LSU EE 7700-2 **Homework 1** Solution **Due: 25 January 2013**

Problem 1: Read up to, and including, Section II, in the paper "GPU Computing," by Owens *et al.* The paper can be found at `http://www.ece.lsu.edu/gp/srefs/Owens-gpus-2008.pdf`. It is freely accessible within the `lsu.edu` domain, outside you will be prompted for a user name and password. The user name is `ee4720`. The password will be given in class. If you've forgotten the password here's a hint: It's why you shouldn't climb into the cage near Tiger Stadium. The password is all lower case and there are no spaces.

This paper provides a brief history of GPUs and explains how and why they were adopted for scientific computation. The material after Section II is dated and so should not be read.

The following definitions may be helpful in reading the paper: *S*hading:Computing the color of a point on a primitive. The color is computed based on the material properties of the primitive (what one might think of as its color) and also on the location and brightness of light sources, etc. *C*oordinate transformation to screen space: Coordinates refer to the location of the vertices of the primitives (usually triangles). The application programmer specifies these in some convenient coordinate space. The screen coordinate space refers to the x and y location of pixels (unit of screen location), and transformation here refers to the mathematical operation of mapping from user coordinate space to screen space. The operation itself consists of multiplying a transformation matrix, a 4×4 matrix, by the user-space coordinate, written as a four-component column vector (with the fourth component set to 1).

After reading the paper answer the following question[−1].

(*a*) The paper might be difficult to read for first-year graduate students without a strong computer architecture background. Find a term or sentence in the paper that you could not understand. Write down the sentence, which page it's on, and your best guess as to what it means.

Here are some of the questions that were asked, along with the answers. The questions are paraphrased for the sake of clarity.

JA asks: *How is it that a graphics operation can take 1000's of cycles, as stated in the penultimate paragraph on page 881, while a CPU pipeline has a latency of* ≈ 20 *cycles.*

The confusion arises from the vast difference between the sense of the word *pipeline* used to describe hardware in a CPU and the sense used to describe a GPU.

The term *pipeline* when used to describe CPU hardware refers to a set of hardware units, *stages*, that are used to execute a CPU machine instruction. Instructions proceed through the stages in order and at any one time different stages can (and for efficiency should) be occupied by different instructions. The number of stages varies from four or five for simpler designs, to a dozen or more in more advanced designs. Instructions spend only one clock cycle, maybe $\approx 0.5\,\text{ns}$ in current designs, in a stage.

The term *pipeline* when used in the paper to describe a GPU refers to the set of steps needed to process a vertex, this is often called a *rendering pipeline*. The rendering pipeline steps are performed by short programs, in some cases by processors that can be user programmed, called *programmable shaders*, in other cases by hardware with the program fixed in the hardware. The set of steps in the rendering pipeline are defined by standards such as OpenGL (Khronos) and Direct3D (Microsoft).

Given these usages of the term pipeline, contrasting the 20-cycle instruction latency through a CPU pipeline with the 1000's of cycles taken by a vertex until it reaches the frame buffer is an apples-to-oranges (less relevant) comparison.

The paper is trying to point out the fact that there are no dependencies between vertices (until they reach the frame buffer). That means there is no need for the processing of one vertex to wait until a prior vertex finishes. This enables vertex processing to be overlapped or done entirely in parallel, something which GPUs are designed to exploit.

In a CPU one instruction may depend upon another, limiting the amount of overlap. In general purpose CPUs the amount of delay due to dependencies is determined by the latency of the functional unit (say, 1 cycle for an integer operation) which is less than the full pipeline latency (5 to over 12 cycles).

TSR asks: *Why are triangles the only kind of primitive processed by a GPU?*

To limit the complexity of the hardware some compromises need to be made, and for early GPUs the compromise was using 2D primitives. The reason for going even further, limiting primitives to just triangles, is that it is a simple matter for a CPU to *tessellate* a 2D primitive into triangles, whereas the modifications so that a GPU could it would make early GPUs more complex (and overall would yield no performance gain). GPU APIs, such as OpenGL, did once accept arbitrary polygons as primitives, but they would probably tessellate them on the CPU and just send triangles to the GPU.

Note that curved surfaces can be very effectively approximated by triangles, for example, in the classroom demo presented in the beginning of the semester.

Modern GPUs (or GPU APIs such as OpenGL) no longer require triangles as inputs. Instead the CPU can send a collection of vertices that define some kind of surface. User-written GPU code, for *tessellation shaders*, a *geometry shader*, or both, would convert the vertices (which are coordinates in 3D space) into triangles covering a surface (whatever surface the programmer wanted). Past the geometry shader stage of the rendering pipeline GPUs continue to use triangles as primitives.

SI asks: *Asks how it is possible to find the closest fragment to the camera so easily, as mentioned on page 881.*

Each fragment carries a transformed coordinate. The original coordinate was in a coordinate system convenient to the programmer. For example, in a flight simulator program the coordinate system might put the origin at the start of the runway, $+z$ might be north, and $x = 1$ might be one foot (because people in aviation seem stuck on English units). These coordinates are transformed into a space in which the $+z$ direction points away from the camera (or users eye). So, given two fragments one need only compare their z components to determine which is closer. This comparison occurs when its time to write a fragment to the frame buffer. The fragment's x and y components are in units of pixels, and so are used to find an address (more precisely and index) in the frame buffer. In typical use there are at least two buffers, a *color buffer* where the fragment color will be written, and a z *buffer* where the fragment's z component will be written. Before writing anything a fragment reads from the z buffer to see if the fragment that is already there is closer to the camera. If so, the arriving fragment is dropped, of not the color and z value of the arriving fragment are written to the respective buffers.

The fragment operations are computationally demanding *in contrast to vertex operations* because for each set of three vertices (which make a primitive) there can be many, hundreds or more, fragments. (A fragment is a pixel covered by a primitive.)

The rendering pipelines used by GPUs do not use ray casting and do not sort fragments. Ray casting (used to implement) ray tracing requires unstructured memory access patterns that are too demanding. It is still possible to use a GPU for ray tracing by performing ray tracing as a non-graphical computation (using CUDA, OpenCL, or an OpenGL *compute shader*).

The OpenGL rendering pipeline was defined specifically to avoid the need to sort fragments by camera distance. Fragments arrive at the frame buffer in the same order their respective primitives entered the pipeline, and so it is not possible to, for example, compare an arriving fragment to the fragment which is closest to it in camera distance. This makes no difference if fragments are opaque (the closest one completely overwrites the others), but if fragments are partly transparent sorting by distance is better.

RK asks: *What does the paper mean in saying that the CPU divides the pipeline in time while the GPU divides the pipeline in space?*

The pipeline being referred to is the *rendering pipeline*, a set of steps needed to convert primitives (a set of vertices) to values written into the frame buffer. In a CPU each step might be done by a different routine, for example, `transformVertices()`, `rasterize()`, `applyTextures()`, `updateFrameBuffer()`. In an ordinary program, these routines would be called one at a time, and so the pipeline is divided in time. In older GPU designs each of these steps

would be performed by separate pieces of hardware, for example, a *vertex processor* for transforming vertices (which can run programs called *vertex shaders*), and a *fragment processor* replacing the `applyTextures()` routine. Since these were physically separate the pipeline was said to be divided in space.

Modern CPUs and GPUs cannot be categorized so conveniently. A multicore CPU can easily run the different routines on different cores (actually on different threads that the OS would distribute to cores), and in that case they would be divided in space too. Modern GPUs no longer use separate hardware for vertex and fragment processing, and so a GPU might first run the vertex shader and then after that's done use the same hardware for subsequent steps. If so the GPU execution of the rendering pipeline would be divided in time too.

In summary, *divided in time* made sense for old CPUs with one core because with just one core there was no choice but to do the steps at different times. *Divided in pace* made sense for old GPUs with more specialized hardware than now because there was no choice but to do the steps in different places.

NOTE: The questions below **are not** *based on the Owens paper from the previous problem. Answer the questions below using material covered in class.*

Problem 2: LISP machines were developed for use in artificial intelligence (AI) research. Many video games have an AI component. Why do you think AI processing units were not (or would not be) successful?

There is no point in spending vast sums of money developing a specialized processor if it would not be much faster than a CPU. As pointed out in class there was a time in which many types of specialized processors were contemplated and few of these were developed, but they were not successful because their benefit was small and development costs were large. Lisp machines for AI was given in class as an example.

Problem 3: You are in charge of a computing system that computes daily reports. Currently they take 6 hours to compute using eight cores, that is, $t(8) = 6$ hours. The single-core execution time is $t(1) = 32$ hours. The program is written to run on any number of cores.

Your goal is to get the system to compute a result in 4 hours using a budget of $6000. There are two options, buy an additional eight cores for $6000 or hire a programmer at the rate of $20 to $100 per hour (depending on who you hire).

Choose an option and justify your answer. Either answer can be correct. As part of the solution estimate the execution time of the program on a sixteen-core system and the chances that the programmers you hire can improve speedup sufficiently for a 4-hour run.

Note that this is only a fictional assignment, you don't get to keep any left over money.

Before considering whether to buy machines or peoples' time lets look at the numbers. The single-thread (core) time is $t(1) = 32$ hours. With linear (ideal) speedup the 8-core time would be $t_{\text{ideal}}(8) = \frac{32}{8} = 4$ hours. The 8-core time with the existing parallel code is $t_{\text{existing}}(8) = 6$.

To decide whether to hire programmers or buy more cores we need to estimate the run time on a 16-core system. It would be naïve to expect the 16-core time to be $_{\text{naive}}(16) = \frac{t(8)}{2}$ because we did not get linear speedup with 8 cores.

The problem said nothing about the nature of the parallel code so we need to make assumptions. Lets assume the code follows the Amdahl's Law model in which a parallel program has a serial portion and a parallel portion. The serial portion runs on one core while all the other cores are idle (either because they are finished or because they are waiting for the results from the serial portion). The run times on n cores is given by

$$t_{\text{Amdahl}}(n) = ft(1) + (1 - f)\frac{t(1)}{n},$$

where $ft(1)$ is the execution time of the serial portion. The quantity f can be though of as the fraction of $t(1)$ that has not been parallelized, and perhaps can not be parallelized no matter how smart and hard working you are.

If a program fits this model we can solve for f using two execution times, $t(1)$ and $t(n)$ for $n \neq 1$. It can be shown that:

$$f = \frac{n\frac{t(n)}{t(1)} - 1}{n - 1}.$$

For our situation $f = 1/14$ and the serial portion is 2.29 hours. Substituting we get

$$t_{\text{Amdahl}}(32) = 2.29 + \frac{32 - 2.29}{16} = 4.14,$$

which is about $8\frac{1}{2}$ minutes too long for us. This is not the only possible model, but knowing nothing else we'll stick with it.

If this model is correct, then buying another 8 cores won't work, the run time will still be greater than our goal of 4 hours.

The argument for buying 8 more cores would be that the parallel execution time fits some other model. Perhaps there is a $k\frac{\log_2 n}{n}$ component for communication present only when $n > 1$. Then

$$t_{\text{othermodel}}(n) = \begin{cases} 32, & \text{if } n = 1; \\ \frac{32 + k \log_2 n}{n}, & \text{otherwise.} \end{cases}$$

For this model and our data $k = 16/3$ and so $t_{\text{othermodel}}(32) = 3\frac{1}{3}$.

The argument for hiring programmers would be that the original parallelization effort was not performed thoroughly, and perhaps that the serial program from which $t(1)$ was obtained also was not well written. In that case the need to achieve linear speedup (often a difficult goal) is not as scary as it sounds.