of the X-ray detector. There is a one-to-one correspondence between detector regions and projection data; thus, roughly 3 million threads are created for each of the 15 forward projections. For GPUs with smaller amounts of memory (less than 2 GB), the data for the entire set of projections cannot fit into the GPU memory at once, so segmenting of the computation is required — this is naturally accomplished by computing each of the 15 projections one at a time and swapping them off of the GPU after computation is complete. This is not an issue for GPUs with larger memory capacities.

During the backward projection phase we map threads to image voxels, where a single thread computes the contributions of all X-ray source positions to the attenuation coefficient of a voxel. For the backward phase, creating one thread per voxel has a number of advantages over the detector-based mapping used in forward projection. If the detector-based mapping were used in the backward phase, each thread would compute a portion of the attenuation coefficient for all voxels it encountered on its path between source and detector. Doing so would require the use of atomic updates, which degrade performance, or require multiple 3-D images to be kept (one per projection) that would have to be merged to obtain the final image. Merging the data requires both extra calculation and larger data transfers. Further, because multiple rays from the same projection can impact a voxel, synchronization would still be required to ensure coherent updates. Using a voxel-based mapping avoids these limitations. Using the voxel-based approach also allows segmenting of the 3-D image into 2-D slices for GPUs with smaller memories and for multi-GPU execution.

Other data segmentation techniques are indeed possible (as shown in [8] for distributed environments), though the most effective require dynamically determining the boundaries of segments at runtime based on the geometry of the X-ray sources, or otherwise, suffer from blurred reconstruction at segment boundaries. Instead, the mappings presented here have proven to be very effective when targeting GPU and multi-GPU environments.

40.3.3 Optimizing with Texture Memory

When data on the GPU is accessed through the texture unit, additional data with spatial locality are automatically cached on chip in the texture cache. If cached data are accessed by a thread from the same SM, it can be retrieved from the cache without putting any pressure on the global memory bus.

In the backward projection algorithm used by DBT, when the path from the source to the detector cell falls between multiple vertices of the Cartesian grid, data from each vertex contribute to the calculation (as shown in Figure 40.4). Depending on the proximity to each vertex, the values retrieved

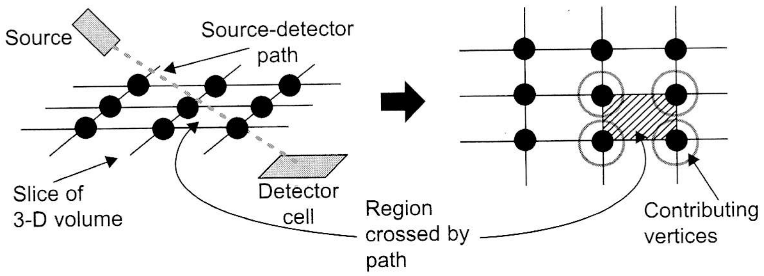


FIGURE 40.4

Calculating the intensity of X-rays from source to detector requires considering multiple voxel values in each cross-sectional slice of the image.

are given a weight to determine their influence on the final value. Because the weights of the values vary per thread and are determined live, we cannot take advantage of hardware interpolation that would otherwise reduce the number of accesses. However, if threads with spatial locality are grouped on the same SM, then multiple threads will end up accessing data from the same region, and using texture memory will serve to decrease pressure on the global memory bus.

Another benefit of texture memory is that boundary conditions can be handled automatically by hardware. For example, if an out-of-bounds index is accessed through the texture unit, the unit can be programmed to return the edge-pixel value, to return zero (using the CUDA C construct called *surfaces*), or to cause the kernel to fail. Aside from the obvious reduction of code complexity, removing conditional statements is generally an important optimization on current GPU SIMD hardware because conditionals cause thread divergence and prevent the processing units from performing meaningful work. Algorithms 1 and 2 show the code for bounds checking in DBT when global memory is used and the removal of conditionals when texture memory is used, respectively.

Algorithm 1: Snippet of code from a kernel accessing data in global memory.

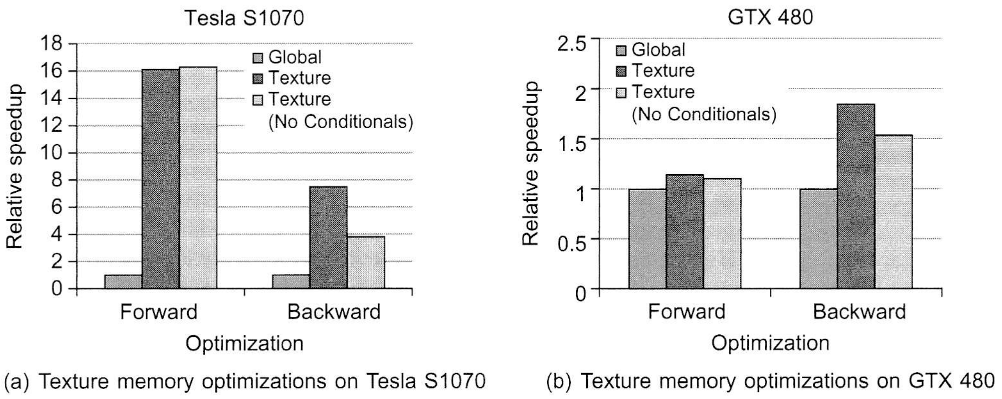
```
// When accessing global memory, bounds must be checked before data can be
   retrieved.
// The pixels are weighted using the values w1, w2, w3, and w4,
   respectively.
if $x1 \geq 0$ AND $x2 < imageWidth$ AND $y1 \geq 0$ AND $y2 < imageHeight$ then
    $tmp = image(x1,y1) * w1 + image(x1,y2) * w2 + image(x2,y1) * w3 + image(x2,y2) * w4$
else if $x1 \geq 0$ AND $x2 < imageWidth$ AND $y2 == imageHeight$ then
    $tmp = image(x1,y1) * w1 + image(x1,y2) * w2$
else if $x1 \geq 0$ AND $x2 < imageWidth$ AND $y2 == 0$ then
    $tmp = image(x2,y1) * w3 + image(x2,y2) * w4$
else if $x2 == imageWidth$ AND $y1 \geq 0$ AND $y2 < imageHeight$ then
    $tmp = image(x1,y1) * w1 + image(x2,y1) * w3$
else if $x2 == 0$ AND $y1 \geq 0$ AND $y2 < imageHeight$ then
    $tmp = image(x1,y2) * w2 + image(x2,y2) * w4$
end if
```

In the case of DBT, using texture memory provided a very large performance gain on a CUDA architecture of compute capability 1.3 (Tesla S1070), and a significant (although much smaller) benefit on a Fermi architecture of compute capability 2.0 (GTX 480). The GTX 480 memory system features automatic caching for global memory that stores accessed data in a shared L2 cache and also in an on-chip, low-latency L1 cache. Because the automatic caching takes advantage of some spatial locality, it has an effect similar to the texture cache, and we see less benefit from using the texture unit on the GTX 480 than on the Tesla S1070 architecture.

Using the texture unit to handle boundary conditions, we were able to remove many of the conditional statements for the DBT kernel. Interestingly, removing these conditional statements actually showed a slight decrease in performance compared with the texture approach with conditionals in place. The reason for the decrease in performance is the memory-bound execution profile of the DBT application. Relative to the texture cache access time, the time to compute conditional outcomes is small

Algorithm 2: Snippet of code from a kernel accessing data in texture memory. The use of texture memory allows removal of bounds-checking code.

```
// When accessing texture memory, out-of-bounds indexes can be set to
   return 0
// using CUDA surfaces and the cudaBoundaryModeZero parameter.
// The pixels are weighted using the values w1, w2, w3, and w4,
   respectively.
$val1 = surf2Dread(image, x1, y1, cudaBoundaryModeZero) * w1$
$val2 = surf2Dread(image, x1, y2, cudaBoundaryModeZero) * w2$
$val3 = surf2Dread(image, x2, y1, cudaBoundaryModeZero) * w3$
$val4 = surf2Dread(image, x2, y2, cudaBoundaryModeZero) * w4$
$tmp = val1 + val2 + val3 + val4;$
```



(a) Texture memory optimizations on Tesla S1070 (b) Texture memory optimizations on GTX 480

FIGURE 40.5

A performance comparison across optimizations and hardware. The baseline algorithm (*Global*) reads data from global memory, the *Texture* optimization reads data from texture memory, and *Texture* (*No Conditionals*) uses the bounds-checking feature of the texture unit to remove conditional statements from the algorithm.

because the texture cache is not a low-latency cache. Leaving the conditionals in place could possibly save up to two texture accesses for many threads, and although accesses to texture cache decreases off-chip memory traffic, the time to access the cache is still on the order of an off-chip memory access. Figure 40.5 shows the performance benefits for the algorithm before and after mapping the forward and backward projections to texture memory on the S1070 and GTX 480 GPUs.

40.3.4 Acceleration with Multiple GPUs

Computing with multiple GPUs allows the computation to be split based on the properties of the thread mappings discussed previously, in which projections and image slices are treated as units (i.e., in forward projection, projections are split between GPUs, and in backward projection, slices of image voxels are split between GPUs). Figure 40.6 shows the distribution of slices on up to eight GPUs. If all GPUs