GPU中的数据格式与转换

X. Shen W. Chen

一 计算机中的数字表示

二 GPU中的数据格式

三 GPU中的格式转换

四 软硬件实现

计算机中的数字表示

名词概念

1、进位数制（进制）：利用固定的数字符号和统一的规则来计数的方法

2、数码：用不同的数字符号来表示一种数制的数值

3、基数：一种数制中所使用的数码的个数

4、位权：数制中每一位数所具有的值

eg：对于一个二进制，数码：0、1；基数：2；位权：1、2、4、8……

5、常见的进制：二进制（B）、八进制（Q）、十进制（D）、十六进制（H）

eg：2’b10、8’h0010 = 16

为什么计算机中信息采用二进制表示?

①电子元件只有两种稳定状态，即高电位和低电位；②运算法则简单

浮点数与整形数

数据类型	关 键 字	在内存中占用的字节数	取 值 范 围	默 认 值
布尔型	boolean	1个字节（8位）	true，false	false
字节型	byte	1个字节（8位）	-128～127	0
字符型	char	2个字节（16位）	0～2^{16}-1	'\u0000'
短整型	short	2个字节（16位）	-2^{15}～2^{15}-1	0
整型	int	4个字节（32位）	-2^{31}～2^{31}-1	0
长整型	long	8个字节（64位）	-2^{63}～2^{63}-1	0
单精度浮点型	float	4个字节（32位）	1.4013E-45～3.4028E+38	0.0F
双精度浮点型	double	8个字节（64位）	4.9E-324～1.7977E+308	0.0D

定义与用途

- 整形（int）表示离散数值，即无小数部分的数字。适合用于计数、索引。
- 浮点型（float）表示实数，适用于科学计算、图形处理等需要精确表达的数据。

存储方式

- int按照固定位宽存储（有符号整数最高位为符号位，无符号整数则全部用于表示数值大小）
- float将小数部分、指数部分分开存储

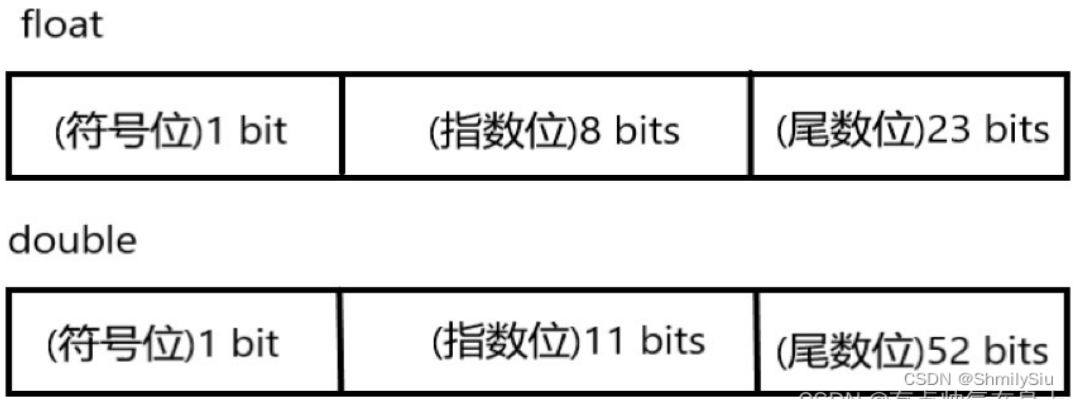
GPU中的数据存储方式

1、float、double型

- 符号位S（Sign）：0 代表正数，1 代表负数
- 指数位 E（Exponent）：决定数据的范围
- 尾数位 M（Mantissa）：决定了数据的精度

2、int型

- 只包括符号位、指数位，无尾数位

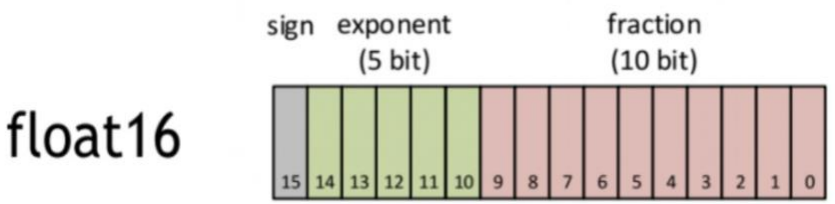


- 在计算机中，任何一个数都可以表示为$1.xxx \times 2^n$的形式，其中n是指数位，xxx是尾数位
- eg：float 9.125，9→1001，0.125→0.001；所以 9.125 表示为 1001.001，其二进制的科学计数法表示为1.001001×2^3
- 浮点数精度：双精度（FP64）、单精度（FP32、TF32）、半精度（FP16、BF16）、8位精度（FP8）、4位精度（FP4、NF4）

深入理解浮点数格式

以FP16为例，指数位E为5bit，范围为00001~11110即1~30（00000和11111是特殊数据）；

尾数位M为10bits，范围为0~1023



（1）计算FP16可表示的数据范围

FP16 可以表示的数据大小为：$(-1)^S * 2^{E-15} * (1 + \frac{M}{2^{1024}})$

因此 FP16 可以表示的最大的正数为：$0\ 11110\ 1111111111 = (-1)^0 * 2^{30-15} * (1 + \frac{1023}{1024}) = 65504$

可以表示的最小的负数为：$1\ 11110\ 1111111111 = (-1)^1 * 2^{30-15} * (1 + \frac{1023}{1024}) = -65504$

所以 FP16 可以表示的数据范围为 $[-65504, 65504]$

与 FP16 相比，FP32 和 BF16 可以表示的数据范围为 $[-3.4 \times 10^{38}, 3.4 \times 10^{38}]$

（2）特殊情况分析

指数位 E 为00000或11111时

- $E = 00000$时，FP16 可以表示的数据大小为：$(-1)^S * 2^{1-15} * (0 + \frac{M}{2^{1024}})$
- $E = 11111$时，若 M 全为 0，表示 ± inf；若 M 不全为 0，表示 NAN

（3）数值精度（即两个不同FP16数值的最小间隔）

FP16 可以表示的最小的正数为：$0\ 00001\ 0000000000 = (-1)^0 * 2^{1-15} * (1 + \frac{0}{1024}) = 6.10 \times 10^{-5}$

零的表示方式

指数+尾数 的表示方法会造成一个数的表示不唯一 → IEEE754

IEEE754 规定：如果 E=0 并且 M=0，则这个数的真值为 ±0

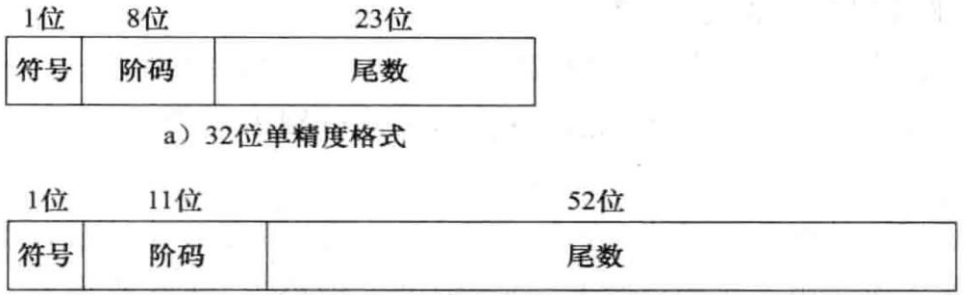


图 2.3　IEEE 754 浮点数格式

因此 +0 的机器码为：0 00000000 000 0000 0000 0000 0000 0000

-0 的机器码为：1 00000000 000 0000 0000 0000 0000 0000

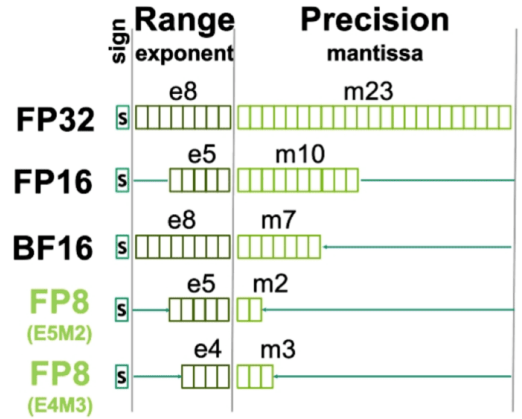
注意：计算时浮点数不能精确表示 0，而是以很小的数来近似表示 0，故+0与-0浮点数的真值不同

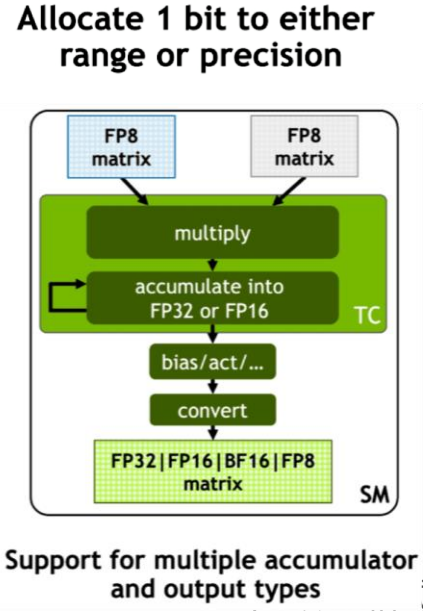
GPU中的数据格式

GPU中的数据格式

最早的 GPU 默认使用 FP32 类型进行运算，但随着模型越来越大，FP32 类型占内存资源大且运算速度慢的问题逐渐暴露

数据类型	bits	S	E	M	设计原理	说明
FP32	32	1	8	23		大部分CPU/GPU/深度学习框架中默认使用FP32，FP32可以作为精度baseline
FP16	16	1	5	10		推理，混合精度训练
TF32	19	1	8	10	TF32与FP32的数值范围相同，与FP16的数值精度相同	TF32 (TensorFloat) 是 Nvidia 在 Ampere 架构的 GPU 上推出的用于 TensorCore 的数据格式，在 A100 上使用 TF32 的运算速度是在 V100 上使用 FP32 CUDA Core 运算速度的 8 倍。
BF16	16	1	8	7	BF16与 FP32数值范围相同；数值范围远大于 FP16，但精度略低于 FP16	BF16 是由Google Brain开发的，是一种最适合大模型训练的数据类型，但目前只适配于 Ampere 架构的 GPU（如 A100）
INT8	8	1	7	0		模型推理、混合计算





GPU中的格式转换

GPU中的格式转换

一、特殊值分析

1. Normalized 正规数
2. Subnormalized 次正规数：指数位全为0
3. Inf：指数位全为1，尾数位全为0
4. NaN：指数位全为1，尾数位不为0

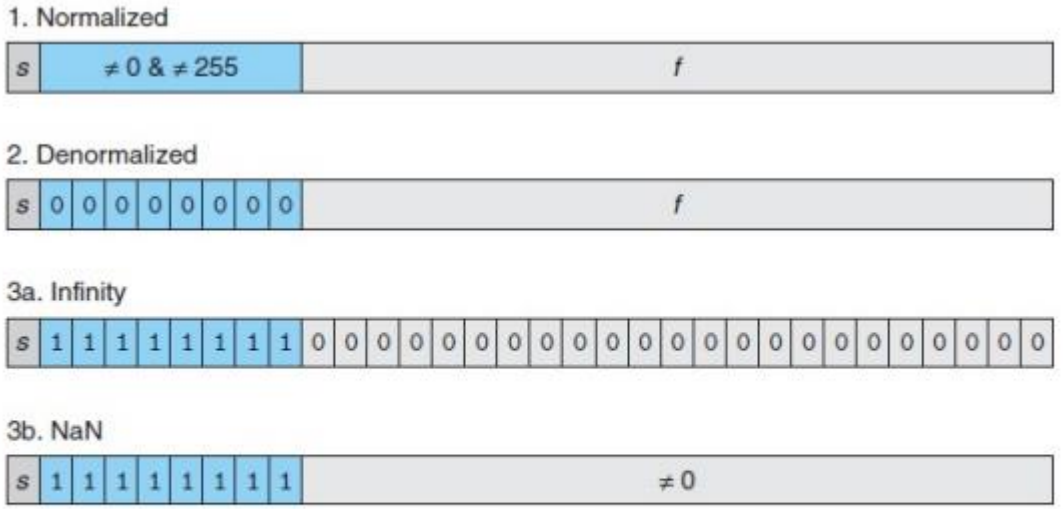


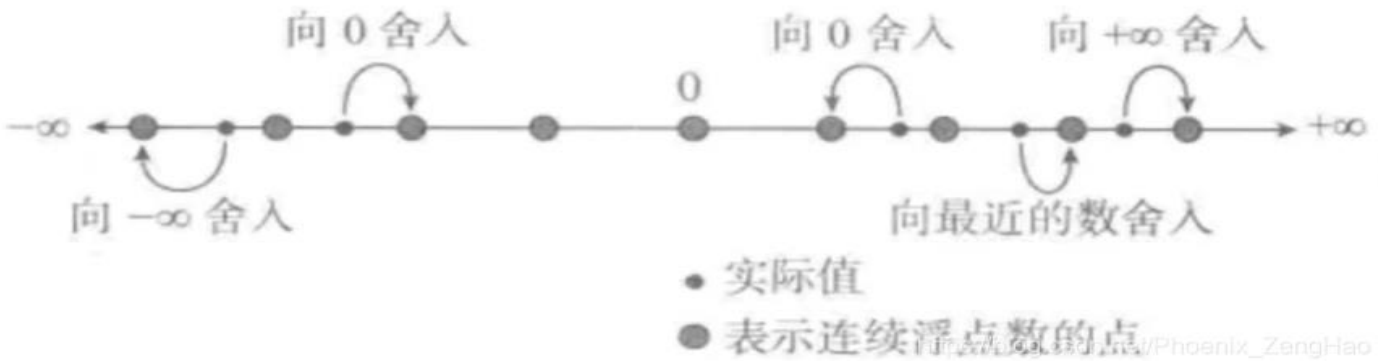
Figure 2.32 **Categories of single-precision, floating-point values.** The value of the exponent determines whether the number is (1) normalized, (2) denormalized, or a (3) special value.

二、扩位与缩位

- 扩位（低精度转高精度）：**位扩展**，无精度损失
- 缩位（高精度转低精度）：存在精度损失、溢出、舍入误差。需处理特殊值

舍入原理

1. 向偶数舍入：向最近的偶数舍入
2. 向零舍入：直接截断多余位
3. 向正无穷舍入：总是向上舍入
4. 向负无穷舍入：总是向下舍入

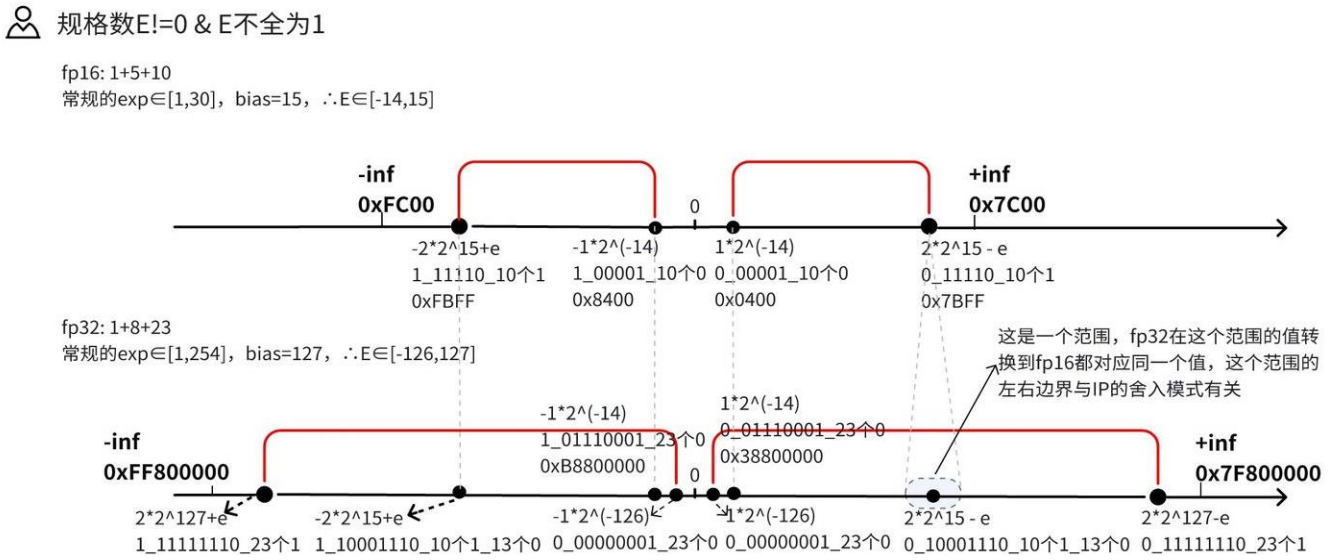


- 若原来的值是舍入值的中间值时，采取向偶数舍入
- 其他情况，则有选择性的使用向上和向下舍入，但总是会向最接近的值舍入

高精度转低精度（以FP32→FP16为例）

1、正规数normal

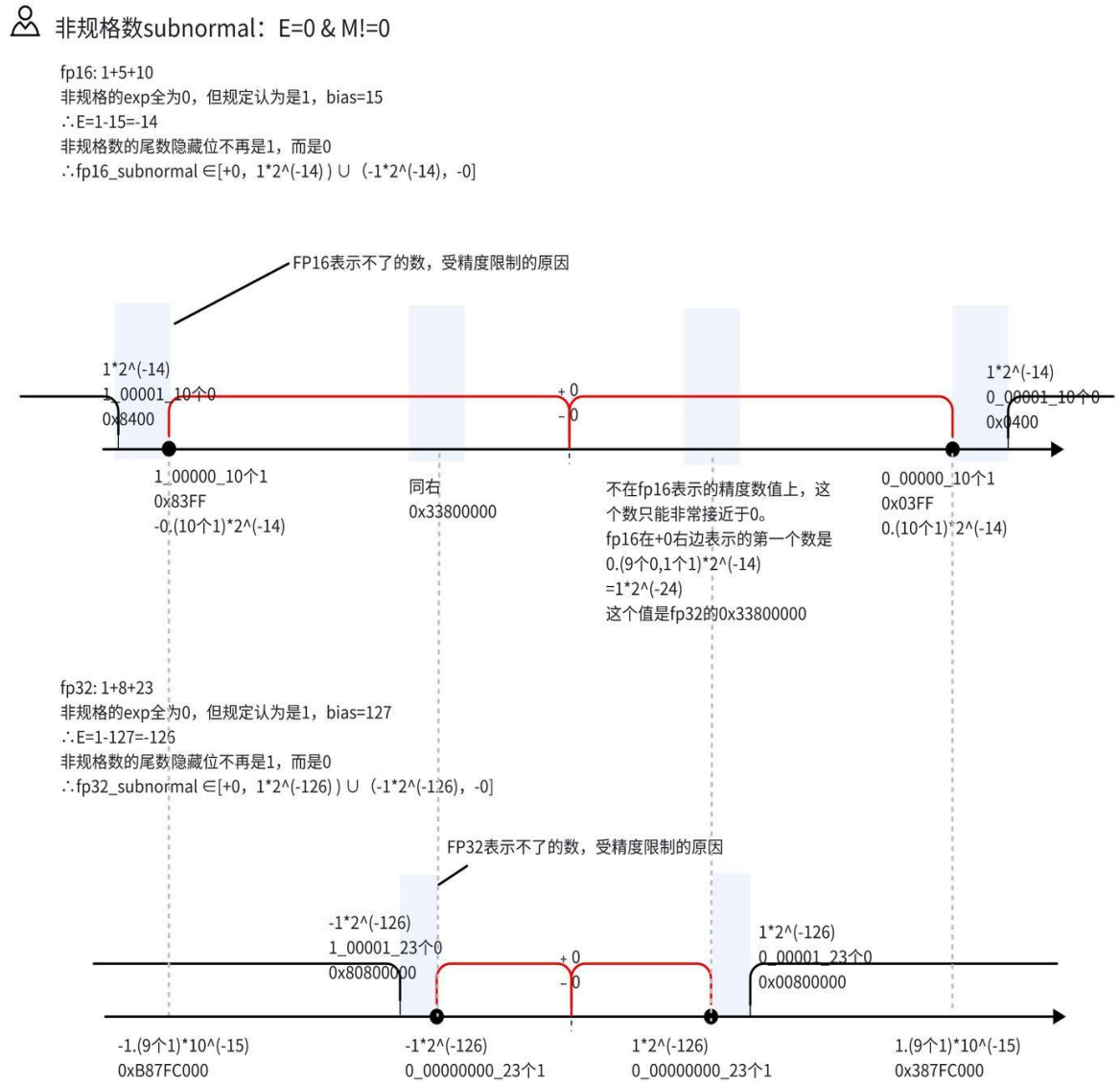
- 边界值
  - fp16的最小值：0x0400，用fp32表示为1*2^(-14)，则尾数M=23'b0（隐藏前导是1，所以M是1.0），指数是-14+Bias=-14+127=113=8'b0111_0001（fp32规定的Bias是-127），所以用fp32表示是0x38000000
  - fp16的最大值：0x7BFF，同理，用fp32表示是0x0477FE00
- 舍入
  - 存在fp32向fp16转换时的精度损失，比如上图中画蓝色阴影部分的数，在转换过程根据舍入模式，都会被舍入到fp16的0x7BFF，存在精度损失



高精度转低精度（以FP32→FP16为例）

2、次正规数subnormal

- fp32的Subnormal转换为fp16之后都应该是0
- fp16的Subnormal在fp32的范围内都是Normal



低精度转高精度（以FP16→FP32为例）

分析：

- fp16的数在fp32内都能够完全表示
- fp16的Subnormal在fp32的范围内都是Normal
- fp16的inf转换为fp32仍为inf，符号不变
- fp16的NAN转换为fp32仍为NAN，符号不变

结论：

FP16→FP32只需要注意特殊值Inf和NaN的转换，其他数值直接通过指数位和尾数位移位得到

步骤：

1. FP32高16位为FP16的位表示，低位16位补零
2. 保持符号位不变，指数位偏移（FP32 bias=127，FP16 bias=15），尾数位从10位补零扩充到23位

软硬件实现

软件实现

```
union FPConvertHelper {
    float value;
    uint32_t data;
};

template<typename Dtype, typename Stype, typename Otype>
__device__ __inline__
float QuantizeScalarFloating(
    const Dtype value, const Stype scale, const Otype offset,
    const int exponent, const int mantissa,
    const float clip_min, const float clip_max,
    const Rounding rounding){
    /**
     * PPQ Quantization Function implementation.
     * This function convert an float value to low-precision float
     */
    FPConvertHelper helper; FPConvertHelper rounding_helper;
    float Unscaled_FP32 = static_cast<float>(value) / scale;
```

1、首先定义union结构FPConvertHelper

- FP64→FP8，只需要将float改成double即可
- FP16→FP8，无法直接采用union结构转换

2、若想实现FP16→FP8

- 思路1：用float的value保存FP16的值，用uint16_t类型的data保存其编码。放缩用value计算，计算后的结果转换为对应的data编码
- 思路2：FP16→FP32→FP16

3、从高精度到低精度，首先除以缩放系数scale进行量化操作

软件实现

```
helper.value = Unscaled_FP32;
    int32_t exponent_min  = -(1 << (exponent - 1)) + 1;
int32_t exponent_max  = (1 << (exponent - 1));

// Following code will process exponent overflow
/* For FP8 E4M3, the maximum exponent value should be 8.
/* The Maximum number of FP8 E4M3 should be (0 1111 111) = 480
/* We call it as theoretical_maximum, FP8 E4M3 can not represent a number larger t
uint32_t fp32_sign    = 0;
int32_t fp32_exp      = (exponent_max + 127) << 23;
int32_t fp32_mantissa = ~(0x007FFFFF >> mantissa) & 0x007FFFFF;
helper.data = fp32_sign + fp32_mantissa + fp32_exp;  //按位拼接 得到最大FP8正规数转换为FP32后的格式
float theoretical_maximum = helper.value;

if (Unscaled_FP32 > min(clip_max, theoretical_maximum))
    return min(clip_max, theoretical_maximum);
if (Unscaled_FP32 < max(clip_min, -theoretical_maximum))
    return max(clip_min, -theoretical_maximum);
```

4、判断量化后的FP32值是否发生了上溢。若设置了范围限制clip_min~clip_max，则根据符号位取clip_max或clip_min

软件实现

```
// Code start from here will convert number within fp8 range.
// Following code will Split float32 into sign, exp, mantissa
/* IEEE 754 Standard: 1 bit sign, 8 bit exponent, 23 bit mantissa */

/* In binary 10000000 00000000 00000000 00000000 = 0x80000000 in Hex */
/* In binary 01111111 10000000 00000000 00000000 = 0x7F800000 in Hex */
/* In binary 00000000 01111111 11111111 11111111 = 0x007FFFFF in Hex */

/* Tool: https://www.h-schmidt.net/FloatConverter/IEEE754.html */
helper.value  = Unscaled_FP32;
fp32_sign     = helper.data & 0x80000000;
fp32_exp      = helper.data & 0x7F800000;
fp32_mantissa = helper.data & 0x007FFFFF;

// Following code will process exponent underflow
/* Float underflow means fp32_exp is smaller than exponent_min          */
/* Where exponent_min is the minimum exponent value of quantized float. */
/* For FP8 E4M3, the minimum exponent value should be -7.               */
/* The Min Subnormal value of FP8 E4M3 should be (0 0000 001) = 2^-9    */
/* The Min normal value of FP8 E4M3 should be (0 0001 000) = 2^-6       */
    if (((fp32_exp >> 23) - 127) < exponent_min + 1){  //判断是否小于FP8正规数的最小指数
    // following divide might have some problems
    // but it is the simplest method with very limited error.
    float min_subnormal = 1.0f / (1 << ((1 << (exponent - 1)) + mantissa - 2));  //将放缩后的数，除以最小次正规数进行舍入，再乘以最小次正规数。
    return _round2int(Unscaled_FP32 / min_subnormal, rounding) * min_subnormal;
    }
```

5、随后判断下溢。如果下溢，要将对应的FP32的正规数对应到FP8的相应非规范数，FP32的非正规数全映射到FP8的0

下溢的两种情况:

（1）放缩后的数（FP32正规数）落在FP16的非正规数区间内，在这个区间内进行舍入

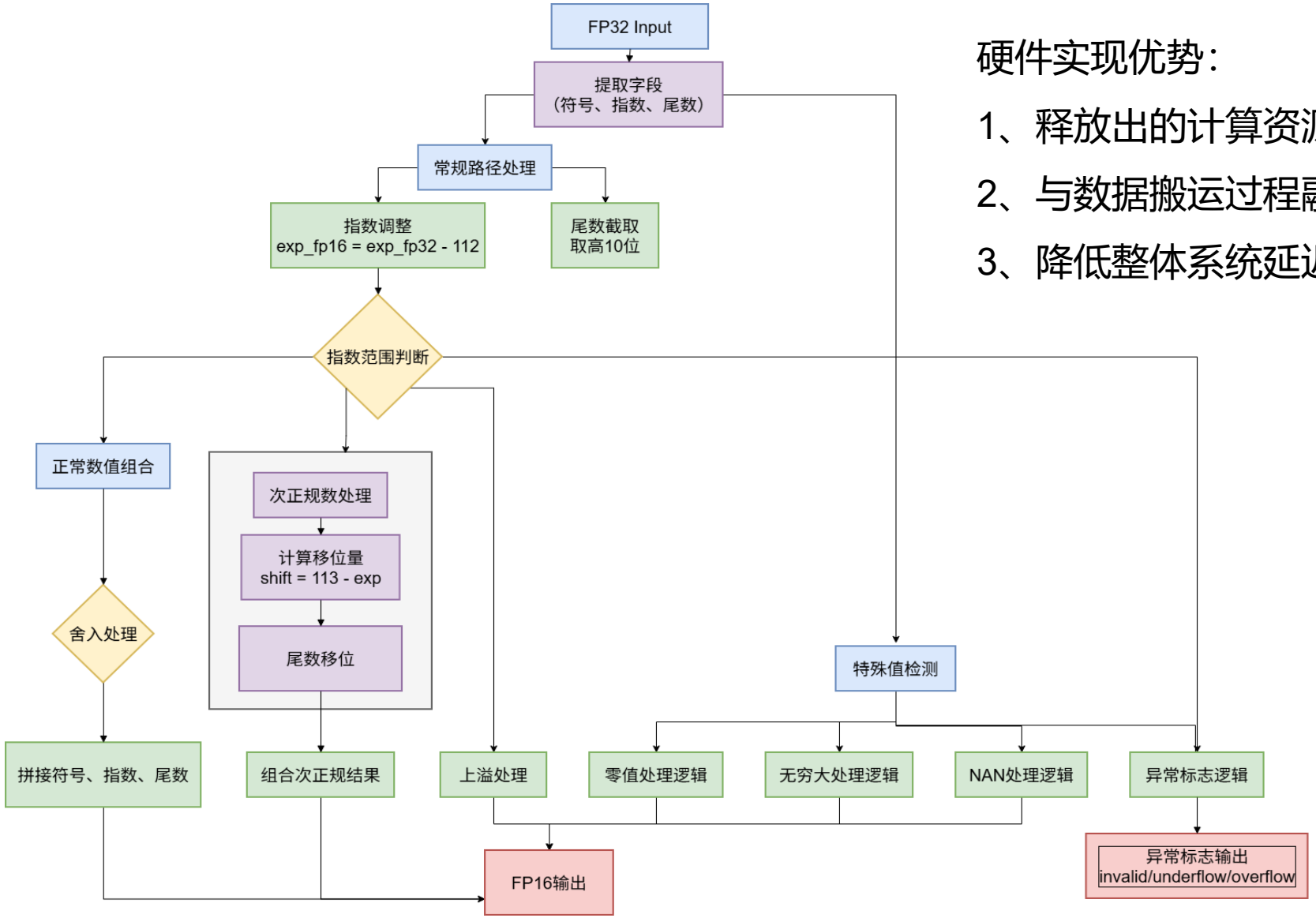
（2）放缩后的数（FP32正规数或次正规数）对应FP8的0

```
    /* high precision mantissa convert to low precision mantissa requires rounding
    /* Here we apply a tricky method to round mantissa:
    /* We create another float, which sign = 0, exponent = 127, mantissa = fp32_mantis
    /* Then we directly round this float to int, result here is what we want, you can
    rounding_helper.data = ((fp32_mantissa << (mantissa)) & 0x007FFFFF) + 0x3F800000;
    uint32_t round_bit = _round2int(rounding_helper.value - 1, rounding);

    // process mantissa
    fp32_mantissa = ((fp32_mantissa >> (23 - mantissa)) + round_bit) << (23 - mantissa
    helper.data = fp32_sign + fp32_mantissa + fp32_exp;

    return CLIP<float>(helper.value, clip_min, clip_max);
}
```

6、最后是FP32正规数对应FP8正规数的情况

- 截取尾数的高3位，舍入处理，得到FP8的尾数部分
- 指数部分，FP32偏移为127，FP8偏置为7，所以要加上127-7=120得到FP8的指数部分
- 符号位不变

硬件实现



硬件实现优势：

1、释放出的计算资源可用于执行更复杂的数学运算

2、与数据搬运过程融合，缓解内存带宽瓶颈

3、降低整体系统延迟与功耗

为什么大模型训练时会因精度问题崩溃?

原因一：数值下溢

问题：在训练非常深的网络（如Transformer）时，需要连续进行矩阵乘法和激活函数计算。当使用FP16/FP8时，其能表示的最小正数比FP32大得多

- 后果：梯度消失，变为0。当零梯度用于更新权重时，模型无法学习

原因二：数值上溢

问题：计算产生的数值超过了该格式能表示的最大值时，会上溢为Inf（无穷大）

- 后果：权重更新中若出现Inf，模型状态被破坏，训练立即失败

原因三：舍入误差累积

问题：浮点数表示不精确，每次计算都可能引入舍入误差。存在大数吃小数，训练中可能导致权重更新、梯度累积（梯度汇总时可能因为舍入误差而丢失关键信息）

- 后果：训练过程变得不稳定、收敛缓慢，或者最终收敛到一个很差的局部最优点

计算顺序对结果的影响?

底层原因：**浮点数不满足结合律**

例1：求和顺序

Q：计算 [1e10, -1e10, 1, 2, 3] 的和，使用FP32。

- 顺序1：(1e10 + (-1e10)) + (1 + 2 + 3) = (0) + 6 = 6
- 顺序2：(1e10 + (-1e10 + 1)) + (2 + 3) = (1e10 + (-1e10)) + 5 = 5

大数吃小数！

例2：乘法顺序

Q：计算A * B * C

- 顺序1：(A * B) * C
- 顺序2：A * (B * C)

编译器需要花费大量精力

寻找最优的矩阵乘法顺序

课程基本信息

- 课程名称：基于RISC-V的开源GPU架构与设计探索
- 授课地点：深圳市学苑大道南山智园C3栋20层2005（清华大学深圳国际研究生院）
- 授课时间：十周（每周六，9:00-12:00 14:00-18:00）
- 本课程聚焦RISC-V开源架构与GPU设计的**热门交叉前沿**，构建从指令集到众核计算架构的**探索性知识体系**，以**最短路径**实现RISC-V GPU体系架构入门，并探讨3D/3.5D Chiplet、开源EDA等技术在开源GPU中的应用潜力
- 课程核心内容涵盖：RISC-V指令集架构（ISA）与GPU并行计算架构的融合设计，开源GPU核心模块（如流处理器、存储架构、TensorCore），核心编译器的原理与探索实现。通过C-Model仿真与FPGA实现，学生将掌握从架构建模、功能验证到部署实现的精简芯片设计流程，并探索开源RISC-V GPU的大模型（Transformer）应用，为进入AI芯片/GPU编程等前沿领域的研究生学习或职业发展奠定**核心竞争力与先发优势**
- https://github.com/chenweiphd/OpenRVGPUCourse