PHY 250

PHY 250 Introduction to General Purpose GPU Computation
Winter 2024 W 13:10-15:00
Instructor: Andrew Diggs
email: amdiggs@ucdavis.edu
office: phy 404

About me

Andrew Diggs (he, him, his)

Fifth year physics graduate student.

Field of study is condensed matter theory with a focus on the solar cell performance and degradation.

Using GPU computation for about two years. I began by learning OpenGL, computer graphics, which lead me to general purpose GPU computation with OpenCL.

Before college I worked as a cowboy.

2011 Horse lake crew: Derick Murrer, Ellie Ward, Andrew Diggs, and Dave Ward

About the classroom

I want everyone to have fun and learn this quarter.

This classroom **will be safe and welcoming for everyone.**

Any behavior that makes anyone feel uncomfortable will not be tolerated, and this absolutely includes me.

About the course

This course is not intended to be a full course on heterogeneous computing. The goal of this course is to provide the functional background needed to incorporate GPGPU computation into our projects.

There will be weekly homework assignments much of which will be done in class.

The final will consist of a project and presentation. The project can be anything that you find interesting that use GPGPU computation.

This course is designed to focus more on learning outcome rather than proficiency, and thus grading will reflect these principles.

This course will broken down into three main sections.

1. C/C++ fundamentals.
2. Host-device communication, talking to the GPU, using OpenCL.
3. Write code for the GPU using the OpenCL kernel language.

Why do we want to use a GPU

Python: Ultimate luxury
Very user friendly and performs very well for most tasks.

C/C++: Ultimate performance
The C language is much less user friendly but in return you get up to 300X faster than native python.

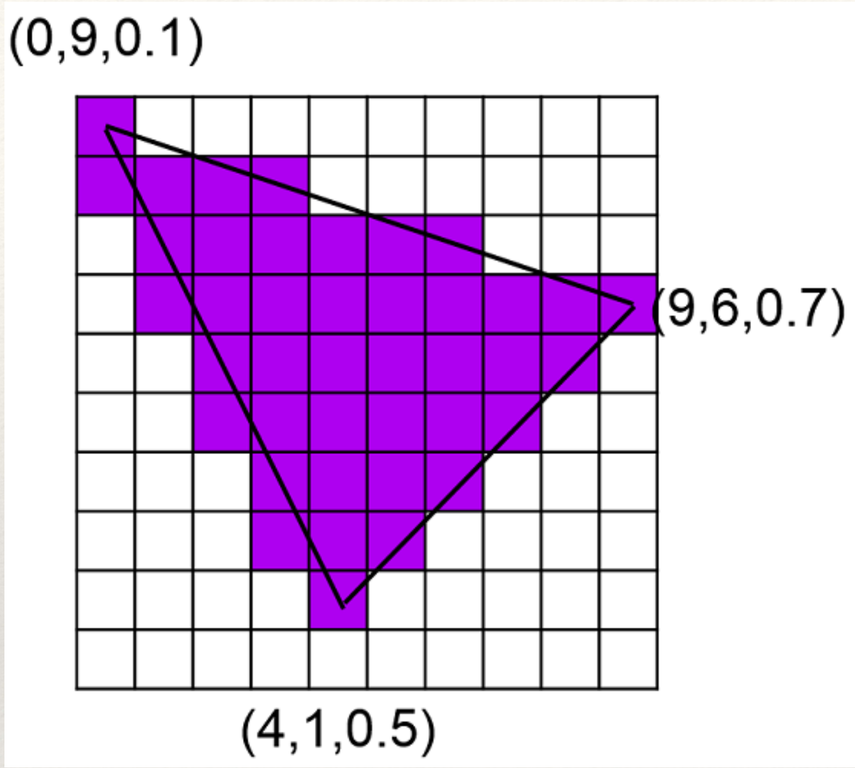
OpenCl/CUDA: Ultimate speed
Less user friendly C and can only perform simple tasks. However, when used properly can achieve performance up to 4000X faster than C.

General Purpose GPU Computation

GPUs are designed to do one thing… render graphics.

Graphics are rendered using a method called scan-line polygon filling.

Let's discuss how we might develop an algorithm to do this.

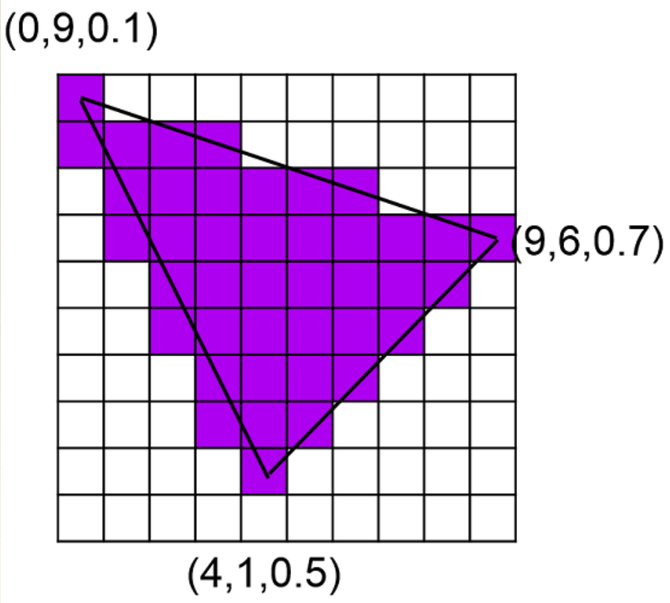


General Purpose GPU Computation

A 4 K monitor has about 8.3 M pixels.

To rasterize even a single triangle with serial execution for this many pixels would take about 20 ms on a modern cpu.

To raster and render the extremely complicated graphics of the games we love requires Massively Multi-Threaded (MMT) execution.

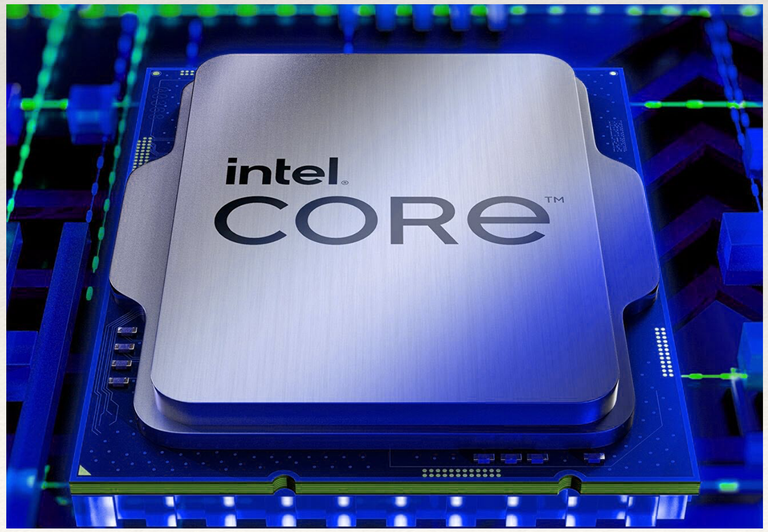


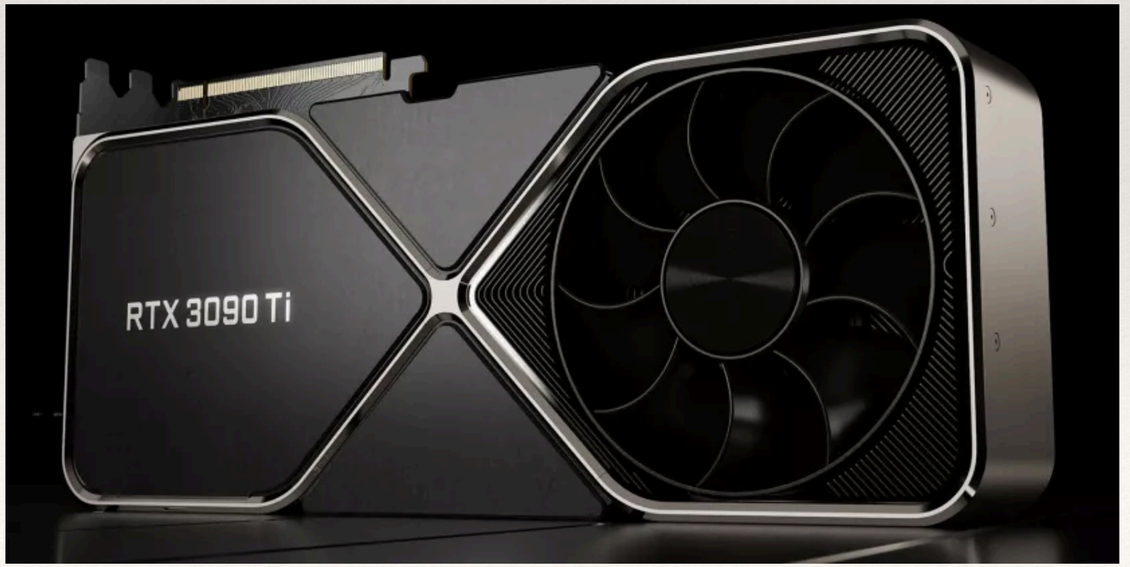
CPU vs GPU

CPU can handle very long complicated instruction sets with direct user I/O and memory management. Multi-core CPUs can perform multi-threaded/ multi -process execution; however, even the best CPUs are limited to ~ 112 threads and low memory bandwidth typically bottlenecks performance at much lower thread counts.

GPUs are limited to much more simple instruction sets with no, or very little, direct user I/O and no memory management. However, The NVIDIA A100 has 6912 cores, threads, for parallel execution.

GPUs can perform a very large number of independent simple tasks each cycle.





General Purpose GPU Computation

GPUs can perform a very large number of independent simple tasks in parallel execution.

Where else does this show up in programming?

```
int main(int argc, const char * argv[]) {

    for(int i = 0; i < num_el; i++){
        Do_Something(My_Array[i]);
    }


    return 0;
}
```

```
__kernel void GPU_Loop(__global float* My_Array, const int num_el)
{

    int g_id = get_global_id(0);
    // Here we use the thread id just like i in the for loop

    Do_Something(My_Array[g_id]);

}
```

We can take a serial process executed via a for loop and replace it with a parallel process executed on the GPU. This is the essence of GPGPU computation.

But first let's hone our skills

The OpenCL and CUDA kernel language are nearly identical, and/because they are based on the Graphics Language Shader Language (GLSL).

GLSL primarily follows C 98 syntax; however, there are a few unique aspects. As mentioned before there is very little or no direct user I/O in GPU programming. This means that there is no debugging support and in many cases there is not even print to output. So before we jump into GPGPU programming let's hone our C.

C/C++: Copies, References, and Pointers

```
void My_Func( float cp, float& ref, float* pt){
    cp = 100.0;
    ref = 100.0;
    pt = 100.0;

}
```

```
int main(int argc, const char * argv[]) {

    float a = 0.0;
    float b = 0.0;
    float c = 0.0;
    printf("a = %.1f\n",a);
    printf("b = %.1f\n",b);
    printf("c = %.1f\n",c);

    My_Func(a,b,&c);

    printf("a = %.1f\n",a);
    printf("b = %.1f\n",b);
    printf("c = %.1f\n",c);
    return 0;
}
```

What will our output be?

Error!!!!

```
void My_Func( float cp, float& ref, float* pt){
    cp = 100.0;
    ref = 100.0;
    pt = 100.0;

}
```

Assigning to 'float *' from incompatible type 'double'

A pointer is an address. Think of it like a post office box. If we want a float we need to get inside first.

C/C++: Copies, References, and Pointers

```
void My_Func( float cp, float& ref, float* pt){
    cp = 100.0;
    ref = 100.0;
    *pt = 100.0;

}
```

```
int main(int argc, const char * argv[]) {

    float a = 0.0;
    float b = 0.0;
    float c = 0.0;
    printf("a = %.1f\n",a);
    printf("b = %.1f\n",b);
    printf("c = %.1f\n",c);

    My_Func(a,b,&c);

    printf("a = %.1f\n",a);
    printf("b = %.1f\n",b);
    printf("c = %.1f\n",c);
    return 0;
}
```

Now what will our output be?

Listing 1.1 Creating and distributing a matrix-vector multiplication kernel: matvec.c

```
#define PROGRAM_FILE "matvec.cl"
#define KERNEL_FUNC "matvec_mult"

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

#ifdef MAC
#include <OpenCL/cl.h>
#else
#include <CL/cl.h>
#endif

int main() {
   cl_platform_id platform;
   cl_device_id device;
   cl_context context;
   cl_command_queue queue;
   cl_int i, err;

   cl_program program;
   FILE *program_handle;
   char *program_buffer, *program_log;
   size_t program_size, log_size;
   cl_kernel kernel;
   size_t work_units_per_kernel;

   float mat[16], vec[4], result[4];
   float correct[4] = {0.0f, 0.0f, 0.0f, 0.0f};
   cl_mem mat_buff, vec_buff, res_buff;

   for(i=0; i<16; i++) {
      mat[i] = i * 2.0f;
   }
```

Initialize data


```
   for(i=0; i<4; i++) {
      vec[i] = i * 3.0f;
      correct[0] += mat[i]    * vec[i];
      correct[1] += mat[i+4]  * vec[i];
      correct[2] += mat[i+8]  * vec[i];
      correct[3] += mat[i+12] * vec[i];
   }

   clGetPlatformIDs(1, &platform, NULL);
   clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1,
      &device, NULL);
   context = clCreateContext(NULL, 1, &device, NULL,
      NULL, &err);
```

Initialize data

Set platform/ device/context

```
   program_handle = fopen(PROGRAM_FILE, "r");
   fseek(program_handle, 0, SEEK_END);
   program_size = ftell(program_handle);
   rewind(program_handle);
   program_buffer = (char*)malloc(program_size + 1);
   program_buffer[program_size] = '\0';
   fread(program_buffer, sizeof(char), program_size,
      program_handle);
   fclose(program_handle);

   program = clCreateProgramWithSource(context, 1,
      (const char**)&program_buffer, &program_size, &err);
   free(program_buffer);
   clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

   kernel = clCreateKernel(program, KERNEL_FUNC, &err);
   queue = clCreateCommandQueue(context, device, 0, &err);

   mat_buff = clCreateBuffer(context, CL_MEM_READ_ONLY |
      CL_MEM_COPY_HOST_PTR, sizeof(float)*16, mat, &err);
   vec_buff = clCreateBuffer(context, CL_MEM_READ_ONLY |
      CL_MEM_COPY_HOST_PTR, sizeof(float)*4, vec, &err);
   res_buff = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
      sizeof(float)*4, NULL, &err);
   clSetKernelArg(kernel, 0, sizeof(cl_mem), &mat_buff);
   clSetKernelArg(kernel, 1, sizeof(cl_mem), &vec_buff);
   clSetKernelArg(kernel, 2, sizeof(cl_mem), &res_buff);

   work_units_per_kernel = 4;
   clEnqueueNDRangeKernel(queue, kernel, 1, NULL,
      &work_units_per_kernel, NULL, 0, NULL, NULL);

   clEnqueueReadBuffer(queue, res_buff, CL_TRUE, 0,
      sizeof(float)*4, result, 0, NULL, NULL);
   if((result[0] == correct[0]) && (result[1] == correct[1])
      && (result[2] == correct[2]) && (result[3] == correct[3])) {
      printf("Matrix-vector multiplication successful.\n");
   }
   else {
      printf("Matrix-vector multiplication unsuccessful.\n");
   }
```

Read program file

Compile program

Create kernel/queue

Set kernel arguments

Execute kernel

```
   clReleaseMemObject(res_buff);
   clReleaseKernel(kernel);
   clReleaseCommandQueue(queue);
   clReleaseProgram(program);
   clReleaseContext(context);

   return 0;
}
```

This source file is long but straightforward. Most of the code is devoted to c

```
for(i=0; i<4; i++) {
   vec[i] = i * 3.0f;
   correct[0] += mat[i]    * vec[i];
   correct[1] += mat[i+4]  * vec[i];
   correct[2] += mat[i+8]  * vec[i];
   correct[3] += mat[i+12] * vec[i];
}
```

Initialize data

```
clGetPlatformIDs(1, &platform, NULL);
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1,
   &device, NULL);
context = clCreateContext(NULL, 1, &device, NULL,
   NULL, &err);
```

Set platform/ device/context

```
program_handle = fopen(PROGRAM_FILE, "r");
fseek(program_handle, 0, SEEK_END);
program_size = ftell(program_handle);
rewind(program_handle);
program_buffer = (char*)malloc(program_size + 1);
program_buffer[program_size] = '\0';
fread(program_buffer, sizeof(char), program_size,
   program_handle);
fclose(program_handle);
```

Read program file

```
program = clCreateProgramWithSource(context, 1,
   (const char**)&program_buffer, &program_size, &err);
free(program_buffer);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
```

Compile program

```
kernel = clCreateKernel(program, KERNEL_FUNC, &err);
queue = clCreateCommandQueue(context, device, 0, &err);
```

Create kernel/queue

```
mat_buff = clCreateBuffer(context, CL_MEM_READ_ONLY |
   CL_MEM_COPY_HOST_PTR, sizeof(float)*16, mat, &err);
vec_buff = clCreateBuffer(context, CL_MEM_READ_ONLY |
   CL_MEM_COPY_HOST_PTR, sizeof(float)*4, vec, &err);
res_buff = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
   sizeof(float)*4, NULL, &err);
clSetKernelArg(kernel, 0, sizeof(cl_mem), &mat_buff);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &vec_buff);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &res_buff);
```

Set kernel arguments

```
work_units_per_kernel = 4;
clEnqueueNDRangeKernel(queue, kernel, 1, NULL,
   &work_units_per_kernel, NULL, 0, NULL, NULL);
```

Execute kernel

```
clEnqueueReadBuffer(queue, res_buff, CL_TRUE, 0,
   sizeof(float)*4, result, 0, NULL, NULL);
if((result[0] == correct[0]) && (result[1] == correct[1])
   && (result[2] == correct[2]) && (result[3] == correct[3])) {
   printf("Matrix-vector multiplication successful.\n");
}
else {
   printf("Matrix-vector multiplication unsuccessful.\n");
}

clReleaseMemObject(mat_buff);
clReleaseMemObject(vec_buff);
```

```
    clReleaseMemObject(res_buff);
    clReleaseKernel(kernel);
    clReleaseCommandQueue(queue);
    clReleaseProgram(program);
    clReleaseContext(context);

    return 0;
}
```

This source file is long but straightforward. Most of the code is devoted to c