密级:_____

中国科学院研究生院

硕士学位论文

非结构网格 Eular 方程求解器 GPU 加速技术研究

作者姓名: 宋慎义

指导教师: 陆忠华　研究员

中国科学院计算机网络信息中心

学位类别: 工学硕士

学科专业: 计算机软件与理论

培养单位: 中国科学院计算机网络信息中心

2012 年 5 月

Acceleration of Eular Solver for Unstructured Grids on GPU

By

Song Shenyi

A Dissertation Submitted to

Graduate University of Chinese Academy of Sciences

In partial fulfillment of the requirement

For the degree of

Master of Engineering

Computer Network Information Center,

China Academy of Sciences

May, 2012

声　　明

本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果。就我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

作者签名：　　日期：

关于论文使用授权的说明

中国科学院计算机网络信息中心、中国科学院研究生院有权处理、保留送交论文的复印件，允许论文被查阅和借阅；并可以公布论文的全部或部分内容，可以采用影印、缩印或其它复制手段保存该论文。

作者签名：　　导师签名：　　日期：

摘 要

在计算流体力学问题的求解中，基于非结构网格的求解器渐渐地成为主流研究方向，但是 GPU 通用计算发展近 10 年，GPU 上的求解程序还是以结构网格的程序为主，基于 GPU 的非结构网格程序大多采用隐式格式，而对于 CFD 中广泛使用的显式格式，非结构网格下的 GPU 求解器很难达到较高的加速比。GPU 通用计算发展几年来，架构更新换代很快，一些原本被研究人员公认的优化理念和方法可能并不适合非结构网格的 CFD 求解，一些优化方法随着产品的更新换代可能已经失去了效果。

本文在 GPU 平台上实现了非结构网格有限体积法求解器，并结合 Fermi 架构的硬件特点，对 GPU 上的程序进行再次优化，大幅提升了加速效果。本文的主要内容和成果如下：

1. 介绍了非结构网格上的有限体积法求解过程，并行处理器的发展以及 GPU 从显示部件慢慢进化为并行处理器的历程，详细描述了最新的 Fermi 架构作为并行处理器的新特性。

2. 将二维非结构网格的 Eular 方程有限体积法求解器移植到 GPU 平台，使用了结构体的数组到数组的结构体转换、减少数据传输、循环展开、指令优化等加速方法，对 GPU 程序进行优化，并与 CPU 程序作对比。

3. 提出了调整非结构网格数据的存储顺序、降低 warp 占用率等方法，针对非结构网格求解程序在 GPU 上加速困难的原因，结合 Fermi 架构的硬件特点，在原来的基础上大幅提高了性能，深度挖掘了 GPU 的计算潜力，原本加速 10 倍的 GPU 程序，经过优化，最终相对于 CPU 加速约 40 倍。

【关键字】 计算流体力学，GPU 通用计算，非结构网格

Abstract

In research to solve the CFD (Computational Fluid Dynamics) problem, the focus has been gradually changed to the unstructured grid solver. Though GPGPU (General Purpose GPU) has been developed for about 10 years, most of CFD solvers on GPU are still based on the structured grid. The unstructured grid solver implemented on the GPUs nowadays mainly used the implicit difference scheme. Meanwhile, all the unstructured grid solvers with explicit difference scheme, which are widely used in CFD, have low speed up on GPU than the structured ones. As the development of GPGPU in the recent years, the architecture of GPUs updated rapidly. A lot of general optimization methods were recommended by the researchers, but they might not be suitable for unstructured CFD solver. Some methods are even no longer helpful for the new architectures.

We implement a unstructured CFD solver on GPU. In considered to the features of the Fermi architecture, we re-optimized the CFD solver on GPU and get more speed up. The main contribution of this paper is:

1. We summarize the workflow of a FVM (Finite Volume Method) solver based on the unstructured grid, and introduce the development of parallel process unit, especially the changes of GPU from a graphic unit to a parallel processor. The new features of the Fermi architecture, as also detailed as a parallel processor.

2. Some optimization methods are used on the unstructured grid CFD solver on GPU. A two-dimension unstructured grid CFD solver has been implemented on GPU, which uses Euler function and FVM. Then the program is optimized using many methods, such as changing array-of-structure to structure-of-array, decreasing data transfer, unrolling loops, optimizing instructions.

3. We have made a special effort to analyze the performance bottleneck of our un unstructured grid solver. In consideration to the features of the Fermi architecture, we propose a new method fit to the unstructured grid. The storage sequence of unstructured

grid data is adjusted, and the "occupancy" of warps is reduced to explore more potentialities of GPU and gain much more speedup. Experimental results show that the GPU program finally gains about 40X speedup after the optimization, in contract to the initial 10X speedup in the beginning.

Keywords: Computational Fluid Dynamics; GPGPU; Unstructured Grid.

目录

第一章 绪论

1.1 研究背景

计算流体力学（Computational Fluid Dynamics， CFD）是从 20 世纪 60 年代起伴随计算机科学迅速发展而形成的，通过计算机数值模拟对包含有流体流动和热传导等相关物理现象进行计算机数值分析和研究的学科。计算流体力学以计算机为工具，应用各种离散化方法，对近代流体力学的基础理论问题和工程应用问题进行数值模拟和数值分析，探讨工程实践中的流体流动从而揭示新的流体规律。

CFD 使用数值方法求解 Navier-Sotkes（N-S）方程或 Euler 方程，严格的来讲，CFD 是一个计算密集与通信密集学科的结合体，有严格的计算能力的要求。一般来说，对全机 CFD 计算采用 N-S 方程的浮点运算量达到 10^{15}~10^{18}flops[1]，这只能在并行计算机上完成；对于机翼的计算，采用 N-S 方程的运算量也在 10^{14}flops，在单颗 CPU 上也是非常耗时的。计算流体力学研究的最关键问题就是高效率低成本地解决计算问题。随着高性能计算机的发展，高效并行的航空计算软件业越来越受到各国的重视，并将其规划到影响未来发展的高科技领域当中，这也进一步提升了航空计算对高性能计算能力的需求。

近几年来，基于图形处理器（Graphic Process Unit, GPU）的通用计算技术的发展，如 NVIDIA 的统一计算设备架构（Compute Unified Device Architecture，CUDA），AMD 的应用程序接口 Brook+，苹果的 OpenCL 等等，为解决 CFD 计算问题提供了一条有利途径。目前的主流 GPU 拥有高出主流 CPU 一个数量级的浮点运算性能。充分利用 GPU 的浮点性能，在 PC 上快速进行一般规模（如机翼结构）的 CFD 计算，或者在异构的高性能集群上完成大规模（如全机）的 CFD 计算任务，将会对高精度大规模 CFD 应用产生巨大的推动作用。

1.2 国内外研究现状

使用 GPU 对 CFD 问题进行加速，国外开展的较早，有一些经验，但是时至今日，仍然停留在科学研究的层面，没有实用化、商业化的产品。

2003 年，斯坦福大学的 Ian Buck 等人对 ANSI C 进行扩展，开发了基于 Cg 的 Brook 源到源编译器，简化了 GPGPU 的开发。但 Brook 编译器效率不高，仍然使用 SIMD 模型进行编程，应用范围依然有限。[2]

2008 年斯坦福大学利用 Brook 对临界翼形进行模拟，得到最高 18 倍的加速。[3]

2008 年 Boise University 利用 CUDA 对三维方腔流的模拟发现，相比 CPU 的串行程序，单 GPU 平台上获得了 33 倍的加速比，四核 GPU 平台上得到 100 倍的加速。[4]

2009 年，明尼苏达大学的 Peter Bailey 等人用 Lattice Boltzmann 方法模拟流动，得到了比 4 核 CPU 高达 28 倍的加速。[5]

2010 年，University of Patras （希腊）的 Anotoniou 等人在 Muti-GPU 平台上实现了计算流体力学中的著名的 WENO 格式。[6]

国内的科研单位在 GPU 应用于计算流体力学方面也有一些研究。

2009 年开始，南京航空航天大学的张兵等人在 GPU 上实现了结构网格和非结构网格的隐式格式 CFD 并行计算。隐式格式的 CFD 计算将物理模型转化为矩阵，通过 GPU 进行矩阵求解。[7]

上海大学和其他的一些研究机构则在 GPU 上使用 LBM 方法进行 CFD 计算。[8] LBM 方法是近年来发展的一种不同于传统有限元、有限体积、有限差分法的计算方法，LBM 方法由分子动力学模型推出，计算具有局部性，非常适合 GPU 并行计算，国内外很多学者也在做类似的研究。LBM 方法理论基础趋近成熟，但是相关的工程应用还是很少。

中科院计算机网络信息中心超级计算中心一直紧跟国际上高性能计算的前沿，积极在新的高性能计算架构和高效率计算方法方面做出创新。中科院超级计算中心的董廷星和李森就进行了 GPU 上的计算流体力学问题的加速。董廷星在 GPU 上实现了结构网格有限差分法求解，[9] 李森在 GPU 集群上实现了有限差分法的加速，充分利用了 GPU 集群的运算效率。[10]

虽然 GPU 近几年大举进军高性能计算领域，但是 GPU 真正用于航空计算软件业的条件还不是很成熟，一方面，当前基于 GPU 平台上的计算流体力学研究以结构网格和 LBM 方法等易于在 GPU 平台上加速的方法为主。另一方面，航空动力学中广泛应用的非结构网格显式格式求解器在 GPU 上实现起来比较困难，加速效果也并不明显，而且国内外对 GPU 平台上非结构网格显式求解的研究也比较少。

另外，一些在工业中广泛应用的开源和商业 CFD 软件往往比较庞大，移植到新的平台非常困难，虽然以 NVIDIA 为首的硬件厂商努力推进，但是 NASA、ANSYS 等组织对 GPU 的重视程度还有待加强。而我们国家目前并没有成熟的国产 CFD 软件，在基于 GPU 的高性能计算方面也缺少核心技术，在这方面的研究更加落后。

不过仍有部分学者对非结构网格上的 CFD 求解感兴趣，2009 年 George Mason University 的 Andrew Corrigan 等人曾经尝试过使用 GPU 在非结构网格上有限元方法求解，加速 7.4 倍。[11]

1.3 本文的研究内容和出发点

目前在 GPU 上成功实现的 CFD 应用以结构化网格上的求解为主。单块结构化网格数据结构比较简单，数据排列比较整齐，有利于 GPU 多线程的合并访问，能够更有效地利用显存带宽，发挥 GPU 的高浮点运算性能的优势。而且结构化网格的程序，容易获得更高的加速比，更容易受到以 NVIDIA 为首的 GPU 制造商的支持和吹捧。

但是，结构化网格只能用于形状规则的图形的求解，适用范围比较窄。单块网格虽然很容易在单机上并行，但是多块结构化网格由于网格内部很难再划分，大规模并行时很难达到负载平衡。

非结构网格生成比较容易，能灵活地离散复杂的拓扑结构，易于自适应处理，容易进行并行划分。近些年流行的各种流体力学求解器以基于非结构网格求解的居多，但是基于 GPU 的非结构网格求解器还是很少，本文将重点分析非结构网格在 GPU 上加速的难点，并结合 CUDA 体系结构的特点，探索一些解决的方法。更重要的希望提炼出来在 GPU 上对非结构网格的求解进行性能优化的方法，为这一领域的研究者提供借鉴。

1.4 论文的组织结构

本文各章节的内容安排如下：

第一章 绪论。首先介绍计算流体力学和 GPU 通用计算，然后论述国内外相关课题的研究现状，并阐述本文的主要研究工作和出发点，最后给出本文的结构安排。

第二章 非结构网格上的有限体积法求解。介绍有限体积法，描述通过有限体

积法求解 CFD 问题的过程以及一些加速收敛的方法。介绍非结构网格的特点，以及在计算时生成的复杂的数据结构。

第三章 众核平台架构的介绍及特性。首先介绍了并行计算的基础知识和发展，GPU 作为通用计算的发展路程。详细介绍了 NVIDIA 的产品，包括 CUDA 的编程模型，线程层次以及存储层次，针对 GPU 的特点，介绍主要的优化方法。

第四章 众核平台程序结构及主要优化方法。列出 GPU 程序的结构并描述各部分的作用，详细介绍各种优化方法的使用情况及效果，其中包括针对非结构网格程序特点所做的优化。

第五章 性能测试及分析。选取一些算例，对优化后的程序进行性能分析，包括各种优化方法对性能的影响。

第六章 总结与展望。对本论文所做的工作给予全面的归纳总结，并指出目前工作的不足之处和未来的工作方向。

第二章 非结构网格上的有限体积法求解

2.1 控制方程

描述流体运动基本规律，包括质量、动量和能量守恒律的数学方程组是计算流体力学研究问题的出发点，CFD 中最常用的方程为 NS 方程和 Euler 方程。下面给出两种方程的表现形式，推导过程可参考《流体力学》[13] 。

2.1.1 Navier-Stokes 方程

$$\frac{\partial}{\partial t}\int_{\Omega}\boldsymbol{W}d\Omega+\int_{\partial\Omega}\boldsymbol{F}\cdot\vec{n}ds=\int_{\partial\Omega}\boldsymbol{F_v}\cdot\vec{n}ds$$

其中，$\boldsymbol{W}$为守恒变量组成的一个矢量(密度、密度与 x 方向速度的乘积、密度与 y 方向速度的乘积、密度与能量的乘积)，$\boldsymbol{F}$为无粘通量矢量，$\boldsymbol{F_v}$为粘性通量矢量，$\boldsymbol{\Omega}$为控制体，$\partial\boldsymbol{\Omega}$为控制体单元的边界，$d\boldsymbol{\Omega}$为体积微元，$d\boldsymbol{s}$为面积微元。

N-S 方程是一个非线性方程组，求解 N-S 方程需要考虑粘性通量。

2.1.2 Euler 方程

如果流体的雷诺数足够大，并且我们所感兴趣的只是宏观流动时，可以忽略粘性效应，这样的流动可以看做无粘流动。如飞行器绕流的流场，若雷诺数足够大，物面附近粘性边界层很薄，它对飞行器表面压力分布及定常气动特性的影响可以忽略。[14] 这时 N-S 方程右端的粘性项为 0，方程简化为：

$$\frac{\partial}{\partial t}\int_{\Omega}\boldsymbol{W}d\Omega+\int_{\partial\Omega}\boldsymbol{F}\cdot\vec{n}ds=0$$

其中，$\boldsymbol{W}$为守恒变量组成的一个矢量(密度、密度与 x 方向速度的乘积、密度与 y 方向速度的乘积、密度与能量的乘积)，$\boldsymbol{F}$为无粘通量矢量，$\boldsymbol{\Omega}$为控制体，$\partial\boldsymbol{\Omega}$为控制体单元的边界，$d\boldsymbol{\Omega}$为体积微元，$d\boldsymbol{s}$为面积微元。

2.2 有限体积法

2.2.1 空间离散格式

在对偏微分方程进行数值求解时，我们常用的离散方法有：有限差分法和有限体积法。为了便于处理物面边界条件，我们将微分方程的差分表达式，转化为等价的积分方程的代数表达式，这种代数表达式适用于任意的曲线坐标网格，包括以物面为边界的曲线网格。积分方程的代数表达式是以空间的体积元素为对象

进行离散化的，这种方法叫做有限体积法。

迎风格式在目前 CFD 计算方法中被广泛采用，与中心差分格式相比，它具有更好的激波、接触间断捕捉能力。迎风格式包括通量矢量分裂(FVS)和通量差分分裂(FDS)两种，本文中采用的是 Roe 格式，属于 FDS。Roe 格式表述如下：

$$\boldsymbol{F}=\frac{1}{2}(\boldsymbol{F}_L+\boldsymbol{F}_R)-\frac{1}{2}\boldsymbol{A}\cdot(\boldsymbol{W}_L+\boldsymbol{W}_R)$$

其中：$\boldsymbol{F_L}$和$\boldsymbol{F_R}$分别为控制单元表面法向方向两侧的无粘通量（矢量），$\boldsymbol{A}$是采用 Roe 格式平均变量计算的流通量雅可比矩阵，$\boldsymbol{W_L}$和$\boldsymbol{W_R}$分别为控制单元表面两侧的守恒量矢量。

2.2.2 时间积分格式

时间积分格式采用四阶龙格库塔法。

经典的具有四阶精度的龙格库塔法为：

$$\boldsymbol{W}_{i,j}^{(0)}=\boldsymbol{W}_{i,j}^{n}$$

$$\boldsymbol{W}_{i,j}^{(1)}=\boldsymbol{W}_{i,j}^{(0)}-a_1\,\Delta\, t\boldsymbol{P}_{i,j}^{(0)}$$

$$\boldsymbol{W}_{i,j}^{(2)}=\boldsymbol{W}_{i,j}^{(0)}-a_2\,\Delta\, t\boldsymbol{P}_{i,j}^{(1)}$$

$$\boldsymbol{W}_{i,j}^{(3)}=\boldsymbol{W}_{i,j}^{(0)}-a_3\,\Delta\, t\boldsymbol{P}_{i,j}^{(2)}$$

$$\boldsymbol{W}_{i,j}^{(4)}=\boldsymbol{W}_{i,j}^{(0)}-a_4\,\Delta\, t\boldsymbol{P}_{i,j}^{(3)}$$

$$\boldsymbol{W}_{i,j}^{n+1}=\boldsymbol{W}_{i,j}^{(4)}$$

其中，$\boldsymbol{W}_{i,j}^{n}$，$\boldsymbol{W}_{i,j}^{n+1}$是第 n 步时间上的初值和终值。

由于龙格库塔格式对残值要做四次计算，且中间变量必须保存，若采用细网格进行计算时要占用大量的计算机内存，因此该格式的效率并不高。为此，我们使用简化多步格式。

简化的龙格库塔格式：[23]

$$\boldsymbol{W}_{i,j}^{(0)}=\boldsymbol{W}_{i,j}^{n}$$

$$\boldsymbol{W}_{i,j}^{(1)}=\boldsymbol{W}_{i,j}^{(0)}-a_1\,\Delta\, t\boldsymbol{P}_{i,j}^{(0)}$$

$$\boldsymbol{W}_{i,j}^{(2)}=\boldsymbol{W}_{i,j}^{(0)}-a_2\,\Delta\, t\boldsymbol{P}_{i,j}^{(1)}$$

$$\boldsymbol{W}_{i,j}^{(3)} = \boldsymbol{W}_{i,j}^{(0)} - a_3 \Delta t \boldsymbol{P}_{i,j}^{(2)}$$

$$\boldsymbol{W}_{i,j}^{(4)} = \boldsymbol{W}_{i,j}^{(0)} - a_4 \Delta t \boldsymbol{P}_{i,j}^{(3)}$$

$$\boldsymbol{W}_{i,j}^{n+1} = \boldsymbol{W}_{i,j}^{(4)}$$

其中，四步格式的系数为：a_1=0.0833, a_2=0. 2069, a_3=0. 4265, a_4=1.0。

2. 3 非结构网格

结构化网格是指网格区域内所有非边界点都有相同的邻居单元，内部点不需要存储邻居的编号信息，可以容易地实现区域的边界拟合。结构化网格出现的很早，是和当时的计算硬件以及算法的发展是有关系的。当时计算机内存比较小，只能进行小数量网格的计算，而当时的计算算法，大多采用的是差分算法，因此采用结构化网格是非常合适的。然而，现实世界的复杂性对网格提出了更高的要求。比如，复杂的几何外形，如果采用结构网格，往往难以进行剖分。这时，计算机硬件的飞速发展，大容量内存的出现，导致了非结构网格的诞生。

非结构化网格是指网格区域内的内部点不具有相同的毗邻单元，即与网格剖分区域内的不同内点相连的网格数目不同。非结构网格的主要优势在于几何适应性好，毫不夸张的说，没有非结构网格适应不了的外形。非结构网格其他的优点是网格生成比较容易，易于自适应处理，容易进行并行划分，而且可以在连续区域内实施网格密度控制。经过 40 多年的研究积累，通过多重网格法、隐式残差光顺等加速措施，非结构网格已获得了与结果网格相比拟的收敛速度。而且近些年流行的动网格技术，在非结构网格上也比较容易实现。

当然，非结构网格的出现于算法的发展也是分不开的。固体计算中，有限元法逐渐取得了优势；流体计算中，有限体积法也逐渐地确定了领导地位；这两种方法都可以使用非结构网格。

事实上，结构化网格和非结构网格的主要差别并非外形，而是体现在数据结构上，关键看数据结构中是否存储有邻居单元的信息。结构化网格所有的内部点具有相同的邻居单元，而非结构化网格没有这个特征，事实上，结构化网格也可以是三角形、四面体等形状（如图 1 ），只要按照一定的规律排列，数据结构中就不用存储邻居信息，仍然可以视为结构化网格。而所有的结构化网格都可以以非

结构网格的数据结构进行存储。

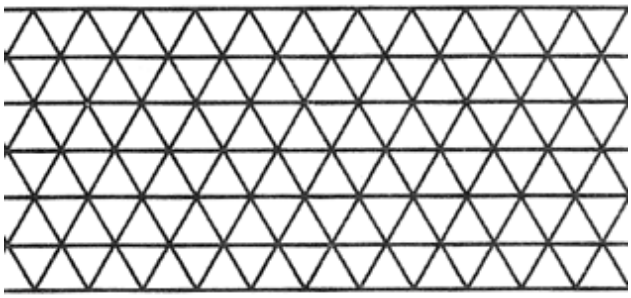

图1 三角形网格也可以作为非结构网格进行存储

非结构网格也有一些缺点，比如无法很好的处理粘性问题，比如网格填充效率不高等等。但是综合来看，非结构网格的优势还是很大的，比如网格节点间无需固定规划，在网格自适应方面，非结构网格的能力要远强于结构化网格。随着计算机技术的发展，非结构网格的一些缺点也能在一定程度上得到改善。比如粘性问题，常常采用棱柱网格予以解决。而填充效率低主要体现在生成的网格数量过多，但是随着计算机技术的进步，大容量的存储器和高性能的计算机的使用，这些都不是问题。

而计算精度问题，主要在于网格的质量（正交性、长宽比等），并不取决于采用结构化还是非结构化网格。事实上，有些软件，比如 fluent，无论输入什么样的网格，在求解器内部都按照非结构网格重新组织数据结构，使用非结构网格进行处理。

近几十年高性能计算机和数值计算方法快速发展，高性能的计算设备和方法可以处理更加复杂和更大的区域，对多块结构化网格的生成和计算提出了新的要求。单块网格虽然很容易在单机上并行，但是多块结构化网格由于网格内部很难再划分，大规模并行时很难达到负载平衡。

近些年流行的各种流体力学求解器以基于非结构网格求解的居多，基于 GPU 的非结构网格求解器很少，在 GPU 上成功实现的 CFD 应用以结构化网格上的求解为主。单块结构化网格数据结构比较简单，数据排列比较整齐，有利于 GPU 多线程的合并访问，能够更有效地利用显存带宽，发挥 GPU 的高浮点运算性能的优势。

本文的研究就是主要基于非结构网格的有限体积求解，采用常见的三角形网

格。

图 2 为 RAE2822 翼型的三角形网格：

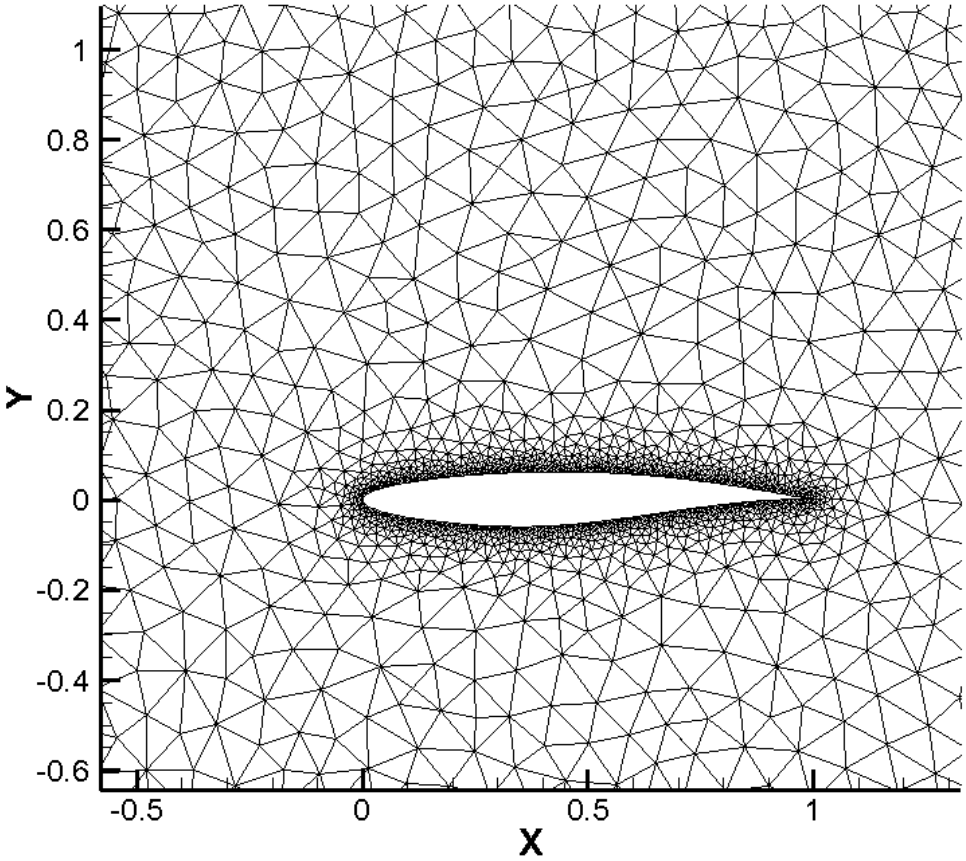

图2 RAE2822 翼型三角形网格

下图为三角形网格的结构：

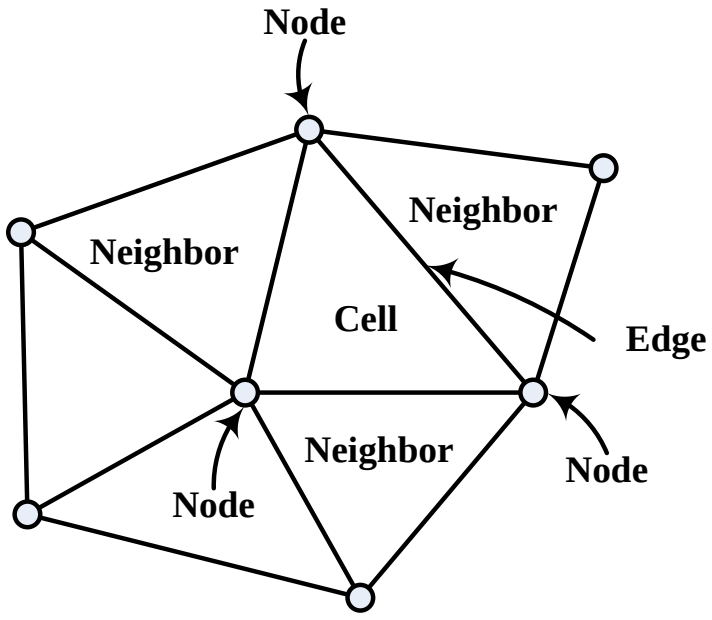

图3 非结构三角形网格示意图

由图 3 可以看出，三角形网格的主要数据结构由三角形元胞、点、边组成。由于非结构网格没有规则的拓扑关系，无论以元胞、点、边中的任何一个为主要顺序，另外两组数据都不是顺序存储的。

下面提供三种主要数据结构的结构体，每个元胞与三个点相邻，与至多三个元胞相邻，与三条边相邻；每个点与未知个数的边和元胞相邻；每条边与两个点相邻，与至少一个元胞相邻。

以下是元胞（Cell）的数据结构体。

```
typedef struct Neighbor
{
	int		celledge;	// 相邻的边的编号
	int		neicell;	// 相邻的 Cell 的编号
}Neighbor;

typedef struct CELL
{
	int		CLog;
	int		Point[3];	// 相邻的点的编号
	double	deltU[2][4];
	double	Umax[4],Umin[4];
	double	center[2];	// 中心点位置
	Neighbor	neighbor[3];	// 三组邻居（包括边和元胞）
}CELL;

typedef struct W	// W 中存储着 Cell 中的物理量(密度、压强等)
{
	double density;		// 密度
	double density_U;	// X 方向的动量密度
	double density_V;	// Y 方向的动量密度
	double density_E;	// 能量密度
	double P;		// 压强
	double T;		// 温度
	double A;		// 声速
}W;
```

以下是点（Node）的数据结构体。

```
typedef struct NODE
{
	int	NLog,Nlog1;
	double	x;		// 点的位置
```

```
        double y;              // 点的位置
        double ROU,U,V,E;
        double P,T,A;
        double Mach;
        double RoundArea;
}NODE;
```

以下是边（Edge）的数据结构体。

```
typedef struct EDGE
{
        int    ELog,Elog1;
        int    left_cell; // 左边的单元编号
        int    right_cell;// 右边的单元编号
        int    farfieldid;// 如果某一边是远场，则表示远场的编号
        int    wallid;    // 如果某一边是固壁，则表示固壁的编号
        int    node1;     // 相邻的点的编号
        int    node2;     // 相邻的点的编号
        double vectorn;
        double vectorx;
        double vectory;
        double midx,midy;
}EDGE;

typedef struct RESD      // RESD 中存储着 Edge 中的物理量(流通量)
{
        double data[4];
}RESD;
```

2.4 程序流程

CPU 上的有限体积求解程序是一个守恒量与流通量循环迭代的结构（如图 4 ），首先加载边界条件，然后在元胞上根据流通量求出守恒量对时间的积分，再根据守恒量求出流通量，如此循环迭代。对于非结构网格的求解程序而言，在从守恒量求解流通量的过程中，需要根据元胞上的密度、动量、能量等物理量求出边上的流通量，并将流通量插值回元胞，这个过程以边为顺序进行计算，在写回元胞的过程中，会产生写冲突，后面并行时我们采用了一些方法来避免这些写冲突。

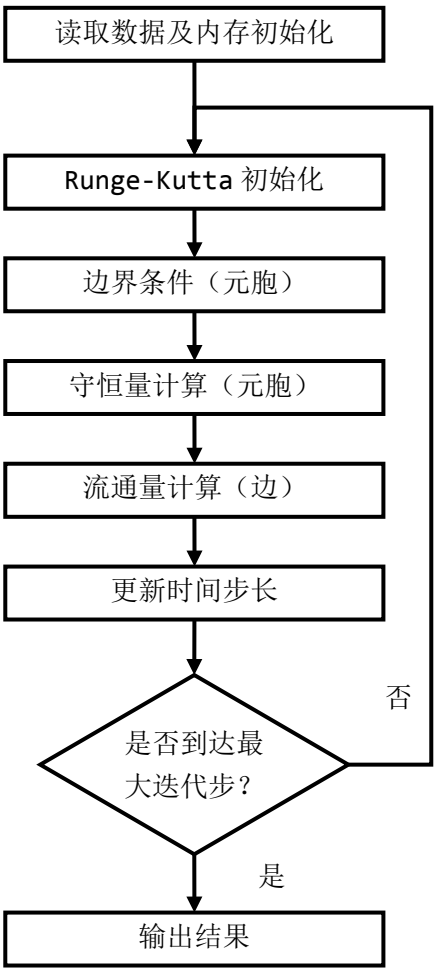

图4 串行程序流程图

下图为串行程序的函数调用关系图。

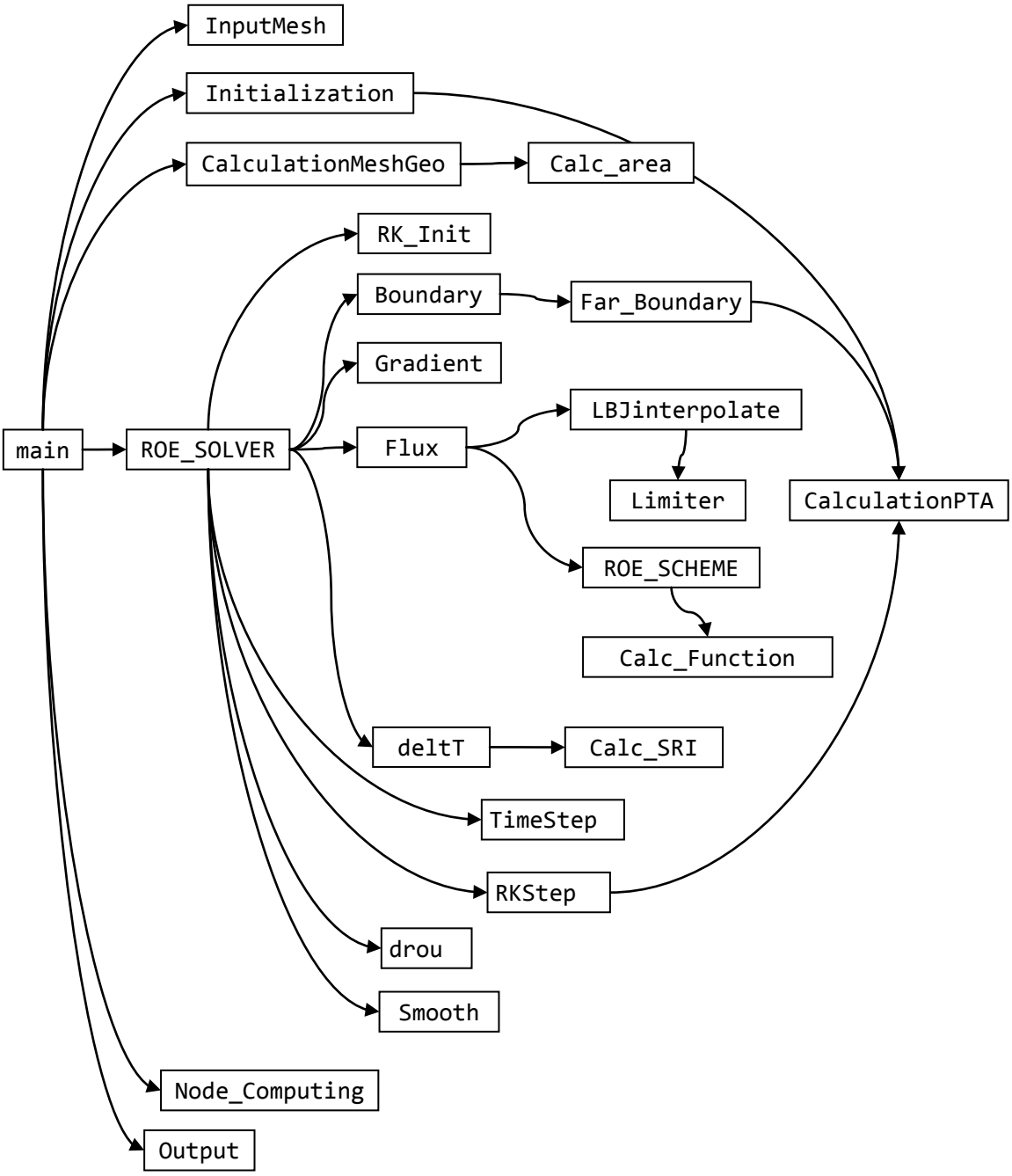

图5 串行程序函数调用图

InputMesh、Initialization、CalculationMeshGeo：初始化网格和数据，计算元胞的物理量。

ROE_SOLVER：迭代求解

RK_Init：Runge-Kutta 初始化

Boundary：元胞的边界条件，其中包括远场边界条件和固壁边界条件。远场边界条件比较复杂，需要根据流入流出量来计算边界元胞的守恒量。

Gradient：计算每个元胞守恒量的散度。

Flux：根据元胞上的守恒量计算每条边上的流通量，这一步骤分为两部分，第一部分 LBJinterpolate 根据边与左右元胞的距离在元胞中插值新的守恒量，第二部分就是 Roe 格式的计算，根据每条边左右两边的元胞中刚刚计算出的守恒量，计算边上的流通量，并将流通量加到两边元胞的残差里。

RKStep：在元胞中将残差加到守恒量上，从而更新守恒量。

drou：计算全局残差。

Smooth：残差平滑。

2.5 加速收敛方法

数值模拟过程耗时巨大，评价一个模拟过程是否稳定的一项重要指标就是残差曲线，理想情况下，全局残差应该随着迭代步数的增加收敛到 0。由于数值计算是一个离散过程，计算机内存储的数据也是离散的数据，计算过程中难免会发生数值耗散，一般情况下残差会收敛到一个很小的值或者在很小的范围内波动。残差收敛的过程是比较漫长的，因为数值模拟本质上是一个通过迭代不断逼近稳定解的过程，迭代的步数与时间步长、空间步长、网格数量等多种因素相关，残差曲线一般下降的比较慢，有时一些数值耗散会导致残差无法完全收敛。数值模拟中发展出来很多加速收敛的方法，如多重网格法、局部时间步长法、残差平滑等等。本文的工作用用了局部时间步长和残差平滑两种方法加速收敛，这两种方法的实现比较简单，加速效果也非常明显。

2.5.1 局部时间步长

显格式的优点是在每个时间步长计算中不需要求解线性方程组，但是为了保证数值计算格式的收敛性和稳定性，时间和空间步长要满足 CFL 条件。CFL 条件规定，时间步长与空间步长成正比，当网格单元的尺度差别很大时，统一的时间步长会大大降低显格式的计算效率。因此，我们使用了局部时间步长法，它是一种时间步长随空间步长改变，满足局部 CFL 条件但不一定满足全局 CFL 条件的方法。[22]

在数值求解时使用局部时间步长，即各元胞不采用全部流场统一的时间步长，而是采用局部的稳定性和收敛性所允许的最大时间步长，流场处处以稳定性极限

向前推进，这样能够加速流场向定常解收敛，但是不同点的时间步长的不一致性，这样收敛前的数值结果并不代表同一时刻的流场，求解的中间过程没有明确的物理意义，不过收敛的结果一般仍然是所求的定常解，这种加速收敛的机理是加快扰动的传播速度。（图 6 和图 7 是 3558 网格量时使用全局时间步长和局部时间步长的残差曲线，可以看到，使用全局时间步长大约在 7000 步残差才收敛，而使用局部时间步长，大约 6000 步残差就能收敛。）

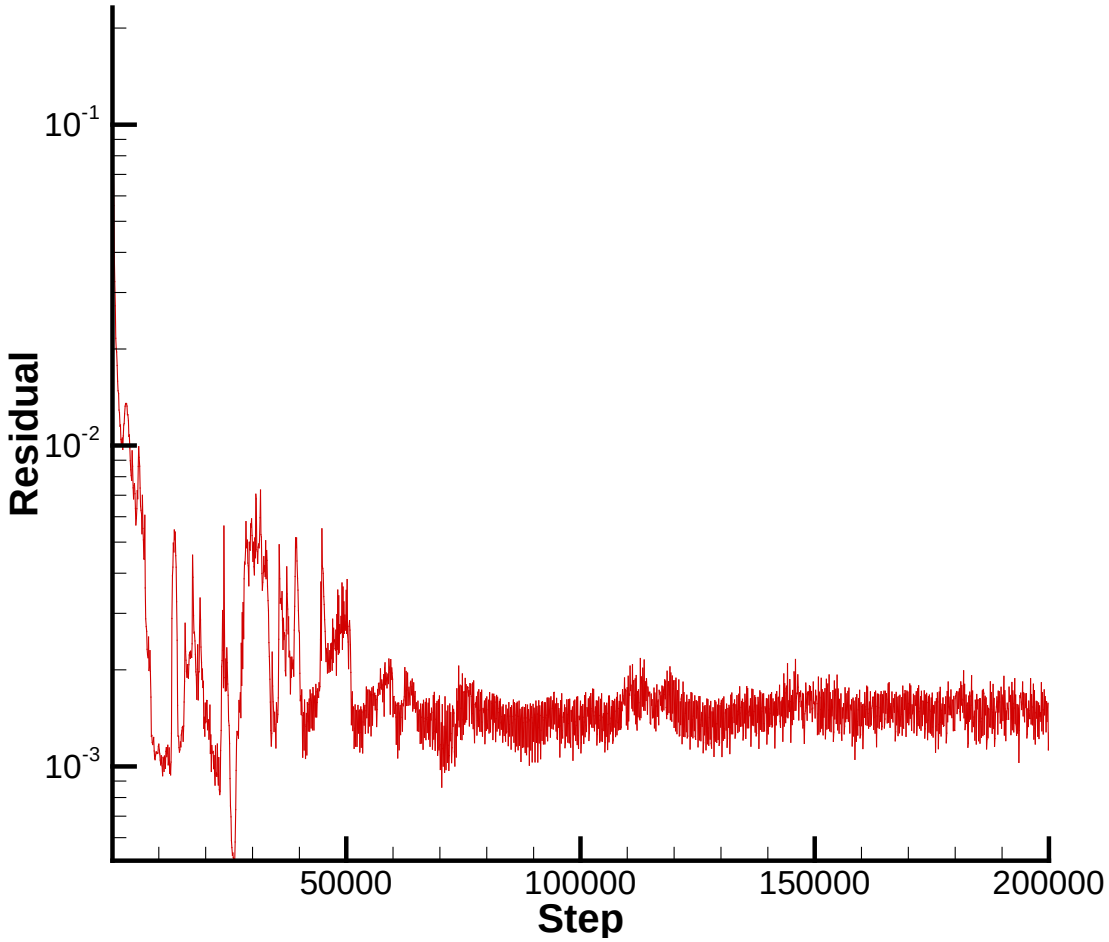

图6 使用全局时间步长的残差曲线

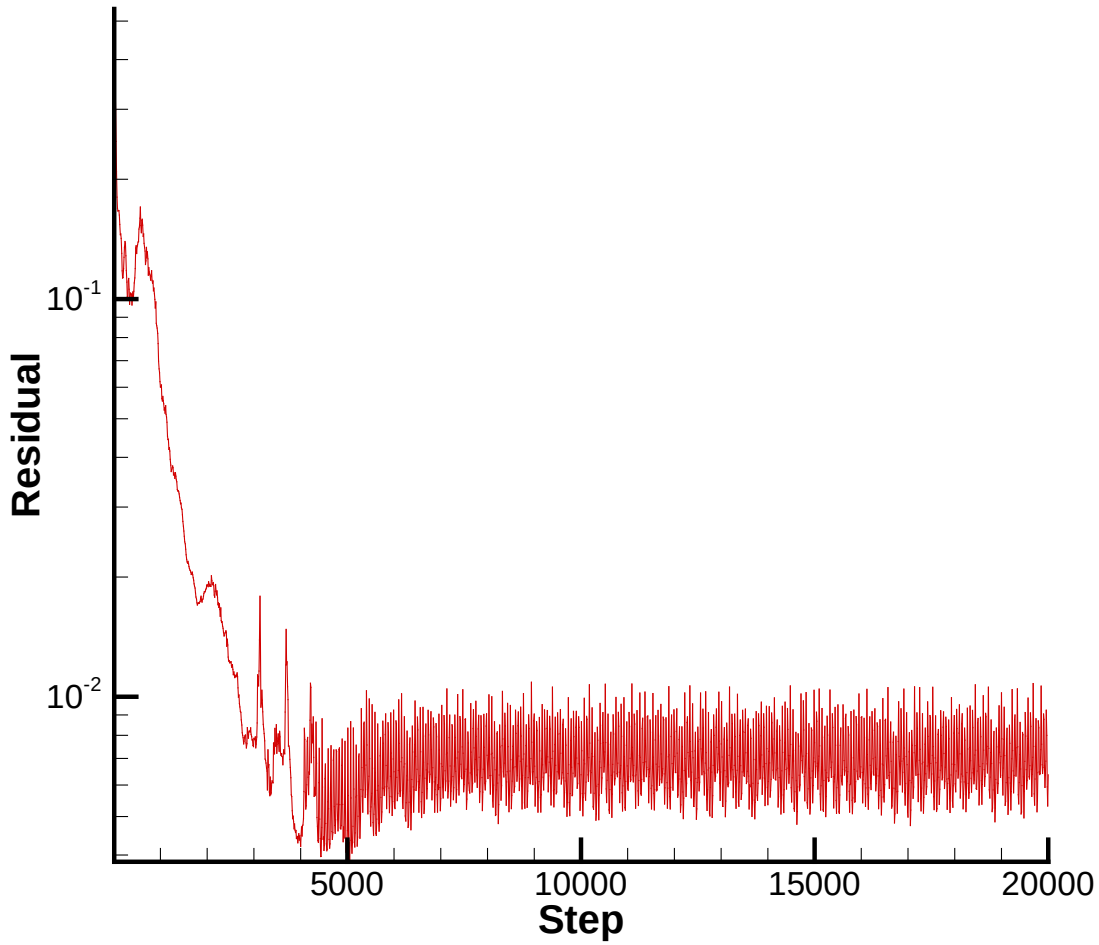

图7 使用局部时间步长的残差曲线

2. 5. 2 残差平滑

Jameson 和 Baker 提出了一种应用于显式格式残差平滑的技术，可以提高稳定条件的 CFL 数，加速收敛过程。[25] 残差平滑方法可以显式或者隐式的使用。我们这里使用了非结构网格中的中心格式隐式残差平滑法（Central Implicit Residual Smoothing, CIRS）。

经过平滑后的残差$\vec{\boldsymbol{R}}_{\boldsymbol{I}}^{*}$由下式给出：[24]

$$\vec{\boldsymbol{R}}_{\boldsymbol{I}}^{*}+\sum_{\boldsymbol{J}=\boldsymbol{0}}^{\boldsymbol{N_A}}\boldsymbol{\epsilon}\left(\vec{\boldsymbol{R}}_{\boldsymbol{I}}^{*}-\vec{\boldsymbol{R}}_{\boldsymbol{J}}^{*}\right)=\vec{\boldsymbol{R}}_{\boldsymbol{l}}$$

其中N_A代表每个元胞周围的所有其他元胞的个数，平滑系数$0.5 \leq \epsilon \leq 0.8$，通过这个式子，进行雅可比迭代，求出$\vec{\boldsymbol{R}}_{\boldsymbol{I}}^{*}$，雅可比迭代次数一般为 2 次。在本文的工作中，残差平滑技术对收敛过程中残差曲线的高频振荡有很好的抑制作用。（图 8 为采用残差平滑之后的残差曲线，与图 7 的曲线相比，曲线平滑性较好。）

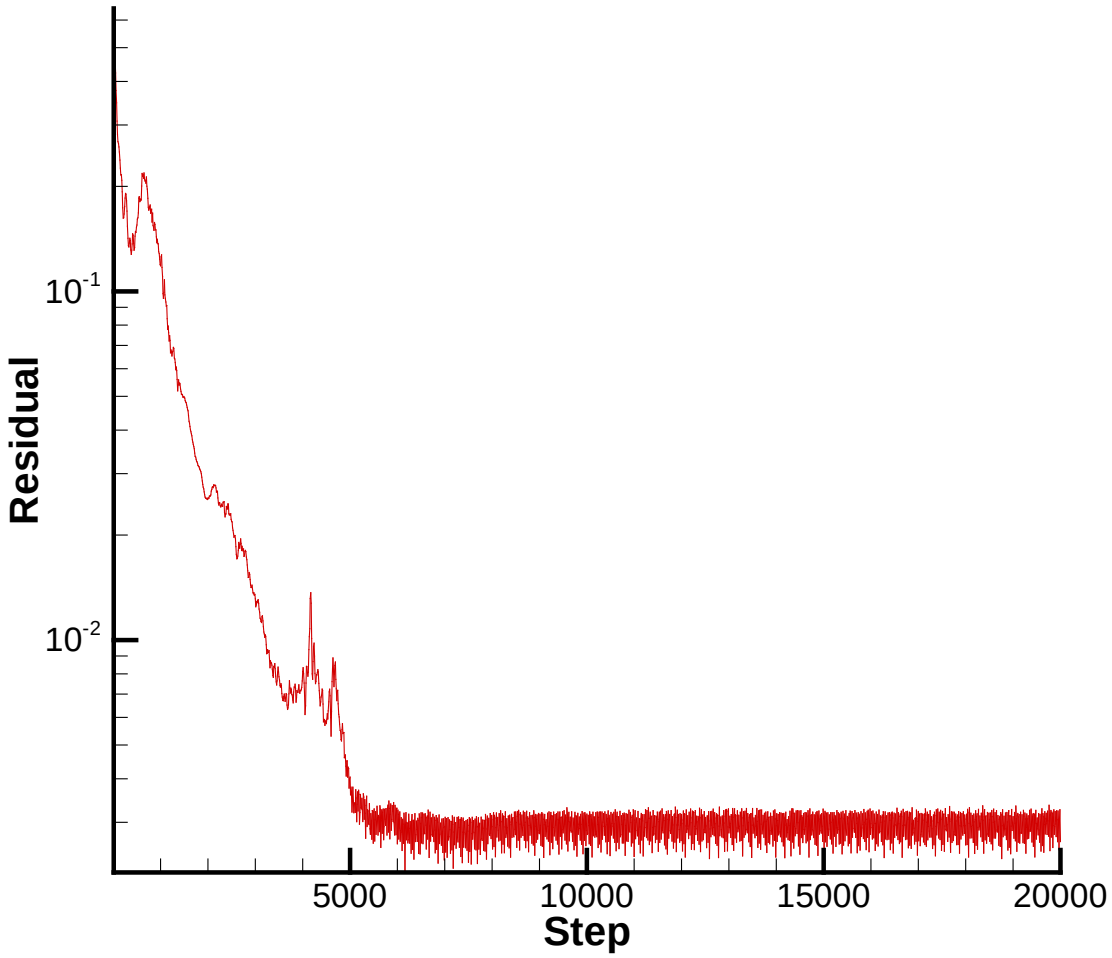

图8 采用残差平滑之后的残差曲线

第三章 众核平台架构介绍及特性

3.1 单线程串行计算的瓶颈

在应用中，我们经常需要比串行计算机所能提供的能力更强的计算能力。克服这种限制的一种方法是增加处理器和其他部件的运算速度，以使他们能够提供应用所需的更强大的计算能力。传统的提高 CPU 性能的主要手段是提高单个处理器的工作频率，现在看来，尽管这在一定程度上是可能的，但未来的发展还是受到了光速、热力学定律和资金的限制。[15] 随着制造工艺的提高，晶体管的尺寸接近原子量级，能耗和发热量居高不下，依靠提高单个核心的频率的来提高性能的方法越来越难。在芯片越来越小，工艺越来越好，单核频率越来越高的过程中，半导体厂商也一直在探索其他的提升性能的方法，其中的一个方法就是并行。（并行并不是最好的方法，并行其实是天才们解决问题时的一个无奈的选择，如果天才们能突破前面提到的三个限制－－光速、热力学定律、钱－－中的任何一个的话，他们都不会选择并行的。）

并行计算（parallel computing）是指在并行计算机上，将一个任务分解为多个子任务，分配给不同的处理单元，各个处理单元之间相互协同，并行地执行子任务，从而达到加快处理速度，或者提高求解应用问题规模的目的。

并行计算的三个基本条件是：并行计算机、并行算法、并行计算任务。我们生活和工作中遇到的大多数问题都是并行任务，计算流体力学求解也不例外。这里着重讲一下并行计算机以及并行算法。

3.2 并行计算技术的分类以及发展

并行计算技术可分为指令级并行、线程级并行、任务级并行。

3.2.1 指令级并行

指令并行（Instruction-Leverl Parallelism，ILP）又可以根据空间和时间两个方面划分。[16]

空间并行技术主要是 SIMD 处理，SIMD 即 Single Instruction Multiple Data，单指令多数据流，早期的 SIMD 机器由多个相同的处理单元组成，如世界上第一台并行机 ILLIAC IV，它有 32 个处理单元，每台处理单元有局部内存。后来出现了很多变种，后来由于种种原因，向量处理机成功的存活下来并继续发展，而其

他的很多变种如昙花一现，只在历史舞台上留下了一个影子。

向量处理的代表产品是 CRAY-1，CRAY-1 是一个向量机，向量机的特点是有一个很宽寄存器（CRAY 为 64*64bits），能够在一个时钟周期内同时执行 64 个浮点运算。向量机中，一个寄存器存储着一列数据，能够同时执行一个相同的指令，以提高计算效率。向量机具有深远的影响意义，目前主流半导体厂商的向量运算技术，如 Intel 的 MMX、SSE、AVX，AMD 的 3DNOW!技术等，包括主流 GPU 的并行计算技术，都借鉴了向量机的思想。图 9 为 CRAY-1 的处理器结构。

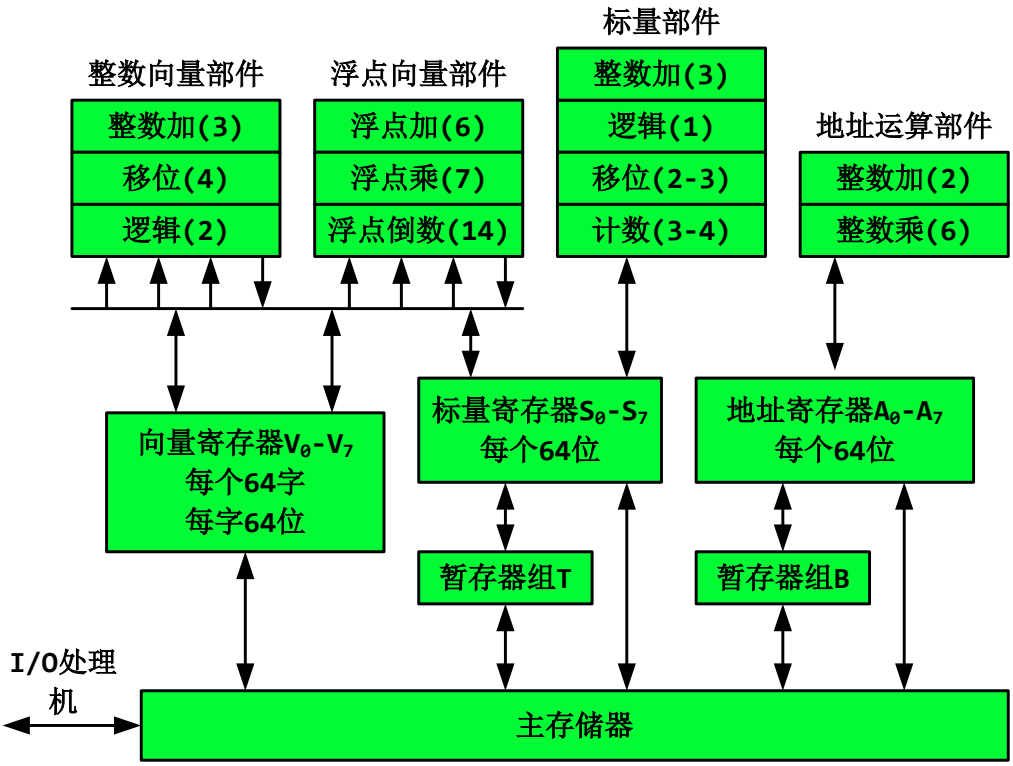

图9 CRAY-1 处理单元的结构

时间并行技术主要是流水线技术，指令的执行一般可以分为取指（IF）、译码（ID）、执行（EX）、访存（MEM）、写回（WEB）等过程。整个指令执行周期一般会比较长，流水线技术能够在指令执行过程中重叠不同的阶段，使执行速度加快。[19] 流水线技术最早出现在 RISC 结构的处理器上，代表产品是 MIPS R4000（龙芯）。

有了流水线之后，20 多年来，体系结构的主要设计目标就变成了怎样填满流水线，喂饱处理单元。

在流水线基础上发展了几种改进的时间并行的方法：乱序执行技术，超流水

线技术，超标量流水线技术，超长指令字技术和分支预测技术。

1. 乱序执行（动态流水线）：一个已经编译好的程序中，数据相关性决定了指令级并行在多大程度上能够得到应用。一个简单的静态调度流水线负责取指令并发射指令，当刚取到的指令与已经在流水线中的指令之间存在数据相关，且不能通过旁路技术或直接通路技术来避免时，就发生取指令和发射指令停顿，如果存在实在不可避免的数据相关，那么检测冲突的硬件将禁止有相关的指令及其后面的指令进入流水线，相关性被消除之前不会再取出和发射新的指令。

而乱序执行技术能够由硬件动态调整指令执行顺序以减少停顿的影响，它能够处理某些在编译阶段无法知道的相关关系（如涉及内存引用时），并简化编译器设计。在动态调度流水线中，指令在发射阶段都是顺序的（顺序发射），但从读操作数阶段开始就可以提前或者推迟执行（乱序执行），代表产品 Pentium Pro（1995）。

乱序执行技术最早出现于 CDC 6600（1974）。IBM 360/91（1969）中第一次使用了 Tomasulo 算法，现代所有乱序执行技术的处理器全部使用的 Tomasulo 算法的变种，他们的做法是跟踪指令的相关性，使得指令所需要的操作数一准备好就允许指令执行，同时运用寄存器重命名技术来避免读写和写写冲突。一些其他的技术如寄存器重命名等也随着乱序执行的发展而产生。

2. 超流水线：超流水线技术就是将流水线内部的每一个流水段进一步细分，使指令能在其中以更快的速度通过，甚至在一个时钟周期内能流出多条指令。当然，超流水线和流水线之间并没有明显的区别。

3. 超标量流水线：超标量流水线是指处理器中有两条或多条流水线，能够同时发射多条指令。（让一条指令从译码（ID）流动到（执行段）的操作通常称为发射指令。[18] ）超标量流水线可以通过编译器进行静态调度，也可以基于 Tomasulo 算法进行动态调度。代表产品有 SUN 的 SPARC 处理器。[21]

4. 超长指令字：超长指令字处理器（Very Long Instruction Word，VLIW）将并发执行的指令并入到一个组里，作为一个超长的指令字（一百位到几百位），VLIW 的并行执行基于一个确定的调度，由编译器静态完成，处理器减少了指令调度的复杂性。（同时提高了编译器的复杂性，而且这种并发执行的方法只有当编译器能够生成有效代码时才有用）近些年，随着 64 位处理器的流行，VLIW 技术重

新走入历史舞台，代表产品有 AMD 的 Opteron 和 Intel 的 Itanium。

5. 分支预测：程序中一般会包含很多的条件控制结构（if、case、goto 等），这些结构都有可能造成流水线加载无效的指令，当执行到一个条件分支指令时，只有在执行点才知道分支将转向何处，对于流水线来说，准确的分支预测是相当重要的。分支预测是建立在乱序执行的技术基础上的，通常使用预测调度表来处理，就是将分支执行历史数据存入一个表中，根据表中的数据推算下一次分支转移的结果。分支猜测技术经过 20 多年的发展，已经有很高的预测准确率，Alpha 21264（申威）是分支预测技术的代表产品。

3.2.2 线程级并行

线程级并行(Thread-Level Parallelism，TLP)是指在一个处理器内同时执行多个相同或者不同的指令，达到加速的目的。

使用多条流水线只能增加处理器运算的效率，不能增加运算的理论峰值，因为流水线属于调度单元，运算最终还是由执行单元运行的。于是，从 20 世纪 80 年代开始，处理器中都有多个运算执行单元。多个运算单元能够以很少的代价增加理论峰值，但是对流水线的设计提出了新的挑战。事实上，多执行单元出现的比多发射要早，多执行单元与多发射是共同发展，相辅相成的。

CPU 中多个线程可以并发执行，多线程并发执行并不能增加 CPU 的理论峰值，但是由于多个线程往往互不相关，指令之间没有依赖，可以更有效地利用流水线。

线程级并行技术的代表产品是 Intel 的 Pentium 处理器，Pentium 处理器于 1993 年正式发布，是 Intel 的一个突破性产品，它是一种 RISC 和 CISC 结合的产品，采用了超标量流水线整数处理、超流水线浮点处理，分离式指令 Cache 与数据 Cache，最重要的是，Pentium 处理器有两个整数运算单元和一个浮点运算单元，能够独立执行两个硬件线程。（Pentium 并不支持指令乱序执行，所以它的指令执行效率并不高，Pentium 只是一个昙花一现的过渡产品，Intel 很快推出了带乱序执行的升级产品，并命名为 Pentium Pro。）

3.2.3 任务级并行

任务级并行是指将计算任务划分为几部分，每一部分由并行计算机的一个或多个结点完成，这个结点可以是一个集群、或者一台计算机、或者一个处理器、

或者一个处理器核心。任务级并行是最常见也是最容易理解的并行方式，但是在并行计算机的发展史上出现的并不早，主要原因是多个结点之间的通信一直是设计任务级并行计算机的瓶颈。

这一类计算机出现于 20 世纪 70 年代，当时的多向量处理机可以认为是任务级并行计算技术的雏形。但是任务级并行真正开始大发展还是得益于多任务操作系统的出现，1970 年，UNIX 诞生，从此，并行计算从单纯的硬件技术变成了软件与硬件相结合的技术。

任务级并行的代表产品有很多，20 世纪 90 年代，一些北桥芯片即可以支持多路处理器，2000 年 IBM 发布 Power4，这是第一款商业化的多核处理器，但是 2005 年之前的多核处理器两个核心之间的通信还需要北桥的参与，后来各大厂商纷纷推出了多核心深度融合的产品。如今的各种超级计算机，大部分都是将通用的处理单元（CPU、刀片）连接起来，通过互联网络并行计算，这种计算方式也是典型的任务级并行。

3.3 并行算法的分类和发展

并行算法的分类方法比较多，我个人比较倾向于以下这种分类方法：[17]

1. 几何分解：问题范围被分解为更小的域，每一个进程执行其中一个部分的算法。

2. 迭代分解：有一些程序建立在循环执行之上，在这些循环中每一次迭代可以独立地进行。这种方法用一个包含可执行任务的中央队列实现，它相当于任务播种并行。

3. 递归分解：这种策略首先将原始问题分解为一些子问题，然后在并行地解决。这种方法相当于分治策略。

4. 投机分解：一些问题可以使用一种投机分解方法：同时用 N 种解决方法进行尝试，当其中的某个首先返回一个正确值后，其余的（N-1）个都将被抛弃。在某些情况下可能会缩短整个任务的执行时间，达到优化目的。

5. 功能分解：程序被分为许多不同的阶段，每个阶段在相同的问题中执行不同的算法，比如流水线。

并行算法的几个重要发展阶段可分为：[12]

1. 基于向量机的并行算法

这类算法是 70 年代末和 80 年代初，随着第一代向量机的出现而出现的，而支持向量计算的编程语言如 fortran 等对这类算法也有推动作用。值得一提的是，在这个时期，我国在向量机方面有很好的研究和应用成果（银河 I），这个时期是我们国家高性能计算发展的巅峰时期。

2. 基于多向量机的并行算法

这类算法随着多向量处理机的产生而流行于 80 年代，这类算法既要考虑单个处理机上的向量并行，又要考虑多个处理机之间的并行，这类算法为今后 MIMD 类算法的发展打下了很好的基础，目前日本仍在做这方面的研究。

3. SIMD 并行机上的并行算法

这类算法运行在 SIMD 类并行机上，但是由于 SIMD 指令没有统一的标准，这类算法对硬件依赖性很大，所以并没有流行很长时间。但是 SIMD 的思想从这时深入人心，CUDA 编程模型中的 GRID 就可以看做一个 SIMD 并行机，CUDA 编程模型就是由多个 SIMD 并行机组成的集合。

4. MIMD 并行机上的并行算法

80 年代初期开始，单个处理器功能已经比较成熟了，能独立处理各类复杂运算，而且随着多机通讯协议的产生和发展，把多台处理机绑在一起变得更加容易，各类多机组合的并行机开始出现。MIMD 类的算法开始兴起，处理器之间通过通信来交换数据和同步，这类算法通常用作大粒度的并行。

5. 现代并行算法

近些年来，大部分超级计算机以 SMP 与 cluster 相结合的结构出现，并行算法的设计仍以 MIMD 为主流，并要求具有可扩展性和可移植性。但随着 CPU、众核计算和网络技术的发展，高性能的算法必须要求并行算法的设计满足多个发展需求：第一，粗粒度任务级并行和细粒度并行相结合；第二，在每个进程，组织便于向量计算的数据结构、算法设计和通信方式。目前在大规模集群上广泛使用的 MPI+OpenMP 以及 MPI+Pthread，甚至 GPU 集群上的 MPI+CUDA 都有这部分思想。

算法和应用程序的内在并行性导致了并行机的出现和发展，反过来，并行机

的出现也很大程度上影响着并行算法和并行应用的发展。实际上，并行机和并行算法的发展是相互依存，缺一不可的。

3.4 GPU 体系结构以及作为通用计算的发展历程

目前市场上常见的显示芯片有三家厂商生产，AMD、Intel、NVIDIA。三家厂商的产品架构有很大不同，下面以 NVIDIA 的产品为例介绍一下 GPU 的体系结构和发展历程。

最早的 GPU 可以追溯到 80 年代，斯坦福大学的教授 Jim Clark 与学生创立了 SGI 公司，并与 1984 年开发出世界上第一个通用图形工作站 IRIS 1400。

1991 年，S3 Graphics 推出了第一款单芯片的 2D 图像加速器，S3 86C911。

1999 年，Nvidia 推出 Geforce 256 图形芯片，并给它起了个新名字，叫 GPU。Geforce 256 所采用的新技术有硬件 T&L（多边形转换与光源处理）、纹理压缩和凹凸映射贴图、立方环境材质贴图和顶点混合、双重纹理四像素 256-bit 渲染引擎等，其中硬件 T&L 技术可以说是 GPU 概念形成的标志。（之前的 T&L 是由 CPU 完成的）

硬件 T&L 的效果就是，3D 模型可以用更多的多边形来描绘，这样就拥有了更加细腻的效果。而对于光源处理来说，光照数据就不必通过 CPU 计算，通过显卡能获得更好的性能。图 10 为这个时期 GPU 图形处理的流程。

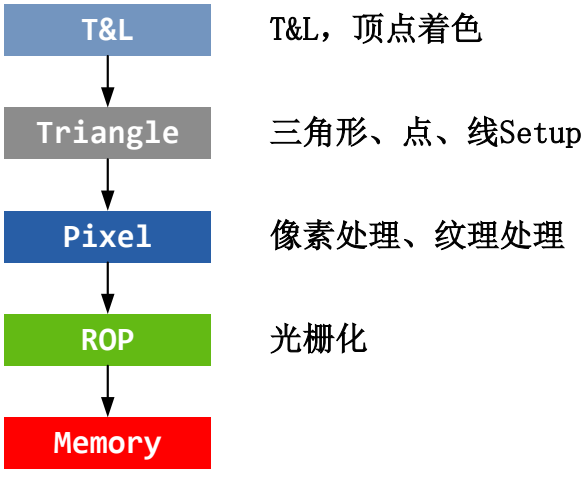

图10 早期的 GPU 处理图形的流程

GPU 的下一次重大改进是使用 Shader（着色器），Shader 翻译过来是着色器，原本是属于图形渲染管线的一部分，是专门负责执行重要的 3D 模型信息的转换、赋值、基本运算等功能的，Shader 是配合 DirectX 8 的规格产生的图形芯片架构技

术，它进一步扩展了 GPU 的能力，使原来的渲染管线和 T&L 有了编程能力，将原有的渲染管线分离出可编程的部分来给编程者更多的设计空间。Shader 从编程角度包括顶点着色器（Vertex Shader）和像素着色器（Pixel Shader）。DirectX 10.0 之后又提出了几何着色器（Geometry Shader），承担一部分顶点着色器的任务。

2001 年，NVIDIA 首先引入了可编程的顶点着色器，2002 年又引入了可编程的像素着色器，拿科学计算来比喻图形处理器的执行过程，顶点着色器相当于打网格，像素着色器相当于填数据，如果数据特别复杂，不是单一的点，那就是纹理。图 11 为这个时期的 GPU 处理图形的简化流程。

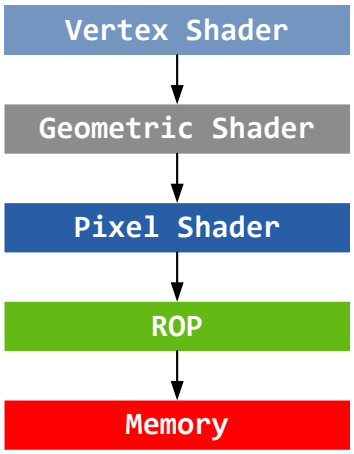

图11 基于着色器的 GPU 图形处理过程

图形处理所需要处理的步骤比较简单，没有很复杂的逻辑，而且是大量没有数据相关性的计算，由于光速、热力学定律和资金的限制，工程师们很容易的想出一个方法解决这种大数据量的计算，那就是并行，图形处理器中有大量的着色器，同时处理多路信息，这样在设计着色器时不用花太大工夫在提高主频和执行效率上，将几十上百个着色器绑在一起就行了。从 GeForce 6800 Ultra 开始，NVIDIA 公司 GPU 内部的顶点着色器流水线使用 MIMD 方式控制，像素着色器流水线使用 SIMD 结构。

2003 年，GPGPU 的概念第一次被提出，这是最早的对针对非图形应用程序的 GPU 的探索。通过使用图形编程语言如 DirectX、OpenGL，将并行算法导入 GPU。诸如蛋白质折叠、股票期权定价、SQL 查询及 MRI 重建等问题都能通过 GPU 获得加速。

事实上，顶点着色器和像素着色器处理的都是四元组数据，顶点着色器处理

用于表示坐标的 w、x、y、z（权重 weight，空间坐标 xyz），像素着色器处理用于表示颜色的 a、r、g、b（透明度、红绿蓝），顶点着色器需要比较高的计算精度，计算单元比较复杂；而像素着色器则可以使用较低的精度，计算单元简单，可以增加在单位面积上的计算单元数量。两种着色器都可以并行处理，架构也是相似的。

在传统的顶点着色器和像素着色器分离的架构中，两种着色器的比例是固定的。各种着色器在 GPU 核心设计完成时，各种着色器就已经确定下来，著名的“黄金比例”就是顶点着色器与像素着色器的数量比例为 1：3。

当然，不同的应用对顶点着色器和像素着色器的需求是不同的，如果场景中有少量的三角形，则像素着色器必须满负荷工作，而顶点着色器则会被闲置；如果场景中有大量三角形，又会发生相反的情况。因此，固定比例的着色器设计无法发挥 GPU 的计算潜能。

如果能将两种着色单元统一化，然后在使用时实时决定着色单元的性质，就不会产生如上的问题。

2006 年，从 GPU 的角度讲是里程碑式的一年，GeForce 8800GTX 显卡发布。相对于前一代的产品，GeForce 8800GTX 的架构 G80 不仅仅是架构的更新，而是摒弃了传统的 T&L+渲染线的结构，取而代之的是采用全新的流处理器（Stream Processor，SP）并行结构，官方上说的是统一渲染管线结构。这种结构中的每个着色单元是完全相同的，都是可编程的通用着色单元(Unified Shader)，在计算过程中根据需要来扮演不同的角色，执行不同的任务。而且，NVIDIA 为了提高 SP 的执行效率，为 SP 配备了 L1 和 L2 缓存。图 12 为 G80 架构下 GPU 处理过程，图 13 为简化的 G80 架构。

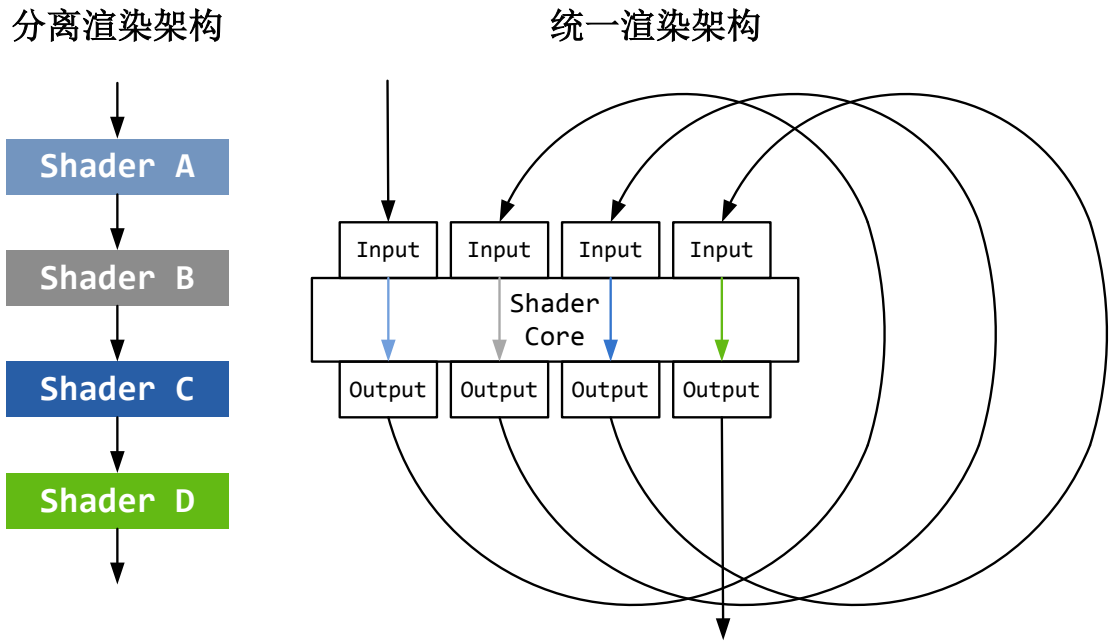

图12 G80 架构下 GPU 图形渲染过程

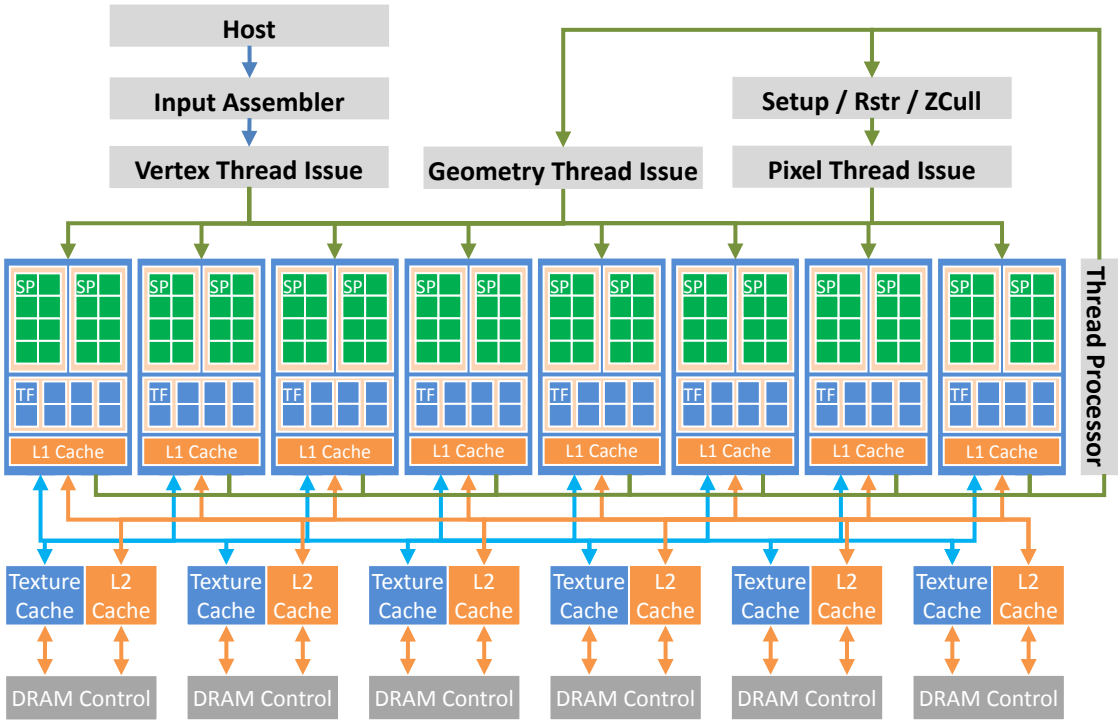

图13 简化的 G80 架构图

从架构上看，G80 已经具有通用计算的能力，G80 架构中包含 128 个 SP（流处理器），每 8 个 SP 为一组，组成一个 SM(Stream Multiprocessor)，两个 SM 组成一个 TPC(Texture Processor Cluster)，整个处理器由 8 个 TPC 组成。（图 14 ）

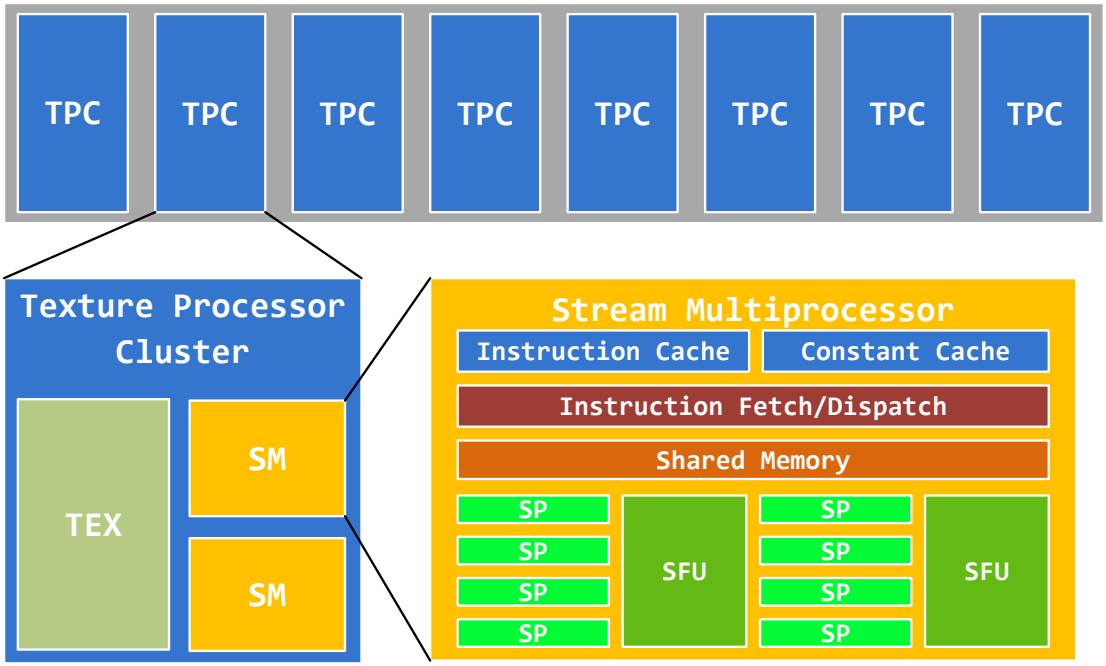

图14 G80 架构计算单元分解图

今天看来，以 GeForce8800 为代表的 G80 架构是一款非常有针对性的硬件架构，NVIDIA 推出这种架构不仅仅是为了提高图形处理的效率，也是为进军通用计算市场做铺垫。

2007 年 6 月，NVIDIA 在 G80 的基础上推出了统一计算设备结构（Computer Unified Device Architecture, CUDA）。CUDA 是一种将 GPU 作为数据并行计算设备的软硬件体系，它包含了 CUDA 指令集结构(ISA)以及 GPU 内部的并行计算引擎，开发人员可以使用 C 语言来为 CUDA 架构编写程序。CUDA 架构的最大的优势是降低了 GPU 通用计算的编程门槛，使众多的程序员不用学习复杂的图形处理的知识就可以编写 GPU 通用计算程序。

2008 年 6 月，NVIDIA 对 G80 架构进行改进，推出 GT200 架构，增加了 SP 数量，使用硬件内存合并访问以提高存储器存取的效率，支持双精度浮点数。

紧接着，NVIDIA 推出了升级的 Fermi 架构，本文中使用的 GPU 就是基于 Fermi 架构的 Tesla C2050。Fermi 架构，是以处理器为目标进行设计的。因为 Fermi 架构上已经应用了非常多的并行技术，在 Fermi 身上可以看到以前 GPU 上从来没有的东西，每个 SM 上拥有两条指令流水线，流水线静态调度，指令顺序执行，拥有可定制的 L1 cache 和统一的 L2 cache，支持大量的原子操作等等，同时在 Fermi 架构上应用了 GigaThread 技术，能够同时执行多个 kernel 函数。这些技术使 GPU

卡上能够同时处理多个任务，GPU 已经开始具备 MIMD 的特征。

在 Fermi 架构的发布会上，Nvidia 的 Tony Tamasi 先生（NVIDIA 高级副总裁，产品与技术总监）表示：“以前的 G80 架构是非常出色的图形处理器。但 Fermi 则是一款图形处理同样出色的并行处理器。”[28]

这句话揭示了 Fermi 的与众不同，它已经不再面向图形领域设计了，因为更为广阔的通用计算市场在等待它。Fermi 将为通用计算市场带来前所未有的变革，图形性能和游戏被提及已经越来越少。[28]

第四章 众核平台程序结构及主要优化方法

4.1 CUDA 的执行方式以及适用于 GPU 的算法

4.1.1 CUDA 的硬件模型以及 Tesla C2050 介绍

CUDA 的硬件模型相当于显卡硬件模型的一个封装，将显卡的着色调度单元封装成线程执行管理器（Thread Execution Manager），将 L1 cache 封装成可主动配置的 shared memory 和 L1 cache，保留 L2 cache、Texture Fetch 等器件。其指令调度和执行方式与显卡相同，唯一不同的是数据流已经不再是三维图形数据，而变成了任意的整型或浮点数据。

以 G80 为例，说明一下 CUDA 架构中计算单元的组成（图 15 ），前面提到，G80 由 8 个 TPC 组成，每个 TPC 由独立的 2 个 SM 组成，每个 SM 由 8 个 SP、2 个 SFU、一部分指令缓存和常量缓存，一个线程调度器（流水线）和一块共享内存组成，而 SP 则是 GPU 最基本的计算单元，因为一般认为，SP 是整数和浮点计算的执行单元。所以说 GPU 是由上百个计算单元组成的，这个计算单元指的就是 SP。

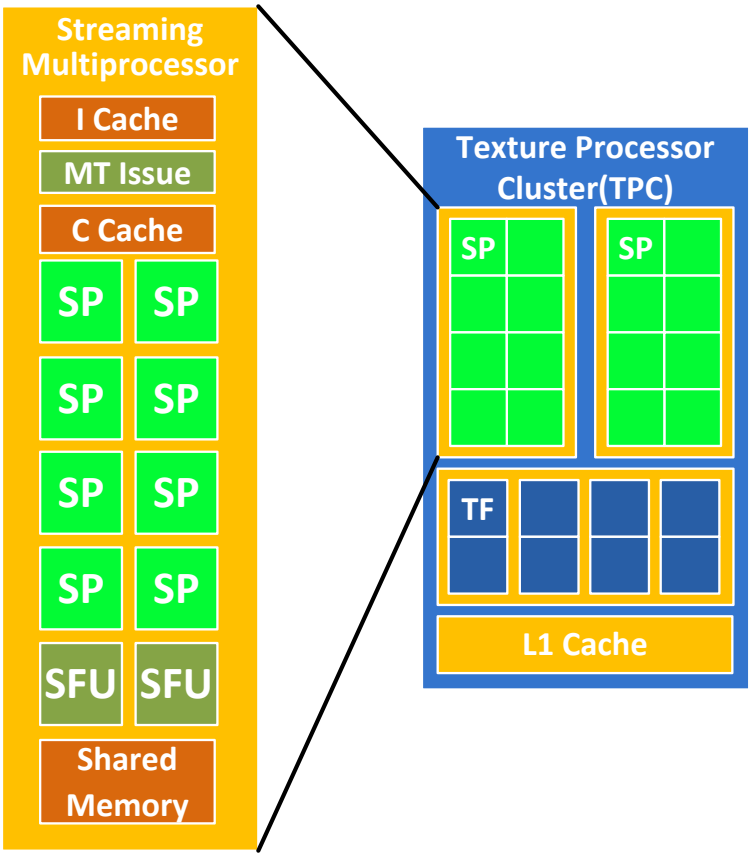

图15 G80 架构中 SM 和 TPC 的组成结构

虽然在 NVIDIA 的宣传里，CUDA Core 或者 SP 就是一个核，但是从目前的硬件来看，SP 其实只是一个功能单元而已，真正能比较接近于我们常说的核心，则是 SM，因为目前只有在 SM 这一级才具有 PC（Program Counter，程序计数器）、调度资源以及寄存器堆。

GPU 处理的图形数据和科学计算数据没有本质区别，通过对比图 16 和图 17 可以看出作为 CUDA 的 G80 和作为显卡工作的 G80 最大的不同在于封装了缓存和线程管理器。G80 原本有 L1 和 L2 缓存，但是作为 CUDA 架构使用时将其屏蔽掉了，将 6 个内存控制器封装成全局内存和局部内存，将顶点着色调度器、像素着色调度器、几何着色调度器封装成统一的线程执行管理器。相对于传统的 GPU，CUDA 在通用化、统一化方面又迈出了重要的一步。

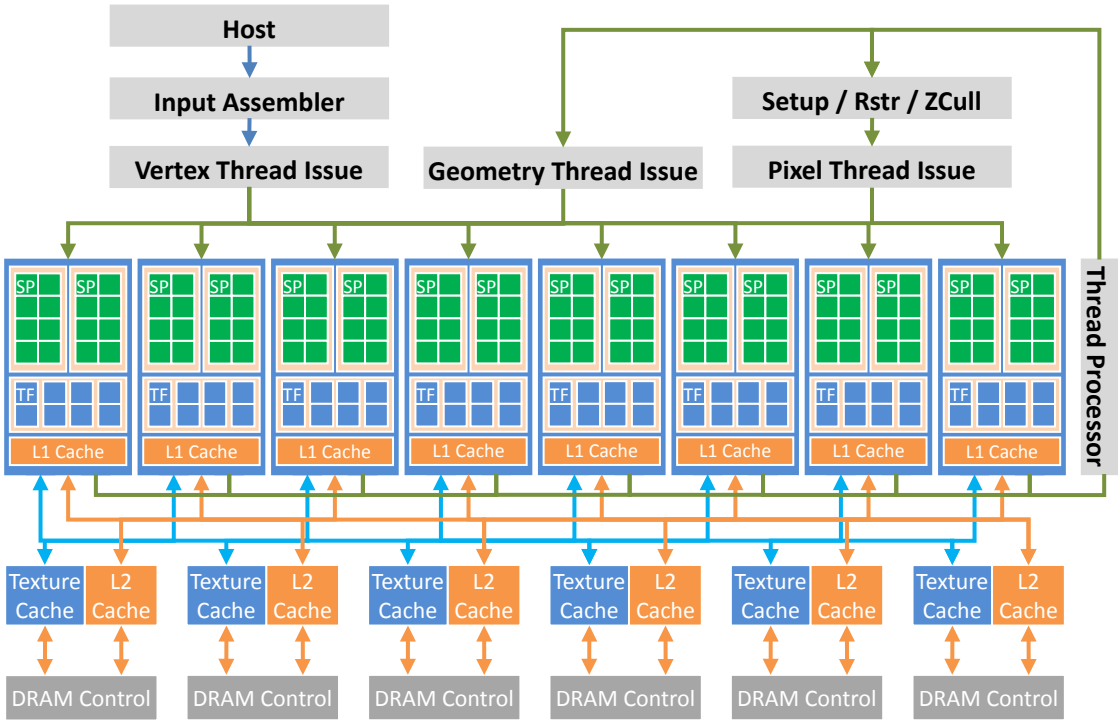

图16 作为显卡的 G80 架构图

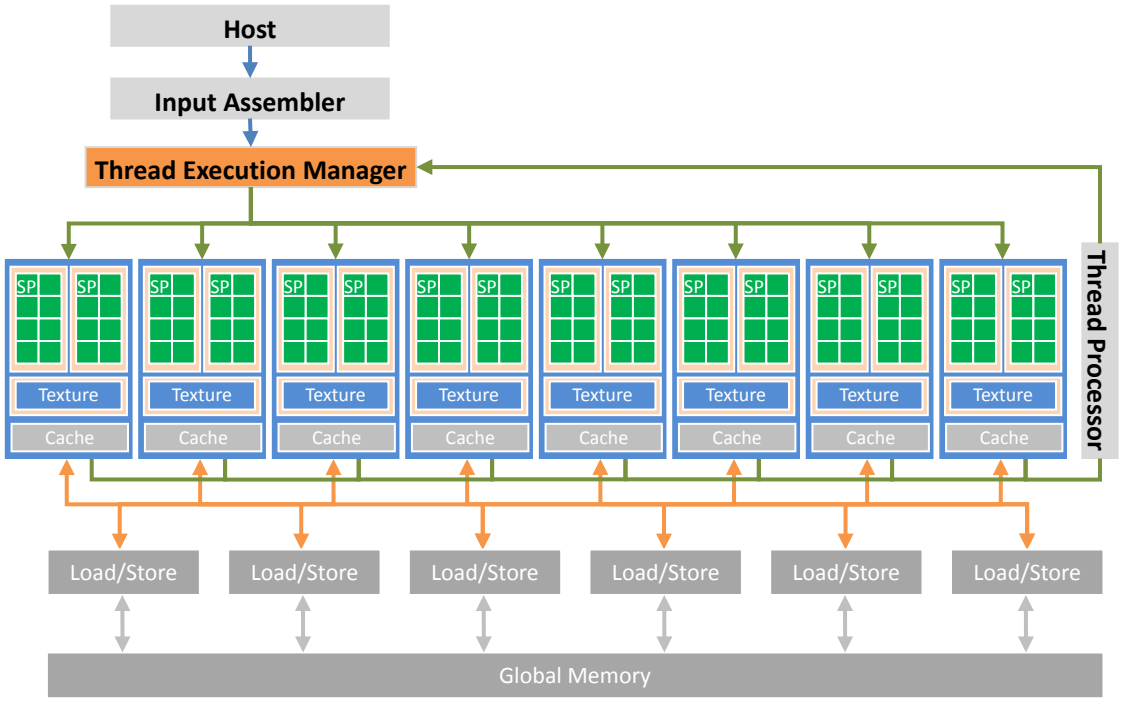

图17 封装成 CUDA 架构的 G80 架构图

G80 架构经过几代的更新换代已经发展到了 GF100 架构，代号为 Fermi，代表产品是 Tesla C2050/2070，Fermi 相对于 G80 不仅仅是数量上的变化，也增加了很多适用于高性能计算的新功能。

下面介绍一下作为并行处理机的 Tesla C2050 具有的硬件特性：（图 18 ）

- 包含 448 个 SP，14 个 SM，每个 SM 有 32 个 Core（CUDA 核心，其实就是 SP，NVIDIA 为了推新产品，给 SP 换了个名字）。
- SM 以 32 个并行线程（称为 warp）为单位组成的线程组为单位进行调度。每个 SM 含有两条指令流水线，每条指令流水线每两个时钟周期发射一条指令，指令静态调度，顺序执行，每个 SM 上的 32 个 Core 分为两组，每组 16 个 Core，每个 Core 在两个时钟周期内执行一个 warp。
- 每个 SM 上有 64KB 的可定制高速缓存，可以定义为 shared memory+L1 cache。
- 拥有 768KB 的统一 L2 高速缓存，用于所有的读取、存储、纹理操作，原子操作性能也大幅提升。
- 每个 SM 上有四个特殊功能单元（SFU），能 够快速执行 sin()，sqrt()等复杂的数学函数。
- 支持 40bit 的统一寻址空间，能够将 local memory、shared memory、global

memory 统一为一个单独、连续的空间，对 C/C++的指针提供完全的支持。

- 显存支持 ECC 校验。
- 支持并发 kernel 执行。

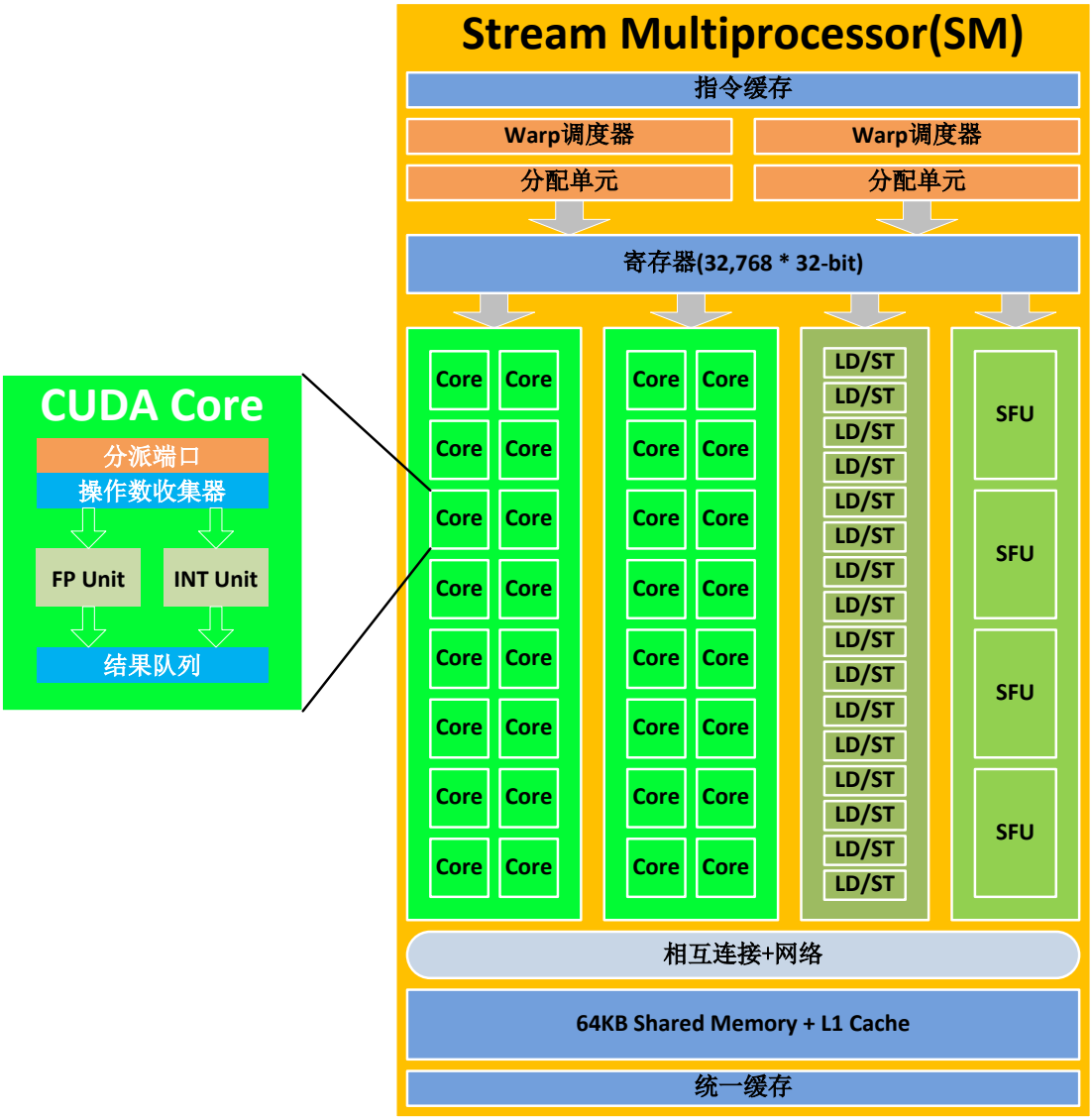

图18 Fermi 架构的 SM 结构图

这里我们总结一下 Fermi 是一个怎样的并行机。首先看 Fermi 的并行计算的硬件特征：

- 多核，一个芯片上有 16 个 SM，（Tesla C2050 上有 14 个 SM）每个 SM 上可以运行不同的指令，但是 SM 内部必须运行相同的指令。
- 超标量流水线，每个 SM 上有 2 个 warp 调度单元。
- 多执行单元，每个 SM 上有 2 组 SP，每组 16 个。

- 向量计算，每组 SP 在同一时刻执行完全相同的指令，只有数据不同，2 个时钟周期完成 32 组数据的计算。

可见 Fermi 已经是一个 MIMD 类型的并行机了，虽然计算核心相对于 CPU 核心仍然非常简陋，但是这种片内多向量处理的思想可能会成为未来高性能计算的主流思想。目前，Intel 和 AMD 也有类似架构的产品出现，说明硬件厂商对这种架构的发展潜力是有很大共识的。

4.1.2 CUDA 线程模型

CUDA 的基本思想是支持大量的轻量级线程并行，是一种高度线程并行化的解决方案。

CUDA 程序是一种 C++的扩展，通常以内核函数（kernel）的形式来运行。内核函数以网格（Grid）的方式由 CPU 中的可执行文件调用执行，每个网格由若干个块（Block）组成，每一个 Block 又由上百个线程（Thread）组成，同一个 Block 中的线程不仅能并行执行，而且能够通过 SM 上的共享内存(Shared Memory)和同步栅(Barrier)来实现块内线程的通信和同步。各 Block 是并发执行的，Block 间无法通信，也没有执行顺序。在 G80 架构中，硬件上仅能同时存在一个 Grid，目前一个 kernel 函数中只有一个 Grid，而未来支持 DirectX11 的硬件将采用 MIMD（多指令多数据）结构，允许在一个 kernel 中存在多个不同的 Grid。（图 19 ）[28]

由于一个 Grid 中的所有线程执行的是相同的程序，所以 NVIDIA 给 CUDA 的线程模型起了一个形象的名字叫做 SIMT（Single Instruction Multiple Thread，单指令多线程）。

4.1.3 CUDA 存储模型

GPU 中存储空间可配置性较强，由于 GPU 没有 CPU 中复杂的缓存策略、寄存器重命名等功能，所以 CUDA 中很多存储部件是可以手动配置的。CUDA 架构中，线程在执行中可以访问多个存储器空间结构，如图 20 。根据存储器在物理位置和存储速度不同，存储结构可分为片外内存和片上内存，根据存储器作用范围不同，存储结构可分为私有内存、共享内存、全局内存。这里介绍 6 种主要的存储器。

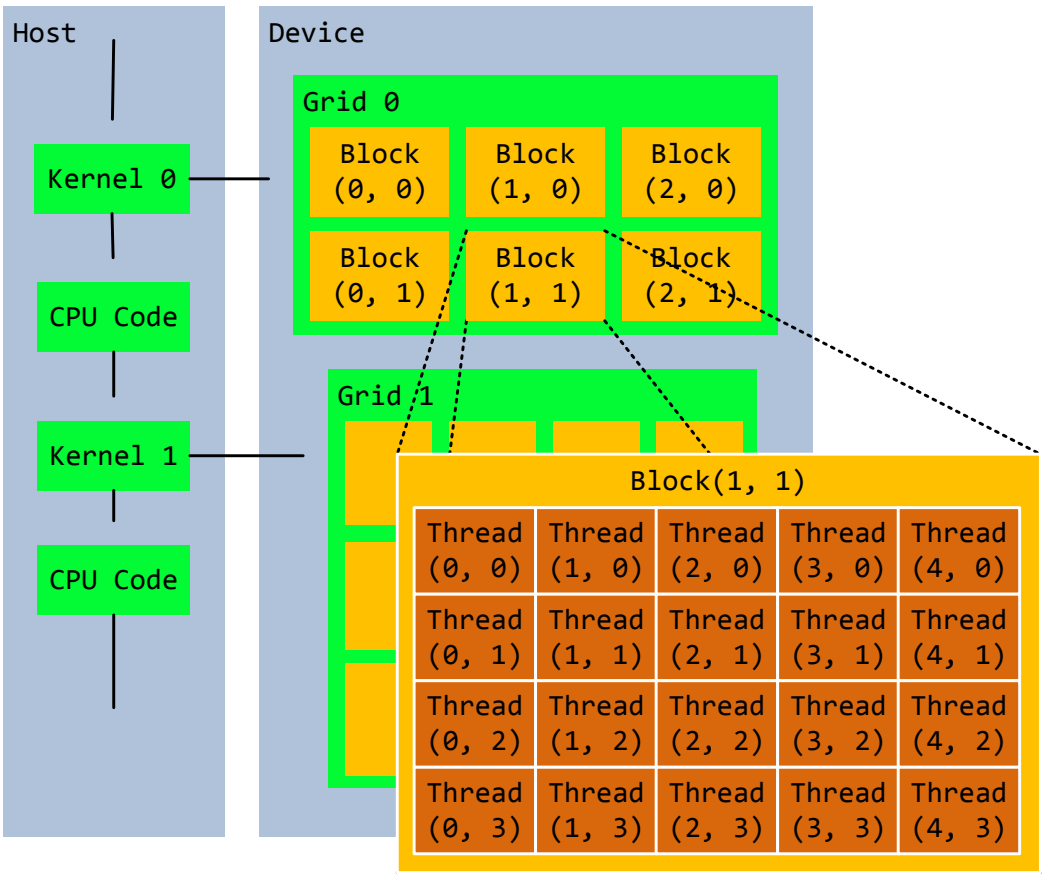

图19 CUDA 的线程模型

寄存器(Register)：和 CPU 一样，对于每个线程来说，寄存器都是线程私有的。寄存器的访问速度非常快，单次访问同一个寄存器只需要一个时钟周期(cycle)的延迟。每个寄存器宽度为 32-bit，所以如果处理双精度浮点数需要两个寄存器（G80 不支持双精度浮点数的计算）。每个线程块中的线程占用的寄存器是静态分配的（GPU 不支持乱序执行，当然也就没有寄存器重命名的功能），寄存器的生命周期类似于栈的生命周期，只有在函数返回时才会释放资源（NVCC 目前没有动态管理内存的功能，也没有堆的概念），而且寄存器的数量十分有限，在 G80 架构中，一个 SM 上只有 8192 个寄存器，而一个线程所使用的最大寄存器数量也被限制在 124 个。如果寄存器资源被消耗完，数据将被存储在本地内存(Local memory)中。

本地内存(Local Memory)：本地内存这个名字比较有迷惑性，本地内存并不代表访问速度快，事实上，本地内存属于片外内存，在 G80 架构上，访问需要 500 个左右的时钟周期。本地内存之所以叫本地内存是因为它是每个线程私有的内存，当寄存器资源不足时，线程的私有数据将放在本地内存中。G80 架构上，每个线

程最多支持 16KB 的本地内存。

共享内存(Shared Memory)：共享存储器可以被一个块中的所有线程访问，并且可读可写，如果不存在块冲突，它的访问速度几乎和寄存器一样快，是实现块内通信的最方便也是最快捷的方法。共享内存建立在片内的 Cache 上，但是不同于 CPU 的 cache 由硬件自动调度，共享内存可以由程序员显式的操作。G80 中，每个 SM 上有 16KB 的共享内存。

全局内存(Global Memory)：使用的是普通的内存颗粒，也是显卡工作时的显存，G80 架构中使用的是 DDR3 内存，内存颗粒在片外，所以访存延迟很严重，大约 500 个时钟周期。但是 G80 架构的全局内存是由 6 个 64-bit 的内存控制器控制的，位宽达到 384-bit，所以全局内存的带宽还是相当可观的。G80 架构中并没有对全局内存做缓存，如果需要完全利用好这么高的带宽，需要做到线程合并访问，在 G80 架构中，当连续的 16 个线程访问连续的全局内存是，如果内存地址满足对其要求，可以合并为一个访问事务(Event)，一次访问最多可以访问 384bits 的信息，如果不满足合并访问的要求，就会多次访问，效率会下降。合并访问的规则与 GPU 的架构和计算能力相关，有比较复杂的规则。全局内存一般比较大，有 1.5G~6GB。

常量内存(Constant Memory)：常量内存空间较小，只有 64KB，访问延迟也比较高，好处是在每个 SM 上有 8KB 常量缓存，当同一个 half-warp 中的线程同时访问常量内存中的同一个数据时，如果发生缓存命中，只需一个时钟周期就可获得，而且由于常量内存是只读的，编程人员不需要关心数据一致性问题。

纹理内存(Texture Memory)：纹理计算是由纹理拾取单元封装而成，纹理内存在 GPU 的硬件架构中并不是一块专门的存储器，而是涉及到显存、纹理缓存、纹理拾取的纹理流水线。在 CUDA 编程中，编程人员可以将显存中的某块区域与纹理内存进行绑定然后通过纹理拾取的方法去访问，通过纹理拾取方法访问的数据可以在纹理缓存中得到缓存，但是纹理拾取是只读的操作。

表 1 中列举了 GPU 中可访问的几种存储结构和特点，在使用 GPU 加速计算的过程中，要灵活运用多种存储结构，才能发挥 GPU 的计算潜能。

表1 GPU 中不同存储器的特性

存储器	Memory	访问速度 (cycle)	读写权限	作用范围	物理位置	能否缓存
寄存器	Register	1	读、写	线程私有	片上	
共享内存	Shared Memory	1	读、写	块内共享	片上	
本地内存	Local Memory	~500	读、写	线程私有	片外	否
全局内存	Global Memory	~500	读、写	全局共享	片外	否
常量内存	Constant Memory	~500	只读	全局共享	片外	能
纹理内存	Texture Memory	~500	只读	全局共享	片外	能

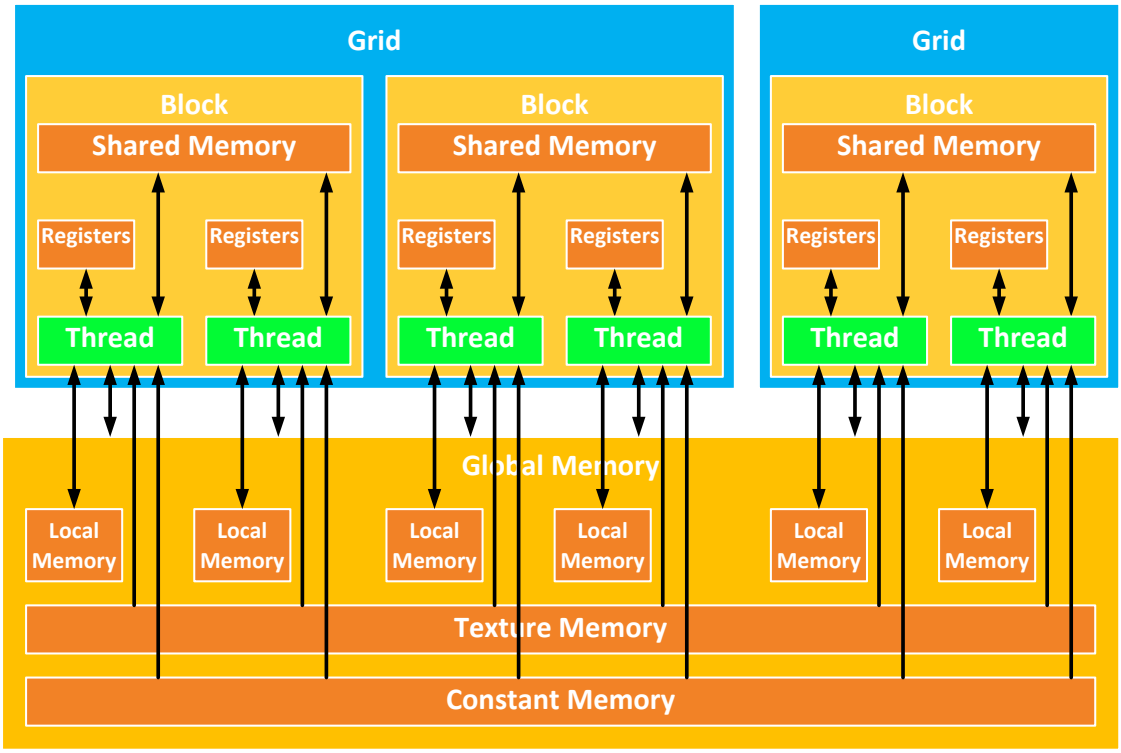

图20 G80 架构下的 CUDA 存储模型

Fermi 架构中的存储模型与 G80 有一些不同，比较大的进步是 Fermi 中支持对全局内存和本地内存的 L1 和 L2 缓存，而且每个 SM 上有 64KB 的可配置 L1 缓存，可以配置成 16KB Cache+48KB Shared Memory 或者 48KB Cache+16KB Shared Memory 的模式，对于不同类型的程序，这种配置可以获得性能提升。另外 Fermi 也可以配置是否所有数据都通过 L1 进行访问，这些可配置的架构带来了更高的编程灵活性。图 21 是 Fermi 架构的简化存储模型。除了架构上的不同，Fermi 的各

项存储指标相对于之前的产品也有了变化，每个线程支持的本地内存提升到512KB，每个 SM 上的寄存器数提升到 32KB 等，不过令人匪夷所思的是，Fermi 上每个线程所支持的最大寄存器数减少到了 63 个，比 G80 的 124 还要少！对于动辄上百个变量的计算流体力学程序而言，无疑是个巨大的挑战。

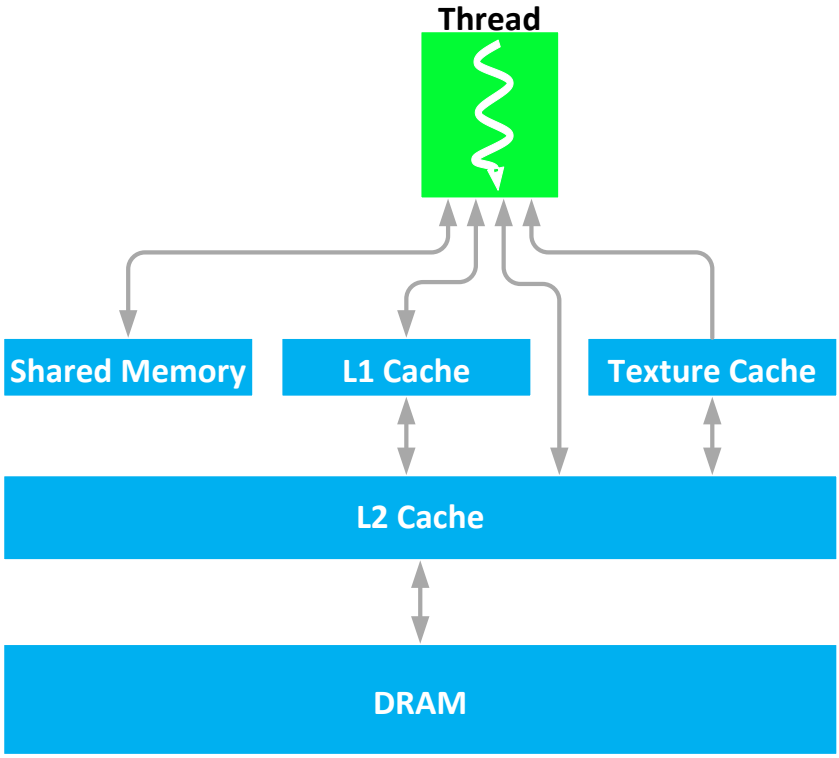

图21 Fermi 架构的简化存储模型

4.1.4 CUDA 执行模型

CUDA 中的 kernel 函数以在编程上被分为 block，同一个 Block 中的线程可以通过 shared memory 共享数据，shared memory 是每个 SM 私有的，所以它们必须在同一个 SM 中发射。所以这里需要注意：一个 Block 必须被分配到同一个 SM 中，但是一个 SM 中同一时刻可以有多个 Block 在执行或者等待执行（目前版本最多 8 个），当一个 Block 的某个 warp 进行访问片外内存或者同步等高延迟操作时，另一个 Block 或者另一个 warp 就可以占用 SP，（硬件线程切换速度非常快，最快只需要一个时钟周期。）最大限度利用 SM 的运算能力，目前的 GPU 不支持指令的乱序执行，多个 Block 占据一个 SM 是隐藏延迟的主要手段。[28]

在实际运行中，Block 会被分割为更小的组合，就是 warp。G80 每个 SM 上有一条流水线，每 4 个时钟周期发射一条指令，每条指令有 8 个 SP 执行，所以每 32

个线程会执行完全相同的指令，这 32 个线程就被称为一个 warp，warp 是 GPU 最基本的调度单元。warp 的大小不是唯一的，是由硬件的计算能力决定的，在抽象的 CUDA 编程模型中并不用考虑 warp 的概念。

Fermi 架构的一项重要技术为双级分布式线程调度器。在芯片一级，全局任务分配引擎为不同 SM 提供线程块。而在 SM 一级，每一个 warp 调度器为其执行单元分配 32 个线程。Fermi 的每一个 SM 都有 2 条流水线，每条流水线每 2 个时钟周期发射一条指令，2 条流水线的指令由 2 个 warp 调度器分别调度（图 22 ），[28] Fermi 的双 warp 调度机制可以同时并发调度两个 warp 的一条指令分别在 16 个一组的 CUDA Core 上执行，因此 2 个时钟周期刚好能够执行以 32 个线程构成的一个 warp 的指令。G80 架构中使用了 GigaThread 引擎，可实时管理 12288 个线程。Fermi 在这一基础上得到了大幅改进，不仅显著提高了线程吞吐率，同时加快了并发 kernel 执行等操作。

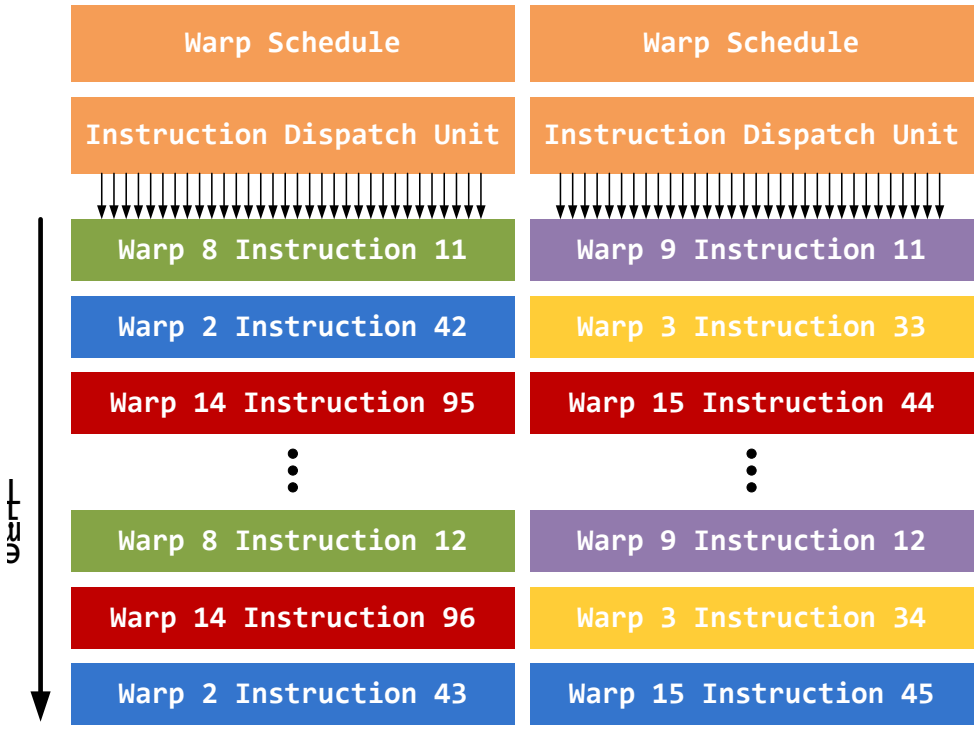

图22 Fermi 的双 warp 调度策略

并发 kernel 执行，即同一应用上下文中的不同内核能够同时在 GPU 上执行。（图 23 ）并发 kernel 执行允许程序执行大量小型 kernel，以最大限度利用整个 GPU 的资源。同时，借助改进的上下文交换性能，来自不同应用上下文的 kernel 仍能高效顺序执行。有了并发 kernel 执行功能之后，NVIDIA 就可以名正言顺的说

CUDA 是 MIMD 了，因为不同的 block 之间可以执行完全不同的 kernel 函数了。这样，CUDA 中，每个 block 可以看做 SIMT，block 之间可以看做 MIMD，一个 warp 内部的行为更像是 SIMD，block 内允许分支，warp 内如果有分支，性能会降低。

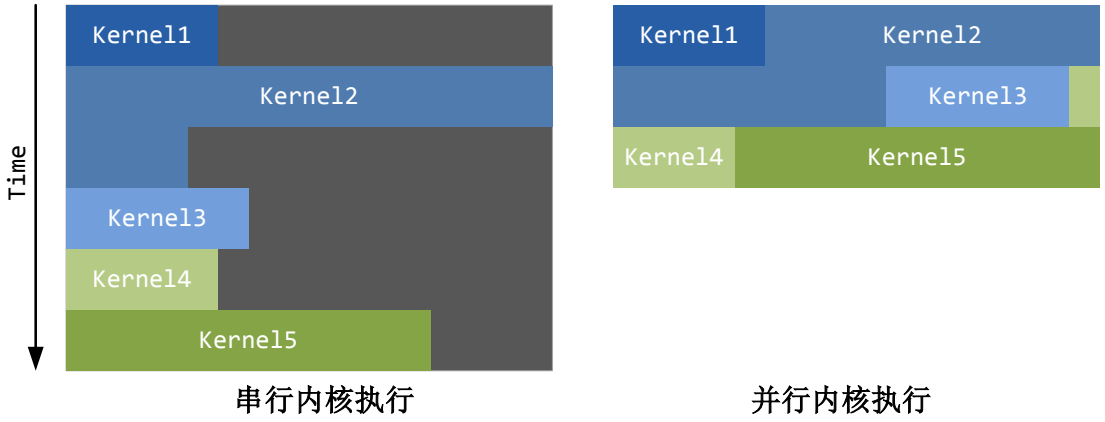

图23 Fermi 的并发 kernel 执行

4.2 GPU 程序结构

求解器的 CPU 程序由初始化、迭代求解、结果输出三部分组成，其中计算量比较大的是迭代求解部分，主要包括边界条件的设置、守恒量的计算、流通量的计算、全局量（时间步长、全局残差）的计算。所以在初始化完成后将数据从内存中全部传入 GPU 中，然后在 GPU 中完成全部的迭代求解。（图 24 图 25 ）中间只涉及 1~2 次数据传输，一次是要求出迭代的时间步，用来下一步的迭代，另一次是求出全局的残差，用来描绘残差曲线。理论上这两次数据传输都无法避免，而且都涉及全局归约的操作，不过本文的工作中采用了一些加速方法来提高全局归约的效率并减少数据传输。

其中的难点在于，守恒量是存储在元胞上的数据，而流通量存储在三角形的边上。流通量计算完成后，需要将其对应到网格单元上。而元胞和边并不是一一对应的数据结构，存储顺序也不一样，访存时也无法合并访问。在 CPU 程序中，程序串行执行，数据有效性不会受到影响，而且 CPU 有成熟的缓存结构和大容量的二级缓存，性能也不会下降，所以 CPU 程序中流通量在计算过程中虽然按照边的顺序进行计算，但是结果写入元胞的数据结构内。

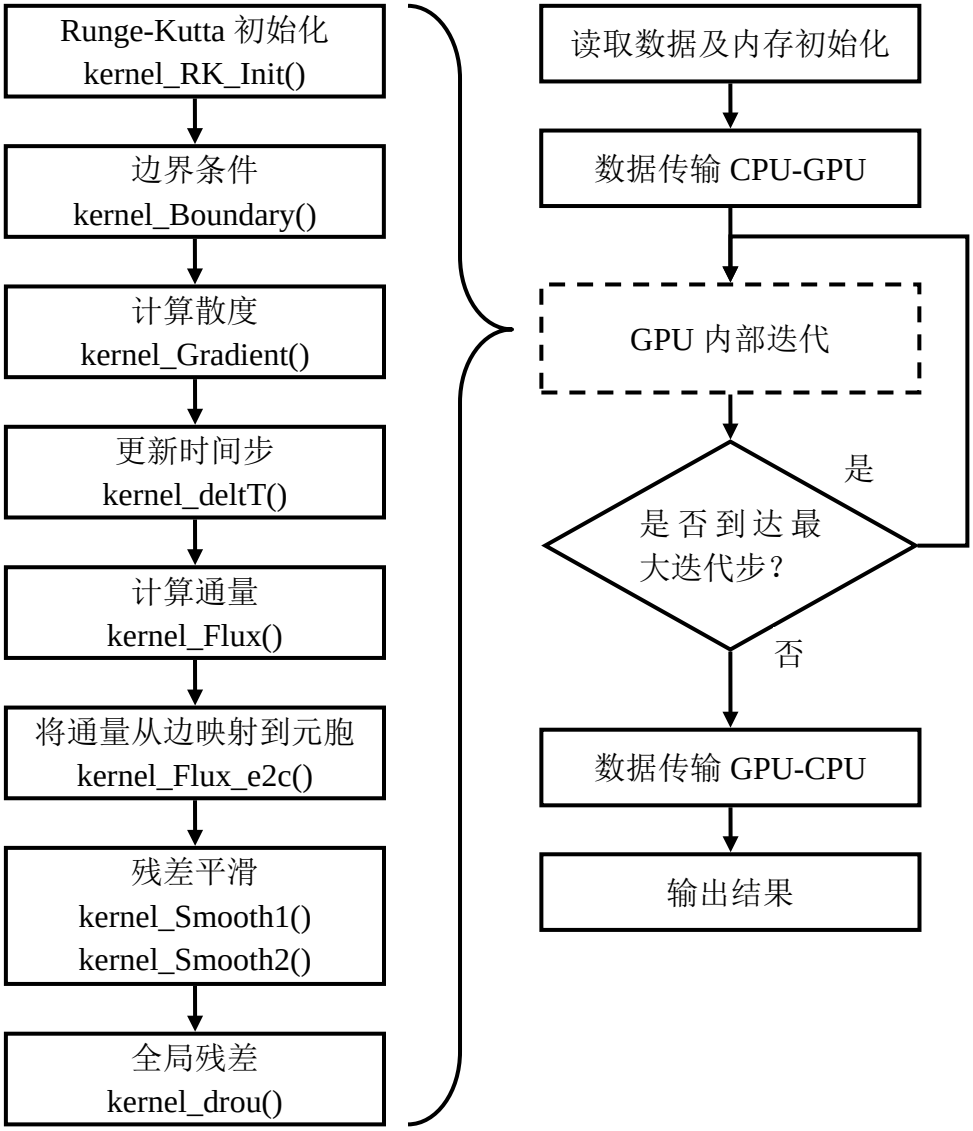

图24 GPU 程序流程图

而在 GPU 这样的并行计算架构中，如果仍然采用上面的方法计算流通量，随机访存会影响整体的运算速度，数据有效性问题会影响结果的正确性，虽然 CUDA 支持原子操作，但是效率很低。所以在 GPU 程序中，将流通量模块 Flux()一分为二，以边为顺序计算流通量，并且以边为顺序存储，然后再启动一个 kernel 函数，以元胞为顺序读取流通量，并存储到元胞的数据结构中。

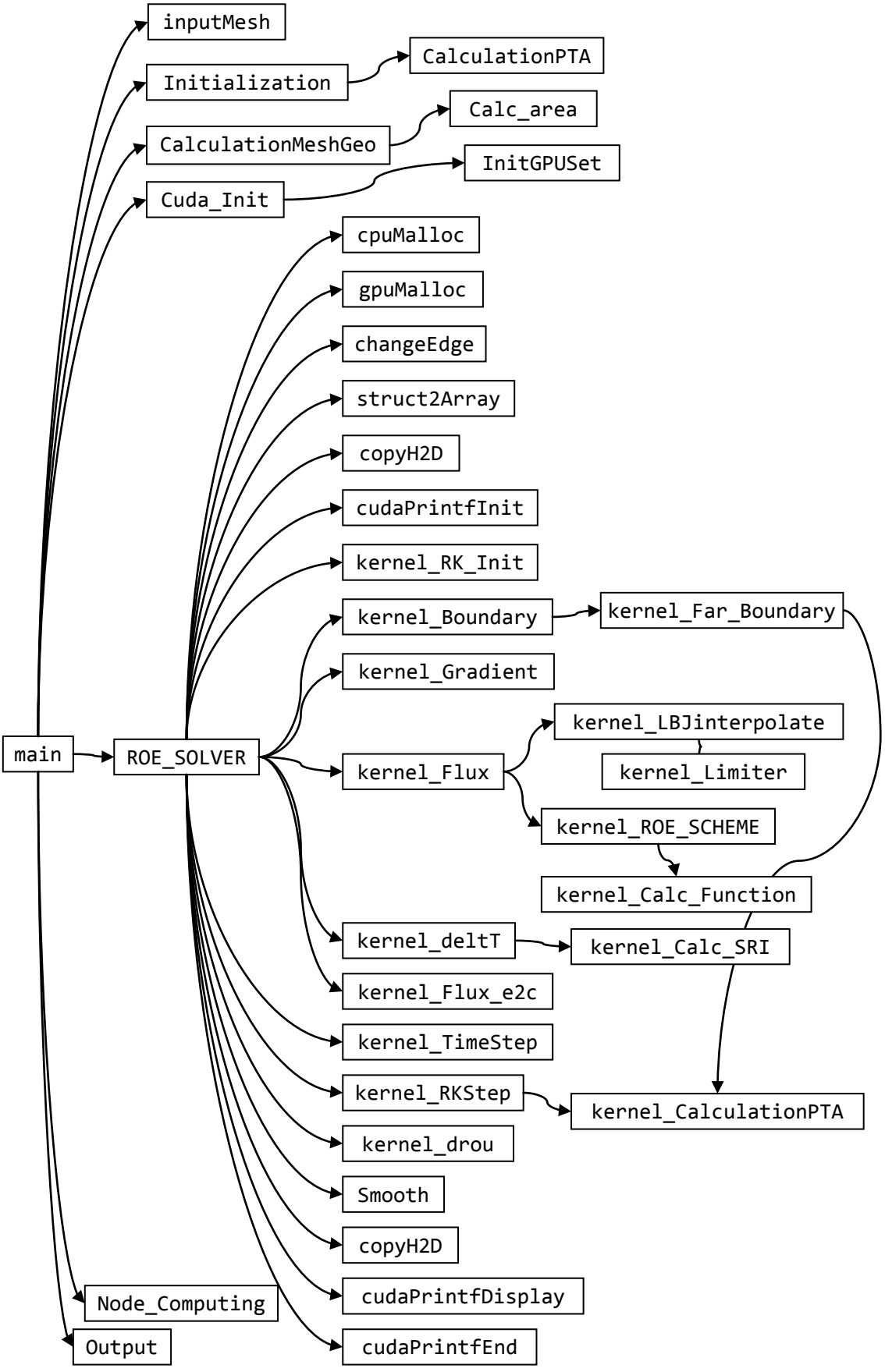

图25 GPU 程序的函数调用图

CPU 是串行处理，并且有大容量的缓存和更为智能的预取策略，可以很好的处理随机存储的数据结构。GPU 虽然有较高的访存带宽，但是高带宽是通过高访存位宽实现的，只有连续存储的大块数据才能有效的发挥合并访存的优势，对于随机存储的数据，不能保证数据存储的连续性，虽然有 384bit~512bit 的访存位宽，但每次读写的数据有效性不好，大大影响访存速度。

4.3 优化方法

科学计算程序在模块耗时上往往有“二八原则”，即 20%的代码占用了 80%甚至更多的计算时间。在进行 GPU 优化前，使用性能分析工具对程序性能进行分析是有必要的。常见的性能分析工具有 Microsoft Visual Studio 中集成的性能分析器，Intel VTune，以及开源的 gprof 等。由于本文开发及测试环境都是 Linux 平台，所以使用的性能分析工具是 GNU 的 gprof。

在 Linux 下使用 g++或者 nvcc 编译和链接时，加上-pg 选项，执行应用程序时即可生成供 gprof 分析的数据文件。gprof 是一个 mcount 函数，会在内存中保存一张函数调用图，并通过函数调用栈查找子函数和父函数的地址，这张调用图也保存了所有与函数相关的调用时间、调用次数等信息。

CPU 程序能够很方便的通过 gprof 分析热点等性能信息，GPU 的 kernel 程序是异步执行，kernel 函数调用后会立刻返回，所以无法使用 CPU 平台上传统的性能分析工具进行分析，NVIDIA 提供了用于 CUDA 的性能分析工具 computeprof，但是由于本文所采用的测试环境并没有安装此产品，所以在代码中加上时间戳来获取各部分代码执行时间。

表 2 为 CPU 程序与优化前的 GPU 程序性能对比，其中运算时间为在 51800 个网格单元的网格上推进一个时间步的平均时间（平均时间由 1000 个时间步平均产生，单位为 ms，下同）。所有浮点运算都使用双精度浮点数。

表2 CPU 程序与优化前的 GPU 程序性能对比

CPU 函数	CPU(ms)	GPU 函数	GPU(ms)
RK_Init	4.4107	kernel_RK_Init	6.1331
Boundary	6.6673	kernel_Boundary	0.62661
Gradient	197.45	kernel_Gradient	21.418

Flux	420.67	kernel_Flux	13.644
		kernel_Flux_e2c	25.197
Smooth	143.70	kernel_Smooth1 kernel_Smooth2	8.248
RKestep	21.794	kernel_RKstep	1.9349
Total	797.30	Total	77.361

由表 1 可以看出， Gradient()、Flux()、Smooth()三个部分是耗时比较大的热点，需要针对热点进行优化，RK_Init()部分在 CPU 程序中虽然不是热点，但是移植到 GPU 上之后，包含一些数据传输函数，增加了耗时。

如果不对 GPU 程序进行进一步优化，双精度下的加速比只有 10 倍左右，并没有发挥出 GPU 全部的潜力，需要根据程序以及 GPU 的硬件架构进行进一步的优化。

4. 3. 1 结构体的数组到数组的结构体的转换

NVIDIA 建议在 CUDA 程序中使用数组的结构体(structure of array)代替结构体的数组(array of structure)，结构体的数组和数组的结构体的区别还是很明显的。在串行程序中，结构体的数组应用比较广泛，一个是内存的位宽一般为 128bit 或者 256bit，结构体有利于 CPU 的合并访问，而且结构体在分配内存等操作上比较方便。但是 GPU 中大部分结构体无法满足合并访问的规则，而且一个结构体内部成员变量较多，在一个函数中，并不是每个成员变量都需要载入寄存器。所以使用数组的结构体能够增加合并访问的效率，减少访问的数据大小。但是这个转换也是有代价的，一个是要为结构体中的每个指针分配空间，第二个是 C/C++语言对结构体是深拷贝，能够很容易的复制一个结构体，但是转换为数组的结构体后，就要显式的复制每一个成员变量，这会增加很多代码量。而且实话实说，代码的可读性会变差。

经过转换后，元胞、边和点的结构体变为如下形式：

```
typedef struct CELL_S
{
        int         *CLog;
        int         *Point[3];
        double      *deltU[2][4];
```

```
        double     *Umax[4],*Umin[4];
        double     *center[2];
        int        *celledge[3];
        int        *neicell[3];
}CELL_S;

typedef struct W_S
{
        double     *density;
        double     *density_U;
        double     *density_V;
        double     *density_E;
        double     *P;
        double     *T;
        double     *A;
}W_S;
```

```
typedef struct EDGE_S
{
        int        *ELog;
        int        *left_cell;
        int        *right_cell;
        int        *farfieldid;
        int        *wallid;
        int        *node1;
        int        *node2;
        double     *vectorn;
        double     *vectorx;
        double     *vectory;
        double     *midx,*midy;
}EDGE_S;
```

```
typedef struct NODE_S
{
        int        *NLog,*Nlog1;
        double     *x;
        double     *y;
        double     *ROU,*U,*V,*E;
        double     *P,*T,*A;
        double     *Mach;
        double     *RoundArea;
}NODE_S;
```

单纯地将结构体的数组转化为数组的结构体可能并没有很大的性能提升，但是这种转化对程序优化的意义是很大的。这样在能够在此基础上进行一些其他的优化，比如更改数据的存储顺序、使用 shared memory 等等。

经过转化， Gradient()的计算时间由 21.418 ms 减少到 18.452 ms，Flux()的计算时间由 13.644 ms 减少到 11.913 ms，Flux_e2c()的计算时间由 25.197 ms 减少到 18.688 ms， Smooth()的计算时间由 8.248 ms 减少到 7.1775 ms。

4. 3. 2 更改数据的存储顺序

分支和循环是最基本的流程控制，也是编程中最基本的概念。我们知道 CPU 是为单线程程序运行加速而设计的，GPU 是为并行程序设计的。首先 CPU 是指令并行，只有指令间并行的架构才需要分支预测，而 GPU 是线程间并行，指令按顺序发射，没有分支预测。GPU 的分支能力是靠线程挂起来实现的，换而言之是频繁的线程切换把分支延迟都给掩盖了。如果一个 warp 内的 32 个线程分成了 3 个分支，那么原来执行一个周期的指令会分为 3 个周期，3 个周期分别执行 3 个分支。所以当 GPU 遇到分支问题时，性能会发生较大的变化，而且不同架构的性能变化程度也不一样。但总的来说，分支对 GPU 的执行提出了巨大的挑战。

非结构网格一个主要特征是数据存储结构不整齐，相邻的两块数据很可能具有不同的属性，而编译器和执行单元并不知道这种特征。对于本文所使用的非结构网格的数据结构，大部分数据处于流场范围内，但是在远场和壁面有边界条件。这些具有远场和壁面边界条件的元胞和边与流场的元胞和边夹杂在一起存储。

这样一来，在计算一个单元之前，需要先判断它的属性，是否是边界条件，如果是边界条件，就需要调用其他的函数进行处理，这样在程序中就会产生分支。前面提到，GPU 并没有乱序执行和分支预测，在处理分支时的效率是比较低的，比较好的解决方法就是尽量在一个 warp 中执行相同的分支。

另外，GPU 有上百个 SP，（Tesla C2050 有 448 个 SP），但是只有 6 个内存控制器，全局内存的访存位宽只有 384-bit，NVIDIA 公司虽然大肆宣扬自己的显存带宽大，但是一直在刻意回避 400 多个 SP 共享这些带宽的事实。事实上，如果将带宽除以 SP 的个数，SP 的带宽是想当可怜的。在 CPU 编程中，内存带宽一般不是首要考虑条件，但是在 GPU 中，内存带宽是一个程序调优的首要考虑。

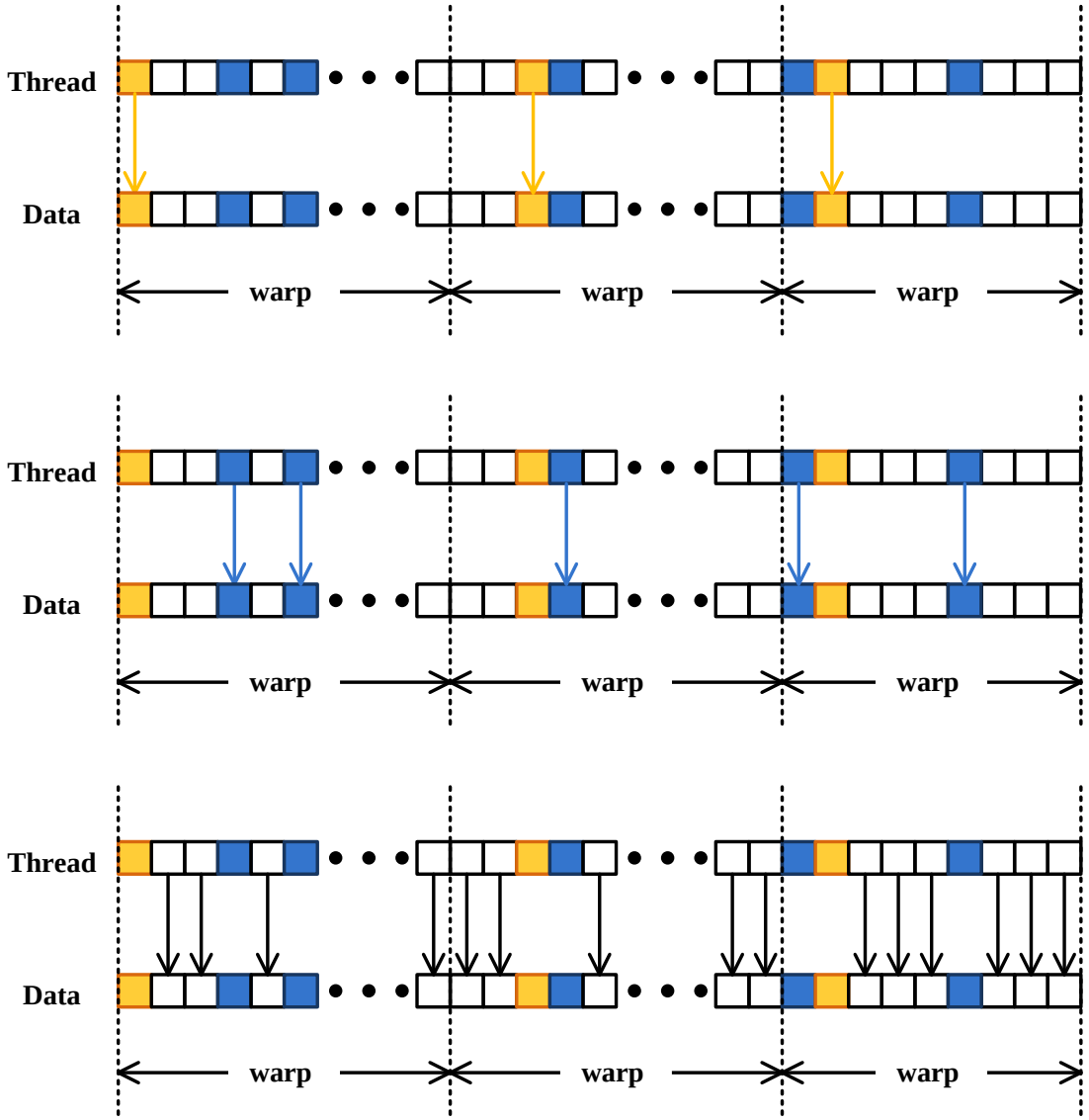

图26 不同属性的线程无法完成合并访问

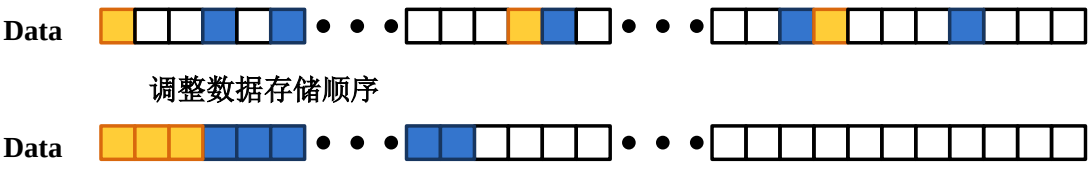

图27 将数据根据边界条件进行重新排列

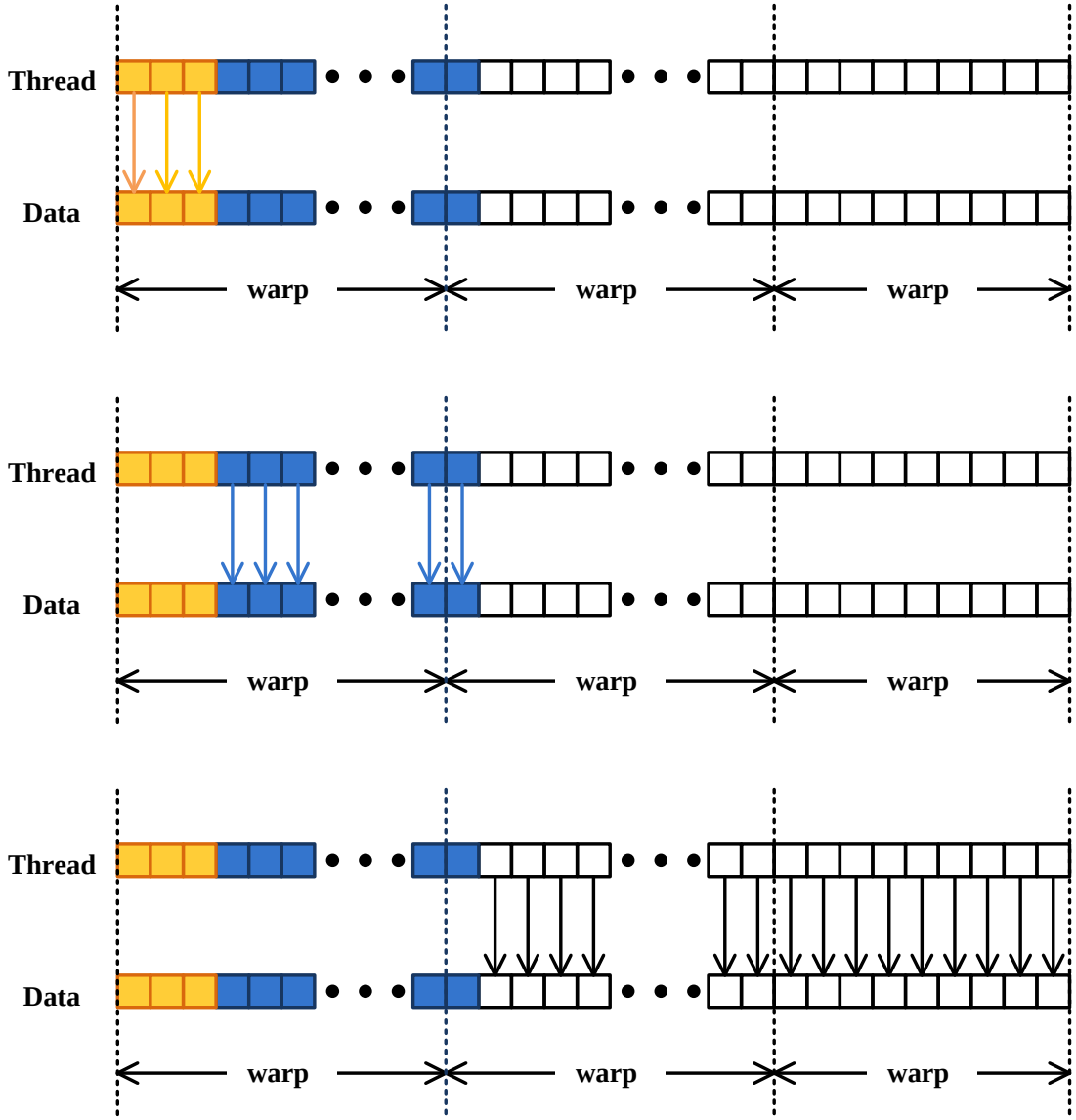

图28 相同属性的数据排列在一起，可以完成合并访问，并减少分支冲突

GPU 通过多线程合并访问来更有效的利用内存带宽。合并访问的规则比较复杂，[26] 总的来说就是 GPU 将所有对全局内存的访问都打包为 128 字节的任务进行操作，如果一个 warp 一次读取的内存正好是每个线程 4 字节，并且满足合并条件的话，就可以打包为一个 128 字节的任务进行提交。如果一个 warp 一次读取的内存是每个线程 8 字节，并且满足合并条件的话，就要打包为 2 个 128 字节的任务进行提交。只要满足合并访问条件，就可以高效的访问内存。

在迭代计算前，可以先改变数据的存储顺序，将相同属性的单元放在一起，这样每个 warp 就会取到相同属性的单元，从而在判断时执行相同的分支。同时，自由来流边界单元会读取自由来流边界条件数据，壁面边界单元会读取壁面边界

条件的数据，而流场边界单元会读取流场数据，如果各种边界条件的元胞和边读取不同的数据，很难完成合并访问，经过存储数据的改变，相同属性的单元会读取相同属性的数据，更容易满足合并访问的条件。（图 26 ，图 27 ，图 28 ）这样的改变对边界条件的设置和流通量的计算两个模块的性能有明显的提升效果。

经过这样的变换， Boundary()的计算时间由 0.65179 ms 减少到 0.4901 ms，Gradient()的计算时间由 18.452 ms 减少到 16.605 ms，Flux()的计算时间由 11.913 ms 减少到 10.042 ms，Flux_e2c()的计算时间由 18.688 ms 减少到 12.437 ms，Smooth()的计算时间由 7.1775 ms 减少到 6.9164 ms。

4. 3. 3 减少 CPU-GPU 数据传输

在 CUDA 中，GPU 不能显式地访问主机内存，只能访问显存。因此，需要将数据从主内存先复制到显卡内存中，进行运算后，再将结果从显卡内存中复制到主内存中。CUDA 提供了 GPU 存储器复制 API cudaMemcpy()，其执行方式是同步执行，能阻塞 CPU 线程，并且若前一个 kernel 函数未执行完毕，则等待。cudaMemcpy()是通过 PCI-E 总线进行数据传输，使用 PCI Express x16 时，PCI Express 1.0 可以提供双向各 4GB/s 的带宽，而 PCI Express 2.0 则可提供 8GB/s 的带宽，当然这都是理论值，相对于内存复制速度较慢，并且有巨大的延迟，频繁地使用 cudaMemcpy()函数会明显的影响计算速度。

CFD 程序中存在一些归约操作，如全局最大值，全局求和等，有研究者已经提出了归约操作的优化方法，效果显著。[5]

GPU 程序每个迭代步完成后需要把一些物理量复制到 CPU 内存中处理，包括归约操作，这些操作主要存在于 kernel_RK_Init()、kernel_drou()和 kernel_Timestep()函数中。优化后，需要归约的物理量减少为两个（全局时间不步长，全局残差），并使用 GPU 上成熟的归约方法处理。

由于操作系统是分页管理内存，从内存复制数据到显存的时候，内存可能随时会被操作系统移动，因此 CUDA 会先将数据复制到一块内部的内存中，才能利用 DMA 将数据复制到显卡内存中。如果想要避免这个重复的复制动作，可以使用 cudaMallocHost 函数，在主内存中取得一块页锁定的内存。不过，如果要求太大量的页锁定的内存，将会影响到操作系统对内存的管理，可能会减低系统的效

率。

经过减少 CPU-GPU 数据传输，使用高效率的归约操作等优化方法，RK_Init()的每步计算时间由 3.4915 ms 减少到 0.0154 ms；RKstep()的每步计算时间由 1.9393 ms 减少到 1.4692 ms。

4.3.4 减少全局内存访问

Nvidia 公司提供的编程与优化建议中提到 CUDA 编程优化应该首先提高 SM(Streaming Multiprocessor，流处理器群)占用率，由于 SM 内可以存在多个 warp，所以在指令间相互依赖，无法乱序执行的最差情况下，一个 warp 的延迟要被 block 中其他 warp 的执行所隐藏。但是提高 SM 占用率主要的方法是节省资源(如在编译时使用-maxrregcount 限制每个线程的寄存器数量)，这些措施可能造成性能下降，所以只有在确认由于活动 warp 数量不足造成严重性能下降的情况下采用。

本文中使用的求解器占用寄存器资源比较多，如果限制寄存器的使用，将会大幅影响性能，所以并没有使用限制寄存器的方法来提高 SM 占用率。

全局内存(Global memory)存在于显存中，访问延迟很大，受到显存和存储器控制器设计影响，按照一定方式访问时才能获得最大带宽，虽然 Fermi 架构的硬件上有对全局内存的 cache 机制，但是对全局内存的过多访问仍然会影响性能。

GPU 速度比较快的存储器有寄存器(Register)和共享内存(shared memory)，寄存器变量和共享变量都是显式定义，而且由于寄存器和共享内存数量有限，多余的寄存器变量会作为局部变量(Local memory)存储在显存中(访问延迟很长，有 cache 机制)，如果共享变量过多，编译会报错。

全局存储器访问比较慢，减少全局存储器的访问次数是一种有效的优化方法，将全局存储器中的数据载入寄存器变量，能够减少访存压力，加快计算速度。

另外 CUDA 也支持 C99 标准中的__restrict__关键字，__restrict__关键字可以用于限定指针，并表明指针是访问一个数据对象的唯一且初始的方式。在定义 kernel 函数时，在指针变量加入__restrict__关键字，编译器会自动优化全局存储器的访存。

4.3.5 使用更快的存储器

前面提到，寄存器数量有限，Fermi 架构的 GPU 每个线程最多支持 63 个 32

位的寄存器，多余的寄存器变量会作为局部变量存储在显存中。局部变量为线程私有，但是访存比较慢，有缓存机制。编译时加上-Xptxas=-v 选项可以查看存储器的使用情况，在编译时加上-keep 选项可以保留编译的中间文件，由于 CUDA 的编译时形成.ptx 文件，ptx 是类似汇编语言的代码，可以通过阅读 ptx 文件，查看局部存储器的使用情况。[27] 然后针对存储器的使用情况，减少不必要的变量定义，将一部分局部变量(Local Memory)用速度更快的共享变量(Shared Memory)代替。这里一定要注意，NVIDIA 在 CUDA 编程手册中提到，对于一个 warp 中的所有线程，在没有块冲突(bank conflicts)的情况下，访问 Shared Memory 和访问寄存器的速度是一样的。这句话在对计算能力 1.X 的设备描述的时候提出，对于计算能力 2.X 的设备，并没有类似的描述，在计算能力 2.X 的设备上（比如 Fermi），Shared Memory 是无法达到寄存器的访问速度的。Shared Memory 中的每个块(bank)具有每 2 个时钟周期 32-bit 的带宽，也就是说，Shared Memory 中的数据虽然最快 1 个时钟周期就能返回数据，但是实际上 Shared Memory 每 2 个时钟周期才能访问一次。在双精度计算时，虽然一个双精度浮点数能够存在 2 个 bank 中，但是一个 SM 上只有 32 个 bank，相当于一个 SM 一次只能访问 16 个双精度浮点数，而且下一次访问要至少 2 个时钟周期之后，可见 Shared Memory 使用时还是有诸多限制，要想性能接近峰值，使用寄存器是最好的选择。

不过对于 CFD 求解器而言，即使寄存器使用到达上限（Fermi 架构的上限是 63 个 32-bit 寄存器），仍然有数据会被存储在 Local Memory 里，这时，将一部分数据存储在 Shared Memory 中，可以在一定程度上缓解片外内存的延迟。由于 Fermi 架构中 Shared Memory 和 L1 Cache 共用 64KB 的空间，这里涉及到一个空间分配的问题，CUDA 编程可以在程序内部通过 API 设置每个 kernel 函数如何分配这 64KB 空间，可以有两种配置方式，一种是 48KB Shared Memory 和 16KB L1 Cache（cudaDeviceSetCacheConfig(cudaFuncCachePreferShared)），另一种是 16KB Shared Memory 和 48KB L1 Cache（cudaDeviceSetCacheConfig (cudaFuncCachePreferL1)）。

表3 优化前存储器使用情况和计算时间

函数	Register /Thread	Shared /Block	Local /Thread	Time(ms)
Gradient	63	0B	24B	16.605

Flux	63	0B	312B	10.042
Flux_e2c	52	0B	24B	12.437

表4 优化后存储器使用情况和计算时间

函数	Register /Thread	Shared /Block	Local /Thread	Time(ms)
Gradient	57	7168B	0B	9.8324
Flux	62	8192B	164B	9.0184
Flux_e2c	26	0	0B	4.6710

GPU 计算的瓶颈在于存储器，存储器的优化对运算速度的影响是最显著的。（表 3、表 4）

经过存储器的优化，Gradient()的每步计算时间由 16.605 ms 减少到 9.8324 ms，Flux()的每步计算时间由 10.042 ms 减少到 9.0184 ms，Flux_e2c()的每步计算时间由 12.437 ms 减少到 4.6710 ms，Smooth()的每步计算时间由 6.9164 ms 减少到 5.5601 ms。

4. 3. 6 使用纹理内存与常量内存

GPU 中还有纹理内存(Texture Memory)和静态内存(Constant Memory)，都有缓存机制，其中 NVCC 编译器会自动的使用静态内存，将一些计算中的只读变量放入静态内存中加快计算速度。

CUDA 支持纹理操作。在 CUDA 的 kernel 程序中，可以利用显示芯片的纹理单元读取纹理数据。纹理内存在显卡上有一定大小的专用缓存，并且通过缓存只能读取，不能写入，如果纹理缓存能够命中，那么即使不符合合并访问的规则，也有很高的效率。纹理操作来源于显卡中的纹理处理，这些特性在 CUDA 中得以继承，读取纹理时，可以利用 GPU 纹理过滤功能，也可以快速转换数据类型，例如可以直接将 32 位 RGBA 的数据转换成四个 32 位单精度浮点数。

对于已经能符合合并访问规则的数据，使用全局内存通常会比使用纹理要快。

本文使用的优化方法中并未使用纹理内存和静态内存。

不使用纹理内存的原因是纹理内存使用方法较复杂，而且要想使用纹理内存的专用缓存，必须使用纹理拾取的方法进行读取数据，而经过对全局内存的优化，对全局数据的读取次数已经减少为一次。

不使用静态内存的原因是 GPU 对静态内存的大小有限制，只有全局 64KB，对于我们的程序，显然是不够用的，64KB 无法存储任何一个物理量。

4.3.7 使用内联函数

每一次函数调用都要把原来正在运行的函数的大量数据以及状态标志等压栈，所以每一次函数调用都要消耗栈空间，而频繁的调用小函数便会大量的消耗栈空间。于是 C++中特别引入了__inline__修饰符，表示为内联函数。内联函数事实上是在调用程序中对被调用的内联函数进行展开，这样便避免了频繁调用函数对栈内存重复开辟所带来的消耗。

因为 global 函数和 device 函数只能在设备上执行，只能对设备存储器进行操作，因为当今的 GPU 设备虽有有大量的计算单元，但存储单元非常有限，如果 global 程序每次调用 device 程序时都进行压栈操作，会浪费珍贵的设备存储空间，所以编译器会根据情况将 device 函数变为内联函数。

G80 架构中默认所有的 device 函数都是内联函数，而 Fermi 架构中改变了函数的调用方式，使用 ABI（应用二进制接口，Application Binary Interface）进行函数的调用，默认的 device 函数就不是内联。

CUDA 中支持在 device 函数前加__forceinline__关键字对函数进行强制内联，这样可以节约稀缺的存储资源。

程序中只有 Boundary()、Flux()和 RKstep()函数调用了 device 函数，通过强制内联后，Boundary()的每步计算时间由 0.49111 ms 提高到 0.48587 ms，Flux ()的每步计算时间由 9.0184 ms 提高到 7.9974 ms，RKstep()的每步计算时间由 1.4700 ms 提高到 0.8040 ms。

4.3.8 循环展开

循环操作会产生一些调度的开销，在循环体比较小的时候（如循环体内只有一行代码），这些开销对速度的影响就比较明显。CPU 程序进行优化时，经常会使用#pragma unroll()宏指令或者手动将比较小的循环体打开，让其顺序执行。这种方法在 GPU 中同样有效。

4.3.9 浮点运算指令优化

GPU 的运算指令中，浮点除法指令的计算速度比浮点乘法的速度要慢很多，

如果能够将除数的倒数先算出来保存，能够加快计算速度。

比如，在 CFD 求解中，一个比较常用的物理量是元胞体积，在二维情况下即网格单元的面积，很多物理量如密度、能量密度都是通过除以元胞体积得到的，如果将元胞体积的倒数求出来，以后每次乘以这个数就可以了。

另外 CUDA 提供了一个编译选项-use_fast_math，使用这个编译选项可以加快一些浮点运算指令，代价是损失一部分精度。[6]由于 CFD 求解是一个收敛的求解过程，所以只要损失的精度不会造成结果发散，这部分优化是值得的。编译选项-use_fast_math 包括四部分-ftz、-prec-div、-prec-sqrt、-fmad，其中只有-fmad 是对双精度浮点数起作用的，-fmad 的作用是开启浮点数加乘操作，可以使用一条指令完成一个加乘的过程。

经过循环展开和指令优化，Gradient()的每步计算时间由 9.8397 ms 减少到 7.8382 ms，Flux()的每步计算时间由 9.0184 ms 减少到 7.6717 ms，Flux_e2c()的每步计算时间由 4.6710 ms 减少到 3.5211 ms。

4. 3. 10 减少 warp 占用率，优化块的大小

NVIDIA 在编程指南中提出了一个 warp 占用率(occupancy，warp 占用率是指在一个 SM 上，激活的 warp 数与 SM 所能激活的 warp 的最大值之比)的概念，并指出程序的优化应该以提高 warp 占用率为原则。

这个原则与 CPU 程序设计中以填满流水线为原则的设计思想是一致，但是这个原则决不能照搬到 GPU 程序设计中。因为 CPU 中有大量的缓存，为了填满流水线，完全可以在程序中多创建几个线程，由于线程之间有指令无关性，CPU 的流水线又支持乱序执行，流水线很容易达到较高的占用率。另外寄存器虽然访存只有一个时钟周期，在写完数据之后是无法在一个时钟周期取到更新的值的，CPU 有寄存器重命名功能，可以掩盖这个延迟。GPU 中运算核心和流水线调度都比较简单，为了达到较高的流水线占用率，NVIDIA 公司建议的方法就是在 SM 上调度尽可能多的线程，通过频繁的线程切换来掩盖读取全局内存、共享内存块冲突(bank conflict)、存储器写后读等延迟。Fermi 架构的 GPU 上每个 SM 上最多支持 1536 个线程，即 48 个 warp，当 SM 上拥有 48 个 warp 在同时调度时，warp 占用率即为 100%。

所以为了达到更高的 warp 占用率，增加每个 block 中的线程数就是理所当然的了，NVIDIA 曾经建议，计算能力 2.0 的设备上，每个 block 中要有 192 个或 384 个的线程才能有效的隐藏延迟。

隐藏延迟这个出发点是好的，但是一味地提高占用率并不是唯一的方法，在有些时候，也并不是最好的方法。

我们知道，GPU 中各种存储器资源十分宝贵，Fermi 架构的 GPU，每个 SM 上有 32768 个 32-bit 的寄存器，每个 SM 上最大只有 48KB 的 Shared Memory。如果遵循 NVIDIA 的建议，提高 warp 占用率到 100%，每个 SM 上有 1536 个线程，那么每个线程只能分到 32768/1536=21 个 32-bit 寄存器和 49152/1536=32 字节的 Shared Memory，一共 116 字节。超出这 116 字节的存储空间，只能放入 Local Memory，Local Memory 属于片外内存，访存延迟约为 500 个时钟周期，虽然 Local Memory 能够被缓存，但是 GPU 上的缓存空间更加有限（全局二级缓存 768KB，每个 SM 的一级缓存与 Shared Memory 一共 64KB，在 Shared Memory 使用 48KB 是，一级缓存只有 16KB），所有的缓存平均到每个线程上只有 (786432/14+16384)/1536=47 个字节，这还是在所有缓存线都被占满的情况下！所以，在 warp 占用率 100%的情况下，一个线程能够高速访问的存储空间的理论最大值只有 116+47=163 字节。如果无法再高速的存储器上存储，就只能在片外的 Local Memory 空间存储数据。

但是如果 kernel 函数比较复杂（需要的存储空间大于 163 字节），需要的 Local Memory 很多，通过高速的切换线程是无法隐藏延迟的，原因如下：

Fermi 架构的 GPU Tesla C2050 双精度浮点数计算能力是 515Gflops，双精度浮点数占 8 个字节，所以所有 SP 的数据吞吐量是 4120GB/s，但是显存带宽只有 144GB/s，如果 kernel 函数需要大量的访问 Local Memory 并无法在缓存中命中的话，线程会访问片外的 Local Memory 空间，显存带宽是无法同时满足所有 SP 的访存需求的。所有 SP 的数据吞吐量是显存带宽的 4120/144=28.6 倍，理想情况下，读取片外 Local Memory 的概率在 1/28.6=3.5%以下是可以隐藏延迟的。

所以，在 warp 占用率 100%的情况下，如果能够隐藏读取片外存储的延迟，一个 kernel 函数的线程中最经常访问的存储空间要放到 163 个字节中，并保证这

163 字节之外的存储空间访问概率低于 3.5%。对于一般的程序设计而言，这是很难的，所以通过提升 warp 占用率来接近峰值的程序往往都是程序功能比较单一并且简单，单个 kernel 函数比较短的程序，如线性代数库、FFT 等一些数学库。

如果使用显式格式进行求解，将 kernel 函数优化到使用这么少的存储资源是不可能的。所以这种情况下太高 warp 占用率就无法获得很好的加速效果。我们初期的 GPU 优化中已经精简了 kernel 函数使用存储器的数量，我们认为，在 kernel 函数使用存储器数量较多的情况下，应该以减少片外 Local Memory 和 Global Memory 的访问为首要原则，适当地降低 warp 占用率，提高每个线程的寄存器和 Shared Memory 使用量。

我们测试了不同块大小的计算性能（表 5），当线程块大小为 64 时，此时 warp 占用率只有 33%，但是计算时间最短，这和我们上面的分析是一致的。

表5 不同大小的块的计算时间

块大小	32	64	128	192	256	384	512
时间(ms)	28.364	20.598	21.762	22.938	23.880	24.611	26,383

4. 3. 11 编译选项设置

NVCC 在编译 CUDA 程序中有很多编译选项可以选择，比较常用的编译选项有：

-Xptxas=-v 打印编译结果中各个 kernel 函数对各类存储器的使用情况，此选项不会影响编译结果，对生成的可执行文件没有任何改变。

-Xptxas –dlcm 此选项对片外内存起作用，决定访问全局内存时只使用 L2 Cache (-Xptxas –dlcm=cg)还是使用 L1 和 L2 Cache (-Xptxas –dlcm=ca)。如果使用 L1 和 L2 Cache，那么片外内存使用 128-bit 的对齐规则，而如果只使用 L2 Cache，片外内存使用 32-bit 的对其规则并且有些数据可以跳过 L1 Cache 直接从 L2 Cache 到达寄存器，某些情况下会比使用 L1 Cache 获得更好的效果。

-Xptxas –abi 是否使用 ABI（Application Binary Interface，应用二进制接口）。ABI 描述了应用程序（或其他类型）和操作系统之间或其他应用程序的低级接口，Fermi 架构中开始支持 ABI，使得显卡与操作系统和应用程序有更多的交互。而且

ABI 对函数的调用也有更好的支持，使用 ABI 之后，CUDA 可以更加高效地执行非内联函数的调用，所以没有显式声明内联的函数会默认为非内联，这个设置是与 G80 架构不同的。但是经过测试，大多数情况下，非内联的函数执行效率要低于内联函数的执行。所以我们在编译时将此选项设置为-Xptxas –abi=no。

-maxrregcount 设置一个线程中最大可以使用寄存器数，超过这个限制的变量会放入 Local Memory，具体哪个变量放入寄存器，哪个变量放入 Local Memory 是在编译阶段就确定好的，运行阶段不会改变。

-use_fast_math 可以加快一些浮点运算指令，代价是损失一部分精度。编译选项-use_fast_math 包括四部分-ftz、-prec-div、-prec-sqrt、-fmad，其中只有-fmad 是对双精度浮点数起作用的，-fmad 的作用是开启浮点数加乘操作，可以使用一条指令完成一个加乘的过程。我们发现在我们的程序中这个选项并没有带来预想的加速效果，所以我们并没有加入这个选项。

4. 3. 12 减少 CPU-GPU 同步

GPU 中 kernel 函数与 CPU 的函数是异步执行的。CPU 在调用 kernel 函数后会立刻返回。如果没有 cudaThreadSynchronize()这样的同步函数，CPU 会同时调用多个 kernel 函数 而 Fermi 架构支持同时执行多个 kernel 函数(最多 16 个)，并且 kernel 间切换时间大大降低。因此一组流内的不同 kernel 可以充分占用 GPU 的执行单元。

经过优化，每个时间步的迭代时间由 19.750 ms 减少到 19.087 ms。

第五章 性能测试及分析

5.1 各种优化方式对性能的影响

使用中科院超算中心的 Tesla GPU 集群中 c0304 节点进行性能测试。

硬件配置：

AMD Phenom 9850 处理器，2.5G 主频，8G 内存。

两块 Tesla C2050 显卡，每块 3G 显存，448 个 CUDA Core，主频 1.15GHz。

软件配置：

RHEL 4.1.2-44，内核版本 2.6.18-128.el5.

GCC 4.1.2 编译器，NVCC 4.1 编译器，编译选项为 O2。

将各种优化方法对结果的影响列为下表（表 6）：

表6 各种不同的优化方法对每个部分的优化效果(单位 ms)

	RK_Init	Boundary	Gradient	Flux	Flux_e2c	Smooth	RKstep
未优化	6.1331	0.62661	21.418	13.644	25.197	8.248	1.9349
数据结构转换	4.1289	0.65179	18.452	11.913	18.688	7.1775	1.9338
更改存储顺序	3.4915	0.49010	16.605	10.042	12.437	6.9164	1.9393
减少数据传输	0.0154	0.49144	16.605	10.042	12.437	6.9164	1.4692
存储器优化	0.0154	0.49111	9.8397	9.0184	4.6710	5.5601	1.4700
内联函数	0.0152	0.48587	9.8324	7.9974	4.6749	5.5610	0.8040
循环展开	0.0152	0.48383	7.8382	7.8714	4.0272	4.5104	0.6866
指令优化	0.0155	0.47889	6.3544	7.6717	3.5211	4.2032	0.5831
优化块大小	0.0153	0.47235	5.6237	6.7375	2.9758	4.0904	0.5872
编译选项	0.0152	0.46601	5.4148	6.3603	2.9180	3.9816	0.5828

各种优化方法对整体的影响列为下表（表 7）：

表7 各种不同的优化方法对整个程序的优化（单位 ms）

	计算时间	加速比
未优化	77.361	10.306
数据结构转换	60.180	13.249

更改存储顺序	51.909	15.359
减少数据传输	48.054	16.592
存储器优化	31.209	25.547
内联函数	29.515	27.013
循环展开	25.495	31.273
指令优化	22.938	34.759
优化块大小	20.598	38.708
编译选项	19.750	40.370
减少同步	19.087	41.772

通过表 6、表 7 可以看出，存储器优化、数据结构转换这两种优化方式是相当有效的，存储器优化是大家一直采用的优化方式，我们在优化过程中，定量分析了 Fermi 架构不同于上一代产品的新特性，没有使用大家普遍认为的 warp 占用率作为指标，大量使用 register 和缓存，限制使用 shared memory，减少进程并发数，减轻了真正的瓶颈—全局内存—的带宽压力。我们结合非结构网格的特点，更改网格数据的存储顺序，更进一步提升了 GPU 运算速度。

5. 2 不同规模的算例加速情况

我们还测试了 GPU 程序对不同规模的算例的加速情况。（表 8）

表8 不同规模的算例计算速度和加速比（单位 ms）

网格量	3558	13513	51800	93806	211838
CPU 计算时间	45.365	186.03	811.75	1602.6	3720.4
GPU 计算时间	1.4080	4.7469	19.109	43.255	101.27
加速比	32.220	39.191	42.480	37.049	36.737

可以看到，在网格量比较大的时候加速效果稍好。而且，GPU 轻量级线程和共享存储的结构，非常适合大规模的并行扩展，计算规模的瓶颈可能在 GPU 有限的全局内存上。CFD 的计算需要考虑边界条件，这部分工作不能完全适应 GPU 架构，不过我们使用了一些技巧避免了这个开销。

5. 3 计算结果与 CPU 程序及实验数据对比

在 13513 网络量的网格上进行计算，并做出压力云图（图 29 ），同时提取翼

型上下表面的压力数据，与 CPU 程序及实验数据对比（图 30 ）。

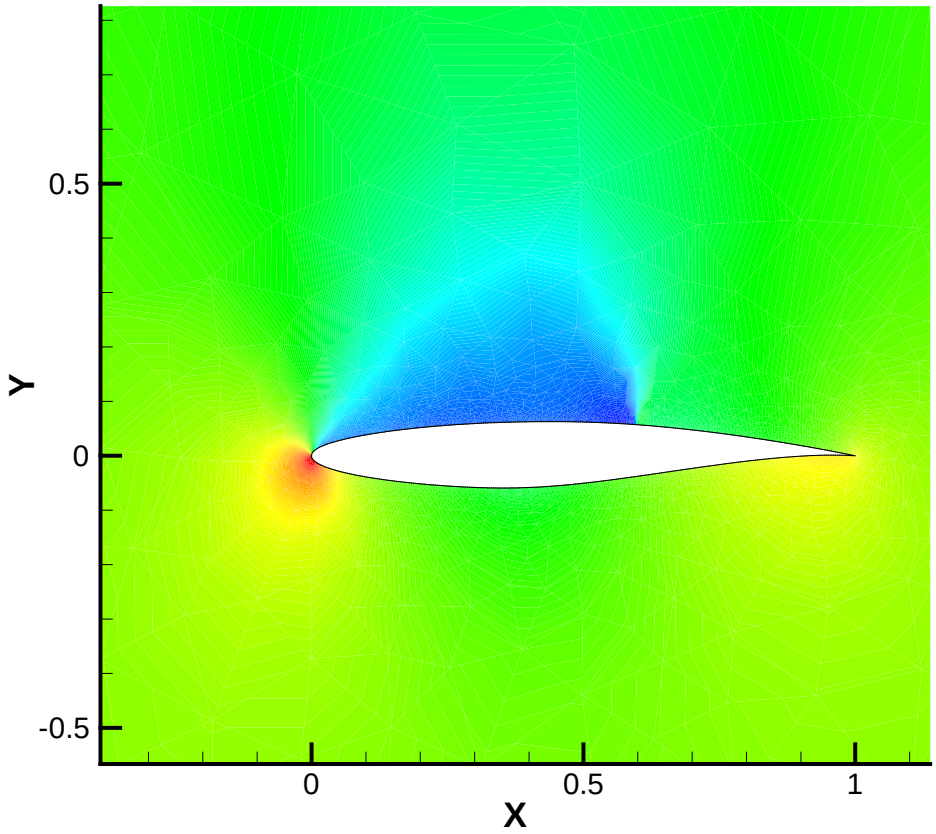

图29 RAE2822 翼型的压力云图

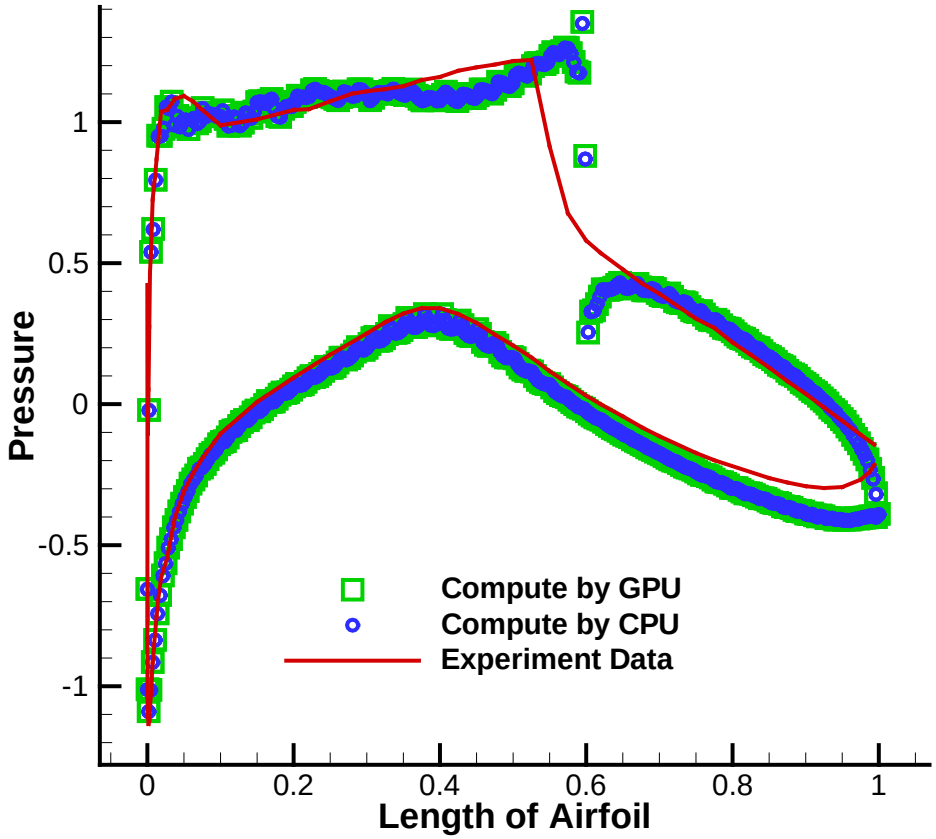

图30 RAE2822 翼型上下表面压力

如图 30 ，GPU 求解器计算的结果与 CPU 计算的结果几乎完全相同，但与实验结果有些差距，激波位置明显后移，下表面后缘压力有偏差，主要原因是本文工作所采用的求解器是 Euler 方程求解器，没有考虑粘性项的计算。希望在后续过程中能够有所改进。

第六章 总结与展望

6.1 总结

随着 NVIDIA、AMD、Intel 等公司的大力推动，众核计算成为硬件架构上的突破点，也成为摩尔定律得以延续的重要产品。

GPU 通用计算发展多年，已经很多应用在 GPU 上获得很好的加速效果，但是非结构网格下的数值模拟（如 CFD）的加速效果往往不如结构网格下的程序。而且 GPU 通用计算发展日新月异，很多新的体系结构会对 GPU 算法的优化起着重要的影响，在 CUDA 发展初期被大家公认的一些优化方法可能已经不适合目前的架构，比如 Fermi 架构中每个 SP 的平均 Shared Memory 带宽比前一代产品还要低，提高 warp 占用率并不适用于所有应用程序等等。GPU 程序的优化一定要随着硬件架构的发展和应用领域的不同而与时俱进，才能充分发挥 GPU 并行计算的潜能。本文研究非结构网格 CFD 求解器在 GPU 上的优化，做了以下几点工作：

1. 研究 GPU 通用计算的发展历程和未来趋势，分析 GPU 的硬件架构是如何从一个建模着色部件演变成一个具有视频渲染功能的并行处理器的。

2. 分析有限体积法 CFD 求解器的性能瓶颈和存储瓶颈，使用 CUDA 并行体系，将原有基于 CPU 的求解器移植到 Fermi 平台上。

3. 着重分析非结构网格的数据特点，并结合 GPU 的硬件架构，使用调整非结构网格存储顺序等方法，将 GPU 程序性能进一步提升，最后达到约 40 倍的加速比。

6.2 展望

在本文的工作中，以下工作还需要加强：

1. 本文选用的求解器是基于二维非结构网格的 Eular 方程求解器。二维问题不具有代表性，而且 Eular 方程无法处理粘性问题，在工业应用中受到诸多限制。计算流体力学最终还是要回归到应用上，研究能够适用大型飞机设计的高性能求解器才是本文工作的最终目的。

2. 还有很多优化方法等待挖掘。由于硬件环境的限制，本文所做的工作没有采用 NVIDIA 提供的一些优化工具如 computeprof 等，如果能高效的利用优化工具，可以使程序的性能更上一层楼。同时，可以尝试多 GPU 运算。

NVIDIA 公司在各大半导体厂商伤透脑筋解决单个处理器核心的流水线调度问题时，跳出这个思维定势，使用 SIMD 来提高计算峰值，使用大量线程的切换来隐藏访存和流水线的延迟，使 GPU 上的通用计算程序也有了不错的计算效率，总的来说，基于 CUDA 模型的 G80、GT200、GF100(Fermi)等架构是非常成功的创新产品。

Fermi 的众多特性，已经明明白白告诉用户，这不是仅为游戏或者图形运算设计的 GPU，而是面向图形和通用计算综合考虑的成果。全局 ECC 设计、可读写缓存、更大的 Shared Memory、甚至出现了分支预测概念……这次 Fermi 抛弃长期使用的“流处理器”称谓方式，更明确体现了 NVIDIA 的意图。

6.3 众核计算，路在何方？

NVIDIA 敢于提出图形性能和通用计算并重，说明 GPU 设计的重点和难点都在通用计算而非图形。因为一颗演化了十几年的 GPU 肯定能够做好自己的老本行图形计算，但要做通用计算，需要更加强大的线程管理能力，更强大的仲裁机制，丰富的共享 Cache 和寄存器资源以及充足的发射端……如果做不好这些东西，GPU 永远都是 PC 中的配角，永远都是一颗流处理器，一颗协处理器。

以 GPU 为主的众核计算要想在市场上有所突破，以下几个技术必须有很大的发展。

1. 更好的解决存储带宽问题。单纯的加大显存位宽不是治本的办法，未来的众核计算机可能会使用复杂的存储结构，如 UMA、NUMA 等等，未来的 GPU 需要更大的缓存和寄存器资源。

2. 更加丰富的指令集和流水线调度。丰富的指令集是为了支持复杂的运行逻辑，而优化的流水线调度是为了提升指令的运行效率，虽然目前 GPU 的流水线还没有成为瓶颈。

3. 适合众核计算的操作系统。目前的操作系统，包括多线程操作系统，都不是为众核计算准备的，Intel 的 MIC 上可以运行 Unix 和 Linux，但是并不能很好的发挥众核计算的潜能。我认为这个是相当必要的，因为只有有了操作系统，GPU 才能真正的摆脱 CPU，完成多个 GPU 之间的交互，实现更为复杂的 MIMD 算法及应用。

4. 面向并行计算的编程语言和编译器。并行编程语言从 PVM、HPF 发展到现在基于 C 和 Fortran 的 MPI、OpenMP、pthread 等并行库，目前尚未出现很完美的并行编译器，但是 Intel 编译器的指令级并行做的已经很不错了。并行编程语言和并行库的发展的基本原则就是对代码的改动越少越好，程序员基本上都是用脚投票，哪个编程语言和编译器好用，哪个方便，大家都会选哪个。CUDA 终究会像汇编一样，变为深度优化的工具，NVIDIA 目前大力推广 OpenACC，希望未来能够在并行编程语言中占领一席之地。

5. 融合处理器。将 CPU 和 GPU 整合在芯片内部，统一的总线控制模式。2008 年 4 月，半导体产业爆发大规模口水战，先是 Intel 在 IDF 上宣称显卡产业将会消亡，而后黄仁勋宣布 GPU 将取代 CPU 的地位。几年后，Intel 已经发布了 Sandy Bridge，AMD 也有了 APU，NVIDIA 已经发布了 Tegra。不过各家的融合之路看起来不太顺畅，Intel 的融合层次过于浅表，AMD 的 APU 没有得到应用环境的职称，NVIDIA 的融合处理器还在纸面上。

参考文献

[1] Bergman C M, Vos J B. Parallelization of CFD codes. Computer Method in Applied Mechanics and Engineering, 1991, 89(1-3): 523-528.

[2] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman etc. Brook for GPUs: Stream Computing on Graphics Hardware. ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2004, 2004, 23(3).

[3] Erich Elsen, Patrick LeGresley, Eric Darve. Large calculation of the flow over a hypersonic vehicle using a GPU, Journal of Computational Physics, 2008, 227(24):10148-10161.

[4] Julien Thibault, Dnanc Senocak. CUDA implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows. American Institute of Aeronautics and Astronautics, 2009-758.

[5] Joe Myre, Stuart D. C. Walsh, David J. Lilja, Martin O. Saar. Accelerating Lattice Boltzmann Fluid Flow Simulations Using Graphics Processors. Parallel Processing, 2009. ICPP '09. 2009: 550-557.

[6] Athanasios S. Antoniou, Konstantinos I. Karantasis, Eleftherios D. Polychronopoulos, John A. Ekaterinaris. Acceleration of a Finite-Difference WENO Scheme for Large-Scale Simulation on Many-Core Architectures, American Institute of Aeronautics and Astronautics, 2010-0525.

[7] 张兵，韩景龙，基于 GPU 和隐式格式的 CFD 并行计算方法，北京：航空学报，2010, 2(31): 249-256.

[8] 封卫兵，张武，基于 CUDA 的 LBM 方法研究. 高性能计算发展与应用，2009, 4: 50-53.

[9] 董廷星， 基于 GPU 的机翼跨音速绕流的数值模拟，中国科学院计算机网络信息中心，2010.

[10] 李森，计算流体问题的GPU加速研究，中国科学院计算机网络信息中心，2011.

[11] Andrew Corrigan, Fernando Camelli, Rainald Lohner, John Wallin. Running

Unstructured Grid Based CFD Solvers on Modern Graphics Hardware. 19th AIAA Computational Fluid Dynamics, June 22-25, San Antonio, Texas, 2009.

[12] 张林波，迟学斌，莫则尧，李若. 并行计算导论. 北京：清华大学出版社，2006.

[13] 吴望一. 流体力学. 北京: 北京大学出版社，1982。

[14] 傅德薰，马延文. 计算流体力学. 北京: 高等教育出版社, 2002.

[15] Rajkumar BUyya, High Performance Cluster Computing, Architectures and Systems, Volume 1, Upper Saddle River: Prentice Hall, 1999. 1-3.

[16] 白中英, 杨旭东. 并行计算机系统结构. 北京: 科学出版社. 2002.

[17] G. Wilson. Practical Parallel Programming. Cambridge, MA: MIT Press, 1995.

[18] John L. Hennessy, David A. Patterson, Computer Architecture: A Quantitative Approach. San Francisco: Morgan Kaufmann, 2006.

[19] Ananth Grama, George Karypis, Vipin Kumar, Anshul Gupta. Introduction to Parallel Computing. Boston: Addison-Wesley, 2003, 11.

[20] 郑飞. 超标量与超流水线混合结构微处理器 Pentium. 微处理机，1994(4): 1-5.

[21] 李三立. 超标量 RISC—SuperSPARC 体系结构设计特点. 小型微型计算机系统, 1993, 14(4): 1-9.

[22] 吴迪，蔚喜军，徐云. 局部时间步长间断有限元方法求解三维欧拉方程，计算物理，2011, 28(1): 1-9

[23] 韦祥文. MPI 平台下二维欧拉方程数值解法. 西安: 西北工业大学，2003, 3.

[24] J. Blazek. Computational Fluid Dynamics: Principles and Applications. Oxford: Elsevier. 2001, 303

[25] Jameson, A., Baker, T.J, Solution of the Euler Equation for Complex Configurations, AIAA Paper, 1983, 83-1929: 293-302.

[26] NVIDIA. CUDA C Programming Guide, Version 4.1. (20112-11). http://developer.nvidia.com/nvidia-gpu-computing-documentation

[27] NVIDIA. Parallel Thread Execution ISA version 3.0. (2012-1). http://developer.nvidia.com/nvidia-gpu-computing-documentation

[28] 改变翻天覆地，史上最全 Fermi 架构解读.

http://space.itpub.net/22893636/viewspace-630918

致 谢

值此论文完成之际，我由衷地感谢我的导师陆忠华研究员三年来对我的支持和帮助，无论是 2011 年去美国的交流访问，还是参加各种会议，以及研究选题，陆老师都尽可能的为我创造条件，正是陆老师的这种支持和信任，给我一个自由宽松的学术环境，也让我有机会接触更多的专家学者，接触最新的技术和思想。在超算中心的三年里，陆老师行为人师，学为世范，为我今后的工作树立了优秀的榜样。

特别感谢迟学斌研究员三年以来一直关心着我的学习、科研和生活，迟老师专业知识渊博，理论水平深厚，学术态度严谨，工作作风认真，这些都给我留下了深刻的印象，并对我产生了深刻的影响。感谢他引导我走上高性能计算的研究道路。

感谢王彦棡老师、赵永华老师、王龙老师、吕海峰老师和姜金荣老师在我工作和学习中的帮助和指导。

感谢张鉴老师、陈刚、胡晓东、聂宁明、马文鹏、王磊、刘冰、杜夏威、张晓蒙在平时的科研工作中给予我悉心的指导和鼓励，还为我的进步创造诸多条件。每每回首与你们朝夕相处的日子，都让我倍感鼓舞。

在论文前期的研究过程中，力学所的李新亮老师和梁贤老师给予了非常多的帮助。李老师学识渊博，总是不厌其烦的为我讲解计算流体力学的知识和程序设计等工作，在此，我再一次表达对李老师的感谢。

感谢汪云海、仲倩黎、曹宗雁、高晋芳、董廷星、常红旭、李森、李虹、阳卫清、郭建勇等师兄师姐对我的帮助，尤其是董廷星和李森师兄，他们确实是爱科研、爱编程、爱师弟师妹的好师兄！感谢蔡长青、贾伟乐、陈建华、张亚南、吴响、张拓宇、刘阳、汪丽杰等同学，感谢他们三年来的陪伴，怀念 315 的日子，那是我三年来最美好的回忆。

感谢明尼苏达大学的 David Yuen 教授和张淑霞老师，中科院地球物理所的刘洪老师，中国地震局的祝爱玉老师。你们的关心和鼓舞，让我受益终生。感谢北京邮电大学的肖井华老师，肖老师是我科研道路的启蒙者，虽然我已经离开北邮

三年，但是他仍然一直非常关心我的学习和工作，并教我如何管理时间和规划学术生涯，虽然我没有如他所愿继续深造，但是我会记住他的教诲，努力工作。

感谢刘利萍、王珏、何平、刘芳、阳卫清在研三这一年对我的帮助。

感谢超算中心的每位成员，与大家一起学习和工作的这段日子是我一生中难以忘怀的时光。感谢 2009 级的所有兄弟姐妹，时光虽短，友情无限，愿各位在今后的道路上都事业有成。

感谢我最亲爱的爸爸、妈妈和妹妹，感谢他们二十多年来一直陪伴着我，他们的爱是我不断前进的动力，是我最温暖的港湾，祝他们永远幸福快乐。

在网络中心的研究生生活让我受益很多，感谢所有关心、支持和帮助过我的每一个人！祝你们永远健康、幸福。

作者简介

【 基本情况 】

姓名： 宋慎义 性别：男 民族： 汉

出生日期：1987.10.28 籍贯：河北邢台 政治面貌： 中共党员

【 学历及工作经历 】

- 2009.09-2012.06 中国科学院研究生院 中国科学院计算机网络信息中心 计算机软件与理论专业 工学硕士
- 2011.01-2011.05 明尼苏达大学 地质与地球物理学院/明尼苏达超级计算中心 短期交流访问学者
- 2005.09-2009.06 北京邮电大学 理学院 应用物理学 理学学士

【 论文发表情况 】

宋慎义，王彦棡，刘冰，陆忠华。基于 GPU 的非结构网格 CFD 求解器的设计与优化。科研信息化技术与应用，2012,3(1)

【 参加科研项目情况 】

2010.01~2011.01 国家高技术研究发展计划项目“面向大型飞机设计的万核级流场数值模拟软件”

2011.01~2012.06 国家自然科学基金项目“飞行器高雷诺数下气动优化及动边界问题高精度快速算法研究”

【 攻读学位期间获奖情况】

2010 年度中国科学院计算机网络信息中心年终考核三等奖

2011 年度中国科学院计算机网络信息中心年终考核三等奖

Email: songshenyi@gmail.com