Project 6: Superword Level Parallelism

ECE 566 Fall 2015

This project is optional for ECE 466

~~Due: December 4, 2015~~
Due: December 6, 2015
You are encouraged to comment directly on this document!

Objectives

- Identify SIMD vectorization opportunities in LLVM IR.
- Gain a basic understanding of the principles of vectorization.
- Bonus:
    - Gain experience and a deeper understanding of the vector instruction support in LLVM by generating LLVM IR for vector operations.
    - Evaluate your implementation and its effectiveness on a set of applications.

Description

In class, we've discussed parallelism in the context of scheduling for VLIW processors. The compiler selects independent instructions to execute in the same cycle, subject to the availability of resources. Another form of parallelism that's available in most processors today is Vector-Level Parallelism, also known as Single Instruction Multiple Data (SIMD).

A vector is like an array--it contains multiple elements of the same type in a single register. Vector instructions operate on all the elements in the array simultaneously. Vectors may appear in multiple lengths and may support integer, float, or double operations.

To exploit vector level parallelism, the compiler should choose which values to pack into registers and which operations should be vectorized. There are two main approaches to this problem. The classical school of thought is to analyze inner-loops for operations on arrays, like this:

```
for (i=0; i<100; i++)
   A[i] = B[i]*4 + C[i]
```

Ideally, the compiler will recognize that if the loop is unrolled, then we can form a vector out of adjacent memory references, leading to pseudo-code like this:

```
for (i=0; i<100; i+=4)
   A[i:i+3] = B[i:i+3]*4 + C[i:i+3]
```

Note, A[i:i+3] means the elements in A from A[i] to A[i+3]. Now, we can generate vector code. This approach works well if we can prove that there is no overlap between arrays A, B, and C in memory and if we can prove that there are no data dependences through memory between these statements. For this loop, that is easy to do, but for many loops of a similar form it is nearly impossible.

A more recent approach that has gained some traction is called Superword Level Parallelism. In this approach, code is analyzed for isomorphic instructions--instructions with the same opcode that operate on the same types, like this:

```
%mul11 = fmul double %14, %13
%mul17 = fmul double %10, %9
```

We want to convert it into a vector operation like this:

```
// %v1 = <%14,%10>
// %v2 = <%13,%9>
%v.mul11 = fmul <2,double> %v1, %v2
```

As long as we can find isomorphic instructions, we can group them into vector operations. Theoretically, we could just look for these opportunities and vectorize any instructions that are isomorphic. However, this would not be efficient and would lead to overheads. One reason is that the operands for vector instructions need to be *packed* into vectors, and this requires extra instructions. Another reason is that the original instruction's result may be used later in the program, which would require unpacking the vector so it could be used again. As a result we may end up with extra instructions like this:

```
%v.ie6 = insertelement <2 x double> zeroinitializer, double %13, i32 0
%v.ie7 = insertelement <2 x double> %v.ie6, double %9, i32 1
%v.fmul8 = fmul <2 x double> %v.ie5, %v.ie7
%v.fadd = fadd <2 x double> %v.fmul8, %v.fmul
%v.ev12 = extractelement <2 x double> %v.fadd, i32 0
%v.ev13 = extractelement <2 x double> %v.fadd, i32 1
```

The insertelement instructions pack the vectors, and the extractelement instructions unpack the vectors. We should try to collect a chain of dependent vector operations so that intermediate results need not be unpacked until the very end.

Ideally, we want to find long sequences of vector operations with relatively little packing and unpacking overhead. We also want to find very wide vector operations so that we can exploit the full width of the vector unit on the target processor. Also, we want to find the best sequences of all candidates sequences. However, we can't achieve all of these goals in a simple project.

To limit the set of alternatives, we'll look for two kinds of sequences of vector operations: those originating at loads and those originating at stores--since they don't need as much packing/unpacking support. We'll also limit the width of the vector to only two operations.

Specification

For every basic block, you will run the SuperWordParallelism pseudocode below. It will form two isomorphic instructions that will serve as a seed for a longer dependent chain of vector operations. It calls collectIsomorphicInsn to verify that the instructions are suitable for vectorization and to track backward along their use-def chains to find suitable instructions to vectorize.

```
SuperWordParallelism(BasicBlock BB):
    foreach instruction, I, in BB:
        if I is a store and it is not already vectorized:
            for each store, J, in BB, such that J comes before I:
                if AdjacentStores(I,J) and Isomorphic(I,J):
                    list = collectIsomorphicInstructions({},I,J)
                    if size(list) >= 2 instructions:
                        keep candidate list
                        break (stop considering I)
        if I is a load and it is not already vectorized:
            for each load, J, in BB, such that J follows I:
                if AdjacentLoads(I,J) and Isomorphic(I,J):
                    list = collectIsomorphicInstructions({},I,J)
                    if size(list) >= 2 instructions:
                        keep candidate list
                        break (stop considering I)


        if found a suitable list and it is transformable:
            vectorize(list);  // Optional
            update stats;   // Required
            // after transformation, you may need to repair your iterator

Isomorphic(Instruction I, Instruction J):
    if Opcode(I) != Opcode(J):
        return false
    if TypeOf(I)!=TypeOf(J):
        return false
    if NumOperands(I) != NumOperands(J):
        return false
    foreach i in range 0:NumOperands(I):
        if TypeOf( op(I,i) ) != TypeOf( op(J,i) ):
            return false
    return true
```

AdjacentLoads(Instruction I, Instruction J):

```
    gep1 = LLVMGetOperand(I,0)
    gep2 = LLVMGetOperand(J,0)
    if !LLVMIsAGetElementPtrInst(gep1) or !LLVMIsAGetElementPtrInst(gep2)
        return false
    gep1 and gep2 must have the same base address pointer (operand 0), and their last
operand must be constant and have a difference of 1
    if they do, return true
    else return false
```

The AdjacentStores routine will work similarly.

The following function builds a list of instructions that can be vectorized. First, it validates that it's arguments are candidates for vectorization, and then it checks each operand and tries to add them to the list. Once it has recursively added as many instructions as it can, it gives up and returns the result to the caller.

collectIsomorphicInstructions(List L, Instruction I, Instruction J):

```
    if !shouldVectorize(I,J):
        return NULL
    if I or J is already in L:
        return L
    append (I,J) into L
    foreach i in range 0:NumOperands(I):
        if Isomorphic(op(I,i), op(J,i)):
            collectIsomorphicInstructions(L,op(I,i),op(J,i))
```

Even though I and J are isomorphic, they may be instructions that should not be vectorized for a variety of reasons. shouldVectorize makes sure I and J don't fall into any of the bad cases.

shouldVectorize(Instruction I, Instruction J):

```
    // since we already know they are isomorphic, we can just check I for most of checks
    if TypeOf(I) is not an integer, float, or pointer kind:
        return false
    if I and J are not in the same basic block:
        return false
    if I is a terminator:
        return false
```

```
if I is a volatile load or a volatile store:
    return false
if I is a PHI, Call, Atomic*,ICmp, FCmp, Extract*,Insert*,AddrSpaceCast:
    return false
if I and J are loads  and they do not load adjacent memory locations:
    return false
if I and J are stores and they do not store adjacent memory locations:
    return false
// there may be some other cases that will matter if you try to transform the code
// once we reach the end, we can vectorize I and J
return true
```

Transformation (Bonus)

Once we verify that the instructions can be vectorized, we loop through them and perform the necessary transformation. We know that each pair gets replaced with a single vector operation. We must transform the code so that the first operand of I is placed in a vector register with the first operand of J. Likewise for each additional operand. Then, we can build an instruction in the usual way, using the builder, the only difference is that it produces a vector result.

For operands that are inputs to L, i.e. not produced by instructions in L, we have to pack the vector using insertelement instructions. For any uses of I and J outside of L, we have to extract the value from the result of the new instruction using extractelement instructions.

In the code below, vmap is a value map data structure that associates each instruction/register in the original code with a new instruction/register in the vectorized code. As we make new instructions, we update the map to reflect that the result of that instruction is now available as a vector operand. This makes generating later instructions easier as we can just look up their operands in the vmap.

vectorize(List L):

```
    vmap = {} // maps original values to vector values
    for each pair (I,J) in L in dominance order:
        ops[] = {}
        for i in range 0:NumOperands(I):
            if vmap[Op(I,i)] is not found:
                ops[i] = packVector(op(I,i),op(J,i))
                vmap[op(I,i)] = ops[i]
                vmap[op(J,i)] = ops[i]
            else:
                ops[i] = vmap[op(I)]
        // Position builder just before I or J, pick the one later in BB
        newInsn = Build(opcode(I),ops)
        vmap[I] = newInsn
        vmap[J] = newInsn
    for each pair (I,J) in L:
        if I has uses:
            // Reposition builder
            ev = BuildExtractElement(vmap[I],0) // index 0
            LLVMReplaceAllUsesWith(I,ev)
            LLVMInstructionEraseFromParent(I)
        if J has uses:
            // Reposition builder
            ev = LLVMBuildExtractElement(vmap[I],1) // index 1
            LLVMReplaceAllUsesWith(I,ev)
            LLVMInstructionEraseFromParent(I)
    Remove any dead extractelements we inserted
```

Stuff to Print Out

1. Your pass should print a histogram of the list sizes that were vectorized for each module analyzed. Something like this (you need not match the exact format):

```
SLP Results:
   Size: Count
      2: 3
      3: 1
      4: 1
     >5: 1
   Store-chain: 5
   Load-chain: 1
```

You can merge all lists greater than 5 instructions long into the last entry for 5 instructions.

2. Also, print each list of instructions you select for vectorization. If you implement the transformation, then only print the sequences that were legally transformed.

Infrastructure Details

1. Provide an implementation for the optimizations described earlier. A starting point is provided for you in the projects directory.
    a. The code you implement should be added to the LICM library, not the p5 tool, so that it can be re-used in later projects.
    b. Your code should be added to `projects/lib/SLP/SLP_C.c` for C or `projects/lib/SLP/SLP_Cpp.cpp` for C++. A stub version of the entry point

to the optimization function has already been implemented for you in each file..

   c. You may not change the name of the `SLP[_C/_Cpp]` function.

2. A tool has already been implemented that calls the library code and links against it. You may need to update your repository to get the latest version of code. Then rebuild everything:
   a. `cd path/to/ECE566Projects`
   b. `git pull`
      i. If this command fails, it's because you have modified files. You can either commit them or stash them (but not both). A commit will keep your changes in the local directory, but ***stash will remove your changes*** ***and save them*** elsewhere. Pick the best one for your case:
         1. `git commit -a -m"some changes I made blah blah blah"`
            Or...
         2. `git stash`

         Now, go back and re-execute `git pull`.
   c. `cd path/to/ECE566Projects/projects/build`

   d. Note: If your build directory is already configured for the language you want, then you can skip this step and go straight to building (step e). Otherwise, follow the instructions for the language you prefer. For C, remove the build directory and re-run the cmake configure command inside the build directory to choose C (and I think the default language is C++ for p2):
      i. `export LLVM_DIR=/usr/local/llvm/3.6.2/install-debug/share/llvm/cmake`
      ii. `cmake` **`-DUSE_C=1`** `-DLLVM_CMAKE_DIR=/usr/local/llvm/3.6.2/install-debug/share/llvm/cmake ..`

      For C++, remove the previous build directory and re-run the cmake configure command inside the build directory to choose C++ (I think for C++ you can skip this part):
      iii. `export LLVM_DIR=/usr/local/llvm/3.6.2/install-debug/share/llvm/cmake`
      iv. `cmake` **`-DUSE_CPP=1`** `-DLLVM_CMAKE_DIR=/usr/local/llvm/3.6.2/install-debug/share/llvm/cmake ..`

   e. `cmake --build .`

3. Make a new p6-test directory in the same directory as wolfbench. Then, you can configure the project as shown below.
   a. `../wolfbench/configure` **`--enable-customtool`**``=`cd ../projects/build/tools/p6/; pwd`/p6``

b. Then, build the test code:
    i. `make all test`
    ii. `../wolfbench/timing.py `find . -name '*.time'``
c. If you want the output to be less verbose, run it this way:
    i. `make -s all test`
d. To validate that your optimized code produces the correct output, use the compare rule:
    i. `make compare`
e. If you encounter a bug, you can run your tool in a debugger this way:
    i. `make clean`
    ii. `make DEBUG=1`

    This will launch gdb on your tool with one of the input files. You can set breakpoints directly in your SLP implementation.

Collecting Results

As in previous projects, you should use the make commands to test your project:

make CUSTOMFLAGS=”-summary” test
make compare

Or, if you want to run without SLP

make EXTRA_SUFFIX=.no-slp CUSTOMFLAGS=”-summary -no-slp”

I recommend you setting OPTFLAGS to ensure a large amount of optimization happens before your SLP pass:

make OPTFLAGS=”-sroa -early-cse -gvn -loop-rotate -licm -loop-unroll “ EXTRA_SUFFIX=.no-slp CUSTOMFLAGS=”-summary -no-slp”

And, it never hurts to add your own stats and dump them to the screen or your summary report.

The Original Paper

A full description of the original SLP algorithm can be found in this paper:

Samuel Larsen and Saman Amarasinghe. *Exploiting superword level parallelism with multimedia instruction sets.* In Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00). [link]

Getting Help

History shows that my specs are sometimes incomplete or incorrect. Therefore, please start early. If you run into problems, please post a question with a relevant tag on Piazza.

Grading

ECE 566 students must work individually, but ECE 466 students are allowed and *encouraged* to form groups of two. Only one student needs to submit in ECE 466, but both names must appear in the submission document and in the comments of all your code.

Uploading instructions: Upload your SLP_C.c or SLP_Cpp.cpp file.

However, if you modified other files, then please upload an archive of your entire projects folder, and add a note to your submission in the Notes field indicating which language you used. We will test your code using the test cases provided in wolfbench and with some secret cases we did not provide.

The assignment is out of 100 points total. If you make no attempt and submit the provided code without meaningful changes (i.e. white space and comments do not count), you earn 10 points. Otherwise, assigned as follows:

ECE 566
10 points: Compiles properly with no warnings or errors
10 points: Code is well commented and written in a professional coding style
30 points: Meets or exceeds all specifications
35 points: Fraction of tests that pass
15 points: Fraction of secret tests that pass (these may overlap with provided tests)

+30 points: Bonus. To earn these points, you must transform the code to use vector operations

and it must generate correct code.

Bonus points will be distributed to your other projects (or homework grades) if you earn more than 100. This will be graded for partial credit.

ECE 466 (for Bonus points only)

30 points: Fraction of provided tests that pass (identify vectorization candidates)
30 points: Full vectorization transformation.

Bonus points will be distributed to your other projects (or homework grades). This will be graded for partial credit.