life.augmented

AN4044
应用笔记

基于STM32微控制器上的浮点单元的性能演示

前言

本应用笔记介绍了如何使用STM32 Cortex®-M4和STM32 Cortex®-M7微控制器中可用的浮点单元（FPU），并对浮点运算作了简要介绍。

X-CUBE-FPUDEMO固件是为改进双精度FPU而开发，并能演示使用此硬件实现所带来的改进。

*第 4 节：应用程序示例*中给出了两个示例。

目录

表格索引

图片索引

1 浮点算法

浮点数用来表示非整数。它们包含三个字段：

- 符号
- 指数
- 小数

这样的表示可实现非常宽的数字编码范围，使得浮点数成为处理实数的最佳方式。可以使用集成在处理器中的浮点单元（FPU）来加速浮点计算。

1.1 定点或浮点

可替代浮点的一种方式是定点，其中指数字段是固定的。但是如果要在无FPU的处理器上获得更好的定点计算速度，那么数字范围及其动态范围就会较低。因此，使用定点技术的开发人员必须要仔细检查算法中的缩放/饱和问题。

表**1.** 整数动态

编码	动态
Int8	48 dB
Int16	96 dB
Int32	192 dB
Int64	385 dB

C语言为浮点运算提供了**float**和**double**类型。更高层次上，模块化工具，如MATLAB或Scilab，主要使用float或double来生成C代码。不支持浮点意味着会修改所生成的代码，将其改编为定点。所有定点运算都必须由程序员手动编码。

表**2.** 浮点数动态

编码	动态
半精度	180 dB
单精度	1529 dB
双精度	12318 dB

浮点运算直接用于代码中时， 可以减少项目的开发时间。它是实现任何数学算法的最有效方法。

1.2 浮点单元（FPU）

对于两个数字之间的任意操作，浮点计算需要大量资源。例如，我们需要：

- 对齐这两个数字（使它们具有相同的指数）
- 执行运算
- 对结果进行舍入
- 对结果进行编码

在无FPU的处理器上，所有这些操作都由软件通过C编译器库来完成，程序员不可见；但是其性能非常低。

在有FPU的处理器上，对于大多数指令，所有操作由硬件在一个周期内全部完成。C编译器不使用其自己的浮点库，而是直接生成FPU本机指令。

在有FPU的微处理器上执行数学算法时，程序员不必为芯片性能和开发时间上纠结。FPU带来了可靠性，允许直接使用高级工具（例如MATLAB或Scilab）所生成的代码，并具有最高的性能水平。

2 浮点运算的IEEE标准（IEEE 754）

浮点运算的使用自早期以来就一直是计算机科学的需要。30年代末，当Konrad Zuse在德国开发Z系列时，浮点技术就已经存在。但是支持浮点运算的硬件实现复杂，导致其被放弃使用了数十年。

50年代中期，IBM，利用其704，在大型机中引入了FPU；而到了70年代，多种平台都可支持浮点运算，但是要使用它们自己的编码技术。

其统一发生在1985年，当时IEEE发布了标准754来定义支持浮点运算的通用方法。

2.1 概述

多年来各种类型的浮点实现促使IEEE将以下元素进行标准化：

- 数字格式
- 算术运算
- 数字转换
- 特殊值编码
- 四种舍入模式
- 五种异常及其处理

2.2 数字格式

所有值均包含三个字段：

- 符号：s
- 偏置指数：
  - 指数和 = e
  - 常量值 = bias
- 分数（或小数）：f

这些值可以以各种长度进行编码：

- 16位：半精度格式
- 32位：单精度格式
- 64位：双精度格式

图1. IEEE.754单精度和双精度浮点编码

IEEE定义了五种不同的数据类型：

- 归一化数字
- 非归一化数字
- 零
- 无穷数
- NaN（非数字）

不同类别的数字通过这些字段的特定值来识别。

2.2.1 归一化数字

归一化数字是一个“标准的”浮点数字。其值由上式给出：

$$\text{归一化数字} = (-1)^s \times (1 + \sum f_i \times 2^{-i}) \times 2^{e-bias} \quad (i > 0)$$

偏置是针对每种格式（8位，16位，32位和64位）定义的固定值。

表3. 归一化数字范围

模式	指数	Exp.偏置	Exp.距离	小数	最小值 值	最大值 值
半	5位	15	-14, +15	10位	$6,10.10^{-5}$	65504
单	8位	127	-126,+127	23位	$1,18.\ 10^{-38}$	$3,40.10^{38}$
双	11位	1023	-1022,+1023	52位	$2,23.10^{-308}$	$1,8.10^{308}$

示例：-7的单精度编码

- 符号位 = 1
- $7 = 1.75 \times 4 = (1 + 1/2 + 1/4) \times 4 = (1 + 1/2 + 1/4) \times 2^2$
- 指数 = 2 + 偏置 = 2 + 127 = 129 = 0b10000001
- 小数 = $2^{-1} + 2^{-2}$ = 0b11000000000000000000000
- 二进制值 = 0b 1 10000001 11000000000000000000000
- 十六进制值 = 0xC0E00000

2.2.2 非归一化数字

非归一化数字是用来表示太小而不能进行归一化表示的数据（此时指数等于0）。其值由下式给出：

$$非归一化数字 = (-1)^s \times (\sum f_i \times 2^{-i}) \times 2^{-bias} \quad (i > 0)$$

表4. 非归一化数字范围

模式	最小值
半	$5{,}96.10^{-8}$
单	$1{,}4.10^{-45}$
双	$4{,}94.10^{-324}$

2.2.3 零

零值是带符号的，通过正负提示饱和度。

2.2.4 无穷数

带符号的无穷值用来表示 +∞ 或 -∞。无穷值是因为溢出或者0做为除数的结果。指数设置为其最大值，而小数部分为空。

2.2.5 NaN（非数字）

NaN用来表示运算的未定义结果，例如0/0或负数的平方根。指数置为其最大值，而小数不为空。小数的MSB表示它是否为Quiet NaN（会通过下次操作进行传播）或Signaling NaN（会产生一个错误）。

2.2.6 总结

表5. IEEE.754数字格式的数值范围

符号	指数	分数	号
0	0	0	+0
1	0	0	-0
0	最大值	0	+∞
1	最大值	0	-∞
[0, 1]	最大值	!=0 & MSB=1	QNaN
[0, 1]	最大值	!=0 & MSB=0	SNaN
[0, 1]	0	!=0	非归一化数字
[0, 1]	[1，Max-1]	[0，Max]	归一化数字

2.3 舍入模式

定义了四种主要的舍入模式：

- 就近舍入
- 直接向 +∞ 舍入
- 直接向 -∞ 舍入
- 直接向0舍入

就近舍入是默认的舍入模式（最常用）。如果最近的两个值同样接近，则选择LSB等于0的那个。

舍入模式非常重要，因为它可以改变算术运算的结果。可通过FPU配置寄存器来改变它。

2.4 算术运算

IEEE.754标准定义了6种算术运算：

- 加
- 减
- 乘
- 除
- 求余
- 开平方根

2.5 数字转换

IEEE标准还定义了一些格式转换操作和比较：

- 浮点和整数转换
- 将浮点值舍入为整数值
- 二进制到十进制转换
- 比较

2.6 异常和异常处理

可支持5种异常：

- 无效运算：运算结果为NaN
- 除以0
- 上溢出：运算结果为 ±∞ 或 ±Max，取决于舍入模式
- 下溢出：运算结果为非归一化数字
- 不精确结果：由舍入导致

可以通过两种方法来处理异常：

- 产生一个捕获。捕获处理程序会返回一个代表异常结果的值。
- 产生一个中断。中断处理程序并不返回代表异常结果的数值。

2.7 总结

IEEE.754标准定义了如何编码和处理浮点数。

硬件中的FPU实现可加速IEEE 754浮点计算。因此，它可实现整个IEEE标准或子集。关联软件库管理非加速功能。

对于“基本的”使用，浮点处理对用户来说是透明的，就好像在C代码中使用float一样。对于更高级的应用，可以通过捕获或中断来处理异常。

3 STM32 Cortex®-M浮点单元（FPU）

表 6显示了STM32 Cortex®-M4和Cortex®-M7的FPU实现。

表6. STM32 Cortex®-M4/-M7 中的FPU实现

特性	可配置选项	STM32实现	
		STM32F3xx STM32F4xx STM32F74x/5x STM32L4xx	**STM32F76x/7x**
FPU	无FPU	-	-
	仅单精度（SP）	有	-
	SP和DP	-	有

Cortex® M4 FPU是采用ARM® FPv4-SP单精度FPU实现。

它拥有自己的32位单精度寄存器集（S0-S31），可对操作数和结果进行处理。这些寄存器可以看作16个双字寄存器（D0-15），用于加载/存储操作。

Status & Configuration寄存器存储FPU配置（舍入模式和特殊配置）、条件码位（负、零、进位和溢出）和异常标志。

一些IEEE.754操作不被硬件支持，可由软件来完成：

- 求余
- 将浮点数舍入为整数值浮点数
- 二进制到十进制转换和十进制到二进制转换
- 单精度值和双精度值的直接比较

异常由中断进行处理（不支持捕获）。

Cortex®-M7双精度FPU是采用ARM® FPv5浮点单元实现。FPv5完全支持单精度和双精度数，还提供了定点和浮点数据格式之间的转换，并可支持浮点常量指令。

FPU可支持符合IEEE754标准的32位单精度和64位双精度浮点值运算。

FPU提供了扩展寄存器文件，包含32个单精度寄存器。它们可以看作：

- 16个64位双字寄存器（D0-D15），与FPv4一样，无需额外的寄存器。
- 32个32位单字寄存器（S0-S31），加载/存储指令与FPv4支持的指令相同，FPv4已包含对64位数据类型的支持。

FPv5可支持非归一化数和所有IEEE Standard 754-2008舍入模式。

3.1 特殊操作模式

Cortex®M4 FPU完全符合IEEE.754规范。但是，一些非标准操作模式可能会被激活：

- 可选半精度格式（AHP控制位）
  - 特殊16位模式，无指数值，不支持非归一化数字。

$$可选半精度格式 = (-1)^s \times (\sum f_i \times 2^{-i}) \times 2^{16}$$

- 清除为零模式（FZ控制位）
  - 所有非归一化数都作零处理。输入和输出清除关联到一个标志。
- 默认NaN模式（DN控制位）
  - 任何以NaN为输入的操作，或者产生NaN的操作都返回默认NaN（Quiet NaN）。

3.2 浮点状态和控制寄存器（FPSCR）

FPSCR存储FPU的状态（条件位和异常标志）和配置（舍入模式和可选模式）。

因此，当上下文发生变化时，此寄存器可保存在堆栈中。

FPSCR通过专用指令来访问：

- 读：VMRS Rx，FPSCR
- 写：VMSR FPSCR，Rx

表7. FPSCR寄存器

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
N	Z	C	V	Res.	AHP	AHP	FZ	RM		保留					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留								IDC	保留		IXC	UFC	OFC	DZC	IOC
											rw	rw	rw	rw	rw

3.2.1 代码条件位：N，Z，C，V

它们在比较操作后进行设置。

3.2.2 模式位：AHP，DN，FZ，RM

它们配置可选模式（AHP，DN，FZ）和舍入模式（RM）。

3.2.3 异常标志

在下列情况下发生异常时会出现异常标志：

- 清除为零（IDC）
- 不精确结果（IXC）
- 下溢出（UFC）
- 上溢出（OFC）
- 除以0（DZC）
- 无效操作（IOC）

注： 异常标志不会被下一条指令复位。

3.3 异常管理

异常不能被捕获。它们由中断控制器进行管理。

五种异常标志（IDC，UFC，OFC，DZC，IOC）进行OR运算，并连接到中断控制器。不支持个别异常标志位屏蔽，，FPU中断的使能/禁用在中断控制器级完成。

因为IXC标志发生率很高，它没有连接到中断控制器，所有不会产生中断。如有需要，必须通过轮询对其进行管理。

如果使能了FPU，可以使用以下三种方法将其上下文保存到CPU堆栈中：

- 无浮点寄存器保存
- 预留性保存/恢复（实际上只是在堆栈中为其预留相应存储空间）
- 浮点寄存器自动保存/恢复

堆栈帧包含17个条目：

- FPSCR
- S0至S15

3.4 程序员模型

当MCU从复位启动之后，必须使能FPU，通过协处理访问控制寄存器CPACR指定代码对FPU的访问级别（拒绝、特权、全权）。

FPSCR可以配置为备选模式或舍入模式。

FPU还拥有5个系统寄存器：

- FPCCR（FP上下文控制寄存器）用来指示已分配FP堆栈帧时的上下文和上下文保存设置。
- FPCAR（FP上下文地址寄存器）指向为S0保留的堆栈位置。
- FPDSCR（FP默认状态控制寄存器），其中存储了可选半精度模式、默认NaN模式、清除为零模式和舍入模式的默认值。
- MVFR0 & MVFR1（媒体和VFP功能寄存器0和1），其中详细介绍了FPU所支持的功能。

3.5 FPU指令

FPU可支持算术运算、比较、转换和加载/存储指令。

3.5.1 FPU算术指令

FPU提供以下算术指令：

- 绝对值（1个周期）
- 一个或多个浮点数取反（1个周期）
- 加（1个周期）
- 减（1个周期）
- 乘法，乘法加/减，乘法加/减，然后取反（3个周期）
- 除（14个周期）
- 开平方根（14个周期）

表 8显示了一些浮点单精度数据处理指令：

表8. 一些浮点单精度数据处理指令

指令	说明	周期
VABS.F32	绝对值	1
VADD.F32	加	1
VSUB.F32	减	1
VMUL.F32	承	1
VDIV.F32	除	14
VCVT.F32	整数/定点数转换	1
VSQRT.F32	开平方根	14

表 9显示了一些浮点双精度数据处理指令：

表9. 一些浮点双精度数据处理指令

指令	说明	周期
VADD.F64	增加	3
VSUB.F64	减	3
VCVT.F<32|64>	整数/定点数转换	3

所有MAC操作都可以是标准的或者是多种操作糅合的（在MAC结束时进行舍入，可以获得更好的准确性）。

3.5.2 FPU的比较与转换指令

FPU具有比较指令（1个周期）和一个转换指令（1个周期）。

转换可以在整数、定点数、半精度数和浮点数之间进行。

3.5.3 FPU加载/存储指令

FPU遵循标准加载/存储架构：

- 基于多个双精度、多个浮点数、单个双精度数或单个浮点数的双向加载/存储
- 基于浮点或双精度立即数以及内核寄存器的双向加载/存储
- 基于控制器或状态寄存器的双向加载/存储
- 从堆栈弹出或向堆栈推入双精度数或浮点数

4 应用程序示例

本应用笔记提供了两个示例，说明STM32 FPU所带来的优势。

第一个示例是Julia集，其中重点说明了硬件FPU和软件FPU之间的性能比较。

第二个示例是Mandelbrot集，重点说明利用双精度浮点FPU相比单精度FPU的精度提升。

4.1 Julia集

目标是计算一个简单的数学分形：Julia集。

这种数学对象的生成算法非常简单：对于复平面的每个点，我们对一个定义序列的发散速度进行估计。该序列的Julia集方程为：

$$z_{n+1} = z_n^2 + c$$

对于复平面的每个 x + i.y 点，我们利用 $c = c_x + i.c_y$计算序列：

$$x_{n+1} + i.y_{n+1} = x_n^2 - y_n^2 + 2.i.x_n.y_n + c_x + i.c_y$$

$$x_{n+1} = x_n^2 - y_n^2 + c_x，\ y_{n+1} = 2.x_n.y_n + c_y$$

一旦产生的复数值超出给定圆（数字量级大于圆半径），序列就会发散，并且达到此极限所经过的迭代次数与该点相关。该值可被表示为一种颜色，以图形方式显示复平面上的点的发散速度。

经过一定次数的迭代后，如果得到的复数值仍然在圆中，则计算停止，认为该序列不发散：

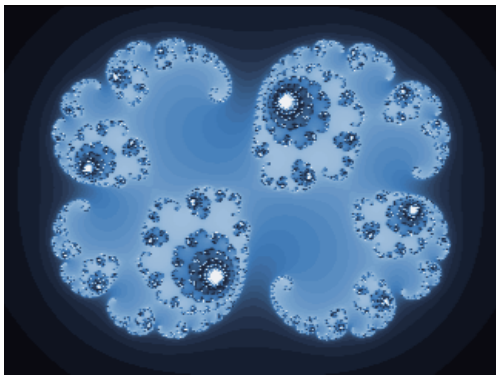
```
void GenerateJulia_fpu(uint16_t size_x, uint16_t size_y, uint16_t offset_x, uint16_t offset_y,
uint16_t zoom, uint8_t * buffer)
{
    float      tmp1, tmp2;
    float      num_real, num_img;
    float      radius;

    uint8_t          i;
    uint16_t       x,y;

    for (y=0; y<size_y; y++)
    {
        for (x=0; x<size_x; x++)
        {
            num_real = y - offset_y;
            num_real = num_real / zoom;
            num_img = x - offset_x;
            num_img = num_img / zoom;
            i=0;
            radius = 0;
        while ((i<ITERATION-1) && (radius < 4))
            {
                tmp1 = num_real * num_real;
                tmp2 = num_img * num_img;
                  num_img = 2*num_real*num_img + IMG_CONSTANT;
```

```
                num_real = tmp1 - tmp2 + REAL_CONSTANT;
                radius = tmp1 + tmp2;
                i++;
            }
            /* Store the value in the buffer */
            buffer[x+y*size_x] = i;
        }
    }
}
```

这种算法非常有效地显示了FPU的优点：无需修改代码，只需在编译阶段激活或不激活FPU。

无需额外代码来管理FPU，因为它在默认模式下使用。

图**2.** 其值采用**8 bpp**蓝色编码的**Julia**集（**c=0.285+i.0.01**）



4.2 在STM32F4上实现

为了在STM3240G-EVAL评估板的RGB565屏幕上有更好的渲染，我们使用一个特殊的调色板来编码颜色值。

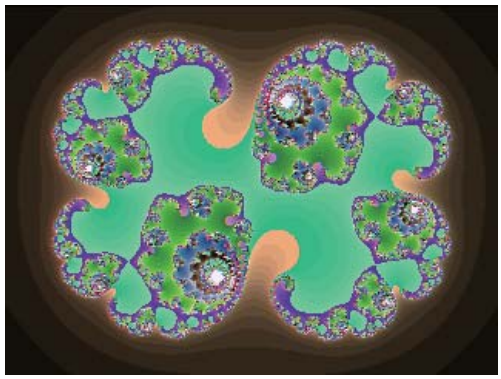
最大迭代值设置为128。因此，调色板会有128个条目。圆半径设置为2。

主程序调用板子的所有初始化函数来设置显示屏和按钮。

- WAKUP按钮从自动模式（连续放大或缩小）切换为手动模式。
- 手动模式下，KEY按钮用来启动另一个计算，选择使用或不适用FPU，对二者进行性能比较。

整个项目都在FPU使能的情况下进行编译，GenerateJulia_noFPU.c除外，它编译时要强制关闭FPU。

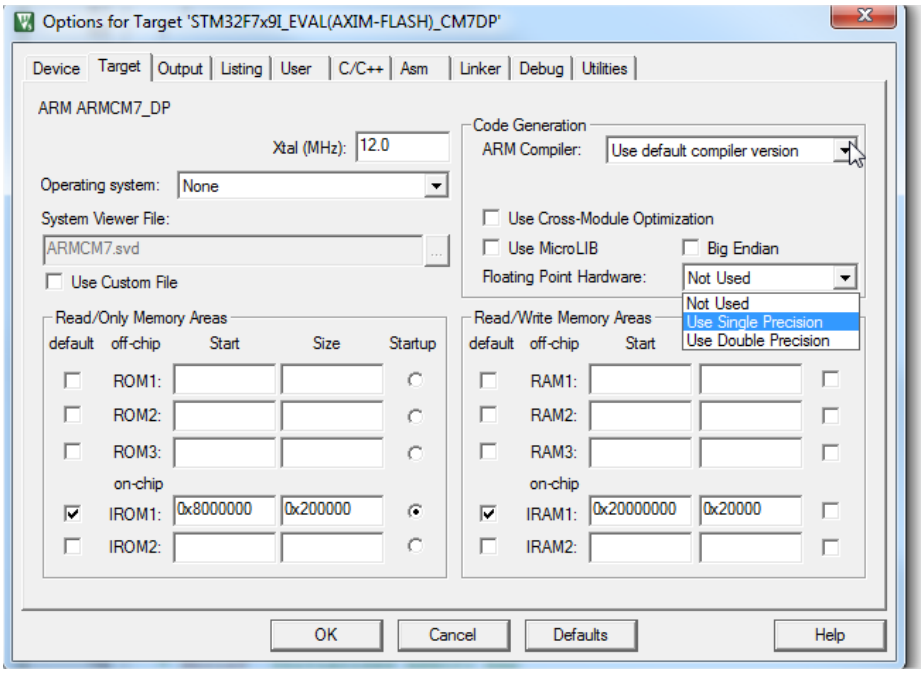
图3. 其值采用RGB565调色板编码的Julia集（c=0.285+i.0.01）



4.3 在STM32F7上实现

在STM32F769i-Eval上实现同样的算法。微控制器以216MHz运行，具有以下两种配置：FPU单精度使能，以及FPU双精度使能。这通过RealView Microcontroller Development Kit（MDK-ARM™）工具链V5.17完成，如图 4所示。

图4. 利用MDK-ARM™工具链V5.17配置FPU



对于STM32F7，只可使用手动模式，一旦检测到触摸屏，就会启动另一个计算。

算法也发生了改变：

```
void GenerateJulia_fpu(uint16_t size_x, uint16_t size_y, uint16_t
offset_x, uint16_t offset_y, uint16_t zoom, uint8_t * buffer)
{
    double      tmp1, tmp2;
    double      num_real, num_img;
    double      radius;
    uint8_t       i;
    uint16_t      x,y;

    for (y=0; y<size_y; y++)
    {
        for (x=0; x<size_x; x++)
        {
            num_real = y - offset_y;
            num_real = num_real / zoom;
            num_img = x - offset_x;
            num_img = num_img / zoom;
            i=0;
            radius = 0;
        while ((i<ITERATION-1) && (radius < 4))
            {
                tmp1 = num_real * num_real;
                tmp2 = num_img * num_img;
                num_img = 2*num_real*num_img + IMG_CONSTANT;
                num_real = tmp1 - tmp2 + REAL_CONSTANT;
                radius = tmp1 + tmp2;
                i++;
            }
            /* Store the value in the buffer */
            buffer[x+y*size_x] = i;
        }
   }
}
```

4.4 结果

表 *10*显示了基于Cortex®-M4的STM32F4计算Julia集所花费的时间，针对几种缩放因子，如演示固件所示。

表**10.** 基于硬件单精度**FPU**与软件运算的**CORTEX® -M4**性能比较
利用**MDK-ARM™**工具链**V5.17**的**FPU**

帧	缩放	**HW FPU**持续时间**[ms]**	**SW**实现**FPU**持续时间**[ms]**	比值
0	120	195	2426	12,44
1	110	170	2097	12,34
2	100	146	1782	12,21
3	150	262	3323	12,68
4	200	275	3494	12,71
5	275	261	3307	12,67
6	350	250	3165	12,66
7	450	254	3221	12,68
8	600	240	3038	12,66
9	800	235	2965	12,62
10	1000	230	2896	12,59
11	1200	224	2824	12,61
12	1500	213	2672	12,54
13	2000	184	2293	12,46
14	1500	213	2672	12,54
15	1200	224	2824	12,61
16	1000	230	2896	12,59
17	800	235	2965	12,62
18	600	240	3038	12,66
19	450	254	3221	12,68
20	350	250	3165	12,66
21	275	261	3307	12,67
22	200	275	3494	12,71
23	150	262	3323	12,68
24	100	146	1781	12,20
25	110	170	2097	12,34

表 *11*显示了基于Cortex®-M7的STM32F7计算Julia集所花费的时间，采用与基于Cortex®-M4的STM32F4运行的相同算法，针对几种缩放因子，如演示固件所示。

表**11.** 基于硬件单精度**FPU**与软件运算的**CORTEX® -M7**性能比较
利用**MDK-ARM™**工具链**V5.17**的**FPU**

帧	缩放	**HW FPU**持续时间**[ms]**	**SW**实现**FPU**持续时间**[ms]**	比值
0	120	134	1759	13,13
1	110	118	1519	12,87
2	100	102	1291	12,66
3	150	179	2407	13,45
4	200	187	2529	13,52
5	275	178	2396	13,46
6	350	171	2294	13,42
7	450	174	2335	13,42
8	600	165	2204	13,36
9	800	161	2150	13,35
10	1000	157	2101	13,38
11	1200	154	2048	13,30
12	1500	146	1936	13,26
13	2000	127	1661	13,08
14	1500	146	1936	13,26
15	1200	154	2048	13,30
16	1000	157	2101	13,38
17	800	161	2150	13,35
18	600	165	2204	13,36
19	450	174	2335	13,42
20	350	171	2294	13,42
21	275	178	2396	13,46
22	200	187	2529	13,52
23	150	179	2407	13,45
24	100	102	1291	12,66
25	110	118	1519	12.87

表 *12*显示了基于Cortex®-M7的STM32F7采用上述算法计算Julia集所花费的时间，针对几种缩放因子，如演示固件所示。

表**12.** 基于硬件双精度**FPU**与软件运算的**CORTEX® -M7**性能比较
利用**MDK-ARM™**工具链**V5.17**

帧	缩放	**HW DP FPU**持续时间**[ms]**	**SW**实现**FPU**持续时间**[ms]**	比值
0	120	408	2920	7,16
1	110	355	2523	7,11
2	100	305	2145	7,03
3	150	550	3995	7,26
4	200	577	4197	7,27
5	275	547	3971	7,26
6	350	524	3799	7,25
7	450	533	3866	7,25
8	600	504	3643	7,23
9	800	492	3557	7,23
10	1000	481	3476	7,23
11	1200	470	3390	7,21
12	1500	446	3206	7,19
13	2000	386	2752	7,13
14	1500	446	3206	7,19
15	1200	470	3390	7,21
16	1000	481	3476	7,23
17	800	492	3557	7,23
18	600	504	3643	7,23
19	450	533	3866	7,25
20	350	524	3799	7,25
21	275	547	3971	7,26
22	200	577	4197	7,27
23	150	550	3995	7,26
24	100	305	2145	7,03
25	110	355	2523	7,11

我们可以观察到，使用硬件SP FPU与使用FPU软件实现之间的比值，优于使用硬件DP FPU与FPU软件实现之间的比值，只有当用户需要得到更高精度时才应使用“double”。但是，如果用户想获得更好性能的同时并希望减少RAM的开销，则应使用“float”。

4.5 Mandelbrot集

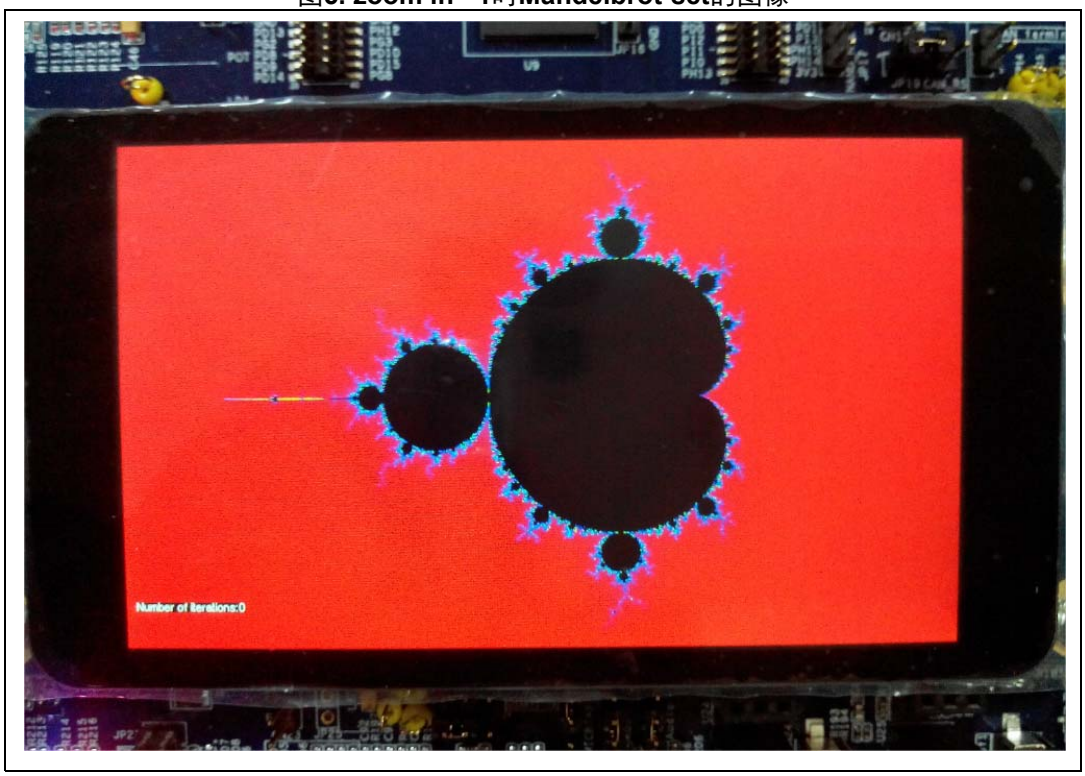
要生成Mandelbrot集，我们使用与Julia Set相同的迭代函数，如果像素和z从位置（x=0，y=0）开始，则c变量表示位置。

```
void drawMandelbrot_Double(float centre_X, float centre_Y, float Zoom, uint16_t IterationMax)
{
 double X_Min = centre_X - 1.0/Zoom;
 double X_Max = centre_X + 1.0/Zoom;
 double Y_Min = centre_Y - (YSIZE_PHYS-CONTROL_SIZE_Y) / (XSIZE_PHYS * Zoom);
 double Y_Max = centre_Y + (YSIZE_PHYS-CONTROL_SIZE_Y)  / (XSIZE_PHYS *Zoom);

 double dx = (X_Max - X_Min) / XSIZE_PHYS;
 double dy = (Y_Max - Y_Min) / (YSIZE_PHYS-CONTROL_SIZE_Y) ;

 double y = Y_Min;

 double c;

  for (uint16_t j = 0; j < (YSIZE_PHYS-CONTROL_SIZE_Y); j++)
{
    double x = X_Min;

    for (uint16_t i = 0;  i < XSIZE_PHYS; i++)
    {
      double Zx = x;
      double Zy = y;
      int n = 0;
      while (n < IterationMax)
      {
        double Zx2 = Zx * Zx;
        double Zy2 = Zy * Zy;
        double Zxy = 2.0 * Zx * Zy;

        Zx = Zx2 - Zy2 + x;
        Zy = Zxy + y;

        if(Zx2 + Zy2 > 16.0)
        {
          break;
        }
        n++;
      }
      x += dx;
    }
    y += dy;
```

```
}
}
```

图 5显示zoom=1时所生成的图像。

图5. zoom in =1时Mandelbrot-set的图像

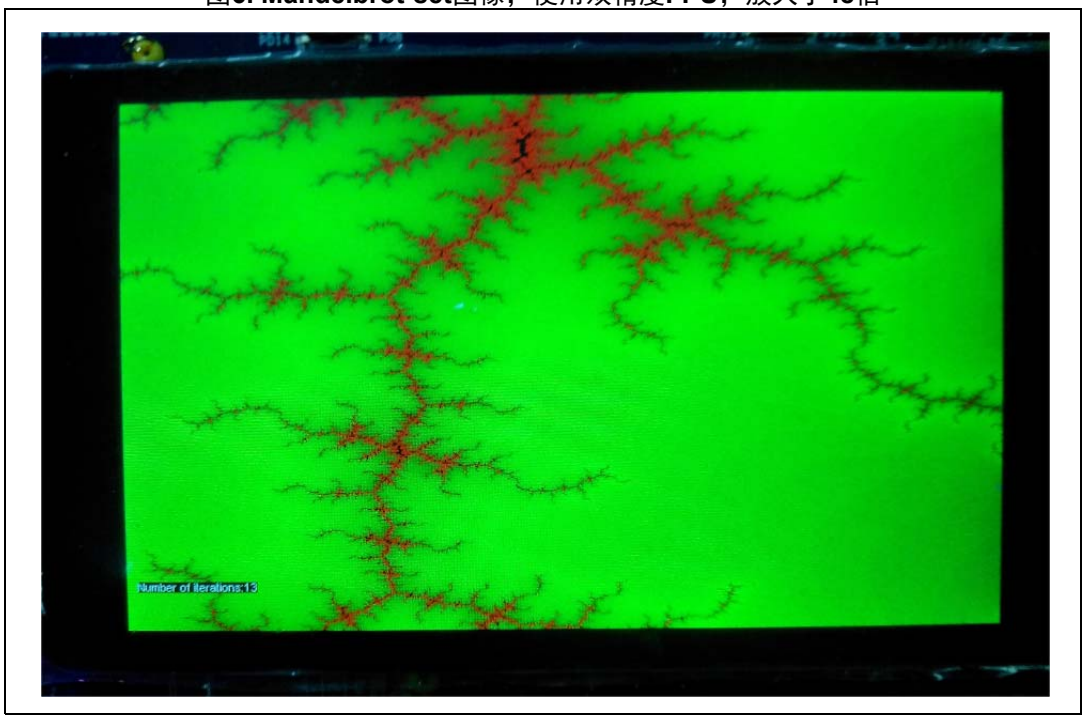


每次用户触摸屏幕时，该图片会被放大四倍。

图 6显示了放大的图片，达到了64位浮点数的数字限制。

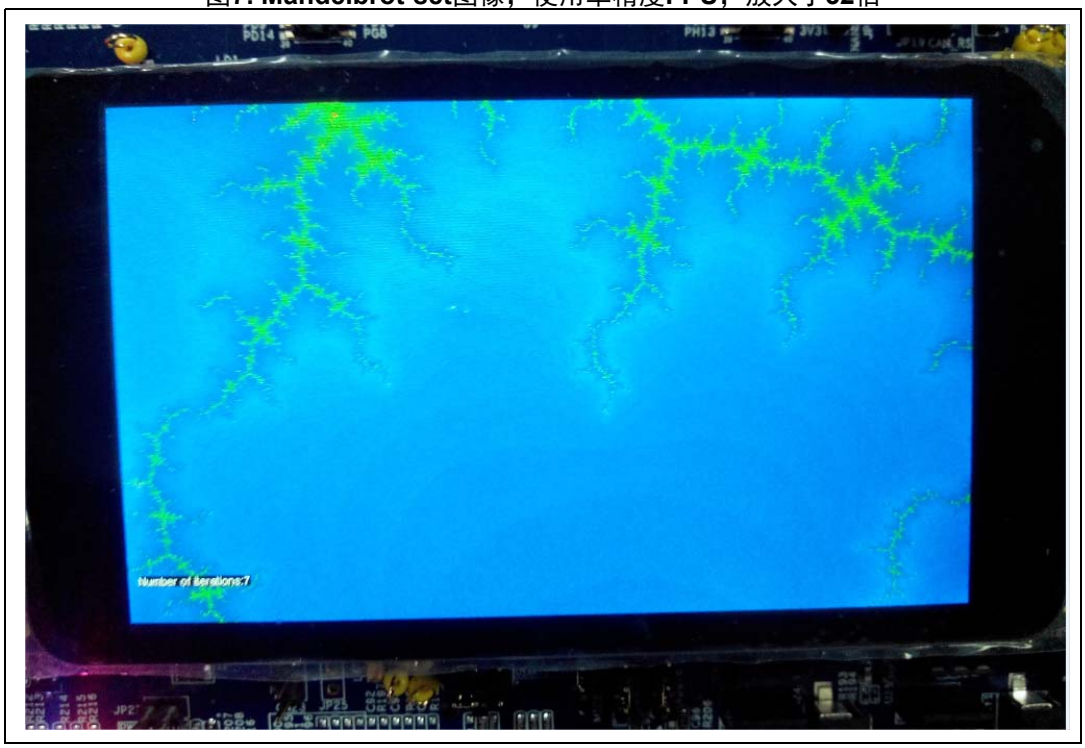
它开始呈现出块状。不过它放大了超过48倍。

图6. Mandelbrot-set图像，使用双精度FPU，放大了48倍



使用了与单精度FPU相同的算法。图 7显示，图片在放大32倍之后开始呈现出块状。

图7. Mandelbrot-set图像，使用单精度FPU，放大了32倍



使用硬件双精度，FPU不仅可以实现更短的计算时间，像我们在运行Julia Set时已经看到的那样，而且可以使我们得到更高的精度。

4.6 结论

使用“float”时，硬件FPU使Julia Set算法快了12.5倍，使用“double”时则快了7.2倍。无需修改代码，FPU在编译器选项中激活。

它可实现更宽的精度范围。

STM32 FPU可实现非常快速的浮点和双精度数学运算。

对于许多需要浮点算术处理的应用，如循环控制、音频处理或音频解码或数字滤波，FPU是一项重要优势。

它使开发过程从高级设计工具到软件生成都变得更快速更安全。

5 参考文档

表13. 参考文档

标题	作者	编辑
第一台电脑 历史与架构	Raul Rojas Ulf Hashagen	MIT出版社
IEEE754-2008 标准	IEEE	IEEE
ARMv-7M架构参考手册	ARM	ARM
ARM® Cortex® -M7处理器	ARM	ARM
RM0385参考手册	意法半导体	意法半导体
RM0090参考手册	意法半导体	意法半导体

6 版本历史

表14. 文档版本历史

日期	版本	变更
2012 年 3 月 16 日	1	初始版本。
2016年5月30日	2	更新了： – 前言 – 第 *3*节：*STM32 Cortex®-M*浮点单元（*FPU*） – 第 *4.6*节：结论 – 表 *10*：基于硬件单精度*FPU*与软件运算的*CORTEX®M4*性能比较 利用*MDK-ARM™*工具链*V5.17*的*FPU* – 表 *13*：参考文档 增加了： – 表 *6*：*STM32 Cortex®-M4/-M7* 中的*FPU*实现 – 表 *7*：*FPSCR*寄存器 – 表 *8*：一些浮点单精度数据处理指令 – 表 *9*：一些浮点双精度数据处理指令 – 表 *11*：基于硬件单精度*FPU*与软件运算的*CORTEX®M7*性能比较 利用*MDK-ARM™*工具链*V5.17*的*FPU* – 图 *4*：利用*MDK-ARM™*工具链*V5.17*配置*FPU* – 图 *5*：*zoom in =1*时*Mandelbrot-set*的图像 – 图 *6*：*Mandelbrot-set*图像，使用双精度*FPU*，放大了*48*倍 – 图 *7*：*Mandelbrot-set*图像，使用单精度*FPU*，放大了*32*倍 删除旧表*1*：适用产品和工具。

表15. 中文文档版本历史

日期	版本	变更
2017 年9月25日	1	中文初始版本。