Creating teaching materials with xeus-cpp - final presentation

Hristiyan Shterev

Main goals of this project

The main goal of this project was to create interactable examples for the for the CUDA and OpenMP programming models, targeting beginners who want hands-on experience with parallel computing on both the GPU and CPU.

Each notebook builds on the previous one, introducing new concepts gradually through working code examples.

Together the 16 notebooks cover the full beginner to intermediate journey - from launching a first thread to understanding memory hierarchies, synchronisation primitives, and performance optimization on both CPU and GPU.

CUDA notebooks

The CUDA notebooks include 8 different examples that add to one another. Here is what each one explains:

1. The first notebook is a basic introduction to CUDA with simple examples like the __global__ kernel usage and calling it.
2. Next we introduce some more fundamental concepts.
3. The third notebook shows how using parallel programming can speed up if we use the threads the right way.
4. After that we show thread cooperation. The threads split the work instead of having one thread per task
5. Then we demonstrate The Julia set. A complex mathematical shape.
6. The sixth example creates a ripple pattern.
7. Next we demonstrate the dot product. A mathematical operation that takes two equal-length vectors and returns a single regular number
8. Lastly there is a simple ray tracing example.

Notebook examples

Introduction to CUDA

CUDA lets you run code directly on the GPU by writing special functions called **kernels**. This not
to do:

1. Move data onto the GPU
2. Launch a kernel to process it
3. Move the result back to the CPU

Part 1 — Adding two numbers on the GPU

The `__global__` keyword tells the compiler that this function runs on the GPU but is called f

[2]:
```
__global__ void gpu_add(int a, int b, int *result) {
    *result = a + b;
}
```

To call it we use the `<<<blocks, threads>>>` launch syntax — `<<<1,1>>>` means one b
the kernel finishes we pull the value back with `cudaMemcpy`.

[3]:
```
#include <cstdio>

int host_result;
int *dev_result;

cudaMalloc((void**)&dev_result, sizeof(int));

gpu_add<<<1, 1>>>(5, 9, dev_result);

cudaMemcpy(&host_result, dev_result, sizeof(int), cudaMemcpyDeviceToHost);

printf("5 + 9 = %d\n", host_result);
cudaFree(dev_result);
```

```
5 + 9 = 14
```

7. Unified Memory

Normal CUDA requires you to manually allocate on the GPU with `cudaMalloc`, copy data
automatically migrates pages between host and device as needed.

The trade-off: easier to write, but the automatic migration has overhead. For small datasets

[13]:
```
__global__ void double_unified(int* data, int n) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < n)
        data[tid] *= 2;
}
```

[14]:
```
const int K = 5;
int* managed;

// cudaMallocManaged creates memory accessible from both host and device
CUDA_CHECK(cudaMallocManaged(&managed, K * sizeof(int)));

// Write directly on the CPU — no memcpy needed
for (int i = 0; i < K; i++)
    managed[i] = i;

double_unified<<<1, K>>>(managed, K);

// Wait for GPU to finish before reading on CPU
CUDA_CHECK(cudaDeviceSynchronize());

// Read directly on the CPU — no memcpy needed
printf("No cudaMemcpy needed.\n");
for (int i = 0; i < K; i++)
    printf("data[%d] = %d\n", i, managed[i]);

CUDA_CHECK(cudaFree(managed));
```

```
No cudaMemcpy needed.
data[0] = 0
data[1] = 2
data[2] = 4
data[3] = 6
data[4] = 8
```

CUDA benchmark vs the CPU

Here we can see one of the examples in the notebooks. The example adds two vectors with 10 million elements each into a third one. This is done 3 times using different methods.

The first one is a basic CPU only demonstration. The time is around 21 ms.

The second example is using the GPU but with only 1 thread per block. We can see that this is slower compared to the first one. This is a very unoptimised way to use the device.

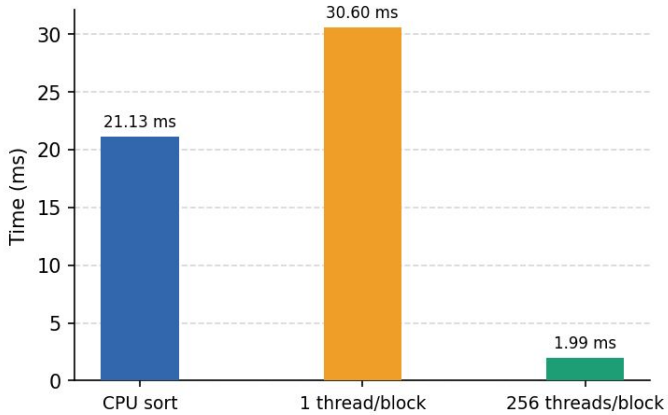
The third method is now a lot faster than the other 2. We make each block use 256 threads which speeds up the time by a lot - around 2 milliseconds.

```
[8]: printf("Time (CPU sort): %f ms\n", duration.count());

     printf("Time (1 thread/block): %f ms\n", time_block_method);

     printf("Time (256 threads/block): %f ms\n", time_thread_method);
```

```
Time (CPU sort): 21.129480 ms
Time (1 thread/block): 30.603071 ms
Time (256 threads/block): 1.990752 ms
```



The Julia Set

One more example worth exploring is the Julia Set - a fractal that reveals the hidden structure of complex number iteration.

The kernel works by assigning each pixel to its own GPU thread. Every thread maps its screen coordinate, then repeatedly applies the formula $Z_{n+1} = Z_n^2 + C$, where C is a fixed constant. If the value escapes a boundary radius of 2 within 256 iterations, the pixel is coloured using a blue-to-white gradient. If it stays bounded, it's coloured black, marking it as outside of the fractal set.

Since every pixel is computed entirely independently, there's no communication or synchronisation needed between threads.

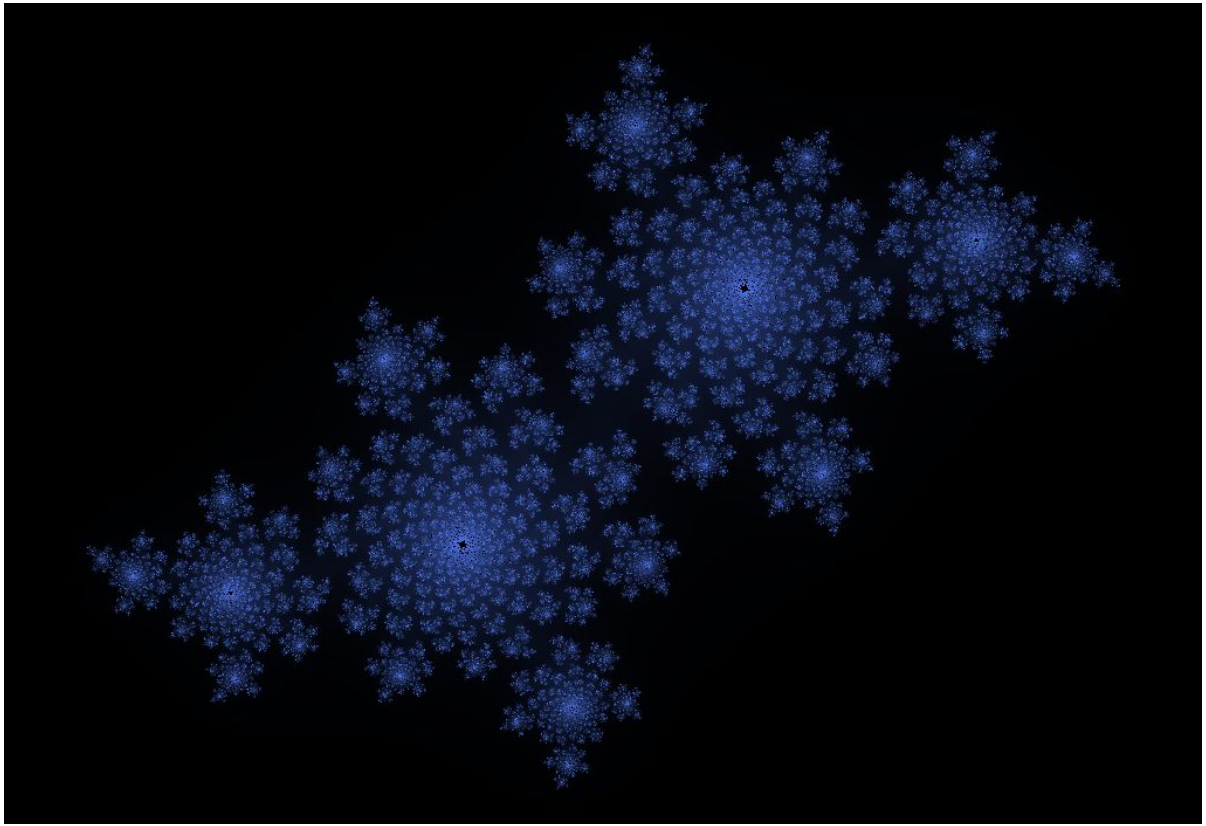
```
// Kernel: one thread per pixel, launched on a 2-D grid
__global__ void julia_fractal(unsigned char *img) {
    int col = threadIdx.x + blockIdx.x * blockDim.x;
    int row = threadIdx.y + blockIdx.y * blockDim.y;
    if (col >= W || row >= H) return;

    // Map pixel to the complex plane  [-1.6, 1.6] × [-1.6, 1.6]
    float cr = 1.6f * (2.f * col - W) / W;
    float ci = 1.6f * (2.f * row - H) / H;

    // Julia constant - gives a dendrite-like fractal
    vec2 c{ -0.4f, 0.6f };
    int n = escape_count(vec2{cr, ci}, c);

    // Colour: points inside the set - black; outside - blue-to-white gradient
    unsigned char v = (n == MAX_ITER) ? 0 : (unsigned char)(255 * n / MAX_ITER);
    int px = (row * W + col) * 4;
    img[px + 0] = v / 3;      // R (keep low for blue tint)
    img[px + 1] = v / 2;      // G
    img[px + 2] = v;           // B (dominant)
    img[px + 3] = 255;         // A
}
```

The Julia Set



OpenMP notebooks

The OpenMP notebooks also include 8 different examples that add to one another. Here is what each one explains:

1. The first notebook is a basic introduction to OpenMP with simple examples like the #pragma omp parallel directive and thread creation.
2. Next we introduce the fork-join model and how threads are spawned and joined back together.
3. Then we demonstrate the Pi integral. A mathematical problem solved by splitting the work across multiple threads.
4. The fourth notebook calculates the area of the Mandelbrot set. A complex mathematical shape rendered in parallel by assigning different regions to different threads.
5. After that we show linked list traversal. How pointer-based data structures interact with parallel execution.
6. The sixth example demonstrates race conditions. What happens when threads write to the same memory without protection and how to fix it.
7. Next we demonstrate false sharing. How threads can slow each other down even when touching different variables due to CPU cache line behaviour.
8. Lastly there is Conway's Game of Life. A grid simulation where threads compute the next generation in parallel using double buffering to avoid race conditions by design.

Notebook examples

[4]:
```
for (i = 1; i <= 4; i++) {
    sum = 0.0;
    omp_set_num_threads(i);
    start_time = omp_get_wtime();
    #pragma omp parallel
    {
        double x;
        num_threads_allocated = omp_get_num_threads();

        #pragma omp single
            printf("Num threads allocated for this run: %d\n", num_threads_allocated);

        #pragma omp for reduction(+ : sum)
            for (j = 1; j <= num_steps; j++) {
                x = (j - 0.5) * step;
                sum = sum + 4.0 / (1.0 + x * x);
            }
    }
    pi = step * sum;
    run_time = omp_get_wtime() - start_time;
    printf("pi is %f in %f seconds using %d threads\n\n", pi, run_time, num_threads_allocated);
}
```

```
Num threads allocated for this run: 1
pi is 3.141593 in 1.106737 seconds using 1 threads

Num threads allocated for this run: 2
pi is 3.141593 in 0.551873 seconds using 2 threads

Num threads allocated for this run: 3
pi is 3.141593 in 0.368655 seconds using 3 threads

Num threads allocated for this run: 4
pi is 3.141593 in 0.289997 seconds using 4 threads
```

[1]:
```
#include <iostream>
#include <omp.h>
```

[2]:
```
void example1() {
  std::cout << "Hello World!" << std::endl;
}
example1();
```

```
Hello World!
```

Now, let's use OpenMP to run the same code in parallel. By a

`#pragma omp parallel` This directive tells the compiler

The result is that instead of printing once, you will see "Hello

[3]:
```
void example2() {
    #pragma omp parallel
    {
    std::cout << "Hello World!" << std::endl;
    }
}
example2();
```

```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!Hello World!
```

False sharing: benchmark

False sharing - 0.2729s Threads write to different counters but all 4 fit in one 64-byte cache line. Every write forces all other cores to reload the entire line. The threads spend more time fighting over cache ownership than actually counting.

Padded - 0.0315s - 8.7x faster Each counter is padded to 64 bytes so it occupies its own cache line. Threads now write completely independently with zero interference. The 8.7x speedup comes from one struct change - no algorithm changes, no extra threads.

Private vars - 0.0425s - 6.4x faster Each thread counts in its own stack variable with no sharing at all. Results are merged at the end with a reduction. Slightly slower than padded because the merge step has a small synchronisation cost.

Method	Time	Speedup vs FS
False sharing:	0.2729 s	1.0x
Padded:	0.0315 s	8.7x
Private vars:	0.0425 s	6.4x

Thank you!