

Why are elementwise additions much faster in separate loops than in a combined loop?

Asked 13 years ago Modified 1 year, 1 month ago Viewed 261k times



Suppose `a1`, `b1`, `c1`, and `d1` point to heap memory, and my numerical code has the following core loop.

2448



```
const int n = 100000;

for (int j = 0; j < n; j++) {
    a1[j] += b1[j];
    c1[j] += d1[j];
}
```

This loop is executed 10,000 times via another outer `for` loop. To speed it up, I changed the code to:

```
for (int j = 0; j < n; j++) {
    a1[j] += b1[j];
}

for (int j = 0; j < n; j++) {
    c1[j] += d1[j];
}
```

Compiled on [Microsoft Visual C++ 10.0](#) with full optimization and [SSE2](#) enabled for 32-bit on a [Intel Core 2 Duo](#) (x64), the first example takes 5.5 seconds and the double-loop example takes only 1.9 seconds.

Disassembly for the first loop basically looks like this (this block is repeated about five times in the full program):

```
movsd    xmm0,mmword ptr [edx+18h]
addsd    xmm0,mmword ptr [ecx+20h]
movsd    mmword ptr [ecx+20h],xmm0
movsd    xmm0,mmword ptr [esi+10h]
addsd    xmm0,mmword ptr [eax+30h]
movsd    mmword ptr [eax+30h],xmm0
movsd    xmm0,mmword ptr [edx+20h]
addsd    xmm0,mmword ptr [ecx+28h]
movsd    mmword ptr [ecx+28h],xmm0
movsd    xmm0,mmword ptr [esi+18h]
addsd    xmm0,mmword ptr [eax+38h]
```

Each loop of the double loop example produces this code (the following block is repeated about three times):

```

addsd    xmm0,mmword ptr [eax+28h]
movsd    mmword ptr [eax+28h],xmm0
movsd    xmm0,mmword ptr [ecx+20h]
addsd    xmm0,mmword ptr [eax+30h]
movsd    mmword ptr [eax+30h],xmm0
movsd    xmm0,mmword ptr [ecx+28h]
addsd    xmm0,mmword ptr [eax+38h]
movsd    mmword ptr [eax+38h],xmm0
movsd    xmm0,mmword ptr [ecx+30h]
addsd    xmm0,mmword ptr [eax+40h]
movsd    mmword ptr [eax+40h],xmm0

```

The question turned out to be of no relevance, as the behavior severely depends on the sizes of the arrays (n) and the CPU cache. So if there is further interest, I rephrase the question:

- Could you provide some solid insight into the details that lead to the different cache behaviors as illustrated by the five regions on the following graph?
- It might also be interesting to point out the differences between CPU/cache architectures, by providing a similar graph for these CPUs.

Here is the full code. It uses `TBB Tick_Count` for higher resolution timing, which can be disabled by not defining the `TBB_TIMING` Macro:

```

#include <iostream>
#include <iomanip>
#include <cmath>
#include <string>

// #define TBB_TIMING

#ifdef TBB_TIMING
#include <tbb/tick_count.h>
using tbb::tick_count;
#else
#include <time.h>
#endif

using namespace std;

// #define preallocate_memory new_cont

enum { new_cont, new_sep };

double *a1, *b1, *c1, *d1;

void allo(int cont, int n)
{
    switch(cont) {
        case new_cont:
            a1 = new double[n*4];
            b1 = a1 + n;
            c1 = b1 + n;
            d1 = c1 + n;

```

```

        break;
    case new_sep:
        a1 = new double[n];
        b1 = new double[n];
        c1 = new double[n];
        d1 = new double[n];
        break;
    }

    for (int i = 0; i < n; i++) {
        a1[i] = 1.0;
        d1[i] = 1.0;
        c1[i] = 1.0;
        b1[i] = 1.0;
    }
}

void ff(int cont)
{
    switch(cont){
        case new_sep:
            delete[] b1;
            delete[] c1;
            delete[] d1;
        case new_cont:
            delete[] a1;
    }
}

double plain(int n, int m, int cont, int loops)
{
#ifdef preallocate_memory
    allo(cont,n);
#endif

#ifdef TBB_TIMING
    tick_count t0 = tick_count::now();
#else
    clock_t start = clock();
#endif

    if (loops == 1) {
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++){
                a1[j] += b1[j];
                c1[j] += d1[j];
            }
        }
    } else {
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                a1[j] += b1[j];
            }
            for (int j = 0; j < n; j++) {
                c1[j] += d1[j];
            }
        }
    }

    double ret;

#ifdef TBB_TIMING
    tick_count t1 = tick_count::now();

```

```

        ret = 2.0*double(n)*double(m)/(t1-t0).seconds();
    #else
        clock_t end = clock();
        ret = 2.0*double(n)*double(m)/(double)(end - start)
        *double(CLOCKS_PER_SEC);
    #endif

    #ifndef preallocate_memory
        ff(cont);
    #endif

    return ret;
}

void main()
{
    freopen("C:\\test.csv", "w", stdout);

    char *s = " ";

    string na[2] = {"new_cont", "new_sep"};

    cout << "n";

    for (int j = 0; j < 2; j++)
        for (int i = 1; i <= 2; i++)
    #ifdef preallocate_memory
        cout << s << i << "_loops_" << na[preallocate_memory];
    #else
        cout << s << i << "_loops_" << na[j];
    #endif

    cout << endl;

    long long nmax = 1000000;

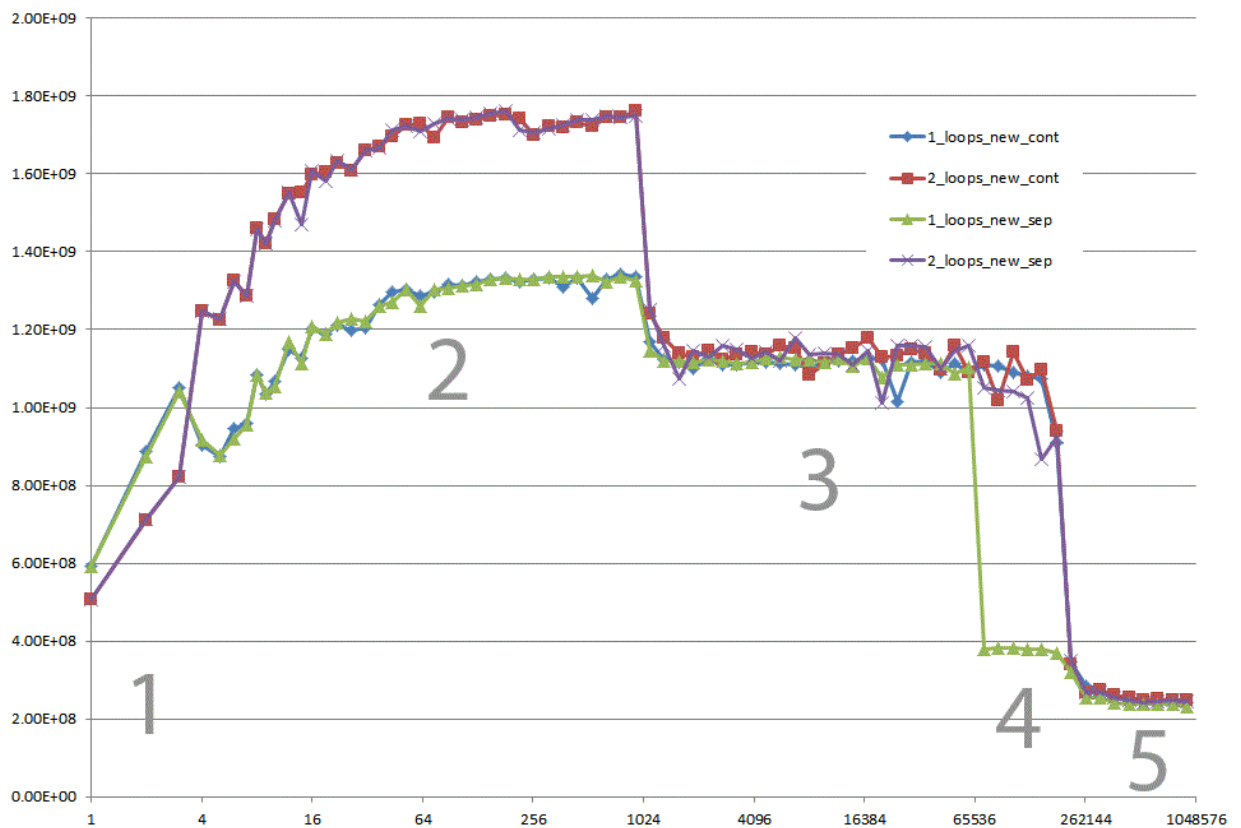
    #ifdef preallocate_memory
        allo(preallocate_memory, nmax);
    #endif

    for (long long n = 1L; n < nmax; n = max(n+1, long long(n*1.2)))
    {
        const long long m = 10000000/n;
        cout << n;

        for (int j = 0; j < 2; j++)
            for (int i = 1; i <= 2; i++)
                cout << s << plain(n, m, j, i);
        cout << endl;
    }
}

```

It shows FLOPS for different values of `n`.



c++ performance x86 vectorization compiler-optimization

Share

Improve this question

Follow

edited Aug 2, 2023 at 18:52



InSync

10.2k ● 4 ● 13 ● 50

asked Dec 17, 2011 at 20:40



Johannes Gerer

25.8k ● 5 ● 30 ● 36

- 4 Could be the operating system which slows while search the physical memory each time you access it and has something like cache in case of secondary access to the same memblock. – AlexTheo Dec 17, 2011 at 20:47
- 9 Are you compiling with optimizations? That looks like a lot of asm code for O2... – Luchian Grigore Dec 17, 2011 at 20:53
- 1 I asked what appears to be a [similar question](#) some time ago. It or the answers might have information of interest. – Mark Wilkins Dec 17, 2011 at 21:10
- 74 Just to be picky, these two code snippets are not equivalent due to potentially overlapping pointers. C99 has the `restrict` keyword for such situations. I don't know if MSVC has something similar. Of course, if this were the issue then the SSE code would not be correct. – user510306 Dec 17, 2011 at 22:28
- 9 This may have something to do with memory aliasing. With one loop, `d1[j]` may alias with `a1[j]`, so the compiler may retract from doing some memory optimisations. While that doesn't happen if you separate the writings to memory in two loops. – rturrado Dec 19, 2011 at 11:50



Upon further analysis of this, I believe this is (at least partially) caused by the data alignment of the four-pointers. This will cause some level of cache bank/way conflicts.

1791



If I've guessed correctly on how you are allocating your arrays, they ***are likely to be aligned to the page line.***



This means that all your accesses in each loop will fall on the same cache way.

However, Intel processors have had 8-way L1 cache associativity for a while. But in reality, the performance isn't completely uniform. Accessing 4-ways is still slower than say 2-ways.

EDIT: It does in fact look like you are allocating all the arrays separately. Usually when such large allocations are requested, the allocator will request fresh pages from the OS. Therefore, there is a high chance that large allocations will appear at the same offset from a page-boundary.

Here's the test code:

```
int main(){
    const int n = 100000;

#ifdef ALLOCATE_SEPERATE
    double *a1 = (double*)malloc(n * sizeof(double));
    double *b1 = (double*)malloc(n * sizeof(double));
    double *c1 = (double*)malloc(n * sizeof(double));
    double *d1 = (double*)malloc(n * sizeof(double));
#else
    double *a1 = (double*)malloc(n * sizeof(double) * 4);
    double *b1 = a1 + n;
    double *c1 = b1 + n;
    double *d1 = c1 + n;
#endif

    // Zero the data to prevent any chance of denormals.
    memset(a1, 0, n * sizeof(double));
    memset(b1, 0, n * sizeof(double));
    memset(c1, 0, n * sizeof(double));
    memset(d1, 0, n * sizeof(double));

    // Print the addresses
    cout << a1 << endl;
    cout << b1 << endl;
    cout << c1 << endl;
    cout << d1 << endl;

    clock_t start = clock();

    int c = 0;
    while (c++ < 10000){

#ifdef ONE_LOOP
```

```

        for(int j=0;j<n;j++){
            a1[j] += b1[j];
            c1[j] += d1[j];
        }
    #else
        for(int j=0;j<n;j++){
            a1[j] += b1[j];
        }
        for(int j=0;j<n;j++){
            c1[j] += d1[j];
        }
    #endif

}

clock_t end = clock();
cout << "seconds = " << (double)(end - start) / CLOCKS_PER_SEC << endl;

system("pause");
return 0;
}

```

Benchmark Results:

EDIT: Results on an *actual* Core 2 architecture machine:

2 x Intel Xeon X5482 Harpertown @ 3.2 GHz:

```

#define ALLOCATE_SEPERATE
#define ONE_LOOP
00600020
006D0020
007A0020
00870020
seconds = 6.206

#define ALLOCATE_SEPERATE
//define ONE_LOOP
005E0020
006B0020
00780020
00850020
seconds = 2.116

//define ALLOCATE_SEPERATE
#define ONE_LOOP
00570020
00633520
006F6A20
007B9F20
seconds = 1.894

//define ALLOCATE_SEPERATE

```

```
//#define ONE_LOOP
008C0020
00983520
00A46A20
00B09F20
seconds = 1.993
```

Observations:

- **6.206 seconds** with one loop and **2.116 seconds** with two loops. This reproduces the OP's results exactly.
- **In the first two tests, the arrays are allocated separately.** You'll notice that they all have the same alignment relative to the page.
- **In the second two tests, the arrays are packed together to break that alignment.** Here you'll notice both loops are faster. Furthermore, the second (double) loop is now the slower one as you would normally expect.

As @Stephen Cannon points out in the comments, there is a very likely possibility that this alignment causes **false aliasing** in the load/store units or the cache. I Googled around for this and found that Intel actually has a hardware counter for **partial address aliasing** stalls:

http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/~amplifierxe/pmw_dp/events/partial_address_alias.html

5 Regions - Explanations

Region 1:

This one is easy. The dataset is so small that the performance is dominated by overhead like looping and branching.

Region 2:

~~Here, as the data sizes increase, the amount of relative overhead goes down and the performance "saturates". Here two loops is slower because it has twice as much loop and branching overhead.~~

I'm not sure exactly what's going on here... Alignment could still play an effect as Agner Fog mentions [cache bank conflicts](#). (That link is about Sandy Bridge, but the idea should still be applicable to Core 2.)

Region 3:

At this point, the data no longer fits in the L1 cache. So performance is capped by the L1 <-> L2 cache bandwidth.

Region 4:

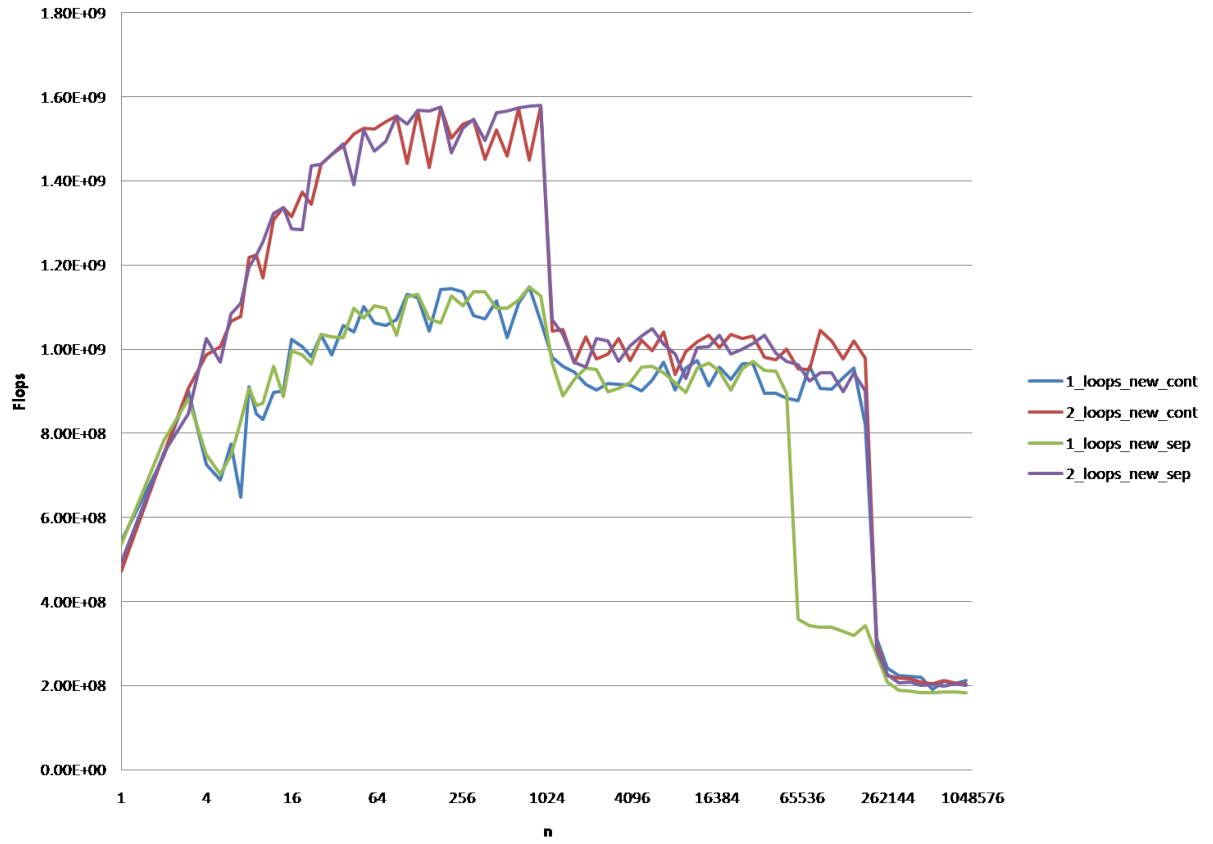
The performance drop in the single-loop is what we are observing. And as mentioned, this is due to the alignment which (most likely) causes **false aliasing** stalls in the processor load/store units.

However, in order for false aliasing to occur, there must be a large enough stride between the datasets. This is why you don't see this in region 3.

Region 5:

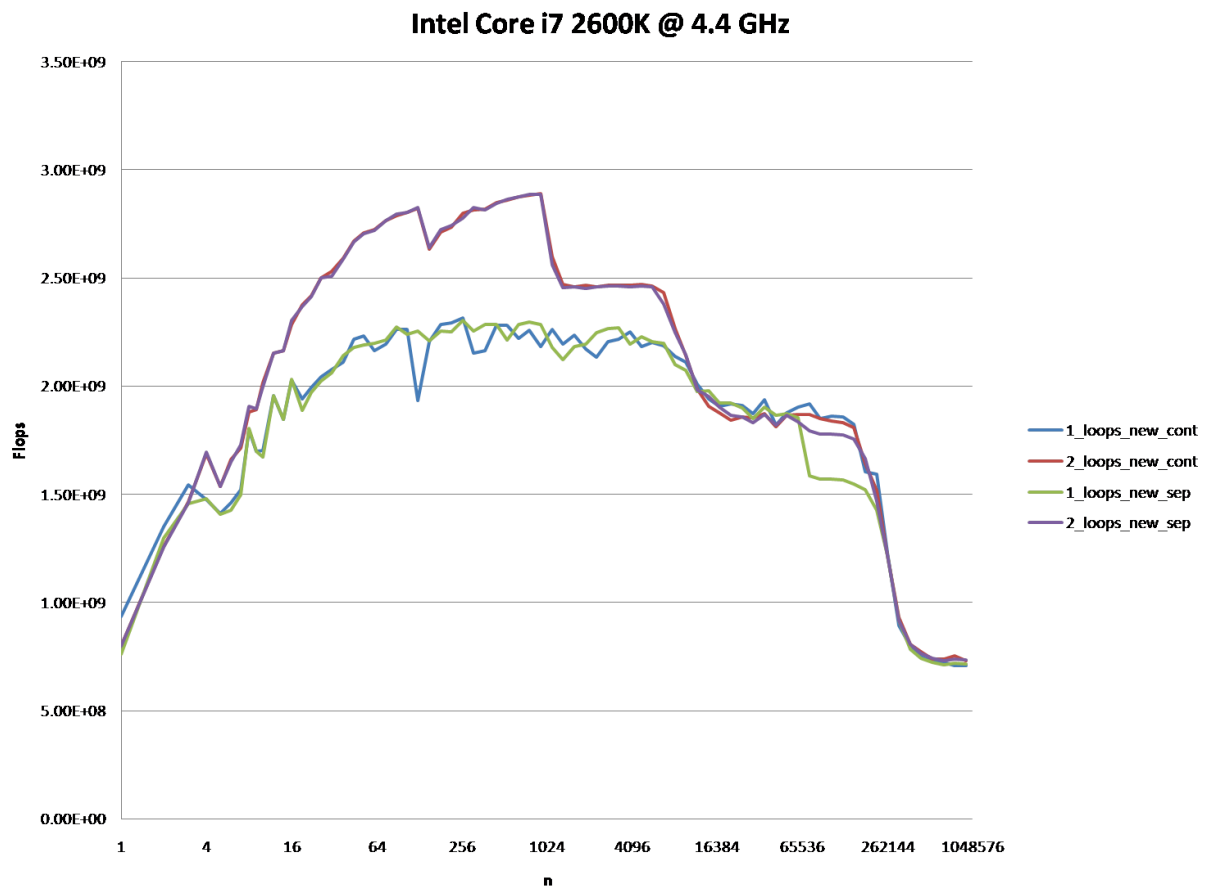
At this point, nothing fits in the cache. So you're bound by memory bandwidth.

2 x Intel Xeon X5482 Harpertown @ 3.2 GHz



Intel Core i7 860 @ 2.8 GHz





Share

Improve this answer

Follow

edited Dec 4, 2020 at 15:27



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Dec 17, 2011 at 21:17



Mysticial

471k ● 46 ● 339 ● 336

183 +1: I think this is the answer. Contrary to what all the other answers say, it's not about the single loop variant inherently having more cache misses, it's about the particular alignment of the arrays causing the cache misses. – [Oliver Charlesworth](#) Dec 17, 2011 at 21:20 ✎

36 This; a *false aliasing* stall is the most likely explanation. – [Stephen Canon](#) Dec 18, 2011 at 1:04



233



OK, the right answer definitely has to do something with the CPU cache. But to use the cache argument can be quite difficult, especially without data.

There are many answers, that led to a lot of discussion, but let's face it: Cache issues can be very complex and are not one dimensional. They depend heavily on the size of the data, so my question was unfair: It turned out to be at a very interesting point in the cache graph.

@Mysticial's answer convinced a lot of people (including me), probably because it was the only one that seemed to rely on facts, but it was only one "data point" of the truth.

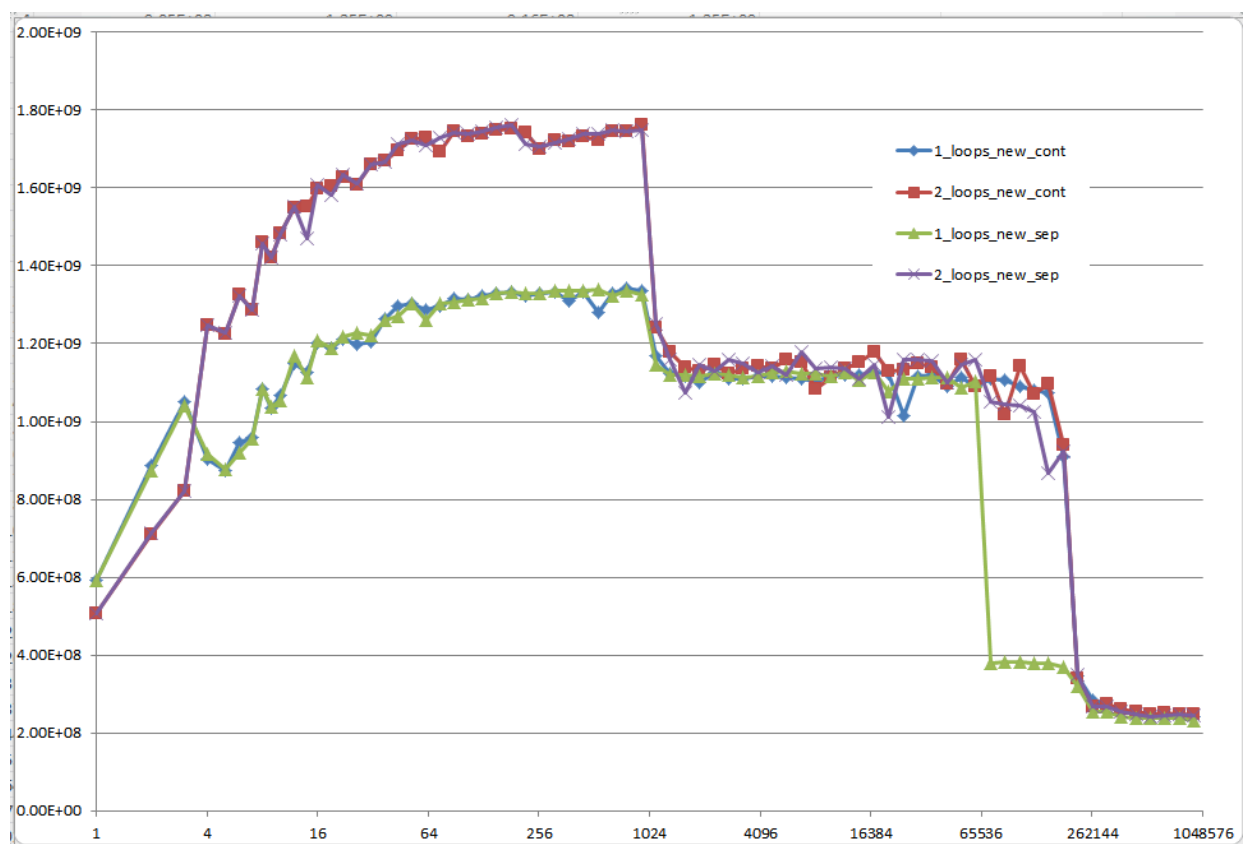
That's why I combined his test (using a continuous vs. separate allocation) and @James' Answer's advice.

The graphs below shows, that most of the answers and especially the majority of comments to the question and answers can be considered completely wrong or true depending on the exact scenario and parameters used.

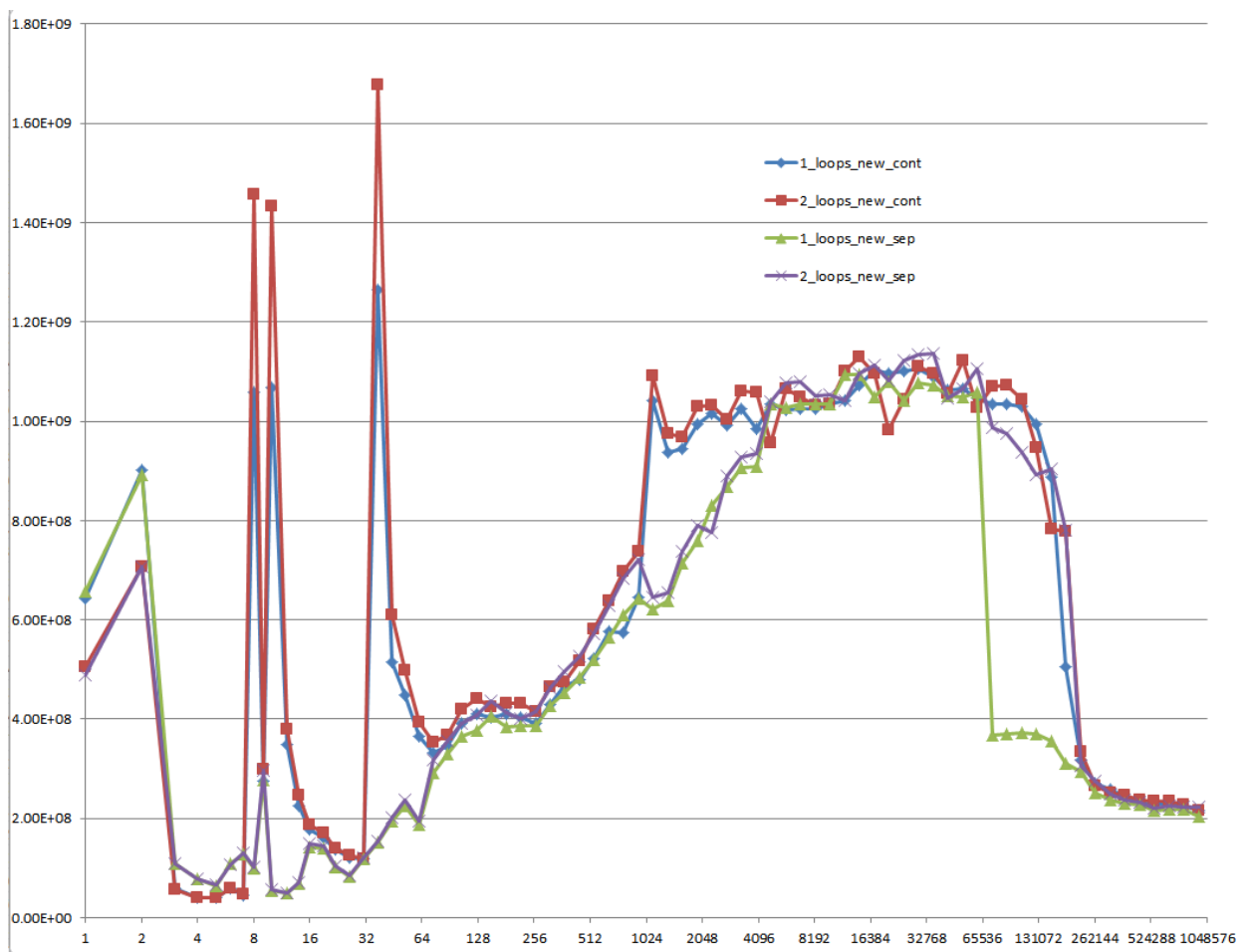
Note that my initial question was at $n = 100.000$. This point (by accident) exhibits special behavior:

1. It possesses the greatest discrepancy between the one and two loop'ed version (almost a factor of three)
2. It is the only point, where one-loop (namely with continuous allocation) beats the two-loop version. (This made Mysticial's answer possible, at all.)

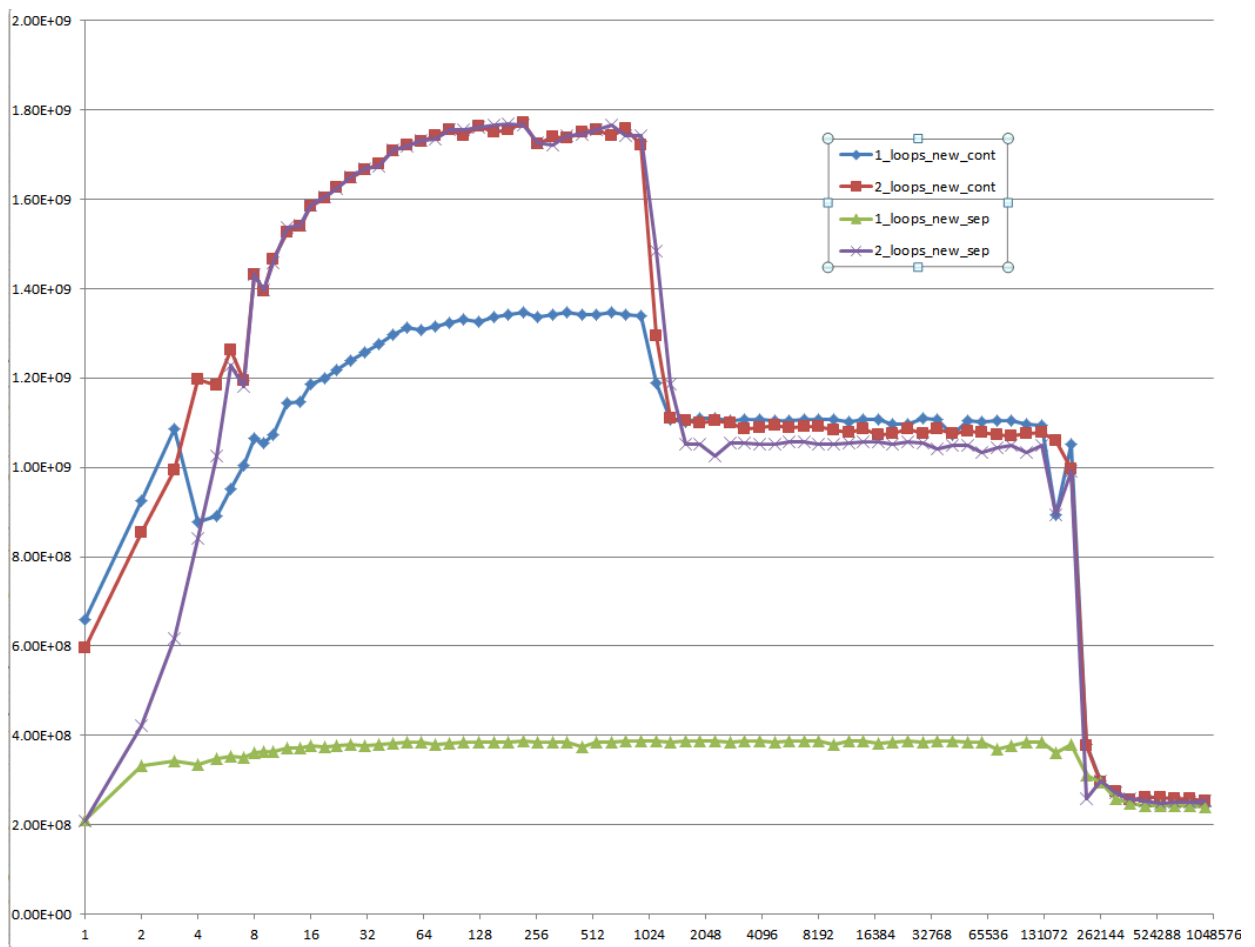
The result using initialized data:



The result using uninitialized data (this is what Mysticial tested):



And this is a hard-to-explain one: Initialized data, that is allocated once and reused for every following test case of different vector size:



Proposal

Every low-level performance related question on Stack Overflow should be required to provide MFLOPS information for the whole range of cache relevant data sizes! It's a waste of everybody's time to think of answers and especially discuss them with others without this information.

Share

Improve this answer

Follow

edited Sep 22, 2016 at 17:12



Ozgur Vatansever

52k ● 17 ● 88 ● 120

answered Dec 18, 2011 at 1:29



Johannes Gerer

25.8k ● 5 ● 30 ● 36

18 +1 Nice analysis. I didn't intend to leave the data uninitialized in the first place. It just happened that the allocator zeroed them anyway. So the initialized data is what matters. I just edited my answer with results on an *actual* Core 2 architecture machine and they are a lot closer to what you are observing. Another thing is that I tested a range of sizes `n` and it shows the same performance gap for `n = 80000`, `n = 100000`, `n = 200000`, etc...
– [Mysticial](#) Dec 18, 2011 at 1:48



85

The second loop involves a lot less cache activity, so it's easier for the processor to keep up with the memory demands.

Share Improve this answer Follow



answered Dec 17, 2011 at 20:47



Puppy

147k ● 40 ● 266 ● 477



48

Imagine you are working on a machine where `n` was just the right value for it only to be possible to hold two of your arrays in memory at one time, but the total memory available, via disk caching, was still sufficient to hold all four.



Assuming a simple LIFO caching policy, this code:



```
for(int j=0;j<n;j++){
    a[j] += b[j];
}
for(int j=0;j<n;j++){
    c[j] += d[j];
}
```

would first cause `a` and `b` to be loaded into RAM and then be worked on entirely in RAM. When the second loop starts, `c` and `d` would then be loaded from disk into RAM and operated on.

the other loop

```
for(int j=0;j<n;j++){
    a[j] += b[j];
    c[j] += d[j];
}
```

will page out two arrays and page in the other two **every time around the loop**. This would obviously be **much** slower.

You are probably not seeing disk caching in your tests but you are probably seeing the side effects of some other form of caching.

There seems to be a little confusion/misunderstanding here so I will try to elaborate a little using an example.

Say `n = 2` and we are working with bytes. In my scenario we thus have **just 4 bytes of RAM** and the rest of our memory is significantly slower (say 100 times longer access).

Assuming a fairly dumb caching policy of *if the byte is not in the cache, put it there and get the following byte too while we are at it* you will get a scenario something like this:

- With

```
for(int j=0;j<n;j++){
    a[j] += b[j];
}
for(int j=0;j<n;j++){
    c[j] += d[j];
}
```

- cache `a[0]` and `a[1]` then `b[0]` and `b[1]` and set `a[0] = a[0] + b[0]` in cache - there are now four bytes in cache, `a[0]`, `a[1]` and `b[0]`, `b[1]`. Cost = $100 + 100$.
- set `a[1] = a[1] + b[1]` in cache. Cost = $1 + 1$.
- Repeat for `c` and `d`.
- Total cost = $(100 + 100 + 1 + 1) * 2 = 404$
- With

```
for(int j=0;j<n;j++){
    a[j] += b[j];
    c[j] += d[j];
}
```

- cache `a[0]` and `a[1]` then `b[0]` and `b[1]` and set `a[0] = a[0] + b[0]` in cache - there are now four bytes in cache, `a[0]`, `a[1]` and `b[0]`, `b[1]`. Cost = 100 + 100.
- eject `a[0]`, `a[1]`, `b[0]`, `b[1]` from cache and cache `c[0]` and `c[1]` then `d[0]` and `d[1]` and set `c[0] = c[0] + d[0]` in cache. Cost = 100 + 100.
- I suspect you are beginning to see where I am going.
- Total cost = $(100 + 100 + 100 + 100) * 2 = 800$

This is a classic cache thrash scenario.

Share

edited Oct 3, 2018 at 11:12

answered Dec 18, 2011 at 1:36

Improve this answer



OldCurmudgeon

65.7k ● 18 ● 125 ● 218

Follow

15 This is incorrect. A reference to a particular element of an array does not cause the entire array to be paged in from disk (or from non-cached memory); only the relevant page or cache line is paged in. – Brooks Moses Dec 18, 2011 at 21:38

Four read streams (two of them also being writes) is pretty much fine on modern CPUs, not significantly worse than two read streams (one of them also being written). HW L2 prefetch on modern Intel CPUs for example can track one forward stream per page. – Peter Cordes Dec 8, 2021 at 23:56



36



It's not because of a different code, but because of caching: RAM is slower than the CPU registers and a cache memory is inside the CPU to avoid to write the RAM every time a variable is changing. But the cache is not big as the RAM is, hence, it maps only a fraction of it.

The first code modifies distant memory addresses alternating them at each loop, thus requiring continuously to invalidate the cache.

The second code don't alternate: it just flow on adjacent addresses twice. This makes all the job to be completed in the cache, invalidating it only after the second loop starts.

Share Improve this answer Follow

answered Dec 17, 2011 at 20:49



Emilio Garavaglia

20.7k ● 3 ● 49 ● 64



21

I cannot replicate the results discussed here.

I don't know if poor benchmark code is to blame, or what, but the two methods are within 10% of each other on my machine using the following code, and one loop is



usually just slightly faster than two - as you'd expect.



Array sizes ranged from 2^{16} to 2^{24} , using eight loops. I was careful to initialize the source arrays so the `+=` assignment wasn't asking the [FPU](#) to add memory garbage interpreted as a double.

I played around with various schemes, such as putting the assignment of `b[j]`, `d[j]` to `InitToZero[j]` inside the loops, and also with using `+= b[j] = 1` and `+= d[j] = 1`, and I got fairly consistent results.

As you might expect, initializing `b` and `d` inside the loop using `InitToZero[j]` gave the combined approach an advantage, as they were done back-to-back before the assignments to `a` and `c`, but still within 10%. Go figure.

Hardware is [Dell XPS 8500](#) with generation 3 [Core i7](#) @ 3.4 GHz and 8 GB memory. For 2^{16} to 2^{24} , using eight loops, the cumulative time was 44.987 and 40.965 respectively. Visual C++ 2010, fully optimized.

PS: I changed the loops to count down to zero, and the combined method was marginally faster. Scratching my head. Note the new array sizing and loop counts.

```
// MemBufferMystery.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"
#include <iostream>
#include <cmath>
#include <string>
#include <time.h>

#define dbl    double
#define MAX_ARRAY_SZ    262145    //16777216    // AKA (2^24)
#define STEP_SZ    1024    // 65536    // AKA (2^16)

int _tmain(int argc, _TCHAR* argv[]) {
    long i, j, ArraySz = 0, LoopKnt = 1024;
    time_t start, Cumulative_Combined = 0, Cumulative_Separate = 0;
    dbl *a = NULL, *b = NULL, *c = NULL, *d = NULL, *InitToOnes = NULL;

    a = (dbl *)calloc( MAX_ARRAY_SZ, sizeof(dbl));
    b = (dbl *)calloc( MAX_ARRAY_SZ, sizeof(dbl));
    c = (dbl *)calloc( MAX_ARRAY_SZ, sizeof(dbl));
    d = (dbl *)calloc( MAX_ARRAY_SZ, sizeof(dbl));
    InitToOnes = (dbl *)calloc( MAX_ARRAY_SZ, sizeof(dbl));
    // Initialize array to 1.0 second.
    for(j = 0; j< MAX_ARRAY_SZ; j++) {
        InitToOnes[j] = 1.0;
    }

    // Increase size of arrays and time
    for(ArraySz = STEP_SZ; ArraySz<MAX_ARRAY_SZ; ArraySz += STEP_SZ) {
        a = (dbl *)realloc(a, ArraySz * sizeof(dbl));
        b = (dbl *)realloc(b, ArraySz * sizeof(dbl));
        c = (dbl *)realloc(c, ArraySz * sizeof(dbl));
        d = (dbl *)realloc(d, ArraySz * sizeof(dbl));
```

```

// Outside the timing loop, initialize
// b and d arrays to 1.0 sec for consistent += performance.
memcpy((void *)b, (void *)InitToOnes, ArraySz * sizeof(dbl));
memcpy((void *)d, (void *)InitToOnes, ArraySz * sizeof(dbl));

start = clock();
for(i = LoopKnt; i; i--) {
    for(j = ArraySz; j; j--) {
        a[j] += b[j];
        c[j] += d[j];
    }
}
Cumulative_Combined += (clock()-start);
printf("\n %6i milliseconds for combined array sizes %i and %i loops",
        (int)(clock()-start), ArraySz, LoopKnt);
start = clock();
for(i = LoopKnt; i; i--) {
    for(j = ArraySz; j; j--) {
        a[j] += b[j];
    }
    for(j = ArraySz; j; j--) {
        c[j] += d[j];
    }
}
Cumulative_Separate += (clock()-start);
printf("\n %6i milliseconds for separate array sizes %i and %i loops
\n",
        (int)(clock()-start), ArraySz, LoopKnt);
}
printf("\n Cumulative combined array processing took %10.3f seconds",
        (dbl)(Cumulative_Combined/(dbl)CLOCKS_PER_SEC));
printf("\n Cumulative separate array processing took %10.3f seconds",
        (dbl)(Cumulative_Separate/(dbl)CLOCKS_PER_SEC));
getchar();

free(a); free(b); free(c); free(d); free(InitToOnes);
return 0;
}

```

I'm not sure why it was decided that MFLOPS was a relevant metric. I thought the idea was to focus on memory accesses, so I tried to minimize the amount of floating point computation time. I left in the `+=`, but I am not sure why.

A straight assignment with no computation would be a cleaner test of memory access time and would create a test that is uniform irrespective of the loop count. Maybe I missed something in the conversation, but it is worth thinking twice about. If the plus is left out of the assignment, the cumulative time is almost identical at 31 seconds each.

Share

edited Dec 16, 2017 at 1:55

answered Dec 30, 2012 at 1:34

Improve this answer



Hovercraft Full Of Eels

285k ● 25 ● 264 ● 385



user1899861

Follow



It's because the CPU doesn't have so many cache misses (where it has to wait for the array data to come from the RAM chips). It would be interesting for you to adjust the

19

size of the arrays continually so that you exceed the sizes of the [level 1 cache](#) (L1), and then the [level 2 cache](#) (L2), of your CPU and plot the time taken for your code to execute against the sizes of the arrays. The graph shouldn't be a straight line like you'd expect.



Share

Improve this answer

Follow

edited Feb 16, 2013 at 9:38



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Dec 17, 2011 at 20:52



James

9,268 ● 3 ● 36 ● 50



The first loop alternates writing in each variable. The second and third ones only make small jumps of element size.

13

Try writing two parallel lines of 20 crosses with a pen and paper separated by 20 cm.

Try once finishing one and then the other line and try another time by writing a cross in each line alternately.



Share

Improve this answer

Follow

edited Feb 16, 2013 at 9:47



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Aug 17, 2012 at 15:23



Guillaume Kiz

443 ● 4 ● 10



The Original Question

6

Why is one loop so much slower than two loops?



Conclusion:

Case 1 is a classic interpolation problem that happens to be an inefficient one. I also think that this was one of the leading reasons why many machine architectures and developers ended up building and designing multi-core systems with the ability to do multi-threaded applications as well as parallel programming.

Looking at it from this kind of an approach without involving how the *hardware*, OS, and *compiler(s)* work together to do heap allocations that involve working with RAM, cache, page files, etc.; the mathematics that is at the foundation of these algorithms shows us which of these two is the better solution.

We can use an analogy of a **Boss** being a **Summation** that will represent a **For Loop** that has to travel between workers **A** & **B**.

We can easily see that **Case 2** is at least half as fast if not a little more than **Case 1** due to the difference in the distance that is needed to travel and the time taken

between the workers. This math lines up almost virtually and perfectly with both the *benchmark times* as well as the number of differences in *assembly instructions*.

I will now begin to explain how all of this works below.

Assessing The Problem

The OP's code:

```
const int n=100000;

for(int j=0;j<n;j++){
    a1[j] += b1[j];
    c1[j] += d1[j];
}
```

And

```
for(int j=0;j<n;j++){
    a1[j] += b1[j];
}
for(int j=0;j<n;j++){
    c1[j] += d1[j];
}
```

The Consideration

Considering the OP's original question about the two variants of the `for` loops and his amended question towards the behavior of caches along with many of the other excellent answers and useful comments; I'd like to try and do something different here by taking a different approach about this situation and problem.

The Approach

Considering the two loops and all of the discussion about cache and page filing I'd like to take another approach as to looking at this from a different perspective. One that doesn't involve the cache and page files nor the executions to allocate memory, in fact, this approach doesn't even concern the actual hardware or the software at all.

The Perspective

After looking at the code for a while it became quite apparent what the problem is and what is generating it. Let's break this down into an algorithmic problem and look at it

from the perspective of using mathematical notations then apply an analogy to the math problems as well as to the algorithms.

What We Do Know

We know is that this loop will run 100,000 times. We also know that `a1` , `b1` , `c1` & `d1` are pointers on a 64-bit architecture. Within C++ on a 32-bit machine, all pointers are 4 bytes and on a 64-bit machine, they are 8 bytes in size since pointers are of a fixed length.

We know that we have 32 bytes in which to allocate for in both cases. The only difference is we are allocating 32 bytes or two sets of 2-8 bytes on each iteration wherein the second case we are allocating 16 bytes for each iteration for both of the independent loops.

Both loops still equal 32 bytes in total allocations. With this information let's now go ahead and show the general math, algorithms, and analogy of these concepts.

We do know the number of times that the same set or group of operations that will have to be performed in both cases. We do know the amount of memory that needs to be allocated in both cases. We can assess that the overall workload of the allocations between both cases will be approximately the same.

What We Don't Know

We do not know how long it will take for each case unless if we set a counter and run a benchmark test. However, the benchmarks were already included from the original question and from some of the answers and comments as well; and we can see a significant difference between the two and this is the whole reasoning for this proposal to this problem.

Let's Investigate

It is already apparent that many have already done this by looking at the heap allocations, benchmark tests, looking at RAM, cache, and page files. Looking at specific data points and specific iteration indices were also included and the various conversations about this specific problem have many people starting to question other related things about it. How do we begin to look at this problem by using mathematical algorithms and applying an analogy to it? We start off by making a couple of assertions! Then we build out our algorithm from there.

Our Assertions:

- We will let our loop and its iterations be a Summation that starts at 1 and ends at 100000 instead of starting with 0 as in the loops for we don't need to worry about the 0 indexing scheme of memory addressing since we are just interested in the algorithm itself.
 - In both cases we have four functions to work with and two function calls with two operations being done on each function call. We will set these up as functions and calls to functions as the following: `F1()` , `F2()` , `f(a)` , `f(b)` , `f(c)` and `f(d)` .
-

The Algorithms:

1st Case: - Only one summation but two independent function calls.

```
Sum n=1 : [1,100000] = F1(), F2();
                F1() = { f(a) = f(a) + f(b); }
                F2() = { f(c) = f(c) + f(d); }
```

2nd Case: - Two summations but each has its own function call.

```
Sum1 n=1 : [1,100000] = F1();
                F1() = { f(a) = f(a) + f(b); }

Sum2 n=1 : [1,100000] = F1();
                F1() = { f(c) = f(c) + f(d); }
```

If you noticed `F2()` only exists in `Sum` from `Case1` where `F1()` is contained in `Sum` from `Case1` and in both `Sum1` and `Sum2` from `Case2` . This will be evident later on when we begin to conclude that there is an optimization that is happening within the second algorithm.

The iterations through the first case `Sum` calls `f(a)` that will add to its self `f(b)` then it calls `f(c)` that will do the same but add `f(d)` to itself for each 100000 iterations. In the second case, we have `Sum1` and `Sum2` that both act the same as if they were the same function being called twice in a row.

In this case we can treat `Sum1` and `Sum2` as just plain old `Sum` where `Sum` in this case looks like this: `Sum n=1 : [1,100000] { f(a) = f(a) + f(b); }` and now this looks like an optimization where we can just consider it to be the same function.

Summary with Analogy

With what we have seen in the second case it almost appears as if there is optimization since both for loops have the same exact signature, but this isn't the real issue. The issue isn't the work that is being done by `f(a)` , `f(b)` , `f(c)` , and `f(d)` . In

both cases and the comparison between the two, it is the difference in the distance that the Summation has to travel in each case that gives you the difference in execution time.

Think of the *for* loops as being the *summations* that does the iterations as being a **Boss** that is giving orders to two people **A** & **B** and that their jobs are to meet **C** & **D** respectively and to pick up some package from them and return it. In this analogy, the *for* loops or summation iterations and condition checks themselves don't actually represent the **Boss**. What actually represents the **Boss** is not from the actual mathematical algorithms directly but from the actual concept of **Scope** and **Code Block** within a routine or subroutine, method, function, translation unit, etc. The first algorithm has one scope where the second algorithm has two consecutive scopes.

Within the first case on each call slip, the **Boss** goes to **A** and gives the order and **A** goes off to fetch **B's** package then the **Boss** goes to **C** and gives the orders to do the same and receive the package from **D** on each iteration.

Within the second case, the **Boss** works directly with **A** to go and fetch **B's** package until all packages are received. Then the **Boss** works with **C** to do the same for getting all of **D's** packages.

Since we are working with an 8-byte pointer and dealing with heap allocation let's consider the following problem. Let's say that the **Boss** is 100 feet from **A** and that **A** is 500 feet from **C**. We don't need to worry about how far the **Boss** is initially from **C** because of the order of executions. In both cases, the **Boss** initially travels from **A** first then to **B**. This analogy isn't to say that this distance is exact; it is just a useful test case scenario to show the workings of the algorithms.

In many cases when doing heap allocations and working with the cache and page files, these distances between address locations may not vary that much or they can vary significantly depending on the nature of the data types and the array sizes.

The Test Cases:

First Case: On first iteration the **Boss** has to initially go 100 feet to give the order slip to **A** and **A** goes off and does his thing, but then the **Boss** has to travel 500 feet to **C** to give him his order slip. Then on the next iteration and every other iteration after the **Boss** has to go back and forth 500 feet between the two.

Second Case: The **Boss** has to travel 100 feet on the first iteration to **A**, but after that, he is already there and just waits for **A** to get back until all slips are filled. Then the **Boss** has to travel 500 feet on the first iteration to **C** because **C** is 500 feet from **A**. Since this **Boss(Summation, For Loop)** is being called right after working with **A** he then just waits there as he did with **A** until all of **C's** order slips are done.

The Difference In Distances Traveled

```
const n = 100000
distTraveledOfFirst = (100 + 500) + ((n-1)*(500 + 500));
// Simplify
distTraveledOfFirst = 600 + (99999*1000);
distTraveledOfFirst = 600 + 99999000;
distTraveledOfFirst = 99999600
// Distance Traveled On First Algorithm = 99,999,600ft

distTraveledOfSecond = 100 + 500 = 600;
// Distance Traveled On Second Algorithm = 600ft;
```

The Comparison of Arbitrary Values

We can easily see that 600 is far less than approximately 100 million. Now, this isn't exact, because we don't know the actual difference in distance between which address of RAM or from which cache or page file each call on each iteration is going to be due to many other unseen variables. This is just an assessment of the situation to be aware of and looking at it from the worst-case scenario.

From these numbers it would almost appear as if *algorithm one* should be 99% slower than *algorithm two*; however, this is only the Boss's part or responsibility of the algorithms and it doesn't account for the actual workers A, B, C, & D and what they have to do on each and every iteration of the Loop. So the boss's job only accounts for about 15 - 40% of the total work being done. The bulk of the work that is done through the workers has a slightly bigger impact towards keeping the ratio of the speed rate differences to about 50-70%

The Observation: - *The differences between the two algorithms*

In this situation, it is the structure of the process of the work being done. It goes to show that **Case 2** is more efficient from both the partial optimization of having a similar function declaration and definition where it is only the variables that differ by name and the distance traveled.

We also see that the total distance traveled in **Case 1** is much farther than it is in **Case 2** and we can consider this distance traveled our *Time Factor* between the two algorithms. **Case 1** has considerable more work to do than **Case 2** does.

This is observable from the evidence of the assembly instructions that were shown in both cases. Along with what was already stated about these cases, this doesn't account for the fact that in **Case 1** the boss will have to wait for both A & C to get back before he can go back to A again for each iteration. It also doesn't account for

the fact that if **A** or **B** is taking an extremely long time then both the **Boss** and the other worker(s) are idle waiting to be executed.

In **Case 2** the only one being idle is the **Boss** until the worker gets back. So even this has an impact on the algorithm.

The OP's Amended Question(s)

EDIT: The question turned out to be of no relevance, as the behavior severely depends on the sizes of the arrays (n) and the CPU cache. So if there is further interest, I rephrase the question:

Could you provide some solid insight into the details that lead to the different cache behaviors as illustrated by the five regions on the following graph?

It might also be interesting to point out the differences between CPU/cache architectures, by providing a similar graph for these CPUs.

Regarding These Questions

As I have demonstrated without a doubt, there is an underlying issue even before the Hardware and Software becomes involved.

Now as for the management of memory and caching along with page files, etc. which all work together in an integrated set of systems between the following:

- *The architecture* (hardware, firmware, some embedded drivers, kernels and assembly instruction sets).
- *The OS* (file and memory management systems, drivers and the registry).
- *The compiler* (translation units and optimizations of the source code).
- And even the *source code* itself with its set(s) of distinctive algorithms.

We can already see that there is a bottleneck that is happening within the first algorithm before we even apply it to any machine with any arbitrary *architecture*, *OS*, and *programmable language* compared to the second algorithm. There already existed a problem before involving the intrinsics of a modern computer.

The Ending Results

However; it is not to say that these new questions are not of importance because they themselves are and they do play a role after all. They do impact the procedures and the overall performance and that is evident with the various graphs and assessments from many who have given their answer(s) and or comment(s).

If you paid attention to the analogy of the **Boss** and the two workers **A** & **B** who had to go and retrieve packages from **C** & **D** respectively and considering the mathematical notations of the two algorithms in question; you can see without the involvement of the computer hardware and software **Case 2** is approximately **60%** faster than **Case 1**.

When you look at the graphs and charts after these algorithms have been applied to some source code, compiled, optimized, and executed through the OS to perform their operations on a given piece of hardware, you can even see a little more degradation between the differences in these algorithms.

If the **Data** set is fairly small it may not seem all that bad of a difference at first. However, since **Case 1** is about **60 - 70%** slower than **Case 2** we can look at the growth of this function in terms of the differences in time executions:

```
DeltaTimeDifference approximately = Loop1(time) - Loop2(time)
//where
Loop1(time) = Loop2(time) + (Loop2(time)*[0.6,0.7]) // approximately
// So when we substitute this back into the difference equation we end up with
DeltaTimeDifference approximately = (Loop2(time) + (Loop2(time)*[0.6,0.7])) -
Loop2(time)
// And finally we can simplify this to
DeltaTimeDifference approximately = [0.6,0.7]*Loop2(time)
```

This approximation is the average difference between these two loops both algorithmically and machine operations involving software optimizations and machine instructions.

When the data set grows linearly, so does the difference in time between the two. Algorithm 1 has more fetches than algorithm 2 which is evident when the **Boss** has to travel back and forth the maximum distance between **A** & **C** for every iteration after the first iteration while algorithm 2 the **Boss** has to travel to **A** once and then after being done with **A** he has to travel a maximum distance only one time when going from **A** to **C**.

Trying to have the **Boss** focusing on doing two similar things at once and juggling them back and forth instead of focusing on similar consecutive tasks is going to make him quite angry by the end of the day since he had to travel and work twice as much. Therefore do not lose the scope of the situation by letting your boss getting into an interpolated bottleneck because the boss's spouse and children wouldn't appreciate it.

Amendment: Software Engineering Design Principles

-- *The difference between local Stack and heap allocated computations within iterative for loops and the difference between their usages, their efficiencies, and effectiveness*
--

The mathematical algorithm that I proposed above mainly applies to loops that perform operations on data that is allocated on the heap.

- **Consecutive Stack Operations:**

- If the loops are performing operations on data locally within a single code block or scope that is within the stack frame it will still sort of apply, but the memory locations are much closer where they are typically sequential and the difference in distance traveled or execution time is almost negligible. Since there are no allocations being done within the heap, the memory isn't scattered, and the memory isn't being fetched through ram. The memory is typically sequential and relative to the stack frame and stack pointer.
- When consecutive operations are being done on the stack, a modern *processor* will cache repetitive values and addresses keeping these values within local cache registers. The time of operations or instructions here is on the order of nano-seconds.

- **Consecutive Heap Allocated Operations:**

- When you begin to apply heap allocations and the processor has to fetch the memory addresses on consecutive calls, depending on the architecture of the CPU, the bus controller, and the RAM modules the time of operations or execution can be on the order of micro to milliseconds. In comparison to cached stack operations, these are quite slow.
- The CPU will have to fetch the memory address from RAM and typically anything across the system bus is slow compared to the internal data paths or data buses within the CPU itself.

So when you are working with data that needs to be on the heap and you are traversing through them in loops, it is more efficient to keep each data set and its corresponding algorithms within its own single loop. You will get better optimizations compared to trying to factor out consecutive loops by putting multiple operations of different data sets that are on the heap into a single loop.

It is okay to do this with data that is on the stack since they are frequently cached, but not for data that has to have its memory address queried every iteration.

This is where software engineering and software architecture design comes into play. It is the ability to know how to organize your data, knowing when to cache your data,

knowing when to allocate your data on the heap, knowing how to design and implement your algorithms, and knowing when and where to call them.

You might have the same algorithm that pertains to the same data set, but you might want one implementation design for its stack variant and another for its heap-allocated variant just because of the above issue that is seen from its $O(n)$ complexity of the algorithm when working with the heap.

From what I've noticed over the years, many people do not take this fact into consideration. They will tend to design one algorithm that works on a particular data set and they will use it regardless of the data set being locally cached on the stack or if it was allocated on the heap.

If you want true optimization, yes it might seem like code duplication, but to generalize it would be more efficient to have two variants of the same algorithm. One for stack operations, and the other for heap operations that are performed in iterative loops!

Here's a pseudo example: Two simple structs, one algorithm.

```
struct A {
    int data;
    A() : data{0}{}
    A(int a) : data{a}{}
};
struct B {
    int data;
    B() : data{0}{}
    A(int b) : data{b}{}
}

template<typename T>
void Foo( T& t ) {
    // Do something with t
}

// Some looping operation: first stack then heap.

// Stack data:
A dataSetA[10] = {};
B dataSetB[10] = {};

// For stack operations this is okay and efficient
for (int i = 0; i < 10; i++ ) {
    Foo(dataSetA[i]);
    Foo(dataSetB[i]);
}

// If the above two were on the heap then performing
// the same algorithm to both within the same loop
// will create that bottleneck
A* dataSetA = new [] A();
B* dataSetB = new [] B();
for ( int i = 0; i < 10; i++ ) {
    Foo(dataSetA[i]); // dataSetA is on the heap here
    Foo(dataSetB[i]); // dataSetB is on the heap here
}
```

```

} // this will be inefficient.

// To improve the efficiency above, put them into separate loops...

for (int i = 0; i < 10; i++ ) {
    Foo(dataSetA[i]);
}
for (int i = 0; i < 10; i++ ) {
    Foo(dataSetB[i]);
}
// This will be much more efficient than above.
// The code isn't perfect syntax, it's only pseudo code
// to illustrate a point.

```

This is what I was referring to by having separate implementations for stack variants versus heap variants. The algorithms themselves don't matter too much, it's the looping structures that you will use them in that do.

Share

edited Dec 8, 2021 at 19:55

answered Jan 30, 2017 at 14:00

Improve this answer



Francis Cugler

7,865 ● 2 ● 35 ● 66

Follow

- 2 Four read streams (two of them also being writes) is pretty much fine on modern CPUs, not significantly worse than two read streams (one of them also being written). HW L2 prefetch on modern Intel CPUs for example can track one forward stream per page. RAM is random-access; "distance travelled" between elements isn't the key factor. It would only be a problem if the cache-line containing `a[i+0..7]` or whatever gets evicted between reading / writing those elements. (Or if the compiler can't see there's no aliasing, so it defeats SIMD vectorization.) – [Peter Cordes](#) Dec 9, 2021 at 0:02
- 1 Stack vs. heap are just different parts of the same virtual address-space, backed by the same cache hierarchy ending with DRAM. [What Every Programmer Should Know About Memory?](#). Touching newly allocated pages on the heap is slow (page faults, see [Idiomatic way of performance evaluation?](#)) but that's actually also true of the stack. It's just that the stack doesn't unmap memory when your function returns so repeatedly calling a function that does `int arr[10000]` only encounters page faults on the first call. – [Peter Cordes](#) Dec 9, 2021 at 0:05 ✎
- 1 There is no "naturally occurring" bottleneck. Hardware / software always matter. You could just as easily argue that you'd naively expect the reduced loop overhead to make loop fusion faster than loop fission. The reason you seem to be basing your argument on is a wrong model of how hardware works. As the accepted answer shows, the actual reason is having 4 read/write streams at the same alignment relative to a page boundary, and thus cache aliasing and possible Intel 4k aliasing effects like a false dependency delaying loads. – [Peter Cordes](#) Dec 9, 2021 at 6:53 ✎
- 1 You're inventing a specific cost-model that isn't random access, and arguing based on that. It's not more fundamental, it's another specific model of computation, and it's not a good model for RAM (random-access memory), or for cache hierarchies with small set-associative caches and DRAM "pages". Your model would predict that `a[i] += 1` would be *vastly* faster than `a[i] += b[i]` because there'd be no seeking at all. But that's not how it performs when compiled and run on real CPUs. It's only the 4k conflicts between two separate write streams that creates this performance pothole. – [Peter Cordes](#) Dec 9, 2021 at 9:44 ✎

- 1 You are fundamentally wrong on this point. It's the same amount of work, differing only in access pattern. Some models of computation tolerate multiple streams of locality, some (like a simple CPU with no cache) don't care about locality *at all*, other than for keeping stuff in registers. The RAM abstract model of computation en.wikipedia.org/wiki/Random-access_machine doesn't have any benefit for locality; it's assumed that there's equal cost for accessing any element of memory at any time in any order. Of course that's a model without cache, but your argument fails there. – [Peter Cordes](#) Dec 9, 2021 at 10:04 ✎



It may be old C++ and optimizations. On my computer I obtained almost the same speed:

3

One loop: 1.577 ms



Two loops: 1.507 ms



I run Visual Studio 2015 on an E5-1620 3.5 GHz processor with 16 GB RAM.



Share

edited Nov 17, 2018 at 16:22

answered Jul 11, 2018 at 7:00

Improve this answer



[Peter Mortensen](#)

31.6k ● 22 ● 109 ● 133



[mathengineer](#)

160 ● 7

Follow



To make this code run fast, the CPU will need to do cache prefetching. Basically the CPU learns that you are accessing sequential data, and reads data from RAM before it is actually needed.

1



The double loop has two input and two output streams, so it needs four separate pre-fetching operations to be fast. The second loops only need two separate pre-fetching operations. If you run this code on a CPU that can prefetch two but not four cache lines automatically, then the first version will be slower.



On an improved CPU the problem would go away. In that case you could change the code to add three arrays to a fourth one, and probably the better CPU can prefetch 4 but not eight streams and will show the exact same effect.



Share Improve this answer Follow

answered Nov 20, 2023 at 9:53



[gnasher729](#)

52.5k ● 5 ● 79 ● 102

-
- 1 As the existing answers show, CPUs at the time could already handle 4 streams. The problem was 4k false aliasing; the same offset-within-page but in different pages, making the CPU think a load was going to reload a recent store, but then after checking the upper address bits it found out that wasn't happening. This is still a problem on modern Intel CPUs with 12 LFBs (like Skylake) or maybe more in later CPUs. Intel's L2 streamer prefetcher can IIRC track one forward and one backward stream per 4k page; at least that was true several years ago, I haven't read if that's changed. – [Peter Cordes](#) Nov 20, 2023 at 10:09 
-
- 1 I mean yeah, if you did run this code on a CPU that could only track 2 prefetch streams, that would also be a problem, but I don't think that's a very relevant concern in 2023, and wasn't the problem at the time. Loops with 6 or 7 streams are usually ok, I think, on Skylake CPUs; Intel's optimization manual recommended 4 as a heuristic target for loop fusion / fission last I checked, which is very conservative. – [Peter Cordes](#) Nov 20, 2023 at 10:12 
-



Highly active question. Earn 10 reputation (not counting the [association bonus](#)) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.