

Benchmarks Chosen and Base Performance

For benchmarking my compiler optimizations, and concretely measuring how much of a performance improvement I was creating, I used the *example29* and *example31* test programs provided. These programs make sense as benchmark programs primarily because they are CPU-intensive in their behavior. Specifically, *example29.c* multiplies 500x500 matrices represented as 250,000 element 1-D arrays, while *example31.c* multiplies 500x500 matrices represented as 500x500 2-D arrays. Both programs are highly CPU-bound, involving repetitive arithmetic operations and memory accesses. This makes them particularly sensitive to optimizations in the generated x86 code, especially in contrast with a program that depends on external outputs, has a lot of I/O, etc.

Starting off, before the optimizations, the performance of the programs was as follows (note I used user time as a measure of program's CPU time for all experiments). All performance numbers are averages across three trials, and were run on the UGRAD5 machine (Linux, 8 cores and 16 GiB RAM, shared machine).

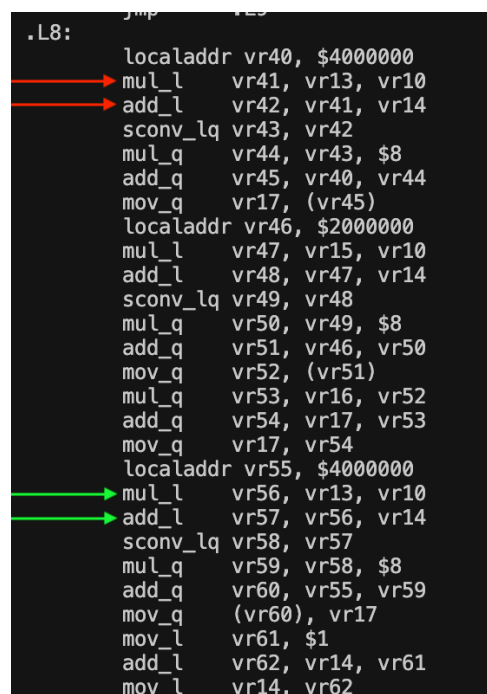
Example 29 -- user 0m1.776s

Example 31 -- user 0m1.791s

Local Value Numbering, Copy Propagation, Dead Store Elimination

The first optimization I implemented was really three transformations of the control flow graph in succession: local value numbering (LVN), copy propagation, dead store elimination. The purpose of these transformations was to identify and eliminate redundant computations within a basic block, so these are all local transformations (operating on a single basic block).

Specifically, in the image below, we can see the red arrows which identify a computation, and then the green arrows delineate the exact same computation a few instructions later.



```
.L8:
    jmp     .L9
    localaddr vr40, $4000000
    mul_l   vr41, vr13, vr10
    add_l   vr42, vr41, vr14
    sconv_lq vr43, vr42
    mul_q   vr44, vr43, $8
    add_q   vr45, vr40, vr44
    mov_q   vr17, (vr45)
    localaddr vr46, $2000000
    mul_l   vr47, vr15, vr10
    add_l   vr48, vr47, vr14
    sconv_lq vr49, vr48
    mul_q   vr50, vr49, $8
    add_q   vr51, vr46, vr50
    mov_q   vr52, (vr51)
    mul_q   vr53, vr16, vr52
    add_q   vr54, vr17, vr53
    mov_q   vr17, vr54
    localaddr vr55, $4000000
    mul_l   vr56, vr13, vr10
    add_l   vr57, vr56, vr14
    sconv_lq vr58, vr57
    mul_q   vr59, vr58, $8
    add_q   vr60, vr55, vr59
    mov_q   (vr60), vr17
    mov_l   vr61, $1
    add_l   vr62, vr14, vr61
    mov_l   vr14, vr62
```

The image shows a snippet of assembly code. Red arrows point to the first three instructions of a block: `mul_l vr41, vr13, vr10`, `add_l vr42, vr41, vr14`, and `sconv_lq vr43, vr42`. Green arrows point to the corresponding instructions later in the block: `mul_l vr56, vr13, vr10`, `add_l vr57, vr56, vr14`, and `sconv_lq vr58, vr57`. This illustrates how the same computations are repeated, which the optimization aims to eliminate.

There is no reason to do this same calculation twice, as the original result will definitely be available to reuse (as we are in the same basic block), so the first step of local value numbering is to identify these two final results (vr42 and vr57 in this case) as the same value. Local value numbering does exactly this, going through all instructions, assigning value numbers to each value in the high level code, and virtual registers with the same value number are guaranteed to have the same actual value in them.

Importantly, while numbering the values, if we come across a value that has already been calculated, instead of emitting the same computation instruction (like the `mul_l` and `add_l` in the above example), we replace that instruction with a move from the original computation's location instead.

```

.L8:      jmp      .L9
         localaddr vr40, $4000000
         mul_l     vr41, vr13, vr10
         add_l     vr42, vr41, vr14
         sconv_lq  vr43, vr42
         mul_q     vr44, vr43, $8
         add_q     vr45, vr40, vr44
         mov_q     vr17, (vr45)
         localaddr vr46, $2000000
         mul_l     vr47, vr15, vr10
         add_l     vr48, vr47, vr14
         sconv_lq  vr49, vr48
         mul_q     vr50, vr49, $8
         add_q     vr51, vr46, vr50
         mov_q     vr52, (vr51)
         mul_q     vr53, vr16, vr52
         add_q     vr54, vr17, vr53
         mov_q     vr17, vr54
         localaddr vr55, $4000000
         mov_l     vr56, vr41
         mov_l     vr57, vr42
         sconv_lq  vr58, vr57
         mul_q     vr59, vr58, $8
         add_q     vr60, vr55, vr59
         mov_q     (vr60), vr17
         mov_l     vr61, $1
         add_l     vr62, vr14, vr61
         mov_l     vr14, vr62

```

Here we can see, the original computation instructions are unchanged, but the subsequent instructions (green) that we previously identified as having the same values have been replaced with move instructions, moving from the locations where the previous computations were stored. We have now eliminated the recomputation (the `mul_l` and `add_l` instructions) of that value.

The next step here is copy propagation. Instead of recomputing values, we move from vr41 to vr56, and from vr42 to vr57; but instead of that, we can directly use vr41 and vr42 anytime vr56 and vr57 are subsequently used; and those move instructions become unnecessary. Copy propagation does precisely this, as the `sconv_lq` instruction (pink) would be replaced with

`sconv_lq vr58, vr42`. Any other references to `vr57` or `vr56` would also be treated in this manner, being replaced with the `vreg` where its computation was originally stored. Copy propagation simplifies code by reducing the number of unique `vregs` (and ultimately machine registers and stack memory locations), improves readability, and importantly enables the next transformation (dead store elimination).

This copy propagation now means that the two `mov_l` instructions (green) are unnecessary, since in this case we don't read from `vr56` and `vr57` if the pink instruction is modified as mentioned. Those are now considered "dead stores" as they store to a virtual register that is immediately dead, ie. never used. We can eliminate those instructions with no issues, and this transformation is called dead store elimination.

```

.L8:      jmp      .L9
         localaddr vr40, $4000000
         mul_l    vr41, vr13, vr10
         add_l    vr42, vr41, vr14
         sconv_lq vr43, vr42
         mul_q    vr44, vr43, $8
         add_q    vr45, vr40, vr44
         mov_q    vr17, (vr45)
         localaddr vr46, $2000000
         mul_l    vr47, vr15, vr10
         add_l    vr48, vr47, vr14
         sconv_lq vr49, vr48
         mul_q    vr50, vr49, $8
         add_q    vr51, vr46, vr50
         mov_q    vr52, (vr51)
         mul_q    vr53, vr16, vr52
         add_q    vr54, vr17, vr53
         mov_q    vr17, vr54
         localaddr vr55, $4000000
         sconv_lq vr58, vr42
         mul_q    vr59, vr58, $8
         add_q    vr60, vr55, vr59
         mov_q    (vr60), vr17
         mov_l    vr61, $1
         add_l    vr62, vr14, vr61
         mov_l    vr14, vr62

```

The result of this is that the two `mov_l` instructions (previously green) are removed, and the pink instruction now uses `vr42` (replacing `vr57`) directly. Since `vr56` also wasn't read from, that line was also eliminated. Dead store elimination reduces memory and register writes, improving performance and as well as freeing registers and stack memory for parts of the program

The performance after these transformations is as follows:

Example 29 -- user 0m1.624s

Example 31 -- user 0m1.696s

There clearly was a speed up, but this speed up isn't as large as one may expect. The reason for this is, upon inspection of original and resulting assembly code for these programs, there isn't a massive number of these repeated computations to take advantage of. However, in general, the combination of these three transformations identifies opportunities to remove redundant computations, replaces those computations, and finally removes the instructions that

end up being unnecessary. Two rounds of these transformations are carried out in succession, as one round of the transformation may create new opportunities for more optimization, so therefore they are run twice.

Machine Register Allocation

In the originally generated assembly code, machine registers are not used for storing local variables at all, and as a result, there is an overwhelming amount of stack memory accesses involved in the assembly code.

```
movl    -6000496(%rbp), %r10d /* mov_l    vr10, vr18 */  
movl    %r10d, -6000560(%rbp)  
  
movl    $250000, -6000488(%rbp) /* mov_l    vr19, $250000 */  
  
movl    -6000488(%rbp), %r10d /* mov_l    vr11, vr19 */  
movl    %r10d, -6000552(%rbp)
```

For example, these are two moves between virtual registers, but because everything is in stack memory, which is markedly slower than CPU registers, these lead to inefficient code. The solution is to use CPU registers, which are far more performant than storing everything in stack memory. Of course, not everything can be stored in registers, so my approach was to store all local variables in callee-saved registers (specifically r12, r13, r14, r15, rbx). This includes variables that are accessed globally throughout the function. Variables that are frequently accessed throughout the function like counter/index variables, and other data that is required to be accessible throughout the function, their virtual registers are allocated machine registers instead of stack memory.

```
movl    -6000496(%rbp), %r12d /* mov_l    vr10, vr18 */  
  
movl    $250000, -6000488(%rbp) /* mov_l    vr19, $250000 */  
  
movl    -6000488(%rbp), %r13d /* mov_l    vr11, vr19 */
```

As we can see, what was previously two *movl* statements, and used two different stack memory locations, now uses just one source stack memory location, and stores the value in a callee-saved machine register. Accessing these values is now far more performant.

```

movl    -6000120(%rbp), %r10d /* mov_l    vr13, vr65 */
movl    %r10d, -6000536(%rbp)

```

.L7:

```

movl    -6000536(%rbp), %r10d /* cmplt_l  vr66, vr13, vr10 */
cmpl    -6000560(%rbp), %r10d

```

Here is another example of using machine registers for local variables. Another added benefit we can see here as a result of using machine registers is that the *movl* instruction prior to the *cmpl* becomes unnecessary. Both the operands to *cmpl* cannot be memory locations, so when we use stack memory, we need an additional move there. Now in cases where one or both of the operands to the *cmpl* are CPU registers, no moves are needed at all prior to the comparison.

```

movl    -6000120(%rbp), %r15d /* mov_l    vr13, vr65 */

```

.L7:

```

cmpl    %r12d, %r15d /* cmplt_l  vr66, vr13, vr10 */

```

```

movl    -6000536(%rbp), %r10d /* mul_l    vr41, vr13, vr10 */
imull    -6000560(%rbp), %r10d

```

Finally, here is one more example of registers being used in an ALU instruction. Since values are in registers, we don't need the memory accesses, making this more efficient.

```

movl    %r15d, %r10d /* mul_l    vr41, vr13, vr10 */
imull    %r12d, %r10d

```

The performance after this transformation is as follows:

Example 29 -- user 0m1.392s

Example 31 -- user 0m1.451s

Clearly, we have made improvements in speed over the code being generated after the LVN/copy propagation/dead store elimination optimization. By moving at least some of the values being used by the program to CPU registers, not only is this faster because of their proximity and access by the CPU, but we also remove many of the stores back to memory, which are slow. Additionally, because there is less loading to temporary CPU registers (like r10) to do computations, we also limit move instructions overall, reducing the number of instructions and code size. Finally, using machine registers also opens the door to peephole optimizations, which we discuss next.

Peephole Optimization

While we now use registers, and reuse computations to the greatest degree possible, there are still a lot of idioms in our generated assembly code that could be made more efficient. The initial code generation only looks at one high level instruction at a time (like a move, an ALU computation, etc), and therefore it doesn't have the context to collapse slightly larger sequences of instructions into more efficient, shorter idioms. What peephole optimization does is look for sequences of instructions that match particular patterns (sequences of instructions with specific opcodes and types of operands), and when those sequences match, they can be replaced with alternate sequences that accomplish the same thing, but perhaps in fewer instructions or in a more efficient way.

I implemented several different peephole patterns, for which I will include before and after transformations of the low-level x86 assembly code. Note that this transformation is done on the x86 assembly code, not the high level intermediate representation.

Sign-extension

```
movl    %r12d, %r10d    /* sconv_lq vr14, vr11 */  
movslq  %r10d, %r10  
movq    %r10, -64(%rbp)
```

```
movslq    %r12d, %r10    /* sconv_lq vr14, vr11 */  
movq      %r10, -64(%rbp)
```

When sign-extending the value in a machine register, that value is first moved to a temporary register. However, this isn't necessary, and the value can be directly sign-extended, stored in a temporary, and then moved to its final location.

Multiplication instructions

```
movq    -64(%rbp), %r10  /* mul_q  vr15, vr14, $8 */  
imulq   $8, %r10  
movq    %r10, -56(%rbp)
```

```
imulq     $8, -64(%rbp), %r10 /* mul_q  vr15, vr14, $8 */  
movq      %r10, -56(%rbp)
```

X86 assembly code offers a three-operand version of the *imulq* instruction, which makes the first *movq* instruction in this original sequence unnecessary. When multiplying with an immediate value, we can directly use this three-operand version, reducing 3 lines of code to 2.

Comparison for jump conditions

```
cmpl    -24(%rbp), %r12d    /* cmplt_l  vr20, vr11, vr19 */
setl    %r10b
movzbl  %r10b, %r11d
movl    %r11d, -16(%rbp)
cmpl    $0, -16(%rbp)       /* cjmp_t   vr20, .L0 */
jne     .L0
```

```
cmpl    -24(%rbp), %r12d    /* cmplt_l  vr20, vr11, vr19 */
jl      .L0
```

In an effort to make our code generation general enough to handle the case where a comparison result may be used as a value, our initially generated code is very long and drawn-out when computing a logical value for a jump decision. When the result of the comparison is used purely to make the jump decision, we can use this far shorter and simpler idiom.

Simplify moves of immediate values

```
movl    $0, -6000464(%rbp) /* mov_l   vr22, $0 */
movl    -6000464(%rbp), %r15d /* mov_l   vr13, vr22 */

movl    $0, %r15d          /* mov_l   vr22, $0 */
```

When moving an immediate value, the temporary destination is unnecessary. We can easily remove it and move the immediate directly to its final destination. This prevents an unnecessary stack memory access.

Simplifying comparison with immediate values

```
movl    $250000, -6000024(%rbp) /* mov_l   vr77, $250000 */
cmpl    -6000024(%rbp), %r15d /* cmplt_l  vr78, vr13, vr77 */

cmpl    $250000, %r15d          /* mov_l   vr77, $250000 */
```

When doing a comparison instruction with an immediate value, the temporary move to another location is unnecessary. We can conduct the comparison directly on the value, removing an unnecessary stack memory access.

Simplifying ALU instructions

```
movl    %r15d, %r10d      /* add_l    vr29, vr13, vr28 */
addl    -6000416(%rbp), %r10d
movl    %r10d, -6000408(%rbp)
movl    -6000408(%rbp), %r15d /* mov_l    vr13, vr29 */

addl    -6000416(%rbp), %r15d /* add_l    vr29, vr13, vr28 */
```

When doing ALU instructions where at least one of the operands is a machine register, and the result is stored back to that register, we can remove the unnecessary moves and memory access in between. These four instructions can be collapsed into one because one of the source operand memory registers is also the destination for the computation.

Simplifying ALU instructions with immediate values

```
movl    $1, -6000416(%rbp) /* mov_l    vr28, $1 */
addl    -6000416(%rbp), %r15d /* add_l    vr29, vr13, vr28 */

addl    $1, %r15d          /* mov_l    vr28, $1 */
```

When one of the operands is an immediate, and the other operand is also the destination for the ALU computation, we can avoid the storing of the immediate value, and do this as one instruction.

Loading effective address into machine register

```
leaq    -6000000(%rbp), %r10 /* localaddr vr20, $0 */
movq    %r10, -6000480(%rbp)
movq    -6000480(%rbp), %rdi /* mov_q    vr1, vr20 */

leaq    -6000000(%rbp), %rdi /* localaddr vr20, $0 */
```

When using *leaq* to generate a memory address and then store that in a machine register, our initial code generated had a few unnecessary temporary moves. Those can be removed, as the net result is simply moving that address to the destination register. This is done with just one instruction.

The performance after the peephole optimization transformation is as follows:

Example 29 -- user 0m1.134s

Example 31 -- user 0m1.213s

We have made even further improvements in speed over our initial code generation. By using peephole optimization, we simplified code significantly, removing unnecessary instructions which reduces the code size in addition to improving execution speed. This also allows us, during the initial code generation step, to generate suboptimal code in order to be as general as possible, and correct those inefficiencies down the line. Peephole optimizations also allow the compiler to use more context than just one high-level instruction when generating code, while

still remaining a relatively low-cost optimization since it focuses on a smaller number of instructions, in a localized scope, at a time.

Remaining Inefficiencies

Overall, there have been significant gains in the speed of the code generated. *Example29* went from 1.776s to 1.134 seconds, approximately a 36% improvement. *Example31* went from 1.791s to 1.213s, approximately a 32% improvement. However, there are still inefficiencies that remain in the generated code that can be worked on and optimized.

Redundant Computations

While I did implement local value numbering, that only looks to identify redundant operations within a basic block. Superlocal value numbering is an optimization that would solve this issue, looking across basic blocks, at a larger scope, to detect and eliminate computational redundancies.

Limited Register Allocation

While I used machine registers for local variables, there are still many stack memory accesses in the code, being used for temporary virtual register translations. Doing some more local register allocation would rectify this, as moving some or all of those temporary virtual registers to machine registers (instead of stack memory) would definitely create performance gains.

Suboptimal Function Inlining

There were a number of functions in the example programs provided that have relatively short functions that are called many times. The overhead of repeatedly entering and exiting those functions definitely adds up, and these small functions can likely be inlined to prevent that function call overhead. Function inlining is an optimization that can be implemented here.

Constants

While I used LVN and carried out copy propagation in an effort to reduce redundant computation, another extension of this would be constant propagation. Right now, there are some cases where constants are regarded as registers or memory locations, when in reality their values are known at compile-time. Propagating such values through the program using constant propagation would solve this issue, and reduce unneeded memory accesses, as well as removing some computations which lead to known compile-time values.

Instruction Scheduling

The ordering of instructions as it currently is exactly the same order as instructions from the high level code representation, which directly translates the source code from the AST. However, especially when working with ALU computations, pipeline stalls are one thing to keep in mind. Currently, instructions aren't necessarily scheduled to avoid pipeline stalls or make efficient use of the parallelism modern CPUs offer. Instruction scheduling of some kind to reorder instructions based on data dependencies and pipeline characteristics could solve this inefficiency.

Unoptimized Loop Structures

No additional intelligence has been applied to the issue of loops, and as a result they are not optimized for things like iteration counts, branching overhead, or loop invariant code motion. Possible optimizations we could implement are loop unrolling or loop fusion; this would reduce the overhead associated with loops. Moving computations that do not depend on the loop out of the loop would also be prudent.