

GPU Programming APIs



| epcc |

Reusing this material



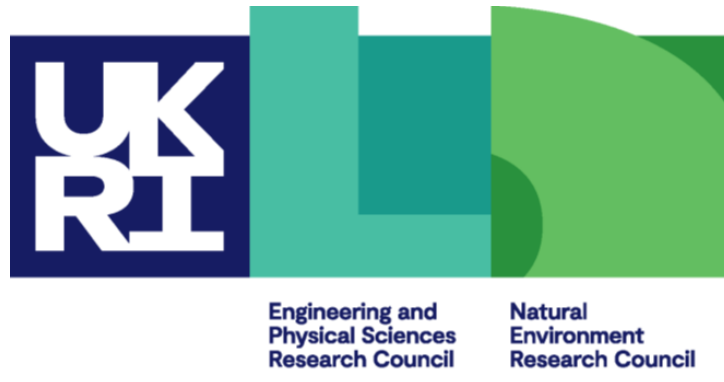
This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

This means you are free to copy and redistribute the material and adapt and build on the material under the following terms: You must give appropriate credit, provide a link to the license and indicate if changes were made. If you adapt or build on the material, you must distribute your work under the same license as the original.

Note that this presentation contains images owned by others. Please seek their permission before reusing these images.

Partners



THE UNIVERSITY
of EDINBURGH



Hewlett Packard
Enterprise

Introduction



I want my application to run on GPUs, but there is a bewildering variety of APIs to program them. How do I choose between them?



Overview



- We will try to explain the differences (and similarities) between them.
- Focus on common HPC base languages (C, C++, Fortran)
 - won't cover e.g. Python, Julia, or ML frameworks
- We will look at:
 - programming style (directives vs kernels vs frameworks)
 - which base languages are supported
 - portability across hardware platforms
 - library support and interoperability
 - whether they are open or proprietary specifications and who administers them



who

Programming style



Three more-or-less distinct programming styles used by GPU programming APIs, *kernels*, *directives* and *portability frameworks*.

Kernels

- Programmer writes a function (or C++ lambda) containing the code to be executed on GPU.
- Function will be executed by every thread on the GPU – loop over threads is implicit.
- On the host (CPU) we pass the function to a method that offloads to the GPU and controls the number of threads.
- Data movement between host and device handled by API library calls.

Directives

- Programmer annotates a block of code (typically a loop) with offload directive(s).
- Code block will be executed on GPU, with loop iterations assigned to GPU threads.
- Number of threads can be specified or set via implementation defaults.
- Data movement controlled by clauses on the directive (or other directives) - some can be done implicitly.

Programming style



Portability Frameworks

- Programmer specifies abstract parallel execution (usually parallel loops)
- Can also specify abstract data layouts
- Framework can generate code for execution on CPU or GPU using a low-level API (e.g., OpenMP, CUDA)
- Typically restricted to C++

Programming style – pros and cons



Directive-based APIs typically:

- 😊 Require less programming effort
- 😊 Are more familiar to HPC programmers (c.f. OpenMP for CPUs)
- 😐 Have fewer features
- 😐 Are more reliant on compiler/runtime for optimisation
- 😐 Can make it difficult to do low-level tuning

Programming style – pros and cons



Portability frameworks typically:

😊 Facilitate a single code base that runs efficiently on CPUs and GPUs

😐 Have limited features

😐 Are very reliant on compiler/runtime for optimisation

😐 Can make it difficult to do low-level tuning

😐 Are restricted to C++

Base languages



- All the APIs we are considering are (loosely speaking) extensions of C, C++ or Fortran.
- Some of the APIs support all three base languages.
- Some only support one (usually C++).
- APIs may differ in the versions of the base languages they support.
 - If you want to use the latest features in your favourite base language, you may need to read the small print carefully to see whether the GPU API supports these too.

Portability



- Until fairly recently, the HPC GPU market was completely dominated by NVIDIA.
- Now seeing competition from AMD and Intel (probably!).
- Portability across GPU hardware platforms now has to be a serious consideration for applications (if it wasn't already).
- Note that functional portability does not necessarily imply performance portability!
- It would also be convenient for the same code to work on both CPUs and GPUs.
 - Performance portability is an even bigger problem here.

Library support and interoperability



- For some applications, access to numerical libraries (e.g., BLAS, FFTs) on the GPU is very useful.
- Some APIs are better than others in this respect.
- However, some APIs are interoperable with others, so, for example, one can have OpenMP directives and CUBLAS calls in the same code.

Open vs proprietary specifications



Open specifications are:

- 😊 More likely to be portable across hardware platforms.
- 😊 More likely to be supported by a sound specification process, run by a consortium of users and implementors.
- 😊 More likely to survive commercial implosions and inter-vendor political squabbles.
- 😊 More likely to retain backwards compatibility with older versions.
- 😞 Less likely to exploit vendor-specific hardware features.
- 😞 Less likely to react quickly to support new base language features.

Example: SAXPY



Sequential CPU code:

```
for (int i = 0; i < n; i++) {  
    y[i] = a * x[i] + y[i];  
}
```

```
do i = 1, n  
    y(i) = a*x(i) + y(i)  
end do
```

CUDA



Programming style: explicit kernels

Limited directive support in CUDA Fortran.

Portability: NVIDIA only

It is possible to translate semi-automatically to HIP, which will run on AMD.

Base languages: C (originally), C++, CUDA Fortran is a language extension.

Library support: good: BLAS, FFTs, maths, sparse BLAS, direct linear solvers, random numbers.

Interoperability with OpenMP and OpenACC.

Specification: proprietary to NVIDIA

Notes: well-established API for NVIDIA hardware (see also HIP).

CUDA SAXPY



Host code

```
cudaMalloc(&d_x,    N*sizeof(float));
cudaMalloc(&d_y,    N*sizeof(float));
cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);

cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
cudaFree(d_x);
cudaFree(d_y);
```

Kernel

```
__global__ void saxpy(int n, float a, float *x, float *y) {

    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
```


Programming style: kernels

Portability: AMD and NVIDIA (via shim layer)

Base languages: C++

HIPFort provides a Fortran interface on the host side, but the kernels are C++ only.

Library support: OK: BLAS, FFTs, random numbers, sparse BLAS

On NVIDIA, these are just wrappers for CUDA libraries.

Specification: proprietary to AMD

Notes: API directly shadows CUDA; not all the latest CUDA features may be implemented.

HIP SAXPY



Host code

```
hipMalloc(&d_x,    N*sizeof(float));
hipMalloc(&d_y,    N*sizeof(float));
hipMemcpy(d_x, x, N*sizeof(float), hipMemcpyHostToDevice);
hipMemcpy(d_y, y, N*sizeof(float), hipMemcpyHostToDevice);

saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);

hipMemcpy(y, d_y, N*sizeof(float), hipMemcpyDeviceToHost);
hipFree(d_x);
hipFree(d_y);
```

Kernel

```
__global__ void saxpy(int n, float a, float *x, float *y) {

    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
```

OpenCL



Programming style: kernels

Portability: AMD, NVIDIA , Intel, will also run on CPUs.

Base languages: C only

Library support: Lots, but mainly open source / community efforts.

Specification: Open, administered by Kronos Group.

Notes: Designed to be portable across CPUs and GPUs, but performance on both is questionable. Kernels are JIT (just-in-time) compiled. Lower-level and more cumbersome than CUDA/HIP, requires a lot of verbose setup code.

OpenCL SAXPY Host code (1)



```
// Get platform and device information
cl_platform_id * platforms = NULL;
cl_uint      num_platforms;
//Set up the Platform
cl_int clStatus = clGetPlatformIDs(0, NULL, &num_platforms);
platforms = (cl_platform_id *)
malloc(sizeof(cl_platform_id)*num_platforms);
clStatus = clGetPlatformIDs(num_platforms, platforms, NULL);
//Get the devices list and choose the device you want to run on
cl_device_id      *device_list = NULL;
cl_uint          num_devices;
clStatus = clGetDeviceIDs( platforms[0], CL_DEVICE_TYPE_GPU, 0,NULL, &num_devices);
device_list = (cl_device_id *)
malloc(sizeof(cl_device_id)*num_devices);
```

OpenCL SAXPY Host code (2)



```
clStatus = clGetDeviceIDs( platforms[0],CL_DEVICE_TYPE_GPU, num_devices, device_list, NULL);
// Create one OpenCL context for each device in the platform
cl_context context;
context = clCreateContext( NULL, num_devices, device_list, NULL, NULL, &clStatus);
// Create a command queue
cl_command_queue command_queue = clCreateCommandQueue(context, device_list[0], 0, &clStatus);
// Create memory buffers on the device for each vector
cl_mem X_clmem = clCreateBuffer(context, CL_MEM_READ_ONLY,VECTOR_SIZE * sizeof(float), NULL,
&clStatus);
cl_mem Y_clmem = clCreateBuffer(context, CL_MEM_READ_ONLY,VECTOR_SIZE * sizeof(float), NULL,
&clStatus);
// Copy the Buffer X and Y to the device
clStatus = clEnqueueWriteBuffer(command_queue, X_clmem, CL_TRUE, 0, VECTOR_SIZE *
sizeof(float), X, 0, NULL, NULL);
clStatus = clEnqueueWriteBuffer(command_queue, Y_clmem, CL_TRUE, 0, VECTOR_SIZE *
sizeof(float), Y, 0, NULL, NULL);
```

OpenCL SAXPY Host code (3)



```
// Create a program from the kernel source
cl_program program = clCreateProgramWithSource(context, 1, (const char **)&saxpy_kernel,
NULL, &clStatus);
// Build the program
clStatus = clBuildProgram(program, 1, device_list, NULL, NULL, NULL);
// Create the OpenCL kernel
cl_kernel kernel = clCreateKernel(program, "saxpy_kernel", &clStatus);
// Set the arguments of the kernel
clStatus = clSetKernelArg(kernel, 0, sizeof(float), (void *)&a);
clStatus = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&Y_clmem);
clStatus = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&X_clmem);
// Execute the OpenCL kernel on the list
size_t global_size = VECTOR_SIZE; // Process the entire lists
size_t local_size = 64;           // Process one item at a time
clStatus = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, &global_size,
&local_size, 0, NULL, NULL);
```

OpenCL SAXPY Host code (4)



```
// Read the cl memory Y_clmem on device to the host variable Y
    clStatus = clEnqueueReadBuffer(command_queue, Y_clmem, CL_TRUE, 0, VECTOR_SIZE *
sizeof(float), Y, 0, NULL, NULL);

    // Clean up and wait for all the comands to complete.
    clStatus = clFlush(command_queue);
    clStatus = clFinish(command_queue);
    // Finally release all OpenCL allocated objects and host buffers.
    clStatus = clReleaseKernel(kernel);
    clStatus = clReleaseProgram(program);
    clStatus = clReleaseMemObject(X_clmem);
    clStatus = clReleaseMemObject(Y_clmem);
    clStatus = clReleaseCommandQueue(command_queue);
    clStatus = clReleaseContext(context);
```

OpenCL SAXPY Kernel



```
__kernel void saxpy_kernel(float a, __global float *x, __global float *y) {  
  
    int index = get_global_id(0);  
    y[index] = a * x[index] + y[index];  
  
}
```


SYCL



Programming style: kernels, but with support for parallel loops

Portability: AMD, NVIDIA, Intel

Base languages: C++ only

Library support: Limited: BLAS and parallel STL

Specification: Open, administered by Kronos Group.

Notes: Higher level interface, properly integrated with C++. Kernels typically take the form of C++ lambdas. Relatively immature with not many (compiler) implementations. Can also generate CPU code, but not as abstract as portability frameworks.

SYCL SAXPY



```
sycl::queue q(sycl::default_selector{});  
const float a;  
sycl::buffer<float,1> d_X { X.data(), sycl::range<1>(X.size()) };  
sycl::buffer<float,1> d_Y { Y.data(), sycl::range<1>(Y.size()) };  
  
q.submit([&](sycl::handler& h) {  
    auto X = d_X.get_access<sycl::access::mode::read>(h);  
    auto Y = d_Y.get_access<sycl::access::mode::read_write>(h);  
    h.parallel_for<class axpy>( sycl::range<1>{length}, [=] (sycl::id<1> it) {  
        const size_t i = it[0];  
        Y[i] += a * X[i] + Y[i];  
    });  
});  
  
q.wait();
```

OpenMP offloading



Programming style: directives

Portability: AMD, NVIDIA , Intel

Meta-directives help support CPU and GPU versions in the same source base.

Base languages: C, C++, Fortran

Library support: Not much, but interoperable with e.g. CUDA libraries.

Specification: Open, administered by OpenMP ARB.

Notes: Now relatively mature and pretty widely supported. Fully integrated with CPU OpenMP on the host side. Designed for devices other than GPUs as well, hence some minor quiriness!

OpenMP SAXPY



```
#pragma omp target teams distribute parallel for map(tofrom:y[0:n]) map(to:a,x[0:n])  
for (int i = 0; i < n; i++) {  
    y[i] = a * x[i] + y[i];  
}
```

```
!$omp target teams distribute parallel do map(to from: y) map(to: a, x)  
do i = 1, n  
    y(i) = a*x(i) + y(i)  
end do  
!$omp end target teams distribute parallel do
```

OpenACC



Programming style: directives

Portability: AMD, NVIDIA

Base languages: C, C++, Fortran

Library support: Not much, but interoperable with e.g. CUDA libraries.

Specification: Open, administered by OpenACC Organization.

Notes: Pre-dates OpenMP offloading, but now has fewer implementations than OpenMP offloading. OpenMP and OpenACC have very similar functionality, although OpenACC is a little more GPU-specific.

OpenACC SAXPY



```
#pragma acc data copyin(x[0:n], y[0:n]) copyout(y[0:n])
{
    #pragma acc parallel loop
    for (int i = 0; i < n; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

```
!$omp acc data copyin(x, y) copyout(y)
!$omp acc parallel loop
do i = 1, n
    y(i) = a*x(i) + y(i)
end do
!$omp acc end parallel loop
!$omp acc end data
```

Kokkos



Programming style: portability framework

Portability: AMD, NVIDIA, Intel, CPUs

Base languages: C++

Library support: Limited: some BLAS, sparse BLAS, graph ops, subset of Trilinos.

Specification: Open-ish, administered by consortium of (mainly) US labs.

Notes: Designed to generate efficient code for GPUs and CPUs from the same source, and is implemented on top of e.g., CUDA, HIP, OpenMP, C++ threads.

Kokkos SAXPY



```
parallel_for(N, [=] (const size_t i)
{
    y[i] = a * x[i] + y[i];
});
```


Programming style: portability framework

Portability: AMD, NVIDIA, Intel, CPUs

Base languages: C++

Library support: Limited/none

Specification: Informal. Designed and administered by Lawrence Livermore National Laboratory (LLNL).

Notes: Designed to generate efficient code for GPUs and CPUs from the same source, and is implemented on top of e.g., CUDA, HIP, SYCL, OpenMP offload, OpenMP(CPU), Intel TBB. Unlike Kokkos, has interfaces to use specific lower-level API features.

RAJA SAXPY



```
RangeSegment seg(0, N)
forall <loop_exec> (seg, [=] (int i)
{
    y[i] = a * x[i] + y[i];
});
```

References



Further reading and resources:

1. [Introduction to GPU programming models — GPU programming: why, when and how? Documentation](#)
2. [11 Directive-based GPU programming - Parallel and High Performance Computing \[Book\]](#)
3. [Directive-based Programming Models for Scientific Applications - A Comparison | IEEE Conference Publication | IEEE Xplore](#)
4. [A Complete Introduction to GPU Programming With Practical Examples in CUDA and Python | Cherry Servers](#)