Agenda

- What is the R600 backend?
- Introduction to AMD GPUs
- R600 backend overview
- Future work
What is the R600 backend?

- Component of AMD’s Open Source GPU drivers.
  - Provides implementation of several popular APIs.
  - All AMD GPU generations are supported.
  - Collaborative effort between AMD and the Open Source community.

- Used for compiling GLSL and OpenCL™ C programs.

- It is not the AMDIL backend.
  - AMDIL backend used by proprietary driver for OpenCL™
  - R600 emits ISA, AMDIL emits low-level assembly language

- Why is it called R600?
  - We generally name our Open Source components after the first generation they support.

- Why use LLVM?
  - Reduces development time.
  - GPU programs are starting to look more like CPU programs.
  - Testing coverage.
Generic GPU Overview

- **Terms**
  - Thread - A single element of execution (OpenCL™ work item).
  - Wave - A group of threads that are executed concurrently.
  - Execution Unit - Where the code is run.
  - Compute Unit - A collection of execution units that share resources.
  - Vector component (vec.x, vec.y, vec.z vec.w).

- **GPU Architecture**
  - GPUs have hundreds or thousands of individual execution units.
  - Execution units are grouped together into compute units.
  - Compute unit resources are shared among execution units.

- **Control Flow**
  - All threads in a wave share a program counter - branching is not always possible.
  - Control flow implemented using execution masks.
  - Only structure control flow is supported.
AMD GPU Overview

- Two distinct architectures supported by R600 backend:
  - VLIW4/VLIW5
  - Graphics Core Next (GCN)
- Within each architecture there are different GPU 'generations':
  - VLIW4/VLIW5 (R600, R700, EvergreenNI, Cayman)
  - GCN (Southern Islands, Sea Islands)
- For generations with the same architecture, the ISA is 95% the same, but not compatible.
- Each generation contains several variants.
- ISA is compatible between variants, but compiler must be aware of differences between variants in order to achieve optimal performance.
Control Flow Instructions

- Handle program flow (branches, loops, function calls).
- Used for writing data to global memory.
- Can initiate a clause.
  - Clause is a group of lower-level instructions.
  - Three types of clauses (ALU, Texture, Vertex).
  - Each clause can execute a limited number of instructions.
### VLIW4/VLIW5 ALUs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Source(s)</th>
<th>Target(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_INT</td>
<td>T1.Z, PV.W, PS</td>
<td></td>
</tr>
<tr>
<td>BIT_ALIGN_INT</td>
<td>T3.W, T2.W, T2.W, literal.y, BS:VEC_201</td>
<td></td>
</tr>
<tr>
<td>LSHR</td>
<td>*T4.W, T2.W, literal.z, 7(9.809089e−45), 19(2.662467e−44) 10(1.401298e−44), 0(0.000000e+00</td>
<td></td>
</tr>
</tbody>
</table>

- 4 or 5 wide depending on the variant.
- Can execute 4 or 5 *different* instructions at once.
- ALU.X, ALU.Y, ALU.Z, ALU.W, ALU.TRANS (VLIW5 only).
- ALU.X may only write to X component, ALU.Y to Y, etc.
- ALU.TRANS can write to any component.
- 3 Classes of instructions:
  - Any - ALU.[XYZW] or ALU.Trans
  - Vector - ALU.[XYZW] Only
  - Scalar - ALU.Trans Only

7 | A Detailed Look at the R600 Backend | November 5, 2013
## VLIW4/VLIW5 Instruction Inputs

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Input 1</th>
<th>Input 2</th>
<th>Input 3</th>
<th>Input 4</th>
<th>Input 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_INT</td>
<td>T1.Z, PV.W, PS,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BIT_ALIGN_INT</td>
<td>T3.W, T2.W, T2.W, literal.y, BS:VEC_201</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSHR</td>
<td>T4.W, T2.W, literal.z, 7(9.809089e−45), 19(2.662467e−44)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10(1.401298e−44), 0(0.000000e+00)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Literal Constants**
- **Vector Registers**
  - 128 <4 x 32 bit> Registers
  - Most instruction write to one component of the vector (e.g. T0.X or T0.Y).
  - No data dependency between components of the same vector.
- **Constant Registers**
  - Used to access values in the constant memory cache.
  - Cache is filled at the beginning of each ALU clause.
VLIW4/VLIW5 Source Restrictions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Sources</th>
<th>Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_INT</td>
<td>T1.Z, PV.W, PS,</td>
<td></td>
</tr>
<tr>
<td>BIT_ALIGN_INT</td>
<td>T3.W, T2.W, T2.W, literal.y, BS:VEC_201</td>
<td></td>
</tr>
<tr>
<td>LSHR</td>
<td>* T4.W, T2.W, literal.z,</td>
<td>7(9.809089e-45), 19(2.662467e-44)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10(1.401298e-44), 0(0.000000e+00)</td>
</tr>
</tbody>
</table>

- There are a lot of restrictions.
- Loading of inputs takes place over 3 cycles.
- On each cycle only one GPR.X, GPR.Y, GPR.Z, and GPR.W value can be read.
- Order of source fetches must be specified by the compiler writer.
Differences from VLIW4/VLIW5

- Control Flow instructions replaced by "Scalar" ALU.
- Two different ALU types: "Scalar" and "Vector".
- Scalar registers.
- Compiler manages the execution mask.
GCN - ALU Types

- **SALU**
  - One per wave.
  - Responsible for control flow.
  - Limited instruction set.
  - 102 32-bit registers (Scalar Registers).

- **VALU**
  - One VALU per thread in a wave (64 VALUs per wave).
  - Complete instruction set.
  - 256 32-bit register (Vector Registers).

- Programs can intermix SALU and VALU instructions.
- Instructions are always executed in sequence regardless of ALU type.
- VALU can directly access SALU registers.
- Copying data from VALU registers to SALU registers is not always possible.
GCN

S_LOAD_DWORD  SGPR2, SGPR0_SGPR1, 11
S_LOAD_DWORD  SGPR3, SGPR0_SGPR1, 12
S_WAITCNT  lgkmcnt(0)
V_MOV_B32_e32  VGPR0, SGPR3
V_ADD_F32_e64  VGPR0, SGPR2, VGPR0, 0, 0, 0, 0
S_LOAD_DWORDX2  SGPR0_SGPR1, SGPR0_SGPR1, 9
S_MOV_B64  SGPR4_SGPR5, 0
S_MOV_B32  SGPR6, 0
S_MOV_B32  SGPR7, 61440
S_WAITCNT  lgkmcnt(0)
V_MOV_B32_e32  VGPR1, SGPR0
V_MOV_B32_e32  VGPR2, SGPR1
BUFFER_STORE_DWORD  VGPR0, SGPR4_SGPR5_SGPR6_SGPR7 +
                     VGPR1_VGPR2 + 0
S_ENDPGM

Variable pointer sizes.
  ▶ 64-bit for global / constant memory.
  ▶ 32-bit for local memory (LDS).
  ▶ 128-bit, 256-bit, 512-bit resource descriptors for texture / buffer instructions.
Instruction Operands

- **VLIW4/VLIW5 instructions** have a large number of operands.
- **Most operands** are configuration bits for the instruction:
  - **Modifiers for instruction inputs outputs:**
    - Inputs: ABS, NEG
    - Output: CLAMP, OMOD (Multiply floating-point result by a power of two)
  - **Predicate bits**
  - **Indirect addressing bits**
Instruction Operands

<table>
<thead>
<tr>
<th>UEM: $update_exec_mask</th>
<th>UP: $update_pred</th>
<th>WRITE: $write</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMOD: $omod</td>
<td>REL: $dst_rel</td>
<td>CLAMP: $clamp</td>
</tr>
<tr>
<td>R600_Reg32: $src0</td>
<td>NEG: $src0_neg</td>
<td>REL: $src0_rel</td>
</tr>
<tr>
<td></td>
<td>ABS: $src0_abs</td>
<td>SEL: $src0_sel</td>
</tr>
<tr>
<td>R600_Reg32: $src1</td>
<td>NEG: $src1_neg</td>
<td>REL: $src1_rel</td>
</tr>
<tr>
<td></td>
<td>ABS: $src1_abs</td>
<td>SEL: $src1_sel</td>
</tr>
<tr>
<td>LAST: $last</td>
<td>R600_Pred: $pred_sel</td>
<td></td>
</tr>
<tr>
<td>LITERAL: $literal</td>
<td>BANK_SWIZZLE: $bank_swizzle</td>
<td></td>
</tr>
</tbody>
</table>

How to match instructions with so many operands?

class OperandWithDefaultOps< ValueType ty , dag defaultops > : Operand< ty > {
   dag DefaultOps = defaultops;
}

def MUL_INT24 cm : R600_2OP <0x5B , "MUL_INT24" ,
   [( set i32:$dst , ( mul i24:$src0 , i24:$src1 ) )], VecALU >;
How to efficiently set ABS, NEG bits?

- Use ComplexPatterns?

```c
bool AMDGPUISelDAGToDAG::SelectSrc(SDValue Src,
                                   SDValue &Reg, SDValue &Abs, SDValue &Neg) const;
```

- This would be the ideal solution, however...
- It breaks instruction encoding.
- Does not work with stand-alone patterns.

- Post-process the DAG?
- This is what the R600 backend does
- It works, but...
- We need to write a lot of a custom code.
- Most of the code is duplicating things TableGen could do for us.
Accessing Operands

▶ How to figure which operand index maps to a configuration bit?
  ▶ Configuration bits may have a different index depending on the instruction.

▶ Solution:

```cpp
let UseNamedOperandTable = 1;
```

▶ Generates `getNamedOperandIdx()` function:

```cpp
int16_t getNamedOperandIdx(uint16_t Opcode,
                           uint16_t NamedIdx);
```

▶ You can query the operand index using the operand names defined in TableGen.

```cpp
int AbsIdx = AMDGPU::getNamedOperandIdx(
    AMDGPU::ADD,
    AMDGPU::OpName::src0_abs);

MI.getOperand(AbsIdx).setImm(1);
```
Indirect Addressing

- Instructions may use the address registers to indirectly access any register.
  - For Example: ADD T0.X, T[3 + ADDR].X, T0.
  - Used for accessing arrays stored in registers.
  - Makes optimization difficult.
- Solution 1:
  - Assign a virtual register to each item in they array.
  - If an instruction uses indirect addressing for its result have it implicitly define all items in the array.
  - If its uses indirect addressing for sources, implicitly use all items.
  - Use REG_SEQUENCE to fit the array into GPRs.
  - Advantage: Produces highly optimized code.
  - Disadvantages: Requires tracking uses and defs through basic blocks.
Indirect Addressing

Solution 2:
- Reserve a block of GPRs for a ’register address space’.
- Use loads and stores to model indirect addressing.
- Lower loads and stores to ALU instructions after register allocation.
- Advantage: Easy to implement.
- Disadvantage: Produces inefficient code.
- This is the solution we are using for OpenCL™ C programs

Solution 3:
- Model arrays using vectors, rather than alloca, load, store.
- Advantages:
  - We can accurately track the live range for arrays.
  - Register allocator can allocate registers for arrays.
- Disadvantage:
  - For OpenCL™ C, we must convert array allocas to vectors.
  - We require larger vector sizes than TableGen supports.
- We are using this solution for GLSL shaders on GCN hardware.
Problem:
  - Two ALUs (SALU and VALU) with different by intersecting instruction sets.
  - Data flows only one way: SALU to VALU.
  - How do we tell the ISel pass which instructions to use?

Best solution would be if ISel could select the instruction based on the register classes.

Current solution:
  - Only write TableGen patterns for SALU instructions.
  - Add a pass to move instruction from VALU to SALU to satisfy data dependencies.
R600 Scheduling

- Scheduling is complicated due to:
  - VLIW packet source restrictions.
  - Different kinds of instruction clauses (Alu, Vertex, Texture).
- Minimizing register usage is very important.
  - There is one register pool per compute unit.
  - The hardware allocates registers for each thread from this pool.
  - A thread can use at most 128 \(<4 \times 32\) bit\> registers, but...
  - There are not enough registers for all threads to use the maximum.
  - For optimal utilization of compute units, the maximum number of registers is much smaller.
  - The actual number depends on the variant.
R600 Scheduling

- We need to switch scheduling strategies once we reach the 'utilization maximum'.
  - We have basic register pressure tracking to help us schedule texture/vertex instructions.
  - We do not currently take advantage of MachineScheduler’s register pressure tracking.
Future Work

- Support for new hardware.
- Full support for GPU programming languages: OpenCL™ C, GLSL.
- Other ideas:
  - MachineScheduler for GCN
  - Common intrinsics for GLSL (LunarGLASS?)
  - SelectionDAG replacement?
  - Backend error reporting
  - Performance Improvements
- More GPU backends in LLVM!
Resources

Installation guide for Open Source compute with R600 backend:
  ▶ http://dri.freedesktop.org/wiki/GalliumCompute/

GPU ISA Documentation
  ▶ http://www.x.org/docs/AMD/

Mesa3D (Userspace driver):
  ▶ http://www.mesa3d.org/

LunarGlass:
  ▶ http://www.lunarglass.org/

Where to ask questions:
  ▶ Mesa mailing list - mesa-dev@lists.freedesktop.org
  ▶ Mesa IRC channels - #radeon, #dri-devel on irc.freenode.net
  ▶ LLVM mailing list - llvmdev@cs.uiuc.edu
  ▶ LLVM IRC channel - #llvm on irc.oftc.net