Performance Engineering of Software Systems September 21, 2021
Massachusetts Institute of Technology 6.172
Saman Amarasinghe, Charles E. Leiserson, and Jonathan Ragan-Kelley Handout 5

Homework 3: Vectorization

Due: 11:59 P.M. (ET) on Tuesday, September 28, 2021

(Last Updated: September 22, 2021)

In this homework you will experiment with vectorization. You will practice examining and comparing the LLVM IR and assembly outputs of `clang` *for vectorized code. You will examine cases when* `clang` *can and cannot vectorize code. You will experiment with compiler builtins to vectorize code by hand.*

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.*

Contents

1 Getting started

You can get this assignment's code using MIT's internal GitHub system:

```
$ git clone git@github.mit.edu:6172-fall21/homework3_<your_kerberos>.git homework3
```

This repository contains a `compilervec/` subdirectory and a `matmul/` subdirectory. The `compilervec/` subdirectory contains the code for Section 2 and the first five write-up questions. The `matmul/` subdirectory contains code for Section 3.

```c
#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];
  }
}
```

Figure 1: Original C code in `example1.c`.

Submitting your solutions

We will use the same submission procedures as in Homework 2. Submit your write-up on Gradescope and your code via Git by the deadline stated at the top of this handout. *For each write-up question (some write-ups include multiple questions, e.g., write-up 10), respond with a short (1–3 sentence) response or a code snippet (if requested).* **Please ensure that all the times you quote are obtained with `awsrun`.**

2 Vectorization in `clang`

Consider a loop that performs an elementwise operation, such as addition, between two independent arrays `A` and `B`, storing the result in array `C`. This loop is an example of a ***data parallel*** loop, since the data processed in distinct iterations i_1 and i_2 can be safely distributed across different hardware processing elements and processed in parallel. Compilers can take advantage of data parallelism using vectorization, which means directing the hardware to process different data elements in distinct lanes of the processor's vector units. Vector units perform the same operation simultaneously on every lane of the vector unit. This pattern of parallel processing is called ***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.

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

Figure 2: Example vectorization report from compiling `example1.c`. For more information on autovectorization reports see `https://llvm.org/docs/Vectorizers.html`

2.1 Example 1

We will start with the simple loop shown in Figure 1, which is available in the `compilervec/` subdirectory of the Git repository. Using this example, we shall examine the LLVM IR and assembly code `clang` generates for a simple vectorizable loop. We shall also examine some simple ways to control how `clang` vectorizes code. The provided `Makefile` allows you to generate the compiled and optimized LLVM IR for this vectorizable loop using the `LLVMIR=1` flag, as follows:

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

Similarly, you can generate the assembly code for this example using the `ASSEMBLE=1` flag:

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

The `VECTORIZE=1` flag directs `clang` to generate a ***vectorization report***, which indicates which loops in the program were successfully vectorized and which were not. You should see the vectorization report shown in Figure 2 as output when you run either of these commands. This report indicates that the loop has been vectorized. But this report doesn't tell the whole story, as we shall see when we investigate the LLVM IR and assembly outputs for the example. Let's first inspect the LLVM IR output from running the above `make` command with `LLVMIR=1`. This command will produce the file `example1.ll`, which contains the optimized LLVM IR for the example. You should find that the contents of `example1.ll` resembles that in Figures 3 and 4. **The line numbers will most likely differ on your machine. For line numbers in this homework, refer to the documented code below.** The vectorized operations in the LLVM IR output are those that operate on an LLVM vector type, such as `<16 x i8>` in lines 29–69 in `example1.ll`. *Note:* For all examples, you might find additional content in the compiled LLVM IR and assembly outputs, such as `!dbg` metadata tags and calls to `@llvm.dbg.value` in the LLVM IR, and additional comments, labels, and `.loc` directives in the assembly output. This additional output reflects the debugging symbols compiled with the example codes and can safely be ignored when studying vectorization.

Now run the `make` command above with the flag `ASSEMBLE=1` to generate the assembly code for this example. The command will generate the file `example1.s`, which contains the assembly code for this example. You should find that the contents of `example1.s` resembles that shown in Figures 5 and 6.

```
; Function Attrs: argmemonly norecurse nounwind uwtable
define dso_local void @test(i8* nocapture, i8* nocapture readonly)
local_unnamed_addr #0 {
  %3 = getelementptr i8, i8* %0, i64 65536
  %4 = getelementptr i8, i8* %1, i64 65536
  %5 = icmp ugt i8* %4, %0
  %6 = icmp ugt i8* %3, %1
  %7 = and i1 %5, %6
  br i1 %7, label %45, label %8

; <label>:8:                                      ; preds = %2, %8
  %9 = phi i64 [ %43, %8 ], [ 0, %2 ]
  %10 = getelementptr inbounds i8, i8* %1, i64 %9
  %11 = bitcast i8* %10 to <16 x i8>*
  %12 = load <16 x i8>, <16 x i8>* %11, align 1, !tbaa !2, !alias.scope !5
  %13 = getelementptr inbounds i8, i8* %10, i64 16
  %14 = bitcast i8* %13 to <16 x i8>*
  %15 = load <16 x i8>, <16 x i8>* %14, align 1, !tbaa !2, !alias.scope !5
  %16 = getelementptr inbounds i8, i8* %0, i64 %9
  %17 = bitcast i8* %16 to <16 x i8>*
  %18 = load <16 x i8>, <16 x i8>* %17, align 1, !tbaa !2, !alias.scope !8, !noalias !5
  %19 = getelementptr inbounds i8, i8* %16, i64 16
  %20 = bitcast i8* %19 to <16 x i8>*
  %21 = load <16 x i8>, <16 x i8>* %20, align 1, !tbaa !2, !alias.scope !8, !noalias !5
  %22 = add <16 x i8> %18, %12
  %23 = add <16 x i8> %21, %15
  %24 = bitcast i8* %16 to <16 x i8>*
  store <16 x i8> %22, <16 x i8>* %24, align 1, !tbaa !2, !alias.scope !8, !noalias !5
  %25 = bitcast i8* %19 to <16 x i8>*
  store <16 x i8> %23, <16 x i8>* %25, align 1, !tbaa !2, !alias.scope !8, !noalias !5
  %26 = or i64 %9, 32
  %27 = getelementptr inbounds i8, i8* %1, i64 %26
  %28 = bitcast i8* %27 to <16 x i8>*
  %29 = load <16 x i8>, <16 x i8>* %28, align 1, !tbaa !2, !alias.scope !5
  %30 = getelementptr inbounds i8, i8* %27, i64 16
  %31 = bitcast i8* %30 to <16 x i8>*
  %32 = load <16 x i8>, <16 x i8>* %31, align 1, !tbaa !2, !alias.scope !5
  %33 = getelementptr inbounds i8, i8* %0, i64 %26
  %34 = bitcast i8* %33 to <16 x i8>*
  %35 = load <16 x i8>, <16 x i8>* %34, align 1, !tbaa !2, !alias.scope !8, !noalias !5
  %36 = getelementptr inbounds i8, i8* %33, i64 16
  %37 = bitcast i8* %36 to <16 x i8>*
  %38 = load <16 x i8>, <16 x i8>* %37, align 1, !tbaa !2, !alias.scope !8, !noalias !5
  %39 = add <16 x i8> %35, %29
  %40 = add <16 x i8> %38, %32
```

Figure 3: First part of LLVM IR from compiling the code in Figure 1.

```
%41 = bitcast i8* %33 to <16 x i8>*
store <16 x i8> %39, <16 x i8>* %41, align 1, !tbaa !2, !alias.scope !8, !noalias !5
%42 = bitcast i8* %36 to <16 x i8>*
store <16 x i8> %40, <16 x i8>* %42, align 1, !tbaa !2, !alias.scope !8, !noalias !5
%43 = add nuw nsw i64 %9, 64
%44 = icmp eq i64 %43, 65536
br i1 %44, label %72, label %8, !llvm.loop !10

; <label>:45:                                     ; preds = %2, %45
%46 = phi i64 [ %70, %45 ], [ 0, %2 ]
%47 = getelementptr inbounds i8, i8* %1, i64 %46
%48 = load i8, i8* %47, align 1, !tbaa !2
%49 = getelementptr inbounds i8, i8* %0, i64 %46
%50 = load i8, i8* %49, align 1, !tbaa !2
%51 = add i8 %50, %48
store i8 %51, i8* %49, align 1, !tbaa !2
%52 = or i64 %46, 1
%53 = getelementptr inbounds i8, i8* %1, i64 %52
%54 = load i8, i8* %53, align 1, !tbaa !2
%55 = getelementptr inbounds i8, i8* %0, i64 %52
%56 = load i8, i8* %55, align 1, !tbaa !2
%57 = add i8 %56, %54
store i8 %57, i8* %55, align 1, !tbaa !2
%58 = or i64 %46, 2
%59 = getelementptr inbounds i8, i8* %1, i64 %58
%60 = load i8, i8* %59, align 1, !tbaa !2
%61 = getelementptr inbounds i8, i8* %0, i64 %58
%62 = load i8, i8* %61, align 1, !tbaa !2
%63 = add i8 %62, %60
store i8 %63, i8* %61, align 1, !tbaa !2
%64 = or i64 %46, 3
%65 = getelementptr inbounds i8, i8* %1, i64 %64
%66 = load i8, i8* %65, align 1, !tbaa !2
%67 = getelementptr inbounds i8, i8* %0, i64 %64
%68 = load i8, i8* %67, align 1, !tbaa !2
%69 = add i8 %68, %66
store i8 %69, i8* %67, align 1, !tbaa !2
%70 = add nuw nsw i64 %46, 4
%71 = icmp eq i64 %70, 65536
br i1 %71, label %72, label %45, !llvm.loop !12

; <label>:72:                                     ; preds = %8, %45
ret void
}
```

Figure 4: Second part of LLVM IR from compiling the code in Figure 1.

```
107 test:                                   # @test
108         .cfi_startproc
109 # %bb.0:
110         leaq    65536(%rsi), %rax
111         cmpq    %rdi, %rax
112         jbe     .LBB0_2
113 # %bb.1:
114         leaq    65536(%rdi), %rax
115         cmpq    %rsi, %rax
116         jbe     .LBB0_2
117 # %bb.4:
118         xorl    %eax, %eax
119         .p2align        4, 0x90
120 .LBB0_5:                                # =>This Inner Loop Header: Depth=1
121         movzbl  (%rsi,%rax), %ecx
122         addb    %cl, (%rdi,%rax)
123         movzbl  1(%rsi,%rax), %ecx
124         addb    %cl, 1(%rdi,%rax)
125         movzbl  2(%rsi,%rax), %ecx
126         addb    %cl, 2(%rdi,%rax)
127         movzbl  3(%rsi,%rax), %ecx
128         addb    %cl, 3(%rdi,%rax)
129         addq    $4, %rax
130         cmpq    $65536, %rax            # imm = 0x10000
131         jne     .LBB0_5
132         jmp     .LBB0_6
```

Figure 5: First part of assembly output from compiling the code in Figure 1.

Both the LLVM IR and assembly output show that `clang` uses ***multiversioning*** to vectorize the loop. Consider the LLVM IR, for example. On lines 21–26, the code first checks if there is any ***aliasing*** between the arrays `a` and `b`. Aliasing means that the arrays overlap, such that some memory locations accessed through `a` are also accessed through `b`. If there is aliasing, then a simple non-vectorized loop is run (lines 72–102). If there is no aliasing, then a vectorized version of the loop is run (lines 29–69).

Write-up 1: Compare the LLVM IR output and the assembly output for `example1.c`. Which lines of the assembly output correspond to the following ranges of lines of the LLVM IR output?

- Lines 21–26
- Lines 29–69
- Lines 72–102

```
.LBB0_2:
        xorl    %eax, %eax
        .p2align        4, 0x90
.LBB0_3:                                # =>This Inner Loop Header: Depth=1
        movdqu  (%rsi,%rax), %xmm0
        movdqu  16(%rsi,%rax), %xmm1
        movdqu  (%rdi,%rax), %xmm2
        paddb   %xmm0, %xmm2
        movdqu  16(%rdi,%rax), %xmm0
        paddb   %xmm1, %xmm0
        movdqu  32(%rdi,%rax), %xmm1
        movdqu  48(%rdi,%rax), %xmm3
        movdqu  %xmm2, (%rdi,%rax)
        movdqu  %xmm0, 16(%rdi,%rax)
        movdqu  32(%rsi,%rax), %xmm0
        paddb   %xmm1, %xmm0
        movdqu  48(%rsi,%rax), %xmm1
        paddb   %xmm3, %xmm1
        movdqu  %xmm0, 32(%rdi,%rax)
        movdqu  %xmm1, 48(%rdi,%rax)
        addq    $64, %rax
        cmpq    $65536, %rax            # imm = 0x10000
        jne     .LBB0_3
.LBB0_6:
        retq
```

Figure 6: Second part of assembly output from compiling the code in Figure 1.

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

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

Figure 7: First modification to `example1.c`, which uses the `restrict` keyword.

Although this code is vectorized, multiversioning introduces additional overhead due to the initial check for aliasing and the size of the code. In our case, we know that the arrays `a` and `b` never alias, meaning that these overheads are unnecessary. We can get `clang` to generate faster vectorized code, without the overheads of multiversioning, by informing `clang` that `a` and `b` never alias. To accomplish this, we can annotate the pointers using the `restrict` qualifier in standard C, as shown in Figure 7.

Compiling the code in Figure 7 with `LLVMIR=1` should produce LLVM IR resembling that shown in Figures 8 and 9. Notice that the function pointer arguments in the LLVM IR are marked with the `noalias` attribute, reflecting the `restrict` qualifier added to the function arguments in the C code. Compiling the code in Figure 7 with `ASSEMBLE=1` should produce assembly code resembling that shown in Figure 10.

The generated code avoids the overheads of multiversioning, but it can still be improved. Some processors can perform more efficient vector operations on ***aligned*** data, which is stored at memory addresses that are multiples of the vector width. In the example code, both the generated LLVM IR and assembly indicate that the compiler does not assume that the data is aligned. In the LLVM IR, the `align` attribute on the vector `load` and `store` instructions shows that `clang` only assumes that the data are 1-byte aligned. Correspondingly, the assembly code uses the `movdqu` instruction, which performs an unaligned move. There are various ways we can get `clang` to generate more efficient vectorized code for aligned data. One way is to define a custom data type with an attribute that conveys the data alignment of that type. Another is to use a specialized memory-allocation routine, such as `aligned_alloc` in modern C, to ensure that dynamically allocated memory is properly aligned. Third, `clang` supports the `__builtin_assume_aligned` intrinsic that we can use to tell `clang` to assume that a given pointer has a specified alignment.

Modify `example1.c` to use the `__builin_assume_aligned` intrinsic as shown in Figure 11. Then, recompile `example1.c` to produce LLVM IR output. The LLVM IR should resemble that shown in Figure 12. As the LLVM IR shows, the `align` attribute on the vector `load` and `store` operations matches the specified alignment of 16 bytes.

Write-up 2: The optimized assembly code in Figure 13 is shorter than the previous version, shown in Figure 10. What changed? In other words, how else has `clang` optimized the assembly code, thanks to the alignment information?

Now, finally, we get the nice and tight vectorized code (`movdqa` is an 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 elements at a time, which it did not do before. The question is, now that we understand what we need to tell the compiler, how much more complex can the loop be before autovectorization fails.

The `Makefile` allows us to compile `example1.c` with AVX2 instructions using the `AVX2=1` flag. Compile the assembly code for `example1.c` with AVX2 instructions using the following com-

```
; Function Attrs: argmemonly norecurse nounwind uwtable
define dso_local void @test(i8* noalias nocapture, i8* noalias nocapture readonly)
local_unnamed_addr #0 {
  br label %3

; <label>:3:                                      ; preds = %3, %2
  %4 = phi i64 [ 0, %2 ], [ %38, %3 ]
  %5 = getelementptr inbounds i8, i8* %1, i64 %4
  %6 = bitcast i8* %5 to <16 x i8>*
  %7 = load <16 x i8>, <16 x i8>* %6, align 1, !tbaa !2
  %8 = getelementptr inbounds i8, i8* %5, i64 16
  %9 = bitcast i8* %8 to <16 x i8>*
  %10 = load <16 x i8>, <16 x i8>* %9, align 1, !tbaa !2
  %11 = getelementptr inbounds i8, i8* %0, i64 %4
  %12 = bitcast i8* %11 to <16 x i8>*
  %13 = load <16 x i8>, <16 x i8>* %12, align 1, !tbaa !2
  %14 = getelementptr inbounds i8, i8* %11, i64 16
  %15 = bitcast i8* %14 to <16 x i8>*
  %16 = load <16 x i8>, <16 x i8>* %15, align 1, !tbaa !2
  %17 = add <16 x i8> %13, %7
  %18 = add <16 x i8> %16, %10
  %19 = bitcast i8* %11 to <16 x i8>*
  store <16 x i8> %17, <16 x i8>* %19, align 1, !tbaa !2
  %20 = bitcast i8* %14 to <16 x i8>*
  store <16 x i8> %18, <16 x i8>* %20, align 1, !tbaa !2
  %21 = or i64 %4, 32
  %22 = getelementptr inbounds i8, i8* %1, i64 %21
  %23 = bitcast i8* %22 to <16 x i8>*
  %24 = load <16 x i8>, <16 x i8>* %23, align 1, !tbaa !2
  %25 = getelementptr inbounds i8, i8* %22, i64 16
  %26 = bitcast i8* %25 to <16 x i8>*
  %27 = load <16 x i8>, <16 x i8>* %26, align 1, !tbaa !2
  %28 = getelementptr inbounds i8, i8* %0, i64 %21
  %29 = bitcast i8* %28 to <16 x i8>*
  %30 = load <16 x i8>, <16 x i8>* %29, align 1, !tbaa !2
  %31 = getelementptr inbounds i8, i8* %28, i64 16
  %32 = bitcast i8* %31 to <16 x i8>*
  %33 = load <16 x i8>, <16 x i8>* %32, align 1, !tbaa !2
```

Figure 8: First part of LLVM IR from compiling the code in Figure 7.

```
%34 = add <16 x i8> %30, %24
%35 = add <16 x i8> %33, %27
%36 = bitcast i8* %28 to <16 x i8>*
store <16 x i8> %34, <16 x i8>* %36, align 1, !tbaa !2
%37 = bitcast i8* %31 to <16 x i8>*
store <16 x i8> %35, <16 x i8>* %37, align 1, !tbaa !2
%38 = add nuw nsw i64 %4, 64
%39 = icmp eq i64 %38, 65536
br i1 %39, label %40, label %3, !llvm.loop !5

; <label>:40:                                     ; preds = %3
ret void
}
```

Figure 9: Second part of LLVM IR from compiling the code in Figure 7.

```
test:                                   # @test
        .cfi_startproc
# %bb.0:
        xorl    %eax, %eax
        .p2align        4, 0x90
.LBB0_1:                                # =>This Inner Loop Header: Depth=1
        movdqu  (%rsi,%rax), %xmm0
        movdqu  16(%rsi,%rax), %xmm1
        movdqu  (%rdi,%rax), %xmm2
        paddb   %xmm0, %xmm2
        movdqu  16(%rdi,%rax), %xmm0
        paddb   %xmm1, %xmm0
        movdqu  32(%rdi,%rax), %xmm1
        movdqu  48(%rdi,%rax), %xmm3
        movdqu  %xmm2, (%rdi,%rax)
        movdqu  %xmm0, 16(%rdi,%rax)
        movdqu  32(%rsi,%rax), %xmm0
        paddb   %xmm1, %xmm0
        movdqu  48(%rsi,%rax), %xmm1
        paddb   %xmm3, %xmm1
        movdqu  %xmm0, 32(%rdi,%rax)
        movdqu  %xmm1, 48(%rdi,%rax)
        addq    $64, %rax
        cmpq    $65536, %rax            # imm = 0x10000
        jne     .LBB0_1
# %bb.2:
        retq
```

Figure 10: Assembly output from compiling the code in Figure 7.

```
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];
  }
}
```

Figure 11: Second modification to `example1.c`, to instruct `clang` to assume a particular alignment on pointers.

mand:

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

You should see assembly output like the one in Figure 14. From that output, we can confirm that the loop is vectorized using the `vmov` and `vpadd` AVX2 instructions and uses the 256-bit `%ymm` registers.

Write-up 3: The AVX2-vectorized code uses unaligned move instructions. Modify `example1.c` to make sure it uses aligned move instructions for the best performance, and paste the relevant assembly code in your writeup. Commit and push your final implementation of `example1.c`.

2.2 Example 2

The next example illustrates how different implementations of a loop can lead to different vectorizations. Consider the code in `example2.c`, which is reproduced in Figure 15. Examine the LLVM IR and assembly that `clang` compiles for `example2.c`. You can use similar commands to those described in Section 2.1:

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

Contrast the LLVM IR and assembly output from compiling `example2.c` to the output you get if you modify `example2.c` as shown in Figure 16. You should find that, compared to the original, the revised version of `example2.c` produces a tighter vectorized loop. For example, the assembly output for the second implementation should look similar to that shown in Figure 17.

```
; <label>:9:                                      ; preds = %9, %2
  %10 = phi i64 [ 0, %2 ], [ %44, %9 ]
  %11 = getelementptr inbounds i8, i8* %1, i64 %10
  %12 = bitcast i8* %11 to <16 x i8>*
  %13 = load <16 x i8>, <16 x i8>* %12, align 16, !tbaa !2
  %14 = getelementptr inbounds i8, i8* %11, i64 16
  %15 = bitcast i8* %14 to <16 x i8>*
  %16 = load <16 x i8>, <16 x i8>* %15, align 16, !tbaa !2
  %17 = getelementptr inbounds i8, i8* %0, i64 %10
  %18 = bitcast i8* %17 to <16 x i8>*
  %19 = load <16 x i8>, <16 x i8>* %18, align 16, !tbaa !2
  %20 = getelementptr inbounds i8, i8* %17, i64 16
  %21 = bitcast i8* %20 to <16 x i8>*
  %22 = load <16 x i8>, <16 x i8>* %21, align 16, !tbaa !2
  %23 = add <16 x i8> %19, %13
  %24 = add <16 x i8> %22, %16
  %25 = bitcast i8* %17 to <16 x i8>*
  store <16 x i8> %23, <16 x i8>* %25, align 16, !tbaa !2
  %26 = bitcast i8* %20 to <16 x i8>*
  store <16 x i8> %24, <16 x i8>* %26, align 16, !tbaa !2
  %27 = or i64 %10, 32
  %28 = getelementptr inbounds i8, i8* %1, i64 %27
  %29 = bitcast i8* %28 to <16 x i8>*
  %30 = load <16 x i8>, <16 x i8>* %29, align 16, !tbaa !2
  %31 = getelementptr inbounds i8, i8* %28, i64 16
  %32 = bitcast i8* %31 to <16 x i8>*
  %33 = load <16 x i8>, <16 x i8>* %32, align 16, !tbaa !2
  %34 = getelementptr inbounds i8, i8* %0, i64 %27
  %35 = bitcast i8* %34 to <16 x i8>*
  %36 = load <16 x i8>, <16 x i8>* %35, align 16, !tbaa !2
  %37 = getelementptr inbounds i8, i8* %34, i64 16
  %38 = bitcast i8* %37 to <16 x i8>*
  %39 = load <16 x i8>, <16 x i8>* %38, align 16, !tbaa !2
  %40 = add <16 x i8> %36, %30
  %41 = add <16 x i8> %39, %33
  %42 = bitcast i8* %34 to <16 x i8>*
  store <16 x i8> %40, <16 x i8>* %42, align 16, !tbaa !2
  %43 = bitcast i8* %37 to <16 x i8>*
  store <16 x i8> %41, <16 x i8>* %43, align 16, !tbaa !2
  %44 = add nuw nsw i64 %10, 64
  %45 = icmp eq i64 %44, 65536
  br i1 %45, label %46, label %9, !llvm.loop !5
```

Figure 12: LLVM IR from compiling the code in Figure 11.

```
test:                                   # @test
        .cfi_startproc
# %bb.0:
        xorl    %eax, %eax
        .p2align        4, 0x90
.LBB0_1:                                # =>This Inner Loop Header: Depth=1
        movdqa  (%rdi,%rax), %xmm0
        movdqa  16(%rdi,%rax), %xmm1
        movdqa  32(%rdi,%rax), %xmm2
        movdqa  48(%rdi,%rax), %xmm3
        paddb   (%rsi,%rax), %xmm0
        paddb   16(%rsi,%rax), %xmm1
        movdqa  %xmm0, (%rdi,%rax)
        movdqa  %xmm1, 16(%rdi,%rax)
        paddb   32(%rsi,%rax), %xmm2
        paddb   48(%rsi,%rax), %xmm3
        movdqa  %xmm2, 32(%rdi,%rax)
        movdqa  %xmm3, 48(%rdi,%rax)
        addq    $64, %rax
        cmpq    $65536, %rax            # imm = 0x10000
        jne     .LBB0_1
# %bb.2:
        retq
```

Figure 13: Assembly compiled from the code in Figure 11.

Write-up 4: Provide a theory for why the compiler generates dramatically different assembly for these two different implementations of `example2.c`.

2.3 Example 3

Consider `example3.c`, whose code is reproduced in Figure 18. Generate either the LLVM IR or assembly for `example3.c`, using `make` commands similar to those in Section 2.1.

Write-up 5: (Optional) Determine why `clang` does not generate vector instructions for this code. Do you think it would be faster if it did vectorize? Explain.

```
test:                                   # @test
        .cfi_startproc
# %bb.0:
        xorl    %eax, %eax
        .p2align        4, 0x90
.LBB0_1:                                # =>This Inner Loop Header: Depth=1
        vmovdqu (%rdi,%rax), %ymm0
        vmovdqu 32(%rdi,%rax), %ymm1
        vmovdqu 64(%rdi,%rax), %ymm2
        vmovdqu 96(%rdi,%rax), %ymm3
        vpaddb  (%rsi,%rax), %ymm0, %ymm0
        vpaddb  32(%rsi,%rax), %ymm1, %ymm1
        vpaddb  64(%rsi,%rax), %ymm2, %ymm2
        vpaddb  96(%rsi,%rax), %ymm3, %ymm3
        vmovdqu %ymm0, (%rdi,%rax)
        vmovdqu %ymm1, 32(%rdi,%rax)
        vmovdqu %ymm2, 64(%rdi,%rax)
        vmovdqu %ymm3, 96(%rdi,%rax)
        subq    $-128, %rax
        cmpq    $65536, %rax            # imm = 0x10000
        jne     .LBB0_1
# %bb.2:
        vzeroupper
        retq
```

Figure 14: Assembly output from compiling the code in Figure 11 with AVX2 instructions.

```
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];
  }
}
```

Figure 15: Original C code in `example2.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() */
    x[i] = (y[i] > x[i]) ? y[i] : x[i];
  }
}
```

Figure 16: Modified C code for `example2.c`.

```
test:                                   # @test
        .cfi_startproc
# %bb.0:
        xorl    %eax, %eax
        .p2align        4, 0x90
.LBB0_1:                                # =>This Inner Loop Header: Depth=1
        movdqa  (%rsi,%rax), %xmm0
        movdqa  16(%rsi,%rax), %xmm1
        pmaxub  (%rdi,%rax), %xmm0
        pmaxub  16(%rdi,%rax), %xmm1
        movdqa  %xmm0, (%rdi,%rax)
        movdqa  %xmm1, 16(%rdi,%rax)
        movdqa  32(%rsi,%rax), %xmm0
        movdqa  48(%rsi,%rax), %xmm1
        pmaxub  32(%rdi,%rax), %xmm0
        pmaxub  48(%rdi,%rax), %xmm1
        movdqa  %xmm0, 32(%rdi,%rax)
        movdqa  %xmm1, 48(%rdi,%rax)
        addq    $64, %rax
        cmpq    $65536, %rax            # imm = 0x10000
        jne     .LBB0_1
# %bb.2:
        retq
```

Figure 17: Assembly output from compiling the code in Figure 16.

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

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

Figure 18: Original C code in `example3.c`.

3 Optimizing matrix multiplication using vectorization

We will now explore how to optimize dense square matrix multiplication using vectorization. For this section, we will be working with the matrix-multiplication code in `matmul.c` within the `matmul/` subdirectory of the Git repository. This code implements a simple tiled algorithm for square matrix multiplication, where the dimension n of the matrices is 1024. The `matmul_base` routine `matmul.c` is called to process a single tile. We will investigate a couple aspects of how `clang` can automatically vectorize this code. We will then use an extension supported by `clang` to implement a more efficient vectorized base case ourselves.

3.1 autovectorization of matrix multiplication

Let us first investigate how `clang` vectorizes the code `matmul.c`. Compile `matmul.c` using `make` with AVX2 and fused multiply add (FMA) instructions as follows:

```
$ make VECTORIZE=1 AVX2=1 FMA=1
```

You will see from the vectorization report that this matrix multiplication code — specifically, the vectorization report indicates the loop in `matmul_base` — is not vectorized:

```
matmul.c:45:7: remark: loop not vectorized [-Rpass-missed=loop-vectorize]
      for (int k = 0; k < size; ++k) {
      ^
```

In addition, you can examine the LLVM IR and assembly generated from compiling `matmul.c` and verify that the compiled `matmul_base` function does not include vector instructions. You can generate LLVM IR or assembly for `matmul.c` by passing the `LLVMIR=1` and `ASSEMBLE=1` flags, respectively, to `make`. The `vmulsd` and `vaddsd` instructions operate on scalar floating-point values.

The reason `clang` does not vectorize the given `matmul.c` code is in part because of floating-point arithmetic and in part because of limitations in `clang`'s autovectorization capabilities. Floating-point arithmetic is not associative, meaning that reordering floating-point operations can change the value those operations produce. Some applications that use floating-point arithmetic are sensitive to such changes. To support such applications, compilers are not allowed by default to reorder floating-point computation. This restriction inhibits `clang`'s ability to find an efficient vectorization of the program.

We have a couple of options for addressing this issue. First, because we do not mind slight changes in the floating-point values computed when multiplying matrices, it would be acceptable for us to pretend that floating-point arithmetic is associative. We can instruct `clang` to assume that floating-point arithmetic is associative by passing the `-ffast-math` flag at compile time. The Makefile allows us to pass the `-ffast-math` flag to `clang` at compile time by specifying the flag `EXTRA_CFLAGS="-ffast-math"` as follows:

```
$ make VECTORIZE=1 AVX2=1 FMA=1 EXTRA_CFLAGS="-ffast-math"
```

Alternatively, we can reorder the loops in `matmul_base` to enable vectorization, even without the `-ffast-math` flag. *Hint:* The LLVM IR and assembly output from compiling `matmul.c` is substantially more complicated than what you have seen in previous examples. It can be hard, therefore, to identify the LLVM IR or assembly code for the matrix-multiplication routine in particular. One way to find the relevant LLVM IR or assembly output is to search the output file for the two calls to the timing code, such as `clock_gettime`, because the matrix-multiplication code of interest should appear between these calls. Another strategy is to use `perf record` and `perf report` to help search for the matrix-multiplication code. Because a large fraction of the running time of this program is spent in the matrix-multiplication code, this code should appear near the top of `perf`'s profile. When using this second strategy, be careful not to confuse the matrix-multiplication code you are optimizing with that used to check correctness.

Write-up 6: Compile the original `matmul` code and run it using `awsrun` to measure its original running time. Then, try to enable vectorization using `-ffast-math`, and examine the output of the vectorization report. Does the `matmul` code vectorize? Why or why not? Note that the vectorization report might contain a second entry for the loop in `matmul_base` if `clang` inlines the `matmul_base` function into its caller function, `main`.

Write-up 7: You can mandate that `clang` vectorize a particular loop using a pragma directive. For example, to require `clang` to vectorize the `k` loop in `matmul_base`, you can add the following pragma before the loop:

```
#pragma clang loop vectorize(enable) interleave(enable)
```

Add a pragma before the `k` loop to require vectorization of that loop. Verify that the vectorization report confirms that `clang` now vectorizes the loop. Run the resulting executable with `awsrun`. How does the performance of the program with the pragma compare to that of the original? From examining the LLVM IR or assembly output for this version of `matmul`, propose an explanation for the new performance you observed.

Write-up 8: (Optional) Remove the pragma added by the previous write-up, and now try to enable vectorization by reordering the loops in `matmul_base`. You should find an order of loops that allows `clang` to vectorize (without `-ffast-math`). What's the running time of this vectorized code, as measured with `awsrun`?

3.2 Data types and vectorization

In some situations, one can use lower-precision floating-point arithmetic and still produce acceptable results. Such an optimization can improve performance, not only by reducing the space required, but also by enabling vectorization to operate on more elements of input at a time.

Write-up 9: (Optional) Change the element type of the matrices from `double` to `float`. You can make this change by changing the `typedef` statement that defines the `el_t` type, which is the type of the matrices used in this matrix-multiplication code. How does this change affect the vectorization of the code? What's the running time of the new code, as measured with `awsrun`?

3.3 A simple outer-product base case

For matrix multiplication, we can use the vector hardware more intelligently than `clang` does. In this section, you'll implement a vectorized base case by hand, using compiler built-ins. This base case is a simplified version of that used in the matrix-multiplication case study in Lecture 1. For simplicity, we'll consider this base case for the problem of multiplying two $n \times n$ matrices `A` and `B`.

Although matrix multiplication is typically formulated using dot products between rows of `A` and columns of `B`, a more efficient base case can be developed by considering the computation of a $w \times v$ submatrix of `C` using *outer products* of w-height subcolumns of `A` and v-length subrows of `B`. In other words, consider the v elements $\langle c_{i,j}, c_{i,j+1}, \ldots, c_{i,j+v-1} \rangle$ in a row of a $w \times v$ submatrix of `C`. This row can be computed using the following formula on sets of v consecutive elements in rows of `B`:

$$\langle c_{i,j}, c_{i,j+1}, \ldots, c_{i,j+v-1} \rangle = \sum_{k=0}^{n-1} a_{i,k} \cdot \langle b_{k,j}, b_{k,j+1}, \ldots, b_{k,j+v-1} \rangle$$

This outer-product base case offers several features that make it efficient to compute using vector instructions. By choosing the dimensions of the submatrix carefully, the whole $w \times v$ submatrix of `C` can be stored in vector registers, and most of the computation can be performed directly on vector registers, without writing results back to memory. In addition, each product between $a_{i,k}$ and v consecutive elements in a row of `B` can be computed using elementwise products between vectors. By choosing v to equal the vector width, for example, each product can be performed by ***broadcasting*** the element $a_{i,k}$ to all entries of a vector register and then performing an elementwise product between that vector and a second vector register storing the v consecutive elements of `B`. Finally, each sum into a row of the `C` submatrix can be performed using an elementwise sum between vectors.

The GCC vector extension

The compiler's autovectorization capabilities struggle to figure out this outer-product base case, so we're going to implement it ourselves.

To simplify the task of implementing hand-vectorized code, `clang` supports the ***GCC vector extension*** to C. This vector extension provides an attribute for defining a ***vector type***, as follows:

```
typedef float vfloat_t __attribute__((__vector_size__(32)));
```

This type definition defines a new type, `vfloat_t`, which is a vector of `float`'s whose total size, indicated by the argument to the `__vector_size__` attribute, is 32 bytes. With this definition of a vector type, one can write C code that defines vector variables using standard C syntax. For example, the following code uses the above type definition to declare the variable `b_vec` as a vector of `float`'s and the variables `a_vec` and `c_vec` as arrays of 2 `vfloat_t`'s each:

```
vfloat_t b_vec;
vfloat_t a_vec[2], c_vec[2];
```

One can express elementwise vector operations using C's primitive operations — such as `+`, `-`, `*`, and so on — on variables of a vector type. The following code, for example, computes the elementwise product between `a_vec[0]` and `b_vec` and adds that product elementwise into `c_vec[0]`:

```
c_vec[0] += a_vec[0] * b_vec;
```

Individual elements of a vector-type variable can be accessed using standard C notation for indexing arrays. For example, the following code initializes the entries in `b_vec` with consecutive elements in an array `B`, starting at index `i`:

```
.LBB0_5:                                #   Parent Loop BB0_2 Depth=1
                                        #     Parent Loop BB0_3 Depth=2
                                        #       Parent Loop BB0_4 Depth=3
                                        # =>      This Inner Loop Header: Depth=4
        vmovaps %ymm3, %ymm4
        vmovaps %ymm2, %ymm5
        vmovaps %ymm1, %ymm6
        vmovups (%rdx), %ymm7
        vbroadcastss    (%rsi,%r11), %ymm3
        vfmadd213ps     %ymm4, %ymm7, %ymm3 # ymm3 = (ymm7 * ymm3) + ymm4
        vbroadcastss    (%r12,%r11), %ymm2
        vfmadd213ps     %ymm5, %ymm7, %ymm2 # ymm2 = (ymm7 * ymm2) + ymm5
        vbroadcastss    (%rax,%r11), %ymm1
        vfmadd213ps     %ymm6, %ymm7, %ymm1 # ymm1 = (ymm7 * ymm1) + ymm6
        vbroadcastss    (%rbx,%r11), %ymm4
        vfmadd231ps     %ymm4, %ymm7, %ymm0 # ymm0 = (ymm7 * ymm4) + ymm0
        addq    $4, %r11
        addq    %rcx, %rdx
        addq    $-1, %r8
        jne     .LBB0_5
```

Figure 19: Example assembly output for the innermost loop from compiling an implementation of the outer-product base case.

```
for (int e = 0; e < sizeof(vfloat_t)/sizeof(float); ++e)
  b_vec[e] = B[i + e];
```

From examining the LLVM IR or assembly for this code, you should find that `clang` compiles and optimizes this loop into a vector load from the address `&B[i]`. Similarly, you can broadcast the value of the `i`-th entry of an array `A` to each element in `a_vec[0]` as follows:

```
for (int e = 0; e < sizeof(vfloat_t)/sizeof(float); ++e)
  a_vec[0][e] = A[i];
```

You should find that `clang` compiles and optimizes this loop over the vector elements to replace it with a single vector broadcast instruction in assembly, such as `broadcast` or `vbroadcast`. You can find further documentation about the GCC vector extension at the following webpage: `https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html`.[1]

We can use the GCC vector extension to implement the outer-product base case by hand. Through careful coding, we can produce a matrix multiplication code with a highly efficient base case that

---

[1] You can also find documentation on the GCC vector extension here: `https://releases.llvm.org/9.0.0/tools/clang/docs/LanguageExtensions.html#vectors-and-extended-vectors`. This page includes particulars of `clang`'s support for the GCC vector extension, but mixes in discussion of other vector extensions, including the OpenCL, AltiVec, and NEON vector extensions, which can be confusing. For this exercise, the documentation in this handout and on the GCC webpage should suffice.

outperforms what `clang`'s autovectorization can produce. Figure 19 presents an example of the assembly code of the innermost loop of the base case that `clang` produces from an implementation of the outer-product base case using the GCC vector extension as described here. This implementation improves the running time of the matrix-multiplication code to approximately 0.1 seconds, as measured via `awsrun`.

Write-up 10: Modify the `matmul_base` function in `matmul.c` to implement the outer-product base case, using `clang`'s support for the GCC vector extension. You can modify the `matmul_base` liberally — such as by changing the loops in `matmul_base` or creating new functions in `matmul.c` and calling them from `matmul_base` — but your changes should be restricted to the `matmul_base` subroutine. Examine the LLVM IR and assembly to verify that `clang` produces vectorized code for your implementation of this base case. Run the compiled `matmul` executable and allow it to check that the optimized code correctly multiplies matrices. For bonus points, try to optimize your implementation of the base case to beat the performance of `clang`'s autovectorization. (But don't invest too much 6.172 time into this write-up, at the expense of your project!) What dimensions did you choose for the `C` submatrix computed by this outer-product base case, in order to use the vector registers efficiently? How did you choose those dimensions? How did you modify the loops in `matmul_base` to execute your base case efficiently? How did the performance of your final implementation compare to that of `clang`'s autovectorization? Commit and push your final optimized implementation of `matmul.c`.

Hint: To generate code that uses the fused multiply add instruction, `vfmadd`, compile the code with the `-ffast-math` flag.

4 Turn-in

When you've written up answers to all of the above questions, turn in your write-up by uploading it to Gradescope, and commit and push your code to your Git repository.