Code Generation and Autovectorization with LLVM

Seminar Automation, Compilers, and Code-Generation

Simon Grätzer

High-Performance and Automatic Computing

06.07.2016

Introduction

- Code Generation Process in LLVM
  - ➔ The frontend of LLVM outputs target independent LLVM IR code
  - ➔ Compiler Backend has to transform this into machine code for a specific platform
  - ➔ In this process it can apply optimizations for the targeted platform
- Auto-Vectorization in LLVM
  - ➔ Converts code using only scalar data types and operations into code using vector-types and operations.
  - ➔ Vectorizing can lead to significant performance gains.
  - ➔ The compiler can perform vectorizing for the programmer (sometimes).

Agenda

Code Generation

Overview

- Code generation is one of the most complex tasks in the LLVM
- We need to "lower" the LLVM IR into the corresponding machine code
- Optimize code and replace unsupported data types and operations for the target platform
- LLVM supports this process through it's **target-independent code generator** framework
- This presentation focuses on this framework

Processing Steps

1. Instruction Selection
2. Scheduling and Formation
3. SSA-based Machine Code Optimizations
4. Register Allocation
5. Prolog/Epilog Code Insertion
6. Final Optimizations
7. Code Emission

Reminder: LLVM IR

Let's consider a simple C function:

```
int foo(int aa, int bb, int cc) {
  int sum = aa + bb;
  return sum / cc;
}
```

Reminder: LLVM IR

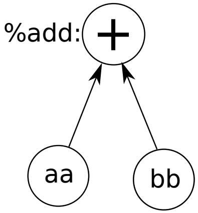
Is transforms into the LLVM IR: clang −cc1 foo.c −emit−llvm

```
define i32 @foo(i32 %aa, i32 %bb, i32 %cc) {
  entry:
    %add = add nsw i32 %aa, %bb
    %div = sdiv i32 %add, %cc
    ret i32 %div
  }
```

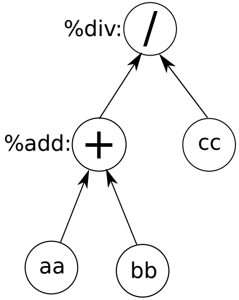
Note: This is still a very high level code representation. Variables are uniquely named and assigned exactly once this is called static single assignment (SSA) form.

Directed Acyclic Graph

DAG's represent the program code as basic blocks:



[aa + bb]



[sum / cc]

1. Leaf nodes represent identifiers, names or constants.
2. Interior nodes can represent operators.
3. Interior nodes also represent results of expressions or identifiers of locations where values are to be stored.

SelectionDAG

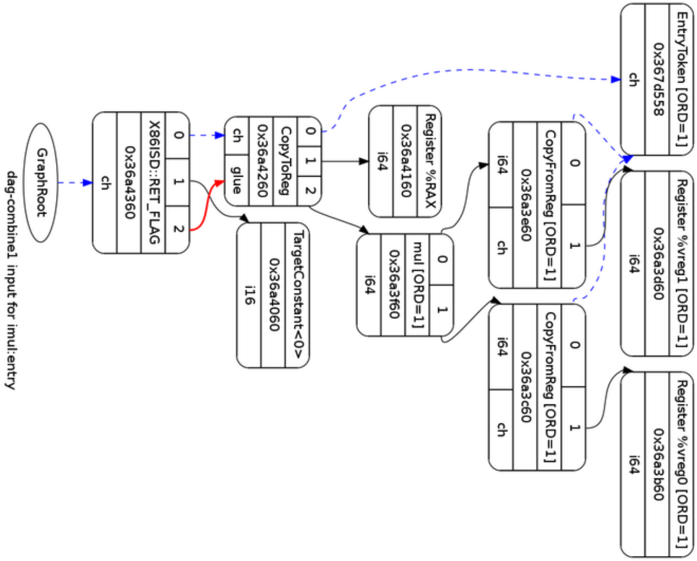
- In LLVM the DAG is called a SelectionDAG, based on the SDNode class.
- LLVM defines a series of node types, which represent operators, vars etc.
- Works well for many phases of code generation.
- It is possible to perform transformations on the SelectionDAG by using techniques like **pattern matching**
- Transformations are performed in multiple passes on the DAG.
- Many performance optimizations are performed directly on this representation (e.g. peephole optimizations)

SelectionDAG: Example

```
define i64 @imul(i64 %a, i64 %b) nounwind readnone
entry:
  %mul = mul nsw i64 %b, %a
   ret i64 %mul
}
```

- This gets expanded into an initial SelectionDAG.
- Afterwards the first optimization pass is performed.

SelectionDAG: Example



Legalize Types Phase

- The current SelectionDAG is considered *illegal*, because not all types might be supported by the target platform.
- For example a target might require that all single floating point values are promoted to doubles or that bytes/shorts are handled as 32bit integers.
- The following operations have to be applied until the DAG is *legal*:
  - ➔ Promoting: Small types are transformed to larger types.
  - ➔ Expanding: Large integer types are broken up into smaller ones.
- The same needs to be applied for LLVMs vector types (used for SIMD instructions)

Legalizes Operations Phase

- Converts a DAG to only use the operations that are natively supported.
- Some CPU's have constraints, like not supporting every operation on all data types.

  E.g. x86 does not support byte conditional moves.
- The legalizer has to convert those operations:
  - Expansion: Replace the operation within a sequence of supported operations (open-coding)
  - Promotion: Change a type to a larger type which supports the operation.

Optimization of DAG

- The optimization phase is run three times on the DAG.
- First in the beginning on the original DAG:
  - ➔ Allows the initial code to be cleaned up.
  - ➔ For example allows to perform optimizations which depend on knowledge of the original data types.
- Once after both "legalization" phases (introduced later):
  - ➔ Clean up the potentially messy code generated by these passes, allows them to remain simple.
  - ➔ Optimizes the inserted sign and zero extensions.

Selection Phase
Generating Machine Instructions

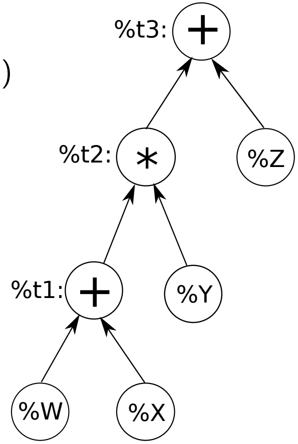
The SelectionDAG is transformed into another DAG based on the MachineSDNode class.
An instance of MachineSDNode contains all information necessary to generate the machine code instructions.

Example LLVM IR:

```
1 = fadd float %W, %X
2 = fmul float %t1 , %Y
3 = fadd float %t2 , %Z
```

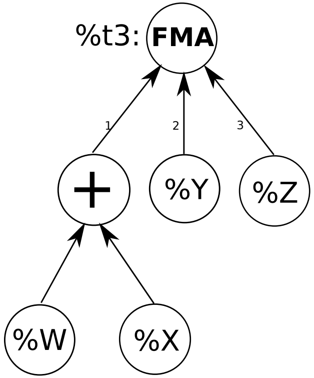
Selection Phase
Example

The previous IR code would be equivalent to the following DAG:
(fadd:f32 (fmul:f32 (fadd:f32 W, X), Y), Z)



Selection Phase
Example

If a target supports floating point multiply-and-add (**FMA**) operations the output can be simplified. On the PowerPC platform the *FMADDS* operation multiplies the first two parameters and adds the third.
Therefore the DAG can be simplified:
(FMADDS (FADDS W, X), Y, Z)



Scheduling and Formation

- Takes the *legal* SelectionDAG, which now contains the actual target instructions.
- The Scheduler assigns an order to every node which contains an instruction.
- There scheduler needs to take the constraints of the machine into account, e.g.
  - Optimize order for minimal register pressure, i.e. since the number of registers is limited we want to minimize the number of memory loads or stores (*spills* and *reloads*).
  - Take instruction latencies into account.
- This pass outputs a list of machine instructions, the SelectionDAG is discarded.

Register Allocation Phase

- Up until now, we still use an unlimited amount of *virtual registers* in SSA-form.
- These need to be mapped to a limited amount of *physical registers*.
- When the number of physical registers is not big enough, we need to map some virtual registers into memory.
- These virtual registers are called *spilled virtuals*.

Register Allocation Phase
Live Intervals

- Live Intervals are the intervals where a variable is actually used.
- We need to determine if two or more virtual registers are live at the same point and require the same physical register.
- In this case one virtual register must be *spilled*.
- The register allocator can use this information to determine the registers to spill.

Final Phase
Prolog/Epilog Code Insertion

- Throwing an exception in a function requires *unwinding*.
- Every time a function is calls another function, it's registers are written onto the stack as a stack frame.
- When an exception is thrown it is handled in a *catch block* somewhere in a function.
- Stack Unwinding means to clean up the stack frames of the functions below this function's frame.
- For example in C++ all the objects need to be deallocated etc.

Final Phase
Code Emission

- Currently the code representation still contains labels or directives (assembly code)
  - ➔ Labels are used to identify locations in the program.
  - ➔ Directives are commands which influence the code emission in some way.
  - ➔ For example to place some string data into a section of the program file.
- Needs to be converted into the actual opcode, which can be executed by the processor.
- Address locations need to be computed, object file needs to be generated, ...

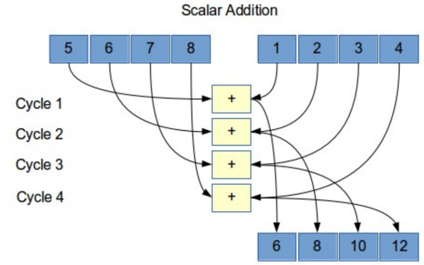
Auto-Vectorization

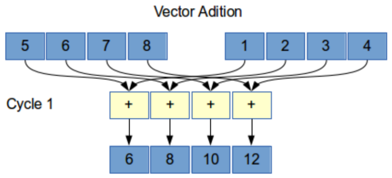
Introduction

- Aarray programming is concept of applying operations to entire vectors of scalars.
- **Vectorization** is the process of converting a scalar program into a program using vectors.
- CPU instructions working on vectors are called **Single Instruction Multiple Data**.
- Most modern CPUs support SIMD instructions and have vector registers, with varying sizes.
- For example on modern x86 processors vector registers can have sizes of 128 (SSE), 256 (AVX2), 512 (AVX-512) bits.
- Vectorization let's us take advantage of modern CPU's to gain speed, improve energy efficiency and have a smaller code base.

Visualization

Instead of sequential operations, we process everything in parallel





Vectorization Example

For example consider this C program:

```
int a[8]; int b[8];
for (i=0; i < 8; i++)
    a[i] = a[i] + b[i];
```

Using the GCC vector extension* we can rewrite this:

```
// using 256bit AVX2 register
typedef int vec8 __attribute__ ((vector_size (32)))
vec8 a, b;
a = a + b;
```

*) Also supported by LLVM

Auto-Vectorization

- Since there are many target specific ways of working with vectors, the compiler should take care of this for the programmer.
- The LLVM compiler has two vectorizers:
  - ➔ Loop Vectorizer, which operates on Loops
  - ➔ SLP Vectorizer, which merges scalars into vectors

The LLVM IR supports vector data types natively:

```
%vec1 = load <8 x i32>* %addr1
%vec2 = load <8 x i32>* %addr2
%vec3 = fmul <8 x i32> %vec1, %vec2
```

Challenges

- The data we want to work on must be in a certain format
  - ➔ It must fit into the vector registers of the CPU.
  - ➔ The data must be correctly laid out for the SIMD instructions.
- The program behaviour must be preserved, to this end the compiler guarantees:
  - ➔ Data dependencies must be respected.
  - ➔ The precision of the data types must be preserved, otherwise results become less precise.

Data Dependencies

There are three types of data dependencies:

Flow dependency An instruction depends on the result of a previous instruction (read-after-write), e.g. $A = 5; B = A + 1;$

Anti-dependency An instruction requires a value that is later updated (write-after-read), e.g. $A = 5; B = A + 1; A = 1;$

Output dependency Ordering of instructions will affect the final output value of a variable, e.g. $A = 5; A = 1;$

The last two dependencies can be removed by using unique variable names.

Loop Vectorization

- The Loop Vectorizer rewrites loops to reduce the number of total operations.
- The idea is to "widen" instructions in loops to operate on multiple consecutive iterations.
- Optimizes the inner most loop.

Loop Vectorization
Example

- Let's consider the previous C loop, but now with 1024 entries.
- Each 256 bit vector register can contain $256/32 = 8$ integers with 32 bits.
  - ➔ We can convert the loop into one with $512/8 = 64$ iterations.

```
int a[512]; int b[512];
for (i=0; i < 512; i += 8)
    a[i:i+7] = a[i:i+7] + b[i:i+7];
```

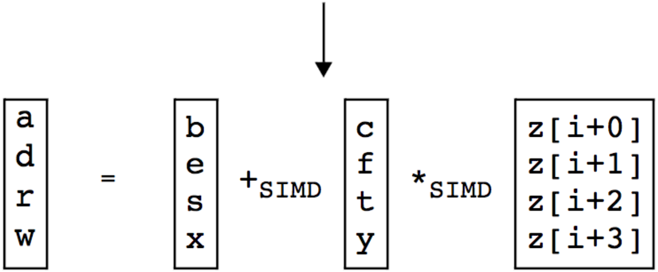
Superword Level Parallelismn

- Can exploit possible parallelism of inline code in a code block.
- Combine similar independent instructions into vector operations.
- Can unroll loops and optimize the inline code.
- Uses the control flow graph and looks into basic blocks for instructions to combine.

Superword Level Parallelismn
Example

The following statements are contained in the same block and are structurally the identical (isomorphic). They are then parallized by a technique called *statement packing*.

```
a = b + c * z[i+0]
d = e + f * z[i+1]
r = s + t * z[i+2]
w = x + y * z[i+3]
```



Superword Level Parallelismn
SLP Vectorizer Steps

- As previously stated it uses the codes basic blocks to look for scalar instructions to combine.
- The Vectorizers processes the code from the bottom-up
- The SLP vectorizers uses the following stages
  1. Identify potential instruction patterns to vectorize
  2. Determine if it is profitable to vectorize the code
  3. Vectorize the code

Summary

Summary

- Code Generation
  - ➔ The code generation process optimizes code for the target platform.
  - ➔ Can replace unsupported operations and data types.
  - ➔ Allows compiler frontends to avoid having to deal with target specific constraints.
- Auto Vectorization
  - ➔ Promises speedups for a lot of multimedia computations.
  - ➔ The compiler frontend doesn't need to support vector types on it's own.
  - ➔ Can optimize reasonably complex loops including control flow.
  - ➔ Can greatly improve the energy efficency.

Reference

- The LLVM documentation `http://llvm.org/docs/`
- S. Larsen and S. Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation, Vancouver, B.C., June 2000
- M. Pandey and S. Sarda. LLVM Cookbook. Packt Publishing, May 2015
- Eli Bendersky. A deeper look into the LLVM code generator, Article online `http://eli.thegreenplace.net/2013/02/25/a-deeper-look-into-the-llvm-code-generator-part-1`

Thank you for your kind attention!

Simon Grätzer – simon.graetzer@rwth-aachen.de

`http://graetzer.org/`