Program Optimizations

2024 Fall ECE454: Computer Systems Programming — Lecture 5

Jon Eyolfson

1.0.1

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License

Today, We're Creating a Vector in C

Recall: a vector is a managed dynamically allocated array

The functions we've already written:

```
/* Create vector of specified length */
vec_ptr new_vec(int len);

/* Retrieve vector element at index, store at *dest
   Return 0 if out of bounds, 1 if successful */
int get_vec_element(vec_ptr v, int index, int *dest);

/* Return pointer to start of vector data */
int *get_vec_start(vec_ptr v);
```

What We'll be Optimizing: Combining Data

We want to be generic, to combine we'll accumlate the result of an operation between all elements of the vector

It'll have an operation, `OP`, and identity value `IDENT`, where `x OP IDENT == IDENT`

For for a sum, `OP` is `+` and `IDENT` is `0`

For for a product, `OP` is `*` and `IDENT` is `1`

We’ll Measure Cycles per Element (CPE) for Performance

The system I’ll use today is an AMD Ryzen 1800X (it’s a bit old)
    We’ll also calculate the sum

This could be our first attempt:

```
/* combine1: Maximum use of data abstraction */
void combine(vec_ptr v, data_t *dest) {
    *dest = IDENT;
    for (int64_t i = 0; i < vec_length(v); i++) {
            data_t val;
            get_vec_element(v, i, &val);
            *dest = *dest OP val;
    }
}
```

If we compile it with no optimizations and debugging we get a CPE of 20.22

Without Changing Anything, Lets Add Optimizations

We'll use -O2 as our default

```
/* combine1: Maximum use of data abstraction */
void combine(vec_ptr v, data_t *dest) {
    *dest = IDENT;
    for (int64_t i = 0; i < vec_length(v); i++) {
            data_t val;
            get_vec_element(v, i, &val);
            *dest = *dest OP val;
    }
}
```

Now when we run this we get a CPE of 10.00

The improvements are basically better register allocation and scheduling

What did the compiler miss that we could do?

Aside: Using `perf`

You should now be able to use `perf` on the UG machines

You can also send commands to `perf` using named pipe FDs
  See example: `examples/perf-wrapper.py` and `examples/src/benchmark.c`

It starts `perf` with `--delay -1` meaning disabled,
  after calling `setup` it enables `perf`

The Compiler Did not Lift `vec_length` Out of the Loop

- Since it's in a different compliation unit, the compiler has to assume it has side effects and it may not be deterministic
- Side effects may include reading or writing global state that may change
  - For example, if you did a `printf` in `vec_length` moving it would change the behaviour of your program

- Also, without knowing the final link step, even if it knew the implementation, you may not actually use that function at runtime

Let's Manually Do LICM, Since We Know It's Safe

We'll change our code to:

```
/* combine2: Take vec_length() out of loop */
void combine(vec_ptr v, data_t *dest) {
    int64_t length = vec_length(v);
    *dest = IDENT;
    for (int64_t i = 0; i < length; i++) {
            data_t val;
            get_vec_element(v, i, &val);
            *dest = *dest OP val;
    }
}
```

Now we get a healthy speedup, our CPE is now 7.00

What's the next thing we can do?

We Can Manually Inline `get_vec_element`

We'll change our code around, and get rid of the bounds check since we know it's valid:

```
/* combine3: Array reference to vector data */
void combine(vec_ptr v, data_t *dest) {
    int64_t length = vec_length(v);
    data_t *data = get_vec_start(v);

    *dest = IDENT;
    for (int64_t i = 0; i < length; i++) {
            *dest = *dest OP data[i];
    }
}
```

Now, our CPE is down to 1.68

We Can Try Removing the Memory Read in the Loop

We can create a local variable called `acc` to store the result:

```
/* combine3w: Update *dest within loop only with write */
void combine(vec_ptr v, data_t *dest) {
    int64_t length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;

    /* Initialize in event length <= 0 */
    *dest = acc;

    for (int64_t i = 0; i < length; i++) {
            acc = acc OP data[i];
            *dest = acc;
    }
}
```

Turns out that didn't do much, our CPE is still 1.68

We Actually Don’t Need to Write Everytime in the Loop

Since we only have one thread, we know it’s not possible for
anything else to read `dest`
(not entirely true, what else could happen?)

```
/* combine4: Array reference, accumulate in temporary */
void combine(vec_ptr v, data_t *dest) {
    int64_t length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;

    for (int64_t i = 0; i < length; i++) {
            acc = acc OP data[i];
    }
    *dest = acc;
}
```

This helps a bit more, our CPE is now 1.47

If We Think Array Indexing is Slow, We Can Try Pointers

We're just adding a constant value to the memory address,
why do we need to re-calculate?

```
/* combine4p: Pointer reference, accumulate in temporary */
void combine(vec_ptr v, data_t *dest) {
    int64_t length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t *dend = data+length;
    data_t acc = IDENT;

    for (; data < dend; data++)
            acc = acc OP *data;
    *dest = acc;
}
```

Turns out our compiler already optimized this for us, the CPU is still 1.47

Summary of Results so Far

Benchmark	CPE
combine1g	20.22
combine1	10.00
combine2	7.00
combine3	1.68
combine3w	1.68
combine4	1.47
combine4p	1.47

Don't Overuse Pointers in C/C++

It's very difficult for the compiler to reason about raw pointers
  (especially when basically anything is possible)

You should use local variables whenever possible
  The compiler can reason about the lifetime of local variables

Only update global state when you have to

What About Trying Loop Unrolling?

We can use the compiler flag `-funroll-loops` without changing the code

```
/* combine4: Array reference, accumulate in temporary */
void combine(vec_ptr v, data_t *dest) {
    int64_t length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;

    for (int64_t i = 0; i < length; i++) {
            acc = acc OP data[i];
    }
    *dest = acc;
}
```

This this compiler flag enabled we get a CPE of 1.05

Looking at the Assembly

Meson generates a `compile_commands.json` file in the build directory

I wrote a script for this example, `show-assembly.py`, it shows the generated assembly using the same compiler arguments as the build

Otherwise, you can get the compiler to output assembly using the `-S` flag
For x86 assmebly you may want to also add `-masm=intel`

Automatic Loop Unrolling Every 8 Elements

```
add         edx, DWORD PTR [rax]
add         rax, 32
add         edx, DWORD PTR -28[rax]
add         edx, DWORD PTR -24[rax]
add         edx, DWORD PTR -20[rax]
add         edx, DWORD PTR -16[rax]
add         edx, DWORD PTR -12[rax]
add         edx, DWORD PTR -8[rax]
add         edx, DWORD PTR -4[rax]
```

Assumed CPU Capabilities

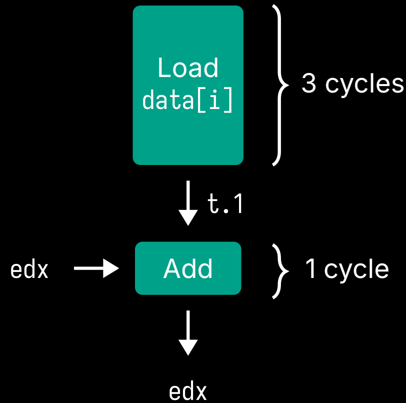
Some instructions can actually run in parallel:

- 1 load
- 1 store
- 2 integer (one may be branch)
- 1 FP addition
- 1 FP multiplication or division

Instructions Take > 1 Cycle, but Can Be Pipelined

Instruction	Latency	Cycles/Issue
Load / Store	3	1
Integer Add / Branch	1	1
Integer Multiply	4	1
Double/Single FP Multiply	5	2
Double/Single FP Add	3	1
Integer Divide	36	36
Double/Single FP Divide	38	38

Since We've Unrolled, It's Easier to Overlap Operations



What About Manual Loop Unrolling?

Let's unroll 3 times:

```
/* combine5uX: Manual loop unrolling X times. */
void combine(vec_ptr v, data_t *dest) {
    int64_t length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;

    const int BATCH_SIZE = 3;
    int64_t limit = length - BATCH_SIZE + 1;
    int64_t i;
    for (i = 0; i < limit; i+=BATCH_SIZE) {
            acc = acc OP data[i];
            acc = acc OP data[i+1];
            acc = acc OP data[i+2];
    }
    /* Fix up any remaining elements */
    for (; i < length; ++i) {
            acc = acc OP data[i];
    }
    *dest = acc;
}
```

This isn't any better than what the compiler did, our CPE for this is 1.05

Surely 4 Times is Better!

We'll also try this unrolling 5, 8, and 16 times

```
/* combine5uX: Manual loop unrolling X times. */
void combine(vec_ptr v, data_t *dest) {
    int64_t length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;

    const int BATCH_SIZE = 4;
    int64_t limit = length - BATCH_SIZE + 1;
    int64_t i;
    for (i = 0; i < limit; i+=BATCH_SIZE) {
            acc = acc OP data[i];
            acc = acc OP data[i+1];
            acc = acc OP data[i+2];
            acc = acc OP data[i+3];
    }
    /* Fix up any remaining elements */
    for (; i < length; ++i) {
            acc = acc OP data[i];
    }
    *dest = acc;
}
```

This is a bit of an improvement, our CPE is 0.61

Why Was 4 Times Better?

```
mov         rcx, rdx
add         rdx, 1
sal         rcx, 4
movdqu         xmm1, XMMWORD PTR [rax+rcx]
paddd         xmm0, xmm1
cmp         rdx, rsi
jb         .L3
movdqa         xmm1, xmm0
and         rdi, -4
psrldq         xmm1, 8
mov         rdx, rdi
paddd         xmm0, xmm1
add         rdx, 4
movdqa         xmm1, xmm0
psrldq         xmm1, 4
paddd         xmm0, xmm1
```

They used new registers, each register can store four 32-bit integers

This is part of the SSE2 ×86-64 Extension for SIMD instructions
- SIMD is short for single instruction multiple data

It Turns out Unrolling 8 Times is the Best

```
movdqu      xmm2, XMMWORD PTR [rdx]
movdqu      xmm3, XMMWORD PTR 16[rdx]
add         rcx, 1
add         rdx, 32
paddd       xmm1, xmm2
paddd       xmm0, xmm3
cmp         rcx, rsi
jb          .L3
paddd       xmm0, xmm1
and         rdi, -8
movdqa      xmm1, xmm0
mov         rdx, rdi
psrldq      xmm1, 8
add         rdx, 8
paddd       xmm0, xmm1
movdqa      xmm1, xmm0
psrldq      xmm1, 4
paddd       xmm0, xmm1
```

It unrolled it a bit more for us, and used more SSE2 registers

Maybe It's Better to Change the Order of Operations?

```
/* combine6: Try a different order of operations */
void combine(vec_ptr v, data_t *dest) {
    int64_t length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;

    const int BATCH_SIZE = 8;
    int64_t limit = length - BATCH_SIZE + 1;
    int64_t i;
    for (i = 0; i < limit; i+=BATCH_SIZE) {
        acc = acc OP (
            (data[i] OP data[i+1]) OP (data[i+2] OP data[i+3]) OP
            (data[i+4] OP data[i+5]) OP (data[i+6] OP data[i+7])
        );
    }
    /* Fix up any remaining elements */
    for (; i < length; ++i) {
            acc = acc OP data[i];
    }
    *dest = acc;
}
```

Turns out it won't be better than unrolling 8 or 16 times, our CPE is 0.80

However, it's better than manual loop unrolling!

Changing the Order of Operations is Better Since the Processor Can Overlap More

```
mov     edx, DWORD PTR 4[rax]
mov     ecx, DWORD PTR 12[rax]
add     rax, 32
add     ecx, DWORD PTR -24[rax]
add     edx, DWORD PTR -32[rax]
add     edx, ecx
mov     ecx, DWORD PTR -12[rax]
add     ecx, DWORD PTR -16[rax]
add     edx, ecx
mov     ecx, DWORD PTR -4[rax]
add     ecx, DWORD PTR -8[rax]
add     edx, ecx
add     esi, edx
cmp     rdi, rax
jne     .L3
and     r9, -8
add     r9, 8
```

The trade-off is we use more registers, increasing pressure
(if we use too many they'll spill on the stack and slow us down)

Summary of Loop Unrolling

Benchmark	CPE
combine4u	1.05
combine5u3	1.05
combine5u4	0.61
combine5u5	1.05
combine5u8	0.51
combine5u16	0.50
combine6	0.80

Some of Your Biggest Optimizations Come from Domain-specific Knowledge

For example, what if we were computing the product of all elements?

Is there an optimization we could make under certain conditions to skip the calculation?

Takeaways

Always profile to make sure you're optimizing the right thing!

Get the most out of your compiler before going manual
- Trade-off: manual optimization vs readable/maintainable code

Limit use of pointers, prefer local variables
- Reduce pointer-based arrays and pointer arithmetic
- Function pointers and virtual functions (unfortunately)

For highly performance-critical code:
- Look at assembly for optimization opportunities
- Consider the instruction-parallelism capabilities of CPU