Task 1 Part 1: Vector processing and SIMD

Part 1: Auto-vectorization

Vectorization is a process by which mathematical operations found in loops in scientific code are executed in parallel on special vector hardware found in CPUs and coprocessors. A "vector" is a contiguous set of data of a uniform type, usually floating-point numbers. Each number in the vector is called an element. In vectorized code, basic operations such as addition and multiplication are performed on pairs of small, fixed-sized vectors of numerical values. Corresponding elements in the two small vectors are processed simultaneously by the vector floating-point unit. The net effect of vectorization is a speedup in floating-point computations which ideally is equal to the length of the vectors (number of elements processed simultaneously).

Many compilers vectorize code automatically as part of their code optimization process. This process, however, is not perfect. Certain code constructs can make it difficult or impossible for the compiler to properly assess if floating-point intensive loops can be vectorized. Also, inefficient use of cache and memory can have a negative impact on performance increases obtained by vectorization. As vector lengths have grown in modern CPUs, there is more performance to gain from vectorizing code, and greater penalty for failing to vectorize. This can make it worthwhile to put some amount of effort into removing obstacles that inhibit vectorization, particularly in code sections that consume a lot of time.

Enabling Vectorization

Most compilers are capable of vectorizing code automatically. This section tells you how to instruct the compiler to perform automatic vectorization, as well as how to analyze the compiler's output to identify the areas of code that it was unable to vectorize.

For GCC compilers, automatic vectorization occurs at an optimization level of `-O3`. But it turns out that this may not be sufficient for getting the very best results. GNU compilers (all versions) currently produce SSE instructions by default because SSE is nearly universal across different processor types. Today, however, the 128-bit vector length specified by SSE is often less than optimal. This means that the developer must give additional options to the compiler to generate binary code that is suitable for a more recent target architecture.

For the GNU compilers, the `-march=native` flag (subsequent to `-O3` on the command line) produces a binary suitable for the machine on which the source is compiled. If the target machine is different from the machine the source code is compiled, the architecture-specific option must be provided. For example, for Skylake is `-march=skylake-avx512`.

To get more details about *march* values, two commands can be used:

- The first command tells the compiler not to do any linking (-c), and instead of interpreting the --help option for clarifying command line options, it now shows if certain options are enabled or disabled (-Q). In this case, the options shown are those enabled for the selected target:

  ```
  gcc -c -Q -march=native --help=target
  ```

- The second command will show the compiler directives for building the header file, but without performing the steps and instead showing them on the screen (-###). The final output line is the command that holds all the optimization options and architecture selection:

  ```
  gcc -### -march=native /usr/include/stdlib.h
  ```

The GCC option to disable vectorization altogether is *-fno-tree-vectorize* (must be subsequent to *-O3* on the command line).

Important note: For code with function calls, the compiler may be able to achieve better optimization, including vectorization, if interprocedural optimization is in effect. Enable it using *-fwhole-program* for GCC.

Optimization Reports

Optimization reports provide valuable information related to loops that the compiler has or has not been able to vectorize. For the GNU compilers, the corresponding options are *-fopt-info-all* for a complete optimization report, and *-fopt-info-vec optimized* for just the portion from the vectorizer. GCC's default verbosity lists the loops that were vectorized. To add the loops that were not vectorized use *-fopt-info-vec -missed*:

```
gcc -O2 -ftree-vectorize -fopt-info-vec-all
```

Note: *-ftree-vectorize* option turns on auto-vectorization and it is automatically set when using *-O3*.

The vectorization report is displayed in your terminal window (on stderr) unless *=<filename>* is specified.

Be aware that these reports can be quite lengthy, especially the higher-numbered levels.

Vector reports are essential for determining where the compiler is or is not vectorizing. Remember, the compiler must be conservative when vectorizing; it can sometimes fail to vectorize a loop just because it cannot rule out a potential problem. Vectorization reports allow the developer to locate such situations and implement fixes. It may be possible for the developer to supply hints (or guarantees) that allow the compiler to adjust its

assumptions about potential data dependencies, e.g., or about other constructs that are inhibiting vectorization.

Seeing the assembly code

It will be very helpful to be able to have a look at the assembly code produced by the compiler. This way we will get a much better understanding of how the CPU will actually "see" our program.

No worries, we certainly do not expect you to understand everything that you see in the assembly code. For us it is enough that you can find the most relevant part of the assembly code and get a rough understanding of what are the most relevant instructions there.

To see what the assembly code produced by the compiler looks like, we can simply add the directive `-S` when we invoke the compiler. For example:

```
gcc -S -O3 -march=native <filename>.c
```

This will produce a file called *filename.s* that you can open in a text editor, and it contains precisely the same code that the processor sees when it runs the program.

Unfortunately, the assembly code that the compiler produces is often a bit hard to follow, or to even see which part of it corresponds to which part of the source code. With high optimization settings (`-O3`), there is no straightforward one-to-one correspondence between the source code and the assembly code.

Here is a simple trick that often helps a lot. We simply surround the part of the code with instructions such as

```
asm("# foo");
```

This basically asks the compiler to add the assembly language instruction `# foo` as such in the assembly code it generates. But for the assembler anything that starts with a # symbol is a comment, so the assembler will ignore this. We can however find these comments easily in the assembly code.

Important note: If the "asm" instruction is added to the code of a loop, the loop may not be vectorized.

Exercise 1: GCC auto-vectorization

This exercise will introduce using a vectorizing compiler. We will work with code containing a tight loop that should be easily vectorizable by the compiler. Our goal is to try out various compiler options and compare vectorized with non-vectorized code.

1. Identify the SIMD instructions supported by the microprocessor of the laboratory computer (or your computer). Provide the CPU model and the list of supported SIMD instructions.

2. Obtain a copy of the example code simple2.c. Feel free to review the code. It is a simple program that performs a basic multiply-add of arrays many times within a loop.

3. The GCC compiler will automatically apply vectorization at optimization level -O3. Try the -O3 level and inspect the vectorization report to verify that if any loop has been vectorized. For example:

```
gcc -O3 -march=native -fopt-info-vec-optimized -o simple2
simple2.c

simple2.c:26:9: note: loop vectorized
simple2.c:19:5: note: loop vectorized
```

   This report shows us that two loops were vectorized: the loop that initialized the data values, and the main computation loop. Provide the GCC version and explain the report you get.

4. Now that we have determined which loops are vectorizable, let's look at what happens to the report when we compile with vectorization totally disabled. Call the executable simple2_no_vec.

```
gcc -O3 -march=native -fno-tree-vectorize -fopt-info-vec-
optimized -o simple2_no_vec simple2.c
```

   Here, the compiler does not show any report for a pretty obvious reason, namely, the inclusion of the `-fno-tree-vectorize` flag. Check that.

5. Let's check the assembly code that the compiler generates for the vectorized and no-vectorized code.

```
gcc -S -O3 simple2.c
mv simple2.s simple2_o3.s

gcc -S -O3 -fno-tree-vectorize simple2.c
mv simple2.s simple2_o3_native.s

diff simple2_o3.s simple2_o3_native
```

   Explain the main differences between both assembly codes focused on the SIMD instructions generated by the compiler.

Part 2: Vector intrinsics

The most low-level way to use SIMD is to use the assembly vector instructions directly — they aren't different from their scalar equivalents at all — but we are not going to do that. Instead, we will use intrinsic functions mapping to these instructions that are available in modern C/C++ compilers.

To a programmer, intrinsics look just like regular library functions; you include the relevant header, and you can use the intrinsic. To add four float numbers to another four numbers, use the `_mm_add_ps` intrinsic in your code. In the compiler-provided header declaring that intrinsic (in GCC, is <xmmintrin.h> for SSE), you'll find this declaration:

```
extern __m128 _mm_add_ps( __m128 _A, __m128 _B );
```

But unlike library functions, intrinsics are implemented directly in compilers. The above `_mm_add_ps` SSE intrinsic typically compiles into a single instruction, `addps`. For the time it takes CPU to call a library function, it might have completed a dozen of these instructions.

The `__m128` built-in data type is a vector of four floating point numbers; 32 bits each, 128 bits in total. CPUs have wide registers for that data type, 128 bits per register. Since AVX was introduced in 2011, in current PC processors these registers are 256 bits wide, each one of them can fit eight float values, four double-precision float values, or a large number of integers, depending on their size.

Source code that contains sufficient amounts of vector intrinsics or embeds their assembly equivalents is called *manually vectorized code*. Modern compilers and libraries already implement a lot of stuff with them using intrinsics, assembly, or a combination of the two. For example, some implementations of the memset, memcpy, or memmove standard C library routines use SSE2 instructions for better throughput. Yet outside of niche areas like high-performance computing, game development, or compiler development, even very experienced C and C++ programmers are largely unfamiliar with SIMD intrinsics.

Setup

To use x86 intrinsics, we need to do a little groundwork. First, we need to determine which extensions are supported by the hardware. On Linux, you can call `cat /proc/cpuinfo` and there should be a flags section that lists the codes of all supported vector extensions.

There is also a special CPUID assembly instruction that lets you query various information about the CPU, including the support of particular vector extensions. It is primarily used to get such information in runtime and avoid distributing a separate binary for each microarchitecture. Its output information is returned very densely in the form of feature masks, so compilers provide built-in methods to make sense of it. Here is an example:

```
#include <stdio.h>

int main(void) {
  __builtin_cpu_init();
  printf("%d\n", __builtin_cpu_supports ("sse"));
  printf("%d\n", __builtin_cpu_supports ("avx"));
}
```

Second, we need to include a header file that contains the subset of intrinsics we need, for example <x86intrin.h>. And last, we need to tell the compiler that the target CPU actually supports these extensions. This can be done either with `#pragma GCC target(...)`, or with the `-march=...` flag in the compiler options. If you are compiling and running the code on the same machine, you can set `-march=native` to auto-detect the microarchitecture, as we did before.

SIMD Registers

The most notable distinction between SIMD extensions is the support for wider registers:

- SSE (1999) added 16 128-bit registers called xmm0 through xmm15.
- AVX (2011) added 16 256-bit registers called ymm0 through ymm15.
- AVX512 (2017) added1 16 512-bit registers called zmm0 through zmm15.

As you can guess from the naming, and also from the fact that 512 bits already occupy a full cache line, x86 designers are not planning to add wider registers anytime soon.

C/C++ compilers implement special vector types that refer to the data stored in these registers:

- 128-bit __m128, __m128d and __m128i types for single-precision floating-point, double-precision floating-point and various integer data respectively;
- 256-bit __m256, __m256d, __m256i;
- 512-bit __m512, __m512d, __m512i.

Registers themselves can hold data of any kind: these types are only used for type checking. You can convert a vector variable to another vector type the same way you would normally convert any other type, and it won't cost you anything.

SIMD Intrinsics

Intrinsics are just C-style functions that do something with these vector data types, usually by simply calling the associated assembly instruction.

For example, here is a cycle that adds together two arrays of 64-bit floating-point numbers using AVX intrinsics:

```
double a[100], b[100], c[100];

// iterate in blocks of 4,
// because that's how many doubles can fit into a 256-bit register
for (int i = 0; i < 100; i += 4) {
    // load two 256-bit segments into registers
```

```
    __m256d x = _mm256_loadu_pd(&a[i]);
    __m256d y = _mm256_loadu_pd(&b[i]);

    // add 4+4 64-bit numbers together
    __m256d z = _mm256_add_pd(x, y);

    // write the 256-bit result into memory, starting with c[i]
    _mm256_storeu_pd(&c[i], z);
}
```

The main challenge of using SIMD is getting the data into contiguous fixed-sized blocks suitable for loading into registers. In the code above, we may in general have a problem if the length of the array is not divisible by the block size. There are two common solutions to this:

1. We can "overshoot" by iterating over the last incomplete segment either way. To make sure we don't segfault by trying to read from or write to a memory region we don't own, we need to pad the arrays to the nearest block size (typically with some "neutral" element, e.g., zero).
2. Make one iteration less and write a little loop in the end that calculates the remainder normally (with scalar operations).

Humans prefer #1 because it is simpler and results in less code, and compilers prefer #2 because they don't really have another legal option.

Instruction References

Most SIMD intrinsics follow a naming convention similar to

`_mm<size>_<action>_<type>`

and correspond to a single analogously named assembly instruction. They become relatively self-explanatory once you get used to the assembly naming conventions, although sometimes it does seem like their names were generated by cats walking on keyboards (explain this: `punpcklqdq`).

Here are a few more examples, just so that you get the gist of it:

- `_mm_add_epi16`: add two 128-bit vectors of 16-bit extended packed integers, or simply said, shorts.
- `_mm256_acos_pd`: calculate elementwise arccos for 4 packed doubles.
- `_mm256_broadcast_sd`: broadcast (copy) a double from a memory location to all 4 elements of the result vector.
- `_mm256_ceil_pd`: round up each of 4 doubles to the nearest integer.
- `_mm256_cmpeq_epi32`: compare 8+8 packed ints and return a mask that contains ones for equal element pairs.
- `_mm256_blendv_ps`: pick elements from one of two vectors according to a mask.

As you may have guessed, there is a combinatorially very large number of intrinsics, and in addition to that, some instructions also have immediate values — so their intrinsics

require compile-time constant parameters: for example, the floating-point comparison instruction has 32 different modifiers.

For some reason, there are some operations that are agnostic to the type of data stored in registers, but only take a specific vector type (usually 32-bit float) — you just have to convert to and from it to use that intrinsic. To simplify the examples, we will mostly work with 32-bit integers (epi32) in 256-bit AVX2 registers.

A very helpful reference for x86 SIMD intrinsics is the Intel Intrinsics Guide, which has groupings by categories and extensions, descriptions, pseudocode, associated assembly instructions, and their latency and throughput on Intel microarchitectures. You may want to bookmark that page: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

The Intel reference is useful when you know that a specific instruction exists and just want to look up its name or performance info. When you don't know whether it exists, this cheat sheet may do a better job: https://db.in.tum.de/~finis/x86%20intrinsics%20cheat%20sheet%20v1.0.pdf.

Instruction selection. Note that compilers do not necessarily pick the exact instruction that you specify. Similar to the scalar c = a + b we discussed before, there is a fused vector addition instruction too, so instead of using 2+1+1=4 instructions per loop cycle, compiler rewrites the code above with blocks of 3 instructions like this:

```
vmovapd ymm1, YMMWORD PTR a[rax]
vaddpd  ymm0, ymm1, YMMWORD PTR b[rax]
vmovapd YMMWORD PTR c[rax], ymm0
```

Sometimes, although quite rarely, this compiler interference makes things worse, so it is always a good idea to check the assembly and take a closer look at the emitted vector instructions (they usually start with a "v").

Also, some of the intrinsics don't map to a single instruction but a short sequence of them, as a convenient shortcut: broadcasts and extracts are a notable example.

Moving Data: Aligned Loads and Stores

Operations of reading and writing the contents of a SIMD register into memory have two versions each: `load/loadu` and `store/storeu`. The letter "u" here stands for "unaligned." The difference is that the former ones only work correctly when the read/written block fits inside a single cache line (and crash otherwise), while the latter work either way, but with a slight performance penalty if the block crosses a cache line.

Sometimes, especially when the "inner" operation is very lightweight, the performance difference becomes significant (at least because you need to fetch two cache lines instead of one). As an extreme example, this way of adding two arrays together:

```
for (int i = 3; i + 7 < n; i += 8) {
    __m256i x = _mm256_loadu_si256((__m256i*) &a[i]);
    __m256i y = _mm256_loadu_si256((__m256i*) &b[i]);
    __m256i z = _mm256_add_epi32(x, y);
    _mm256_storeu_si256((__m256i*) &c[i], z);
}
```

…is ~30% slower than its aligned version:

```
for (int i = 0; i < n; i += 8) {
    __m256i x = _mm256_load_si256((__m256i*) &a[i]);
    __m256i y = _mm256_load_si256((__m256i*) &b[i]);
    __m256i z = _mm256_add_epi32(x, y);
    _mm256_store_si256((__m256i*) &c[i], z);
}
```

In the first version, assuming that arrays a, b and c are all 64-byte aligned (the addresses of their first elements are divisible by 64, and so they start at the beginning of a cache line), roughly half of reads and writes will be "bad" because they cross a cache line boundary.

Note that the performance difference is caused by the cache system and not by the instructions themselves. On most modern architectures, the `loadu/storeu` intrinsics should be equally as fast as load / store given that in both cases the blocks only span one cache line. The advantage of the latter is that they can act as free run time assertions that all reads and writes are aligned.

This makes it important to properly align arrays and other data on allocation, and it is also one of the reasons why compilers can't always auto-vectorize efficiently. For most purposes, we only need to guarantee that any 32-byte SIMD block will not cross a cache line boundary, and we can specify this alignment with the `alignas` specifier:

```
alignas(32) float a[n];

for (int i = 0; i < n; i += 8) {
    __m256 x = _mm256_load_ps(&a[i]);
    // ...
}
```

The built-in vector types already have corresponding alignment requirements and assume aligned memory reads and writes — so you are always safe when allocating an array of v8si, but when converting it from int* you have to make sure it is aligned.

Similar to the scalar case, many arithmetic instructions take memory addresses as operands — vector addition is an example — although you can't explicitly use it as an intrinsic and have to rely on the compiler. There are also a few other instructions for reading a SIMD block from memory, notably the non-temporal load and store operations that don't lift accessed data in the cache hierarchy.

Making Constants

If you need to populate not just one element but the entire vector, you can use the `_mm256_setr_epi32` intrinsic:

```
__m256 iota = _mm256_setr_epi32(0, 1, 2, 3, 4, 5, 6, 7);
```

The “r” here stands for “reversed” — from the CPU point of view, not for humans. There is also the `_mm256_set_epi32` (without “r”) that fills the values from the opposite direction. Both are mostly used to create compile-time constants that are then fetched into the register with a block load. If your use case is filling a vector with zeros, use the `_mm256_setzero_si256` instead: it xor-s the register with itself.

GCC Vector Extensions

If you feel like the design of C intrinsics is terrible, you are not alone. Intrinsics are not only hard to use but also neither portable nor maintainable. In good software, you don’t want to maintain different procedures for each CPU: you want to implement it just once, in an architecture-agnostic way.

One day, compiler engineers from the GNU Project thought the same way and developed a way to define your own vector types that feel more like arrays with some operators overloaded to match the relevant instructions.

In GCC, here is how you can define a vector of 8 integers packed into a 256-bit (32-byte) register:

```
typedef int v8si __attribute__ (( vector_size(32) ));
```

The *int* type specifies the *base type*, while the attribute specifies the *vector size* for the variable, measured in bytes. For example, the declaration above causes the compiler to set the mode for the `v8si` type to be 32 bytes wide and divided into *int sized units*. For a 32-bit int this means a vector of 8 units of 4 bytes.

Unfortunately, this is not a part of the C or C++ standard, so different compilers use different syntax for that. There is somewhat of a naming convention, which is to include size and type of elements into the name of the type: in the example above, we defined a “vector of 8 signed integers.” But you may choose any name you want, like vec, reg or whatever.

The main advantage of using these types is that for many operations you can use normal C operators instead of looking up the relevant intrinsic.

```
typedef int v4si __attribute__ (( vector_size(16) ));

v4si a = {1, 2, 3, 5};
v4si b = {8, 13, 21, 34};

v4si c = a + b;

for (int i = 0; i < 4; i++)
    printf("%d\n", c[i]);

c *= 2; // multiply by scalar

for (int i = 0; i < 4; i++)
    printf("%d\n", c[i]);
```

Subtraction, multiplication, and division operate in a similar manner. Likewise, the result of using the unary minus operator on a vector type is a vector whose elements are the negative value of the corresponding elements in the operand.

You can declare variables and use them in function calls and returns, as well as in assignments and some casts. You can specify a vector type as a return type for a function. Vector types can also be used as function arguments. It is possible to cast from one vector type to another, provided they are of the same size (in fact, you can also cast vectors to and from other datatypes of the same size).

You cannot operate between vectors of different lengths or different signedness without a cast.

A port that supports hardware vector operations, usually provides a set of built-in functions that can be used to operate on vectors. For example, a function to add two vectors and multiply the result by a third could look like this:

```
v4si f (v4si a, v4si b, v4si c)
{
  v4si tmp = __builtin_addv4si (a, b);
  return __builtin_mulv4si (tmp, c);
}
```

Note: `__builtin_addv4si` and `__builtin_mulv4si` functions do not exists.

GCC provides a large number of built-in functions other than the ones mentioned above, including `__builtin_cpu_init()` and `__builtin_cpu_supports(…)` used at the beginning of this section. You can read more about them in the "Extensions to the C Language Family" section of the online documentation for your version of the GCC compiler.

With vector types we can greatly simplify the "a + b" loop we implemented with intrinsics before:

```
typedef double v4d __attribute__ (( vector_size(32) ));
v4d a[100/4], b[100/4], c[100/4];

for (int i = 0; i < 100/4; i++)
    c[i] = a[i] + b[i];
```

As you can see, vector extensions are much cleaner compared to the nightmare we have with intrinsic functions. Their downside is that there are some things that we may want to do are just not expressible with native C/C++ constructs or built-in functions, so we will still need intrinsics for them. Luckily, this is not an exclusive choice, because vector types support zero-cost conversion to the `_mm` types and back:

```
v8f x;
int mask = _mm256_movemask_ps((__m256) x)
```

There are also many third-party libraries for different languages that provide a similar capability to write portable SIMD code and also implement some, and just in general are nicer to use than both intrinsics and built-in vector types.

Exercise 2: Use of intrinsics

This exercise will introduce the use of vector intrinsics. We will work with the `simple2.c` program from previous exercise. Our goal is to vectorized the code manually.

- Obtain a copy of the example code *simple2.c* and rename it as *simple2_intrinsics.c*. Remember it is a simple program that performs a basic multiply-add of arrays many times within a loop.

- Our target loop is the inner loop of the second loop. Before the second loop, we declare two `__m256d` variables: one to store the constant value 1.0001 (the m value) and the second to store the partial sums:

```
__m256d mm = {1.0001, 1.0001, 1.0001, 1.0001};
__m256d sum = {0.0, 0.0, 0.0, 0.0}; // to hold partial sums
```

- Now, we must change the increment value of index *i* from *i++* to *i += 4* because we want to operate 4 doubles in parallel (vector of 256 bits).

```
for (i=0; i < ARRAY_SIZE; i += 4) {
```

- Inside the loop, first we have to load the 4 doubles from the input arrays to vectors and then, compute c += m*a+b. To do that, we must split the operation in two operations: tmp = m*a+b and then c += tmp:

```
// Load arrays
__m256d va = _mm256_load_pd(&a[i]);
__m256d vb = _mm256_load_pd(&b[i]);

// Compute m*a+b
__m256d tmp = _mm256_fmadd_pd (mm, va, vb);
// Accumulate results
sum = _mm256_add_pd (tmp, sum);
```

- Now *sum* holds summations of all products in four parts and we want a scalar result. There are two options. The simplest one is to performs the vector sum with C arithmetic operators:

```
for (i = 0; i < 4; i++) {
  c += sum[i];
}
```

The other option is to perform vector sum with intrinsics:

```
// Get sum[2], sum[3]
__m128d xmm = _mm256_extractf128_pd (sum, 1);

// Extend to 256 bits: sum[2], sum[3], 0, 0
__m256d ymm =  _mm256_castpd128_pd256(xmm);

// Perform sum[0]+sum[1], sum[2]+sum[3], sum[2]+sum[3], 0+0
sum = _mm256_hadd_pd (sum, ymm);
```

```
// Perform sum[0]+sum[1]+sum[2]+sum[3]…
sum = _mm256_hadd_pd (sum, sum);
c = sum[0];
```

See the Intel Intrinsics Guide to read the description and understand the operation of the intrinsics used above.

- Print the final value (c variable) of this version and compare it with the original *simple2.c* program to be sure *simple2_intrinsics.c* program does not change the output. Add the necessary code to both programs. Change the value of NUMBER_OF_TRIALS to 1 to make it easier to compare results.

- Vectorize the first (initialization) loop using intrinsics. Tip: Create initial vectors.

- Compare the execution time for different values of NUMBER_OF_TRIALS: from 100.000 to 1.000.000 in steps of 100.000. Plot the result in a graph. Explain the results.

  Note: Use *gettimeofday()* before and after the second loop to get the execution time.

Exercise 3: Vectorize an image processing algorithm

The file *greyScale.c* includes a program that applies an image processing algorithm to convert any image to grey scale. This algorithm has been provided to us, as a proof of concept, with the idea that it finally processes images from a video stream in real time. To do this, students are reminded that a video is usually composed of approximately 30 fps (frames per second). That is, the program would have to process 30 images in one second.

Answer the questions reasonably:

0. Compile and run the program using some images as arguments. Examine the results that were generated and analyze briefly the provided program.
1. The program includes two loops. The first loop (indicated as Loop 0) iterates over the arguments applying the algorithm to each of them. The second loop (indicated as Loop 1) computes the grey scale algorithm. Is this loop optimal to be vectorized? Why? Tip: We recommend using Compiler Explorer (https://godbolt.org/).
2. We know that data access order is important to allow auto-vectorization. Please correct the code to help the compiler to vectorize the loop. Explain your changes.
   a. It is imperative that the program continue to perform the same algorithm, so only changes should be made to the program that do not change the output.
3. Vectorize the grey scale algorithm using intrinsics. Fill in a table with time and speedup results compared to the original version and auto-vectorized version for images of different resolutions (SD, HD, FHD, UHD-4k, UHD-8k). You must include a column with the fps at which the program would process. Discuss the results.
   a. It is imperative that the program continue to perform the same algorithm, so only changes should be made to the program that do not change the output.
   b. Explain how you translate the serial algorithm to a vectorized version and the intrinsics used.

Material to submit

You must write a report answering the questions proposed in each exercise, plus the requested files. Submit a zip file through Moodle. Check submission date in Moodle (deadline is until 11:59 pm of that date).

- From exercise 1:
  - CPU model and list the supported SIMD instructions of your CPU.
  - Explain the main differences between both assembly codes (vectorized and non-vectorized) focused on the SIMD instructions generated by the compiler.
- From exercise 2:
  - Provide the source code of *simple2_intrinsics.c* after the vectorization of the loops.
  - Explain how you have carried out the vectorization of the code.
  - Plot the results of the experiment and explain them.
- From exercise 3:
  - Answer all the questions in the report.
  - Provide the source code of the auto-vectorized version of the code.
  - Provide the source code after manually vectorizing the code. Explain your solution.
  - Create a table with the results of the experiment and explain them.

References

- Sergey Slotin, “Algorithms for Modern Hardware”, available online: https://en.algorithmica.org/hpc/. Section SIMD Parallelism.