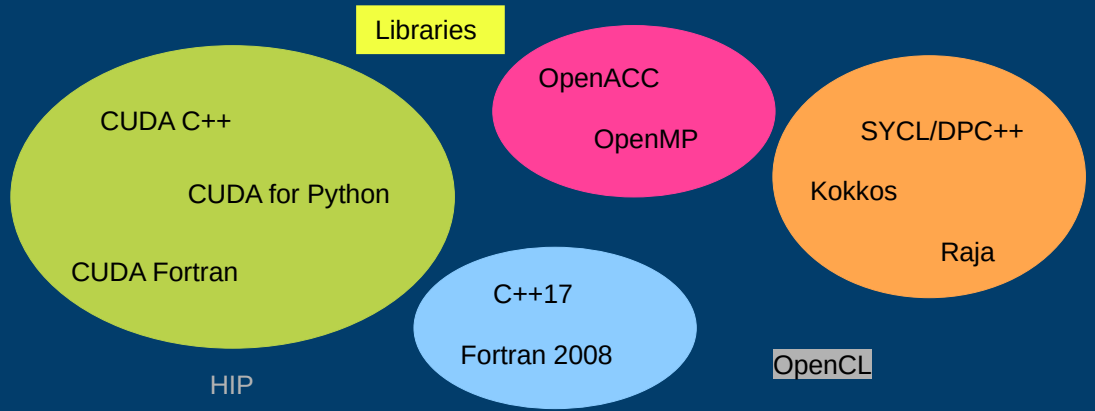
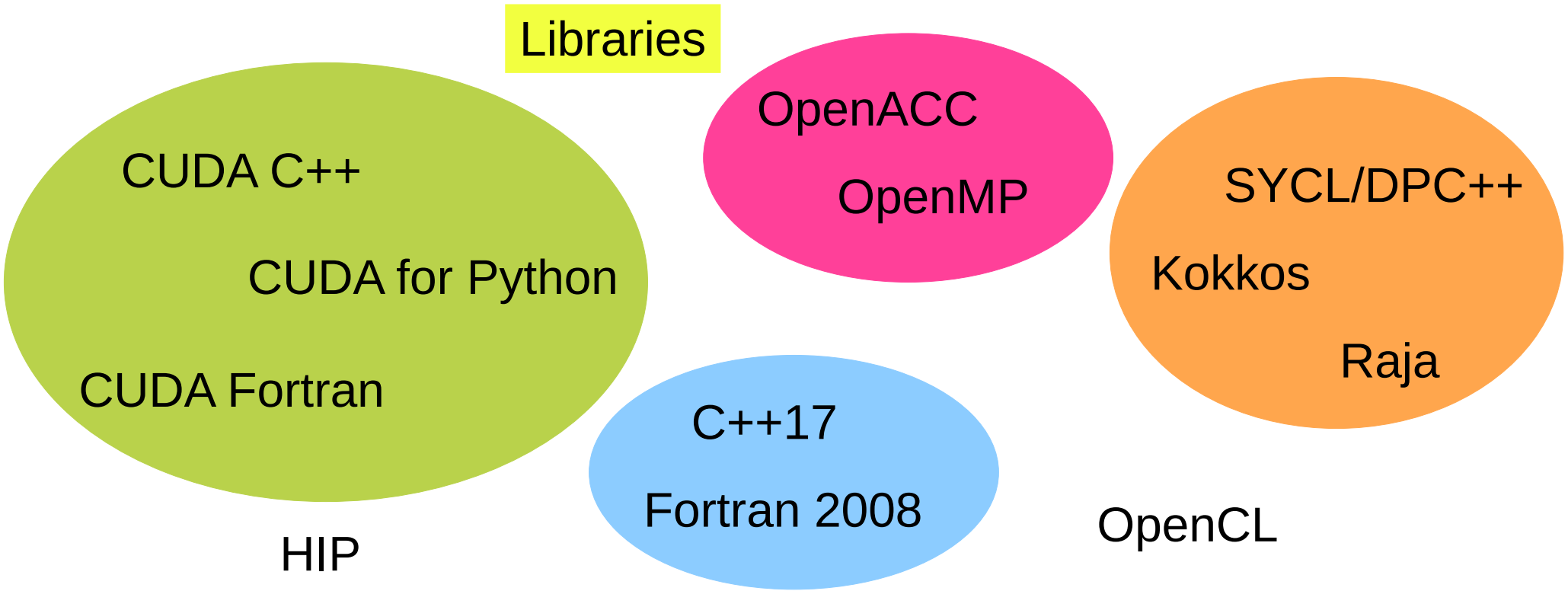
BEYOND CUDA C++

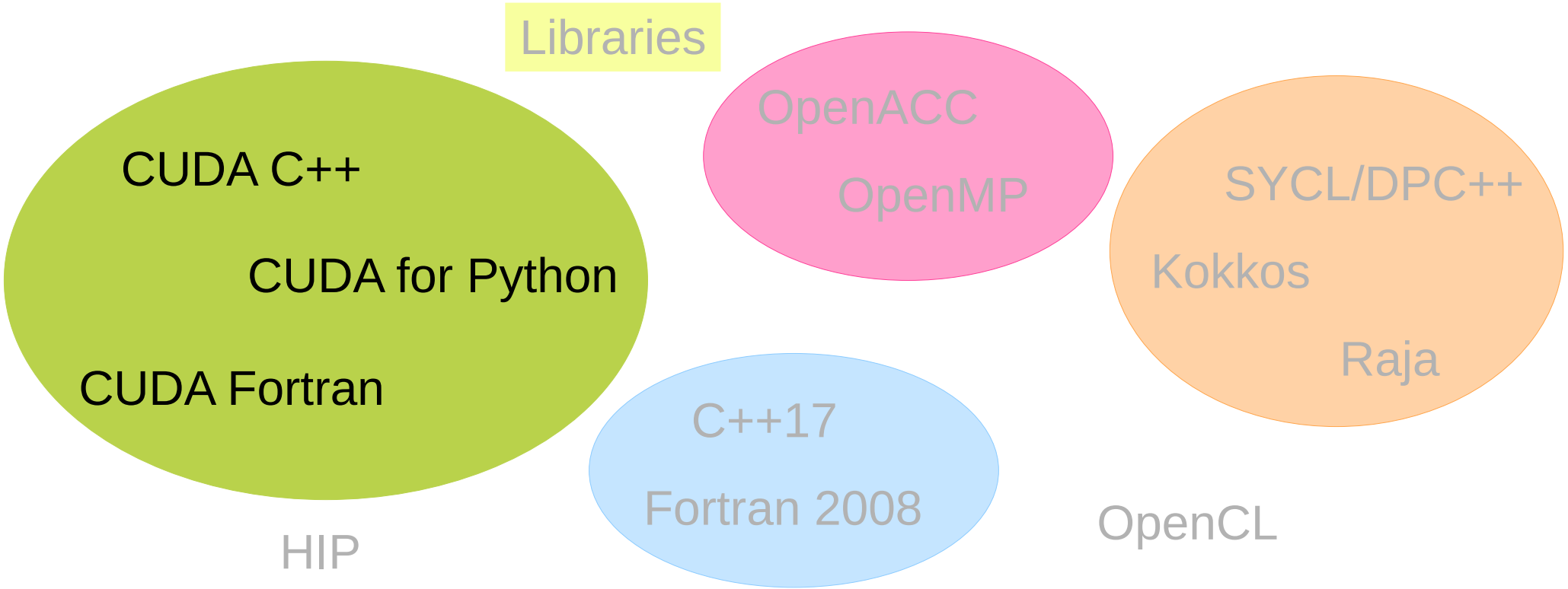
21 January 2021 | JAN H. MEINKE



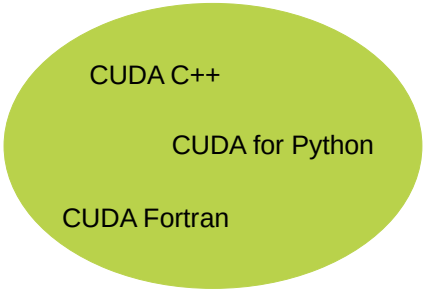
PROGRAMMING MODELS



THE CUDA FAMILY



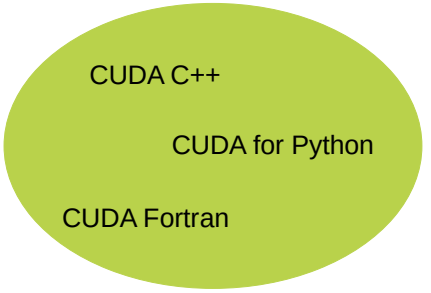
CUDA FOR FORTRAN




```
module mathOps
contains
 attributes(global) subroutine vecAdd(a,b)
 implicit none

  real :: a(:)
  real,value :: b
  integer :: i, n

  n = size(a)
  i= blockDim%x*(blockIdx%x-1)+threadIdx%x
  if (i=<n) then
    a(i)=a(i)+b
  endif

 end subroutine vecAdd
end module mathOps
```

```
program testVecAdd
use mathOps
use cudafor
 integer, parameter :: N = 40000
 real :: a(N)
 real,device :: a_d(N)
 integer tBlock, grid

 a = 10.0
 a_d = a
 tBlock = 256
 grid = ceiling(real(N)/tBlock)
 call vecAdd<<<grid,tBlock>>>(a_d,1.0)
 a = a_d
 print*,"max_diff=", maxval(a-11.0)

end program testVecAdd
```

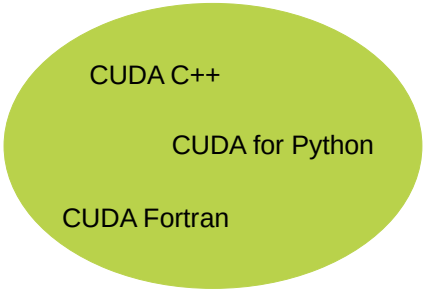
CUDA FOR FORTRAN




```
module mathOps
contains
 attributes(global) subroutine vecAdd(a,b)
 implicit none

  real :: a(:)
  real,value :: b
  integer :: i, n

  n = size(a)
  i= blockDim%x*(blockIdx%x-1)+threadIdx%x
  if (i=<n) then
    a(i)=a(i)+b
  endif

 end subroutine vecAdd
end module mathOps
```

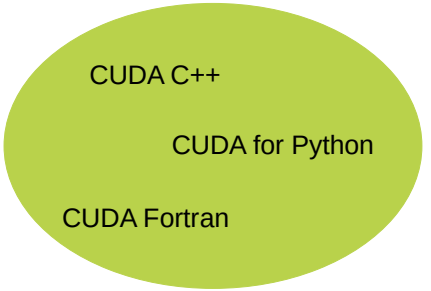
```
program testVecAdd
use mathOps
use cudafor
 integer, parameter :: N = 40000
 real :: a(N)
 real,device :: a_d(N)
 integer tBlock, grid

 a = 10.0
 a_d = a
 tBlock = 256
 grid = ceiling(real(N)/tBlock)
 call vecAdd<<<grid,tBlock>>>(a_d,1.0)
 a = a_d
 print*,"max_diff=", maxval(a-11.0)

end program testVecAdd
```

CUDA FOR PYTHON



- Part of Numba
- Open source, https://numba.pydata.org/
- JIT compiles Python with the help of LLVM
- CUDA's LLVM backend can be used to compile for GPU
- Very similar module for ROCm exists as well

CUDA FOR PYTHON

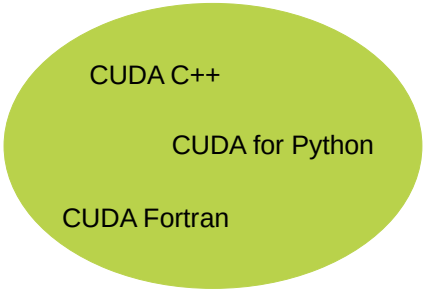


vectorize

```
from numba import vectorize

@vectorize("int64(complex128, int64)",
           target="cuda")
def escape_time(p, maxtime):
    """Perform the Mandelbrot iteration until
    it's clear that p diverges or the maximum
    number of iterations has been reached.
    """
    z = 0j
    for i in range(maxtime):
        z = z ** 2 + p
        if abs(z) > 2:
            return i
    return maxtime
```

```
import numpy
x = numpy.linspace(-2, 2, 500)
y = numpy.linspace(-1.5, 1.5, 375)
zr, zc = numpy.meshgrid(x, 1j * y)

M = escape_time(zr + zc, 500)
```

CUDA FOR PYTHON

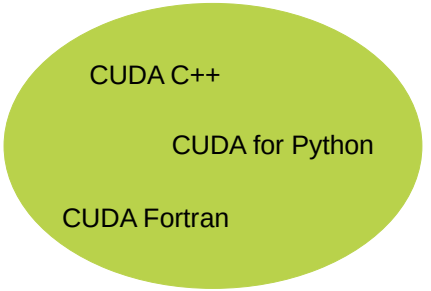


cuda.jit

```
@cuda.jit
def mm(A, B, C):
    """Take two matrices A and B, calculate
       the matrix product and store it in C."""
    i, j = cuda.grid(2)
    if i < A.shape[0] and j < B.shape[1]:
        tmp = 0
        for k in range(A.shape[1]):
            tmp += A[i, k] * B[k, j]
        C[i, j] = tmp
```

```
import numpy
A = numpy.random.random((1024, 512))
B = numpy.random.random((512, 256))
C = numpy.empty(1024, 256)
block = (32, 32)
grid = (1024 // 32, 256 // 32)
mm[grid, block](A, B, C)
```

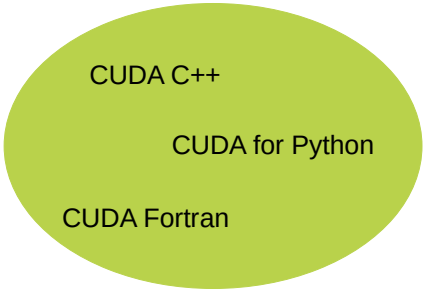
CUDA FOR PYTHON



cuda.jit

```
@cuda.jit
def mm(A, B, C):
    """Take two matrices A and B, calculate
       the matrix product and store it in C."""
    i, j = cuda.grid(2)
    if i < A.shape[0] and j < B.shape[1]:
        tmp = 0
        for k in range(A.shape[1]):
            tmp += A[i, k] * B[k, j]
        C[i, j] = tmp
```

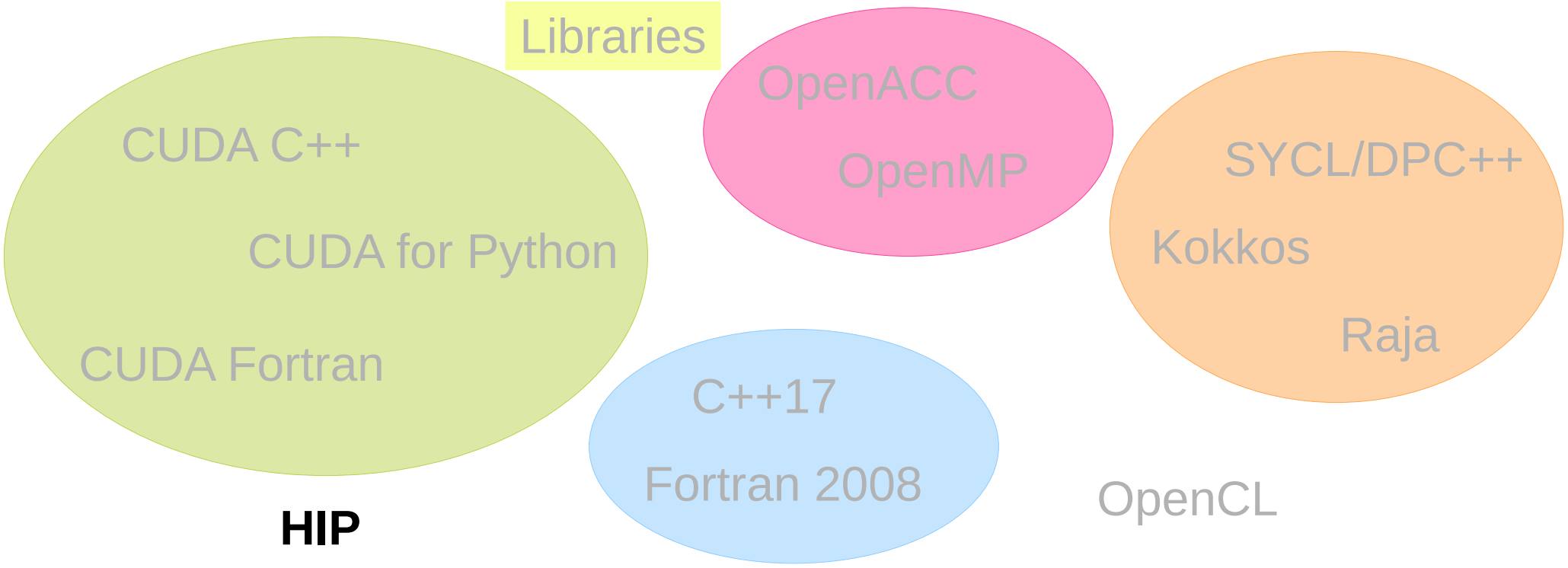
```
import numpy
A = numpy.random.random((1024, 512))
B = numpy.random.random((512, 256))
C = numpy.empty(1024, 256)
block = (32, 32)
grid = (1024 // 32, 256 // 32)
mm[grid, block](A, B, C)
```



Exercises

exercises/01-cuda-family

HETEROGENEOUS-COMPUTE INTERFACE FOR PORTABILITY (HIP)



HIP

- C++ (very similar to CUDA)
- Subset of CUDA, but specialization possible
- Open source project @ https://github.com/ROCm-Developer-Tools/HIP
- CUDA code can be converted using hipify (Cafe framework was ported to hip that way.)
- nvcc or hip-clang as backend

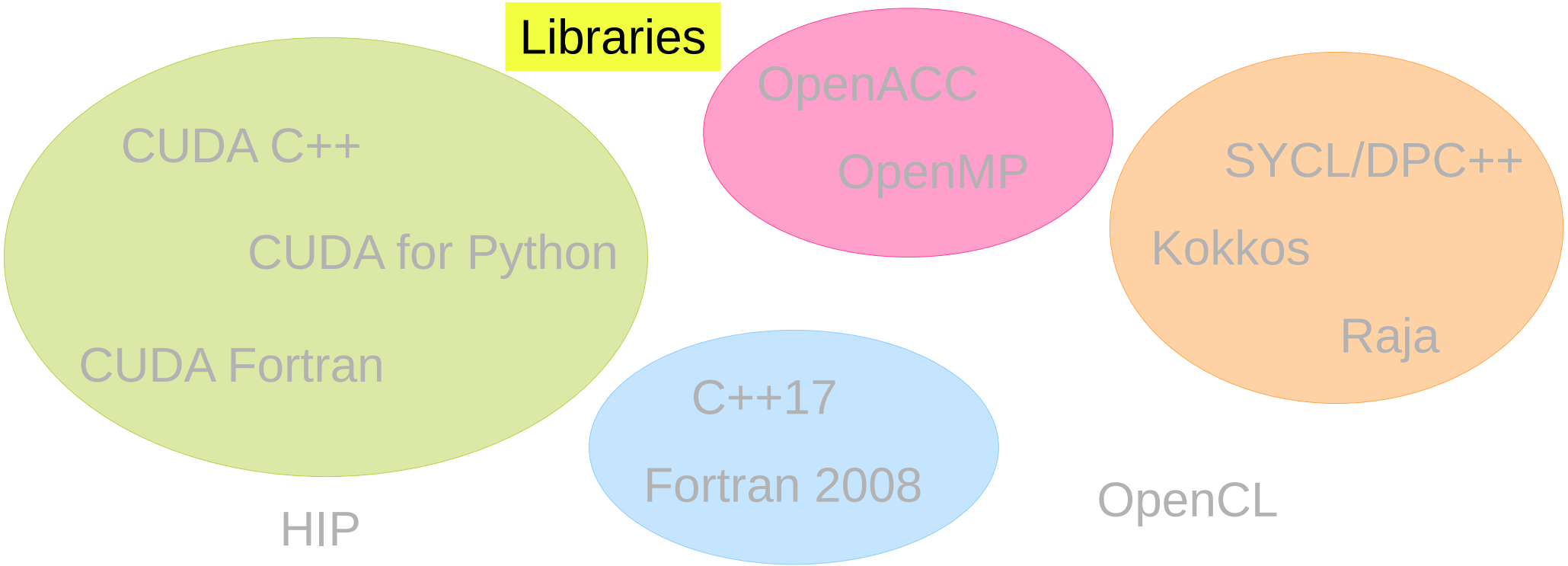
HIP KERNEL

```
__global__ void saxpy(float* y, const float* x, float a, size_t N){
    int i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x;
    if (i < N) {
        y[i] = a * x[i] + y[i];
    }
}
```

HIP API

```
// ...
    hipMalloc(&x_gpu, N * sizeof(float));
    hipMalloc(&y_gpu, N * sizeof(float));
    hipMemcpy(x_gpu, x.data(), N * sizeof(float), hipMemcpyHostToDevice);
    hipMemcpy(y_gpu, y.data(), N * sizeof(float), hipMemcpyHostToDevice);

    size_t threadsPerBlock = 256;
    size_t blocks = (N % threadsPerBlock == 0) ? N / threadsPerBlock : N / threadsPerBlock + 1;

    hipLaunchKernelGGL(saxpy, dim3(blocks), dim3(threadsPerBlock), 0, 0, y_gpu, x_gpu, a, N);

    hipMemcpy(y.data(), y_gpu, N * sizeof(float), hipMemcpyDeviceToHost);
 // ...
```

LIBRARIES

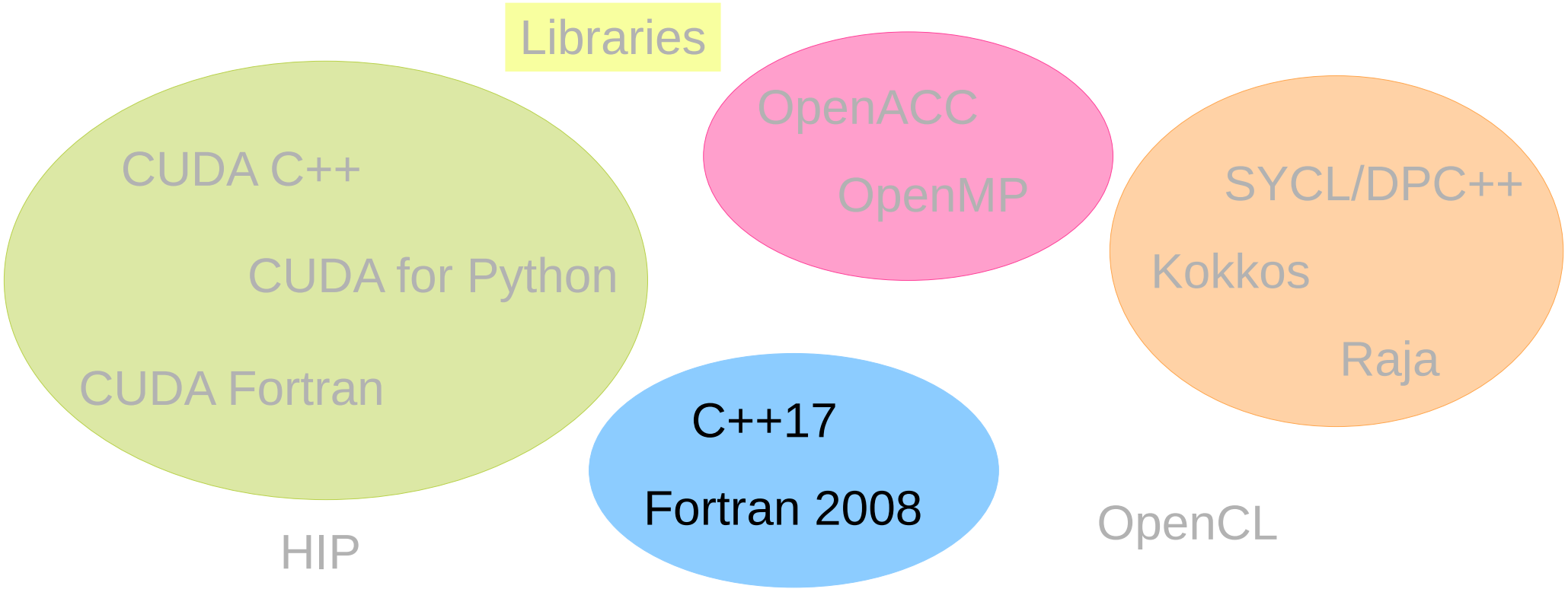

Libraries
CUDA C++
CUDA for Python
CUDA Fortran
HIP
OpenACC
OpenMP
C++17
Fortran 2008
SYCL/DPC++
Kokkos
Raja
OpenCL

LIBRARIES

- cuBLAS, cuSPARSE, cuRAND, cuSOLVER, cuFFT, cuDNN, …
- Interfaces for different languages, but the best integration is cupy:
  - cp.random→ cuRAND
  - @ → cuBLAS
  - …

```
import cupy as cp
import sys, time
width = 100 if len(sys.argv) < 2
          else int(sys.argv[1])
A = cp.random.random((width, width))
B = cp.random.random((width, width))
device = cp.cuda.Device()
t0 = time.time()
C = A @ B
device.synchronize()
t1 = time.time()
print(f"Call to cublasDGEMM took"
      "{t1 - t0:.3f} s.")
```

LANGUAGE STANDARDS



PARALLEL STL (PSTL)

C++17

Fortran 2008

execution::par

sort

execution::unseq

transform

execution::seq

for_each

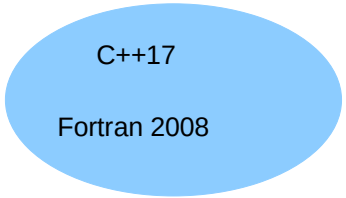
reduce

execution::par_unseq

accumulate

https://en.cppreference.com/w/cpp/algorithm

A PSTL EXAMPLE




```
#include <execution>
#include <iostream>
#include <numeric>
#include <vector>

int main(){
    size_t N = 10'000;
    std::vector x(N, 1.0 / N);
    std::cout << "The sum of the elements of x is " <<
       std::reduce(std::execution::par_unseq, x.begin(), x.end(), 0.0);
}
```

FUNCTION OBJECT (AKA FUNCTOR)

C++17

Fortran 2008

```
template <class T>
class In_range {
    const T val1;
    const T val2;
public:
    In_range(const T& v1, const T& v2) : val1(v1), val2(v2) {}
    bool operator()(const T& x) const {return (x >= val1 && x < val2);}
};
```

Can be used, e.g., in std::count():

```
std::count_if(v.begin(), v.end(), In_range<int>(3, 6));
```

LAMBDAS

C++17

Fortran 2008

```
auto lambda = [](const int& x){return (x >= 3 && x < 6);}
```

Can be used, e.g., in std::count_if():

```
std::count_if(v.begin(), v.end(), [](const int& x){return (x >= 3 && x < 6);});
```

LAMBDAS

C++17

Fortran 2008

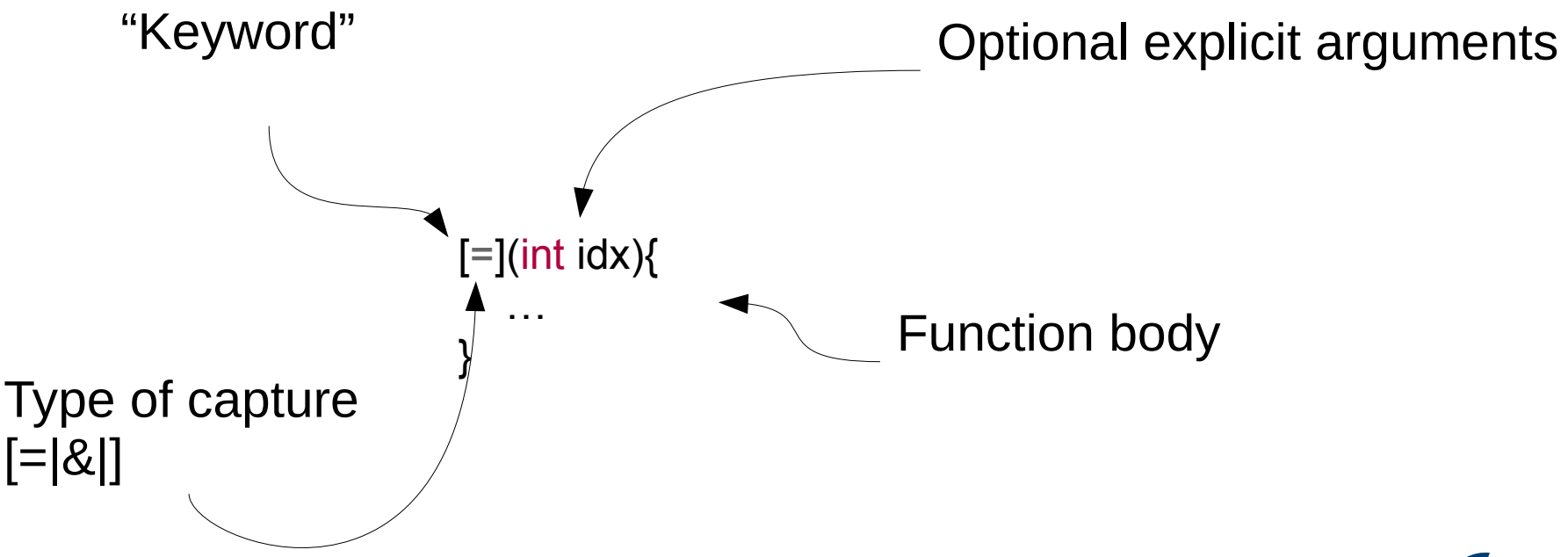
```
std::vector<int> v {5, 1, 1, 3, 1, 4, 1, 3, 3, 2};
int a = 3;
int b = 6;
auto lambda = [&](const int x){return (x >= a && x < b);}
auto ct36 = std::count_if(v.begin(), v.end(), lambda);
```

LAMBDAS

C++17

Fortran 2008

Lambdas are anonymous functions that can capture variables.



STD::TRANSFORM + LAMBDAS

C++17

Fortran 2008

```
#include <algorithm>
#include <execution>
#include <vector>

Template <class T>
void scale_vector(std::vector<T> &&x, std::vector<T> &&y, T a) {
    std::transform(x.begin(), x.end(), y, [=](auto x) {
      return  a * x;});
}
```

COUNTING

C++17

Fortran 2008

Sometimes it’s easier to use an index:

- Container of indices

```
std::vector idx(x.size(), 0);
std::iota(idx.begin(), idx.end(), 0);
std::for_each(idx.begin(), idx.end(), …)
```

- Counting iterator (for example from thrust)

```
auto r = thrust::counting_iterator<int>(0);
std::for_each(r, r+ N,...)
```

CATCHING POINTERS BY VALUE

C++17

Fortran 2008

Access to CPU memory not allocated with new → memory access error

- Reference capture of scalars → use value capture instead

Value capture of vector can also lead to problems → use pointer instead

```
auto ptr_x = x.data();
```

TRANSFORM_REDUCE

C++17

Fortran 2008

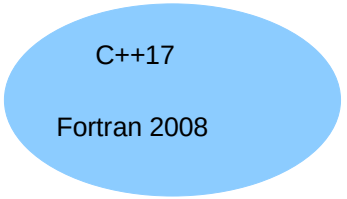
Transformation (map) and reduction (reduce) are often combined.

C++ offers transform_reduce to do it in one call:

```
std::transform_reduce(x.begin(), x.end(), y.begin(),
                      -1.0, [](auto a, auto b){return std::max(a, b);},
                      [](auto a, auto b){ return std::abs(a - b);}
                      );
```

First comes the **reduction** operation, **then** comes the **transform** operation.

ISO FORTRAN




```
subroutine vecAdd(a,b)
implicit none

real :: a(:)
real :: b
integer :: i, n

n = size(a)
do concurrent (i = 1: n)
  a(i)=a(i)+b
enddo

end subroutine vecAdd
```

- Only available with nvfortran
- Add option -stdpar
- Data transfer managed by runtime

ISO FORTRAN

C++17

Fortran 2008

```
do i = 1, n
   do j =1,m
      C(i,j)=a(i)+b(j)
   enddo
enddo
```

```
do concurrent (i = 1: n, j=1: m)
   C(i,j)=a(i)+b(j)
enddo
```

do concurrent (…) [locality spec]

Locality specification:

- local(list)
- local_init(list)
- share(list)

Specify locality of variables to ensure thread safety

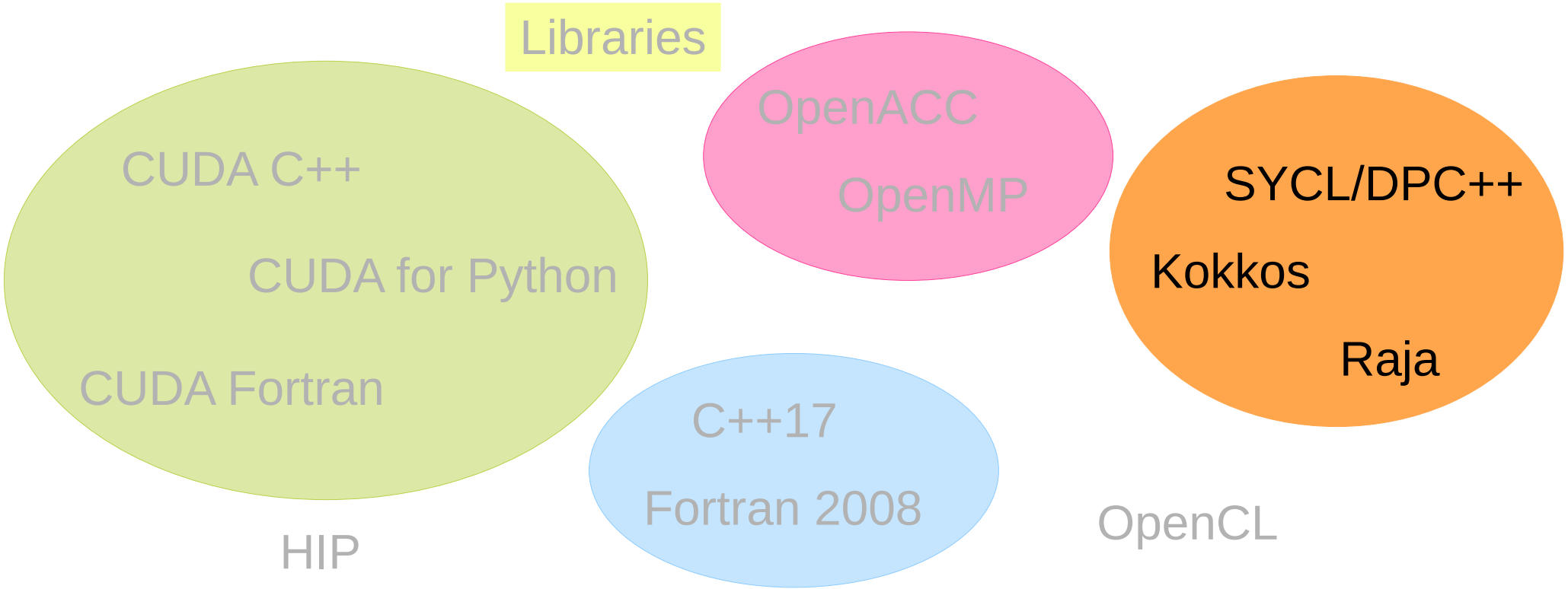
C++17

Fortran 2008

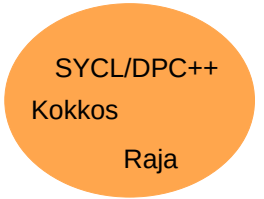
Exercises

exercises/02-iso-standards

C++-APIS


Libraries
CUDA C++
CUDA for Python
CUDA Fortran
HIP
OpenACC
OpenMP
C++17
Fortran 2008
SYCL/DPC++
Kokkos
Raja
OpenCL

PORTABLE C++-APIS

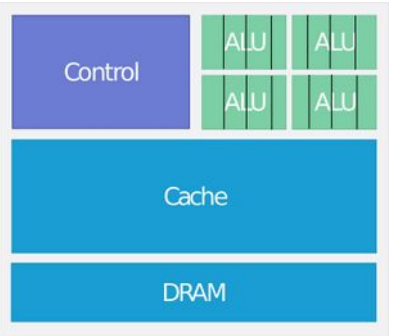


- CUDA is limited to Nvidia GPUs
- OpenCL too verbose and not performance portable
- Thrust provides algorithms similar to pSTL
- Kokkos and Raja provide performance portable frameworks (CPU, GPU)
- SYCL provides a single source C++ API for heterogeneous platforms
- DPC++ (data parallel C++) is an Intel-supported compiler for SYCL with extensions

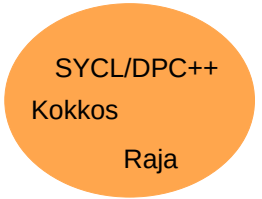
CHALLENGES OF PERFORMANCE PORTABILITY CPU-GPU

SYCL/DPC++
Kokkos
Raja

- Memory spaces
- SIMD vs. SIMT



KOKKOS



- Memory spaces → multidimensional views

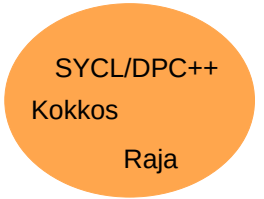
```
// 3D view of doubles in default space
View<double***> data("data", O, N, M);
// 2D view in CudaUVMSpace with last
// dimension set to size M at compile time
View<double*[M], CudaUVMSpace> a("a", N);
```

- SIMD vs. SIMT → execution spaces

https://github.com/kokkos

```
double sum = 0;
Kokkos :: parallel _reduce( " Label " ,
   RangePolicy<Cuda>(0 , size ),
   KOKKOS_LAMBDA ( const int64_t index ,
      valueToUpdate += array ( index );
   },
sum );
```

SYCL/DPC++



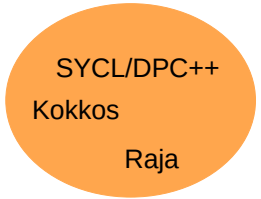
SYCL is a model to program accelerators

- Queue for each device
- Buffers for memory spaces

```
#include <CL/sycl.hpp>
int main(){
    sycl::queue Q; // default queue
    size_t N = 1024;
    float a = 3.1415;
    float* x = sycl::malloc_host<float>(N, Q);
    float* y = sycl::malloc_shared<float>(N, Q);
    …
    Q.parallel_for(N, [=](auto i){
        y[i] = a * x[i] + y[i];
    }).wait();
}
```

SYCL/DPC++

Buffer

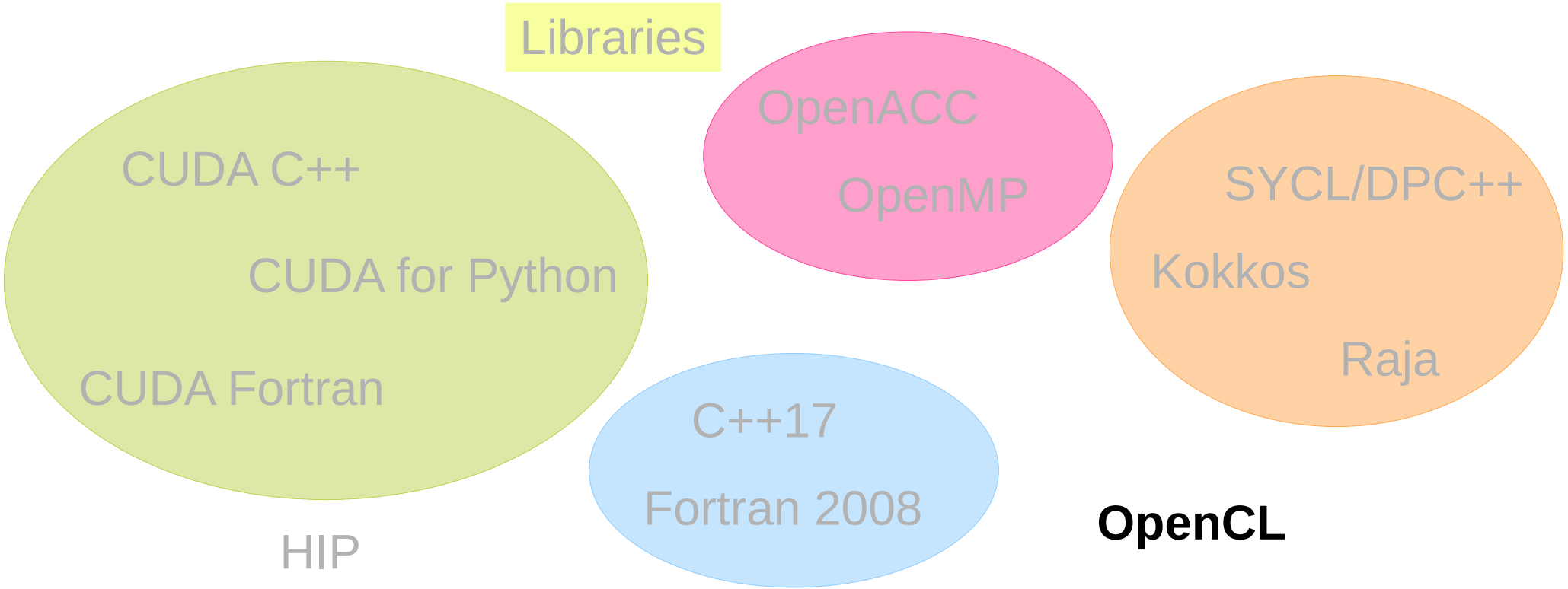


SYCL is a model to program accelerators

- Queue for each device
- Buffers for memory spaces

```
…
std::vector x(N, 0.0);
std::vector y(N, 1.0);
sycl::range r(x.size());
sycl::buffer<double, 1> x_buf(x.data(), r);
sycl::buffer<double, 1> y_buf(y.data(), r);
Q.submit([&](auto &h){
    sycl::accessor x(x_buf, h, sycl::read_only);
    sycl::accessor x(y_buf, h, sycl::read_write);
    h.parallel_for(N, [=](auto i){
        y[i] = a * x[i] + y[i];
    });
});
…
```

OPENCL


Libraries
CUDA C++
CUDA for Python
CUDA Fortran
HIP
OpenACC
OpenMP
C++17
Fortran 2008
SYCL/DPC++
Kokkos
Raja
OpenCL

OPENCL

- Industry standard maintained by Khronos
- Support for many different devices: CPU, GPU, FPGA, …
- Kernel can be build at runtime
- Considered too verbose by many, but mostly for “hello world”
- OpenCL 1.2 available on more platforms than any other model

OPENCL 1.2 KERNEL

```
kernel void saxpy(global float* y,
                  const global float* x,
                  float a, size_t N){
    int i = get_global_index(0);
    if (i < N) {
        y[i] = a * x[i] * y[i];
    }
}
```

- Kernel loaded as string
- Compiled at run time
- Syntax similar to CUDA kernels

OPENCL C++ API

Platform

```
std::vector<cl::Platform> platforms;
cl::Platform::get(&platforms);

// Get a list of all available GPU devices
//on the first platform
std::vector<cl::Device> devices;
platforms[0].getDevices(CL_DEVICE_TYPE_GPU, &devices);

// Create a context for the first device of the first platform and build the program
cl::Context context(devices[0]);
```

OPENCL C++ API

Compile kernel

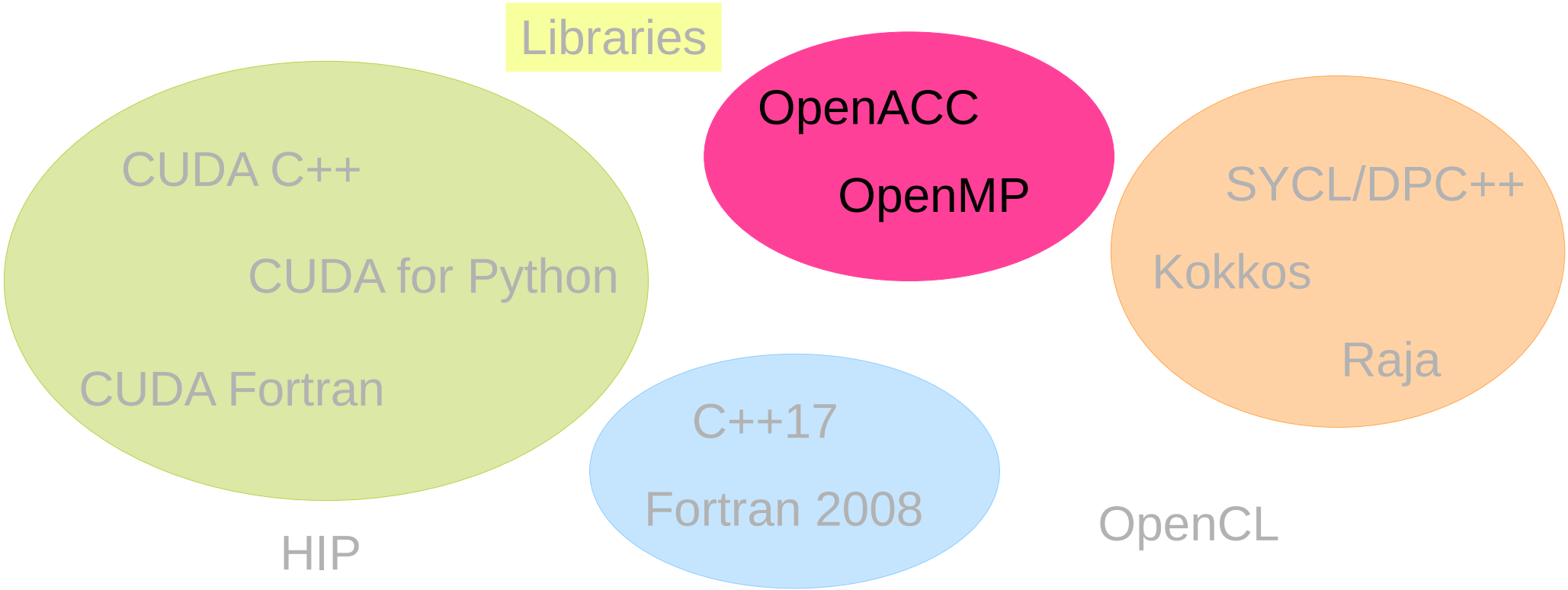
```
// Load the source code as a string
std::ifstream kernelSource("saxpy.cl");
std::string src(std::istreambuf_iterator<char>(kernelSource),
                (std::istreambuf_iterator<char>()));
cl::Program::Sources sources(1, std::make_pair(src.c_str(), src.length() + 1));

// Create a context for the first device of the first platform and build the program
cl::Program program(context, sources);
auto err = program.build();
```

OPENCL C++ API

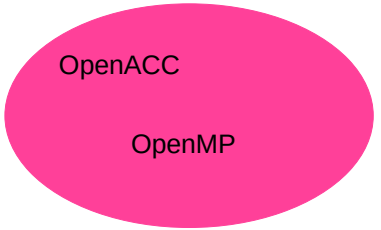
Call kernel

```
cl::CommandQueue queue(context);
// A program can contain multiple kernels. We pick the kernel by name.
cl::make_kernel<cl::Buffer, cl::Buffer, float, int> saxpy(program, "saxpy");

// If we wrote the program using the CUDA runtime API, the above steps would have been
// done automatically and the kernel would have been compiled explicitly.
cl::Buffer x_gpu(context, x.begin(), x.end(), true);
cl::Buffer y_gpu(context, y.begin(), y.end(), false);

saxpy(cl::EnqueueArgs(queue, cl::NDRange(N)), y_gpu, x_gpu, a, N);
cl::copy(queue, y_gpu, y.begin(), y.end());
```

PRAGMAS

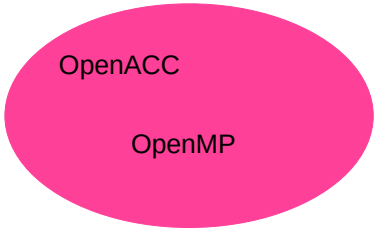

Libraries
CUDA C++
CUDA for Python
CUDA Fortran
HIP
OpenACC
OpenMP
C++17
Fortran 2008
SYCL/DPC++
Kokkos
Raja
OpenCL

PRAGMAS



- Annotate the code
- Ignored by compilers if unknown
- Available for C/C++ and Fortran
- Allow porting existing code to GPU incrementally

OPENACC

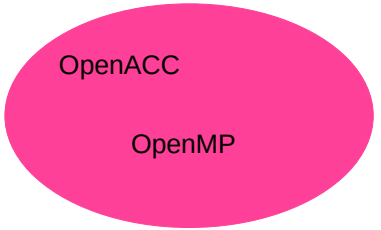



```
#pragma acc kernels
for(size_t i = 0; i < N; ++i){
    y[i] = a * x[i] + y[i];
}
```

```
!$acc kernels
do i = 1, N
    y[i] = a * x[i] + y[i]
end do
!$acc end kernels
```

- Good support on Nvidia GPUs
- Best support in Nvidia HPC SDK
- Support in GCC (also coming for AMD GPUs)
- Cray
- Clang → in development

OPENACC



acc kernels

```
#pragma acc kernels
{
  for(size_t i = 0; i < N; ++i){
     y[i] = a * x[i] + y[i];
  }

  sum = 0;
  for(size_t i = 0; i < N; ++i){
     sum += y[i];
  }
}
```

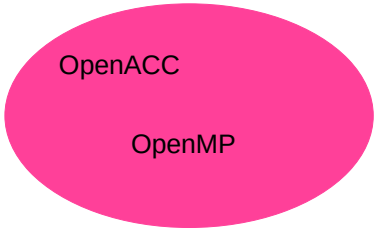
map kernel

reduction kernel

- Two equivalent ways to annotate kernels:
  - #pragma acc kernels
  - #pragma acc parallel loop

→ We'll start with kernels

OPENACC



acc parallel loop

```
#pragma acc parallel loop
for(size_t i = 0; i < N; ++i){
    y[i] = a * x[i] + y[i];
}
```

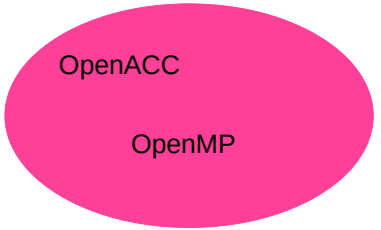
} map kernel

```
sum = 0;
#pragma acc parallel loop reduction(+:sum)
for(size_t i = 0; i < N; ++i){
     sum += y[i];
}
```

} reduction kernel

- acc parallel loop is more explicit
- Very much like OpenMP
- Reduction has to be given explicitely

OPENACC



Controlling data flow

```
#pragma acc data copy(y[0:N]) copyin(x[0:N])
{
  #pragma acc parallel loop
  for(size_t i = 0; i < N; ++i){
     y[i] = a * x[i] + y[i];
  }

  sum = 0;
  #pragma acc parallel loop reduction(+:sum)
  for(size_t i = 0; i < N; ++i){
       sum += y[i];
  }
}
```

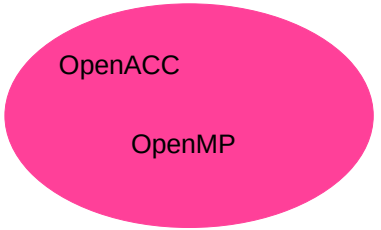
map kernel

reduction kernel

- Two kernel running on the GPU
- Data is copied back and forth!

  → control data flow
- Copy clauses can also be added to loop statement.
- acc data can also be used with kernels

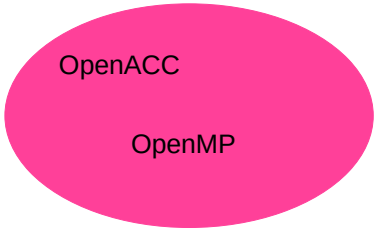
OPENMP




```
#pragma omp target parallel for
for(size_t i = 0; i < N; ++i){
    y[i] = a * x[i] + y[i];
}
```

```
!$omp target parallel do
do i = 1, N
    y[i] = a * x[i] + y[i]
end do
!$omp end target
```

- Support by Nvidia HPC SDK
- Support in GCC (also coming for AMD GPUs)
- AOMP (clang-based compiler by AMD)
- Clang → in development

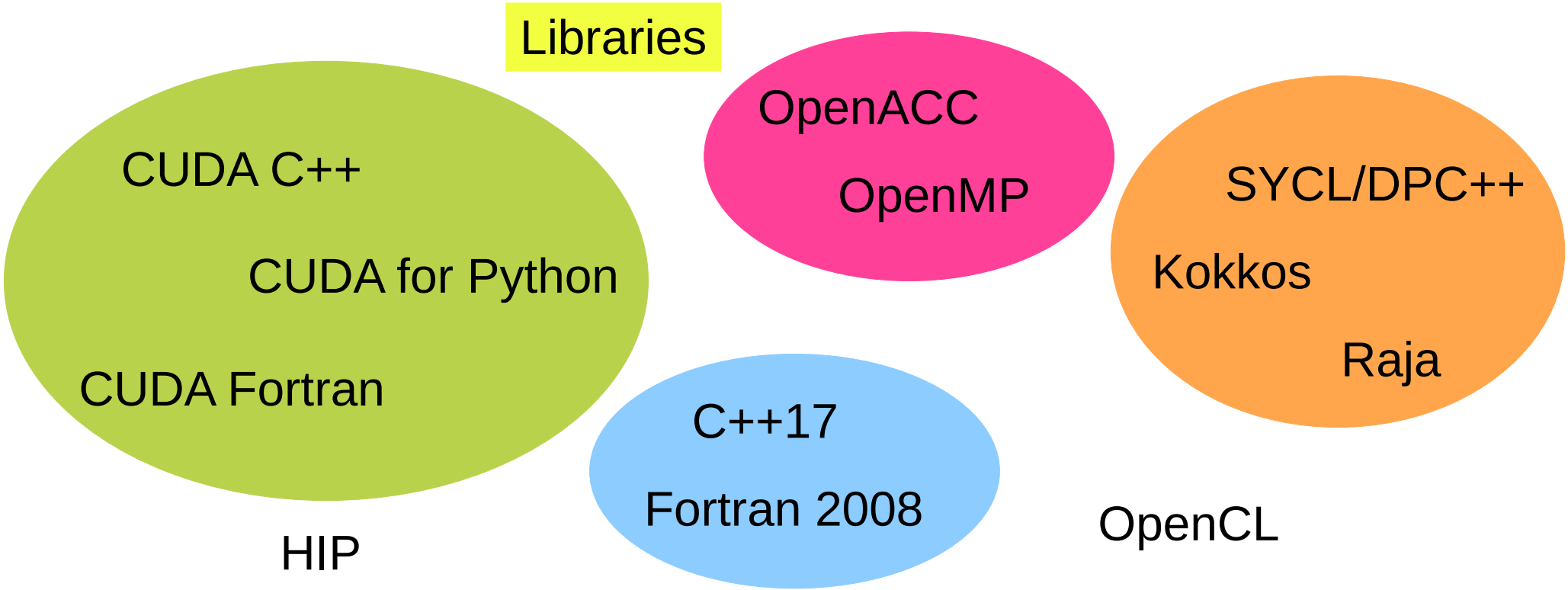


Exercise

exercises/03-pragmas

PROGRAMMING MODELS

Which and when



REFERENCES

- Accelerating Standard C++ with GPUs Using stdpar, https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar/
- Accelerating Fortran DO CONCURRENT with GPUs and the NVIDIA HPC SDK, https://developer.nvidia.com/blog/accelerating-fortran-do-concurrent-with-gpus-and-the-nvidia-hpc-sdk/