Exercise
Parallel Code with LLVM
SIMDization

1 Use C/C++ extensions to write vector code

This exercise explores the LLVM vector instruction set by manually writing vector code using the gcc and clang vector extensions for C and C++. The primary goal of this exercise is to understand how to quickly generate vector IR from C code to serve as reference for LLVM-IR optimization that will be developed later.

1.1 Vector Addition

We begin with a simple scalar loop program, vector addition:

```
#define N (1024 * 1024 * 1024)

void vector_addition(float *A, float *B) {
  // We assume that N is a multiple of 4.
  for (long i = 0; i < N/4; i++)
    A[i] += B[i];
}
```

1. Translate the scalar program into LLVM-IR using the command:
   `clang -S -emit-llvm -O3 -fno-vectorize vector-addition.c -o vector-addition.ll`
2. Use the generic C/C++ vector extensions to vectorize the IR.
3. Generate LLVM-IR for your vectorized code using again the command:
   `clang -S -emit-llvm -O3 vector-addition.simd.c -o vector-addition.simd.ll`
4. Generate assembly code for both examples and compare the output.
5. Generate binaries and compare their performance.

6. Compile the scalar program without `-fno-vectorize` and compare generated LLVM-IR, assembly, and performance with the manually optimized program.

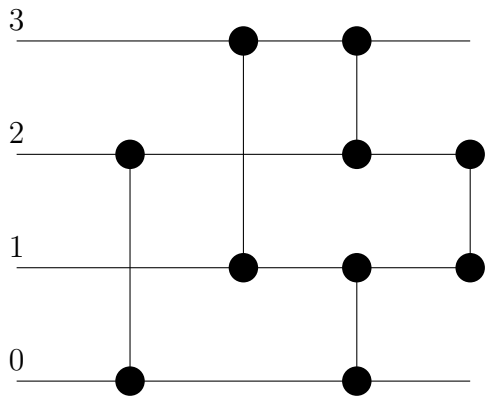
7. Use `-Rpass=loop-vectorize` to verify that the scalar loop has been automatically vectorized and use `-Rpass-missed=loop-vectorize` to verify that the manually vectorized loop has not been vectorized automatically.

8. Instead of just adding a vector B, add a vector B0, B1, B2, ..., B9. Try first with B0 - B5 and then with B0 - B9. Check again for each configuration if the loop is vectorized. Understand the reason why the loop is not vectorized and provide the relevant information to get it vectorized automatically. How is the additional information that you provided represented in LLVM-IR?

1.2 Sorting Networks

To sort a small set of numbers efficiently in registers, sorting networks can be used. Sorting networks sort a set of N input elements by repeatedly performing pairwise `swaps`, where the swap operation ensures that the corresponding element pair is sorted (e.g., using <=).

Sorting network: Batcher's Merge-Exchange for N=4



This sorting network uses 5 comparators grouped into 3 parallel operations:

(0, 2)	(1, 3)
(0, 1)	(2, 3)
(1, 2)	

1. Use C/C++ vector extensions to implement the sorting network above.
2. Analyze the IR generated by:
   `clang -O3 -std=c++1z -march=corei7-avx sorting-network.cpp -S -emit-llvm`
3. Analyze the assembly code generated by:
   `clang -O3 -std=c++1z -march=corei7-avx sorting-network.cpp -S`

Help:

- C/C++ vector extensions: `https://clang.llvm.org/docs/LanguageExtensions.html#vectors-and-extended-vectors`
- `__builtin_shufflevector`: `https://clang.llvm.org/docs/LanguageExtensions.html#langext-builtin-shufflevector`
- The `swap` operation can be implemented with element-wise min and max operations. Use for these the following intrinsics:

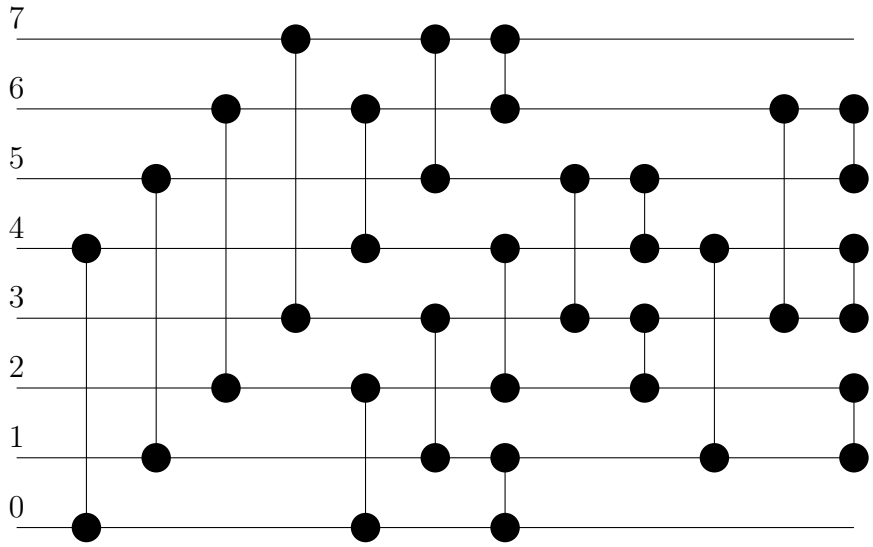
```
double4 min(double4 A, double4 B) {
  return __builtin_ia32_minpd256(A, B);
}

double4 max(double4 A, double4 B) {
  return __builtin_ia32_maxpd256(A, B);
}

double2 min(double2 A, double2 B) {
  return __builtin_ia32_minpd(A, B);
}

double2 max(double2 A, double2 B) {
  return __builtin_ia32_maxpd(A, B);
}
```

Bonus: Batcher's Merge-Exchange for N=8



This sorting network uses 5 comparators grouped into 6 parallel operations:

(0,4)	(1,5)	(2,6)	(3,7)
(0,2)	(1,3)	(4,6)	(5,7)
(2,4)	(3,5)	(0,1)	(6,7)
(2,3)	(4,5)		
(1,4)	(3,6)		
(1,2)	(3,4)	(5,6)	

2 A simple loop vectorizer

Write a simple loop vectorizer that is capable of vectorizing the following code automatically:

```
void vector_addition(float *A, float *B) {
  for (long i = 0; i < 1024; i++)
```

```
    A[i] += B[i];
}
```

The goal of this exercise is to learn about canonicalizations which LLVM passes and analysis expect, to understand how to use LoopInfo and ScalarEvolution, and to understand how to analyse the LLVM-IR.

1. Translate your example to LLVM-IR and canonicalize the IR: `-mem2reg -instnamer -loop-rotate`.

2. Write a simple pass that lists all innermost loops, the basic blocks they contain, as well as the number of times each loop is executed (use scalar evolution for the latter).

3. Compute for each memory access the stride (distance) with which subsequent loop iterations of the same load/store instruction access memory.

4. Extend this pass to only match loops that are canonical. A loop is (very) canonical if:

   - The loop is in loop simplify form
   - Scalar evolution can compute the backedge taken count
   - Only two basic blocks: a loop latch (backedge) and a loop header exist
   - All instructions but the induction variable increment and the exit comparison are in the loop header (and are consequently executed before the loop is exited in a given iteration).
   - The induction variable starts at zero, is incremented by one and is compared against the upper loop bound by using a signed-less-than comparison..
   - All memory accesses are stride one.
   - The body only contains load, store, fadd, and getelementptr instructions.
   - The number of loop iterations is a multiple of the vectorization factor (e.g., 4)

5. Adjust the loop to increment in strides of the vector factor and replicate each instruction 'vector-factor' times. Use a local map to update the operands of each instruction to access the right version of each instruction.

6. Vectorize: use a vector load whenever you encounter a stride-one load, use a vector add whenever both operands are already available as vector values, and use a vector store whenever the load to be stored is available as vector value and the store is stride one.

7. Run your code on the above test case and show that the output is vectorized.

Bonus

1. Support stride-zero and stride-X (scatter, gather) loads and stores.

2. Support mixed vector and non-vector code.

3. Add a simple cost model using TargetTransformInfo

4. Collect the minimal and maximal memory access offsets using Scalar Evolution and derive run-time alias checks for your arrays.