Vectorization in LLVM

Nadav Rotem, Apple
Arnold Schwaighofer, Apple
LLVM-based vectorizers

- AnySL
- Intel OpenCL
- Polly
- ISPC

- Hal's BB vectorizer
- Loop vectorizer
- SLP vectorizer
Thanks!

Thanks to all of the people who contributed to the vectorizer in the last year
Performance

*\text{x86, +AVX, Sandybridge}
Performance

LLVM test suite

**x86_64, +avx, sandybridge, higher better
Usage

• Loop Vectorizer.
  • -fvectorize / -fno-vectorize

• SLP Vectorizer.
  • -fslp-vectorize / -fno-slp-vectorize

• Both on by default on -Os, -O2 and -O3
Loop Vectorizer

- Vectorizes innermost loops
- Unrolls loops for ILP
Features

• Loops with unknown trip count

```c
void foo(float *A, float* B, int start, int end) {
    for (int i = start; i < end; ++i)
        A[i] *= B[i];
}
```
Features

• Loops that count backwards

```c
void foo(int *A, int n) {
    for (int i = n; i > 0; --i)
        A[i] += i;
}
```
Features

- Runtime array bounds check

```c
void foo(float *A, float *B, float K) {
    for (int i = 0; i < N; ++i)
}
```
Features

• Reductions

```c
int foo(int *A, int *B, int K) {
    int sum = 0;

    for (int i = 0; i < N; ++i)
        sum += A[i] + K;

    return sum;
}
```
Features

- Inductions

```c
int foo(int *A) {
    for (int i = 0; i < N; ++i)
        A[i] = i;
}
```
Features

• If-conversion (loops with ifs)

```c
int foo(int *A, int *B) {
    int sum = 0;

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

    return sum;
}
```
Features

• Pointer and C++ iterators vectorization

```cpp
int foo(int *A, int n) {
    return std::accumulate(A, A + n, 0);
}
```
Features

- Partial vectorization (parts of the code are scalar)

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

• Vectorization of mixed types

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

- Vectorization of some function calls

```c
int foo(float *A) {
    for (int i = 0; i < 1024; ++i)
        A[i] += floorf(f[i]);
}
```
Features

• Unrolling for ILP during vectorization

```c
for (i = 0; i < (N/8)*8; i+=8) {
    r1 += A[i:i+3];
    r2 += A[i+4:i+7];
}
```

```c
for (i = 0; i < N; ++i) {
    r += A[i];
}
```

```c
r = r1 <+> r2
...```

Design

• 3 phases:
  • Legality
  • Profitability
  • Transform
Legality

- Inductions/Reductions

- Memory access safety

- Memory checks

- Vectorizable intrinsics

```c
for (int i = 0; i < N; i++)
    r += A[i] + i;
```
```c
for (int i = 1; i < N; i++)
    A[i] = A[i-1];
```
```c
// A !overlap B
for (int i = 0; i < N; i++)
    A[i+1] = B[i];
```
```c
for (int i = 0; i < N; i++)
    A[i] = pow(B[i], 2.0);
```
Profitability

- Query cost model
- Choose vector width with lowest cost

\[ \text{unsigned getArithmeticInstrCost(unsigned Opcode, Type *Ty);} \]
How it works

• As much as possible from TargetLowering

TLI->isOperationLegalOrPromote(Opcode)
TLI->getTypeLegalizationCost(Ty)

• Generic rules

Cost = 1;
if (isExpand) Cost = 2;

Cost *= LegalizationCost;
Cost *= Width; // Scalarize.

• Exceptions from target specific tables

{ ISD::ZEXT, MVT::v4i8, 1 },
{ ISD::SEXT, MVT::v4i8, 3 },
if (A overlap B)  
goto scalar loop

for (i = 0; i < (N/8)*8; i+=8) {
    r1 += A[i:i+3];
    r2 += A[i+4:i+7];
}

r = r1 <-> r2;  // Horizontal

for (; i < N; i++) {
    r += A[i];  // Remainder
}
Unrolling for ILP

- Modern processors can execute many instructions at once
- Reductions introduce data-hazards (compute depends on previous iteration)

```c
for (i = 0; i < N; ++i)
    sum += A[i];
```
Unrolling in the vectorizer

for (i = 0; i < n; ++i)
    sum += A[i];
for (i = 0; i < n; i+=2) {
    sum0  += A[i];
    sum1  += A[i+1];
}
Why unroll in the vectorizer?

- Same kind of analysis that the vectorizer does (e.g. reductions + tail loop)
- Loop Vectorizer often unrolls and keeps the code scalar
Loop Vectorizer TODO

• Support for vectorizer #pragma
• Vectorization with library functions
• Inlining + restrict
• Vectorization of interleaved data
• Support for AVX512 (predication, ...)

#pragma

- Control vectorization width and unrolling
- Mark as legal (no runtime checks needed)

```c
void foo(int *A, int *B) {
    #pragma vectorize factor(2) unroll(2)
    for (i = 0; i < N; i++)
        B[i+1] = A[i];
}
```
Vectorization with library functions

- Vectorized library function calls available
- Use vectorized library function

```c
void foo(float *A, float *B, float P) {
    for (int i = 0; i < 256; ++i)
        A[i] = pow(B[i], P);
}

void foo(float *A, float *B, float P) {
    for (int i = 0; i < 256; i += 4)
        A[i:i+3] = vector_pow(B[i:i+4], <P, P, P, P>);
}
```
Inlining and restrict

- After inlining we lose 'restrict' keyword

```c
int foo(int *restrict A, int *restrict B) {
    for (int i = 0; i < N; i++)
        A[i+1] = B[i];
}

void bar() {
    foo(Ptr1, Ptr2)
}
```

```c
void bar() {
    int *restrict Ptr1 = Ptr1; int *restrict Ptr2 = Ptr2;
    for (int i = 0; i < N; i++)
        Ptr1[i+1] = Ptr2[i];
}
```
Vectorization of interleaved data

- Vectorizer looks at each instruction individually
- Deemed too expensive due to gather/scather

```c
void foo(float *A, float *B, float *C) {
    for (int i = 0; i < 256; ++i) {
    }
}
```

4 Loads + Inserts into vector just for B[2*i]
Look at all accesses

\[
\begin{align*}
\end{align*}
\]

4 Vector Loads

2 Vector Stores
SLP Vectorization
SLP vectorizer

- Superword-Level Parallelism
- Combines multiple scalars into one vector operation
- Reduce code size and register pressure
- Excellent for graphics code that uses RGBA
- Example:

```c
void foo(double *A, double *B) {
    A[0] = B[0] + 56.0;
}
```
Example

• 2X boost in performance on matmul_f64_4x4:

```c
static void mul4(double *Out, double A[4][4], double B[4][4]) {
    unsigned n; double Res[16];
```

```
    Res[ 0] = A[0][0]*B[0][0] + A[0][1]*B[1][0] + A[0][2]*B[2][0] + A[0][3]*B[3][0];
    ...
}
```
How SLP Vectorization works

- Bottom-up search
- Vectorize profitable trees

```c
int foo(int *A, int *B) {
    A[0] = B[0] + 56;
}
```
SLP Vectorization Phases

1. Build a vectorizable tree
2. Estimate the cost of the tree
3. Vectorize the tree
Finding roots

• Consecutive stores
  \[ A[i] = \ldots \]
  \[ A[i+1] = \ldots \]

• Arithmetic reductions
  \[ \text{sum} += \ldots \]
  \[ \text{sum} += \ldots \]

• PHI node sequences

• Other popular patterns
Features

- Gather and broadcast sequences
Features

- Multiple basic blocks
- Vectorize PHIs to reduce register pressure
Features

- Diamond-shaped tree graph
Features

- Swizzle binary operations
Features

• External users
TODO

- Function call vectorization
- Cost model for vector width = 3
- Loop aware multi-block cost model tuning
- Additional root patterns
How can you help?

• Analyze workloads
• Benchmark LLVM
  • Compare to other compilers
  • Try different cpu features, vector width, etc
• Improve vector code generation and cost model
• Implement a new feature
Questions?