Software Performance Engineering February 2, 2025
Michigan State University CSE 491
Xuhao Chen Handout 6

Homework 3: Vectorization

Due: 10:00 P.M. (ET) on Tuesday, February 11, 2025

(Last Updated: February 1, 2025)

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 GitHub:

```
$ git clone git@github.com:CSE491-spring25/homework3_<your_netid>.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 `example1-0.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* <u>*short*</u> *(1–3 sentence) response or a code snippet (if requested).* **Please ensure that all the times you quote are obtained with `telerun`.**

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 example1-0.o
```

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

```
$ make ASSEMBLE=1 VECTORIZE=1 example1-0.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 `example1-0.ll`, which contains the optimized LLVM IR for the example. The vectorized operations in the LLVM IR output are those that operate on an LLVM vector type, such as `<16 x i8>` in `example1-0.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 `example1-0.s`, which contains the assembly code for this example.

Both the LLVM IR and assembly output show that `clang` uses ***multiversioning*** to vectorize the loop. Consider the LLVM IR, for example. In `example1-0.ll`, the function definition `@test` (corresponding to the function definition `test` in `example1-0.c`) contains multiple basic blocks. In the first basic block labeled `iter.check`, the code first checks if there is any ***aliasing*** between

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

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

Figure 3: First modification to `example1-0.c`, which uses the `restrict` keyword. Code can be found in `example1-1.c`.

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 which is the basic block `for.body`. If there is no aliasing, then a vectorized version of the loop is run which is the basic block `vector.body`.

If you further investigate the assembly code generated in `example1-0.s`, you will find blocks that correspond to the basic blocks in LLVM-IR; there will also be annotations with the LLVM-IR basic blocks labels to help you identify them.

Write-up 1: Compare the LLVM IR output and the assembly output for `example1-0.c`:

1. Paste the lines of assembly code that correspond to the basic block `iter.check` in LLVM-IR.
2. If you investigate both the assembly and the LLVM-IR code that corresponds to the aliasing check, you will find that they perform two comparisons. Why is that?
3. The assembly code that performs the aliasing check does two comparisons and then a conditional jump after each to the same label. What does this label correspond to: the vectorized or non-vectorized version of the loop? Provide evidence by pasting the line containing the conditional jump as well as some of the code getting executed at the label that the jump corresponds to. Which instructions and registers used make you think that this is the vectorized or non-vectorized version of the loop?
4. Going back to the aliasing check in the assembly code, the code performs two conditional jumps both to the same label corresponding to exactly one of the vectorized or non-vectorized version of the loop. Why is there no need for logic to branch to the other version?

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`

```
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 4: Second modification to `example1-0.c`, to instruct `clang` to assume a particular alignment on pointers. Code can be found in `example1-2.c`

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 3.

Compile the code in Figure 3 with `LLVMIR=1` to generate the LLVM IR in `example1-1.ll`.

```
$ make LLVMIR=1 VECTORIZE=1 example1-1.o
```

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 3 with `ASSEMBLE=1` should produce assembly code in `example1-1.s`.

```
$ make ASSEMBLE=1 VECTORIZE=1 example1-1.o
```

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.

`example1-2.c` is modified to use the `__builtin_assume_aligned` intrinsic as shown in Figure 4. Recompile `example1-2.c` to produce LLVM IR output in `example1-2.ll`.

```
$ make LLVMIR=1 VECTORIZE=1 example1-2.o
```

As the LLVM IR shows, the `align` attribute on the vector `load` and `store` operations matches the specified alignment of 16 bytes.

Compiling the code in `example1-2.c` with `ASSEMBLE=1` should produce assembly code in `example1-2.s`.

```
$ make ASSEMBLE=1 VECTORIZE=1 example1-2.o
```

Write-up 2: The optimized assembly code in `example1-2.s` is shorter than the previous version in `example1-1.s`. 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 `example1-2.c` with AVX2 instructions using the `AVX2=1` flag. Compile the assembly code for `example1-2.c` with AVX2 instructions using the following command:

```
$ make ASSEMBLE=1 VECTORIZE=1 AVX2=1 example1-2.o
```

You should see assembly output in `example1-2.s`. 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 vectorized code uses unaligned move instructions. Modify `example1-2.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 `example1-2.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 5. Examine the LLVM

```
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 5: Original C code in `example2-0.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 6: Modified C code for `example2-1.c`.

IR and assembly that `clang` compiles for `example2-0.c`. You can use similar commands to those described in Section 2.1:

```
$ make LLVMIR=1 VECTORIZE=1 example2-0.o
$ make ASSEMBLE=1 VECTORIZE=1 example2-0.o
```

Contrast the LLVM IR and assembly output from compiling `example2-0.c` to the output you get from compiling `example2-1.c` as shown in Figure 6. You should find that, compared to the original, the revised version in `example2-1.c` produces a tighter vectorized loop.

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

```
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 7: Original C code in `example3.c`.

2.3 Example 3

Consider `example3.c`, whose code is reproduced in Figure 7. 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.

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:

```
$ make clean; make VECTORIZE=1 AVX2=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`, `vaddsd`, and `vfmadd231sd` instructions operate on scalar double-precision 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 clean; make VECTORIZE=1 AVX2=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.

Another good way to quickly view the assembly and LLVMIR of `matmul_base` is to use the Compiler Explorer. To obtain the assembly for `matmul_base`, make sure to set the programming language in the Compiler Explorer to C, and set the compiler to x86-64 clang 18.1.0. Then, in the "Compiler options..." field, paste the compiler flags that get passed to `clang-spe` when you compile `matmul` locally. `make` will print out the `clang` command that it runs, so you can get the flags from there. Now, paste the `matmul_base` function along with associated code like the `typedef double el_t` above `matmul_base`. You will see the assembly for the code, which should automatically update as you type. You can also see how the lines correspond to the assembly through color-coding. To see LLVM IR, you can click the "Add new..." dropdown above the assembly output and select "LLVM IR". By using the Compiler Explorer, you will be able to iterate on the code much faster, but as a disclaimer, the assembly output may not exactly match the `clang-spe` output, so when citing assembly code in your response to write-ups, you should always use locally generated assembly with the `ASSEMBLY=1` flag.

Write-up 6: Compile the original `matmul` code and run it using `telerun` to measure its original running time. Then, recompile the code using `-ffast-math`, and examine the output of the vectorization report. Does the `matmul` code vectorize? Why or why not? Run the recompiled code with `telerun`, and discuss how the running time has changed. 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)
```

Recompile your code by running

```
$ make clean; make VECTORIZE=1 AVX2=1
```

Verify that the vectorization report confirms that `clang` now vectorizes the loop. Run the resulting executable with `telerun`. Discuss how the performance of the program with the pragma compares to that of the original code without any optimizations and the code compiled with `-ffast-math`. Propose an explanation for the new performance you observed by examining the LLVM IR or assembly output for this version of `matmul`.

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`. Which loop does `clang` now report as the vectorized loop? 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 `telerun`? How does it compare to your previous vectorized codes? Explain your numbers by investigating the LLVM-IR or assembly and see how the generated vector code now compare to the generated vector code from before.

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: 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 `telerun`?

3.3 Further optimization by reordering operations

We can compute matrix multiplication with a different ordering of operations that allows us to vectorize more intelligently than `clang` does. To do this, you will need to use compiler builtins and manually vectorize the matrix multiplication code.

But first, let us discuss how reordering operations can improve performance. When computing $C = A \cdot B$, our program currently iterates over each cell of C, computing them by looking up values in the corresponding row of A and the corresponding column of B. So, to calculate the first element of the first row of C (which we will call $C_{1,1}$ from here on), we need to multiply the first row of A with the first column of B and add all of the resulting values.

However, when we go to calculate the second element of the first row of C (which we will call $C_{1,2}$ from here on), we will again need to load the first row of A, though we will now be multiplying it with the second column of B. This raises the question: could we avoid loading the first row of A repeatedly as we calculate the first row of C?

As it turns out, it is possible with vectorization if we are willing to reorder our operations slightly.

Let's formally write the formula for $C_{1,1}$ and $C_{1,2}$:

$$C_{1,1} = \sum_{i=1}^{n} A_{1,i} \cdot B_{i,1}$$
$$C_{1,2} = \sum_{i=1}^{n} A_{1,i} \cdot B_{i,2}$$

Note that both $B_{n,1}$ and $B_{n,2}$ are multiplied by $A_{1,n}$. So, to avoid loading $A_{1,n}$ on two separate instances, what if we grouped $B_{n,1}$ and $B_{n,2}$ into a single vector and multiplied them in a single vector operation? Then, our computation will look like:

$$\begin{bmatrix} C_{1,1} & C_{1,2} \end{bmatrix} = \sum_{i=1}^{n} A_{1,n} \cdot \begin{bmatrix} B_{n,1} & B_{n,2} \end{bmatrix}$$

We can fit more than 2 floats in a single vector register. In fact, we can fit 8 floats. So, even more efficiently, we can compute 8 entries of C at a time:

$$\begin{bmatrix} C_{1,1} & C_{1,2} & \dots & C_{1,8} \end{bmatrix} = \sum_{i=1}^{n} A_{1,n} \cdot \begin{bmatrix} B_{n,1} & B_{n,2} & \dots & B_{n,8} \end{bmatrix}$$

This is a decent amount of vectorization, but we can still do better. To see how, let's look at the formula for the second row of C:

$$\begin{bmatrix} C_{2,1} & C_{2,2} & \dots & C_{2,8} \end{bmatrix} = \sum_{i=1}^{n} A_{2,n} \cdot \begin{bmatrix} B_{n,1} & B_{n,2} & \dots & B_{n,8} \end{bmatrix}$$

Notice how this computation uses the same set of vectors $\begin{bmatrix} B_{n,1} & B_{n,2} & \dots & B_{n,8} \end{bmatrix}$ as the first row! Thus, it might be a good idea to reuse $\begin{bmatrix} B_{n,1} & B_{n,2} & \dots & B_{n,8} \end{bmatrix}$ when adding into the second row after we already used it in the first row. The main problem with this approach is that, if we try to compute too many rows of C at once, we will eventually run out of vector registers to store C in. Once the compiler runs out of registers to store C in, it will try to use the stack instead, which is slow.

These observations alone should allow you to substantially optimize `matmul`, but the implementation may be trickier than you think. The next section will talk about how we can manually implement our optimization strategy.

The GCC vector extension

The compiler's autovectorization capabilities struggle to figure out the optimizations we just discussed, 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__(64)));
```

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 64 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`:

```
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. Indeed, with a simple implementation of the methods we have discussed, one may expect a running time of approximately 0.13 seconds, as measured via `telerun`. Another more optimized version achieves approximately 0.025 seconds, as measured via `telerun`.

Now, it's your turn. You will be optimizing `matmul` using the techniques described above. When optimizing, remember to examine the LLVM IR and assembly (perhaps through the Compiler Explorer) to verify that `clang` is producing the vectorized instructions that you expect. Run the compiled `matmul` executable and allow it to check that the optimized code correctly multiplies matrices.

Write-up 10: Manually vectorize `matmul` as discussed above by modifying the `matmul_base` function in `matmul.c`. 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 time into this write-up, at the expense of your project!) How did you make the most use out of all of the vector registers? 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`.

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.