Autovectorization with LLVM

Hal Finkel

April 12, 2012

The LLVM Compiler Infrastructure 2012 European Conference

1. Introduction

2. Basic-Block Autovectorization
   - Algorithm
   - Parameters
   - Benchmark Results
   - Future Directions

3. Conclusion

Why Vectorization?

Taking full advantage of modern CPU cores requires making use of their (SIMD) vector instruction sets:

- MMX, SSE*, 3DNow, AVX (i686/x86_64)
- AltiVec, VSX (PowerPC)
- NEON (ARM)
- VIS (SPARC)
- And many others.

And what can these buy you?

- Speed!
- Energy Efficiency
- Smaller Code

Why Autovectorization?

Turning scalar code into vector code sometimes requires significant ingenuity, but like many other compilation tasks, is often formulaic. A compiler can reasonably be expected to handle the formulaic cases.

What's formulaic? Loops:

```
for (int i = 0; i < N; ++i)
a[i] = b[i] + c[i]*d[i];
```

Independent Combinable Operations:

```
a = b + c*d;
e = f + g*h;
...
```

Vector Operations in LLVM

LLVM has long supported an extensive set of vector data types and operations, has support for generating vector instructions in several backends, and contains generic lowering and scalarization code to handle code generation for operations without native support.

Some example LLVM IR vector operations:

```
%mul8 = load <2 x double>* %addr, align 8
%mul11 = fmul <2 x double> %mul8, %add10
%add12 = fadd <2 x double> %add7, %mul11
%vaddr = bitcast double* %addr2 to <2 x double>*
store <2 x double> %add12, <2 x double>* %vaddr, align 8
%Y2 = insertelement <2 x double> undef, double %A1, i32 0
%Y1 = insertelement <2 x double> %Y2, double %B2, i32 1
%Z1 = shufflevector <2 x double> %Y1, <2 x double> undef, <2 x i32> <i32
      1, i32 1>
%q = extractelement <2 x double> %Z1, i32 0
```

Basic-Block Autovectorization

Unlike loop autovectorization, whole-function autovectorization, etc. which operate on regions with non-trivial control flow, basic-block autovectorization operates within each basic block independently.

This makes the domain simpler, but in many ways, makes the underlying problem harder: Without the ability to use loops or other structures as "templates", basic-block autovectorization needs to search the potentially-large space of combinable instructions in order to create vectorized code out of scalar code.

```
%A1 = fadd double %B1, %C1
%A2 = fadd double %B2, %C2
```

```
%A = fadd <2 x double> %B, %C
```

Basic-Block Autovectorization Algorithm

How the LLVM implementation actually works...

The basic-block autovectorization stages:

- Identification of potential instruction pairings
- Identification of connected pairs
- Pair selection
- Pair fusion
- Repeat the entire procedure (fixed-point iteration)

After all this is done, instsimplify and GVN are used for cleanup.

Basic-Block Autovectorization Algorithm: Stage 1

```
foreach (instruction in the basic block) {
  if (instruction cannot possibly be vectorized)
    continue;

  foreach (successor instruction in the basic block)
    if (the two instructions can be paired)
      record the instruction pair as a vectorization candidate;
}
```

What instructions can be paired:

- Loads and stores (only simple ones)
- Binary operators
- Intrinsics (sqrt, pow, powi, sin, cos, log, log2, log10, exp, exp2, fma)
- Casts (for non-pointer types)
- Insert- and extract-element operations

Note: Determining whether two instructions can be paired depends on alias analysis, scalar evolution analysis and use tracking.

Basic-Block Autovectorization Algorithm: Stage 2

Motivation: Not all vectorization is profitable! We want to keep vector data in vector registers as long as possible with the largest amount of reuse.

```
foreach (candidate instruction pair) {
  foreach (successor candidate pair)
    if (both instructions in the second pair use some result from the first pair)
      record a pair connection;
}
```

A successor candidate pair is one where the first instruction in the second pair is a successor to the first instruction in the first pair.

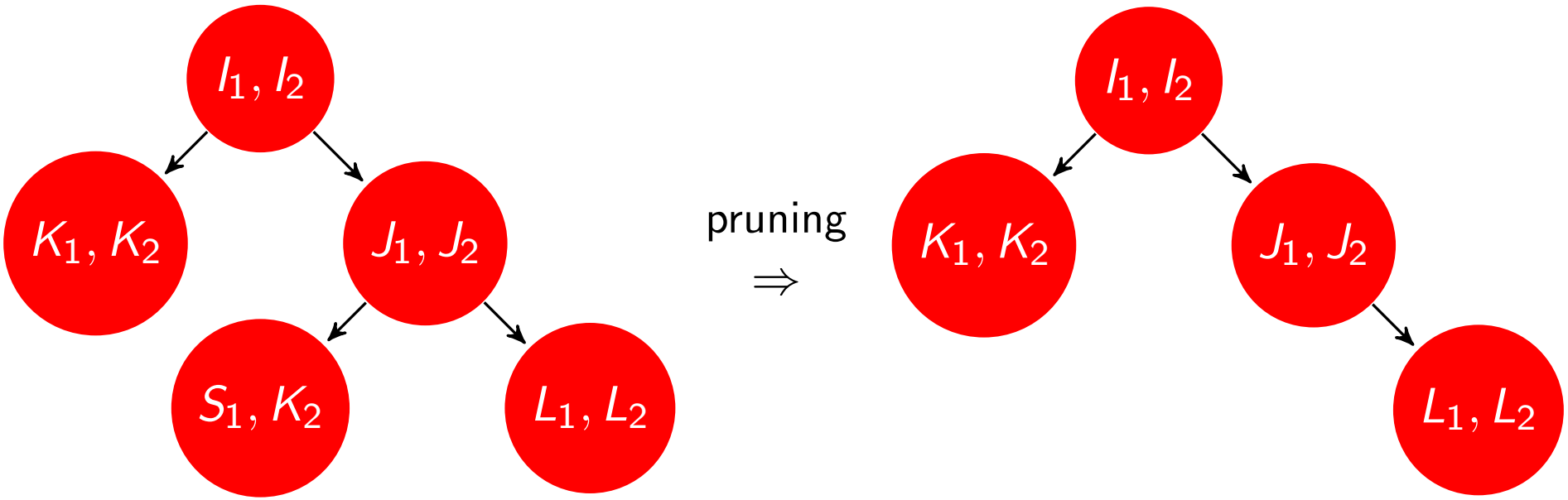
Basic-Block Autovectorization Algorithm: Stage 3

```
foreach (pairable instruction that is part of a remaining candidate pair) {
  best tree = null;
  foreach (candidate pair of which this instruction is a member) {
    if (this candidate pair conflicts with an already selected pair)
      continue;

    build and prune a tree with this pair as the root (and possibly make this tree
          the best tree) [see next slide];
  }

  if (best tree has the necessary size and depth) {
    remove from candidate pairs all pairs not in the best tree that share
        instructions with those in the best tree;
    add all pairs in the best tree to the list of selected pairs;
  }
}
```

Basic-Block Autovectorization Algorithm: Stage 3 (cont.)

build and prune a tree with this pair as the root:
  build a tree from all pairs connected to this pair (transitive closure);
  prune the tree by removing conflicting pairs (preferring pairs that have the deepest children);

  if (the tree has the required depth and more pairs than the best tree)
    best tree = this tree;



Basic-Block Autovectorization Algorithm: Conflict, Pruning: Why?

Non-trivial pairing-induced dependencies!

%div77 = fdiv double %sub74, %mul76.v.r1 <-> %div125 = fdiv double %mul121, %mul76.v.r2 (div125 depends on mul117)
%add84 = fadd double %sub83, 2.000000e+00 <-> %add127 = fadd double %mul126, 1.000000e+00 (add127 depends on div77)
%mul95 = fmul double %sub45.v.r1, %sub36.v.r1 <-> %mul88 = fmul double %sub36.v.r1, %sub87 (mul88 depends on add84)
%mul117 = fmul double %sub39.v.r1, %sub116 <-> %mul97 = fmul double %mul96, %sub39.v.r1 (mul97 depends on mul95)

(derived from a real example)

There are two mechanisms to deal with this:

- A full cycle check (used when the graph is small)
- "Late abort" during instruction fusion

Basic-Block Autovectorization Algorithm: Stage 4

```
foreach (instruction in a remaining selected pair) {
  form the input operands (generally using insertelement and shufflevector);
  clone the first instruction, mutate its type and replace its operands;
  form the replacement outputs (generally using extractelement and shufflevector
      );
  move all uses of the first instruction after the second;
  insert the new vector instruction after the second instruction;
  replace uses of the original instructions with the replacement outputs;
  remove the original instructions;
  remove this instruction pair from the list of remaining selected pairs.
}
```

One complication: If we're vectorizing address computations, then alias analysis may start returning different values as the fusion process continues. As a result, all needed alias-analysis queries need to be cached prior to beginning instruction fusion.

Basic-Block Autovectorization Algorithm: Depth Factors

Most instructions have a depth of one except:

- extractelement and insertelement have a depth of zero (and are never really fused).
- load and store each get half of the minimum required tree depth.

Basic-Block Autovectorization: Parameters

- **bb-vectorize-req-chain-depth - The required chain depth (default: 6)**
- bb-vectorize-search-limit - The maximum search distance for instruction pairs (default: 400)
- bb-vectorize-splat-breaks-chain - Replicating one element to a pair breaks the chain (default: false)
- **bb-vectorize-vector-bits - The size of the native vector registers (default: 128)**
- bb-vectorize-max-iter - The maximum number of pairing iterations (default: 0 = none)
- bb-vectorize-max-instr-per-group - The maximum number of pairable instructions per group (default: 500)
- bb-vectorize-max-cycle-check-pairs - The maximum number of candidate pairs with which to use a full cycle check (default: 200)

Basic-Block Autovectorization: Parameters (cont.)

- **bb-vectorize-no-ints - Don't vectorize integer values (default: false)**
- **bb-vectorize-no-floats - Don't vectorize floating-point values (default: false)**
- bb-vectorize-no-casts - Don't vectorize casting (conversion) operations (default: false)
- bb-vectorize-no-math - Don't vectorize floating-point math intrinsics (default: false)
- bb-vectorize-no-fma - Don't vectorize the fused-multiply-add intrinsic (default: false)
- bb-vectorize-no-mem-ops - Don't vectorize loads and stores (default: false)
- **bb-vectorize-aligned-only - Only generate aligned loads and stores (default: false)**
- bb-vectorize-no-mem-op-boost - Don't boost the chain-depth contribution of loads and stores (default: false)

Basic-Block Autovectorization: Benchmark Results

Benchmarks using clang/LLVM r154298 and gcc 4.7.0 on an Intel Xeon E5430 @ 2.66GHz. Autovectorization benchmark by Maleki, et al. (An Evaluation of Vectorizing Compilers - PACT'11):

- gcc: -std=c99 -O3 -funroll-loops -fivopts -flax-vector-conversions -ffast-math -funsafe-math-optimizations -msse4.1 (with -fno-tree-vectorize to turn off autovectorization)
- clang: -O3 -mllvm -unroll-allow-partial -mllvm -unroll-runtime -funsafe-math-optimizations -ffast-math (with -mllvm -vectorize -mllvm -bb-vectorize-aligned-only for autovectorization)

With autovectorization:
Tests for which clang/LLVM was faster than gcc: 43 (by $< 1\%$ in 4 cases)
Tests for which gcc was faster: 108 (by $< 1\%$ in 12 cases)
Without autovectorization:
Tests for which clang/LLVM was faster than gcc: 72 (by $< 1\%$ in 15 cases)
Tests for which gcc was faster: 79 (by $< 1\%$ in 24 cases)

Basic-Block Autovectorization: Speedup #1

Test S1119

```
for (int i = 1; i < LEN2; i++) {
  for (int j = 0; j < LEN2; j++) {
    aa[i][j] = aa[i-1][j] + bb[i][j];
  }
}
```

With autovectorization:
clang/LLVM: 3.92
gcc: 3.93

Without autovectorization:
clang/LLVM: 8.66
gcc: 8.69

Basic-Block Autovectorization: Speedup #2

Test S431

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

With autovectorization:
clang/LLVM: 25.26
gcc: 25.88

Without autovectorization:
clang/LLVM: 55.75
gcc: 58.26

Basic-Block Autovectorization: Speedup #3

Test S252: loop with ambiguous scalar temporary

```
t = (float) 0.;
for (int i = 0; i < LEN; i++) {
  s = b[i] * c[i];
  a[i] = s + t;
  t = s;
}
```

With autovectorization:
clang/LLVM: 4.24
gcc: 6.01

Without autovectorization:
clang/LLVM: 6.08
gcc: 6.34

Basic-Block Autovectorization: Speedup #4

Test S128: coupled induction variables with a jump in data access

```
j = -1;
for (int i = 0; i < LEN/2; i++) {
  k = j + 1;
  a[i] = b[k] - d[i];
  j = k + 1;
  b[k] = a[i] + c[k];
}
```

With autovectorization:
clang/LLVM: 9.02
gcc: 11.31

Without autovectorization:
clang/LLVM: 11.39
gcc: 11.30

Basic-Block Autovectorization: Speedup #5

Test S1115: triangular saxpy loop (linear dependence)

```
for (int i = 0; i < LEN2; i++) {
  for (int j = 0; j < LEN2; j++) {
    aa[i][j] = aa[i][j]*cc[j][i] + bb[i][j];
  }
}
```

With autovectorization:
clang/LLVM: 11.34
gcc: 13.97

Without autovectorization:
clang/LLVM: 13.44
gcc: 13.93

Basic-Block Autovectorization: Needs Improvement #1

Test S3113: maximum of absolute value

```
max = abs(a[0]);
for (int i = 0; i < LEN; i++) {
  if ((abs(a[i])) > max) {
    max = abs(a[i]);
  }
}
```

With autovectorization:
clang/LLVM: 34.83
gcc: 8.08

Without autovectorization:
clang/LLVM: 34.82
gcc: 33.03

Basic-Block Autovectorization: Needs Improvement #2

Test S2275: interchange needed

```
for (int i = 0; i < LEN2; i++) {
  for (int j = 0; j < LEN2; j++) {
    aa[j][i] = aa[j][i] + bb[j][i] * cc[j][i];
  }
  a[i] = b[i] + c[i] * d[i];
}
```

With autovectorization:
clang/LLVM: 32.13
gcc: 7.94

Without autovectorization:
clang/LLVM: 32.15
gcc: 32.88

Basic-Block Autovectorization: Needs Improvement #3

Test vsumr: vector sum reduction

```
sum = 0.;
for (int i = 0; i < LEN; i++) {
  sum += a[i];
}
```

With autovectorization:
clang/LLVM: 72.04
gcc: 18.03

Without autovectorization:
clang/LLVM: 72.05
gcc: 72.04

Basic-Block Autovectorization: Benchmark Synopsis

Simple loops work well, we need improvement where:

- Loop interchange is required
- Reasoning is required over multiple basic blocks (loops with if statements)
- Reductions

Cases where autovectorization makes the performance worse (by $> 1\%$):
clang/LLVM: 6 (only 3 were $> 2\%$)
gcc: 14 (all were $> 2\%$)

Cases where autovectorization gives a $> 40\%$ speedup:
clang/LLVM: 10
gcc: 42

Cases where autovectorization changes the performance by $< 1\%$:
clang/LLVM: 110
gcc: 56

Basic-Block Autovectorization: Future Directions

- Improved basic-block autovectorization:
  - Improvements to asymptotic complexity
  - Cost model (integration with TLI and more)
  - A lot of tuning and (probably) more heuristics
  - Instruction duplication
  - Asymmetric parings (add/sub pairings, add/shift pairings, etc.)
- Other types of autovectorization:
  - Loop vectorization (moving experience and code from Polly into LLVM's core)
  - Whole-function vectorization (work by Ralf Karrenberg, et al.)
  - Loop basic-block autovectorization: replacing unroll+vectorize with a loop-dependency-analysis-enhanced basic-block autovectorizer

Conclusion

- LLVM is now an autovectorizing compiler!
- Currently implemented: a basic-block autovectorizer
- The search space for basic-block autovectorization is large, heuristics must be used
- Going forward, loop vectorization, etc. should also be implemented
- Going forward, a better cost model will be needed

Acknowledgments

- The US Department of Energy and Argonne National Laboratory - For paying my salary.
- Tobi Grosser - For doing the bulk of the code review.
- Sebastian Pop and Roman Divacky - For reviewing the code, testing, and making some good suggestions.
- All of the other code reviewers!
- ARM Ltd. - For making this talk possible!
- The LLVM community - For making this project possible and worthwhile.