Data Parallelism

CSCI 5828: Foundations of Software Engineering
Lecture 26 — 11/20/2014
Goals

• Cover the material in Chapter 7 of the Concurrency Textbook
  • Data Parallelism
    • Using OpenCL to create parallel programs on the GPU
    • Also look at an example that uses OpenCL and OpenGL together
Data Parallelism

• Data Parallelism is an approach to concurrent programming that can perform number crunching on a computer’s GPU
  
  • You typically create a bunch of arrays and load them onto the GPU
  
  • You also create a “kernel”; a program that is loaded onto the GPU
  
  • The kernel then applies the operations of its program to the elements of the array in parallel.
    
    • So, if you had two arrays of 1024 floating point numbers and you wanted to multiply those arrays together, the GPU might be able to perform that multiplication on all 1024 pairs “at once”
      
      • In actuality, the kernel is transformed into a program for your particular GPU that makes use of pipelining and multiple ALUs to process the operation as efficiently as possible
Pipelining (I)

- One way that data parallelism can be implemented is via pipelining
  - int result = 13 * 2;
- We might think of the above multiplication as an atomic operation
  - but on the CPU, at the level of “gates”, it takes several steps
- Those steps are typically arranged as a pipeline

  ![Diagram of pipelining](image)

- In this example, if each step takes one clock cycle, it would take 5 clock cycles to perform the multiplication
Pipelining (II)

- If we only sent two numbers to this pipeline, our code would be inefficient.
- Instead, if we have lots of numbers to multiply, we can insert new numbers to multiply at the start of the pipeline for each new clock cycle.

Here, arrays A and B are being multiplied and we’re inserting one new element of each array into the start of the pipeline, each clock cycle.

- 1000 numbers would take ~1000 clock cycles to multiply (rather than 5000)!
ALUs (I)

- Arithmetic logic units (ALUs) perform computations for CPUs
  - They take two inputs and perform a result
- Another way to perform parallel operations on a GPU is to:
  - create a series of linked ALUs that compute a result
  - and combine it with a “wide bus” that allows multiple operands to be fetched at the same time

![Diagram of ALU operation]

So multiplying a thousand pairs of numbers takes a whisker over a thousand clock cycles, not the five thousand we might expect from the fact that multiplying a single pair takes five clock cycles.

Multiple ALUs

The component within a CPU that performs operations such as multiplication is commonly known as the arithmetic logic unit, or ALU:

```
Operand 1  Operand 2  Result
```

Couple multiple ALUs with a wide memory bus that allows multiple operands to be fetched simultaneously, and operations on large amounts of data can again be parallelized, as shown in Figure 12, Large Amounts of Data Parallelized with Multiple ALUs, on page 192.

GPU s typically have a 256-bit or wider memory bus, allowing (for example) eight or more 32-bit floating-point numbers to be fetched at a time.

A Confused Picture

To achieve their performance, real-world GPU s combine pipelining and multiple ALUs with a wide range of other techniques that we’ll not cover here. By itself, this would make understanding the details of a single GPU complex. Unfortunately, there’s little commonality between different GPU s (even those produced by a single manufacturer). If we had to write code that directly targeted a particular architecture, GPGPU programming would be a nonstarter.

OpenCL targets multiple architectures by defining a C-like language that allows us to express a parallel algorithm abstractly. Each different GPU reports erratum • discuss

Day 1: GPGPU Programming

© Kenneth M. Anderson, 2014
Our First OpenCL Program

To parallelize our array multiplication task with OpenCL, we need to divide it up into work-items that will then be executed in parallel.

Work-Items

If you're used to writing parallel code, you will be used to worrying about the granularity of each parallel task. Typically, if each task performs too little work, your code performs badly because thread creation and communication overheads dominate.

OpenCL work-items, by contrast, are typically very small. To multiply two 1,024-element arrays pairwise, for example, we could create 1,024 work-items (see Figure 13, Work Items for Pairwise Multiplication, on page 193).

Your task as a programmer is to divide your problem into the smallest work-items you can. The OpenCL compiler and runtime then worry about how best to schedule those work-items on the available hardware so that that hardware is utilized as efficiently as possible.
The Basics of OpenCL (I)

• The key to any OpenCL program is defining its work items
  • In traditional approaches to concurrency, you have to
    • create a bunch of threads and make sure that there is enough work to do so that you fully utilize all of the cores on your system
    • otherwise, you end up with poor performance as thread overhead dominates your program’s run-time as opposed to actual work
  • With OpenCL programs, the goal is to create lots of small work items
    • The smaller the better, so that OpenCL can make use of pipelining and multiple ACLs to maximally distribute the computations across all of the cores in the GPU
The Basics of OpenCL (II)

• “multiplying two arrays” example
  • create one work item for each multiplication
  • depending on the structure of our GPU, we might be able to perform all of the multiplications with one instruction
    • if not, we’d do as many multiplications as possible at once and then load the rest
      • since it’s being done in parallel, the number of instructions will be drastically less than doing the same amount of work on the CPU

![Diagram of work items for pairwise multiplication](image)

Optimizing OpenCL

You won’t be surprised to hear that the real-world picture isn’t quite this simple.

Optimizing an OpenCL application often involves thinking carefully about the available resources and providing hints to the compiler and runtime to help them schedule your work-items. Sometimes this includes restricting the available parallelism for efficiency purposes.

As always, however, premature optimization is the root of all programming evil. In the majority of cases, you should aim for maximum parallelism and the smallest possible work-items and worry about optimization only afterward.

Kernels

We specify how each work-item should be processed by writing an OpenCL kernel. Here’s a kernel that we could use to implement the preceding:

```cl
__kernel void multiply_arrays(__global const float* inputA, __global const float* inputB, __global float* output)
{
    int i = get_global_id(0);
    output[i] = inputA[i] * inputB[i];
}
```

This kernel takes pointers to two input arrays, `inputA` and `inputB`, and an output array, `output`. It calls `get_global_id()` to determine which work-item it’s handling and then simply writes the result of multiplying the corresponding elements of `inputA` and `inputB` to the appropriate element of `output`.

To create a complete program, we need to embed our kernel in a host program that performs the following steps:
The Basics of OpenCL (III): Find the Platform

- OpenCL programs have a basic structure
  - You first ask if a platform is available
    - In C
      - `cl_platform_id platform;`
      - `clGetPlatformIDs(1, &platform, NULL);`
    - In Java
      - `CL.create();`
      - `CLPlatform platform = CLPlatform.getPlatforms().get(0);`
  - Getting access to the platform object, allows you to look for devices
Once you have the platform, you can locate a device to use

In C

• `cl_device_id device;`
• `clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);`

In Java

• `List<CLDevice> devices = platform.getDevices(CL_DEVICE_TYPE_TYPE_GPU);`

Here, we are specifically asking for GPUs, but there can be more than one type of device and we could ask for all of them (if needed)

• devices can also include the CPU and specialized OpenCL accelerators
The Basics of OpenCL (V): Get a Context

- Once you have a device (or devices), you can create a context for execution
  - In C
    - cl_context context =
      - clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
  - In Java
    - CLContext context =
      - CLContext.create(platform, devices, null, null, null);
- Contexts can be used to pull in other information for OpenCL, such as OpenGL drawing environments (as we will see in a later example)
  - but the primary use of a context is to create a queue for processing work items
The Basics of OpenCL (VI): Create a Queue

• Now we are ready to create a command queue
  
  • In C
    
    • `cl_command_queue queue =` 
      
      `clCreateCommandQueue(context, device, 0, NULL);`
  
  • In Java
    
    • `CLCommandQueue queue =`
    
    • `clCreateCommandQueue(context, devices.get(0), 0, null);`
  
• We now have the ability to send commands to the device and get it to perform work for us
  
• I’m now going to switch to showing an example in C, we’ll return to the Java example later
The Basics of OpenCL (VII): Compile a Kernel

- OpenCL defines a C-like language that allows work-items to be specified
  - Programs written in this language are called kernels
  - Before we can use our queue, we need to compile the kernel. It is this step that creates a program that works with your specific GPU
    - char* source = read_source("multiply_arrays.cl");
    - cl_program program =
      - clCreateProgramWithSource(
        - context, 1, (const char**)&source, NULL, NULL);
    - free(source);
    - clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
    - cl_kernel kernel = clCreateKernel(program, "multiply_arrays", NULL);
  - At the end of this step, we have our kernel in memory, ready to execute
    - I’ll show you the “multiply_arrays.cl” code in a moment
The Basics of OpenCL (VIII): Create Buffers

• Kernels are typically passed buffers (i.e. arrays) of data that they operate on
  • #define NUM_ELEMENTS 1024
  • cl_float a[NUM_ELEMENTS], b[NUM_ELEMENTS];
  • random_fill(a, NUM_ELEMENTS);
  • random_fill(b, NUM_ELEMENTS);
  • cl_mem inputA = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(cl_float) * NUM_ELEMENTS, a, NULL);
  • cl_mem inputB = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(cl_float) * NUM_ELEMENTS, b, NULL);
  • cl_mem output = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(cl_float) * NUM_ELEMENTS, NULL, NULL);

• Here we create two C arrays, fill them with random numbers, and then copy them into two OpenCL buffers. We also create a buffer to store the output
The Basics of OpenCL (IX): Perform the Work

- Now, we need to pass the buffers to the kernel
  - `clSetKernelArg(kernel, 0, sizeof(cl_mem), &inputA);`
  - `clSetKernelArg(kernel, 1, sizeof(cl_mem), &inputB);`
  - `clSetKernelArg(kernel, 2, sizeof(cl_mem), &output);`

- Perform the work
  - `size_t work_units = NUM_ELEMENTS;`
  - `clEnqueueNDRangeKernel(
      queue, kernel, 1, NULL, &work_units, NULL, 0, NULL, NULL);`

- Retrieve the results
  - `cl_float results[NUM_ELEMENTS];`
  - `clEnqueueReadBuffer(queue, output, CL_TRUE, 0, sizeof(cl_float) * NUM_ELEMENTS, results, 0, NULL, NULL);`
The Basics of OpenCL (X): Clean Up

• Finally, we have to deallocate all OpenCL-related data structures
  • clReleaseMemObject(inputA);
  • clReleaseMemObject(inputB);
  • clReleaseMemObject(output);
  • clReleaseKernel(kernel);
  • clReleaseProgram(program);
  • clReleaseCommandQueue(queue);
  • clReleaseContext(context);

• Now, we’re ready to look at a kernel
Our First Kernel: Multiply Arrays (I)

• Here’s a small kernel, written in OpenCL’s C-like language

```c
__kernel void multiply_arrays(
    __global const float* inputA,
    __global const float* inputB,
    __global float* output) {
    int i = get_global_id(0);
    output[i] = inputA[i] * inputB[i];
}
```

• This is NOT C, it’s just designed to look like C.

• The OpenCL compiler can take this program and generate machine code to run on a particular device (in this case a GPU) in a massively parallel fashion
Our First Kernel: Multiply Arrays (II)

```c
__kernel void multiply_arrays(
  __global const float* inputA,
  __global const float* inputB,
  __global float* output) {
  int i = get_global_id(0);
  output[i] = inputA[i] * inputB[i];
}
```

- We see that this kernel expects three inputs: each an array of floats
  - Our call to `clSetKernelArg()` on slide 16 assigns our buffers to these args
  - All three of these arrays are stored in the GPU’s global memory
- To perform work, we find out what work item we are `(get_global_id())`
  - We use that id to index into the arrays
- OpenCL will try to complete as many work items in parallel as it can
The single threaded version?

• How do we decide if all this work is worth it
  
  • Let’s compare the GPU version of the program with the single-threaded version that runs on the CPU. Here it is
    
    • for (int i = 0; i < NUM_ELEMENTS; ++i) {
        • results[i] = a[i] * b[i];
        • }
    
    • (This code reuses the definitions of a, b, and results seen previously)

• The book shows how to profile OpenCL code (see Chapter 7 for details)
The results?

• Multiply two arrays of 200,000 random float values

• Total (GPU): 1717845 ns
  • Elapsed (GPU): 86000 ns \(\leq\) how long did the multiplications take?

• Elapsed (CPU): 886752 ns \(\leq\) single-threaded version

• The GPU version is ten times faster
  • Finishing in .08 milliseconds

• The CPU finished in 0.8 milliseconds

• Worth it? Not for this simple program. But, in general, YES!
Working in Multiple Dimensions

• Our first example showed how to operate on buffers that were the same length and that had a single index to organize the work.
  • We can work with multiple dimensions (such as multiplying matrices) by employing a common trick.
    • We store a multidimensional matrix into a linear array.
      • If we have a 2x2 matrix, we can store its values in a 4-element array.
    • We then use a “width” parameter and x,y coordinates to calculate where in the linear array a particular value is stored. Rather than
      • \(a[x][y] = 10\)
  • We write
    • \(a[x \times \text{width} + y] = 10\)
Our Second Kernel: Matrix Multiplication

Here's code that implements this sequentially:

```c
#define WIDTH_OUTPUT WIDTH_B
#define HEIGHT_OUTPUT HEIGHT_A
float a[HEIGHT_A][WIDTH_A] = «initialize a»;
float b[HEIGHT_B][WIDTH_B] = «initialize b»;
float r[HEIGHT_OUTPUT][WIDTH_OUTPUT];
for (int j = 0; j < HEIGHT_OUTPUT; ++j)
    for (int i = 0; i < WIDTH_OUTPUT; ++i)
        float sum = 0.0;
        for (int k = 0; k < WIDTH_A; ++k)
            sum += a[j][k] * b[k][i];
        r[j][i] = sum;
As you can see, as the number of elements in our array increases, the work required to multiply them increases dramatically, making large-matrix multiplication a very CPU-intensive task indeed.

Parallel Matrix Multiplication

Here's a kernel that can be used to multiply two-dimensional matrices:

DataParallelism/MatrixMultiplication/matrix_multiplication.cl

```c
__kernel void matrix_multiplication(uint widthA, __global const float* inputA, __global const float* inputB, __global float* output) {

    int i = get_global_id(0);
    int j = get_global_id(1);

    int outputWidth = get_global_size(0);
    int outputHeight = get_global_size(1);
    int widthB = outputWidth;

    float total = 0.0;
    for (int k = 0; k < widthA; ++k) {
        total += inputA[j * widthA + k] * inputB[k * widthB + i];
    }
    output[j * outputWidth + i] = total;
}

Our work items have been configured to have two dimensions.
The size of the output matrix was set as a global variable.
Configuring OpenCL to Work with this Kernel

```c
size_t work_units[] = {WIDTH_OUTPUT, HEIGHT_OUTPUT};
CHECK_STATUS(clEnqueueNDRangeKernel(queue, kernel, 2, NULL, work_units, 
    NULL, 0, NULL, NULL));
```

To ensure our kernel has the information it needs, we have to change how we add work items to the queue

The “2” tells OpenCL that the work items have two dimensions

The work_units array tells OpenCL the range of the two dimensions
The results?

• The book multiplies a 200x400 matrix (of random floating point values) by a 300x200 matrix producing a 300x400 matrix as a result

• Total (GPU): 4899413 ns
  • Elapsed (GPU): 3840000 ns <= 78% of the time spent multiplying
  • Elapsed (CPU): 65459804 ns <= single-threaded version

• The GPU version is 17 times faster!
  • Finishing in 3.84 milliseconds
• The CPU finished in 65.5 milliseconds

• Worth it? For a program that has to do a lot of these multiplications?
  • YOU BET!
OpenCL and OpenGL: Match Made in Heaven?

• One use of OpenCL code is to work with OpenGL to perform graphics-related calculations on the GPU, freeing up the CPU to perform other operations
  • OpenGL is a standard for creating 3D programs/animations

• Our book presents two OpenGL applications that make use of OpenCL to perform operations on triangles
  • In the first example, we create a “mesh” of triangles and use OpenGL to display them
    • we then send the vertices of the mesh to an OpenCL program that multiplies there values by 1.01 increasing the spacing of the vertices by 1% on each iteration
    • This has the visual effect of zooming in on the mesh
Background (I)

• The mesh of triangles can be conceptualized like this

![Diagram of a triangle mesh]

There’s an index buffer that captures the existence of each index: 0, 1, 2

There’s also a vertex buffer that captures the position of each index:

Index 1 => {0, 0, 0} i.e. x, y, z

```java
int vertexBuffer = glGenBuffers();
glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
BufferData(GL_ARRAY_BUFFER, mesh.vertices, GL_DYNAMIC_DRAW);

int indexBuffer = glGenBuffers();
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer);
BufferData(GL_ELEMENT_ARRAY_BUFFER, mesh.indices, GL_STATIC_DRAW);
```

When we create the vertex buffer, we tell OpenGL that it will change
Background (II)

- Our kernel for zooming is simple

```c
__kernel void zoom(__global float* vertices) {
    unsigned int id = get_global_id(0);
    vertices[id] *= 1.01;
}
```

- get each value, (x, y, z), and increase its size by 1%
Background (III)

• The key to making this work is to then associate an OpenGL buffer with an OpenCL buffer
  - In a loop do the following
    • draw the OpenGL buffer
    • associate the OpenGL buffer with an OpenCL buffer
    • allow OpenCL to apply the kernel to the OpenCL buffer (which changes the OpenGL buffer automatically)
    • call “finish” to ensure that all OpenCL operations have completed
  • OpenGL will then draw the new mesh, the next time through the loop
Background (IV): In Code

```java
while (!Display.isCloseRequested()) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(0.0f, 0.0f, planeDistance);
    glDrawElements(GL_TRIANGLE_STRIP, mesh.indexCount, GL_UNSIGNED_SHORT, 0);
    Display.update();

    Util.checkCLError(clEnqueueAcquireGLObjects(queue, vertexBufferCL, null, null));
    kernel.setArg(0, vertexBufferCL);
    clEnqueueNDRangeKernel(queue, kernel, 1, null, workSize, null, null, null, null);
    Util.checkCLError(clEnqueueReleaseGLObjects(queue, vertexBufferCL, null, null));
    clFinish(queue);
}
```

**DEMO**
Ripple (I)

• In the previous example, the z values were initialized to zero
  • And, $0 \times 1.01 = 0$

• As a result, the “zoom” of the previous example was achieved by just spacing out the triangles' x and y values until they were offscreen

• The second example is more interesting, the kernel targets the z value of a triangle’s vertices
  • As a result, it will morph the mesh into “3D shapes”
• And, just to be fancy, the program supports up to 16 ripples at a time!
Ripple (II)

• Each click on the mesh (up to 16), creates a “center point” for a wave that ripples out over the mesh

• Our kernel looks at each vertex and calculates the impact of each ripple on the z coordinate of that vertex, using the following equation and conceptual model

\[ z = A e^{-Dr^2} e^{-W\delta^2} \cos(F\pi\delta) \]

A is the amplitude of the wave; D is the decay of the wave; W is the width of the wave; F is the frequency of the wave.
Ripple (III): The Kernel

Three arrays are passed along with two parameters to each one-dimensional work item.

Our id retrieves the vertex that we’re working on. The centers array contains the center point of each ripple. The times array contains the time each ripple was created. The now variable contains the current time.

With that information, we can use the equation on the previous slide, to update the z value for each ripple.

OpenCL ensures that ALL vertices are updated IN PARALLEL. That’s the true power of this approach.

© Kenneth M. Anderson, 2014
Summary

• We scratched the service of data parallelism and GPU programming with OpenCL

  • We looked at a range of examples of OpenCL kernels
    • an abstract way of defining a “work item”
      • This specification is compiled into code that performs the specified operations on as many data points as possible in parallel
    • We saw the power of this technique by showing how OpenCL can support the transformation of OpenGL objects, the GPU performs most of the calculations, freeing up the CPU to handle other tasks

• This approach stands in contrast to the other concurrency alternatives
  • our programs were all single threaded; instead we used the GPU to perform calculations in parallel when it was needed
Coming Up Next

• FALL BREAK!!
  • Reminder: Theme: Sleeping and Eating
    • Try to do as much of that as possible!