高技术通讯 2009 年 第 19 卷 第 5 期:511 ~ 518

doi:10.3772/j.issn.1002-0470.2009.05.013

可选主元 LU 分解流水线算法设计与 FPGA 实现①

牛 新② 周 杰 窦 勇 雷元武

(国防科技大学计算机学院 长沙 410073)

摘 要 提出了一种可以进行列主元选取的细粒度 LU 分解流水线算法并在现场编程门阵列(FPGA)上得到了实现。该算法可以在进行列主元选取的同时,充分利用数据的重用性,以减少数据读写次数。对其中的关键运算实现了细粒度全流水,提高了分解性能。与 Celeron(R) 3.07GHz 通用处理器主机相比可以得到平均 6 到 7 倍的加速比。与其他在 FPGA 上实现的 LU 分解算法相比,该算法在占用相对较少资源和保持高分解效率的前提下提高了计算的精确度和稳定性。

关键词 LU 分解,流水线,并行算法,列主元选取,现场编程门阵列(FPGA)

0 引 言

LU 分解是一种科学计算中的矩阵分解方法,常用于线性方程的求解、矩阵求逆和行列式计算,在图形图像处理、科学计算等诸多领域也有广泛应用。现场编程门阵列(FPGA)技术的进步,使其运算速度和存储空间都得到了很大的提高,功能变得更加强大灵活。例如在 Xilinx 和 Altera 的流行系列上通常都能集成上百个数字信号处理器(DSP)和几十兆的存储空间,这使得在其上实现大规模细粒度并行计算应用成为可能。

目前在可重构器件上实现了多种 LU 分解算法,但是很少有可进行部分选主元的高性能的 LU 分解算法。如果 LU 分解不进行主元选取,会影响到计算精度和正确性,在碰到一些奇异矩阵时,甚至会导致计算失败。目前已有的算法有 Kieron Turkington 等实现的通过软流水循环展开调度的算法[1],Seonil Choi 等实现的基于自顶向下(up-down)LU 分解的流水线算法[2]。还有一些是比较经典的 LU 分解 systolic 阵列[3,4]和分块算法[5]。他们的可重构分解方法都可以极大地挖掘分解过程中的并行性,提高运算的效率。但是 Kieron Turkington 的分解方法是一种在功能级别上的一种粗粒度并行,计算负载上的不平衡和数据传递时的大量计算停顿,都会影响计算效率。而 Seonil Choi 的方法和现有的 systolic 阵列都由于简化设计没有考虑主元的选取。而那些分块算法由于将整列分割成块,本质上就不能进行选主元。针对上述算法的问题,本文提出了一种面向大规模矩阵的可以进行列主元选取的流水线 LU 分解算法,并在 FPGA 上得到了实现。该算法在通过列主元选取提高计算精度的同时,还可通过充分发掘并行性来提高分解性能。

1 LU 分解算法概述

列主元选取部分行交换的矩阵 LU 分解的原理可用式 $L_nP_nL_{n-1}P_{n-1}\cdots L_1P_1A = U$ 表示,其中 A 是一个 $n \times n$ 矩阵,$U = (u_1, u_2, \cdots, u_n)$ 是一个 $n \times n$ 的上三角矩阵,$P_i(1 \leqslant i \leqslant n)$ 是第一种行初等矩阵:将第 i 行与第 $j(i < j)$ 行进行交换。其中

$$L_i = \begin{pmatrix} 1 & & & & \\ & \ddots & & 0 & \\ & & 1 & & \\ & & l_{i+1,i} & & \\ 0 & & \vdots & 0 & \ddots \\ & & l_{n,i} & & 1 \end{pmatrix}$$

易知

$$L_i^{-1} = \begin{pmatrix} 1 & & & & \\ & \ddots & & 0 & \\ & & 1 & & \\ & & -l_{i+1,i} & & \ddots \\ 0 & & \vdots & 0 & \ddots \\ & & -l_{n,i} & & 1 \end{pmatrix}$$

① 863 计划(2007AA01Z106)和国家自然科学基金(60633050,60621003)资助项目。

② 男,1983 年生,博士生;研究方向:高性能计算机体系结构,遥感图像处理;联系人,E-mail: xinniu2008@gmail.com

(收稿日期:2008-06-02)

令 $L = L_1 L_2 \cdots L_{n-1} L_n$

$$= \begin{pmatrix} 1 & & & & & \\ \vdots & \ddots & & & 0 & \\ l_{i,1} & \cdots & 1 & & & \\ l_{i+1,1} & \cdots & l_{i+1,i} & \ddots & & \\ \vdots & \vdots & \vdots & \vdots & \ddots & \\ l_{n,1} & \cdots & l_{n,i} & \cdots & \cdots & 1 \end{pmatrix}$$

其中 L 的第 i 列就是 L_i 的第 i 列，这就包含了所有 L_i 的信息。

对于线性方程 $Ax = b$，如果分解出矩阵 U, L，并且记录下交换矩阵 P_i，则可得到相应的快速求解过程：$Ux = y = L_n^{-1} P_n L_{n-1}^{-1} P_{n-1} \cdots L_1^{-1} P_1 b$。

将一个矩阵进行 LU 分解的方法很多。而对于大规模矩阵的分解，由于 FPGA 内部存储容量有限，需要将大规模矩阵进行分块处理。将矩阵进行分块，能够更多发掘分解过程中的并行性。但为了进行列主元选取，就不能将整列在分块时拆开。这就要求能够进行列主元选取的 LU 分解的并行算法结构要以完整列块为组成单位[6]。为此，我们选择一种叫做 left-looking[7] 的串形 LU 分解算法作为并行化的基础，它是以整列块为单位进行分解的。根据更新过程中迭代步数 k 和选取更新元素下标 i, j 在循环层次中排列的相对顺序，这种 left-looking LU 分解又称为列主元 Gauss 消去法的 Kji 方法[6]。

为了解释算法过程，首先定义如下向量运算：

scale(k)：用第 k 列对角线元素 $a_{k,k}$ 遍除第 k 列对角线以下的所有元素：

```
For i = k + 1, k + 2, ···, n
    a_{i,k} = a_{j,k} / a_{k,k};
Endfor
```

axpy(k,j)：列 j 被列 k 做如下向量更新：

```
For i = k + 1, k + 2, ···, n
    a_{i,j} = a_{i,j} - a_{k,j} * a_{I,k};
Endfor
```

swaprow(k, j)：将第 k 行和第 j 行 k 列以后的元素进行行交换：

```
For i = k, k + 1, ···, n
    {temp = a_{j,i};   a_{j,i} = a_{k,i};   a_{k,i} = temp;}
Endfor
```

由于只交换了部分行，所以我们称这个过程为部分行交换。

swap(a, b)：将元素 a,b 进行交换：

```
{temp = a;   a = b;   b = temp;}
```

imax(k)：选取 k 列中的按模最大元素记为主元并记录其行号

```
max = |a_{k,k}|
For i = k + 1, ···n
    If(|a_{k,i}| > max) {max = |a_{k,i}|;   imax = i;}
Endfor
```

则 LU 串行分解过程如算法 1 伪代码所示。

在每一个迭代步 k 中，要进行 4 个步骤操作：

S1：选出第 k 列的主元并记录其所在列 $pivot\_k$。

S2：将第 k 行与第 $pivot\_k$ 行 k 列以后的元素进行交换，即交换 $a_{k,j}$ 与 $a_{pivot\_k,j}(k \leqslant j \leqslant n)$，是交换行的后半部分。

S3：用第 k 列对角线元素，即新选出的主元 $a_{k,k}$ 遍除第 k 列对角线以下的所有元素。

S4：用第 k 列对其后所有列 $j(k < j \leqslant n)$ 作 axpy 更新。

由于我们要将矩阵分为整列块操作，所以对于 S2 的行交换操作，就必须打碎到每个列块中进行，即可以放在 S4 步中。在每个 axpy 操作执行前先将第 j 列的第 k 行元素 $a_{k,j}$ 和 $pivot\_k$ 行元素 $a_{pivot\_k,j}$ 进行交换。这样在 k 列对其后所有 $j(k < j \leqslant n)$ 列作完 axpy 运算后，也同时完成了 k 行和 $pivot\_k$ 行的部分行交换。这就导出了将部分行交换放入 S4 循环中的等价串行算法，而这个算法是我们后面要提到的并行算法的基础。

算法 1：列主元选取 LU 串行分解过程

Kji version：	**将部分行交换放入循环中即**：
For k = 1,2,···,n	For k = 1,2,···,n
S1: pivot_k = imax(k);	S1: pivot_k = imax(k);
S2: swaprow(k, pivot_k)	S2: swap(a(k,k), a(pivot_k,k));
S3: scale(k);	S3: scale(k);
S4: For j = k + 1, k + 2, ···, n	S4: For j = k + 1, k + 2, ···, n
axpy(k,j);	swap(a(k,j), a(pivot_k,j));
Endfor	axpy(k,j);
Endfor	Endfor
	Endfor

在每一迭代步 k 的 S3 步结束后，列 k 其实已经被更新完毕，作为最终结果在每一迭代步的最后将会被保存。而保存前，要用列 k 执行 S4，更新其后每一列。矩阵 A 就这样被一列一列地分解出来。一直做到第 n 步，矩阵 A 即分解完毕，此时 U 存储于原矩阵的右上角，而 L 对角线以下的元素存储于原矩阵左下角。而每一步选出的主元所在行信息 pivot_k 实际上记录了每一步的交换矩阵 P_i。

已有研究[1]表明,Kji-LU 分解超过 90%的处理时间消耗在所有迭代过程中的 S4 步 axpy 计算过程中。显然,为了得到显著的性能提升,必须对这个阶段进行加速。所以如何根据数据流动的特点对 axpy 操作进行并行化处理是加速 LU 算法的关键。

2　列主元选取 LU 分解并行算法

从待更新的列 j 的角度看,列 j 更新完毕成为已更新列,就是从已更新列 1 开始,列 j 接受从左到右已更新列 $k(k<j)$依次对其进行 axpy(k,j)处理。之后列 j 进行选主元并将主元和对角线元素交换,再用选出的主元除对角线以下元素。经过这些处理,列 j 就成为已更新列。这样就自然导出了 Kji 版本 LU 分解的流水线算法。其实质是将 S4 循环展开后流水化向量运算 axpy。

按此想法,我们提出一种一维阵列的流水线结构来加速分解过程。就是利用数据的可重用性,在得到每个已更新列 k 后,使其常驻流水单元 PE 中,每个常驻 PE 的已更新列,对其后进入流水线的列 j 流水进行 axpy 处理。

整个流水分解过程分为多个迭代。在每次迭代中,每个 PE 都要经历填充、流水、排空三个阶段。在填充阶段,每个 PE 顺序生成本次流水需常驻其中的已更新列;流水阶段,每个 PE 中的常驻已更新列对流经它的各列作 axpy 运算;排空阶段,各个常驻 PE 的已更新列写回外存。如果流水线中有 p 个 PE,则每次迭代就分解出 p 个已更新列,并用这 p 个已更新列对其后的所有的列作 axpy 运算,对整个矩阵进行一遍更新。若矩阵规模为 n,则经过 n/p 次迭代,整个矩阵就以列为单位分解完毕。

要流水加速的 axpy 运算实际上就是将一个向量乘以一个标量后与另一个向量相加,这个过程可以流水化为向量中每个元素的乘加操作:$a_{ij} = a_{ij}a_{kj} * a_{ik}$。为了让之后的 axpy 操作能不间断流水作业下去,中间不被元素交换和临时存储操作打断,我们将原来在算法 1 第 S4 步中每次 axpy 前作的交换操作集中处理,使得列 j 的每一个元素在已更新列 k 对其作完 axpy(k,j)运算后,可以直接流入下一个 PE 让已更新列 $k+1$ 对其进行 axpy($k+1$,j)运算,不用被交换打断。实现 axpy 运算过程的全流水化。

按此思想设计的流水线算法如算法 2 所示。流水线分为PE0,PEj,PEN三种PE。各个PE的功能和

算法 2:列主元选取 LU 分解流水线并行算法

```
//PE0并行工作过程
For k=0 to [n/p]
S1:   LoadColumn (k*p+1);
      SendColumnToPE  1(k*p+1);
      For i=k*p+2 to (k+1)*p
          RecFromPE  1(Pivot(i-1));
          LoadColumn (i);
          For l=k*p+1 to i-1
              Swap (a(l,i),a(Pivot(l),i));
          Endfor
          SendColumnToPE  1(i);
      Endfor
S2:   For i=(k+1)*p+1 to n
          LoadColumn (i);
          For l=k*p+1 to (k+1)*p
              Swap (a(l,i),a(Pivot(l),i));
          Endfor
          SendColumnToPE  1(i);
      Endfor
  Endfor

  //PEN并行工作过程
  For k=0 to [n/p]
S1:   For i=(k+1)*p+1 to n
          RecColumnFromPEp  (i);
          StoreColumn  (i);
      Endfor
S2:   For i= k*p+1 to (k+1)*p
          RecColumnFromPEp  (i);
          StoreColumn  (i);
      Endfor
  Endfor
```

```
//PEj并行工作过程
For k=j to n step p
S1:   For i=[k/p]*p to n
          RecFromPEj  -1(ain);
          Max(ain);
      Endfor
      SendToPEj  -1(Pivot(k));
S2:   Swap (a(k,k),a(Pivot(k),k));
      Scale (k);
S3:   For i=k+1 to ([k/p]+1)*p
S31:      For l=[k/p]*p to n
              RecFromPEj  -1(ain);
              aout =axpy(ain);
              SendToPEj  +1(aout);
          Endfor
S32:      RecFromPEj  +1(Pivot(i));
S33:      Swap (a(i,k),a(Pivot(i),k));
S34:      SendToPEj  -1(Pivot(i));
      Endfor
S4:   For i=([k/p]+1)*p+1 to n
          For l=k+1 to n
              RecFromPEj  -1(ain);
              aout =axpy(ain);
              SendToPEj  +1(aout);
          Endfor
      Endfor
S5:
S51:  For i=([k/p]+1)*p to k
          Swap (a(i,k),a(Pivot(i),k));
      Endfor
S52:  For l=[k/p]*p to n
          RecFromPEj  -1(ain);
          aout =ain;
          SendToPEj  +1(aout);
      Endfor
S53:  For     l=[k/p]*p to n
          SendToPEj  +1(a(l,k));
      Endfor
  Endfor
```

算法步骤如算法 2 为代码所述:

PE0:从外存读取每一列,经过集中交换处理后送入流水线。对每一次迭代 $k(0 \leqslant k < [n/p])$

S1:此时流水线正处于填充过程,PE0 从外存中读出列 $k*p+i$,根据之前接收到的列主元所在行信息,按照第 $k*p+1$ 行到 $k*p+i-1$ 行的顺序,与第 Pivot($k*p+1$)行到 Pivot($k*p+i-1$)行进行交换。交换后列 $k*p+i$ 进入流水线,经过流水线处理后生成已更新列贮存在 PEi 中。之后 PEi 逆向传递给 PE0 列 $k*p+i$ 主元所在行:Pivot($k*p+i$),PE0 接收保存后读取下一列进行同样操作直到流水线填充完毕。

S2:此时流水线填满已更新列,开始 axpy 流水阶段。PE0 从外存中读取 $(k+1)*p+i$ 列,按照第 $k*p+1$ 行到 $(k+1)*p$ 行的顺序与第 Pivot($k*p+1$)行到 Pivot($(k+1)*p$)行进行交换。交换后列 $k*p+i$ 进入流水线。

PEj:($0<j\leqslant p$)流水线中的分解单元。p 是流水线中 PE 的个数。在每次迭代 k 中:

S1:接收存储列 k 的元素,同时作列主元选取。

S2:将列 k 主元所在行 Pivot(k)与 k 行元素进行交换。之后进行 Scale(k)运算,此时 k 成为已更新列贮存于 PEj 中。

S3:填充流水线过程。辅助生成流水线中其余 PE 中需要常驻的已更新列,并接收传递这些列的主元所在行信息,进行集中交换。这个过程一直循环直到流水线填满为止。假设 $m+i$ 是其后 PEi($j<i\leqslant p$)中要驻留的已更新列号,填充过程为:

S31:流经 PEj 的 $m+i$ 列元素流水作 axpy 处理。

S32,**S33**,**S34**:列 $m+i$ 更新成为已更新列存储在 PEi 后,逆向传递给 PEj 列 $m+i$ 主元所在行 Pivot($m+i$),PEj 接收保存后将列 k 的 $m+i$ 行和 Pivot($m+i$)行进行交换。之后再将 Pivot($m+i$)传递给上一个处理单元,PEj - 1。

S4:流水线进入 axpy 全流水阶段,接收 PEj-1 传来的元素,经过 axpy 处理发送给 PEj + 1。

S5:流水线排空过程。分为三步:

S51:PEj 根据保存的 PEj + 1 到 PEp 的列主元所在行信息,将 $l+p$ 行到 $k+1$ 行,按这种顺序分别与 Pivot($1+p$)与 Pivot($k+1$)行进行交换。这其实是对 S3 中交换进行的逆操作。使其恢复填充流水线前的行顺序。

S52:向后传递写回的原驻留 PEi($i<j$)中的已更新列。

S53:传递完所有之前处理单元的已更新列后,将本 PE 中存储的已更新列 k 向后发送传递写回。

3.PEN:流水线中的写回单元。接收最后一个分解单元 PEp 发送来的各个矩阵列,并写回外存。

S1:对应 axpy 流水阶段,接收被 axpy 流水处理的各列的中间结果,写回外存。

S2:对应流水线排空阶段,接收流水线处理单元中存储的已更新列,写回外存。

这里略去算法正确性的数学证明。主要思想就是在每一列进入流水线进行更新之前,提前根据流水线中 PE 传递来的 Pivot 信息,将要进行的交换先打包集中处理。这样在 axpy 流水阶段,经过流水线各 PE 时,每一列就不需再交换,从而可以实现 axpy 操作的全流水。一个 3PE 的全流水 LU 分解流水线逻辑结构和时空图如图 1 所示。

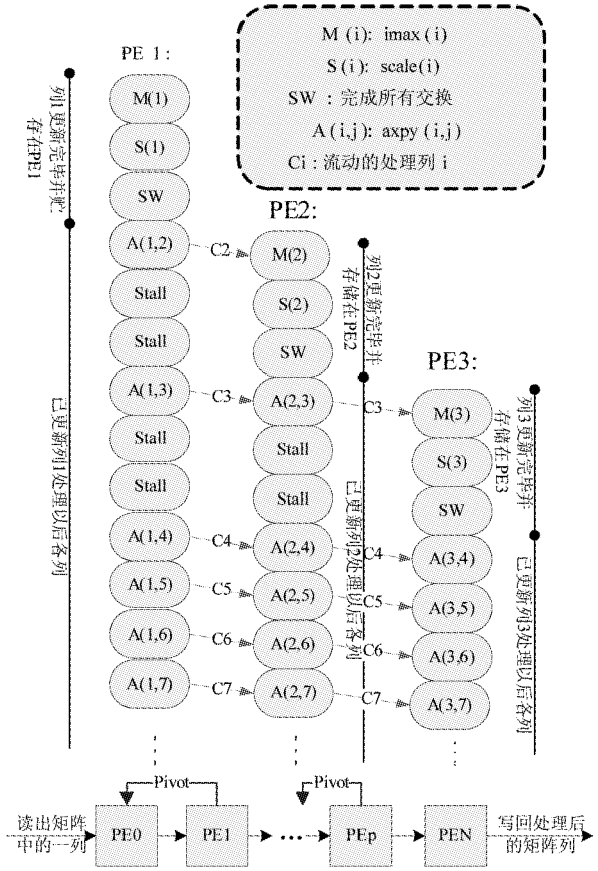

图 1 3PE 流水线 LU 分解结构及时空图

3 LU 分解流水线结构与实现

图 2 给出了 LU 分解流水线的硬件结构和 PE 内部结构。LU 分解流水线加速器主要包括一片 FPGA、一条 SDRAM 存储器和 I/O 接口(USB 和 JTAG)。

微机通过 USB 接口将启动命令和数据发送给加速器,并接收完成信号。矩阵初始数据及分解结果数据存于 SDRAM 中。LU 分解流水线通过存控完成和 SDRAM 数据的交互。PE0 和 PEn 每拍通过存控,从 SDRAM 中读取或写回一个 32bit 浮点数。下面简要说明 LU 算法加速器中的分解单元 PE 和其中主要模块 Axpy 和 Divider 的结构。发送单元 PE0 和写回单元 PEn 的结构比较简单,不作介绍。

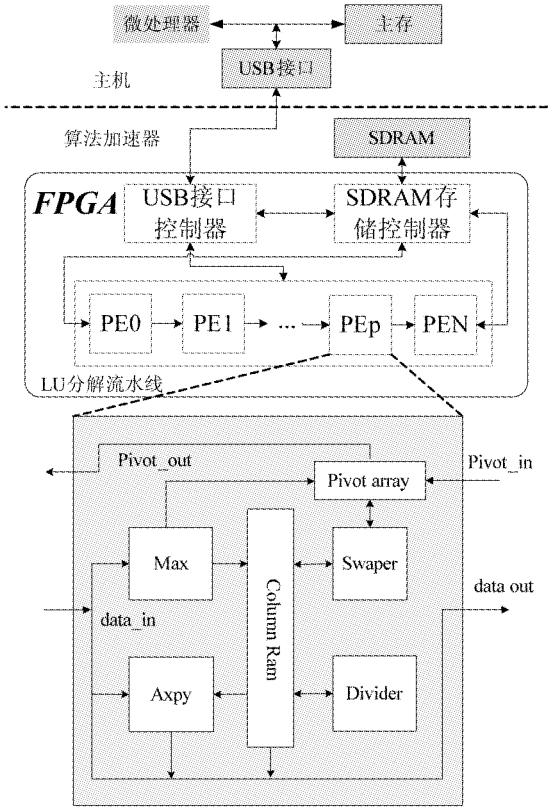

图 2　LU 分解流水线的硬件结构和 PE 内部结构

3.1　PEj 整体结构

每个流水线 PE 由 Swapper, Divider, Axpy, Max, Piovt array, Column Ram 等功能部件组成。矩阵列元素通过 data _ in 和 data _ out 通道在 PE 中流动,列主元行信息通过 Pivot _ in 和 Pivot _ out 通道逆向传递。每个浮点部件的设计都遵照 IEEE-754 浮点数标准。下面简要介绍每个部件的功能。

Column Ram 用来储存 PE 中的更新列,它是一个 32bit * N 的 RAM,其中 N 是流水线可以分解的最大的矩阵的边长,N = 1024 时 RAM 大小为 4kB。

Piovt array 是一个大小为 logN * P 的寄存器堆,用来记录流水线中驻留在各个 PE 中的已更新列的列主元信息。其中每个寄存器宽度为 logN,保存对应列主元所在行,即 RAM 索引地址,P 是流水线中 PE 个数。例如 pivot[i]中保存的就是 PEi 中已更新列的主元所在行。

Max 部件对由 data _ in 通道流入 PE 的更新列的元素进行列主元选取,并且同时记录此主元所在的行号 pivot,在更新列所有元素流入 PE 后,将 pivot 信息发送给上一个 PE,并保存在本 PE 的 pivot array 中。Max 在选主元的同时也将流入 PE 的元素按先入先出的顺序存入 Column Ram 中。

之后 Swapper 根据 Piovt array 中记录的列主元所在行信息对 Column Ram 中存储的更新列中的元素进行行交换。例如如果 pivot[i] = = j, 则 Column Ram 中所在列的第 i 行和 j 行元素进行交换。

交换之后,Divider 用主元 max 遍除更新 Column Ram 列中对角线以下的所有元素。

Axpy 部件对从 data _ in 通道中流入 PE 的处理列 j 作 axpy 操作。

3.2　Axpy 流水模块

Axpy 运算完成两个向量的乘加操作,结构如图 3所示。其流水线由一个三栈的浮点乘法器 Mult 接一个 13 栈的浮点减法器 Sub,和一个 13 级的浮点寄存器队列 a[i]组成。由 data _ in 通道流入 Axpy 部件的处理列 j 元素 Aji 先从 Axpy _ in 流入寄存器队列 a[i]中,并选取 j 列的第 k(k 为本 PE 编号)个元素 Ajk 作为 axpy 乘积因子保存在 scaler 寄存器中,对之后流进来的元素 Ajm($m > k$),Axpy 部件开始从 Column Ram 中读取对应元素 Akm 由 mem _ data 流入,与 scaler 一起流入 Mult 中作乘法。乘积再与 Ajm 流入 Sub 中作减法。不同的列元素选择乘法结果和已更新列元素对其作减法的时机不同,而列中不作 axpy 的元素部分直接从队列中流出。

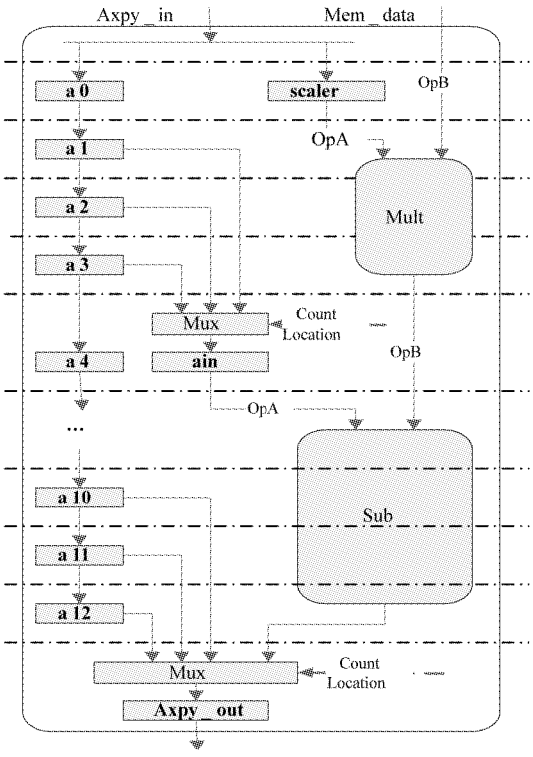

图 3　Axpy 部件结构

其中,减法部件是一个浮点迭代减法,乘法部件是对尾数分段相乘再相加的流水线。将 2 个要相乘的 24 位的尾数拆成 4 个 12bit 的高低位部分,用 4 个 12bit 的乘法器同时相乘后再将高低位相加,关键路径是最后对高低位乘积部分相加的 36bit 的加法。

3.3 Divider 流水模块

13 栈除法器的结构如图 4 所示,分为三个部分,分别对浮点除法的符号、指数和尾数进行计算。对尾数商的计算使用基为 4 的商选择迭代算法[9],对于 23 位的尾数需要 13 级迭代。关键路径是每次迭代时求余数的 28bit 减法。

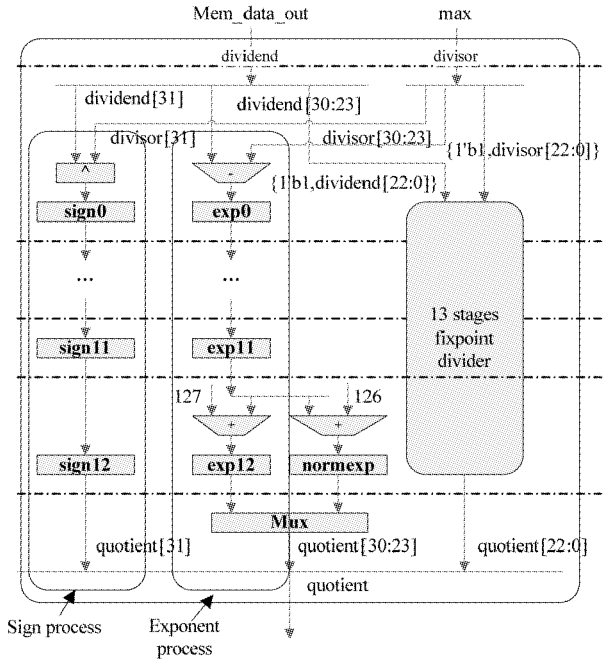

图 4 Divider 部件结构

4 实验与性能分析

4.1 实验环境

LU 分解算法的实现测试平台主要由一片 FPGA(EP2S130C5),一条容量为 1GB 的 SDRAM(MT16LSDT12864A),以及 USB2.0 接口控制器构成。主机配置为 Celeron(R) 3.07GHz CPU,512M 内存。

4.2 正确性验证

首先,使用软件模拟(ModelSim SE 6.1d)保证时序上的逻辑的正确性;其次,通过将分解后的矩阵从 SDRAM 中读回主机,和在主机上用软件分解的矩阵相比较来保证结果的正确性。

4.3 实验结果

4.3.1 综合性能

在 EP2S130C5 上实现 24 个 PE 的流水线,设定矩阵最大规模为 1k * 1k,即每个 PE 中保存一个 32bit * 1k = 4kB 的 RAM。24PE 流水线由 1 个发送单元 PE0,24 个处理单元 PEj,1 个写回单元 PEn 组成,其综合性能如表 1 所示。其中整个流水线系统与一个存控一起综合。由于系统采用与存控同步的时钟,故整个系统工作在存控设定的 96.01MHz 的水平。但 24PE 流水线可以综合到 137MHz。

表 1 24PE 流水线综合性能

	24PE 系统	PE0	PEj	PEN
ALUTs	91987 (87%)	799 (<1%)	3957 (4%)	412 (<1%)
Mem bit	944099 (14%)	66176 (<1%)	34209 (<1%)	32768 (<1%)
9-bit DSP	196 (39%)	2 (<1%)	8 (2%)	2 (<1%)
CLK	96.01MHz	160MHz	137MHz	157MHz

每个工作 PEj 中 axpy 部件的 3 级浮点乘法流水线由 4 个 12bit 的乘法器组成,需要用 8 个 9bit 的 DSP 部件搭建。关键路径是除法器中求余数的 28bit 定点减法。分解 1000 规模的矩阵,流水线中 RAM 并不是瓶颈而是 ALUTs。如果增加每个 PE 中的 RAM 大小,可以分解更大规模的矩阵。但是同时会增大 RAM 访问时间,降低频率。

4.3.2 分解性能

LU 分解算法中,每一个分解出来的已更新列 k 对其他列进行一遍更新后,矩阵中 k 行以上的元素都已经分解出来。假设流水线分解 PE 个数为 p,在每次迭代的时候,流水线都分解出来 p 列。这样在每次迭代 i,只要读每列的 $n\text{-}ip$ 行以下的元素。

这样,流水线在计算过程中一共需要读入的元素个数为 $\sum_{i=0}^{\lfloor \frac{n}{p} \rfloor}(n-ip)^2+(n-\lfloor \frac{n}{p} \rfloor p)^2$。

当然也要写入这么多元素。这个次数与 p 成反比,在 n 较大时,与如果 n^3 成正比。如果恰好 n 能被 p 整除,则这个数字为

$$F(p)=\frac{p^2(\frac{n}{p})(\frac{n}{p}+1)(\frac{2n}{p}+1)}{6}=\frac{2n^3+3n^2p+p^2n}{6p}$$

在设计的时候,将发送 PE0 设计为双缓存,这样在从 SDRAM 读取一列时,PE0 可以将缓存的另一列交换后直接发送,由于流水线长度相矩阵规模较

小,而读取写回数据又相对计算较慢,这样就隐藏了计算时间。流水分解的速度也就相当于从外存中读取和写回上述元素的时间。

24PE 流水线分解性能与算法 1 在 Celeron(R) 3.07GHz CPU 的主机上运行比较如表 2 所示。

表 2　24PE 流水线分解性能

矩阵规模	微机	流水线	加速比
400	0.17s	26.8ms	6.3
500	0.341s	49.6ms	6.87
600	0.571s	81.6ms	7.0
700	0.921s	126.5ms	7.27
800	1.372s	185.4ms	7.4
900	1.912s	260.4ms	7.34
1000	2.584s	353.1ms	7.32

分解规模为 1000 的矩阵,算法 1 一共需要 666166500个浮点操作,用时 353.1ms,据此计算出整个流水线平均性能为 1.89GFLOPs。由于在 axpy 流水阶段,每个 PE 每拍作一个乘加操作,则 24PE 流水线工作在 96.01MHz 的峰值性能为 4.6GFLOPs。

根据前面对读写次数的分析可知,计算规模为 1000 的矩阵,算法需要从 SDRAM 中读写 28785k 个元素。不计每次访问存控的启动时间,仅访问这些元素,在 96.01MHz 下每拍读写一个元素,就需要 299.8ms。占整个分解时间的 84.9%。可见算法决定于读写次数和时间。

4.3.3　与其他 FPGA 实现的 LU 分解算法的比较

目前在可重构器件上有很多实现的 LU 分解算法,但是很少有可以进行部分选主元的高性能的 LU 分解算法。我们将本流水线算法,与已知的 Kieron Turkington[1]的选主元算法(其实现的 linpack1000 中的 dgefa 部分)和 Seonil Choi[2]的不选主元的算法,在效率、精度和资源占用情况方面分别进行比较。

首先比较运算精度及稳定性。由表 3 比较可知,只有我们的算法和 Kieron 的算法是支持单精度 32bit 运算且可以进行主元选取的,Seonil 的算法只有 16bit 精度且不支持主元选取,故其精度和稳定性都要差些,在遇到奇异矩阵的情况下,可能导致分解失败。

表 3　各 LU 算法精度及稳定性比较

	流水线	Kieron	Seonil
运算精度	32bit	32bit	16bit
可否选主元	可	可	不可
可否避免奇异矩阵分解失败	可	可	不可

其次,在资源占用情况上。Seonil 在 Xilinx 综合下占用 3550 单位 slice, 32 个 Mult 单元,是比较的算法中相对占用资源最少的。但是其原因也很明显,首先其精度选取低,只有 16bit,而我们的算法是 32bit 的。其次,它不进行选主元操作,而是一遍扫描式的分解,故逻辑简单。而从表 4 可以看到,若比较两个选主元的算法,我们的算法要比效率相当的选主元的 Kieron 的算法占用较少的资源(只是相比 28PE 的 Kieron 算法,ALUTs 稍多一点,DSPs 多一些)。

表 4　选主元 LU 算法占用资源情况比较

	24PE 流水线	36PE Kieron	28PE Kieron
Device	EP2S130	EP2S180	XV4LX200
ALUTs(slices)	91987	135487	89086
Mem bits	944099	1920000	1824000
DSPs	196	296	96

表 5 给出了各 LU 算法分解规模 1000 矩阵的效率的比较,从中可以看到,我们的 24PE 流水线基本可以达到 Kieron 算法 28PE 流水线的效率,但我们却还可以以较低的器件得到较高的综合频率(如果不算存控,可以综合到 137MHz),这是由于我们使用的是运算级的细粒度的并行分解方法,而 Kieron 只是功能级的粗粒度分解。虽然相比不选主元的 Seonil 算法,选主元算法不但在频率上还是在分解效率上,是要相对差些,但仍在一个数量级之内(353.1 比 182.5)。这也是由于不选主元相比选主元逻辑上要简单很多,而且其不需要多次迭代。

表 5　各 LU 算法分解规模 1000 矩阵效率比较 (Seonil 分解 1024)

	24PE 流水线	28PE Kieron	36PE Kieron	Seonil
频(MHz)	137	42.1	44.4	120
延时(ms)	353.1	330	260	182.5
是否选主元	是	是	是	否

但从上面的各方面比较来看,我们的流水线算法在提高运算精度,使用选主元方法确保算法稳定性的同时,依然可以对矩阵进行高效率的分解。

本实验使用的是一个连接单端口 SDRAM 的存控接口。在流水线需要同时读写 SDRAM 的时候会发生冲突。如果将 PE0 和 PEn 分别连接两个存控接口,就可以把写回元素的时间隐藏,只计读出时

间。由于读写元素的个数相等,这样可以隐藏一半的访问时间。而由于算法效率主要取决于读写时间,这样作便可以极大提高分解性能。在模拟规模为 800 的 24PE 分解中,双存控分解只需时 130.4ms,比现在单存控 185.4ms 可以提高性能 29.7%。

5 结 论

本文提出一种可以进行列主元选取的细粒度 32bit 精度的 LU 分解流水线算法,并用 FPGA 实现。此算法以整列块为分解单位,将用来对矩阵进行更新的列存储于 PE 中从而实现了列主元的选取并利用了数据的可重用性。算法将交换操作在流水前集中进行,实现了 axpy 运算的全流水。算法的性能取决于读写的次数。24PE 的 LU 分解流水线与 Celeron (R) 3.07GHz 通用处理器主机相比可以得到平均 6 到 7 倍的加速比。相比其他不选主元的 LU 分解相比,结果更加精确,稳定性更高。相比其他选主元的 LU 分解算法,此算法可以在相对占用更少资源的情况下,得到更高的综合频率和相当的运算效率。

参考文献

[1] Turkington K J, Masselos K, Constantinides G A, et al. FPGA acceleration of the LINPACK benchmark: a high level code transformation approach. In: Proceedings of the 16th International Conference on Field Programmable Logic and Applications (FPL 2006). Madrid, Spain: Springer, 2006.375-380

[2] Choi S. Time and energy efficient matrix factorization using FPGAs. In: Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL 2003). Lisbon, Portugal: Springer, 2003.507-519

[3] Casseau E, Degrugillier D. A linear systolic array for LU decomposition. In: Proceedings of the 7th International Conference on VLSI Design (VLSI Design 1994). Calcutta, India: IEEE Computer Society, 1994.353-358

[4] Navarro J J, Llaberia J M, Numez F J, et al. LU-decomposition on a linear systolic array processor. *International Journal of Mini and Microcomputers*, 1989, 11(1):4-8

[5] Barbosa J, Morais C N, Padilha A J. Simulation of data distribution strategies for LU factorization on heterogeneous machines. In: Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS 2003). Nice, France: IEEE Computer Society, 2003.103

[6] Toledo S. A survey of out-of-core algorithms in numerical linear algebra. In: External Memory Algorithms. Boston, MA, USA: American Mathematical Society, 1999.161-179

[7] Dongarra J, Hammarling S, Walker D. Key concepts for parallel out-of-core LU factorization. *Computers and Mathematics with Applications*, 1998, 35:13-31

[8] Parhi K K, Srinivas H R. A fast radix-4 division algorithm and its architecture. *IEEE Transactions on Computers*, 1995, 44(6):826-831

A pipelined parallel LU decomposition algorithm with column partial pivoting

Niu Xin, Zhou Jie, Dou Yong, Lei Yuanwu

(Department of Computer Science, National University of Defense Technology, Changsha 410073)

Abstract

This paper presents a fine-grained pipeline algorithm for LU decomposition with column partial pivoting and gives the description of its implementation on field-programmable gate arrays (FPGA). The pipeline algorithm makes full use of the data reuse property of the LU decomposition during the column partial pivoting in order to reduce the I/O cost. Since the critical functions are pipelined in fine-granularity, the decomposition performance can be improved. The experimental result shows that the computing speed can be 6 times higher than that of the software execution of the serial algorithm on Celeron(R) 3.07GHz. Compared with other FPGA implementations, the proposed design has the better computational accuracy and stability due to the pivoting scheme, while demanding less resource and keeping the high efficiency.

Key words: LU decomposition, pipeline, parallel algorithm, partial pivot, field-programmable gate arrays (FPGA)