Introduction to GPU programming models

Andrey Alekseenko

Science for Life Laboratory, KTH Royal Institute of Technology, Sweden

Introduction to GPU, Stockholm/Zoom, 15 September, 2023

Presentation based on the ENCCS lesson "Introduction to GPU programming models"

GPU Programming models

GPUs are great! How do we use them?

- C++/Fortran:
  - Standard-based solutions
  - Directive-based programming
  - Non-portable kernel-based models
  - Portable kernel-based models
- Higher-level languages
- Exotic languages

N.B.: The classification above is neither definitive nor exhaustive.

Standard-based models

Use C++/Fortran to offload algorithms from the standard library to the GPU.

Despite being "standard", the support for them is extremely limited.

Limited flexibility and performance.

C++ Standard Parallelism

- Introduced in C++17
- Automatic parallization of algorithms from C++ standard library

```
int ave_age = // TODO: Replace
    std::transform_reduce(std::execution::par_unseq,
                          employees.begin(), employees.end(),
                          0, std::plus<int>(),
                          [](const Employee& emp){
                              return emp.age();
                          })
    / employees.size();
```

- Supported on NVIDIA GPUs via NVIDIA HPC SDK (`nvc++`)
- Experimental support for other GPUs via hipSYCL/OpenSYCL.

Fortran Standard Parallelism

```
subroutine saxpy(x,y,n,a)  $ TODO: replace
  real :: a, x(:), y(:)
  integer :: n, i
  do concurrent (i = 1: n)
    y(i) = a*x(i)+y(i)
  enddo
end subroutine saxpy
```

- Supported on NVIDIA GPUs via NVIDIA HPC SDK (`nvfortran`)

Directive-based programming

- Annotate existing *serial* code with *hints* to indicate which loops and regions to execute on the GPU.
- Minimal changes to the C++/Fortran source code.
- Limited performance and expressiveness.

OpenACC

Developed in 2010 specifically to target accelerators.

OpenMP

Developed in 1997 for multicore CPUs, accelerator support added later.

OpenACC

```
#pragma acc parallel loop
for (i = 0; i < NX; i++) {
  vecC[i] = vecA[i] + vecB[i];
}
```

```
!$acc parallel loop
do i = 1, nx
  vecC(i) = vecA(i) + vecB(i)
end do
!$acc end parallel loop
```

OpenMP

```
#pragma omp target
for (i = 0; i < NX; i++) {
  vecC[i] = vecA[i] + vecB[i];
}
```

```
!$omp target
do i = 1, nx
  vecC(i) = vecA(i) + vecB(i)
end do
!$omp end target
```

Kernel-based programming

- Requires deep knowledge of GPU architecture.
- The code to execute on the GPU is outlined into a separate function (kernel).
- Low-level code (conceptually), more control, higher performance (or not).

Non-portable models

CUDA

- First mainstream GPU computing framework, widely used and supported.
- Developed by NVIDIA, works only on NVIDIA GPUs.
- Extensive ecosystem, libraries, tutorial.
- Stable, feature-rich, well-supported.

HIP

- Developed by AMD, works on AMD and NVIDIA* GPUs.
- *Very* similar to CUDA, automatic conversion tools exist.
- Less mature, no official support on consumer GPUs.

CUDA

```
__global__ void vector_add(float *A, float *B, float *C, int n) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if (tid < n) {
    C[tid] = A[tid] + B[tid];
  }
}
```

```
// Allocate GPU memory
cudaMalloc((void**)&Ad, N * sizeof(float)); // ...
// Copy the data from the CPU to the GPU
cudaMemcpy(Ad, Ah, sizeof(float) * N, cudaMemcpyHostToDevice); // ...
// Define grid dimensions: how many threads to launch
dim3 blocks{256, 1, 1};
dim3 threads{(N/256)i*256 + 1, 1, 1};
vector_add<<<blocks, threads>>>(Ad, Bd, Cd, N);
```

HIP

```
__global__ void vector_add(float *A, float *B, float *C, int n) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if (tid < n) {
    C[tid] = A[tid] + B[tid];
  }
}
```

```
// Allocate GPU memory
hipMalloc((void**)&Ad, N * sizeof(float)); // ...
// Copy the data from the CPU to the GPU
hipMemcpy(Ad, Ah, sizeof(float) * N, hipMemcpyHostToDevice); // ...
// Define grid dimensions: how many threads to launch
dim3 blocks{256, 1, 1};
dim3 threads{(N + 255) / 256, 1, 1};
vector_add<<<blocks, threads>>>(Ad, Bd, Cd, N);
```

Portable models: OpenCL

- Cross-platform open standard for accelerator programming, based on C.
- Supports GPUs, FPGAs, embedded devices.
- Supported by all major vendors, but to a varying degree. No latest features.
- Good performance requires vendor extensions, defeating portability aspect.
- Separate-source model, no compiler support required: just headers and a runtime library.

OpenCL: kernel

```
static const char* kernel_source = R"(
  __kernel void vector_add(__global float *A, __global float *B, __global float *C, int n) {
    int tid = get_global_id(0);
    if (tid < n) {
      C[tid] = A[tid] + B[tid];
    }
  }
)";
```

OpenCL: launch code

```
// Boilerplate...
cl_platform_id platform;
clGetPlatformIDs(1, &platform, NULL);
cl_device_id device;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
cl_context context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
cl_command_queue queue = clCreateCommandQueue(context, device, 0, NULL);
cl_program program = clCreateProgramWithSource(context, 1, &kernel_source, NULL, NULL);
clBuildProgram(program, 1, &device, NULL, NULL, NULL);
cl_kernel kernel = clCreateKernel(program, "vector_add", NULL);
// Allocate the arrays on GPU
cl_mem d_A = clCreateBuffer(context, CL_MEM_READ_ONLY, N * sizeof(float), NULL, NULL); // ...
// Copy the data from CPU to GPU
clEnqueueWriteBuffer(queue, d_A, CL_TRUE, 0, N * sizeof(float), Ah, 0, NULL, NULL); // ..
// Set arguments and launch the kernel
clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_A);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_B);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_C);
cl_int N_as_cl_int = N;
clSetKernelArg(kernel, 3, sizeof(cl_int), &N_as_cl_int);
size_t globalSize = N;
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalSize, NULL, 0, NULL, NULL);
```

Portable models: SYCL

- Cross-platform open standard for accelerator programming, inspired by OpenCL.
- Single-source model based on C++17, requires compiler support and C++ knowledge.
- Two major implementations: Intel oneAPI DPC++ and hipSYCL/OpenSYCL.
- Supported officially only by Intel, works via CUDA/HIP for NVIDIA/AMD.
- Interoperability with native features (at the expense of portability)

SYCL: USM approach

```
// Boilerplate
sycl::queue queue{{sycl::property::queue::in_order()}};
// Allocate GPU memory
int* A = sycl::malloc_device<float>(N, queue);  // ...
// Copy the data from CPU to GPU
queue.copy<float>(Ah, A, N); // ...
// Submit a kernel into a queue; cgh is a helper object
queue.submit([&](sycl::handler &cgh) {
  cgh.parallel_for<class Kernel>(sycl::range<1>{N}, [=](sycl::id<1> i) {
      C[i] = A[i] + B[i];
  });
});
```

There is also support for automatic memory management via SYCL buffers.

Kokkos

- Open-source programming model for heterogeneous parallel computing, mostly developed at Sandia National Laboratories.
- Single source model, similar to SYCL (but earlier).
- No official vendor support, implemented on top of other frameworks.
- Interoperability with native features (at the expense of portability)

Higher-level language support: Julia

Julia has first-class support for GPU programming through the following packages that target GPUs from all three major vendors:

- `CUDA.jl` for NVIDIA GPUs
- `AMDGPU.jl` for AMD GPUs
- `oneAPI.jl` for Intel GPUs
- `Metal.jl` for Apple M-series GPUs

Higher-level language support: Python

- CuPy (NVIDIA-only)
- cuDF (NVIDIA-only)
- PyCUDA (... you guessed it)
- Numba (NVIDIA and deprecated AMD support)

Exotic GPU-specific languages

- Chapel: developed by Cray/HPE for parallel scientific programming, early AMD/NVIDIA support.

```
on here.gpus[0] {
  var A, B, C : [0..#N] real(32); // Allocate
  A = Ah; // Copy from CPU to GPU, ...
  forall i in 0..#N { C[i] = A[i] + B[i]; };
}
```

- Futhark: functional data-parallel language for GPU programming with OpenCL and CUDA backends.

```
def vecadd xs ys =
  map (\\(x,y) -> x + y) (zip xs ys)
```

Summary

- Standard-based: if you already use STL algorithms and are on NVIDIA
- Directive-based: if you have serial code and no time
- CUDA or HIP: if you are targeting specific vendor and want best performance
- OpenCL: if you want portability on any hardware
- Kokkos or SYCL: if you want performance and portability on modern GPUs
- Higher-level languages: if you have dislike C++ or Fortran
- Exotic languages: if you feeling adventurous

References

TODO

The official OpenMP website