A List of Scheduling Primitives

A.1 Loop Transformations

Scheduling operation	Code transformation	Safety conditions

`reorder_loops(p, loops)`	`for i:`
	`  for j:`
	`    s`
	⇝
	`for j:`
	`  for i:`
	`    s`
		The `j` loop's bounds cannot depend on `i`. The loop body `s` commutes for different (i, j) and (i', j') pairs.

`divide_loop(`
`  p,`
`  loop,`
`  c,`
`  [io, ii],`
`  tail_strategy`
`)`
	`# tail_strategy=perfect`
	`for i < I:`
	`  s`
	⇝
	`for io < I/c:`
	`  for ii < c:`
	`    s[i ↦ c*io+ii]`
	
	`# tail_strategy=guard`
	`for i < I:`
	`  s`
	⇝
	`for io < I/c:`
	`  for ii < c:`
	`    if c*io+ii < I:`
	`      s[i ↦ c*io+ii]`
	
	`# tail_strategy=cut`
	`for i < I:`
	`  s`
	⇝
	`for io < I/c:`
	`  for ii < c:`
	`    s[i ↦ c*io+ii]`
	`for ii < I%c:`
	`  s[i ↦ c*I/c+ii]`
	
	`# tail_strategy=cut_and_guard`
	`for i < I:`
	`  s`
	⇝
	`for io < I/c:`
	`  for ii<c:`
	`    s[i ↦ c*io+ii]`
	`if I%c > 0:`
	`  for ii < I%c:`
	`    s[i ↦ c*I/c+ii]`
		For perfect `tail_strategy`, the loop bound is perfectly divisible by `c`.

`divide_with_recompute(`
`  p, loop, N, c, [io, ii]`
`)`
	`for i < I:`
	`  s`
	⇝
	`for io < N:`
	`  for ii < c+I-N*c:`
	`    s[i ↦ c*io+ii]`
		`s` is idempotent and `N*c ≤ I`.

`mult_loops(p, loops, k)`	`for i < I:`
	`  for j < c:`
	`    s`
	⇝
	`for k < I*c:`
	`  s[i ↦ k/c, j ↦ k%c]`
		The `j` loop is the only statement in the `i` loop's body. Also, `c` is constant.

`cut_loop(p,loop,e)`	`for i in l, h:`
	`  s`
	⇝
	`for i in l, e:`
	`  s`
	`for i in e, h:`
	`  s`
		The cutoff `e` lies between the loop bounds `l` and `h`

`join_loops(p, loop, loop2)`	`for i in l1, h1:`
	`  s`
	`for i2 in l2, h2:`
	`  s`
	⇝
	`for i in l1, h2:`
	`  s`
		The loops are adjacent, have identical bodies, and `h1 = l2`.

`shift_loop(p, loop, e)`	`for i in l, h:`
	`  s`
	⇝
	`for i in e, h+e-l:`
	`  s`
		The new lower bound `e` ≥ 0.

`fission(p, gap)`	`for i:`
	`  s1`
	`  ----`
	`  s2`
	⇝
	`for i:`
	`  s1`
	`----`
	`for i:`
	`  s2`
		`s2` cannot depend on allocations in `s1`, and effects of `s1` and `s2` commute

`remove_loop(p, loop)`	`for i:`
	`  s`
	⇝ `s`
		`s` is idempotent and cannot depend on `i`. Loop executes at least once.

`add_loop(p, s, i, hi, guard)`	`# guard = False`
	`s` ⇝
	`for i < hi:`
	`  s`
	
	`# guard = True`
	`s` ⇝
	`for i < hi:`
	`  if i == 0:`
	`    s`
		`s` is idempotent and `hi` is positive.

`unroll_loop(p, loop)`	`for i in lo, hi:`
	`  s`
	⇝
	`s[i ↦ lo]`
	`...`
	`s[i ↦ hi - 1]`
		`hi` and `lo` are constants, `hi - lo` > 0

A.2 Code Rearrangement

Scheduling operation	Code transformation	Safety conditions
`reorder_stmts(p, s1, s2)`	s1 s2 ⇝ s2 s1	s1 and s2 commute.
`commute_expr(p, e)`	x+y ⇝ y+x x*y ⇝ y*x	N/A

A.3 Scope Transformations

Scheduling operation	Code transformation	Safety conditions

`specialize(p, s, conds)`	s ⇝ if conds[0]:
	  s
	else:
	  if conds[1]:
	    s
	  else:
	    ...
		conds only contains valid boolean expressions.

`fuse(p, scope, scope2)`	for i<I:
	  s
	for i<I:
	  s2
	⇝ for i<I:
	  s
	  s2
		Equivalent upper loop bound I in its context. Iterations of s commute with any iteration of s2.

	if e:
	  s
	if e:
	  s2
	⇝ if e:
	  s
	  s2
		Equivalent if conditional expression e in its context.

`lift_scope(p, scope)`		For all cases, scope is the only statement in its parent's body.

	if e:
	  if e2: # scope
	    s
	  else:
	    s2
	else:
	  s3
	⇝ if e2:
	  if e:
	    s
	  else:
	    s3
	else:
	  if e:
	    s2
	  else:
	    s3
		N/A

	if e: # scope
	  for i:
	⇝ for i:
	  if e:
		if statement cannot have an else clause.

	for i:
	  if e: # scope
	    s
	  else:
	    s2
	⇝ if e:
	  for i:
	    s
	else:
	  for i:
	    s2
		e cannot depend on i.

A.4 Multiple procedures

Scheduling operation	Code transformation	Safety conditions
`inline(p, foo)`	Inline a callsite of foo	N/A
`replace(p,s,instr)`	Replace s with a call to an instruction instr	s and the instr function are unifiable.
`call_eqv(p,foo,bar)`	Replace foo with a call to an equivalent procedure bar	The two procedures foo and bar are equivalent, e.g. scheduled from the same procedure
`extract_subproc(p,s,foo)`	Isolates s into a function named foo, and replaces s with a call to foo.	N/A

A.5 Buffer Transformations

Scheduling operation	Code transformation	Safety conditions

`lift_alloc(p, a)`	`for i:`
	`  a: T[sz]`
	`  s`
	⤳
	`a: T[sz]`
	`for i:`
	`  s`
		Dimensions `sz` don't depend on `i`. No loop-carry dependencies.

`sink_alloc(p, a)`	`a: T`
	`for i:`
	`  s`
	⤳
	`for i:`
	`  a: T`
	`  s`
		`a` is only accessed in the `i` loop. No loop-carried dependencies.

`delete_buffer(p, a)`	`a: T`
	`s`
	⤳ `s`
		`a` should be dead.

`reuse_buffer(p, a, b)`	`a : T[sz]`
	`...`
	`b : T[sz]`
	`s`
	⤳
	`a : T[sz]`
	`...`
	`s[b ↦ a]`
		`a` and `b` have the same type and size. `a` is dead after `b`'s allocation.

`resize_dim(`
`  p, a, dim, sz, off, fold`
`)`
	`## fold = False`
	`a: T[_]`
	`s`
	⤳
	`a: T[_]`
	`s[a[e] ↦ a[e-off]]`
		`l < h`, `a` only accessed in `dim`-th dimension between `l` and `h`.

	`## fold = True`
	`a: T[sz]`
	`s`
	⤳
	`a: T[sz]`
	`s[a[e] ↦ a[(e-off)%sz]]`
		`l < h`, `a` only accessed in `dim`-th dimension between `l` and `h`. Also, accesses to `a` never access more than `sz` earlier than the largest access so far.

`expand_dim(p, a, sz, e)`	`a: T[_]`
	`s`
	⤳
	`a: T[_]`
	`s[a[_] ↦ a[e, _]]`
		`sz` is positive, `e` only uses existing variables, and `e < sz` in all contexts.

`rearrange_dim(p, a, p_vec)`	`# p_vec = [2, 0, 1]` `a: T[N, M, K]` ⤳ `a: T[K, N, M]`	`a` cannot be windowed or passed in function calls.

`divide_dim(p, a, dim, c)`	`# dim = 0, c = 4`
	`a: R[12, _]`
	`s`
	⤳
	`a: R[3, 4, _]`
	`s[a[i, _] ↦ a[i/4, i%4, _]]`
		`a`'s `dim`-th dimension size is constant and divisible by `c`

`mult_dim(p, a, dim, dim2)`	`# dim = 0, dim2 = 2`
	`a: R[n, _, 4]`
	`s`
	⤳
	`a: R[4*n,_]`
	`s[a[i, _, j] ↦ a[4*i+j, _]]`
		`a` cannot be windowed or passed in function call. One of the dimensions is constant size.

`unroll_buffer(p, a, dim)`	`a: T[c]` ⤳
	`a1: T`
	`...`
	`ac: T`
		For this dimension, `a` has constant size and index accesses. `a` cannot be windowed along this dimension.

`bind_expr(p, e, a, cse)`	`s` ⤳
	`a: T`
	`a = e`
	`s[e ↦ a]`
		N/A

`stage_mem(p, s, a, w, tmp)`	`# w = a[0:n, j-1:j]`
	`for i < n-1: # s`
	`  a[i,j] = a[i+1,j-1]`
	⤳
	`tmp: T[n, 2]`
	`for k0 < n:`
	`  for k1 < 2:`
	`    tmp[k0, k1] = a[k0,j-1+k1]`
	`for i < n-1:`
	`  tmp[i, 1] = tmp[i+1, 0]`
	`for k0 < n:`
	`  for k1 < 2:`
	`    a[k0, j-1+k1] = tmp[k0, k1]`
		The code block `s` does not access buffer `a` outside of the given `window`.

A.6 Simplification

Scheduling operation	Code transformation	Safety conditions
`simplify(p)`	does arithmetic simplifications and trivial branch elimination on the entire procedure	N/A
`eliminate_dead_code(p, scope)`	`for i in l, h:` $\rightsquigarrow$ `pass`	The loop runs 0 times, e.g. `l`≥`h`.

	`if e:`
	`  s`
	`else`
	`  s2` $\rightsquigarrow$ `s` or `s2`
		`e` is equivalent to `True` or `False` in its context.

`rewrite_expr(p, e, e')`	`e` $\rightsquigarrow$ `e'`	`e` and `e'` are equivalent in the context in which `e` appears.
`merge_writes(p, s1, s2)`	`x = e1` `x = e2` $\rightsquigarrow$ `x = e2`	The two statements write to the same destination `x`. `e2` cannot read from `x`.
	`x += e1` `x = e2` $\rightsquigarrow$ `x = e2`	
	`x = e1` `x += e2` $\rightsquigarrow$ `x = e1 + e2`	
	`x += e1` `x += e2` $\rightsquigarrow$ `x += e1 + e2`	
`inline_window(p, w)`	`w = a[_]` `s` $\rightsquigarrow$ `s[w ↦ a[_]]`	N/A
`inline_assign(p, x = e)`	`x = e` `s` $\rightsquigarrow$ `s[x ↦ e]`	`x` cannot be written to in the block `s` after `s`.

A.7 Backend-Checked Annotations

All the scheduling primitives that we have discussed so far are safety-checked within the their rewrite process. By contrast, consistency of precision types, memory annotations, and window annotations are performed as back-end checks after all scheduling is complete; immediately prior to code generation.

Scheduling operation	Code transformation	Compilation safety check
`set_memory(p, a, MEM')`	`a @ MEM` $\rightsquigarrow$ `a @ MEM'`	Ensures that the accesses to the buffer obeys the custom memory read/write/window definition, and the caller and callee have the same memory types
`set_precision(p, a, T')`	`a: T` $\rightsquigarrow$ `a: T'`	Checks the caller, callee, and the both sides of binary operation have the same precision types
`parallelize_loop(p, loop)`	Annotate loop as parallel	Ensures that the loop iterations do not have RAW or WAW dependencies.
`set_window(p, a)`	`a: T[_]` $\rightsquigarrow$ `a: [T][_]`	Checks the caller and callee have the same window shapes

A.8 Configuration State

Scheduling operation	Code transformation	Safety conditions
`bind_config(p, e, cfg, field)`	`s` $\rightsquigarrow$ `cfg.field = e` `s[e ↦ cfg.field]`	`cfg.field` is not read by code which executes afterwards.

`delete_config(p, cfg.field = _)`	`s`
	`cfg.field = _`
	`s2` $\rightsquigarrow$ `s`
	`s2`
		`cfg.field` is not read by code which executes afterwards.

`write_config(p, gap, cfg, field, e)`	`s`
	`s2` $\rightsquigarrow$ `s`
	`cfg.field = e`
	`s2`
		`cfg.field` is not read by code which executes afterwards.