Open in app Get started

Aman Singh Follow
Mar 29 · 5 min read · Listen

Save

Migrating from CUDA* to DPC++

In this post, I am going to demonstrate how to port a CUDA kernel to Intel's Data Parallel C++(or DPC++ for short) compiler

Introduction

Let's begin by what is CUDA and then we will move on, CUDA stands for Compute Unified Device Architecture. CUDA is a heterogeneous programming language(i.e. code runs on two different platforms: host's CPU and GPU devices)from NVIDIA that exposes GPU for general purpose program. Simply put, CUDA is Nvidia-owned, parallel computing platform and programming model to run software on GPUs

In this article, I'll be explaining how one might port CUDA code to Intel's oneAPI toolkit. Intel's oneAPI toolkit refer to the DPC++ programming model along with a number of APIs intended to support high-performance computing applications. DPC++ is a compiler built on LLVM's Clang compiler, extending modern C++ capabilities, to allow C++ applications to target heterogeneous systems.

Now, there is a question that needs to be answered, i.e.

Why port CUDA to DPC++

CUDA has widespread usage in the community(research community and otherwise) for machine learning. So, first I need to tell you about some of the benefits of DPC++ programming model to give you a better comparison model for your own reference:

Firstly, DPC++ can work with FPGA accelerator with the same ease as they work with GPUs

Open in app Get started

Finally, and I can't stress this enough is the ability to deploy oneAPI software to the Intel DevCloud, a cloud environment providing CPUs, GPUs, and FPGAs at your disposal.

CUDA Application

Header files to be included

```
#include <cmath>
#include <cstdint>
#include <cstdlib>
#include <iostream>

#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "stb_image_write.h"
```

Since, this is for demonstration purposes , we will be simply porting the venerable Mandelbrot fractal generator as we're more interested in learning the DPC++ programming model itself.

```
struct complex_num
{
    float real;
    float imaginary;
};
// __device__ := Invoke this function from device and execute it on
device

__device__ complex_num operator*(complex a, complex b)
{
    return {a.real * b.real - a.imaginary * b.imaginary, a.real *
b.imaginary + a.imaginary * b.real};
}

__device__ complex_num operator+(complex a, complex b)
{
   return {a.real + b.real, a.imaginary + b.imaginary};
}
```


Open in app Get started

After a brief scan of the above code, We know that it is simply multiplying two complex numbers, add two complex numbers, and compute the squared magnitude of a complex number.

In CUDA, functions we intend on invoking on the accelerator(GPUs) device require the `__device__` attribute. Now, we'll write the function that computes the mandelbrot "value" associated with each pixel:

```
constexpr static uint32_t max_iterations = 12000u;

__device__ uint32_t mandelbrot_pixel(complex c)
{
    // Evaluate iteratively z_{n + 1} = z_n^2 + c
    // Terminate if the max iterations are reached or if the norm
exceeds 2
    complex z = {};

    uint32_t i = 0;
    for (; i != max_iterations; ++i) {
        complex z_next = z * z + c;
        if (sqr_magnitude(z_next) > 4.0) {
            return i;
        } else {
            z = z_next;
        }
    }

    return i;
}
```

This function accepts a constant `c`, initializes a variable `z` to `0`, then continuously evaluates `z_next = z^2 + c; z = z_next` until the magnitude of the `z_next` exceeds the value of `2`. The above function returns the number of iterations needed for this event to occur.

Next, what we need is a kernel function which will evaluate and write out the color of the pixel corresponding to each invocation.


Open in app Get started

```
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x >= width || y >= height)
    {
        return;
    }

    // Remap pixel values to a range from [-2, 1] for the real part
and
    // [-1.5, 1.5] for the imaginary part
    complex c = {static_cast<float>(x) / width * 3.f - 2.f,
    static_cast<float>(y) / height * 3.f - 1.5f};

    // Evaluate the mandelbrot iterations for a single thread and
write out the
    // result after first normalizing to the range [0, 256)
    uint32_t iterations = mandelbrot_pixel(c);

    // Tonemap color
    uint32_t color = iterations * 6;

    // For stylistic reasons, draw saturated values as black
    output[y * width + x] = color >= 256 ? 0 : color;
}
```

In the above function, the `__global__` attribute is used to indicate that it will be invoked in the host. The coordinates of the pixel are used to evaluate a color, which is then written out to the output buffer.

```
int main(int argc, char* argv[])
{
    constexpr static int width          = 512;
    constexpr static int height         = 512;
    constexpr static size_t buffer_size = width * height;

    // Allocate a 512x512 256-bit greyscale image on device
    uint8_t* buffer;
    cudaMalloc(&buffer, buffer_size);

    // Operate with 8x8 workgroup sizes (1 AMD wavefront, 2 NVIDIA
warps)
    dim3 workgroup_dim{8, 8};
    dim3 workgroup_count{width / workgroup_dim.x, height /
workgroup_dim.y};

    mandelbrot<<<workgroup_count, workgroup_dim>>>(buffer, width,
```

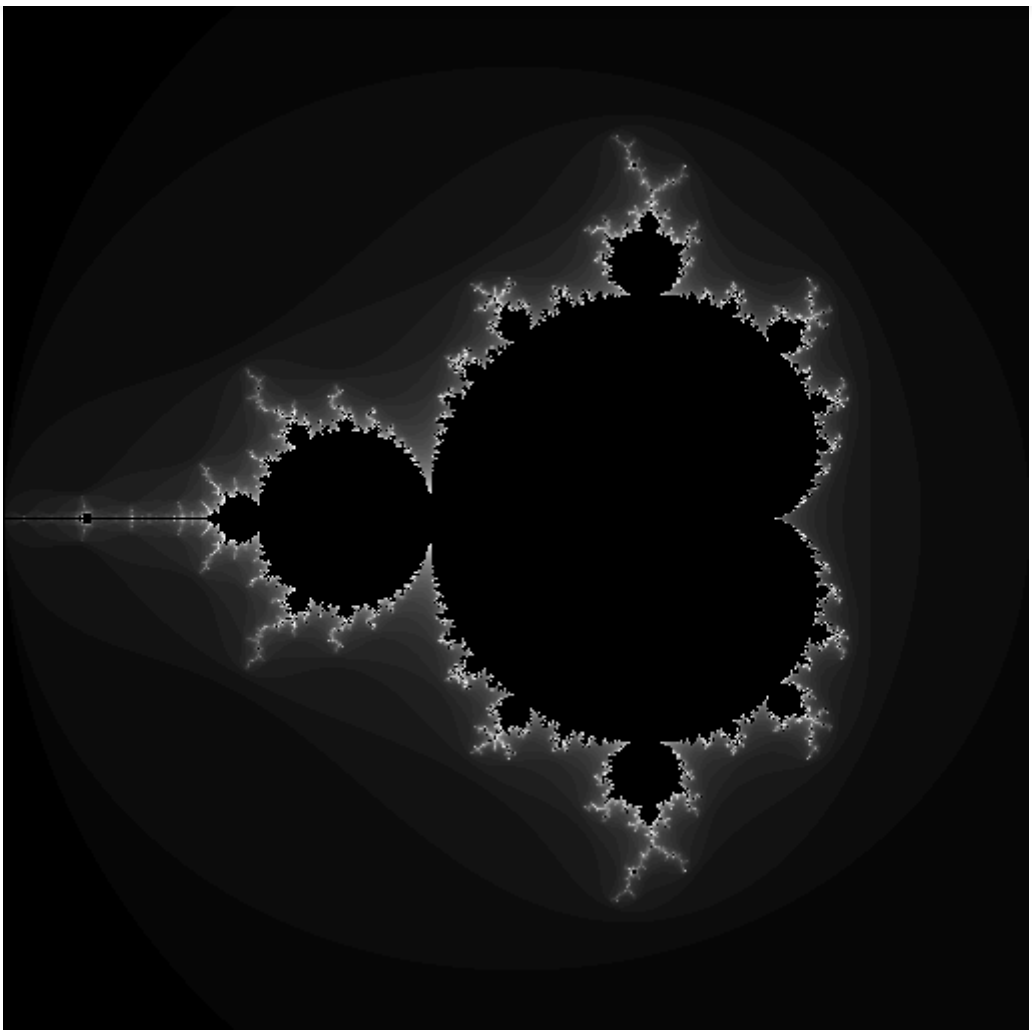

Open in app Get started

```
    // Write back device memory to host memory and deallocate device
memory
    uint8_t* host_buffer = reinterpret_cast<uint8_t*>
(std::malloc(buffer_size));
    cudaMemcpy(host_buffer, buffer, width * height,
cudaMemcpyDeviceToHost);
    cudaFree(buffer);

    // Write out results to an image
    int result = stbi_write_png("mandelbrot.png", width, height, 1,
host_buffer, width);

    std::free(host_buffer);

    return 0;
}
```

Finally, we need a `main` function to allocate device memory to output to, allocate host memory, dispatch our kernel,readback the output, and finally write the output to an image.

For emitting the image, we'll use the source file stb_image_write.h from the venerable stb library collection.


Open in app Get started



Ouput Image

DPC++ comes into picture

To port from CUDA to DPC++, we could either painstakingly “translate” CUDA code to DPC++ or we can utilise the DPC++ Compatibility Tool(provided by Intel) to streamline the porting process.

First, we’ll need to ensure that we have both DPC++ and the compatibility tool installed on your machine. The simplest way to do this is to install the oneAPI toolkits(Both the compiler and compatibility tool are provided in the base toolkit).

Next, after opening a shell window in the OS, we’ll need to invoke a shell script to locally modify various environment variables needed to ensure that the Intel oneAPI libraries and executables are locatable.

On UNIX-like platforms, the script is called **setvars.sh** and is located in the installation

Open in app Get started

After verifying that the `PATH` is set correctly, the dpct compatibility tool should be available. For our simple use case with a single *main.cu* file, the following command is enough to perform the conversion and the output is emitted to the *dpct_output* folder in the same directory.

```
#BAT#
dpct --extra-arg="-std=c++17" main.cu --out-root dpct_output
```

To compile the code and test it, invoke the following command:

```
mkdir build
cd build
dpcpp ../main.dp.cpp -o mandelbrot -lsycl -lOpenCL
```

On Windows, you'll want to emit an executable with the *.exe* extension instead. In the same terminal, executing the mandelbrot program should generate an identical image to what we produced above with CUDA.

Now, the execution of the same program should generate the same/identical image to what we produced on CUDA.

Conclusion

In this article, we've demonstrated how to port an existing CUDA application to DPC++ and how to compile it.


Open in app Get started

Download on the App Store GET IT ON Google Play