KURUKULASOORIYA KVR **200332X**

CS 4553 : SCIENTIFIC COMPUTING
DEPENDENCY ANALYSIS AND SCHEDULING REPORT

Exo is a low-level user-schedulable language to help performance engineers optimize high-performance computing kernels on new hardware accelerators in an efficient way.

Step 01
1.1 Scheduling Primitives

A1: Loop Transformations : To modify loop structures

- reorder_loops(p, loops): Changes the nesting order of loops
- divide_loop(p, loop, c, [i0, i1], tail_strategy): Splits a loop into chunks
- divide_with_recompute(...): Similar to divide_loop(p, loop, c, [i0, i1], tail_strategy), but recalculates some values to make it useful when recomputation is cheaper than memory access.
- mult_loops(p, loops, k): Loops are merged by multiplying their iteration spaces
- cut_loop(p, loop, e): Splits a loop into two at a specific iteration
- join_loops(p, loop1, loop2): Merges adjacent loops with identical bodies
- shift_loop(p, loop, e): Changes the loop's start point
- fission(p, gap): Splits a loop body into multiple loops
- remove_loop(p, loop): Eliminates a loop if it iterates only once or is unnecessary
- add_loop(p, s, i, hi, guard): Adds a loop around a statement block
- unroll_loop(p, loop, hi): Replaces a loop with repeated statements

A2: Code Rearrangement : To change the order of statements or expressions

- reorder_stmts(p, s1, s2): Swaps the order of two statements when they do not interfere.
- commute_expr(p, e): Changes the order of operands in commutative expressions like x + y to y + x

A3: Scope Transformations : To restructure the program’s control flow

- specialize(p, s, conds): Splits a block into conditionally executed sub blocks based on boolean conditions
- fuse(p, scope, scope2): Merges multiple loops or conditional blocks into one when their iterations or conditions align
- lift_scope(p, scope): Moves inner scopes outward when possible

A4. Multiple Procedures : To restructure code across function boundaries

- inline(p, foo): Replaces a function call with the function body
- replace(p, s, instr): Swaps a statement with a new one
- call_eqv(p, foo, bar): Replaces a function with an equivalent
- extract_subproc(p, s, foo): Turns a code block into a separate function

A5. Buffer Transformations : To modify buffer allocations/ dimensions and structure

Allocation and Reuse

- lift_alloc(p, a): Moves buffer allocation outside a loop to avoid redundant allocations when dimensions are loop invariant.
- sink_alloc(p, a): Moves buffer allocation inside a loop to minimize its lifetime if the buffer is used only inside the loop.
- delete_buffer(p, a): Removes an unused buffer
- reuse_buffer(p, a, b): Reuses buffer a in place of b when both are compatible t

Dimension Transformations

- resize_dim(...): Adjusts the size of a buffer dimension by shifting or folding
- expand_dim(p, a, sz, e): Adds a dimension to a buffer
- rearrange_dim(p, a, pvec): Changes the order of buffer dimensions like from NHWC to NCHW

Reshape and Indexing

- divide_dim(p, a, dim, c): Splits a dimension into two by factoring it into chunks of size c,
- mult_dim(p, a, dim1, dim2): Merges two dimensions into one for flattened access or reshaping tensors.
- unroll_buffer(p, a, dim): Converts a dimension into separate buffer elements when size is constant
- bind_expr(p, e, a, cse): Binds a sub expression to a buffer to eliminate redundant computations.

Staging

- stage_mem(p, s, a, w, tmp): Creates a temporary buffer to hold a local window (tile) of a buffer a

A6 : Simplification : To simplify the code to avoid complex and unnecessary executions

- simplify(p): Performs arithmetic simplifications and basic branch elimination
- eliminate_dead_code(p, scope): Removes branches of code that are never executed based on known boolean conditions.
- rewrite_expr(p, e, e'): Replaces expression e with e' throughout the code.
- merge_writes(p, s1, s2): Combines consecutive writes to the same variable x into one, where possible.
- inline_window(p, w): Inlines the definition of window w directly into the code.
- inline_assign(p, x = e): Replaces assignments to variable x with its value e across the code.

A7 : Backend-Checked Annotations: To enable validations on backend

- set_memory(p, a, MEM'): Sets a new memory access annotation for buffer a.
- set_precision(p, a, T'): Changes the precision type of value a to T'.
- parallelize_loop(p, loop): Annotates a loop to be executed in parallel.
- set_window(p, a): Updates the window shape of buffer a.

A8 : Configuration State : To modify config field

- bind_config(p, e, cfg, field): Binds a value e to a configuration field during execution.
- delete_config(p, cfg.field = _): Removes a configuration field assignment.
- write_config(p, gap, cfg, field, e): Updates the configuration field cfg.field with a new value e.

1.2: Implementation of Scheduling Primitives using different architectures

A1: Loop Transformations

Pthreads	● Transformations like divide_loop and reorder_loops can split loops into segments and reorder loop nesting to improve cache locality, enabling greater load balancing and efficient scheduling of threads. ● Unrolling reduces loop overhead by minimizing loop control instructions, which is helpful in enhancing multithreading performance. ● Shift loops balance the work among threads by shifting the starting position of the loop.
CUDA	● Through transformations like divide_loop and join_loops, work can be mapped onto different threads or thread blocks in an effective way, leading to optimized GPU kernel execution. ● Reordering loops can align memory accesses to improve memory coalescing and reduce memory transaction overhead. ● CUDA highly benefits from loop unrolling, which reduces control flow instructions and fills the GPU instruction pipeline better
Vector Extensions	● Unrolling allows several data points to be combined in one iteration so that SIMD units can process them in parallel. ● Loop reordering optimizes data locality to access data sequentially to keep the vector registers busy and reduce memory access latencies.

A2: Code Rearrangement

Pthreads	● By rearranging code to place nearby data together, memory access can be coalesced, reducing cache misses and multithreaded program efficiency. ● By carefully rearranging statements, threads can carry out independent work without synchronization or communication, reducing bottlenecks.
CUDA	● Rearranging code so that threads access memory in a coalesced manner optimizes GPU memory bandwidth.

	● Statement reordering in kernel code can minimize dependencies, obtaining greater utilization of the warp-level parallelism of the GPU.
Vector Extensions	● Code reordering so that multiple data points are processed in a contiguous fashion allows SIMD instructions to process data more effectively, reducing overhead from branch misprediction and memory misalignment.

A3: Scope Transformations

Pthreads	● Scope modification can break down a large issue into tiny sub-tasks that can be run in parallel by several threads, streamlining workload distribution and parallelism. ● Minimizing the scope of critical sections can minimize the amount of code that must be locked, thus decreasing contention between threads.
CUDA	● By restructuring the scope of kernel launches, workloads can be divided into smaller kernels that better utilize the GPU's architecture and memory resources. ● Scope transformations can be utilized to reduce branch divergence within warps, improving the overall execution efficiency on CUDA devices.
Vector Extensions	● Control flow transformations can prevent branch divergence bottlenecks by rearranging loops and conditions to more closely fit SIMD execution, allowing the processor to execute vectorized operations without interruption.

A4: Multiple Procedures

Pthreads	● Breaking up large functions into small, independent procedures allows for more parallelism between threads. Small functions also reduce synchronization overhead. ● Dividing tasks into separate procedures allows the CPU's cache to hold more useful information, with less cache misses during runtime.
CUDA	● By placing computations in separate procedures or kernels, workloads can be more evenly distributed across thread blocks. Each kernel can focus on various aspects of computation, with minimal thread contention.

	● Procedures can be coded to take advantage of shared memory and registers in CUDA, reducing memory latency.
Vector Extensions	● Splitting tasks into separate functions reduces instruction overlap and ensures SIMD instructions are only used where needed, getting the best out of vector registers.

A5: Buffer Transformations

Pthreads	● Buffer transformations can be utilized to allocate buffers in contiguous memory locations to improve cache efficiency when threads access the buffers. ● By altering buffer sizes and memory layouts, work can be distributed between threads, improving performance and reducing memory contention.
CUDA	● Buffer transformations can rearrange data storage in shared memory, reducing global memory accesses and improving memory throughput. ● Buffers can be constructed to ensure that threads access memory in a coalesced pattern, reducing global memory access latency.
Vector Extensions	● Buffers can be aligned to ensure that memory accesses are aligned with the SIMD instruction set requirements. This reduces memory access penalties and improves performance.

A6: Simplification

Pthreads	● By eliminating redundant operations and loops, you reduce overhead, allowing threads to focus on valuable computation. ● Simplified code reduces usage of complex synchronization, improving parallelism and reducing thread contention.
CUDA	● Lower CUDA kernel complexity can result in fewer instruction pipelines, reducing execution time and improving GPU performance. Removing redundant calculations or memory operations can execute the code faster.
Vector Extensions	● Lowering control flow complexity and eliminating unnecessary branching can allow more efficient use of SIMD execution, where all lanes within a vector perform the same operation in parallel.

A7: Backend-Checked Annotations

Pthreads	• Backend-checked annotations can ensure that thread synchronization and memory access patterns are correct, preventing race conditions and memory corruption, which can result in better stability and performance in multithreaded environments
CUDA	• By inserting backend annotations for CUDA kernel launches and memory access, we can identify inefficiencies like underutilization of threads or non-optimal memory access patterns, for better kernel performance.
Vector Extensions	• Annotations can check if the program is exploiting the complete capability of SIMD lanes and vector registers, and help find bottlenecks and potential inefficiencies in vectorized code.

A8: Configuration State

Pthreads	• Changing the configuration can help in adjusting the number and scheduling of threads, to maximize CPU core utilization and minimize contention.
CUDA	• The configuration state change of kernel launches can optimize the thread block sizes and grid sizes, so as to make the best use of the GPU resources.
Vector Extensions	• Tweaking the configuration for how vector operations are performed like choosing appropriate vector lengths can improve SIMD performance via better utilization of vector registers.

Step 02
Please refer to https://github.com/Zury7/exo-QuizTry for the changed code.

Step 03
Quiz 1

Incorrect Output:

```
PS C:\Users\ASUS\Downloads\exo-QuizTry\examples\quiz1> exocc quiz1.py
def vec_double_optimized(N: size, inp: f32[N] @ DRAM, out: f32[N] @ DRAM):
    assert N % 8 == 0
    two_vec: R[8] @ AVX2
    for ii in seq(0, 8):
        two_vec[ii] = 2.0
    for io in seq(0, N / 8):
        out_vec: f32[8] @ AVX2
        inp_vec: f32[8] @ AVX2
        for i0 in seq(0, 8):
            inp_vec[i0] = inp[i0 + 8 * io]
        for ii in seq(0, 8):
            out_vec[ii] = two_vec[ii] * inp_vec[ii]
        for i0 in seq(0, 8):
            out[i0 + 8 * io] = out_vec[i0]
```

Based on the incorrect output shown above, the buggy code has the form of vectorized SIMD operations but does not use real vector intrinsics. Thus it emulates vector operations by looping over array elements manually, but the operations remain scalar not parallelized.

Solution:

For enabling parallel processing, memory buffers such as inp_vec, out_vec, and two_vec must be decorated as AVX2.

```
76      # Replace with AVX instructinos
77      avx_instrs = [vector_assign_two, vector_multiply, \
78      # Set the memory types to be AVX2 vectors
79      for name in ["two", "out", "inp"]:
80          p = set_memory(p, f"{name}_vec", AVX2)
81      p = replace_all(p, avx_instrs)
82
```

It accomplishes memory type annotations to enable replace_all(p, avx_instrs) to substitute the scalar code using vector intrinsics.

Corrected Output:

```
PS C:\Users\ASUS\Downloads\exo-QuizTry\examples\quiz1> exocc quiz1.py
def vec_double_optimized(N: size, inp: f32[N] @ DRAM, out: f32[N] @ DRAM):
    assert N % 8 == 0
    two_vec: R[8] @ AVX2
    vector_assign_two(two_vec[0:8])
    for io in seq(0, N / 8):
        out_vec: f32[8] @ AVX2
        inp_vec: f32[8] @ AVX2
        vector_load(inp_vec[0:8], inp[8 * io + 0:8 * io + 8])
        vector_multiply(out_vec[0:8], two_vec[0:8], inp_vec[0:8])
        vector_store(out[8 * io + 0:8 * io + 8], out_vec[0:8])
```

Quiz 2

Incorrect output:

```
PS C:\Users\ASUS\Downloads\exo-QuizTry\examples\quiz2> exocc quiz2.py
Traceback (most recent call last):
  File "<frozen runpy>", line 198, in _run_module_as_main
  File "<frozen runpy>", line 88, in _run_code
  File "C:\Users\ASUS\AppData\Local\Programs\Python\Python313\Scripts\exocc.exe\__main__.py", line 7, in <module>
    sys.exit(main())
             ~~~~^^
  File "C:\Users\ASUS\AppData\Local\Programs\Python\Python313\Lib\site-packages\exo\main.py", line 58, in main
    for proc in get_procs_from_module(load_user_code(mod)):
                                      ~~~~~~~~~~~~~~^^^^^
  File "C:\Users\ASUS\AppData\Local\Programs\Python\Python313\Lib\site-packages\exo\main.py", line 107, in load_user_code
    loader.exec_module(user_module)
    ~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^
  File "<frozen importlib._bootstrap_external>", line 1026, in exec_module
  File "<frozen importlib._bootstrap>", line 488, in _call_with_frames_removed
  File "C:\Users\ASUS\Downloads\exo-QuizTry\examples\quiz2\quiz2.py", line 52, in <module>
    w = wrong_schedule(scaled_add)
  File "C:\Users\ASUS\Downloads\exo-QuizTry\examples\quiz2\quiz2.py", line 48, in wrong_schedule
    p = fission(p, vector_assign.after())
  File "C:\Users\ASUS\AppData\Local\Programs\Python\Python313\Lib\site-packages\exo\API_scheduling.py", line 100, in __call__
    return self.func(*bound_args.args, **bound_args.kwargs)
           ~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Problem

The code does unsafe loop fission, which creates a compiler error as above. The fission is initiated after vector allocation (vec_1) within an inner loop, causing invalid handling of memory in the process of exo scheduling.

Solution

Lift vector allocations out of the loop before fission

```
41      for i in range(num_vectors):
42          vector_reg = p.find(f"vec: _ #{i}")
43          p = expand_dim(p, vector_reg, 8, "ii")
44          p = lift_alloc(p, vector_reg)
45
46      for i in range(num_vectors):
47          vector_assign = p.find(f"vec = _ #{i}")
48          p = fission(p, vector_assign.after())
```

Corrected Output:

```
PS C:\Users\ASUS\Downloads\exo-QuizTry\examples\quiz2> exocc quiz2.py
def scaled_add_scheduled(N: size, a: f32[N] @ DRAM, b: f32[N] @ DRAM,
                         c: f32[N] @ DRAM):
    assert N % 8 == 0
    for io in seq(0, N / 8):
        vec: R[8] @ DRAM
        vec_1: R[8] @ DRAM
        vec_2: f32[8] @ DRAM
        vec_3: R[8] @ DRAM
        vec_4: R[8] @ DRAM
        vec_5: f32[8] @ DRAM
        for ii in seq(0, 8):
            vec_1[ii] = 2
        for ii in seq(0, 8):
            vec_2[ii] = a[8 * io + ii]
        for ii in seq(0, 8):
            vec[ii] = vec_1[ii] * vec_2[ii]
        for ii in seq(0, 8):
            vec_4[ii] = 3
        for ii in seq(0, 8):
            vec_5[ii] = b[8 * io + ii]
        for ii in seq(0, 8):
            vec_3[ii] = vec_4[ii] * vec_5[ii]
        for ii in seq(0, 8):
            c[8 * io + ii] = vec[ii] + vec_3[ii]
```

Quiz 3:

Problem:

The bug is because the get_loops_at_or_above function includes only the loops over xo_loop (the yo_loop) but not the xo_loop itself incorrectly. Thus, the memory optimization is done incorrectly, and the width dimension (W) isn't minimized as needed as shown below.

Incorrect output:

Please find on the other page

```
PS C:\Users\ASUS\Downloads\exo-QuizTry\examples\quiz3> exocc quiz3.py
def tile_and_fused_blur_scheduled(W: size, H: size, blur_y: ui16[H, W] @ DRAM,
                                  inp: ui16[H + 2, W + 2] @ DRAM):
    assert H % 32 == 0
    assert W % 256 == 0
    blur_x: ui16[34, W] @ DRAM
    for yo in seq(0, H / 32):
        for xo in seq(0, W / 256):
            for yi in seq(0, 34):
                for xi in seq(0, 256):
                    blur_x[yi + 32 * yo - 32 * yo, xi + 256 *
                           xo] = (inp[yi + 32 * yo, xi + 256 * xo] +
                                  inp[yi + 32 * yo, 1 + xi + 256 * xo] +
                                  inp[yi + 32 * yo, 2 + xi + 256 * xo]) / 3.0
            for yi in seq(0, 32):
                for xi in seq(0, 256):
    assert H % 32 == 0
    assert W % 256 == 0
    blur_x: ui16[34, W] @ DRAM
    for yo in seq(0, H / 32):
        for xo in seq(0, W / 256):
            for yi in seq(0, 34):
                for xi in seq(0, 256):
                    blur_x[yi + 32 * yo - 32 * yo, xi + 256 *
                           xo] = (inp[yi + 32 * yo, xi + 256 * xo] +
                                  inp[yi + 32 * yo, 1 + xi + 256 * xo] +
    assert H % 32 == 0
    assert W % 256 == 0
    blur_x: ui16[34, W] @ DRAM
    for yo in seq(0, H / 32):
        for xo in seq(0, W / 256):
            for yi in seq(0, 34):
                for xi in seq(0, 256):
    assert H % 32 == 0
    assert W % 256 == 0
    blur_x: ui16[34, W] @ DRAM
    for yo in seq(0, H / 32):
    assert H % 32 == 0
    assert W % 256 == 0
    assert H % 32 == 0
    assert H % 32 == 0
    assert W % 256 == 0
    blur_x: ui16[34, W] @ DRAM
for yo in seq(0, H / 32):
    for xo in seq(0, W / 256):
        for yi in seq(0, 34):
            for xi in seq(0, 256):
                blur_x[yi + 32 * yo - 32 * yo, xi + 256 *
                       xo] = (inp[yi + 32 * yo, xi + 256 * xo] +
                              inp[yi + 32 * yo, 1 + xi + 256 * xo] +
                              inp[yi + 32 * yo, 2 + xi + 256 * xo]) / 3.0
        for yi in seq(0, 32):
            for xi in seq(0, 256):
                blur_y[yi + 32 * yo, xi + 256 * xo] = (
                    blur_x[yi + 32 * yo - 32 * yo, xi + 256 * xo] +
                    blur_x[1 + yi + 32 * yo - 32 * yo, xi + 256 * xo] +
                    blur_x[2 + yi + 32 * yo - 32 * yo,
                           xi + 256 * xo]) / 3.0
```

Solution:

Modify the get_loops_at_or_above function to include the xo_loop by changing loops = [] into loops = [cursor] .

```
32  def get_loops_at_or_above(cursor):
33      loops = [cursor]
34      while not isinstance((parent := cursor.parent()), InvalidCursor):
35          loops.append(parent)
36          cursor = parent
37      return list(reversed(loops))
```

Corrected Output:

```
PS C:\Users\ASUS\Downloads\exo-QuizTry\examples\quiz3> exocc quiz3.py
def tile_and_fused_blur_scheduled(W: size, H: size, blur_y: ui16[H, W] @ DRAM,
                                  inp: ui16[H + 2, W + 2] @ DRAM):
    assert H % 32 == 0
    assert W % 256 == 0
    for yo in seq(0, H / 32):
        blur_x: ui16[34, 256] @ DRAM
        for xo in seq(0, W / 256):
            for yi in seq(0, 34):
                for xi in seq(0, 256):
                    blur_x[yi + 32 * yo - 32 * yo, xi + 256 * xo - 256 *
                           xo] = (inp[yi + 32 * yo, xi + 256 * xo] +
                                  inp[yi + 32 * yo, 1 + xi + 256 * xo] +
                                  inp[yi + 32 * yo, 2 + xi + 256 * xo]) / 3.0
            for yi in seq(0, 32):
                for xi in seq(0, 256):
                    blur_y[yi + 32 * yo, xi +
                           256 * xo] = (blur_x[yi + 32 * yo - 32 * yo,
                                               xi + 256 * xo - 256 * xo] +
                                        blur_x[1 + yi + 32 * yo - 32 * yo,
                                               xi + 256 * xo - 256 * xo] +
                                        blur_x[2 + yi + 32 * yo - 32 * yo,
                                               xi + 256 * xo - 256 * xo]) / 3.0
```