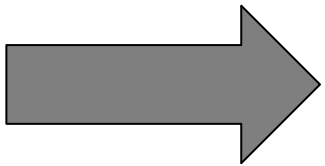
SSA-based Register Allocation for GPU Architectures

Connor Abbott, Daniel Schürmann (Valve)

SSA Form

```
if (...) {
   v1 = ...
} else {
   v1 = ...
}
```

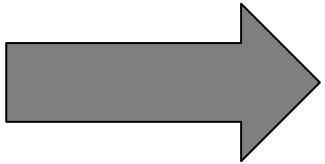


```
if (...) {
   v1_0 = ...
} else {
   v1_1 = ...
}
v1_2 = φ(v1_0, v1_1)
```

```
// PHINode - The PHINode class is used to represent the magical mystical PHI
// node, that can not exist in nature, but can be synthesized in a computer
// scientist's overactive imagination.
```

SSA Form: Deconstruction

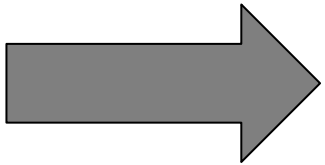
```
if (...) {
   v1_0 = ...
} else {
   v1_1 = ...
}
v1_2 = φ(v1_0, v1_1)
```



```
if (...) {
   v1_0 = ...
   v1 = v1_0
} else {
   v1_1 = ...
   v1 = v1_1
}
```

Register Allocation

```
v0 = load ...
v1 = load ...
v2 = load ...
v3 = add v0, v1
v4 = add v3, v2
```



```
r0 = load ...
r1 = load ...
r2 = load ...
r0 = add r0, r1
r0 = add r0, r2
```

Register Allocation: Optimality

- As few copies as possible?
- Less and well-placed spill code?
- Using as few registers as possible?
- Avoid pipeline stalls (RAW, WAR, …)?

Traditional Register Allocation

- first *deconstruct* SSA, then run Register Allocation
- Existing approaches: *graph-coloring, linear-scan*

Traditional Register Allocation

- Coalescing and Register Allocation are decoupled
- Spilling and Register Allocation are done at once

SSA-Based Register Allocation

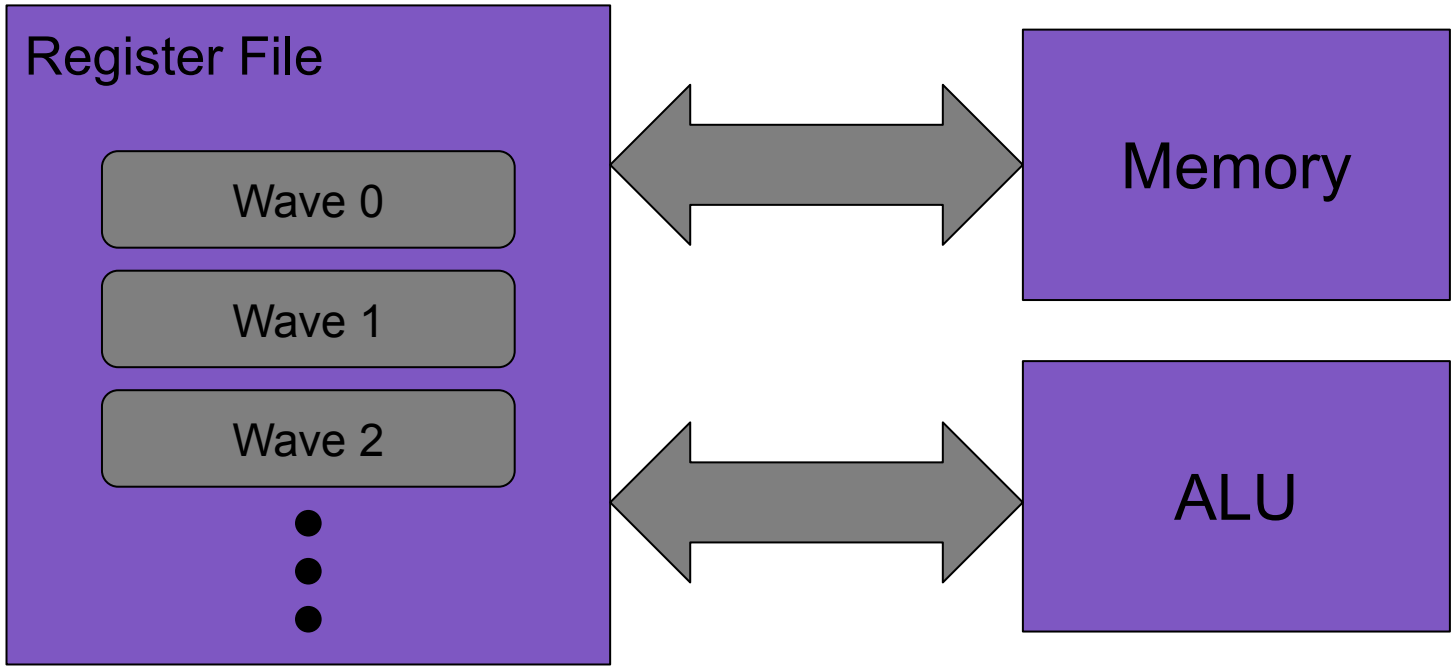
- "Optimal Register Allocation for SSA-form Programs in polynomial Time" by Sebastian Hack and Gerhard Goos
  - Not *actually* optimal!
- First run register allocation, then deconstruct SSA
- Phi nodes get registers assigned!

Register Allocation and SSA

- Coalescing is implicit
- Spilling can be decoupled

What about GPUs?

Dynamic register sharing



GPUs

- Might benefit from using less registers
- Spilling is *expensive* on GPUs
- -> SSA-based allocators are much better

The Algorithm

First Steps

- Our initial architecture:
  - No branching (single basic block)
  - N registers, all exactly the same

Liveness and Kill Flags

```
v0 = load ...
v1 = load ...
v3 = add v0, v1(kill)
v4 = add v0(kill), v2(kill)
```

Baby's First Register Allocator

```
available = {r1, r2, ..., rN}
for each instruction:
    for each use of V:
        if use.kill:
            available += V.reg
    for each definition V:
        V.reg = pick_physreg(available)
        available -= V.reg
```

Handling Control Flow

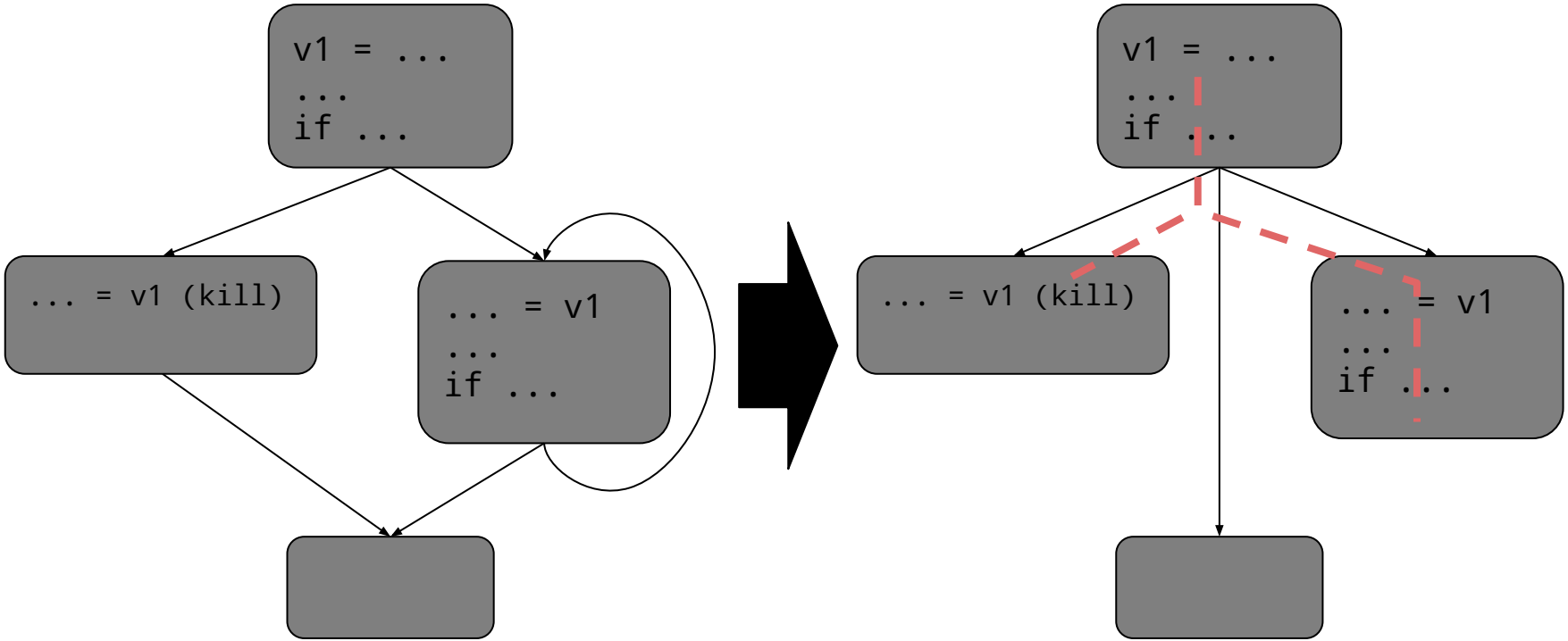
Handling Control Flow

- Use classic dataflow algorithm to find liveness
- Blocks have *live-in* and *live-out* sets
- Still have kill flags as before

Interlude: Dominance and Liveness

- A *dominates* B if every path from the start to B goes through A
- SSA definitions always dominate their uses

Interlude: Dominance and Liveness



Handling Control Flow

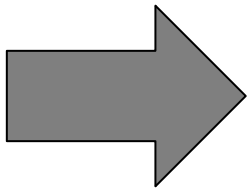
```
for each block, ordered by dominance:
    available = {r1, ..., rN}
    foreach live-in value V:
        available -= V.reg
    // main part same as before
    for each instruction in block:
        ...
```

Phi Nodes?

- We may assign phi sources and destination to different registers
- Phi nodes happen *in parallel*

Phi Nodes Example

```
a = ...
d = ...
if (...) {
    e = a
    c = d
} else {
    e = ...
    c = a
}
... = e
... = c
```



```
a = ...
d = ...
if (...) {
    if:
} else {
    else:
    e_0 = ...
}
e_1 = φ(if: a, else: e_0)
c = φ(if: d, else: a)
```

Phi Nodes Example, Continued

```
a:r0 = ...
d:r1 = ...
if (...) {
    if:
} else {
    else:
    e_0:r1 = ...
}
e_1:r0 = φ(if: a:r0, else: e_0:r1)
c:r1 = φ(if: d:r1, else: a:r0)
```

Swap Instructions

- Many targets already have a suitable swap instruction
- Xor trick:

```
x = x ^ y
y = y ^ x
x = x ^ y
```

Resolving Phi Nodes

- May have to split critical edges
- Create a *transfer graph*, resolve piece-by-piece
- Similar to SSA deconstruction
  - See "Revisiting Out-of-SSA Translation for Correctness, Code Quality, and Efficiency" by Boissinot et. al.
- Need to consider *affinities* in `pick_physreg()`

Live Range Splitting

Vector Registers

- Load/store series of registers
- For example:

```
// equivalent to:
// r0 = load ...
// r1 = load ... + 1
r[0:1] = load.v2 ...
...
store.v2 r[1:2] // store r1 and then r2
```

Vector Registers and SSA

- Need to add split/collect instructions
- Similar in spirit to phi nodes

```
v2 = collect v0, v1
store.v2 v2, ...
v3 = load.v2 ...
v4, v5 = split v3
... = v4
... = v5
```

The Problem

```
v0 = load.v3 ...
v1, v2, v3 = split v0
... = v1 (kill)
... = v3 (kill)
v4 = load.v2 ...
```

The Problem

```
v0:r[0-2] = load.v3 ...
v1:r0, v2:r1, v3:r2 = split v0:r[0-2]
... = v1:r0 (kill)
... = v3:r2 (kill)
v4:??? = load.v2 ...
```

r0	r1	r2
v0	v0	v0
v1	v2	v3
	v2	v3
	v2	

The Solution: Live Range Splitting

```
v0:r[0-2] = load.v3 ...
v1:r0, v2:r1, v3:r2 = split v0
... = v1:r0 (kill)
... = v3:r2 (kill)
r2 = r1
v4:r[0-1] = load.v2 ...
```

Live-Range Splitting: Worst-case Scenario

```
foo = load.v2 ...
```

r0	r1	r2	r3	r4	...	rN-2	rN-1	rN
	v0	v0	v1	v1	...	vM	vM	

Live Range Splitting and Control Flow

```
v1:r1 = ...
if (...) {
    ...
    r2 = r1 // live-range split
} else {
    ...
}
... = v1 (kill)
```

Live Range Splitting and Control Flow

- Need to repair SSA:
  - Either create new Phis
  - Or copy the value back

Conclusion

- Simple idea, complex implementation
- Code Quality depends entirely on Coalescing
- -> Workshop tomorrow at 16:35

Questions?

Live Range Splitting

- Three types of values:
  - killed uses
  - live-through
  - definitions
- Insert *parallel copies*
- First solution: sort/compact values
  - live-through, then killed uses