Branching in Data-Parallel Languages using Predication with LLVM

Marcello Maggioni

Codeplay Software Ltd.

EuroLLVM 2014

Outline

Outline

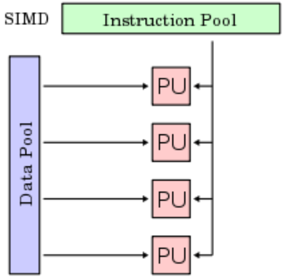
Data-Parallel Languages
OpenCL, CUDA, Renderscript ...

- Heavily parallel
- Many threads running the same code/program on a varying dataset
- SIMD-architecture friendly

Outline

SIMD Architectures

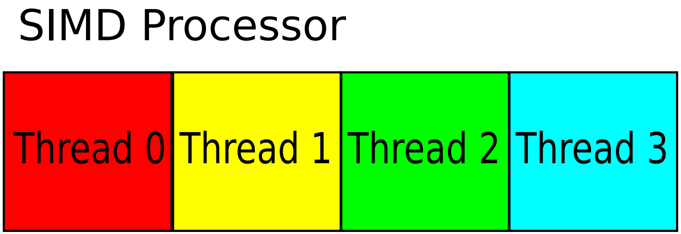
- Heavily Parallel
- Very efficient at running data parallel workloads with uniform control flow
- Very common today (Today's CPUs all have SIMD capabilities. Most GPUs are SIMD at heart)



Outline

SIMD + Data-Parallel Approach

- Each SIMD processing unit (PU) runs a different Data-Parallel thread

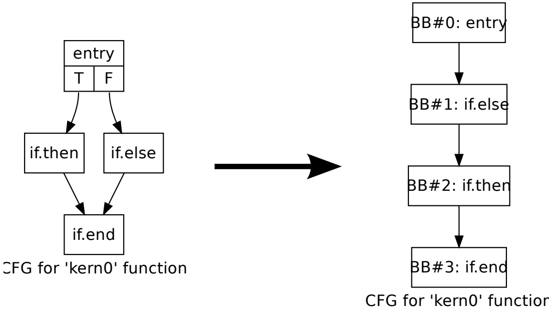


SIMD + Data-Parallel: Challenges

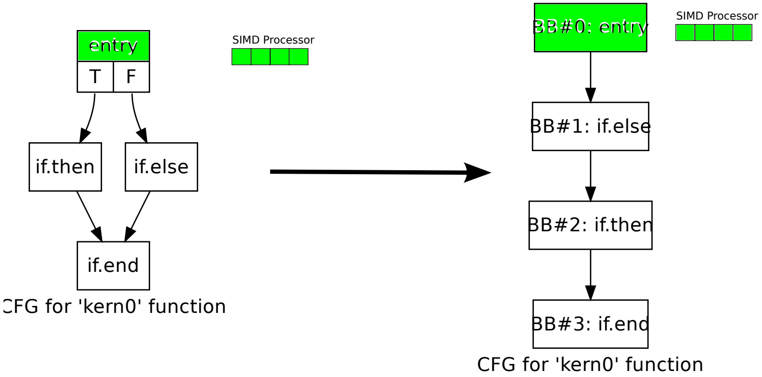
- Divergent branching happens when different SIMD PUs want to follow different code paths
- Needs special handling on many SIMD hardware as each individual unit is not independent.
- SIMD units share the same PC (need to execute the same instructions)

Branching on SIMD: Principles

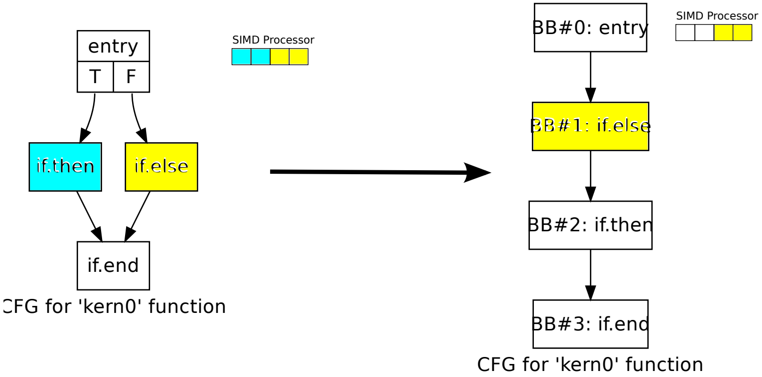
- We want to auto-vectorize the entire instruction stream over all the PUs of the SIMD Processor.
- Linearize the entire CFG after register allocation
- After linearization Basic Blocks that shouldn't run on a certain SIMD PU should have the execution of the instructions in that Basic Block disabled



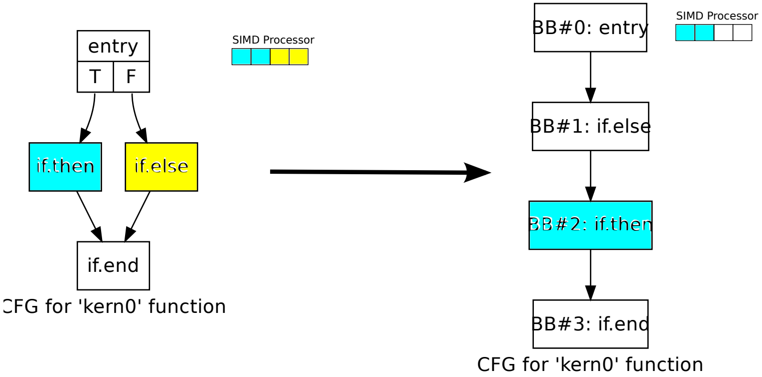
Branching on SIMD: Principles



Branching on SIMD: Principles



Branching on SIMD: Principles



Approaches

- IR-approach
  - Preferred if the architecture doesn't support full-predication
  - Needs special handling for side-effected instructions (trapped instructions, function calls ...)
- Backend-approaches
  - Can make full use of the features of the hardware
  - Hardware predication can be exploited

Predication

- Predication is an hardware feature that conditionally disables side effects of instructions

```
cmp m0, r0, r1
setmask m0
// Execute only if mask for unit is true
addvp r0, r1, r2
```

Outline

Predicating instructions

- Predicable instructions are defined in TableGen with an additional predicate operand in the backend
- The predicate operand has a default value which typically equals to the "always execute" predicate

Predicating instructions (2)

```
def LDRT_POST
  : ARMAsmPseudo<"ldrt${q}␣$Rt,␣$addr",
                  (ins addr_offset_none:$addr, pred:$q),
                  (outs GPR:$Rt)>;
```

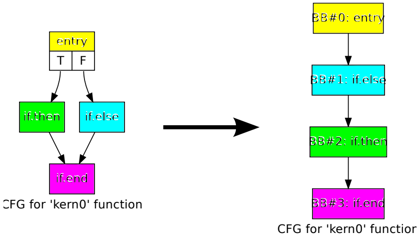
Predicating instructions (3)

```
// ARM Predicate operand. Default to 14 = always (AL). Second part is CC
// register whose default is 0 (no register).
def CondCodeOperand : AsmOperandClass { let Name = "CondCode"; }
def pred : PredicateOperand<OtherVT, (ops i32imm, i32imm),
                                (ops (i32 14), (i32 zero_reg))> {
  let PrintMethod = "printPredicateOperand";
  let ParserMatchClass = CondCodeOperand;
  let DecoderMethod = "DecodePredicateOperand";
}
```

Outline

Determining an execution schedule

- The execution schedule is the order of execution in the linearized CFG.
- The schedule needs to be chosen such that every possible predecessor of a BB is executed before the BB itself
- The Reverse Post Order traversal of the CFG follows this rule (Can use the Reverse Post Order Iterator from LLVM)

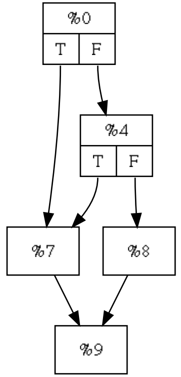


Determining an execution schedule (2)

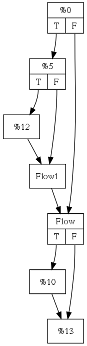
- Structurizing the CFG in this phase generates naturally a valid execution schedule and simplifies later passes.
- Can be done using the StructurizeCFG pass from LLVM

StructurizeCFG

```
int main() {
  volatile int a = 5;
  volatile int b = 6;

  if (a == 5 || b < 2) {
    b = 6;
  } else {
    b = 7;
  }

  return 0;
}
```



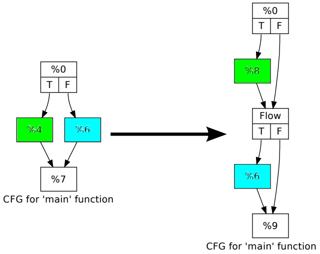
CFG for 'main' function



CFG for 'main' function

StructurizeCFG (2)

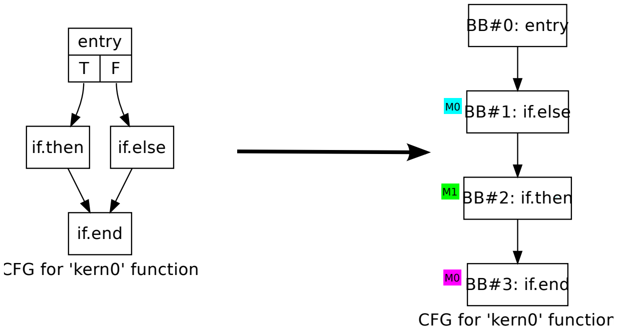
```
int main() {

  volatile int a = 5;
  volatile int b = 6;

  if (a < 5) {
    a = 5;
  } else {
    b = 10;
  }


  return 0;

}
```



Outline

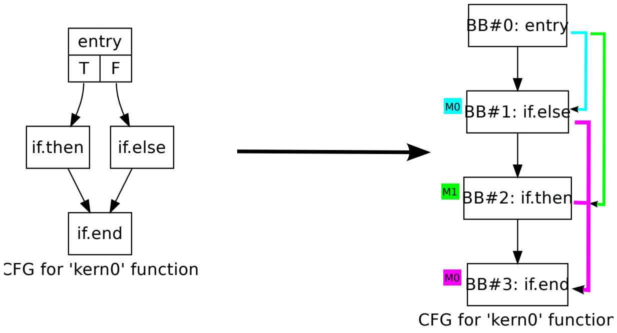
Execution Mask Allocation

- To each basic-block is associated an execution mask.
- Each execution mask needs to be stored and retrieved at the appropriate time



Execution Mask Life

- A Mask resource should be alive until used by the BB it is associated to
- In a loop masks should remain alive until the end of such loop



Execution Mask Allocation Strategies

1. One register per mask (using LLVM virtual registers)
   - Needs the CFG to have being structurized to have consistent allocation of registers.
   - Might be wasteful using unnecessary registers
2. Custom Mask allocation

Custom Mask Allocation

- Requires you to allocate Mask resources manually.
- Depending on the architecture you might be able to pack Masks more tightly
- Might need to reserve some registers to use as Mask registers
- Might need to implement some way to save and restore these registers

Outline

Mask Handling Insertion

- Code at the beginning of a basic block to load the mask into hardware is needed
- Code to store the generated masks for successors basic block is needed
- This code is better being inserted Post-RA usually

```
bb0:
cmpvp m0, r0, r1
notvp m1, m0

bb1:
setmask m0 // Loads mask from
ormask m0  // register and zeroes
           // out the register
// CODE OF ELSE BRANCH

bb2:
setmask m1
ormask m0
// CODE OF THEN BRANCH

bb3:
setmask m0
// CODE OF BB3
```

Why Mask manipulation post-RA?

- LLVM already run the spiller and PHI-elimination
- Much safer to insert your code now that everything is almost finalized
- If you need to emit a sequence of code nothing can get in between two emitted instructions

Outline

CFG Linearization

- If needed reorder the basic-blocks to the schedule order determined originally
- Remove all the branches
- Keep only back-edges
    - Condition for backedge jumps should be "if no other unit is active in the loop"

Outline

Predicate Instructions

- The predicate of instructions needs to be changed from “Always” to “Execute if predicate true”
- All instructions should be predicated
    - Mask manipulation instructions shouldn’t be predicated
    - Recognize them and don’t predicate them or predicate instructions before inserting them

Outline

Optimizations

- Avoid control flow linearization is the main optimization:
    - Static approach
    - Runtime approach
- The best is probably a combination of the two approaches if possible

Static optimizations

- Require analysis of the CFG to identify branches that are 100% uniform.
- Is simpler to do if the CFG is structurized.
- Avoid insertion of Mask manipulation instructions and saves Mask resources
- Might miss some cases (same old problem of static optimizations)

Runtime optimizations

- Available if the instruction set supports at least conditional branches over execution conditions of other units
- Quite simple to add to your implementation
- Doesn't remove the need for execution mask allocation
- Mask allocation code still needs to be added/executed
- Potentially catches more cases

Runtime optimizations (2)

```
bb0:
cmpvp m0, r0, r1
notvp m1, m0

bb1:
setmask m0 // Loads mask from
ormask m0  // register and zeroes
           // out the register
// CODE OF ELSE BRANCH

bb2:
setmask m1
ormask m0
// CODE OF THEN BRANCH

bb3:
setmask m0
// CODE OF BB3
```

```
bb0:
cmpvp m0, r0, r1
notvp m1, m0

bb1:
setmask m0 // Loads mask from
ormask m0  // register and zeroes
           // out the register
jallz bb2
// CODE OF ELSE BRANCH

bb2:
setmask m1
ormask m0
jallz bb3
// CODE OF THEN BRANCH

bb3:
setmask m0
// CODE OF BB3
```

Outline

Advantages

- Low register pressure thanks to post-RA CFG linearization
- Easy to implement optimization because of access to all the hardware features
- Support for both structured and unstructured CFGs
- Integrates well with LLVM infrastructure
    - No need to tweak register allocation
    - No need to tweak existing LLVM passes

Summary

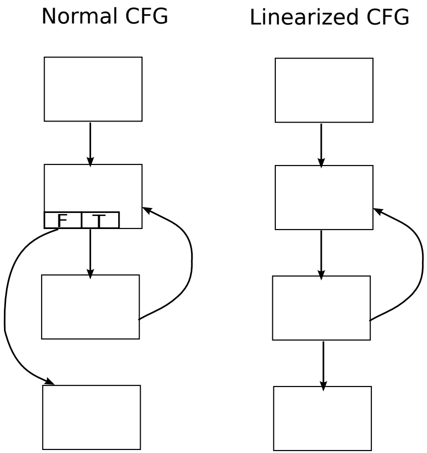
1. Introduction
   - Data-Parallel Languages
   - SIMD Architectures
   - SIMD + Data-Parallel Approach
2. Implementation
   - Predicating Instructions
   - Determining an execution schedule
   - Execution mask allocation
   - Mask Handling Insertion
   - CFG Linearization
   - Predicate instructions
   - Optimizations
   - Advantages

Contacts

Marcello Maggioni

- marcello.maggioni@gmail.com
- marcello@codeplay.com

Loop Example


Normal CFG
Linearized CFG
F
T