# GPU Programming II

Andrey Alekseenko

KTH Royal Institute of Technology & SciLifeLab

# Quiz time!

Which statement is false?

A. Host (CPU) schedules tasks on the device (GPU).
B. Host and device cannot access each others memory.
C. CPUs have more control logic per core than GPUs.
D. GPUs tend to offer more FLOPs than CPUs (per Watt/$).

# Quiz time!

Which statement is false?

A. Host (CPU) schedules tasks on the device (GPU).
B. Host and device cannot access each others memory.
C. CPUs have more control logic per core than GPUs.
D. GPUs tend to offer more FLOPs than CPUs (per Watt/$).

# Quiz time!

What is the significance of over-subscribing the GPU?

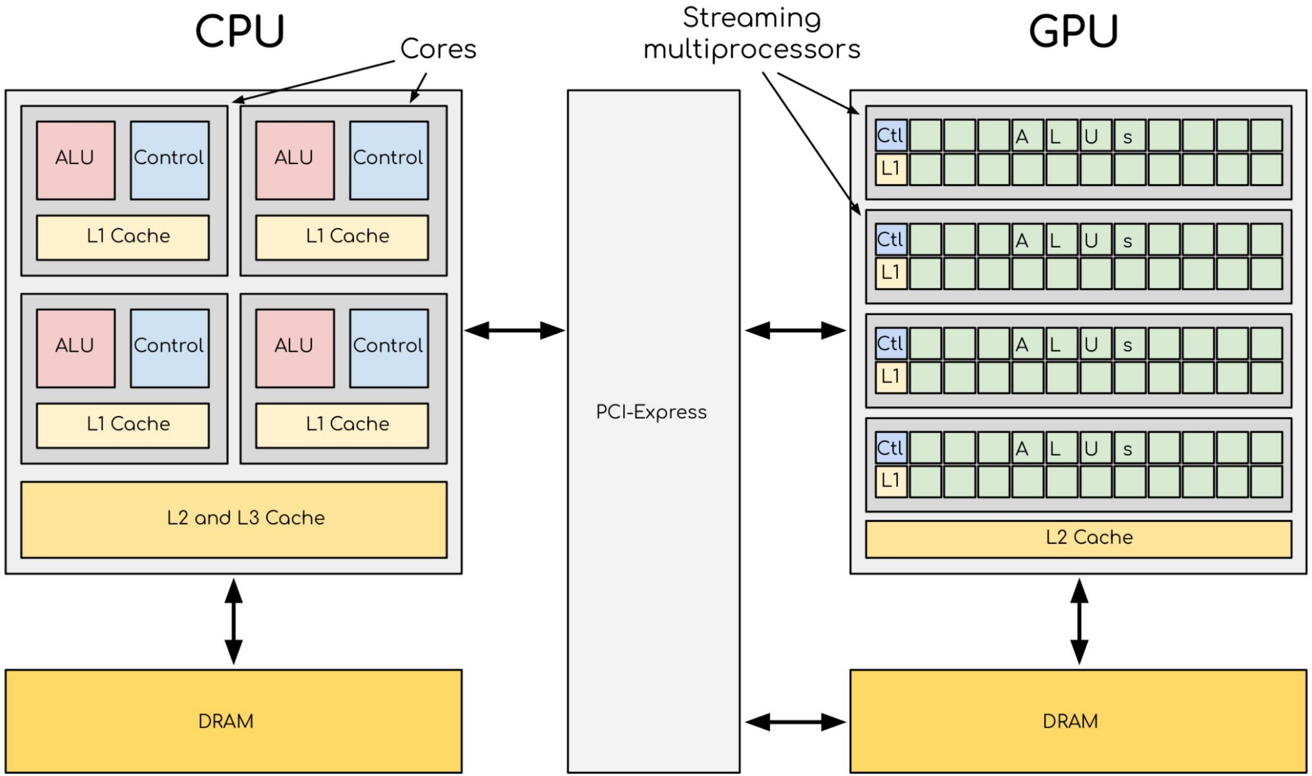
A. It reduces the overall performance of the GPU.
B. It helps to hide latencies and ensure high occupancy of the GPU.
C. It leads to a memory overflow in the GPU.
D. It ensures that there are more cores than work-groups present on the device.

# Quiz time!

What is the significance of over-subscribing the GPU?

A. It reduces the overall performance of the GPU.
B. It helps to hide latencies and ensure high occupancy of the GPU.
C. It leads to a memory overflow in the GPU.
D. It ensures that there are more cores than work-groups present on the device.

# Refresher: GPU architecture

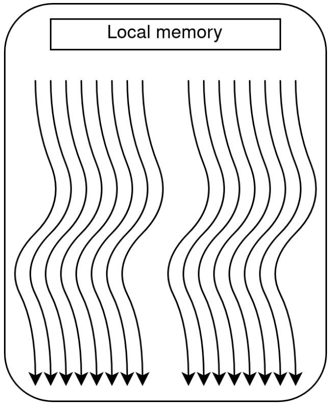



Generated with AI / DALL-E 3 / Microsoft Image Creator

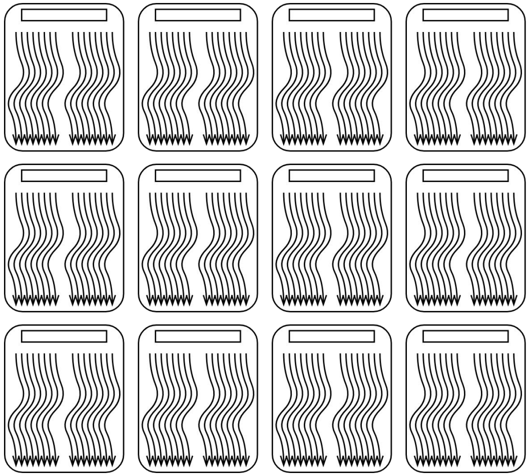
# Refresher: Data parallelism

- **Work-item** is a single thread of execution; it has its own **private** memory
- Work-items in a **work-group** can share data via **local** memory
- All work-items in all work-groups have to **global** memory
- All work-items run the same **kernel**

Work-item




Work-group



# Refresher: SYCL vector add

```
sycl::queue q{{sycl::property::queue::in_order()}};

float *Ad = sycl::malloc_device<float>(N, q); // Allocate the arrays on GPU
q.copy<float>(Ah.data(), Ad, N); // Copy the input data from host to the device

sycl::range<1> global_size(N); // Define grid dimensions
// Run our kernel
q.submit([&](sycl::handler &h) {
  h.parallel_for<class VectorAdd>(global_size, [=](sycl::item<1> threadId) {
    int tid = threadId[0]; // Get thread index
    Cd[tid] = Ad[tid] + Bd[tid]; // Do the math
  });
});

q.copy<float>(Cd, Ch.data(), N); // Copy results back to the host
// All the operations before were asynchronous!
q.wait(); // Wait for the copy to finish
sycl::free(Ad, q); // Free the GPU memory
```

# Events

- Most queue submissions return an event

```
sycl::event ev_copy = q.copy<float>(...);
sycl::event ev_kernel = q.submit([&](...) { ... }); // Will run after copy
```

- They can be used for fine-grained synchronization

```
// Will wait for the copy, but not for the kernel
ev_copy.wait();
```

- SYCL events are similar to OpenCL events, but different from CUDA/HIP events.

# Events

- Events can also be used for timing the kernels

```
sycl::queue q{{sycl::property::queue::in_order(),
               sycl::property::queue::enable_profiling()}};

sycl::event ev_kernel = q.submit([&](...) { ... });

uint64_t kernel_start = ev_kernel
          .get_profiling_info<sycl::info::event_profiling::command_start>();
uint64_t kernel_end = ev_kernel
          .get_profiling_info<sycl::info::event_profiling::command_end>();


uint64_t kernel_duration_ns = kernel_end - kernel_start;
```

# Work-item indexing

- So far, we only dealt with flat, 1-dimensional indexing:

```
q.submit([&](sycl::handler &cgh) {
  cgh.parallel_for<class Kernel>(sycl::range<1>{N},
    [=](sycl::item<1> threadId) {
      int index = threadId[0];
    }
  );
});
```

- But we often deal with 2D or 3D data
- And what about work-groups and local memory?

# Multi-dimensional indexing

```
sycl::range<1> kernel_range{N}
q.submit([&](sycl::handler &cgh) {
  cgh.parallel_for<class Kernel1D>(
    kernel_range,
    [=](sycl::item<1> threadId) {
      int index = threadId[0];
      out[index] = in[index];
    }
  );
});
```

```
sycl::range<2> kernel_range{W, H};
q.submit([&](sycl::handler &cgh) {
  cgh.parallel_for<class Kernel2D>(
      kernel_range,
      [=](sycl::item<2> threadId) {
        int x_id = threadId[0];
        int y_id = threadId[1];
        int index = x_id * H + y_id;
        out[index] = in[index];
      }
  );
});
```

# Multi-dimensional indexing

- Can be up to 3D
- It is just syntactic sugar!

```
sycl::item<1> idx{x}, sycl::range<1> r{A};
int index = idx[0] = x;

sycl::item<2> idx{x, y}, sycl::range<2> r{A, B};
int index = idx[0] * r[1] + idx[1] = x * B + y;

sycl::item<3> idx{x, y, z}, sycl::range<3> r{A, B, C};
int index = (idx[0] * r[1] + idx[1]) * r[2] + id[2]
          = (x * B + y) * C + z;
```

# Multi-dimensional indexing

- Can be up to 3D
- It is just syntactic sugar!

```
sycl::item<1> idx{x}, sycl::range<1> r{A};
int index = idx[0] = x;

sycl::item<2> idx{x, y}, sycl::range<2> r{A, B};
int index = idx[0] * r[1] + idx[1] = x * B + y;

sycl::item<3> idx{x, y, z}, sycl::range<3> r{A, B, C};
int index = (idx[0] * r[1] + idx[1]) * r[2] + id[2]
          = (x * B + y) * C + z;
```

```
int index = idx.get_linear_id();
```

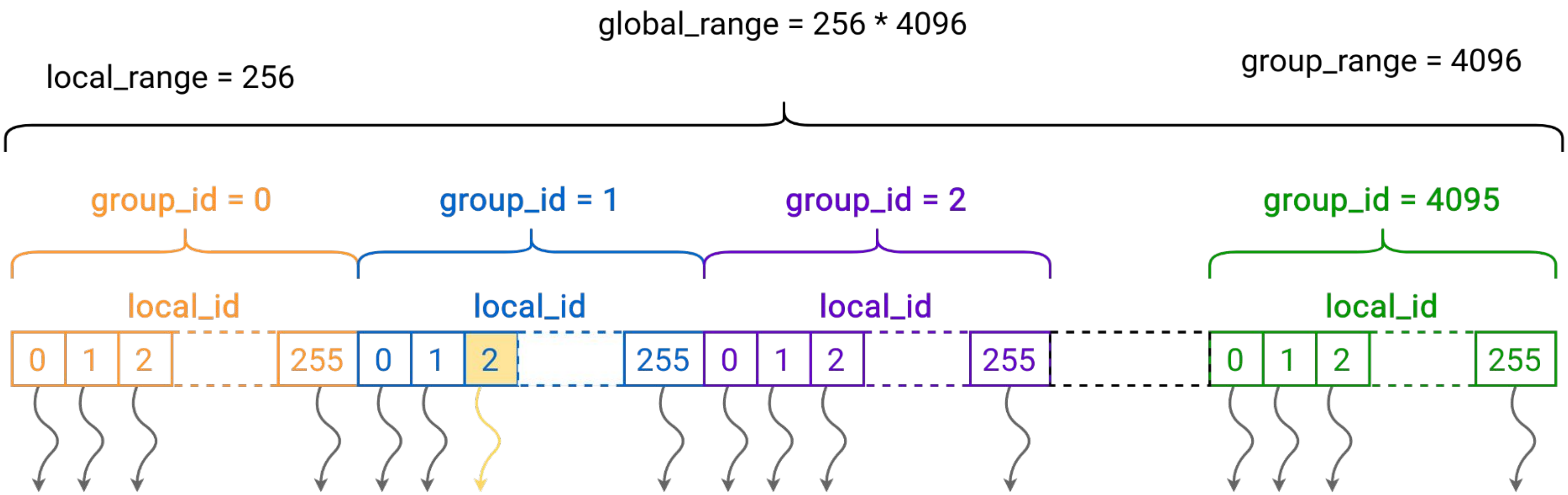
# Multi-dimensional indexing

- Can be up to 3D
- It is just syntactic sugar!
- Portability warning: order is different in CUDA/HIP/OpenCL!

```
// SYCL
sycl::item<2> idx{x, y}; sycl::range<2> r{A, B};
int index = idx[0] * r[1] + idx[1] = x * B + y;
```

```
// CUDA/HIP
int2 idx{x, y}; dim2 r{A, B};
int index = idx.y * r.y + idx.x = y * A + x;
```

# Work-group indexing




global_id = group_id * local_range + local_id

global_id = 1 * 256 + 2 = 258

# Range vs. NDRange

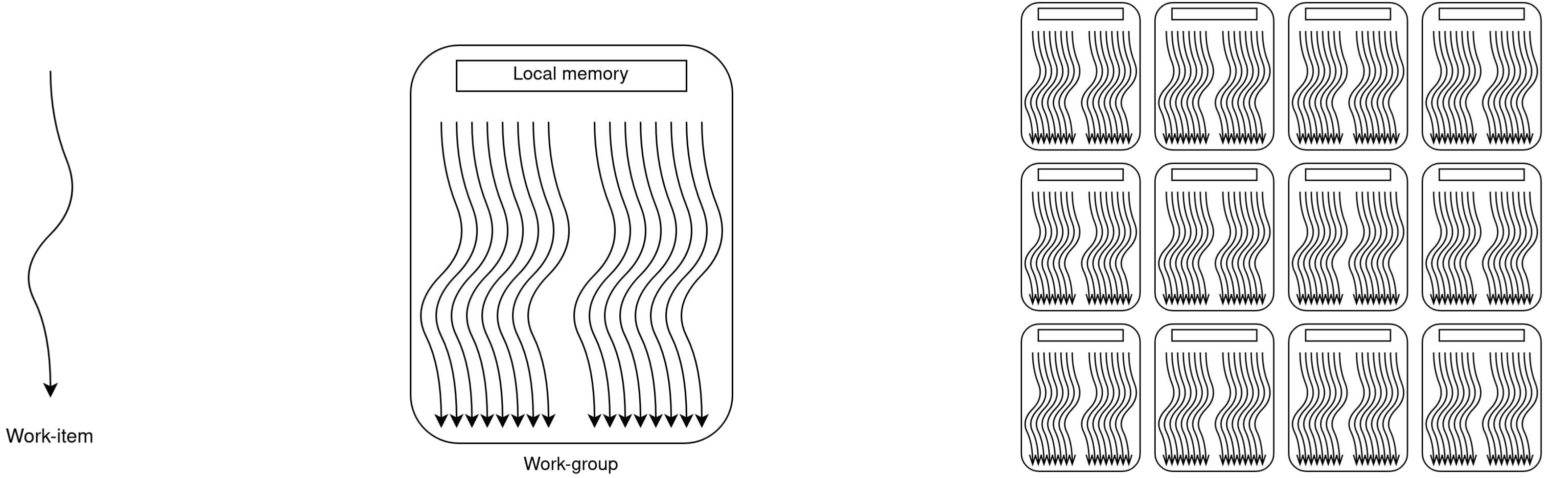
```
sycl::range<1> global_range{n}; // n work-items in total
q.submit([&](sycl::handler &cgh) {
  cgh.parallel_for(global_range, [=](sycl::item<1> item) {
    int global_id = item.get_id(0); // [0; n)
  });
});
```

# Range vs. NDRange

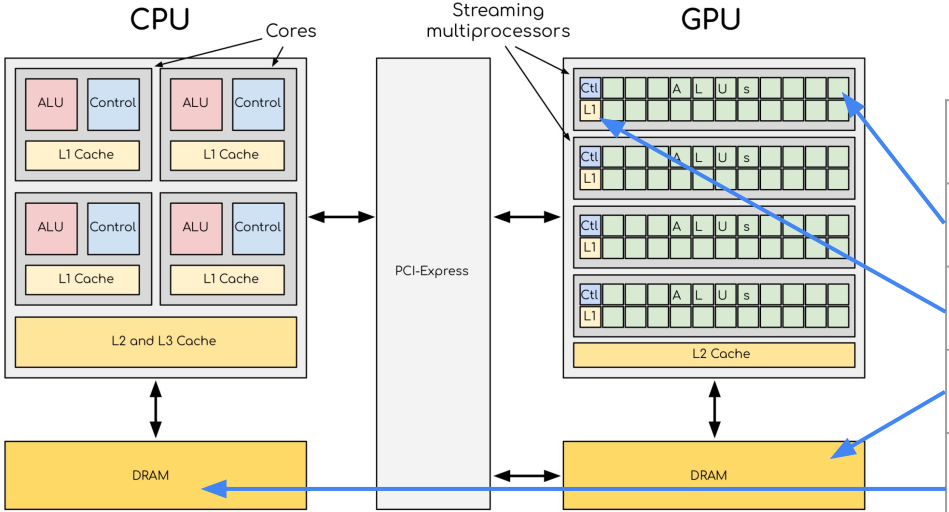
```
sycl::range<1> global_range{n}; // n work-items in total
q.submit([&](sycl::handler &cgh) {
  cgh.parallel_for(global_range, [=](sycl::item<1> item) {
    int global_id = item.get_id(0); // [0; n)
  });
});

sycl::range<1> global_range{n}; // n work-items in total
sycl::range<1> local_range{k}; // k work-items in each work-group
sycl::nd_range<1> kernel_range{global_size, local_size};
q.submit([&](sycl::handler &cgh) {
  cgh.parallel_for(kernel_range, [=](sycl::nd_item<1> item) {
    int local_id = item.get_local_id(0);   // 0..k
    int group_id = item.get_group(0);      // 0..n/k
    int global_id = item.get_global_id(0); // 0..n
  });
});
```

# Why NDRange?

- Local memory
- Collective operations
- Another knob for performance tuning



# Local memory




| Memory | Scope | Speed |
|---|---|---|
| private | work-item | fast |
| local | work-group | fast |
| device | device | slow |
| host | device and host | very slow |

# Local memory

| SYCL | Scope | Speed |
|---|---|---|
| private | work-item | fast |
| local | work-group | fast |
| device | device | slow |

```
q.submit([&](sycl::handler &cgh) {
  cgh.parallel_for(kernel_range, [=](sycl::nd_item<1> item) {
    int local_id = item.get_local_id(0); // 0..wg_size
    int global_id = item.get_global_id(0); // 0..n
    float x1 = input_data[global_id];
    float x2 = input_data[global_id + wg_size - 2 * local_id];
  });
});
```

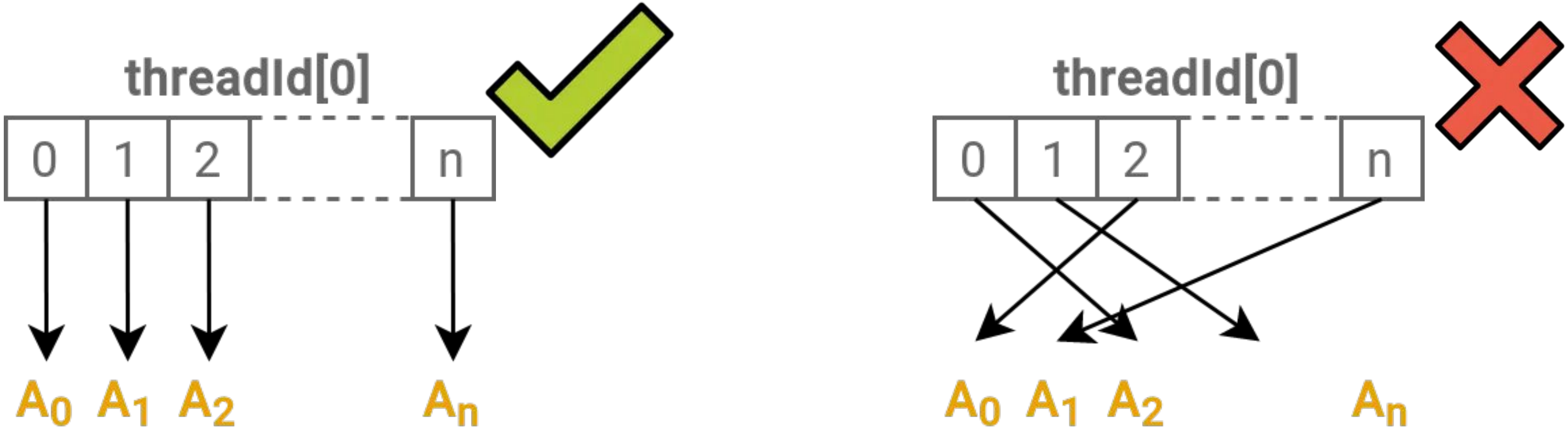
# Local memory

| SYCL | Scope | Speed |
|---|---|---|
| **private** | work-item | fast |
| **local** | work-group | fast |
| **device** | device | slow |

```
q.submit([&](sycl::handler &cgh) {
  cgh.parallel_for(kernel_range, [=](sycl::nd_item<1> item) {
    int local_id = item.get_local_id(0); // 0..wg_size
    int global_id = item.get_global_id(0); // 0..n
    float x1 = input_data[global_id];
    float x2 = input_data[global_id + wg_size - 2 * local_id];
  });
});
```

# Coalesced memory access

- When adjacent threads access adjacent elements, operations are combined
- Exact rules are complicated and hardware dependent

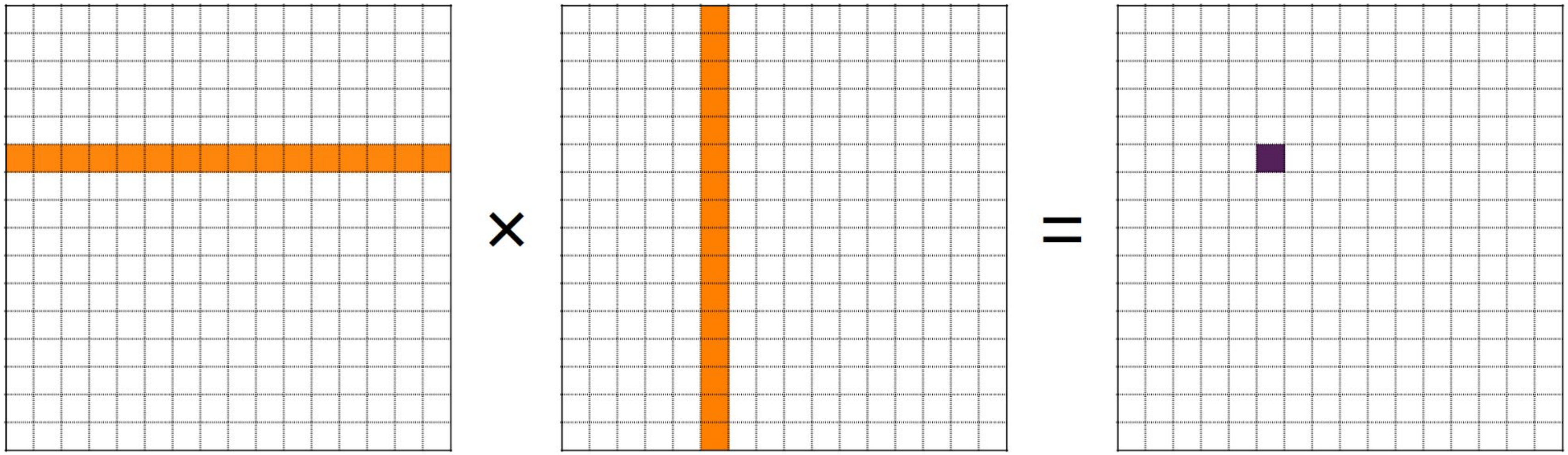


# Local memory

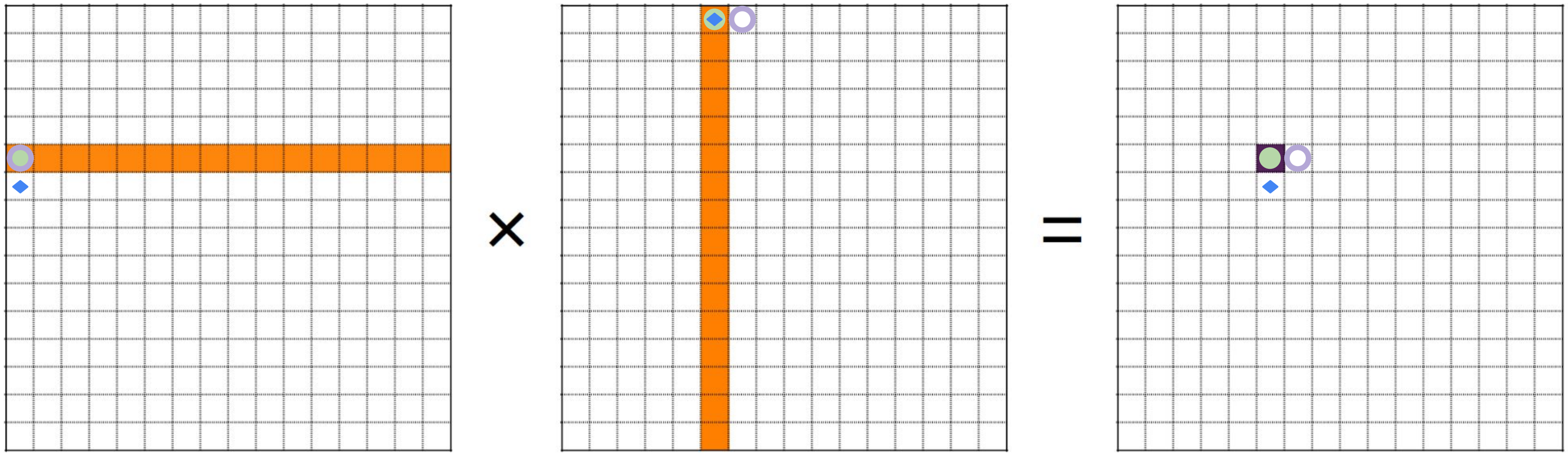
| SYCL | Scope | Speed |
|---|---|---|
| **private** | work-item | fast |
| **local** | work-group | fast |
| **device** | device | slow |

```
q.submit([&](sycl::handler &cgh) {
  sycl::local_accessor<float, 1> local_buffer{wg_size, cgh};
  cgh.parallel_for(kernel_range, [=](sycl::nd_item<1> item) {
    int local_id = item.get_local_id(0); // 0..wg_size
    int global_id = item.get_global_id(0); // 0..n
    local_buffer[local_id] = input_data[global_id];
    item.barrier();
    float x2 = local_buffer[wg_size - local_id];
  });
});
```

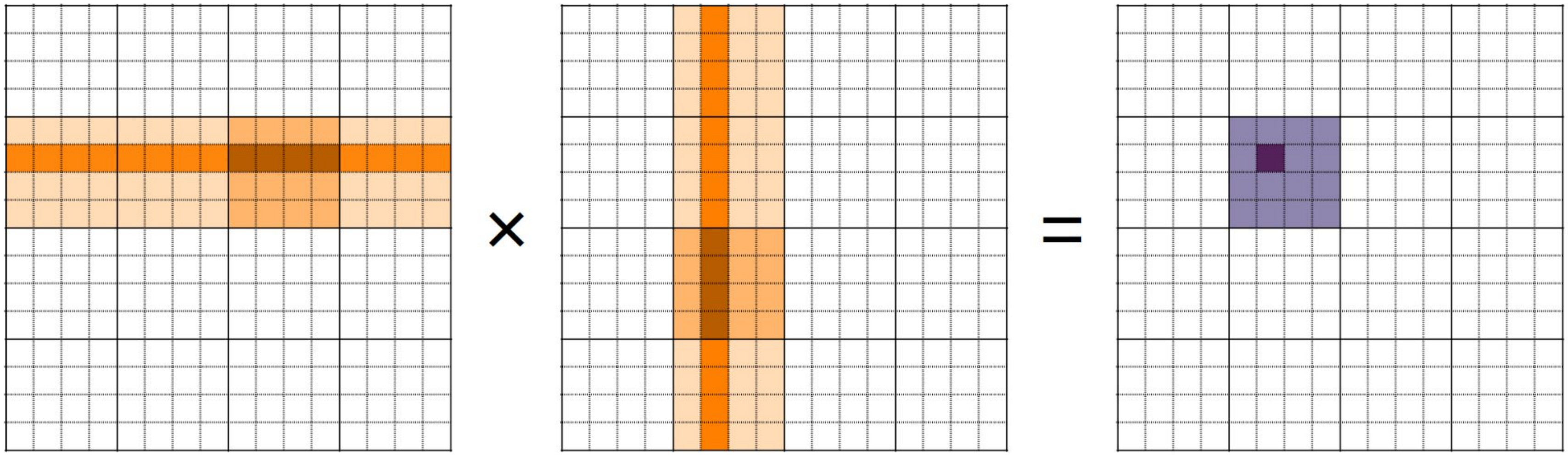
# Matrix multiplication



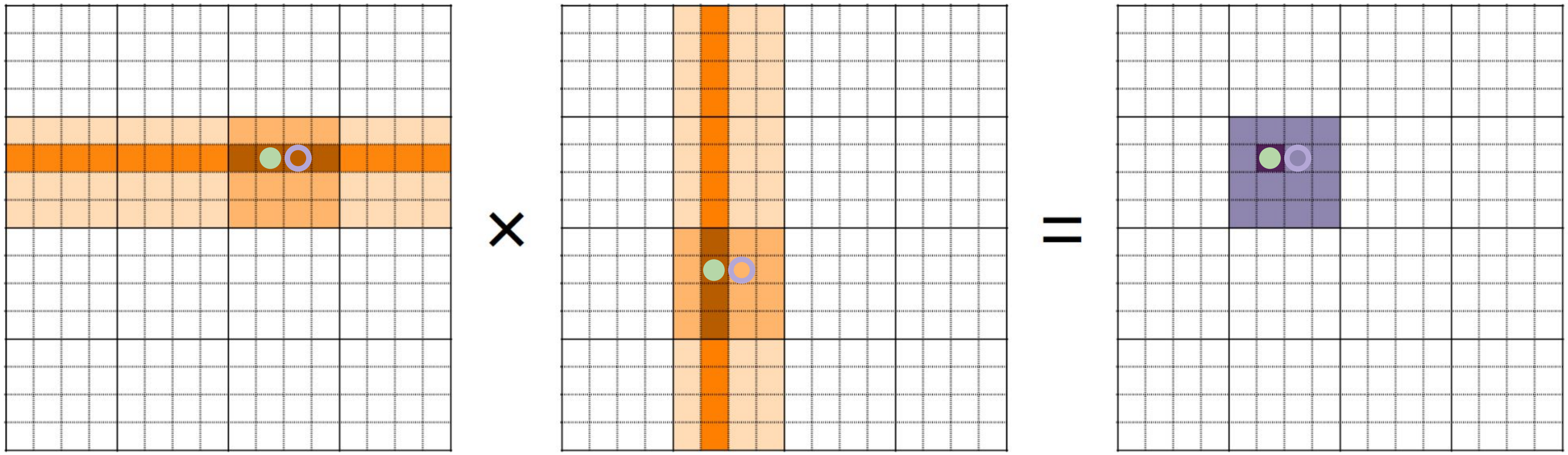
# Matrix multiplication



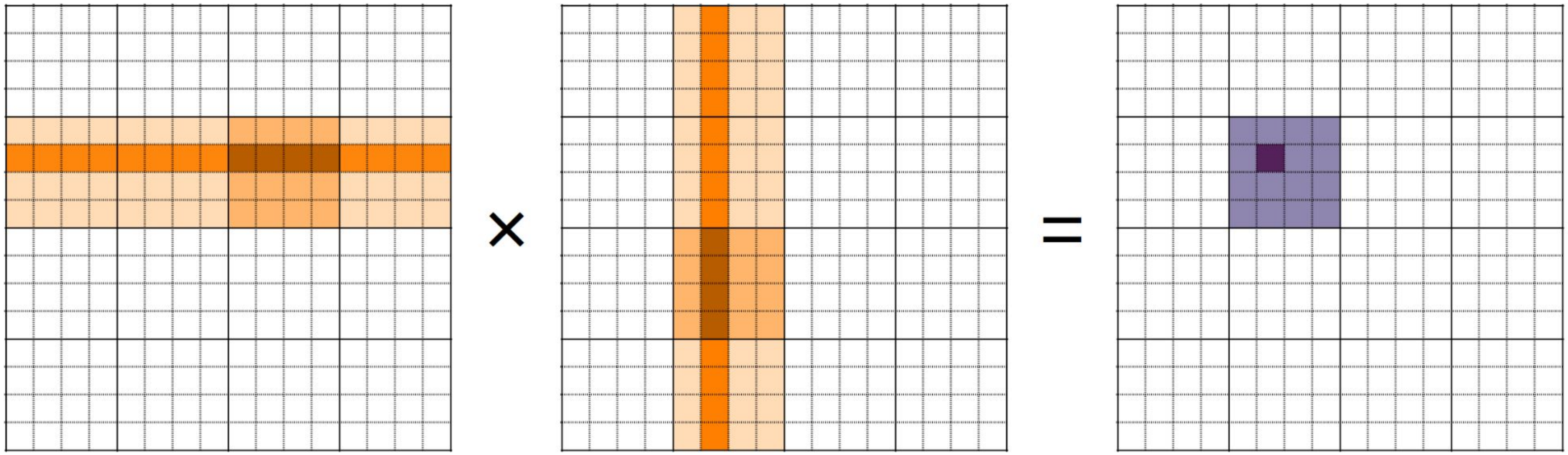
# Matrix multiplication



# Matrix multiplication



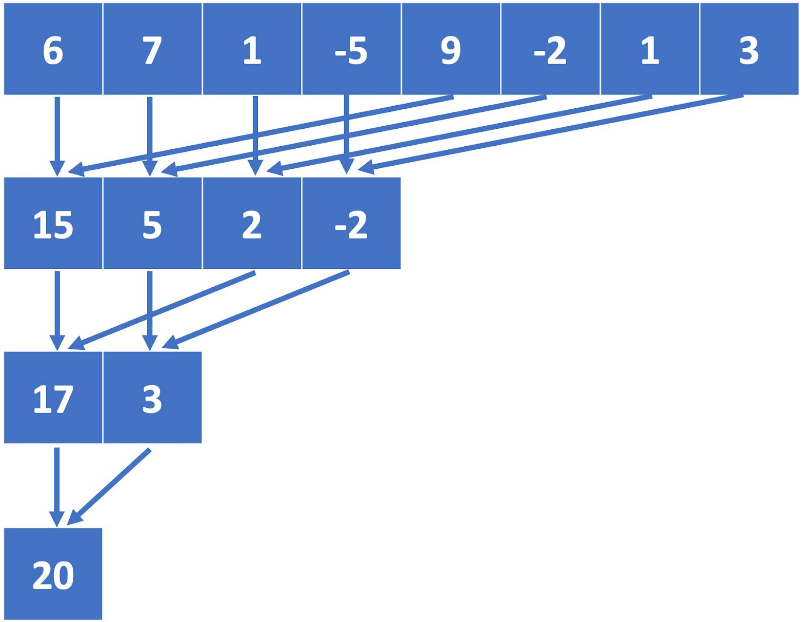
# Matrix multiplication



There are also specialized hardware units…

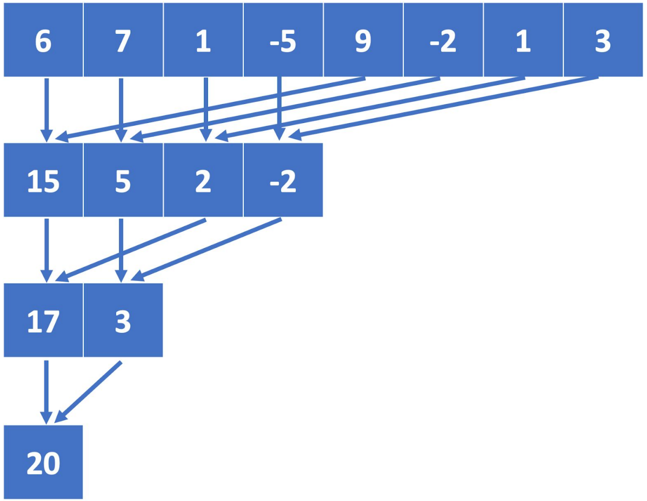
Just use DGEMM

# Reductions



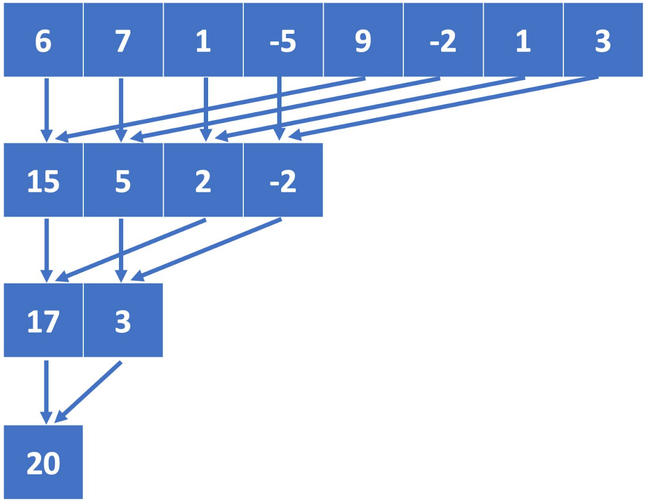
# Reductions

```
q.submit([&](sycl::handler &cgh) {
  sycl::local_accessor<int, 1> buf{{wg_size}, cgh};
  cgh.parallel_for(kernel_range,
    [=](sycl::nd_item<1> item) {
      int id = item.get_local_id(0);
      buf[id] = x[item.get_global_id(0)];
      for (int s = wg_size / 2; s > 0; s /= 2) {
        item.barrier();
        if (id < s) {
          buf[id] += buf[id + s];
        }
      }
      if (id == 0) {
        sum[item.get_group(0)] = buf[0];
      }
    });
});
```



# Reductions

```
q.submit([&](sycl::handler &cgh) {
  sycl::local_accessor<int, 1> buf{{wg_size}, cgh};
  cgh.parallel_for(kernel_range,
    [=](sycl::nd_item<1> item) {
      int id = item.get_local_id(0);
      buf[id] = x[item.get_global_id(0)];
      for (int s = wg_size / 2; s > 0; s /= 2) {
        item.barrier();
        if (id < s) {
          buf[id] += buf[id + s];
        }
      }
      if (id == 0) {
        sum[item.get_group(0)] = buf[0];
      }
    });
});
```




Can also use sub-groups…

# Quiz time!

Which of the following programs will **not** benefit from using local memory?

A. Matrix transpose.
B. Matrix multiplication.
C. Vector addition.
D. Vector dot product.

# Quiz time!

Which of the following programs will **not** benefit from using local memory?

A. Matrix transpose.
B. Matrix multiplication.
C. Vector addition.
D. Vector dot product.

# Dealing with large kernels

```
int do_compute() {
  sycl::queue q;
  // allocate memory
  // initialize data
  q.submit([&](sycl::handler &h) {
    h.parallel_for<class ComplexKernelA>(global_size_1,
      [=](sycl::id<1> threadId) {
        // Do lots of math
        // Like, really a lot
        // Maybe a thousand lines
      }
    );
  });
  // do things
  q.submit([&](sycl::handler &h) {
    h.parallel_for<class ComplexKernelB>(global_size_2,
      [=](sycl::id<1> threadId) {
        // Oh no, more code
        // Much more
      }
    );
  });
  // copy back
}
```

# Dealing with large kernels

```
int main() {
  q.submit([&](sycl::handler &h) {
    h.parallel_for<class VectorAdd>(
      global_size,
      [=](sycl::id<1> threadId) {
        int tid = threadId.get(0);
        Ad[tid] = Ad[tid] + Bd[tid];
      }
    );
  });
}
```

```
auto kernel(float *A, float *B, int N) {
  return [=](sycl::id<1> threadId) {
    int tid = threadId.get(0);
    A[tid] = A[tid] + B[tid];
  };
}

int main() {
  q.submit([&](sycl::handler &h) {
    h.parallel_for<class VectorAdd>(
      global_size,
      kernel(Ad, Bd, N)
    );
  });
}
```