NEON Intrinsics plus C++ 11 Threading

This program gives you experience with SIMD programming *and* threading. On the ARM, the SIMD (vector) instruction set is called NEON. In this project you will leverage the NEON instruction set directly **but without using assembly language** per se.

Instead, we will use *intrinsics*. An intrinsic is a higher level language passthrough directly to assembly language.

The idea of this program is to sum two vectors (large arrays of floats) together placing the results in an equally sized third vector. Each of the source vectors will be populated with floating point numbers between 0 and 1 chosen at random. This means the average of each source vector is 0.5.

You will write four different (but related) solutions:

- single threaded naive implementation (just c++)
- multithreaded version using the naive implementation
- single threaded version using intrinsics
- multithreaded version using intrinsics

Partnership Rules

All work is solo.

Command line options

```
p5 $> ./a.out -h
help:
-h       prints this
-s size  sets size (default is 128 - will be made divisible by 16)
-i iter  sets iterations (default is 1)
p5 $>
```

`size` is the number of floats that will be operated upon in 3 arrays. In your code, make sure that size is made to be a multiple of 16 if the user supplied value is not.

`iter` is the number of times each version of the code you will write is run. You will use `gettimeofday` to get the time before starting a loop of `iter` times calling each version. After the last iteration, you will use `gettimeofday` a second time to compute an elapsed time for all of the repeated calls to your function. Then compute the average elapsed time. By looping a number of times, changes in machine load should average out.

Step 1 - First version - just get basics working

You'll want these:

```
#include <iostream>
#include <getopt.h>
#include <cstdlib>
#include <iomanip>
#include <cassert>
#include <arm_neon.h>
#include <vector>
#include <thread>
#include <sys/time.h>
#include <cstring>
```

You'll need to compile with something like this:

```
g++ -std=c++11 -pthread main.cpp
```

Later something like this that enables full optimization:

```
g++ -std=c++11 -O3 -pthread main.cpp
```

You will write a program that creates 3 float arrays of the size specified on the command line (after ensuring the size is a multiple of 16). I'll call these `a`, `b`, and `c`.

Allocate these dynamically using a memory aligned version of `malloc` like so:

```
float * a = (float *) aligned_alloc(16, size * sizeof(float));
```

Remember to ensure that `size` is a multiple of 16.

While doing `aligned_alloc` may not be strictly necessary in this program, they often are in programs that use special instructions such as SIMD instructions. Now you know how to do that.

C++ 17 adds an alignment option to `new` but we're using c++ 11.

In `main()` call `srand()` with the current time to get non-repeated random number seeds.

Write a function that returns random numbers between 0 and 1 **inclusive**.

Fill `a` and `b` with these random numbers (between 0 and 1 inclusive). Each array element will average out at 0.5. On average, when `a[i]` is added to `b[i]` you'll get around 1 in `c[i]`.

Write a function that initializes one array completely to zero. Pass `c` to this function **before launching each** of what will become four loops (four different timing tests). This is to ensure that old summed values are not being reused due to a bug in your code. This function should be implemented using `memset`. That is, **points will be deducted if you code your own loop.**

I give you this function below:

```
void SingleCore(float * a, float * b, float * c, int size);
```

It adds each entry in `a` to the same entry in `b` and puts the result in the same slot in `c`. This function will be your baseline for timing and also to calculate a known correct result.

Also write:

```
double SumOfSums(float * p, int size);
```

This adds up the first `size` floats at the given address and returns the sum. `SumOfSums()` will be used to validate your later variations of `SingleCore()`.

Note that `SumOfSums()` returns a `double`.

Here is my `SingleCore()`:

```
void SingleCore(float * a, float * b, float * c, int size) {

    //cout << __FUNCTION__ << " " << hex << size << dec << " " << size << endl;
    // Ensure compatibility with a 16-times unrolled loop.
    assert((size & 0xF) == 0);
    size = size / 16;

    for (int i = 0; i < size; i++) {
        *(c++) = *(a++) + *(b++);
        *(c++) = *(a++) + *(b++);
        *(c++) = *(a++) + *(b++);
        *(c++) = *(a++) + *(b++);
        *(c++) = *(a++) + *(b++);
        *(c++) = *(a++) + *(b++);
        *(c++) = *(a++) + *(b++);
        *(c++) = *(a++) + *(b++);
        *(c++) = *(a++) + *(b++);
        *(c++) = *(a++) + *(b++);
        *(c++) = *(a++) + *(b++);
        *(c++) = *(a++) + *(b++);
        *(c++) = *(a++) + *(b++);
        *(c++) = *(a++) + *(b++);
        *(c++) = *(a++) + *(b++);
        *(c++) = *(a++) + *(b++);
        *(c++) = *(a++) + *(b++);
        *(c++) = *(a++) + *(b++);
        *(c++) = *(a++) + *(b++);
    }
}
```

I have unrolled the loop to account for a guarantee of the length being a multiple of 16. I specifically did this to provide a more *apples to apples* comparison to the SIMD instructions which work on 4 floats simultaneously with one instruction. **I unrolled my SIMD loop 4 times (4 x 4 == 16).**

Call `SingleCore` to add `a` to `b` placing the results into `c` as stated above.

Print out something like this:

```
p5 $> ./a.out -s 500000 -i 10
Array size: 500000 total size in MB: 5
Iterations: 10
Single core timings...
Naive: 0.002438 Check: 500289.843750
p5 $>
```

This made arrays of 500000 floats **(which is rounded up to a multiple of 16 if necessary)** and then SingleCore was timed running 10 times.

The value after `Check:` is the result of the call to `SumOfSums()` giving it the `c` array from the last loop iteration. In one invocation of the program, all subsequent `Check:` print outs should produce the same value if your code is correct.

Step 2 - SIMD

See here for a simple concrete example.

SIMD means *single instruction multiple data*. ARM offers the NEON instruction set. These are assembly language instructions that provide many of the same operations as the standard instructions you've already used such as add, subtract, multiply etc.

However, the NEON instructions differ in an important way. They operate over several registers in parallel. In fact, the NEON instruction set adds a **new** set of registers to the ARM CPU designed for this purpose.

Intrinsics are proxies for assembly language instructions that are callable directly from C and C++.

Here is a link to a document describing NEON intrinsics and the whole underlying instruction set architecture. By the way, the only way little ARM machines (like a Chromecast, Firestick, your phone, etc) can process video so well is by virtue of extended instruction sets like NEON.

For example, `vaddq_f32()` is a proxy for the `VADD.F32` instruction. This is the intrinsic used for this project.

To use an intrinsic in C or C++ we need new types. We are using floats, four at a time. The type that represents this is: `float32x4_t`. Having the right type is necessary for type checking purposes but more importantly in this case, for computing the size (four floats).

I am not going to tell you much about `vaddq_f32()` other than it takes *floats (plural)* in `a` and `b` and returns *floats (plural)* for `c`.

Write a function with the same signature as `SingleCore()`.

```
void Intrinsic(float * a, float * b, float * c, int size)
```

Use intrinsics to duplicate the functionality of the ordinary version.

Now you should be able to produce this (compiled with `-O3`):

```
$ ./a.out -s 50000000 -i 10
Array size: 50000000 total size in MB: 572
Iterations: 10
Available cores: 2
Single core timings...
Naive: 0.014291 Check: 50003000.761409
NEON:  0.010942 Check: 50003000.761409
$
```

These timings were taken on a Linux VM on my shiny new ARM-based Apple M1 so yours will be different.

Notice all values of `Check:` are the same. This is because `a` and `b` didn't change but the way `a[i]` was added to `b[i]` to yield `c[i]` switched from non-SIMD to SIMD.

If you are running on an ARM emulator / VM, your timings will be slower (possibly a lot slower).

Step 3 - C++ Threading (C++ 11 and later)

C++ threading is an abstraction and wrapper of `pthreads` which is why some compilers might require `-pthread` on the command line to compile and link properly.

There are some wrinkles like all arguments to the thread worker are passed by value.

There is a way around this if you want to pass by reference. Notice, I designed the computation function to take pointers to the arrays.

Here is a link to the cplusplus web site reference on C++ threads.

I am repeating their example here:

```
// thread example
#include <iostream>       // std::cout
#include <thread>         // std::thread

void foo()
{
  // do stuff...
}

void bar(int x)
```

```
{
  // do stuff...
}

int main()
{
  std::thread first(foo);     // spawn new thread that calls foo()
  std::thread second(bar,0);  // spawn new thread that calls bar(0)

  std::cout << "main, foo and bar now execute concurrently...\n";

  // synchronize threads:
  first.join();                // pauses until first finishes
  second.join();               // pauses until second finishes

  std::cout << "foo and bar completed.\n";

  return 0;
}
```

Launching a thread is as easy as making a thread object. The arguments to the thread constructor include the name of the thread worker and the arguments that should be passed to it. Notice the thread constructor has variable arguments. The thread constructor devolves into a template that matches the arguments to the thread worker against the signature of the thread worker. Being a template, if anything goes wrong you may get thousands of errors. Have fun with that.

You will launch as many threads as available cores. Get the number of available cores with `hardware_concurrency()`. Your ARM VM starter `alias` is configured to create 2 cores.

Notice you have to `join` every thread you launch in order to release their completed spirit back to the universe. Since you don't know in advance how many cores you have, you have to store each thread in some kind of data structure. I used a `vector`. There may be some unexpected syntax problems in doing this. Tutorials online that are readily found give examples.

You next write a loop that breaks down the total work into chunks and assign each chunk to a processor. If you have four cores, each core will do one quarter of the work.

You must use the same computation routines that you have finished writing by the time you get here.

Remember that your computing functions take addresses. Inside the loop starting one thread per core, factor this in to ensure each thread processes the right data.

Here is a wrinkle

You know the number of floats in the arrays are divisible by 16 as this is a stated requirement in this specification.

The number of multiples of 16 may not be a multiple of the number of cores you have. You have to account for this. *If there is a remainder, one of your threads need to get a larger size than the others to account for stragglers.*

After dividing size by the number of cores, *that* result may not be a factor of 16. Therefore, you may need to shrink size a little more of each chunk sent to each core. *If there is a remainder, one of your threads need to get a larger size than the others to account for it.*

At this point I have repeated *If there is a remainder, one of your threads need to get a larger size than the others to account for it.* twice. Just saying.

After getting multithreading working with the `SingleCore` function, you will be able to produce this:

```
p5 $> ./a.out -s 500000 -i 10
Array size: 500000 total size in MB: 5
Iterations: 10
Available cores: 4
Single core timings...
Naive: 0.008689 Check: 499725.843750
NEON:  0.004288 Check: 499725.843750
Threaded timing...
Naive: 0.006911 Check: 499725.843750
p5 $>
```

And finally, when completed for the NEON case:

```
$ ./a.out -s 500000 -i 10
Array size: 500000 total size in MB: 5
Iterations: 10
Available cores: 4
Single core timings...
Naive: 0.008689 Check: 499725.843750
NEON:  0.004288 Check: 499725.843750
Threaded timing...
Naive: 0.006911 Check: 499725.843750
NEON:  0.005136 Check: 499725.843750
$
```

and here it is on my Mac Studio:

```
% ./a.out -s 500000 -i 10
Array size: 500000 total size in MB: 5
Iterations: 10
```

```
Available cores: 10
Single core timings...
Naive: 0.000171 Check: 499739.350804
NEON:  0.000062 Check: 499739.350804
Threaded timing...
Naive: 0.000151 Check: 499739.350804
NEON:  0.000119 Check: 499739.350804
%
```

Wowser!

I want you to try out various sizes of arrays. Look for patterns.

Also, if you're running on an ARM emulator, change the number of cores. You start your emulator with something like:

```
alias ARM='qemu-system-aarch64 -M virt -m 1024 -cpu cortex-a53 -kernel vmlinuz-4.9.0-8-arm64
```

The final argument is `-smp 2`. This sets the number of cores. Try setting it to a larger number and repeat all experiments.

Timing

The idea is this:

```
- Capture the current time (start)
- Run all the iterations of one function
- Capture the current time (end)
- Subtract start from end to get the elapsed time
- Divide by the number of iterations to get the average
```

For example:

```
gettimeofday(&start, NULL);
for (int i = 0; i < loops; i++) {
    SingleCore(a, b, c, size);
}
gettimeofday(&end, NULL);
timersub(&end, &start, &elapsed);
```

I am *specifically* not completing the calculation of average time - you need to derive how that is done for yourselves by reading up on `timeval`.

Setting expectations

With some comments, my solution comes to about 210 lines. This is not a challenge but is provided so as to set your level of expectations for this project.

An old grading rubric

The following is a grading rubric I used years ago. It is incomplete. I removed a bunch of the standard project independent stuff to concentrate on what to expect in this project.

There is no guarantee that this rubric will be applied again this year.

```
-10     Warnings
-10     Joins threads immediately after launching them
-10     Very inefficient use of the language (no or little pointer arithmetic)
-5      Computation of average is wrong
-5      Inefficient use of the language (not enough use of pointer smarts)
-5      Inclusion of system("pause") a la Windows as this cannot have worked on Linux
-10     Did not implement iterations correctly.
-10     Does not free dynamically allocated memory
-20     Actually made effort to blow away memory
-10     Approach to dividing work among cores is awkward or worse
-5      Gross overwork due to not using getopt
```