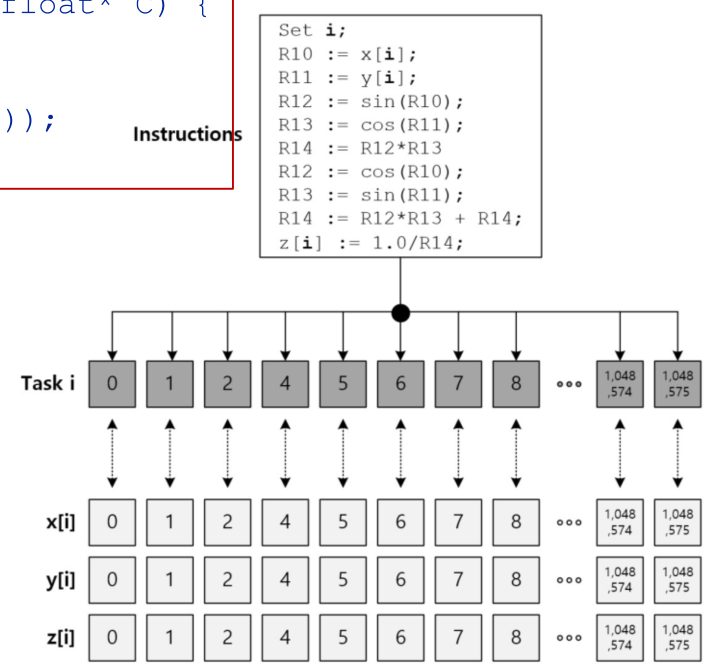
Kernel Programming Model

- Defines how the concurrency model is mapped to physical hardware.

```
void combine_two_arrays_CPU(float *A, float *B, float *C, int n) {
    for (int i = 0; i < n; i++) {
        C[i] = 1.0f / (sin(A[i])*cos(B[i]) + cos(A[i])*sin(B[i]));
    }
}
```

```
__kernel void CombineTwoArrays(__global float* A, __global float* B,
                                                    __global float* C) {
    int i = get_global_id(0);

    C[i] = 1.0f / (sin(A[i])*cos(B[i]) + cos(A[i])*sin(B[i]));
}
```

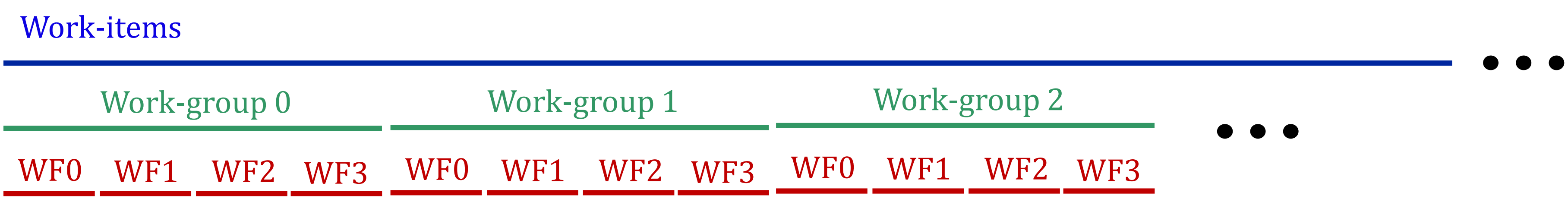


Execute in lock-step.

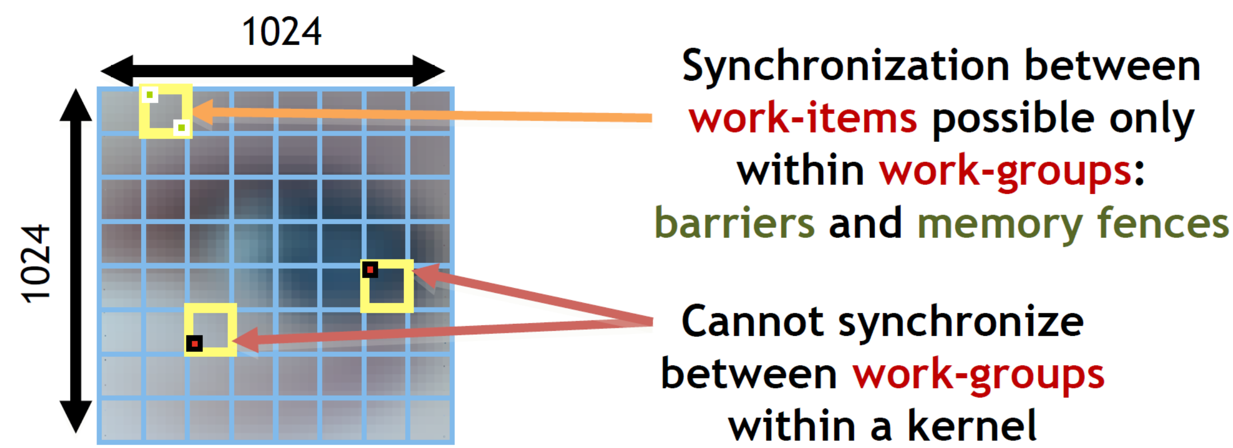
- Dimension, work-item, and work-group
  - **Work-item:** the unit of concurrent execution in OpenCL C
  - Work-items of *n* **dimension** are divided into smaller, equally sized, n-dimensional **work-groups**.

- Notes
  - Each work-group is assigned to a compute unit of device.
  - Each work-group can be assigned to the compute unit only when the device can afford the resources that the work-group needs.
  - **AMD GCN GPU**
    - Each work-group is divided into wavefronts of 64 work-items each.
    - Each wavefront is assigned to a SIMD unit within the compute unit.
    - Vector instructions executed by a wavefront are issued to the SIMD unit over four cycles.
    - Every four cycles, a new instruction can be issued to the SIMD unit.
  - Work-groups are processed in parallel in an unpredictable order.
  - Wavefronts in a work-group are processed in parallel in an unpredictable order.
  - Synchronization between work-items possible only within work-groups: barriers and memory fences.

- **1D example**
  - 134,217,728 work-items/ 256 work-items per work-group



- **2D example**
  - 1,024x1,024 work-items/ 128x28 work-items per work-group



- How do a work-item get information about itself during kernel execution?

Table 5.2 Functions related to work-items

Function	Purpose
`uint get_work_dim()`	Returns the number of dimensions in the kernel's index space
`size_t get_global_size(uint dim)`	Returns the number of work-items for a given dimension
`size_t get_global_id(uint dim)`	Returns the element of the work-item's global ID for a given dimension
`size_t get_global_offset(uint dim)`	Returns the initial offset used to compute global IDs

Table 5.3 Functions related to work-groups

Function	Purpose
`size_t get_num_groups(uint dim)`	Returns the number of work-groups for a given dimension
`size_t get_group_id(uint dim)`	Returns the ID of the work-item's work-group for a given dimension
`size_t get_local_id(uint dim)`	Returns the ID of the work-item within its work-group for a given dimension
`size_t get_local_size(uint dim)`	Returns the number of work-items in the work-group for a given dimension

Example: **Reduction**

- 1,048,576 work-items in total
- 256 work-items per work-group
- 4,096 work-groups

Imagine how this OpenCL kernel runs over the work-items/work-groups/wavefronts.

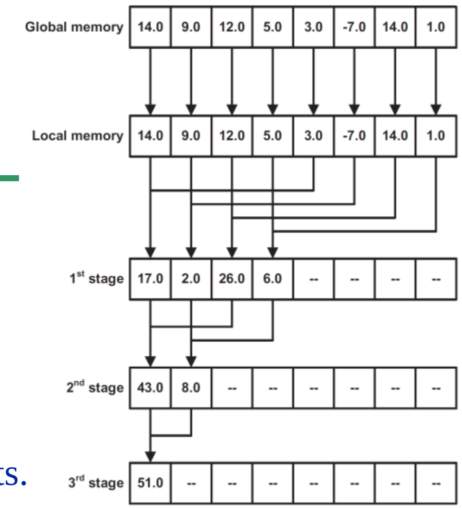


Figure 10.1 Multistage reduction in OpenCL

```
__kernel void reduction_scalar(__global float* data,
      __local float* partial_sums, __global float* output) {

   int lid = get_local_id(0);
   int group_size = get_local_size(0);

   partial_sums[lid] = data[get_global_id(0)];
   barrier(CLK_LOCAL_MEM_FENCE);

   for(int i = group_size/2; i>0; i >>= 1) {
      if(lid < i) {
         partial_sums[lid] += partial_sums[lid + i];
      }
      barrier(CLK_LOCAL_MEM_FENCE);
   }

   if(lid == 0) {
      output[get_group_id(0)] = partial_sums[0];
   }
}
```

1 **Read data to local memory**

Perform reduction stages

Write output to global memory

From *OpenCL in Action* by M. Scarpino (2012)

- **Three reduction methods on the CPU (Serial)**

```
void generate_random_float_array(float *array, int n) {
    srand((unsigned int)201803); // Always the same input data
    for (int i = 0; i < n; i++) {
        //array[i] =  2.0f*((float)rand() / RAND_MAX - 0.5f);  ← Data A
        array[i] =  100.0f*((float)rand() / RAND_MAX);  ← Data B
        //array[i] = 1.0f;
    }
}

float reduction_on_the_CPU(float *array, int n) {
    int i;
    float sum = 0.0f;

    for (i = 0; i < n; i++) {
        sum += array[i];
    }
    return sum;
}
```

```
float reduction_on_the_CPU_reduction(float *array, int n) {
    int i, j;
    float sum = 0.0f;

    float *array_b = (float *)malloc(sizeof(float)*n);
    if (array_b == NULL) {
        fprintf(stderr, "+++ Error: cannot allocate memory for array_b...\n");
        exit(EXIT_FAILURE);
    }
    memcpy(array_b, array, sizeof(float)*n);

    for (i = n/2; i > 0; i >>= 1) {
        for (j = 0; j < i; j++) {
            array_b[j] += array_b[j + i];
        }
    }
    sum = array_b[0];
    free(array_b);
    return sum;
}

float reduction_on_the_CPU_KahanSum(float *array, int n) { // From Wikipedia!!!
    int i;
    float sum = 0.0f, c = 0.0f, t, y;

    for (i = 0; i < n; i++) {
        y = array[i] - c;     t = sum + y;
        c = (t - sum) - y;    sum = t;
    }
    return sum;
}
```

- # Execution results

[Data A]

```
The number of elements to be reduced = 16777216

^^^ Test 1: serial reduction on the CPU ^^^

     * Time by host clock for simple reduction = 35.452ms

     * Time by host clock for pairwise reduction = 62.109ms

     * Time by host clock for reduction through Kahan sum = 118.810ms

^^^ Test 2: parallel reduction on the GPU ^^^

   KERNEL =  reduction_scalar

      * Time by device clock:
       - Time from QUEUED to END = 0.777ms
       - Time from SUBMIT to END = 0.757ms
       - Time from START to END = 0.756ms

      + Check PASSED!
            [-3.376091e+03/-3.376028e+03/-3.376028e+03(CPU) = -3.376033e+03(GPU)]

   KERNEL =  reduction_vector

      * Time by device clock:
       - Time from QUEUED to END = 0.403ms
       - Time from SUBMIT to END = 0.393ms
       - Time from START to END = 0.392ms

      + Check PASSED!
            [-3.376091e+03/-3.376028e+03/-3.376028e+03(CPU) = -3.376039e+03(GPU)]

Program ended with exit code: 0
```

[Data B]

```
The number of elements to be reduced = 16777216

^^^ Test 1: serial reduction on the CPU ^^^

     * Time by host clock for simple reduction = 35.445ms

     * Time by host clock for pairwise reduction = 59.801ms

     * Time by host clock for reduction through Kahan sum = 119.155ms

^^^ Test 2: parallel reduction on the GPU ^^^

   KERNEL =  reduction_scalar

      * Time by device clock:
       - Time from QUEUED to END = 0.700ms
       - Time from SUBMIT to END = 0.651ms
       - Time from START to END = 0.650ms

      + Check FAILED!
            [8.144400e+08/8.386920e+08/8.386920e+08(CPU) = 8.386916e+08(GPU)]

   KERNEL =  reduction_vector

      * Time by device clock:
       - Time from QUEUED to END = 0.403ms
       - Time from SUBMIT to END = 0.392ms
       - Time from START to END = 0.391ms

      + Check FAILED!
            [8.144400e+08/8.386920e+08/8.386920e+08(CPU) = 8.386927e+08(GPU)]

Program ended with exit code: 0
```