LLVM Auto-Vectorization

Past
Present
Future

Renato Golin
LLVM Auto-Vectorization

- Plan:
  - What is auto-vectorization?
  - Short-history of the LLVM vectorizer
  - What do we support today, and an overview of how it works
  - Future work to be done

- This talk is **NOT** about:
  - Performance of the vectorizer compared to scalar LLVM
  - Performance of the LLVM vectorizer against GCC's
  - Feature comparison of any kind...
  - All that is too controversial and not beneficial for understanding
What is auto-vectorization?

- It's the art of detecting instruction-level parallelism,
- And making use of SIMD registers (vectors)
- To compute on a block of data, in parallel
Auto-Vectorization?

- **What is auto-vectorization?**
  - It can be done in any language
  - But some are more expressive than others
  - All you need is a sequence of repeated instructions

```c
int a[N], b[N], c[N];
for (i=0; i<N; i++) {
    a[i] = b[i] + c[i];
}
```
The Past

How we came to be...
Where did it all come from?
Past

- Up until 2012, there was only Polly
  - Polyhedral analysis, high-level loop optimizations
  - Preliminary support for vectorization
  - No cost tables, no data-dependent conditions
  - And it needed external plugins to work

- Then, the BBVectorizer was introduced (Jan 2012)
  - Basic-block only level vectorizer (no loops)
  - Very aggressive, could create too many shuffles
  - Got a lot better over time, mostly due to the cost model

```c
%X1 = fsub double %A1, %B1
%X2 = fsub double %A2, %B2
%Y1 = call double @llvm.fma.f64(, ,)
%Y2 = call double @llvm.fma.f64(, ,)
%Z1 = fadd double %Y1, %B1
%Z2 = fadd double %Y2, %B2
%R = fmul double %Z1, %Z2
%X1 = fsub <2 x double> %X1.v.i0.2, %X1.v.i1.2
%Y1 = call <2 x double> @llvm.fma.v2f64(<2 x double>)
%Z1 = fadd <2 x double> %Y1, %X1.v.i1.2
%Z1.v.r1 = extractelement <2 x double> %Z1, i32 0
%Z1.v.r2 = extractelement <2 x double> %Z1, i32 1
%R = fmul double %Z1.v.r1, %Z1.v.r2
```
The Loop Vectorizer (Oct 2012)
- It could vectorize a few of the GCC's examples
- It was split into Legality and Vectorization steps
- No cost information, no target information
- Single-block loops only

Example 1:
```c
int a[256], b[256], c[256];
foo () {
    int i;
    for (i=0; i<256; i++) {
        a[i] = b[i] + c[i];
    }
}
```

Example 8:
```c
int a[M][N];
foo (int x) {
    int i, j;
    /* feature: support for multidimensional arrays */
    for (i=0; i<M; i++) {
        for (j=0; j<N; j++) {
            a[i][j] = x;
        }
    }
}
```

Example 12: induction:
```c
for (i = 0; i < N; i++) {
    a[i] = i;
}
```
Past

- The cost model was born (Late 2012)
  - Vectorization was then split into three stages:
    - Legalization: can I do it?
    - Cost: Is it worth it?
    - Vectorization: create a new loop, vectorize, ditch the older
  - Only X86 was tested, at first

- Cost tables were generalized for ARM, then PPC
  - A lot of costs and features were added based on manuals and benchmarks for ARM, x86, PPC
  - It should work for all targets, though
  - Reduced a lot of the regressions and enabled the vectorizer to run at lower optimization levels, even at -Os
  - The BB-Vectorizer started to benefit from it as well
Past

- The SLP Vectorizer (Apr 2013)
  - Stands for *superword-level parallelism*
  - Same principle as BB-Vec, but bottom-up approach
  - Faster to compile, with fewer regressions, more speedup
  - It operates on multiple basic-blocks (trees, diamonds, cycles)
  - Still doesn't vectorize function calls (like BB, Loop)

- Loop and SLP vectorizers enabled by default (-Os, -O2, -O3)
  - -Oz is size-paranoid
  - -O0 and -O1 are debug-paranoid
  - Reports on x86_64 and ARM have shown it to be faster on real applications, without producing noticeably bigger binaries
  - Standard benchmarks also have shown the same thing
The Present
What do we have today?
Present - Features

- Supported syntax
  - Loops with unknown trip count
- Reductions
- If-Conversions
- Reverse Iterators
- Vectorization of Mixed Types
- Vectorization of function calls

```c
for (int i = start; i < end; ++i)
```

```c
unsigned sum = 0;
for (int i = 0; i < n; ++i)
    sum += A[i] + 5;
return sum;
```

```c
for (int i = 0; i < n; ++i)
    if (A[i] > B[i])
        sum += A[i] + 5;
```

```c
for (int i = n; i > 0; --i)
    A[i] += 1;
```

```c
int foo(int *A, char *B, int n, int k) {
    for (int i = 0; i < n; ++i)
        A[i] += 4 * B[i];
}
```

```c
for (int i = 0; i <= 1024; ++i)
    f[i] = floor(f[i]);
```

See [http://llvm.org/docs/Vectorizers.html](http://llvm.org/docs/Vectorizers.html) for more info.
Present - Features

- Supported syntax
- Runtime Checks of Pointers
- Inductions
- Pointer Induction Variables
- Scatter / Gather
- Global Structures Alias Analysis
- Partial unrolling during vectorization

See http://llvm.org/docs/Vectorizers.html for more info.
Present - Validation

• **CanVectorize()**
  - Multi-BB loops must be able to if-convert
  - Exit count calculated with Scalar Evolution of induction
  - Will call canVectorizeInstrs, canVectorizeMemory

• **CanVectorizeInstrs()**
  - Checks induction strides, wrap-around cases
  - Checks special reduction types (add, mul, and, etc)

• **CanVectorizeMemory()**
  - Checks for simple loads/stores (or annotated parallel)
  - Checks for dependent access, overlap, read/write-only loop
  - Adds run-time checks if possible
Present - Cost

• **Vectorization Factor**
  • Make sure target supports SIMD
  • Detect widest type / register, number of lanes
  • -Os avoids leaving the tail loop (ex. Run-time checks)
  • Calculates cost of scalar and all possible vector widths

• **Unroll Factor**
  • To remove cross-iteration deps in reductions, or
  • To increase loop-size and reduce overhead
  • But not under -Os/-Oz

• If not beneficial, and not -Os, try to, *at least*, unroll the loop
Present - Vectorization

• Creates an empty loop

• ForEach BasicBlock in the Loop:
  • Widens instructions to <VF x type>
  • Handles multiple load/stores
  • Finds known functions with vector types
  • If unsupported, scalarizes (code bloat, performance hit)

• Handles PHI nodes
  • Loops over all saved PHIs for inductions and reductions
  • Connects the loop header and exit blocks

• Validates
  • Removes old loop, cleans up the new blocks with CSE
  • Update dominator tree information, verify blocks/function
The Future

What will come to be?
Future – General

- Future changes to the vectorizer will need re-thinking some code
  - Adding call-backs for error reporting for pragmas
  - Adding more complex memory checks, stride access
  - More accurate/flexible cost models

- Unify the feature set across all vectorizers
  - Migrate remaining BB features to SLP vectorizer
  - Implement function vectorization on all
  - Deprecate the BB vectorizer

- Integrate Polly and Loop Vectorizer
  - Allow outer-loop transformations and more complicated cases
  - Make Polly an integral part of LLVM
Future – Pragmas

• Hints to the vectorizer, doesn't compromise safety
  • The vectorizer will still check for safety (memory, instruction)

• \#pragma vectorize
  • disable/enable helps work around cost model problems
  • width(N) controls the size (in elements) of the vector to use
  • unroll(N) helps spotting extra cases

```c
#pragma vectorize width(4) unroll(2)
for (i = 0; i < N; ++i) {
    A[i] = B[i];
    C[i] = D[i];
}
```

```c
for (i = 0; i < N; i+=8) {
    A[i:i+3] = B[i:i+3];
    A[i+4:i+7] = B[i+4:i+7];
    C[i:i+3] = D[i:i+3];
    C[i+4:i+7] = D[i+4:i+7];
}
```

• Safety pragmas still under discussion...
Future – Strided Access

- LLVM vectorizer still doesn't have non-unit stride support

```c
for (i..N/3) {
    a[3*i] = b[3*i];
    a[3*i+1] = b[3*i+1];
    a[3*i+2] = b[3*i+2];
}
```

- Some strided access can be exposed with loop re-roller

```c
for (i..N/3) {
    a[3*i] = b[3*i] + K;
    a[3*i+1] = b[3*i+1] + K;
    a[3*i+2] = b[3*i+2] + K;
}
```
Future – Strided Access

• But if the operations are not the same, we can't re-roll

```c
for (i .. N/3) {
    a[3*i] = b[3*i] + I;
    a[3*i+1] = b[3*i+1] + J;
    a[3*i+2] = b[3*i+2] + K;
}
```

• We have to unroll the loop to find interleaved access

```c
for (i .. N/3) {
    a[3*i] = b[3*i] + I;
    a[3*i+3] = b[3*i+3] + I;
    a[3*i+6] = b[3*i+6] + I;
    a[3*i+9] = b[3*i+9] + I;
    a[3*i+1] = b[3*i+1] + J;
    a[3*i+2] = b[3*i+2] + K;
    a[3*i+8] = b[3*i+8] + K;
}
```
Thanks & Questions

• Thanks to:
  • Nadav Rotem
  • Arnold Schwaighofer
  • Hal Finkel
  • Tobias Grosser
  • Aart J.C. Bik's “The Software Vectorization Handbook”

• Questions?
References

• **LLVM Sources**
  • lib/Transform/Vectorize/LoopVectorize.cpp
  • lib/Transform/Vectorize/SLPVectorizer.cpp
  • lib/Transform/Vectorize/BBVectorizer.cpp

• **LLVM vectorizer documentation**
  • http://llvm.org/docs/Vectorizers.html

• **GCC vectorizer documentation**
  • http://gcc.gnu.org/projects/tree-ssa/vectorization.html

• **Auto-Vectorization of Interleaved Data for SIMD**
  • http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.91.6457