高技术通讯 2013 年 第23 卷 第8 期:789 ~796

doi:10.3772/j.issn.1002-0470.2013.08.003

稀疏矩阵 LU 分解的 FPGA 实现①

邬贵明②* 王 淼** 谢向辉* 窦 勇***

(*数学工程与先进计算国家重点实验室 无锡 214125)

(**江南计算技术研究所 无锡 214125)

(***国防科学技术大学计算机学院 长沙 410073)

摘 要 研究了直接法求解稀疏线性方程组过程中最耗时的稀疏矩阵 LU 分解的数值计算,提出了一种稀疏矩阵 LU 分解并行算法,该算法可通过动态的相关性检测来开发更多的并行性。同时提出了基于现场可编程门阵列(FPGA)实现该并行算法的硬件结构,该结构不依赖于分解因子的稀疏结构信息,分解因子的数据结构可动态生成。与相关工作比较,这种新的硬件结构具有更好的通用性。实验结果表明,这种新的结构的性能优于通用处理器的软件实现。

关键词 稀疏矩阵,LU 分解,并行算法,现场可编程门阵列(FPGA),任务并行

0 引 言

科学计算或工程应用中的很多问题最后都归结为稀疏线性方程组求解,比如在天气数值预报、计算流体力学、有限元分析、偏微分方程求解等领域,而求解稀疏线性方程组过程中最耗时的是稀疏矩阵 LU 分解的数值计算。现场可编程门阵列(FPGA)具有大量逻辑资源和存储资源,计算能力随着半导体工艺的进步有了很大提升,目前成为加速各种计算应用的理想平台,可用以实现大规模并行计算。目前很少有基于 FPGA 可重构器件实现稀疏矩阵 LU 分解的算法,而已有方法都是针对特定领域的稀疏矩阵,缺乏通用性。现有的稀疏矩阵 LU 分解硬件结构分为两类:针对电力系统的结构[1-4]和针对电路模拟的结构[5]。Wang 等[1]使用基于 FPGA 的多个软处理器 IP 核构建了稀疏矩阵 LU 分解的并行处理器,将软核按照多指令多数据并行处理器进行组织。使用了适合于电力流分析的加边对角块(bordered diagonal block,BDB)稀疏矩阵存储方式和相应算法,实时电力系统的稀疏矩阵都可以采用这种方法进行处理,然而其它领域的稀疏矩阵却很难转换成 BDB 格式,因此该结构仅适用于电力系统应用。电力系统的负载流计算问题可归结于求解 Newton Raphson 方程,其中 Jacobian 矩阵形成的线性方程组求解最为耗时,Chagnon 等[2,3]面向这类线性方程组设计了专用的稀疏矩阵 LU 分解硬件结构。该结构实现了基于行的选主元 LU 分解算法,使用面向列的映射方法来降低主元搜索的复杂度。硬件主要由一个专用数据通路和一个专用 cache 组成,它们的设计参数(比如 cache 大小、cache 行大小和缓冲深度等)需要对稀疏矩阵进行深入的分析后才能够确定。简单的 cache 结构不能提供很好的性能,需要专门支持索引操作的硬件[4]。针对电路模拟(比如 SPICE[5]),Kapre 等[6]提出了一种数据流的阵列结构,它需要用高效的调度算法和工具来把计算流图映射到并行结构上,由于算法处理能力有限,性能受到了影响。

这些相关研究提出的稀疏矩阵 LU 分解并行结构不能适用于更多的稀疏矩阵,并且依赖于其它的辅助工具。针对这些问题,本文提出并实现了一种面向 FPGA 的稀疏矩阵 LU 分解的并行算法和相应的硬件结构。该算法可以处理更多的稀疏矩阵,具有更好的通用性,且不依赖于其它工具。实验结果表明,提出的结构性能优于通用处理器的软件实现。

① 国家自然科学基金(61125201)资助项目。

② 男,1981 年生,博士;研究方向:高性能计算机体系结构,可重构计算;联系人,E-mail:wuguiming@nudt.edu.cn

(收稿日期:2012-07-19)

1 稀疏矩阵和稀疏矩阵 LU 分解

稀疏矩阵被定义为一个具有大量零元素的矩阵[7],其最基本的思想是稀疏矩阵的零元素不用存储,从而可以节省大量存储空间。稀疏矩阵最常用的是压缩条格式,包括压缩稀疏行(compressed sparse row, CSR)格式和压缩稀疏列(compressed sparse column, CSC)格式。CSR 格式是按行对非零元素进行压缩,它包括三个数组:

- *val*:浮点型数组,按行连续存储非零元素的值;
- *col _ ind*:整型数组,存储 *val* 数组中对应非零元素的列索引;
- *row _ ptr*:整型数组,数组的元素 *row _ ptr*[*i*] 存储了第 *i* 行的第一个非零元素在 *val* 和 *col _ ind* 的偏移,即该行的起始地址。表示每行非零元素个数的数组 *len* 可由数组 *row _ ptr* 得到:*len*[*i*] = *row _ ptr*[*i* + 1] - *row _ ptr*[*i*]。

CSC 格式与 CSR 格式相似。CSC 格式为 CSR 格式按列压缩的版本,它是按列连续存储非零元素,对应的三个数组为 *val*、*row _ ind* 和 *col _ ptr*。

本文关注 LU 分解的 Left-Looking 算法,该算法是 *j* 为最外层循环时的 *jki* 版本,如图 1 所示。该算法先使用当前第 *j* 列之前的所有列来更新第 *j* 列(第 3 – 7 行),然后对第 *j* 列进行 *scale* 操作(第 9 – 11 行)。算法的主要操作是列-列更新,在 $a(k,j)$ 不为零时,第 *j* 列"主动"用第 *k* 列更新自己。

算法1:Left-Looking LU分解算法

```
1:  for j = 1 to n  do
2:     /* Left-Looking 列-列更新*/
3:     for k = 1 to j-1 where a(k, j) ≠0 do
4:          for i = k+1  to n where a(i, k) ≠0 do
5:               a(i, j) = a(i, j) - a(i, k) * a(k, j);
6:          end for
7:     end for
8:    /* 缩放第 j列(scale)*/
9:     for i = j + 1 to n where a(i, j) ≠0 do
10:          a(i, j) = a(i, j)/a(j, j);
11:    end for
12:  end for
```

图 1 稀疏矩阵 LU 分解

2 稀疏矩阵 LU 分解并行算法

首先介绍一种能够开发任务级并行的计算模型,然后给出基于该计算模型的稀疏矩阵 LU 分解并行算法。

2.1 计算模型

图 2 给出了一种能够开发任务级并行的计算模型。当前 FPGA 内部均有大量局部存储器(local memories),比如 Xilinx 器件中的 BRAM,这些 BRAM 都可以并行访问,提供了非常大的片内访存带宽。图 2 中的并行计算模型可以充分开发并行访存的特性。

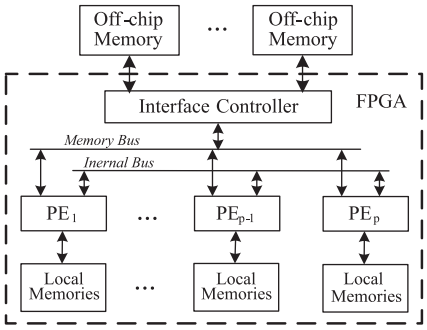

图 2 开发任务级并行的计算模型

在该计算模型中,初始数据并不是放在片内,而是放在片外存储器(off-chip memory)。整个计算系统由接口控制器和含有 *P* 个 PE 的处理单元阵列组成,PE 需要通过接口控制器才能访问外部存储器。每个 PE 的地位都是一致的,它们通过存储总线(Memory Bus)来获得外部存储器中的数据,多个 PE 同时访问时按照轮转的仲裁方式取得访问权。PE 还可以通过内部总线(Internal Bus)访问其它 PE 的局部存储器,同样,当多个 PE 同时要访问其它 PE 的局部存储器时,按照轮转的仲裁方式取得访问权。最后,这些结果会从本地存储器写回到外部存储器。

2.2 算法分析

Left-Looking LU 分解的特征在于被更新的列主动寻找前面的列来更新自己,被更新的列根据自己的非零结构可以动态地判定需要哪些列来更新自己,这个思想是对 LU 分解进行面向 FPGA 并行化分析的基础。

图 1 的算法中主要存在以下三种并行性:

- 不同的列可以同时被其它不同的列或同一列更新。从算法 1 第 3 行可以看出,仅当 $a(k,j) \neq 0$ 时才会使用第 *k* 列来更新第 *j* 列,而 $a(k,j)$,其中 $1 \leqslant k \leqslant j-1$,为第 *j* 列对角线以上的元素,因此根据对角线以上的元素就可以判定使用哪些列来更新

第 j 列。

● 多个 *scale* 之间的并行性。当两列之间由于稀疏结构不存在依赖关系,且这两列已经被相应的列更新完毕时,它们的 *scale* 任务可以同时运行。

● *scale* 和列 - 列更新之间的并行性。当某列不依赖于正在 *scale* 的列时,它进行的更新操作可以和这一列的 *scale* 同时运行。

对于第一种并行性,$a(k,j)$ 可能是更新时生成的填充,因此应该动态判定对角线以上的元素。后两种并行性来源于稀疏矩阵的非零结构特征,两列之间的依赖关系限制着这两种并行性的开发,可以通过动态检测两列的依赖关系来尽可能地开发这两种并行性。

2.3 任务调度

下面定义两种任务:*scale*(j) 和 *merge*(j, k, *temp*)。*scale*(j) 任务使用对角线元素 $a(j,j)$ 去除对角线以下的所有非零元素,即图 1 算法中的第 9 - 11 行。*merge*(j,k,*temp*) 任务使用 L 分解因子的第 k 列去更新第 j 列,$k < j$,标量 *temp* 为更新因子 $a(k,j)$,执行图 1 算法中的第 4 - 6 行,实际上是进行列 - 列更新,对应于稠密矩阵 LU 分解中的 *daxpy* 操作[8],当稀疏矩阵使用 CSC 格式时执行的是两列合并的操作。

稀疏矩阵被分成多个条块(panel),条块以轮转方式分配给 PE。为简化并行算法设计,本文采用简单的静态数据调度策略,同时执行的条块全部处理完毕之后才加载下一批条块进行操作。本文采用的任务调度策略具有以下特点:

● 采用 *scale*(k) 和 *merge*(j,k,*temp*) 两种以列为计算粒度的任务。计算粒度影响并行性的开发,以列为计算粒度进行调度的算法更适合于 FPGA 结构。

● 动态相关性分析和检测。在检测到多个任务之间不存在相关性时,多个任务便会并行执行;而指导并行执行的信息是动态建立的。

在一般文献中,常使用有向无环图(DAG)来表示稀疏矩阵 LU 分解和稠密矩阵 LU 分解的执行过程[9,10],DAG 的节点表示计算任务,边表示计算任务之间的相关性。Kurzak 等[9]借助 DAG 来实现稠密 LU 分解数据流的动态调度,Fu 等[10]利用 DAG 为划分任务后的稀疏 LU 分解的并行性进行建模,并使用图调度算法对任务进行动态调度,从而能够平衡计算负载并使计算时间与通讯时间重叠。DAG 往往都是计算前静态建立,然后引导调度器进行动态调度。文献[9]中任务节点为操作在小方块数据上的任务,文献[10]中任务节点为条块级任务,而本文采用的是列级任务节点,任务或为 *scale*(k),或为 *merge*(j,k)。*merge*(j,k) 为 *merge*(j,k,*temp*) 的简化。

与传统的 DAG 调度不同,我们并没有静态或动态地建立 DAG 数据结构,而是动态地检测任务之间的相关性,在不违背相关性的条件下尽可能地调度任务并行执行,但我们仍可以通过 DAG 来说明我们的动态调度机制。LU 分解因子的非零结构也是动态建立的,动态相关性检测正好与 L 和 U 分解因子的动态生成机制吻合在一起。另外,*merge*(j,k) 操作有可能在新的位置生成非零元素(即填充),而填充的数可能引入新的数据相关,这就要求我们的调度机制能动态地检测填充引入的数据相关。

通过分析图 1 中算法的第 3 - 7 行,可以得到关于判断 *merge*(j,k) 任务是否执行的两个重要结论:

● *merge*(j,k) 任务是否执行依赖于 $a(k,j)$ 是否为零,只有不为零时才会执行,而 k 满足 $k < j$,因此需要检测第 j 列的对角线以上的元素是否为零;

● 按照图 1 中算法第 3 行的语义,k 的取值范围是从 1 到 $j-1$,这说明 $a(k,j)$ 的检测过程是对对角线以上元素由上至下进行,若有填充发生,填充的位置同样能够被检测到,因为填充的位置一定在 $a(k,j)$ 的位置下面。

下面说明 DAG 如何揭示稀疏 LU 分解中的并行性,又如何进行任务调度来开发这些并行性。

图 3 给出了一个稀疏矩阵和对应的 LU 分解任务图,箭头表示任务之间具有相关性。*scale*(1) 和 *scale*(2) 不依赖于其它任务,因此可以和 *scale*(0) 并行执行。由于 $a(0,4)$ 不等于零,第 4 列依赖于第 0 列,因此任务 *merge*(4,0) 依赖于 *scale*(0),同样 *merge*(3,2) 依赖于 *scale*(2)。任务 *merge*(4,0) 产生了新的非零元素 $a(1,4)$,因此需要执行 *merge*(4,1),这时 *merge*(4,1) 依赖于 *merge*(4,0) 和 *scale*(1)。最后,*scale*(3)、*scale*(4) 分别依赖于 *merge*(4,1) 和 *merge*(3,2)。

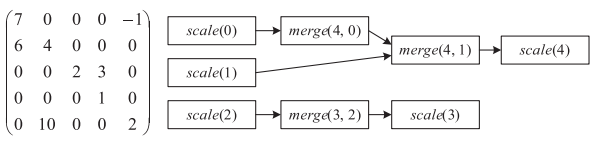

图 3 任务图示例

假设有 5 个 PE,每个 PE 分别分配一列进行计算,即第 0 列到第 4 列分别分配给 PE_1 到 PE_5。任务的分配基于简单的“拥有者 - 计算”原则,即对某列进行操作的任务都分配给该列所在的处理单元。图 4 说明了对图 3 中的任务图进行调度后的并行计算过程。PE_4 在进行 *merge*(3,2)之前要等待 PE_3 把 *scale*(2)计算完,PE_5 在进行 *merge*(4,0)之前要等待 PE_1 把 *scale*(0)计算完,PE_4 和 PE_5 需要分别访问 PE_3 和 PE_1 的局部存储器。图 4 没有显示访问其它 PE 的通讯开销,这个开销与图 2 并行计算模型的 Internal Bus 总线带宽相关。

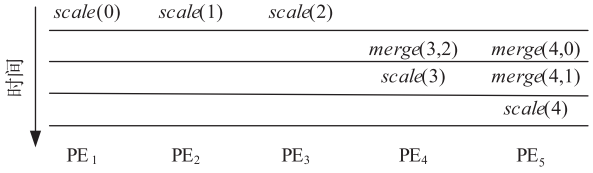

图 4 任务调度示例

2.4 并行算法

在 LU 分解并行算法中,串行算法将被分解成主算法和从算法,主算法仅负责数据的分配,从算法负责主要的计算。根据我们的计算模型,应该由 PE 执行从算法。由于没有依赖稀疏矩阵的符号分解,LU 分解因子的数据结构动态生成,PE 并不知道其计算的 LU 分解因子应该存放的地址。我们利用主算法进行数据分配解决了这一问题,主算法按照 PE 的编号顺序以轮转方式分配数据,从而确定了 LU 分解因子在外部存储器中的存放顺序。

下面定义一些变量和数据结构:

A _ row _ ind[ ]、*A _ val*[ ]和 *A _ len*[ ]:输入稀疏矩阵 A 的数据结构;

n:A 的维度;

L _ row _ ind[ ]、*L _ val*[ ]和 *L _ len*[ ]:分解因子 L 的数据结构;

U _ row _ ind[ ]、*U _ val*[ ]和 *U _ len*[ ]:分解因子 U 的数据结构;

k:当前条块的第一列的列号;

kk:当前列的列号;

pid:PE 号;

first _ row _ ind:矩阵 A 的当前列第一个非零元素的行号;

num _ col:一个条块包含的列数;

P:PE 个数。

图 5 给出了稀疏矩阵 LU 分解并行算法。该从算法采用单程序多数据(SPMD)方式进行描述。算法首先进行初始化,每个 PE 分配一个起始的条块(第 1 行)。接着,进入 while 循环,等待主算法送来分配的数据;while 循环的核心是列 - 条块更新,即遍历分配来的 $P(j)$ 的每一列,动态检测该列与其它列的相关性,并访问相应的 L。在列 - 条块更新过程中,首先从当前列的行索引数组中取出第一个非零元素的行索引值,并存储到变量 *first _ row _ ind* 中(第 6 行);然后把 *first _ row _ ind* 和对应的非零元素值加入到 U 分解因子的数据结构中(第 8、9 行);接着检测 *first _ row _ ind* 是否等于 *kk*,若相等则调用 *scale* 任务完成对第 *kk* 列的分解(第 12 行),若不相等则调用 *merge* 任务,用 L 分解因子的第 *first _ row _ ind* 列更新第 *kk* 列,更新后 *A _ row _ ind* 和 *A _ val* 被新的值所覆盖,算法跳至第 6 行检测新行索引数组中的第一个非零元素。若 *first _ row _ ind* 小于 *k*,在更新之前需要从其它 PE 或外部存储器中取来 L 的第 *first _ row _ ind* 列,否则 L 的第 *first _ row _ ind* 列已在 PE 的局部存储器中,不需要访问其它 PE 或外部存储器。PE 完成计算后进入同步等待状态(第 21 行),这是为了保证 PE 按照严格的顺序把计算结果写入外部存储器(第 22 行)。前一个 PE 把结果存储到外部存储器后,后一个 PE 才接着进行存储操作。严格的顺序保证了结果存储到正确的位置。算法最后为分配下一批数据做准备(第 23 行)。算法重复执行上述过程,直到稀疏矩阵所有的数据计

算法2:并行LU分解(PE[pid], $1 \leqslant pid \leqslant P$)

```
1:  k = num_col * pid;  /*初始化*/
2:  while (k < n) do
3:      加载A的一个条块P(k)
4:      for  kk = k to k + num_col - 1  do /*列-条块更新*/
5:          len = U_len[kk];
6:          从A_row_ind[]得到first_row_ind;
7:          len = len + 1;
8:          将first_row_ind加入U_row_ind[];
9:          将A_val[first_row_ind]加入U_val[] ;
10:         temp = A_val[first_row_ind];
11:         if (first_row_ind == kk)
12:             scale(kk);
13:         else  if (first_row_ind < k)
14:                 从外部存储器或其它PE获取L的第first_row_ind列;
15:             end if
16:             merge(kk, first_row_ind, temp); /*列-列更新*/
17:             go to 6;
18:         end if
19:         U_len[kk] = len;
20:     end for
21:     等待所有PE完成操作;  /*同步*/
22:     存储L和U到外部存储器; /*存储结果*/
23:     k = k + num_col * P;  /* 转移到下一条块*/
24: end while
```

图 5 稀疏矩阵 LU 分解并行算法

算完毕。

此算法有三个地方可能会引起 PE 等待:(1)多个 PE 同时访问外部存储器或其它 PE 的局部存储器,只有一个 PE 获得访问权,其它 PE 需要等待;(2)根据数据相关性,某个 PE 需要访问另外一个 PE 的局部存储器,若被访问的 PE 还没有计算完毕,则该 PE 需要等待;(3)每个 PE 的计算量不同,在同步时计算量小的 PE 需要等待计算量大的 PE。这三种等待会导致 PE 的浮点运算部件得不到充分的利用。

3 稀疏矩阵 LU 分解硬件结构

本节提出实现稀疏矩阵 LU 分解并行算法的硬件结构,该结构包括动态相关性检测和总线访问仲裁。首先提出总体结构,然后介绍外部存储访问逻辑和内部存储访问逻辑,最后阐述 *merge* 算法及其实现。

3.1 总体结构

提出的稀疏矩阵 LU 分解并行结构完全遵从开发任务级并行的计算模型,主要包括两条总线、连接外部存储器的接口控制器和多个 PE 组成的处理单元阵列。PE 通过 Internal Bus 访问其它 PE 的局部存储器,通过 Memory Bus 访问外部存储器。两条总线访问权限的获取通过轮转方式进行仲裁,因此还需要两个仲裁器和存储访问逻辑。

PE 结构如图 6 所示。PE 包括一个 Merge 单元、一个 Separate 单元和三个局部存储器。A _ RAM 用于保存输入稀疏矩阵一个条块的数据结构; L _ RAM 用于保存 L 分解因子一个条块的数据结构; U _RAM 用于保存 U 分解因子一个条块的数据结构。Separate 单元用于从 A _ RAM 中获得 U 分解因子的非零元素并存储于 U _ RAM 中。Merge 单元用

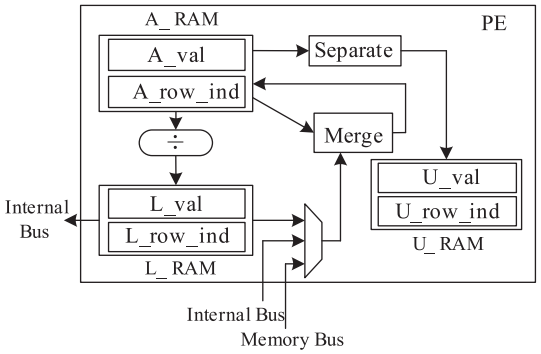

图 6 并行结构中的 PE

于实现 *merge* 任务,从 L _ RAM 中取出当前列去更新存储于 A _ RAM 中的条块,更新结果写回 A _ RAM 中。

并行算法的 *scale* 任务由一个除法器完成,*merge* 任务由 Merge 单元完成,包含一个乘加部件。执行 *merge* 任务时,L 分解因子可能来自三个地方:PE 自身的局部存储器、其它 PE 的局部存储器和外部存储器。因此,需要设置一个选择器来选择 Merge 单元可能的三个输入:L _ RAM、*Internal Bus* 和 *Memory Bus*,而 *first _ row _ ind* 的值决定了选择器选择哪一个作为输入。

3.2 存储访问逻辑

外部存储访问和内部存储访问只有在图 5 算法的第 14 行运行时才启动,这时 *first _ row _ ind* $< k$,这里的 k 为 PE 的局部 k,称 *pid* $= 0$ 时的 k 为全局 k。对于某个 PE,当 *first _ row _ ind* 小于局部 k 而大于或等于全局 k 时,说明另外一个 PE 在计算所需的 L 分解因子,启动内部存储访问,将该 PE 的 ID(即 *pid*)和 *first _ row _ ind* 作为请求信息发送给内部存储访问控制逻辑;当 *first _ row _ ind* 小于全局 k 时,说明所需的 L 分解因子已存放在外部存储中,启动外部存储访问,并将该 PE 的 ID(即 *pid*)和 *first _ row _ ind* 作为请求信息发送给外部存储访问控制逻辑。当 *first _ row _ ind* 大于局部 k 时,所需的 L 分解因子已存储在本地,不需启动外部或内部存储访问。

为实现高效的直接内存存取(DMA)数据传输,采用了两个外部存储器:一个外部存储器存储稀疏矩阵的 *val* 和 *row _ ind*,另一个外部存储器存储稀疏矩阵每一列的起始地址和每一列的非零元素个数(称为附加数据结构)。两个存储器可以同时访问,当访问某一列的 *val* 和 *row _ ind* 时,下一列的附加数据结构已从另一存储器中得到,当前列访问完毕可以立即开始访问下一列,隐藏了 DMA 访问的准备时间。

图 7 给出了外部存储访问控制逻辑,包括控制稀疏矩阵数据的读取和控制附加数据结构的读取,主要由 4 个 FIFO、一个仲裁器、一个 DMA 引擎和一个读控制器(read controller)组成。该控制逻辑的运行过程是:首先,来自 PE 的外部存储器访问请求通过仲裁器仲裁,获得访问权的请求的所有信息缓存在 Add FIFO 和 PID FIFO 里,Add FIFO 存放 *first _ row _ ind*,*first _ row _ ind* 实际上为附加数据结构在外部存储器的地址,PID FIFO 存放获得访问权的 PE

的 *pid*;从 Add FIFO 中读出一个地址,使用该地址从外部存储器中读出附加数据结构,并缓存在 Add _ size FIFO 里,为 DMA 引擎启动 DMA 操作提供起始地址和数据长度;DMA 引擎从外部存储器读出稀疏矩阵的 *val* 和 *row _ ind*,并缓存在一个数据 FIFO 里,PID FIFO 中的 *pid* 作为数据的目的地和数据一起加载到 Memory Bus 上。

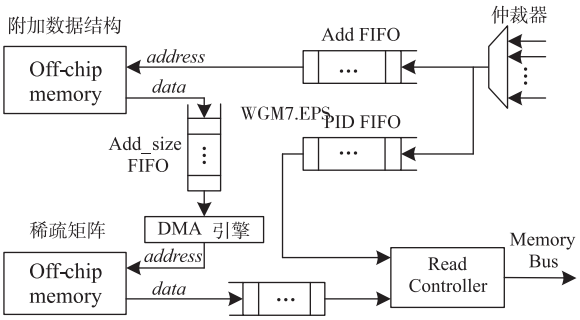

图 7 外部存储访问控制逻辑

内部存储访问控制逻辑控制着 PE 如何访问其它 PE 的局部存储器,所有 PE 通过 Internal Bus 进行访问,因此需要一个仲裁器对多个访问请求进行仲裁。内部存储访问控制逻辑与外部存储访问控制逻辑类似,这里不再赘述。

3.3 Merge 算法和结构

Merge 单元执行的是列 - 列更新算法,如图 8 所示。该算法把两列的稀疏结构合并成一列,输入为稀疏矩阵 *A* 的一列(数据结构为 *A _ row _ ind*、*A _ val* 和 *A _ len*)、L 分解因子的一列(数据结构为 *L _ row _ ind*、*L _ val* 和 *L _ len*)和标量 *temp*,输出为新的一列(数据结构为 *New _ A _ row _ ind*、*New _ A _ val* 和 *New _ A _ len*)。该算法采用了文献[4]的描述方式,主要包括三个条件分支:复制、更新和填充。主要思想是通过比较两个向量的行索引生成新向量的行索引,实际是对两个向量的行索引合并并重新排序,得到新的行索引 *New _ A _ row _ ind*,相应的非零元素值也跟着进行合并并经过一定计算后重新排序,得到新向量的非零元素值。

算法3:A的列和L的列合并算法

```
1:  i= 0; j= 0; k= 0;
2:  do
3:      A_index= A_row_ind[i];
4:      L_index= L_row_ind[j];
5:      if (A_index< L_index)  /* 复制*/
6:          New_A_row_ind[k] = A_index;
7:          New_A_val[k] = A_val[i];
8:          k = k + 1; i = i + 1;
9:       else if(A_index== L_index)  /*更新*/
10:             New_A_row_ind[k] = A_index;
11:             New_A_val[k] = A_val[i] -temp* L_val[j];
12:             k = k + 1; i = i + 1; j = j + 1;
13:           else /*填充*/
14:             New_A_row_ind[k] = L_index;
15:             New_A_val[k] = -temp* L_val[j];
16:             k = k + 1;j=j+1;
17:           end if
18:      end if
19:  } while (i< A_len|| j< L_len);
20:  New_A_len =k;
```

图 8 列 - 列更新算法

如图 9 所示,Merge 单元分成两部分:上半部分实现行索引的比较、排序和合并;下半部分根据行索引的比较结果得到新向量。*A _ index* 和 *L _ index* 分别是从两个 FIFO 中读出来的索引值。

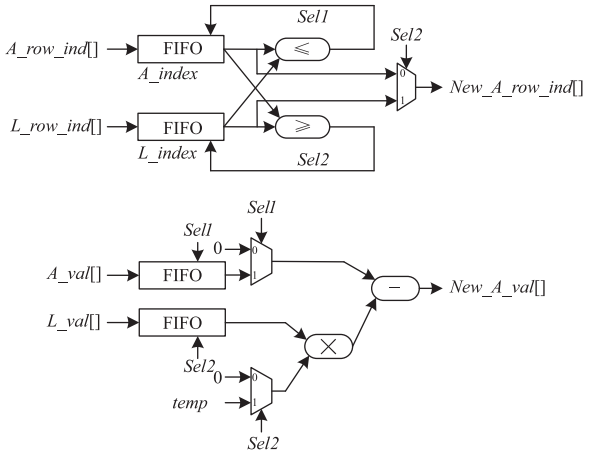

图 9 Merge 单元电路逻辑

*Sel*1 控制着缓存 *A _ row _ ind* 和 *A _ val* 的两个 FIFO 的读操作,*Sel*2 控制着缓存 *L _ row _ ind* 和 *L _ val* 的两个 FIFO 的读操作。FIFO 读操作的同步使 *A _ val* 和 *L _ val* 的合并与 *A _ row _ ind* 和 *L _ row _ ind* 的合并完全同步。当 *A _ index* < *L _ index* 时,*Sel*1 有效而 *Sel*2 无效,*A _ index* 被选择作为输出,并从缓存 *A _ row _ ind* 的 FIFO 中读出新的 *A _ index*,同时下半部分电路执行第 7 行;当 *A _ index* = *L _ index* 时,*Sel*1 和 *Sel*2 均有效,*L _ index* 被选择作为输出,并分别从缓存 *A _ row _ ind* 和 *L _ row _ ind* 的两个 FIFO 中读出新的 *A _ index* 和新的 *L _ index*,同时下半部分电路执行第 11 行;当 *A _ index* > *L _ index* 时,*Sel*2 有效而 *Sel*1 无效,*L _ index* 被选择作为输出,并从保存 *L _ row _ ind* 的 FIFO 中读出新的 *L _ index*,同时下半部分电路执行第 15 行。

Merge 单元由于采用了全流水结构,上部分电

路执行时间的下限为 min(A_len,L_len),上限为 $A\_len + L\_len$;下部分电路执行时间的上下限等于上部分电路执行时间的上下限加上浮点乘加电路的流水线级数。

4 实 验

4.1 实验环境建立

我们采用 Verilog HDL 对设计进行描述,以 Xilinx Virtex-5 XC5VLX330 为目标器件。我们先使用 ModelSim6.2h 进行模拟验证,保证逻辑正确;然后使用 Xilinx ISE 10.1 进行物理综合和布局布线。

软件实现运行于 Intel Core 2 Quard Q6600 CPU,运行频率为 2.4 GHz,内存为 8 GB DDR2。测试的稀疏矩阵来自 Florida 大学 Timothy A Davis 教授维护的 Sparse Matrix Collection[11],该稀疏矩阵库中的矩阵都是来自实际应用的矩阵,覆盖了结构工程、计算流体力学、电磁学、半导体器件、热动力学、最优化、电路模拟等领域。我们选取的矩阵 494_bus、1138_bus、poewersim 来自于电力网络,c-18、c-19、c-20 来自于非线性优化,memplus 来自于电路模拟,msc01050、nasa1824 来自于结构问题,problem1 来自于有限元应用。硬件设计的测试输入是由 Matlab 进行 AMD[12] 排序后生成。

4.2 实验结果

在 XC5VLX330 上实现了规模为 4、8 和 14 个 PE 的三种硬件设计,每个 PE 局部存储器深度为 2048 时,综合频率达到 179MHz。在实验评测时,我们的结构采用了一条外部存储访问总线和一条内部存储访问总线。综合性能如表 1 所示。

表 1 综合性能

	1 PE	8 PE	14 PE
Registers	7360(3.5%)	63754(30%)	104029(50%)
LUTs	9501(4.6%)	87486(42%)	145345(70%)
Slices	4137(8.0%)	33869(65%)	50269(96%)
DSP48Es	10(5.2%)	80(41%)	141(73%)
BlockRAMs	17(5.9%)	161(55%)	263(91%)

表 2 对硬件设计与软件实现的性能进行了比较。软件实现采用 Matlab 中的 *lu* 函数,该函数实现的是 UMFPACK[13]。可以看出,相对于软件实现,FPGA 实现的含 14 个 PE 的并行结构能达到 9.48 的加速比,加速比最差为 1.39。

表 2 性能比较

测试矩阵	微机运行时间(ms)	FPGA 运行时间(ms)	加速比
494_bus	0.59	0.24	2.45
1138_bus	1.16	0.56	2.07
Powersim	21.68	8.03	2.69
c-18	43.55	4.72	9.22
c-19	191.85	23.74	8.08
c-20	71.33	7.52	9.84
Memplus	22.59	16.15	1.39
Msc01050	17.62	10.23	1.72
Nasa1824	100.66	38.01	2.64
Problem1	1.65	1.01	1.63

5 结 论

本文提出了一种稀疏矩阵 LU 分解并行算法和相应的硬件结构,并用 FPGA 得到了实现。与通用处理器相比,其性能比较有优势。本文提出的并行算法不依赖于符号分解,这可以减少一定的数据传输代价。然而,符号分解提供的信息有利于同时开发稀疏 LU 分解的并行性和数据重用性,可以指导硬件结构进行更有效的数据和任务调度。因此,研究基于符号分解的稀疏 LU 分解并行结构,将是未来的研究工作,同时我们还将研究总线带宽对性能的影响。

参考文献

[1] Wang X, Ziavras S. Parallel LU factorization of sparse matrices on FPGA-based configurable computing engines. *Concurrency and Computation: Practice and Experience*, 2004, 16(4): 319-343

[2] Chagnon T, Johnson J, Vachranukunkiet P, et al. Sparse LU decomposition using FPGA. In: Proceedings of the 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing, Trondeim, Norway, 2008

[3] Vachranukunkiet P. Power-flow computation using Field Programmable Gate Arrays: [Ph. D dissertation]. Drexel University, 2007

[4] Chagnon T. Architectural support for direct sparse LU algorithms: [Master dissertation]. Drexel University, 2010

[5] Nagel L. SPICE2: a computer program to simulate semiconductor circuits: [Ph. D dissertation]. University of California, Berkeley, 1975

[6] Kapre N, DeHon A. Parallelizing sparse matrix solve for SPICE circuit simulation using FPGAs. In: Proceedings of the 2009 IEEE International Conference on Field-Programmable Technology, Sydney, Australia, 2009. 190-198

[7] Demmel J. Applied Numerical Linear Algebra. The Society of Industrial and Applied Mathematics. 1997

[8] Wu G, Dou Y, Lei Y, et al. A fine-grained pipeline implementation of the LINPACK benchmark on FPGAs. In: Proceedings of the 2009 IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, California, USA, 2009. 183-190

[9] Kurzak J, Dongarra J. Fully dynamic scheduler for numerical computing on multicore processors. University of Tennessee LAPACK Working Note #220, 2010

[10] Fu C, Jiao X, Yang T. Efficient sparse LU factorization with partial pivoting on distributed memory architectures. *IEEE Transactions on Parallel and Distributed Systems*, 1998, 9(2): 109-125

[11] Davis T A, Hu Y. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 2011, 38(1): 1-25

[12] AMD. http://www. cise. ufl. edu/research/sparse/amd, 2012

[13] Davis TA. Algorithm 832: UMFPACK V4. 3-an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 2004, 30(2): 196-199

Implementation of sparse LU decomposition using FPGAs

Wu Guiming*, Wang Miao**, Xie Xianghui*, Dou Yong***

(* State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi 214125)

(** Jiangnan Institute of Computing Technology, Wuxi 214125)

(*** Computer School, National University of Defense Technology, Changsha 410073)

Abstract

The most time-consuming numerical computation in sparse LU decomposition with the direct method was studied, and a parallel sparse LU decomposition algorithm was presented, with which more parallelisms can be developed by dynamic dependence analysis. And a hardwate structure implemented using the parallel sparse LU decomposition algorithm based on field programmable gate orrays (FPGAs) was proposed. The design of the hardware structure does not need the sparsity structural informafion of the decomposition factors, and the data structures of decomposition factors are generated dynamically. The proposed design is more general than that proposed in related work. The experimental results show that this new LU decomposition design outperforms the software implementation on the general-purpose processors.

Key words: sparse matrix, LU decomposition, parallel algorithm, FPGA, task parallelism