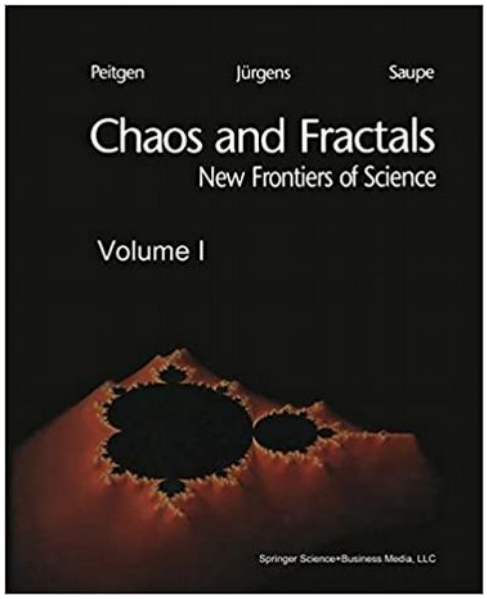
The never ending hunt for the fastest Mandelbrot

Or how to write efficient code on the 68080 and the SAGA chipset

Introduction



Fractals like the Mandelbrot are fascinating and beautiful. It was at the beginning of the 90s when I first heard about these mathematical objects and the formulas that stand behind them. Having worked a whole summer in a super market to earn enough money to buy a (then) very expensive 80486 DX2 50 Mhz (which offered an 80bit FPU and around 40 MIPS of computing power) I started exploring this mysterious world. During this "quest into the unknown" the book *Chaos and Fractals – New Frontiers of Science* by Peitgen/Jürgens/Saupe (see photo) was particularly helpful. The theory behind an uncountable number of fractals was very well explained and for many of them there were simple programs in BASIC available that could easily be adapted for ones own purposes.

Personally, my programming experience started with Turbo Pascal. It was a nice language and quite fast. Unfortunately, the existing graphics library (under MS-DOS) only offered 16 colors. But I had a class mate who had done some assembly programming on an 80286/80386 and who then showed me how to directly access my VGA card using interrupts. It was nice because suddenly I was able to use resolutions up to 1024x768 pixels in 16 Million colors. This was my first contact with assembly programming – a language that had always been rumored to be very difficult and inaccessible (so "not for me" I thought). Turbo Pascal offered a very nice way to write "inline assembly" (which made these first steps into assembly particularly easy and satisfying) and shortly after that I found myself reading many "Hardware Reference Manuals" in order to dig deeper.

In other words: I had been infected by the virus …

This virus consisted in optimizing the computationally intensive fractal calculations to make them as fast as possible. The 80486 was a quite fast CPU for the time – but depending on how you wrote the code and what algorithms you used, the calculations could take minutes or just seconds. The fascinating part was that sometimes you thought that you had really reached the final barrier and that no further optimizations were possible beyond that horizon. And then, suddenly, reading some manual or documentation, you stumbled upon something new – and a new idea about how to overcome the hurdle started growing.

I haven't done any assembly programming since the 90s. CPUs have become tremendously fast nowadays – multicore processors clocked at several Gigahertz. Theoretically, you can use a very inefficient "brute force" algorithm in combination with a very inefficient (even interpreted) programming language and do a Mandelbrot calculation in seconds or even fractions of seconds. What's the problem with that? Well, in my opinion there are two: first of all, you are wasting ressources; and second, you don't know what your compiler and computer is really doing. I like to compare that to the red and blue pill that Morpheus offers to Neo in the famous movie of the "Matrix".

Red stands stands for the real adventure, the “bare metal programming”, the truth – whereas blue stands for a life in the compiler, in the abstraction ...
Of course, it is almost impossible nowadays to write an entire operating system or even a kernel in assembly language (like it was the case for exec on AmigaOS). But assembly language still makes sense today and can be used in some very specific and time critical points to tune your code. In these cases, only assembly gives you the full control to do such optimizations (because a high level language hides all that from you and doesn’t give you access to these features).
Compared to my old 80486 DX2 50Mhz with 40 MIPS, the new Apollo 68080 with 85 Mhz and roughly 160 MIPS is around 4 times faster. It offers a nice FPU and a superscalar, pipelined instruction architecture which offers many possibilities for optimizations. So why not chose – again – the red pill and continue the adventure from the 90s? On the following pages I’d like to show some of these optimization techniques using the Mandelbrot as a concrete example.
I hope you enjoy this journey – have fun! Amiga rulez!

Languages & Compilers

All the examples presented her have been written and compiled with VASM (http://www.compilers.de/vasm.html) and SAS/C (http://www.pjhutchison.org/tutorial/sas_c.html). The reason for this is simple: VASM is, at least for the moment, more or less the only assembler that supports most of the new 68080 instructions and registers. As to SAS/C it is (still) one of the best C-compilers on Amiga. SAS/C was also the compiler used for AmigaOS itself (which guarantees a maximum of compatibility). Of course, all examples should also work with other compilers (have a look at this page to see other options like VBCC, StormC and GCC). There is also a modern cross compiler (https://github.com/bebbo/amiga-gcc) that you can use under Linux. Personally, I prefer to do all my work directly on my V4SA using a simple text editor like Annotate (https://aminet.net/search?query=annotate) that offers syntax highlighting and compiling my programs from the command line (shell).
As to debugging, there are, in my opinion, two programs that can be very helpful. First of all, there is CodeProbe (cpr) which is included with SAS/C. Even if CodeProbe has been developed at the time when no 68080 existed, it allows you to actually follow even very modern code thanks to the linedebug / symbol flush options of SAS/C and VASM (of course, you cannot see the contents of modern registers, but these limitations can be overcome by using classic registers for those values you need during the debugging). Second, there is a new update for MonAm which was/is the debugger for Devpac. The new version 3.09 of MonAm allows you to see and modify all the new registers and instructions for the 68080. Again, this is, for the moment, the only debugger which offers this.
Many of these tools are included in the two main distributions that are available for the Vampires, i.e. ApolloOS (http://www.apollo-computer.com/downloads.php), a fully free distribution based on AROS, Coffin (https://getcoffin.net/), based on AmigaOS 3.9 and specifically configured for the Vampires. A third option is CaffeineOS, another distribution based on AROS which uses DOPUS as a Workbench replacement (https://drive.google.com/drive/folders/1TsafNzgYVh45JIzW7alxhuIChPBacRiD).

The SAGA chipset [example0/simple_saga0.c]

Before we can start with our Mandelbrot, we have to do some preparatory work. First of all we need some graphics routines to be able to draw our calculations on the screen. We can do that by using the SAGA chipset. SAGA is a continuation of the classic Amiga chipset: There was OCS (Amiga 500 / 1000 / 2000), then ECS (Amiga 500+ / 600 / 3000) and finally AGA (Amiga 1200 / 4000). SAGA is the abbreviation for Super-AGA, so it is the continuation of AGA. In comparison to the classic chipset (with its “planar screen modes” where colors are encoded bit by bit and in separate bitplanes) SAGA

offers so called “chunky screen modes” (where every pixel is represented by a BYTE, a WORD or a LONGWORD), thus a much simpler way to draw things on the screen.
In this first example – and to keep things simple – we are going to do everything in C, so let’s start defining the custom registers for SAGA that we will need for our graphics routines [source code: example0/simple_saga0.c]:

```
UWORD* SAGAMODEW=(UWORD*)0xdff1f4;      /* SAGA mode and pixel format - write */
UWORD* SAGAMODER=(UWORD*)0xdfe1f4;      /*  "    "  "   "     "   - read  */
ULONG* SAGAPTRW=(ULONG*)0xdff1ec;       /* SAGA screen buffer - write */
ULONG* SAGAPTRR=(ULONG*)0xdfe1ec;       /*  "    "     "    - read  */
```

As you can see, the SAGA registers, like on classic Amiga, are divided into read (starting with $df**e**…) and write registers (starting with $df**f**…). The SAGAMODE register is a WORD (that’s why we define a word pointer UWORD*), SAGAPTR is a 32-bit pointer (so ULONG*). Note that we have to cast the hexadeximal values (0x…) we assign to these pointers because otherwhise our C-compiler will grant us with a warning.
We will also need some variables to store values:

```
UWORD oldmode;      /* save the actual graphics mode to restore it later */
ULONG oldbuffer;    /*  “   “   “     screen buffer   “     “        “ */
UWORD newmode;      /* new graphics mode that we will use for the Mandelbrot */
ULONG rawbuffer;    /*  “    screen buffer   “    “   “    “   “   “      “      */
ULONG newbuffer;    /* aligned screen buffer  */
ULONG memsize;      /* size of the new screen buffer because we need to allocate / deallocate memory for it */
```

Now let’s define our SetMode() and RestoreMode() functions. In our examples we will only use 8-bit (= 1 BYTE / pixel) screen modes:

```
UBYTE* Set8BitMode(UWORD mode, UWORD width, UWORD height) {
        /* save actual mode */
        oldmode=*SAGAMODER;
        oldbuffer=(ULONG)*SAGAPTRR;
        /* prepare new buffer */
        memsize=width*height+16;
        if ((rawbuffer=(ULONG)AllocMem(memsize, MEMF_ANY|MEMF_CLEAR))==NULL) return NULL;
        newbuffer=(rawbuffer+16) & 0xfffffff0;
        /* set new mode */
        *SAGAMODEW=newmode=mode;
        *SAGAPTRW=(ULONG)newbuffer;
        /* store resolution in global variables */
        resx=width;
        resy=height;
        return (BYTE*)newbuffer;
}

void RestoreMode(void) {
        /* restore old mode, pixel format and screen buffer */
        *SAGAMODEW=oldmode;
        *SAGAPTRW=(ULONG)oldbuffer;
        /* free screen buffer */
        if (rawbuffer) FreeMem((void*)rawbuffer, memsize);
}
```

Using C-pointers we very elegantly can read and write to the custom registers to store the actual mode (so that RestoreMode() can switch back at the end of our program), allocate some memory for the screen buffer and set our mode by writing to SAGAMODEW (mode and pixel format) and SAGAPTRW (screenbuffer). Note that our screenbuffer must be 64-bit (= 8 BYTES) aligned. Unfortunately, AllocMem() doesn't guarantee that, so that we simply reserve 16 more BYTES and then do a bitwise AND & with 0xfffffff0 (this guarantees that the lowest 4 bits of our address will be zeroed which means that our buffer is aligned to 128 bits – so more than necessary, but its clearer to write 0xfffffff0 than 0xfffffff8 which would zero the last 3bits to make a 64-bit alignment). After our alignment, when freeing the memory later, we just have to be careful to use the original *rawbuffer* pointer (otherwise, in a system with no Memory Management Unit (MMU) we will make the whole machine crash).
Now, what are the screen modes that we can set in SAGA? They are published on the Apollo website and are as follows (http://www.apollo-core.com/sagadoc/gfxmode.htm):

BIT#	FUNCTION	DESCRIPTION
15.08	Resolution	$00
		$01 = 320x200
		$02 = 320x240
		$03 = 320x256
		$04 = 640x400
		$05 = 640x480
		$06 = 640x512
		$07 = 960x540
		$08 = 480x270
		$09 = 304x224
		$0A = 1280x720
		$0B = 640x360
		$0C = 800x600
		$0D = 1024x768
		$0E = 720x576
		$0F = 848x480
		$10 = 640x200
07.00	Pixelformat	$00
		$01 = 8bit (indexed)
		$02 = 16bit R5G6B5
		$03 = 15bit 1R5G5B5
		$04 = 24bit R8G8B8
		$05 = 32bit A8R8G8B8
		$06 = YUV
		$07 =
		$08 = PLANAR 1BIT
		$09 = PLANAR 2BIT
		$0A = PLANAR 4BIT

So, if we want to set up a screen at 1280x720 resolution with 256 colors (8 Bit) we use *0x0a* (for the resolution) and *0x01* (for 8-bit CLUT).

```
SetMode(0x0a01, 1280, 720);
```

We also need a PutPixel() routine which we also write specifically for an 8-bit screen mode we are going to use:

```
void Put8bitPixel(UWORD x, UWORD y, UBYTE color) {
	*((UBYTE*)newbuffer+y*resx+x)=color;
}
```

This somewhat cryptic looking line simply means: We take the ULONG value newbuffer (that contains the address of our new SAGA screen buffer), cast it to a BYTE pointer with (UBYTE*) – because 8-bit means that every pixel uses 1 BYTE in memory – and finally do some nice C-pointer arithmetic: Our screen width is *resx*, so when we want to draw a Pixel on – say – line 11, we multiply *resx* by 11 and then add *x* to get the address of the pixel.
Finally, we also want to specifically set the rgb-values of the colors that we are going to use, so we define a SetColor() function. To make things easier we add again a port color register pointer SAGACOLORW:

```
ULONG* SAGACOLORW=(ULONG*)0xdff388;		/* chunky port color register */

void SetColor(ULONG color, ULONG red, ULONG green, ULONG blue) {
	ULONG value;
	value = (color<<24)|(red<<16)|(green<<8)|(blue);
	*(SAGACOLORW)=value;
}
```

As we see, we can use SAGACOLORW = $dff388 and write a long word to it that contains 4 BYTES:

1st BYTE (highest, bits 31-24) = *color* number we want to set (0-255)
2nd BYTE (bits 23-16) = *red* value (0-255)
3rd BYTE (bits 15-8) = *green* value (0-255)
4th BYTE (lowest, bits 7-0) = *blue* value (0-255)

All we have to do is to shift the values accordingly (24, 16, 8 and 0 bits to the left) and combine them into a LONGWORD using the binary OR | operator. For our example we are going to set color 0 to black rgb=(0,0,0) and the other colors to random values (using the rand() function from stdlib.h):

```
SetColor(0,0,0,0);
for (color=1; color<=255; color++) {
	SetColor(color, rand()%256, rand()%256, rand()%256);
}
```

So, now that we have defined all these graphical functions, we can write a simple loop that is going to show all 256 colors using one color per line:

```
for (y=0; y<resy; y++) {
	for (x=0; x<resx; x++) {
		Put8BitPixel(x,y,y%256);
	}
}
```

To compile this first program with SAS/C we simply open a shell, go to the drawer that contains our program (simple_saga.c) and then type:

sc link simple_saga0.c

And run it by typing:

simple_saga0

Well, that's all … or almost. Unfortunately we can't admire our marvelous work because after drawing the lines, the program immediately exits. So, let's add a function that waits for a mouse click. We can again use the custom register to check if the mouse button has been pressed [source code: example0/simple_saga1.c]:

```
UWORD* CIAAPRA=(UWORD*)0xbfe001;        /* Complex Interface Adapter => for mouse click */

void WaitMouse(void) {
        UWORD old_value = (*CIAAPRA);
        while (old_value==)(*CIAAPRA));
}
```

Then we just add this function before RestoreMode(), compile with *sc link simple_saga1.c*, run it with *simple_saga1* and now the SAGA screen stays active until we click the mouse. Note that – to be precise – it is the bit 6 of register $bfe001 that controls the "fire button" (joystick or mouse), but for simplicity we just check if this register changes (and loop as long as it doesn't).

SAGA in assembly [example1/saga_main0.c]

As we've seen, it is perfectly possible to open, close and write to SAGA screens in C without using any assembly. But we will for sure need some assembly programming later on so maybe it's a good moment to use this language right from the beginning to have a chance to get familiar with it and the way we have to compile our program in order to use both C and assembly at the same time. We will therefore take the exact same program we wrote for the SAGA chipset and write all graphics routines in assembly. If you are not familiar with assembly instructions (or assembly in general) I strongly recommend you to bookmark the website of Markey Jester (https://mrjester.hapisan.com/04_MC68/) and visit it whenever necessary. It is very complete and explains everything in more detail.
So, let's start again with the custom registers. In C we used pointers, in assembly we can just use EQU (= "is equal to") to define these "symbols". Instead of 0x for "hexadecimal value" we use the dollar $ sign [source code: example1/saga_asm0.s]:

```
SAGAMODEW       EQU     $dff1f4         ; SAGA mode and pixel format - write
SAGAMODER       EQU     $dfe1f4         ; "    "   "   "     "   - read
SAGAPTRW        EQU     $dff1ec         ; SAGA screen buffer - write
SAGAPTRR        EQU     $dfe1ec         ; "    "     "   - read
SAGACOLORW      EQU     $dff388         ; chunky port color register
```

You'll notice that the comments are also a bit different: We can either use a semicolon ; or an asterisk *. Now we define the variables for the SetMode() / RestoreMode() routines:

```
oldmode:        dc. w 0         ; save the actual graphics mode to restore it later
oldbuffer:      dc.l 0          ;   "    "    "      screen buffer    "      "         "
newmode:        dc.w 0          ; new graphics mode that we will use for SAGA screen
newbuffer:      dc.l 0          ;   "   screen buffer     "    "   "    "   "   "         "
```

We can use the DC (= define constant) command with the extensions .W (for a WORD value) or .L (LONGWORD). Note that there is no difference between “signed” or “unsigned” in assembly (it’s just “data” and we have to care ourselves if it has to be considered signed or unsigned). Every variable uses a label that must start on the first row in the edior (= on the left, position 0) and that can be terminated by a colon : or be omited on certain assemblers.
Now, let’s define our SetMode() routine: In assembly callable subroutines simply start with a label and end with the RTS (= return from subroutine) instruction:

```
_Set8BitMode:
        move.w      SAGAMODER,oldmode       ; save actual graphics mode in oldmode
        move.l      SAGAPTRR,oldbuffer      ; save actual graphics buffer in oldbuffer
        move.w      d0,SAGAMODEW            ; set new graphics mode and pixelformat
        move.l      #buffer,d0              ; copy buffer address to d0
        add.l       #16,d0                  ; align buffer
        and.l       #$fffffff0,d0           ;   "     "
        move.l      d0,SAGAPTRW             ; copy aligned buffer pointer to SAGAPTRW
        move.w      d1,_resx                ; set global C variable resx
        move.w      d2,_resy                ; set global C variable resy
        rts
```

So, we use several MOVE instruction to copy data from the source operand (on the left) to the destination operand (on the right). Lines 1-2 copy the actual mode and buffer to the variables *oldmode* / *oldbuffer*. The mode is a WORD value, so we use MOVE**.W**. The buffer is a 32-bit pointer, so we use MOVE**.L** for a LONGWORD. Lines 3-7 set the new mode and screenbuffer. We start by copying our new screen buffer to the register D0. To align it for SAGA we again add 16 to the LONGWORD register-value (the dash # is important, it means: we take the immediate value 16, whereas without # it means: read the memory at address 16), and then use AND.L #$fffffff0 (again: the dash # is important) to zero the lowest 4 bits of our buffer address. We then write the aligned buffer address to SAGAPTRW (and return it to the main program by leaving the value in D0 because D0 is, by default, the register that contains the return value). Lines 8-9 finally set the variables *resx* / *resy* (that we will define in the C-program, they can be accessed by our assembly part by just using an underscore before the name we use in C).
When you have a look at the assembly file *example1/fractal_asm0.s* you notice that it contains several “sections” (marked with the keyword SECTION). Sections are simply different parts of our assembly program: We have one section for the code (that we call “code”), one for initialized data (that we call “data1”) and one for uninitialized data (that we call “data2”):

```
        SECTION data2, BSS

buffer:         ds.b    1280*720+16             ; reserve 1280x720+16 = 921’616 bytes for screen buffer

        END
```

The keyword BSS (= **B**lock **S**tarting **S**ymbol) means that the memory with the name / label “buffer” and a size 1280*720+16 BYTES will be allocated dynamically at runtime (i.e. when the program is loaded). So, it replaces our original AllocMem() function call in the C-program (others call the BSS section the “**B**etter **S**ave **S**pace” section because it won’t be included in the final executable which therefore allows us to keep it small).
The RestoreMode() is also very simple in assembly:

```
_RestoreMode:
        move.w oldmode,SAGAMODEW            ; copy oldmode back to SAGAMODEW
        move.l oldbuffer,SAGAPTRW           ; copy oldbuffer back to SAGAPTRW
        rts
```

That's all! Two MOVE instructions between a label (= name of the subroutine) and the RTS (= return from subroutine). As to the screenbuffer, we don't have to care about it (because being in a BSS section it will automatically be freed by the operating system when the program terminates).
Now, how can we make sure that our main program (written in C) can call the assembly routines that we just defined? And how can we be sure that the assembly programs can use C-variables we define in the main program? To make this work, we have to declare our functions (also called "symbols") in both the assembly- and the C-source code. In C we have to declare what is called the "prototypes" for the assembly functions:

```
UBYTE* __asm Set8BitMode(register __d0 UWORD mode,
        register __d1 UWORD resx,
        register __d2 UWORD resy);
void __asm RestoreMode(void);
```

In assembly we just have to define the function / symbol with the XDEF (= cross definition) keyword:

```
        XDEF _Set8BitMode
        XDEF _RestoreMode
```

The prototype in C tells the compiler (1) that it is an ASM-function (keyword *__asm* with two underscores) and (2) where to put the parameters to transfer them to the assembly subroutine (in this case, we tell it to put the variable *mode* in register D0, *resx* in D1 and *resy* in D2) and what size they are (in this case UWORD for Set8BitMode and void = no parameter for RestoreMode). Note that we have to use an underscore before the symbol in assembly, so the functions (labels) are called _Set8BitMode and _Restore8-bitMode and we have to write *_resx* and *_resy* in order to access the global C-variables defined in the main program from our assembly code. As to the return value – in this case, the address of the screen buffer – we can just put it inside D0 because this is the default register to return an integer value to the main program.
Now, we are ready to compile our program for the first time. At the difference of the *example0/simple_saga1.c* (where we only had C-code, i.e. 1 file, to compile) this time we must compile and link the program in three steps: (1) compile assembly code (using VASM), (2) compile the C-file (using SAS/C) and (3) link the program using SLINK which is included in SAS/C).

(1) *vasm -m68080 -Fhunk saga_asm0.s -o saga_asm0.o*
(2+3) *sc link saga_main0.c saga_asm.o*

So, step (1) creates the *saga_asm0.o* object file (using the option *-o* which means "output"). Steps (2+3) can be combined: *sc* compiles the program *saga_main0.c* and the option *link* directly links the object code *saga_asm.o* to the program to build the final executable.
Since it would be a bit cumbersone to write these commands in the shell everytime we want to compile our program, we can create a "makefile" with the name *smakefile* [source code: example1/smakefile]:

saga_main0:

```
vasm -m68080 -Fhunk saga_asm0.s -o saga_asm0.o
sc link saga_main0.c saga_asm0.o
```

After that we can simply type:

smake

or

smake saga_main0

to automatically compile and link the 2 files. The second variant is useful when we have different versions of the same (slightly modified) program in the same drawer and want to compile a specific version (if we just type *smake* the automatic builder SMAKE included in SAS/C will simply build the first entry it finds in the *smakefile*).
As a standard in this tutorial there will always be a *smakefile* for all examples and you can always use the following *smake* commands to compile and build everything from source:

```
smake nameX         ; example: smake brute0 => compiles version 0 = brute_force0.c
smake all           ; builds all examples (versions) in the drawer
smake clean         ; cleans all objects-, link- and info-files of the examples
smake cleanexe      ; deletes all exe files of the examples
smake allclean      ; builds all executables and deletes obts-, link- and info-files
```

Now that we have successfully built a first version of our program, we will create a second version named *example1/saga_main1.c* and *example1/saga_asm1.s* in which we will transform the rest of our functions to assembly. We start with the SetColor() function. Let's start defining the prototype in C:

```
void __asm SetColor(register __d0 ULONG color,
        register __d1 ULONG red,
        register __d2 ULONG green,
        register __d3 ULONG blue);
```

In the assembly file we need to add the XDEF declaration plus the assembly code for _SetColor (again same name as in C but with an underscore).

```
        XDEF _SetColor

_SetColor:
        and.l   #$ff,d0         ; clear bits 31-8 of d0
        and.l   #$ff,d1         ; idem for d1
        and.l   #$ff,d2         ; idem for d2
        and.l   #$ff,d3         ; idem for d3
        lsl.l   #8,d0           ; color<<8
        add.l   d1,d0           ; d0=color(bits 15-8)+red(bits 7-0)
        swap    d0              ; d1=color(bits 31-24)+red(bits 23-16)
        lsl.l   #8,d2           ; green<<8
        add.l   d2,d0           ; d0=color(bits 31-24)+red(bits 23-16)+green(15-8)
```

```
        add.l   d3,d0                   ; d0=color(31-24)+red(23-16)+green(15-8)+blue(7-0)
        move.l  d0,SAGACOLORW           ; write the complete LONGWORD to SAGA color register
        rts
```

Now, what does this code do? The first 4 lines simply clear the upper 3 of the 4 LONGWORD-BYTES (this is just to be sure that will only be shifting the lowest BYTE and that the rest is 0). After that we do some left-shifting on the LONGWORD register (LSL.L) like in the C-code. The only difference is that we don't shift 24 bits (like in C), but only 8, and then SWAP (= exchange) the upper 2 BYTES (= WORD = bits 31-16) of the register D0 with the lower 2 BYTES (= WORD = bits 15-0). And like in C we "or" everything together (using OR.L) and write the final LONGWORD to the SAGA register SAGACOLORW.
We also replace our Put8BitPixel() routine defining again a prototype in the C-file:

```
void __asm Put8BitPixel(register __d0 ULONG x,
        register __d1 ULONG y,
        register __d2 ULONG color);
```

In the assembly file we add the XDEF definition and write the corresponding assembly code:

```
_Put8BitPixel:
        movem.l         a0,-(sp)        ; save address register a0 on stack
        move.l          SAGAPTRR,a0     ; copy saga screen buffer base address to a0
        mulu.w          _resx,d1        ; multiply d1 (=y) with global C variable resx (word)
        add.l           d0,d1           ; add d0 (=x) to d1 (=y*resx+x)
        move.b          d2,(a0,d1.l)    ; write byte (pixel) in d2 (= color) to address a0+d1 = buffer+y*resx+x
        movem.l         (sp)+,a0        ; restore address register a0 from stack
        rts
```

In this routine we have to calculate the address (inside the buffer) where we want to write to. That's why we first save the 32-bit address register A0 on the stack (using MOVEM.L or "move multiple registers" - in this case, it's only one register, namely A0). After that (in the second line) we copy our SAGA screen buffer pointer to A0 (copying it directly from the custom register). Our parameters – as declared in the C-prototype – are in D0 (= *x*), D1 (= *y*) and D2 (= *color*). So, we have to multiply register D1 (= *y*) with the width of the screen (= *resx*). We then add register D0 (= *x*) to it. In line 5 we use indirect addressing as a destination to write the BYTE in register D2 (= *color*) to the screen: (A0,D1.L) means "take the address in A0 as the base and add the LONGWORD-value D1 to it". Finally, we restore the content of A0 (from the stack) and return to the calling program via RTS.
To make things complete, we also add our WaitMouse() function by declaring the prototype:

```
void __asm WaitMouse(void);
```

Then we add our declarations and code for the subroutine in the assembly file:

```
CIAAPRA         EQU     $bfe001         ; Complex Interface Adapter => for mouse click

_WaitMouse:
        btst    #6,CIAAPRA              ; bit 6 in CIAAPRA = fire / mouse button (1 = pressed / 0 = not pressed)
        bne     _WaitMouse              ; while button is not pressed => go back to _WaitMouse (= check bit 6 again)
        rts
```

This time we explicitely check the bit 6 (= fire / mouse button) in the register and loop as long as it is not set (BNE = “branch if not equal” = branch while bit 6 is zero). Please note that if you want to wait for several mouse clicks that come shortly one after the other, this routine might cause problems (because the mouse button might still be pressed when it is checked a second time). That’s why I add a second routine that combines two loops to wait until the mouse button has been released again (this way we can be sure that mouse button isn’t pressed any more when a second check occurs shortly after):

```
_WaitMouseUp:
        btst      #6,CIAAPRA                ; wait for fire / mouse button pressed (like in _WaitMouse)
        bne.s   _WaitMouse
.mouseup:                                   ; local label
        btst    #6,CIAAPRA                  ; loop while fire / mouse button is pressed
        beq.s   .mouseup
        rts
```

We also discover two new specialities here: First of all, the label *.mouseup* is a “local label” (which means that you could use the label *.mouseup* several times in different subroutines). Second, we use BNE.S (“branch if not equal short”) and BEQ.S (“branch if equal short”) because the branching distance here is very short (< 128 BYTES) and using .S makes the compiler produce a shorter opcode (we will later see that such small details can also speed up the execution time, especially inside loops). This is very well explained here: https://mrjester.hapisan.com/04_MC68/Sect05Part03/Index.html
Now that we have “translated” all C-functions to assembly subroutines you can compile the program with *smake saga_main1* and run it with *saga_main1*.

Turning off multitasking, measuring time and mouse pointer [example2/saga_time.c]

I realize that we’ve already written many lines of code – and still no Mandelbrot … But these preparatory steps are necessary so that we can focus on the essential things later. So, please bear with me one more moment – we will soon reach the point where we can really start … ! :)
Our goal is to optimize code and to know if our program is really getting faster we need (1) a very precise method to mesure the time and (2) the possibility to turn off multitasking. The latter is necessary because – if we leave multitasking on – there might be other programs running at the same time and thus “stealing” ressources from the CPU so that our mesurements will never be exact.
Let’s start by how we can measure time. The 68080 offers a register named CCC (which stands for “Clock Cycle Count”) that constantly counts the “clocks” the CPU runs through when executing instructions. A clock is the shortest time during which a CPU can do something and it depends directly on the frequency of the processor (or “core” in the case of the 68080). Most of the 68080 cores are clocked at 12x the base frequency of the original PAL-Amiga which means 7.09379 Mhz (= 7’093’790 Hz) or a clock duration of 1 second / 7’093’790 = 0.00000014 seconds). Since the Vampire is clocked at 12x this speed (= 85.12548 Mhz) a clock cycle is 12 times shorter and thus corresponds to 0.000000011747364 seconds). The CCC-register is 32 bits long, which means that it can hold values up to 2^32 = 4’294’967’296. We can now take this value and divide it by 85’125’480 which gives us 50.4545442328196 seconds. So, the CCC-register is perfect to measure very precise execution times up to a maximum of roughly 50 seconds.
Let’s again start with our C-code and declare some basic functions that we call SetStart(), SetStop(), GetStart(), GetStop():

```
void __asm SetStart(void);
```

```
void __asm SetStop(void);
ULONG __asm GetStart(void);
ULONG __asm GetStop(void);
```

They are all very simple because they don't need any parameters. The GetStart(), GetStop() simply return an ULONG value (= that corresponds to the LONGWORD CCC-register values that we previously store with SetStart() and SetStop()).
In assembly [example2/saga_asm.s] the code looks as follows:

```
_SetStart:
        movec   CCC,d0                          ; CCC = Clock Cycle Count register
        move.l  d0,time_start
        rts

_SetStop:
        movec   CCC,d0
        move.l  d0,time_stop
        rts

_GetStart:
        move.l  time_start,d0
        rts

_GetStop:
        move.l  time_stop,d0
        rts
```

Again, very simple MOVE instructions. The only difference is that we have to use MOVEC to read the special register CCC. For the GetStart() / GetStop() routines we use again the default register D0 for the return value. Of course, we have to define the corresponding variables in the data1-section:

```
time_start:     dc.l 0
time_stop:      dc.l 0
```

Now that we can get the CCC-register value from C let's define a GetTime() function in C that returns the seconds between time_start and time_stop:

```
double GetTime(void) {
        ULONG start, stop, difference;
        ULONG frequency=12*7.09379*1000000;             /* 12x core */
        start=GetStart();
        stop=GetStop();
        difference=(stop>start) ? (stop-start) : (0xffffffff-(start-stop));
        return (double)difference/frequency;
}
```

Since the CCC-register simply "wraps around" when it reaches the maximum, we need to check if *stop* is higher than *start* (in this case it has wrapped around). We then calculated the difference between *start* and *stop* and divide by the *frequency* to get the seconds (as a floating point value). Then we place our SetStart() / SetStop()-function in the C-file wherever we want to start / stop the measuring and at the end of our program we print the result:

```
execime=GetTime();
printf("Execution time: %lf seconds\n", exectime);
```

Please note that – since we are going to turn multitasking off (and therefore also the operating system) during our calculation – we can only use this function in the end (after closing the SAGA screen and turning multitasking on again, so that the operating system can write to the shell).
So, how do we turn the multitasking off and on? Basically, we use again two custom registers (DMACON and INTENA) to read the actual values (= of the running operating system), then turn all DMA-channels and all interrupts off – and finally restore them at the end. These WORD registers have the particularity that bit 15 controls whether the other bits in these registers should be cleared or set. I won't go much into details at this point (because there are other, very good tutorials for that, for example: https://www.reaktor.com/blog/crash-course-to-amiga-assembly-programming/). For the moment, it is just important to know that after saving the original values we have to "or" them with $8000 to be able to write them back ($8000 corresponds to bit 15 set and or'ing them to the original value makes sure we can set the bits back when we restore the registers). In any case, it is not really necessary to understand 100% how this code works – you can just take it and use it … :)

```
_MultiTaskOff:
        movem.l d0,-(sp)            ; save d0 on stack
        move.w  $dff002,d0          ; save dmacon
        or.w    d0,dmacon           ; dmacon | $8000 ($8000 = bit 15 set to write value back later)
        move.w  #$7fff,$dff096      ; all dma off
        move.w  $dff01c,d0          ; save intena
        or.w    d0,intena           ; intena | $8000 (again: set bit 15 to write value back)
        move.w  #$7fff,$dff09a      ; all intena off
        movem.l (sp)+,d0            ; restore d0 from stack
        rts

_MultiTaskOn:
        move.w  #$7fff,$dff096
        move.w  dmacon,$dff096      ; restore old dmacon
        move.w  intena,$dff09a      ; restore old intena
        rts
```

Of course, we have to declare again our prototypes in the C-file, declare these symbols (labels) with XDEF in the assembly file [source code: *example2/saga_time.c*]. We also need the variables *dmacon* and *intena* that will contain the values that we can write back:

```
dmacon:     dc.w $8000
intena:     dc.w $8000
```

As mentioned, the default value for these two is $8000 so that we can do

```
        or.w    d0,dmacon
        or.w    d0,intena
```

Before writing them back in MultiTaskOn():

```
        move.w  dmacon,$dff096
        move.w  intena,$dff09a
```

Last thing: When we turn off multitasking our mouse pointer will still be on the screen (and look like "frozen") . We definitely want to turn it off so that we can admire our Mandelbrot even better … :)

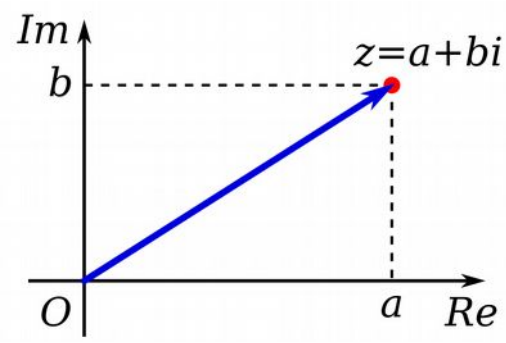
```
_MouseOff:
        move.w  $dfe1d0,mouse           ; save sprite 0 pointer to mouse
        clr.w   $dff1d0                 ; clear sprite 0 pointer to hide mouse (= "empty sprite")
        rts

_MouseOn:
        move.w  mouse,$dff1d0           ; restore old mouse pointer
        rts
```

Again: Please use this code without much explanation. It basically saves and then clears (= sets to 0 or a null pointer) the custom register SPRHSTRT (= pointer for sprite 0) and writes it back again in the end. Note that on AmigaOS, the mouse pointer is a sprite (and corresponds to sprite 0). This is also the case in SAGA screen modes (just that when using chunky screen modes the mouse pointer is the only classic sprite that will be blitted onto the screen).
One last thing: Since our program now uses floating point variables (for time measurement), we must add new compiler options:

sc link cpu=68040 math=68881 saga_time.c saga_asm.o

Specifying the 68881, which is one of the two classic FPUs that were available on Amiga, makes sure that SAS/C will generate floating point instructions for our C-program. As to the 68040 we just add it to get better code quality (because newer 68040 instructions will be used). In any case the 68080 is fully backwards compatible (so this option is not so important). As said earlier these options have also been included in the *smakefile*, so you can simply compile this example typing *smake saga_time* in the shell (and running it with *saga_time*).

The Mandelbrot formula



Right, a dozen pages to get started … I thought it would be shorter – but now we are definitely set to start talking about the Mandelbrot formula. There are, of course, many sites that describe this well known fractal (e.g. Wikipedia https://en.wikipedia.org/wiki/Mandelbrot_set). The important thing for us is that the so called Mandelbrot set is calculated in a plane with complex numbers, which means that the x-axis corresponds to the real part of the complex number and the y-axis to the imaginary part. Traditionally, a complex number is represented using the imaginary number *i*, which is defined as the *square root of -1*: *i = sqrt(-1)* or *i^2 = -1*. Complex numbers like *z=a+bi* are then calculated like any other arithmetical term, meaning: *(a+bi) * (a+bi) = a^2 + 2*bi – b^2*. In other words, the new number consists again of a real part of *a^2 – b^2* and an imaginary part of *2*b*.
The Mandelbrot formula is then defined as a recursive function with *zn+1=zn^2 + c*. Both *zn* and *c* are complex numbers (as is *zn+1*). The starting number is defined as *z0 = 0 + 0i*. The constant *c*

corresponds to the point on the complex plane that we want to calculate, so *c = cx + cyi*. We can now write the formula as:

z0 = 0 + 0i
c = cx + cyi

z1 = (0 + 0i)^2 + (cx + cyi) = cx + cyi
*z2 = (cx + cyi)^2 + (cx + cyi) = cx^2 + 2*cx*cyi – cyi^2 + cx + cyi*

or in general:

*zn+1 = (xn + yni)^2 + (cx + cyi) = xn^2 + 2*xn*yni – yn^2 + cx + cyi*
*zn+1 = (x^2 – yn^2 + cx) + (2*xn*yn + cy) * i*

or with the real and imaginary part separated:

xn+1 = x^2 – yn^2 + cx
*yn+1 = 2*xn*yn + cy*

Complex numbers also have a distance function that defines their distance to the origin = 0 + 0i. According to Pythagoras a complex number *z = a + bi* has a distance *d* of:

d = sqrt(a^2 + b^2)

We can also avoid to calculate the square root (which is computationally intensive) and just say that the square distance *d* is equal to to squares of (the absolute values of) *a* and *b*:

d^2 = a^2 + b^2

Last thing we need to know is that when we calculate a series of points z0, z1 … zn using the recursive formula, there are two different type of points:

Type 1: These points grow bigger and bigger, which means that when you measure the distance from the origin the point gets further away with every iteration (and tends to infinity).

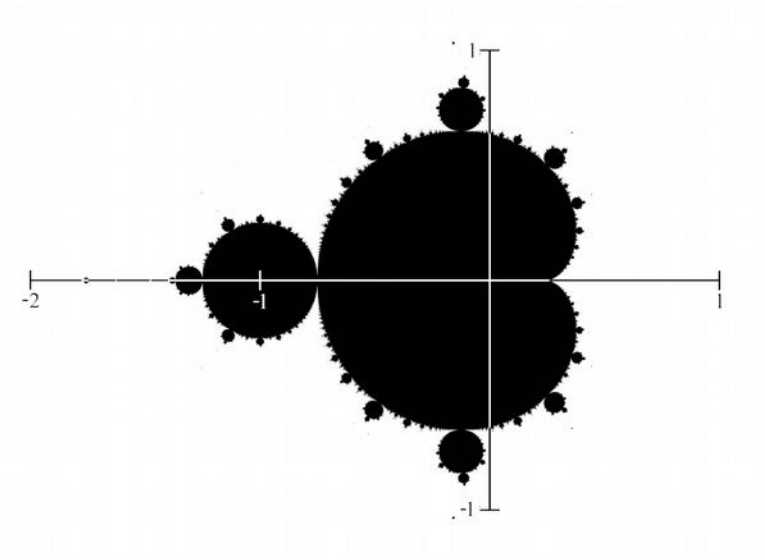
Type 2: These points either converge to a fix number or “cycle forever” (between various values) without escaping to infinity.

The points that converge (type 2) are considered to be part of the Mandelbrot set (and are generally colored black). The points that escape to infinity (type 1) are not part of the set (and are generally colored according to the number of iterations that they need to escape a previously defined radius around the origin).

The brute force algorithm [example3/brute_force0.c]

Now let's write our first program using the so called "brute force" algorithm. This very inefficient algorithm consists in calculating every point (pixel) in the complex plane of our screen and coloring it either black (type 2) or using the number of iterations it needed to escape a certain radius (type 1). Traditionally, this radius (distance) is set to *d = 2* or *d^2 = 4* because we can be sure that all points with this (or a greater) distance have no chance to converge but will for sure escape to infinity. Let's say we have a screen of 1280x720 pixels with 256 colors. Since the Mandelbrot set lies within certain boundaries, for example *-2* and *1* on the real axis and *-1* and *1* on the imaginary axis, we define the borders of our screen accordingly.

minr = -2.25
maxr = 1.25
mini = -1.25
maxi = 1.25



We can now calculate the complex numbers that correspond to the pixels:

stepr = (maxx-minx) / resx
stepi = (maxy-miny) / resy

With:

resx = 1280
resy = 720

Our complex number *c* (= constant for the calculation) for a given pixel (x,y) is then:

*cx = x * stepr + minx*
*cy = y * stepi + miny*

With all these elements given, we write now our brute force algorithm:

```
for (y=0; y<resy; y++) {
        for (x=0; x<resx; x++) {
                yn = cy = y*stepi+mini;
                xn = cx = x*stepr+minr;
                i=MaxIter;
                while ((i) && (xn*xn+yn*yn<=4.0)) {
                        xn1 = xn*xn - yn*yn + cx;
                        yn1 = 2*xn*yn + cy;
                        xn=xn1;
                        yn=yn1;
                        i--;
                }
                Put8BitPixel(x,y,i);
        }
}
```

Compile it with *smake brute0* and run it with *brute_force0*. Pretty slow (around 41.71 seconds on my V4SA core 7.4). A good point to start our optimizations.

Classic FPU-code [example3/brute_force1.c]

When we optimize code we always have to find out where (in which part) we have the highest chance to improve performance. In this case, it is – of course – the inner loop (= the part inside the two for-loops) of the programs because this part is executed a lot of times:

minimum:	1280*720*1 = 921’600 times (if point escapes after one iteration)
maximum:	1280*720*256 (MaxIter) = 235’929’600 times (= points are part of Mandelbrot)

So, what we will do is simply to replace our loop by classic FPU-code. Classic means that we will only use the classic floating point registers (FP0-FP7) and instructions that the 68881/68882 (and the integrated FPUs in the 68040 and 68060) can execute.
Our inner loop in C:

```
while ((i) && (xn1+yn1<=4)) {
        yn=2*xn*yn+cy;
        xn=xn1-yn1+cx;
        xn1=xn*xn;
        yn1=yn*yn;
        i--;
}
```

Will be replaced by:

```
i=IterateAsmClassic(cx,cy,MaxIter);
```

We declare the protoype:

```
UWORD __asm IterateAsmClassic(register __fp0 double x,
        register __fp1 double y,
        register __d0 UWORD maxit);
```

And write our assembly subroutine:

```
_IterateAsmClassic:
        ; fp0: cx
        ; fp1: cy
        fmovem  fp2-fp6,-(sp)
        fmove   #0,fp2                  ; xn=0
        fmove   #0,fp4                  ; xn1=0
        fmove   #0,fp3                  ; yn=0
        fmove   #0,fp5                  ; yn1=0
.loop:
        fmove   fp4,fp6                 ; xn1
        fadd    fp5,fp6                 ; xn1+yn1
        fcmp.s  #4,fp6
        fbgt    .exit                   ; xn1+yn1>4 => .exit
        fmul    fp2,fp3                 ; yn=xn*yn
        fadd    fp3,fp3                 ; yn=2*xn*yn
```

```
        fadd    fp1,fp3                 ; yn=2*xn*yn+cy
        fmove   fp4,fp2                 ; xn=xn1
        fsub    fp5,fp2                 ; xn=xn1-yn1
        fadd    fp0,fp2                 ; xn=xn1-yn1+cx
        fmove   fp2,fp4                 ; xn1=xn
        fmul    fp4,fp4                 ; xn1=xn*xn
        fmove   fp3,fp5                 ; yn1=yn
        fmul    fp5,fp5
        dbra.w  d0,.loop                ; d0>0 => .loop
        moveq.l #0,d0                   ; iteration = 0
.exit:
        fmovem  (sp)+,fp2-fp6
        rts                                     ; return iterations (D0)
```

Please note that this is not (yet) the most efficient code we can generate. It is an exact 1:1 translation of our C-code (so that we can understand how this all works). What we do here is:

- We save registers FP2-FP6 on the stack (so that we can use them as temporary registers for our calculation and restore them back at the end). As to the registers FP0 and FP1 we don't have to care about them (since we declared them as parameter-registers in our prototype, the compiler will save and restore them if necessary).
- We use registers FP2-FP4 for xn, yn, xn1 and yn1 and set them all to zero using FMOVE (which is the floating point equivalent to the integer MOVE instruction of the CPU).
- We create a local label *.loop* that replaces the C-while-loop. The most efficient way to do a loop in assembly is to use the dbra-instruction: DBRA.W D0,.loop means "subtract 1 to WORD-register D0 and if it is not 0 => branch to *.loop* (at the end – after maxit iterations – of the loop D0 will contain $ffff, we then set D0 to 0 with MOVEQ.L #0,d0 because this point is part of the Mandelbrot set).
- Inside *.loop* and DBRA we do our calculation. All these instructions use always two (source at the left, destination at the right) registers. We always use the full registers (instead of FMOVE, FADD, FSUB etc. we could also write FMOVE.X, FADD.X, FSUB.X etc. – but when we don't write it, the compiler will assume that it is an .X instruction). The only difference is the comparison FCMP.S #4,fp6 because #4 is an immediate value (and we can force the compiler to create somewhat shorter and thus faster opcodes by using the .S suffix which indicates that it is a short floating point variable).
- We break out of the loop if FP6 (= *xn1+yn1*) is greater than 4 by using FBGT *.exit* (= float branch if greater than"; the FBGT instruction uses the flags of the FPU previously set by the FCMP comparison).
- We finally restore the registers FP2-FP6 and return from the subroutine (RTS).

So, let's see how this code behaves by compiling it with *smake brute1* and running it with *brute_force1*. The new execution time is 30.31 seconds, so around 11.4 seconds (27.33%) faster than our first program. Not bad!

Vampire FPU as a 3-operant machine with new registers [example3/brute_force2.c]

At the difference of the classic FPUs the Vampire is a 3-operant machine, which means: instead of having only 1 source and 1 destination register (as we saw in all our code until now), the Vampire can have 2 source and 1 destination registers. This can typically be useful for FMOVE-instructions that are followed by an FSUB, FADD, FMUL, FDIV or other instructions:

Classic FPU:

```
        fmove   fp1,fp3         ; copy fp1 => fp3 (copy, because we still need fp1!)
        fmul    fp2,fp3         ; multiply fp2*fp3
```

Vampire:

```
        fmul    fp1,fp2,fp3     ; multiply fp1*fp2 and put the result in fp3
```

Speedwise this makes, of course, a difference because in the first case, the FMOVE and FMUL will be executed separately (= 2 clock cycles), whereas in the second case it only takes 1 clock cycle.
The Vampire also introduces new registers that can be used by the FPU. These registers are named E0-E23 (= 24 registers) and are declared as scratch registers. This means that we don't have to save them on the stack (so this saves us another clock cycle at the beginning and at the end). Let's integrate all that and see how these changes improve the performance of our iteration code snippet:

```
_IterateAsmVampire0:
        ; fp0: cx
        ; fp1: cy
        ; use e2-e6 as scratch registers
        ; use 3-operant instructions
        fmove   #0,e2                   ; xn=0
        fmove   #0,e4                   ; xn1=0
        fmove   #0,e3                   ; yn=0
        fmove   #0,e5                   ; yn1=0
.loop:
        ; fmove   fp4,fp6               ; xn1
        ; fadd    fp5,fp6               ; xn1+yn1
        fadd    e4,e5,e6                ; 3 operant fadd replaces fmove+fadd (= 2 instructions)
        fcmp.s  #4,e6
        fbgt    .exit                   ; xn1+yn1>4 => .exit
        fmul    e2,e3                   ; yn=xn*yn
        fadd    e3,e3                   ; yn=2*xn*yn
        fadd    fp1,e3                  ; yn=2*xn*yn+cy
        ; fmove   fp4,fp2               ; xn=xn1
        ; fsub    fp5,fp2               ; xn=xn1-yn1
        fsub    e5,e4,e2                ; 3 operant fsub replaces fmove+fsub (= 2 instructions)
        fadd    fp0,e2                  ; xn=xn1-yn1+cx
        ; fmove   fp2,fp4               ; xn1=xn
        ; fmul    fp4,fp4               ; xn1=xn*xn
        fmul    e2,e2,e4                ; 3 operant fmul replaces fmove+fadd (= 2 instructions)
        ; fmove   fp3,fp5               ; yn1=yn
        ; fmul    fp5,fp5
        fmul    e3,e3,e5
        dbra.w  d0,.loop                ; d0>0 => .loop
        moveq.l #0,d0                   ; iteration = 0

.exit:
   rts                                  ; return
```

To show the modifications we made, the classic instructions are commented out in the code. It is always two instructions (FMOVE + FMUL/FADD) and the 3-operant-instruction that replace them comes immediately after.
Let's compile this new IterateAsmVampire0() function with *smake brute2* and see how it performs. The new execution time is 28.61 seconds – not a huge performance win but still 31.41% faster than our first version.

Latency and instruction scheduling [example3/brute_force3.c]

Talking about the FMOVE and FMUL instruction above, I insinuated that these instructions execute in one cycle … Well, that's not exactly true (unfortunately). Thanks to its superscalar and pipelined architecture the Apollo 68080 can execute up to 4 classic instructions in 1 cycle (that's the good news), but (and here comes that bad news) it can only take advantage of this advanced architecture, if our code respects certain conditions. So, to come back to the question about execution times: The 68080 can indeed start (!) one FADD, FSUB, FMUL, FDIV etc. instruction every clock cycle!

```
fadd fp0,fp1
fadd fp2, fp3
fadd fp4, fp5
fadd fp6, fp7
etc.
```

Here, the 4th instruction FADD fp6,fp7 is indeed started after 3 (= on the 4th) clock cycle. The question now is: When do the instructions finish? This is called the "latency" which means the full execution time of an instruction (= when it has gone through all the superscalar and pipelined architecture and stages). There is no actual table for the latencies[1], but it seems that most of the instructions on actual cores execute in 6 clock cycles. Others – like fdiv – execute in 10 cycles (and more complex instructions – like fsin or similar – might need hundreds of cycles …).
So, what does this mean? Let's take again the instruction sequence from above and modify it slightly:

```
fadd fp0,fp1
fadd fp1,fp2
fadd fp2, fp3
fadd fp3, fp4
etc.
```

We have again 4 instructions that follow each other, but this time the 2nd instruction (FADD FP1,FP2) needs the result of the 1st instruction before it can start. As a consequence, the 2nd instruction has to **wait** until the 1st instruction finishes. Since we created a kind of a cascade it is the same for the following instructions. So, the 4th instruction (FADD FP3,FP4) can only start when all the 3 before have finished which means that we have a full "speed penalty" of 3 x 6 = 18 clock cycles of latency! Compared to the 3 clock cycles in the first examples this is a huge difference (factor 6).
Let's take the problem the other way around: Let's say our code must do these 4 additions as shown in the sequence above (with 18 clock cycles of latency). The good thing is: Knowing where our latencies are, we can now insert other instructions in between (= when these latencies occur anyway), that will get executed "for free":

1 For the interested, there are tables for older cores available on web.archive.org: https://web.archive.org/web/20191014165614/https://wiki.apollo-accelerators.com/doku.php/fpu

```
        fadd fp0,fp1
        fmul e1,e2      ; for free
        fmul e3,e4      ; for free
        fmul e5,e6      ; for free
        fmul e7,e8      ; for free
        fmul e9,e10     ; for free
        fadd fp1,fp2
        fmul e11,e12    ; for free
        fmul e13,e14    ; for free
        fmul e15,e16    ; for free
        fmul e17,e18    ; for free
        fmul e19,e20
        fadd fp2, fp3
        fmul e21,e22    ; for free
        fmul e1,e2      ; for free – fmul e1,e2 from above has finished now => we can reuse e1,e2
        fmul e3,e4      ; for free – idem
        fmul e5,e6      ; for free – idem
        fmul e7,e8      ; for free – idem
        fadd fp3, fp4
        etc.
```

I haven't acutally tested this code, but assuming that the latency of FADD is 6, these 3x5=15 instructions wil get executed for free (which means: the whole code still executes in 18 cycles + last instruction). In other words: in 18 cycles you can execute just 3 instructions (waisting the rest with latency) or 18 instructions (+ wait for the last instructions to finish).
Let's see how this applies in a very practical example to our iteration code:

```
_IterateAsmVampire1:
        ; fp0: cx
        ; fp1: cy
        ; use e2-e6 as scratch registers
        ; use 3-operant instructions
        ; use instruction scheduling
        fmove   #0,e2                   ; xn=0
        fmove   #0,e4                   ; xn1=0
        fmove   #0,e3                   ; yn=0
        fmove   #0,e5                   ; yn1=0
.loop:
        fmul    e2,e3           ; * yn=xn*yn
        fadd    e4,e5,e6                ; 3 operant fadd replaces fmove+fadd (= 2 instructions)
        fsub    e5,e4,e2        ; * 3 operant fsub replaces fmove+fsub
        fcmp.s  #4,e6
        fbgt    .exit                   ; xn1+yn1>4 => .exit
        ; fmul    e2,e3                 ; # yn=xn*yn  => move it upwards!
        fadd    e3,e3                   ; yn=2*xn*yn
        ; fadd    fp1,e3                ; # yn=2*xn*yn+cy => move it downwards!
        ; fsub    e5,e4,e2              ; # 3 operant fsub replaces fmove+fsub (= 2 instructions) => move it upwards!
        fadd    fp0,e2                  ; xn=xn1-yn1+cx
        fadd    fp1,e3          ; * yn=2*xn*yn+cy
        fmul    e2,e2,e4                ; 3 operant fmul replaces fmove+fadd (= 2 instructions)
        fmul    e3,e3,e5
        dbra.w  d0,.loop        ; d0>0 => .loop
        moveq.l #0,d0           ; iteration = 0
.exit:
        rts                                     ; return
```

Have a look at the 3 instructions that have been commented out. The first two (and the one in the middle that hasn't been commented out) use all the register E3 (three times in a row). This clearly creates a latency, so we move one upwards and one downwards (they are marked with * opposed to # which corresponds to the instructions that have been moved. We do the same with the FSUB-instruction because it creates a "register collision" with E2. Of course, we have to be careful with these displacements: For obvious reasons the instructions cannot go outside of the loop and we can only move it as long as the register is not needed or modified by another instruction!
Let's compile this code with *smake brute3* and see how it performs. The execution time has now come down to 21.18 seconds (49.22% faster). Please note that in the last step we only moved 3 lines of code and that this gave us a speed increase of 7.43 seconds (17.81%). That's really not bad for a copy&paste optimization! :)

Parallelizing the calculation [example4/parmandel1.c]

So, can we go even faster? Of course … ! :) When we look at our inner loop above, we notice that there is little room to move instructions around (because the loop is relatively short). Until now we have calculated all Mandelbrot points individually: First, point *(0,0)*, then *(1,0)*, then *(2,0)*, then *(3,0)* … up to *(1279,719)*. But we know that the calculations of these points are completely independent of each other (that's why, for example, you can use modern 32-core processors which, each, calculate a part of the final picture speeding up considerable the calculation time).
As we have seen, we have a relatively small loop that doesn't allow us to optimize our latencies as we would like to do. But calculating several points in the same loop will give us more room to move instructions around. So, let's start by calculating 2 points in "parallel". Please note that in the following example we will use an even more optimized iteration routine _SingleIterateAsm that we are going to parallelize [example4/parmandel1.c]:

```
_SingleIterateAsm:
        fmove   #0,e3               ; i
        fmove   #0,e2               ; r
.loop:
        ; (1)
        fmul    e2,e2,e4            ; r2=r*r
        fmul    e3,e3,e5            ; i2=i*i
        ; (2)
        fadd    e4,e5,e6            ; r2+i2
        fmul    e2,e3
        ; (3)
        fcmp.s  #4,e6               ; >=4?
        fbgt    .exit
        ; (4)
        fsub    e5,e4,e2
        fadd    e3,e3
        ; (5)
        fadd    fp0,e2              ; r=r2-i2+x
        fadd    fp1,e3              ; i=2*r*i+y

        dbra.w    d0,.loop
        clr.l   d0
.exit:
        rts
```

As you can see, we have eliminated the two of the four FMOVE #0,… instructions because they are simply not necessary (we just kept them from the original C-code). Now how do we “parallelize” this routine? Well, basically we (1) just copy code and (2) modify the register values. To make things more visible, we numerated the different parts (from 1-5):

```
_Par2IterateAsm:
        ; parameters
        ; fp0/fp1: x1, y1 (point 1)  -- regs: e2-e10     (=0)
        ; fp2/fp3: x2, y2 (point 2) -- regs: e12-e20 (+10)
        ; d0: MaxIter
        ; prepare

        move.w  #0,_IterP1          ; default value = 0
        move.w  #0,_IterP2
        clr.l   d0                  ; d0: mark points as "not done" (0)
        ; start
        fmove   #0,e3               ; i (x1)
        fmove   #0,e2               ; r (y1)
        ; ------
        fmove   #0,e13              ; y2
        fmove   #0,e12              ; x2

.loop:
        ; (1)
        fmul    e2,e2,e4    ; r2=r*r
        fmul    e3,e3,e5    ; i2=i*i
        ; ------
        fmul    e12,e12,e14
        fmul    e13,e13,e15
        ; (2)
        fadd    e4,e5,e6    ; r2+i2
        fmul    e2,e3
        ; ------
        fadd      e14,e15,e16
        fmul    e12,e13
        ; (3)

        ; check if points are done
        btst.l  #0,d0               ; p1 done?
        bne.s   .contp1             ; yes => continue with p2
        fcmp.s  #4,e6               ; p1>=4?
        fblt    .contp1             ; no => continue
        or.l    #1,d0               ; mark p1 as done
        move.w  d1,_IterP1          ; store iterations for p1
        ; ------
.contp1:
        btst.l  #1,d0               ; p2 done?
        bne.s   .contp2             ; yes => continue with calculation
        fcmp.s  #4,e16              ; p2>4?
        fblt    .contp2             ; no => continue
        or.l    #2,d0               ; mark p2 as done
        move.w  d1,_IterP2          ; store iterations for p2
.contp2:
        ; (4)
```

```
        fsub    e5,e4,e2
        fadd    e3,e3
        ; ------
        cmp.l   #3,d0
        beq     .exit
        ; ------
        fsub    e15,e14,e12
        fadd    e13,e13
        ; (5)
        fadd    fp0,e2              ; r=r2-i2+x
        fadd    fp1,e3              ; i=2*r*i+y
        ; ------
        fadd    fp2,e12
        fadd    fp3,e13
        ; ------
        dbra.w  d1,.loop
.exit:
        ; d0 = result
        rts
```

So, we basically just copy each part and place it immediately afterwards (step 1). Then we change the number of the E-registers in the copy (step 2). In this example we have just added 10 to each E-register number, so E2 becomes E12, E3 becomes E13 etc. (and, of course, at this point we are very happy to have these extra scratch registers that the Vampire offers:).
Now, this copied code, if we ran it like that, would be really fast (almost twice the speed). But unfortunately, we are now calculating two points in parallel and our code cannot run without additional modifications. The point is that we only have one main loop and in the original calculation this loop could be exited when the single point we calculated was escaping to infinity. With two points calculated in parallel, we can't do that: There will certainly be one point that escapes to infinity before the other one; or one point escapes to infinity and the other doesn't. Imagine two points: A escapes to infinity and B is inside the set (= doesn't escape). At a certain point, the distance of A becomes >4 and we exit the loop. This means that the calculation for B (which needs the maximum iterations) cannot finish. So, as a third step, we need to implement code that keeps track of whether a point has finished its calculation (and store the iterations it took to finish) and break out of the loop only when both points have finished.
In our example, we use the register D0 (bits 0 and 1 to be exact) to keep track of whether a point has finished or not. Clearing it at the beginning (with CLR.L D0) means bits 0 and 1 are zero (= calculation has not finished). Now, at each iteration, we check the distance for point 1 and 2. If point 1 finishes, we will set bit 0 in D0 to 1 (so that we know that it has finished) and write the actual iterations to *_IterP1* (MOVE.W D1,_IterP1). The same applies to point 2 (with bit 1 that we set to 1 and MOVE.W D1,_IterP2). In the C-file we define UWORD *IterP1*, *IterP2*; so that our assembly routine can access them (via an initial underscore). Now, we just have to make sure that our loop stops when both points have finished. We can do that with:

```
        cmp.l   #3,d0
        beq     .exit
```

Of course, we try to "schedule" our instructions to the maximum (and that's why we place these two lines of code somewhere between (4) and (5) so that they eventually get executed "for free").

You can try to play around yourself with this code and see, if you find a scheduling that is even faster. But taking the code as presented and compiling it with *smake par1* will lead to an execution time of 15.66 seconds. So, we are now 62.49% faster than the first version!

We can even go further and calculate 4 points in the same loop. We can simply use the registers E2/E7/E12/E17 and E3/E8/E13/E18 etc. and the bits 2 and 3 in D0 (plus a slighty adapted comparison). Our code (loop) is now getting even larger offering even more room for instruction scheduling. Let's see what influence this has on our performance! Compile it with *smake par2* and run it with *parmandel2*. The execution time is now 13.42 seconds or 67.75% faster.

The algorithm … it's always the algorithm!

At this point we could, of course, continue to optimize our calculation by concentrating even more on the code and technical details of the CPU/FPU (for example, we could translate the main program into assembly and this would certainly lead to a further speed improvement; we could also try to use some AMMX optimizations that the Vampire offers). But, as we said earlier, we have always to analyze very carefully where we have the most potential for an efficient optimization. So, time has come to talk about our algorithm …

Unfortunately, programmers often tend to look at things in a very technical manner without considering that the problem could be elsewhere. We know that there are algorithms that are simple to implement but slow (bubble sort). And then there are algorithms that are more difficult to implement, but much faster (quick sort). With the Mandelbrot it is the same: brute force is simple (and – almost – everyone has done it). But what better algorithms are there and why can they exist?

Better algorithms always take advantage of additional particularities that the "problem" presents. In the case of the Mandelbrot there is a characteristic that is very interesting: It is mathematically proven that the Mandelbrot set is "connected". Connected means that when you find a black point (= belongs to the set) you can get to any other part of the set going only from one black point to another. In other words: If we find an area that has a border only in black, we can be sure that all points inside that area are also black! Which means: We don't have to calculate the inside points, but can just use a fill algorithm to paint them all in black. Since black points are the ones that must iterate until the maximum, this can be a huge speed gain!

Now, what about the escaping points? Do they behave the same? I admit that I am no mathematician, so that I don't know if meanwhile it has been proven or not. But there is a strong hypothesis that says that escaping points behave the same as the black points inside the Mandelbrot. Which means (again): If we can find an area whose border consists only of points that have the same escaping (iteration) value, we can simply fill this area with the same color (and don't have to calculate them).

Since we are looking for color "borders" this technique is generally called "border trace". But again, there are different ways to implement border trace. A relatively simple way is to divide the areas into squares (for example vertically in the first step, then in the second step you divide both new squares horizontally into 2x2=4 squares and so on; whenever the border of the entire square is of the same color you can just use rectfill to paint the inside). The only problem with the Mandelbrot is that it is not really squared … so you always end up calculating more points than necessary.

Boundary trace algorithm [example5/boundary_trace0.c]

So, let's go for the true boundary trace algorithm! There is a nice video where you can see it in action (https://www.youtube.com/watch?v=rVQMaiz0ydk). The author, Joel Ylilhuoma, has also published the source code (https://bisqwit.iki.fi/jutut/kuvat/programming_examples/mandelbrotbtrace.pdf). In

order to speed things up a bit, let's just adapt this code for our Vampire and SAGA. We basically need some variables (of course):

```
UBYTE *Data;
UBYTE *Done;
ULONG *Queue;
ULONG DataSize;
ULONG DoneSize;
ULONG QueueSize;
ULONG QueueHead=0, QueueTail=0;

enum { Loaded=1, Queued=2 };
```

For which we allocate some memory:

```
void AllocateBoundary(void) {
   DataSize=sizeof(UBYTE)*resx*resy;
   DoneSize=sizeof(UBYTE)*resx*resy;
   QueueSize=sizeof(ULONG)*((resx*resy)*4);
   Data=AllocMem(DataSize, MEMF_PUBLIC | MEMF_CLEAR);
   Done=AllocMem(DoneSize, MEMF_PUBLIC | MEMF_CLEAR);
   Queue=AllocMem(QueueSize, MEMF_PUBLIC | MEMF_CLEAR);
}
```

Done is simply an array in which we keep track of whether a point has already been calculated or not. *Data* contains the iteration values (that define the colors). The *Queue* finally is a new element in the boundary trace algorithm: We can use the function AddQueue() to add new elements (points) to the *Queue*. A point is added to the *Queue* whenever one of the surrounding points doesn't have the same color (because it is a pixel immediately to the right or the left has a different color, it means that we are on a "border"). So, the algorithm checks for every point in the *Queue* if the surrounding points are the same color (if it is the case, the point is not added to the *Queue*, otherwise it is added). There are 9 surrounding points that have to be checked: left *(x-1,y)*, right *(x+1,y)*, up *(x,y-1)*, down *(x,y+1)*; up-left *(x-1,y-1)*, up-right *(x+1,y-1)*, down-left *(x-1,y+1)*, down-right *(x+1,y+1)*. Note that the *Queue* doesn't deal with two-dimensional coordinates *(x,y)* but only with one LONGWORD value *p* that is *y*resx+x* (exactly like in SAGA). We can calcute the corresponding *x* and *y* values from *p* by using divisions:

```
x = p % resx;          /* modulo division */
y = p / resx           /* rounded to int */
```

The algorithm passes through 3 phases:

(1) All border points are added to the *Queue*:

```
 for (y=0; y<resy; ++y) {
   AddQueue(y*resx);
   AddQueue(y*resx + (resx-1));
 }
 for(x=1; x<resx-1; ++x) {
   AddQueue(x);
   AddQueue((resy-1)*resx + x);
 }
```

(2) The *Queue* is then processed until it is empty:

```
while(QueueTail != QueueHead) {
      if(QueueHead <= QueueTail || ++flag & 3) {
         p = Queue[QueueTail++];
         if(QueueTail == QueueSize) QueueTail=0;
      } else p = Queue[--QueueHead];
      Scan(p);
}
```

Each element *p* of the *Queue* is sent to the function Scan() which checks if neighbouring points are different from the center color (adding it to the *Queue* if this is the case). As to the *flag* variable, it just guarantees that the mandelbrot will be calculated symmetrically.

(3) Finally, all uncalculated points are filled with the color that surrounds them. This part is actually very simple: The filling can be done line by line, there's always a starting color followed by all non calculated pixel and finally an end color (that is the same as the starting color).

Let's compile it with *smake boundary0* and run it with *boundary_trace0*. Isn't that nice how the Mandelbrot is calculated? (Much nicer than the brute force algorithm for sure!) Performance-wise we get an execution time of … 3.34 seconds – which means that this version is now 92% faster than the original!

Filling routines using words and longwords [example5/boundary_trace1.c]

Let's make one last optimization in our code that is again on the technical side. As we saw in one of our first example [*example2/saga_time.c*] filling the screen does also take time – around 0.57 seconds to be exact. Of course, when you start optimizing a Mandelbrot calculation that takes 41.71 seconds your problem is not exactly these 0.57 seconds … They are just negligeable! But now that we have brought our calculation down to 3.34 seconds we can start thinking about such details.
Actually, our Put8BitPixel() routine writes one BYTE after the other (calculating *y*resx+x* everytime). This is especially annoying in our final filling routine. No need to recalculate the offset everytime! In addition, if we have several points that have the same color, we could write WORDS (= 2 pixels) or LONGWORDS instead (actually, the best for 68080 is to write two LONGWORDS (= 64 bits) in one MOVE instruction).
Another thing that we can optimize is the *Data* variable. For the moment, we are using 256 iterations, so instead of having a separate Data and SAGA screen buffer (*newbuffer*) we could simply set the SAGA buffer to the *Data* buffer … This way, when we write to the *Data* buffer, we write directly on the screen. All this has been included in *example5/boundary_trace1.c*. You can compile it with *smake boundary1* and run it with *boundary_trace1*. The execution time comes down to 3.14 seconds … so we are now 92.48% faster than the initial program.

Compiler optimizations [example5/boundary_trace2.c]

I almost forgot. Of course, there are also compiler optimizations that we can use (eventually). So, *example5/boundary_trace2.c* is a last example where the source code contains several *__inline* directives (which means that the compiler will try to avoid a call and instead include the code directly inside the main program). Of course, this will result in a slightly bigger executable.

We can also add these options in the command line (see *example5/smakefile*, under *boundary2*) to tell VASM and SAS/C to optimize for speed:

For VASM: -opt-speed
For SAS/C: optimize opttime

Activating these options gives us a – rather meager – speed increase of 0.1 seconds. Which means that the final execution time is 3.04 seconds (or 92.72% faster than our initial version).

Is this the end … ?

Certainly not! As the title says: Optimizing code is a never ending process. Of course, there is a physical limit – somewhere – of what can ultimately be done. But in general this barrier lies very far ahead. The only thing we can say for sure is that the "last mile" is the hardest one … Which means: Bringing our calculation down from 41.71 to 3.04 seconds has been relatively easy. Every fraction of a second that we want to gain now will be hard work … But it can be done … In the last drawer called "teaser" you will find a Mandelbrot that calculates in 2.41 seconds (that is again 24% faster that our last example …).
And here comes, of course, my challenge: Can you write an even faster routine that calculates a Mandelbrot with the same parameters as given in this tutorial? Don't hesitate to give me a call if/when you succeed … ! :)
(But please, no, don't use x-axis symmetry or cardioid bulb optimizations … these work only with main view of the Mandelbrot, not when you zoom in … !:)

Special Thanks

… go to PaulUK and Kamelito who took the time to reread this document and send me corrections; the ApolloTeam (and especially BigGun) who patiently answered many questions; Frank Wille, Tommo, Beppo (and others) who – as true Amiga (and Atari:) fans – work passionately on key tools like VASM, MonAm and a modern GCC; and many other people (probably more or less my age, seriously fellows, we are getting old ...) who are part of this fascinating journey to make a dream come true – a new fast and modern Amiga (euh pardon: Amiga compatible) in 2019[2]!

For the rest:
Stay hungry, stay foolish!
Amiga rulez!

Fribourg / Switzerland, September 2022
by RedBug

(corrections / comments welcome – feel free to contact me on Discord)

2 https://www.forbes.com/sites/marcochiappetta/2019/10/22/the-next-generation-amiga-that-never-materialized-just-went-up-for-pre-order/?sh=929eee550dec