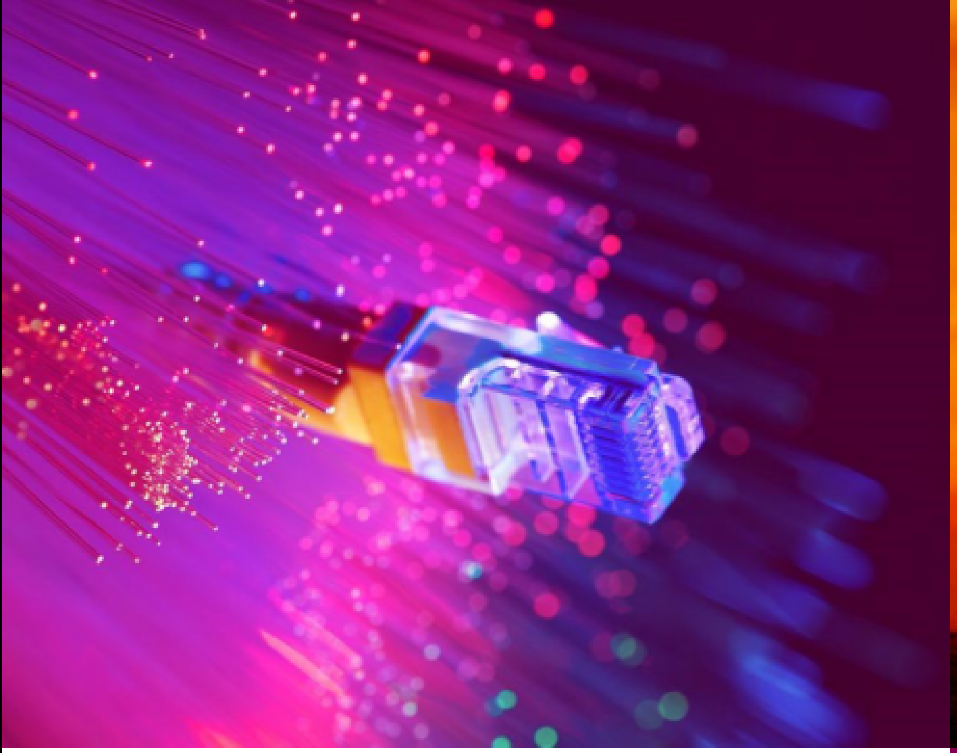

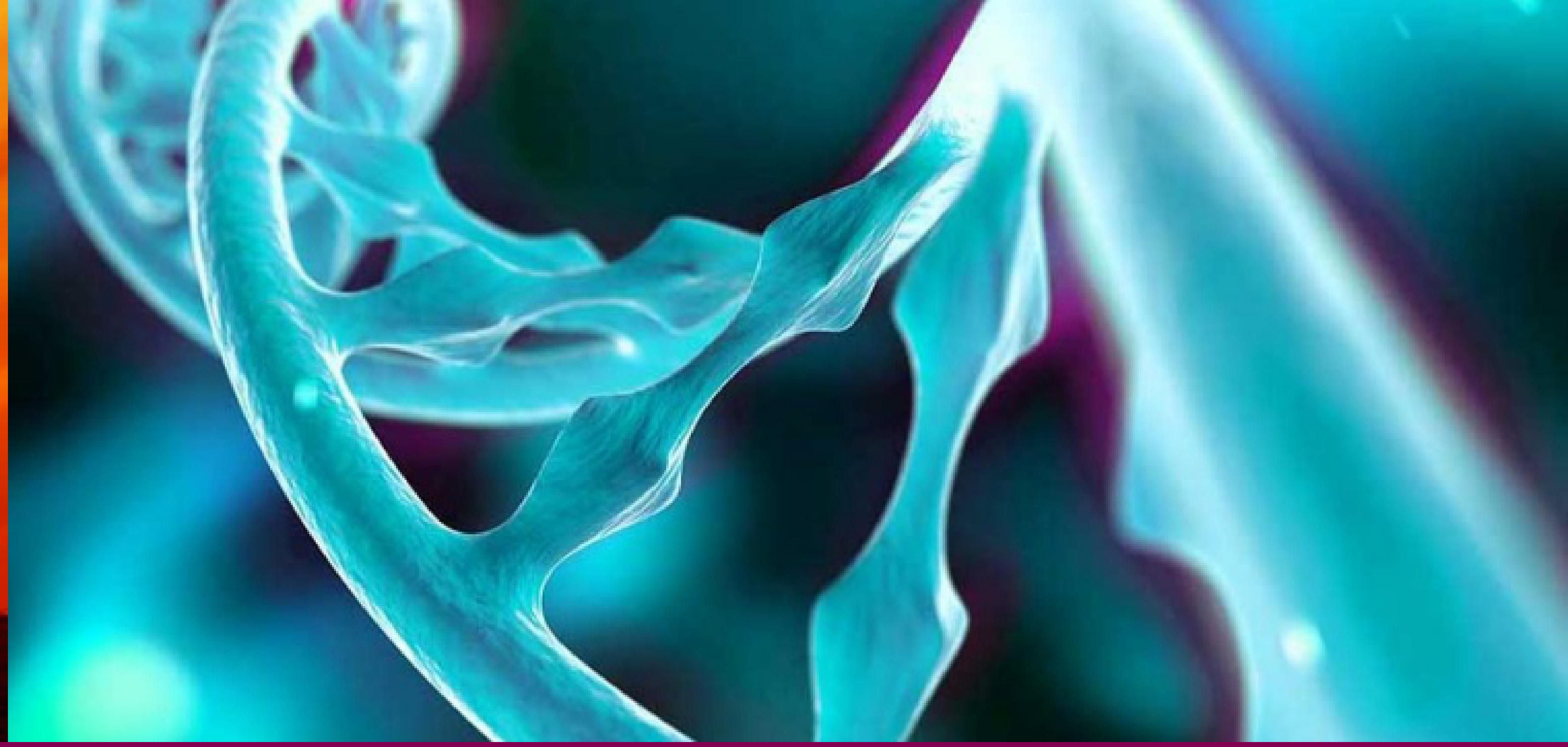

CSC

Kokkos C++ Performance Portability Ecosystem

High-Level GPU Programming

2024-02

CSC Training

CSC – Finnish expertise in ICT for research, education and public administration

Kokkos Ecosystem

- Kokkos is a C++ performance portability ecosystem developed primarily at Sandia National Laboratories since 2011
- It provides an abstraction layer for various parallel programming models like CUDA, HIP, SYCL, HPX, OpenMP, and C++ threads
- The ecosystem includes three main components, ie, Kokkos Core, Kokkos Kernels, and Kokkos Tools for GPU program development
- Kokkos (like SYCL) heavily utilizes modern C++ features like lambdas/functors and templates

Kokkos Core component

- Kokkos Core is a programming model for parallel algorithms on shared memory many-core architectures
- The model provides abstractions, such as execution spaces, patterns and policies, as well as memory spaces, layouts and traits
- The developer implements the algorithms using these abstractions which allows Kokkos to map and optimize the code for the desired target architectures
- Kokkos Core offers also some architecture-specific features for further optimization, but this breaks the portability of the code

Kokkos Kernels component (not covered in more detail)

- Kokkos Kernels is a software library featuring linear algebra and graph algorithms for optimal performance across various architectures
- The library is written using the Kokkos Core programming model for portability and good performance
- It includes architecture-specific optimizations and vendor-specific versions of mathematical algorithms
- Kokkos Kernels library reduces the need to develop architecture-specific software, lowering the modification cost for achieving good performance

Kokkos Tools component (not covered in more detail)

- Kokkos Tools is a plug-in software interface with a set of performance measurement and debugging tools for analyzing software execution and memory performance
- It relies on the Kokkos Core programming model interface and uses the user provided labels to identify data structures and computations
- A developer can use these tools for performance profiling and debugging to evaluate their algorithmic design and implementation, and to identify areas for improvement

Kokkos Compilation

- Usage of cross-platform portability libraries could require module/package maintainers to compile and offer multiple instances if different projects on the same system require different compilation settings (when used as an installed package)
- For instance, with Kokkos, one project might prefer CUDA as the default execution space, while another requires a CPU
- In addition to package install, Kokkos supports inline building of the Kokkos library with the user project, by specifying Kokkos compilation settings and including the Kokkos Makefile in the user Makefile
- Kokkos docs: https://kokkos.github.io/kokkos-core-wiki/building.html

Inline build: Hello Makefile example

```
default: build

# Set compiler
KOKKOS_PATH = $(shell pwd)/kokkos
CXX = hipcc

# Variables for the Makefile.kokkos
KOKKOS_DEVICES = "HIP"
KOKKOS_ARCH = "VEGA90A"

# Include Makefile.kokkos
include $(KOKKOS_PATH)/Makefile.kokkos

build: $(KOKKOS_LINK_DEPENDS) $(KOKKOS_CPP_DEPENDS) hello.cpp
 $(CXX) $(KOKKOS_CPPFLAGS) $(KOKKOS_CXXFLAGS) $(KOKKOS_LDFLAGS) hello.cpp $(KOKKOS_LIBS) -o hello
```

- To build a hello.cpp project with the above Makefile, no steps other than cloning the Kokkos project into the current directory is required

- Kokkos docs: https://kokkos.org/kokkos-core-wiki/ProgrammingGuide/Compiling.html#using-kokkos-gnu-makefile-system

Kokkos programming

- Kokkos code starts with Kokkos initialization and ends with finalization,

```
Kokkos::initialize(int& argc, char* argv[]);
.
.
Kokkos::finalize();
```

- Optional initialization parameters can be passed as follows:

```
Kokkos::initialize(Kokkos::InitializationSettings()
                   .set_device_id(0)                /* select the device (eg, 0th gpu of the total of 4 gpus) */
                   .set_disable_warnings(false)     /* disable warning messages */
                   .set_num_threads(1)              /* set the number of threads */
                   .set_print_configuration(true)); /* print the configuration after initialization */
```

- Kokkos docs: https://kokkos.github.io/kokkos-core-wiki/API/core/Initialize-and-Finalize.html

Kokkos programming - Execution and Memory Spaces

- Kokkos uses an execution space model to abstract the details of parallel hardware
- The execution space instances map to the available backend options such as CUDA, HIP, OpenMP, or SYCL
- Similarly, Kokkos uses a memory space model for different types of memory, such as host memory or device memory
- If the execution space or memory space are not explicitly chosen by the programmer in the source code, the default spaces are used (chosen during compile time)

Kokkos programmin - hello example

- The following is a full example of a Kokkos program that initializes Kokkos and prints the execution space and memory space instances

```
#include <Kokkos_Core.hpp>
#include <iostream>

int main(int argc, char* argv[]) {
  Kokkos::initialize(argc, argv);
  std::cout << "Execution Space: " <<
    typeid(Kokkos::DefaultExecutionSpace).name() << std::endl;
  std::cout << "Memory Space: " <<
    typeid(Kokkos::DefaultExecutionSpace::memory_space).name() << std::endl;
  Kokkos::finalize();
  return 0;
}
```

- The Kokkos API is accessed through `Kokkos_Core.hpp` header file

Kokkos memory management (malloc-based)

- Kokkos supports using raw pointers as well as buffers (Kokkos Views)
- With raw pointers, one can simply allocate and deallocate memory by

```
int* ptr = (int*) Kokkos::kokkos_malloc<Kokkos::SharedSpace>(n * sizeof(int));
...
Kokkos::kokkos_free<Kokkos::SharedSpace>(ptr);
```

where `Kokkos::SharedSpace` maps to any potentially available memory of “Unified Shared Memory” type, ie,

- Cuda -> CudaUVMSpace
- HIP -> HIPManagedSpace
- SYCL -> SYCLSharedUSMSpace
- Backends running on host -> HostSpace

- Kokkos docs: https://kokkos.org/kokkos-core-wiki/API/core/c_style_memory_management.html

Kokkos memory management (View-based)

- For Kokkos Views, an optimal data layout is determined at compile time depending on the computer architecture
- A 1-dimensional `View` of type `int*` into default and host memory spaces can be created by

```
Kokkos::View<int*>  dev_array("dev_array", n); // "dev_array" is a label, and n is the size of the allocation in ints
.
.
Kokkos::View<int*, Kokkos::HostSpace>  host_array("host_array", n); // same as above, but allocates to host space
```

- Memory copies between host and device spaces are handled explicitly:

```
Kokkos::deep_copy(dev_array, host_array); // a copy from host to device
```

- Kokkos docs: https://kokkos.github.io/kokkos-core-wiki/API/core/View.html

Kokkos parallel execution 1

- Kokkos provides three different parallel operations: `parallel_for`, `parallel_reduce`, and `parallel_scan`
    - The `parallel_for` operation is used to execute a loop in parallel
    - The `parallel_reduce` operation is used to execute a loop in parallel and reduce the results to a single value
    - The `parallel_scan` operation implements a prefix scan
- The following executes a simple for loop with `i` ranging from `0` to `n-1`:

```
Kokkos::parallel_for(n, KOKKOS_LAMBDA(const int i) {
  c[i] = a[i] * b[i];
});
```

- Kokkos docs: https://kokkos.github.io/kokkos-core-wiki/API/core/ParallelDispatch.html

Kokkos parallel execution 2

- The following executes a simple reduction loop with `i` ranging from `0` to `n-1` where `lsum` is a local sum variable and `sum` is the final global sum variable (`sum` need not be accessible from the device):

```
Kokkos::parallel_reduce(n, KOKKOS_LAMBDA(const int i, int &lsum) {
  lsum += i;
}, sum);
```

- Sum reduction is the default reduction operation, and if other reduction operations are desired, this must be indicated in the `parallel_reduce` call
- Kernel launches are asynchronous with host, use `Kokkos::fence()` to synchronize device and host execution

- Kokkos docs: https://kokkos.org/kokkos-core-wiki/API/core/parallel-dispatch/parallel_reduce.html#

Run Kokkos in simple steps

1. Create a folder with source file and Makefile, eg, `hello.cpp` and `Makefile`
2. Execute `git clone https://github.com/kokkos/kokkos.git` (in the same folder if using the Makefile shown at earlier page)
3. Run `make`
4. Run executable with, eg, `./hello` or `srun ./hello` with appropriate args

- **With inline build strategy, no separate step to manually compile and link Kokkos is required!**

Summary

- Kokkos is a portable GPU programming ecosystem supporting CUDA, HIP, SYCL, HPX, OpenMP, and C++ threads
- The ecosystem includes three main components, ie, Kokkos Core, Kokkos Kernels, and Kokkos Tools for GPU program development
- Kokkos (like SYCL) utilizes modern C++ features like lambdas/functors and templates for loop construction and memory management
- Kokkos is not a very popular choice for parallel programming, and therefore, learning and using Kokkos can be more difficult compared to more established programming models such as CUDA/HIP or OpenMP
- See Kokkos docs for more: https://kokkos.github.io/kokkos-core-wiki/index.html