CS4202 P02 - Compiler Optimisations

150015673

16 November 2018

Contents

1 Task

The aim of this practical was to examine and evaluate 4-5 GCC optimisations using a personally developed test-suite in order to gain a better understanding of computer architecture concepts and the effect of compiler optimisations.

2 Setup

- CPU: Intel Core i5-6500 @ 3.20GHz
- RAM: 8GiB DDR4
- OS: Fedora 28 (64-bit, kernel `4.18.16-200.fc28.x86_64`)
- GCC: `8.2.1 20181105`

3 Compiler Optimisations Chosen

I chose the optimisations below because they each are fairly simple, yet exploit different things, e.g. sequential execution or vector registers. I also chose them because although they are simple, they should allow for some speedup, e.g. loop unrolling should speed up a loop-heavy program, and vectorisation should speed up array-heavy programs.

Optimisation flags require a value for the '`-O`' flag in order to work[1]. I chose to use `-O` to avoid any additional optimisations the compiler had.

3.1 Loop unrolling (`-funroll-loops`)

Branching and branch-prediction is expensive. Since loops repeat for a given condition, e.g. `i < 10`, we can limit the number of times that condition has to be checked by unrolling the loop. This leads to speedups both because of the fewer branches, but also because there are more sequential steps to execute, which is something processors are good at. The main downside is that the code size increases.

3.2 Tail recursion (`-foptimize-sibling-calls`)

Tail-recursive functions differ from recursive functions by doing the accumulation of the recursion as part of the parameters, instead of when the recursive call returns. This means that the compiler can replace the tail-recursive call with a `GOTO` statement or similar, thereby saving stack space since there is no need for recursively calling the function.

3.3 Loop interchange (`-floop-interchange`)

Loop interchange is swapping the inner and outer loops around. Doing this can improve exploitation of spatial locality and might also expose other optimisations, e.g. vectorisation.

[1](*Using the GNU Compiler Collection (GCC): Optimize Options*, 2018)

3.4 Tree vectorisation (`-ftree-vectorize`)

GCC uses trees as an internal representation of the compiled programs. This flag allows GCC to use SIMD vector instructions to optimise the program code. This should speed up code which does a lot of array manipulation.

4 Test-Suite

I wrote a variety of programs. Not all of them target the same optimisation, and it is possible that some of them cannot be optimised using one of the chosen optimisations. In order to achieve a measurable run-time, I defined the '`loops.h`' file which contains the number of iteration each program should run the function to measure. Apart from the matrix multiplication, I set this to $100,000$ iterations. To ensure that the compiler would not simply optimise away the program, the output is printed in programs where there is one output, and an element of the array is printed in programs whose result is an array.

4.1 `2d_mmult.c`

A simple, naïve 2d-array matrix multiplication of 1024×1024 matrices. This program should be optimisable through loop unrolling, loop interchange, and vectorisation. The dimension of $1024 = 2^{10}$ was chosen with the idea that it might fit nicer into vector registers.
Since matrix multiplication of matrices this large is already computationally expensive, only 5 iterations were used instead of the default $100,000$.

4.2 `1d_mmult.c`

A simple, naïve matrix multiplication of 1024×1024 matrices, this time using 1d-arrays instead of 2d-arrays. This was done to examine if the compiler is better at optimising 1d-arrays than 2d-arrays.
Since matrix multiplication of matrices this large is already computationally expensive, only 5 iterations were used instead of the default $100,000$.

4.3 `arr_incr.c`

Increments all the numbers in an array by a given constant. The idea of this program was mainly to target vectorisation although loop unrolling should improve it as well.

4.4 `bst.c`

A small, simple Binary Search Tree (BST) implementation. Values are inserted into the BST and it is traversed in-order. This was done to target recursive function optimisations.

4.5 `mymap.c`

An implementation of a 'map' function over integers. This program's purpose was to examine if/how well the compiler can optimise function pointers.

4.6 `ptest.c`

A simple primality test[2]. The `while` loop might be unrollable. However, I mainly chose to include a primality test because it should be difficult to optimise automatically and so I wanted to see if the compiler could optimise it.

4.7 `reduce.c`

The 'reduction' of an array, i.e. the sum of all its elements. The array in this file is an integer array with $2^{14} = 16384$ elements. This size was, similar to the matrices' dimension, chosen to potentially be easily divisible into vector registers.

4.8 `sum.c`

Sums all the numbers from 1 to $10,000$ inclusive. The main optimisation target of this program was loop unrolling.

4.9 `trfact.c`

Calculates the factorial of the given number using tail-recursion. To achieve the maximum number of calls 20!, the largest factorial that fits in an `unsigned long long`, was used. This program targets tail-recursive optimisations.

5 Manually Implemented Optimisations

I manually implemented two optimisations: inline functions and loop unrolling. The source for these can be found in `man-inline-src` and `man-funroll-src` respectively. To measure how effective the manual implementations were, I compiled the files without any optimisations.

5.1 Inline Functions

All code that could be moved to `main` was moved there. For the tail-recursive factorial, I also transformed the tail-recursive function to a loop. This is an additional optimisation, however it was something I did because it seemed logical: you can achieve the tail-recursive function inline by converting it into a loop. Apart from this, and the BST implementation, all function code was moved to `main` where possible.
The advantage of this should be that there are fewer operations due to avoiding the overhead of calling and returning from functions. The disadvantage is that the code might be less legible, and will likely not adhere to the "Don't Repeat Yourself" (DRY) principle.

5.2 Loop Unrolling

Where applicable, I manually unrolled loops in my programs using the generalised version of Duff's Device[3]. This increases the code size and is almost the exact opposite of DRY, but it might increase execution speed due to there being less branching. The code size and readability suffers especially heavily when the two matrix multiplication programs are unrolled.

[2] taken from (*Primality test - Wikipedia*, 2018)
[3] (Holly, 2005)

6 Data Collection

All programs were compiled and run on the scratch space of the PC used, to remove any delay there might be by having the files on the synchronised user drive. A slight variation in timing was observed through a couple of manual runs of the programs. To compensate for this variation, each program was run 10 times. The data from each flag was put in separate CSV-files with the program name indicating whether the flag was en- or disabled.

For timing I initially used `/usr/bin/time`. However, this times in lower resolution than the built-in `time` shell keyword. Therefore, I decided to use the latter. To output to CSV, `TIMEFORMAT` was set to "`${prog_name},%4U`". Additionally, displaying outputs takes time and completely clutters the terminal when running $100,000$ iterations of the same function. To combat this, stdout was piped to `/dev/null`. The `benchmark.sh` script takes around 45-60 minutes to complete.

7 Results and Analysis

All plots include error bars representing the standard error of the mean (SEM).

7.1 Compiler Optimisations

7.1.1 Loop Interchange

For loop interchange, the main difference was observed in terms of matrix multiplication. This was expected, as loop interchange is a common matrix multiplication optimisation. For the other programs, it had no effect[4].

[4] see plot B.1: 13

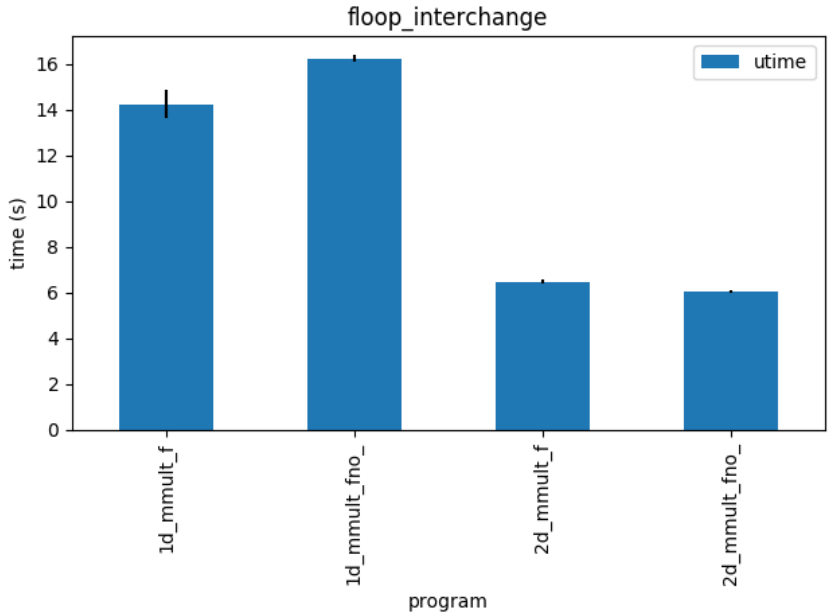


Figure 1: Loop-interchange on matrix multiplication.

For the 2d-array matrix multiplication, the time seems to have increased slightly with the flag enabled. However, this seems to be a measurement error: looking at the assembler code reveals that the two files are identical, meaning that the compiler did not find a way to interchange the loops.

2d_mmult_fno_loop_interchange.s

```
mmult:
.LFB25:
        .cfi_startproc
        testl   %ecx, %ecx
        jle     .L17
        pushq   %r12
        .cfi_def_cfa_offset 16
        .cfi_offset 12, -16
        pushq   %rbp
        .cfi_def_cfa_offset 24
        .cfi_offset 6, -24
        pushq   %rbx
        .cfi_def_cfa_offset 32
        .cfi_offset 3, -32
        movq    %rdi, %r10
        movq    %rsi, %r11
        leal    -1(%rcx), %r9d
        leaq    8(%rdi,%r9,8), %r12
        leaq    4(,%r9,4), %rbp
        movl    $0, %ebx
        jmp     .L11
.L15:
        movq    %rcx, %rax
.L12:
        movq    (%rdx,%rax,8), %rcx
        movl    (%rcx,%rdi), %ecx
        imull   (%r8,%rax,4), %ecx
        addl    %ecx, %esi
        leaq    1(%rax), %rcx
        cmpq    %r9, %rax
        jne     .L15
        movq    (%r10), %rax
        movl    %esi, (%rax,%rdi)
        addq    $4, %rdi
        cmpq    %rbp, %rdi
        je      .L13
.L14:
        movq    (%r11), %r8
        movq    %rbx, %rax
        movl    $0, %esi
        jmp     .L12
.L13:
        addq    $8, %r10
        addq    $8, %r11
        cmpq    %r12, %r10
        je      .L9
.L11:
        movl    $0, %edi
        jmp     .L14
.L9:
        popq    %rbx
        .cfi_def_cfa_offset 24
```

2d_mmult_floop_interchange.s

```
mmult:
.LFB25:
        .cfi_startproc
        testl   %ecx, %ecx
        jle     .L17
        pushq   %r12
        .cfi_def_cfa_offset 16
        .cfi_offset 12, -16
        pushq   %rbp
        .cfi_def_cfa_offset 24
        .cfi_offset 6, -24
        pushq   %rbx
        .cfi_def_cfa_offset 32
        .cfi_offset 3, -32
        movq    %rdi, %r10
        movq    %rsi, %r11
        leal    -1(%rcx), %r9d
        leaq    8(%rdi,%r9,8), %r12
        leaq    4(,%r9,4), %rbp
        movl    $0, %ebx
        jmp     .L11
.L15:
        movq    %rcx, %rax
.L12:
        movq    (%rdx,%rax,8), %rcx
        movl    (%rcx,%rdi), %ecx
        imull   (%r8,%rax,4), %ecx
        addl    %ecx, %esi
        leaq    1(%rax), %rcx
        cmpq    %r9, %rax
        jne     .L15
        movq    (%r10), %rax
        movl    %esi, (%rax,%rdi)
        addq    $4, %rdi
        cmpq    %rbp, %rdi
        je      .L13
.L14:
        movq    (%r11), %r8
        movq    %rbx, %rax
        movl    $0, %esi
        jmp     .L12
.L13:
        addq    $8, %r10
        addq    $8, %r11
        cmpq    %r12, %r10
        je      .L9
.L11:
        movl    $0, %edi
        jmp     .L14
.L9:
        popq    %rbx
        .cfi_def_cfa_offset 24
```

Figure 2: The assembler code for `2d_mmult` with and without optimisation (right and left respectively) is identical.

This seems to indicate that the compiler does not only find it easier to optimise 1d-arrays, it might find it impossible to optimise 2d-arrays. Looking at vectorisation further strengthens this hypothesis.

7.1.2 Vectorisation

For vectorisation, several programs are sped up. The most drastic improvements are on the `reduce` and `arr_incr` programs where the execution time is almost halved. 1d-matrix multiplication is sped up by $\approx 15\%$[5] and 2d-matrix multiplication is once again unchanged[6], seemingly confirming that the compiler cannot optimise 2d-arrays.

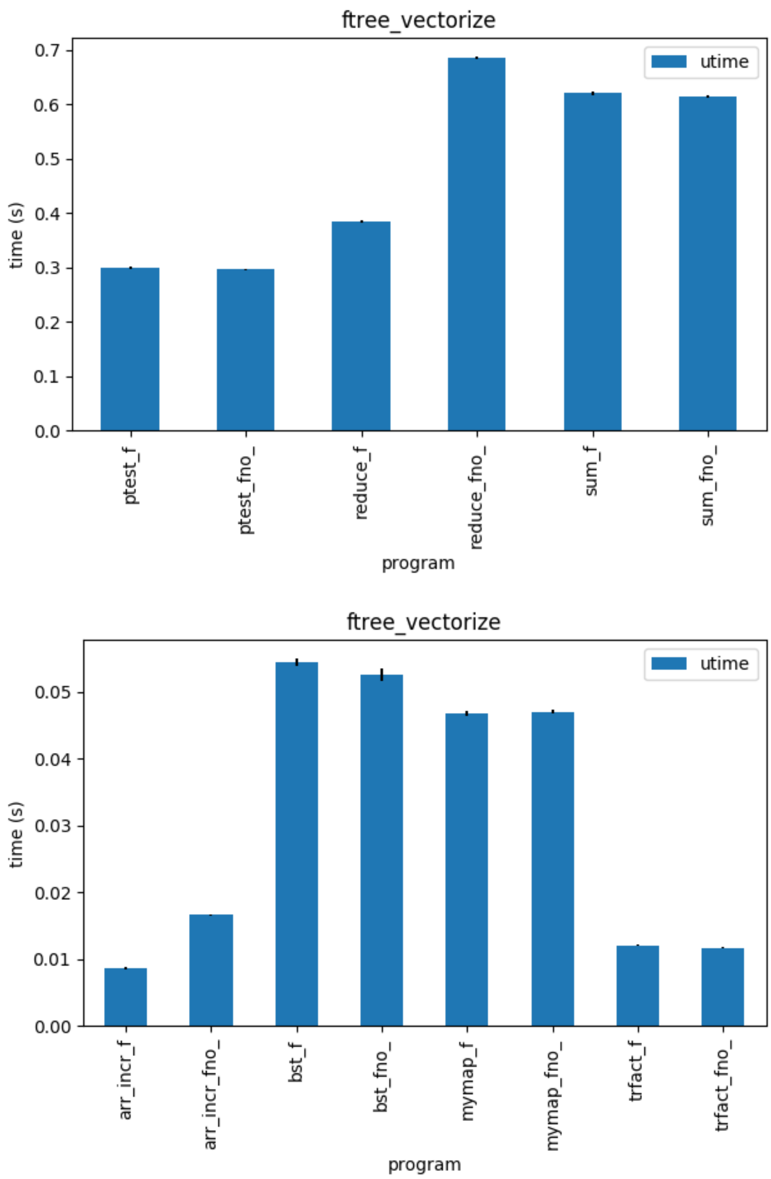


Figure 3: Both `reduce` and `arr_incr` are sped up significantly by vectorisation.

It makes sense that `reduce` and `arr_incr` are the programs on which vectorisation has the greatest effect: `arr_incr` is adding a constant to each element of a vector, and `reduce` can be split

[5] see plot B.1: 14
[6] see assembler A: 10

into multiple vectors which can then be added together until all that remains is a vector register with the total sum in it[7]. When examining vectorisation, I came across an interesting feature of `x86` vector instructions: in both `reduce`, but surprisingly also in `bst`, 0 is derived by using `pxor` on the same register[8] instead of simply using the constant `$0`. By searching online, I found that this achieves greater performance, as the `pxor` is evaluated at the register rename stage[9]. However, as zeroing it is not the bulk of the `bst` program, there is no speed-up to be gained from doing so.

An optimisation which affected similar programs to the ones sped up with vectorisation is loop unrolling.

7.1.3 Loop Unrolling

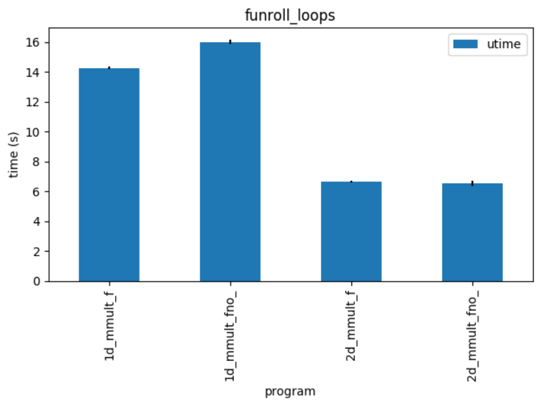
With loop unrolling, the programs that were affected by vectorisation were also affected by loop unrolling, with `sum` and `my_map` being affected as well. This is likely due to the fact that vectorisation targets large arrays, which will typically be iterated over in big loops, hence being suitable for loop unrolling as well. My theory for why `my_map` was optimised here and not through vectorisation is that the compiler cannot know that the pointer to the function won't change and so it does not want to replace it with the function. For `sum` my theory is that vectorising it would require vectors whose elements were the numbers of the loop and that these would be more expensive to construct than the speedup of having them would win back. When comparing the results in Figure 4 and Figure 3, it becomes clear that although loop unrolling helps, it is not as fast as hardware tools like vector registers. Programs that optimised well using vectorisation still optimise with loop-unrolling. However, the performance gain is not as great as with vectorisation. Since the `sum` program is purely loop-based, its execution time is greatly decreased: from $\approx 0.65s$ to $\approx 0.30s$. The program `my_map` does not improve its execution time much. However, this is still an improvement compared to no optimisation with vectorisation.

As expected, the past three optimisations have not affected the recursive programs, i.e. `bst` and `trfact`.

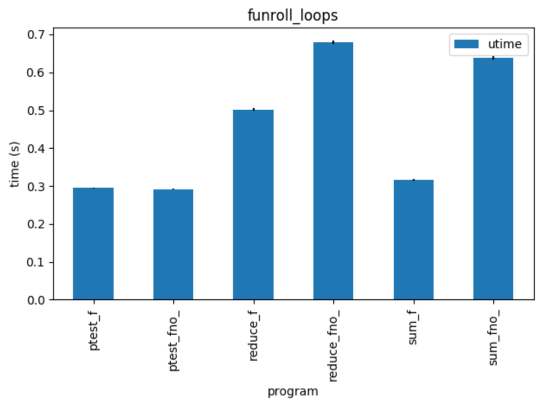
[7] see assembler A: 11

[8] see assembler A: 11 and 12

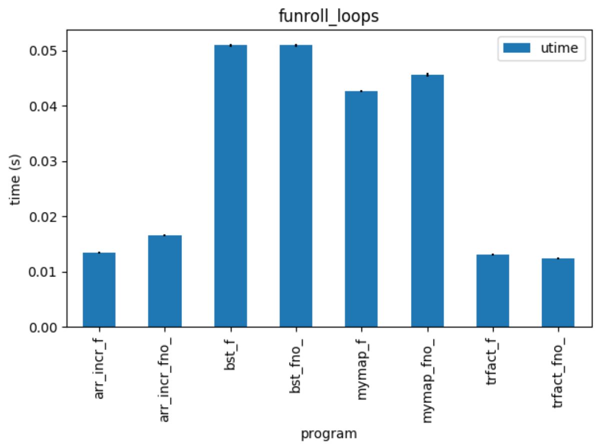
[9] (Fog, 2018) and (Cordes & balajimc55, 2017)



(a)



(b)



(c)

Figure 4: Loop unrolling speeds up the same programs as vectorisation, and some other programs as well.

7.1.4 Sibling and Tail Recursive Calls

When optimising sibling and tail recursive calls, both `bst` and `trfact` improve their execution time. Other programs remain unaffected[10] which was expected as they do not use any recursive calls.

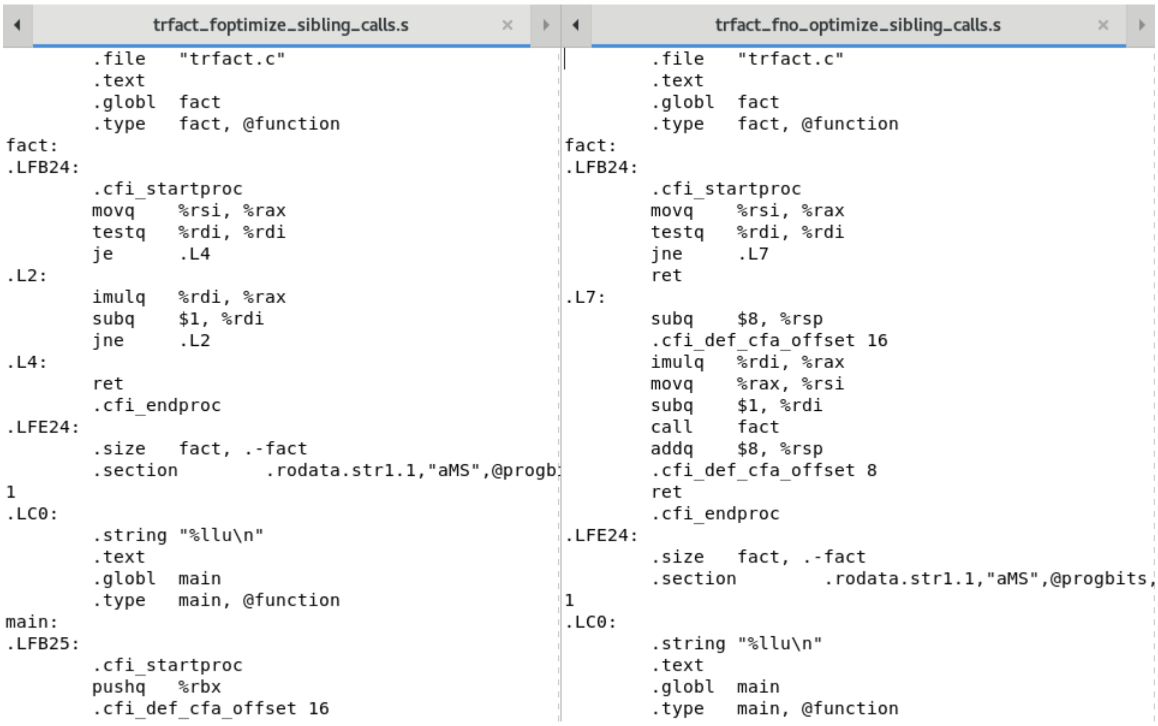


Figure 5: The tail-recursive function in `trfact` has been optimised into a loop by the compiler.

Examining the assembler code reveals that the compiler has optimised `trfact` as expected: the tail-recursive function has been transformed into a loop. Looking at the assembler code for `bst` reveals that the speedup is gained by re-ordering some of the instructions[11]. There are still recursive calls to the function, they just happen later in the function.

[10] see plots in B.1: 15
[11] see assembler A: 12

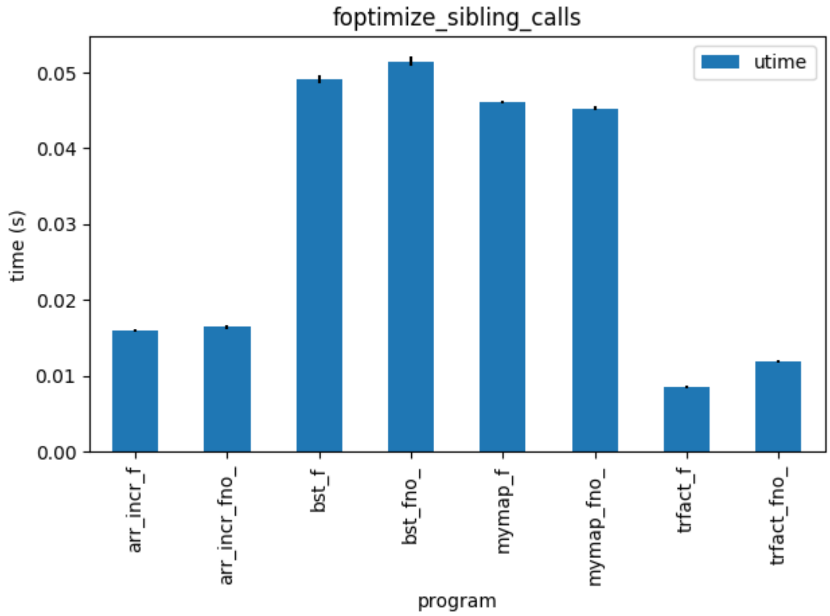


Figure 6: Both `trfact` and `bst` improve their execution time when using `-foptimize-sibling-calls`.

As can be seen in the figure above, the reduction in execution time is not big: less than $0.01s$. Despite this, since no other optimisation has improved the execution time of these programs, I would claim that *a* speedup is still significant. And this indicates that recursive function calls are difficult to optimise.

The compiler optimisations were effective in varying ways. For comparison, a couple of manual optimisations were implemented.

7.2 Manual Optimisations

7.2.1 Inline functions

For inline functions, some programs were more affected than others. Most programs were unaffected[12]. The program that improved the most was `mymap`. This is likely because I knew, contrary to the compiler, that the function pointer would not change and as such could substitute its definition into the main loop.

[12] see plots B.2: 16

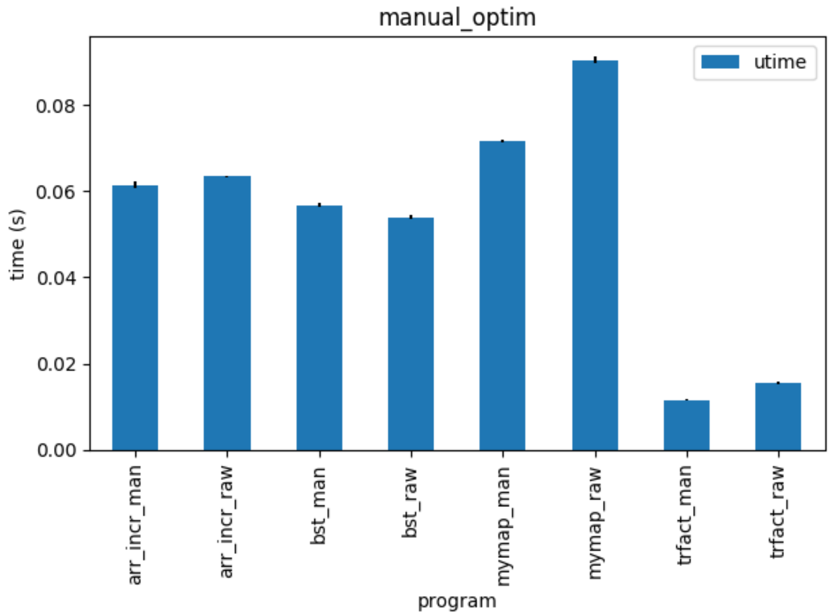


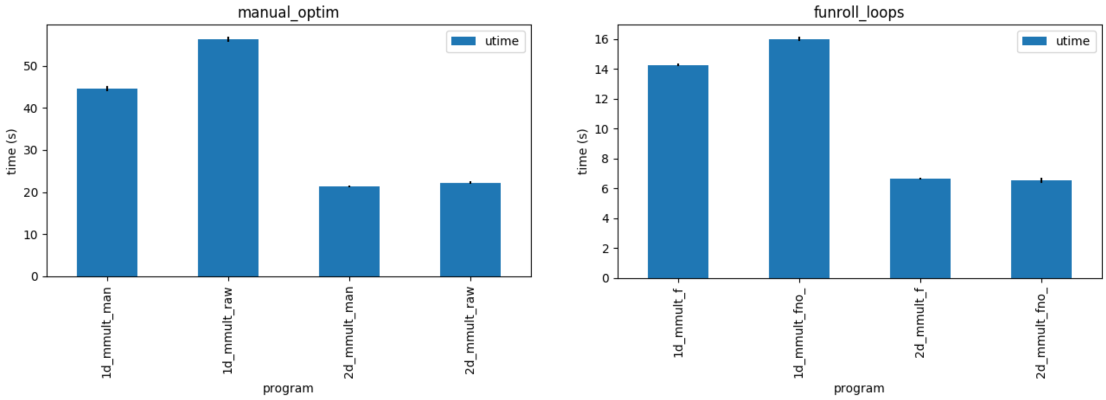
Figure 7: By using knowledge unavailable to the compiler, `mymap` has improved execution time.

The tail-recursive factorial implementation is also improved. However, this is because of the manual 'inline' transformation to a loop. It shows that this improvement is fairly simple to do and does improve performance, but it is not simply moving the function inline.

The readability of the code was not too heavily affected by moving the functions, that could be moved, inline.

7.2.2 Loop Unrolling

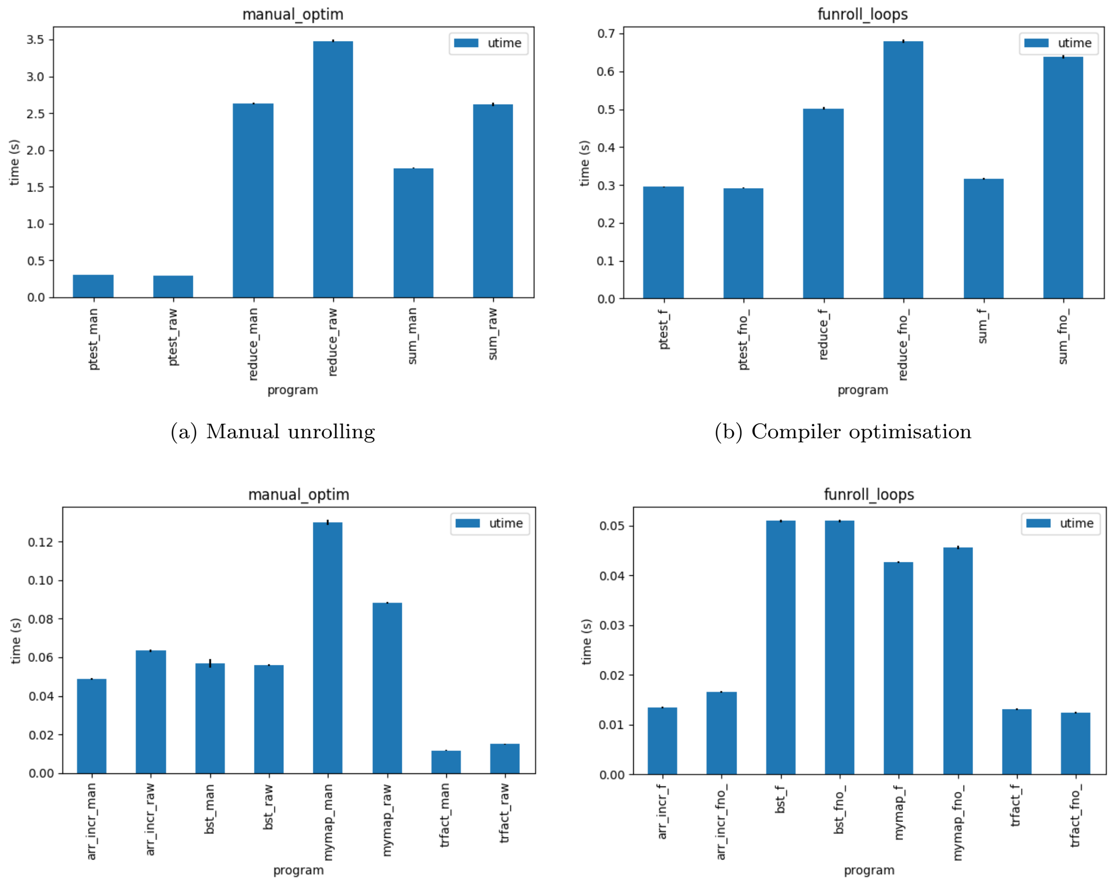
When comparing the results for `mmult`, it initially seems like the manual optimisation was better. However, taking a look at the y-ticks reveals that while the proportional increase might be slightly better, the time-scale has tripled. This is likely due to the compiler unrolling the loops in assembler compared to the unrolling in C which is a higher level and hence might be slower. The C unrolled version is also mostly impossible to read and much longer than the original program. What is interesting is that the 2d-array implementation is still not affected. This suggests that the problem lies elsewhere, perhaps in the memory access of 2d-arrays.



(a) Manual optimisation (b) Compiler optimisation

Figure 8: Comparing manual loop unrolling to compiler unrolling for `mmult`.

The time-scale changing remains true for the other programs as well. And for `mymap`, the manual loop unrolling even increased the execution time. For most of the manually unrolled programs, it is also the case that the new proportion is close to, but not quite as good as, the compiler's optimisation.



(a) Manual unrolling (b) Compiler optimisation

(c) Manual unrolling (d) Compiler optimisation

Figure 9: Comparing manual loop unrolling to compiler unrolling for various programs.

8 Given More Time

Given more time, I would have liked look further into why the 2d-array matrix multiplication is not being optimised. I would start off by examining if this was due to multiple optimisations being required on top of each other, or whether the problem was the 2d-arrays themselves.
I would also have liked to examine the impact of my chosen optimisation in "the grand scheme of things", i.e. enabled a general optimisation level, specifically disabled a single optimisation, and then compared the timings between all optimisations being enabled and all optimisations except from one being enabled. This would have given an insight into how much the chosen optimisations

affect, and are affected by, other optimisations being present.

9 Conclusion

In this practical, I constructed various small programs for testing compiler optimisations and evaluated a subset of these. For comparison, I then implemented some optimisations by hand and compared the results with the original code, and with the compiler. By doing so, I have confirmed that compilers are usually better at optimising programs than humans, discovered some peculiarities about computer architecture, and confirmed that primality testing is hard to optimise (I did not manage to manually find a method for unrolling the loop in the primality test).

This practical has helped me gain a better and more thorough understanding of compiler optimisations, the need for these, and which programs can be optimised how and why/why not.

References

Cordes, P., & balajimc55. (2017, May 23). *What is the best way to set a register to zero in x86 assembly: xor, mov or and?* Retrieved from `https://stackoverflow.com/questions/33666617/what-is-the-best-way-to-set-a-register-to-zero-in-x86-assembly-xor-mov-or-and` ([Online; accessed Nov. 2018])

Fog, A. (2018). *The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers.* Technical University of Denmark. Retrieved from `https://www.agner.org/optimize/microarchitecture.pdf`

Holly, R. (2005, Aug 1). *A Reusable Duff Device.* Retrieved from `http://www.drdobbs.com/a-reusable-duff-device/184406208?queryText=%2522duff%2527s%2Bdevice%2522` ([Online; accessed Nov. 2018])

Primality test - Wikipedia. (2018, Oct). Retrieved from `https://en.wikipedia.org/wiki/Primality_test#Pseudocode` ([Online; accessed Nov. 2018])

Using the GNU Compiler Collection (GCC): Optimize Options. (2018, Jul). Retrieved from `https://gcc.gnu.org/onlinedocs/gcc-8.2.0/gcc/Optimize-Options.html#Optimize-Options` ([Online; accessed Nov. 2018])

A Assembler Code

A.1 Vectorised

```
mmult:
.LFB25:
        .cfi_startproc
        testl   %ecx, %ecx
        jle     .L17
        pushq   %r12
        .cfi_def_cfa_offset 16
        .cfi_offset 12, -16
        pushq   %rbp
        .cfi_def_cfa_offset 24
        .cfi_offset 6, -24
        pushq   %rbx
        .cfi_def_cfa_offset 32
        .cfi_offset 3, -32
        movq    %rdi, %r10
        movq    %rsi, %r11
        leal    -1(%rcx), %r9d
        leaq    8(%rdi,%r9,8), %r12
        leaq    4(,%r9,4), %rbp
        movl    $0, %ebx
        jmp     .L11
.L15:
        movq    %rcx, %rax
.L12:
        movq    (%rdx,%rax,8), %rcx
        movl    (%rcx,%rdi), %ecx
        imull   (%r8,%rax,4), %ecx
        addl    %ecx, %esi
        leaq    1(%rax), %rcx
        cmpq    %r9, %rax
        jne     .L15
        movq    (%r10), %rax
        movl    %esi, (%rax,%rdi)
        addq    $4, %rdi
        cmpq    %rbp, %rdi
        je      .L13
.L14:
        movq    (%r11), %r8
        movq    %rbx, %rax
        movl    $0, %esi
        jmp     .L12
.L13:
        addq    $8, %r10
        addq    $8, %r11
        cmpq    %r12, %r10
        je      .L9
.L11:
        movl    $0, %edi
        jmp     .L14
.L9:
        popq    %rbx
        .cfi_def_cfa_offset 24
```

```
mmult:
.LFB25:
        .cfi_startproc
        testl   %ecx, %ecx
        jle     .L17
        pushq   %r12
        .cfi_def_cfa_offset 16
        .cfi_offset 12, -16
        pushq   %rbp
        .cfi_def_cfa_offset 24
        .cfi_offset 6, -24
        pushq   %rbx
        .cfi_def_cfa_offset 32
        .cfi_offset 3, -32
        movq    %rdi, %r10
        movq    %rsi, %r11
        leal    -1(%rcx), %r9d
        leaq    8(%rdi,%r9,8), %r12
        leaq    4(,%r9,4), %rbp
        movl    $0, %ebx
        jmp     .L11
.L15:
        movq    %rcx, %rax
.L12:
        movq    (%rdx,%rax,8), %rcx
        movl    (%rcx,%rdi), %ecx
        imull   (%r8,%rax,4), %ecx
        addl    %ecx, %esi
        leaq    1(%rax), %rcx
        cmpq    %r9, %rax
        jne     .L15
        movq    (%r10), %rax
        movl    %esi, (%rax,%rdi)
        addq    $4, %rdi
        cmpq    %rbp, %rdi
        je      .L13
.L14:
        movq    (%r11), %r8
        movq    %rbx, %rax
        movl    $0, %esi
        jmp     .L12
.L13:
        addq    $8, %r10
        addq    $8, %r11
        cmpq    %r12, %r10
        je      .L9
.L11:
        movl    $0, %edi
        jmp     .L14
.L9:
        popq    %rbx
        .cfi_def_cfa_offset 24
```

Figure 10: 2d-array matrix multiplication is unaffected by vectorisation.

reduce_ftree_vectorize.s

```
main:
.LFB24:
	.cfi_startproc
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	pushq	%r14
	pushq	%r13
	pushq	%r12
	pushq	%rbx
	.cfi_offset 14, -24
	.cfi_offset 13, -32
	.cfi_offset 12, -40
	.cfi_offset 3, -48
	subq	$65536, %rsp
	movq	%rsp, %r14
	movl	$100000, %r13d
	movl	$0, %r12d
	leaq	65536(%r14), %rbx
.L2:
	movq	%r14, %rax
	pxor	%xmm1, %xmm1
	pxor	%xmm4, %xmm4
.L3:
	movdqu	(%rax), %xmm0
	movdqa	%xmm4, %xmm2
	pcmpgtd	%xmm0, %xmm2
	movdqa	%xmm0, %xmm3
	punpckldq	%xmm2, %xmm3
	paddq	%xmm3, %xmm1
	punpckhdq	%xmm2, %xmm0
	paddq	%xmm0, %xmm1
	addq	$16, %rax
	cmpq	%rax, %rbx
	jne	.L3
	movdqa	%xmm1, %xmm0
	psrldq	$8, %xmm0
	paddq	%xmm1, %xmm0
	movq	%xmm0, %rax
	addq	%rax, %r12
	movq	%r12, %rsi
	movl	$.LC0, %edi
	movl	$0, %eax
	call	printf
	subl	$1, %r13d
	jne	.L2
	movl	$0, %eax
	leaq	-32(%rbp), %rsp
	popq	%rbx
	popq	%r12
```

Figure 11: `reduce` optimised using vector registers.

```
makeNode:
.LFB24:
    .cfi_startproc
    pushq   %rbx
    .cfi_def_cfa_offset 16
    .cfi_offset 3, -16
    movl    %edi, %ebx
    movl    $24, %edi
    call    malloc
    movl    %ebx, (%rax)
    pxor    %xmm0, %xmm0
    movups  %xmm0, 8(%rax)
    popq    %rbx
    .cfi_def_cfa_offset 8
    ret
    .cfi_endproc
```

```
makeNode:
.LFB24:
    .cfi_startproc
    pushq   %rbx
    .cfi_def_cfa_offset 16
    .cfi_offset 3, -16
    movl    %edi, %ebx
    movl    $24, %edi
    call    malloc
    movl    %ebx, (%rax)
    movq    $0, 16(%rax)
    movq    $0, 8(%rax)
    popq    %rbx
    .cfi_def_cfa_offset 8
    ret
    .cfi_endproc
```

Figure 12: Using `pxor` to set left and right child to 0 instead of using the constant `$0`.

A.2 Sibling and Tail Recursive Calls

```
    .globl  search
    .type   search, @function
search:
.LFB25:
    .cfi_startproc
    movq    %rdi, %rax
    testq   %rdi, %rdi
    je  .L8
    movl    (%rdi), %edx
    cmpl    %esi, %edx
    jne .L5
    ret
.L6:
    movq    16(%rax), %rax
.L7:
    testq   %rax, %rax
    je  .L8
    movl    (%rax), %edx
    cmpl    %esi, %edx
    je  .L8
.L5:
    cmpl    %edx, %esi
    jle .L6
    movq    8(%rax), %rax
    jmp .L7
.L8:
    ret
    .cfi_endproc
.LFE25:
    .size   search, .-search
    .globl  insert
    .type   insert, @function
insert:
.LFB26:
```

```
search:
.LFB25:
    .cfi_startproc
    testq   %rdi, %rdi
    je  .L6
    subq    $8, %rsp
    .cfi_def_cfa_offset 16
    movl    (%rdi), %edx
    movq    %rdi, %rax
    cmpl    %esi, %edx
    je  .L3
    jge .L5
    movq    8(%rdi), %rdi
    call    search
.L3:
    addq    $8, %rsp
    .cfi_remember_state
    .cfi_def_cfa_offset 8
    ret
.L5:
    .cfi_restore_state
    movq    16(%rdi), %rdi
    call    search
    jmp .L3
.L6:
    .cfi_def_cfa_offset 8
    movq    %rdi, %rax
    ret
    .cfi_endproc
.LFE25:
    .size   search, .-search
    .globl  insert
    .type   insert, @function
insert:
```

(a)

```
insert:
.LFB26:
    .cfi_startproc
    testq   %rdi, %rdi
    je  .L16
    pushq   %rbx
    .cfi_def_cfa_offset 16
    .cfi_offset 3, -16
    movq    %rdi, %rbx
    cmpl    %esi, (%rdi)
    jge .L12
    movq    8(%rdi), %rdi
    call    insert
    movq    %rax, 8(%rbx)
    movq    %rbx, %rax
.L10:
    popq    %rbx
    .cfi_def_cfa_offset 8
    ret
.L16:
    .cfi_restore 3
    movl    %esi, %edi
    jmp makeNode
.L12:
    .cfi_def_cfa_offset 16
    .cfi_offset 3, -16
    movq    16(%rdi), %rdi
    call    insert
    movq    %rax, 16(%rbx)
    movq    %rbx, %rax
    jmp .L10
    .cfi_endproc
.LFE26:
    .size   insert, .-insert
    .section    .rodata.str1.1,"aMS",@progbits,1
.LC0:
    .string "%d\n"
    .text
    .globl  inorder
    .type   inorder, @function
inorder:
```

```
insert:
.LFB26:
    .cfi_startproc
    pushq   %rbx
    .cfi_def_cfa_offset 16
    .cfi_offset 3, -16
    testq   %rdi, %rdi
    je  .L17
    movq    %rdi, %rbx
    cmpl    %esi, (%rdi)
    jge .L15
    movq    8(%rdi), %rdi
    call    insert
    movq    %rax, 8(%rbx)
    movq    %rbx, %rax
.L12:
    popq    %rbx
    .cfi_remember_state
    .cfi_def_cfa_offset 8
    ret
.L17:
    .cfi_restore_state
    movl    %esi, %edi
    call    makeNode
    jmp .L12
.L15:
    movq    16(%rdi), %rdi
    call    insert
    movq    %rax, 16(%rbx)
    movq    %rbx, %rax
    jmp .L12
    .cfi_endproc
.LFE26:
    .size   insert, .-insert
    .section    .rodata.str1.1,"aMS",@progbits,1
.LC0:
    .string "%d\n"
    .text
    .globl  inorder
    .type   inorder, @function
inorder:
```

(b)

```
    .type   inorder, @function
inorder:
.LFB27:
    .cfi_startproc
    pushq   %rbx
    .cfi_def_cfa_offset 16
    .cfi_offset 3, -16
    movq    %rdi, %rbx
    testq   %rdi, %rdi
    je  .L17
.L19:
    movq    16(%rbx), %rdi
    call    inorder
    movl    (%rbx), %esi
    movl    $.LC0, %edi
    movl    $0, %eax
    call    printf
    movq    8(%rbx), %rbx
    testq   %rbx, %rbx
    jne .L19
.L17:
    popq    %rbx
    .cfi_def_cfa_offset 8
    ret
    .cfi_endproc
.LFE27:
    .size   inorder, .-inorder
    .globl  makeTestBST
    .type   makeTestBST, @function
makeTestBST:
```

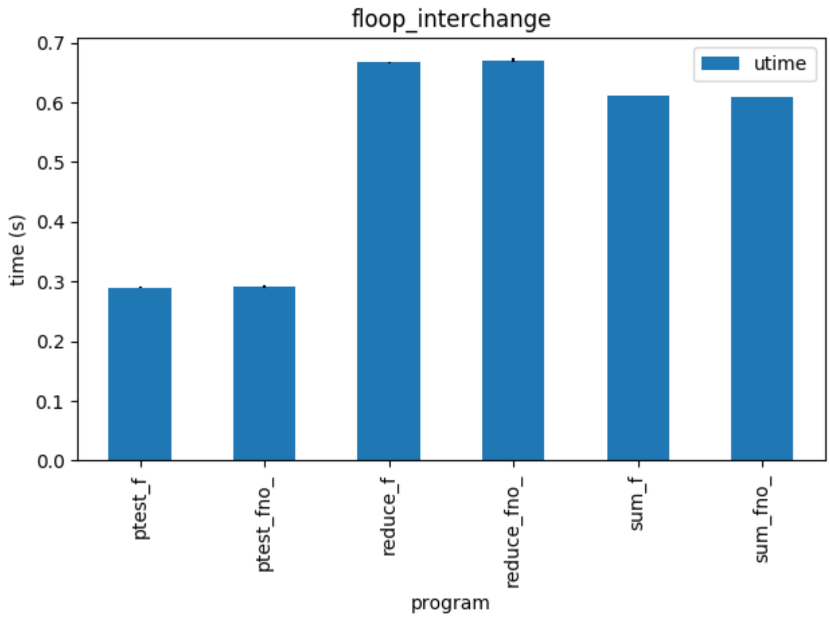
```
inorder:
.LFB27:
    .cfi_startproc
    testq   %rdi, %rdi
    je  .L21
    pushq   %rbx
    .cfi_def_cfa_offset 16
    .cfi_offset 3, -16
    movq    %rdi, %rbx
    movq    16(%rdi), %rdi
    call    inorder
    movl    (%rbx), %esi
    movl    $.LC0, %edi
    movl    $0, %eax
    call    printf
    movq    8(%rbx), %rdi
    call    inorder
    popq    %rbx
    .cfi_def_cfa_offset 8
    ret
.L21:
    .cfi_restore 3
    ret
    .cfi_endproc
.LFE27:
    .size   inorder, .-inorder
    .globl  makeTestBST
    .type   makeTestBST, @function
makeTestBST:
.LFB28:
```

(c)

Figure 12: The optimised vs. unoptimised `bst` assembler code, with the differences highlighted.

B Plots

B.1 Compiler Optimisations



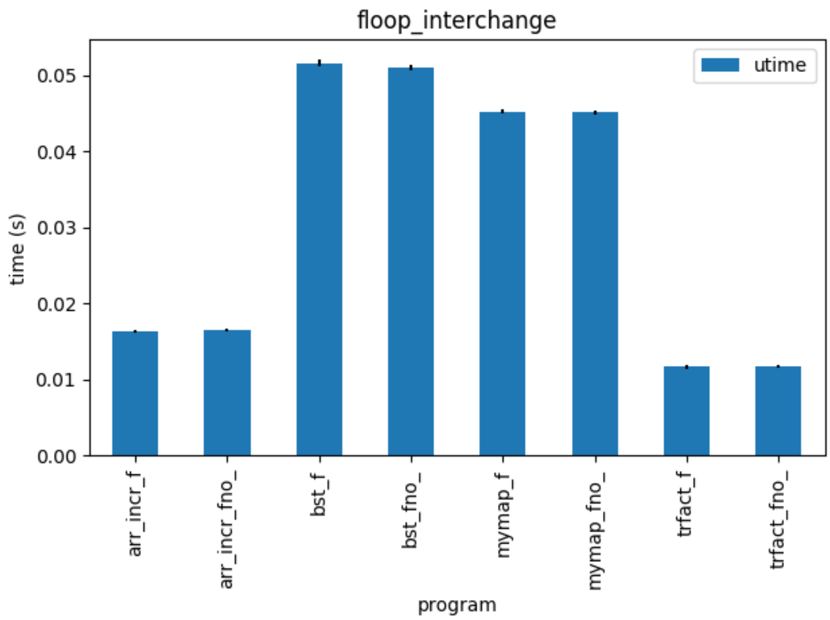


Figure 13: Loop interchange does not make a difference on the non matrix multiplication programs.

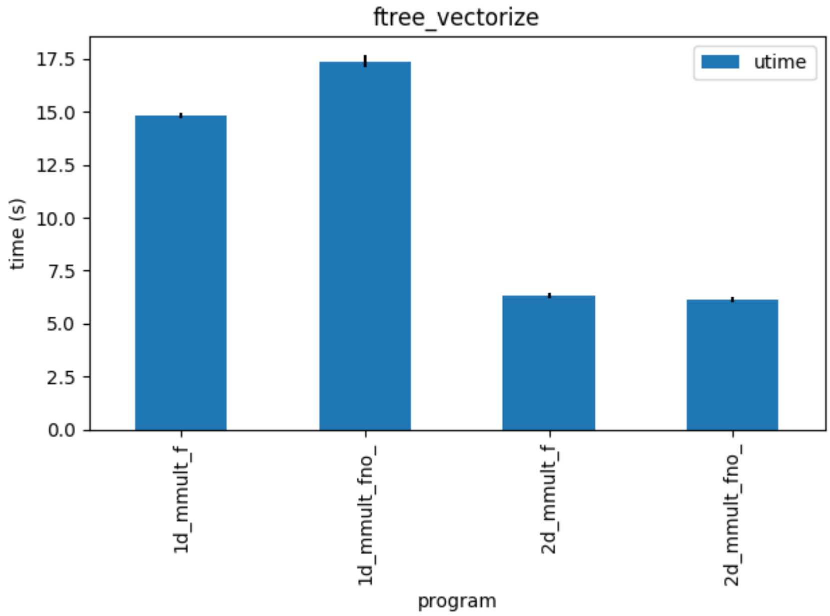


Figure 14: 1d-array matrix multiplication is sped up slightly using vectorisation. 2d remains unchanged.

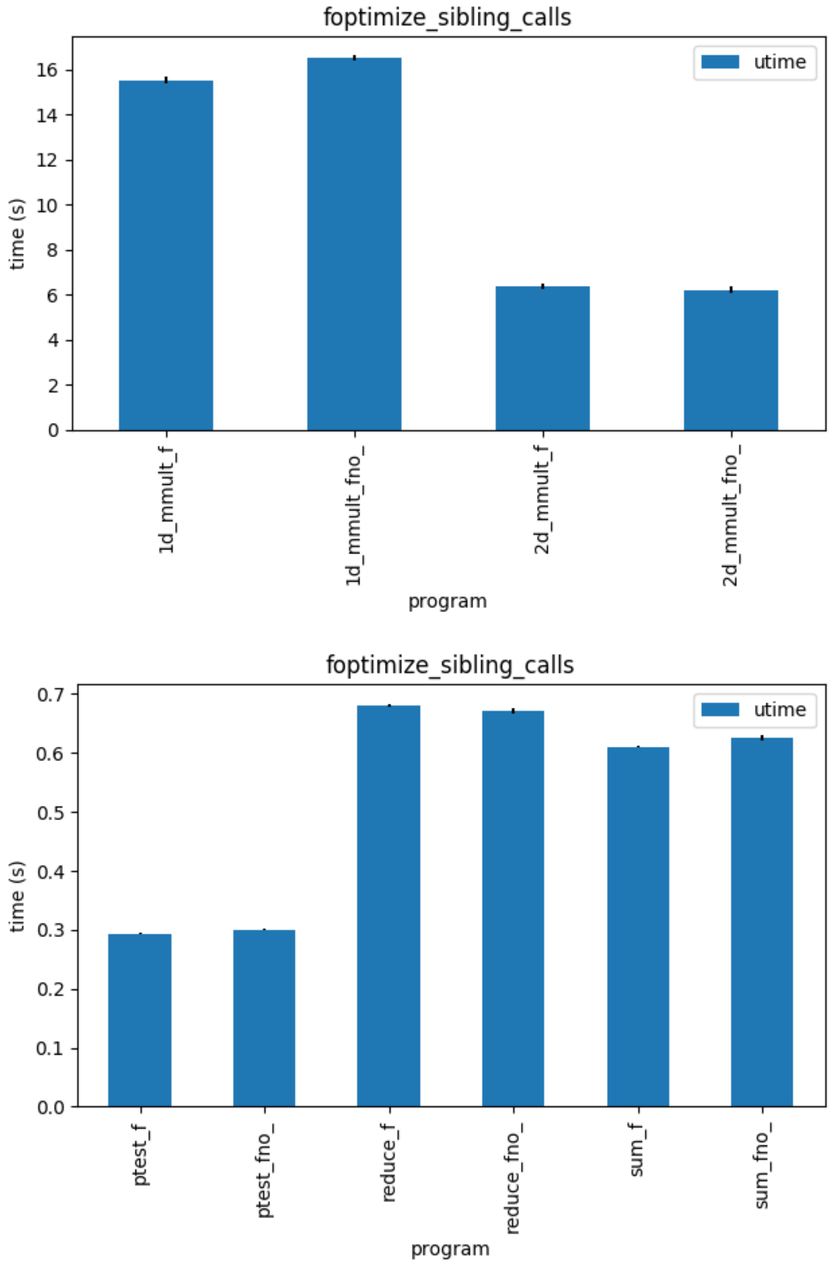
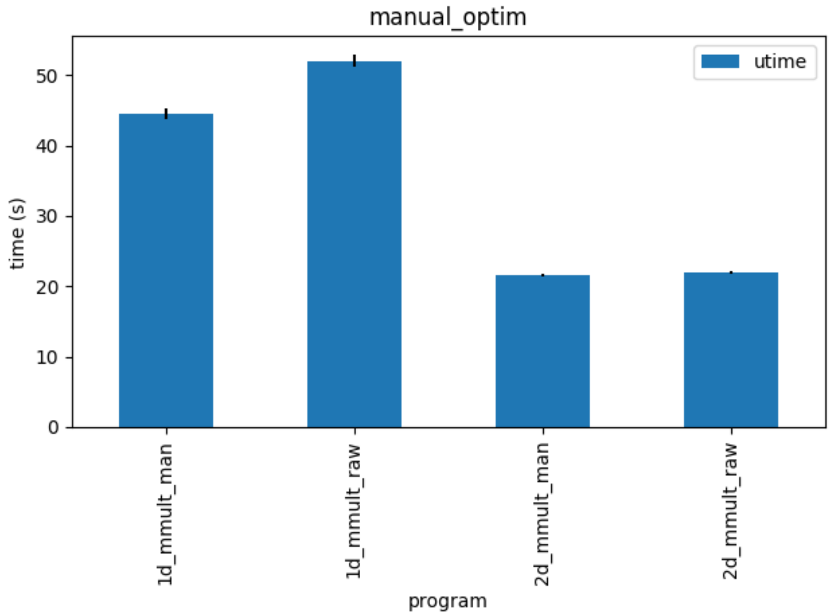


Figure 15: Several programs are unaffected by recursive call optimisations as they do not contain any.

B.2 Manual Optimisations



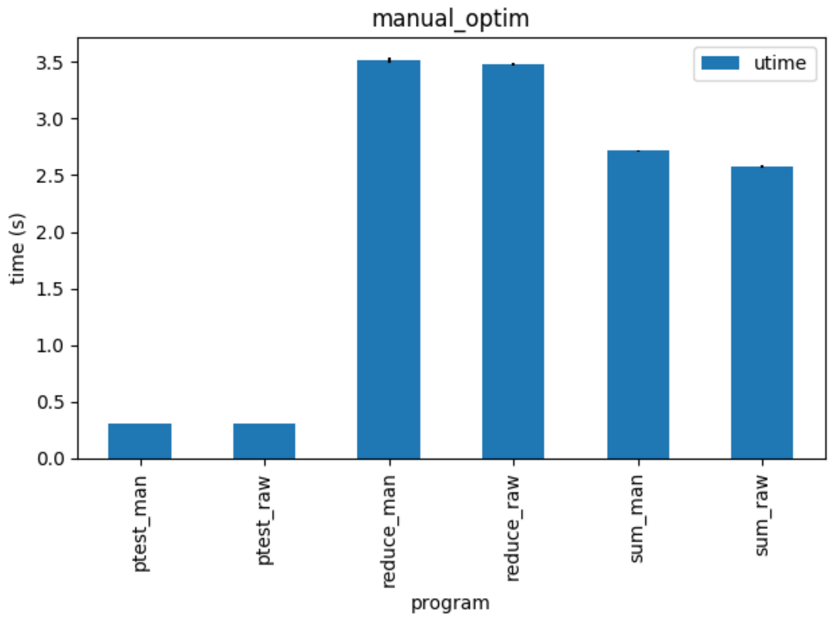


Figure 16: Most programs were unaffected by manually moving function definitions inline.