Optimizing HIP Assembly

Solving performance regressions through comparative analysis
(on AMD GPUs)

Outline

Some questions:

- HIP Assembly, huh?
- *Why* do we want to look at assembly?
- *How* can we optimize assembly from a high level?
- Case study: Compiler regression
- Questions & discussion

GPU Assembly & HIP

- Who has written a kernel with HIP or CUDA?
- Who has read assembly (x86, PTX/SASS, AMDGCN, etc.)?

GPU Assembly & HIP

- We write in a high-level language: C++/HIP (AMD's CUDA)
- The compiler compiles both host and device code
  - Device code compiles to AMDGCN (SASS in CUDA-land)
  - We're only interested in device code!
  - The compiler is a black box that *automagically* produces somewhat optimized assembly
- The quality of the assembly translates directly to performance

GPU Assembly & HIP

STREAM High Performance Computing

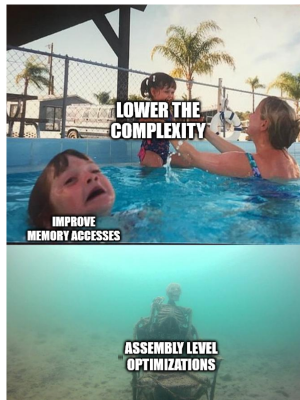
```
__global__ void
k(int* a_, int* b_, int* c_)
{
    int t = threadIdx.x;
    int a = a_[t];
    int b = b_[t];

    bool check = a & 1;
    c_[t] = check ? a : b;
}
```

```
s_load_dwordx4 s[0:3], s[4:5], 0x0
s_load_dwordx2 s[6:7], s[4:5], 0x10
v_lshlrev_b32_e32 v0, 2, v0
s_waitcnt lgkmcnt(0)
global_load_dword v1, v0, s[0:1]
global_load_dword v2, v0, s[2:3]
s_waitcnt vmcnt(1)
v_and_b32_e32 v3, 1, v1
v_cmp_eq_u32_e32 vcc, 0, v3
s_waitcnt vmcnt(0)
v_cndmask_b32_e32 v1, v1, v2, vcc
global_store_dword v0, v1, s[6:7]
```

But why dive in so deep?

STREAM
High Performance
Computing

- Exhausted high-level optimizations
- Performance regressions due to compiler changes
- It's cool and fun!
- **Understanding how code translates to optimized assembly**



It's not that bad I swear!

The idea

- There are multiple ways to express the same thing:
  - Both high-level code, and compiled assembly
- The compiler cannot explore all possible optimizations:
  - Compute, hardware, and algorithm limitations
  - Effectively a search problem
- If we reduce the steps the compiler needs to take to reach our optimal assembly, we are more likely to produce it.

The how

Requirements:

- Extracting assembly
  - `amdclang --save-temps`
  - `llvm-objdump`
- Knowing the ISA (Instruction Set Architecture)
  - E.g. CDNA3 ISA (AMD MI300)
- **A good diff tool**
  - Regex filter support
  - I use Meld

The how

Conceptually, it should be pretty easy:

1. Find two assemblies with (almost) similar behavior and different performance
2. Find the instructions that differ that do express the same thing
3. Reverse engineer how the faster one would be expressed in C++-land
4. Replace the code and benchmark!

Case Study: Compiler regression

We used this method in the real world![1]

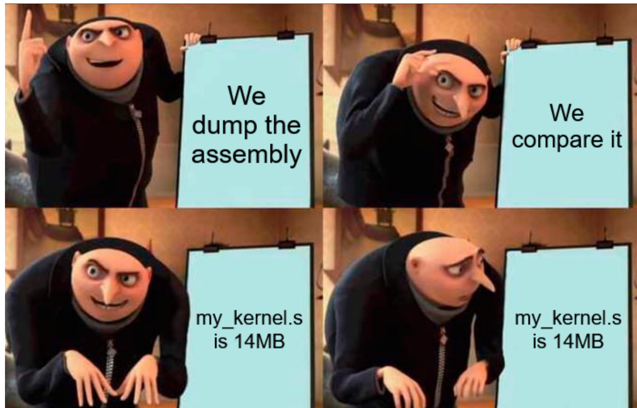
- 8% performance regression in a newer compiler
- Occupancy related
  - On new compiler we're using a few extra registers
  - Registers are a shared resource on GPU
  - Less parallelization means lower performance
- Goal: Find out the cause of this extra register usage (duh)

Should be easy, right? Just follow the plan!

[1] `https://github.com/ROCm/rocm-libraries/pull/2127`

Case Study: Compiler regression

STREAM
High Performance
Computing



Just like planned.

Case Study: Compiler regression

So it's not *that* easy:

- Problem 1: How do we identify where new registers are introduced?
- Problem 2: How do we spit through so much data?

Identifying extra register usage

STREAM
High Performance
Computing

- Vector registers (VGPR) are alloacted in ascending order
  - E.g. v1, v2, v3, v4
  - This also holds for scalar registers (SGPR)
- VGPRs may also be reused
  - E.g. we can write to v1 if v1 is no longer used!
- If a mechanism now requires an extra VGPR, all subsequent identifiers will be incremented
- Seems suspiciously monotonic

```
...
v_and_b32_e32 v11, 1, v10
; reusing registers:
v_mov_b32_e32 v10, 0x10
; allocate register:
v_mov_b32_e32 v12, 0x11
...
```

Identifying extra register usage

- Given two kernels, we can compare the VGPR indices get an indication if extra registers have been allocated before!

```
...
v_mov_b32_e32 v10 , 0x10
v_mov_b32_e32 v11 , 0x11

v_mov_b32_e32 v12 , 0x20
v_mov_b32_e32 v13 , 0x21
...
```

```
...
v_mov_b32_e32 v10 , 0x10
v_mov_b32_e32 v11 , 0x11
v_mov_b32_e32 v12 , 0xff
v_mov_b32_e32 v13 , 0x20
v_mov_b32_e32 v14 , 0x21
...
```

Identifying extra register usage

- Compiler changes can introduce other optimizations: noise!
- We are only interested in extra register usage
- Idea: binary search on the somewhat monotonic VGPR indices!

Using binary search

Begin halfway through the kernel:

```
...
v_add_f64 v[118:119], v[116:117], -v[118:119]
v_add_f64 v[114:115], v[114:115], -v[118:119]
v_add_f64 v[112:113], v[112:113], v[114:115]
v_add_f64 v[112:113], v[116:117], v[112:113]
...
```

```
...
v_add_f64 v[120:121], v[118:119], -v[120:121]
v_add_f64 v[116:117], v[116:117], -v[120:121]
v_add_f64 v[114:115], v[114:115], v[116:117]
v_add_f64 v[114:115], v[118:119], v[114:115]
...
```

Extra register must be introduced earlier with VGPR $<$ `v[114:115]`

Using binary search

STREAM
High Performance
Computing

Recurse the search into the first half:

```
...
v_cndmask_b32_e64 v99, v20, v18, s[0:1]
v_and_b32_e32 v98, 0x80000000, v96
v_xor_b32_e32 v96, v96, v93
v_and_b32_e32 v96, 0x80000000, v96
v_fma_f64 v[90:91], v[90:91], v[94:95], v[100:101]
v_cndmask_b32_e64 v94, v18, v20, s[0:1]
v_cndmask_b32_e64 v95, v19, v21, s[0:1]

...
```

```
...

v_and_b32_e32 v98, 0x80000000, v96
v_xor_b32_e32 v96, v96, v93
v_and_b32_e32 v96, 0x80000000, v96
v_fma_f64 v[90:91], v[90:91], v[94:95], v[100:101]
v_cndmask_b32_e64 v94, v20, v16, s[0:1]
v_cndmask_b32_e64 v95, v21, v17, s[0:1]
v_cndmask_b32_e64 v20, v16, v20, s[0:1]
...
```

Extra register introduced later with VGPR > `v[100:101]`

Found it!

STREAM High Performance Computing

```
...
s_mov_b32 s4, 0x1100000
v_mov_b32_e32 v108, 0x7ff80000
v_mov_b32_e32 v109, 0x7ff00000

v_mov_b32_e32 v110, 0xffffff80

s_waitcnt vmcnt(62)
v_mov_b32_e32 v0, v17
s_mov_b32 s95, 0x1200000
...
```

```
...
s_mov_b32 s4, 0x1100000
v_mov_b32_e32 v106, 0x9908b0df
v_mov_b32_e32 v107, 0x7ff00000
v_mov_b32_e32 v108, 0x100
v_mov_b32_e32 v109, 0xffffff80
v_mov_b32_e32 v110, 0x7ff80000
s_waitcnt vmcnt(62)
v_mov_b32_e32 v113, v15
s_mov_b32 s95, 0x1200000
...
```

Two new magic values are introduced here, why? Time to investigate `v106` and `v108`!

Investigating the cause

STREAM
High Performance Computing

```
...
s_or_b64 exec, exec, s[92:93]
v_cndmask_b32_e64 v16, v20, v16, s[2:3]
v_and_b32_e32 v20, 0x80000000, v6
v_and_or_b32 v20, v16, s33, v20
v_bfe_i32 v16, v16, 0, 1
v_and_b32_e32 v16, 0x9908b0df, v16

v_lshrrev_b32_e32 v20, 1, v20
v_xor_b32_e32 v16, v20, v16
v_xor_b32_e32 v95, v16, v80
...
```

```
...
s_or_b64 exec, exec, s[92:93]
v_cndmask_b32_e64 v0, v5, v0, s[2:3]
v_and_b32_e32 v5, 0x80000000, v6
v_and_or_b32 v5, v0, s33, v5
v_and_b32_e32 v0, 1, v0
v_cmp_eq_u32_e32 vcc, 1, v0
v_cndmask_b32_e32 v0, 0, v106, vcc
v_lshrrev_b32_e32 v5, 1, v5
v_xor_b32_e32 v0, v0, v5
v_xor_b32_e32 v93, v0, v79
...
```

- What is `0x9908b0df`? Magic number!
- What is the original code?

Investigating the cause

Identify the magic number in code, aligns with assembly:

```
u32 comp(u32 mt_i, u32 mt_i_1, u32 mt_i_m)
{
const u32 y
    = (mt_i   & mt19937_constants::upper_mask)
    | (mt_i_1 & mt19937_constants::lower_mask);
const u32 mag
    = (y & 0x1U)
    * mt19937_constants::matrix_a;
return mt_i_m ^ (y >> 1) ^ mag;
}
```

```
...
s_or_b64 exec, exec, s[92:93]
v_cndmask_b32_e64 v16, v20, v16, s[2:3]
v_and_b32_e32 v20, 0x80000000, v6
v_and_or_b32 v20, v16, s33, v20
v_bfe_i32 v16, v16, 0, 1
v_and_b32_e32 v16, 0x9908b0df, v16
v_lshrrev_b32_e32 v20, 1, v20
v_xor_b32_e32 v16, v20, v16
v_xor_b32_e32 v95, v16, v80
...
```

- The assembly represents the left expression
- However, could you infer the code from the assembly!

Reverse engineering

STREAM
High Performance Computing

What about the compiler with the regression?

```
u32 comp(u32 mt_i, u32 mt_i_1, u32 mt_i_m)
{
const u32 y
    = (mt_i   & mt19937_constants::upper_mask)
    | (mt_i_1 & mt19937_constants::lower_mask);
const u32 mag
    = (y & 0x1U)
    * mt19937_constants::matrix_a;
return mt_i_m ^ (y >> 1) ^ mag;
}
```

```
...
s_or_b64 exec, exec, s[92:93]
v_cndmask_b32_e64 v0, v5, v0, s[2:3]
v_and_b32_e32 v5, 0x80000000, v6
v_and_or_b32 v5, v0, s33, v5
v_and_b32_e32 v0, 1, v0
v_cmp_eq_u32_e32 vcc, 1, v0
v_cndmask_b32_e32 v0, 0, v106, vcc
v_lshrrev_b32_e32 v5, 1, v5
v_xor_b32_e32 v0, v0, v5
v_xor_b32_e32 v93, v0, v79
...
```

- Recall that `v106` is `0x9908b0df`
- Both assemblies represent the same expression!

Reverse engineering

STREAM High Performance Computing

How would we express the assembly on the right?

```
...
const u32 mag
    = ???
     ???;
...
```

```
...
s_or_b64 exec, exec, s[92:93]
v_cndmask_b32_e64 v16, v20, v16, s[2:3]
v_and_b32_e32 v20, 0x80000000, v6
v_and_or_b32 v20, v16, s33, v20
v_bfe_i32 v16, v16, 0, 1
v_and_b32_e32 v16, 0x9908b0df, v16
v_lshrrev_b32_e32 v20, 1, v20
v_xor_b32_e32 v16, v20, v16
v_xor_b32_e32 v95, v16, v80
...
```

- `v_bfe` (bit-field extract): broadcasts a single bit to all bits in register
- `v_and_b32`: bitwise 'and' operation

Reverse engineering

We can translate literally!

```
u32 comp(u32 mt_i, u32 mt_i_1, u32 mt_i_m)
{
const u32 y
    = (mt_i   & mt19937_constants::upper_mask)
    | (mt_i_1 & mt19937_constants::lower_mask);
const u32 mag
    = (y & 1 ? -1 : 0)
    & mt19937_constants::matrix_a;
return mt_i_m ^ (y >> 1) ^ mag;
}
```

```
...
s_or_b64 exec, exec, s[92:93]
v_cndmask_b32_e64 v16, v20, v16, s[2:3]
v_and_b32_e32 v20, 0x80000000, v6
v_and_or_b32 v20, v16, s33, v20
v_bfe_i32 v16, v16, 0, 1
v_and_b32_e32 v16, 0x9908b0df, v16
v_lshrrev_b32_e32 v20, 1, v20
v_xor_b32_e32 v16, v20, v16
v_xor_b32_e32 v95, v16, v80
...
```

- We could use an intrinsic for `bfe`, but not needed
- Does it actually work?

Results

STREAM
High Performance Computing

Yes! Regression vs. fix:

```
Kernel info:
codeLenInByte = 108508
NumSgprs: 100
NumVgprs: 129
NumAgprs: 0
TotalNumVgprs: 129
ScratchSize: 0
MemoryBound: 0
FloatMode: 240
IeeeMode: 1
LDSByteSize: 0 bytes/workgroup (compile time only)
SGPRBlocks: 12
VGPRBlocks: 32
NumSGPRsForWavesPerEU: 100
NumVGPRsForWavesPerEU: 129
Occupancy: 1
```

```
Kernel info:
codeLenInByte = 109176
NumSgprs: 100
NumVgprs: 128
NumAgprs: 0
TotalNumVgprs: 128
ScratchSize: 0
MemoryBound: 0
FloatMode: 240
IeeeMode: 1
LDSByteSize: 0 bytes/workgroup (compile time only)
SGPRBlocks: 12
VGPRBlocks: 31
NumSGPRsForWavesPerEU: 100
NumVGPRsForWavesPerEU: 128
Occupancy: 2
```

Results

- Benchmarks show regression restored.
- What about `v_mov_b32_e32 v108, 0x100`?

Results

What did it do? Regression vs. fix:

```
...
v_mov_b32_e32 v106, 0x9908b0df
v_mov_b32_e32 v107, 0x7ff00000
v_mov_b32_e32 v108, 0x100
v_mov_b32_e32 v109, 0xffffff80
v_mov_b32_e32 v110, 0x7ff80000
s_waitcnt vmcnt(62)
...
```

```
...
v_mov_b32_e32 v106, 0x3ff00000
v_mov_b32_e32 v107, 0x7ff00000

v_mov_b32_e32 v108, 0xffffff80
v_mov_b32_e32 v109, x7ff80000
s_waitcnt vmcnt(62)
...
```

- Why did the magic number of `0x9908b0df` change?
- Where did `0x100` go?

Results

Regression vs fix:

```
...
v_cmp_gt_f64_e32 vcc, s[58:59], v[7:8]
v_cndmask_b32_e32 v0, 0, v108, vcc

v_ldexp_f64 v[7:8], v[7:8], v0
v_cndmask_b32_e32 v0, 0, v109, vcc
...
```

```
...
v_cmp_gt_f64_e32 vcc, s[58:59], v[9:10]
v_cndmask_b32_e64 v0, 0, 1, vcc
v_lshlrev_b32_e32 v0, 8, v0
v_ldexp_f64 v[7:8], v[9:10], v0
v_cndmask_b32_e32 v0, 0, v110, vcc
...
```

- Recall `v108` was `0x100`: $1 << 8 =$ `0x100`
- Use extra registers for less instructions
  - Similar for other magic number
- We can get side effects! Not necessarily bad: benchmark!
- Food for thought: could the assembly be even more optimal?

Conclusion

- When dealing with compiler-related regressions, assembly diffs can be used to identify differing code.
- The compiler can be nudged to compile thins one way or another way by writing code in a certain way.
- We can reverse engineer assembly to build higher level code that resembles the final output better.

The end

Questions?

- *Can this be applied to CUDA?*
  Probably? I have not tried it with PTX nor SASS.
- *Can this be used to reduce inliner and loop-unroller costs?*
  Really want to try this! This is a main pain point in header-only GPU libraries!

Extra #1

Which is "faster"? [2]

```
// ... load a_, b_, c_
#pragma unroll
for(auto i = 0; i < 8; ++i) {
    const bool cond = c_[i] != 0;
    r_[i] = cond ? a_[i] : b_[i];
}
// ... store r_
```

```
// ... load a_, b_, c_
#pragma unroll
for(auto i = 0; i < 8; ++i) {
    const bool cond = c_[i] != 0;
    if (cond) {
        r_[i] = a_[i];
    } else {
        r_[i] = b_[i];
    }
}
// ... store r_
```

[2] https://godbolt.org/z/nY6evrebv

Extra #2

STREAM High Performance Computing

What...

```
; Kernel info:
; codeLenInByte = 292
; TotalNumSgprs: 20
; NumVgprs: 20
```

```
; Kernel info:
; codeLenInByte = 320
; TotalNumSgprs: 20
; NumVgprs: 24
```

Conclusion: ternaries are your friend :)