Project 6: Superword Level Parallelism

ECE 466/566 Fall 2014

Due: December 3, 2014

*You are encouraged to comment directly on this document!*

# Objectives

* Implement a simple vectorizing transformation from a high-level specification.
* 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.

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. This result is not ideal. So, 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. However, I also want to design a project that’s doable in a two week period, so we will make some simplifications that almost certainly guarantee poor performance but that will allow you to learn about vectorization.

## Specification

The requirements for 466 and 566 differ, so read carefully to identify the components you are required to complete:

For every basic block, you will run the SuperWordParallelism pseudocode below. It will for 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):**

repeat until no changes and at most 3 times:

foreach instruction, I, in BB:

foreach instruction J, in BB, such that J comes before I:

if Isomorphic(I,J):

list = collectIsomorphicInstructions({},I,J)

if size(list) >= 2 instructions && calcScore(list) is best:

keep list

if found a suitable list and it is transformable:

vectorize(list) // ECE 566 only

update stats;

break to outer repeat loop;

**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

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

insert (I,J) into L in dominance order

foreach i in range 0:NumOperands(I):

if Isomorphic(op(I,i), op(J,i)):

append collectIsomorphicInstructions(L,op(I,i),op(J,i)) to L

return L

Even though I and J are isomorphic, they may be instructions that should not be vectorized for a variety of reasons. shouldVectorize checks through the list to make 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 is a load or a store that comes from an alloca that holds the address of an integer, float, or double:

return true

else

return false

if I is dependent on J: // you must consider full backward slice of I within BB

return false

// otherwise, we can vectorize I and J

return true

Finally, once we have a candidate list for vectorization, compute a score that indicates its profitability. The score roughly computes the net change in instruction count with more weight given to removing floating point instructions.

**calcScore(List L):**

// the lower the better!

score = 0

foreach pair (I,J) in L:

if TypeOf(I) is floating point kind:

score -= 4 // replace two ex

pensive instructions with 1 vector insn

else:

score -= 1 // replace two cheap instructions with 1 vector insn

if I is ever used outside of L:

score++ // add an extractelement

if J is ever used outside of L:

score++ // add an extractelement

for each operand,op, in I:

if op is not defined by an instruction in L:

score++ // need an insertelement

for each operand,op, in J:

if op is not defined by an instruction in L:

score++ // need an insertelement

return score

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): // 566 only**

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] = newInsns

vmap[J] = newInsns

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

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 vectorize before they are transformed.

# Infrastructure Details

1. Provide an implementation for the optimizations described earlier. A starting point is provided for you in the projects directory.
   1. The code you implement should be added to the SLP library, not the p6 tool, so that it can be re-used in later projects.
   2. 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..
   3. 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. To get the working tool and library stub, do the following:
   1. If you already have a working projects directory, you either to commit your current work or stash it. To save it as a commit, do this:
      1. git commit -a -m”save current work”
      2. git pull

To stash:

* + 1. git stash
    2. git pull
    3. git apply stash

From the projects directory:

* + 1. make && make install
  1. If you are checking out a clean projects directory:
     1. cd ECE566Projects/projects
     2. git pull
     3. On VCL or Linux: *(on your linux install, remember to adjust the paths below)*
        1. ./configure --disable-optimized --with-llvmsrc=/usr/ece566/llvm-3.4 --with-llvmobj=/usr/ece566/llvm-build --prefix=`pwd`/install
     4. On a Mac:
        1. ./configure --disable-optimized --with-llvmsrc=`cd ..; pwd`/llvm-3.4 --with-llvmobj=`cd ..; pwd`/llvm-build --prefix=`pwd`/install --target=x86\_64
     5. make && make install
  2. *For faster compile time*: If you wish to disable building earlier projects, then remove all other directory names from the DIRS variable in projects/tools/Makefile other than the one you are working on.
  3. **For C++**. Both C and C++ implementations are called from the same tool in this project. But, it is configured for C by default. **To use C++, you must do this:**
     1. Open projects/tools/p5/main.cpp with a text editor
     2. Near the top of the file, comment out: #define UseC

1. Your project will be tested using the wolfbench repository configured using the tool you implement. Configure your testing directory as follows:
   1. In the parent directory for your wolfbench directory, make a testing directory:
      1. mkdir p6-tests; cd p6-tests
      2. Assuming that wolfbench, p6-tests, and projects are all in the same parent directory, you can configure the project like this:
         1. For VCL:
            1. ../wolfbench/configure **--enable-customtool**=`cd ../projects/install/bin; pwd`/**p6**
         2. For a Mac (make sure the ece566 install path is found first in the PATH variable before the host version of clang):

../wolfbench/configure --enable-customtool=`cd ../projects/install/bin; pwd`/**p6**

* + 1. Then, build the test code:
       1. make all && make test
       2. ../wolfbench/timing.py `find . -name ‘\*.time’`
       3. make compare

# 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”

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](http://groups.csail.mit.edu/cag/slp/SLP-PLDI-2000.pdf)]

Do not use the algorithm described in the paper without permission from the instructor.

**Uploading instructions:** Make a backup of your source tree.

cp –R ./projects/ project6

Then, zip up the folder using either zip or tar+gzip to create a single compressed archive.

tar czf project6.tgz ./project6

Upload the archive in Moodle on Project 6 assignment page. Also, please 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.

**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)

+20 points: Bonus points for implementing the full vectorization functionality. These points will be distributed to your other projects (or homework grades) if you earn more than 100. This will be graded for partial credit. Include a note indicating you implemented the extra support. If it doesn’t fully work, submit your code with extra work disabled.