Minotaur:
A SIMD-Oriented Synthesizing Superoptimizer

Zhengyang Liu, John Regehr (University of Utah)
On one hand: Interesting hardware offering very high peak performance

On the other: Applications and high-level languages and that want to exploit that performance

Domain Specific Lang. (Halide, Taichi)

SLP Vectorizers

Auto-Vectorizers

Intel ISPC

Video Codecs

Intel DPC++
Compilers such as GCC, LLVM and ISPC support **IR-level Vector Operations**

- Platform-independent and platform-dependent instructions, intrinsics, data movement instructions...
- Exposing vector instructions and intrinsics to middle-end optimizations

Compiler Developers write **rules** for optimizing vector operations

- Requires experts' time
- There are always rules missing, and therefore optimizations get missed
Vector Optimizations in LLVM’s Middle-End

Instruction Combiner (InstCombine) pass
- A large collection of peephole optimizations, originally for scalar operations
- They also work on element-wise vector operations

Vector Combiner pass
- Handles vector operations such as shuffle that aren’t element-wise
- Smaller and less mature than InstCombine
Compute the difference between adjacent pairs of array values:

```c
for (int i = 0; i < n; i += 2) {
    t[i] = s[i] - s[i + 1];
}
```
...%26 = shufflevector <8 x i16> %s, poison, < 0, 2, 4, 6>%27 = shufflevector <8 x i16> %s, poison, < 1, 3, 5, 7>%28 = zext <4 x i16> %26 to <4 x i32>%29 = zext <4 x i16> %27 to <4 x i32>%30 = sub nsw <4 x i32> %28, %29

| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

Default code from LLVM... can we do better?
pmaddwd, multiply and add packed integers

input 1:


input 2:


output:


but what do we do with it?
Input 1 is our data:

\[
\begin{array}{cccccccc}
\end{array}
\]

Input 2 is a vector of constants:

\[
\begin{array}{cccccccc}
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\
\end{array}
\]

Output:

\[
\begin{array}{cccc}
\end{array}
\]

\[
%30 = \text{llvm.x86.sse2.pmadd.wd}(\%s, \langle 1, \ -1, \ \ldots, \rangle)
\]
Minotaur:
A SIMD-Oriented Synthesizing Superoptimizer

Our goal: Automatically infer integer vector optimizations

Not a goal: Loop Vectorization
Synthesizing Vector Optimizations

- For each SSA value, extract how it is computed using a program slicer
- **Enumerate** rewrites for each extracted fragment
- Use the Alive2 **refinement checker** to discard incorrect rewrites
- Use a **cost model** to find the best correct rewrite
- **Cache** this rewrite for later reuse
Slicing LLVM Functions

- For each SSA value in the original program
  - Recursively walk the SSA graph backwards
  - Control flow instructions and memory operations require special care
    - Avoid extracting loops
    - Preserve pointer dependencies
  - Unsupported instructions become free inputs
- Each extracted program fragment is a Left Hand Side (LHS)
Enumerating Rewrites

We’re searching for a Right Hand Side (RHS) that is a cheaper implementation of the LHS

The RHS might be...

- A constant
- A compatible value already available somewhere higher in the LHS
- A new instruction
- A DAG of new instructions

We enumerate these vaguely in order of increasing complexity
Enumerating Rewrites

%a = shufflevector <8 x i16> %s, poison, <4 x i32> <0, 2, 4, 6>
%b = shufflevector <8 x i16> %s, poison, <4 x i32> <1, 3, 5, 7>
%c = zext <4 x i16> %a to <4 x i32>
%d = zext <4 x i16> %b to <4 x i32>
%e = shl <4 x i32> %d, <i32 16, i32 16, i32 16, i32 16>

%LHS = add <4 x i32> %c, %e

%RHS = %e
%RHS = or %p, %e
%RHS = sub %s, const
%RHS = add %a, %b
%RHS = llvm.x86.avx2.pshuf.b (%s, %b)
How Does Enumeration Scale?

It scales very poorly, of course!

Exponential in the number of synthesized instructions

Typically...
- 50 - 150 rewrites for 1 new instruction
- 100 - 1500 rewrites for 2 new instructions
- Etc.

However, most peephole optimizations in existing compilers only create 1-2 new instruction
Checking Refinement using Alive2

- Alive2 was created in order to find the bugs in LLVM

- Given a pair of LLVM functions with the same signature, Alive2 proves that the target refines or does not refine the source

- In Minotaur, we instead use Alive2 to check if a generated rewrite is valid
  - Compiler optimizations must be sound!
Candidate rewrites often contain symbolic constants

\[
%r = \text{mul } %a, \langle 8, 16, 32, 64 \rangle \quad \Rightarrow \quad %r = \text{shl } %a, C
\]

- We cannot possibly enumerate all possible values for C

- We require a synthesis strategy better than enumeration
Synthesizing Constants on the RHS

- Large constants are difficult to synthesize and we spent a lot of time looking at different strategies for doing this
- ... and ended up using the simplest: An exists-forall solver query
  - “Does there exist a constant C that works for all inputs x to the RHS”

$$\exists C. \forall x. LHS(x) \Rightarrow RHS(x, C)$$

- Several examples of synthesized constants on the RHS in this talk
Semantics for SIMD instructions

To make Minotaur work, we added semantics for SIMD instructions to Alive2

- All platform-independent, non-memory vector operations, including trivial arithmetics on vectors (add, sub, ...), insertelement, shufflevector, ...
- Major platform-independent memory operations (load, store, gather, ...)
- 165 Intel X86 integer non-memory vector intrinsics

About 2500 lines of code implementing new semantics
Cost Model

- There are often many valid RHSs that all refine a given LHS
- Minotaur wants the best one -- so we need an accurate cost model
- Alas, predicting the execution cost of LLVM IR is difficult
- Solution: Compile IR to object code, then invoke the LLVM-MCA
Why do we need llvm-mca?

$$\%0 = \text{shufflevector} \ <32 \times i8> \ %a, \ <32 \times i8> \ %b,$$
$$\ <64 \times i32> \ <0, 32, 1, 33, 2, 34, 3, 35, 4, 36, 5, 37, 6, \ldots>$$

1 LLVM instruction
9 X86 instructions, 12 uOps

$$\%zext = \text{zext} \ <32 \times i8> \ %a \ \text{to} \ <32 \times i16>$$
$$\%zext2 = \text{zext} \ <32 \times i8> \ %b \ \text{to} \ <32 \times i16>$$
$$\%shl = \text{shl} \ <32 \times i16> \ %zext1, \ <i16 \ 8, \ i16 \ 8, \ i16 \ 8, \ldots>$$
$$\%or = \text{or} \ <32 \times i16> \ %2, \ %zext.i$$
A Few Examples
Optimization: eliminate unnecessary round trip

%0 = zext <16 x i8> %x to <16 x i16>
%1 = zext <16 x i8> %y to <16 x i16>
%2 = call @llvm.x86.avx2.pavg.w(<16 x i16> %0, <16 x i16> %1)
%3 = trunc <16 x i16> %2 to <16 x i8>
ret <16 x i8> %3

⇒

%0 = call @llvm.x86.sse2.pavg.b(<16 x i8> %x, <16 x i8> %y)
ret <16 x i8> %0
Recognizing an Open-Coded Popcount

```
%1 = lshr i64 %0, 1
%2 = and i64 %1, 0x5555555555555555
%3 = sub i64 %0, %2
%4 = lshr i64 %3, 2
%5 = and i64 %3, 0x3333333333333333
%6 = and i64 %4, 0x3333333333333333
%7 = add nuw nsw i64 %6, %5
%8 = lshr i64 %7, 4
%9 = add nuw nsw i64 %8, %7
%10 = and i64 %9, 0xf0f0f0f0f0f0f0f
ret i64 %10
```

from 19 uOps to 13 uOps

```
%1 = bitcast i64 %0 to <8 x i8>
%2 = call @llvm.ctpop(<8 x i8> %1)
%3 = bitcast <8 x i8> %2 to i64
ret i64 %3
```

from libgmp
Speedups in libYUV Image Conversion Library Speedups (551 kernels), geomean=1.015x

9,208 LHSs, 357 optzns (3.9%)
Why Do Some Programs Slow Down?

We’ve not looked too closely at these yet, but...

- llvm-mca is far from 100% accurate
- Sometimes a Minotaur optimization will interact poorly with other parts of the optimization pipeline
16,831 LHSs, 431 optzns (2.6%)
SPEC CPU'17 CINT Speedups

102,879 LHSs, 1,978 optzns (1.9%)
Future work

- Automatically lowering combining rules to LLVM VectorCombine code
- Scale to synthesizing 3-5 instructions
- Support optimization of entire loops
- Support vectors of floating point values
  - Z3 is the bottleneck
  - There may be shortcuts we can take that avoid modeling IEEE floats in their full glory
Thank you!

Minotaur is in active development, and the code is open-sourced at:

github.com/minotaur-toolkit/minotaur