Dealing with Register Hierarchies

Matthias Braun (MatzeB) / LLVM Developers' Meeting 2016
Register Allocation

- Rewrite program with unlimited number of virtual registers to use actual registers

- Techniques: Interference Checks, Assignment, Spilling, Splitting, Rematerialization

```assembly
%0 = const 5
%1 = const 7
%2 = add %0, %1
return %2
```

```assembly
r0 = const 5
r1 = const 7
r0 = add r0, r1
return r0
```
Register Allocation for GPUs

- Hundreds of registers available, but using fewer increases parallelism
- Mix of Scalar (single value) and Vector (multiple values) operations
- Load/Store instructions work on multiple registers (high latency, high throughput)

```
r[0:3] = load_x4  # Load r0, r1, r2, r3
r4 = add r0, 1
r5 = add r1, 2
r6 = add r2, 3
r7 = add r3, 4
store_x4 r[4:7]  # Store r4, r5, r6, r7
```
Liveness Tracking

- Linearize program
- Number instructions consecutively (SlotIndexes)

```
b1:  %1 = const 5
    jmp b3

b2:  store %0
    %1 = def

b3:  2% = add %1, 1

%0 = def
cmp ...
jeq b2
```
Liveness Tracking

- Linearize program
- Number instructions consecutively (SlotIndexes)

```
0 %0 = def
1 cmp ... 
2 jeq b2

3 b1:  
4 %1 = const 5 
5 jmp b3

6 b2:  
7 store %0  
8 %1 = def 

9 b3: 
10 2% = add %1, 1
```
Liveness Tracking

- Linearize program
- Number instructions consecutively (SlotIndexes)
- Liveness as sorted list of intervals (segments)

...%1: [4:6] [8:9] [9:10] ...

```
0 %0 = def
1 cmp ...
2 jeq b2
3 b1:
4 %1 = const 5
5 jmp b3
6 b2:
7 store %0
8 %1 = def
9 b3:
10 2% = add %1, 1
```
Modeling Register Hierarchies
Tuple Registers

\[
\begin{align*}
r[0:3] &= \text{load}_x4 \\
r4 &= \text{add } r0, 1 \\
r5 &= \text{add } r1, 2 \\
r6 &= \text{add } r2, 3 \\
r7 &= \text{add } r3, 4 \\
\text{store}_x4 &= r[4:7]
\end{align*}
\]
Tuple Registers

%0, %1, %2, %3 = load_x4
%4 = add %0, 1
%5 = add %1, 2
%6 = add %2, 3
%7 = add %3, 4
store_x4 %4, %5, %6, %7
Tuple Registers

No relation between virtual registers but need to be consecutive
%0 = load_x4
%1.sub0 = add %0.sub0, 1
%1.sub1 = add %0.sub1, 2
%1.sub2 = add %0.sub2, 3
%1.sub3 = add %0.sub3, 4
store_x4 %1
Tuple Registers

- Register class contains tuples
- Allocator picks a single (tuple) register
- Parts called subregisters or lanes
- Select parts with subregister index (\textit{xxx} Syntax)

\begin{verbatim}
%0 = load_x4
%1.sub0 = add %0.sub0, 1
%1.sub1 = add %0.sub1, 2
%1.sub2 = add %0.sub2, 3
%1.sub3 = add %0.sub3, 4
store_x4 %1
\end{verbatim}
Construction

- `reg_sequence` defines multiple subregisters (for SSA) (there is also `insert_subreg`, `extract_subreg`)

```c
%0 = load
%1 = const 42
%2 = reg_sequence %0, sub1, %1, sub0
store_x2 %2
```
Construction

- `reg_sequence` defines multiple subregisters (for SSA) (there is also `insert_subreg`, `extract_subreg`)

- TwoAddressInstruction pass translates to `copy` sequence
Construction

- `reg_sequence` defines multiple subregisters (for SSA) (there is also `insert_subreg`, `extract_subreg`)
- TwoAddressInstruction pass translates to `copy` sequence
- RegisterCoalescing pass eliminates copies
Improving Register Allocation
Subregister Liveness

%0 = load_x4
%1.sub0 = add %0.sub0, 1
%1.sub1 = add %0.sub1, 2
%1.sub2 = add %0.sub2, 3
%1.sub3 = add %0.sub3, 4
store_x4 %1
Subregister Liveness

%0 = load_x4
%1.sub0 = add %0.sub0, 1
%1.sub1 = add %0.sub1, 2
%1.sub2 = add %0.sub2, 3
%1.sub3 = add %0.sub3, 4
store_x4 %1
Can allocate v0 and v1 to the same register tuple
Subregister Liveness: Lane Masks

- **Lane Mask**: 1 bit per subregister
- Annotate subregister liveness parts with lane mask
- Start with whole virtual register; Split and refine as necessary
Subregister Liveness: Lane Masks

- **Lane Mask**: 1 bit per subregister
- Annotate subregister liveness parts with lane mask
- Start with whole virtual register; Split and refine as necessary

Lane Masks:
- sub0: 0b0001
- sub1: 0b0010
- sub2: 0b0100
- sub1_sub2: 0b0110
- sub3: 0b1000
- all: 0b1111

```
%0 = load_x4
store_x4 %0

%1 = load_x4
%1.sub0 = const 13
%1.sub3 = const 42
store_x4 %1
```
Subregister Liveness: Lane Masks

• **Lane Mask**: 1 bit per subregister

• Annotate subregister liveness parts with lane mask

• Start with whole virtual register; Split and refine as necessary

Lane Mask: %0 = load_x4, store_x4 %0
%1 = load_x4, %1.sub0 = const 13, %1.sub3 = const 42, store_x4 %1

Lane Masks:
- sub0: 0b0001
- sub1: 0b0010
- sub2: 0b0100
- sub1_sub2: 0b0110
- sub3: 0b1000
- all: 0b1111
Subregister Liveness: Lane Masks

- **Lane Mask**: 1 bit per subregister
- Annotate subregister liveness parts with lane mask
- Start with whole virtual register; Split and refine as necessary

Lane Masks:

- `sub0`: 0b0001
- `sub1`: 0b0010
- `sub2`: 0b0100
- `sub1_sub2`: 0b0110
- `sub3`: 0b1000
- `all`: 0b1111

%0 = load_x4
store_x4 %0

%1 = load_x4
%1.sub0 = const 13
%1.sub3 = const 42
store_x4 %1
Assignment Heuristics

- Default: Assign in program order
Assignment Heuristics

• Default: Assign in program order
Assignment Heuristics

- Default: Assign in program order
Assignment Heuristics

- Default: Assign in program order
- Wide pieces may not fit in holes left by small ones
Assignment Heuristics

- Default: Assign in program order
- Wide pieces may not fit in holes left by small ones
Assignment Heuristics

- Default: Assign in program order
- Wide pieces may not fit in holes left by small ones
Assignment Heuristics

- Default: Assign in program order
- Wide pieces may not fit in holes left by small ones
- Tweak: Prioritize bigger classes
Assignment Heuristics

- Default: Assign in program order
- Wide pieces may not fit in holes left by small ones
- Tweak: Prioritize bigger classes
Assignment Heuristics

- Default: Assign in program order
- Wide pieces may not fit in holes left by small ones
- Tweak: Prioritize bigger classes
Interference Checks: Register Units

• Tuples multiply number of registers

• Interference check of single register in target with 1-10 tuples: 45 aliases!
Interference Checks: Register Units

- Each register mapped to one or more units:
  Registers alias iff they share a unit

- Liveness/Interference checks of actual registers uses register units
Usage, Results, Future Work
Use in LLVM

- Declare Subregister Indexes + Subregisters in XXXRegisterInfo.td
- TableGen computes register units and combined subregister indexes/classes
- Enable fine grained liveness tracking by overriding `TargetSubtargetInfo::enableSubRegLiveness()`
- `AllocationPriority` part of register class specification
Results: Apple GPU Compiler

- Compared various benchmarks and captured application shaders
- Average 20% reduction in register usage (-6% up to 50%)!
- Speedup 2-3% (-4% up to 70%)
Results: AMDGPU Target

raedonsi LLVM Performance Improvements

Posted on January 1, 2015

I just pushed up a new branch to my LLVM repo that enables two important LLVM codegen features (machine scheduling and subreg liveness) for SI+ targets, which should improve performance of the radeonSI driver.

The biggest improvement that I'm seeing with this branch is the luxmark luxball OpenCL demo which is about 60% faster on my Bonaire. Other tests I've done show 10% – 25% improvements in performance. I haven't done much OpenGL benchmarking, but I expect these changes will have much bigger impact on the OpenCL benchmarks, so OpenGL improvements may be in the lower end of that range. I still need more benchmark results to know for sure.
Results: AMDGPU Target

radeonsi LLVM Performance Improvements

Posted on January 1, 2015

I just pushed up a new branch to my LLVM repo that enables two important LLVM codegen features (machine scheduling and subreg liveness) for SI+ targets, which should improve performance of the radeonsi driver.

about 60% faster on my Bonaire. Other tests I’ve done show 10% - 25% improvements in performance. I haven’t done much OpenGL benchmarking, but I expect these changes will have much bigger impact on the OpenCL benchmarks, so OpenGL improvements may be in the lower end of that range. I still need more benchmark results to know for sure.
Future Work

• Support partially dead/undef operands
• Early splitting and rematerialization (before register limit)
• Partial registers spilling
• Consider partial liveness in register pressure tracking
• Missed optimizations (no obvious use/def relation for lanes)
Thank You for Your Attention!
Register Hierarchies

- CPU registers can overlap. Partial register accessible by subregister. Also called lanes (Vector Regs)

X86 GP Register

```
movw 0xABCD, %ax # Put 16bits into %ax
movb %al, x # Uses lower 8 bits: 0xCD
movb %ah, y # Uses upper 8 bits: 0xAB
```
Register Allocation Pipeline

- DetectDeadLanes
- ProcessImplicitDefs
- PHIElimination
- TwoAddressInstruction
- RegisterCoalescer
- RenameIndependentSubregs
- MachineScheduler
- RegAllocGreedy
- VirtRegRewriter
- StackSlotColoring
Subregister Indexes

- Subregister indexes relate wide/small registers on virtual registers.
- Writes may be marked undef if other parts of register do not matter.
- LLVM synthesizes combined indexes (`sub0_low16bits`).

Register Allocation:

```plaintext
%0 = load_x4
%1.sub0<undef> = add %0.sub2, 13
%1.sub1 = const 42
store_x2 %1
```

```plaintext
r4_r5_r6_r7 = load_x4
r0 = add r6, 13
r1 = const 42
store_x2 r0_r1
```
Slot Indexes

- Position in a program; Each instruction is assigned a number (incremented by 4 so we need to renumber less often when inserting instructions)

- Slots describe position in the instruction:
  - Block/Base (Block begin/end, PHI-defs)
  - EarlyClobber (early point to force interference with normal def/use)
  - Register (normal def/uses use this)
  - Dead (liveness of dead definitions ends here)
Constraints & Classes

- A register class is set of registers; Models register constraints

- Class defined for each register operand of LLVM MI Instruction (MCInstrDesc)

- Each virtual register has a class