

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:

```
01 #include <stdint.h>
02 #include <stdlib.h>
03 #include <math.h>
04
05 #define SIZE (1L << 16)
06
07 void test(uint8_t * a, uint8_t * b) {
08     uint64_t i;
09
10     for (i = 0; i < SIZE; i++) {
11         a[i] += b[i];
12     }
13 }
```

```
$ 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.

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

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

```

17 # %bb.0:                                     # %entry
18     #DEBUG_VALUE: test:a <- %rdi
19     #DEBUG_VALUE: test:a <- %rdi
20     #DEBUG_VALUE: test:b <- %rsi
21     #DEBUG_VALUE: test:b <- %rsi
22     #DEBUG_VALUE: test:i <- 0
23     .loc      1 12 3 prologue_end             # example1.c:12:3
24
25     leaq      65536(%rsi), %rax
26     cmpq     %rdi, %rax
27     jbe      .LBB0_2
28 # %bb.1:                                     # %entry
29     #DEBUG_VALUE: test:b <- %rsi
30     #DEBUG_VALUE: test:a <- %rdi
31     leaq      65536(%rdi), %rax
32     cmpq     %rsi, %rax
33     jbe      .LBB0_2
34 # %bb.4:                                     # %for.body.preheader
35     #DEBUG_VALUE: test:b <- %rsi
36     #DEBUG_VALUE: test:a <- %rdi
37     .loc      1 0 3 is_stmt 0                 # example1.c:0:3
38
39     movq     $-65536, %rax                     # imm = 0xFFFF0000
40
41     .p2align  4, 0x90
42 .LBB0_5:                                     # %for.body
43                                     # =>This Inner Loop Header: Depth=1
44     #DEBUG_VALUE: test:b <- %rsi
45     #DEBUG_VALUE: test:a <- %rdi
46 .Ltmp0:
47     .loc      1 13 13 is_stmt 1               # example1.c:13:13
48
49     movzbl   65536(%rsi,%rax), %ecx
50     .loc      1 13 10 is_stmt 0               # example1.c:13:10
51
52     addb     %cl, 65536(%rdi,%rax)
53     .loc      1 13 13                         # example1.c:13:13
54
55     movzbl   65537(%rsi,%rax), %ecx
56     .loc      1 13 10                         # example1.c:13:10
57
58     addb     %cl, 65537(%rdi,%rax)
59     .loc      1 13 13                         # example1.c:13:13
60
61     movzbl   65538(%rsi,%rax), %ecx
62     .loc      1 13 10                         # example1.c:13:10
63
64     addb     %cl, 65538(%rdi,%rax)
65     .loc      1 13 13                         # example1.c:13:13
66
67     movzbl   65539(%rsi,%rax), %ecx
68     .loc      1 13 10                         # example1.c:13:10
69
70     addb     %cl, 65539(%rdi,%rax)

```

```

71 .Ltmp1:
72     .loc    1 12 17 is_stmt 1      # example1.c:12:17
73
74     addq    $4, %rax
75 .Ltmp2:
76     .loc    1 12 3 is_stmt 0      # example1.c:12:3
77
78     jne     .LBB0_5
79     jmp     .LBB0_6
80 .LBB0_2:                                # %vector.body.preheader
81     #DEBUG_VALUE: test:b <- %rsi
82     #DEBUG_VALUE: test:a <- %rdi
83     .loc    1 0 3                  # example1.c:0:3
84
85     movq    $-65536, %rax          # imm = 0xFFFF0000
86     .p2align 4, 0x90
87 .LBB0_3:                                # %vector.body
88                                     # =>This Inner Loop Header: Depth=1
89     #DEBUG_VALUE: test:b <- %rsi
90     #DEBUG_VALUE: test:a <- %rdi
91 .Ltmp3:
92     .loc    1 13 13 is_stmt 1      # example1.c:13:13
93
94     movdqu  65536(%rsi,%rax), %xmm0
95     movdqu  65552(%rsi,%rax), %xmm1
96     .loc    1 13 10 is_stmt 0      # example1.c:13:10
97
98     movdqu  65536(%rdi,%rax), %xmm2
99     paddb   %xmm0, %xmm2
100    movdqu  65552(%rdi,%rax), %xmm0
101    movdqu  65568(%rdi,%rax), %xmm3
102    movdqu  65584(%rdi,%rax), %xmm4
103    movdqu  %xmm2, 65536(%rdi,%rax)
104    paddb   %xmm1, %xmm0
105    movdqu  %xmm0, 65552(%rdi,%rax)
106    .loc    1 13 13                # example1.c:13:13
107
108    movdqu  65568(%rsi,%rax), %xmm0
109    .loc    1 13 10                # example1.c:13:10
110
111    paddb   %xmm3, %xmm0
112    .loc    1 13 13                # example1.c:13:13
113
114    movdqu  65584(%rsi,%rax), %xmm1
115    .loc    1 13 10                # example1.c:13:10
116
117    movdqu  %xmm0, 65568(%rdi,%rax)
118    paddb   %xmm4, %xmm1
119    movdqu  %xmm1, 65584(%rdi,%rax)
120 .Ltmp4:
121     .loc    1 12 26 is_stmt 1      # example1.c:12:26
122
123     addq    $64, %rax
124     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.

```
125 void test(uint8_t * restrict a, uint8_t * restrict b) {  
126     uint64_t i;  
127  
128     for (i = 0; i < SIZE; i++) {  
129         a[i] += b[i];  
130     }  
131 }
```

Now you should see the following assembly code:

```

132 # %bb.0:                                     # %entry
133     #DEBUG_VALUE: test:a <- %rdi
134     #DEBUG_VALUE: test:a <- %rdi
135     #DEBUG_VALUE: test:b <- %rsi
136     #DEBUG_VALUE: test:b <- %rsi
137     movq     $-65536, %rax                    # imm = 0xFFFF0000
138 .Ltmp0:
139     #DEBUG_VALUE: test:i <- 0
140     .p2align    4, 0x90
141 .LBB0_1:                                     # %vector.body
142                                     # =>This Inner Loop Header: Depth=1
143     #DEBUG_VALUE: test:b <- %rsi
144     #DEBUG_VALUE: test:a <- %rdi
145     .loc        1 13 13 prologue_end        # example1.c:13:13
146
147     movdqu     65536(%rsi,%rax), %xmm0
148     .loc        1 13 10 is_stmt 0          # example1.c:13:10
149
150     movdqu     65536(%rdi,%rax), %xmm1
151     paddb      %xmm0, %xmm1
152     movdqu     65552(%rdi,%rax), %xmm0
153     movdqu     65568(%rdi,%rax), %xmm2
154     movdqu     65584(%rdi,%rax), %xmm3
155     movdqu     %xmm1, 65536(%rdi,%rax)
156     .loc        1 13 13                    # example1.c:13:13
157
158     movdqu     65552(%rsi,%rax), %xmm1
159     .loc        1 13 10                    # example1.c:13:10
160
161     paddb      %xmm1, %xmm0
162     movdqu     %xmm0, 65552(%rdi,%rax)
163     .loc        1 13 13                    # example1.c:13:13
164
165     movdqu     65568(%rsi,%rax), %xmm0
166     .loc        1 13 10                    # example1.c:13:10
167
168     paddb      %xmm2, %xmm0
169     .loc        1 13 13                    # example1.c:13:13
170
171     movdqu     65584(%rsi,%rax), %xmm1
172     .loc        1 13 10                    # example1.c:13:10
173
174     movdqu     %xmm0, 65568(%rdi,%rax)
175     paddb      %xmm3, %xmm1
176     movdqu     %xmm1, 65584(%rdi,%rax)
177 .Ltmp1:
178     .loc        1 12 26 is_stmt 1          # example1.c:12:26
179
180     addq       $64, %rax
181     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`:

```
182 void test(uint8_t * restrict a, uint8_t * restrict b) {
183     uint64_t i;
184
185     a = __builtin_assume_aligned(a, 16);
186     b = __builtin_assume_aligned(b, 16);
187
188     for (i = 0; i < SIZE; i++) {
189         a[i] += b[i];
190     }
191 }
```

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

```

192 # %bb.0:                                     # %entry
193     #DEBUG_VALUE: test:a <- %rdi
194     #DEBUG_VALUE: test:a <- %rdi
195     #DEBUG_VALUE: test:b <- %rsi
196     #DEBUG_VALUE: test:b <- %rsi
197     movq    $-65536, %rax                    # imm = 0xFFFF0000
198 .Ltmp0:
199     #DEBUG_VALUE: test:i <- 0
200     .p2align    4, 0x90
201 .LBB0_1:                                     # %vector.body
202                                     # =>This Inner Loop Header: Depth=1
203     #DEBUG_VALUE: test:b <- %rsi
204     #DEBUG_VALUE: test:a <- %rdi
205     .loc      1 16 10 prologue_end          # example1.c:16:10
206
207     movdqa    65536(%rdi,%rax), %xmm0
208     movdqa    65552(%rdi,%rax), %xmm1
209     movdqa    65568(%rdi,%rax), %xmm2
210     movdqa    65584(%rdi,%rax), %xmm3
211     paddb     65536(%rsi,%rax), %xmm0
212     paddb     65552(%rsi,%rax), %xmm1
213     movdqa    %xmm0, 65536(%rdi,%rax)
214     movdqa    %xmm1, 65552(%rdi,%rax)
215     paddb     65568(%rsi,%rax), %xmm2
216     paddb     65584(%rsi,%rax), %xmm3
217     movdqa    %xmm2, 65568(%rdi,%rax)
218     movdqa    %xmm3, 65584(%rdi,%rax)
219 .Ltmp1:
220     .loc      1 15 26                      # example1.c:15:26
221
222     addq      $64, %rax
223     jne       .LBB0_1
224 .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
```



```

225 # %bb.0:                                     # %entry
226     #DEBUG_VALUE: test:a <- %rdi
227     #DEBUG_VALUE: test:a <- %rdi
228     #DEBUG_VALUE: test:b <- %rsi
229     #DEBUG_VALUE: test:b <- %rsi
230     movq    $-65536, %rax                    # imm = 0xFFFF0000
231 .Ltmp0:
232     #DEBUG_VALUE: test:i <- 0
233     .p2align    4, 0x90
234 .LBB0_1:                                     # %vector.body
235                                         # =>This Inner Loop Header: Depth=1
236     #DEBUG_VALUE: test:b <- %rsi
237     #DEBUG_VALUE: test:a <- %rdi
238     .loc      1 16 10 prologue_end         # example1.c:16:10
239
240     vmovdqu   65536(%rdi,%rax), %ymm0
241     vmovdqu   65568(%rdi,%rax), %ymm1
242     vmovdqu   65600(%rdi,%rax), %ymm2
243     vmovdqu   65632(%rdi,%rax), %ymm3
244     vpaddb    65536(%rsi,%rax), %ymm0, %ymm0
245     vpaddb    65568(%rsi,%rax), %ymm1, %ymm1
246     vpaddb    65600(%rsi,%rax), %ymm2, %ymm2
247     vmovdqu   %ymm0, 65536(%rdi,%rax)
248     vmovdqu   %ymm1, 65568(%rdi,%rax)
249     vmovdqu   %ymm2, 65600(%rdi,%rax)
250     vpaddb    65632(%rsi,%rax), %ymm3, %ymm0
251     vmovdqu   %ymm0, 65632(%rdi,%rax)
252 .Ltmp1:
253     .loc      1 15 26                     # example1.c:15:26
254
255     addq      $128, %rax
256     jne       .LBB0_1
257 .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`:

```
258 void test(uint8_t * restrict a, uint8_t * restrict b) {
259     uint64_t i;
260
261     uint8_t * x = __builtin_assume_aligned(a, 16);
262     uint8_t * y = __builtin_assume_aligned(b, 16);
263
264     for (i = 0; i < SIZE; i++) {
265         /* max() */
266         if (y[i] > x[i]) x[i] = y[i];
267     }
268 }
```

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:

```
269 void test(uint8_t * restrict a, uint8_t * restrict b) {
270     uint64_t i;
271
272     a = __builtin_assume_aligned(a, 16);
273     b = __builtin_assume_aligned(b, 16);
274
275     for (i = 0; i < SIZE; i++) {
276         /* max() */
277         a[i] = (b[i] > a[i]) ? b[i] : a[i];
278     }
279 }
```

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

```

280 # %bb.0:                                     # %entry
281     #DEBUG_VALUE: test:a <- %rdi
282     #DEBUG_VALUE: test:a <- %rdi
283     #DEBUG_VALUE: test:b <- %rsi
284     #DEBUG_VALUE: test:b <- %rsi
285     movq    $-65536, %rax                      # imm = 0xFFFF0000
286 .Ltmp0:
287     #DEBUG_VALUE: test:i <- 0
288     .p2align    4, 0x90
289 .LBB0_1:                                     # %vector.body
290                                     # =>This Inner Loop Header: Depth=1
291     #DEBUG_VALUE: test:b <- %rsi
292     #DEBUG_VALUE: test:a <- %rdi
293     .loc      1 17 15 prologue_end          # example2.c:17:15
294
295     movdqa    65536(%rsi,%rax), %xmm0
296     movdqa    65552(%rsi,%rax), %xmm1
297     .loc      1 17 14 is_stmt 0            # example2.c:17:14
298
299     pmaxub    65536(%rdi,%rax), %xmm0
300     pmaxub    65552(%rdi,%rax), %xmm1
301     .loc      1 17 12                      # example2.c:17:12
302
303     movdqa    %xmm0, 65536(%rdi,%rax)
304     movdqa    %xmm1, 65552(%rdi,%rax)
305     .loc      1 17 15                      # example2.c:17:15
306
307     movdqa    65568(%rsi,%rax), %xmm0
308     movdqa    65584(%rsi,%rax), %xmm1
309     .loc      1 17 14                      # example2.c:17:14
310
311     pmaxub    65568(%rdi,%rax), %xmm0
312     pmaxub    65584(%rdi,%rax), %xmm1
313     .loc      1 17 12                      # example2.c:17:12
314
315     movdqa    %xmm0, 65568(%rdi,%rax)
316     movdqa    %xmm1, 65584(%rdi,%rax)
317 .Ltmp1:
318     .loc      1 15 28 is_stmt 1            # example2.c:15:28
319
320     addq      $64, %rax
321     jne       .LBB0_1
322 .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
```

```
323 void test(uint8_t * restrict a, uint8_t * restrict b) {
324     uint64_t i;
325
326     for (i = 0; i < SIZE; i++) {
327         a[i] = b[i + 1];
328     }
329 }
```

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`.

```
330 double test(double * restrict a) {
331     size_t i;
332
333     double *x = __builtin_assume_aligned(a, 16);
334
335     double y = 0;
336
337     for (i = 0; i < SIZE; i++) {
338         y += x[i];
339     }
340     return y;
341 }
```

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

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

```

342 .LBB0_1:                                     # %for.body
343                                     # =>This Inner Loop Header: Depth=1
344     #DEBUG_VALUE: test:x <- %rdi
345     #DEBUG_VALUE: test:a <- %rdi
346 .Ltmp1:
347     #DEBUG_VALUE: test:y <- %xmm0
348     .loc      1 18 7 prologue_end           # example4.c:18:7
349
350     addsd     524288(%rdi,%rax,8), %xmm0
351 .Ltmp2:
352     #DEBUG_VALUE: test:y <- %xmm0
353     addsd     524296(%rdi,%rax,8), %xmm0
354 .Ltmp3:
355     #DEBUG_VALUE: test:y <- %xmm0
356     addsd     524304(%rdi,%rax,8), %xmm0
357 .Ltmp4:
358     #DEBUG_VALUE: test:y <- %xmm0
359     addsd     524312(%rdi,%rax,8), %xmm0
360 .Ltmp5:
361     #DEBUG_VALUE: test:y <- %xmm0
362     addsd     524320(%rdi,%rax,8), %xmm0
363 .Ltmp6:
364     #DEBUG_VALUE: test:y <- %xmm0
365     addsd     524328(%rdi,%rax,8), %xmm0
366 .Ltmp7:
367     #DEBUG_VALUE: test:y <- %xmm0
368     addsd     524336(%rdi,%rax,8), %xmm0
369 .Ltmp8:
370     #DEBUG_VALUE: test:y <- %xmm0
371     addsd     524344(%rdi,%rax,8), %xmm0
372 .Ltmp9:
373     #DEBUG_VALUE: test:y <- %xmm0
374     .loc      1 17 17                       # example4.c:17:17
375
376     addq      $8, %rax
377 .Ltmp10:
378     .loc      1 17 3 is_stmt 0              # example4.c:17:3
379
380     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 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 see a difference in the output.

```

381 # %bb.0:                                     # %entry
382     #DEBUG_VALUE: test:a <- %rdi
383     #DEBUG_VALUE: test:a <- %rdi
384     #DEBUG_VALUE: test:x <- %rdi
385     #DEBUG_VALUE: test:x <- %rdi
386     xorpd    %xmm0, %xmm0
387 .Ltmp0:
388     #DEBUG_VALUE: test:i <- 0
389     #DEBUG_VALUE: test:y <- 0.000000e+00
390     movq     $-65536, %rax                    # imm = 0xFFFF0000
391     xorpd    %xmm1, %xmm1
392     .p2align    4, 0x90
393 .LBB0_1:                                     # %vector.body
394     #=>This Inner Loop Header: Depth=1
395     #DEBUG_VALUE: test:x <- %rdi
396     #DEBUG_VALUE: test:a <- %rdi
397 .Ltmp1:
398     .loc      1 18 7 prologue_end            # example4.c:18:7
399
400     addpd     524288(%rdi,%rax,8), %xmm0
401     addpd     524304(%rdi,%rax,8), %xmm1
402     addpd     524320(%rdi,%rax,8), %xmm0
403     addpd     524336(%rdi,%rax,8), %xmm1
404     addpd     524352(%rdi,%rax,8), %xmm0
405     addpd     524368(%rdi,%rax,8), %xmm1
406     addpd     524384(%rdi,%rax,8), %xmm0
407     addpd     524400(%rdi,%rax,8), %xmm1
408 .Ltmp2:
409     .loc      1 17 26                        # example4.c:17:26
410
411     addq      $16, %rax
412     jne       .LBB0_1
413 # %bb.2:                                     # %middle.block
414     #DEBUG_VALUE: test:x <- %rdi
415     #DEBUG_VALUE: test:a <- %rdi
416 .Ltmp3:
417     .loc      1 18 7                        # example4.c:18:7
418
419     addpd     %xmm0, %xmm1
420     movapd    %xmm1, %xmm0
421     movhps    %xmm0, %xmm0                    # xmm0 = xmm0[1,1]
422
423     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., $2\times$, $3\times$, 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` flag¹. 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
```

¹If 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?

²We 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://l1vm.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,

```
424 for (j = 0; j < N; j+=2) {  
425     C[j] = A[j] + B[j];  
426 }
```

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:

```
427 for (j = 0; j < N; j++) {  
428     total += A[j];  
429 }
```

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.