GPU Programming (2)

M. Sato

RIKEN R-CCS
and University of Tsukuba

Contents

- Cuda (NVIDIA)
- HIP(AMD)
- OpenCL
- SYCL
- OneAPI (Intel ?)

- OpenACC
- OpenMP

- C++ template programming
  - Kokkos
  - Raja

Supercomputers in US around 2021-2023

- **Frontier (ORNL, 2021), El Capitan (LLNL/SNL, 2023?)**
  - **AMD CPU + AMD GPU**
- **Aurora (ANL, 2022)**
  - **Intel CPU + Intel GPU (Xe)**
- **NERSC-9 Perlmutter (LBNL/NERSC)**
  - **AMD CPU + NVIDIA GPU**

- **All US supercomputers have (different!) GPUs! Targeting to ExaFLOPS**

Upcoming Generation: Programming Models OpenMP 5, CUDA, HIP and DPC++ depending on machine

NERSC Perlmutter
AMD CPU / NVIDIA GPU
CUDA / *OpenMP 5* [c]

ORNL Frontier
AMD CPU / AMD GPU
HIP / *OpenMP 5* [d]

ANL Aurora
Xeon CPUs / Intel GPUs
DPC++ / *OpenMP 5* [e]

LLNL El Capitan
AMD CPU / AMD GPU
HIP / *OpenMP 5* [d]

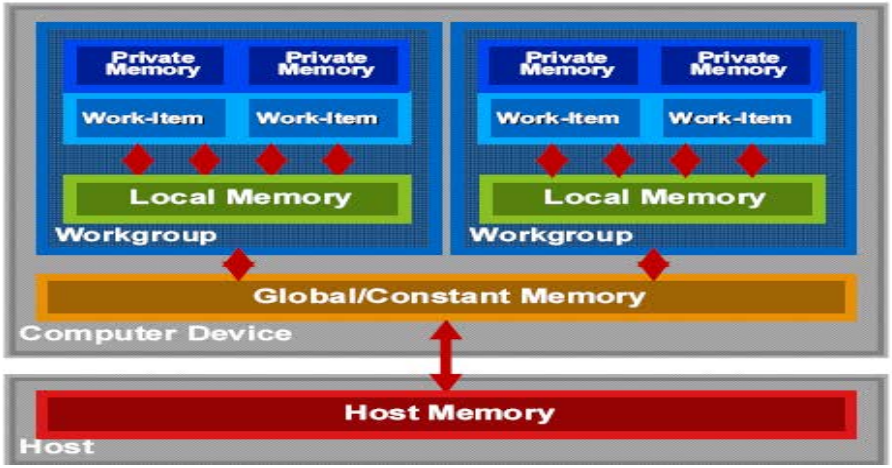
OpenCL

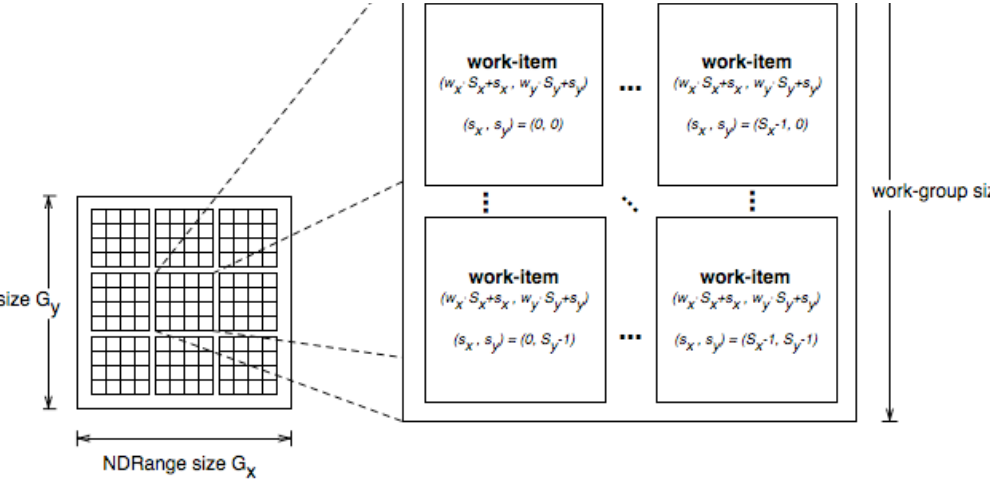
- **Programming language for general purpose GPU computing.**
- **While C for CUDA is proprietary by NVIDIA, OpenCL is targeting cross-platform environments.**
  - **Only only for GPU such as NVIDIA and AMD(ATI), but also for conventional multicore CPU and many-core, such as Cell Broadband Engine(Cell B.E) and Intel MIC**
- **The point is that it targets for data parallel program by GPU and also for task-parallel of multi-core.**
- **What is different from CUDA? : Similar programming mode for kernel, but different in execution environment.**
  - **OpenCL is supported to other GPU such as AMD and Intel**

Kernel and Memory model

OpenCL Memory Model

- **Private Memory**
  - Per work-item
- **Local Memory**
  - Shared within a workgroup (16Kb)
- **Local Global/Constant Memory**
  - Not synchronized
- **Host Memory**
  - On the CPU



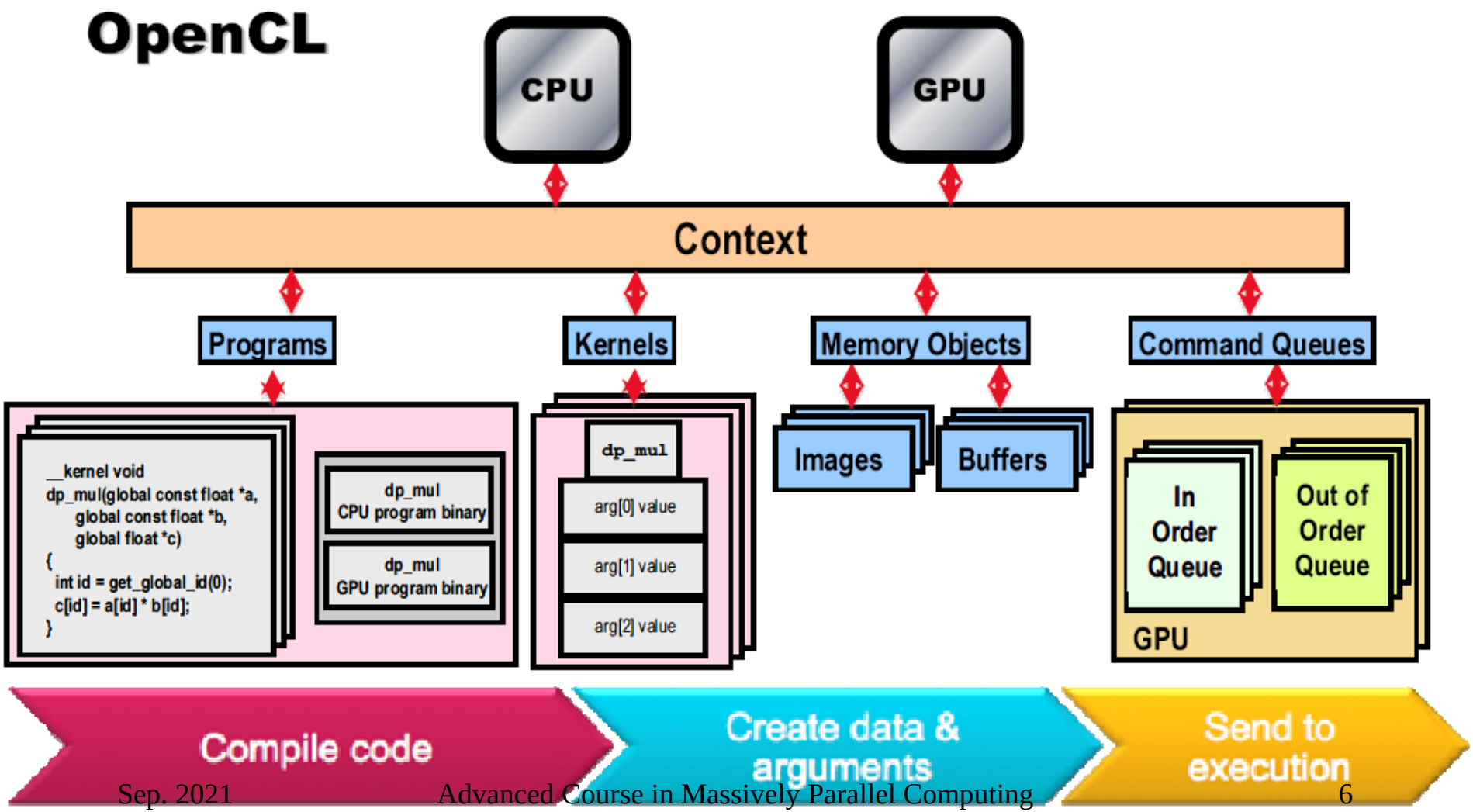


Data Parallel

```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *result)
{
  int id = get_global_id(0);

  result[id] = a[id] * b[id];
}
// execute dp_mul over "n" work-items
```

Execution Environment of OpenCL



Example

- **Saxpy kernel**

```
__kernel void saxpy(
        __global float* restrict arrayX,
        __global float* restrict arrayY,
        const float a
        )
{
        unsigned int i = get_global_id(0);
        arrayY[i] += arrayX[i]*a;
}
```

```
int main(int argc, char **argv)
{
        cl_device_id devId;
        cl_context context = NULL;
        cl_command_queue cmdQueue = NULL;
        cl_program prog = NULL;
        cl_kernel kern0 = NULL;
        cl_platform_id platformId =  NULL;
        cl_uint numDevices;
        cl_uint numPlatforms;
        cl_int ret;
        cl_event event0;
        cl_mem clxArray, clyArray;
        size_t globalWorkSize[3] = {1};
        size_t localWorkSize[3]  = {1};
        const char options[] = "";
        cl_int binStat;
        FILE *fp;
        char filename[] = "./saxpy.aocx";
        struct stat st;
        unsigned char *programBin = NULL;
        size_t programBinLength;
        unsigned int size;
        float a;
        double start, end, elapsed;
        float *xArray;
        float *yArray;
        ….
```

Declaration of OpenCL context and Related variable

```
ret = clGetPlatformIDs(1, &platformId, &numPlatforms);
if (ret != CL_SUCCESS) {
        …
        exit(-1);
}

ret = clGetDeviceIDs(platformId, CL_DEVICE_TYPE_ALL, 1, &devId, &numDevices);
if (ret != CL_SUCCESS) {
        ….
        exit(-1);
}

context = clCreateContext(NULL, 1, &devId, NULL, NULL, &ret);
cmdQueue = clCreateCommandQueue(context, devId, 0, &ret);

clxArray = clCreateBuffer(context, CL_MEM_READ_WRITE |
                  CL_MEM_COPY_HOST_PTR, sizeof(float)*size, xArray, &ret);
clyArray = clCreateBuffer(context, CL_MEM_READ_WRITE |
              CL_MEM_COPY_HOST_PTR, sizeof(float)*size, yArray, &ret);
prog = clCreateProgramWithBinary(context, 1, &devId,
     &programBinLength, (const unsigned char **)&programBin, &binStat, &ret)
if (ret != CL_SUCCESS) {
        fprintf(stderr, "[%d] ", ret);
        fprintf(stderr, "[error] clCreateProgramWithBinary¥n");
        exit(-1);
}

clBuildProgram(prog, 1, &devId, options, NULL, NULL);
```

Platform

Device

Command Que

Context

Allocation GPU memory

Program (kernel) Registrat And Build

```
// Create CL Kernel
kern0 = clCreateKernel(prog, "saxpy", &ret);
// set kernel args
ret = clSetKernelArg(kern0, 0, sizeof(cl_mem), &clxArray);
ret = clSetKernelArg(kern0, 1, sizeof(cl_mem), &clyArray);
ret = clSetKernelArg(kern0, 2, sizeof(float), &a);

//enqueuetask
/*
ret = clEnqueueTask(cmdQueue,kern0,0,NULL, &event0);
*/
globalWorkSize[0] = size;
ret = clEnqueueNDRangeKernel(cmdQueue,kern0,
        1,                  // work dimention
        NULL,               // global work offset
        globalWorkSize,     // global work size
        localWorkSize,      // local work size
        0,                  // num of depending events
        NULL,               // pointer to depending event list
        &event0             // event
        );
if (ret != CL_SUCCESS) {
        …
        exit(-1);
}
clWaitForEvents(1, &event0);
```

Create Kernel

Set arguments

Enqueue with NDRange

Wait for completion

```
// get results
ret = clEnqueueReadBuffer(cmdQueue,clyArray,true,
        0,sizeof(float)*size,yArray,0,NULL,NULL);
if (ret != CL_SUCCESS) {
        …
        exit(-1);
}

clReleaseEvent(event0);
clFlush(cmdQueue);
clFinish(cmdQueue);
clReleaseKernel(kern0);
clReleaseCommandQueue(cmdQueue);
clReleaseContext(context);
free(programBin);

// … end …
```

Get results

Finalizing …

Many API calls are required to do the same kernel call in CUDA. Func <<< >>>> (,,,)

SYCL

- **One source code for host and GPUs**
- **SYCL offers simple abstractions to core OpenCL features.**
  - **Rather than just putting C++ classes on top of OpenCL objects, these abstractions have been designed with C++ and Object Oriented programming paradigms in mind.**
- **A great reduction over bare OpenCL C, and even the C++ wrappers!**
  - **Note also that the kernel is inlined with the code: The kernel is still valid C++ code, and we can still run it on the host if there is no device available or if we want to debug it.**

SYCL example

- https://www.codeplay.com/portal/sycl-tutorial-1-the-vector-addition

```
#include <sycl.hpp>

using namespace cl::sycl

#define TOL (0.001) // tolerance used in floating point comparisons
#define LENGTH (1024) // Length of vectors a, b and c

int main() {
        std::vector h_a(LENGTH); // a vector
        std::vector h_b(LENGTH); // b vector
        std::vector h_c(LENGTH); // c vector
        std::vector h_r(LENGTH, 0xdeadbeef); // d vector (result)
        // Fill vectors a and b with random float values
        int count = LENGTH;
        for (int i = 0; i < count; i++) {
                h_a[i] = rand() / (float)RAND_MAX;
                h_b[i] = rand() / (float)RAND_MAX;
                h_c[i] = rand() / (float)RAND_MAX;
        }
        {
        // Device buffers
        buffer d_a(h_a);
        buffer d_b(h_b);
        buffer d_c(h_c);
        buffer d_r(h_d);
        queue myQueue;
        command_group(myQueue, [&]()
        {
        // Data accessors
        auto a = d_a.get_access<access::read>();
```

```
#include <sycl.hpp>

using namespace cl::sycl

#define TOL (0.001) //
#define LENGTH (1024) /

int main() {
        std::vector h_
        std::vector h_
        std::vector h_
        std::vector h_
        // Fill vectors a and b
        int count = LENGTH;
        for (int i = 0; i <
                h_a[i] = ran
                h_b[i] = ran
                h_c[i] = ran
        }
        {
        // Device buffers
        buffer d_a(h_a);
        buffer d_b(h_b);
        buffer d_c(h_c);
        buffer d_r(h_d);
        queue myQueue;
        command_group(myQueue, [&]()
        {
        // Data accessors
        auto a = d_a.get_access<access::read>();
```

The fi
inclus
templa
with t
and ob
the *cl*

Data shared between host and device is defined using the SYCL buffer class .

- **The class provides different constructors to initialize the data from various sources. In this case, we use a constructor from STL Vectors, which transfers data ownership to the**

The next thing we need is a queue to enqueue our kernels.

In OpenCL we will need to set up all the other related classes on our own; but using SYCL we can use the default constructor of the queue class to automatically target the first OpenCL-enabled device available.

```
command_group(myQueue, [&]()
{
// Data accessors
auto a = d_a.get_access<access::read>();
auto b = d_b.get_access<access::read>();
auto c = d_c.get_access<access::read>();
auto r = d_r.get_access<access::write>();
// Kernel
parallel_for(count, kernel_functor([ = ](id<> item) {
        int i = item.get_global(0);
        r[i] = a[i] + b[i] + c[i];
        }));
});
}
// Test the results
int correct = 0;
float tmp;
for (int i = 0; i < count; i++) {
        tmp = h_a[i] + h_b[i] + h_c[i]; // assign element i of a+b+c to tmp
        tmp -= h_r[i]; // compute deviation of expected and output result
        if (tmp * tmp < TOL * TOL) // correct if square deviation is less th
        // tolerance squared
        {
                correct++;
        } else {
                printf(" tmp %f h_a %f h_b %f h_c %f h_r %f ¥n", tmp, h_a[i
                h_c[i], h_r[i]);
        }
}
//
```

```
command_group(myQueue, [&]()
{
// Data accessors
auto a = d_a.get_access<acce
auto b = d_b.get_access<acce
auto c = d_c.get_access<acce
auto r = d_r.get_access<acce
// Kernel
parallel_for(count, kernel_f
        int i = item.get_gl
        r[i] = a[i] + b[i]
        }));
});
}
// Test the results
int correct = 0;
float tmp;
for (int i
        t
        t
        i
        /
        {
                correct
        } else {
                printf(
                h_c[i],
        }
}
```

Once we have created the queue object, we can enqueue kernels. Together with the code itself, we need additional information to enqueue and run the kernel, such as the parameters and the dependencies that a certain kernel may have on other kernels. All that information is grouped in command group object .

The constru where we wa a lambda or the kernel accessors.

The accessor class characterizes the access of the kernel to the data it requires, i.e. if it is read, write, read/write, or many other access modes. Accessors are just templated objects that can be created from different types.
This allows the device compiler to generate more efficient code, and the runtime to schedule different command groups as efficiently as possible.

```
command_group(myQueue, [&]()
{
// Data accessors
auto a = d_a.get_access<access::read>();
auto b = d_b.get_access<access::read>();
auto c = d_c.get_access<access::read>();
auto r = d_r.get_access<access::write>();
// Kernel
parallel_for(count, kernel_functor([ = ](id<> item) {
        int i = item.get_global(0);
        r[i] = a[i] + b[i] + c[i];
        }));
});
}
// Test the re
int correct = 
float tmp;
for
```

The first parameter of the parallel for is the number of work-

The lambda used for parallel_for expects an id parameter, which is the class that represents the current work-item. It features methods to get detailed information from it, such as local or work group info or global work group info. In this case, the contents of the lambda represent what will be executed for each work-item.

To run t parallel three di the para execute of times is one o launch kernels i

use lements ameter d as a ce me this sample, a simple C++11 lambda.

```
}
```

Comment on sample code

- **The first thing to write in SYCL is the inclusion of the SYCL headers, providing the templates and class definitions to interact with the runtime library. All SYCL classes and objects are defined in the *cl::sycl* namespace**
- **Data shared between host and device is defined using the SYCL buffer class .**
  - **The class provides different constructors to initialize the data from various sources. In this case, we use a constructor from STL Vectors, which transfers data ownership to the SYCL runtime.**
  - **SYCL buffers do not require read/write information, as this is defined on a per-kernel basis via the accessor class.**
- **The next thing we need is a queue to enqueue our kernels.**
  - **In OpenCL we will need to set up all the other related classes on our own; but using SYCL we can use the default constructor of the queue class to automatically target the first OpenCL-enabled device available.**

Comment on sample code

- **Once we have created the queue object, we can enqueue kernels. Together with the code itself, we need additional information to enqueue and run the kernel, such as the parameters and the dependencies that a certain kernel may have on other kernels. All that information is grouped in command_group object .**
- **In this case we create an anonymous command group object.**
- **The constructor receives the queue where we want to run the kernel, and a lambda or functor which contains the kernel and the associated accessors.**
- **The accessor class characterizes the access of the kernel to the data it requires, i.e. if it is read, write, read/write, or many other access modes. Accessors are just templated objects that can be created from different types.**
- **This allows the device compiler to generate more efficient code, and the runtime to schedule different command groups as efficiently as possible.**

Comment on sample code

- **To run the vector addition in parallel for each element of the three different vectors, so we use the parallel_for statement to execute the kernel a certain number of times. The parallel_for statement is one of the different ways you can launch kernels in SYCL.**
- **The first parameter of the parallel_for is the number of work-items to use, in this case we use one work-item per number of elements in the vector. The second parameter is the kernel itself, provided as a kernel_functor instance. kernel_functor is a convenience class that enables creating the kernel instance from different sources, such as legacy OpenCL kernels or, as is the case in this sample, a simple C++11 lambda.**
- **The lambda used for parallel_for expects an id parameter, which is the class that represents the current work-item. It features methods to get detailed information from it, such as local or work group info or global work group info. In this case, the contents of the lambda represent what will be executed for each work-item.**

Comment on sample code

- **In this case the contents of the kernel are pretty much equal to the ones used in classic OpenCL, but we can access local scalar variables from the kernel without adding additional code.**
- **Also, we can call host functions and methods from inside the kernel, and we use templates and other fancy features inside.**
- **The host will wait so that the data can be copied back to the host when the ownership of the buffer is transferred at the end of the scope.**

Using factor

```
template<typename TYPE>
class vadd_params
{
private:
        buffer<TYPE, 1> * m_va;
        buffer<TYPE, 1> * m_vb;
        buffer<TYPE, 1> * m_vc;
        unsigned int m_nelems;
public:
        vadd_params( buffer<TYPE, 1> * va,
                buffer<TYPE, 1> * vb,
                buffer<TYPE, 1> * vc,
                unsigned int nelems
        ):
        m_va(va), m_vb(vb), m_vc(vc), m_nelems(nelems) { };
        void operator()()
        {
                auto ptrA = m_va->template get_access<access::read>();
                auto ptrB = m_vb->template get_access<access::read>();
                auto ptrC = m_vc->template get_access<access::read_write>();
                parallel_for(m_nelems, kernel_lambda<class vadd_params_kernel>
                ([=] (id<1> i) {
                        ptrC[i] = ptrA[i] + ptrB[i];
                }
        ));
        }
```

Us

The functor is instantiated for floats and passed to the constructor of the command group, which enqueues it on the given queue.

- **Although we only use floats in this sample (as we are following the tutorial), we could be using any type. The compiler will take care of creating the various implementations for us.**

```
template<typename TYPE>
class vadd_params
{
private:
        buffer<TYPE, 1> * m_va;
        buffer<TYPE, 1> * m_vb;
        buffer<TYPE, 1> * m_vc;
        unsigned int m_nelems;
public:
        vadd_params( buffer<TYPE,
                buffer<TYPE, 1> * vb,
                buffer<TYPE, 1> * vc,
                unsigned int nelems
        ):
        m_va(va), m_vb(vb), m_vc(vc), m_nelems(nelems) { };
        void operator()()
        {
                auto ptrA = m_va->template get_access<access::read>();
                auto ptrB = m_vb->template get_access<access::read>();
                auto ptrC = m_vc->template get_access<access::read_write>();
                parallel_for(m_nelems, kernel_lambda<class vadd_params_kernel>
                ([=] (id<1> i) {
                        ptrC[i] = ptrA[i] + ptrB[i];
                }
        ));
```

Main program

```
const unsigned NELEMS = 1024u;
(...)
{ /* A: Create scope */
  buffer<float, 1> bufA(h_A.data(), h_A.size());
  buffer<float, 1> bufB(h_B.data(), h_B.size());
  buffer<float, 1> bufC(h_C.data(), h_C.size());
  buffer<float, 1> bufD(h_D.data(), h_D.size());
  buffer<float, 1> bufE(h_E.data(), h_E.size());
  buffer<float, 1> bufF(h_F.data(), h_F.size());
  buffer<float, 1> bufG(h_G.data(), h_G.size());
  /* The default constructor will use a default selector */
  queue myQueue;
  /* Now we create the command group objects to enqueue the command group
   * objects with different parameters.
   */
  command_group(myQueue, vadd_params<float>(&bufA, &bufB, &bufC, NELEMS))
  command_group(myQueue, vadd_params<float> (&bufE, &bufC, &bufD, NELEMS)
  command_group(myQueue, vadd_params<float> (&bufG, &bufD, &bufF, NELEMS)
} /* B: Will wait until execution here */
(...)
```

Mai

```
const unsigned NELEMS = 1024u;
(...)
{ /* A: Create scope */
  buffer<float, 1> bufA(h_A.da
  buffer<float, 1> bufB(h_B.da
  buffer<float, 1> bufC(h_C.da
  buffer<float, 1> bufD(h_D.da
  buffer<float, 1> bufE(h_E.da
  buffer<float, 1> bufF(h_F.da
  buffer<float, 1> bufG(h_G.da
  /* The default constructor will use a default selector */
  queue myQueue;
  /* Now we create the command group objects to enqueue the command group
   * objects with different parameters.
   */
  command_group(myQueue, vadd_params<float>(&bufA, &bufB, &bufC, NELEMS))
  command_group(myQueue, vadd_params<float> (&bufE, &bufC, &bufD, NELEMS)
  command_group(myQueue, vadd_params<float> (&bufG, &bufD, &bufF, NELEMS)
} /* B: Will wait until execution here */
(...)
```

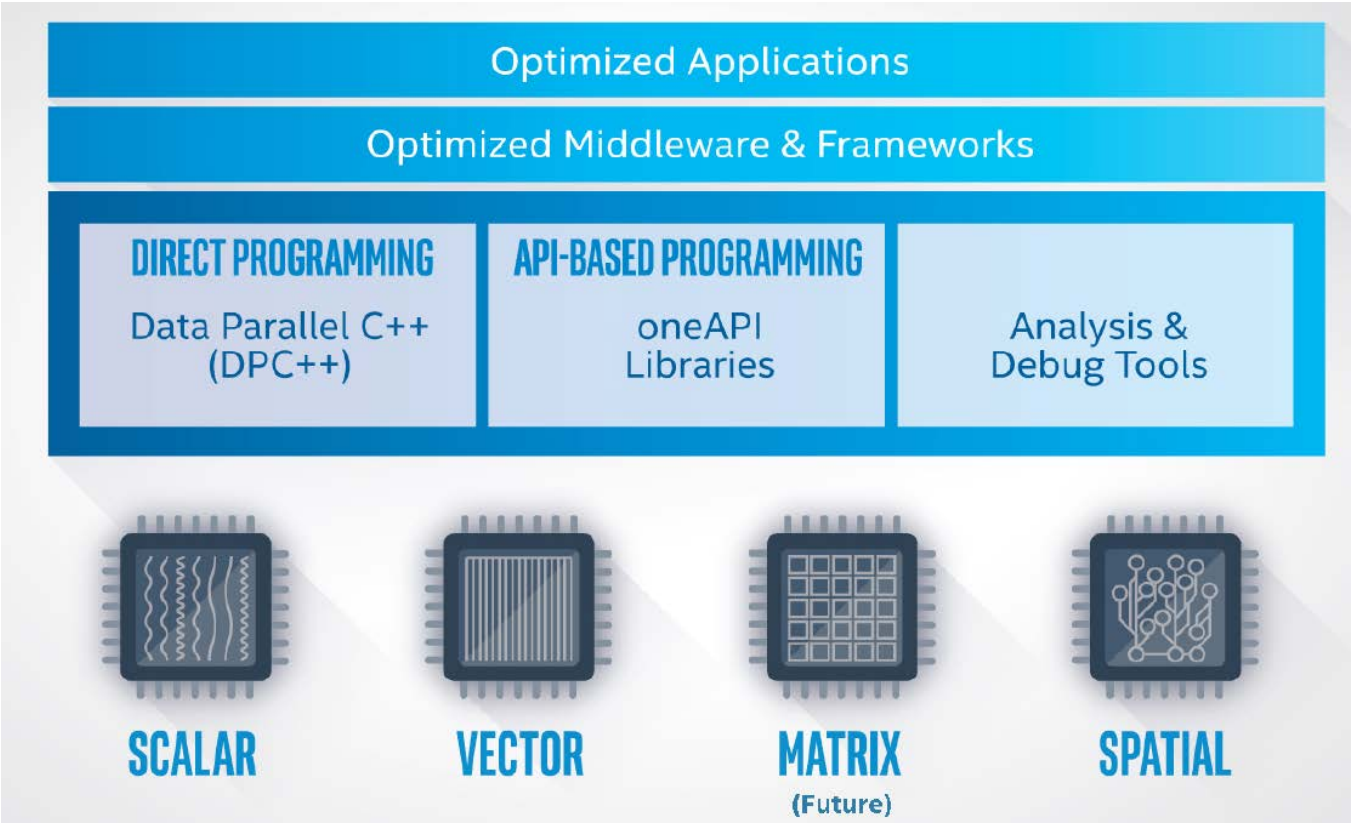
The three command groups will then be executed in order. When the execution reaches the end of the block statement at B, the destructor of the buffers will trigger the copying back of the result.

- **Note also that we are not copying data in and out for each kernel, and the runtime will take care of copying the data required for each kernel.**

- **The functor is instantiated for floats and passed to the constructor of the command group, which enqueues it on the given queue.**
  - **Although we only use floats in this sample (as we are following the tutorial), we could be using any type. The compiler will take care of creating the various implementations for us.**
- **The three command groups will then be executed in order. When the execution reaches the end of the block statement at B, the destructor of the buffers will trigger the copying back of the result.**
  - **Note also that we are not copying data in and out for each kernel, and the runtime will take care of copying the data required for each kernel.**

OneAPI

- **For Intel CPU, GPU, FPGA, and AI accelerators**



OneAPI

- **Data Parallel C++ Language for Direct Programming :** an evolution of C++ that incorporates SYCL*.
  - It allows code reuse across hardware targets and enables high productivity and performance across CPU, GPU, and FPGA architectures, while permitting accelerator-specific tuning.
- **Libraries for API-Based Programming**
  - including deep learning, math, and video processing—
  - include pre-optimized, domain-specific functions to accelerate compute-intense workloads on Intel® CPUs and GPUs
- **Advanced Analysis and Debug Tools**

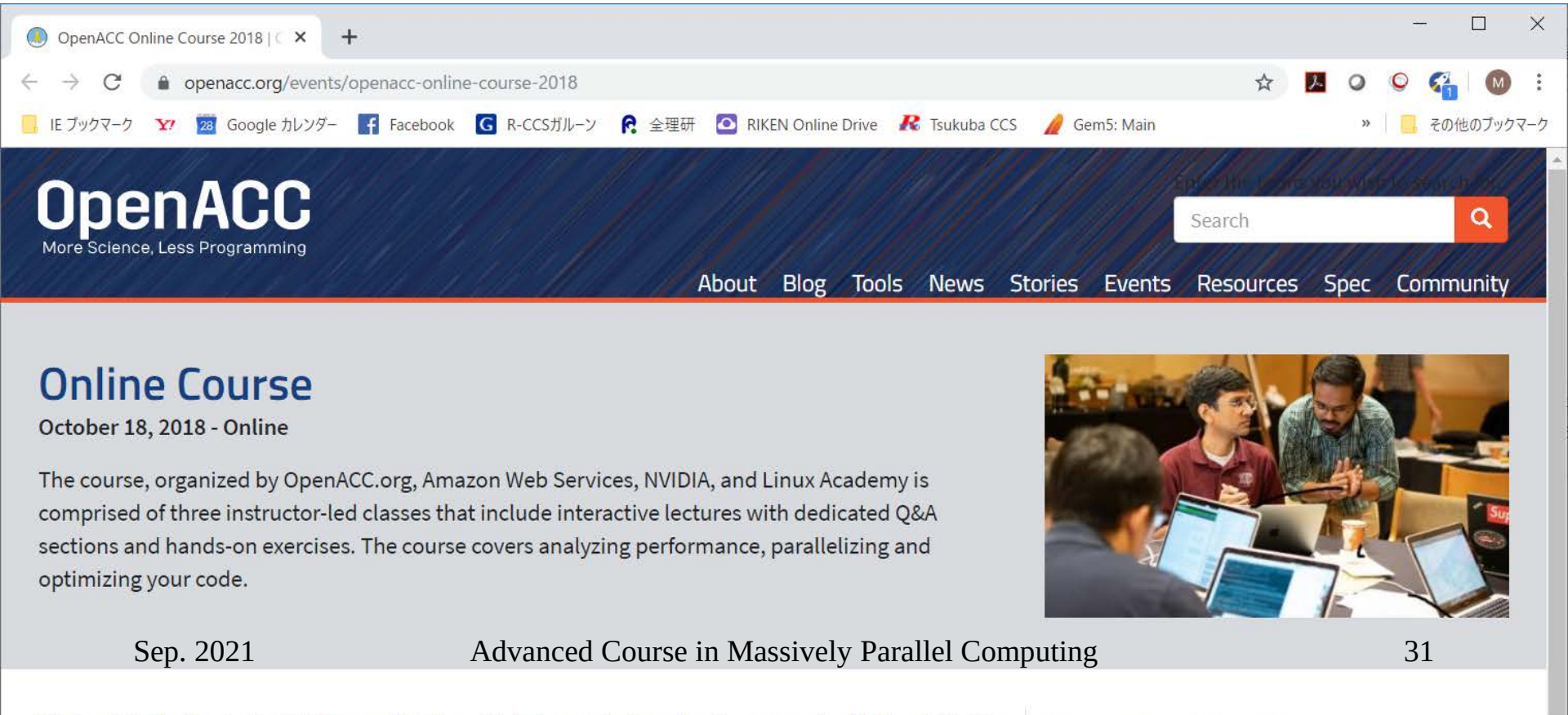
OpenACC

- A spin-off activity from OpenMP ARB for supporting accelerators such as GPGPU and MIC
- NVIDIA, Cray Inc., the Portland Group (PGI), and CAPS enterprise
- Directive to specify the code offloaded to GPU.

OpenACC®
DIRECTIVES FOR ACCELERATORS

OpenACC Online Course recording

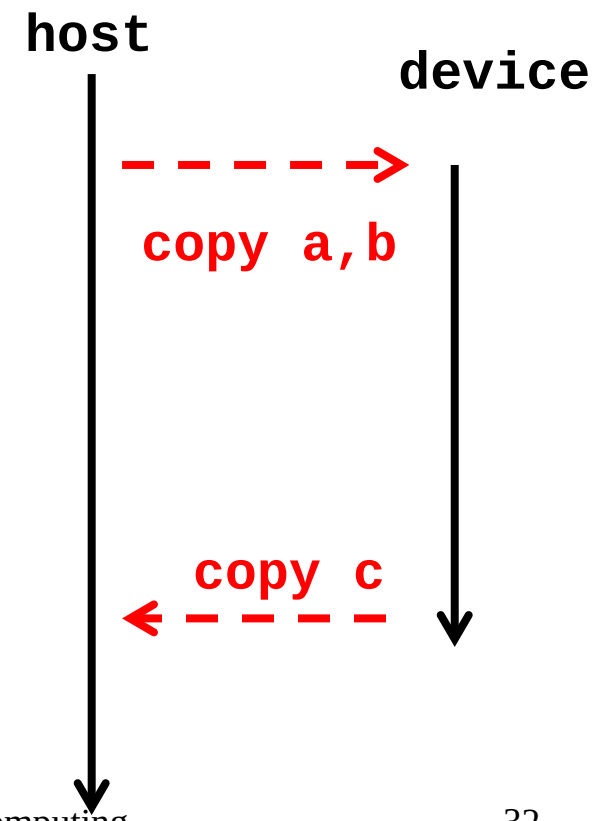
- https://www.openacc.org/events/openacc-online-course-2018



A simple example

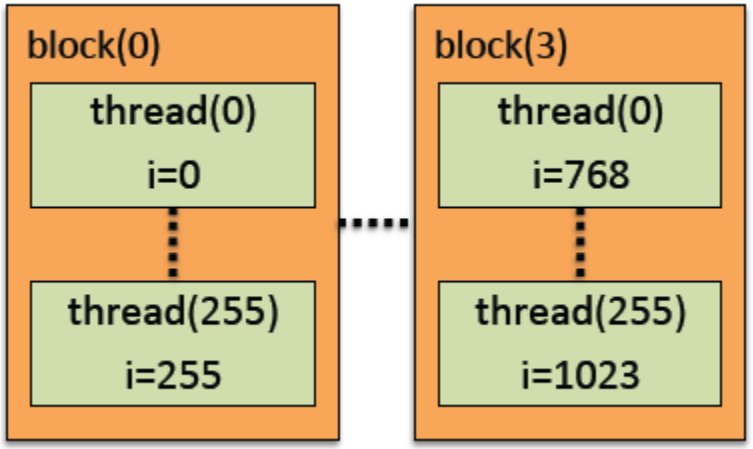
direction	copy	copyin	copyout
Host->device	○	○	
Device->Host	○		○

```
#define N 1024
int main(){
int i;
int a[N], b[N],c[N];
#pragma acc data copyin(a,b) copyout(c)
{
 #pragma acc parallel
 {
 #pragma acc loop
   for(i = 0; i < N; i++){
      c[i] = a[i] + b[i];
   }
 }
}
}
```



A simple example

```
#define N 1024
int main(){
int i;
int a[N], b[N],c[N];
#pragma acc data copyin(a,b) copyout(c)
{
 #pragma acc parallel
 {
 #pragma acc loop
   for(i = 0; i < N; i++){
      c[i] = a[i] + b[i];
   }
 }
}
}
```



execute iterations like CUDA kernel

Matrix Multiply in OpenACC

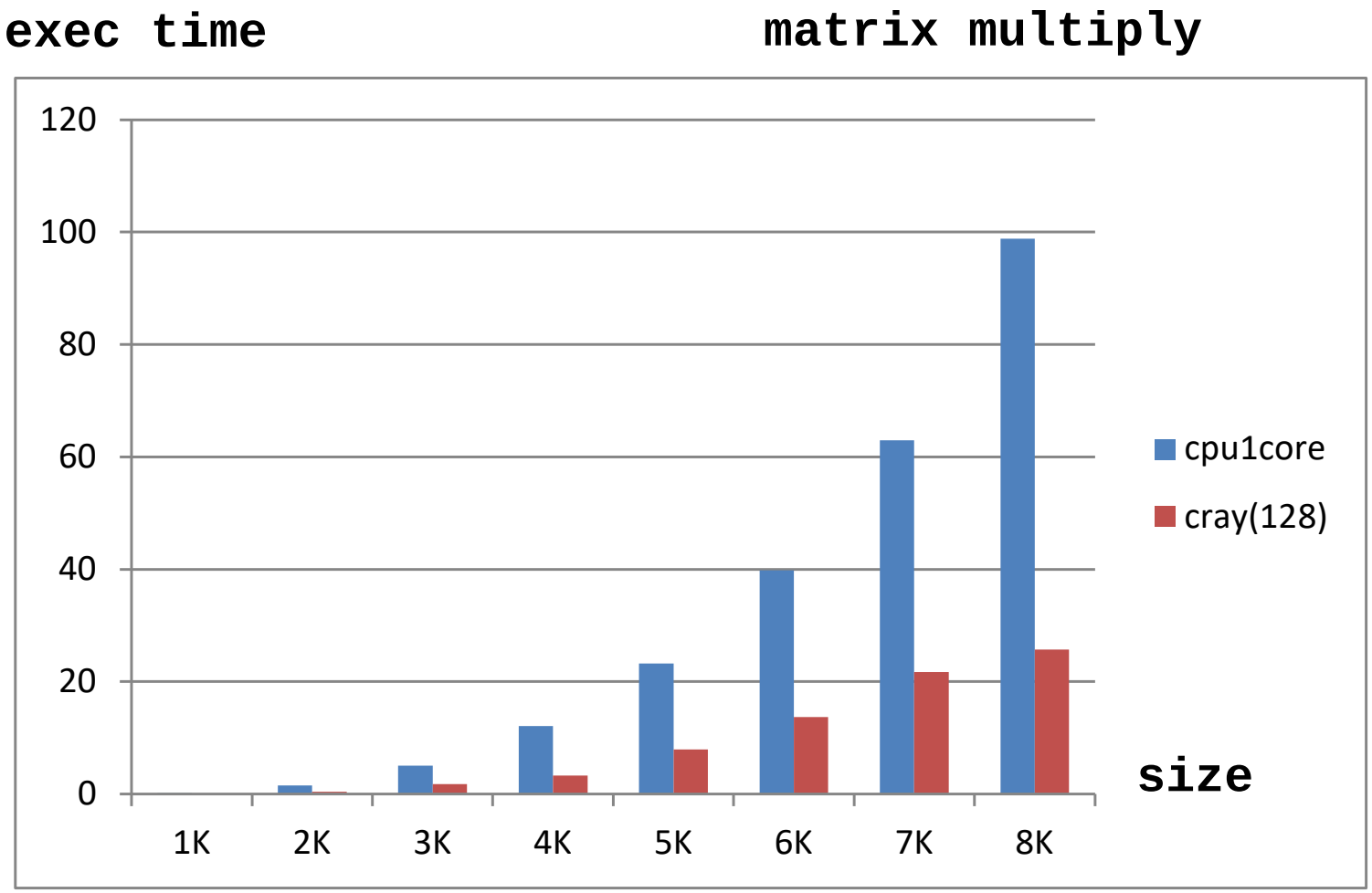
```
#define N 1024

void main(void)
{
  double a[N][N], b[N][N], c[N][N];
  int i,j;
  // ... setup data ...
#pragma acc parallel loop copyin(a, b) copyout(c)
  for(i = 0; i < N; i++){
#pragma acc loop
    for(j = 0; j < N; j++){
      int k;
      double sum = 0.0;
      for(k = 0; k < N; k++){
       sum += a[i][k] * b[k][j];
      }
      c[i][j] = sum;
    }
  }
}
```

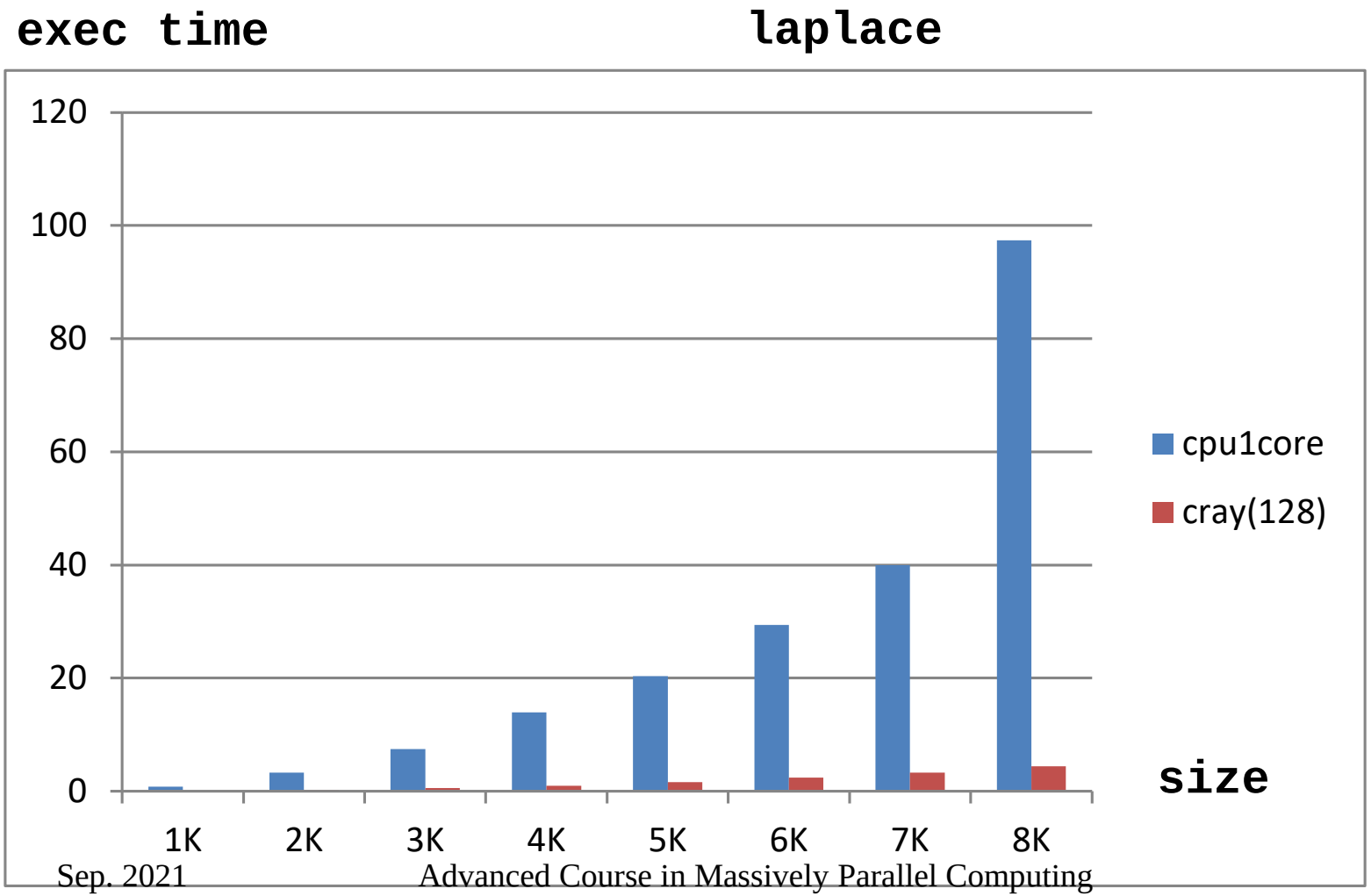
Stencil Code (Laplace Solver) in OpenACC

```
#define XSIZE 1024
#define YSIZE 1024
#define ITER 100
int main(void){
  int x, y, iter;
  double u[XSIZE][YSIZE], uu[XSIZE][YSIZE];
  // setup ...
#pragma acc data copy(u, uu)
  {
    for(iter = 0; iter < ITER; iter++){
      //old <- new
#pragma acc parallel loop
      for(x = 1; x < XSIZE-1; x++){
#pragma acc loop
        for(y = 1; y < YSIZE-1; y++)
          uu[x][y] = u[x][y];
      }
      //update
#pragma acc parallel loop
      for(x = 1; x < XSIZE-1; x++){
#pragma acc loop
        for(y = 1; y < YSIZE-1; y++)
            u[x][y] = (uu[x-1][y] + uu[x+1][y]
                       + uu[x][y-1] + uu[x][y+1]) / 4.0;
      }}
  } //acc data end
}
```

Performance of OpenACC code



Performance of OpenACC code



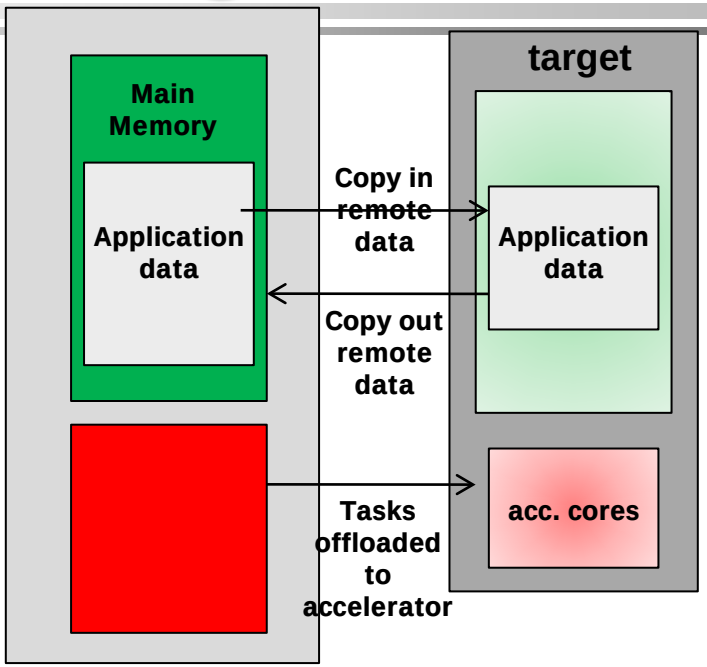
OpenMP 4.0

- **Released July 2013**
  - **http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf**
  - **A document of examples is expected to release soon**
- **Changes from 3.1 to 4.0 (Appendix E.1):**
  - ***Accelerator: 2.9***
  - ***SIMD extensions: 2.8***
  - ***Places and thread affinity: 2.5.2, 4.5***
  - ***Taskgroup and dependent tasks: 2.12.5, 2.11***
  - ***Error handling: 2.13***
  - ***User-defined reductions: 2.15***
  - ***Sequentially consistent atomics: 2.12.6***
  - ***Fortran 2003 support***

slide by Yonghong@UH

Accelerator (2.9): offloading

- Execution Model: Offload data and code to accelerator
- *target* construct creates tasks to be executed by devices
- Aims to work with wide variety of accs
  - GPGPUs, MIC, DSP, FPGA, etc
  - A target could be even a remote node, intentionally




```
#pragma omp target
{
    /* it is like a new task
     * executed on a remote device */
{
```


slide by Yonghong@UH

target and map examples

```
void vec_mult(int N)
{
   int i;
   float p[N], v1[N], v2[N];
   init(v1, v2, N);
   #pragma omp target map(to: v1, v2) map(from: p)
   #pragma omp parallel for
   for (i=0; i<N; i++)
     p[i] = v1[i] * v2[i];
   output(p, N);
}
```

```
void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   init(v1, v2, N);
   #pragma omp target map(to: v1[0:N], v2[:N]) map(from: p[0:N])
   #pragma omp parallel for
   for (i=0; i<N; i++)
     p[i] = v1[i] * v2[i];
   output(p, N);
}
```

Kokkos

- C++ template library for both CPU (SIMD/Multicore) and GPU
- Background: All US exascale systems will have GPUs
- Pushed by US ECP (Exascale Computing Project)

- Online Resources:
  - Primary Kokkos GitHub Organization
    - https://github.com/kokkos:
  - Lecture Series:
    - https://github.com/kokkos/kokkos-tutorials/
    - Find the slides shown in this lecture in later pages

The HPC Hardware Landscape

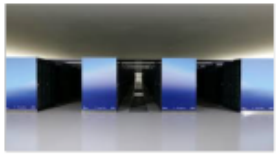
Current Generation: Programming Models OpenMP 3, CUDA and OpenACC depending on machine

LANL/SNL Trinity
Intel Haswell / Intel KNL
OpenMP 3

LLNL SIERRA
IBM Power9 / NVIDIA Volta
CUDA / *OpenMP*[a]

ORNL Summit
IBM Power9 / NVIDIA Volta
CUDA / *OpenACC / OpenMP*[a]

SNL Astra
ARM CPUs
OpenMP 3



Riken Fugaku
ARM CPUs with SVE
OpenMP 3 / *OpenACC*[b]

Upcoming Generation: Programming Models OpenMP 5, CUDA, HIP and DPC++ depending on machine

NERSC Perlmutter
AMD CPU / NVIDIA GPU
CUDA / *OpenMP 5*[c]

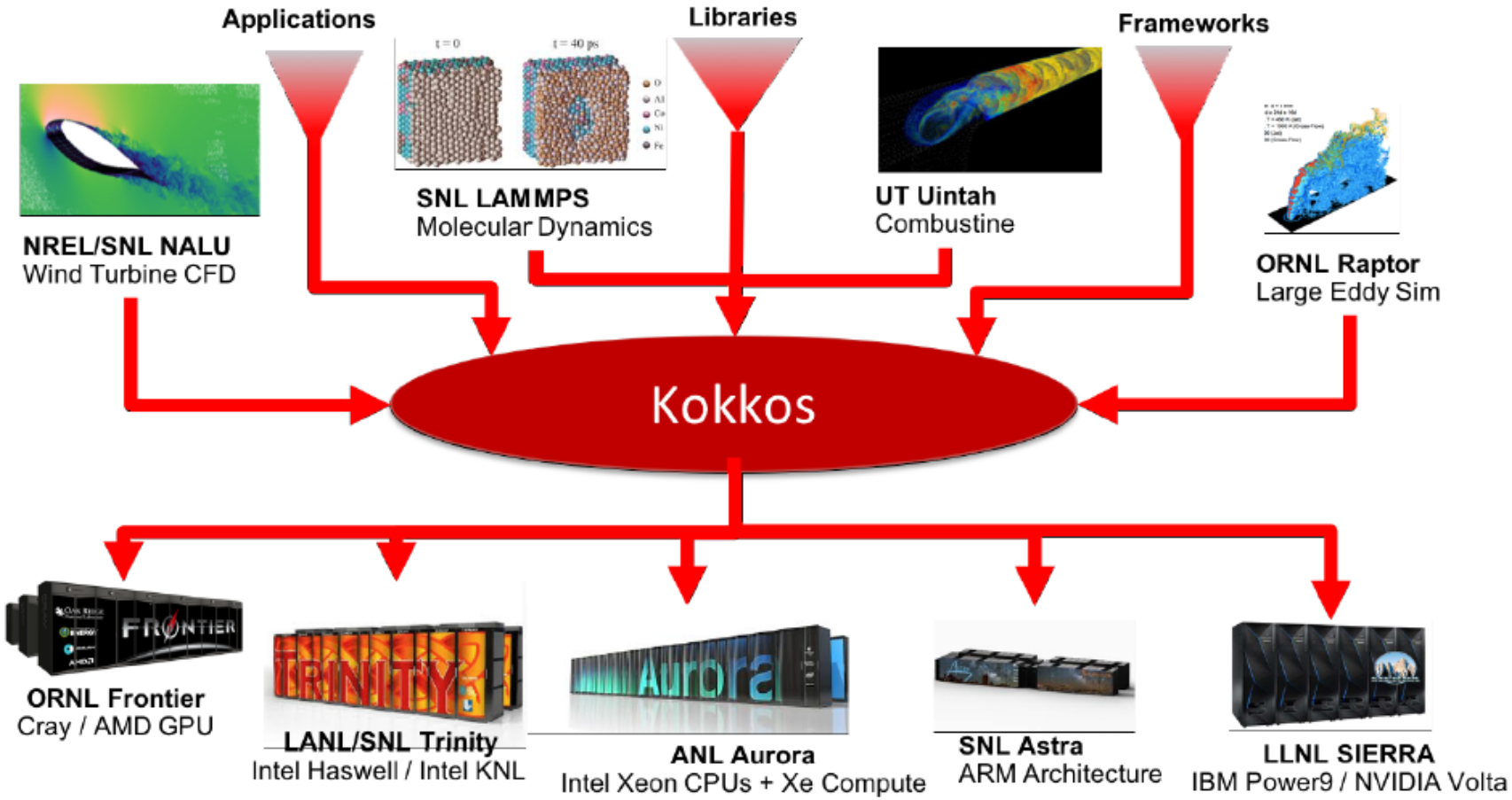
ORNL Frontier
AMD CPU / AMD GPU
HIP / *OpenMP 5*[d]

ANL Aurora
Xeon CPUs / Intel GPUs
DPC++ / *OpenMP 5*[e]

LLNL El Capitan
AMD CPU / AMD GPU
HIP / *OpenMP 5*[d]

(a) Initially not working. Now more robust for Fortran than C++, but getting better.
(b) Research effort.
(c) OpenMP 5 by NVIDIA.
(d) OpenMP 5 by HPE.
(e) OpenMP 5 by Intel.

Kokkos at the Center



What is Kokkos?

- A C++ Programming Model for Performance Portability
  - Implemented as a template library on top CUDA, HIP, OpenMP, ...
  - Aims to be descriptive not prescriptive
  - Aligns with developments in the C++ standard
- Expanding solution for common needs of modern science and engineering codes
  - Math libraries based on Kokkos
  - Tools for debugging, profiling and tuning
  - Utilities for integration with Fortran and Python
- Is is an Open Source project with a growing community
  - Maintained and developed at `https://github.com/kokkos`
  - Hundreds of users at many large institutions

Important Point

There's a difference between *portability* and *performance portability*.

Example: implementations may target particular architectures and may not be *thread scalable*.

(e.g., locks on CPU won't scale to 100,000 threads on GPU)

Goal: write **one implementation** which:

- compiles and **runs on multiple architectures**,
- obtains **performant memory access patterns** across architectures,
- can leverage **architecture-specific features** where possible.

Kokkos: performance portability across manycore architectures

Concepts: Patterns, Policies, and Bodies

Pattern Policy

Body

```
for (element = 0; element < numElements; ++element) {
  total = 0;
  for (qp = 0; qp < numQPs; ++qp) {
    total += dot(left[element][qp], right[element][qp]);
  }
  elementValues[element] = total;
}
```

Terminology:

- **Pattern**: structure of the computations
  for, reduction, scan, task-graph, ...
- **Execution Policy**: how computations are executed
  static scheduling, dynamic scheduling, thread teams, ...
- **Computational Body**: code which performs each unit of work; *e.g.*, the loop body

⇒ The **pattern** and **policy** drive the computational **body**.

What if we want to **thread** the loop?

```
#pragma omp parallel for
for (element = 0; element < numElements; ++element) {
  total = 0;
  for (qp = 0; qp < numQPs; ++qp) {
    total += dot(left[element][qp], right[element][qp]);
  }
  elementValues[element] = total;
}
```

(Change the *execution policy* from "serial" to "parallel.")

OpenMP is simple for parallelizing loops on multi-core CPUs, but what if we then want to do this on **other architectures**?

Intel PHI *and* NVIDIA GPU *and* AMD GPU *and* ...

"Parallel for" on a GPU via pragmas

Option 1: OpenMP 4.5

```
#pragma omp target data map(...)
#pragma omp teams num_teams(...) num_threads(...) private(...)
#pragma omp distribute
for (element = 0; element < numElements; ++element) {
  total = 0
#pragma omp parallel for
  for (qp = 0; qp < numQPs; ++qp)
    total += dot(left[element][qp], right[element][qp]);
  elementValues[element] = total;
}
```

Option 2: OpenACC

```
#pragma acc parallel copy(...) num_gangs(...) vector_length(...)
#pragma acc loop gang vector
for (element = 0; element < numElements; ++element) {
  total = 0;
  for (qp = 0; qp < numQPs; ++qp)
    total += dot(left[element][qp], right[element][qp]);
  elementValues[element] = total;
}
```

Portable, but not performance portable

A standard thread parallel programming model *may* give you portable parallel execution *if* it is supported on the target architecture.

But what about performance?

Performance depends upon the computation's **memory access pattern**.

Data parallel patterns and work

```
for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex) {
  atomForces[atomIndex] = calculateForce(...data...);
}
```

Kokkos maps **work** to execution resources

- each iteration of a computational body is a **unit of work**.
- an **iteration index** identifies a particular unit of work.
- an **iteration range** identifies a total amount of work.

Important concept: Work mapping

You give an **iteration range** and **computational body** (kernel) to Kokkos, and Kokkos decides how to map that work to execution resources.

How are computational bodies given to Kokkos?

As **functors** or *function objects*, a common pattern in C++.

Quick review, a **functor** is a function with data. Example:

```
struct ParallelFunctor {
  ...
  void operator()( a work assignment ) const {
    /* ... computational body ... */
  ...
};
```

How is work assigned to functor operators?

A total amount of work items is given to a Kokkos pattern,

```
ParallelFunctor functor;
Kokkos::parallel_for(numberOfIterations, functor);
```

and work items are assigned to functors one-by-one:

```
struct Functor {
  void operator()(const int64_t index) const {...}
}
```

Warning: concurrency and order

Concurrency and ordering of parallel iterations is *not* guaranteed by the Kokkos runtime.

Putting it all together: the complete functor:

```
struct AtomForceFunctor {
  ForceType _atomForces;
  AtomDataType _atomData;
  AtomForceFunctor(/* args */) {...}
  void operator()(const int64_t atomIndex) const {
    _atomForces[atomIndex] = calculateForce(_atomData);
  }
};
```

Q/ How would we **reproduce serial execution** with this functor?

Serial

```
for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex){
  atomForces[atomIndex] = calculateForce(data);
}
```

Functor

```
AtomForceFunctor functor(atomForces, data);
for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex){
  functor(atomIndex);
}
```

The complete picture (using functors):

1. Defining the functor (operator+data):

```
struct AtomForceFunctor {
  ForceType _atomForces;
  AtomDataType _atomData;

  AtomForceFunctor(ForceType atomForces, AtomDataType data) :
    _atomForces(atomForces), _atomData(data) {}

  void operator()(const int64_t atomIndex) const {
    _atomForces[atomIndex] = calculateForce(_atomData);
  }
}
```

2. **Executing** in parallel with Kokkos pattern:

```
AtomForceFunctor functor(atomForces, data);
Kokkos::parallel_for(numberOfAtoms, functor);
```

How does this compare to OpenMP?

Serial

```
for (int64_t i = 0; i < N; ++i) {
  /* loop body */
}
```

OpenMP

```
#pragma omp parallel for
for (int64_t i = 0; i < N; ++i) {
  /* loop body */
}
```

Kokkos

```
parallel_for(N, [=] (const int64_t i) {
  /* loop body */
});
```

Important concept

Simple Kokkos usage is **no more conceptually difficult** than OpenMP, the annotations just go in different places.

Example: Scalar integration

OpenMP

```
double totalIntegral = 0;
#pragma omp parallel for reduction(+:totalIntegral)
for (int64_t i = 0; i < numberOfIntervals; ++i) {
  totalIntegral += function(...);
}
```

Kokkos

```
double totalIntegral = 0;
parallel_reduce(numberOfIntervals,
  [=] (const int64_t i, double & valueToUpdate) {
    valueToUpdate += function(...);
  },
  totalIntegral);
```

- The operator takes **two arguments**: a work index and a value to update.
- The second argument is a **thread-private value** that is managed by Kokkos; it is not the final reduced value.

Final remarks

- **GPGPU is a good solution for apps which can be parallelized for GPU.**
  - **It can be very good esp. when the app fits into one GPU.**
  - **If the apps needs more than one GPU, the cost of communication will kill performance. (in case of HPC)**

- **Programming in CUDA is still difficult ...**
  - **Performance tuning, memory layout ...**
  - **OpenACC and OpenMP will help you!**

- **GPU is now a main device to accelerate many kinds of computing**
  - **Not only NVIDA, but also AMD and Intel**
  - **Kokkos is supposed to support a variety of GPU and also CPU**

- **Many programming models and environments are proposed**