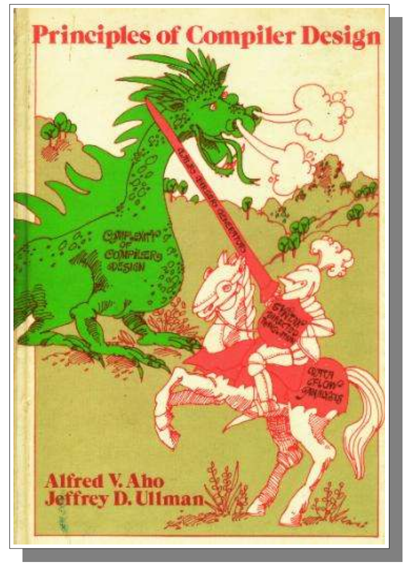
The never ending hunt for the fastest Mandelbrot

Part 2: In the eye of the compiler(s) – porting, compiling, analysis & results

Introduction



Why a second part? (Glad you asked … :) Because the first part kind of felt incomplete. All the examples were done for SAS/C, a compiler that was chosen for mainly two reasons. First, it was the original compiler used by Commodore for AmigaOS[1]. Second, it was said - or rumored - to be particularly good[2]. However, using SAS/C has its downsides. It's a proprietary compiler, so everyone may not have access to it, and it was released in 1992, making it quite old now.

That's why the idea came up to port the examples to other (free) compilers. These include GCC 2.95.3, VBCC 0.908 and GCC 6.5. The first two are included as "ready-to-use-out-of-the-box" options both in Coffin and ApolloOS (the two main operating systems used nowadays on our beloved Vampires). They can be used natively on the m68k architecture (which is nice and, personally, my preferred choice) or as cross-compilers on Linux/Mac/Windows (which can be useful as an additional option). For GCC 6.5 cross-compiling is, for the moment, the only option because versions beyond 3.x have not (yet) been ported to AmigaOS.

In addition to porting the code, I also wanted to compare results: How does each of these compilers perform in terms of code quality, speed and stability? What influence do compiler flags have on the executables? What differences can we observe in the generated assembly code? I have also put much effort into making the tutorial more accessible by adding icons and compiler scripts. In other words: Everyting should be "at your finger tips" now. So, get your Vampires ready and enjoy the read!

Chapter 1 – Short Overview

How to get started

To begin, extract the LHA archive and you'll find a folder named *Mandelbrot* containing all the necessary materials. The default configuration is for ApolloOS. For Coffin, simply click on the *Configure4Coffin* script. If you haven't read the first part of this tutorial (refer to the *Mandelbrot_Tutorial_68080.pdf* in the *PDF* folder), I strongly recommend doing so. The unmodified[3] sources and executables from the original tutorial have been included (in the *Original* folder). Once you have completed the first part, you can start exploring this PDF and the new materials included in the *Sources* folder. If you simply wish to explore without making any modifications, you can run the different examples by clicking on the icons in the *bin* folder. The same applies to all the source codes (C and assembly). You will find the handwritten sources in the main folder of each example and the automatically generated assembly code in the corresponding *asm* folder.

1 Due to its TRIPOS heritage, the first versions of AmigaOS (up to 1.4) were written in BCPL (and assembly, of course). SAS/C (formerly called Lattice C) was used from AmigaOS 2.0 on, see: http://obligement.free.fr/articles/programmation_amiga.php

2 Gilles Soulet, "C - choix du compilateur, efficacité du code généré", July 1992, in: http://obligement.free.fr/articles/c_choix_compilateur.php

3 Except for a bugfix in the boundary trace algorithm (example5) that was necessary for system stability. The fix (and the reasons for it) will be explained in Chapter 2 (Modification 6).

If you intend to modify and recompile the programs yourself, you must first “start” the compilers. This essentially means setting up the necessary assigns for all the tools and includes for GCC and/or VBCC. You can do this by either clicking individually on the scripts provided by both ApolloOS and Coffin[4] or by clicking on the *start_compilers* script, which automatically starts both VBCC and GCC. For GCC 6.5 please follow the instructions on the official Github page[5]. For SAS/C you can find some information in the Amiga C Tutorial by Peter John Hutchison[6].

The compilers

The compilers we are going to present span almost 30 years of computer history! Let’s begin by taking a quick look at the main characteristics of each one in (more or less) chronological order:

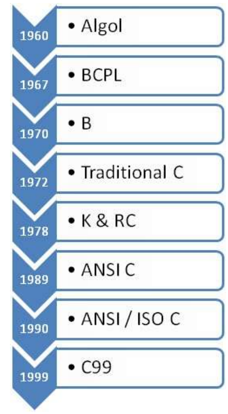
	SAS/C 6.58	GCC 2.95.3	VBCC 0.908	GCC 6.5[7]
Author	SAS Institute[8]	FSF	Volker Barthelmann	FSF[9]
Release date[10]	1992	2001	2022[11]	2018
License	proprietary	GPL	proprietary (free, even commercial use for AmigaOS m68k)	GPL
C/C++ Standard	C89 (ANSI-C)[12]	C99 / C++98[13]	C99	C++11
native / cross	+ / -[14]	+ / +	+ / +	- / +
Targets	m68k	many	several	many

Moving from left to right, we can observe that the newer the compiler, the higher the supported C/C++ standard[15]. We can also categorize native-only compilers[16] (SAS/C), compilers that support both native and cross-compiling (GCC 2.95.3, VBCC) and cross-compiling-only compilers (GCC 6.5). Last but not least, it is worth mentioning that VBCC[17] and GCC 6.5 are still actively beeing developped. While GCC has more association with the **nix* world (and can be useful for porting software from this

4 Note that the start script for GCC is included in Programs:Developer/ADE for ApolloOS and Programs:Programming/ADE for Coffin. When using GCC, you must also make the necessary assigns for VASM (included with VBCC).
5 https://github.com/bebbo/amiga-gcc
6 http://www.pjhutchison.org/tutorial/sas_c.html (Link to the C tutorial: http://www.pjhutchison.org/tutorial/amiga_c.html)
7 This version is under active development, we are using commit 50721c1bcdc79a96d1a832afbf6c0e101da1b866 (2023-03-31).
8 https://www.sas.com/en_us/software/base-sas.html
9 https://www.fsf.org/
10 https://gcc.gnu.org/releases.html
11 Even if VBCC only supports C99 (and no C++ at all) certain parts, and especially the assembler VASM, get updated on a regular basis.
12 SAS/C offers some very early (= not standardized) C++ features, but the only true standard supported is a (very strict) C89.
13 C++98 not fully complete (but far better C++ support than SAS/C).
14 Emulated “cross-compiling” is, of course, possible with VAMOS or WinUAE, FS-UAE etc.
15 There is now even a GCC cross-compiler version 12.2 which supports the C++23 standard: https://github.com/BartmanAbyss/vscode-amiga-debug
16 This category basically includes all “classic” compilers like Aztec C, DICE C, Storm C, HiSoft C++ (see https://en.wikipedia.org/wiki/Amiga_programming_languages)
17 This is especially true for the assembler VASM included in VBCC: VASM was one of the first assemblers to support 68080 instructions.

origin), SAS/C and VBCC are more directly linked to the Amiga platform. Additionally, VBCC is also a PPC compiler that can be used for and on platforms like MorphOS, for example.

Chapter 2 – Porting the code



C/C++ Standards

So, what efforts are necessary to port our Mandelbrot examples to these compilers? As far as the main programs, written in C, are concerned, they strictly comply with the (old) C89 (ANSI-C) standard. Therefore, apart from some minor details that need special attention, it should be possible to use these sections of the code almost as-is. The assembly part, though, is a bit more tricky: Each compiler has its own (= non standard) way to declare the prototype of an assembly function. In the following paragraphs, we are going to describe in more detail what exactly needs to be changed and why.

Modification 1: Global variables with GCC

When recompiling code with another compiler, even when it complies with a standard, you are never safe from surprises! Let’s take the following code snippet from *example0/simple_saga0.c* which looks innocent. But recompiling it with GCC 2.95.3 didn’t produce the expected result – not at all!

```
UWORD color;

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

/* define some random colors */
SetColor(0,0,0,0);
for (color=1; color<=255; color++) {
        SetColor(color, rand()%256, rand()%256, rand()%256);
}
```

After running the program the colors started flickering on the screen and the machine locked up, requiring a reboot. After some investigations, it turned out that GCC didn’t treat the parameter *color* as a new local variable with scope inside the function *SetColor* (as it should), but as identical to the global variable *color*. Theoretically, GCC offers the option *-fargument-noalias-global* which should fix the problem (and make GCC behave the same way as SAS/C and VBCC). But, for some unknown reason, that didn’t work. In the end, there was no other solution than to define the *color* variable inside main, which makes it a local variable with scope main.

Modification 2: A bug in MouseWait

A second surprise came up when recompiling *example0/simple_saga1.c* which includes *MouseWait* function. It turned out that this code – that seemed to work with SAS/C (but actually didn’t, it only worked when clicking the mouse very quickly) – produced a loop that didn’t react to anything on GCC and VBCC. So, this part had to be fixed by reading a WORD instead of a BYTE and by checking only for bit 6[18]:

---

18 In an attempt to simplify things in the C-code, we only checked globally if anything changed in CIAPRA, but the correct way according to http://www.amigadev.elowar.com/read/ADCD_2.1/Hardware_Manual_guide/node012E.html is to check for bit 6 in a BYTE (as we do later in assembly).

```
/* wait for mouse click */
void WaitMouse(void) {
        /* code rewritten - BUG in the 1st tutorial: read only a UBYTE (not WORD)  */
        /* use volatile keyword to be sure the compiler makes no optimizations */
        volatile UBYTE old_value=(*(volatile UBYTE*)CIAAPRA) & 64;
        while (old_value==((*((volatile UBYTE*)CIAAPRA) & 64)));
}
```

The volatile keyword ensures that the compilers do not make any optimizations on the variable *old_value*. Also, the variable CIAAPRA has been replaced by a #define (which actually makes more sense given that it refers to a fixed hardware register).

Modification 3: Explicit type casts

VBCC turned out to be rather picky when it comes to type checking, so additional explicit type casts had to be added in *example0/simple_saga1.c* to get rid of the compiler warnings. For example, in the following line:

```
/* (void*) cast necessary for vbcc */
if (!(newbuffer=(ULONG)(void*)Set8BitMode(0x0a01,1280,720))) return 1;
```

a direct cast from UBYTE* (= return value of *Set8BitMode*) to ULONG (type of *newbuffer* variable) was only possible via a void pointer *void** (so that the whole thing ends up in a double or triple cast). Anyway, these are “compiler eccentricities”. The code works perfectly even with the warning, but a good program doesn’t throw any warnings when it compiles. The problem, by the way, only occurs with the function *Set8BitMode* written in C (VBCC doesn’t seem to have a problem when this function is written in assembly, so that the double cast can be omitted).

Modification 4: Assembly function prototypes

This was the part that required most of the changes because keywords like *__asm* or *__d0 … _d7* and *__fp0 … _fp7* for registers are specific to SAS/C (and not known or different on the other compilers). Some investigations on the “English Amiga Board” (EAB) revealed that there is a nice include file that contains some macros that make it possible to compile the same code on different compilers[19]. As an example, by including “asm_call.h” a function like

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

in SAS/C can be rewritten

```
#include “asm_call.h”

ASM UBYTE* Set8BitMode( REG(d0,UWORD),
        REG(d1, UWORD),
        REG(d2, UWORD));
```

and then compiles perfectly on SAS/C, VBCC and GCC (both 2.95.3 and 6.5).

19 The thread can be found here: https://eab.abime.net/showthread.php?t=80274 (direct link to the archive containing the include file: https://eab.abime.net/attachment.php?s=65ddf3cd4f4954972f2f161990856283&attachmentid=50878&d=1479231099)

Modification 6: Boundary Trace and Buffer Overflow

Another (rather nasty) bug was lurking in boundary trace examples. These examples seemed to work well with SAS/C and on AmigaOS. But once compiled with VBCC & GCC and tested on ApolloOS, the result was an instant freeze or a system crash. This problem was due to the final filling routine that creates a buffer overflow in the main loop:

```
for (p=0; p<resx*resy-1; ++p) {
        /* rest of the filling routine */
}
```

The variable *p* points to the SAGA screen buffer (which is of size *resx*resy*). But the point is: Doing the pre-increment *++p*, the loop must stop at *resx*resy-1*. It was a small buffer overflow (1 Byte) that passed undetected on AmigaOS – but on ApolloOS, it was fatal!

Modification 7: Clock Multiplyer

Most of the Apollo 68080 cores run at 85 Mhz, which corresponds to a clock multiplyer of 12 compared to the original frequency of the 68000. But from time to time, the team releases cores that run at 92 Mhz (13x) or 100 Mhz (14x). In these cases, the function *GetTime* will of course return a wrong value. So, we are going to fix that with the line:

```
ULONG frequency=(*((UBYTE*)(0xDFF3FC+1)))*7.09379*1000000;     /* read clock multiplyer */
```

The custom register DFF3FC contains two pieces of information:

```
| BIT#  | FUNCTION | DESCRIPTION                               |
+-------+----------+-------------------------------------------+
| 15-08 | Card     | Card Version                              |
|       |          | 1=V600, 2=V500, 3=V4_500(Firebird),       |
|       |          | 4=V4_1200(Icedrake), 5=V4SA, 6=V1200      |
| 07-00 | Clock    | Clock multiplier                          |
+-------+----------+-------------------------------------------+
```

So, we simply replace the static value 12 by the value we are reading at address DFF3FC+1 as a BYTE (the register has WORD size, we are reading the lower BYTE).

Modification 8: Printf with floats

The original tutorial prints out the execution time at the end of each example by using a printf-function with a floating point variable *exectime*:

```
printf("Execution time: %f seconds\n", exectime);
```

Unfortunately, this simple statement may cause problems with GCC in the sense that *printf*, by default, only prints integer values and needs a math library (or something similar) to print floats. In the case of GCC, the standard way is to use the *ixemul* library which works great, but … depending on the version may add a lot of code to the executable. There are other alternatives (e.g. *libnix*) that add less overhead. But the point is that we want to compare the sizes of the executables and therefore, we decided not to use any additional library just to print a float variable. We can do this by rewriting this part as:

```
ULONG intpart;
ULONG floatpart;

intpart=(ULONG)exectime;
floatpart=(ULONG)((exectime-(double)intpart)*10000);
printf("Execution time (%dx core): %u.%04u\n", (*((UBYTE*)(0xDFF3FC+1))), intpart, floatpart);
```

We’ve also integrated the clock multiplyer. The reason why we multiply by 10000 and only use %u (instead of %lu) is that we want to avoid any problem with differences related to variable sizes on different compilers (and also to avoid warnings popping up with VBCC who doesn’t seem to like the %lu format). An *unsigned int* (UWORD) can normally hold values up to 65535, so multiplying by 10000 is safe and will give us 4 digits after the coma (or decimal point). This will be accurate enough for our performance tests.

Remaining issues: Warnings

As we’ve said it is good practice to treat warnings like errors and rewrite the code, so that the warning doesn’t pop up any more. Nonetheless, there still remain some warnings that are difficult to get rid off. On GCC 6.5, for example, this warning comes up:

```
boundary_trace2.c:50:12: warning: built-in function 'yn' declared as non-function
double xn, yn, xn1, yn1, cx, cy, d, stepr, stepi, maxr, minr, maxi, mini;
```

Considering that it is not critical and not related in any way to AmigaOS and what we are doing here we’ve decided to keep it that way[20].

Chapter 3 - Compiling

Flags and Options

After all these modifications and bugfixes the examples finally worked on all compilers. So, the next question was: What options should be used for the compiling? In general, there are different categories. GCC, for example, offers the following possibilities:

1. General options: These options are related to general compiler behavior and output format. They typically include options like *-o* to specify the output file, *-c* to compile without linking, and *-s* to strip the executable of non-essential symbols.
2. C/C++ language standard: Specify the language standard to be used. For example, *-std=c89* for C89 (ANSI-C) standard, *-std=c99* for C99 standard, or *-std=c++11* for C++11 standard.
3. Debugging information: Options related to debugging information, such as *-g* to include debugging symbols in the executable.
4. Processor options: These options allow you to optimize the code specifically for the target processor architecture. For example, *-march=xxx* to specify the target architecture, *-mcpu=xxx* to optimize for a specific CPU, or *-mfpu=xxx* to specify the floating-point unit.
5. Optimization options: The most important set of options for performance optimization. Some commonly used options include *-O1, -O2, -O3* for different levels of optimization (higher number corresponds to more aggressive optimization), *-Os* to optimize for code size, or *-finline-functions* to enable function inlining.

The most interesting part for our purpose is, of course, the optimization options. Generally, with almost every compiler, you can improve speed or size. Optimizing for speed often means increasing the size of the executable and vice versa. So, these two opimizations goals are, at least up to a certain extent, mutually exclusive.

---

20 https://github.com/open-power/HTX/issues/15

Nonetheless, we can try to reduce the size of our executables – as a secondary goal – by stripping out non necessary debugging information. While SAS/C and VBCC offer a relatively modest selection of global options, GCC excells with an incredibly large number of very fine-grained and detailed “flags”[21] Facing the sheer number of options, it is simply impossible to discuss them in detail here. Instead, we are pragmatically going to stick to the most important ones:

	SAS/C 6.58	GCC 2.95.3	VBCC 0.908	GCC 6.5
CPU C-standard	cpu=68040 math=68881	-m68040 -m68881	-fpu=68881[22]	-m68040 -m68881
Speed	optimize opttime optalias optinline	-O3 -fomit-frame-pointer	-O4 -speed	-O3 -fomit-frame-pointer
Size	stripdebug	-s[23]	-[24]	-noixemul[25] -s

Optimization 1: Speed

When it comes to optimizing for speed, the most important option for VBCC and GCC is the *-Ox* flag, where x indicates the optimization level. In our case, we will choose the highest level available, which is *-O3* for GCC and *-O4* for VBCC[26]. Additionally, for GCC, we will include the option *fomit-frame-pointer*. This option instructs the compiler to avoid creating a stack frame whenever possible when a function is called. For SAS/C, the available options are more limited. However, we can activate the *opttime* option, which is a general option for speed optimizations. We can also enable *optalias*, which is similar to the *-fomit-frame-pointer* option in GCC, and *optinline*, which includes function code directly in the main program to avoid function calls.

Optimization 2: Size

To reduce the size of the executable, we can use te *-s* flag (GCC) and the *stripdebug* option (SAS/C) that remove non-essential debugging information. It's important to note that when compiling with GCC, we need to be cautious about size because GCC, being a compiler from the **nix* world, may require the ixemul library to emulate certain functions. As a general rule, we should use the *noixemul* option whenever possible to avoid unnecessary library dependencies.

Optimization 3: Assembly

To take advantage of the ApolloCore's new features and achieve optimal performance, many parts of the examples were written in pure 68080 assembly. In the first part of the tutorial, these assembly functions were compiled with VASM, and we will continue to use it for the same reasons. VASM is one of the few assemblers that supports almost all new features of the Vampire. It produces object files that can be linked with any of the presented compilers, allowing seamless integration of assembly code with the C/C++ examples.

21 https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html

22 VBCC also offers an -cpu=x option, but for some unexplained reason, values like 68020 or 68040 don’t word (so we don’t use any specific value here).

23 On GCC 2.95.3 it turns out that the executables are actually smaller than with the -noixemul option (so that we do not use it).

24 As far as we understand it, VBCC executables by default do not contain debugging information, but use -g option to explicitely add it.

25 As already explained, we use the -noixemul option to get a smaller size of the executable (ixemul adds an overhead of around 65 KB, or 80 KB in total, for an executable that reaches 15 KB without it).

26 The following page gives a detailed explanation of what the different optimization levels include: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. For VBCC you might have a look at the manual: http://www.ibaug.de/vbcc/doc/vbcc.pdf

Chapter 4 – Results

Let’s compare … !

Now let's move on to the important part, i.e., the comparison of the four compilers. We will start with some general observations and then delve into more specific aspects such as stability, compiling time, size and execution time of the executables. Finally, we will conduct a more detailed analysis of the generated assembly code.

Comparison 1: General Observations

One of the first and obvious observations is that the Mandelbrot colors of the compilers are different (but consistent for the same compiler). This is simply due to the *rand* function specific to each compiler that returns different values. Other than that the results are very similar, although there are some interesting differences as to the size and execution time, as we will see.

Comparison 2: Stability

As we all know, AmigaOS is an operating system which lacks (or purposely doesn’t make use of) memory protection, with all its joys and sorrows. On the positive side, we have the overall performance of the system (clocked at only 85 Mhz in the case of the Vampire, it is amazingly fast) and the fact that everything is accessible to anyone. The latter has been very important in the evolution of AmigaOS. For example, it wouldn't be possible to have modern, high-resolution screens nowadays if AmigaOS hadn't allowed, by design, the patching of system functions. On the downside, we have all the problems related to “memory trashing”. All programs are responsible for handling memory allocation correctly, or they may crash the system. Therefore, this topic is of utmost importance for the "stability" of a program.

It is evident that measuring the "stability" of a program is difficult because a system crash may occur sporadically and may not even be related to the program itself. The culprit might be another program running in the background, or it might be the system – the combination of different components – that is inherently unstable. Nonetheless, we will define a simple test scenario: we will use a script called *build_all_sasc/gcc/vbcc/gcc6* (located in the *Scripts* folder) that will compile all 15 examples in one run. We will see if we can successfully run this test several times in a row (and how many times). We will run this test on Coffin R62 (Linux Debian for GCC 6.5). If we manage to run the test 20 times without any major issues (system freeze or crash), we will consider the compiler to be reasonably stable. In case of a crash, we will reboot and retry, assuming that the crash was not caused by the compiler itself but by something else. The results are as follows:

	SAS/C 6.58	GCC 2.95.3	VBCC 0.908	GCC 6.5
Successful builds	20	2	20	20
Ranking	2	4	2	1

We were able to complete 20 successful runs with three compilers: SAS/C, VBCC and GCC 6.5. However, with GCC 2.95.3, it was simply impossible to achieve more than two successful builds in a row. Most of the time, the compiler stopped, the system froze, or even crashed completely. We tried different stack sizes (ranging from 60000 KB up to 1 MB), but it didn't help. Perhaps there is an important configuration option we missed, or maybe the compiler is stable, but the linker or another component is causing the instabilities. However, as an overall impression and to be honest, this compiler was truly a nightmare[27] to work with! Therefore, in our ranking, the first place goes to GCC 6.5, as it can typically handle an almost infinite number of runs when running on a system with memory protection. Regarding SAS/C and VBCC, it is difficult to determine which one is more stable, so we rank them both at second place. The last place, without a doubt, belongs to GCC 2.95.3.

Comparison 3: Time

The next aspect we want to examine is the speed of each compiler. For this test, we are once again using the same build scripts, which means we will compare the overall building time for all 15 examples, including the assembly part with VASM. Due to various factors such as the speed of the drive and other running tasks, the building times can vary slightly. We conducted this test multiple times and tried to note the best and worst results as accurately as possible:

	SAS/C 6.58	GCC 2.95.3	VBCC 0.908	GCC 6.5
Time (seconds)	25-32	118-131	188-223	2
System (processor, core, frequency, OS)	Vampire V4 85 Mhz (core 9392) Coffin	Vampire V4 85 Mhz (core 9392) Coffin	Vampire V4 85 Mhz (core 9392) Coffin	Intel Core2Duo 2.53 Mhz Debian Buster
Ranking	2	3	4	1

In this aspect as well, the clear winners are GCC 6.5 (for cross-compiling) and SAS/C (for native compiling). However, to be fair, we must mention that the option *-O4* has a particularly negative impact on the compiling time in the case of VBCC. This is especially evident in the boundary trace examples: building the first 12 examples only takes 37 seconds, while the last 3 examples take the remaining time, i.e. 2.5-3 minutes.

Comparison 3: Size

The next aspect we are interested in is the size of the generated executables. As we've already mentioned, this depends heavily on the libraries we choose to link with the final executable. We have made efforts to find optimal options for each compiler, and the final results are as follows:

	SAS/C 6.58		GCC 2.95.3		VBCC 0.908		GCC 6.5	
simple_saga0 simple_saga1	2548 2604	1	5612 5680	4	2928 3072	3	2636 2676	2
saga_main0 saga_main1	2596 2552	1	5392 5332	4	2728 2432	2	2604 2592	3
saga_time	12004	4	6308	1	9532	2	10684	3
brute_force0 brute_force1 brute_force2 brute_force3	12400 12392 12495 12596	4	6872 6749 6848 6944	1	10156 10084 10188 10288	2	11228 11092 11196 11296	3

27 To be fair, two full builds means that we were able to compile 2x15=30 examples with GCC before it became unusable. So, compiling single programs with GCC should work most of the time. However, this has a negative impact on productivity when trying larger batches of programs (as we are doing here).

parmandel0 parmandel1 parmandel2	13372 13416 13520	4	7724 7800 7984	1	11020 11080 11256	2	12072 12136 12272	3
boundary_trace0 boundary_trace1 boundary_trace2	14744 15036 17032	3	11692 12232 14376	1	15572 16108 16108	4	13488 13936 15564	2
FLAGS	optimize opttime optalias optinline stripdebug		-fomit-frame-pointer -O3 -s		-O4 -speed		-noixemul -fomit-frame-pointer -O3 -s	
Ranking [Total]	4	17	1	12	2	15	3	16

As we can observe, there are indeed some differences between the compilers, but in our opinion, they are only minor, and the results are very similar.

Comparison 4: Code Quality & Speed

Finally, let's examine what we are most interested in: the speed of the generated executables. Can we surpass the record set in our first tutorial (3.04 seconds for *boundary_trace2* with SAS/C) by using any of the other compilers? For the following tests, we run the executables multiple times and record the best result[28]:

	SAS/C 6.58		GCC 2.95.3		VBCC 0.908		GCC 6.5	
saga_time	0.33 (0.57)[29]	1	0.33	1	0.73	4	0.33	1
brute_force0 brute_force1 brute_force2 brute_force3	42.58 (41.71) 30.08 (30.31) 28.37 (28.61) 20.96 (21.18)	4	29.91 29.71 28.01 20.59	1	41.77 30.15 28.45 21.04	3	30.04 29.77 28.08 20.57	2
parmandel0 parmandel1 parmandel2	21.49 (21.51) 15.63 (15.66) 13.40 (13.42)	4	21.14 15.53 13.41	2	21.09 15.41 13.31	1	21.22 15.50 13.36	3
boundary_trace0 boundary_trace1 boundary_trace2	3.27 (3.34) 3.06 (3.14) 3.03 (3.04)	3	2.95 2.75 2.73	1	3.51 3.18 3.18	4	3.13 2.87 2.70	4
Ranking [Total]	3	12	1	5	3	12	2	10

This test indeed demonstrates that both versions of GCC produce very good results. GCC 2.95.3 emerges as the clear winner, but GCC 6.5 also delivers strong performance and even produces the fastest executable in the entire test. SAS/C and VBCC achieve similar levels of performance.

28 The differences between different runs are, in general, very small (typically around 0.02 seconds at most), because we are turning off multitasking (so that no other task – or even DMA – can interfere) and we use the *movec CCC,register* instruction to get a very precise mesurement. The only differences can come from inside the core – like a branch prediction that works a little bit better on one run than on the other.

29 Performance values in red are from the first tutorial.

Chapter 5 – Analysis

Let's gain some insights

The intriguing question now is: Why are certain executables faster than others? The answer lies in the generated "machine code" (opcodes). Since our build script uses the *-S* option, which creates an output of the assembly code generated by the compiler, we can now examine some examples and make comparisons. Examples that exhibit significant differences are particularly interesting:

- *example2/saga_time*: Why is VBCC half as fast than the other compilers?
- *example3/brute_force0*: What contributes to GCC speed advantage compared to the other two compilers?
- *example5/boundary_trace2*: What accounts for the speed increase in GCC?

Analysis 1: *example2/saga_time*

The generated assembly file can be found in the *asm* folder. Each file has the .s extension (for assembly code) and contains the name of the compiler (sasc=SAS/C, vbcc=VBCC, gcc=GCC 2.95.3, gcc6=GCC 6.5). The first challenge when inspecting automatically generated assembly code is: How do we identify the part of code we are interested in? Well, there is a relatively straightforward way to do this: All our programs utilize the *setstart* and *setstop* assembly function to mesure the main, time critical part of our program. Therefore, we primarily need to locate the corresponding BSR (branch to subroutine) instruction and focus on the code between them.
The second challenge we encounter is that each compiler uses its own format[30], and sometimes there is additional information that might be confusing. So, taking the example of SAS/C, let's clean up the source a bit. For instance, we can convert this:

```
              BSR.W         SetStart                  ;6100 0000
              CLR.W         __MERGEDBSS+$16(A4)       ;426c 0016
              BRA.B         ___main__10               ;6034
___main__6:
              CLR.W         __MERGEDBSS+$14(A4)       ;426c 0014
              BRA.B         ___main__8                ;6020
___main__7:
              MOVEQ.L       #$0,D0                    ;7000
              MOVE.W        __MERGEDBSS+$14(A4),D0    ;302c 0014
              MOVEQ.L       #$0,D1                    ;7200
              MOVE.W        __MERGEDBSS+$16(A4),D1    ;322c 0016
              MOVE.W        #$ff,D2                   ;343c 00ff
              AND.W         D1,D2                     ;c441
              MOVEQ.L       #$0,D3                    ;7600
              MOVE.W        D2,D3                     ;3602
              MOVE.L        D3,D2                     ;2403
              BSR.W         Put8BitPixel              ;6100 0000
              ADDQ.W        #$1,__MERGEDBSS+$14(A4)   ;526c 0014
___main__8:
              MOVE.W        __MERGEDBSS+$14(A4),D0    ;302c 0014
              CMP.W         __MERGEDBSS+$18(A4),D0    ;b06c 0018
              BCS.B         ___main__7                ;65d6
___main__9:
```

30 Beacause each compiler uses its own assembler: *asm* for SAS/C, *gas* for GCC and *vasm* for VBCC.

```
              ADDQ.W        #$1,__MERGEDBSS+$16(A4)  ;526c 0016
___main__10:
              MOVE.W        __MERGEDBSS+$16(A4),D0   ;302c 0016
              CMP.W         __MERGEDBSS+$1a(A4),D0   ;b06c 001a
              BCS.B         ___main__6               ;65c2
___main__11:
              BSR.W         SetStop                  ;6100 0000
```

to this:

saga_time_sasc.s (SAS/C 6.58)	Corresponding C-Code

<pre>1   BSR.W   SetStart
2   CLR.W   y
3   BRA.B   main10
4   CLR.W   x
5   BRA.B   main8
6 main7:
7   MOVEQ.L #$0,D0
8   MOVE.W   x,D0
9   MOVEQ.L #$0,D1
10   MOVE.W   y,D1
11   MOVE.W   #$ff,D2
12   AND.W   D1,D2
13   MOVEQ.L #$0,D3
14   MOVE.W   D2,D3
15   MOVE.L   D3,D2
16   BSR.W   Put8BitPixel
17   ADDQ.W   #$1,x
18 main8:
19   MOVE.W   x,D0
20   CMP.W   resx,D0
21   BCS.B   main7
22 main9:
23   ADDQ.W   #$1,y
24 main10:
25   MOVE.W   y,D0
26   CMP.W   resy,D0
27   BCS.B   main6
28 main11:
29   BSR.W   SetStop</pre>
	<pre>for (y=0; y<resy; y++) {
	      for (x=0; x<resx; x++) {
	            Put8BitPixel(x,y,y%256);
	      }
	}
	
	MERGEDBSS+$14(A4) = x
	MERGEDBSS+$16(A4) = y
	
	MERGEDBSS+$18(A4) = resx
	MERGEDBSS+$1a(A4) = resy</pre>

We can observe several things here:

1) SAS/C keeps the variables *x* and *y* in memory (CLR.W in lines 2 and 4, ADDQ.W in lines 17 and 23) and copies them to a register D0 for comparisons with *resx* and *resy* (lines 19-20 and 25-26).
2) SAS/C utilizes the BSR, BRA.B and BCS.B instructions for branching and subroutine calls, particularly in the case of *Put8BitPixel*, which will be called 1280x720 = 921600 times inside the loop.
3) SAS/C replaces the modula division *resy%255*, which, in binary, is a special case, by an AND instruction.

Now, let's discuss this approach:

1. Loop variables in memory: In handwritten assembly you would probably aim to avoid this because, in general, memory accesses are slower than register operations. Since there is only a very small number of variables used here (*x*, *y*, *resx*, *resy* and *resy*%255), it would probably be possible to do all this in registers D0-D7.
2. Branching with Bcc: This is advantageous! The alternative would be to use a JMP instruction, but the opcode for a JMP instruction is longer than a BCC[31]. A smaller opcode results in a faster program.
3. Modulo for multiples of 2: This is also a positive optimization, which is only possible when the divisor is a power of 2.

Another positive aspect is that the parameters for *Put8BitPixel* are passed through registers (D0=x, D1=y, D2=color). However, this is not solely the compiler's merit, as we specifically enforced this calling convention in our prototype declaration.

Now, let's compare with what VBCC does:

saga_time_vbcc.s (VBCC 0.908)				Corresponding C-Code / Optimizations / Comments		
1		jsr	_SetStart	for (y=0; y<resy; y++) {		
2		move.w	#0,_y	for (x=0; x<resx; x++) {		
3		move.w	_y,d7	Put8BitPixel(x,y,y%256);		
4		cmp.w	_resy,d7	}		
5		bcc	l45	}		
6	l42					
7		move.w	#0,_x			
8		move.w	_x,d7			
9		cmp.w	_resx,d7			
10		bcc	l46	Improved code:		
11	l43			(DIV):	(AND):	(MOVE.B):
12		moveq	#0,d2	clr.l d2	clr.l d2	clr.l d2
13		move.w	_y,d2	move.w _y,d2	move.w _y,d2	move.b _y+1,d2
14		move.l	#256,-(a7)	divu.w #255,d2	and.l #$ff,d2	-
15		move.l	d2,-(a7)	swap d2	-	-
16		public	__ldivs	-	-	-
17		jsr	__ldivs	-	-	-
18		addq.w	#8,a7	-	-	-
19		move.l	d1,d0	-	-	-
20		move.l	d0,d2	-	-	-
21		moveq	#0,d1			
22		move.w	_y,d1			
23		moveq	#0,d0			
24		move.w	x,d0			
25		jsr	Put8BitPixel	D0=x, D1=y, D2=y%256		
26		moveq	#1,d0	This kind of code is very fast on the Vampire:		
27		add.w	_x,d0	The ApolloCore 68080 can "fuse" lines 26-27		

31 BSR.S is 2 bytes, BSR.W is 4 bytes whereas JSR is 6 bytes. See: https://mrjester.hapisan.com/04_MC68/Sect05Part06/Index.html

28	`        move.w   d0,x`	and execute 2 instructions in 1 cycle![32]
29	`        cmp.w    _resx,d0`	
30	`        bcs      l43`	
31	`l46`	
32	`        moveq    #1,d0`	Same comment as for lines 26-27:
33	`        add.w    _y,d0`	moveq+add get “fused” (= executed in 1 cycle)
34	`        move.w   d0,y`	
35	`        cmp.w    _resy,d0`	
36	`        bcs      l42`	
37	`l45`	
38	`        jsr      _SetStop`	

VBCC also keeps the variables in memory and performs similar operations for comparisons (copying *x* and *y* to *d0* and comparing them to *resx* and *resy* from memory). It inlcudes one extra instruction for addition (compared to SAS/C, which performs the addition directly in memory), but this difference is minor. What really changes in VBCC is:

1. It uses the JSR instead of BSR for Put8Pixel (line 25). This is unlikely to have a significant impact on performance (but can be tested to confirm).
2. It utilizes a function call for the modula division (lines 14-20). This is a major drawback, especially considering that it uses the stack to pass the function parameters (lines 14-15) and must transfer the return value from *D0* to *D2* for the *Put8BitPixel* call (lines 19-20).

Fortunately, with VBCC we have the ability to modify the assembly code and recompile it to measure the difference. We will test the following optimization (all the files can be found in the *rec* folder, the included optimizations are highlighted in purple):

1. Replace the function call by the DIV instruction (*saga_time_vbcc_opt1*).
2. Use the AND instruction (like SAS/C) instead of DIV (*saga_time_vbcc_opt2*).
3. Use a MOVE.B instruction as an even better optimization (saga_time_vbcc_opt3).
4. Replace JSR by BSR (*saga_time_vbcc_opt4*).

For the recompiling we use the following commands (alternatively, you can use the *vbcc_recompile* script in the *rec* folder):

vasm -m68080 -m68881 -Fhunk saga_asm.s -o saga_asm.o
vc saga_time_vbcc_optx.s saga_time.o -o saga_time_vbcc_opt

The results are as follows:

Optimization	Time (seconds)	Percentage
none	0.73	100%
1	0.5	68%
2	0.29	40%
3	0.28	38%
4	0.28	38%

32 For more details have a look at: http://www.apollo-core.com/knowledge.php?b=4&note=21090

We can clearly see that small changes to the code can have a significant impact on performance. By replacing 7 lines of code with just 2 optimized instructions, we are able to improve the programs speed more than twice! Replacing JSR by BSR, on the other hand, has no measurable effect on performance.

Now, let's compare with what GCC 2.95.3 does:

	saga_time_gcc.s	Comments
1	jbsr SetStart	
2	clrw y	← Use registers and update this only in the end
3	tstw resy	← tstw and jeq instruction is not necessary!
4	jeq L22	(probably an "early exit" optimization by GCC)
5	clrw d6	← Use d1 directly (= y for Put8Pixel call)
6	.even	
7	L24:	
8	clrw x	← As for x use register d0 (= x for Put8Pixel)
9	cmpw resx,d6	← Why this cmpw here? (there's one later)
10	jcc L23	
11	clrl d5	From here on, GCC starts to clear registers and
12	clrl d4	then (line 16ff) starts moving around values …
13	clrl d3	(this is unnecessary and wastes cpu time)
14	.even	
15	L28:	
16	movew y,d0	
17	movew d0,d1	
18	andw #255,d1	
19	movew d1,d5	
20	movew d0,d4	
21	movew x,d3	
22	movel d5,d2	
23	movel d4,d1	
24	movel d3,d0	
25	jbsr Put8BitPixel	
26	movew x,d0	GCC keeps memory variable x always updated: we
27	movew d0,d1	know that this is not necessary because our
28	addqw #1,d1	loop will run through the whole range: 0-1279
29	movew d1,x	
30	addqw #1,d0	
31	cmpw resx,d0	
32	jcs L28	
33	L23:	
34	movew _y,d0	Same comment as for x (for y: 0-719)
35	movew d0,d2	Lines 35-36 (and 27-28) are very fast
36	addqw #1,d2	(= execute in 1 cycle thanks to "fusing"[33]).
37	movew d2,_y	
38	addqw #1,d0	
39	cmpw _resy,d0	
40	jcs L24	
41	L22:	
42	jbsr _SetStop	

33 For more details have a look at: http://www.apollo-core.com/knowledge.php?b=4&note=21090

We know that this code runs fast, but let's be honest: To a human programmer this code looks rather verbose … ! Let's see if we can simplify everything sticking to the essentials (and be at least as fast as this automatically generated code). So, let's rewrite this code with some principles in mind:

1. Use registers, do not do any memory access during the loop (if possible).
2. Use directly the registers that will be used for the function call to *Put8BitPixel* (d0=x, d1=y, d2=color)[34]
3. Use AND (or MOVE) for modulo.

The rewritten code, which is considerably shorter, now takes the following form:

saga_time_gcc_opt1.s	Opimizations
`        jbsr SetStart`	
`        clrw d1`	y
`        .even`	
`loopy:`	
`        clrw d0`	x
`        .even`	
`loopx:`	
`        movew d1,d2`	y++
`        and.l #255,d2`	y%255
`        movem d0-d1,-(sp)`	The two movem are (unfortunately) necessary
`        jbsr Put8BitPixel`	because Put8Pixel trashes d0 and d1 ...
`        movem (sp)+,d0-d1`	
`addx:`	
`        addqw #1,d0`	x++
`        cmpw resx,d0`	x<resx?
`        jcs loopx`	
`addy:`	
`        addqw #1,d1`	y++
`        cmpw _resy,d1`	y<resy?
`        jcs loopx`	
`L22:`	
`        movew d0,x`	this is not strictly required (but let's update
`        movew d1,y`	these memory variables anyway)
`        jbsr _SetStop`	

As with VBCC, we can proceed with recompiling the code by using the following command (or utilizing the *gcc_recompile* script in the *rec* folder):

gcc saga_time_gcc_opt.o saga_asm.o -o saga_time_opt_gcc

Upon execution, we obtain a runtime of 0.29 seconds, resulting in a modest improvement of only 0.04 seconds. In this case, the speed increase is not particularly significant. However, it is evident that handwritten assembly code is more intuitive in the sense that it is (1) shorter and (2) easier to understand.

34 Unfortunately this can't be fully done because Put8Pixel trashes the registers d0 and d1 (so that they have to be saved somewhere – we opted for the stack). Of course, all this could be further optimized, but it is not the purpose of this tutorial.

Analysis 2: *example3/brute_force0*

Let’s move on to our next example: *example3/brute_force0*. This case is interesting because the main loop is entirely written in C. The execution times range from 42.58 seconds (SAS/C) and 41.77 seconds (VBCC) to 29.91 seconds (GCC 2.95.3):

```
/* Mandelbrot brute force algorithm */
for (y=0; y<resy; y++) {
        for (x=0; x<resx; x++) {
                /* "optimized escape time" for inner loop */
                xn1=xn=0;
                yn1=yn=0;
                cy = y*stepi+mini;
                cx = x*stepr+minr;
                i=MaxIter;
                while ((i) && (xn1+yn1<=4)) {
                        yn=2*xn*yn+cy;
                        xn=xn1-yn1+cx;
                        xn1=xn*xn;
                        yn1=yn*yn;
                        i--;
                }
                Put8BitPixel(x,y,i%256);
        }
}
```

Unfortunately, recompiling with SAS/C is a cumbersome process[35], so we'll use the VBCC code instead, which exhibits only a slight performance advantage over SAS/C.

brute_force0_vbcc.s			C-Code equivalent
	move.w	#0,_y	y=0
	move.w	_y,d7	
	cmp.w	_resy,d7	y<rexy?
	bcc	l58	
l54			
	move.w	#0,_x	x=0
	move.w	_x,d7	
	cmp.w	_resx,d7	x<resx?
	bcc	l59	
l55			
	move.l	#$00000000,_xn	xn1=xn=0;
	move.l	#$00000000,4+_xn	
	move.l	4+_xn,4+_xn1	
	move.l	_xn,_xn1	
	move.l	#$00000000,_yn	yn1=yn=0;
	move.l	#$00000000,4+_yn	
	move.l	4+_yn,4+_yn1	

35 In general, the automatically generated code from SAS/C can not be recompiled without manual modifications. For example, SAS/C adds the opcodes of the instructions at the end of each line after a semicolon that marks a comment. Unfortunately, if a line is very long, the semicolon follows immediately the last token of the instruction which will generate a compiler error. Also, many of the symbols have to be adapted manually (which means adding or deleting underscores before symbol names). Last but not least, when recompiling you have to take care to link the correct libraries.

18		move.l	_yn,_yn1	
19		moveq	#0,d7	
20		move.w	_y,d7	cy = y*stepi+mini;
21		fmove.l	d7,fp0	
22		fmul.d	_stepi,fp0	
23		fadd.d	_mini,fp0	
24		fmove.d	fp0,(0+l76,a7)	
25		move.l	(4+l76,a7),4+_cy	
26		move.l	(0+l76,a7),_cy	
27		moveq	#0,d7	
28		move.w	_x,d7	cx = x*stepr+minr;
29		fmove.l	d7,fp0	
30		fmul.d	_stepr,fp0	
31		fmove.d	_minr,fp6	
32		fadd.x	fp0,fp6	
33		fmove.d	fp6,_cx	
34		move.w	_MaxIter,_i	i=MaxIter;
35		beq	l24	i>0?
36		fmove.d	_xn1,fp4	while ((i) && (xn1+yn1<=4))
37		fmove.d	_yn1,fp5	
38		fmove.x	fp5,fp0	
39		fadd.x	fp4,fp0	fp0 = xn1 + yn1
40		fmove.d	fp6,(8+l76,a7)	
41		fcmp.d	#$4010000000000000,fp0	
42		fbgt	l24	xn1+yn1<=4?
43		fmove.d	_yn,fp6	yn=2*xn*yn+cy;
44		move.w	_i,d4	
45	l52			
46		fmove.d	_xn,fp0	
47		fmul.d	#$4000000000000000,fp0	fp0 = 2*xn
48		fmul.x	fp6,fp0	fp0 = 2*xn*yn (= fp6)
49		fadd.d	(0+l76,a7),fp0	+ c
50		fmove.x	fp0,fp6	xn=xn1-yn1+cx;
51		fmove.x	fp4,fp1	fp4 = fp1 = xn1
52		fsub.x	fp5,fp1	fp1 = xn1-yn1 (= fp5)
53		fadd.d	(8+l76,a7),fp1	+ cx
54		fmove.d	fp1,_xn	
55		fmove.x	fp1,fp3	xn1=xn*xn;
56		fmul.x	fp1,fp3	
57		fmove.d	fp3,_xn1	
58		fmove.x	fp0,fp2	yn1=yn*yn;
59		fmul.x	fp0,fp2	
60		fmove.d	fp2,_yn1	
61		move.w	d4,d0	i--;
62		subq.w	#1,d0	*=> Lines 61+62 get "fused" (very*
63		move.w	d0,d4	*fast)*[36]
64		tst.w	d0	
65		beq	l60	i>0? (test while condition again)
66		fmove.x	fp3,fp4	fp3 = xn1
67		fmove.x	fp2,fp5	fp2 = yn1

36 For more details have a look at: http://www.apollo-core.com/knowledge.php?b=4&note=21090

68		fmove.x	fp2,fp0	
69		fadd.x	fp3,fp0	fp0 = xn1+yn1
70		fcmp.d	#$4010000000000000,fp0	xn1+yn1<=4? (tst while cond. again)
71		fble	l52	
72	l60			
73		move.w	d4,_i	
74		fmove.d	fp6,_yn	
75	l24			
76		moveq	#0,d2	i%256
77		move.w	_i,d2	
78		move.l	#256,-(a7)	
79		move.l	d2,-(a7)	
80		public	__ldivs	
81		jsr	__ldivs	
82		addq.w	#8,a7	
83		move.l	d1,d0	
84		move.l	d0,d2	i%256 = D2
85		moveq	#0,d1	
86		move.w	_y,d1	y = D1
87		moveq	#0,d0	
88		move.w	_x,d0	x = D0
89		jsr	_Put8BitPixel	Put8BitPixel(x,y,i%256);
90		moveq	#1,d0	=> *See comment lines 62+63 (Fusing)*
91		add.w	_x,d0	x++
92		move.w	d0,_x	
93		cmp.w	_resx,d0	x<resx?
94		bcs	l55	
95	l59			
96		moveq	#1,d0	y++
97		add.w	_y,d0	=> *See comment lines 62+63 (Fusing)*
98		move.w	d0,_y	y<resy?
99		cmp.w	_resy,d0	
100		bcs	l54	

In the given code, we can identify three parts that can be optimized:

- BLUE: We will (again) address the inefficient function call by replacing it with an AND or MOVE instruction (lines 76-84, *brute_force0_vbcc_opt1.s*).
- ORANGE: These sections consist of sequences of 2-4 instructions that can be optimized by more efficient alternatives (*brute_force0_vbcc_opt2.s*).
- PURPLE: This represents the main loop (l52), which requires further discussion regarding potential optimizations (*brute_force0_vbcc_opt3.s*).

The blue part is now trivial (we did that already in the previous example). So, let's focus on the orange sections and examine the specific sequences (lines 11-18 and 25-26):

Original		Optimization
11	move.l #$00000000,_xn	fmove.s #0,fp0
12	move.l #$00000000,4+_xn	fmove.d fp0,_xn
13	move.l 4+_xn,4+_xn1	fmove.d fp0,_yn

14	move.l _xn,_xn1	fmove.d fp0,_xn1
15	move.l #$00000000,_yn	fmove.d fp0,_yn1
16	move.l #$00000000,4+_yn	
17	move.l 4+_yn,4+_yn1	
18	move.l _yn,_yn1	
25	move.l (4+l76,a7),4+_cy	fmove.d fp0, _cy
26	move.l (0+l76,a7),_cy	

In the given code snippet, VBCC is working with *double* variables (64 bit = 8 bytes). Surprisingly, VBCC uses two MOVE.L instructions for each double, even though there is the FMOVE.D instruction available in the FPU, which can perform the same operation in a single instruction. Additionally, the FMOVE.**S** (= single or 32 bit) instruction can be used to set FP0 to 0, as it results in a shorter opcode.
A similar situation arises in the following lines (19-21 and 27-29):

	Original	Optimization
27	moveq #0,d7	fmove.w _y,fp0
28	move.w _y,d7	*=> Lines 28+27 get fused (very*
29	fmove.l d7,fp0	*fast), but fmove.w is faster.*[37]
61	moveq #0,d7	fmove.w _x,fp0
62	move.w _x,d7	*=> Lines 61+62 get fused (very*
63	fmove.l d7,fp0	*fast), but fmove.w is faster.*

In the given code snippet, it seems that VBCC performs a MOVEQ #0,d0 instruction to ensure that the entire 32-bit register is set to 0 before moving the 16-bit values. However, in this context, it is unnecessary because the values need to be stored in the floating-point register FP0. To optimize the code further, we can make two smaller improvements in lines 41, 47, and 70:

	Original	Optimization
41	fcmp.d #$4010000000000000,fp0	fcmp.s #4,fp0
47	fmul.d #$4000000000000000,fp0	fmul.s #2,fp0 (or: fadd fp0,fp0)

Again, the .S suffix gives us a slightly smaller opcode. For the multiplication by 2, we could optionally use an ADD instruction on the register (which is also very fast).
And finally, the loop counter part (with variable *i* for the iterations that is decremented) in lines 61-65:

	Original	Optimization
61	move.w d4,d0	subq #1,d4
62	subq.w #1,d0	-
63	move.w d0,d4	-
64	tst.w d0	-
65	beq l60	beq l60

37 For more details have a look at: http://www.apollo-core.com/knowledge.php?b=4&note=21090

Moving the value from D4 to D0 and then doing a TST.W is actually superfluous. The TST instruction is not necessary because the conditional flags are already set by the SUBQ instruction. As we’ve always said, loops have the highest potential for optimizations, and on the m68k this is particularly true for loops that count downwards (like here). Normally, whenever possible, we should use something like *DBRA D4,label*, because the DBRA does a SUB and and Bcc at the same time! Unfortunately, it is not that easy to integrate the DBRA here, because two conditions have to be checked, so that we will leave it as it is.
Finally, let’s have a look at the innermost loop situated between the label l52 (line 45) and the FBLE l52 (line 71). VBCC does several memory accesses here, in order to keep all the variables updated. But a closer look reveals that lines 57 and 60 (memory accesses for *_xn1* and *_yn1*) are completely useless (we can completely eliminate them). There is no MOVE instruction from these variables to a destination, so it is not necessary to keep these variables in sync with the registers, which are faster.
So let’s see how our three optimized versions perform:

Optimization	Time (seconds)	Percentage
none	41.77	100%
1	41.28	99%
2	40.06	96%
3	35.31	85%

As we can see, the first two optimizations only result in a very small speed increase. This might seem disappointing because in our first example, it was more than twice the speed. However, the difference is that the modula operation was an substantial part of the loop in the first example, whereas here it is only one of many operations. The best speed increase (11%) comes from that last optimization which actually removes two lines of code. This highlights again the beauty of assembly programming: we have to find the exact point with the most potential for optimizations, and even small changes can lead to substantial improvements in performance.

Despite these optimizations, our code is still about 5.4 seconds slower than the code produced by GCC 2.95.3. So, the question is, what is better in GCC?

	brute_force0_gcc.s	Comments
1	clrw _y	
2	tstw _resy	← Lines 2-3 can be deleted without any
3	jeq L22	harm (probably and “early exit”
4	clrl d7	optimization)
5	clrl d6	
6	.even	GCC clears several registers “in a row”
7	L24:	lines 4-5 and 12-14: very interesting
8	clrw _x	for a CPU like the 68080 which can
9	tstw _resx	execute instructions in parallel!
10	jeq L23	
11	fmoved #0r0,fp7	
12	clrl d5	
13	clrl d4	
14	clrl d3	
15	.even	

16	L28:	
17	fmoved fp7,_xn	We see that GCC uses here the FMOVE.D
18	fmoved fp7,_xn1	instruction to initialize _xn, _yn,
19	fmoved fp7,_yn	_xn1, _yn1 (VBCC didn't use them)
20	fmoved fp7,_yn1	
21	movew _y,d5	
22	fdmoved _stepi,fp1	
23	fdmull d5,fp1	
24	fdaddd _mini,fp1	
25	fmoved fp1,_cy	
26	movew _x,d3	
27	fdmoved _stepr,fp0	
28	fdmull d3,fp0	
29	fdaddd _minr,fp0	
30	fmoved fp0,_cx	
31	movew _MaxIter,_i	
32	movew _MaxIter,d0	
33	lea _Put8BitPixel,a0	This is curious: GCC loads the address
34	jeq L30	of the function Put8BitPixel into
35	fdmovex fp1,fp6	register A0 and then uses it to branch
36	fdmovex fp0,fp5	to that function in line 77!
37	fdmovex fp7,fp1	
38	fdmovex fp1,fp4	
39	fdmovex fp1,fp3	
40	fdmovex fp1,fp2	
41	movew d0,d1	
42	.even	
43	L31:	
44	fdaddx fp1,fp1	This is the main (most inner loop) of
45	fdmulx fp1,fp4	the calculation: it is very well
46	fdaddx fp6,fp4	optimized! (Everything is done in
47	fdmovex fp3,fp1	registers.)
48	fdsubx fp2,fp1	
49	fdaddx fp5,fp1	
50	fdmovex fp1,fp3	
51	fdmulx fp3,fp3	
52	fdmovex fp4,fp2	
53	fdmulx fp2,fp2	
54	movew d1,d0	
55	subqw #1,d1	
56	cmpw #1,d0	
57	jeq L47	
58	fdmovex fp3,fp0	
59	fdaddx fp2,fp0	
60	fcmpd #0r4,fp0	
61	fjle L31	
62	L47:	
63	movew d1,_i	Here, GCC updates the main variables.
64	fmoved fp2,_yn1	(And as with VBCC this is completely
65	fmoved fp3,_xn1	useless … :)
66	fmoved fp4,_yn	

67	fmoved fp1,_xn	
68	L30:	
69	clrw d0	How does GCC do the modulo %256? It
70	moveb _i+1,d0	simply moves the lower byte of word i –
71	movew d0,d7	that is clever!
72	movew _y,d4	Preparation for Put8BitPixel call
73	movew _x,d6	(D0/D1/D2) is somewhat inefficient
74	movel d7,d2	(because certain values are moved twice,
75	movel d4,d1	e.g. _i => d0 => d2)
76	movel d6,d0	
77	jbsr a0@	This is the curious call of Put8BitPixel
78	movew _x,d0	via address register A0! (But actually,
79	movew d0,d1	a direct call is faster!)
80	addqw #1,d1	
81	movew d1,_x	
82	addqw #1,d0	
83	cmpw _resx,d0	
84	jcs L28	
85	L23:	
86	movew _y,d0	Curious: GCC copies _y into two
87	movew d0,d2	registers (d0/d1) and then does two adds
88	addqw #1,d2	(lines 88/90) and finally keeps the
89	movew d2,_y	memory variable _y updated (line 89).
90	addqw #1,d0	The same is true in lines 78ff for _x.
91	cmpw _resy,d0	Can all that be efficient?!
92	jcs L24	

As we can observe, the innermost loop (after label L31, marked in orange) is particularly well optimized: everything is done in registers (no memory accesses). Nonetheless, let's see if we can do better by modifying certain parts of the code one by one:

1. Let's replace the indirect JBSR A0@ with a direct call JBSR *_Put8BitPixel*. This means at the same time that we don't need the preceeding LEA _Put8BitPixel,A0 in line 33 (*brute_force0_gcc_opt1.s*).
2. Eliminate everything from the source code that is not strictly necessary, i.e. all the lines (2-3, 9-10, 64-67) marked in green (*brute_force0_gcc_opt2.s*).
3. Make the function call to *_Put8BitPixel* more efficient by writing the values directly to the needed registers when possible. In addition we, will avoid subsequent memory accesses by rearranging the order of the instructions (*brute_force0_gcc_opt3.s*):

Original		Modified
69	clrw d0	movew _y,d1 ; place mem access
70	moveb _i+1,d0	clrl d2 ; here (avoid two
71	movew d0,d7	moveb _i+1,d2 ; subsequent mem
72	movew _y,d4	- ; accesses)
73	movew _x,d6	movew _x,d6 ; backup _x
74	movel d7,d2	-
75	movel d4,d1	-
76	movel d6,d0	movel d6,d0
77	jbsr a0@	jbsr _Put8BitPixel
78	movew _x,d0	movew d6,d0 ; restore _x

4. Finally, let's see if we can be more efficient for the code that increments and compares the variables *_x* and *_y* (*brute_force0_gcc_opt4.s*):

Original		Modified
86	movew _y,d0	addqw #1,_y
87	movew d0,d1	movew _y,d0
88	addqw #1,d1	cmpw _resx,d0
89	movew d1,_y	jcs L26
90	addqw #1,d0	-
91	cmpw _resy,d0	=> Do the same also with _x
92	jcs L26	(lines 78ff)

After applying these optimizations, the performance of the code improves. Here are the results:

Optimization	Time (seconds)	Percentage
none	29.91	100%
1	29.87	99.86%
2	29.86	99.83%
3	29.83	99.73%
4	29.82	99.70%

These results show that even with multiple optimizations, the improvements in performance are minimal, ranging from 0.13% to 0.30%. It indicates that the original code generated by GCC 2.95.3 is already highly optimized, especially in the innermost loop where all the values are kept in registers. It becomes challenging to further enhance the performance beyond what has already been achieved by the compiler.

Analysis 3: *example5/boundary_trace2*

The last example we are going to analyze is the fastest in the whole series: *example5/boundary_trace2*, which ran in 3.04 seconds in the first part of the tutorial. Using GCC 6.5 the same example now runs at 2.70 seconds (which already represents a nice speed increase of 11%). But can we go even faster? Unfortunately, the main code section of this example spans from lines 335-1038 (= 703 lines) which is simply too much to discuss in detail here. However, let's take a look at some specific parts of it. As always, the parts that offer most potential for optimizations are those called many times, such as inner loops. In our case that corresponds to this part of the C-code:

```
 /* (2) process the queue (which is actually a ring buffer) */
flag=0;
while(QueueTail != QueueHead) {
   if(QueueHead <= QueueTail || ++flag & 3) {
      p = Queue[QueueTail++];
      if(QueueTail == QueueSize) QueueTail=0;
   } else p = Queue[--QueueHead];
   Scan(p);
}
```

Which corresponds to the lines 449-892 (= 443 lines) of *example5/asm/boundary_trace2_gcc6.s*. So, still a lot of code. But let’s see if we can spot some snippets that offer room for optimizations. There are, for example, indirect function calls that we can replace with direct ones like in the previous example:

boundary_trace2_gcc6.s		optimization
[38]500	jsr (a3)	jsr _SingleIterateAsm
506	jsr (a4)	jsr _Put8BitPixel
297	jsr (a2)	jsr _rand

Another optimization opportunity is eliminating unnecessary instructions, such as CLR.L in the following code block (and since similar blocks come up several times we are going to apply the same modification everywhere):

501	move.l d0,d4	move.l d0,d4
502	clr.l d2	and.l #255,d4 ; use and.l to clear d4
503	move.w d0,d2	move.w d4,d2
504	move.l d6,d1	move.l d6,d1
505	move.l d3,d0	move.l d3,d0
505	jsr _Put8BitPixel	jsr _Put8BitPixel
506	or.b #1,([_Done],a2.l)	or.b #1,([_Done],a2.l)
507	move.b d4,([_Data],a2.l)	move.b d4,([_Data],a2.l)
508	and.l #255,d4	- ; moved to line 502

Please note that the preceding function, *_SingleIterateAsm*, and the following *_Put8BitPixel* function present a highly unideal register allocation. *_SingleIterateAsm* returns iterations in *D0*, and this value is needed in *D2* for *_Put8BitPixel*. Additionally, the *x* and *y* values, present in registers *D6* and *D3*, need to be copied to *D0* and *D1*. This could certainly be optimized by adapting these 2 functions (for example, *_SingleIterateAsm* could return iterations in D2, allowing direct use by *_Put8BitPixel*). However, we will not make those optimizations because they would involve modifications at the source code level, and our focus here is solely on assembly optimizations.
Furthermore, we can eliminate an AND.L instruction and optimize the size of the FMUL instruction in the following (and similar) blocks:

529	move.l a5,d1	move.l a5,d1 ; a5 = p
530	divul.l d7,d7:d1	divul.l d7,d7:d1 ; d7 = resx
531	and.l #65535,d1	
532	fdmove.d _stepi,fp1	fdmove.d _stepi,fp1
533	fdmul.l d1,fp1	fdmul.**w** d1,fp1 ; d1 = y (WORD)
534	fdmove.d _stepr,fp0	fdmove.d _stepr,fp0
535	fdmul.l d7,fp0	fdmul.**w** d7,fp0 ; d2 = x (WORD)

In the following sequence, we can avoid a memory access for *_QueueHead* and compare directly to a register:

821	move.l _QueueHead,d0	move.l _QueueHead,d0
822	move.l d0,d1	move.l d0,d1
823	addq.l #1,d1	addq.l #1,d1
824	move.l d1,_QueueHead	move.l d1,_QueueHead
825	move.l a0,([_Queue],d0.l*4)	move.l a0,([_Queue],d0.l*4)

38 These indirect calls occur in other lines also – for our optimization we have replaced them everywhere. We also eliminated all corresponding LEA *function,register* instructions.

826	`move.l _QueueHead,d0`	`-`	`; no mem access`
827	`cmp.l _QueueSize,d0`	`cmp.l _QueueSize,d1`	`; compare to d1`

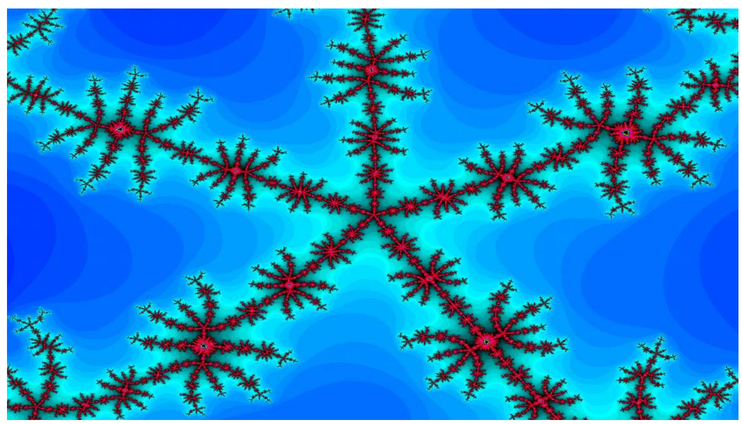
GCC 6.5 does other memory accesses that are not necessary. In the following example, GCC seems to run out of registers and decides to store the variable *y* temporarily on the stack (lines 627 and 632). But we can temporarily hijack the register *D6* instead:

627	`move.l d1,(56,sp)`	`move.l d1,d6`	`; d6 = y (temp)`
628	`jsr _SingleIterateAsm`	`jsr _SingleIterateAsm`	
629	`move.w d0,d6`	`move.l d6,d1`	`; d6 => d1 (for`
630	`and.l #255,d6`	`move.w d0,d6`	`; _Put8BitPixel)`
631	`move.w d6,d2`	`and.l #255,d6`	
632	`move.l (56,sp),d1`	`move.w d6,d2`	
633	`move.l d7,d0`	`move.l d7,d0`	
634	`jsr _Put8BitPixel`	`jsr _Put8BitPixel`	

From this point on, we could continue in the same style – eliminating instructions here and there – but the fact is that these optimizations do not significantly improve performance. In terms of numbers, all the modifications we've presented result in an execution time of 2.69 or 2.68 seconds, which is only a slight improvement compared to the 2.70 seconds generated by the compiler. Therefore, we come to the conclusion (and truly believe) that GCC 6.5, just like its predecessor GCC 2.95.3, generates very efficient code!

Chapter 6 – What else … ?

To make this second part of the tutorial more complete, we have decided to include the following additions (located in the *Icons/More* drawer).



Mandelbrot in FullHD calculated with FlashMandelVE.

FlashMandelVE – Vamped Edition

Without any doubt, FlashMandel is one of the best fractal programs on the Amiga! In was published in 2001 by Dino Papararo and is available via Aminet[39]. The author included the source code, allowing for the creation of a "Vamped Edition" of the program. This special version utilizes features specific to the Vampire that we presented in the first part of the tutorial, such as parallelizing calculations and utilizing 3-operand instructions. As a result, this version runs approximately 30% faster. The Vamped Edition and the corresponding readme can be found on Github[40]. Please note that this version has to be considered BETA and does not yet work very well on ApolloOS due to incompatibilities.

Syntax Highlighting for Annotate

We have also included an XML file that provides syntax highlighting definitions for Annotate, a free text editor available on Aminet[41] (and included in Coffin). The provided definitions are specifically designed for C and Assembly code and work best on ApolloOS, though they can be adapted for use with Coffin. To utilize the syntax highlighting, you can either copy the XML file to the home drawer of Annotate or add it to ENVARC.

39 https://aminet.net/package/gfx/fract/FlashMandel
40 https://github.com/r3dbug/FlashMandelVE
41 https://aminet.net/package/text/edit/Annotate

Chapter 7 – Conclusions

Verdict? Better or worse? It depends ...

We have conducted various comparisons and applied rankings for different criteria. Does this mean that we are now going to recommend one compiler and “ban” all the others? No! We truly believe that each of the tested compilers has it strengths and its weaknesses, and no single compiler is perfect in all the areas we tested.

So, the answer to the question “What compiler would you recommend” is: It depends!

For example: If you want do develop on the Vampire itself, your choices will be among SAS/C, VBCC and GCC 2.95.3. However, if you are willing to develop your programs on another (“non-amigan” - heresy!:) machine, then GCC 6.5 (or even a newer version) or VBCC will be logical options. Similarly, when it comes to the C standard, if you want to use C++, you definitely can’t go with SAS/C nor VBCC. However, if you prioritize and wish to develop directly on the Vampire, these compilers are better options that GCC 2.95.3.

Therefore, as a final conclusion, we will provide a “synoptic summary” which highlights the advantages (green) and disadvantages (red) offered by each compiler. Ultimately, the decision will be yours!

Synoptic Summary

SAS/C 6.58	
• native compiler • very stable • very fast compiling • excellent compatibility (with AmigaOS) • relatively small executables	• code quality (not always good) • proprietary compiler • only C89 • not actively developped any more • recompiling difficult
GCC 2.95.3	
• native compiler (+ evtl. cross-compiler) • porting (*nix programs) • excellent code quality (best!) • relatively small executables • some C++98 • easy tuning / recompiling	• rather unstable • rather slow compiling • compatibility (with AmigaOS) • not actively developped (but source code available)
VBCC 0.908	
• both native compiler + cross-compiler • reasonably stable • compatibility (AmigaOS) • free compiler (for m68k AmigaOS) • actively developped (VBCC & VASM) • compiling for PPC possible (AmigaOS 4 and Morphos) • easy tuning / recompiling	• slow compiling • code quality • only C99
GCC 6.5	
• very fast compiling • very stable • porting (*nix programs) • very good code quality • free compiler • actively developped • C++11 & access to modern *nix libraries • easy tuning / recompiling	• cross-compiler • compatibility (with AmigaOS) • bigger executables (escpecially when used with ixemul)

The decision will also be different if you want to write a program entirely in C (then a "good" compiler is important) or – as we've done in this tutorial – use the C-Compiler only as the main structure of the program and write the time-critical parts in assembly (then you don't necessarily need the fastest C-compiler).

Final Word

Never forget: No compiler will ever be as good as handwritten assembly! That's why, in the first part already, we included the last example: *teaser/boundary_trace*. All the core parts of this example have been hand-tuned in assembly and it executes in 2.41 seconds (on a 12x core). This is still 11% faster than the fastest executable generated by a compiler! And even the "fastest example" (2.70 seconds) produced by GCC 6.5 was only possible because the compiler could rely on many optimized assembly functions that we constantly added during the tutorial.

To conclude:
Use the best of both worlds – C/C++ for the main structure of the program and hand-tuned assembly for the time-critical parts.

Special Thanks

… go to Tim (and his cats, Rosie and Cooper) for taking the time to reread this document!

*
*   *

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

Fribourg / Switzerland, June 2023
by RedBug

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