西北工業大學

本科毕业设计（论文）

题　目　基于华为鲲鹏平台的 SpTRSV 并行与优化研究

专业名称　计算机科学与技术

学生姓名　严愉程

指导教师　王云岚

完成时间　2021 年 6 月 1 日

摘　要

稀疏下三角矩阵求解器，是数值计算中的一个重要的方法，在科学计算中有着广泛的应用。由于其离散的数据访存、任务之间存在着很强的依赖、需要细粒度的同步、任务之间负载不均衡等特点，在现代多核系统中，如何提升 SpTRSV 算法的并行度与扩展性，仍然是一个挑战性的课题。

华为鲲鹏 920 处理器基于 ARMv8 架构，最多拥有 64 个核心，支持 ARM NEON 128bits 浮点运算。华为鲲鹏系统支持 NUMA 内存架构，实现最多 4 个 920 互联，最高支持 256 个计算核心。

本文基于华为鲲鹏 920 芯片，进行稀疏下三角矩阵求解算法的设计与优化。

主要工作包括:

1. 设计并实现了一套高效的稀疏下三角矩阵求解算法。
2. 针对华为鲲鹏 920 处理器众核体系结构以及 ARMv8 指令集的特点进行了优化。
3. 采用 ARM NEON 指令，对向量乘法部分进行了 SIMD 的优化。
4. 结合处理器 NUMA 架构的特性，通过分配矩阵数据的存储，充分利用内存带宽。

关键词： 稀疏下三角矩阵求解器, 鲲鹏 920, 众核优化, 并行算法

Abstract

The sparse triangular solve kernel, SpTRSV, is an important building block for a number of numerical linear algebra routines. It is commonly considered challenging to parallelize and scale even on a moderate number of cores. This challenge is due to the fact that triangular solver typically has frequent discrete memory access, load embalances between the tasks.

Huawei Kunpeng920 is an ARMv8-based server CPU with up to 64 cores, supporting ARM NEON 128-bit SIMD instruction set.

The main work of this article includes:

1. Design and implement an efficient sparse matrix triangular solve algorithm.
2. Optizing for Huawei Kunpeng920 multi-core processor and features of ARMv8 instruction set.
3. Utilizing ARM NEON 128bits SIMD instruction set to speed up the vector multiplication which is part of SpTRSV algorithm.
4. By taking NUMA feature into consideration and allocating matrix data properly, achieve better use of memory bandwidth.

Key Words: SpTRSV, Kunpeng920, Multi-core Optimization, Parallel Algorithm

目 录

第 一 章　绪论

1.1 课题背景

稀疏下三角矩阵求解 (Sparse Triangular Solve, SpTRSV) 并行算法是很多数值计算方法的重要组成部分，比如用直接法求解稀疏矩阵的线性方程组 [1]，迭代法求解稀疏矩阵的预条件子 [2]，以及最小二乘法的求解 [3]，是现代科学计算中一个广泛使用的计算核心，在数值模拟计算中，通常会使用迭代法或直接法求解大规模稀疏线性方程组，而 SpTRSV 的效率直接影响了线性方程组的求解效率，故提高 SpTRSV 算法的性能至关重要。

相比于稠密下三角举证解法器 [4] 或者其他稀疏矩阵计算方法，例如稀疏矩阵转置 [5]，稀疏矩阵向量乘法 [6][7]，和稀疏矩阵乘法 [8]，SpTRSV 频繁且离散地数据访存、任务之间存在着很强的依赖、需要细粒度的同步、任务之间负载不均衡等特点使得并行优化更加难以进行。

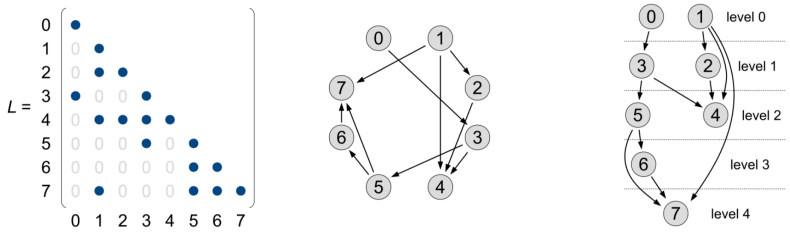
SpTRSV 算法主要是从方程组 $Lx = b$ 中求解未知数向量 x，其中 L 是下三角矩阵，且对角线上的元素都不为 0。要计算其中的一个未知数 x_k，需要先计算出其前置未知数 x_0, ..., x_{k-1} 中的一个子集。这些任务间的依赖关系，将 SpTRSV 的计算转换成了一个任务依赖图的计算（Task Dependent Graph，TDG），并且这个 TDG 是一张有方无环图 (Directed Acyclic Graph)。

华为作为鲲鹏计算产业的成员，聚焦于发展华为鲲鹏 + 昇腾双引擎芯片族，通过“硬件开放、软件开源、使能合作伙伴”来推动计算产业的发展。华为鲲鹏 920 处理器兼容了 ARMv8 架构。最多拥有 64 核心。其浮点及 SIMD 运算单元支持每拍 2 条 ARM Neon 128bits 浮点运算。华为鲲鹏系统支持 NUMA 内存架构，实现最多 4 个鲲鹏 920 互联和最高 256 个物理核的 NUMA 架构。

稀疏矩阵运算在科学计算、数据分析、机器学习等应用中十分常见，而稀疏矩阵频繁且离散地数据访存、任务之间存在着很强的依赖、需要细粒度的同步、任务之间负载不均衡等特点又给代码优化带来了巨大的挑战。

1.2 研究现状

有研究者 Saad[9]，以及 Saltz[10] 根据 SpTRSV 算法是一张任务依赖图2-4(b)的特点，提出了基于 level-sets 的方法，在预处理阶段构建一个任务依赖图，如图1-1(c)，之后就可以在 level 内部进行并行，在 level 之间设置同



(a) 下三角矩阵　　(b) 任务依赖图　　(c) level-sets

步屏障，一个 level 执行完了再执行下一个 level。

随后有研究人员基于 level-sets 算法，在不同的体系结构上进行了优化：CPU[11][12][13][14] 以及 GPU[15][16][17] 进行了优化。

基于 level-sets 的算法，存在着两个缺陷：第一，虽然可以在每个 level 上进行并行取得良好的并行度，但是需要在预处理阶段构建一个 TAG 图，计算每个任务的 level，以及处理任务之间任务负载均衡的问题，在预处理阶段会花费大量的时间。写出一个有良好扩展性的并行预处理算法同样具有挑战性，往往可能会导致预处理的时间远远大于并行计算任务图的时间，在计算任务图上获得的加速甚至不能抵消预处理阶段产生的时间消耗。第二，基于 level-sets 的的算法需要在每个 level 之间都要使用一个屏障确保进行同步，等待该 level 内的任务都执行完了再继续下个一个任务，在负载不均衡的情况下，这会产生大量的等待时间，随着核心数量的上升，任务依赖图 level 数量的增加，该同步方式的时间消耗也会随之上升。

面对以上两个 level-sets 的缺陷，有研究人员提出了以下的优化算法。

Jongsoo Park[12] 在基于 level-sets 算法进行了一些优化。他发现传统的基于 level-sets 的算法所使用的屏障式的同步方式会产生大量的开销。因此作者提出了一种 peer-to-peer 的同步方式。线程在任务执行完了之后不是等待其他线程执行到该屏障处之后再继续执行，而是会判断下一个任务的前置需求是否被完成，如果已经完成了，那么就继续运行下去。相比于基于屏障的同步方式，peer-to-peer 的同步方式具有更好的扩展性。

在减少同步所需消耗方面，作者还发现，在 SpTRSV 算法的任务依赖图中，大部分的依赖都是多余的，作者期望通过一步预处理操作来删除这些多余依赖。例如在 2→3→5 和 2→5 这种边存在的情况下，删除了 2→5 这条边。另一方面，作者发现在这些多余依赖当中，大部分都是两跳的（比如上述的 2→5 这条边），少部分是三跳或者三跳以上。为了减少预处理的计算量，作

者选择用粗略但快速的算法，只删除两跳的依赖边，而不是删除所有的多余依赖。减少了大约 90% 的多余依赖，具体效果如1-1。该算法运行在 12 核的 Xeon 处理器上，相比于传统的基于 level-sets 以及屏障式同步的算法获得了至少 1.6 倍的加速比。

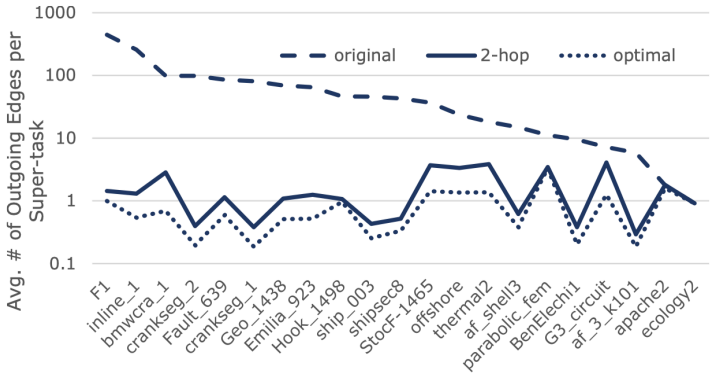


图 1-1　每个 super-task 的出度有了显著的减少

同时作者也做了负载均衡的优化，根据每行非零元个数的多少来进行任务的合并，组合成一定数量的 super-task，使得每个 super-task 有相似的计算量。

刘伟峰 [18, 19] 从减少算法预处理和减少算法同步所需时间的角度出发，提出了无需同步的（synchronization-free）SpTRSV 算法。基于传统 level-set 的方法，随着矩阵大小的增加，其在预处理阶段所需的消耗会剧烈增加，且同步所需的时间在计算总时间中的占比也会大量增加。他的算法在预处理阶段只计算了每个任务的 in-degree 数组（用来记录每一个节点需要有多少入度），而不是构造一个依赖图，大幅减少了预处理时间，而且该预处理方法能够轻易地使用多核来进行并行加速。对于每个 warp 使用自旋锁，不断判断前置依赖是否都被满足来决定该结点是否开始计算。当一行计算完成后，使用原子操作来“通知”后继节点，减少了同步的时间。同时作者结合 GPU 的体系结构，针对 GPU 的片上内存和全局内存进行了优化，该算法在 GPU TitanX 上比 Nvidia 提供的算法快 2-3 倍。

倪鸿、刘鑫 [20] 根据国产异构众核处理器 SW26010 体系结构的特点，针对非结构网络计算，提出了一种基于流水线串行-局部并行思想的通用众核 SpTRSV 优化方法。首先，为了减少预处理时间，根据核心的个数，将向量 x 平均划分为多个向量块。每个向量块 x 交给一个从核进行处理。每个从核的操作流程为：(1) 等待接受来自依赖向量块的数据，直到满足进入

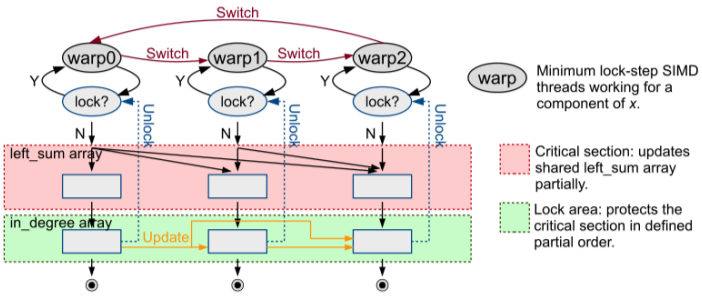


图 1-2　synchronization-free SpTRSV 算法的示意图

(2)。(2) 进行 x 的计算。(3) 发送计算得到的 x 给需要的核心。其次，为了解决 SW26010 寄存器通讯的局限性，作者还搭建了众核通信架构。同时还有数据压缩、LDM 缓冲、访存掩盖以及数据压缩等优化。

Wang, X., Xu, P.[21] 等研究人员通过分析总结现有的 SpTRSV 算法，给出了一个基于生产者和消费者的统一模型，且具有一定的泛化能力。针对不同体系结构的给出了编程指导性意见。同时利用这个模型，在 SW26010 上提出了快速的 SpTRSV 算法。

1.3 研究内容

本文基于鲲鹏 920 处理器进行 SpTRSV 算法的设计与优化。研究内容包括:

1. 设计一个无需预先构建任务依赖图，并且使用原子操作进行同步的 SpTRSV 算法。并且针对多核 CPU 架构的特点，以及 ARMv8 指令集进行优化。设计一个与华为鲲鹏平台的软硬件特性相适应的 SpTRSV 并行化方法。矩阵采用 CSC 压缩方式，将矩阵每行视作一个子任务每个任务有三种状态：等待依赖数据，计算，结束。多个核心在任务池中并行对子任务进行如下操作：a) 自旋判断当前任务前置依赖否被满足：首先在任务入口处，自旋判断当前任务的前置依赖是否已经被满足，如果被满足则进入计算阶段。b) 在计算阶段，算法对 x[i]，进行计算。c) 通过依赖该任务的算法，进行更新操作。
2. 结合鲲鹏 920 处理器的体系结构特点以及 ARMv8 指令集架构进行性能优化。主要包括针对缓存的优化；针对自旋等待性能的优化，以及针对原子指令的优化。
3. 针对 NUMA 架构进行优化。SpTRSV 是典型的 memory-bound 和

latency-bound 计算任务，利用 NUMA 架构可以显著提升内存带宽。目前对于 SpTRSV 算法，研究人员从任务划分、LDM 空间优化、负载均衡、自适应等角度进行了优化，取得了不错的成果。但是，缺少利用计算系统的 NUMA 特性进行优化的相关研究。我打算利用操作系统提供的 libnuma 编程接口，根据稀疏下三角矩阵本身是只读数据的特点，将下三角矩阵分为 4 个副本，分配到 4 个 NUMA 结点当中。

4. 在 SpTRSV 算法中在求解 x[i] 时，存在的向量乘法的操作，通过使用 SIMD 指令提升算法的性能。编译器支持自动向量化功能，其会自动利用 NEON 属性，编译时将代码向量化。启用自动向量化功能前需要打开相应的编译选项，且并非所有代码均可向量化，其需要符合一定的编码方式和规律，以提供更多的提示信息给编译器，进一步触发编译器进行代码的向量化；也可以使用 NEON intrinsic 函数进行显示的优化。NEON intrinsic 函数是一系列 C 函数调用，编译器可将其替换为适当的 NEON 指令或 NEON 指令序列。NEON intrinsic 函数几乎提供与编写 NEON 汇编指令相同的功能，但是将寄存器分配等工作留给编译器，以便开发人员可以专注于算法开发。与使用 NEON 汇编指令编码相比，NEON intrinsic 方式的代码有更好的可维护性。Arm 编译器、GCC 和 LLVM 编译器都支持 NEON intrinsic
5. 通过查阅相关文献搜集测试用的矩阵，进行正确性的验证，确保算法的正确性。结合性能分析工具，分析计算的特点，对算法进行性能的调优。最后，对并行效率和算法的可扩展性进行分析。

1.4 本文构成

本文基于华为鲲鹏 920 处理以及 ARMv8 架构，设计高效的并行 SpTRSV 算法，并结合计算系统体系结构的特点进行针对性地优化。本文的组织结构如下：

第一章绪论，主要介绍了课题背景，研究现状，研究内容，以及本文构成。在课题背景当中，我介绍了稀疏下三角矩阵求解的研究背景、应用场景及其重要意义。在研究现状当中，分析回顾了现有国内玩研究基础、研究现状。在研究内容中，我概括性地说明研究的重点内容，及其创新特色。

第二章介绍了相关理论与技术介绍，这里我提到了稀疏线性方程组求解常用的方法，其中稀疏下三角矩阵的求解效率的优化对直接法和不完全乔列

斯基共轭梯度下降法求解线性方程组具有重要意义。接着介绍了三种常用的稀疏矩阵压缩方式，以及我为什么选用 CSC 压缩方法的原因。然后我介绍了现存的 SpTRSV 算法，包括两种基于不同压缩格式的 SpTRSV 算法，以及在 GPU 和 CPU 上目前性能较好的 SpTRSV 算法。在第二章中，我还介绍了我所使用的优化技术及其原理，为后面算法设计打下理论基础。

第三章介绍了 SpTRSV 算法的设计与实现，包括如何得到下三角稀疏矩阵，如何进行正确性验证。然后我介绍了设计的 SpTRSV 算法和优化方法的一些具体细节，及其实现方式。

第四章我提出了算法的总体性能，一些优化方法对性能提升的效果，以及结合理论得出的分析与思考。

第五章总结与展望，对 SpTRSV 算法进行了总结与分析，指出了该算法的不足之处；在未来展望部分，我提出了一些仍有希望的优化思路。

第 二 章　相关理论与技术介绍

2.1 稀疏线性方程组求解

经典的求解线性方程组的方法一般分为两类：直接法和迭代法。前者例如高斯消元法, LU 分解等，后者的例子包括共轭梯度法等。直接法指在不考虑计算舍入误差的情况下，通过包括矩阵分解和三角方程组求解等有限步的操作求得方程组的精确解，因此又称精确法; 迭代法指给定一个初始解向量，通过一定的计算构造一个向量列 (一般通过逐次迭代得到一系列逼近精确值的近似解)，向量列的极限为方程组理论上的精确解。

2.2 稀疏矩阵的压缩方式

存储矩阵的一般方法是采用二维数组，其优点是可以随机地访问每一个元素，因而能够容易实现矩阵的各种运算。对于稀疏矩阵，它通常具有很大的维度，有时甚大到整个矩阵（零元素）占用了绝大部分内存采用二维数组的存储方法既浪费大量的存储单元来存放零元素，又要在运算中浪费大量的时间来进行零元素的无效运算。因此必须考虑对稀疏矩阵进行压缩存储（只存储非零元素)。

2.2.1 COO Format

最简单的稀疏矩阵的存储格式称为 coordinate(COO) format，这个形式只保留哪些非 0 的值，分别使用三个数组：val, rolIdx, colIdx 来对应存储数值、行号以及列号，这三个数组的大小都为矩阵非零元的个数 NNZ。举例说明，

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 2 & 4 & 0 & 6 & 0 \\ 3 & 0 & 5 & 0 & 0 \\ 4 & 7 & 0 & 8 & 9 \end{pmatrix}$$

该矩阵在 COO 存储格式下，三个数组分别为：

COO Format

```
1: val = {1, 2, 4, 6, 3, 5, 4, 7, 8, 9}
2: rolIdx = {0, 1, 1, 1, 2, 2, 3, 3, 3, 3}
```

```
3: colIdx = {0, 0, 1, 3, 0, 2, 0, 1, 3, 4}
```

2.2.2 CSR Format

CSR（Compressed Sparse Row）是一种按行压缩的矩阵存储形式。分别使用三个数组：val, rowPtr, colIdx 来对应存储数值、行指针、以及列号。其中 val 和 colIdx 的数组大小为非零元的个数 NNZ，而 rowPtr 数组的大小一般为 m+1，其中 m 为矩阵行的个数。例如图2-1。

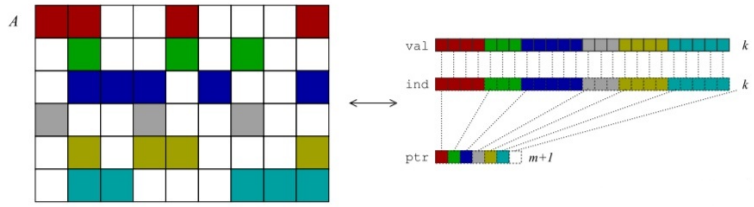


图 2-1　CSR 矩阵存储格式

2.2.3 CSC Format

CSC（Compressed Sparse Column）与 CSR 相似，是一种按列压缩的矩阵存储格式。分别使用是那个数组：val, colPtr, rowIdx 来对应存储数值、列指针、以及行号。其中 val 和 rowIdx 的数组大小为非零元的个数 NNZ，而 colPtr 数组的大小一般为 n+1，其中 n 为矩阵列的个数。例如图2-2

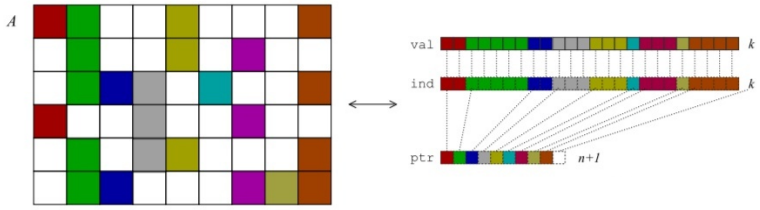


图 2-2　CSC 矩阵存储格式

2.3 现存 SpTRSV 算法

2.3.1 基于 CSR 格式的 SpTRSV 串行算法

Algorithm 2.1: CSR Based Serial SpTRSV

```
1 MALLOC(*left_sum, n) ;
2 MEMSET(*left_sum, 0) ;
3 for i = 0; i < n; i++ do
4 |   for j = row_ptr[i] ; j < row_ptr[i + 1] - 1 ; j++ do
5 |   |   left_sum[i] ← left_sum[i] + val[j] ∗ x[col_idx[j]] ;
6 |   end
7 |   x[i] ← (b[i] - left_sum[i]) / val[col_ptr[i]] ;
8 end
```

基于 CSR 格式的 SpTRSV 串行算法第一个循环从矩阵的第 0 行开始遍历这个矩阵。在第二个循环进行向量乘法的操作，求出 left_sum[i]。之后再求出未知数 x[i]。

2.3.2 基于 CSC 格式的 SpTRSV 串行算法

Algorithm 2.2: CSC Based Serial SpTRSV

```
1 MALLOC(*left_sum, n) ;
2 MEMSET(*left_sum, 0) ;
3 for i = 0; i < n; i++ do
4 |   x[i] ← (b[i] - left_sum[i]) / val[col_ptr[i]] ;
5 |   for j = col_ptr[i] + 1 ; j < col_ptr[i + 1] ; j++ do
6 |   |   left_sum[rowIdx[j]] ← left_sum[row_idx[j]] + val[j] ∗ x[i] ;
7 |   end
8 end
```

该算法的第一个循环从矩阵的第 0 行开始遍历这个矩阵。首先求出该算该行未知数 $x[i]$，接着遍历该列的上的非零元，对该列非零元所对应的 $left\_sum$ 进行更新，$left\_sum[rowIdx[j]] += val[j] \times x[i]$。

基于 CSR 压缩格式 SpTRSV 算法以及基于 CSC 压缩格式的 SpTRSV 算法，两者在访存模式上存在着一定的差异。基于 CSR 格式的算法需要进行频繁离散读取未知数向量 x[col_idx[j]]，相反基于 CSC 格式的 SpTRSV 算法只需要读取当前行号 i 所对应的 x[i]。在写数据方面，基于 CSR 的并行算

法需要集中的写 left_sum[i] 的数据，而基于 CSC 的并行算法则是离散的写 left_sum[rowIdx[j]]，在多核运算的条件下，离散的写比集中的写同一位置的数据更容易进行并行化。

2.3.3 基于 level-sets 的 SpTRSV 并行算法

J.Park 在 level-sets 算法的基础上提出除了更高效的并行 SpTRSV 算法[12]，该算法主要分为两个阶段：预处理阶段和计算阶段。

在预处理阶段主要有三个处理步骤：

1. 使用由 Chhugani[22] 提出的高度优化的并行广度优先搜索算法（BFS），构建了一个任务依赖图2-4(b)。
2. 在任务依赖图的基础上，作者又通过将同 level 的任务合并为 super-task 的形式实现负载均衡，确保每个线程被分配有相似计算量的 super-task，这种合并的操作有进一步减少了任务之间的依赖个数，进一步减少了同步所需的消耗。
3. 作者发现任务依赖图中存在着大量的多余依赖边，并开发了一个高效的过滤算法，去除了两条的依赖边。

在计算阶段作者使用了与传统 level-sets 算法不同的调度方式，作者使用 peer-to-peer 的同步方式替换为了在 level 之间使用屏障2.3进行同步的方式。下面是两种不同同步方式的伪代码：算法2.3、算法2.4

Algorithm 2.3: Level-scheduling with barrier

```
foreach level l do
    solve the unknowns of task_l^t ;
    barrier;
end
```

Algorithm 2.4: Level-scheduling with peer-to-peer synchronization

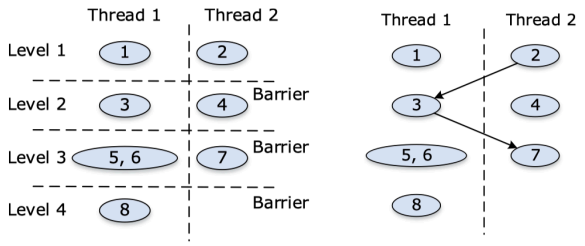
```
foreach level l do
    wait until all the parents are done wait ;
    solve the unknowns of task_l^t ;
    done[task_l^t] ;
end
```

矩阵名称	预处理时间（ms）	SpTRSV 计算阶段花费时间	SpTRSV 计算阶段时间分解		level 的个数
			同步时间消耗	计算时间消耗	
FEM/ship 003	92.46	12.95	10.96	1.99	4367
FEM/Cantilever	47.89	9.60	5.62	3.98	2397
chipcool0	8.74	1.99	1.15	0.84	534
nlpkkt160	484.67	38.30	0.01	38.29	2

表 2-1　基于 level-sets 算法 [12] 的任务耗时分解图

对比两种不同的调度算法，可以发现使用 peer-to-peer 的调度方式，能够使得计算资源能够得到更好的利用。因为当一个线程完成任务计算之后，基于屏障的同步方式会让该线程进入等待的状态。而基于 peer-to-peer 的调度方式则会让该线程继续进行下去。如图2-3，如果采用基于 peer-to-peer 的同步方式，线程 2 在完成任务 2 之后可以不用等在线程 1 完成任务 1，可以接着运行下去。这在一定程度上也有利于负载均衡。

图 2-3　两种不同同步方式的对比示意图



(a) 通过屏障同步　　(b) 点对点同步

2.3.4 sync-free 的 SpTRSV 算法

基于 level-sets 的算法需要花费大量的时间在预处理和任务间的同步上，运行 J.Park 的算法并分步进行统计（预处理阶段，同步所需耗时，计算阶段），我们就可以得到表2-1。所用的稀疏矩阵来自于佛罗里达大学的稀疏矩阵集合 [23]，观察这张表我们可以主要得到以下几点信息：

1. 预处理阶段所消耗的时间，要远大于计算阶段所消耗的时间，平均大概是 8 至 10 倍左右。预处理占总时间大约 80%。
2. 同步所需的开销会导致的时间消耗，会随着矩阵 level 个数的增加而剧烈增加，当稀疏矩阵的 level 较多时，会导致同步的时间消耗远大于计算的时间消耗。这也可能是同 level 间负载不均衡所导致的。

例如 nlpkkt160 只有两个 level，同步所占用的时间要远小于计算所需的时间。作为对比矩阵 FEM/ship 003，虽然矩阵的规模小于 lpkkt160，但是由于其 level 相对较多，导致了同步所占用的时间也较多。

刘伟锋从减少预处理和减少同步所需时间的角度出发，提出了名为 sync-free 的 SpTRSV 计算方法 [18]。该算法能够在 GPU 上能够取得较好的效果。与传统基于 level-sets 的算法不同的是，sync-free 的 SpTRSV 算法只在预处理阶段计算每个任务的入度，该入度表示当前任务有多少的前置依赖。作者使用一个 warp，32 个线程来处理每个任务。任务之间通过使用共享变量以及原子操作的方式，来决定任务之间的顺序。算法的伪代码见2.5refsync-free alogrithm solve

Algorithm 2.5: sync-free SpTRSV 预处理阶段算法

```
malloc(*d_left_sum, *s_left_sum, *d_in_degree, *s_in_degree, n);
memset(*d_left_sum, *s_left_sum, *d_in_degree, *s_in_degree, 0);
Function Preprocess() is
    Parallel for i = 0; i < nnz; i++ do
        atomic-add(&d_in_degree[row_idx[i]], 1) ;
    end
end
```

Algorithm 2.6: sync-free SpTRSV 计算阶段的算法

```
Function Solve() is
    Parallel for int i = 0; i < n; i++ do using one warp for one entry
        while s_in_degree[i] + 1 ≠ d_in_degree[i] do test dependencies
            // busy wait ;
        end
        x[i] ← (b[i] − d_left_sum[i] − s_left_sum[i])/val[colptr[i]];
        Parallel for j in [col_ptr[i] + 1], col_ptr[i + 1] − 1 do one thread for
         one nonzero
            if in local then
                atomic-add(&s_left_sum[rid], val[j] × x[i]) ;
                atomic-add(&s_in_degree[rid], 1) ;
            else
                atomic-add(&s_left_sum[rid], val[j] × x[i]) ;
                atomic-add(&s_in_degree[rid], 1) ;
            end
        end
    end
end
```

2.4 基于缓存的优化技术

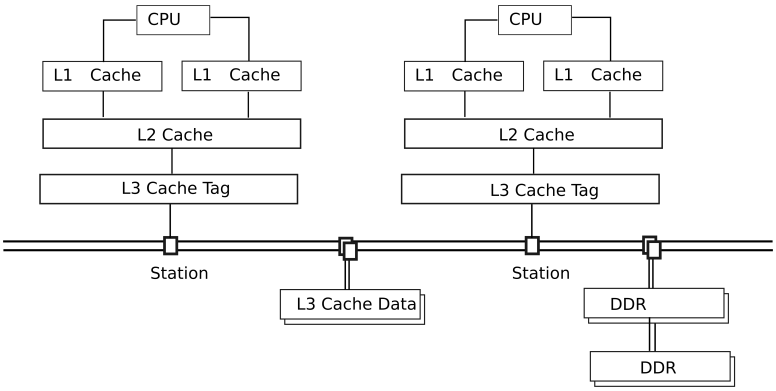


图 2-4 kunpeng cache 示意图

2.4.1 内存 cache

Cache 在计算机系统中，是一种基于分层优化思想的一种优化手段。由于 CPU 的速度远大于内存的速度，出于成本的考虑，人们常用相对告诉的 Cache 来弥补两者之间的差距，提升 CPU 的性能。在鲲鹏 920 芯片中，每个核心独占 L1、L2 两级缓存。离核心最近的是两个 L1 Cache，分开缓存指令和数据，大小分别为 64K。L2 Cache 则要大得多，大小为 512k。而第三级缓存则是鲲鹏 920 芯片上的所有核心共享，其中 L3 Cacheline 的长度为 128 个字节。

2.4.2 Cache Prefetch 优化技术

Cache Prefetch 也是一个针对 Cache 的优化设计。Cache 比实际的内存快很多，所以如果我们可以提前加载部分内存到 Cache 中，就会在性能上有优势。比如一个乘法的时间，就算不考虑流水线，也不过 3-12 个周期，但是从 DDR 内存中读取数据可能会花费 100 个始终周期。所以最好还是在计算开始前，先让数据从内存加载到缓存当中，做到计算和缓存同步进行。

例如，如下代码:

```
1:     for (i=0; i<W; i++) {
2:         for(j=0; j<W; j++) {
3:           c[i][j] = 0;
4:           if (!(j%INT_PER_CACHELINE))
5:                   __builtin_prefetch((const void *)&c[i][j+
                        INT_PER_CACHELINE], 1, 3);
```

```
6:            for (k=0; k<H; k++) {
7:                    c[i][j] += a[i][k]*b[k][j];
8:                    if (!j && !(k%INT_PER_CACHELINE))
9:                            __builtin_prefetch((const void *)&
                                 a[i][k+INT_PER_CACHELINE], 0,
                                 3);
10:                   if (!i && !(j%INT_PER_CACHELINE))
11:                           __builtin_prefetch((const void *)&
                                 b[k][j+INT_PER_CACHELINE], 0,
                                 3);
12:           }
13:         }
14:   }
```

这里的 ___builtin_prefetch 是 gcc 的内置函数，在不同的平台有不同的封装。在上面的算法中，我们进行某个向量单元的计算的时候，已经可以知道后面要计算的内存单元是什么了，我们就可以提前把数据取出来。除了软件在做 prefetch 外，鲲鹏 920 处理器中也有硬件实现的 prefetch 单元，如2-5。这就导致软件层面的 prefetch 可以没有很好的效果。

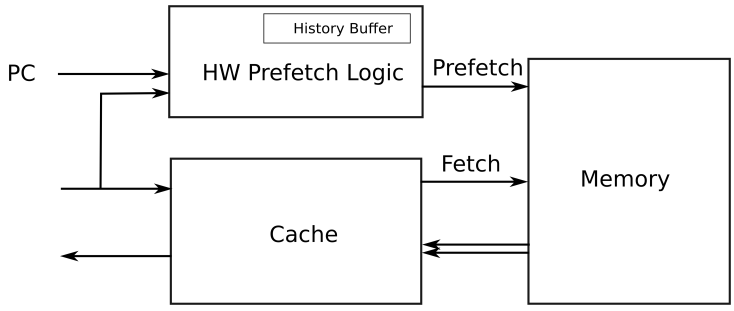


图 2-5　鲲鹏 920 cache prefetch 的硬件结构示意图

2.4.3 缓存一致性协议

在多核 CPU 的情况下，由于每个核心都有自己的独占的缓存，数据在这些缓存中制造了多个备份，这就造成了，到底哪份有效的问题。鲲鹏 920 采用的是监听一致性协议（Snooping），该协议是通过总线广播的形式实现的。最为经典的监听一致性协议是 MESI，该协议由 James Goodman 提出，目前已在 x86、ARM、Power 架构中得到实现。

MESI 协议将 CacheLine 分为四个状态

1. Modified（M）：该状态表示，此 CacheLine 有效，数据只存在本 cache 中，且 CacheLine 中的数据被修改和内存中的不一样。在这种状态下，如果监听到有其他芯片读取缓存行对应主存的操作，该操作就会被延迟：首先将该操作写回，然后本 CacheLine 的状态就会变为 S。
2. exclusive（E）：该状态表示，此 CacheLine 有效，数据只存在于本 cache 中，数据和 Cache 中的数据和对应内存中的数据一致。如果监听到总线中其他线程对该行的读取操作，状态将变为 S。
3. Shared（E）：该状态表示，此 CacheLine 有效，数据存在于多个 Cache 中。如果监听到有其他线程的修改或独享请求，那么改缓存行就变为无效
4. Invalid（I）：该状态表示，此 CacheLine 无效，如果有线程对该数据进行读取，就触发 cache miss，重新从内存中读取对应的数据。

此外还有一种基于目录是的 MESIF 缓存一致性协议，这里就不详细展开，详见 [24]。

总线本质上是一个去中心化的系统，随着核心数量的增加，维护缓存一致性的成本也会增加。例如，假设 32 个线程同时在一个变量上做自旋锁的操作，此时如果有一个核心更新了一次 spinlock，那么就要通知另外 31 个线程。如果总线上发生了冲突，会导致计算性能进一步下降。

2.4.4 伪共享

缓存一致性协议以及数据按照 CacheLine 大小读取到 Cache，这两个特点造成了在多核计算中经常出现的问题：伪共享（false sharing）。

```
1:    #omp parallel for num_threads(10)
2:    for(int i = 0; i < 10; i++){
3:        for(int j = 0; j < 10; i++){
4:            sum[i] += i * j;
5:        }
6:    }
```

例如上述代码，在鲲鹏 920 中，CacheLine 的大小为 128 字节，sum 数组可能会被分配掉同一个 CacheLine 中，这种情况下，会导致算法的并行度无法上升。因为一个核心的写操作都会导致其他核心的缓存失效。当其他线

程需要读写该区域内存的时候需要首先将修改过的 CacheLine 写回，然后重新从内存中读取。

通常可以通过 padding 的方式来解决伪共享的问题，如图2-6。对于多线程频繁读写的数据，通过 padding，将不想管的数据分配到不同的 CacheLine 当中。

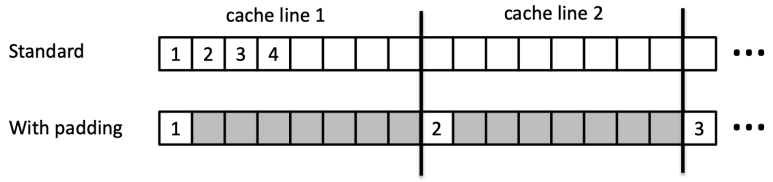


图 2-6　通过 padding 的方式解决伪共享导致的问题

2.5 ARM Neon 指令

鲲鹏 920 处理器基于 ARMv8，支持 128bits 的 SIMD 指令。它通过在 64bits 的 D 寄存器和 128bits 的 Q 寄存器上进行向量化的操作，来实现单指令多数据运算。鲲鹏 920 处理器一共有 16 个 Q 寄存器和 32 个 D 寄存器。图2-7显示了指令 $VADD.I16Q0, Q1, Q2$ 同时进行了 8 路 16bit 整数的加法操作，相比于传统的加法指令有显著的性能提升。ARM Neon 除了提供汇编

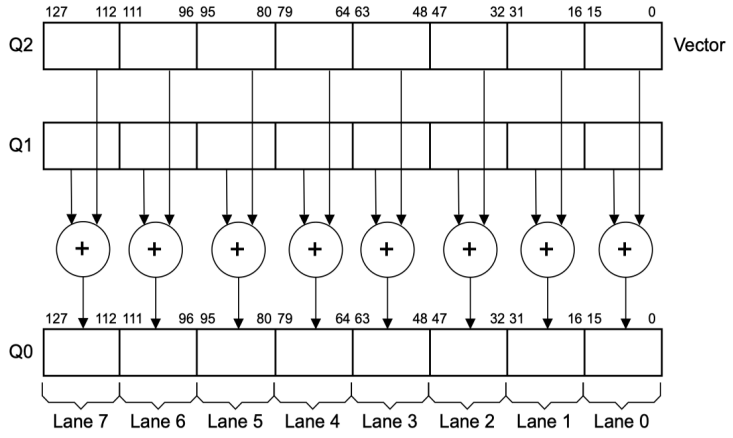


图 2-7　8 路 16 位加法操作

指令之外，还提供了 C 风格的编程接口 Neon intrinsics。相比与 Neon 汇编指令，使用 Neon intrinsics 可以不用考虑寄存器的分配问题，这部分由编译器来实现，用户只需要关注上层代码。同时，也能享受编译器带来的一些优化。Neon intrinsics 了丰富的编程接口，包括：对数据进行的 load 与 store，

各种常用的运算，变量常量的创建等。Neon intrinsics 支持 GCC 编译器，使用时只需 include 头文件 arm_neon.h。

2.6 NUMA 架构

传统多核处理器通常采用 SMP（Symmetric Multi-Processing，对称多处理器）2-8，在该架构下，每个线程的地位都是均等的，对内存使用的延迟与带宽也相同。在操作系统的支持下能够做到很好的负载均衡。鲲鹏处理器支持 NUMA（Non-uniform memory access, 非统一内存访问）架构，如图2-9。使用这种架构能够解决处理器核数限制的问题，同时，通过合理的软件设计，能够提升内存的带宽，解决 SMP 架构下总线瓶颈的问题。

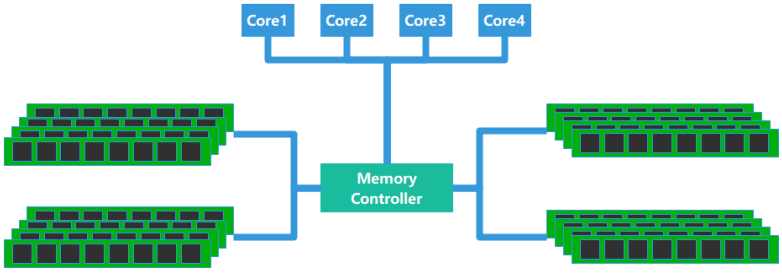


图 2-8　SMP 架构示意图

由于在 NUMA 架构下，多个核心组成一个节点（Node），系统中可以存在着多个节点，每个节点上有一个内存控制器，管理控制一片内存。内存在物理上是分布式的，通过片间网络互联。这就导致了，每个线程访问内存的延迟与带宽取决于软件相对于内存的位置，对软件设计者提出了挑战。

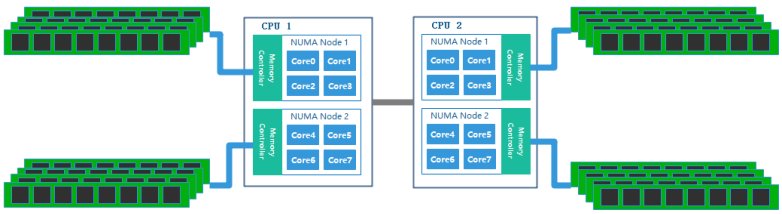


图 2-9　NUMA 架构示意图

linux 从内核版本 2.5 开始就支持 NUMA 架构 [25]，用户可以通过 numactl 命令来手动分配程序的 NUMA 访问策略，例如我们可以将进程或者内存固定在指定的节点上。系统默认的 NUMA 策略是 nodelocal，及总是在线程执行的地方分配内存。用户可以设置为 numactl –interleave，这时操作系统将会在各个节点之间，使用 round-robin 算法均衡分配内存。同时系统

也为用户提供了 libnuma 这一编程接口 [26]，方便用户在程序语言层面，对数据进行操作，例如手动指定数据所在的节点。

值得一提的是，默认 NUMA 策略下，操作系统使用的是“Fist touch”的 NUMA 分配原则，即用户使用 malloc 时操作系统会分配逻辑内存页，当初次访问这个数据时（例如初始化操作），操作系统发现该逻辑页面还没有被分配物理页面，于是就进行分配物理页面的操作，此时操作系统的根据调用线程的 NUMA 分配策略来进行物理页面的分配。这时，在默认 NUMA 分配政策下，操作系统会将数据分配到，离该线程最接近的 NUMA 节点。当然也可以开启 numactl –interleave，来替换覆盖“First touch”的原则。

常用的编程工具 OpenMP 也提供了 NUMA 编程的接口。主要是通过两个环境变量，OMP_PLACES 决定了线程能够创建的位置，可以指定具体的节点，也可指定具体的核心。OMP_PROC_BIND 可以设置线程的分配方式。

2.7 ARMv8 原子指令优化

原子操作是多核程序中一个很重要的操作。x86 通常使用锁总线的方式实现原子指令，保证了原子指令的执行期间不会收到其他指令的干扰。

在 ARMv8.1 发布之前，ARM 处理器实现原子操作的方式是 ll/sc(Load-Link/Store-Conditional)，具体指令就是 LDREX/STREX。

具体原理为，当 CPU-A 调用 LOADEX 命令的时候，会将数据从内存中读取到缓存当中，并将该缓存的状态标记为 EX（Exclusive），此时如果还有一个 CPU-B 也使用了 LOADEX 命令，读取同一个数据，这时 CPU-B 中的标记变成了 EX，而 CPU-A 中缓存行的标记则变成了 S。在 CPU-A 执行完了之后如果要调用 STREX 指令，将数据写回内存，它首先检查标记的情况，发现自己没了 EX，于是写回失败。重新执行 LOADEX，操作数据，STREX 这样的循环。而 CPU-B 则可以成功进行 STREX，然后放弃 EX 标记。CPU-A 不断处于循环状态，这样就会导致性能的损失。

随着 ARM 处理器核心数量的提升，原先 LL/SC 方式的原子操作通过争抢的方式获得原子操作会对性能造成很大的影响。因此，为了支持这种大型系统，在其 ARMv8.1 规范中引入了 LSE (Large System Extensions)，加入了大量原生原子操作指令 [27]，大致原理就是将原子操作放到存储端去做，从而提升多核计算性能，理论上在多核系统的条件下，LSE 指令的性能要好与 ll/sc 原子指令。新的扩展指令包含 CAS, SWP 和 LD/ST<OP> 等，其中

<op> 为 ADD，CLR，SET 等。

gcc 也充分利用了这一特性，提供了基于 LSE 原子指令的优化。

CODE 2.1 基于 LL/SC 的原子指令

```
1:  __LL_SC_PREFIX(arch_atomic_##op(int i, atomic_t *v))

2:  {
3:      unsigned long tmp;
4:      int result;
5:      asm volatile("// atomic_" #op "\n"
6:  "   prfm    pstl1strm, %2\n"
7:  "1: ldxr    %w0, %2\n"
8:  "   " #asm_op " %w0, %w0, %w3\n"
9:  "   stxr    %w1, %w0, %2\n"
10: "   cbnz    %w1, 1b"
11:     "=&r" (result), "=&r" (tmp), "+Q" (v->counter)
12:     : "Ir" (i));
13: }
```

CODE 2.2 基于 LSE 的原子指令

```
1:  static inline void arch_atomic_##op(int i, atomic_t *v)

2:  {
3:      register int w0 asm ("w0") = i;
4:      register atomic_t *x1 asm ("x1") = v;
5:
6:      asm volatile(ARM64_LSE_ATOMIC_INSN(__LL_SC_ATOMIC(op
            ),
7:  "   " #asm_op " %w[i], %[v]\n")
8:      : [i] "+r" (w0), [v] "+Q" (v->counter)
9:      : "r" (x1)
10:     : "x16", "x17", "x30");
11: }
```

对比上述两个不同的实现方式，原来基于 LL/SC 实现的原子操作，需要先 LDXR，然后执行 STXR 并查看是否成功，如果不成功那就重新循环执行，直到成功。而基于 LSE 原子指令的实现则更加简洁，如果是 atomic_add 操作只需一条 LDADD 指令。

2.8 CPU 松弛技术

在自旋等待中，常常会使用到 CPU_relax() 进行优化。也就是在自旋的循环体重插入 nop 操作，让 cpu 进行空转。在核并行的情况下，这种操作能够减轻为了维护内存序带来的性能损失。intel 处理器从 Pentium4 开始引入了 PAUSE 汇编指令。在 ARM 架构下也能通过汇编实现类似的操作。memory 指令是内存屏障的操作，保持 CPU 空转，在运行时防止对内存的操作乱序执行，并且会将所有缓存在寄存器中的所有变量写回内存当中，然后重新从内存中读取。

```
1:    \label{test}
2:    inline void PauseCPU() {
3:        __asm__ __volatile__("yield" : : : "memory");
4:    }
```

2.9 本章小结

本章介绍了 SpTRSV 算法的应用场景，在线性方程求解算法中，LU 分解和预优化的共轭梯度下降法都需要用到稀疏下三角矩阵求解；然后介绍了稀疏矩阵常见的两种几种压缩方式 COO、CSC、CSR；之后又详细介绍了目前现有的性能不错的 SpTRSV 算法及其性能分析。最后则介绍了，基于多核 CPU 体系结构进行优化的一些手段，以及这些优化技术的原理，例如缓存优化技术，伪共享的消除机制，基于 ARM Neon 的 SIMD 指令，已经针对 NUMA 架构的优化。最后还提到了使用 CPU 松弛技术来提高多核并发情况下自旋等待的性能。

第 三 章　SpTRSV 算法的设计与实现

3.1 实现并行的 SpTRSV 的算法

3.1.1 一些准备操作

该模块包含稀疏矩阵的读入。以及从矩阵中获得下三角矩阵。假设稀疏下三角矩阵求解的向量 x_ref，每个元素的值都为 1.0。然后使用矩阵向量乘法，得到等式 $Lx = b$ 右边的结果向量 b。在 SpTRSV 算法并行求解之后，得到待求解的向量 x，通过 x 与 x_ref 对比，来验证结果的正确性。

3.1.2 实现多核架构下的并行 SpTRSV 算法

本算法基于 CSC 压缩格式的矩阵，将一列矩阵的计算抽象为一个任务。在预处理阶段同样只需统计每个任务的前置依赖，见伪代码3.1。该算法使用了一个并行的循环，拥有很好的扩展性，占总体运行的时间可以忽略不记。

Algorithm 3.1: 并行 SpTRSV 算法预处理阶段伪代码

```
1 malloc(*left_sum, *in_degree, n);
2 memset(*left_sum, *in_degree, 0);
3 Function Preprocess() is
4 |   Parallel for i = 0; i < nnz; i++ do
5 |   |   atomic-add(&in_degree[row_idx[i]], 1) ;
6 |   end
7 end
```

在计算阶段，多个线程从索引 0 开始依次分配，进行并行计算。首先通过一个自旋等待来判断前置条件是否满足，该自旋等待使用 CPU 松弛的技术进行优化，通过一定量的空转，减少了处理器为了维护内存序带来的性能损失。当前置条件满足时，首先会进行向量 x[i] 的计算。接着需要计算后继任务 j 所对应的 left_sum[j] 和 in[j]，后者用来表示后继任务有多少前置条件已经被满足。对于第二个循环，由于稀疏矩阵非零元分布特点的不同，在第二个循环可能出现负载不均衡的现象。通过打印第二个循环的任务个数，对于部分矩阵存在着严重的负载不均衡的现象，例如名为 webbase 的矩阵（该矩阵来自于佛罗里达大学的矩阵集合 [23]），该矩阵 93% 的任务在第二个循环当中只需计算 0 次，1.3% 的任务在第二个循环中需要计算 1 次，剩下任务的计算次数从 2 次到 28684 次分布，存在着严重的负载不均衡的现象。

sync-free 的算法使用一个 warp 来处理一个任务，而一个 wrap 有 32 个线程，对于哪些计算数量为 0 的任务，这会导致计算资源的大量浪费。因此我在任务进入计算阶段之前，先进行负载均衡的操作，详见3.3。

Algorithm 3.2: 并行 SpTRSV 算法计算阶段伪代码

```
1  Function Solve() is
2      Parallel for int i = 0; i < n; i++ do using one warp for one entry
3          int in_degree_this_task = in_degree[i] - 1 ;
4          prefetch(&val[col_ptr[i]]) ;
5          prefetch(&b[i]) ;
6          while in[i]≠ in_degree[i] do
7              Pause() ;
               /* cpu relax                                          */
8          end
9          x[i] ← (b[i] − left_sum[i])/val[colptr[i]];
10         for j in [col_ptr[i] + 1], col_ptr[i + 1] − 1 do one thread for one
            nonzero
11             atomic-add(&left_sum[rid], val[j] × x[i]) ;
12             atomic-add(&in[rid], 1) ;
13         end
14     end
15 end
```

在算法3.2的第 4 行，使用了 prefetch 的预取指令。在循环等待之前将稀疏矩阵非零元的值以及方程组等式右边的向量 b[] 提前缓存到 cache 中。

3.2 消除伪共享

由于对数组 left_sum 以及数组 in[rid] 会出现大量的并发访存，会出现伪共享的情况，应该通过 padding 的方式消除。这里选择使用 gcc 的编译属性 ___attribute___ 进行 padding。

CODE 3.1 消除伪共享

```
1: struct Node{
2:     std::atomic<size_t> csrRowHisto_atomic {0};
3:     std::size_t idx = {0};
4:     VALUE_TYPE left_sum {0.0};
5:     VALUE_TYPE xi {0.0};
6: }__attribute__((aligned(128)));
```

3.3 负载均衡

J.Park[12] 通过合并超级任务的方式来进行负载均衡。他将同一个 level 中的任务进行合并，使得每个线程所分得的超级任务有相似的计算量。这种做法适合在已经构建了任务依赖图之后的情况；并且在有些 level 会出现，任务数量小于线程数量，没法进行合并。出于这两点考虑，我选择通过分割大任务的方式来进行负载均衡。设定一个上限 upper_size，将超过上限的任务进行分割。创建多个线程，每个线程平均处理 m 个任务，如图3-1。

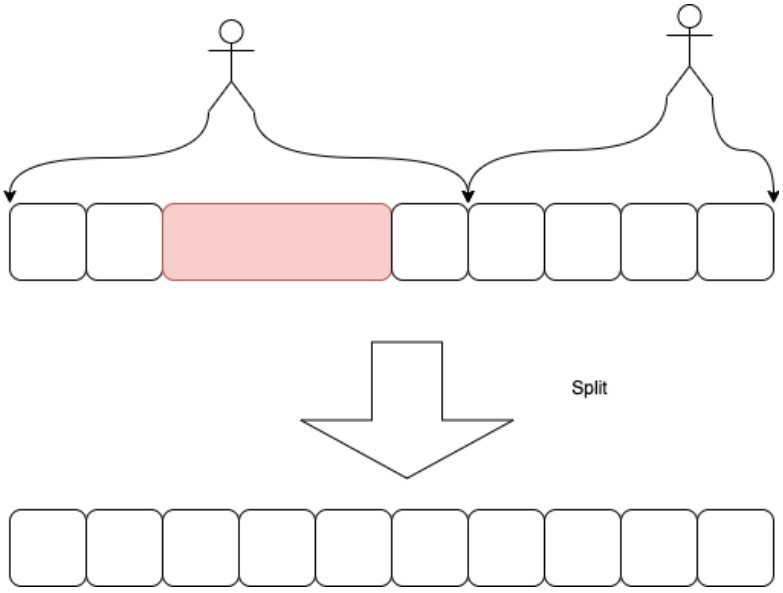


图 3-1　负载均衡示意图

在后续测试中发现，随着矩阵规模的扩大，负载均衡算法会遇到内存带宽的瓶颈，导致由负载均衡的消耗远大于使用负载均衡带来的性能优化。对于这个问题我提出了两种解决方案：一是利用 NUMA 架构的特性，提升内存带宽；而是在预处理阶段统计每个任务的计算任务负载情况的最大值和最小值。只对两者差距不是很大就不进行负载均衡。

3.4 使用 SIMD 指令

并行 SpTRSV 算法的第二个循环可以通过 SIMD 指令进行优化。由于 ARM Neon 只提供了 128bit 的向量寄存器，只能放下两个 double 和四个 float，这里只对 float 类型的数据进行向量化计算的操作。算法流程为：首先用数据类型 float32x_t 创建两个 Neon 寄存器变量 vec_xi、vec_csaVal，该数据类型表示一个向量中有 4 个 float32 数据，分别表示由行号为 i 的未知数 x 组成的向量和稀疏矩阵一列上非零元组成的向量。将原来步长为 1 的循

环，现在要修改为步长为 4 的循环。然后使用 vld1q_f32 指令将稀疏矩阵中的 4 个非零元 load 进入 vec_csaVal 寄存器当中，使用 vld1q_dup_f32，将 xi 复制 4 份，load 进 vec_xi 寄存器中。之后使用 vmulq_f32，执行向量乘法的操作。最后使用 vgetq_lane_f32(vec_result)，将向量乘法的结果从寄存器中取出，并写会内存当中。

在将结果写会内存的时候，由于一列上的非零元的行号都是离散的，因此对应产生结果的行号也不是连续的，这就导致我们无法通过一个 store 命令直接将结果写如 left_sum 中，而是需要分开进行独立的写操作。

CODE 3.2 SIMD 指令优化

```
1:     float32x4_t vec_xi;
2:     float32x4_t vec_csaVal;
3:     vec_xi = vld1q_dup_f32(&xi);
4:     for each nonzero in column (step_size = 4) {
5:         result[0] = vgetq_lane_f32(vec_csaVal, 0);
6:         result[1] = vgetq_lane_f32(vec_csaVal, 1);
7:         result[2] = vgetq_lane_f32(vec_csaVal, 2);
8:         result[3] = vgetq_lane_f32(vec_csaVal, 3);
9:         STORE result;
10:    }
```

3.5 使用 LSE 指令

由于我使用了大量的原子操作，例如对于 left_sum 和 in_degree 使用了很多的原子操作，使用原有争抢式的原子指令会对程序的性能产生消极影响。在开启 LSE 原子指令之前 CAS 操作需要通过 LDXR 和 STXR 并通过不断判断的方式实现，而在开启 LSE 只需一条 casal 指令，同样原子加法操作也只需一条 LDADD 指令。

具体实现方式为，通过编译器参数'-march=armv8-a+lse' 来实现

3.6 使用 NUMA 架构

NUMA 架构的优点是扩展了系统的内存带宽，提升了系统的 CPU 核心数目，缓解了缓存一致性的冲突。但是却带来了远程内存访问，延迟过高等缺陷。软件设计人员需要充分考虑并利用这个特性，否则会带来性能的损失。

通过 numactl –hardware 命令可以获得本机的 numa 硬件情况，本机使用的 NUMA 特性如图所示3-2。从该图 node distances 可以看出远程访问的

延迟可以为本地访问延迟的 1.6 至 3.3 倍。

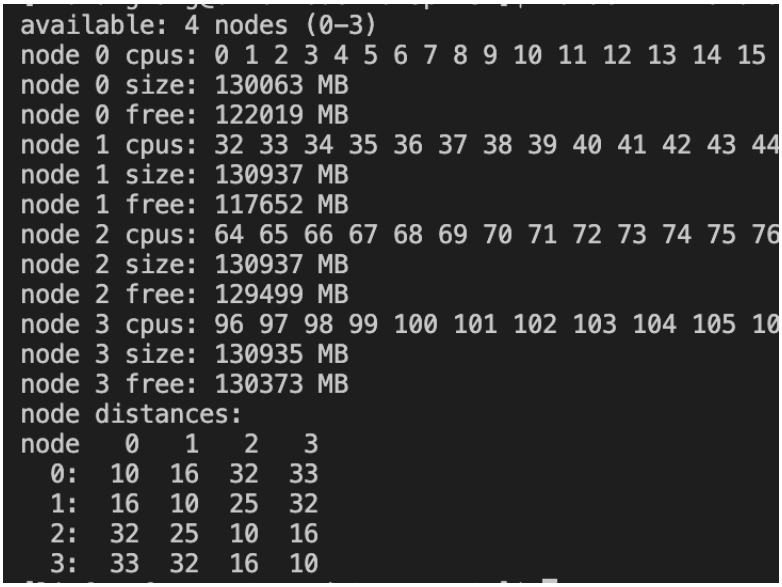


图 3-2 本机所使用的硬件特性

本算法使用的数据按照读写特征主要分为以下二类:

1. 只读数据类型: 稀疏的三个数组, cscCowPtr[]、cscRolIdx[]、cscVal[] 以及等式 $Lx = b$ 右边的向量 b[]。
2. 频繁通过原子操作读写: left_sum[]、in_degree[]、未知数向量 x[]

对于只读的数据类型我主要通过创建数据副本的形式, 将只读数据分成四分, 分配到四个 NUMA 节点上。主要是通过通过 Linux 操作系统提供的 C 语言编程接口 [26]libnuma 中的 void *numa_alloc_onnode(size_t size, int node), 接口参数 int node 表示指定结点的编号。线程首先通过 sched_getcpu() 和 numa_node_of_cpu(cpuid), 得到自己所在结点的信息, 根据这个信息来访问局部数据。然而对于频繁通过原子操作读写的数据, 由于数据读写的随机性, 没法使用 numa 架构进行很好的优化。

总的来说, 我主要尝试了四种种 NUMA 策略。

1. 使用 numactl –cpunodebind, 将所有任务都固定在一个结点上。
2. 使用 numactl –cpunodebind, 将任务固定在两个距离相近的 NUMA 节点上。
3. 使用 numactl –interleave, 将数据均匀分布在四个 numa 节点上。

4. 创建只读数据副本，并将其分配到 4 个结点上。

经过测试发现其实固定在一个节点上，性能是最好的。

3.7 本章总结

在本章中我介绍了本算法的设计以及优化思路。包括一些程序读入，数据整理以及结果验证的辅助操作，基本的并行架构及其在此基础上的优化技巧。主要的优化技巧有：通过 prefetch 指令进行预缓存；使用 CPU 松弛技术提升自旋等待的整体性能；使用 padding 的方法消除伪共享；在分析矩阵任务负载不均衡的基础上提出了负载均衡的策略；使用 SIMD 指令进行优化；通过编译指令的方式开启 LSE 指令集，优化了原子操作的性能；以及针对 NUMA 架构进行了一系列的策略尝试优化。

第 四 章　实验结果分析

4.1 算法总体性能测试

首先是算法整体性能测试，本次测试所用的矩阵主要来自于 [12] 和 [18] 这两片文章中所用到的矩阵，这些矩阵都可以从 MatrixMarket 和弗罗里达大学稀疏矩阵集合中下载。本次测试我所使用的矩阵如表4-1所示。表中最后一列并行性的定义为并行性 = 最大 level 数 ÷ 算法的行数，可以理解为平均每个 level 有多少的任务，其中并行性最好的是矩阵 nlpkkt160，只有两个 level，理论并行性最差的是 hollywood-2009 和 crankseg_1。

矩阵名称	行或列数量	非零元个数	最大 level 数	并行性
nlpkkt160	8,345,600	229,518,112	2	4,172,800
road_central	14,081,816	33,866,826	59	238,675
road_usa	23,947,347	57,708,624	77	311,004
webbase-1M	1,000,005	3,105,536	514	1,946
wiki-Talk	2,394,385	5,021,410	522	4,587
chipcool0	20,082	281,150	534	37
crankseg_1	52,804	10,614,210	4056	13

表 4-1　性能测试所用矩阵

这里我对比了不同平台的 SpTRSV 的算法性能，其中 intel 平台选用的是 MKL11.3 版本，CPU 为双路志强 E5-2695 v3。ARM 平台采用鲲鹏 920 处理器。在 GPU 平台采用了 sync-free 的 SpTRSV 算法，所用硬件为 TiTanX，结果如图4-1。

在图4-1中，我分贝测试了 6 个矩阵，通过 5 个柱状图的形式，以 GFlops 为单位显示了算法的性能，GLflops 计算方法为 $(2 \times NNZL) \div totoal\_time$。$totoal\_time = preprocess\_time + calculate\_time$。NNZL 表示下三角矩阵非零元的个数，为双精度浮点类型，所以前面乘 2。

对于可以看出，由于使用了并行性较好的矩阵如 nlpkkt160,road_central 和 road_usa 算法都取得了相比于串性算法 3 倍以上的加速比。而对于那些并行性较差的矩阵如 wiki-Talk 以及 chipcool0 算法 level 数量较多，能够获得加速比较差，但是仍然能够获得一定的加速。

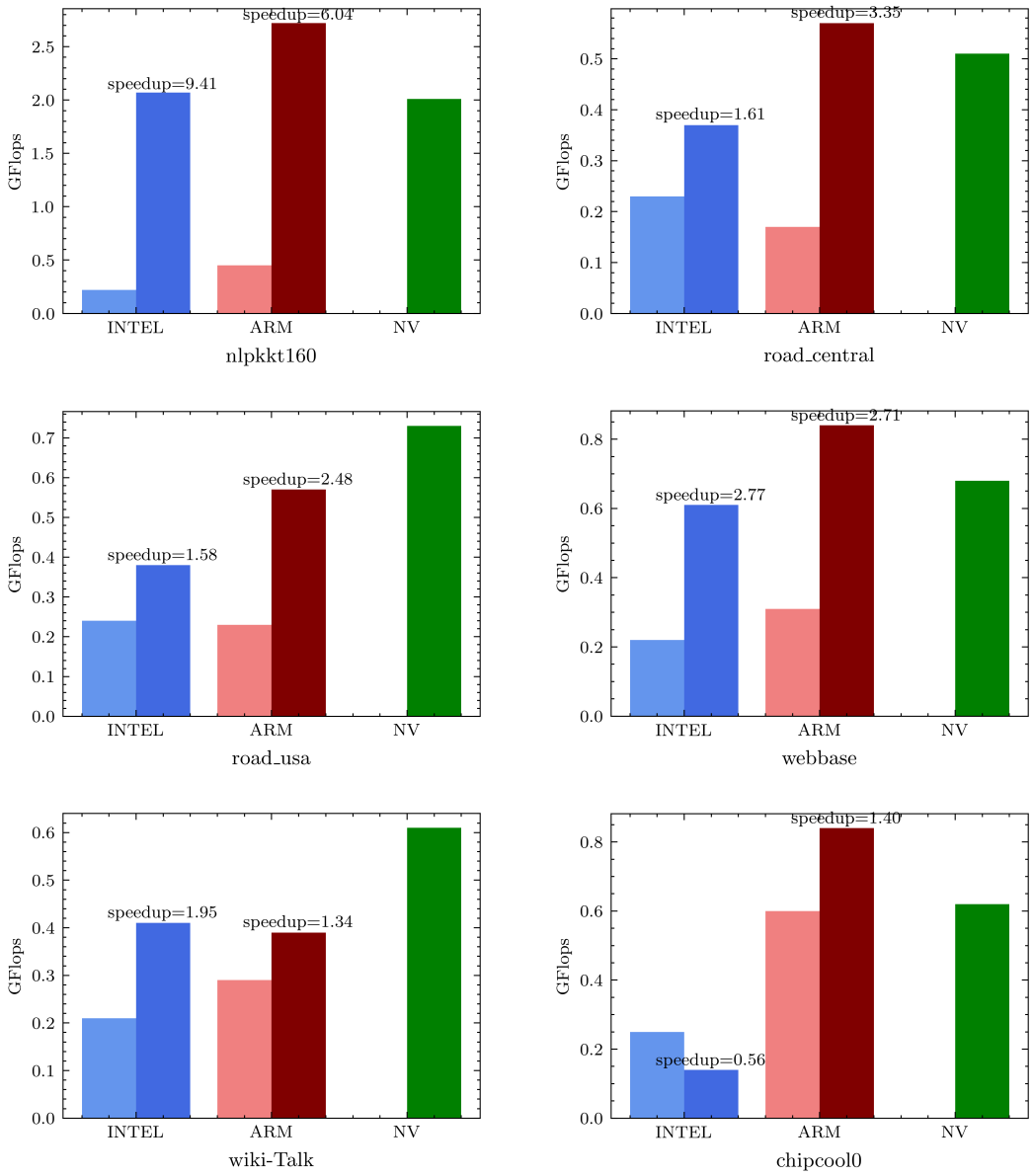


图 4-1　对比不同平台 SpTRSV 算法的结果示意图

矩阵名称	预处理时间	计算消耗时间	总时间（ms）	预处理时间占总时间的比例
nlpkkt160	18.70	68.68	87.38	21.4%
road_central	20.85	87.76	108.61	19.1%
road_usa	42.08	141.9	77	22.8%
webbase-1M	1.30	4.29	514	23.2%
wiki-Talk	3.82	13.91	522	21.5%
chipcool0	0.12	0.24	534	33.3

表 4-2　算法耗时分解表

如表4-2所示，算法在预处理阶段主要进行 in_degree 的计算，已经通过任务划分的方式进行负载均衡的处理。预处理时间占总时间的 20% 左右，且相对稳定。而且，相比于 J.Park[12] 基于 level-sets 的算法2-1，我的算法在预处理阶的时间消耗很少，例如对于矩阵 nlpkkt，基于 level-sets 的算法需要 484.07ms 的时间，而我的算法只需要 18.70ms 的预处理时间，以及 68.68ms 的计算时间。

4.2 测试负载均衡的影响

对于一些任务任务之间，负载相对均衡的矩阵如 nlpkkt160,road_central 和 road_usa，负载均衡优化的开启和关闭，并没有很大的影响，相反对于 webbase 和 chipcool，这两个矩阵之间存在这负载不均衡的想象，所以启用负载均衡起到了优化的作用。由于负载均衡所消耗的时间相对较小，所以并没有对整体的计算时间产生很大的影响。

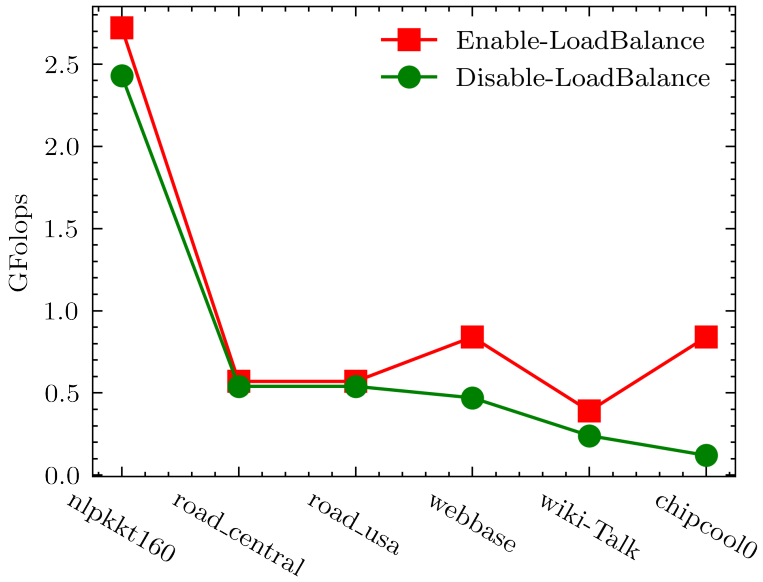


图 4-2　测试负载均衡对 SpTRSV 算法性能的影响

4.3 使用 CPU 松弛技术对性能的提升

CPU 松弛技术对性能地提升相对较低，对于一些任务间依赖相对比较严重的稀疏矩阵有一定的性能提升。

4.4 使用 ARMv8 原子指令对性能的提升

在使用了 ARMv8 原子指令之后，性能相比于 ARM 传统的原子指令，性能有所提升。

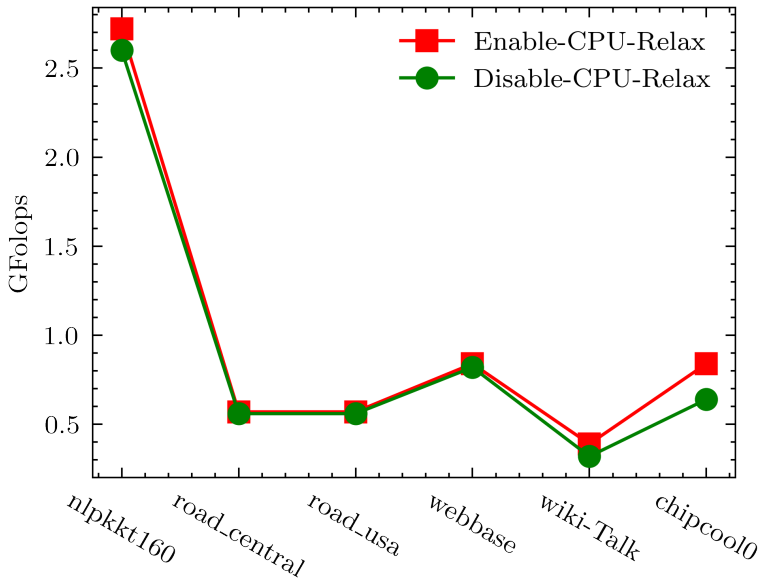


图 4-3　测试 CPU 松弛技术对性能的影响

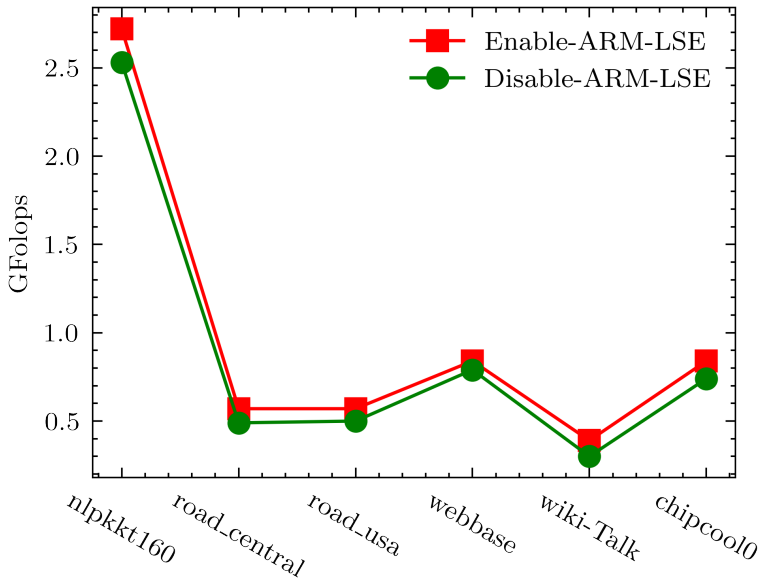


图 4-4　测试 ARMv8 原子指令对性能的影响

4.5 本章总结

在本章中，我展示了本算法的性能，并与其他平台现有的 SpTRSV 算法性能进行了对比，见图4-1。同时我也进一步展示算法的在预处理阶段和计算阶段分别占用的时间，见表4-2，相比于基于 level-sets 的算法，本算法的预处理时间大大减少。最后我也展示了我所使用的三个优化技巧对性能的提升。

第 五 章　总结与展望

5.1 工作总结

SpTRSV 稀疏下三角矩阵求解是现代数值计算中的一个重要算法。是现代科学计算中一个广泛使用的计算核心，在数值模拟计算中，通常会使用迭代法或直接法求解大规模稀疏线性方程组，而 SpTRSV 的效率直接影响了线性方程组的求解效率，提高 SpTRSV 算法的性能至关重要。SpTRSV 频繁且离散地数据访存、任务之间存在着很强的依赖、需要细粒度的同步、任务之间负载不均衡等特点是并行优化更加难以进行，正是这些特点使得通用且高效的 SpTRSV 算法仍然是一个颇具挑战性的课题。

在本文中作者提出了一个通过减少预处理的消耗，使用原子操作以及自旋等待的方式在运行时维护任务之间的运行顺序。并且作者结合了鲲鹏 920 处理器体系结构的特点进行了优化，例如使用了 SIMD 指令，ARMv8 的原子指令，以及一些缓存优化的技巧。最终作者所设计的 SpTRSV 算法对于任务间依赖相对较少的矩阵能够实现相比于串行算法 2 倍以上的的加速比，对于任务依赖较为负载的矩阵，算法的加速性能相对较差。

5.2 未来工作展望

对于 SpTRSV 算法还有很多可以尝试的优化思路，例如：在任务调度方面，基于手动创建的线程池，并设计一种调度器，通过这个调度器，在线程任务分发即阶段实现任务负载均衡的操作，减少预处理的时间。并且基于这个调度器，通过一定的调度策略减少 CPU 自旋等待的时间，这个思路来自于 GPU 的硬件调度器；在访存的优化方面，目前较多的操作需要离散且频繁的从内存进行读取，如何优化与内存的交互关系，还有优化的空间。

参考文献

[1] Davis T A. Direct methods for sparse linear systems[M]. SIAM, 2006.

[2] Elman H C. Iterative methods for large, sparse, nonsymmetric systems of linear equations[D]. Yale University New Haven, Conn, 1982.

[3] Saad Y. Iterative methods for sparse linear systems[M]. SIAM, 2003.

[4] Hogg J D. A fast dense triangular solve in cuda[J]. SIAM Journal on Scientific Computing, 2013, 35(3):C303-C322.

[5] Wang H, Liu W, Hou K, et al. Parallel transposition of sparse data structures[C]// Proceedings of the 2016 International Conference on Supercomputing. 2016: 1-13.

[6] Liu W, Vinter B. Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication[C]//Proceedings of the 29th ACM on International Conference on Supercomputing. 2015: 339-350.

[7] Liu W, Vinter B. Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors[J]. Parallel Computing, 2015, 49:179-193.

[8] Liu W, Vinter B. A framework for general sparse matrix–matrix multiplication on gpus and heterogeneous processors[J]. Journal of Parallel and Distributed Computing, 2015, 85:47-61.

[9] Anderson E, Saad Y. Solving sparse triangular linear systems on parallel computers [J]. International Journal of High Speed Computing, 1989, 1(01):73-95.

[10] Saltz J H. Aggregation methods for solving sparse triangular systems on multiprocessors[J]. SIAM journal on scientific and statistical computing, 1990, 11(1):123-144.

[11] Kabir H, Booth J D, Aupy G, et al. Sts-k: a multilevel sparse triangular solution scheme for numa multicores[C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 2015: 1-11.

[12] Park J, Smelyanskiy M, Sundaram N, et al. Sparsifying synchronization for high-performance shared-memory sparse triangular solver[C]//International Supercomputing Conference. Springer, 2014: 124-140.

[13] Schreiber T, R. Vectorizing the conjugate gradient method.[J]. Proceedings of the Symposium on CYBER 205 Applications, 1982.

[14] Wolf M M, Heroux M A, Boman E G. Factors impacting performance of multithreaded sparse triangular solve[C]//International Conference on High Performance Computing for Computational Science. Springer, 2010: 32-44.

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

[16] Naumov M. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the gpu[J]. NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011, 2011, 1.

[17] Suchoski B, Severn C, Shantharam M, et al. Adapting sparse triangular solution to gpus[C]//2012 41st International Conference on Parallel Processing Workshops. IEEE, 2012: 140-148.

[18] Liu W, Li A, Hogg J, et al. A synchronization free algorithm for parallel sparse triangular solves[M]//Euro Par2016: Parallel Processing: volume 9833. ChamSpringer International Publishing, 2016: 617-630.

[19] Liu W, Li A, Hogg J D, et al. Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides: Fast synchronization-free algorithms for parallel sparse triangular solves[J/OL]. Concurrency and Computation: Practice and Experience, 2017, 29(21):e4244. DOI: 10.1002/cpe.4244.

[20] 倪鸿刘鑫. 非结构网格下稀疏下三角方程求解器众核优化技术研究[M]. 北京大学, 2019.

[21] Wang X, Xu P, Xue W, et al. A Fast Sparse Triangular Solver for Structured-grid Problems on Sunway Many-core Processor SW26010[C/OL]//Proceedings of the 47th International Conference on Parallel Processing. Eugene OR USAACM, 2018: 1-11. DOI: 10.1145/3225058.3225071.

[22] Chhugani J, Satish N, Kim C, et al. Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency[C/OL]//2012 IEEE 26th International Parallel and Distributed Processing Symposium. Shanghai, ChinaIEEE, 2012: 378-389. DOI: 10.1109/IPDPS.2012.43.

[23] Davis T A, Hu Y. The university of florida sparse matrix collection[J]. ACM Transactions on Mathematical Software (TOMS), 2011, 38(1):1-25.

[24] 胡森森计卫星. 片上多核处理器 Cache 一致性协议优化研究综述[M]. 软件学报, 2017.

[25] Sarcar K. What is numa?[EB/OL]. https://www.kernel.org/doc/html/v4.18/vm/numa.html.

[26] Kerrisk M. numa(3) —linux manual page[EB/ OL]. https://man7.org/linux/man-pages/man3/numa.3.html.

[27] Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile[J]. :8248.

致　谢

感谢党和国家

感谢王云岚老师对我的指导和帮助

感谢中科院软件研究所和华为提供的计算资源

毕业设计小结

本次毕业设计是大学阶段的最后一次挑战。期间想了很多优化的方法和思路，但是由于自身对问题认识的不够深刻，提出了一些不切实际的优化思路，并且在这些不靠谱的想法上浪费了很多时间，其中包括企图通过超图分割算法进行任务图的分割，然后绑定到 NUMA 结点上，来提升内存的带宽，减少总线的冲突，后来经过实践，被证实是低效的。后来我有发现了一个名为 taskflow 的计算框架，可以在运行时构建任务依赖图，并且使用 work stealing 调度算法进行并行地执行这张图，性能好于 tpp 和 clik，但是基于该框架实现了 SpTRSV 算法之后，效果不是很理想。

虽然这次毕业设计对我来说是一次艰巨的挑战，但是也意义非凡。通过这次毕业设计我学习与回顾了本科所学的计算机知识，主要包括计算机组成原理和体系结构、计算机操作系统、编译原理，并将这些知识应用于实践当中，期间有想法不成功、遇到 bug 时候的挫败感，也有通过自己不断研究、不断实践，逐渐提升算法性能时的成就感。同时这次毕业设计也让我学到了一些科研的基础技能，为应对未来的挑战打下了基础。

总的来说，这次毕业设计是我从一个学习模仿者转变为一个创新研究者所必须经历的机遇与挑战。