We can first compare two snippets of code before and after optimization:

High Level code generation (example09.c)

```
main:
        enter    $12
        mov_l    vr11, $1
        localaddr vr12, $0
        mov_l    vr13, $0
        sconv_lq vr14, vr13
        mul_q    vr15, vr14, $4
        add_q    vr16, vr12, vr15
        mov_l    (vr16), vr11
        mov_l    vr11, $2
        localaddr vr12, $0
        mov_l    vr13, $1
        sconv_lq vr14, vr13
        mul_q    vr15, vr14, $4
        add_q    vr16, vr12, vr15
        mov_l    (vr16), vr11
        mov_l    vr11, $3
        localaddr vr12, $0
        mov_l    vr13, $2
        sconv_lq vr14, vr13
        mul_q    vr15, vr14, $4
        add_q    vr16, vr12, vr15
        mov_l    (vr16), vr11
        localaddr vr11, $0
        mov_q    vr1, vr11
        mov_l    vr12, $3
        mov_l    vr2, vr12
        call     sum
        mov_l    vr13, vr0
        mov_l    vr10<%rbx>, vr13
        mov_l    vr0, vr10<%rbx>
        jmp      .Lmain_return
.Lmain_return:
        leave    $12
        ret
```

(unoptimized)

```
        .globl main
main:
        enter     $12
        localaddr vr12<%r8>, $0
        mov_l     (vr12)<%r8>, $1
        add_q     vr16<%r9>, vr12<%r8>, $4
        mov_l     (vr16)<%r9>, $2
        add_q     vr16<%r9>, vr12<%r8>, $8
        mov_l     (vr16)<%r9>, $3
        mov_q     vr1, vr12<%r8>
        mov_l     vr2, $3
        call      sum
        mov_l     vr13<%r8>, vr0
        mov_l     vr10<%rbx>, vr13<%r8>
        mov_l     vr0, vr10<%rbx>
        jmp       .Lmain_return
.Lmain_return:
        leave     $12
        ret
```

(optimized)

We can first see that the optimized code show noticeable length reduction comparing to the unoptimized one. Several important reduction include:

1. There are are four instances of localaddr, $0 in the original code, first three assigned to vr12 and the last one assigned to vr11, whereas in the optimized code, localaddr is only done once and stored in vr12, whereas any later instances of use of the address simply refers back to vr12 value (Local value numbering)

2. Constant values, such as the sequence:

   Vr 13 = 0
   Vr 14 = vr 13 = 0
   Vr 15 = vr 14 * 4 = 0
   Vr 16 = vr 12 + vr 15 = vr 12

   In this case, we are able to eliminate the above four instructions and only reserve the correspondence between vr 16 and vr 12, so that whenever vr 16 instance is called (shch as ‘mov_l (vr16), vr11), we would replace it with vr 12 (constant manipulation + copy propagation)

Low Level code generation: (example29.c)

```
.L2:
        movq     $0, -2488(%rbp)     /* mov_q    vr17, $0 */
        leaq     -800(%rbp), %r10    /* localaddr vr18, $1600 */
        movq     %r10, -2480(%rbp)
        movl     -2536(%rbp), %r10d  /* sconv_lq vr19, vr11<%rbx> */
        movslq   %r10d, %r10
        movq     %r10, -2472(%rbp)
        movq     -2472(%rbp), %r10   /* mul_q    vr20, vr19, $8 */
        imulq    $8, %r10
        movq     %r10, -2464(%rbp)
        movq     -2480(%rbp), %r10   /* add_q    vr21, vr18, vr20 */
        addq     -2464(%rbp), %r10
        movq     %r10, -2456(%rbp)
        movq     -2456(%rbp), %r10   /* mov_q    (vr21), vr17 */
        movq     -2488(%rbp), %r11
        movq     %r11, (%r10)
        movl     $1, -2488(%rbp)     /* mov_l    vr17, $1 */
        movl     -2536(%rbp), %r10d  /* add_l    vr18, vr11<%rbx>, vr17
*/
        addl     -2488(%rbp), %r10d
        movl     %r10d, -2480(%rbp)
        movl     -2480(%rbp), %r10d  /* mov_l    vr11<%rbx>, vr18 */
        movl     %r10d, -2536(%rbp)
```

```
.L2:
        leaq     -800(%rbp), %r8      /* localaddr vr18<%r8>, $1600 */
        movslq   %ebx, %r10           /* sconv_lq vr19<%r9>, vr11<%rbx> */
        movq     $0, (%r8,%r10,8)     /* add_q    vr21<%r9>, vr18<%r8>, vr20<%rcx>
*/
```

(Unoptimized) (Optimized)

Here again we see significant improvements. Some of the key simplification includes:

1. Now that most of the virtual registers are assigned with a machine register, we would not need to leaq the address from stack as previously. For example, in the first line of code, instead of up-casting vr11 with size 4 bit to vr19 with size 8 bit, instead of the previous approach of:

   Get vr11 from the stack, move to caller register %r10 → Up-cast %r10 →
   Copy the value of casted %r10 back to another place in stack where vr 19 is stored

   We can instead just do:

   Upscale value of vr11 in %rbx to vr19 in %r9 (machine register allocation)

2. For sections of code with repetitive patterns such as:

   ```
   Movq     -2472(%rbp),   %r10
   Imulq    $8,            %r10
   Movq     %r10,          -2464(%rbp)
   Movq     -2480(%rbp),   %r10
   Addq     -2464(%rbp),   %r10
   Movq     %r10,          -2456(%rbp)
   Movq     -2456(%rbp),   %r10
   Movq     -2488(%rbp),   %r11
   Movq     %r11           (%r10)
   ```

   These sequence of code represent a one-line assignment such as

   Arr[idx] = value

Where -2472(%rbp) is the idx, -2480(%rbp) is the arr pointer start or arr[0] position in stack, and -2488(%rbp) is value.

In the much more optimized code through previous optimization steps and peephole optimization, we can replace -2488(%rbp) (vr17) with constant 0, -2472(%rbp) with machine register %r10 and -2480(%rbp) with machine register %r8. Then we are able to complete the entire line of code in one instruction:

Movq $0, (%r8, %r10, 8)   (peephole optimization)

Overall efficiency

example29.c

```
bella-xia@BellaX:~/compilers-fall2023-tests/assign05$ ./build.rb example29
Generated code successfully assembled, output exe is out/example29
bella-xia@BellaX:~/compilers-fall2023-tests/assign05$ mkdir -p actual_output
bella-xia@BellaX:~/compilers-fall2023-tests/assign05$ time ./out/example29 < data/example29.in > actual_o
le29.out

real    0m1.495s
user    0m1.366s
sys     0m0.011s
bella-xia@BellaX:~/compilers-fall2023-tests/assign05$ diff expected_output/example29.out actual_output/ex
bella-xia@BellaX:~/compilers-fall2023-tests/assign05$ ./build.rb -o example29
Generated code successfully assembled, output exe is out/example29
bella-xia@BellaX:~/compilers-fall2023-tests/assign05$ mkdir -p actual_output
bella-xia@BellaX:~/compilers-fall2023-tests/assign05$ time ./out/example29 < data/example29.in > actual_o
le29.out

real    0m0.461s
user    0m0.421s
sys     0m0.003s
```

example31.c

```
bella-xia@BellaX:~/compilers-fall2023-tests/assign05$ ./build.rb example31
Generated code successfully assembled, output exe is out/example31
bella-xia@BellaX:~/compilers-fall2023-tests/assign05$ time ./out/example31 < data/example31.
le31.out

real    0m1.620s
user    0m1.483s
sys     0m0.002s
bella-xia@BellaX:~/compilers-fall2023-tests/assign05$ ./build.rb -o example31
Generated code successfully assembled, output exe is out/example31
bella-xia@BellaX:~/compilers-fall2023-tests/assign05$ mkdir -p actual_output
bella-xia@BellaX:~/compilers-fall2023-tests/assign05$ time ./out/example31 < data/example31.
le31.out

real    0m0.504s
user    0m0.463s
sys     0m0.002s
bella-xia@BellaX:~/compilers-fall2023-tests/assign05$
```

We can see that there is almost a 3-time improvement in the overall time-wise efficiency between the optimized and the unoptimized machine code.