Branchless Programming Companion

E-Book

;CREEL?

Companion Video

This e-book is a companion to the following video:

https://youtu.be/bVJ-mWWL7cE

Branching is Slow

A branch occurs whenever the CPU has to take one of multiple paths. For instance, during an “if” statement where the CPU can take the True path, or the False path. Loops, switches and the condition operator are also branches.

Called “speculative execution”, the CPU tries to load and execute upcoming instructions. When there is a branch, it has to guess which path it will take before the result of the comparison is actually computed. If the path the CPU guesses is not correct, the instructions it speculatively loaded must be flushed, and the correct path must be loaded. This flushing and loading of the correct path take time.

In this text we will explore branchless programming techniques in order to improve the performance of our code. The examples in this text will be in C++ and Assembly language.

Determining the Smaller of Two Integers

Branchless programming is a collection of techniques we can employ in order to eliminate as many branches in our code as possible. It is often not possible to avoid all branches in our code, but the objective is to minimise the number of branches.

Listing 1: Smaller of two integers

```
int Smaller(int a, int b)
{
	if (a < b)
		return a;
	else
		return b;
}
```

Listing 1 shows a standard function for determining the smaller of two integer parameters. When the CPU executes this code, it does not know prior to performing the comparison of “`a < b`” which path it will take – “return a” or “return b”. This means that 50% of the time, it will guess wrong, and have to flush the incorrectly guessed path.

Note: This 50% success rate assumes the smaller of ‘a’ and ‘b’ is approximately randomly distributed. If the smaller is always the ‘a’ variable, the CPU’s prediction logic learn the correct path over time and the performance will improve.

Listing 1 shows regular branching code. This is a natural way to code, and often it is a good choice because the compiler can see what we are trying to do and encode fast branchless machine code for us. We will examine the disassembly of this code shortly.

Listing 2:

```
int Smaller_Branchless(int a, int b)
{
	return (a * (a < b)) + (b * (b <= a));
}
```

Listing 2 shows the same function, only this time it has been programmed without the use of the “if” and “else. This example shows a common technique in branchless programming: Arithmetic with comparison operators.

The comparison operators ‘<’ and ‘<=’ return 0 to mean false, and 1 to mean true.

If ‘a’ is the smaller, the expression will read as:
```
return (a*1)+(b*0);
```

If ‘b’ is the smaller, the expression will read as:
```
return (a*0)+(b*1);
```

The technique employed here is to multiply the results we want to return by the comparisons which would lead to their being returned. The general pattern is as follows:

```
a*(comparison for a)+
	b*(comparison for b)+
	c*(comparison for c)+
	…
```

Sometimes replacing “if” and “else” with this pattern is enough to gain considerable speed. In this particular example, our branchless code runs much slower! Let us look at why.

Compilers and Branchless Programming

Modern compilers are aware of many optimisation techniques, including branchless programming. Sometimes the compiler can recognise the objective of the code and more accurately apply optimisations.

The best way to know how a compiler is optimising our code is to disassemble the compiled binary (or read an Assembly listing from the compiler). If we look at the disassembly from example 1 from the Microsoft C++ compiler, we can see that the compiler replaces the branch with a CMOVL (conditional move instruction).

Listing 3: Disassembly of listing 1

```
mov          ebx,dword ptr [b]
cmp          dword ptr [a],ebx
cmovl        ebx,dword ptr [a]
```

I have removed most of the code from the disassembly in listing 3, such that only the comparison is shown. This code is very fast, it moves the value of the ‘b’ variable into the register EBX. Then it compares the value to the ‘a’ variable using the CMP instruction. The CMOVL instruction will overwrite EBX with ‘a’ if it is less. CMOVL means Conditionally Move if Less. CMOVcc, the conditional move instructions are branchless and very fast.

So we can see that the compiler was able to understand what we were trying to do and use branchless programming itself.

Listing 4: Disassembly of Listing 2

```
mov          ecx,dword ptr [a] ; Copy A to ECX
xor          r8d,r8d           ; Set R8D to 0
mov          eax,dword ptr [b] ; Copy B to EAX
mov          ebx,r8d           ; Set EBX to 0
cmp          eax,ecx           ; Compare B and A
setle        bl                ; Set BL to 1 if B is less
imul         ebx,eax           ; Multiply B by 1 or 0
cmp          ecx,eax           ; Compare A and B
setl         r8b               ; Set R8B to 1 if A is less
imul         r8d,ecx           ; Multiply A by 1 or 0
add          ebx,r8d           ; Add results
```

Listing 4 shows the disassembly of Listing 2. We tried to use branchless techniques, but this has resulted in much slower Assembly. The compiler was not able to understand what we were trying to do.

The comments were added to listing 4 to explain the logic of the instructions. The resulting code is a fairly direct translation of our branchless C++. This code is also branchless, but it will most likely execute slower.

Listings 3 and 4 show that it is not always beneficial to use branchless C++. Compilers are designed to recognise simple, obvious code – they can best optimise when the objective of our code is clear.

Note: Times will differ wildly between CPU manufacturers and generations of hardware. For instance the reciprocal throughput of the CMOVcc instructions on AMD K10 is 1/3. It is 1/4 for Ryzen. It is 1 for Intel Sandy Bridge, and 1/2 for Skylake. I was using a Skylake CPU to record the numbers in this text. The throughput figures were taken from Agner Fog’s Instruction Table Manual: https://www.agner.org/optimize/instruction_tables.pdf

Branchless Patterns – Sign Masks

We have seen one branchless pattern already - multiplication by the result from comparison operators. Another very common pattern is to manipulate data based on shifting the sign bit of an integer. For instance, consider listing 5.

Listing 5:

```
int abs(int a) {
   int s = a >> 31; // cdq, signed shift, -1 if negative, else 0
   a ^= s;  // ones' complement if negative
   a -= s;  // + one if negative -> two's complement if negative
   return a;
}
```

Source:
https://www.chessprogramming.org/Avoiding_Branches#Absolute_value_of_an_Integer

Listing 5 shows some interesting code to find the absolute value of an integer. The obvious way to do this would be to use an “if” statement - check if the value is negative, and compute the positive version if it is. But listing 5 shows a branchless version.

Note: The code from Listing 5 exploits 2’s complement. 2’s complement is the way most computing devices store signed integers. For more information on how 2’s complement works, see: https://en.wikipedia.org/wiki/Two%27s_complement.

First, we shift the integer right 31 bits, storing the result in a variable “s”. If “a” is positive, then the sign bit will be 0, and this shift will mean “s=0”. But if “a” is negative, then its sign bit will be “1”, and this shift will result in “s=-1”; “s” will be filled with 1’s in binary. In other words, we are smearing or smudging the sign bit from the “a” variable across the entire “s” variable.

Next, the variable “a” is XOR’d with the “s” variable. If “a” was positive, this XOR will do nothing, since “a^0=a”. But, if “a” was negative, then this XOR will result in the 1’s complement of “a”. That is, all of the bits of “a” will be flipped – any 1’s will become 0, and any 0’s will become 1.

Finally, to get the 2’s complement from the 1’s complement of “a”, we need to add 1. Listing 5 adds by subtracting. If “a” was negative, “s” will be -1, and “a- -1” is the same as “a+1”. And if “a” was positive, then this subtraction will become “a-0”, which will not change “a”.

This pattern of smudging or copying a sign bit across a variable, and then using boolean and arithmetic operations to compute a condition without branching is very common in branchless programming.

To Upper

Let us now consider a more complex example. We want to write a function which takes a string of ASCII characters and converts all lower case letters to upper case.

The ASCII characters are designated such that upper case letters have the codes from 65 to 90 for ‘A’ to ‘Z’, and the lower case letters have the codes from 97 to 122 for ‘a’ to ‘z’. So the lower case letters are 32 above their upper case counterparts, e.g. ‘A’ is 65 while ‘a’ is 97, or 65+32. This means we can subtract 32 from any lower case letter, and we will have the uppercase version!

Note: For the following timings I ran the ToUpper fuctions 1,000,000 times, with 1024 characters of lower and upper case letters. The entire array of characters will be in the L1 cache, and there should be minimal cache misses. For a complete listing, see the appendix.

Listing 6: To Upper C++

```
void ToUpperCPP(char* d, int count)
{
     for (int i = 0; i < count; i++)
     {
          if (d[i] >= 'a' && d[i] <= 'z')
               d[i] -= 32;
     }
}
```

Listing 6 shows some basic C++ code for a ToUpper function. We subtract 32 from the elements of `d[i]` if they fall within the range of lower case letters, ‘a’ to ‘z’ inclusive. If they do not fall within this range, then they are either an upper case letter, or they are some other character, a digit or punctuation mark, etc.

Listing 7: Disassembly of ToUpperCPP

```
movzx        edx,byte ptr [rax-1]
lea          ecx,[rdx-61h]
cmp          cl,19h
ja           main+0E2h (07FF7523410E2h)
sub          dl,20h
mov          byte ptr [rax-1],dl
movzx        edx,byte ptr [rax]
lea          ecx,[rdx-61h]
cmp          cl,19h
ja           main+0F2h (07FF7523410F2h)
sub          dl,20h
mov          byte ptr [rax],dl
movzx        edx,byte ptr [rax+1]
lea          ecx,[rdx-61h]
cmp          cl,19h
ja           main+104h (07FF752341104h)
sub          dl,20h
```

```
mov         byte ptr [rax+1],dl
movzx       edx,byte ptr [rax+2]
lea         ecx,[rdx-61h]
cmp         cl,19h
ja          main+116h (07FF752341116h)
sub         dl,20h
mov         byte ptr [rax+2],dl
add         rax,4
sub         r8,1
jne         main+0D0h (07FF7523410D0h)
```

Listing 7 shows the assembly language generated by the Microsoft C++ compiler when it compiles listing 6. We do not need to examine this code in depth to see that it contains multiple branches (highlighted in `yellow`). The `JA` instruction means "Jump if Above", it is a conditional branch. This code takes around 3330ms to complete in the benchmark. Let us see what we can do with some branchless techniques.

Note: The benchmark contains two options for timing, using Ctime's Clock function, and using the RDTSC time stamp (with the TimeStamp function). The clock reports in milliseconds, while the TimeStamp reports in CPU cycles. Reading the cycles will not take into account thermal throttling of the CPU.

Listing 8: Branchless Version

```
void ToUpper_Branchless(char* d, int count)
{
     for (int i = 0; i < count; i++)
     {
          d[i] = (d[i] * !(d[i] >= 'a' && d[i] <= 'z')) +
                    (d[i] - 32) * (d[i] >= 'a' && d[i] <= 'z');
     }
}
```

Listing 8 shows a branchless version of the ToUpper function. This code runs around 3 times faster than that of listing 6, with a time of 1026ms, compared to the original 3330ms for the regular C++ code.

The code presents an expression which can be broken into two halves, the first part "`(d[i] * !(d[i] >= 'a' && d[i] <= 'z'))`" says to set d[i] to the value of d[i] if the current value is not within the range of 'a' to 'z'. In other words, do not change d[i] if it is not a lower case letter. The second part "`(d[i] - 32) * (d[i] >= 'a' && d[i] <= 'z')`" says to subtract 32 from d[i] if it is in this range.

Note: If the original C++ code executes and the data is almost all uppercase already, then the original C++ code runs very fast! Around 559ms. This is due to the branch predictor. If the branch predictor can accurately predict which branch the CPU will take, then the penalty for branching is generally far smaller.

Listing 9: Remove half of the branchless Expression

```
void ToUpper_Branchless2(char* d, int count)
{
     for (int i = 0; i < count; i++)
     {
          d[i] -= 32 * (d[i] >= 'a' && d[i] <= 'z');
     }
}
```

Listing 9 runs in approximately 471ms. We only subtract for lower case, subtracting 0 if the value of d[i] is not lower case. This roughly doubles the performance with listing 9 running at around 7 times faster than the original C++.

Assembly Language

Before we explore SIMD instructions in Assembly language, it may be beneficial to look at Assembly code for some simpler versions of the function.

Listing 10: Assembly Language ToUpper Branching

```
ToUpperASM1 proc
LoopHead:
     mov al, byte ptr [rcx]
     cmp al, 'z'
     jg NotLowerCase
     cmp al, 'a'
     jl NotLowerCase
     sub al, 32     ; Lower case!
NotLowerCase:
     mov byte ptr [rcx], al
     inc rcx
     dec edx
     jnz LoopHead
     ret
ToUpperASM1 endp
```

Listing 10 shows a simple branching version of the ToUpper algorithm in Assembly language. This code branches, and it will be slow, but it is a good start.

This code reads a letter from the array. It then compares the letter to ‘z’, jumping to the label ‘NotLowerCase’ if the letter is greater than ‘z’. Otherwise, it compares the letter to ‘a’, again jumping to the same label if the letter is less than ‘a’.

If the letter is neither greater than ‘z’ nor less than ‘a’, then it is lower case; we subtract 32 from it, and store the result.

The running time for this Assembly version is around 3605ms. This is slower than the original C++ by a considerable margin. This shows that the speed gains from Assembly language are not free. We can often gain a lot of speed from Assembly, but we do have to use it carefully.

Let us code a version using the branchless conditional move instructions, CMOVcc.

Listing 11: Branchless Assembly using CMOVcc

```
ToUpperASM2 proc
LoopHead:
     mov al, byte ptr [rcx]   ; Read a character into AL
     mov r8d, -1    ; Fill R8D with 1's
     mov r9d, 0     ; Fill R9 with 0's
     cmp al, 'z'
     cmovg r8d, r9d ; Clear R8D if x > 'z'
     cmp al, 'a'
     cmovl r8d, r9d ; Clear R8D if x < 'a'
     mov r9d, eax   ; Copy the byte to R9
     sub r9d, 32    ; Subtract 32 from R9’s copy
     cmp r8d, 0     ; Compare R8D to 0
     cmove eax, r9d ; If R8D = 0, overwrite EAX with uppercase
     mov byte ptr [rcx], al   ; store the result
     inc rcx
     dec edx
     jnz LoopHead
     ret
ToUpperASM2 endp
```

Listing 11 shows a branchless assembly version of the algorithm. We use the Conditional Move instructions, CMOVE and CMOVG. Letters are read from the array and compared to ‘z’ and ‘a’. If they fall within the range, we Zero R8D, otherwise, R8D is -1. We subtract 32 from a copy of the character we read, and if the value in R8D is 0, we overwrite the original byte with the Uppercase version.

The time for Listing 11 is 1477ms. This is faster than the original C++, but it is slower than the branchless C++ versions, which ran at 1026ms and 470ms for the two versions we considered in listings 8 and 9.

SIMD Branchless Programming

Branchless programming goes hand in hand with SIMD. In the final listing we employ SIMD using AVX registers. AVX offers 32-way SIMD when dealing with bytes, and the algorithm is also branchless.

Listing 12: SIMD AVX

```
ToUpperASM3 proc
       push rbx      ; Save caller's RBX

       mov eax, edx  ; Copy the count to EAX
       and eax, 31   ; Compute count%31, the residuals for AVX
       shr edx, 5    ; Divide the count by 32 because AVX performs 32 byte ops/instruction

       jz ResidualsLoop     ; If this results in 0 AVX loops, jump to the residuals loop

       vpcmpeqb ymm3, ymm3, ymm3   ; Set to all 1's for flipping with XOR

       vmovdqa ymm6, ymmword ptr [thirty_two]    ; Fill YMM6 with 32
       vmovdqa ymm7, ymmword ptr [gt_eq_a_array]; Fill YMM7 with 'a'-1
       vmovdqa ymm8, ymmword ptr [z_array]      ; Fill YMM8 with 'z'

LoopHead:
       vmovdqu ymm5, ymmword ptr [rcx]    ; Read the data into YMM5

       vpcmpgtb ymm0, ymm5, ymm7   ; YMM0 = Where is Src > 'a'-1(greater or equal to 'a')?

       vpcmpgtb ymm1, ymm5, ymm8   ; Where is Src > 'z'?
       vpxor ymm1, ymm1, ymm3      ; Complement these, so we get <= 'z'

       vpand ymm0, ymm0,ymm1       ; AND the two conditions together, so YMM0
                                   ; will have 11111111 for all lower case letters
                                   ; in YMM5, and 00000000 for all other characters

       vmovdqa ymm4, ymm5          ; Copy the original characters to YMM4, we will use
                                   ; YMM4 for all lower case letters, and YMM5 for the rest

       vpxor ymm2, ymm0, ymm3      ; Complement the lower mask (YMM0), store in YMM2
                                   ; This gives us a mask of all that aren't lower case
       vpand ymm5, ymm5, ymm2      ; AND This with YMM5, so only the chars that were not
                                   ; lower case will remain, all else become 00000000

       vpsubb ymm4, ymm4, ymm6     ; Subtract 32 from all the characters in YMM4
       vpand ymm0, ymm0, ymm4      ; AND XMM4 with our lower case mask, so only the lower
                                   ; characters remain, the rest become 00000000

       vpor  ymm0, ymm0, ymm5      ; Join the characters from YMM4 and YMM5

       vmovdqu ymmword ptr [rcx], ymm0    ; Store

       add rcx, 32          ; Move the data pointer up to the next 32 bytes

       dec edx
       jnz LoopHead

       ; There may be residuals, AVX deals with 32 bytes at once, so there
       ; could be 31 residuals present at the end of the array.
       and eax, eax  ; See what's in EAX, the residuals
       jz Finished   ; If it's Zero, then there's no residuals

       ; Oterwise we begin a scalar loop to convert any residual bytes
ResidualsLoop:
       mov bl, byte ptr [rcx]      ; Read a byte into BL

       mov r8d, -1   ; Fill R8D with 1's
       mov r9d, 0    ; Fill R9 with 0's

       cmp bl, 'z'
       cmovg r8d, r9d       ; Clear R8D if x > 'z'
```

```
        cmp bl, 'a'
        cmovg r8d, r9d       ; Clear R8D if x < 'a'

        mov r9d, ebx         ; Copy the byte
        sub r9d, 32          ; Subtract 32 from the copy
        cmp r8d, 0           ; Check if the byte was lower case?
        cmove ebx, r9d       ; If it was, overwrite the original with the converted byte

        mov byte ptr [rcx], bl      ; Store the result
        inc rcx                     ; Move up to the next byte

        dec eax
        jnz ResidualsLoop

Finished:
        pop rbx       ; Restore the caller's RBX
        ret
ToUpperASM3 endp
```

Listing 12 runs at around 80 to 100ms. This is a very good improvement over the original C++, which was about 3330ms. The total gains here are about 25 to 40 times faster.

Note: Listing 12 is so fast that it is difficult to time. Almost all of the time recorded being from randomising of the data. I have recorded speeds of anything from 30x faster than the original C++ up to 200x faster. The real performance gains seem to be around 25 to 40 times faster.

It is not always possible to get such speed gains from AVX, but in this instance, we were able to get a really good speed up because the original C++ code was branching and the compiler was not auto-vectorising our code.

Conclusion

Branchless programming can greatly improve the speed of code in high level languages, like C++, or low level languages. The total speed gains from branchless programming are often difficult to estimate without examining exactly what the compiler is doing, to ascertain if the compiler is already using branchless programming techniques.

The times for these algorithms may vary a lot depending on the compiler being used, the optimisation settings of that compiler, as well as the hardware running the code.

Branchless SIMD instructions are very fast. If the compiler has not vectorised an algorithm, then a non-branching vectorised version will most likely be extremely fast in comparison. This is especially true when we vectorise algorithms which operate on smaller data types, short integers or bytes.

Branchless techniques will not always help, but they are an interesting study, and when they do help, they can improve our code greatly!

;Creel?

Appendix: Source Code

The source code consists of two files – main.cpp and asm.asm. I compiled for x64 CPU's in Release mode with the instruction set set to AVX2, and with floating point set to fast. I used the Microsoft C++ compiler and Visual Studio 2019 Community.

main.cpp

```cpp
#include <iostream>
#include <ctime>

// George Marsaglia RNG
unsigned int xorState = 12384;
 unsigned int XORRand()
{
	xorState ^= xorState << 13;
	xorState ^= xorState >> 17;
	xorState ^= xorState << 5;
	return xorState;
}

 // Output function to ensure the strings are beging converted
 void PrintString(char* data, int count)
 {
	 for (int i = 0; i < count; i++)
		 std::cout << data[i];

	 std::cout << std::endl;
 }

 // Generates a string of random letters in lower and upper
void RandomizeData(char* d, int count)
{
	for (int i = 0; i < count; i++)
	{
		d[i] = XORRand() % 26 + 'A';

		d[i] += 32 * (XORRand() & 1);
	}
 }

void ToUpperCPP(char* d, int count)
{
	for (int i = 0; i < count; i++)
	{
		if (d[i] >= 'a' && d[i] <= 'z')
			d[i] -= 32;
	}
}

void ToUpper_Branchless(char* d, int count)
{
	for (int i = 0; i < count; i++)
	{
		d[i] = d[i] * !(d[i] >= 'a' && d[i] <= 'z') +
			(d[i] - 32) * (d[i] >= 'a' && d[i] <= 'z');
	}
}

void ToUpper_Branchless2(char* d, int count)
```

```
{
       for (int i = 0; i < count; i++)
       {
              d[i] -= 32 * (d[i] >= 'a' && d[i] <= 'z');
       }
}

// Read the time stamp so CPU hertz aren't governing times
extern "C" unsigned long long TimeStamp();

extern "C" void ToUpperASM1(char* data, int count);
extern "C" void ToUpperASM2(char* data, int count);
extern "C" void ToUpperASM3(char* data, int count);

unsigned long GetTime()
{
       //return TimeStamp();
       return clock();
}

int main()
{
       int COUNT = 1024;
       char* data = new char[COUNT];

       double fastestCPP = 0.0;
       double averageRandomTime = 0.0;
       double fastestRandomize = 0.0;

       double runs = 10;           // Number of runs
       double iterations = 1000000;// Number of times the algorithm is repeated per run

       void (*fn_ptr)(char* data, int count) = ToUpperCPP;
       RandomizeData(data, COUNT);
       std::cout << "Getting C++ regular benchmark time..." << std::endl;
       for (int p = 0; p < runs; p++)
       {
              unsigned long long startTime = GetTime();
              for (int r = 0; r < iterations; r++)
              {
                     RandomizeData(data, COUNT);
                     fn_ptr(data, COUNT);
              }

              unsigned long long  finishTime = GetTime();

              std::cout << "Run " << (p+1) << " of "<< runs
                     <<", Time for both functions: " << (finishTime - startTime)
                     << std::endl;

              if (fastestCPP > (finishTime - startTime) || fastestCPP == 0.0)
              {
                     fastestCPP = finishTime - startTime;
              }
       }

       std::cout << "Done benching C++. Fastest time: " << fastestCPP
                     << "cycles" << std::endl;

       std::cout << "Getting time to randomize..." << std::endl;
       for (int p = 0; p < runs; p++)
       {
              unsigned long long  startTime = GetTime();
              for (int r = 0; r < iterations; r++)
              {
                RandomizeData(data, COUNT);
        }

        unsigned long long  finishTime = GetTime();

        std::cout << "Run " << (p + 1) << " of "<< runs <<", Time: "
                << (finishTime - startTime) << std::endl;

        averageRandomTime += (finishTime - startTime);

        if (fastestRandomize > (finishTime - startTime) || fastestRandomize == 0.0)
        {
                fastestRandomize = finishTime - startTime;
        }
}

averageRandomTime /= runs;

std::cout << "Done getting time to randomize. Fastest is: "
                << fastestRandomize << "cycles" << std::endl;

fastestCPP -= fastestRandomize;

std::cout << "Fastest C++ time without Randomize function: "
                << fastestCPP << std::endl;

while (true)
{
        std::cout << "1. C++ Branching" << std::endl;
        std::cout << "2. C++ Branchless 1" << std::endl;
        std::cout << "3. C++ Branchless 2" << std::endl;
        std::cout << "4. ASM Branching" << std::endl;
        std::cout << "5. ASM Branchless 1" << std::endl;
        std::cout << "6. ASM Branchless 2" << std::endl;

        int option;   // The user's selected code
        std::cin >> option;

        if (option < 1 || option > 6)
                continue;

        double fastest = 0.0;

        switch (option)
        {
        case 1:
                fn_ptr = ToUpperCPP;
                break;
        case 2:
                fn_ptr = ToUpper_Branchless;
                break;
        case 3:
                fn_ptr = ToUpper_Branchless2;
                break;
        case 4:
                fn_ptr = ToUpperASM1;
                break;
        case 5:
                fn_ptr = ToUpperASM2;
                break;
        case 6:
                fn_ptr = ToUpperASM3;
                break;
        default:
                std::cout << "Select 1 to 6" << std::endl;
```

```
                    break;
                }

                for (int p = 0; p < runs; p++)
                {
                    unsigned long long  startTime = GetTime();

                    for (int r = 0; r < iterations; r++)
                    {
                        RandomizeData(data, COUNT);

                        fn_ptr(data, COUNT);
                    }

                    unsigned long long  finishTime = GetTime();

                    std::cout << "Run " << (p + 1) << " of " << runs
                    << ", Time: " << (finishTime - startTime) << std::endl;

                    if (fastest > (finishTime - startTime) || fastest == 0.0)
                    {
                        fastest = finishTime - startTime;
                    }
                }

                std::cout << "Fastest: " << fastest << std::endl;
                std::cout << "Minus time to randomize: " << (fastest-fastestRandomize)
                    << std::endl;
                std::cout << "Ratio to C++: " << (fastestCPP / (fastest-fastestRandomize))
                    << std::endl;
        }

        std::cin.get();

        return 0;
}
```

asm.asm

```
.data
z_array db 32 dup('z')
gt_eq_a_array db 32 dup('a'-1)
thirty_two db 32 dup(32)

.code
TimeStamp proc
        rdtsc
        shl rdx, 32
        add rax, rdx
        ret
TimeStamp endp

ToUpperASM1 proc

LoopHead:
        mov al, byte ptr [rcx]

        cmp al, 'z'
        jg NotLowerCase
        cmp al, 'a'
        jl NotLowerCase
```

```
       ; It's a lower case letter!
       sub al, 32
       mov byte ptr [rcx], al

NotLowerCase:
       inc rcx

       dec edx
       jnz LoopHead
       ret
ToUpperASM1 endp

ToUpperASM2 proc
LoopHead:
       mov al, byte ptr [rcx]

       mov r8d, -1   ; Fill R8D with 1's
       mov r9d, 0    ; Fill R9 with 0's

       cmp al, 'z'
       cmovg r8d, r9d       ; Clear R8D if x > 'z'

       cmp al, 'a'
       cmovg r8d, r9d       ; Clear R8D if x < 'a'

       mov r9d, eax
       sub r9d, 32
       cmp r8d, 0
       cmove eax, r9d

       mov byte ptr [rcx], al
       inc rcx

       dec edx
       jnz LoopHead

       ret
ToUpperASM2 endp

ToUpperASM3 proc
       push rbx      ; Save caller's RBX

       mov eax, edx  ; Copy the count to EAX
       and eax, 31   ; Compute the remainder after division of EAX and 31, residuals for AVX
       shr edx, 5    ; Divide count by 32 because AVX performs 32 byte ops per instruction

       jz ResidualsLoop; If this results in 0 AVX loops, jump to th4e residuals loop

       vpcmpeqb ymm3, ymm3, ymm3; Set to all 1's for flipping with XOR

LoopHead:
       vmovdqu ymm5, ymmword ptr [rcx]    ; Read the data into YMM5

       ; YMM0 = Where is Src > 'a'-1      (greater or equal to 'a')?
       vpcmpgtb ymm0, ymm5, ymmword ptr [gt_eq_a_array]

       vpcmpgtb ymm1, ymm5, ymmword ptr [z_array]; Where is Src > 'z'?
       vpxor ymm1, ymm1, ymm3             ; Complement these, so we get <= 'z'

       vpand ymm0, ymm0,ymm1       ; AND the two conditions together, so YMM0
                                   ; will have 11111111 for all lower case letters
                                   ; in YMM5, and 00000000 for all other characters

       vmovdqa ymm4, ymm5          ; Copy the original characters to YMM4, we will use
                                   ; YMM5 for all lower case letters, and YMM5 for the rest
```

```
        vpxor ymm2, ymm0, ymm3; Complement the lower mask (YMM0), store in YMM2
                             ; This gives us a mask of all letters that aren't lower case
        vpand ymm5, ymm5, ymm2      ; AND mask with YMM5, so only characters that were not
                                    ; lower case will remain, and all else become 00000000

        ; Subtract 32 from all the characters in YMM4
        vpsubb ymm4, ymm4, ymmword ptr [thirty_two]
        vpand ymm0, ymm0, ymm4; AND XMM4 with our lower case mask, so that only the lower
                             ; characters will remain, the rest will become 00000000

        vpor  ymm0, ymm0, ymm5             ; Join the remaining characters from YMM4 and YMM5

        vmovdqu ymmword ptr [rcx], ymm0    ; Store

        add rcx, 32          ; Move the data pointer up to the next 32 bytes

        dec edx
        jnz LoopHead

        ; There may be residuals, AVX deals with 32 bytes at once, so there
        ; could be 31 residuals present at the end of the array.
        and eax, eax  ; See what's in EAX, the residuals
        jz Finished          ; If it's Zero, then there's no residuals

        ; Otherwise we begin a scalar loop to convert any residual bytes
ResidualsLoop:
        mov bl, byte ptr [rcx]; Read a byte into BL

        mov r8d, -1          ; Fill R8D with 1's
        mov r9d, 0           ; Fill R9 with 0's

        cmp bl, 'z'
        cmovg r8d, r9d       ; Clear R8D if x > 'z'

        cmp bl, 'a'
        cmovg r8d, r9d       ; Clear R8D if x < 'a'

        mov r9d, ebx         ; Copy the byte
        sub r9d, 32          ; Subtract 32 from the copy
        cmp r8d, 0           ; Check if the byte was lower case?
        cmove ebx, r9d       ; If it was, overwrite the original with the converted byte

        mov byte ptr [rcx], bl      ; Store the result
        inc rcx                     ; Move up to the next byte

        dec eax
        jnz ResidualsLoop

Finished:

        pop rbx       ; Restore the caller's RBX
        ret
ToUpperASM3 endp
end
```