Lesson 24: Computing $\pi$ – Part II
Recap

- Introduced goal: Compute π in parallel
- Introduced initial algorithm: find points in circle
- Introduced kernel requirements
- Introduced parameter passing
- Next: Find the points!
Grid dimensionality

- No spatial relationship between points
- Points can be evaluated independently
- This makes a multi-dimensional grid unnecessary

```cpp
struct PixelFinderKernel
{
    template <typename Acc>
    ALPAKA_FN_ACC void operator()(Acc const & acc, Points points, float r) const {
        using namespace alpaka;

        uint32_t gridThreadIdx = idx::getIdx<Grid, Threads>(acc)[0];
        /* ... */
    }
};
```
Lesson 24: Computing π – Part II

Accessing memory

- Iterating over a buffer works differently in alpaka
- for loop: One thread accesses elements sequentially
- Thread index: Threads access elements in parallel
- If required, you can mix both approaches!

```cpp
// Using a for loop for buffer access
for(std::size_t i = 0; i < n; ++i)
{
    float x = points.x[i];
    float y = points.y[i];
}

// Using the thread index for buffer access
float x = points.x[gridThreadIdx];
float y = points.y[gridThreadIdx];
```
Lesson 24: Computing π – Part II

Computing the distance

- Use Pythagoras’ theorem for computing the distance
- Use `math::sqrt()` for computing the square root
  - Requires the `acc` parameter!

```cpp
/* ... */
float d = math::sqrt(acc, x * x + y * y);

bool isInside = (d <= r);

points.inside[gridThreadIdx] = isInside;
```

The complete Kernel

```cpp
struct PixelFinderKernel {
    template <typename Acc>
    ALPAKA_FN_ACC void operator()(Acc const & acc, Points points, float r) const {
        uint32_t gridThreadIdx = idx::getIdx<Grid, Threads>(acc)[0];

        float x = points.x[gridThreadIdx];
        float y = points.y[gridThreadIdx];
        float d = math::sqrt(acc, x * x + y * y);

        bool isInside = (d <= r);

        points.inside[gridThreadIdx] = isInside;
    }
};
```