# 4
# Parallel Grid Construction

In our implementation, we have chosen the grid data layout as described in Section 3.1. When using the GPU, the main challenge in this construction procedure is to develop an algorithm that is capable of writing into several primitive lists, all arranged contiguously in memory, using hundreds of concurrent threads. Solving this data-parallel problem means to avoid concurrent writes to the same memory position. For instance, if more than one thread is about to write to the same buffer, it is necessary to serialize write operations, slowing down overall performance.

Another question is how to avoid consecutive buffers to overlap one another. One option is to build each buffer in sequence, so that the end of the previous list is known before starting to write the current one. However, buffer sizes tend to be in the order of tens of primitives. This would restrict the amount of parallelism that could be achieved. Another more efficient approach is to compute all buffer sizes beforehand, thereby reserving sufficient memory space for each one.

To solve these questions, we need to devise a workload distribution for the graphics hardware. Consider the following two different models for this algorithm:

1. If each thread represents a single cell, each primitive list can be maintained separately and there is no potential concurrency hazards.

2. If each thread represents a single primitive, it may be necessary to serialize the writing of each list to avoid losing data.

In order to implement the first algorithm, each thread (cell) must access all primitives in the scene to determine which it contains. In practical terms, the graphics hardware would have to able to support thousands of threads all performing memory read operations of hundreds of thousands of triangles at the same time. Experiments have shown that even using local per-processor caches and memory access coherency, it is impossible to obtain good performance in state of the art hardware.

Likewise, the current GPU architecture restricts the second modeling. Concurrent writes must be serialized because each thread running in parallel would overlap writes to the same cell buffer position, causing loss of data. However, if the cell size could be chosen in order to avoid a very large number of primitives overlapping the same cell, the overhead for writing synchronization could be minimized.

Unfortunately, typical 3D scenes have unevenly distributed triangles of different sizes. Moreover, since the Uniform Grid is a regular spatial subdivision, it is impossible to guarantee that all cells would contain a similar number of primitives. Thus, in practice, the synchronization overhead for concurrent writes still remains prohibitive.

## 4.1 Conceptual Algorithm

We therefore propose a multi-pass algorithm to overcome the above challenges. The main idea is to combine strengths from both approaches mentioned, while at same time minimizing their limitations. For instance, it is possible to maintain separate buffers, avoiding write serialization, while at the same time assigning each primitive to a separate thread.

As mentioned in Section 3.1, the output of the grid construction procedure is the actual grid data, made of a set of primitive lists, accompanied by an index to translate a given cell ID into a buffer start position.

Let us first devise a solution for building the actual grid data. Suppose each thread computes all cells a single primitive overlaps, and writes these values as contiguous lists of (cell ID, primitive ID) pairs. This avoids concurrent write operations but means that cell-primitive pairs are initially sorted according to their primitive IDs.

The desired layout, on the other hand, requires all pairs that have the same cell ID to be arranged contiguously. Our proposed solution is to perform a key-value sorting operation to change the order of these primitive lists, as can be seen in Figure 4.1. As it turns out, this procedure can be efficiently performed in parallel, inside the GPU.

Since in this algorithm all threads would perform simultaneous writes to different positions, it is necessary to avoid overlapping primitive lists. We will adopt the strategy of computing buffer sizes beforehand, reserving sufficient memory space for each one. Observe that the start address of buffer N is the sum of all buffer sizes from 0 to N-1. Given this reasoning, we use the following parallel implementation: each thread (primitive) simply counts the number of cells it overlaps. Afterwards, a parallel-prefix-sum procedure, known as *scan*,

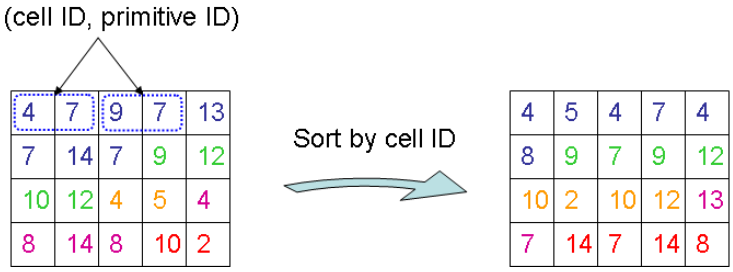



Figure 4.1: Cell-primitive pairs on the left are sorted by primitive ID. The desired result, on the right, with pairs sorted by cell ID. Colors highlight elements which belong to the same list.

is used to accumulate all values and generate the buffer start indices.

Finally, all that remains is to build the index to translate a given cell ID into a buffer start position. It is possible to use a similar solution as the previous one: compute the number of primitives contained by each cell, in parallel, and then accumulate these values. If each thread performs the work of a single primitive, concurrent threads are required to serialize increments to the same cell counter. The other option would be to use each thread to perform the work of a single cell, reading all primitive data and incrementing separate counters. However, both these formulations are not efficient in state-of-the-art GPUs.

There is another, faster approach. Observe that, after the grid sort operation, cell-primitive pairs are arranged in increasing cell ID order. It is therefore possible to use a binary search for a given cell ID, finding the list of primitives it contains. All that remains is to perform a linear search for the buffer start index (to the left) and the buffer end index (to the right). To avoid performing this operation during ray traversal, we will pre-compute buffer start indices as well as their sizes beforehand.

Gathering all the previous ideas, we can summarize the multi-pass algorithm described. Figure 4.2 further details the data generated by each consecutive step.

1. For each primitive, count the number of cells it overlaps.
2. Accumulate the values in Step 1 to compute buffer start indices. These are used in Step 3 to write several buffers in parallel.
3. Write all cells each primitive overlaps using pairs (cell ID, primitive ID).
4. Sort the pairs in Step 3 according to their cell IDs.
5. For each cell ID, perform a binary search in the sorted grid data to find buffer start indices and their sizes.

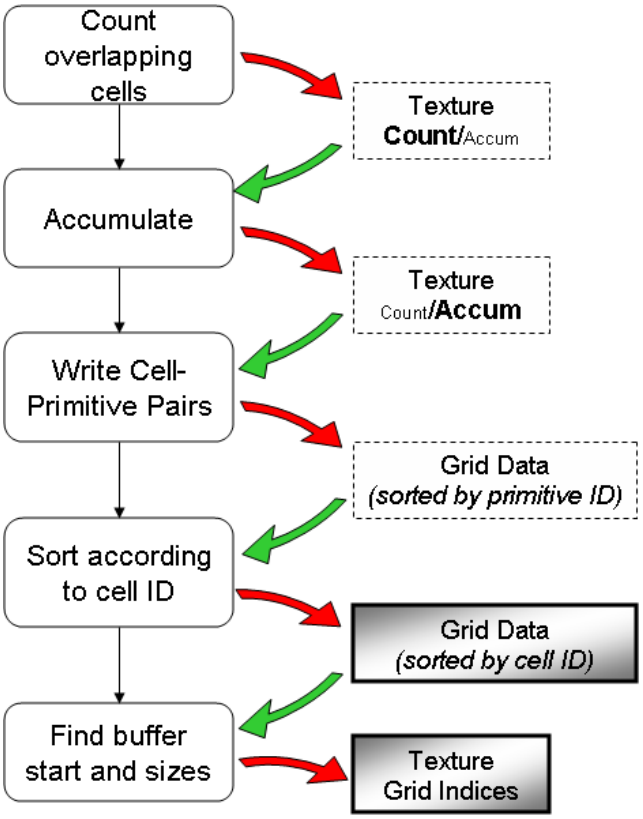



Figure 4.2: Data flow of the proposed grid rebuild algorithm. Red and green arrows indicate write and read operations, respectively. Grey boxes highlight the main output of the procedure.

## 4.2 Proposed Implementation

The algorithm for rebuilding the Uniform Grid is implemented entirely on the GPU. Its main input is a list of vertices that makeup the triangles in the scene. These vertices are stored in a texture, which is accessed during grid construction and ray tracing.

The total number of triangles in the scene determines the grid resolution. We use the equations mentioned in Section 3.1. After the grid resolution is chosen, we use a multi-pass construction algorithm based on the discussion in the previous section.

As highlighted in Chapter 3, we have combined both GLSL and CUDA implementations where each has performed best. Specifically, the parallel-prefix-sum in Step 2 and the key-value sort operation in Step 4 are performed by CUDA procedures, while the other steps can be done efficiently by a single fragment shader each.

In order to transfer data between OpenGL and CUDA procedures, it is necessary to use two different pixel buffer objects: one for data input and

another for the output. Likewise, to store the results of GLSL computations, it is required to use a frame buffer object in order to write the output values into different textures. The GLSL shaders are triggered by drawing a quadrilateral. The 2D coordinates of each fragment are used to access the corresponding input textures in each rendering pass.

The following subsections describe the implementation details of each grid construction step.

### 4.2.1 Counting the Number of Cells Each Primitive Overlaps

The first step in the proposed algorithm can be done efficiently using a single fragment shader. We choose a screen size so that the number of fragments drawn equals the number of primitives in the scene. Therefore, the procedure converts the 2D fragment coordinates into a primitive ID that is used to access the corresponding triangle information.

Afterwards, the shader computes the triangle's axis-aligned bounding box (AABB) and counts how many grid cells this box overlaps. This value is then written in the framebuffer. An alternative method would use a more precise, but computationally expensive, triangle-box overlap algorithm [Möller 2001]. This option, however, has performed poorly during our tests.

### 4.2.2 Computing Primitive Buffer Indices

The output values of the previous step are then copied to a pixel buffer object and processed by a scan procedure implemented in the CUDA Data Parallel Primitives Library or CUDPP [Sengupta et al. 2007]. The resulting accumulated values are stored in a second PBO, which is then read back to an OpenGL texture. This texture now contains, given a primitive ID, the index where its buffer starts in the final grid texture. We store an additional accumulated value at the end, which represents where the last primitive list terminates. This number also represents the required size of the final grid texture.

### 4.2.3 Writing Cell-Primitive Pairs

The third step is to write the (cell ID, primitive ID) pairs using the indices computed in Step 2. Observe that this would require each thread (primitive) to perform several write operations in a single pass. It is possible to implement this algorithm either using CUDA or GLSL.

In order to achieve best performance using the CUDA programming model, it is necessary to arrange sequential threads to perform sequential write operations. This allows the hardware to coalesce a number of small write requests into one large memory operation, amortizing memory latency across several processors. Clearly, this performance prerequisite cannot be achieved in the current step.

Our alternative then is to use GLSL shaders instead. To write several values per primitive, there are two options: using a vertex shader or a geometry shader.

In recent years, a new programmable stage in the graphics pipeline has been introduced, called the *geometry shader*. One of its features is the possibility of sending one primitive to the graphics card and outputting several ones. In practice this allows for drawing a single vertex, representing a scene triangle, and writing several values to different framebuffer positions. This way, writing cell-primitive pairs using geometry shaders is rather straightforward.

Still, experiments have shown that using geometry shaders achieves poor performance. Further investigation lead to an actual limitation of the graphics hardware. Modern GPUs implement this new feature given the constraints of typical graphics applications. For example, a state-of-the-art GPU is not capable of writing more than 1024 scalar values from a single geometry shader. This limits vertex output, in its simplest form, to about 340 vertices. Moreover, outputting more than a dozen primitives per shader results in severe performance degradation.

The alternative is to use multiple draw calls combined with a single vertex shader. Before each draw, a uniform variable is set to tell each vertex shader the current draw index. If this value is greater than or equal to the size of the current primitive buffer, it means that the buffer is already full and the shader must discard the vertex. In the other case, the shader writes the pair (cell ID, primitive ID) at this offset in its corresponding buffer.

However, the main disadvantage of this technique is that the total number of draw commands required equals the largest primitive buffer size. In other words, it is the maximum number of cells a single triangle overlaps. This means that if the scene contains even one large triangle, it would severely slow down the entire grid construction.

All previous approaches have the same paradigm of trying to use each thread to perform several write operations. Let us think backwards then. Given sufficient memory space to write all key-value pairs, how can we determine which pair must be written in each buffer position? In other words, given an index in the resulting grid we need to obtain which cell ID and which primitive

ID need to be written.

As it turns out, we can use the accumulated values computed in Step 2 to determine both these IDs. Recall that these values represent the start index of each primitive list, while an additional accumulated value at the end represents the size of the resulting grid. It is safe to say, then, that all memory positions in the resulting grid are contained by a lower and an upper bound in this accumulated texture.

Note that the lower bound value represents the start of the primitive list where any given cell-primitive pair belongs. In other words, all memory positions in the resulting grid with the same lower bound in the accumulated texture belong to the same primitive list. Therefore, they have the same primitive ID. This ID is simply the memory index of the lower bound value in the accumulated texture.

The cell ID value, on the other hand, can be computed by using a local offset: the difference between the memory position in the resulting grid and the start position of the primitive list (the lower bound value). This local offset indicates the *nth* overlapped cell ID must be written in the *nth* position in the primitive list.

Since the accumulated values are, by definition, sorted in increasing order, we can perform a modified binary search to find the required lower bound for any memory position in the resulting grid. This operation can be performed by a simple fragment shader. The 2D fragment coordinates are converted to a linear index in the resulting grid. We use the last accumulated value to determine the required screen size and thereby avoiding wasting computations. Each shader writes its corresponding (cell ID, primitive ID) pair.

The output of the current step is a texture which contains the cell-primitive pairs sorted by primitive ID.

### 4.2.4 Sorting the Grid Data

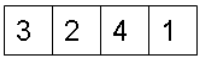
After that, a sort operation is performed to arrange the pairs from Step 3 according to their cell IDs. As usual, the input texture is read to a PBO in order to be processed by the CUDA procedure. We use a modified key-value *radix-sort*. Afterwards, the result is read back from another PBO to the same grid texture. It now contains the final constructed Uniform Grid. All that remains is to determine how to access this data structure.

### 4.2.5
### Compute Indices to Access Grid Data

In Step 5, each thread performs a binary search using the cell ID to find its corresponding primitive list. Afterwards, a linear search is used to compute the buffer start index (to the left) and the buffer end index (to the right). All these operations can be performed by a single fragment shader. We setup the screen size to contain the number of cells in the resulting grid. Therefore, the 2D fragment coordinates are converted to a linear index that corresponds to a given cell ID. The shader writes both the buffer start index and its size in the same pixel.

## 4.3
## Summary

Finally, we can summarize our proposed technique after considering all implementation-specific optimizations mentioned in the previous section. The following Figure 4.3 illustrates an example of our reconstruction procedure. Afterwards, we present an overview of each step during grid construction.

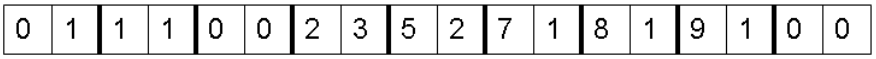



4.3(a): Output from Step 1: the number of cells each of the 4 primitives overlaps.


4.3(b): Step 2 accumulates the values from Step 1, indicating where each primitive list begins. Notice an additional accumulated value at the end, indicating the required grid size.


4.3(c): The (cell ID, primitive ID) pairs from Step 3, initially sorted by primitive ID.


4.3(d): Step 4 output: pairs are sorted by cell ID.




4.3(e): Results from step 5: the cell start index and the number of primitives it contains.

Figure 4.3: Example of our grid construction algorithm using 4 primitives and 9 cells. Colors indicate elements that belong to the same list.

**Phase 1:** Initialization

- **1.1** Given scene AABB and triangle count, compute grid resolution
- **1.2** Setup GLSL: create textures and fragment shaders
- **1.3** Setup CUDA: create scan configurations and PBOs

**Phase 2:** Compute indices to write grid data

- **2.1** **[GLSL]** For each primitive, count the number of cells it overlaps
- **2.2** **[CUDA]** Accumulate the values in Step 2.1 to compute buffer start indices

**Phase 3:** Write the grid data

- **3.1** **[GLSL]** For each position in the resulting grid, use a binary search for its lower bound in the accumulated values from Step 2.2, and determine the (cell ID, primitive ID) values to be written
- **3.2** **[CUDA]** Sort the pairs from Step 3.1 according to their cell IDs

**Phase 4:** Compute indices to access grid data during ray traversal

- **4.1** **[GLSL]** For each cell, use a binary search to find its corresponding primitive list in the sorted grid from Step 3.2, and determine the (buffer start, buffer size) values to be written