# 面向集成电路电容提取与热分析的GPU并行算法研究

（申请清华大学工学硕士学位论文）

培 养 单 位 ： 计算机科学与技术系
学　　科 ： 计算机科学与技术
研 究 生 ： 翟 匡 亚
指 导 教 师 ： 喻 文 健 副 教 授

二〇一三年五月

# Research on GPU-based parallel algorithms for the capacitance extraction and thermal simulation of integrated circuits

Thesis Submitted to

**Tsinghua University**

in partial fulfillment of the requirement

for the professional degree of

**Master of Engineering**

by

**Zhai Kuangya**

**( Computer Science and Technology )**

Thesis Supervisor : Associate Professor Yu Wenjian

**May, 2013**

# 关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容。

本人保证遵守上述规定。

**（保密的论文在解密后应遵守此规定）**

作者签名：_____　　导师签名：_____

日　　期：_____　　日　　期：_____

# 摘 要

在过去的几十年以来，集成电路的性能一直按照摩尔定律在发展，然而，在尝试使集成电路性能在接下来的一段时间内继续按照过去的轨迹发展的过程中，越来越多的难题出现在了研究人员的面前。一方面，由于特征尺寸的减小及集成度的增大，互连线电容已经逐渐超过门电容，成为影响信号时延、信号完整性的主要因素。因此在进行准确的电路仿真之前，必须准确地提取互连寄生电容值。另一方面，为了提高集成度与性能，三维集成电路成为目前看来最有可能成功的技术。但它的散热问题非常突出，需要在物理设计与验证中加以考虑。随着集成电路的规模逐渐增大，对其进行电容提取和热仿真成为了很有挑战性的大规模计算问题。

为了适应图形处理市场的需求，通用图形处理器(GPU)已经发展成为了一种众核，高度并行化的设备。和普通处理器相比，GPU 具有更高的浮点数运算能力，更大的内存带宽。由于其具有的这些优点，如何将GPU应用在非图形计算领域中的具有高运算量的问题中，已经成为了当前的一个热点研究问题。

本文围绕着如何将GPU应用到电容提取与芯片热仿真等CAD问题中，以便提高计算效率，开展了一系列研究。一方面，本文提出了适合GPU 架构的并行随机行走电容算法流程、GPU友好的跳转概率存储结构、以及处理多层介质互连结构的新型随机行走方法等技术。对实际工艺互连结构的电容提取实验显示，本文所提出的技术能使单介质电容提取的速度加快70倍以上，使多介质电容提取速度加快20倍以上。另一方面，本文提出了适合GPU加速的并行GMRES预条件算法，并将它应用到含液态冷却管道的堆叠式三维芯片的热瞬态仿真中，实验结果表明，使用GPU的预条件GMRES求解器比使用CPU的SuperLU方程求解器快63倍，大大提高了三维芯片热仿真的计算速度。

**关键词：** 电容提取　　图形处理器　　随机行走　　三维芯片　　热仿真

# Abstract

During the past several decades, the performance of integrated circuits was progressing according to the Moore's Law. However, in the effort to make integrated circuits progressing with previous trace, challenges emerge. On one hand, with the shrinking of the gate size and the increasing of the integration level, the parasitic capacitance of interconnects surpassed the capacitance of gates, and became the dominant factor influencing signal delay and integrality. As the result, accurate capacitance of interconnects are required to be extracted before the detailed simulation. On the other hand, 3-D circuits are envisioned to be the most promising candidate to reduce packaging size and increase integrate level. While the most significant problem in way for 3-D circuits is the heat dissipation issue.

To fulfill the needs of graphic processing, Graphic Process Units (GPUs) has evolved into many-core, massively parallel device which has much higher computing throughput than the multi-core CPU. Many researches have done to leverage the high computing throughput of GPUs to accelerate non-graphic tasks. How to tackle GPUs to accelerate the time consuming CAD algorithms has also became an hot research in EDA community.

This thesis researches on the efficient leveraging of the computing horsepower of GPUs to accelerate the computation intensive problems such as capacitance extraction and thermal simulation. This thesis proposed three kernel iterative algorithm flow of floating random walk algorithm for capacitance extraction. Together with the GPU-friendly data structure which largely reduce memory accessing delay, numerical experiments results show that, the GPUs based FRW can be more than 70 times faster than the CPU counterpart for single dielectric problem, and more than 20 times faster for multiple dielectric problems. This thesis also proposed a GPU based preconditioned GMRES solver, which can be applied to the transient thermal simulation problem for 3-D circuits with liquid coolant. Numerical results show that with suitable preconditioners, the GPU-based GMRES program can be as faster as 63 times than the CPU-based SuperLU for large thermal simulations.

**Key words:** Capacitance Extraction Graphic Process Units 3-D circuits Floating Random Walk Thermal Simulation

# 目 录

# 主要符号对照表

| | |
|---|---|
| ALU | 算术逻辑单元(Arithmetic Logic Unit) |
| CAD | 计算机辅助设计(Computer-Aid-Design) |
| EDA | 电子设计自动化(Electronic Design Automation) |
| FRW | 悬浮随机行走(Floating Random Walk) |
| CPU | 中央处理器(Central Processing Unit) |
| GPU | 图形处理器(Graphic Processing Unit) |
| SIMT | 单指令多数据流(Single Instruction Multiple Threads) |
| SP | 流处理器(Streaming Processor) |
| SM | 流多处理器(Streaming Multiprocessor) |
| PDF | 概率分布函数(Probability Distribution Function) |
| CDF | 累积分布函数(Cumulative Distribution Function) |
| ICPA | 反向累积概率向量(Inverse Cumulative Probabilities Array) |
| GFT | 格林函数表(Green Function Table) |
| WVT | 权值表(Weight Value Table) |
| TSV | 硅通孔(Through Silicon Via) |
| GMRES | 广义最小剩余法(Generalized minimal residual method) |
| FMM | 快速多极算法(Fast Multi-pole Method) |
| MEMS | 微电子机械系统(Micro-electromechanical Systems) |
| FinFET | 鳍型场效应晶体管(Fin Field-Effect Transistor) |
| AINV | 基于近似逆的(Approximate Inverse-based) |
| CUDA | Common Unified Device Architecture |

# 第 1 章 引言

## 1.1 集成电路互连电容提取

随着集成电路制造工艺的进步，互连线之间的耦合电容对电路性能的影响越来越显著。这些影响主要体现在以下方面，首先，由于特征尺寸的减小，互连线之间的距离也在减小，导致导线间的电容值增大；其次，由于用户对电路功能需求的增加，在相同面积上集成的门越来越多，这导致了互连线总长度的增加，从另一方面增大了互连线路上的电容值。如图 1.1所示，在特征尺寸达到250nm及以下的时候，互连线上的延时已经超过门延时，成为影响电路性能的主要因素。而今天的集成电路制造的特征尺寸远远小于250nm，集成电路互连电容值更是不能忽视，如果想要对电路进行准确的时延仿真，必须要对电路进行准确的电容建模。

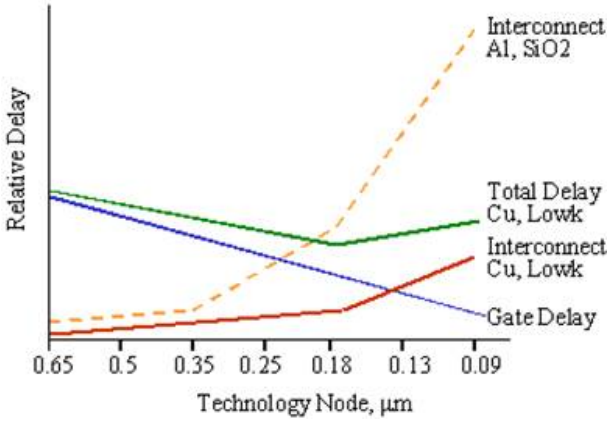



图 1.1 互连线延时及门延时随着特征参数发展的变化

根据摩尔定律[1]，集成电路上的晶体管数目每18个月翻一番，在今天的集成电路制造工艺下，每一块芯片都有数以百万计的门器件，用来连接这些器件的互连线的数量也十分巨大，因此，电容提取是一个典型的高度计算密集型的问题。目前对电容进行提取的方法主要分为解析模型法和数值模拟法。

### 1.1.1 数值模拟法

数值模拟法以电磁场理论为基础，使用数学手段求解静电场中的拉普拉斯方程组，从而能够得到准确的结果。数值模拟法可以分为确定性方法和随行性方法

两大类。

确定性方法一般以解线性方程组为基础，其又可分为有限差分法及边界元法等。有限差分法使用差分代替拉普拉斯方程中的微分，将连续变化的变量进行离散化，从而得到差分方程组的数学形式，通过求解差分方程组便得到静电场的信息。有限差分法的不足之处是其需要将整个空间进行离散化，对于大规模的三维问题，有限差分法的内存消耗及计算复杂度往往十分巨大。边界元方法从边界积分方程出发，有别于有限差分法所进行的空间离散，边界元方法只对边界进行离散，得到边界积分方程组，并通过求解所得到的边界积分方程组得到静电场的信息。边界元方程能够将问题的维度降低，然而边界元方法的不足是它所生成的线性方程组往往是稠密的，造成较大的方程求解时间。确定性方法对小规模的问题十分有效，但是当问题规模很大的时候，其内存消耗和计算时间都迅速增大。基于确定性的方法有很多加速算法，例如快速多极算法FMM(Fast Multiple Pole)[4]是一种著名的基于边界元的电容提取算法，快速多级算法将边界元按照空间位置划分为不同的集合，如果两个不同的集合的距离很远，那么，这两个集合之间的相互作用可以在不损失精度的情况下等效到两个集合的等效中心的相互作用；如果两个集合距离较近，则使用传统的边界元算法求解。FMM算法可将算法的复杂度降到$O(nm)$，这里$m$ 是导体数量，$n$ 是边界元的数目。著名的电容提取软件*FastCap* 就是基于快速多极算法实现的。

蒙特卡洛方法不同于确定性方法，它不需要生成和求解线性方程组，而是使用蒙特卡洛算法计算积分值。这样做的优点是节省存储线性方程组所需消耗的内存，使其更适合求解规模偏大的问题。此外在使用蒙特卡洛算法计算积分值的时候，结果的精度是可以根据采样步数计算出来的，因此蒙特卡洛算法能够获得任意所需的精度。正是由于蒙特卡洛算法所具有的这些优点，使得其逐渐成为今天电容求解领域中的主流算法。代表性的求解电容值的蒙特卡洛方法是悬浮随机行走(Floating Random Walk, FRW)算法，它由Iverson等人在1990年的时候第一次提出[5]。

### 1.1.2 解析模型法

解析模型法以实验、测量或者数值模拟法得到的准确的结果作为基础，首先通过建库，插值拟合等近似方式得到电容值的解析公式。当需要计算一个未知结构的电容时，解析模型法首先对该结构进行几何及材料信息分析，得到该结构所对应的电容解析公式中的相关参数，然后将这些参数代入到对应的拟合好的解析公式之中，计算得到该结构的电容值。解析模型法的优点是其速度快，对每个电容结构进行提取的时候无需求解线性方程组。然而很显而易见的是，解析模型法

的精度不足，难以对复杂的三维结构进行精确地计算。此外，解析模型法通常只对当前工艺中常见的互连结构准确建模，当新工艺出现时，其对新结构的计算结果往往是不准确的，需要花费很长时间对新工艺进行建模及调试。例如当鳍型场效应晶体管(Fin Field-Effect Transistor, FinFET)出现时，因为现有的解析电容模式库未对对其结构做过专门的建库，故对其进行拟合的结果就不是可靠的。例如，本人曾使用数值模拟法对使用FinFET 的电路进行参数提取，以弥补解析模型法的不足[3]。尽管解析模型法具有上述的缺点，但是由于其他的数值方法尚无法对全芯片级的电路在合理的时间内完成电容提取，解析模型法在商业软件中得以广泛使用，例如Synopsys®公司的Star-RCXT[2]寄生参数解决方案中对电容的提取流程就是基于解析模型法。

随着互连线对集成电路的影响越来越显著，电路仿真时对电容精度的要求越来越高，解析模型法等工作已经越来越不能够满足准确仿真的需求。另一方面，随着计算机计算能力的突飞猛进，以及新的快速数值模拟方法的提出，尤其是蒙特卡洛法的发展，数值模拟法正在越来越被重视。

## 1.2 集成电路热分析

在很长一段时间里，集成电路的设计们在设计集成电路的时候，他们并没有意识到他们在设计电路的时候同时也设计了一个热系统。在过去的很多情况下，芯片上的热量对芯片性能的影响都是可以忽略的。然而，随着集成电路制造工艺的逐渐进步，芯片功率密度逐渐增大，芯片温度对芯片性能的影响正在变得越来越显著，芯片功率密度随着MOS管沟道长度变化情况如图 1.2所示。半导体产业分析人士认为，在沟道长度降到0.1微米以下时，温度对电路性能的影响将变得不可忽略[6]，而今天的集成电路的沟道长度远远低于这个标准，更应当引起我们的重视。

随着集成电路的封装密度及总功耗的逐渐增大，芯片的温度也变得越来越高。芯片上的温度上升对集成电路的性能和可靠性有着重要的影响。当芯片的温度过高超过阈值时，甚至会对电路造成结构性的损坏。因此，在芯片设计完成之后，需要进行热仿真，确保其在正常工作时温度不至于过高，之后才能投入生产。对集成电路所进行的热分析必须是快速而准确地，只有这样才能够减少产品开发时间，在第一时间进入市场，获得最大收益。然而，集成电路的规模往往是十分巨大的，并且其规模还在按照摩尔定律而持续地增大。为了克服巨大计算量与要求的快速的仿真流程之间的矛盾，目前的很多热分析模型采用近似的模型来进行热分析，这些模型往往是有缺陷且不准确的，例如在[8] 中所使用了平行互连矩阵

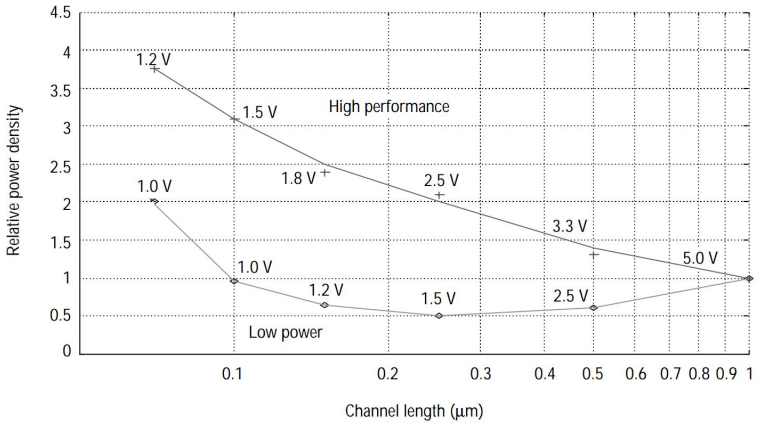
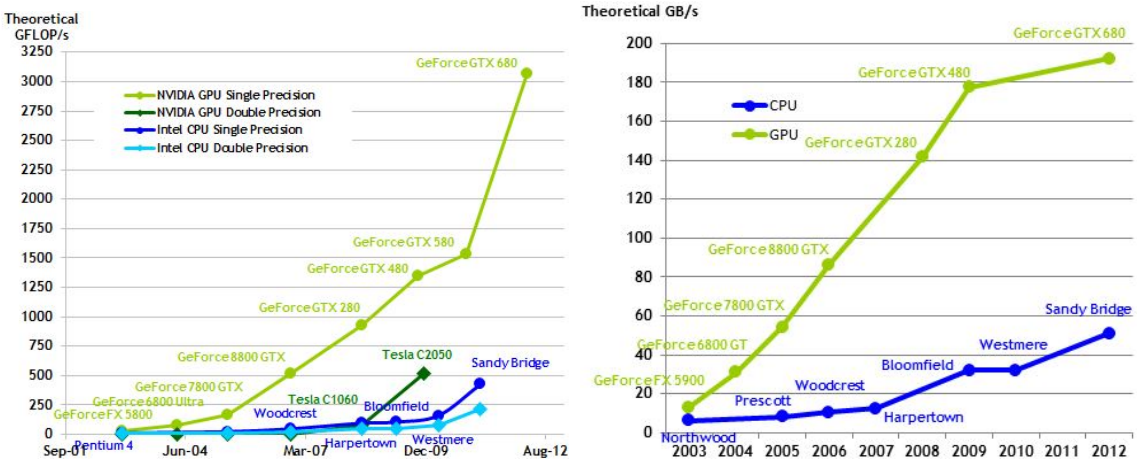



图 1.2 集成电路功率密度随着沟道长度的变化趋势[6]

的温度阻值模型，该模型忽略了其边缘热耗散，从而导致了误差的出现；在[9]中使用保角映射的方法得到解析模型，而该模型只是对单独的导线分析时准确，而有多导线相互作用的时候则有很大的误差。除了近似模型之外，还有的工作从热传导方程出发，使用有限差分法等数值手段，对集成电路进行准确的建模，这些工作往往能够得到准确的结果，然而，受限于其生成以及求解方程组的庞大的计算量，它们往往只能对集成电路的一小部分作分析，而缺乏进行全芯片仿真的能力。

## 1.3 基于图形处理器的并行计算

图形处理单元(Graphic Processing Units, GPU)是在计算机显示卡中的图形处理单元，其主要功能是3D画面运算和图形加速，为了满足上述功能，目前GPU已发展成一种高度并行化的，多核的计算平台。与CPU 相比，GPU具有非常高的浮点计算能力和内存带宽，如图 1.3所示。在GPU的芯片设计中，更多的晶体管被用在了数据处理方面，即用来制造算术逻辑单元(Arithmetic Logic Unit, ALU)，而不是像中央处理单元(Central Processing Unit, CPU)一样将很多的晶体管用于数据缓存，分支预测等方面。因此，GPU 更适合处理流程简单，算术指令密集的任务。GPU的存储主要包括寄存器(Register)，片上共享存储(Shared Memory) 以及片下的全局内存(Global Memory)等。每个流多处理器(Streaming Multiprocessor, SM) 对应一块共享内存，同一块流多处理器上的计算单元能够访问该共享内存。共享内存因为处于芯片上，所以其速度非常快，和寄存器的访问速度相近，然而，共享内存的容量很小，例如对于Fermi$^{TM}$架构的图形处理器，每块流处理器上只有48KB 的共享内存。全局内存是一种全局的片下存储，所有的流处理器上的计算核心都可以访问全局内存。全局内存的容量比较大，一般在几个GB的量级，然而，全局内存




(a) 每秒浮点运算次数　　(b) 内存带宽

图 1.3　GPU的浮点运算能力与内存带宽与GPU的比较

访问的速度十分缓慢，访问延时长达数百个时钟周期，在实际的使用中，全局内存访问的时间开销往往成为整个程序性能的瓶颈。因此，在实际使用全局内存的时候，对全局内存访问所做的优化十分重要。

Common Unified Device Architecture(CUDA™)是Nvidia® 公司发布的基于图形处理器的并行计算环境。通过在C 语言的基础上进行扩展，使得图形处理器能够方便地计算通用的非图形类的问题。通常的CUDA程序是由一个或多个内核(Kernel)组成，每个内核启动很多个可以并行地在图形处理器上运行的线程(Thread)。一定数量的线程组成一个线程块(Thread Block)。CUDA 将每个线程块分配给一个流多处理器(Streaming Multiprocessor, SM)执行，对应地每个线程分配给流多处理器中的一个流处理器(Streaming Processor, SP)，CUDA中线程组织与对应硬件之间的关系如图 1.4 所示。在运行的时候，每个线程块中的32个线程组成一个*warp*。CUDA中任务的执行遵循单指令多线程 (Single Instruction Multiple Threads, SIMT) 的模式，warp 是CUDA 程序调度的最小单元。在相同的时间内，所有的在同一个warp 中的线程只能执行相同的指令，但是可以处理各自分别的数据。倘若不同线程需要处理不同的指令，那么这些线程必须串行地执行，如图 1.5 所示。图形处理器使用单指令多线程并行模式的原因是在设计和制造图形处理器的时候，将更多的晶体管用于数据处理而非指令处理上，在每个流多处理器上只有一个指令处理单元，即一个流多处理器上的所有流处理器共享一个指令处理单元。因此，在基于图形处理器的程序设计中，应该尽可能地减少不同线程处理不同指令的情况，使得可以并行执行的线程个数尽可能地多。

现在的GPU已经能够做大部分CPU所能作的事情，很多人让相信未来的处理器架构发展的趋势是基于中央处理器与图形处理器的整合。例如AMD® 公

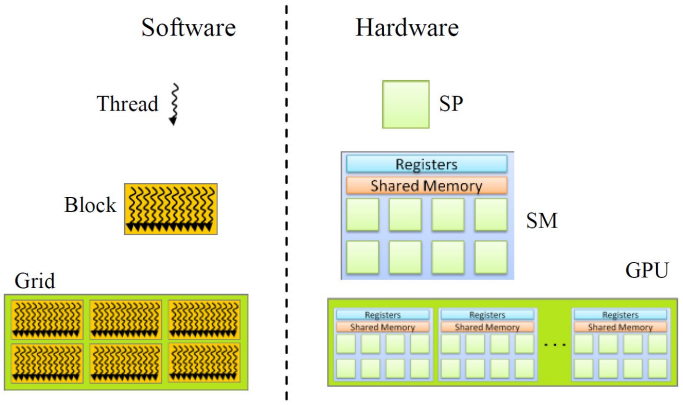



图 1.4 CUDA中线程及硬件的对应关系

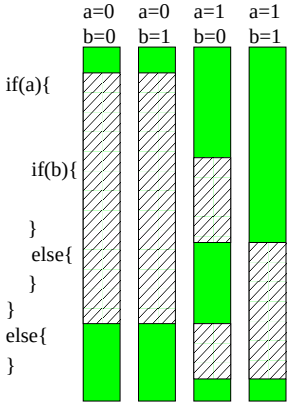



图 1.5 线程指令流出现分歧时的执行

司正在研究的加速处理单元(Accelerated Processing Units, AMD)就是一种将中央处理单元及图形处理单元集成到同一块晶片上，协同计算、彼此加速，从而大幅提升电脑运行效率的下一代处理器。从最近超级计算机的发展趋势也可以看出GPU在计算能力和能源利用率方面的巨大潜力。目前世界上速度最快的超级计算机是由美国Cray® 公司承建的泰坦(Titan)[10]，其为世界上第一台以通用图形处理器(GPGPU) 为主要数据处理单元的超级计算机。泰坦使用AMD提供的皓龙(Opteron)中央处理器链接NVIDIA提供的Tesla图形处理器进行协同运算，其使用18,688颗中央处理器和相同数量的图形处理器，达到了27petaFLOPS的浮点计算能力。除了具有超高的运算性能之外，由于大量使用了GPU，泰坦还具有超高的能源利用效率，其每消耗1W 的电量能够获得2142.77megaFLOPS 的浮点运算性能，在根据性能功耗比进行排名的全球500强超级电脑排名Green500中也名列第三[11]。此外，在最近刚刚落幕的2013年度ASC13总决赛上，清华大学夺得总冠军，他们在3000瓦功耗限制下创造了7.58Tflops的Linpack 成绩[12]，为国际同类大赛最好成绩，他们使用的也是CPU-GPU的异构集群。

## 1.4 本文主要工作

本文对如何将GPU使用到电容提取与热分析的应用中，来减少计算时间提高计算效率这一课题进行了研究，主要贡献如下：

- 提出了针对随机行走算法的三阶段方法，将随机行走流程分为三个小阶段，不同阶段之间使用图形处理器的片上内存进行数据交换。实验结果表明，三阶段方法能够有效减小不同线程之间的指令分歧，增大并行度，从而加快计算速度；对单介质结构，提出了适合图形处理器的数据结构，从而减少了图形处理器的内存访问时间。针对多介质结构，提出了一种新型的采样方法，

使得所提出的对单介质的数据结构能够使用到多介质问题之中；针对大规模互连电容提取问题，提出了多线网并行提取策略来增大并发的线程数量，从而进一步地加快了计算速度。

- 从基本的固、液体热传导方程出发，使用有限差分法对含液体冷却管道的三维芯片进行了热建模，并使用基于GPU 的带有预条件技术的GMRES 算法对其进行了瞬态热仿真。数值实验表明，采用基于GPU 的GMRES求解器，在对三维芯片进行瞬态热仿真时，比使用CPU上的SuperLU并行求解器快63倍。

本文共分为五章，后续各章主要内容如下：

- 第二章介绍了集成电路电容提取及热分析的研究现状。
- 第三章介绍了使用GPU加速电容提取的随机行走算法，以及实验结果。
- 第四章介绍了面向三维芯片热仿真的GPU并行算法，以及实验结果。
- 第五章是总结与展望，对本文的工作进行总结并对未来的工作进行展望。

在本文最后是参考文献，个人简历以及论文发表情况。

# 第 2 章　集成电路电容提取及热分析的研究现状

本章首先介绍用于集成电路电容提取的主流算法，及其优缺点，重点是随机行走算法与边界元算法。其次，本章还介绍了对含有液态冷却管道的三维堆叠式芯片的热建模方法。在本章的最后部分介绍了目前一些在EDA领域中成功使用GPU 进行加速的算法研究现状。

## 2.1　悬浮随机行走电容提取算法

### 2.1.1　算法原理

随机行走方法的基础是下面的积分公式

$$\phi(r) = \oint_S G_\phi(r, r^{(1)})\phi(r^{(1)})dr^{(1)} \tag{2-1}$$

其中$\phi(r)$是在点$r$处的格林函数。$S$是一个包围点$r$的闭合面，被点$S$包围的区域又常被称为转移区域，$G_\phi(r, r^{(1)})$被称为格林函数，其值非负并且在$S$表面的积分为1。因此，根据蒙特卡洛积分的原理，求$\phi(r)$ 点电势的问题可以看成在表面$S$ 上的点$\phi(r^{(1)})$ 采样统计。

在公式 (2-1)中如果$\phi(r^{(1)})$处的电容值未知，我们可以把公式 (2-1)重复运用来求解$\phi(r^{(1)})$ 的值，并得到下面的嵌套的积分公式:

$$\phi(r) = \oint_{S^{(1)}} G_\phi^{(1)}(r, r^{(1)})dr^{(1)} \oint_{S^{(2)}} G_\phi^{(2)}(r^{(1)}, r^{(2)})dr^{(2)} \cdots \oint_{S^{(k+1)}} G_\phi^{(k+1)}(r^{(k)}, r^{(k+1)})dr^{(k+1)} \tag{2-2}$$

其中，$S^{(i)}, (i = 1, \ldots, k+1)$是以$r^{(i-1)}$为中心的第$i$ 个转移区域，$G_\phi^{(i)}(r^{(i-1)}, r^i), (i = 1, \ldots, k+1)$是关联点$r^{(i-1)}$与点$r^{(i)}$之间的电势的格林函数。公式 (2-2)可以看成一个随机行走的过程：对于行走的第$i$ 步，以点$r^{(i-1)}$为中心，构造一个转移区域，然后从该转移区域上按照格林函数$G_\phi^{(i)}$ 作为采样概率密度取点$r^{(i)}$，然后类似地进行第$i+1$步，以及更多的行走，直到点落在一个电势已知的表面。在单介质问题中，如果第$i$ 个转移区域的表面$S^{(i)}$被规格化到单位大小，格林函数$G_\phi^{(i)}(r^{(i-1)}, r^i)$ 对所有的$i = 1, \ldots, k+1$都是一样的，因此，我们可以事先计算好格林函数的的值，并将其保存到内存中，每次行走的时候可以直接读取。因为格林函数可以预先计算好，这样在随机行走的时候就不需要花费计算资源在格林函数计算方面，随机行走算法的主要计算开销成为了几何操作。

公式 (2-2)给出的是使用随机行走计算电势的方法，如果需要计算互连线之间的耦合电容，还需要得到导体感生电荷和导体电势之间的关系。假设在一个结构中有$M$个导体，我们需要求解的目标是第$j$个导体与其它导体的耦合电容。我们首先需要在导体$j$周围做一个包围导体$j$的，并其且不和任何其它导体相交的闭合表面，我们将该表面称为高斯面(Gaussian Surface)，根据高斯定理有

$$Q_j = \oint_{G_j} D(r) \bullet \hat{n}(r) dr = \oint_{G_j} F(r)(-\nabla\phi(r)) \bullet \hat{n}(r) dr \tag{2-3}$$

其中$Q_j$是在导体$j$表面的感生电荷量，$D(r)$是在点$r$ 处的电偏移量，$F(r)$是点$r$处的介电常数，$\hat{n}(r)$是在点面$G_j$在点$r$处的法向量。将式 (2-1)代入式 (2-3)可以得到：

$$Q_j = \oint_{G_j} F(r) g \oint_{S^{(1)}} \omega(r, r^{(1)}) G^{(1)}(r, r^{(1)}) \phi(r^{(1)}) d(r^{(1)}) dr \tag{2-4}$$

其中$g$是一个仅与空间位置相关的常数，$\omega(r, r^{(1)})$是权值，其定义如下：

$$\omega(r, r^{(1)}) = -\frac{\nabla_r G^{(1)}(r, r^{(1)}) \bullet \hat{n}(r)}{g G^{(1)}(r, r^{(1)})} \tag{2-5}$$

其中$\nabla_r$是一个作用在点$r$处的梯度操作符，其满足

$$\oint_{G_j} F(r) g dr = 1 \tag{2-6}$$

我们可以观察到，式 (2-4) 中，第一个积分可以看成是在高斯面$G_j$上的一个随机采样，第二个积分则可以根据前面式 (2-2)所描述的随机行走的过程来计算。至此，对导体感生电荷量的计算可以通过用对电势的计算得到，对电势的计算则能够通过随机行走得到，再根据电容的定义

$$C = QU \tag{2-7}$$

我们就可以通过随机行走算法得到所需计算的电容值。

### 2.1.2　随机行走算法中的一些加速技术

我们可以很简单地推出随机行走算法所消耗的时间为

$$T_{total} = N_{hop} \times N_{walk} \times T_{hop} \tag{2-8}$$

其中$T_{total}$为算法总时间，$N_{walk}$为行走步数，$N_{hop}$为平均每次行走所需要进行的跳转次数，$T_{hop}$为每次跳转所消耗的时间。提高随机行走算法的性能在于减小公式 (2-8)中的每个乘子。

首先我们来考虑如何减小$N_{hop}$。理论上，$S^{(i)}$可以是任何一个包围点$r^{(i-1)}$且不包含任何导体的闭合区域，然而，转移区域$S^{(i)}$ 的形状对$N_{hop}$ 有着至关重要的影

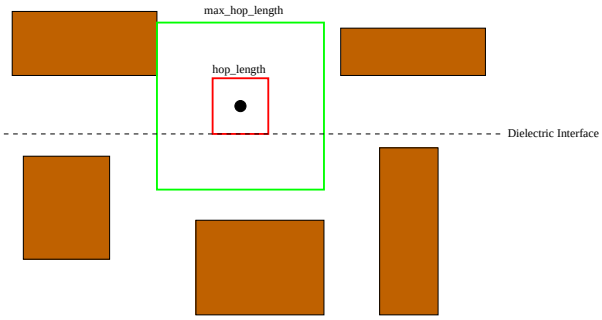



图 2.1　当有多层介质时最大转移区域与实际转移区域的示意图，其中绿色方框为最大转移区域半径，红色方框为实际转移区域半径

响。在Iverson 等人最开始提出随机行走算法的时候，他们所用的转移区域的形状是球形。对于球形转移区域，格林函数有着简单的解析表达式[13,14]。然而，在集成电路中，为了减小相邻导线的串扰，在数字电路中，互连线往往都是曼哈顿结构的，即导线的表面都是呈垂直或水平方向的，而球形转移区域和曼哈顿形状的互连线只能在切点处相交。因此，对于球形转移区域来说，行走点很难落到导体表面而终止，每次采样所需的跳转次数特别长，计算效率不理想。现代的求解互连线电容的随机行走算法一般使用立方体转移区域，其与互连线可以在几个面上相交，因此在跳转的过程中行走点落到导体上而终止的概率较大，每次行走的平均$N_{hop}$值较小，效率较高。当立方体转移区域只位于一个均匀的介质层之中时，格林函数有着解析的表达形式，在[15] 中推导了当转移区域位于单介质中且为立方体的时候的格林函数的推导。

除了转移区域的形状之外，每次生成的转移区域的大小对$N_{hop}$也有着重要的影响。根据Rollins的报告[16]指出，随机行走算法的时间有以下关系：

$$T_{total} \propto \frac{max\_hop\_length^2}{hop\_length^2} \tag{2-9}$$

其中$max_hop_length$为理论上可生成的最大转移区域的半径，$hop_length$为实际使用的转移区域的半径。如图 2.1所示。

图 2.1中，由于有介质交界面的存在，使用单介质的转移区域时，最大半径只能与介质交界面相切，而若使用多介质转移区域，转移区域的最大半径能够大大扩展，使得算法效率提升。然而，当转移区域穿越多层介质时，格林函数并没有简单的解析公式，一些数值方法可以用来得到多介质问题中的格林函数。例如在[17]中，使用有限差分法计算得到穿越两层介质的转移区域的格林函数，本文中当转移区域穿越多层介质的时候，我们采用[17]中的算法。类似单介质，我们可以将多介质的格林函数预先计算并存储在磁盘当中，在随机行走算法开始的时候将格林函数表读取到内存中以供需要时使用。

算法 1是使用立方体转移区域及穿越多层介质格林函数之后的算法，图 2.2 是与之相对应的随机行走过程。

**Algorithm 1** 适用多介质情况下的随机行走电容提取算法

1: 读入预先计算好的对单介质和多介质的转移概率和权值函数表；
2: 构造一个包围主导体且不与其它环境导体相交的立方体表面，即高斯面；
3: $C_{ji} := 0; npath := 0;$
4: **repeat**
5: $npath := npath + 1;$
6: 从高斯面上取一个点 $r^{(0)}$，然后然后生成一个以该点为中心的立方体转移区域（可以跨越多层介质）$T$，从表面$T$按照转移概率取一个点$r^{(1)}$，然后按照预先读入的权值函数表计算出该次行走对应的权值$\omega$；
7: **repeat**
8: 根据导体在空间的位置分布情况构造最大的转移区域；
9: 按照转移概率从转移区域表面取点；
10: **until** 当前点触碰到导体表面；
11: $C_{ji} := C_{ji} \cdot (npath - 1) + \omega / npath$
12: **until** 精度满足要求；

接下来我们考虑如何通过减小$T_{hop}$来提高随机行走算法的效率。在随机行走算法中需要进行成千上万次跳转，因此，减少每次跳转所需要的时间对提高算法性能有着重要的影响。每次跳转多需要进行的核心操作在于如何查找到与当前点最近的导体，并计算得到最近距离，然后用该距离生成转移区域。在电容提取的实际应用中，对于大规模的电路，有非常多的导体需要查找。如果使用遍历的方法，算法所消耗的时间将是非常巨大的。因此，高效的随机行走算法需要有高效的空间管理技术，使得每次跳转的时候需要查找的导体数量尽可能少。

目前主流的空间管理技术可分为基于树状结构与基于网格结构两种。树状结构的空间管理技术主要包括八叉树及K-D树两种。八叉树[18] 是一种著名的用来进行空间管理的数据结构，在八叉树的数据结构中，每个节点管辖一定的三维空间，这个节点保存所有的有可能离该空间中的任意一点最近的候选导体的集合。当候选导体的数量超过一个阈值的时候，则将该三维空间裂解成八个子空间，对应的八叉树的节点也进一步地分为八个子节点，持续这样的过程，直到空间每个节点中的候选导体的数量小于预先设定的阈值为止。当八叉树建立好之后，在每次查找离一个点最近的候选导体集合的时候，只需要从八叉树根节点开始查找包含该点的叶子结点，叶子结点存储这有可能离这个点最近的所有候选导体。图 2.3显

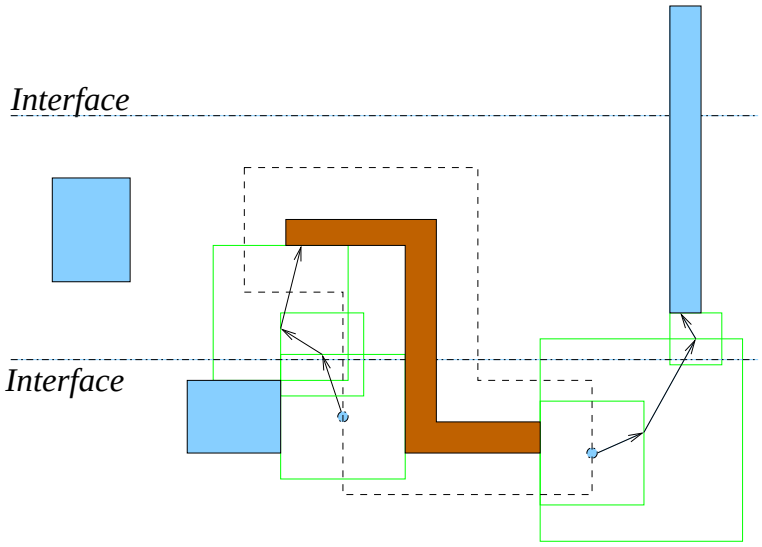



图 2.2　当使用立方体转移区域，并使用跨越多层介质格林函数时的随机行走过程的示意图，其中绿色方框代表当前生成的转移区域

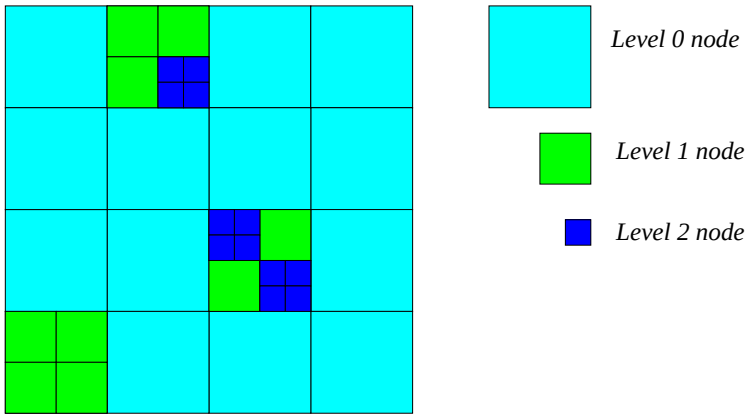



图 2.3　使用八叉树管理空间的示意图，图中显示的是二维空间的示意图，三维空间情况类似

示的是使用四叉树管理空间的一个例子，而八叉树为四叉树在三维空间上的扩展，可以类似地得到。

八叉树的一个问题是每个字节点的深度不一样，导体分布越密集的区域，叶子结点的深度相应地更深。对于深度大的叶子结点，有时候需要花费相当可观的操作才能从根节点遍历寻找到叶子结点。相比之下，K-D树[19]在每次裂解子空间的时候，不是将子空间等大小划分，而是按照每个空间的候选导体数目等分。这样相对于八叉树来说，K-D树往往更加平衡，即所有叶子结点的深度往往更加平均。一个使用K-D 树管理空间的例子如图 2.4 所示，注意该例子中使用K-D树能够生成一棵平衡树，而若使用八叉树则生成的树是不平衡的。

使用K-D树能够减少查找叶子结点所需要的操作，但是K-D 树的问题在于在遍历树的时候的每次跳转的操作比较费时，需要查询如何当前空间是如何分裂

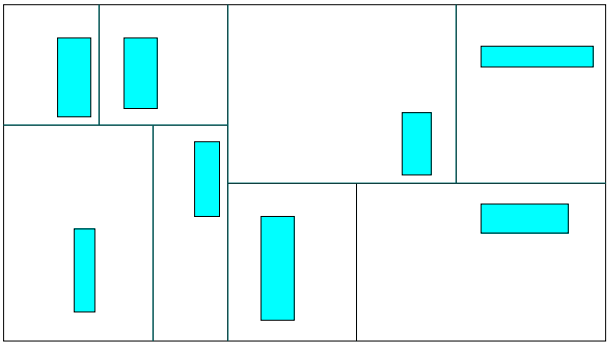


图 2.4　使用K-D树管理空间的示意图，图中显示的是二维空间的示意图，三维空间情况类似

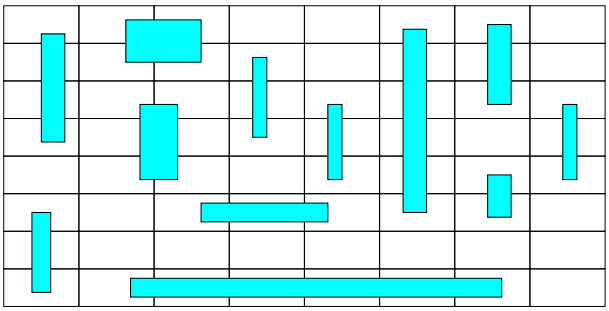


图 2.5　使用网格法管理空间导体的例子，图中显示的是二维空间的示意图，三维空间情况类似

成子空间的，相比之下，八叉树则只需要将空间进行等分。因此，K-D树在实际使用的时候效率并不能保证一定比八叉树效率高。使用树状结构对于导体分布密度不均匀的时候效率很高，然而，在实际的集成电路布局中，会尽量使得各处布线密度均匀。对于规模非常大并且各处布线密度相对均匀的情况下，工业界中往往使用网格的方法来对空间进行管理，例如Synopsys®的随机行走电容计算程序Rapid3d™中，就是使用基于网格的方法来管理空间中的导体[16]。网格法中，首先将空间均匀分成等大小的网格，每个网格记录与其相交的导体的编号。这样，在定位一个点所在的网格的时候，因为各个网格大小均匀，则可以直接通过坐标计算一步得到点所在网格的编号。又由于实际布线中，导体分布相对均匀，各个网格所包办的候选导体数量往往相近。网格法由于其数据结构非常简单，对大规模输入数据支持好，而常被工业界所采用。

## 2.2　边界元电容提取算法

边界元算法(Boundary Element Method, BEM)是一种用来求解线性偏微分方程组的数值方法，它将微分方程转化为边界积分方程，然后使用矩量法[20](Method of Moment, MoM)进行求解。边界元法可应用于流体力学、声学、电磁学和断裂

力学等诸多工程科学领域(在电磁学中，边界元法是矩量法的别名)。边界元法按照将微分方程转化为积分方程的过程中的边界变量及积分公式的不同，可以分为直接边界元法和间接边界元法。

### 2.2.1　边界元方法的理论

在每个均匀的介质之中，静电场满足下面的方程式：

$$\begin{cases} \nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \quad \text{in } \Omega_i, i = 1, \cdots, M \\ u = u_0 \quad \text{on } \Gamma_u \\ q = \frac{\partial u}{\partial \boldsymbol{n}} = 0 \quad \text{on } \Gamma_n \end{cases} \tag{2-10}$$

其中$u$与$q$分别为电势和电场强度；$\Omega_u$为狄利克雷边界，通常为导体表面，在该边界上电势$u$ 已知，为预先设定导体的偏压值；$\Omega_n$ 是纽曼边界，通常为介质区域的外边界，在该边界上的法向电场强度已知为0；$\boldsymbol{n}$表示在纽曼边界条件上的单位外法向量。

在不同介质交界面上电势与电场强度满足连续性方程：

$$\begin{cases} \varepsilon_a \frac{\partial u_a}{\partial \boldsymbol{n_a}} = -\varepsilon_b \frac{\partial u_b}{\partial \boldsymbol{n_b}} \\ u_a = u_b \end{cases} \tag{2-11}$$

其中$\varepsilon_a$与$\varepsilon_b$表示介质$a$与介质$b$ 的介电常数。

使用直接边界元方法，可以将式 (2-10)所示的静电场中的偏微分方程组转化为下面的边界积分方程组：

$$c_s u_s + \int_{\partial\Omega_i} q^* u d\Gamma = \int_{\partial\Omega_i} u^* q d\Gamma \tag{2-12}$$

其中$u_s$是在点$s$的电势；$c_s$是一个与点$s$附近几何特性有关的常数；$u^*$是拉普拉斯方程的基本解；$q^*$ 是$u^*$沿着外边界法向的倒数；$\partial\Omega_i$是第$i$ 个介质区域的边界。通过将方程 (2-12)离散之后，能够得到下面的离散形式的边界积分方程：

$$c_s u_s + \sum_{j=1}^{N_i} u_j \int_{\Gamma_j} q_k^* d\Gamma = \sum_{j=1}^{N_i} q_j \int_{\Gamma_j} u_k^* d\Gamma \quad k = 1, \cdots, N_i \tag{2-13}$$

其中$\Gamma_j$是在离散后的边界$\partial\Omega_i$ 上的第$j$个边界元，$u_j$与$q_j$分别是$\Gamma_j$ 上的电势及法向电场强度。

### 2.2.2　线性方程组的生成

方程 (2-13)中的积分可以分为奇异积分与非奇异积分两种。当积分源点落在积分区间之内的时候，即$k = j$的时候，积分为奇异积分，文献[21]中给出了关于奇

异积分的计算方法：

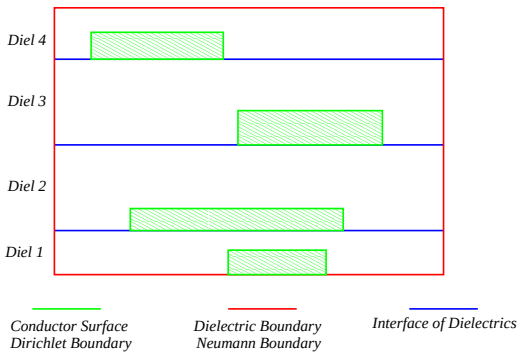
$$\begin{cases} \int_{\Gamma_i} u_i^* d\Gamma = & 0.5 \\ \int_{\Gamma_i} q_i^* d\Gamma = & \frac{1}{\pi} \times r_1 \times (log(\frac{1}{r_1}) + 1) \end{cases} \tag{2-14}$$

其中$r_1$是积分区间的长度的一半。

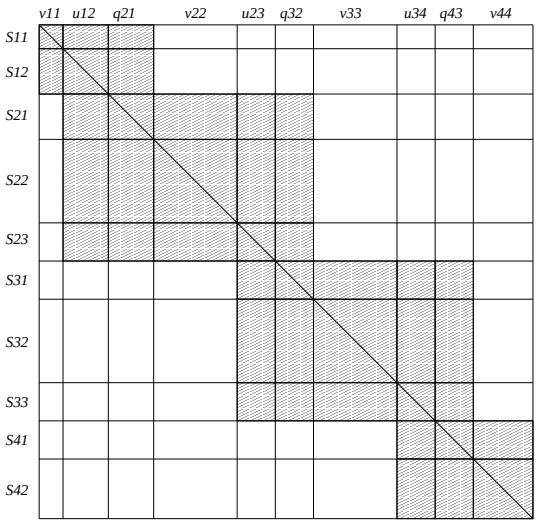
如果在式 (2-13)中，$k \neq j$，那么积分变为非奇异积分，通常地，数值积分方法，例如高斯积分[22]可以被用来求解非奇异积分。通过积分可以求出方程 (2-13)之中的各项系数，然后可以通过移动方程中的未知变量$u$与$q$的位置，使得所有的未知量位于方程的左边，而已知量位于方程的右边，最终可将边界积分方程写成线性方程组的标准形式

$$\boldsymbol{Ax} = \boldsymbol{b} \tag{2-15}$$

在集成电路工艺中，不同介质分层分布，如图 2.6(a)所示，根据边界元算法的理论，不同介质区域中的元素只在介质交界面上相关，通常可以通过调节未知量的编号，使得所有的未知量集中在主对角线两侧，如图 2.6(b)所示。




(a) 分层集成电路工艺横截面视图




(b) 调整未知变量之后的线性方程组非零元分布

图 2.6　对于介质分层工艺的集成电路中调整未知变量之后的线性方程组非零元分布

当所生成线性方程组规模较大的时候，GMRES[23]等迭代解法因具有速度快，内存消耗低等优点[24]，通常可以用来对其进行求解。

## 2.3　基于有限差分的三维芯片热仿真

### 2.3.1　问题背景

三维芯片被认为是最有可能使集成电路性能按照摩尔定律[1]继续进步的技术。基于硅通孔(Through Silicon Vias, TSV) 的三维堆叠技术使得三维芯片能够将异构的器件，例如计算单元、内存及模拟器件等，集成在一起，使得在单位面积上集成的晶体管数量按照摩尔定律持续增加。然而，基于硅通孔的三维芯片也带来了一些其他的问题，例如怎么将芯片内部的热量导出到芯片外部等问题。三维芯片堆叠的层次越多，芯片内部的热流密度也就越高，例如，对于某些三维芯片，其热量密度能够达到250w/cm$^2$[25]，如果不能够有效地将芯片内部的热量转移出去，会使得三维芯片的性能及可靠性受到限制[26]。因此，三维芯片的冷却技术成为当前研究的热点问题。传统的散热技术采用硅通孔与散热片，已有非常多的相关论文得到发表，如[27–31]等。垂直硅通孔不仅能够作为信号传导通路，其具有良好导热性，也能够将热量在垂直方向上传导，从而使芯片内部的热量能够传导到热衬底上。然而，一方面，硅通孔的数量有限，并不能够完全解决三维芯片散热问题，另一方面，硅通孔并不能很好地改善芯片在水平方向温度不均匀的问题，芯片在水平方向的散热也值得关注，在[25] 中提出了一个使用层间加入多个微管道，管道中流通液态冷却剂的方法来将芯片内部的热量导到芯片外部的模型，如图 2.7所示。这将是未来三维芯片发展所需的重要散热技术，然而针对这种结构的热分析的研究还不多，在本节的接下来的内容中将介绍如何针对这种结构进行热建模。

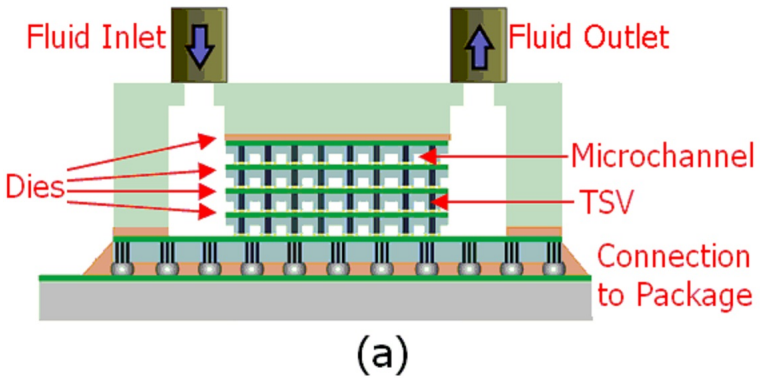
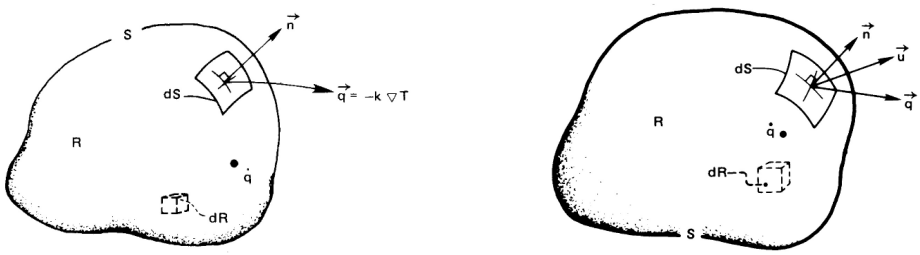



图 2.7　增加了层间液态冷却管道的三维堆叠芯片示意图[25]

### 2.3.2　使用液态冷却的三维芯片的热建模

为了得到固体单位元的热传导模型[27,32,33]，通常首先使用有限差分法对固体仿真区域进行离散化，对于离散化后的每个固体单位元，其热传导模型如图 2.8(a)所示。




(a) 固体单位元　　(b) 液体单位元

图 2.8　固体和液体单位元的热流方程建模示意图[32]

而对于液体仿真区域进行离散化后得到的单位元的热传导模型则如图 2.8(b)所示，可以看出，在液体的导热方程多了对流导热项，流体的热流公式如下：

$$\frac{d}{dt}\int_R \rho\hat{u}dx = -\int_S (\rho\hat{h})\vec{u}\cdot\vec{n}dS - \int_S (-k\nabla T)\cdot\vec{n}dS + \int_R \dot{q}dR \tag{2-16}$$

其中$R$是控制体积，$S$是其表面，$\vec{n}$是表面的外法方向向量，$\rho$是材料密度，$\hat{u}$是内部能量，$\hat{h}$是对流系数，$\vec{u}$是流体流动速度，$k$ 是材料的热导系数，$T$是温度，$\dot{q}$是体积内部产热速度。对 (2-16)使用高斯定理，并根据$du = c_p dT$，可以得到

$$\rho c_p \frac{\partial T}{\partial t} = -\rho c_p \vec{u}\cdot\nabla T + k\nabla^2 T + \dot{q}, \tag{2-17}$$

其中 $c_p$是材料的容积比热。

对方程 (2-17) 使用有限差分法进行离散化，并假设液体的流动方向是$x$轴方向，可以得到

$$\begin{aligned}\rho c_p \frac{T}{t} = {} & \frac{k_{xx}}{\Delta x^2}(T_{i+1,j,k} - 2T_{i,j,k} + T_{i-1,j,k}) + \frac{k_{yy}}{\Delta y^2}(T_{i,j+1,k} - 2T_{i,j,k} + T_{i,j-1,k}) \\ & + \frac{k_{zz}}{\Delta z^2}(T_{i,j,k+1} - 2T_{i,j,k} + T_{i,j,k-1}) + u_{xx}\frac{\rho c_p}{2\Delta x}(T_{i+1,j,k} - T_{i-1,j,k}) + g(\vec{v},t)\end{aligned} \tag{2-18}$$

其中$T_{i,j,k}$是在离散网格$(i,j,k)$上的温度，$k_{xx}$，$k_{yy}$和$k_{zz}$分别是沿着$x$，$y$，$z$方向的热传导系数，$g(\vec{v},t)$ 是体积$\Delta v = \Delta x\Delta y\Delta z$内产生的热量。可以将方程 (2-18)写成如方程 (2-19)所示的普通的偏微分方程组格式

$$\boldsymbol{G}\boldsymbol{T}(t) + \boldsymbol{C}\frac{d\boldsymbol{T}(t)}{dt} = \boldsymbol{B}\boldsymbol{U}(t) \tag{2-19}$$

其中$\boldsymbol{G}$与 $\boldsymbol{C}$分别代表热导及热电容的系数矩阵，$\boldsymbol{B}$是热源的输入位置矩阵，$\boldsymbol{T}(t)$ 是芯片上各点温度的向量，$\boldsymbol{U}(t)$是热源的输入向量。

在液流导热微管道的管道壁处，其热导率为$g_s ide = h_{side}S$，其中$h_{side}$ 是管道壁的热导系数，$S$是管道的表面积，可以用其来计算管道中流体与管道壁的热量交换。

注意在冷却管道内部的液体单位元中，每个单位元的热传导模型有

$$I_c = \rho C_p u_{xx} \frac{T_{i+1,j,k} - T_{i-1,j,k}}{2\Delta x} \tag{2-20}$$

这一项，其代表的是沿着管道方向的对流热量交换。而在芯片的其他位置的固体单位元中，$u_{xx}$为零，因此，这导致了最后的$\boldsymbol{G}$矩阵不对称。

之前的对液态冷却电路的热仿真研究工作中，绝大部分简化了仿真的流程，例如在[27,28]中，使用了基于有限差分高层次的互连线路电导模型来模拟自产热电路的热传导模型，而这种方法只适用于传统的固态电路，并不能模拟带有对流导热的液态冷却技术的电路。此外在[29–31] 的研究工作中，只关心了芯片的达到热稳态时候的仿真，并没有给出芯片达到热稳态之前的瞬态过程，然而电路在到达稳态之前的温度变化同样值得关注。

### 2.3.3　快速的热仿真技术

正如第 2.3.2节中所介绍的内容，为了验证一个新的冷却技术是否能够有效地将热量转移到芯片外部，从而有效地降低芯片温度，需要使用数值等手段来对芯片进行仿真分析。通常地，对芯片进行热仿真需要求解热偏导方程，诸如有限差分法、有限元法及流体动力学等手段能够被用来求解热偏导方程。对于三维芯片对应热电路，其规模非常大，对其进行的基于数值分析方法的仿真的计算量十分巨大。为了能够提高热仿真的效率，需要充分发掘热仿真问题在今天的大规模多核心计算平台上的并行计算潜力。图形处理器平台是如今被研究得最多，使用最广泛的高性能众核计算平台之一。目前在主流的高端图形处理器，例如NVIDIA Tesla T10 上的峰值浮点计算能力约为1TFLOPS，而相比之下，主流的中央处理器，例如i5四核处理器只有80-100GFLOPS的浮点运算能力[34]。目前，图形处理器或基于图形处理器的集群能够很容易地达到万亿次计算机别的计算能力，这是过去只有超级计算机才能够具有的能力。图形处理器为加速计算很多科学或者工程中的问题提供了良好的平台。目前，图形处理器已经能够很好地支持很多稠密线性方程问题，例如，NVIDIA CUDA 中就带有名为CUBLAS[35] 基本线性代数子程序(Basic Linear Algebra Subprogram, BLAS)，能够有效地支持稠密线性方程组中的很多计算。然而，目前为止，图形处理器对稀疏矩阵的求解问题支持得并不是很好，因为稀疏矩阵具有复杂的数据结构，而图形处理器对数据局部性具有很高的要求，因此，尽管已经有一些朝着这个方向努力的研究工作了[36,37]，我们通常还是认为在现代的高性能图形处理器上支持稀疏矩阵的运算是一件困难的事情。而科学计算中很多的线性方程组都是稀疏的，例如使用有限差分法进行热仿真等问题。在[25]中，一个基于CPU的稀疏LU 分解算法被用来求解三维芯片热传

导模型所对应的线性系统。而由于前面的原因，在GPU平台上进行LU分解的效率并不理想，而在另一方面，例如广义最小残差法(Generalized Minimum Residual, GMRES)[23] 等依赖很多矩阵向量乘或者矩阵乘的的迭代的线性方程组求解方法被认为是比较适合图形处理器计算。当加以可以被并良好行化的预条件方法之后，对于大问题来说，在图形处理器上，GMRES的速度有可能会远远快于稀疏LU算法。

## 2.4　使用GPU加速的相关问题研究现状

GPU平台因为具有强大的计算能力以及高的内存带宽，并且已经被成功商品化，在大部分的个人电脑上都包含GPU。基于GPU的通用计算已经被成功地运用在了油气公司，计算金融公司以及类似的寻求降低成本计算的公司和机构之中。EDA研究中有很多问题具有非常大的计算复杂度，尽管很多近似算法已经被采用来减少计算量，但是现在的处理器在计算的时候还是会消耗很长时间。最近一段时间的EDA领域，有一些研究着手于使用GPU来加速EDA中的具有高计算量的问题，并取得了不错的效果。这些工作包括Verilog 模拟、信号完整性与电磁学、计算光刻以及 SPICE 电路模拟等等。例如，在[38]中首次使用GPU对电容提取算法进行了加速，该研究的核心贡献是将多个小的不规则的系数矩阵平组合成大的规则的矩阵，从而优化了GPU的负载平衡，增大了内存带宽，该研究表明，使用GPU加速后的FMM (Fast Multiple Pole) 算法在对单介质中的简单的交叉导线进行电容提取的时候，其速度是CPU 上相同算法的22到30 倍。在[39]的工作中研究了使用GPU加速的EDA 中常见的不规则的稀疏矩阵，图算法的计算，他们实现了GPU加速的稀疏矩阵向量乘操作，并使用该GPU加速的稀疏矩阵向量乘操作加速了用稀疏矩阵表示的广度优先搜索算法，相比起CPU平台，能够获得10倍的加速比，结果显示了基于GPU的计算在EDA 领域中的巨大潜力。在[40]的工作中，他们提出将基于Verilog的RTL描述转化为GPU对应代码的算法，通过使用GPU加速了集成电路设计中的RTL 级别的仿真流程，结果显示基于GPU的仿真流程的速度能够达到商业串行代码的20倍。在[36,37]的工作中，第一次使用GPU 加速了稀疏矩阵的LU分解过程，他们使用基于Eschedular的数据依赖分析方法，提出了聚类及管道两种在不同情况下使用的并行策略，使用GPU加速了left-looking 的LU 分解算法，他们的算法能够比单核CPU程序快7.90倍，比8核CPU程序快1.49 倍。此外诸如此类的在GPU上开展的求解EDA领域中问题的研究还有很多，这都显示了GPU平台已经吸引了越来越多的EDA领域的注意力。

最近也有一些在图形处理器上进行芯片热仿真相关工作，然而这些工作都有一定的局限性。例如在[41]中提出了一个多重网格求解器，并将该求解器用到求解供电网络的直流分析问题之中。在[42]中提出了一个用多重网格算法做预条件的共轭梯度求解器，可以用来对电源分布网络做直流分析，该工作相同的研究组同样使用此多重网格预条件的共轭梯度求解器求解了热仿真问题，在他们的算法中，使用的对称阻抗网络[43]，他们的工作能够从使用的网络的对称性中受益，因为对称网络通常允许一定程度上的优化。然而对于[25]中提出的使用液态冷却技术的三维电路，并不能够将其化为对称网络。此外目前同样已经有一些关于在图形处理器上计算GMRES算法的研究工作，例如在[44] 中，实现了基于GPU的使用不完全LU 分解作为预条件的GMRES算法，然而，该工作只是将一小部分工作在图形处理器上做并行，有很多工作任然需要串行地进行。在[45] 中对图形处理器并行的GMRES 算法进行了更加透彻的研究，但是该工作中使用的稀疏矩阵的数据结构的效率不高[46]，而且该工作中并没有指明计算时候的精度设置，使得无法将其工作与其他人的工作进行比较。最新的关于图形处理器上的GMRES 算法的研究来自[47]，该工作将图形处理器上的GMRES算法的性能与他们自己实现的在中央处理器上的GMRES算法的性能做对比，而不是与公开的求解器的性能进行对比，同样使得其他人无法准确衡量他们的工作的实际效率。

# 第 3 章　面向互连电容提取的GPU并行算法

本章将介绍使用GPU加速的用于随机行走电容提取算法。本章首先介绍了使用GPU加速该算法的困难所在，然后提出了一个三阶段的迭代的算法流程来减小指令流分歧，紧接着介绍了生成额外起始点来优化负载均衡的手段，然后介绍了用于减小内存访问次数的ICPA数据结构，并在多介质的时候通过提出一种新型采样策略将ICPA使用到了多介质转移函数之中。本章的最后针对一些不同的电容结构进行了提取，并根据45nm的工艺生成了一些具有代表性的互连结构，通过与CPU程序性能的对比证明了我们提出的算法的正确性与效率。

## 3.1　GPU并行随机行走算法的困难

随机行走算法的不同行走之间是完全独立的，在使用CPU对其做并行化的时候，能够获得非常好的效果。例如在[17]中，使用八核CPU的时候，能够将程序加速7倍以上。然而，当简单地使用GPU对其进行加速的时候，算法的效率往往不高。主要原因有以下一些：首先，由于随机行走算法的随机性，不同的行走所需的跳转的次数，即$N_{hop}$不一样，这种差异造成了不同线程之间严重的负载不均衡；其次，随机行走算法中有很多的条件分支语句，这些条件分支的执行取决于运行时生成的随机数，受限于图形处理器的SIMT并行模式，条件分支语句会严重妨害并行算法的性能；再次，图形处理器的共享内存的容量太小，使得随机行走算法中的格林函数表等数据无法装入共享内存，只能转而使用全局内存存储，加上随机行走算法对内存存取的随机性，使得数据局部性差，程序效率极其低下。本章的余下部分将分别解决上述问题。

## 3.2　针对单介质互连结构的技术

本节介绍GPU加速单介质环境中的电容提取随机行走算法的技术。这些技术不仅可以用在电容提取的随机算法中，也可推广到其他应用中的随机行走算法在GPU上的加速。

### 3.2.1　三阶段迭代算法流程

对每个电容值进行提取中都需要进行成千上万次独立的随机行走，因此，一种最直接的并行策略就是使GPU的每一个线程单独负责一次行走，最后再统计这

些行走获得的结果并更新电容值。由于GPU的SIMT的并行方式以及最小以warp为调度单位的策略，这样的思想在GPU 上执行的方式是：在程序开始的时候，每个warp中的所有线程统一开始一个随机行走，等该warp上所有的线程都执行完当前行走，并将它们行走所得到的结果对电容值进行更新，该warp中所有的线程再统一开始一组新的随机行走。这样做的一个显而易见的缺点是由于随机算法的不确定性，不同的线程所负责的行走所需的跳转步数有差异，那么一个warp 中所有的线程都要等待warp中行走行走步数最长的线程结束行走之后，才能一起开始下一次并行地行走。在实际使用中，短的行走在跳转几步就能够击中导体并停止，而长的行走有时候需要跳转上百次才能停止，这种行走步长的巨大差异将会造成大量计算资源的浪费。

随机行走算法也可以用在图形学领域中对场景进行仿真渲染。这里的随机行走算法中，接收器首先发射出一系列的光线，光线沿着直线运动，当碰到障碍物将会发生发射，直到最后光线反射回到接收器，记录下光线所碰到的障碍物，一次随机行走过程结束，最后统计所有光线的结果可得到场景中的障碍物分布信息。使用GPU并行该随机行走算法有着与并行电容求解算法中的随机行走算法相同的问题，在[48]中提出了一种行走重启动算法(Walk Regeneration Algorithm)：当一个线程结束了一次行走之后，该线程直接重新从发射器生成一条新的光线，开始一次新的随机行走算法，而不是等待其他的线程结束行走。我们尝试过将该行走重启动算法移植到对电容提取的随机行走算法之中来，但是实验结果并不理想。分析原因，是由于在随机行走电容提取算法中，每次一个线程要重新生成一个新的行走，它所需进行的操作包含：从高斯面上取高斯点$r^{(0)}$，围绕高斯点$r^{(0)}$做转移区域$S^{(0)}$，从转移区域表面$S^{(0)}$ 上取点$r^{(1)}$，并计算得到该次行走的权值等一系列的操作。当一个线程执行这些指令生成一个新的行走的时候，由于SIMT并行方式的限制，其它所有正在进行行走的线程都需要停下来等待。由于生成新行走的操作需要花费不菲的时间，因此，行走重启动算法引入了新的指令流分歧，使得效率并不理想。之所以该方法在图形学中的随机行走算法能够得到成功，一个原因是生成一条新的光线的操作很少，使得新引入的指令流分歧所占的比重很小。

为了减少指令流分歧造成的计算资源的浪费，本文使用了算法流程细分的策略。本文提出将随机行走的过程分成三个子模块，每个子模块所负责的任务如下：

- **开始模块**：从高斯面上取高斯点$r^{(0)}$，围绕高斯点$r^{(0)}$做转移区域$S^{(0)}$，从转移区域表面$S^{(0)}$ 上取点$r^{(1)}$，并计算得到该次行走的权值。将点$r^{(1)}$的坐标及该次行走的权值记录到全局内存之中。

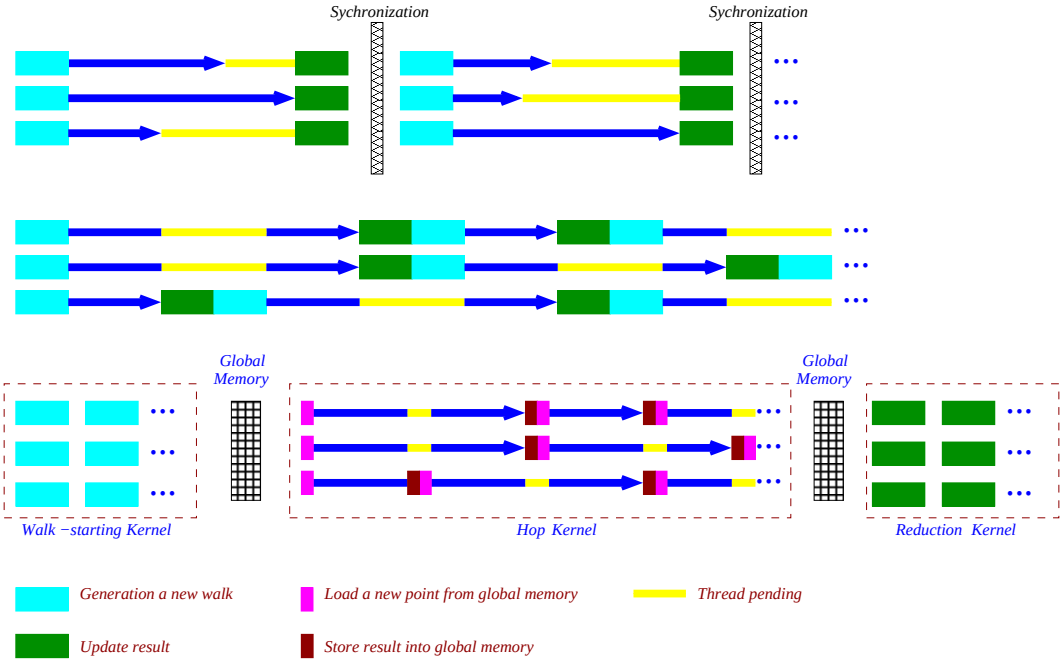



图 3.1　三种不同的随机行走并行策略在GPU上运行的示意图，从上到下依次为直观并行策略、行走重启动策略及三阶段策略。注意每个操作在图示中的长度并不代表其实际运行时间

- **跳转模块：**读取开始模块所生成并存储在全局内存中的点$r^{(1)}$的坐标，使用行走重启动算法并行地进行随机行走接下来的跳转操作，直到点击中导体表面为止。并将该行走所击中的导体的编号记录到全局内存之中。
- **规约模块：**读取前两个模块所生成的每个行走的每个行走的权值及所集中的导体编号，更新当前的电容值及所达到的误差。

通过了将原来完整的随机算法分为三个小模块，每个模块通过全局内存交换数据，使得每个模块所进行的操作的种类得到了简化，出现指令流分歧的可能性大大降低，从而提高了效率。例如在开始模块，每个线程执行的指令都完全相同，这个模块不存在指令分歧；在跳转模块，当一个线程执行完了一次行走并重新生成一个新的行走的时候，它只需要从全局内存之中读取下一个新点$r^{(1)}$的坐标，相比起未处理前的算法，这个时间大大减少了；对于规约模块，目前已经有很多基于GPU实现的高效的规约算法，例如[49]中的*reduction* 算法可以稍加修改就可以高效地利用在电容求解的随机行走算法中。图 3.1显示了三种不同的并行策略在GPU上运行时的示意图。

电容求解的随机行走算法的一个重要的优点就是它的精度是可以控制的。然而在我们上面所提出的算法中，电容的结果和误差值只是在规约模块进行完了才能够得到，在开始模块，我们并不知道总共需要生成多少个点$r^{(1)}$。根据随机行走算法的理论分析可以得出其结果的误差值的平方与行走的步数呈反比例，根据这

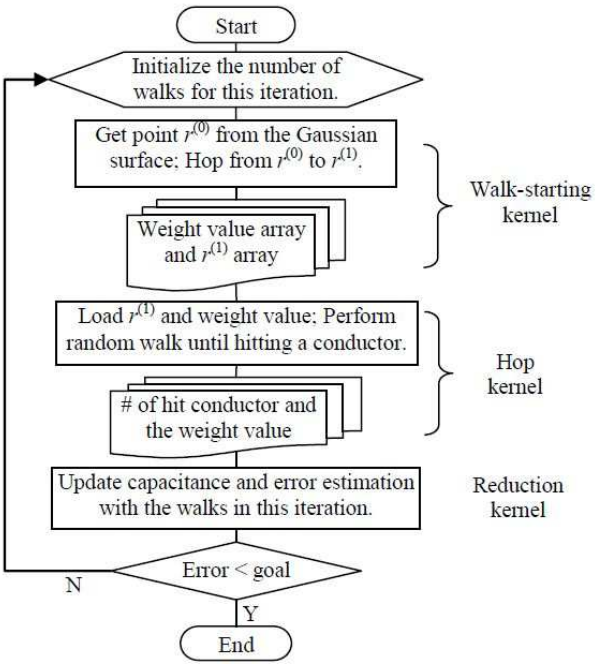



图 3.2　三阶段的迭代求精的算法流程

一关系，本文提出一种迭代的算法流程。假设$n_{cur}$是目前为止累计进行了的行走次数，$err_{cur}$是目前结果的误差值，$err_{goal}$是结果的目标误差值，$n_{rem}$是结果达到目标误差值的时候还需要进行的行走的步数，有以下公式：

$$n_{rem} = n_{cur} \times (\frac{err_{cur}^2}{err_{goal}^2} - 1) \tag{3-1}$$

本文提出的算法在每次规约模块结束的时候，根据公式 (3-1)计算出剩余所需的行走步数，然后从开始模块开始，重新进行下一轮迭代增加精度的过程。在实际计算时，为了使得迭代的次数尽量少，一般可以稍微增大$n_{rem}$的值，使得所有的行走一般在两次迭代就能够达到目标精度。

图 3.2显示了使迭代的三阶段的随机行走算法的流程。我们可以观察到在上述算法之中，各个模块之间的数据交换都是在GPU 端进行的，需要拷贝到CPU的数据仅仅是规约模块的得到的几个浮点数，因此，CPU和GPU 之间的数据通信时间几乎可以忽略。论文相关工作中实现了直观并行策略，行走重启动策略以及本节提出的三阶段策略，相关的数值试验结果在第 **??**节所示。

### 3.2.2　内存管理及额外开始点

前一节将随机行走算法分成三个模块，不同模块通过GPU的全局内存交换数据。为了避免模块内部多线程访问数据时候所需要进行的内存加锁等妨害效率的操作，需要保证每个线程所能访问的内存的地址不一样，即在每个模块开始的

时候给每一个线程分配独享的内存空间。对于行走模块，假设$Walk_{total}$是内存中点$r^{(1)}$的总数目，$Num_{thread}$是总线程个数，$Walk_{thread}$是每个线程所分配的需要行走的步数，由于在开始的时候，我们并不能知道那个线程计算的速度更快，我们只能为每个线程分配等量的任务，即

$$Walk_{thread} = \frac{Walk_{total}}{Num_{thread}} \tag{3-2}$$

在行走模块，由于不同线程速度有差异，当有的线程进行完$Walk_{thread}$次行走时，还需要等待其他所有线程执行完才能一起开始规约模块，这造成了一定的计算资源的浪费。在我们的实验中可以观察得到，在整个算法所消耗的时间中，跳转模块所花费的时间通常占到了90%以上，为了优化整个程序的计算时间，我们提出了为每个跳转模块中的线程生成额外点$r^{(1)}$的算法，使得在跳转模块中，当一个线程完成了预先分配给它的任务，它可以继续代替慢的线程做一些行走，当所有的线程完成了原先预定的$Walk_{total}$那么多个行走，不同的线程所做的行走个数有可能不同，但是保证了它们说消耗的时间是相似的。定义一个大于1的冗余系数$r_{rddt}$，在开始模块为每个跳转模块中的线程所生成的可用的点$r^{(1)}$的个数为

$$Point_{thread} = r_{rddt} \times \frac{Walk_{total}}{Num_{thread}} \tag{3-3}$$

三个模块各自的运行时间与$r_{rddt}$之间的关系如图 3.3所示。可以看出，跳转模块占据了绝大部分的程序时间，引入$r_{rddt}$能使得程序计算总时间减少40%左右，当$r_{rddt}$的值约为1.5的时候，程序性能达到最佳，此后继续增大$r_{rddt}$，跳转模块时间减少不明显，而另外两个模块增加的时间导致总时间增大。在我们的实验中，我们设置$r_{rddt} = 2$，这个取值对于绝大部分实验的结果都接近最优。

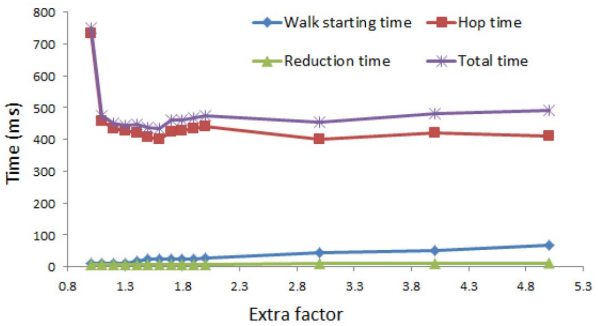



图 3.3　各模块计算时间与$r_{rddt}$之间的关系

为随机行走的跳转模块增加额外开始点之后的示意图见图 3.4所示，其中*Array*1用来存储开始模块生成的点$r^{(1)}$的坐标，*Array*2用来存储跳转模块计算得到的每次行走所击中的导体的编号。

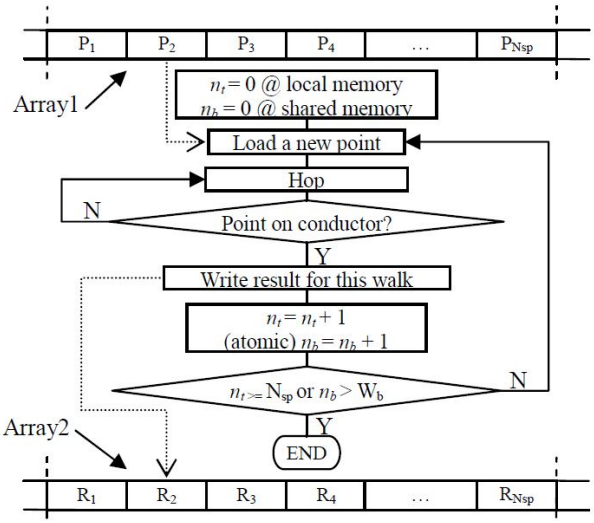



图 3.4　增加了而外开始点后的随机行走跳转模块算法示意图，$n_t$表示每个线程已经完成的行走的数量，$n_b$表示一个线程块已经完成的行走数量总和

### 3.2.3　使用反向累加概率向量对行走进行加速

随机行走算法将格林函数表预先存在全局内存之中，在每一次跳转的时候，都需要访问格林函数表得到下一次跳转到达点的坐标。由于格林函数表存储在全局内存之中，对它的频繁访问将会消耗大量的时间，实验表明，在经过算法细分模块减少指令分歧之后，对全局内存访问所带来的时间开销已经成为了整个程序性能的瓶颈。

立方体转移区域的格林函数是一种概率分布函数(Probability Distribution Function, PDF)，它存储的是从转移区域中心点跳转到转移区域表面离散化的每个面元的概率。假设转移区域表面总共有$N$个面元，跳转到每个面元的概率分别为$p_i, i = 1, \ldots, N$。通常，为了方便使用，我们需要将概率分布函数转化为累计分布函数(Cumulative Distribution Function, CDF)：

$$S_i = \sum_{j=1}^{i} p_j, \qquad i = 1, \cdots, N \tag{3-4}$$

在得到累积分布函数之后，按照均匀分布生成一个随机数$r_{rand} \in [0, 1]$，在$\{s_i\}$上二分查找$r_{rand}$就可以得到所需跳转到的面元的编号。这样，每进行一次跳转，都需要进行二分查找操作，多次访问全局内存；此外，由于每个线程进行二分查找所需要的访存次数不一样，根据GPU的SIMT并行模式，整个warp之中所有的线程都需要等待最慢的一个线程做完二分查找，才能进行接下来的操作，这也同时也造成了计算资源的浪费。

为了解决二分查找对效率造成的损害，我们引入以一种新的数据结构**反向累积概率向量**(Inverse Cumulative Probabilities Array, ICPA)，它的作用是将每个概率

值与所对应的跳转面元建立一一映射的关系。我们首先定义一个分辨率

**定义** 3.1：　$\lambda = \min_{1\le i\le N}\{p_i\}$

ICPA是一个含有$1/\lambda$个元素的整数数组，它的每个元素值的定义如下

**定义** 3.2：　$t_i = \lfloor p_i/\lambda + 0.5\rfloor, \qquad i = 1,\cdots,N$

**定义** 3.3：　$ICPA[\sum_{i=0}^{k-1} t_i + 1 : \sum_{i=0}^{k-1} t_i + t_k] = k, \qquad k = 1,2,\cdots,N$

根据ICPA的定义，当需要按照概率$r_{rand}$做跳转，$ICPA[r_{rand}/\lambda + 0.5]$中存储的就是所需跳转到的面元的编号。有了ICPA之后，每一次跳转只需要访问一次全局内存，与二分查找的策略相比大大减少了内存访问的时间。对于一个一般尺度的转移区域表面离散化，例如表面含有5000个面元，其单介质格林函数的ICPA 所需要的额外存储量约为$10MB$左右，这个存储量对于今天的GPU是可以承受的。

## 3.3　针对多介质互连结构的技术

本节针对单介质问题的技术的前提上，针对实际工艺中的多介质环境下的电容提取遇到的问题及相应的优化技术进行讨论。

### 3.3.1　多介质问题所面临的挑战

今天的VLSI制造工艺中，导线被分层的，不同介电常数的介质所包围。通常使用数值方法得到多介质转移区域的格林函数和权值函数[17]，这样算法就可以跨越多层介质行走。当使用GPU加速多介质的随机行走算法时，将会面临下面一些主要的挑战：

1. 在第 3.2.3节中所提出的ICPA数据结构不能对多介质的格林函数使用，因为多介质有很多格林函数表，都要构造对应ICPA 的话将会带来无法承受的内存需求。因此，当转移区域跨越多层介质的时候，我们必须重新使用效率低下的二分查找。
2. 多介质问题中的每次行走的跳转步数远大于单介质问题，这将带来更多的GPU全局内存访问次数。

### 3.3.2　使用变种采样策略的随机行走算法

为了减少全局内存的访问时间，我们将公式 (2-4)修改如下：

$$Q_j = \oint_{G_j} F(r)g \oint_{S^{(1)}} \omega(r,r^{(1)})\frac{G^{(1)}(r,r^{(1)})}{G_{(0)}(r,r^{(1)})}G_{(0)}(r,r^{(1)})\phi(r^{(1)})d(r^{(1)})dr \qquad (3\text{-}5)$$

其中$G^{(1)}(r,r^{(1)})$表示多层介质的格林函数，而$G_{(0)}(r,r^{(1)})$表示单介质的格林函数。我们进一步地定义

$$\omega'(r,r^{(1)})=\frac{G^{(1)}(r,r^{(1)})}{G_{(0)}(r,r^{(1)})} \tag{3-6}$$

将式 (3-6)代入式 (3-5)可得到

$$Q_j=\oint_{G_j}F(r)g\oint_{S^{(1)}}\omega(r,r^{(1)})\omega'(r,r^{(1)})G_{(0)}(r,r^{(1)})\phi(r^{(1)})d(r^{(1)})dr \tag{3-7}$$

我们将公式 (3-7)所示的采样策略称为一种变种的随机行走算法，而对应地将公式 (2-4)所示的算法成为标准随机行走算法。变种的随机行走算法在转移区域跨越多层介质的时候，使用单介质的格林函数$G_{(0)}(r,r^{(1)})$ 进行行走，并将结果乘以一个权值$\omega'(r,r^{(1)})$。因为只用单介质格林函数被使用，第 3.2.3节所提出的ICPA数据结构可以被使用来对其进行加速。在变种的采样策略中，多介质格林函数没有被使用到，而其值与$\omega'(r,r^{(1)})$ 的值是一一对应的，故可以将$\omega'(r,r^{(1)})$ 用原来存储多介质格林函数的空间进行存储，因此变种的随机行走算法没有引入任何而外的内存开销。

变种随机行走方法的一个缺点是它将是不同行走的结果的变动增大，导致最后结果达到相同精度时所需要的行走数目变大。然而，由于其可使用ICPA所带来的加速效果远远超过了它的不足，有关于变动随机行走方法的数值试验结果参见第 3.8节。

### 3.3.3　多导线并发提取

对于实际的电容提取问题，在一个版图之中，我们往往需要提取超过一条的导线上的电容。因为对于同一个结构，格林函数表，权值表和空间管理等数据都是相同的，原来的针对单条主导体的随机行走算法可以很容易地扩展到处理多条主导体的问题，而不需要而外加载数据。

在第 3.2.1节中提出的将算法分为三个子模块的策略能够极大程度地降低算法不同线程之间的指令分歧。本算法主要的时间瓶颈变为全局内存访问的延时。CUDA可以通过不同warp的切换来掩盖内存访问延时，即，一个流多处理器通过分时的方式同时负责很多个warp 的计算，但在某一个时刻这个流处理器正在计算的warp中有线程的数据没有准备好，这个流多处理器能够将这个warp切换成等待状态，而将其它的已经准备好数据的warp加载进来进行计算。当被切换出去的warp的数据准备好了，例如从全局内存请求的数据返回了，再对其进行计算[50]。这种用计算来掩盖指令延时的方法能够发挥效果的前提是提供足够多

的warp块供流多处理器进行切换，而程序可提供的warp数量与所需要进行的计算量有关。实验结果显示，当版图之中只有一个主导体的时候，程序并不能提供足够多的warp数量给流处理器进行切换掩盖内存延时。由于同一个版图时对不同导体进行提取所需要的数据几乎一样，我们可以在同一时间让GPU对若干条导线同时进行提取，来提供足够多的计算量来掩盖延时。

当同时提取多条导线电容的时候，每一条单独导体的提取都依照第 3.2.1节所提出的迭代流程进行计算，在规约模块结束后，每个导线都按照公式 (3-1)计算该导线所需要的到达收敛精度所剩余的行走步数，然后将计算资源按比例分配来对每个导线进行计算。关于多导体并发提取加速技术的实验结果见第 **??**节所示。

## 3.4　实验结果和分析

本节以CUDA 4.0实现了GPU版本的电容随机行走算法，作为对比的CPU版本的算法以C++实现。多层介质的格林函数表及权值表按照论文[17] 中的方法实现。所有的实验都在一台拥有2.70GHz主频双核奔腾CPU，6GB内存，以及拥有1.5GB的全局内存的Nvidia GTX580 图形处理器上运行。

### 3.4.1　对简单结构的提取

在本节中涉及到两个简单的电容结构。第一个是一个在空间中的孤立的立方体，如图 3.5(a)所示。不同于空间中的孤立球体，没有解析的手段可以用来得到孤立立方体的电容值。之前有一些工作使用数值手段来获得孤立立方体的电容值，如[51–54] 等工作。使用随机行走算法计算孤立立方体电容值的时候，需要设定一个电容值为0*volt*的边界，这样的近似往往会造成计算出来的电容值略微偏大，在本节的实验中，设置电容为0的边界离立方体的距离是立方体边长的1000倍，这样使结果的误差足够小。随机行走算法的终止条件设置为$1-\sigma$误差小于0.1%，这意味着结果落在真值的0.3%误差范围内的置信度为99.7%。表 3.1中显示了使用随机行走算法获得的结果，表 3.1中同时也给出了使用[51–54]中提出的算法得到的结果。可以看到，尽管随机行走算法的结果相比起[52–54]中的结果来说略微偏大，但是都在可以接受的范围内，而且这种偏差是由于边界的设置引起的。

为了验证本文所提出的的在GPU平台上并行随机行走算法的效率，表 3.2给出了在CPU和GPU上的随机行走算法的运行时间对比。结果显示了CPU与GPU上的算法结果吻合，并且尽管在GPU上的随机行走算法的行走步数会多于CPU上的随机行走算法，但是其能够带来达到63倍左右。

表 3.1　使用不同方法计算得到的空间孤立立方体的电容值，电容值单位为$4\pi\varepsilon$

| 使用方法 | 电容结果 |
|---|---|
| Mascagni-Simonov, Random walk on the boundary[53] | $0.6606780 \pm 2.7 \times 10^{-7}$ |
| Hwang-Mascagni-Won, Random walk on the boundary[54] | $0.66067813 \pm 1.01 \times 10^{-7}$ |
| Bontzios-Dimopoulos-Hatzopoulos, Evolutionary method[51] | [0.6738, 0.6820] |
| Lazic-Stefancic-Abraham, The Robin Hood method[52] | $0.66067786 \pm 8 \times 10^{-8}$ |
| 随机行走算法(CPU) | $0.6619 \pm 0.0020$ |

表 3.2　使用CPU与GPU上的随机行走算法计算得到的空间孤立立方体的电容值及它们的运行时间对比，电容值单位为$4\pi\varepsilon$

| 使用方法 | 行走步数 | 电容结果 | 运行时间 | 加速比 |
|---|---|---|---|---|
| CPU上的随机行走算法 | $1.44 \times 10^7$ | $0.6619 \pm 0.0020$ | 78.1 | – |
| GPU上的随机行走算法 | $2.47 \times 10^7$ | $0.6627 \pm 0.0020$ | 1.24 | **62.8** |

本节的第二个电容结构是平行板电容结构，其电容值有解析计算公式。平板的大小是1000$\mu$m×1000$\mu$m，平板的距离为10$\mu$m，如图 3.5(b)所示。两个平行板之间的耦合电容值以及图中点A、B处的沿着Z轴方向的电场强度将被计算，A、B点的坐标分别为(500, 500, 5)与(500, 500, 8)。在表 3.3中给出了电容值与A、B点处的电场强度值。可以看出随机行走算法的结果与解析得到的结果十分相近，随机行走算法得到的电容值为$8.854 \times 10^{-13}$F，比解析值大3%左右，这是由于设置的边界的原因。

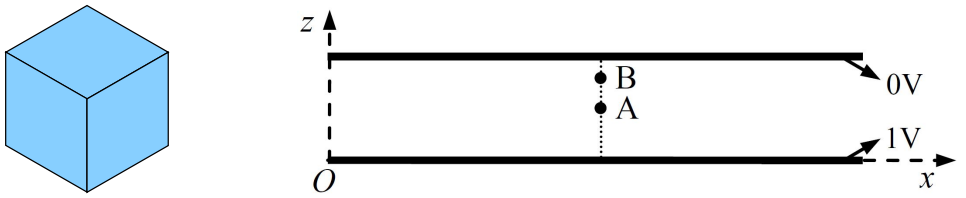
表 3.3　使用随机行走算法及解析公式计算得到的平行板电容结构的数值结果

| 方法 | 电容结果($10^{-13}$F) | A处的电场强度($10^5$V/m) | B处的电场强度($10^5$V/m) |
|---|---|---|---|
| 解析公式 | 8.854 | 1 | 1 |
| 随机行走 | $9.12 \times 10^{-2}$ | $1.000 \pm 0.003$ | $1.000 \pm 0.003$ |

使用CPU及GPU上的随机行走算法分别使用的行走步数及消耗的时间在表 3.4中给出。可以看出，在使用了GPU平台进行加速之后，对平板电容值得计算速度能够达到12倍以上，而对电场强度的计算速度能最高可到到36倍以上。

### 3.4.2　对复杂结构的提取结果

在本节将考虑对两个实际复杂结构进行电容提取的过程。第一个结构是一个从实际VLSI版图中截取出来的一部分，如图 3.6所示，图中的金属导线分布在五个布线层上，图中画出了包围着主导体的高斯面，需要提取的是主导体与其他所有环境导体的耦合电容。另一个结构是一个微电子机械系统(Micro-electromechanical Systems, MEMS)，其被广泛应用在微加速传感器，硬盘驱动器




(a) 空间中孤立立方体示意图　(b) 平行板电容结构的示意图，A点与B点处的电势将会被计算

图 3.5　空间中孤立立方体及平行板电容的结构示意图

表 3.4　使用CPU及GPU上的随机行走算法计算平行板电容值及电场强度的运行时间对比

| | 方法 | 行走步数 | 时间(s) | 加速比 |
|---|---|---|---|---|
| 耦合电容 | CPU随机行走 | $1.86 \times 10^6$ | 2.38 | – |
| | GPU随机行走 | $7.65 \times 10^6$ | 0.189 | **12.6** |
| A点场强 | CPU随机行走 | $7.07 \times 10^5$ | 0.67 | – |
| | GPU随机行走 | $1.72 \times 10^6$ | 0.038 | **17.6** |
| B点场强 | CPU随机行走 | $5.83 \times 10^6$ | 7.39 | – |
| | GPU随机行走 | $9.04 \times 10^6$ | 0.201 | **36.8** |

等设备中，如图 3.7所示，图中的MEMS有着两个由很多复杂矩形结构组成的电极，提取的为两个电极之间的耦合电容。

表 3.5中对比了使用随机行走及基于边界元方法的QBEM求解器提取电容的结果，QBEM电容求解器[24]使用虚拟多介质(Quasi-Multiple Medium, QMM) 方法加速，其对VLSI互联结构的电容提取的速度速度可达到快速多级(Fast Multi-pole Method)方法的10倍。从表 3.5中可以看出，使用随机行走算法与边界元算法获得的电容值结果的差异在2%以内。同时可以看出，基于GPU的随机行走算法比CPU上的随机行走算法快100倍左右，比在CPU上的QBEM 快200倍以上。相

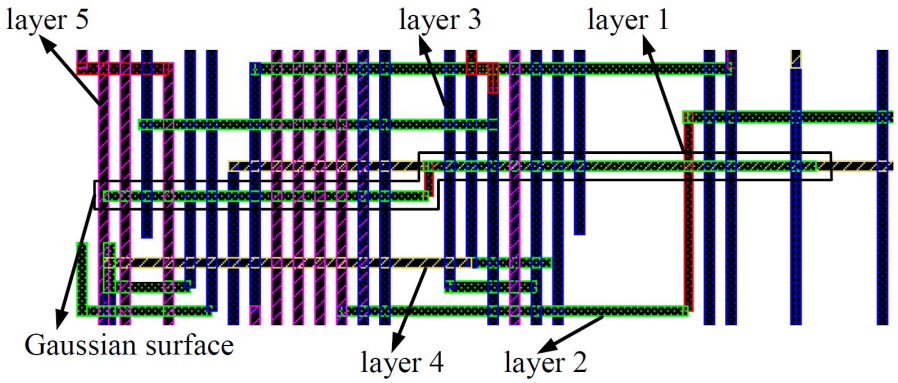



图 3.6　一个从实际VLSI版图中截取结构示意图，其中包含5个不同的金属布线层，主导体用高斯面包围

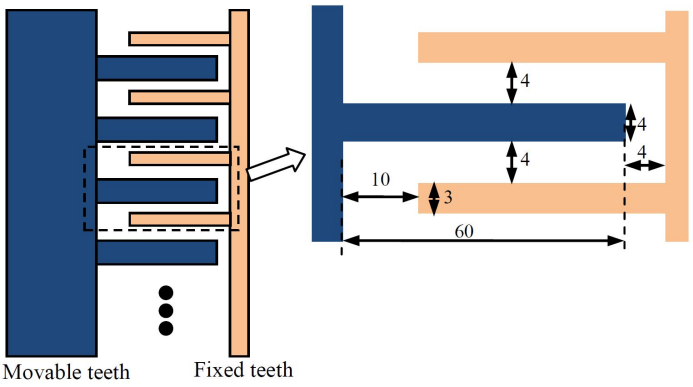



图 3.7　一个MEMS的结构示意图，其包含两个由多个矩形结构组成的复杂电极(图中长度单位为微米)

同CPU 上的随机行走算法比边界元算法快一个数量级，证明了对较大结构的随机行走算法的良好的扩展性。

表 3.5　对两个复杂结构的电容提取的结果

| Case | FRW | FRW (CPU) | | FRW (GPU) | | | | QBEM | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Cap. | #walk | Time(s) | #walk | Time(s) | Sp.[1] | Sp.[2] | #panel | Cap. | Time(s) |
| VLSI | 8.01 | 289K | 1.20 | 545K | 0.054 | **22.2** | **694** | 35,335 | 7.93 | 37.5 |
| MEMS | 371 | 248K | 0.74 | 504K | 0.023 | **32.2** | **287** | 14,382 | 366 | 6.6 |

Sp[1]：相对CPU上的随机行走算法的加速比
Sp[2]：相对CPU上的QBEM程序的加速比

### 3.4.3　对45nm工艺下的互连结构进行电容提取

本节实验的测试用例基于45nm工艺，该工艺的具体参数等细节来自于[55]，我们所生成的例子的的布线位于工艺的最下面三个导线层，与之相关的一些工艺细节如图 3.8所示。

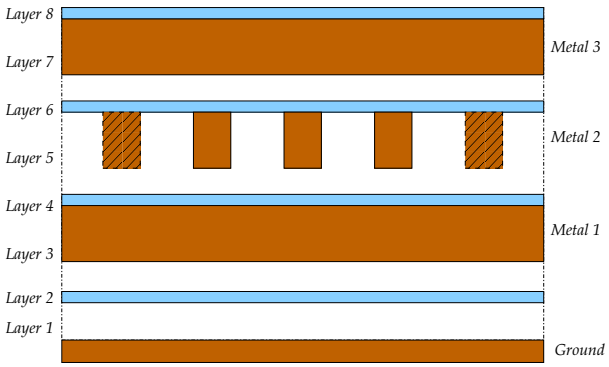



图 3.8　一个45*nm*工艺横截面示意图，其中左侧标注代表介质层编号，右侧标注代表金属层编号。*Layer*2, *Layer*4, *Layer*6, *Layer*8为薄层，介电常数为5.0，其余介质层介电常数为2.6；*Metal*1, *Metal*2, *Metal*3层线宽70nm，线高140nm，布线间距70nm

基于图 3.8所示工艺构造了一些VLSI 电路中常见的具有代表性的互连结构，这些例子具体如下：

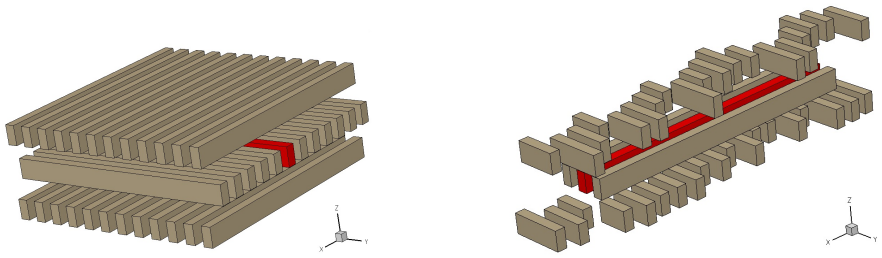
**样例一** 4x4交叉线结构，*Metal*1与*Metal*2布线层分别有4条走向垂直的导线。

**样例二** 6x6交叉线结构，*Metal*1与*Metal*2布线层分别有6条走向垂直的导线。

**样例三** 6x6x6交叉线结构，*Metal*1，*Metal*2与*Metal*3布线层分别有6条走向垂直的导线。

**样例四** 41导线结构，主导体位于*Metal*2层，水平方向两侧各有一条平行的环境导体，*Metal*1, *Metal*3层被一些随机生成的环境导体包围。

其中样例三和样例四的立体视图见图 3.9所示。



(a) 样例三　　(b) 样例四

图 3.9　样例三与样例四的立体视图，其中的红色矩形表示需要提取的主导体，而灰色矩形则表示环境导体

为了比较不同的并行策略对算法效率带来的影响，下面一部分内容实现并比较了了直观并行策略，行走重启动策略以及第 3.2.1节提出的三阶段策略的效率。为了避免第 3.3节中所提出的针对多介质的技术对算法性能的影响，我们首先测试了一些在单介质下的样例，与多介质的例子相比，所做的改动仅是简单地将介质层移除。

表 3.6　GPU上三种不同并行策略的效率比较($10^6$次行走；时间单位为*ms*；电容值单位为$10^{-18}F$)

| Case | CPU程序 | | 直观并行策略 | 行走重启动策略 | 三阶段策略 | | | |
|---|---|---|---|---|---|---|---|---|
| | 电容 | 时间 | 时间 | 时间 | 电容 | 时间 | Sp.[1] | Sp.[2] |
| 样例1 | 39.9 | 3520 | 286 | 607 | 40.1 | 47 | 74 | 6.1 |
| 样例2 | 59.3 | 4285 | 299 | 710 | 59.3 | 49 | 87 | 6.1 |
| 样例3 | 113 | 5347 | 322 | 895 | 113 | 59 | 91 | 5.5 |
| 样例4 | 133 | 7910 | 369 | 1288 | 129 | 63 | 129 | 5.9 |

Sp[1]：相对CPU程序的加速比

Sp[2]：相对直观并行策略的加速比

由表 3.6可以看出，[48]中提出的行走重启动算法对于电容提取的随机行走算法并不适用，使用了行走重启动方法的程序反而比未优化的程序速度更慢。这是因为重新启动一个新行走需要进行非常多的操作，而这些操作将会在算法中引入指令分歧，使得效率受到影响。在第 3.2.1节提出的三阶段算法则能大大提高算法效率，其相对于未做优化的GPU 算法有着6倍左右的加速，而相对CPU上的程序则能达到74倍以上的加速。

在这下面的内容中，我们将展示在第 3.3.2节中针对多介质问题所提出的变动采样策略的效果。所有测试样例的收敛精度设置为与主流商业软件相同，即0.5%的$1-\sigma$误差，其意味着结果有的误差小于1.5%的概率是99.7%。计算结果被列在表 3.7中。 从表 3.7可以看出，使用了我们提出的新的采样策略之后，基

表 3.7　对于多介质问题的基于CPU和GPU的算法的计算结果(收敛标准为0.5%的$1-\sigma$误差；电容的单位为$10^{-18}F$；时间的单位为$s$)

| Case | CPU程序 | | | 标准GPU算法 | | | 使用变动采样策略的GPU 算法 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 步数 | 电容 | 时间 | 步数 | 电容 | 时间 | 步数 | 电容 | 时间 | Sp.[1] | Sp.[2] |
| 样例1 | 606K | 117 | 25.9 | 1088K | 117 | 1.21 | 1800K | 119 | 0.49 | 53 | 2.5 |
| 样例2 | 507K | 176 | 12.5 | 1015K | 174 | 0.69 | 1613 | 177 | 0.31 | 40 | 2.2 |
| 样例3 | 483K | 334 | 10.5 | 909K | 336 | 1.16 | 1614K | 334 | 0.39 | 35 | 3.0 |
| 样例4 | 405K | 353 | 7.84 | 590K | 352 | 0.89 | 609K | 355 | 0.36 | 22 | 2.5 |

Sp[1]：相对CPU程序的加速比

Sp[2]：使用变动采样策略的算法相对于未使用变动采样策略的算法的加速比

于GPU的随机行走算法能够比之前快3 倍以上，并比CPU上相同的算法快22 倍以上。这是因为我们提出的新的采样策略使得ICPA 可以被使用，使得全局内存访问开销和指令流分歧被大大减少。表 3.7同时验证了变动采样策略的结果的正确性，并且可以看出，变动采样的电容达到收敛所需的行走步数要略多于未使用之前，这都与我们之前的理论分析相符合。

下面一部分内容将验证使用了第 3.3.3节之中所提出的的对同一个版图多条导线并发提取来增大可供GPU调度的warp数量的方法的可行性。我们生成了一个类似于图 3.9中的样例三的三层互连结构，不同的是每一层的平行导体数量增大到140条。我们从$M2$层中选出若干条导体做并发提取。

从表 3.8的结果可以看出，使用CPU顺序地对多条导线的电容做提取的时候，所需时间随着所需提取的导线的数目线性增加，而GPU的时间随着导体数目增加的速度则明显要慢很多。这种性质尤其适合于实际的大规模问题中当有成千上万条导线的电容都需要提取的时候。

表 3.8　多条导线并发提取时对效率的影响(每条导体上电容值收敛标准为0.5%的$1-\sigma$误差)

| 并发提取的导线数目 | CPU串行程序的时间 | GPU多导线并发提取时间 | 加速比 |
|---|---|---|---|
| 32 | 304 | 10.1 | 30.1 |
| 64 | 610 | 11.6 | 52.6 |
| 128 | 1246 | 21.0 | 59.3 |

## 3.5　本章小结

本章介绍了使用GPU加速电容提取的随机行走算法的技术，这些技术从减小指令流分歧，优化线程负载均衡，减小内存访问次数，增大并发线程数量等方面着手对算法进行优化。文章的最后通过实验来测试这些技术的效果，大量的实验数据首先验证了这些技术的正确性，与CPU上的随机行走的运行时间的对比显示GPU能够给电容提取的随机行走算法带来平均超过20倍的加速比，并且对于计算量更大的问题，能够达到更高的加速比。

# 第 4 章　面向三维芯片热仿真的GPU并行算法

在本章中，首先介绍了为包含液态冷却管道的三维堆叠式芯片进行准确热建模以及热瞬态仿真的方法，随后介绍了在GPU平台上的并行GMRES 算法，针对其提出了面向GPU加速的预条件技术。在最后的数值试验部分，给出了一个用于本实验研究的含液态冷却管道的三维堆叠式芯片的几何与材料信息，并使用前面介绍的GPU平台上的带预条件的GMRES算法对其进行了瞬态热仿真研究。

## 4.1　背景介绍

为了对一个含微流导热管道的三维芯片进行热建模，首先对其进行离散化。离散化后的三维芯片将包含三种单位元：固态单位元，液态单位元以及位于固液态单位元之间的管道壁单位元，如图 4.1所示，三种单位元有着不同的热流公式。对于非位于导热管道壁上的固态单位元，其热传导方程为

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_R \rho\hat{u}\,\mathrm{d}x = -\int_S (-k\nabla T)\cdot\vec{n}\,\mathrm{d}S + \int_R \dot{q}\,\mathrm{d}R \tag{4-1}$$

其中$R$是控制体积，$S$是其表面，$\vec{n}$是表面的外法向量，$\rho$是材料密度，$\hat{u}$是比内能，$k$是材料导热系数，$T$是温度，$\dot{q}$是控制体积$R$内的发热功率。对公式 (4-1)使用高斯定理，并根据$\mathrm{d}u = c_p\,\mathrm{d}T$这一关系，热传导方程可以写成

$$\rho c_p\frac{\partial T}{\partial t} = k\nabla^2 T + \dot{q} \tag{4-2}$$

其中$c_p$是材料比热。使用有限差分法对式 (4-2)进行离散化，并假设微流管道的方向是沿着$x$轴方向，可以得到

$$\begin{aligned}\rho c_p\frac{\mathrm{d}T}{\mathrm{d}t} = {} & \frac{k_{xx}}{\Delta x^2}(T_{i+1,j,k} - 2T_{i,j,k} + T_{i-1,j,k}) + \frac{k_{yy}}{\Delta y^2}(T_{i,j+1,k} - 2T_{i,j,k} + T_{i,j-1,k}) \\ & + \frac{k_{zz}}{\Delta z^2}(T_{i,j,k+1} - 2T_{i,j,k} + T_{i,j,k-1}) + g(\vec{v},t)\end{aligned} \tag{4-3}$$

其中$T_{i,j,k}$是在离散网格$(i,j,k)$上的温度，$k_{xx}$，$k_{yy}$和$k_{zz}$分别是沿着$x$，$y$，$z$方向的热导系数，$g(\vec{v},t)$ 是体积$\Delta v = \Delta x\Delta y\Delta z$内产生的热量。

对于位于微流管道内部的流体单位元，其热传导公式为

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_R \rho\hat{u}\,\mathrm{d}x = -\int_S (\rho\hat{h})\vec{u}\cdot\vec{n}\,\mathrm{d}S - \int_S (-k\nabla T)\cdot\vec{n}\,\mathrm{d}S + \int_R \dot{q}\,\mathrm{d}R \tag{4-4}$$

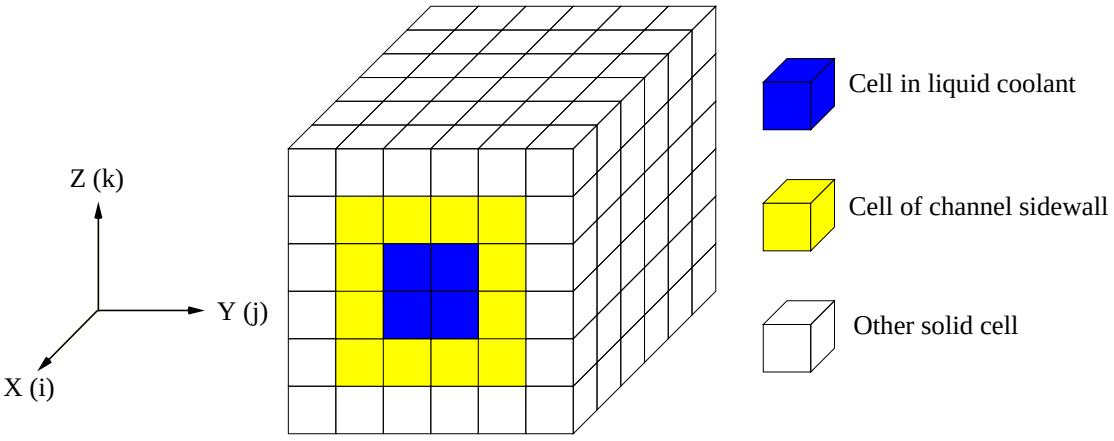



图 4.1　含微流管道的三维芯片离散后的分成的三种单位元示意图

其中$\hat{h}$是对流系数，$\vec{u}$流体流动的速度，其余变量定义同式 (4-1)，并注意其与式 (4-1)的区别。对公式 (4-4)使用高斯定理可以得到

$$\rho c_p \frac{\partial T}{\partial t} = -\rho c_p \vec{u} \cdot \nabla T + k\nabla^2 T + \dot{q} \tag{4-5}$$

使用有限差分法对其进行离散化可以得到

$$\begin{aligned} \rho c_p \frac{\mathrm{d}T}{\mathrm{d}t} = & \frac{k_{xx}}{\Delta x^2}(T_{i+1,j,k} - 2T_{i,j,k} + T_{i-1,j,k}) + \frac{k_{yy}}{\Delta y^2}(T_{i,j+1,k} - 2T_{i,j,k} + T_{i,j-1,k}) \\ & + \frac{k_{zz}}{\Delta z^2}(T_{i,j,k+1} - 2T_{i,j,k} + T_{i,j,k-1}) + u_{xx}\frac{\rho c_p}{2\Delta x}(T_{i+1,j,k} - T_{i-1,j,k}) + g(\vec{v}, t) \end{aligned} \tag{4-6}$$

其中$u_{xx}$是沿着$x$轴方向的流体流速，其余变量定义同公式 (4-3)，并注意与固体热传导公式相比，多出了对流导热项

$$u_{xx}\frac{\rho c_p}{2\Delta x}(T_{i+1,j,k} - T_{i-1,j,k}) \tag{4-7}$$

其可以看成是一个温控热源(类比于电路中的压控电流源)，由于对流导热项的引入，导致了式 (2-19)中的热导矩阵$\boldsymbol{G}$ 不对称。

对于位于管道壁上的单位元，定义一个其与管道中流体热交换的等效热传导系数$h_{side}$，则其余管道中流体的热导为$g_{side} = h_{side} \times S$，其中$S$ 为管道壁与管道中流体的接触面积。

我们在这里强调我们工作与前人的不同，在[41,43]等工作中使用一个很大的等效的热导系数来等效热流的导热效果，即用一项热导来等效$u_{xx}\frac{\rho c_p}{2\Delta x}(T_{i+1,j,k} - T_{i-1,j,k})$这一项，因此，在他们的工作中，只使用到了热导原件，因此生成的式 2-19热导矩阵$\boldsymbol{G}$ 是对称的，因此他们的工作可以用一系列的优化手段来进行加速。然而，他们的工作时不精确的，因为在实际使用中流体的流动速度是实时可变的，目前没有理论能够得到流体速度与等效热导系数的对应关系。

对每一个单位元选择合适的热传导方程，可以将含有液态冷却管道的三维堆叠芯片的热电容建立成通用的偏微分方程组形式，如式 (2-19) 所示，其中的$\boldsymbol{G},\boldsymbol{C},\boldsymbol{B}$ 矩阵及$\boldsymbol{U}$ 向量都只与电路的构成有关，是可以得到的。在本论文中关注的内容是三维电路的热瞬态效应，即对一个三维电路，给其一个初始的温度分布，仿真观察该电路的温度在达到稳定之前的变化过程。瞬态分析要求的是温度$\boldsymbol{T}$随时间变化的函数。假设对问题进行瞬态分析的时候所考虑的步长固定，设为$h$，使用一阶向后差分代替微分，即

$$\frac{d\boldsymbol{T}(t_k)}{dt} = \frac{\boldsymbol{T}(t_k) - \boldsymbol{T}(t_{k-1})}{h} \tag{4-8}$$

将差分代替微分公式带入式 (2-19)，可得

$$\boldsymbol{G}\boldsymbol{T}(t) + \boldsymbol{C}\frac{\boldsymbol{T}(t_k) - \boldsymbol{T}(t_{k-1})}{h} = \boldsymbol{B}\boldsymbol{U}(t) \tag{4-9}$$

调整方程中各项元素的位置，有

$$(\boldsymbol{G} + \frac{\boldsymbol{C}}{h}) \cdot \boldsymbol{T}(t_k) = \boldsymbol{B}\boldsymbol{U}(t_k) + \frac{\boldsymbol{C}}{h} \cdot \boldsymbol{T}(t_{k-1}) \tag{4-10}$$

因为步长$h$是固定的，则矩阵$(\boldsymbol{G} + \frac{\boldsymbol{C}}{h})$在瞬态仿真的每一步都是固定的，定义

$$(\boldsymbol{G} + \frac{\boldsymbol{C}}{h}) \triangleq \boldsymbol{A} \tag{4-11}$$

将式 (4-11)带入式 (4-10)可有

$$\boldsymbol{A} \cdot \boldsymbol{T}(t_k) = \boldsymbol{B}\boldsymbol{U}(t_k) + \frac{\boldsymbol{C}}{h} \cdot \boldsymbol{T}(t_{k-1}) \tag{4-12}$$

在进行热瞬态仿真的时候，首先定义电路的初始温度分布$\boldsymbol{T}(0)$，每一步仿真通过当前温度$\boldsymbol{T}(k-1)$ 与式 (4-12) 可得到下一个温度$\boldsymbol{T}(k)$。每一步仿真的工作相当于求解一个线性方程组$\boldsymbol{A}$。

通常认为，若方程组固定，而右端项变化，对线性方程组进行多次求解时，最佳选择应为直接解法。以LU 解法为例的直接解法的主要时间开销在对线性方程组$\boldsymbol{A}$进行LU 分解上，而LU分解完成之后，只需进行简单的三角矩阵的前代和回代即可求解。对于右端项变化的问题，耗时的LU 分解只需要进行一次，这个时间可以被之后的每次求解的时间所分摊。相比之下，如果用迭代解法来计算仅右端项变化的问题，每次计算都需要进行一个完整的迭代算法。因此通常认为直接解法对仅右端项变化的问题效率优于迭代解法。然而，对三维电路进行热瞬态分析的问题具有其特殊性，因为芯片上的温度在开始工作之后会向稳态发展，当仿真进行到一定步数之后，每次的$\boldsymbol{T}(k-1)$与$\boldsymbol{T}(k)$的差别很小，因此在计算温度$\boldsymbol{T}(k)$的时候，可以用$\boldsymbol{T}(k-1)$来作为迭代的初值。由于初值与解差别很小，因此

迭代解法通常能够在很短的步数内达到收敛，效率很高。此外，使用有限差分法对含液态冷却管道进行热建模所生成的线性方程组是大规模，不对称，高度稀疏的，迭代解法对其求解的效率应该优于直接解法。综合上述考虑，本章的接下来工作中将介绍使用GPU平台并行的GMRES 算法来对其进行瞬态热仿真。

广义最小剩余法(Generalized Minimum Residual, GMRES) 通常用来求解大规模稀疏线性方程组$\mathbf{Ax} = \mathbf{b}$。算法 2显示的是一个带有左预条件的基于Krylov子空间构造法的GMRES算法[56]，其使用投影法生成Kryloy子空间的基向量[23,56]。通过观察可以发现，在算法 2中含有很多适合在GPU 上计算的操作，例如，矩阵向量乘操作，向量求和及向量求二阶范数等操作。在本文所涉及的例子中，矩阵向量乘操作能够占据约50%的用于构造Krylov子空间的总时间[57]。而在GPU平台上并行的矩阵向量乘操作在CUSP[58] 中有着高效实现。本文中使用的并行的GMRES算法如图 4.2所示[57]。CPU与GPU之间的数据传输速度很慢，因此在程序设计的时候应该尽可能地减少CPU与GPU之间的数据交换[34]。在图 4.2所示的算法中，在开始的时候将所有数据从CPU 端拷贝到GPU 端，这个步骤是不可避免的。在每次Arnoldi迭代结束之后，都会将上Hessenberg矩阵中的一列拷贝到CPU端。这样做的原因是每次迭代都要将Hessenberg矩阵三角化，而矩阵三角化的过程具有很强的数据依赖性，很难将其并行化，因此在CPU端的执行效率更高。但由于Hessenberg矩阵的维度最多为$(m+1)\times m$，其中$m$ 为程序重启动值，该值相比起矩阵的维度来说是一个十分小的值，CPU 与GPU 之间的数据交换时间可以忽略。如图 4.2 所示，在每次Arnoldi 过程结束之后，会生成上Hessenberg矩阵的一个列向量$\vec{h}$，调用CUDA的API将其从GPU端的内存拷贝到CPU端，然后在CPU端通过一系列的Givens 旋转将上Hessenberg 矩阵$\tilde{H}$三角化，并判断当前残差是否已经满足收敛要求。除了Hessenberg矩阵之外，所有的矩阵，向量操作都可以在GPU端完成，因此，在图 4.2所示的算法中，算法十分高效。通过大量的实验数据显示，在图 4.2所显示的GPU上的GMRES 算法中，只有0.1%的时间花在了CPU 与GPU之间的数据传输上。

## 4.2　面向GPU加速的预条件技术

在使用如GMRES等迭代方法求解线性方程组的时候，一个重要的方面是构造预条件。迭代解法的预条件被使用来加速迭代的收敛速度。一个良好的预条件有可能使迭代算法的收敛速度大大加快。本节介绍的是在GPU 平台上构造针对GMRES 解法的预条件的方法。对于一个非奇异线性方程组$\boldsymbol{A}x = b$，其中$\boldsymbol{A}$为大规模稀疏非奇异矩阵。对于迭代方法，收敛速度依赖于矩阵$\boldsymbol{A}$的谱特征，因此，

**Algorithm 2** 带有左预条件的GMRES算法

**Require:** $A \in \mathbb{R}^{n\times n}$, $\mathbf{b} \in \mathbb{R}^n$, $\mathbf{x}_0 \in \mathbb{R}^n$ (initial guess), $\mathbf{M} \in \mathbb{R}^{n\times n}$ (preconditioner), $m$ (restart)

**Ensure:** $\mathbf{x} \in \mathbb{R}^n$: $\mathbf{Ax} \simeq \mathbf{b}$

1: $\mathbf{r}_0 \leftarrow \mathbf{M}(\mathbf{b} - \mathbf{A}\mathbf{x}_0)$
2: $\beta \leftarrow \|\mathbf{r}_0\|_2$, $\mathbf{w} \leftarrow \mathbf{r}_0/\beta$
3: **for** $j = 1$ to $m$ **do**
4: $\quad \mathbf{w} = \mathbf{M}\mathbf{A}\mathbf{v}_j$
5: $\quad$ **for** $i = 1$ to $j$ **do**
6: $\quad\quad h_{i,j} \leftarrow \mathbf{w}_i^{\mathrm{T}}\mathbf{v}_j$
7: $\quad\quad \mathbf{w} \leftarrow \mathbf{w} - h_{i,j}\mathbf{v}_i$
8: $\quad$ **end for**
9: $\quad h_{j+1,j} \leftarrow \|\mathbf{w}\|_2$, $\mathbf{v}_{j+1} \leftarrow \mathbf{w}/h_{j+1,j}$
10: **end for**
11: $\mathbf{V}_m \leftarrow [\mathbf{v}_1, \ldots, \mathbf{v}_m]$, $\tilde{\mathbf{H}}_m \leftarrow \{h_{i,j}\}_{1\le i\le j+1,\ 1\le j\le m}$
12: $\mathbf{y}_m \leftarrow \mathrm{argmin}_{\mathbf{y}}\|\beta\mathbf{e}_1 - \tilde{\mathbf{H}}_m\mathbf{y}\|_2$, $\mathbf{x}_m \leftarrow \mathbf{x}_0 + \mathbf{V}_m\mathbf{y}_m$
13: **if** tolerance satisfied **then**
14: $\quad$ Return
15: **else**
16: $\quad \mathbf{x}_0 \leftarrow \mathbf{x}_m$ and go to Line 1
17: **end if**

一个很自然的想法就是将线性方程组转换成具有相同解但是具有更好谱特征的等效情形。预条件矩阵就是可以完成上述转换的矩阵。

目前的绝大部分的预条件可以被分为显式的预条件和隐式预条件两类[59]。通常地，隐式预条件在每一次迭代的过程中需要求解一个线性方程组，对于隐式预条件，我们选择一个非奇异矩阵$\boldsymbol{M}$，使得$\boldsymbol{M} \approx \boldsymbol{A}$，并且应满足对$\boldsymbol{M}$的线性方程组求解的过程比对$\boldsymbol{A}$的线性方程组求解过程要简单。对于显式预条件，需要给出$\boldsymbol{A}$矩阵的近似逆矩阵，在每一次迭代的过程之中，需要计算矩阵向量乘。

隐式预条件方法作为曾今的主流算法，已经被研究了很长时间，并且被成功地应用在很多问题之中。然而，在最近一段时间里，更多的研究注意力被转移到了对显示预条件方法的研究[60]。有几个原因导致了这种研究注意力的转移，首先，隐式预条件法要求在每一次迭代的时候求解一次线性方程组，很多时候对线性方程组的求解过程是串行的，很难被并行化，尽管有很多复杂的并行策略被采

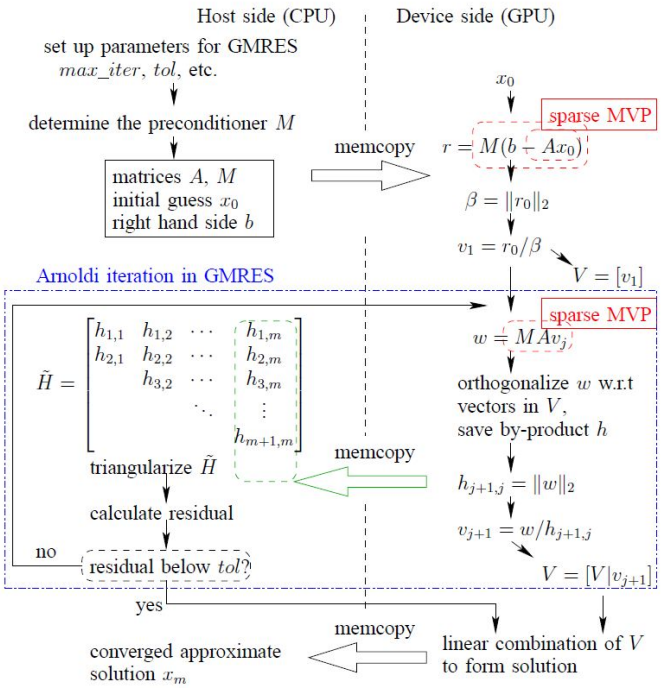



图 4.2　在GPU上并行的GMRES算法的流程图

用了，其并行度依然不高[45]；其次，在构造隐式预条件的过程中有可能会造成中断，即使使用选主元策略，有的时候这种中断还是无法避免[61,62]。随着并行计算架构的发展，隐式预条件方法中的三角矩阵求解过程已经成为了效率的瓶颈。而相比之下，显式预条件方法的预条件操作仅为求解矩阵向量乘，很容易被并行化，非常适合现在的并行计算机架构。因此尽管一般来说，显式预条件的收敛速度要比隐式预条件收敛速度慢，但是因为其并行程度高，反而更适合今天的计算机。

最普通的被广泛被使用的隐式预条件是基于不完全LU 分解(Incomplete LU, ILU)的。对于ILU预条件，定义预条件矩阵$\boldsymbol{M} = \tilde{L}\tilde{U}$，其中$\tilde{L}$ 与$\tilde{U}$分别近似$\boldsymbol{A}$矩阵的完全LU分解的因子$\boldsymbol{L}$与$\boldsymbol{U}$。使用ILU预条件法需要在每一步迭代的时候使用前代和回代方法求解上、下三角阵各一次，由于前代和回代过程的数据依赖性，很难被并行化，这个过程是并行算法的瓶颈所在。对于ILU类型的预条件来说，$\tilde{L}$与$\tilde{U}$矩阵的非零填入元越多，$\tilde{L}\tilde{U}$对矩阵$\boldsymbol{A}$的近似程度增加，使得迭代达到收敛的速度加快；然而，对三角矩阵$\tilde{L}$与$\tilde{U}$ 求解的并行化一般依赖于层次调度的方法，非零填入元的增加，会使得不同列之间的依赖性增加，进而减少并行度，严重降低运行效率。对于在GPU上并行的算法来说，很多时候，增加非零填入元带来的迭代收敛速度的增加反倒比不上其负面效果[45]。因此，在本文的实验中，采取的是ILU 类型预条件中的填入元最少的ILU0预条件方法，即在$\boldsymbol{L}$ 与$\boldsymbol{U}$矩阵的非零元的分布位置与$\boldsymbol{A}$矩阵对应的上下三角部分完全一致。使用ILU0 预条件的另外一个好处是，因为其非零元的位置分布已知，我们可以用这些位置信息来指导对$\boldsymbol{A}$矩

阵的并行的不完全分解[36]；并且其稀疏矩阵的存储结构已知，省去了在计算时动态分配空间的时间开销。

### 4.2.1　隐式预条件

本节介绍在GPU上并行地构造ILU预条件的过程。ILU预条件的构造算法基于稀疏矩阵的不完全LU分解。不完全LU 分解的方法包含right-looking与left-looking两种，由于left-looking 更容易被并行化，在本论文工作中考虑的是left-looking的算法。串行的left-looking的ILU0算法如算法 3所示。其核心操作为向量乘和向量求和操作。 在工作[37]的工作中给出了关于left-looking的完全LU分

**Algorithm 3** 串行的Left-looking的ILU0分解算法

```
L = I
for k = 1, ··· , n do
    // 求解线性方程组 Lx = b,    b = A(:, k)是矩阵A的第 k列
    x = b;
    for j = 1 : k − 1 其中 U(j, k) ≠ 0 do
        //向量乘加操作
        for s = j + 1 : n 其中 U(s, k) ≠ 0 do
            x(s) = x(s) − L(s, j) · x(j);
        end for
    end for
    U(1 : k, k) = x(1 : k);
    L(k : n, k) = x(k : n)/U(k, k);
end for
```

解的并行方法，该方法可以稍加改动用于对ILU0分解算法的并行化中。定义表$S(V, LV)$，其中$V = 1, 2, \ldots, n$，对应于表$S$中的节点，

$$LV = level(k) | 1 \le k \le n \tag{4-13}$$

其中$level(k)$对应于每个节点的层次。每个节点的层次可以根据矩阵$\boldsymbol{A}$的上三角部分$\boldsymbol{U}$的非零元分布位置得到，其的定义如下：

**定义** 4.1：　$level(k) = \max(-1, level(j_1), level(j_2), \ldots) + 1$，其中$j_1$是上三角部分$\boldsymbol{U}$中的第$k$列中非零元素的行坐标

根据定义 4.1可以得到矩阵$\boldsymbol{A}$中每一列的$level$值，具有相同$level$值的列之间没有数据依赖关系，可以并行地计算。例如对图 4.3而言，其左侧是上三角阵$\boldsymbol{U}$中的

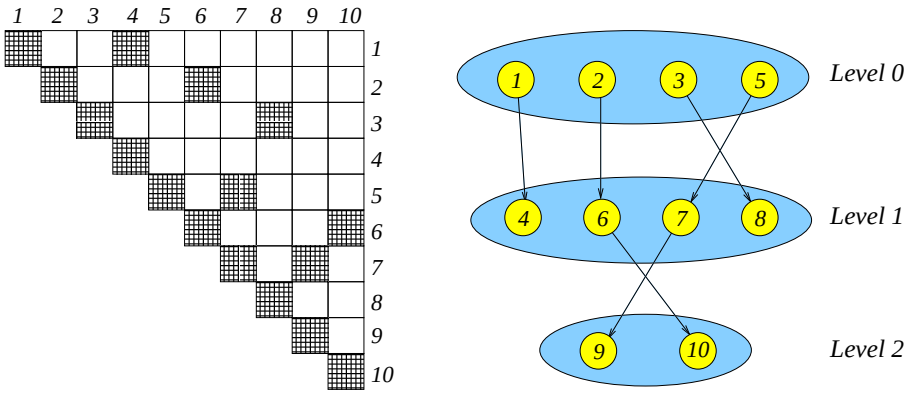



图 4.3　一个10 × 10的矩阵中各列层次计算示意图，左侧表示矩阵上三角部分的非零元分布，右侧的箭头表示列之间的依赖关系，在相同层次的列可以并行地求解，例如层次0中的第1,2,3,5列等

非零元分布，右侧是根据定义 4.1得到的各列所在的层次，在相同层次的列可以并行地求解，例如层次0中的第1,2,3,5列，层次1中的4,6,7,8列等。

在通过ILU0分解得到矩阵的不完全$\boldsymbol{L}, \boldsymbol{U}$分量之后，可以开始GMRES算法的迭代计算。使用ILU0预条件方法的预条件操作位在算法 3的步骤(4)解两次三角矩阵的线性方程组$\boldsymbol{LUv} = \boldsymbol{w}$。如果三角矩阵$\boldsymbol{L}$或$\boldsymbol{U}$矩阵是稠密矩阵，不同的列之间是互相依赖的，亦即各列之间的求解需要按照一定的顺序进行，不能够进行并行化。然而，对于实际的计算中，因为我们使用了非零元填入最少的ILU0预条件方法，最大程度地保证了矩阵$\boldsymbol{L}$和$\boldsymbol{U}$的稀疏性，这使得各列(行)之间的计算具有一定的并行度。类似对矩阵进行不完全LU分解，亦可以通过为每一行(列)定义层次的方法来将三角矩阵求解的过程并行化。以$\boldsymbol{L}$ 矩阵为例，为每一个列设置一个层次值*level*，其定义为

**定义** 4.2：　$level(i) = 1 + \max_j\{level(j)\}, \quad$ for all $j$ such that $L_{ij} \neq 0$

根据层次值得定义，可以通过一个循环简单地得到每一列所在的层次。在求解三角矩阵的时候，位于相同层的列可以并行地计算，而位于不同层的列则互相具有依赖性，算法 4显示的是一个基于将同层的列并行的计算的算法[45]。

### 4.2.2　显式预条件

基于近似逆的预条件(Approximate Inverse-based Preconditioners, AINV)是一类显式预条件方法，其直接生成$\boldsymbol{A}$矩阵的近似逆矩阵$\boldsymbol{M}, \boldsymbol{M} \approx \boldsymbol{A}^{-1}$。在对迭代线性方程组做预条件的时候，例如算法 2中的第四步，其操作为矩阵向量乘，其很容易被并行化。本文相关工作中研究的是在文章[62]中提出的基于A-双共轭的近似逆预条件方法。该算法计算两组A-双共轭的向量$\{z_i\}_{i=1}^n$及$\{w_i\}_{i=1}^n$，即当且仅当$i \neq j$ 时

**Algorithm 4** 基于层次划分的并行的三角矩阵求解算法

```
for lev = 1, ..., nlev do
    j1 = level(lev)
    j2 = level(lev + 1) − 1
    for k = j1, ..., j2 do
        i = q(k)
        for j = ial(i), ..., ial(i + 1) − 1 do
            x(i) ← x(i) − al(j) × x(jal(j))
        end for
    end for
end for
```

有$w_i^T A z_j = 0$。给定一个非奇异的矩阵$\boldsymbol{A} \in \mathbb{R}$，计算$\boldsymbol{A}$的逆与计算两组A-共轭的向量$\{z_i\}_{i=1}^n$及$\{w_i\}_{i=1}^n$有着密切的关系。令

$$\boldsymbol{Z} = [z_1, z_2, \ldots, z_n] \tag{4-14}$$

其中$z_i$为$\boldsymbol{Z}$矩阵的第$i$列，令

$$\boldsymbol{W} = [w_1, w_2, \ldots, w_n] \tag{4-15}$$

其中$w_i$为$\boldsymbol{W}$矩阵的第$i$列。由$z_i$与$w_i$的双共轭性有

$$W^T A Z = D = \begin{pmatrix} p_1 & 0 & \ldots & 0 \\ 0 & p_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & p_n \end{pmatrix} \tag{4-16}$$

其中$p_i = w_i^T \boldsymbol{A} z_i \neq 0$，很容易得出$\boldsymbol{W}$及$\boldsymbol{Z}$非奇异并且有

$$\boldsymbol{A}^{-1} = \boldsymbol{Z}\boldsymbol{D}^{-1}\boldsymbol{W}^T = \sum_{i=1}^{n} \frac{z_i w_i^T}{p_i} \tag{4-17}$$

因此，如果两组A-共轭的向量$\boldsymbol{Z}$及$\boldsymbol{W}$已知，那么$\boldsymbol{A}$的逆矩阵也是可以得到的。注意矩阵$\boldsymbol{Z}$及$\boldsymbol{W}$的选取并不是唯一的，有无穷多种满足式 (4-17)的取值。可以通过对任意两个非奇异的初始矩阵$\boldsymbol{W}^{(0)}, \boldsymbol{Z}^{(0)} \in \mathbb{R}^{n\times n}$的列向量做双共轭操作，得到列向量是A-双共轭的矩阵$\boldsymbol{Z}$及$\boldsymbol{W}$。为了方便起见，选择初始矩阵$\boldsymbol{W}^{(0)} = \boldsymbol{Z}^{(0)} = \boldsymbol{I}_{n\times n}$，并对矩阵的列向量进行双共轭操作。令$a_i^T$代表矩阵$\boldsymbol{A}$的第$i$个列向量，$b_i^T$代表矩阵$\boldsymbol{A}$的第$i$个行向量，则使用双共轭过程求$\boldsymbol{Z}$及$\boldsymbol{W}$的算法描述如算法 5 所示[62]。

其中$\boldsymbol{Z}$与$\boldsymbol{W}$是两个单位上三角矩阵，$\boldsymbol{D}$ 是一个对角阵。$\boldsymbol{Z}$和$\boldsymbol{W}$分别近似矩阵$\boldsymbol{A}$ 的LDU分解$\boldsymbol{A} = \boldsymbol{LDU}$中的$\boldsymbol{U}^{-1}$ 与$\boldsymbol{L}^{-1}$ 部分。它们可以用双共轭的方法直接得到[62]。定义$\boldsymbol{M}_l = \boldsymbol{W}^T$ 与$\boldsymbol{M}_r = \boldsymbol{ZD}^{-1}$，预条件操作的过程既为对$\boldsymbol{M}_l$与$\boldsymbol{M}_r$分别进行矩阵向量乘的过程，这个过程非常容易被并行化。在本论文的相关工作中，我们首先使用CPU计算得到预条件矩阵$\boldsymbol{M}$，然后将其传输到GPU，在GPU 上并行地做带预条件的GMRES 算法。基于A-共轭方法构造AINV的预条件的算法如算法 5所示[60]。

**Algorithm 5** 基于A-共轭方法构造AINV的预条件的算法

**Ensure:** Matrix $\boldsymbol{A}$

**Require:** Matrices of $\boldsymbol{W}, \boldsymbol{Z}$ and $\boldsymbol{D}$

$\boldsymbol{W} \leftarrow \boldsymbol{I}$, i.e., $\boldsymbol{w}_i \leftarrow \boldsymbol{e}_i$
$\boldsymbol{Z} \leftarrow \boldsymbol{I}$, i.e., $\boldsymbol{z}_i \leftarrow \boldsymbol{e}_i$
**for** $i = 1, 2, \ldots, n$ **do**
    **for** $j = i, \ldots, n$ **do**
        $d_j =< \boldsymbol{a}_i, \boldsymbol{z}_j >$
        $q_j =< \boldsymbol{b}_i, \boldsymbol{w}_j >$
    **end for**
    **for** $j = i + 1, \ldots, n$ **do**
        $\boldsymbol{z}_j = \boldsymbol{z}_j - \frac{d_j}{d_i}\boldsymbol{z}_i$
        $\boldsymbol{w}_j = \boldsymbol{w}_j - \frac{q_j}{q_i}\boldsymbol{w}_i$
        Drop value in $\boldsymbol{z}_i$ and $\boldsymbol{w}_i$ if need
    **end for**
**end for**

由于算法 4具有比较大的数据依赖性，对其并行化具有一定的难度，因此在本论文的相关工作中并没有将其做并行化，而是在CPU上串行地求出矩阵$\boldsymbol{Z}, \boldsymbol{D}, \boldsymbol{W}$。然后定义$\boldsymbol{V} \triangleq D^{-1}W^T$，这样

$$\boldsymbol{A}^{-1} = \boldsymbol{ZD}^{-1}\boldsymbol{W}^T = \boldsymbol{ZV} \tag{4-18}$$

在每次迭代中的预条件操作中只需要分别计算两次关于$\boldsymbol{Z}$与$\boldsymbol{V}$的矩阵向量乘，这个操作只有$O(n^2)$的复杂度，并且很容易并行化，在本论文的相关工作中，使用CUSPARSE[63]中定义的矩阵向量乘的实现。

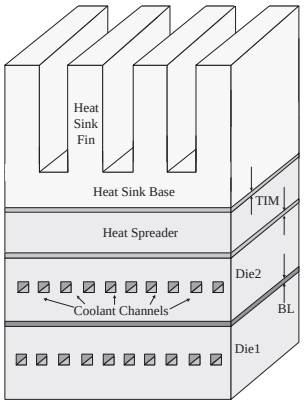



图 4.4　本节实验所采用的带液体冷却管道的三维堆叠式芯片示意图，其中BL表示两个器件层之间的隔离层，TIM 表示散热槽与器件层之间的界面层[57]

表 4.1　本节所使用带液体冷却管道的三维堆叠式芯片的几何及材料信息

| Layers | Geometry (mm) | Material |
|---|---|---|
| Die | $8 \times 8 \times 1$ | Silicon |
| TIM | $8 \times 8 \times 0.25$ | Indium |
| BL | $8 \times 8 \times 0.25$ | Silicon Nitride |
| Heat spreader | $8 \times 8 \times 0.5$ | Copper |
| Heat sink base | $8 \times 8 \times 0.5$ | Aluminum |
| Heat sink fin | $8 \times 8 \times 3$ | Aluminum |

## 4.3　实验结果与分析

我们使用NVIDIA CUDA实现了在GPU上并行的带预条件的GMRES算法，其硬件平台是Tesla C2070，其包含448个频率为1.15GHz的计算核心，拥有5GB 全局内存。作为比较的CPU平台为一台拥有四核Xeon E5620处理器的机器，处理器频率为2.00GHz，机器拥有20GB 的内存。在实验中我们比较了CPU平台的串行GMRES算法，GPU平台的并行GMRES算法，以及作为第三方的SuperLU_MT程序的效率。SuperLU_MT是一个公开的基于LU 分解的并行线性方程求解器，它常作为新方法效率的参考标准[25]。在运行时，CPU 上的SuperLU_MT程序使用四个核心并行计算，而CPU上的GMRES算法使用一个计算核心串行地运行。GMRES算法设置重启动阈值为60，迭代收敛条件为相对残差小于$10^{-6}$。

### 4.3.1　三维堆叠芯片测试用例

不失一般性地，在本章的实验中采用的模型是一个两层的三维堆叠式芯片，如图 4.4所示。其包含有2 个器件层，在其顶部拥有一个散热槽。在每一个器件层中都有微型液体冷却管道，在表 4.1中给出了该模型的具体几何及材料信息。

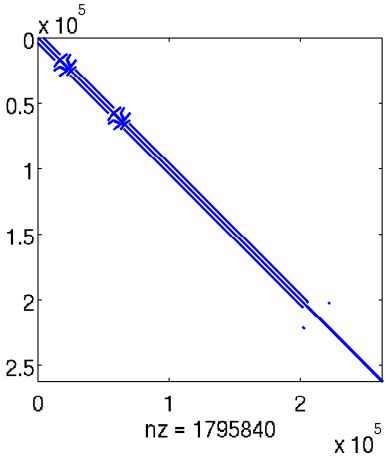



图 4.5　测试用例therm5的矩阵非零元分布图

### 4.3.2　矩阵求解结果的对比

为了比较图形处理器上带预条件的GMRES线性方程组解法的效率，根据图 4.4所示的电路模型生成了5个测试样例，其名称分别为therm1 至therm5，其中测试样例therm5的矩阵非零元分布图如图 4.5所示，可以看出因为其是一个大规模的高度稀疏的矩阵。同时为了方便其他研究人员与我们的求解器进行效率上的对比，本节分别从佛罗里达大学(University of Florida, UFL)的矩阵集(Matrix Collection)[64]挑选了2个例子，从3D-ICE[25] 的测试例子中挑选了4 个例子进行测试。这些例子的大小及非零元的密度各不相同，具有普遍的代表意义。关于这些测试用例的具体信息在表 4.2中列出。

在GPU平台上的带预条件的线性方程组求解器的效率也在表 4.2给出。作为比较，这些例子用下面的一些方法求解：

- CPU上的4核并行直接LU解法，即对矩阵进行完全LU分解得到$\boldsymbol{L}$和$\boldsymbol{U}$矩阵，然后分别求解两个三角矩阵，该解法使用SuperLU_MT的实现。
- 在CPU上串行的GMRES方法，包含无预条件，对角预条件，ILU0预条件及A-共轭的AINV预条件。
- 在GPU上并行的GMRES方法，包含无预条件，对角预条件，ILU0预条件及A-共轭的AINV预条件。

从表 4.2的数据可以看出，相比起不用预条件或者使用简单的对角预条件来说，使用ILU0即AINV预条件之后能够显著提高求解器的性能，使得求解器能够在合理的时间内对所有矩阵进行求解。同时可以观察到，对于小例子来说，GMRES算法并不能比直接LU 解法速度快，这是由于迭代算法和图形处理器的特性决定的，迭代算法的时间复杂度的常数系数比较大，图形处理器需要比较大的数据量来掩盖数据访问延迟，因此对小测试数据来说其效率会受到影响。然而，

表 4.2　使用不同方法求解$\boldsymbol{Ax} = \boldsymbol{b}$的效率对比，SuperLU使用四个CPU核心并行计算。对于部分例子，当没有预条件或者使用对角预条件的时候求解器不能够在一个合理的时间内运行完成，故其时间未在表格中列出。GMRES迭代重启动值为$m = 60$，迭代收敛条件为残差小于$10^{-6}$。所有时间单位为秒(sec)。

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|
| 矩阵 | 矩阵 | 非零元 | 非零元 | SuperLU | | GMRES | | | speedup | |
| 名称 | 维度 | 数目 | 比例 | fact. | sol. | precond | CPU | GPU | $\frac{C5}{C7+C9}$ | $\frac{C5}{C9}$ |
| G2 | 150,102 | 438,388 | 1.94e−5 | 0.8 | 0.03 | NON 0 | 8.9 | 2.2 | 0.4× | 0.4× |
| | | | | | | DIAG 0.003 | 0.3 | 0.04 | 19× | 20× |
| | | | | | | ILU0 0.7 | 0.03 | 0.06 | 1× | 13× |
| | | | | | | AINV 4.1 | 0.4 | 0.1 | 0.2× | 8× |
| G3 | 1,585,478 | 4,623,152 | 1.83e−6 | 30.7 | 1.9 | NON 0 | 211 | 16.5 | 2× | 2× |
| | | | | | | DIAG 0.02 | 7.0 | .024 | 697× | 1023× |
| | | | | | | ILU0 7.5 | 0.8 | 0.14 | 4× | 219× |
| | | | | | | AINV 42 | 8.4 | 0.68 | 0.7× | 45× |
| mc2rm | 20,000 | 109,000 | 2.72e−4 | 0.33 | 0.01 | ILU0 0.14 | 0.11 | 0.15 | 1.2× | 2× |
| | | | | | | AINV 0.18 | 0.14 | 0.15 | 1× | 2× |
| mc4rm | 50,250 | 329,140 | 1.30e−4 | 1.63 | 0.05 | ILU0 0.3 | 0.5 | 0.4 | 2× | 4× |
| | | | | | | AINV 5.0 | 0.5 | 0.2 | 0.3× | 8× |
| pf2rm | 20,000 | 104,100 | 2.60e−4 | 0.3 | 0.009 | ILU0 0.12 | 0.10 | 0.13 | 1× | 2× |
| | | | | | | AINV 1.8 | 0.13 | 0.12 | .2× | 3× |
| solid | 10,000 | 44,600 | 4.46e−4 | 0.03 | 0.002 | NON 0 | 0.17 | 0.35 | 0.1× | 0.1× |
| | | | | | | DIAG 0.001 | 0.08 | 0.15 | 0.2× | 0.2× |
| | | | | | | ILU0 0.06 | 0.05 | 0.12 | 0.2× | 0.2× |
| | | | | | | AINV 0.5 | 0.06 | 0.11 | .05× | 0.3× |
| therm1 | 57,344 | 390,144 | 1.18e−4 | 16.0 | 0.15 | ILU0 0.36 | 1.17 | 0.73 | 15× | 22× |
| | | | | | | AINV 3.02 | 1.18 | 0.63 | 4× | 25× |
| therm2 | 114,688 | 783,872 | 5.96e−5 | 78.4 | 0.42 | ILU0 0.72 | 3.88 | 1.56 | 35× | 50× |
| | | | | | | AINV 5.28 | 5.51 | 1.59 | 11× | 49× |
| therm3 | 147,456 | 1,004,032 | 4.61e−5 | 99.6 | 0.52 | ILU0 0.9 | 5.42 | 1.82 | 37× | 55× |
| | | | | | | AINV 6.5 | 6.3 | 2.0 | 12× | 50× |
| therm4 | 172,032 | 1,174,528 | 3.97e−5 | 177.2 | 0.7 | ILU0 1.1 | 3.37 | 1.29 | 74× | 137× |
| | | | | | | AINV 7.4 | 10.4 | 2.51 | 18× | 70× |
| therm5 | 262,144 | 1,795,840 | 2.61e−5 | 554.1 | 1.45 | ILU0 1.62 | 8.53 | 2.12 | 148× | 261× |
| | | | | | | AINV 11.9 | 17.0 | 3.34 | 36× | 165× |

表 4.3　对于本节的带液态冷却技术的三维堆叠式芯片的热瞬态仿真的结果，其中SuperLU使用4个CPU核心进行LU 分解，第二列的SuperLU的时间包含LU分解及三角矩阵的求解时间。GMRES迭代重启动值为$m = 60$，迭代收敛条件为残差小于$10^{-6}$。所有时间的单位为秒(sec)。

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| circuit | superLU | GMRES | | | Speedup |
| name | fact.+sol. | precond | CPU | GPU | (C2/C5) |
| therm1 | 254.1 | ILU0 | 55.5 | 34.3 | 7× |
| | | AINV | 100.2 | 45.8 | 6× |
| therm2 | 829.8 | ILU0 | 154.3 | 69.5 | 12× |
| | | AINV | 274.2 | 76.0 | 11× |
| therm3 | 1081.7 | ILU0 | 203.6 | 75.9 | 14× |
| | | AINV | 357.2 | 87.4 | 12× |
| therm4 | 1561.8 | ILU0 | 93.4 | 39.6 | 40× |
| | | AINV | 804.4 | 209.2 | 8× |
| therm5 | 3463.1 | ILU0 | 154.6 | 54.9 | 63× |
| | | AINV | 1374.7 | 227.3 | 15× |

当测试数据量增大之后，GMRES算法的效率开始远远领先于直接的LU解法。

### 4.3.3　三维芯片热瞬态分析结果

在这部分的内容中将测试对带液态冷却技术的堆叠式三维芯片的热瞬态仿真。由于在表 4.2的数据中可以观察到，对于这些例子的不含预条件及使用对角预条件的GMRES算法并不收敛，故本节之中的实验只对比直接LU解法以及使用ILU0预条件和AINV预条件的GMRES之间的效率。注意到在瞬态仿真之中，LU分解以及预条件都只需要进行一次，这是因为$\boldsymbol{A} \triangleq \boldsymbol{G} + \boldsymbol{C}/h$，而瞬态仿真的步长$h$恒定，故$\boldsymbol{A}$矩阵在仿真的每一步都是一样的。关于瞬态仿真的结果在表 4.3中给出，表中的计算时间包含除了矩阵输入之外的所有时间。

为了证明所有计算结果的正确性，图 4.6给出了不同解法对测试样例therm5的瞬态分析中同一个点的温度变化曲线。图 4.6中不同计算方法得到的结果高度吻合，验证了我们实现的各种算法的正确性。

从表 4.3可以看出，GMRES算法对热瞬态分析的计算效率非常高。通常认为在做瞬态分析的时候基于矩阵分解的直接解法的效率会更高一些，因为直接解法的主要时间开销是矩阵分解，在得到矩阵的分解结果之后所需要进行的三角矩阵求解只需要进行简单的前代/回代操作，时间复杂度仅为$O(n^2)$，而矩阵分解的过程对于瞬态仿真来说只需要进行一次，矩阵分解的时间开销可以被后面的多步瞬

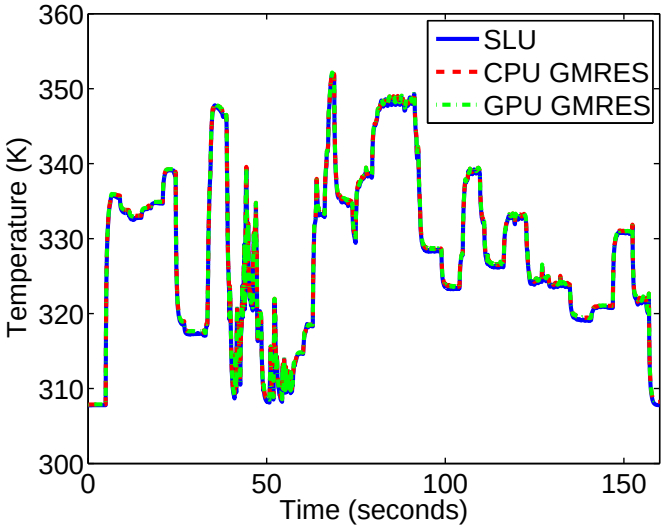



图 4.6　不同解法对对测试样例therm5的瞬态分析中同一个点的温度变化曲线，其中实线是SuperLU的结果，短划线是CPU 上的GMRES算法的结果，点划线是GPU上的GMRES算法的结果。

态仿真所分摊，故其效率高。然而另一方面，对于使用迭代解法的瞬态仿真来说，我们可以使用上一次的结果$x_{k-1}$来作为下一次迭代的初值，因为温度总是朝着稳态发展，故随着仿真的进行，$x_{k-1}$ 与$x_k$ 之间的差别越来越小，故迭代收敛之前所需要的迭代步数也将变得越来越小，有时候只需要几步迭代就可以收敛，故基于迭代的解法在热瞬态分析中同样十分高效。从表中也可以看出，即使是单线程的GMRES算法的效率也要高于串行的LU 解法的效率。

对于ILU0及AINV两种预条件来说，ILU0的效率任然还是要高于AINV预条件的效率，这也是为什么隐式的预条件方法在过去很长的一段时间中一直是研究和应用的主流的原因。然而，在GPU上AINV预条件能够获得的加速比要比ILU0的加速比更大，例如对于therm5而言，GPU上的ILU0预条件GMRES算法相对于CPU而言能够获得约为3 倍的加速比，而GPU上的AINV预条件的GMRES算法先对CPU而言能够获得约6倍的加速比。这是因为AINV预条件更符合多核计算体系的并行特性，是像AINV这样的显式预条件方法在多核计算机迅速发展的今天吸引越来越多研究关注的原因。也许随着将来计算机的并行度进一步的提高之后，更并行友好的AINV预条件能够超过ILU0预条件，成为将来超级计算机中使用的主流方法。

## 4.4　本章小结

本章首先对含有液态冷却管道的三维堆叠式芯片进行准确的热建模，相对于前人的工作，我们没有使用以牺牲精度为代价的近似手段，从而更加精确。文章随后对热建模所生成的大规模稀疏不对称线性方程组使用GMRES 方法进行求解。本章给出了在GPU平台上并行的GMRES算法的算法，并介绍了在GPU 平台上对其进行预条件的ILU0 和基于A-共轭的AINV 方法。在数值实验部分，首先使用不同的线性方程组求解器对一些公开的以及本论文工作中生成的线性方程组进行求解，结果显示针对不同的问题，GPU上的GMRES算法性能有着很大差异，对于本论文工作中生成的线性方程组，GPU上的GMRES算法体现出良好的性能，当线性方程组规模较大的时候，加速比能够达到100倍以上。在本节的最后，使用在不同平台上的不同的线性方程求解器对一个两层的含液态冷却微管道的模型进行瞬态热仿真，实验结果表明在GPU 上的GMRES 算法相比起CPU上的SuperLU 求解器在针对大规模的瞬态仿真时能够到达63 倍的加速比。

# 第 5 章　总结与展望

## 5.1　总结

未来的处理器架构发展的一个趋势是传统中央处理器与图形处理器的整合。因此，研究适合于CPU-GPU异构平台的算法显得尤其重要。本文针对EDA领域中重要的但是又是极其耗费时间的电容提取和热分析问题在GPU平台上的并行化展开了研究，主要做了以下两个方面的工作：

- 提出了适合在图形处理器上运行并行随机行走电容提取算法。随机行走算法因其内存消耗少，对大问题复杂度低，精度可控等已经逐渐成为目前电容提取领域中的主流算法，然而随机行走算法的特性使得其不适合在图形处理器上运行：从算法流程上来说，随机行走算法中每个线程所需要执行的动作是随机的，只有在运行的时候才能知道具体动作，使得不同的线程之间具有严重的指令分歧，由于图形处理器的单指令多数据流的特点，这些分歧的指令需要被串行地执行，导致效率严重降低；从内存访问来说，随机行走中每个线程在跳转的时候需要从格林函数表中查找相应的跳转概率来决定所应该跳转到的下一个位置，格林函数表的大小决定了其不可能存储在GPU的片上共享内存之中，只能存储在片下的延时非常大的全局内存之中，并且由于每个线程访问的数据的位置是随机的，往往导致数据的空间局部性非常差，进一步降低了内存访问速度。为了减小指令分歧带来的效率的降低本文将整个随即行走算法流程分为三个更小的阶段，每个阶段执行相对简单，分歧较小的指令，使得每个阶段的指令分歧能够降到最小；并在三阶段算法的基础上，又提出了根据误差估计剩余步数的迭代的算法流程，可以在达到所需精度的时候所做的行走步数尽可能地少；此外，本文涉及的算法为每个线程分配独立存储空间，使得算法没有昂贵的指令同步的操作；算法为每个线程分配了额外的数据空间，可以使得速度快的线程做更多的运算，最终所有线程在相近的时间停下来，使得并发线程数能够最大化。为了减小内存访问延时对算法效率的影响，本文提出了ICPA数据结构，通过ICPA 能够大大降低内存访问次数，并消除了二分查找带来的指令流分歧及频繁内存访问，对于多介质的问题，无法直接使用ICPA的情况下，本文提出了一种变种的采样策略，以进行更多次行走的代价换来了ICPA在多介质问题上的应用，实验结果证明这种步数的牺牲能够换来更高的效率。在电容提取实际应用中的同一

版图多条线网需要提取的情形，本文提出并行地进行多条线网的提取工作的策略，使得计算量得以增加，能够更好地掩盖内存访问的时间，在原来算法的基础上获得额外的加速。

- 使用图形处理器对带有液态冷却管道的堆叠式三维芯片进行了热瞬态仿真。三维芯片是未来最有可能的延续计算机效率按照摩尔定律发展的技术，然而三维芯片的散热问题成为瓶颈，植入液态冷却管道是一种能够有效给芯片内部降温的手段。然而在对含有液体的芯片做热建模的时候，其方程组将会变得不再对称，传统的一些加速手段不能够使用；此外，之前的工作一般都针对电路温度稳态进行分析，而忽略了分析电路在达到热稳态之前的温度变化过程。本文从流体导热原理出发，使用有线差分法对带液态冷却管道的三维芯片进行了热瞬态的分析。为了加快计算速度，本文使用在图形显示器上的GMRES来求解所生成的线性方程组，在每次迭代的时候，将上一步的温度作为初值开始迭代，使得迭代过程在很短的步数下就能收敛，因此，GMRES算法能够比直接LU解法快数倍。此外在图形处理器上的带有预条件的GMRES算法能够进一步加速计算速度，以基于A-共轭的AINV预条件为代表的显式预条件方法的计算速度虽然仍然比以不完全LU 为代表的隐式预条件方法的速度要慢，但是其展示了在多核平台上的良好的加速潜力，随着未来并行多核计算架构的发展，很有可能成为将来的主流。

## 5.2 展望

在本论文的工作基础之上，还有一些地方可以加以改进。

- 首先，之前提出的随机行走算法虽然能够穿越两层介质，但是当遇到薄层的时候，跳转次数会增加，效率也因此降低。我们开始尝试使用新的格林函数建模法，使得转移区域能够穿越任意多层介质，这样，行走能够以更少的跳转得到计算结果，每个线程的跳转步数的差异也将减少，性能能够提高。这方面的工作上，我们已经能够计算得到穿越任意多层介质的格林函数，下面的工作就是在计算的时候使用这些新的格林函数进行跳转。
- 其次，本文的第四章给出了在GPU上的带有预条件的GMRES算法，这个算法能使用到其他很多的EDA问题之中，例如可以用来对边界元算法所生成的边界积分方程进行加速。
- 最后，未来计算机架构将会向着分布式存储，异构计算的方向发展，在不同计算节点之间的通信也将变得格外重要，在今后的工作中可以尝试构建多块图形处理器的集群，希望能够进一步获得更高的加速，并研究算法在分布式

集群上部署时可能出现的问题，为今后的相关工作打下基础。

# 参考文献


[1] Moore G E, et al. Cramming more components onto integrated circuits, 1965.

[2] StarRC$^{TM}$ Custom: Next-Generation Modeling and Extraction Solution for Custom IC Designs, 2010. http://www.synopsys.com/Tools/Implementation/SignOff/CapsuleModule/Starrc-Custom-Nextgeneration_wp.pdf/.

[3] Zhai K, Zhang Q, Li L, et al. A 3-D Parasitic Extraction Flow for the Modeling and Timing Analysis of FinFET Structures. Proceedings of International Conference on Communications, Circuits and Systems, 2012. 430–434.

[4] Nabors K, White J. FastCap: A multipole accelerated 3-D capacitance extraction program. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 1991, 10(11):1447–1459.

[5] Le Coz Y, Iverson R. A stochastic algorithm for high speed capacitance extraction in integrated circuits. Solid-State Electronics, 1992, 35(7):1005–1012.

[6] Székely V, Rencz M, Courtois B. Tracing the thermal behavior of ics. IEEE Design & Test of Computers, 1998, 15(2):14–21.

[7] Xu C, Jiang L, Kolluri S K, et al. Fast 3-D thermal analysis of complex interconnect structures using electrical modeling and simulation methodologies. Proceedings of IEEE/ACM International Conference on Computer-Aided Design-Digest of Technical Papers, 2009. 658–665.

[8] Im S, Srivastava N, Banerjee K, et al. Scaling analysis of multilevel interconnect temperatures for high-performance ICs. IEEE Transactions on Electron Devices, 2005, 52(12):2710–2719.

[9] Harmon D, Gill J, Sullivan T. Thermal conductance of IC interconnects embedded in dielectrics. Proceedings of IEEE International Integrated Reliability Workshop Final Report, 1998. 1–9.

[10] Oak Ridge Claims No. 1 Position on Latest TOP500 List with Titan. TOP500, 2012. http://www.top500.org/blog/lists/2012/11/press-release/.

[11] Williams L. Titan is Also a Green Powerhouse, 2012. https://www.olcf.ornl.gov/2012/11/14/titan-is-also-a-green-powerhouse/.

[12] Asia Student Supercomputer Challenge (ASC) 2013, 2013. http://online.wsj.com/article/PR-CO-20130423-911024.html/.

[13] Haji-Sheikh A, Sparrow E. The floating random walk and its application to Monte Carlo solutions of heat equations. SIAM Journal on Applied Mathematics, 1966, 14(2):370–389.

[14] Royer G. Monte Carlo Procedure for Theory Problems Potential. IEEE Transactions on Microwave Theory and Techniques, 1971, 19(10):813–818.

[15] 刘志. 基于随机行走算法的电容提取程序的完善. [学士学位论文].北京:清华大学, 2010.

[16] Rollins G. SESS Rapid3D 20X Performance Improvement. http://tinyurl.com/btoprsf, July, 2010.

[17] Zhuang H, Yu W, Hu G, et al. Fast floating random walk algorithm formulti-dielectric capacitance extraction with numerical characterization of Green's functions. Proceedings of 2012 17th Asia and South Pacific Design Automation Conference (ASP-DAC), 2012. 377–382.

[18] Samet H. Applications of spatial data structures. 1990..

[19] Bentley J L. Multidimensional binary search trees used for associative searching. Communications of the ACM, 1975, 18(9):509–517.

[20] Harrington R F, Harrington J L. Field computation by moment methods. Oxford University Press, 1996.

[21] Brebbia C A, Telles J C F, Wrobel L C. Boundary element techniques: theory and applications in engineering. Berlin and New York, Springer-Verlag, 1984, 478 p, 1984..

[22] Abramowitz M E, et al. Handbook of mathematical functions: with formulas, graphs, and mathematical tables, volume 55. Courier Dover Publications, 1964.

[23] Saad Y, Schultz M H. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM Journal on scientific and statistical computing, 1986, 7(3):856–869.

[24] Yu W, Wang Z. Enhanced QMM-BEM solver for three-dimensional multiple-dielectric capacitance extraction within the finite domain. IEEE Transactions on Microwave Theory and Techniques, 2004, 52(2):560–566.

[25] Sridhar A, Vincenzi A, Ruggiero M, et al. 3D-ICE: Fast compact transient thermal modeling for 3D ICs with inter-tier liquid cooling. Proceedings of IEEE Proceedings of the International Conference on Computer-Aided Design, 2010. 463–470.

[26] Li F, Nicopoulos C, Richardson T, et al. Design and management of 3D chip multiprocessors using network-in-memory. ACM SIGARCH Computer Architecture News, 2006, 34(2):130–141.

[27] Huang W, Ghosh S, Velusamy S, et al. HotSpot: A compact thermal modeling methodology for early-stage VLSI design. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2006, 14(5):501–513.

[28] Wang T Y, Chen C C P. 3-D thermal-ADI: A linear-time chip level transient thermal simulator. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2002, 21(12):1434–1445.

[29] Im S, Banerjee K. Full chip thermal analysis of planar (2-D) and vertically integrated (3-D) high performance ICs. Proceedings of IEEE International Electron Devices Meeting Technical Digest, 2000. 727–730.

[30] Li P, Pileggi L T, Asheghi M, et al. Efficient full-chip thermal modeling and analysis. Proceedings of IEEE/ACM International Conference on Computer Aided Design, 2004. 319–326.

[31] Cheng Y K, Raha P, Teng C C, et al. ILLIADS-T: An electrothermal timing simulator for temperature-sensitive reliability diagnosis of CMOS VLSI chips. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 1998, 17(8):668–681.

[32] Lienhard J H. A heat transfer textbook. Courier Dover Publications, 2011.

[33] Bergman T L, Incropera F P, Lavine A S, et al. Fundamentals of heat and mass transfer. John Wiley & Sons, 2011.

[34] Kirk D B, Wen-mei W H. Programming massively parallel processors: a hands-on approach. Morgan Kaufmann, 2010.

[35] Nvidia C. Cublas library. NVIDIA Corporation, Santa Clara, California, 2008, 15.

[36] Ren L, Chen X, Wang Y, et al. Sparse LU factorization for parallel circuit simulation on GPU. Proceedings of Proceedings of the 49th Annual Design Automation Conference. ACM, 2012. 1125–1130.

[37] Chen X, Wu W, Wang Y, et al. An escheduler-based data dependence analysis and task scheduling for parallel circuit simulation. IEEE Transactions on Circuits and Systems II: Express Briefs, 2011, 58(10):702–706.

[38] Zhao X, Feng Z. Fast multipole method on GPU: tackling 3-D capacitance extraction on massively parallel SIMD platforms. Proceedings of 2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC), 2011. 558–563.

[39] Qian H, Deng Y, Wang B, et al. Towards accelerating irregular EDA applications with GPUs. Integration, the VLSI Journal, 2012, 45(1):46–60.

[40] Qian H, Deng Y. Accelerating RTL simulation with GPUs. Proceedings of International Conference on Computer-Aided Design, 2010. 687–693.

[41] Feng Z, Li P. Multigrid on GPU: tackling power grid analysis on parallel SIMT platforms. Proceedings of IEEE/ACM International Conference on Computer-Aided Design, 2008. 647–654.

[42] Feng Z, Zeng Z, Li P. Parallel on-chip power distribution network analysis on multi-core-multi-GPU platforms. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2011, 19(10):1823–1836.

[43] Feng Z, Li P. Fast thermal analysis on GPU for 3D-ICs with integrated microchannel cooling. Proceedings of 2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2010. 551–555.

[44] Wang M, Klie H, Parashar M, et al. Solving sparse linear systems on NVIDIA Tesla GPUs. Proceedings of Computational Science–ICCS 2009. Springer, 2009: 864–873.

[45] Li R, Saad Y. GPU-accelerated preconditioned iterative linear solvers. The Journal of Supercomputing, 2013, 63(2):443–466.

[46] Bell N, Garland M. Implementing sparse matrix-vector multiplication on throughput-oriented processors. Proceedings of Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. ACM, 2009. 18.

[47] Couturier R, Domas S. Sparse systems solving on GPUs with GMRES. The journal of Supercomputing, 2012, 59(3):1504–1516.

[48] Nov*á*k J, Havran V, Dachsbacher C. GPU Computing GEMS. Elsevier Inc., emerald ed., 2011: 401–420.

[49] NVIDIA. NVIDIA GPU Computing SDK. https://developer.nvidia.com/gpu-computing-sdk.

[50] NVIDIA. NVIDIA's Next Generation CUDA$^{TM}$Compute Architecture: Fermi$^{TM}$. http://www.nvidia.com/object/fermi-architecture.html.

[51] Bontzios Y I, Dimopoulos M G, Hatzopoulos A A. An evolutionary method for efficient computation of mutual capacitance for VLSI circuits based on the method of images. Simulation Modelling Practice and Theory, 2011, 19(2):638–648.

[52] Lazić P, Štefančić H, Abraham H. The Robin Hood method — A new view on differential equations. Engineering analysis with boundary elements, 2008, 32(1):76–89.

[53] Mascagni M, Simonov N A. The random walk on the boundary method for calculating capacitance. Journal of Computational Physics, 2004, 195(2):465–473.

[54] Hwang C O, Mascagni M, Won T. Monte Carlo methods for computing the capacitance of the unit cube. Mathematics and Computers in Simulation, 2010, 80(6):1089–1095.

[55] Deschacht D, De Rivaz S, Farcy A, et al. Keep on shrinking interconnect size: is it still the best solution? Proceedings of 2010 34th IEEE/CPMT International Electronic Manufacturing Technology Symposium (IEMT), 2010. 1–4.

[56] Elman H C. Iterative methods for linear systems. Large-Scale Matrix Problems and the Numerical Solution of Partial Differential Equations, 1994, 3:69–177.

[57] Liu X X, Liu Z, Tan S D, et al. Full-chip thermal analysis of 3D ICs with liquid cooling by GPU-accelerated GMRES method. Proceedings of 13th IEEE International Symposium on Quality Electronic Design (ISQED), 2012. 123–128.

[58] Bell N, Garland M. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2010. Version 0.1. 0, 2011, 6.

[59] Benzi M, Meyer C D, Tuma M. A sparse approximate inverse preconditioner for the conjugate gradient method. SIAM Journal on Scientific Computing, 1996, 17(5):1135–1149.

[60] Xu S, Xue W, Wang K, et al. Generating approximate inverse preconditioners for sparse matrices using cuda and gpgpu. Journal of Algorithms & Computational Technology, 2011, 5(3):475–500.

[61] Saad Y. Preconditioning techniques for nonsymmetric and indefinite linear systems. Journal of Computational and Applied Mathematics, 1988, 24(1):89–105.

[62] Benzi M, T$\mathring{u}$ma M. Numerical experiments with two approximate inverse preconditioners. BIT Numerical Mathematics, 1998, 38(2):234–241.

[63] NVIDIA C. CUSPARSE library. NVIDIA Corporation, Santa Clara, California, 2011..

[64] Davis T A, Hu Y. The University of Florida sparse matrix collection. ACM Transactions on Mathematical Software (TOMS), 2011, 38(1):1.

# 致 谢

衷心感谢导师喻文健副教授对本人的精心指导，从本科后段到硕士毕业，他一直对我的工作给予着鼓励和支持，是他指导着我走上了学术的道路，他严谨的治学态度、敏锐的学术眼光、勤奋的工作态度和深厚的学术功底将使我受益终生。

硕士学习期间在美国里弗赛德加州大学(UC Riverside)电子工程系MSLAB进行三个月的合作研究期间，承蒙 Sheldon Tan 教授热心指导与帮助，并感谢MSLAB中的Liu Xue-Xin，Liu Zao及其他同学在生活和科研上对我的帮助。在上海Synopsys 公司的两个暑期实习的时间里，承蒙Star-RCXT 组的各位同事指导与帮助，不胜感激。感谢 EDA 实验室全体老师和同学们的热情帮助和支持!

感谢父母和兄长，在我学习和成长的过程中，他们一直给予我不计回报的鼓励与支持。感谢在我成长的过程中给过我帮助，和我一起奋斗过的同学和朋友们，他们使我变成了现在的我。

# 声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____ 日　期：_____

# 个人简历、在学期间发表的学术论文与研究成果

## 个人简历

1986 年 10 月 03 日出生于广西壮族自治区昭平县。

2006 年 9 月考入清华大学计算机系计算机专业，2010 年 7 月本科毕业并获得工学学士学位。

2010 年 9 月免试进入清华大学计算机系攻读工学硕士学位至今。

## 发表的学术论文

[1] **Zhai Kuangya**, Yu Wenjian and Zhuang Hao. GPU-Friendly Floating Random Walk Algorithm for Capacitance Extraction of VLSI Interconnects. Proceedings of the conference on Design, automation and test in Europe (DATE), 2013, pp. 1661-1666. (CCF推荐B类国际会议)

[2] **Zhai Kuangya**, Zhang Qingqing, Li Li and Yu Wenjian. A 3-D parasitic extraction flow for the modeling and timing analysis of FinFET structures. International Conference on Communications, Circuits and Systems (ICCCAS), 2012, pp. 430-434.

[3] Yu Wenjian, **Zhai Kuangya**, Zhuang Hao and Chen Junqing. Accelerated floating random walk algorithm for the electrostatic computation with 3-D rectilinear-shaped conductors. Simulation Modelling Practice and Theory, 2013, 34(5): 20-36. (SCI收录，影响因子1.0)

[4] **Zhai Kuangya**, Yu Wenjian. The 2-D Boundary Element Techniques for Capacitance Extraction of Nanometer VLSI Interconnects. International Journal of Numerical Modelling: Electronic Networks, Devices and Fields. (SCI收录，影响因子0.6)(一审推荐录取，小修)

[5] Liu Xue-Xin, **Zhai Kuangya**, Liu Zao, Tan Sheldon, Yu Wenjian. Parallel Finite Difference Thermal Analysis of 3D Integrated Circuits on CPU-GPU Platforms. IEEE Transactions on Very Large Scale Integration (VLSI) Systems. (SCI收录，影响因子1.2)(一审中)

## 发明专利

[1] 喻文健，翟匡亚，庄昊. 基于GPU的集成电路电容参数提取系统及方法: 中国发明专利，申请号：201310076174.1（2013年3月11日获申请号）

# 面向集成电路电容提取与热分析的GPU并行算法研究

翟匡亚