Partial evaluation with LLVM

Syoyo Fujita

Agenda

Motivation

How to do it

Conclusion

Motivation

Run-time input-data centric optimization

- Hard to optimize the code in compile time.

Example

- Optimize math kernel for input value range
- Optimize rasterizer code for size of input polygon.
- Runtime code JIT-ing for VM

Example

Compute y = x^n

```
double mypower(double x, int n) {
  int i;
  double ret = x;
  for (i = 1; i < n; i++) {
    ret *= x;
  }
  return ret;
}
```

If we know n = 3 anytime, we can expand the function as follows.

```
double mypower(double x, int n) {
  double ret;
  ret = x * x * x;
  return ret;
}
```

Let LLVM do this optimization

Optimize&emit native code when parameter is constant.

Procedure

1) Read a LLVM bitcode

2) Clone function signature to create specialized function

3) Replace variable n with constant value

4) Create specialized function

5) Apply optimization

1) Read a bitcode

```
// Load the input module...
std::auto_ptr<Module> M;
if (MemoryBuffer *Buffer
      = MemoryBuffer::getFileOrSTDIN(InputFilename,
                                     &ErrorMessage)) {
  M.reset(ParseBitcodeFile(Buffer, &ErrorMessage));
  delete Buffer;
}
```

Input bitcode

```
define double @mypower(double %x, i32 %n) nounwind  {
entry:
   %tmp1019 = icmp slt i32 %n, 1    ; <i1> [#uses=1]
   br i1 %tmp1019, label %bb12, label %bb7

bb7:      ; preds = %bb7, %entry
   %indvar = phi i32 [ 0, %entry ], [ %indvar.next, %bb7 ]     ; <i32> [#uses=2]
   %ret.0.reg2mem.0 = phi double [ %x, %entry ], [ %tmp4, %bb7 ]      ; <double>
[#uses=1]
   %tmp4 = mul double %ret.0.reg2mem.0, %x    ; <double> [#uses=2]
   %tmp6 = add i32 %indvar, 2    ; <i32> [#uses=1]
   %tmp10 = icmp sgt i32 %tmp6, %n      ; <i1> [#uses=1]
   %indvar.next = add i32 %indvar, 1    ; <i32> [#uses=1]
   br i1 %tmp10, label %bb12, label %bb7

bb12:     ; preds = %bb7, %entry
   %ret.0.reg2mem.1 = phi double [ %x, %entry ], [ %tmp4, %bb7 ]      ; <double>
[#uses=1]
   ret double %ret.0.reg2mem.1
}
```

2) Clone function signature to create specialized function

2.1) find the function “mypower” from the module

```
void
doIt(Module *m, std::ostream *Out)
{

    Module::iterator f, e;

    // Iterate over functions in the module.
    for (f = m->begin(), e = m->end(); f != e; ++f) {
        if (!f->isDeclaration()) {
            if (f->getNameStr() == "mypower") break;
        }
    }
```

3) Replace argument n with constant

```
DenseMap<const Value*, Value*> ValueMap;
Function *specializedFunction =
  CloneFunctionInfo(f, ValueMap);
specializedFunction->setName(f->getNameStr() + "_3");
```

```
double mypower(double x, int n)
{
  int i;
  n = 3;
  double ret = x;
  for (i = 1; i <= n; i++) {
    ret *= x;
  }
  return ret;
}
```

```
// Iterate over function argments.
for (Function::arg_iterator i = f->arg_begin(),
     ie = f->arg_end(); i != ie; ++i) {

    Argument *arg = i;

    Value *val;

    if (arg->getNameStr() == "n") {

        // Replace arg 'a' with Const expresison 3.
        val = ConstantInt::get(Type::Int32Ty, 3);

        ValueMap[i] = val;
    }

}
```

```
Function *CloneFunctionInfo(const Function *F,
                            DenseMap<const Value*, Value*> &ValueMap)
{
  std::vector<const Type*> ArgTypes;

  // The user might be deleting arguments to the function by specifying them in
  // the ValueMap.  If so, we need to not add the arguments to the arg ty vector
  //
  for (Function::const_arg_iterator I = F->arg_begin(), E = F->arg_end();
       I != E; ++I)
    if (ValueMap.count(I) == 0)  // Haven't mapped the argument to anything yet?
      ArgTypes.push_back(I->getType());

  // Create a new function type...
  FunctionType *FTy = FunctionType::get(F->getFunctionType()->getReturnType(),
                                        ArgTypes, F->getFunctionType()->isVarArg());

  // Create the new function...
  Function *NewF = Function::Create(FTy, F->getLinkage(), F->getName());

  // Loop over the arguments, copying the names of the mapped arguments over...
  Function::arg_iterator DestI = NewF->arg_begin();
  for (Function::const_arg_iterator I = F->arg_begin(), E = F->arg_end();
       I != E; ++I)
    if (ValueMap.count(I) == 0) {   // Is this argument preserved?
      DestI->setName(I->getName()); // Copy the name over...
      ValueMap[I] = DestI++;        // Add mapping to ValueMap
    }

  return NewF;

}
```

4) Create specialized function

```
std::vector<ReturnInst*> Returns;
ClonedCodeInfo SpecializedFunctionInfo;

CloneAndPruneFunctionInto( specializedFunction,       // NewFunc
                           f,                         // OldFunc
                           ValueMap,                  // ValueMap
                           Returns,                   // Returns
                           ".",                       // NameSuffix
                           &SpecializedFunctionInfo,  // CodeInfo
                           0);                        // TD
```

bitcode of specialized function

```
define double @mypower_3(double %x, i32 %n) {
entry.:
    br label %bb7.

bb7.:      ; preds = %bb7., %entry.
    %indvar. = phi i32 [ 0, %entry. ], [ %indvar.next., %bb7. ]     ; <i32> [#uses=2]
    %ret.0.reg2mem.0. = phi double [ %x, %entry. ], [ %tmp4., %bb7. ]  ; <double>
[#uses=1]
    %tmp4. = mul double %ret.0.reg2mem.0., %x      ; <double> [#uses=2]
    %tmp6. = add i32 %indvar., 2      ; <i32> [#uses=1]
    %tmp10. = icmp sgt i32 %tmp6., 3     ; <i1> [#uses=1]
    %indvar.next. = add i32 %indvar., 1     ; <i32> [#uses=1]
    br i1 %tmp10., label %bb12., label %bb7.

bb12.:     ; preds = %bb7.
    %ret.0.reg2mem.1. = phi double [ %tmp4., %bb7. ]      ; <double> [#uses=1]
    ret double %ret.0.reg2mem.1.
}
```

5) Apply optimization

```
M.getFunctionList().push_back(specializedFunction);
WriteBitcodeToFile(M.get(), *Out);
```

```
$ ./partial_eval mypower.bc -o mypower3.bc -f
$ opt -std-compile-opts mypower3.bc -o mypower3.opt.bc
-f
$ llvm-dis mypower3.opt.bc -f
```

You can also do it in online

by applying LLVM passes in the program

Final specialized & optimized bitcode

```
define double @mypower_3(double %x, i32 %n) nounwind {
entry.:
   %tmp4. = mul double %x, %x    ; <double> [#uses=1]
   %tmp4..1 = mul double %tmp4., %x    ; <double> [#uses=1]
   %tmp4..2 = mul double %tmp4..1, %x      ; <double> [#uses=1]
   ret double %tmp4..2
}
```

Numerical problem?

Accurate constant folding & propagation for floating point value.

- LLVM provides portable IEEE754 **SW** floating point calculation library.
- Same results on every CPU architecture.
  - No x87 nightmare!
- gcc 4.3 or later also accomplish this feature using mfpr and gmp library.

Example

```
double myfunc() {
 return 2.0 * cos(2.0);
}
```

LLVM

```
define double @func() nounwind  {
entry:
  ret double 0xBFEAA22657537205
}
```

```
octave> format hex
octave> 2.0 * cos(2.0)
ans = bfeaa22657537205
```

gcc(prior to 4.3)

```
_func:
00000000 pushl %ebp
00000001 movl  %esp,%ebp
00000003 pushl %ebx
00000004 subl  $0x24,%esp
00000007 movl  $0x00000000,(%esp)
0000000e movl  $0x40000000,0x04(%esp)
00000016 calll 0x00000035
0000001b fstpl 0xf0(%ebp)
0000001e movsd 0xf0(%ebp),%xmm0
00000023 addsd %xmm0,%xmm0
00000027 movsd %xmm0,0xf0(%ebp)
0000002c fldl  0xf0(%ebp)
0000002f addl  $0x24,%esp
00000032 popl  %ebx
00000033 leave
00000034 ret
```

gcc 4.4

```
_func:
00000000 pushl %ebp
00000001 movl  %esp,%ebp
00000003 subl  $0x08,%esp
00000006 calll 0x00000000
0000000b fldl  0x0000000d(%ecx)
00000011 leave
00000012 ret
```

An extract from gcc 4.3 changes

The GCC middle-end has been integrated with the MPFR library. This allows GCC to **evaluate and replace at compile-time calls to built-in math functions having constant arguments with their mathematically equivalent results**. In making use of MPFR, GCC can generate correct results regardless of the math library implementation or floating point precision of the host platform. This also allows GCC to generate identical results regardless of whether one compiles in native or cross-compile configurations to a particular target. The following built-in functions take advantage of this new capability: **`acos`, `acosh`, `asin`, `asinh`, `atan2`, `atan`, `atanh`, `cbrt`, `cos`, `cosh`, `drem`, `erf`, `erfc`, `exp10`, `exp2`, `exp`, `expm1`, `fdim`, `fma`, `fmax`, `fmin`, `gamma_r`, `hypot`, `j0`, `j1`, `jn`, `lgamma_r`, `log10`, `log1p`, `log2`, `log`, `pow10`, `pow`, `remainder`, `remquo`, `sin`, `sincos`, `sinh`, `tan`, `tanh`, `tgamma`, `y0`, `y1` and `yn`**. The `float` and `long double` variants of these functions (e.g. `sinf` and `sinl`) are also handled. The `sqrt` and `cabs` functions with constant arguments were already optimized in prior GCC releases. Now they also use MPFR.

Conclusion

Partial evaluation =

Replace function argument with constant value

Apply optimizer provided by LLVM.