Simple SIMD Programming Example

CompMeth Spring 2018

Davide Pola

April 25, 2018

Contents

1 Introduction

The goal of this report is to illustrate the work done on computing fixed point multiplications in C with and without the SIMD ISA extension from Intel. The work have been entirely developed for the CompMeth course in EURECOM.

1.1 Content

First of all, a small introducing section will generally speak about SIMD and show a very simple example of a SIMD instruction. Later, Sections 3 and 4 will separately cover real and complex fixed point multiplication, with a focus on the developed code. Section 5 covers in detail all the setup required to obtain valuable results and finally Section 6 consist of visualization and comments about the simulations.

2 SIMD

2.1 SIMD support

Most recent high-end CPUs support SIMD, which stands for **S**ingle **I**nstruction **M**ultiple **D**ata. These kind of instructions are specifically targeted to exploit data level parallelism in order to improve performance for specific tasks, such as image and signal processing or scientific calculus.

Every recent CPU architecture has his own SIMD instruction set:

Intel - MMX, SSE, AVX, AVX2

AMD - 3DNow!, SSE, AVX, AVX2

ARM - NEON

In order to discover which SIMD extensions are enabled on a given CPU, either refer to the Product Specification, or issue the command `grep flags /proc/cpuinfo` (on Linux). As the target CPU used in this project is an Intel i7-4510u, all possible informations can be found in [4]

2.2 How to code with SIMD

There are 4 different ways to code using SIMD:

- SIMD enabled libraries
- Compiler auto-vectorization
- SIMD Intrinsics
- SIMD Assembly Code

In this work, most of the code is developed with Intel SSE/AVX Intrinsics as the target machine is a Haswell equipped laptop. Unfortunately, there is no support for AVX-512 extension as it is recent and at the moment of writing (Q2 2018) only targeted to HPC and high-end processors.

2.3 Example of an SIMD Instruction

A regular CPU operates on scalars, one at a time. A vector processor, on the other hand, lines up a whole row of these scalars, all of the same type, and operates on them as a unit. Here is an example that will help to understand a little the SIMD instructions. Figure 1 shows a simple scheme of the PADDD instruction (SSE2).

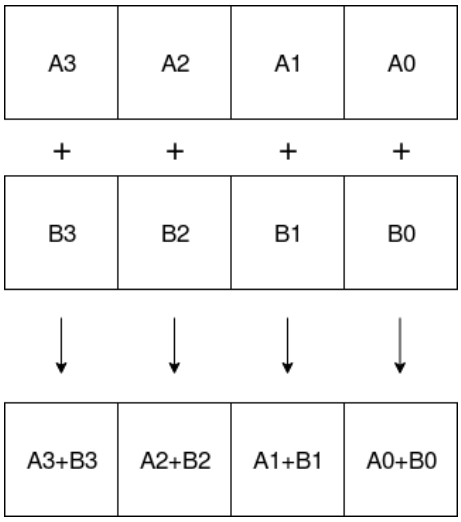

Figure 1: SSE PADDD scheme.

The PADDD mnemonic means packed addition for double-word. This assembly instruction receives

two 128-bit input parameters that contain four 32-bit integers each. The instruction returns a 128-bit output that contains the sum for each of the four 32-bit integers, packed in the 128-bit output. You can calculate the sum for four 32-bit signed integers with a single call to the PADDD instruction. If you have to calculate the sum for 100 couples of 32-bit integers, you can do it with 25 calls to this instruction instead of using a single instruction for each couple. It is possible to achieve very important speedups. However, because it is sometimes necessary to pack the data before calling the SIMD instruction and then unpack the output, it is also important to measure this overhead that adds some code.

3 Real Fixed Point Multiplication

3.1 Scalar

The scalar operation required is quite simple. Two 16-bit numbers (internally Q1.15) need to be multiplied, this is done simply by multiplying them as standard unsigned 32 bits, then shift right the outcome of 15 positions (16-1 because the sign would be on the two most significant bits). Only the lowest 16 bits will remain after the last typecasting to 16-bit. This operation is then repeated in a `for` loop for each element that needs to be computed.

Listing 1: Scalar Real Fixed Point Multiplication

```
for(i=0;i<N;i++){
  z[i] = (uint16_t) ((((uint32_t) x[i]) * y[i]) >> 15);
  }
```

3.2 SIMD

In the SIMD case, the operation is split in chunks of 8 (SSE) or 16 (AVX) integers operated in parallel. The surrounding loop is then iterating on ⅛ or 1/16 of the previous N.

Listing 2: AVX2 Real Fixed Point Multiplication

```
__m256i *x256 = (__m256i *)x;
__m256i *y256 = (__m256i *)y;
__m256i *z256 = (__m256i *)z;

uint32_t i;

N>>=4;
for(i=0; i<N; i++)
{
        z256[i] = _mm256_mulhrs_epi16(x256[i], y256[i]);
}
```

Fortunately, both SSE and AVX extension provide an intrinsic which is perfectly shaped for this purpose:

AVX2 `__m256i _mm256_mulhrs_epi16 (__m256i a, __m256i b)`

SSE3 `__m128i _mm_mulhrs_epi16 (__m128i a, __m128i b)`

As can be seen in Listing 2, the arrays coming need first to be typecasted to the intrinsic variable, then the process could be done. Please note that no packing and unpacking is required as the numbers are already distributed in the correct way (alignment problems will be discussed later).

4 Complex Fixed Point Multiplication

The first problem related to complex multiplication is due to the representation of numbers. They can be either placed in a structure, thus in an interleaved real/imaginary fashion, or have two separate arrays for reals and imaginaries, both the two cases have been treated in this work.

4.1 Interleaved fashion

The basic structure of a complex number is defined in C as follows:

Listing 3: Complex Number Structure

```
typedef struct complex16 {
int16_t r;
int16_t i;
} complex16;
```

4.1.1 Scalar

Again, the scalar code is pretty straightforward, the code can be seen in Listing 4.

Listing 4: Scalar Complex Fixed Point Multiplication - Interleaved Fashion

```
for(i=0;i<N;i++){
  z[i].r = (uint16_t) ((((int32_t) x[i].r) * ((int32_t)y[i].r) -
                       ((int32_t) x[i].i) * ((int32_t) y[i].i)) >> 15);
  z[i].i = (uint16_t) ((((int32_t) x[i].r) * ((int32_t)y[i].i) +
                       ((int32_t) x[i].i) * ((int32_t) y[i].r)) >> 15);
  }
```

4.1.2 SIMD

Here is where the thing becomes a little complicated. Neither SSE nor AVX extension implement function to immediately perform this operation on fixed point representation (they have FMADD but only in floating point). It is then necessary to create a small algorithm that generates the correct output using intrinsics.
Another important aspect to remember is endianess: Intel CPUs operate in Little-Endian, this means that lower addresses correspond to lower significant bits. As an example, in a SSE 128-bit register, the 4 16-bit fixed point numbers it contains will be distributed as depicted in Figure 2. It is then important to know that data is received in this format and that the output needs to be coherent with it.

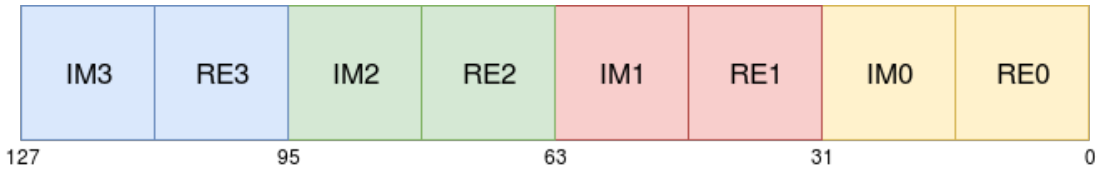

Figure 2: Endianess Example - 4 16-bit Complex Numbers Represented on 128 bits.

The algorithm is the following:

sign : negate imaginary parts of b

madd : multiply and add for real part

srli : shift real parts right to LSB

shuffle : swap adjacent pairs of c and d in order to have columns aligned for the next operation

madd : multiply and add for imaginary part

slli : just erase double sign bit from imaginary part, result is already in MSB

blend : create the final result merging the two vectors

For further explanations regarding the exact operations performed by the intrinsics, please refer to [5].

Listing 5: AVX2 Complex Fixed Point Multiplication - Interleaved Fashion

```
static inline void componentwise_multiply_complex_struct_avx2(
complex16 *x,complex16 *y, complex16 *z, int N){
__m256i *x256 = (__m256i *)x;
__m256i *y256 = (__m256i *)y;
__m256i *z256 = (__m256i *)z;
__m256i const perm_mask=_mm256_set_epi8(
29,28,31,30,25,24,27,26,21,20,23,22,17,16,19,18,
13,12,15,14, 9, 8,11,10, 5, 4, 7, 6, 1, 0, 3, 2);
register __m256i mmtmpb_real;
register __m256i mmtmpb_imag;

int i;
N>>=4;
for(i=0; i<N; i++)
{
  mmtmpb_real     = _mm256_sign_epi16(y256[i],*(__m256i*)reflip_256);
  mmtmpb_real     = _mm256_madd_epi16(x256[i],mmtmpb_real);
  mmtmpb_real     = _mm256_srli_epi32(mmtmpb_real, 15);

  mmtmpb_imag     = _mm256_shuffle_epi8(y256[i],perm_mask);
  mmtmpb_imag     = _mm256_madd_epi16(x256[i],mmtmpb_imag);
  mmtmpb_imag     =_mm256_slli_epi32 (mmtmpb_imag, 1);

  z256[i] = _mm256_blend_epi16(mmtmpb_imag,mmtmpb_real,0xAA);
}
```

4.2 Separated Array Fashion

As already introduced, complex number could be also treated with separated real and complex parts. The functions developed here thus have 4 input arrays instead of 2: a couple for each operand. Obviously output is on two vectors too, one for real and the other for complex.
Actually, due to this particular aspect, the amount of data operated in each iteration is the double than in the other cases, however the total amount of data processed is the same, as the input vectors are half the length of the others.

4.2.1 Scalar

The scalar approach is again straightforward. The code is identical to Listing 4 with the exception that now each vector is indexed separately.

Listing 6: Scalar Complex Fixed Point Multiplication - Separeted Array Fashion

```
for(i=0;i<N;i++){
  x[i] = (uint16_t) ((((int32_t) a[i]) * ((int32_t) c[i]) -
                     ((int32_t) b[i]) * ((int32_t) d[i])) >> 15);
  y[i] = (uint16_t) ((((int32_t) a[i]) * ((int32_t) d[i]) +
                     ((int32_t) b[i]) * ((int32_t) c[i])) >> 15);
}
```

4.2.2 SIMD

Unlike the previous case, here there is no problem related to endianess and data representation, as no structure is involved. However, there is not an instruction able to perform both multiplication and addition at the same time. This means that the two operations need to be performed separately.

The algorithm is pretty simple, first compute the 4 required multiplication and later perform the addition (or subtraction) of the previously computed couples. The resulting code (AVX) can be seen in Listing 7.

Listing 7: AVX2 Complex Fixed Point Multiplication - Separated Array Fashion

```
for(i=0; i<N; i++){
  ac256 = _mm256_mulhrs_epi16(a256[i], c256[i]);
  bc256 = _mm256_mulhrs_epi16(b256[i], c256[i]);
  bd256 = _mm256_mulhrs_epi16(b256[i], d256[i]);
  ad256 = _mm256_mulhrs_epi16(a256[i], d256[i]);

  x256[i] = _mm256_sub_epi16(ac256,bd256);
  y256[i] = _mm256_add_epi16(bc256,ad256);
}
```

5 Simulation Setup and Timing Problems

5.1 Alignment

Alignment is very important for SIMD functions: having unaligned data often causes a strong performance degradation (two memory accesses are required instead of one), also, the compiler will struggle to compile SIMD intrinsics if the memory is not guaranteed to be aligned.
Two solutions are possible for this problem, both equivalent:

- append `__attribute__((aligned(n)))` compiler directive after memory pointers declarations
- use an allocating function able to handle alignment, such as `aligned_alloc(n,size)`

5.2 Compiler optimizations

GCC is the compiler used for this entire project, it supports 6 optimization levels, if you need further information, please refer to [8].

The targets of this experiments are O0, O2 and O3 (O1 is very little used).

O0: no optimization, all the code is traduced almost 1:1 by the compiler. This is the less efficient choice

O2: enable all optimizations but those that require space/performance trade-off

O3: same as O2 but focus on performance and usually enable optimizations that increase the size of the code

One of the optimizations we care much about is **auto-vectorization**: with this feature, the compiler is able to automatically detect parts of the code that could been improved by exploiting SIMD hardware, thus the programmer no more needs to write complex SIMD code. This feature is enabled by default in O3.
Unfortunately, by default, the compiler can not make strong assumptions on the data to be processed, and cannot produce a perfect result. The solution is to give "hints" to the compiler in order to exploit maximum optimization.
One example of this behavior is pointer aliasing, very well explained in [2]. By setting the restrict keyword to parameters in functions declarations, the compiler is able to freely reorder and vectorize loops, leading to better performance.(the very same behavior can be achieved by setting `#pragma GCC ivdep` just before the loop)

5.3 Accurate timing vs operating system

The worst enemy of profilers is for sure the operating system. The OS can cause a lot of nondeterministic overhead to the program due to preemption and context switch, also altering cache and branch predictor status. Multicore processors suffer less from this particular aspect as in general there is less contention on a given core, thus the user application *should* be given more freedom to run.
However, to achieve the best results out of the test machine, something more sophisticated can be

done. Linux kernel provides a nice feature called **isolation**: it is possible to avoid any application to run on one (or more) CPU of the system with the `isolcpus` directive, more information about it could be found in [6]. The running CPU is a quad core system, core 3 was isolated from the rest. After doing this, complete real time is still not guaranteed, it is necessary to set the program as the maximum possible priority. By default Linux uses CFS (Completely Fair Scheduler) that, among the others, implements a FIFO real-time scheduling policy [7]. By setting the code in Listing 8 at the beginning of the C program, it will change itself to the FIFO policy with the maximum priority (99), this will make its execution uninterruptible (to be verified). (be careful, this locks the entire OS until the program is terminates its execution)

Listing 8: Scheduler Policy Change

```
struct sched_param param;
param.sched_priority = 99;
if (sched_setscheduler(0, SCHED_FIFO, &param) != 0) {
  perror("sched_setscheduler");
  exit(EXIT_FAILURE);
  }
```

5.4 Performance counters

Precise clock cycles count can be achieved by reading the performance counters provided by the CPU. The time measured by TSC (accessible with `rdtsc` function) is not exactly cpu clocks, at least not in Haswell CPUs (in general in all those that support DVFS), but CPU clocks referred to a fixed frequency, regardless the actual frequency of the processor. More information related to TSC can be found in [3].
In order to pursuit the best possible cycle count accuracy, an additional library has been used, named **libpfc**. This custom library provides an useful and simple API to the programmer in order to use all the performance counters provided by the CPU, including the variant TSC.(many other possible profiling can be measured, such as issued instructions or cache misses). More information and the installing tutorial can be found in [1].

6 Simulation results

First of all, it is interesting to see the performance change at various compiler optimization levels of SIMD intrinsics coding against a standard scalar approach. According to Figure 3, SIMD intrinsics perform always better at any optimization level for complex "struct" multiplication, similar results have been achieved for the others multiplications.

Other than that, another important evaluation needs to be performed: How do the various SIMD and scalar functions perform at different optimization levels? Figure 4 shows this information for what concerns the real multiplication. For what concerns SIMD intrinsics (orange and light-blue lines), there is approximately no difference between O2 and O3, while there is a huge gap in the scalar counterpart. This is very likely due to auto-vectorization as previously mentioned in Section 5.2. By comparing the disassembled code (not provided here) between O2 and O3, this phenomena is very clear: the multiplication loop has been completely vectorized (a confirmation of this comes from the log produced by `-fopt-info-vect` option passed to GCC).

Another fact worth mentioning is the presence of weird spikes on the figure, noticeable in particular in *O3 SSE4 CMPLX STRUCT* (solid orange line). These spikes can always be found at multiples of 1024 bytes. The root cause of this problem s related to cache size and associativity: cache miss phenomena is made worse when the data size is a multiple of CPU cache's critical stride. In such cases, known as super alignment, previously allocated cache elements are constantly evicted for new elements that share the same cache set and line.

The last comparison is done between the two complex multiplication algorithms. Programming without SIMD intrinsics does not provide competitive results against the handwritten SIMD functions. The speedup ratio is around 4x for SSE4 and 7x for AVX2 in with the struct function, while

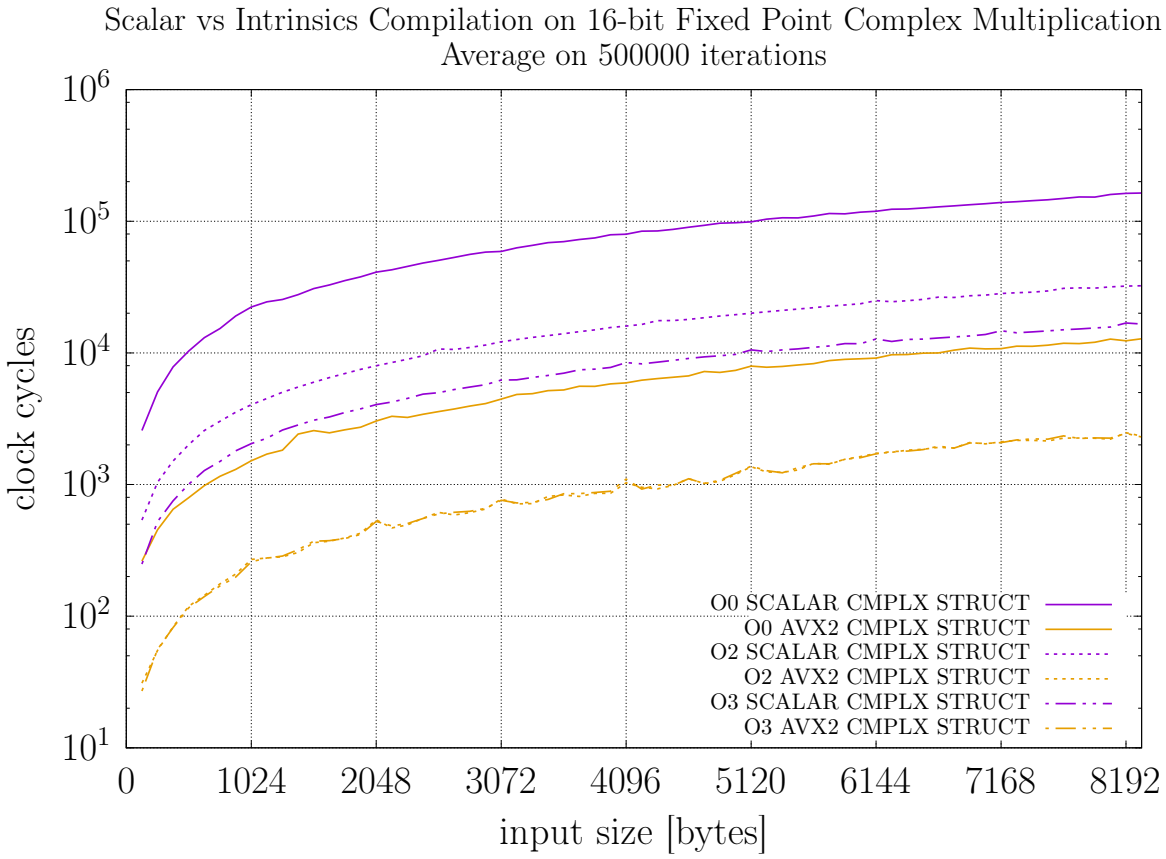

Figure 3: Input dynamic range calculation.

it is between 3x an 1.3x for SSE4 and between 6x and 2.3x in the array function. An overview of the comparison can be seen in Figure 5. It is also interesting to notice that with input size lower than 6144 bytes, the array approach is faster with respect to the struct counterpart, while after this point, the two perform approximately the same.

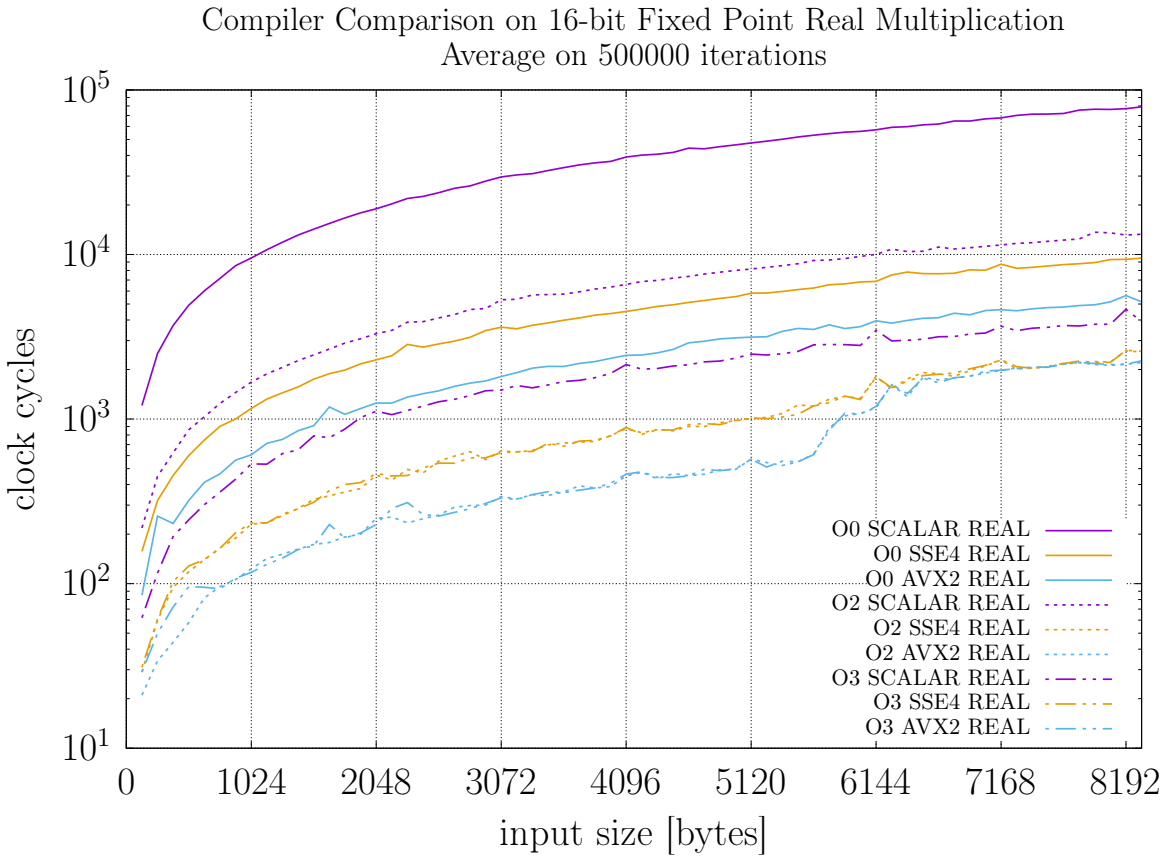

Figure 4: Input dynamic range calculation.

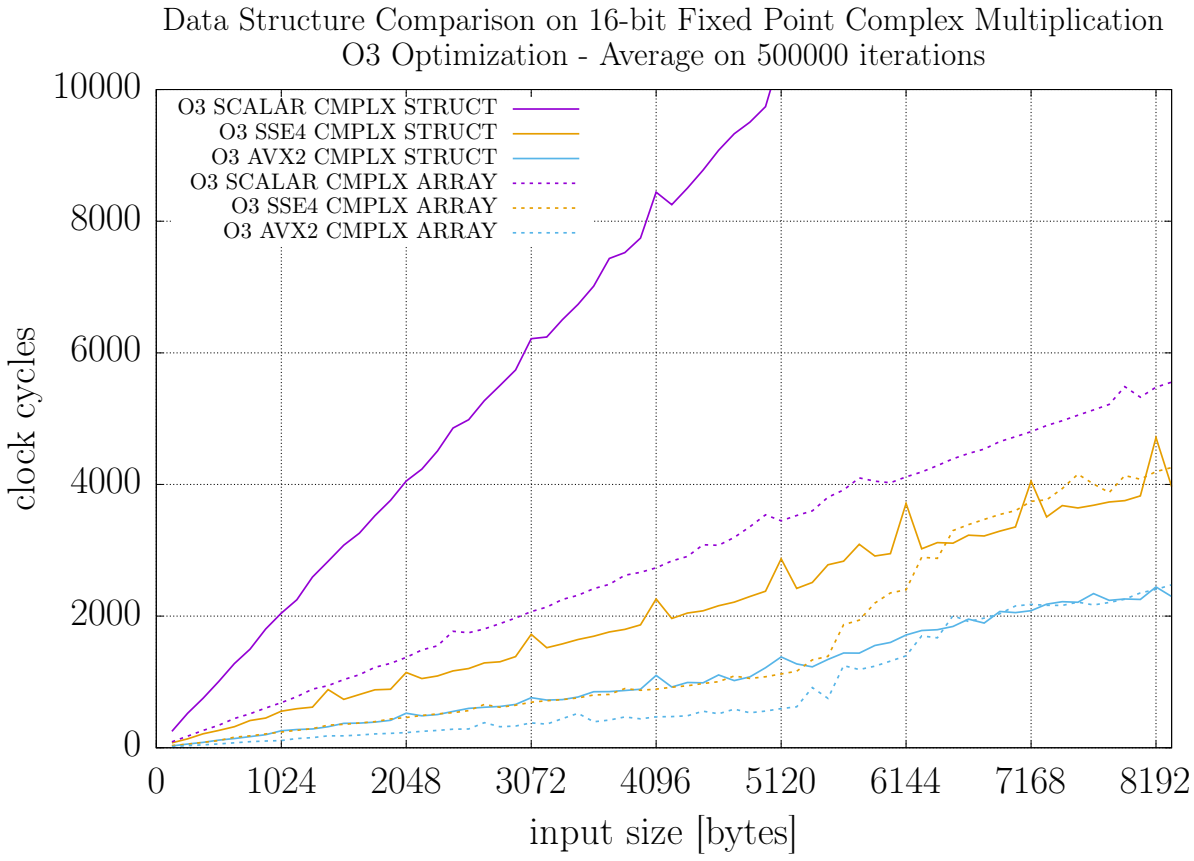

Figure 5: Input dynamic range calculation.

References

[1] O. Bilaniuk. *libpfc - A small library and kernel module for easy access to x86 performance monitor counters under Linux*. URL: https://github.com/obilaniu/libpfc.

[2] D. Cook. *Performance Implications of Pointer Aliasing*. Aug. 1997. URL: ftp://ftp.sgi.com/sgi/audio/audio.apps/dev/aliasing.html.

[3] *Intel® 64 and IA-32 Architectures Software Developer's Manual, Vol 3A*. Mar. 2018. Chap. 17.17. URL: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.

[4] *Intel® Core™ i7-4510U Processor*. Apr. 2014. URL: https://ark.intel.com/products/81015/Intel-Core-i7-4510U-Processor-4M-Cache-up-to-3_10-GHz.

[5] *Intel® Intrinsics Guide*. URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[6] *The kernel's command-line parameters*. URL: https://www.kernel.org/doc/html/v4.14/admin-guide/kernel-parameters.html.

[7] *Tuning the Task Scheduler*. URL: https://doc.opensuse.org/documentation/leap/tuning/html/book.sle.tuning/cha.tuning.taskscheduler.html.

[8] *Using the GNU Compiler Collection (GCC): Optimize Options*. URL: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.