

# 浙江大学

## 本科生课程作业

### 并行矩阵乘法设计

课程名称:	并行计算与多核编程
姓名:	陈卓
学院:	计算机科学与技术学院
专业:	计算机科学与技术
学号:	3170101214
指导老师:	楼学庆

2020 年 4 月 17 日

## 1 设计原理

本实验分别采用 Cannon 算法和 DNS 算法实现  $4 \times 4$  矩阵乘法。

### 1.1 Cannon 算法

对于  $N \times N$  矩阵乘法  $A \times B$ ，本实验 Cannon 算法采用**二维网孔结构**。步骤如下：

- i)  $A_{ij}$  向左循环移动  $i$  步； $B_{ij}$  向上循环移动  $j$  步<sup>1</sup>；各结点清零；
- ii) 各结点做乘法运算并累加；
- iii)  $A_{ij}$  向左循环移动 1 步； $B_{ij}$  向上循环移动 1 步；
- iv) 重复  $N - 1$  次。

### 1.2 DNS 算法

本实验使用**超立方体结构**实现 DNS 算法。步骤如下：

- i)  $A, B$  同时在  $k$  维复制： $A[i, :, :] = A, B[i, :, :] = B$ ；
- ii)  $A$  在  $j$  维复制： $A[i, j, :] = A[i, i, :]$ ；
- iii)  $B$  在  $i$  维复制： $B[i, :, j] = B[i, :, i]$ ；
- iv) 各结点做乘法运算；
- v) 将  $k$  维各结点求和。

## 2 设计思路与实现

### 2.1 Cannon 算法

根据 1.1 中的描述，本实验将  $4 \times 4$  矩阵分块为 16 个结点，每个结点处理两个数的乘法运算。

#### 2.1.1 cannon\_unit 模块

首先设计出表达每个结点的 cannon\_unit 模块。

```
1 module cannon_unit(  
2     input wire clk,  
3     input wire rst,  
4     input wire en,  
5     input wire a_init[7:0],  
6     input wire b_init[7:0],  
7     input wire a_in[7:0],  
8     input wire b_in[7:0],
```

---

<sup>1</sup> $i, j \in [0, N)$

```

9  output wire a_out[7:0],
10 output wire b_out[7:0],
11 output reg s[7:0]
12 );

```

其中 `rst`, `a_init`, `b_init` 用于步骤 i) 的初始化和清零; `a_in`, `b_in`, `a_out`, `b_out` 用于在二维网孔结构中连接其它结点; `s` 用于输出; `en` 用于计算完成后阻断后续计算。

在 `cannon_unit` 模块中有三个寄存器 `a`, `b`, `s`, 分别用于保存  $A, B$  矩阵的元素和结果。在每个时钟周期内, 寄存器 `a`, `b` 接收来自输入 `a_in`, `b_in` 的值, 同时将二者乘积累加在寄存器 `s` 中; 若 `rst` 置位, 则 `s` 清零, `a`, `b` 接收 `a_init`, `b_init` 的值。

### 2.1.2 cannon 模块

`cannon` 模块是一个状态机。根据 Cannon 算法设计 4 个状态, 分别是 `IDLE`, `LOAD`, `CALC`, `END`。状态转移如图 2.1 所示。

```

1 module cannon(
2   input wire clk,
3   input wire rst,
4   input wire A[3:0][3:0][7:0],
5   input wire B[3:0][3:0][7:0],
6   output wire S[3:0][3:0][7:0],
7   output reg finished
8 );

```

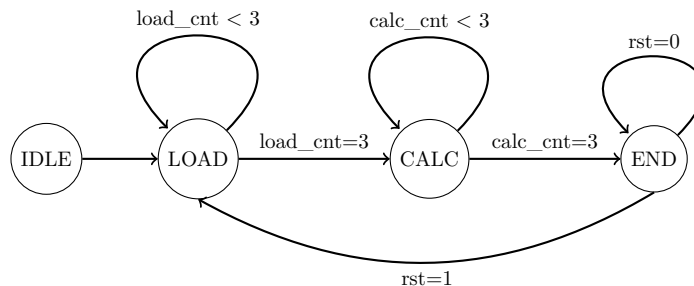


图 2.1: `cannon` 模块状态图

每个状态的动作如表 2.1 所示。

详细代码见 `cannon.v`。

表 2.1: cannon 模块状态表

状态	描述
IDLE	模块初始化
LOAD	加载 $A, B$ 矩阵, 并初始化各结点
CALC	循环位移计算
END	计算结束

## 2.2 DNS 算法

### 2.2.1 dns\_unit 模块

首先设计超立方体的每个结点模块。

```

1 module dns_unit(
2   input wire clk,
3   input wire rst,
4   input wire broadx[7:0],
5   input wire broady[7:0],
6   input wire broadz[7:0],
7
8   output reg a[7:0],
9   output reg b[7:0],
10  output reg s[7:0]
11 );

```

其中 **broadx**, **broady**, **broadz** 用于三个方向上的播送, **a**, **b**, **s** 存储矩阵数据和结果。模块是一个状态机, 状态转移如图 ?? 所示。

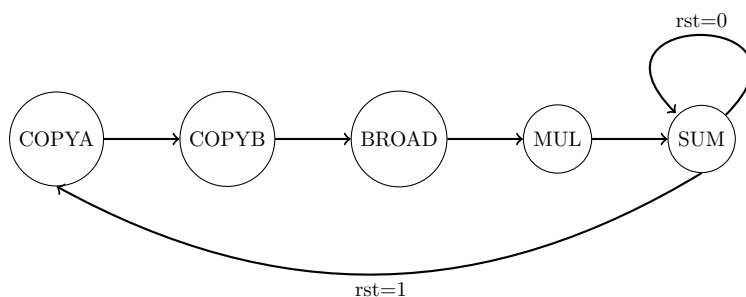


图 2.2: dns\_unit 模块状态图

每个状态描述如表 2.2 所示。

表 2.2: dns\_unit 模块状态表

状态	描述
COPYA	a = broadz
COPYB	b = broadz
BROAD	a = broady b = broadx
MUL	s = a * b
SUM	-

### 2.2.2 dns 模块

```

1 module dns(
2     input wire clk,
3     input wire rst,
4     input wire A[3:0][3:0][7:0],
5     input wire B[3:0][3:0][7:0],
6     output reg S[3:0][3:0][7:0],
7     output reg finished
8 );

```

dns 模块包含 64 个 dns\_unit 模块，以如下形式插入。

```

1 reg [7:0] broadx[3:0][3:0];
2 reg [7:0] broady[3:0][3:0];
3 reg [7:0] broadz[3:0][3:0];
4
5 wire [7:0] a[3:0][3:0][3:0];
6 wire [7:0] b[3:0][3:0][3:0];
7 wire [7:0] s[3:0][3:0][3:0];
8
9 dns_unit m_zyx(
10     .clk(clk_unit),
11     .rst(rst_unit),
12     .broadx(broadx[y][z]),
13     .broady(broady[z][x]),
14     .broadz(broadz[x][y]),
15
16     .a(a[z][y][x]),
17     .b(b[z][y][x]),
18     .s(s[z][y][x])
19 );

```

此外 dns 模块同样用状态机实现，且状态转移与 dns\_unit 相同，其状态描述如表 2.3 所示。

表 2.3: dns 模块状态表

状态	描述
COPYA	$\text{broadz}[i][j] = A[i][j]$
COPYB	$\text{broadz}[i][j] = B[i][j]$
BROAD	$\text{broady}[i][j] = a[i][i][j]$
	$\text{broadx}[i][j] = b[j][i][j]$
MUL	-
SUM	$S[i][j] = \text{sum}(s[:,j][i])$

由此，dns 模块在每个状态为播送通道（broadx，broady，broadz）决定不同的值，以达到实现 DNS 算法的目的。

### 3 评价

#### 3.1 Cannon 算法

设  $k^2$  个结点，记单次传输时间  $t_r$ ，单次计算时间  $t_w$ ，由于步骤 ii) 和 iii) 同时进行，且通常  $t_r \ll t_w$ ，故总时间

$$T = (k - 1)t_r + kt_w$$

此外，若单个结点中分块矩阵的乘法采用串行算法，则

$$\begin{aligned}
 T &= (k - 1)O(1) + kO\left(\left(\frac{n}{k}\right)^3\right) \\
 &= O(k) + O\left(\frac{n^3}{k^2}\right) \\
 &= O(\sqrt{p}) + O\left(\frac{n^3}{p}\right) \\
 &= O\left(\frac{n^3}{p}\right)
 \end{aligned}$$

在本实验中，每个结点仅负责一阶矩阵的运算，故  $T = O(\sqrt{p})$ 。

Cannon 算法的不足之处在于，数据初始化需要  $k - 1$  次的循环位移，在本实验中，由于规模不大，可以通过直接在代码中交换输入实现加速，但在大规模矩阵运算中仍然有这一方面的劣势。

#### 3.2 DNS 算法

相较于 Cannon 算法，DNS 算法不仅使用了更多的结点 ( $n^3$ )，同时加入了**播送**的特性。通过播送，类似 Cannon 算法中初始化的步骤可以在一个通信周期内完成，使得所有数据传送并行进行，大大加快了计算速度。此外，每个结点的乘法运算也是并行进行。因此，DNS 算法复杂度主要集中在步骤 v) 求和阶段。结果矩阵的每一个元素并行地计算  $n$  个数的和，而计算  $n$  个数的和也可以在超立方体的框架下并行，因此复杂度为  $O(\log n)$ 。

## 4 $N \times N$ 矩阵乘法设计

### 4.1 Cannon 算法

Cannon 算法是容易推广的——尽管本实验每个结点仅处理两个数的运算，但在实际使用中可以令每个结点处理两个小矩阵的运算，即 Cannon 算法描述中的**分块**。然而如此一来，如何对大矩阵进行分块，分块方式不同对运算复杂度是否有影响，就成为应该思考的问题。此外，本实验使用 verilog 设计  $4 \times 4$  矩阵乘法，而对于高阶矩阵乘法，我们为它不可能设计新的硬件，应该以小模块为基础，利用高层的并行算法（如 MPI）实现高阶矩阵乘法。

假设已有用 Cannon 算法计算  $S$  阶矩阵乘法的结点  $P$ 。由于任何  $S'(< S)$  阶矩阵，我们可以通过补零的方式用  $P$  计算。因此对于  $N(> S)$  阶矩阵，我们可以将其补成  $N'$  阶矩阵，其中

$$N' = \lceil \frac{N}{S} \rceil S = kS$$

此时通过使用  $k^2$  个  $P$  结点，就可以组成更高层的 Cannon 算法（称为**法一**）。下面与硬件直接实现的  $N$  阶矩阵乘法（称为**法二**）比较。

记循环移动的次数、乘法累加的周期数分别为  $r, w$ 。设法一不同结点之间的带宽是  $S$  阶矩阵的大小，则需要  $k$  个周期初始化移动；接下来的  $k - 1$  次循环移动计算中，每次都需要  $P$  结点内  $S$  个周期初始化，并有  $S - 1$  次结点内的循环移动计算。则有

$$\begin{aligned} r_1 &= k - 1 + k(2S - 1) \\ &= 2N' - 1 \\ w_1 &= kS \\ &= N' \\ r_2 &= 2N - 1 \\ w_2 &= N \end{aligned}$$

从以上可以看出，使用法一所带来的性能损失仅为  $O(S)$ ，因此我们可以使用法一扩展 Cannon 算法。

### 4.2 DNS 算法

DNS 算法也可以像 Cannon 算法那样推广。

假设已有用 DNS 算法计算  $S$  阶矩阵乘法的结点  $P$ 。易得

$$\begin{aligned} T &= O(\log S) + O(\log k) \\ &= O(\log(\lceil \frac{N}{S} \rceil S)) \end{aligned}$$

即使用 DNS 结点组成高层 DNS 网络带来的性能损失为  $O(\log(S))$ ，代价依旧很小。因此我们也可以使用这样的方法扩展 DNS 算法。