Performance Engineering of Software Systems
Massachusetts Institute of Technology 6.172
Prof. Charles E. Leiserson and Prof. Julian Shun Handout 5

Homework 3: Vectorization

In this homework and recitation you will experiment with Intel Vector Extensions. You will learn how to vectorize your code, figure out when vectorization has succeeded and debug when vectorization seems to have worked but you aren't seeing speedup.

Vectorization is a general optimization technique that can buy you an order of magnitude performance increase in some cases. It is also a delicate operation. On the one hand, vectorization is automatic: when `clang` *is told to optimize aggressively, it will automatically try to vectorize every loop in your program. On the other hand, very small changes to loop structure cause* `clang` *to give up and not vectorize at all. Furthermore, these small changes may allow your code to vectorize but not yield the expected speedup. We will discuss how to identify these cases so that you can get the most out of your vector units.*

1 Getting started

[Note: This assignment makes use of AWS and/or Git features which may not be available to OCW users.]

Submitting your solutions

For each question we ask (i.e., each sentence with a question mark), respond with a short (1-3 sentence) responses or a code snippet (if requested). **Please ensure that all the times you quote are obtained from the `awsrun` machines.**

2 Vectorization in `clang`

Consider a loop that performs elementwise addition between two arrays `A` and `B`, storing the result in array `C`. This loop is *data parallel* because the operation during any iteration i_1 is independent of the operation during any iteration i_2 where $i_1 \neq i_2$. In short, the compiler should be allowed to schedule each iteration in any order, or pack multiple iterations into a single clock cycle. The first option will be covered in the next homework. The second case is covered by vectorization, also known as "single instruction, multiple data" or SIMD.

Vectorization is a delicate operation: very small changes to loop structure may cause `clang` to give up and not vectorize at all, or to vectorize your code but not yield the expected speedup. Occasionally, unvectorized code may be faster than vectorized code. Before we can understand this fragility, we must get a handle on how to interpret what `clang` is actually doing when it vectorizes code; in Section 3, you will see the actual performance impacts of vectorizing code.

2.1 Example 1

We will start with the following simple loop:

```
#include <stdint.h>
#include <stdlib.h>
#include <math.h>

#define SIZE (1L << 16)

void test(uint8_t * a,  uint8_t * b) {
  uint64_t i;

  for (i = 0; i < SIZE; i++) {
    a[i] += b[i];
  }
}
```

```
$ make clean; make ASSEMBLE=1 VECTORIZE=1 example1.o
```

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.

```
example1.c:12:3: remark: vectorized loop (vectorization width: 16, interleaved count: 2)
      [-Rpass=loop-vectorize]
  for (i = 0; i < SIZE; i++) {
```

Now, let's inspect the assembly code in `example1.s`. You should see something similar to the following:

```
# %bb.0:                                # %entry
        #DEBUG_VALUE: test:a <- %rdi
        #DEBUG_VALUE: test:a <- %rdi
        #DEBUG_VALUE: test:b <- %rsi
        #DEBUG_VALUE: test:b <- %rsi
        #DEBUG_VALUE: test:i <- 0
        .loc    1 12 3 prologue_end     # example1.c:12:3

        leaq    65536(%rsi), %rax
        cmpq    %rdi, %rax
        jbe    .LBB0_2
# %bb.1:                                # %entry
        #DEBUG_VALUE: test:b <- %rsi
        #DEBUG_VALUE: test:a <- %rdi
        leaq    65536(%rdi), %rax
        cmpq    %rsi, %rax
        jbe    .LBB0_2
# %bb.4:                                # %for.body.preheader
        #DEBUG_VALUE: test:b <- %rsi
        #DEBUG_VALUE: test:a <- %rdi
        .loc    1 0 3 is_stmt 0         # example1.c:0:3

        movq    $-65536, %rax           # imm = 0xFFFF0000

        .p2align        4, 0x90
.LBB0_5:                                # %for.body
                                        # =>This Inner Loop Header: Depth=1
        #DEBUG_VALUE: test:b <- %rsi
        #DEBUG_VALUE: test:a <- %rdi
.Ltmp0:
        .loc    1 13 13 is_stmt 1       # example1.c:13:13

        movzbl    65536(%rsi,%rax), %ecx
        .loc    1 13 10 is_stmt 0       # example1.c:13:10

        addb    %cl, 65536(%rdi,%rax)
        .loc    1 13 13                 # example1.c:13:13

        movzbl    65537(%rsi,%rax), %ecx
        .loc    1 13 10                 # example1.c:13:10

        addb    %cl, 65537(%rdi,%rax)
        .loc    1 13 13                 # example1.c:13:13

        movzbl    65538(%rsi,%rax), %ecx
        .loc    1 13 10                 # example1.c:13:10

        addb    %cl, 65538(%rdi,%rax)
        .loc    1 13 13                 # example1.c:13:13

        movzbl    65539(%rsi,%rax), %ecx
        .loc    1 13 10                 # example1.c:13:10

        addb    %cl, 65539(%rdi,%rax)
```

```
.Ltmp1:
    .loc    1 12 17 is_stmt 1       # example1.c:12:17

    addq    $4, %rax
.Ltmp2:
    .loc    1 12 3 is_stmt 0        # example1.c:12:3

    jne .LBB0_5
    jmp .LBB0_6
.LBB0_2:                                # %vector.body.preheader
    #DEBUG_VALUE: test:b <- %rsi
    #DEBUG_VALUE: test:a <- %rdi
    .loc    1 0 3                   # example1.c:0:3

    movq    $-65536, %rax           # imm = 0xFFFF0000
    .p2align    4, 0x90
.LBB0_3:                                # %vector.body
                                        # =>This Inner Loop Header: Depth=1
    #DEBUG_VALUE: test:b <- %rsi
    #DEBUG_VALUE: test:a <- %rdi
.Ltmp3:
    .loc    1 13 13 is_stmt 1       # example1.c:13:13

    movdqu  65536(%rsi,%rax), %xmm0
    movdqu  65552(%rsi,%rax), %xmm1
    .loc    1 13 10 is_stmt 0       # example1.c:13:10

    movdqu  65536(%rdi,%rax), %xmm2
    paddb   %xmm0, %xmm2
    movdqu  65552(%rdi,%rax), %xmm0
    movdqu  65568(%rdi,%rax), %xmm3
    movdqu  65584(%rdi,%rax), %xmm4
    movdqu  %xmm2, 65536(%rdi,%rax)
    paddb   %xmm1, %xmm0
    movdqu  %xmm0, 65552(%rdi,%rax)
    .loc    1 13 13                 # example1.c:13:13

    movdqu  65568(%rsi,%rax), %xmm0
    .loc    1 13 10                 # example1.c:13:10

    paddb   %xmm3, %xmm0
    .loc    1 13 13                 # example1.c:13:13

    movdqu  65584(%rsi,%rax), %xmm1
    .loc    1 13 10                 # example1.c:13:10

    movdqu  %xmm0, 65568(%rdi,%rax)
    paddb   %xmm4, %xmm1
    movdqu  %xmm1, 65584(%rdi,%rax)
.Ltmp4:
    .loc    1 12 26 is_stmt 1       # example1.c:12:26

    addq    $64, %rax
    jne .LBB0_3
```

Write-up 1: Look at the assembly code above. The compiler has translated the code to set the start index at -2^{16} and adds to it for each memory access. Why doesn't it set the start index to 0 and use small positive offsets?

This code first checks if there is a partial overlap between array `a` and `b`. If there is an overlap, then it does a simple non-vectorized code. If there is overlap, then go to `.LBB0_2`, and do 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.

```
void test(uint8_t * restrict a,  uint8_t * restrict b) {
  uint64_t i;

  for (i = 0; i < SIZE; i++) {
    a[i] += b[i];
  }
}
```

Now you should see the following assembly code:

```
# %bb.0:                                # %entry
        #DEBUG_VALUE: test:a <- %rdi
        #DEBUG_VALUE: test:a <- %rdi
        #DEBUG_VALUE: test:b <- %rsi
        #DEBUG_VALUE: test:b <- %rsi
        movq        $-65536, %rax           # imm = 0xFFFF0000
.Ltmp0:
        #DEBUG_VALUE: test:i <- 0
        .p2align        4, 0x90
.LBB0_1:                                # %vector.body
                                        # =>This Inner Loop Header: Depth=1
        #DEBUG_VALUE: test:b <- %rsi
        #DEBUG_VALUE: test:a <- %rdi
        .loc        1 13 13 prologue_end    # example1.c:13:13

        movdqu        65536(%rsi,%rax), %xmm0
        .loc        1 13 10 is_stmt 0       # example1.c:13:10

        movdqu        65536(%rdi,%rax), %xmm1
        paddb        %xmm0, %xmm1
        movdqu        65552(%rdi,%rax), %xmm0
        movdqu        65568(%rdi,%rax), %xmm2
        movdqu        65584(%rdi,%rax), %xmm3
        movdqu        %xmm1, 65536(%rdi,%rax)
        .loc        1 13 13                 # example1.c:13:13

        movdqu        65552(%rsi,%rax), %xmm1
        .loc        1 13 10                 # example1.c:13:10

        paddb        %xmm1, %xmm0
        movdqu        %xmm0, 65552(%rdi,%rax)
        .loc        1 13 13                 # example1.c:13:13

        movdqu        65568(%rsi,%rax), %xmm0
        .loc        1 13 10                 # example1.c:13:10

        paddb        %xmm2, %xmm0
        .loc        1 13 13                 # example1.c:13:13

        movdqu        65584(%rsi,%rax), %xmm1
        .loc        1 13 10                 # example1.c:13:10

        movdqu        %xmm0, 65568(%rdi,%rax)
        paddb        %xmm3, %xmm1
        movdqu        %xmm1, 65584(%rdi,%rax)
.Ltmp1:
        .loc        1 12 26 is_stmt 1       # example1.c:12:26

        addq        $64, %rax
         jne        .LBB0_1
```

The generated code is better, but it is assuming the data are NOT 16 bytes aligned (`movdqu` is unaligned move). It also means that the loop above can not assume that both arrays are aligned. If `clang` were smart, it could test for the cases where the arrays are either both aligned, or both unaligned, and have a fast inner loop. However, it does not do that currently.

So 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 two 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(uint8_t * restrict a,  uint8_t * restrict b) {
  uint64_t i;

  a = __builtin_assume_aligned(a, 16);
  b = __builtin_assume_aligned(b, 16);

  for (i = 0; i < SIZE; i++) {
    a[i] += b[i];
  }
}
```

After you add the instruction `__builtin_assume_aligned`, you should see something similar to the following output:

```
# %bb.0:                                # %entry
        #DEBUG_VALUE: test:a <- %rdi
        #DEBUG_VALUE: test:a <- %rdi
        #DEBUG_VALUE: test:b <- %rsi
        #DEBUG_VALUE: test:b <- %rsi
        movq      $-65536, %rax         # imm = 0xFFFF0000
.Ltmp0:
        #DEBUG_VALUE: test:i <- 0
        .p2align        4, 0x90
.LBB0_1:                                # %vector.body
                                        # =>This Inner Loop Header: Depth=1
        #DEBUG_VALUE: test:b <- %rsi
        #DEBUG_VALUE: test:a <- %rdi
        .loc      1 16 10 prologue_end    # example1.c:16:10

        movdqa      65536(%rdi,%rax), %xmm0
        movdqa      65552(%rdi,%rax), %xmm1
        movdqa      65568(%rdi,%rax), %xmm2
        movdqa      65584(%rdi,%rax), %xmm3
        paddb       65536(%rsi,%rax), %xmm0
        paddb       65552(%rsi,%rax), %xmm1
        movdqa      %xmm0, 65536(%rdi,%rax)
        movdqa      %xmm1, 65552(%rdi,%rax)
        paddb       65568(%rsi,%rax), %xmm2
        paddb       65584(%rsi,%rax), %xmm3
        movdqa      %xmm2, 65568(%rdi,%rax)
        movdqa      %xmm3, 65584(%rdi,%rax)
.Ltmp1:
        .loc      1 15 26                 # example1.c:15:26

        addq      $64, %rax
        jne      .LBB0_1
.Ltmp2:
```

Now finally, we get the nice tight vectorized code (`movdqa` is aligned move) we were looking for, because `clang` has used packed SSE instructions to add 16 bytes at a time. It also manages to 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.

Next, we try to turn on AVX2 instructions using the following command:

```
$ make clean; make ASSEMBLE=1 VECTORIZE=1 AVX2=1 example1.o
```

```
# %bb.0:                                # %entry
        #DEBUG_VALUE: test:a <- %rdi
        #DEBUG_VALUE: test:a <- %rdi
        #DEBUG_VALUE: test:b <- %rsi
        #DEBUG_VALUE: test:b <- %rsi
        movq        $-65536, %rax           # imm = 0xFFFF0000
.Ltmp0:
        #DEBUG_VALUE: test:i <- 0
        .p2align        4, 0x90
.LBB0_1:                                # %vector.body
                                        # =>This Inner Loop Header: Depth=1
        #DEBUG_VALUE: test:b <- %rsi
        #DEBUG_VALUE: test:a <- %rdi
        .loc        1 16 10 prologue_end    # example1.c:16:10

        vmovdqu        65536(%rdi,%rax), %ymm0
        vmovdqu        65568(%rdi,%rax), %ymm1
        vmovdqu        65600(%rdi,%rax), %ymm2
        vmovdqu        65632(%rdi,%rax), %ymm3
        vpaddb         65536(%rsi,%rax), %ymm0, %ymm0
        vpaddb         65568(%rsi,%rax), %ymm1, %ymm1
        vpaddb         65600(%rsi,%rax), %ymm2, %ymm2
        vmovdqu        %ymm0, 65536(%rdi,%rax)
        vmovdqu        %ymm1, 65568(%rdi,%rax)
        vmovdqu        %ymm2, 65600(%rdi,%rax)
        vpaddb         65632(%rsi,%rax), %ymm3, %ymm0
        vmovdqu        %ymm0, 65632(%rdi,%rax)
.Ltmp1:
        .loc        1 15 26                 # example1.c:15:26

        addq        $128, %rax
        jne        .LBB0_1
.Ltmp2:
```

Write-up 2: This code is still not aligned when using AVX2 registers. Fix the code to make sure it uses aligned moves for the best performance.

2.2 Example 2

Take a look at the second example below in `example2.c`:

```c
void test(uint8_t * restrict a, uint8_t * restrict b) {
  uint64_t i;

  uint8_t * x = __builtin_assume_aligned(a, 16);
  uint8_t * y = __builtin_assume_aligned(b, 16);

  for (i = 0; i < SIZE; i++) {
    /* max() */
    if (y[i] > x[i]) x[i] = y[i];
  }
}
```

Compile example 2 with the following command:

```
$ make clean; make ASSEMBLE=1 VECTORIZE=1 example2.o
```

Note that the assembly does not vectorize nicely. Now, change the function to look like the following:

```c
void test(uint8_t * restrict a, uint8_t * restrict b) {
  uint64_t i;

  a = __builtin_assume_aligned(a, 16);
  b = __builtin_assume_aligned(b, 16);

  for (i = 0; i < SIZE; i++) {
    /* max() */
    a[i] = (b[i] > a[i]) ? b[i] : a[i];
  }
}
```

Now, you actually see the vectorized assembly with the `movdqa` and `pmaxub` instructions.

```
# %bb.0:                                # %entry
        #DEBUG_VALUE: test:a <- %rdi
        #DEBUG_VALUE: test:a <- %rdi
        #DEBUG_VALUE: test:b <- %rsi
        #DEBUG_VALUE: test:b <- %rsi
        movq        $-65536, %rax           # imm = 0xFFFF0000
.Ltmp0:
        #DEBUG_VALUE: test:i <- 0
        .p2align        4, 0x90
.LBB0_1:                                # %vector.body
                                        # =>This Inner Loop Header: Depth=1
        #DEBUG_VALUE: test:b <- %rsi
        #DEBUG_VALUE: test:a <- %rdi
        .loc        1 17 15 prologue_end    # example2.c:17:15

        movdqa        65536(%rsi,%rax), %xmm0
        movdqa        65552(%rsi,%rax), %xmm1
        .loc        1 17 14 is_stmt 0       # example2.c:17:14

        pmaxub        65536(%rdi,%rax), %xmm0
        pmaxub        65552(%rdi,%rax), %xmm1
        .loc        1 17 12                 # example2.c:17:12

        movdqa        %xmm0, 65536(%rdi,%rax)
        movdqa        %xmm1, 65552(%rdi,%rax)
        .loc        1 17 15                 # example2.c:17:15

        movdqa        65568(%rsi,%rax), %xmm0
        movdqa        65584(%rsi,%rax), %xmm1
        .loc        1 17 14                 # example2.c:17:14

        pmaxub        65568(%rdi,%rax), %xmm0
        pmaxub        65584(%rdi,%rax), %xmm1
        .loc        1 17 12                 # example2.c:17:12

        movdqa        %xmm0, 65568(%rdi,%rax)
        movdqa        %xmm1, 65584(%rdi,%rax)
.Ltmp1:
        .loc        1 15 28 is_stmt 1       # example2.c:15:28

        addq        $64, %rax
        jne        .LBB0_1
.Ltmp2:
```

Write-up 3: Provide a theory for why the compiler is generating dramatically different assembly.

2.3 Example 3

Open up `example3.c` and run the following command:

```
$ make clean; make ASSEMBLE=1 VECTORIZE=1 example3.o
```

```c
void test(uint8_t * restrict a, uint8_t * restrict b) {
  uint64_t i;

  for (i = 0; i < SIZE; i++) {
    a[i] = b[i + 1];
  }
}
```

Write-up 4: Inspect the assembly and determine why the assembly does not include instructions with vector registers. Do you think it would be faster if it did vectorize? Explain.

2.4 Example 4

Take a look at `example4.c`.

```c
double test(double * restrict a) {
  size_t i;

  double *x = __builtin_assume_aligned(a, 16);

  double y = 0;

  for (i = 0; i < SIZE; i++) {
    y += x[i];
  }
  return y;
}
```

```
$ make clean; make ASSEMBLE=1 VECTORIZE=1 example4.o
```

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

```
.LBB0_1:                                # %for.body
                                        # =>This Inner Loop Header: Depth=1
        #DEBUG_VALUE: test:x <- %rdi
        #DEBUG_VALUE: test:a <- %rdi
.Ltmp1:
        #DEBUG_VALUE: test:y <- %xmm0
        .loc        1 18 7 prologue_end     # example4.c:18:7

        addsd       524288(%rdi,%rax,8), %xmm0
.Ltmp2:
        #DEBUG_VALUE: test:y <- %xmm0
        addsd       524296(%rdi,%rax,8), %xmm0
.Ltmp3:
        #DEBUG_VALUE: test:y <- %xmm0
        addsd       524304(%rdi,%rax,8), %xmm0
.Ltmp4:
        #DEBUG_VALUE: test:y <- %xmm0
        addsd       524312(%rdi,%rax,8), %xmm0
.Ltmp5:
        #DEBUG_VALUE: test:y <- %xmm0
        addsd       524320(%rdi,%rax,8), %xmm0
.Ltmp6:
        #DEBUG_VALUE: test:y <- %xmm0
        addsd       524328(%rdi,%rax,8), %xmm0
.Ltmp7:
        #DEBUG_VALUE: test:y <- %xmm0
        addsd       524336(%rdi,%rax,8), %xmm0
.Ltmp8:
        #DEBUG_VALUE: test:y <- %xmm0
        addsd       524344(%rdi,%rax,8), %xmm0
.Ltmp9:
        #DEBUG_VALUE: test:y <- %xmm0
        .loc        1 17 17                 # example4.c:17:17

        addq        $8, %rax
.Ltmp10:
        .loc        1 17 3 is_stmt 0        # example4.c:17:3

        jne        .LBB0_1
```

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`. Add the compilation flag `-ffast-math` to the Makefile and compile the program again.

Write-up 5: Check the assembly and verify that it does in fact vectorize properly. Also what do you notice when you run the command

```
$ clang -O3 example4.c -o example4; ./example4
```

with and without the `-ffast-math` flag? Specifically, why do you a see a difference in the output.

```
# %bb.0:                                # %entry
        #DEBUG_VALUE: test:a <- %rdi
        #DEBUG_VALUE: test:a <- %rdi
        #DEBUG_VALUE: test:x <- %rdi
        #DEBUG_VALUE: test:x <- %rdi
        xorpd       %xmm0, %xmm0
.Ltmp0:
        #DEBUG_VALUE: test:i <- 0
        #DEBUG_VALUE: test:y <- 0.000000e+00
        movq        $-65536, %rax           # imm = 0xFFFF0000
        xorpd       %xmm1, %xmm1
        .p2align        4, 0x90
.LBB0_1:                                # %vector.body
                                        # =>This Inner Loop Header: Depth=1
        #DEBUG_VALUE: test:x <- %rdi
        #DEBUG_VALUE: test:a <- %rdi
.Ltmp1:
        .loc        1 18 7 prologue_end     # example4.c:18:7

        addpd       524288(%rdi,%rax,8), %xmm0
        addpd       524304(%rdi,%rax,8), %xmm1
        addpd       524320(%rdi,%rax,8), %xmm0
        addpd       524336(%rdi,%rax,8), %xmm1
        addpd       524352(%rdi,%rax,8), %xmm0
        addpd       524368(%rdi,%rax,8), %xmm1
        addpd       524384(%rdi,%rax,8), %xmm0
        addpd       524400(%rdi,%rax,8), %xmm1
.Ltmp2:
        .loc        1 17 26                 # example4.c:17:26

        addq        $16, %rax
        jne         .LBB0_1
# %bb.2:                                # %middle.block
        #DEBUG_VALUE: test:x <- %rdi
        #DEBUG_VALUE: test:a <- %rdi
.Ltmp3:
        .loc        1 18 7                  # example4.c:18:7

        addpd       %xmm0, %xmm1
        movapd      %xmm1, %xmm0
        movhlps     %xmm0, %xmm0             # xmm0 = xmm0[1,1]

        addpd       %xmm1, %xmm0
```

3 Performance Impacts of Vectorization

We will now familiarize ourselves with what code does/does not vectorize, and discuss how to increase speedup from vectorization.

3.1 The Many Facets of a Data Parallel Loop

In `loop.c`, we have written a loop that performs elementwise an operation — by default, addition — between two arrays `A` and `B`, storing the result in array `C`. If you examine the code, you will see that our loop does no useful work (in the sense that `A` and `B` are not filled with any initial values). We are just using this loop to demonstrate concepts. Further, we have added an outer loop over `I` whose purpose is to eliminate measurement error in `gettime()`.

Let's see what speedup we get from vectorization. Run `make` and run `awsrun ./loop`. Record the elapsed execution time. Then run `make VECTORIZE=1` and run `awsrun ./loop` again. Record the vectorized elapsed execution time. The flag `-mavx2` tells `clang` to use advanced vector extensions with larger vector registers. Run `make VECTORIZE=1 AVX2=1` and run `awsrun ./loop` again. **Note that you must use the `awsrun` machines for this; you may otherwise 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`. Record the vectorized elapsed execution time.

> **Write-up 6:** What speedup does the vectorized code achieve over the unvectorized code? What additional speedup does using `-mavx2` give? 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 `awsrun` 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 `uint32_t`.

3.1.1 Flags to enable and debug vectorization

Vectorization is enabled by default, but can be explicitly turned on with the `-fvectorize` flag[1]. When vectorization is enabled, the `-Rpass=loop-vectorize` flag identifies loops that were successfully vectorized, and the `-Rpass-missed=loop-vectorize` flag identifies loops that failed vectorization and indicates if vectorization was specified (see `Makefile`). Further, you can add the flag `-Rpass-analysis=loop-vectorize` to identify the statements that caused vectorization to fail.

3.1.2 Debugging through assembly code inspection

Another way to see how code is vectorized is to look at the assembly output from the compiler. Run

```
$ make ASSEMBLE=1 VECTORIZE=1
```

[1] If you open `Makefile`, you will see we set up things in a slightly different way. We set `-O3` regardless of vectorization—because we want a fair comparison when the vectorization flag is enabled/disabled. We then *disable* vectorization for when `VECTORIZE=0` by setting the flag `-fno-vectorize`.

This will produce `loop.s`, which contains human-readable x86 assembly like `perf annotate -f` from Recitation 2. Note that the compilation may "fail" with `ASSEMBLE=1` because this flag tells `clang` to not produce `loop.o`.

Write-up 7: Compare the contents of `loop.s` when the `VECTORIZE` flag is set/not set. Which instruction (copy its text here) is responsible for the vector add operation? Which instruction (copy its text here) is responsible for the vector add operation when you additionally pass `AVX2=1`? You can find an x86 instruction manual on LMOD. Look for MMX and SSE2 instructions, which are vector operations. To make the assembly code more readable it may be a good idea to remove debug symbols from release builds by moving the `-g` and `-gdwarf-3` CFLAGS in your Makefile. It might also be a good idea to turn off loop unrolling with the `-fno-unroll-loops` flag while you study the assembly code.

3.1.3 Flavors of vector arithmetic

As discussed in lecture, the vector unit is built directly in hardware. To support more flavors of vector operations (e.g., vector subtract or multiply), additional hardware must be added for each operation.

Write-up 8: Use the `__OP__` macro to experiment with different operators in the data parallel loop. For some operations, you will get division by zero errors because we initialize array `B` to be full of zeros—fix this problem in any way you like. Do any versions of the loop not vectorize with `VECTORIZE=1 AVX2=1`? Study the assembly code for << with just `VECTORIZE=1` and explain how it differs from the AVX2 version.

The results may surprise you. For example, compare the results for `*` and << (shift). The problem is that shifting by a variable amount (`B[j]`) is not a supported vector instruction unless we pass `-mavx2`. Changing `B[j]` to a constant value should allow the code to be vectorizable again.

3.1.4 Packing smaller words into vectors

A big class of optimizations you will use in future projects is optimizing data type width for your application. Consider the arrays `A`, `B`, and `C` which have data type `uint32_t` (given by the `__TYPE__` macro). Changing the data type for each array has an impact in two places:

1. Memory requirements. A smaller data type per element leads to a smaller memory footprint per array.

2. Vector packing. A smaller data type allows more elements to be packed into a single vector register.

Let's experiment with the vector packing idea:

> **Write-up 9:** What is the new speedup for the vectorized code, over the unvectorized code, and for the AVX2 vectorized code, over the unvectorized code, when you change `__TYPE__` to `uint64_t`, `uint32_t`, `uint16_t` and `uint8_t`? For each experiment, set `__OP__` to `+` and do not change `N`.

In general, speedup should increase as data type size decreases. This is a fundamental advantage over unvectorized codes where for fixed `N`, the number of instructions needed to perform elementwise operations over an array of `N` elements is *mostly* independent of the data type width.[2]

3.1.5 To vectorize or not to vectorize

Performance potential from vectorization is also impacted by what operation you wish to perform. Of the operations that vectorize (Section 3.1.3), multiply (`*`) takes the most clock cycles per operation.

> **Write-up 10:** You already determined that `uint64_t` yields the least performance improvement for vectorized codes (Section 3.1.4). Test a vector multiplication (i.e., `__OP__` is `*`) using `uint64_t` arrays. What happens to the AVX2 vectorized code's speedup relative to the unvectorized code (also using `uint64_t` and `*`)? What about when you set the data type width to be smaller — say `uint8_t`?

> **Write-up 11:** Open up the `aws-perf-report` tool for the AVX2 vectorized multiply code using `uint64_t` (as you did in Recitation 2). Remember to first use the `awsrun perf record` tool to collect a performance report. Does the vector multiply take the most time? If not, where is time going instead? Now change `__OP__` back to `+`, rerun the experiment and inspect `aws-perf-report` again. How does the percentage of time taken by the AVX2 vector add instruction compare to the time spent on the AVX2 vector multiply instruction?

[2] We say "mostly" because depending on your processor's architecture, arrays with large data types (e.g., 64 bit and 128 bit) are processed in different ways. For example, you *can* use 128 bit data types using gcc and the type `__int128`. But since ALUs in the `awsrun` machines are only 64 bits wide, the compiler turns each 128 bit operation into several 64 bit operations.

You will see that where time goes changes dramatically when you change `*` to `+`. This is partly due to the data type width (`uint64_t`) and partly due to the `*` operation itself. In particular, the awsrun machine vector units only support 32×32 bit multiplication—wider data types are synthesized from smaller operations. If you experiment with smaller (`uint16_t` and below) data types, you should see that the assembly code for `*` and `+` look more similar

3.2 Vector Patterns

We will now explore some common vector code patterns. We also recommend `https://llvm.org/docs/Vectorizers.html` as a reference guide for when you are optimizing your projects.

3.2.1 Loops with Runtime Bounds

Up to this point, our data parallel loop has been simple for the compiler to handle because `N` was known beforehand and was a power of 2. What about when the loop bound is not known ahead of time?

Write-up 12: Get rid of the `#define N 1024` macro and redefine `N` as: `int N = atoi(argv[1]);` (at the beginning of `main()`). (Setting `N` through the command line ensures that the compiler will make no assumptions about it.) Rerun (with various choices of `N`) and compare the AVX2 vectorized, non-AVX2 vectorized, and unvectorized codes. Does the speedup change dramatically relative to the `N = 1024` case? Why?

Hint: If you look at `loop.s` when you apply this change, you will see the compiler adding termination case code to handle the final loop iterations (i.e., the iterations that do not align with the vector register width). Test this yourself: as you set `__TYPE__` to smaller data types, you should see that the amount of termination-related assembly code emitted by the compiler increases.

3.2.2 Striding

Another simplifying feature in our loop is that its ***stride*** (or step) equals 1. Stride corresponds to how big our steps through the array are; e.g., `j++`, `j+=2`, etc. The `awsrun` machine vector units have some hardware support to accelerate different strides.

For example,

```
for (j = 0; j < N; j+=2) {
  C[j] = A[j] + B[j];
}
```

Write-up 13: Set `__TYPE__` to `uint32_t` and `__OP__` to +, and change your inner loop to be strided. Does `clang` vectorize the code? Why might it choose not to vectorize the code?

`clang` provides a `#pragma Clang loop` directive that can be used to control the optimization of loops, including vectorization. These are described at the following webpage: `http://Clang.llvm.org/docs/LanguageExtensions.html#extensions-for-loop-hint-optimizations`

Write-up 14: Use the `#vectorize` pragma described in the `clang` language extensions webpage above to make `clang` vectorize the strided loop. What is the speedup over non-vectorized code for non-AVX2 and AVX2 vectorization? What happens if you change the `vectorize_width` to 2? Play around with the `clang` loop pragmas and report the best you found (that vectorizes the loop). Did you get a speedup over the non-vectorized code?

Once again, inspecting the assembly code to see how striding is vectorized can be insightful.

3.2.3 Strip Mining

A very common operation is to combine elements in an array (somehow) into a single value. For instance, one might wish to sum up the elements in an array. Replace the data parallel inner loop with such a reduction:

```
for (j = 0; j < N; j++) {
  total += A[j];
}
```

To ensure that `clang` vectorizes the inner loop rather than the outer loop, comment out the outer loop.

Write-up 15: This code vectorizes, but how does it vectorize? Turn on `ASSEMBLE=1`, look at the assembly dump, and explain what the compiler is doing.

As discussed in lecture, this reduction will only vectorize if the combination operation (+) is associative.