LLVM Auto-Vectorization

Past

Present

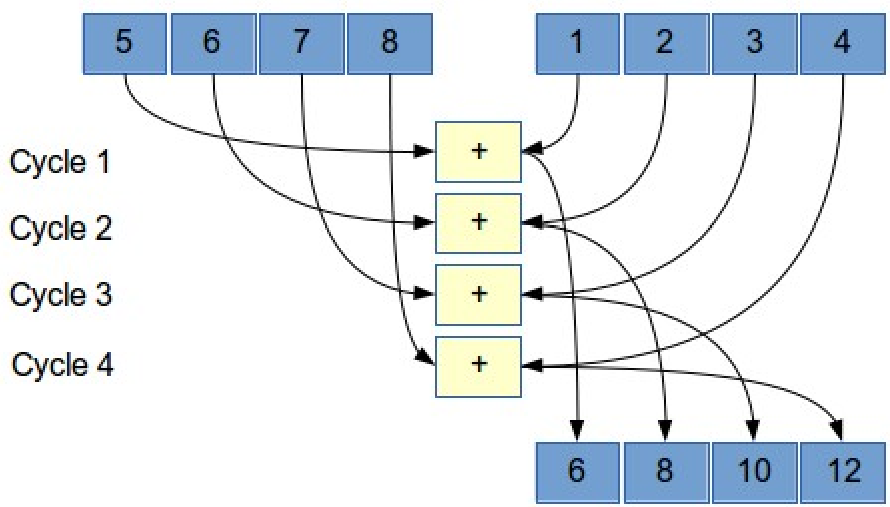
Future

Renato Golin

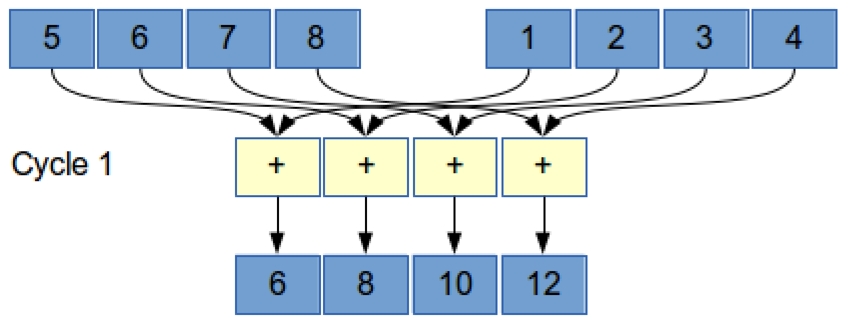
Auto-Vectorization?

- **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

Scalar Addition



Vector Adition



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

```
int a[N], b[N], c[N];

for (1..N, i++) {
  a[i] = b[i] + c[i];
}
```

→

```
int a[N], b[N], c[N];

for (1..N/4, i+=4) {
  a[i+0] = b[i+0] + c[i+0];
  a[i+1] = b[i+1] + c[i+1];
  a[i+2] = b[i+2] + c[i+2];
  a[i+3] = b[i+3] + c[i+3];
}
```

→

```
int a[N], b[N], c[N];

for (1..N/4, i+=4) {
  a[0..3] = b[0..3] + c[0..3];
}
```

```
int a, b, c, d;

if (x > 0) {
  a = c + 4;
  b = d + 4;
}
```

→

```
int a, b, c, d;

if (x > 0) {
  {a,b} = {c,d} + {4,4};
}
```

LLVM Auto-Vectorization

The Past

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

Past

- Up until 2012, there was only Polly (Tobias Grosser)
  - which was hard to setup
  - and only worked on very complicated cases
- Then, Hal Finkel introduced the BBVectorizer (Jan 2012)
  - Basic-block only level vectorizer (no loops)
  - Very aggressive, could create too many suffles
  - Got a lot better over time, mostly due to the cost model

```
%X1 = fsub double %A1, %B1
%X2 = fsub double %A2, %B2
```

→

```
%X1 = fsub <2 x double> %X1.v.i0.2, %X1.v.i1.2
```

```
%Y1 = call double @llvm.fma.f64(
%Y2 = call double @llvm.fma.f64(
```

→

```
%Y1 = call <2 x double> @llvm.fma.v2f64(<2 x double>
```

```
%Z1 = fadd double %Y1, %B1
%Z2 = fadd double %Y2, %B2
%R  = fmul double %Z1, %Z2
```

→

```
%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
```

Past

- Nadav introduced 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

example1:

```
int a[256], b[256], c[256];
foo () {
  int i;

  for (i=0; i<256; i++){
    a[i] = b[i] + c[i];
  }
}
```

example8:

```
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;
     }
   }
}
```

example12: induction:

```
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, maybe ditch the older
  - X86 only at first

  - Cost tables were generalized for ARM, then PPC
    - Arnold and others added **a lot** of costs based on manuals and benchmarks for ARM, x86, PPC
    - It should work for all targets, though
    - Reduced a lof of the regressions and enabled the vectorizer to run at lower optimization levels, even at -Os

Past

- Nadav introduced the SLP Vectorizer (Apr 2013)
  - Stands for *superword-level paralellism*
  - Same principle as BB-Vec, but bottom-up approach
  - But 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

```
for (int i = start; i < end; ++i)
  A[i] *= B[i] + K;
```

  - Reductions

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

  - If-Conversions

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

  - Reverse Iterators

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

  - Vectorization of Mixed Types

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

  - Vectorization of function calls

```
for (int i = 0; i != 1024; ++i)
  f[i] = floorf(f[i]);
```

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

Present - Features

- **Supported syntax**
  - Runtime Checks of Pointers

```
void bar(float *A, float* B, float K, int n) {
  for (int i = 0; i < n; ++i)
    A[i] *= B[i] + K;
}
```

  - Inductions

```
for (int i = 0; i < n; ++i)
  A[i] = i;
```

  - Pointer Induction Variables

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

  - Scatter / Gather

```
for (int i = 0; i < n; ++i)
  A[i*7] += B[i*k];
```

  - Global Structures Alias Analysis

```
struct { int A[100], K, B[100]; } Foo;

int foo() {
  for (int i = 0; i < 100; ++i)
    Foo.A[i] = Foo.B[i] + 100;
}
```

  - Partial unrolling during vectorization

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

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

LLVM Auto-Vectorization

The Future

What will come to be?

Future – A simple loop

- Vectorizing this loop is not trivial...

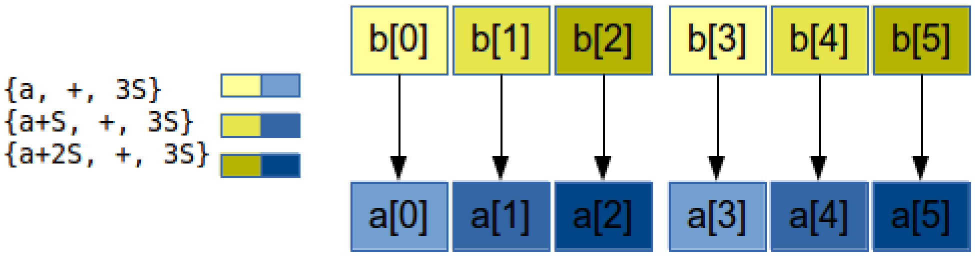
```
for (i..N/3) {
 a++ = b++ + I;
 a++ = b++ + J;
 a++ = b++ + K;
}
```

- It requires a handfull of new optimizations:
  - Detection of non-unit strided access via special unrolling
  - Detection of interleaved access via load/store grouping
  - Similarities between pointer and array access
  - Possible re-rolling loops to expose parallelism
  - Reduction loops should also be recognized...

Future – Strided Access

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

```
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

```
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;
}
```

```
for (i..N) {
  a[i] = b[i] + K;
}
```

Future – Strided Access

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

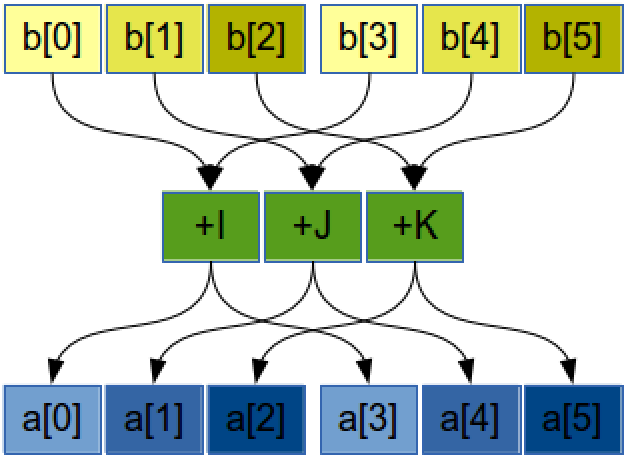
```
for (i..N/3) {
 a[3*i] = b[3*i] + I;
 a[3*i+1] = b[3*i+1] + J;
 a[3*i+2] = b[3*i+2] + K;
}
```

- We have to unroll the loop to find interleaved access

```
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+4] = b[3*i+4] + J;
 a[3*i+7] = b[3*i+7] + J;
 a[3*i+10] = b[3*i+10] + J;

 a[3*i+2] = b[3*i+2] + K;
 a[3*i+5] = b[3*i+5] + K;
 a[3*i+8] = b[3*i+8] + K;
 a[3*i+11] = b[3*i+11] + K;
}
```



Future – Strided Access

- And finally, we'll need to teach the vectorizer the relationship between *pointer access* and *array access:*

```
for (i..N/3) {
 a++ = b++ + K;
 a++ = b++ + K;
 a++ = b++ + K;
}
```

=

```
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;
}
```

- And about reductions:

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

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/BBVectorize.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