What is the fastest way to swap values in C?

Asked 16 years, 3 months ago Modified 4 years, 11 months ago Viewed 72k times

79

I want to swap two integers, and I want to know which of these two implementations will be faster: The obvious way with a temp variable:

```
void swap(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Or the xor version that I'm sure most people have seen:

```
void swap(int* a, int* b)
{
    *a ^= *b;
    *b ^= *a;
    *a ^= *b;
}
```

It seems like the first uses an extra register, but the second one is doing three loads and stores while the first only does two of each. Can someone tell me which is faster and why? The why being more important.

c performance

Share Improve this question Follow

asked Aug 31, 2008 at 15:12

JawnV6

1,427 ● 1 ● 13 ● 11

2 XOR is slower. Use godbolt to check assembler instructions count for both functions. **Note** that if you will use XOR method on values instead of values stored under pointer, speed is the same (at least for GCC compiler) – fider Sep 20, 2017 at 23:04

1 godbolt.org/z/nqVb9q – teknoraver Nov 13, 2019 at 1:40

3 *It seems like the first uses an extra register* More than a bit late here, but why would anyone think that? The belief that bit-twiddling is faster than using a temporary variable ignores the reality of how most computers work, with separate CPUs and memory. A swap using a temporary variable is likely implemented as "load A into register 1, load B into register 2, save register 1 to B, save register 2 to A". "Load both variables into registers, twiddle a bits around,

then do two save operations" is slower. **You have to load both and save both, bit-twiddling along the way is extraneous**. – Andrew Henle Mar 18, 2020 at 13:58

There are lots of methods for integers but they don't work for anything else like structures, floats, doubles, – cup Feb 10 at 10:17

21 Answers

Sorted by: Highest score (default)

113

Number 2 is often quoted as being the "clever" way of doing it. It is in fact most likely slower as it obscures the explicit aim of the programmer - swapping two variables. This means that a compiler can't optimize it to use the actual assembler ops to swap. It also assumes the ability to do a bitwise xor on the objects.

Stick to number 1, it's the most generic and most understandable swap and can be easily templated/genericized.

This wikipedia section explains the issues quite well: http://en.wikipedia.org/wiki/XOR_swap_algorithm#Reasons_for_avoidance_in_practice

Share

Improve this answer

Follow

edited Jan 7, 2011 at 23:35

Matt Davis

46k ● 17 ● 95 ● 125

answered Aug 31, 2008 at 15:19

caramelcarrot

1,130 ● 1 ● 7 ● 8

Spot on. In general, it's best to state your aim to the compiler, rather than trying to trick it into doing what you want. A swap-with-temporary-variable is such a common operation that any decent compiler can optimize it ruthlessly. – Dan Lenski Oct 1, 2008 at 3:19

2 I completely agree. Furthermore, if value swapping is really a bottleneck (proven by measurement), and can't be avoided, implement every way to do it you can think of and measure which is faster *for you* (your machine, OS, compiler, and app). There is no generic answer for low level stuff. – user25148 Mar 5, 2009 at 18:35

I was under the impression that `swap`, at least on x86, was really just calling three successive `xor`s – warren Sep 9, 2009 at 7:35

@warren: xchg %eax, %eax literally is what the standard one-byte NOP instruction code is. It doesn't zero %eax, therefore it's not using xor. – Peter Cordes Aug 5, 2014 at 15:36

@PeterCordes - why would %eax need to be zeroed? – warren Aug 5, 2014 at 19:19

93

The XOR method fails if a and b point to the same address. The first XOR will clear all of the bits at the memory address pointed to by both variables, so once the function returns (*a == *b == 0), regardless of the initial value.

More info on the Wiki page: XOR swap algorithm

Although it's not likely that this issue would come up, I'd always prefer to use the method that's guaranteed to work, not the clever method that fails at unexpected moments.

Share Improve this answer Follow

answered Aug 31, 2008 at 16:17

Ant

4,918 ● 1 ● 32 ● 43

3 It's pretty easy to prevent aliasing by adding a conditon *a != *b. – user9282 Sep 20, 2008 at 6:48

35 Then your swap function has a branch. As much as it's a silly question to begin with, if the OP is after speed then introducing a branch is probably a bad idea. – Matt Curtis Jan 22, 2009 at 3:29

8 @mamama, also, it should be a != b and not *a != *b; the fail is if the address is the same, not the value. – configurator Feb 4, 2009 at 15:17

1 It could be either - you don't need to swap if the values are already the same. But checking (a != b) makes more sense. – Greg Rogers Mar 5, 2009 at 18:12

13 If there is some clever trick to speed this up, your neighborhood compiler has already heard about it and is using it behind your back. Such micro-optimizations (particularly if done by hand) just don't get you anything today, memory access is *much* slower than executing instructions. Obfuscating your code for "performance" hurts in the most expensive part of the equation: Programmer time. – vonbrand Feb 1, 2013 at 21:05

44

On a modern processor, you could use the following when sorting large arrays and see no difference in speed:

```
void swap (int *a, int *b)
{
  for (int i = 1 ; i ; i <<= 1)
  {
    if ((*a & i) != (*b & i))
    {
      *a ^= i;
      *b ^= i;
    }
  }
}
```

The really important part of your question is the 'why?' part. Now, going back 20 years to the 8086 days, the above would have been a real performance killer, but on the latest Pentium it would be a match speed wise to the two you posted.

The reason is purely down to memory and has nothing to do with the CPU.

CPU speeds compared to memory speeds have risen astronomically. Accessing memory has become the major bottleneck in application performance. All the swap algorithms will be spending most of their time waiting for data to be fetched from memory. Modern OS's can have up to 5 levels of memory:

- Cache Level 1 - runs at the same speed as the CPU, has negligible access time, but is small
- Cache Level 2 - runs a bit slower than L1 but is larger and has a bigger overhead to access (usually, data needs to be moved to L1 first)
- Cache Level 3 - (not always present) Often external to the CPU, slower and bigger than L2
- RAM - the main system memory, usually implements a pipeline so there's latency in read requests (CPU requests data, message sent to RAM, RAM gets data, RAM sends data to CPU)
- Hard Disk - when there's not enough RAM, data is paged to HD which is really slow, not really under CPU control as such.

Sorting algorithms will make memory access worse since they usually access the memory in a very unordered way, thus incurring the inefficient overhead of fetching data from L2, RAM or HD.

So, optimising the swap method is really pointless - if it's only called a few times then any inefficiency is hidden due to the small number of calls, if it's called a lot then any inefficiency is hidden due to the number of cache misses (where the CPU needs to get data from L2 (1's of cycles), L3 (10's of cycles), RAM (100's of cycles), HD (!)).

What you really need to do is look at the algorithm that calls the swap method. This is not a trivial exercise. Although the Big-O notation is useful, an O(n) can be significantly faster than a O(log n) for small n. (I'm sure there's a CodingHorror article about this.) Also, many algorithms have degenerate cases where the code does more than is necessary (using qsort on nearly ordered data could be slower than a bubble sort with an early-out check). So, you need to analyse your algorithm and the data it's using.

Which leads to how to analyse the code. Profilers are useful but you do need to know how to interpret the results. Never use a single run to gather results, always average results over many executions - because your test application could have been paged

to hard disk by the OS halfway through. Always profile release, optimised builds, profiling debug code is pointless.

As to the original question - which is faster? - it's like trying to figure out if a Ferrari is faster than a Lambourgini by looking at the size and shape of the wing mirror.

Share
Improve this answer
Follow

edited Jul 3, 2012 at 13:36

user142162

answered Sep 5, 2008 at 10:30

Skizz
71k ● 10 ● 74 ● 109

6 +1 for the unnecessary optimization mention. If you've actually profiled your code and the biggest thing you have to worry about is which of these two ways of swapping a pair of ints is faster, you've written a very fast app. Until then, who cares about the swap? – Ken White Mar 5, 2009 at 23:37

1 @Ken White: I agree and moreover, if profiling shows that most time is spent in swapping it is most probably because you are swapping too many times (bubble sort anyone?), rather than swapping slowly. – David Rodríguez - dribeas Jul 21, 2010 at 22:36

In addition to the hard disk being much slower than RAM, going to swap also means you need to execute some completely different piece of code which probably is in RAM but almost certainly will not be in L1 cache, and likely not in L2 either (unless you are *seriously* short of RAM and swapping *constantly*). So before anything useful gets done, the CPU needs to fetch the part of the memory manager code that actually does the swapping. – user Jun 14, 2013 at 8:26

1 While your basic point is correct, the code you've shown is much slower than the two versions given in the question: Afaik, you get four `int` in one cache line, that means on average you get a latency of less than 30 cycles to load the data (not considering prefetching), you have conditional jumps in your loop (modern architectures hate mispredicting those), so you get much, much more than a cycle for each loop iteration. I'd wager, your swap will take at least 100 to 200 cycles, probably more, but that heavily depends on the numbers you are swapping (how many mispredictions are made). – cmaster - reinstate monica Oct 10, 2013 at 12:10

15

The first is faster because bitwise operations such as xor are usually very hard to visualize for the reader.

Faster to understand of course, which is the most important part ;)

Share Improve this answer Follow

answered Aug 31, 2008 at 15:39

Sander
26.3k ● 3 ● 54 ● 88

Regarding @Harry: Never implement functions as macros for the following reasons:

1. Type safety. There is none. The following only generates a warning when compiling but fails at run time:

   ```
   float a=1.5f,b=4.2f;
   swap (a,b);
   ```

   A templated function will always be of the correct type (and why aren't you treating warnings as errors?).

   EDIT: As there's no templates in C, you need to write a separate swap for each type or use some hacky memory access.

2. It's a text substitution. The following fails at run time (this time, without compiler warnings):

   ```
   int a=1,temp=3;
   swap (a,temp);
   ```

3. It's not a function. So, it can't be used as an argument to something like qsort.

4. Compilers are clever. I mean really clever. Made by really clever people. They can do inlining of functions. Even at link time (which is even more clever). Don't forget that inlining increases code size. Big code means more chance of cache miss when fetching instructions, which means slower code.

5. Side effects. Macros have side effects! Consider:

   ```
   int &f1 ();
   int &f2 ();
   void func ()
   {
     swap (f1 (), f2 ());
   }
   ```

   Here, f1 and f2 will be called twice.

   EDIT: A C version with nasty side effects:

   ```
   int a[10], b[10], i=0, j=0;
   swap (a[i++], b[j++]);
   ```

Macros: Just say no!

EDIT: This is why I prefer to define macro names in UPPERCASE so that they stand out in the code as a warning to use with care.

EDIT2: To answer Leahn Novash's comment:

Suppose we have a non-inlined function, f, that is converted by the compiler into a sequence of bytes then we can define the number of bytes thus:

```
bytes = C(p) + C(f)
```

where C() gives the number of bytes produced, C(f) is the bytes for the function and C(p) is the bytes for the 'housekeeping' code, the preamble and post-amble the compiler adds to the function (creating and destroying the function's stack frame and so on). Now, to call function f requires C(c) bytes. If the function is called n times then the total code size is:

```
size = C(p) + C(f) + n.C(c)
```

Now let's inline the function. C(p), the function's 'housekeeping', becomes zero since the function can use the stack frame of the caller. C(c) is also zero since there is now no call opcode. But, f is replicated wherever there was a call. So, the total code size is now:

```
size = n.C(f)
```

Now, if C(f) is less than C(c) then the overall executable size will be reduced. But, if C(f) is greater than C(c) then the code size is going to increase. If C(f) and C(c) are similar then you need to consider C(p) as well.

So, how many bytes do C(f) and C(c) produce. Well, the simplest C++ function would be a getter:

```
void GetValue () { return m_value; }
```

which would probably generate the four byte instruction:

```
mov eax,[ecx + offsetof (m_value)]
```

which is four bytes. A call instuction is five bytes. So, there is an overall size saving. If the function is more complex, say an indexer ("return m_value [index];") or a calculation ("return m_value_a + m_value_b;") then the code will be bigger.

edited Jan 14, 2020 at 13:30
Karussell
17.4k ● 17 ● 100 ● 203

answered Sep 5, 2008 at 15:58
Skizz
71k ● 10 ● 74 ● 109

4 Your side-effect code is C++, not C (there are no references in C). C programmers don't have templated functions... which may have some type-safety but are an absolutely nightmare to parse and otherwise implement. C++ != C. They have different types and degrees of abstraction and convention. – Dan Lenski Oct 1, 2008 at 3:16

9

For those to stumble upon this question and decide to use the XOR method. You should consider inlining your function or using a macro to avoid the overhead of a function call:

```
#define swap(a, b)   \
do {                 \
    int temp = a;    \
    a = b;           \
    b = temp;        \
} while(0)
```

Share

Improve this answer

Follow

edited Mar 5, 2009 at 18:16

Greg Rogers

36.4k ● 17 ● 69 ● 94

answered Sep 5, 2008 at 11:13

Harry

5,796 ● 6 ● 29 ● 21

2 +1. This is the way to do it in C, when you require speed. The macro can even be made type-flexible if you use the typeof() extension offered by GNU C. – Dan Lenski Oct 1, 2008 at 3:17

7 Err... Why would you use a compiler that can't do it's own inlining? Use functions when you can, macros when you must. Functions are type safe is easier to understand. Will this macro do the right thing with "swap(a++,b++)"?, will a functions? – John Nilsson Mar 5, 2009 at 23:41

1 If you're using a decent compiler, you can use `typeof(a)` or `decltype(a)` to make this more generic. Also, generally speaking, you should add parenthesis to avoid precedence issues (e.g. `#define foo(a, b) bar(a, b, (a) + (b))`). – Joey Adams Dec 18, 2010 at 6:30

1 This is a horrible solution. It will silently fail for floats. It also lacks parentheses. – Petter Sep 5, 2012 at 20:07

1 @John: copying my comment from another answer: typeof often lets you write macros that avoid evaluating their arguments more than once. `#define SWAP_BY_REF(a,b) do{ typeof(a) _a = (a); typeof(b) _b = (b); typeof(*_a) tmp=*_a; *_a=*_b; *_b=tmp;}while(0)`. Or you could do `_a=&a`, so you could use it on values. Hopefully compilers could still optimize away storing registers to memory so they'd have an address to take, for swapping two local variables that were already live in registers. GNU libc header files make a lot of use the `_a=(a)` trick in macros; that's where I first saw it. – Peter Cordes Aug 6, 2014 at 11:44

You are optimizing the wrong thing, both of those should be so fast that you'll have to run them billions of times just to get any measurable difference.

8

And just about anything will have much greater effect on your performance, for example, if the values you are swapping are close in memory to the last value you touched they are lily to be in the processor cache, otherwise you'll have to access the memory - and that is several orders of magnitude slower then any operation you do inside the processor.

Anyway, your bottleneck is much more likely to be an inefficient algorithm or inappropriate data structure (or communication overhead) then how you swap numbers.

Share Improve this answer Follow

answered Aug 31, 2008 at 20:34

Nir

29.6k ● 11 ● 68 ● 104

8

Never understood the hate for macros. When used properly they can make code more compact and readable. I believe most programmers know macros should be used with care, what is important is making it clear that a particular call is a macro and not a function call (all caps). If `SWAP(a++, b++);` is a consistent source of problems, perhaps programming is not for you.

Admittedly, the xor trick is neat the first 5000 times you see it, but all it really does is save one temporary at the expense of reliability. Looking at the assembly generated above it saves a register but creates dependencies. Also I would not recommend xchg since it has an implied lock prefix.

Eventually we all come to the same place, after countless hours wasted on unproductive optimization and debugging caused by our most clever code - Keep it simple.

```
#define SWAP(type, a, b) \
    do { type t=(a);(a)=(b);(b)=t; } while (0)

void swap(size_t esize, void* a, void* b)
{
    char* x = (char*) a;
    char* y = (char*) b;
    char* z = x + esize;

    for ( ; x < z; x++, y++ )
        SWAP(char, *x, *y);
}
```

Share Improve this answer Follow

answered Feb 21, 2013 at 15:53

SugarD

81 ● 1 ● 1

Trunicated? Perhaps SugarRichard would have been more appropriate in the dusk of the great detective. – SugarD Feb 25, 2013 at 16:30

2 How is this better than a function? – Sulthan Dec 5, 2013 at 16:24

1 typeof often lets you write macros that avoid evaluating their arguments more than once. `#define SWAP_BY_REF(a,b) do{ typeof(a) _a = (a); typeof(b) _b = (b); typeof(*_a) tmp=*_a; *_a=*_b; *_b=tmp;}while(0)` . Or you could do _a=&a, so you could use it on values rather than pointers. Hopefully compilers could still optimize away storing registers to memory so they'd have an address to take, for swapping two local variables that were already live in registers. GNU libc header files make a lot of use the `typeof(a) _a=(a)` trick in macros; that's where I first saw it. – Peter Cordes Aug 6, 2014 at 11:36

@PeterCordes `typeof` is a GCC-specific extension. – yyny Nov 12, 2017 at 19:24

5

The only way to really know is to test it, and the answer may even vary depending on what compiler and platform you are on. Modern compilers are *really* good at optimizing code these days, and you should never try to outsmart the compiler unless you can prove that your way is really faster.

With that said, you'd better have a damn good reason to choose #2 over #1. The code in #1 is far more readable and because of that should always be chosen first. Only switch to #2 if you can prove that you *need* to make that change, and if you do - comment it to explain what's happening and why you did it the non-obvious way.

As an anecdote, I work with a couple of people that *love* to optimize prematurely and it makes for really hideous, unmaintainable code. I'm also willing to bet that more often than not they're shooting themselves in the foot because they've hamstrung the ability of the compiler to optimize the code by writing it in a non-straightforward way.

Share Improve this answer Follow

answered Aug 31, 2008 at 15:58

17 of 26

27.4k ● 13 ● 68 ● 85

5

For modern CPU architectures, method 1 will be faster, also with higher readability than method 2.

On modern CPU architectures, the XOR technique is considerably slower than using a temporary variable to do swapping. One reason is that modern CPUs strive to execute instructions in parallel via instruction pipelines. In the XOR technique, the inputs to each operation depend on the results of the previous operation, so they must be executed in strictly sequential order. If efficiency is of tremendous concern, it is

advised to test the speeds of both the XOR technique and temporary variable swapping on the target architecture. Check out here for more info.

Edit: Method 2 is a way of **in-place swapping** (i.e. without using extra variables). To make this question complete, I will add another in-place swapping by using `+/-`.

```
void swap(int* a, int* b)
{
    if (a != b) // important to handle a/b share the same reference
    {
        *a = *a+*b;
        *b = *a-*b;
        *a = *a-*b;
    }
}
```

Share Improve this answer Follow

answered Jan 15, 2014 at 7:23

herohuyongtao
50.6k ● 30 ● 137 ● 176

actually, for the +/- in-place swapping, it's actually not critical to first ensure `a!=b`. Let's suppose we add a line before declaring a const variable, `const int C = *a`, such that `C == *a` and `C == *b` are true. Then: `*a = *a + *b` -> `*a` equals `C+C`; `*b = *a - *b` -> `*b` equals `C+C-C`, i.e. just `C`; `*a = *a - *b` -> `*a` equals `C+C-C`, i.e. just `C`; => `*a == C`, `*b == C` -> OK – CrepeGoat Jul 11, 2016 at 13:37

@Shillard It may not be critical, but useful to skip unnecessary swaps. :P – herohuyongtao Jul 12, 2016 at 3:02

I don't recommend adding a logic branch to your code when it adds no functionality. (Granted, it's justified if you've speed-tested it to be advantageous for your particular situation, i.e. 70+% of the time `a==b` or something... but as this is a general answer, and thus there is no particular situation, the logic branch is best left out.) Also the "important to handle a/b share the same reference" comment in your code is inaccurate. – CrepeGoat Sep 29, 2016 at 2:11

4

I would not do it with pointers unless you have to. The compiler cannot optimize them very well because of the possibility of pointer aliasing (although if you can GUARANTEE that the pointers point to non-overlapping locations, GCC at least has extensions to optimize this).

And I would not do it with functions at all, since it's a very simple operation and the function call overhead is significant.

The best way to do it is with macros if raw speed and the possibility of optimization is what you require. In GCC you can use the `typeof()` builtin to make a flexible version that works on any built-in type.

Something like this:

```
#define swap(a,b) \
  do { \
    typeof(a) temp; \
    temp = a; \
    a = b; \
    b = temp; \
  } while (0)

...
{
  int a, b;
  swap(a, b);
  unsigned char x, y;
  swap(x, y);                 /* works with any type */
}
```

With other compilers, or if you require strict compliance with standard C89/99, you would have to make a separate macro for each type.

A good compiler will optimize this as aggressively as possible, given the context, if called with local/global variables as arguments.

Share Improve this answer Follow

answered Oct 1, 2008 at 1:44

Dan Lenski

79.6k ● 13 ● 84 ● 127

i like your answer. it was the first thing that came to my mind. you might want to add use of "register" for c99 code, which also tells the compiler they don't alias (can be used if the programmer knows the arguments are not the same objects) – Johannes Schaub - litb Mar 5, 2009 at 17:12

4

All the top rated answers are not actually definitive "facts"... they are people who are speculating!

You can definitively *know for a fact* which code takes less assembly instructions to execute because you can look at the output assembly generated by the compiler and see which executes in less assembly instructions!

Here is the c code I compiled with flags "gcc -std=c99 -S -O3 lookingAtAsmOutput.c":

```
#include <stdio.h>
#include <stdlib.h>

void swap_traditional(int * restrict a, int * restrict b)
{
    int temp = *a;
```

```
    *a = *b;
    *b = temp;
}

void swap_xor(int * restrict a, int * restrict b)
{
    *a ^= *b;
    *b ^= *a;
    *a ^= *b;
}

int main() {
    int a = 5;
    int b = 6;
    swap_traditional(&a,&b);
    swap_xor(&a,&b);
}
```

ASM output for swap_traditional() takes >>> 11 <<< instructions (not including "leave", "ret", "size"):

```
.globl swap_traditional
    .type   swap_traditional, @function
swap_traditional:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %ecx
    pushl   %ebx
    movl    (%edx), %ebx
    movl    (%ecx), %eax
    movl    %ebx, (%ecx)
    movl    %eax, (%edx)
    popl    %ebx
    popl    %ebp
    ret
    .size   swap_traditional, .-swap_traditional
    .p2align 4,,15
```

ASM output for swap_xor() takes >>> 11 <<< instructions not including "leave" and "ret":

```
.globl swap_xor
    .type   swap_xor, @function
swap_xor:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %ecx
    movl    12(%ebp), %edx
    movl    (%ecx), %eax
    xorl    (%edx), %eax
    movl    %eax, (%ecx)
    xorl    (%edx), %eax
    xorl    %eax, (%ecx)
    movl    %eax, (%edx)
    popl    %ebp
```

```
    ret
    .size   swap_xor, .-swap_xor
    .p2align 4,,15
```

Summary of assembly output:
swap_traditional() takes 11 instructions
swap_xor() takes 11 instructions

Conclusion:
Both methods use the same amount of instructions to execute and therefore are approximately the same speed on this hardware platform.

Lesson learned:
When you have small code snippets, looking at the asm output is helpful to rapidly iterate your code and come up with the fastest (i.e. least instructions) code. And you can save time even because you don't have to run the program for each code change. You only need to run the code change at the end with a profiler to show that your code changes are faster.

I use this method a lot for heavy DSP code that needs speed.

Share

Improve this answer

Follow

edited Mar 11, 2009 at 1:03

community wiki
4 revs
Trevor Boyd Smith

1 It looks like you didn't enable optimization -- the local variables are getting loaded/stored many times in each function. Also, in modern processors, you can't easily count cycles, because anything that touches memory takes a variable number of cycles, depending on whether the cache hits or not. – Adam Rosenfield Mar 5, 2009 at 18:55

1 I did enable optimization with "-o3" and I even used "restrict" keyword to ensure that the compiler will optimized. What else am I missing? --- Lets say the number of cycles I counted isn't an absolute count. But I at least think it would be a relative count? So the trad. method still wins? – Trevor Boyd Smith Mar 5, 2009 at 19:55

4 -o3 says "name the output file 3". You need -O3 (with a capital O). – Adam Rosenfield Mar 7, 2009 at 1:03

6 On a pipelined superscalar (i.e. conteporary) CPU, you can't just count the number of instructions in the assembly code and call it "cycles". – bendin Mar 10, 2009 at 12:53

2 "Both methods use the same amount of instructions to execute and therefore are approximately the same speed on this hardware platform." And therefore *what*? Your reasoning is completely flawed. Obviously there is more to speed than mere instruction count. – alecov Feb 4, 2017 at 0:09

To answer your question as stated would require digging into the instruction timings of the particular CPU that this code will be running on which therefore require me to

3

make a bunch of assumptions around the state of the caches in the system and the assembly code emitted by the compiler. It would be an interesting and useful exercise from the perspective of understanding how your processor of choice actually works but in the real world the difference will be negligible.

Share Improve this answer Follow

answered Sep 2, 2008 at 19:15

Andrew O'Reilly

1,643 ● 13 ● 15

2

x=x+y-(y=x);

```
float x; cout << "X:"; cin >> x;
float y; cout << "Y:" ; cin >> y;

cout << "----------------------" << endl;
cout << "X=" << x << ", Y=" << y << endl;
x=x+y-(y=x);
cout << "X=" << x << ", Y=" << y << endl;
```

Share Improve this answer Follow

answered Aug 23, 2017 at 15:54

Marcin Snieg

31 ● 2

This ignores the possibility of integer overflow and the resulting undefined behavior.
– Andrew Henle Mar 18, 2020 at 14:08

1

In my opinion local optimizations like this should only be considered tightly related to the platform. It makes a huge difference if you are compiling this on a 16 bit uC compiler or on gcc with x64 as target.

If you have a specific target in mind then just try both of them and look at the generated asm code or profile your applciation with both methods and see which is actually faster on your platform.

Share Improve this answer Follow

answered Oct 10, 2008 at 12:11

Dan Cristoloveanu

2,048 ● 1 ● 14 ● 20

0

If you can use some inline assembler and do the following (psuedo assembler):

```
PUSH A
A=B
POP B
```

You will save a lot of parameter passing and stack fix up code etc.

Share

Improve this answer

Follow

edited Oct 10, 2013 at 11:27

user

6,947 ● 8 ● 46 ● 81

answered Aug 31, 2008 at 16:34

Tim Ring

1,833 ● 1 ● 20 ● 27

watch out: vc++ wont allow inline asm in 64bit mode. hope its relevant or understood as so :) – Joao Vilaca Jan 12, 2009 at 1:30

That swaps the contents of two registers, not of the locations they point to. Inline ASM also makes compilers much less able to optimize, so it's not worth it unless you're doing it for SSE instructions, or your inline asm includes the inner loop. – Peter Cordes Aug 5, 2014 at 15:27

In assembly there is also the xchg command, which swaps two values. – Palle Nov 3, 2015 at 23:01

Whats all the nitpickin for... 1) Psuedo code, I'm not literally pusging register 'A' blah blah. 2) Again, psuedo code, not referencing any particular assembler (xchg). 3) Many people not using 64 bit vc++ (aaargh). – Tim Ring Oct 11, 2016 at 9:47

-1

I just placed both swaps (as macros) in hand written quicksort I've been playing with. The XOR version was much faster (0.1sec) then the one with the temporary variable (0.6sec). The XOR did however corrupt the data in the array (probably the same address thing Ant mentioned).

As it was a fat pivot quicksort, the XOR version's speed is probably from making large portions of the array the same. I tried a third version of swap which was the easiest to understand and it had the same time as the single temporary version.

```
acopy=a;
bcopy=b;
a=bcopy;
b=acopy;
```

[I just put an if statements around each swap, so it won't try to swap with itself, and the XOR now takes the same time as the others (0.6 sec)]

Share

Improve this answer

Follow

edited Sep 5, 2008 at 9:06

answered Sep 4, 2008 at 22:41

paperhorse

4,155 ● 2 ● 24 ● 12

3 I like that evaluation! "It was faster, but it did corrupt the data." Classic. – unwind Mar 10, 2009 at 12:50

-1

If your compiler supports inline assembler and your target is 32-bit x86 then the XCHG instruction is probably the best way to do this... if you really do care that much about performance.

Here is a method which works with MSVC++:

```
#include <stdio.h>

#define exchange(a,b)   __asm mov eax, a \
                        __asm xchg eax, b \
                        __asm mov a, eax

int main(int arg, char** argv)
{
    int a = 1, b = 2;
    printf("%d %d --> ", a, b);
    exchange(a,b)
    printf("%d %d\r\n", a, b);
    return 0;
}
```

Share Improve this answer Follow

answered Mar 22, 2009 at 17:03

jheriko

3,103 ● 1 ● 22 ● 28

2 inline ASM makes it harder for the compiler to optimize. If xchg was faster, compilers would already use it. It isn't, because it has an implicit lock prefix. (VERY slow) – Peter Cordes Aug 5, 2014 at 15:30

1 right. i was not aware of this... thanks for enlightening me :) – jheriko Jul 1, 2015 at 15:29

-3

```
void swap(int* a, int* b)
{
    *a = (*b - *a) + (*b = *a);
}
```

// My C is a little rusty, so I hope I got the * right :)

Share

Improve this answer

Follow

edited Dec 5, 2013 at 16:02

answered Jun 18, 2009 at 14:52

Theofanis Pantelides

4,854 ● 7 ● 31 ● 49

-3

Below piece of code will do the same. This snippet is optimized way of programming as it doesn't use any 3rd variable.

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

Share Improve this answer Follow

answered Nov 9, 2015 at 4:26

Ashwin Balaji Kuppuraj

81 ● 1 ● 1 ● 9

4 Welcome to SO! Please realize that this question dates from 2008 (7 years ago), and that your answer is already part of that question. The OP was actually asking about speed performance, not memory. – ghybs Nov 9, 2015 at 4:50

-4

Another beautiful way.

```
#define Swap( a, b ) (a)^=(b)^=(a)^=(b)
```

Advantage

No need of function call and handy.

Drawback:

This fails when both inputs are same variable. It can be used only on integer variables.

Share

Improve this answer

Follow

edited Oct 7, 2009 at 18:04

answered Oct 7, 2009 at 17:57

Vadakkumpadath

1,485 ● 13 ● 22