In such cases, writes to one image do not affect the contents of any other image. The compiler can therefore be more aggressive about making optimizations that might otherwise be unsafe. Note that by default, the compiler assumes that aliasing of external buffers is possible and is less likely to perform optimizations that may break otherwise well-formed code. (Note GLSL assumes no aliasing of variables and parameters residing within the shader, and fully optimizes based on that.) The `restrict` keyword is used in a similar manner to the `volatile` keyword as described above—that is, it may be added to global or local declarations to effectively add the restrict qualifier to existing image variables in certain scope. In essence, references to memory buffers through `restrict` qualified image variables behave similarly to references to memory through restricted pointers in "C" and C++.

There are three further qualifiers available in GLSL that do not have an equivalent in C. These are `coherent`, `readonly`, and `writeonly`. First, `coherent` is used to control *cache behavior* for images. This type of functionality is generally not exposed by high-level languages. However, as GLSL is designed for writing code that will execute on highly parallel and specialized hardware, `coherent` is included to allow some level of management of where data is placed.

Consider a typical graphics processing unit (GPU). It is made up of hundreds or potentially thousands of separate processors grouped into blocks. Different models of otherwise similar GPUs may contain different numbers of these blocks depending on their power and performance targets. Now, such GPUs will normally include large, multilevel caches that may or may not be fully coherent.[6] If the data store for an image is placed in a noncoherent cache, then changes made by one client of that cache may not be noticed by another client until that cache is explicitly flushed back to a lower level in a memory hierarchy. A schematic of this is shown in Figure 11.5, which depicts the memory hierarchy of a fictitious GPU with a multilevel cache hierarchy.

6. A coherent cache is a cache that allows local changes to be immediately observed by other clients of the same memory subsystem. Caches in CPUs tend to be coherent (a write performed by one CPU core is seen immediately by other CPU cores), whereas caches in GPUs may or may not be coherent.

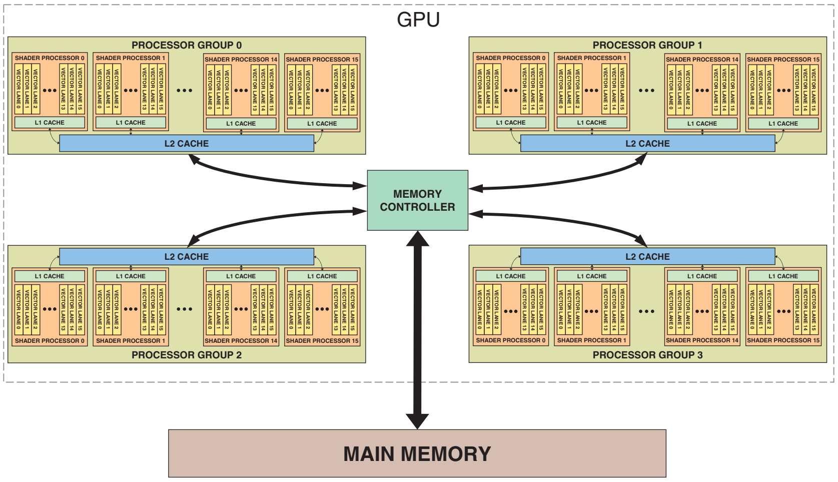

Figure 11.5 Cache hierarchy of a fictitious GPU

In Figure 11.5, each shader processor is made up of a 16-wide vector processor that concurrently processes 16 data items (these may be fragments, vertices, patches, or primitives depending on what type of shader is executing). Each vector processor has its own, small, level-1 cache, which is coherent among all of the shader invocations running in that processor. That is, a write performed by one invocation on that processor will be observed by and its data made available to any other invocation executing on the same processor. Furthermore, there are four shader processor groups, each with 16, 16-element-wide vector processors and a single, shared, level-2 cache, that is, there is a level-2 cache per shader processor group that is shared by 16, 16-wide vector processors (256 data items). There are therefore four independent level-2 caches, each serving 16 processors with 16-wide vectors for a total of 1024 concurrently processing data items. Each of the level-2 caches is a client of the memory controller.

For highest performance, the GPU will attempt to keep data in the highest-level cache, that is, in caches labeled with the smallest number, closest to the processor accessing the data. If data is only to be read from memory, but not written, then data can be stored in noncoherent caches. In such cases, our fictitious GPU will place data in the level-1 caches within the vector processors. However, if memory writes made by one processor must be seen by another processor, (this includes atomics that implicitly read, modify, and write data), the data must be placed in a coherent memory location. Here, we have two choices: the first, to bypass cache altogether, and the second, to bypass level-1 caches and place data in level-2 caches while ensuring that any work that needs to share data is run only in that cache's shader processor group. Other GPUs may have ways of keeping the level-2 caches coherent. This type of decision is generally made by the OpenGL driver, but a requirement to do so is given in the shader by using the **coherent** keyword. An example **coherent** declaration is shown in Example 11.18.

Example 11.18 Examples of Using the **coherent** Keyword

```
#version 420 core

// Declaration of image uniform that is coherent. The OpenGL
// implementation will ensure that the data for the image is
// placed in caches that are coherent, or perhaps used an uncached
// location for data storage.
layout (r32ui} uniform coherent uimageBuffer my_image;

// Declaration of function that does declares its parameter
// as coherent...
uint functionTakingCoherentImage(coherent uimageBuffer i, int n)
{
    // Write i here...
    imageStore(my_image, n, uint(n));

    // Any changes will be visible to all other shader invocations.
    // Likewise, changes made by invocations are visible here.
    uint m = imageStore(my_image, n - 1).x;

    return m;
}
```

The final two image qualifier keywords, **readonly** and **writeonly**, control access to image data. **readonly** behaves somewhat like **const**, being a contract between the programmer and the OpenGL implementation that the programmer will not access a **readonly** image for writing. The difference between **const** and **readonly** applied to an image variable is that **const** applies to the variable itself. That is, an image variable declared as **const** may not be written, however, the shader may write to the image bound to the image unit referenced by that variable. On the other hand, **readonly** applies to the underlying image data. A shader may assign new values to an image variable declared as **readonly**, but it may not write to an image through that variable. An image variable may be declared both **const** and **readonly** at the
same time.

The **writeonly** keyword also applies to the image data attached to the image unit to which an image variable refers. Attempting to read from an