Parallel Programming @ NYCU, Fall 2021

This is the webpage for the Parallel Programming course

Programming Assignment I: SIMD Programming

Parallel Programming by Prof. Yi-Ping You

Due date: **23:59, Oct 14, 2021**

The purpose of this assignment is to familiarize yourself with SIMD (single instruction, multiple data) programming. Most modern processors include some form of vector operations (i.e., SIMD instructions), and some applications may take advantage of SIMD instructions to improve performance through vectorization. Although modern compilers support automatic vectorization optimizations, the capabilities of compilers to fully auto-vectorize a given piece of code are often limited. Fortunately, almost all compilers (targeted to processors with SIMD extensions) provide SIMD intrinsics to allow programmers to vectorize their code explicitly.

Table of Contents

- Programming Assignment I: SIMD Programming
  - Due Date:
  - Evaluation Platform
  - Part 1: Vectorizing Code Using Fake SIMD Intrinsics
  - Part 2: Vectorizing Code with Automatic Vectorization Optimizations
    - 2.1 Turning on auto-vectorization
    - 2.2 Adding the __restrict qualifier
    - 2.3 Adding the __builtin_assume_aligned intrinsic
    - 2.4 Turning on AVX2 instructions
    - 2.5 Performance impacts of vectorization
    - 2.6 More examples
  - Requirements
  - Grading Policy
  - Submission
  - References

Evaluation Platform

Your program should be able to run on UNIX-like OS platforms. We will evaluate your programs on the workstations dedicated for this course. You can access these workstations by `ssh` with the following information. (To learn how to use `ssh` and `scp`, you can refer to the video listed in the references)

The workstations are based on Ubuntu 20.04 with Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz processors. `g++-10` (used in part1) and `clang++-11` (used in part2) have been installed.

IP	Port	User Name	Password
140.113.215.195	37072 ~ 37080	{student_id}	{student_id}

Login example:

```
$ ssh <student_id>@140.113.215.195 -p <port>
```

Please download the code and unzip it:

```
$ wget https://nycu-sslab.github.io/PP-f21/HW1/HW1.zip
$ unzip HW1.zip -d HW1
$ cd HW1
```

1. Part 1: Vectorizing Code Using Fake SIMD Intrinsics

Take a look at the function `clampedExpSerial` in `part1/main.cpp` of the Assignment I code base. The `clampedExp()` function raises `values[i]` to the power given by `exponents[i]` for all elements of the input array and clamps the resulting values at 9.999999. Your job is to vectorize this piece of code so it can be run on a machine with SIMD vector instructions.

Please enter the `part1` folder:

```
$ cd part1
```

Rather than craft an implementation using *SSE* or *AVX2* vector intrinsics that map to real SIMD vector instructions on modern CPUs, to make things a little easier, we're asking you to ***implement your version using PP's "fake vector intrinsics" defined in*** `PPintrin.h`. The `PPintrin.h` library provides you with a set of vector instructions that operate on vector values and/or vector masks. (These functions don't translate to real CPU vector instructions, instead we simulate these operations for you in our library, and provide feedback that makes for easier debugging.)

As an example of using the PP intrinsics, a vectorized version of the `abs()` function is given in `main.cpp`. This example contains some basic vector loads and stores and manipulates mask registers. Note that the `abs()` example is only a simple example, and in fact the code does not correctly handle all inputs! (We will let you figure out why!) You may wish to read through all the comments and function definitions in `PPintrin.h` to know what operations are available to you.

Here are few hints to help you in your implementation:

- Every vector instruction is subject to an optional mask parameter. The mask parameter defines which lanes whose output is "masked" for this operation. A 0 in the mask indicates a lane is masked, and so its value will not be overwritten by the results of the vector operation. If no mask is specified in the operation, no lanes are masked. (Note this equivalent to providing a mask of all ones.)
- *Hint:* Your solution will need to use multiple mask registers and various mask operations provided in the library.
- *Hint:* Use `_pp_cntbits` function helpful in this problem.
- Consider what might happen if the total number of loop iterations is not a multiple of SIMD vector width. We suggest you test your code with `./myexp -s 3`.
- *Hint:* You might find `_pp_init_ones` helpful (use it to initialize any mask!).
- *Hint:* Use `./myexp -l` to print a log of executed vector instruction at the end. Use function `addUserLog()` to add customized debug information in log. Feel free to add additional `PPLogger.printLog()` to help you debug.

The output of the program will tell you if your implementation generates correct output. If there are incorrect results, the program will print the first one it finds and print out a table of function inputs and outputs. Your function's output is after "output = ", which should match with the results after "gold = ". The program also prints out a list of statistics describing utilization of the PP fake vector units. You should consider the performance of your implementation to be the value "Total Vector Instructions". (You can assume every PP fake vector instruction takes one cycle on the PP fake SIMD CPU.) "Vector Utilization" shows the percentage of vector lanes that are enabled.

See the requirements to finish this part.

> The following part is not required for this assignment, but you are encouraged to do it for practice.
>
> Once you have finished part 1, it is time for vectorizing the code using real SIMD intrinsics and see if the program can really get the benefits from vectorization. Vectorize the same piece of code in part 1 so it can be run on a machine with SIMD vector instructions.
>
> Intrinsics are exposed by the compiler as (inline) functions that are not part of any library. Of course the SIMD intrinsics depend on the underlying architecture, and may differ from one compiler to other even for a same SIMD instruction set. Fortunately, compilers tend to standardize intrinsics prototype for a given SIMD instruction set, and we only have to handle the differences between the various SIMD instruction sets.

2. Part 2: Vectorizing Code with Automatic Vectorization Optimizations

Take the exercises below and answer questions Q2-1, Q2-2, and Q2-3.

We are going to start from scratch and try to let the compiler do the brunt of the work. You will notice that this is not a "flip a switch and everything is good" exercise, but it also requires effort from the developer to write the code in a way that the compiler knows it can do these optimizations. The goal of this assignment is to learn how to fully exploit the optimization capabilities of the compiler such that in the future when you write code, you write it in a way that gets you the best performance for the least amount of effort.

Please enter the `part2` folder:

```
$ cd part2
```

Auto-vectorization is enabled by default at optimization levels `-O2` and `-O3`. We first use `-fno-vectorize` to disable automatic vectorization, and start with the following simple loop (in `test1.cpp`):

```
void test1(float* a, float* b, float* c, int N) {
  __builtin_assume(N == 1024);

  for (int i=0; i<I; i++) {
    for (int j=0; j<N; j++) {
      c[j] = a[j] + b[j];
    }
  }
}
```

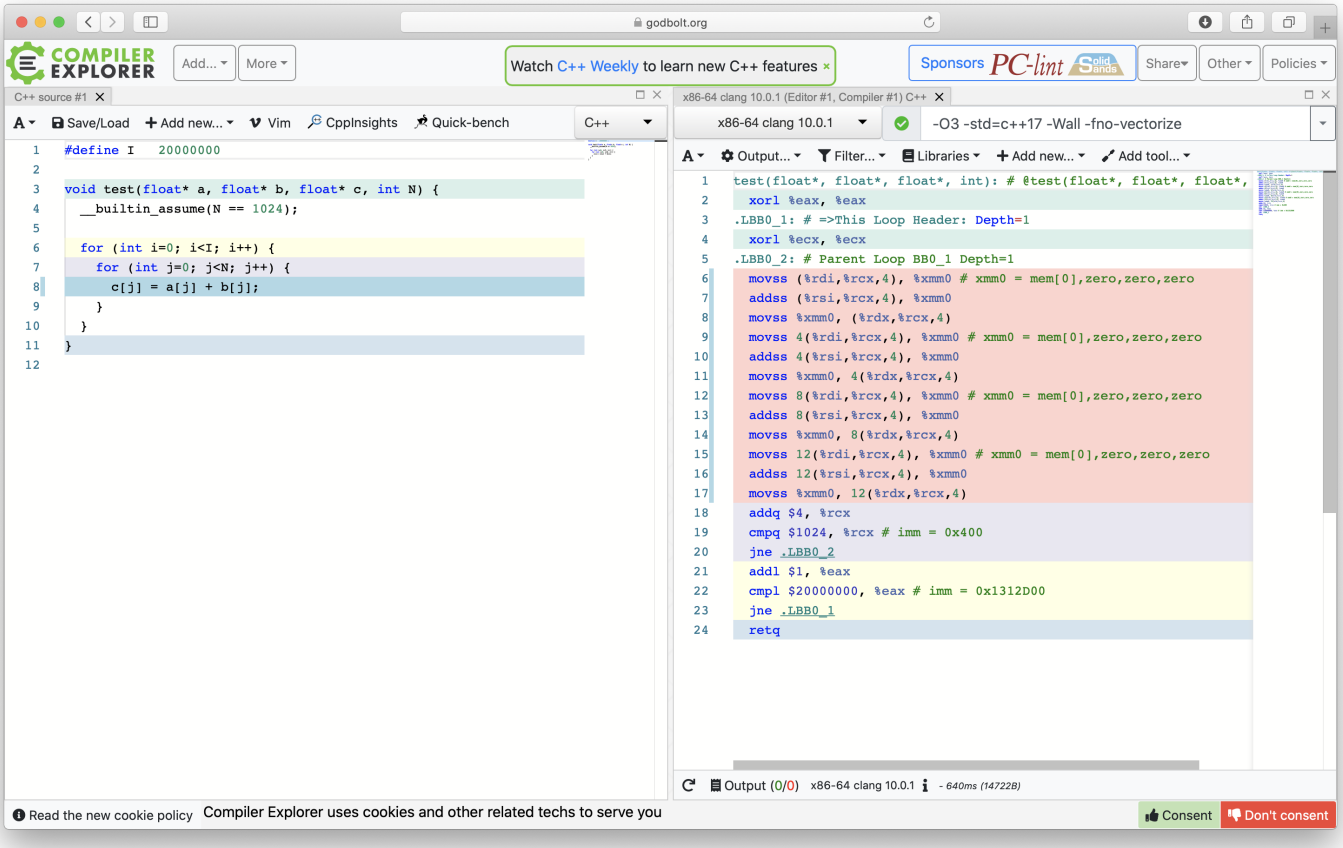
We have added an outer loop over `I` whose purpose is to eliminate measurement error in `gettime()`. Notice that `__builtin_assume(N == 1024)` tells the compiler more about the inputs of the program—say this program is used in a mobile phone and always has the same input size—so that it can perform more optimizations.

You can compile this C++ code fragment with the following command and see the generated assembly code in `assembly/test1.novec.s`.

```
$ make clean; make test1.o ASSEMBLE=1
```

You are recommended to try out Compiler Explorer, a nifty online tool that provides an "interactive compiler". This link is pre-configured for 10.0.1 version of *clang* and compiler flags from the makefile. (To manually configure yourself: select language C++, compiler version x86-64 *clang* 10.0.1 and enter flags `-O3 -`

`std=c++17 -Wall -fno-vectorize` . A screenshot is shown below.



2.1 Turning on auto-vectorization

Let's turn the compiler optimizations on and see how much the compiler can speed up the execution of the program.

We remove `-fno-vectorize` from the compiler option to turn on the compiler optimizations, and add `-Rpass=loop-vectorize -Rpass-missed=loop-vectorize -Rpass-analysis=loop-vectorize` to get more information from *clang* about why it does or does not optimize code. This was done in the makefile, and you can enable auto-vectorization by typing the following command, which generates `assembly/test1.vec.s` .

```
$ make clean; make test1.o ASSEMBLE=1 VECTORIZE=1
```

You should see the following output, informing you that the loop has been vectorized. Although *clang* does tell you this, you should always look at the assembly to see exactly how it has been vectorized, since it is not guaranteed to be using the vector registers optimally.

```
test1.cpp:14:5: remark: vectorized loop (vectorization width: 4, interleaved count: 2) [-Rpass=loop-vectorize]
    for (int j=0; j<N; j++) {
    ^
```

You can observe the difference between `test1.vec.s` and `test1.novec.s` with the following command or by changing the compiler flag on Compiler Explorer.

```
$ diff assembly/test1.vec.s assembly/test1.novec.s
```

2.2 Adding the `__restrict` qualifier

Now, if you inspect the assembly code—actually, you don't need to do that, which is out of the scope of this assignment—you will see the code first checks if there is a partial overlap between arrays `a` and `c` or arrays `b` and `c` . If there is an overlap, then it does a simple non-vectorized code. If there is no overlap, it does a vectorized version. The above can, at best, be called partially vectorized.

The problem is that the compiler is constrained by what we tell it about the arrays. If we tell it more, then perhaps it can do more optimization. The most obvious thing is to inform the compiler that no overlap is possible. This is done in standard C by using the `restrict` qualifier for the pointers. By adding this type qualifier, you can hint to the compiler that for the lifetime of the pointer, only the pointer itself or a value directly derived from it (such as `pointer + 1`) will be used to access the object to which it points.

C++ does not have standard support for `restrict` , but many compilers have equivalents that usually work in both C++ and C, such as the *GCC*'s and *clang*'s `__restrict__` (or `__restrict`), and Visual C++'s `__declspec(restrict)` .

The code after adding the `__restrict` qualifier is shown as follows.

```
void test(float* __restrict a, float* __restrict b, float* __restrict c, int N) {
  __builtin_assume(N == 1024);

  for (int i=0; i<I; i++) {
    for (int j=0; j<N; j++) {
      c[j] = a[j] + b[j];
    }
  }
}
```

Let's modify `test1.cpp` accordingly and recompile it again with the following command, which generates `assembly/test1.vec.restr.s`.

```
$ make clean; make test1.o ASSEMBLE=1 VECTORIZE=1 RESTRICT=1
```

Now you should see the generated code is better—the code for checking possible overlap is gone—but it is assuming the data are **NOT** 16 bytes aligned (`movups` is unaligned move). It also means that the loop above can not assume that the arrays are aligned.

If *clang* were smart, it could test for the cases where the arrays are either all aligned, or all unaligned, and have a fast inner loop. However, it is unable to do that currently.😟

2.3 Adding the `__builtin_assume_aligned` intrinsic

In order to get the performance we are looking for, we need to tell *clang* that the arrays are aligned. There are a couple of ways to do that. The first is to construct a (non-portable) aligned type, and use that in the function interface. The second is to add an intrinsic or three within the function itself. The second option is easier to implement on older code bases, as other functions calling the one to be vectorized do not have to be modified. The intrinsic has for this is called `__builtin_assume_aligned`:

```
void test(float* __restrict a, float* __restrict b, float* __restrict c, int N) {
  __builtin_assume(N == 1024);
  a = (float *)__builtin_assume_aligned(a, 16);
  b = (float *)__builtin_assume_aligned(b, 16);
  c = (float *)__builtin_assume_aligned(c, 16);

  for (int i=0; i<I; i++) {
    for (int j=0; j<N; j++) {
      c[j] = a[j] + b[j];
    }
  }
}
```

Let's modify `test1.cpp` accordingly and recompile it again with the following command, which generates `assembly/test1.vec.restr.align.s`.

```
$ make clean; make test1.o ASSEMBLE=1 VECTORIZE=1 RESTRICT=1 ALIGN=1
```

Let's see the difference:

```
$ diff assembly/test1.vec.restr.s assembly/test1.vec.restr.align.s
```

Now finally, we get the nice tight vectorized code (`movaps` is aligned move.) we were looking for, because *clang* has used packed *SSE* instructions to add 16 bytes at a time. It also manages `load` and `store` two at a time, which it did not do last time. The question is now that we understand what we need to tell the compiler, how much more complex can the loop be before auto-vectorization fails.

2.4 Turning on AVX2 instructions

Next, we try to turn on AVX2 instructions using the following command, which generates `assembly/test1.vec.restr.align.avx2.s`

```
$ make clean; make test1.o ASSEMBLE=1 VECTORIZE=1 RESTRICT=1 ALIGN=1 AVX2=1
```

Let's see the difference:

```
$ diff assembly/test1.vec.restr.align.s assembly/test1.vec.restr.align.avx2.s
```

We can see instructions with prefix `v*`. That's good. We confirm the compiler uses AVX2 instructions; however, this code is still not aligned when using *AVX2* registers.

> ***Q2-1:*** Fix the code to make sure it uses aligned moves for the best performance.
>
> Hint: we want to see `vmovaps` rather than `vmovups`.

2.5 Performance impacts of vectorization

Let's see what speedup we get from vectorization. Build and run the program with the following configurations, which run `test1()` many times, and record the elapsed execution time.

```
# case 1
$ make clean && make && ./test_auto_vectorize -t 1
# case 2
$ make clean && make VECTORIZE=1 && ./test_auto_vectorize -t 1
# case 3
$ make clean && make VECTORIZE=1 AVX2=1 && ./test_auto_vectorize -t 1
```

Note that you may wish to use the workstations provided by this course, which support *AVX2*; otherwise, you may get a message like "Illegal instruction (core dumped)". You can check whether or not a machine supports the *AVX2* instructions by looking for `avx2` in the flags section of the output of `cat /proc/cpuinfo`.

```
$ cat /proc/cpuinfo | grep avx2
```

> ***Q2-2:*** What speedup does the vectorized code achieve over the unvectorized code? What additional speedup does using `-mavx2` give (`AVX2=1` in the `Makefile`)? You may wish to run this experiment several times and take median elapsed times; you can report answers to the nearest 100% (e.g., 2×, 3×, etc). What can you infer about the bit width of the default vector registers on the PP machines? What about the bit width of the *AVX2* vector registers.
>
> Hint: Aside from speedup and the vectorization report, the most relevant information is that the data type for each array is `float`.

You may also run `test2()` and `test3()` with `./test_auto_vectorize -t 2` and `./test_auto_vectorize -t 2`, respectively, before and after fixing the vectorization issues in Section 2.6.

2.6 More examples

2.6.1 Example 2

Take a look at the second example below in `test2.cpp`:

```
void test2(float *__restrict a, float *__restrict b, float *__restrict c, int N)
{
  __builtin_assume(N == 1024);
  a = (float *)__builtin_assume_aligned(a, 16);
  b = (float *)__builtin_assume_aligned(b, 16);
  c = (float *)__builtin_assume_aligned(c, 16);

  for (int i = 0; i < I; i++)
  {
    for (int j = 0; j < N; j++)
    {
      /* max() */
      c[j] = a[j];
      if (b[j] > a[j])
        c[j] = b[j];
    }
  }
}
```

Compile the code with the following command:

```
make clean; make test2.o ASSEMBLE=1 VECTORIZE=1
```

Note that the assembly was not vectorized. Now, change the function with a patch file (`test2.cpp.patch`), which is shown below, by running `patch -i ./test2.cpp.patch`.

```
--- test2.cpp
+++ test2.cpp
@@ -14,9 +14,8 @@
     for (int j = 0; j < N; j++)
     {
       /* max() */
-      c[j] = a[j];
-      if (b[j] > a[j])
-        c[j] = b[j];
+      if (b[j] > a[j]) c[j] = b[j];
+      else c[j] = a[j];
     }
   }
```

Now, you actually see the vectorized assembly with the `movaps` and `maxps` instructions.

> ***Q2-3:*** Provide a theory for why the compiler is generating dramatically different assembly.

2.6.2 Example 3

Take a look at the third example below in `test3.cpp`:

```
double test3(double* __restrict a, int N) {
  __builtin_assume(N == 1024);
  a = (double *)__builtin_assume_aligned(a, 16);

  double b = 0;

  for (int i=0; i<I; i++) {
    for (int j=0; j<N; j++) {
      b += a[j];
    }
  }

  return b;
}
```

Compile the code with the following command:

```
$ make clean; make test3.o ASSEMBLE=1 VECTORIZE=1
```

You should see the non-vectorized code with the `addsd` instructions.

Notice that this does not actually vectorize as the *xmm* registers are operating on 8 byte chunks. The problem here is that *clang* is not allowed to re-order the operations we give it. Even though the the addition operation is associative with real numbers, they are not with floating point numbers. (Consider what happens with signed zeros, for example.)

Furthermore, we need to tell *clang* that reordering operations is okay with us. To do this, we need to add another compile-time flag, `-ffast-math`. Compile the program again with the following command:

```
$ make clean; make test3.o ASSEMBLE=1 VECTORIZE=1 FASTMATH=1
```

You should see the vectorized code with the `addpd` instructions.

3. Requirements

You will need to meet the following requirements and answer the questions (marked with "**Q1** & **Q2**") in a **REPORT** using HackMD. (Markdown is a common format that is widely used for developer documentation (e.g., GitHub), and HackMD is a markdown service which is free, powerful, and the most importantly made in Taiwan. To learn Markdown, you may refer to the video listed in the references.)

3.1 Part 1

1. Implement a vectorized version of `clampedExpSerial` in `clampedExpVector` (using fake vector intrinsics). Your implementation should work with any combination of input array size (`N`) and vector width (`VECTOR_WIDTH`), achieve a **vector utilization higher than 60%**, and of course **pass the verification**. (You can assume the array size is much bigger than the vector width.)
2. Run `./myexp -s 10000` and sweep the vector width from 2, 4, 8, to 16. Record the resulting vector utilization. You can do this by changing the `#define VECTOR_WIDTH` value in `def.h`. **Q1-1**: Does the vector utilization increase, decrease or stay the same as `VECTOR_WIDTH` changes? Why?
3. **Bonus**: Implement a vectorized version of `arraySumSerial` in `arraySumVector`. Your implementation may assume that `VECTOR_WIDTH` is an even number and also a factor of the input array size `N`. Whereas the serial implementation has `O(N)` work-span, your implementation should have at most `O(N / VECTOR_WIDTH + log2(VECTOR_WIDTH))` span. You should achieve a vector utilization higher than 80% and pass the verification. You may find the `hadd` and `interleave` operations useful. (You can assume the array size is much bigger than the vector width.)

3.2 Part 2

Answer the three questions (Q2-1, Q2-2, and Q2-3) embedded in part 2. We don't test your code. If you have code for answering the questions, show the code and explain it thoroughly in your report.

4. Grading Policy

NO CHEATING!! You will receive no credit if you are found cheating.

Total of 110%:

- Part 1 (80%):
  - Correctness (60%): A correct implementation of `clampedExpVector`. The requirements should be met. Notice that you will receive **no credit** if any of the requirements fails.
  - Question (10%): `Q1-1` contributes 10%. Answer to the question will be classified into one of the four reward tiers: excellent (10%), good (7%), normal (3%), and terrible (0%).

    - Bonus (10%): A correct implementation of `arraySumVector`. The requirements should be met. Notice that you will receive **no credit** if any of the requirements fails.
- Part 2 (30%)
    - Questions: For `Q2-1` ~ `Q2-3`, each question contributes 10%. Answers to each question will be classified into one of the four reward tiers: excellent (10%), good (7%), normal (3%), and terrible (0%).

5. Submission

All your files should be organized in the following hierarchy and zipped into a `.zip` file, named `HW1_xxxxxxx.zip`, where `xxxxxxx` is your student ID.

Directory structure inside the zipped file:

- `HW1_xxxxxxx.zip` (root)
    - `vectorOP.cpp`
    - `url.txt`

Notice that you just need to provide the URL of your HackMD report in `url.txt`, and enable the write permission for someone who knows the URL so that TAs can give you feedback directly in your report.

You can use the testing script `test_hw1` to check your answer *for reference only*. Run `test_hw1` in a directory that contains your `HW1_XXXXXXX.zip` file on the workstation. `test_hw1` checks if the zip file is correct, and runs graders.

```
$ test_hw1
```

Be sure to upload your zipped file to new E3 e-Campus system by the due date.

You will get *NO POINT* if your ZIP's name is wrong or the ZIP hierarchy is incorrect.

6. References

- Wikipedia: Analysis of parallel algorithms
- Wikipedia: SIMD
- Clang: built-in functions document
- Video: Markdown 使用教學
- Video: SSH & SCP 使用教學