Introduction to Computer System

Lab Report

Wang, Chen

16307110064

November 10th, 2018

Contents

[I. Program optimization related to this lab 3](#_Toc529579357)

[1. Optimizing Program Performance 3](#_Toc529579358)

[2. General loop optimization method 3](#_Toc529579359)

[3. Loop Unrolling 5](#_Toc529579360)

[1) Advantages 5](#_Toc529579361)

[2) Disadvantages 6](#_Toc529579362)

[4. Enhancing Parallelism 6](#_Toc529579363)

[II. Specifications of this lab 6](#_Toc529579364)

[1. Baseline program 6](#_Toc529579365)

[2. Specifications and details 7](#_Toc529579366)

[III. Implementation 7](#_Toc529579367)

[1. General process of implementation 7](#_Toc529579368)

[2. Details of the runtime 8](#_Toc529579369)

[IV. Experiment results 8](#_Toc529579370)

[1. Laptop with Intel(R) Core(TM) i7 6500U CPU @ 2.50GHz 8](#_Toc529579371)

[1) System Info 8](#_Toc529579372)

[2) Result 9](#_Toc529579373)

[3) Screen shot of the running environment 10](#_Toc529579374)

[2. Desktop with Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz 11](#_Toc529579375)

[1) System Info 11](#_Toc529579376)

[2) Result 13](#_Toc529579377)

[3) Screen shot of the running environment 13](#_Toc529579378)

[V. Result analysis 14](#_Toc529579379)

[1. Expected result 14](#_Toc529579380)

[1) Loop unrolling 14](#_Toc529579381)

[2) Enhancing parallelism 15](#_Toc529579382)

[2. Analysis of the lab result 16](#_Toc529579383)

[3. Proposed calculating method 17](#_Toc529579384)

[VI. References 18](#_Toc529579385)

[VII. Epilogue 19](#_Toc529579386)

# Program optimization related to this lab

## Optimizing Program Performance

In Chapter five of the book Computer System: A Programmer’s Perspective (abbr. CSAPP 3e below), the writer focused on the code-level program performance enhancement both Instruction Set Architecture-relative and Instruction Set Architecture-irrelative[[1]](#footnote-1). These improvements are essential to better utilize the hardware resource when handling large volume of dataset and requiring short response time.

Above all, the primary objective in writing a program must be to make it work correctly under all possible conditions. A program that runs fast but gives incorrect results serves no useful purpose. We ought to write clear and concise code, not only so that they can make sense of it, but also so that others can read and understand the code during code reviews and when modifications are required later.

However, on the other hand, there are many occasions when making a program run fast is also an important consideration. If a program must process video frames or network packets in real time, then a slow-running program will not provide the needed functionality. When a computational task is so demanding that it requires days or weeks to execute, then making it run just 20% faster can have significant impact.

As shown above, the chapter generally introduces various methods of getting program achieve higher speed. In this lab, we are going to observe two methods in particular, i.e. loop unrolling and enhancing parallelism, to see how they can affect the performance of the small program with the identical functionality.

## General loop optimization method

Loop optimization can be viewed as the application of a sequence of specific loop transformations (listed below or in Compiler transformations for high-performance computing) to the source code or intermediate representation, with each transformation having an associated test for legality. A transformation (or sequence of transformations) generally must preserve the temporal sequence of all dependencies if it is to preserve the result of the program (i.e., be a legal transformation). Evaluating the benefit of a transformation or sequence of transformations can be quite difficult within this approach, as the application of one beneficial transformation may require the prior use of one or more other transformations that, by themselves, would result in reduced performance.

Common loop transformations include:

* Fission or distribution – loop fission attempts to break a loop into multiple loops over the same index range, but each new loop takes only part of the original loop's body. This can improve locality of reference; both of the data being accessed in the loop and the code in the loop's body.
* Fusion or combining – this combines the bodies of two adjacent loops that would iterate the same number of times (whether or not that number is known at compile time), as long as they make no reference to each other's data.
* Interchange or permutation – these optimizations exchange inner loops with outer loops. When the loop variables index into an array, such a transformation can improve locality of reference, depending on the array's layout.
* Inversion – this technique changes a standard while loop into a do/while (a.k.a. repeat/until) loop wrapped in an if conditional, reducing the number of jumps by two for cases where the loop is executed. Doing so duplicates the condition check (increasing the size of the code) but is more efficient because jumps usually cause a pipeline stall. Additionally, if the initial condition is known at compile-time and is known to be side-effect-free, the initial if-guard can be skipped.
* Loop-invariant code motion – this can vastly improve efficiency by moving a computation from inside the loop to outside of it, computing a value just once before the loop begins, if the resultant quantity of the calculation will be the same for every loop iteration (i.e., a loop-invariant quantity). This is particularly important with address-calculation expressions generated by loops over arrays. For correct implementation, this technique must be used with inversion, because not all code is safe to be moved outside the loop.
* Parallelization – this is a special case of automatic parallelization focusing on loops, restructuring them to run efficiently on multiprocessor systems. It can be done automatically by compilers (automatic parallelization) or manually (inserting parallel directives like OpenMP).
* Reversal – a subtle optimization that reverses the order in which values are assigned to the index variable. This can help eliminate dependencies and thus enable other optimizations. Certain architectures utilize looping constructs at assembly level that count in a single direction only (e.g., decrement-jump-if-not-zero [DJNZ]).
* Scheduling – this divides a loop into multiple parts that may be run concurrently on multiple processors.
* Skewing – this technique is applied to a nested loop iterating over a multidimensional array, where each iteration of the inner loop depends on previous iterations, and rearranges its array accesses so that the only dependencies are between iterations of the outer loop.
* Software pipelining – a type of out-of-order execution of loop iterations to hide the latencies of processor function units.
* Splitting or peeling – this attempt to simplify a loop or eliminate dependencies by breaking it into multiple loops which have the same bodies but iterate over different portions of the index range. A special case is loop peeling, which can simplify a loop with a problematic first iteration by performing that iteration separately before entering the loop.
* Tiling or blocking – reorganizes a loop to iterate over blocks of data sized to fit in the cache.
* Vectorization – attempts to run as many of the loop iterations as possible at the same time on a SIMD system.
* Unrolling – duplicates the body of the loop multiple times, in order to decrease the number of times the loop condition is tested and the number of jumps, which may degrade performance by impairing the instruction pipeline. Completely unrolling a loop eliminates all overhead (except multiple instruction fetches and increased program load time), but requires that the number of iterations be known at compile time (except in the case of Just-in-time compilation). Care must also be taken to ensure that multiple re-calculation of indexed variables is not a greater overhead than advancing pointers within the original loop.
* Unswitching – moves a conditional from inside a loop to outside of it by duplicating the loop's body, and placing a version of it inside each of the if and else clauses of the conditional.
* Sectioning or strip-mining – introduced for vector processors, loop-sectioning is a loop-transformation technique for enabling SIMD (single instruction, multiple data)-encodings of loops and improving memory performance. This involves each vector operation being done for a size less-than or equal-to the maximum vector length on a given vector machine.

## Loop Unrolling

Loop unrolling, also known as loop unwinding, is a loop transformation technique that attempts to optimize a program’s execution speed at the expense of its binary size, which is an approach known as space–time tradeoff. The transformation can be undertaken manually by the programmer or by an optimizing compiler.

The goal of loop unwinding is to increase a program’s speed by reducing or eliminating instructions that control the loop, such as pointer arithmetic and “end of loop” tests on each iteration; reducing branch penalties; as well as hiding latencies including the delay in reading data from memory[[2]](#footnote-2). To eliminate this computational overhead, loops can be re-written as a repeated sequence of similar independent statements.

### Advantages

The overhead in “tight” loops often consists of instructions to increment a pointer or index to the next element in an array (pointer arithmetic), as well as “end of loop” tests[[3]](#footnote-3). If an optimizing compiler or assembler is able to pre-calculate offsets to each individually referenced array variable, these can be built into the machine code instructions directly, therefore requiring no additional arithmetic operations at run time.

* Significant gains can be realized if the reduction in executed instructions compensates for any performance reduction caused by any increase in the size of the program.
* Branch penalty is minimized.
* If the statements in the loop are independent of each other (i.e. where statements that occur earlier in the loop do not affect statements that follow them), the statements can potentially be executed in parallel.
* Can be implemented dynamically if the number of array elements is unknown at compile time (as in Duff’s device).

Optimizing compilers will sometimes perform the unrolling automatically, or upon request.

### Disadvantages

* Increased program code size, which can be undesirable, particularly for embedded applications. Can also cause an increase in instruction cache misses, which may adversely affect performance.
* Unless performed transparently by an optimizing compiler, the code may become less readable.
* If the code in the body of the loop involves function calls, it may not be possible to combine unrolling with in-lining, since the increase in code size might be excessive. Thus, there can be a trade-off between the two optimizations.
* Possible increased register usage in a single iteration to store temporary variables [dubious – discuss], which may reduce performance, though much will depend on possible optimizations.
* Apart from very small and simple codes, unrolled loops that contain branches are even slower than recursions.

## Enhancing Parallelism

With the loop unrolling technique in the previous section, we can predict that the performance of the program can nearly hit the latency bound of the physical processor. Loop unrolling can greatly eliminate the control overhead induced by the loop control logic. However, there are still some data dependencies between consecutive operations that may make the latter operation wait for the data to be processed. This kind of dependency can be eliminated by enhancing parallelism[[4]](#footnote-4). For a combining operation that is associative and commutative, such as integer addition or multiplication, we can improve performance by splitting the set of combining operations into two or more parts and combining the results at the end.

Enhancing parallelism can be achieved from two perspectives. The first of which is using multiple accumulators. As using just one accumulator can make the consequent data depend on the prior one thus leading to stalling in the pipeline processors, using more than one accumulator can break this kind of dependency because consecutive operations can be saved into different variables so that in the next loop, the variable needed for the first operation is computed in the operation two or more ahead of it. Reassociation Transformation is another way of breaking the chain of data dependency. The general idea of reassociation transformation is to re-organize the order of the operations so that the value needed in one operation is calculated far earlier rather than in the previous adjacent operation which will be sure to lead to data dependency.

# Specifications of this lab

## Baseline program

In this lab, we are assigned to modify the following baseline program code so as to enhance the performance.

*/\* Implementation with maximum use of data abstraction \*/***void** combinel(vec\_ptr v, data\_t \*dest) {  
 **long** i;  
 \*dest = IDENT;  
 **for** (i = 0; i < vec\_length(v); i++) {  
 data\_t val;  
 get\_vec\_element(v, i, &val);  
 \*dest = \*dest  
 OP val;  
 }  
}

## Specifications and details

As stated above, in this lab we are required to use loop unrolling and enhancing parallelism to enhance the performance of the program. Specifically, we use to symbol K and L to denote the degree of loop unrolling and number of accumulators.

In general, the loop can be unrolled to any degree L, and we can accumulate K results in parallel. To make things simpler and to utilize the unrolled loop and the additional accumulators, L should be a multiple of K. However, as Tang has requested, we want the full 144 results rather than when L is a multiple of K.

Breaking the limitation of L is a multiple of K leads to the problem of how the operation should be handled in the condition that L is not a multiple of K. After the assignment of the lab, the teaching assistance has remarked additionally Tang’s opinion of the way of handling this condition. In the code fragment supplied by Tang, he lets the additional accumulators to go further out of the degree of loop unrolling to do more operations in advance. What he omits is that the stride length of each iteration. From my perspective, the stride length of each iteration is the indication of loop unrolling degree. From a higher-level idea, the degree of loop unrolling and the number of accumulators are two independent factors. Based on Tang’s code fragment, we have two choices on the stride of each iteration. The first one of which is keep the stride of each iteration the same as the unrolling degree. In this case, the operations that go further beyond the current iterations are overwritten in the consequent iteration by some other accumulators. In other words, such accumulators are wasted. The second of which is set the stride of each iteration larger so as to cover all the operation-accessing array indexes of all the operations. This way, in the contrary, wasted no accumulator. However, as I have proposed above, makes the value of loop unrolling degree vail. As we can expect, in such way, all the K × L loop unrolling where L < K is equivalent to K × K loop unrolling in the real fact. In this way, the process of measuring such performance values seem to be meaningless.

In general, above are only the personal opinion on the decision when meeting the values breaking the restriction that L is a multiple of K. Nevertheless, I cherish each opinion proposed by others while holding my own opinion. Therefore, I have chosen the first method of dealing with such occasion in my version of lab because it holds the main principle of K and L are separate factors and the intrinsic quality of different case should not overlap with each other.

# Implementation

## General process of implementation

The general specification of this lab and the controversy detail have both been stated above. And my implementation just goes as required. Above all, the different combination of L and K is generated dynamically by the C language method *methodGen*. This method works in accordance with the specifications I have stated in the previous sections. The different methods are generated into C files that contain only a method called *combine*. After generating the C files, these files are then compiled into dynamically link libraries that will be linked into the testing method dynamically. The last step is linking the dynamic link library into the main testing file and switch between cases. As required, a GUI tool is provided for the convenience of performing the three procedures above. The reason of separating the whole testing into three steps is for the convenience of observing in case of any intermediate errors.

Because my application ran on various computers to get more general result, the computer system info part in the generated-out file has been omitted. Instead, I attach with each out file a system info file so as to indicate the computer performance.

## Details of the runtime

* It requires an *executing\_files* folder in the folder that contains the executable files;
* The intermediate files like C files and SO files are saved in this folder;
* The final CPE result is saved in the same folder as that containing the executable file;
* If you want to reproduce the generating So files, the header files *combine.h* and *vec.h* should be provided in the *executing\_files* folder.
* If you want to run the GUI application to manipulate the procedures, keep the executable files the same path as you run the GUI path.

# Experiment results

## Laptop with Intel(R) Core(TM) i7 6500U CPU @ 2.50GHz

### System Info

System information report, generated by Sysinfo: 11/9/2018 10:21:55 PM

http://sourceforge.net/projects/gsysinfo

SYSTEM INFORMATION

Running Ubuntu Linux, the Ubuntu 18.04 (bionic) release.

GNOME: 3.28.2 (Ubuntu)

Kernel version: 4.15.0-36-generic (#39-Ubuntu SMP Mon Sep 24 16:19:09 UTC 2018)

GCC: 7 (x86\_64-linux-gnu)

Xorg: unknown

Hostname: straybird-Lenovo-XiaoXin-Air-13-Pro

Uptime: 0 days 0 h 18 min

CPU INFORMATION

GenuineIntel, Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz

Number of CPUs: 4

CPU clock currently at 1000.010 MHz with 4096 KB cache

Numbering: family(6) model(78) stepping(3)

Bogomips: 5184.00

Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant\_tsc art arch\_perfmon pebs bts rep\_good nopl xtopology nonstop\_tsc cpuid aperfmperf tsc\_known\_freq pni pclmulqdq dtes64 monitor ds\_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4\_1 sse4\_2 x2apic movbe popcnt tsc\_deadline\_timer aes xsave avx f16c rdrand lahf\_lm abm 3dnowprefetch cpuid\_fault epb invpcid\_single pti ssbd ibrs ibpb stibp tpr\_shadow vnmi flexpriority ept vpid fsgsbase tsc\_adjust bmi1 avx2 smep bmi2 erms invpcid mpx rdseed adx smap clflushopt intel\_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp\_notify hwp\_act\_window hwp\_epp flush\_l1d

MEMORY INFORMATION

Total memory: 7859 MB

Total swap: 947 MB

STORAGE INFORMATION

HARDWARE INFORMATION

MOTHERBOARD

Host bridge

Intel Corporation Skylake Host Bridge/DRAM Registers (rev 08)

Subsystem: Lenovo Xeon E3-1200 v5/E3-1500 v5/6th Gen Core Processor Host Bridge/DRAM Registers

PCI bridge(s)

Intel Corporation Sunrise Point-LP PCI Express Root Port (rev f1) (prog-if 00 [Normal decode])

Intel Corporation Sunrise Point-LP PCI Express Root Port #5 (rev f1) (prog-if 00 [Normal decode])

Intel Corporation Sunrise Point-LP PCI Express Root Port #9 (rev f1) (prog-if 00 [Normal decode])

Intel Corporation Sunrise Point-LP PCI Express Root Port (rev f1) (prog-if 00 [Normal decode])

ISA bridge

Intel Corporation Sunrise Point-LP LPC Controller (rev 21)

Subsystem: Lenovo Sunrise Point-LP LPC Controller

GRAPHIC CARD

VGA controller

Intel Corporation HD Graphics 520 (rev 07) (prog-if 00 [VGA controller])

Subsystem: Lenovo Skylake GT2 [HD Graphics 520]

SOUND CARD

Multimedia controller

Intel Corporation Sunrise Point-LP HD Audio (rev 21)

Subsystem: Lenovo Sunrise Point-LP HD Audio

NETWORK

Network controller

Intel Corporation Dual Band Wireless-AC 3165 Plus Bluetooth (rev 99)

Subsystem: Intel Corporation Dual Band Wireless-AC 3165 Plus Bluetooth

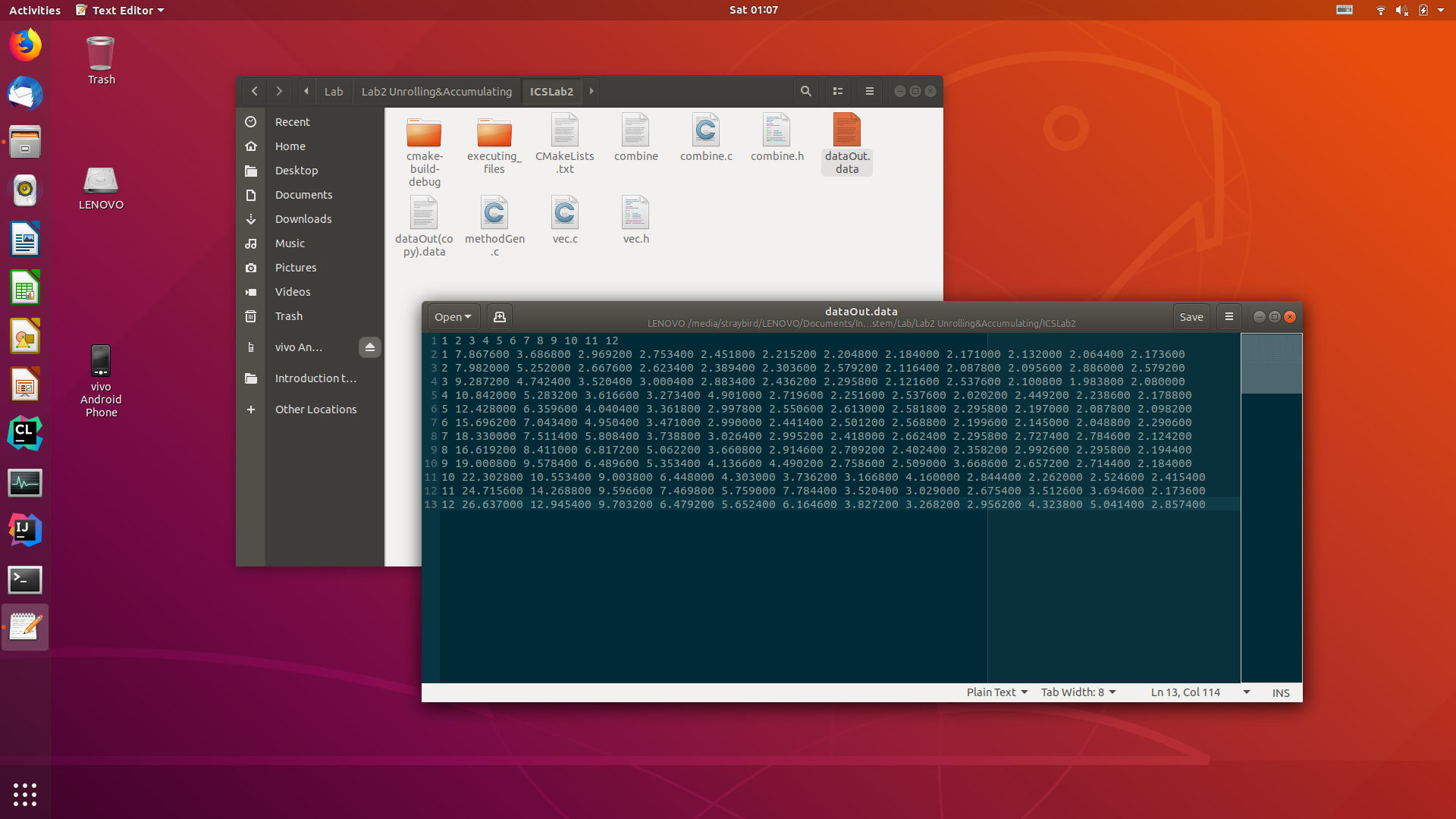
* The GUI launcher

### Result

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Unrolling Factor L | | | | | | | | | | | |
|  |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| Accumulators | 1 | 7.8676 | 3.6868 | 2.9692 | 2.7534 | 2.4518 | 2.2152 | 2.2048 | 2.184 | 2.171 | 2.132 | 2.0644 | 2.1736 |
| 2 | 7.982 | 5.252 | 2.6676 | 2.6234 | 2.3894 | 2.3036 | 2.5792 | 2.1164 | 2.0878 | 2.0956 | 2.886 | 2.5792 |
| 3 | 9.2872 | 4.7424 | 3.5204 | 3.0004 | 2.8834 | 2.4362 | 2.2958 | 2.1216 | 2.5376 | 2.1008 | 1.9838 | 2.08 |
| 4 | 10.842 | 5.2832 | 3.6166 | 3.2734 | 4.901 | 2.7196 | 2.2516 | 2.5376 | 2.0202 | 2.4492 | 2.2386 | 2.1788 |
| 5 | 12.428 | 6.3596 | 4.0404 | 3.3618 | 2.9978 | 2.5506 | 2.613 | 2.5818 | 2.2958 | 2.197 | 2.0878 | 2.0982 |
| 6 | 15.696 | 7.0434 | 4.9504 | 3.471 | 2.99 | 2.4414 | 2.5012 | 2.5688 | 2.1996 | 2.145 | 2.0488 | 2.2906 |
| 7 | 18.33 | 7.5114 | 5.8084 | 3.7388 | 3.0264 | 2.9952 | 2.418 | 2.6624 | 2.2958 | 2.7274 | 2.7846 | 2.1242 |
| 8 | 16.619 | 8.411 | 6.8172 | 5.0622 | 3.6608 | 2.9146 | 2.7092 | 2.4024 | 2.3582 | 2.9926 | 2.2958 | 2.1944 |
| 9 | 19.000 | 9.5784 | 6.4896 | 5.3534 | 4.1366 | 4.4902 | 2.7586 | 2.509 | 3.6686 | 2.6572 | 2.7144 | 2.184 |
| 10 | 22.302 | 10.553 | 9.0038 | 6.448 | 4.303 | 3.7362 | 3.1668 | 4.16 | 2.8444 | 2.262 | 2.5246 | 2.4154 |
| 11 | 24.715 | 14.268 | 9.5966 | 7.4698 | 5.759 | 7.7844 | 3.5204 | 3.029 | 2.6754 | 3.5126 | 3.6946 | 2.1736 |
| 12 | 26.637 | 12.945 | 9.7032 | 6.4792 | 5.6524 | 6.1646 | 3.8272 | 3.2682 | 2.9562 | 4.3238 | 5.0414 | 2.8574 |

### Screen shot of the running environment

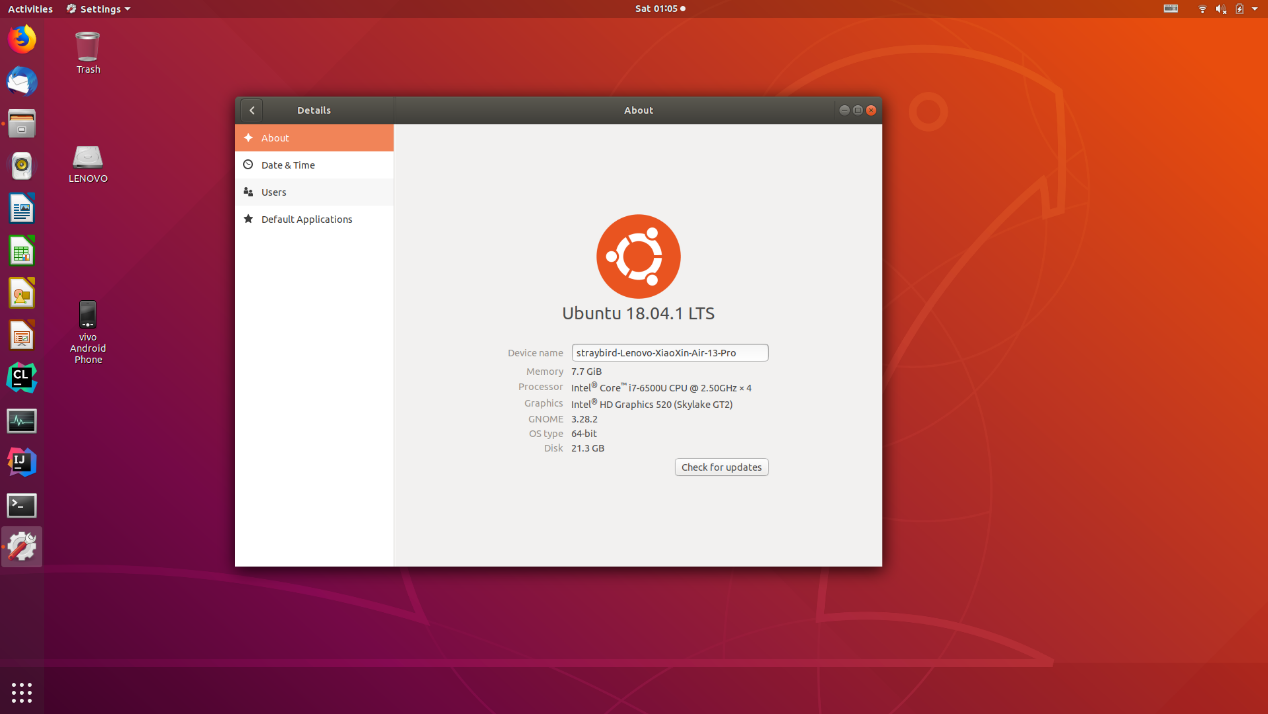
* The measured data



* The GUI launcher



* The system info



## Desktop with Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz

### System Info

System information report, generated by Sysinfo: 11/9/2018 9:43:54 PM

http://sourceforge.net/projects/gsysinfo

SYSTEM INFORMATION

Running Ubuntu Linux, the Ubuntu 18.04 (bionic) release.

GNOME: 3.28.2 (Ubuntu)

Kernel version: 4.15.0-36-generic (#39-Ubuntu SMP Mon Sep 24 16:19:09 UTC 2018)

GCC: 7 (x86\_64-linux-gnu)

Xorg: unknown

Hostname: wangchen-OptiPlex-7050-China-HDD-Protection

Uptime: 0 days 0 h 12 min

CPU INFORMATION

GenuineIntel, Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz

Number of CPUs: 8

CPU clock currently at 799.996 MHz with 8192 KB cache

Numbering: family(6) model(158) stepping(9)

Bogomips: 7200.00

Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant\_tsc art arch\_perfmon pebs bts rep\_good nopl xtopology nonstop\_tsc cpuid aperfmperf tsc\_known\_freq pni pclmulqdq dtes64 monitor ds\_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4\_1 sse4\_2 x2apic movbe popcnt tsc\_deadline\_timer aes xsave avx f16c rdrand lahf\_lm abm 3dnowprefetch cpuid\_fault epb invpcid\_single pti ssbd ibrs ibpb stibp tpr\_shadow vnmi flexpriority ept vpid fsgsbase tsc\_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm mpx rdseed adx smap clflushopt intel\_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp\_notify hwp\_act\_window hwp\_epp flush\_l1d

MEMORY INFORMATION

Total memory: 7854 MB

Total swap: 2047 MB

STORAGE INFORMATION

SCSI device - scsi0

Vendor: ATA

Model: ST1000DM003-1SB1

HARDWARE INFORMATION

MOTHERBOARD

Host bridge

Intel Corporation Intel Kaby Lake Host Bridge (rev 05)

Subsystem: Dell Intel Kaby Lake Host Bridge

PCI bridge(s)

Intel Corporation 200 Series PCH PCI Express Root Port #4 (rev f0) (prog-if 00 [Normal decode])

Texas Instruments XIO2001 PCI Express-to-PCI Bridge (prog-if 00 [Normal decode])

ISA bridge

Intel Corporation 200 Series PCH LPC Controller (Q270)

Subsystem: Dell 200 Series PCH LPC Controller (Q270)

GRAPHIC CARD

VGA controller

Intel Corporation HD Graphics 630 (rev 04) (prog-if 00 [VGA controller])

Subsystem: Dell HD Graphics 630

SOUND CARD

Multimedia controller

Intel Corporation 200 Series PCH HD Audio

Subsystem: Dell 200 Series PCH HD Audio

NETWORK

Ethernet controller

Intel Corporation Ethernet Connection (5) I219-LM

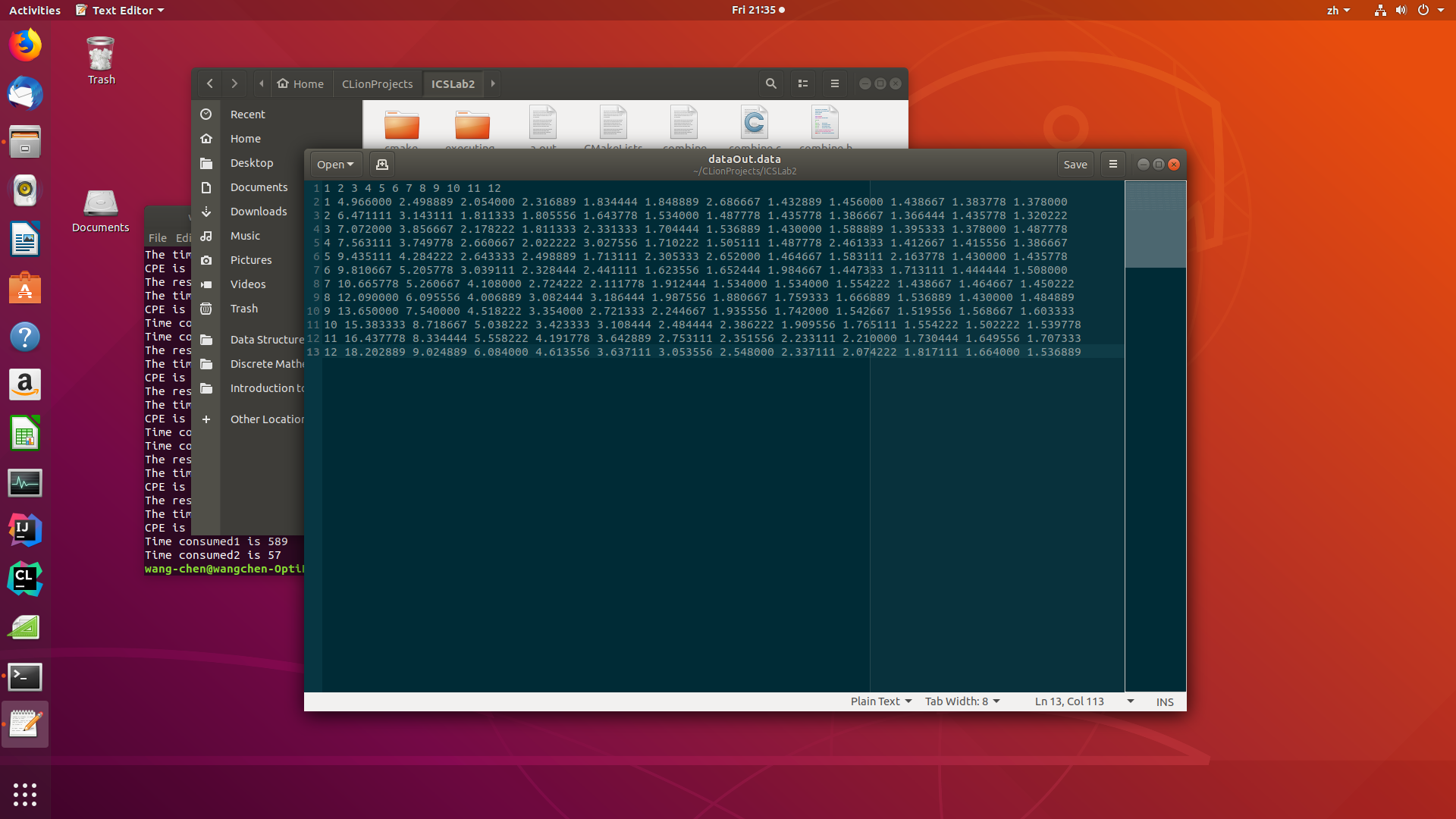
Subsystem: Dell Ethernet Connection (5) I219-LM

### Result

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Unrolling Factor L | | | | | | | | | | | |
|  |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| Accumulators | 1 | 4.966 | 2.499 | 2.054 | 2.317 | 1.834 | 1.849 | 2.687 | 1.433 | 1.456 | 1.439 | 1.384 | 1.378 |
| 2 | 6.471 | 3.143 | 1.811 | 1.806 | 1.644 | 1.534 | 1.488 | 1.436 | 1.387 | 1.366 | 1.436 | 1.320 |
| 3 | 7.072 | 3.857 | 2.178 | 1.811 | 2.331 | 1.704 | 1.537 | 1.430 | 1.589 | 1.395 | 1.378 | 1.488 |
| 4 | 7.563 | 3.750 | 2.661 | 2.022 | 3.028 | 1.710 | 1.505 | 1.488 | 2.461 | 1.413 | 1.416 | 1.387 |
| 5 | 9.435 | 4.284 | 2.643 | 2.499 | 1.713 | 2.305 | 2.652 | 1.465 | 1.583 | 2.164 | 1.430 | 1.436 |
| 6 | 9.811 | 5.206 | 3.039 | 2.328 | 2.441 | 1.624 | 1.652 | 1.985 | 1.447 | 1.713 | 1.444 | 1.508 |
| 7 | 10.666 | 5.261 | 4.108 | 2.724 | 2.112 | 1.912 | 1.534 | 1.534 | 1.554 | 1.439 | 1.465 | 1.450 |
| 8 | 12.090 | 6.096 | 4.007 | 3.082 | 3.186 | 1.988 | 1.881 | 1.759 | 1.667 | 1.537 | 1.430 | 1.485 |
| 9 | 13.650 | 7.540 | 4.518 | 3.354 | 2.721 | 2.245 | 1.936 | 1.742 | 1.543 | 1.520 | 1.569 | 1.603 |
| 10 | 15.383 | 8.719 | 5.038 | 3.423 | 3.108 | 2.484 | 2.386 | 1.910 | 1.765 | 1.554 | 1.502 | 1.540 |
| 11 | 16.438 | 8.334 | 5.558 | 4.192 | 3.643 | 2.753 | 2.352 | 2.233 | 2.210 | 1.730 | 1.650 | 1.707 |
| 12 | 18.203 | 9.025 | 6.084 | 4.614 | 3.637 | 3.054 | 2.548 | 2.337 | 2.074 | 1.817 | 1.664 | 1.537 |

### Screen shot of the running environment

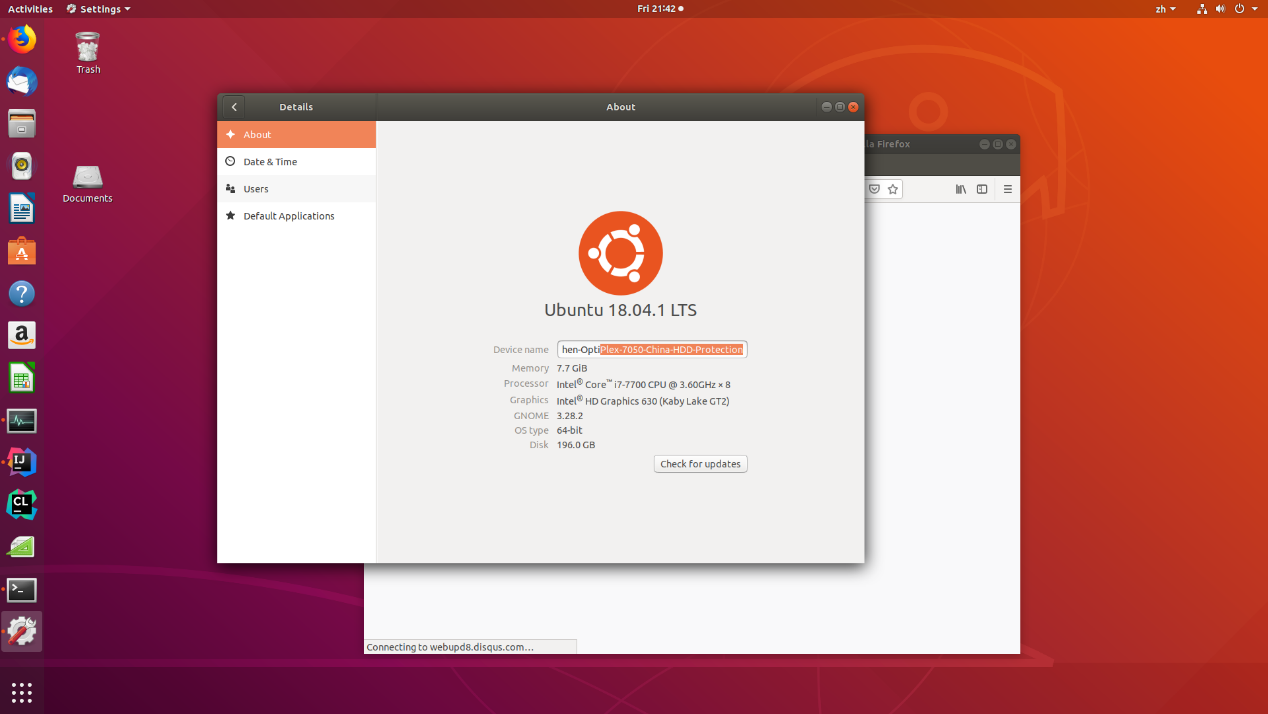
* The measured data



* The GUI launcher



* The system info



# Result analysis

## Expected result

### Loop unrolling

Loop unrolling (also called loop unwinding) is a program transformation that reduces the number of iterations for a loop by increasing the number of elements computed on each iteration. It is one of the most common methods that the compilers implement to optimize the loops. It utilizes the CPU pipeline in the best possible way. Other methods for loop optimization include: peephole optimizations, local optimizations, global optimizations, machine code optimization, etc. From the textbook, we saw an example of this with the function psum2 (Figure 5.1), where each iteration computes two elements of the prefix sum, thereby halving the total number of iterations required. Loop unrolling can improve performance in two ways. First, it reduces the number of operations that do not contribute directly to the program result, such as loop indexing and conditional branching. Second, it exposes ways in which we can further transform the code to reduce the number of operations in the critical paths of the overall computation.

The goal of the loop unrolling is to increase the program performance by reducing branching penalties and pointer arithmetic and increasing the efficiency of pipelining. The process of loop unwinding is often achieved by dispersing the innermost loops to several independent instructions. Therefore, the level of usage of this method was limited because it had to be specified by the user for maximum performance. In order to eliminate this limitation, loop quantization was introduced as a general technique for automatically unwinding multiply nested loops.

Loop unrolling can contribute to improve the instruction level parallelism (ILP) and execution performance of programs, by enabling more optimization that are affected by code expansion. Although, this code expansion can lead to instruction-cache performance degradation, if not carefully applied. If loop unrolling is applied before the register allocation phase, register pressure can be increased, leading to the insertion of more spill and reload operations in the code. However, a standard compiler cannot use loop unrolling directly if worst-case execution time (WCET) reduction is desirable, due to the instability of the execution path that generates the worst possible execution time and negative cache effects[[5]](#footnote-5). Some techniques were proposed in the literature to achieve WCET reduction using loop unrolling. In these works, loops are carefully unrolled to promote WCET reduction and limit code increase. But, only loops with fixed execution counts are considered.

Therefore, we can predict that the larger the loop unrolling factor is, the lower the overhead costs, thus leading to lower CPE. Here we always assume that the data volume is greatly larger than the loop unrolling degree, in other words, there is never the occasion that the loop unrolling factor is larger than the data volume so that the data cannot go into the loop even for the first time. Actually, in the experiment, we always use vectors with a length of 1e6 or 1e7 or so, so that the volume of the data is greatly larger than the loop unrolling factor.

### Enhancing parallelism

From the previous subsection, we can conclude that loop unrolling can greatly eliminate the control overhead induced by the loop control logic. However, there are still some data dependencies between consecutive operations that may make the latter operation wait for the data to be processed[[6]](#footnote-6). This kind of dependency can be eliminated by enhancing parallelism. For a combining operation that is associative and commutative, such as integer addition or multiplication, we can improve performance by splitting the set of combining operations into two or more parts and combining the results at the end.

As has been mentioned in the previous sections, the process of enhancing parallelism can be achieved from two perspectives. The first of which is using multiple accumulators. As using just one accumulator can make the consequent data depend on the prior one thus leading to stalling in the pipeline processors, using more than one accumulator can break this kind of dependency because consecutive operations can be saved into different variables so that in the next loop, the variable needed for the first operation is computed in the operation two or more ahead of it. Reassociation Transformation is another way of breaking the chain of data dependency. The general idea of reassociation transformation is to re-organize the order of the operations so that the value needed in one operation is calculated far earlier rather than in the previous adjacent operation which will be sure to lead to data dependency.

Therefore, we have acknowledged that enhancing parallelism contributes to eliminating data dependency. And from the physical structure of the CPU, we know that the extend of parallelism is not limited. In conclusion, when in small numbers, enhancing parallelism can enhance the performance of the program so as to reduce CPE, while when reaching some extend, the performance of the application comes to be reaching the limit set by the physical CPU.

## Analysis of the lab result

In order to have more intuitive sense of the data result, the data set is formatted with color scales. The red ones indicate smaller value while the green ones indicate larger value. Despite the accidental irregular data entries, both of the data result indicates different trend against the previous conclusion.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 1 | 7.868 | 3.687 | 2.969 | 2.753 | 2.452 | 2.215 | 2.205 | 2.184 | 2.171 | 2.132 | 2.064 | 2.174 |
| 2 | 7.982 | 5.252 | 2.668 | 2.623 | 2.389 | 2.304 | 2.579 | 2.116 | 2.088 | 2.096 | 2.886 | 2.579 |
| 3 | 9.287 | 4.742 | 3.52 | 3 | 2.883 | 2.436 | 2.296 | 2.122 | 2.538 | 2.101 | 1.984 | 2.08 |
| 4 | 10.84 | 5.283 | 3.617 | 3.273 | 4.901 | 2.72 | 2.252 | 2.538 | 2.02 | 2.449 | 2.239 | 2.179 |
| 5 | 12.43 | 6.36 | 4.04 | 3.362 | 2.998 | 2.551 | 2.613 | 2.582 | 2.296 | 2.197 | 2.088 | 2.098 |
| 6 | 15.7 | 7.043 | 4.95 | 3.471 | 2.99 | 2.441 | 2.501 | 2.569 | 2.2 | 2.145 | 2.049 | 2.291 |
| 7 | 18.33 | 7.511 | 5.808 | 3.739 | 3.026 | 2.995 | 2.418 | 2.662 | 2.296 | 2.727 | 2.785 | 2.124 |
| 8 | 16.62 | 8.411 | 6.817 | 5.062 | 3.661 | 2.915 | 2.709 | 2.402 | 2.358 | 2.993 | 2.296 | 2.194 |
| 9 | 19 | 9.578 | 6.49 | 5.353 | 4.137 | 4.49 | 2.759 | 2.509 | 3.669 | 2.657 | 2.714 | 2.184 |
| 10 | 22.3 | 10.55 | 9.004 | 6.448 | 4.303 | 3.736 | 3.167 | 4.16 | 2.844 | 2.262 | 2.525 | 2.415 |
| 11 | 24.72 | 14.27 | 9.597 | 7.47 | 5.759 | 7.784 | 3.52 | 3.029 | 2.675 | 3.513 | 3.695 | 2.174 |
| 12 | 26.64 | 12.95 | 9.703 | 6.479 | 5.652 | 6.165 | 3.827 | 3.268 | 2.956 | 4.324 | 5.041 | 2.857 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 1 | 4.966 | 2.499 | 2.054 | 2.317 | 1.834 | 1.849 | 2.687 | 1.433 | 1.456 | 1.439 | 1.384 | 1.378 |
| 2 | 6.471 | 3.143 | 1.811 | 1.806 | 1.644 | 1.534 | 1.488 | 1.436 | 1.387 | 1.366 | 1.436 | 1.32 |
| 3 | 7.072 | 3.857 | 2.178 | 1.811 | 2.331 | 1.704 | 1.537 | 1.43 | 1.589 | 1.395 | 1.378 | 1.488 |
| 4 | 7.563 | 3.75 | 2.661 | 2.022 | 3.028 | 1.71 | 1.505 | 1.488 | 2.461 | 1.413 | 1.416 | 1.387 |
| 5 | 9.435 | 4.284 | 2.643 | 2.499 | 1.713 | 2.305 | 2.652 | 1.465 | 1.583 | 2.164 | 1.43 | 1.436 |
| 6 | 9.811 | 5.206 | 3.039 | 2.328 | 2.441 | 1.624 | 1.652 | 1.985 | 1.447 | 1.713 | 1.444 | 1.508 |
| 7 | 10.67 | 5.261 | 4.108 | 2.724 | 2.112 | 1.912 | 1.534 | 1.534 | 1.554 | 1.439 | 1.465 | 1.45 |
| 8 | 12.09 | 6.096 | 4.007 | 3.082 | 3.186 | 1.988 | 1.881 | 1.759 | 1.667 | 1.537 | 1.43 | 1.485 |
| 9 | 13.65 | 7.54 | 4.518 | 3.354 | 2.721 | 2.245 | 1.936 | 1.742 | 1.543 | 1.52 | 1.569 | 1.603 |
| 10 | 15.38 | 8.719 | 5.038 | 3.423 | 3.108 | 2.484 | 2.386 | 1.91 | 1.765 | 1.554 | 1.502 | 1.54 |
| 11 | 16.44 | 8.334 | 5.558 | 4.192 | 3.643 | 2.753 | 2.352 | 2.233 | 2.21 | 1.73 | 1.65 | 1.707 |
| 12 | 18.2 | 9.025 | 6.084 | 4.614 | 3.637 | 3.054 | 2.548 | 2.337 | 2.074 | 1.817 | 1.664 | 1.537 |

After thoroughly thinking about the process of calculation, I have realized the potential mistaken point. In my calculation, I have always used a static number to calculate the number of cycles between some time interval. Take my laptop, for example, I have used 2.5GHz to represent the number of instructions that the CPU can start to execute. However, things can get much more complex. Above all, the frequency of CPU varies from time to time, at a minimal interval. Besides, there are much more processes in one core and they are switched at an also minimal interval. For my laptop, there are 2 physical cores and 4 logic cores. Because my combine method is not a multi-processes or multi-thread process, it will be executed on one core at any time. Nonetheless, there are thousands of processes sharing the four logic cores, indicating that my process, the combine method specifically, is not always executed even a small portion of the time between its start off and its finish.

What’s making things worse is that the frequency varies all the time. That is, we can neither use the peek frequency nor get a test frequency data at some point of the program execution process.

## Proposed calculating method

Let denote the frequency of the core that is executing the program we want to measure at the time point of absolute time . Let denote the number of the process that we want to measure is switched into CPU. Let denote the absolute time of the th switching the process we want to measure into the CPU and . Let denote the cycles used in the process of the combine method of vector length .

Considering that the frequency of CPU will not change within a time unlimitedly small but have a concrete *least change frequency period*. Let denote the *least change frequency period* on any particular CPU, we can simplify the integral above as the equation below.

With the equation above, I believe that the cycle number should accurately represent the cycles for executing the process we want. And the CPE can be calculated in the following equation.

However, much of the work like switching between different processes are finished by the operating system and quiring all the data seems much too complex for this single lab. Therefore, the improving part is currently left for the future.

# References

Allen, J. R., Kennedy, K., Porterfield, C., & Warren, J. (1983). Conversion of control dependence to data dependence. *POPL '83 Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (pp. 177-189). Austin, Texas: ACM.

Bryant, R. E., & O'hallaron, D. R. (2001). *Introducing computer systems from a programmer's perspective.* Boston: Pearson Education.

Carminati, A., Starke, R. A., & Oliveira, R. S. (2017). Combining loop unrolling strategies and code predication to reduce the worst-case execution time of real-time software. *Applied Computing and Informatics, 13*(2), 184-193. doi:10.1016/j.aci.2017.03.002

Gustafsson, Jan, Betts, Adam, Ermedahl, Andreas, & Lisper, Björn. (2010). The Mälardalen WCET Benchmarks: Past, Present And Future. *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010).* *15*, pp. 136-146. Dagstuhl, Germany: Schloss Dagstuhl--Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2010.136

Lin, C. Y., Jiang, Z., Fu, C., So, H. K., & Yang, H. (2017). FPGA High-level Synthesis versus Overlay: Comparisons on Computation Kernels. *ACM Sigarch Computer Architecture News, 44*(4), 92-97. Retrieved November 10, 2018

Reinhard v. Hanxleden, & Ken Kennedy. (1992). Relaxing SIMD control flow constraints using loop transformations. *PLDI '92 Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation* (pp. 188-199). San Francisco, California, USA: ACM.

Velkoski, G., Gusev, M., & Ristov, S. (2014). The performance impact analysis of loop unrolling. *international convention on information and communication technology electronics and microelectronics.*, 307-412.

# Epilogue

Despite the textbook, the references listed above, in general, didn’t help much in the process of finishing this lab. The requirement is never unachievable, thus in the process of finishing the fundamental measuring part of this lab and the designing the GUI, I referred to none of the materials above.

Writing the code as I have done to finish the measurement is not hard. However, the great problem is what I have said in the result analysis part. After the analysis, I found that I have problem getting to the real CPE number. And measuring the exact cycle number doesn’t seem to be that easy. In the pursuit of getting a better measuring method, I fall in the end in vain. Even though there are masses of the articles talking about enhancing the program performance, none of them talked about the process of getting an accurate CPE. And it is in this process that I came across the articles listed above and learned something about their opinions about the performance optimization and techniques.

In particular, the article *Conversion of control dependence to data dependence* talked about the solution to the phenomenon that control dependences do not fit conveniently into dependence-based program translators. As to the article *combining loop unrolling strategies and code predication to reduce the worst-case execution time of real-time software*, the goal of this paper is to propose a different way to perform loop unrolling on data-dependent loops using code predication targeting WCET reduction, because existing techniques only consider loops with fixed execution counts. The conference report *The Mälardalen WCET Benchmarks: Past, Present And Future* is actually a detailed guideline of the WCET. The articles *FPGA High-level Synthesis versus Overlay: Comparisons on Computation Kernels* and *Relaxing SIMD control flow constraints using loop transformations* talked about optimization about FPGA and SIMD separately. The article *The performance impact analysis of loop unrolling* generalized the impact of loop unrolling.

1. Bryant, R. E., & O'hallaron, D. R. (2001). Introducing computer systems from a programmer's perspective. Boston: Pearson Education. [↑](#footnote-ref-1)
2. Velkoski, G., Gusev, M., & Ristov, S. (2014). The performance impact analysis of loop unrolling. international convention on information and communication technology electronics and microelectronics., 307-412. [↑](#footnote-ref-2)
3. Carminati, A., Starke, R. A., & Oliveira, R. S. (2017). Combining loop unrolling strategies and code predication to reduce the worst-case execution time of real-time software. Applied Computing and Informatics, 13(2), 184-193. doi:10.1016/j.aci.2017.03.002 [↑](#footnote-ref-3)
4. Allen, J. R., Kennedy, K., Porterfield, C., & Warren, J. (1983). Conversion of control dependence to data dependence. POPL '83 Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (pp. 177-189). Austin, Texas: ACM. [↑](#footnote-ref-4)
5. Gustafsson, Jan, Betts, Adam, Ermedahl, Andreas, & Lisper, Björn. (2010). The Mälardalen WCET Benchmarks: Past, Present And Future. 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010). 15, pp. 136-146. Dagstuhl, Germany: Schloss Dagstuhl--Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2010.136 [↑](#footnote-ref-5)
6. Reinhard v. Hanxleden, & Ken Kennedy. (1992). Relaxing SIMD control flow constraints using loop transformations. PLDI '92 Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation (pp. 188-199). San Francisco, California, USA: ACM. [↑](#footnote-ref-6)