了编程的难度，使得 CUDA 更适用于 GPU 通用计算。NVIDIA 借助 CUDA 后来居上，超过竞争对手 AMD/ATI。2008 年，OpenCL 1.0 规范发布，OpenCL 全称为 Open Computing Language，是一个异构平台并行编程的开放标准，也是一个编程框架。支持 OpenCL 的硬件主要集中在 CPU、GPU、FPGA 和 Intel 近期推出的 MIC 加速卡，目前提供 OpenCL 开发环境的主要有 NVIDIA、AMD、ARM 和 Intel^[12]。2010 年，NVIDIA 公司推出了全新设计的 Fermi 架构，提供了更多科学计算急需的特性，比如对双精度浮点数的支持、原子操作以及支持 IEEE-754 (2008) 浮点标准。2012 年，AMD 推出了 GCN 架构的全新 GPU，GCN 架构吸取了 NVIDIA 公司 GPU 成功的经验，放弃了 VLIW（超长指令字），转而采用标量运算。HD7970 作为旗舰产品性能得到大幅提升。2012 年 3 月，NVIDIA 推出了 Kepler 架构，Kepler 架构并不简单地

Fermi 架构进行简单的升级，全新设计了的 SMX，极大地降低了功耗，同时还增加了动态并行、更快的原子操作，内核函数可以给自己分配计算任务、分离编译等。2014 年，Maxwell 作为 Kepler 的升级版正式推出，Maxwell 提升了核心的计算效率，降低了指令延迟，支持对指令执行时间进行 profiling。

就运算而言，CPU 和 GPU 各有适用的场景，这是由它们不同的体系结构和架构设计特点决定的。一般来说，CPU 擅长处理的是不规则数据结构和不可预测的数据存取模式，以及递归调用、条件分支密集和嵌套的代码，这类程序生成的指令复杂，具体的指令包括调度、循环终止判断、分支判断、逻辑运算等。GPU 能够处理的是规则数据存取模式，GPU 适合那些具有大规模的并行度和计算密度的应用。而 GPU 通用计算就是同时结合 CPU 和 GPU，实现协同计算，兼具两者的优势，达到加速的目的。

2.2 CUDA 编程概述

CUDA 是由 NVIDIA 于 2006 年 11 月推出的、用于将 GPU 作为通用计算能力的软硬件体系。它包括了硬件的体系结构（G80、GT200、Fermi、Kepler）、硬件的 CUDA 计算能力及 CUDA 程序如何映射到 GPU 上执行；作为语言它提供了编写 GPU 程序的语法、编译器等方面。CUDA 的架构包括其编程模型、存储器模型和执行模型。

CUDA 的计算能力由一个整数加一个一位小数表示，整数表示硬件架构，1 表示是 Tesla 架构（这里与 Tesla 系列 GPU 不同），2 为 Fermi，3 表示 Kepler 架构，Maxwell 为 5。而小数部分则表示性能提升、架构改进。通常 CUDA 计算能力变大意味着性能提升和可编程性提高，寄存器、共享存储器等资源数目的增加。一般而言，CUDA 计算能力越强越好。

目前，CUDA 提供两种不同的 API（运行时 API 和驱动 API）以满足不同人群的需要。运行时（runtime）API 基于驱动（driver）API 构建，程序也可以使用驱动 API。驱动 API 通过展示底层的抽象提供了额外的控制功能，如 CUDA Context（上下文），类似于设备上的主机进程，以及 CUDA 模块，类似于动态链接库。使用 runtime API 时，程序的初始化、模块管理、上下文的创建都是透明的，因此代码更加简洁。

相比 OpenCL，绝大多数情况下，同样的程序 CUDA 代码明显简洁不少，代码量更少，编程相对更加容易。并且 CUDA 提供了更多的功能支持，相关工具也更加丰富。

2.2.1 流处理器简介

GPU 的计算能力来源是流处理器簇（也称为 SM）[13]。流处理（SM）是 GPU 中执行内核函数的部分，一个流处理包含以下几个部分：

- 多达成千上万的可以被划分到执行线程的寄存器

- 若干种不同类型的缓存：
 - 用以在线程之间快速交换数据的共享内存
 - 用以快速从常量内存中读取数据的常量缓存
 - 用以从纹理内存中聚合带宽的纹理缓存
 - 用以减少访问本地或全局内存造成延迟的一级缓存
- 线程束调度器 (warp scheduler)，它可以在线程之间快速切换上下文，并将就绪指令发给待执行的线程束资源变得可用的时候。
- 浮点数变量和整数变量的执行核心，可以进行如下操作：
 - 整形和单精度浮点数的计算
 - 双精度浮点数的操作
 - 特殊函数单元 (Special Function Unit, SFU)，用来实现求倒数、平方根倒数、正余弦以及对数/指数的函数。

为了最大化 GPU 的吞吐量，GPU 包含了数量众多的寄存器，并且硬件可以在不同的线程之间快速切换上下文。GPU 线程切换的开销非常小。通过线程的切换，GPU 可以掩盖读指令执行后数据从设备中读出的数百时钟周期的访问延迟。

SM 是多用途处理器，但是 SM 的设计和 CPU 执行核心有很多不同：SM 的时钟频率通常比 CPU 的时钟频率要低很多；支持指令并行，但是不支持递归函数，也不支持分支预测和预测执行；没有或只有很少的缓存。

2.2.2 CUDA 线程模型

内核函数以线程块构成的网格进行启动。这些线程可以进一步分为 32 个线程组成的线程束 (warp)，而每个线程束中的单个线程被称为一个束内线程。线程块被独立调度到 SM 中，来自于同一线程块在同一 SM 中执行。如图 4-1 所示的是二维线程块 (4×4) 组成的一个二维网格 Grid 1 (2×3)。

每个线程块可以由高达 512 或 1024 个线程组成，而线程块中的线程之间可以通过 SM 的共享内存进行通信。一个网格中的线程块有可能会被分配给不同的 SM。为了使硬件吞吐量最大化，一个给定的 SM 可以在同一时间内运行来自不同线程块的线程与线程束。CUDA 编程模型并不能保证线程的执行次序，或者是某些线程块或线程是否可以并行运行。

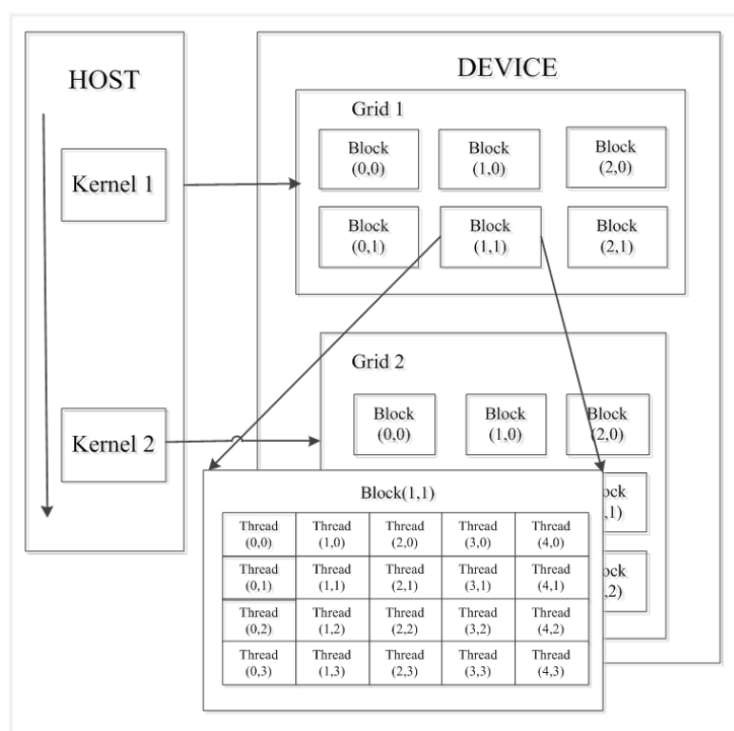


图 5: CUDA 线程模型

2.2.3 CUDA 存储模型

- 全局内存 (Global Memory) 即显存, Grid 中所有的 thread 都可以读写其中的任意地址, host 端负责分配全局内存, device 端和 host 端都可以初始化全局内存。编程人员可以用标准的 C 操作符来计算和引用地址, 包括指针算术运算和引用操作符: `*`、`[]`、和 `->`。通过合并内存事务, 可以获得最佳的使用性能。多线程同时对某一内存位置进行读写时, 为了避免发生读写冲突, 应使用原子操作。原子操作可以确保多个线程同时对同一个内存位置进行读操作时避免发生读写冲突, 以保证内存更新的正确性。在整个操作期间, GPU 硬件会强制在该内存位置上执行互斥访问。

- 常量内存 (Constant Memory) 用来加速常数的存取。所有 thread 都可以读常量存储器中的内容, 但是不能对其修改, 即不得在内核函数执行时从设备进行修改。常量内存只能由 host 端负责分配和初始化。常量内存驻留在设备内存上, 并且通过缓存进行了优化, 可以将读请求的结果广播到所有需要访问同一内存位置的线程。常量内存设置为只读, 简化了缓存管理, GPU 无需实现写回策略来处理被更新的内存。

- 共享内存 (Shared Memory) 共享内存属于片上存储, 具有高带宽和低延迟的特性, 但其

容量较小。共享内存对于 block 是私有的，但同一 block 内的 threads 可以同时访问共享内存中的内容，可用于 block 内线程之间的通信。每个 SM 有自己的共享内存，共享内存可以影响在同一 SM 中可执行的 block 数量，进而影响线程的占用率和并行度。共享内存有交替排列的存储体（bank）构成，并且通常是针对 32 位访问来优化的。如果一个线程束中多于一个线程访问同一存储体，存储体冲突（bank conflict）就发生了。

- 寄存器（Register）每个 SM 包含了成千上万个 32 位寄存器，当内核函数启动时，这些寄存器会被分配到指定的线程中。在 SM 中，寄存器的速度是最快的，也是数量最多的存储资源。例如 Kepler 架构下计算能力为 3.5 的 SM 包含了 65536 个寄存器，容量达 256KB，而纹理寄存器只有 48KB。CUDA 寄存器可以装入整形或浮点型数据。如果硬件支持的话（SM 1.3 或更高）可以进行双精度算术运算，操作数被放在偶数寄存器对中。开发人员可以使用命令行选项 `-ptxas` 和 `-verbose` 来让 `nvcc` 报告一个内核使用的寄存器数量。

- 本地内存（Local Memory）本地内存作为寄存器的补充，用来容纳寄存器溢出的数据，并存储着被索引的局部变量，因为这些局部变量的索引在编译时并不能够确定。本地内存与 L1、L2 缓存有着同样的优点和延迟特性。本地内存的寻址方式使内存事务可以自动合并（coalesced）。硬件包括了用来加载和存储本地内存的特殊指令：对于 Tesla 架构的 GPU，对应的 SASS 指令是 LLD/LST，而对于 Fermi 和 Kepler 架构的设备则是 LDL/STL。

- 纹理存储器（Texture Memory）纹理存储器是一种只读存储器，纹理存储器的数据位于显存，但可以通过纹理缓存加速读取。

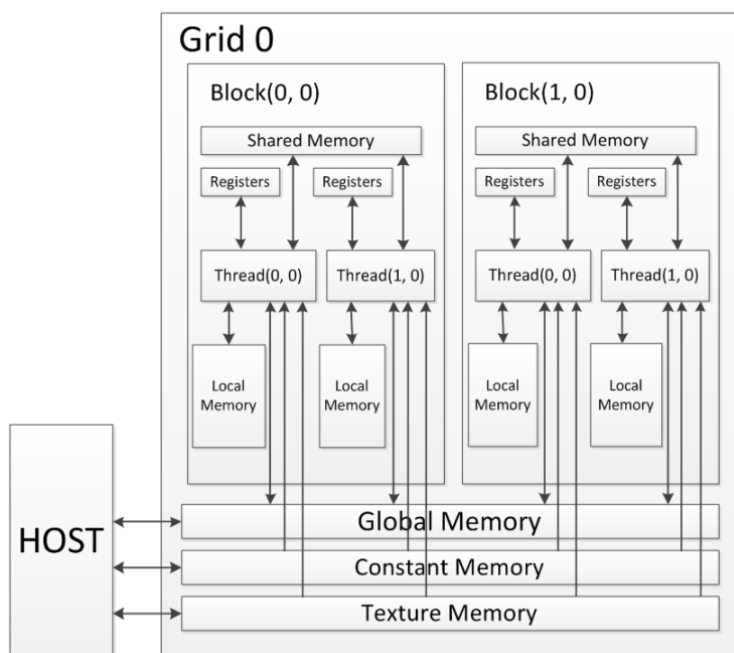


图 6: CUDA 存储模型

2.2.4 CUDA 执行模型

在 CUDA 中，可以用于计算的硬件包括两个部分，一个是 CPU（称为 host 端），一个是 GPU（称为 device 端），CPU 控制程序的执行流程，GPU 只是 CPU 的协处理器。CUDA 程序的执行可以分为两部分，一部分是在 host 端执行的普通 CPU 程序，另一部分是在 device 端执行的内核函数（kernel），host 端定义上下文，并通过上下文管理 device 设备，以及执行内核函数。从最早期的 CUDA 版本开始，GPU 与 CPU 都是并发执行的，也就是说内核函数启动时异步的：控制权会在 GPU 完成请求的操作之前返回给 CPU。此外，计算数据需要显式地拷贝到 GPU 并从 CPU 拷贝回来。

2.2.4.1 内核函数

在使用 CUDA 运行时的时候，内核函数的启动由三对尖括号来指定：

```
1 Kernel<<<gridsize, blocksize, sharedmem, stream>>>( parameters... )
```

其中，Kernel 指定待启动内核函数的名称，gridsize 指定一个 dim3 类型的网格的大小，blocksize 指定 dim3 类型的线程块的维度，sharedmem 指定为每个线程块预留的附加的共享内

存，stream 指定内核函数启动所属于的流。dim3 类型包含有 3 个成员：x、y 和 z。

2.2.4.2 主机内存

主机内存就是 CPU 内存，主机内存通常是采用虚拟内存管理器管理，使用分页的形式进行管理，通常为 4KB 大小，它们可被交换到磁盘上，有效地使用比实际物理内存更多的虚拟内存。

锁页主机内存由 CUDA 使用 cuMemHostAlloc()/cudaHostAlloc() 分配。该内存是页锁定的 (page-locking)，这样被标记为页锁定的内存页就不能被换出了，所以物理内存地址不能被修改。一旦内存被锁页，驱动程序便可以使用 DMA 硬件访问内存物理地址。DMA 避免了数据数据需要 CPU 的参与，并且通过 DMA 数据传输可以得到更好的总线表现。

因为锁页的内存页使其不再为其它分配所用，太多的锁页反而会影响计算机性能。映射锁页内存是指将锁页内存映射到 CUDA 上下文中的地址空间中，所以内核可以读写这块内存，与映射锁页内存不同的是，锁页内存不被映射到 CUDA 地址空间中，所以不能被内核函数合法的访问。

3 Stencil 相关优化方法

3.1 Stencil 算法优化方法

本章将介绍一些流行的 Stencil 算法。下图涵盖了从算法、Cache blocking、和硬件流水线等针对 Stencil 的优化方法，并按照编程和调试难度做了划分。一些算法虽然在 CPU 上会有比较好的性能，但由于 GPU 硬件设计不同，一些算法并不适合移植到 GPU 上。例如 GPU 的 L1/L2 cache 的作用并不与 CPU 的 cache 缓存相同，主要是为了减缓对全局内存访问的压力，提高带宽，数据并没有被缓存，所以一般使用共享内存来实现一些对 cache 优化的算法。本章将探讨不同算法在 GPU 上的可移植性和相应的优缺点。

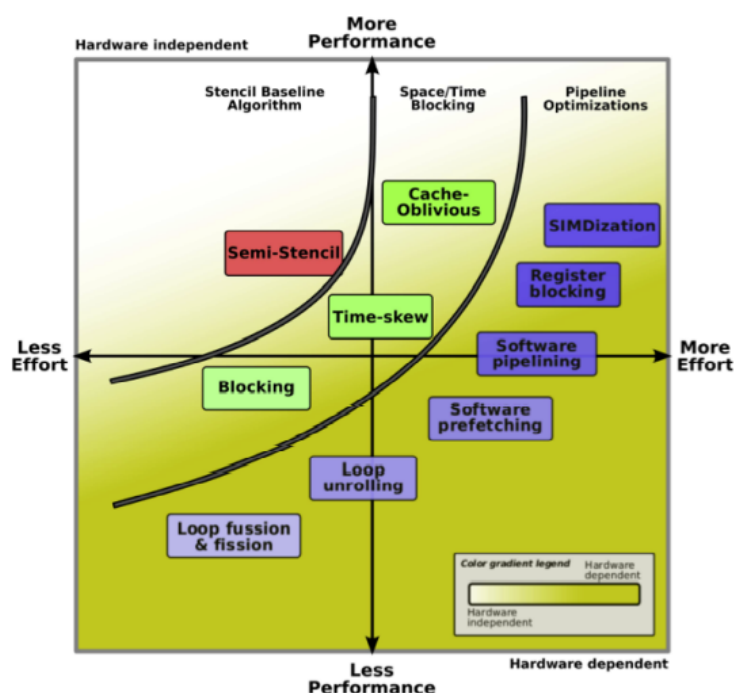


图 7: Stencil 算法优化技术

5 点的 Stencil 计算，并未使用任何优化算法和优化技术。我们将从最基本的 base 版本出发，作为研究的对象和性能比较基准。以下的伪代码为标准的未经优化的版本。

```
1 __global__ void stencil(int row_num, int col_num, int *arr_data, int *result) {  
2     auto index = blockIdx.x * blockDim.x + threadIdx.x;  
3     auto current_row = index / col_num;
```

```

4     auto current_col = index % col_num;
5     auto data0 = arr_data[index];
6     // up
7     auto data1 = arr_data[(current_row + row_num - 1) % row_num * col_num + current_col];
8     // down
9     auto data2 = arr_data[(current_row + 1) % row_num * col_num + current_col];
10    // left
11    auto data3 = arr_data[current_row * col_num + (current_col + col_num - 1) % col_num];
12    // right
13    auto data4 = arr_data[current_row * col_num + (current_col + 1) % col_num];
14    result[index] = data1 + data2 + data3 + data4 - 4 * data0;
15 }

```

3.1.1 Blocking

Blocking[5] 是一种局部缓存的优化技术，它也被称为 tiling 优化方法。在传统的 CPU 处理器编程技巧和 GPU 程序优化中都被广泛使用。它的核心思想是利用程序的局部性原理，将多次使用的数据保留在 L1Cache 或者共享内存中，利用共享内存访问速度比全局内存高的特点，减少对全局内存的访问以提高程序性能。

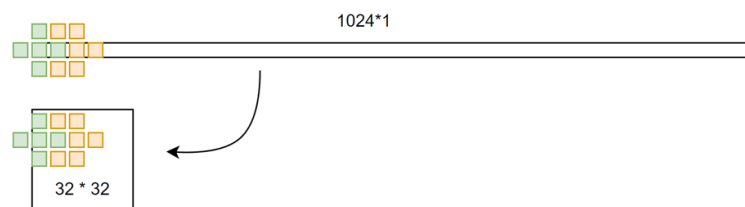


图 8: 5 点 Stencil 算法迭代

如下图红色块, 原本只能重用一行的数据。分块后能重用上下三行的数据

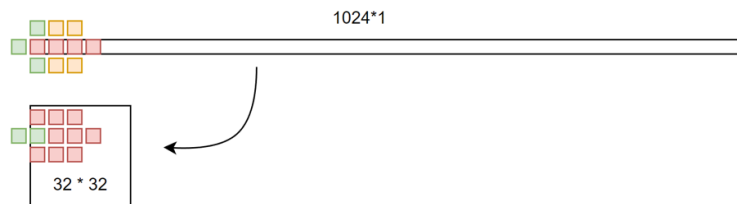


图 9: 5 点 Stencil 算法数据重用

3.1.2 Boundary Neat

base 版本的代码，对于周期性边界的每维度 512/1024 个元素，为了两端的两个元素需要付出额外的计算。为此可以简单的分类讨论来减少这部分的计算。

3.1.3 Register Reuse

Register Reuse 算法 [14] 是对 blocking 算法的改进，其核心思想是利用寄存器比共享内存的访问还要快，利用寄存器来替代共享内存。

在 CUDA 中，用户自定义的局部变量都会默认先存储在寄存器中，当寄存器大小不足以容纳所有数据时，数据将会被存储在本地内存中。开发人员可以使用命令行选项 `-ptxas` 和 `-verbose` 来让 `nvcc` 报告一个内核函数使用的寄存器数量。每一次编译使用的最大寄存器数，可以通过 `-ptxas-option-maxregcount N` 来指定。

在 blocking 算法中，我们使用了 3 个位于共享内存的数组 `f0`、`f1`、`f2`，分别表示当前需要计算的数据所需的在上下两个方向上的数据，在本算法中，我们不使用共享内存，而是使用寄存器替代。即对每一个执行内核的函数的线程分配两个局部变量 `f0` 和 `f1`，用以存储当前点 `f0` 的 `y` 方向上下两个点的值。计算完当前点之后，将 `f1` 的值赋给 `f0`，将 `f2` 的值赋给 `f1`，`f2` 所代表的那个点变成当前需要更新的点。

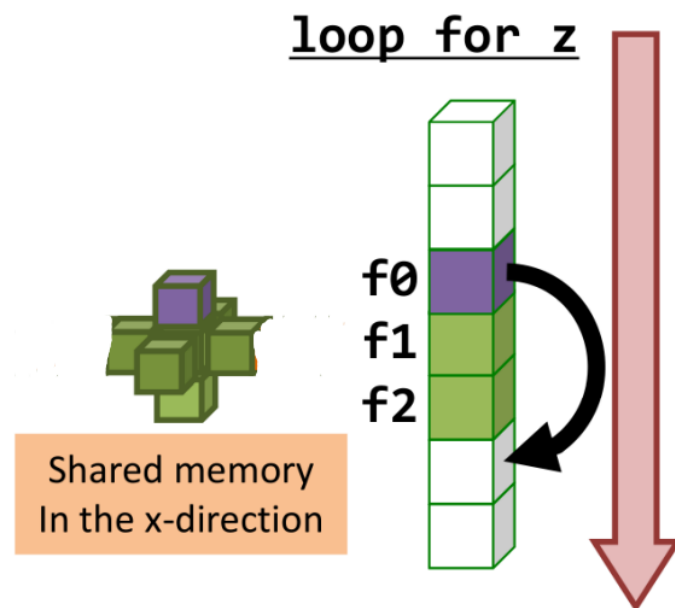


图 10: 5 点 Stencil register reuse 算法

3.1.4 Max SMEM-Usage

由于 GPU 中的 Cache 的作用并不与 CPU 相同，通常我们使用共享内存来实现更快速的 block 算法，所以也有人将共享内存称为用户管理的缓存。实际上共享内存与 L1 Cache 使用的是相同的存储体，其大小可以在启动内核函数的时候通过配置参数改变。

本文在实现过程中，增加了 halo 数据，halo 是对分割后的区域添加再扩大一圈，是为了在计算 block 边界数据的时候仍然可以从共享内存中读取数据。如此处理之后，更新一个点只需要从全局内存读取一次到共享内存，然后从共享内存中读取其上下左右和自己的 5 个点的数据，大大减少了对全局内存的访问。

通过 X 方向展开 block，来最大化 SMEM 的使用。由于横向展开元素足够多，通过改变访问模式，直接将所有的线程按行连续访问 (如图中的红色和黄色行节点)。两侧数据单独读取 (如图中的绿色节点)，可以将 load 平均次数从 5 次变成 2 次。如下图，每种颜色需要读取一次。

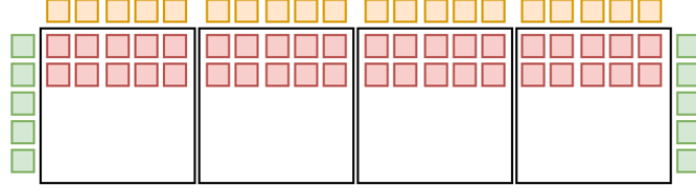


图 11: 5 点 Stencil Max SMEM-Usage 划分

3.1.5 Cache Oblivious

Cache oblivious[4] 算法实际上是一种隐式的 blocking 算法，但是与 blocking 算法不同的是，blocking 算法的 block 大小在程序运行过程中是固定大小的，而 cache oblivious 算法采用递归的过程，不断将需要计算的网格分割，直到分割后的部分能恰好全部放进 cache 中计算。

Cache oblivious 算法定义了一个 $n+1$ 维的时空区域，其中 n 指 stencil 算法的网格空间维度，该算法每一步的执行过程要么是通过将这 $n+1$ 维的时空区域在时间和空间上分割，要么当分割粒度足够小或者不能再分割的时候就开始计算。所以该算法对各级缓存都适用，递归的过程只要设置不同的参数都可以找到合适的 block 大小。

下图给出了一个在空间上一维的示例图，横纵表示空间上的维度，纵轴表示时间维度， dx_0 和 dx_1 表示每个时间步在 x 轴上走过的距离。

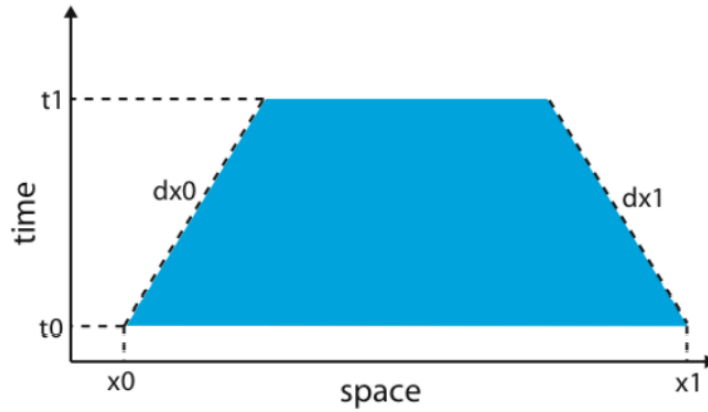


图 12: 二维时空区域

每一步分割，如果空间上的长度足够长，那么 cache oblivious 算法将继续递归地在空间上分割，如果划分的时空区域细且长，即纵轴相对比较长，那么就在时间维度上划分。如下图所示。

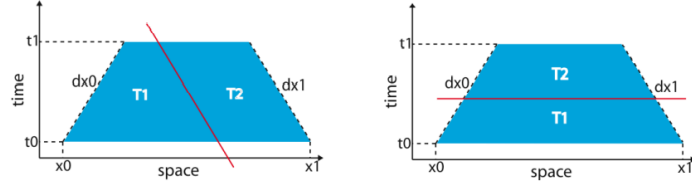


图 13: Cache oblivious 时空划分

但是对于 GPU 来说并不支持递归函数的实现，所以该算法并不适合在 GPU 实现。

3.1.6 Time Skewing

Time Skewing[15, 16] 也是一种以减少对内存的访问，尽可能利用提高 cache 的利用率来提高程序性能算法。

图 3-6 给出了空间上一维的 Time Skewing 算法示意图，在构成的二维时空网格上，用斜线将网格分割成若干个小 cache 块，类似于 cache oblivious 算法。之所以分割并不是采用水平或垂直分割的原因是为了遵循该 Stencil 算法中数据的相关性。

但是，由 Time Skewing 生成的 block 可能会导致负载不均衡的现象，每个 block 的计算量并不一致，而且由于通常 Stencil 算法需要迭代多次来得到计算结果，并且每次的分割与前面的计算并没有直接关联，通常是不变的，所以最后程序的效率就由最慢的那个 block 所决定。

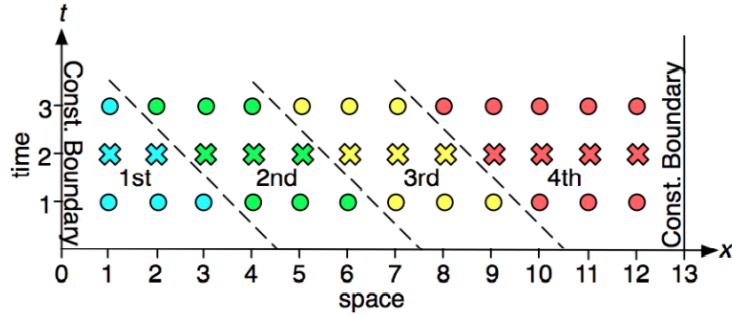


图 14: 二维 time skewing 算法

3.1.7 Circular Queue

Circular queue[8] 算法用到了一种特殊的额外的数据结构：循环队列。循环队列也称为圆形队列，是一种特殊的队列，表示一个队列长度大小固定的、头尾相连的数据结构。

在一个循环队列中，rear 指针指向队列最后一个元素的下一个位置，front 指向队列的第一个元素，判断队列为空的条件是 front 与 rear 相等，新元素入队时把值放入 rear 所在位置，rear+1 后取余 ($\text{rear} = \text{rear} + 1 \bmod \text{maxsize}$ ，maxsize 为队列的最大长度)，出队时 front+1 后对 maxsize 取余。判断队列是否为满比较复杂，一是可以增加一个参数用来记录当前队列所有的元素，也可以最后一个存储空间不用，当 rear+1=front 时表示队列为满。循环队列的一个特性是当一个元素出队后，其余元素不需要移动其存储位置，相反，在一般的线性存储的队列中，元素出队后，其余元素需要往前移动。

3.2 CUDA 程序性能优化

在软件开发中，性能分析 (Performance analysis 或 Profiling) 是指以收集程序运行过程中的信息与数据来研究程序性能和行为的分析过程，是一种动态的程序分析方法。在追求性能和硬件最大利用化的高性能计算领域，性能剖析是必不可少的步骤。

CUDA 开发套件包含了性能剖析工具和相关的 API，借助于性能剖析工具，可以获得 CUDA 程序运行过程中运行时间、内存消耗、硬件使用率和基本事件等信息，通过这些相关运行事件和数据，将使开发人员了解内核函数运行、与主机交互的细节等，帮助对程序进行优化。

本文所采用的性能分析工具是 NVIDIA 公司提供的命令行工具 nvprof 和图形化界面工具 NVIDIA Visual Profiler

3.2.1 GPU 占用率

GPU 占用率 (GPU occupancy) 是在 SM 中活动的 warp 数量与其理论上能支持的 warp 数的比值，在 SM 中执行的线程块都会需要一定数量的寄存器和共享内存，这些使用的寄存器和共享内存都是线程块独占的，而每个 SM 所拥有的寄存器和共享内存都是有限的，当线程块所需寄存器或者共享内存较多时，SM 拥有的资源不足以容纳多个线程块，SM 将限制在该 SM 上能同时运行的线程块数量。如果一个线程块所要的资源超过了 SM 所拥有的资源数，那么该内核函数将不能够运行。

增加占用率能提高性能是因为 warp 的切换开销可以忽略不计，而当某线程需要访问存储或因所要的资源并不能够立即得到时，通过线程的切换让其它线程执行其它指令，掩盖访存的延迟。然而高占用率也并不一定都意味着都有好的性能，如果一个内核函数是带宽受限类型的，那么提高 GPU 占用率可以对性能有提高，但是如果内核函数的瓶颈并不是访存带宽而是计算受限，那么提高占用率并不能对性能有影响。

3.2.2 访存带宽

为了提高访问全局内存的效率，CUDA 通过合并访问的机制来加速全局存储的访问。合并访问要求相邻的线程访问地址相邻的存储空间。

另外，我们可以在处理器需要某一数据之前，预先请求该数据，以降低缺失代价。其原理是将执行过程与数据预取过程重叠起来。只有当处理器在预取数据时能够继续工作的情况下，预取才有实用意义，也就是说当处理器待返回预取数据时不会停顿，而是继续执行指令和数据。

Samuel Williams 等人提出了 Roofline 模型 [17] 来对浮点数计算性能和带宽峰值进行建模，Roofline 模型分析了实测浮点数计算性能与理论浮点数计算性能、峰值带宽及操作密度的关系，通过该模型可以确定一个计算程序是访存受限还是计算受限。

我们不仅要通过实际的访存带宽来评价性能，那是因为一个程序可能是计算受限的，我们并不知道访存带宽能提升多少，而是关注于程序的访存效率，即合并访问是否得到了有效的利用。

3.2.3 共享存储器访问效率

共享内存被用来在同一线程块内的线程交换数据。物理上，共享内存属于 SM 可见的存储。在速度上，共享内存的访问延迟比寄存器的访问延迟要高，但是又比全局内存低不少，介于两者之间。因此，共享内存通常也作为寄存器与全局内存间的“缓存”，减少线程访问全局内存的延迟。

共享存储器以存储体为单位组织，不同的存储体可以同时被不同线程访问。在 Fermi 及之后的架构中，共享存储器由 32 个不同的存储体构成，访存单元也变成了线程束。如果一个 warp 内同时有两个或以上的线程访问同一个存储体上的数据时，就不能再一次访存事务中满足所有线程的需求，只能顺序访问，这称为共享内存冲突 (bank conflict)。为了避免发生访问共享内存冲突，要求 warp 内线程访问不同的存储体。但是当线程束中的所有线程从相同的位置读取数据也是非常快的，因为硬件提供了一种广播机制。此时所有线程对同一位置的数据的请求只发生一次，所有线程都能得到该数据。

3.2.4 分支优化

由于 GPU 硬件的原因，GPU 通常没有分支预测机制，线程束内的线程会执行相同的代码路径，而且 CUDA 的执行模式使得在遇上分支时只能让 warp 内的线程在每个分支上都执行一遍，分支会极大地削弱程序性能，因此要尽量减少分支的数目。但是由于线程执行都是以 warp 为最小单元执行的，所以不同 warp 间的分支并不会对程序性能造成太大的影响。

在 CUDA 编译器生成的中间代码 PTX 层次中，线程的控制流是由指令分支、调用、索引调用、返回和退出描述的，PTX 汇编代码分析了 PTX 分支图，对其进行优化，实现速度较快的指令序列。

由于在管理分支和汇聚（convergence）需要额外成本 [18]，CUDA 编译器在短指令序列中使用断定技术。大多数指令的效果可以通过条件判断出来，如果条件为假，则指令被禁用。如果这一断定发生的足够早，将会减少本应产生的性能损失。

在依赖条件变化的指令数比较小时往往会使用断定技术，C 语言中三元操作符（即？）被看作是一个使用断定技术较好的情形。但是，对于大量的条件判断代码，断定技术就变得低效，因为每一条指令都要被执行，而不管指令是否真正会影响计算结果。当大量指令导致断定执行的花销超过了断定本身带来的好处时，CUDA 就会使用条件分支。但线程束内代码的执行依据判断而呈现几条不同的执行路径时，我们称这样的为分支（divergent）。

常用的分支优化方法有：

- 将分支移到内核函数的外面，交给 CPU 处理。
- 合并分支条件以减少分支路径数量。
- 利用统一的表达式来计算不同分支的结果。例如：

```
1 int x,y; // x 为判断条件, y 为计算结果
2 int flag ;
3 flag = (x == true )? 1 : 0;
4 y = code_1*flag + code_2*(1-flag); // 根据 flag 的值, y 实际对应为执行不同的代码段
```

在实际应用中，for 循环也是一种条件分支语句，因为每一次循环都要判断下一次循环的条件是否满足，再进行下一次循环。但是当循环迭代过程是不相关的情况下，我们可以利用循环展开来提高性能。循环展开是一种编译时的优化技术，可以通过添加预处理指令 #pragma unroll 借助编译器简单实现。

4 实验结果和分析

4.1 实验环境

P40 的测试环境一栏如下:

```
# shaojiemike @ snode0 in ~/github/StencilGPU on git:main o [9:29:24]
$ neofetch

      .-/+00ssss00+/-,
      `:+ssssssssssssssssss+:`
      -+ssssssssssssssssss+-
      .osssssssssssssssssdMMMMNyssso.
      /ssssssssshdmmNNmmyNNMMMHssssss/
      +ssssssssshmydMMMMMMNdoddysssssss+
      /ssssssshNNMMYhhyyyhmmNNMMNHssssss/
      .sssssssdMMMNhssssssshNNMMdssssss.
      +ssssshhhyNNMMNysssssssssyNNMMYssssss+
      ossyNNMMNyMMHssssssssssshmmhssssssso
      ossyNNMMNyMMHssssssssssshmmhssssssso
      +ssssshhhyNNMMNysssssssssyNNMMYssssss+
      .sssssssdMMMNhssssssshNNMMdssssss.
      /ssssssshNNMMYhhyyyhdNNMMNHssssss/
      +sssssssdmydMMMMMMNdoddysssssss+
      /ssssssssshdmmNNNmyNNMMMHssssss/
      .osssssssssssssssssdMMMMNyssso.
      -+ssssssssssssssssssyyssss+-
      `:+ssssssssssssssss+:`
      .-/+00ssss00+/-,

      shaojiemike@snode0
      -----
      OS: Ubuntu 20.04.2 LTS x86_64
      Host: SYS-4028GR-TR 123456789
      Kernel: 5.4.0-104-generic
      Uptime: 100 days, 3 hours, 38 mins
      Packages: 1969 (dpkg), 2 (snap)
      Shell: zsh 5.8
      Resolution: 1024x768
      Terminal: /dev/pts/0
      CPU: Intel Xeon E5-2695 v4 (36) @ 3.300GHz
      GPU: NVIDIA Tesla P40
      GPU: NVIDIA Tesla P40
      GPU: NVIDIA Tesla P40
      GPU: NVIDIA Tesla P40
      GPU: NVIDIA Tesla P40
      GPU: NVIDIA Tesla P40
      GPU: NVIDIA Tesla P40
      Memory: 3960MiB / 128812MiB
```

图 15: Intel + P40 的测试环境一

A100 的测试环境一栏如下:

```
# shaojiemike @ hades1 in ~ [15:30:44]
$ neofetch

      .-/+00ssss00+/-,
      `:+ssssssssssssssssss+:`
      -+ssssssssssssssssss+-
      .osssssssssssssssssdMMMMNyssso.
      /ssssssssshdmmNNmmyNNMMMHssssss/
      +ssssssssshmydMMMMMMNdoddysssssss+
      /ssssssshNNMMYhhyyyhmmNNMMNHssssss/
      .sssssssdMMMNhssssssshNNMMdssssss.
      +ssssshhhyNNMMNysssssssssyNNMMYssssss+
      ossyNNMMNyMMHssssssssssshmmhssssssso
      ossyNNMMNyMMHssssssssssshmmhssssssso
      +ssssshhhyNNMMNysssssssssyNNMMYssssss+
      .sssssssdMMMNhssssssshNNMMdssssss.
      /ssssssshNNMMYhhyyyhdNNMMNHssssss/
      +sssssssdmydMMMMMMNdoddysssssss+
      /ssssssssshdmmNNNmyNNMMMHssssss/
      .osssssssssssssssssdMMMMNyssso.
      -+ssssssssssssssssssyyssss+-
      `:+ssssssssssssssss+:`
      .-/+00ssss00+/-,

      shaojiemike@hades1
      -----
      OS: Ubuntu 20.04.4 LTS x86_64
      Host: AS -4124GS-TNR 0123456789
      Kernel: 5.4.0-105-generic
      Uptime: 87 days, 22 hours, 56 mins
      Packages: 1584 (dpkg), 5 (snap)
      Shell: zsh 5.8
      Resolution: 1024x768
      Terminal: /dev/pts/10
      CPU: AMD EPYC 7543 (128) @ 2.800GHz
      GPU: NVIDIA 41:00.0 NVIDIA Corporation Device 20b5
      GPU: NVIDIA e1:00.0 NVIDIA Corporation Device 20f1
      GPU: NVIDIA 81:00.0 NVIDIA Corporation Device 20f1
      GPU: NVIDIA 25:00.0 NVIDIA Corporation Device 20f1
      Memory: 37408MiB / 2052015MiB
```

图 16: AMD + A100 的测试环境二

P40 和 A100 的参数基本对比图:

Tensor Model	P4	P40	P100	P100	P100	V100	V100	V100	T4	A100	A100
Bus	PCI-E 3.0	PCI-E 3.0	PCI-E 3.0	PCI-E 3.0	SXM4	HGX-1	PCI-E 3.0	SXM2	PCI-E 3.0	PCI-E 4.0	SXM4
GPU	GP104	GP102	GP100	GP100	GP100	GV100	GV100	GV100	TU104	GA100	GA100
FP32 Cores	2,560	3,840	3,584	3,584	3,584	5,120	5,120	5,120	2,560	6,912	6,912
FP64 Cores	640	960	1,792	1,792	1,792	2,560	2,560	2,560	-	3,456	3,456
Tensor Cores	-	-	-	-	-	640	640	640	320	432	432
Base Core Clock Speed	810 MHz	1,303 MHz	1,126 MHz	1,126 MHz	1,338 MHz	823 MHz	1,097 MHz	1,372 MHz	585 MHz	1,265 MHz	1,265 MHz
GPU Boost Clock Speed	1,063 MHz	1,531 MHz	1,303 MHz	1,303 MHz	1,480 MHz	918 MHz	1,224 MHz	1,530 MHz	1,590 MHz	1,410 MHz	1,410 MHz
SMs	20.0	30.0	56.0	56.0	56.0	80	80	80	40	108	108
Base FP16 Tensor Core FP16 ACC. Teraflops	-	-	-	-	-	-	-	-	-	-	-
Peak FP16 Tensor Core FP16 ACC. Teraflops	-	-	-	-	-	100.0	112.0	125.0	65.1	312/624	312/624
Base FP16 Tensor Core FP32 ACC. Teraflops	-	-	-	-	-	-	-	-	-	-	-
Peak FP16 Tensor Core FP32 ACC. Teraflops	-	-	-	-	-	100.0	112.0	125.0	65.1	312/624	312/624
Base BF16 Tensor Core FP32 ACC. Teraflops	-	-	-	-	-	-	-	-	-	-	-
Peak BF16 Tensor Core FP32 ACC. Teraflops	-	-	-	-	-	-	-	-	-	312/624	312/624
Base TF32 Tensor Core, Teraflops	-	-	-	-	-	-	-	-	-	-	-
Peak TF32 Tensor Core, Teraflops	-	-	-	-	-	-	-	-	-	156/312	156/312
Base FP64 Tensor Core, Teraflops	-	-	-	-	-	-	-	-	-	-	-
Peak FP64 Tensor Core, Teraflops	-	-	-	-	-	-	-	-	-	19.5	19.5
Base INT8 Tensor Core, Teraops	-	-	-	-	-	-	-	-	-	-	-
Peak INT8 Tensor Core, Teraops	-	-	-	-	-	-	-	-	-	624/1,248	624/1,248
Base INT4 Tensor Core, Teraops	-	-	-	-	-	-	-	-	-	-	-
Peak INT4 Tensor Core, Teraops	-	-	-	-	-	-	-	-	-	1,248/2,496	1,248/2,496
Base INT8, Teraops	16.6	40.0	-	-	-	-	-	-	-	-	-
Peak INT8, Teraops	21.8	47.0	-	-	-	50.2	56.0	62.8	120.0	-	-
Base INT4, Teraops	16.6	40.0	-	-	-	-	-	-	-	-	-
Peak INT4, Teraops	21.8	47.0	-	-	-	25.0	28.0	31.2	260.0	-	-
Base FP16, Teraflops	-	-	-	-	-	-	-	-	-	-	-
Peak FP16, Teraflops	-	-	18.7	18.7	21.2	25.1	28.0	31.4	-	78.0	78.0
Base BF16, Teraflops	-	-	-	-	-	-	-	-	-	-	-
Peak BF16, Teraflops	-	-	18.7	18.7	21.2	12.5	14.0	15.6	-	39.0	39.0
Base FP32, Teraflops	-	-	-	-	-	-	-	-	-	-	-
Peak FP32, Teraflops	5.5	11.8	9.3	9.3	10.6	12.6	14.0	15.7	8.1	19.5	19.5
Base FP64, Teraflops	-	-	-	-	-	-	-	-	-	-	-
Peak FP64, Teraflops	0.2	0.4	4.7	4.7	5.3	6.2	7.00	7.80	0.25	9.70	9.70
Base INT32, Teraops	-	-	-	-	-	-	-	-	-	-	-
Peak INT32, Teraops	-	-	-	-	-	12.6	14.00	15.70	-	19.50	19.50
GDR5 or GDDR6/HBM2 Memory	8 GB	24 GB	12 GB	16 GB	16 GB	16 GB	16/32 GB	16/32 GB	16 GB	40 GB	40 GB
Memory Clock Speed	3.0 GHz	3.6 GHz	703 MHz	703 MHz	703 MHz	877.5 MHz	877.5 MHz	877.5 MHz	1,200 MHz	1,215 MHz	1,215 MHz
Memory Bandwidth	192 GB/sec	346 GB/sec	540 GB/sec	720 GB/sec	720 GB/sec	900 GB/sec	900 GB/sec	900 GB/sec	320 GB/sec	1,555 GB/sec	1,555 GB/sec
Power Draw	50/75 W	250 W	250 W	250 W	300 W	150 W	250 W	300 W	70 W	400 W	400 W

* Base Teraops or Teraflops unknown

图 17: Nvidia 各显卡参数对比

4.2 实验结果

本文使用命令行工具 nvprof 和 nvCompute 进行相应的带宽测试和性能分析。

我们知道，GPU 的全局内存理论带宽通常在实际中都达不到，其值要高于程序实际运行带宽。在开启 ECC 内存错误检测和纠正功能的情况下，实际带宽能达到理论带宽的 75%-85% 左右。为了得到在实际应用过程中的带宽大小，我们利用 Stream benchmark 分别对不同的 GPU 进行了全局内存的带宽测试，Stream benchmark 是 HPC 领域通用的带宽测试基准。我们运行 Stream benchmark 程序，然后采用 NVIDIA Visual Profiler 对 Stream benchmark 所达到的带宽进行测量，并以 Stream benchmark 所测得的带宽为全局内存带宽的基准。

	P40	A100	P40	A100	P40	A100	P40	A100
	Time		Bandwidth		Step Speedup		Cumulative Speedup	
Baseline	539.8ms	411ms	15.1GB/s					
Blocking	376.7ms	183ms	33.3GB/s		1.43x	2.24x	1.43x	2.24x
boundary	255.1ms	155ms	85.3GB/s		1.47x	1.18x	2.11x	2.65x
SMEM	269.5ms	194ms	72.7GB/s		0.94x	0.79x	2.00x	2.11x
Data2GPU	79ms	15ms	66.5GB/s	80GB/s	3.41x	10.8x	6.83x	27.4x
MaxUseSMEM	29.6ms	7ms	96.3GB/s	236.7GB/s	2.67x	2.14x	18.23x	58.7x
SASS normalize	9ms	1.7ms	280.8GB/s	1.28TB/s	3.38x	4.11x	60.0x	241.7x

图 18: 各步骤优化提升对比

	P40	A100	P40	A100	P40	A100	P40	A100
	Time		Bandwidth		Step Speedup		Cumulative Speedup	
CPU单核+O3	4315ms							
CPU+Openmp16核+O3	381ms							
Baseline(GPU并行)	539.8ms	411ms	15.1GB/s					
Blocking	376.7ms	183ms	33.3GB/s		1.43x	2.24x	1.43x	2.24x
boundary	255.1ms	155ms	85.3GB/s		1.47x	1.18x	2.11x	2.65x
SMEM	269.5ms	194ms	72.7GB/s		0.94x	0.79x	2.00x	2.11x
Data2GPU	79ms	15ms	66.5GB/s	80GB/s	3.41x	10.8x	6.83x	27.4x
MaxUseSMEM	29.6ms	7ms	96.3GB/s	236.7GB/s	2.67x	2.14x	18.23x	58.7x
SASS normalize	9ms	1.7ms	280.8GB/s	1.28TB/s	3.38x	4.11x	60.0x	241.7x

图 19: 与 CPU 运行耗时对比

4.3 结果分析

经过上一章的各步骤的优化，将 GPU 的带宽都有充分的利用。

GPU	优化后带宽使用	理论带宽上限	使用率
P40	280.8GB/s	347.04GB/s	80.9%
A100	1.28 TB/s	1.52TB/s	84.2%

符合预期，最后 15-20% 的带宽和 Stencil 两侧的数据读取不连续有关。(尝试整个一行读取，会不满足 128 字节的 cacheLine 大小)

5 总结与展望

5.1 总结

随着近年 GPU 通用计算技术 (GPGPU) 的不断改进和发展, 计算能力更强, 价格更加低廉和其可编程性不断的提高, 使用 GPU 加速计算密集型程序已经成为一个研究热点。而随着可编程性的降低, 人们对性能的追求更加强烈, 然而由于 GPU 与 CPU 的体系结构的迥异与编程模型的差异等, 在 GPU 上的性能分析与调优与传统的 CPU 编程方式下并不相同, 给程序性能提升带来了困难。

数值求解偏微分方程是一类典型的计算密集型应用, 其涉及十分广泛, 如在物理学、化学、生物、经济等自然科学、社会科学和工程技术都有紧密联系。Stencil 计算又称模板计算, 是指在多维网格中的一点的更新是根据周围格点在时间与空间上加权求和的过程。一个偏微分方程的求解过程就是不断地迭代 Stencil 计算的过程。

本报告围绕在数值计算中普遍使用的 Stencil 算法在 Nvidia 显卡下的性能分析与调优这一主题, 进行了以下几方面的工作:

- 研究了 Stencil 算法中 5 点 Stencil 程序的特点和常见的优化算法, 利用性能剖析工具提出了 GPU 硬件性能指标和相应的优化方法。
- 结合 Pascal 和 Ampere 架构和 Stencil 算法的特点, 实现了 Stencil 算法在两架构上的性能调优。

5.2 未来工作的展望

一方面, 引入时间维度的考虑。对于多时间步, SMEM 的数据重用效果会更明显, 但是会有相邻区域的 SMEM 的数据传输问题有待解决。

另一方面, 对于更加复杂的 Stencil 应用, 例如长程 Stencil 算法 3DFD, 更新 3DFD 中的一个点, 需要自己及上下左右前后 24 个点, 共 25 点, 即 3D25P; 在石油勘探领域应用普遍的三维纵横波分离的弹性波方程模拟 (3D pure P and S wave elastic wave equation modeling) 等做深入细致的研究。

5.3 开源代码

<https://github.com/Kirrito-k423/StencilAcc>

参考文献

- [1] 纪璎芮, 袁良, and 张云泉, “红黑 gauss-seidel stencil 并行性和局部性优化,” 计算机科学, vol. 49, pp. 363–370, 5 2022.
- [2] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.
- [3] 李文强, “基于 stencil 算法的 nvidia kepler 架构下的性能分析与调优,” 硕士论文, 上海交通大学, 2016.
- [4] M. Frigo and V. Strumpen, “Cache oblivious stencil computations,” in *Proceedings of the 19th annual international conference on Supercomputing*, pp. 361–366, 2005.
- [5] G. Rivera and C.-W. Tseng, “Tiling optimizations for 3d scientific computations,” in *SC’00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, pp. 32–32, IEEE, 2000.
- [6] H. Dursun, K.-i. Nomura, L. Peng, R. Seymour, W. Wang, R. K. Kalia, A. Nakano, and P. Vashishta, “A multilevel parallelization framework for high-order stencil computations,” in *European Conference on Parallel Processing*, pp. 642–653, Springer, 2009.
- [7] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, “Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures,” in *SC’08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pp. 1–12, IEEE, 2008.
- [8] Y. Yang, H.-M. Cui, X.-B. Feng, and J.-L. Xue, “A hybrid circular queue method for iterative stencil computations on gpus,” *Journal of Computer Science and Technology*, vol. 27, no. 1, pp. 57–74, 2012.
- [9] J. Meng and K. Skadron, “Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus,” in *Proceedings of the 23rd international conference on Supercomputing*, pp. 256–265, 2009.
- [10] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, “Physis: an implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2011.

- [11] A. Vizitiu, L. Itu, C. Niță, and C. Suci, “Optimized three-dimensional stencil computation on fermi and kepler gpus,” in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–6, IEEE, 2014.
- [12] 刘文志, 并行编程方法与优化实践. 北京: 机械工业出版社, 2015.
- [13] N. Wilt, *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education, 2013.
- [14] G. Jin, T. Endo, and S. Matsuoka, “A multi-level optimization method for stencil computation on the domain that is bigger than memory capacity of gpu,” in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pp. 1080–1087, IEEE, 2013.
- [15] J. McCalpin and D. Wonnacott, “Time skewing: A value-based approach to optimizing for memory locality,” tech. rep., Rutgers University, 1998.
- [16] D. Wonnacott, “Using time skewing to eliminate idle time due to memory bandwidth and network limitations,” in *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, pp. 171–180, IEEE, 2000.
- [17] S. Williams, A. Waterman, and D. Patterson, “Roofline: an insightful visual performance model for multicore architectures,” *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.
- [18] S. Ryoo, C. I. Rodrigues, S. S. Stone, J. A. Stratton, S.-Z. Ueng, S. S. Baghsorkhi, and W. H. Wen-mei, “Program optimization carving for gpu computing,” *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1389–1401, 2008.