

Task 1: Vector processing and SIMD

Vector Processing

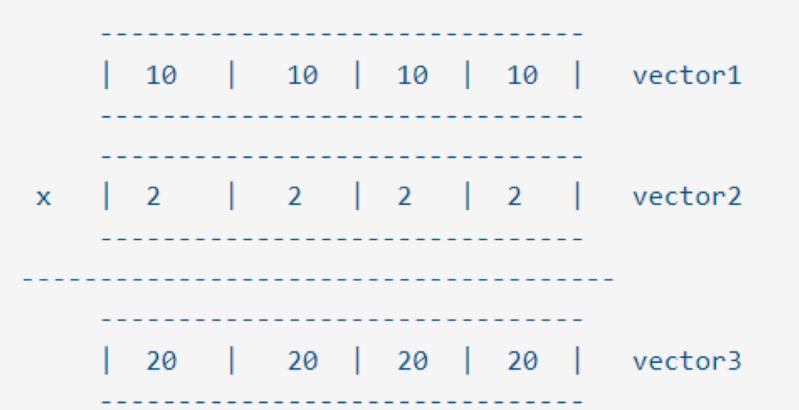
Vector processing and SIMD

- ❖ All modern processors are actually “vector processors” under the hood.
 - ❖ Modern processors are able to compute one-dimensional arrays of data.
- ❖ Every time you write `float s = a + b;` you’re leaving a lot of performance on the table.
 - ❖ The processor could have added four float numbers to another four numbers, or even eight numbers to another eight numbers if that processor supports AVX.
 - ❖ Similarly, when you write `int i = j + k;` to add 2 integer numbers, you could have added four or eight numbers instead, with corresponding SSE2 or AVX2 instructions.

Vector processing and SIMD

- ❖ SIMD stands for “Single Instruction Multiple Data”
 - ❖ The same operation happens to all the data (the values in the vector) at the same time.
- ❖ Each value in the vector is computed independently.
 - ❖ Vector operations include logic and math.
 - ❖ Math within a single vector is called horizontal math.
 - ❖ Math between two vectors is called vertical math.

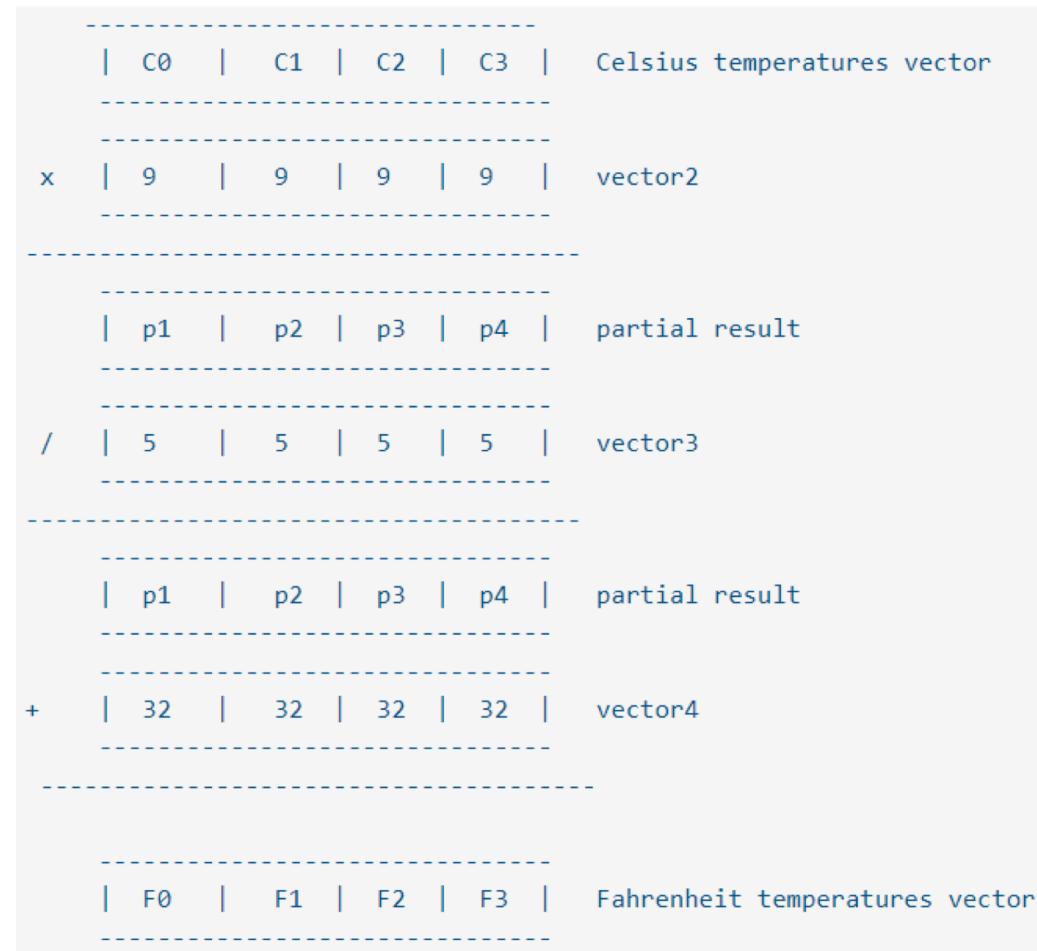
All 10s are multiplied by
all 2s at the same time.



- ❖ SIMD instructions are available on many platforms: AMD64, ARM, etc.

Vector processing and SIMD

- ❖ For example, to convert Celsius to Fahrenheit using $F = (9/5) * C + 32$ for a vector of temperatures in Celsius:



Note:

$$F = (9/5)C + 32 \rightarrow F = (9*C)/5 + 32$$

Vector processing and SIMD

- ❖ Typically, general-purpose microprocessors use modular arithmetic, in which values exceeding the maximum value "wrap around" to the minimum value, like the hours on a clock passing from 12 to 1.
 - ❖ In hardware, modular arithmetic with a minimum of zero and a maximum of $2^n - 1$, can be implemented by simply discarding all but the lowest n digits.
- ❖ Vector processing units were initially designed for signal processing and graphics processing.
 - ❖ For example, imagine we want to add/subtract two grayscale images together, we would be losing a lot of time if we had to clip the result by hand after the operation.

Vector processing and SIMD

- ❖ Fortunately, vector processing units support saturation arithmetic.
- ❖ Saturation arithmetic is like normal arithmetic except that when the result of the operation that would overflow or underflow an element in the vector, is clamped at the end of the range and not allowed to wrap around
 - ❖ For example, with saturation arithmetic in the range [0, 255], we have:
 - ❖ $200 + 70 = 255$
 - ❖ $20 - 70 = 0$
- ❖ Vector processing can be applied to floating and integer math.
 - ❖ Integer math is useful in case your vector hardware is integer-only or if integer math is much faster than floating-point math.
 - ❖ Integer math will be an approximation of floating-point math, but you might be able to get a faster answer that is acceptably close.

Vector processing and SIMD

- ❖ The first option in integer math is rearranging operations. If your formula is simple enough, you might be able to rearrange operations to preserve precision.
 - ❖ For example, you could rearrange $F = (9/5)C + 32$ to $F = (9*C)/5 + 32$.
 - ❖ If $9 * C$ doesn't overflow the type you're using, precision is preserved.
 - ❖ Precision is lost if you divide by 5; so, do the division after the multiplication.
 - ❖ Rearrangement may not work for more complex formulas.

Vector processing and SIMD

- ❖ The second choice is scaled math. In the scaled math option, you decide how much precision you want, then multiply both sides of your equation by a constant, round or truncate your coefficients to integers, and work with that. The final step to get your answer then is to divide by that constant.
 - ❖ $F = (9/5)C + 32 = 1.8C + 32$ -- but we can't have 1.8, so multiply by 10
 - ❖ $\text{sum} = 10F = 18C + 320$ -- 1.8 is now 18: now all integer operations
 - ❖ $F = \text{sum}/10$
- ❖ If you multiply by a power of 2 instead of 10, you change that final division into a shift, which is almost certainly faster, though harder to understand.

Vector processing and SIMD

- ❖ The third choice for integer math is shift-and-add. Shift-and-add is another method based on the idea that a floating-point multiplication can be implemented with several shifts and adds.
 - ❖ So, our troublesome $1.8C$ can be approximated as:
 - ❖ $1.0C + 0.5C + 0.25C + \dots$ or $C + (C \gg 1) + (C \gg 2) + \dots$
- ❖ Again, it's almost certainly faster, but harder to understand.

From MMX to AVX in x86 ISA

- ❖ MMX
 - ❖ 64-bit MMX registers for integers (MM0-MM7)
- ❖ SSE (Streaming SIMD Extensions)
 - ❖ SSE-1: 128-bit XMM registers for integers and single-precision floating point (XMM0-XMM7)
 - ❖ SSE-2: Double-precision floating point (XMM0-XMM15)
 - ❖ SSE-3, SSSE3 (supplemental): horizontal math
 - ❖ SSE-4: New instructions (not multimedia specific), shuffle operations
- ❖ AVX (Advanced Vector Extensions)
 - ❖ AVX: 256-bit floating point (YMM0-YMM15)
 - ❖ AVX2: 256-bit floating point with FMA (Fused Multiply Add)
 - ❖ AVX-512: 512-bit (ZMM0-ZMM31)

https://www.intel.sg/content/dam/www/public/apac/xa/en/pdfs/ssg/Intel_Processor_Architecture SIMD_Instructions.pdf

https://gcc.gnu.org/wiki/cauldron2014?action=AttachFile&do=get&target=Cauldron14_AVX-512_Vector_ISA_Kirill_Yukhin_20140711.pdf

Intel SIMD extensions

ISA	MMX	SSE, SSE2, SSE3, SSE4.1, SSE4.2	AVX, AVX2	AVX512
Tamaño vector	64 bits	128 bits	256 bits	512 bits
Registros	mm	xmm	ymm	zmm
Tipo dato single	<code>__m64</code>	<code>__m128</code>	<code>__m256</code>	<code>__m512</code>
Tipo dato double	<code>__m64, double</code>	<code>__m128d</code>	<code>__m256d</code>	<code>__m512d</code>
Tipo dato int	<code>__m64, long</code> <code>long, int64_t</code>	<code>__m128i</code>	<code>__m256i</code>	<code>__m512i</code>
Prefijo	mm	mm	mm256	mm512

Auto-vectorization

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.
- ❖ Corresponding elements in two 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).

Auto-vectorization

- ❖ 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.
 - ❖ 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 auto-vectorization

- ❖ For GCC compilers, automatic vectorization occurs at an optimization level of -O3.
 - ❖ GNU compilers (all versions) currently produce SSE instructions by default because SSE is nearly universal across different processor types.
 - ❖ 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) produces a binary suitable for the machine on which the source is compiled.
 - ❖ If the target machine is different, the architecture-specific option must be provided.
 - ❖ For example, for Skylake is -march=skylake or -march=skylake-avx512

Enabling auto-vectorization

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

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

- ❖ Shows if certain options are enabled or disabled (-Q). In this case, the options shown are those enabled for the selected target.

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

- ❖ 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

Enabling auto-vectorization

- ❖ Optimization reports for GCC:
 - ❖ Provide valuable information related to loops that the compiler has or has not been able to vectorize.
 - ❖ -fopt-info-all for a complete optimization report
 - ❖ -fopt-info-vec-optimized for the vectorized part of the report (default)
 - ❖ -fopt-info-vec-missed for the non-vectorized part of the report
- ❖ The vectorization report is displayed in your terminal window (on stderr) unless =<filename> is specified
- ❖ Note: -ftree-vectorize option turns on auto-vectorization and it is automatically set when using -O3. The option to disable vectorization is -fno-tree-vectorize

Enabling auto-vectorization

- ❖ Assembly Code
 - ❖ It is very helpful to get a much better understanding of how the CPU will actually “see” our program.
 - ❖ 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.

Exercise : GCC Auto-vectorization

- ❖ Identify SIMD ISA of your processor
- ❖ Apply auto-vectorization (example code simple2.c)
- ❖ Generate vectorization reports
- ❖ Compare assembly code for vectorized and non-vectorized code

Compiler Explorer

Compiler Explorer

❖ <https://godbolt.org/>

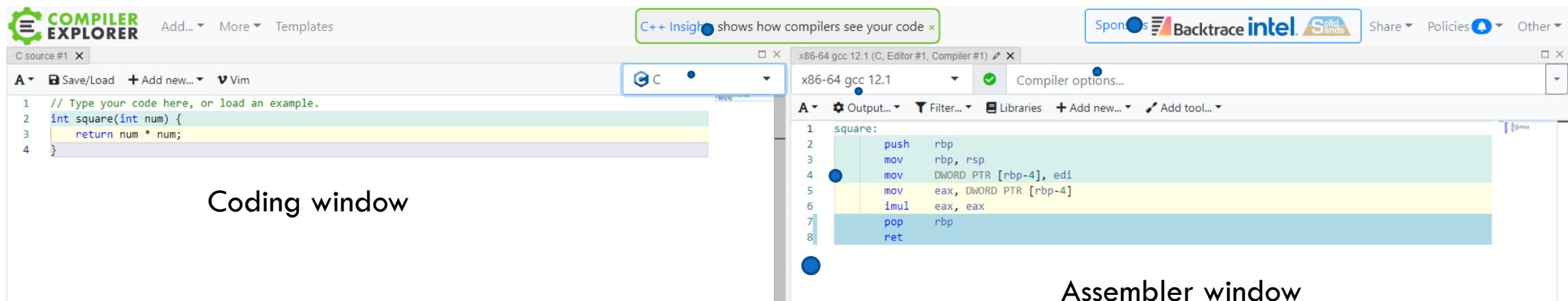
The screenshot shows the Compiler Explorer interface with two tabs: 'C++ source #1' and 'x86-64 gcc 12.2'. The C++ source code is:

```
1 // Type your code here, or load an example.
2 int square(int num) {
3     return num * num;
4 }
```

The assembly output is:

```
1 square(int):
2     push    rbp
3     mov     rbp, rsp
4     mov     DWORD PTR [rbp-4], edi
5     mov     eax, DWORD PTR [rbp-4]
6     imul   eax, eax
7     pop    rbp
8     ret
```

Select prog. language and compiler version



Auto-vectorization

The Compiler Explorer interface shows the C source code 'simple2.c'. The code defines two arrays, 'a' and 'b', of size 2048, and performs a computation where each element of array 'c' is the sum of elements from arrays 'a' and 'b' at indices from 0 to i-1. The code uses SIMD operations like vpbroadcastd and vmovdqa.

```
#include <stdlib.h>
#include <stdio.h>

#define ARRAY_SIZE 2048
#define NUMBER_OF_TRIALS 1000000

/*
 * Statically allocate our arrays. Compilers can
 * align them correctly.
 */
static double a[ARRAY_SIZE], b[ARRAY_SIZE], c;

int main(int argc, char *argv[]) {
    int i, t;
    double m = 1.0001;

    /* Populate A and B arrays */
    for (i=0; i < ARRAY_SIZE; i++) {
        b[i] = i;
        a[i] = i+1;
    }

    /* Perform an operation a number of times */
    for (t=0; t < NUMBER_OF_TRIALS; t++) {
        for (i=0; i < ARRAY_SIZE; i++) {
            c += m*a[i] + b[i];
        }
    }

    return 0;
}
```

simple2.c

The Compiler Explorer interface shows the assembly output for the code and compiler messages. The assembly output includes SIMD instructions like vpbroadcastd, vmovdqa, vcvtdq2pd, vmovapd, vpaddd, vextracti128, vcvtdq2pd, and vmovapd. The compiler messages indicate that the loop was vectorized using 32 byte vectors.

```
x86-64 gcc 12.1 (C, Editor #1, Compiler #1) x
x86-64 gcc 12.1 -O3 -march=native -fopt-info-vec-all
main:
    mov    edx, 8
    vpbroadcastd ymm2, edx
    vmovdqa ymm3, YMMWORD PTR .LC0[rip]
    mov    edx, 1
    xor    eax, eax
    vpbroadcastd ymm1, edx
.L2:
    vmovdqa ymm0, ymm3
    vcvtdq2pd ymm4, xmm0
    vmovapd YMMWORD PTR b[rax], ymm4
    vextracti128 xmm4, ymm0, 0x1
    vpaddd ymm0, ymm0, ymm1
    vcvtdq2pd ymm4, xmm4
    vmovapd YMMWORD PTR b[rax+32], ymm4

Output (0/8) x86-64 gcc 12.1 - 2964ms (102628) ~521 lines filtered
Output of x86-64 gcc 12.1 (Compiler #1)
:25:17: missed: couldn't vectorize loop
:27:21: missed: not vectorized: no vectype for stmt: _4 = a[i_29];
scalar_type: double
:26:21: optimized: loop vectorized using 32 byte vectors
:19:17: optimized: loop vectorized using 32 byte vectors
:13:5: note: vectorized 2 loops in function.
:25:17: note: **** Analysis failed with vector mode V4DF
:25:17: note: **** Skipping vector mode V32QI, which would repeat the analysis for V4DF
Compiler returned: 0
```

Output window

Compiler options

Click on a message to see the related line of code

Vector intrinsics

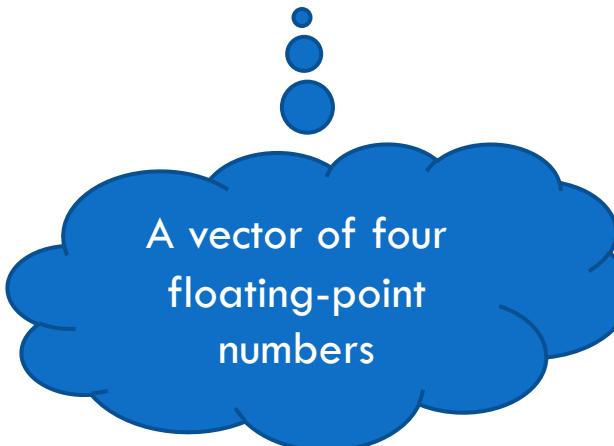
Vector intrinsics

- ❖ Low-level way to use SIMD is to use assembly vector instructions directly
 - ❖ But we will use intrinsic functions mapping vector instructions
- ❖ Intrinsics look like regular library functions
 - ❖ Include the header, for example <xmmmintrin.h> for SSE
 - ❖ Use the intrinsic, for example, to add four floats:

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

Vector intrinsics

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



A vector of four floating-point numbers



Compiles into a single instruction
addps

- ❖ Since AVX these registers are 256 bits wide...
 - ❖ They can fit eight float values, four double-precision float values, or many integers, depending on their size
- ❖ ... and in AVX-512 these registers are 512 bits wide!

Vector intrinsics

- ❖ Before use intrinsics, determine which extensions are supported by your hardware
 - ❖ In Linux: `cat /proc/cpuinfo`
- ❖ There is a special CPUID assembly instruction to check the support at runtime:

```
#include <stdio.h>

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

SIMD 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`.

SIMD Extensions x86 SSE/AVX

- **SSE**

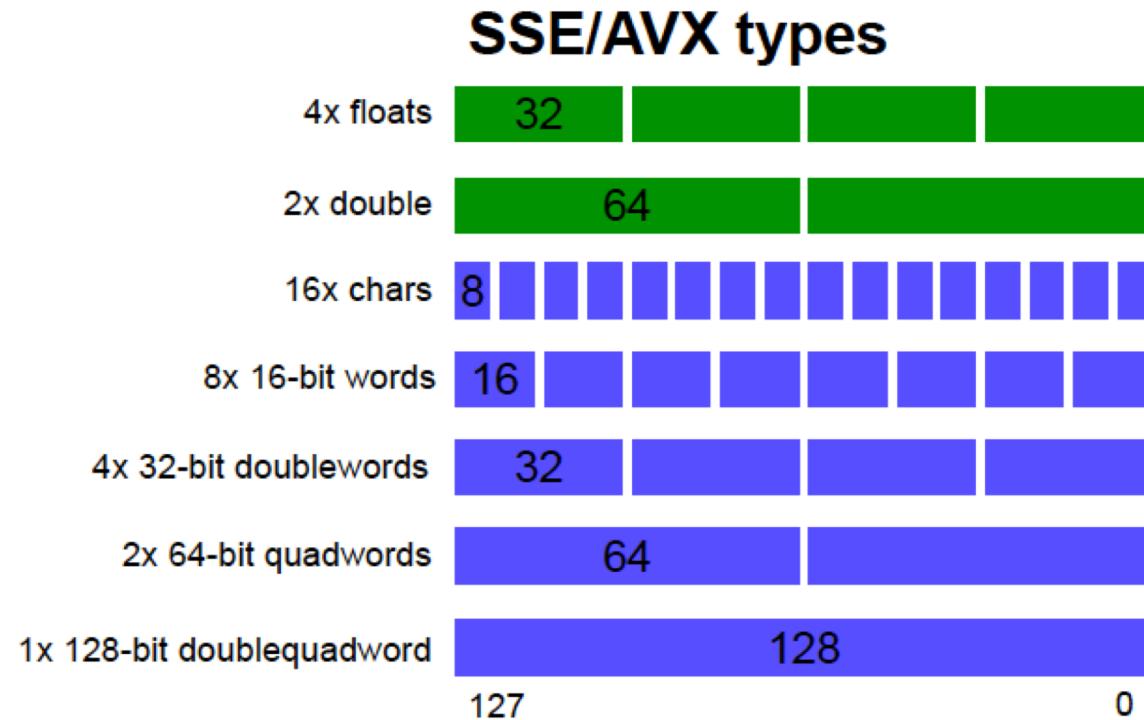
- $16 \times 128\text{-bit SIMD registers XMM0—XMM15}$ (in x86-64 mode)

- **AVX2**

- $16 \times 256\text{-bit SIMD registers YMM0—YMM15}$

- **AVX-512**

- $32 \times 512\text{-bit SIMD registers ZMM0—ZMM31}$



AVX2 types

Source: https://software.intel.com/sites/default/files/m/d/4/1/d/8/Intro_to_Intel_AVX.pdf



Intel SIMD extensions

_mm256_abs_epi16

- ❖ Tamaño del vector
- ❖ Operación
- ❖ Tipo de dato:
 - ❖ ps: packed *single* (*precision*).
 - ❖ pd: packed *double* (*precision*).
 - ❖ epiN: extended packed *integer* (*N bits*).
 - ❖ epuN: extended packed *unsigned integer* (*N bits*).
 - ❖ si: *single integer*.

Example of 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);  
}
```

Data MUST be stored contiguous in memory

Problem if the length of the array is not divisible by 4

Example of 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);
}
```

‘u’ means “unaligned”
(cross a cache line)

Example of 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);
}
```

Example of 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);
}
```

‘u’ means “unaligned”
(cross a cache line)

Examples of SIMD intrinsics

- ❖ `_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.
- ❖ Check Intel Intrinsics Guide for more details
<https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html>

Declare SIMD constants

```
__m256 values = {0, 1, 2, 3, 4, 5, 6, 7};
```

or

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

- ❖ “r” means “reversed” from the CPU point of view.
- ❖ There is also a version without “r”.

- ❖ Filling a vector with zeros, use the `_mm256_setzero_si256`

Exercise 2: Use of intrinsics

- ❖ Learn how to apply intrinsics to vectorize the inner loop of the second loop of simple2.c
- ❖ Vectorize the first loop (initialization)
- ❖ Compare non-vectorized vs vectorized.

Exercise 2: Tips

```
/* Perform an operation a number of times */  
for (t=0; t < NUMBER_OF_TRIALS; t++) {  
    for (i=0; i < ARRAY_SIZE; i++) {  
        c += m*a[i] + b[i];  
    }  
}
```

```
/* Populate A and B arrays */  
for (i=0; i < ARRAY_SIZE; i++) {  
    b[i] = i;  
    a[i] = i+1;  
}
```

__m256d m = {1.0001, 1.0001, 1.0001, 1.0001};

This operation is very common. Maybe an intrinsic computes it.

Think in parallel... store partial sums.

Create a vector __m256d sum = {0.0, 0.0, 0.0, 0.0};
And after the loop, reduce the partial sums.

```
c = 0;  
for (i = 0; i < 4; i++)  
    c += sum[i]; // vector sum is an array ☺
```

Initialize b vector as __m256d b = {0, 1, 2, 3}.
Similar idea for a.
And then you can add the same constant vector
to both vectors ¿which values?

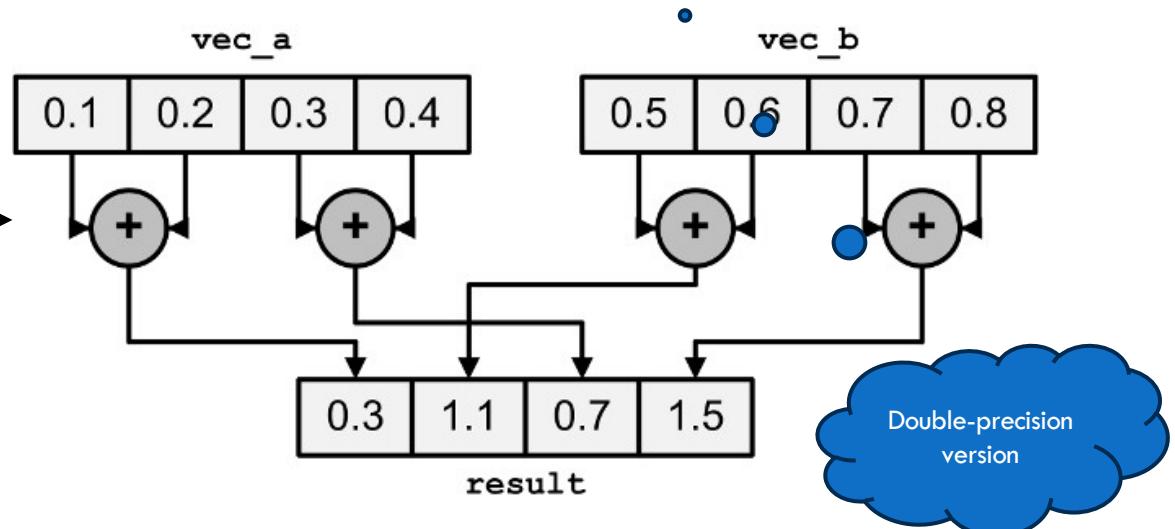
Exercise 4: Vectorize an image proc. algorithm

- ❖ Compile, execute and review the code.
- ❖ Apply auto-vectorization and answer the questions.
- ❖ Make changes to help auto-vectorization.
- ❖ Manually vectorized (intrinsics) the loop that computes the grey scale. Compare this version to the original version and auto-vectorized version

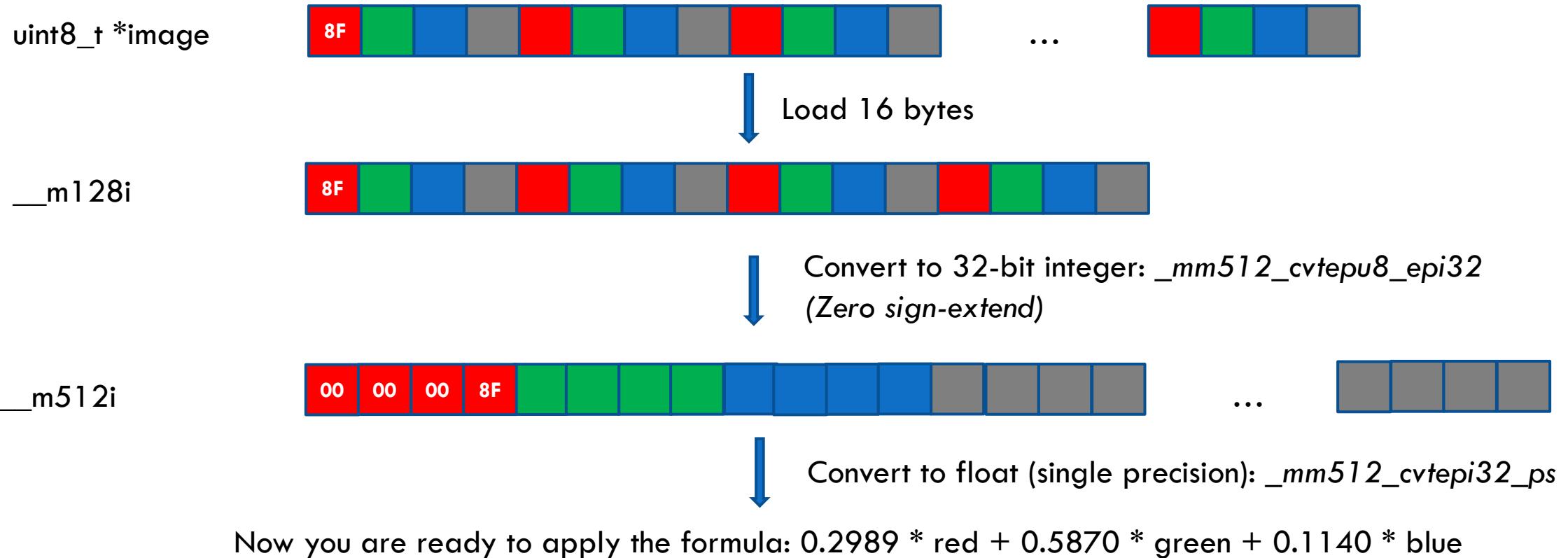
Exercise 4: Tips

- ❖ Sometimes more work is faster. You can read the complete pixel (4 bytes) and use arithmetic/logic operations to not consider the 4th byte.
- ❖ Convert chars into doubles before compute. Take care with sign extension.
- ❖ Use ideas from exercise 2.
- ❖ Horizontal add can be useful. → Think how to use it.

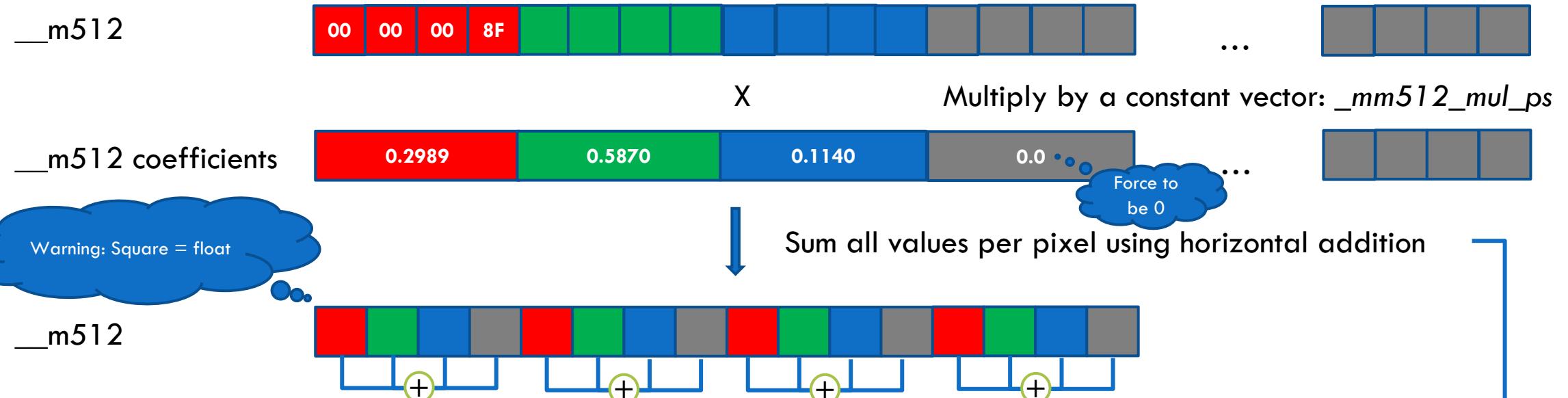
```
_m256d result = _mm256_hadd_pd(vec_a, vec_b);
```



Exercise 4: Solution AVX-512 (512 bits)



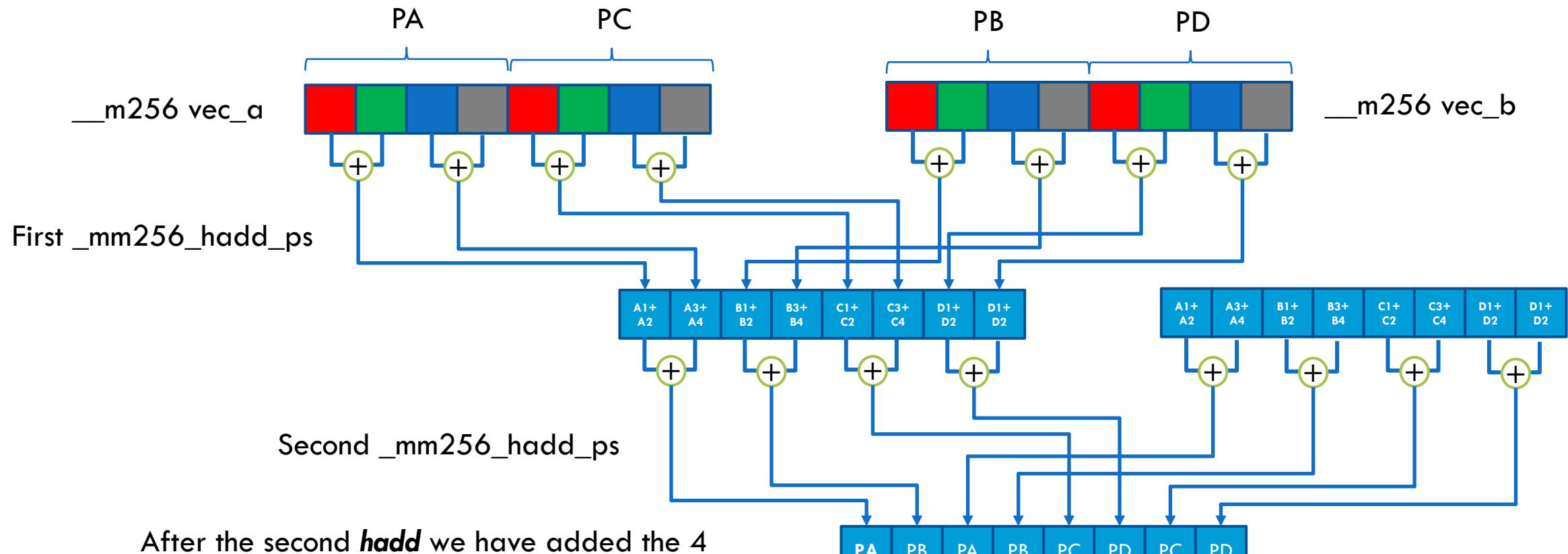
Exercise 4: Solution AVX-512 (512 bits)



hadd that operates with `_mm256` so:

1. Split `_mm512` in two `_mm256` using `_mm512_extractf32x8_ps`
2. You have to vectors: `_mm256 vec_a` and `_mm256 vec_b`
3. Compute ***hadd*** using `vec_a` and `vec_b`. The result is adding only two “bytes” of each pixel. We must add the four bytes!!!!!!

Exercise 4: Solution AVX-512 (512 bits)



Exercise 4: Solution AVX-512 (512 bits)

_m256



Reorder the values: `_mm256_permutevar8x32_ps`
Second parameter is the index (permutation) ↗values?

_m256



Convert to 32-bit integer:
`_mm_cvtps_epi32(_mm256_extractf128_ps ())`

_m128i

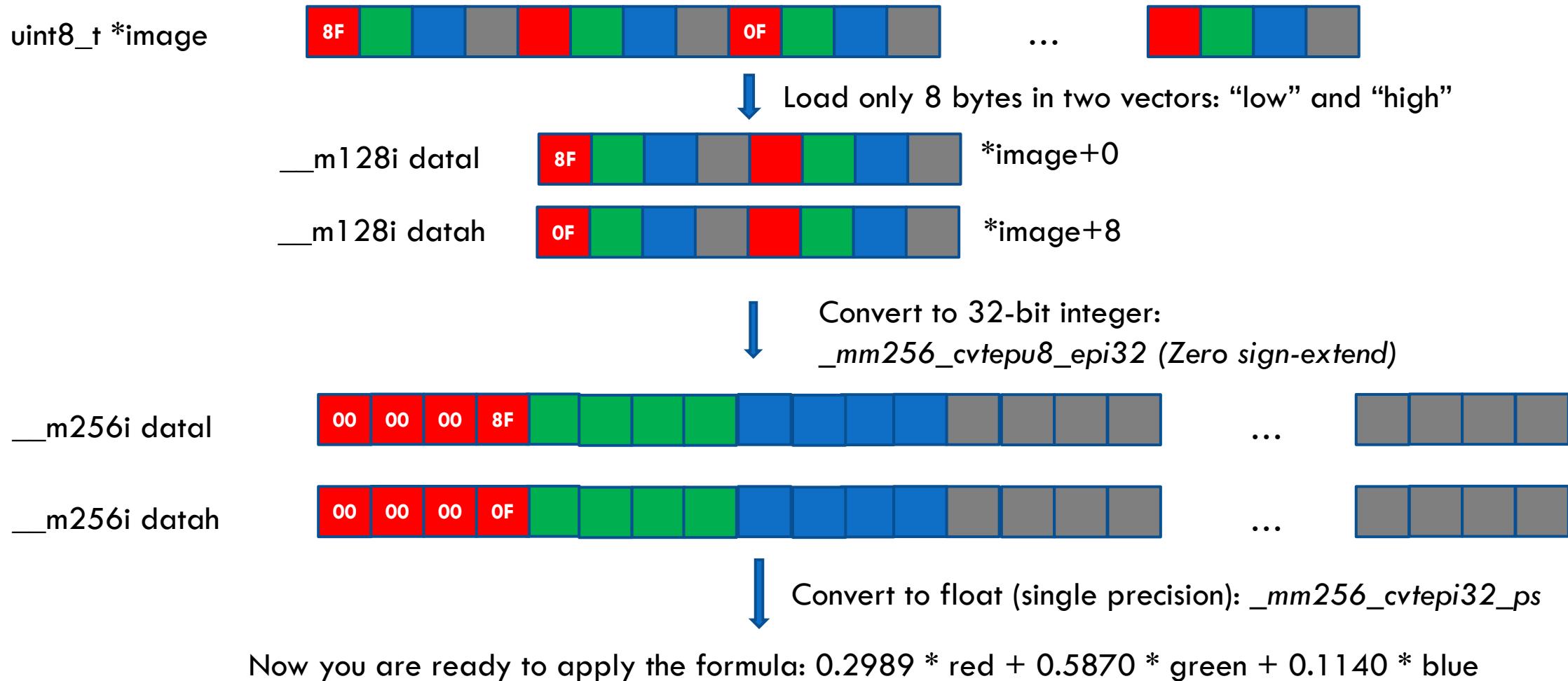


Store in memory

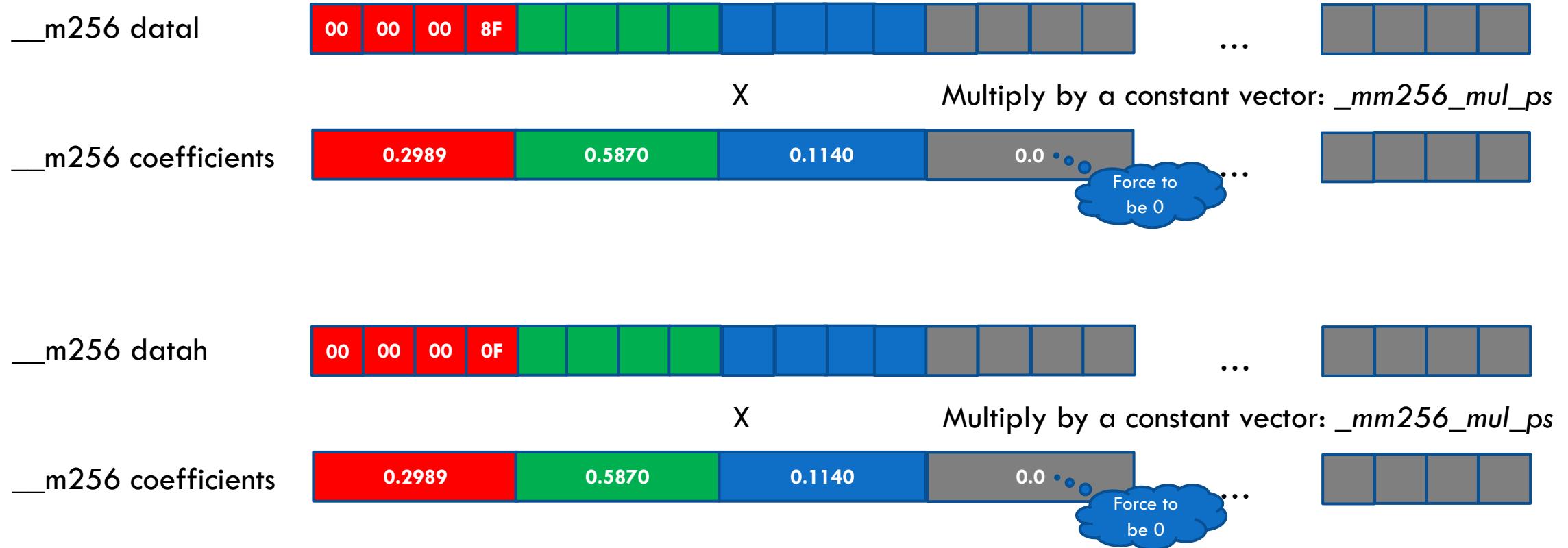
`uint32_t *greyimage`



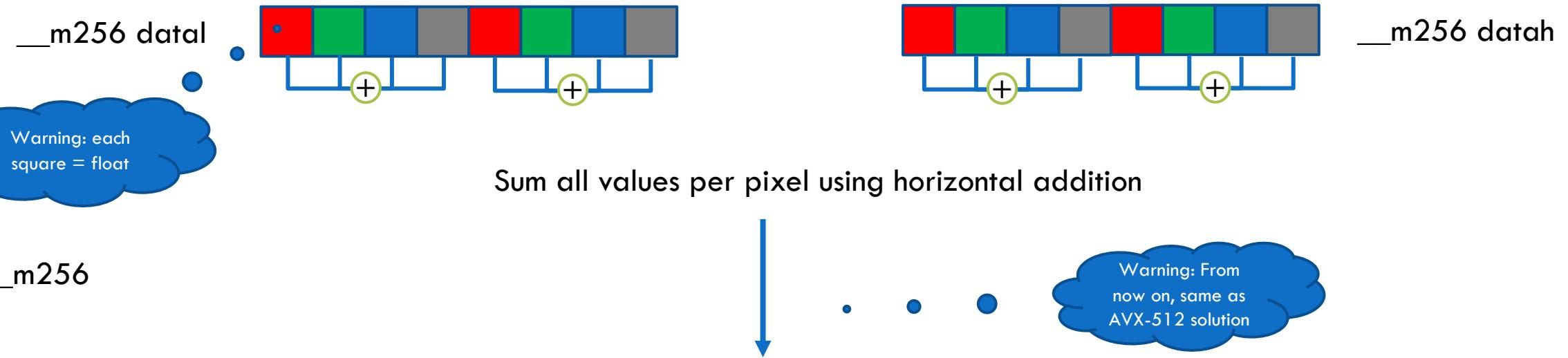
Exercise 4: Solution AVX2 (256 bits)



Exercise 4: Solution AVX2 (256 bits)



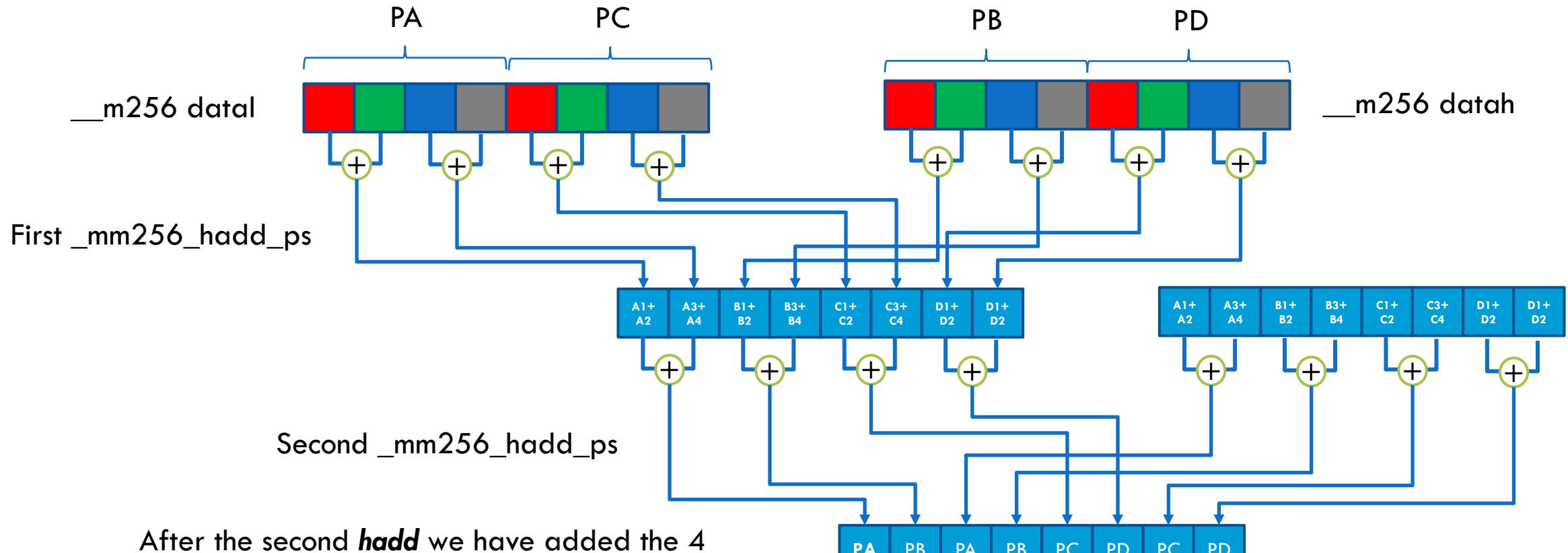
Exercise 4: Solution AVX2 (256 bits)



`hadd` operates with `_mm256` so:

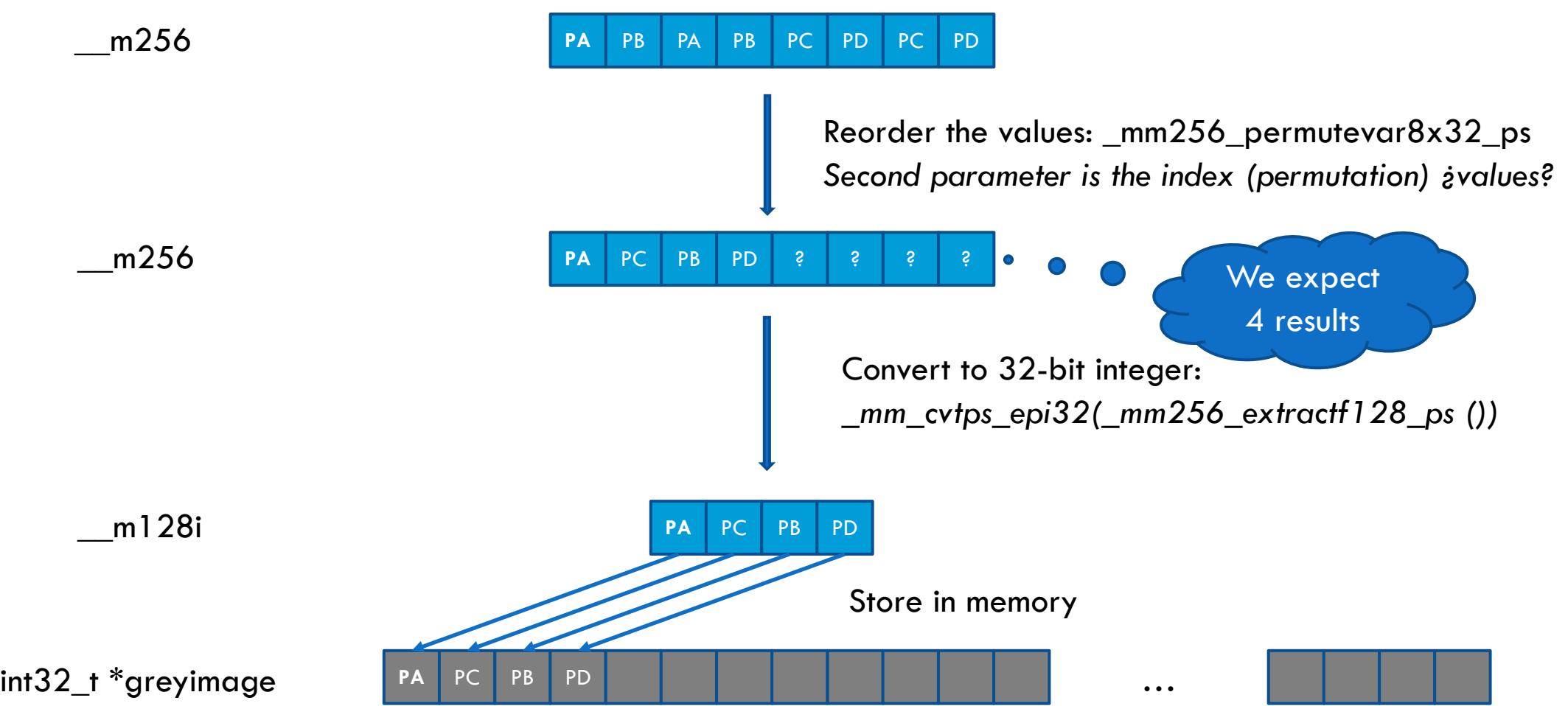
1. You have to vectors: `_mm256 data` and `_mm256 datah`
2. Compute **`hadd`** using `data` and `datah`. The result is adding only two “bytes” of each pixel. We must add the four bytes!!!!!!

Exercise 4: Solution AVX2 (256 bits)



After the second `hadd` we have added the 4 bytes of each pixel, but the result is out of order and includes duplicates.

Exercise 4: Solution AVX2 (256 bits)



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).