第 5 章

GPGPU 运算单元架构

GPGPU 的巨大算力源于内部大量的硬件运算单元。这些运算单元可分为多种类型，例如在 NVIDIA 的 GPGPU 中，存在为通用运算服务的 CUDA 核心单元(CUDA core)、特殊功能单元(Special Function Unit，SFU)、双精度单元(Double Precision Unit，DPU)和张量核心单元(tensor core)。数量巨大、类型多样的运算单元成为 GPGPU 架构不同于 CPU 的显著特点。GPGPU 以可编程多处理器为划分粒度，将各种类型的运算单元按照一定比例分组并组织在一起，从而支持通用计算、科学计算和神经网络计算等各种场景下多种多样的数据处理需求。

本章将介绍 GPGPU 运算单元架构，包括支持的数据类型及多种运算单元的基本结构和组织方式。

5.1 数值的表示

以晶体管的开关特性为基础，绝大部分处理器都以二进制的方式存储和处理数据。数据根据是否有小数点可分为整数和小数，在计算机中采用整型数(integer number)和浮点数(Floating Point number，FP)来表示。本节将介绍常用的整型数和浮点数表示方法，并在此基础上讨论近年来新出现的一些浮点数表示方法。

5.1.1 整型数据

整型数据是不包含小数部分的数值型数据，采用二进制的形式表达。

由于计算和存储硬件的限制，计算机只能以有限的位数原生地表示数据，这意味着可表示的整型数据范围是有限的。例如，使用 8 比特能表达的范围为 $0000\ 0000_2$～$1111\ 1111_2$，即十进制中 0～255[①]。如果需要表达的整型数据超过这一范围，就会存在偏差，这就是有限字长效应。有限字长效应不仅体现在计算上，而且数据的存储和传输都会受到限制。

① 完整表示应为 0_{10}～255_{10}。为简化表述，后文对十进制数据的表示如非必要不再添加下标。

无符号整型数据在计算机上表达的方式较为简单，将十进制整型数据直接转化为二进制数据即可。但在大部分情况下，整型数据存在正数和负数之分，整型数据的计算也需要符号位的参与，因此表示有符号的整型数据非常重要。整型数据的编码方式主要有三种：原码、反码和补码。

原码的编码方式为符号位加真值的绝对值，即第一位表示符号，其余位表示绝对值。一般情况下，第一位为0代表正数，为1代表负数。例如，对于8比特的二进制表达形式，+1的原码为$0000\ 0001_2$，-1的原码为$1000\ 0001_2$，表示范围为[$1111\ 1111_2$，$0111\ 1111_2$]，即十进制的[-127，127]。虽然原码的编码方式非常符合人的直觉，但并不适用于计算机。由于在计算机中，加减法是最基本的运算，人们希望通过复用加法电路也能计算减法，省去额外的减法器电路。为实现这一计算方式，就要求编码的符号位也参与计算。如果想要计算1-1，计算机的等价计算为1+(-1)，采用原码计算就是$0000\ 0001_2+1000\ 0001_2=1000\ 0010_2=-2_{10}$，这显然是错误的。

为了解决原码带符号计算的问题，出现了反码的编码方式。反码表达正数与原码一致，例如+1的反码仍为$0000\ 0001_2$。表达负数时，在原码的基础上要求除符号位外按位取反，例如-1的反码为$1111\ 1110_2$。因此，8比特反码的二进制表示范围为[$1000\ 0000_2$，$0111\ 1111_2$]，即十进制[-127，127]。反码的符号可以直接参与运算，如果想要计算1-1，按照反码方式，计算机需要计算$0000\ 0001_2+1111\ 1110_2=1111\ 1111_2=-0_{10}$。虽然反码的符号位参与计算仍然可以得到正确的结果，但会产生新的问题。在反码中，$1111\ 1111_2$表示-0，$0000\ 0000_2$表示+0，这意味着0的表示出现了冗余。

为了解决冗余问题，人们又提出了补码的编码方式。补码表达正数与原码和反码一致，表达负数则要求在反码的基础上加1。例如，-1的补码形式为$1111\ 1111_2$。相比于原码和反码，同位宽情况下补码表示的范围更大。例如，8比特补码的二进制表示范围为[$1000\ 0000_2$，$0111\ 1111_2$]，即十进制[-128，127]。补码的符号同样可以参与运算，如果想要计算1-1，按照补码方式，计算机需要运算$0000\ 0001_2+1111\ 1111_2=0000\ 0000_2=0_{10}$。补码把原先反码中$1000\ 0000_2$的冗余消除，并且可以表示为十进制-128，因此表示范围相比反码更大。

上面的例子通过三种编码方式的比较解释了计算机选择补码的原因，现代计算机中普遍采用补码形式表示整型数据。

5.1.2 浮点数据

在计算机科学中，浮点数是一种对实数数值的近似表示。由于实数是稠密的，计算机的数据受到有限字长的限制，因此浮点数也无法完全表示所有的实数，只能是一种有限精度的近似。

IEEE二进制浮点数算术标准(IEEE 754)是自20世纪80年代以来使用最广泛的浮点数标准，它规定了浮点数的格式、特殊数值的表示、浮点运算准则、舍入规则及例外情况的处理方式。经过后续不断地完善和补充，IEEE 754目前主要规定了半精度浮点(16位，

FP16)、单精度浮点(32 位,FP32)和双精度浮点(64 位,FP64)等不同长度浮点数的标准。IEEE 754 标准浮点数被广泛应用于各类浮点计算过程中,其中 FP16、FP32 及 FP64 分别在人工智能、通用计算和科学计算中应用最为广泛。

1. 浮点数的格式

IEEE 754 标准浮点数的格式如图 5-1 所示。所有精度的浮点数表示都被分为三个部分:符号位(sign,s),指数位(exponent,e)和尾数位(fraction,f)。借助这三个字段,二进制浮点数均可以表示成$(-1)^s \times 1.f \times 2^{e-b}$ 的形式。

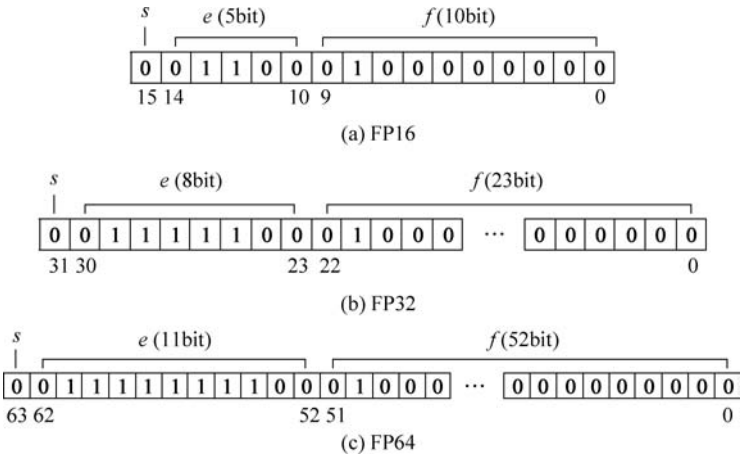

图 5-1　IEEE 754 标准所定义的浮点数格式

(1) 符号位 s 表示一个浮点数的符号,当 s 为 0 时表示正数,否则为负数。

(2) 指数位 e 表示以 2 为基(带偏移的)的幂指数。在 IEEE 754 标准中,e 并不直接表示幂指数,需要减去一个偏移量(bias,b)。b 的选取与 e 的位宽有关,即 $b=2^{\text{len}(e)-1}-1$。这是由于直接以 2 为基的二进制数表示幂指数可能为负值,代表小于 1 的浮点数,所以在增加偏移量 b 后,e 可以用无符号整型数表示浮点数所有的幂指数,使得浮点数之间的大小比较更加便利。e 为 0 或全 1 时,表示特殊值或非规格化的浮点数。

(3) 尾数 f 表示浮点数中的有效数据。它舍弃了最高位的 1,而仅记录首个 1 后面的数据,从而可以多表示一位有效数据。

不同精度浮点数的差异在于指数位和尾数位字段的位数。FP16、FP32 和 FP64 就是这三个字段的不同组合。

(1) FP16 包含 1 位标志位,5 位指数位,10 位尾数,指数偏移量为 15。如图 5-1(a)中例子所示,其表达的浮点数真值为 $1.01_2 \times 2^{0\ 1100-15}=0.156\ 25_{10}$。

(2) FP32 包含 1 位标志位,8 位指数位,23 位尾数,指数偏移量为 127。如图 5-1(b)中例子所示,其表达的浮点数真值为 $1.01_2 \times 2^{0111\ 1100-127}=0.156\ 25_{10}$。

(3) FP64 包含 1 位标志位,11 位指数位,52 位尾数,指数偏移量为 1023。如图 5-1(c)中例子所示,其表达的浮点数真值为 $1.01_2 \times 2^{011\ 1111\ 1100-1023}=0.156\ 25_{10}$。

2. 特殊数值的表示

除上述规格化数外，IEEE 754 浮点数标准利用指数位的不同数值，还可以表示四种类型的特殊数据，具体如表 5-1 所示。

表 5-1 IEEE 754 浮点数表示

形　式	指　数	小数部分
零	0	0
非规格化形式	0	非 0
规格化形式	$1\sim2^e-2$	任意
无穷(∞)	2^e-1(全 1)	0
NaN	2^e-1(全 1)	非 0

(1) 当指数位为 0 时，如果尾数为 0，则表示浮点数为 0。由于符号位可能为 0 或 1，IEEE 754 浮点数可以表示正 0 或负 0。

(2) 对于规格化浮点数，其指数偏移量最小为 1。以 FP32 为例，其指数的最小值为 −126。这意味着规格化浮点数的绝对值(与零点的距离)最小为 1.0×2^{-126}，那么为了表示 0 与 1.0×2^{-126} 之间的数值，可以采用非规格化浮点数。IEEE 754 标准规定非规格化浮点数指数位为 0，表示非规格化浮点数值为 $(-1)^s\times0.f\times2^{-126}$。

(3) 当指数位为全 1，如果尾数部分为 0，则表示无穷(∞)。根据符号位的不同，可以表示为正无穷和负无穷。

(4) 当指数位为全 1，若尾数非 0，则表示为非合法数(Not a Number，NaN)。

3. 舍入和溢出方式

由于浮点数能表示的数值是有限的，它往往是一个无法表示的数值的近似。为了能够给出最接近实际数值的浮点表示，IEEE 754 规定了四种舍入模式，为编程人员提供合适的近似策略，如表 5-2 所示。

表 5-2 IEEE 754 的舍入规则

舍入模式	舍　入　前	舍　入　后	描　述
就近舍入	1.100_0111	1.100	即向最接近的数舍入，如果处于两个最接近的数中间则根据保留位最低位舍入。默认舍入方式
	1.100_1011	1.101	
	1.100_1000	1.100	
	1.011_1000	1.100	
向 0 舍入	1.100_1011	1.100	直接舍弃低位
	−1.100_1001	−1.100	
向正无穷舍入	1.100_1011	1.101	正浮点数进位，负浮点数舍弃低位
	−1.100_1011	−1.100	
向负无穷舍入	1.100_1011	1.100	正浮点数舍弃低位，负浮点数进位
	−1.100_1011	−1.101	

(1) 就近舍入。如果就近的值唯一,则向其最接近的值舍入。如果处于两个最接近的数中间,则需要舍入到偶数。一般可以通过三个位来判断舍入情况,即保护位(guard bit,G),舍入位(round bit,R)和黏着位(sticky bit,S)。G 是舍入后的最后一位,R 则是被舍弃的第一位,S 一般情况下代表 R 后面被舍弃的部分。对于就近舍入而言,如果 R 为 0 则全部舍弃;如果 R 为 1 且 S 不为 0,则进位。如表 5-2 所示,假设舍入下画线后 4 位,由于 1.100_0111 的 R 为 0,则舍去低位尾数。而 1.100_1011 的 R 为 1,S 不为 0,则需要进位。如果就近的值不唯一,即 R 为 1,S 为 0,那么要看舍入后结果 G 是否是偶数。如果是偶数则直接舍去后面的数不进位,如果是奇数则进位后再舍去后面的数。例如,1.100_1000 的 G 为偶数,则直接舍入为 1.100。1.011_1000 的 G 为奇数,则需要先进位,再进行舍入,结果为 1.100。

(2) 向 0 舍入。本质上为将低位全部舍去。这种舍入方法无论正负,舍去低位尾数即可。例如,1.100_1011 和−1.100_1001 均需要舍去低位尾数。

(3) 向正无穷舍入。即使得舍入后的浮点数比舍入前大,表现为正浮点数进位,负浮点数舍弃低位。例如,1.100_1011 需要进位,−1.100_1011 需要舍弃尾数。

(4) 向负无穷舍入。即使得舍入后的浮点数比舍入前小,表现为正浮点数舍弃低位,负浮点数进位。例如,1.100_1011 需要舍弃尾数,−1.100_1011 需要进位。

舍入通常发生在尾数计算操作之后。例如,在执行浮点数加法时,指数较小的浮点数需要进行右移,这样尾数相加后得到的位数会超过最终需要的位数,从而需要进行舍入。在舍入时,尾数部分的低位有可能就会丢失,从而产生误差。

溢出通常发生在指数运算完成后,两个浮点数相加或相减可能会导致尾数上溢出或下溢出。由于只能用有限的字长表示一个浮点数,因此对于一个过大或过小的浮点数则无法表示。例如,包含规格化浮点数在内的 FP32 表示范围为 $\pm 2^{-149} \sim \pm(2-2^{-23}) \times 2^{127}$,约等于十进制中 $\pm 1.4 \times 10^{-45} \sim \pm 3.4 \times 10^{38}$。当超过这个范围的上限,浮点数会发生上溢出(正上溢和负上溢),导致指数偏移量全为 1,那么只能表示成无穷或 NaN。非规格化数在一定程度上可以处理浮点数的下溢出(正下溢和负下溢),因为其指数偏移量全为 0。如果浮点数过小,那么只能表示为 0。对于上溢出,IEEE 754 规定如果指数超过最大值,返回 $+\infty$ 或 $-\infty$。如果发生下溢出,会返回一个小于等于该数量级中最小正规格化数的数。

5.1.3 扩展讨论:多样的浮点数据表示

IEEE 754 规范了浮点数的表示方法,但并不是唯一的标准。近年来,随着深度神经网络的发展和普及,人们发现 IEEE 754 标准浮点数并不完全适合用于神经网络计算。为此,许多公司如 NVIDIA、Google 和 Intel 通过对 IEEE 754 标准浮点数进行修改,提出了新的浮点数表示方法,在减小硬件开销和存储空间的同时,不会给神经网络计算带来明显的精度损失,从而优化推理和训练的时间。

1. BF16 格式

在 IEEE 754 标准中,FP16 只有 5 位指数位,动态范围太窄。为此,Google 公司在 2018

年提出了 BFLOAT16(BF16)试图解决这一问题。BF16 采用 8 位指数位,提供了与 FP32 相同的动态范围。如图 5-2 所示,相比于 FP32,BF16 截去尾数至 7 位,其余保持不变。

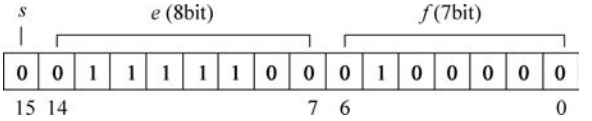

图 5-2 BF16 浮点数格式

BF16 计算浮点数真值的过程与 IEEE 754 标准一致,例如图 5-2 中的例子表达的浮点数真值为 1.01×2^{-3}。BF16 的格式可以快速地与 FP32 进行转换。在 FP32 转换为 BF16 格式时,FP32 的指数位被保留,而尾数截断至 7 位。在 BF16 转化为 FP32 格式时,由于两者标志位和指数位相同,直接在尾数后补充 0 直至 23 位即可。

BF16 首先被应用于张量处理器(Tensor Processing Unit,TPU)中。Google 公司认为,在神经网络计算中,浮点数指数位比尾数更加重要,因此 BF16 的性能相比于 FP16 更好,在 TPU 神经网络计算中逐渐取代 FP16。这是由于在神经网络训练过程中,激活和权重的张量数据大体在 FP16 数值表示的范围内,而权重更新很有可能小于 FP16 的表示精度,造成下溢出。为了解决这一问题,可以将训练损失乘以比例因子,即通过损耗缩放技术成比例放大梯度,缓解 FP16 的下溢出问题。BF16 其表示的范围和 FP32 相同,在训练和运行深度神经网络时几乎是 FP32 的替代品,将会大大缓解 FP16 的溢出问题。

从 Ampere 架构开始,NVIDIA GPGPU 的 Tensor Core 支持 BF16。

2. TF32 格式

BF16 虽然在 FP16 的基础上增大了浮点数的表示范围,但由于它和 FP16 均采用 16 位表示,不得不牺牲尾数的精度。这意味着相比于 FP16,BF16 在相同指数的情况下,表达二进制浮点数的精度更低。针对这一问题,NVIDIA 公司提出了另一种新的浮点数表示格式,称为 TensorFloat32(TF32)。如图 5-3 所示,它采用 19 位来表示浮点数。TF32 与 FP32 有相同的 8 位指数,与 FP16 有相同的 10 位尾数。

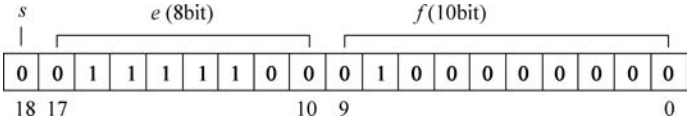

图 5-3 TF32 浮点数格式

TF32 计算浮点数真值的过程与 IEEE 754 标准浮点数一致,例如图 5-3 中的例子表达的浮点数真值为 1.01×2^{-3}。FP32 格式转换为 TF32 格式时,需要将尾数的低 13 位截去。而 TF32 格式转化为 FP32 格式时需要将尾数低位补 0 直至 23 位。

NVIDIA 公司的 GPGPU 从 Ampere 架构开始在张量核心上支持 TF32。支持 TF32 的张量核心借助 NVIDIA 库函数可以将 A100 单精度训练峰值算力提升至 156 TFLOPS,达到 V100 FP32 的 10 倍。A100 还可使用 FP16/BF16 自动混合精度(AMP)训练,通过微

量的代码修改，使得 TF32 性能再提高 2 倍，达到 312 TFLOPS。

3. FlexPoint 格式

神经网络以张量为基础进行运算，这些运算包含大量的张量乘和张量加，操作的数据格式大部分为 FP16 或 FP32。然而，这些数据格式可能并不适合大规模神经网络计算，因为大量的张量数据之间存在着较多冗余信息，对硬件面积、功耗、运算速度及存储空间占用来说都有不利的影响。

通过修改数据格式，文献[4]认为冗余信息的问题可以得到缓解。通过对基于 CIFAR-10 数据集训练 ResNet 过程中共计 164 个 epoch 的权重、特征图及权重更新数值的分布统计发现，这些数值有着较为集中的动态分布范围。如图 5-4 所示，epoch 0 和 epoch 164 的大部分权重数值都分布在 $2^{-8}\sim2^{0}$，大部分的特征图(激活)数值都分布在 $2^{-7}\sim2^{1}$。而 epoch 0 的大部分权重更新数值都分布在 $2^{-20}\sim2^{-12}$，epoch 164 的大部分权重更新数值分布在 $2^{-24}\sim2^{-16}$。集中的动态范围意味着相似的指数。换句话说，如果固定每个张量的指数，可以通过 16 比特的尾数来表示张量中每个数的精确值。因此，可以利用一种称为 FlexPoint 的新的数据格式来优化浮点数据的表示。

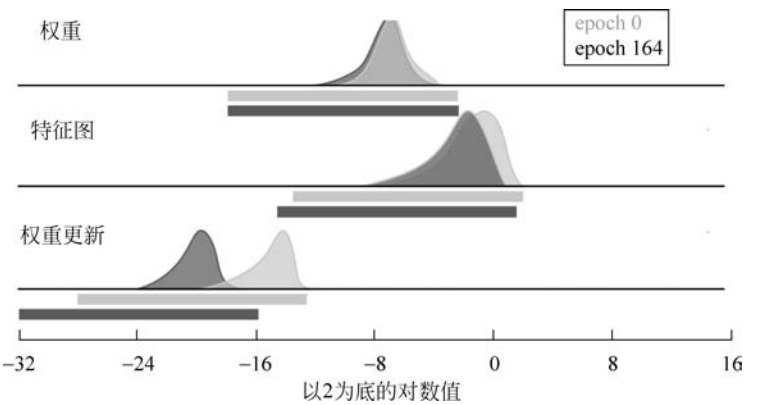

图 5-4　ResNet 训练过程中权重、特征及权重更新的分布

FlexPoint 具有 m 位尾数和 n 位指数。在一个张量中，每个数据都具有自己的 m 位尾数存储于设备端，而整个张量具有共同的 n 位指数存储于主机端。这种数据格式称为 FlexPoint $m+n$。图 5-5 给出了尾数为 16 而指数为 5，即 FlexPoint 16+5 的数据格式。可以看到，这种格式有两个优点。

(1) 张量内部各个数据之间求和与定点数求和一致，不需要考虑指数的影响，避免了浮点数求和中复杂的指数对齐过程。

(2) 张量之间乘积与浮点数乘积相同，这种乘积方式较为简单。

FlexPoint 也存在一定缺点。例如，要在硬件中有效地实现 FlexPoint，必须在两个张量运算之前确定输出张量的指数，否则就需要存储高精度的中间结果，这会增加硬件的开销。为了解决这一问题，该文献还提出一种指数管理算法 Autoflex。Autoflex 针对迭代优化算

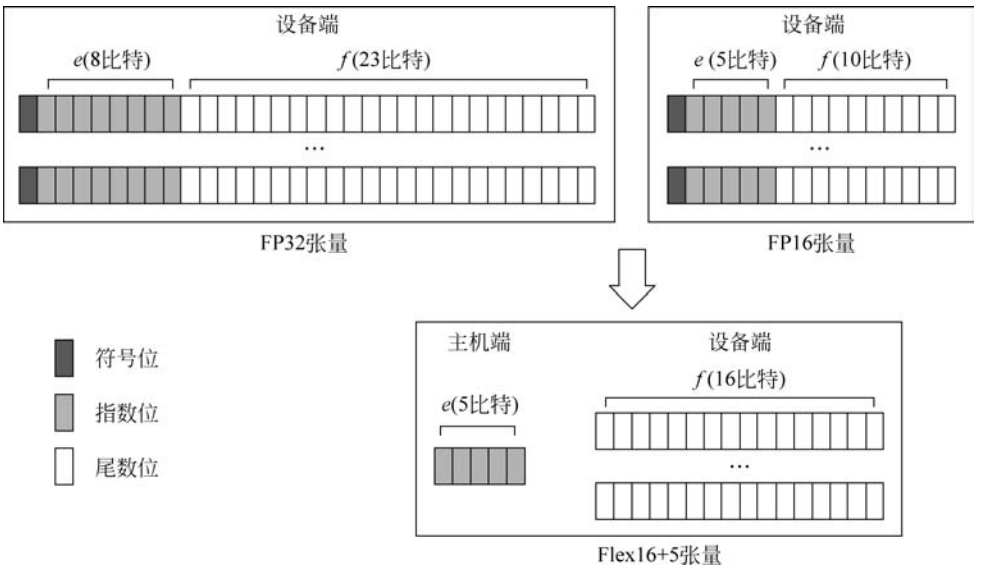

图 5-5 FlexPoint16＋5 的浮点数据格式

法，比如随机梯度下降算法，通过统计每次迭代中张量数据的指数变化来优化输出张量的指数。具体而言，对于输出张量 $\boldsymbol{T}$，Autoflex 会将 $\boldsymbol{T}$ 最近迭代产生的最大尾数存放于队列中。首先，判断队列中绝对值的最大值 a，如果 a 有上升趋势或向上越界，则直接增加整个张量 $\boldsymbol{T}$ 的指数。反之，如果 a 有下降趋势或向下越界，则减少指数的数值。其次，统计队列中所有尾数的标准差，根据标准差和最大值预测其增长趋势，并且预测下一次迭代中张量 $\boldsymbol{T}$ 可能出现的最大结果。最后，预测得到的最大结果转化为下一次迭代输出张量 $\boldsymbol{T}$ 整体的指数。FlexPoint 的假设基于神经网络训练中数值变化的过程较为缓慢。基于这种假设，Autoflex 通过张量变化的历史来预测未来输出张量的指数也是合理的。

5.2 GPGPU 的运算单元

运算单元是 GPGPU 实施计算操作的核心。在硬件上，NVIDIA 公司的 GPGPU 提供了 CUDA 核心单元、双精度单元、特殊功能单元及张量核心单元，解决不同场景的计算问题。CUDA 核心单元主要面向通用运算，如定点数的基本运算和浮点数的基本运算；特殊功能单元面向超越函数计算；张量核心单元面向矩阵运算，以应对大量的神将网络计算需求。本节将重点介绍除张量核心单元之外的运算单元结构。

5.2.1 整型运算单元

整型运算中应用较多的是算术加法、乘法和逻辑运算。

1. 整型数加法

在计算机中，正整数以原码方式存储，负整数多以二进制补码的方式存储。因此，通过整型加法可以直接带着二进制补码的符号进行运算，得到正确的结果。

对于两个整型数 X 和 Y，其二进制加法的一般步骤如下：

(1) 按位相加。从 X 和 Y 的低位开始逐位相加，低位得到的进位参与高位的相加，逐渐传递进位。

(2) 判断溢出。由于位数的限制，整型数只能表示一定范围的数值。如果加法运算的结果超过这一范围，则会导致溢出。

(3) 溢出处理，输入结果。不同的语言和编译器会采用不同的方式处理溢出。在C/C++语言中，通常会采用取模运算以保证结果非溢出。

根据整数加法的原理，一个简单的整型数加法器可以由全加器(Full Adder，FA)级联得到。例如，4比特整型数加法器可以由4个全加器计算得到。一个全加器的输入为 a、b 两个加数对应的两个比特位 a_x 和 b_x 及进位输入 c_x，输出为三者相加得到的结果 s_x 和进位输出 c_{x+1}。通过串行求解输入位的和并传递进位，可以完成所有位的计算。

这种简单的加法器称为行波进位加法器，原理简单，但由于串行进位导致速度较慢。为了提高加法的计算速度，需要更快的进位链传播，也由此产生了多种不同类型的加法器结构设计，如进位选择加法器、超前进位加法器等快速加法器结构。具体可参见文献[5]中关于加法器设计的介绍。

计算结果需要判断是否溢出。一般情况下，加法器通过最高位进位和次高位进位的异或结果进行判断。例如，对于 k 位整型数相加，会判断其 c_{k-1} 和 c_k。如果 c_{k-1} 为1而 c_k 为0，则产生正溢，即超过了整型数表示范围的最大值。若 c_{k-1} 为0而 c_k 为1，则会产生负溢，即超过了整型数表示范围的最小值。两种溢出情况下，c_{k-1} 和 c_k 异或运算的结果均为1。判断溢出后，进行溢出处理。如果运算结果有效，有些处理器还会检测其是否为负数或0，并设置处理器中相应的状态标志位。

2. 整型数乘法

整型乘法器实际上就是一个复杂的加法器阵列，因此乘法器代价很高并且速度更慢。但由于许多场景下的计算问题常常都是由乘法运算速度决定的，因此现代处理器都会将整个乘法单元集成到数据通路中。

对于一个二进制数乘法，假定乘数是 m 比特和 n 比特的无符号数，且 $m \leqslant n$。最简单的方法是采用一个两输入的加法器，通过不断地移位和相加，把 m 个部分积累加在一起。每个部分积是 n 位被乘数与 m 位乘数中的一位相乘的结果，这个过程实际上就是一个“与”操作，然后将结果移位到乘数的对应位置进行累加。这种方法原理简单，但迭代计算时间长，无法应用在高性能处理器中提供快速的乘法计算。

实际的快速乘法器则采用类似于手工计算的方式。利用硬件的并行性同时产生所有的部分积并组成一个阵列，通过将多个部分积快速累积得到最终的结果。这种方式比较容易映射到硬件的阵列结构上，因此也称为阵列乘法器。它一般集成了三个步骤：部分积产生、

部分积累加和相加。

(1) 部分积产生。部分积的个数取决于 m 位乘数中 1 的个数,可能有 m 个,也可能是 0 个。因此部分积产生一般会采用 Booth 编码,将部分积的数量减少一半。Booth 编码只需要一些简单的逻辑门,但可以显著地降低延时和面积。

(2) 部分积累加。部分积产生之后,需要将它们全部相加。阵列加法器实际上采用了多操作数的加法,消耗比较大的面积。更为优化的做法是以树结构的方式完成加法。利用全加器和半加器,通过反复地覆盖部分积中的点,把部分积的阵列结构转化成为一个 Wallace 树结构,同时减少关键路径的长度和所需要的加法器数目。

(3) 相加。这是乘法的最后一个步骤,加法器的选择取决于部分积累加树的结构和延时。加法器对于乘法的计算速度有直接的影响。

综合利用以上技术,结合细致的时序分析和版图设计可以实现高性能的乘法器,满足现代处理器的计算需求。

3. 逻辑与移位单元

除了算术运算,GPGPU 还提供基本的逻辑和移位运算。逻辑运算包括了二进制之间基本的与(and)、或(or)、异或(xor)、非(not)等运算,以及 C/C++ 中应用较为广泛的逻辑非(cnot)。逻辑非可通过判断源操作数是否为零来对目的操作数进行 0/1 赋值。移位运算包括拼接移位运算(shf)、左移(shl)、右移(shr)等。

5.2.2 浮点运算单元

浮点数运算单元能够根据 IEEE 754 标准处理单精度浮点数和双精度浮点数的算术运算,包括加法、乘法和融合乘加运算。

1. 浮点数加法单元

假设两个浮点数 X 和 Y,按照 IEEE 754 标准表示为 $X=1.f_x\times 2^x$,$Y=1.f_y\times 2^y$。假设 $x\geqslant y$,那么两个数加法运算的步骤如下。

(1) 求阶差:$x-y$。

(2) 对阶:$Y=1.f_y\times 2^{y-x}\times 2^x$,使两个数的指数相同。一般会将较小的那个数的小数点左移,避免移动较大数带来的有效数字丢失。

(3) 尾数相加:$f_x+1.f_y\times 2^{y-x}$。

(4) 结果规格化并判断溢出:$X+Y=1.(f_x+1.f_y\times 2^{y-x})\times 2^x$。得出的结果可能不符合规格化的要求,需要转换为规格化浮点数。之后,还需要对指数进行判断,可能出现溢出的情况导致浮点数出现特殊值。

(5) 舍入:如果尾数比规定位数长,则舍入。

(6) 再次规格化:舍入后可能会导致数据不符合规格化浮点数的标准,例如,1.111_1100_2 舍入后得到 10.000_2,这并不符合规格化浮点数的标准,所以需要再次规格化为 1.000×2^1。

图 5-6 显示了浮点数加法器的结构框图。与浮点数加法步骤第一步相对应，浮点数加法器硬件首先需要将输入的操作数 x 和 y 进行拆分。在拆分的过程中，硬件会检测输入浮点数的特殊值情况，如果其中存在 0、NaN 或无穷，那么可以直接输出另外一个操作数作为结果、输出 NaN 或无穷。

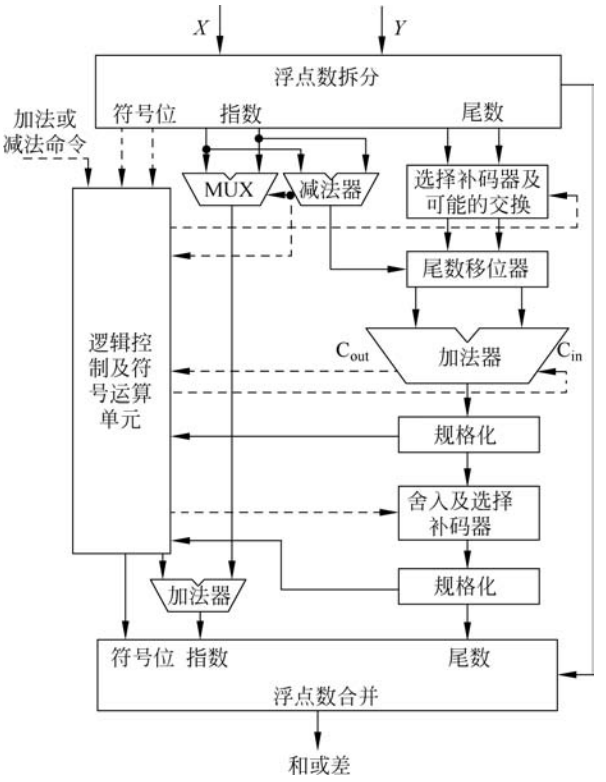

图 5-6　浮点数加法器的结构框图

拆分后的符号位会作为逻辑控制及符号运算单元的输入。符号位、指数偏移位比较和尾数计算过程中产生的结果会影响最终结果的标志位。指数偏移位比较大小，偏移位较大的会直接进入结果前的加法器中，与尾数计算中产生的进位相加，得到最终的指数偏移位结果。

尾数的计算比较复杂。两个尾数首先会进入选择补码器中，由于可能会出现不同符号数相加/减，根据控制逻辑选择补码器可能会将其中一个尾数转为补码。在设计选择补码器的过程中，为了减少硬件开销，一般只支持一个数进行补码操作，此时有可能需要两个尾数进行交换，完成指定尾数转换补码操作。根据尾数偏移位的减法结果在尾数移位器中进行对阶操作。对阶完成后，两个尾数可以直接在补码加法器中完成加法操作，得到的进位需要输入控制逻辑，用于最终结果的指数偏移位计算。

补码加法器得到的结果需要规格化，因为计算得到的结果可能出现进位，因此需要右移

一位。也可能得到的结果很小，在补码中表现为有超过两个的0或1存在于尾数的高位上，此时需要左移多位。

规格化结束后需要对尾数根据IEEE 754舍入标准进行舍入，通常情况下是就近舍入。具体可参见表5-2的说明。舍入后的结果也可能需要进行补码运算和规格化过程，并最后输入成为结果的尾数。

最终，浮点数加/减法器需要将输出的符号位、指数位移位和尾数位重新打包，并且判断结果是否存在溢出或出现了非规格化值情况。硬件加法器处理非规约浮点数的运算比较困难，很多情况下都需要软件方案的辅助。

2. 浮点数乘法单元

相比于浮点数加法，浮点数乘法的计算过程较为简单。这里依然以浮点数 $X=1.f_x\times 2^x$ 和浮点数 $Y=1.f_y\times 2^y$ 为例，介绍浮点数的乘法运算过程。

(1) 阶码相加：$x+y$。

(2) 尾数相乘：$1.f_x\times 1.f_y$。

(3) 结果规格化并判断溢出：$X\times Y=1.f_x\times 1.f_y\times 2^{x+y}$。结果计算完毕后，计算结果可能不符合规格化浮点数的要求，或计算结果可能直接溢出。

(4) 舍入：如果尾数比规定位数长，则舍入。

(5) 再次规格化：与浮点数加法相似，舍入后可能会导致数据不符合规格化浮点数的标准，需要再次规格化。

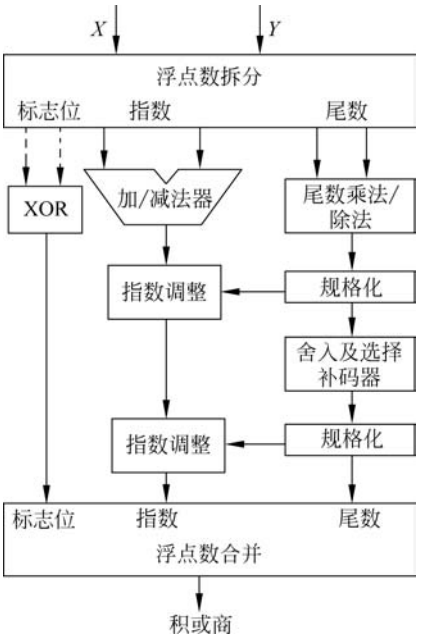

图5-7 浮点数乘法器的结构框图

浮点数乘法单元的运算过程也与浮点数乘法过程类似。如图5-7所示，浮点数操作数首先根据IEEE 754标准进行拆解和特殊数值判断。拆解得到的浮点数符号位会进行异或操作，得出结果的符号位。指数偏移位会通过定点加法器进行有符号加法操作。

两个尾数会进入一个无符号乘法器中进行运算。由于每个有效数都带有一个隐藏的1和小数，尾数乘法单元将是一个无符号的$(l+1)\times(l+1)$的乘法器，其中l是尾数的长度，通常产生一个$2l+2$位的结果。由于这个结果要在输出处舍入到$l+1$位，因此在乘法器设计中，可以忽略这个范围之外的计算。

与加法器类似，尾数乘法器输出的结果也可能过大或者过小，所以需要先进行规格化，再进行舍入，舍入后还需进行规格化。两次规格化可能会导致指数偏移量的变化，这些变化需要增加到指数偏

移量加法器中得到最后的指数偏移量结果。

最后将结果组合，进行溢出判断并输出最终结果。

3. 浮点数融合乘加单元

浮点数的融合乘加运算(Fused Multiply-Add，FMA)要求完成形如 $c=ax+b$ 的操作，一般会记为两个操作。一种简单的方法是按照分离的乘法和加法分别计算。例如，先完成 $a\times x$ 的运算，将其结果数值限定到 N 个比特，然后与 b 的数值相加，再把结果限定到 N 个比特。融合乘加的计算方法则是先完成 $a\times x$ 的运算，在不修剪中间结果的基础上再进行加法运算，最终得到完整的结果后再限定到 N 个比特。由于减少了数值的修剪次数，融合乘加操作可以提高运算结果的精度。

因此，一种浮点融合乘加单元的简单设计就是一个浮点乘法器后级联一个浮点加法器。乘法器需要保留中间结果的所有位数，之后需要一个 2 倍位数的浮点数加法器进行累加。但这种方法的延时较长，约为乘法和加法延迟的和。

另一种性能更优的浮点数融合乘加单元则从多个角度进行了设计优化。如图 5-8 所示，输入 a、x 和 b 的尾数表示为 f_a、f_x 和 f_b，相比于基本的融合乘加单元，具有以下优点。

(1) 使用了预先移位器。根据 a、x 和 b 的指数偏移量，融合乘加单元会将 f_b 左移或右移。移位后的 f_b 将是一个 3 倍位宽的数，以保留任何方向移出的位。

(2) 在 f_a 和 f_x 乘法的累加树结构中，将 f_b 作为一个部分积直接与 f_a 和 f_x 的部分积相累加，节省了单独加法器的步骤，也节省了单独加法的延时。

(3) 使用连续 0/1 预测器，预测在规格化中需要进行的位移。

通过这些技术的运用和设计，图 5-8 的融合乘加单元的延时与乘法器相当。

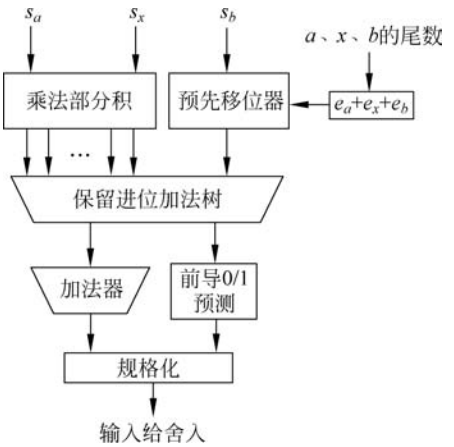

图 5-8　浮点数融合乘加运算器的结构框图

4. 双精度浮点单元

在面向科学计算的 GPGPU 中，还会为双精度浮点配备相应的双精度运算单元，以支持符合 IEEE 754 标准中 64 位双精度浮点的操作。与单精度浮点单元类似，双精度浮点单元

支持包括以加法、乘法、融合乘加和格式转化为主的操作。双精度浮点的融合乘加指令可以在软件中用来实现更高精度的除法和平方根等运算。

双精度浮点单元在设计原理上与单精度浮点单元类似，只不过支持的数据格式发生了改变。考虑到双精度更长的尾数和指数位，电路上的延时会更大，因此设计和实现高性能的双精度单元难度会更高，同时面积也会相应增加。因此，GPGPU 中双精度浮点单元的数量相比于单精度浮点单元会更少。

5.2.3 特殊功能单元

为了提高 GPGPU 在科学计算和神经网络计算中的性能表现，GPGPU 还配备了特殊运算单元来提供对一些超越函数的加速操作。在科学计算中，许多数学运算涉及超越函数。在神经网络运算中，许多激活函数，如 sigmoid、tanh 在引入非线性的同时，也要求 GPGPU 可以快速处理这些激活函数以避免性能瓶颈。

1. 特殊功能函数及计算方式

特殊功能函数多种多样。在 NVIDIA 的 GPGPU 设计中，特殊功能函数主要包括在数值计算中常用的超越函数，如正弦(sine，$\sin(x)$)、余弦(cosine，$\cos(x)$)、除法(division，x/y)、指数(exponential，e^x)、幂乘(power，x^y)、对数(logarithm，$\log(x)$)、倒数(reciprocal，$1/x$)、平方根(square-root，$\sqrt{x}$)和平方根倒数(reciprocal square-root，$1/\sqrt{x}$)函数。

在传统的 CPU 中，直接计算这些超越函数非常困难，一般会通过调用专门的数学库函数，借助数学变换和数值方法对它们进行高精度的求解。这种方式计算的结果可以保持很高的精度，但求解速度慢。GPGPU 则提供了两种计算方式，一是通过类似调用数学库函数的方式，利用通用运算单元(如 NVIDIA 的 CUDA 核心)来完成高精度的计算，二是利用 GPGPU 提供的特殊功能单元专用硬件，完成快速的近似计算。

针对上述 9 种超越函数，CUDA 数学库提供了精确计算的函数和快速近似计算的函数，如表 5-3 所示。从中可以看到，CUDA 提供了所有 9 个超越函数的精确计算版本供编程

表 5-3 在 CUDA 核心和特殊功能单元上计算超越函数对比

函数	CUDA 代码		PTX 指令	
	CUDA 核心执行版本	特殊功能单元执行版本	CUDA 核心执行版本	特殊功能单元执行版本
x/y	x/y	__fdividef(x,y) &-ftz=true	div.rn.f32 %f3,%f1,%f2	div.approx.ftz.f32 %f3,%f1,%f2
$1/x$	1/x	__frcp_[rn,rz,ru,rd](x) &-ftz=true	rcp.rn.f32 %f2,%f1	rcp.approx.ftz.f32 %f2,%f1
$\sqrt{x}$	sqrtf(x)	__fsqrt_[rn,rz,ru,rd](x) &-ftz=true	sqrt.rn.f32 %f2,%f1	sqrt.approx.ftz.f32 %f2,%f1

续表

函数	CUDA 代码		PTX 指令	
	CUDA 核心执行版本	特殊功能单元执行版本	CUDA 核心执行版本	特殊功能单元执行版本
$1/\sqrt{x}$	1.0/sqrtf(x)	rsqrtf(x) && -ftz=true	sqrt.rn.f32 %f2,%f1 rcp.rn.f32 %f2,%f1	rsqrt.approx.ftz.f32 %f2,%f1
x^y	powf(x)	__powf(x,y) && -ftz=true	非常复杂	lg2.approx.ftz.f32 %f3,%f1 mul.ftz.f32 %f4,%f3,%f2 ex2.approx.ftz.f32 %f5,%f4
e^x	expf(x)	__expf(x) && -ftz=true	非常复杂	mul.ftz.f32 %f2,%f1,0f3FB8AA3B ex2.approx.ftz.f32 %f3,%f2
$\log(x)$	logf(x)	__logf(x) && -ftz=true	非常复杂	lg2.approx.ftz.f32 %f2,%f1 mul.ftz.f32 %f3,%f2,0f3F317218
$\sin(x)$	sinf(x)	__sinf(x) && -ftz=true	非常复杂	sin.approx.ftz.f32 %f2,%f1
$\cos(x)$	cosf(x)	__cosf(x) && -ftz=true	非常复杂	cos.approx.ftz.f32 %f2,%f1

人员调用，只需要声明 math_functions.h 头文件。CUDA 还对应提供了快速近似计算的版本，只需要声明 device_functions.h 头文件。另外，快速近似版本的函数调用形式基本上是在精确版本的函数前面加上“__”作为前缀。例如，如果编程人员需要利用特殊功能单元完成余弦函数的计算，可以调用__cos(x)函数并指明“-ftz=true”，它使得所有的非规格化浮点数均为 0，或通过指明“-use_fast_math”这一编译选项，让 nvcc 编译器强制调用快速近似版本，利用特殊功能单元实现硬件加速。CUDA 提供的丰富函数类型和调用方法使得超越函数的运算在 CUDA 级别上基本就可以完成，同时也为编程人员提供了运算精度和速度之间的选择权。

表 5-3 对应给出了精确计算的 PTX 指令和快速近似计算的 PTX 指令形式。例如，对于 x^y、e^x、$\log(x)$、$\sin(x)$和 $\cos(x)$函数，精确计算会将它们转换为一连串复杂的 PTX 代码。如果选择了带有“__”前缀的快速近似函数或使用了“-use_fast_math”的编译选项，那么往往只需要少数几条 PTX 指令就可以完成所有超越函数的计算。近似计算的 PTX 函数

往往具有如下形式

```
function.approx.ftz.f32 %f3, %f1, %f2;
```

其中，“approx”表示近似计算，“ftz”表示对于非规格化数采用近似到0的策略，“f32”表示单精度浮点类型。

上述的超越函数计算主要是针对单精度浮点数据进行的。如果是双精度浮点的超越函数计算，一般只能采用精确的CUDA库函数进行。只有少数的超越函数，如rcp和rsqrt，特殊功能单元提供了近似计算的版本。

另外，特殊功能函数还支持属性插值及纹理映射和过滤操作。

2. 特殊功能单元的结构

GPGPU配备了专门的特殊功能单元来对超越函数的快速近似计算提供硬件支持。使用硬件计算超越函数有多种方法。已有研究表明，基于增强的最小逼近的二次插值算法是硬件实现数值逼近的一种有效的方法，它可以实现快速且近似的超越函数计算。这个算法主要包括三个主要步骤。

(1) 判断输入函数和输入是否存在特殊值情况。

(2) 根据输入的高位组成增强的最小逼近的二次插值算法的参数。

(3) 根据输入的低位计算最终的近似结果。

基于这个算法，图5-9给出了一种特殊功能单元的设计方法。具体来说，它的输入是 n 位的 X 及对应需要求解的函数 f，输出是该函数的近似解 $f(X)$。

根据步骤(1)，针对输入的参数 X 和函数 f，通过专门的检查逻辑判断 X 是否为特定的数值，以确定 X 是否要继续后续的计算。

根据步骤(2)，为了计算近似解 $f(X)$，X 会被分为两部分，即 m 位高位组成的 X_u 及 $n-m$ 位低位组成的 X_l。由于算法后期会利用 X_l 计算出 $f(X)\approx C_0+C_1X_0+C_2X_l^2$ 来给出近似的结果，因此为了得到 C_0、C_1 和 C_2 的值，特殊功能单元的设计还包括使用 X_u 作为地址来获得 C_0、C_1 和 C_2 的查表结构。

根据步骤(3)，为了计算 $f(X)\approx C_0+C_1X_l+C_2X_l^2$，在查表获取 C_0、C_1 和 C_2 三个系数的同时，X_l 会进入Booth编码器和专门设计的平方器进行编码和计算，通过 C_1 和 C_2 硬连线得到 $C_2X_l^2$ 和 C_1X_l 的结果。平方器是经过特殊优化的，相比于传统的乘法器，平方器会更快地处理两个相同数的乘积。为了优化截断误差，平方器添加了一个与输入函数相关的偏差值。在后续求和过程中，为了利用 $C_2X_l^2$ 优化后续计算，特殊功能单元还可以将结果进行编码。

如果特殊功能单元支持多个函数，不同函数的 C_0、C_1 和 C_2 系数并不同。这可以通过查找表中系数的适当排列或根据乘积结果显式移位器来适应。特殊功能单元还会在求和树中兼顾特定函数的偏差。该偏差是基于对每个函数的大量数据模拟而预先确定的，其目的是使误差分布居中，减少总体误差的最大值。

将求和的结果规格化，进行合并和选择之后即可输出一个近似的 $f(X)$ 结果。具体算法和数学变换可参见文献[8-10]。

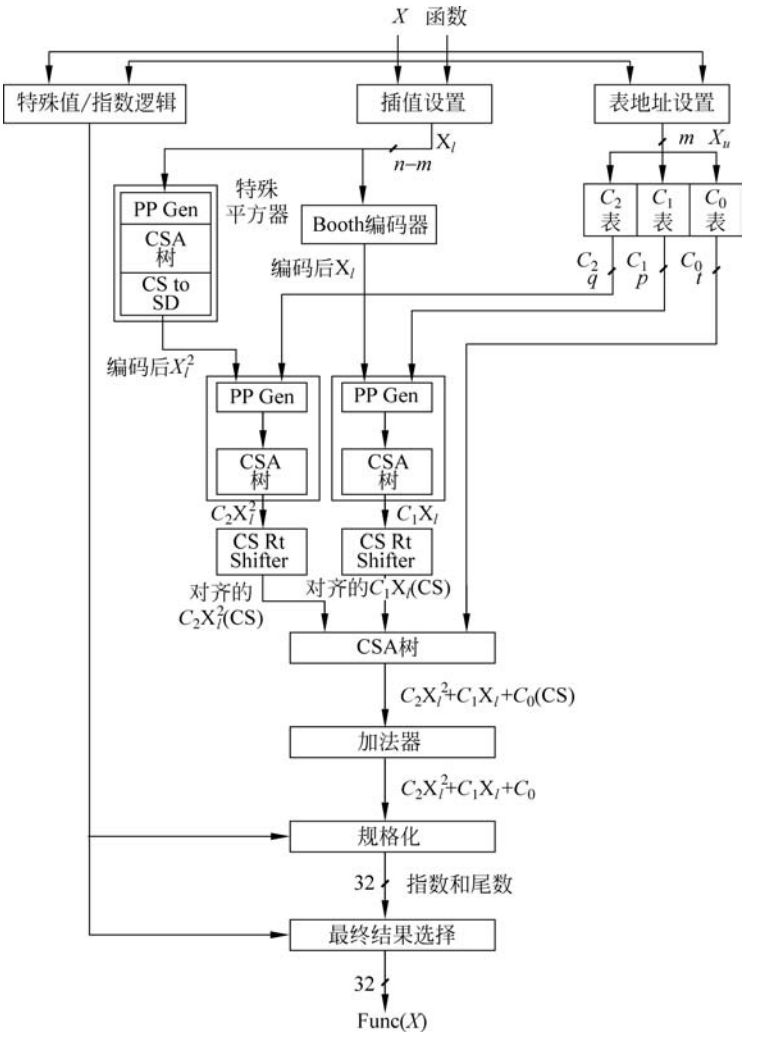

图 5-9　特殊功能单元的硬件结构框图

3. 精确与近似计算的对比

相比于通过 CUDA 核心执行数学库函数来准确计算超越函数的过程，特殊功能单元的计算更为快速，但是精度较低。然而，对于许多 GPGPU 的通用计算场景来说，恪守精度并不是必需的要求，与最末位损失的精度相比，更高的计算吞吐率则是更为重要的。

表 5-4 显示了两者的计算精度对比，其误差的默认基本单位为 ulp（unit in the last place）。对于给定的浮点格式，ulp 是与此浮点数值左右最近的两个浮点数的距离。例如，对于实数 0.1，如果用 FP32 表示，距离 0.1 最近的两个数为 $3DCCCCCC_{16}$ 及 $3DCCCCCD_{16}$，其对应的十进制数为 $0.099\ 999\ 994\ 039\ 536_{10}$ 及 $0.100\ 000\ 001\ 490\ 12_{10}$。两者之间的距离为 $0.000\ 000\ 007\ 450\ 76_{10}$。因此可以说，0.1 在 FP32 表示下的 1ulp 等于 0.000 000 007 450 76。

表 5-4 基于 CUDA 核心的数学库与特殊功能单元计算超越函数的精度对比

数学库函数	CUDA 核心误差	特殊功能单元函数	特殊功能单元误差
x/y	0	__fdividef(x,y)	当 y 属于$[2^{-126},2^{126}]$,最大误差为 2
1/sqrt(x)	0(编译选项增加－prec－sqrt＝true)	rsqrtf(x)	2
expf(x)	2	__expf(x)	最大误差 2＋floor(abs(1.16 * x))
exp10f(x)	2	__exp10f(x)	最大误差 2＋floor(abs(2.95 * x))
logf(x)	1	__logf(x)	当 x 属于[0.5,2],最大绝对误差为 $2^{-21.41}$,否则 ulp 误差大于 3
log2f(x)	1	__log2f(x)	当 x 属于[0.5,2],最大绝对误差为 2^{-22},否则 ulp 误差大于 2
log10f(x)	2	__log10f(x)	当 x 属于[0.5,2],最大绝对误差为 2^{-24},否则 ulp 误差大于 3
sinf(x)	2	__sinf(x)	当 x 属于$[-\pi,\pi]$,最大绝对误差为 $2^{-21.41}$,否则更大
cosf(x)	2	__cosf(x)	当 x 属于$[-\pi,\pi]$,最大绝对误差为 $2^{-21.19}$,否则更大

在计算速度方面,精确计算的超越函数会占用更多的执行时间。例如,sinf()和 cosf()函数可能会达到上百个周期以上。基于特殊功能单元的近似计算,则主要依赖特殊功能单元的电路设计。例如,在 NVIDIA Fermi GT200 GPGPU 中,__sinf()和__cosf()需要 16 个周期,平方根倒数或对数操作需要 32 个或更长。

5.2.4 张量核心单元

NVIDIA 公司在最近几代 GPGPU(如 Volta、Turing 和 Ampere)中添加了专门为深度神经网络设计的张量核心(tensor core),大幅提升 GPGPU 在低位宽数据下的矩阵运算算力,并专门设计了支持张量核心单元的编程模型和矩阵计算方式。本书第 6 章将专门针对张量核心单元进行详细的介绍,介绍张量核心的架构设计特点,进而理解现代 GPGPU 对深度神经网络运算的加速方式。

5.3 GPGPU 的运算单元架构

GPGPU 的运算核心包含了不同类型的运算单元,用以实现不同的运算指令,支持不同场景下多种多样的数据计算。例如,整型运算指令在整型运算单元上执行,单精度浮点运算指令在单精度浮点运算单元上执行,超越函数运算指令在特殊功能单元上执行。

每种类型的运算单元又包含了大量相同的硬件,遵照 SIMT 计算模型的方式组织并运行,因此用较少的控制代价提供了远高于 CPU 的算力。硬件上大量的运算单元与可编程多处理器中的线程束调度器和指令发射单元相配合,实现了指令流水线中段和后段的连接,

构建了高效的运算单元架构，为大规模的数据处理提供了底层的硬件支持。

5.3.1　运算单元的组织和峰值算力

算力指计算能力。一般来讲，不同的计算平台对算力有不同的衡量方法。GPGPU 注重通用计算场景中单精度浮点运算的能力，所以会采用每秒完成的单精度浮点操作个数，即 FLOPS 来衡量。峰值算力指理想情况下所有运算单元都工作时所能够提供的计算能力，它往往是硬件能力的一种衡量，并不能代表软件运行时的实际性能。对于单精度浮点操作而言，每个 CUDA 核心每周期可以支持一个单独的加法或乘法操作，还支持浮点融合乘加指令(FFMA)同时完成乘加两个操作，所以一般 GPGPU 的峰值算力是按照 CUDA 核心满载执行 FFMA 指令的 2 个操作来计算的。其他类型数据的算力也可以类似计算。

GPGPU 以可编程多处理器为划分粒度，将不同类型的大量运算单元分别组织在一起，形成多条不同的运算通路，实现了以运算为主的运算单元架构。图 5-10～图 5-12 分别给出了 NVIDIA 公司历代主流 GPGPU 架构中可编程多处理器的几种典型架构，重点展示了其运算单元架构及它们与其他主要模块的关系。

图 5-10 展示了早期 Fermi 架构的可编程多处理器结构框图，主要包括 2 个线程束调度器、2 个指令发射单元、128KB 寄存器文件、32 个 CUDA 核心、16 个 LD/ST 单元、4 个 SFU 和 64KB 的 L1 数据缓存/共享存储器等。

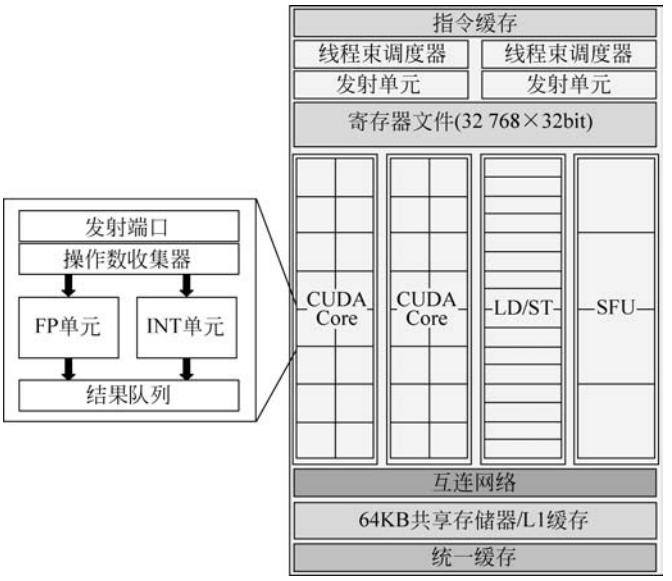

图 5-10　Fermi 架构可编程多处理器结构框图

图 5-11 展示了 Kepler 架构的可编程多处理器结构框图，主要包括 4 个线程束调度器、8 个指令发射单元、256KB 寄存器文件、192 个 CUDA 核心、32 个双精度浮点单元、32 个 LD/ST 单元、32 个 SFU 和 64KB 的 L1 数据缓存/共享存储器。

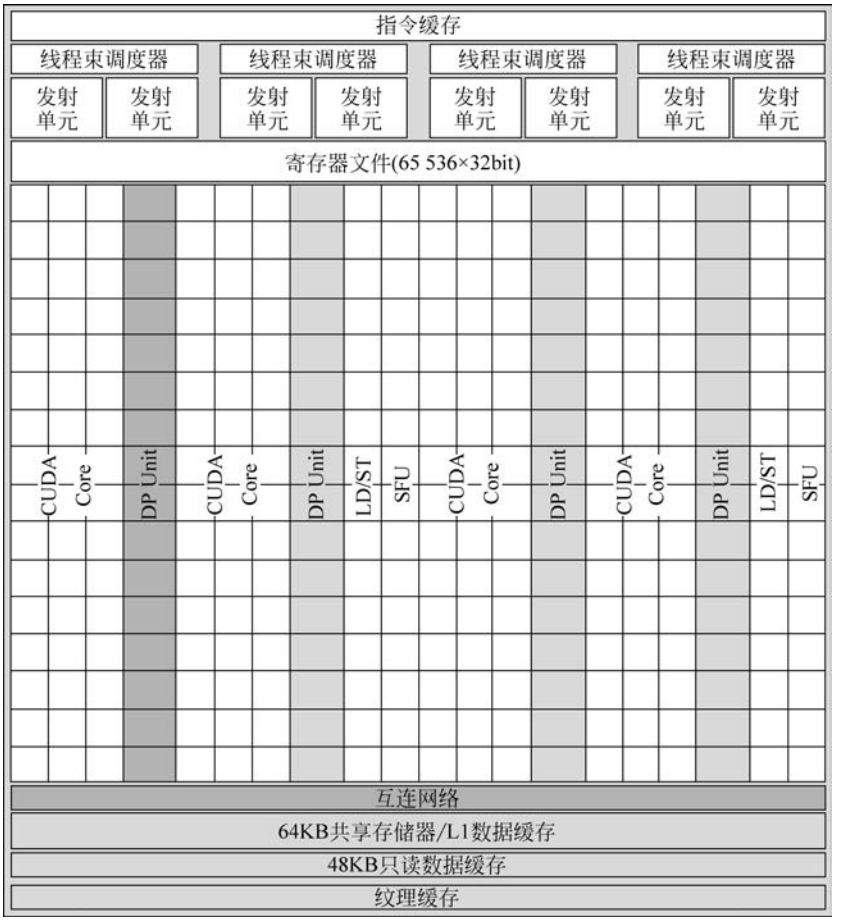

图 5-11 Kepler 架构可编程多处理器结构框图

图 5-12 展示了 Volta 架构的可编程多处理器结构框图，主要包括 4 个处理子块(processing block)及 128KB 的 L1 数据缓存/共享存储器。其中，每个处理子块又包括 1 个线程束调度器、1 个指令发射单元、64KB 寄存器文件、8 个双精度浮点单元、16 个整型单元、16 个浮点单元、2 个张量核心、8 个 LD/ST 单元和 4 个 SFU。

仅从 GPGPU 运算单元架构中独立的数据通路来看，各种类型的峰值算力可以根据运算单元的个数独立计算。表 5-5 和表 5-6 分别总结了历代主流 GPGPU 的核心工作频率、可编程多处理器和处理子块的个数、CUDA 核心单元、双精度浮点单元、特殊功能单元的个数和处理能力。从这些数据可以得到不同架构的峰值算力。对于一种数据类型及其精度而言，GPGPU 的峰值计算能力可以使用单个周期执行的操作数量、参与运算的单元数及单元的工作频率相乘计算得出。

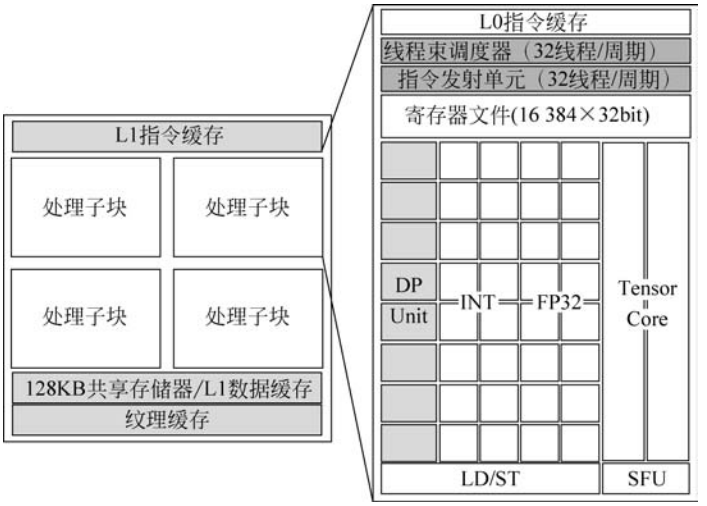

图 5-12　Volta 架构可编程多处理器结构框图

表 5-5　NVIDIA 主流架构 GPGPU 的关键硬件参数

架　　构	核心频率/MHz	SM 数量	处理子块数量/SM	CUDA 核心单元	双精度单元	特殊功能单元
Fermi(GF100)	701	16	N/A	512	N/A	64
Kepler(GK110)	875	15	N/A	2880	960	240
Maxwell(GM200)	1114	24	4	3072	96	768
Pascal(GP100)	1480	56	2	3584	1792	896
Volta(GV100)	1530	80	4	5376	2560	1280
Turing(TU102)	1770	72	4	4608	N/A	1152
Ampere(GA100)	1410	128	4	6912	4096	2048

表 5-6　NVIDIA 主流架构 GPGPU 的峰值算力　　单位：TFLOPS

架　　构	整　　型	FP32	FP64	FP16
Fermi(GF100)	N/A	1.5	0.768	N/A
Kepler(GK110)	N/A	5.04	1.68	N/A
Maxwell(GM200)	N/A	6.84	0.21	N/A
Pascal(GP100)	N/A	10.6	5.3	21.2
Volta(GV100)	15.7	15.7	7.8	31.4
Turing(TU102)	16.3	16.3	N/A	32.6
Ampere(GA100)	19.5	19.5	9.7	78

以 Ampere 架构为例，单精度浮点 FP32 单元可以在一个周期完成一次全通量的 FFMA 运算，相当于乘法和加法两个操作。由表 5-5 可知，A100 配备有 6912 个 FP32 单

元，每个单元的工作频率[①]为 1410MHz。所以，A100 的 FP32 峰值算力为 2 OPs×6912×1410MHz=19.5TFLOPS。

对于双精度浮点 FP64 单元的峰值性能，同样可以使用上述计算公式，A100 包含 3456 个 FP64 单元，故 FP64 的峰值算力为 2OPs×3456×1410MHz=9.7TFLOPS。

32 位整型运算单元与 FP32 单元数量相等，核心工作频率相等，可以用整型 FMA 指令来评估 A100 的 32 位整型峰值算力为 2OPs×6912×1410MHz=19.5TFLOPS。

值得注意的是，还可以用 FP32 和 FP64 单元支持 FP16 数据运算。一般认为一个 FP32 核心可以支持 2 个 FP16 数据的运算，因此 Pascal、Volta 和 Turing 架构中非张量核心实现的 FP16 峰值算力为 FP32 峰值算力的 2 倍。在 A100 架构中，除 FP32 单元外，FP64 单元也可能被设计成支持 4 个 FP16 数据的运算，因此 A100 中非张量核心实现的 FP16 峰值算力表示为两部分峰值算力之和，为 78TFLOPS。

特殊功能单元的计算吞吐率取决于函数的类型，所以很难用峰值算力来衡量。表 5-5 中仅给出了其在可编程多处理器中的数量。

另外，NVIDIA GPGPU 的架构耦合了通用计算和图形处理的功能，但不同的产品有所侧重。自 Fermi 架构以来，每一代计算卡和图形卡都包含了 FP32 单元，并且将 FP32 单元和 INT32 的 ALU 整合在一个 CUDA 核心中。从 Volta 架构起，计算卡的 FP32 和 INT32 单元分离，而图形卡依然绑定在一起。但图形卡中的 FP64 单元往往都会大幅缩水，如表 5-7 所示。这也是两者面向不同的领域而导致的不同设计。

表 5-7 NVIDIA 主流架构 FP32 和 FP64 的峰值算力

架　构	计　算　卡	图　形　卡
Fermi	同图形卡	GF100,32:0
Kepler	GK110,192:64	GK104,192:0
Maxwell	GM200,128:4	GM204,128:0
Pascal	GP100,64:32	GP104,128:0
Volta	GV100,256:128	只有计算卡
Turing	只有图形卡	TU102,FP32:FP64 256:0
Ampere	GA100,256:128	GA102,256:0(只计算单独的 FP32)

5.3.2 实际的指令吞吐率

峰值算力体现的是 GPGPU 的硬件能力，需要每个周期都有指令在执行，这在实际内核函数的执行过程中很难达到。软件代码的结构(如条件分支、跳转等)及数据的相关性(如 RAW 相关)都会影响指令的发射。硬件结构(如线程束调度器和发射单元)及片上存储的访问带宽(如寄存器文件和共享存储器)也会影响指令的发射，使得实际的指令吞吐率很难达到峰值算力。指令吞吐率是指每个时钟周期发射或完成的平均指令数量，代表了实际应

① GPGPU 的核心工作频率分为基频和睿频，在计算峰值性能时采用睿频得到性能的上界。

用执行时的效率，是刻画 GPGPU 性能的重要指标。本节将重点关注线程束调度器、发射单元等硬件结构对指令发射吞吐率带来的影响。

1. 调度器和发射单元的影响

不同的 GPGPU 架构会采用不同的运算单元架构。不仅运算单元的组织方式有所不同（如 5.3.1 节的介绍），而且不同架构的指令调度和发射方式也不同，并且与运算单元的数量和组织方式有着密切的关系。

1）Fermi 运算单元架构

图 5-10 展示了 NVIDIA 公司的 Fermi 架构的运算核心，可编程多处理器框图主要包括 CUDA 核心、特殊功能单元和 LD/ST 单元。CUDA 核心内主要包括整型运算单元（INT Unit）和浮点运算单元（FP Unit）。整型运算单元采用完全流水化的方式支持 32 位全字长精度的整型运算指令，也支持 64 位整型的运算。浮点运算单元主要支持单精度浮点指令，有时也对其他精度的指令提供支持。

图 5-13 给出了 Fermi 架构下不同类型运算单元与线程束调度器、发射单元的关系。线程束调度器负责从活跃线程中选择合适的线程束，经过指令发射单元发射到功能单元上。Fermi 架构每个可编程多处理器包含 2 个线程束调度器和对应的发射单元。可以看到，32 个 CUDA 核心被分成了 2 组，与特殊功能单元、LD/ST 单元共享 2 个线程束调度器和发射单元。在 Fermi 架构 GPGPU 中，CUDA 核心的频率是调度单元的两倍，因此 16 个 CUDA 核心就可以满足一个线程束 32 个线程的执行需求。为了让 32 个 CUDA 核心满载，Fermi 架构配备了 2 个线程束调度器，这样就可以让包括 CUDA 核心在内不同类型的运算单元能够尽可能保持忙碌。但倍频的设计造成 Fermi 架构功耗偏高，后续设计取消了这种方式，转而采用其他技术来平衡调度器、发射单元和功能单元的关系。

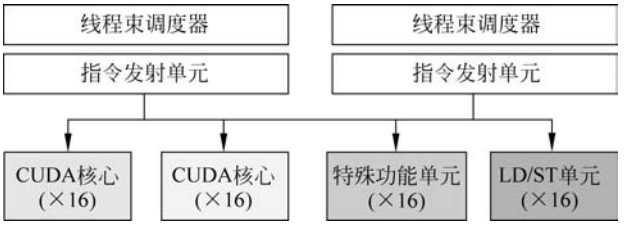

图 5-13　Fermi 架构可编程多处理器中功能单元和线程束调度器发射单元的关系

指令发射单元与功能单元也并非一一对应的关系。不同功能的指令可能会共用发射单元，通过竞争来获得发射机会。Fermi 架构允许两条整型指令、两条单精度浮点指令、整型指令/单精度浮点指令/LDST 指令/SFU 指令的混合同时发射，但双精度指令只能单独发射，不能与任何其他指令配对。一方面可能是由于寄存器文件的带宽限制，另一方面可能是双精度浮点单元要复用两组 FP32 的数据线才能满足数据的带宽需求。

有些类型的功能单元数量较少，这使得对应类型的指令可能需要多个周期才能完成一个线程束 32 个线程的执行。例如，在 Fermi 架构中，每个可编程多处理器只有 4 个特殊功

能单元，这意味着线程束的特殊函数计算指令需要占用特殊功能单元 8 个周期才能完成执行。同样的情况也会出现在 LD/ST 单元上。不过，在一个功能单元的多周期执行过程中，线程束调度器还可以调度其他类型的指令到空闲的功能单元上，从而让所有的功能单元都能保持忙碌状态。

另外，从图 5-13 中可以看到，CUDA 核心的设计使得整型运算单元和浮点运算单元共享一个指令发射端口，这意味着整型运算和浮点运算需要通过竞争来获得执行机会。一旦其中一类指令独占了发射端口，它会阻塞另一类指令进入 CUDA 核心。因此，发射端口的限制导致整型单元和浮点单元不能同时得到指令，降低了执行效率。但考虑到分离的设计会进一步增加调度器的数量，因此这也是一种折中的方案。

2）Kepler 运算单元架构

对应图 5-11 展示的 Kepler 运算单元架构，图 5-14 给出了一个可编程多处理器中功能单元和线程束调度器、发射单元的关系。

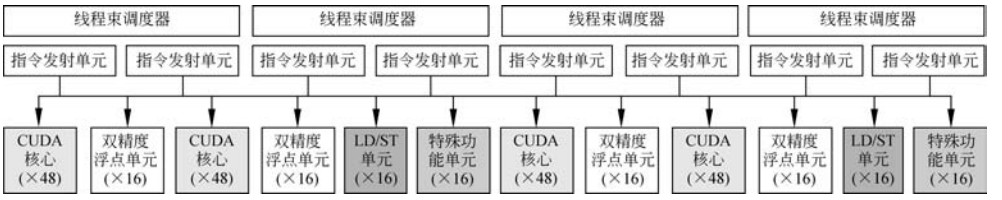

图 5-14　Kepler 架构可编程多处理器中功能单元和线程束调度器、发射单元的关系

芯片集成度的提升使得 GPGPU 中运算单元的数量得以持续增长。Fermi 架构之后，Kepler 架构的 CUDA 核心数量增加到 192 个。原则上，这需要 6 个线程束指令才能满载。考虑到还存在其他功能单元，如双精度单元和 LD/ST 单元，指令数目的需求会更高。然而，由于调度器的硬件复杂性，同时配备这么多调度器在硬件设计上并不明智。因此，Kepler 架构选择了 4 个调度器的设计，并增加双发射单元来弥补调度能力的不足。

双发射技术允许指令发射单元每个周期从调度器选择的线程束中取出两条连续的、不存在数据相关性的指令，同时发射到功能单元上执行。大多数指令都支持双发射，例如两条整型指令、两条单精度浮点指令、整型指令/单精度浮点指令/LDST 指令/SFU 指令的混合，都可以被同时发射。不过，出于硬件复杂度的考虑，存在相关性的指令或第一条指令为分支跳转指令的情况还是会被禁止双发射。实际上，双发射类似于 CPU 中常见的多发射，也是利用指令之间无相关性来提升 ILP 的技术。

在 Kepler 架构中，调度器有 4 个，采用双发射技术理论上可以发射 8 条指令来填满 CUDA 核心和其他功能单元，从而保证峰值算力的需求。由于没有增加调度器的数量，而后的 Maxwell 和 Pascal 架构也都采用了这种技术来应对运算单元数量的增加。由此可见，调度器和发射单元的数量是由功能单元来决定的，以追求最高的指令吞吐率并降低硬件的复杂性为目标。

3）Volta 运算单元架构

对应图 5-12 展示的 Volta 运算单元架构，图 5-15 给出了 Volta 架构一个处理子块中功能单元和线程束调度器、发射单元的关系。

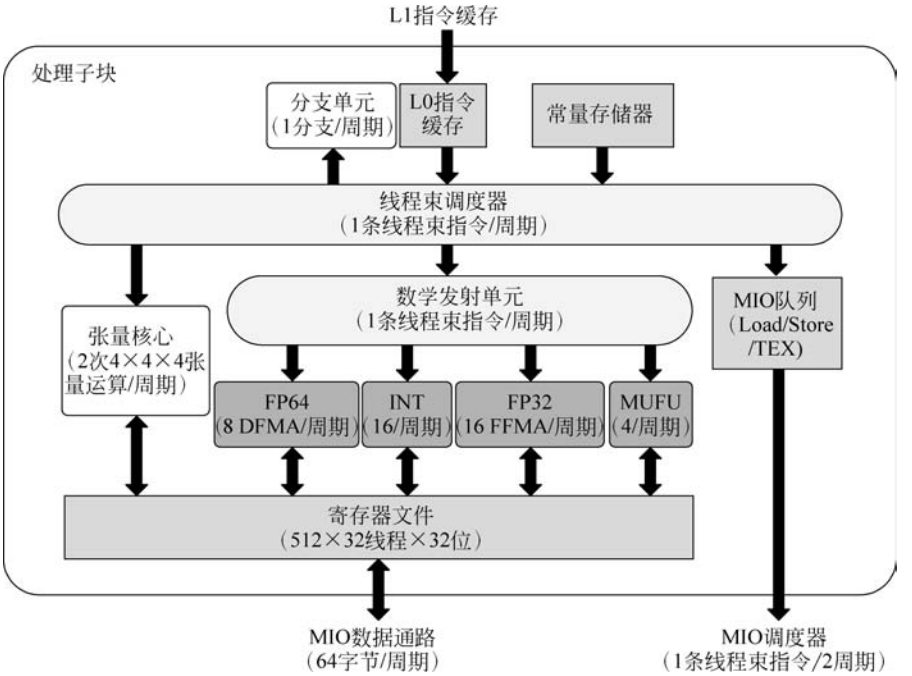

图 5-15　Volta 架构一个处理子块中功能单元和线程束调度器、发射单元的关系

自 Volta 架构起，整型单元和浮点单元从一个 CUDA 核心中分离出来，分属于不同的执行流水线，两种类型指令的发射也不再需要竞争分发端口，因此二者可以同时以全通量的方式执行 FP32 和 INT32 操作，提高了指令发射的吞吐率。由于许多浮点指令包括一个执行指针算术运算的循环，通常地址运算只涉及整型数据，因此采用这种分离的设计将改善这类指令执行的效率。存储器 I/O 流水线与 CUDA 核心流水线相解耦，使得浮点运算、地址运算和数据加载/写回操作的并行执行成为可能。

Volta 架构的指令发射单元也做了相应调整。整型单元、单精度浮点单元、双精度浮点单元和特殊功能单元接收来自数学发射单元(math dispatch unit)的指令，每类功能单元独立具有一个发射端口。Volta 架构新引入的张量核心单元、MIO 队列(包括 LD/ST 单元、纹理单元)和分支单元(BRanch Unit，BRU)独立于数学发射单元，允许在张量核心单元、MIO 队列、分支单元被占用时发射其他算术运算指令。

同时可以看到，Volta 架构中采用了子块结构，将原来 1 个可编程多处理器拆分成 4 个处理子块(processing block)。由于增加了张量核心单元，相应减少了整型、单精度浮点和双精度浮点单元的数量，需要同时调度的线程束指令数量也减少，因此 Volta 每个处理子块中只配备了 1 个线程束调度器，也不需要双发射，简化了指令调度和发射逻辑的设计。此

时，由于每个处理子块只有 16 个整型单元、单精度浮点单元，因此需要分 2 个周期才能执行一个完整的线程束指令。双精度浮点单元只有 8 个，所以需要分 4 个周期才能执行一个双精度浮点指令。在功能单元占用期间，调度器可以选择和发射其他类型的指令。这再次说明了调度器和发射单元的数量是由功能单元来决定的，以追求最高的指令吞吐率并降低硬件的复杂性为目标。

4）小结

不同 GPGPU 架构中运算单元、线程束调度器、指令发射单元的数量及比例对指令发射逻辑和指令吞吐率有着重要影响。表 5-8 统计了 NVIDIA 主流 GPGPU 架构中一个可编程多处理器中与指令发射相关的硬件参数。可以看到，由于 CUDA 核心数量比较多，在 Kepler 到 Pascal 的连续三代架构中都配备了多个调度器并且每个调度器配备 2 个发射单元，动态选择合适的 CUDA 核心执行，并尽可能满足包括 CUDA 核心在内的多种功能单元的执行需要。从 Volta 到 Ampere 架构，可编程多处理器规模并没有进一步扩大，还引入了处理子块的层次，对运算单元进行了明确的分区，使得调度器、发射单元和功能单元的对应关系更为固定。同时，张量单元的引入一定程度上也降低了每个处理子块中 CUDA 核心的数量。例如，在 Volta 架构的处理子块中，CUDA 核心的数量减少为 16 个，一个线程束的执行需要两个周期，因此也就不需要配备更多的调度器和发射单元。单个线程束调度器和发射单元的配置在没有分支跳转和数据相关性及存储带宽充足的情况下就可以保证功能单元的利用率。

表 5-8 NVIDIA 主流 GPGPU 架构中一个流多处理器内与指令发射逻辑相关的参数对比

架构名称	CUDA 核心单元数量	处理子块数量/SM	线程束调度器数量	指令发射单元数量
Fermi(GF100)	32	N/A	2	2
Kepler(GK110)	192	N/A	4	8
Maxwell(GM200)	128	4	4	8
Pascal(GP100)	64	2	2	4
Volta(GV100)	64	4	4	4
Turing(TU102)	64	4	4	4
Ampere(GA100)	64	4	4	4

2. 寄存器文件和共享存储器的影响

寄存器文件的访问带宽是影响指令发射吞吐率的重要因素之一。4.2 节介绍了 GPGPU 的寄存器文件多采用多板块设计。如果板块数目过少，无法满足多个操作数并行访问的需要。当板块数目达到 4 个时，结合操作数收集器的设计，对于 FMA 这样需要读取三个源操作数和写回一个目的操作数的指令，在没有板块冲突的情况下，仍然可以在一个周期内取得所有源操作数，同时还可以完成前序指令的写回。从指令吞吐率的角度看，4 个板块可以实现 FMA 指令寄存器操作数同时访问。但如果有多个调度器连读执行 FMA 指令时，也会由于寄存器带宽的限制而无法满足几条指令多个操作数同时访问的需求。但考虑

到多数指令只包含1个或2个源操作数，在未发生板块冲突的情况下，寄存器文件仍然可以支持2条或更多指令源操作数的并行读取，让功能单元保持忙碌状态。

值得注意的是，寄存器文件的板块数目也并非越多越好，文献[15]通过实验验证了对于可编程多处理器内128KB的寄存器文件，将其板块数量从16增加到32个，指令的执行性能并没有获得显著提升，反而在功耗和芯片面积上造成了很大的开销。另外，多端口的板块设计也会引入很高的开销。因此，平衡寄存器文件的开销和操作数访问的并行度也是影响指令吞吐率的重要问题。

与寄存器文件类似，共享存储器的带宽也会对GPGPU的指令发射吞吐率造成影响。如4.3.2节介绍的共享存储器多数采用32个板块，每个板块的数据位宽为32比特的结构。假设每个线程束指令读取一个共享存储器数据，则一个周期可以支持2个半线程束或1个完整线程束的并行访问。但能否充分利用共享存储器的板块级并行还取决于共享存储器的地址访问方式，不同程度的板块冲突会导致指令发射吞吐率不同程度的降低。因此，在并行程序设计时需要尽可能避免出现上述情形。

另外，寄存器文件和共享存储器的容量也会对指令发射的吞吐量造成间接的影响。在GPGPU可编程多处理器中，线程并行度受到诸多因素的制约。如果单个线程占用的寄存器数量过多，或需要的共享存储器容量过大，线程束并行度会随之受到影响。此时，活跃线程数量会大幅减少，导致线程束调度器无法选择合适的指令进行发射，也会导致指令的吞吐率降低。

5.3.3 扩展讨论：脉动阵列结构

当具备大量运算单元时，硬件架构应该如何组织？GPGPU架构借助SIMT计算模型将它们组织成多个并行通道，使得数据级并行(Data-Level Parallelism，DLP)的计算能够在各个通道上独立完成。当然，这不是唯一的方法。大量的运算单元硬件还可以组织成其他形式。例如，近年来随着神经网络，尤其是卷积神经网络的兴起，还可以采用脉动阵列的组织结构，高效地支持通用矩阵乘法(GEneral Matrix Multiply，GEMM)运算。本节将介绍脉动阵列的结构及它如何高效地支持GEMM运算。

1. 脉动阵列的基本结构

脉动阵列最早是由H. T. Kung在1982年提出的。它利用简单且规则的硬件结构，支持大规模并行、低功耗、高吞吐率的积分、卷积、数据排序、序列分析和矩阵乘法等运算。2016年，Google公司发布的第一代张量处理器(Tensor Processing Unit，TPU)中就基于脉动阵列结构加速卷积计算，使得该架构再次受到人们的广泛关注。

图5-16展示了传统计算模型和脉动阵列计算模型的区别。假设一个处理单元与存储器组成的系统中存储器的读写带宽为10MB/s，每次操作需要读取或写入2字节的数据，那么即便处理单元的运算速度再快，该系统的最大运算吞吐率也仅为5MOPS。如果采用脉动阵列的结构设计，将6个处理单元串联在一起，则在相同的数据读写带宽下，运算吞吐率能提高到30MOPS，因相邻处理单元之间可以直接交换数据。一般情况下，由于数据访存

的时间要高于数据处理的时间，系统的性能往往受限于访存效率。脉动阵列的设计思想就是让中间数据尽可能在处理单元中流动更长的时间，减少对集中式存储器不必要的访问，以降低访存开销带来的影响。例如，第 1 个数据进入第 1 个处理单元，经过运算后被送入下一个处理单元，同时第 2 个数据进入第 1 个处理单元，以此类推，直到第 1 个数据流出最后一个处理单元，该数据无须多次访存却已被处理多次。脉动阵列通过多次复用输入和中间数据，以较小的存储带宽开销获得了更高的运算吞吐率。

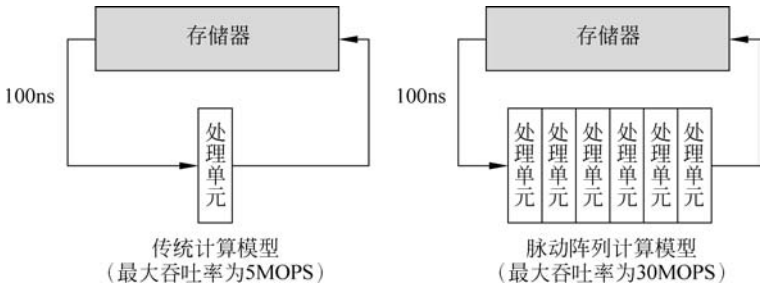

图 5-16　传统计算模型和脉动阵列计算模型对比

在脉动阵列计算模型的基础上，脉动阵列的结构往往被设计成为由若干数据处理单元组成的矩阵形式。如图 5-17 所示，处于相同行和相同列的处理单元之间设置有单向的数据通路，输入数据和大量的中间数据在固定的行、列方向上流过阵列，降低了访存操作成为瓶颈的可能性，从而实现更高的处理效率。以 GEMM 运算为例，脉动阵列接收矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 作为输入数据，存放在阵列左侧和顶部的缓冲区中。根据不同的脉动方式，控制信号会选择一侧或两侧缓冲区中的数据，按照固定的节奏发射到阵列中，触发阵列单元的计算。水平的缓冲区之间也支持相邻或跨越多行的数据互传，为数据的跨行复用提供了一种更加灵活便捷的途径。

当参与 GEMM 计算的矩阵过大或通道过多时，如果把所有通道的数据都拼接在一列上，则很可能造成脉动阵列溢出。由于过大规模的阵列不仅不利于电路实现，还会造成效率和资源利用率的下降，因此可以采用分块计算、末端累加的方式来解决这个问题。把大规模的矩阵分割成几个部分，每个部分都能够适合脉动阵列的大小，然后依次对每个部分的子矩阵进行脉动计算，计算完成的中间结果会临时存放在如图 5-17 所示的底部 SRAM 中。当下一组数据完成计算并将部分和矩阵输出时，从 SRAM 中读取合适位置的部分和结果并累加，实现对两次分块运算结果的整合。直到所有分块都经历了乘加运算，且累加器完成了对所有中间结果的累加后，SRAM 可以输出最终的运算结果。整个数据流的控制由控制单元完成。

在支持 GEMM 的脉动阵列中，每个处理单元专注于执行乘加运算，其结构如图 5-17 右侧所示，其中包括以下内容。

（1）1 个水平向寄存器，用于存储水平输入的元素。该寄存器接收来自左侧相邻处理单元的数据，也可以接收来自水平缓冲区中的数据，实际上取决于当前处理单元在脉动阵列

中的位置。水平向寄存器有两个输出通路，其中一条通路可以将输入元素送入乘法单元进行计算，以生成当前处理单元的计算结果，另一条通路则允许将输入元素直接传递给右侧相邻的处理单元，实现数据在行维度的滑动。

（2）1个竖直向寄存器，用于存储另一个输入矩阵的元素。该寄存器可以接收来自上方相邻处理单元的数据，也可以直接接收来自顶部缓冲区的数据，取决于当前处理单元在脉动阵列中的位置。与水平向寄存器的数据通路类似，竖直向寄存器也存在两条数据通路，将输入数据送入乘法单元或直接传递出去，实现数据在列维度的滑动。

（3）部分和寄存器。根据不同的脉动方式，可能还需要部分和寄存器，用于接收来自上方相邻单元或当前单元产生的部分和，并将该数据送入加法电路中执行累加运算。

（4）乘加电路，对来自水平和竖直方向的输入元素执行乘法运算，然后将结果送入加法电路与来自部分和寄存器的数据进行累加，产生新的部分和。根据脉动的方式，更新部分和寄存器，或将该数据送入下方相邻单元的单元或寄存器内。

（5）控制器，图5-17并未显式画出控制器的位置及布线方式。控制器接收和存储控制信号，以决定启用水平向寄存器和竖直向寄存器的数据通路。此外，控制器也存在一条连接相邻处理单元中控制器的信号通路，以实现控制信号的传递和共享。

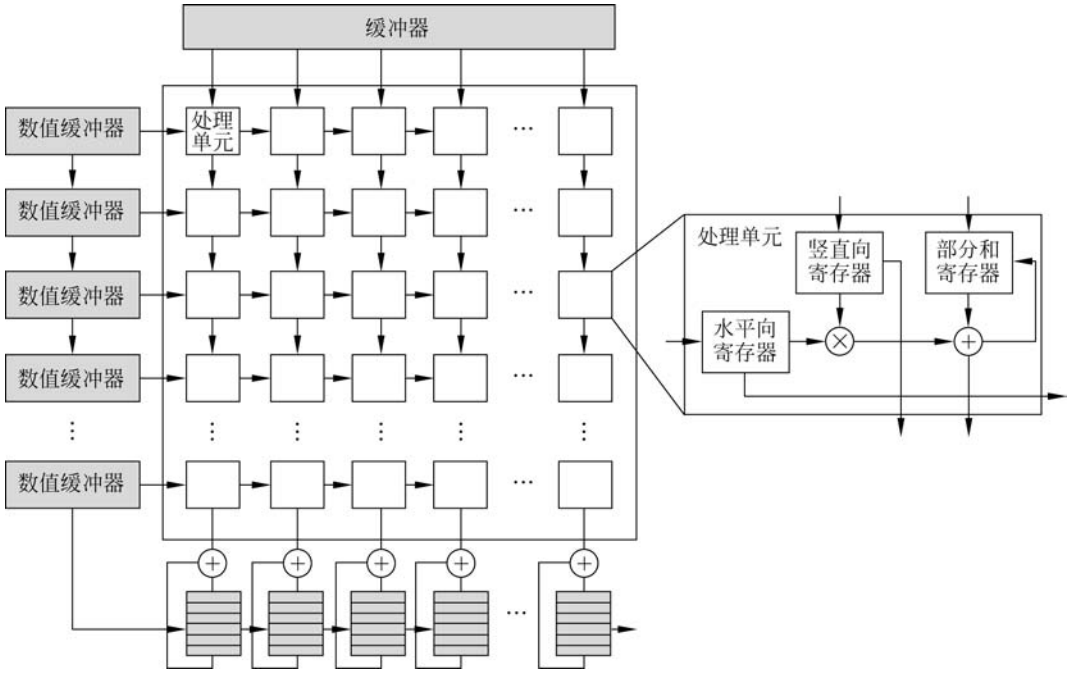

图5-17　脉动阵列的架构设计

脉动阵列的计算模型决定了脉动阵列结构的方式相对固定，因此虽然能够为GEMM、积分、卷积、数据排序、序列分析等运算提供比较高的计算效率，降低对访存和寄存器文件的访问，但能够支持的计算类型也比较单一，结构和控制方式也比较固定，在通用性和可编程

性方面有所不足。

2. GEMM 的脉动计算方式

由于 GEMM 是科学计算、图像和信号处理等应用的核心算子，加速 GEMM 运算将显著改善这些应用的执行效率。自从 H. T. Kung 提出脉动阵列的概念后，许多基于脉动阵列的 GEMM 算法和脉动阵列的改进型结构如“雨后春笋”般涌现出来。

1) 经典脉动阵列结构

在经典脉动阵列架构中，若干数据处理单元被组织成二维矩形结构，相同行列内处理单元之间存在单向数据通路，数据只能在水平或竖直方向内移动。由于每个数据处理单元内设置有寄存器，可以暂存运算所需的数据，因此一些可复用的数据可以被预先加载到脉动阵列并常驻其中，然后其他数据流动起来，便可以形成不同的 GEMM 算法。具体来讲，假设有两个 3×3 的矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 进行如下的 GEMM 运算。根据预加载数据的类型，可以分为 2 种 GEMM 算法的变体：①固定矩阵 $\boldsymbol{C}$，使矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 分别从左侧和顶端流入脉动阵列；②固定矩阵 $\boldsymbol{A}$(或 $\boldsymbol{B}$)，使矩阵 $\boldsymbol{B}$(或 $\boldsymbol{A}$)和 $\boldsymbol{C}$ 分别从左侧和顶端流入脉动阵列。接下来分别以这两种 GEMM 算法的变体为例，分析经典脉动阵列结构执行一个 3×3 矩阵乘法的运算过程。

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

对于第一类情形，固定矩阵 $\boldsymbol{C}$。如图 5-18(a)所示，初始状态时 0 被预加载到脉动阵列中作为 $\boldsymbol{C}$ 的初始值，然后矩阵 $\boldsymbol{A}$ 的元素从左端按照图示方式每周期依次流入不同行，同时矩阵 $\boldsymbol{B}$ 的元素从顶端按照图示方式流入不同列。每个处理单元进行矩阵 $\boldsymbol{A}$ 与 $\boldsymbol{B}$ 的一个元素相乘，然后将乘积与部分和寄存器中的值进行累加。经过图 5-18(b)的第 1 个周期和图 5-18(c)的第二个周期，到达如图 5-18(d)所示的第 3 个周期，矩阵 $\boldsymbol{A}$ 的元素 a_{13} 进入脉动阵列第 1 行的第 1 个处理单元，与顶部流入的数据 b_{31} 进行乘法，然后与部分和寄存器中的值累加，自此 c_{11} 处的处理单元已经完成了第 3 次更新，产生了结果矩阵中的 c_{11}，而这个元素会固定在该处理单元的部分和寄存器中。可以发现，每个部分和经过 3 次更新后即得到结果矩阵中对应位置的元素，这些元素存储在脉动阵列的部分和寄存器中，再将结果输出出来。

对于第二类情形，以固定矩阵 $\boldsymbol{B}$ 为例，它被预加载到脉动阵列中，矩阵 $\boldsymbol{A}$ 从阵列左端流入，矩阵 $\boldsymbol{C}$ 从阵列顶部流入，其初始值都为 0。如图 5-19(a)所示，初始状态时，a_{11} 和 $c_{11}=0$ 同时进入脉动阵列第 1 行的第 1 个处理单元，记为单元(1,1)。此时，a_{11} 与 b_{11} 先执行乘法运算，然后将乘积与 $c_{11}=0$ 累加得到 $a_{11}\times b_{11}$，完成 c_{11} 部分和的第 1 次更新。接下来如图 5-19(b)所示的第 1 个周期，同时发生以下 4 个操作。

(1) 矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{C}$ 分别沿各自的方向移动一个单元，此时 c_{11} 部分和从单元(1,1)流入单元(2,1)。

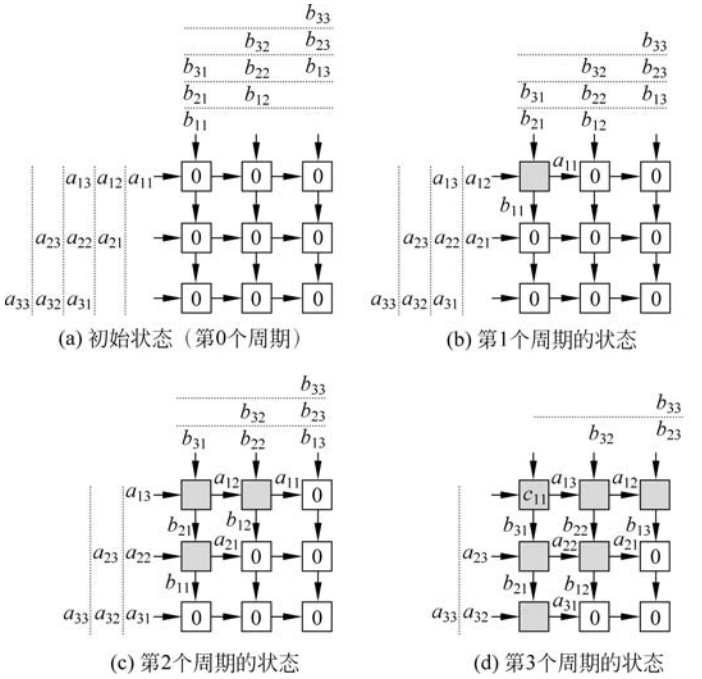

图 5-18　固定矩阵 ***C*** 执行 GEMM 的示例

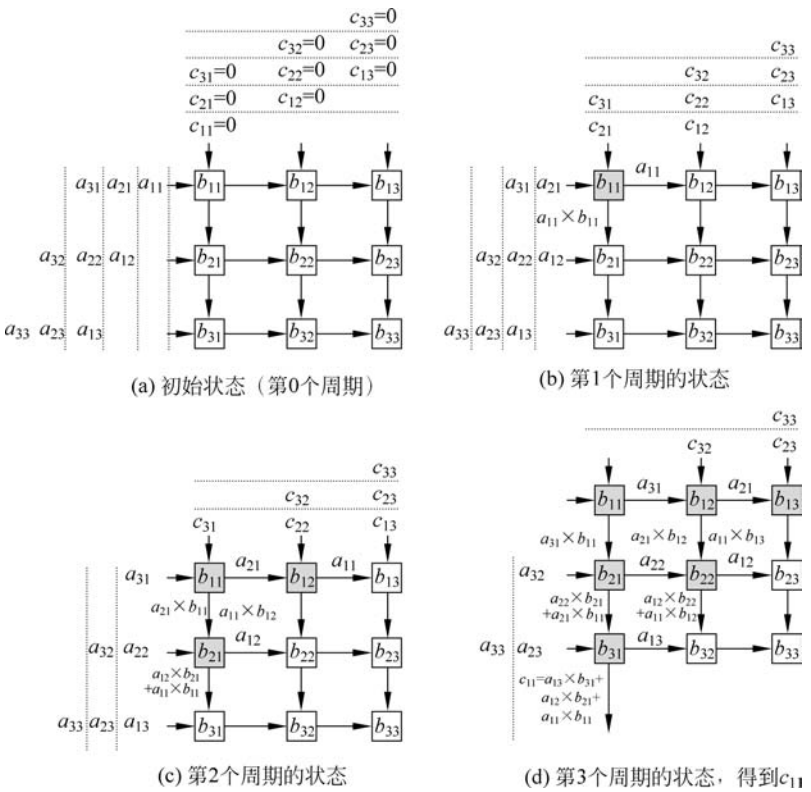

图 5-19　固定矩阵 ***B*** 执行 GEMM 的示例

(2) 第 2 行左侧流入的 a_{12} 和预载入的 b_{21} 相乘,并与流入的 c_{11} 部分和进行累加得到 $a_{11}\times b_{11}+a_{12}\times b_{21}$,完成 c_{11} 部分和的第 2 次更新。

(3) a_{11} 进入单元(1,2)与预载入的 b_{12} 相乘,然后与顶部流入的 $c_{12}=0$ 累加得到 $a_{11}\times b_{12}$,完成 c_{12} 部分和的第 1 次更新。

(4) 矩阵 $\boldsymbol{A}$ 的第 2 个元素 a_{21} 和 $c_{21}=0$ 也进入单元(1,1),执行乘加运算后得到 $a_{21}\times b_{11}$,完成 c_{21} 部分和的第 1 次更新。

第 3 个周期如图 5-19(d)所示。此时 c_{11} 在单元(3,1)中完成其第 3 次更新,产生结果矩阵中的第 1 个元素并将其从阵列底部输出,而其他部分和仍为中间结果,此时被占用的处理单元为灰色三角形区域,展现出数据在脉动阵列中是以倒阶梯状传播的。值得注意的是,这种脉动方式允许结果矩阵在完成运算后自动流出阵列,而不需要像第一种脉动方式那样,添加任何额外的步骤进行结果输出。不过,这种方式要求一个输入矩阵(如矩阵 $\boldsymbol{B}$)是固定的。

2) 双向数据通路结构

上述脉动阵列是一种单向数据通路结构,即处理单元之间的数据流向是单向的,整个阵列只支持数据向右和向下移动,而且需要结果矩阵或输入矩阵两者固定之一,才能完成 GEMM 运算。假设有一种更加灵活的脉动阵列结构,它允许输入矩阵在阵列处理单元之间流动,结果也能够同时流出矩阵,则更符合 GEMM 运算的需求。本节介绍的双向数据通路脉动结构能够达到这个目标。

图 5-20 展示了一种双向数据通路设计的脉动阵列架构及 GEMM 计算过程。阵列中水平方向数据单元之间仍为单向通路,但是在竖直方向增加了一条反向数据通路,使得相邻两个数据单元之间允许数据的双向传递。在执行矩阵乘法运算的过程中,一个矩阵,如矩阵 $\boldsymbol{A}$,沿水平方向从左向右流入脉动阵列,同时另外两个矩阵,如矩阵 $\boldsymbol{B}$ 和 $\boldsymbol{C}$,分别从顶端和底端沿着相反的方向流入。但为了匹配 GEMM 的运算,需要将三个输入的元素间隔起来。为了控制数据进入脉动阵列的间隔,可以在数据的传播路径上添加数量不等的寄存器并进行合理控制。在这种脉动结构和脉动方式下,例如在第 4 个时钟周期,b_{31} 从左侧流入脉动阵列第 1 行第 1 列单元(1,1),与顶部流入的 a_{13} 相乘,然后再将乘积与底部流入的部分和 c_{11} 相加,得到新的部分和。由于 c_{11} 在第 3、4 周期已分别与 $a_{11}\times b_{11}$ 和 $a_{12}\times b_{21}$ 累加,当前为 c_{11} 部分和的最后一次更新,因此下个周期可以在脉动阵列第 1 列顶部得到 GEMM 结果矩阵的第 1 个元素。

在双向数据通路的脉动阵列结构中,所有输入和输出数据都在阵列中流动,允许计算结果自动流出,这是双向结构相较于上述单向结构中 GEMM 算法的一个优势。然而,这种优势是以更大的阵列面积和更长的流水周期为代价的。

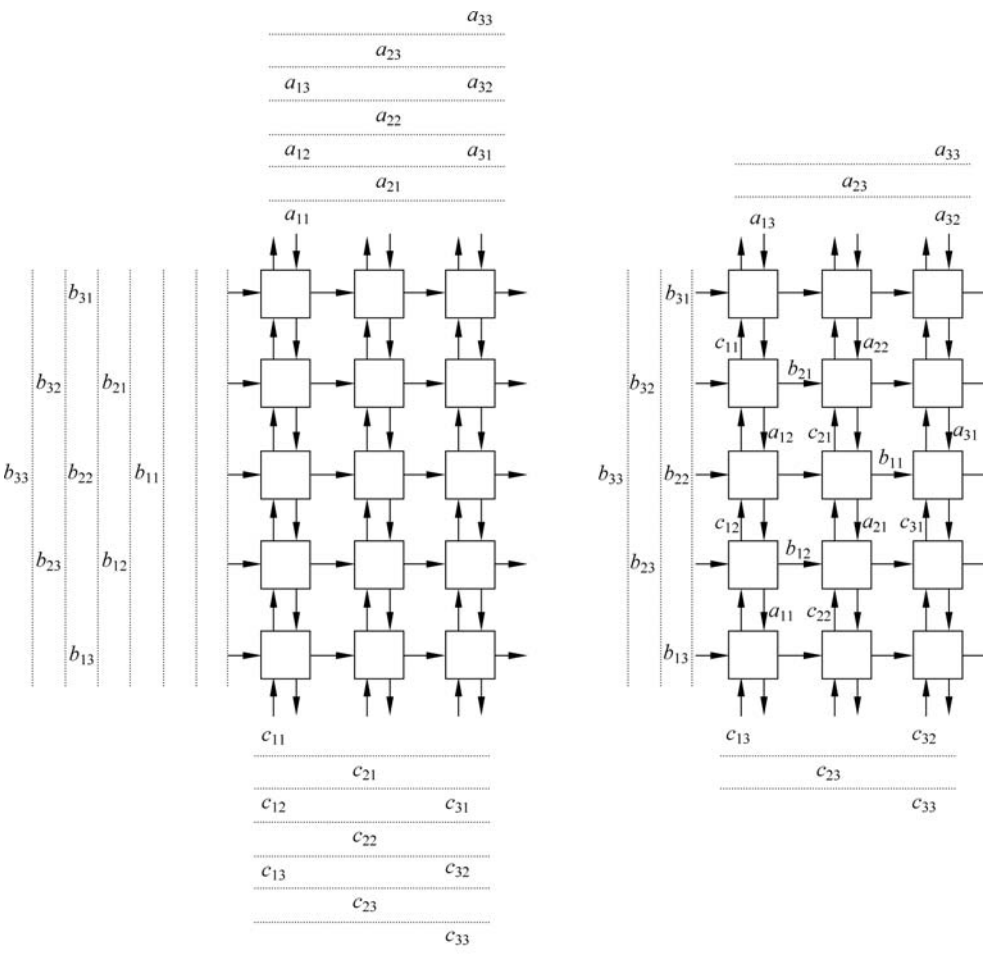

图 5-20　双向数据通路结构下的 GEMM 运算

参 考 文 献

[1] Wikipedia. IEEE 754[Z]. [2021-08-12]. https://zh.wikipedia.org/wiki/IEEE_754.

[2] Paul Teich. TEARING APART GOOGLE'S TPU 3.0 AI COPROCESSOR[Z]. [2021-08-12]. https://www.nextplatform.com/2018/05/10/tearing-apart-googles-tpu-3-0-ai-coprocessor/.

[3] NVIDIA. NVIDIA A100 Tensor Core GPU Architecture[Z]. [2021-08-12]. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[4] Köster, Urs, et al. Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks[C]. 2017 31st Neural Information Processing Systems(NIPS).

[5] Jan M. Rabaey, Anantha Chandrakasan, Borivoje Nikolie. Digital integrated circuits[M]. Englewood

Cliffs: Prentice hall,2002.

[6] Behrooz Parhami. COMPUTER ARITHMETIC Algorithms and Hardware Designs[M]. 2nd ed. NEW YORK: OXFORD UNIVERSITY PRESS, 2010.

[7] Ang Li, Shuaiwen Leon Song, Mark Wijtvliet, et al. SFU-Driven Transparent Approximation Acceleration on GPUs[C]. In Proceedings of the 2016 International Conference on Supercomputing (ICS'16). New York: Association for Computing Machinery, 2016: 1-14.

[8] Pineiro J A, Oberman S F, Muller J M,et al. High-speed function approximation using a minimax quadratic interpolator[J]. IEEE Trans. Computers, 2005. 54(3): 304-318.

[9] Tang P T P. Table-lookup algorithms for elementary functions and their error analysis[R]. Argonne National Lab., IL(USA), 1991.

[10] Sarma D D, Matula D W. Faithful bipartite ROM reciprocal tables[C]. Proceedings of the 12th Symposium on Computer Arithmetic. IEEE, 1995: 17-28.

[11] Oberman S F, Siu M Y. A high-performance area-efficient multifunction interpolator[C]. In 17th IEEE Symposium on Computer Arithmetic(ARITH). IEEE, 2005: 272-279.

[12] David Kanter. NVIDIA's GT200: Inside a Parallel Processor[Z]. [2021-08-12]. https://www.realworldtech.com/gt200/9/.

[13] NVIDIA Corporation. (2012, April). NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210[EB/OL]. (2020-05-04)[2021-08-12]. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf.

[14] NVIDIA Corporation. (2017, December 7). NVIDIA TESLA V100 GPU ARCHITECTURE. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[15] Jing N,Chen S,Jiang S,et al. Bank stealing for conflict mitigation in GPGPU register file[C]. 2015 IEEE/ACM International Symposium on Low Power Electronics and Design(ISPLED). IEEE,2015: 55-60.

[16] Kung H T. Why systolic architectures?[J]. Computer, 1982(1): 37-46.

[17] Jouppi N P,Young C,Patil N,et al. In-datacenter performance analysis of a tensor processing unit [C]. 44th Annual International Symposium on Computer Architecture(ISCA). IEEE,2017: 1-12.

[18] Kung H T,Leiserson C E. Algorithms for VLSI processor arrays,Sparse Matrix Proceedings[J]. SIAM Press,1978: 256-282.

[19] Kung H T, Leiserson C E. Systolic Arrays for(VLSI)[R]. Carnegie-Mellon Univ Pittsburgh Pa Dept of Computer Science,1978.

[20] Kung H T. The structure of parallel algorithms[M]. Advances in computers. Elsevier, 1980, 19: 65-112.

[21] Wan C R,Evans D J. Nineteen ways of systolic matrix multiplication[J]. International Journal of Computer Mathematics. 1998,68(1-2): 39-69.

[22] Paulius Micikevicius. Mixed-precision training of deep neural networks[Z]. [2021-08-12]. https://developer.nvidia.com/blog/mixed-precision-training-deep-neural-networks/.