An Introduction to GPU Computing

Background

Graphics Processing Units (GPUs) are extremely powerful compute devices. To understand where GPUs are coming from, consider a three dimensional scene. Typically, three dimensional objects are constructed from a large number of flat surface triangles with coordinates (x_i, y_i, z_i). Now imagine a camera that is moving through the room. In order to display a 2-dimensional image, for each triangle we need to compute the projection onto the two dimensional viewing plane represented through the camera parameters. In addition, we need to deal with surface textures, light sources and computations which triangles are visible and which aren't. This is a huge number of operations that can be done independently for each triangle. GPUs have been designed to perform these highly parallel operations extremely fast. The secret is that GPUs consist of a large number of very simple processing units that are optimised to do these very simple operations but are inefficient for complex operations such as branching.

In the early two thousands it was first recognized that GPUs may be able to be used to accelerate mathematical algorithms other than graphics calculations. In particular, if we have a large number of independent operations on highly structured data, GPUs can give significant performance benefits compared to CPUs. However, not all types of computations can benefit from GPU acceleration. Examples include:

- Highly serial algorithms. If there is no inherent parallelism, a GPU won't help much. CPUs are much better devices for single threaded applications than individual GPU processing units.
- Strongly memory bound computations. If we have large amounts of data but very little to do per data unit, a GPU may not be well suited. We need to transfer the data to the compute device, which for discrete accelerators means system bus transfers that are slow compared to direct communication between CPU and memory. We may interleave memory operations and computations. But this only works well have the computational load is sufficiently high. Some modern integrated CPU/GPU architectures share the RAM. For these such considerations are less relevant.
- Computations on highly unstructured data, or adaptive methods. Computations involving complex data structures that may change during the course of the computation are a challenge for GPU accelerators. The more complex the flow of the computation, the harder to match to a GPU, and for highly adaptive computations CPUs are typically preferable.

When GPU computing first took off there were many publications that promised incredible speed-ups of factors of fifty or even one hundred by using cheap gaming GPUs instead of high-end CPUs. This is not the reality. Such speed-ups are only achievable in very specific highly parallel applications (e.g. certain Monte-Carlo methods) or in synthetic benchmarks. Also, authors often compared a carefully tuned GPU implementation with a non-optimised single-threaded CPU code. Today, the field of GPU computing is much more mature and the trend is towards heterogeneous computing in which CPUs and GPUs work together for different parts of a computation.

GPU Computing Standards and languages

There are a number of ways to develop software for GPU accelerators. Here, we want to give a brief overview of the most important ones.

- Nvidia Cuda is arguably the most advanced development platform for GPU computing. Cuda can be used from C/C++, Fortran, Python, Matlab, Julia, and others. There exists a large ecosystem of GPU computing libraries that are built on Cuda. The major disadvantage of Cuda is that it depends on Nvidia's proprietary Cuda drivers and is restricted to Nvida hardware. Thus, it is not a completely open or portable solution.
- OpenCL is the open heterogeneous computing standard from the Khronos Group. It does not only support GPU accelerators but also CPUs, FPGA's and essentially any type of device that provides an OpenCL interface. OpenCL has a number of similarities to Cuda. However, it is slightly more complex in its usage owning to the fact that it needs to support devices from many different vendors. It is widely used but much less common in High-Performance Computing than Cuda. Most vendors, including Nvidia, AMD and Intel support OpenCL. For HPC projects it may be useful to consider Sycl instead of OpenCL. It grew out of OpenCL and is quickly gaining support in the HPC world, largely due to signficant software investment from Codeplay and Intel.
- Sycl is a modern heterogeneous compute standard from the Khronos Group. It is built on standard C++ and quickly gaining usage share in the HPC world. Codeplay has in recent years developed efficient compilers for Sycl that can target a number of OpenCL and Cuda type devices. Intel's OneAPI is compatible to Sycl and is

the standard language for Intel's Xe architecture of GPU accelerators.

- OneApi is Intel's contribution to the development of an Industry wide standard for heterogeneous computing. It is essentially an extension of Sycl and mostly compatible to Sycl. Intel is in the process of building up a complete software ecosystem around OneAPI and supports open-source OneAPI compiler development based on the LLVM platform.
- OpenACC is a standard for compiler pragma's that support offloading of computations to accelerator devices. It supports C, C++ and Fortran. It is mainly pushed by Cray in partnership with Nvidia. Support is contained in recent GCC compilers and it is used at a number of HPC centers.
- OpenMP was originally a standard for easily developing parallel multithreaded applications on CPU platforms. It provides simple compiler pragmas to enable parallel execution of source code segments in C, C++ and Fortran. Since OpenMP 4 it also allows offloading to GPU devices, which has been again improved in OpenMP 5. OpenMP is widely supported by all manufacturers and a good solution for many heterogeneous compute scenarios.

The heterogeneous computing landscape is quickly evolving and the choice of tools is not always easy. If it is known that the application only runs on Nvidia then Cuda is a good choice. However, this makes portability difficult. If a portable solution is desired Sycl is a good future proof choice. Signficant backing by Intel makes sure that Sycl has a good future. Intel's OneAPI is just a variant of Sycl with some extensions, most of which will likely be rolled back into the Sycl standard anyway. So for portability it makes sense to make sure to only use Sycl standard functions. OpenMP 5 is also a viable heterogeneous compute solution. Especially for C and Fortran users this is the most viable solutions. C++ users may prefer Cuda and Sycl as these are based on C++ directly.

Hardware for GPU computing.

Nvidia

All modern Nvida GPUs are well suited for GPU computing. Nvida cards support all major heterogeneous compute standards. The top of the line GPU accelerator from Nvidia is the A100, which has a double precision peak performance of 10 TFlops and a single precision performance of 20 TFlops.

AMD

Most of AMD's compute support comes through the Rocm platform (https://rocmdocs.amd.com/en/latest/). This is supported for cards up to the Vega architecture, but not officially for RDNA, the current generation of accelerators. AMD's data center GPU compute solutions are still based on Vega. AMD announced CDNA, a new architecture for data centers for compute purposes. While AMD is closing up to Nvidia in terms of performance it's compute platform support is much less straight forward than Nvidia.

Intel

Intel traditionally has very good support for OpenCL on its CPU and GPU devices. However, Intel's current GPUs are not competitive for compute applications (though their performance is not bad for certain applications). This is changing with Intel's Xe generation of GPU acceleration that starts coming out in 2020. Intel Xe will power several planned Exascale architectures and most likely well support OpenCL, OpenMP and OneAPI.

Access to GPU Computing in Python

Python has strong support for OpenCL and Cuda. The PyCUDA and PyOpenCL libraries are mature packages that allow to write GPU kernels in C and to launch them from Python. With PyOpenCL and a CPU OpenCL drivers we can also launch compute kernels for CPUs. Since PyOpenCL knows native SIMD types this is a great way to write low-level SIMD accelerated Code for CPUs.

In the following we provide simple examples from the PyCUDA and PyOpenCL documentations for launching kernels with the respective frameworks.

- Cuda

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy

from pycuda.compiler import SourceModule
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
    drv.Out(dest), drv.In(a), drv.In(b),
    block=(400,1,1), grid=(1,1))

print(dest-a*b)
```

- OpenCL

```
#!/usr/bin/env python

import numpy as np
import pyopencl as cl

a_np = np.random.rand(50000).astype(np.float32)
b_np = np.random.rand(50000).astype(np.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

mf = cl.mem_flags
a_g = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a_np)
b_g = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b_np)

prg = cl.Program(ctx, """
__kernel void sum(
__global const float *a_g, __global const float *b_g, __global float *res_g)
{
  int gid = get_global_id(0);
  res_g[gid] = a_g[gid] + b_g[gid];
}
""").build()

res_g = cl.Buffer(ctx, mf.WRITE_ONLY, a_np.nbytes)
prg.sum(queue, a_np.shape, None, a_g, b_g, res_g)

res_np = np.empty_like(a_np)
cl.enqueue_copy(queue, res_np, res_g)

# Check on CPU with Numpy:
print(res_np - (a_np + b_np))
print(np.linalg.norm(res_np - (a_np + b_np)))
assert np.allclose(res_np, a_np + b_np)
```

Numba and Cuda

In this module we will dive into GPU computing using Numba. Numba allows the JIT Compilation of Python code to Cuda devices. It is very easy to use and supports almost all modern Cuda features. A simple taster example is given below. It is a Cuda accelerated Numba version of the Mandelbrot example from https://developer.nvidia.com/blog/numba-python-cuda-acceleration/.

```
%matplotlib inline

import numpy as np
from numba import cuda
from matplotlib.pyplot import imshow


@cuda.jit(device=True)
def mandel(x, y, max_iters):
  """
  Given the real and imaginary parts of a complex number,
  determine if it is a candidate for membership in the Mandelbrot
  set given a fixed number of iterations.
  """
  c = complex(x, y)
  z = 0.0j
  for i in range(max_iters):
    z = z*z + c
    if (z.real*z.real + z.imag*z.imag) >= 4:
      return i

  return max_iters

@cuda.jit
def mandel_kernel(min_x, max_x, min_y, max_y, image, iters):
  height = image.shape[0]
  width = image.shape[1]

  pixel_size_x = (max_x - min_x) / width
  pixel_size_y = (max_y - min_y) / height

  startX = cuda.blockDim.x * cuda.blockIdx.x + cuda.threadIdx.x
  startY = cuda.blockDim.y * cuda.blockIdx.y + cuda.threadIdx.y
  gridX = cuda.gridDim.x * cuda.blockDim.x;
  gridY = cuda.gridDim.y * cuda.blockDim.y;

  for x in range(startX, width, gridX):
    real = min_x + x * pixel_size_x
    for y in range(startY, height, gridY):
      imag = min_y + y * pixel_size_y
      image[y, x] = mandel(real, imag, iters)

gimage = np.zeros((1024, 1536), dtype = np.uint8)
blockdim = (32, 8)
griddim = (32,16)

d_image = cuda.to_device(gimage)
mandel_kernel[griddim, blockdim](-2.0, 1.0, -1.0, 1.0, d_image, 20)
d_image.to_host()

imshow(gimage)
```

By Timo Betcke