High Performance Assembly Programming

Ray Seyfarth

June 29, 2012

Outline

1. Optimizations common to C/C++ and assembly

2. Optimizations the compiler can do in C, but you only in assembly

3. For assembly only

Use a better algorithm

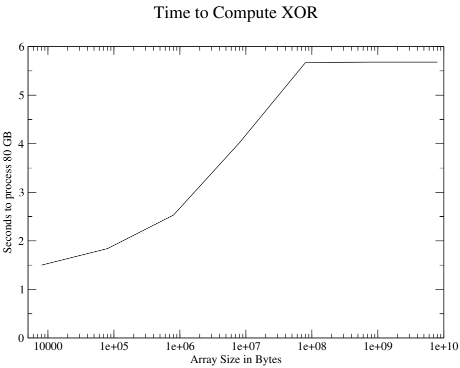
- A highly efficient insertion sort is still $O(n^2)$
- Using `qsort` from C is generally faster
- Using the C++ STL `sort` is faster still
- A hash table is $O(1)$ for lookup
- In you need an ordered dictionary, perhaps the STL `map` is best
- Tuning an $O(n^2)$ algorithm in assembly will not convert it to $O(n \lg n)$

Use C or C++

- An optimizing compiler will implement nearly all of the general optimizations
- It will do them tirelessly, missing nearly nothing
- Most of a program is not time-critical
- Perhaps 10% of a program is worth optimizing
- You must usually find a non-obvious technique to get better performance than the compiler
- Use the `-S` option to get an assembly listing
- Learn the compiler's tricks
- Perhaps you can do the compiler's tricks better

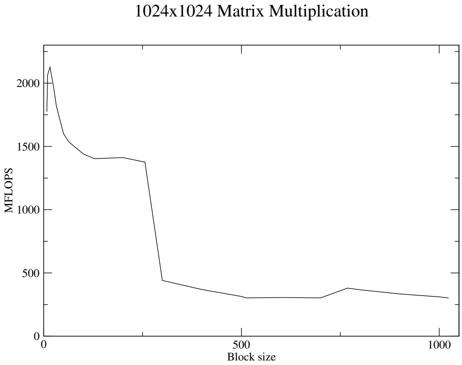
Efficient use of cache

- The CPU operates at about 3 GHz
- Main memory can provide perhaps 7 bytes per machine cycle
- Cache is much faster than main memory
- Organize your algorithm to work on data in blocks which fit in cache
- The plot below shows time versus array size for computing 10 billion exclusive or operations



Efficient use of cache(2)

- The plot below illustrates a dramatic performance gain through better use of cache
- The task was to compute a 1024×1024 matrix multiplication
- The code was written in C using 6 nested loops
- The 3 inner-most loops multiplied one block by another



Common sub-expression elimination

- The compiler will probably do this better than you
- You can examine its generated code and perhaps notice something you have overlooked
- I would bet my money of the compiler with this trick

Strength reduction

- This refers to using a simpler mathematical technique
- Dividing an integer by 8 could be a shift right 3 bits
- Getting a remainder after division by 1024, can be done using `and`
- Rather than using `pow(x,3)` use `x*x*x`
- Computer x^4 by computing x^2 and then squaring that
- Avoid division by a floating point number x, but computing $1/x$ and use multiplication instead
- Again the compiler will do this tirelessly

Use registers efficiently

- The compiler will do this automatically
- Place commonly-used values in registers
- If you unroll a loop, use different registers to allow parallel execution of parts of your computation

Use fewer branches

- Branches interrupt the instruction pipeline
- The compiler will frequently re-order blocks of code to reduce branches
- Study the compiler's generated code
- Use conditional moves for simple computations

Convert loops to branch at the bottom

- The compiler generally does this to reduce the number of instructions in a loop and, especially, the number of branches
- Here is a C `for` loop

```
for ( i = 0; i < n; i++ ) {
     x[i] = a[i] + b[i];
}
```

- By adding an if at the start you can loop with a branch at the bottom
- Don't do this in C. The compiler will handle this.

```
if ( n > 0 ) {
     i = 0;
     do {
          x[i] = a[i] + b[i];
          i++;
     } while ( i < n );
}
```

Unroll loops

- Use `-funroll-all-loops` to have `gcc` unroll loops
- Unrolling means repeated occurrences of the loop body with multiple parts of the data being processed
- Try to make each unrolling use different registers to reduce instruction dependence
- This frees up the CPU to do out-of-order execution
- It can do more pipelining and more parallel execution

Assembly code adding numbers in an array, unrolled

- The addition is done as 4 sub-sums which are added later
- The four unrolled parts accumulate into 4 different registers

```
.add_words:
        add     rax, [rdi]
        add     rbx, [rdi+8]
        add     rcx, [rdi+16]
        add     rdx, [rdi+16]
        add     rdi, 32
        sub     rsi, 4
        jg      .add_words
        add     rcx, rdx
        add     rax, rbx
        add     rax, rcx
```

Merge loops

- If 2 loops have some loop limits, consider merging the bodies
- There will be less loop overhead
- The following 2 loops can be profitably merged

```
for ( i = 0; i < 1000; i++ ) a[i] = b[i] + c[i];
for ( j = 0; j < 1000; j++ ) d[j] = b[j] - c[j];
```

- After merging values for `b[i]` and `c[i]` can be used twice

```
for ( i = 0; i < 1000; i++ ) {
    a[i] = b[i] + c[i];
    d[i] = b[i] - c[i];
}
```

Split loops

- Didn't I just suggest merging loops?
- Sometimes the data is unrelated and merging doesn't help
- Perhaps splitting uses cache better
- Test your code

Interchange loops

```
for ( j = 0; j < n; j++ ) {
    for ( i = 0; i < n; i++ ) {
        x[i][j] = 0;
    }
}
```

- The previous loop steps through the `x` array in large increments
- The loop below steps through the array one element after the other
- Cache fetches are better used

```
for ( i = 0; i < n; i++ ) {
    for ( j = 0; j < n; j++ ) {
        x[i][j] = 0;
    }
}
```

Move loop-invariant code outside the loop

- You can do this in C, but the compiler will do it for you
- The assembler does not move loop-invariant code
- Again, study the generated code

Remove recursion

- Eliminating tail-recursion is generally useful
- If you have to simulate a “stack” like recursion gives you, recursion will probably be faster

Eliminate stack frames

- Use `-fomit-frame-pointers` with `gcc`
- Use this for debugged code
- Using the `rbp` register is optional
- Leaf functions don't even need to worry about stack alignment
- Unless you are using some local data requiring 16 byte alignment

Inline functions

- The compiler can do this painlessly
- In assembly you will make your code less readable
- Explore using macros

Reduce dependencies to allow super-scalar execution

- Use different registers to try to reduce dependencies
- The CPU has multiple computational units in 1 core
- You can benefit from out-of-order execution
- You can get more out of pipelines
- You can keep more computational units busy

Use specialized instructions

- The compiler will have a harder time doing this than you
- There are SIMD integer instructions
- There are also SIMD floating point instructions
- The AVX instructions are a new feature which allow twice as many floating point values in the SIMD registers

